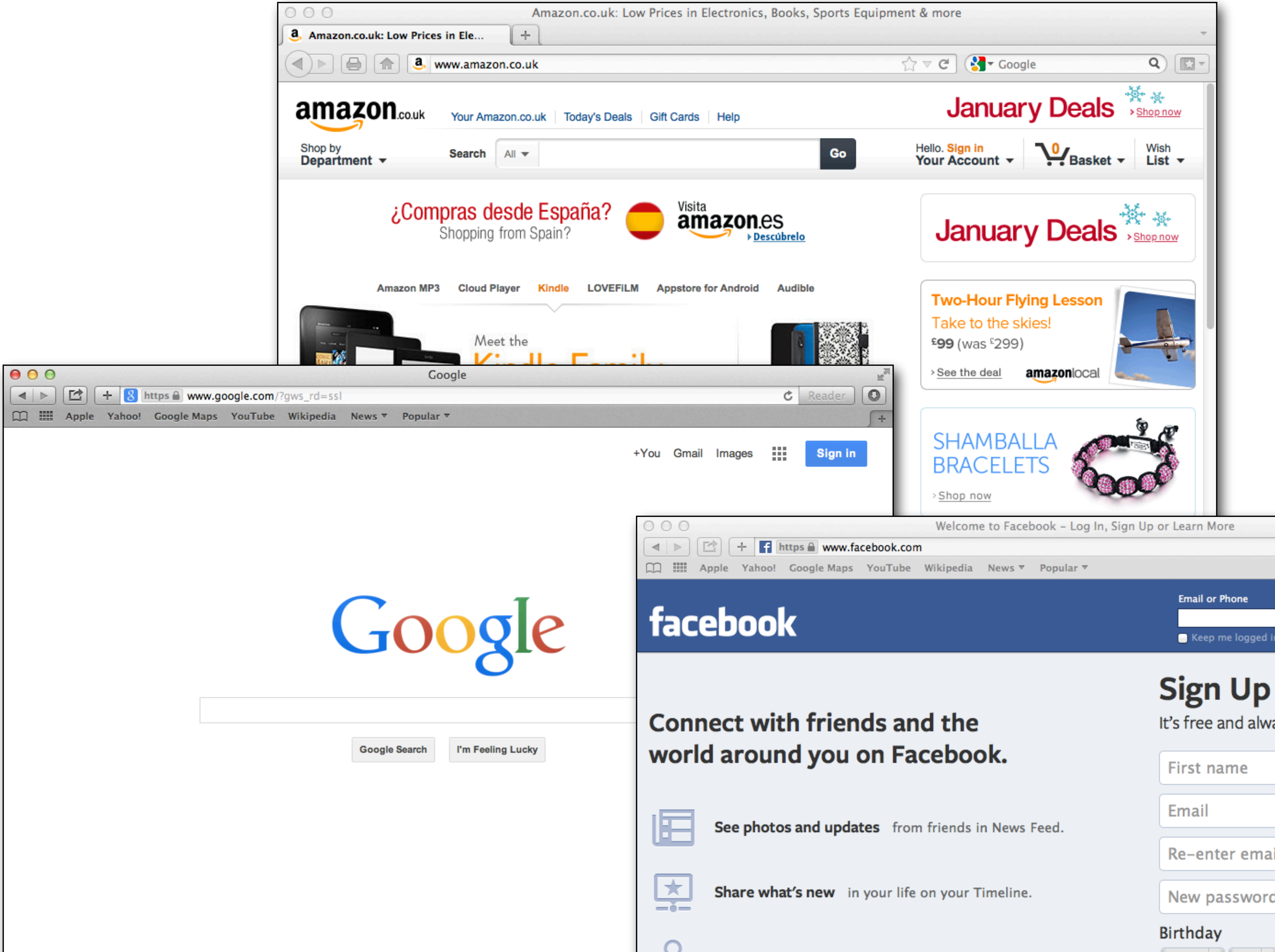
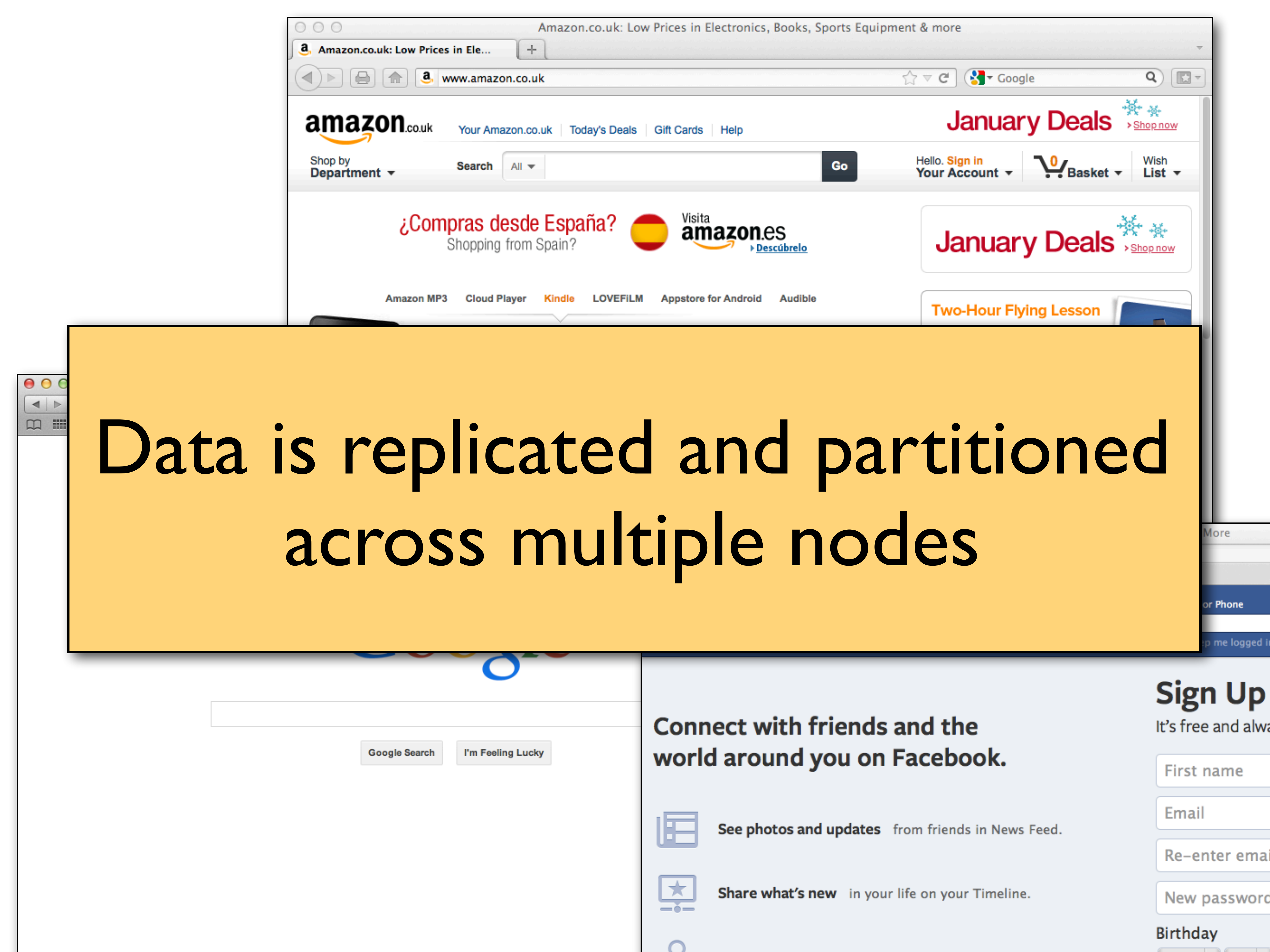
Reasoning about data consistency in distributed systems

Alexey Gotsman

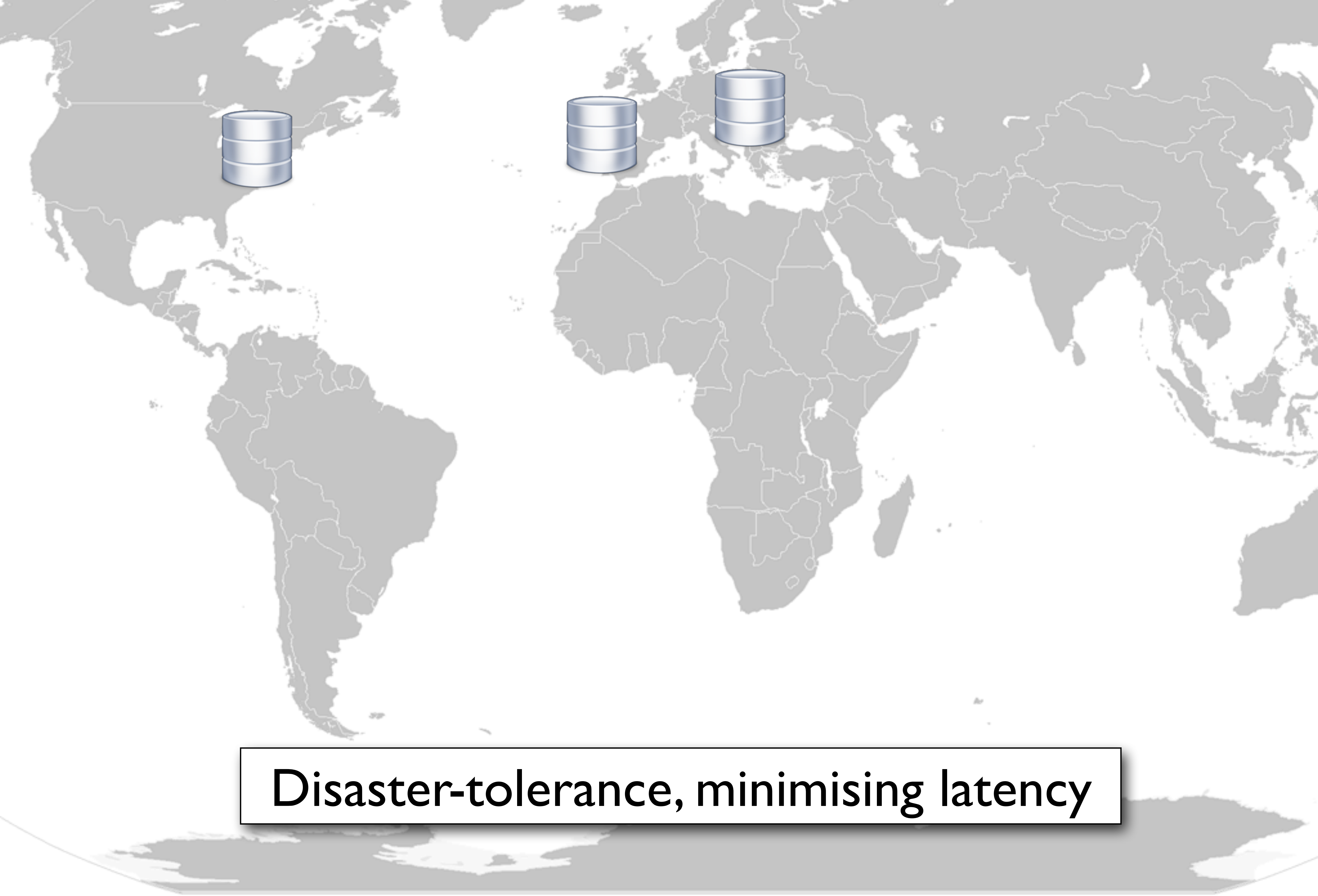
IMDEA Software Institute, Madrid, Spain


Amazon.co.uk: Low Prices in Electronics, Books, Sports Equipment & more
Amazon.co.uk: Low Prices in Ele...
www.amazon.co.uk
Google
amazon.co.uk
Your Amazon.co.uk
Today's Deals
Gift Cards
Help
January Deals
Shop now
Shop by Department
Search
All
Go
Hello. Sign in Your Account
0 Basket
Wish List
¿Compras desde España?
Shopping from Spain?
Visita amazon.es
Descúbrelo
January Deals
Shop now
Amazon MP3
Cloud Player
Kindle
LOVEFiLM
Appstore for Android
Audible
Meet the
Two-Hour Flying Lesson
Take to the skies!
£99 (was £299)
See the deal
amazonlocal
SHAMBALLA BRACELETS
Shop now
Google
https www.google.com/?gws_rd=ssl
Reader
Apple
Yahoo!
Google Maps
YouTube
Wikipedia
News
Popular
+You
Gmail
Images
Sign in
Google
Google Search
I'm Feeling Lucky
Welcome to Facebook – Log In, Sign Up or Learn More
https www.facebook.com
Apple
Yahoo!
Google Maps
YouTube
Wikipedia
News
Popular
facebook
Email or Phone
Sign Up
Connect with friends and the world around you on Facebook.
See photos and updates from friends in News Feed.
Share what's new in your life on your Timeline.
First name
Email
New password
Birthday

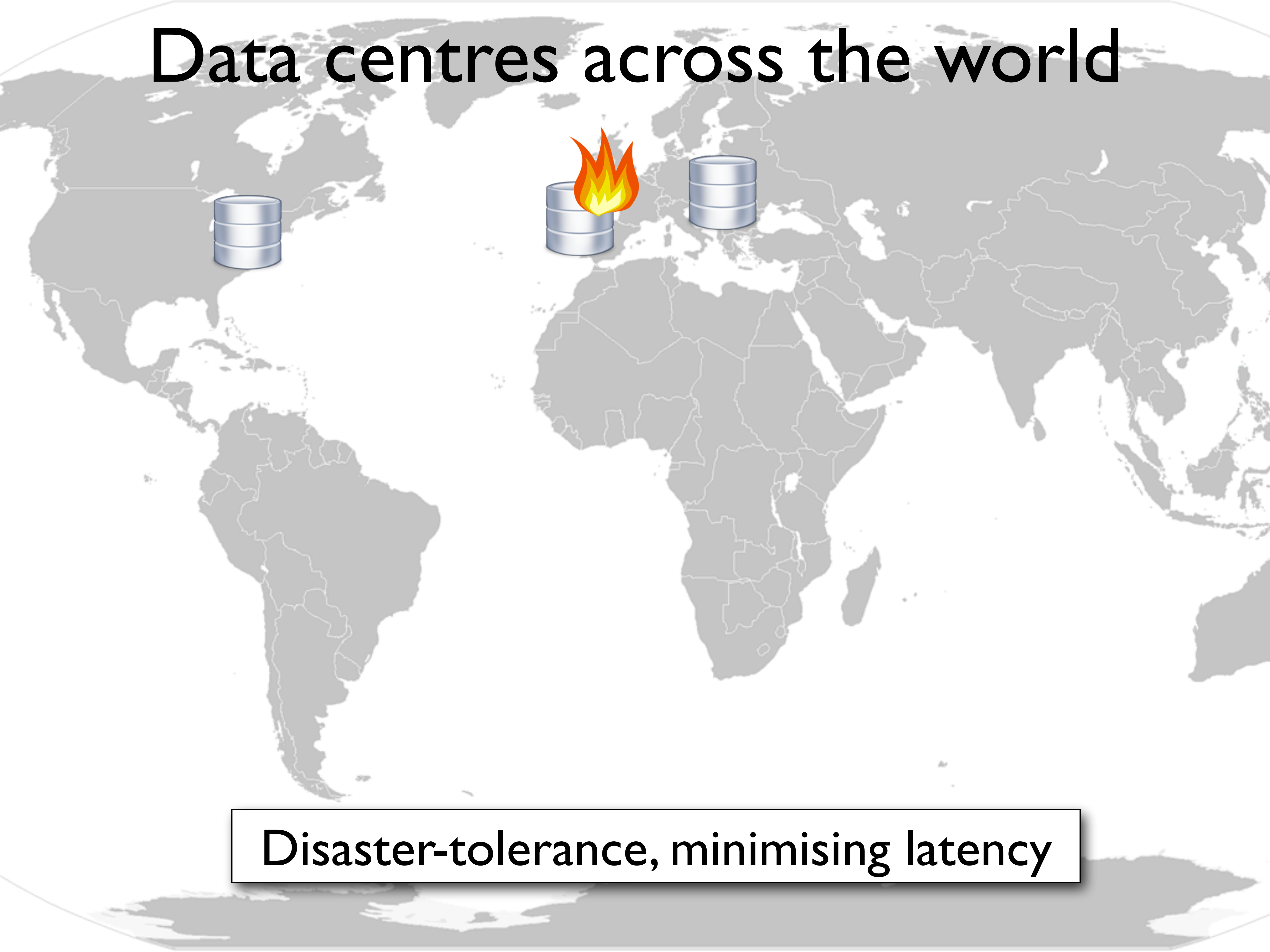


Data is replicated and partitioned across multiple nodes

Data centres across the world

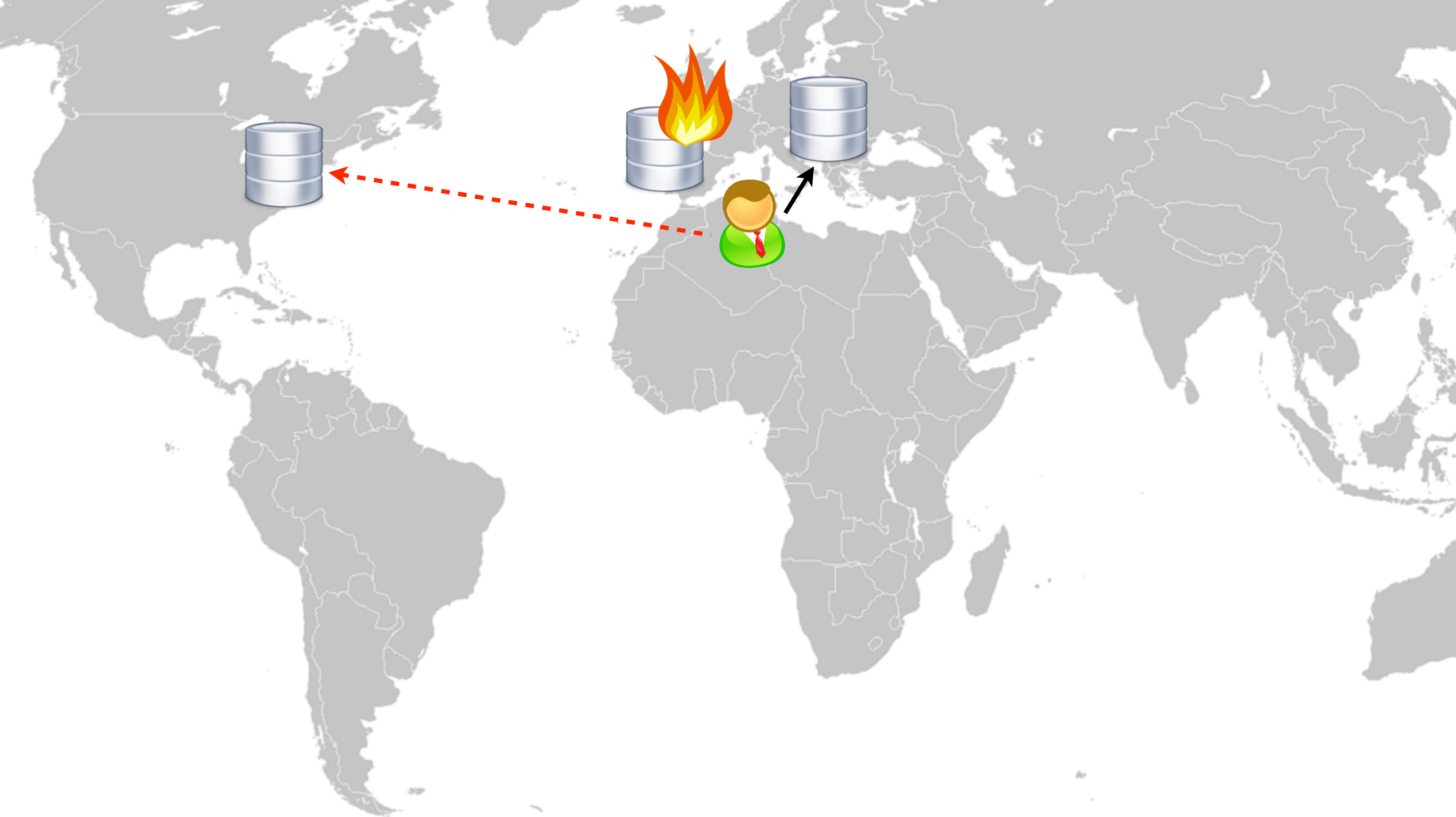


Data centres across the world



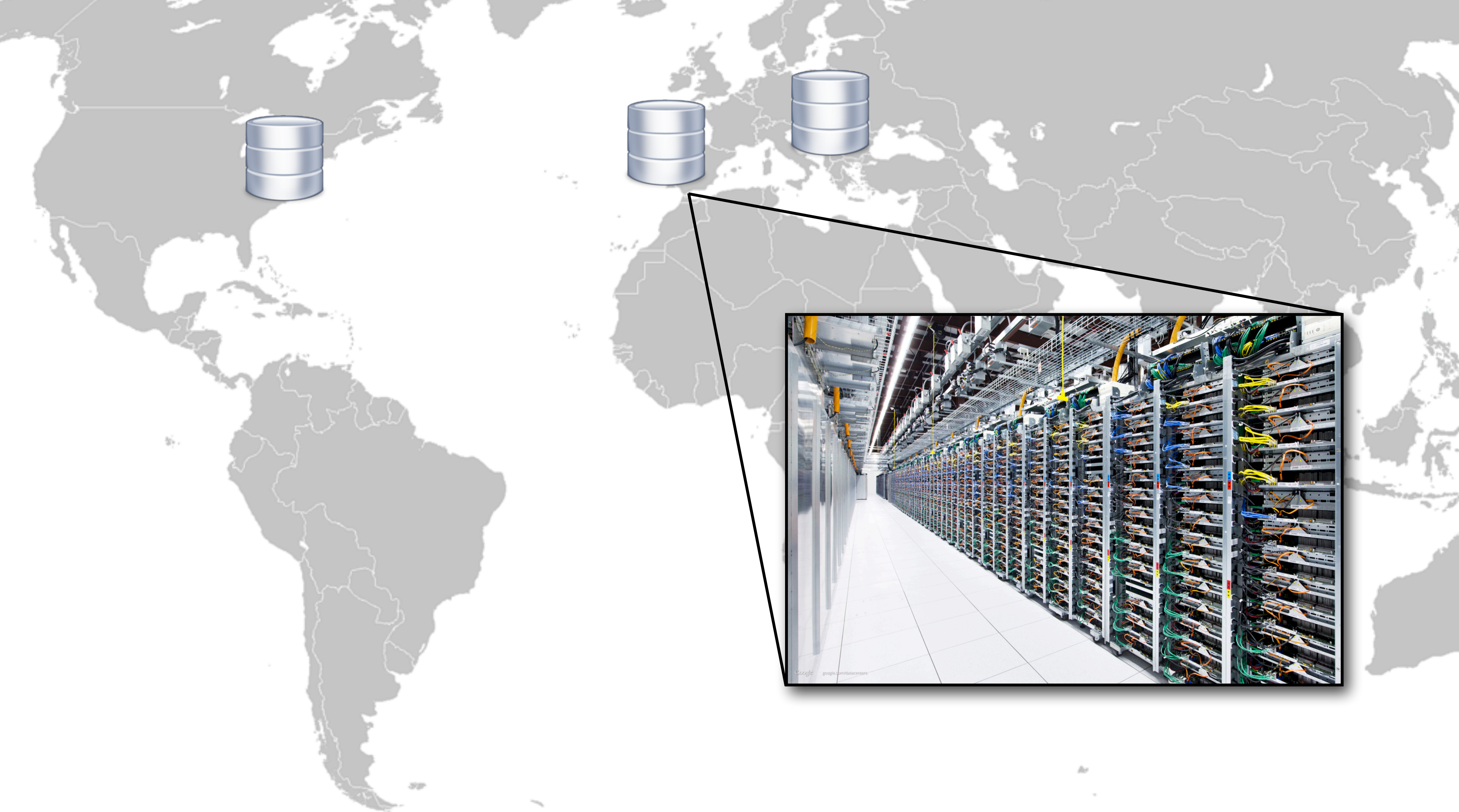
Disaster-tolerance, minimising latency

Data centres across the world



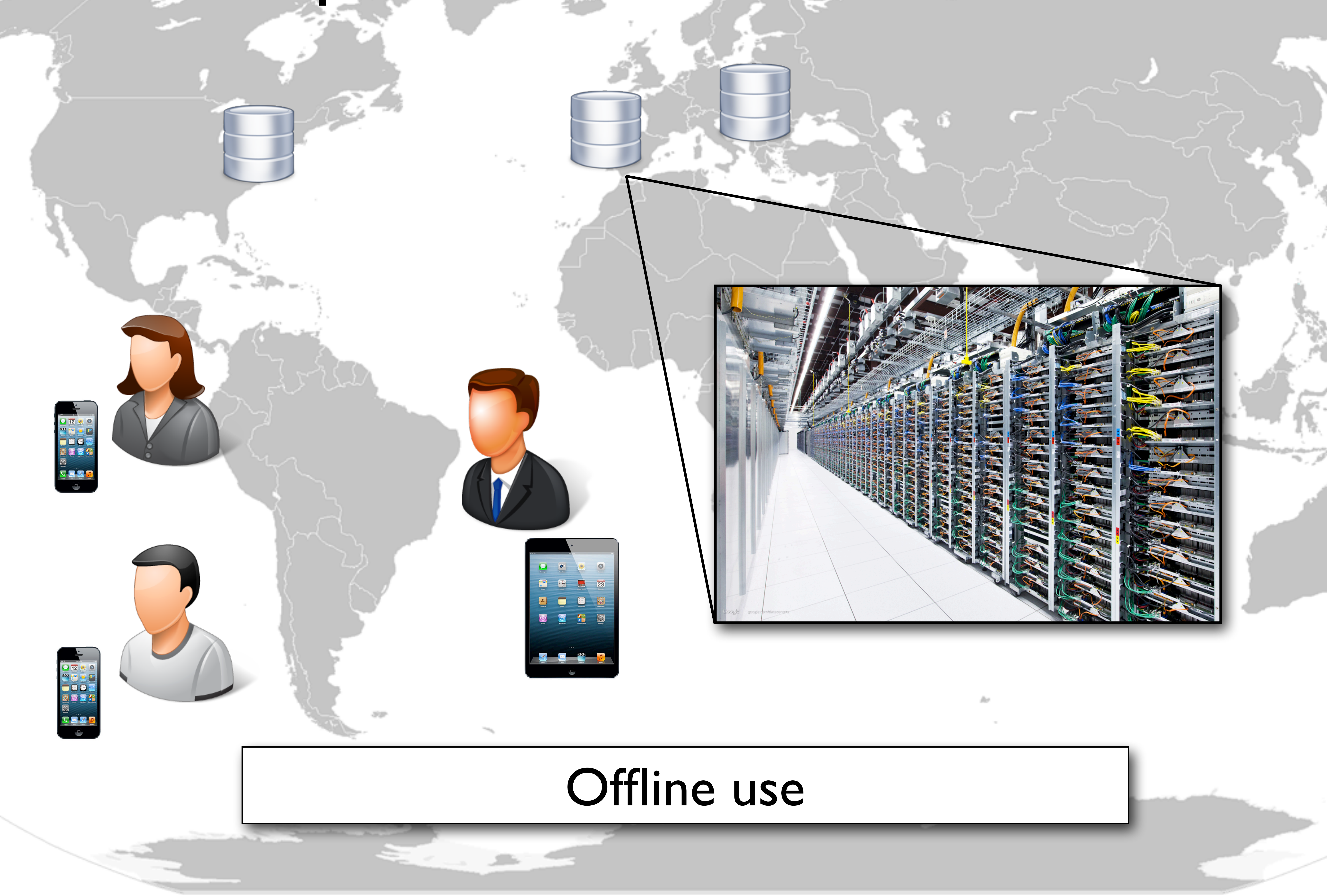
Disaster-tolerance, minimising latency

With thousands of machines inside

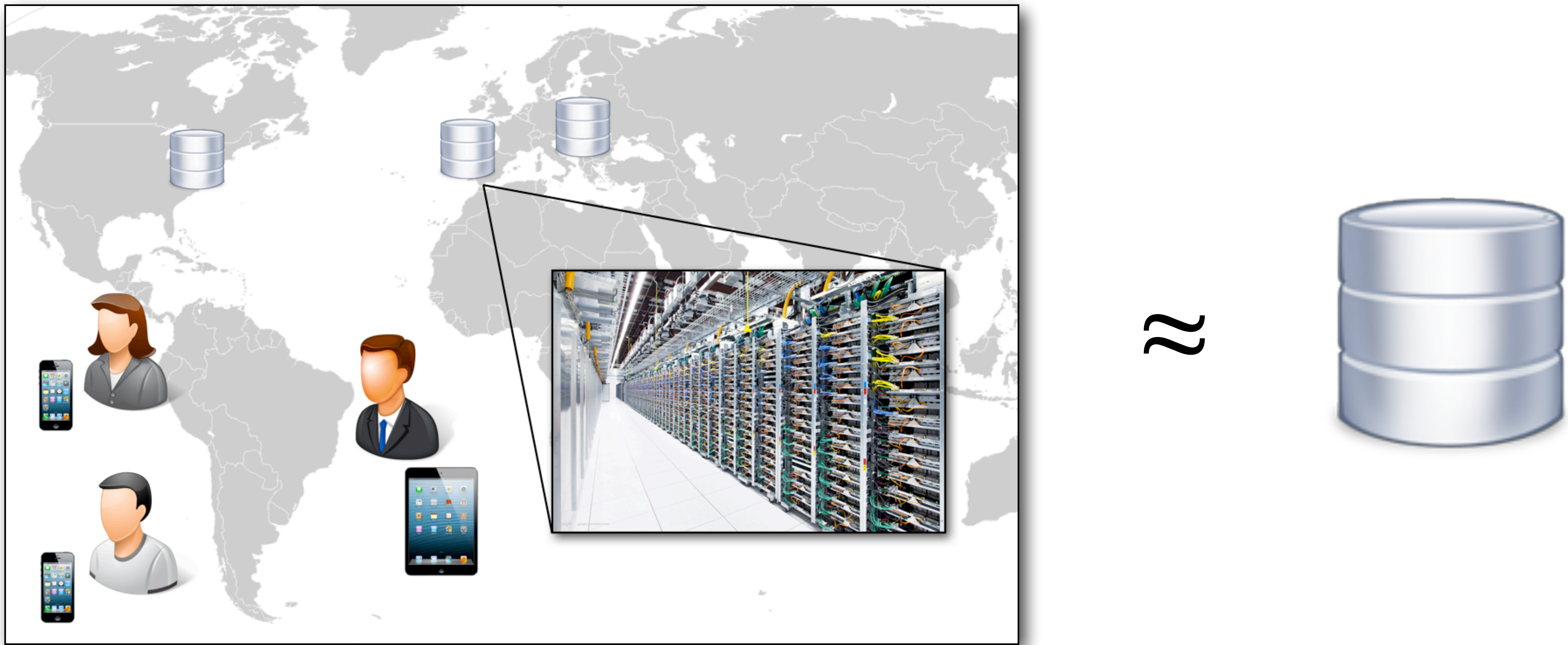


Load-balancing, fault-tolerance

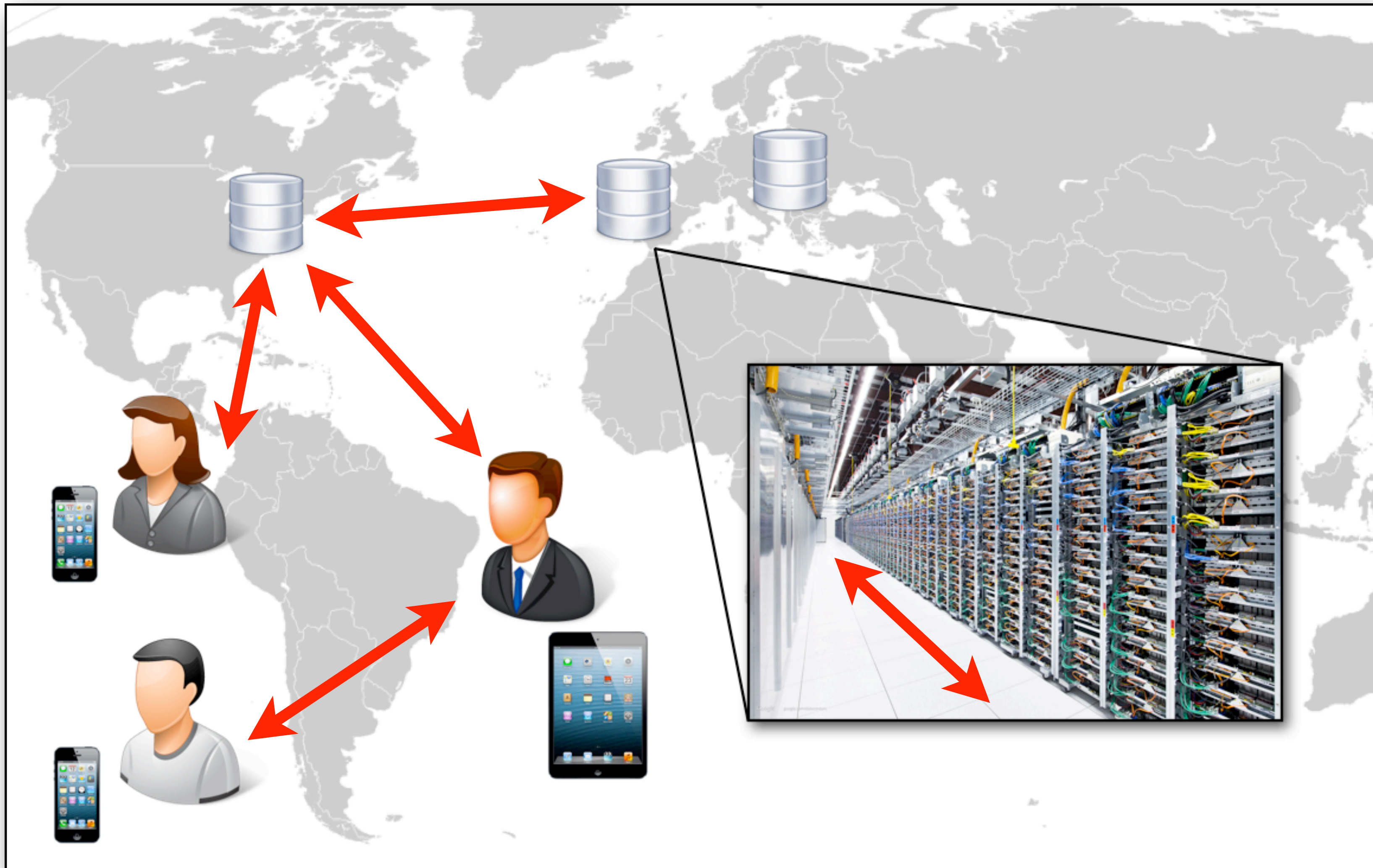
Replicas on mobile devices



Offline use



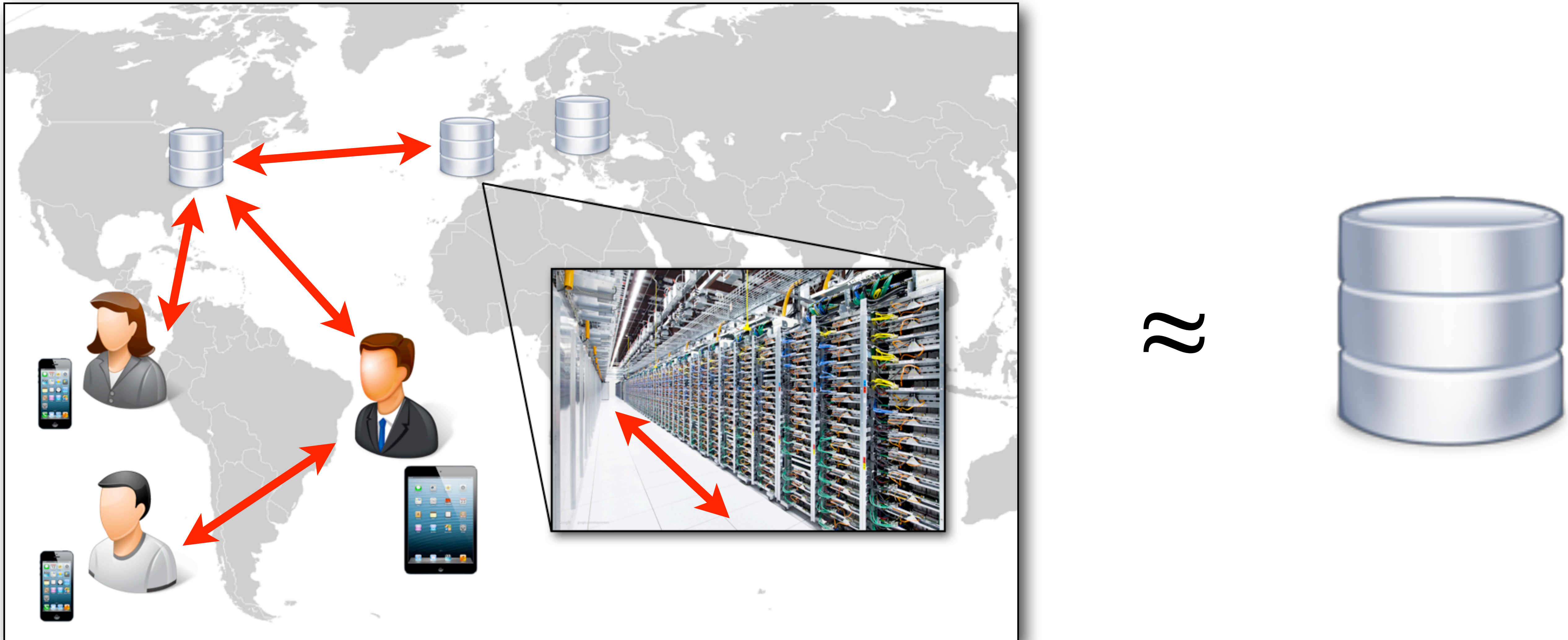
- Strong consistency model: the system behaves as if it processes requests serially on a centralised database - linearizability, serializability



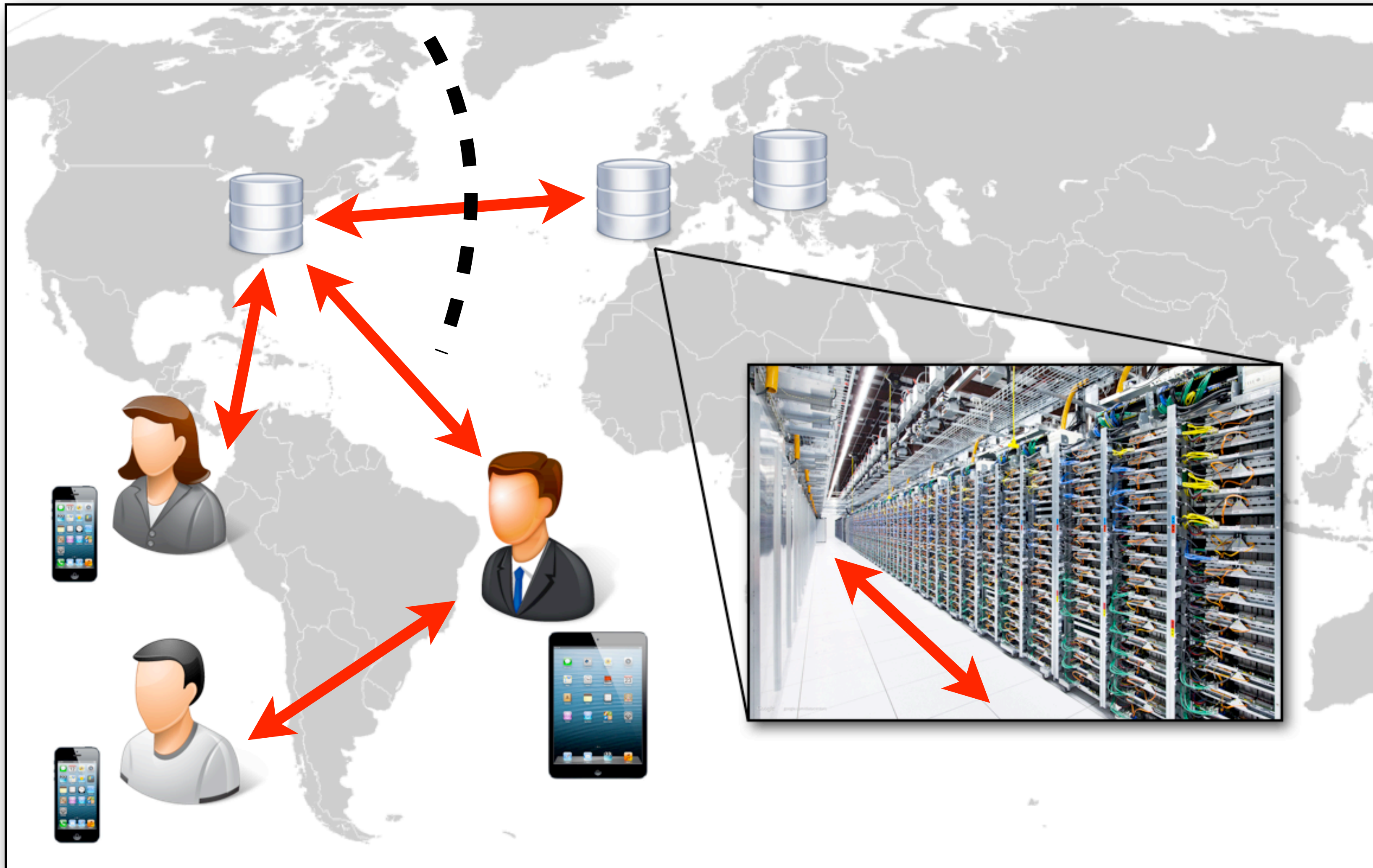
≈

- Strong consistency model: the system behaves as if it processes requests serially on a centralised database - linearizability, serializability

- Requires synchronisation: contact other replicas when processing a request

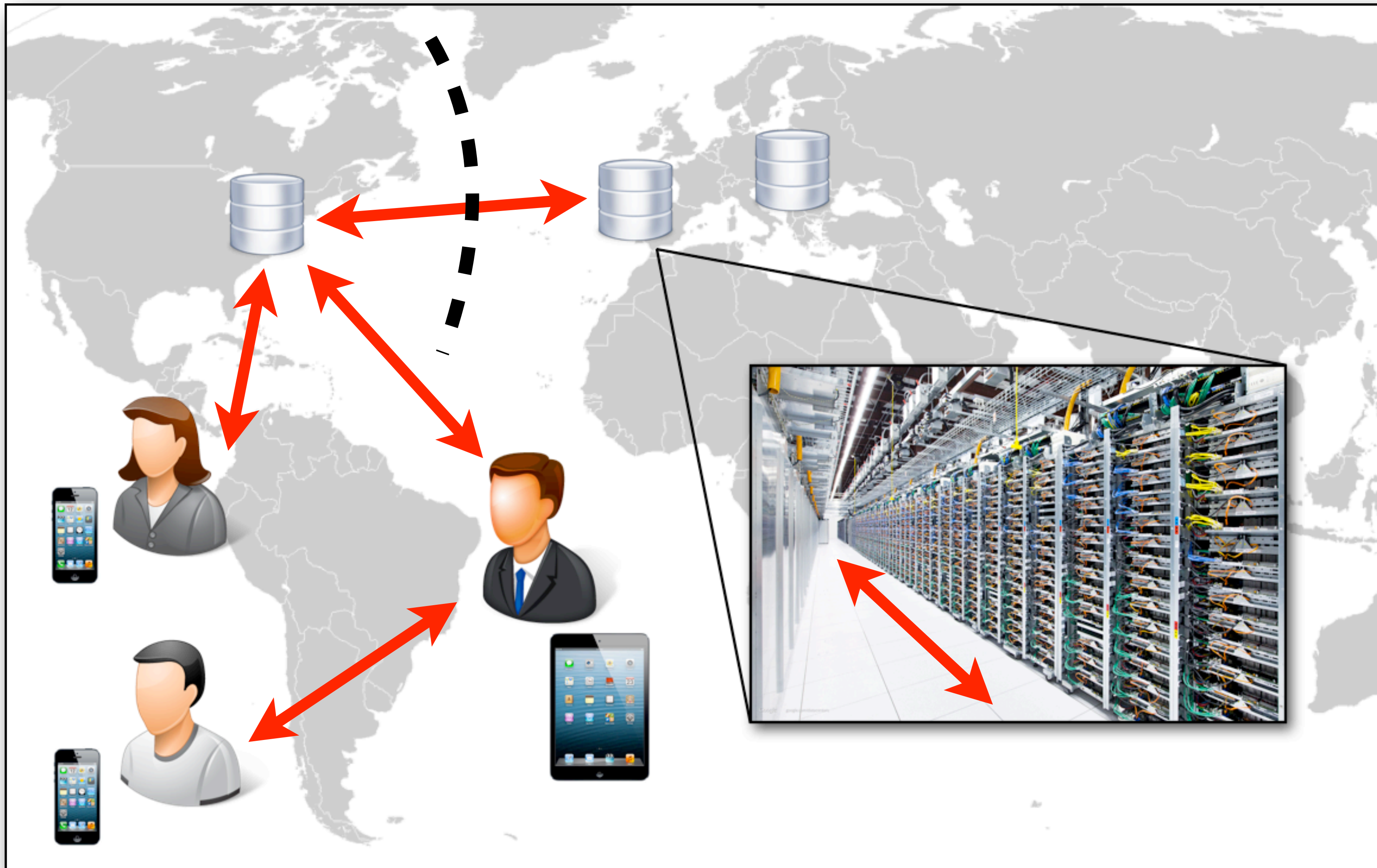


- Expensive: communication increases latency
- Impossible: either strong Consistency or Availability in the presence of network Partitions [CAP theorem]



$\approx$

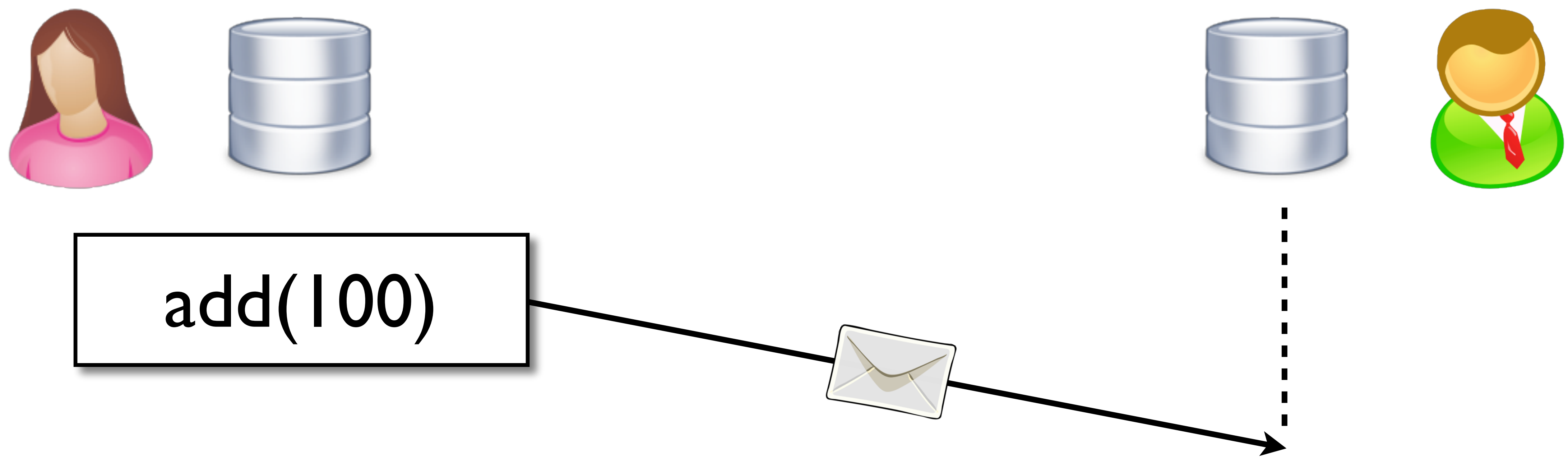
- Expensive: communication increases latency
- Impossible: either strong Consistency or Availability in the presence of network Partitions [CAP theorem]



≈

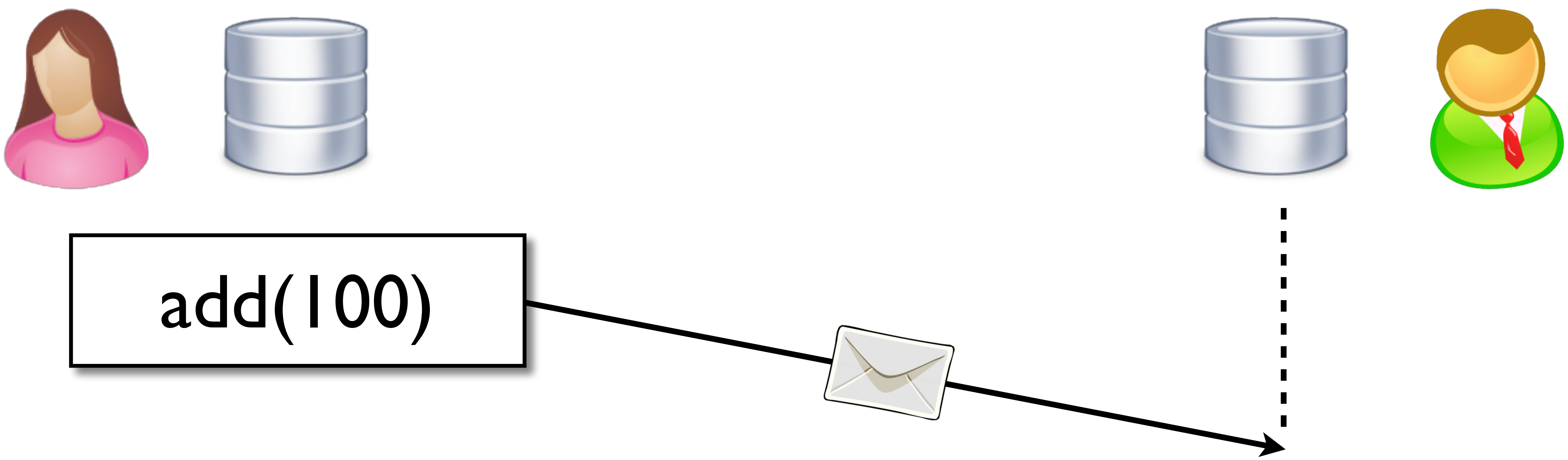
- Expensive: communication increases latency
- Impossible: either ~~strong Consistency~~ or Availability in the presence of network Partitions [CAP theorem]

Relaxing synchronisation



Process an update locally, propagate effects to other replicas later

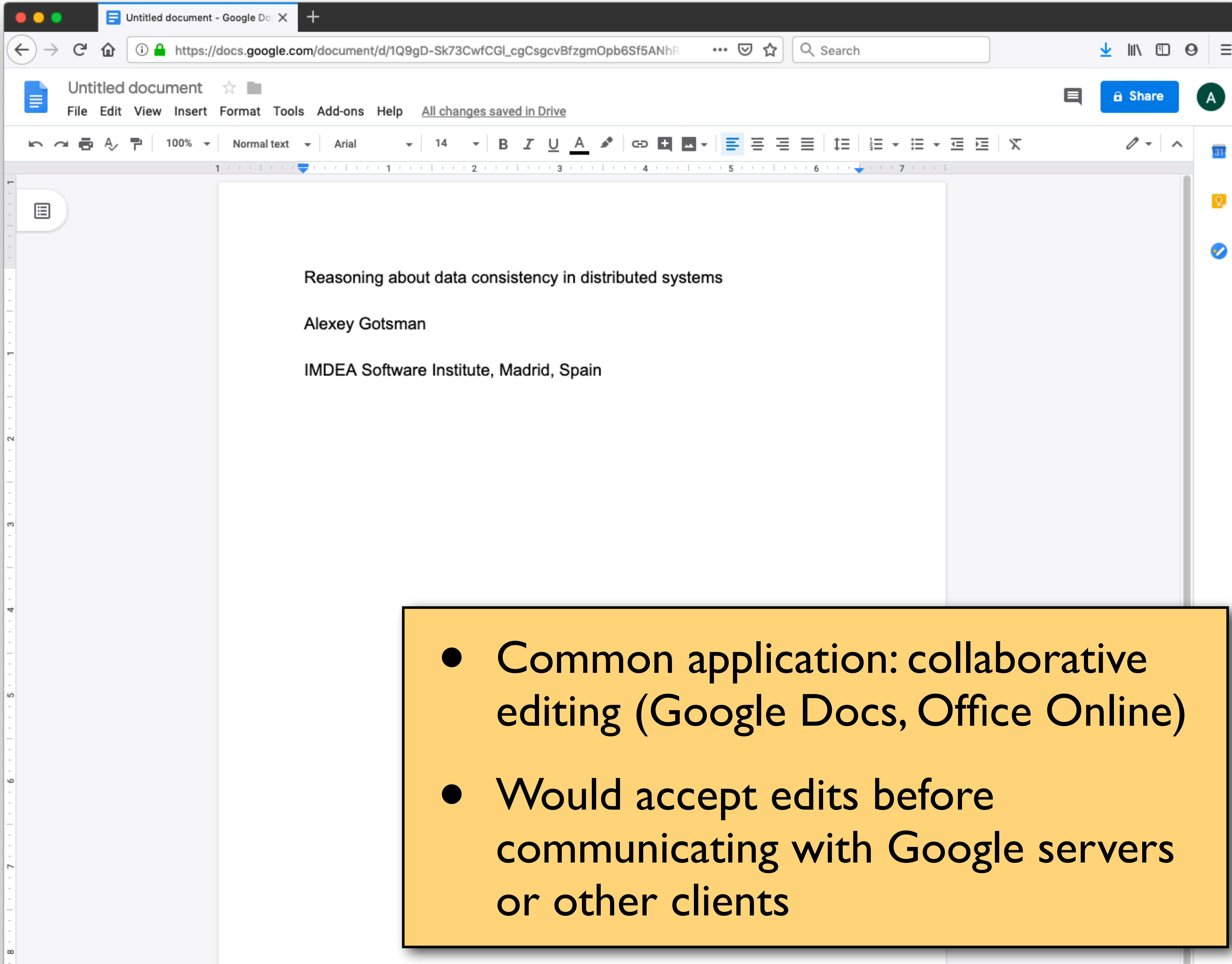
Relaxing synchronisation



Process an update locally, propagate effects to other replicas later

+ Better scalability & availability

- Weakens consistency: deposit seen with a delay



- Common application: collaborative editing (Google Docs, Office Online)
- Would accept edits before communicating with Google servers or other clients

NoSQL data stores

New generation of data stores with high scalability and low latency, but weak consistency

Cassandra

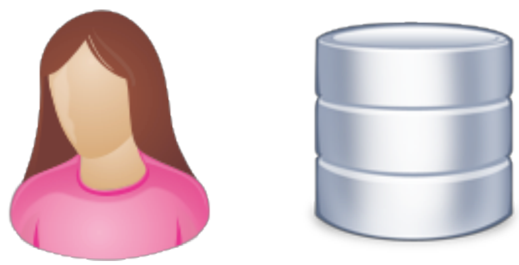
amazon DynamoDB

Microsoft Azure DocumentDB

riak

So what consistency guarantees do they provide?

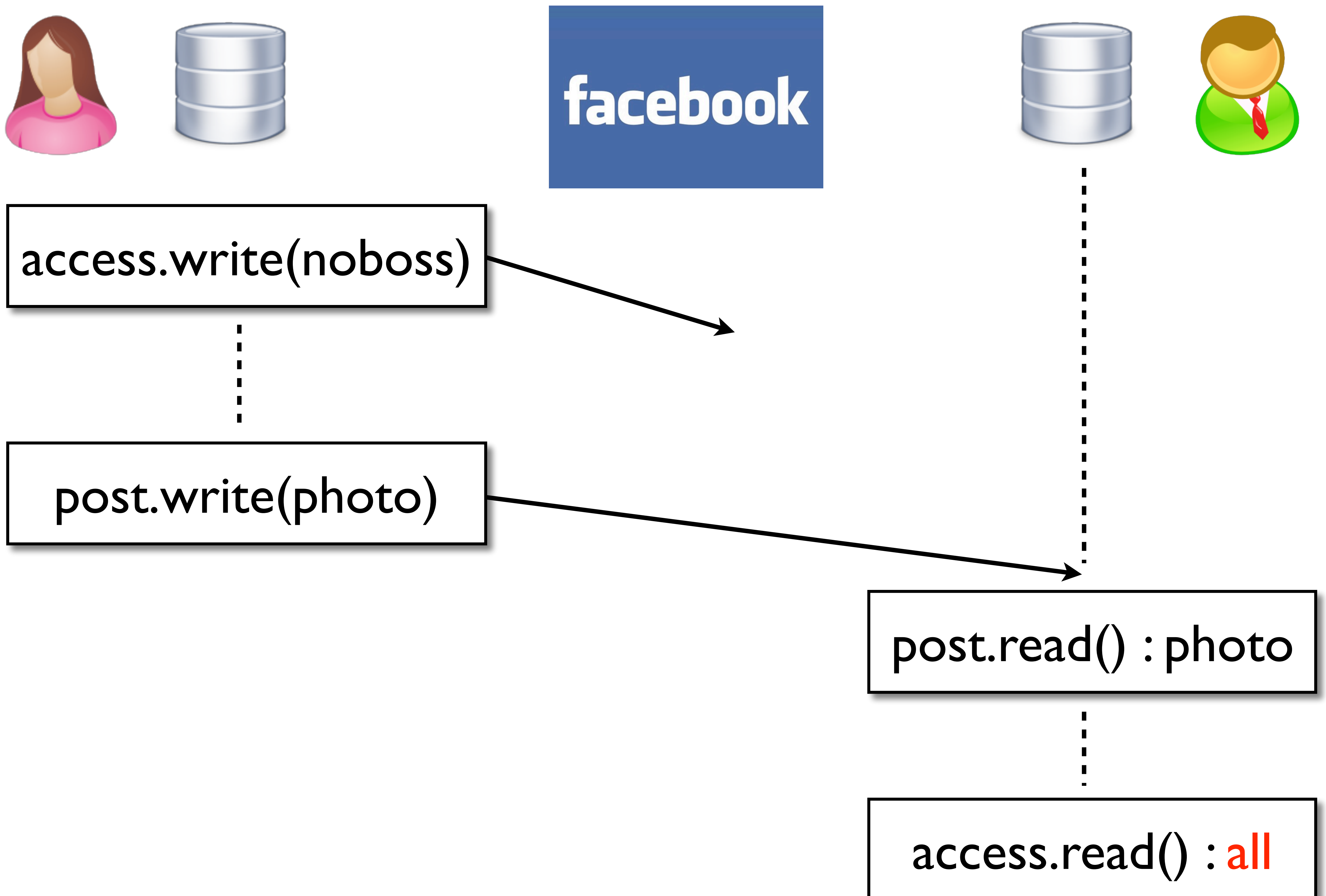
Anomalies



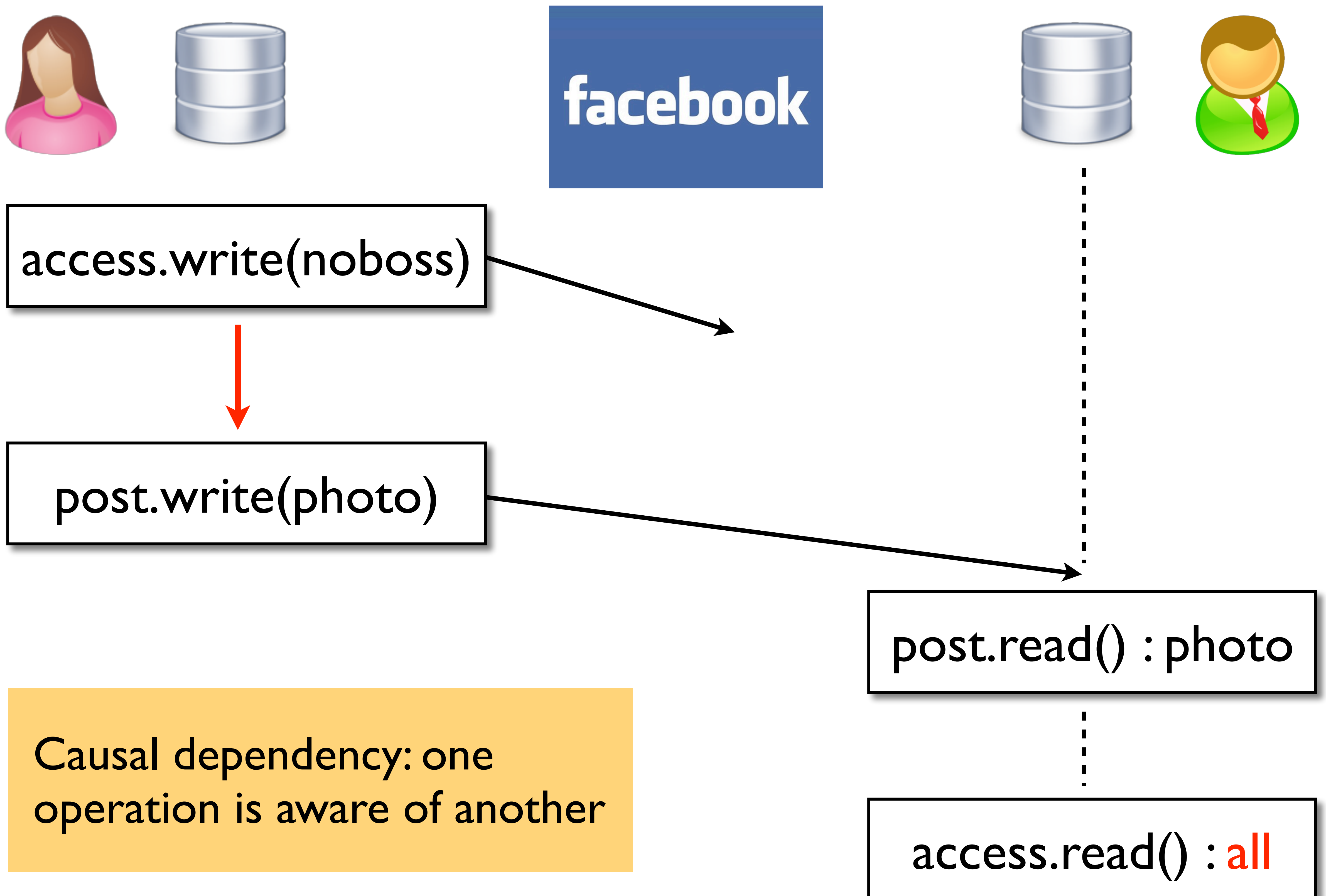
access.write(noboss)

post.write(photo)

Anomalies

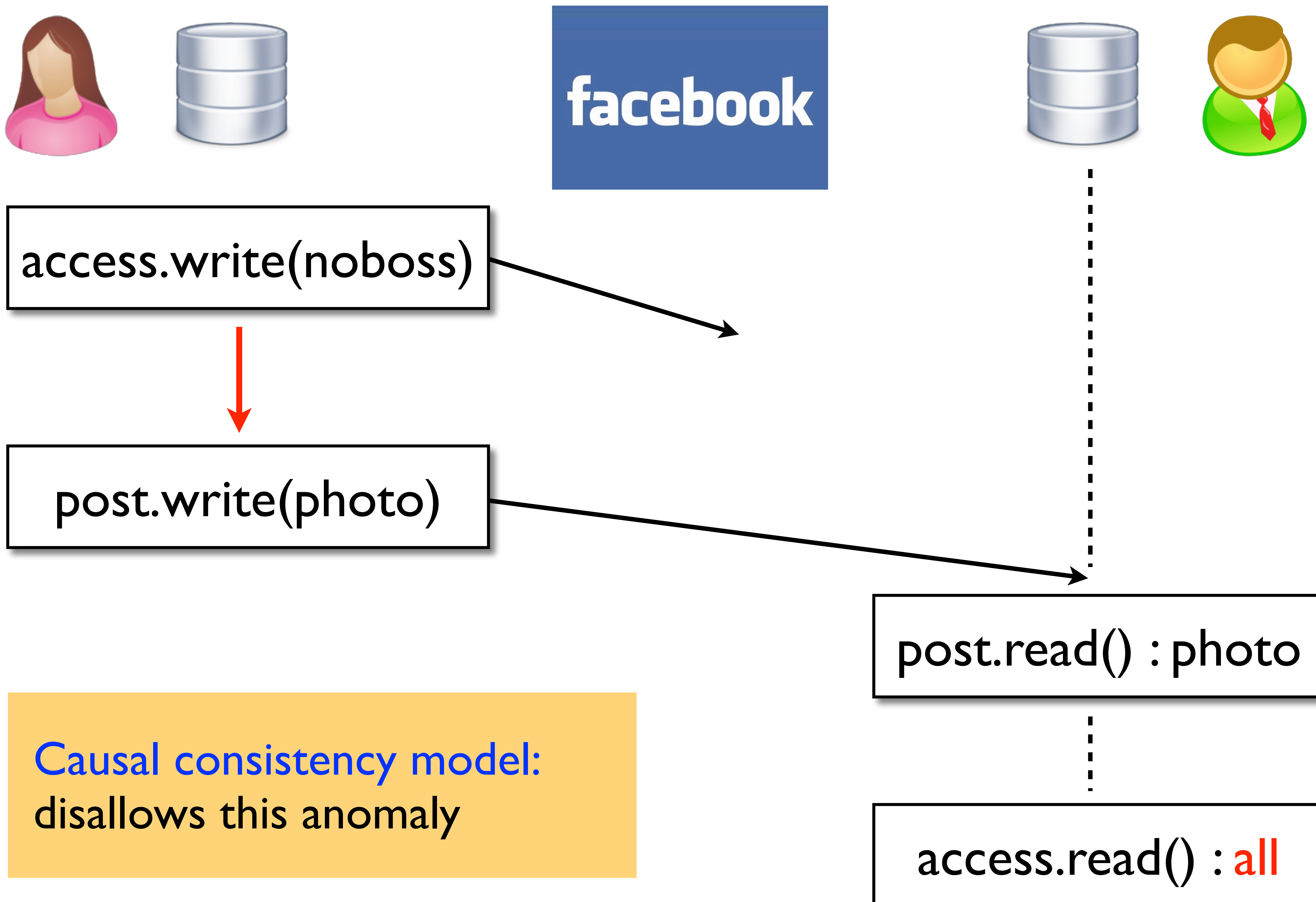

facebook
access.write(noboss)
post.write(photo)
post.read() : photo
access.read() : all

Anomalies



Causal dependency: one operation is aware of another

Anomalies



Causal consistency model:
disallows this anomaly

Early days

Poor guidelines on how to use the weakly consistent data stores: are we weakening consistency too much, too little, just right?

Early days

Poor guidelines on how to use the weakly consistent data stores: are we weakening consistency too much, too li

"If no new updates are made to the database, then replicas will eventually reach a consistent state"

practice

DOI:10.1145/1435417.1435432

Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.

BY WERNER VOGELS

Eventually Consistent

AT THE FOUNDATION of Amazon's cloud computing are infrastructure services such as Amazon's S3 (Simple Storage Service), SimpleDB, and EC2 (Elastic Compute Cloud) that provide the resources for constructing Internet-scale computing platforms and a great variety of applications. The requirements placed on these infrastructure services are very strict; they need to score high marks in the areas of security, scalability, availability, performance, and cost-effectiveness, and they need to meet these requirements while serving millions of customers around the globe, continuously.

Under the covers these services are massive distributed systems that operate on a worldwide scale. This scale creates additional challenges, because when a system processes trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and must be accounted for upfront in the design and architecture of the system. Given the worldwide scope of these systems, we use replication techniques ubiquitously to guarantee consistent performance and high availability. Although replication brings us closer to our goals, it cannot achieve them in a perfectly transparent manner; under a number of conditions the customers of these services will be confronted with the consequences of using replication techniques inside the services.

One of the ways in which this manifests itself is in the type of data consistency that is provided, particularly when many widespread distributed systems provide an *eventual consistency* model in the context of data replication. When designing these large-scale systems at Amazon, we use a set of guiding principles and abstractions related to large-scale data replication and focus on the trade-offs between high availability and data consistency. Here, I present some of the relevant background that has informed our approach to delivering reliable distributed systems that must operate on a global scale. (An earlier version of this article appeared as a posting on the "All Things Distributed" Weblog and was greatly improved with the help of its readers.)

Historical Perspective

In an ideal world there would be only one consistency model: when an update is made all observers would see that update. The first time this surfaced as difficult to achieve was in the database systems of the late 1970s. The best "period piece" on this topic is "Notes on Distributed Databases" by Bruce Lindsay et al.[5] It lays out the fundamental principles for database replication and discusses a number of techniques that deal with achieving consistency. Many of these techniques try to achieve *distribution transparency*—that is, to the user of the system it appears as if there is only one system instead of a number of collaborating systems. Many systems during this time took the approach that it was better to fail the complete system than to break this transparency.[2]

In the mid-1990s, with the rise of larger Internet systems, these practices were revisited. At that time people began to consider the idea that availability was perhaps the most impor-

Early days

Poor guidelines on how to use the weakly consistent data stores: are we weakening consistency too much, too li

"If no new updates are made to the database, then replicas will eventually reach a consistent state"


practice

DOI:10.1145/1435417.1435432

Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.

BY WERNER VOGELS

Eventually Consistent

AT THE FOUNDATION of Amazon's cloud computing are infrastructure services such as Amazon's S3 (Simple Storage Service), SimpleDB, and EC2 (Elastic Compute Cloud) that provide the resources for constructing Internet-scale computing platforms and a great variety of applications. The requirements placed on these infrastructure services are very strict; they need to score high marks in the areas of security, scalability, availability, performance, and cost-effectiveness, and they need to meet these requirements while serving millions of customers around the globe, continuously.

Under the covers these services are massive distributed systems that operate on a worldwide scale. This scale creates additional challenges, because when a system processes trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and must be accounted for upfront in the design and architecture of the system. Given the worldwide scope of these systems, we use replication techniques ubiquitously to guarantee consistent performance and high availability. Although replication brings us closer to our goals, it cannot achieve them in a perfectly transparent manner; under a number of conditions the customers of these services will be confronted with the consequences of using replication techniques inside the services.

One of the ways in which this manifests itself is in the type of data consistency that is provided, particularly when many widespread distributed systems provide an *eventual consistency* model in the context of data replication. When designing these large-scale systems at Amazon, we use a set of guiding principles and abstractions related to large-scale data replication and focus on the trade-offs between high availability and data consistency. Here, I present some of the relevant background that has informed our approach to delivering reliable distributed systems that must operate on a global scale. (An earlier version of this article appeared as a posting on the "All Things Distributed" Weblog and was greatly improved with the help of its readers.)

Historical Perspective

In an ideal world there would be only one consistency model: when an update is made all observers would see that update. The first time this surfaced as difficult to achieve was in the database systems of the late 1970s. The best "period piece" on this topic is "Notes on Distributed Databases" by Bruce Lindsay et al.[5] It lays out the fundamental principles for database replication and discusses a number of techniques that deal with achieving consistency. Many of these techniques try to achieve *distribution transparency*—that is, to the user of the system it appears as if there is only one system instead of a number of collaborating systems. Many systems during this time took the approach that it was better to fail the complete system than to break this transparency.[2]

In the mid-1990s, with the rise of larger Internet systems, these practices were revisited. At that time people began to consider the idea that availability was perhaps the most impor-

Early days

Poor guidelines on how to use the weakly consistent data stores: are we weakening consistency too much, too li

"If no new updates are made to the database, then replicas will eventually reach a consistent state"

practice

DOI:10.1145/1435417.1435432

Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.

BY WERNER VOGELS

Eventually Consistent

AT THE FOUNDATION of Amazon's cloud computing are infrastructure services such as Amazon's S3 (Simple Storage Service), SimpleDB, and EC2 (Elastic Compute Cloud) that provide the resources for constructing Internet-scale computing platforms and a great variety of applications. The requirements placed on these infrastructure services are very strict; they need to score high marks in the areas of security, scalability, availability, performance, and cost-effectiveness, and they need to meet these requirements while serving millions of customers around the globe, continuously.

Under the covers these services are massive distributed systems that operate on a worldwide scale. This scale creates additional challenges, because when a system processes trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and must be accounted for upfront in the design and architecture of the system. Given the worldwide scope of these systems, we use replication techniques ubiquitously to guarantee consistent performance and high availability. Although replication brings us closer to our goals, it cannot achieve them in a perfectly transparent manner; under a number of conditions the customers of these services will be confronted with the consequences of using replication techniques inside the services.

One of the ways in which this manifests itself is in the type of data consistency that is provided, particularly when many widespread distributed systems provide an *eventual consistency* model in the context of data replication. When designing these large-scale systems at Amazon, we use a set of guiding principles and abstractions related to large-scale data replication and focus on the trade-offs between high availability and data consistency. Here, I present some of the relevant background that has informed our approach to delivering reliable distributed systems that must operate on a global scale. (An earlier version of this article appeared as a posting on the "All Things Distributed" Weblog and was greatly improved with the help of its readers.)

Historical Perspective

In an ideal world there would be only one consistency model: when an update is made all observers would see that update. The first time this surfaced as difficult to achieve was in the database systems of the late 1970s. The best "period piece" on this topic is "Notes on Distributed Databases" by Bruce Lindsay et al.[5] It lays out the fundamental principles for database replication and discusses a number of techniques that deal with achieving consistency. Many of these techniques try to achieve *distribution transparency*—that is, to the user of the system it appears as if there is only one system instead of a number of collaborating systems. Many systems during this time took the approach that it was better to fail the complete system than to break this transparency.[2]

In the mid-1990s, with the rise of larger Internet systems, these practices were revisited. At that time people began to consider the idea that availability was perhaps the most impor-

2008

TOWARDS A CLOUD COMPUTING RESEARCH AGENDA

Ken Birman, Gregory Chockler, Robbert van Renesse

This particular example is a good one because, as we'll see shortly, if there was a single overarching theme within the keynote talks, it turns out to be that strong synchronization of the sort provided by a locking service must be avoided like the plague. This doesn't diminish the need for a tool like Chubby; when locking actually can't be avoided, one wants a reliable, standard, provably correct

2008

TOWARDS A CLOUD COMPUTING RESEARCH AGENDA

Ken Birman, Gregory Chockler, Robbert van Renesse

This particular example is a good one because, as we'll see shortly, if there was a single overarching theme within the keynote talks, it turns out to be that strong synchronization of the sort provided by a locking service must be avoided like the plague. This doesn't diminish the need for a tool like Chubby; when locking actually can't be avoided, one wants a reliable, standard, provably correct

2013

F1: A Distributed SQL Database That Scales

Jeff Shute, Chad Whipkey, David Menestrina, Radek Vingralek, Eric Rollins, Stephan Ellner, Traian Stancescu, Bart Samwel, Mircea Oancea, John Cieslewicz, Himani Apte, Ben Handy, Kyle Littlefield, Ian Rae*

Google, Inc.
*University of Wisconsin-Madison

ABSTRACT

F1 is a distributed relational database system built at Google to support the AdWords business. F1 is a hybrid database that combines high availability, the scalability of NoSQL systems like Bigtable, and the consistency and us-

consistent and correct data.

Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains.

Strong vs weak consistency

- Pay-off from weakening consistency often worth it: higher scalability, lower latency in geo-distribution, offline access
  - Both strong and weak systems used in industry
- But programmers need help in using it:
  - Programming abstractions for weak consistency
  - Methods for reasoning about how weakening consistency affects application correctness

Also centralised SQL databases

Don't provide strong consistency either by default or at all: to exploit single-node concurrency

Also centralised SQL databases

Don't provide strong consistency either by default or at all: to exploit single-node concurrency

...since 1975

Granularity of Locks and Degrees of Consistency
in a Shared Data Base

J.N. Gray
R.A. Lorie
G.R. Putzolu
I.L. Traiger

IBM Research Laboratory
San Jose, California

ABSTRACT: In the first part of the paper the problem of choosing the granularity (size) of lockable objects is introduced and the related tradeoff between concurrency and overhead is discussed. A locking protocol which allows simultaneous locking at various granularities by different transactions is presented. It is based on the introduction of additional lock modes besides the

Also centralised SQL databases

Don't provide strong consistency either by default or at all: to exploit single-node concurrency

...since 1975

Granularity of Locks and Degrees of Consistency
in a Shared Data Base

J.N. Gray
R.A. Lorie
G.R. Putzolu
I.L. Traiger

IBM Research Laboratory
San Jose, California

ABSTRACT: In the first part of th
the granularity (size) of lockabl
related tradeoff between concurren
locking protocol which allows s
granularities by different transac
on the introduction of additional lock modes besides the

Are applications OK with this?

[SIGMOD'17]

ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications

Todd Warszawski, Peter Bailis
Stanford InfoLab

ABSTRACT

In theory, database transactions protect application data from corruption and integrity violations. In practice, database transactions frequently execute under weak isolation that exposes programs to a range of concurrency anomalies, and programmers may fail to correctly employ transactions. While low transaction volumes mask many potential concurrency-related errors under normal operation, determined adversaries can exploit them programmatically for fun and profit. In this paper, we formalize a new kind of attack on database-backed applications called an *ACIDRain attack*, in which an adversary systematically exploits concurrency-related vulnerabilities via programmatically accessible APIs. These attacks are not theoretical: ACIDRain attacks have already occurred in a handful of applications in the wild, including one attack which bankrupted a popular Bitcoin exchange. To proactively detect the potential for ACIDRain attacks, we extend the theory of weak isolation to analyze latent potential for non-serializable behavior under concurrent web API calls. We introduce a language-agnostic method for detecting potential isolation anomalies in web applications, called Abstract Anomaly Detection (2AD), that uses dynamic traces of database accesses to efficiently reason about the space of possible concurrent interleavings. We apply a prototype 2AD analysis tool to 12 popular self-hosted eCommerce applications written in four languages and deployed on over 2M websites. We identify and verify 22 critical ACIDRain attacks that allow attackers to corrupt store inventory, over-spend gift cards, and steal inventory.

(a)
```
def withdraw(amt, user_id):
  bal = readBalance(user_id)
  if (bal >= amt):
    writeBalance(bal − amt, user_id)
```

(b)
```
def withdraw(amt, user_id):
  beginTxn()
  bal = readBalance(user_id)
  if (bal >= amt):
    writeBalance(bal − amt, user_id)
  commit()
```

Figure 1: (a) A simplified example of code that is vulnerable to an ACIDRain attack allowing overdraft under concurrent access. Two concurrent instances of the `withdraw` function could both read balance \$100, check that \$100 $\geq$ \$99, and each allow \$99 to be withdrawn, resulting \$198 total withdrawals. (b) Example of how transactions could be inserted to address this error. However, even this code is vulnerable to attack at isolation levels at or below Read Committed, unless explicit locking such as `SELECT FOR UPDATE` is used. While this scenario closely resembles textbook examples of improper transaction use, in this paper, we show that widely-deployed eCommerce applications are similarly vulnerable to such ACIDRain attacks, allowing corruption of application state and theft of assets.

[SIGMOD'17]

ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications

Todd Warszawski, Peter Bailis
Stanford InfoLab

ABSTRACT

In theory, database transactions protect application data from corruption and integrity violations. In practice, database transactions frequently execute under weak isolation that exposes programs to a range of concurrency anomalies, and programmers may fail to correctly employ transactions. While low transaction volumes mask many potential concurrency-related errors under normal operation, determined adversaries can exploit them programmatically for fun and profit. In this paper, we formalize a new kind of attack on database-backed applications called an *ACIDRain attack*, in which an adversary systematically exploits concurrency-related vulnerabilities via programmatically accessible APIs. These attacks are not theoretical: ACIDRain attacks have already occurred in a handful of applications in the wild, including one attack which bankrupted a popular Bitcoin exchange. To proactively detect the pot
ACIDRain attacks, we extend the theory of weak isolation t
latent potential for non-serializable behavior under concu
API calls. We introduce a language-agnostic method for
potential isolation anomalies in web applications, called
Anomaly Detection (2AD), that uses dynamic traces of
accesses to efficiently reason about the space of possible c
interleavings. We apply a prototype 2AD analysis tool to 1
self-hosted eCommerce applications written in four langu
deployed on over 2M websites. We identify and verify 2
ACIDRain attacks that allow attackers to corrupt store i
over-spend gift cards, and steal inventory.

```
def withdraw(amt, user_id):                    (a)
  bal = readBalance(user_id)
  if (bal >= amt):
    writeBalance(bal − amt, user_id)
```

```
def withdraw(amt, user_id):                    (b)
  beginTxn()
  bal = readBalance(user_id)
  if (bal >= amt):
    writeBalance(bal − amt, user_id)
  commit()
```

corruption of application state and theft of assets.

No! E-commerce applications can be hacked by exploiting weak consistency of back-end databases

Weak shared-memory models

- Multicore processors: x86, ARM

  Multiprocessor ~ distributed system

- Programming languages: C/C++, Java

  Due to compiler optimisations

This course

- Programming abstractions for weak consistency
- Methods for specification
- Methods and tools for reasoning about application correctness and consistency needs
- Implementing strong consistency

Strong consistency and the CAP theorem

Data model

- Database system manages a set of objects:
  Obj = $\{x, y, z \ldots\}$
- Objects associated with types Type = $\{\tau, \ldots\}$
- For each type $\tau \in$ Type:
  - Set of operations Op_τ, including arguments
  - Return values: Val_τ

Data model

- Integer register
  - $\text{Op}_{\text{intreg}} = \{\text{read}, \text{write}(k) \mid k \in \mathbb{Z}\}$
  - $\text{Val}_{\text{intreg}} = \mathbb{Z} \cup \{\text{ok}\}$
- Counter:
  - $\text{Op}_{\text{counter}} = \{\text{read}, \text{add}(k) \mid k \in \mathbb{N}\}$
  - $\text{Val}_{\text{counter}} = \mathbb{N} \cup \{\text{ok}\}$

Sequential semantics

- Semantics in an ordinary programming language
- For each type $\tau \in$ Type: set of states State_τ, initial state $\sigma_0 \in \text{State}_\tau$
  - $\text{State}_{\text{intreg}} = \mathbb{Z}$
  - $\text{State}_{\text{counter}} = \mathbb{N}$
- Semantics of an operation op:
  - $[\![\text{op}]\!]_{\text{val}} \in \text{State}_\tau \to \text{Value}_\tau$
  - $[\![\text{op}]\!]_{\text{state}} \in \text{State}_\tau \to \text{State}_\tau$

Register semantics

- State = $\mathbb{Z}$
- $[\![\text{write}(k)]\!]_{\text{state}}(\sigma) = k$
- $[\![\text{write}]\!]_{\text{val}}(\sigma) = \text{ok}$
- $[\![\text{read}]\!]_{\text{state}}(\sigma) = \sigma$
- $[\![\text{read}]\!]_{\text{val}}(\sigma) = \sigma$

Counter semantics

- State = $\mathbb{N}$
- $[\![\text{add}(k)]\!]_{\text{state}}(\sigma) = \sigma+k$
- $[\![\text{add}(k)]\!]_{\text{val}}(\sigma) = \text{ok}$
- $[\![\text{read}]\!]_{\text{state}}(\sigma) = \sigma$
- $[\![\text{read}]\!]_{\text{val}}(\sigma) = \sigma$

Counter semantics

- State = $\mathbb{N}$
- $[\![\mathrm{add}(k)]\!]_{\mathrm{state}}(\sigma) = \sigma + k$
- $[\![\mathrm{add}(k)]\!]_{\mathrm{val}}(\sigma) = \mathrm{ok}$
- $[\![\mathrm{read}]\!]_{\mathrm{state}}(\sigma) = \sigma$
- $[\![\mathrm{read}]\!]_{\mathrm{val}}(\sigma) = \sigma$

read-only operation:
$[\![\mathrm{op}]\!]_{\mathrm{state}}(\sigma) = \sigma$

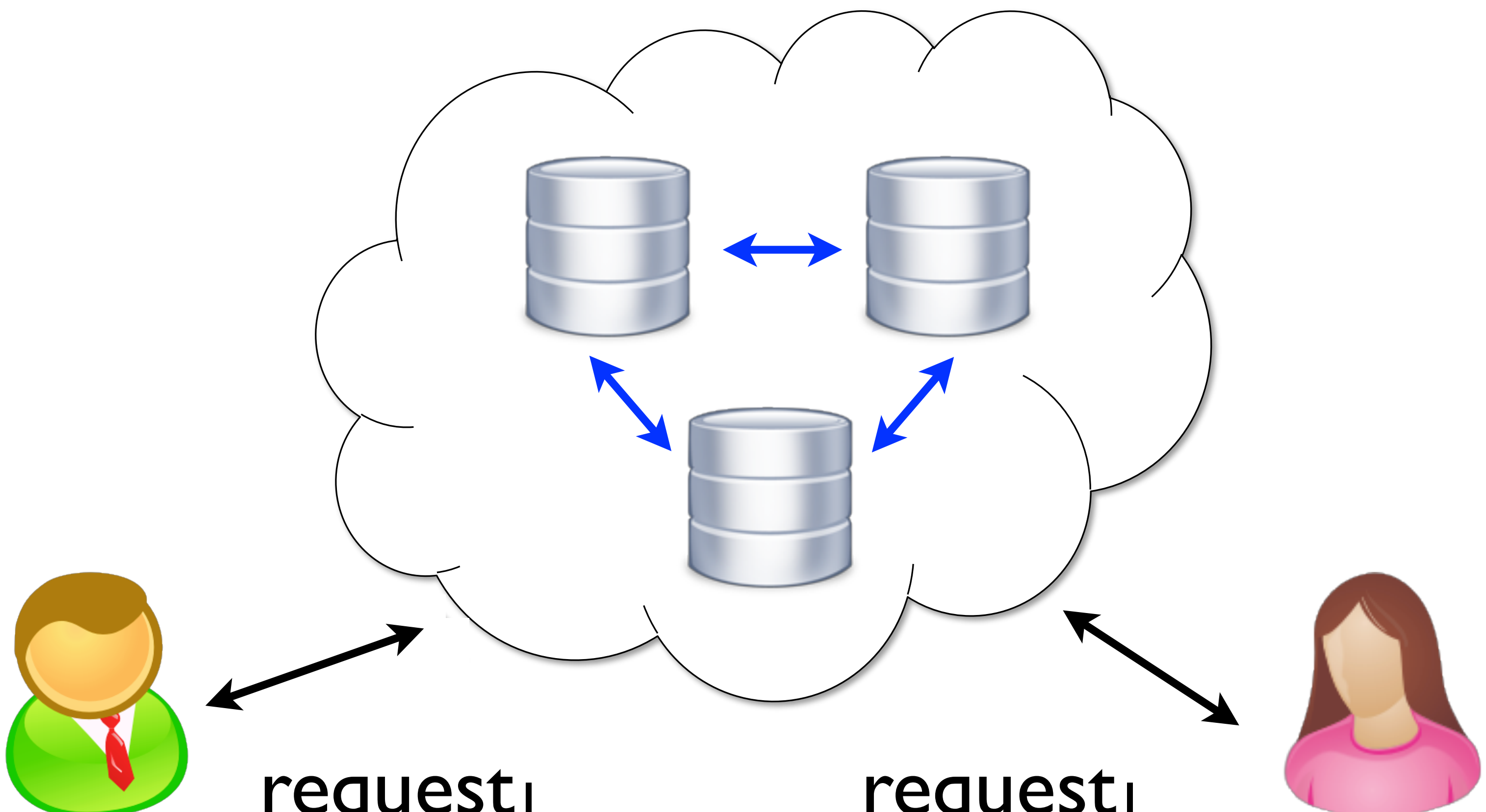
Counter semantics

- State = $\mathbb{N}$
- $[\![\text{add}(k)]\!]_{\text{state}}(\sigma) = \sigma + k$
- $[\![\text{add}(k)]\!]_{\text{val}}(\sigma) = \text{ok}$
- $[\![\text{read}]\!]_{\text{state}}(\sigma) = \sigma$
- $[\![\text{read}]\!]_{\text{val}}(\sigma) = \sigma$

update operation

read-only operation:
$[\![\text{op}]\!]_{\text{state}}(\sigma) = \sigma$

Consistency specification

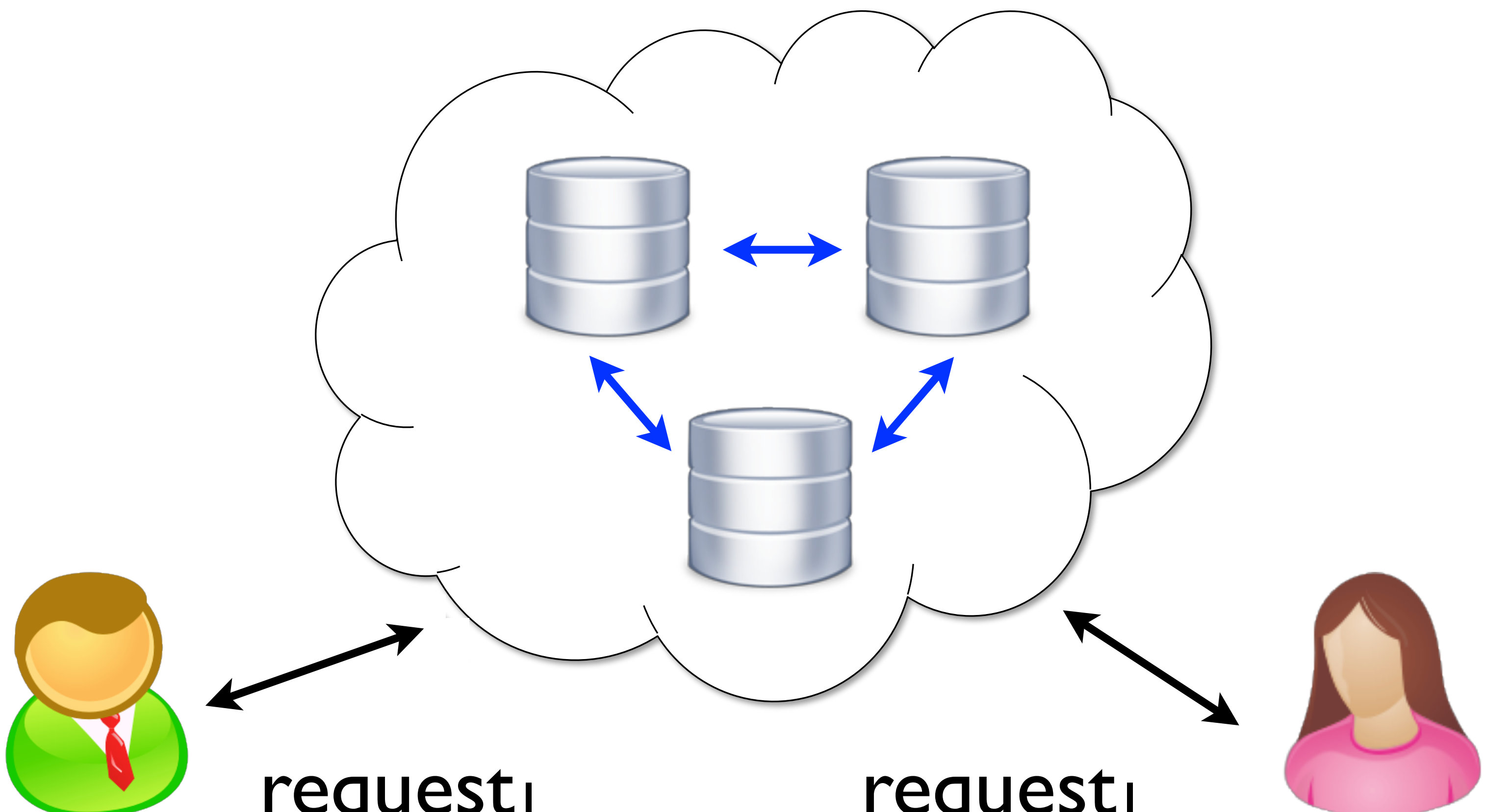


request$_1$
response$_1$
request$_2$
response$_2$
...

request$_1$
response$_1$
request$_2$
response$_2$
...

Clients issue requests and get responses:
history records the interactions in a single execution

Consistency specification

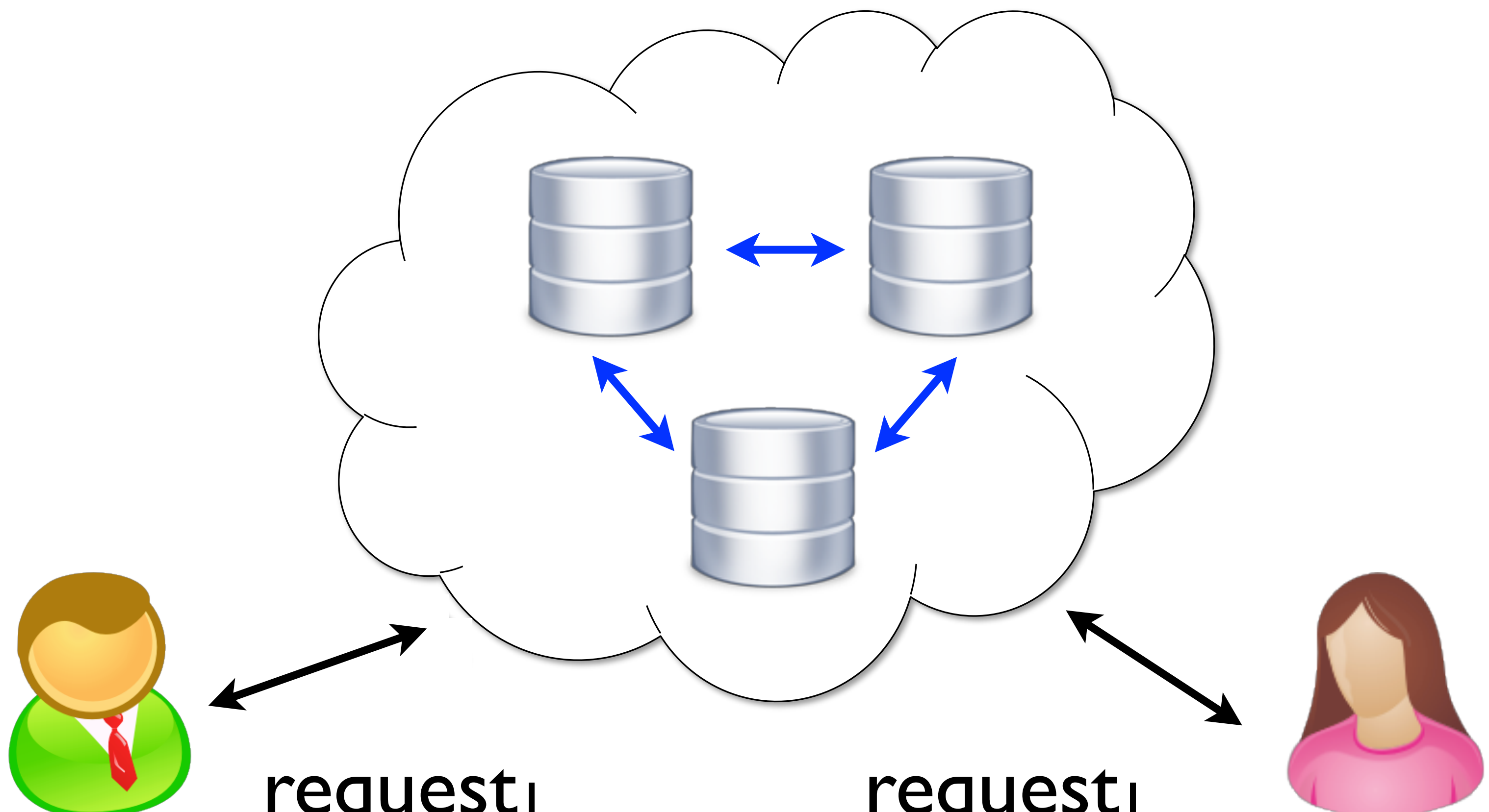


request$_1$
response$_1$
request$_2$
response$_2$
...

request$_1$
response$_1$
request$_2$
response$_2$
...

Assume every request yields a response
No next request until the previous one responded

Consistency specification

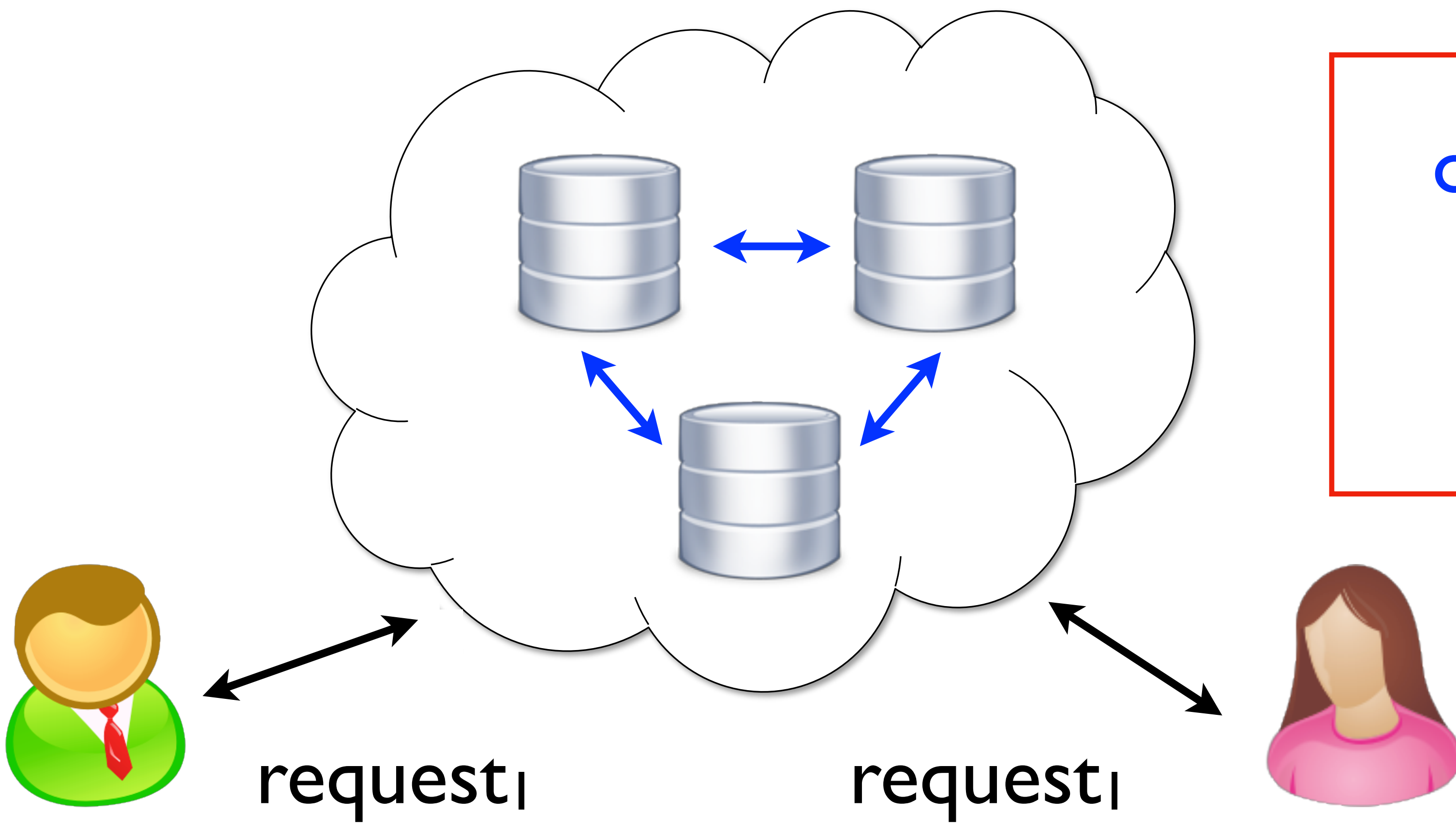


request$_1$	request$_1$
response$_1$	response$_1$
request$_2$	request$_2$
response$_2$	response$_2$
…	…

$\left.\begin{matrix}\text{request}_2\\ \text{response}_2\end{matrix}\right\}$ event e

Assume every request yields a response
No next request until the previous one responded

Consistency specification



obj(e) op(e) rval(e)

x.write(42) : ok

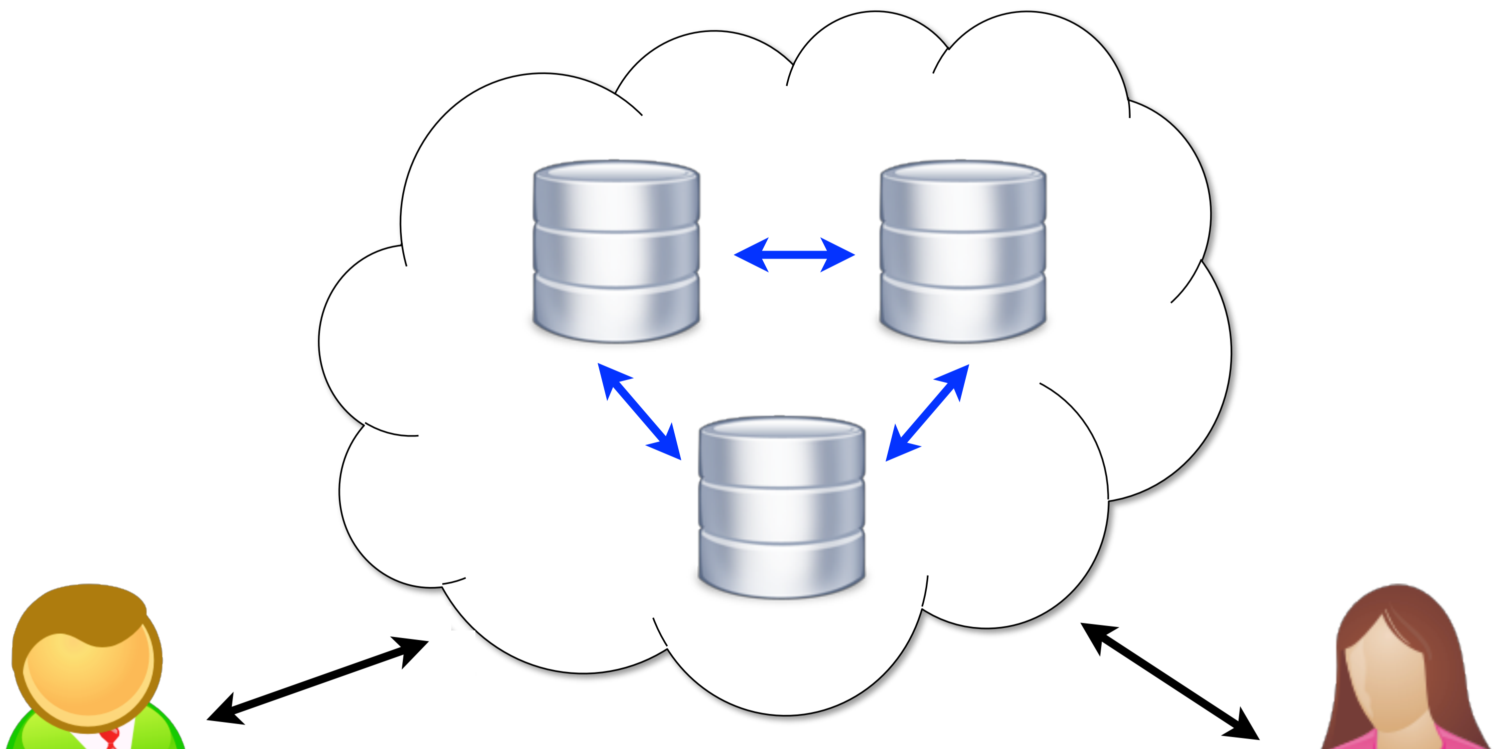
request$_1$	request$_1$
response$_1$	response$_1$
request$_2$	request$_2$
response$_2$	response$_2$
…	…

$\left.\begin{matrix}\text{request}_2\\ \text{response}_2\end{matrix}\right\}$ event e

Assume every request yields a response

No next request until the previous one responded

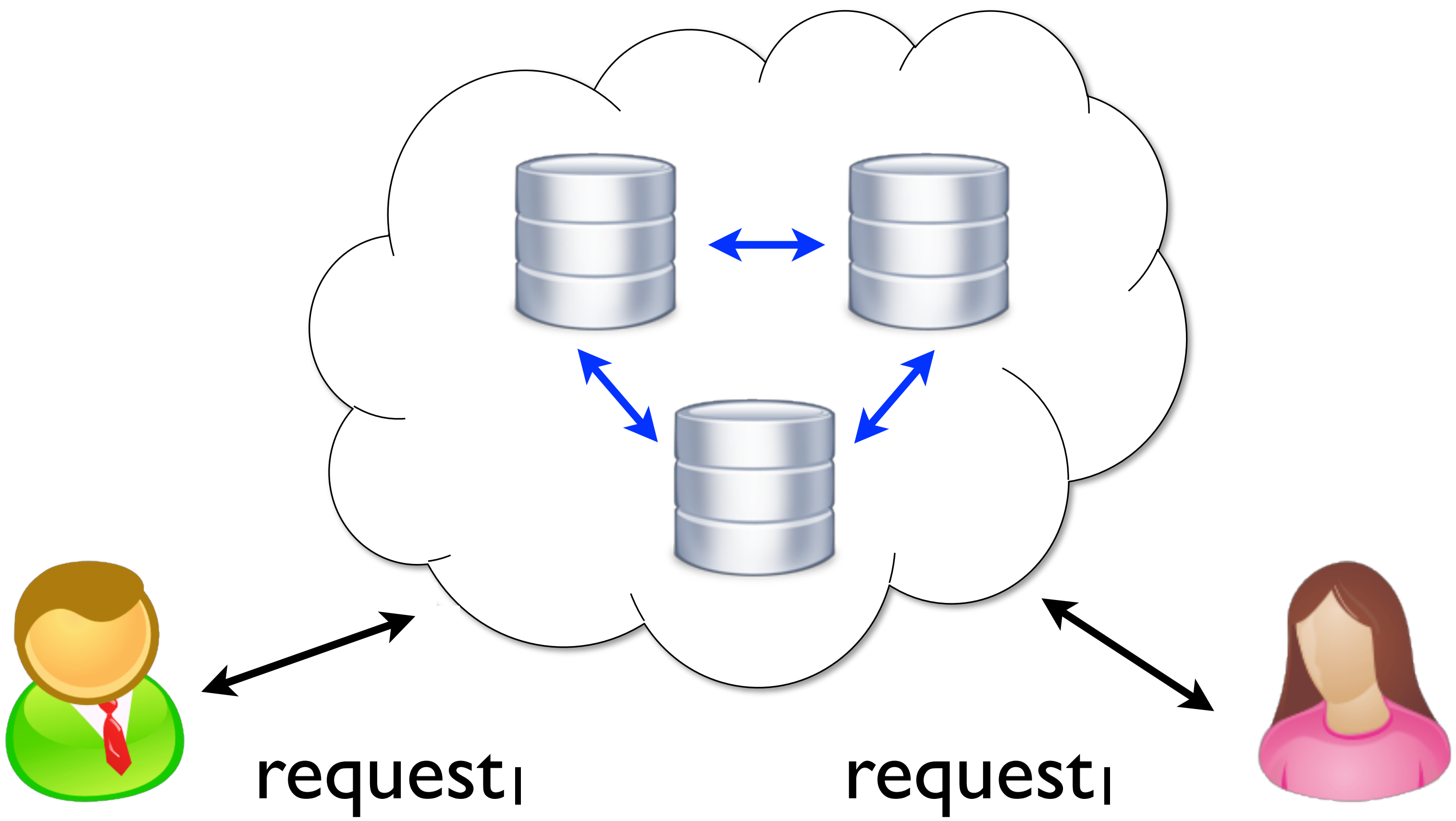
Consistency specification



request$_1$	request$_1$
response$_1$	response$_1$
request$_2$	request$_2$
response$_2$	response$_2$
...	...

event e

Consistency specification



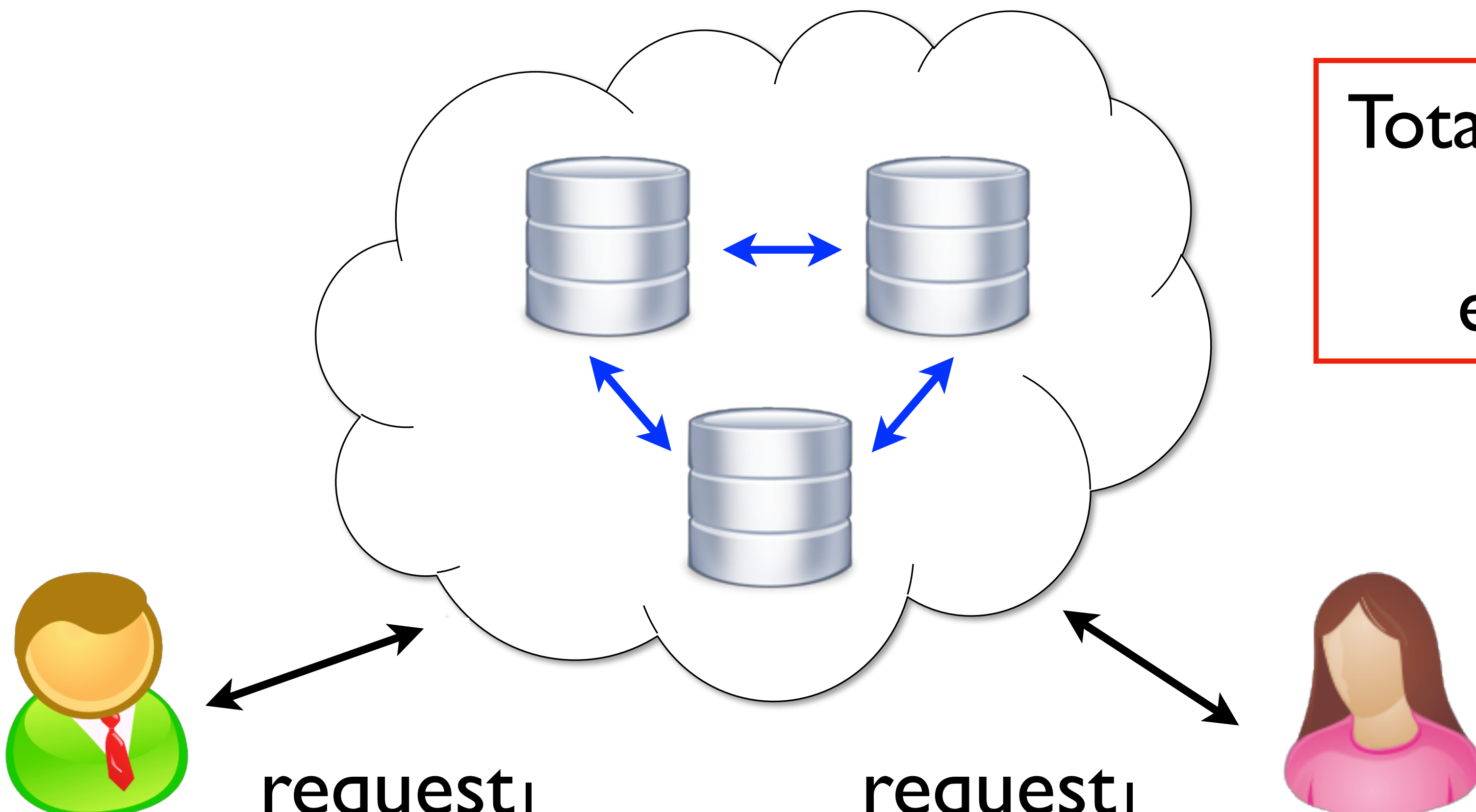
request$_1$	request$_1$
response$_1$	response$_1$
request$_2$	request$_2$
response$_2$	response$_2$
…	…

event e

session
(= process, thread)

Session order so: the order in which events are issued: union of total per-client total orders

Consistency specification



Total order: transitive and irreflexive relation ordering any pair of elements one way or another

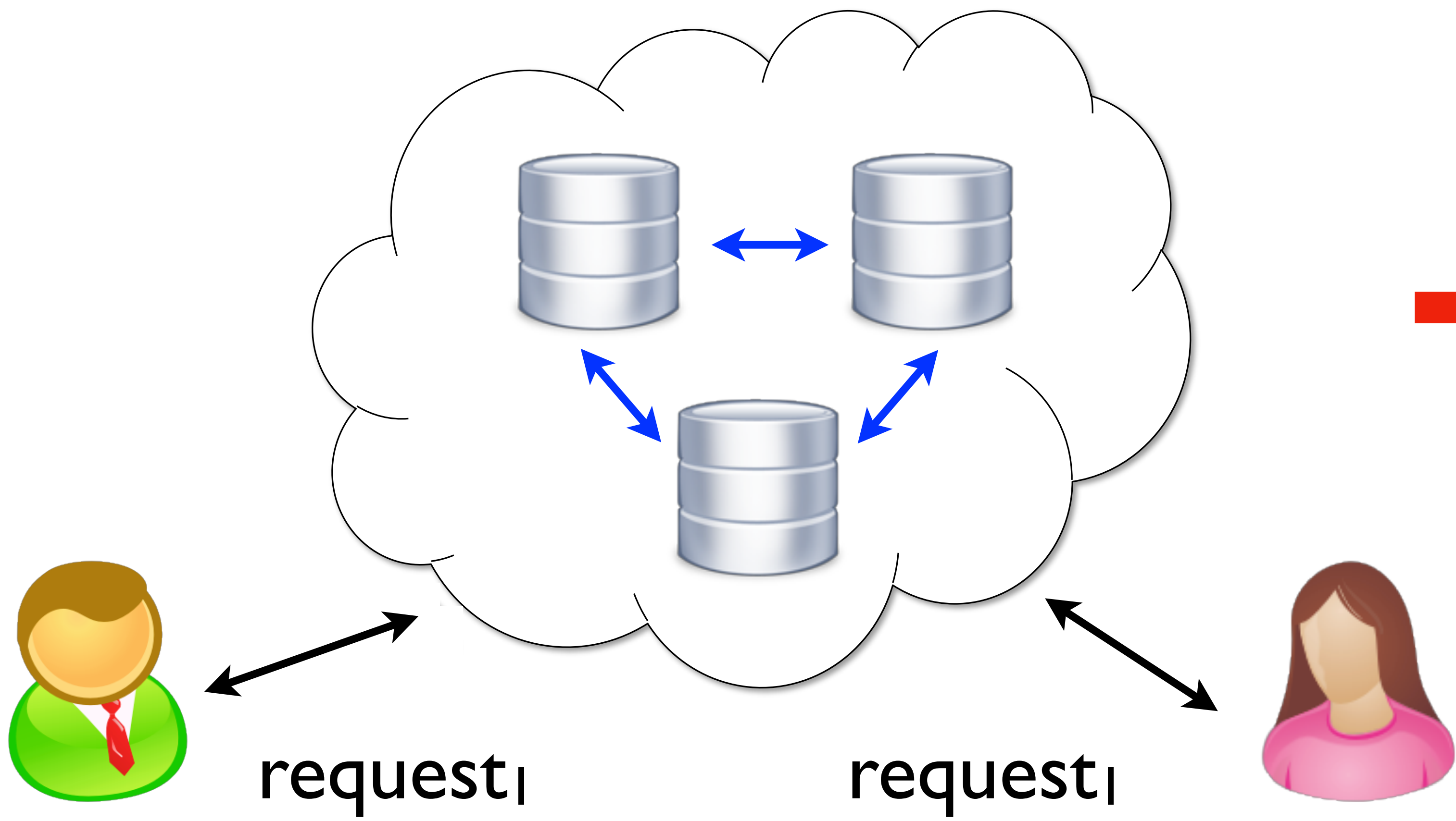
request$_1$	request$_1$
response$_1$	response$_1$
request$_2$	request$_2$
response$_2$	response$_2$
…	…

$\left.\begin{array}{l}\text{request}_2\\ \text{response}_2\end{array}\right\}$ event e

session
(= process, thread)

Session order so: the order in which events are issued: union of total per-client total orders

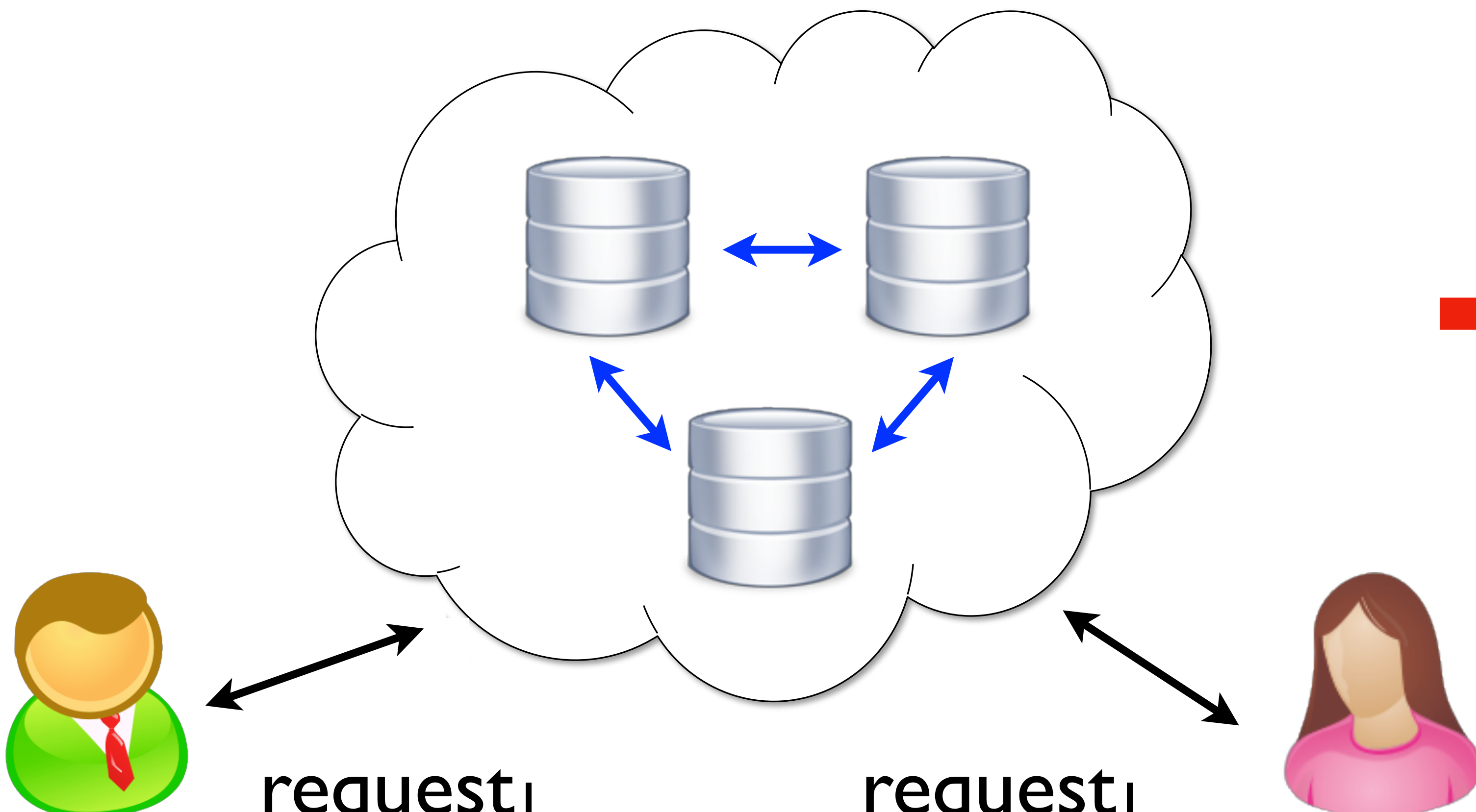
Consistency specification



request$_1$	request$_1$
response$_1$	response$_1$
request$_2$	request$_2$
response$_2$	response$_2$
…	…

➡ History $H = (E, so)$

Consistency specification



➡ History H = (E, so)

request₁ response₁ request₂ response₂ …

request₁ response₁ request₂ response₂ …

Consistency model - a set of histories $\mathscr{H}$:
the set of allowed database behaviours

Visualising histories

$$
\begin{tikzcd}
\text{x.read: 0} \arrow[d, "\text{so}"] & \text{x.write(1)} \arrow[d, "\text{so}"] \\
\text{y.write(1)} \arrow[d, "\text{so}"] & \text{c.add(1)} \arrow[d, "\text{so}"] \\
\text{z.write(2)} \arrow[d, "\text{so}"] & \text{c.read: 1} \arrow[d, "\text{so}"] \\
\text{c.add(1)} \arrow[d, "\text{so}"] & \text{z.read: 2} \\
\text{c.add(1)} &
\end{tikzcd}
$$

Visualising histories

$\text{x.read: 0} \xrightarrow{\text{so}} \text{y.write(1)} \xrightarrow{\text{so}} \text{z.write(2)} \xrightarrow{\text{so}} \text{c.add(1)} \xrightarrow{\text{so}} \text{c.add(1)}$

$\text{x.write(1)} \xrightarrow{\text{so}} \text{c.add(1)} \xrightarrow{\text{so}} \text{c.read: 1} \xrightarrow{\text{so}} \text{z.read: 2}$

Using a consistency model

- Consistency model $\mathscr{H}$: behaviour of the database under arbitrary clients

Using a consistency model

- Consistency model $\mathscr{H}$: behaviour of the database under arbitrary clients
- Program P ➔ set of all executions $[\![P]\!]$ under arbitrary behaviour of the database

Using a consistency model

- Consistency model $\mathscr{H}$: behaviour of the database under arbitrary clients
- Program P ➔ set of all executions $[\![P]\!]$ under arbitrary behaviour of the database
- Semantics of P when using $\mathscr{H}$:
  $[\![P, \mathscr{H}]\!] = \{X \in [\![P]\!] \mid \text{history}(X) \in \mathscr{H}\}$

Using a consistency model

- Consistency model $\mathscr{H}$: behaviour of the database under arbitrary clients
- Program P → set of all executions $[\![P]\!]$ under arbitrary behaviour of the database
- Semantics of P when using $\mathscr{H}$:
  $[\![P, \mathscr{H}]\!] = \{X \in [\![P]\!] \mid \text{history}(X) \in \mathscr{H}\}$

```
P:
r1 = x.read();
r2 = x.read();
y.write(r1==r2);
```

Using a consistency model

- Consistency model $\mathscr{H}$: behaviour of the database under arbitrary clients
- Program P → set of all executions $[\![P]\!]$ under arbitrary behaviour of the database
- Semantics of P when using $\mathscr{H}$:
  $[\![P, \mathscr{H}]\!] = \{X \in [\![P]\!] \mid \text{history}(X) \in \mathscr{H}\}$

P:	$[\![P]\!]$:	
r1 = x.read();	x.read(): 42;	x.read(): 42;
r2 = x.read();	x.read(): 42;	x.read(): 43;
y.write(r1==r2);	y.write(1);	y.write(0);

Using a consistency model

- Consistency model $\mathcal{H}$: behaviour of the database under arbitrary clients
- Program P → set of all executions $[\![P]\!]$ under arbitrary behaviour of the database
- Semantics of P when using $\mathcal{H}$:
  $[\![P, \mathcal{H}]\!] = \{X \in [\![P]\!] \mid \text{history}(X) \in \mathcal{H}\}$

$[\![P, \mathcal{H}]\!]$:

P:

```
r1 = x.read();
r2 = x.read();
y.write(r1==r2);
```

$[\![P]\!]$:

x.read(): 42;	~~x.read(): 42;~~
x.read(): 42;	~~x.read(): 43;~~
y.write(1);	~~y.write(0);~~

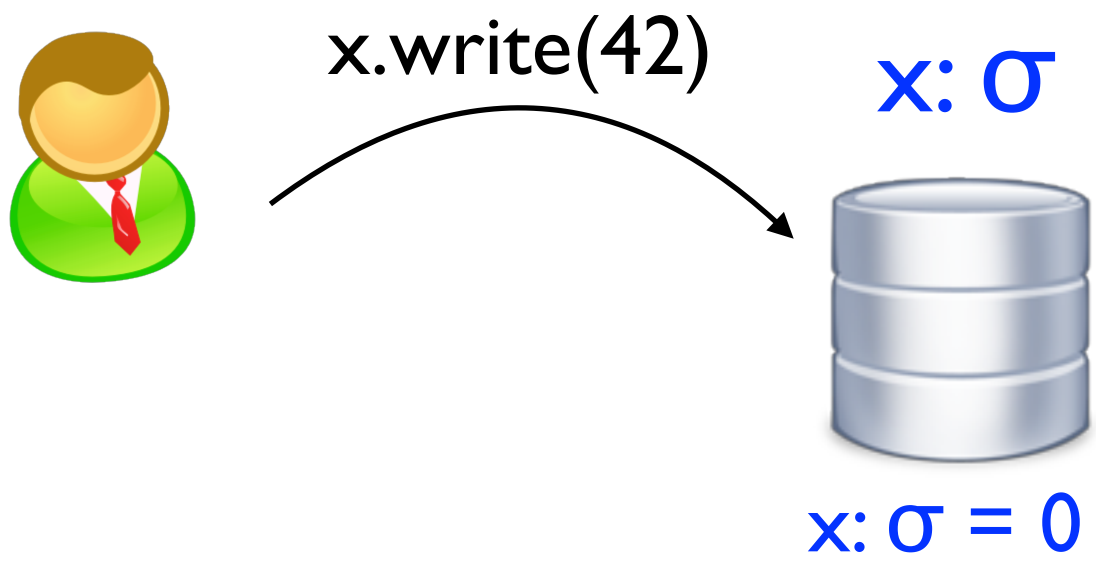
Defining a consistency model

- Operational specification: by an idealised implementation
- Axiomatic specification: more declarative

Strong consistency operationally

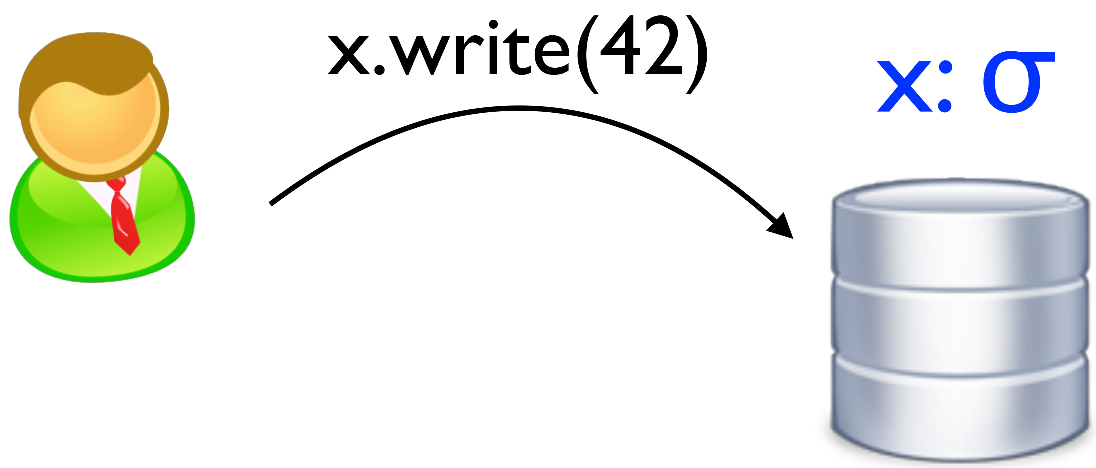
- Server with a single copy of all objects
- Clients send request to the server and wait for a reply
- Server processes operation sequentually in the receipt order

Strong consistency operationally



- Server with a single copy of all objects
- Clients send request to the server and wait for a reply
- Server processes operation sequentually in the receipt order

Strong consistency operationally



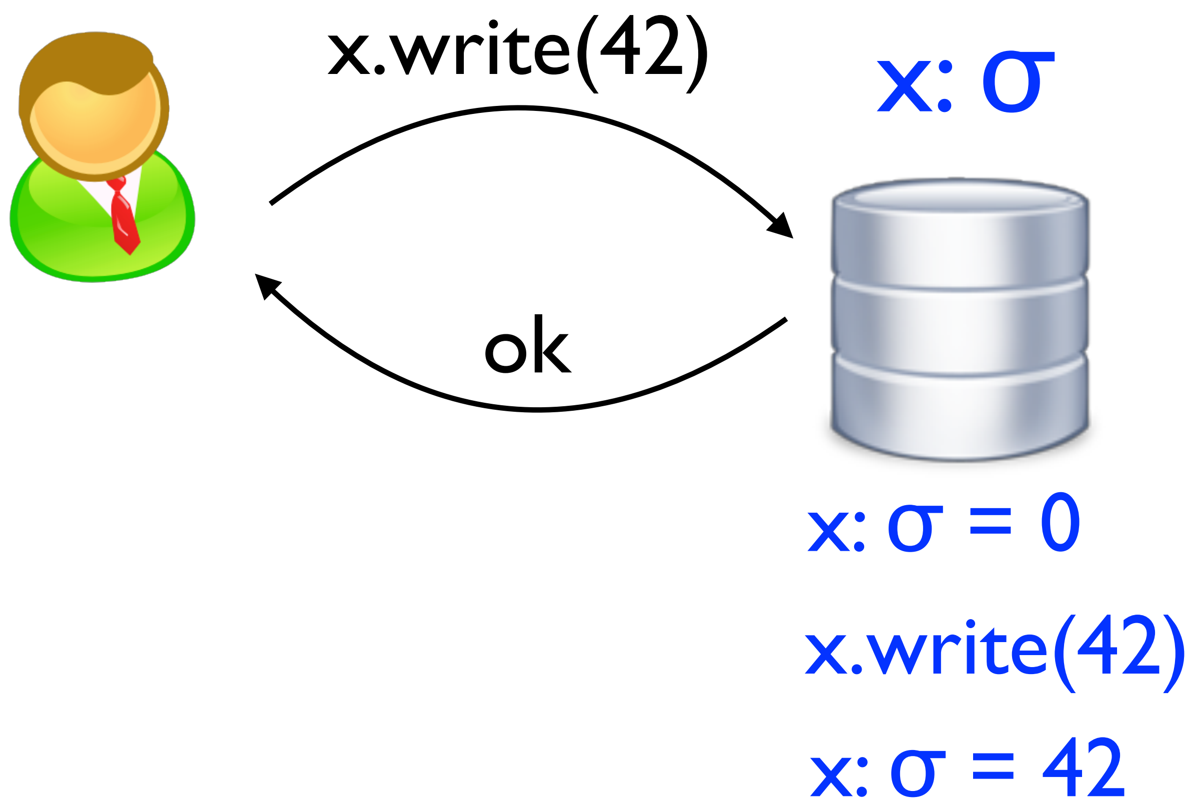
$x: \sigma = 0$

x.write(42)

$x: \sigma = 42$

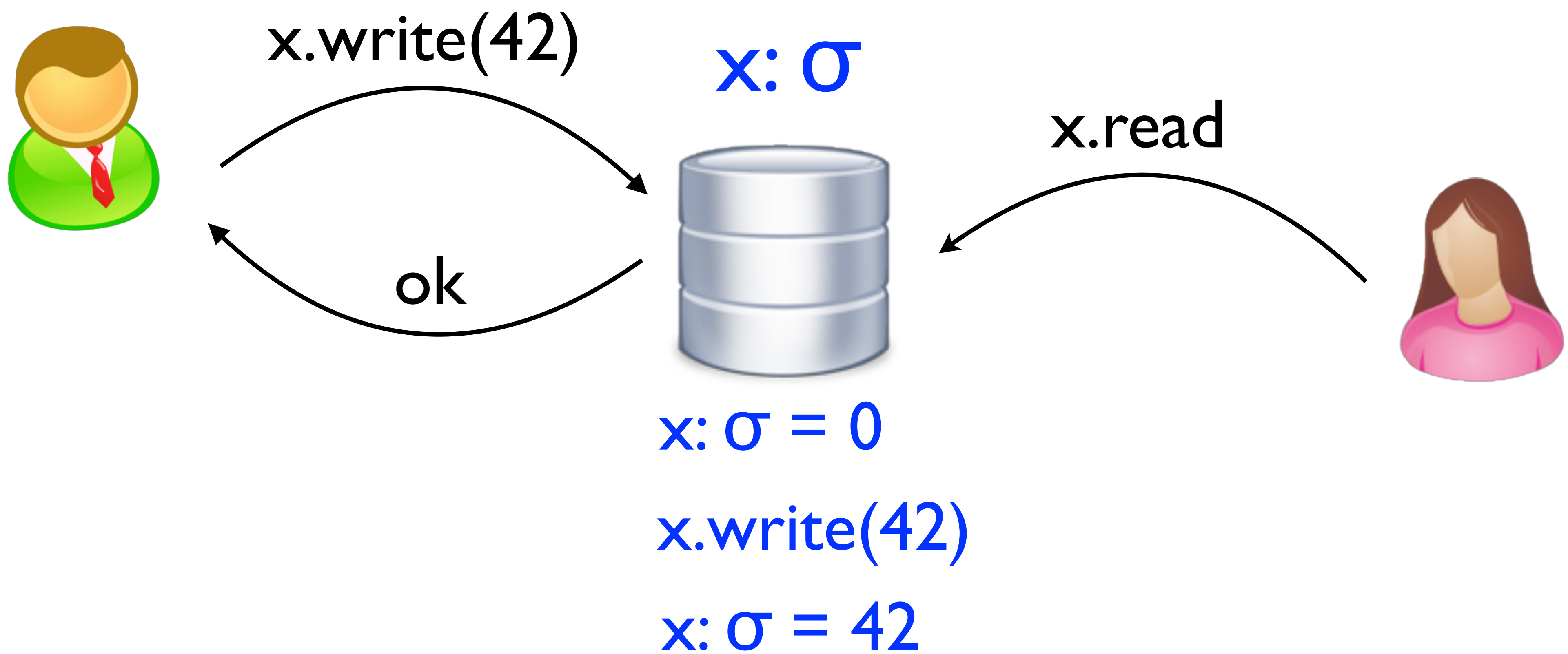
- Server with a single copy of all objects
- Clients send request to the server and wait for a reply
- Server processes operation sequentually in the receipt order

Strong consistency operationally



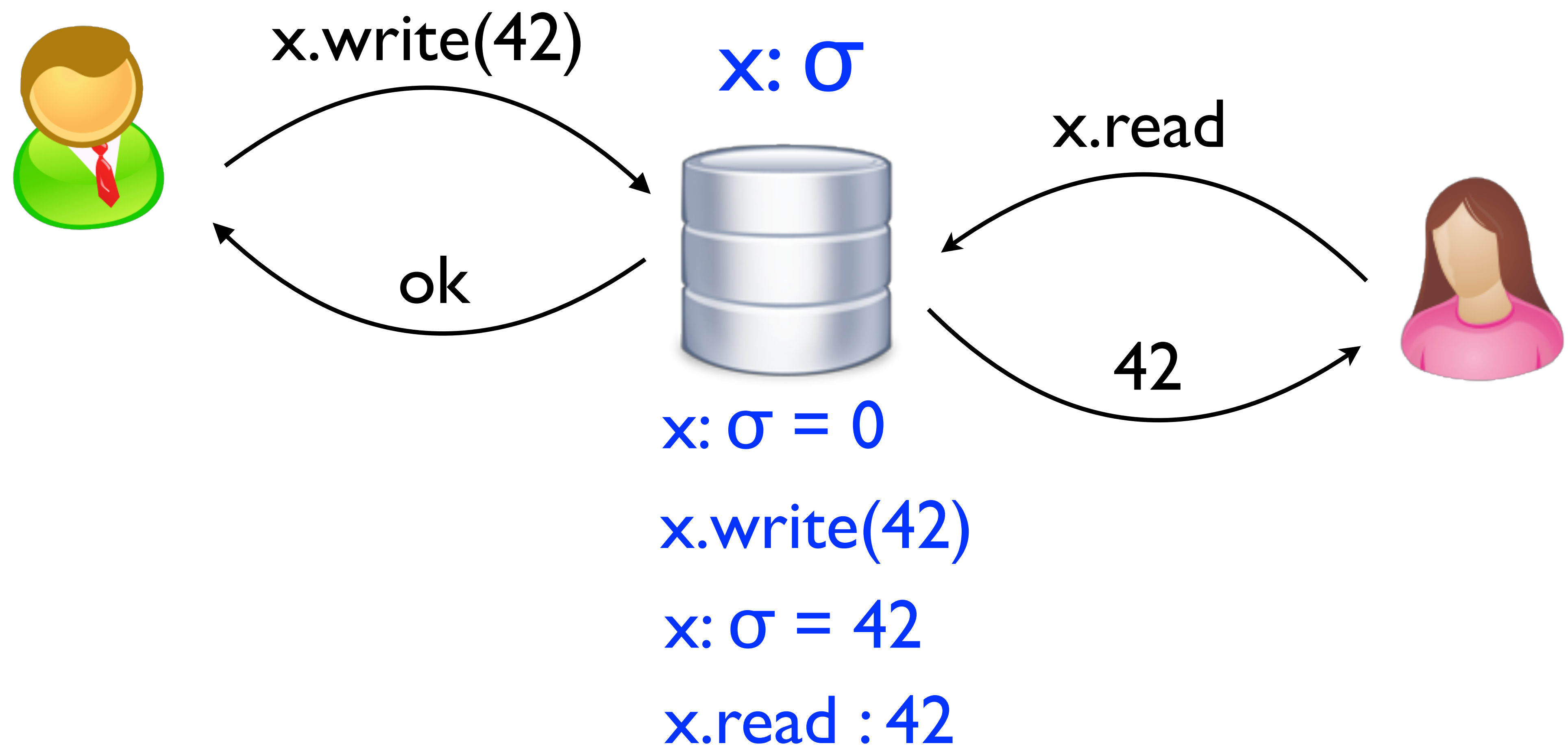
- Server with a single copy of all objects
- Clients send request to the server and wait for a reply
- Server processes operation sequentually in the receipt order

Strong consistency operationally



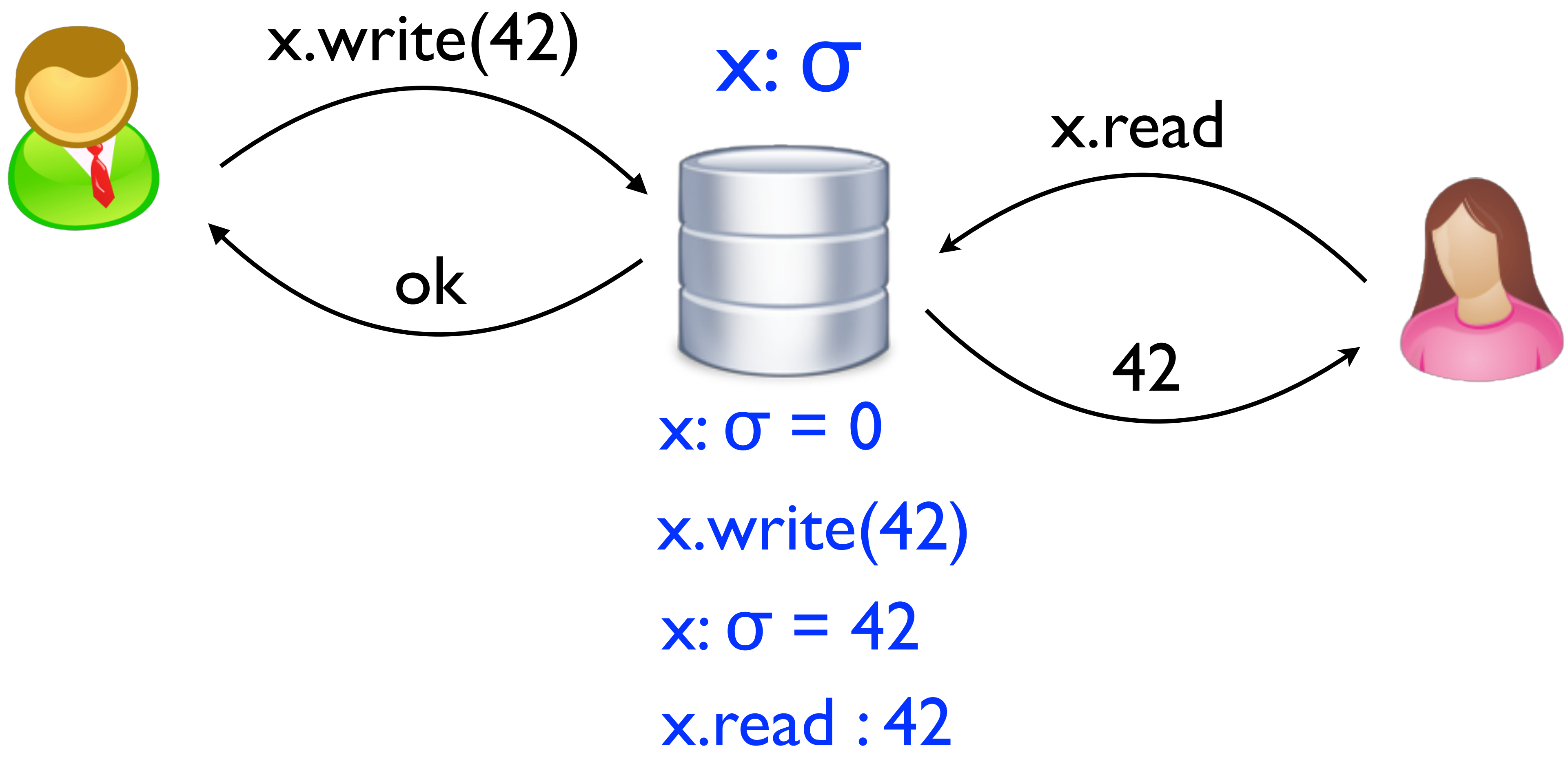
- Server with a single copy of all objects
- Clients send request to the server and wait for a reply
- Server processes operation sequentually in the receipt order

Strong consistency operationally



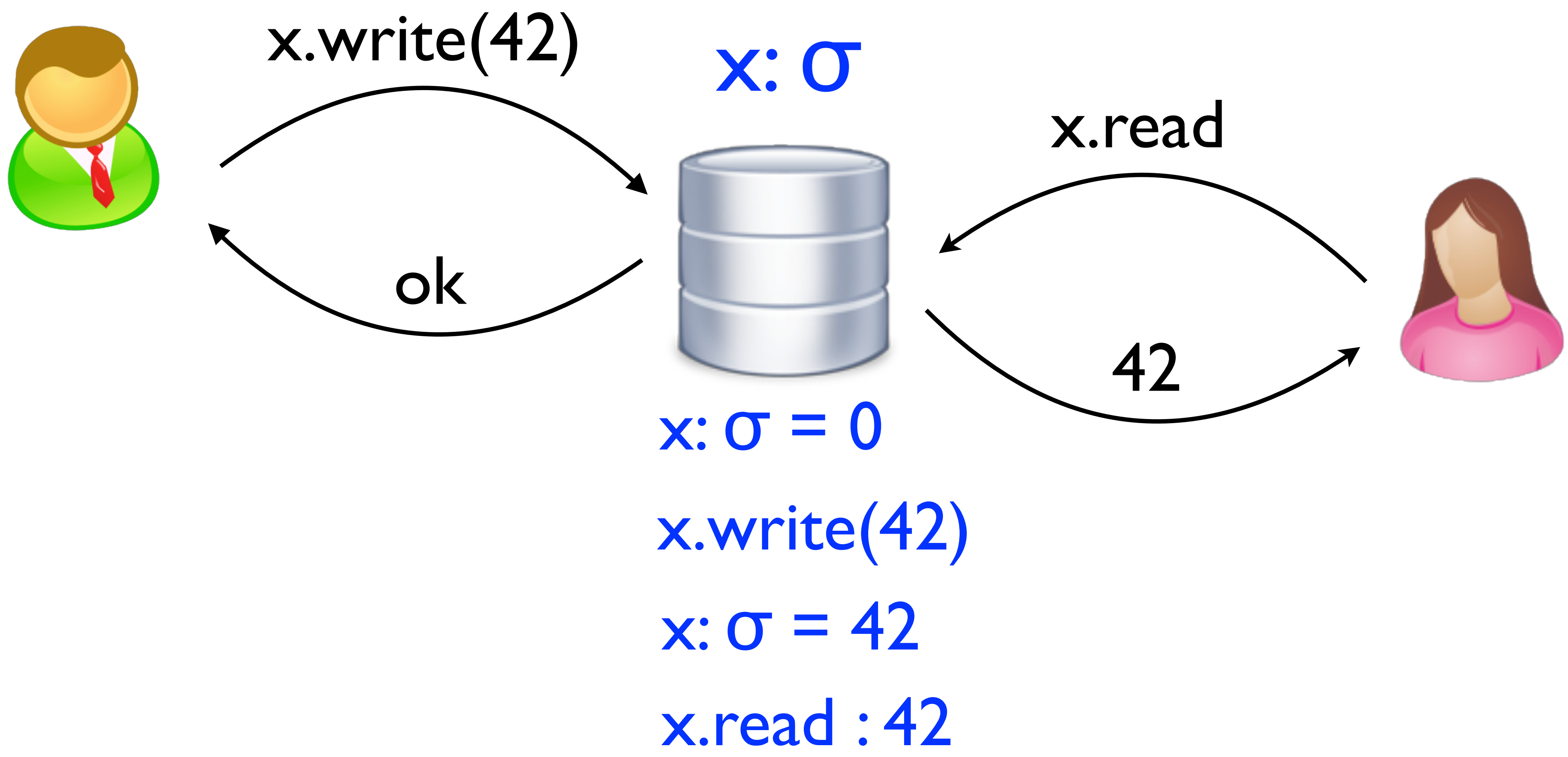
- Server with a single copy of all objects
- Clients send request to the server and wait for a reply
- Server processes operation sequentually in the receipt order

Strong consistency operationally



Could write a formal operational semantics: maintain the state of the database, clients and sets of messages between them

Strong consistency operationally



- Consistency model = {H | ∃ execution with history H produced by the abstract implementation}
- Sequential consistency: one form of strong consistency
- Weaker than linearizability: takes into acount the duration of operations

Operational specifications

- Let one understand intuitions behind implementations
- May become unwieldy for weaker consistency models
- Sometimes overspecify behaviour

Axiomatic specifications

- Choose a set of relations over events: $r_1, \ldots, r_n$

  Abstractly specify essential information about how operations are processed inside the system

- Abstract execution $(H, r_1, \ldots, r_n) = (E, so, r_1, \ldots, r_n)$

- Choose a set of axioms $\mathscr{A}$ constraining abstract executions

- Consistency model = $\{H \mid \exists\, r_1, \ldots, r_n.\ (H, r_1, \ldots, r_n) \models \mathscr{A}\}$

Axiomatic specifications

- Choose a set of relations over events: $r_1, \ldots, r_n$

  Abstractly specify essential information about how operations are processed inside the system

- Abstract execution $(H, r_1, \ldots, r_n) = (E, so, r_1, \ldots, r_n)$

- Choose a set of axioms $\mathscr{A}$ constraining abstract executions

- Consistency model = $\{H \mid \exists\, r_1, \ldots, r_n.\ (H, r_1, \ldots, r_n) \models \mathscr{A}\}$

vs $\boxed{\text{Consistency model} = \{H \mid \exists \text{ execution with history } H \text{ produced by the abstract implementation}\}}$

Axiomatic specifications

- Choose a set of relations over events: $r_1, \ldots, r_n$

  Abstractly specify essential information about how operations are processed inside the system

- Abstract execution $(H, r_1, \ldots, r_n) = (E, so, r_1, \ldots, r_n)$

- Choose a set of axioms $\mathscr{A}$ constraining abstract executions

- Consistency model = $\{H \mid \exists\, r_1, \ldots, r_n.\ (H, r_1, \ldots, r_n) \models \mathscr{A}\}$

vs

Consistency model = {H | ∃ execution with history H produced by the abstract implementation}

Axiomatic specifications

- Choose a set of relations over events: $r_1, \ldots, r_n$

  Abstractly specify essential information about how operations are processed inside the system

- Abstract execution $(H, r_1, \ldots, r_n) = (E, so, r_1, \ldots, r_n)$

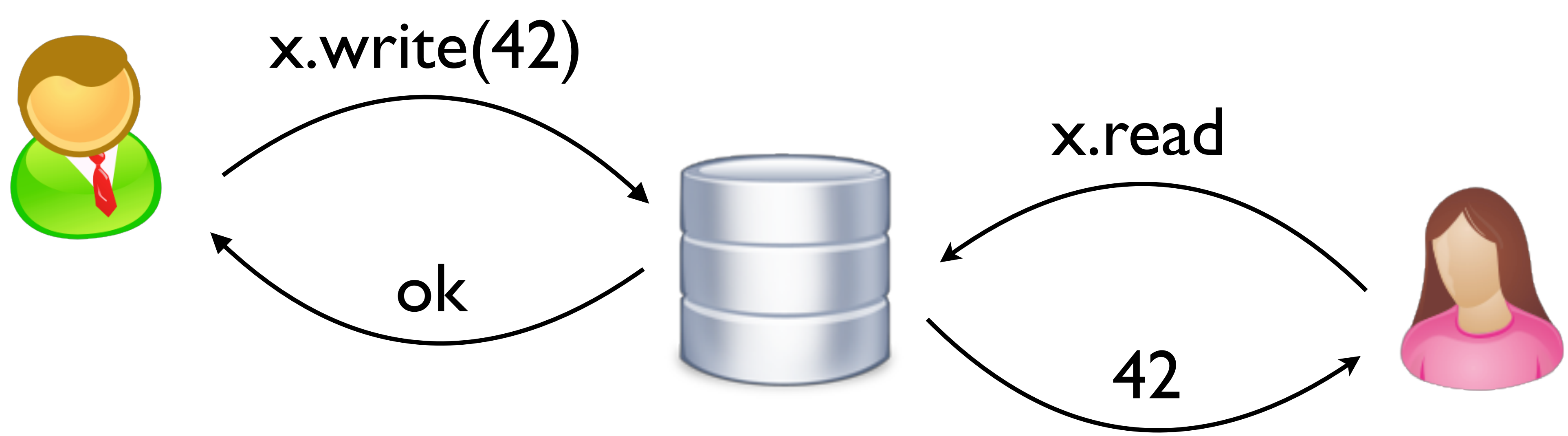
- Choose a set of axioms $\mathscr{A}$ constraining abstract executions

- Consistency model = $\{H \mid \exists\, r_1, \ldots, r_n.\ (H, r_1, \ldots, r_n) \models \mathscr{A}\}$

vs $\boxed{\text{Consistency model} = \{H \mid \exists \text{ execution with history } H \text{ produced by the abstract implementation}\}}$

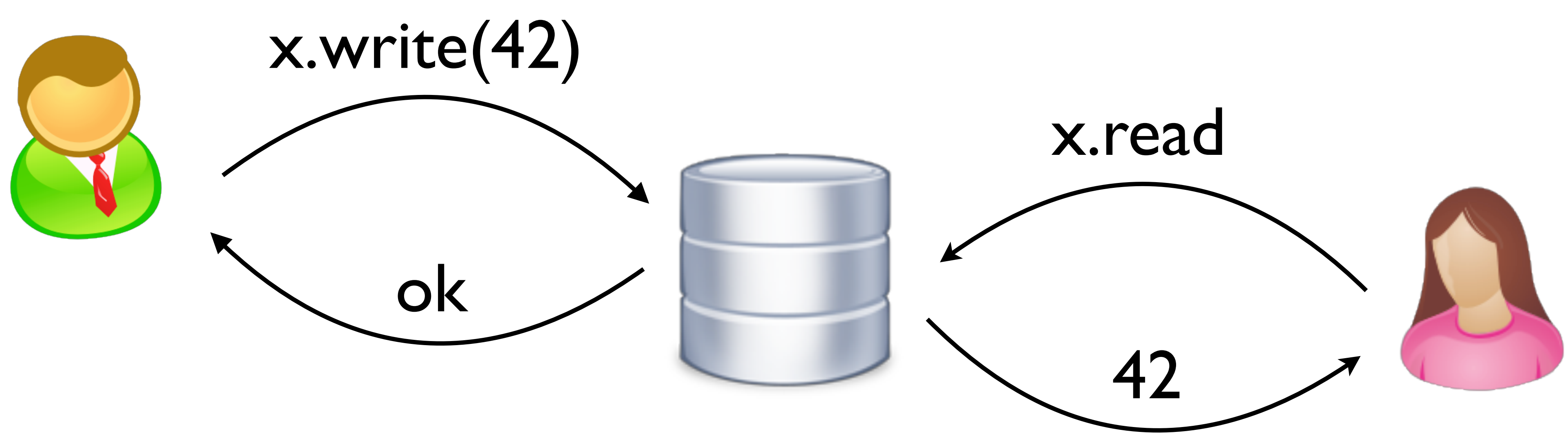
Sequential consistency axiomatically

An SC history can be explained by a total order over all events: the order in which the server processes client operations



Sequential consistency axiomatically

An SC history can be explained by a total order over all events: the order in which the server processes client operations



Abstract execution: (H, to) = (E, so, to), where to $\subseteq$ E×E

SC = {(E, so) | $\exists$ total order to. (E, so, to) $\models \mathcal{A}_{\mathrm{SC}}$}

$(E, so, to) \vDash \mathscr{A}_{SC}$ iff

1. $so \subseteq to$
2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

$(E, so, to) \models \mathscr{A}_{SC}$ iff

1. $so \subseteq to$

2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

   $\forall e \in E.\, type(obj(e)) = (\sigma_0, [\![-]\!]_{val}, [\![-]\!]_{state})$

$(E, so, to) \vDash \mathscr{A}_{SC}$ iff

1. $so \subseteq to$

2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

   $\forall e \in E.\ type(obj(e)) = (\sigma_0, [\![-]\!]_{val}, [\![-]\!]_{state})$

   $rval(e) = [\![op(e)]\!]_{val}(\sigma)$

$(E, so, to) \vDash \mathscr{A}_{SC}$ iff

1. $so \subseteq to$

2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

$\forall e \in E.\ type(obj(e)) = (\sigma_0, [\![-]\!]_{val}, [\![-]\!]_{state})$

$rval(e) = [\![op(e)]\!]_{val}(\sigma)$

$\sigma = [\![op(e_n)]\!]_{state}(\ldots[\![op(e_1)]\!]_{state}(\sigma_0))$

$e_1, \ldots, e_n = to^{-1}(e).select(obj(e)).sort(to)$

$(E, so, to) \models \mathscr{A}_{SC}$ iff

1. $so \subseteq to$
2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

$\forall e \in E.\ type(obj(e)) = (\sigma_0, [\![-]\!]_{val}, [\![-]\!]_{state})$

$rval(e) = [\![op(e)]\!]_{val}(\sigma)$

$\sigma = [\![op(e_n)]\!]_{state}(\ldots[\![op(e_1)]\!]_{state}(\sigma_0))$

$e_1, \ldots, e_n = to^{-1}(e).select(obj(e)).sort(to)$

Integer registers: a read returns the value written by the last preceding event in to (or 0 if there are none)

x.write(0); x.write(42); x.read: 42

$(E, so, to) \models \mathscr{A}_{SC}$ iff

$\boxed{SC = \{(E, so) \mid \exists to.\ (E, so, to) \models \mathscr{A}_{SC}\}}$

1. $so \subseteq to$
2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

$\forall e \in E.\ type(obj(e)) = (\sigma_0, [\![-]\!]_{val}, [\![-]\!]_{state})$

$rval(e) = [\![op(e)]\!]_{val}(\sigma)$

$\sigma = [\![op(e_n)]\!]_{state}(\ldots[\![op(e_1)]\!]_{state}(\sigma_0))$

$e_1, \ldots, e_n = to^{-1}(e).select(obj(e)).sort(to)$

Integer registers: a read returns the value written by the last preceding event in to (or 0 if there are none)

x.write(0); x.write(42); x.read: 42

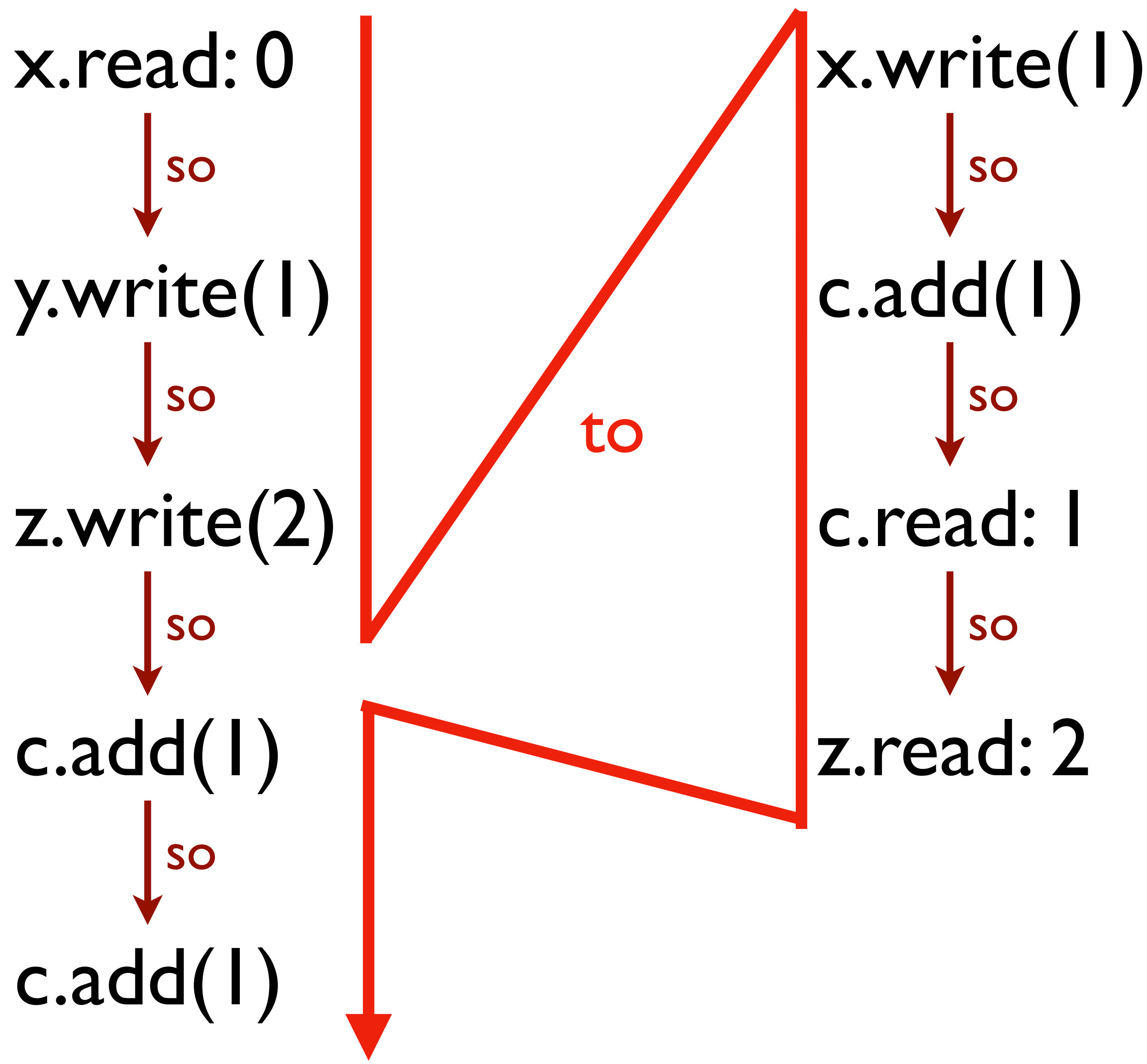
SC example

$$\boxed{SC = \{(E, so) \mid \exists to.\ (E, so, to) \models \mathscr{A}_{SC}\}}$$

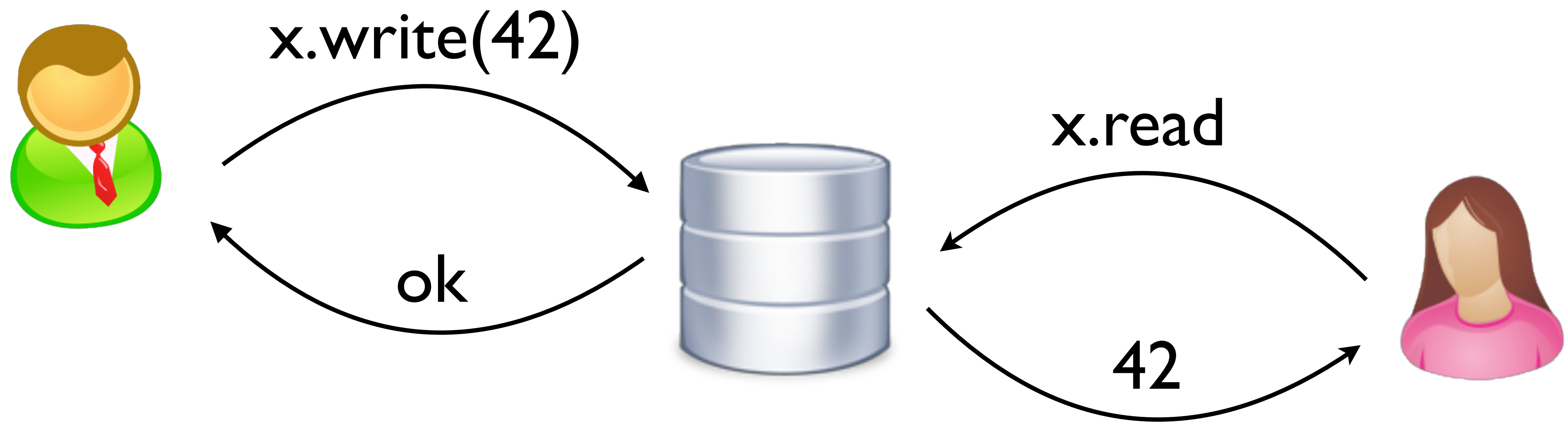
$$\begin{tikzcd}
\text{x.read: 0} \arrow[d, "so"] & \text{x.write(1)} \arrow[d, "so"] \\
\text{y.write(1)} \arrow[d, "so"] & \text{c.add(1)} \arrow[d, "so"] \\
\text{z.write(2)} \arrow[d, "so"] & \text{c.read: 1} \arrow[d, "so"] \\
\text{c.add(1)} \arrow[d, "so"] & \text{z.read: 2} \\
\text{c.add(1)} &
\end{tikzcd}$$

SC example

$$\boxed{SC = \{(E, so) \mid \exists to.\ (E, so, to) \models \mathscr{A}_{SC}\}}$$



Operational vs axiomatic



- Got rid of messages between clients and the server, but didn't go far from the operational spec
- There's more difference for weaker models: complex processing can be concisely specified by axioms

Process A:

```
x.write(1)
if (y.read() == 0)
  print "A wins"
```

Process B:

```
y.write(1)
if (x.read() == 0)
  print "B wins"
```

Dekker example

Process A:

```
x.write(1)
if (y.read() == 0)
  print "A wins"
```

Process B:

```
y.write(1)
if (x.read() == 0)
  print "B wins"
```

Claim: under sequential consistency, there can be at most one winner

```
Process A:                    Process B:

x.write(1)                    y.write(1)
if (y.read() == 0)            if (x.read() == 0)
  print "A wins"                print "B wins"
```

Assume there are two winners. Then there must exist an abstract execution for the history:

$$
\begin{tikzcd}
\text{x.write(1)} \arrow[d, "so"] & \text{y.write(1)} \arrow[d, "so"] \\
\text{y.read(): 0} & \text{x.read(): 0}
\end{tikzcd}
$$

Need to construct a total order to

```
Process A:                  Process B:

x.write(1)                  y.write(1)
if (y.read() == 0)          if (x.read() == 0)
  print "A wins"              print "B wins"
```

Assume there are two winners. Then there must exist an abstract execution for the history:

$$
\begin{tikzcd}
x.\mathrm{write}(1) \arrow[d, "\mathrm{so,\ to}"] & y.\mathrm{write}(1) \arrow[d, "\mathrm{so,\ to}"] \\
y.\mathrm{read}():0 & x.\mathrm{read}():0
\end{tikzcd}
$$

$\mathrm{so} \subseteq \mathrm{to}$

```
Process A:                    Process B:

x.write(1)                    y.write(1)
if (y.read() == 0)            if (x.read() == 0)
  print "A wins"                print "B wins"
```

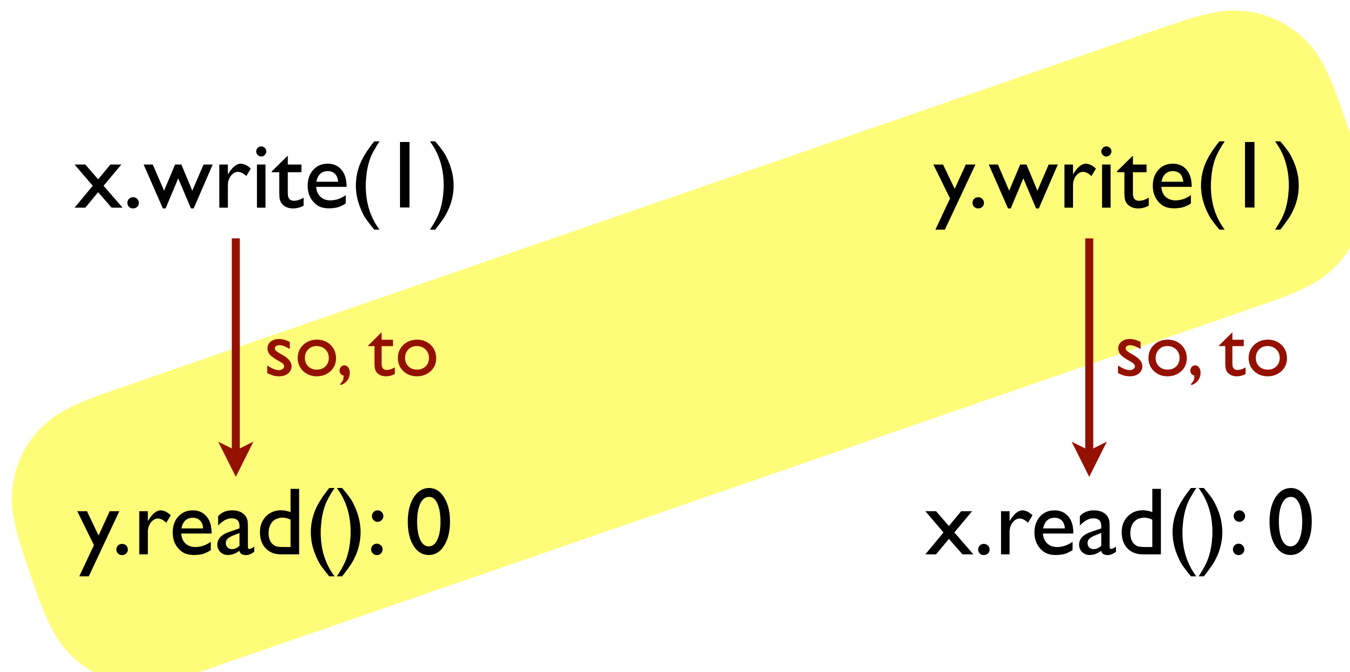
Assume there are two winners. Then there must exist an abstract execution for the history:

$$\begin{tikzcd} \text{x.write(1)} \arrow[d, "\text{so, to}"] & \text{y.write(1)} \arrow[d, "\text{so, to}"] \\ \text{y.read(): 0} & \text{x.read(): 0} \end{tikzcd}$$

```
Process A:                    Process B:

x.write(1)                    y.write(1)
if (y.read() == 0)            if (x.read() == 0)
  print "A wins"                print "B wins"
```

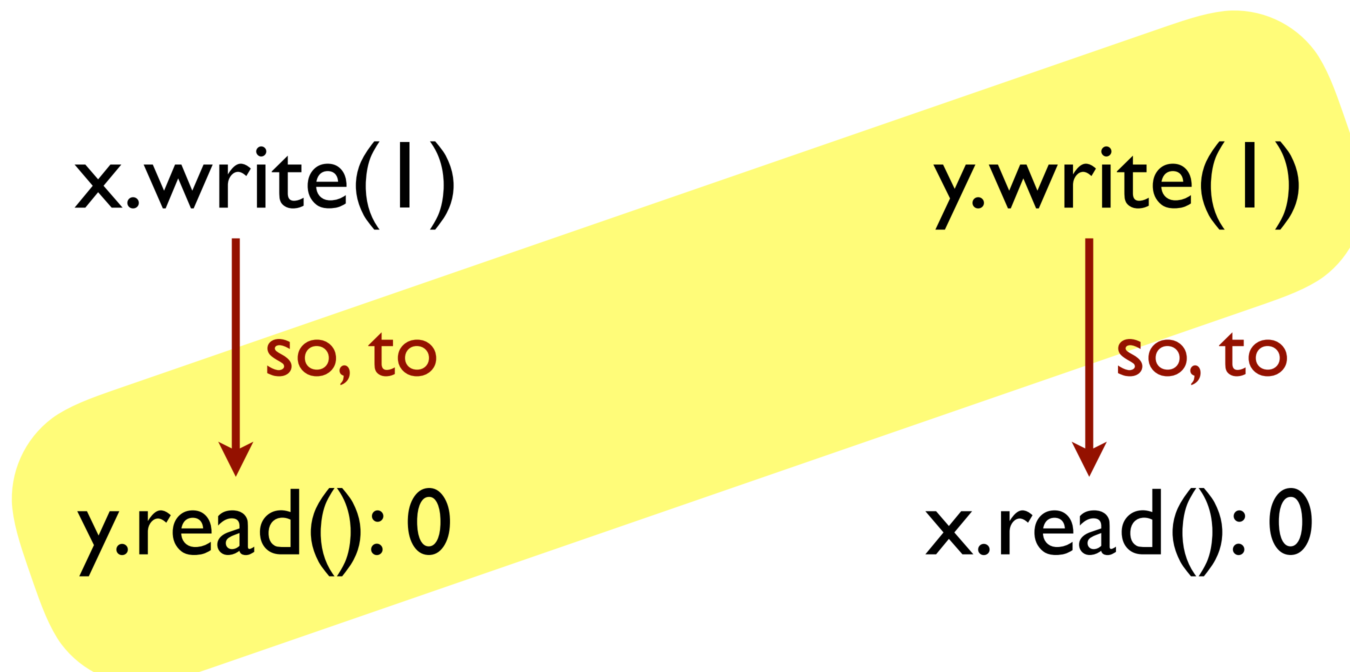
Assume there are two winners. Then there must exist an abstract execution for the history:



Process A:	Process B:

x.write(1)
if (y.read() == 0)
  print "A wins"
	y.write(1)
	if (x.read() == 0)
	  print "B wins"

Assume there are two winners. Then there must exist an abstract execution for the history:

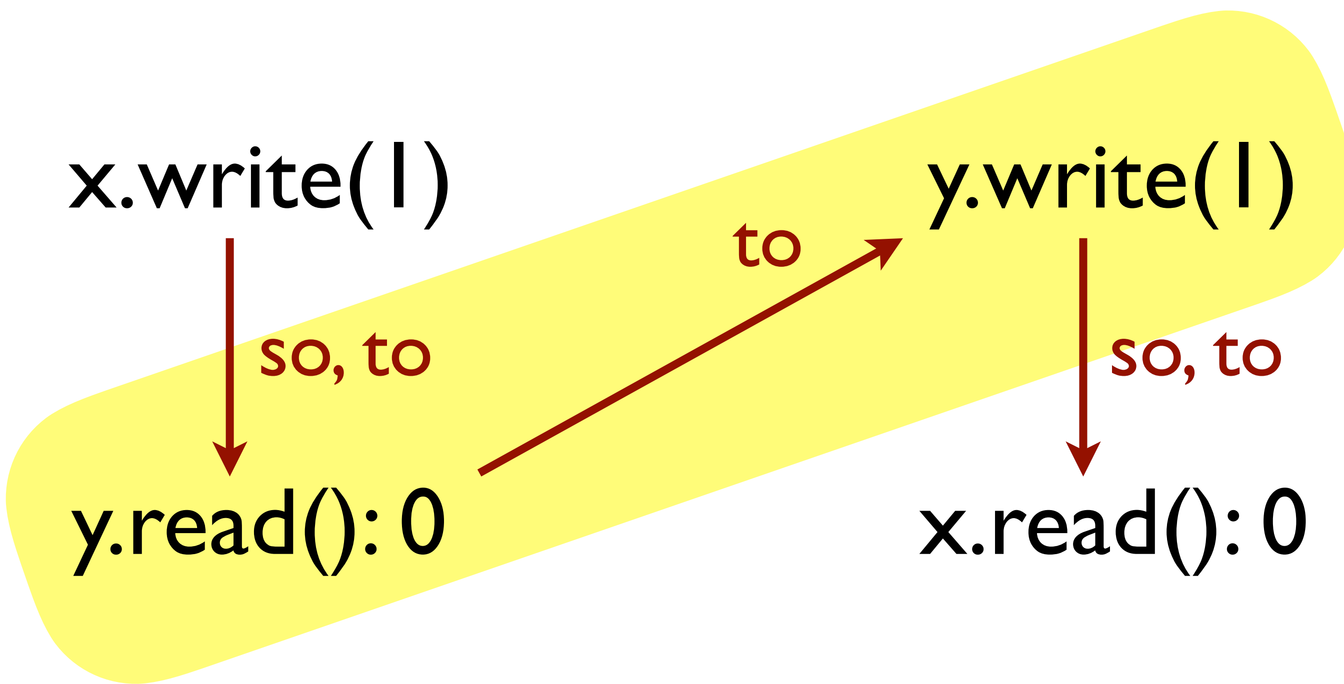


Reads return the most recent write in to, but this read doesn't see the write

```
Process A:                   Process B:

x.write(1)                   y.write(1)
if (y.read() == 0)           if (x.read() == 0)
  print "A wins"               print "B wins"
```

Assume there are two winners. Then there must exist an abstract execution for the history:

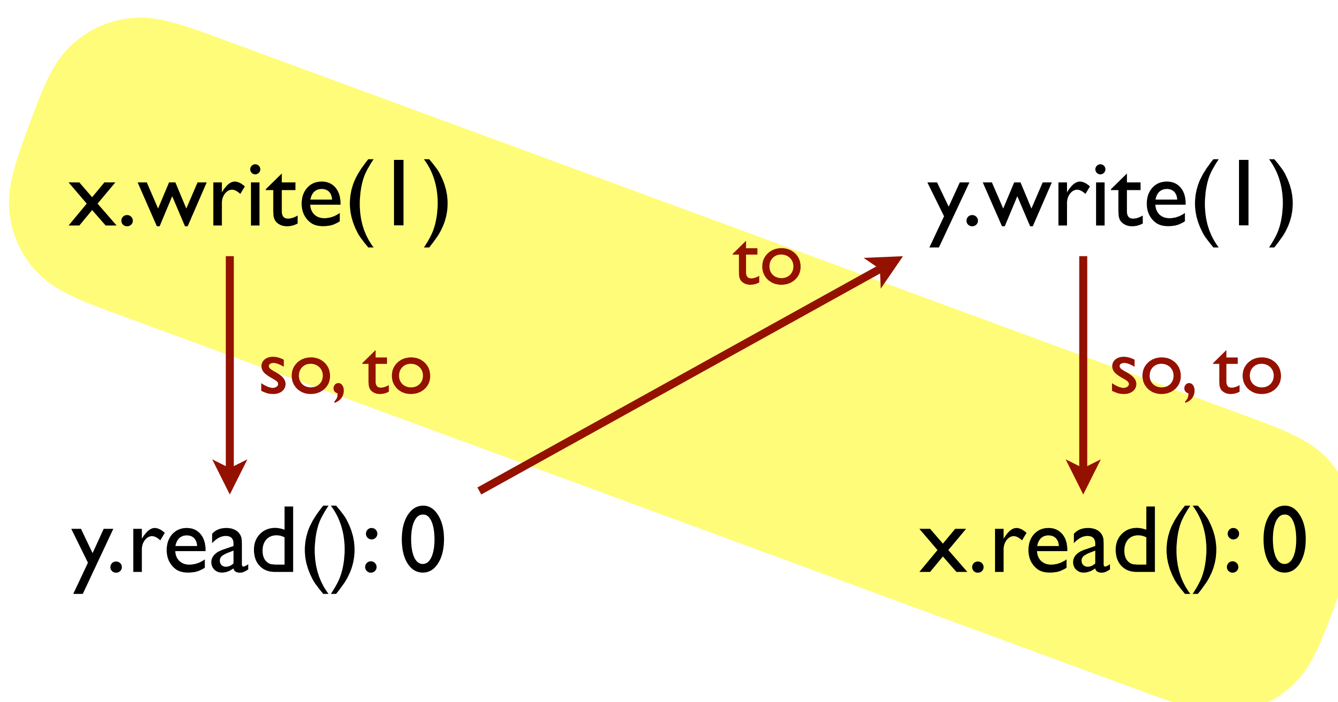


Reads return the most recent write in to, but this read doesn't see the write

```
Process A:                  Process B:

x.write(1)                  y.write(1)
if (y.read() == 0)          if (x.read() == 0)
  print "A wins"              print "B wins"
```

Assume there are two winners. Then there must exist an abstract execution for the history:

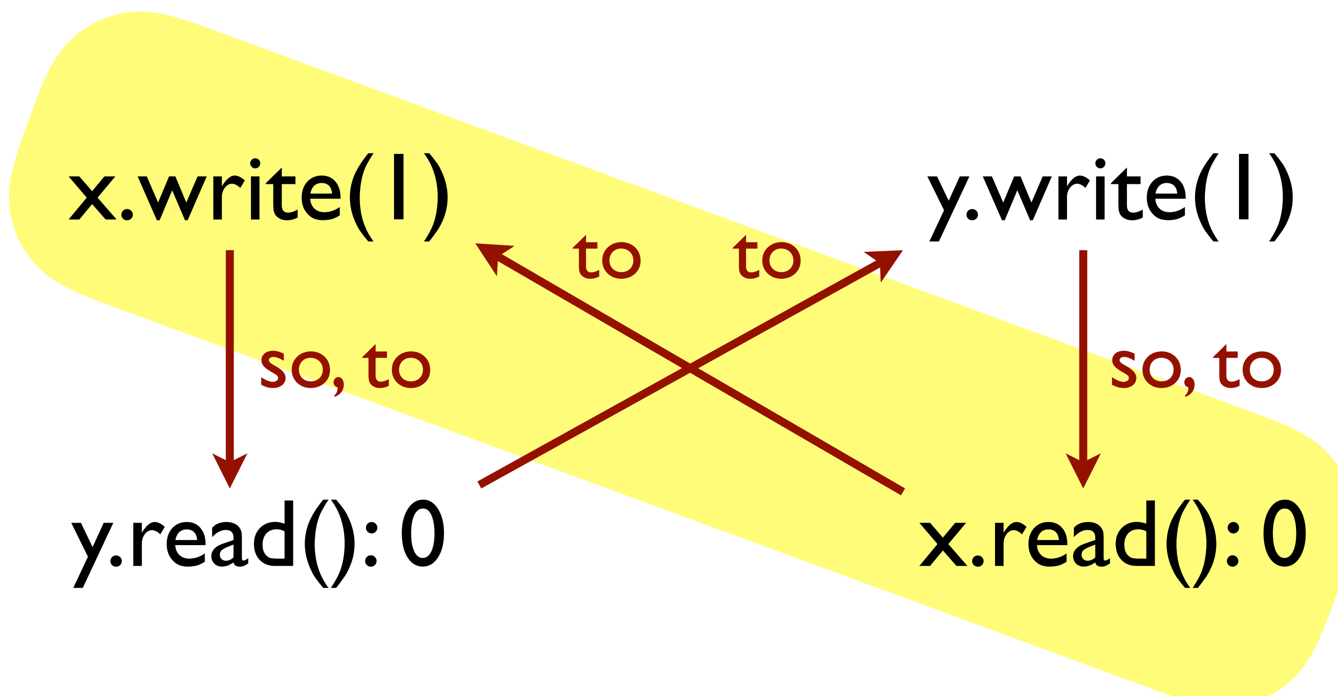


Reads return the most recent write in to, but this read doesn't see the write

```
Process A:                  Process B:

x.write(1)                  y.write(1)
if (y.read() == 0)          if (x.read() == 0)
  print "A wins"              print "B wins"
```

Assume there are two winners. Then there must exist an abstract execution for the history:

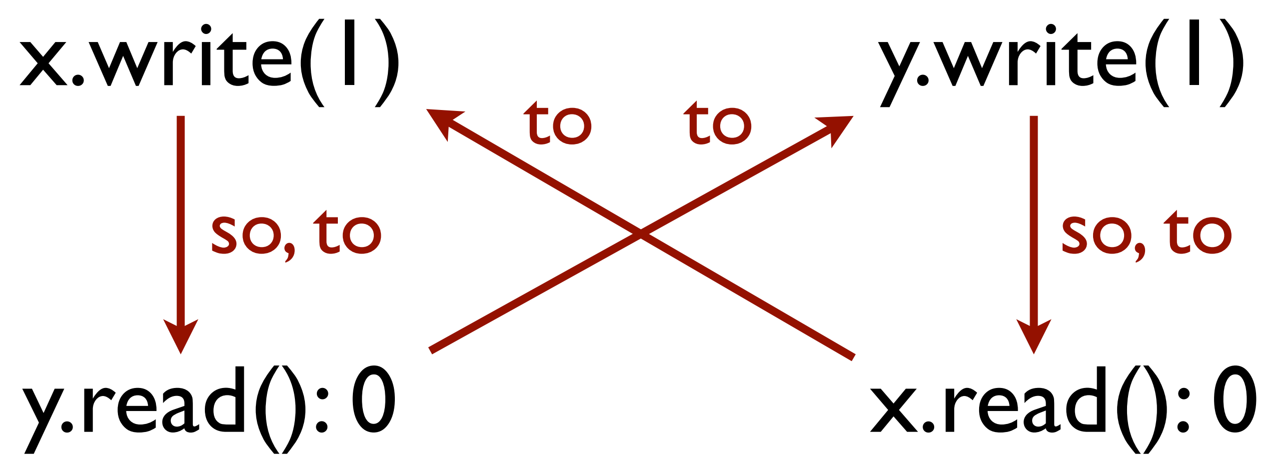


Reads return the most recent write in to, but this read doesn't see the write

```
Process A:                Process B:

x.write(1)                y.write(1)
if (y.read() == 0)        if (x.read() == 0)
  print "A wins"            print "B wins"
```

Assume there are two winners. Then there must exist an abstract execution for the history:



But to must be acyclic, so no such total order exists - QED.

CAP theorem

No system with at least 2 processes can implement a read-write register with strong consistency, availability, and partition tolerance

- strong consistency = sequential consistency
- availability = all operations eventually complete
- partition tolerance = system continues to function under permanent network partitions

  (processes in different partitions can no longer communicate in any way)

CAP proof

No system with at least 2 processes can implement a read-write register with strong consistency, availability, and partition tolerance

- By contradiction: assume the desired system exists
- Run some experiments with the Dekker program
- Network is partitioned between the two processes

Process A:	Process B:

x.write(1)
if (y.read() == 0)
  print "A wins"
	y.write(1)
	if (x.read() == 0)
	  print "B wins"

Process A	Process B
x.write(1) if (y.read() == 0) print "A wins"	

- Process A runs its code, process B is idle

execution X_A of process A

Process A

```
x.write(1)
if (y.read() == 0)
  print "A wins"
```

Process B

- Process A runs its code, process B is idle
- Availability $\Longrightarrow$ A must terminate and produce an execution X_A

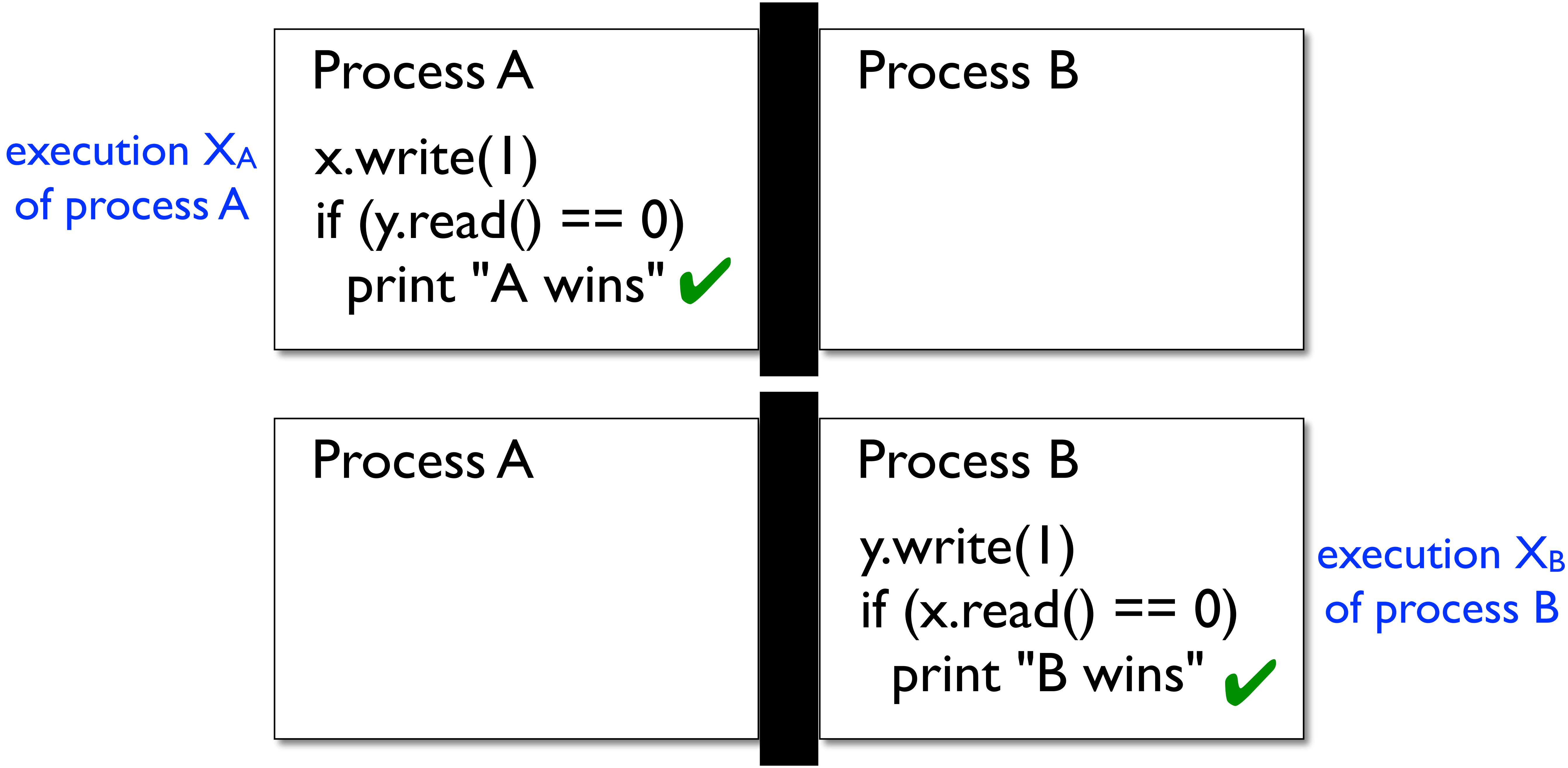
execution X_A
of process A

Process A

```
x.write(1)
if (y.read() == 0)
  print "A wins" ✔
```

Process B

- Process A runs its code, process B is idle
- Availability $\Longrightarrow$ A must terminate and produce an execution X_A
- Sequential consistency $\Longrightarrow$ X_A must print "A wins"



- Process B runs its code, process A is idle
- Availability $\Longrightarrow$ B must terminate and produce an execution X_B
- Sequential consistency $\Longrightarrow$ X_B must print "B wins"

execution X_A of process A

Process A

```
x.write(1)
if (y.read() == 0)
  print "A wins" ✔
```

Process B

```
y.write(1)
if (x.read() == 0)
  print "B wins" ✔
```

execution X_B of process B

- Network is partitioned in both experiments: processes didn't receive any messages
- $X_A; X_B$ is an execution of A || B, i.e., Dekker
- $X_A; X_B$ not SC $\Longrightarrow$ contradiction, QED

execution X_A of process A

Process A

```
x.write(1)
if (y.read() == 0)
  print "A wins" ✔
```

Process B

```
y.write(1)
if (x.read() == 0)
  print "B wins" ✔
```

execution X_B of process B

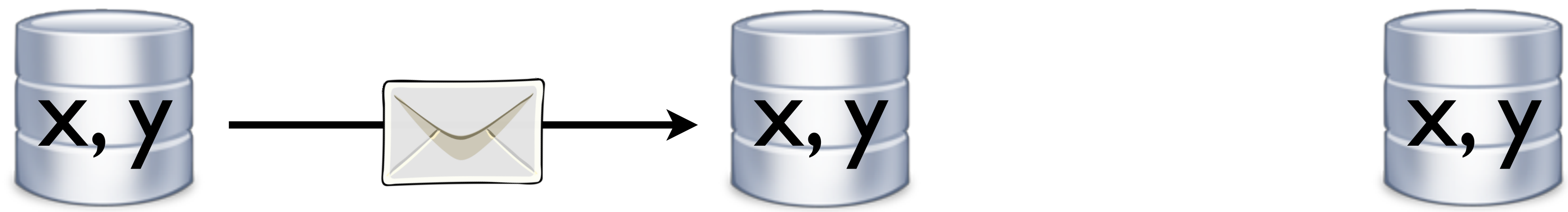
- Processes have to talk to each other (synchronise) to guarantee strong consistency

Eventual consistency and replicated data types, operationally

System model

- Database system consisting of multiple replicas (= data centre, machine, mobile device)
- Each replica stores a copy of all objects

System model



- Replicas can communicate via channels
- Asynchronous: no bound on how quickly a message will be delivered

  (in particular, because of network partitions)
- Reliable: every message is eventually delivered

  (so every partition eventually heals)
- For now: replicas are reliable too

High availability

- Clients connect to a replica of their choice

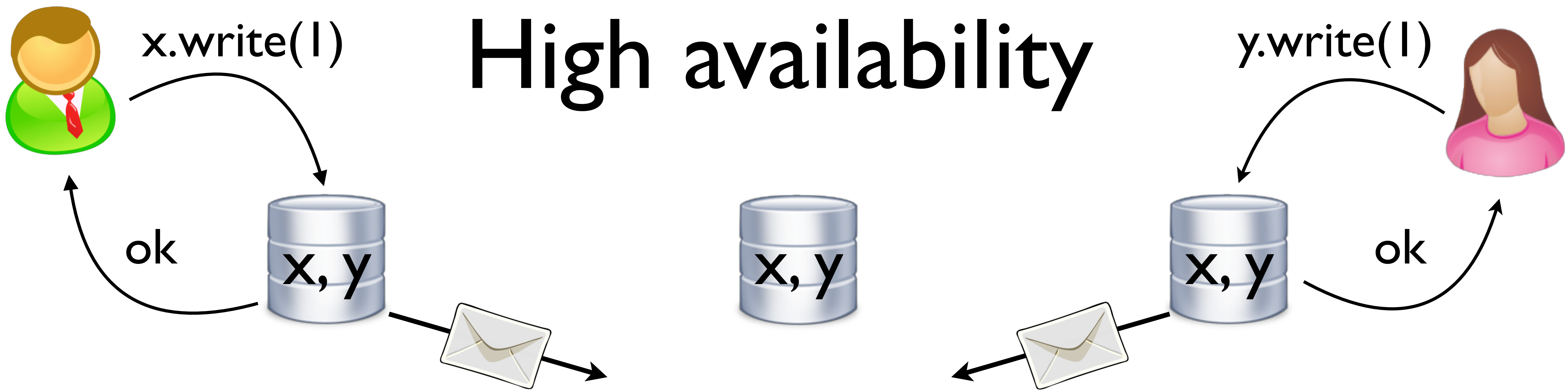
High availability

- Clients connect to a replica of their choice

High availability

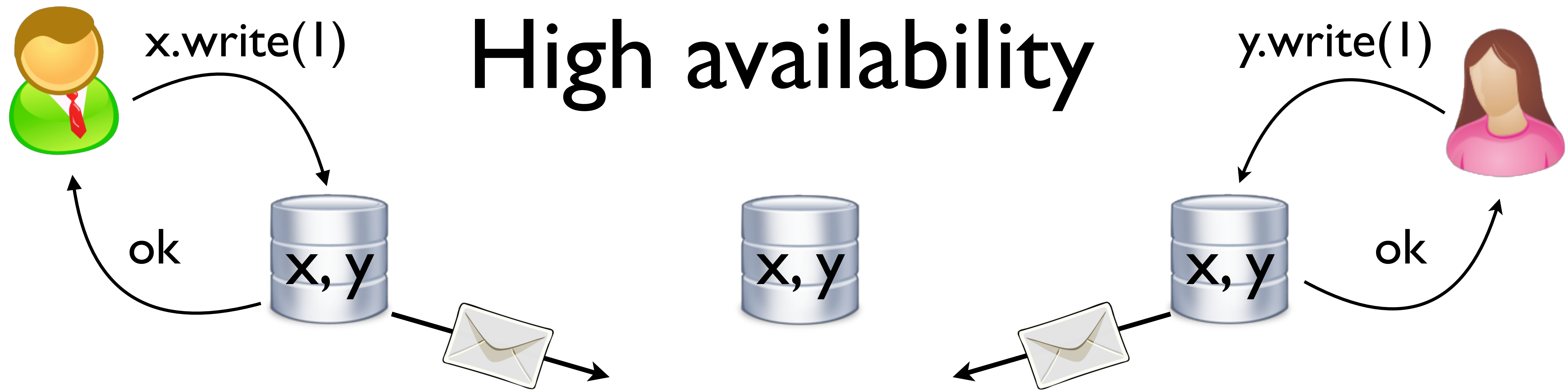
- Clients connect to a replica of their choice
- Replica has to respond to operations immediately, without communicating with others

High availability



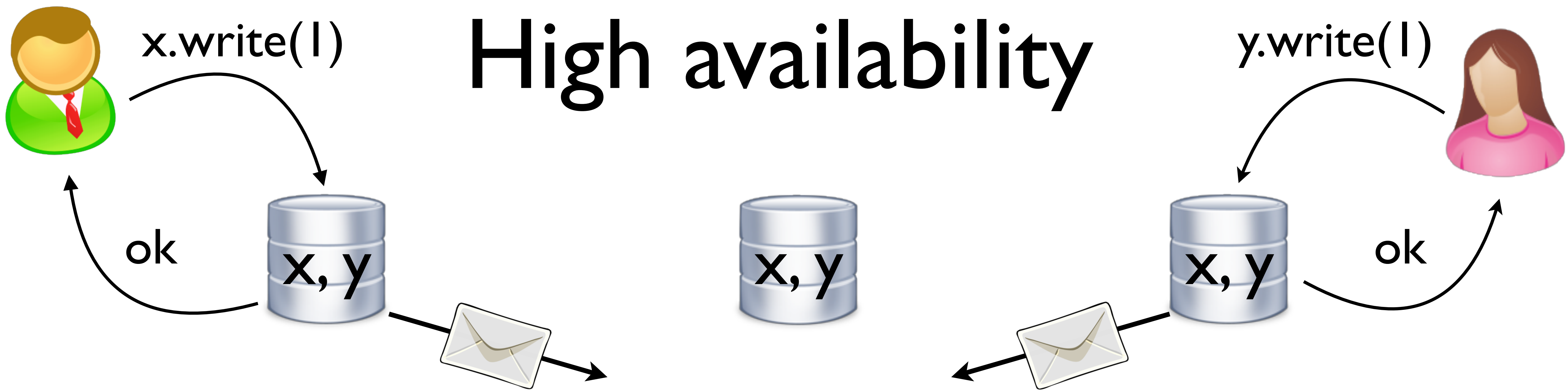
- Clients connect to a replica of their choice
- Replica has to respond to operations immediately, without communicating with others
- Propagate effects to other replicas later

High availability



- Clients connect to a replica of their choice
- Replica has to respond to operations immediately, without communicating with others
- Propagate effects to other replicas later
- Always available, low latency, but may not be strongly consistent

High availability



- Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

- Later more precise and stronger formulations of eventual consistency

Replicated data types

- Need a new kind of replicated data type: object state now lives at multiple replicas
- Aka CRDTs: commutative, convergent, conflict-free

  Just one type: operation-based replicated data types
- Object ➔ Type ➔ Operation signature

  For now fix a single object and type

Sequential semantics recap

- Set of states State
- Initial state $\sigma_0 \in$ State
- $[\![op]\!]_{val} \in$ State $\rightarrow$ Value
- $[\![op]\!]_{state} \in$ State $\rightarrow$ State

Replicated data types

σ

Object state at a replica: $\sigma \in \mathrm{State}$

Replicated data types

σ

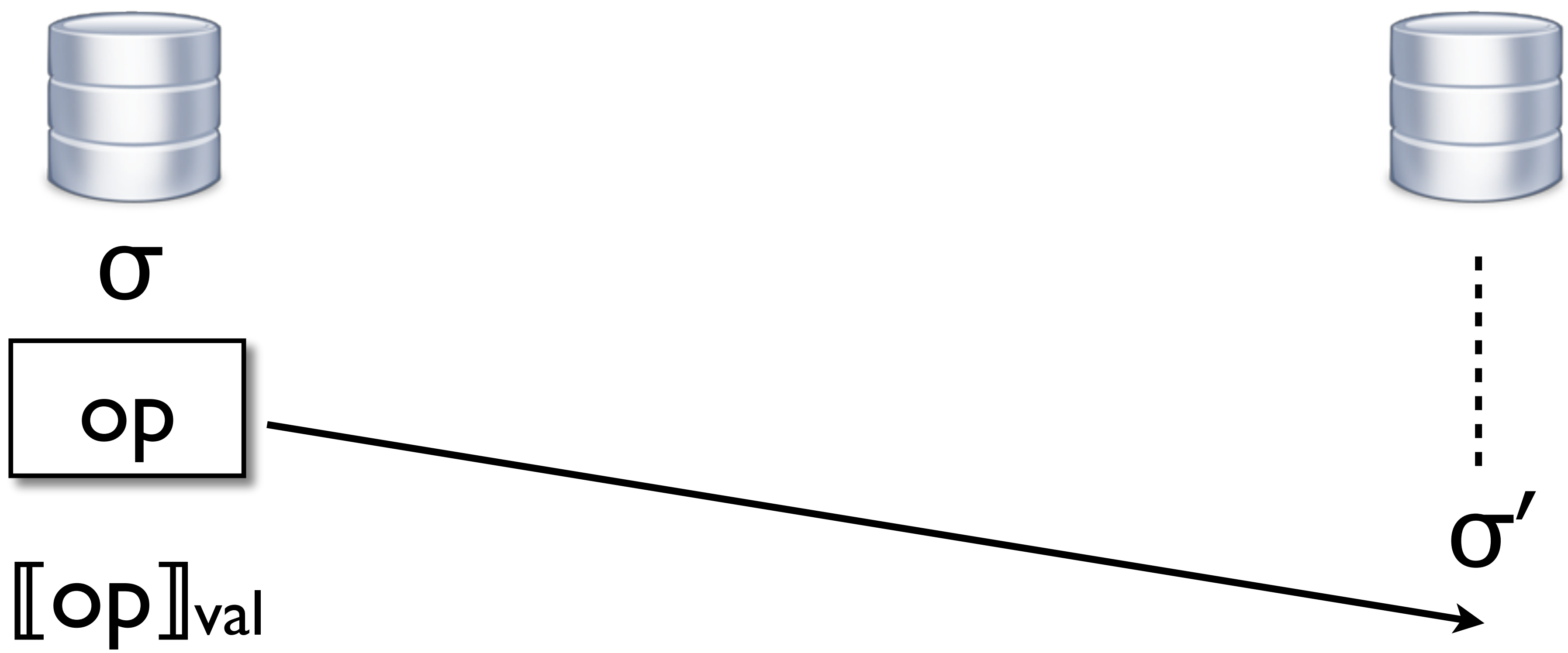
$\boxed{op}$

$[\![op]\!]_{val}$

Object state at a replica: $\sigma \in State$

Return value: $[\![op]\!]_{val} \in State \to Value$

Replicated data types

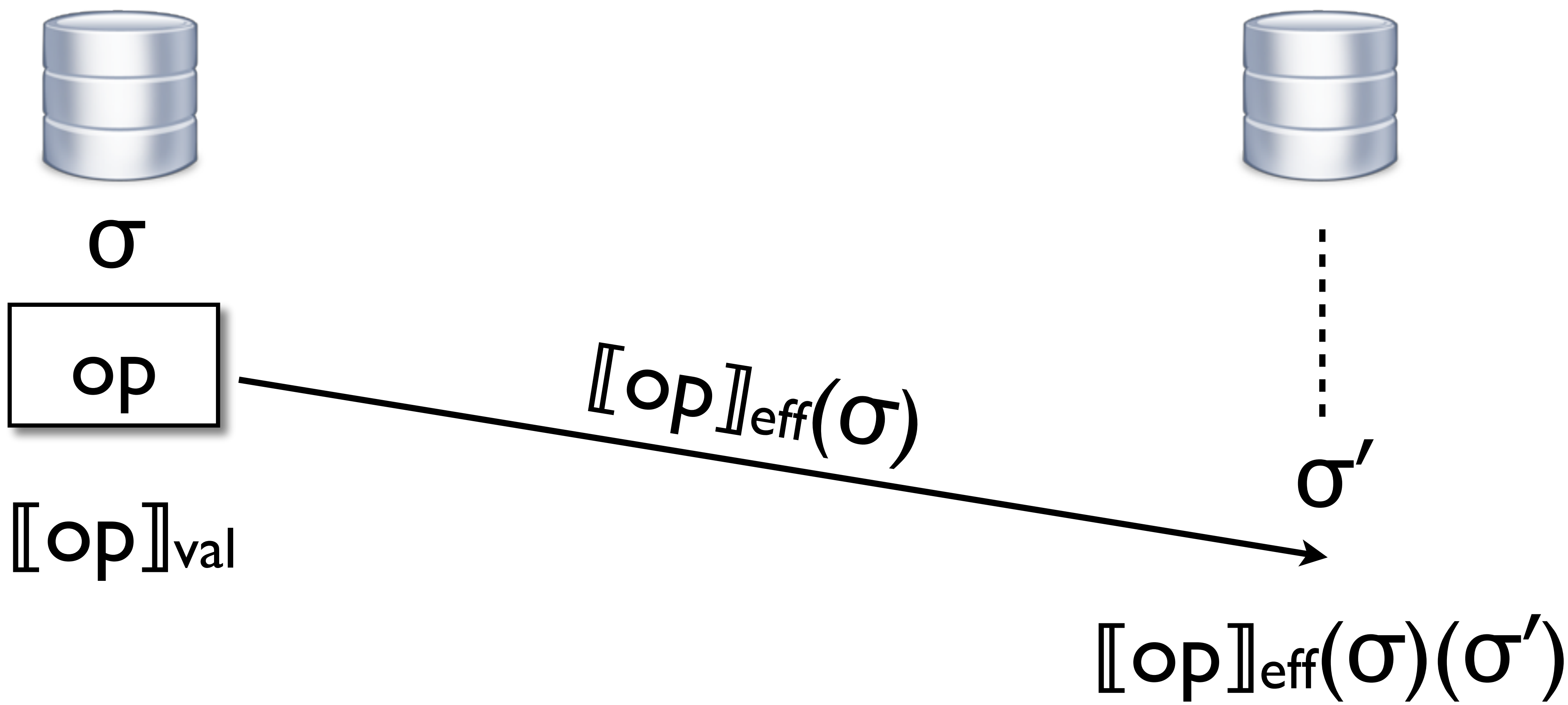


Object state at a replica: $\sigma \in \mathrm{State}$

Return value: $[\![\mathrm{op}]\!]_{\mathrm{val}} \in \mathrm{State} \rightarrow \mathrm{Value}$

The operation affects a different state σ'!

Replicated data types

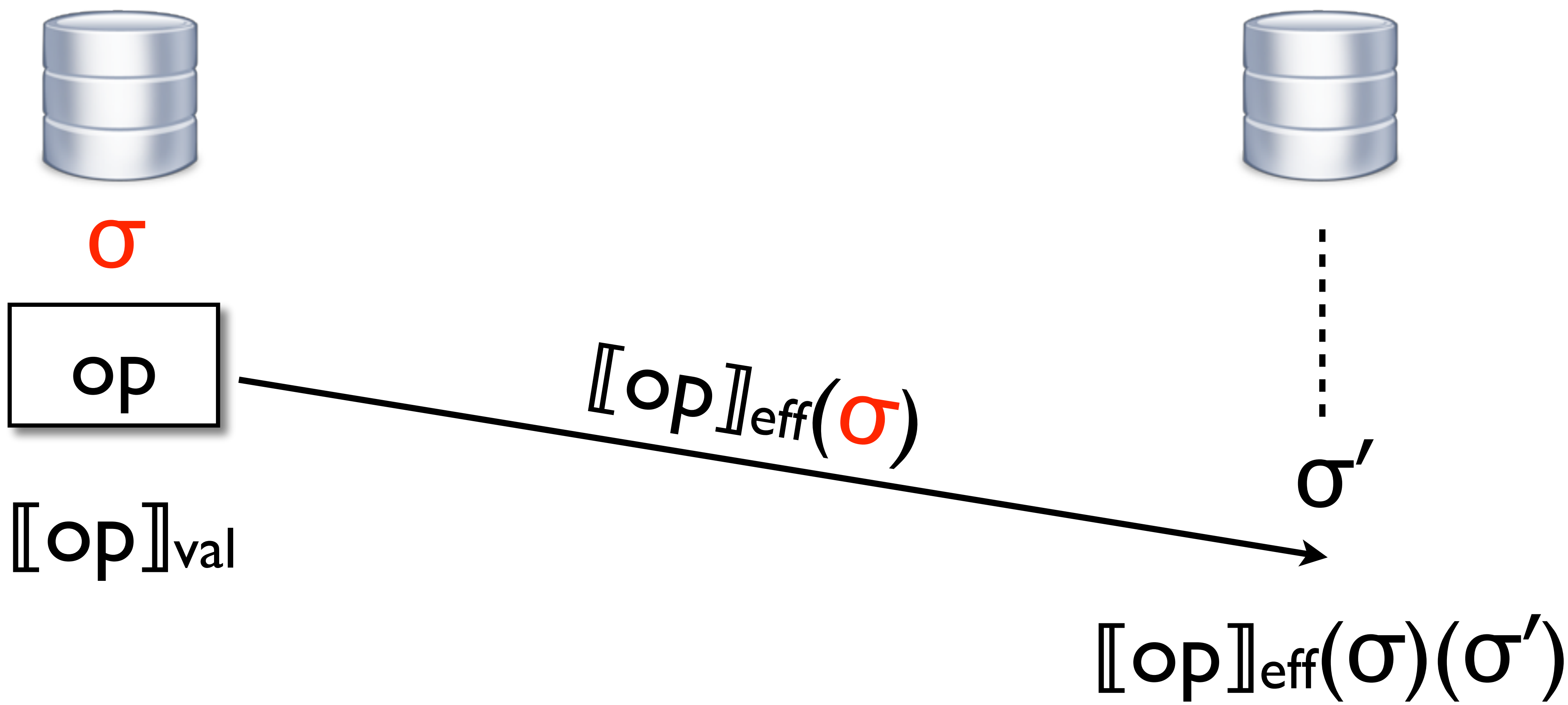


Object state at a replica: $\sigma \in \mathrm{State}$

Return value: $[\![op]\!]_{val} \in \mathrm{State} \to \mathrm{Value}$

Effector: $[\![op]\!]_{eff} \in \mathrm{State} \to (\mathrm{State} \to \mathrm{State})$

Replicated data types

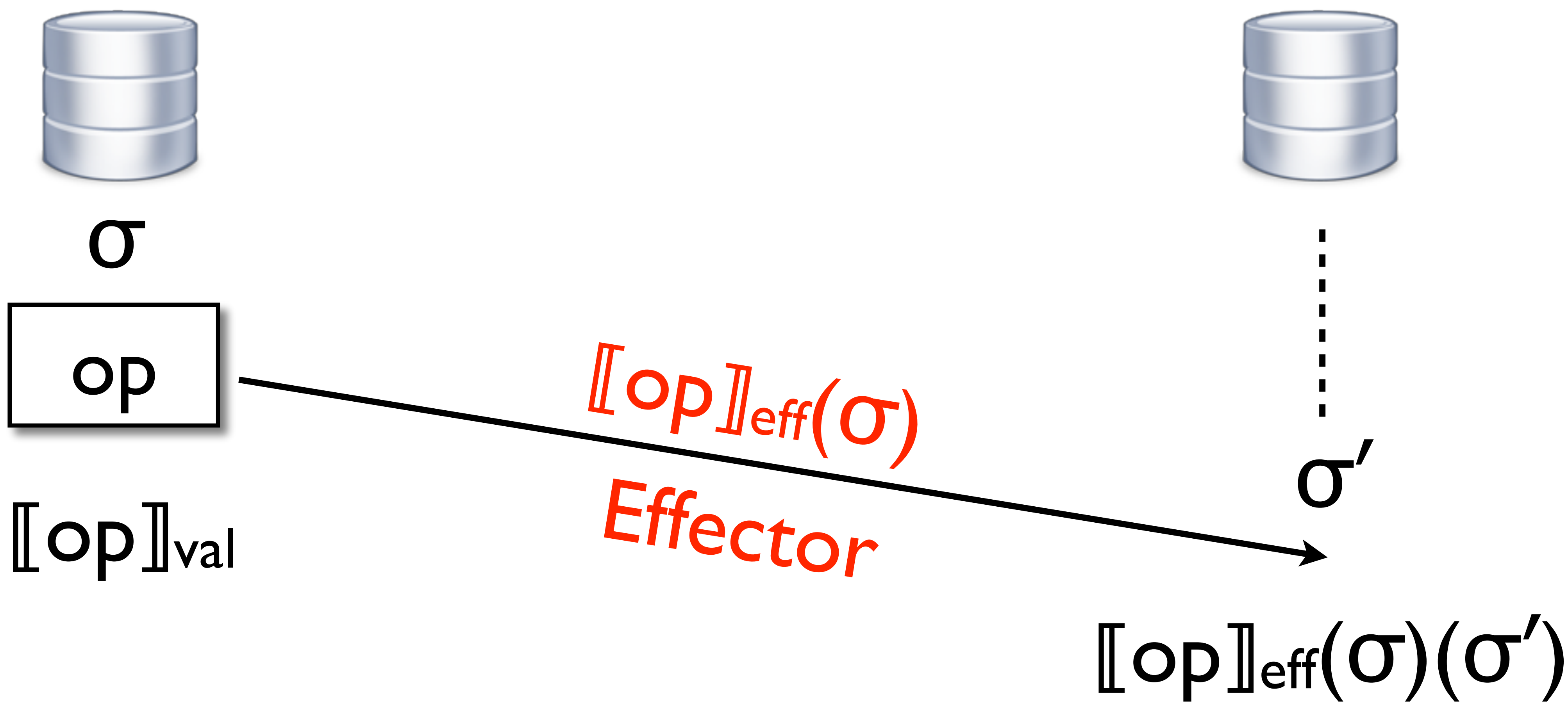


Object state at a replica: $\sigma \in \text{State}$

Return value: $[\![op]\!]_{val} \in \text{State} \to \text{Value}$

Effector: $[\![op]\!]_{eff} \in \text{State} \to (\text{State} \to \text{State})$

Replicated data types

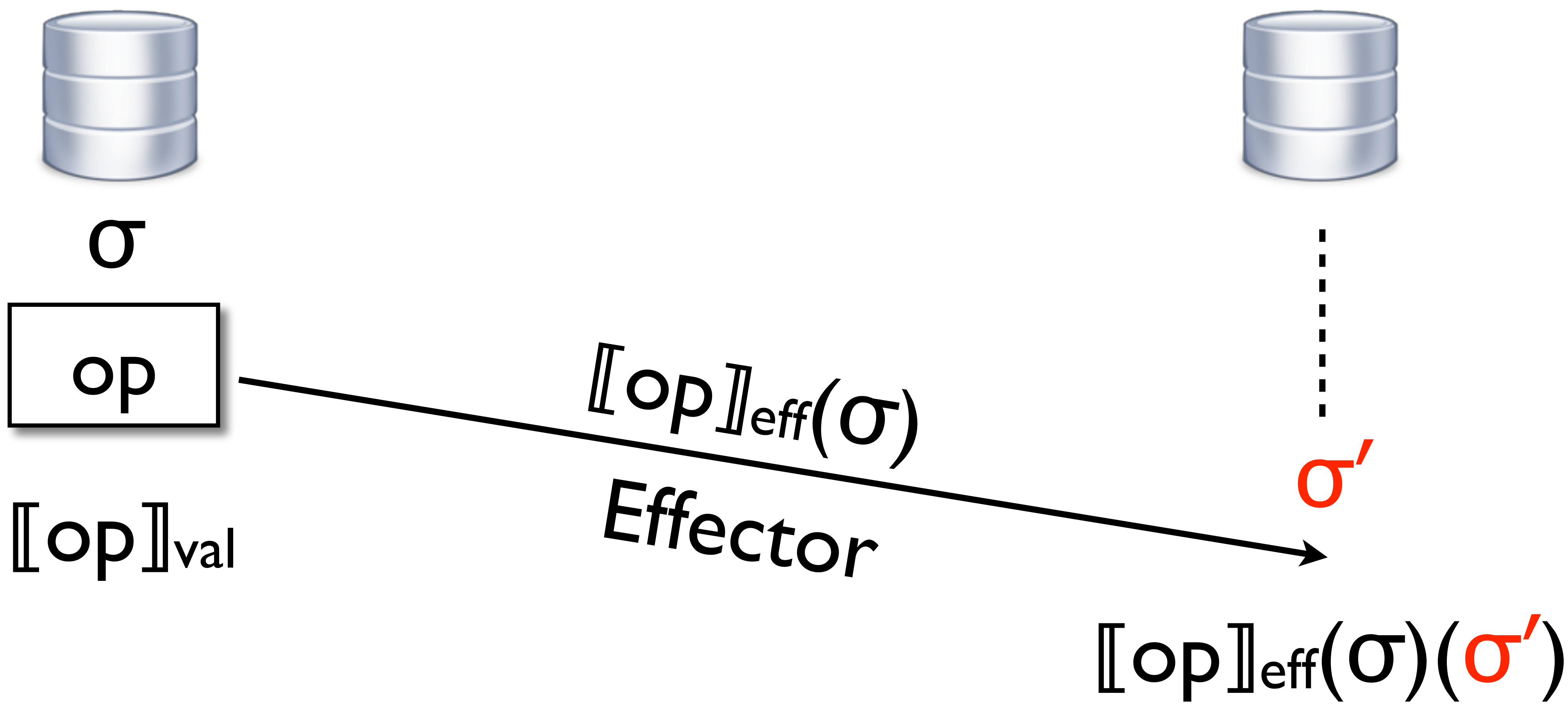


Object state at a replica: $\sigma \in \mathrm{State}$

Return value: $[\![\mathrm{op}]\!]_{\mathrm{val}} \in \mathrm{State} \to \mathrm{Value}$

Effector: $[\![\mathrm{op}]\!]_{\mathrm{eff}} \in \mathrm{State} \to (\mathrm{State} \to \mathrm{State})$

Replicated data types

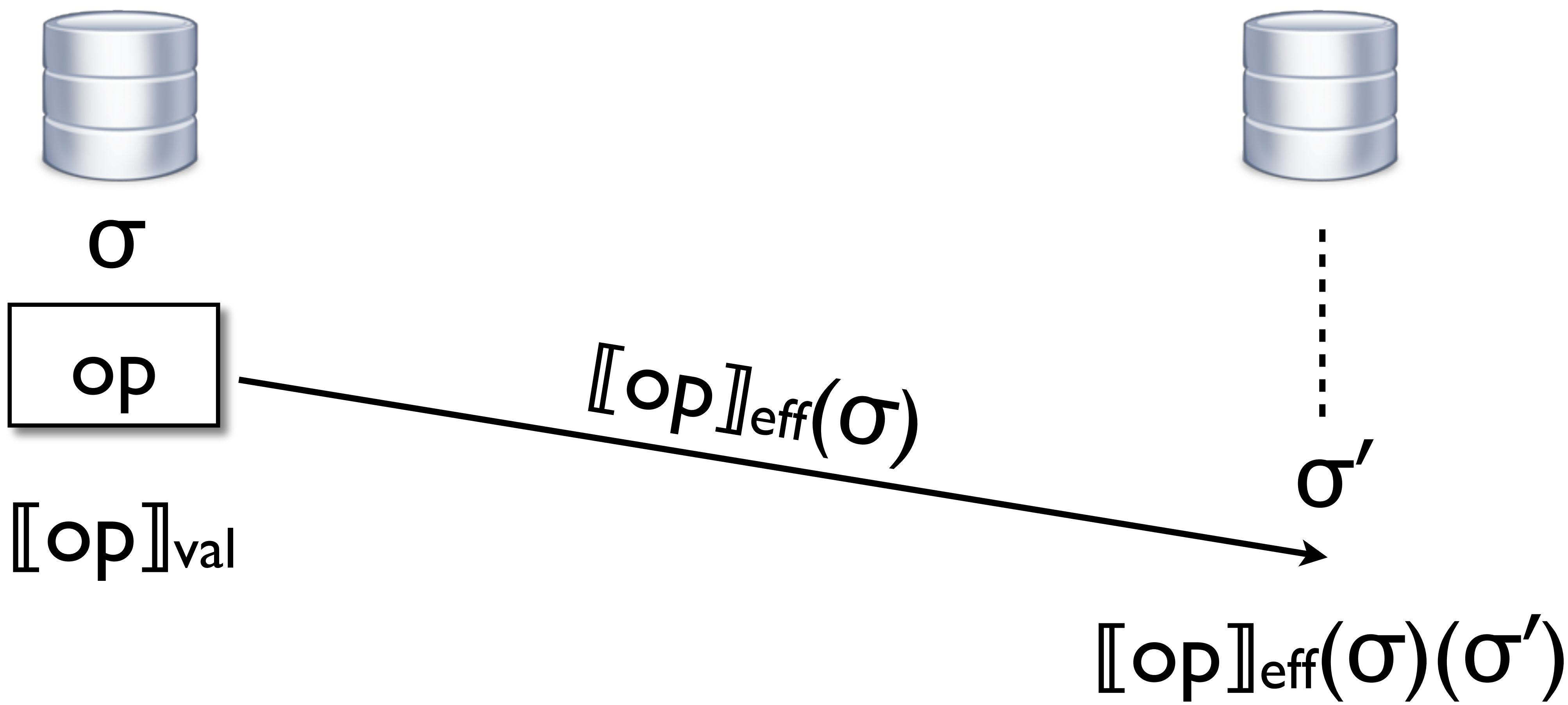


Object state at a replica: $\sigma \in \mathrm{State}$

Return value: $[\![op]\!]_{val} \in \mathrm{State} \rightarrow \mathrm{Value}$

Effector: $[\![op]\!]_{eff} \in \mathrm{State} \rightarrow (\mathrm{State} \rightarrow \mathrm{State})$

Counter

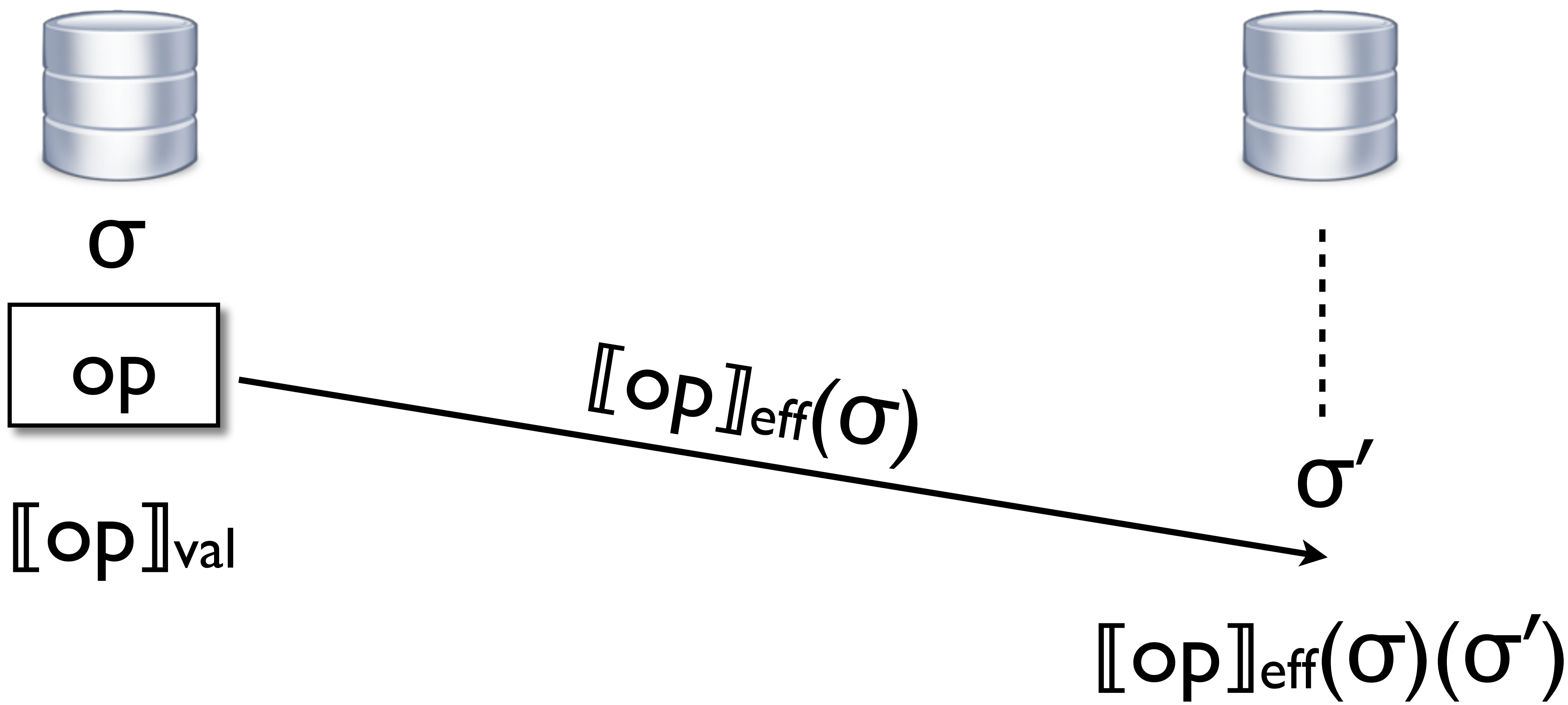


$$\text{State} = \mathbb{N}$$

$$[\![\text{read}()]\!]_{\text{val}}(\sigma) = \sigma$$

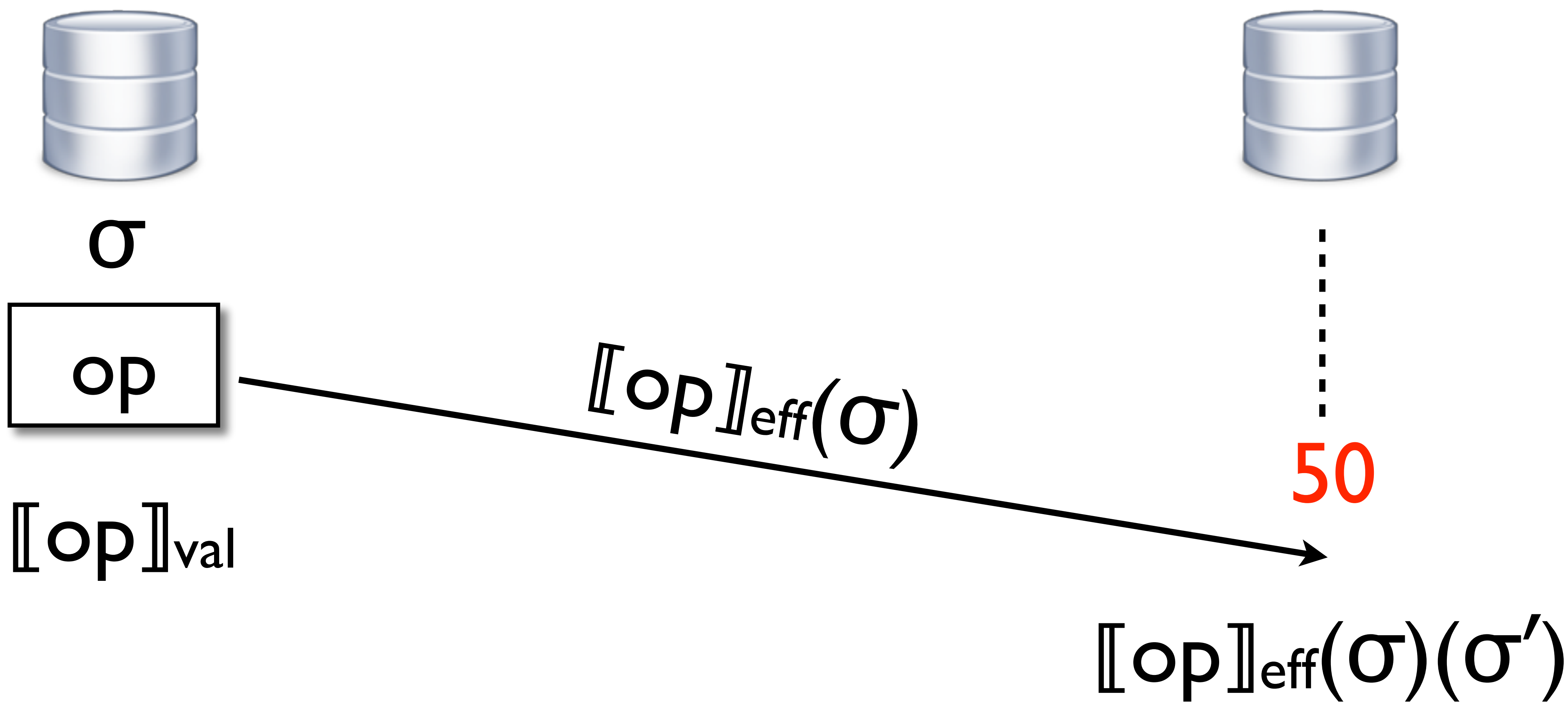
$$[\![\text{read}()]\!]_{\text{eff}}(\sigma) = \lambda\sigma.\,\sigma$$

Counter



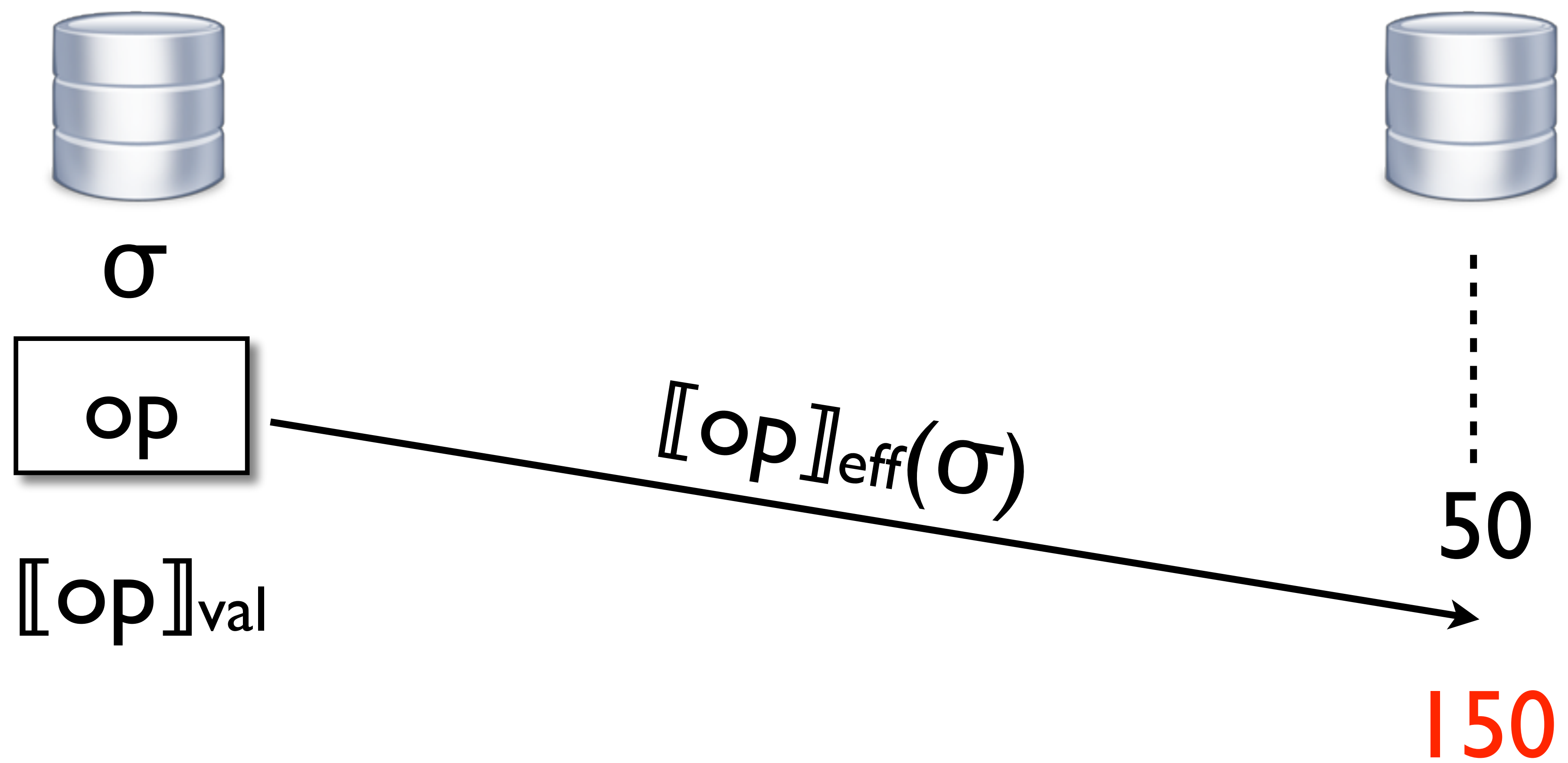
$$[\![\mathrm{add}(100)]\!]_{\mathrm{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma' + 100)$$

Counter



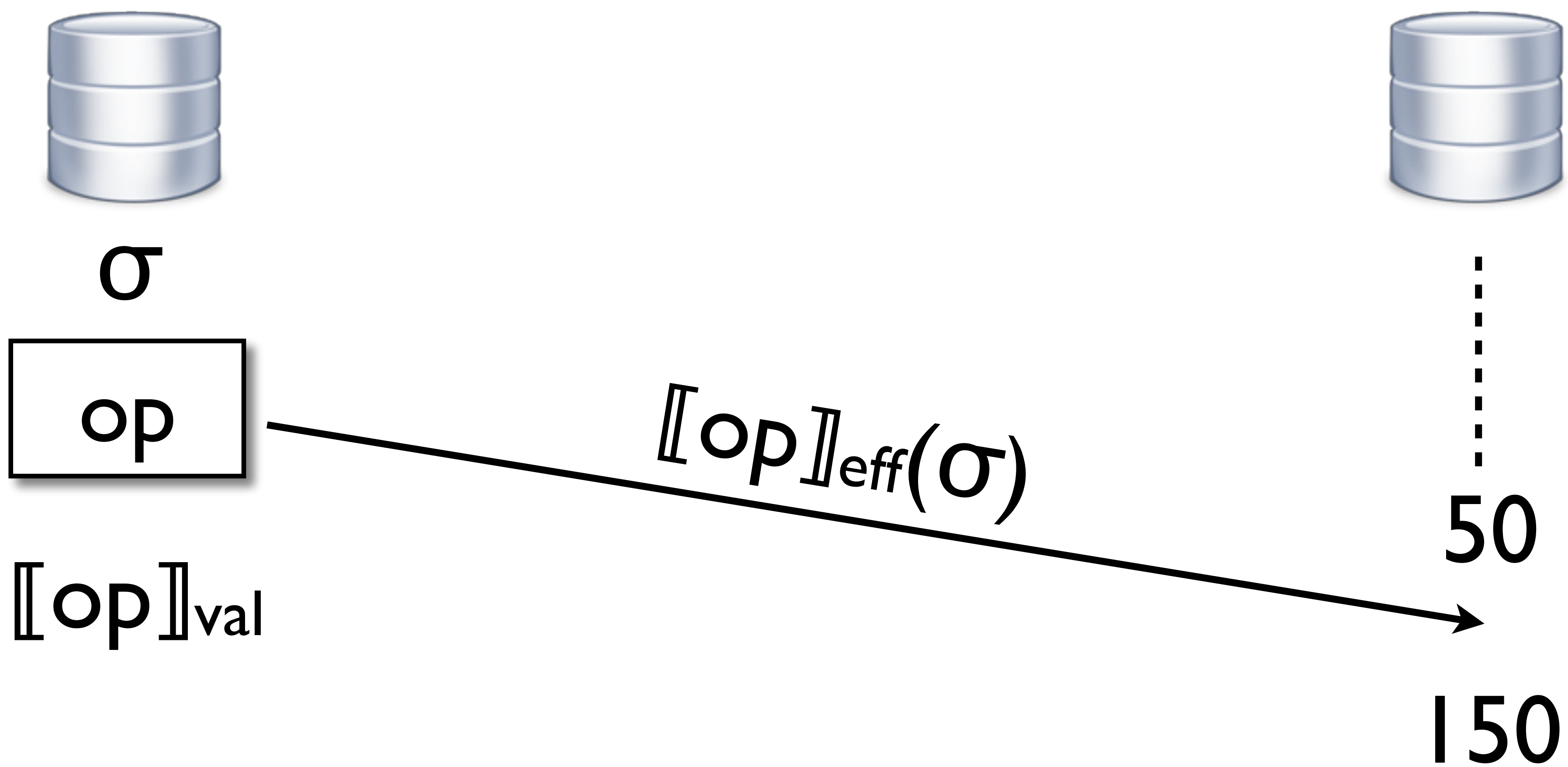
$$[\![\mathrm{add}(100)]\!]_{\mathrm{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma' + 100)$$

Counter



$$[\![\mathrm{add}(100)]\!]_{\mathrm{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma' + 100)$$

Counter



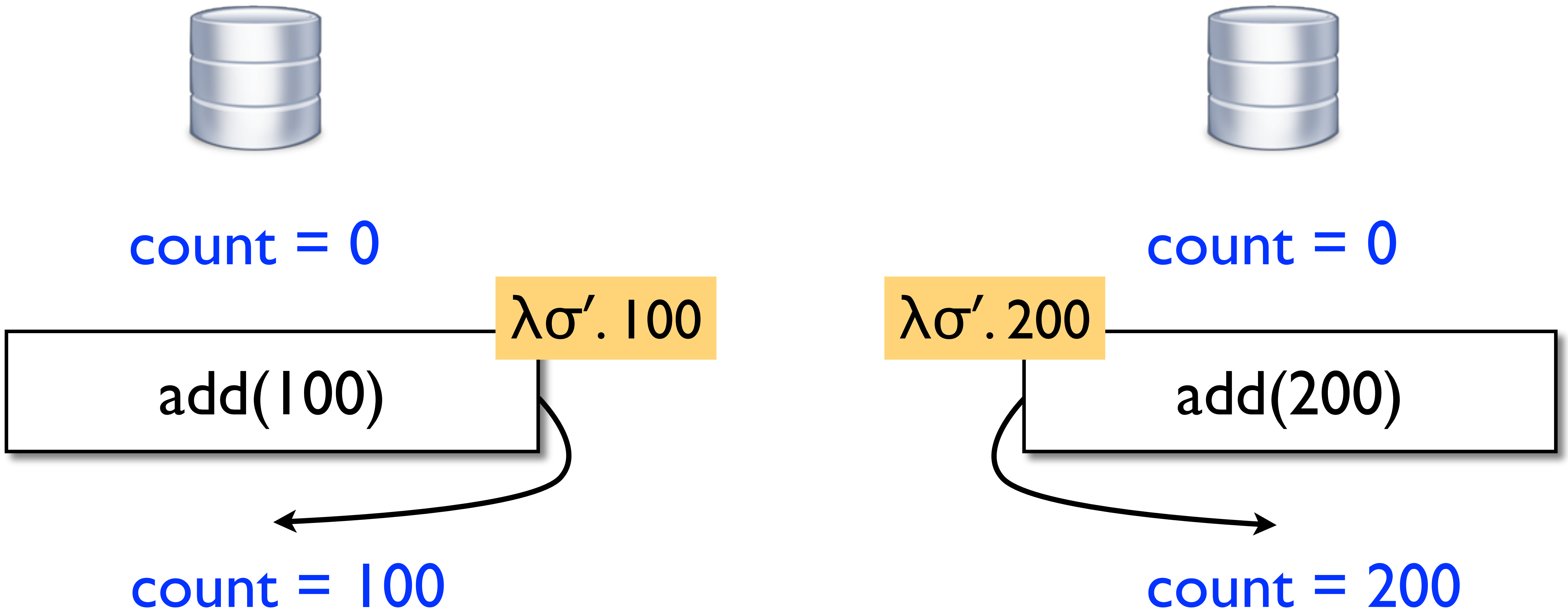
$$[\![\mathrm{add}(100)]\!]_{\mathrm{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma + 100)$$

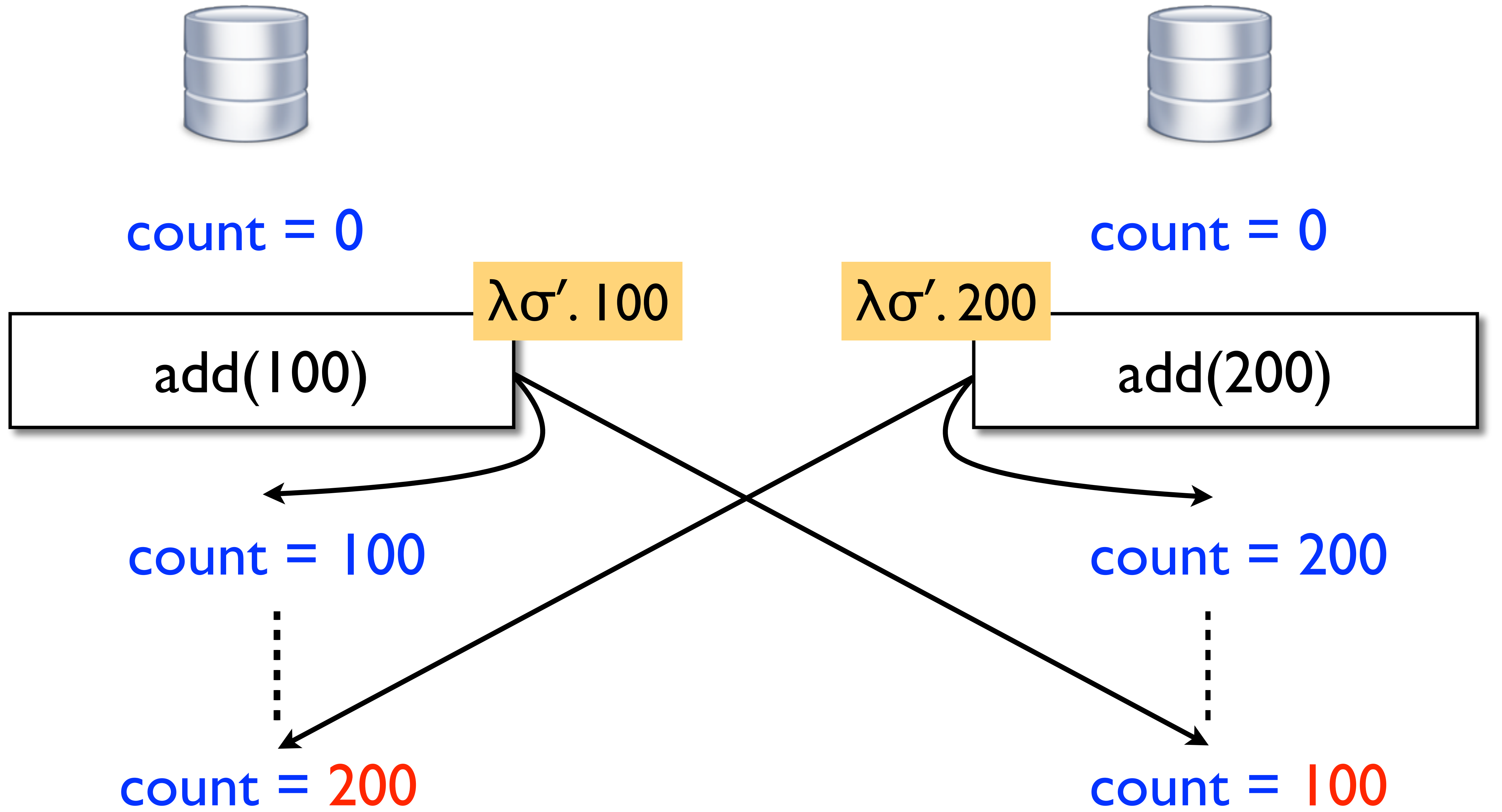
count = 0

add(100)

count = 0

add(200)


count = 0
λσ′. 100
add(100)
count = 100
count = 0
λσ′. 200
add(200)
count = 200



Quiescent consistency violated: all updates have been delivered, yet replicas will never converge

Ensuring quiescent consistency

- Effectors have to commute:

$$\forall op_1, op_2, \sigma_1, \sigma_2.\ [\![op_1]\!]_{eff}(\sigma_1)\ ;[\![op_2]\!]_{eff}(\sigma_2) = [\![op_2]\!]_{eff}(\sigma_2)\ ;[\![op_1]\!]_{eff}(\sigma_1)$$

- Convergence: replicas that received the same sets of updates end up in the same state

  (even when messages are received in different orders)

Ensuring quiescent consistency

- Effectors have to commute:

$$\forall op_1, op_2, \sigma_1, \sigma_2.\ [\![op_1]\!]_{eff}(\sigma_1)\ ;\ [\![op_2]\!]_{eff}(\sigma_2) = [\![op_2]\!]_{eff}(\sigma_2)\ ;\ [\![op_1]\!]_{eff}(\sigma_1)$$

- Convergence: replicas that received the same sets of updates end up in the same state

  (even when messages are received in different orders)

- Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

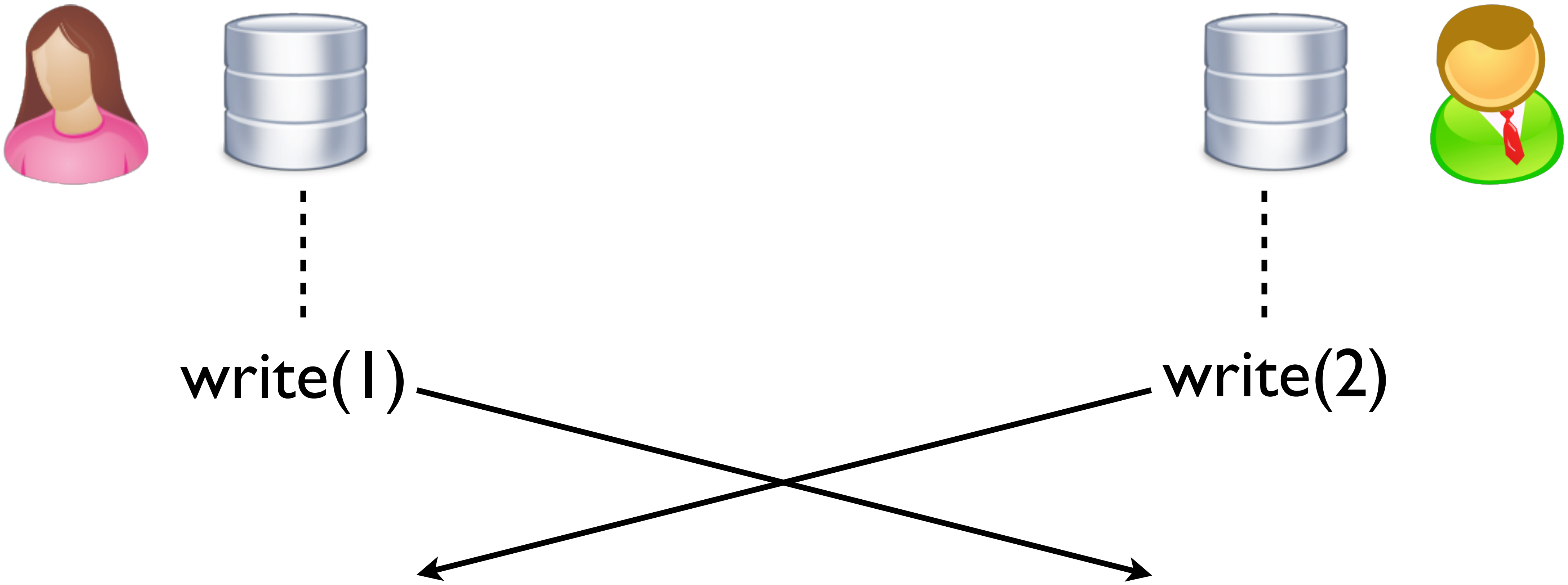
  (because update get eventually delivered)

Replicated data types

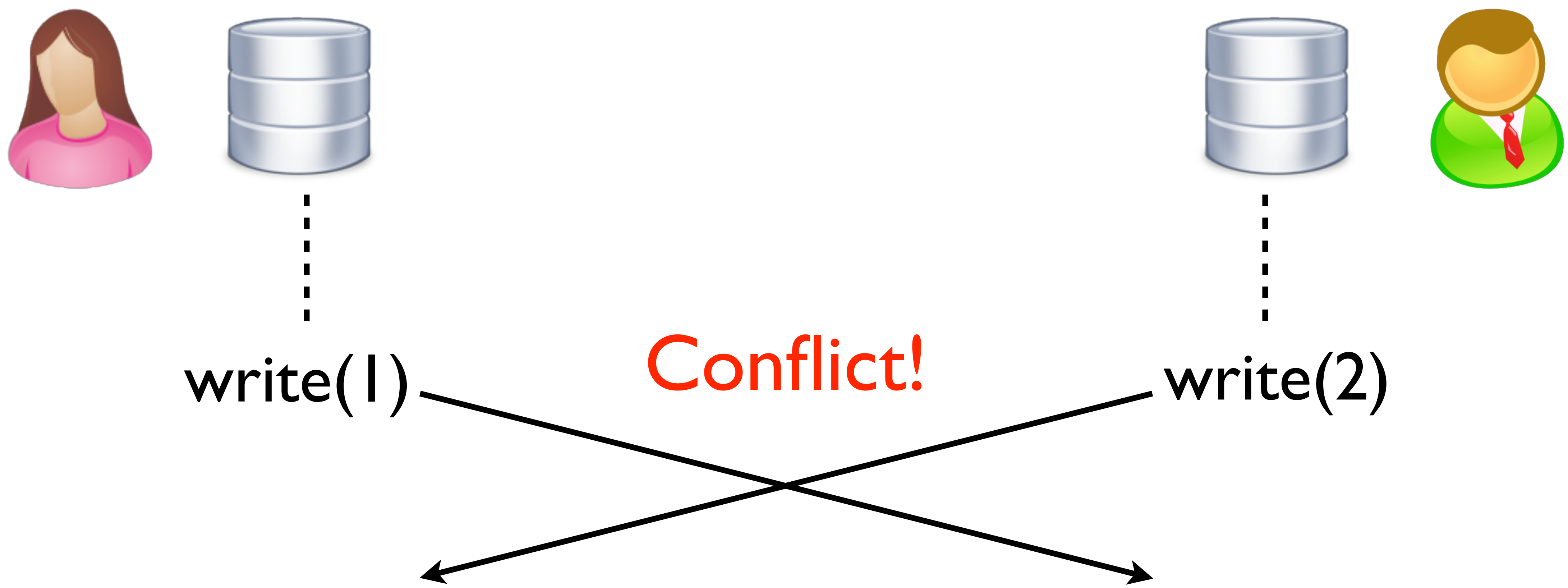
- Counter
- Last-writer-wins register
- Multi-valued register
- Add-wins set
- Remove-wins set
- List
- ...

Read-write register

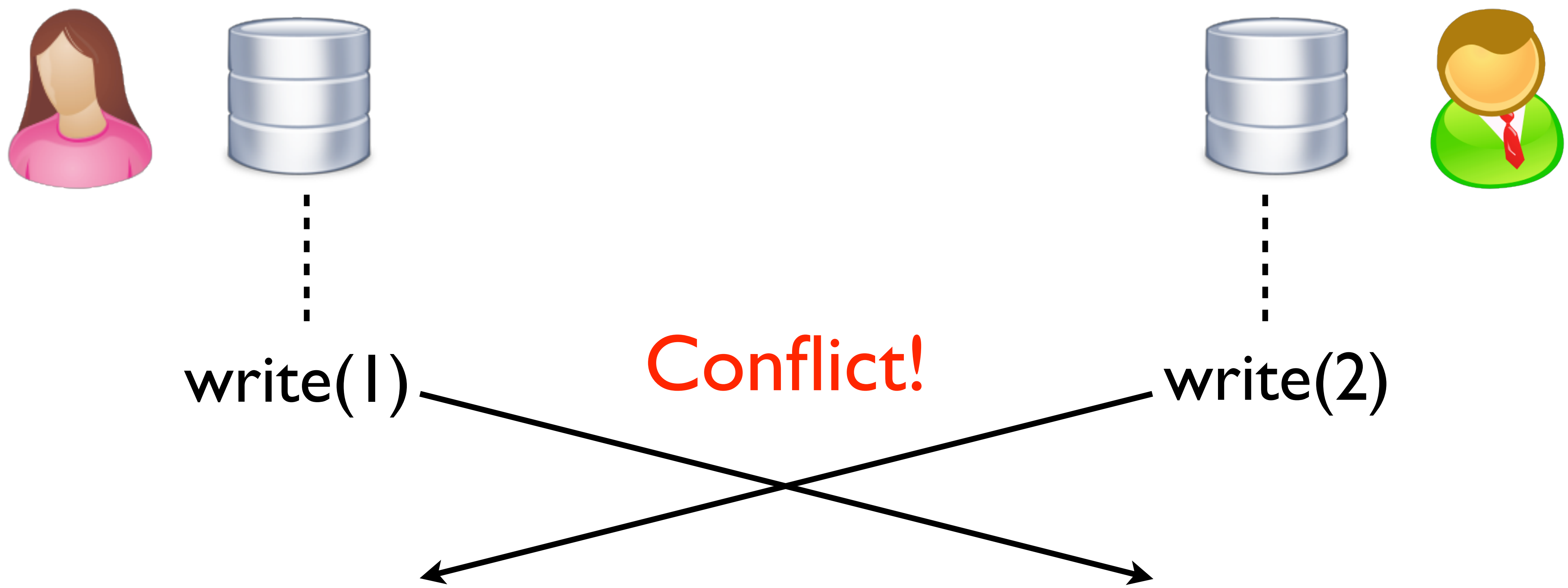
Read-write register



Read-write register



Read-write register



- No right or wrong solutions: depends on the application requirements
- E.g., could report the conflict to the user: multi-valued register

Last-writer-wins register

- Shared memory: an arbitrary write will win
- Conflict arbitrated using timestamps: last write wins
- Link to shared-memory consistency models

Last-writer-wins register

$$\text{State} \;=\; \text{Value} \times \text{Timestamp}$$

$$[\![\text{read}()]\!]_{\text{val}}(v, t) \;=\; v$$

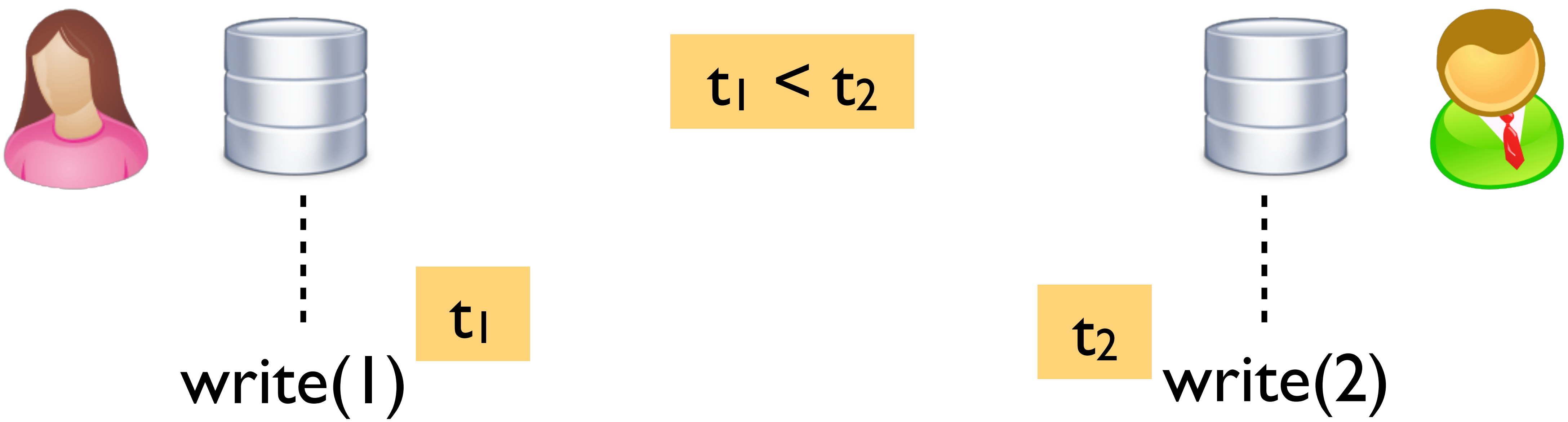
Last-writer-wins register

$$[\![\mathsf{write}(v_{new})]\!]_{eff}(v, t) \;=$$

$$\mathsf{let}\ t_{new} = \mathit{newUniqueTS}()\ \mathsf{in}$$

$$\lambda(v', t').\ \mathsf{if}\ t_{new} > t'\ \mathsf{then}\ (v_{new}, t_{new})\ \mathsf{else}\ (v, t)$$

Last-writer-wins register

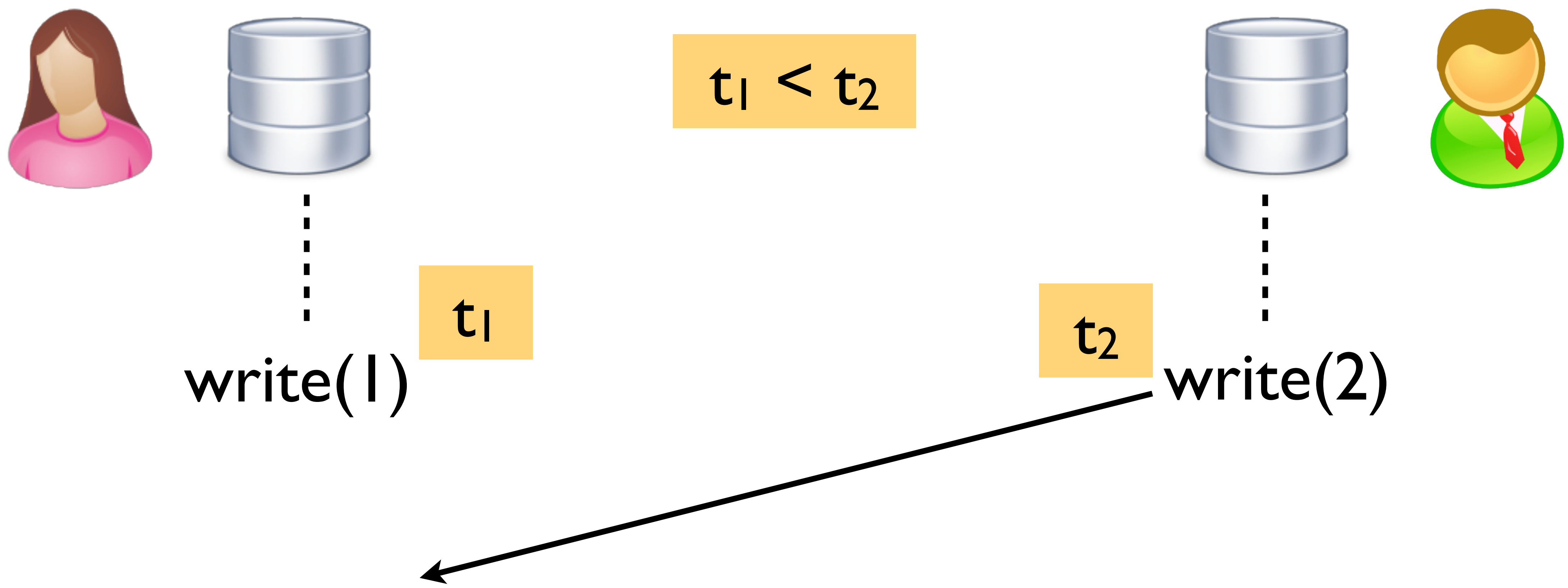


$$[\![\mathrm{write}(v_{new})]\!]_{eff}(v, t) \ =$$

$$\quad \mathrm{let}\ t_{new} = \mathit{newUniqueTS}()\ \mathrm{in}$$

$$\quad \lambda(v', t').\ \mathrm{if}\ t_{new} > t'\ \mathrm{then}\ (v_{new}, t_{new})\ \mathrm{else}\ (v, t)$$

Last-writer-wins register

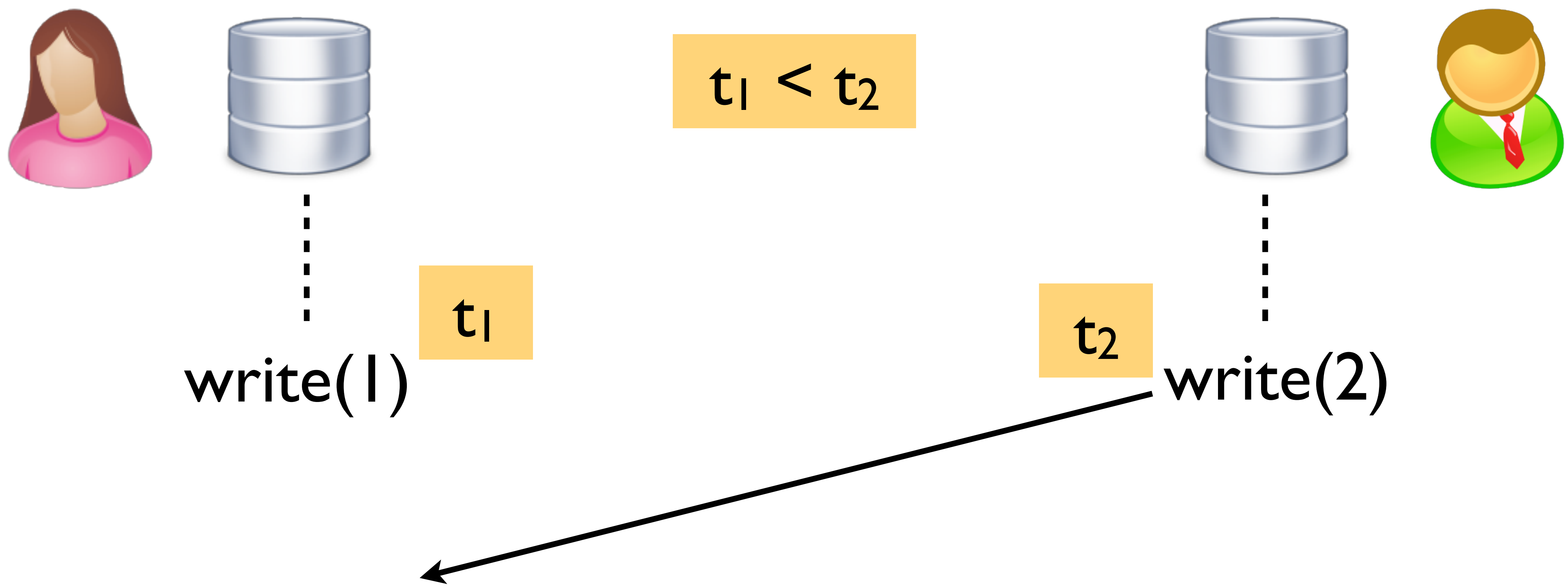


$[\![\mathrm{write}(v_{new})]\!]_{eff}(v, t) =$

$\quad \mathrm{let}\ t_{new} = \mathit{newUniqueTS}()\ \mathrm{in}$

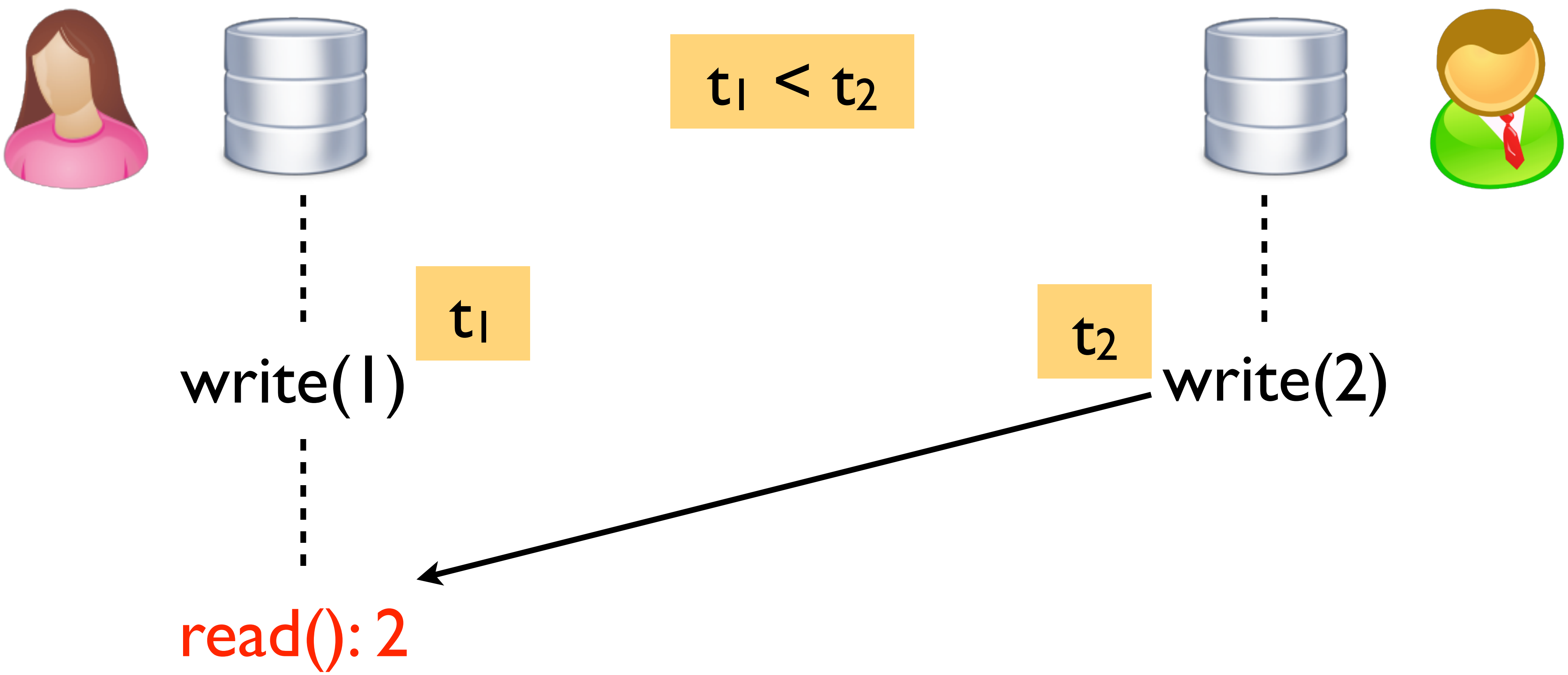
$\quad \lambda(v', t').\ \mathrm{if}\ t_{new} > t'\ \mathrm{then}\ (v_{new}, t_{new})\ \mathrm{else}\ (v, t)$

Last-writer-wins register



$$
\begin{aligned}
&[\![\mathrm{write}(v_{new})]\!]_{\mathrm{eff}}(v, t) \;\; = \\
&\quad \mathrm{let}\ t_{new} = \mathit{newUniqueTS}()\ \mathrm{in} \\
&\quad \lambda(v', t').\ \mathrm{if}\ t_{new} > t'\ \mathrm{then}\ (v_{new}, t_{new})\ \mathrm{else}\ (v, t)
\end{aligned}
$$

Last-writer-wins register

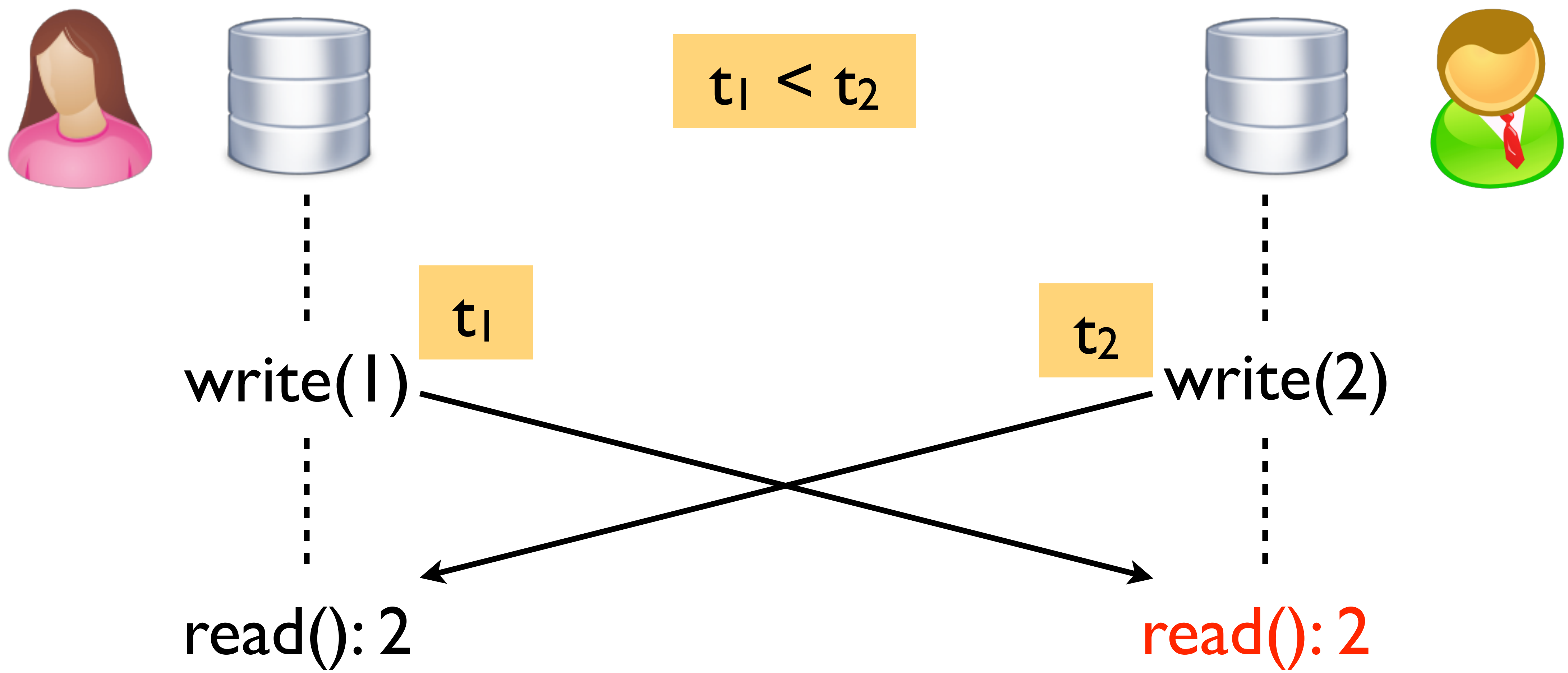


$[\![\text{write}(v_{new})]\!]_{eff}(v, t) =$

$\quad \text{let } t_{new} = \textit{newUniqueTS}() \text{ in}$

$\quad \lambda(v', t').\ \text{if } t_{new} > t' \text{ then } (v_{new}, t_{new}) \text{ else } (v, t)$

Last-writer-wins register

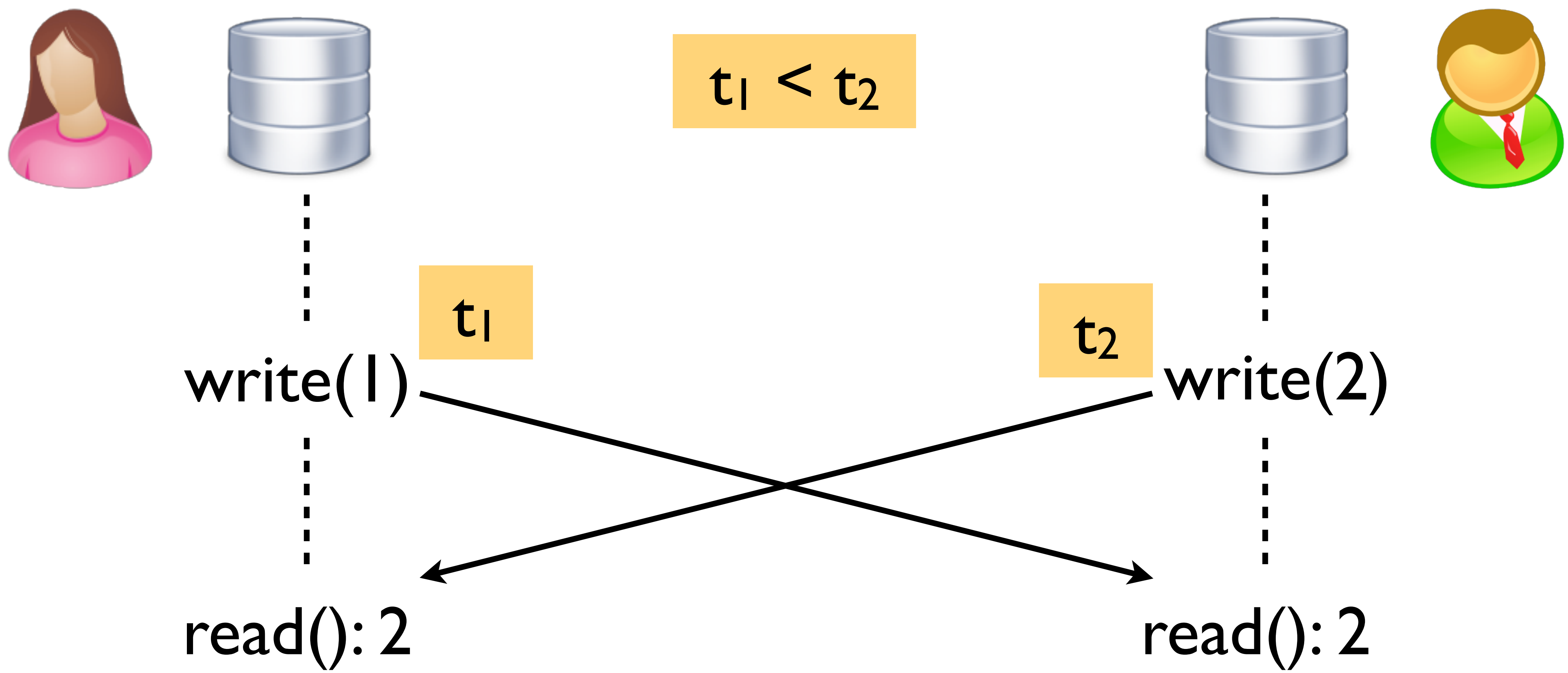


$$[\![\mathrm{write}(v_{new})]\!]_{eff}(v, t) \;=$$

$$\quad \mathrm{let}\ t_{new} = \mathit{newUniqueTS}()\ \mathrm{in}$$

$$\quad \lambda(v', t').\ \mathrm{if}\ t_{new} > t'\ \mathrm{then}\ (v_{new}, t_{new})\ \mathrm{else}\ (v, t)$$

Last-writer-wins register

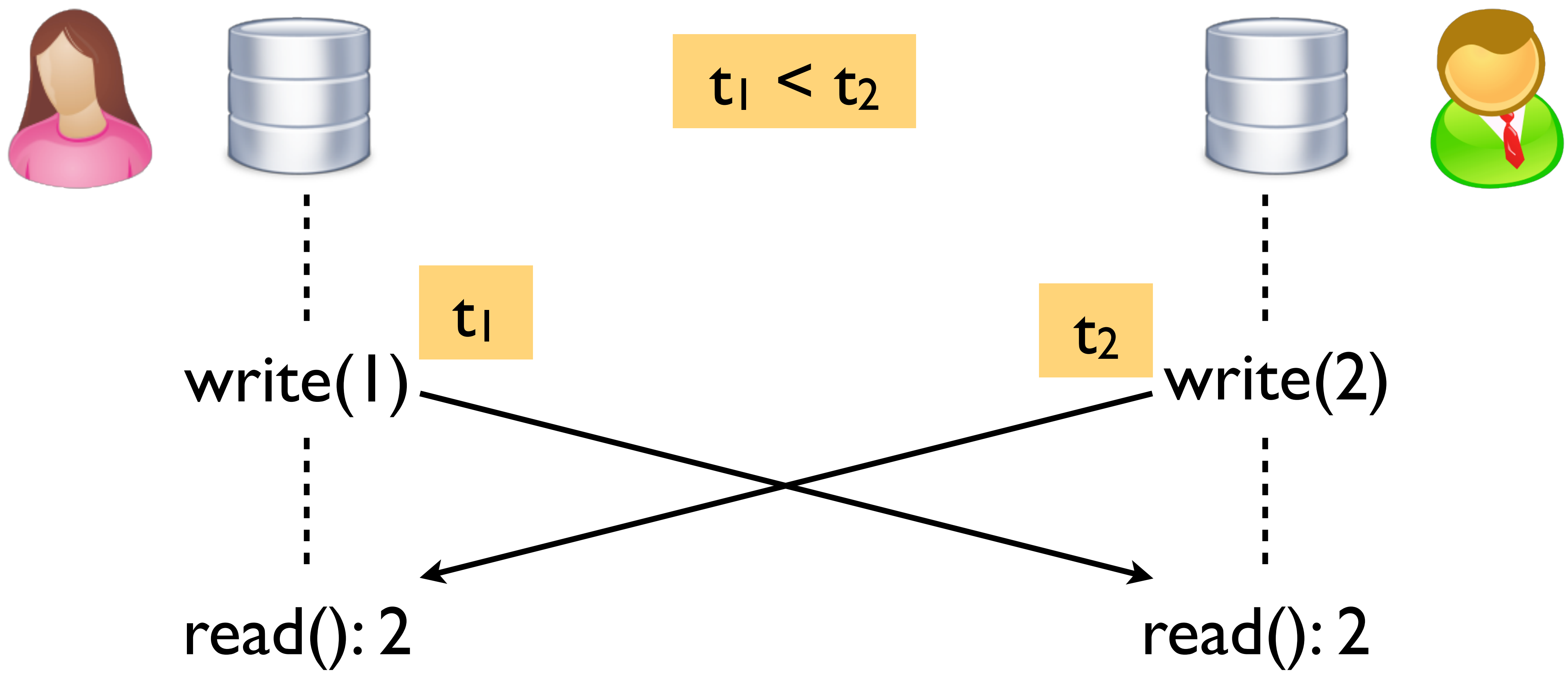


$$[\![\mathrm{write}(v_{new})]\!]_{eff}(v, t) \; = $$

$$\text{let } t_{new} = \mathit{newUniqueTS}() \text{ in}$$

$$\lambda(v', t').\ \text{if } t_{new} > t' \text{ then } (v_{new}, t_{new}) \text{ else } (v, t)$$

Last-writer-wins register

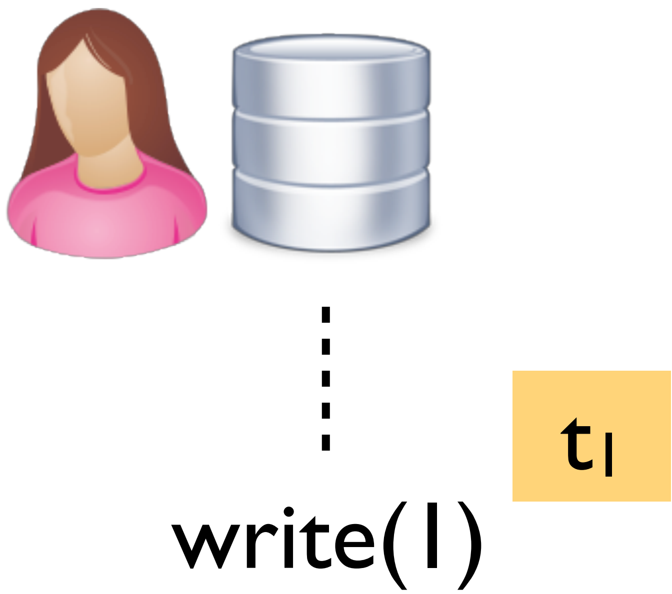


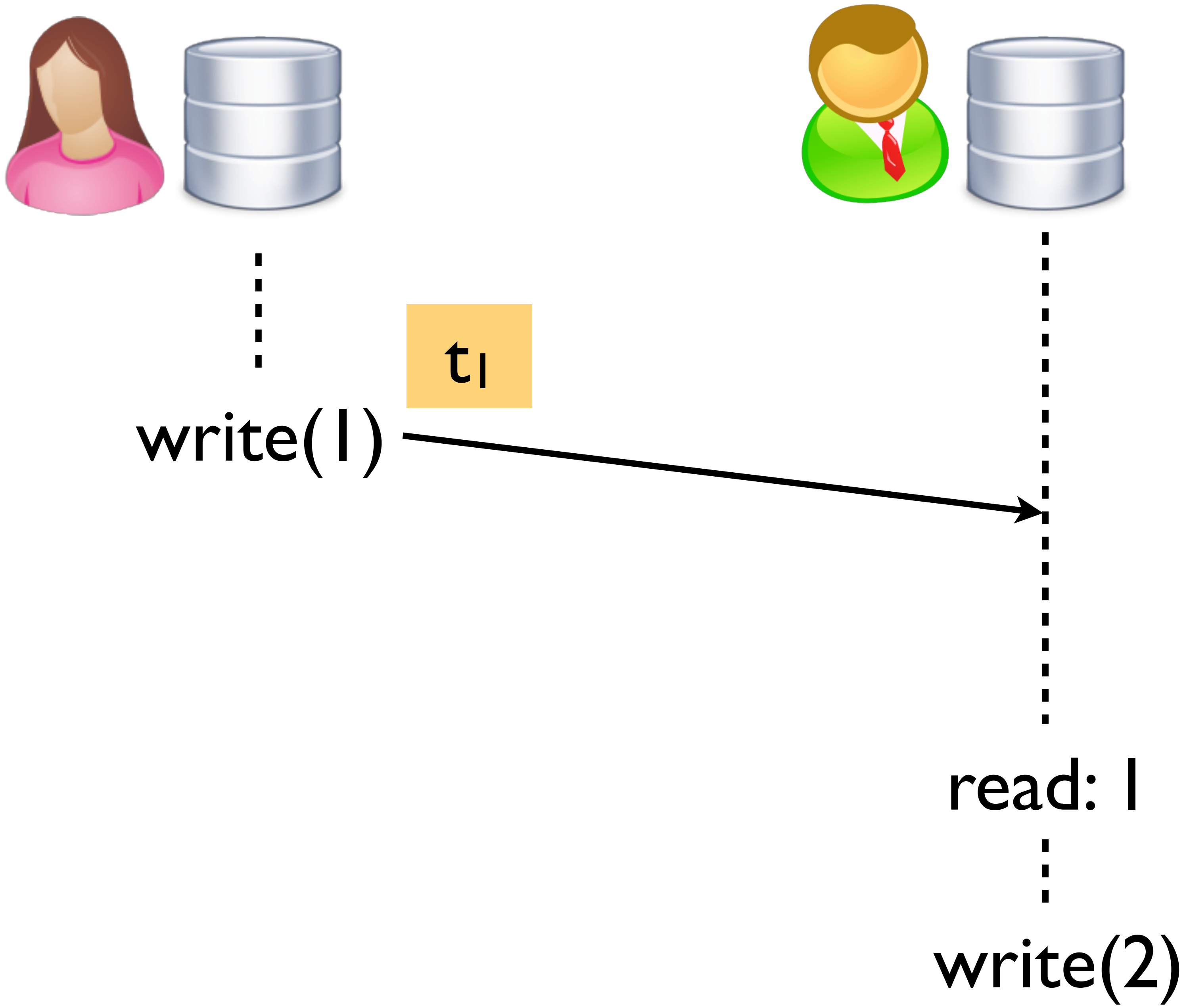
Effectors are commutative: the write with the highest timestamp wins regardless of the order of application

Generating timestamps

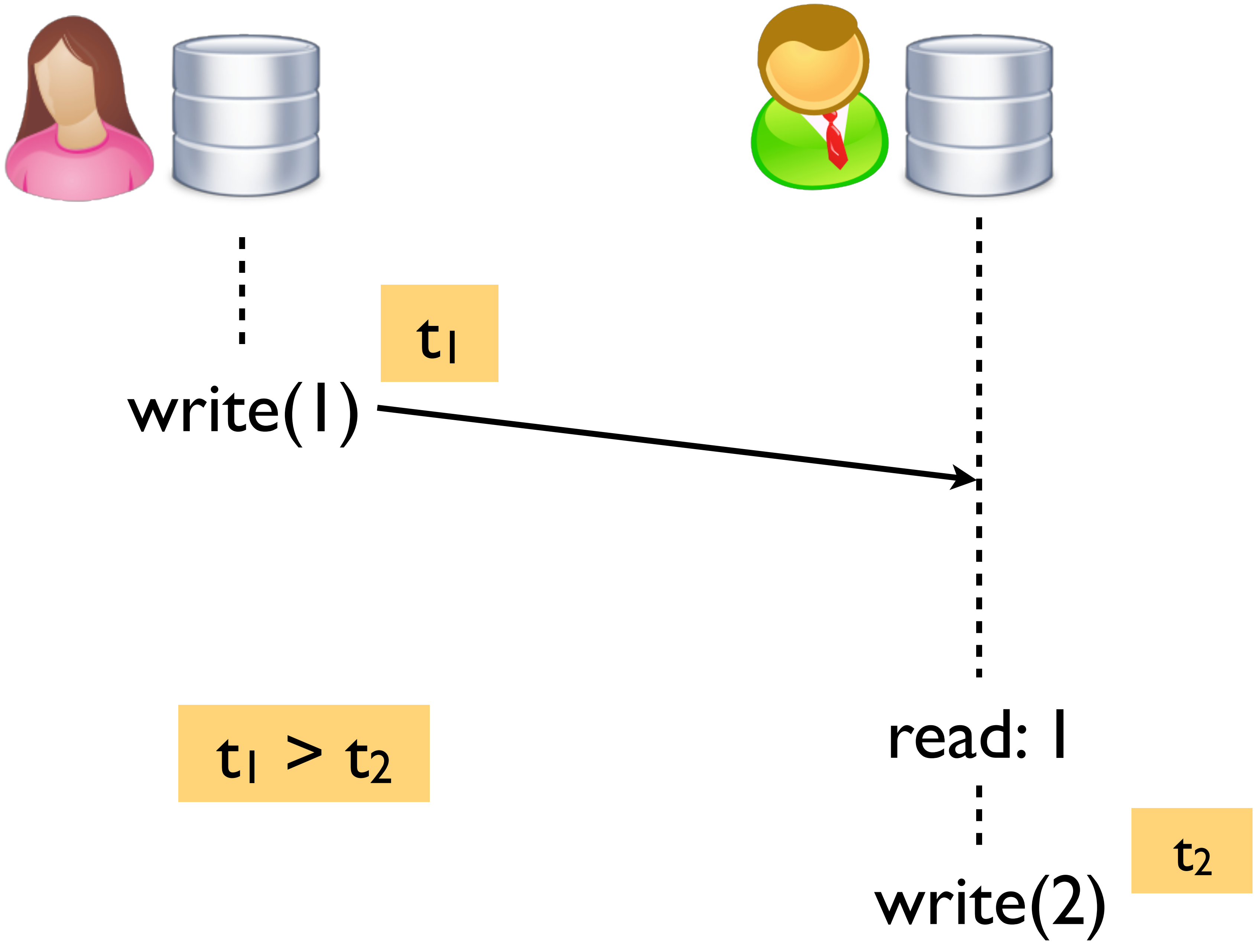
- Can use wall-clock time at the machine

- But can lead to strange results when clocks are out of sync

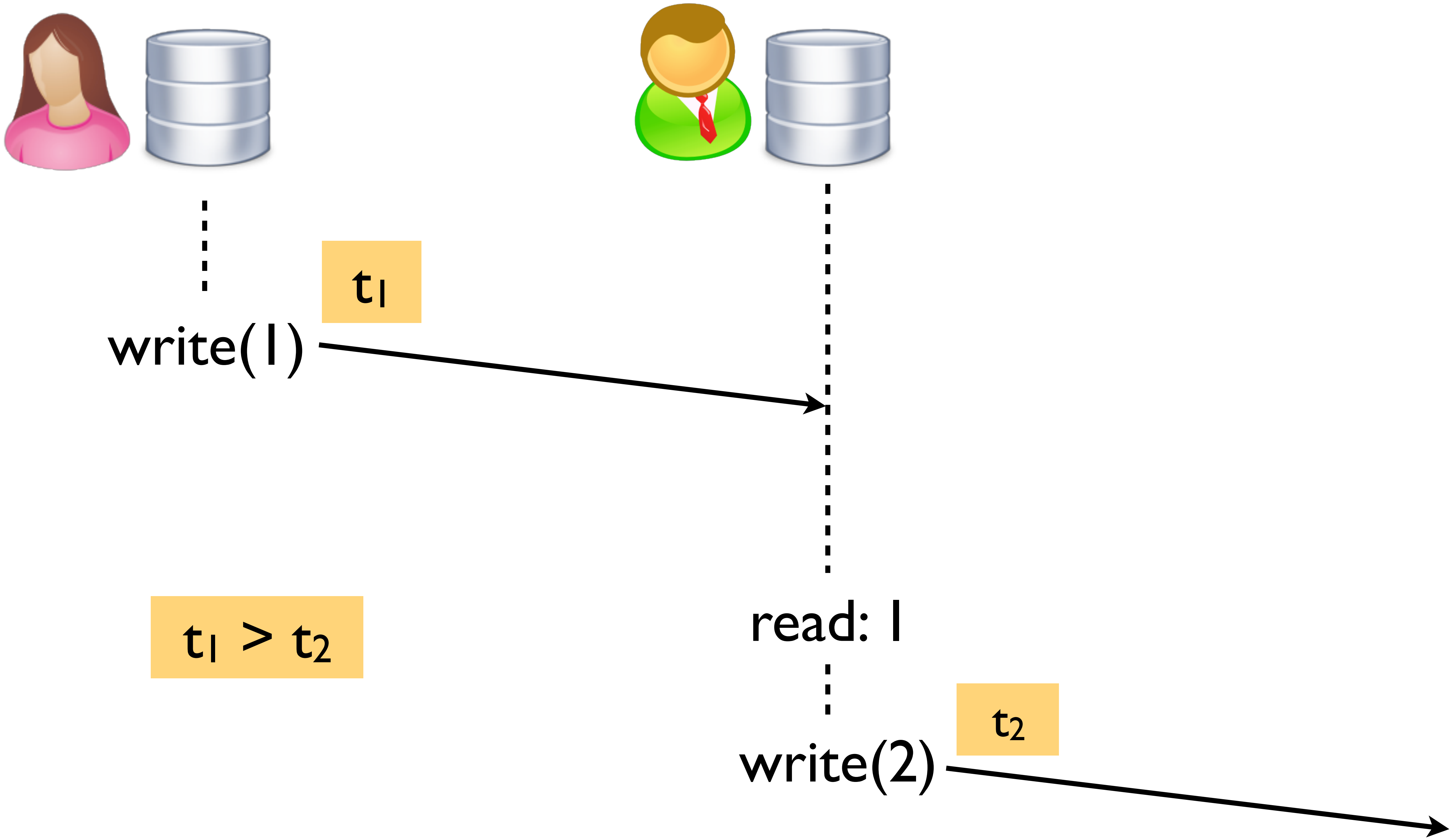

t1
write(1)

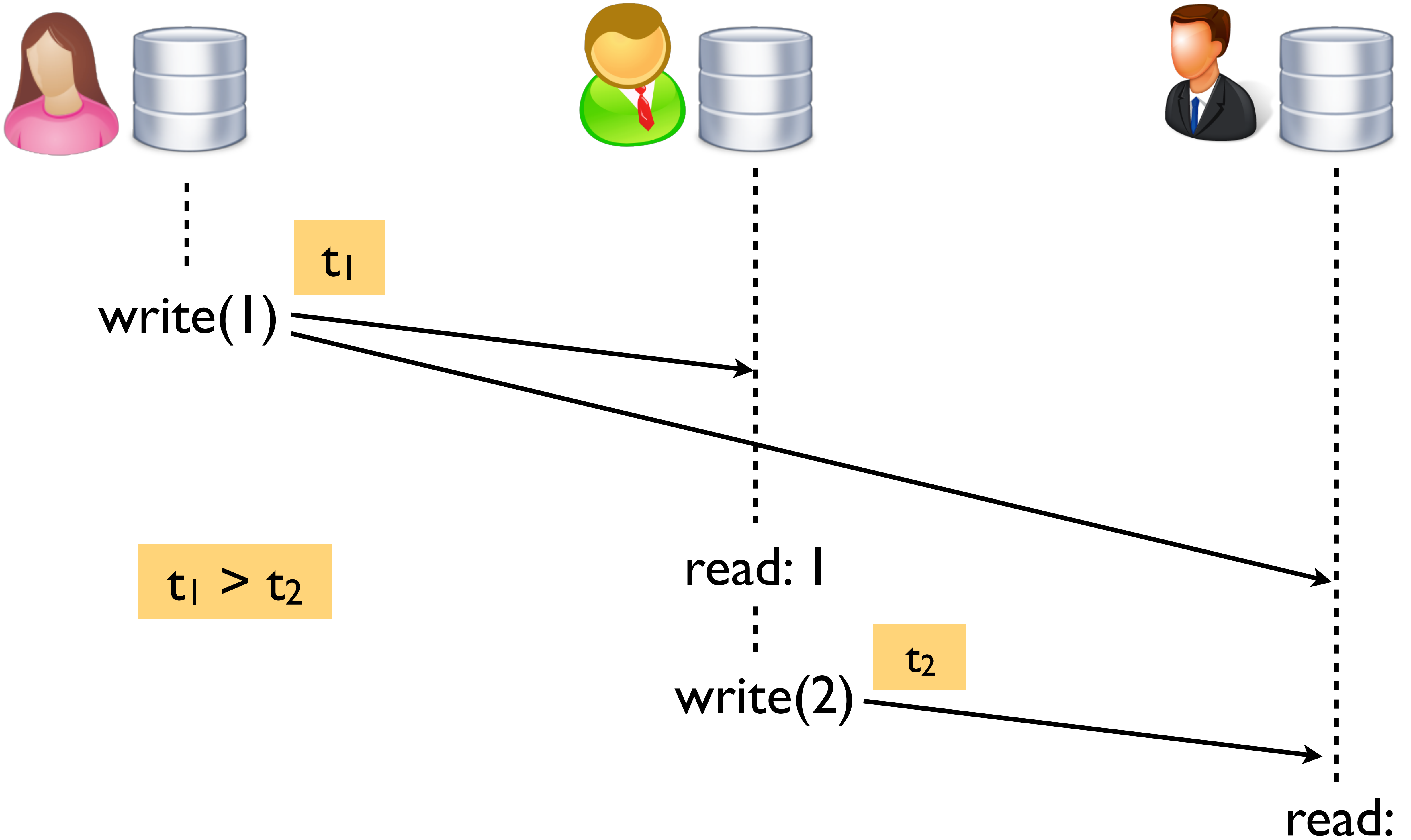


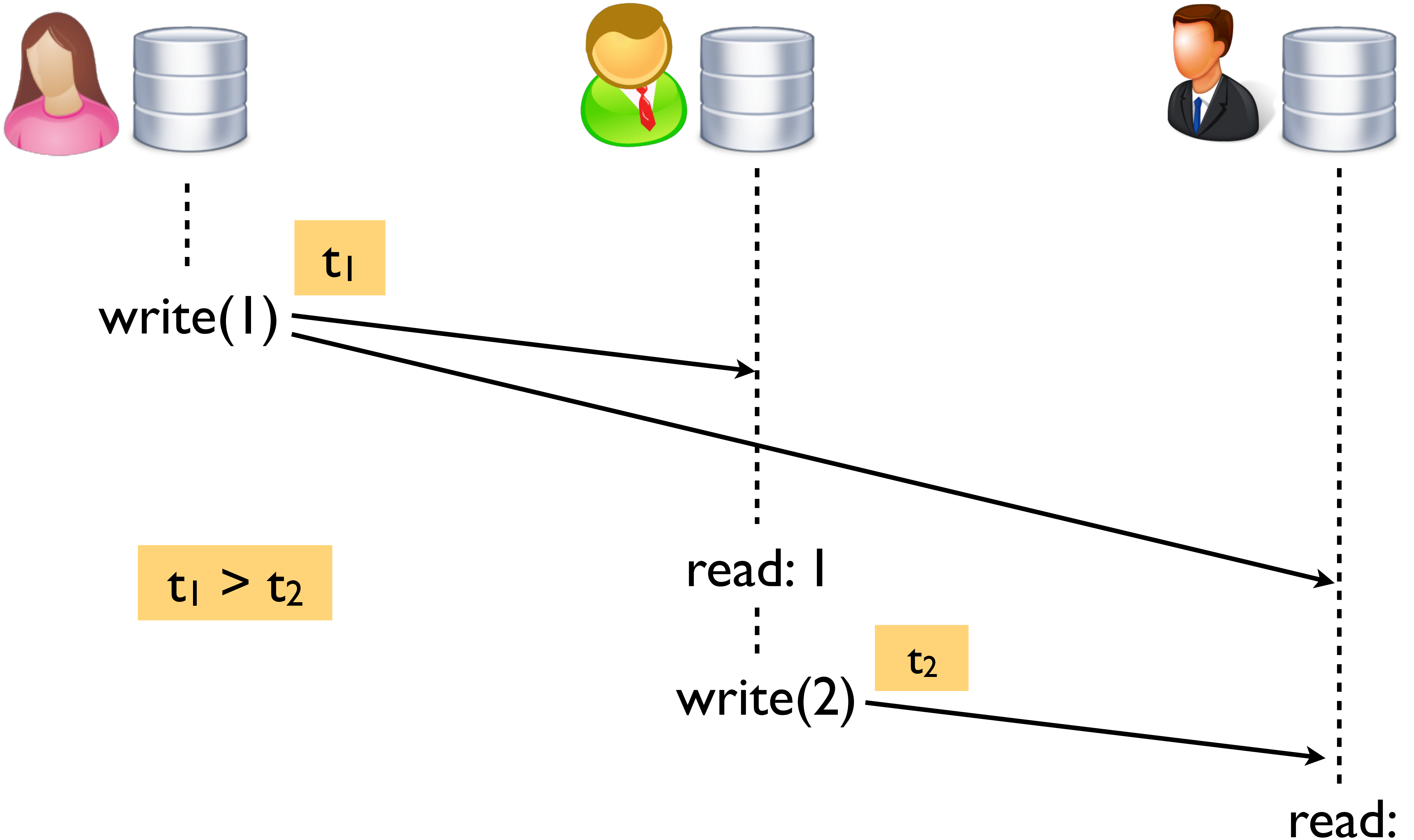
t1
write(1)
read: 1
write(2)



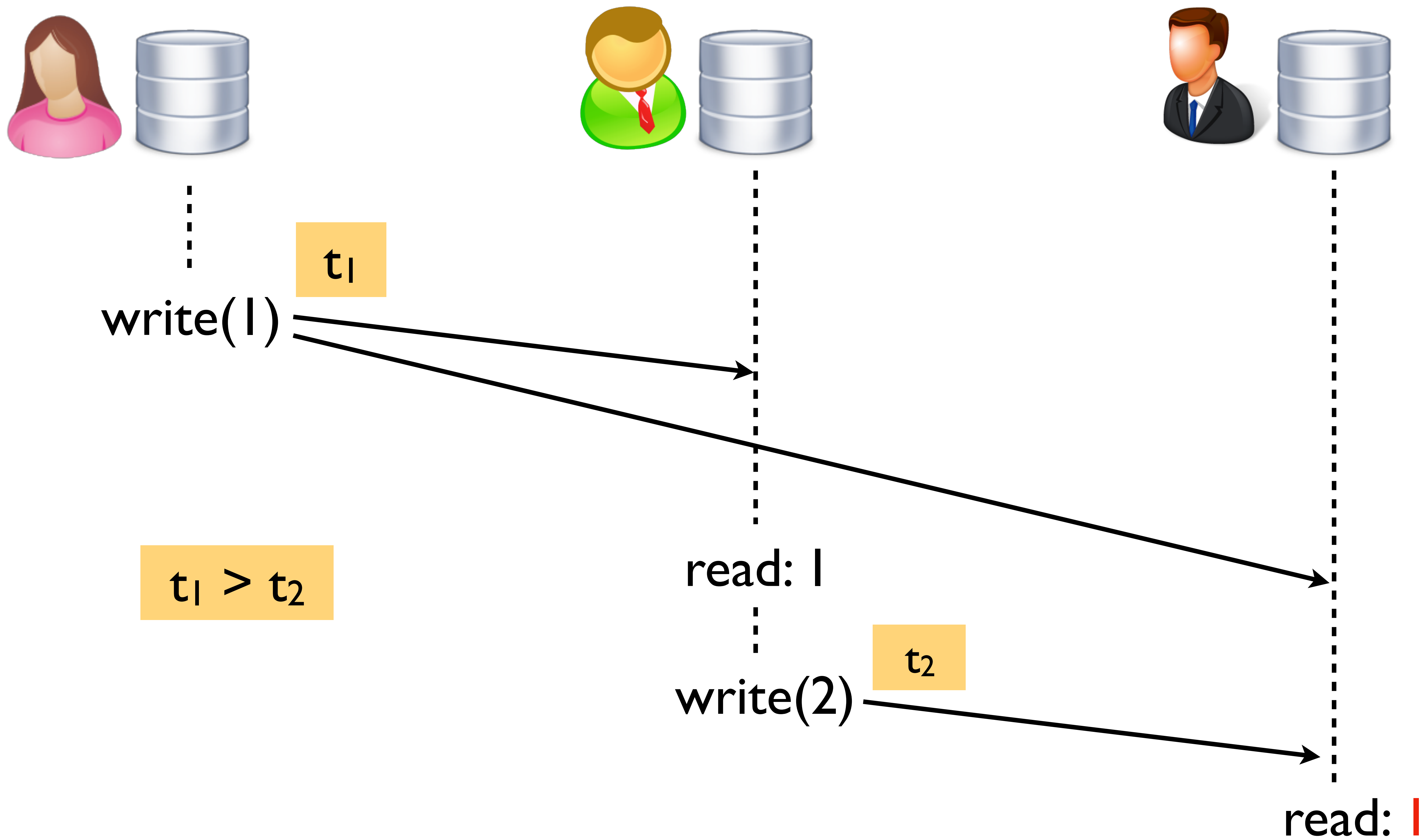
t_1
write(1)
$t_1 > t_2$
read: 1
t_2
write(2)


t_1
write(1)
read: 1
$t_1 > t_2$
t_2
write(2)


t1
write(1)
t1 > t2
read: 1
t2
write(2)
read: 1



- Undesirable: 2 was meant to supersede 1



- Undesirable: 2 was meant to supersede 1
- Use logical (Lamport) clocks instead

Lamport clock

Replica maintains a counter, incremented on each operation:

time = 1

Lamport clock

Replica maintains a counter, incremented on each operation:

time = 1

write(1) 1

time = 2

Lamport clock

Replica maintains a counter, incremented on each operation:

time = 1

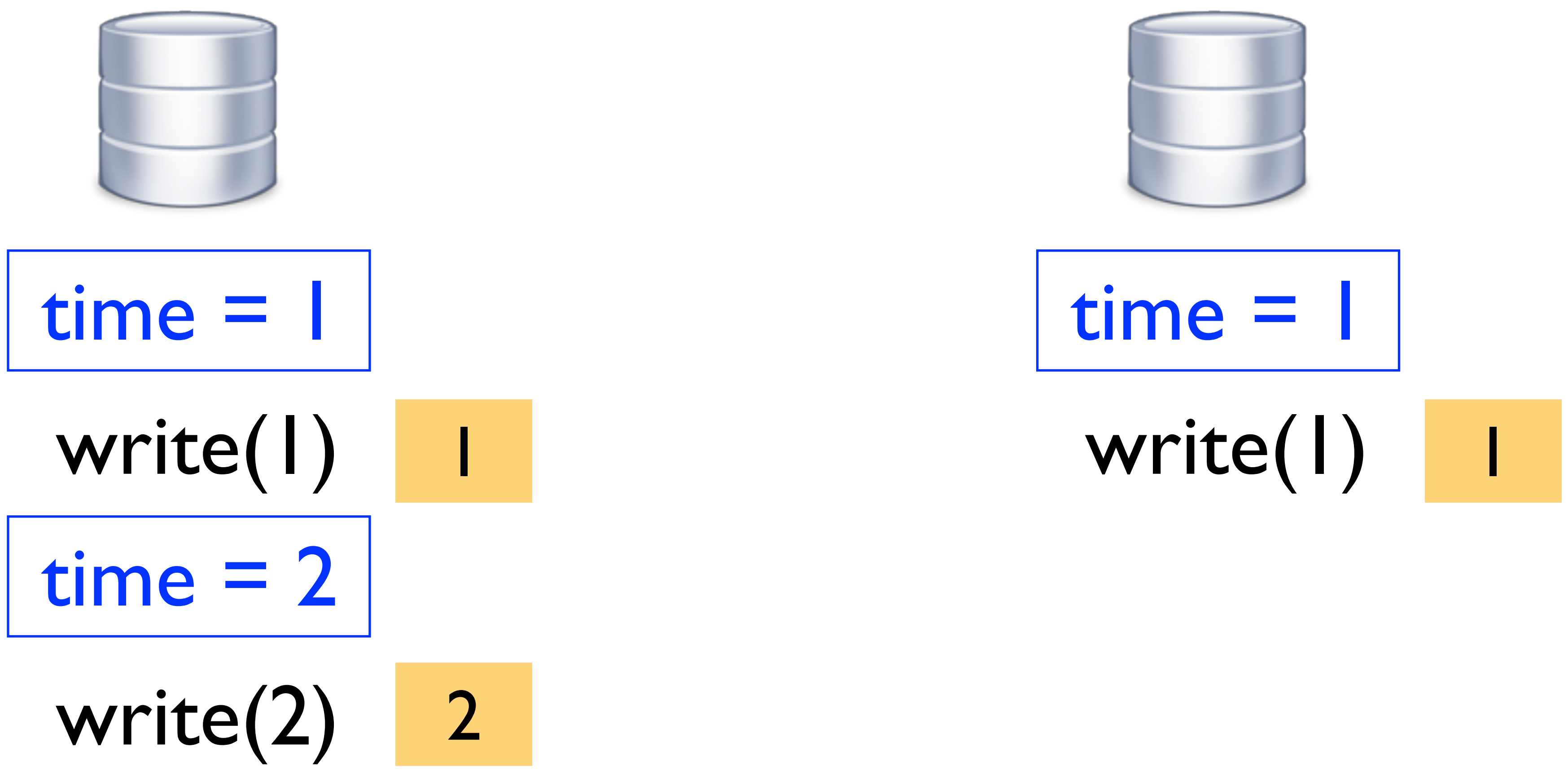
write(1) 1

time = 2

write(2) 2

Lamport clock

Replica maintains a counter, incremented on each operation:



Lamport clock

Replica maintains a counter, incremented on each operation:

time = 1

write(1) 1

time = 2

write(2) 2

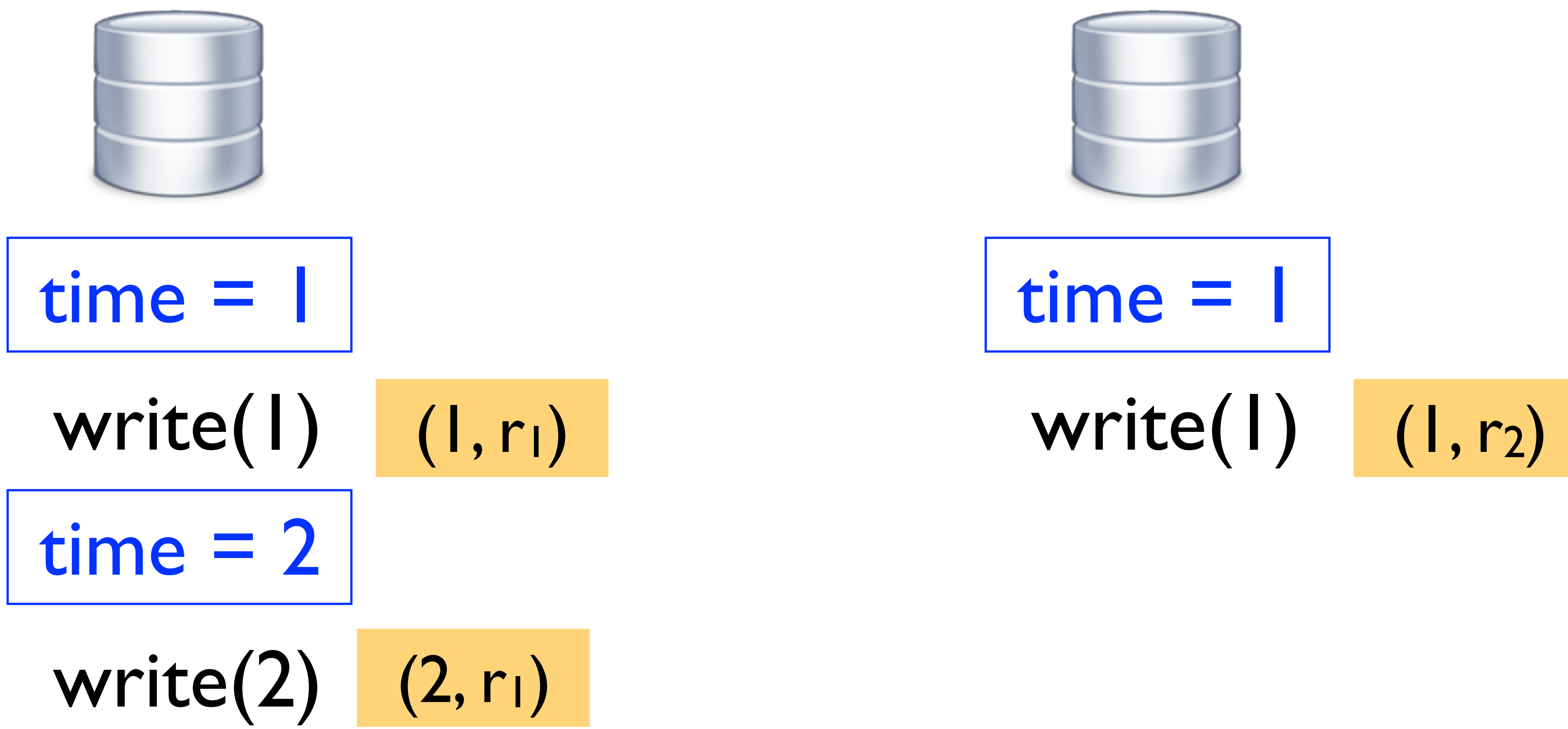
time = 1

write(1) 1

Timestamps need to be unique: ts = (CounterValue, ReplicaID)

Lamport clock

Replica maintains a counter, incremented on each operation:



Timestamps need to be unique: ts = (CounterValue, ReplicaID)

Lamport clock

Replica maintains a counter, incremented on each operation:

time = 1

write(1) $(1, r_1)$

time = 2

write(2) $(2, r_1)$

time = 1

write(1) $(1, r_2)$

Timestamps need to be unique: ts = (CounterValue, ReplicaID)

$$(c_1, r_1) < (c_2, r_2) \iff c_1 < c_2 \lor (c_1 = c_2 \land r_1 < r_2)$$

Lamport clock

Replica maintains a counter, incremented on each operation:

time = 1

write(1) $(1, r_1)$

time = 2

write(2) $(2, r_1)$

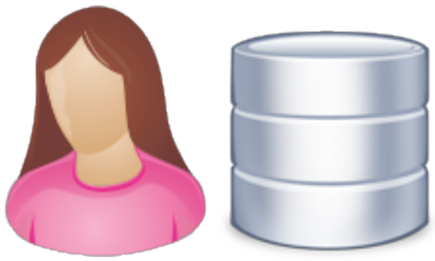
time = 1

write(1) $(1, r_2)$

$(1, r_1) < (1, r_2)$

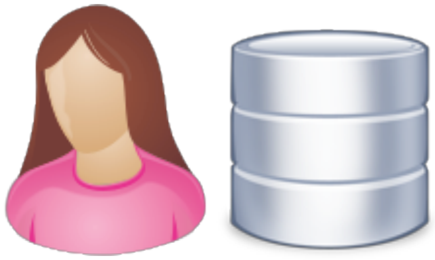
Timestamps need to be unique: ts = (CounterValue, ReplicaID)

$(c_1, r_1) < (c_2, r_2) \iff c_1 < c_2 \lor (c_1 = c_2 \land r_1 < r_2)$



time = t_1

write(1)

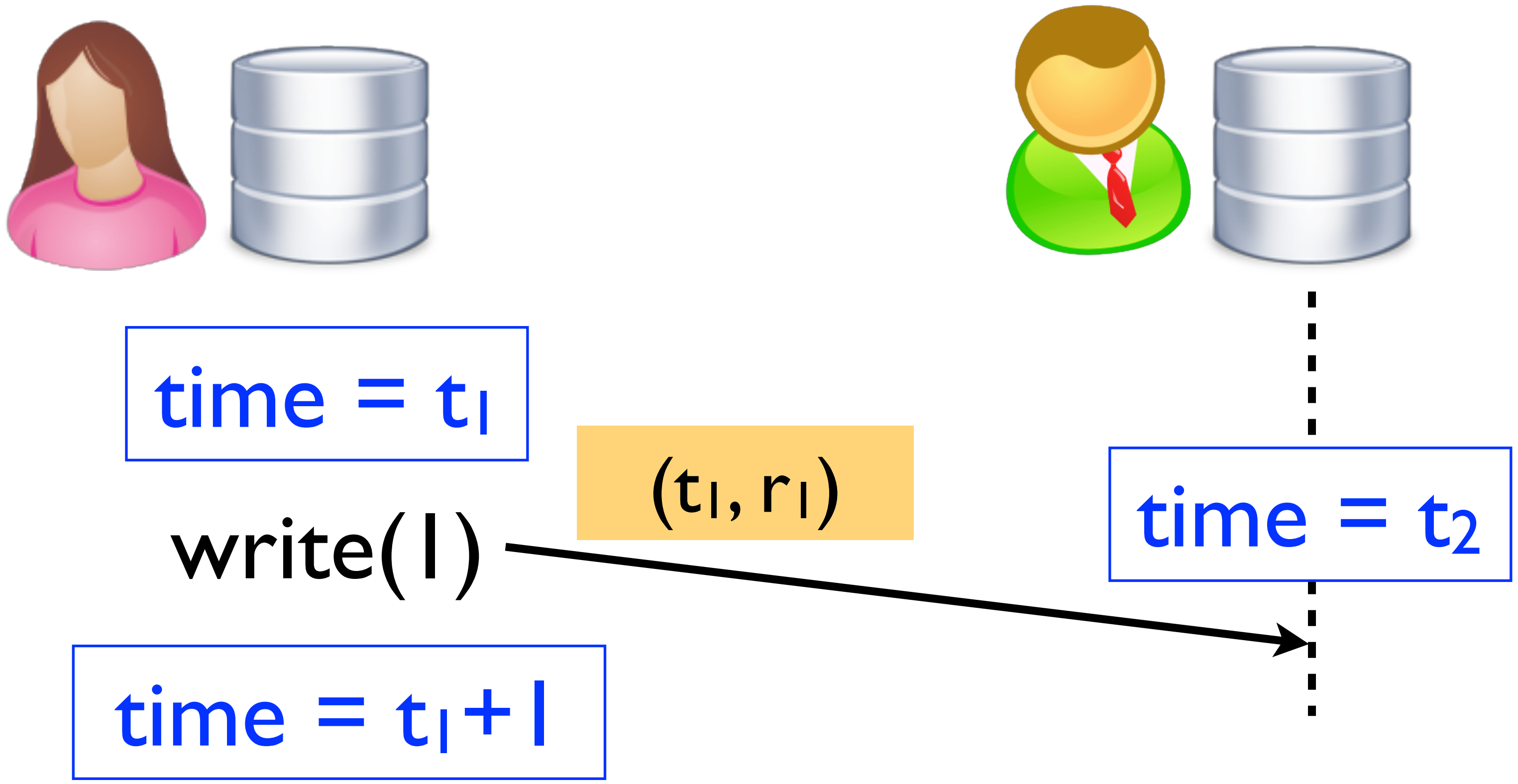


$\boxed{time = t_1}$

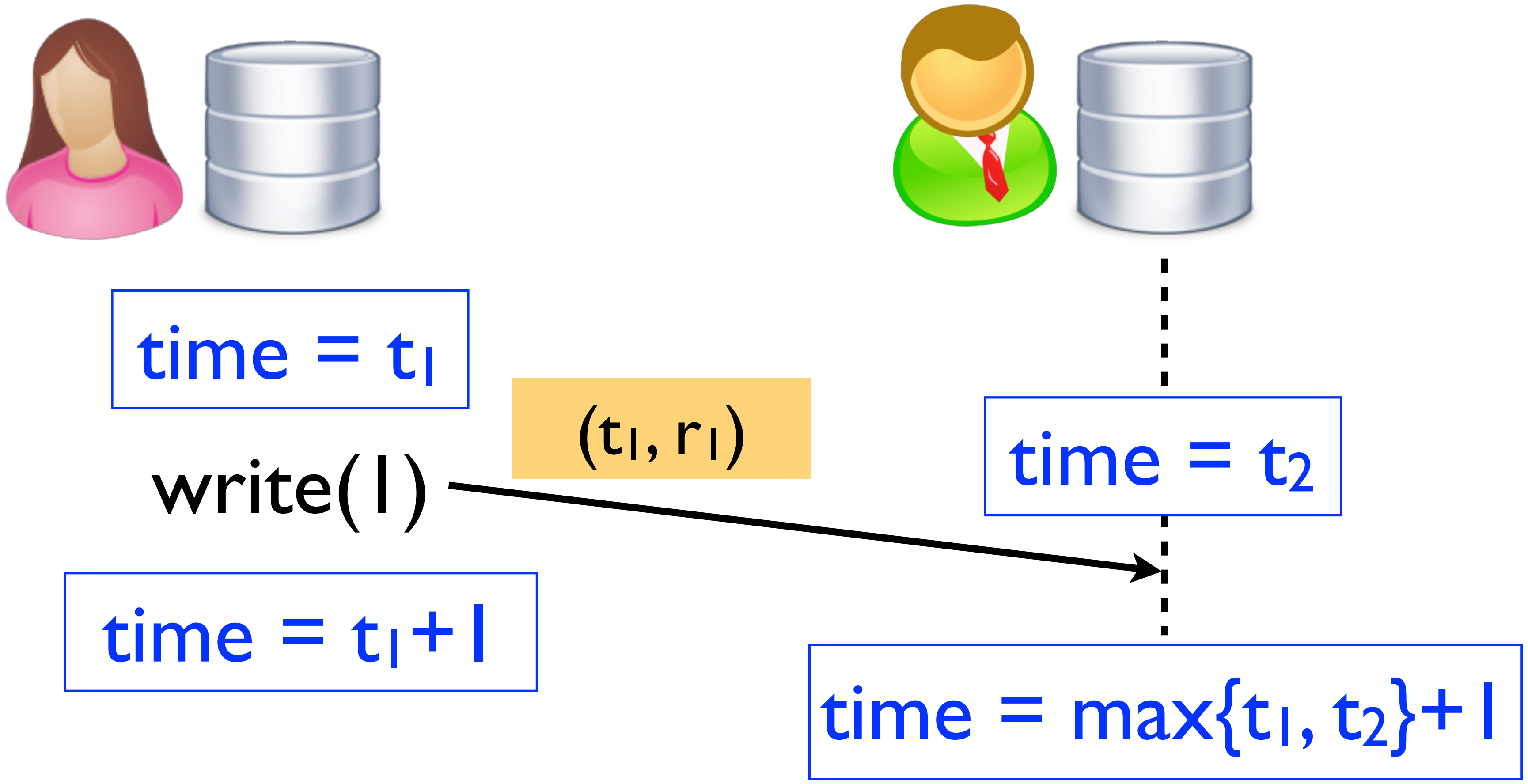
write(1)

(t_1, r_1)

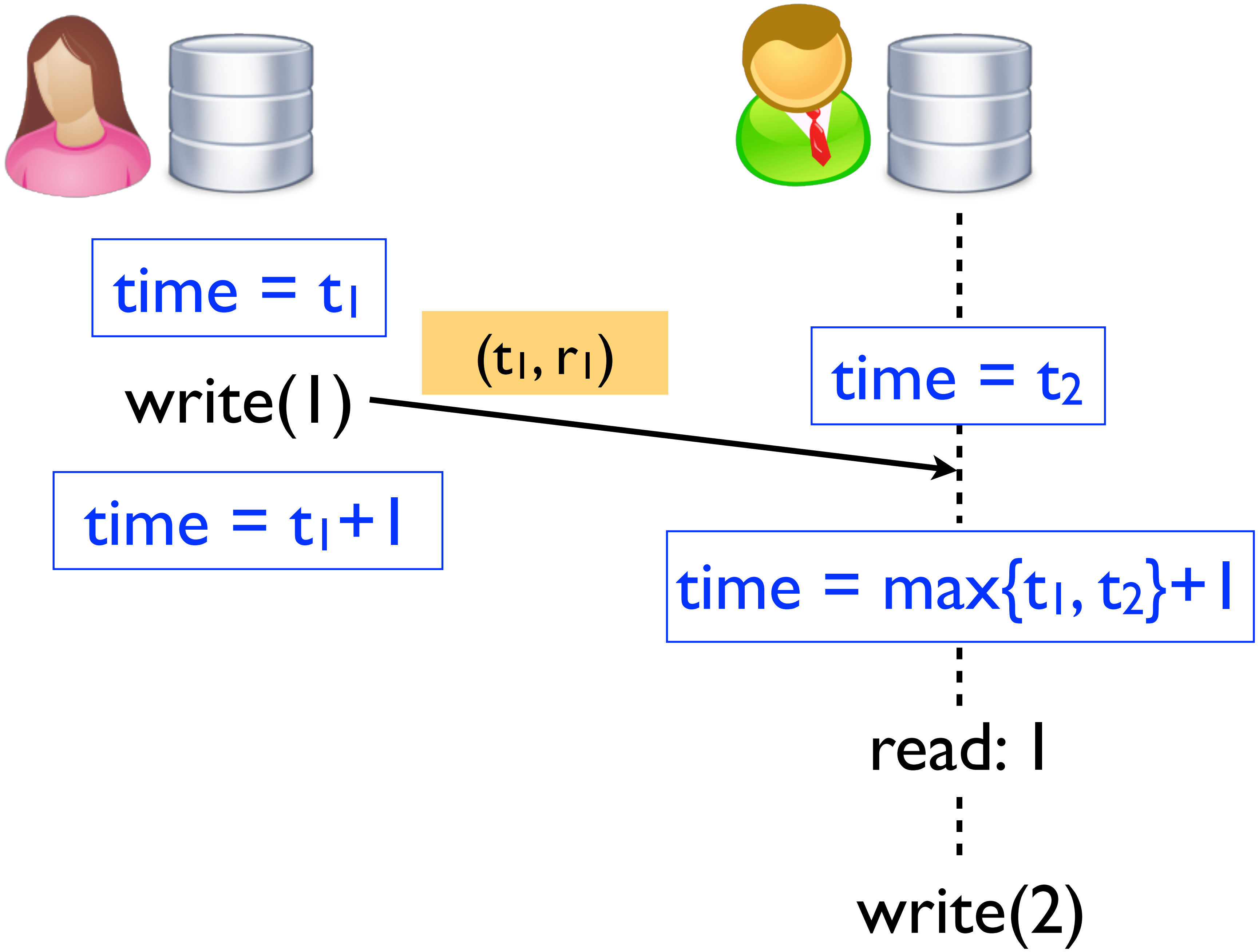
$\boxed{time = t_1 + 1}$



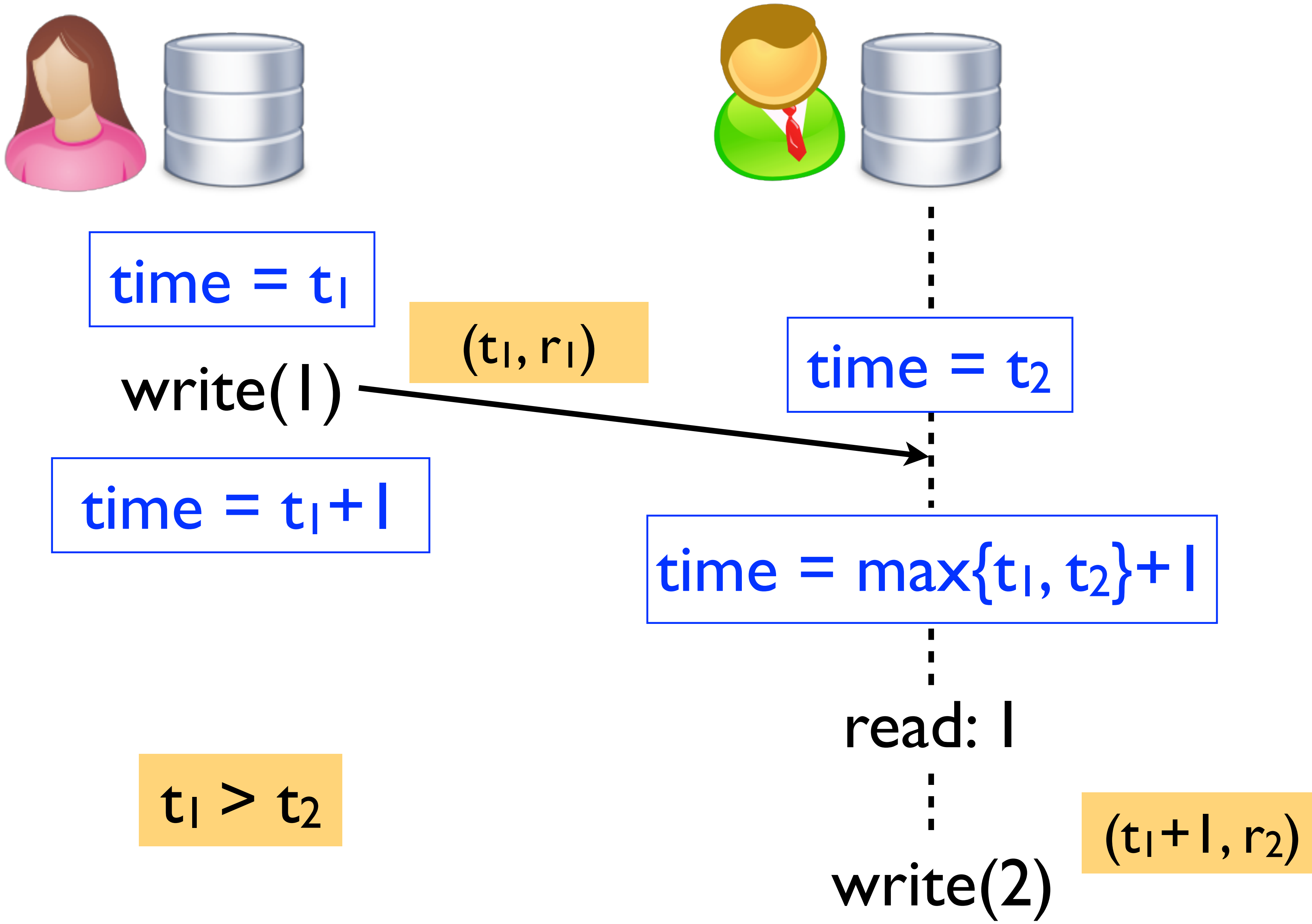
time = t1
(t1, r1)
time = t2
write(1)
time = t1+1



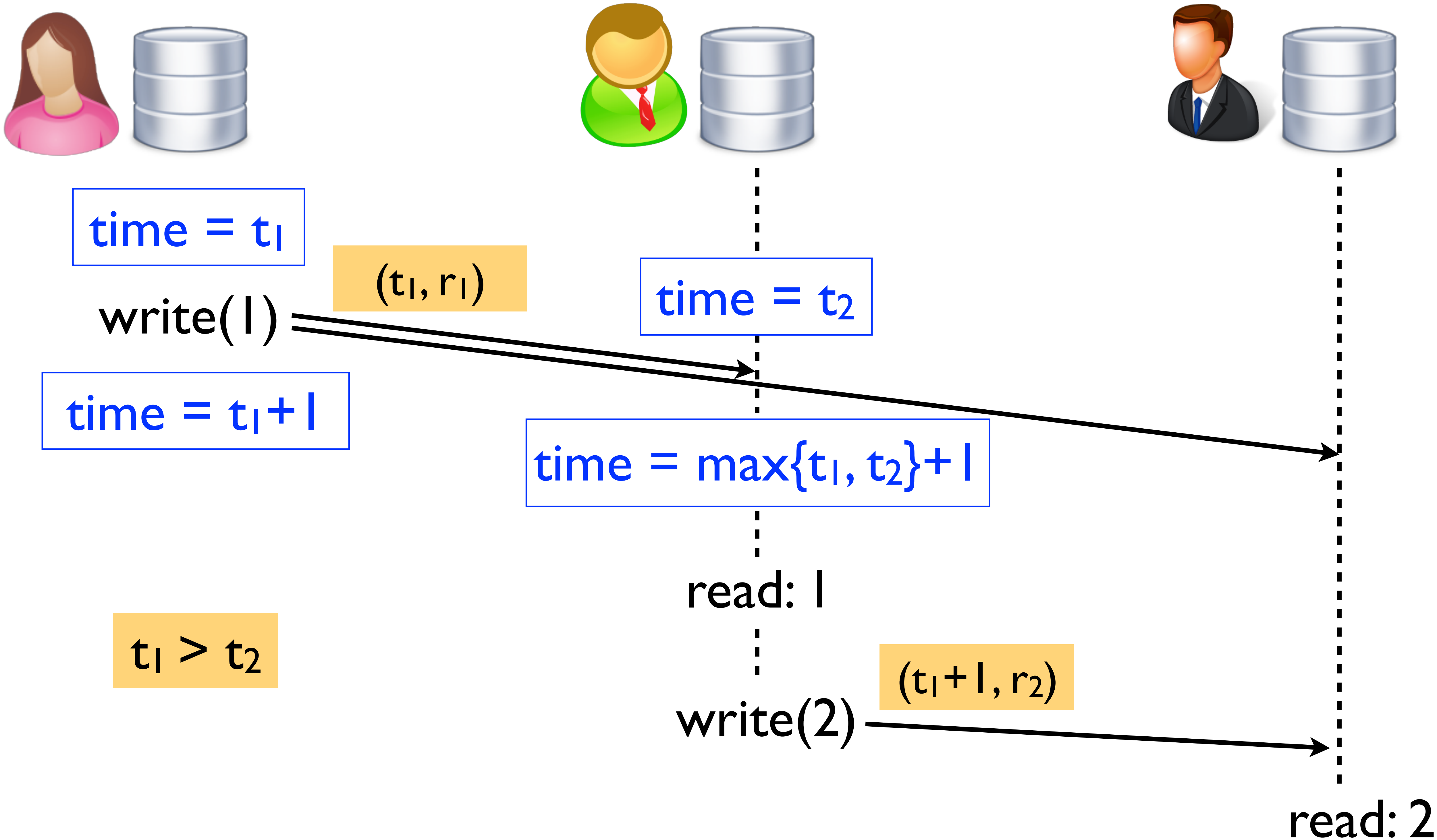
When receiving an effector, bump up your clock above its timestamp



When receiving an effector, bump up your clock above its timestamp

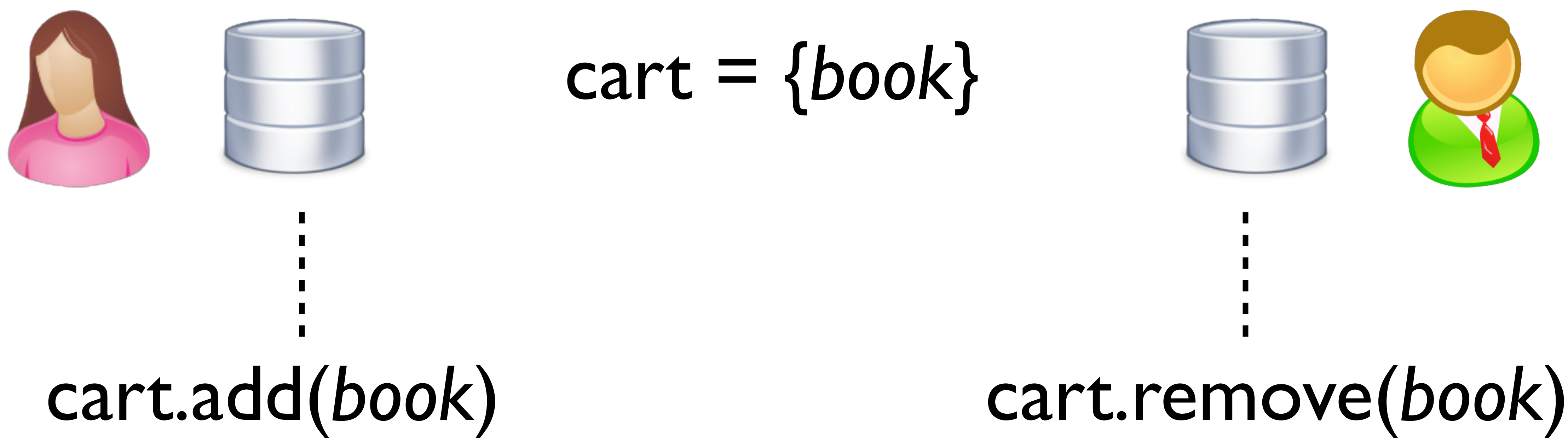


When receiving an effector, bump up your clock above its timestamp

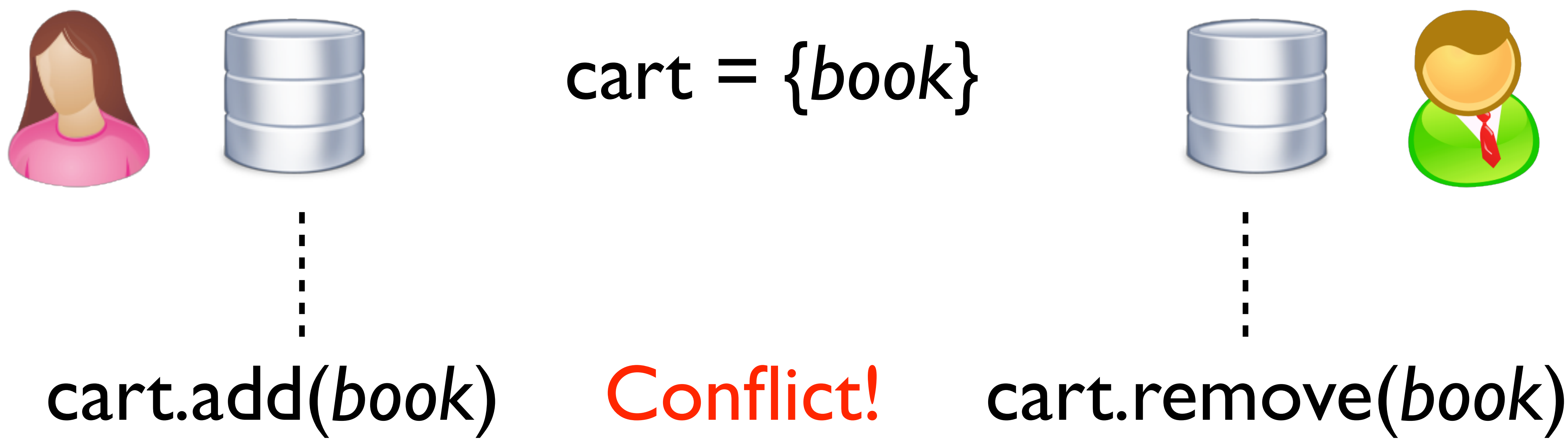


When receiving an effector, bump up your clock above its timestamp

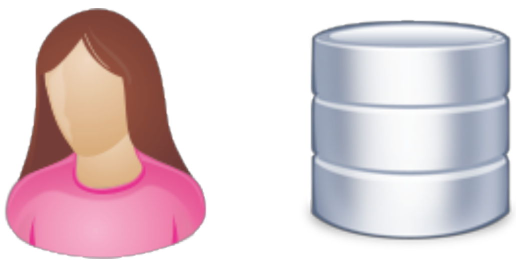
Replicated set



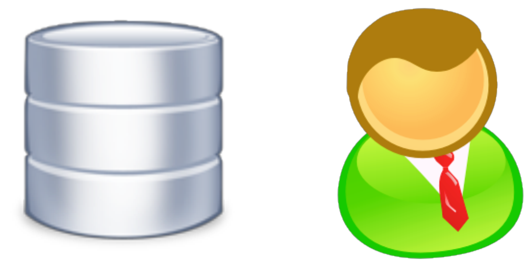
Replicated set



Replicated set



cart = {*book*}

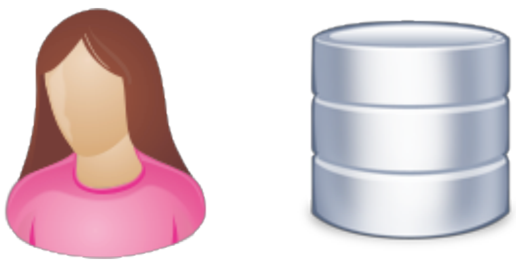


cart.add(*book*) Conflict! cart.remove(*book*)

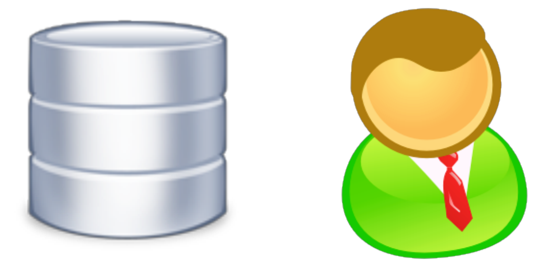
Should the remove cancel the concurrent add?

Depends on application requirements

Replicated set



cart = {*book*}



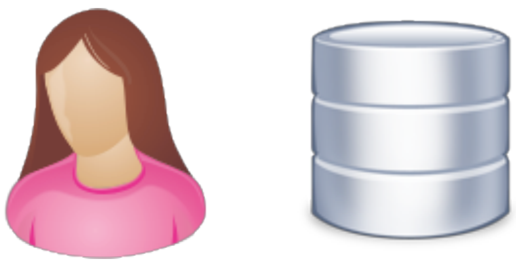
cart.add(*book*)   Conflict!   cart.remove(*book*)

Last writer wins: choose based on operation time-stamps

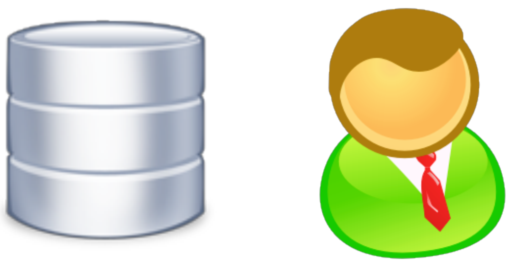
Remove wins: cart = $\varnothing$

Add wins: cart = {*book*}

Add-wins set



cart = {*book*}

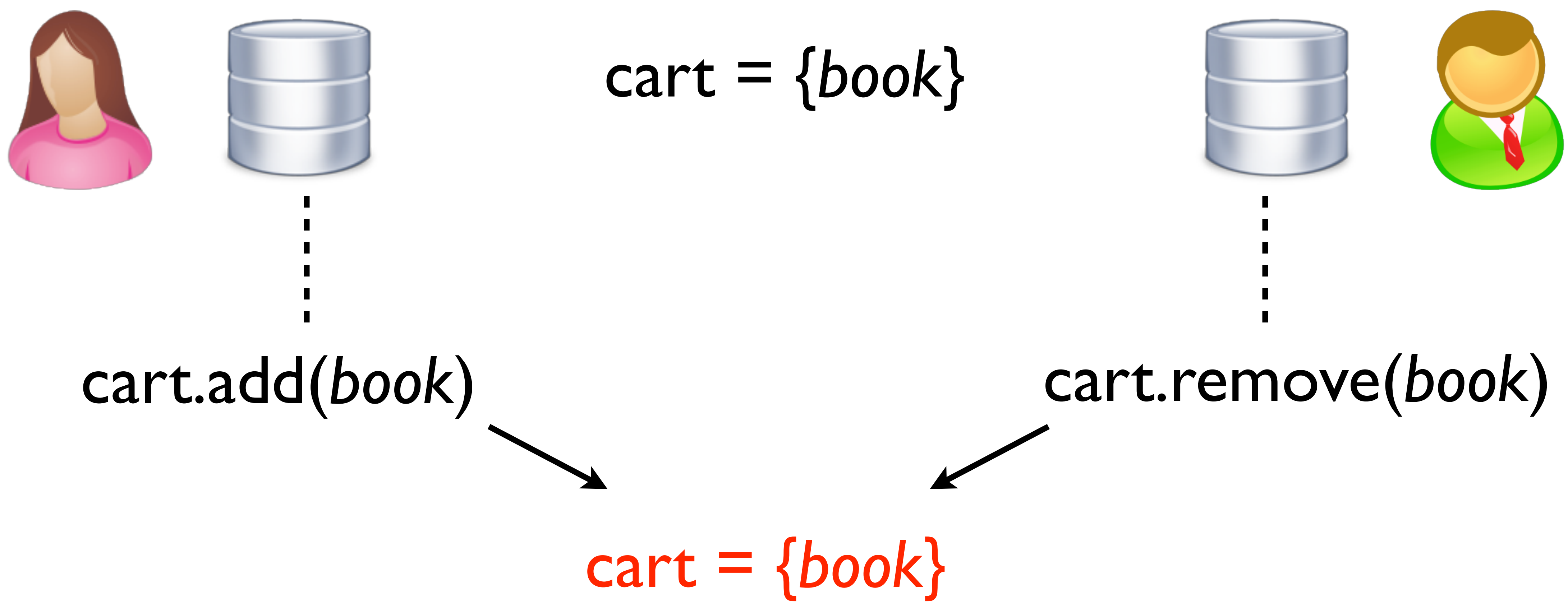


cart.add(*book*)

cart.remove(*book*)

cart = {*book*}

Add-wins set



- remove() acts differently wrt add() depending on whether it's concurrent or not
- Each addition creates a new instance:
  State = set of pairs (element, unique id)

$\{(\mathit{book}, 1)\}$

add(book)

Each add() creates a new element instance:

$$[\![\mathrm{add}(v)]\!]_{\mathrm{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma' \cup \{(v, \mathit{uniqueid}())\})$$

$\{(book, 1)\}$

$\mathrm{add}(book)$ $\lambda\sigma'.\, \sigma' \cup \{(book, 2)\}$

$\{(book, 1), (book, 2)\}$

Each add() creates a new element instance:

$$[\![\mathrm{add}(\mathrm{v})]\!]_{\mathrm{eff}}(\sigma) \;=\; \lambda\sigma'.\, (\sigma' \cup \{(\mathrm{v}, \mathit{uniqueid}())\})$$

$\{(\mathit{book},1)\}$

add(*book*)

$\{(\mathit{book},1), (\mathit{book},2)\}$

$\{(book, 1)\}$

add($book$)

$\{(book, 1), (book, 2)\}$

read() : $\{book\}$

Instance ids ignored when reading the set:

$$[\![\mathrm{read}()]\!]_{val}(\sigma) \;=\; \{v \mid \{\exists\, id.\, (v, id)\} \in \sigma\}$$

$\{(book, 1)\}$

add($book$)

$\{(book, 1), (book, 2)\}$

$\{(book, 1)\}$

remove($book$)

$\{(book, 1)\}$

add($book$)

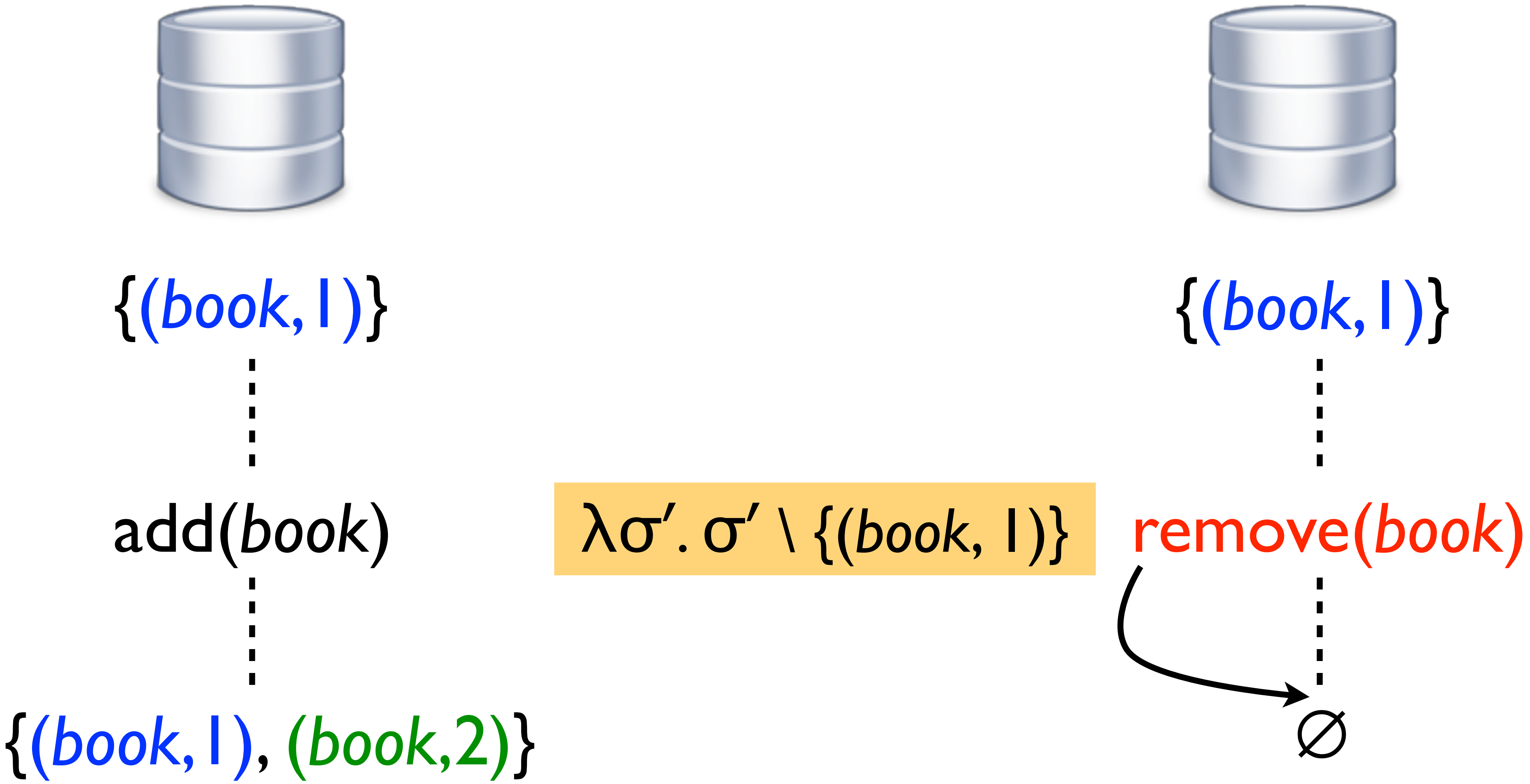
$\{(book, 1), (book, 2)\}$

$\{(book, 1)\}$

remove($book$)

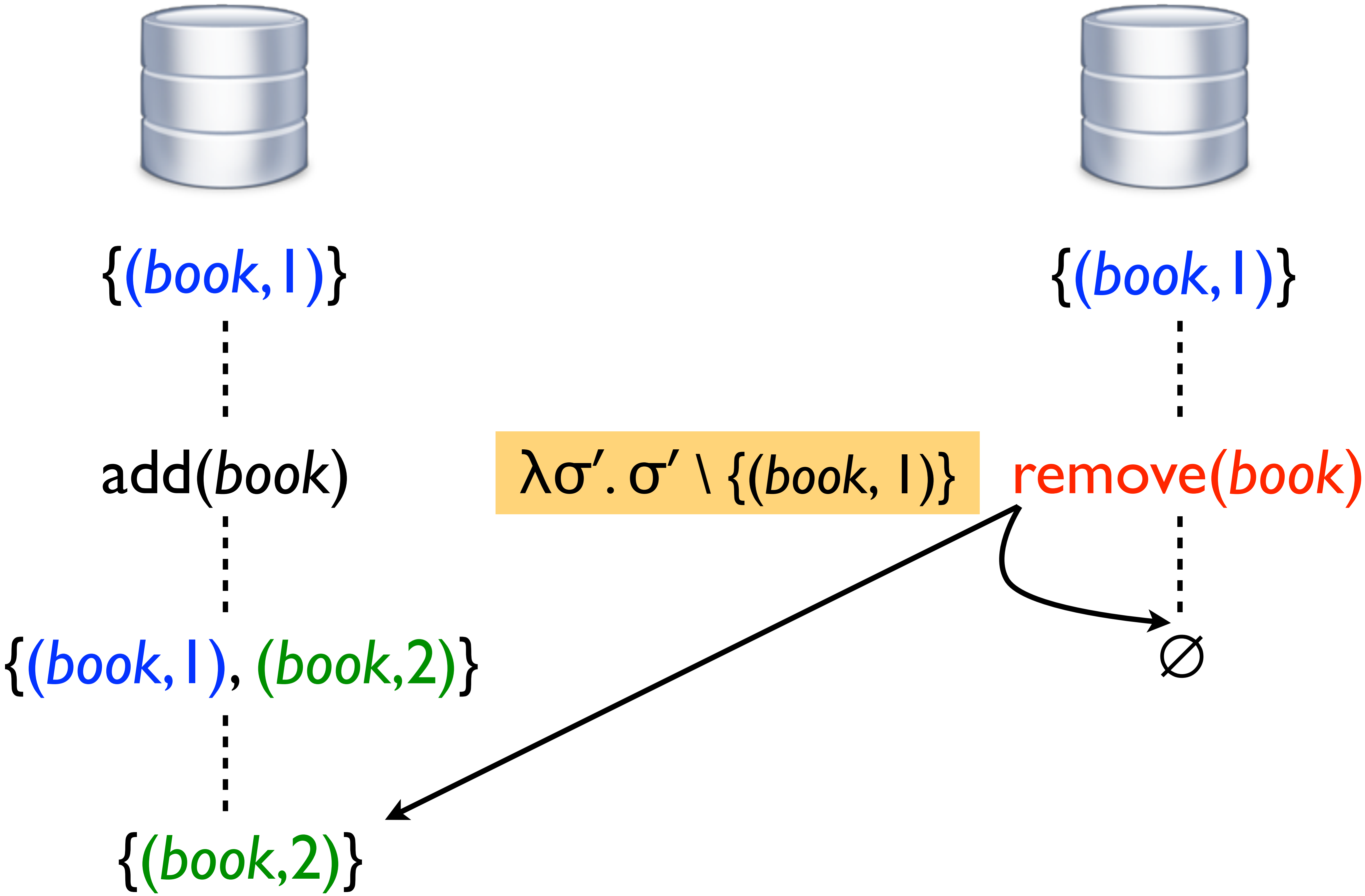
remove(v) removes all currently present instances of x:

$$[\![\mathrm{remove}(v)]\!]_{\mathrm{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma' \setminus \{(v, id) \in \sigma\})$$



remove(v) removes all currently present instances of x:

$$[\![\mathrm{remove}(v)]\!]_{\mathrm{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma' \setminus \{(v, id) \in \sigma\})$$



remove(v) removes all currently present instances of x:

$$[\![\text{remove}(v)]\!]_{\text{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma' \setminus \{(v, id) \in \sigma\})$$

$\{(\mathit{book}, 1)\}$

$\mathrm{add}(\mathit{book})$

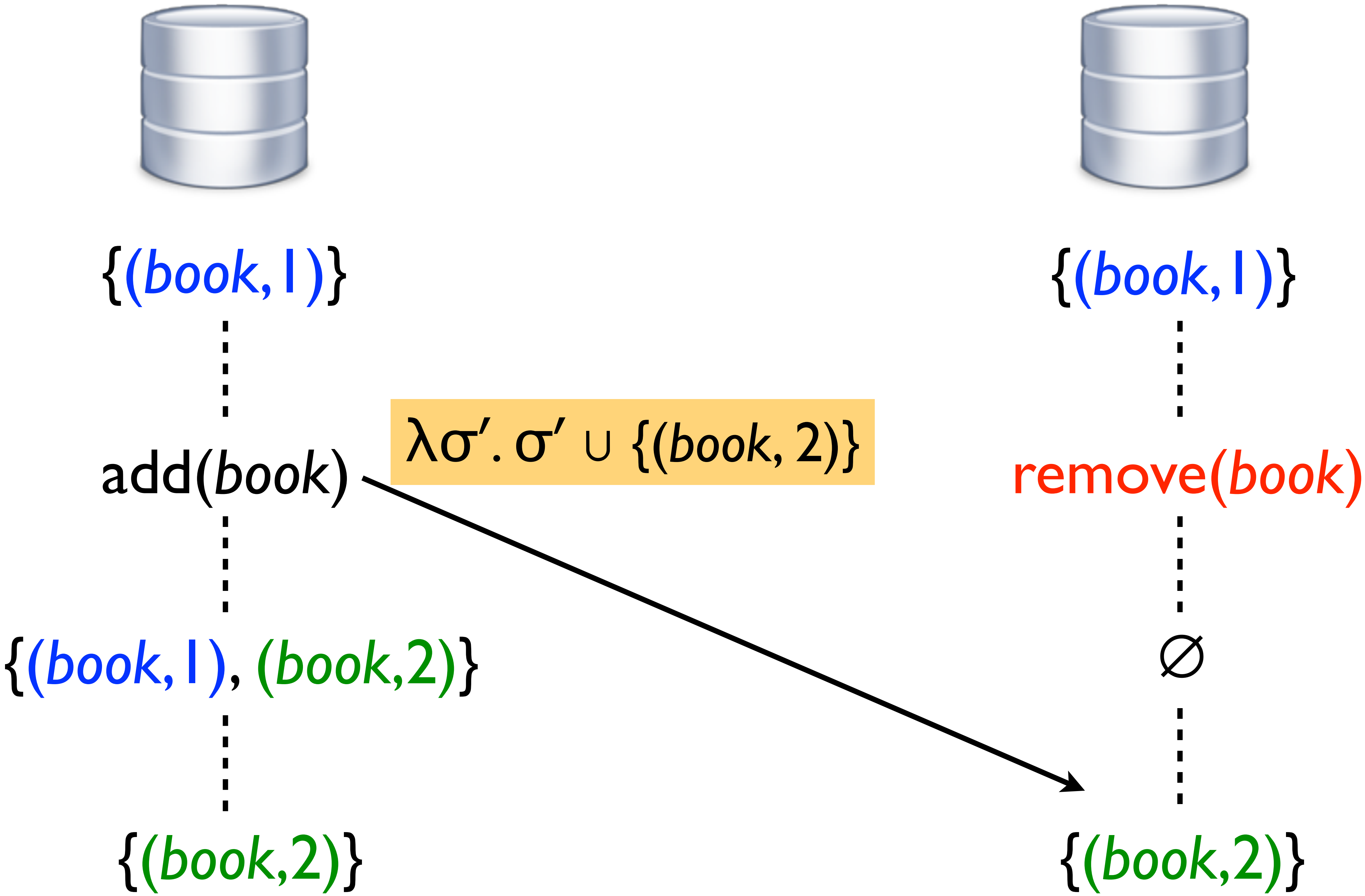
$\{(\mathit{book}, 1), (\mathit{book}, 2)\}$

$\{(\mathit{book}, 2)\}$

$\{(\mathit{book}, 1)\}$

$\mathrm{remove}(\mathit{book})$

$\varnothing$



Effectors commutative → replicas converge

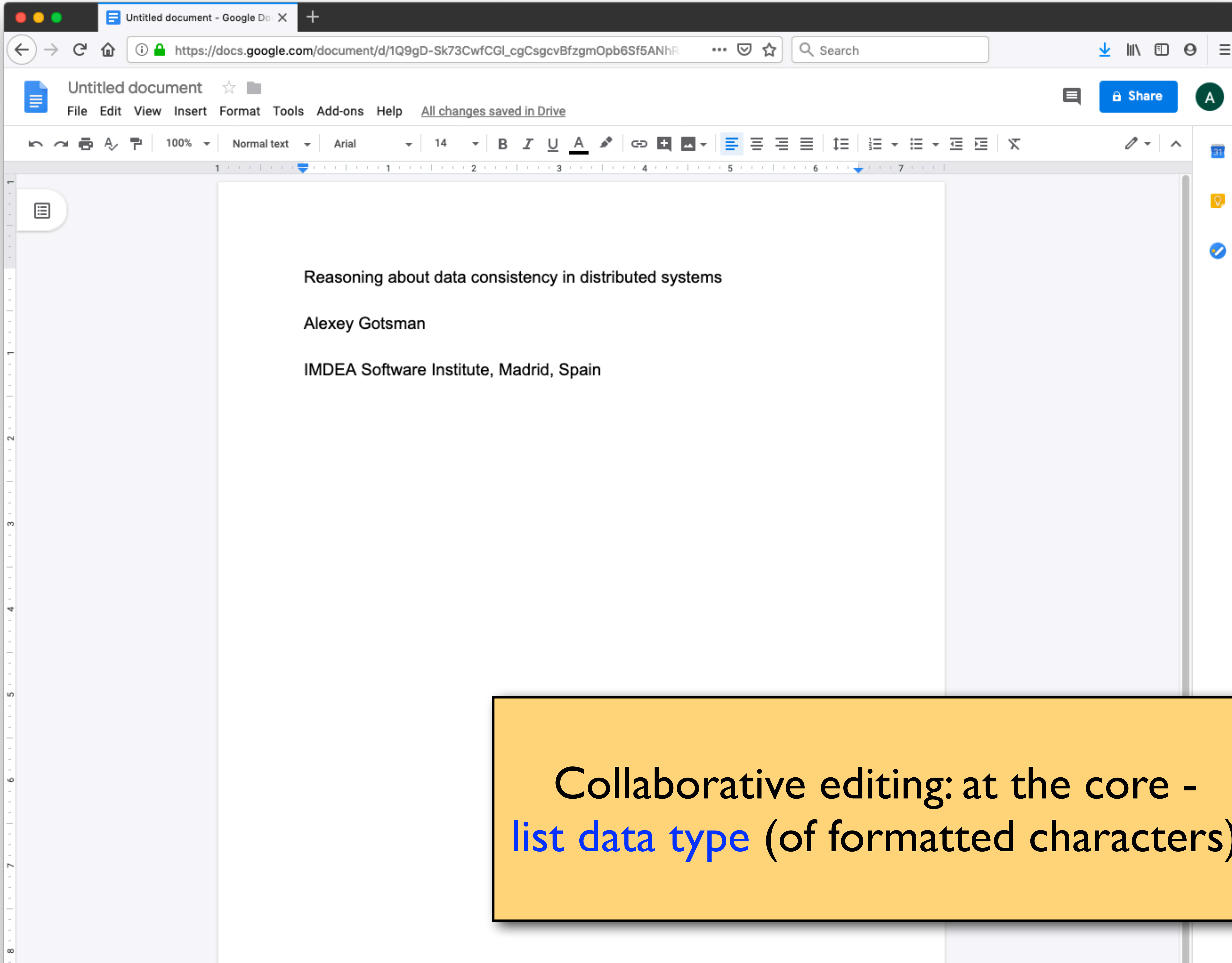
Take-aways

- Need to ensure commutativity to guarantee quiescent consistency
- Need to make choices about how to resolve conflicts

Replicated data type uses

- Provided by some data stores:

- Implemented by programmers on their own:



Collaborative editing: at the core - list data type (of formatted characters)

Operational specification

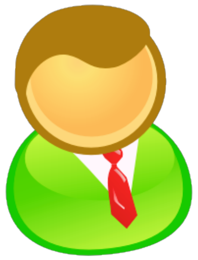
- Given a database with a set of objects of replicated data types

- Eventual consistency model = set of all histories produced by arbitrary client interactions with the data type implementations (with any allowed message deliveries)

- Implies quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

Eventual consistency and replicated data types, axiomatically

Anomalies

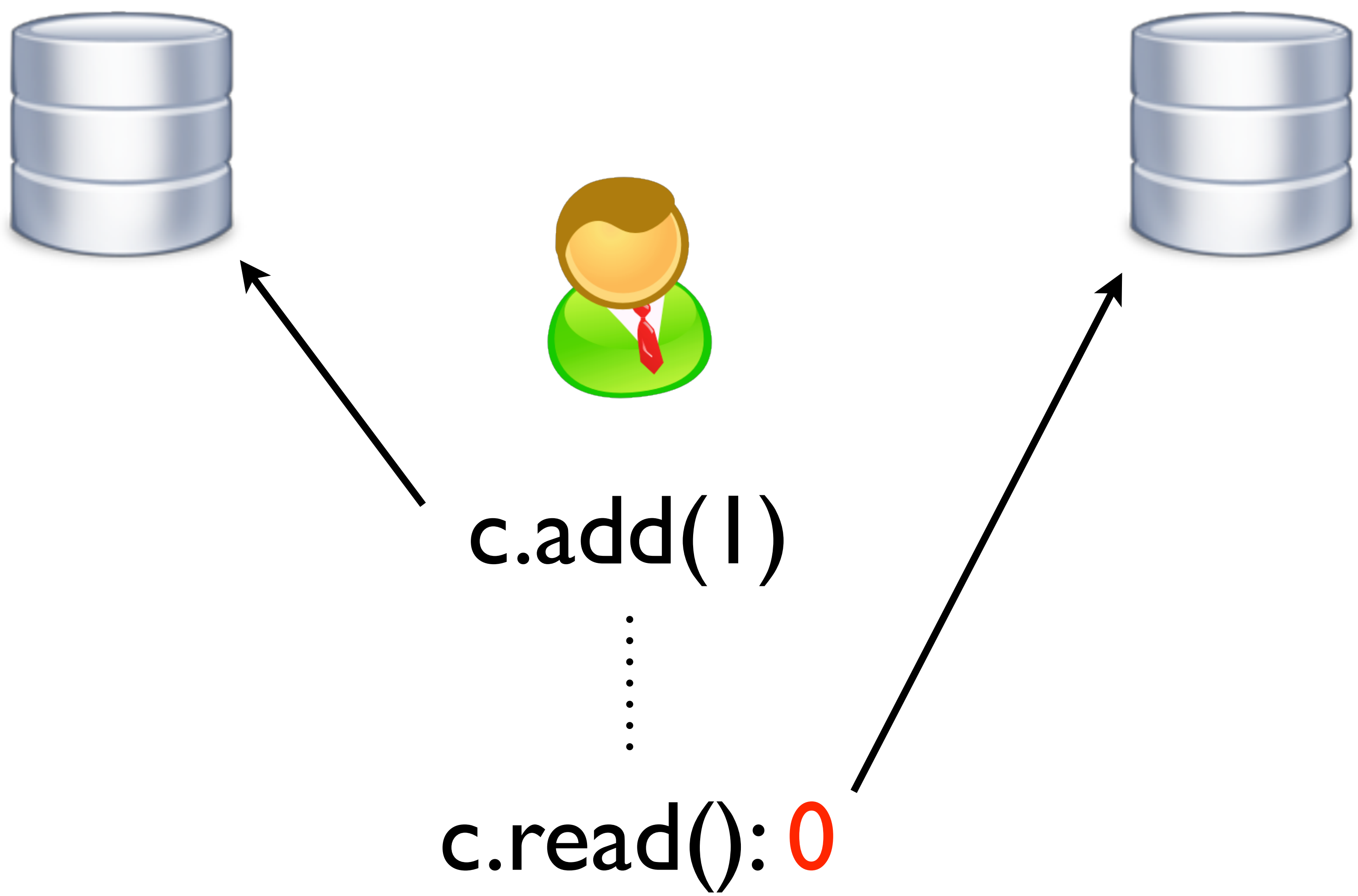


c.add(1)

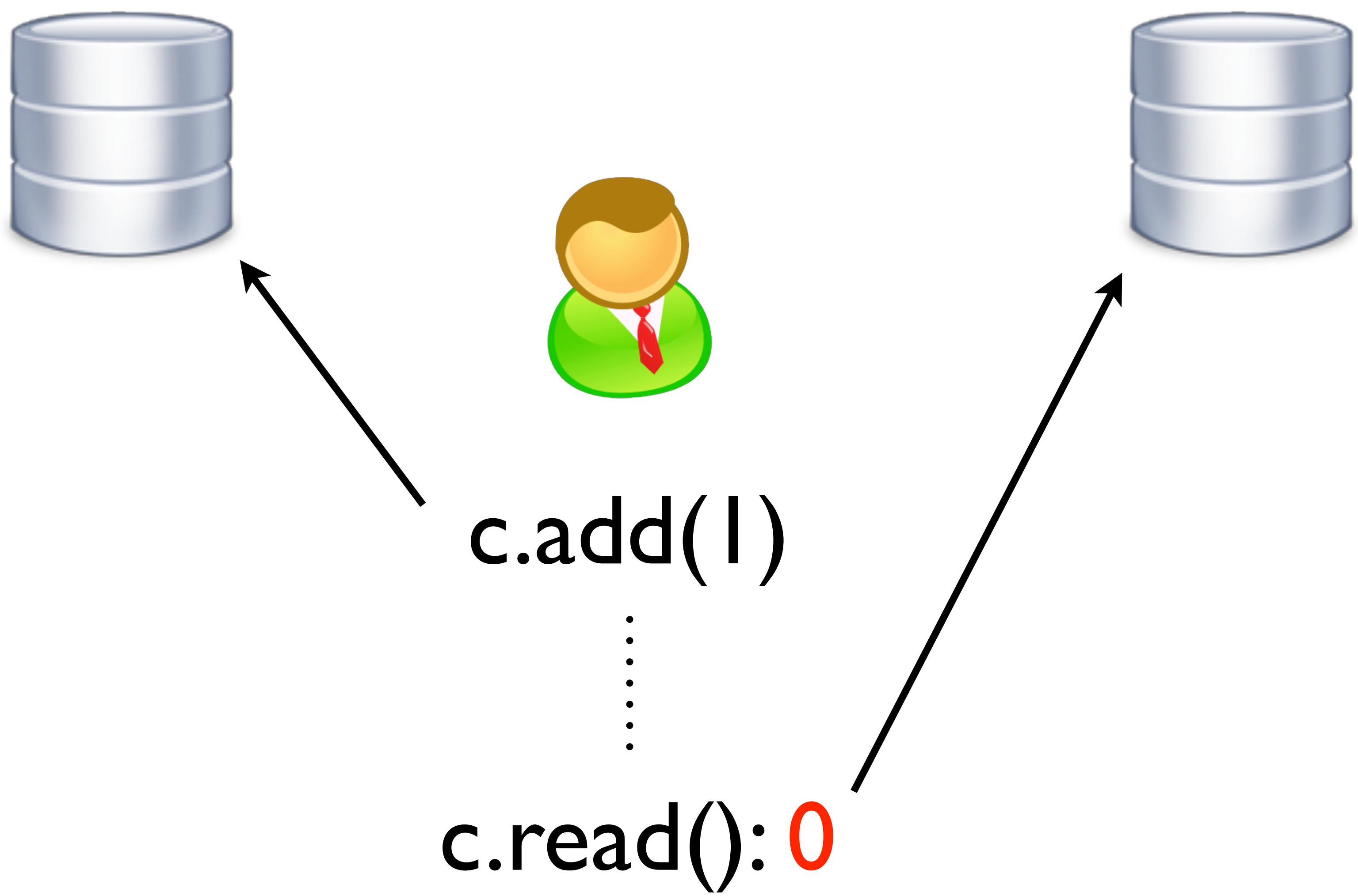
⋮

c.read(): ?

Anomalies



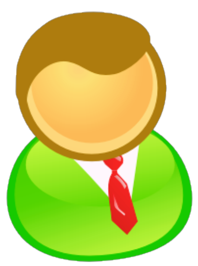
Anomalies



Can be disallowed if the client sticks to the same replica:
Read Your Writes guarantee

Anomalies

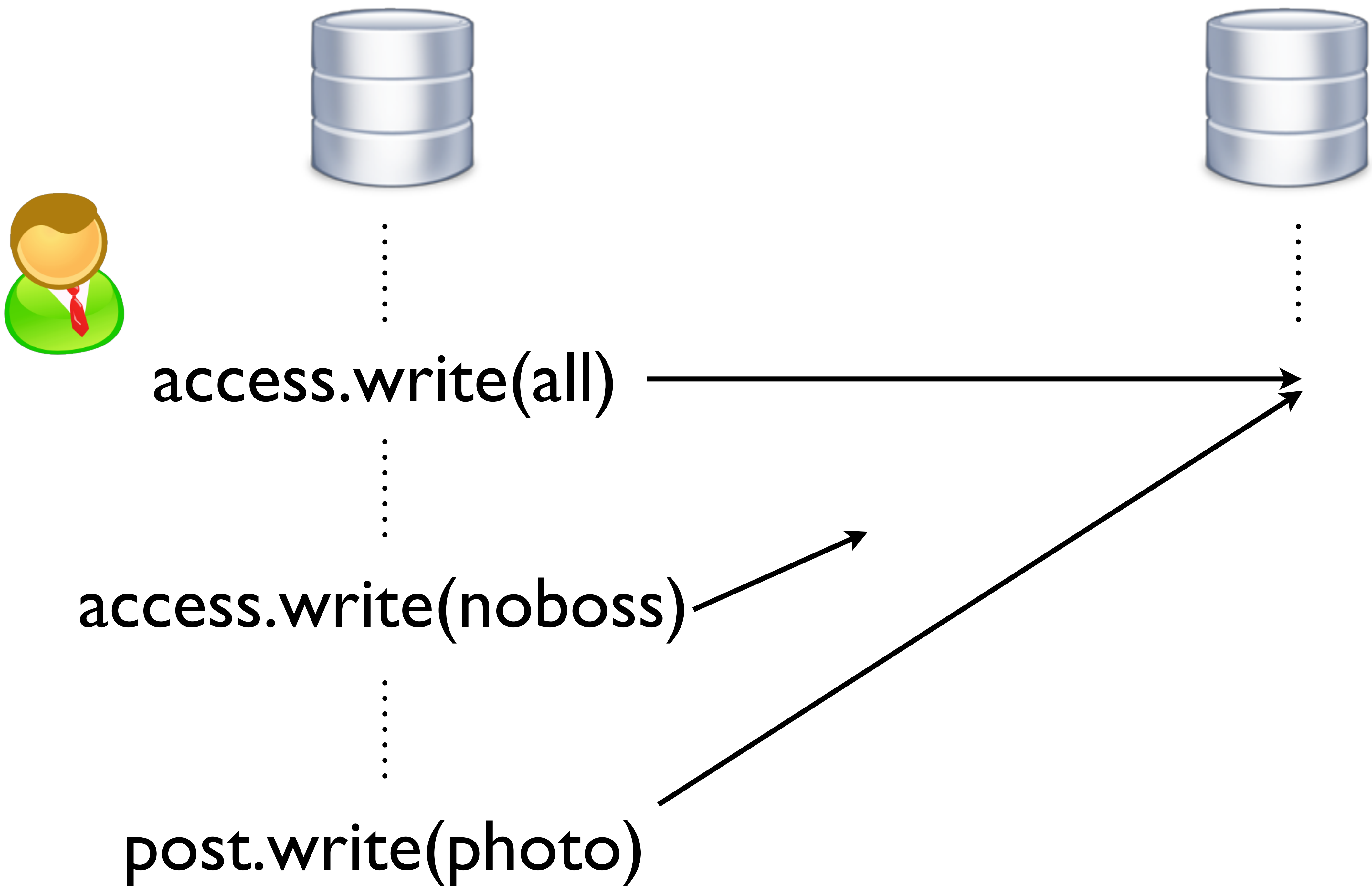


access.write(all)

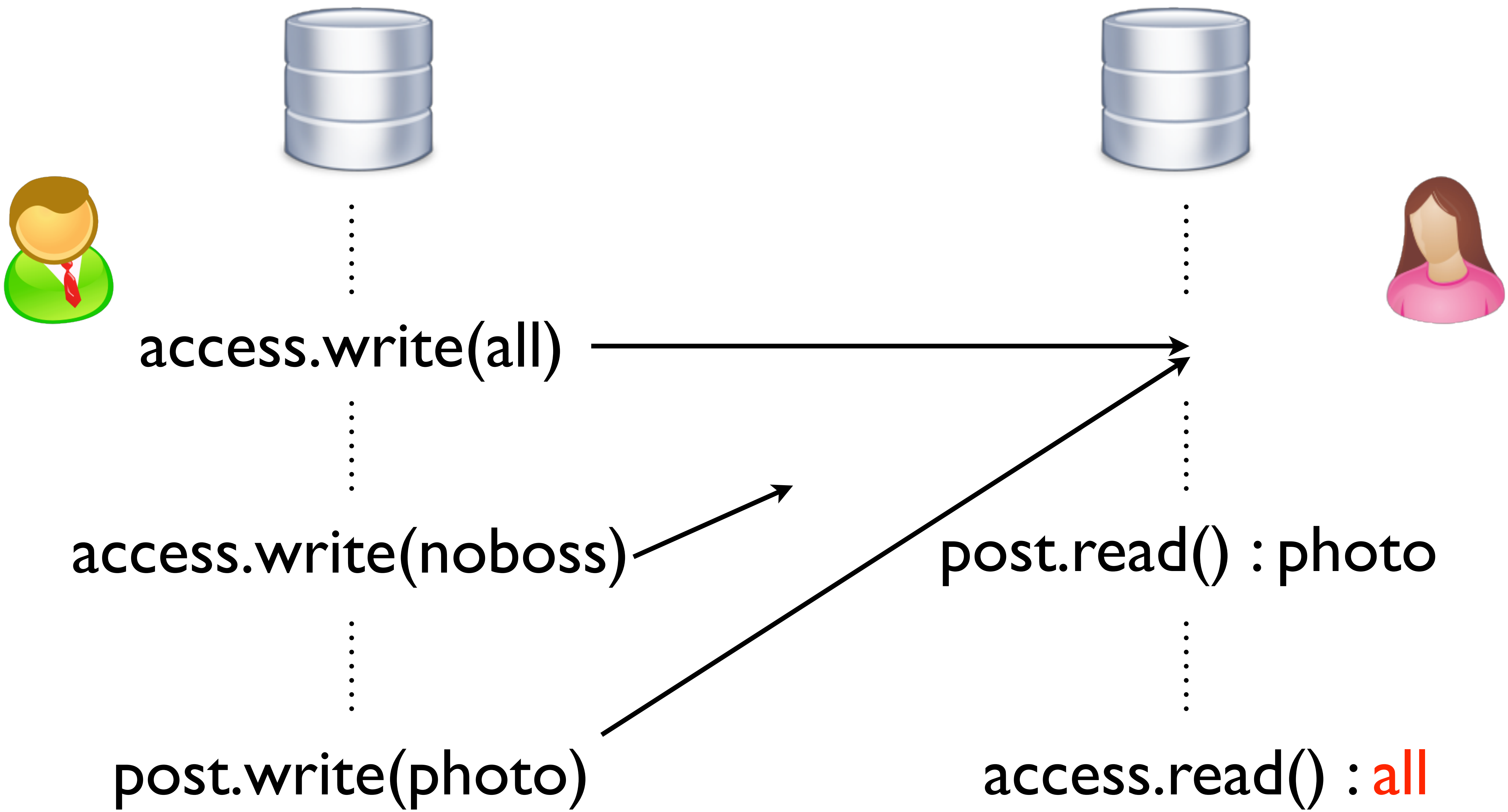
access.write(noboss)

post.write(photo)

Anomalies

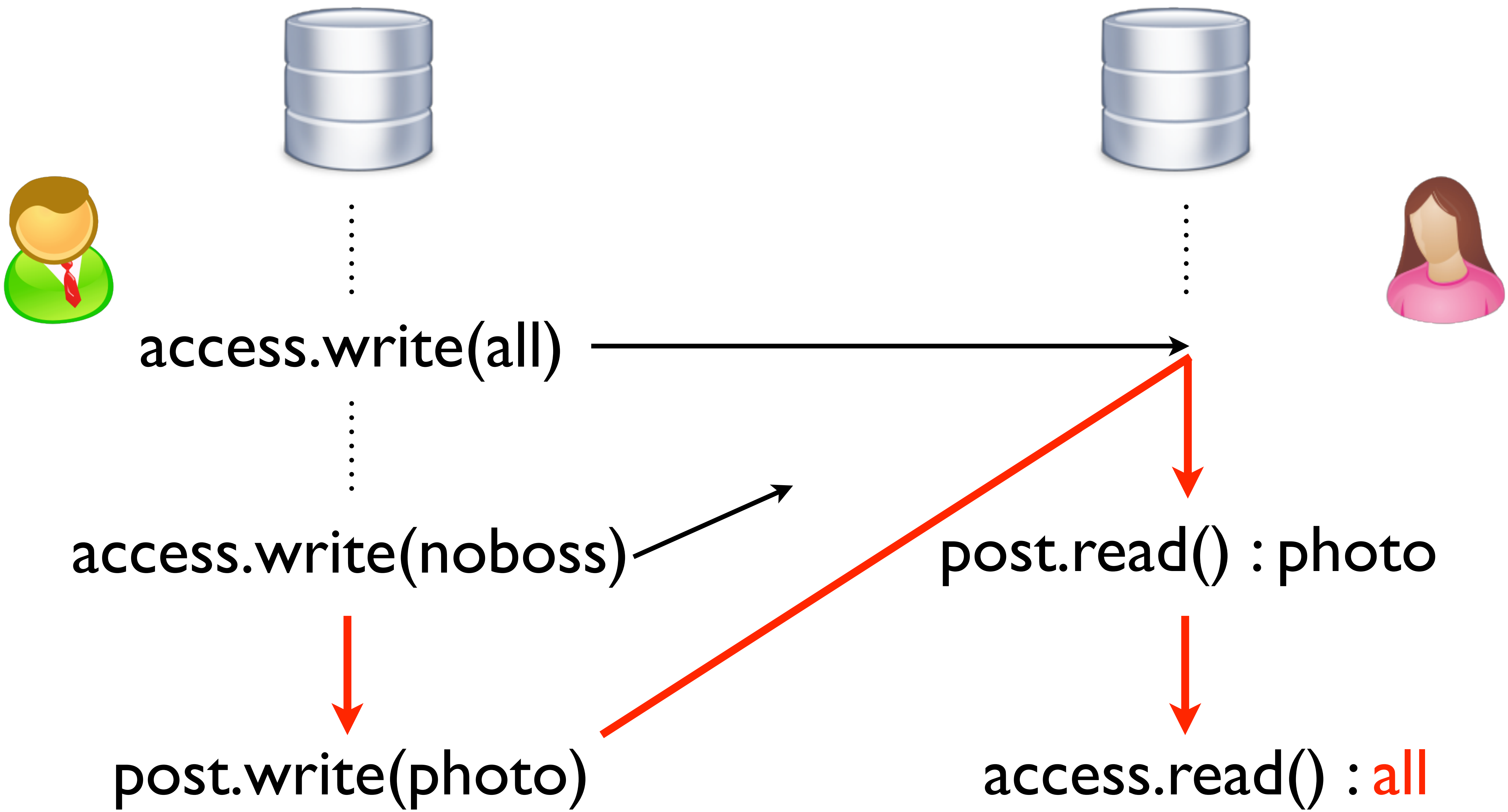


Anomalies

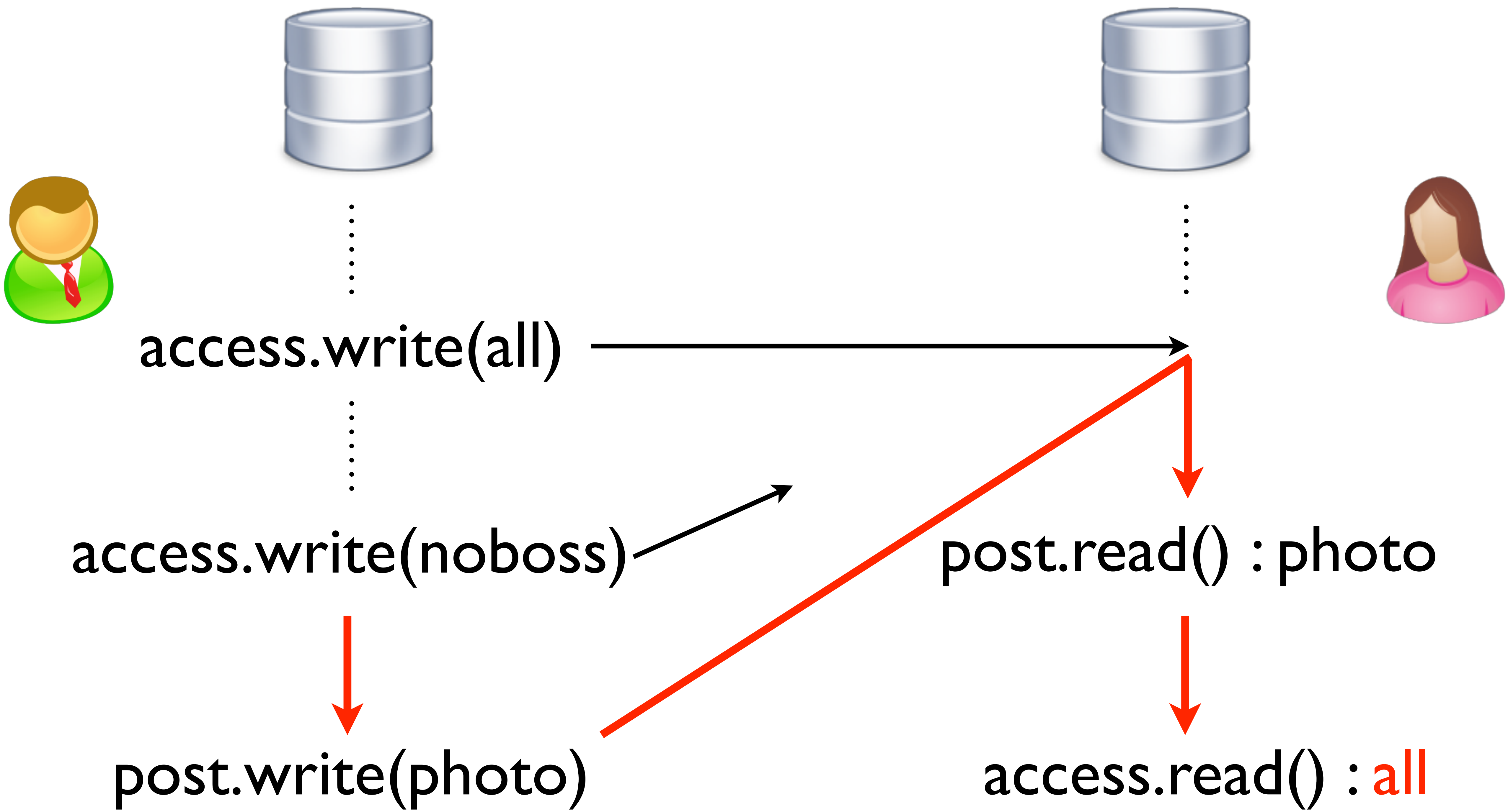


access.write(all)
access.write(noboss)
post.write(photo)
post.read() : photo
access.read() : all

Anomalies

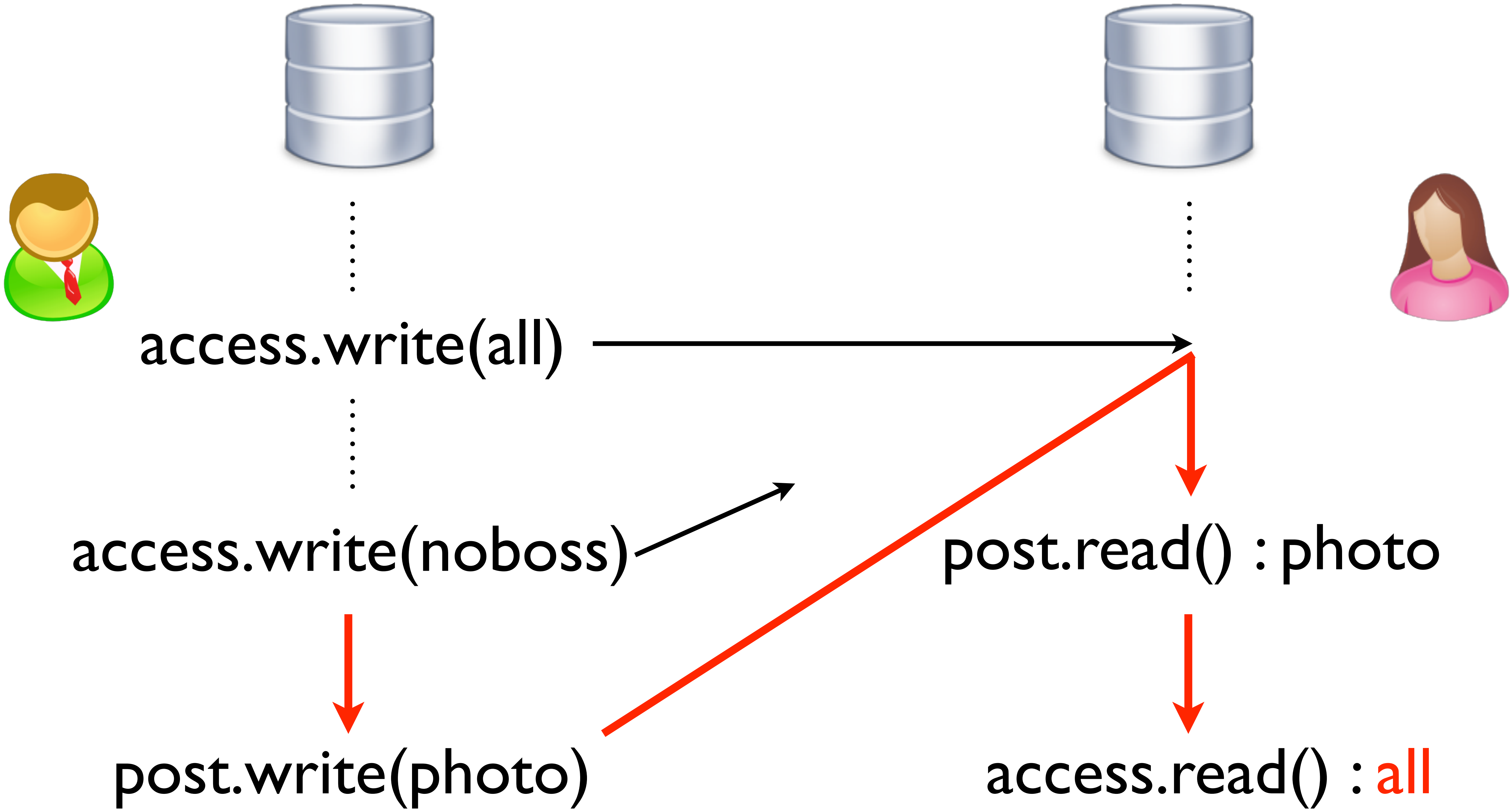

access.write(all)
access.write(noboss)
post.write(photo)
post.read() : photo
access.read() : all

Anomalies



Causality violation: disallowed by causal consistency

omalies



Causality violation: disallowed by causal consistency

Specification

- Lots of replicated data type implementations: e.g., can send snapshots of object states instead of operations
- Lots of message delivery guarantees: different implementations of causal consistency
- Want specifications that abstract from implementation details: both replicated data types and anomalies

Axiomatic specifications

- Choose a set of relations over events: $r_1, \ldots, r_n$

  Abstractly specify essential information about how operations are processed inside the system

- Abstract execution $(H, r_1, \ldots, r_n)$

- Choose a set of axioms $\mathscr{A}$ constraining abstract executions

- Consistency model = $\{H \mid \exists\, r_1, \ldots, r_n.\ (H, r_1, \ldots, r_n) \models \mathscr{A}\}$

Sequential consistency

(E,so) | ∃ total order to. (E, so, to) satisfies:

1. so ⊆ to
2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

Sequential consistency

Partial orders

(E,so) | ∃ total order to. (E, so, to) satisfies:

1. so ⊆ to
2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

Sequential consistency

Partial orders

(E,so) | ∃ total order to. (E, so, to) satisfies:

1. so $\subseteq$ to

Order inclusion axioms: anomalies

2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

Sequential consistency

Partial orders

(E,so) | ∃ total order to. (E, so, to) satisfies:

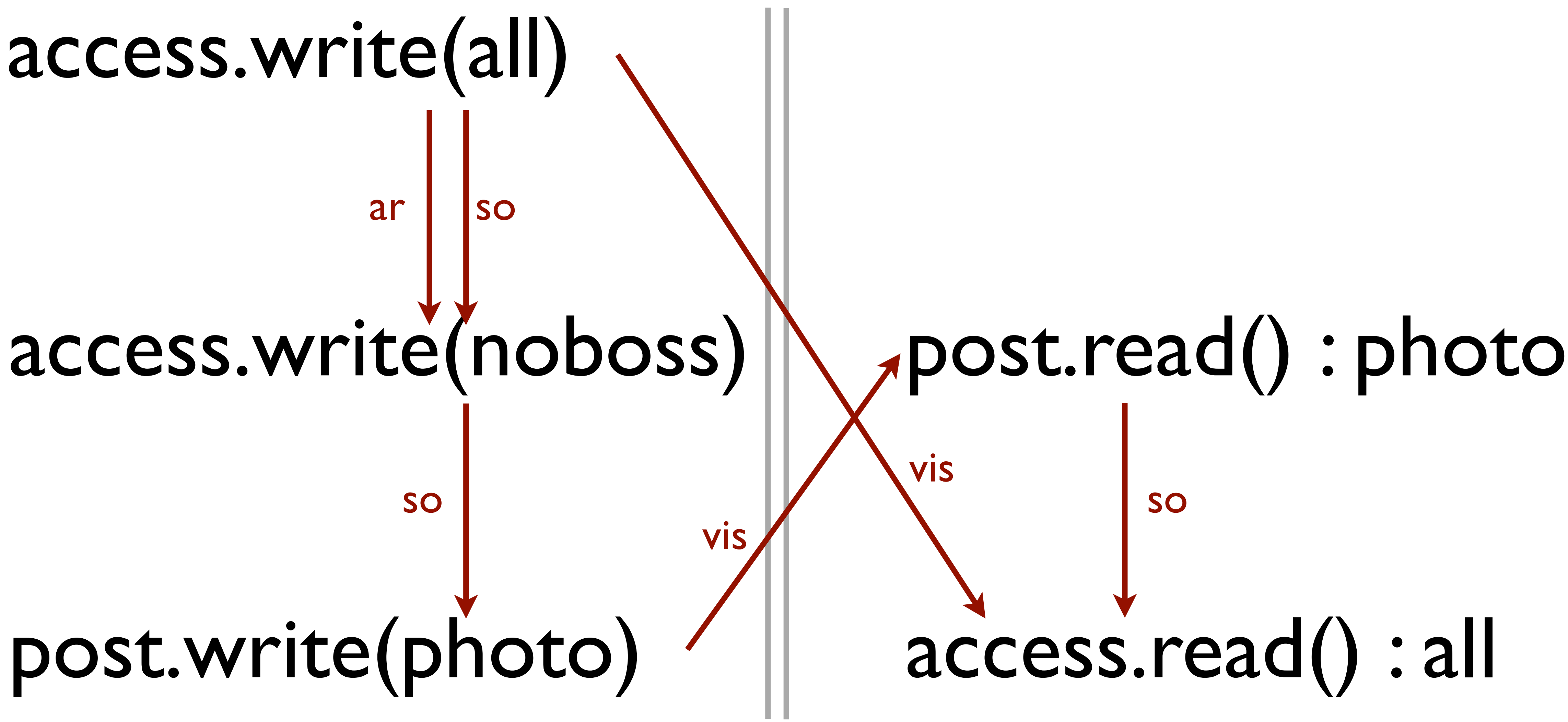
1. so ⊆ to

Order inclusion axioms: anomalies

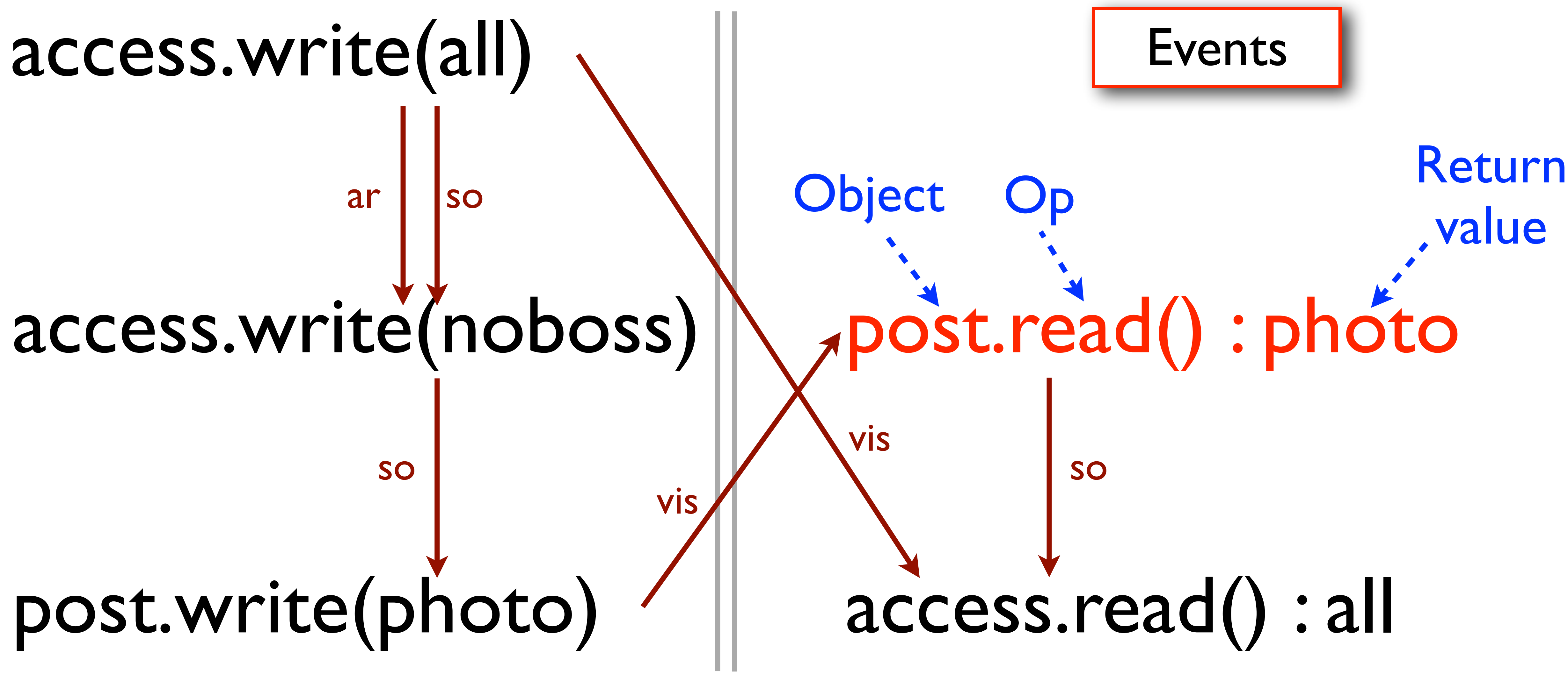
2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

Return value axiom: replicated data types

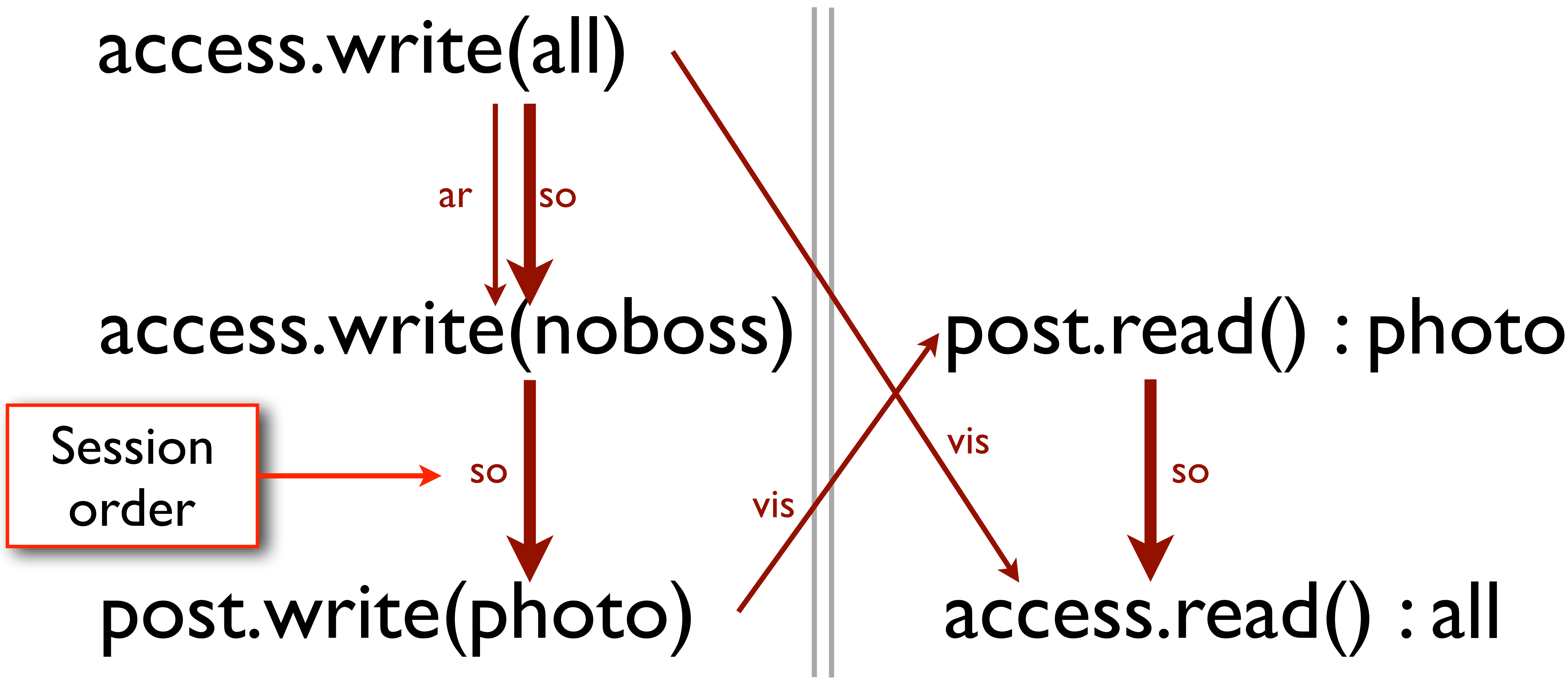
Execution: (E, so, vis, ar)



Execution: (E, so, vis, ar)

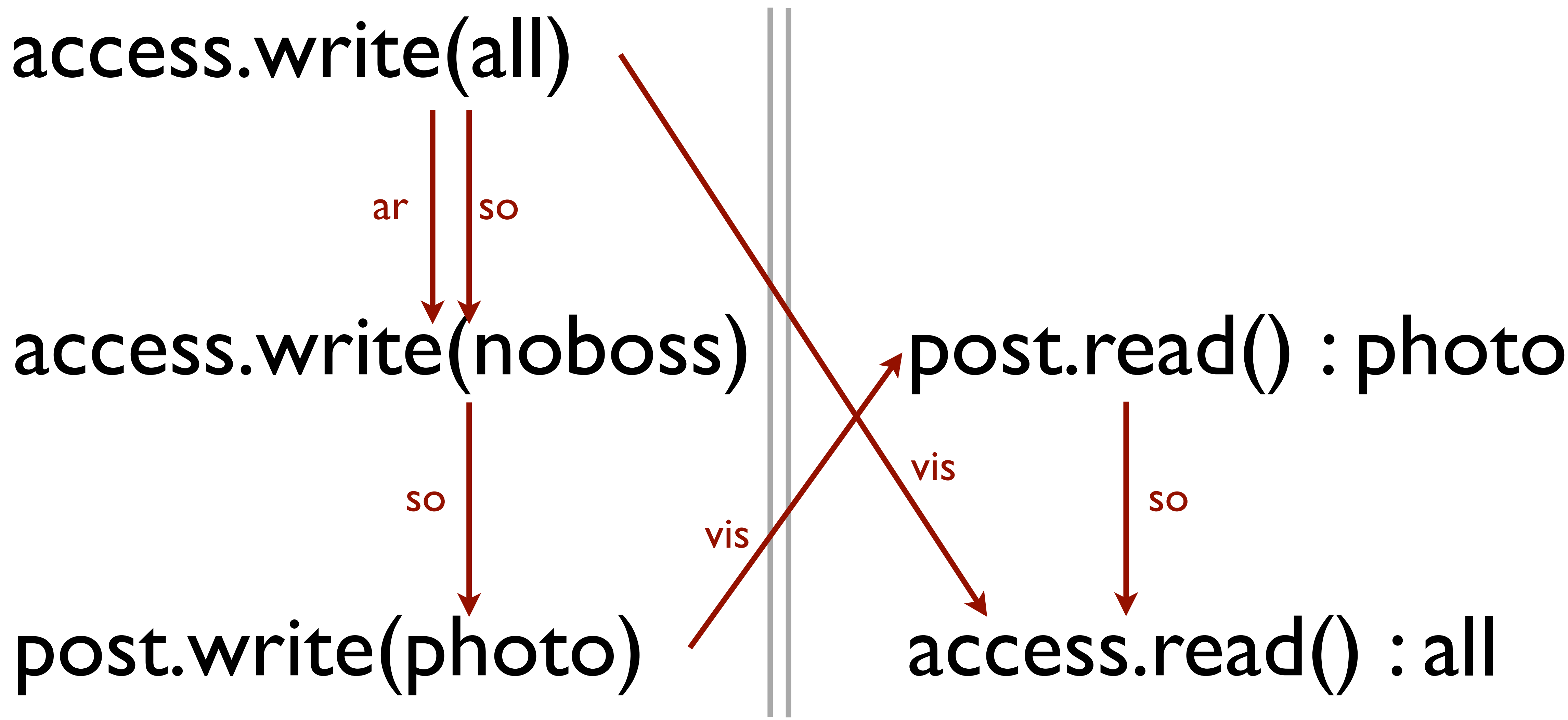


Execution: (E, so, vis, ar)

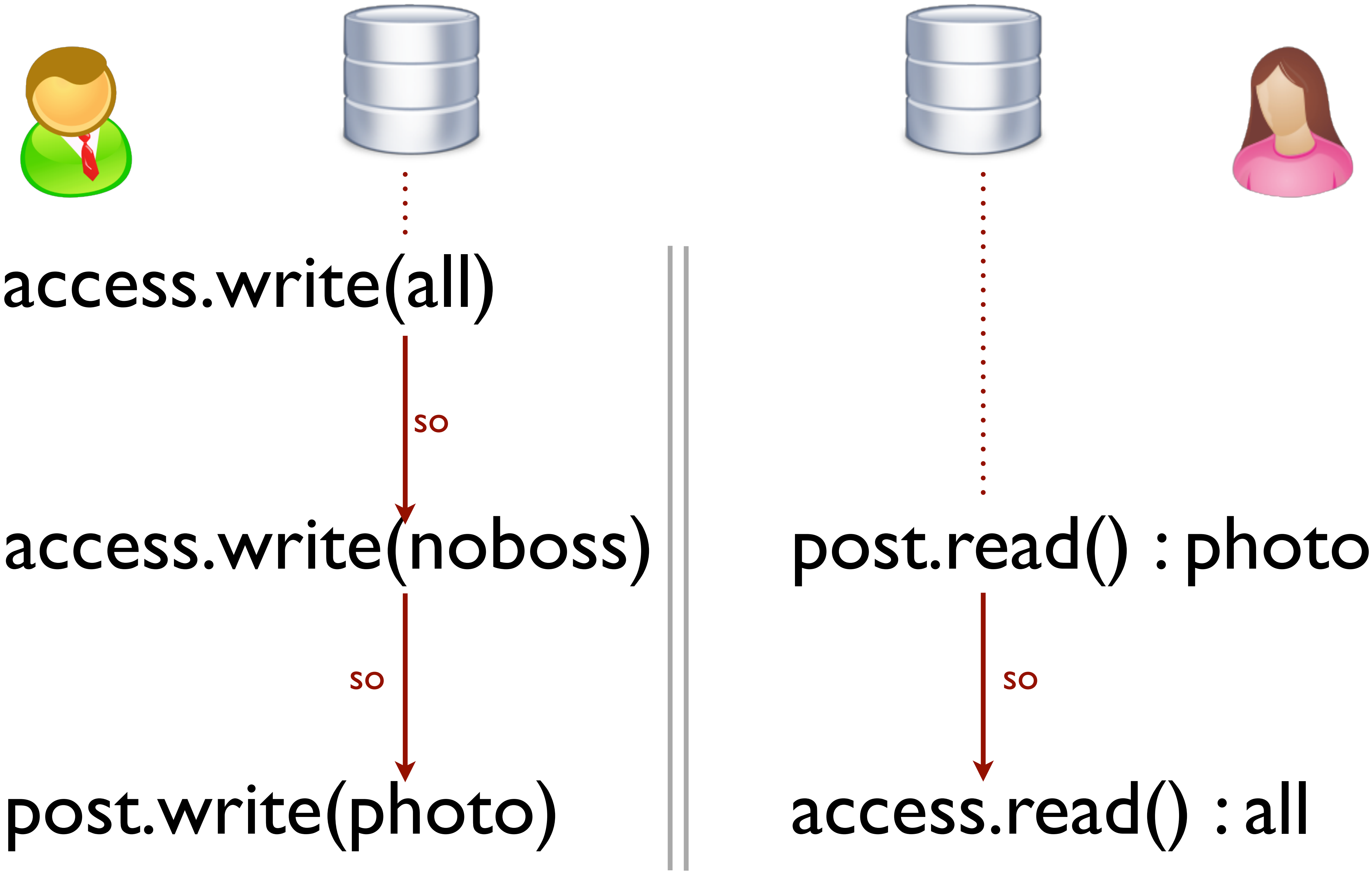


The order of requests by the same session

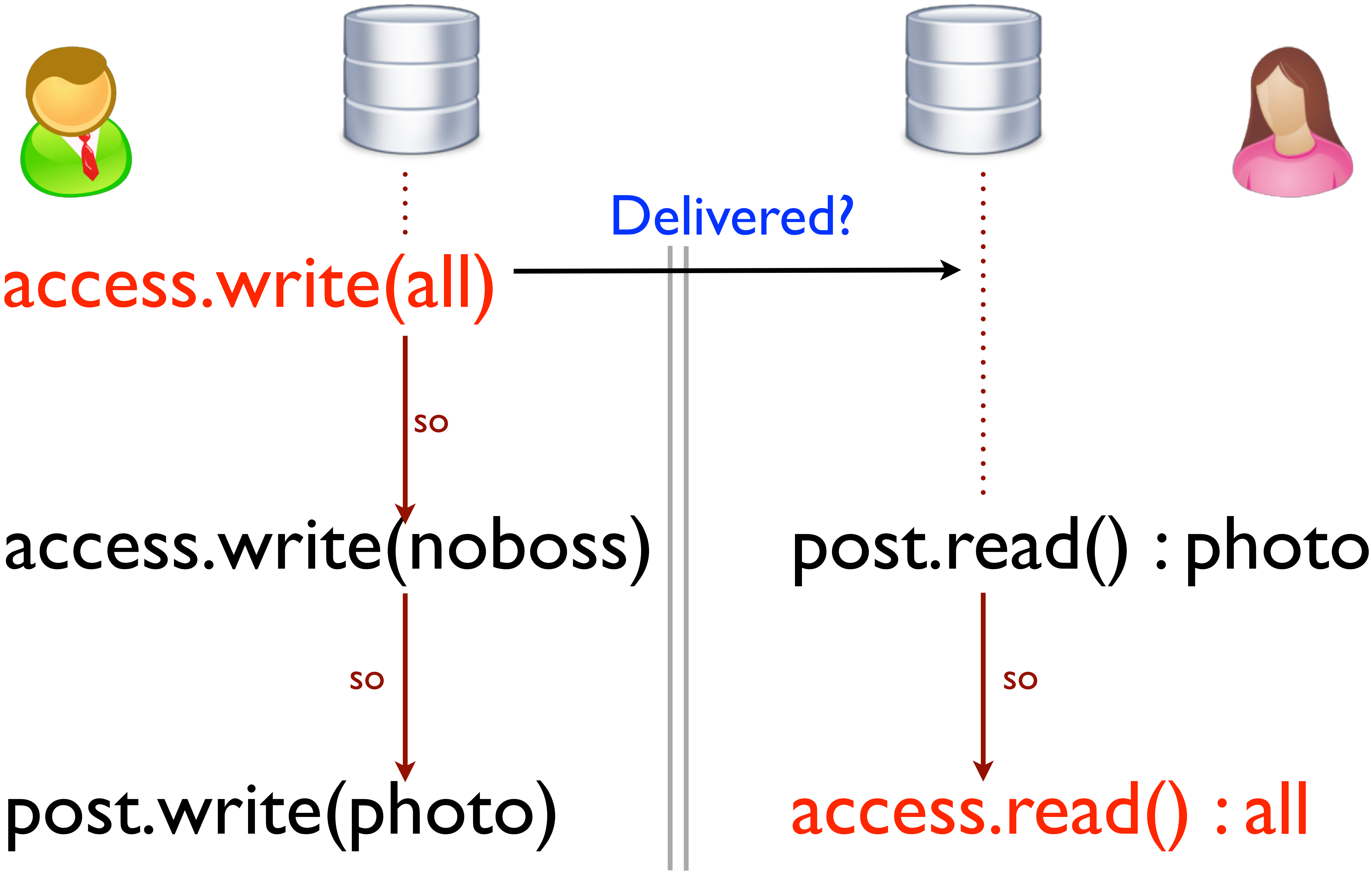
Execution: (E, so, vis, ar)

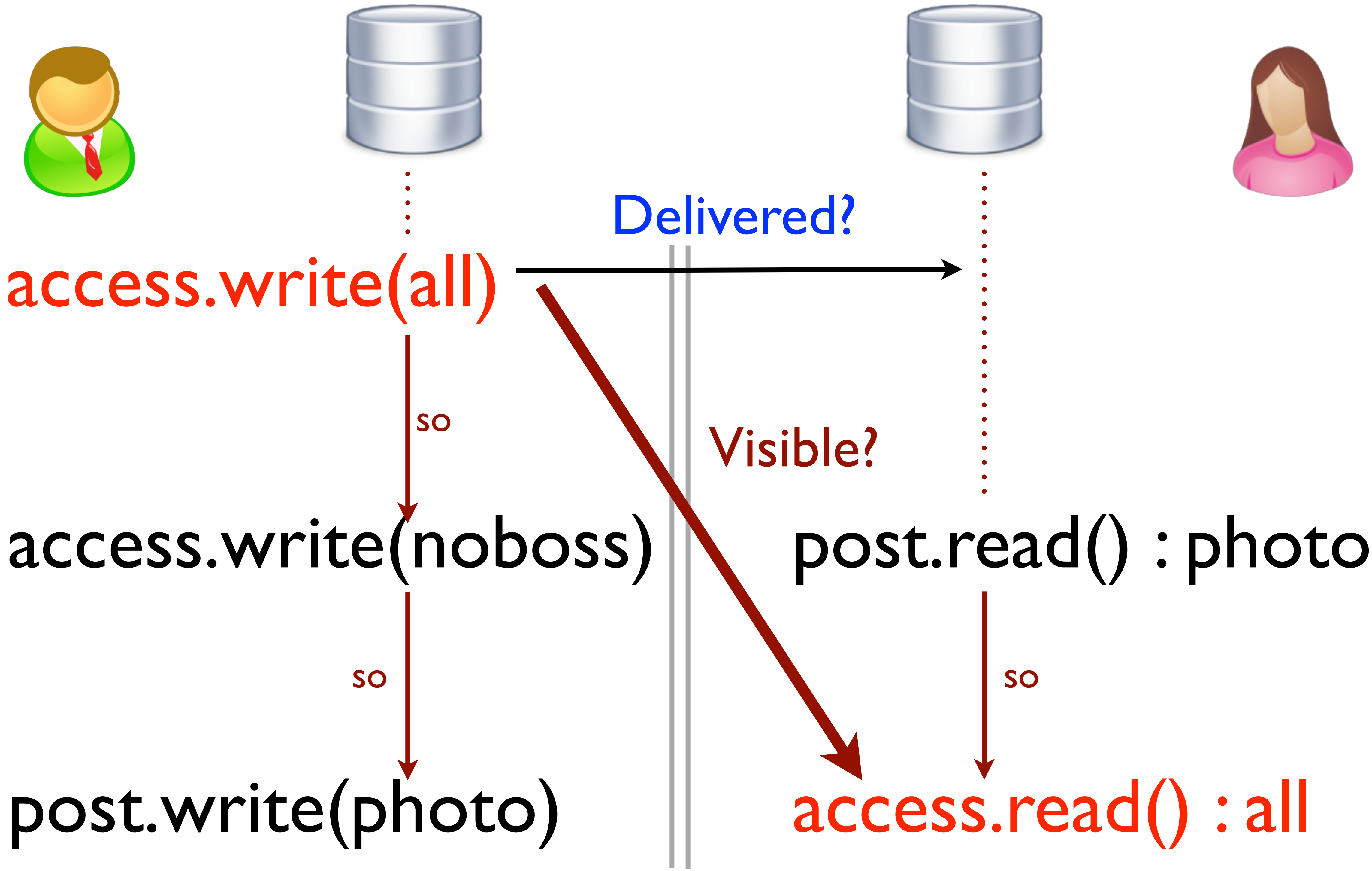


Declaratively specify ways in which the database processes requests

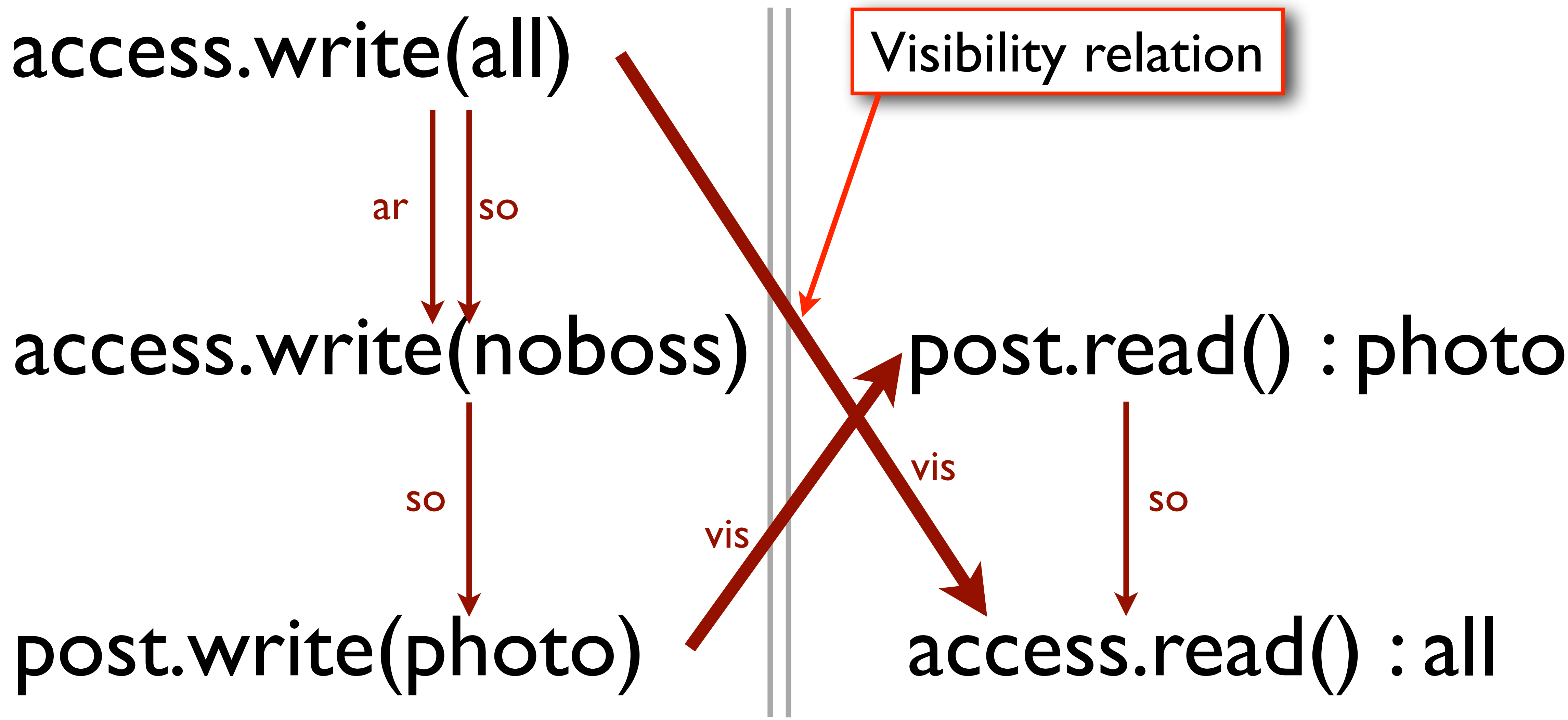


access.write(all)
so
access.write(noboss)
so
post.write(photo)
post.read() : photo
so
access.read() : all

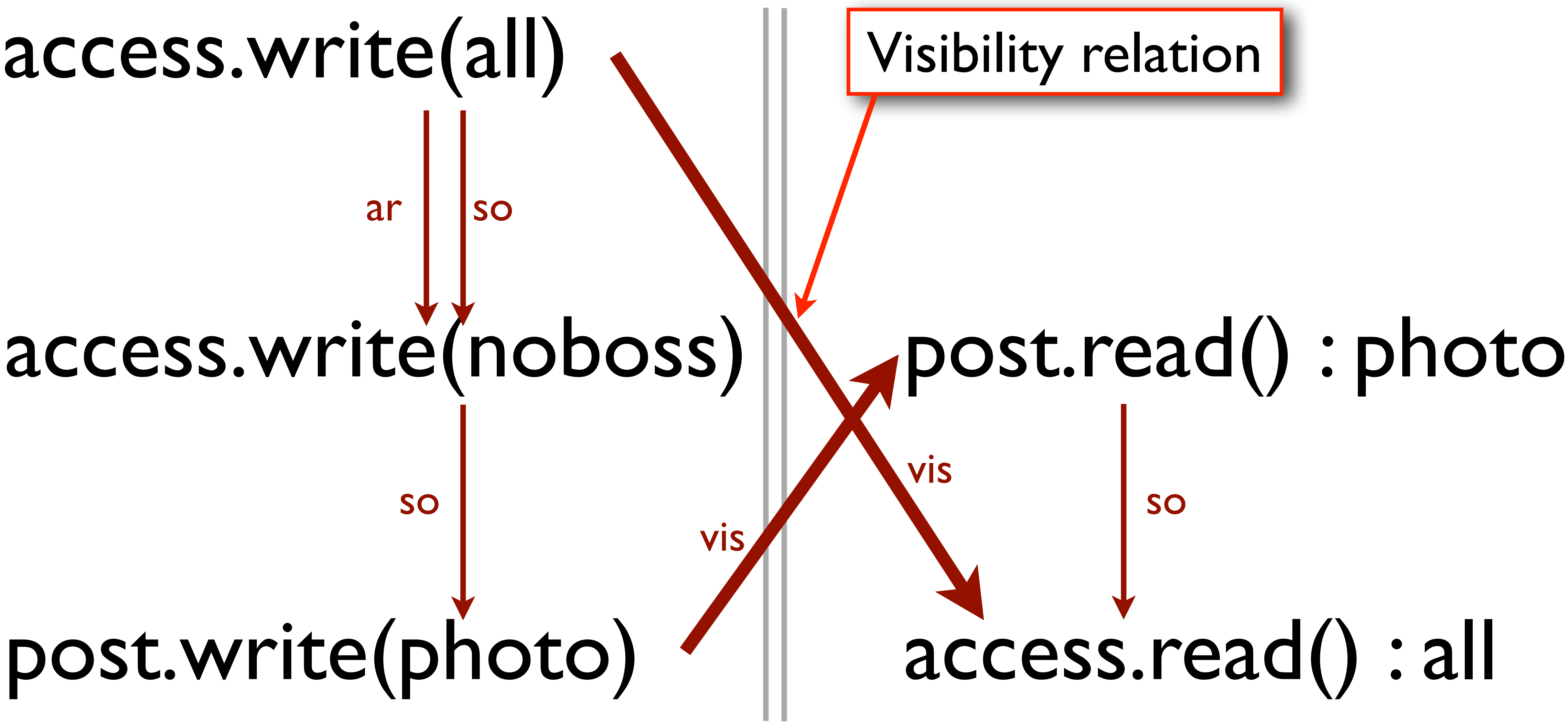

Delivered?
access.write(all)
so
access.write(noboss)
so
post.write(photo)
post.read() : photo
so
access.read() : all


Delivered?
access.write(all)
so
Visible?
access.write(noboss)
post.read() : photo
so
so
post.write(photo)
access.read() : all

Execution: (E, so, vis, ar)

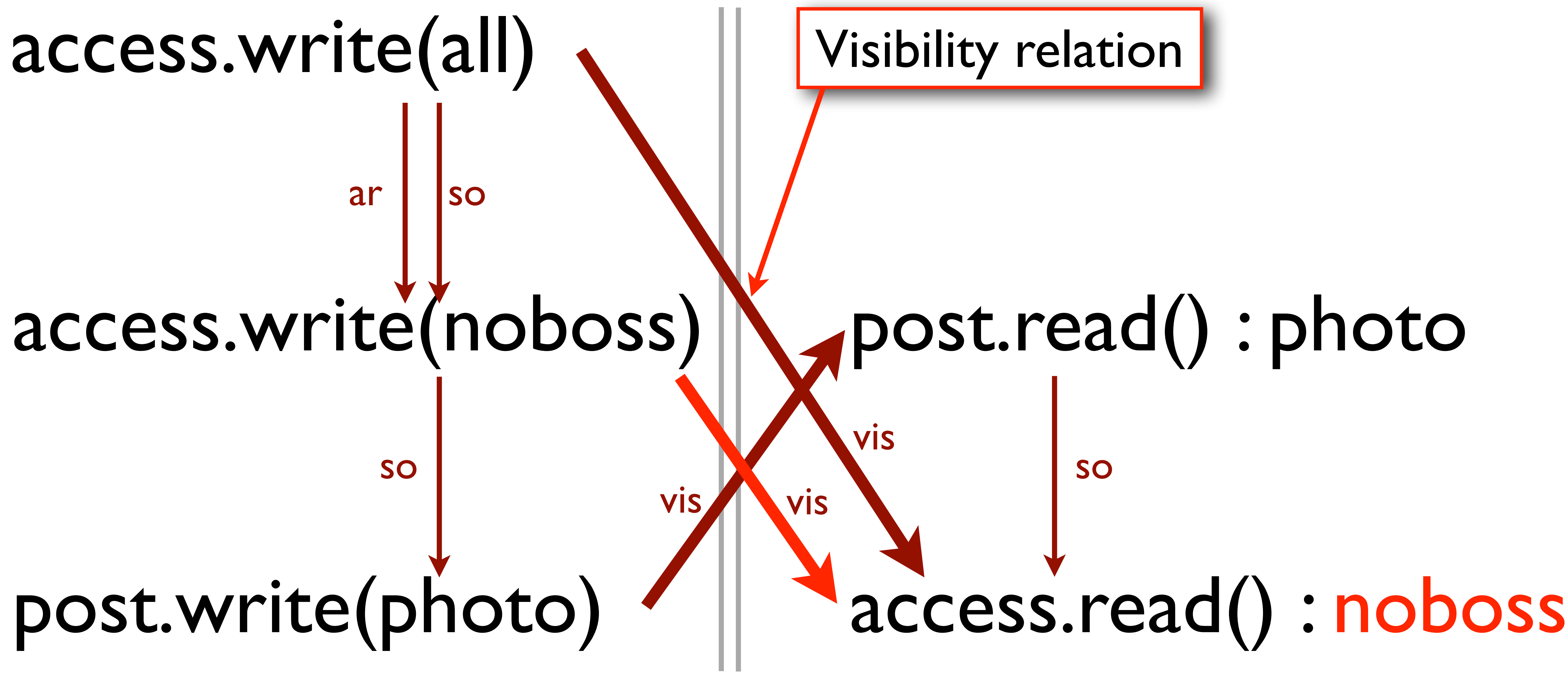


Execution: (E, so, vis, ar)

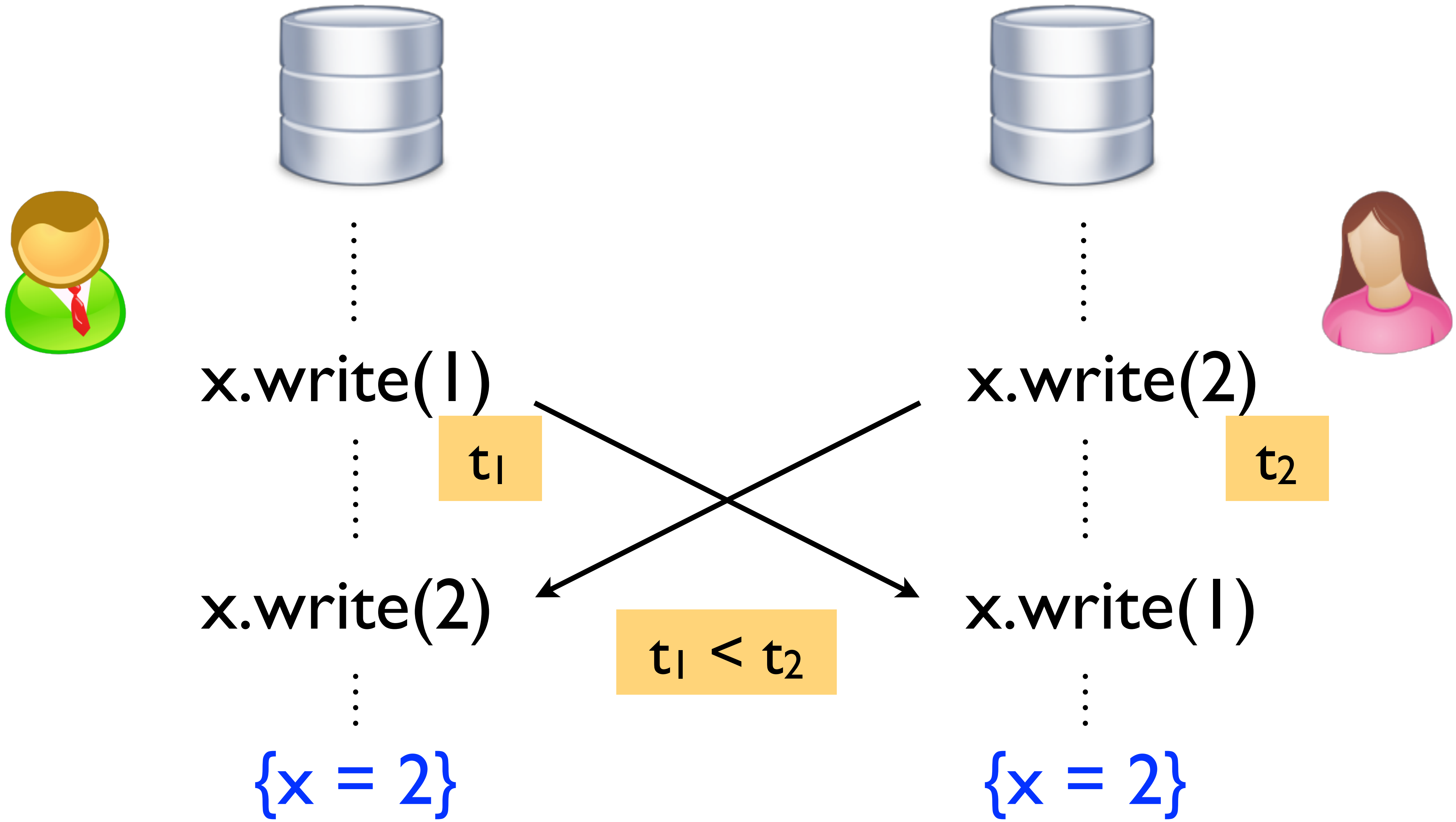


vis is irreflexive and acyclic

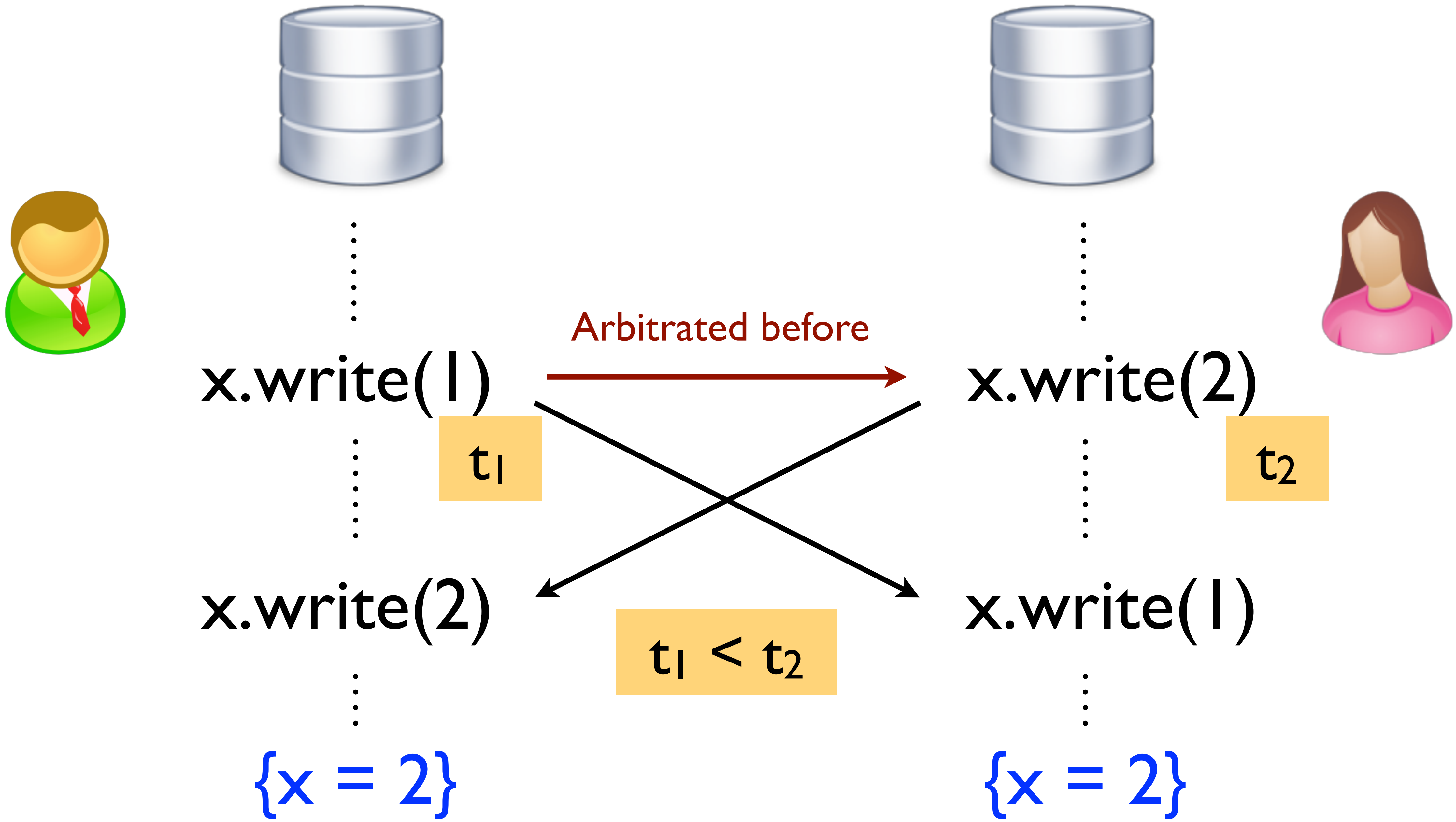
Execution: (E, so, vis, ar)



vis is irreflexive and acyclic

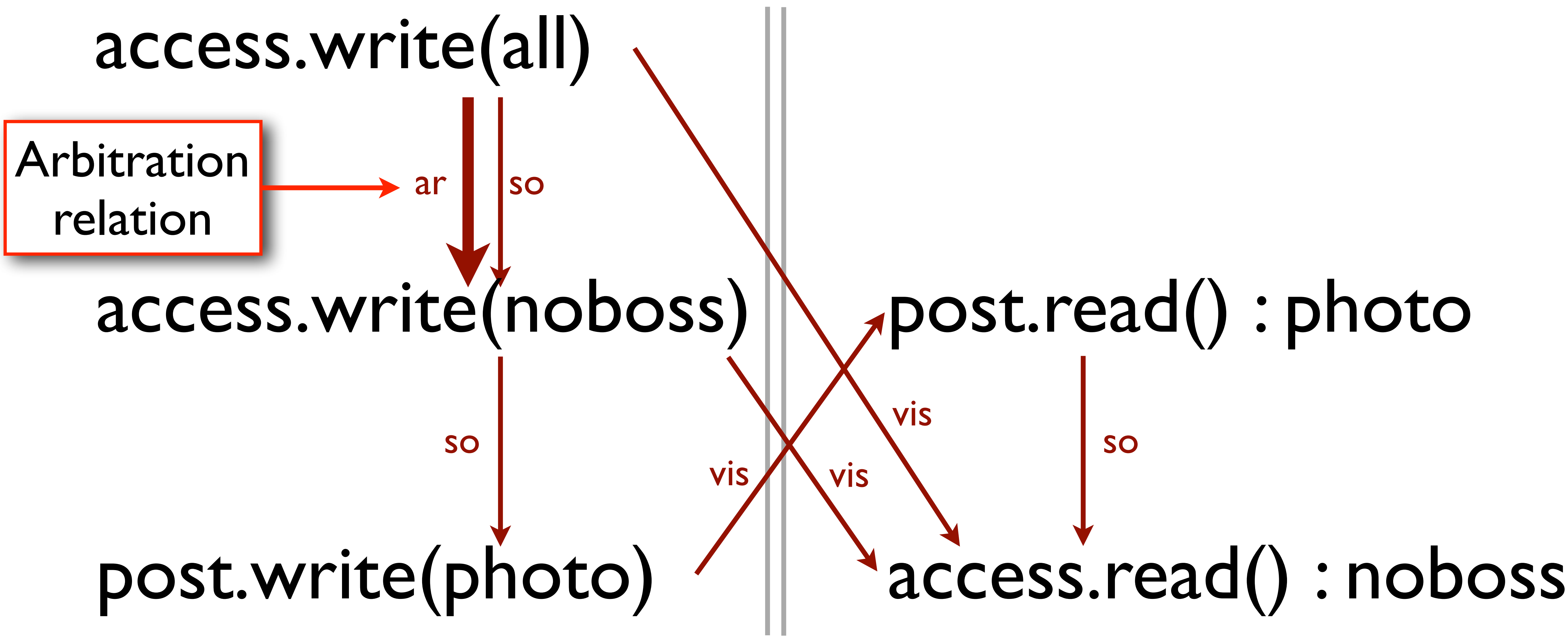


System includes a time-stamping mechanism that can be used in conflict resolution



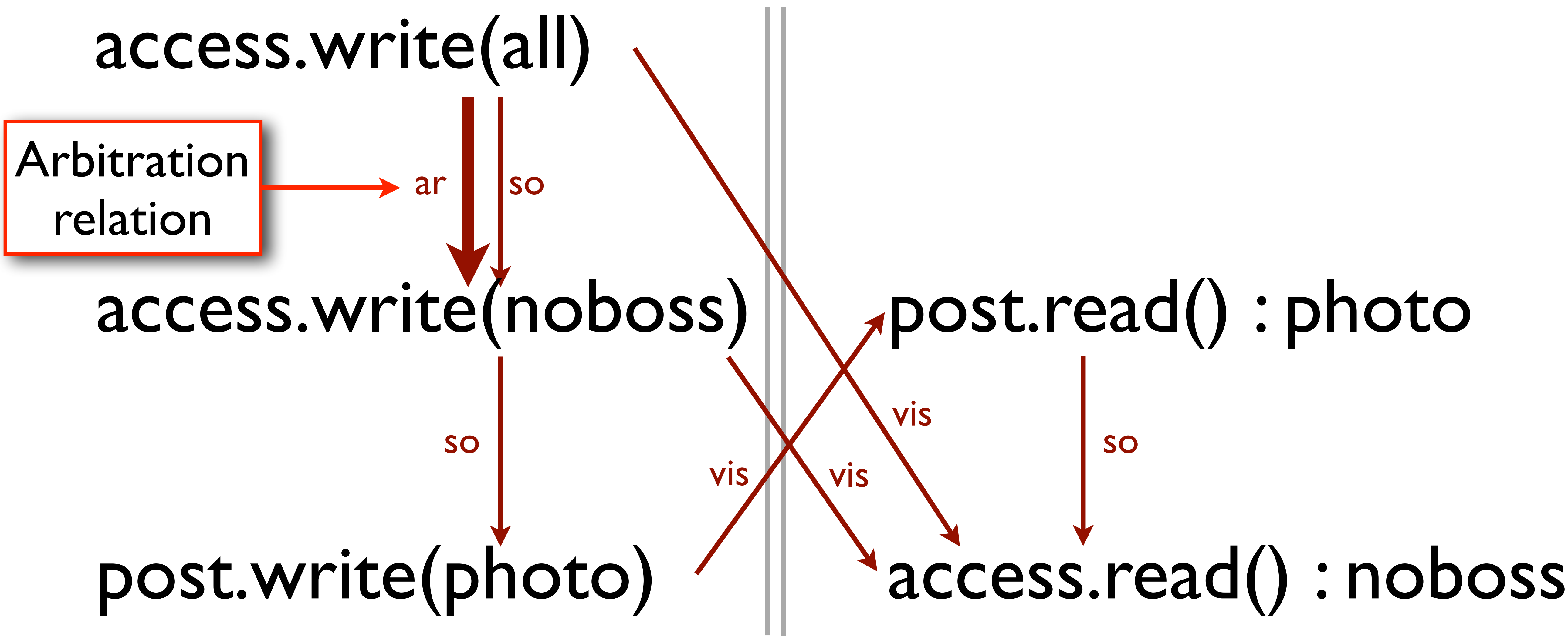
System includes a time-stamping mechanism
that can be used in conflict resolution

Execution: (E, so, vis, ar)



System includes a time-stamping mechanism
that can be used in conflict resolution

Execution: (E, so, vis, ar)

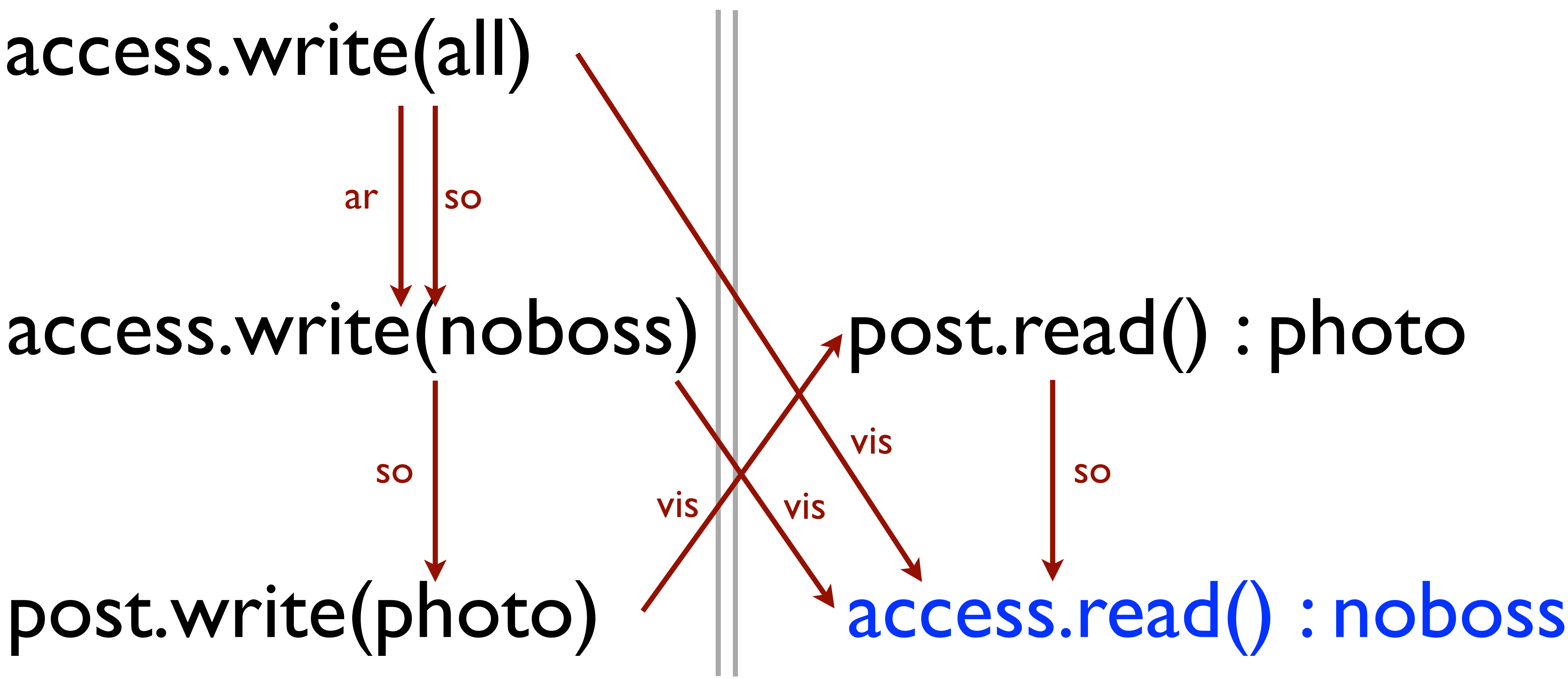


System includes a time-stamping mechanism that can be used in conflict resolution

ar is total on E and vis ⊆ ar

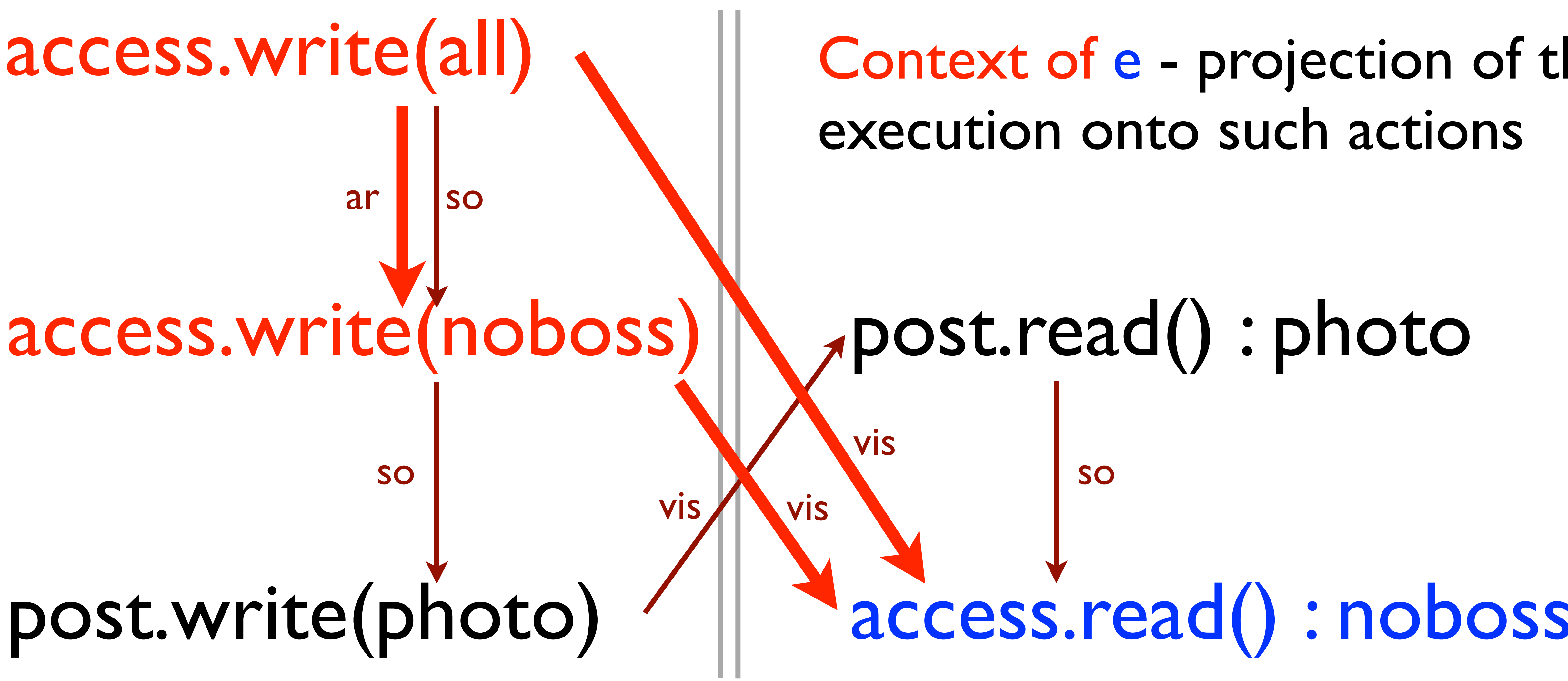
Data type specification

- How do I compute the return value of an event e?
- Only actions on the same object visible to e are important: have been delivered to the replica performing e



Data type specification

- How do I compute the return value of an event e?
- Only actions on the same object visible to e are important: have been delivered to the replica performing e



Context of e - projection of the execution onto such actions

Data type specification

F: context of e → return value of e

$$\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$$

$$\begin{tikzcd}
\text{access.write(all)} \arrow[d, "\text{ar}"'] \arrow[ddr, "\text{vis}"] & \\
\text{access.write(noboss)} \arrow[dr, "\text{vis}"] & \\
& \text{access.read() : noboss}
\end{tikzcd}$$

Data type specification

F: context of e → return value of e

$$\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$$

access.write(all)

ar

access.write(noboss)

vis

vis

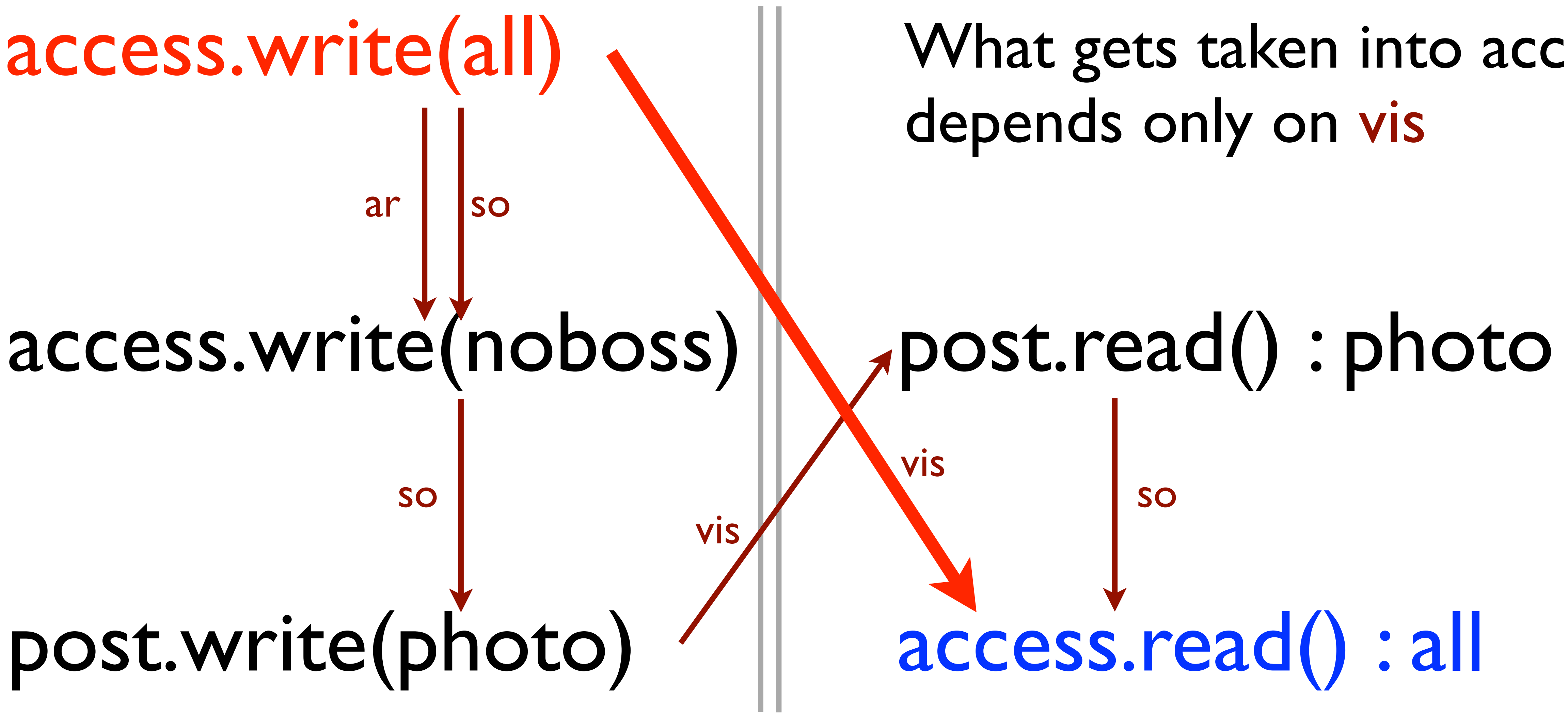
access.read() : noboss

F for Last-Writer-Wins registers: sort all actions according to ar and return the last value written

Data type specification

F: context of e → return value of e

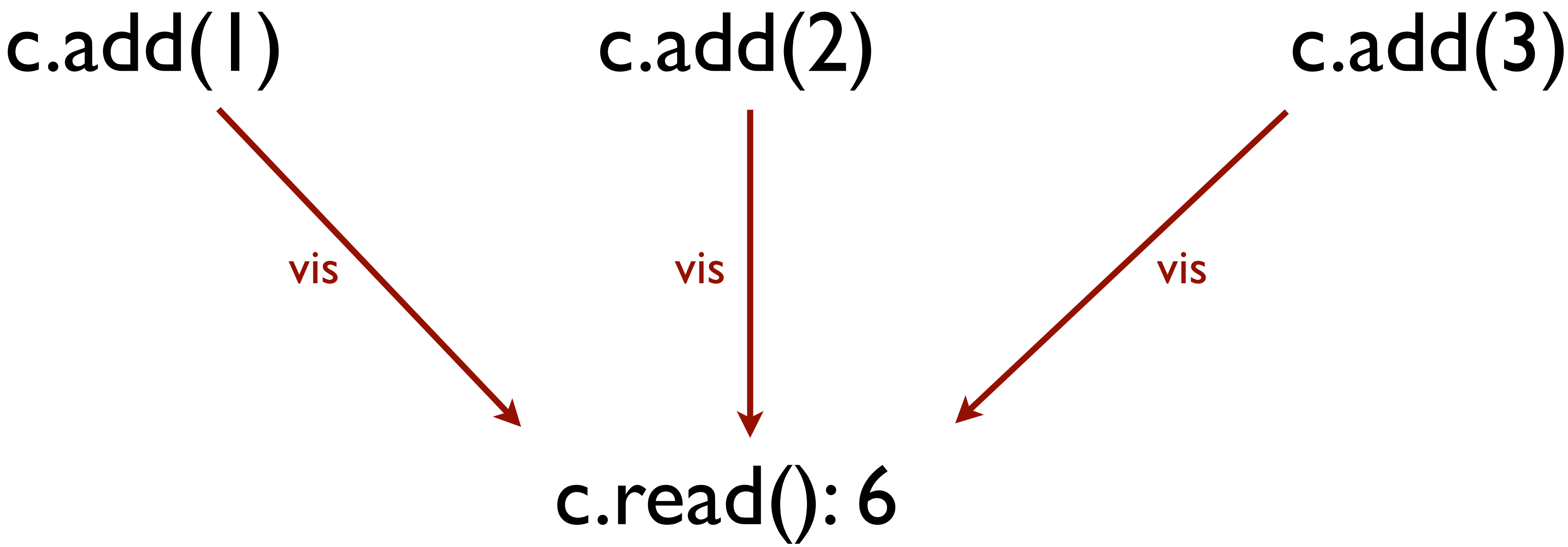
$$\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$$



What gets taken into account depends only on vis

Counter

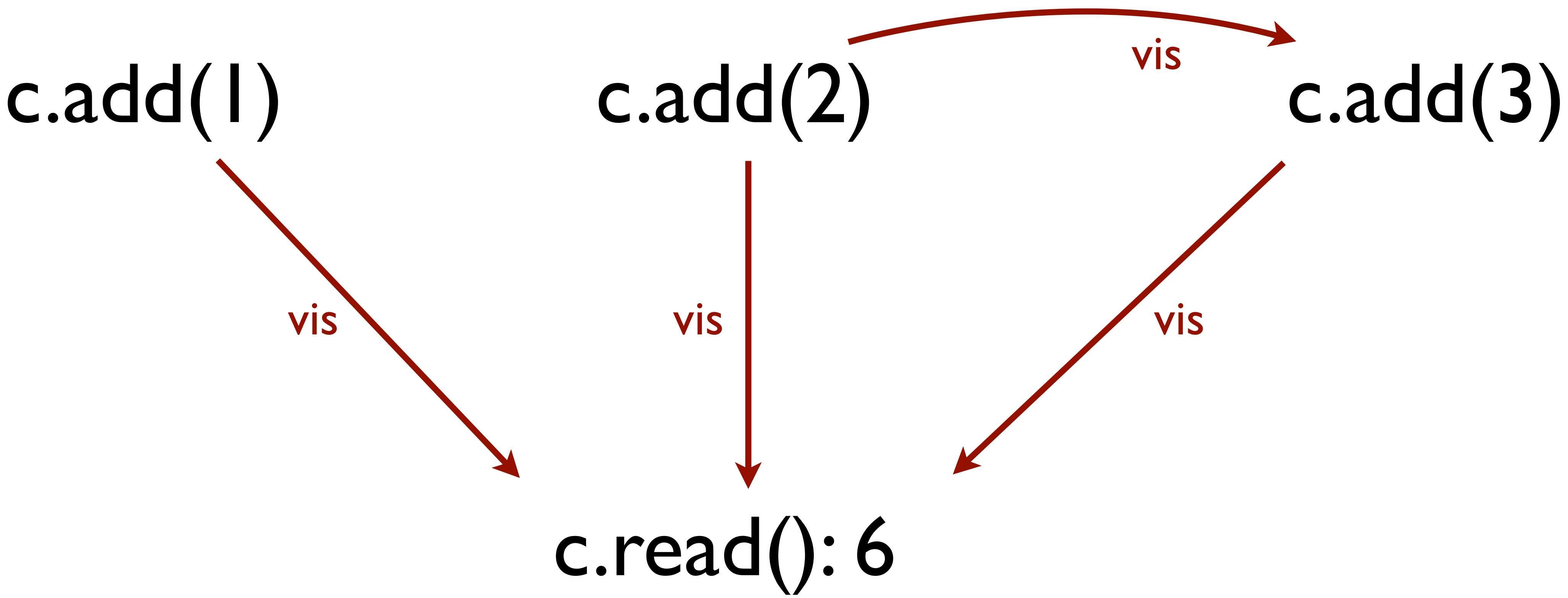
F: context of e → return value of e



F: reads return the sum of all additions in the context

Counter

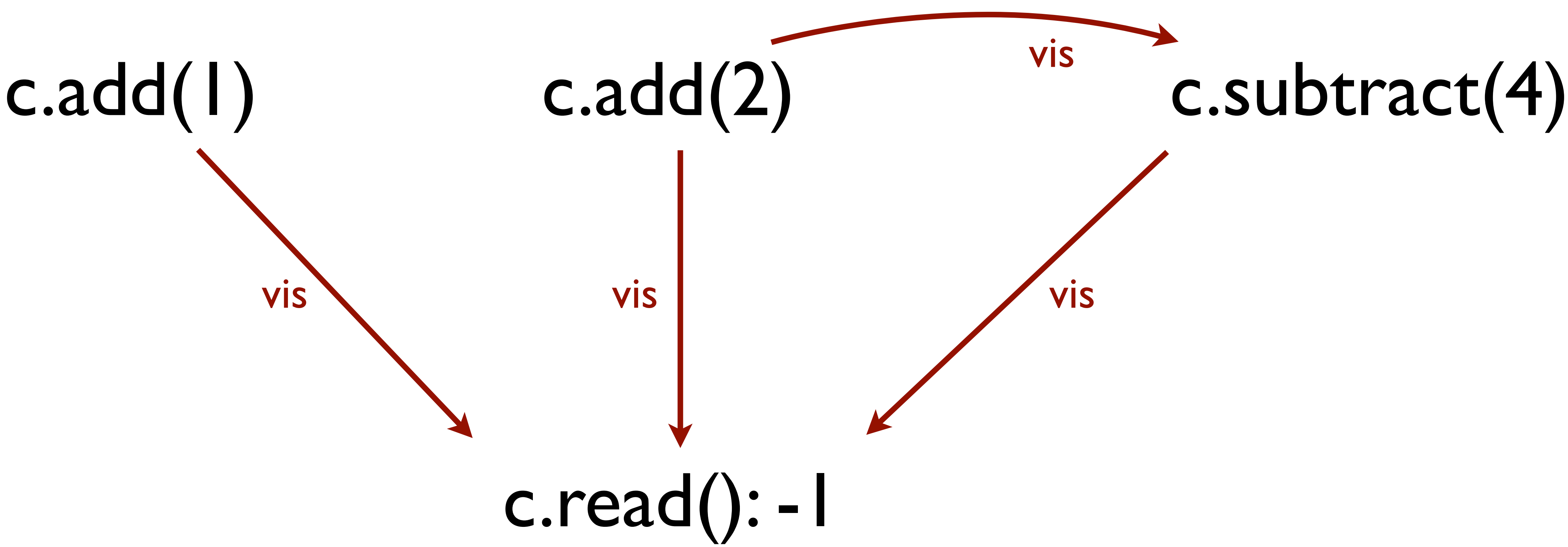
F: context of e → return value of e



Relations between events in the context don't matter

Counter with decrements

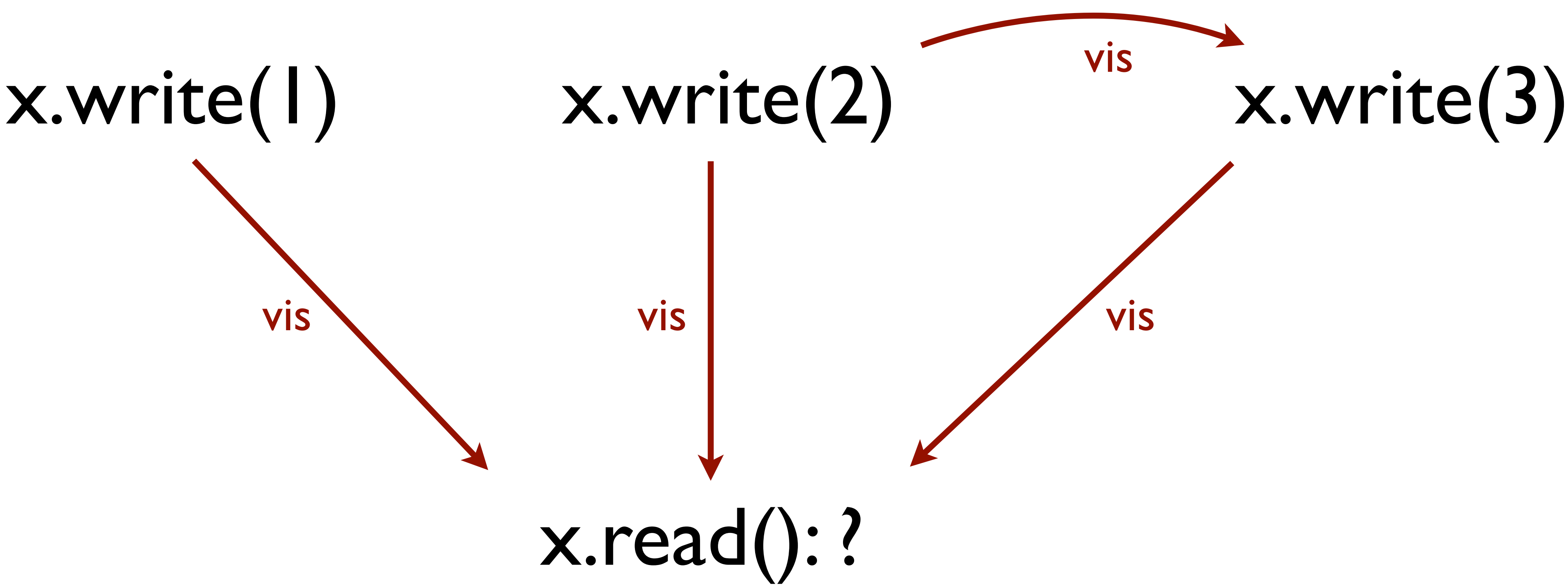
F: context of e → return value of e



F: reads return additions minus subtractions

Multi-valued register

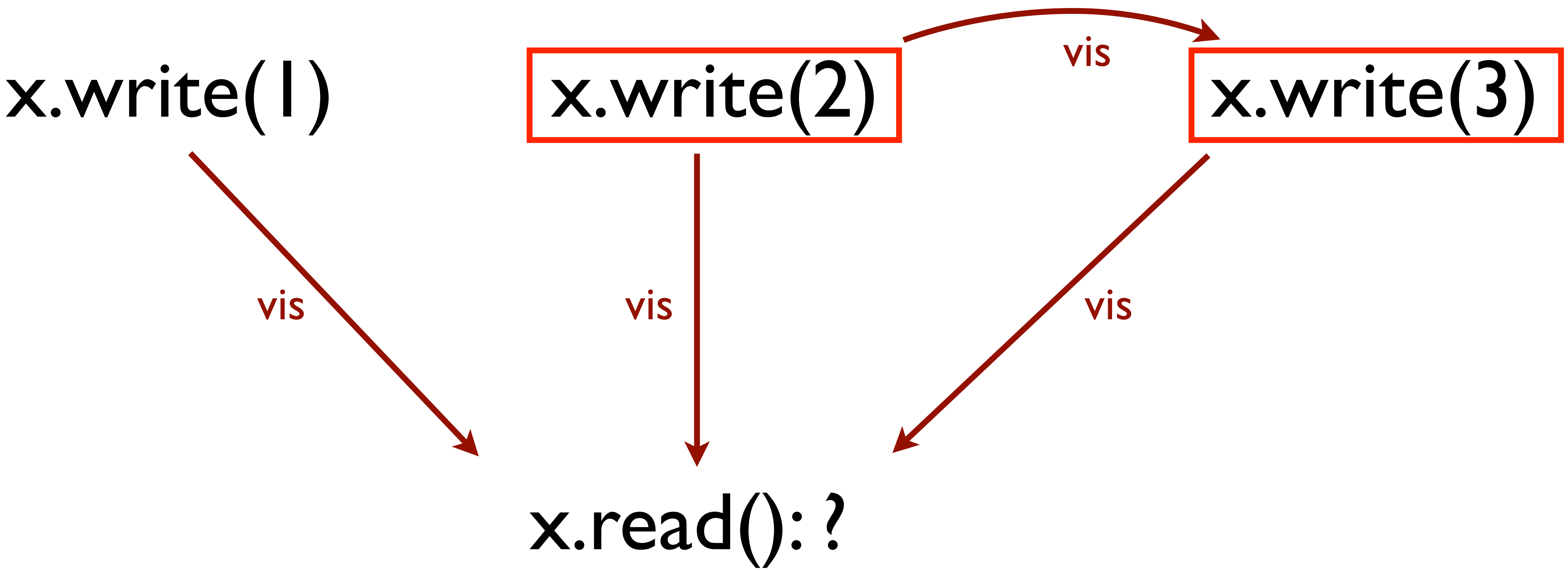
F: context of e → return value of e



F: reads return the set of all conflicting writes

Multi-valued register

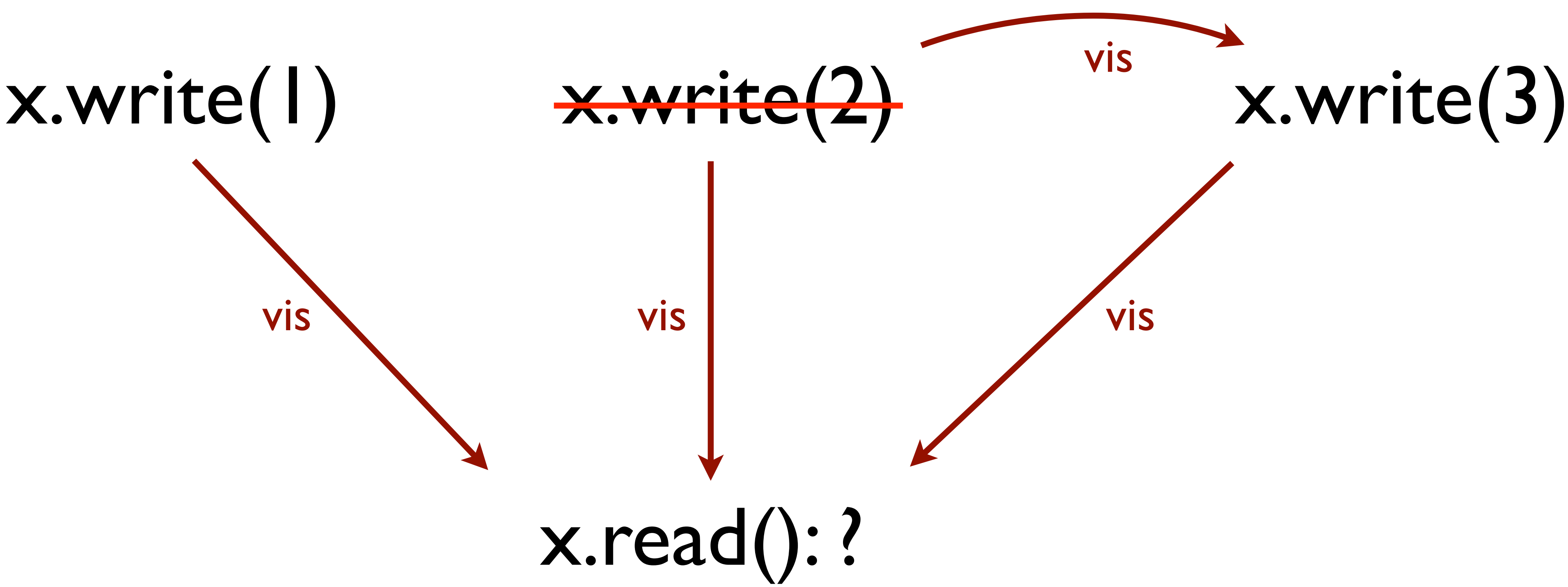
F: context of e → return value of e



F: reads return the set of all conflicting writes

Multi-valued register

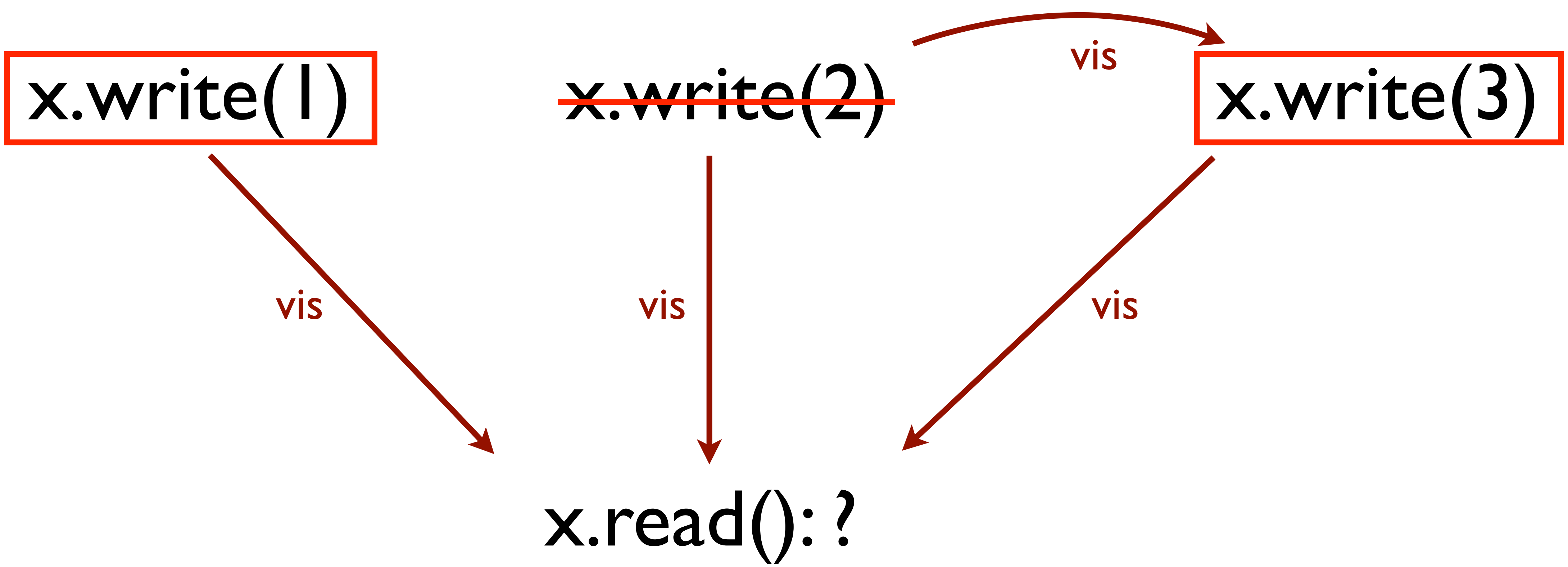
F: context of e → return value of e



F: reads return the set of all conflicting writes

Multi-valued register

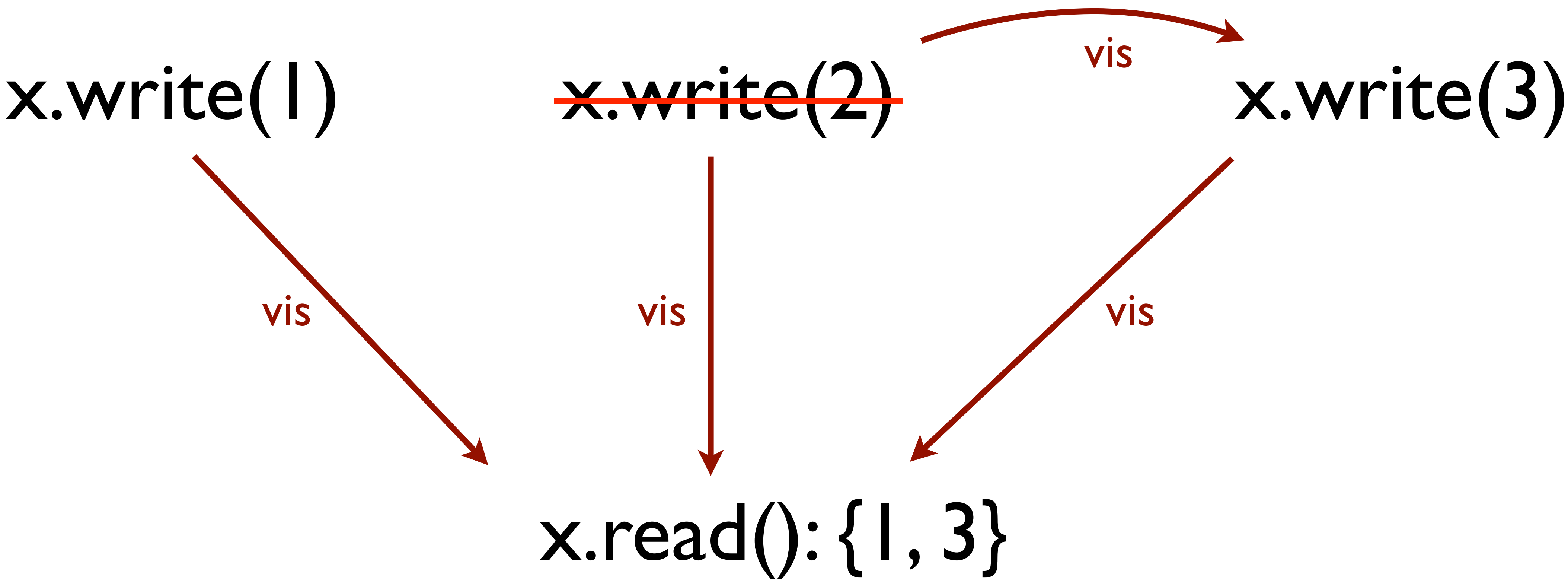
F: context of e → return value of e



F: reads return the set of all conflicting writes

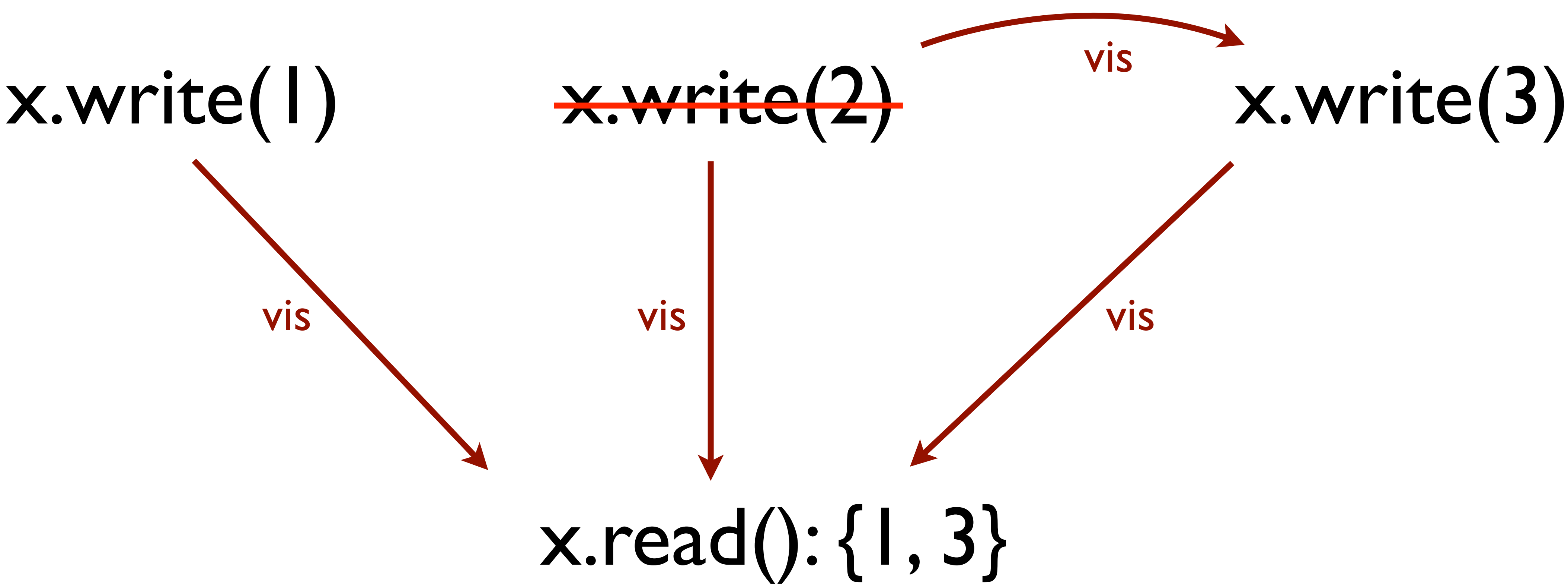
Multi-valued register

F: context of e → return value of e



Multi-valued register

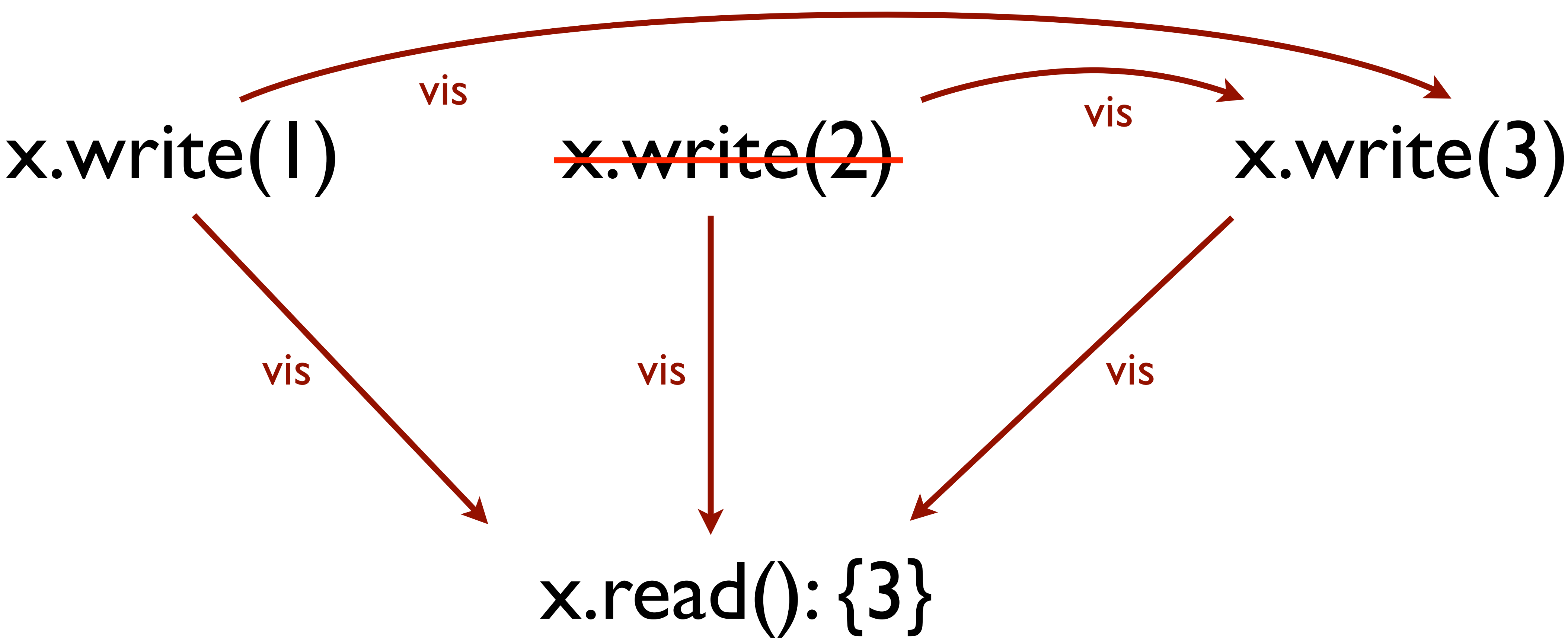
F: context of e → return value of e



F: discard all writes seen by a write

Multi-valued register

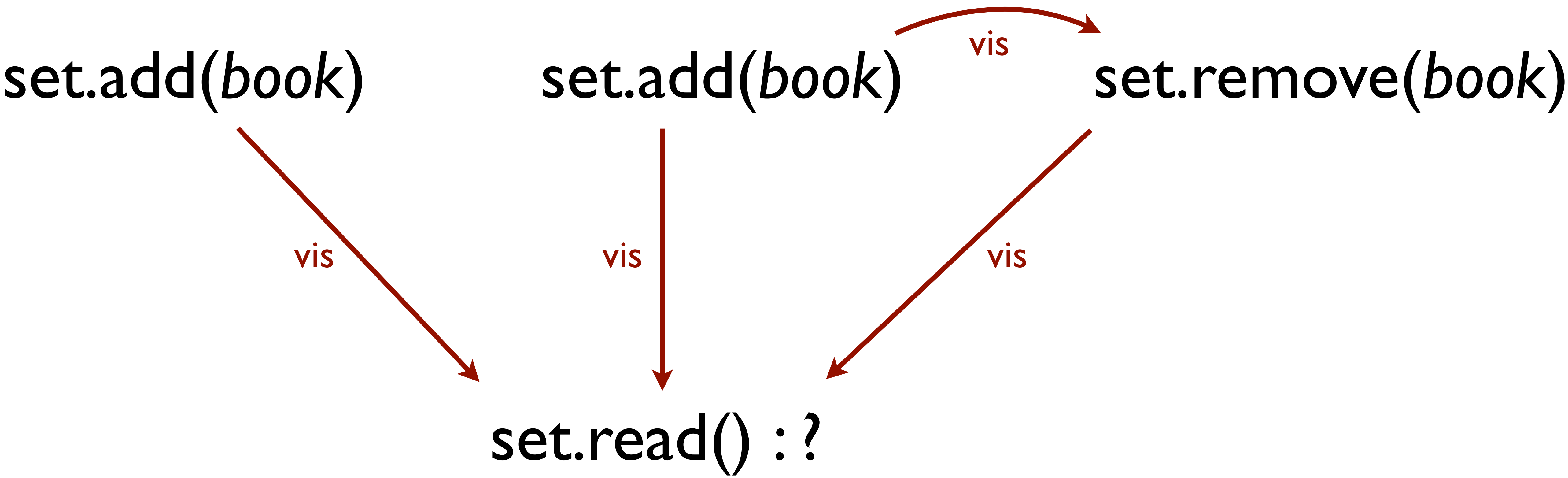
F: context of e → return value of e



F: discard all writes seen by a write

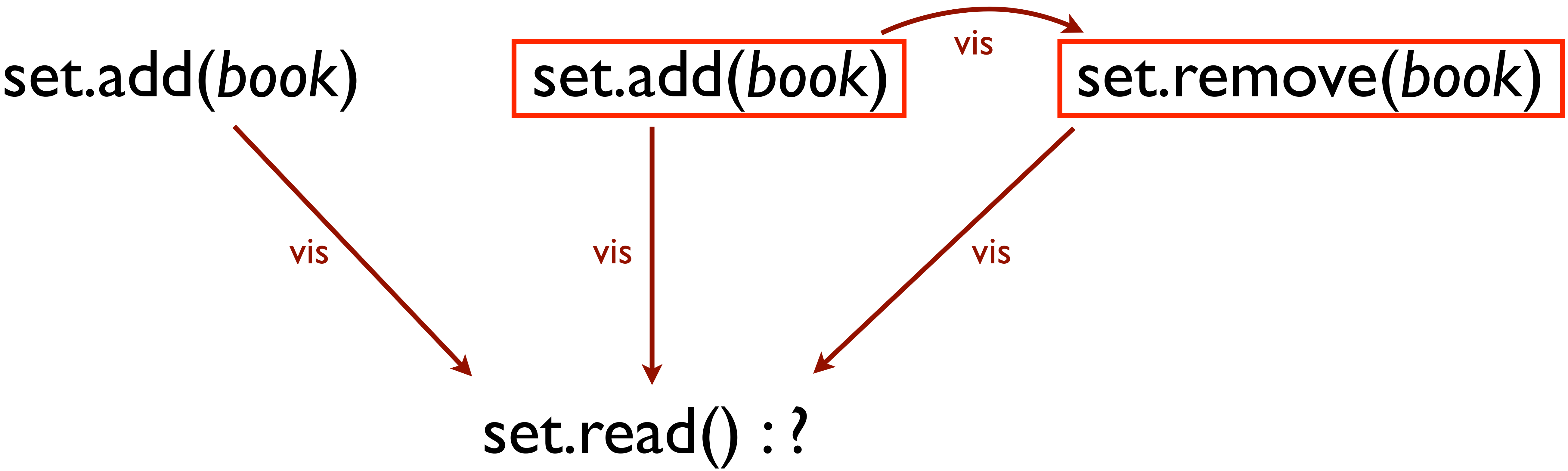
Add-wins set

F: context of e → return value of e



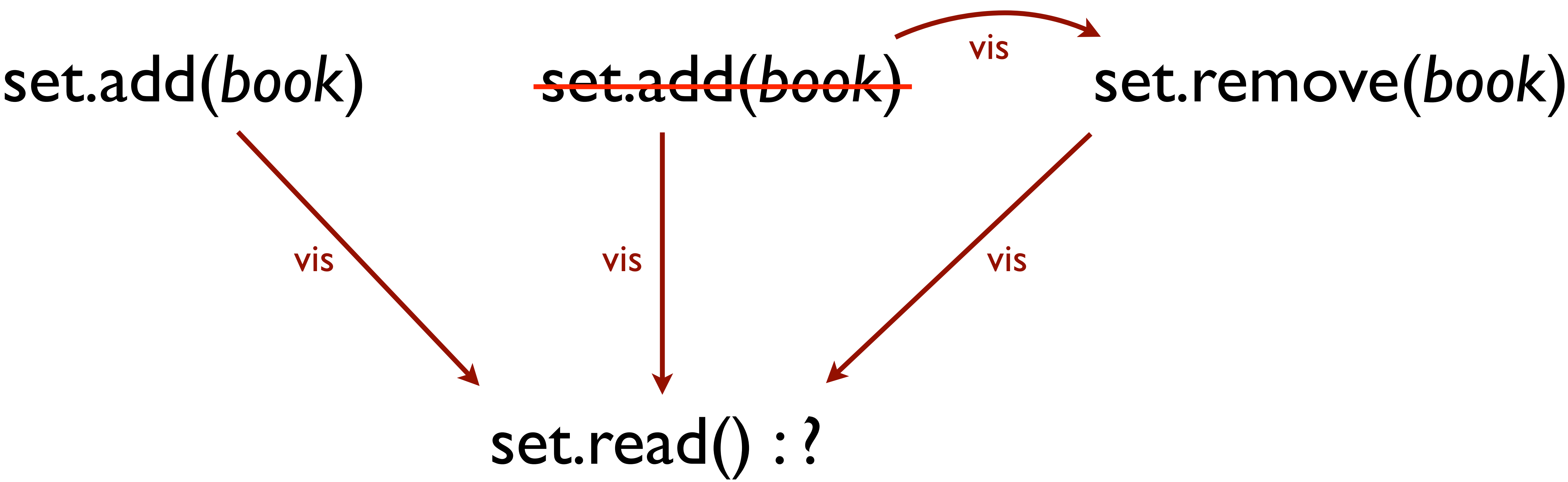
Add-wins set

F: context of e → return value of e



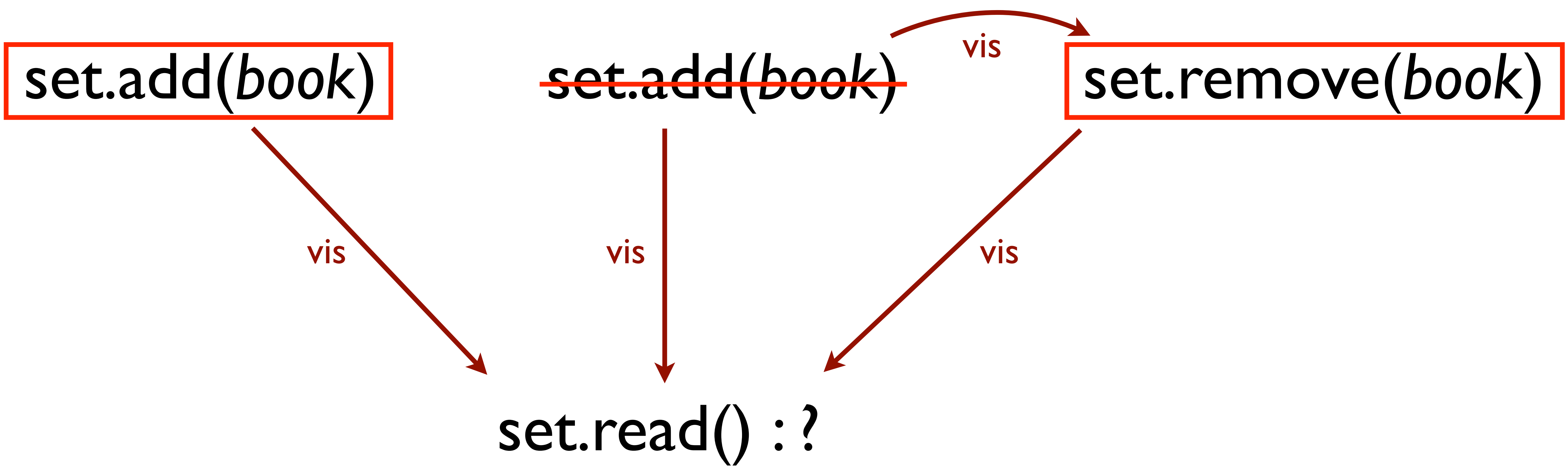
Add-wins set

F: context of e → return value of e



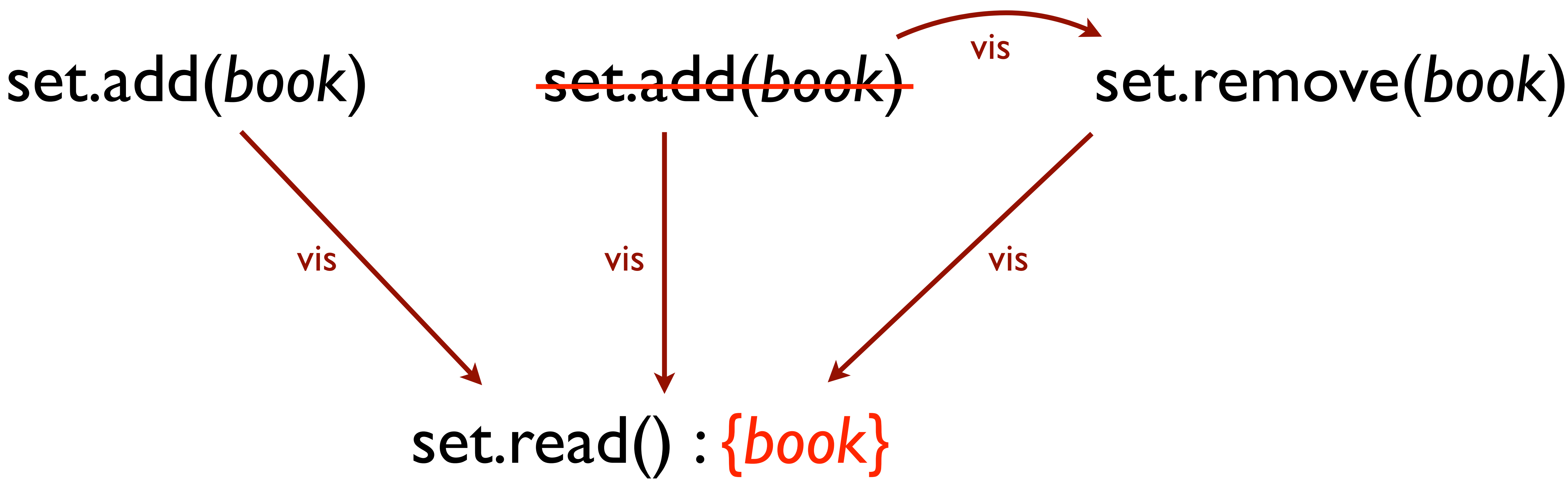
Add-wins set

F: context of e → return value of e



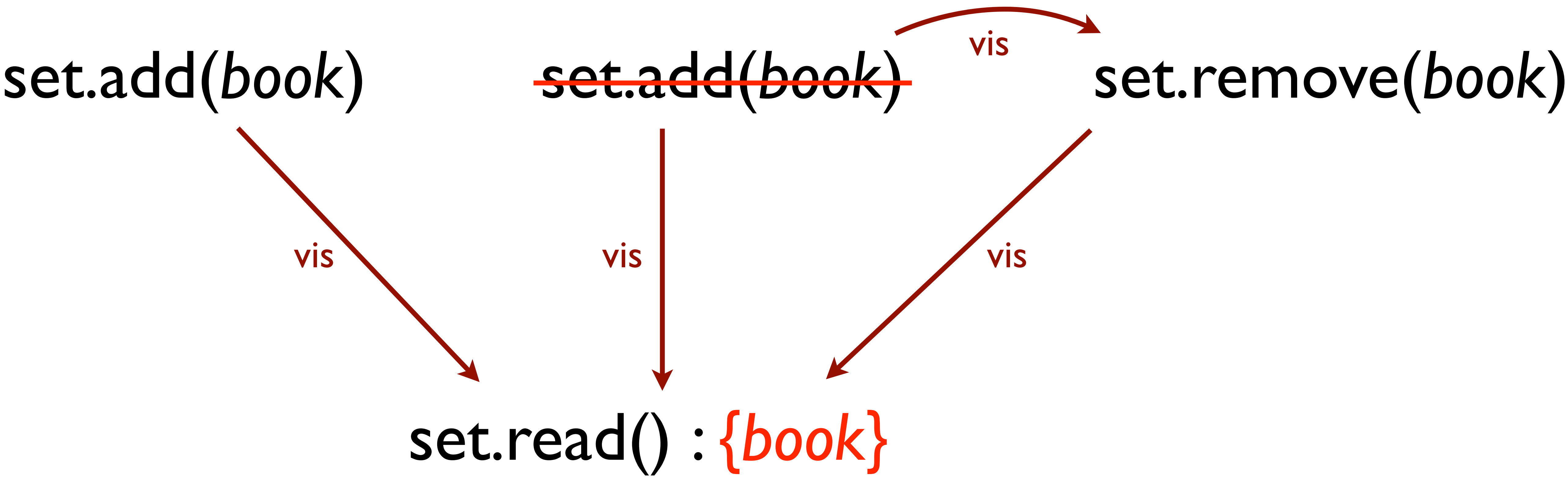
Add-wins set

F: context of e → return value of e



Add-wins set

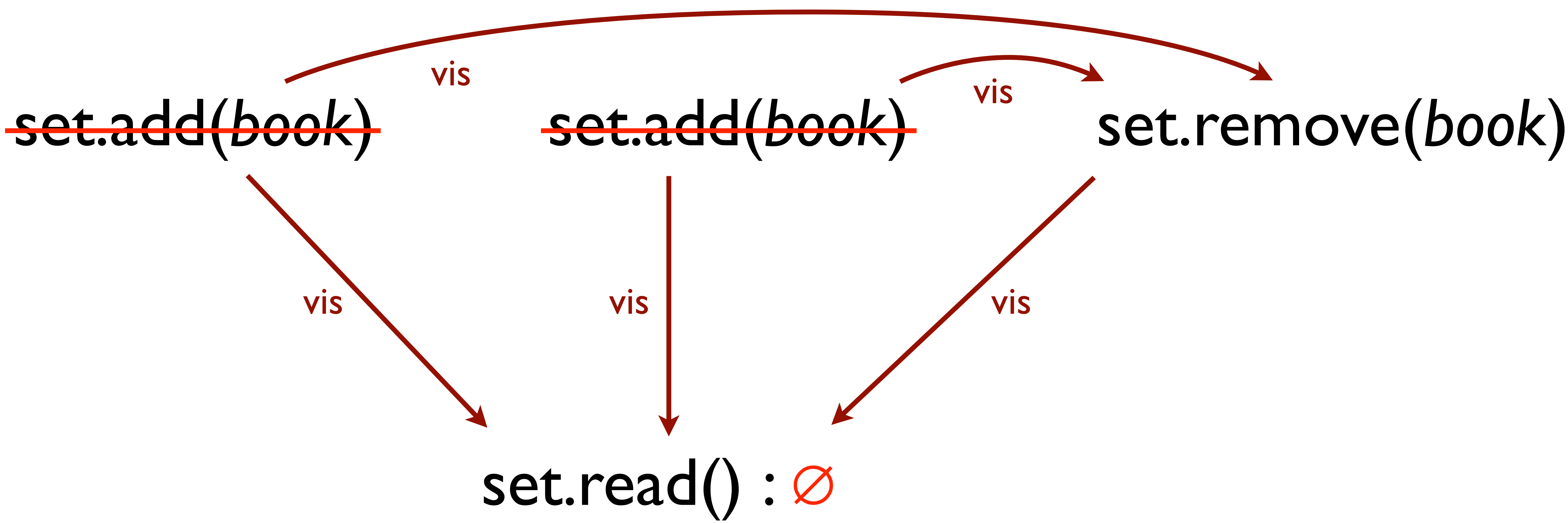
F: context of e → return value of e



F: cancel all adds seen by a remove

Add-wins set

F: context of e → return value of e



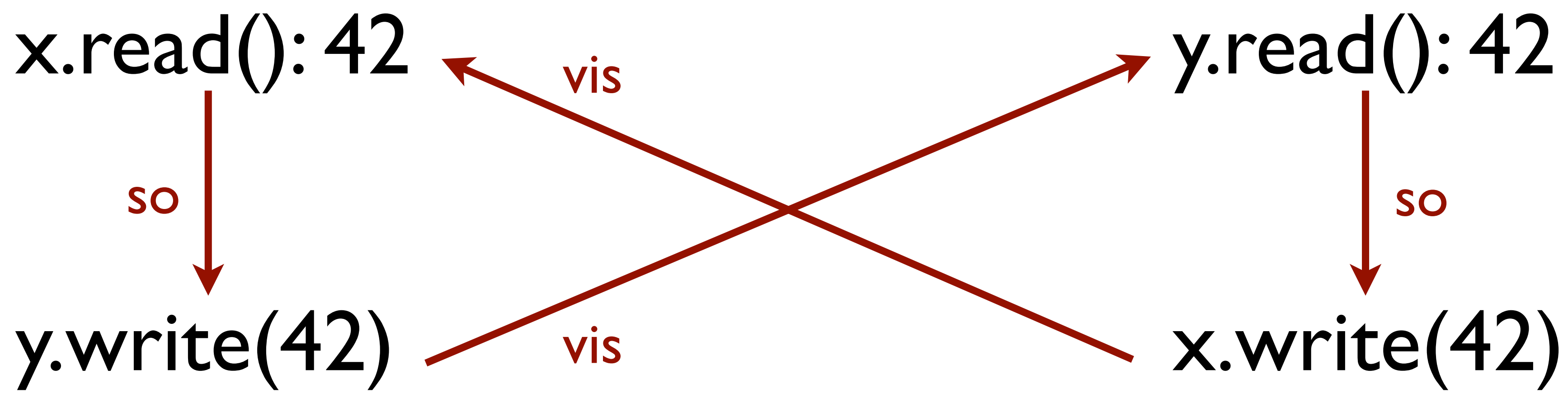
F: cancel all adds seen by a remove

Data type specification

F: context of e → return value of e

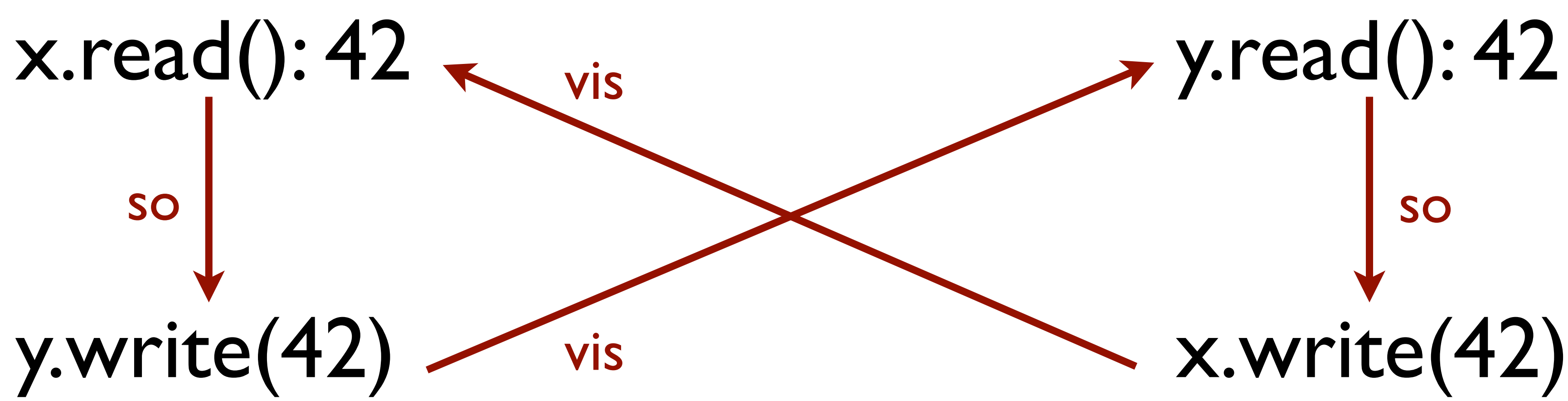
$\forall e \in E.\ rval(e) = F_{type(obj(e))}(context(e))$

"No causal cycles" axiom



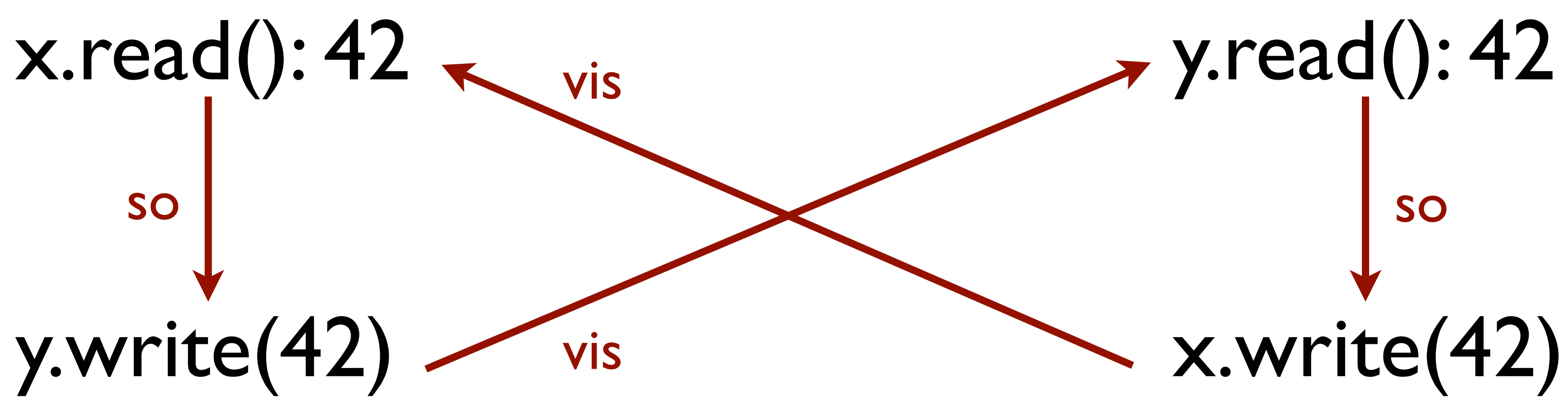
- so ∪ vis is acyclic: no causal cycles/out-of-thin-air values
- so and vis consistent with execution order

"No causal cycles" axiom



- so ∪ vis is acyclic: no causal cycles/out-of-thin-air values
- so and vis consistent with execution order
- Could result from speculative execution, uncommon in distributed systems

"No causal cycles" axiom



- so ∪ vis is acyclic: no causal cycles/out-of-thin-air values
- so and vis consistent with execution order
- Could result from speculative execution, uncommon in distributed systems
- Some forms allowed by shared-memory models (ARM, C++, Java): defining semantics is an open problem

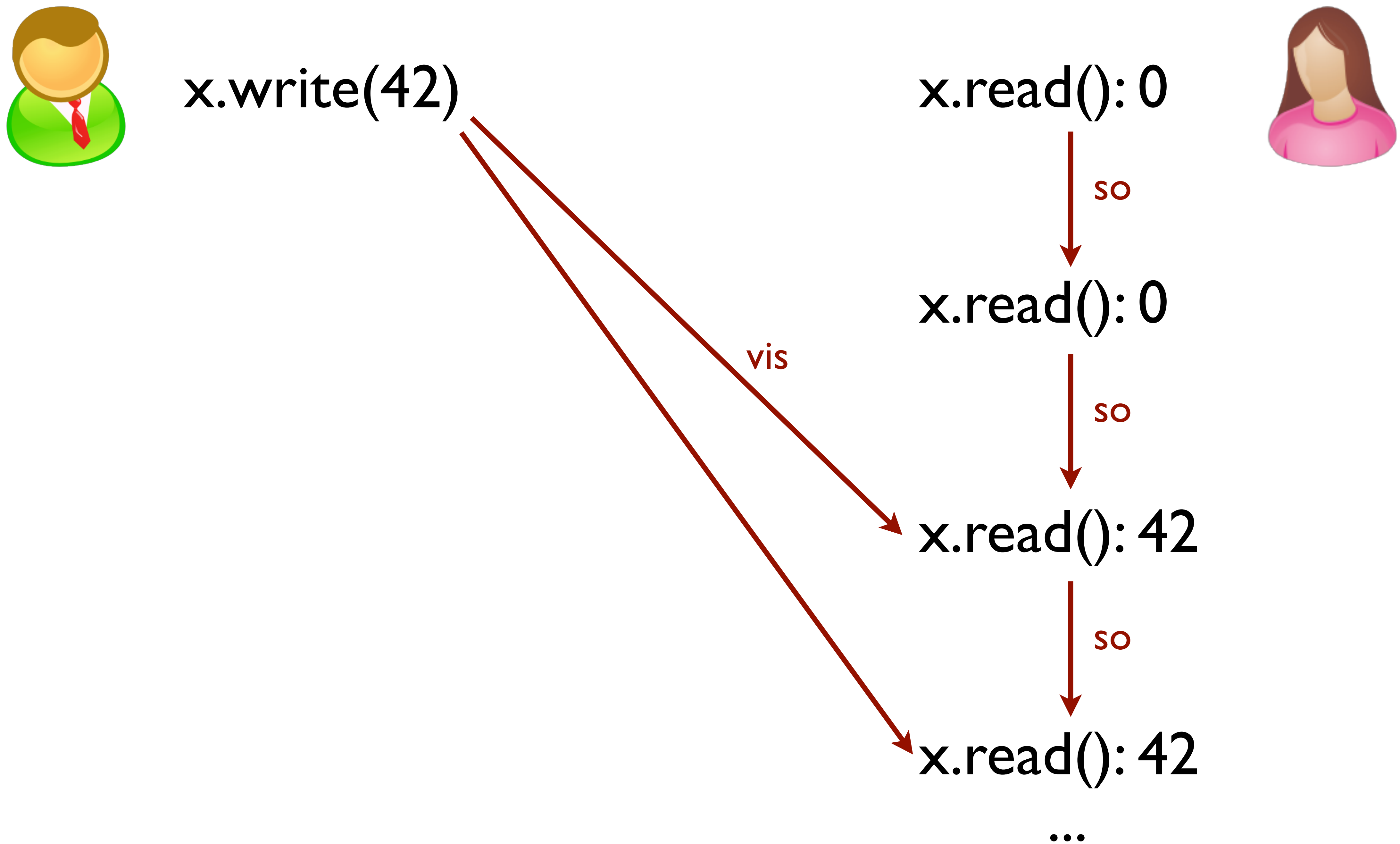
Eventual visibility

x.write(42)

$$\begin{array}{c} \text{x.read(): 0} \\ \Big\downarrow{\scriptstyle \text{so}} \\ \text{x.read(): 0} \\ \Big\downarrow{\scriptstyle \text{so}} \\ \text{x.read(): 0} \\ \Big\downarrow{\scriptstyle \text{so}} \\ \text{x.read(): 0} \\ \dots \end{array}$$

$\forall e \in E.\ e \xrightarrow{\text{vis}} f$ for all but finitely many $f \in E$

Eventual visibility



$\forall e \in E.\ e \xrightarrow{\text{vis}} f$ for all but finitely many $f \in E$

Eventual consistency summary

The set of histories (E, so) such that for some vis, ar:

- Return values consistent with data type specs:

  $\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$

- No causal cycles: so $\cup$ vis is acyclic

- Eventual visibility:

  $\forall e \in E.\ e \xrightarrow{\mathrm{vis}} f$ for all but finitely many $f \in E$

Eventual consistency summary

The set of histories (E, so) such that for some vis, ar:

- Return values consistent with data type specs:

  $\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$

- No causal cycles: so $\cup$ vis is acyclic

- Eventual visibility:

  $\forall e \in E.\ e \xrightarrow{\mathrm{vis}} f$ for all but finitely many $f \in E$

Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

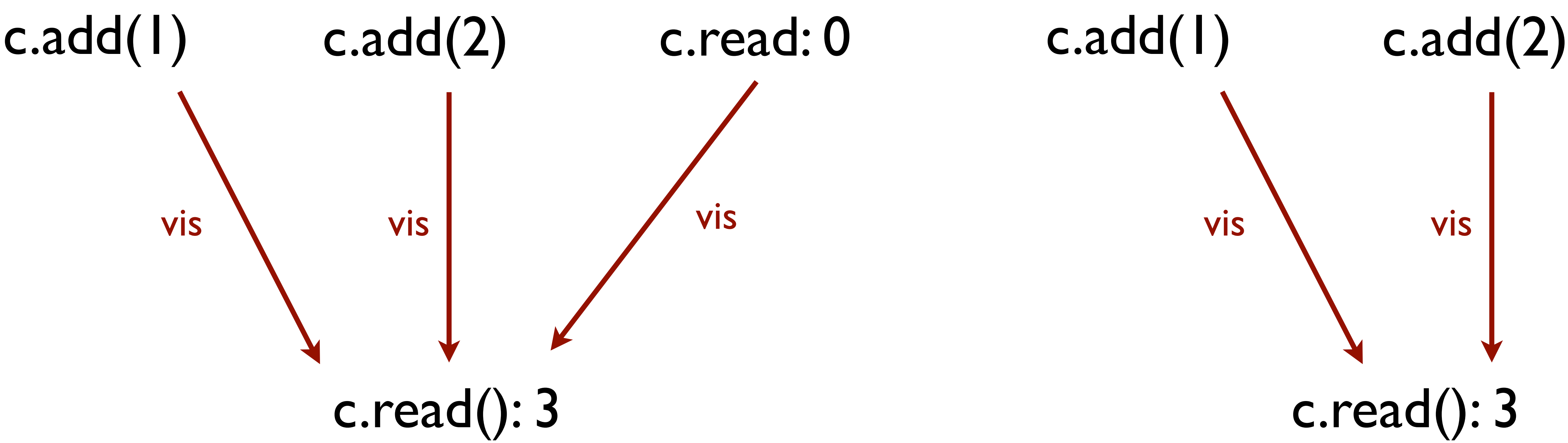
- **Convergence:** events with the same context return the same value: $\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$

Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

- Convergence: events with the same context return the same value: $\forall e \in E.\ rval(e) = F_{type(obj(e))}(context(e))$
- Assumption: deleting read-only operations from the context doesn't change the return value: F(X) = F(X-ReadOnlyEvents)

Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

- **Convergence:** events with the same context return the same value: $\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$
- **Assumption:** deleting read-only operations from the context doesn't change the return value: F(X) = F(X-ReadOnlyEvents)



Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

- Convergence: events with the same context return the same value: $\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$
- Assumption: deleting read-only operations from the context doesn't change the return value: $F(X) = F(X\text{-ReadOnlyEvents})$

Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

- Convergence: events with the same context return the same value: $\forall e \in E.\ rval(e) = F_{type(obj(e))}(context(e))$
- Assumption: deleting read-only operations from the context doesn't change the return value: F(X) = F(X-ReadOnlyEvents)
- Convergence': two operations with the same context projection to updates return the same value

Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

- Convergence: events with the same context return the same value: $\forall e \in E.\ rval(e) = F_{type(obj(e))}(context(e))$
- Assumption: deleting read-only operations from the context doesn't change the return value: $F(X) = F(X\text{-}ReadOnlyEvents)$
- Convergence': two operations with the same context projection to updates return the same value
- Eventual visibility: each update is seen by all but finitely many ops

Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

- Convergence: events with the same context return the same value: $\forall e \in E.\ rval(e) = F_{type(obj(e))}(context(e))$
- Assumption: deleting read-only operations from the context doesn't change the return value: F(X) = F(X-ReadOnlyEvents)
- Convergence': two operations with the same context projection to updates return the same value
- Eventual visibility: each update is seen by all but finitely many ops
- Assuming finitely many updates, all but finitely many ops will see all of these updates

Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

- **Convergence:** events with the same context return the same value: $\forall e \in E.\ rval(e) = F_{type(obj(e))}(context(e))$
- **Assumption:** deleting read-only operations from the context doesn't change the return value: F(X) = F(X-ReadOnlyEvents)
- **Convergence':** two operations with the same context projection to updates return the same value
- **Eventual visibility:** each update is seen by all but finitely many ops
- Assuming finitely many updates, all but finitely many ops will see all of these updates
- **Quiescent consistency:** assuming finitely many updates, all but finitely many operations on a given object return values computed based on the same context: same op $\Longrightarrow$ same rval

Eventual consistency summary

The set of histories (E, so) such that for some vis, ar:

- Return values consistent with data type specs:

  $\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$

- No causal cycles: so $\cup$ vis is acyclic

- Eventual visibility:

  $\forall e \in E.\ e \xrightarrow{\mathrm{vis}} f$ for all but finitely many $f \in E$

Eventual consistency summary

The set of histories (E, so) such that for some vis, ar:

- Return values consistent with data type specs:

  $\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$

- No causal cycles: so $\cup$ vis is acyclic

- Eventual visibility:

  $\forall e \in E.\ e \xrightarrow{\mathrm{vis}} f$ for all but finitely many $f \in E$

Stronger than quiescent consistency, but still weak

Strengthen consistency by adding additional axioms on vis and ar

Why is this spec sound wrt implementations?

...mmary

The set of histories (E, so) such that for some vis, ar:

- Return values consistent with data type specs:

  $\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$

- No causal cycles: so $\cup$ vis is acyclic

- Eventual visibility:

  $\forall e \in E.\ e \xrightarrow{\mathrm{vis}} f$ for all but finitely many $f \in E$

Stronger than quiescent consistency, but still weak

Strengthen consistency by adding additional axioms on vis and ar

Specification soundness

The set of all histories (E, so) such that for some vis, ar the abstract execution (E, so, vis, ar) satisfies consistency axioms $\mathscr{A}$

Specification soundness

The set of all histories (E, so) such that for some vis, ar the abstract execution (E, so, vis, ar) satisfies consistency axioms $\mathscr{A}$

The set of all histories (E, so) produced by arbitrary client interactions with the data type implementations with any allowed message deliveries

Specification soundness

The set of all histories (E, so) such that for some vis, ar the abstract execution (E, so, vis, ar) satisfies consistency axioms $\mathscr{A}$

$$\supseteq$$

The set of all histories (E, so) produced by arbitrary client interactions with the data type implementations with any allowed message deliveries

Specification soundness

The set of all histories (E, so) such that for some vis, ar the abstract execution (E, so, vis, ar) satisfies consistency axioms $\mathscr{A}$

$\supseteq$

The set of all histories (E, so) produced by arbitrary client interactions with the data type implementations with any allowed message deliveries

- $\forall$ concrete execution of the implementation with a history (E, so)
- $\exists$ vis, ar. (E, so, vis, ar) satisfies the axioms $\mathscr{A}$

Specification soundness

- Proofs depend on replicated data types
- Example: replicated counters and last-writer-wins registers
- There are also generic proof techniques that work for whole classes of data types

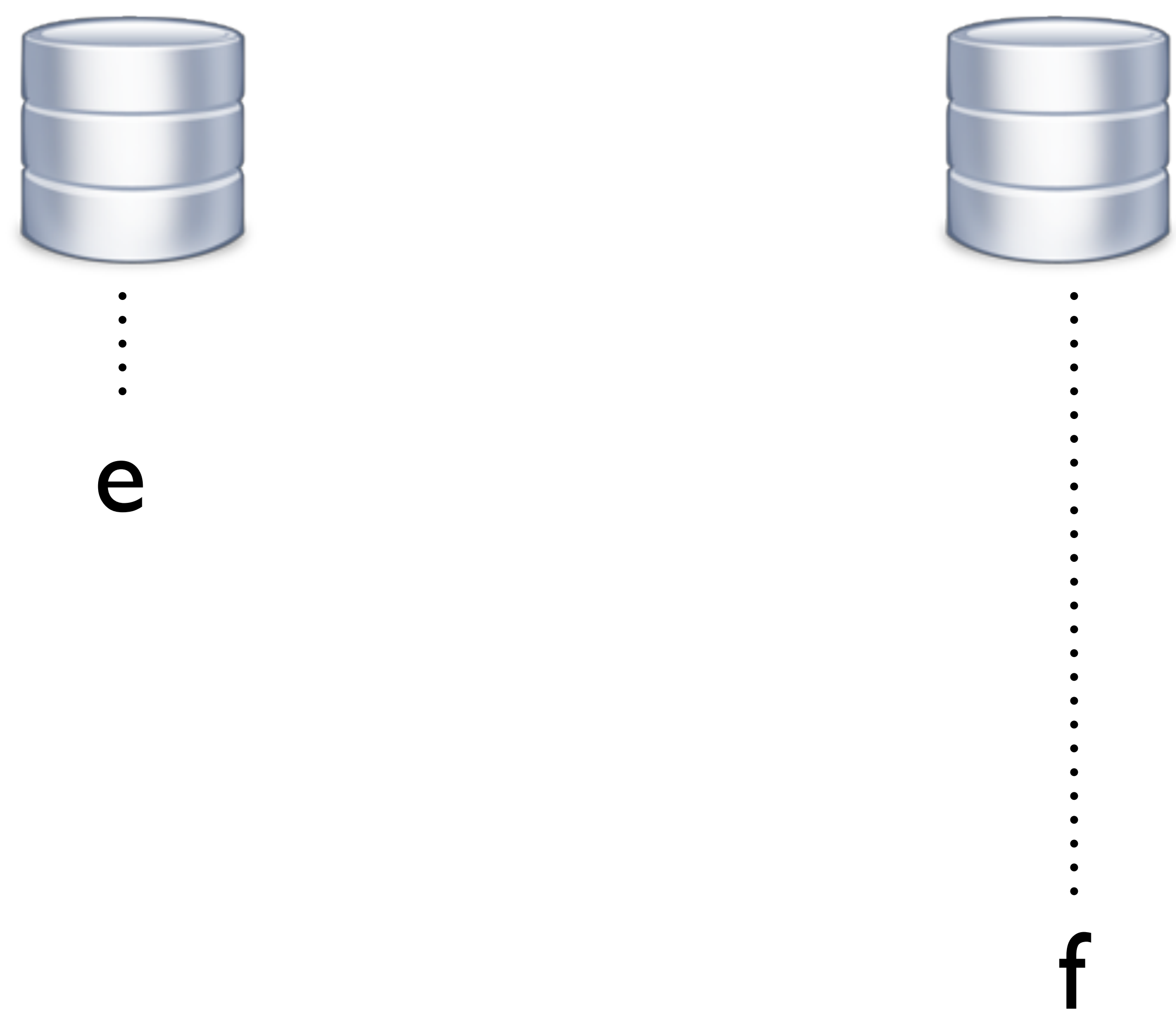
- $\forall$ concrete execution of the implementation with a history (E, so)
- $\exists$ vis, ar. (E, so, vis, ar) satisfies the axioms $\mathscr{A}$

Specification soundness

- Proofs depend on replicated data types
- Example: replicated counters and last-writer-wins registers
- There are also generic proof techniques that work for whole classes of data types

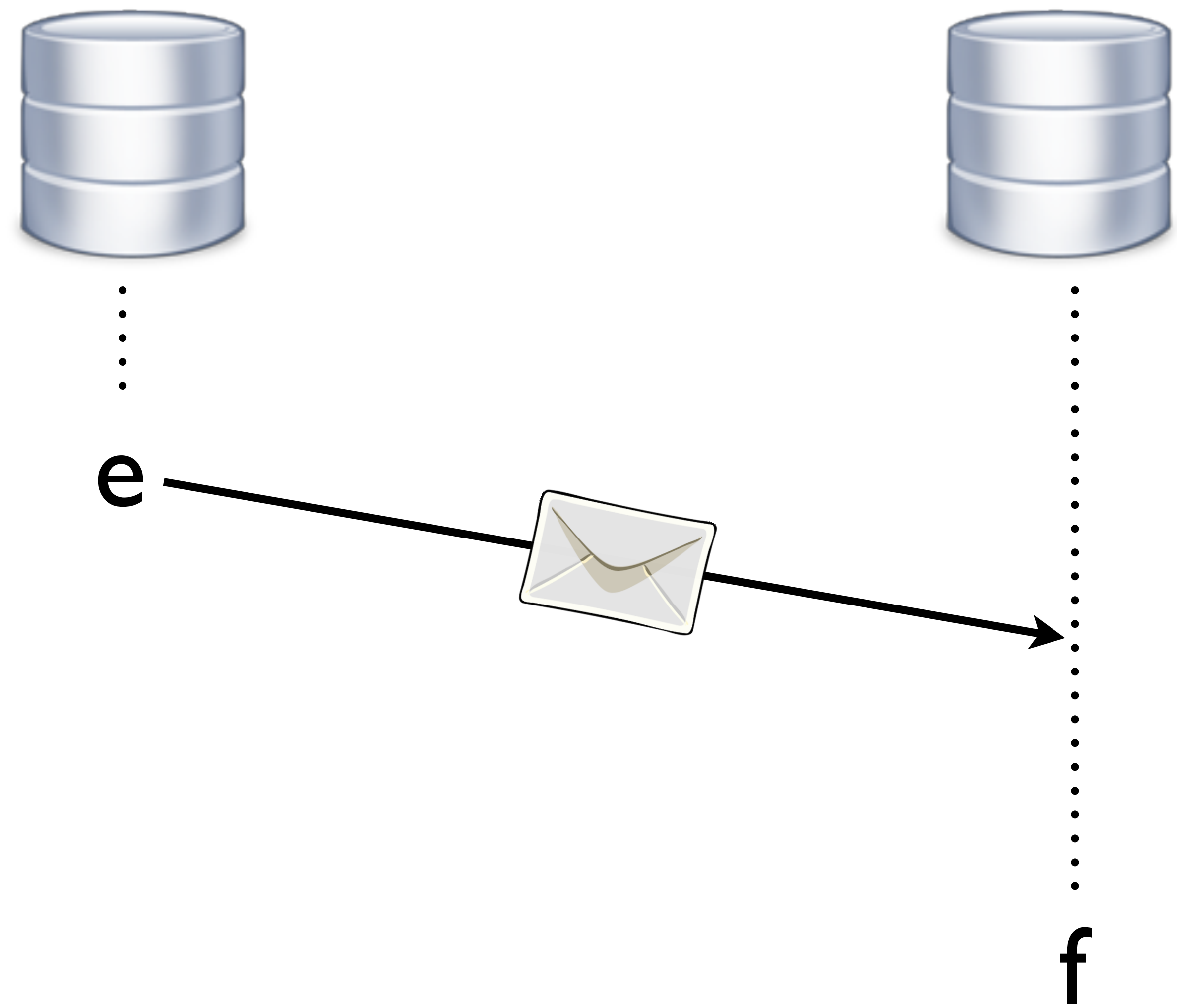
- $\forall$ concrete execution of the implementation with a history (E, so)
- $\exists$ vis, ar. (E, so, vis, ar) satisfies the axioms $\mathscr{A}$

Constructing vis



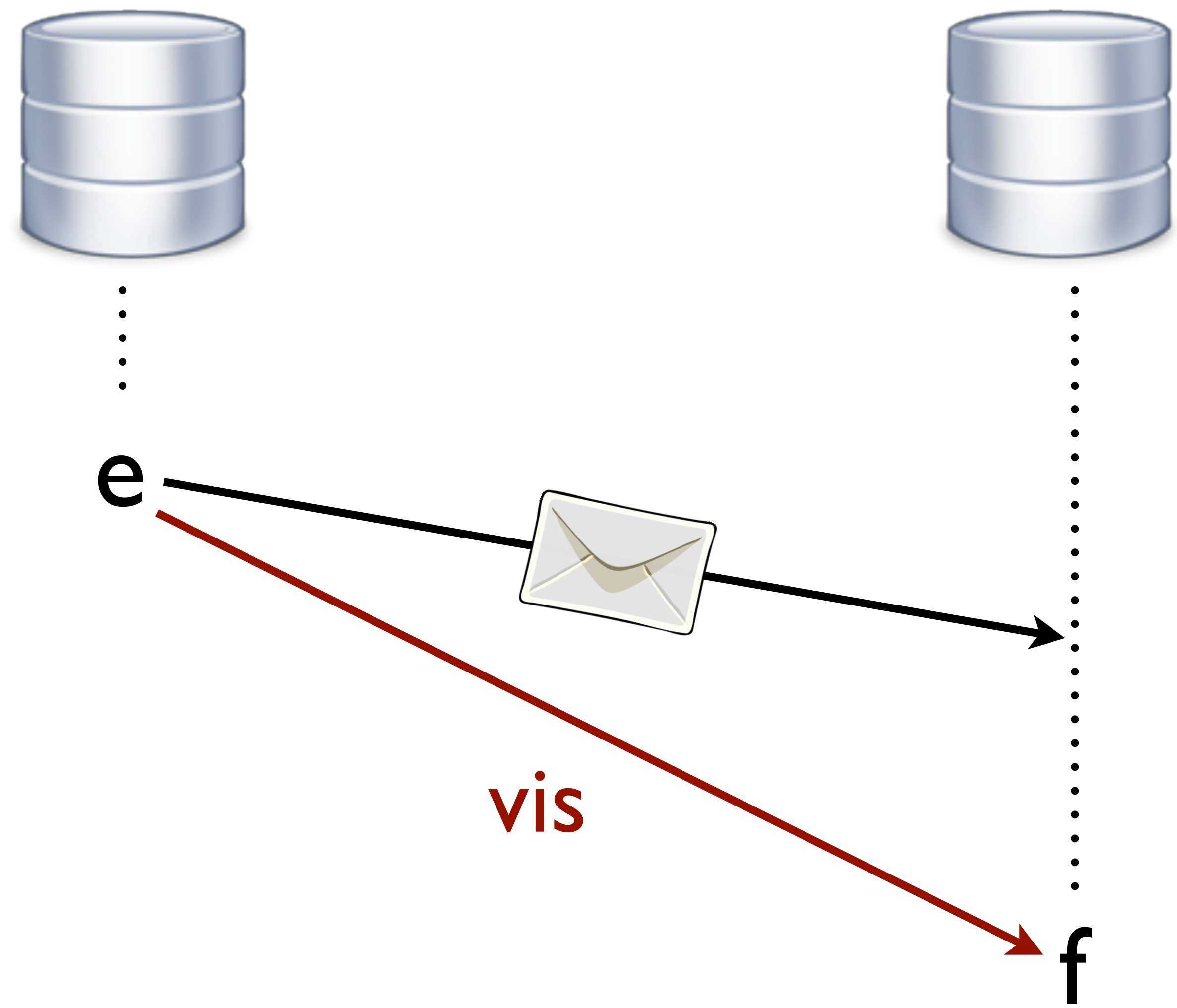
$e \xrightarrow{\text{vis}} f \iff$ effector of e delivered to replica of f before f is executed

Constructing vis



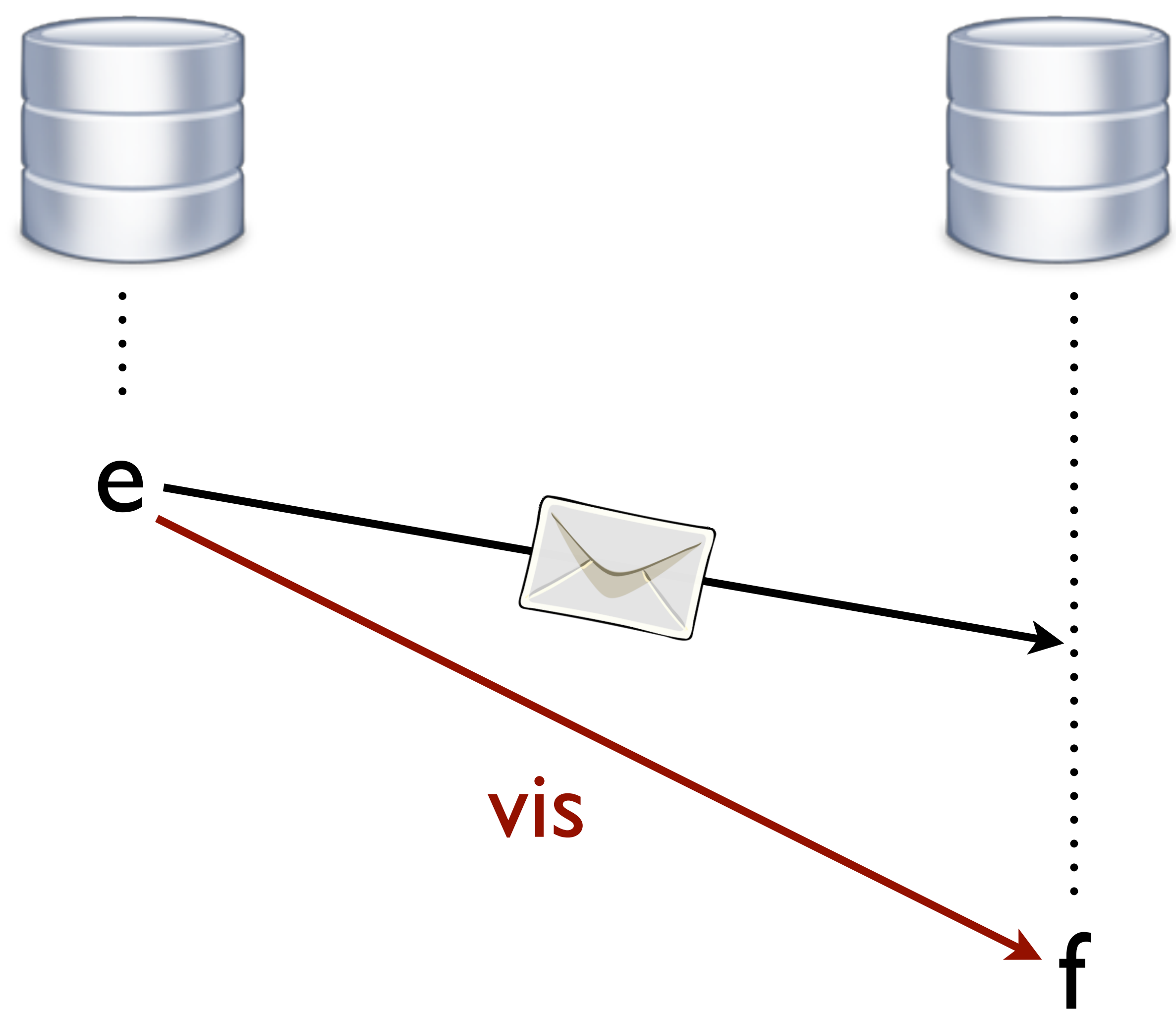
$e \xrightarrow{\text{vis}} f \iff$ effector of e delivered to replica of f before f is executed

Constructing vis



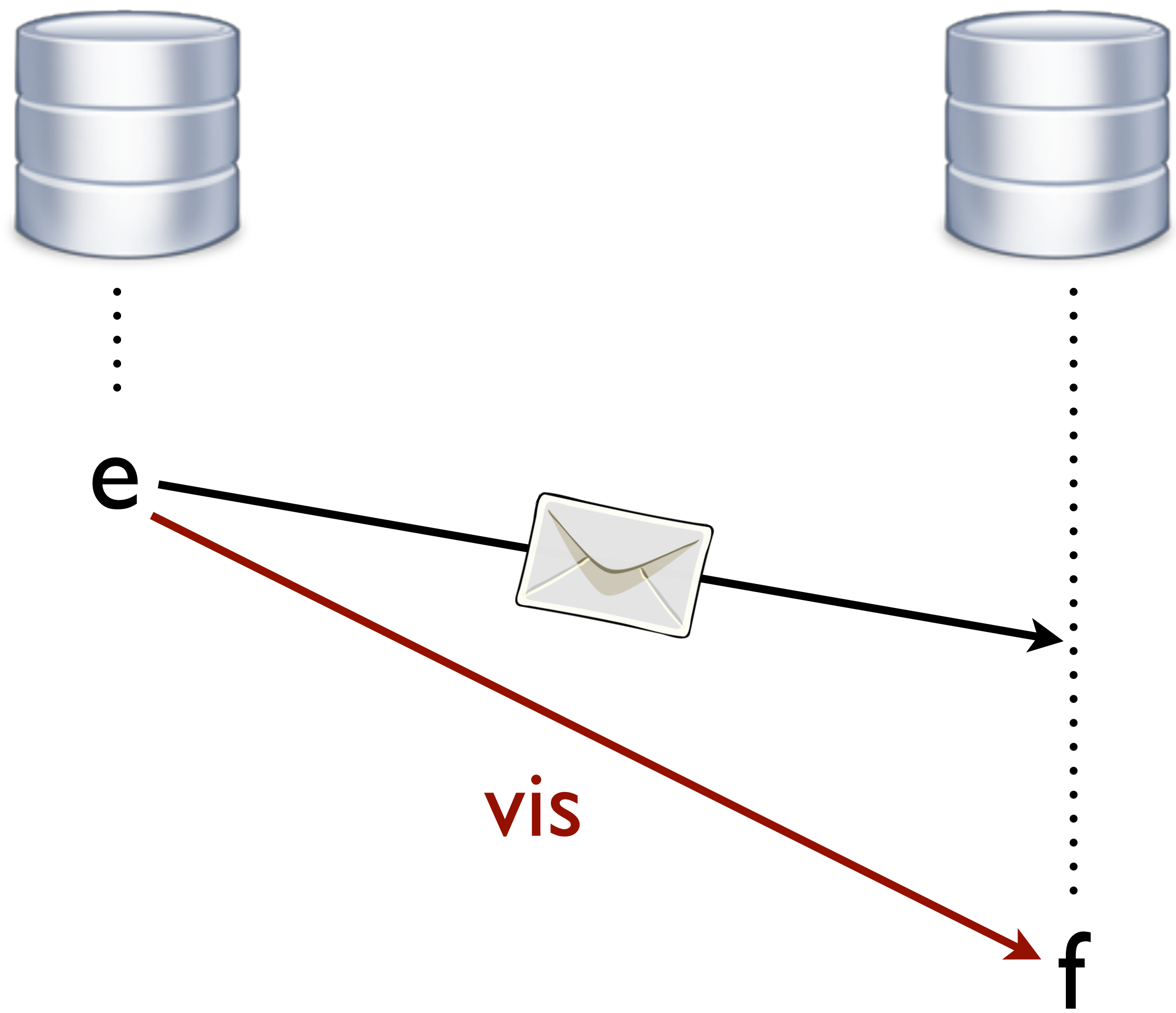
$e \xrightarrow{\text{vis}} f \iff$ effector of e delivered to replica of f before f is executed

so ∪ vis is acyclic?



$e \xrightarrow{\text{vis}} f \iff$ effector of e delivered to replica of f before f is executed

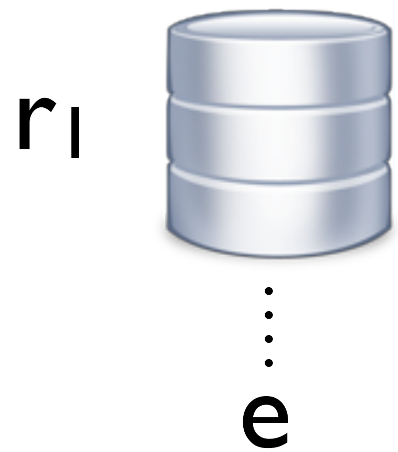
so ∪ vis is acyclic?



$e \xrightarrow{\text{vis}} f \vee e \xrightarrow{\text{so}} f \Longrightarrow$ e was issued before f in the operational execution

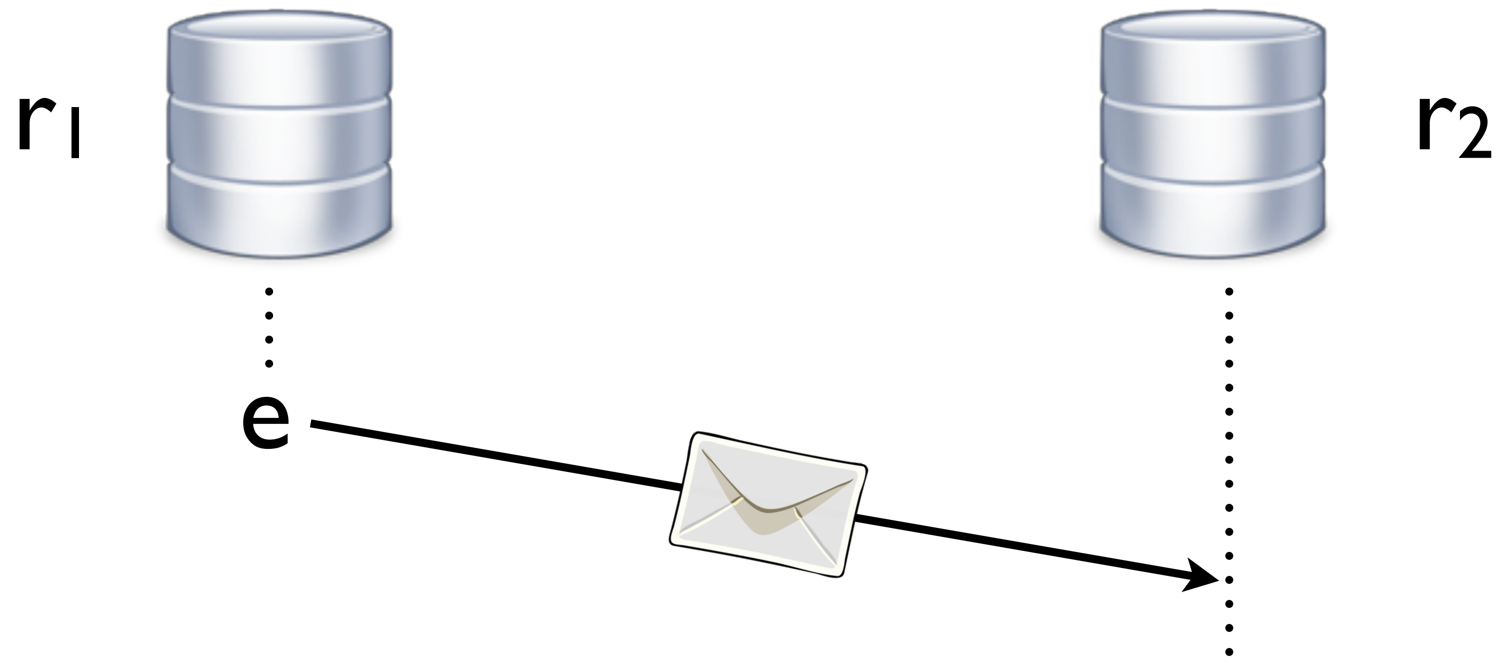
$\forall e \in E.\ e \xrightarrow{\text{vis}} f$ for all but finitely many $f \in E$

$$\forall e \in E.\ e \xrightarrow{\text{vis}} f \text{ for all but finitely many } f \in E$$



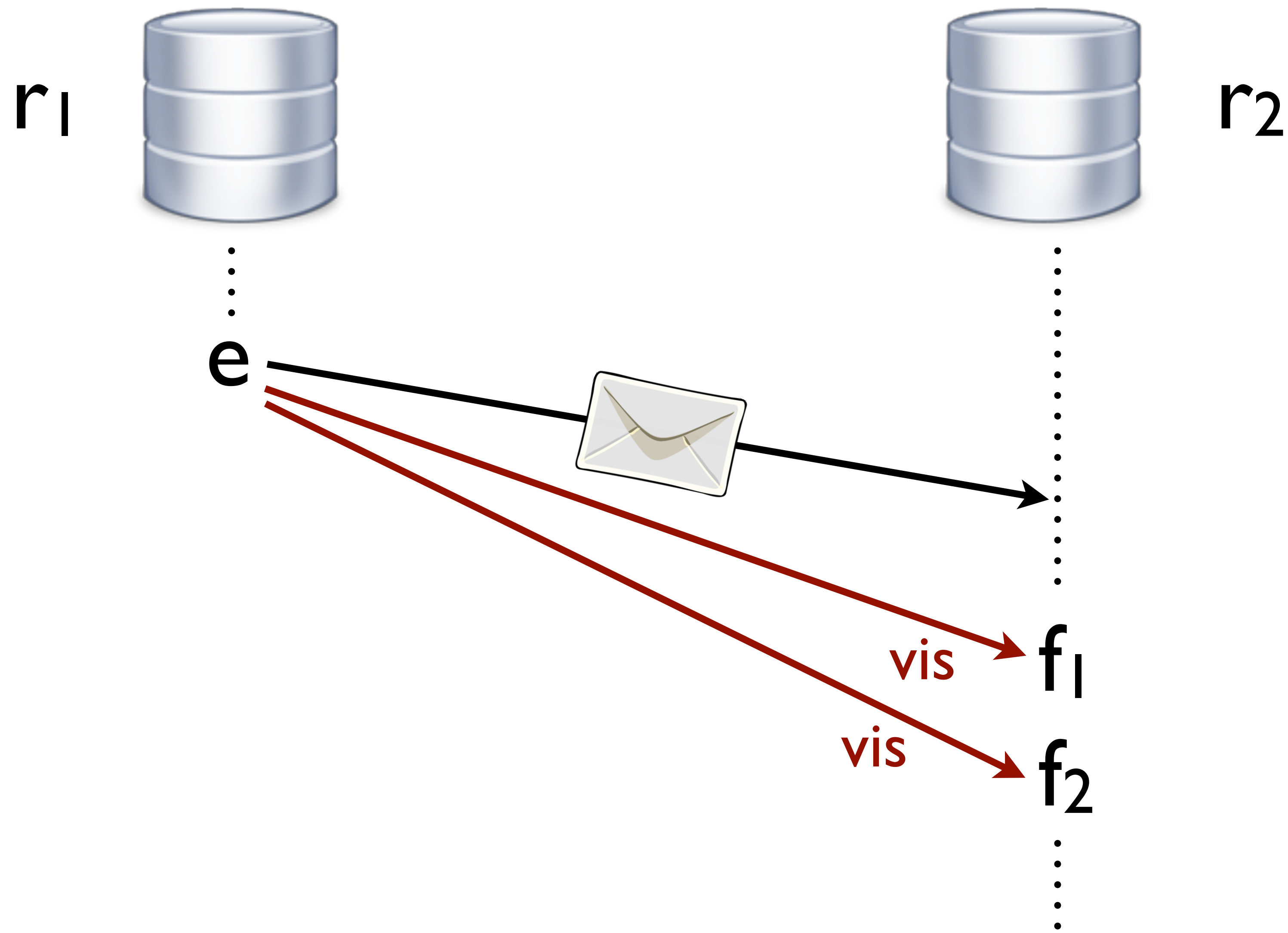
$$\forall e \in E.\ e \xrightarrow{\text{vis}} f \text{ for all but finitely many } f \in E$$

$\forall e \in E.\ e \xrightarrow{\text{vis}} f$ for all but finitely many $f \in E$



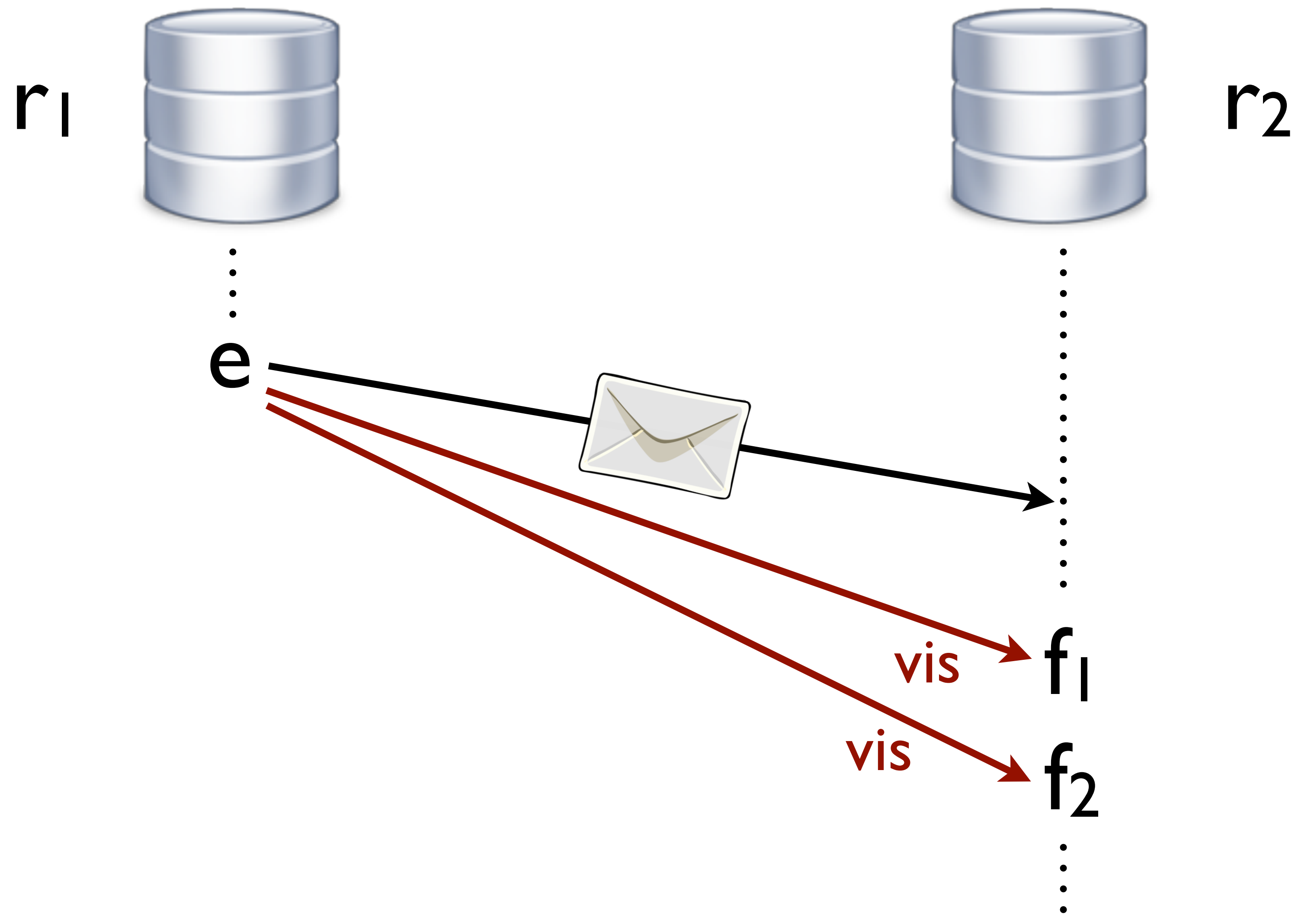
- Channels are reliable (every partition eventually heals) $\Longrightarrow$ the effector of e is eventually delivered to r_2

$$\forall e \in E.\ e \xrightarrow{\text{vis}} f \text{ for all but finitely many } f \in E$$



- Channels are reliable (every partition eventually heals) $\Longrightarrow$ the effector of e is eventually delivered to r_2
- From some point on, all events f_i at the replica r_2 see e

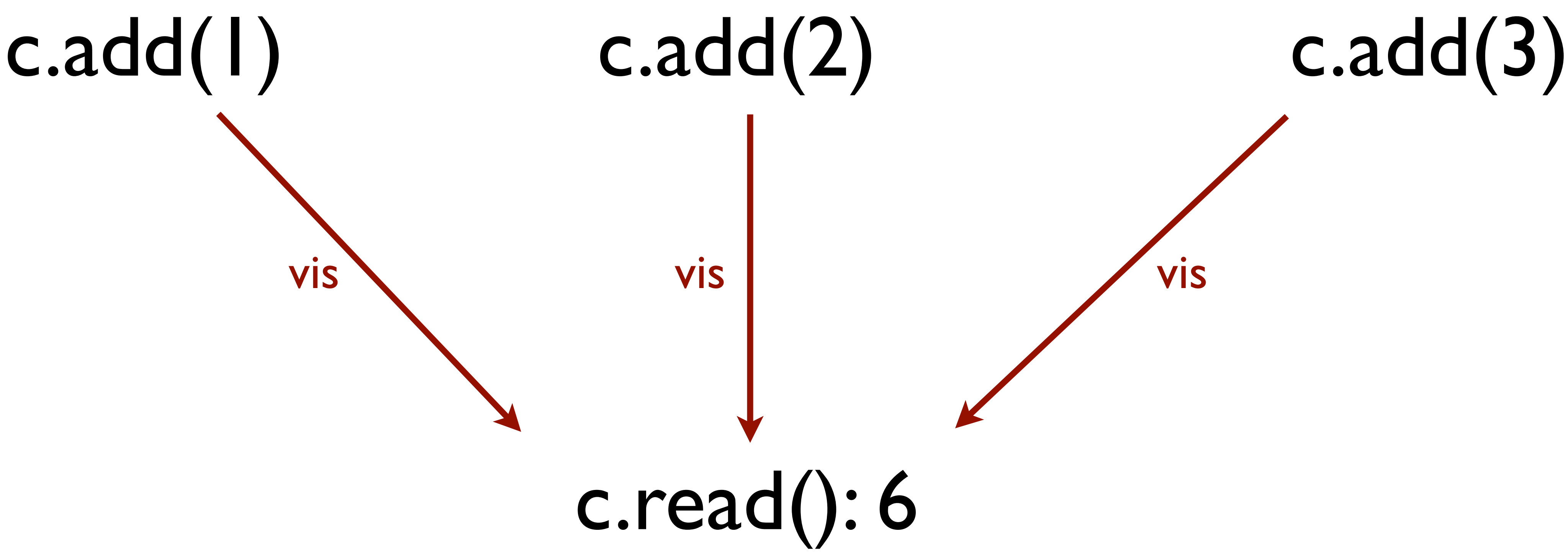
$$\forall e \in E.\ e \xrightarrow{\text{vis}} f \text{ for all but finitely many } f \in E$$



- Channels are reliable (every partition eventually heals) $\Longrightarrow$ the effector of e is eventually delivered to r_2
- From some point on, all events f_i at the replica r_2 see e
- True for any replica $\Longrightarrow$ only finitely many events don't see e

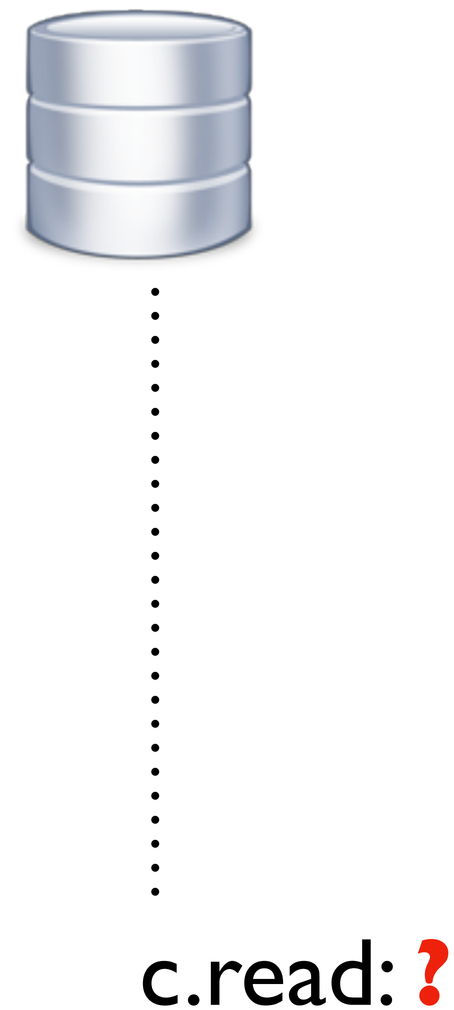
Correctness of counters

$$\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$$

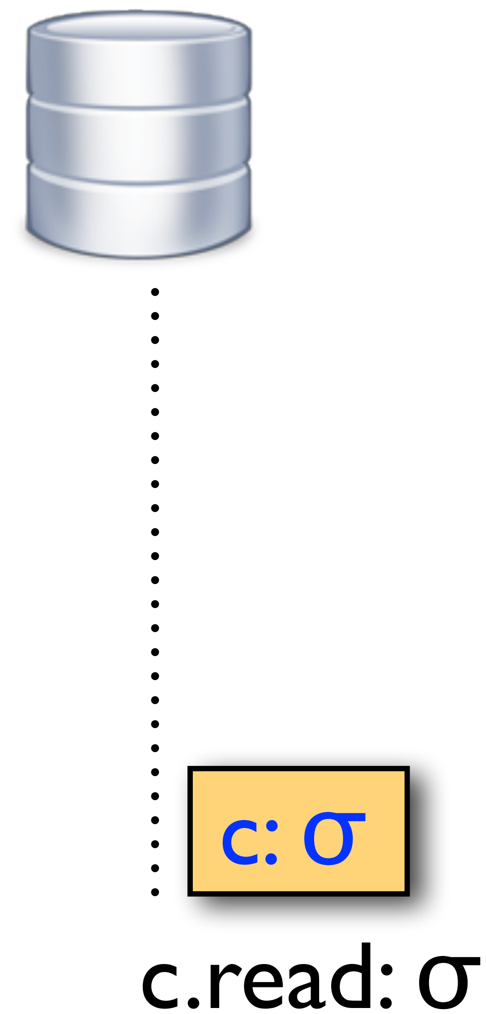


F: reads return the sum of all additions in the context

Correctness of counters



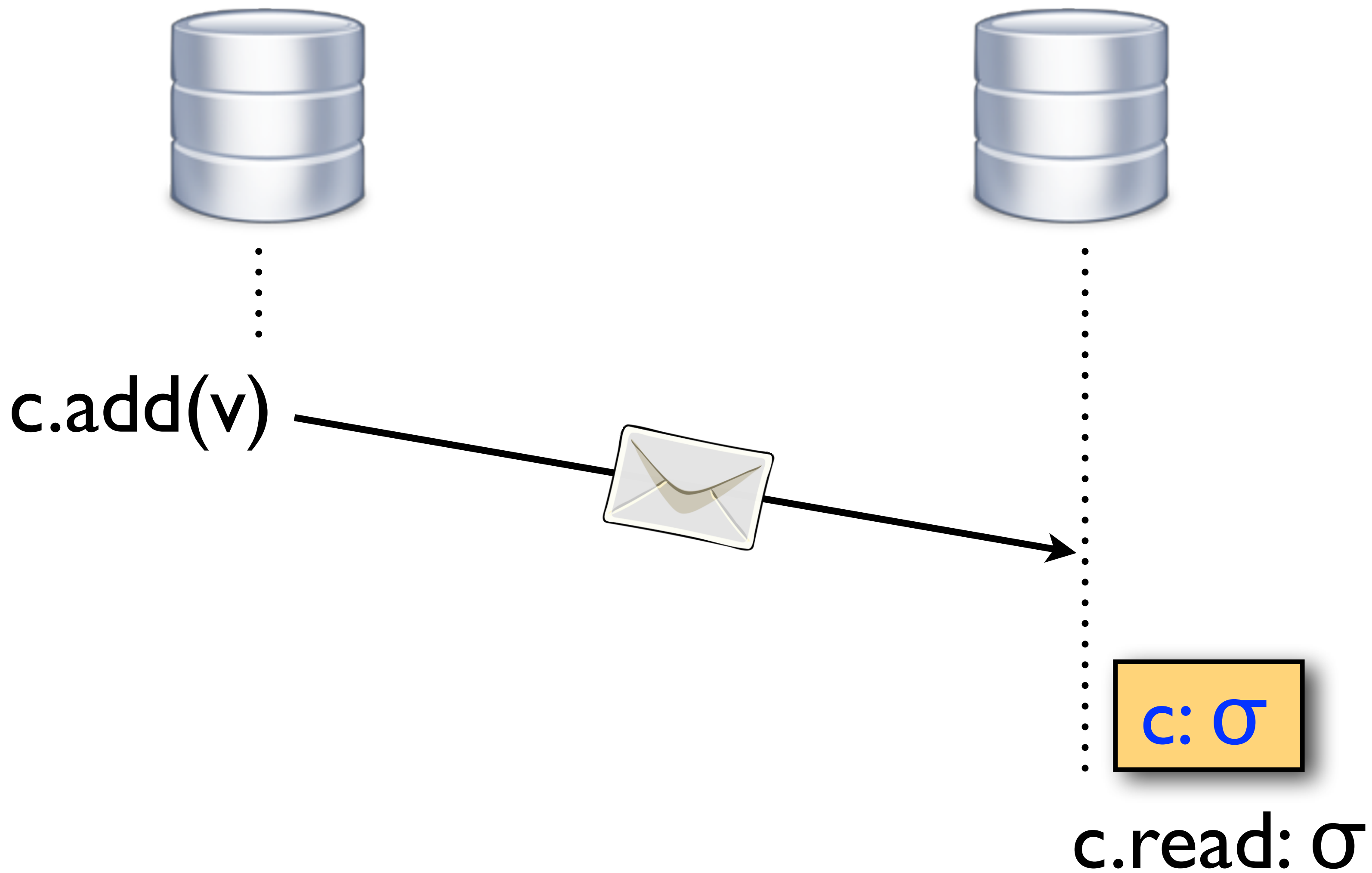
Correctness of counters



A read returns the value of the counter at the replica:

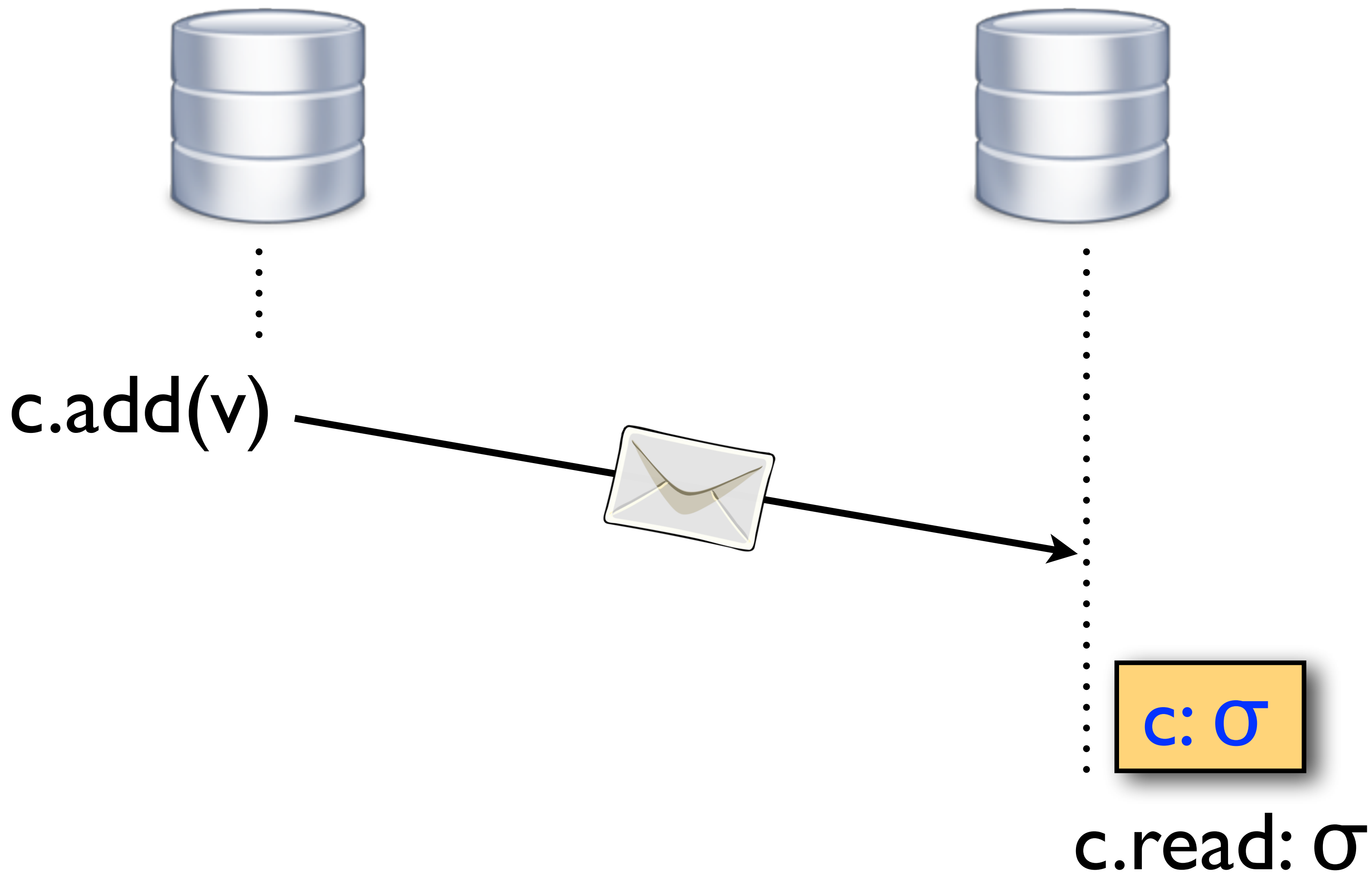
$[\![\mathrm{read}()]\!]_{\mathrm{val}}(\sigma) \;=\; \sigma$

Correctness of counters



Invariant: the value of a counter at a replica is the sum of all increments of the counter delivered to it

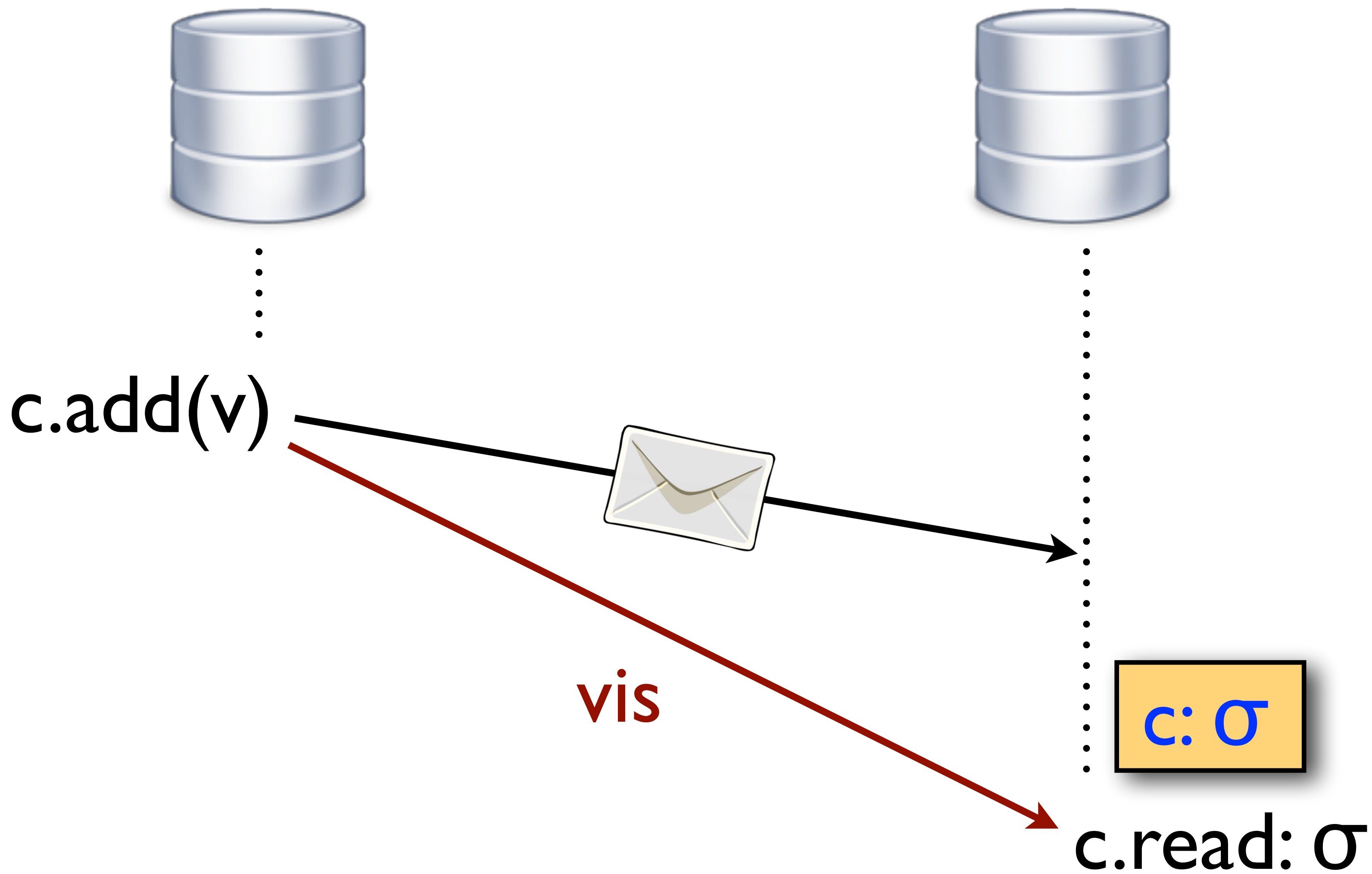
Correctness of counters



Invariant: the value of a counter at a replica is the sum of all increments of the counter delivered to it

$$[\![\text{add}(v)]\!]_{\text{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma' + v)$$

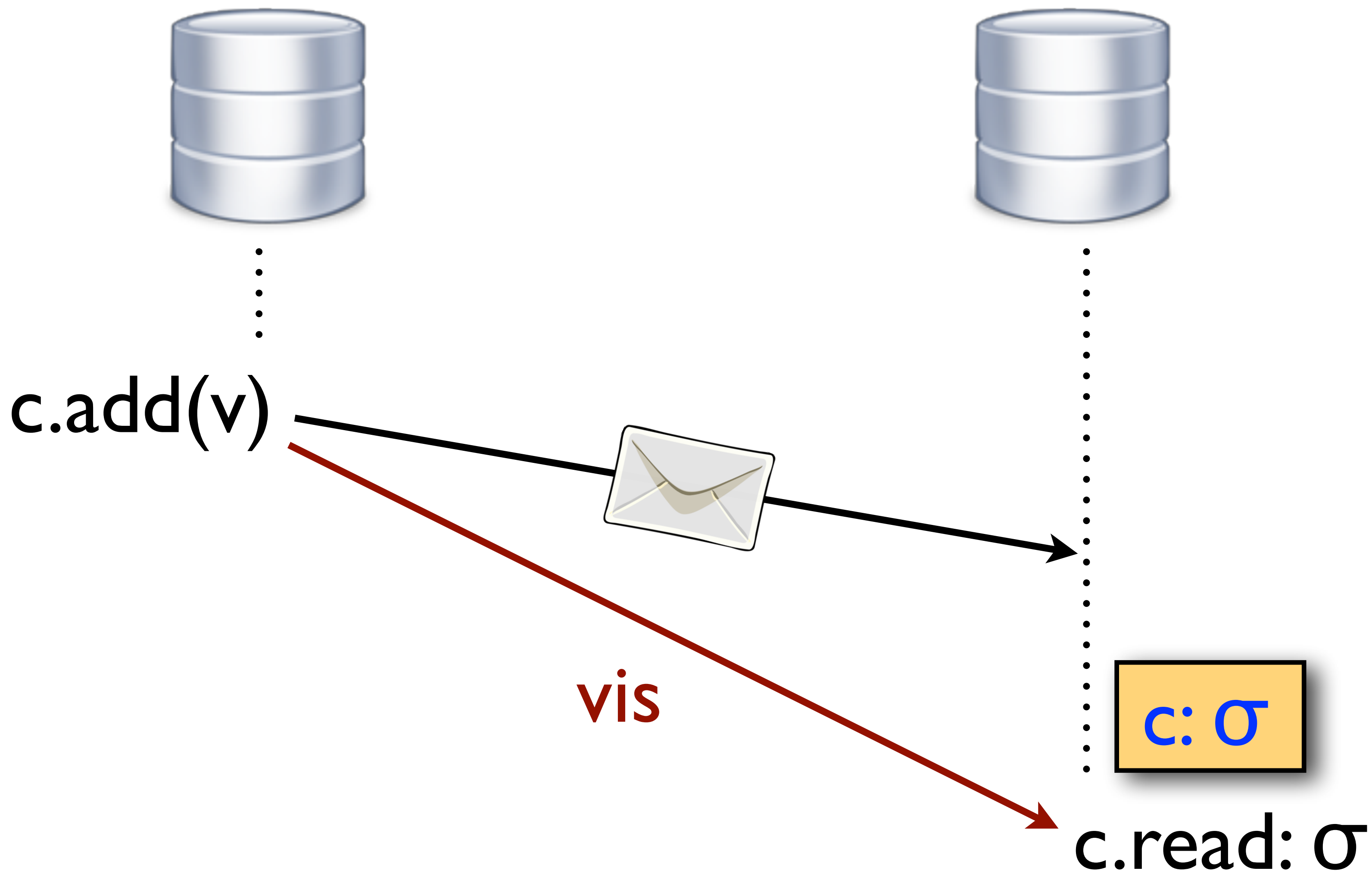
Correctness of counters



Invariant: the value of a counter at a replica is the sum of all increments of the counter delivered to it

$$[\![\mathrm{add}(v)]\!]_{\mathrm{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma' + v)$$

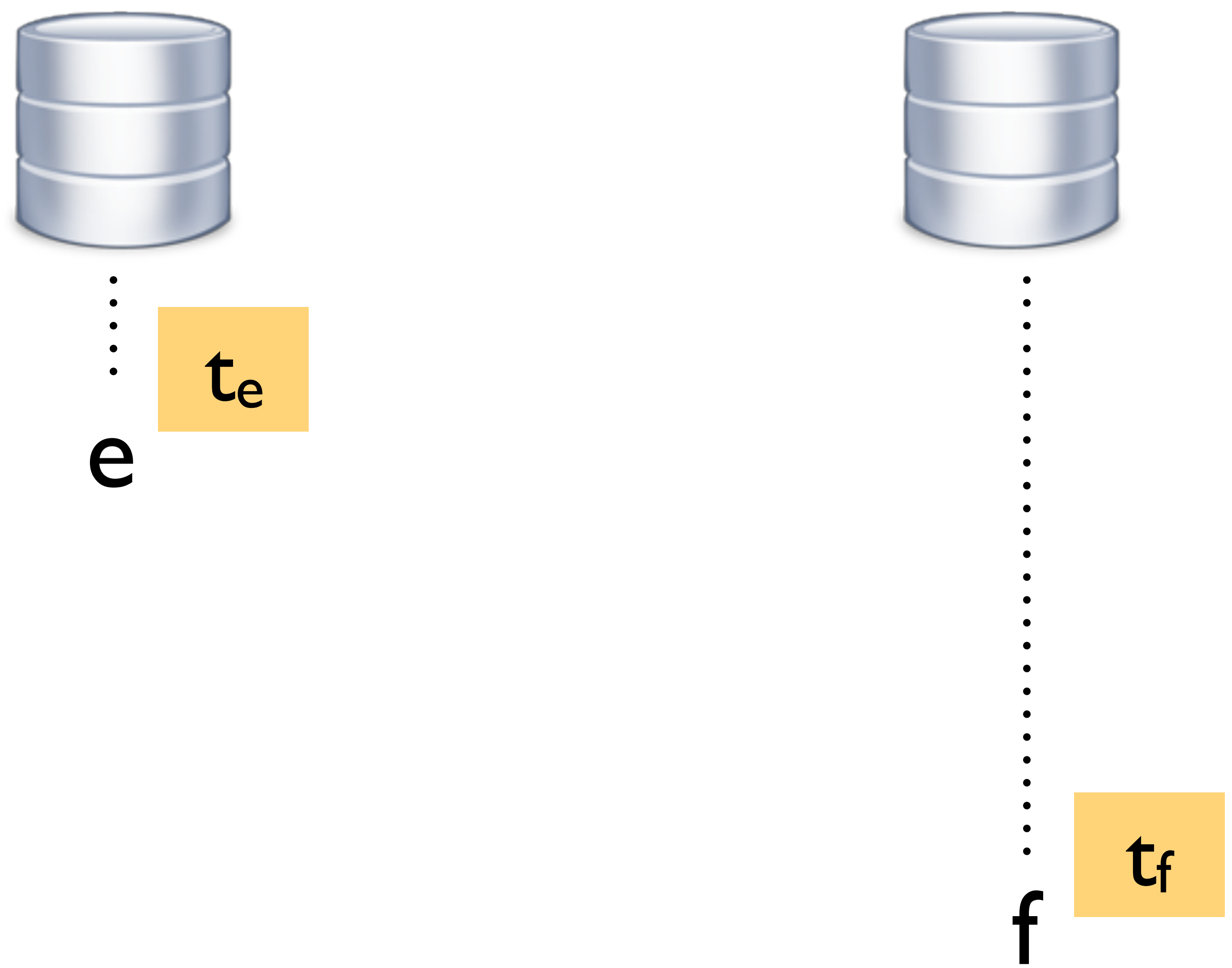
Correctness of counters



Invariant: the value of a counter at a replica is the sum of all increments of the counter delivered to it

= increments visible to the read, QED.

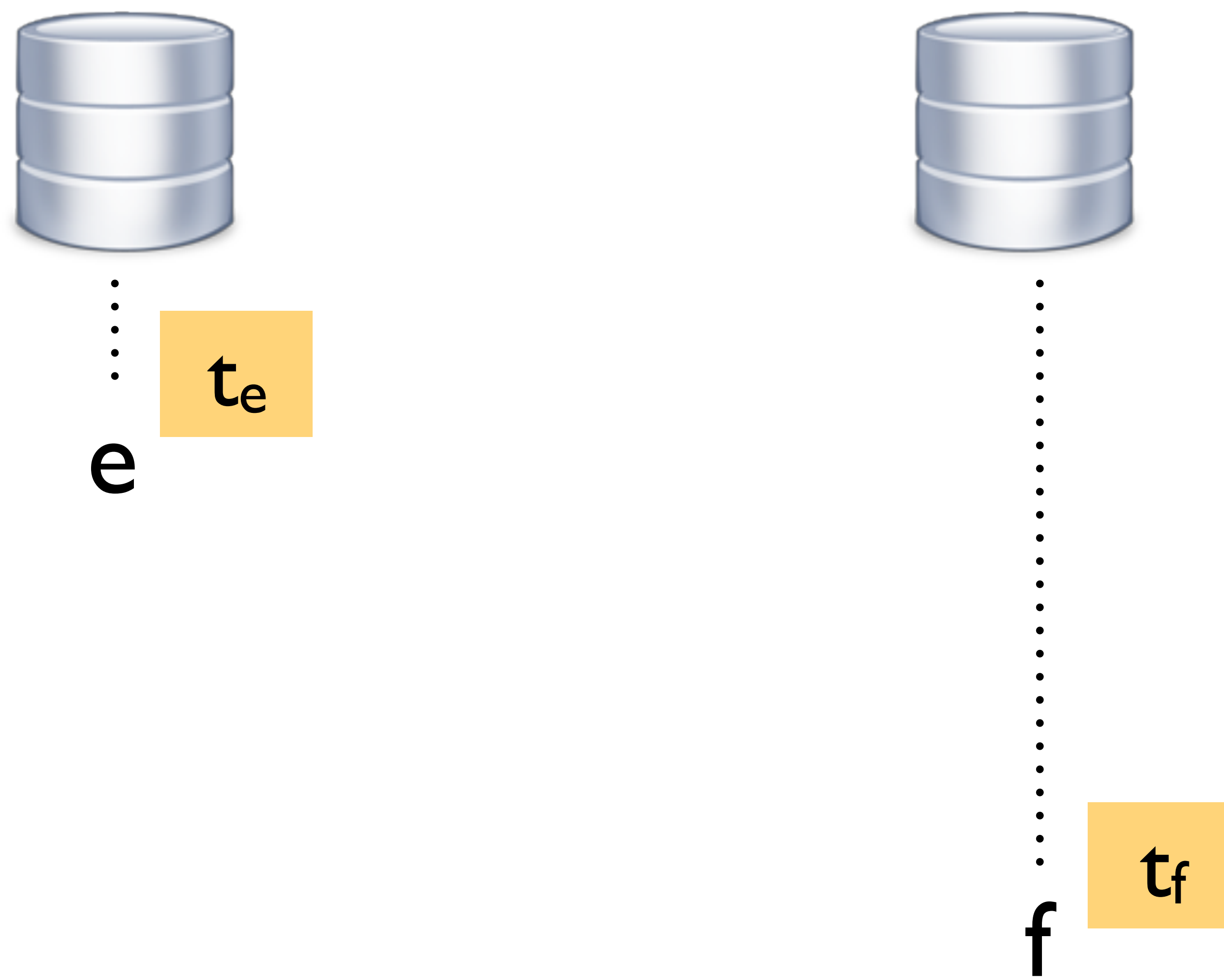
Constructing ar



Every event e gets assigned a timestamp t_e from a logical Lamport clock

$$e \xrightarrow{ar} f \iff t_e < t_f$$

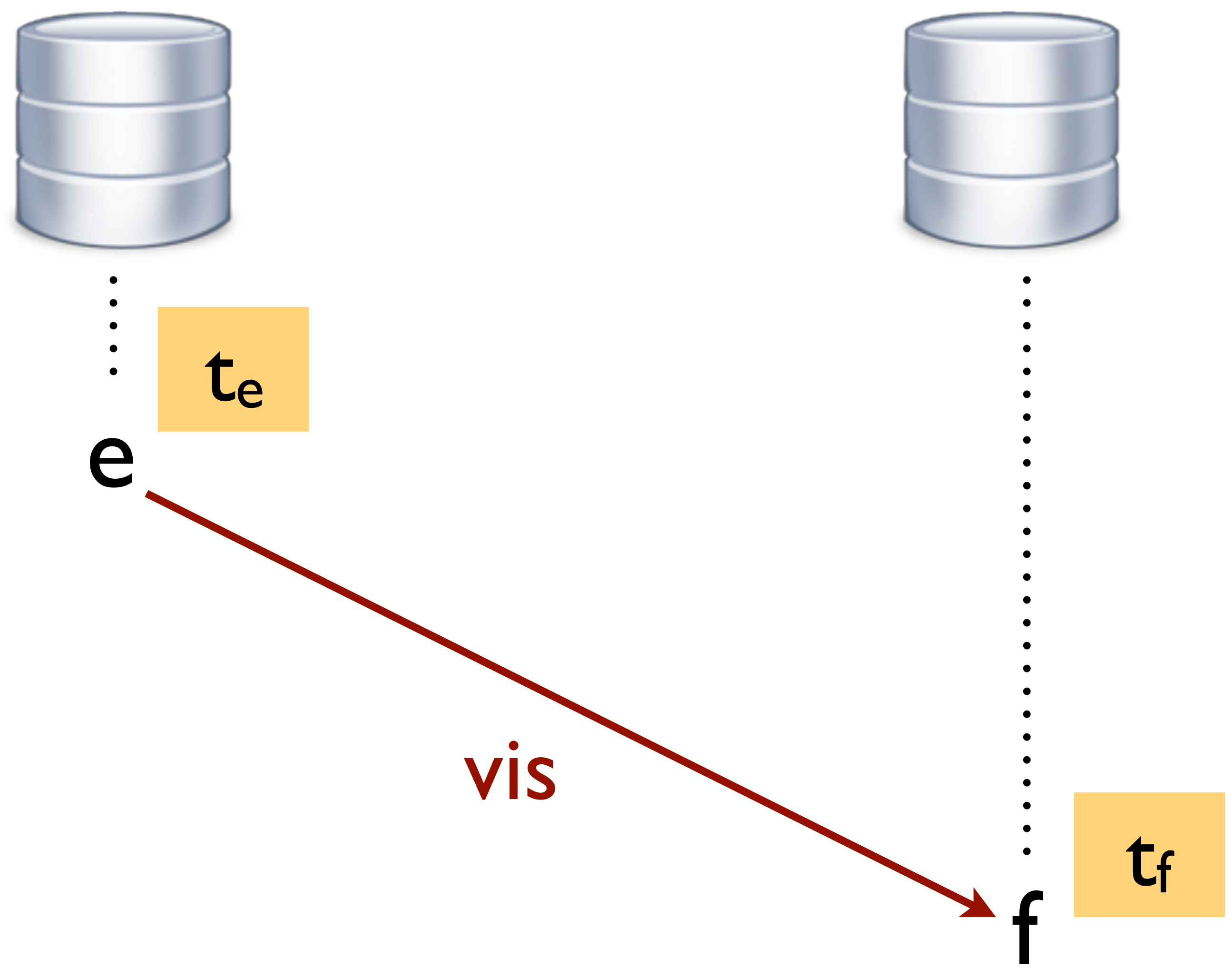
$\text{vis} \subseteq \text{ar}$



Every event e gets assigned a timestamp t_e from a logical Lamport clock

$$e \xrightarrow{\text{ar}} f \iff t_e < t_f$$

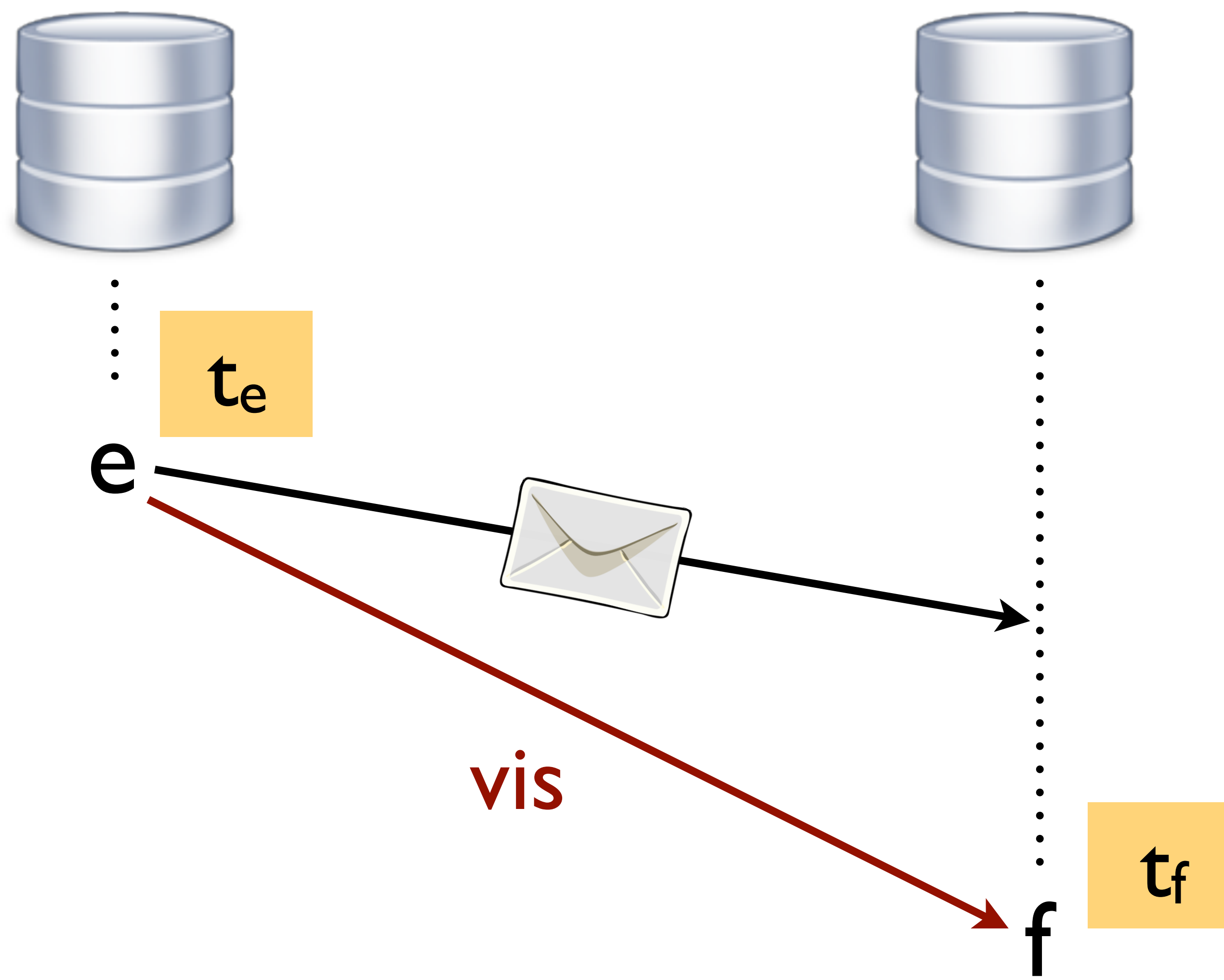
vis ⊆ ar



Every event e gets assigned a timestamp t_e from a logical Lamport clock

$$e \xrightarrow{ar} f \Longleftrightarrow t_e < t_f$$

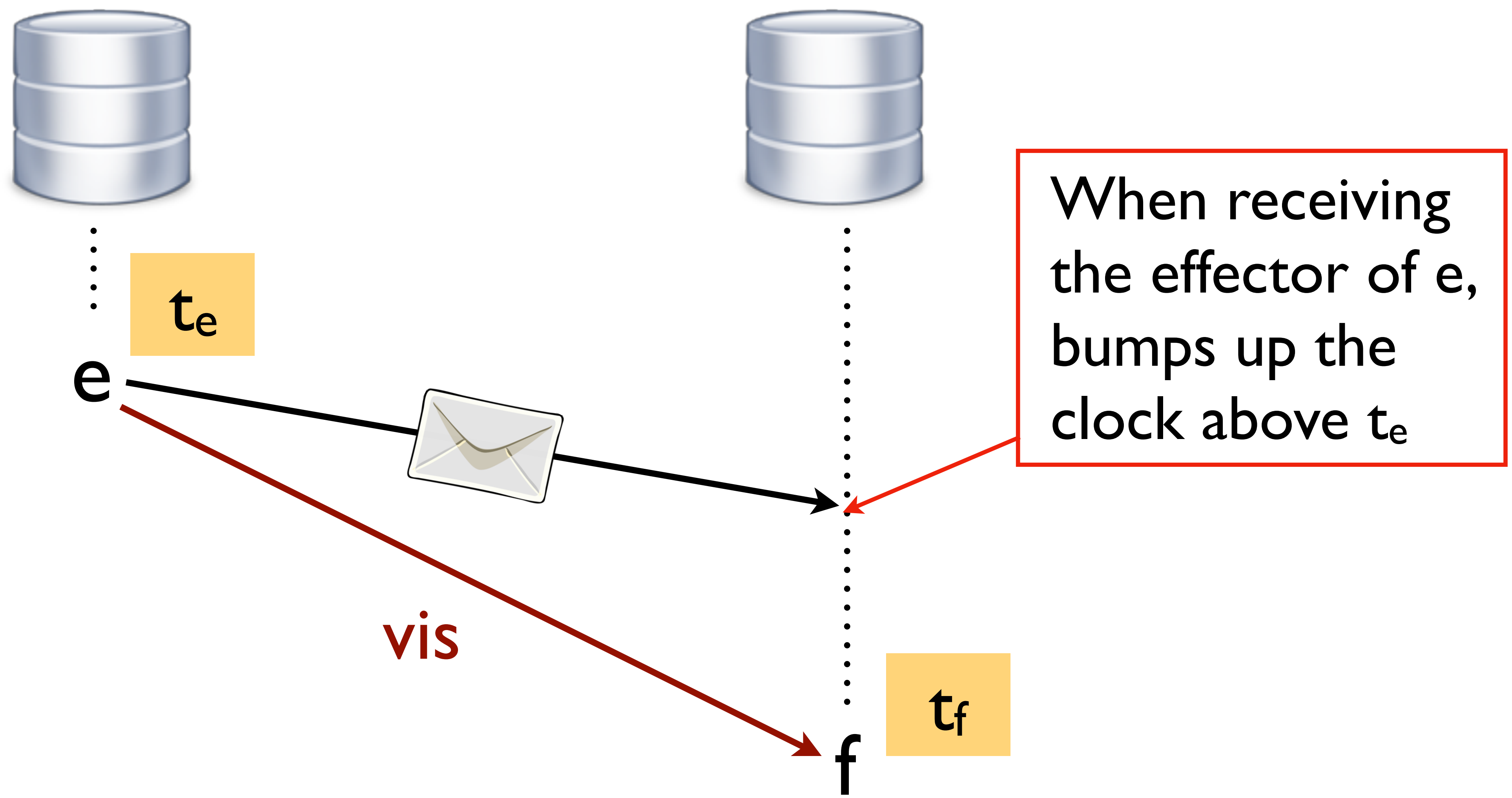
$vis \subseteq ar$



Every event e gets assigned a timestamp t_e from a logical Lamport clock

$$e \xrightarrow{ar} f \iff t_e < t_f$$

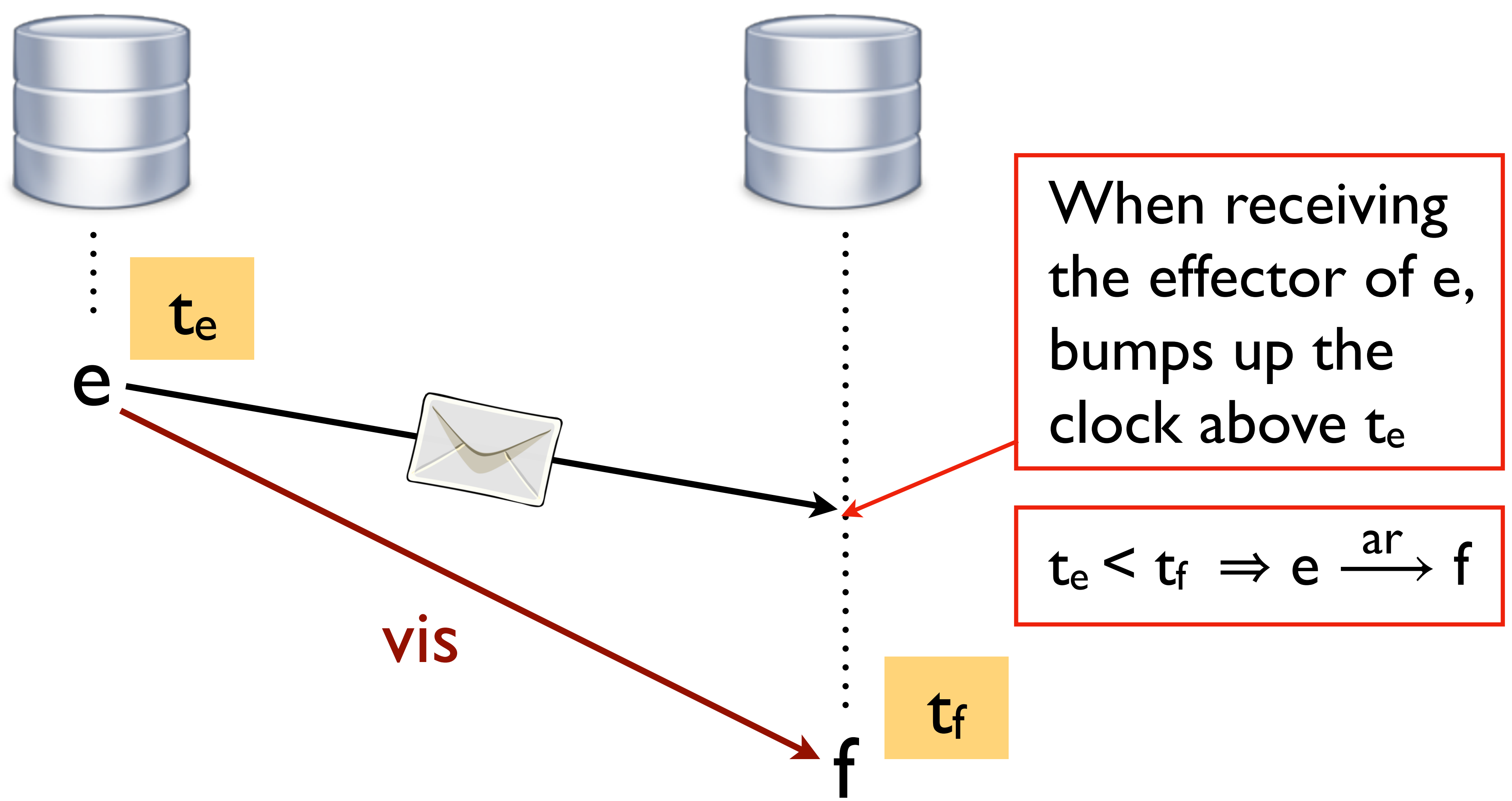
vis ⊆ ar



Every event e gets assigned a timestamp t_e from a logical Lamport clock

$$e \xrightarrow{ar} f \iff t_e < t_f$$

vis $\subseteq$ ar



Every event e gets assigned a timestamp t_e from a logical Lamport clock

$$e \xrightarrow{ar} f \iff t_e < t_f$$

Correctness of registers

$$\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$$

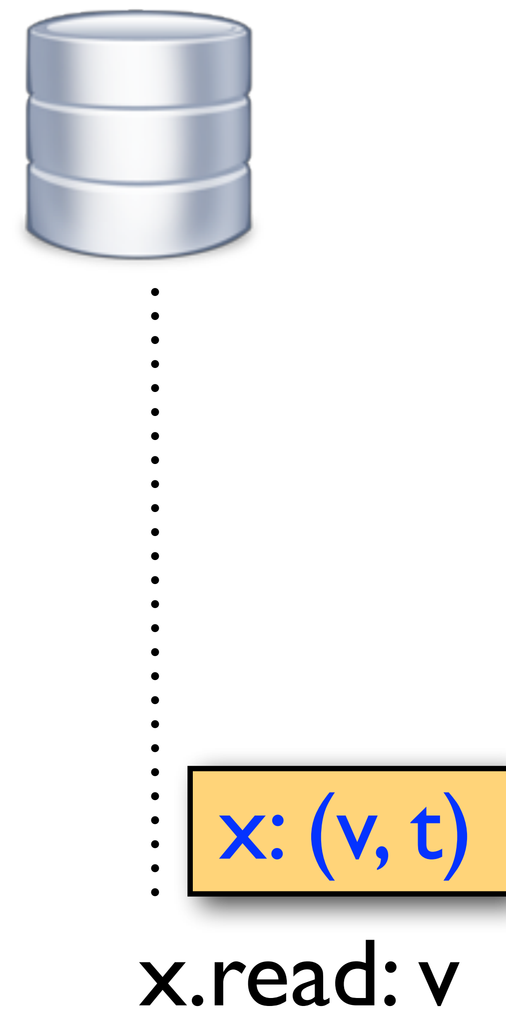
$$\begin{tikzcd}
\mathrm{x.write}(1) \arrow[rr, "\mathrm{ar}"] \arrow[dr, "\mathrm{vis}"'] & & \mathrm{x.write}(2) \arrow[dl, "\mathrm{vis}"] \\
& \mathrm{x.read}(): 2 &
\end{tikzcd}$$

F: reads return the last value in ar

Correctness of registers

x.read: ?

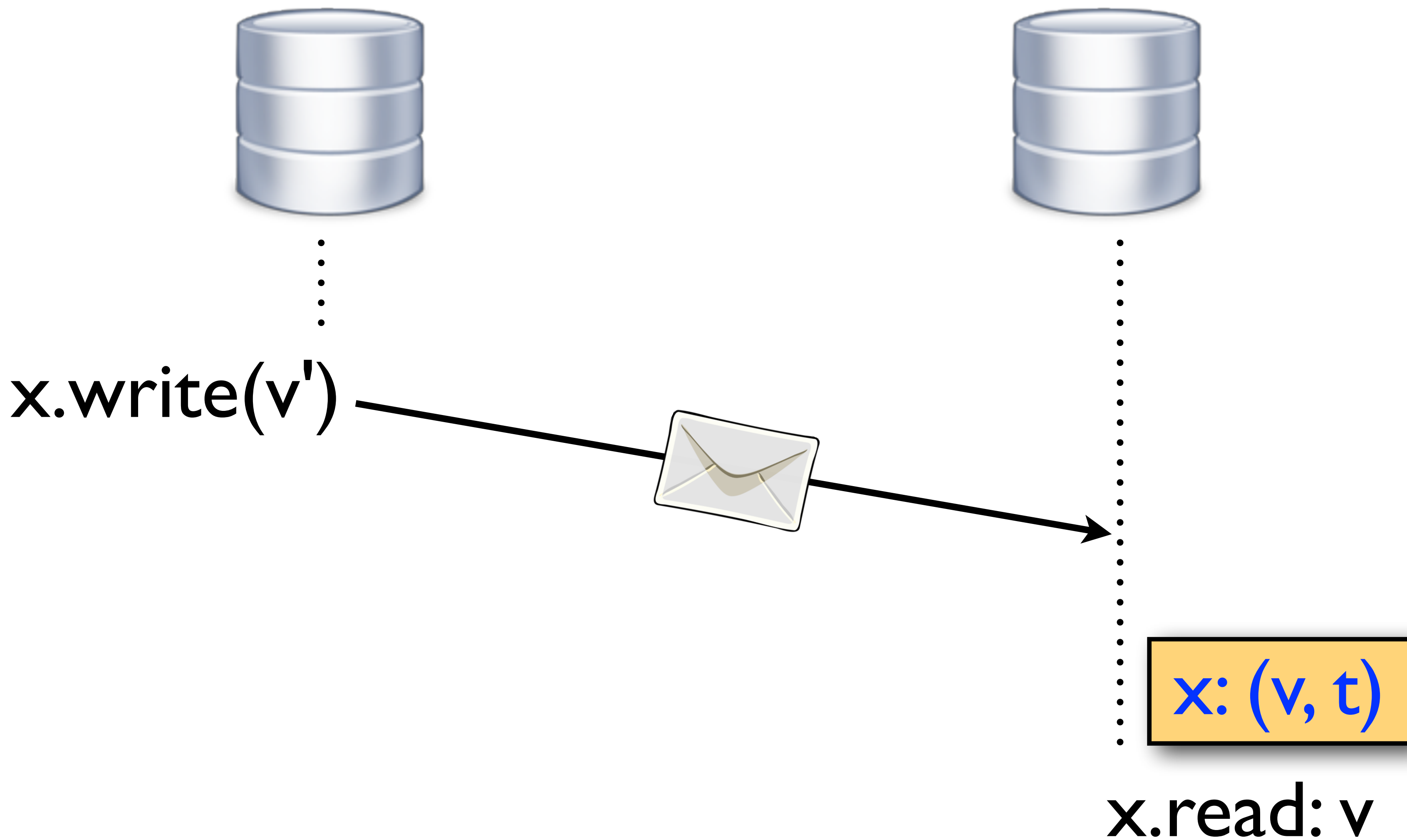
Correctness of registers



A read returns the value part of the register at the replica:

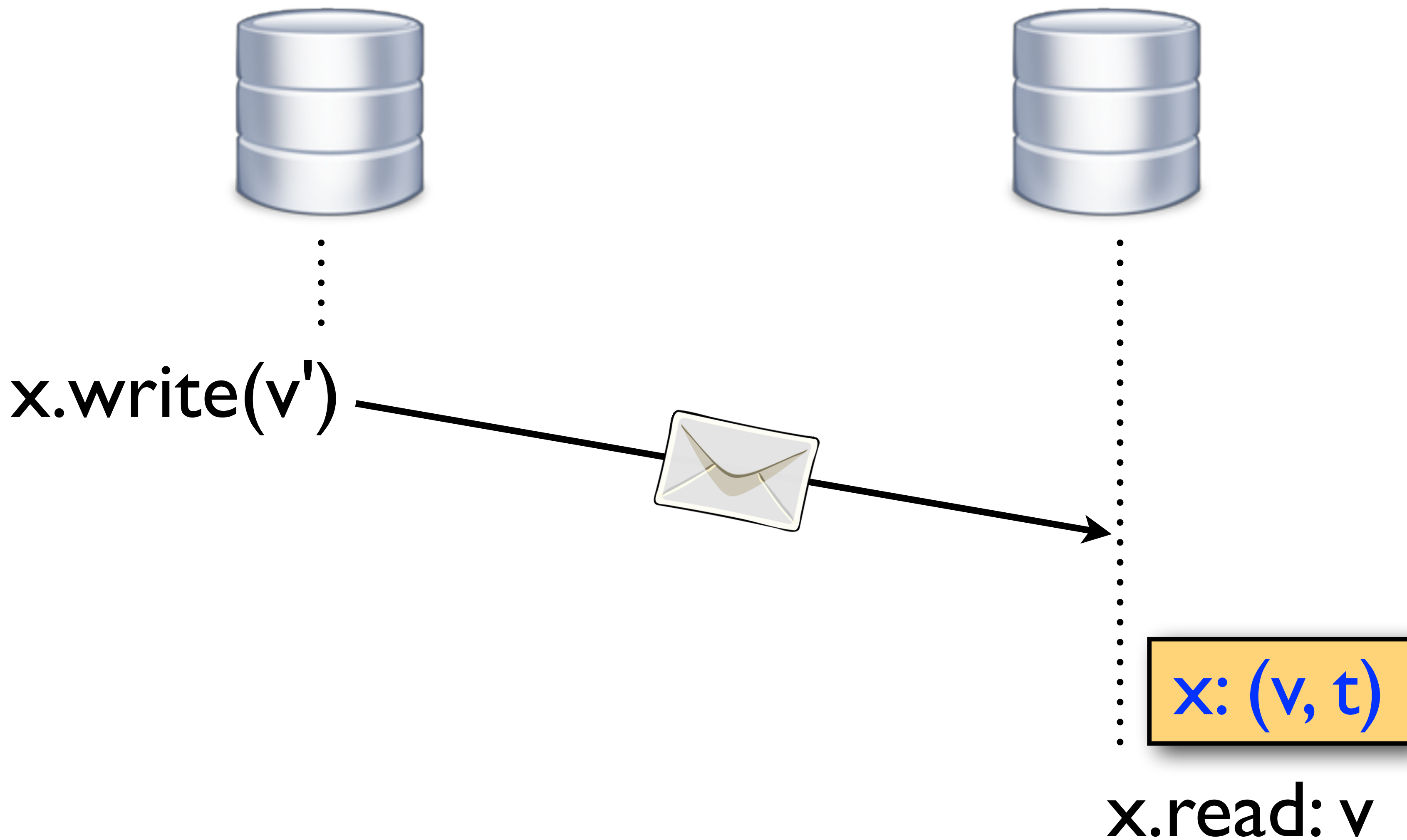
$[\![\text{read}()]\!]_{val}(v, t) \;=\; v$

Correctness of registers



Invariant: the value of a register at a replica is the one with the highest timestamp out of all delivered writes

Correctness of registers

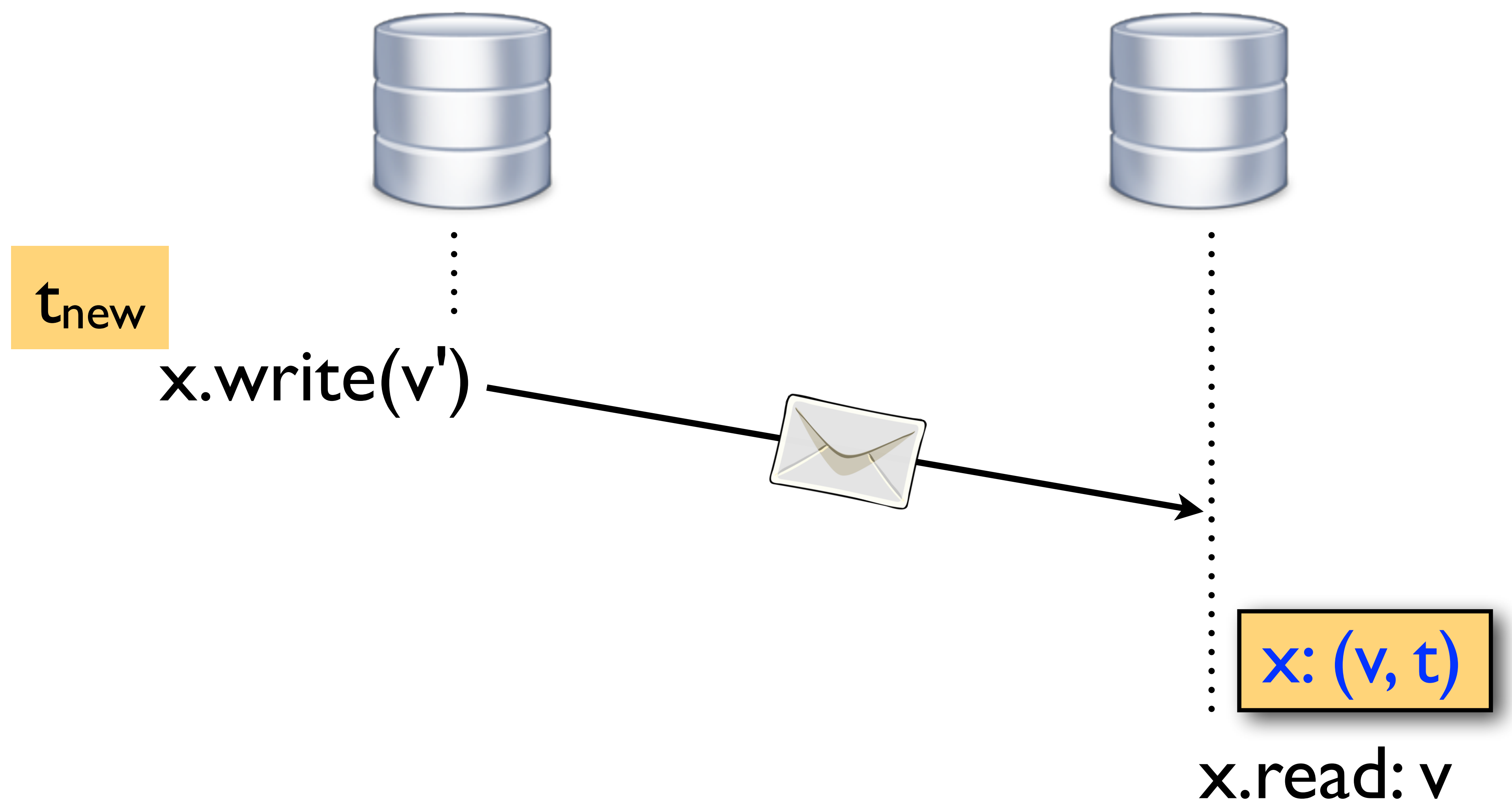


Invariant: the value of a register at a replica is the one with the highest timestamp out of all delivered writes

$[\![\text{write}(v_{new})]\!]_{eff}(v, t) \;=$ let (t_{new} = *newUniqueTS*()) in

$\lambda(v', t')$. if $t_{new} > t'$ then (v_{new}, t_{new}) else (v, t)

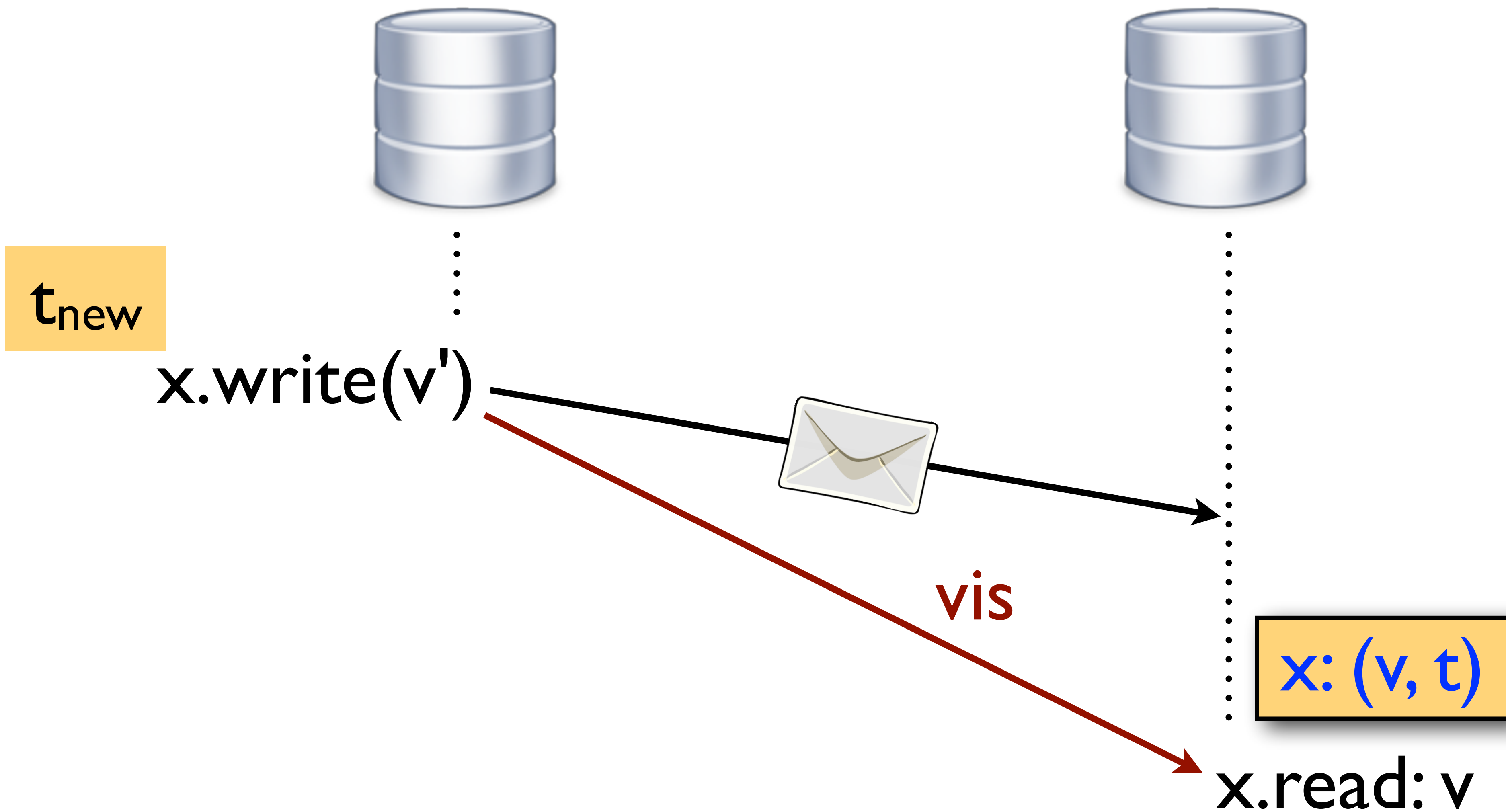
Correctness of registers



Invariant: the value of a register at a replica is the one with the highest timestamp out of all delivered writes

$[\![\mathrm{write}(v_{new})]\!]_{eff}(v, t) \ = \mathrm{let}\ (t_{new} = \mathit{newUniqueTS}())\ \mathrm{in}$
$\lambda(v', t').\ \mathrm{if}\ t_{new} > t'\ \mathrm{then}\ (v_{new}, t_{new})\ \mathrm{else}\ (v, t)$

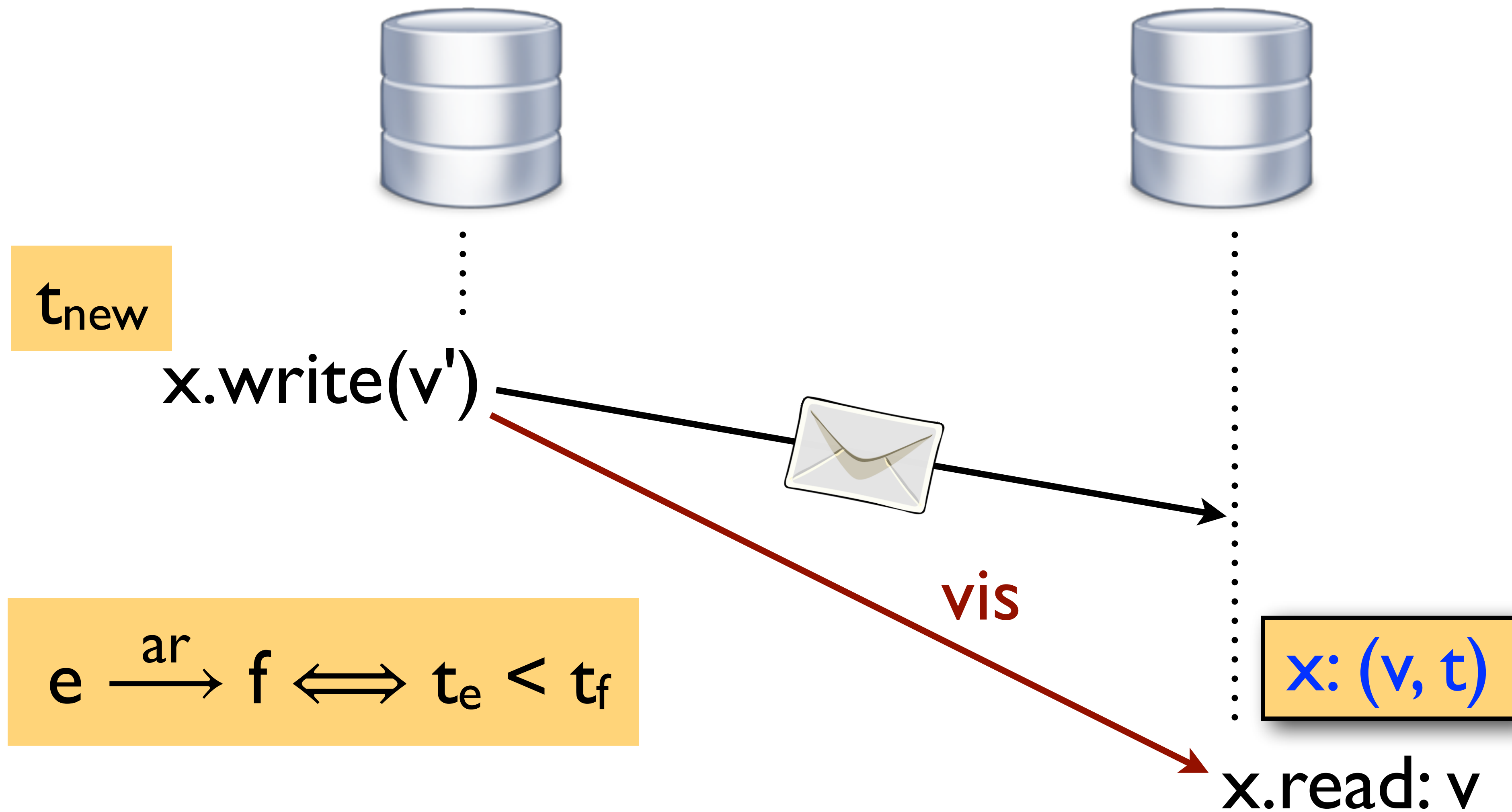
Correctness of registers



Invariant: the value of a register at a replica is the one with the highest timestamp out of all delivered writes

$[\![\text{write}(v_{new})]\!]_{\text{eff}}(v, t) = \text{let } (t_{new} = \mathit{newUniqueTS}()) \text{ in}$
$\lambda(v', t').\ \text{if } t_{new} > t' \text{ then } (v_{new}, t_{new}) \text{ else } (v, t)$

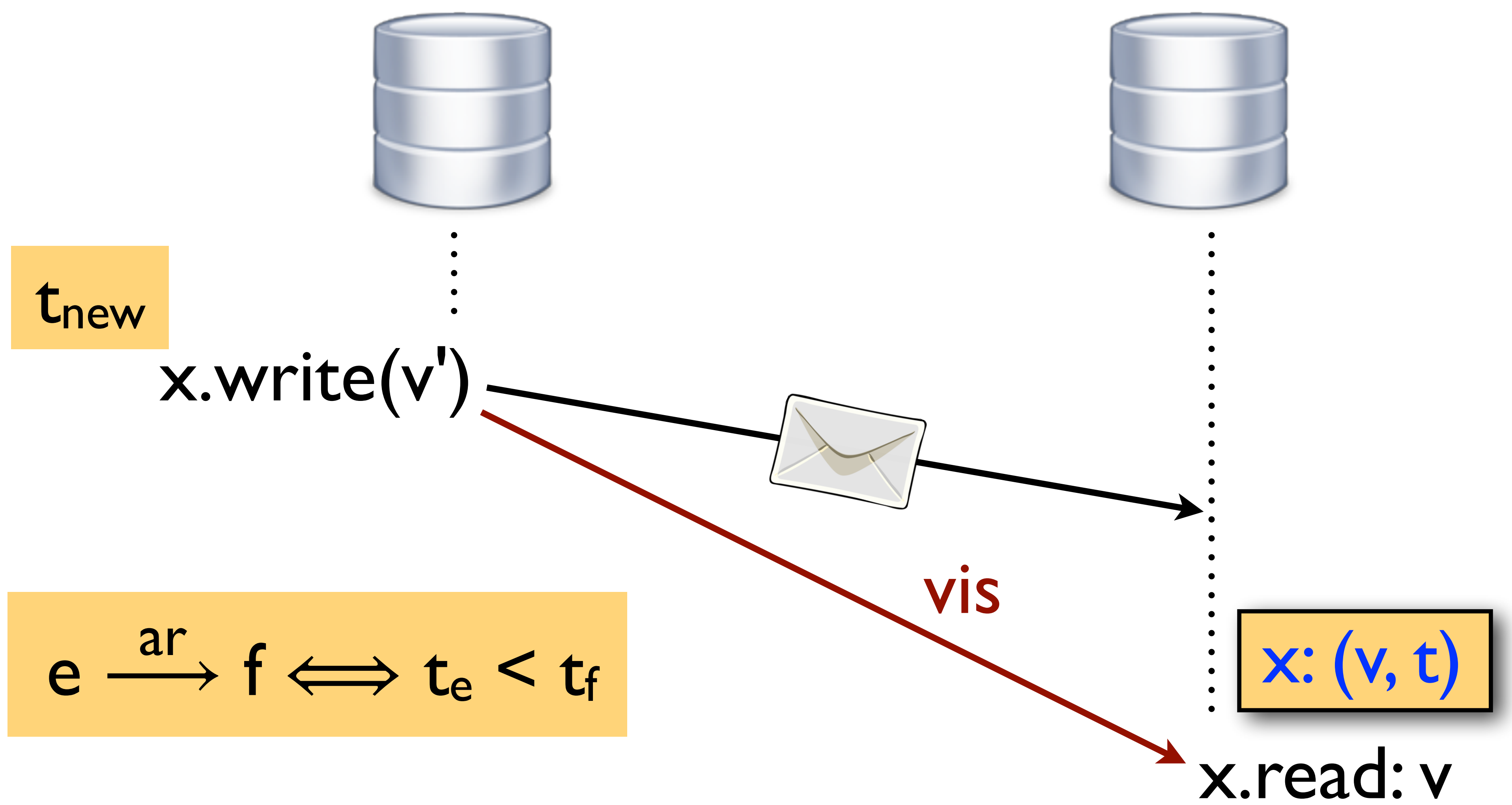
Correctness of registers



Invariant: the value of a register at a replica is the one with the highest timestamp out of all delivered writes

$[\![\text{write}(v_{new})]\!]_{\text{eff}}(v, t) = \text{let } (t_{new} = \textit{newUniqueTS}()) \text{ in}$
$\lambda(v', t').\ \text{if } t_{new} > t' \text{ then } (v_{new}, t_{new}) \text{ else } (v, t)$

Correctness of registers



Invariant: the value of a register at a replica is the one with the highest timestamp out of all delivered writes

= the last write in arbitration out of the ones visible to the read, QED.

Proof technique summary

- $\forall$ concrete execution of the implementation with a history (E, so)
- $\exists$ vis, ar. (E, so, vis, ar) satisfies the axioms $\mathscr{A}$

Proof technique summary

- $\forall$ concrete execution of the implementation with a history (E, so)
- $\exists$ vis, ar. (E, so, vis, ar) satisfies the axioms $\mathscr{A}$

- Construct vis from message deliveries and ar from timestamps

Proof technique summary

- $\forall$ concrete execution of the implementation with a history (E, so)
- $\exists$ vis, ar. (E, so, vis, ar) satisfies the axioms $\mathscr{A}$

- Construct vis from message deliveries and ar from timestamps
- Prove invariants relating replica state with message deliveries: *the value of a counter at a replica is the sum of all increments of the counter delivered to it*

Proof technique summary

- $\forall$ concrete execution of the implementation with a history (E, so)
- $\exists$ vis, ar. (E, so, vis, ar) satisfies the axioms $\mathscr{A}$

- Construct vis from message deliveries and ar from timestamps
- Prove invariants relating replica state with message deliveries: *the value of a counter at a replica is the sum of all increments of the counter delivered to it*
- Use the invariants to prove that return values of operations correspond to data type specs

In-between eventual and strong consistency

Eventual consistency summary

The set of histories (E, so) such that for some vis, ar:

- Return values consistent with data type specs:
  $\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$
- No causal cycles: so $\cup$ vis is acyclic
- Eventual visibility:
  $\forall e \in E.\ e \xrightarrow{\mathrm{vis}} f$ for all but finitely many $f \in E$

Eventual consistency summary

The set of histories (E, so) such that for some vis, ar:

- Return values consistent with data type specs:

  $\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$

- No causal cycles: so $\cup$ vis is acyclic

- Eventual visibility:

  $\forall e \in E.\ e \xrightarrow{\mathrm{vis}} f$ for all but finitely many $f \in E$

Stronger than quiescent consistency, but still weak

Strengthen consistency by adding additional axioms on vis and ar

Consistency zoo

Eventual consistency

Session guarantees

Causal consistency

Prefix consistency

Sequential consistency

Consistency zoo

Eventual consistency

Session guarantees

Causal consistency

Prefix consistency

Sequential consistency

Keep soundness justifications informal:
can be shown using previous techniques

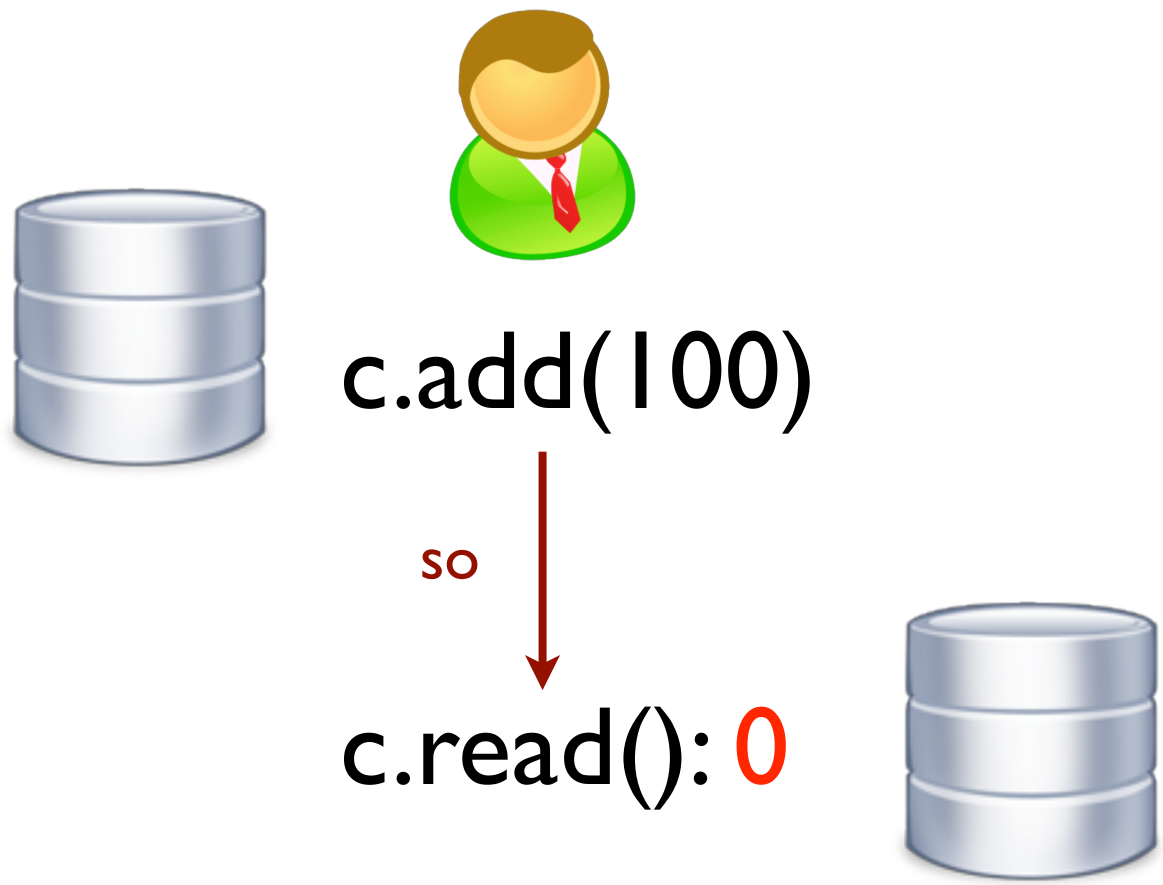
Read Your Writes

c.add(100)

$\Big\downarrow$ so

c.read(): 0

Read Your Writes



Read Your Writes

c.add(100)

$\downarrow$ so

c.read(): 0

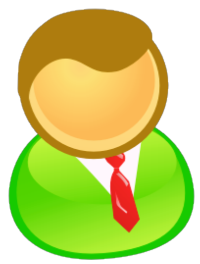
Read Your Writes

$$\begin{tikzcd} \text{c.add(100)} \arrow[d, shift right, "\text{so}"'] \arrow[d, shift left, "\text{vis}"] \\ \text{c.read(): 100} \end{tikzcd}$$

$\boxed{\text{so} \subseteq \text{vis}}$

- An operation sees all prior operations by the same process
- Session guarantees: clients only accumulate information

Read Your Writes



c.add(100)

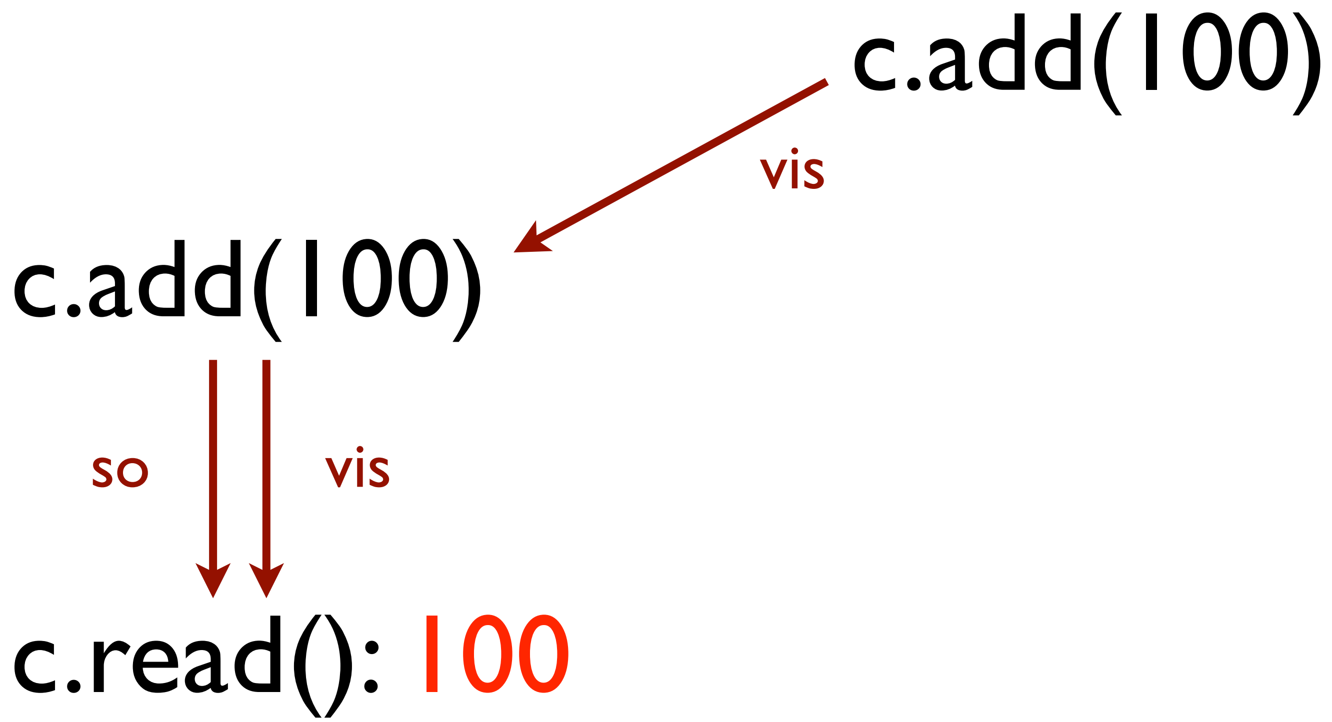
so vis

c.read(): 100

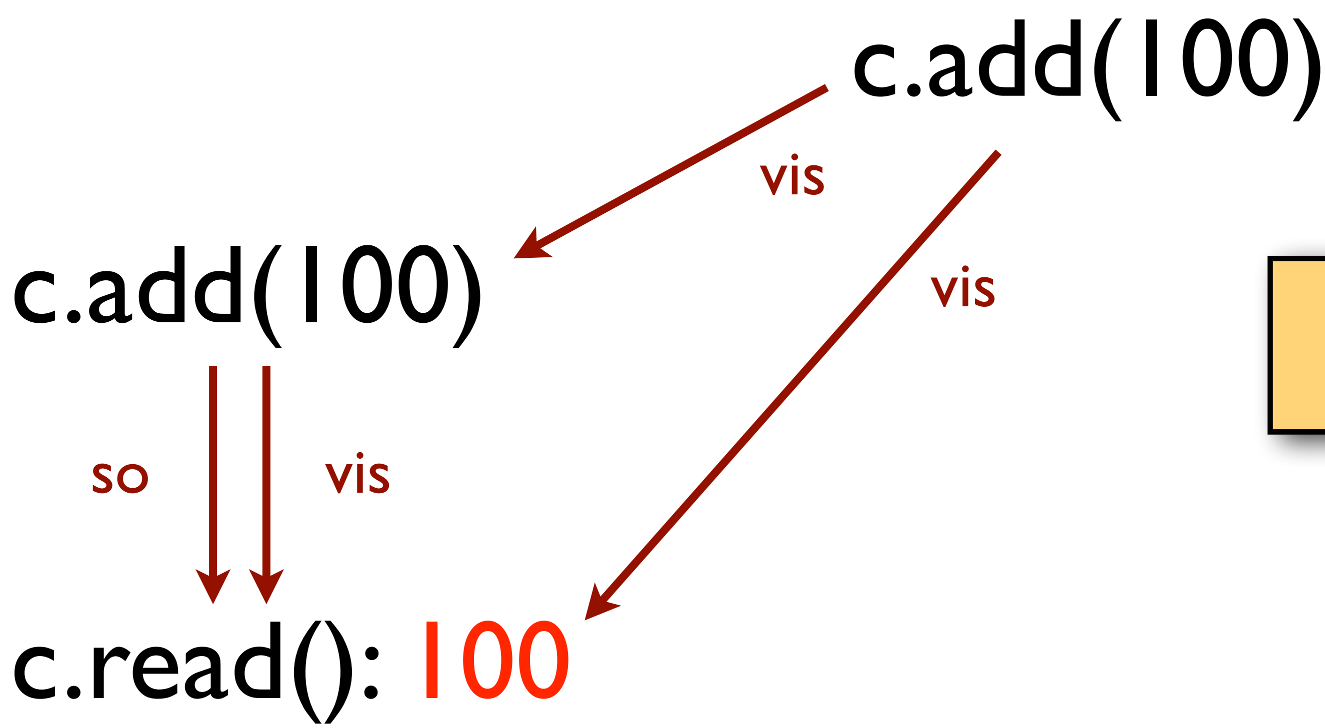
$\text{so} \subseteq \text{vis}$

- An operation sees all prior operations by the same process
- Session guarantees: clients only accumulate information
- Implementation: client sticks to the same replica

Monotonic Reads



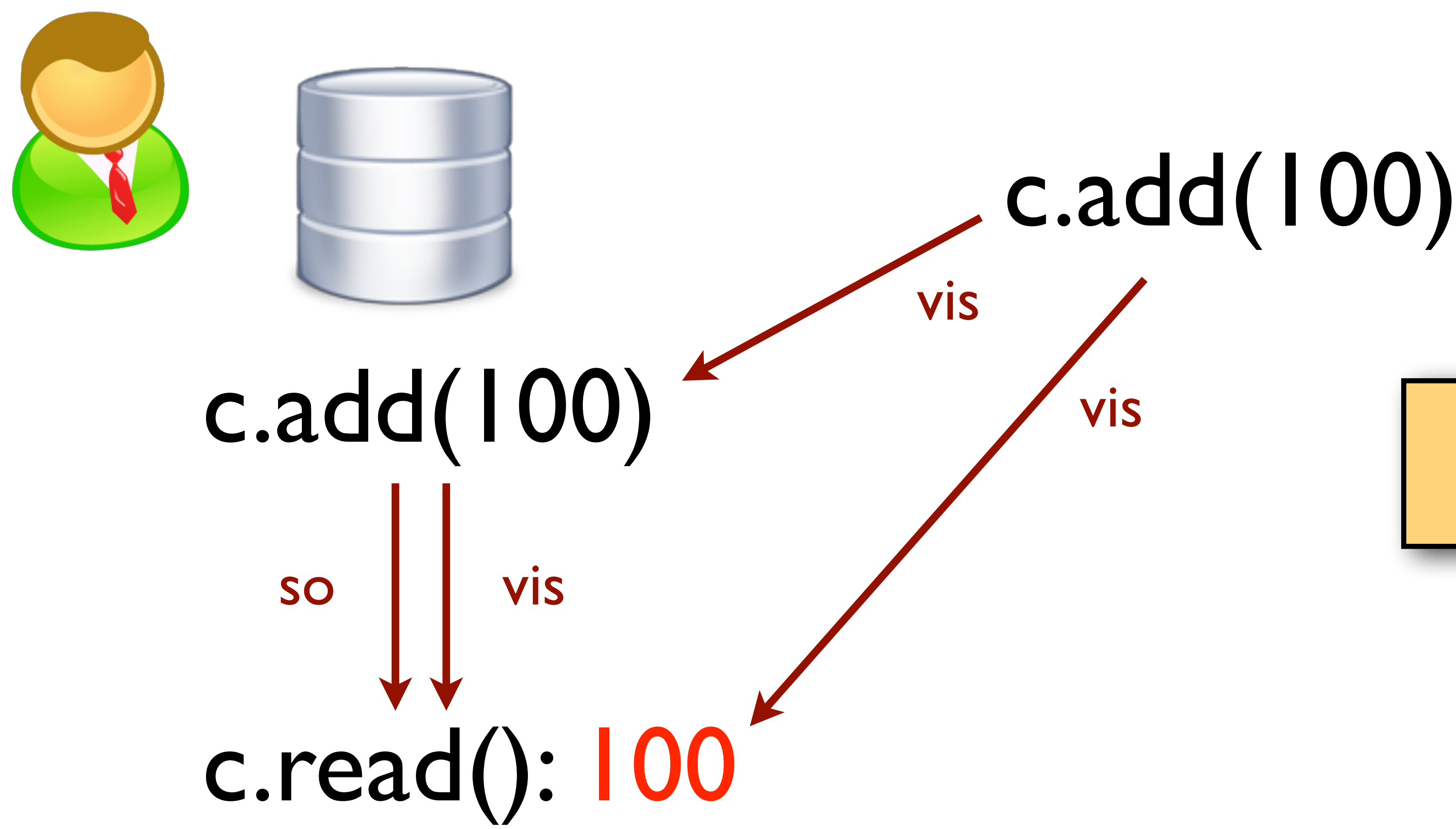
Monotonic Reads



$\boxed{\text{vis; so} \subseteq \text{vis}}$

- An operation sees what prior operations by the same session see

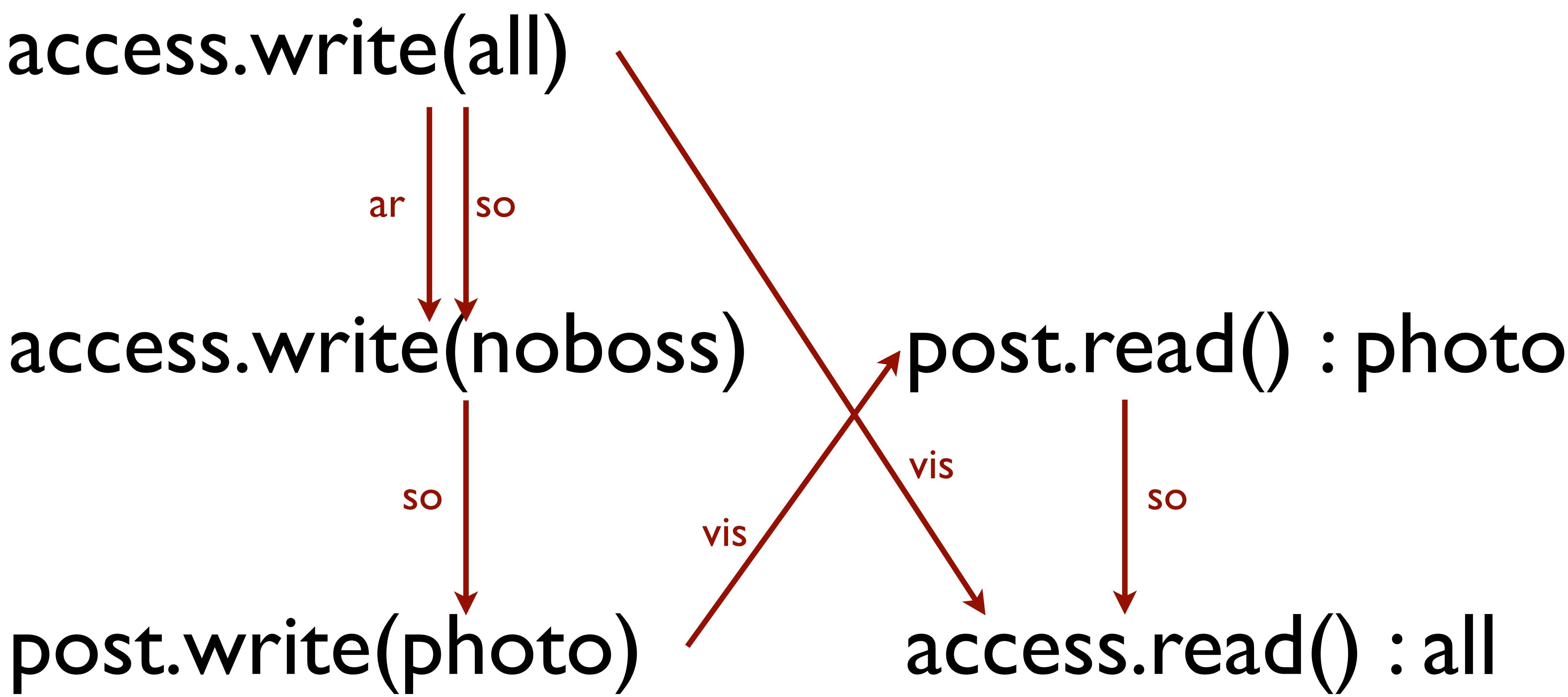
Monotonic Reads



$vis; so \subseteq vis$

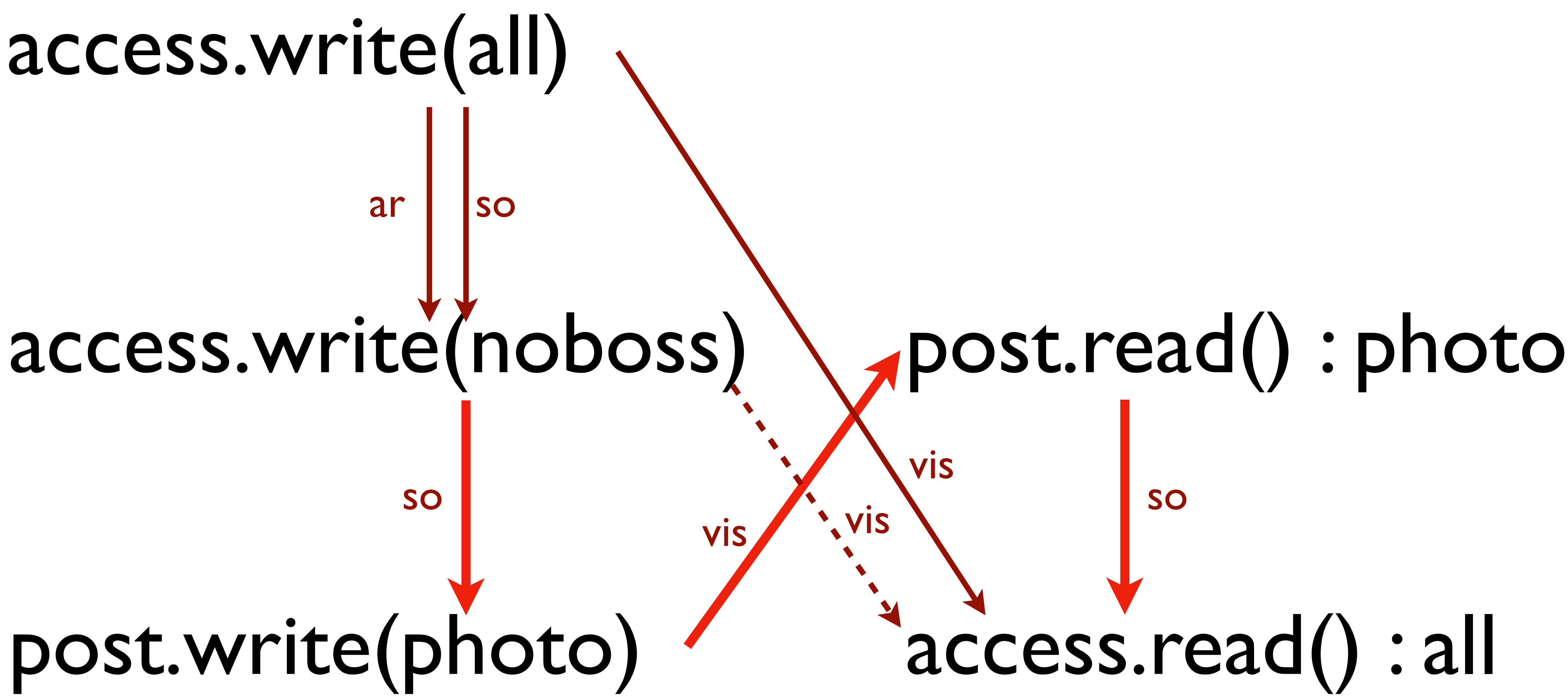
- An operation sees what prior operations by the same session see
- Implementation: client sticks to the same replica

Causal consistency



Disallows causality violation anomaly

Causal consistency

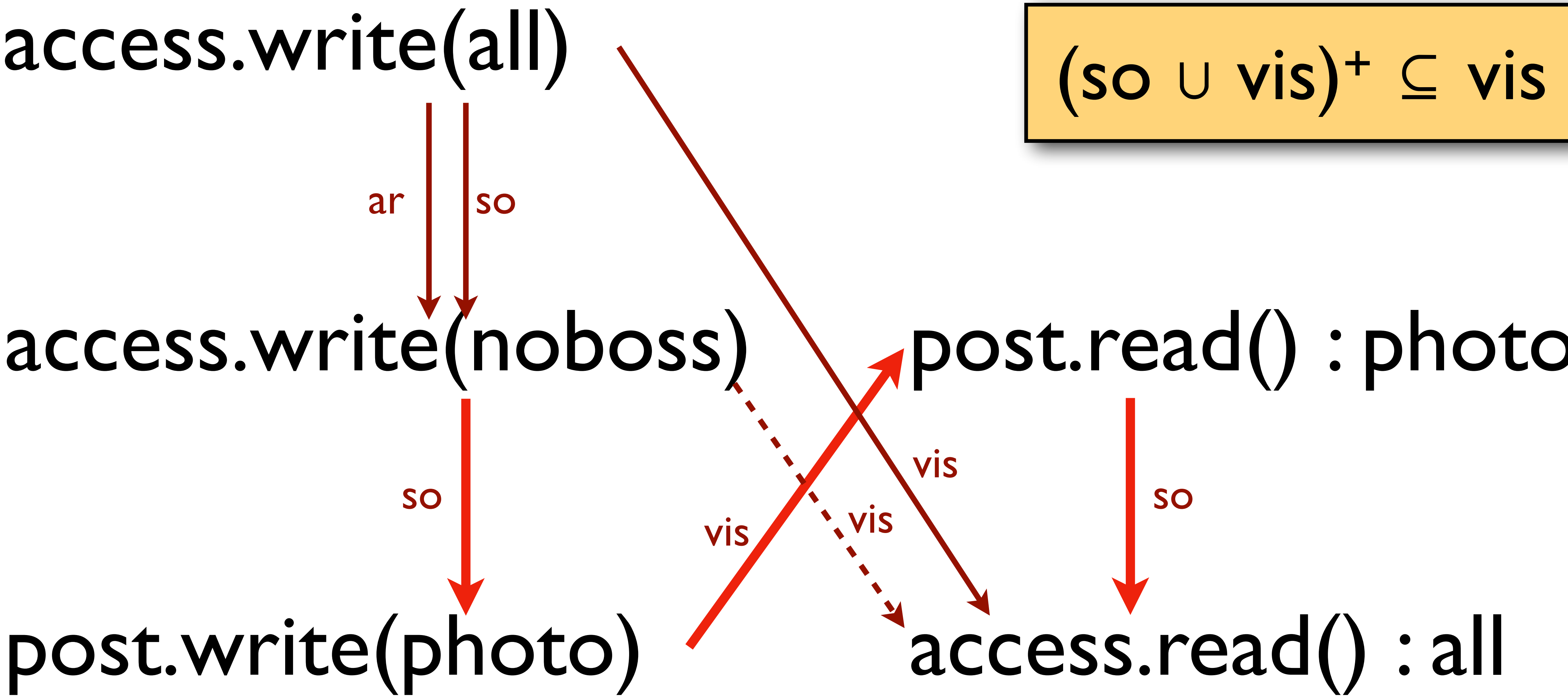


Unintuitive: chain of so and vis edges from write(noboss) to the read: write happened before the read

Mandate that all actions that happened before an action be visible to it

Causal consistency

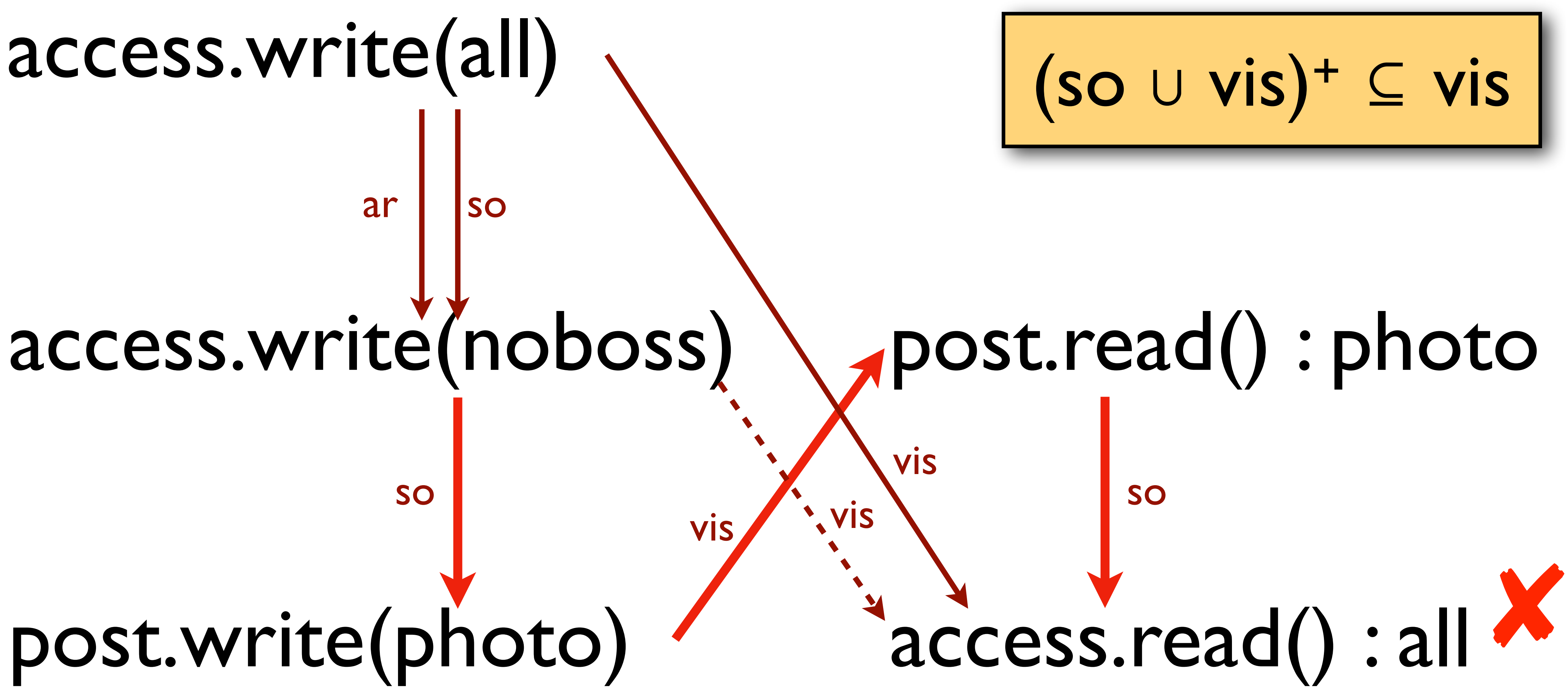
$$(so \cup vis)^+ \subseteq vis$$



Unintuitive: chain of so and vis edges from write(noboss) to the read: write happened before the read

Mandate that all actions that happened before an action be visible to it

Causal consistency

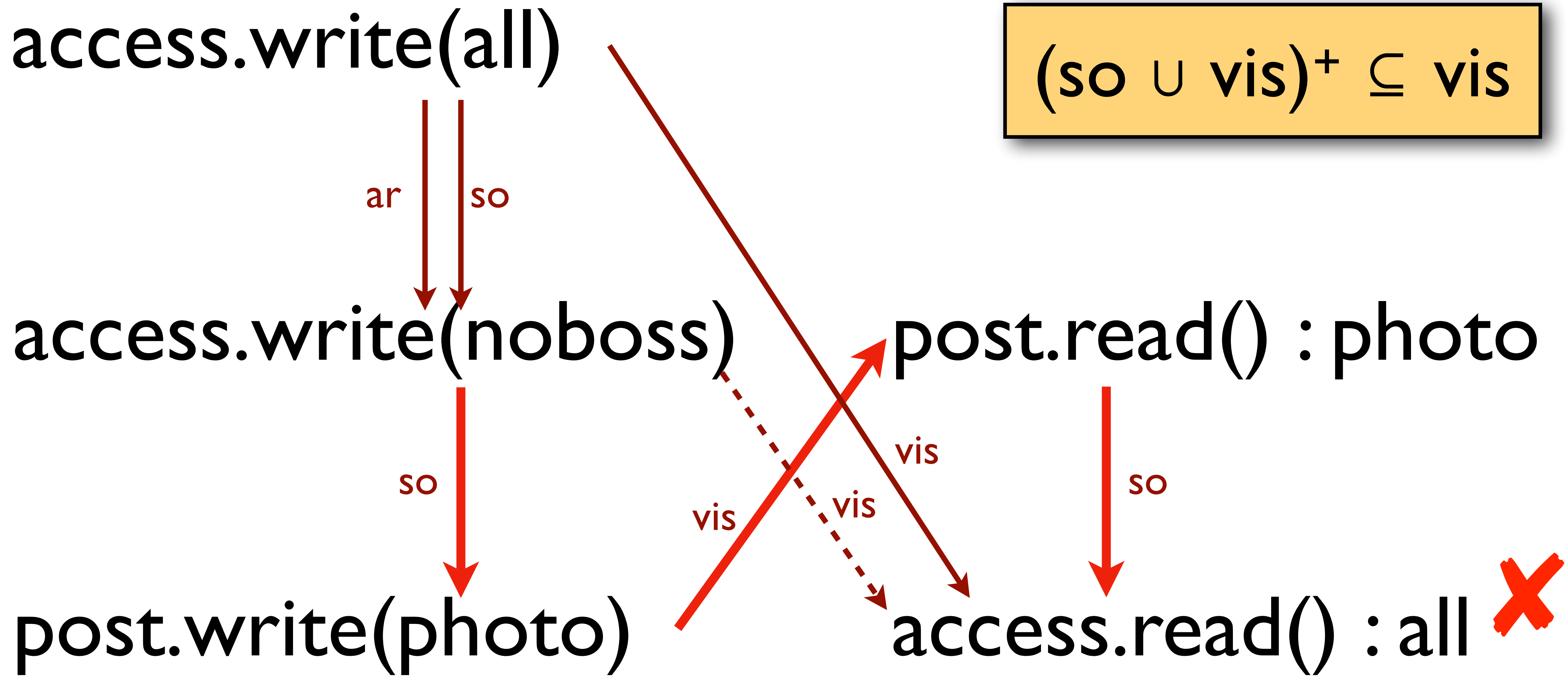


Unintuitive: chain of so and vis edges from write(noboss) to the read: write happened before the read

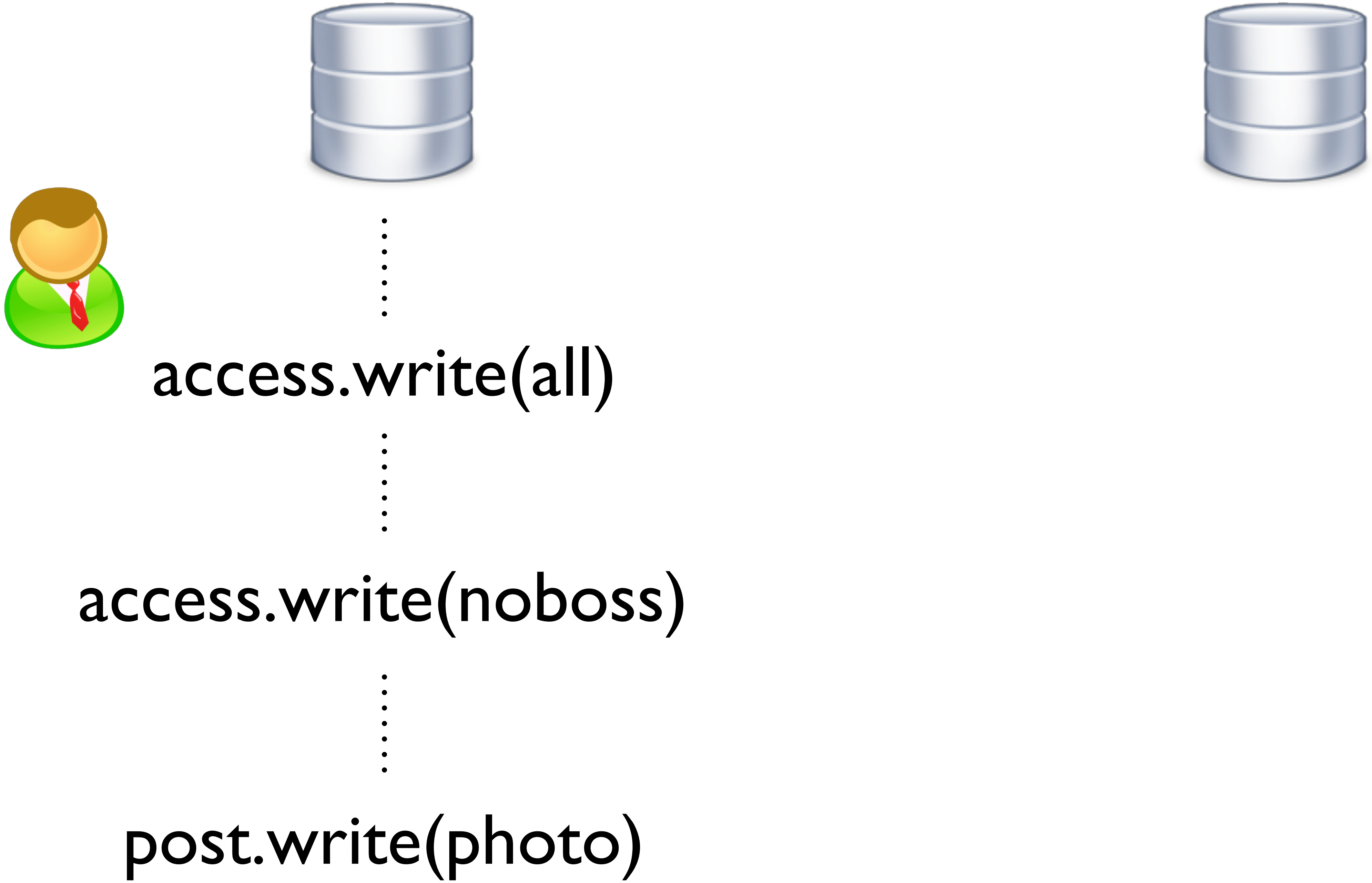
Mandate that all actions that happened before an action be visible to it

Causal consistency

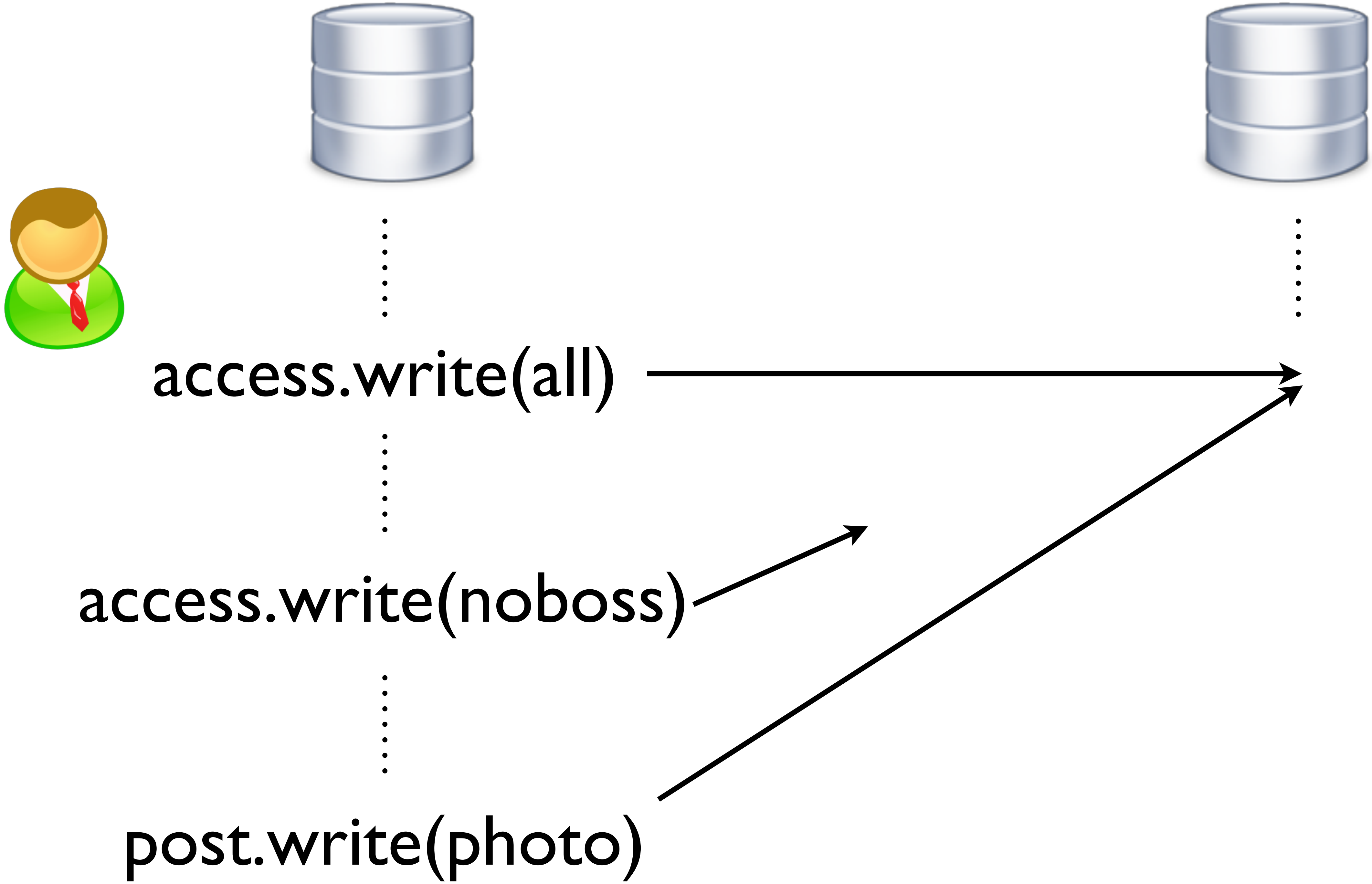
$$(so \cup vis)^+ \subseteq vis$$



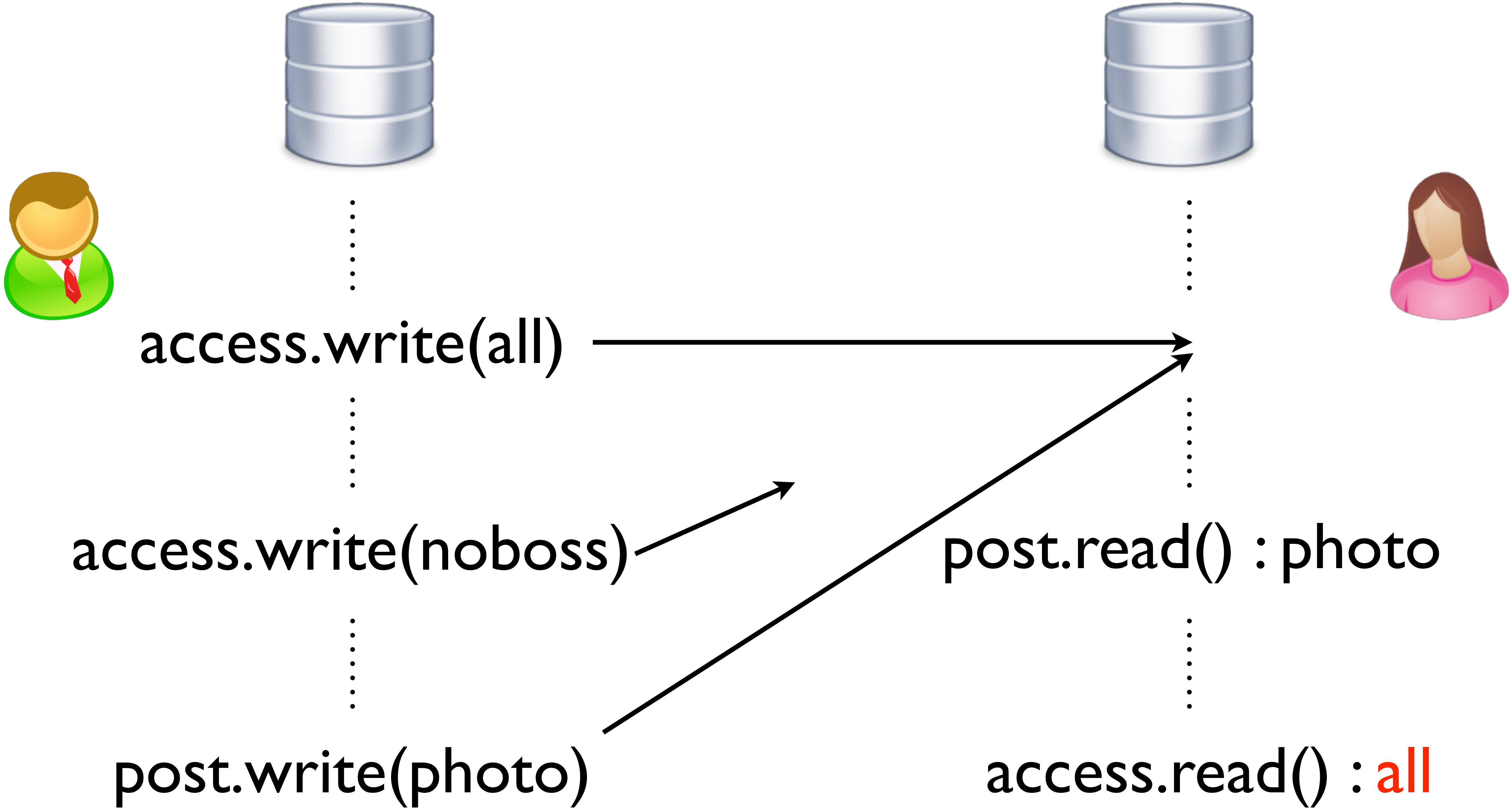
Implies session guarantees: $so \subseteq vis$ and $vis; so \subseteq vis$



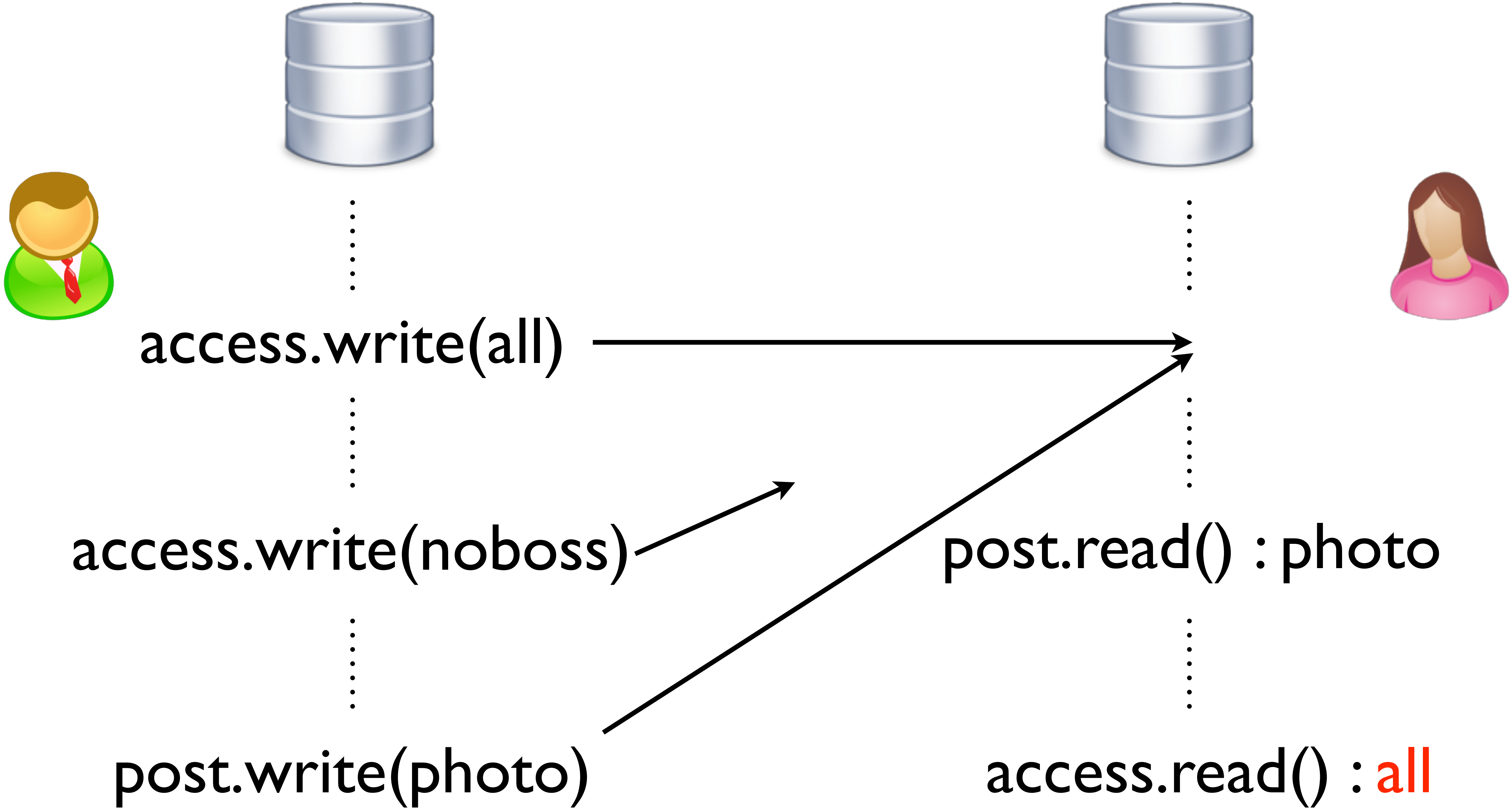
Clients stick to the same replica



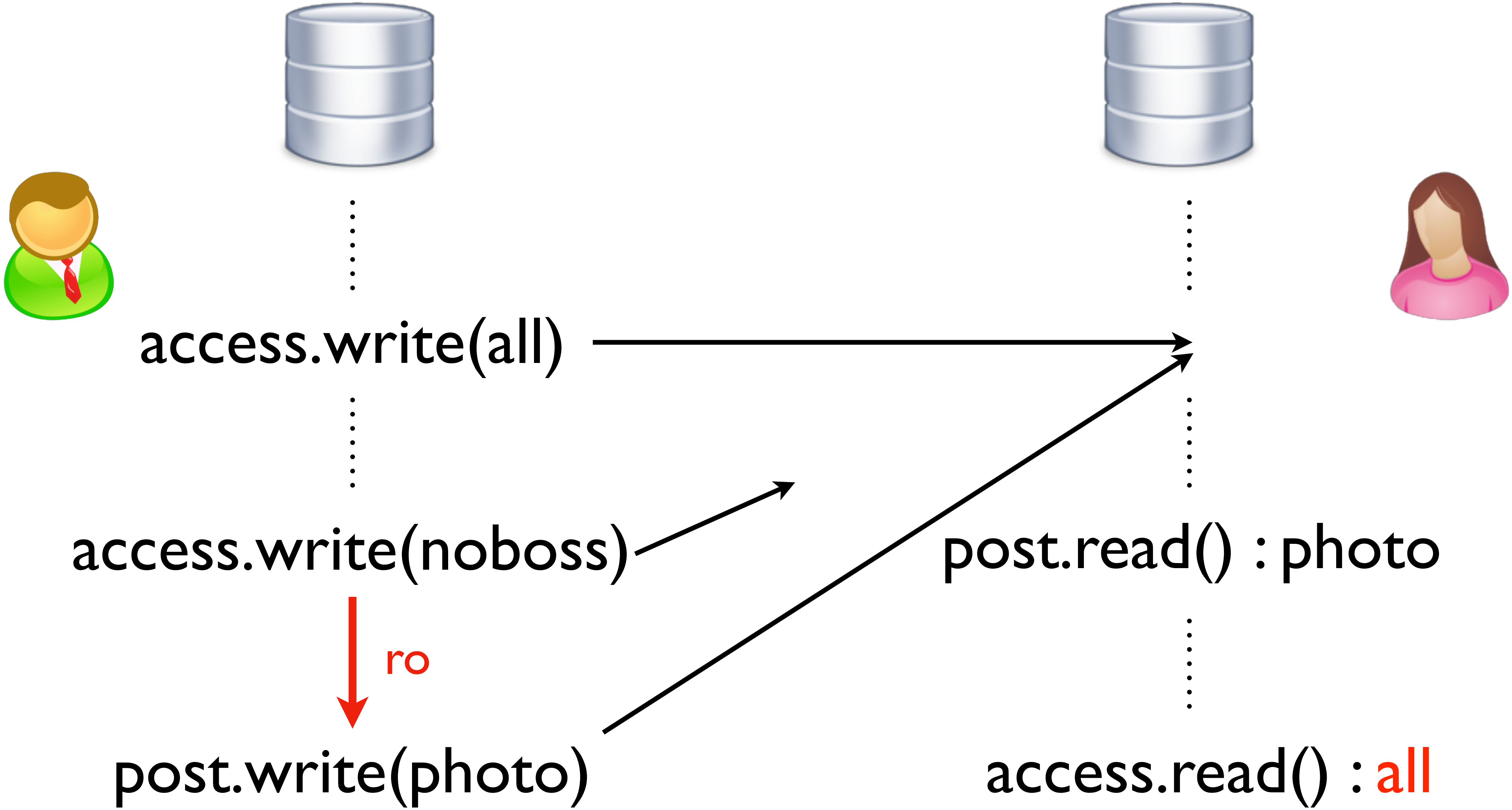
Clients stick to the same replica



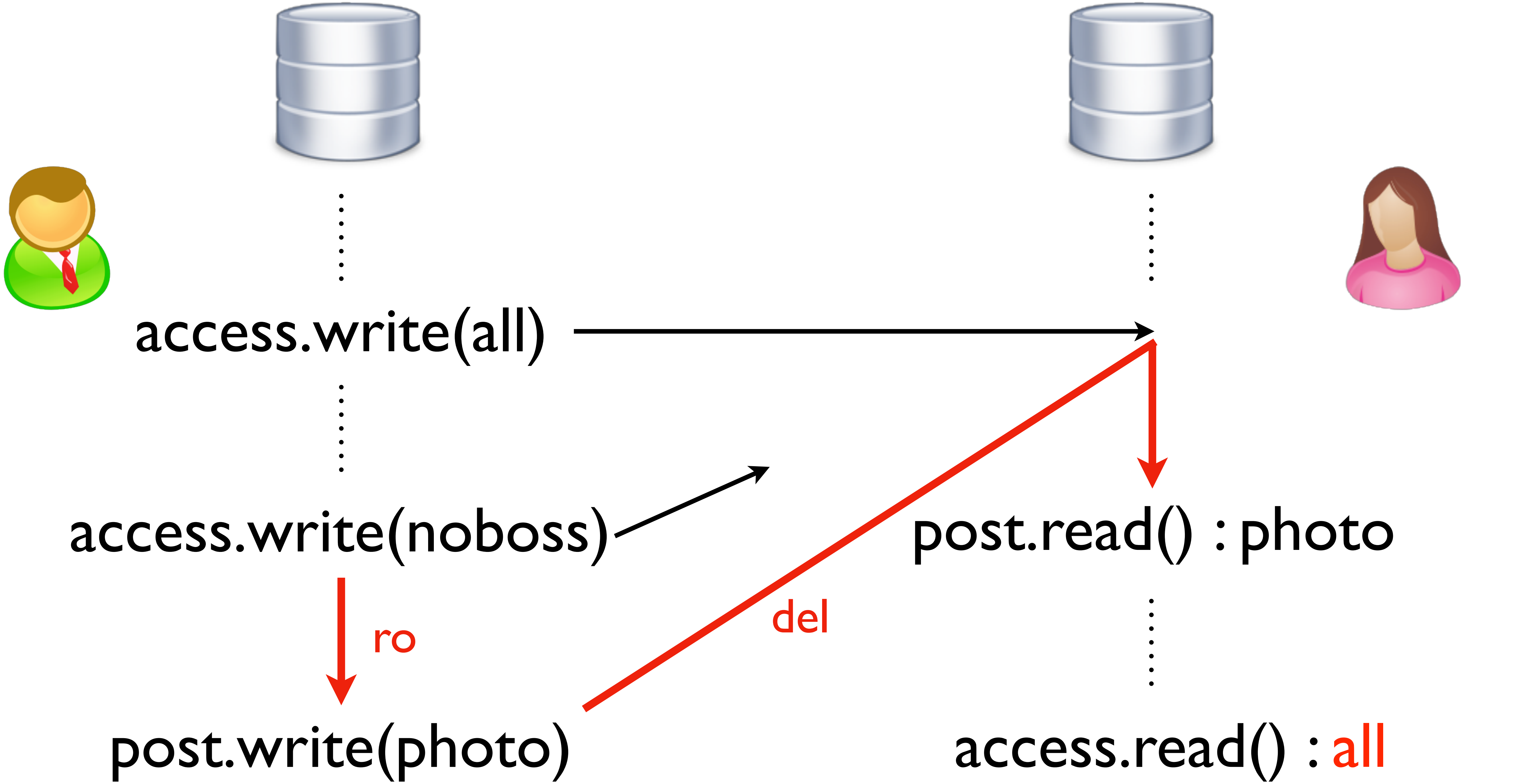
Clients stick to the same replica



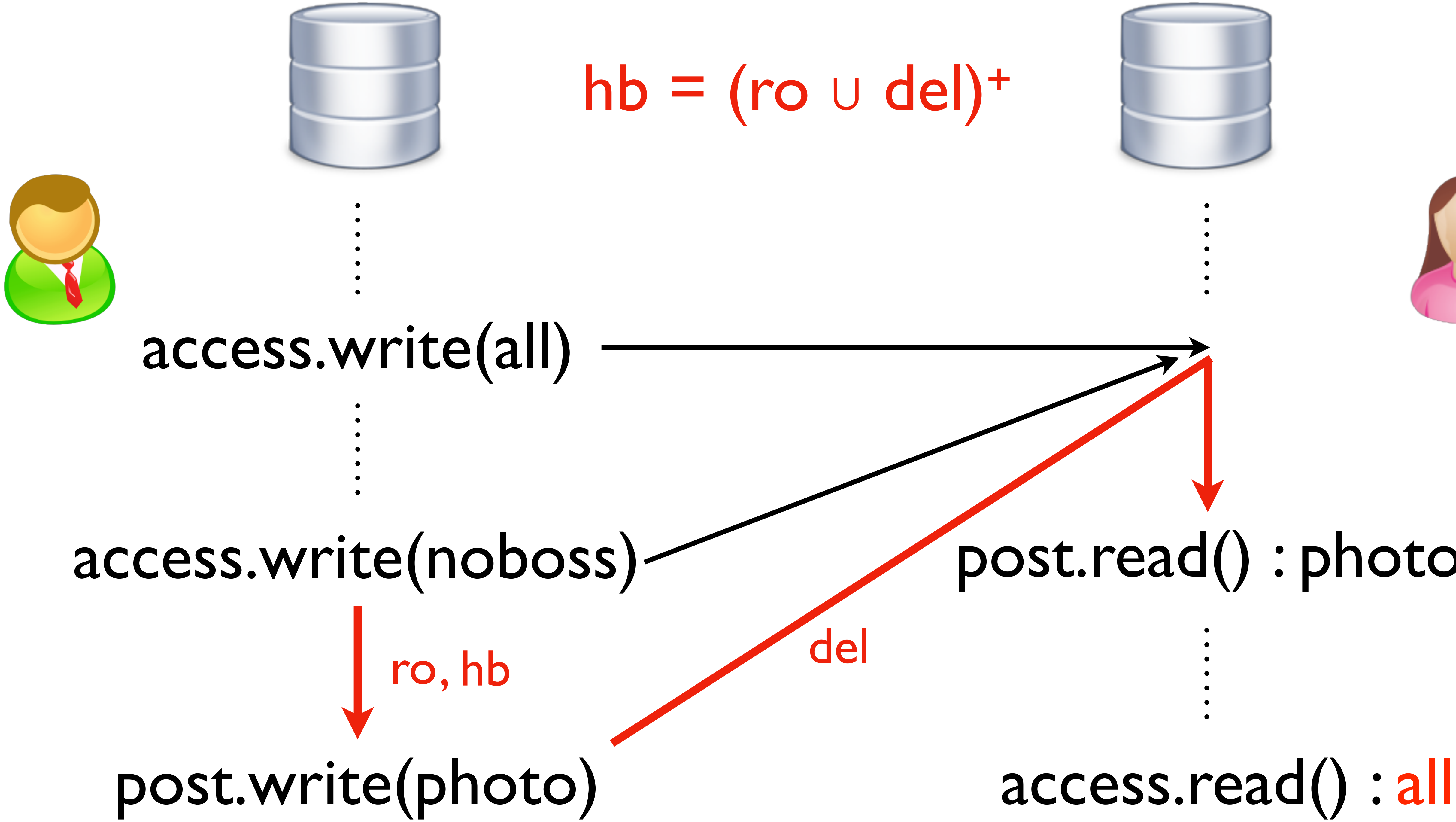
Cannot deliver an operation before delivering its causal dependencies



Replica order ro: the order in which operations are issued at a replica



Delivery order del: one operation got delivered before another was issued

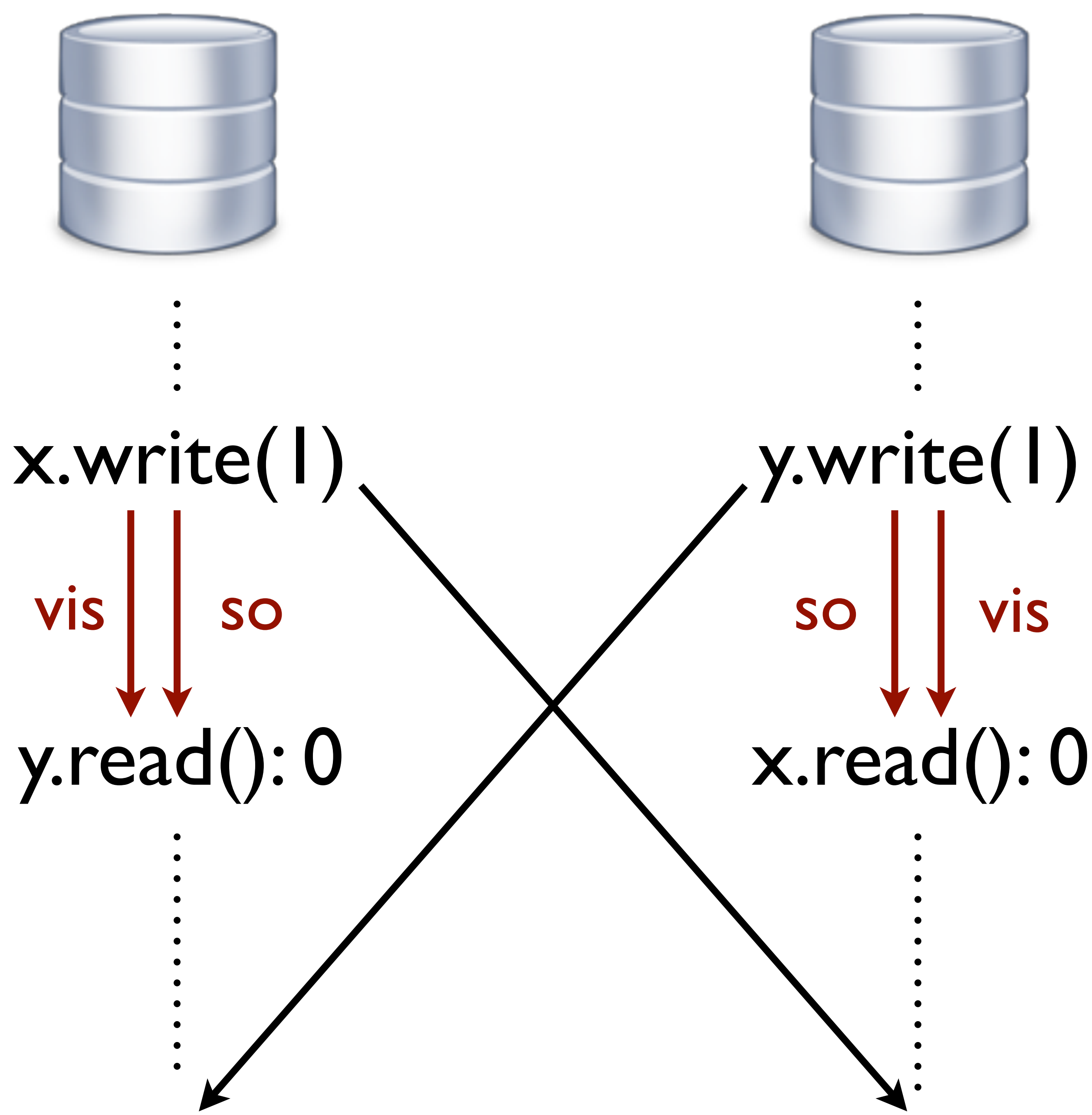


- Causal dependencies of e: $hb^{-1}(e)$
- An op can only be delivered after all its causal dependencies
- Implementations summarise dependencies concisely

Dekker example

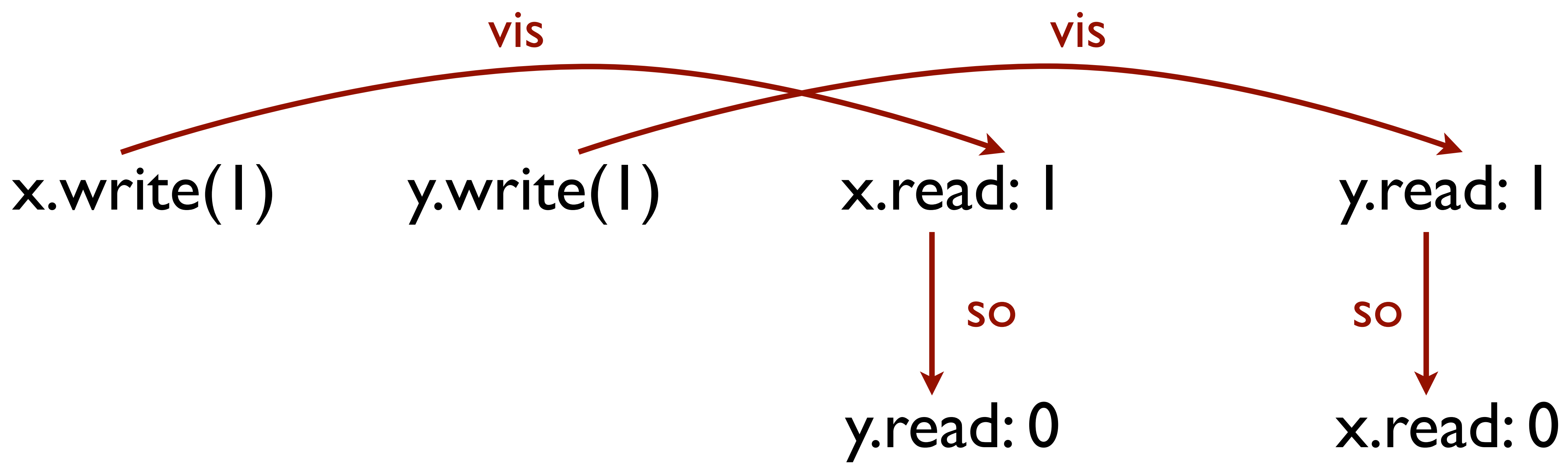
$$
\begin{tikzcd}
x.\mathrm{write}(1) \arrow[d, shift right, "\mathrm{vis}"'] \arrow[d, shift left, "\mathrm{so}"] & & y.\mathrm{write}(1) \arrow[d, shift right, "\mathrm{so}"'] \arrow[d, shift left, "\mathrm{vis}"] \\
y.\mathrm{read}(): 0 & & x.\mathrm{read}(): 0
\end{tikzcd}
$$

Dekker example

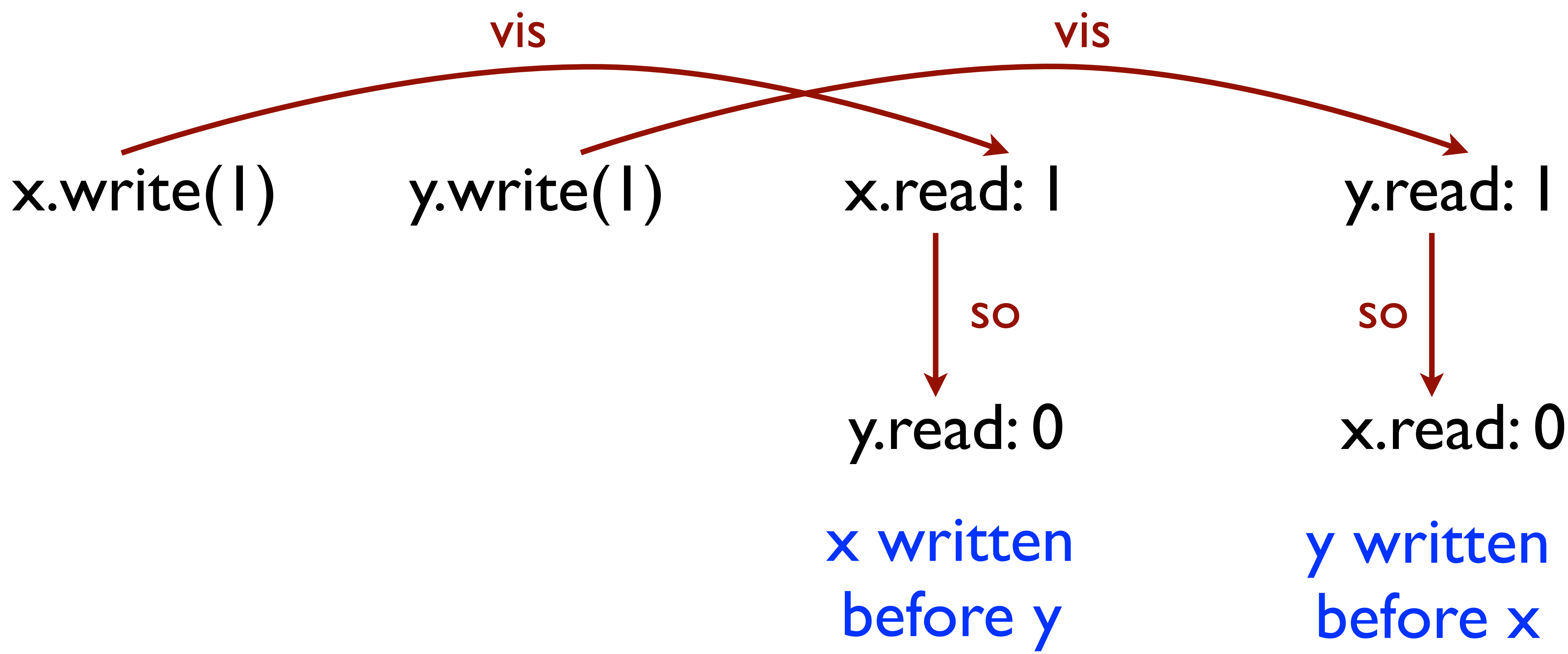


Implementations: updates delivered later

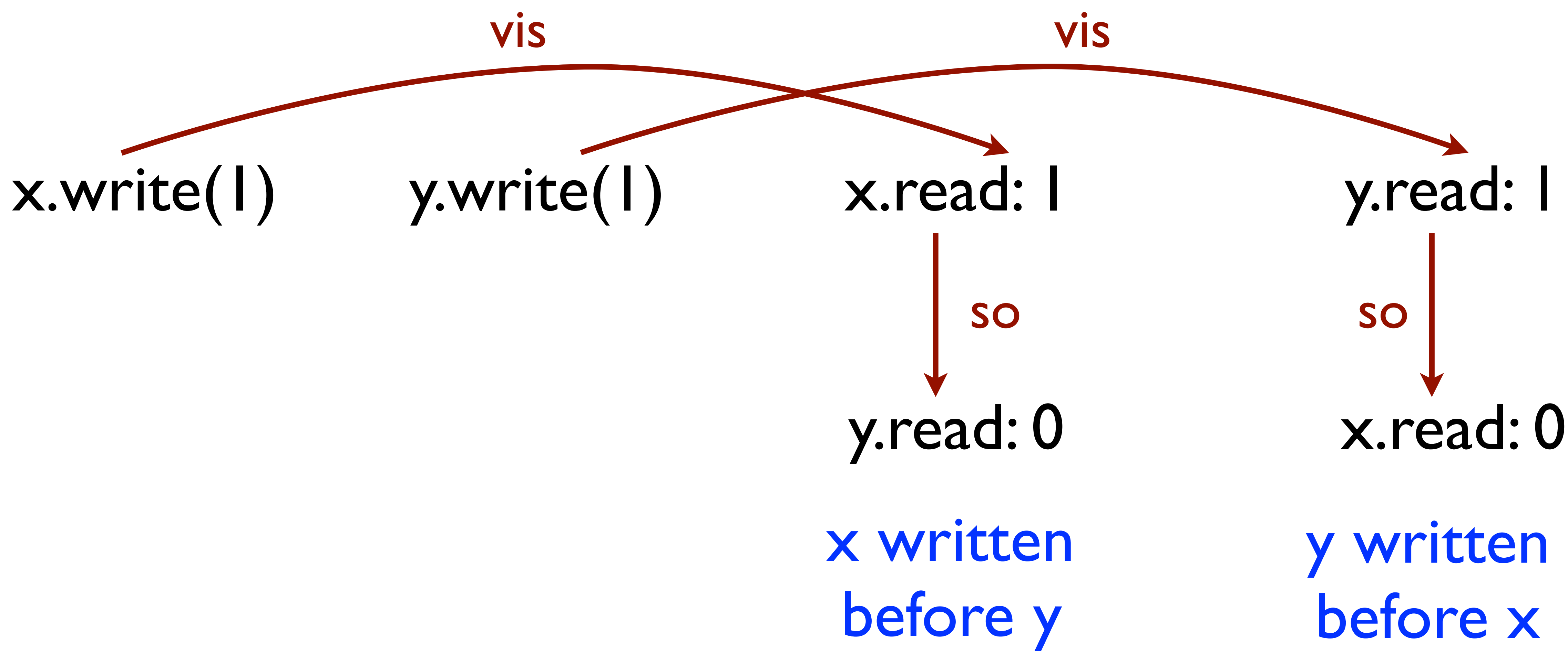
Independent reads of independent writes (IRIW)



Independent reads of independent writes (IRIW)

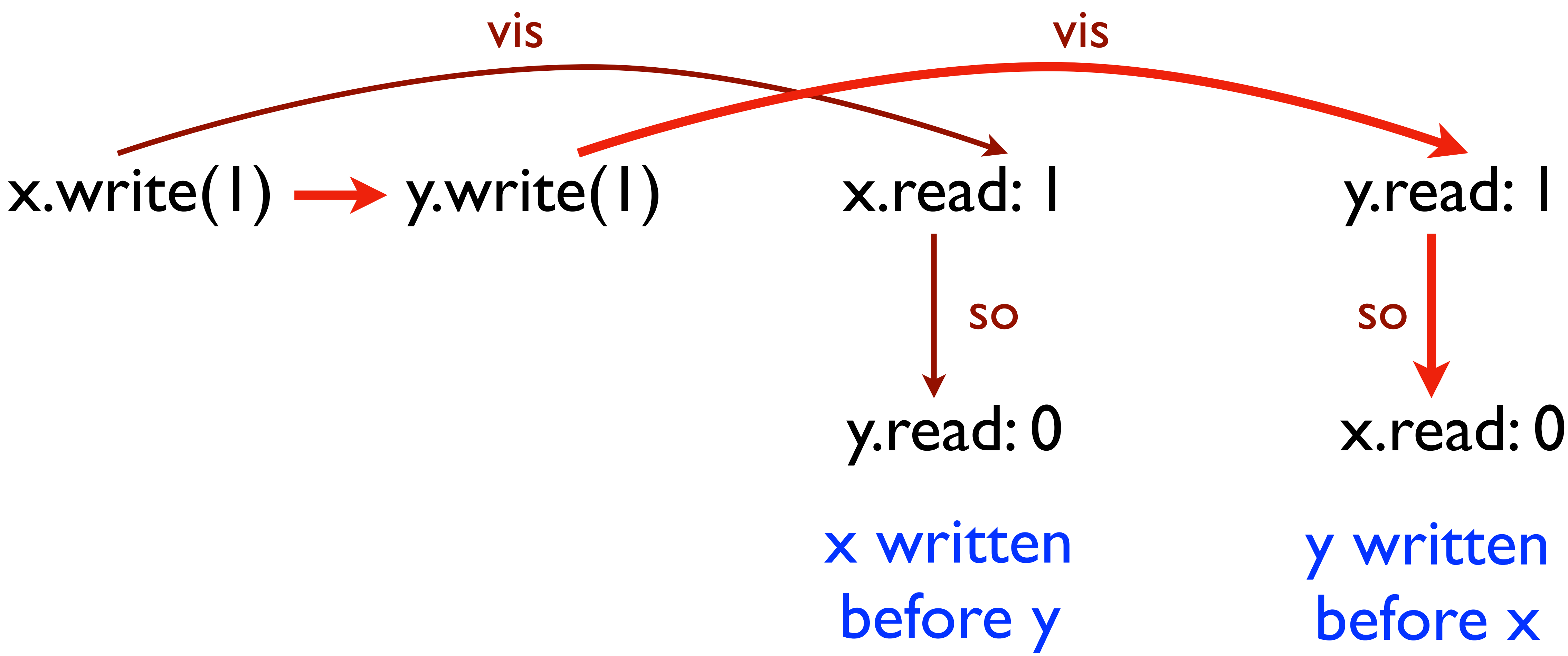


Independent reads of independent writes (IRIW)

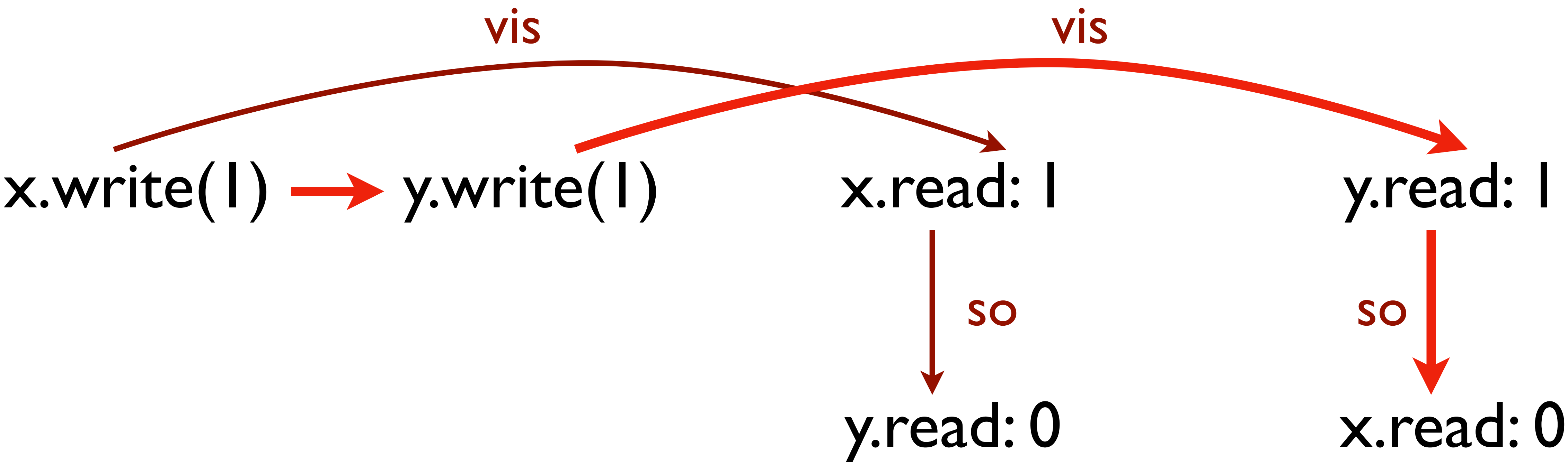


Implementations: no causal dependency between the two writes
➜ can be delivered in different orders at different replicas

Independent reads of independent writes (IRIW)



Independent reads of independent writes (IRIW)

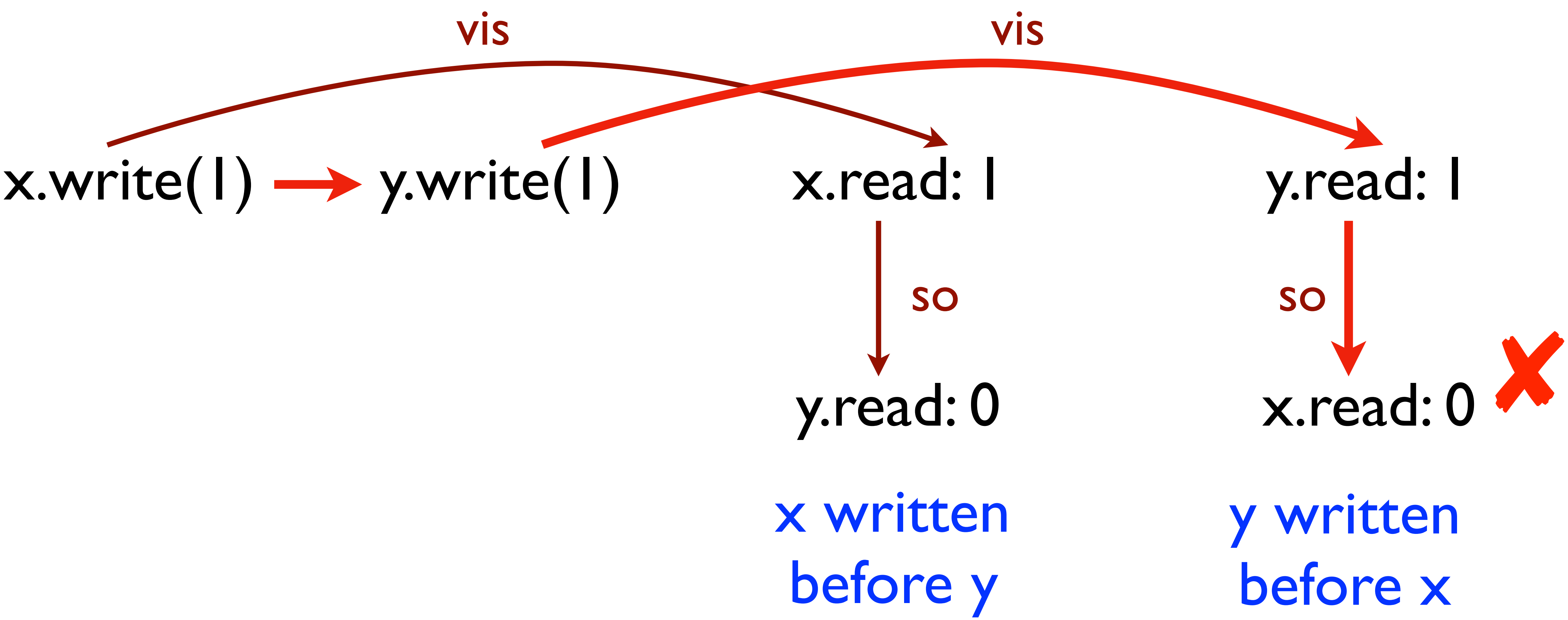


x written before y

y written before x

Not sequentially consistent

Independent reads of independent writes (IRIW)



Not sequentially consistent

Sequential consistency

- so $\subseteq$ vis and vis is total
- vis $\subseteq$ ar $\implies$ can equivalently require so $\subseteq$ vis = ar
- Every operation sees the effect of all operations preceding it in vis
- Like the original definition with to = vis = ar

Dekker example

x.write(1)

vis so

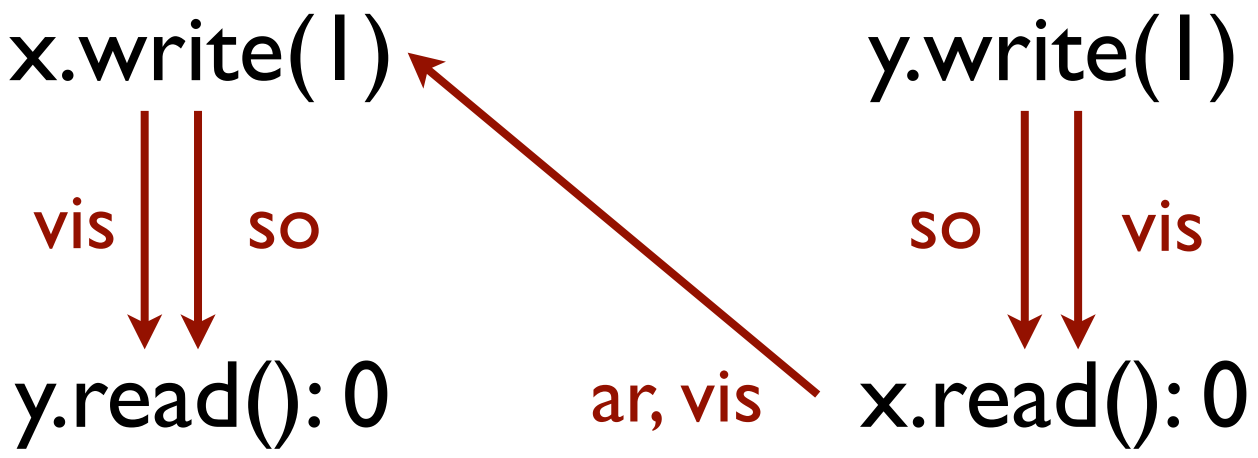
y.read(): 0

y.write(1)

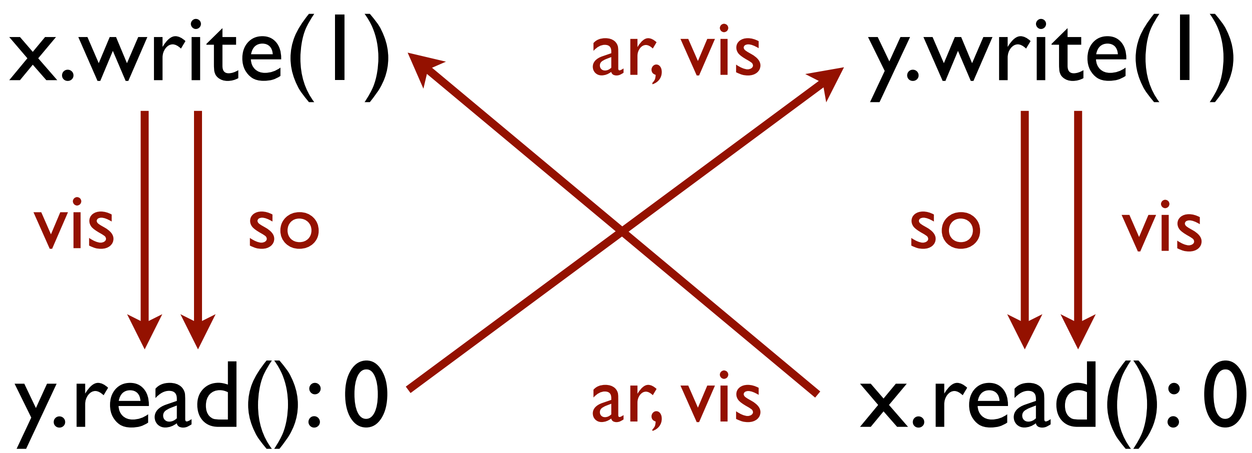
so vis

x.read(): 0

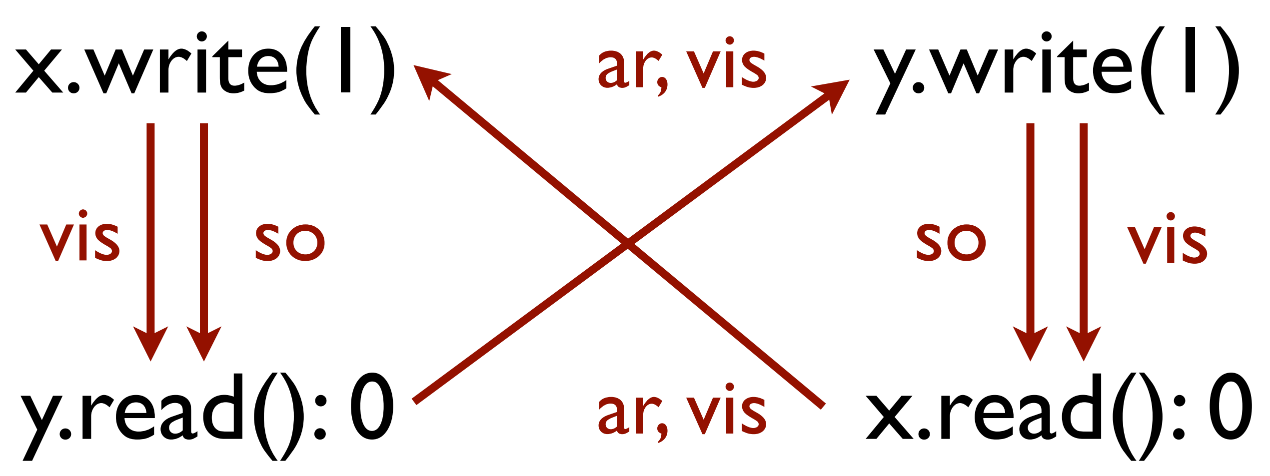
Dekker example



Dekker example



Dekker example



No execution with such history

Consistency zoo

- Eventual consistency
- Session guarantees: Dekker, IRIW, causality violation

  $so \subseteq vis$, $vis; so \subseteq vis$
- Causal consistency: Dekker, IRIW

  $(so \cup vis)^+ \subseteq vis$
- Prefix consistency: Dekker

  $ar; (vis \setminus so) \subseteq vis$
- Sequential consistency

  $vis = ar$

Shared-memory models

- Sequential consistency first proposed in the context of shared memory (1979)
- Processors and languages don't provide sequential consistency: weak memory models, due to processor and compiler optimisations
- Our specifications similar to weak memory model definitions
- Consistency axioms for last-writer-wins registers ~ shared-memory models

Consistency zoo

- Eventual consistency
- Session guarantees: Dekker, IRIW, causality violation

  $so \subsetneq vis$, $vis; so \subsetneq vis$
- Causal consistency: Dekker, IRIW

  $(so \cup vis)^+ \subsetneq vis$
- Prefix consistency: Dekker

  $ar; (vis \setminus so) \subsetneq vis$
- Sequential consistency

  $vis = ar$

for last-writer-wins = C++ release/acquire

Theoretical results

- Eventual consistency
- Session guarantees

  $so \subsetneq vis,\ vis; so \subsetneq vis$
- Causal consistency

  $(so \cup vis)^+ \subsetneq vis$
- Prefix consistency

  $ar; (vis \setminus so) \subsetneq vis$
- Sequential consistency

  $vis = ar$

- What's the best we can do while staying available under network partitionings?
- Causal consistency is a strongest such model [Attiya et al., 2015]

Theoretical results

- Eventual consistency
- Session guarantees

  $so \subsetneq vis, vis; so \subsetneq vis$
- Causal consistency

  $(so \cup vis)^+ \subsetneq vis$

- Prefix consistency

  $ar; (vis \setminus so) \subsetneq vis$
- Sequential consistency

  $vis = ar$

- What's the best we can do while staying available under network partitionings?
- Causal consistency is a strongest such model [Attiya et al., 2015]

Theoretical results

- Eventual consistency
- Session guarantees

  $so \subseteq vis$, $vis; so \subseteq vis$
- Causal consistency

  $(so \cup vis)^+ \subseteq vis$

- Prefix consistency

  $ar; (vis \setminus so) \subseteq vis$
- Sequential consistency

  $vis = ar$

- What's the best we can do while staying available under network partitionings?
- Causal consistency is a strongest such model [Attiya et al., 2015]

Terms and conditions apply:

- for a certain version of CC and a certain class of implementations
- **a** strongest model: cannot be strengthened, but can be other alternative incomparable models

Theoretical results

- Application of eventual consistency - collaborative editing: Google Docs, Office Online
- At the core: list data type (of formatted characters)
- List data type has an inherently high metadata overhead: can't discard a character when deleting it from a Google Docs document! [Attiya et al., 2016]
- Discarding may allow previously deleted elements to reappear

Determining the right level of consistency

Application correctness

- Does an application satisfy a particular correctness property?

  Integrity invariants: account balance is non-negative

- Is an application robust against a particular consistency model?

  Application behaves the same as when using a strongly consistent database

Application correctness

- Does an application satisfy a particular correctness property?

  Integrity invariants: account balance is non-negative

- Is an application robust against a particular consistency model?

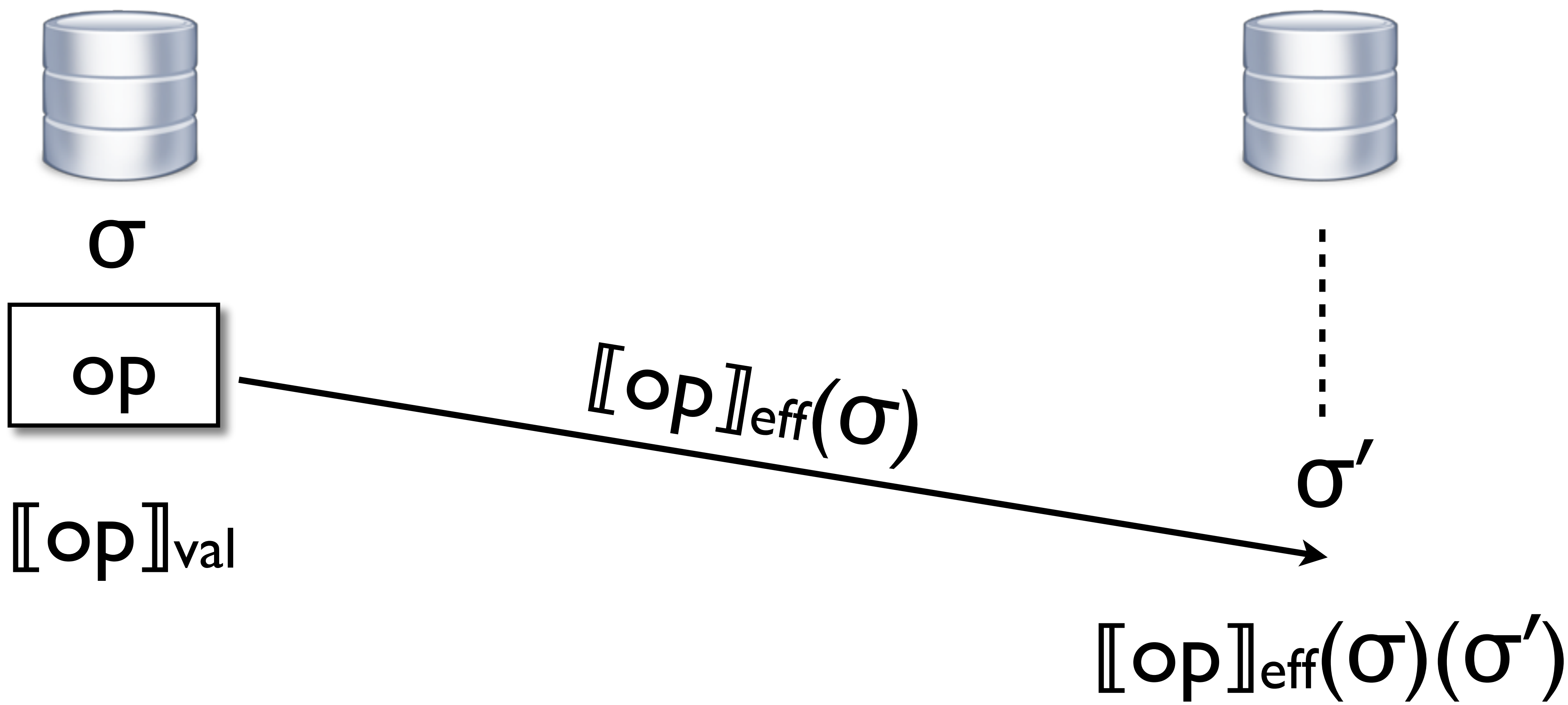
  Application behaves the same as when using a strongly consistent database

Challenge

Vanilla weak consistency often too weak to preserve correctness

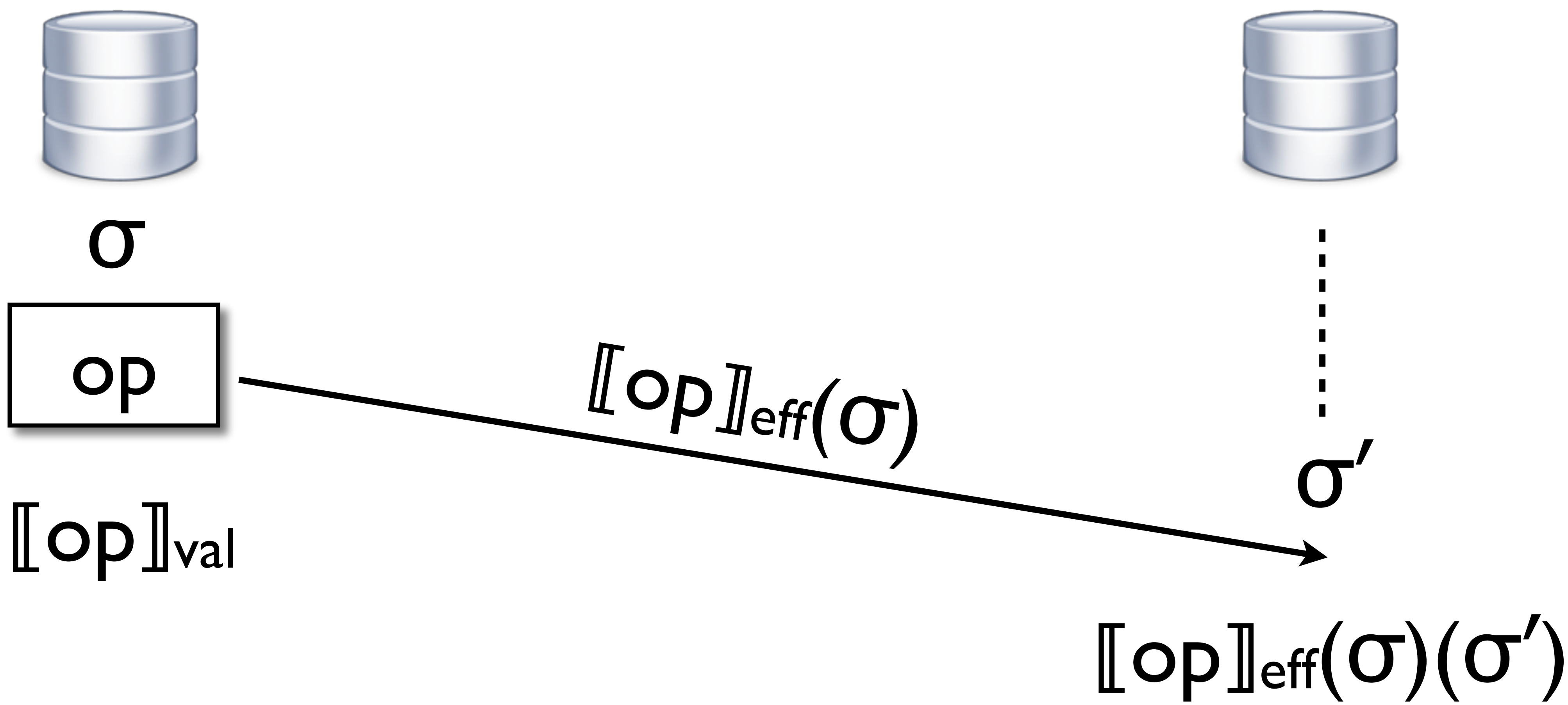
Need to strengthen consistency in parts of the application

Deposits



$$[\![add(100)]\!]_{eff}(\sigma) \; = \; \lambda\sigma'.\,(\sigma' + 100)$$

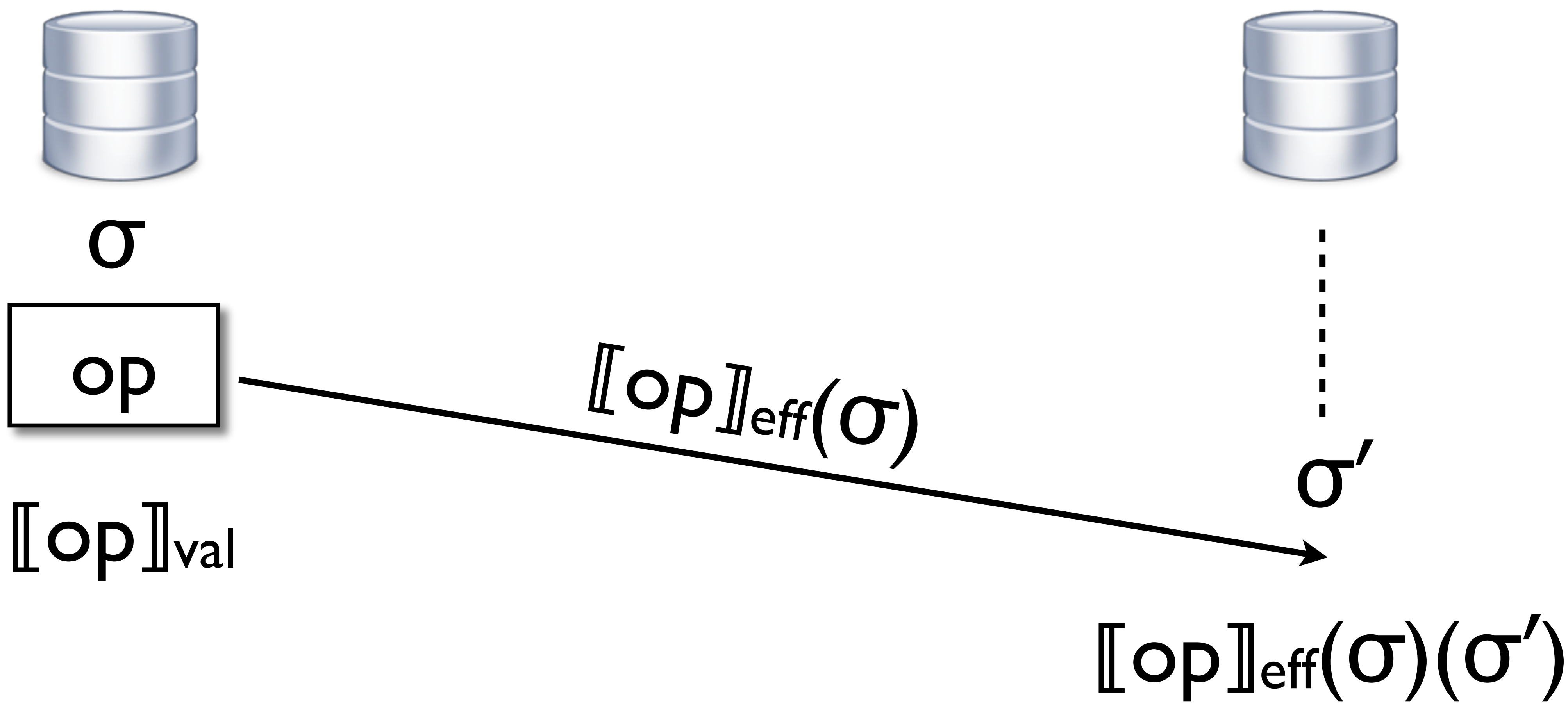
Withdrawals



$$[\![\text{withdraw}(100)]\!]_{\text{eff}}(\sigma) \;=$$

$$\text{if } \sigma \geq 100 \text{ then } (\lambda\sigma'.\, \sigma' - 100) \text{ else } (\lambda\sigma'.\, \sigma')$$

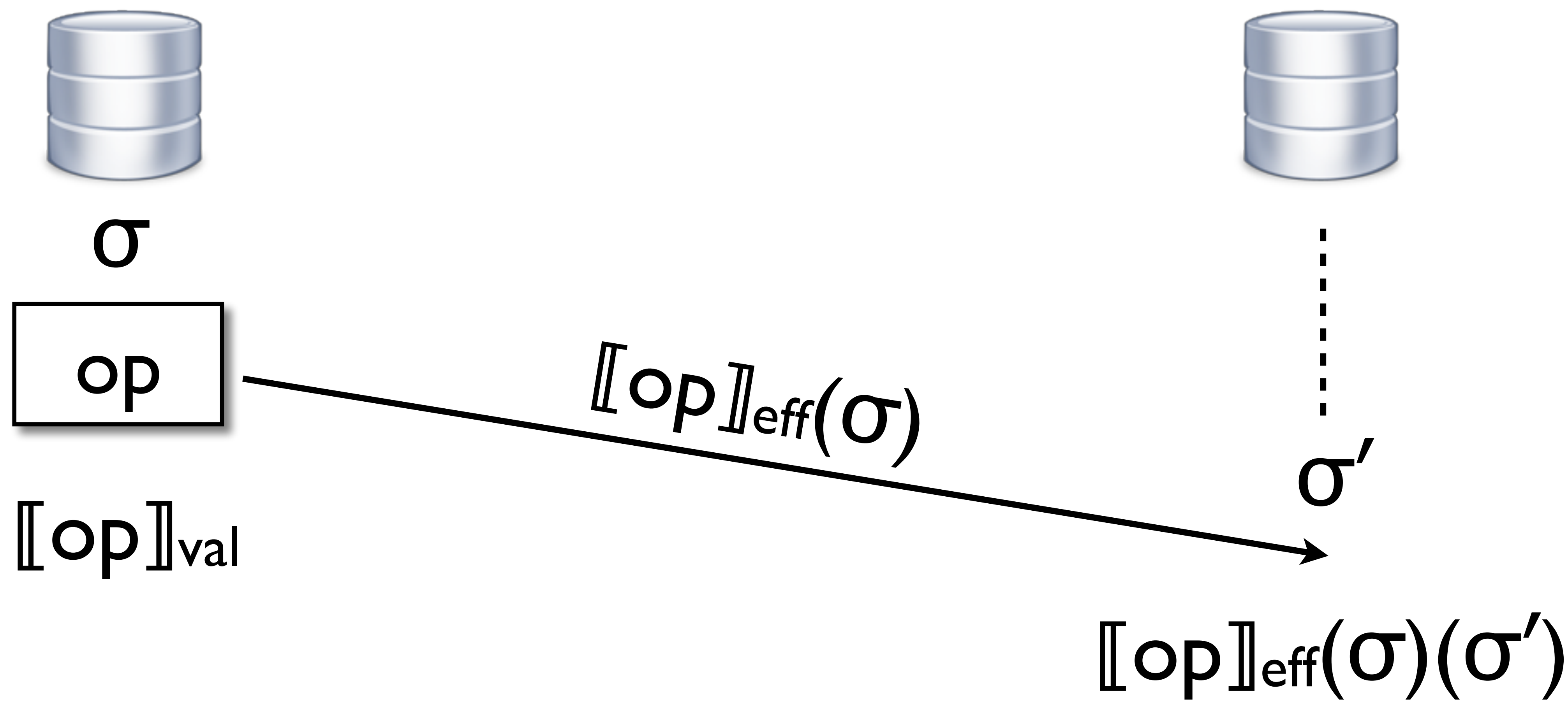
Withdrawals



$[\![\text{withdraw}(100)]\!]_{\text{eff}}(\sigma) \;=$

$\quad \text{if } \sigma \geq 100 \text{ then } (\lambda\sigma'.\, \sigma' - 100) \text{ else } (\lambda\sigma'.\, \sigma')$

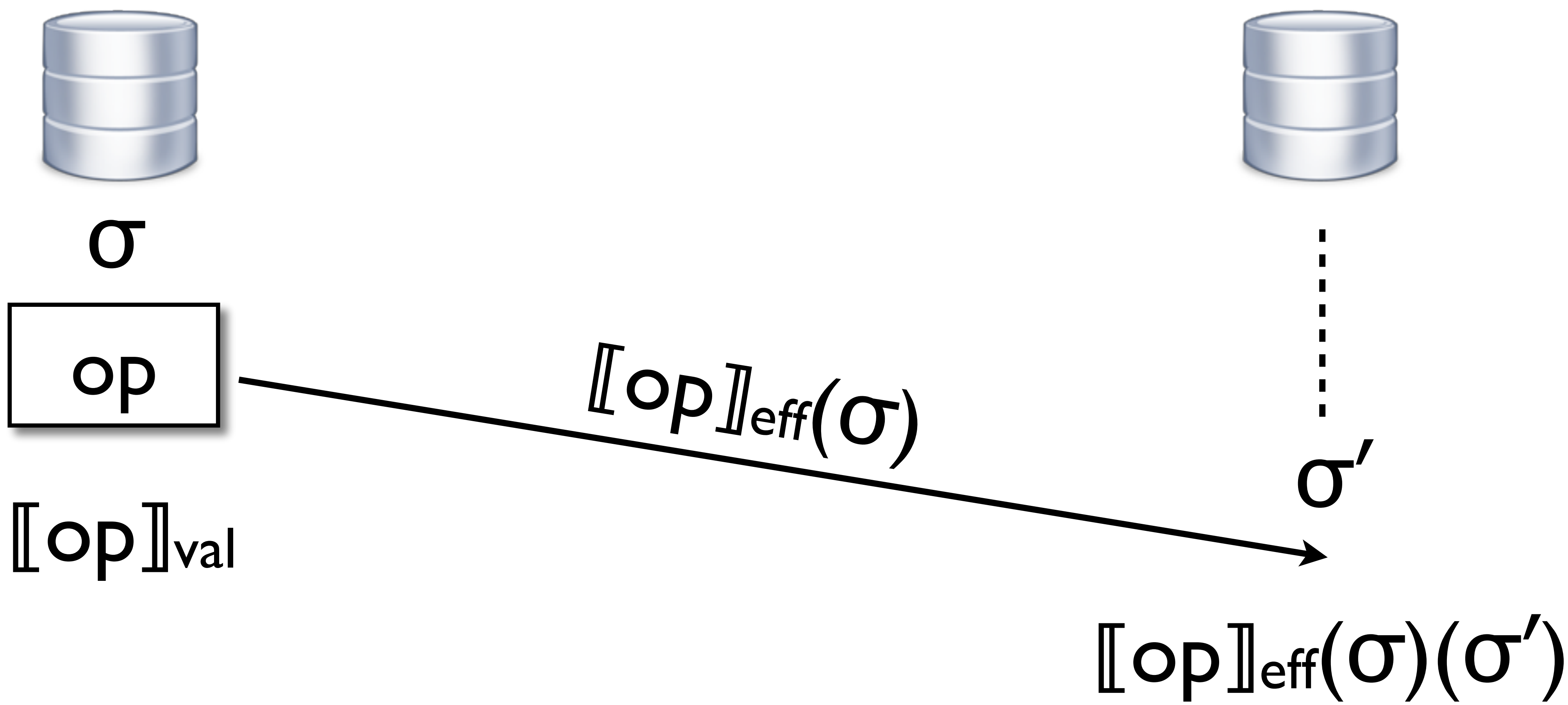
Withdrawals



$$[\![\mathrm{withdraw}(100)]\!]_{\mathrm{eff}}(\sigma) \;=$$

$$\text{if } \sigma \geq 100 \text{ then } (\lambda\sigma'.\, \sigma' - 100) \text{ else } (\lambda\sigma'.\, \sigma')$$

Withdrawals



$\llbracket \mathrm{withdraw}(100) \rrbracket_{\mathrm{eff}}(\sigma)$ =

if $\sigma \geq 100$ then $(\lambda\sigma'.\, \sigma' - 100)$ else $(\lambda\sigma'.\, \sigma')$

balance = 100

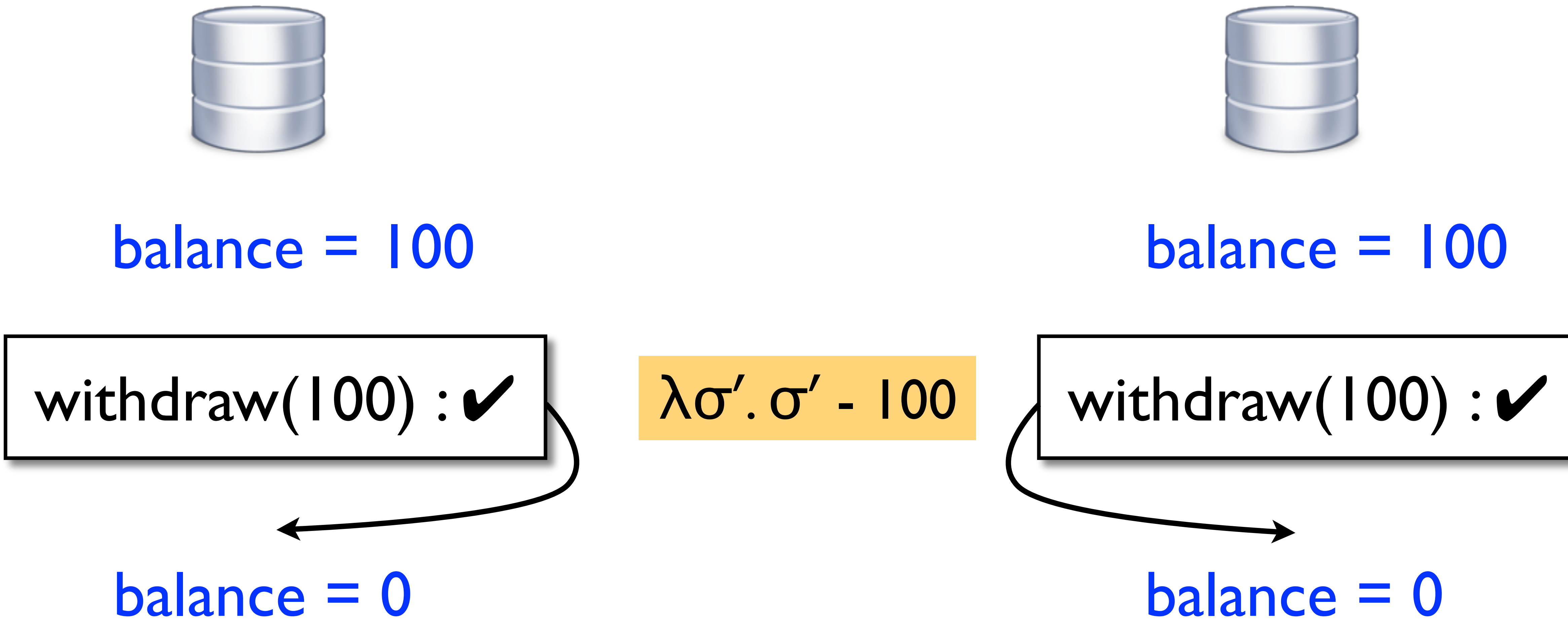
withdraw(100) : ✔

balance = 100

withdraw(100) : ✔

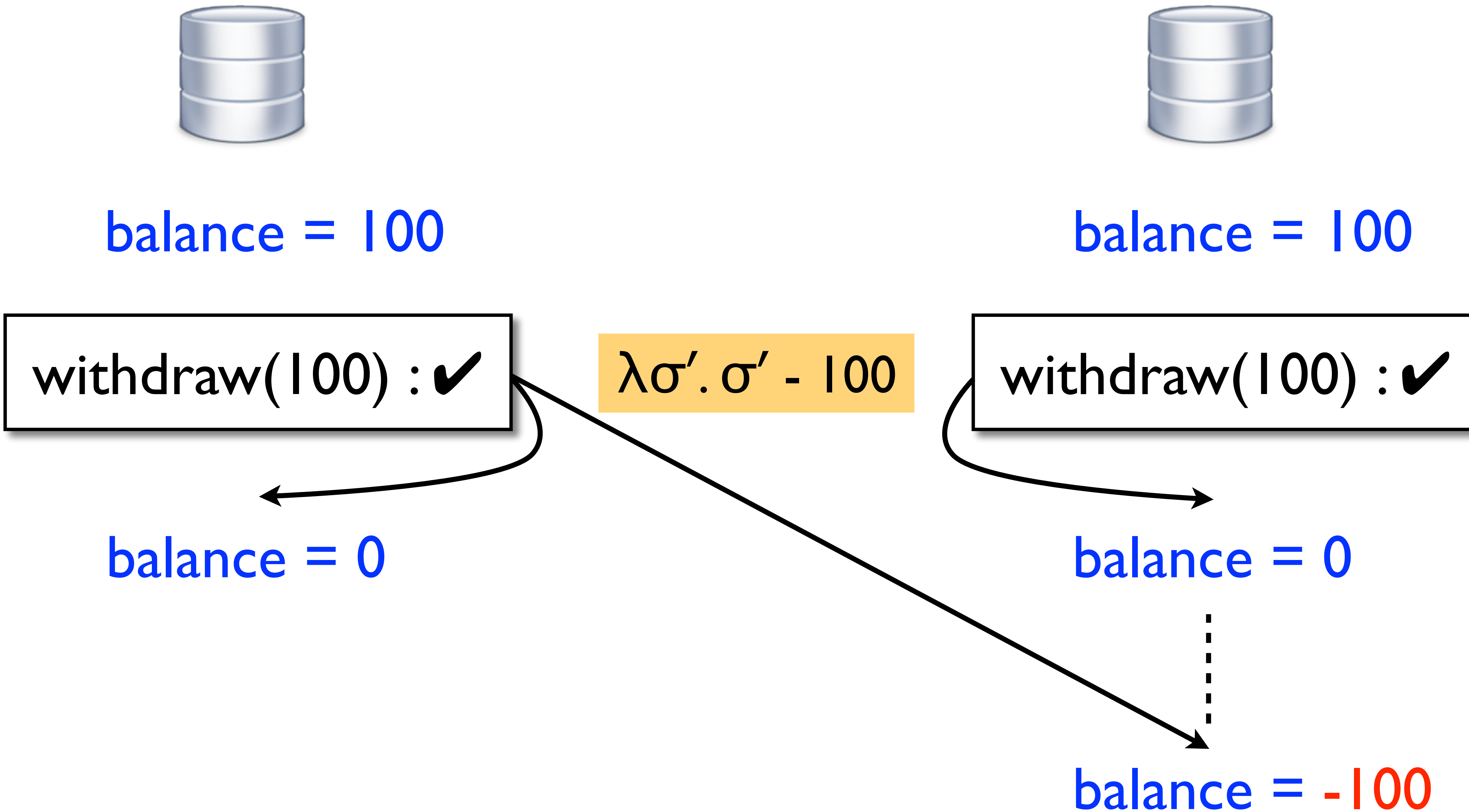
$$[\![\text{withdraw}(100)]\!]_{\text{eff}}(\sigma) \;=$$

$$\text{if } \sigma \geq 100 \text{ then } (\lambda\sigma'.\, \sigma' - 100) \text{ else } (\lambda\sigma'.\, \sigma')$$



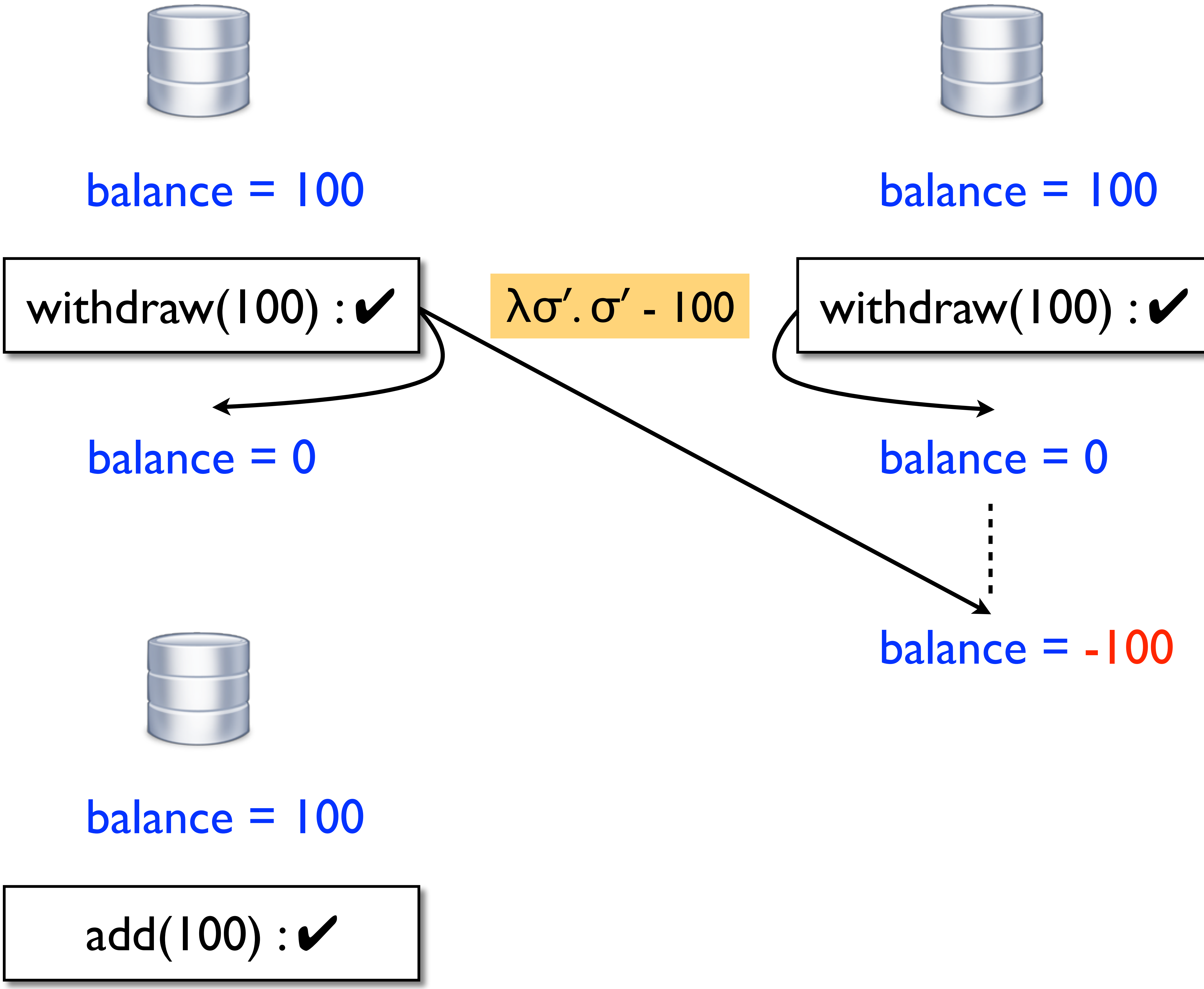
$$[\![\text{withdraw}(100)]\!]_{\text{eff}}(\sigma) \;=$$

$$\text{if } \sigma \geq 100 \text{ then } (\lambda\sigma'.\,\sigma' - 100) \text{ else } (\lambda\sigma'.\,\sigma')$$

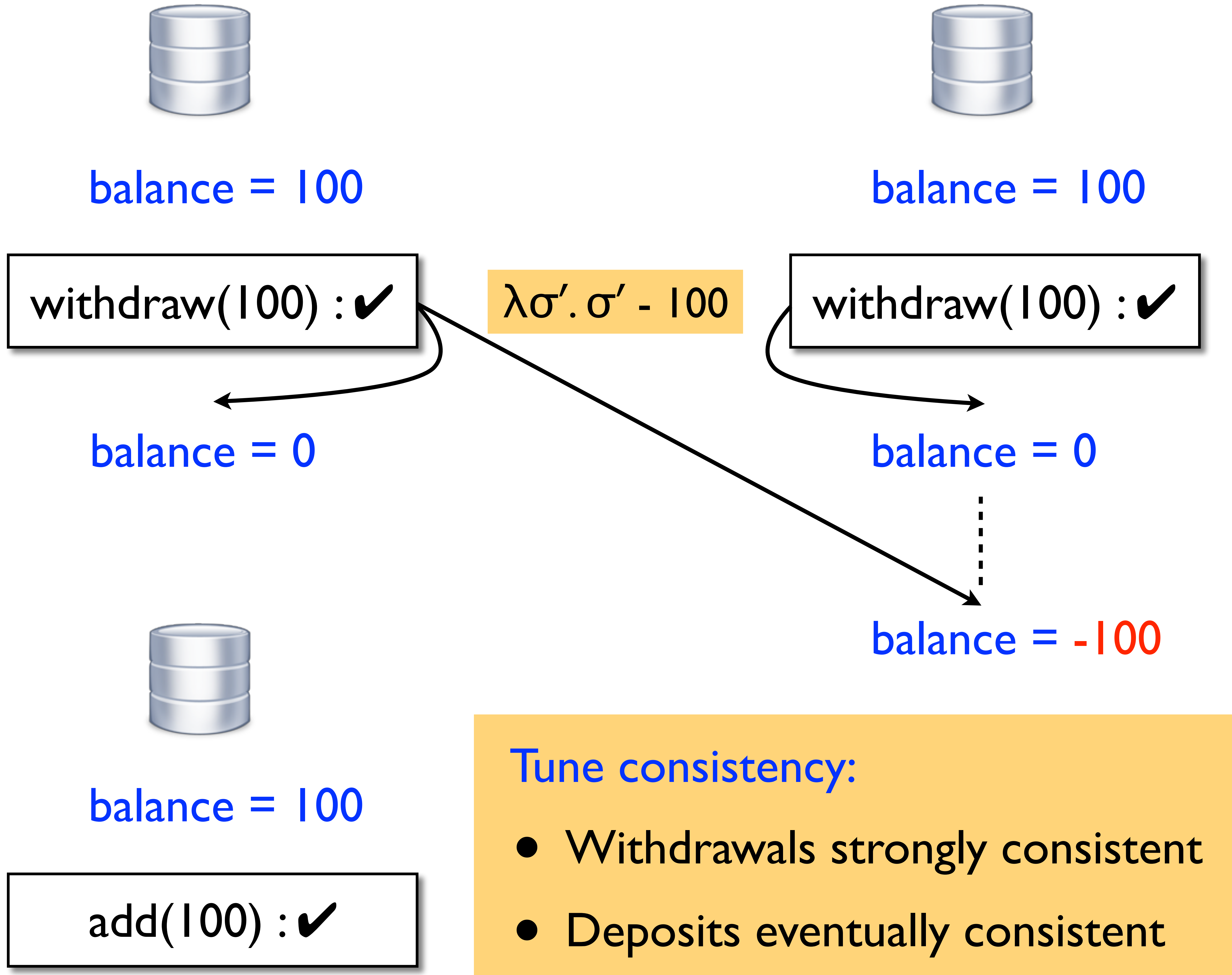


$$[\![\text{withdraw}(100)]\!]_{\text{eff}}(\sigma) =$$

$$\text{if } \sigma \geq 100 \text{ then } (\lambda\sigma'.\,\sigma' - 100) \text{ else } (\lambda\sigma'.\,\sigma')$$



balance = 100
balance = 100
withdraw(100) : ✔
λσ′. σ′ - 100
withdraw(100) : ✔
balance = 0
balance = 0
balance = -100
balance = 100
add(100) : ✔

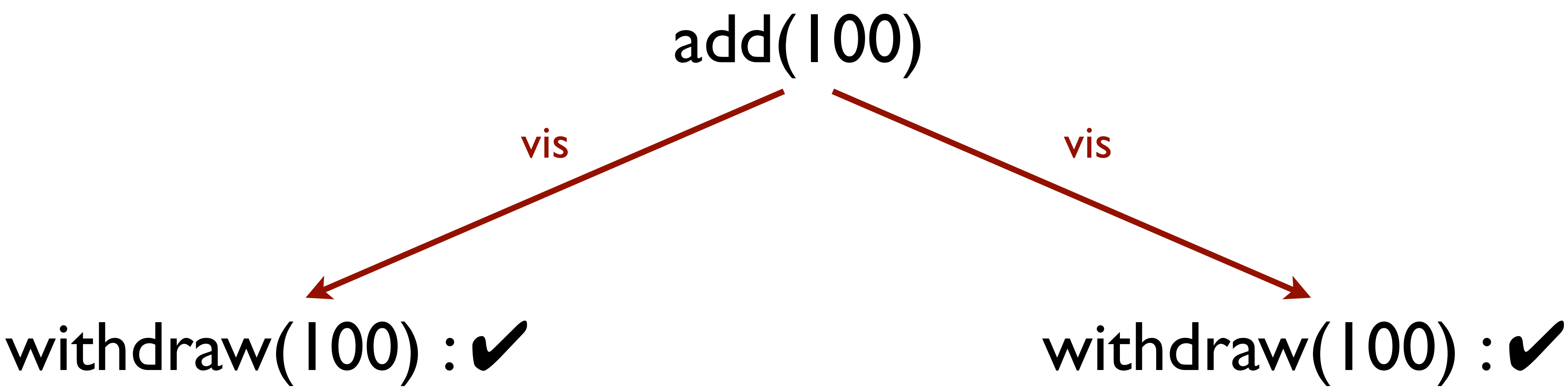

balance = 100
balance = 100
withdraw(100) : ✔
λσ′. σ′ - 100
withdraw(100) : ✔
balance = 0
balance = 0
balance = -100
balance = 100
add(100) : ✔
Tune consistency:
• Withdrawals strongly consistent
• Deposits eventually consistent

Strengthening consistency

$$\begin{tikzcd} & \text{add(100)} \arrow[dl, "\text{vis}"'] \arrow[dr, "\text{vis}"] & \\ \text{withdraw(100)} : \checkmark & & \text{withdraw(100)} : \checkmark \end{tikzcd}$$

- Baseline model: causal consistency
- Problem: withdrawals are causally independent

Strengthening consistency



- Symmetric conflict relation on operations: $\bowtie \subseteq Op \times Op$, e.g., $withdraw \bowtie withdraw$
- Conflicting operations cannot be causally independent: $\forall\, e, f \in E.\ op(e) \bowtie op(f) \implies e \xrightarrow{vis} f \vee f \xrightarrow{vis} e$

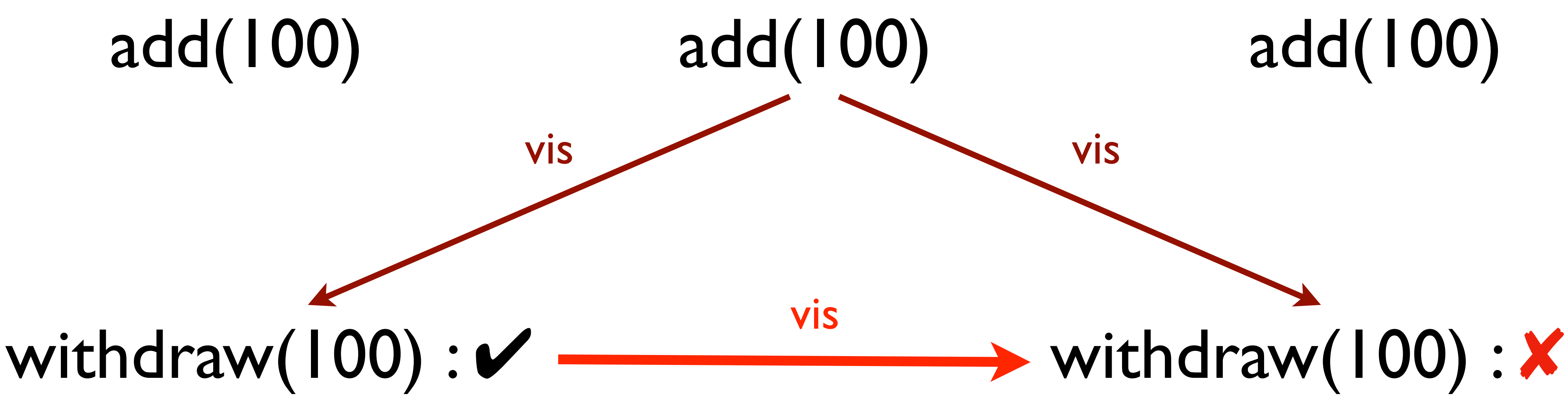
Strengthening consistency

$$\begin{tikzcd}
& \mathrm{add}(100) \arrow[dl, "\mathrm{vis}"'] \arrow[dr, "\mathrm{vis}"] & \\
\mathrm{withdraw}(100) : \checkmark \arrow[rr, "\mathrm{vis}"] & & \mathrm{withdraw}(100) : \times
\end{tikzcd}$$

- Symmetric conflict relation on operations:
  $\bowtie \subseteq \mathrm{Op} \times \mathrm{Op}$, e.g., $\mathrm{withdraw} \bowtie \mathrm{withdraw}$
- Conflicting operations cannot be causally independent:

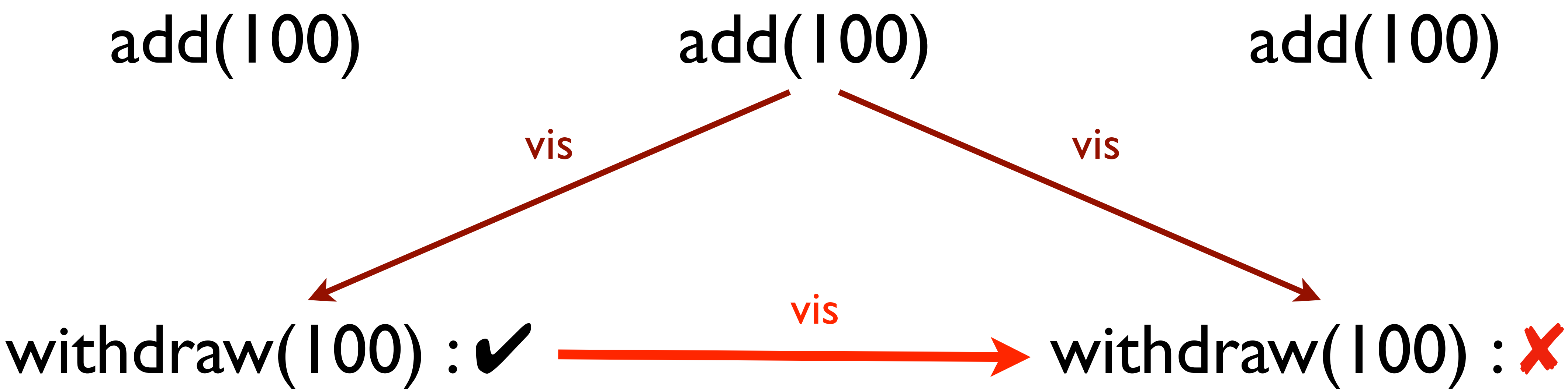
  $\forall\, e, f \in E.\ \mathrm{op}(e) \bowtie \mathrm{op}(f) \Longrightarrow e \xrightarrow{\mathrm{vis}} f \vee f \xrightarrow{\mathrm{vis}} e$

Strengthening consistency



- Symmetric conflict relation on operations:
  $\bowtie \subseteq \mathsf{Op} \times \mathsf{Op}$, e.g., $\mathsf{withdraw} \bowtie \mathsf{withdraw}$
- Conflicting operations cannot be causally independent:
  $\forall\, e, f \in E.\ op(e) \bowtie op(f) \implies e \xrightarrow{vis} f \lor f \xrightarrow{vis} e$
- No constraints on additions: $\neg(\mathsf{add} \bowtie \mathsf{op})$

Strengthening consistency



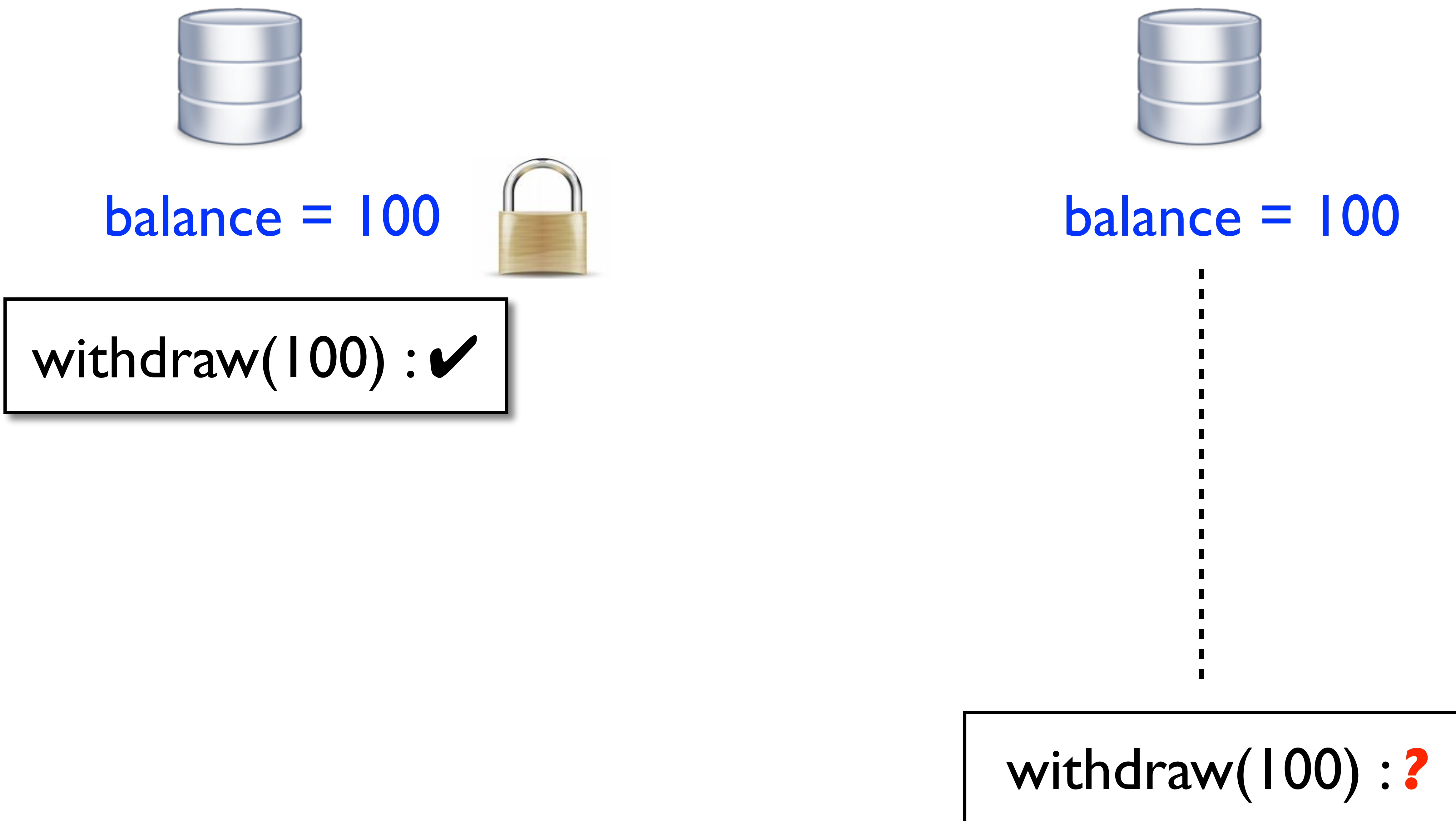
- Implementation requires replicas executing withdraw() to synchronise
- add() doesn't need synchronisation

withdraw $\bowtie$ withdraw: as if withdraw grabs an exclusive lock (mutex) on the account

balance = 100

withdraw(100) : ✔

withdraw $\bowtie$ withdraw: as if withdraw grabs an exclusive lock (mutex) on the account



withdraw ⋈ withdraw: as if withdraw grabs an exclusive lock (mutex) on the account

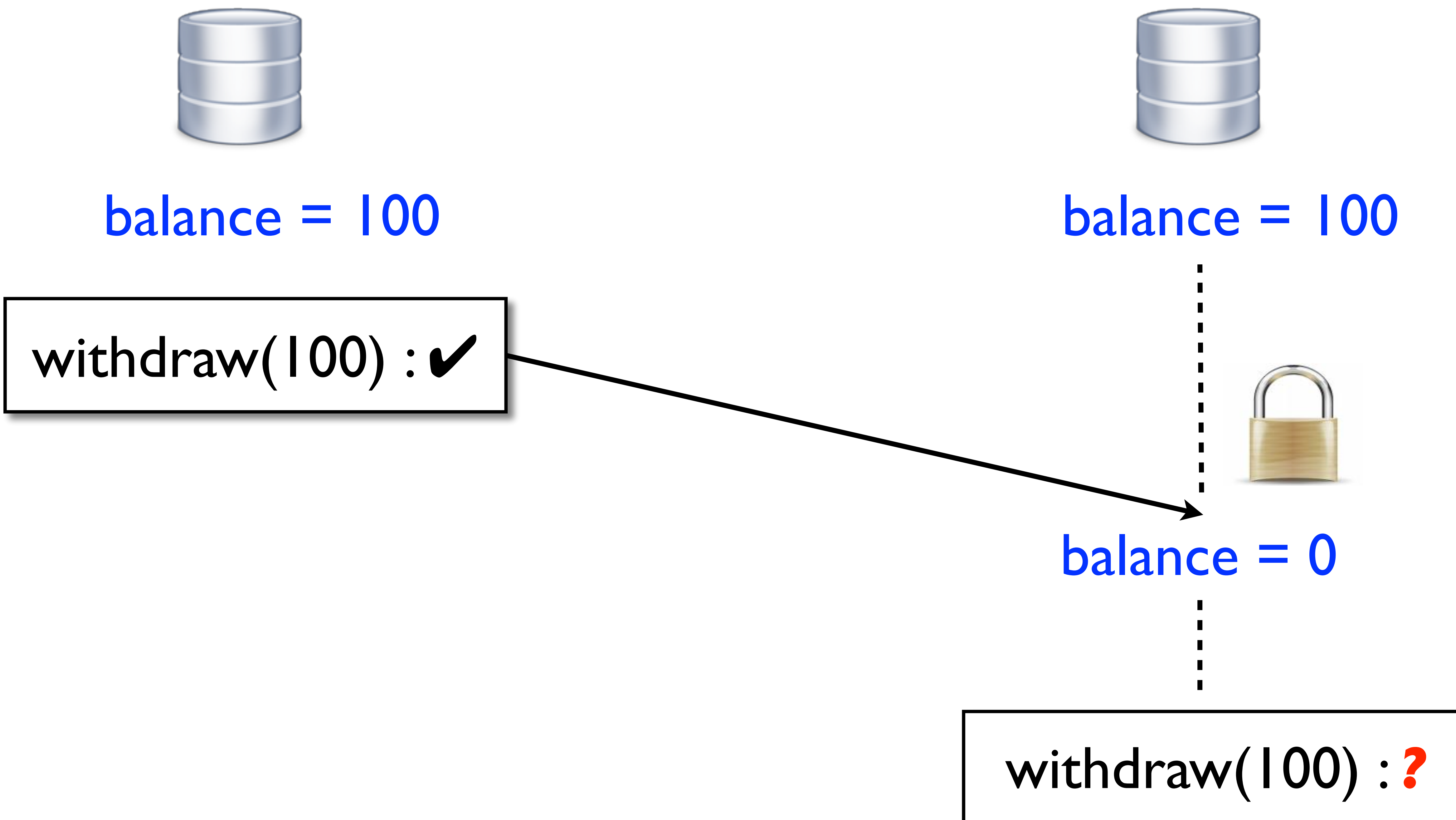
balance = 100

withdraw(100) : ✔

balance = 100

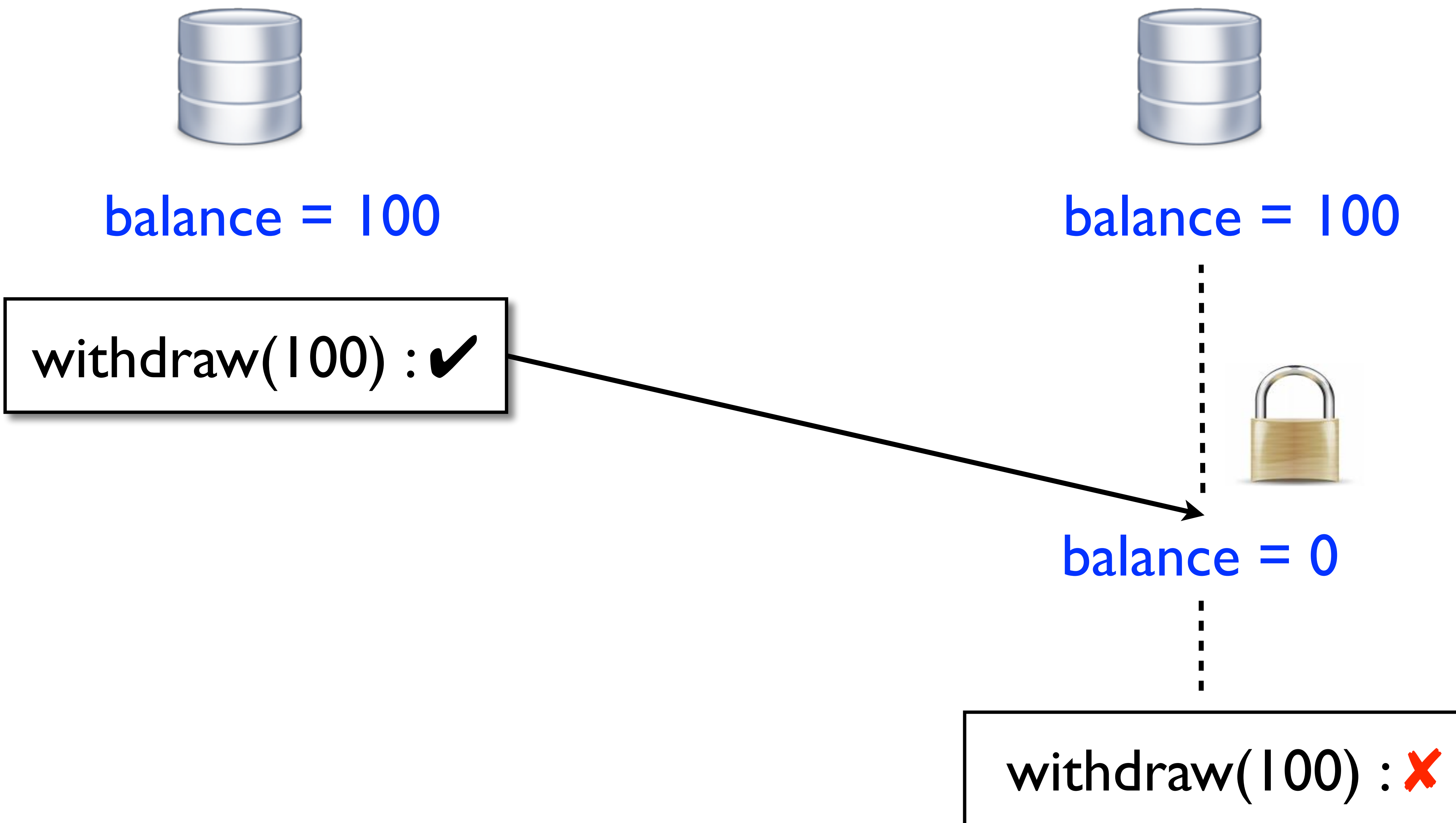
withdraw(100) : ?

withdraw $\bowtie$ withdraw: as if withdraw grabs an exclusive lock (mutex) on the account

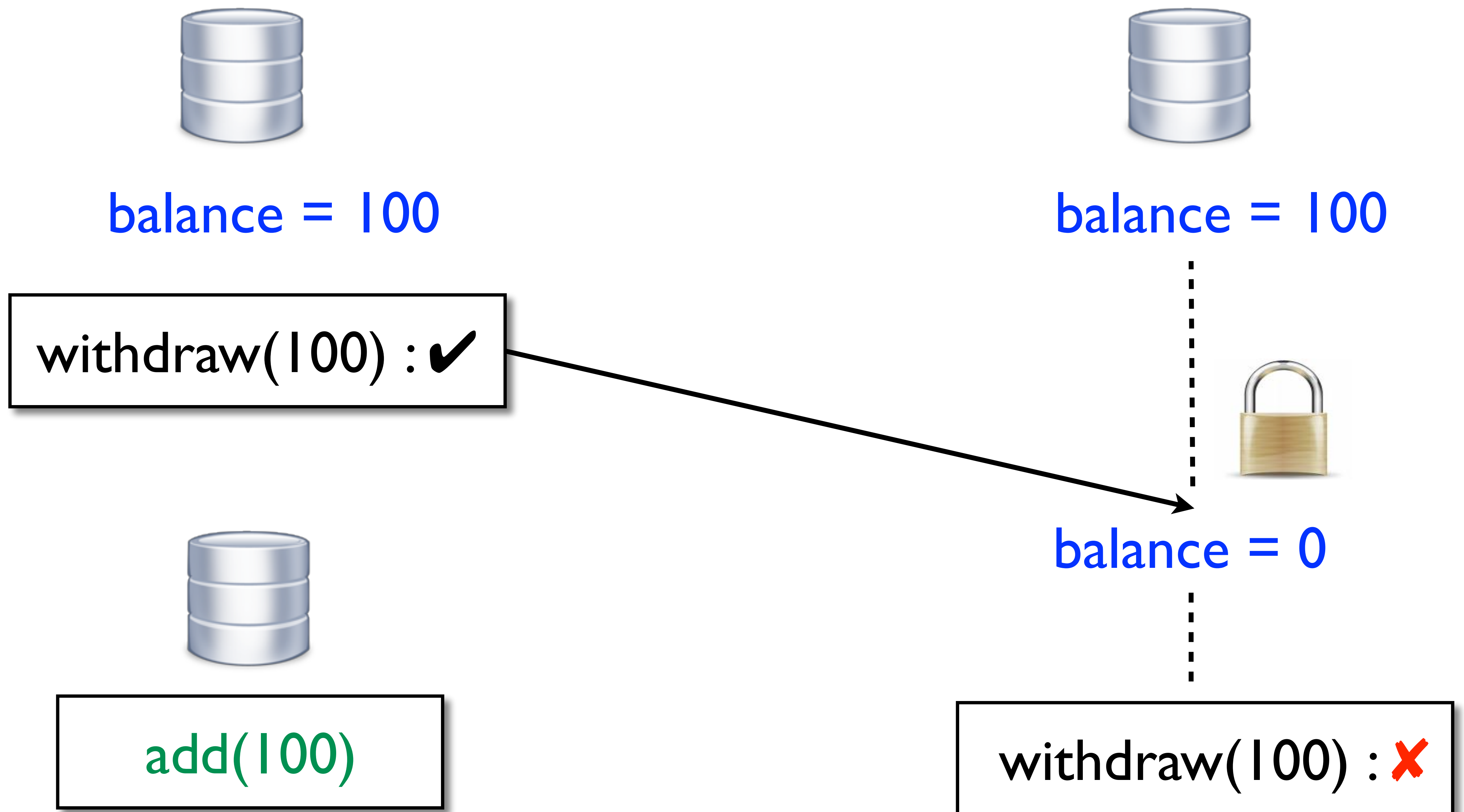


Acquiring the lock requires bringing all operations the replica holding it knows about

withdraw $\bowtie$ withdraw: as if withdraw grabs an exclusive lock (mutex) on the account



withdraw $\bowtie$ withdraw: as if withdraw grabs an exclusive lock (mutex) on the account



¬(add $\bowtie$ op): no locks, so no synchronisation

Consistency choices

- Databases with multiple consistency levels:
  - Commercial: Amazon DynamoDB, Microsoft DocumentDB
  - Research: Li+ OSDI'12; Terry+ SOSP'13; Balegas+ EuroSys'15; Li+ USENIX ATC'18
- Stronger operations require synchronisation between replicas
- Pay for stronger semantics with latency, possible unavailability and money

Consistency choices

- Hard to figure out the minimum consistency level necessary to maintain correctness
- Reason about all possible abstract executions?
  - Abstract from some of implementation details, but still describe behaviour of the whole system
  - Number of possible executions is exponential: e.g., choices of vis = order of message deliveries
- Need verification techniques that limit the exponential blow-up

Verification problem

Given

- a set of operations: withdraw(), deposit(), ...
- a conflict relation: *withdraw* $\bowtie$ *withdraw*

Do the operations always preserve a given integrity invariant?

$I = (\text{balance} \geq 0)$

Verification problem

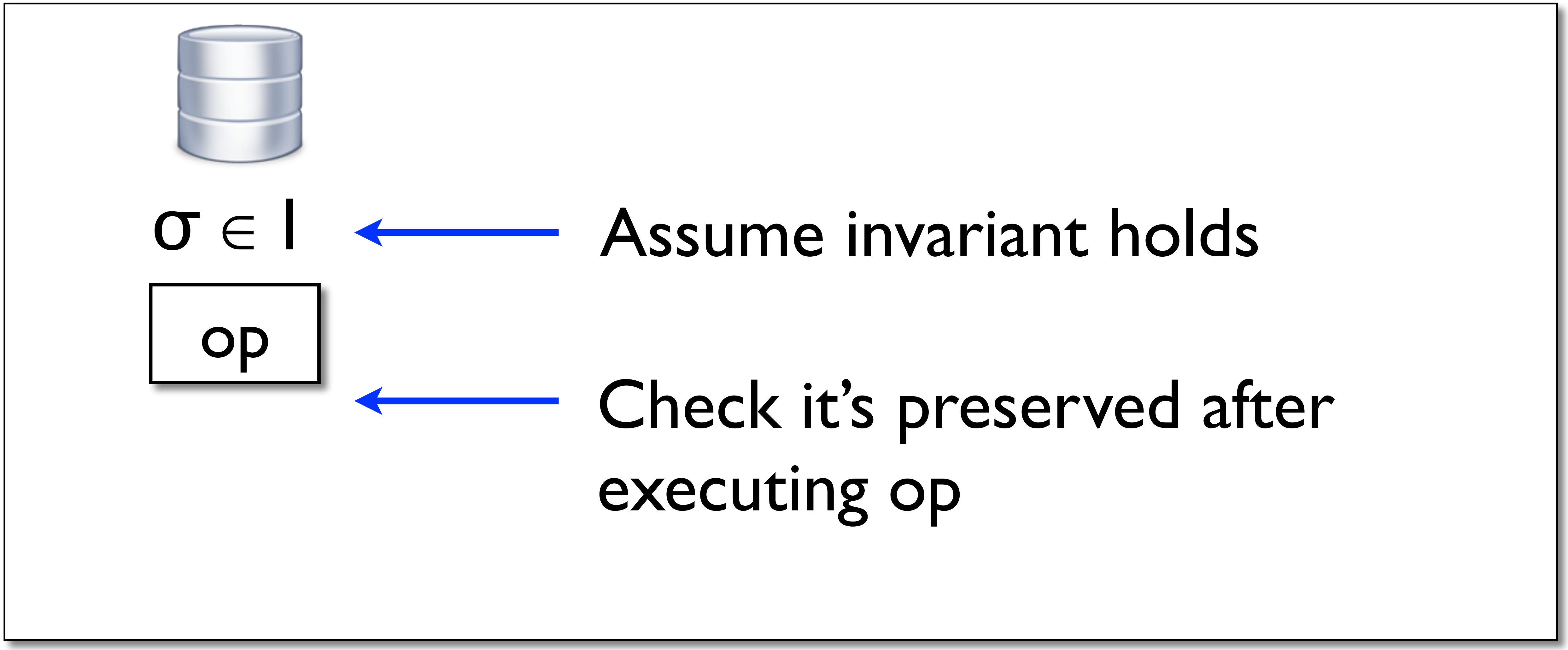
Given

- a set of operations: withdraw(), deposit(), ...
- a conflict relation: *withdraw* $\bowtie$ *withdraw*

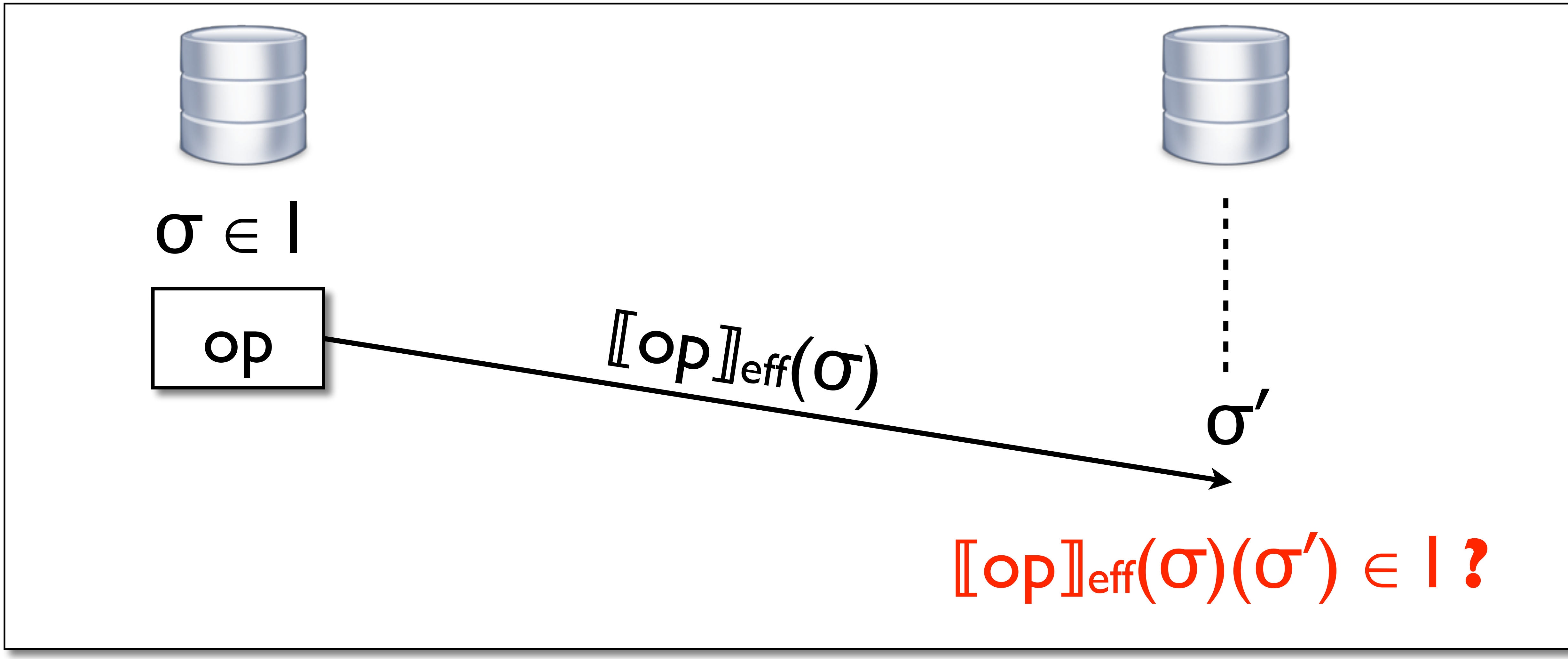
Do the operations always preserve a given integrity invariant?

$I = (\text{balance} \geq 0)$

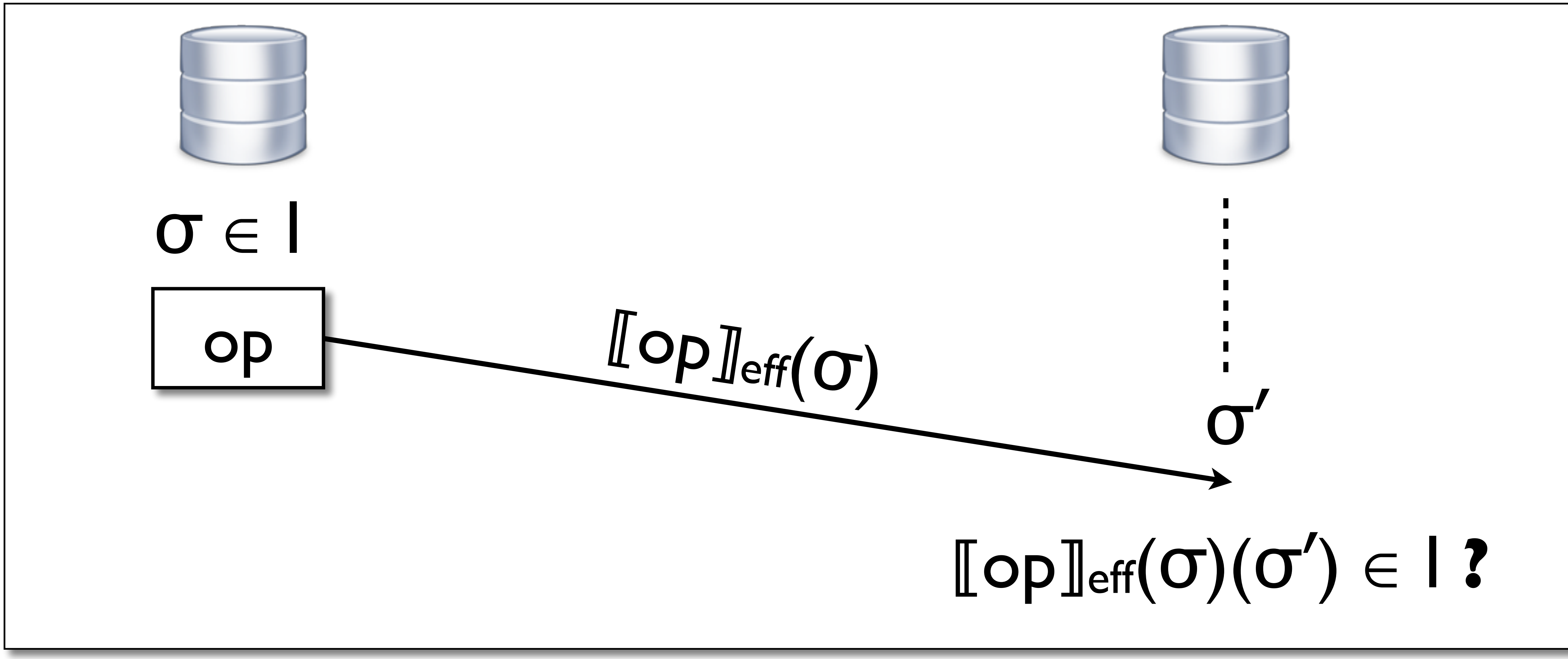
Later: operations ➜ whole transactions



Single check: no state-space explosion from concurrency



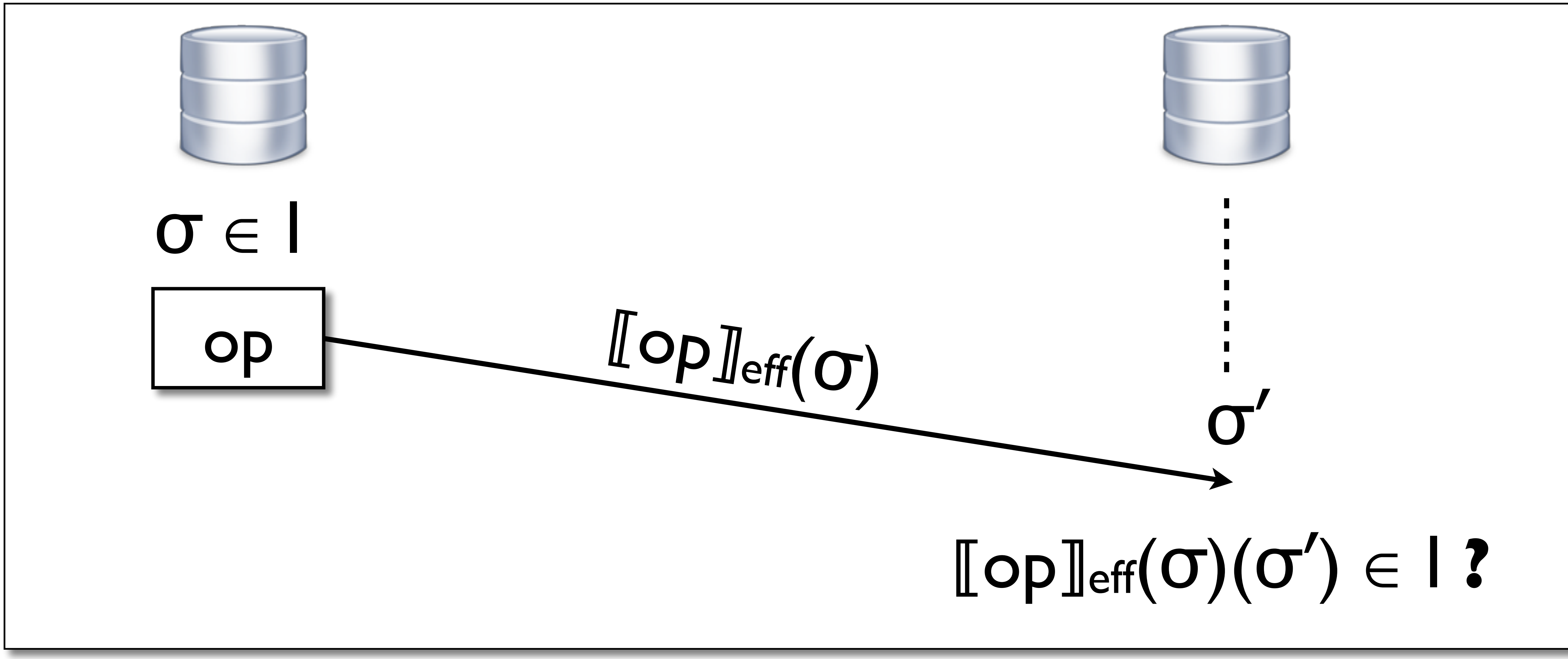
Effect applied in a different state!



$[\![op]\!]_{eff}(\sigma)$ = if $P(\sigma)$ then $f(\sigma)$ else if...

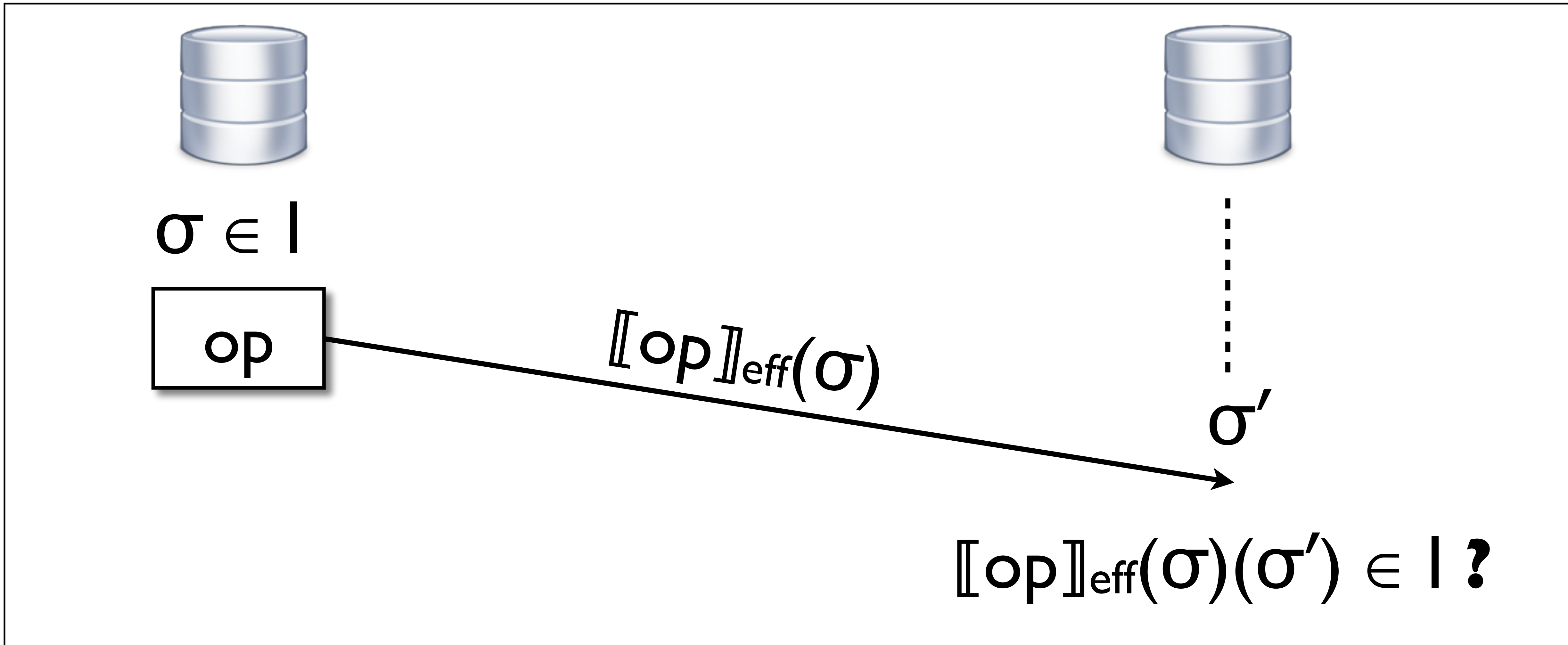
$[\![withdraw(100)]\!]_{eff}(\sigma)$ =

if $\sigma \geq 100$ then $(\lambda\sigma'.\, \sigma' - 100)$ else $(\lambda\sigma'.\, \sigma')$



$[\![op]\!]_{eff}(\sigma)$ = if $P(\sigma)$ then $f(\sigma)$ else if...

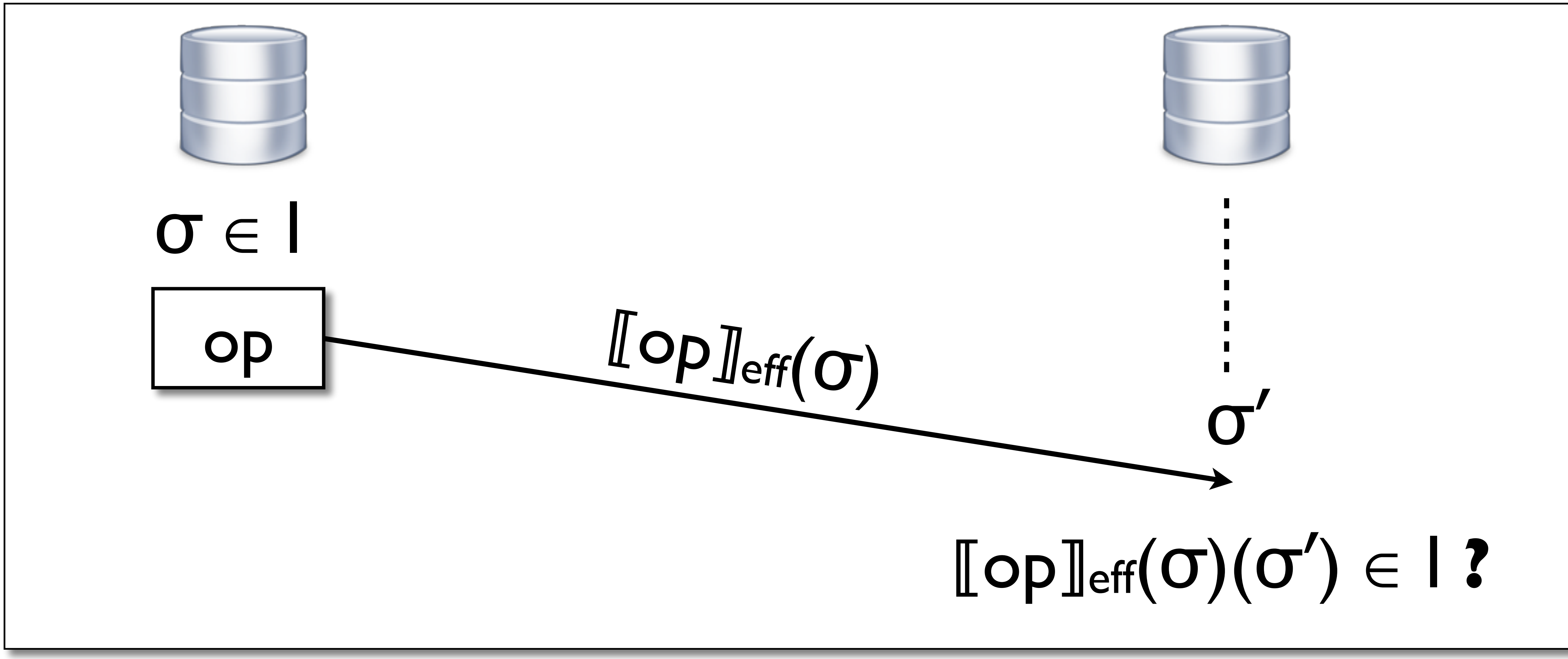
1. Effector safety: $f(\sigma)$ preserves I when executed in any state satisfying P: $\{I \wedge P\}\ f(\sigma)\ \{I\}$



$[\![op]\!]_{eff}(\sigma)$ = if $P(\sigma)$ then $f(\sigma)$ else if...

1. Effector safety: $f(\sigma)$ preserves I when executed in any state satisfying P: $\{I \wedge P\}\ f(\sigma)\ \{I\}$

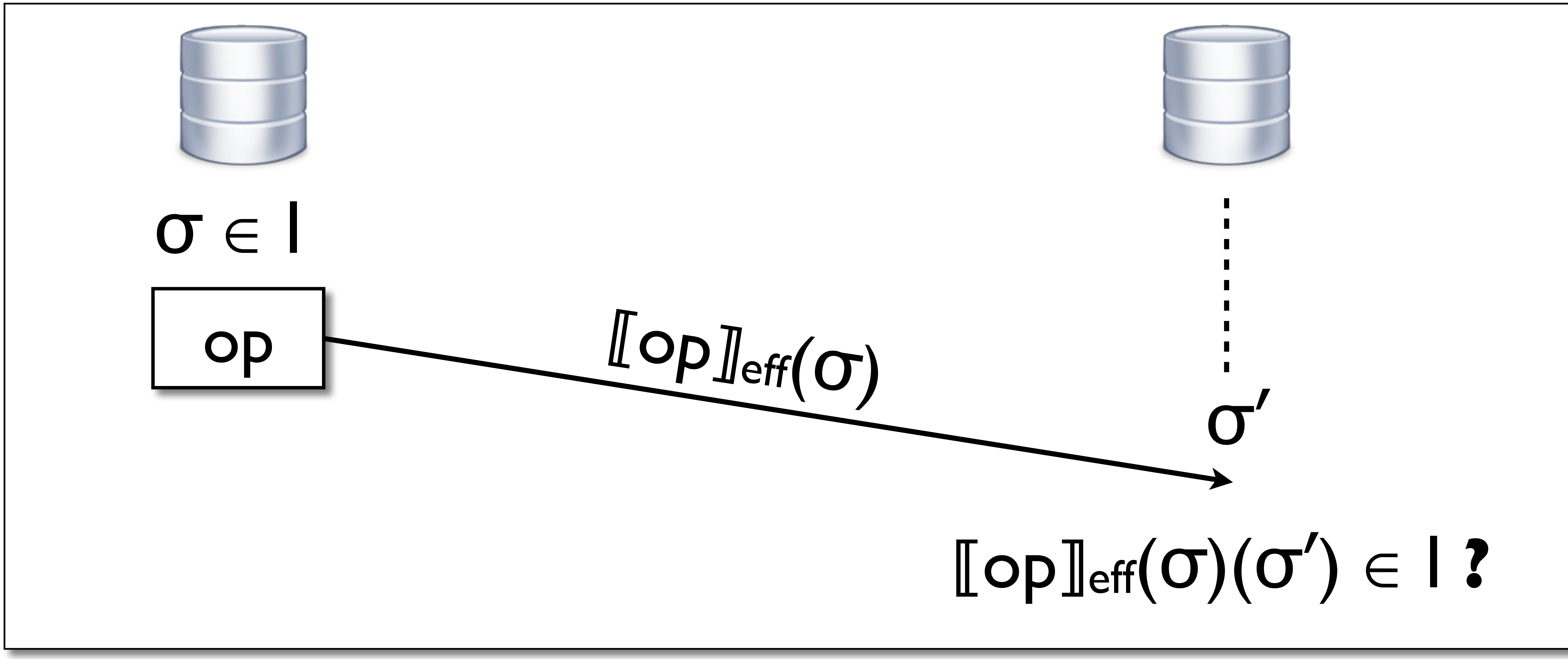
$$\{bal \geq 0 \wedge bal \geq 100\}\ \ bal := bal\text{-}100\ \ \{bal \geq 0\}$$



$[\![op]\!]_{eff}(\sigma)$ = if $P(\sigma)$ then $f(\sigma)$ else if…

1. Effector safety: $f(\sigma)$ preserves I when executed in any state satisfying P: $\{I \wedge P\}\ f(\sigma)\ \{I\}$

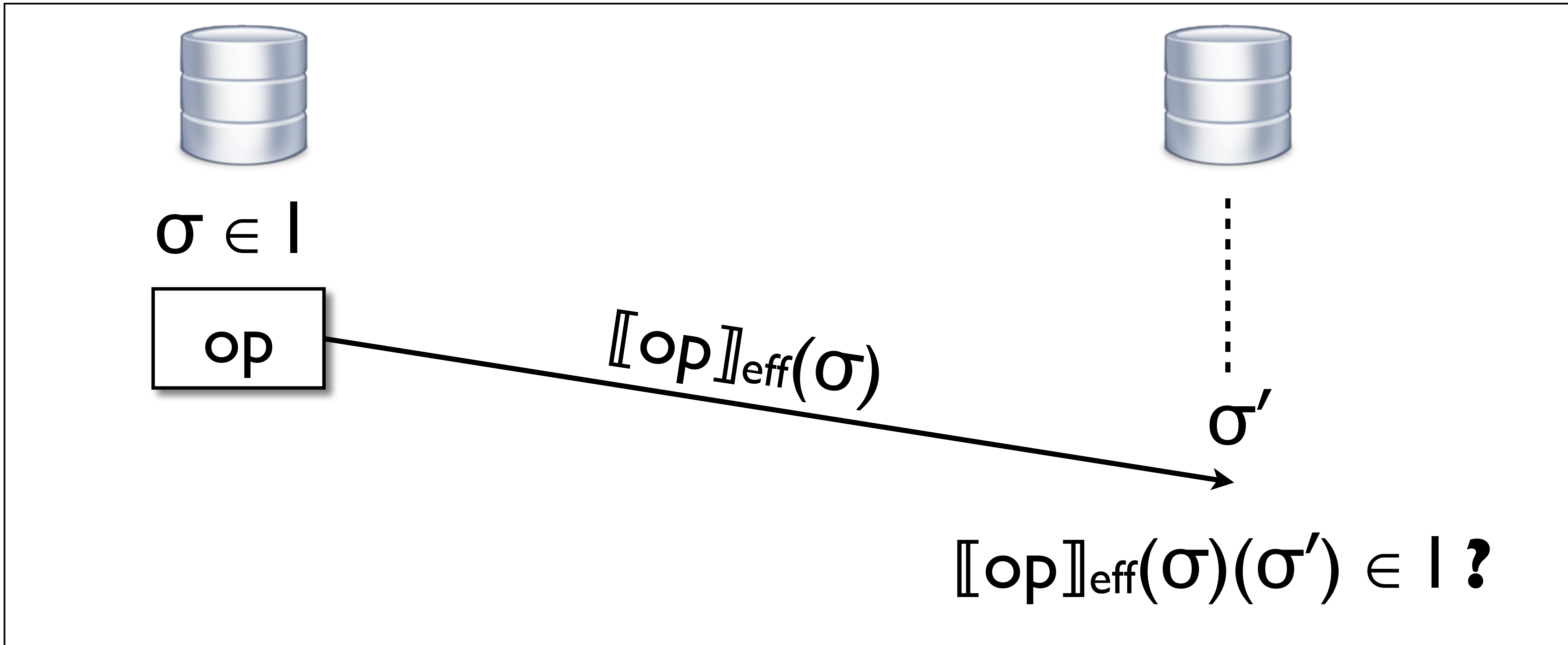
$$\{bal \geq 0 \wedge bal \geq 100\}\ \ bal := bal\text{-}100\ \ \{bal \geq 0\}$$



$[\![op]\!]_{eff}(\sigma)$ = if $P(\sigma)$ then $f(\sigma)$ else if...

1. Effector safety: $f(\sigma)$ preserves I when executed in any state satisfying P: $\{I \wedge P\}\ f(\sigma)\ \{I\}$

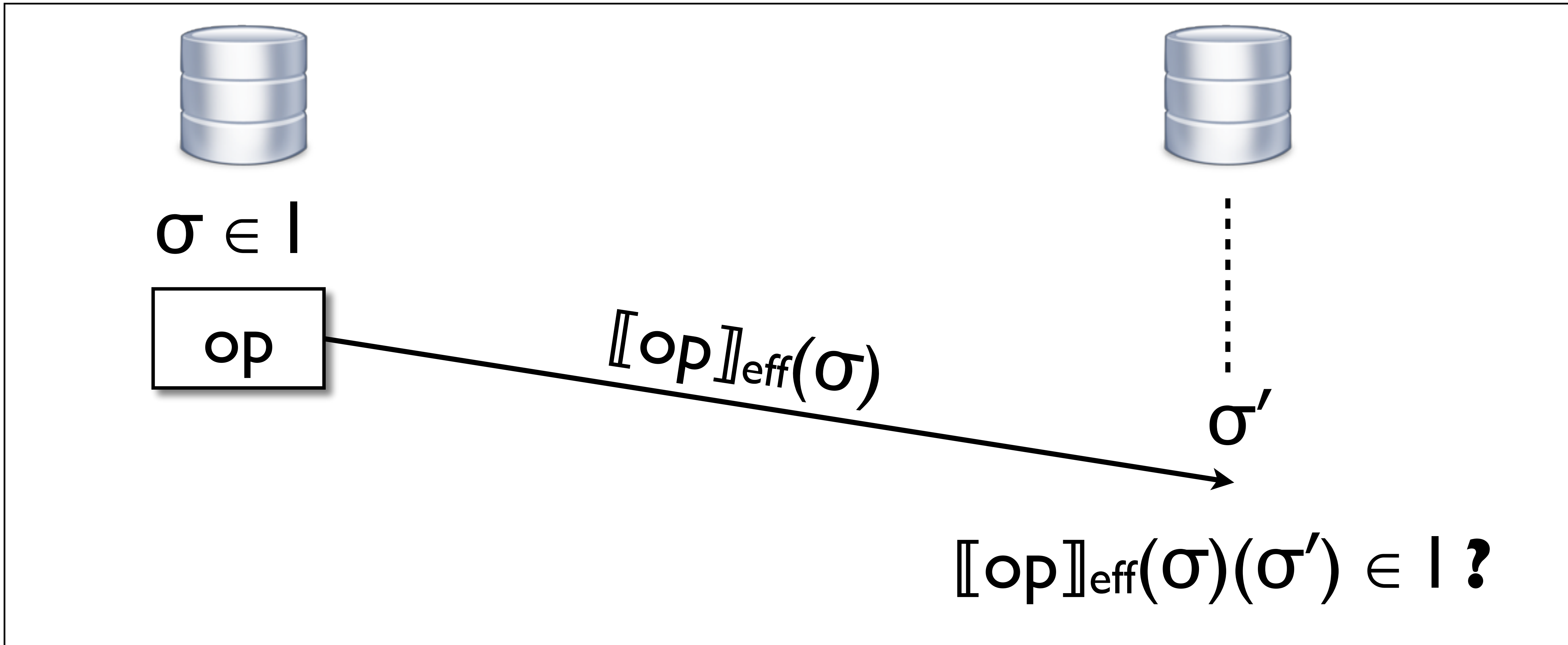
$\{bal \geq 0 \wedge bal \geq 100\}$ bal := bal-100 $\{bal \geq 0\}$



$[\![op]\!]_{eff}(\sigma)$ = if $P(\sigma)$ then $f(\sigma)$ else if...

1. Effector safety: $f(\sigma)$ preserves I when executed in any state satisfying P: $\{I \wedge P\}\ f(\sigma)\ \{I\}$

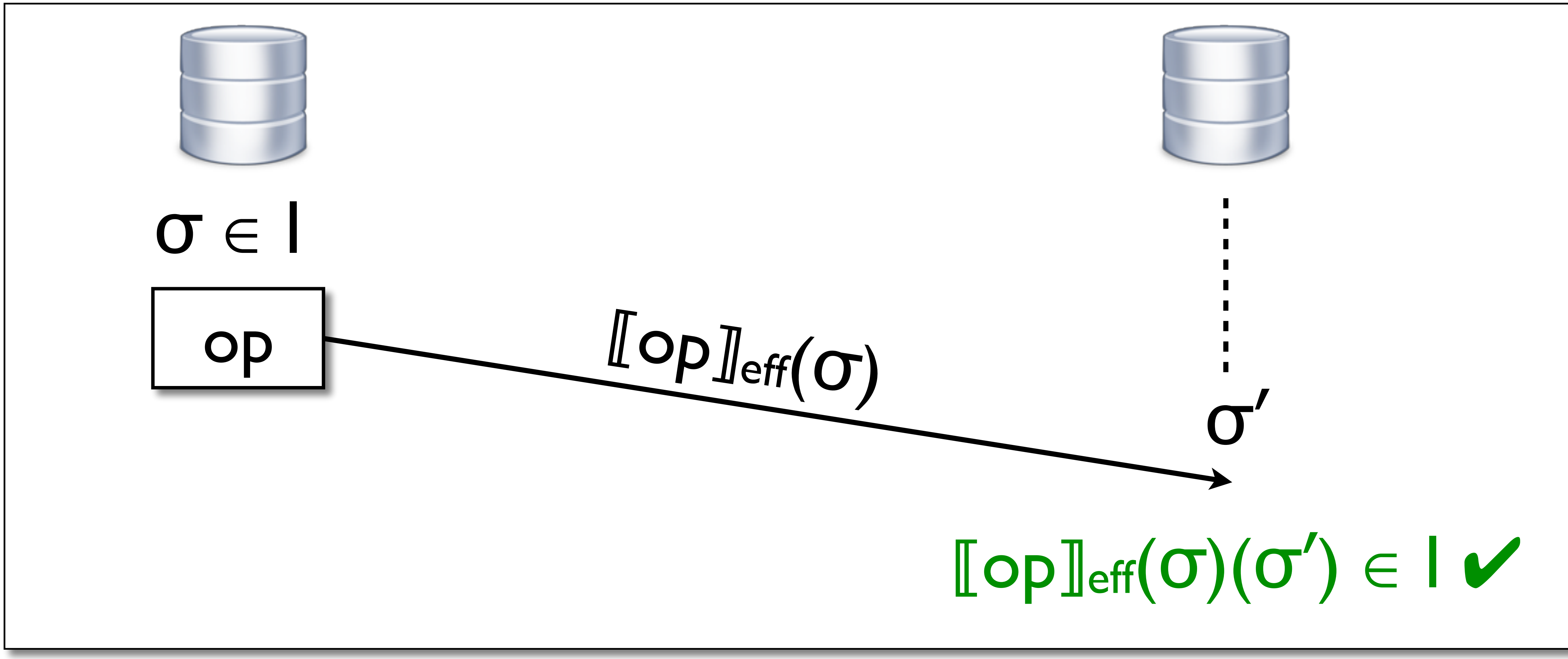
$\{bal \geq 0 \wedge bal \geq 100\}\ \ bal := bal\text{-}100\ \ \{bal \geq 0\}$



$[\![op]\!]_{eff}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if}\ldots$

1. Effector safety: $f(\sigma)$ preserves I when executed in any state satisfying P: $\{I \wedge P\}\ f(\sigma)\ \{I\}$

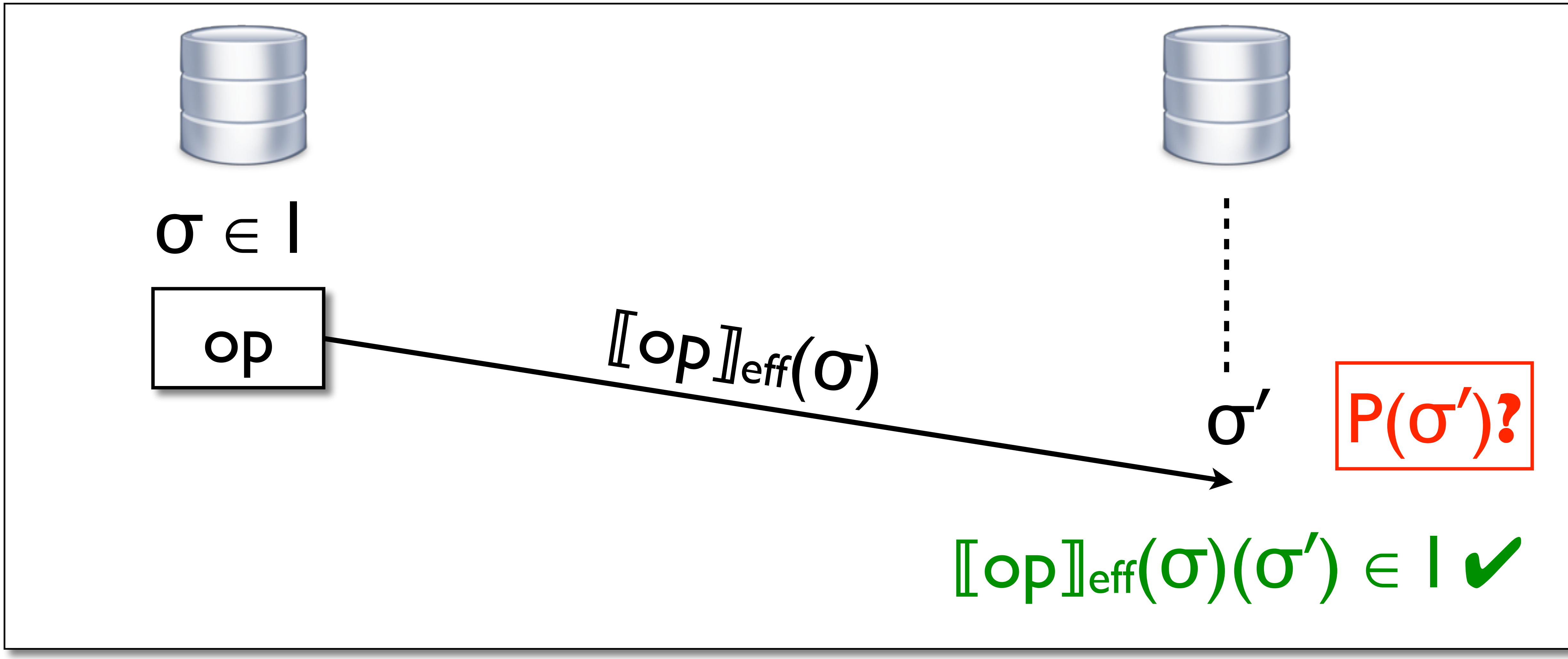
$$\{bal \geq 0 \wedge bal \geq 100\}\ \ bal := bal\text{-}100\ \ \{bal \geq 0\}$$



$[\![op]\!]_{eff}(\sigma)$ = if $P(\sigma)$ then $f(\sigma)$ else if...

1. Effector safety: $f(\sigma)$ preserves I when executed in any state satisfying P: $\{I \wedge P\}\ f(\sigma)\ \{I\}$

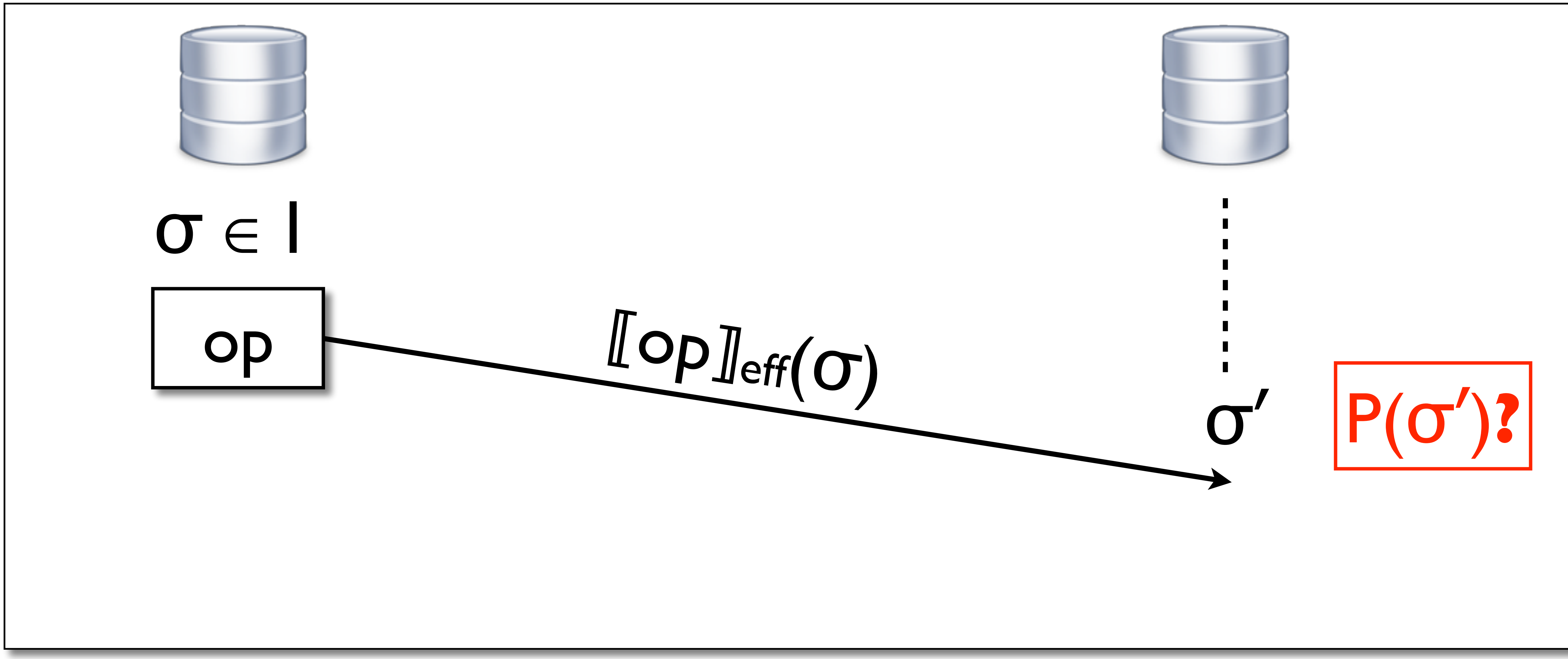
$$\{bal \geq 0 \wedge bal \geq 100\}\ \ bal := bal\text{-}100\ \ \{bal \geq 0\}$$



$[\![op]\!]_{eff}(\sigma)$ = if $P(\sigma)$ then $f(\sigma)$ else if…

1. Effector safety: $f(\sigma)$ preserves I when executed in any state satisfying P: $\{I \wedge P\}\ f(\sigma)\ \{I\}$

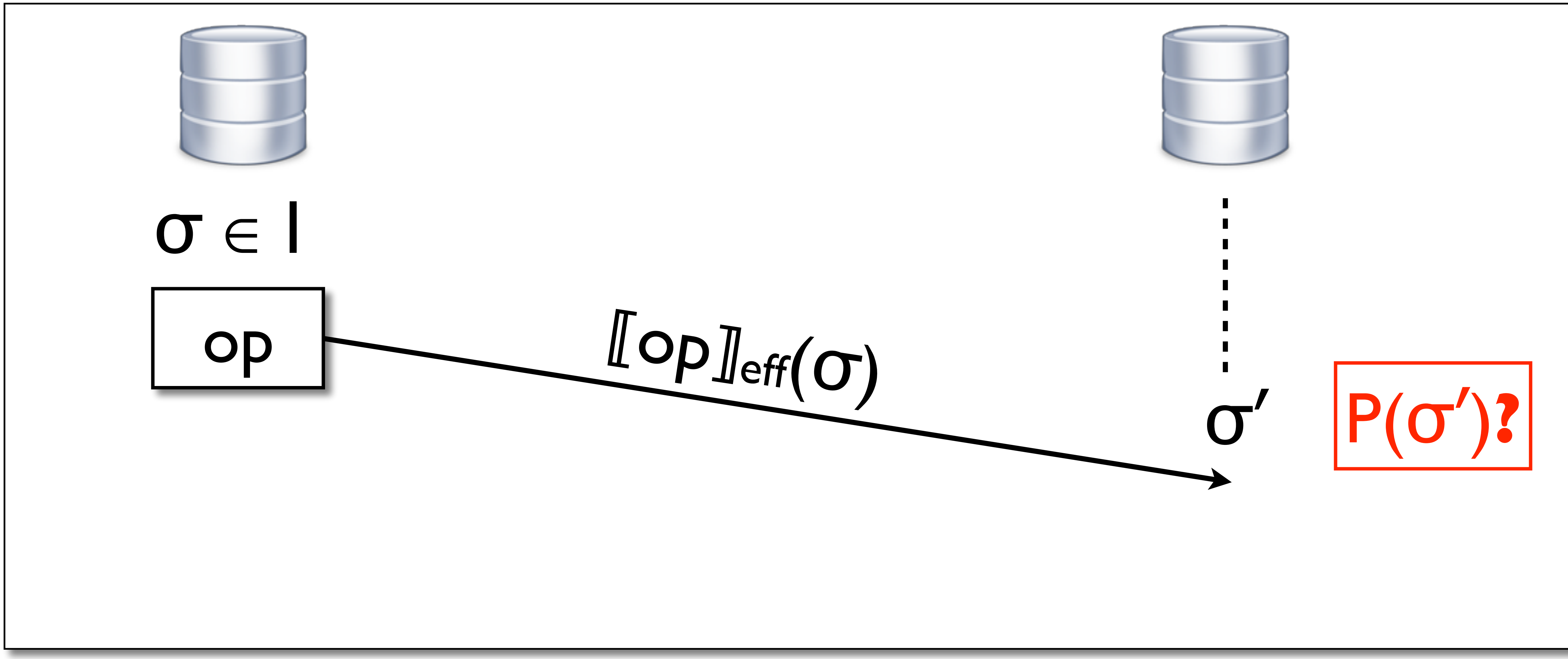
$$\{bal \geq 0 \wedge bal \geq 100\}\ \ bal := bal\text{-}100\ \ \{bal \geq 0\}$$

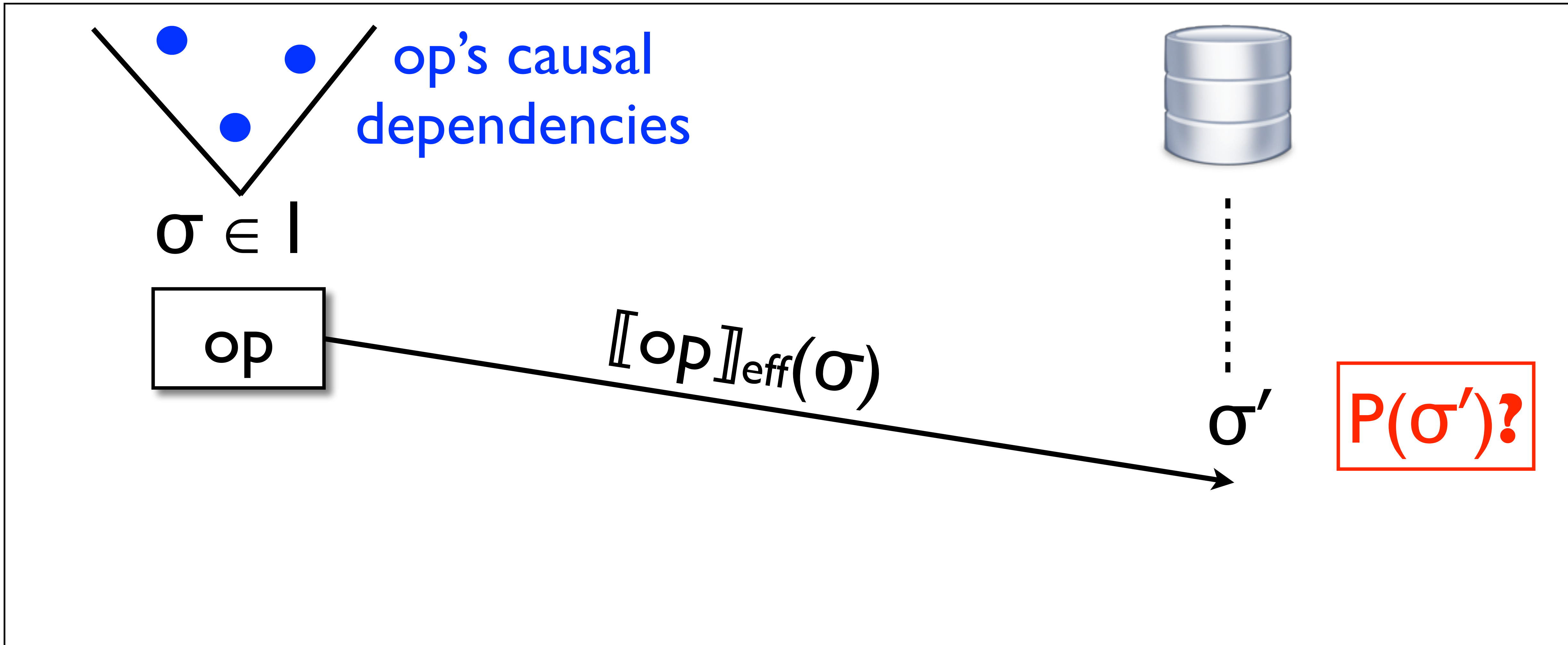


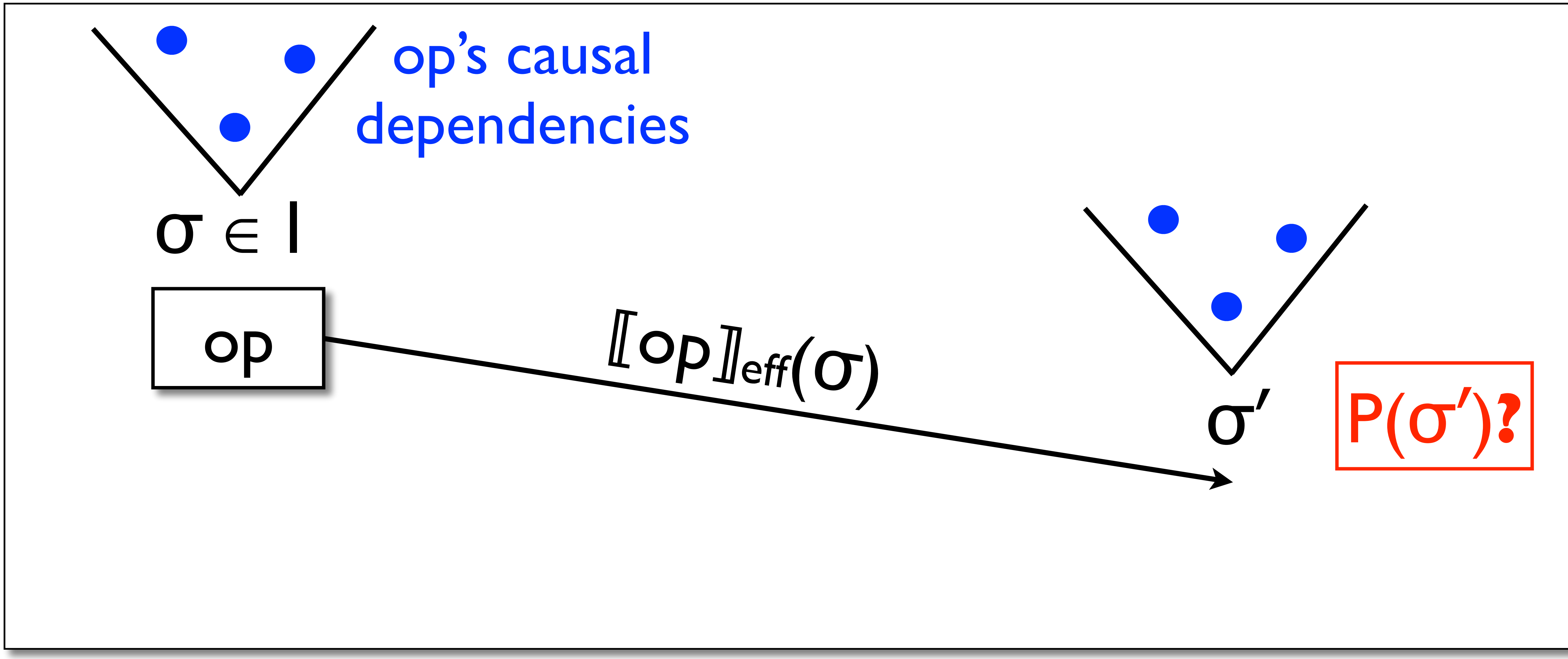
$[\![op]\!]_{eff}(\sigma)$ = if $P(\sigma)$ then $f(\sigma)$ else if...

1. Effector safety: $f(\sigma)$ preserves I when executed in any state satisfying P: $\{I \wedge P\}\ f(\sigma)\ \{I\}$

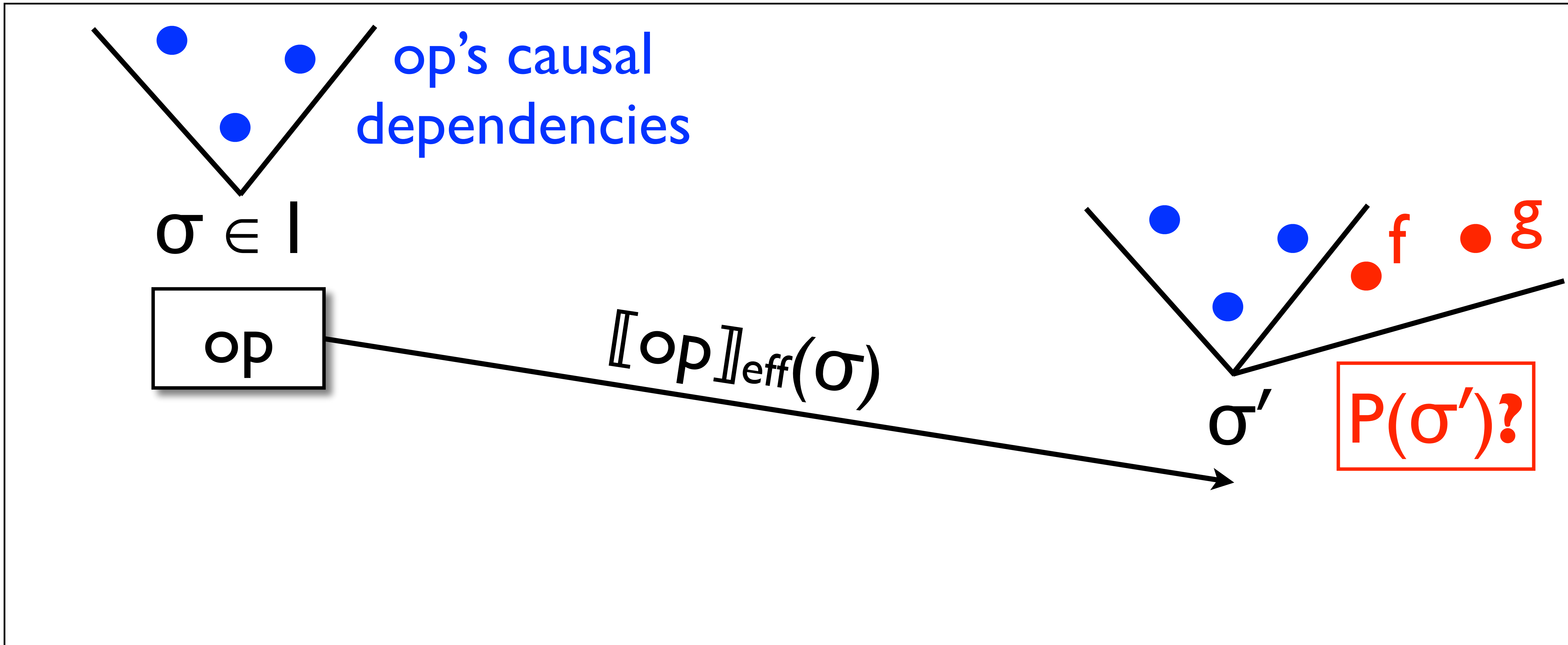
2. Precondition stability: P will hold when $f(\sigma)$ is applied at any replica


σ ∈ I
op
⟦op⟧eff(σ)
σ′
P(σ′)?

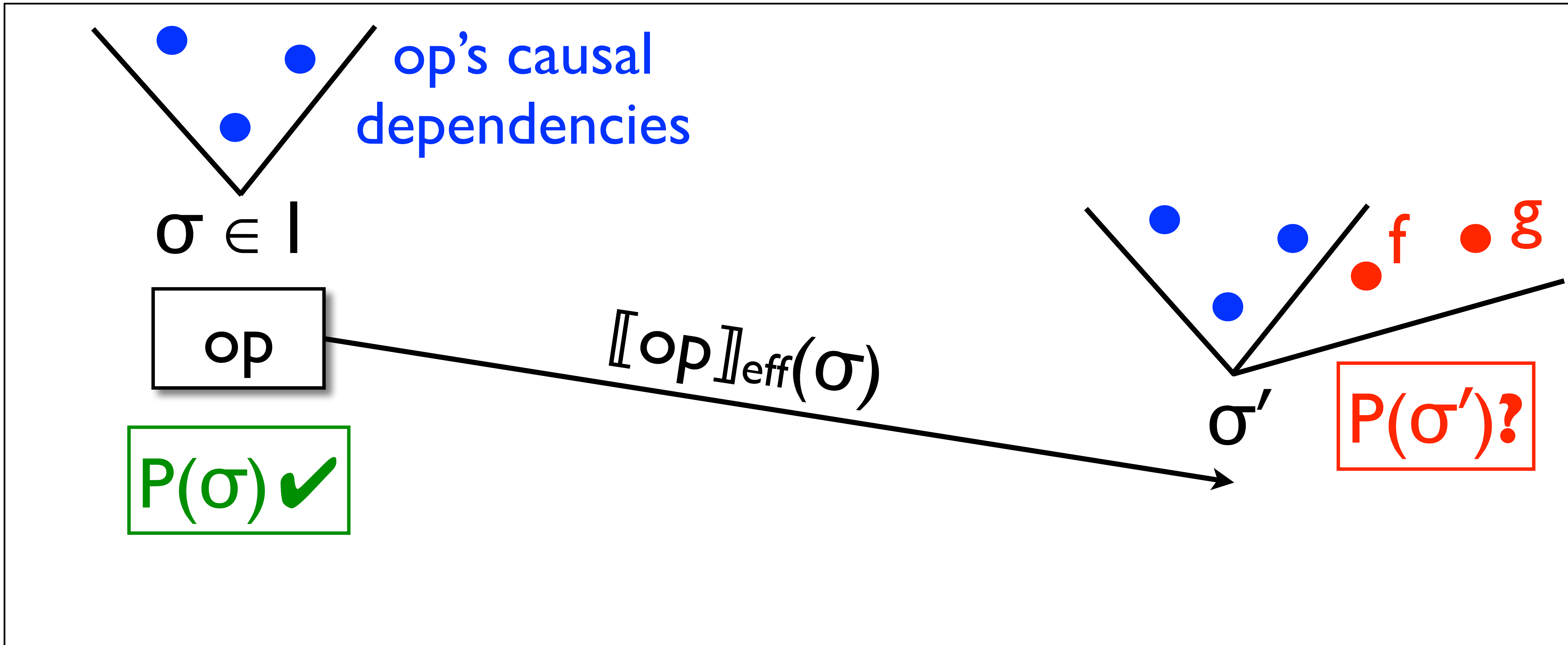

op's causal dependencies
σ ∈ I
op
⟦op⟧eff(σ)
σ′
P(σ′)?



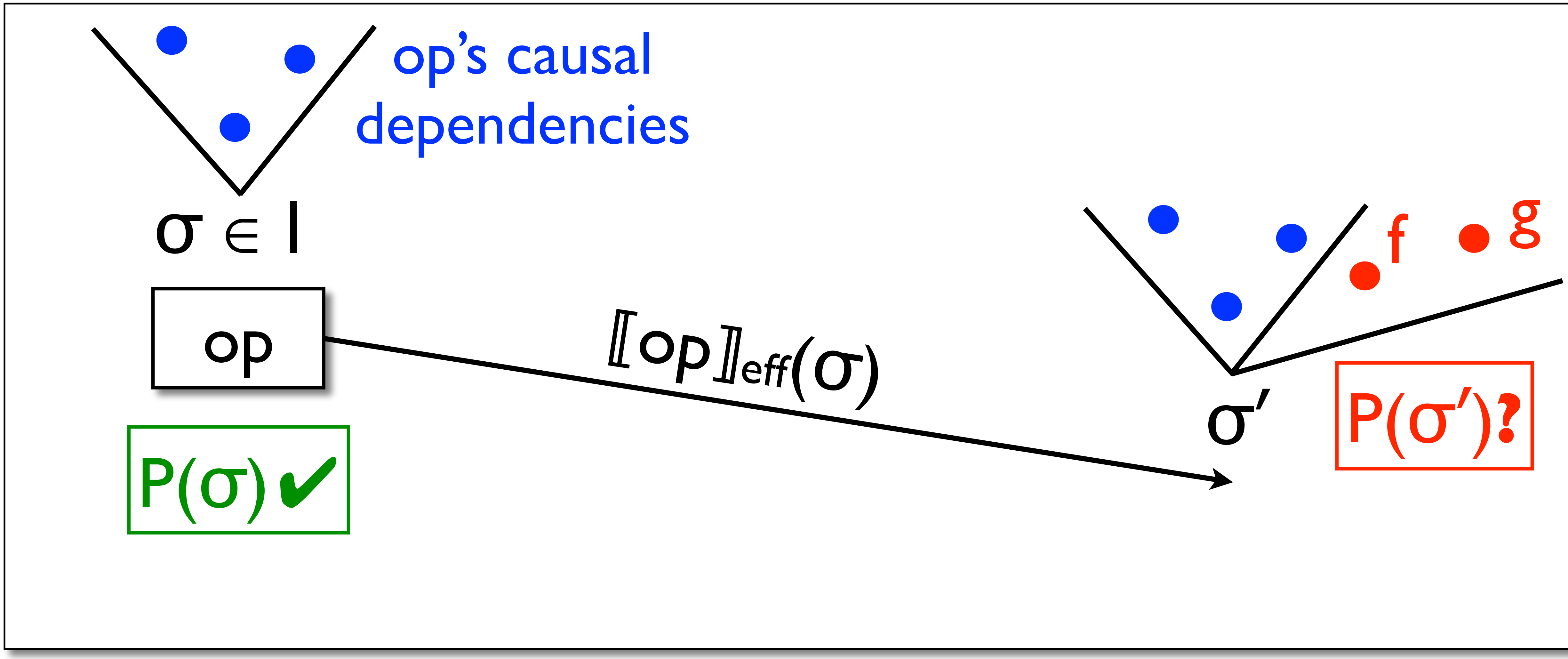
- Causal consistency ➜ receive op's causal dependencies before receiving op



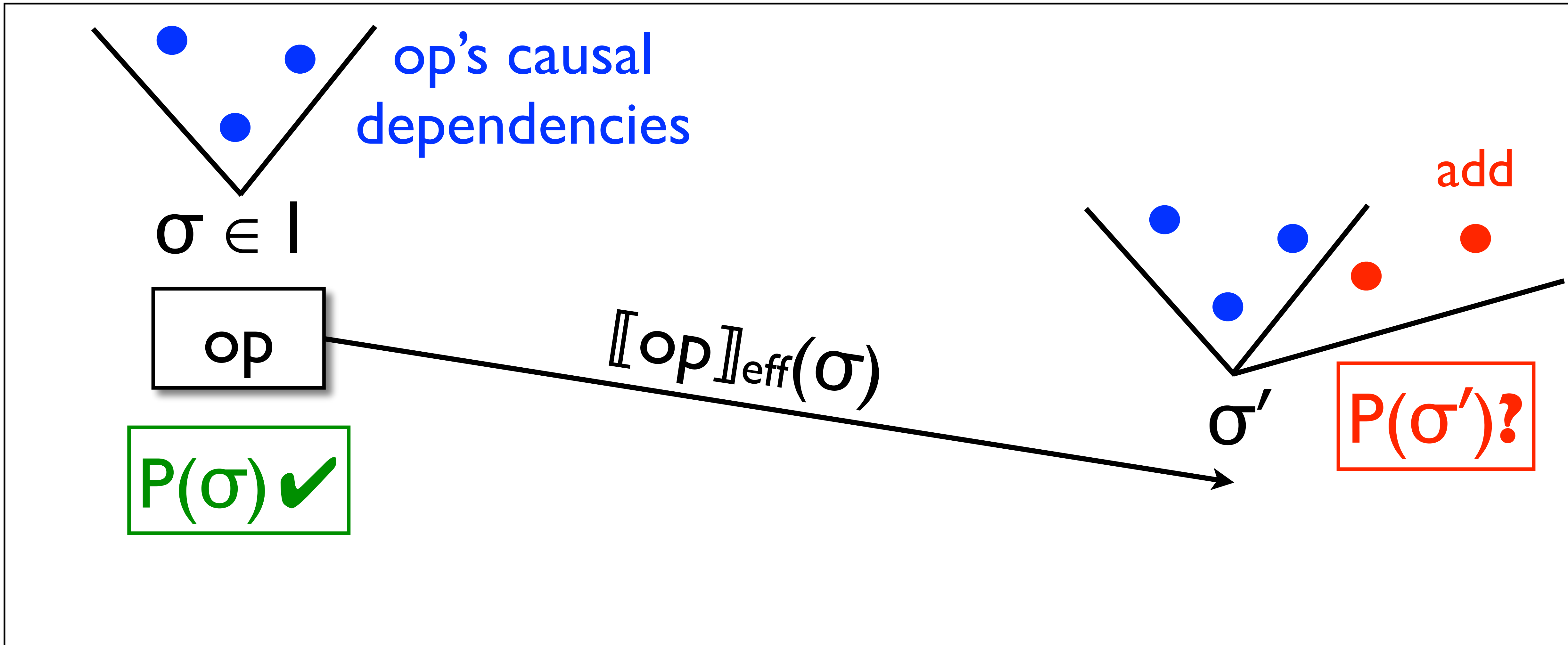
- Causal consistency ➜ receive op's causal dependencies before receiving op
- But can have additional effectors of operations concurrent with op: f, g, ...
- Effectors commute, so $\sigma' = (f; g; \ldots)(\sigma)$



- Causal consistency ➜ receive op's causal dependencies before receiving op
- But can have additional effectors of operations concurrent with op: f, g, ...
- Effectors commute, so $\sigma' = (f; g; \ldots)(\sigma)$

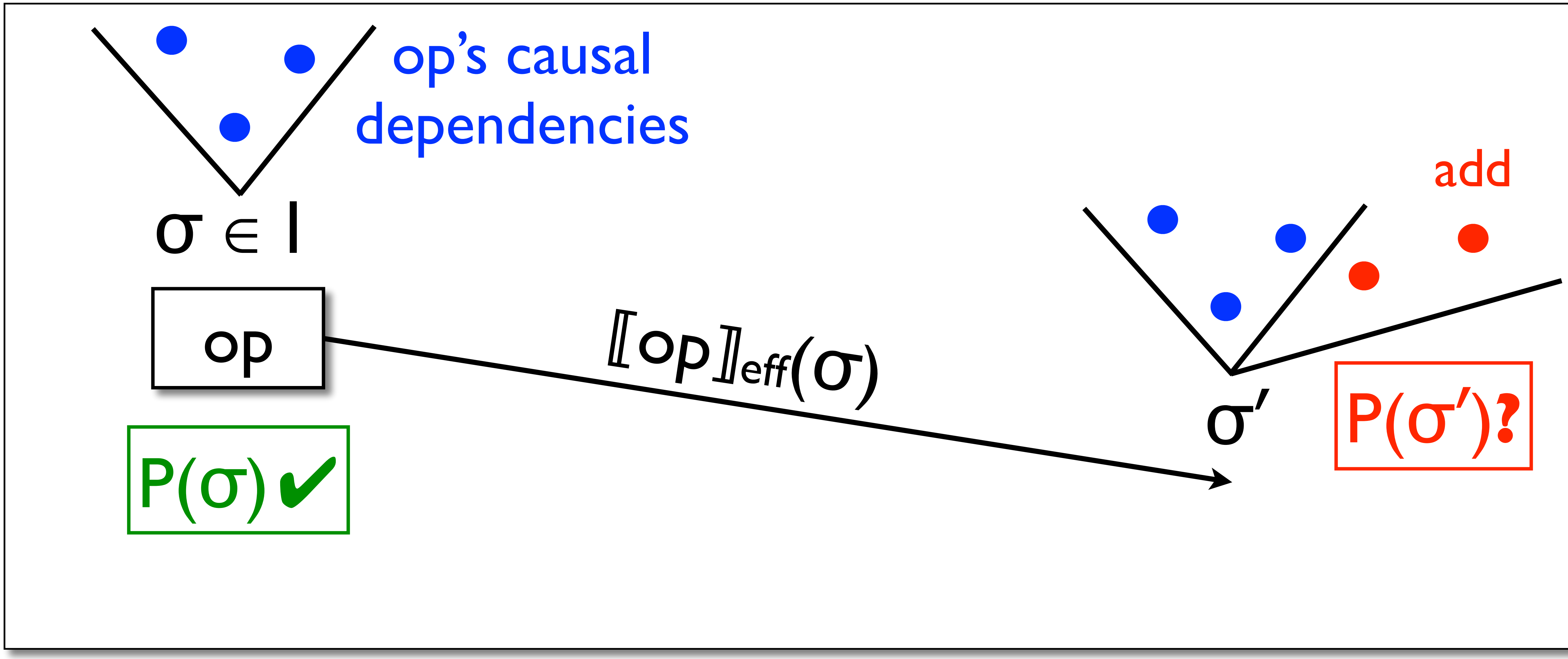


Precondition stability: P is preserved by any effector f of any operation: {P} f {P}



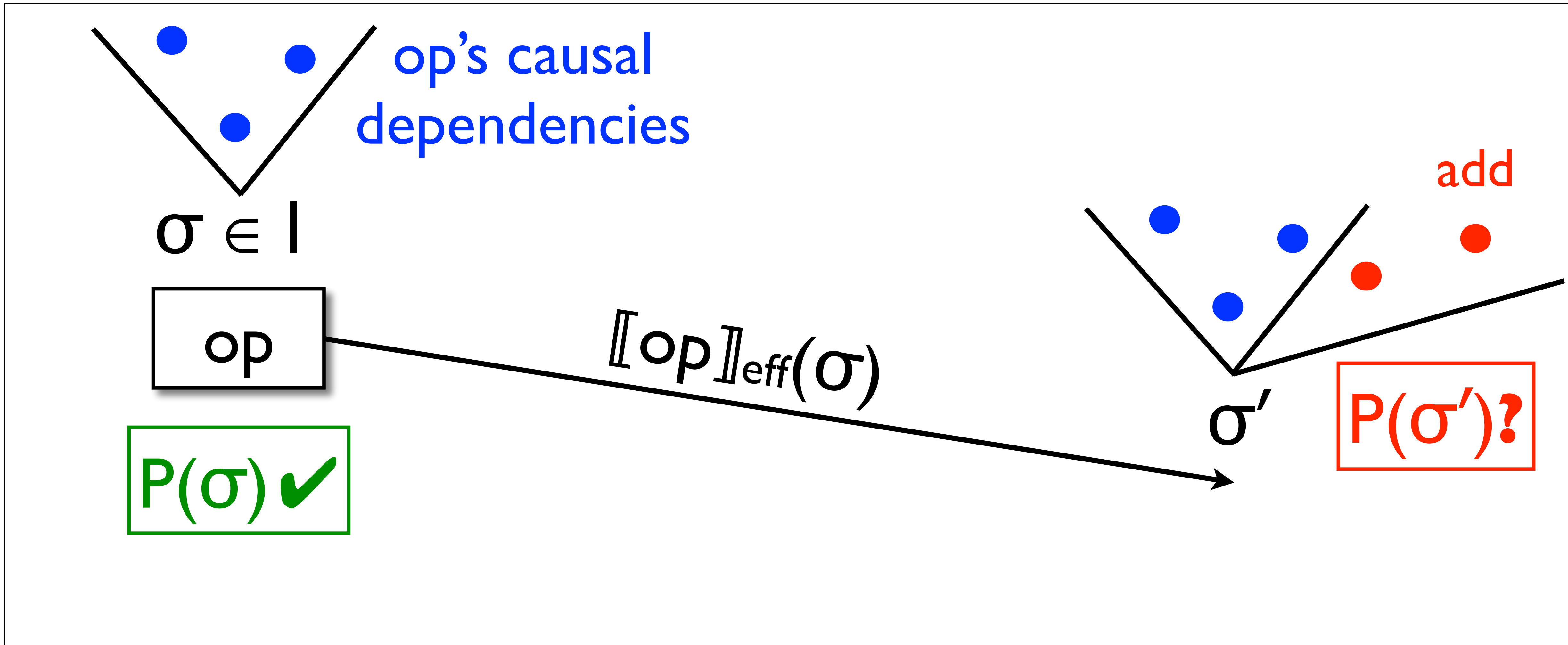
Precondition stability: P is preserved by any effector f of any operation: {P} f {P}

{bal ≥ 100} bal := bal+100 {bal ≥ 100}



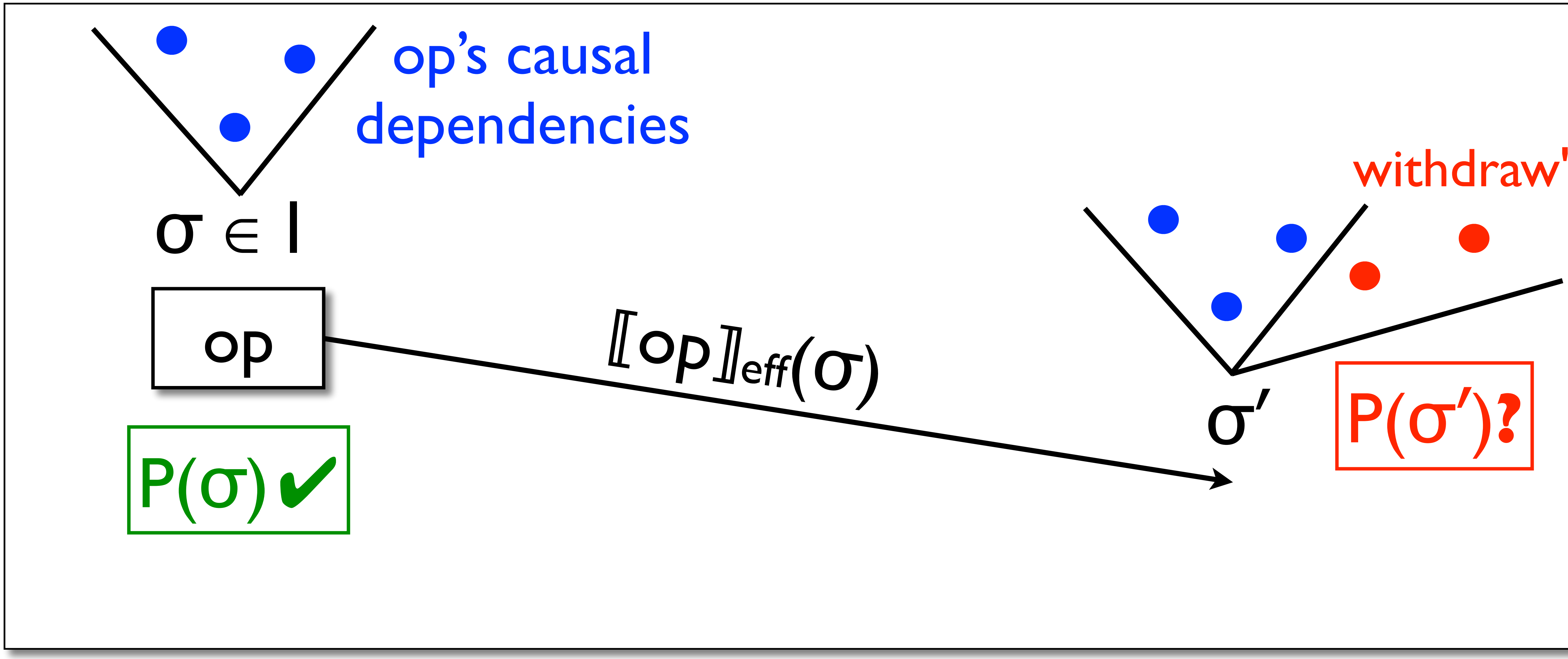
Precondition stability: P is preserved by any effector f of any operation: {P} f {P}

{bal ≥ 100} bal := bal+100 {bal ≥ 100}



Precondition stability: P is preserved by any effector f of any operation: {P} f {P}

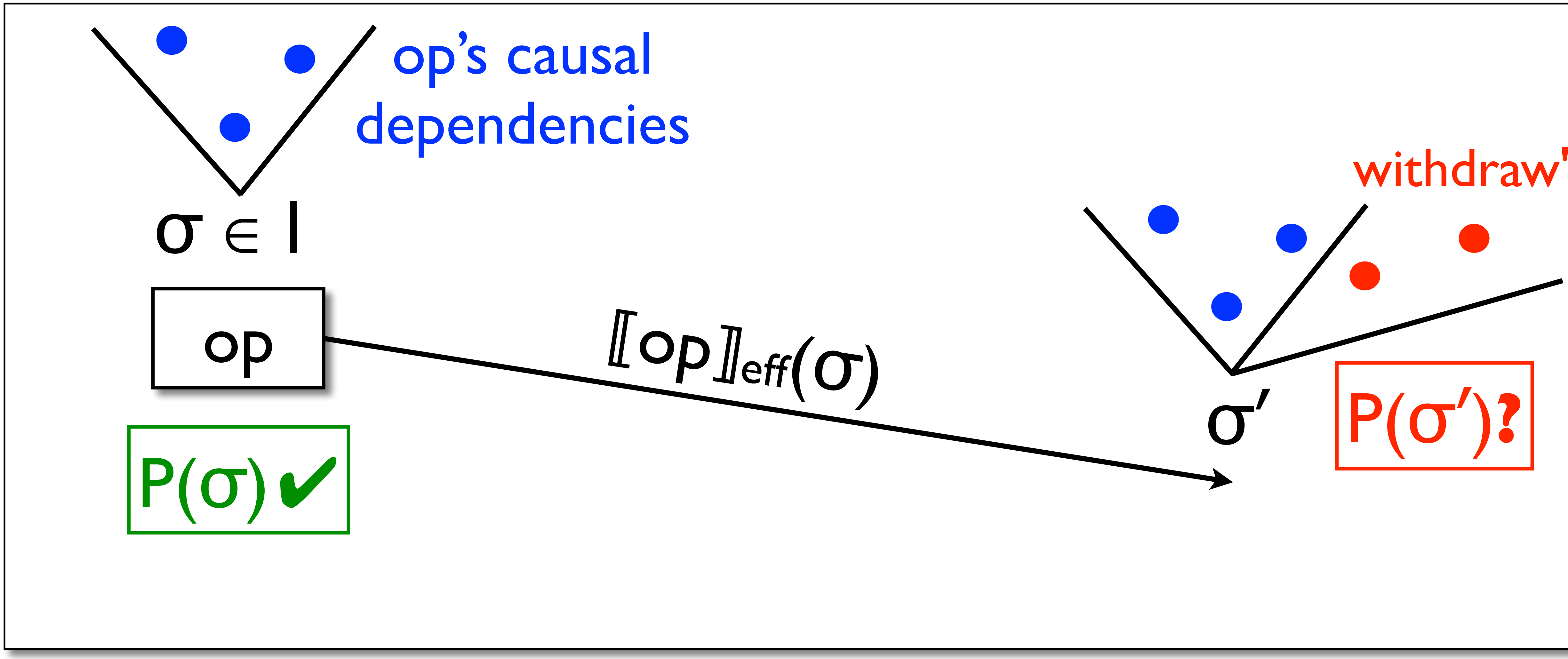
{bal ≥ 100} bal := bal+100 {bal ≥ 100} ✔



Precondition stability: P is preserved by any effector f of any operation: {P} f {P}

{bal ≥ 100} bal := bal+100 {bal ≥ 100} ✔

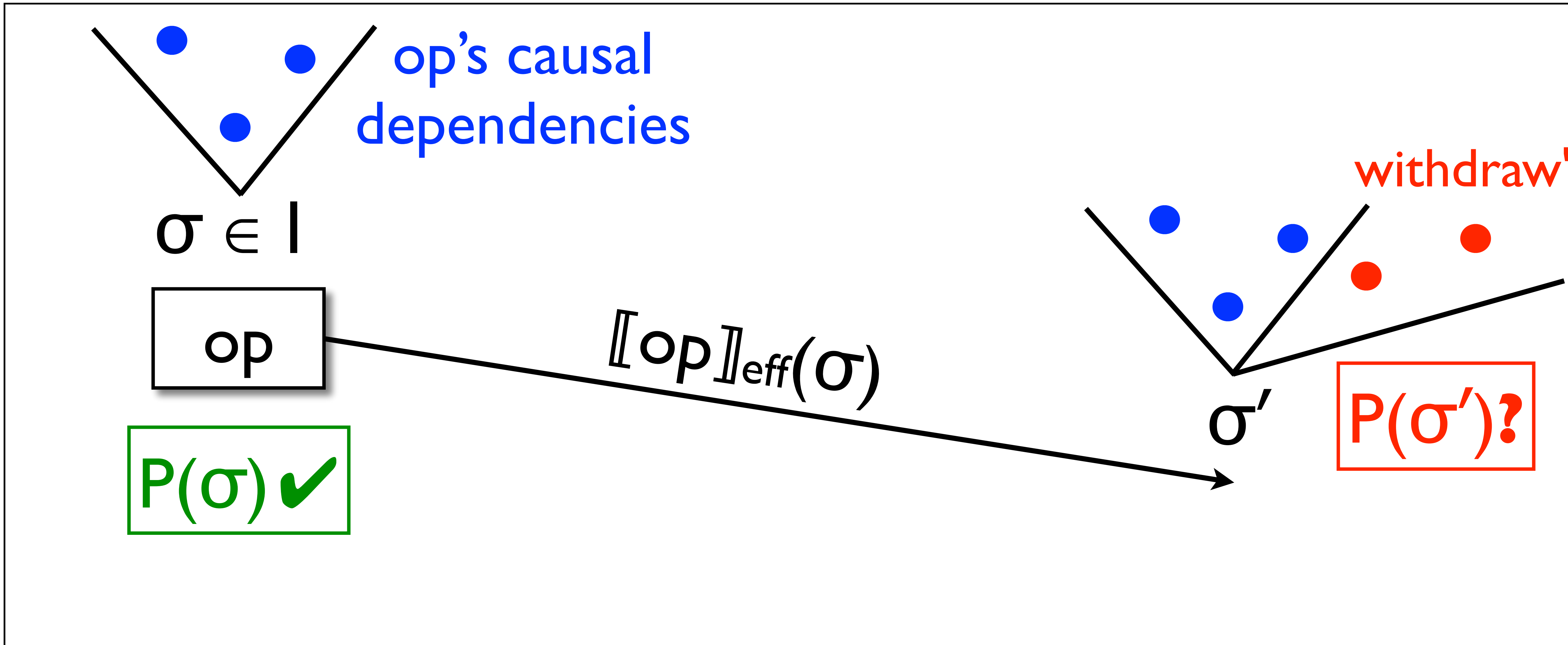
{bal ≥ 100} bal := bal-100 {bal ≥ 100}



Precondition stability: P is preserved by any effector f of any operation: {P} f {P}

{bal ≥ 100} bal := bal+100 {bal ≥ 100} ✔

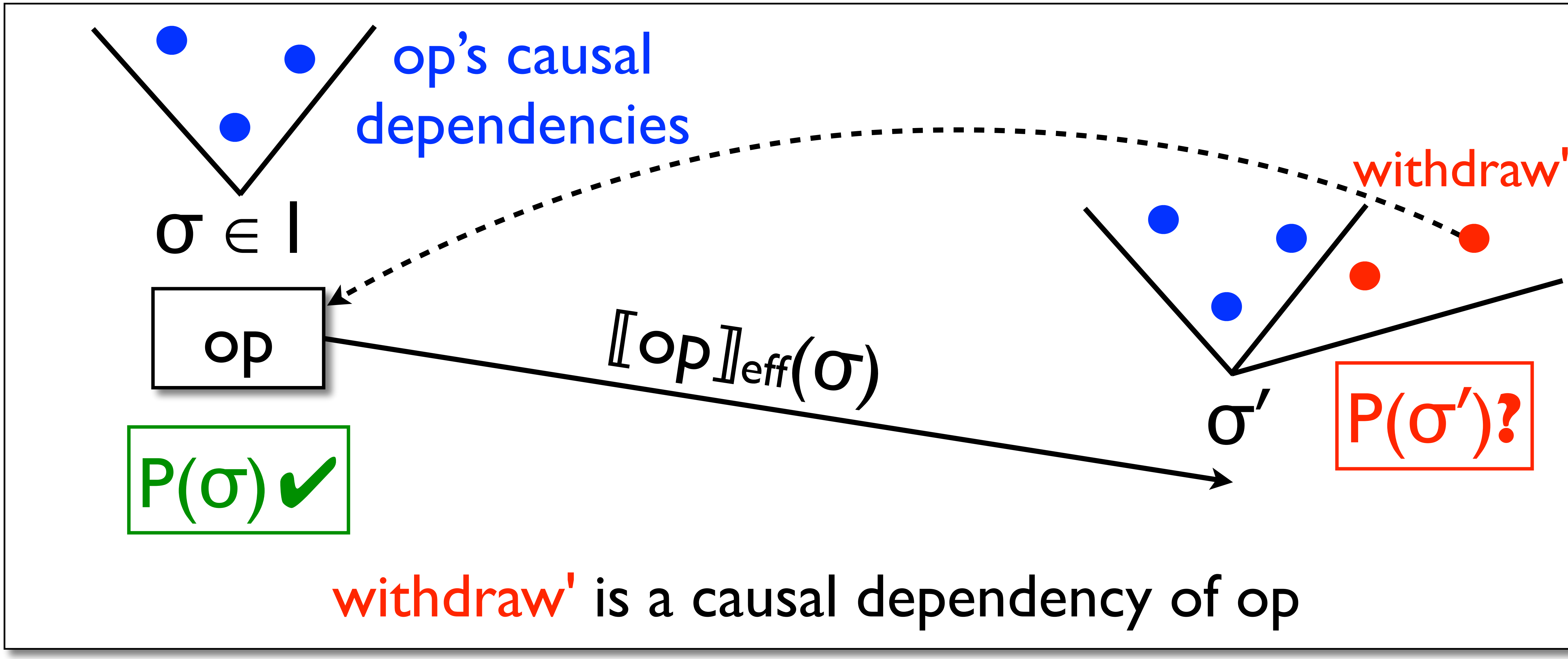
{bal ≥ 100} bal := bal-100 {bal ≥ 100}



Precondition stability: P is preserved by any effector f of any operation: {P} f {P}

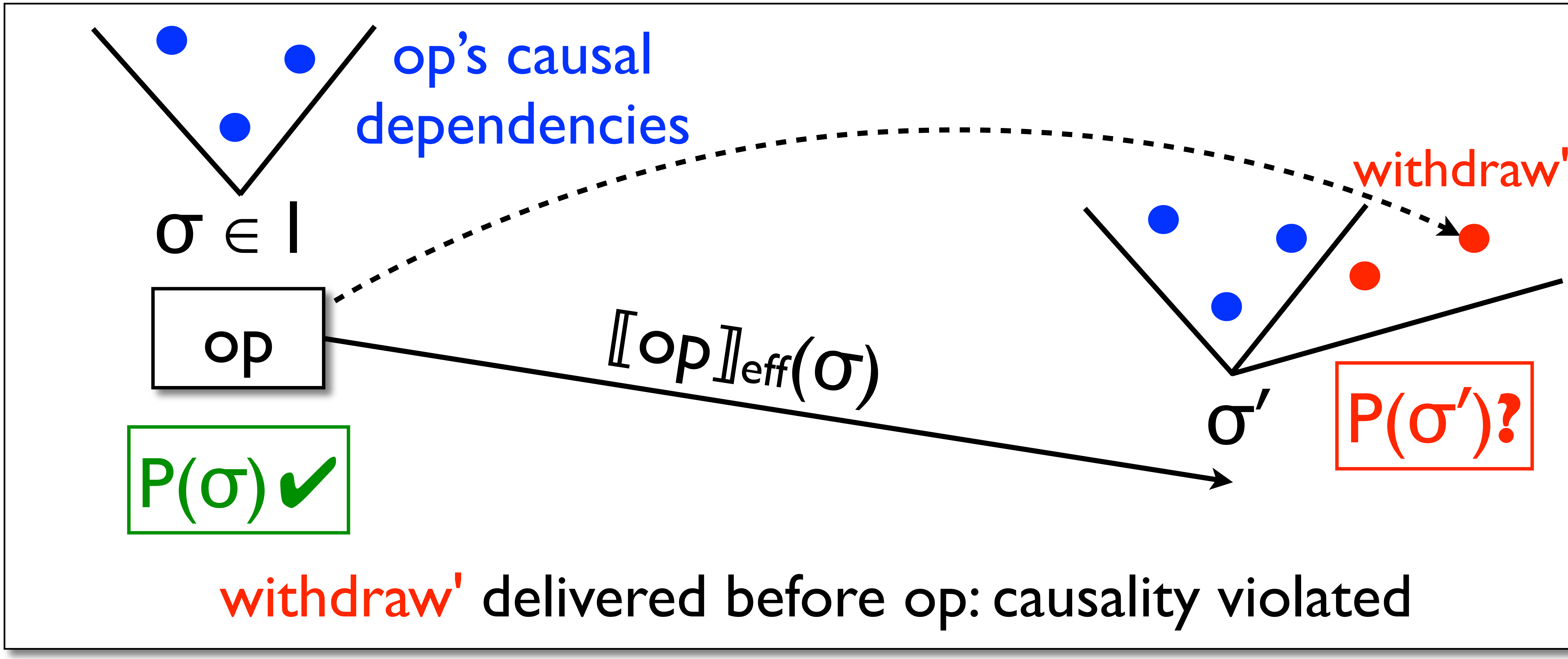
{bal ≥ 100} bal := bal+100 {bal ≥ 100} ✔

{bal ≥ 100} bal := bal-100 {bal ≥ 100} ✘



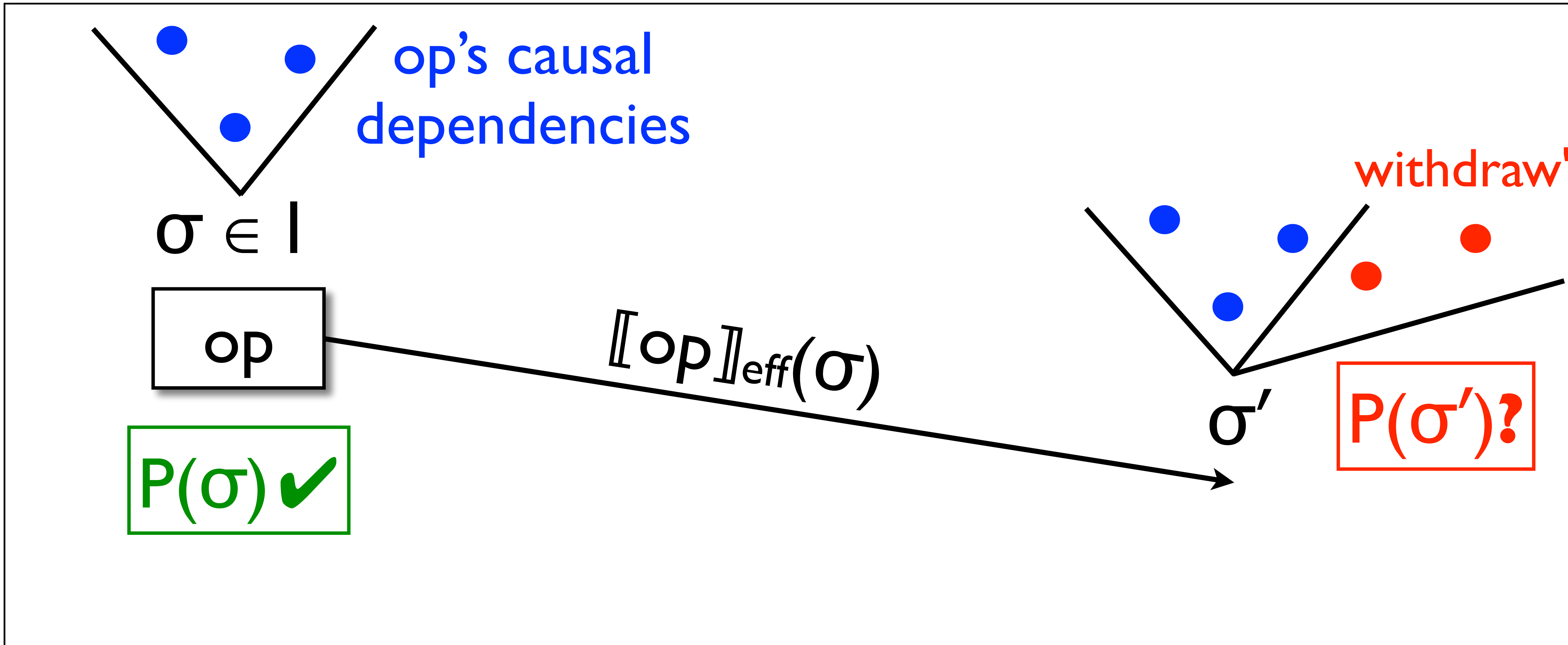
Precondition stability: P is preserved by any effector f of any non-conflicting operation: $\{P\}\ f\ \{P\}$

withdraw ⋈ withdraw; ¬(add ⋈ withdraw) ✔



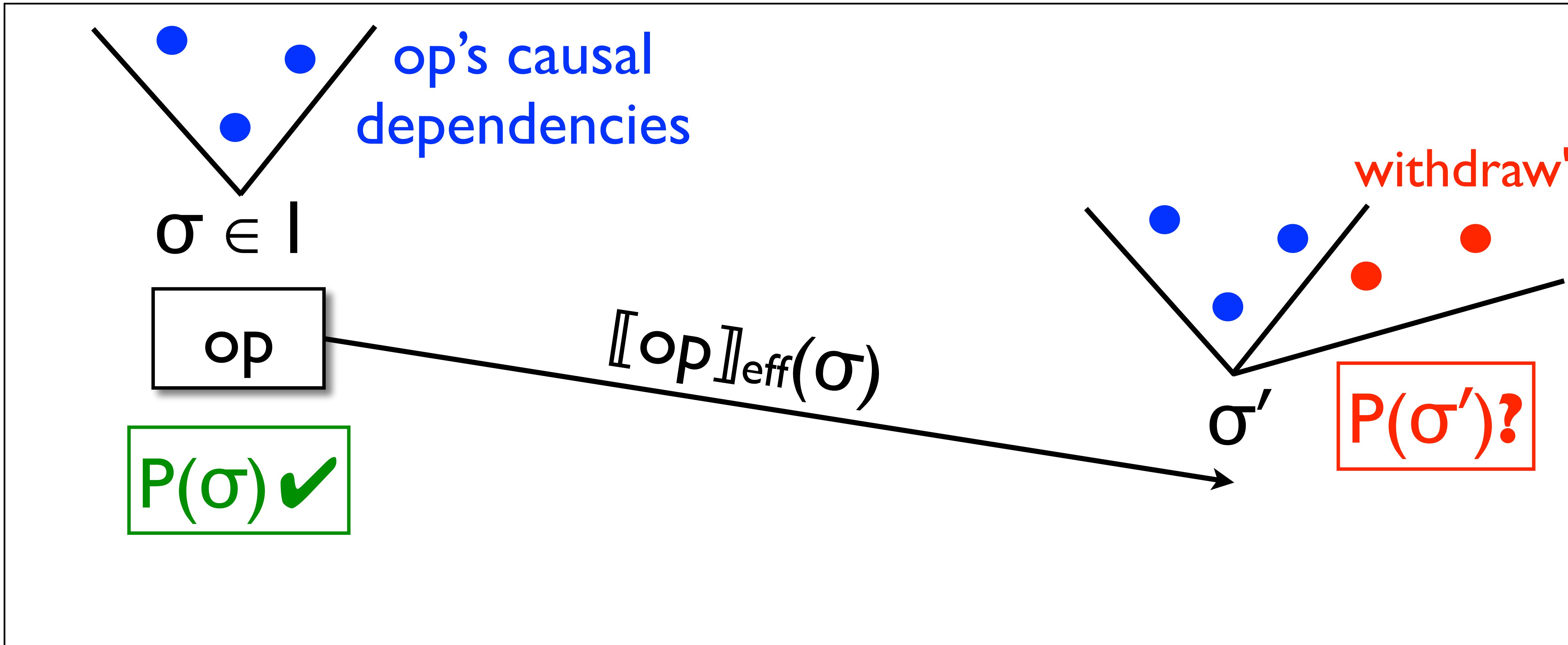
Precondition stability: P is preserved by any effector f of any non-conflicting operation: {P} f {P}

withdraw ⋈ withdraw; ¬(add ⋈ withdraw) ✔



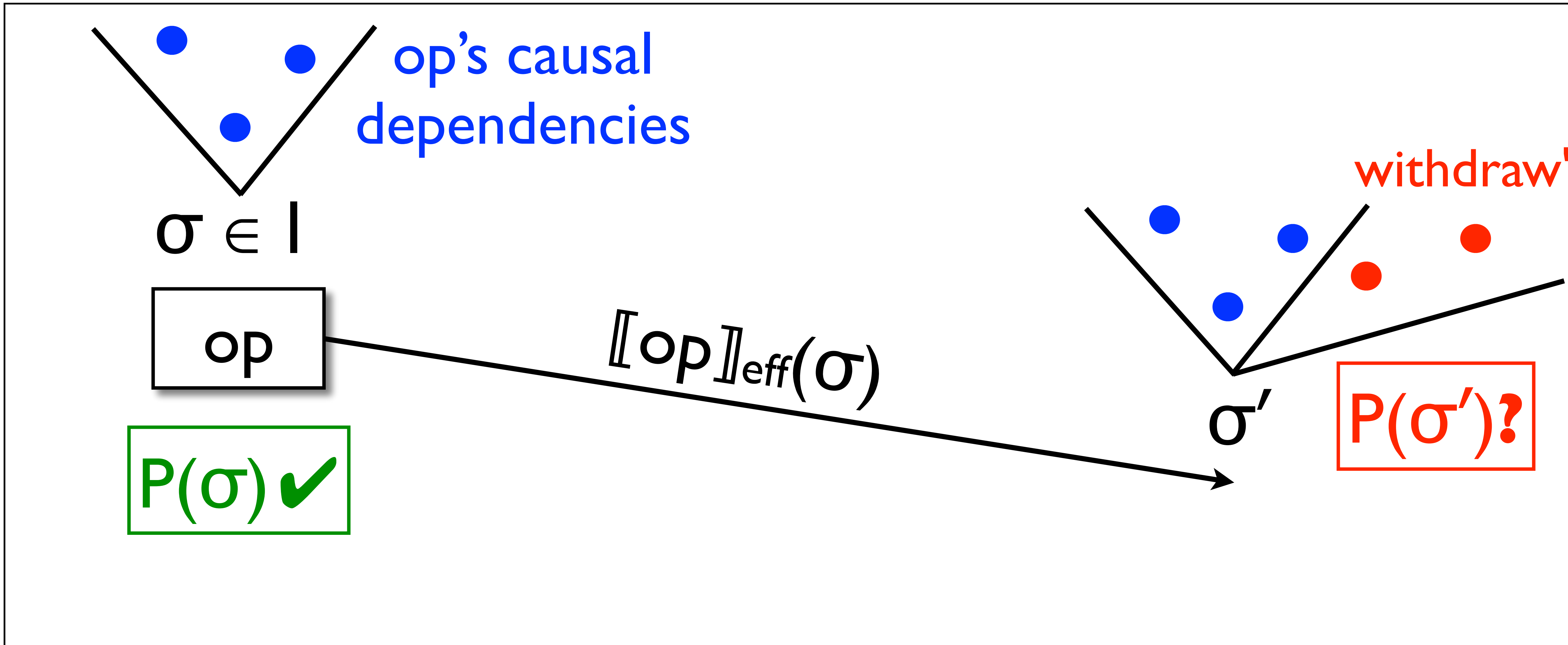
Precondition stability: P is preserved by any effector f of any non-conflicting operation: {P} f {P}

withdraw ⋈ withdraw; ¬(add ⋈ withdraw) ✔



Precondition stability: P is preserved by any effector f of any non-conflicting operation: {P} f {P}

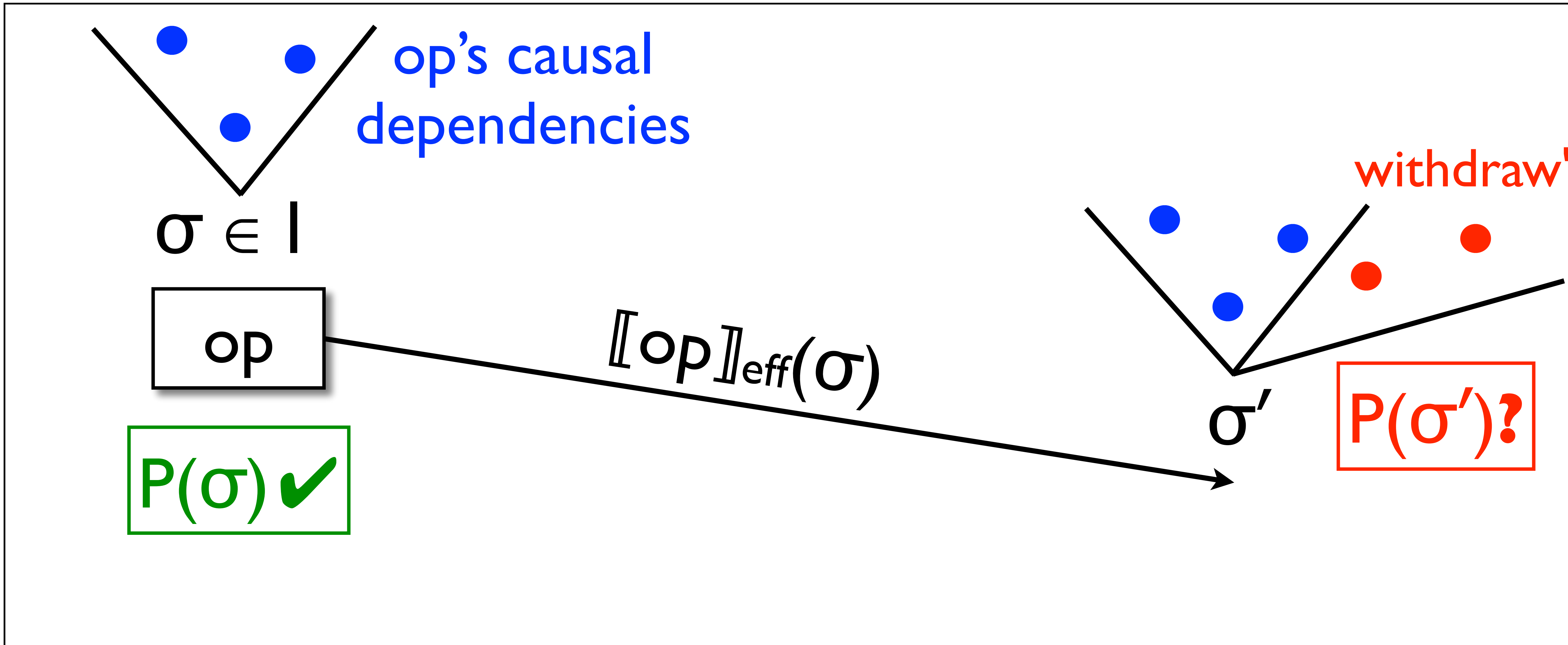
Only requires checking each pair of operations: no exponential explosion!



Can infer the conflict relation $\bowtie$: $op_1 \bowtie op_2$ if the precondition of op_1 unstable under the effector of op_2

Pre of withdraw under effector of add:

$\{bal \geq 100\}$ $bal := bal+100$ $\{bal \geq 100\}$ ✔, no $\bowtie$



Can infer the conflict relation ⋈: op_1 ⋈ op_2 if the precondition of op_1 unstable under the effector of op_2

Pre of withdraw under effector of withdraw:

{bal ≥ 100} bal := bal-100 {bal ≥ 100} ✗, need ⋈

Correct Eventual Consistency Tool

- Developed by Sreeja Nair (UPMC, Paris)
- Model application in a domain-specific language, including replicated data type libraries
- Model compiled into a Boogie program encoding the conditions of the proof rule
- Discharged using SMT
- Automatically infers a conflict relation

https://github.com/LightKone/correct-eventual-consistency-tool

Demo

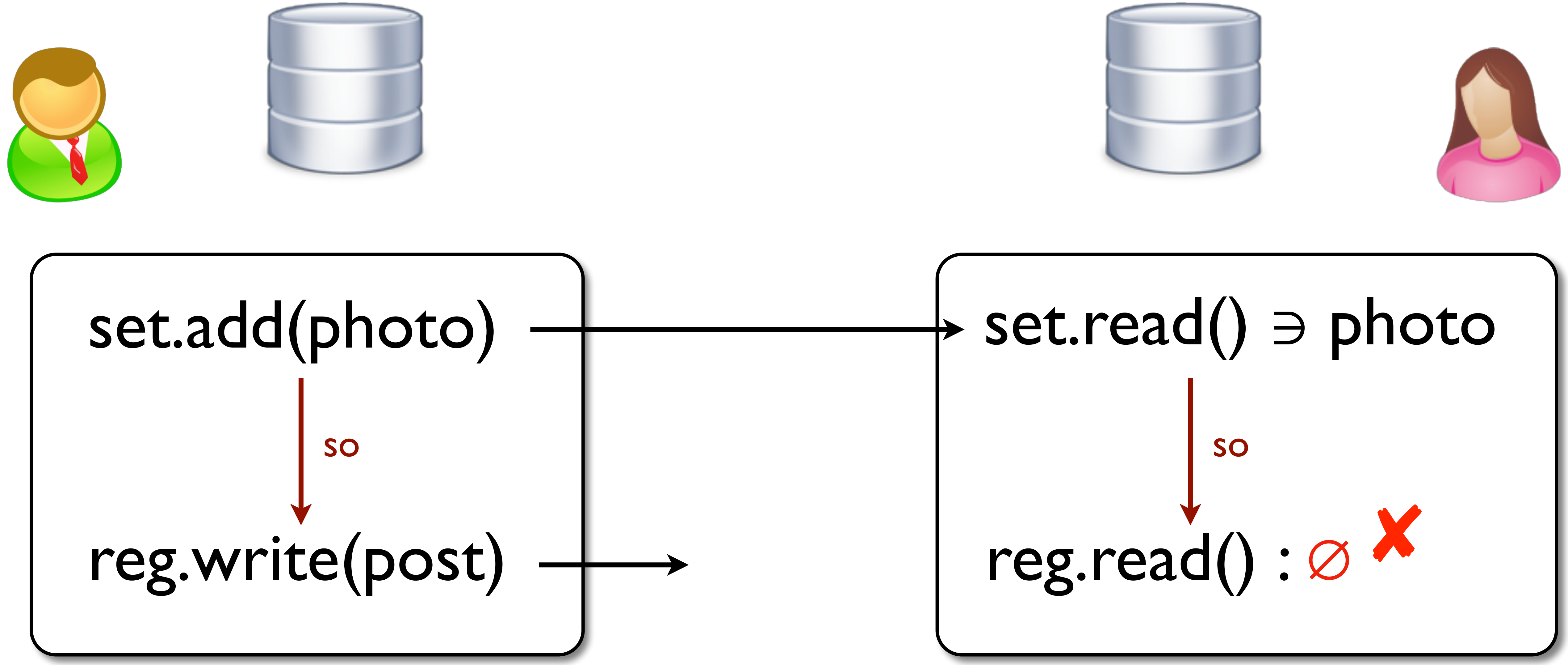
Transactions

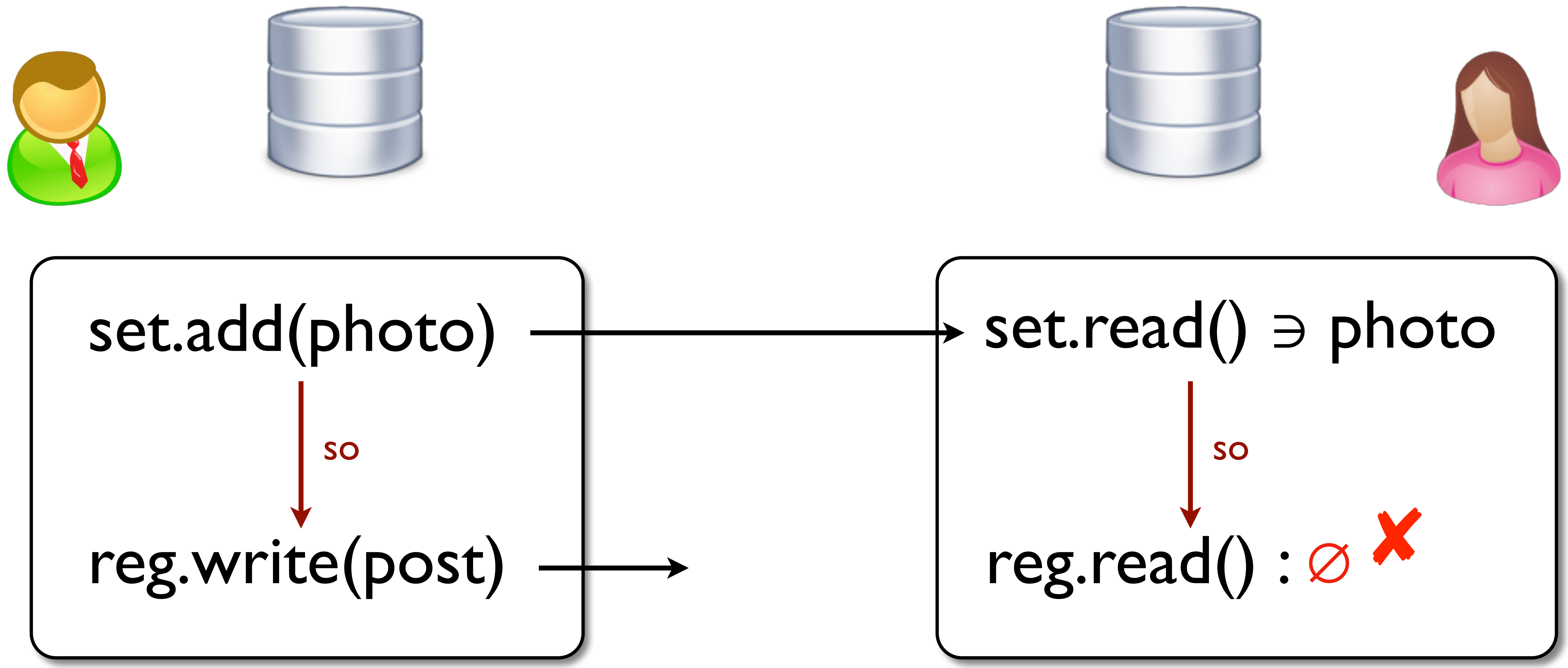
Transactions

- Fundamental abstraction in databases
- Allow clients to group operations to be processed indivisibly
- Provided by virtually any single-node SQL database
- NoSQL data stores: starting to reappear

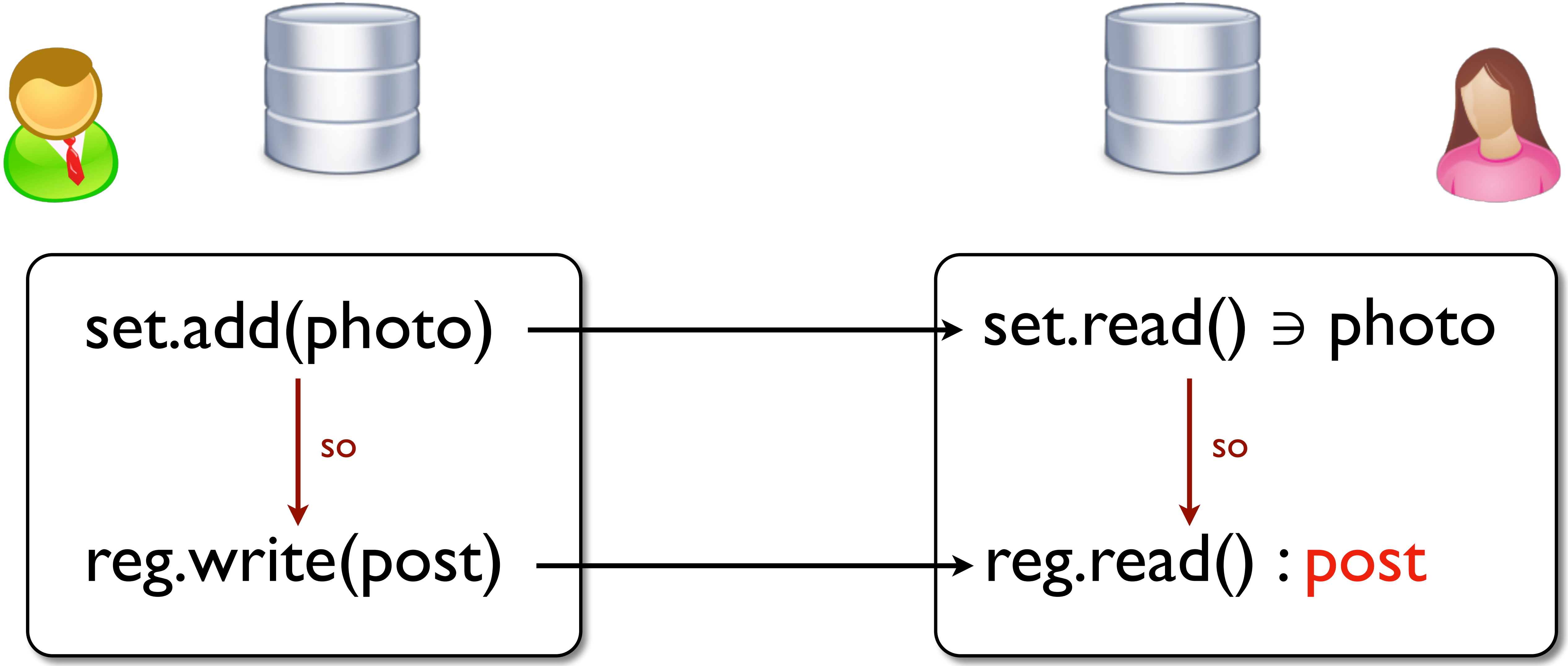
$$\begin{tikzcd} \text{set.add(photo)} \arrow[d, "\text{so}"] \\ \text{reg.write(post)} \end{tikzcd}$$

$$\begin{tikzcd} \text{set.read()} \ni \text{photo} \arrow[d, "\text{so}"] \\ \text{reg.read()} : \varnothing \end{tikzcd}$$


set.add(photo)
so
reg.write(post)
set.read() ∋ photo
so
reg.read() : ∅



Causal consistency isn't enough


set.add(photo)
so
reg.write(post)
set.read() ∋ photo
so
reg.read() : post

set.add(photo)

so

reg.write(post)

set.read() ∋ photo

so

reg.read() : post

set.add(photo)

$\downarrow$ so

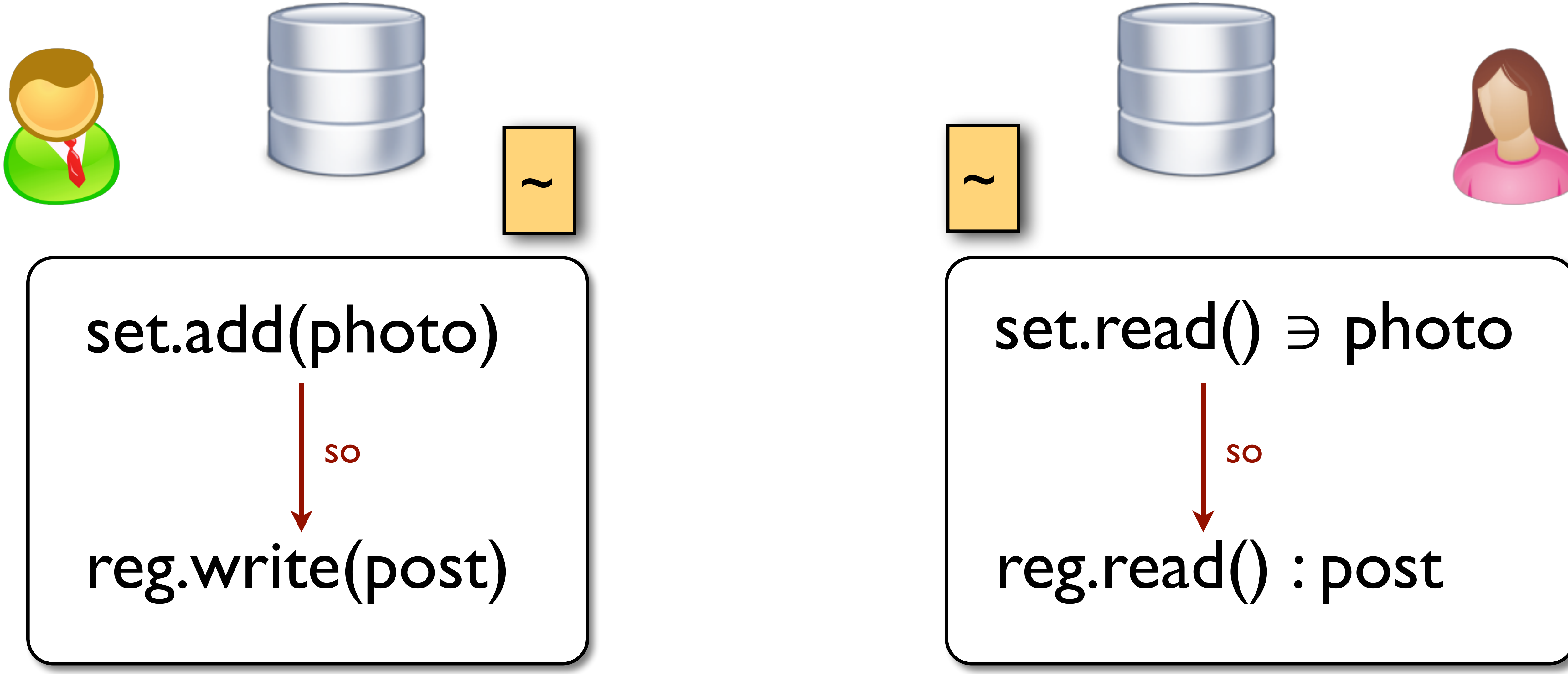
reg.write(post)

set.read() ∋ photo

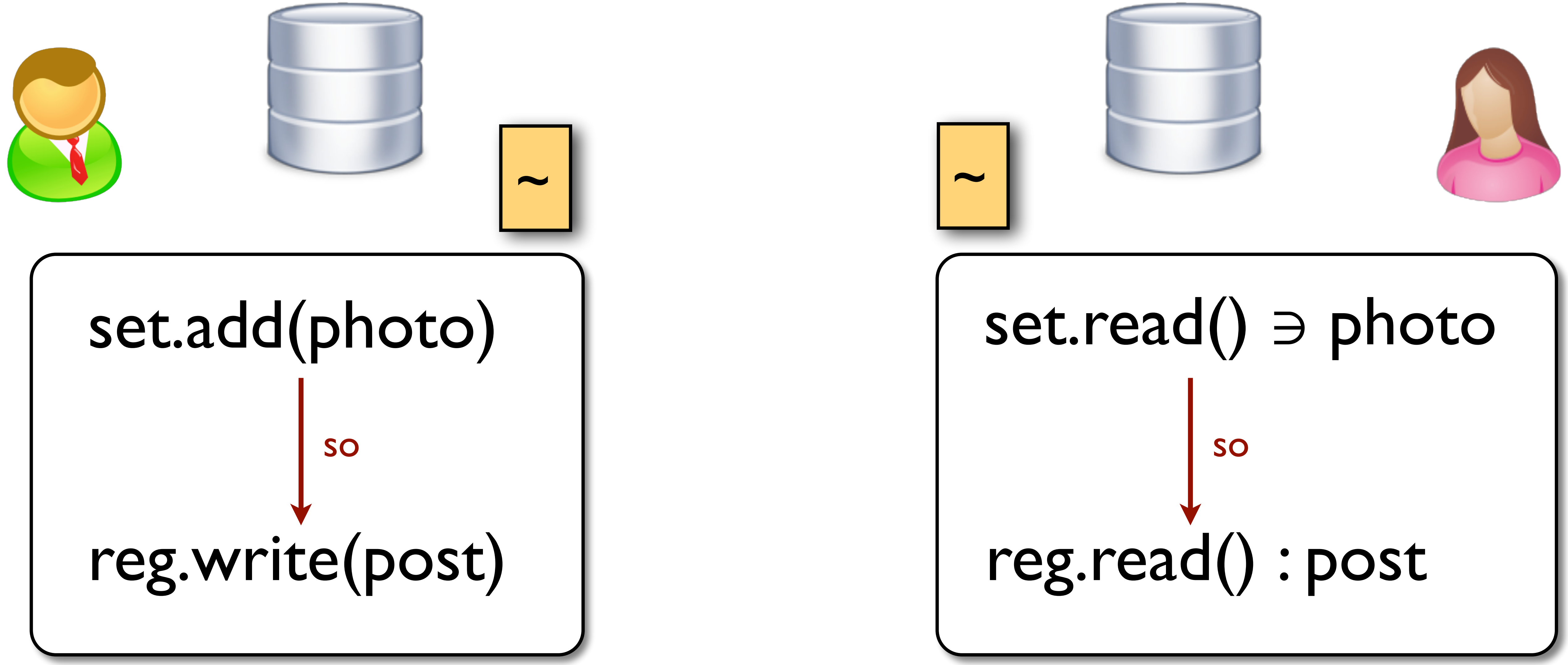
$\downarrow$ so

reg.read() : post

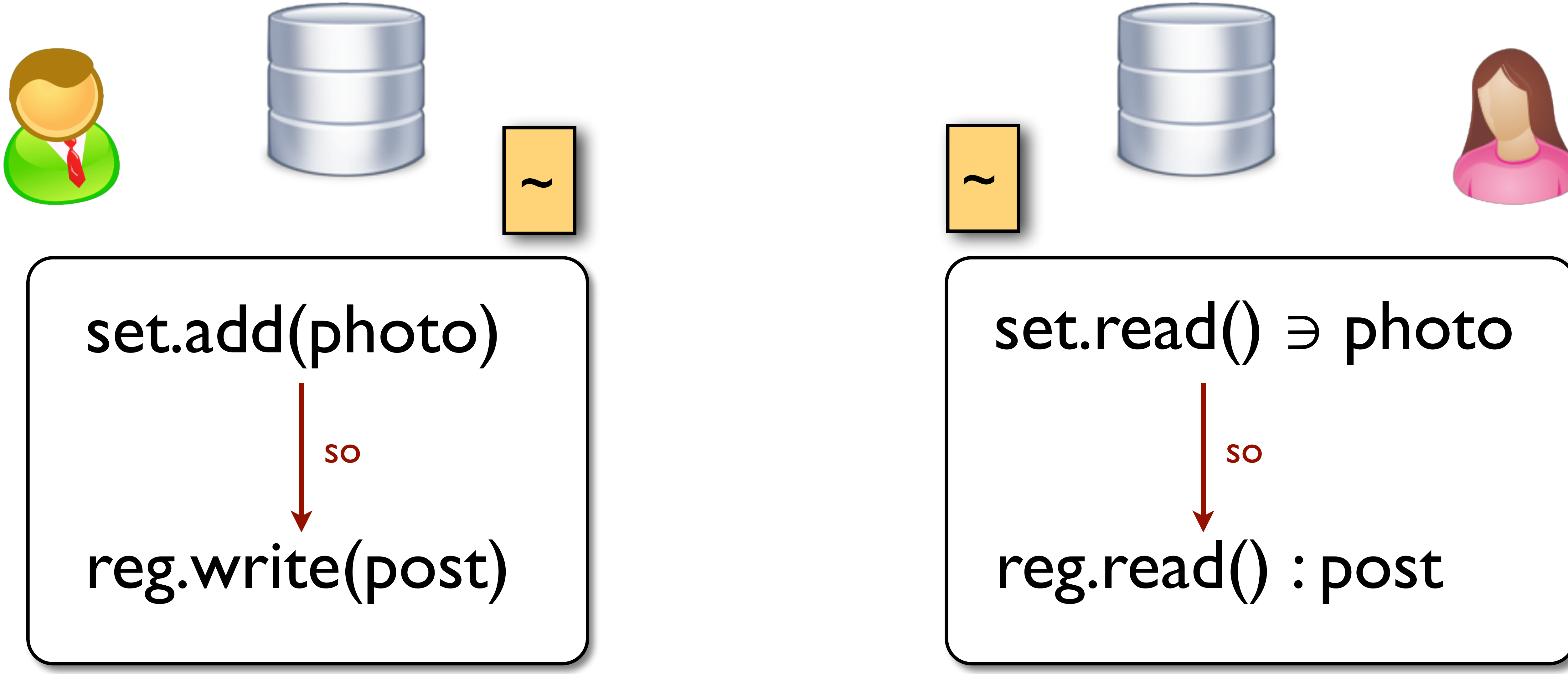
- Consistency model = set of histories (E, so, ~)



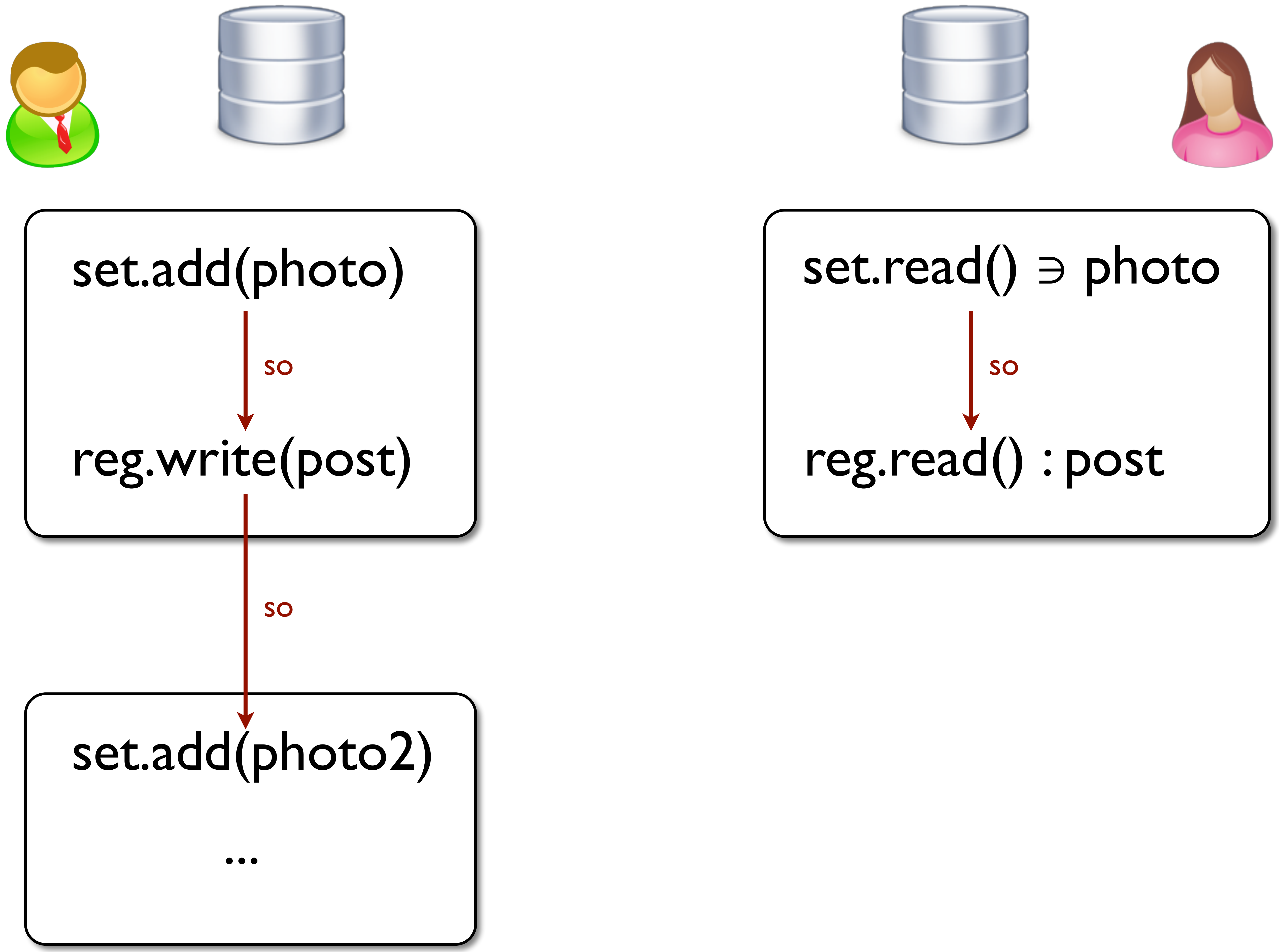
- Consistency model = set of histories (E, so, ~)
- ~: equivalence relation that groups events from the same transaction: transitive, symmetric, reflexive



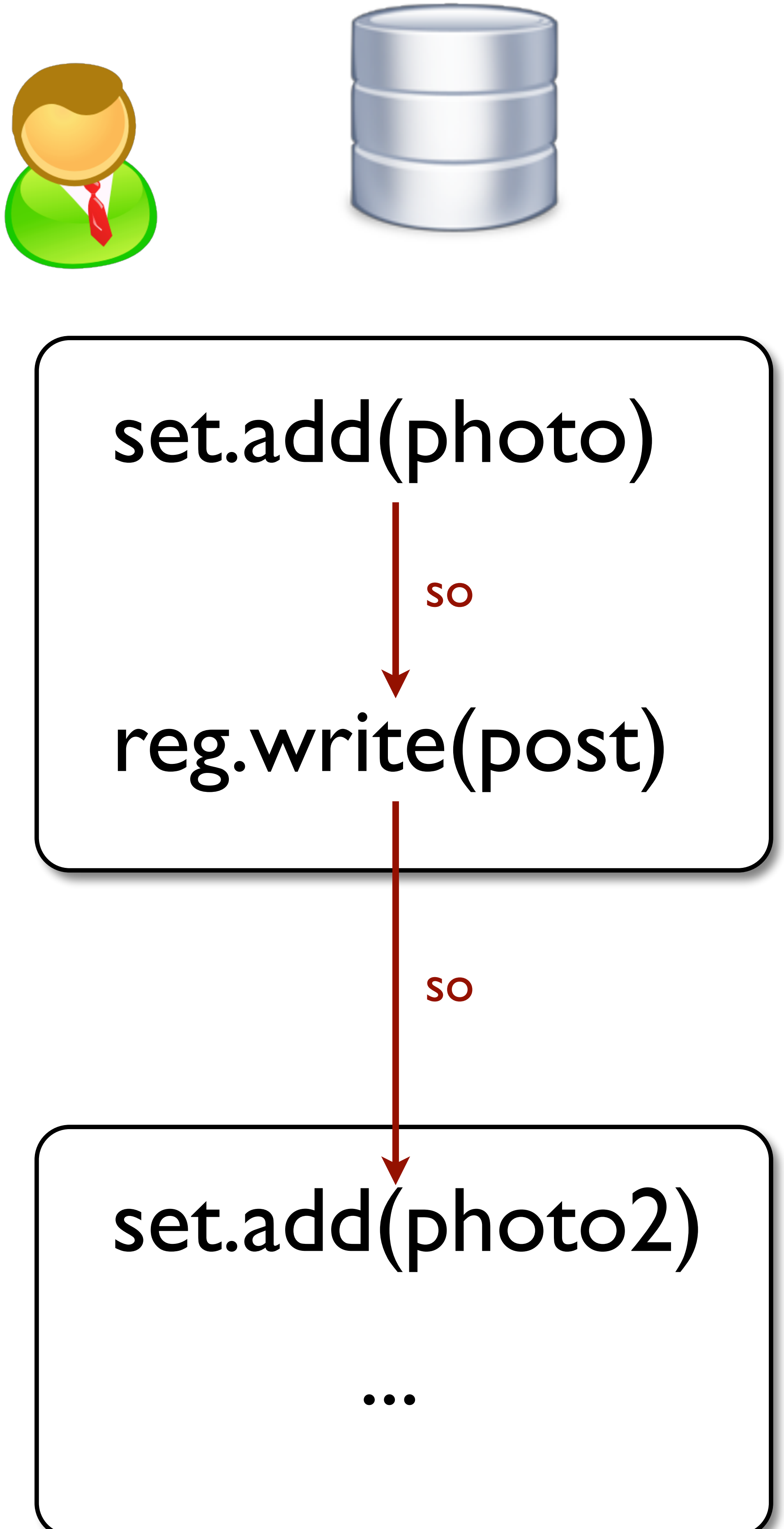
- Consistency model = set of histories (E, so, ~)
- ~: equivalence relation that groups events from the same transaction: transitive, symmetric, reflexive
- For simplicity, assume every transaction completes

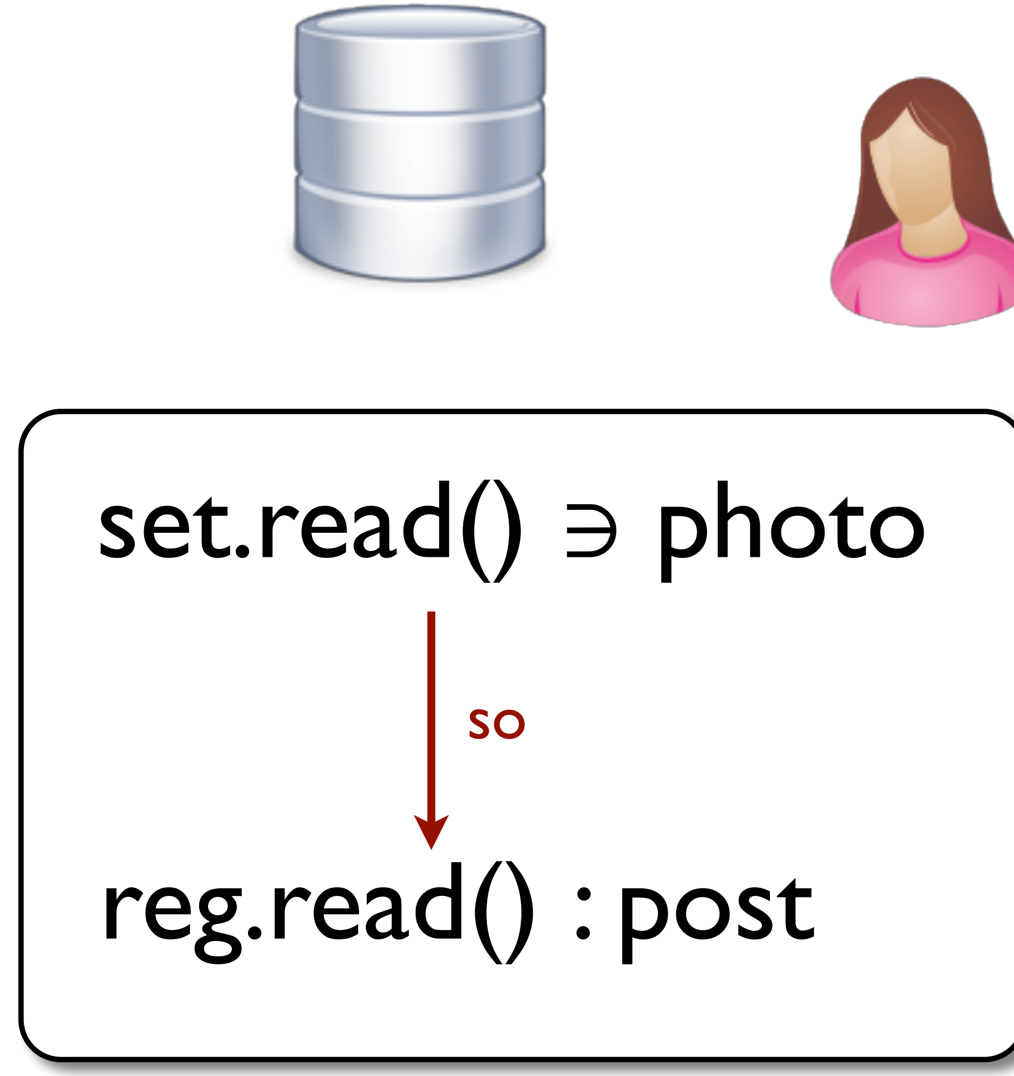


- Consistency model = set of histories (E, so, ~)
- ~: equivalence relation that groups events from the same transaction: transitive, symmetric, reflexive
- For simplicity, assume every transaction completes
- Transaction T: equivalence class of events of ~



set.add(photo)
so
reg.write(post)
so
set.add(photo2)
...
set.read() ∋ photo
so
reg.read() : post





A session is a sequence of transactions: events from the same transaction contiguous in so

$$\forall e, f, g \in E.\ e \xrightarrow{so} f \xrightarrow{so} g \wedge e \sim g$$
$$\Longrightarrow e \sim f \sim g$$

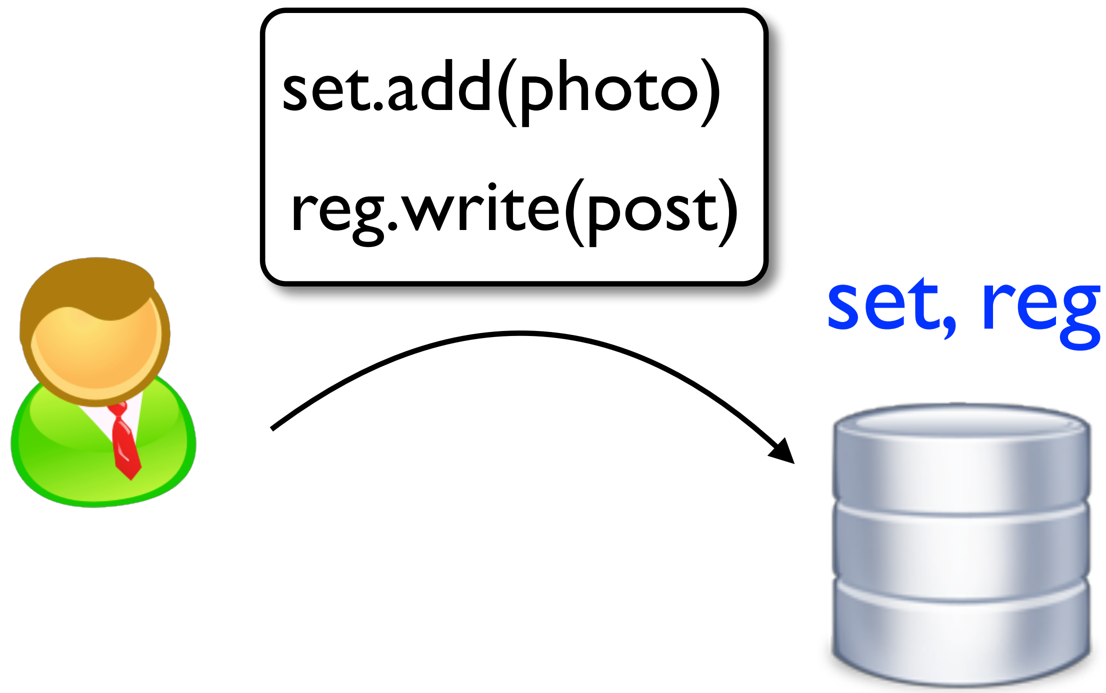
Strongly consistent transactions

Sequential consistency ~ serializability

Serializability operationally

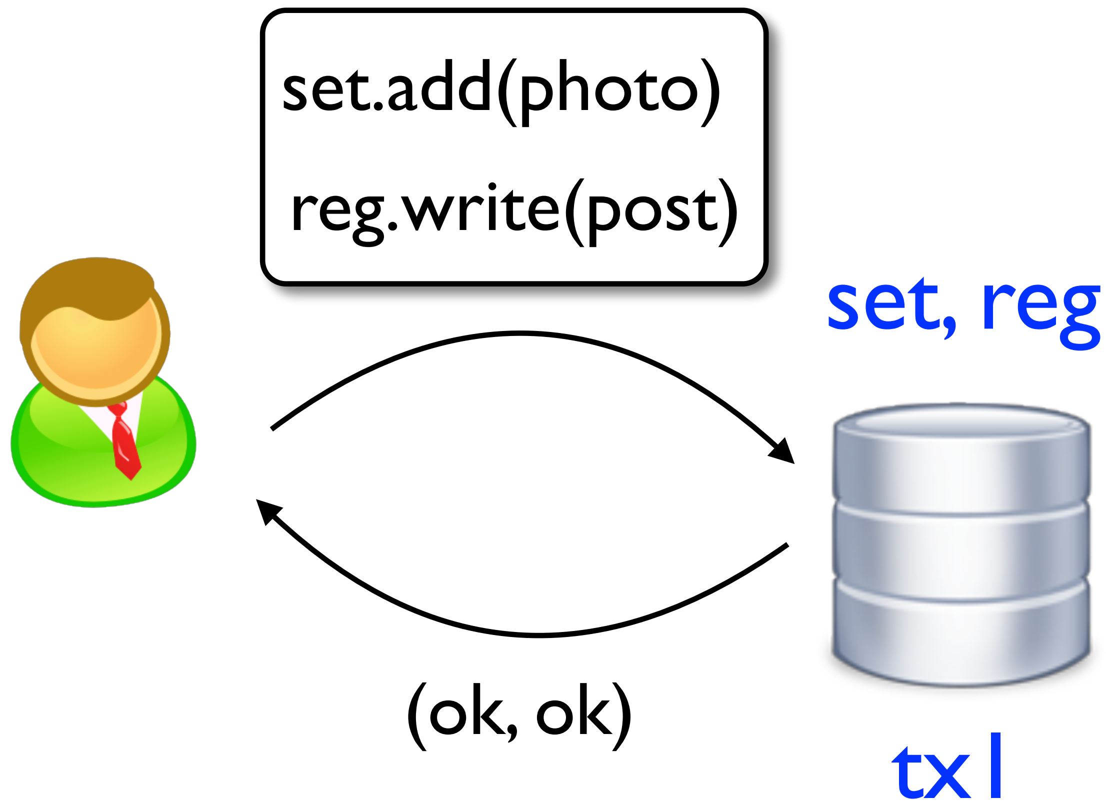
- Server with a single copy of all objects
- Clients send txs to the server and wait for a reply
- Server processes txs atomically in the receipt order

Serializability operationally



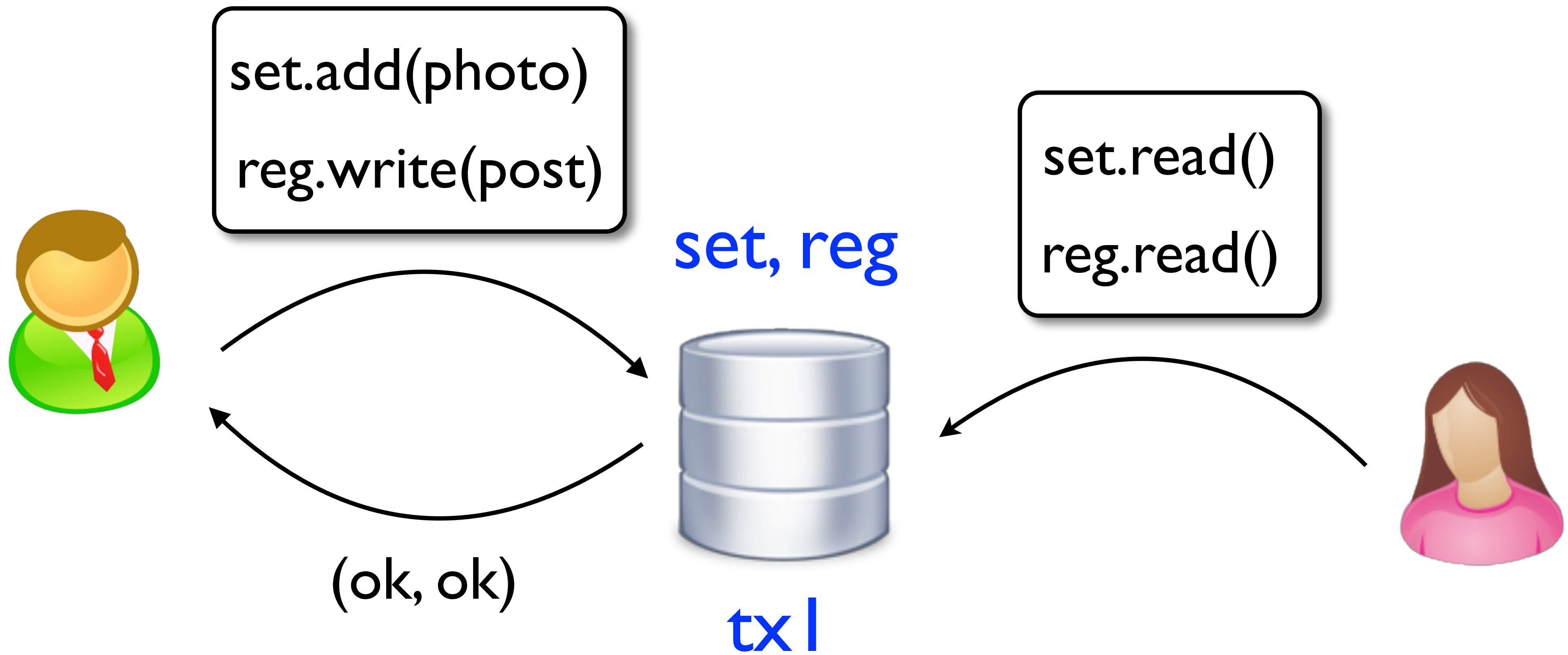
- Server with a single copy of all objects
- Clients send txs to the server and wait for a reply
- Server processes txs atomically in the receipt order

Serializability operationally



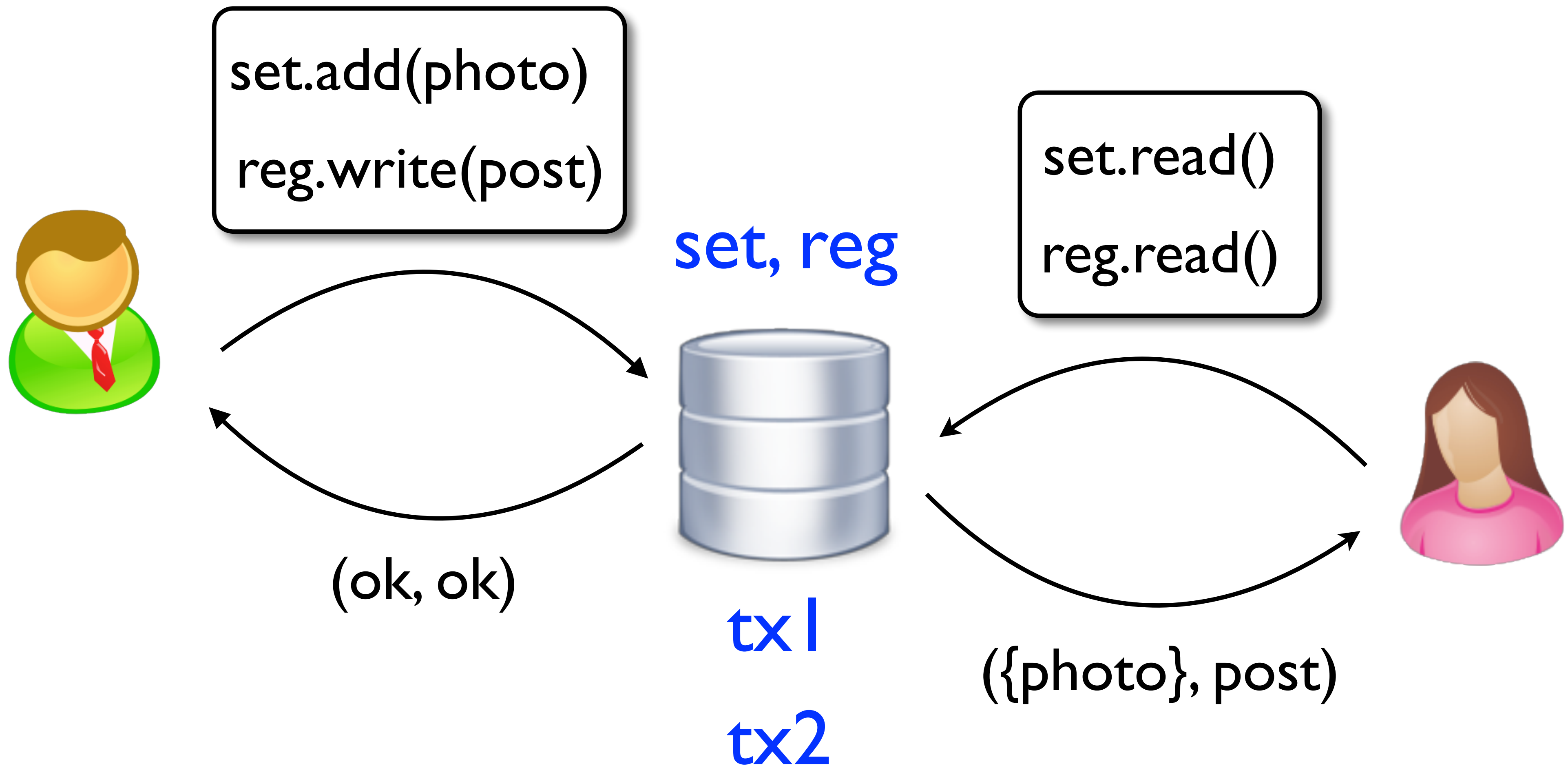
- Server with a single copy of all objects
- Clients send txs to the server and wait for a reply
- Server processes txs atomically in the receipt order

Serializability operationally



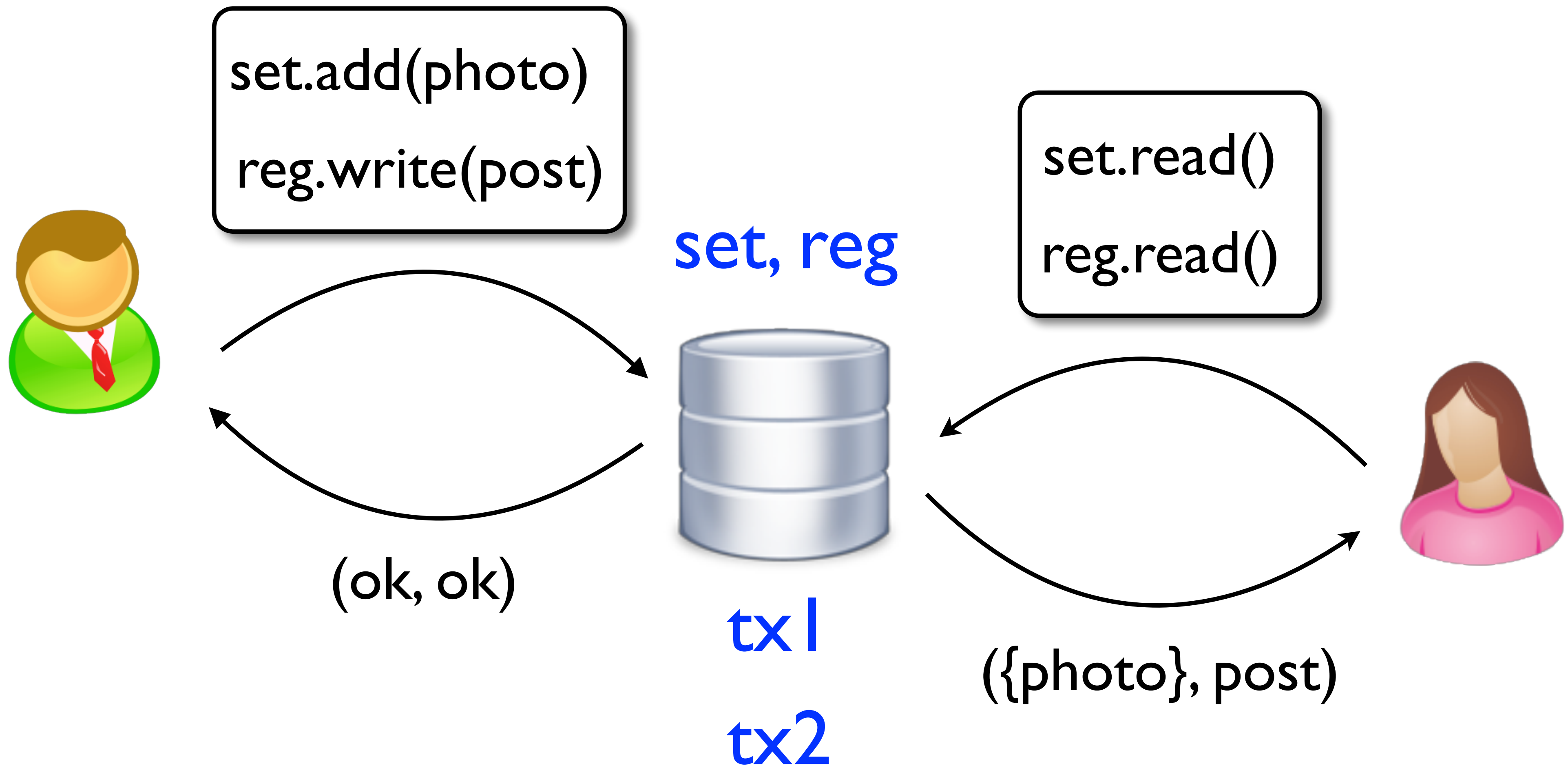
- Server with a single copy of all objects
- Clients send txs to the server and wait for a reply
- Server processes txs atomically in the receipt order

Serializability operationally



- Server with a single copy of all objects
- Clients send txs to the server and wait for a reply
- Server processes txs atomically in the receipt order

Serializability operationally



Serializability = {H | ∃ execution with history H produced by the abstract implementation}

Sequential consistency

(E,so) | ∃ total order to. (E, so, to) satisfies:

1. so ⊆ to
2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

Serializability

$(E, so, \sim) \mid \exists$ total order to. $(E, so, \sim, to)$ satisfies:

1. $so \subseteq to$
2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to
3. Operations from the same transaction are contiguous in to

set.add(photo)

$\downarrow$ so

reg.write(post)

$\downarrow$ so

set.add(photo2)

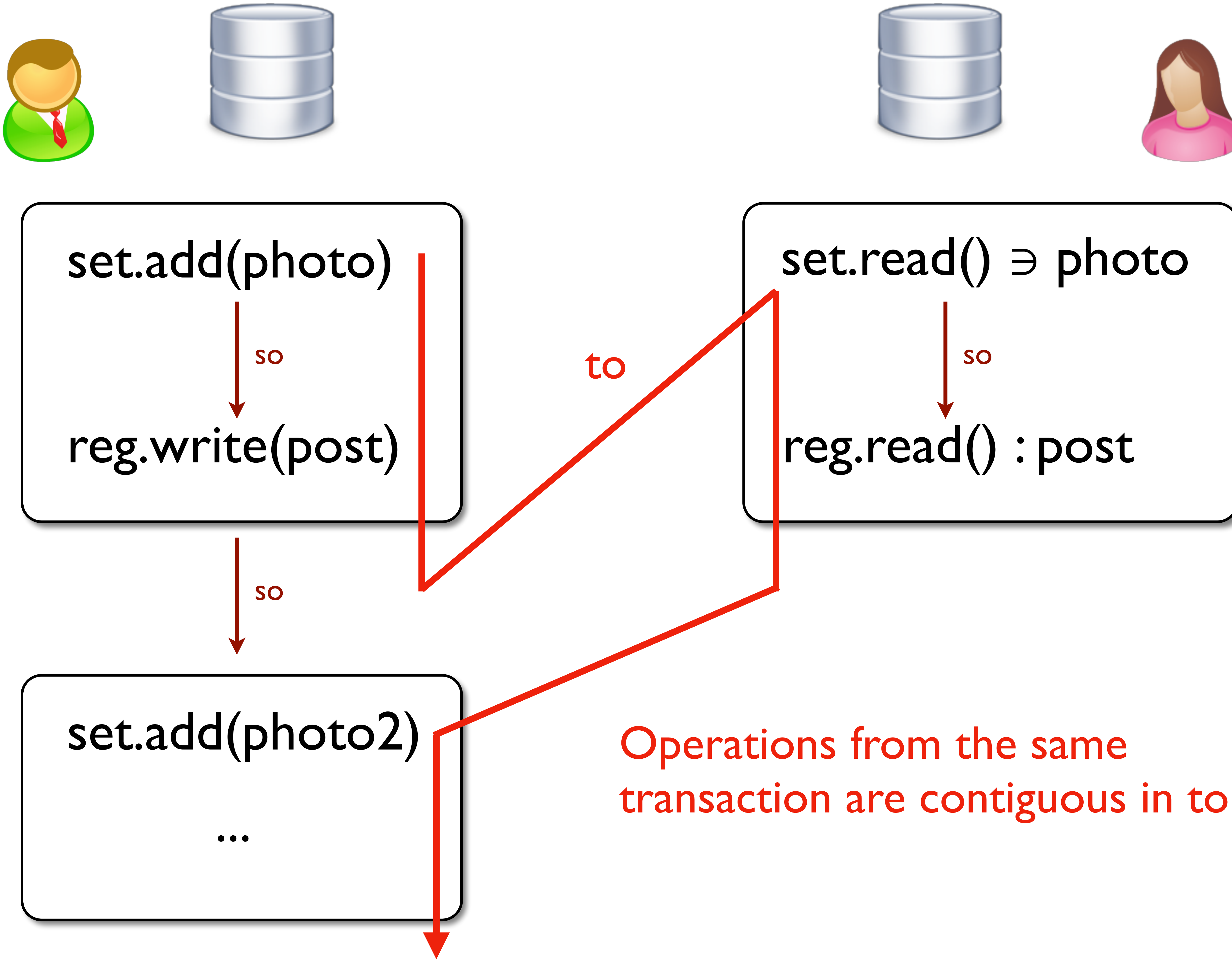
...

set.read() ∋ photo

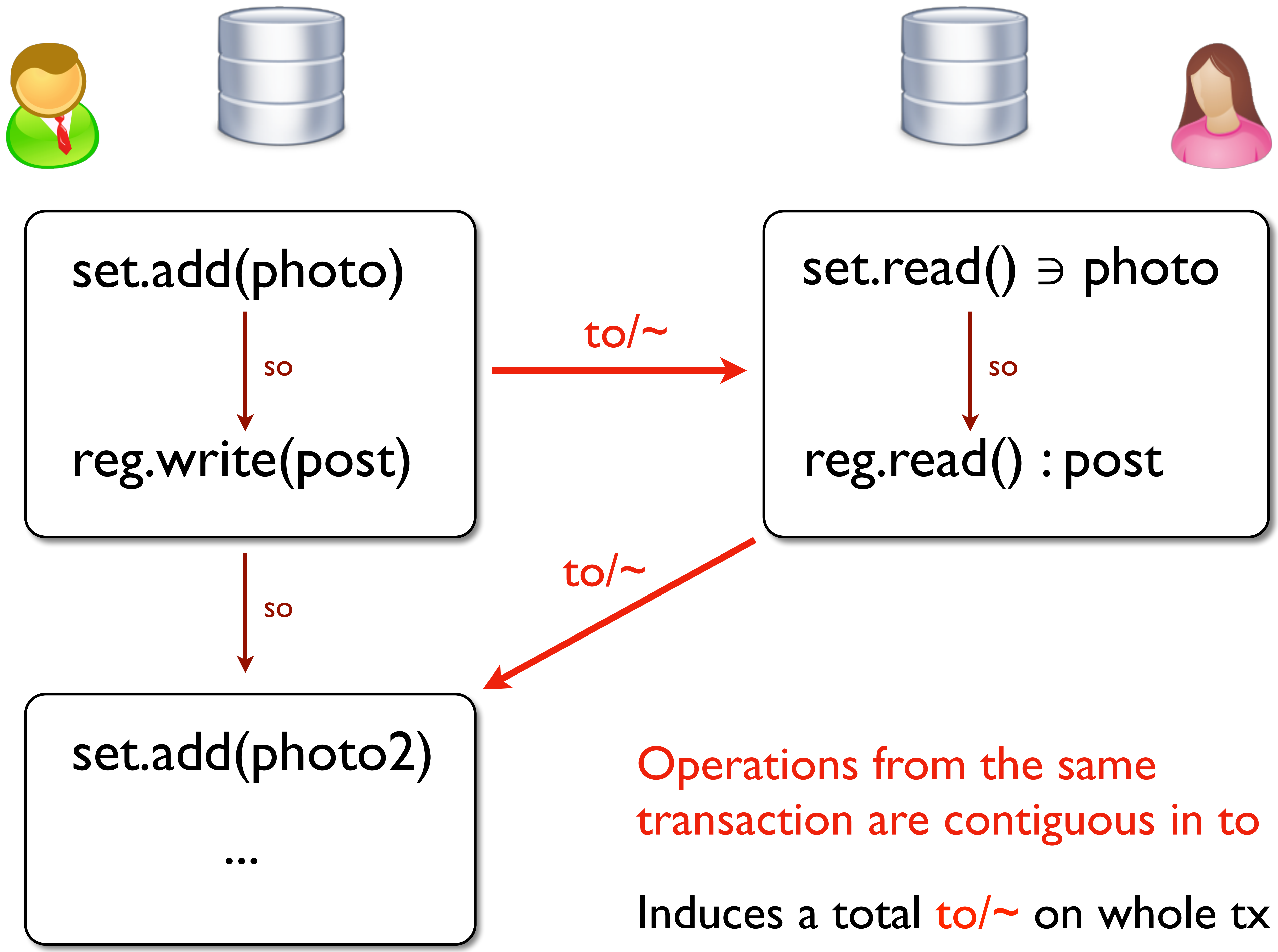
$\downarrow$ so

reg.read() : post

Operations from the same transaction are contiguous in to



Operations from the same transaction are contiguous in to

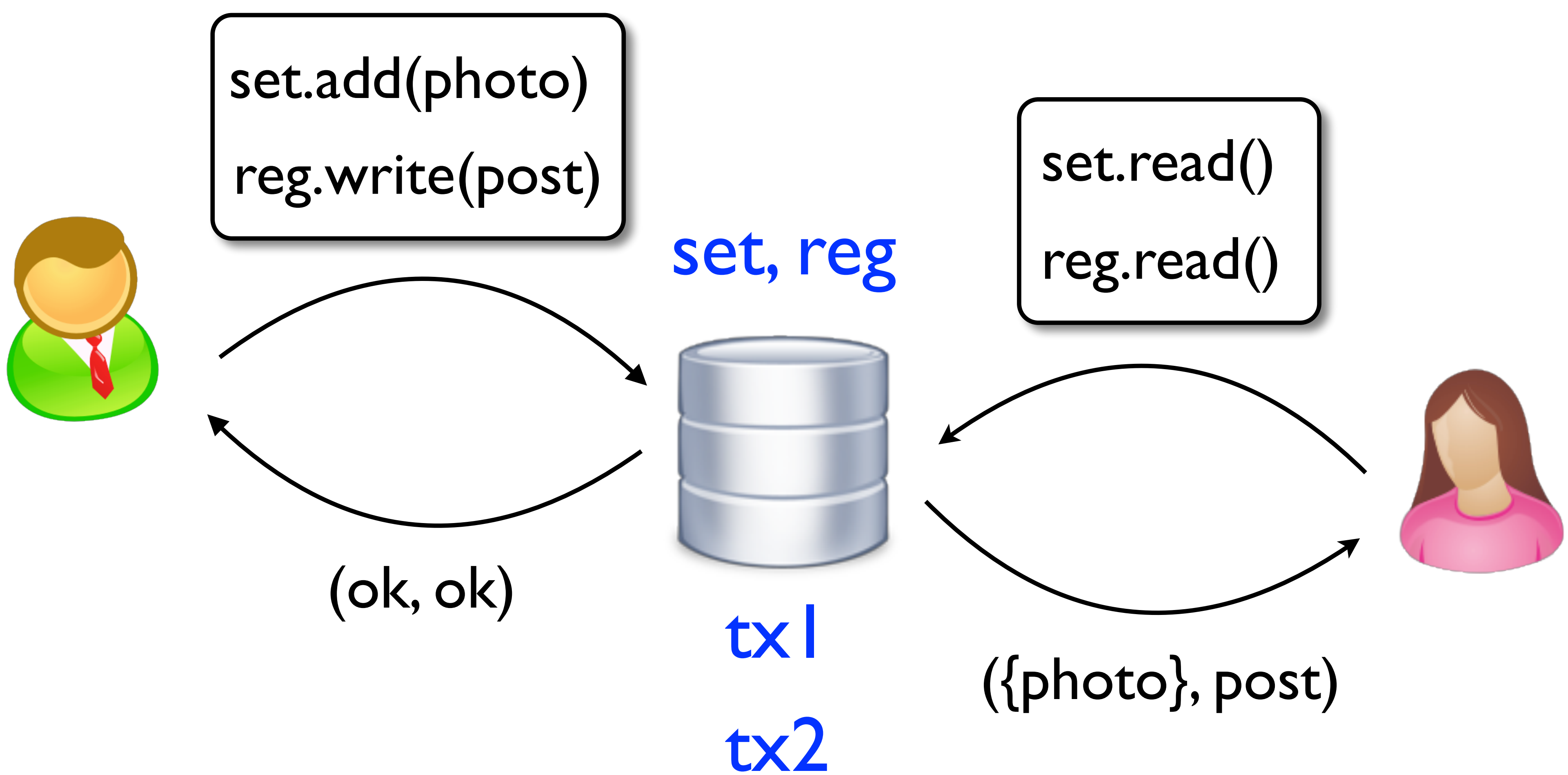

set.add(photo)
so
reg.write(post)
to/~
set.read() ∋ photo
so
reg.read() : post
so
to/~
set.add(photo2)
...
Operations from the same transaction are contiguous in to
Induces a total to/~ on whole tx

Weakening consistency

- Even single-node databases don't provide serializability either by default or at all: read committed, snapshot isolation, ...

Weakening consistency

- Even single-node databases don't provide serializability either by default or at all: read committed, snapshot isolation, ...
- To better exploit single-node parallelism



Eventually consistent transactions

- Single-node consistency models also applicable in distributed setting
- But many still require some synchronisation between replicas: unavailability, high latency
- Want eventually consistent transactions: always available, low latency
- Preserve some aspects of the invisibility abstraction

System model recap

- Database system consisting of multiple reliable replicas
- Each replica stores a copy of all objects of replicated data types
- Replicas can communicate via asynchronous reliable channels

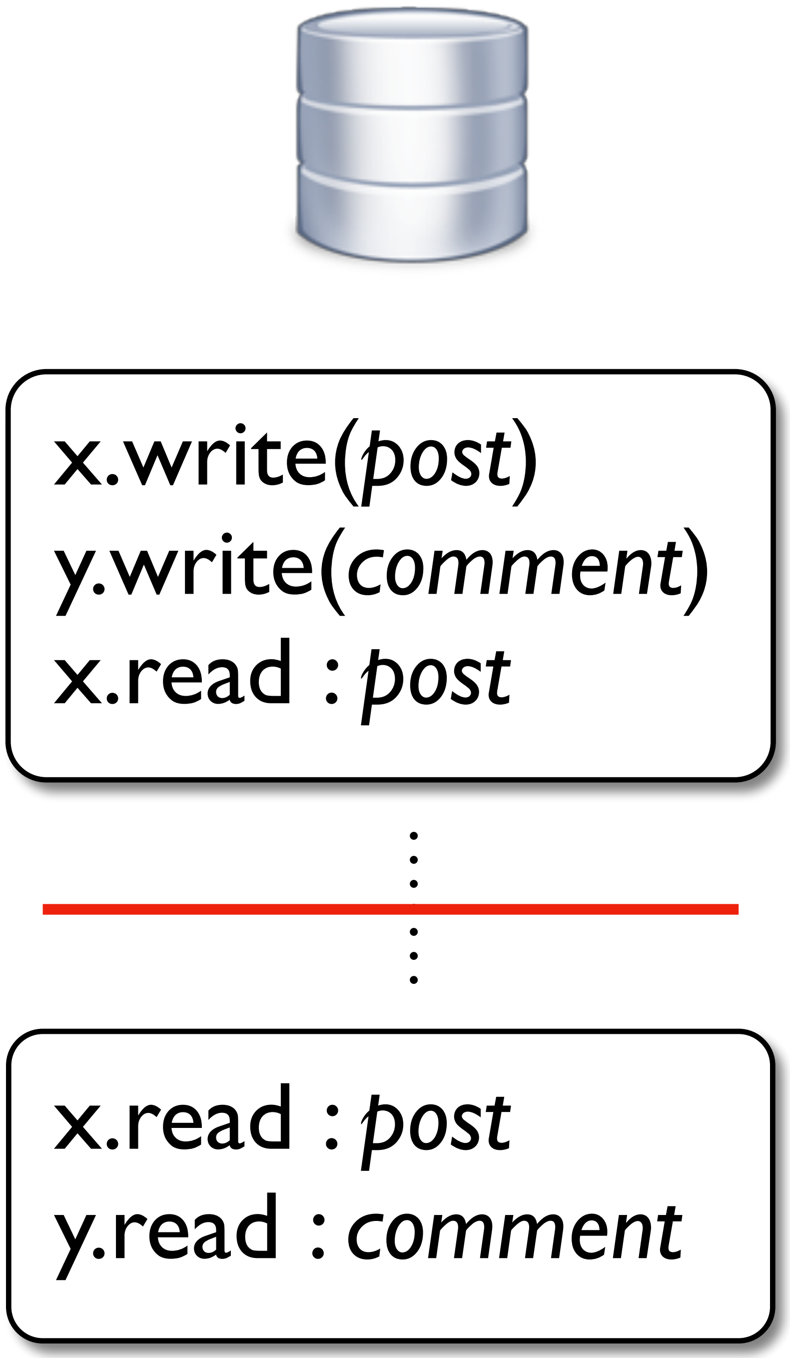
x.write(*post*)
y.write(*comment*)
x.read : *post*

- A client connects to a replica and issues transactions
- High availability: the transaction commits immediately, without communication with other replicas, no aborts!

x.write(*post*)
y.write(*comment*)
x.read : *post*

x.read : *post*
y.read : *comment*

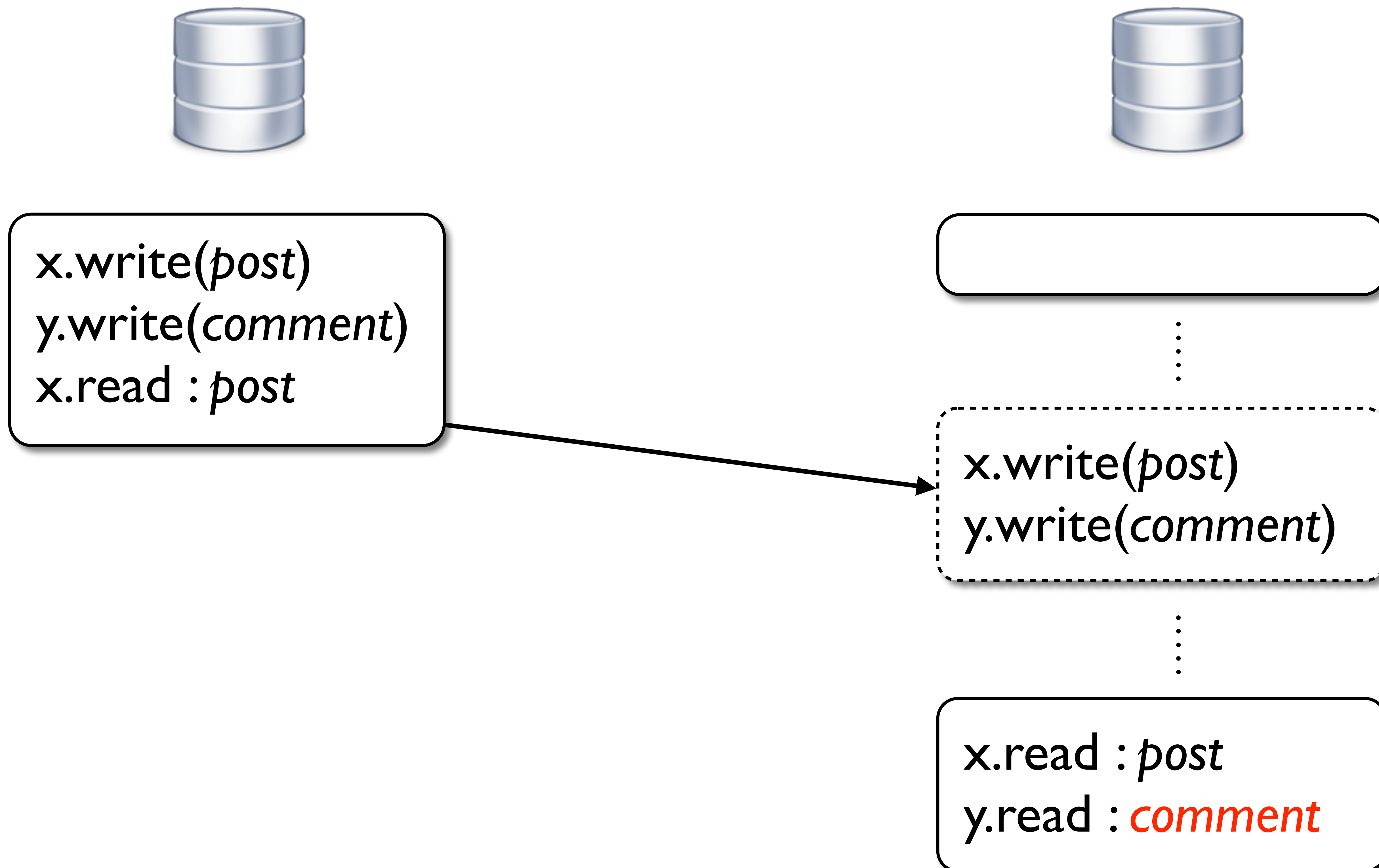
- A client connects to a replica and issues transactions
- High availability: the transaction commits immediately, without communication with other replicas, no aborts!
- Replica processes transactions sequentially: anomalies arising from single-node concurrency covered by the absence of inter-node synchronisation



- A client connects to a replica and issues transactions
- High availability: the transaction commits immediately, without communication with other replicas, no aborts!
- Replica processes transactions sequentially: anomalies arising from single-node concurrency covered by the absence of inter-node synchronisation
- Reads are indivisible: access a fixed snapshot of the database (plus own writes)

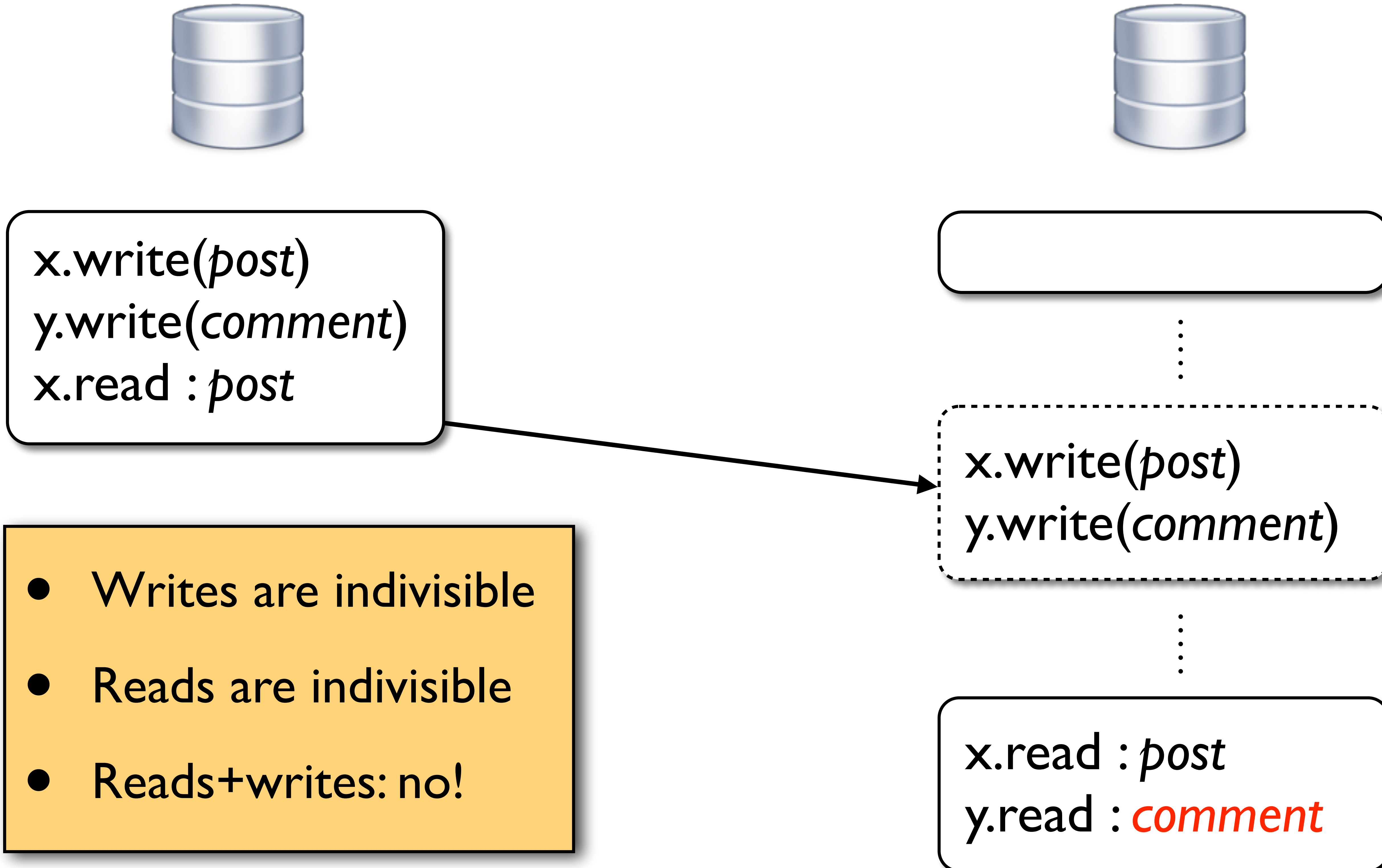
x.write(*post*)
y.write(*comment*)
x.read : *post*

Upon commit: send the effectors of all tx operations to other replicas together



Upon commit: send the effectors of all tx operations to other replicas together

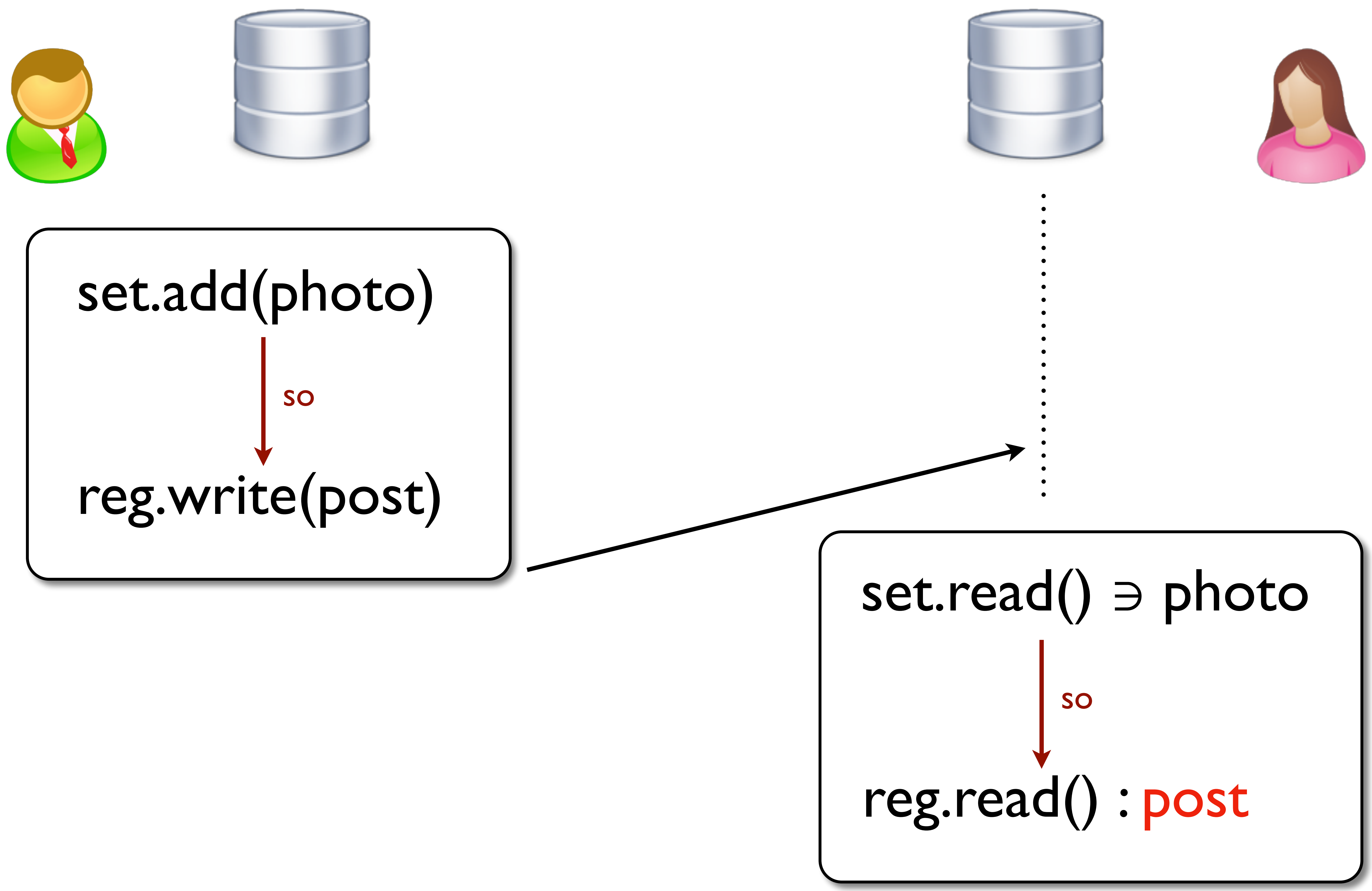
Receive in between txs: incorporate all the updates together



Upon commit: send the effectors of all tx operations to other replicas together

Receive in between txs: incorporate all the updates together

Reads/writes indivisibility



No reads+writes indivisibility

reg: last-writer-wins register, initially 0

v = reg.read()   // 0

$\downarrow$ so

reg.write(v+1) // 1

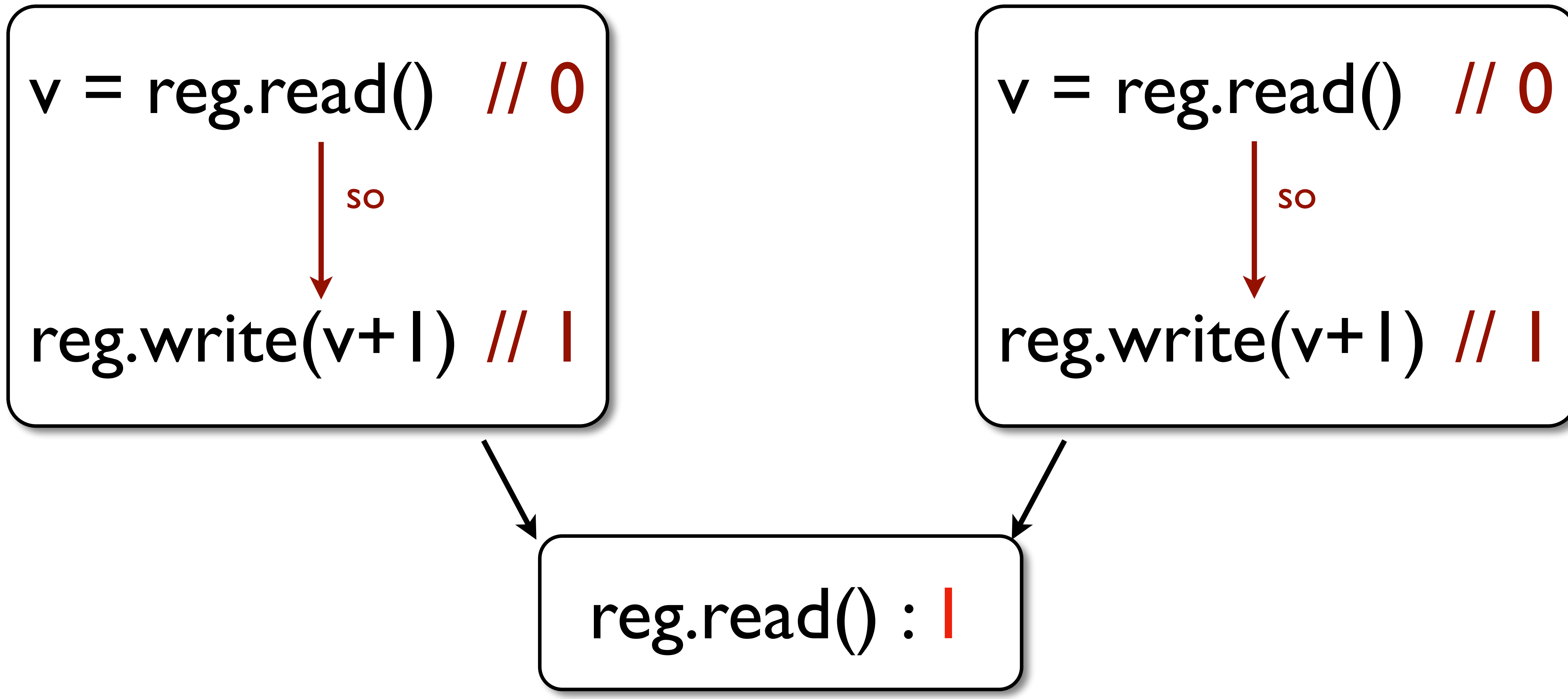
v = reg.read()   // 0

$\downarrow$ so

reg.write(v+1) // 1

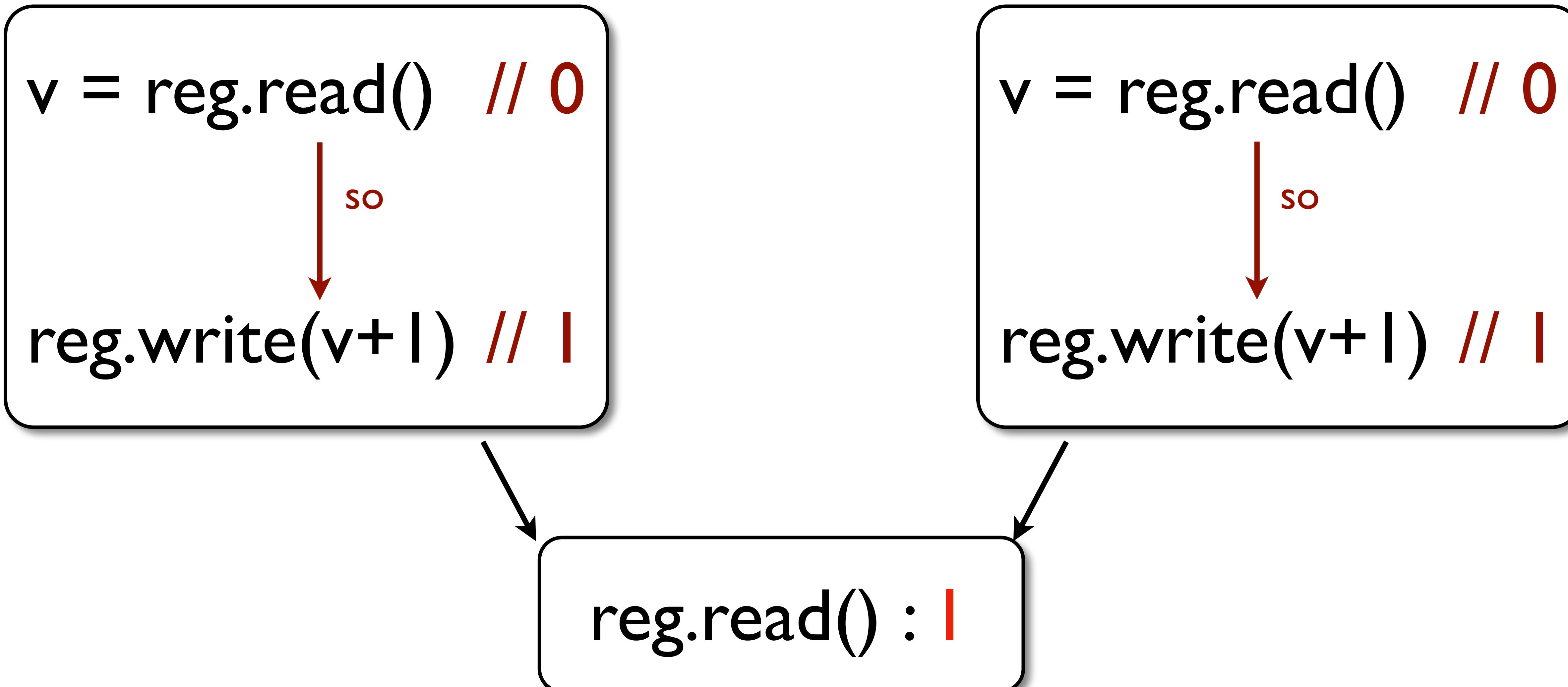
No reads+writes indivisibility

reg: last-writer-wins register, initially 0



No reads+writes indivisibility

reg: last-writer-wins register, initially 0



Lost update anomaly

Use appropriate data type

counter: replicated counter, accumulates increments
initially 0

Operational specification

- Eventual consistency with transactions = the set of all histories produced by arbitrary client interactions with the data type implementations (with any allowed message deliveries)
- Implies quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

Axiomatic specification

- Serializability: operations from the same transaction are contiguous in the total order to
- Approach: require the same of vis and ar

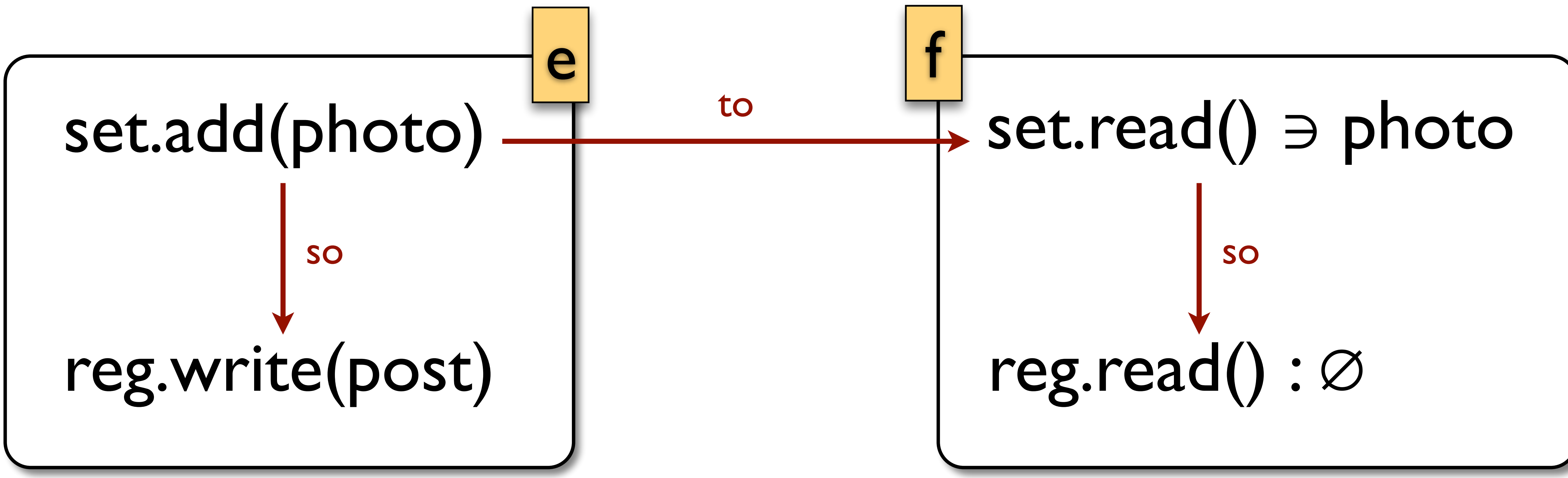
Serializability: (E, so, ~, to)

$$\begin{tikzcd} \text{set.add(photo)} \arrow[d, "\text{so}"] \\ \text{reg.write(post)} \end{tikzcd} \qquad \begin{tikzcd} \text{set.read()} \ni \text{photo} \arrow[d, "\text{so}"] \\ \text{reg.read()} : \varnothing \end{tikzcd}$$

Operations from the same transaction are contiguous in to:

$$\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\text{to}} f \sim f' \implies e' \xrightarrow{\text{to}} f'$$

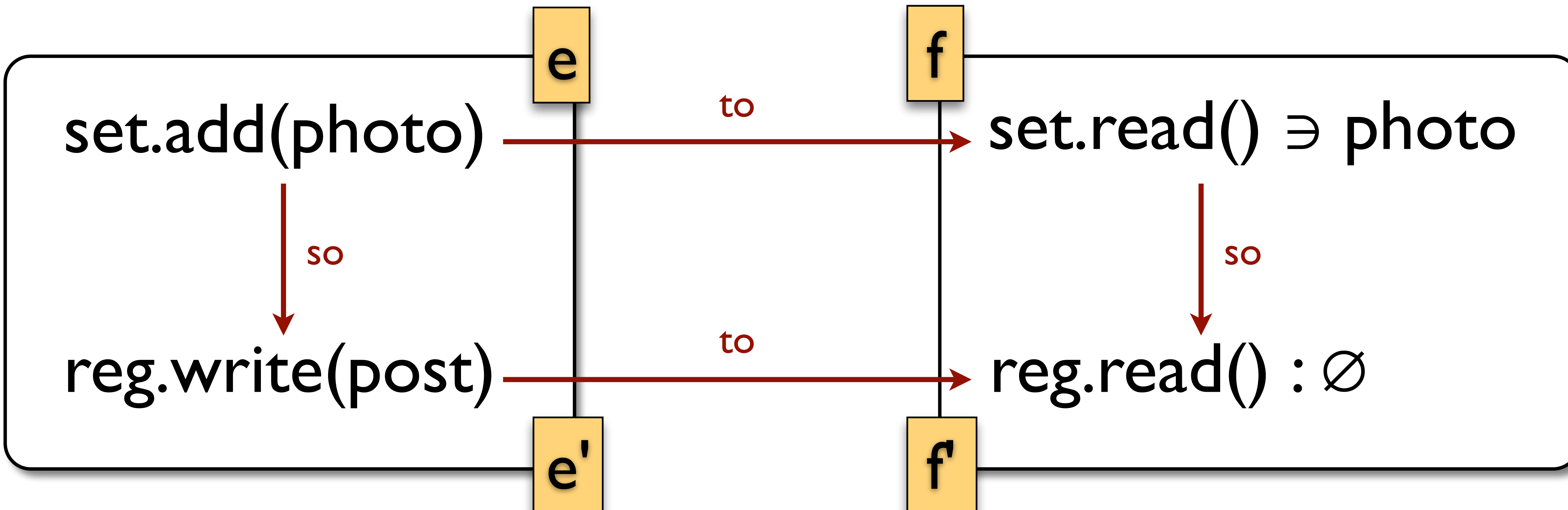
Serializability: (E, so, ~, to)



Operations from the same transaction are contiguous in to:

$$\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\text{to}} f \sim f' \implies e' \xrightarrow{\text{to}} f'$$

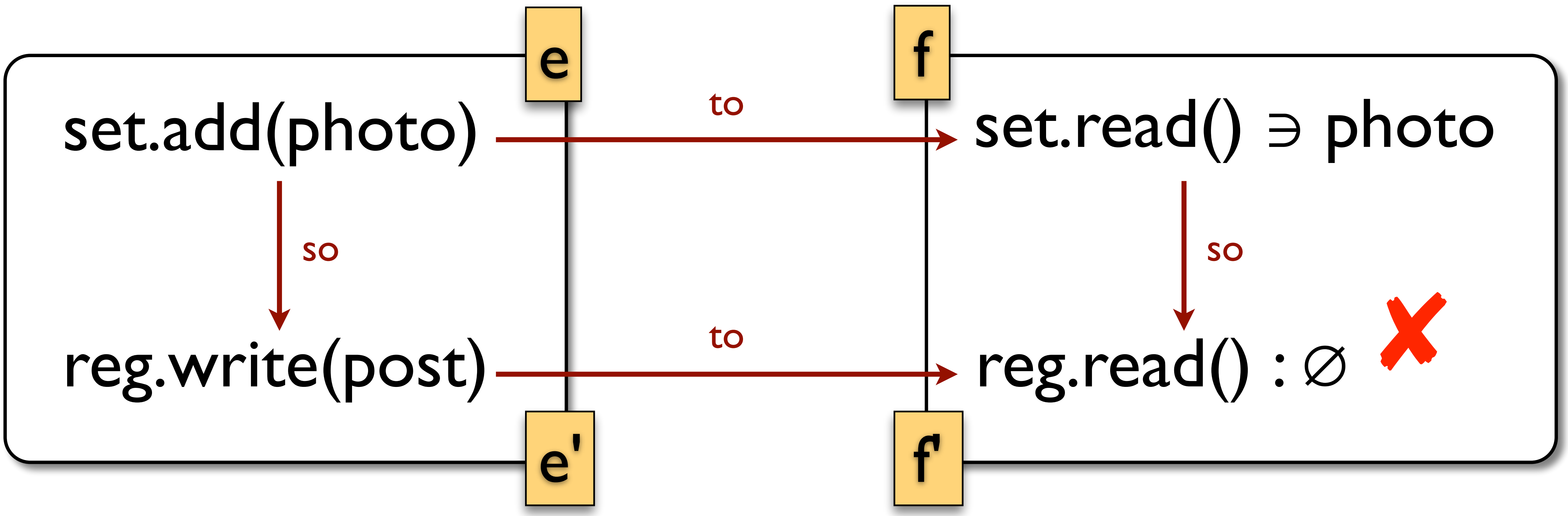
Serializability: (E, so, ~, to)



Operations from the same transaction are contiguous in to:

$$\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\text{to}} f \sim f' \implies e' \xrightarrow{\text{to}} f'$$

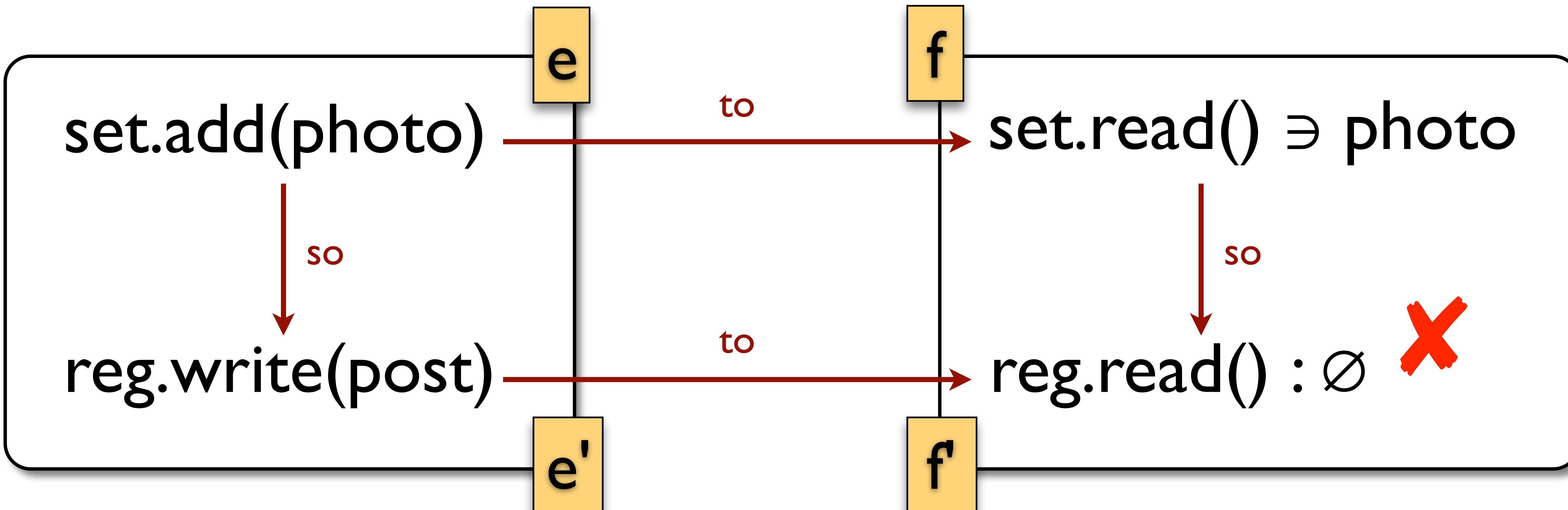
Serializability: (E, so, ~, to)



Operations from the same transaction are contiguous in to:

$$\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\text{to}} f \sim f' \implies e' \xrightarrow{\text{to}} f'$$

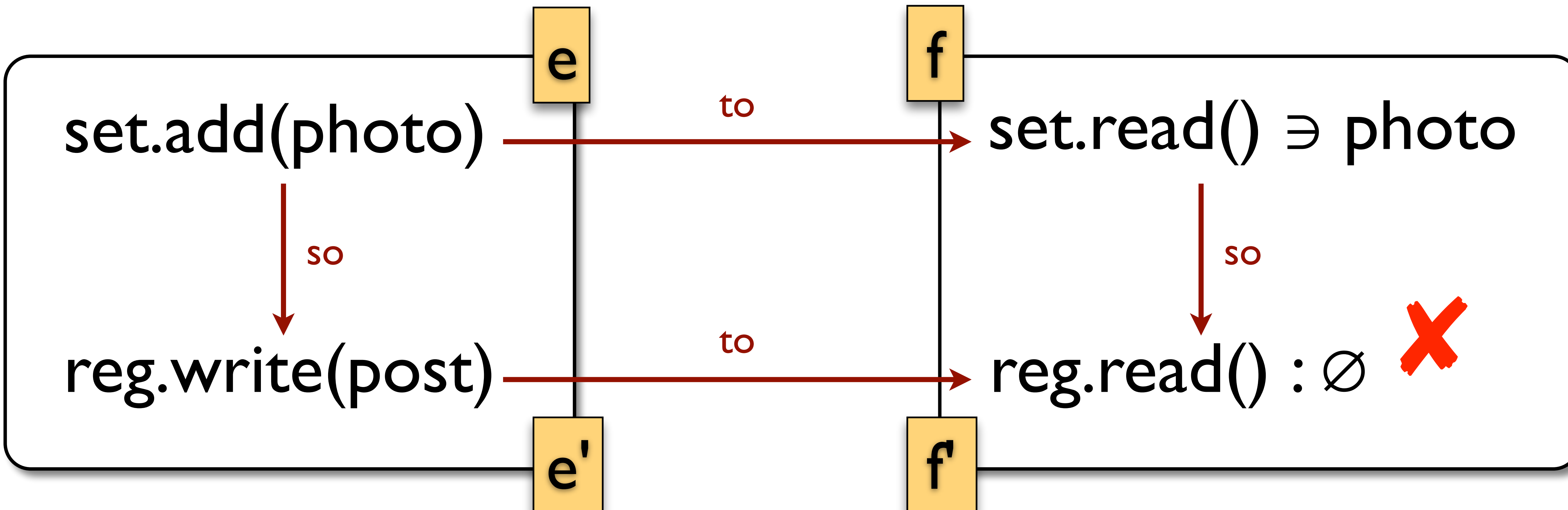
Serializability: (E, so, ~, to)



Operations from the same transaction are contiguous in to:

$$\forall e, f, e', f'.\ \boxed{e \nsim f} \wedge e' \sim e \xrightarrow{\text{to}} f \sim f' \implies e' \xrightarrow{\text{to}} f'$$

Serializability: (E, so, ~, to)



Operations from the same transaction are contiguous in to:

$$\forall e, f, e', f'.\ \boxed{e \not\sim f} \wedge e' \sim e \xrightarrow{\text{to}} f \sim f' \implies e' \xrightarrow{\text{to}} f'$$

to treats events in a transaction uniformly

Execution: (E, so, ~, vis, ar)

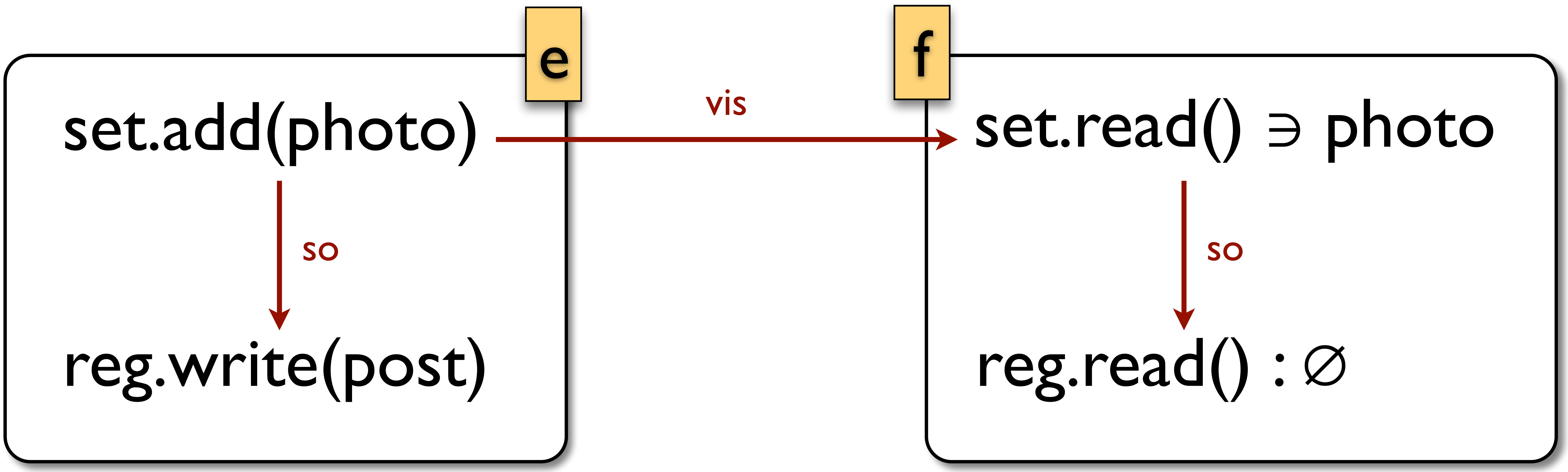
$$\begin{tikzcd} \text{set.add(photo)} \arrow[d, "\text{so}"] \\ \text{reg.write(post)} \end{tikzcd} \qquad\qquad \begin{tikzcd} \text{set.read()} \ni \text{photo} \arrow[d, "\text{so}"] \\ \text{reg.read()} : \varnothing \end{tikzcd}$$

vis, ar treat events in a transaction uniformly:

$$\forall e, f, e', f'.\ e \not\sim f \wedge e' \sim e \xrightarrow{\text{vis}} f \sim f' \implies e' \xrightarrow{\text{vis}} f'$$

$$\forall e, f, e', f'.\ e \not\sim f \wedge e' \sim e \xrightarrow{\text{ar}} f \sim f' \implies e' \xrightarrow{\text{ar}} f'$$

Execution: (E, so, ~, vis, ar)

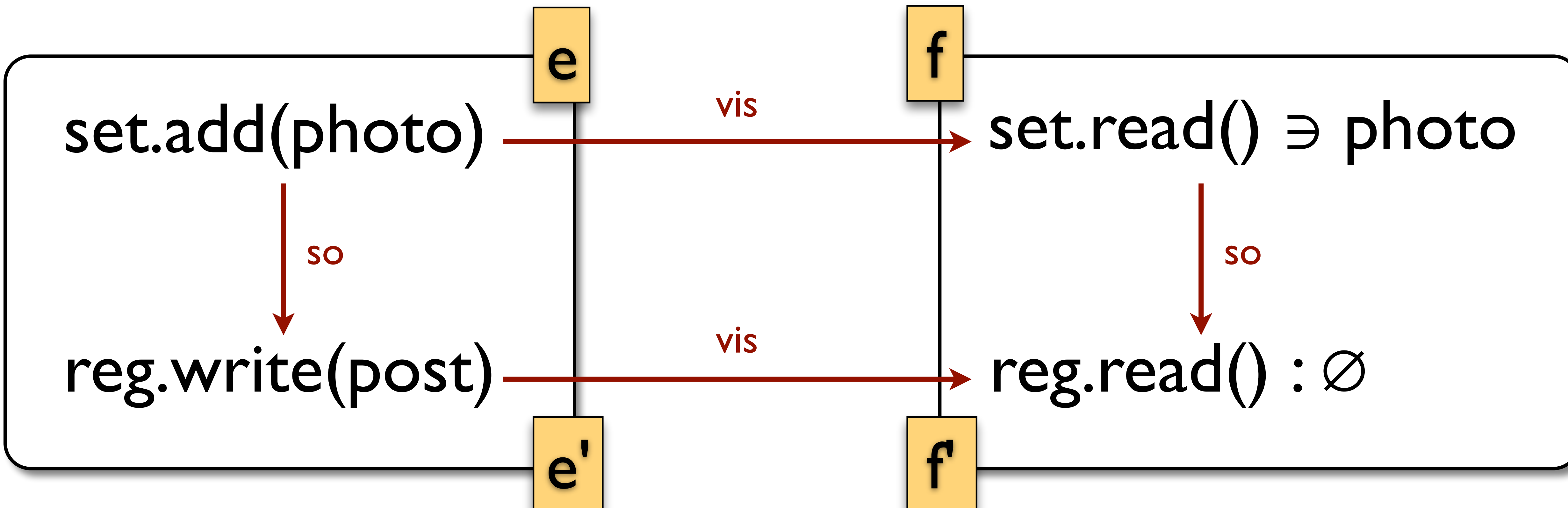


vis, ar treat events in a transaction uniformly:

$$\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\text{vis}} f \sim f' \implies e' \xrightarrow{\text{vis}} f'$$

$$\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\text{ar}} f \sim f' \implies e' \xrightarrow{\text{ar}} f'$$

Execution: (E, so, ~, vis, ar)

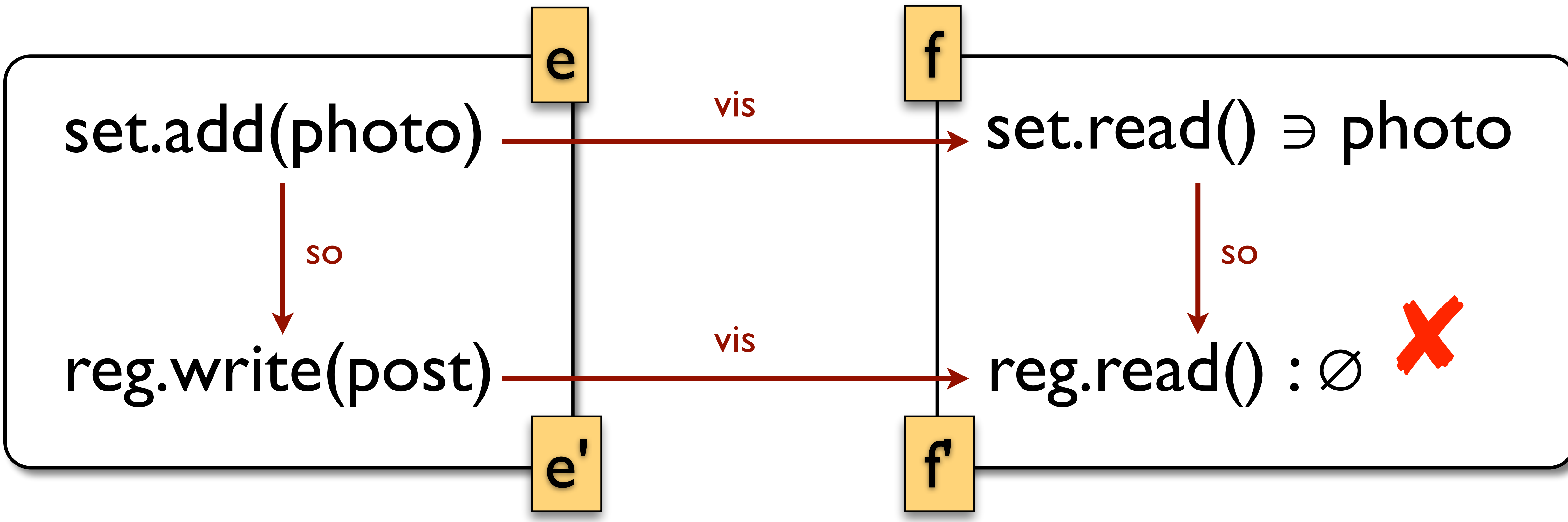


vis, ar treat events in a transaction uniformly:

$$\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\text{vis}} f \sim f' \implies e' \xrightarrow{\text{vis}} f'$$

$$\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\text{ar}} f \sim f' \implies e' \xrightarrow{\text{ar}} f'$$

Execution: (E, so, ~, vis, ar)

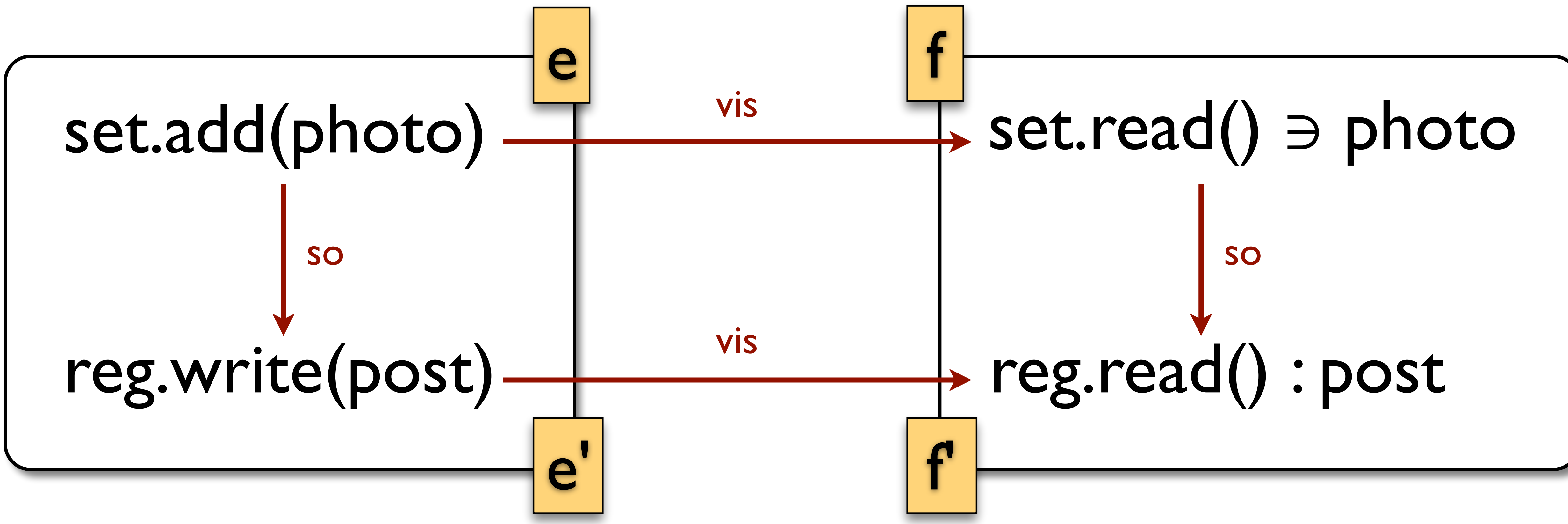


vis, ar treat events in a transaction uniformly:

$$\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\text{vis}} f \sim f' \implies e' \xrightarrow{\text{vis}} f'$$

$$\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\text{ar}} f \sim f' \implies e' \xrightarrow{\text{ar}} f'$$

Execution: (E, so, ~, vis, ar)

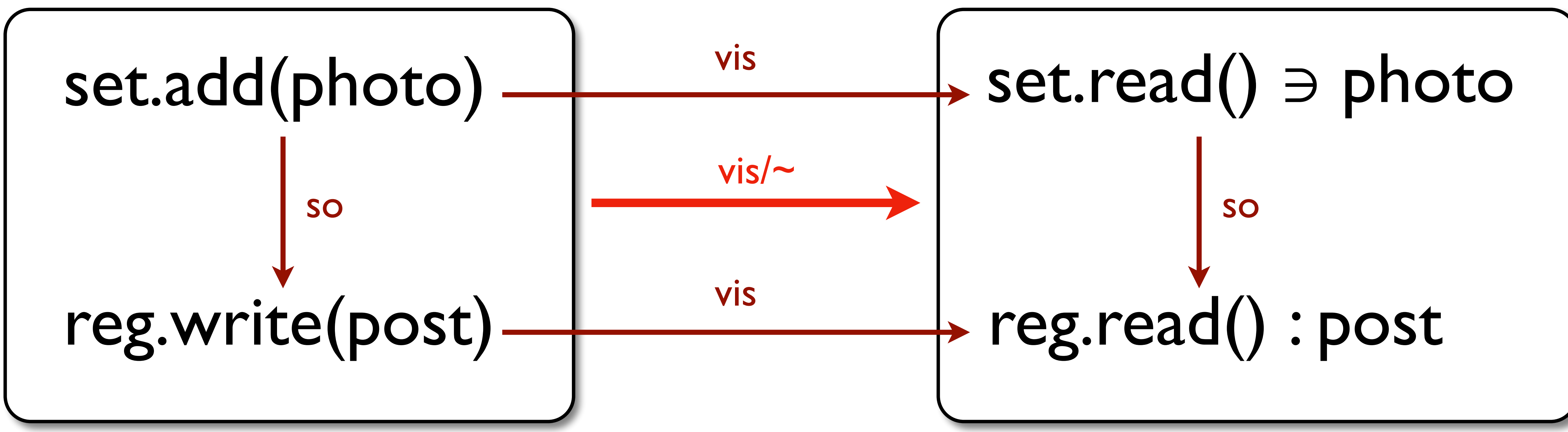


vis, ar treat events in a transaction uniformly:

$$\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\text{vis}} f \sim f' \implies e' \xrightarrow{\text{vis}} f'$$

$$\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\text{ar}} f \sim f' \implies e' \xrightarrow{\text{ar}} f'$$

Execution: (E, so, ~, vis, ar)



vis, ar induce acyclic vis/~, ar/~ on whole txs:

$$T \xrightarrow{\mathrm{vis}/\sim} S \iff \exists e \in T, f \in S.\ e \xrightarrow{\mathrm{vis}} f$$

$$T \xrightarrow{\mathrm{ar}/\sim} S \iff \exists e \in T, f \in S.\ e \xrightarrow{\mathrm{ar}} f$$

Eventually consistent transactions

The set of histories (E, so, ~) such that for some vis, ar:

- Return values consistent with data type specs:

  $\forall e \in E.\ rval(e) = F_{type(obj(e))}(context(e))$

- No causal cycles: so $\cup$ vis is acyclic

- Eventual visibility:

  $\forall e \in E.\ e \xrightarrow{vis} f$ for all but finitely many $f \in E$

- Transaction indivisibility:

  $\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{vis} f \sim f' \Longrightarrow e' \xrightarrow{vis} f'$

  $\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{ar} f \sim f' \Longrightarrow e' \xrightarrow{ar} f'$

Define transactional variants of other consistency models by just adding prior axioms

Serializability: vis = ar

...sistent transactions

..., so, ~) such that for some vis, ar:

- Return values consistent with data type specs:

  $\forall e \in E.\ \mathrm{rval}(e) = F_{\mathrm{type}(\mathrm{obj}(e))}(\mathrm{context}(e))$

- No causal cycles: so $\cup$ vis is acyclic

- Eventual visibility:

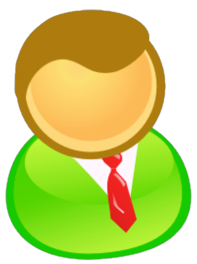
  $\forall e \in E.\ e \xrightarrow{\mathrm{vis}} f$ for all but finitely many $f \in E$

- Transaction indivisibility:

  $\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\mathrm{vis}} f \sim f' \Longrightarrow e' \xrightarrow{\mathrm{vis}} f'$

  $\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{\mathrm{ar}} f \sim f' \Longrightarrow e' \xrightarrow{\mathrm{ar}} f'$

Session guarantees



set.add(photo)

$\downarrow$ so

reg.write(post)

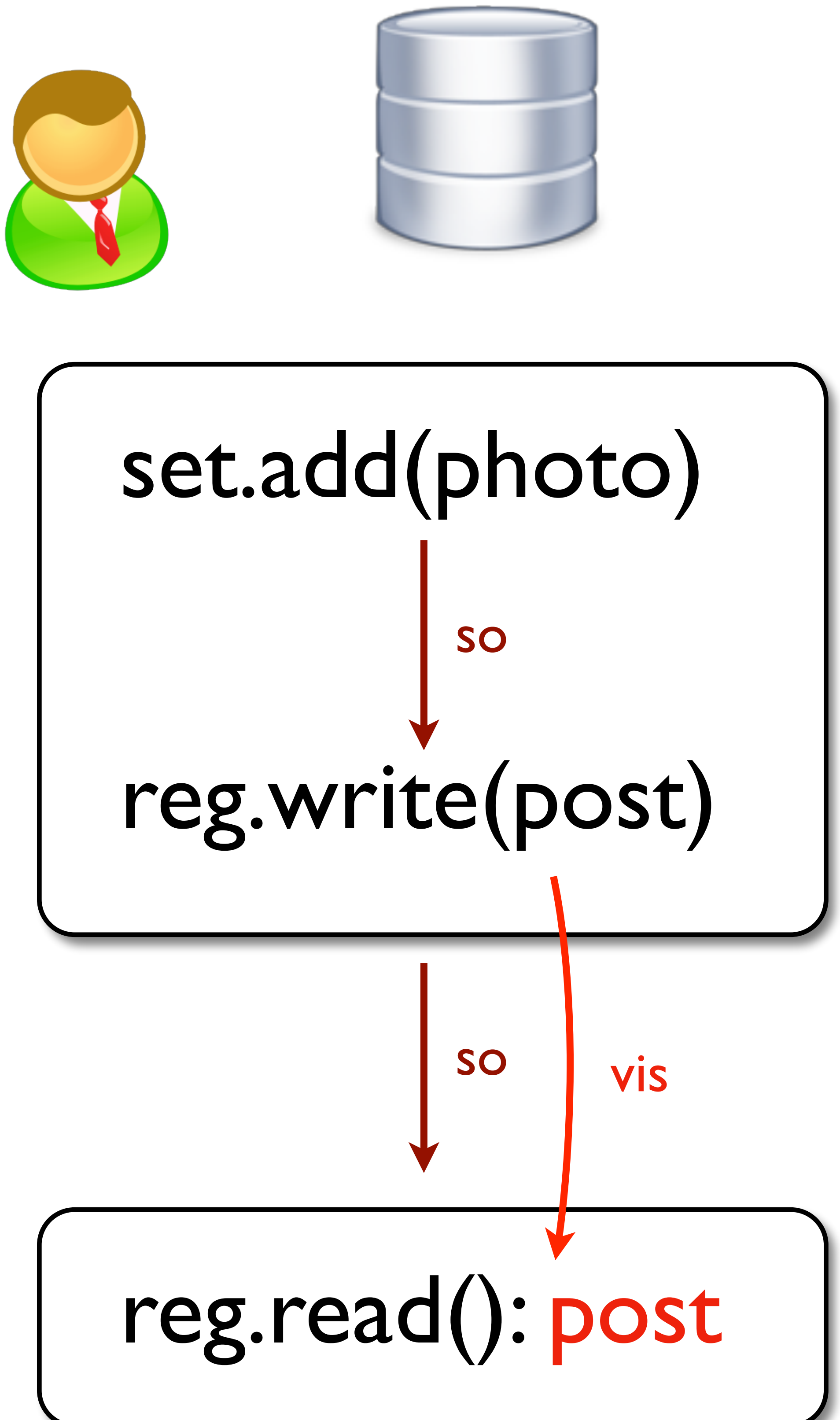
$\downarrow$ so

reg.read(): ?

so ⊆ vis

Transactions in the same session only accumulate information

Session guarantees



$so \subseteq vis$

Transactions in the same session only accumulate information

Causal consistency

$$\boxed{(so \cup vis)^+ \subseteq vis}$$

Causal consistency

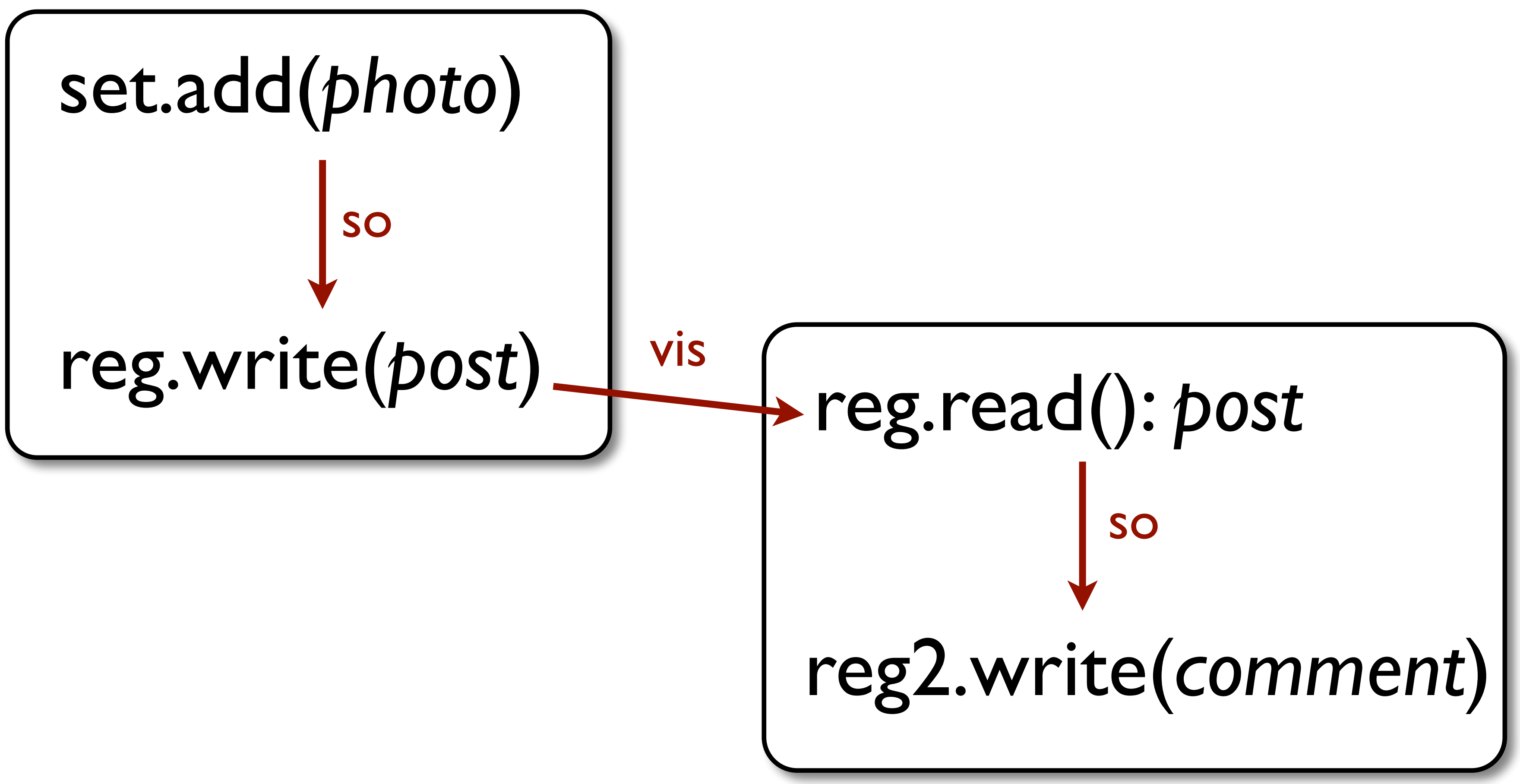
set.add(*photo*)

$\big\downarrow$ so

reg.write(*post*)

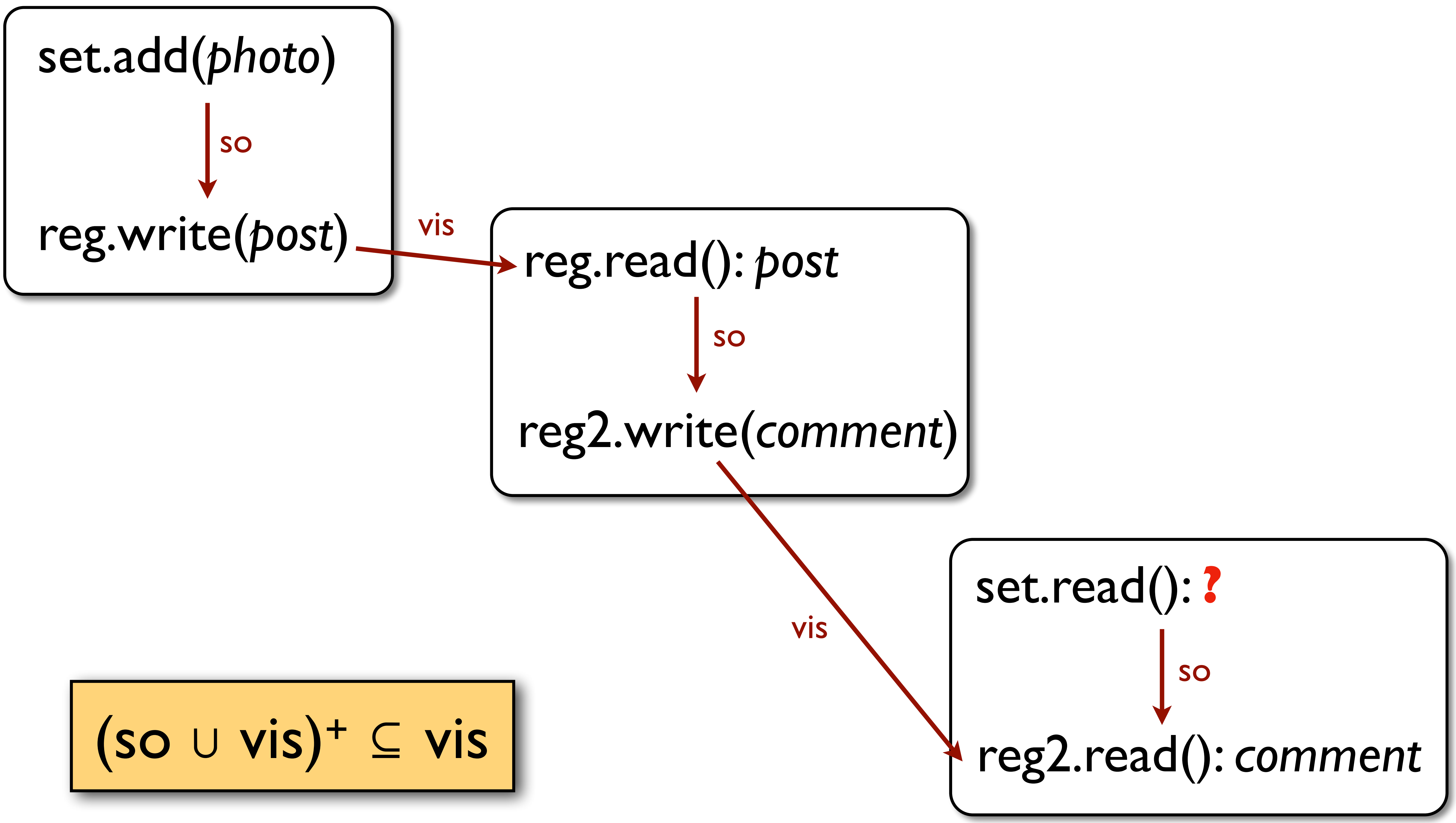
$(\mathrm{so} \cup \mathrm{vis})^+ \subseteq \mathrm{vis}$

Causal consistency



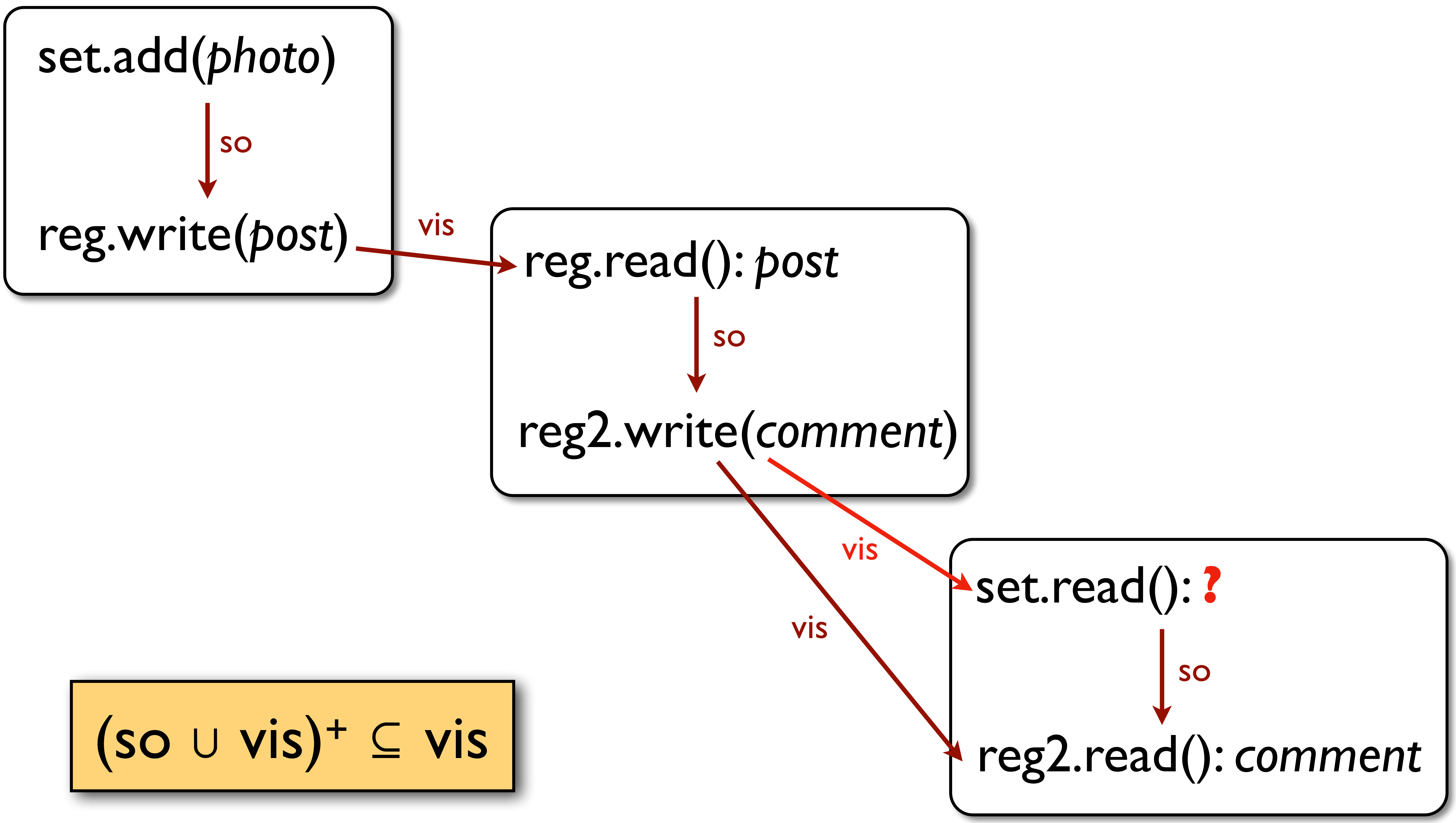
$$(\mathsf{so} \cup \mathsf{vis})^+ \subseteq \mathsf{vis}$$

Causal consistency

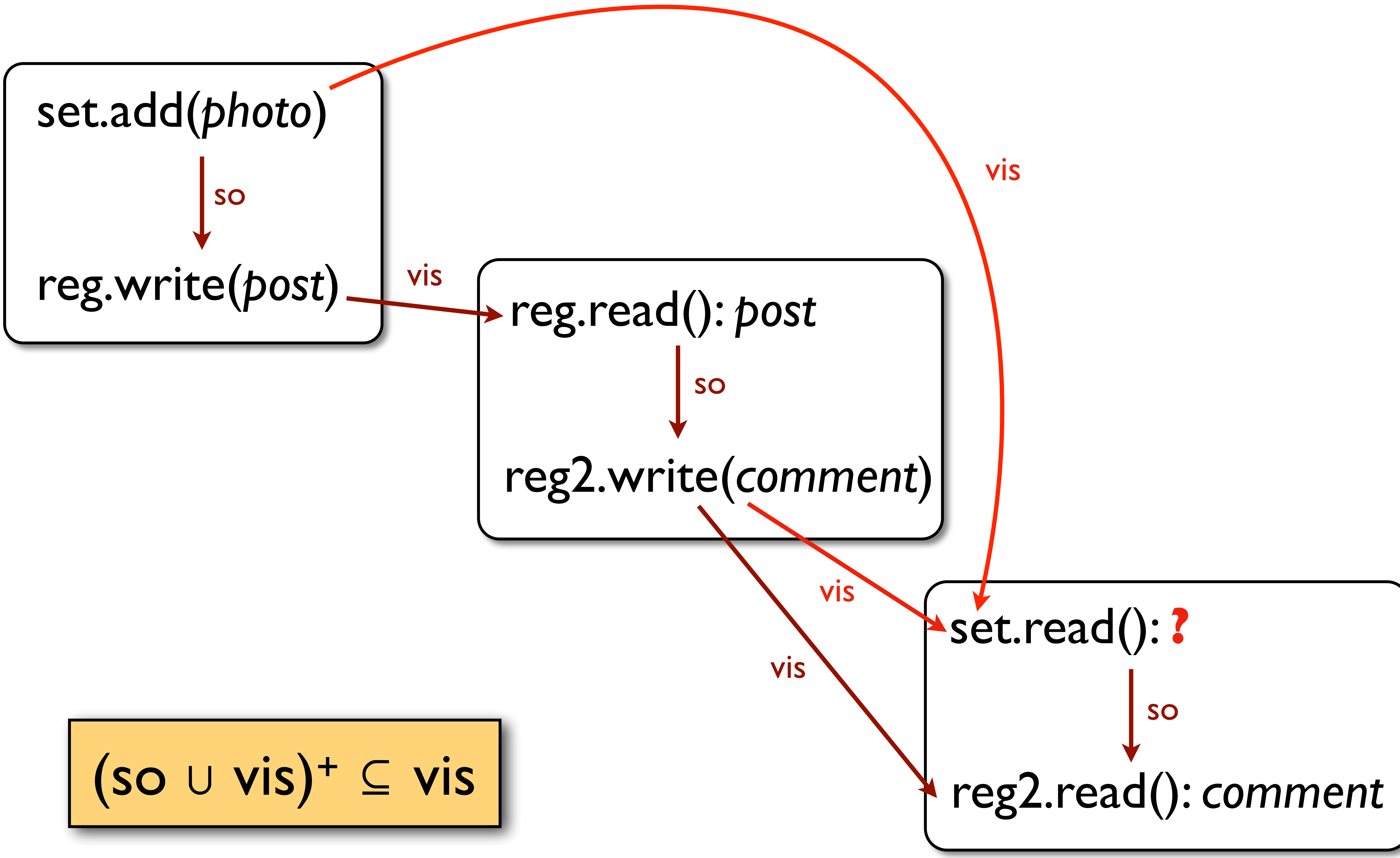


$$(\mathrm{so} \cup \mathrm{vis})^+ \subseteq \mathrm{vis}$$

Causal consistency

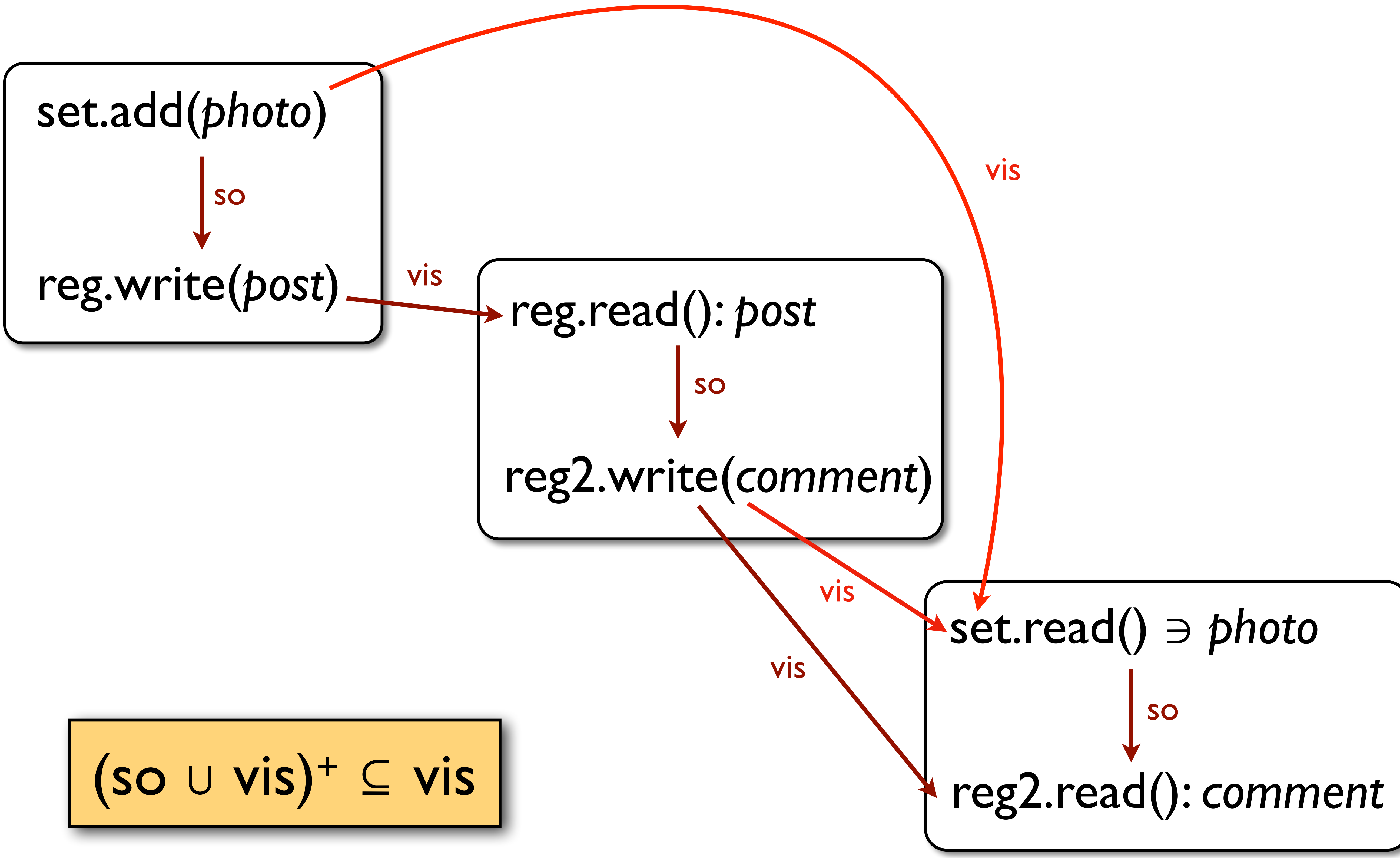


Causal consistency



$(\mathrm{so} \cup \mathrm{vis})^+ \subseteq \mathrm{vis}$

Causal consistency



$$(\mathsf{so} \cup \mathsf{vis})^+ \subseteq \mathsf{vis}$$

Concurrent withdrawals

c: counter with decrements, initially 100

v = c.read()
if (v≥100) so
  c.subtract(100)

v = c.read()
if (v≥100) so
  c.subtract(100)

Concurrent withdrawals

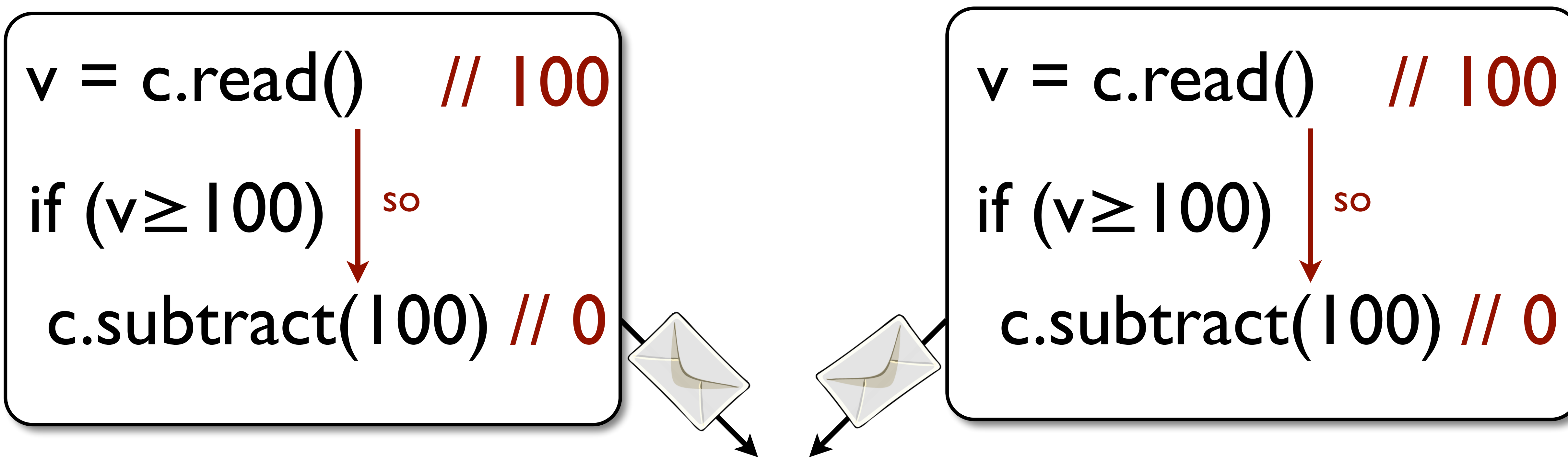
c: counter with decrements, initially 100

```
v = c.read()      // 100
if (v≥100)        so
  c.subtract(100) // 0
```

```
v = c.read()      // 100
if (v≥100)        so
  c.subtract(100) // 0
```

Concurrent withdrawals

c: counter with decrements, initially 100



Both transactions decremented successfully - synchronisation needed!

Recap: strengthening consistency

withdraw(100) : ✔

withdraw(100) : ✔

- Baseline model: causal consistency
- Symmetric conflict relation on operations:
  $\bowtie \subseteq \mathsf{Op} \times \mathsf{Op}$, e.g., $\mathsf{withdraw} \bowtie \mathsf{withdraw}$
- Conflicting operations cannot be causally independent:

  $\forall e, f \in E.\ op(e) \bowtie op(f) \implies e \xrightarrow{vis} f \lor f \xrightarrow{vis} e$

Recap: strengthening consistency

$$\text{withdraw(100)} : \checkmark \xrightarrow{\text{vis}} \text{withdraw(100)} : \text{✗}$$

- Baseline model: causal consistency
- Symmetric conflict relation on operations:
  $\bowtie \subseteq \mathrm{Op} \times \mathrm{Op}$, e.g., $\mathrm{withdraw} \bowtie \mathrm{withdraw}$
- Conflicting operations cannot be causally independent:

  $\forall e, f \in E.\ op(e) \bowtie op(f) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e$

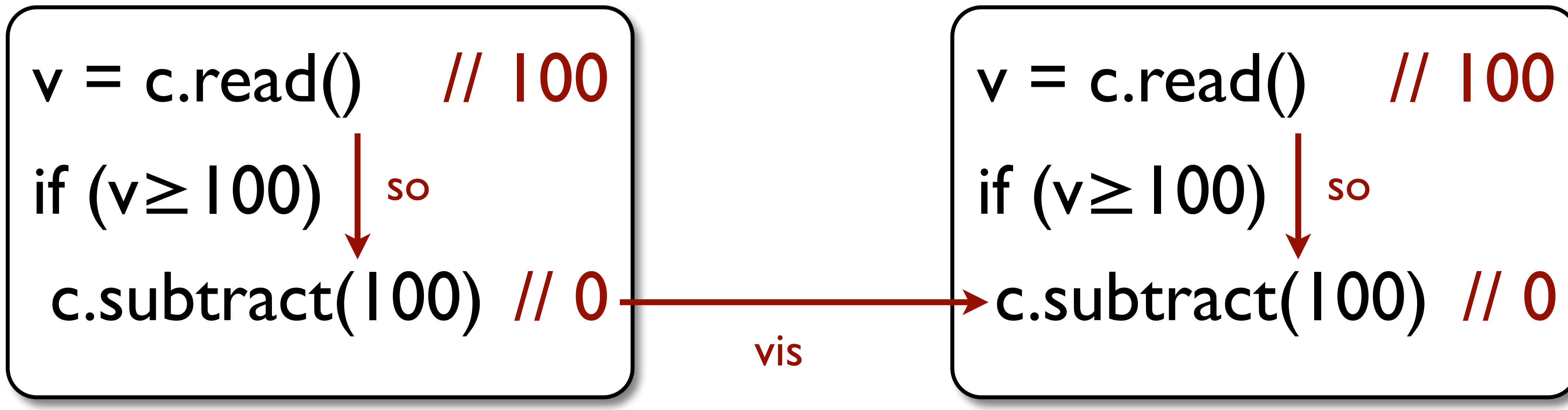
Strengthening transactions

v = c.read()   // 100
if (v≥100) $\downarrow$ so
  c.subtract(100) // 0
	v = c.read() // 100
	if (v≥100) $\downarrow$ so
	  c.subtract(100) // 0

- Baseline model: causal consistency
- Symmetric conflict relation on operations:
  $\bowtie \subseteq \mathsf{Op} \times \mathsf{Op}$, e.g., subtract $\bowtie$ subtract
- Conflicting operations cannot be causally independent:

  $\forall e, f \in E.\ \mathrm{op}(e) \bowtie \mathrm{op}(f) \implies e \xrightarrow{\mathsf{vis}} f \vee f \xrightarrow{\mathsf{vis}} e$

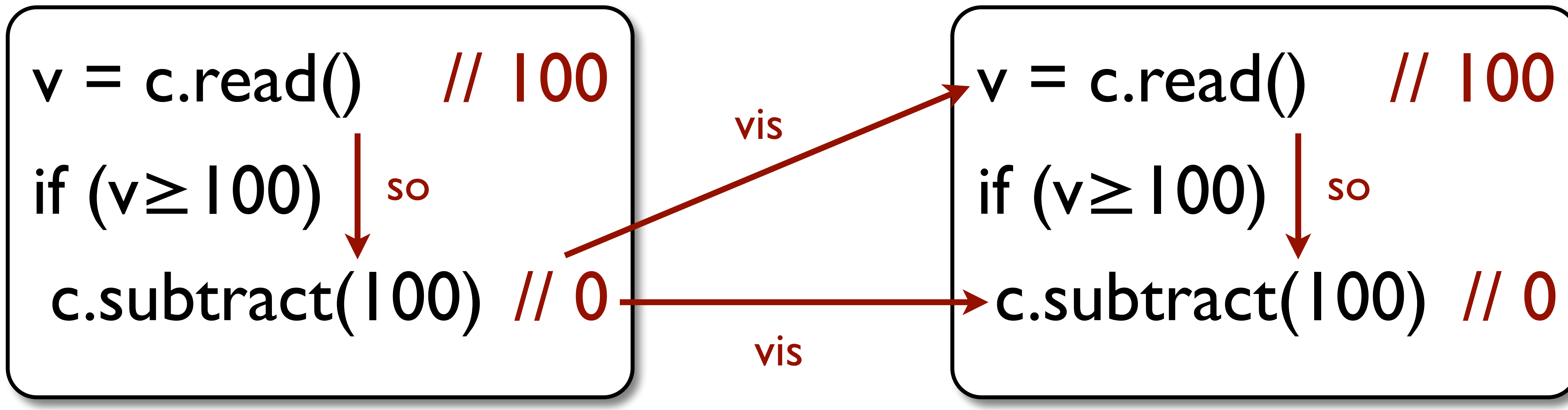
Strengthening transactions



- Baseline model: causal consistency
- Symmetric conflict relation on operations:
  $\bowtie\ \subseteq \mathrm{Op} \times \mathrm{Op}$, e.g., subtract $\bowtie$ subtract
- Conflicting operations cannot be causally independent:

  $\forall\, e, f \in E.\ \mathrm{op}(e) \bowtie \mathrm{op}(f) \implies e \xrightarrow{\mathrm{vis}} f \lor f \xrightarrow{\mathrm{vis}} e$

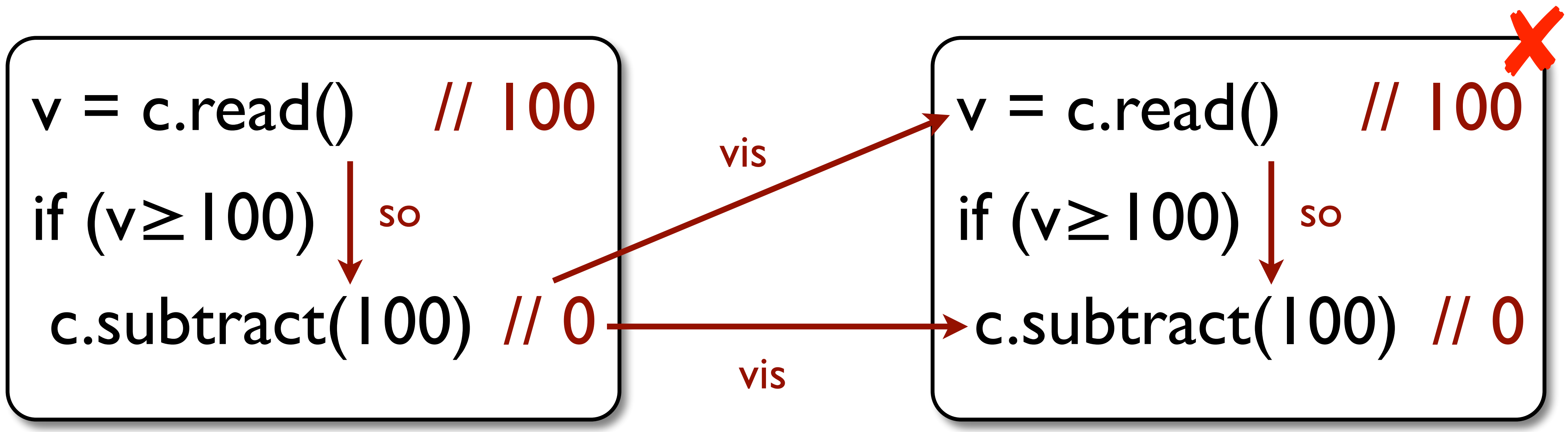
Strengthening transactions



- Baseline model: causal consistency
- Symmetric conflict relation on operations:
  $\bowtie \subseteq \mathrm{Op} \times \mathrm{Op}$, e.g., subtract $\bowtie$ subtract
- Conflicting operations cannot be causally independent:

  $\forall\, e, f \in E.\ \mathrm{op}(e) \bowtie \mathrm{op}(f) \implies e \xrightarrow{\mathrm{vis}} f \vee f \xrightarrow{\mathrm{vis}} e$

Strengthening transactions



- Baseline model: causal consistency
- Symmetric conflict relation on operations: $\bowtie \subseteq \mathrm{Op} \times \mathrm{Op}$, e.g., subtract $\bowtie$ subtract
- Conflicting operations cannot be causally independent:

$$\forall e, f \in E.\ \mathrm{op}(e) \bowtie \mathrm{op}(f) \Longrightarrow e \xrightarrow{\mathrm{vis}} f \vee f \xrightarrow{\mathrm{vis}} e$$

Strengthening transactions

c.add(100)

c.add(100)

$\neg(\text{add} \bowtie \text{op})$

- Baseline model: causal consistency
- Symmetric conflict relation on operations:
  $\bowtie \subseteq \text{Op} \times \text{Op}$, e.g., subtract $\bowtie$ subtract
- Conflicting operations cannot be causally independent:

  $\forall\, e, f \in E.\ \text{op}(e) \bowtie \text{op}(f) \implies e \xrightarrow{\text{vis}} f \vee f \xrightarrow{\text{vis}} e$

Recap: implementation

c.withdraw(100) : ✔

c.withdraw(100) : ?

- withdraw $\bowtie$ withdraw: as if withdraw grabs an exclusive lock on the account
- Acquiring the lock requires bringing all operations the replica holding it knows about

Recap: implementation

c.withdraw(100) : ✔ ⟶ c.withdraw(100) : ?

- withdraw $\bowtie$ withdraw: as if withdraw grabs an exclusive lock on the account
- Acquiring the lock requires bringing all operations the replica holding it knows about

Recap: implementation

- withdraw ⋈ withdraw: as if withdraw grabs an exclusive lock on the account
- Acquiring the lock requires bringing all operations the replica holding it knows about

Implementation for transactions

subtract ⋈ subtract

```
v = c.read()      // 100
if (v≥100)
 c.subtract(100)  // 0
```

✔

Implementation for transactions

subtract ⋈ subtract

```
v = c.read()      // 100
if (v≥100)
 c.subtract(100)  // 0
```

✔

Implementation for transactions

subtract $\bowtie$ subtract

```
v = c.read()      // 100
if (v≥100)
 c.subtract(100)  // 0
```

✔

```
v = c.read()      // 100
```

Implementation for transactions

subtract $\bowtie$ subtract

```
v = c.read()      // 100
if (v≥100)
 c.subtract(100)  // 0
```

✔

```
v = c.read()      // 100
if (v≥100)
```

Implementation for transactions

subtract ⋈ subtract

```
v = c.read()        // 100
if (v≥100)
  c.subtract(100)   // 0
```

✔

```
v = c.read()        // 100
if (v≥100)
  c.subtract(100)
```

Implementation for transactions

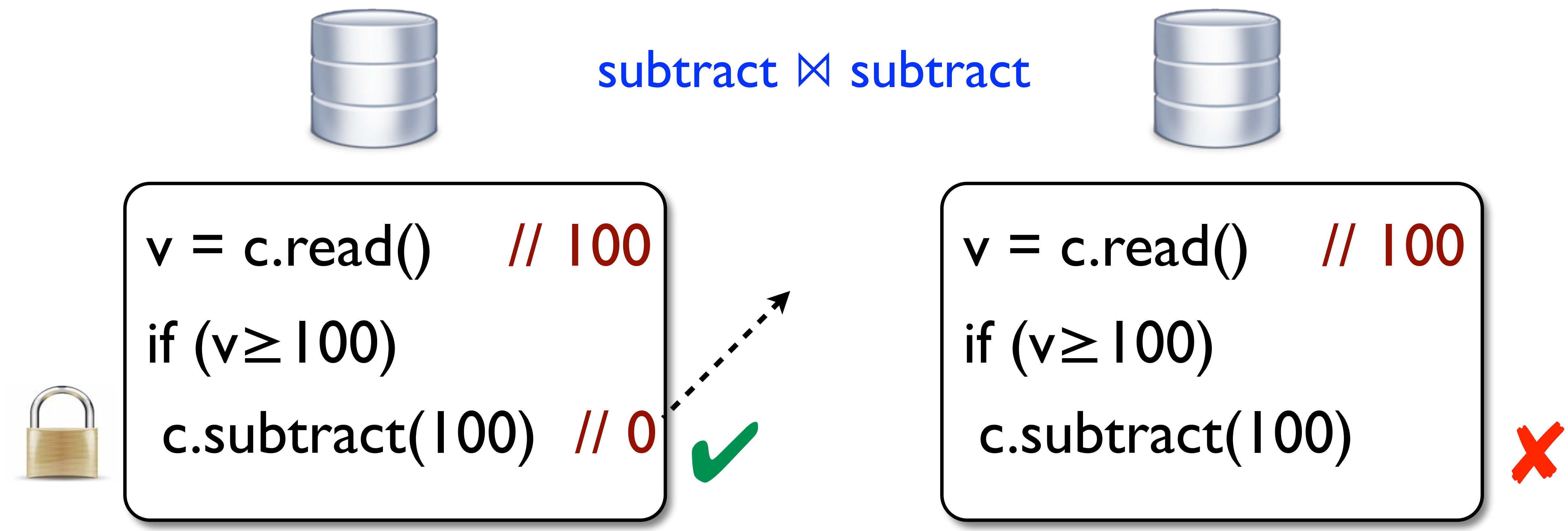
subtract ⋈ subtract

```
v = c.read()      // 100
if (v≥100)
 c.subtract(100)  // 0
```

```
v = c.read()      // 100
if (v≥100)
 c.subtract(100)
```

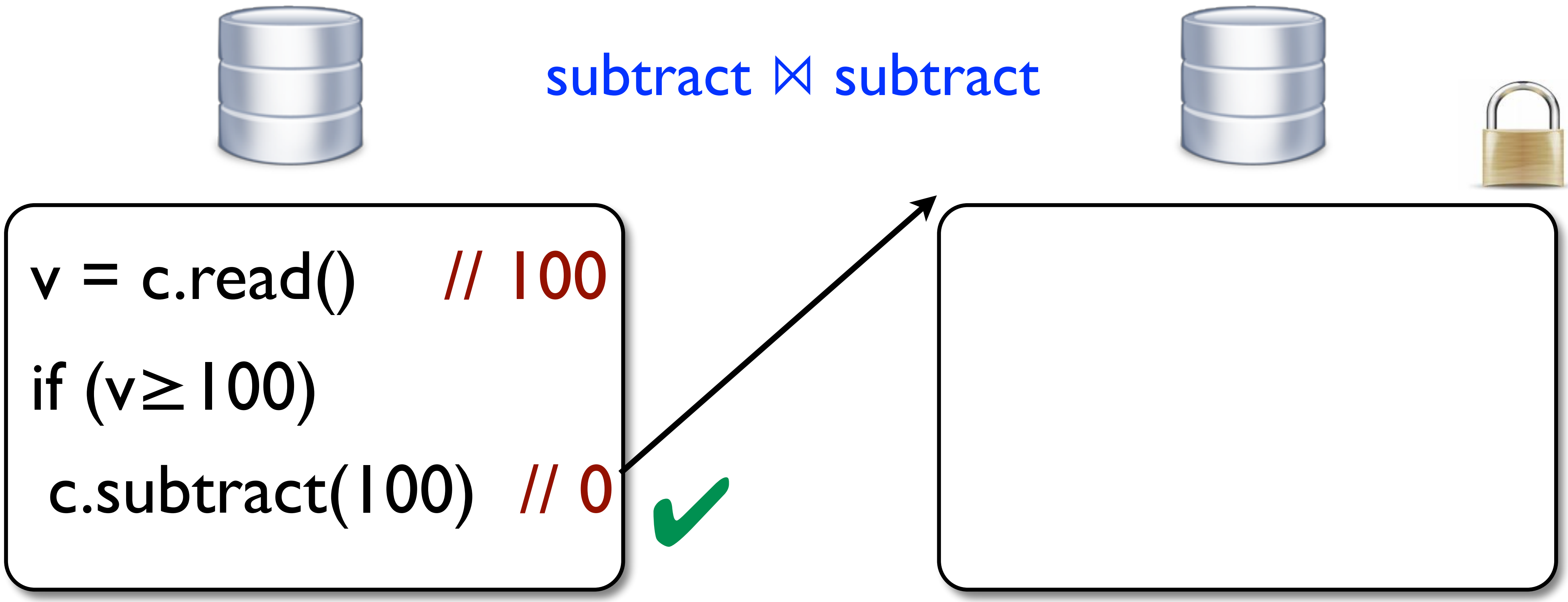
- Need to incorporate the effector of the previous transaction

Implementation for transactions



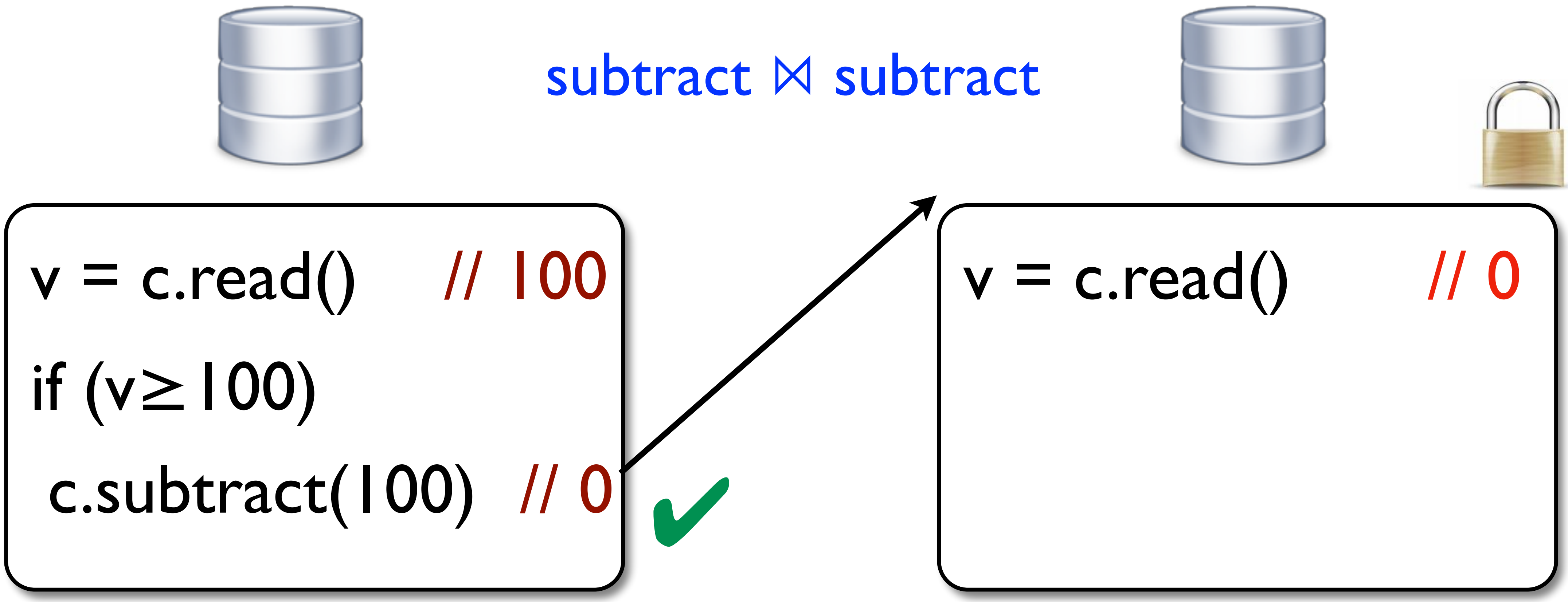
- Need to incorporate the effector of the previous transaction
- Recall: transactions execute on a fixed snapshot
- Too late: effectors from other replicas only get applied in-between transactions
- Have to abort the transaction and re-execute it

Implementation for transactions



- Need to incorporate the effector of the previous transaction
- Recall: transactions execute on a fixed snapshot
- Too late: effectors from other replicas only get applied in-between transactions
- Have to abort the transaction and re-execute it

Implementation for transactions



- Need to incorporate the effector of the previous transaction
- Recall: transactions execute on a fixed snapshot
- Too late: effectors from other replicas only get applied in-between transactions
- Have to abort the transaction and re-execute it

Chosing $\bowtie$

- Want to choose $\bowtie$ to preserve application invariants
- Previous proof rule for checking invariants applies
- Instead of an effector of a single operation, consider a sequential composition of effectors of all operations in a transaction
- Can also fix $\bowtie$ so that it's easier to program: new consistency models, disallowing some classes of anomalies

Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

  $\forall e, f \in E.\ \mathrm{obj}(e) = \mathrm{obj}(f) \wedge \mathrm{update}(\mathrm{op}(e)) \wedge \mathrm{update}(\mathrm{op}(f))$
  $\implies e \xrightarrow{\mathrm{vis}} f \vee f \xrightarrow{\mathrm{vis}} e$

Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

  $\forall e, f \in E.\ \mathrm{obj}(e) = \mathrm{obj}(f) \wedge \mathrm{update}(\mathrm{op}(e)) \wedge \mathrm{update}(\mathrm{op}(f))$
  $\implies e \xrightarrow{\mathrm{vis}} f \vee f \xrightarrow{\mathrm{vis}} e$

- No overdrafts:

```
v = c.read()        // 100
if (v≥100)          so
  c.subtract(100)   // 0
```

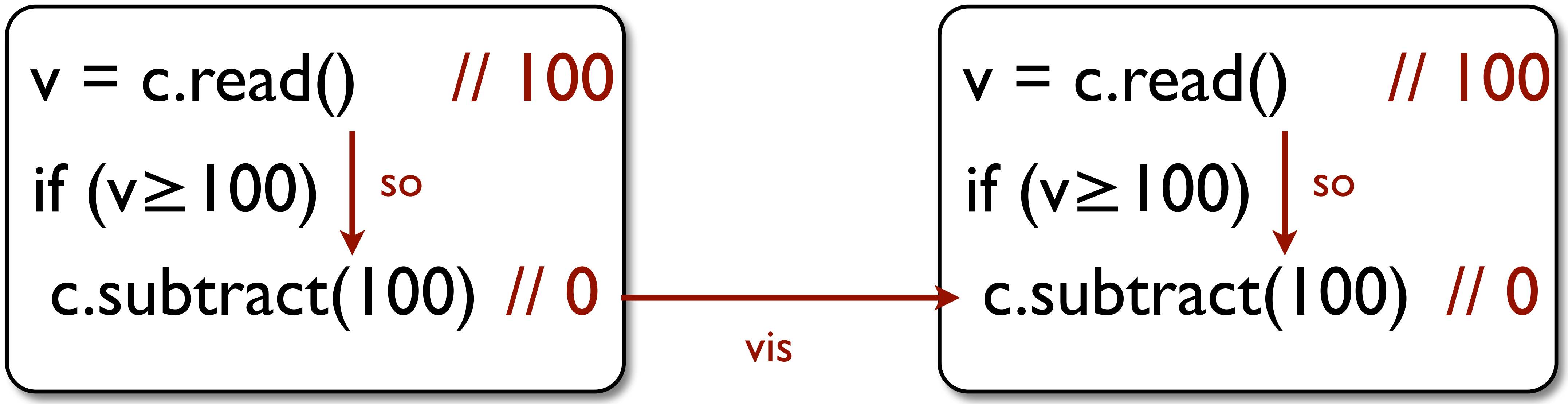
```
v = c.read()        // 100
if (v≥100)          so
  c.subtract(100)   // 0
```

Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

  $\forall e, f \in E.\ \mathrm{obj}(e) = \mathrm{obj}(f) \wedge \mathrm{update}(\mathrm{op}(e)) \wedge \mathrm{update}(\mathrm{op}(f))$
  $\Longrightarrow e \xrightarrow{\mathrm{vis}} f \vee f \xrightarrow{\mathrm{vis}} e$

- No overdrafts:

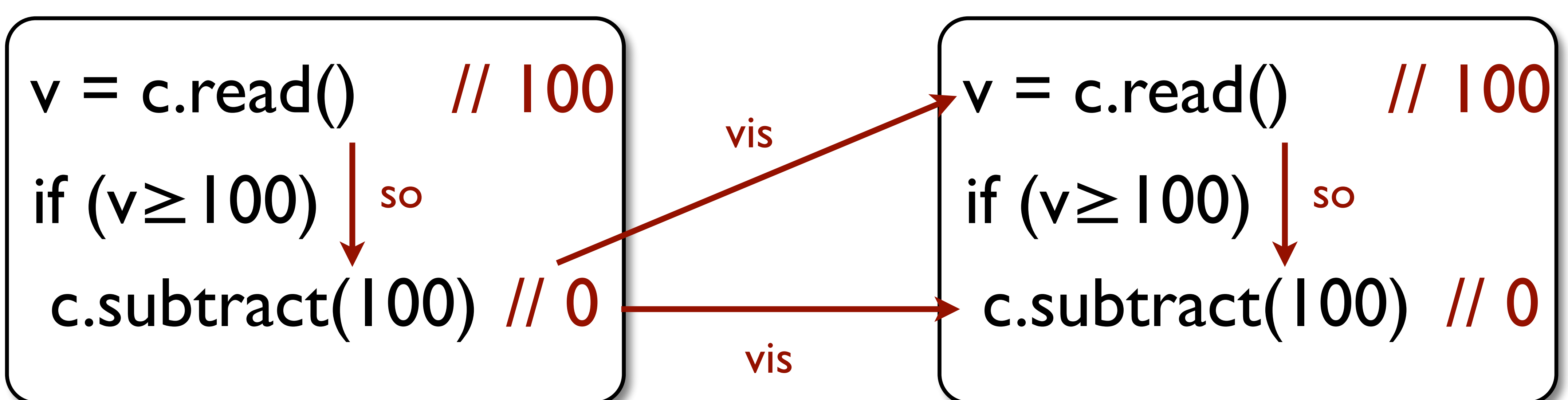


Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

$$\forall e, f \in E.\ \mathrm{obj}(e) = \mathrm{obj}(f) \wedge \mathrm{update}(\mathrm{op}(e)) \wedge \mathrm{update}(\mathrm{op}(f))$$
$$\Longrightarrow e \xrightarrow{\mathrm{vis}} f \vee f \xrightarrow{\mathrm{vis}} e$$

- No overdrafts:

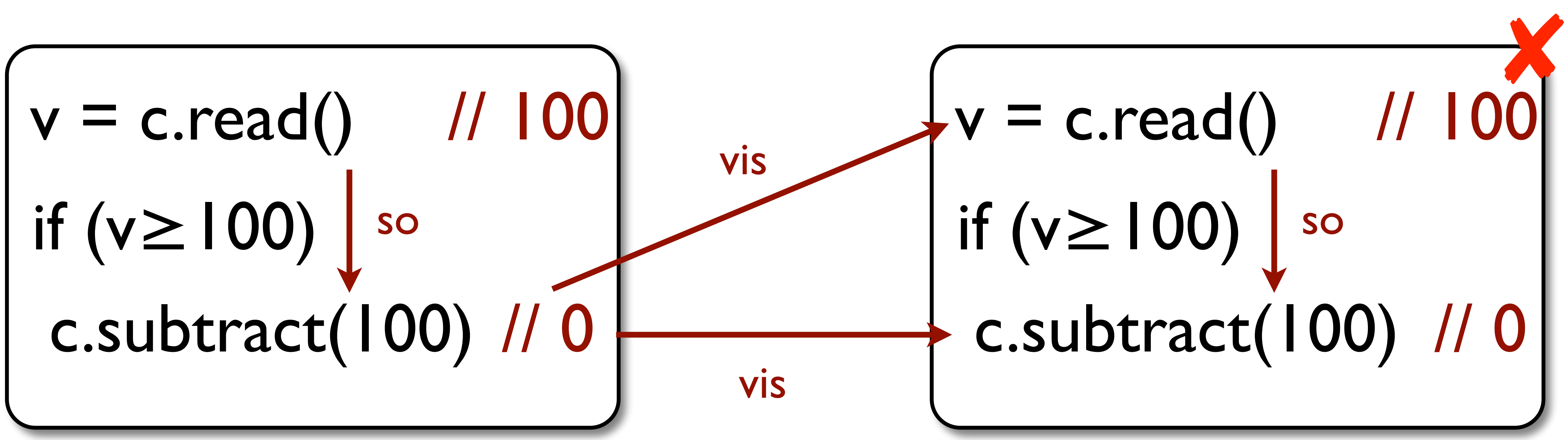


Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

  $\forall e, f \in E.\ \mathrm{obj}(e) = \mathrm{obj}(f) \land \mathrm{update}(\mathrm{op}(e)) \land \mathrm{update}(\mathrm{op}(f)) \implies e \xrightarrow{\mathrm{vis}} f \lor f \xrightarrow{\mathrm{vis}} e$

- No overdrafts:



Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

  $\forall e, f \in E.\ \mathrm{obj}(e) = \mathrm{obj}(f) \wedge \mathrm{update}(\mathrm{op}(e)) \wedge \mathrm{update}(\mathrm{op}(f))$
  $\Longrightarrow e \xrightarrow{\mathrm{vis}} f \vee f \xrightarrow{\mathrm{vis}} e$

- No lost updates:

$$v = \mathrm{reg.read}() \quad // \ 0 \ \Big\downarrow \mathrm{so} \ \mathrm{reg.write}(v+1) \quad // \ 1$$

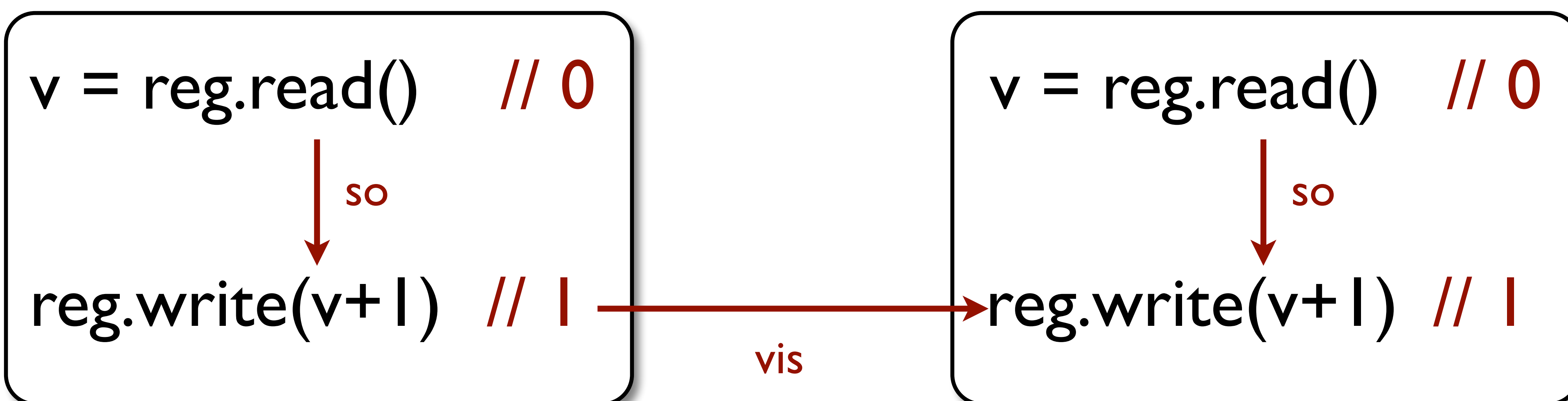
$$v = \mathrm{reg.read}() \quad // \ 0 \ \Big\downarrow \mathrm{so} \ \mathrm{reg.write}(v+1) \quad // \ 1$$

Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

  $\forall e, f \in E.\ \mathrm{obj}(e) = \mathrm{obj}(f) \wedge \mathrm{update}(\mathrm{op}(e)) \wedge \mathrm{update}(\mathrm{op}(f))$
  $\Longrightarrow e \xrightarrow{\mathrm{vis}} f \vee f \xrightarrow{\mathrm{vis}} e$

- No lost updates:

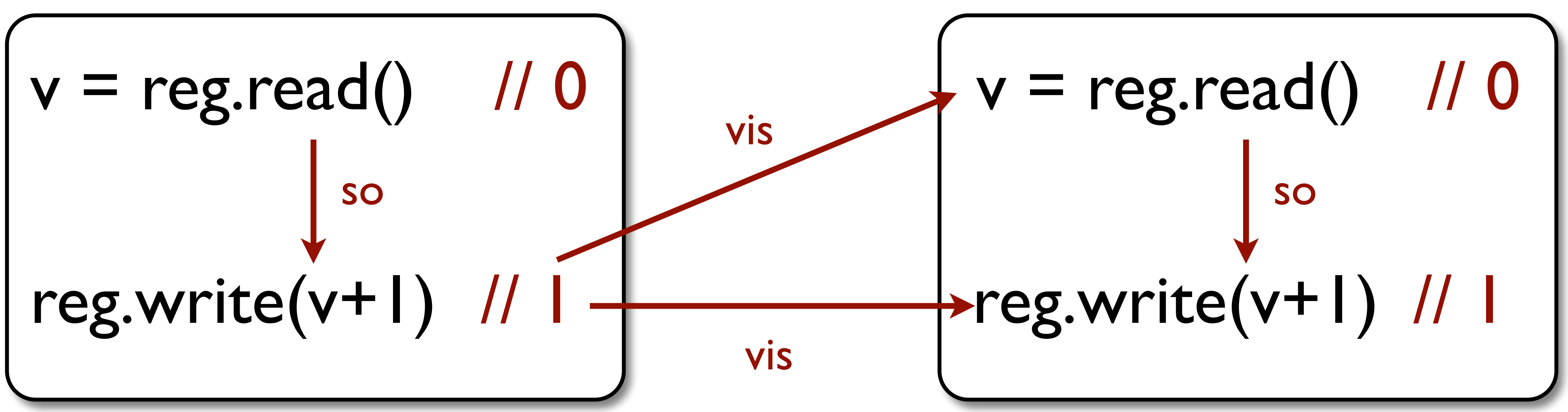


Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

  $$\forall e, f \in E.\ \mathrm{obj}(e) = \mathrm{obj}(f) \wedge \mathrm{update}(\mathrm{op}(e)) \wedge \mathrm{update}(\mathrm{op}(f))$$
  $$\Longrightarrow e \xrightarrow{\text{vis}} f \vee f \xrightarrow{\text{vis}} e$$

- No lost updates:

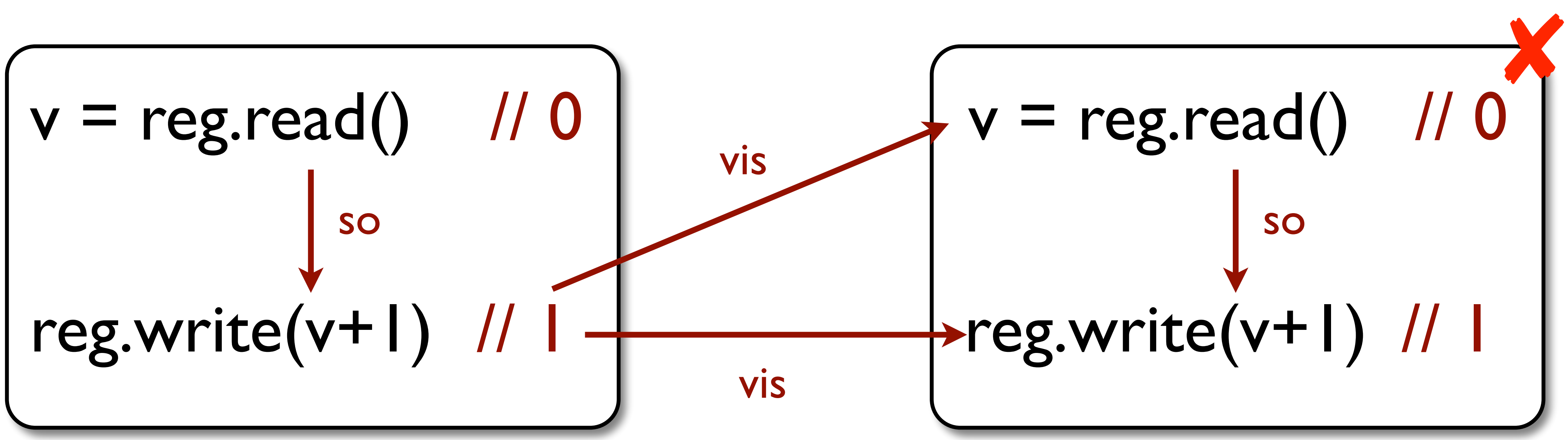


Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

  $\forall e, f \in E.\ obj(e) = obj(f) \wedge update(op(e)) \wedge update(op(f))$
  $\Longrightarrow e \xrightarrow{vis} f \vee f \xrightarrow{vis} e$

- No lost updates:



Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

  $\forall e, f \in E.\ \mathsf{obj}(e) = \mathsf{obj}(f) \wedge \mathsf{update}(\mathsf{op}(e)) \wedge \mathsf{update}(\mathsf{op}(f))$
  $\implies e \xrightarrow{\mathsf{vis}} f \vee f \xrightarrow{\mathsf{vis}} e$

- Updates on different accounts can go in parallel:

v = reg.read() // 0

$\downarrow$ so

reg.write(v+1) // 1

v = reg'.read() // 0

$\downarrow$ so

reg'.write(v+1) // 1

Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

  $$\forall e, f \in E.\ \mathrm{obj}(e) = \mathrm{obj}(f) \wedge \mathrm{update}(\mathrm{op}(e)) \wedge \mathrm{update}(\mathrm{op}(f)) \implies e \xrightarrow{\mathrm{vis}} f \vee f \xrightarrow{\mathrm{vis}} e$$

- Visibility totally orders transactions updating the same object $\implies$ don't need replicated data types, don't need ar

$$\begin{tikzcd}
\text{set.add(1)} \arrow[r, "\mathrm{vis}"] \arrow[dr, "\mathrm{vis}"'] & \text{set.remove(1)} \arrow[r, "\mathrm{vis}"] \arrow[d, "\mathrm{vis}"'] & \text{set.add(2)} \arrow[dl, "\mathrm{vis}"] \\
& \text{set.read(): \{2\}} &
\end{tikzcd}$$

Write-conflict detection

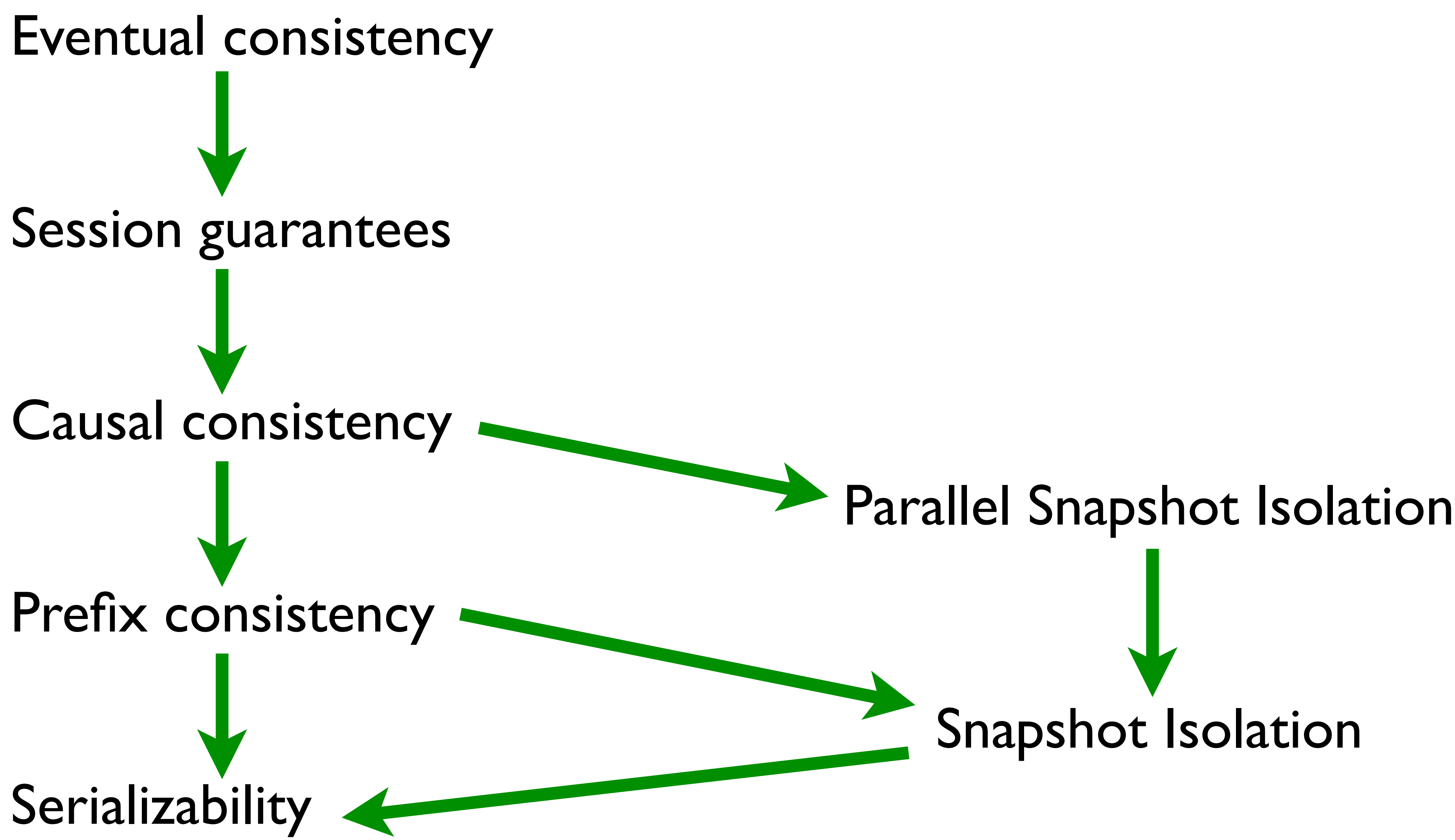
- Operations updating the same object conflict, so cannot be causally independent:

  $\forall e, f \in E.\ obj(e) = obj(f) \wedge update(op(e)) \wedge update(op(f))$
  $\implies e \xrightarrow{vis} f \vee f \xrightarrow{vis} e$

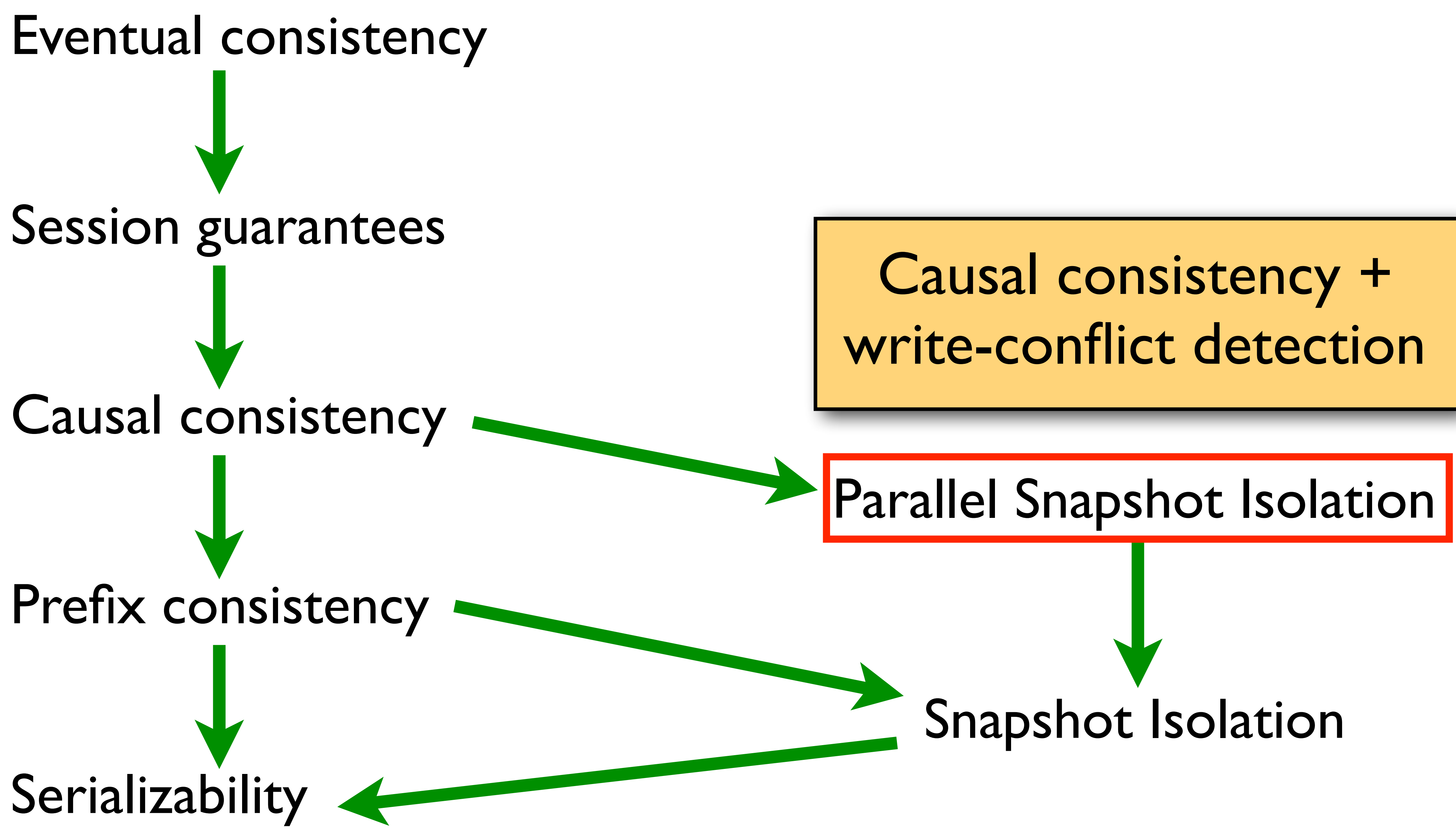
- Visibility totally orders transactions updating the same object $\implies$ don't need replicated data types, don't need ar

- Can use sequential data types: from now on just sequential read-write registers

Transactional consistency zoo



Transactional consistency zoo



Robustness

Application correctness

- Does an application satisfy a particular correctness property?

  Integrity invariants: account balance is non-negative

- Is an application robust against a particular consistency model?

  Application behaves the same as when using a strongly consistent database

Application correctness

- Does an application satisfy a particular correctness property?

  Integrity invariants: account balance is non-negative

- Is an application robust against a particular consistency model?

  Application behaves the same as when using a strongly consistent database

Parallel shapshot isolation

- Database with only sequential read-write registers
- Assume there is an implicit transaction writing initial values to all registers

PSI = the set of histories $(E, so, \sim)$ such that for some vis:

- No causal cycles: $so \cup vis$ is acyclic
- Eventual visibility: $\forall e \in E.\ e \xrightarrow{vis} f$ for all but finitely many $f \in E$
- Transaction indivisibility:

  $\forall e, f, e', f'.\ e \nsim f \wedge e' \sim e \xrightarrow{vis} f \sim f' \implies e' \xrightarrow{vis} f'$
- Causality preservation: $(so \cup vis)^+ \subseteq vis$
- Write-conflict detection:

  $\forall e, f \in E.\ obj(e) = obj(f) \wedge op(e) = write(-) \wedge op(f) = write(-)$
  $\implies e \xrightarrow{vis} f \vee f \xrightarrow{vis} e$
- A read event returns the value written by the last preceding write in vis

PSI = the set of histories $(E, so, \sim)$ such that for some vis:

- No causal cycles: $so \cup vis$ is acyclic
- Eventual visibility: $\forall e \in E.\ e \xrightarrow{vis} f$ for all but finitely many $f \in E$
- Transaction indivisibility:

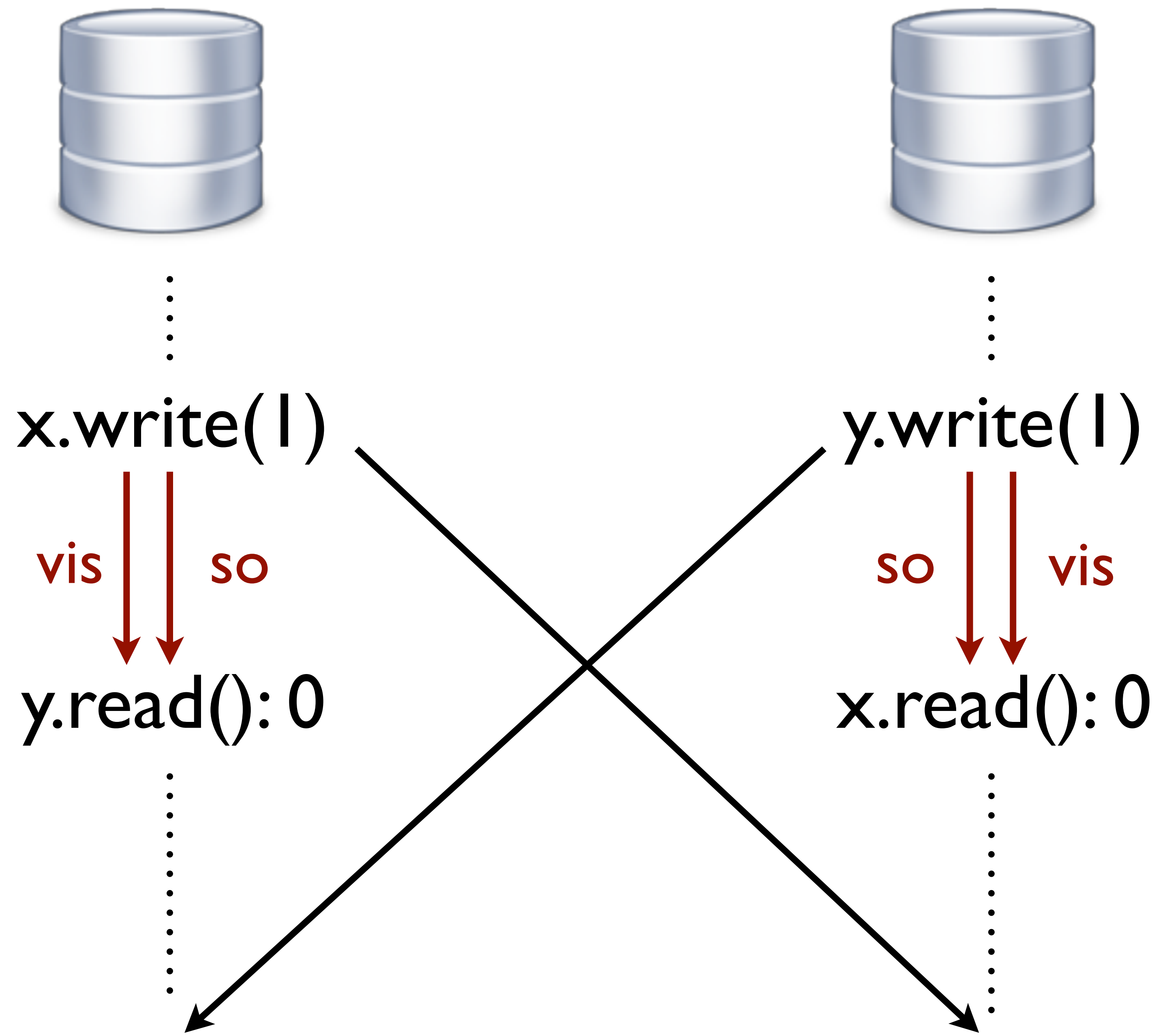
  $\forall e, f, e', f'.\ e \nsim f \land e' \sim e \xrightarrow{vis} f \sim f' \implies e' \xrightarrow{vis} f'$
- Causality preservation: $(so \cup vis)^+ \subseteq vis$
- Write-conflict detection:

  $\forall e, f \in E.\ obj(e) = obj(f) \land op(e) = write(-) \land op(f) = write(-)$
  $\implies e \xrightarrow{vis} f \lor f \xrightarrow{vis} e$
- A read event returns the value written by the last preceding write in vis

Well-formed because of write-conflict detection

Dekker example

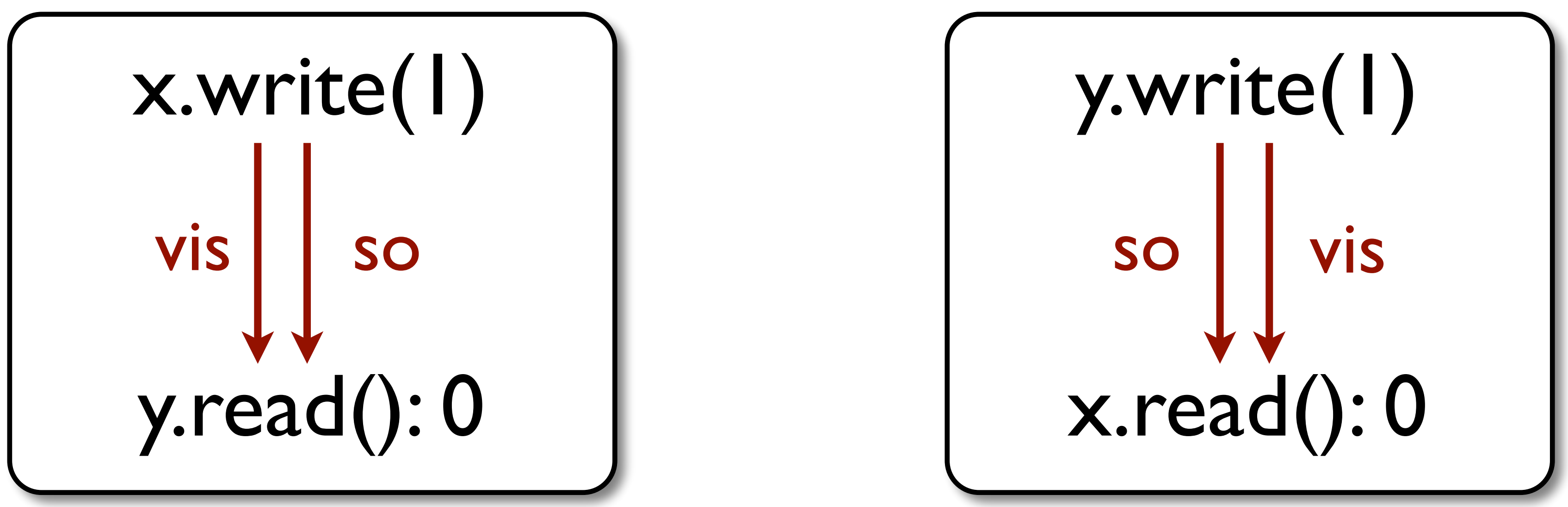
Dekker example



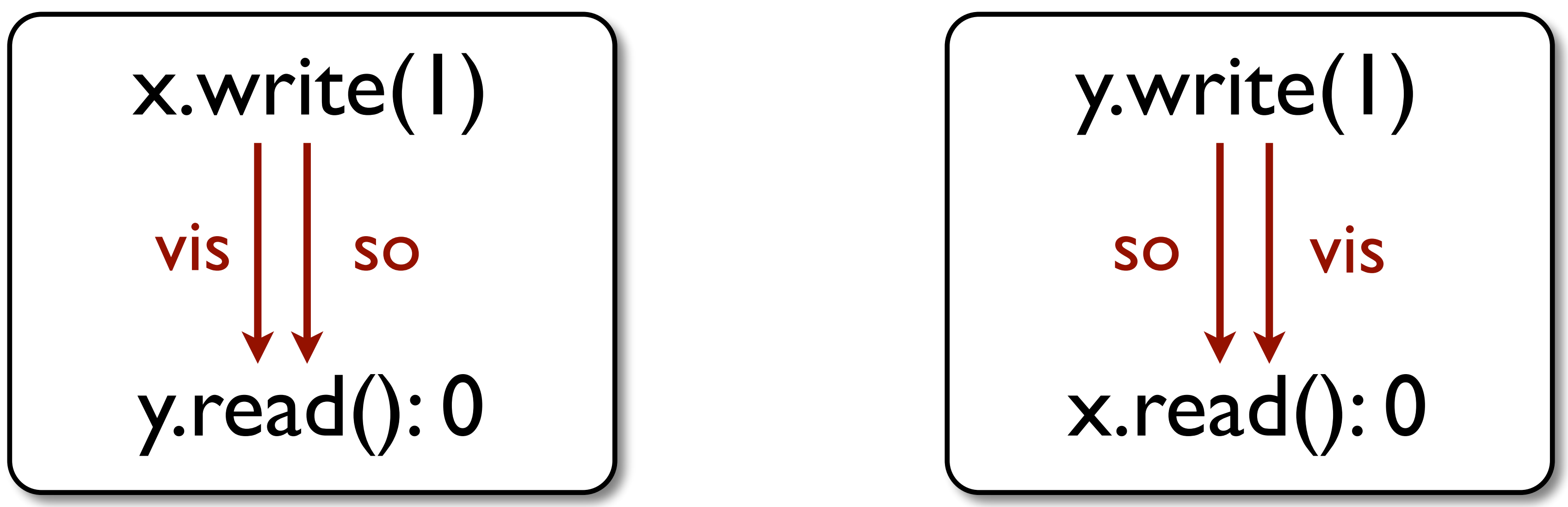
Dekker example

$$
\begin{tikzcd}
\text{x.write(1)} \arrow[d, shift right, "\text{vis}"'] \arrow[d, shift left, "\text{so}"] & & \text{y.write(1)} \arrow[d, shift right, "\text{so}"'] \arrow[d, shift left, "\text{vis}"] \\
\text{y.read(): 0} & & \text{x.read(): 0}
\end{tikzcd}
$$

Transactional Dekker = write skew

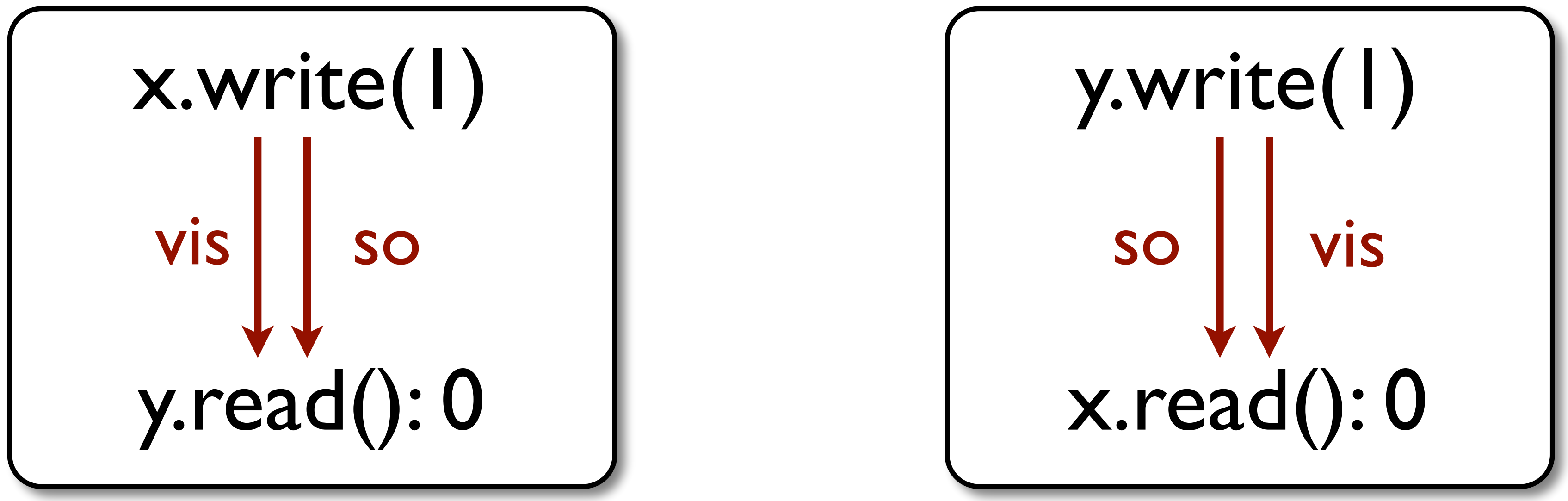


Transactional Dekker = write skew

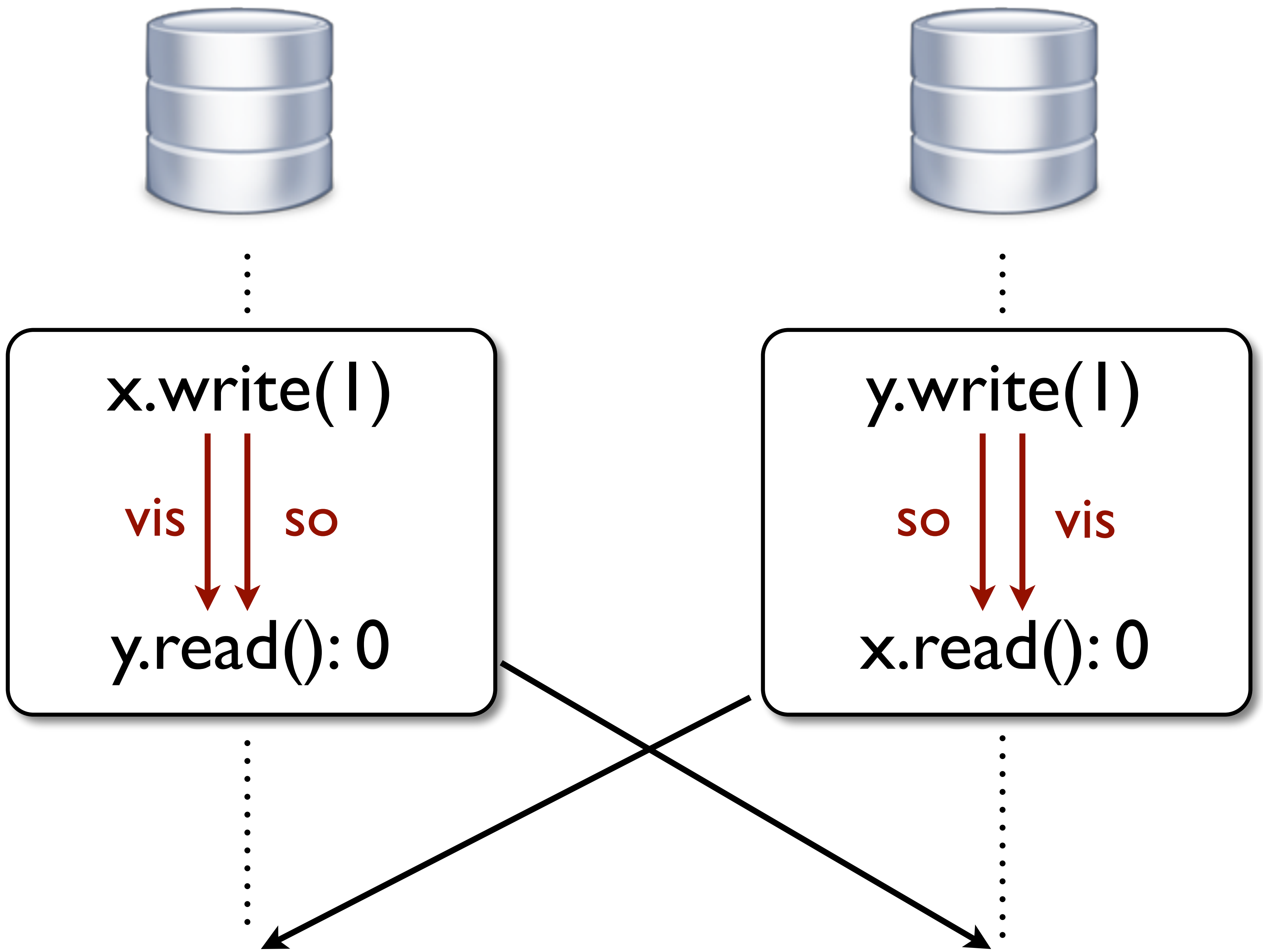


Not serializable, allowed by transactional causal consistency and parallel snapshot isolation

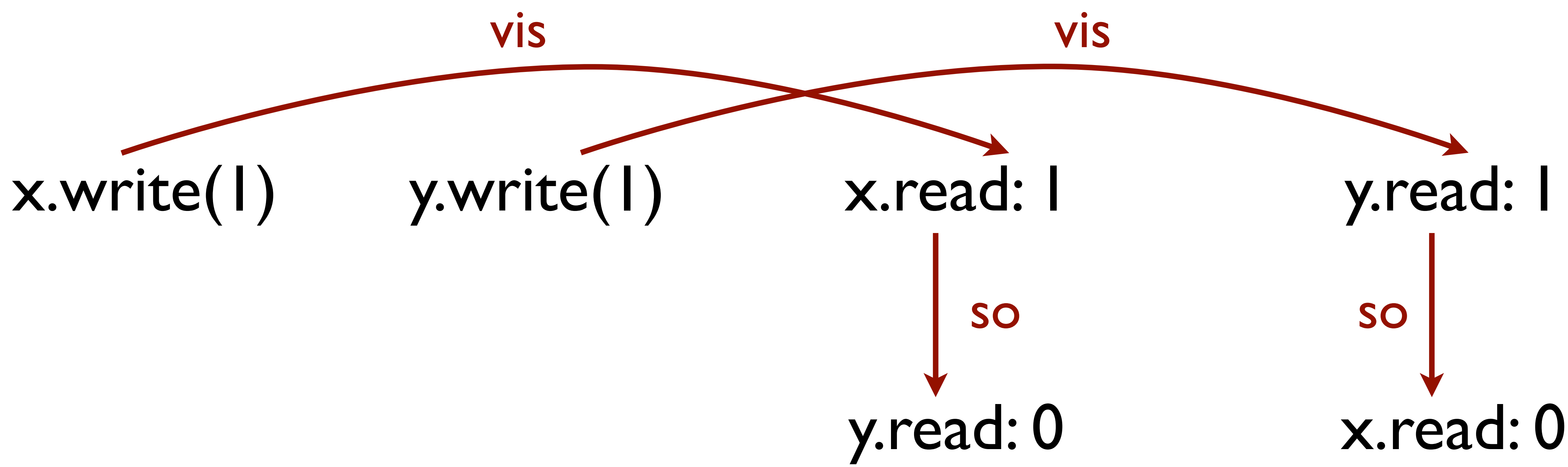
Transactional Dekker = write skew



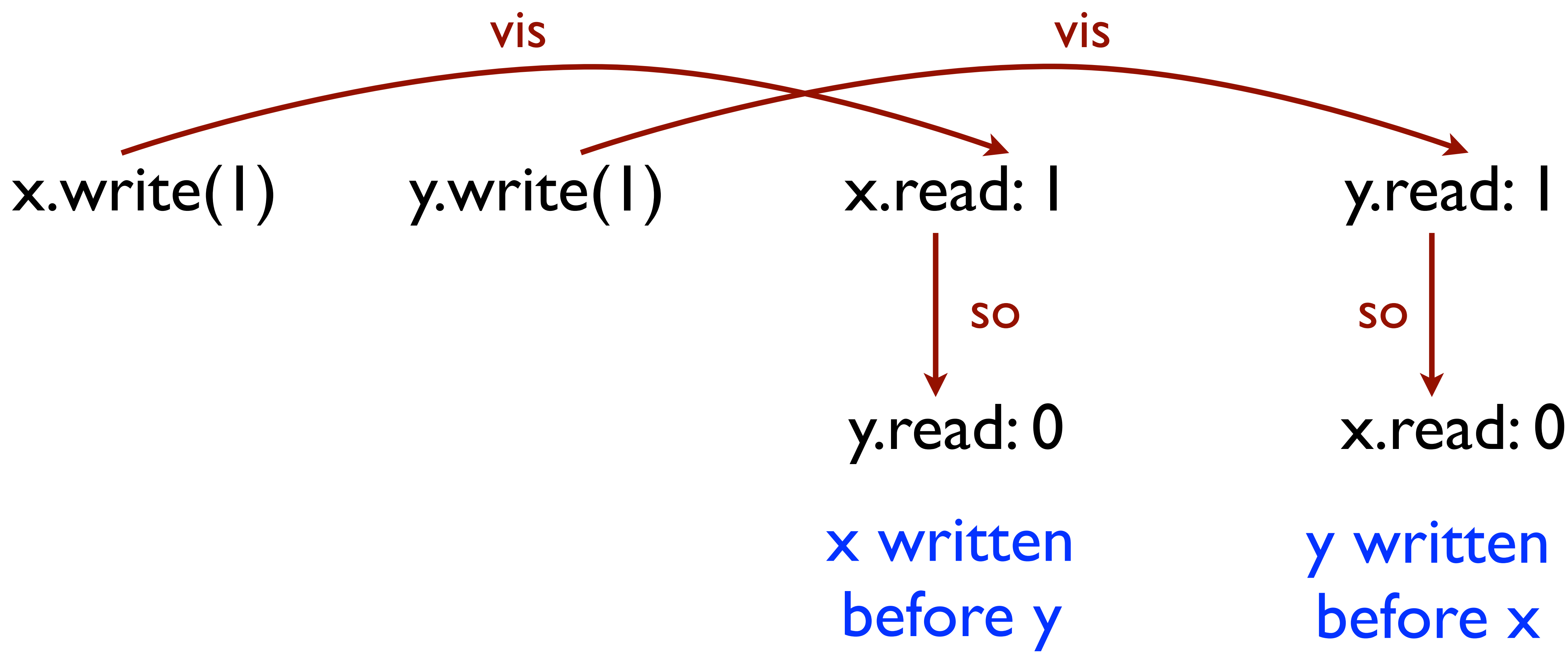
Transactional Dekker = write skew



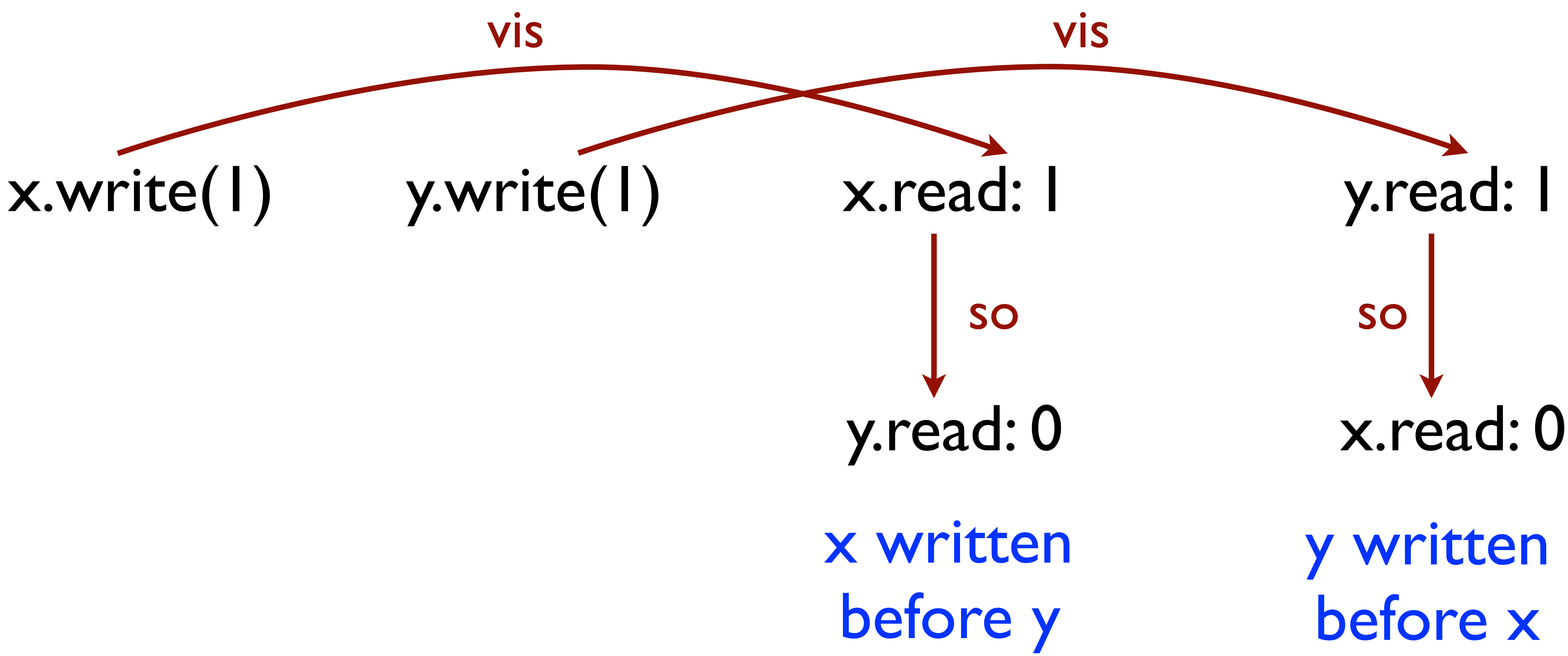
Independent reads of independent writes (IRIW)



Independent reads of independent writes (IRIW)

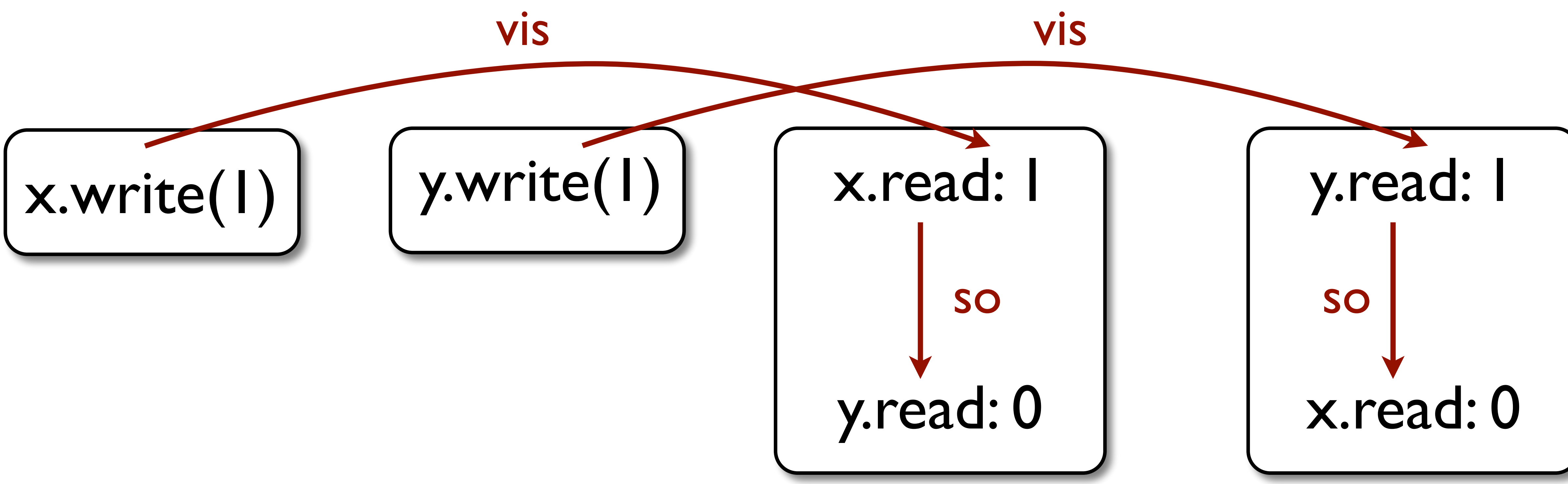


Independent reads of independent writes (IRIW)

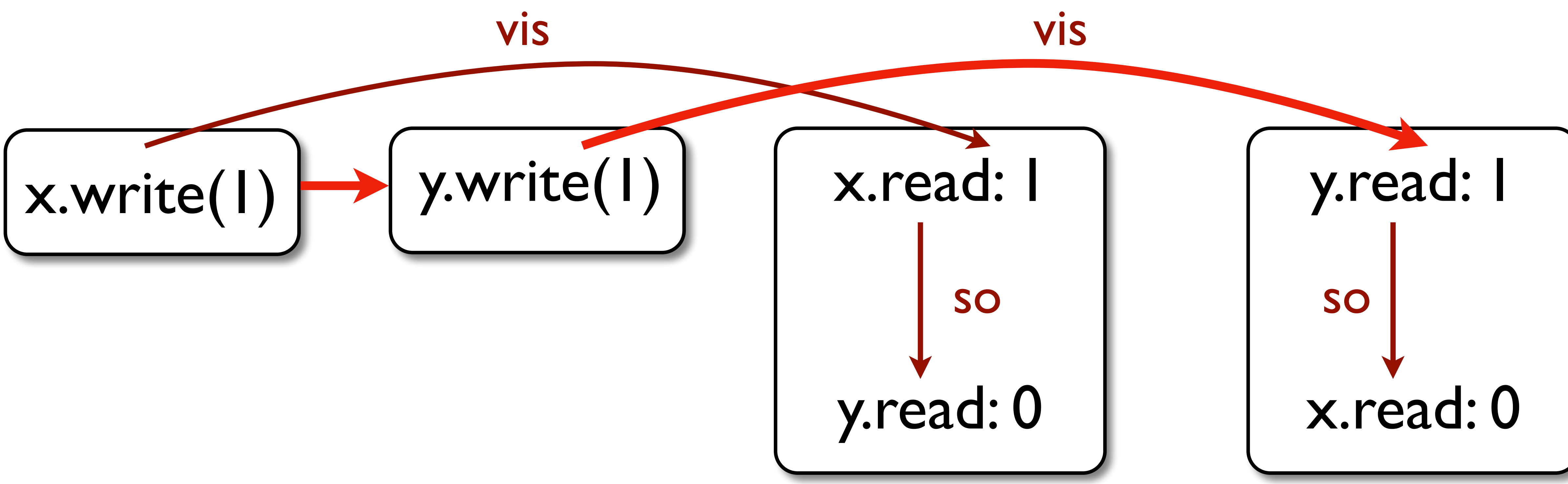


Implementations: no causal dependency between the two writes
➜ can be delivered in different orders at different replicas

Transactional IRIW = long fork



Transactional IRIW = long fork



Not serializable, allowed by transactional causal consistency and parallel snapshot isolation

Robustness

- Is an application robust against a particular consistency model?

  Application behaves the same as when using a strongly consistent database

Application behaves the same whether using a PSI or a serializable database: $[\![A]\!]_{PSI} = [\![A]\!]_{SER}$

Robustness

- Application: set of transactional programs $\{P_1, ..., P_n\}$

```
tx lookup() {
  return acct.bal
}
```

```
tx deposit(n) {
  acct.bal += n
}
```

  - Every program can generate multiple transactions at run time
  - Simplification: every program is in its own session

Robustness

- Application: set of transactional programs $\{P_1, ..., P_n\}$

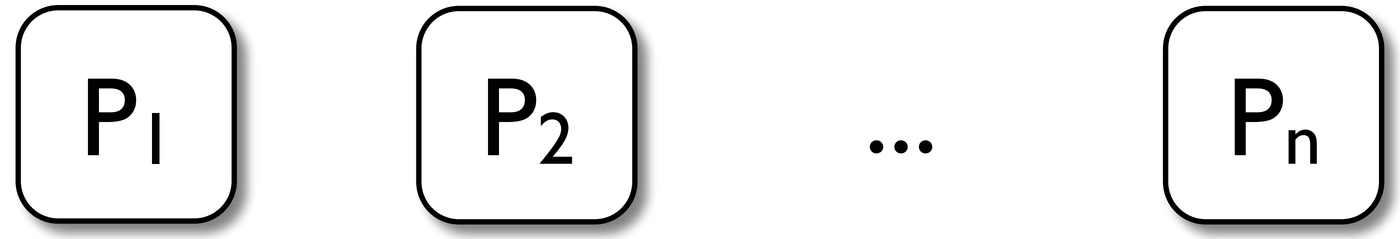
```
tx lookup() {
  return acct.bal
}
```

```
tx deposit(n) {
  acct.bal += n
}
```

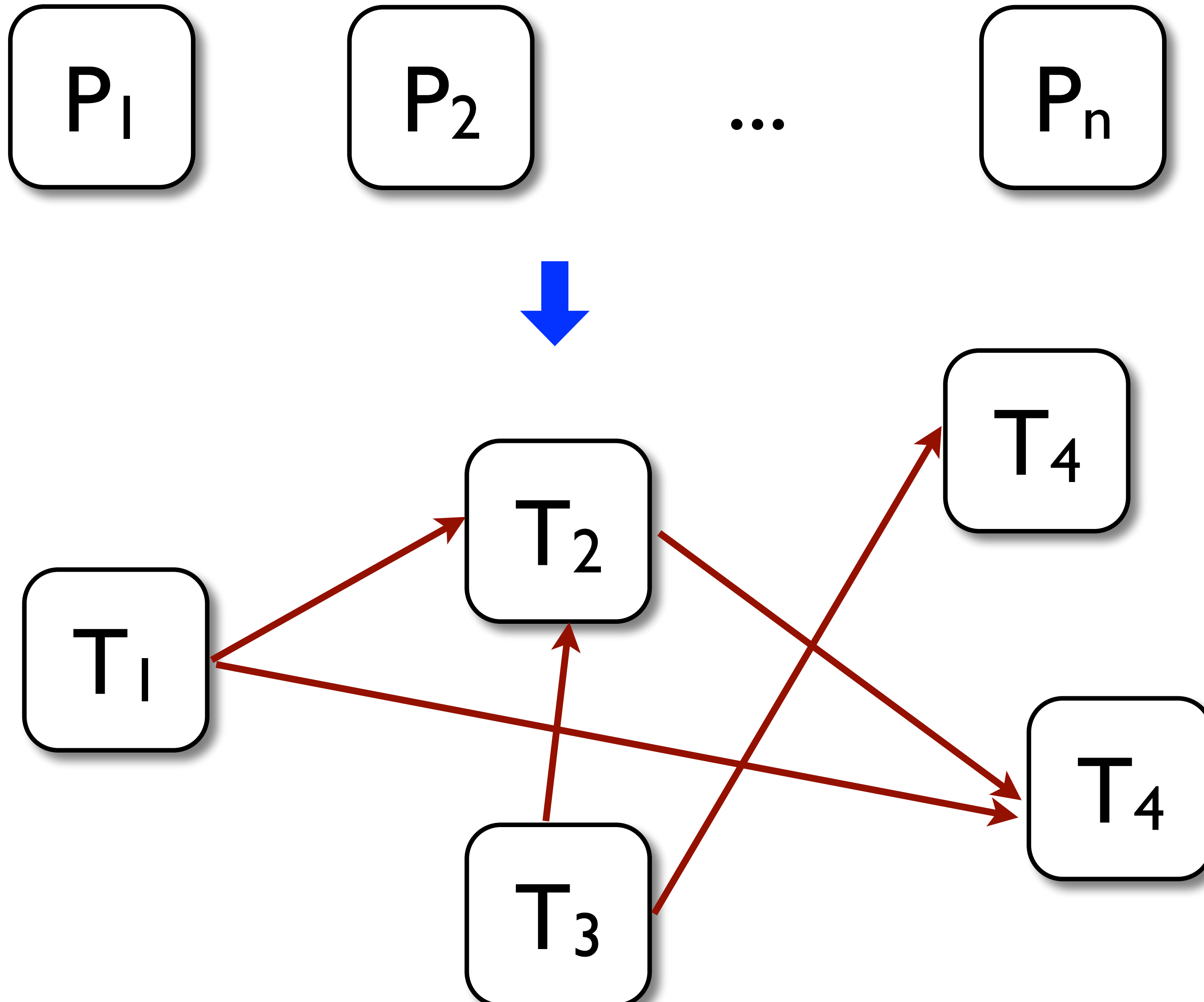
  - Every program can generate multiple transactions at run time
  - Simplification: every program is in its own session

- Checking robustness via static analysis: over-approximate the set of program behaviours

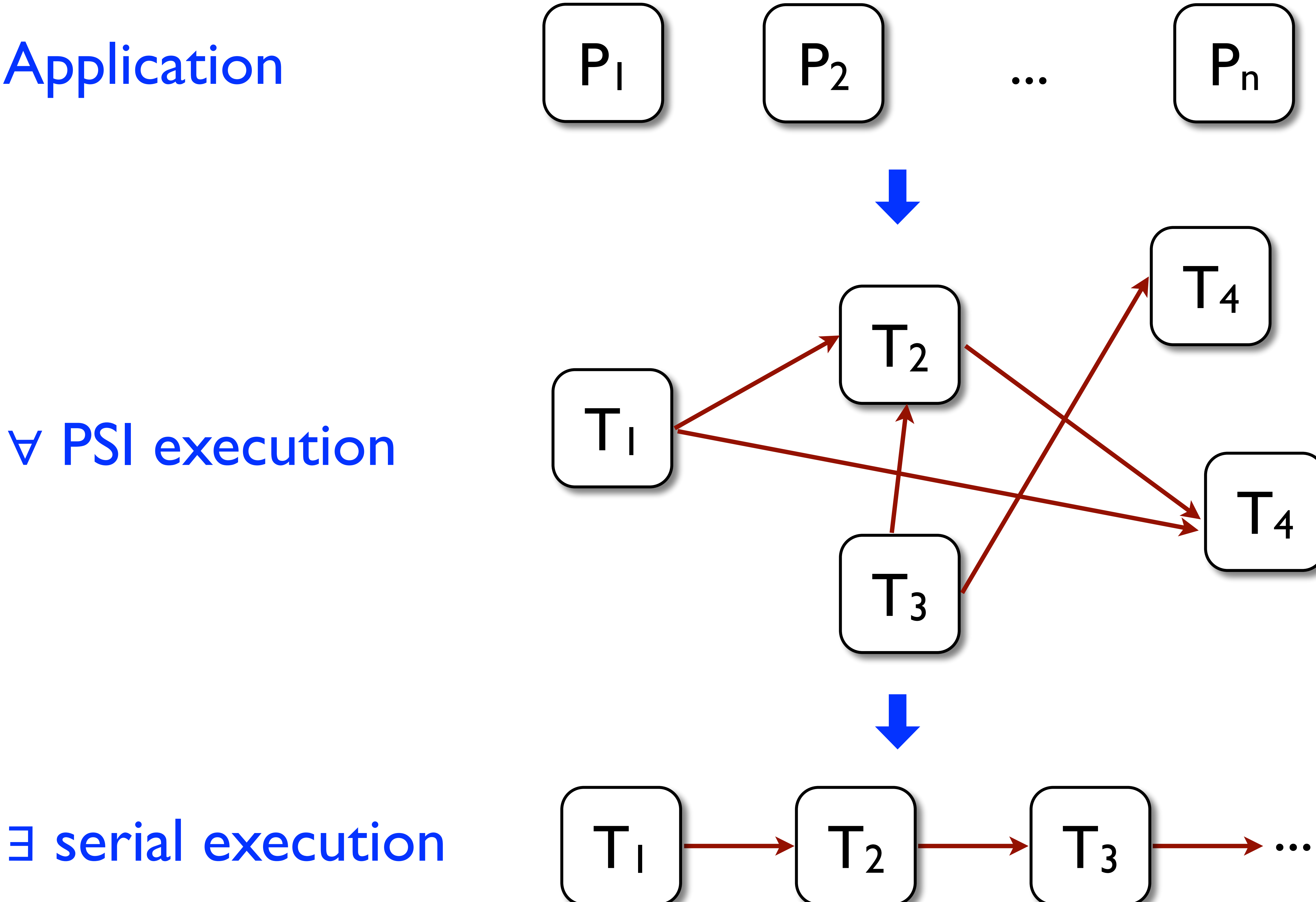
Application



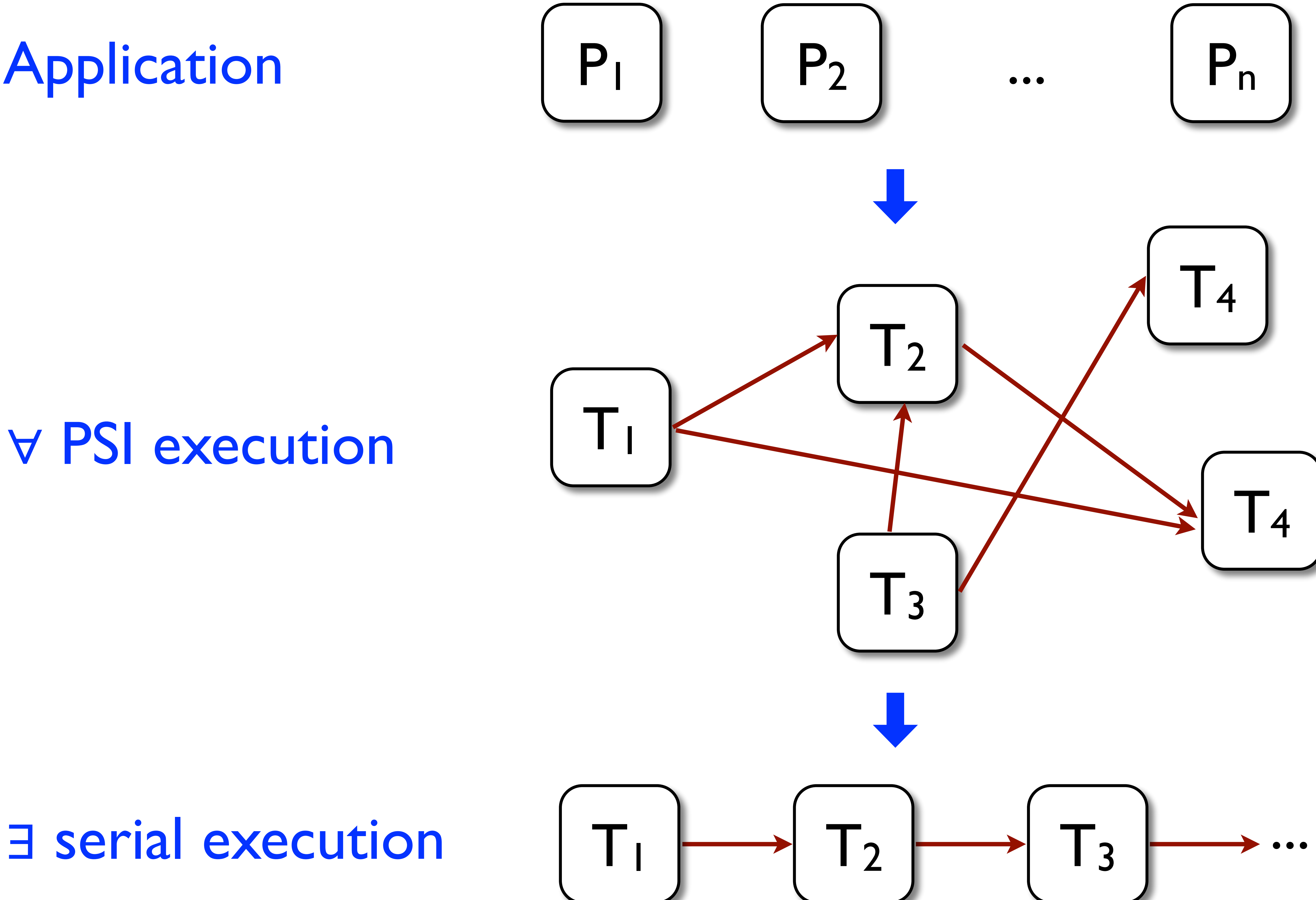
Application



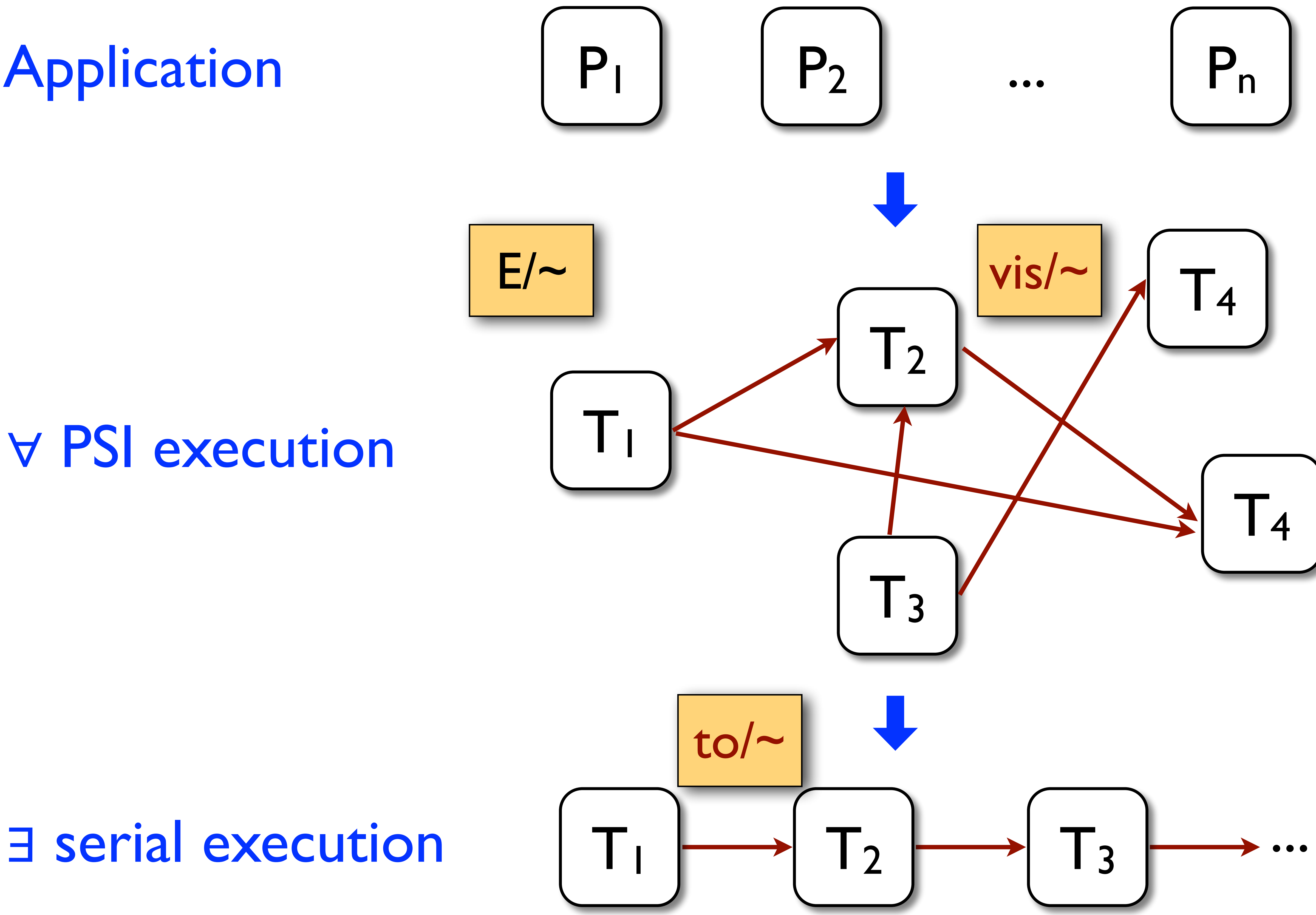
$\forall$ PSI execution



Application
P1
P2
...
Pn
∀ PSI execution
T1
T2
T3
T4
T4
∃ serial execution
T1
T2
T3
...



Each read returns the value written by the last write

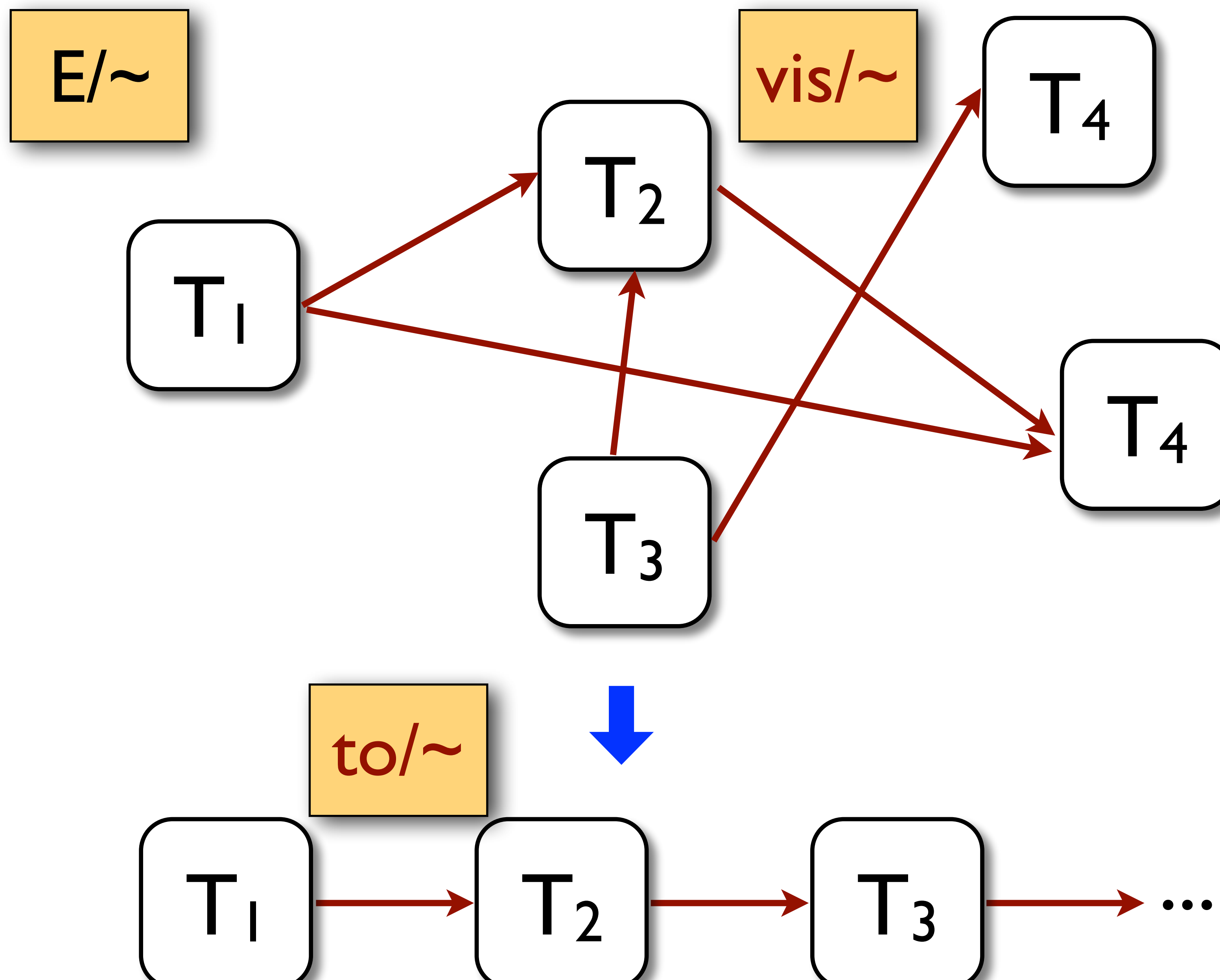


Each read returns the value written by the last write

First determine if a given PSI execution is serializable

∀ PSI execution

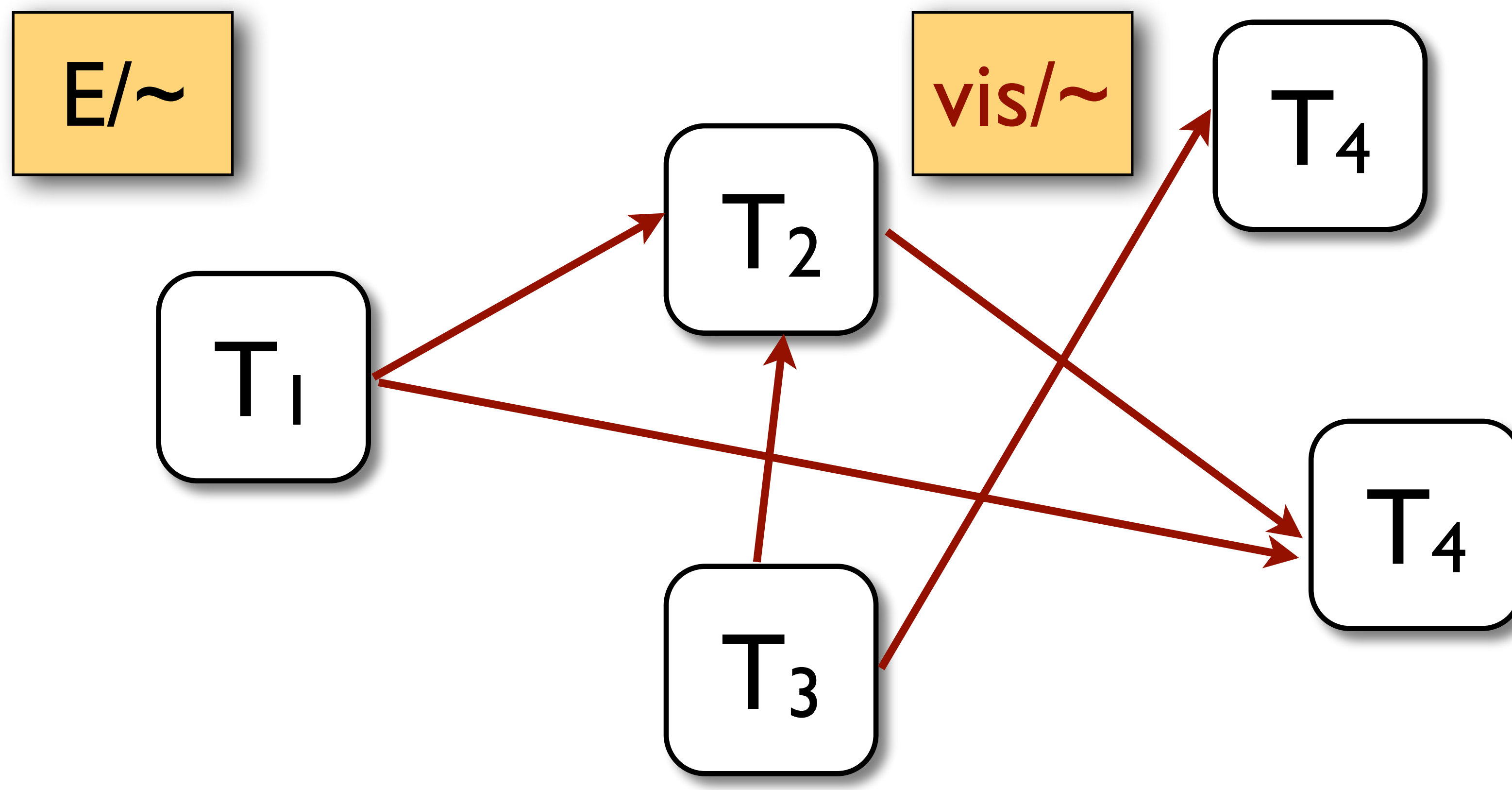
∃ serial execution



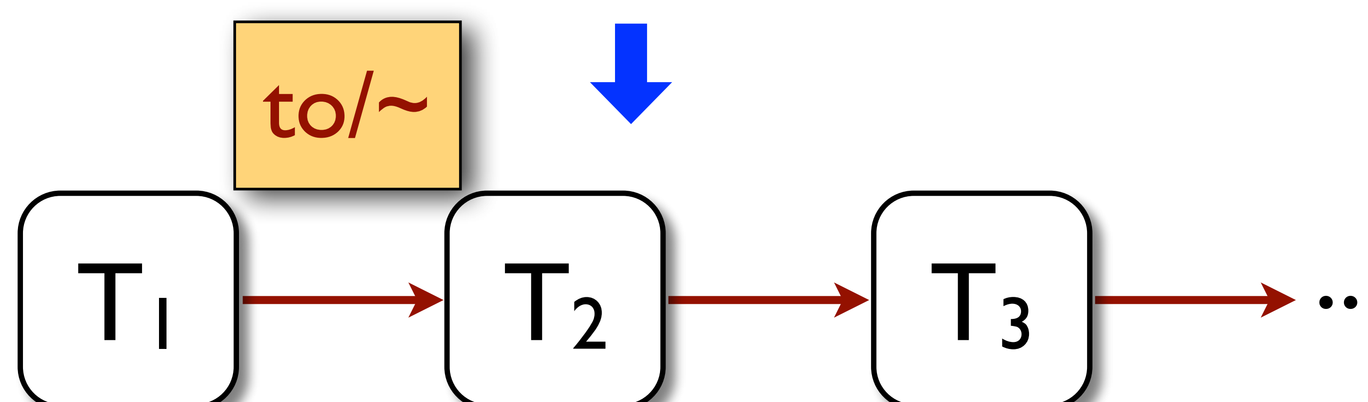
Each read returns the value written by the last write

Build constraints on the serial order: relations on E/~ that should be included into to/~ - transactional dependencies

∀ PSI execution



∃ serial execution



Each read returns the value written by the last write

Write-read dependency (wr)

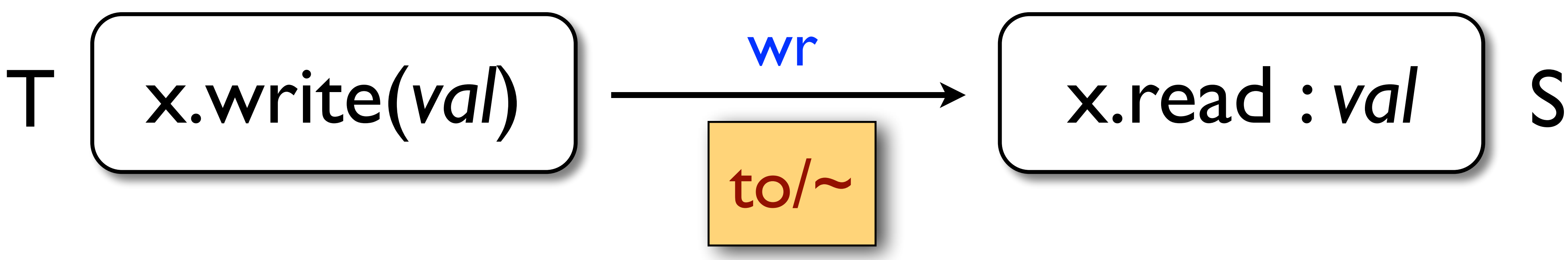
Given a PSI execution (E, ~, vis) and T, S ∈ E/~

$$T \ \boxed{\text{x.write}(\mathit{val})} \xrightarrow{\quad \text{wr} \quad} \boxed{\text{x.read} : \mathit{val}} \ S$$

$T \xrightarrow{\text{wr}} S \iff$ S reads a value written by T

Write-read dependency (wr)

Given a PSI execution (E, ~, vis) and T, S ∈ E/~



$T \xrightarrow{\text{wr}} S \iff$ S reads a value written by T

Write-read dependency (wr)

Given a PSI execution $(E, \sim, \text{vis})$ and $T, S \in E/\sim$

$$T \quad \boxed{x.\text{write}(\mathit{val})} \xrightarrow[\text{to}/\sim]{\text{wr}} \boxed{x.\text{read} : \mathit{val}} \quad S$$

$T \xrightarrow{\text{wr}} S \iff$ S reads a value written by T

$T \xrightarrow{\text{wr}} S \iff T \neq S \wedge$ T contains the most recent write of an object x visible to a read from x in S according to vis

Write-write dependency (wr)

Given a PSI execution (E, ~, vis) and T, S ∈ E/~

$$T\ \boxed{x.\mathrm{write}(\mathit{old})} \xrightarrow{\;\;ww\;\;} \boxed{x.\mathrm{write}(\mathit{new})}\ S$$

$T \xrightarrow{ww} S \iff$ S overwrites a value written by T

Write-write dependency (wr)

Given a PSI execution (E, ~, vis) and T, S ∈ E/~

$$T \quad \boxed{\text{x.write}(\textit{old})} \xrightarrow[\boxed{\text{to/}\sim}]{\text{ww}} \boxed{\text{x.write}(\textit{new})} \quad S$$

$T \xrightarrow{\text{ww}} S \iff$ S overwrites a value written by T

Write-write dependency (wr)

Given a PSI execution (E, ~, vis) and T, S ∈ E/~

$$T\ \boxed{x.\mathrm{write}(\mathit{old})} \xrightarrow[\text{to/}\sim]{\text{ww}} \boxed{x.\mathrm{write}(\mathit{new})}\ S$$

$T \xrightarrow{\text{ww}} S \iff$ S overwrites a value written by T

$T \xrightarrow{\text{ww}} S \iff$ T and S contain writes to the same object x and $T \xrightarrow{\text{vis}/\sim} S$

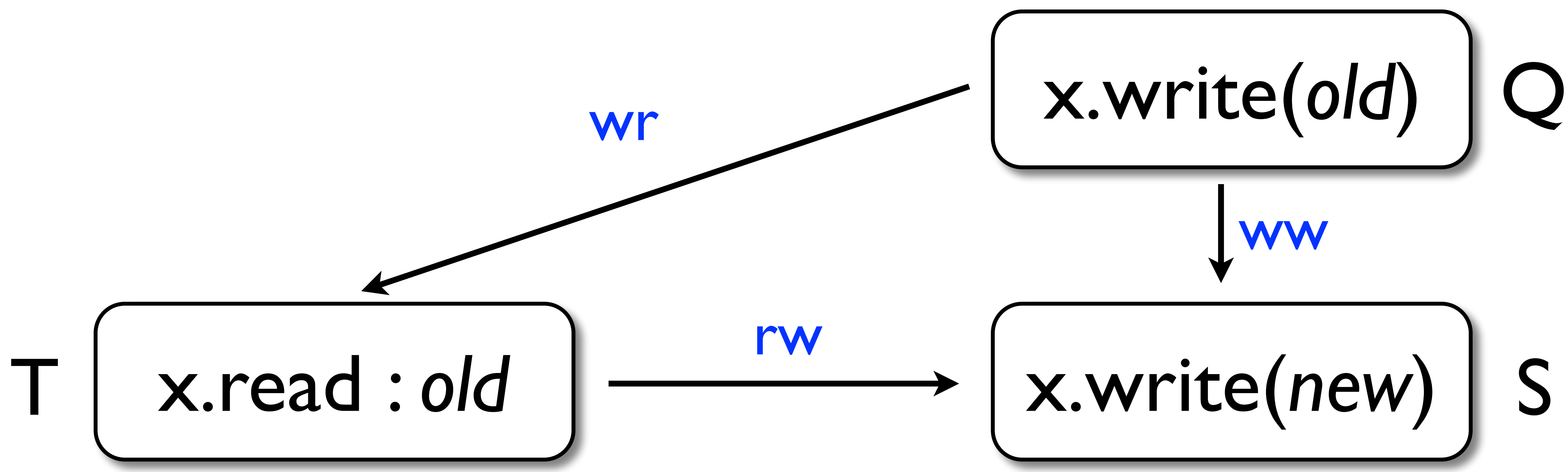
Read-write dependency (rw)

$$T \ \boxed{\text{x.read} : \textit{old}} \xrightarrow{\text{rw}} \boxed{\text{x.write}(\textit{new})} \ S$$

$$T \xrightarrow{\text{rw}} S \iff T \neq S \wedge S \text{ overwrites a value read by } T$$

$$T \xrightarrow{\text{rw}} S \iff T \neq S \wedge \exists Q.\, Q \xrightarrow{\text{wr}} T \wedge Q \xrightarrow{\text{ww}} S$$

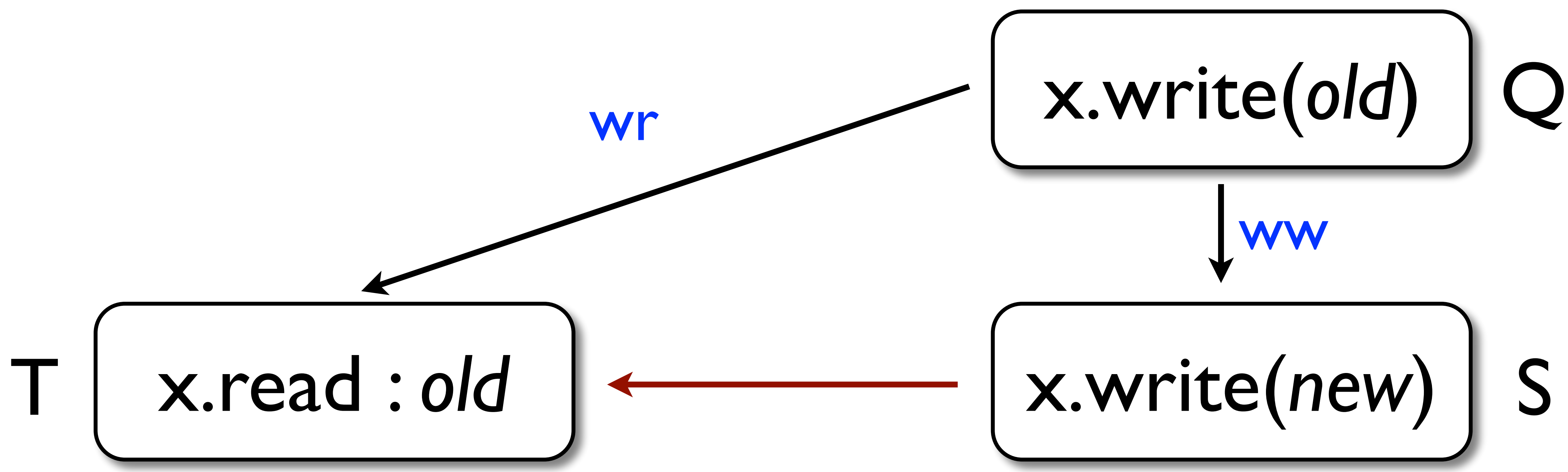
Read-write dependency (rw)



$$T \xrightarrow{rw} S \iff T \neq S \wedge S \text{ overwrites a value read by } T$$

$$T \xrightarrow{rw} S \iff T \neq S \wedge \exists Q.\, Q \xrightarrow{wr} T \wedge Q \xrightarrow{ww} S$$

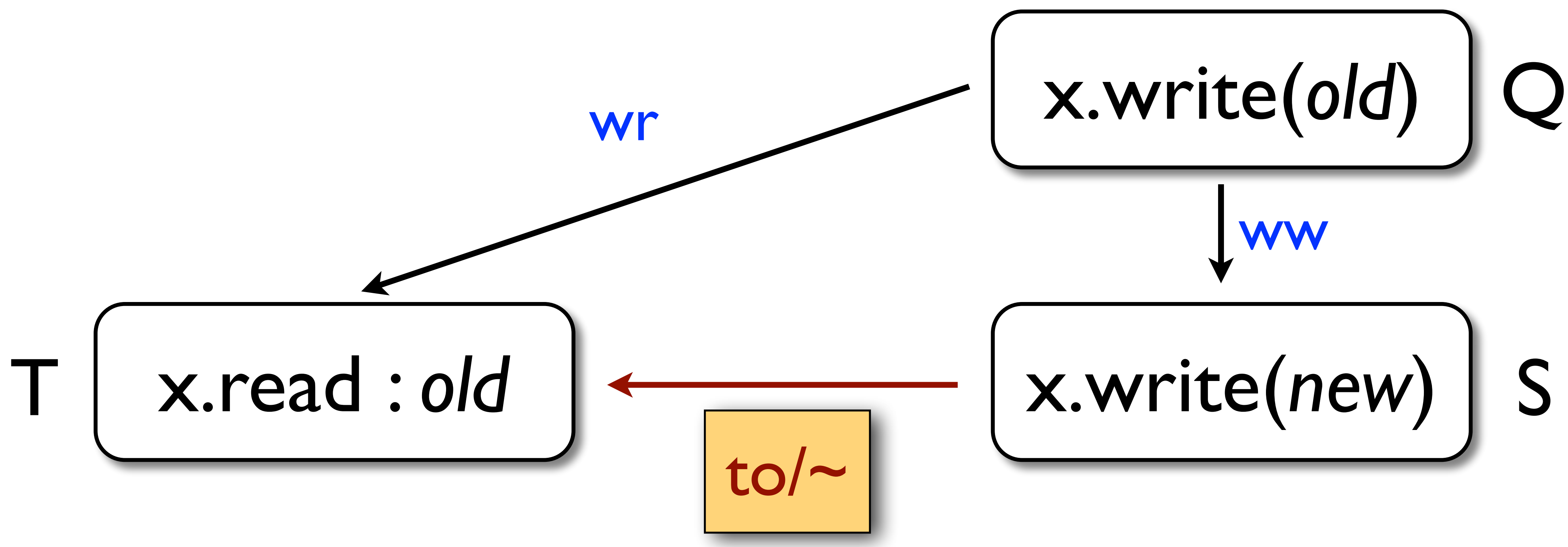
Read-write dependency (rw)



$$T \xrightarrow{\text{rw}} S \iff T \neq S \wedge S \text{ overwrites a value read by } T$$

$$T \xrightarrow{\text{rw}} S \iff T \neq S \wedge \exists Q.\, Q \xrightarrow{\text{wr}} T \wedge Q \xrightarrow{\text{ww}} S$$

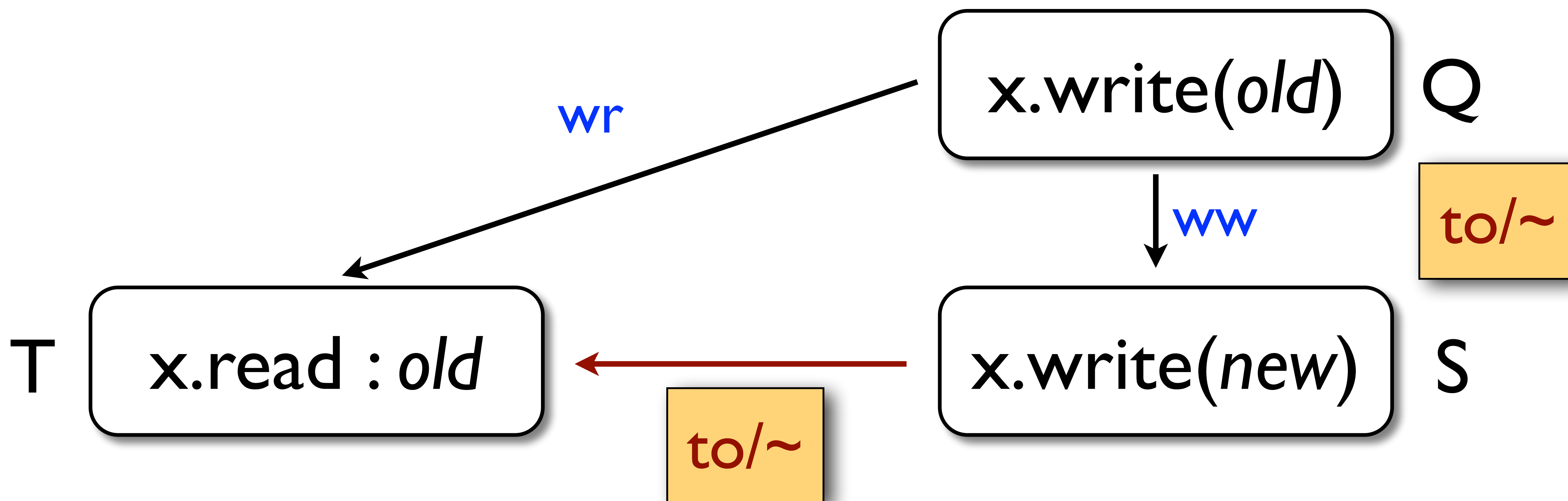
Read-write dependency (rw)



$$T \xrightarrow{rw} S \iff T \neq S \wedge S \text{ overwrites a value read by } T$$

$$T \xrightarrow{rw} S \iff T \neq S \wedge \exists Q.\, Q \xrightarrow{wr} T \wedge Q \xrightarrow{ww} S$$

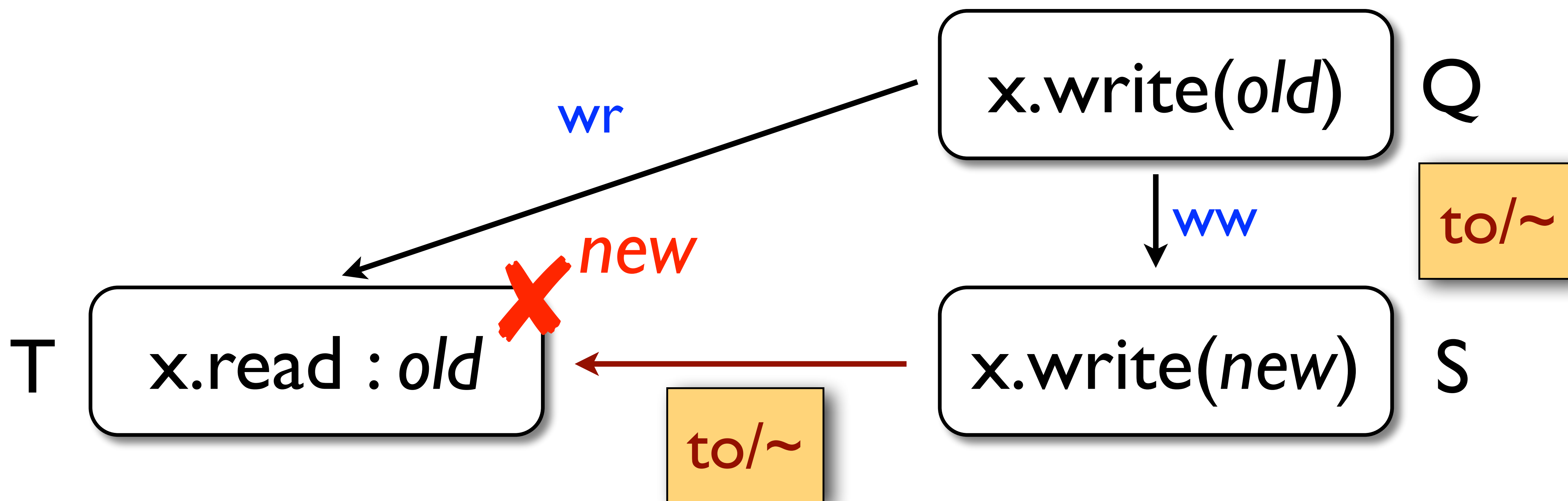
Read-write dependency (rw)



$$T \xrightarrow{rw} S \iff T \neq S \wedge S \text{ overwrites a value read by } T$$

$$T \xrightarrow{rw} S \iff T \neq S \wedge \exists Q.\, Q \xrightarrow{wr} T \wedge Q \xrightarrow{ww} S$$

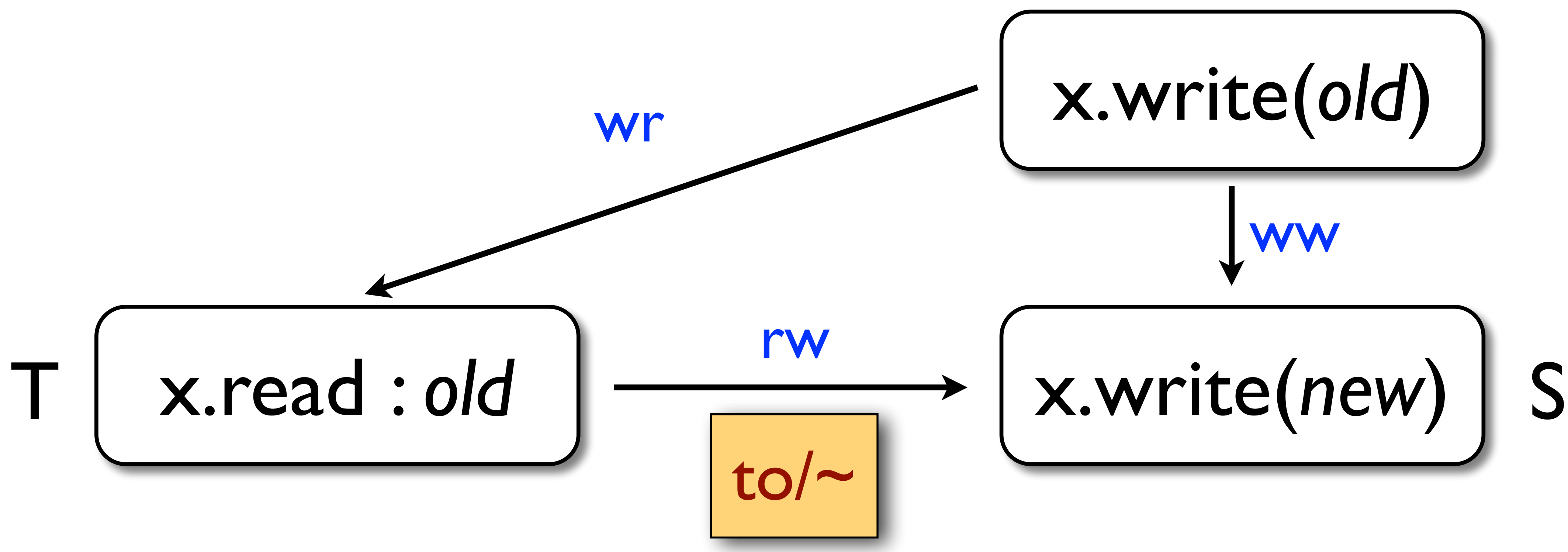
Read-write dependency (rw)



$$T \xrightarrow{rw} S \iff T \neq S \wedge S \text{ overwrites a value read by } T$$

$$T \xrightarrow{rw} S \iff T \neq S \wedge \exists Q.\, Q \xrightarrow{wr} T \wedge Q \xrightarrow{ww} S$$

Read-write dependency (rw)



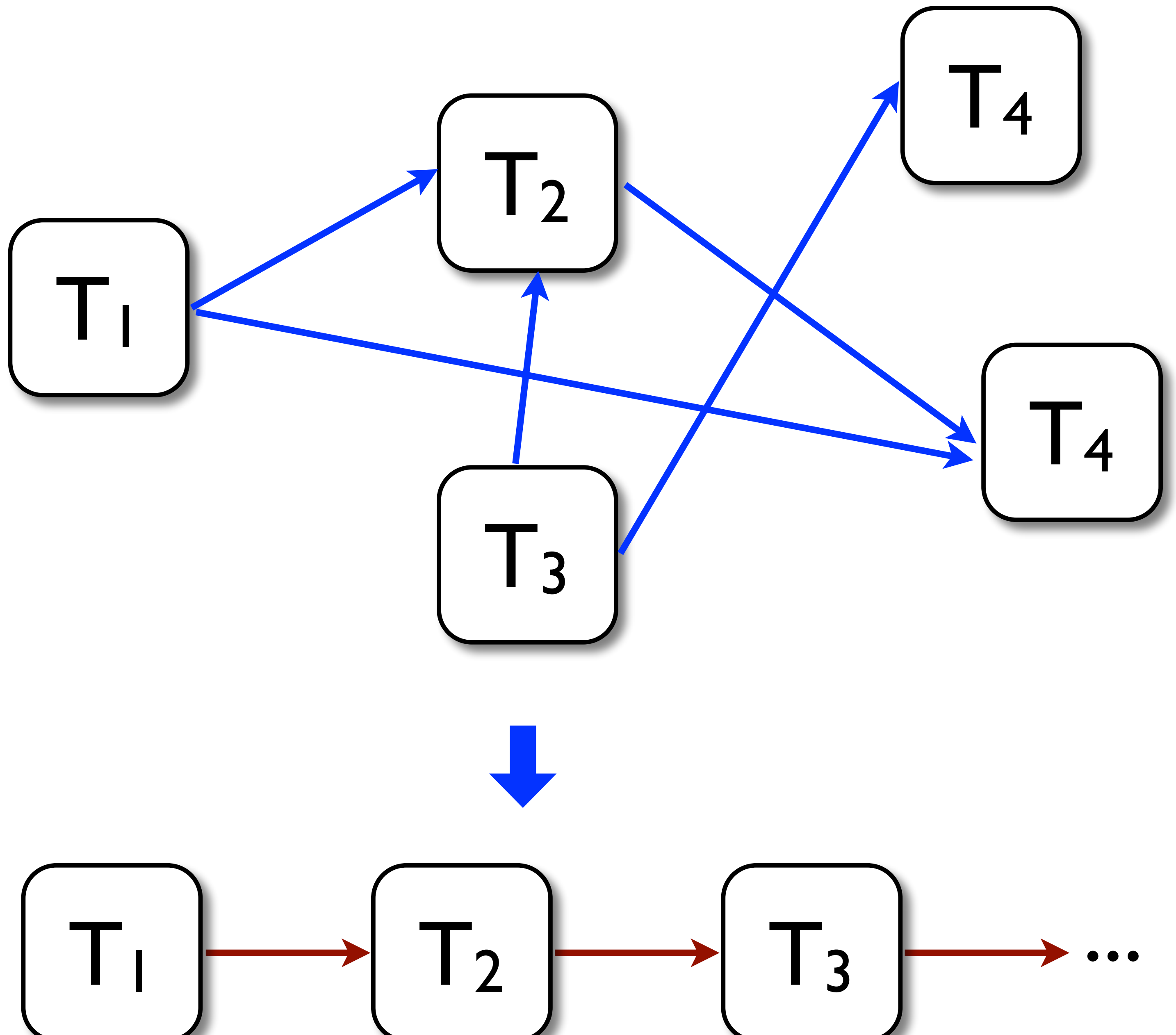
$$T \xrightarrow{\text{rw}} S \iff T \neq S \wedge S \text{ overwrites a value read by } T$$

$$T \xrightarrow{\text{rw}} S \iff T \neq S \wedge \exists Q.\, Q \xrightarrow{\text{wr}} T \wedge Q \xrightarrow{\text{ww}} S$$

Dependency graphs

- PSI execution $(E, \sim, vis)$ ➜ dependency graph $(E/\sim, wr, ww, rw)$
- Theorem: If the dependency graph is acyclic, then the execution is serializable

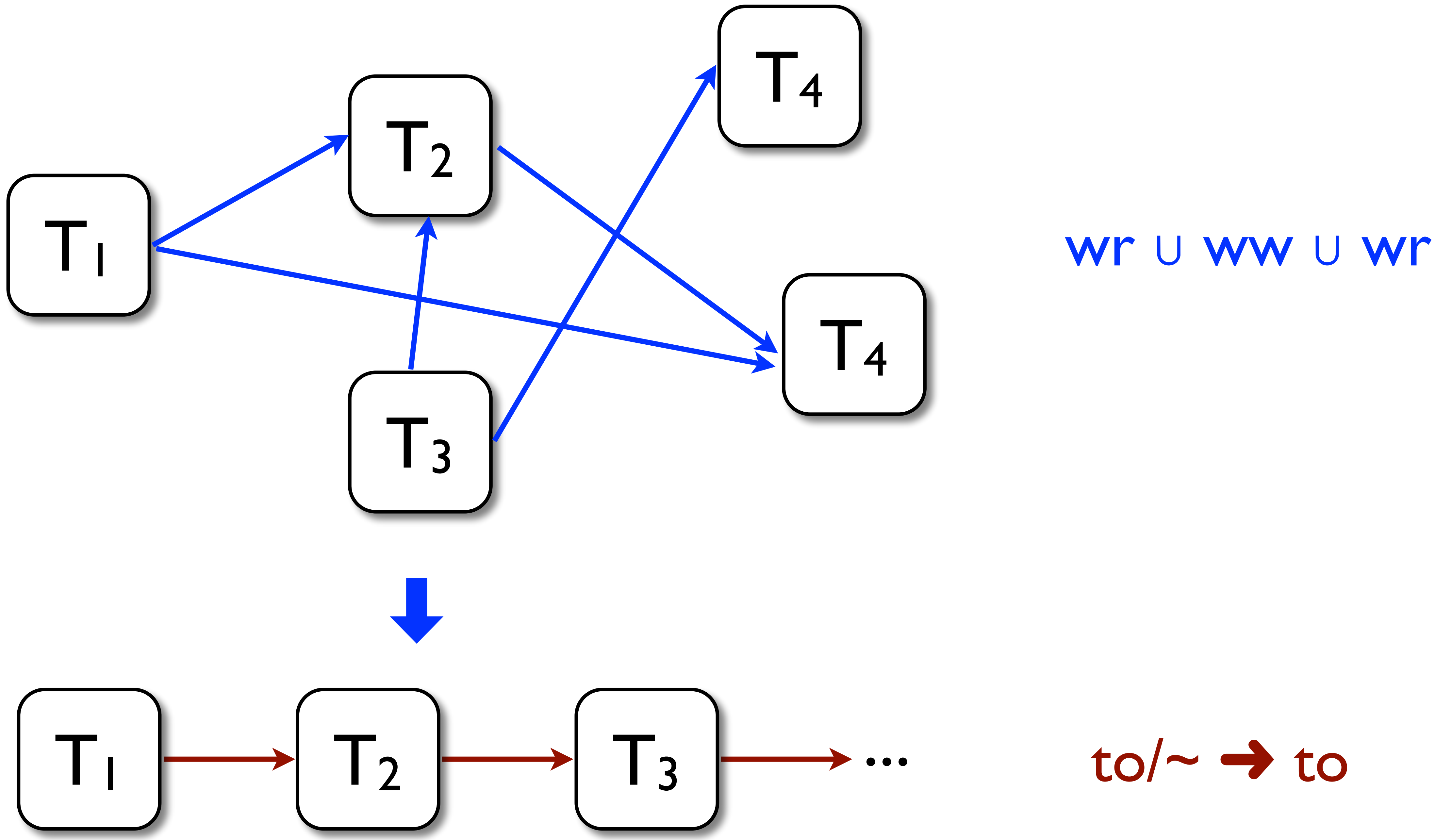
If (wr ∪ ww ∪ wr) is acyclic, then there is a total order on E/~ containing it [order-extension principle] ➜ the desired order to



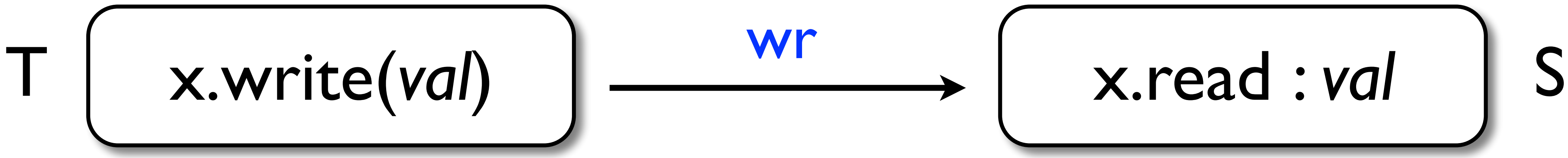
wr ∪ ww ∪ wr

to/~ ➜ to

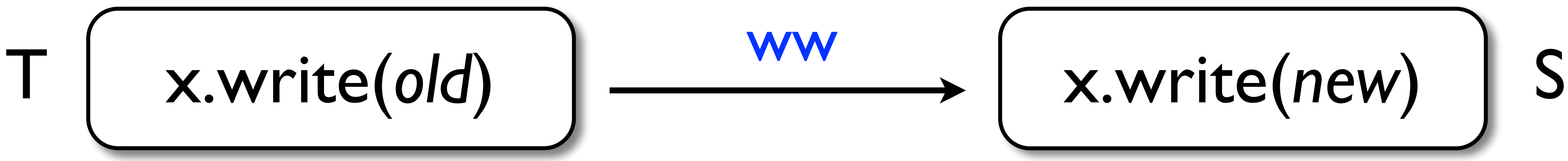
If (wr ∪ ww ∪ wr) is acyclic, then there is a total order on E/~ containing it [order-extension principle] ➜ the desired order to



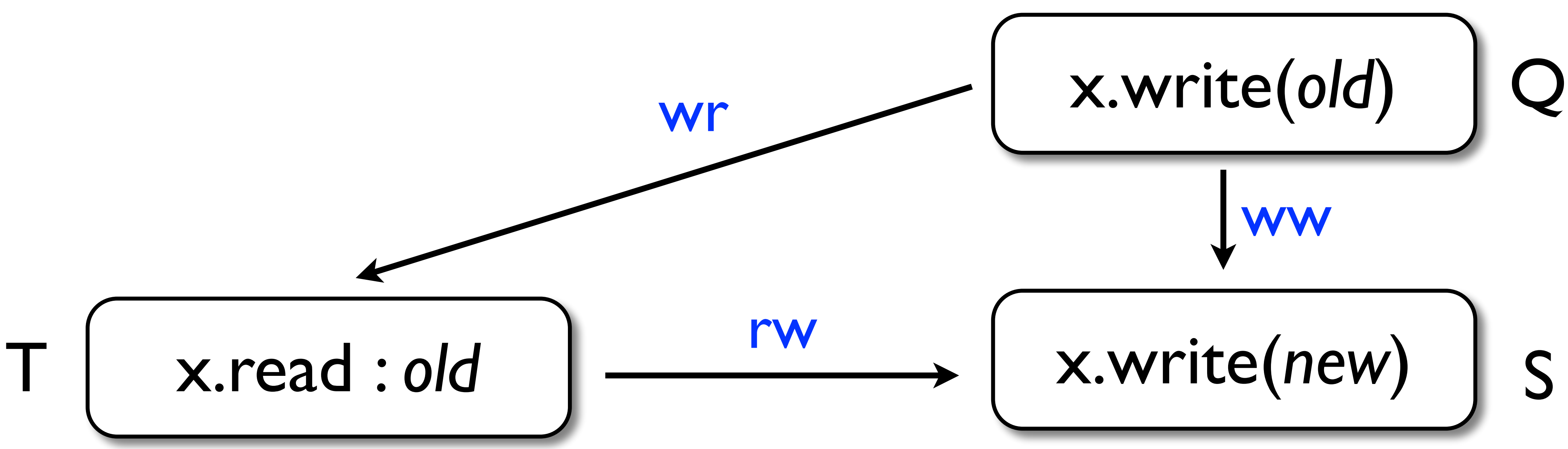
Each read returns the value written by the last write in to?



$T \xrightarrow{\mathsf{wr}} S \iff T \neq S \wedge T$ contains the most recent write of an object x visible to a read from x in S according to vis



$T \xrightarrow{\mathsf{ww}} S \iff$ T and S contain writes to the same object x and $T \xrightarrow{\mathsf{vis}/\sim} S$



$T \xrightarrow{\mathsf{rw}} S \iff T \neq S \wedge \exists Q.\, Q \xrightarrow{\mathsf{wr}} T \wedge Q \xrightarrow{\mathsf{ww}} S$

If the dependency graph $(E/{\sim}, wr, ww, rw)$ of a PSI execution $(E, {\sim}, vis)$ is acyclic, then the execution is serializable

If the dependency graph (E/~, wr, ww, rw) of a PSI execution (E, ~, vis) is acyclic, then the execution is serializable

Transactional programs P_1, P_2, ..., P_n

⬇

Set of all their PSI executions (E, ~, vis)

If the dependency graph $(E/\sim, wr, ww, rw)$ of a PSI execution $(E, \sim, vis)$ is acyclic, then the execution is serializable

Transactional programs $P_1, P_2, \ldots, P_n$

⬇

Set of all their PSI executions $(E, \sim, vis)$

⬇

Set of corresponding dependency graphs $(E/\sim, wr, ww, rw)$

If the dependency graph $(E/\sim, wr, ww, rw)$ of a PSI execution $(E, \sim, vis)$ is acyclic, then the execution is serializable

Transactional programs $P_1, P_2, \ldots, P_n$

⬇

Set of all their PSI executions $(E, \sim, vis)$

⬇

Set of corresponding dependency graphs $(E/\sim, wr, ww, rw)$

⬇

Check $wr \cup ww \cup wr$ is acyclic in each graph

If the dependency graph $(E/\sim, wr, ww, rw)$ of a PSI execution $(E, \sim, vis)$ is acyclic, then the execution is serializable

Transactional programs $P_1, P_2, \ldots, P_n$

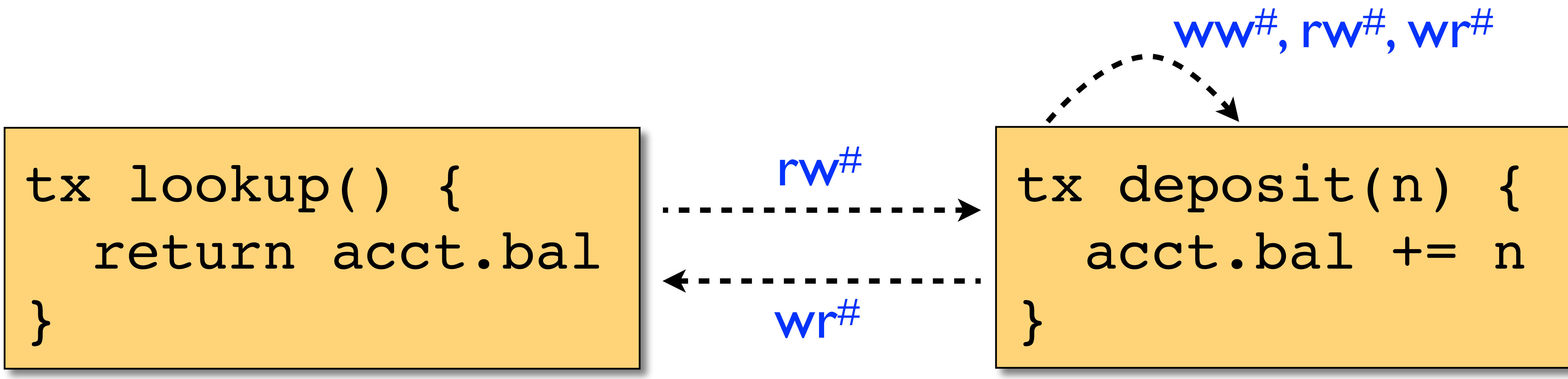
Set of all their PSI executions $(E, \sim, vis)$

Set of corresponding dependency graphs $(E/\sim, wr, ww, rw)$

Check $wr \cup ww \cup wr$ is acyclic in each graph

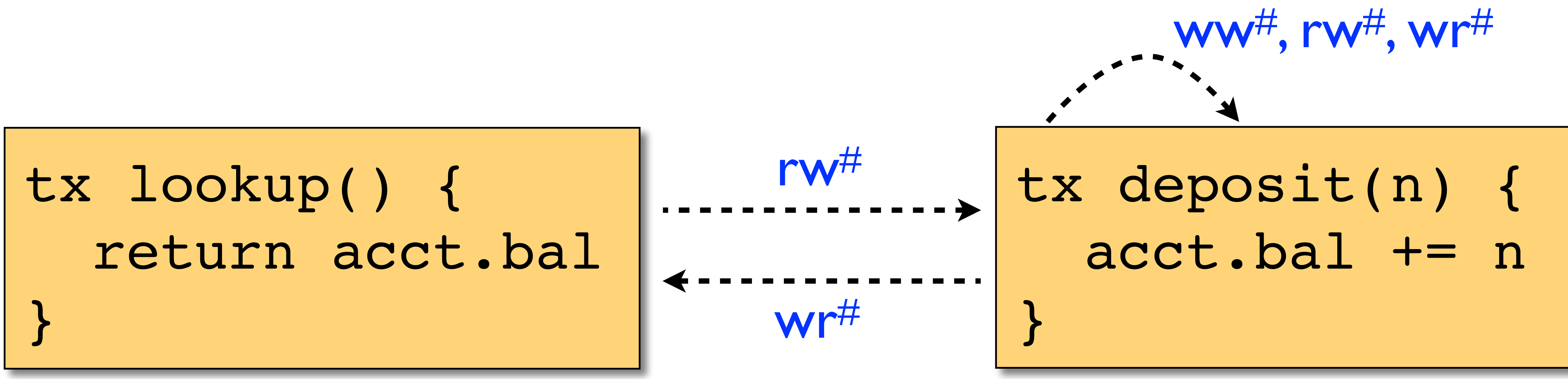
Over-approximate the set of possible dependency graphs from the program text

Static dependency graphs



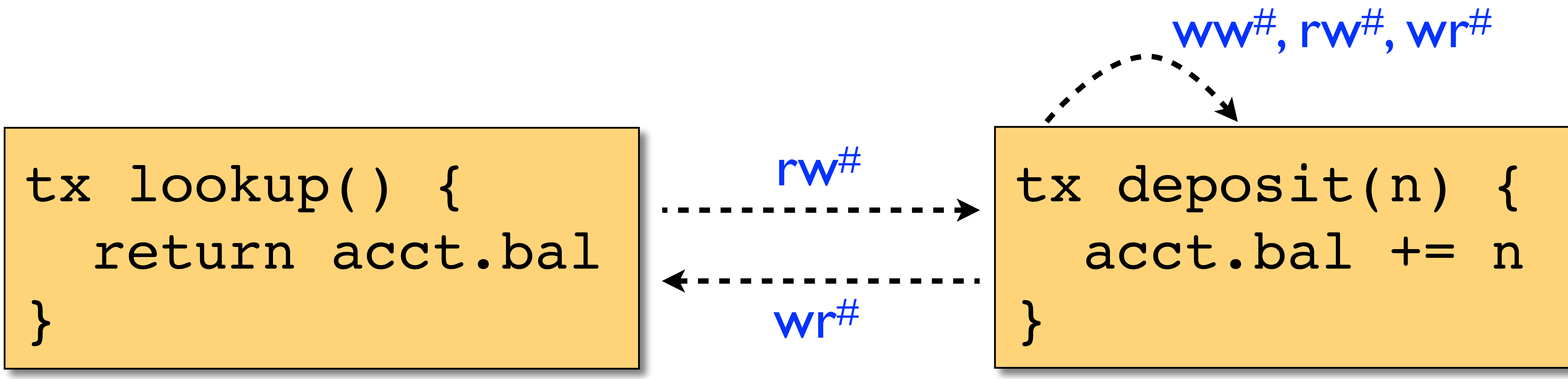
- Nodes: transactional programs
- Edges: over-approximations of dependencies wr#, ww#, rw#

Static dependency graphs



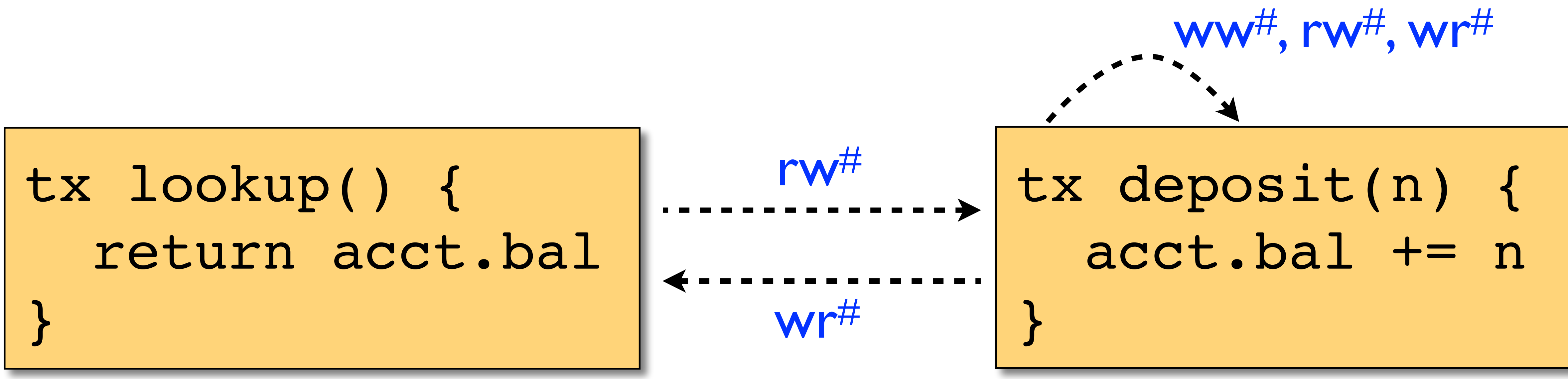
- Nodes: transactional programs
- Edges: over-approximations of dependencies wr#, ww#, rw#
- $T \xrightarrow{wr^\#} S \iff \exists x.\, \mathrm{writes}(T, x) \wedge \mathrm{reads}(T, x)$: over-approximated by static analyses (or even by hand)

Static dependency graphs

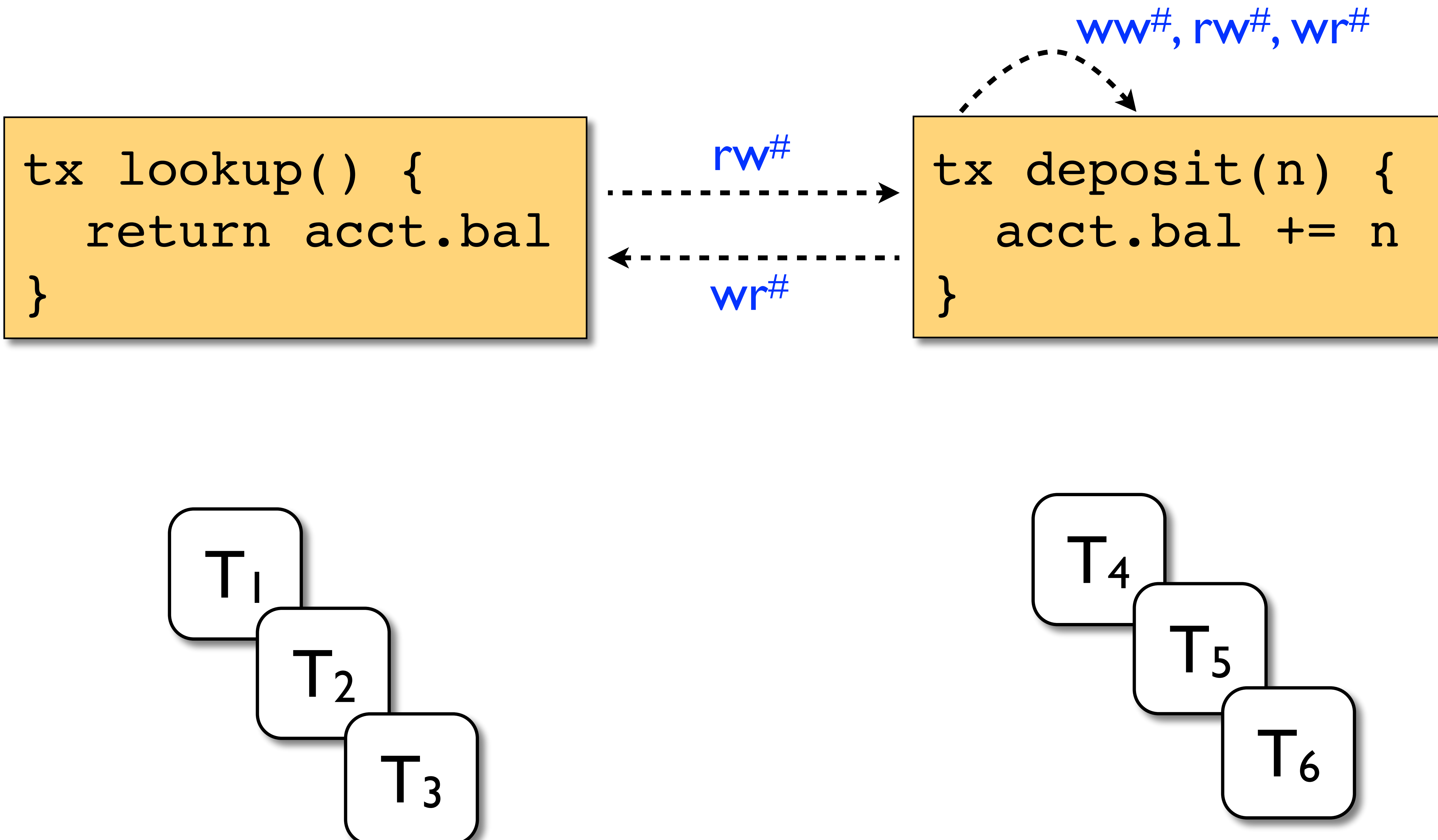


- Nodes: transactional programs
- Edges: over-approximations of dependencies wr#, ww#, rw#
- $T \xrightarrow{wr^{\#}} S \iff \exists x.\ \mathrm{writes}(T, x) \wedge \mathrm{reads}(T, x)$: over-approximated by static analyses (or even by hand)
- Represents an over-approximation of all dynamic dependency graphs that can be produced by the programs

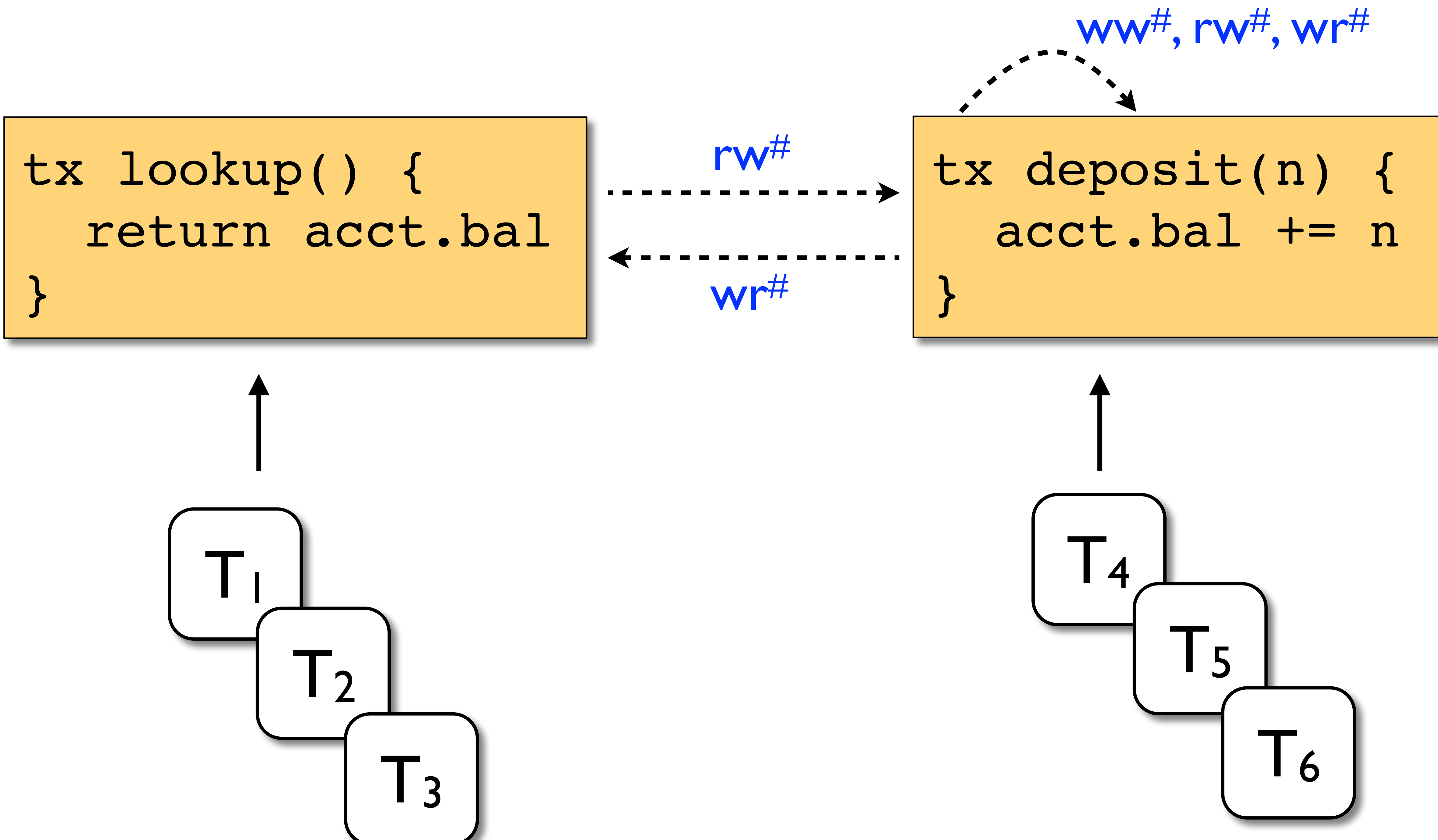
Dynamic dependency graph ➜ a subgraph of the static dependency graph



Dynamic dependency graph ➜ a subgraph of the static dependency graph

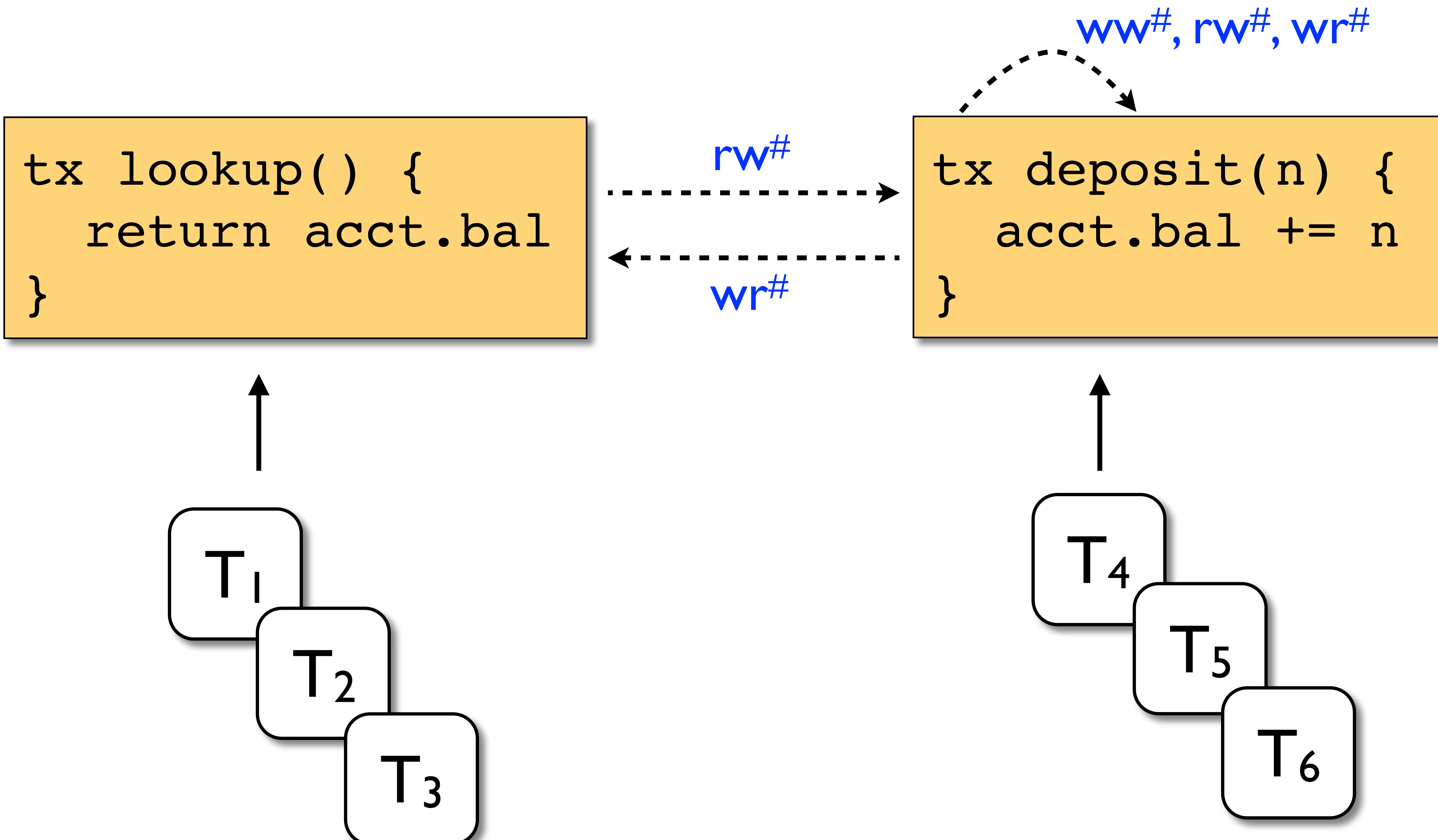


Dynamic dependency graph ➜ a subgraph of the static dependency graph



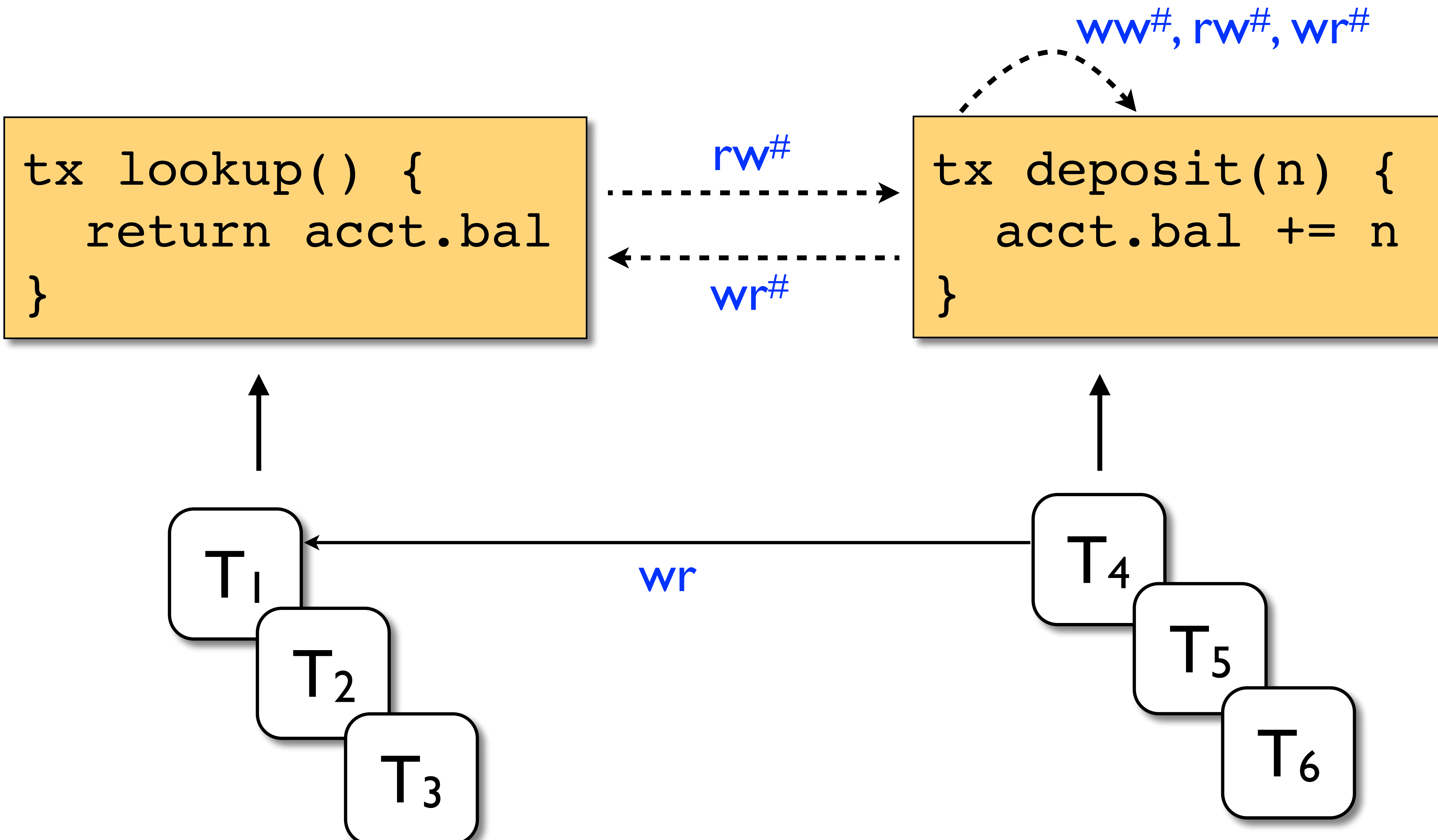
Transactions arising from the same program map to the same node

Dynamic dependency graph ➜ a subgraph of the static dependency graph



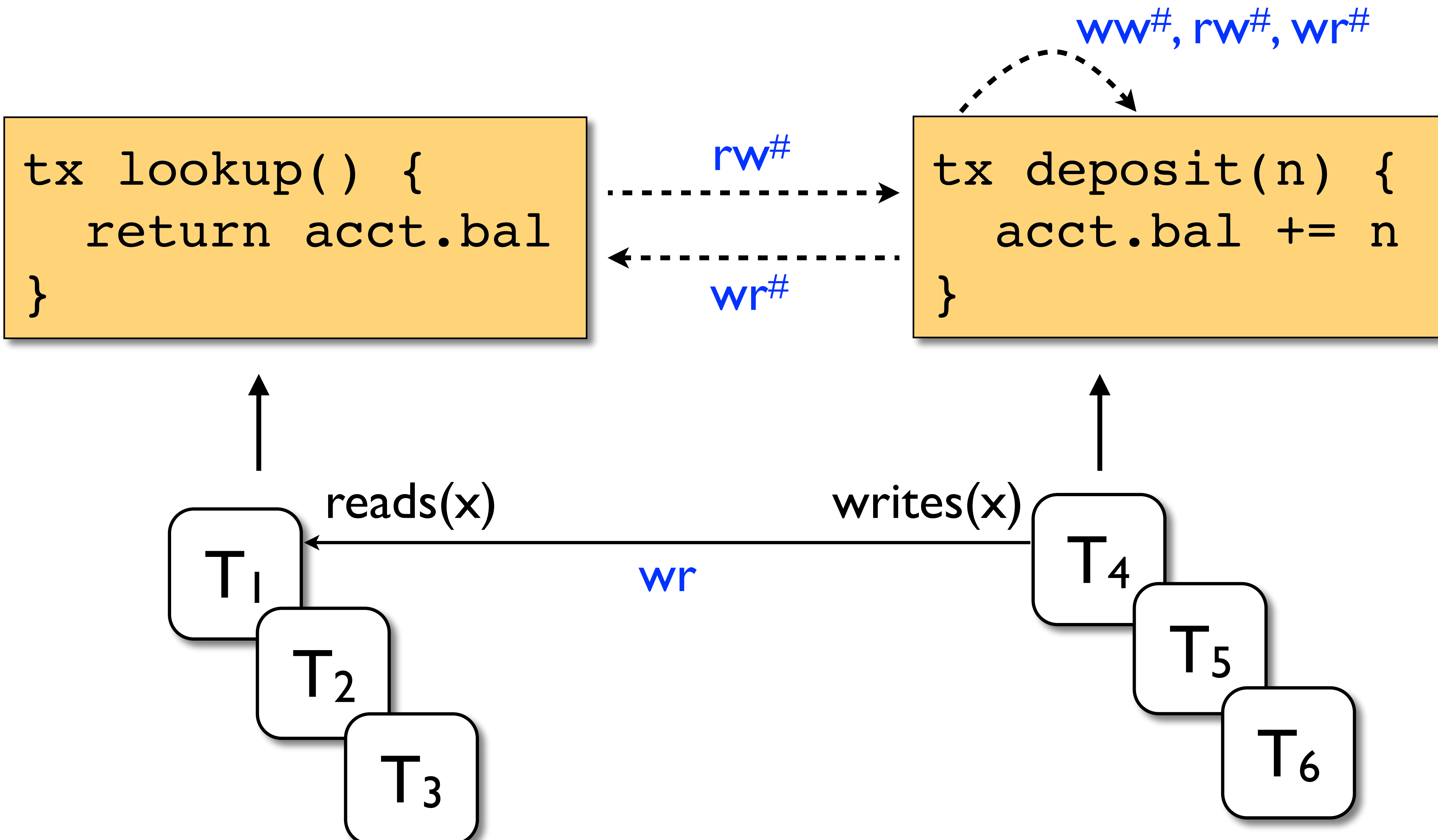
Edge in the dynamic graph ➜ corresponding edge in the static graph

Dynamic dependency graph ➜ a subgraph of the static dependency graph



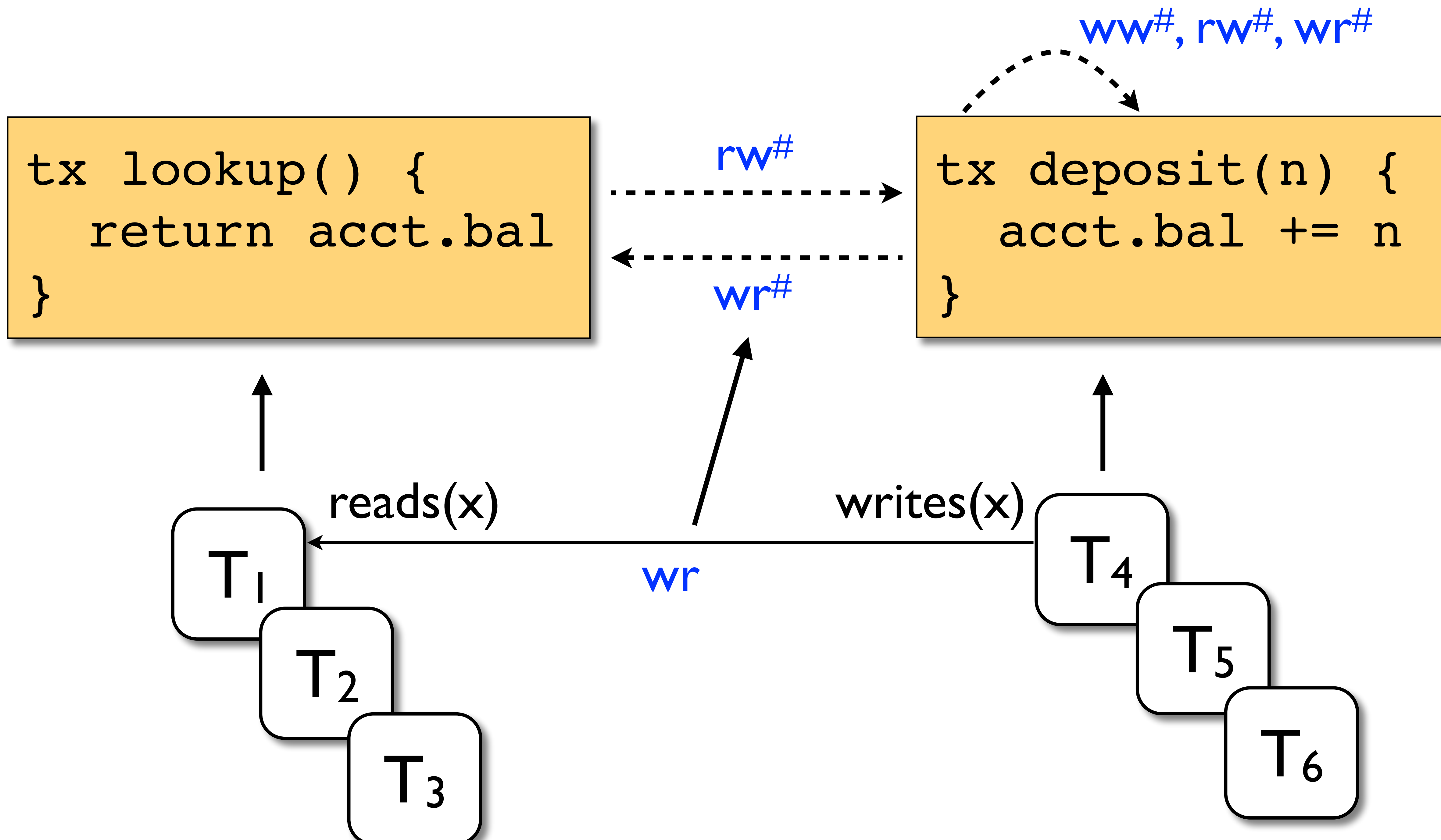
Edge in the dynamic graph ➜ corresponding edge in the static graph

Dynamic dependency graph ➜ a subgraph of the static dependency graph



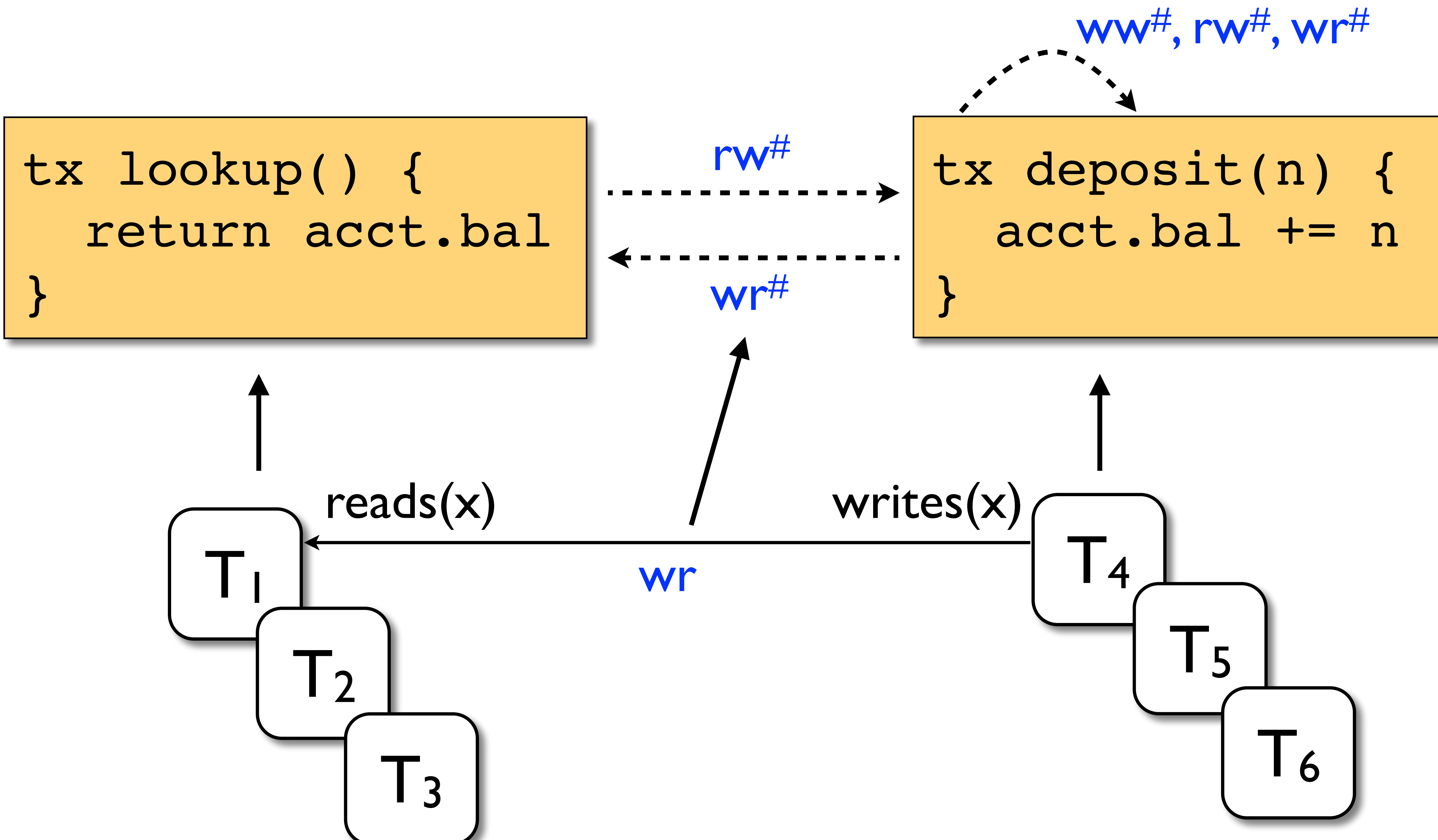
Edge in the dynamic graph ➜ corresponding edge in the static graph

Dynamic dependency graph ➜ a subgraph of the static dependency graph



Edge in the dynamic graph ➜ corresponding edge in the static graph

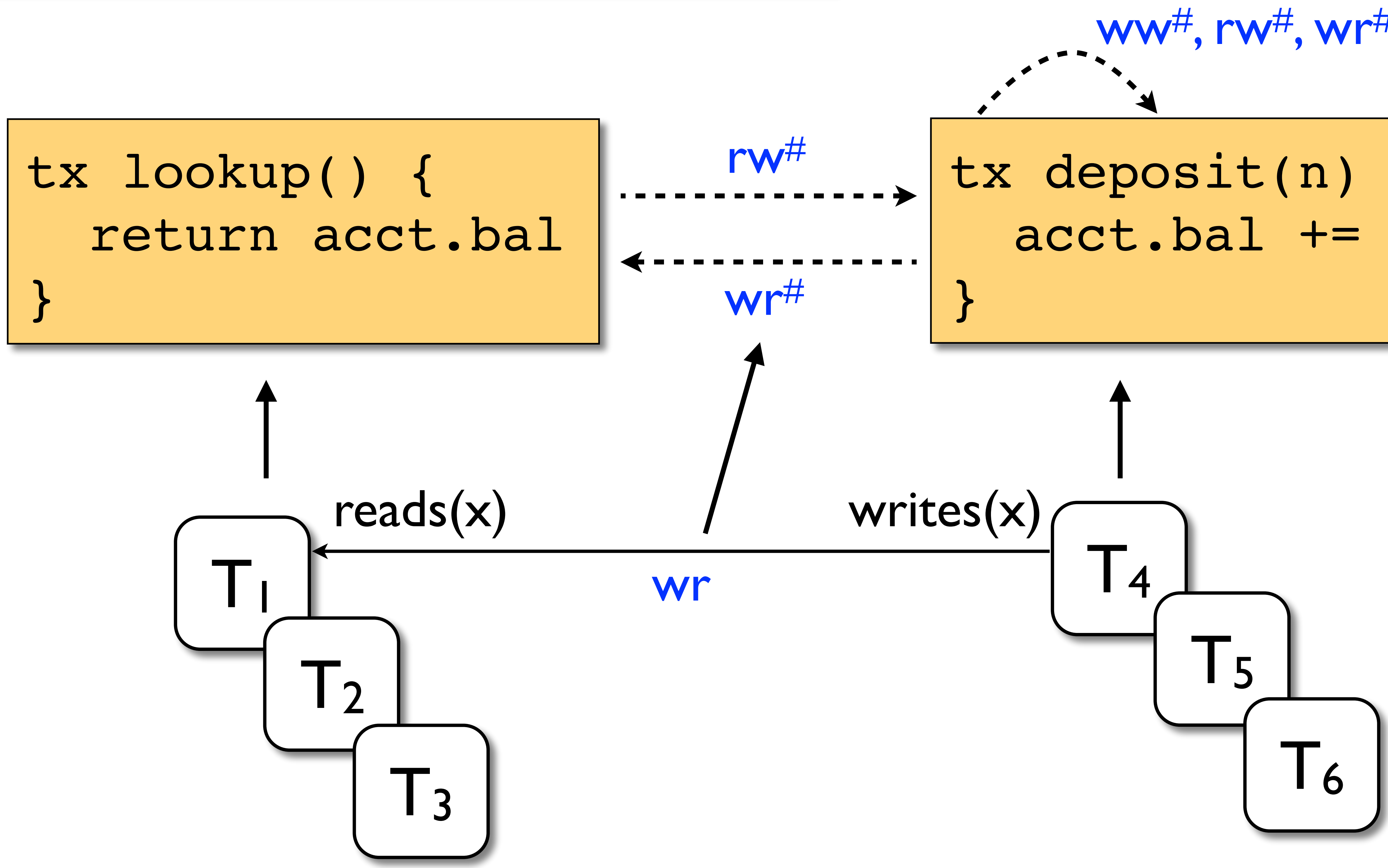
Dynamic dependency graph ➜ a subgraph of the static dependency graph



Cycle in the dynamic graph ➜ cycle in the static graph
If the static graph is acyclic, so is the dynamic one

We're considering PSI executions:
some cycles can't occur

subgraph of the static



Cycle in the dynamic graph ➜ cycle in the static graph
If the static graph is acyclic, so is the dynamic one

T $\boxed{\text{x.write}(\mathit{val})} \xrightarrow{\text{wr}} \boxed{\text{x.read} : \mathit{val}}$ S

$T \xrightarrow{\text{wr}} S \iff T \neq S \wedge T$ contains the most recent write of an object x visible to a read from x in S according to vis

---

T $\boxed{\text{x.write}(\mathit{old})} \xrightarrow{\text{ww}} \boxed{\text{x.write}(\mathit{new})}$ S

$T \xrightarrow{\text{ww}} S \iff T$ and S contain writes to the same object x and $T \xrightarrow{\text{vis}/\sim} S$

$$T \ \boxed{\mathrm{x.write}(\mathit{val})} \xrightarrow{\mathrm{wr}} \boxed{\mathrm{x.read} : \mathit{val}} \ S$$

$T \xrightarrow{\mathrm{wr}} S \iff T \neq S \wedge$ T contains the most recent write of an object x visible to a read from x in S according to vis

---

$$T \ \boxed{\mathrm{x.write}(\mathit{old})} \xrightarrow{\mathrm{ww}} \boxed{\mathrm{x.write}(\mathit{new})} \ S$$

$T \xrightarrow{\mathrm{ww}} S \iff$ T and S contain writes to the same object x and $T \xrightarrow{\mathrm{vis}/\sim} S$

$\mathrm{wr} \cup \mathrm{ww} \subseteq \mathrm{vis}/\sim$ - acyclic

$$T \ \boxed{\text{x.write(}\mathit{val}\text{)}} \xrightarrow{\quad \text{wr} \quad} \boxed{\text{x.read : }\mathit{val}} \ S$$

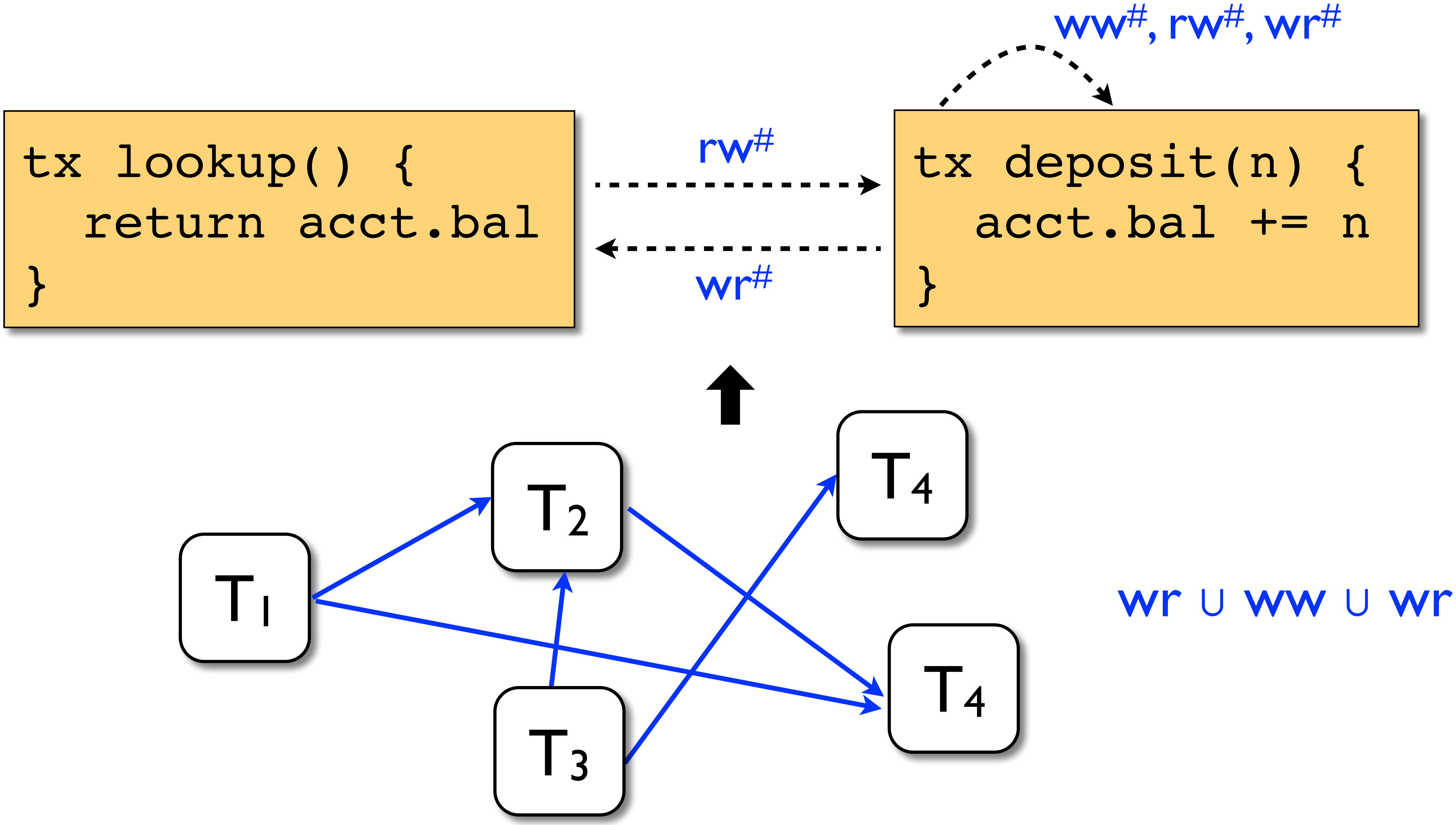
$T \xrightarrow{\text{wr}} S \iff T \neq S \wedge T$ contains the most recent write of an object x visible to a read from x in S according to vis

$$T \ \boxed{\text{x.write(}\mathit{old}\text{)}} \xrightarrow{\quad \text{ww} \quad} \boxed{\text{x.write(}\mathit{new}\text{)}} \ S$$

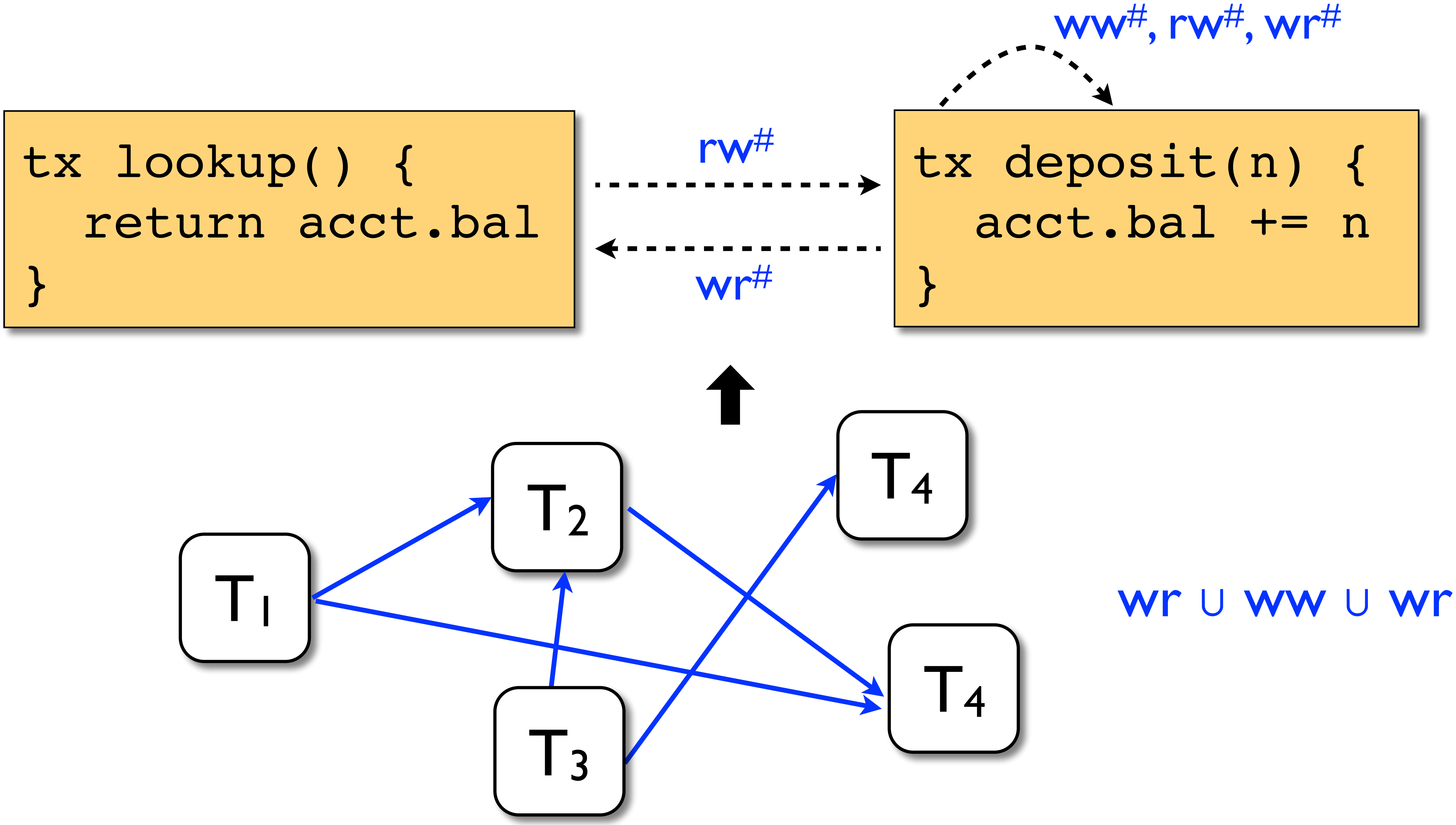
$T \xrightarrow{\text{ww}} S \iff T$ and S contain writes to the same object x and $T \xrightarrow{\text{vis}/\sim} S$

$\text{wr} \cup \text{ww} \subseteq \text{vis}/\sim$ - acyclic

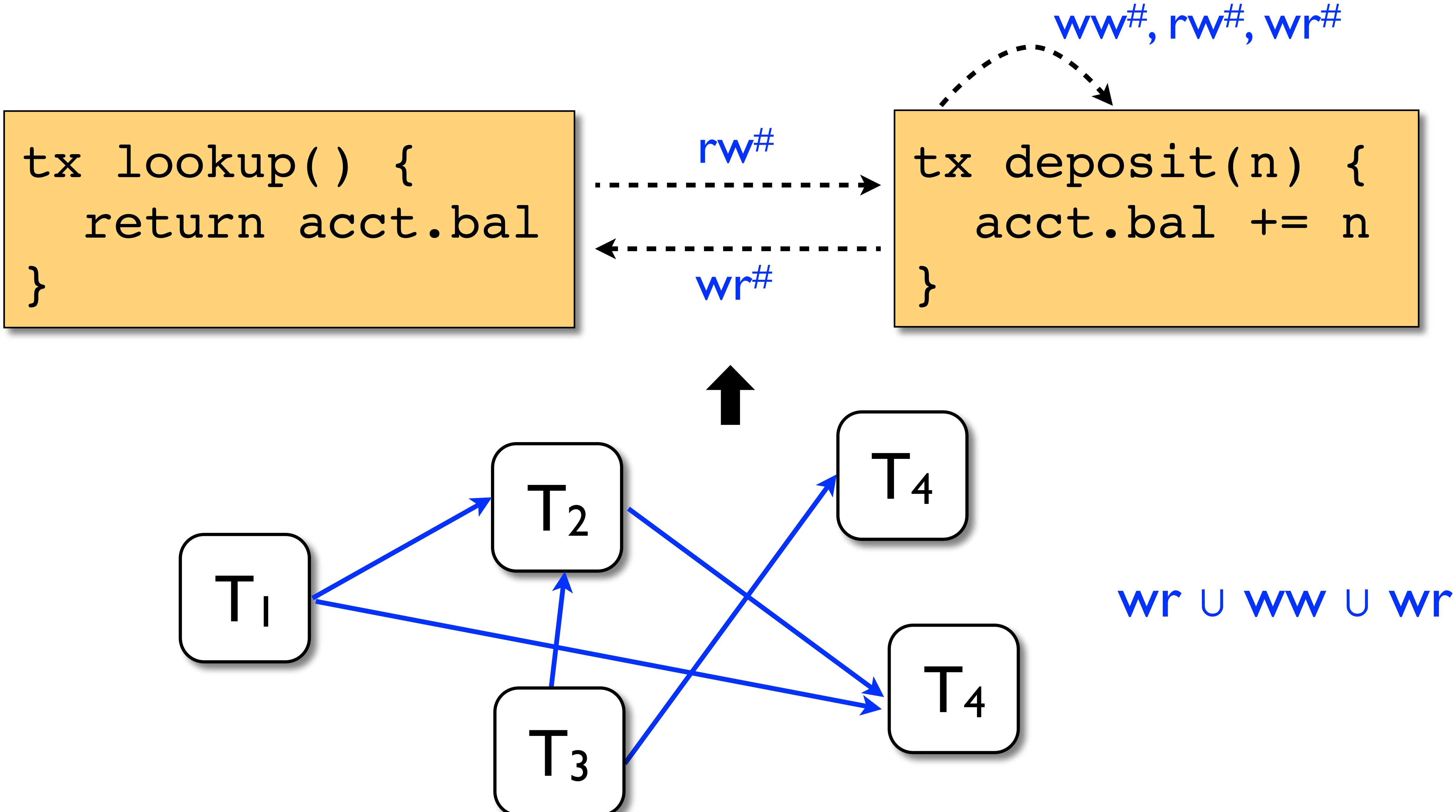
PSI allows only cycles in $(\text{wr} \cup \text{ww} \cup \text{rw})$ with at least one rw edge



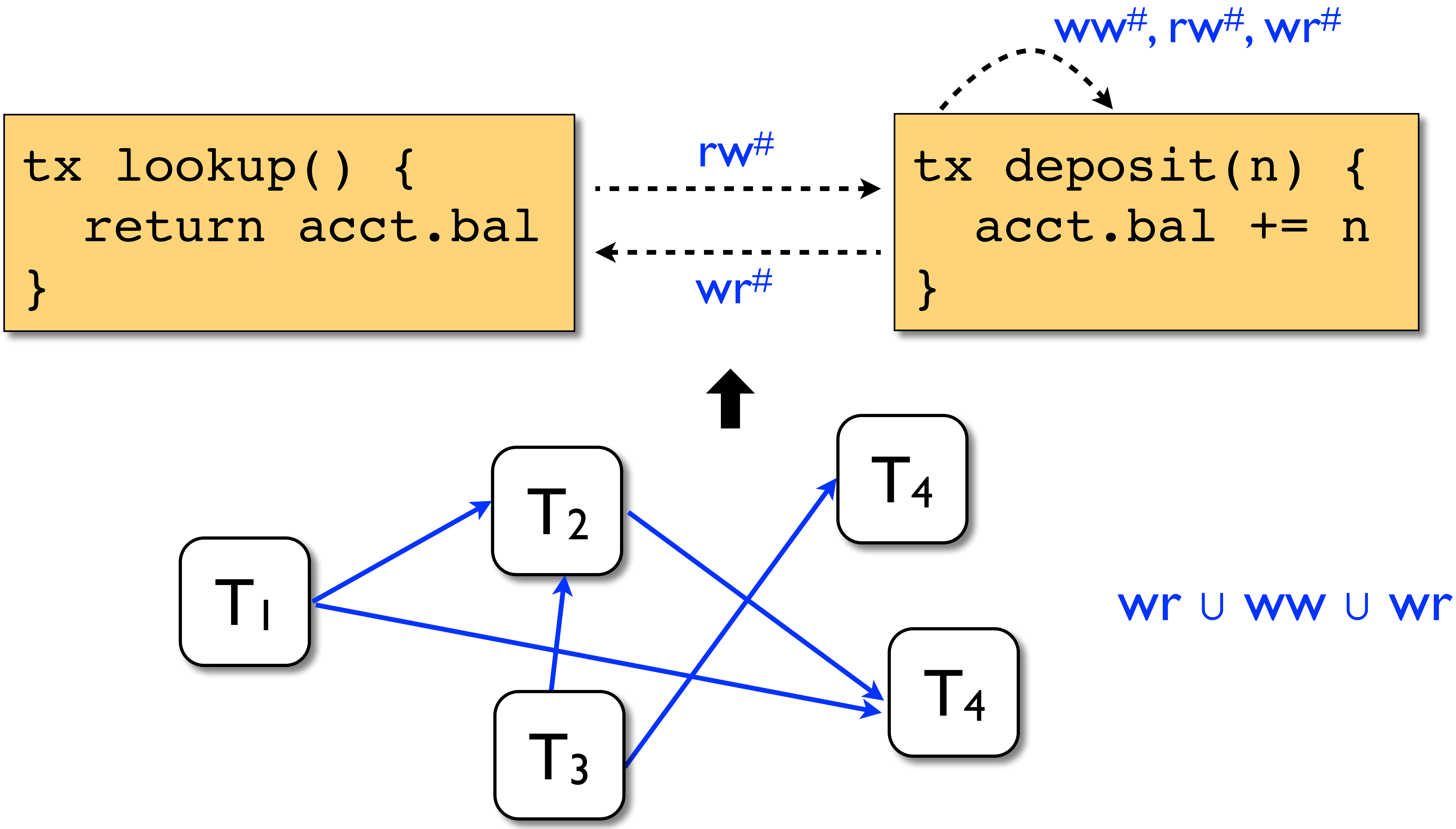
Dynamic dependency graph ➜ a subgraph of the static dependency graph



- Dynamic cycles with no rw edges aren't PSI ➜ don't represent robustness violations



- Dynamic cycles with no rw edges aren't PSI ➜ don't represent robustness violations
- Enough to check no cycles in (wr ∪ ww ∪ rw) with ≥ 1 rw



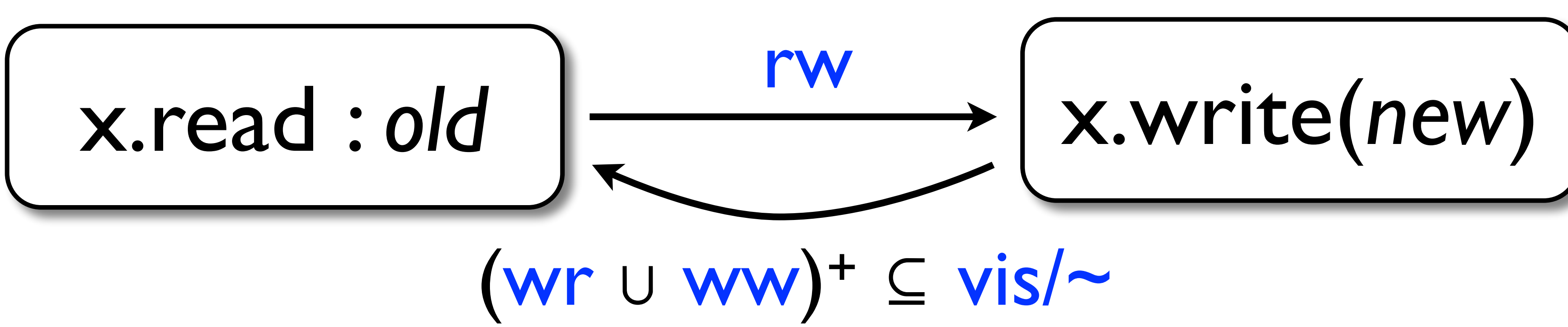
- Dynamic cycles with no rw edges aren't PSI ➜ don't represent robustness violations
- Enough to check no cycles in (wr ∪ ww ∪ rw) with ≥ 1 rw
- Enough to check no cycles in ($wr^{\#}$ ∪ $ww^{\#}$ ∪ $rw^{\#}$) with ≥ 1 $rw^{\#}$

Tightening up the criterion

PSI allows only cycles in (wr ∪ ww ∪ wr) with at least two distinct rw edges

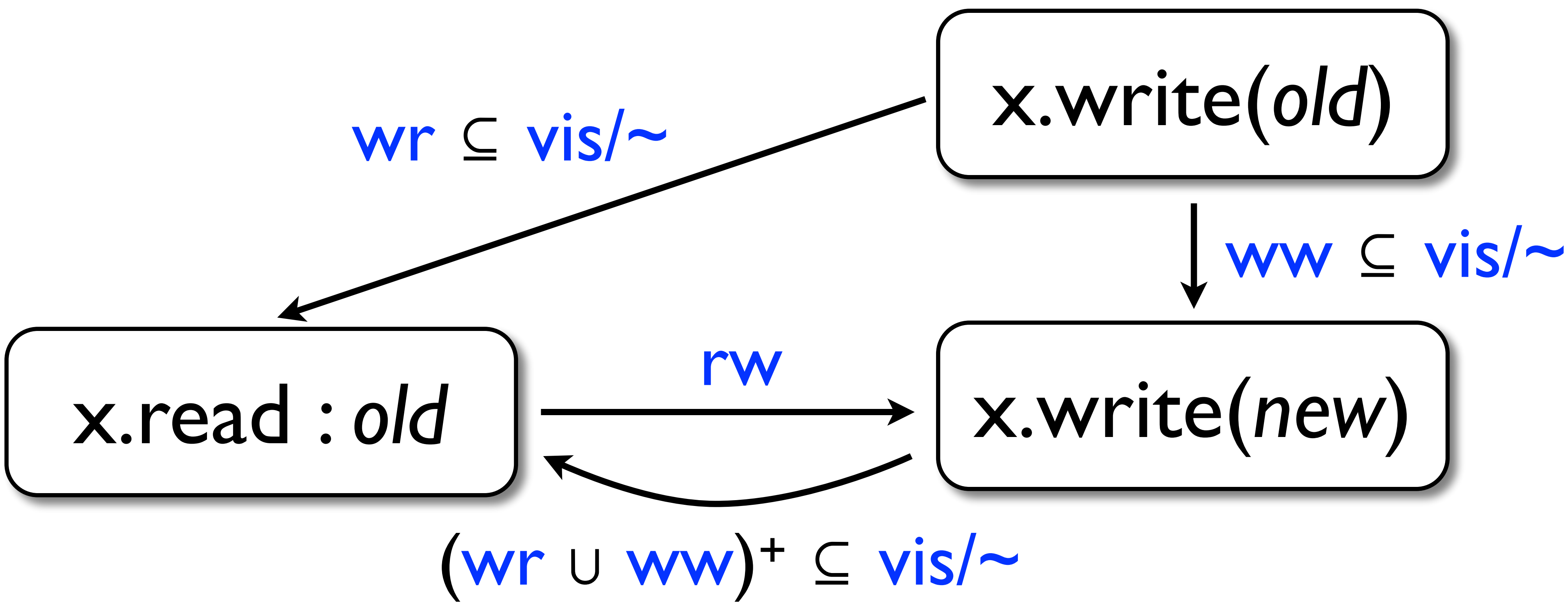
Tightening up the criterion

PSI allows only cycles in ($\mathsf{wr} \cup \mathsf{ww} \cup \mathsf{wr}$) with at least two distinct rw edges



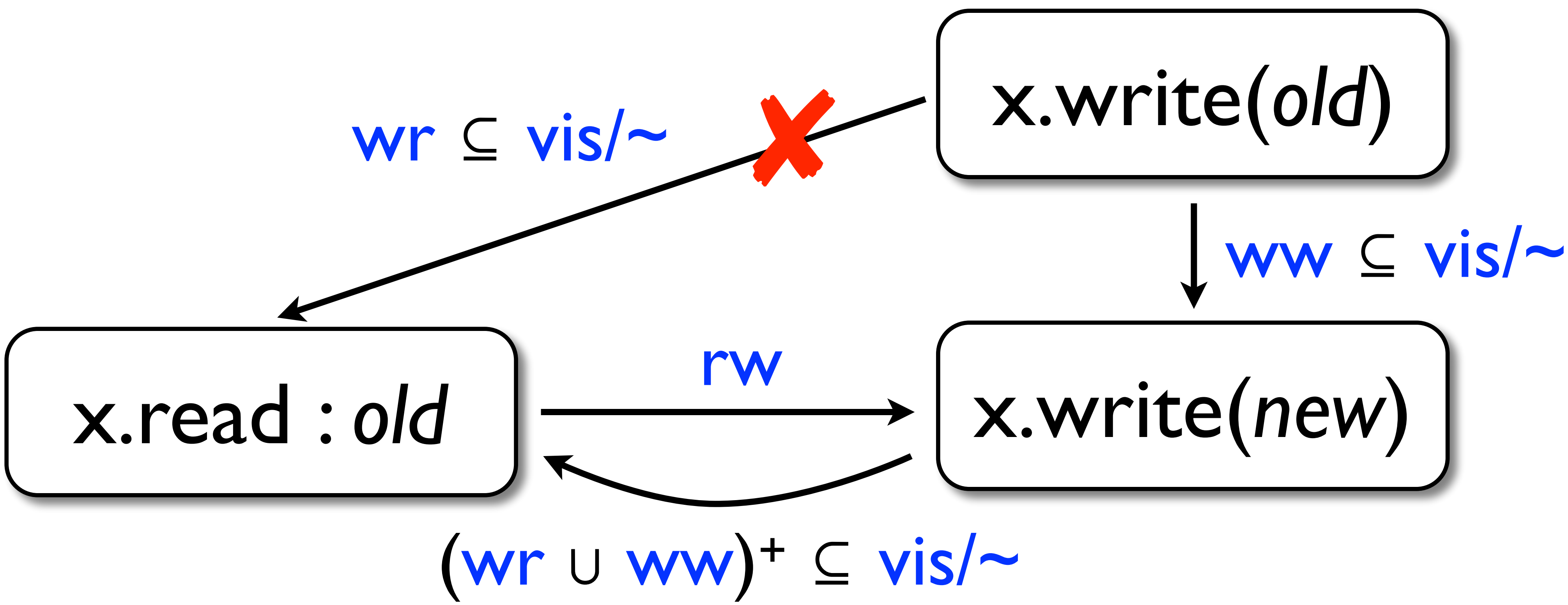
Tightening up the criterion

PSI allows only cycles in (wr ∪ ww ∪ wr) with at least two distinct rw edges



Tightening up the criterion

PSI allows only cycles in (wr ∪ ww ∪ wr) with at least two distinct rw edges



Tightening up the criterion

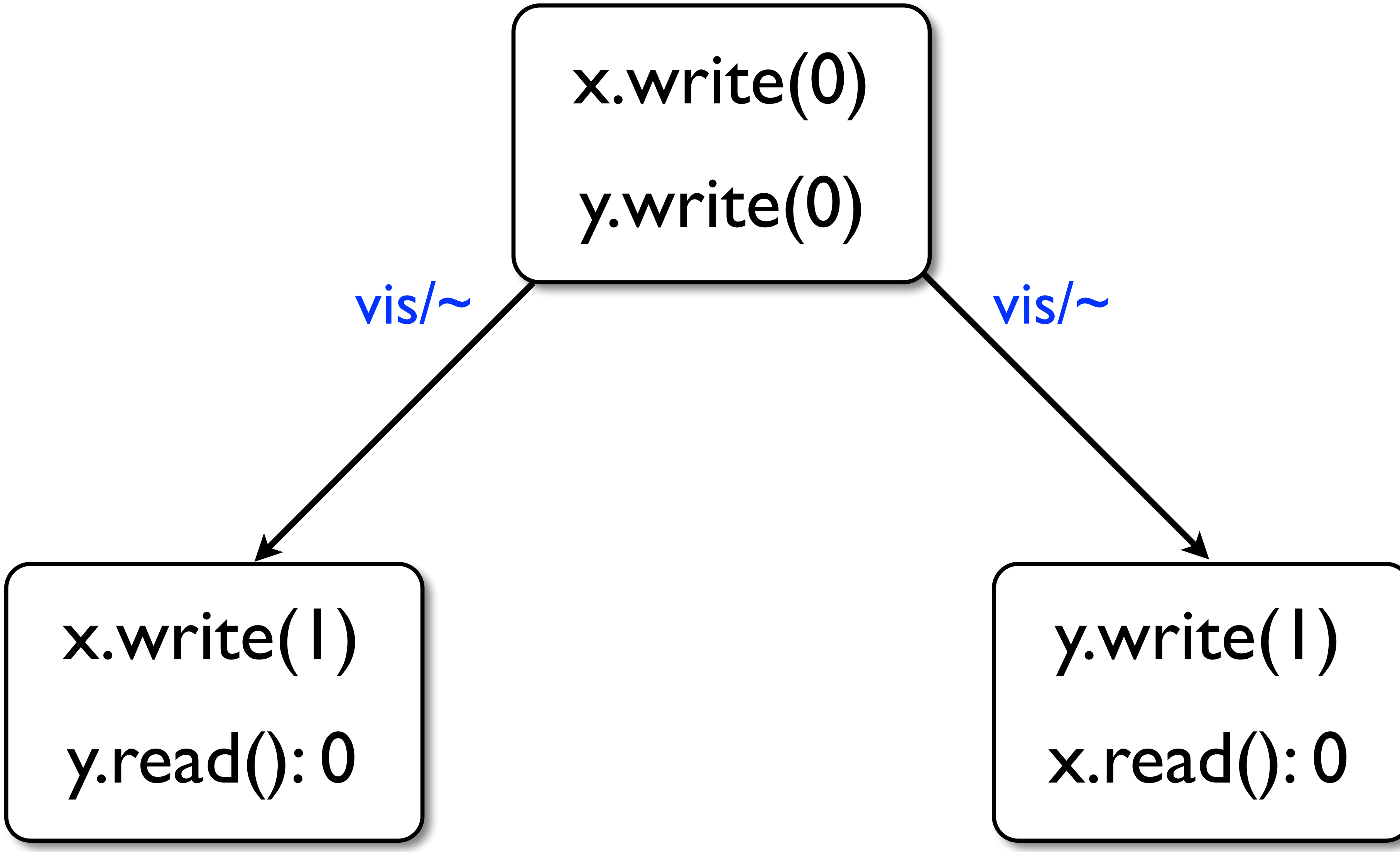
PSI allows only cycles in (wr ∪ ww ∪ wr) with at least two distinct rw edges

⬇

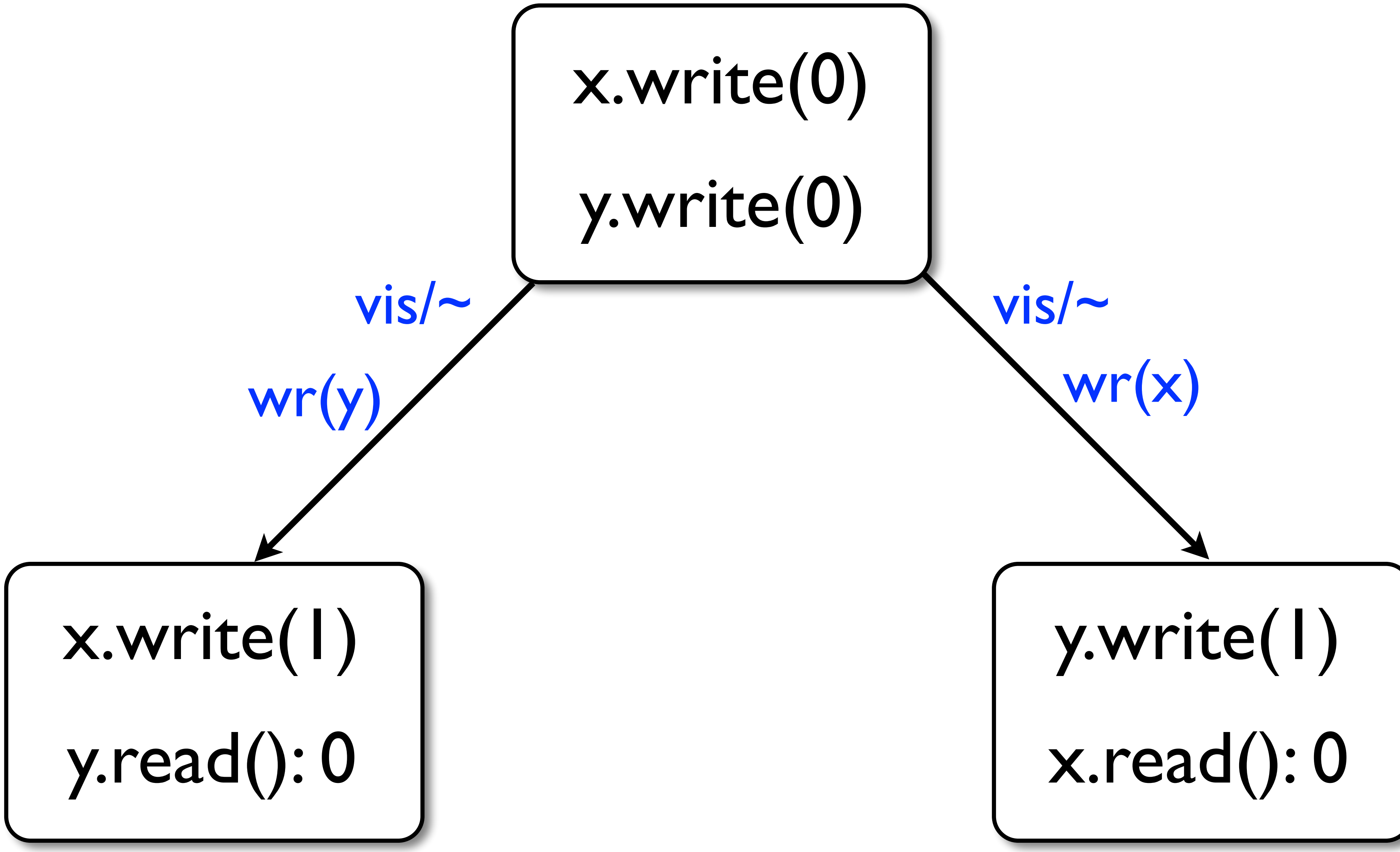
If (wr ∪ ww ∪ wr) for a PSI execution contains a cycle, then it also contains one:

- with at least two rw edges, and
- where all rw edges are due to distinct objects

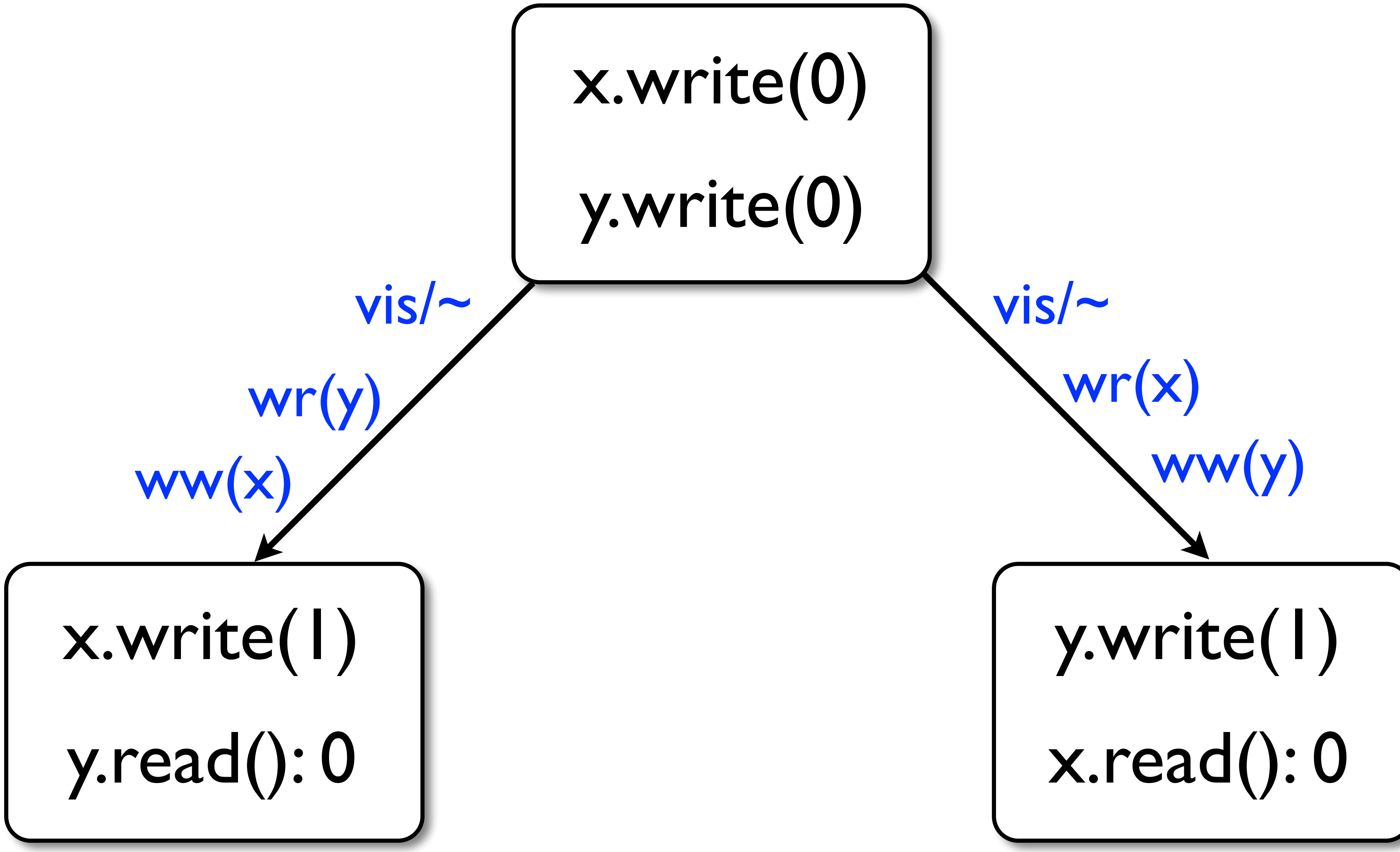
Transactional Dekker = write skew



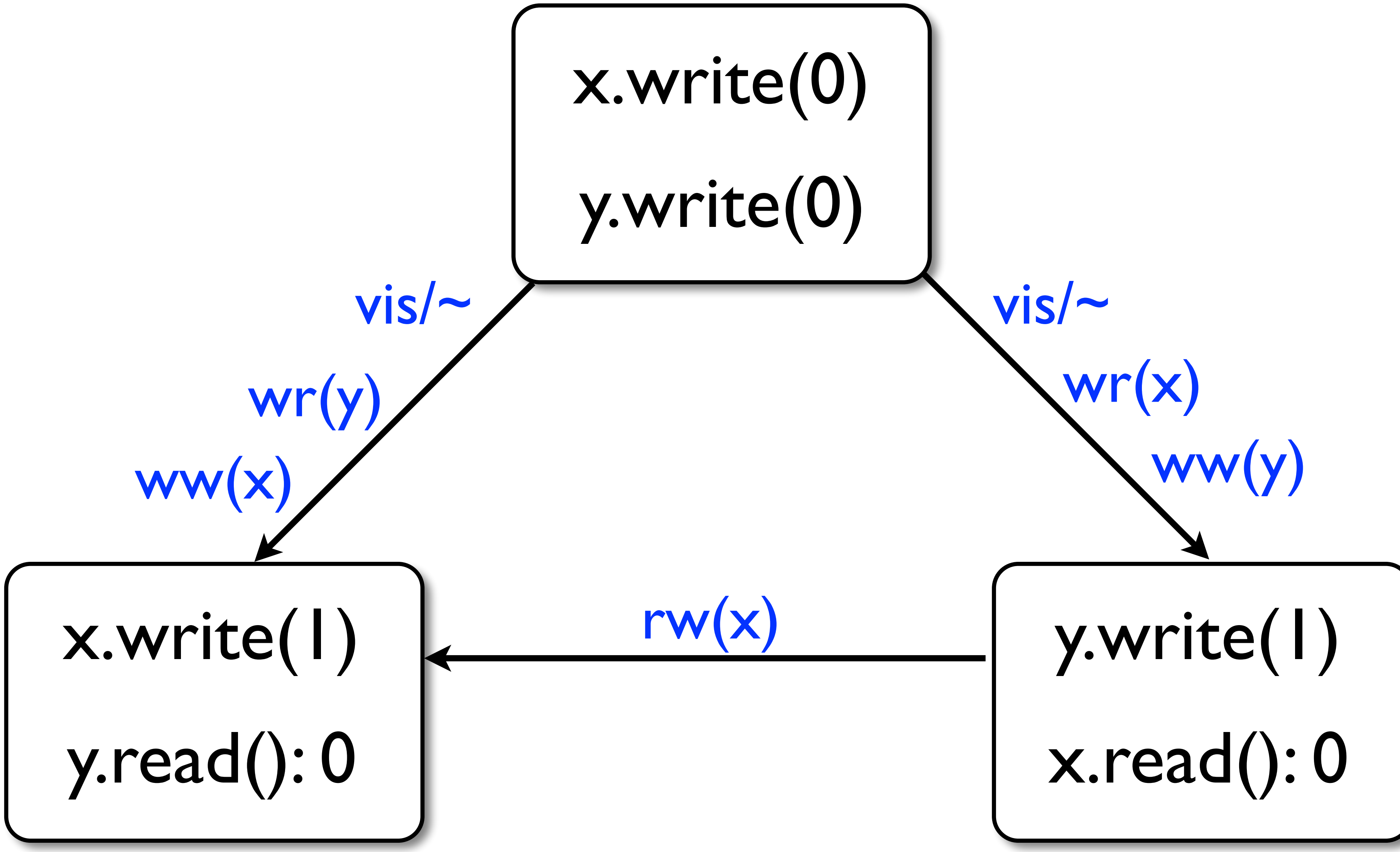
Transactional Dekker = write skew



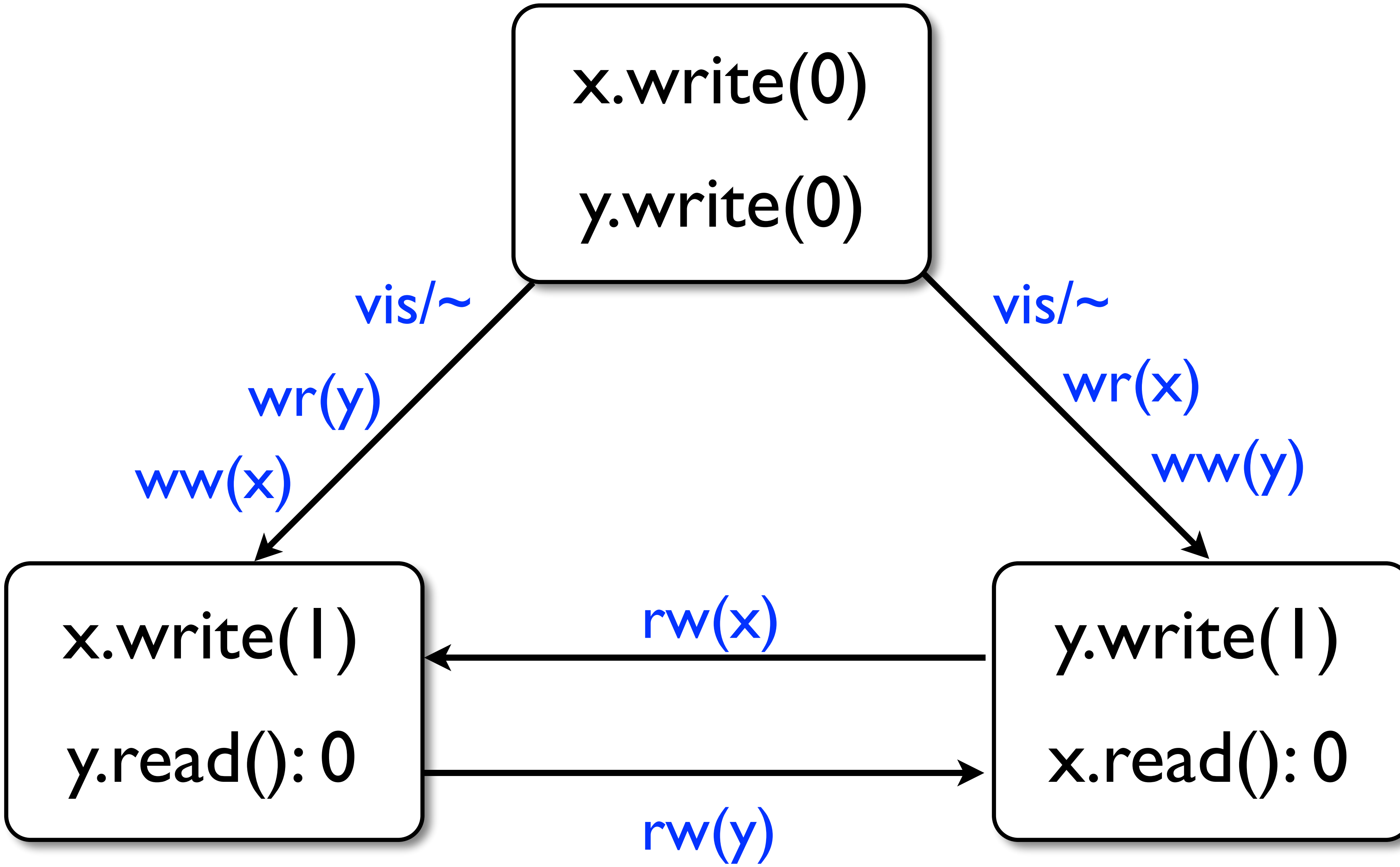
Transactional Dekker = write skew



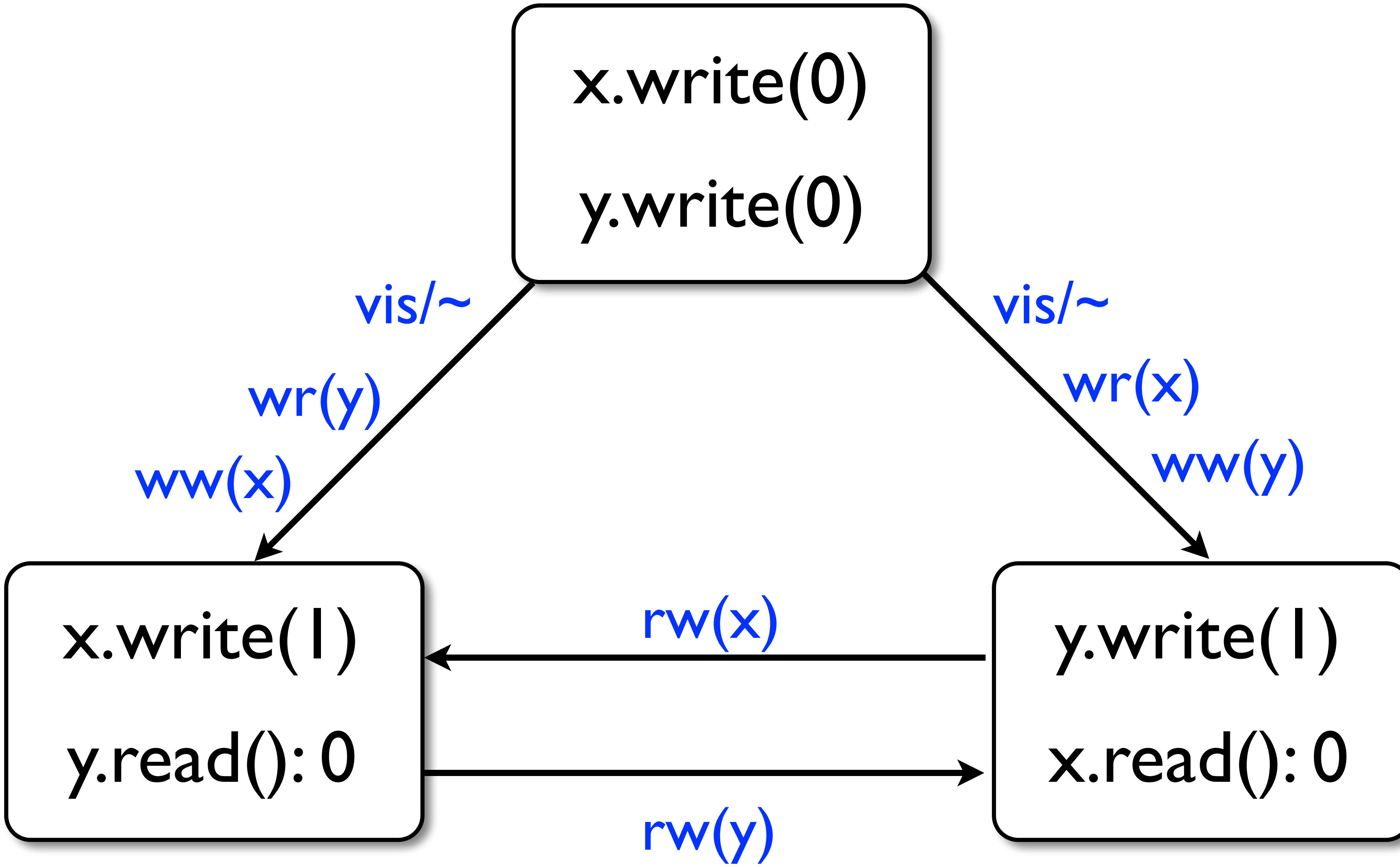
Transactional Dekker = write skew



Transactional Dekker = write skew

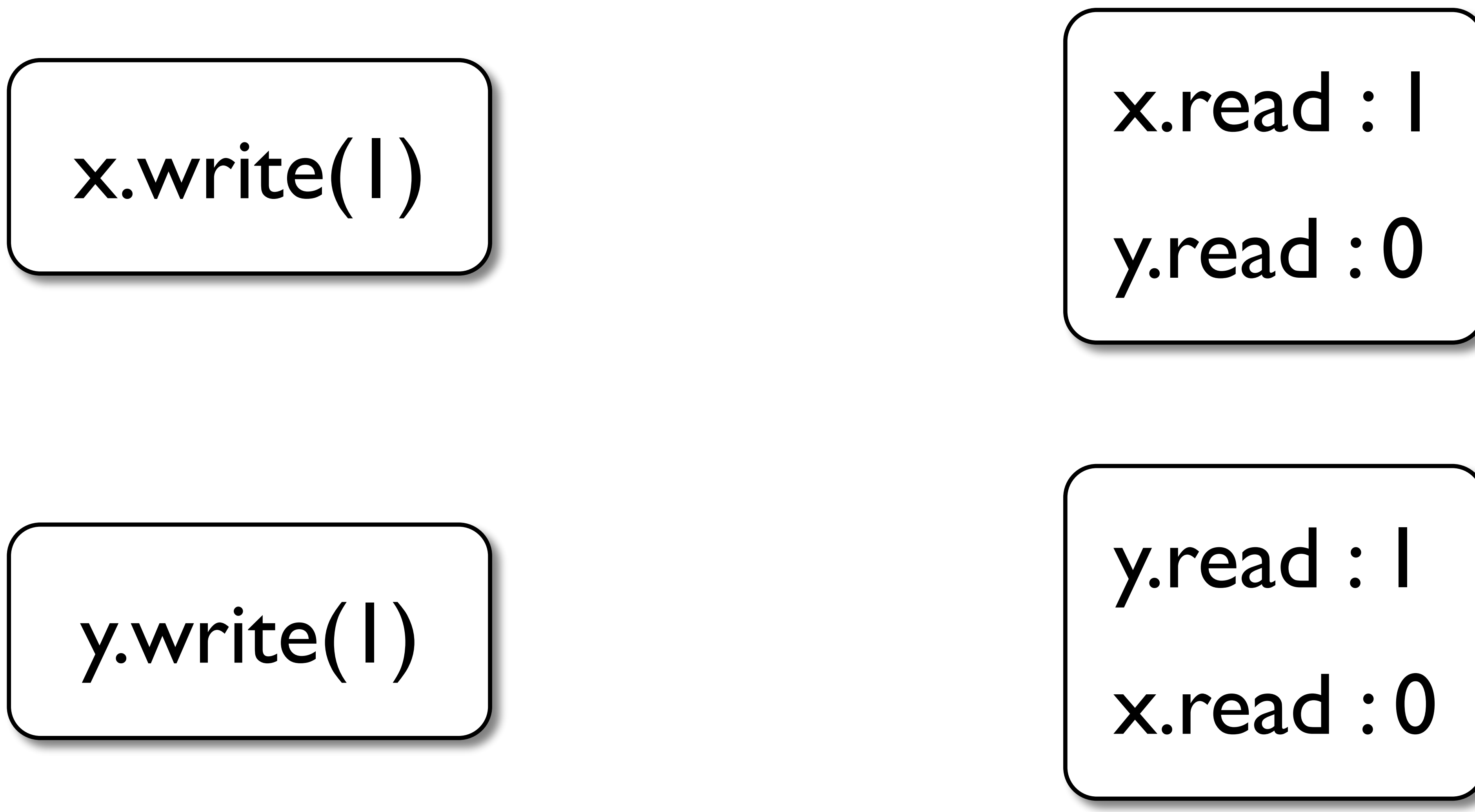


Transactional Dekker = write skew

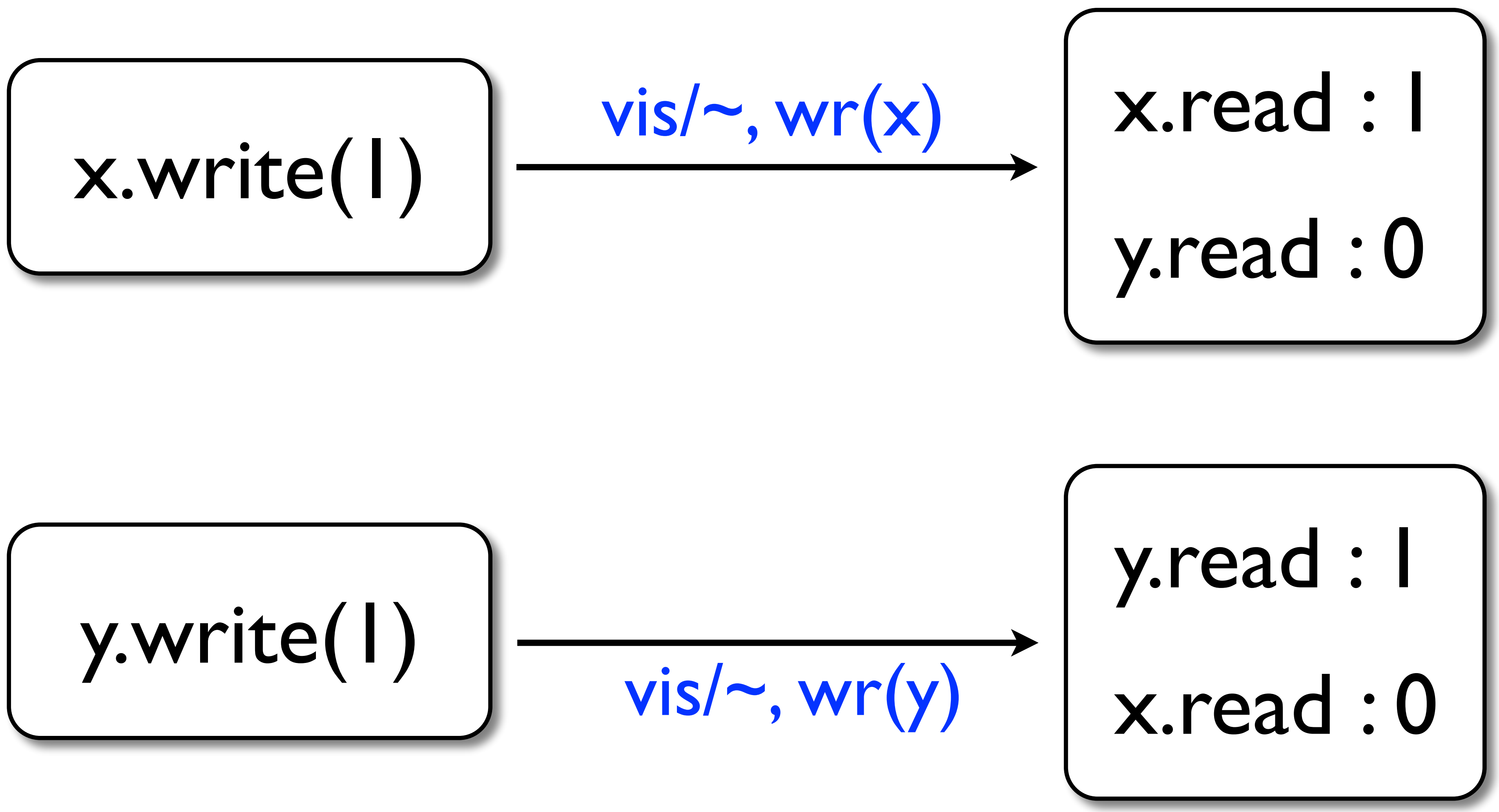


Cycle with 2 rw on different objects: allowed by PSI

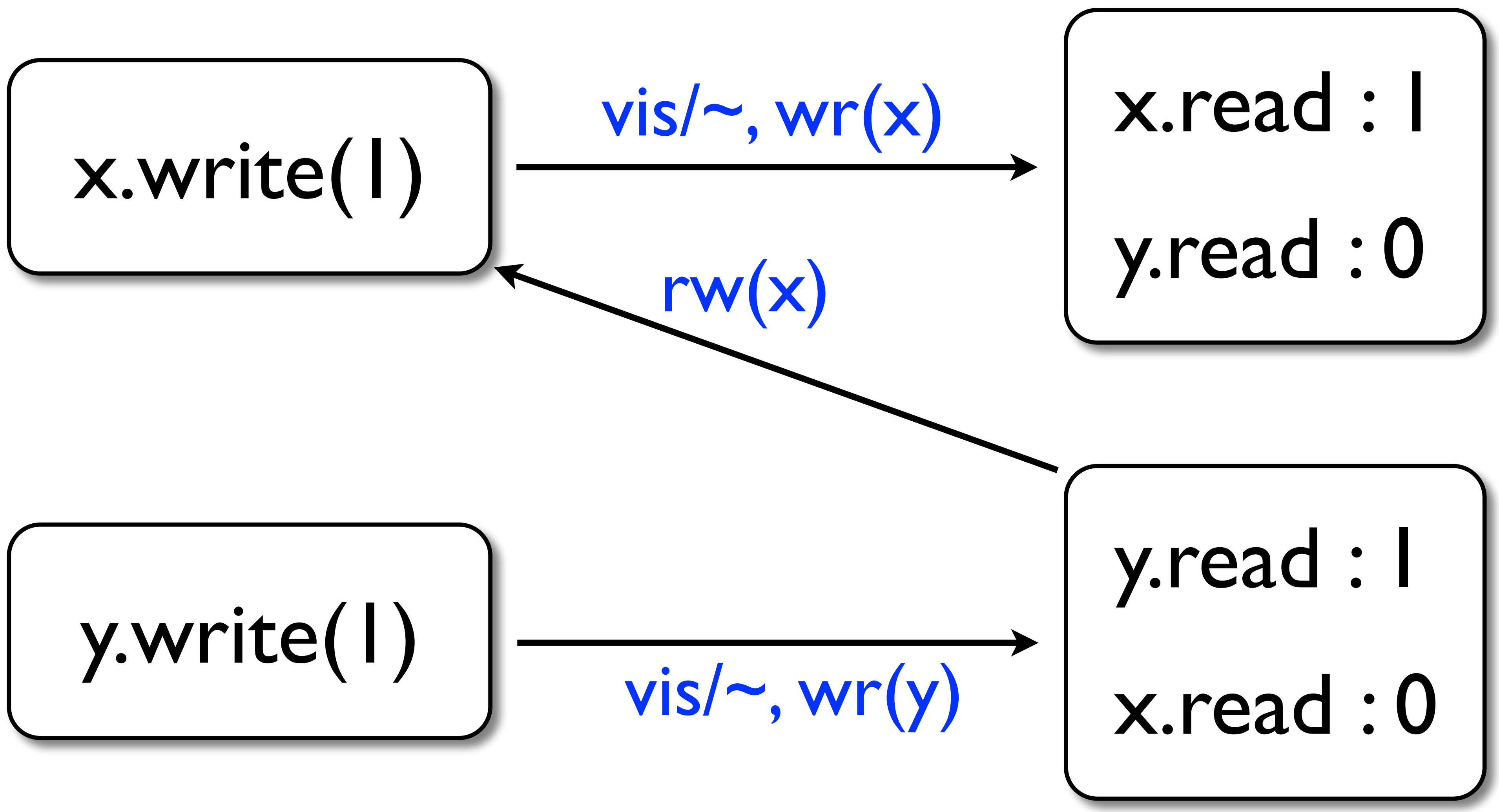
Transactional IRIW = long fork



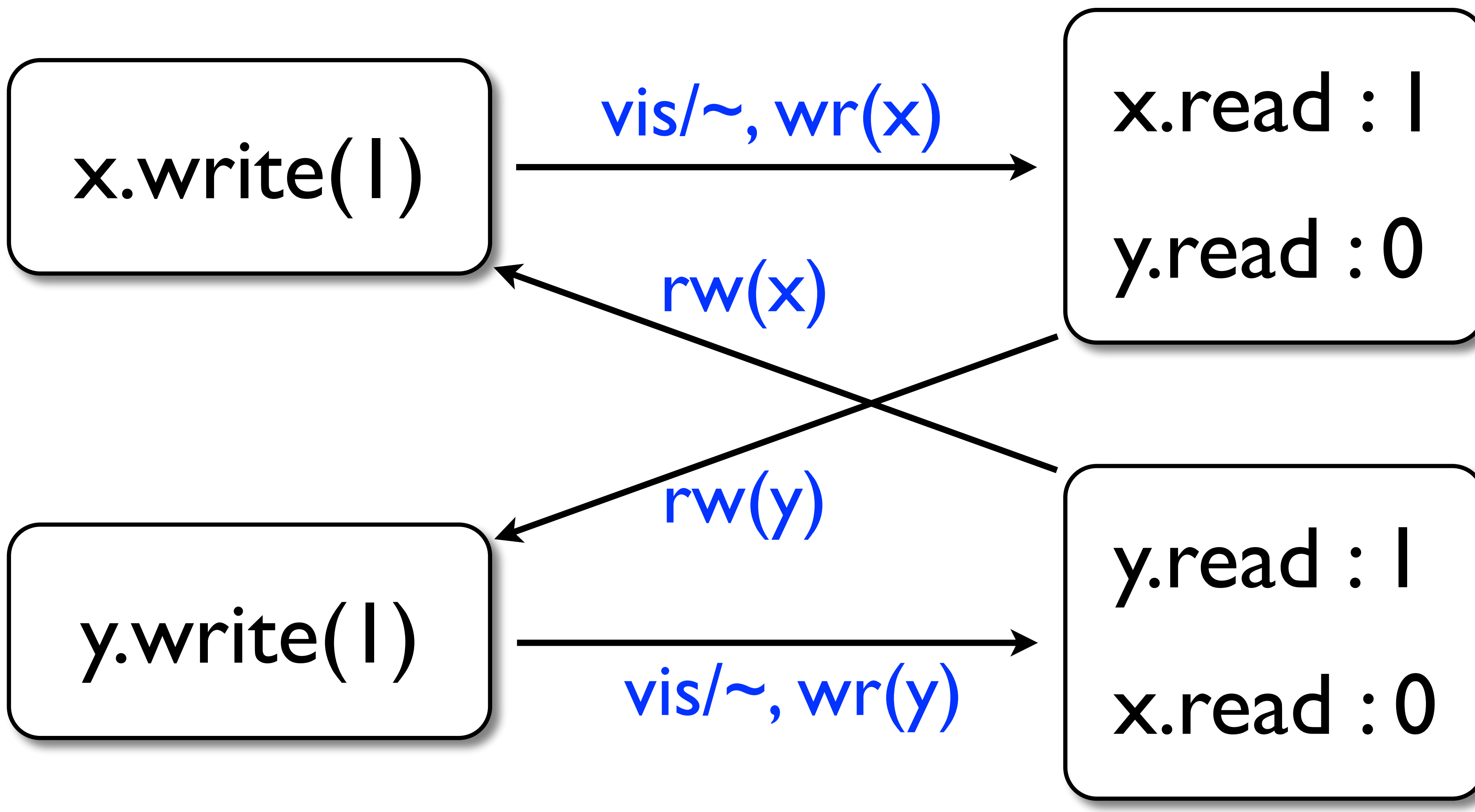
Transactional IRIW = long fork



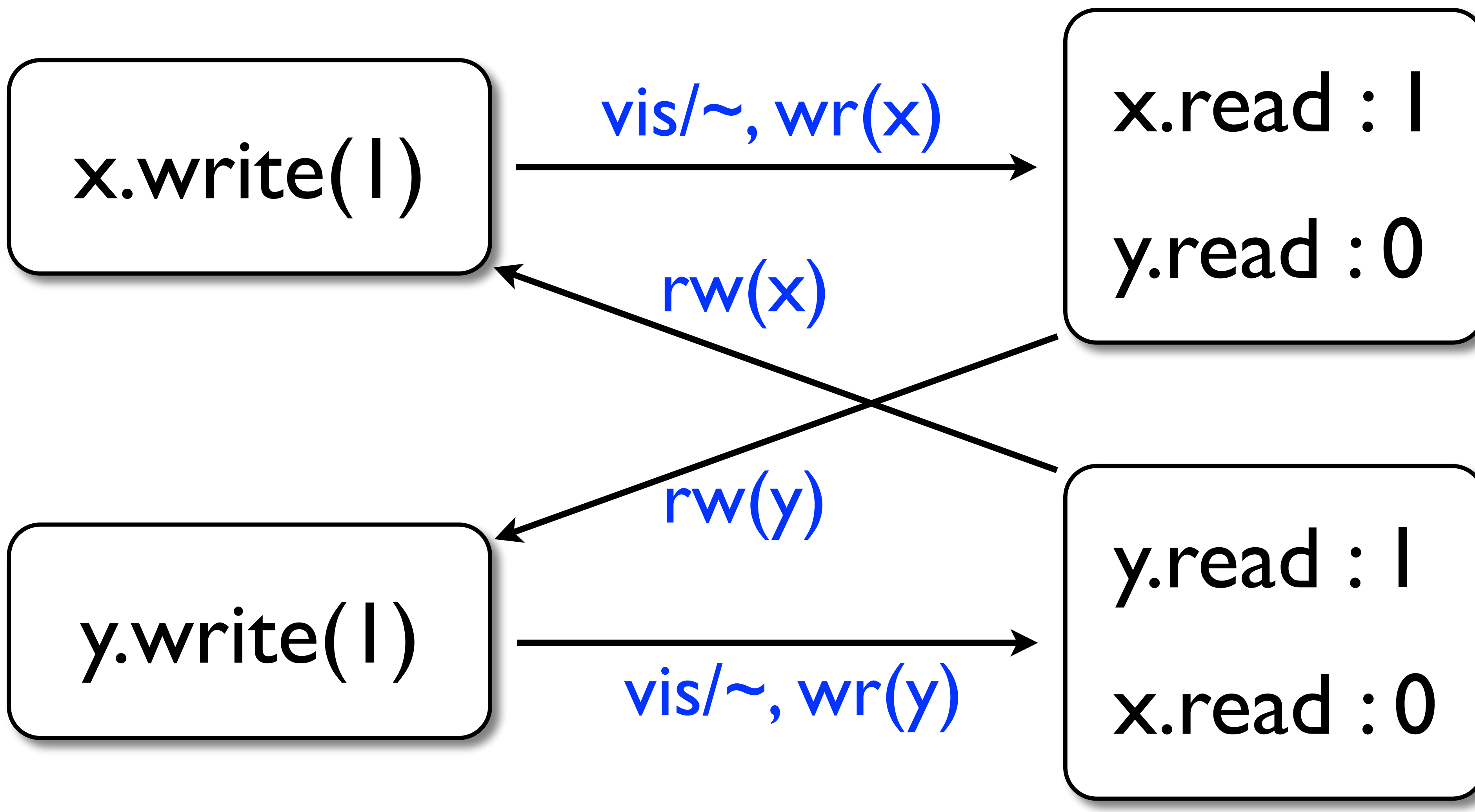
Transactional IRIW = long fork



Transactional IRIW = long fork

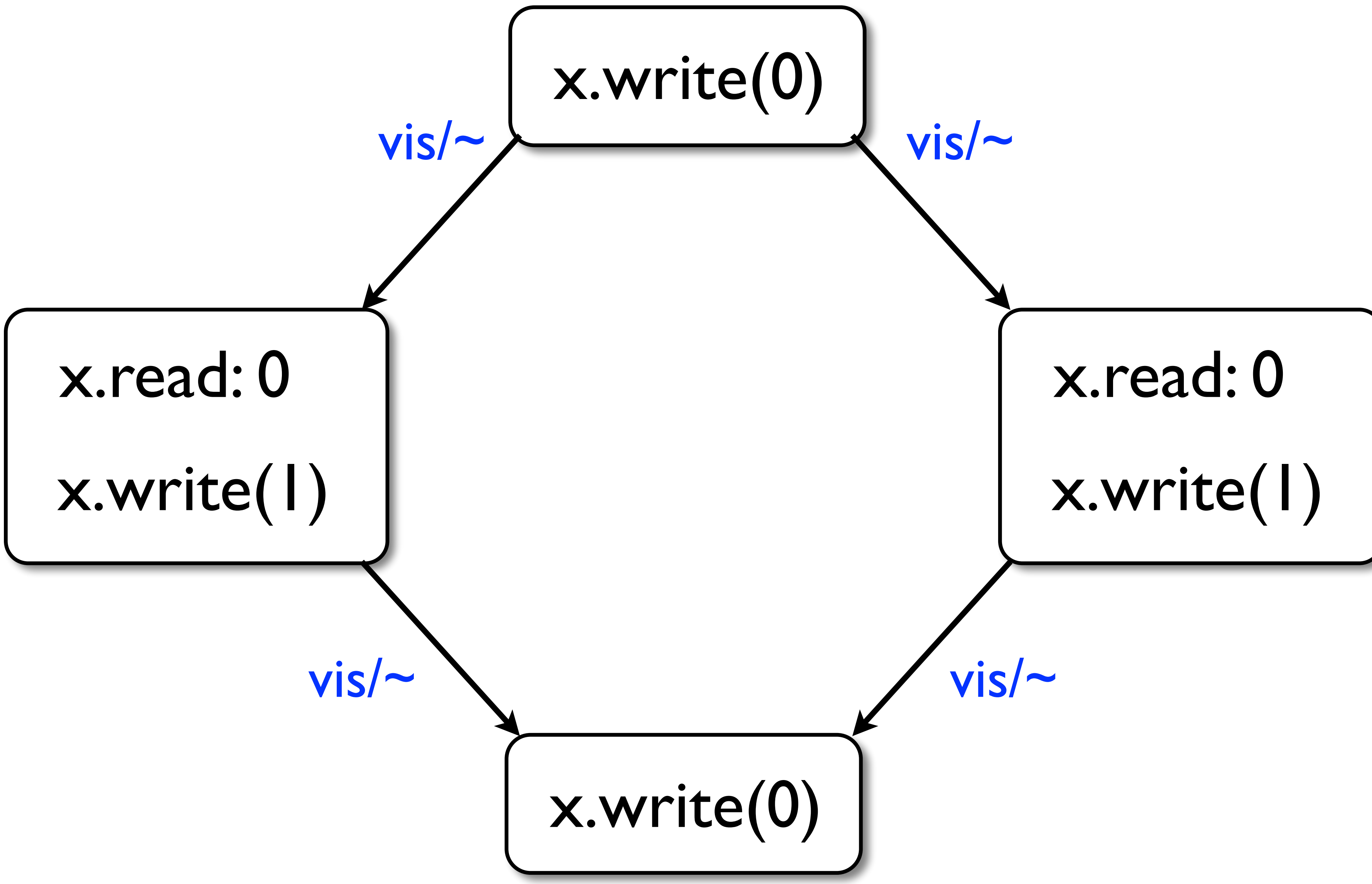


Transactional IRIW = long fork



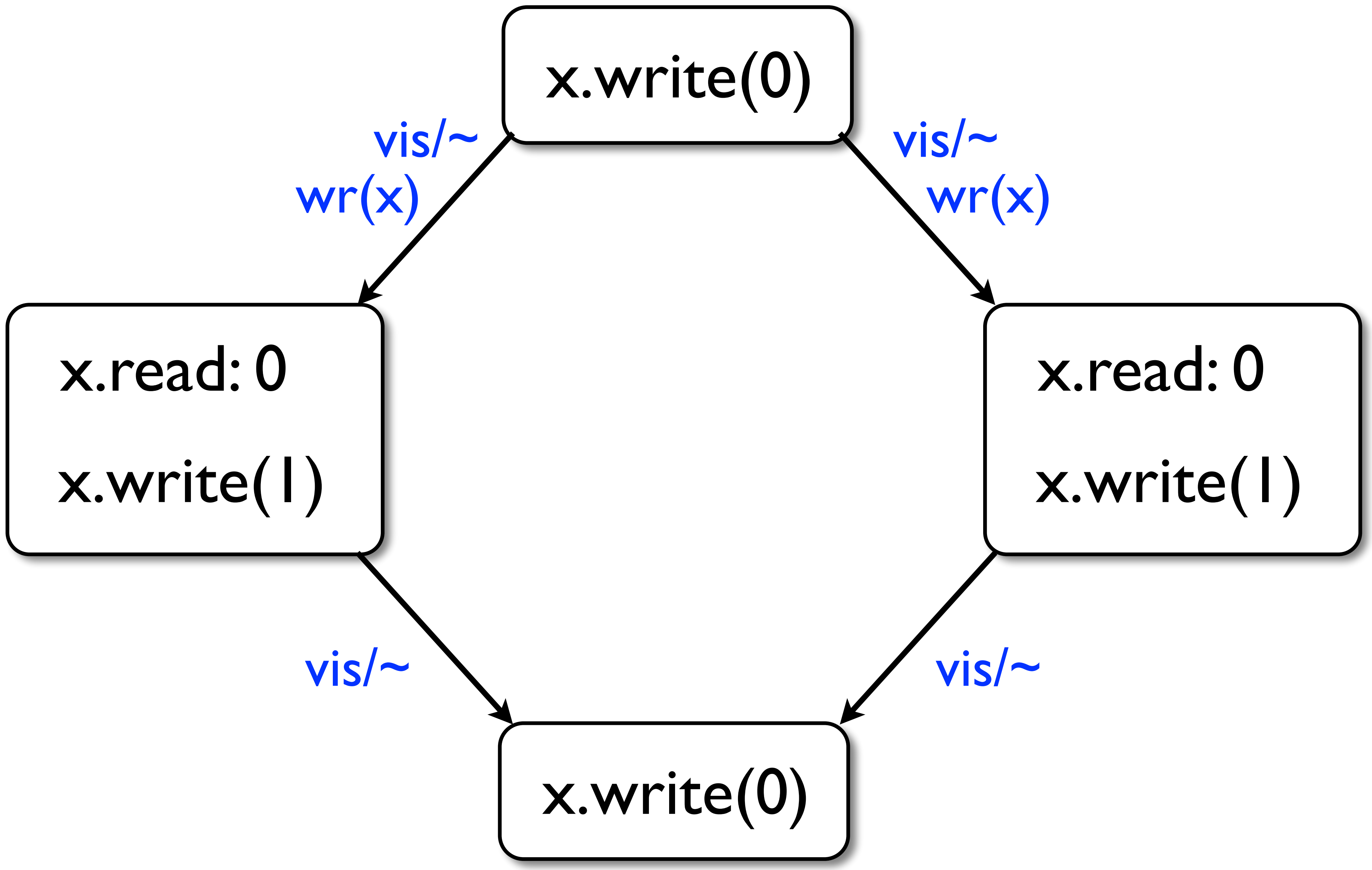
Cycle with 2 rw on different objects: allowed by PSI

Lost update anomaly



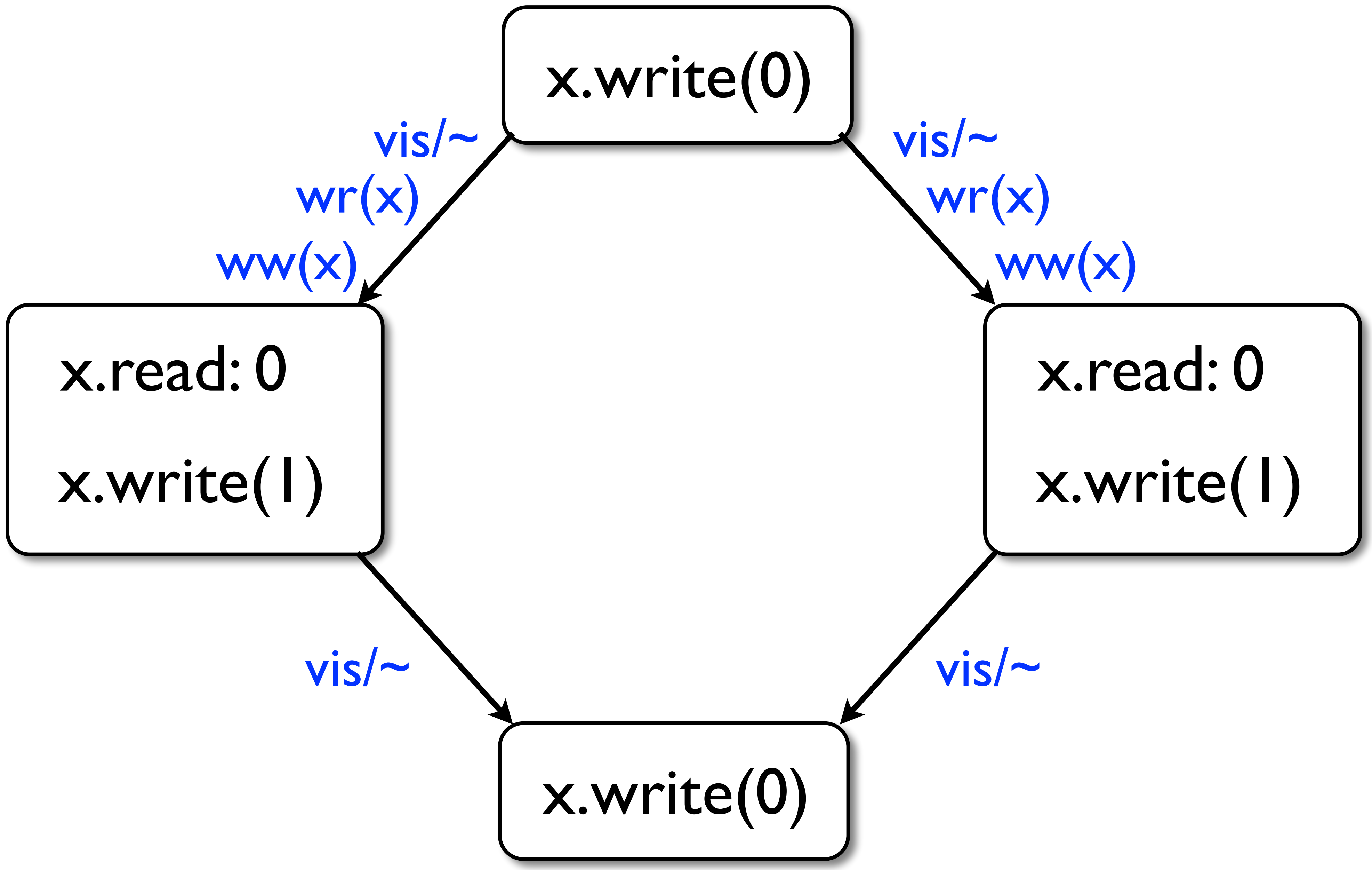
Not a valid PSI execution: violates write-conflict detection

Lost update anomaly



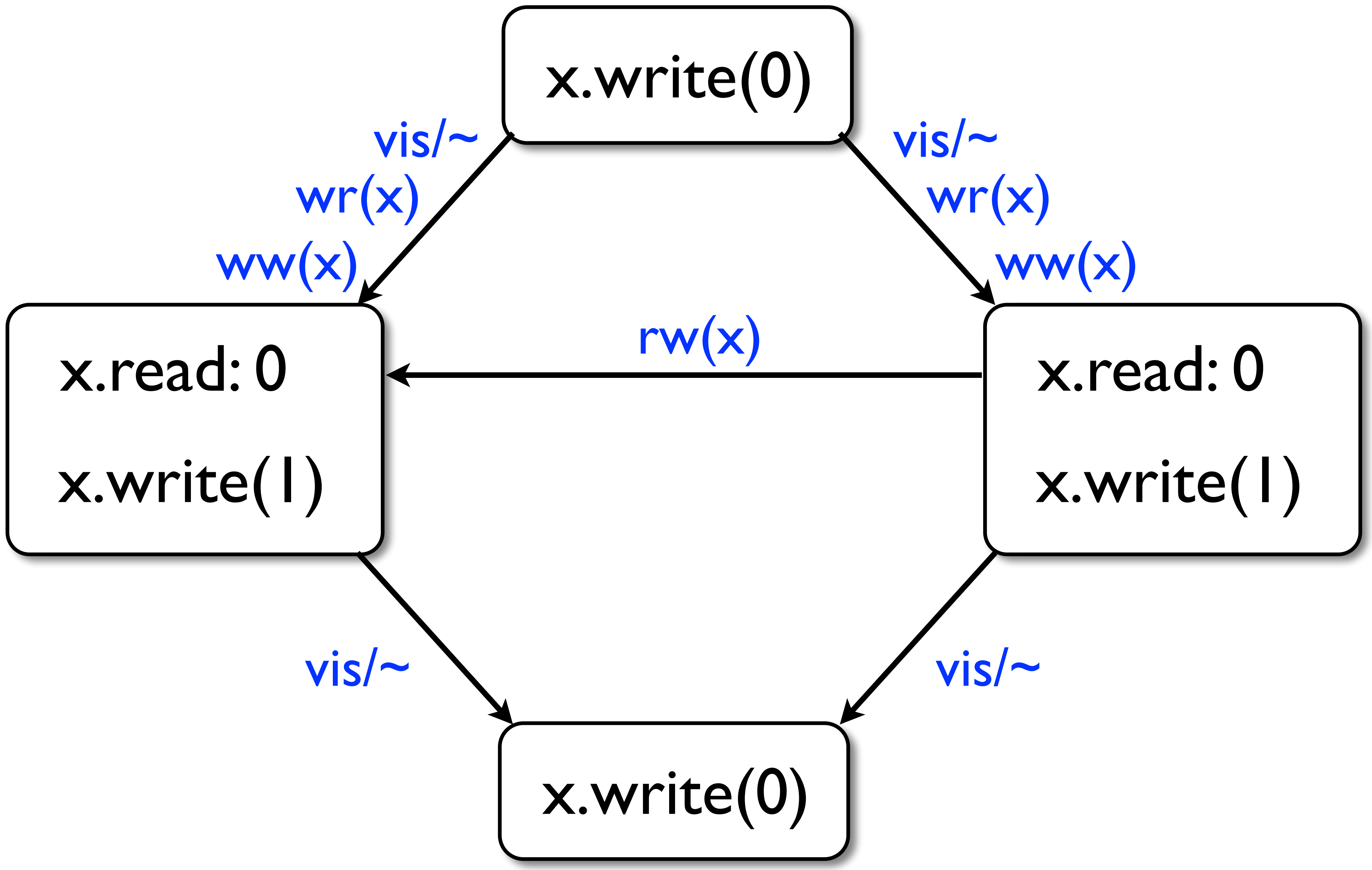
Not a valid PSI execution: violates write-conflict detection

Lost update anomaly



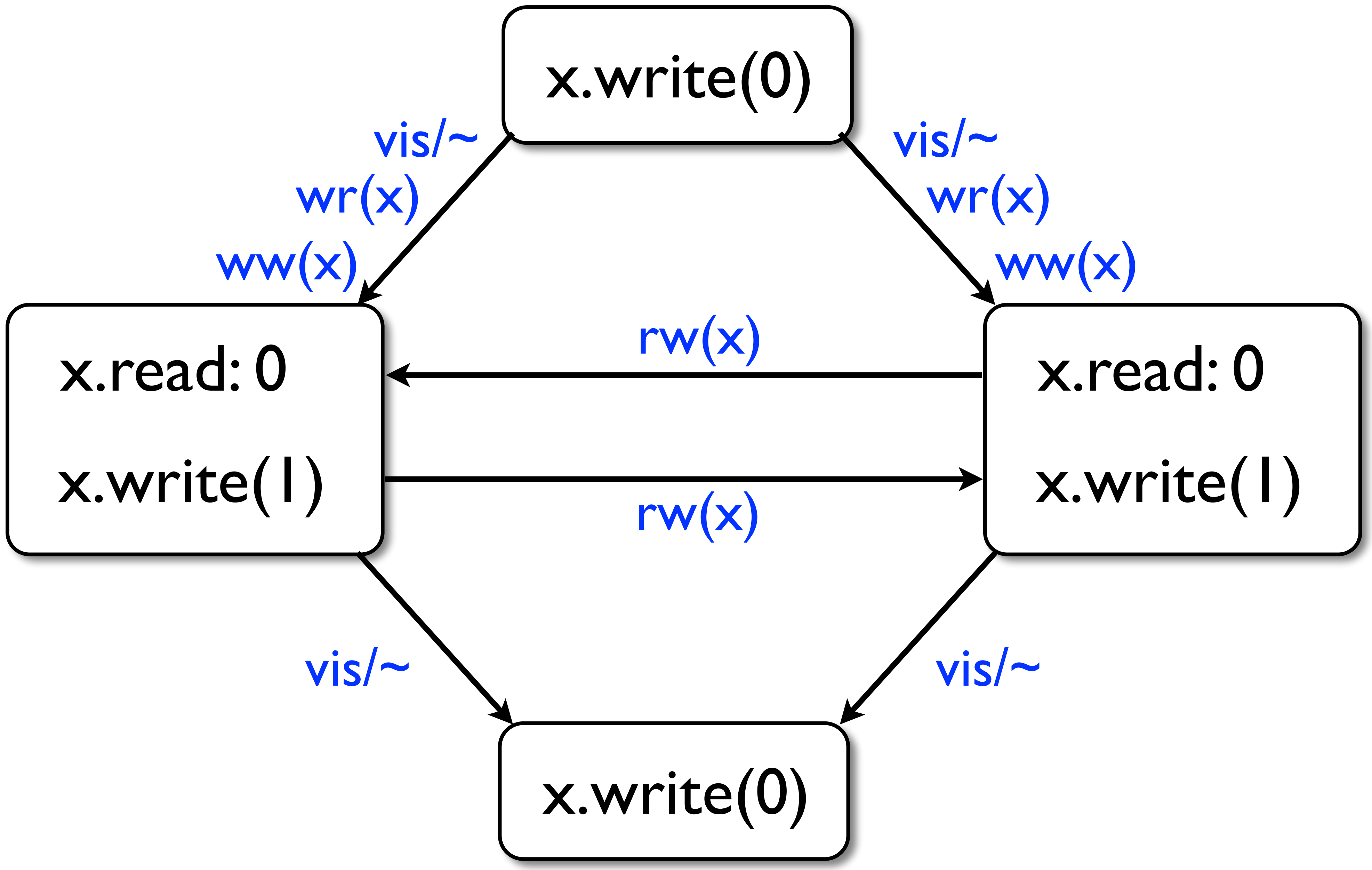
Not a valid PSI execution: violates write-conflict detection

Lost update anomaly



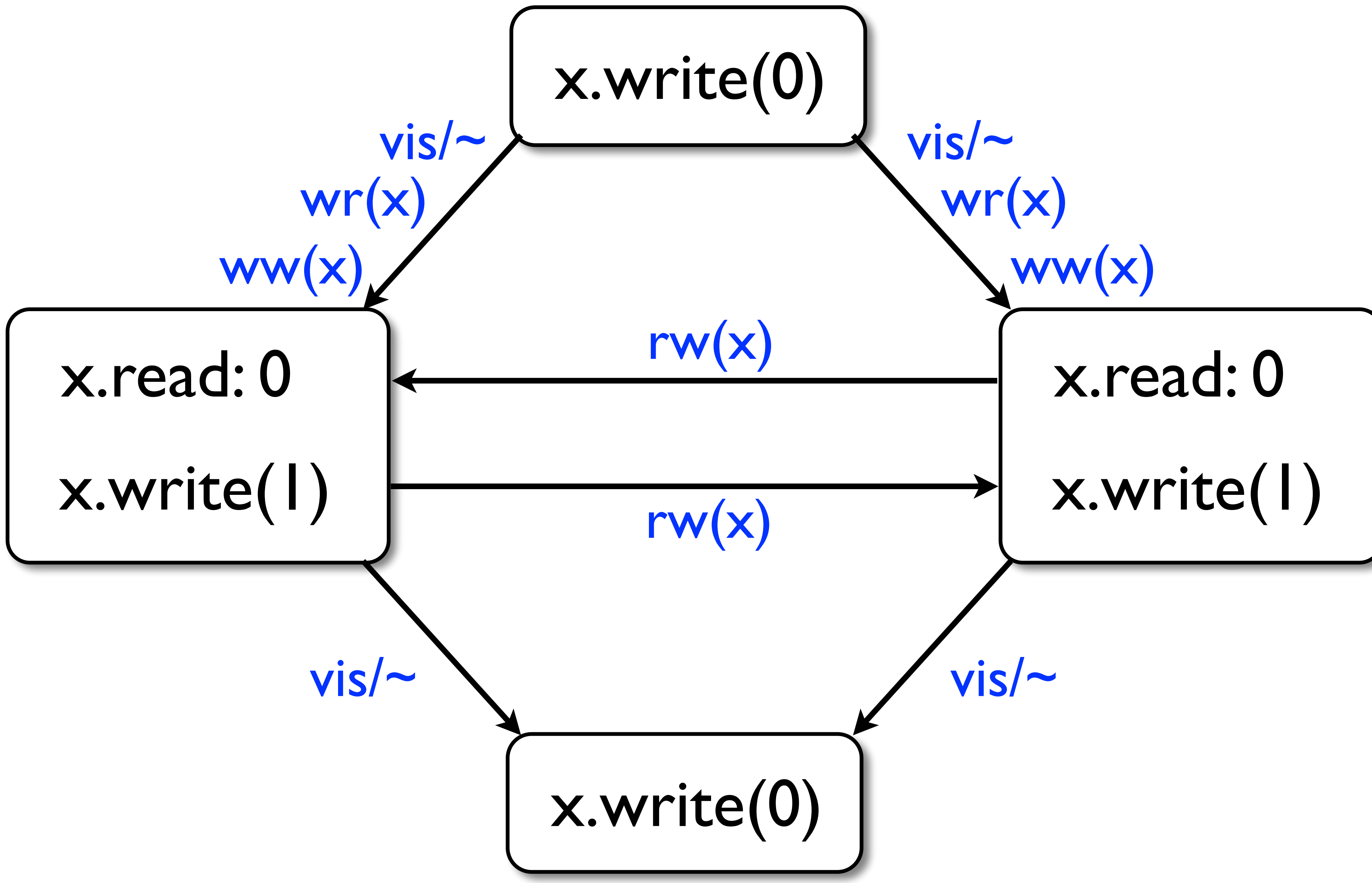
Not a valid PSI execution: violates write-conflict detection

Lost update anomaly



Not a valid PSI execution: violates write-conflict detection

Lost update anomaly



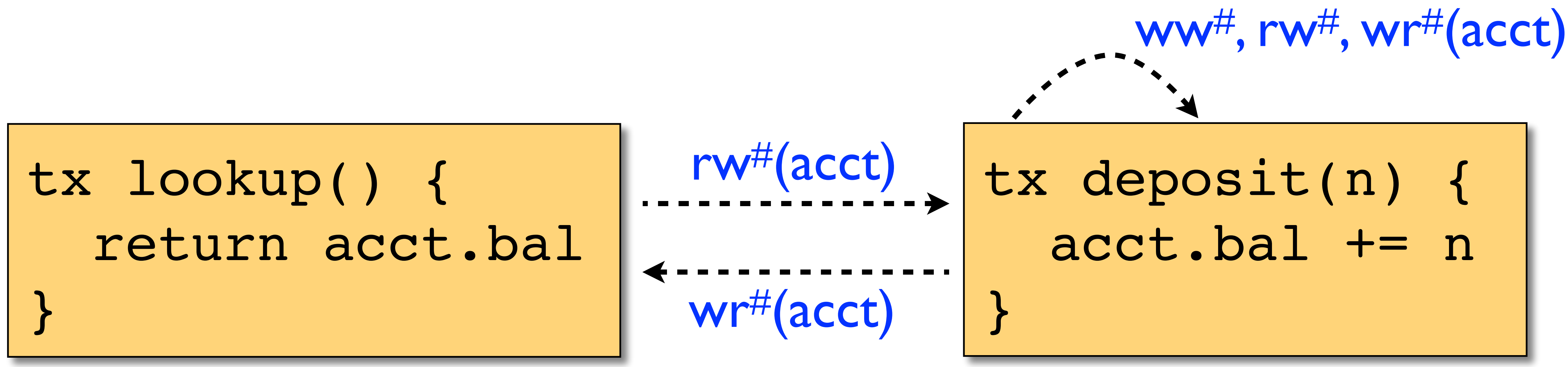
Not a valid PSI execution: violates write-conflict detection

The 2 rw edges are due to the same object

Static robustness criterion

If a dependency graph of a PSI execution contains a cycle, then it also contains one:

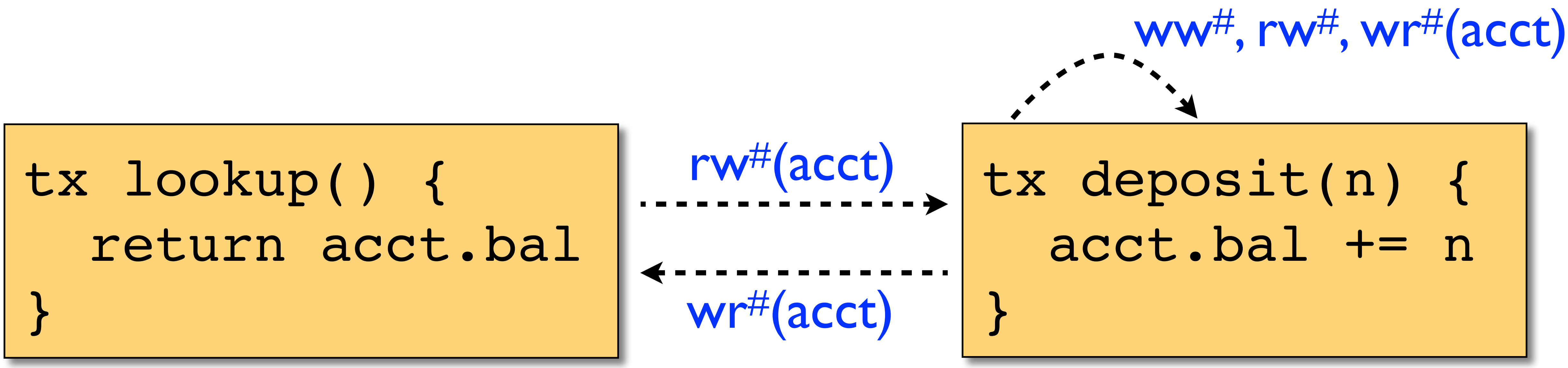
- with at least two rw edges, and
- where all rw edges are due to distinct objects



Static robustness criterion

If a dependency graph of a PSI execution contains a cycle, then it also contains one:

- with at least two rw edges, and
- where all rw edges are due to distinct objects



No cycles in wr# ∪ ww# ∪ rw# with all rw# on different objects

Static robustness criterion

> If a dependency graph of a PSI execution contains a cycle, then it also contains one:
>
> - with at least two rw edges, and
> - where all rw edges are due to distinct objects

$ww^{\#}, rw^{\#}, wr^{\#}(acct)$

```
tx lookup() {
  return acct.bal
}
```

$\xrightarrow{rw^{\#}(acct)}$

$\xleftarrow{wr^{\#}(acct)}$

```
tx deposit(n) {
  acct.bal += n
}
```

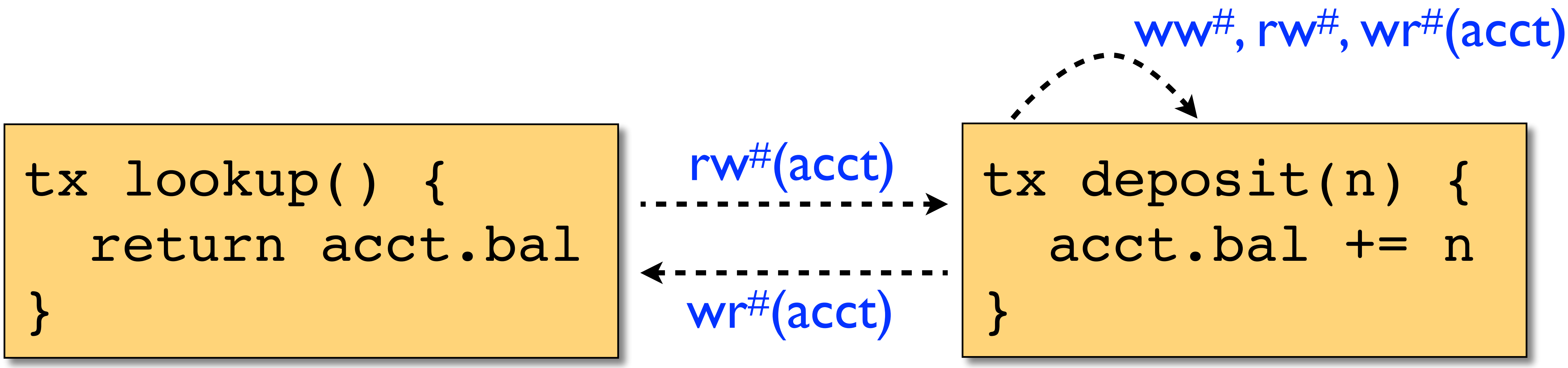
No cycles in $wr^{\#} \cup ww^{\#} \cup rw^{\#}$ with all $rw^{\#}$ on different objects

$\Longrightarrow$ no such cycles in $wr \cup ww \cup rw$

Static robustness criterion

If a dependency graph of a PSI execution contains a cycle, then it also contains one:

- with at least two rw edges, and
- where all rw edges are due to distinct objects



No cycles in $wr^\# \cup ww^\# \cup rw^\#$ with all $rw^\#$ on different objects

$\Longrightarrow$ no such cycles in $wr \cup ww \cup rw$

$\Longrightarrow$ application is serializable

Non-robustness

```
tx lookupAll() {
  return acct[*].bal
}
```

rw#(*) →

← wr#(*)

ww#, rw#, wr#(*)

```
tx deposit(i, n) {
  acct[i].bal += n
}
```

deposit(1, €100) $\xrightarrow{\text{vis}/\sim}$ lookupAll : 1/€100, 2/€0

deposit(2, €100) $\xrightarrow{\text{vis}/\sim}$ lookupAll : 1/€0, 2/€100

Automatic robustness checking

- Methods for other consistency models are similar
- Basis for practical tools [Warszawski et al., SIGMOD'17, Brutschy et al., PLDI'18; Nagar et al., CONCUR'18]
- Static criterion on graphs sometimes used to prune the search space before a more expensive analysis with more semantic information
- Can be used for bug-finding in the absence of specifications

Automatic robustness checking

ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications

Todd Warszawski, Peter Bailis
Stanford InfoLab

ABSTRACT

In theory, database transactions protect application data from corruption and integrity violations. In practice, database transactions frequently execute under weak isolation that exposes programs to a range of concurrency anomalies, and programmers may fail to correctly employ transactions. While low transaction volumes mask many potential concurrency-related errors under normal operation, determined adversaries can exploit them programmatically for fun and profit. In this paper, we formalize a new kind of attack on database-backed applications called an *ACIDRain attack*, in which an adversary systematically exploits concurrency-related vulnerabilities via programmatically accessible APIs. These attacks are not theoretical: ACIDRain attacks have already occurred in a handful of applications in the wild, including one attack which bankrupted a popular Bitcoin exchange. To proactively detect the potential for ACIDRain attacks, we extend the theory of weak isolation to analyze latent potential for non-serializable behavior under concurrent web API calls. We introduce a language-agnostic method for detecting potential isolation anomalies in web applications, called Abstract Anomaly Detection (2AD), that uses dynamic traces of database accesses to efficiently reason about the space of possible concurrent interleavings. We apply a prototype 2AD analysis tool to 12 popular self-hosted eCommerce applications written in four languages and

```
def withdraw(amt, user_id):                    (a)
  bal = readBalance(user_id)
  if (bal >= amt):
    writeBalance(bal - amt, user_id)
```

```
def withdraw(amt, user_id):                    (b)
  beginTxn()
  bal = readBalance(user_id)
  if (bal >= amt):
    writeBalance(bal - amt, user_id)
  commit()
```

Figure 1: (a) A simplified example of code that is vulnerable to an ACIDRain attack allowing overdraft under concurrent access. Two concurrent instances of the `withdraw` function could both read balance \$100, check that \$100 $\geq$ \$99, and each allow \$99 to be withdrawn, resulting \$198 total withdrawals. (b) Example of how transactions could be inserted to address this error. However, even this code is vulnerable to attack at isolation levels at or below Read Committed, unless explicit locking such as `SELECT FOR UPDATE` is used. While this scenario closely re-

Implementing strong consistency

Designing consistency protocols

- So far implementations have been lightweight: "*an operation can only be delivered after all its causal dependencies*"
- In reality, designing consistency protocols and proving them correct is very difficult!
- Even more so for strong consistency protocols

Strong consistency

c.withdraw(100) : ?

c.withdraw(100) : ?

Strong consistency

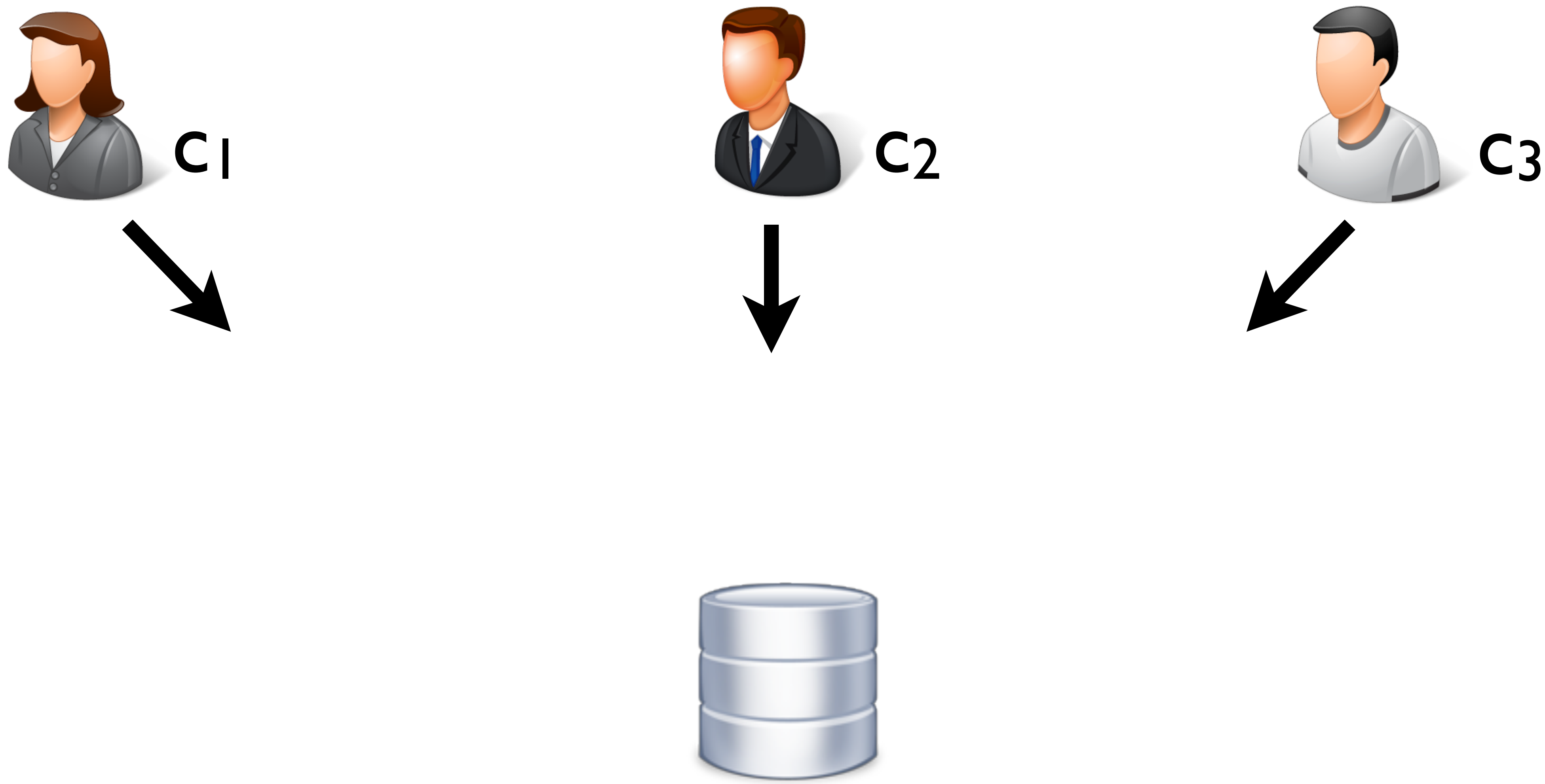


c.withdraw(100) : ✔

c.withdraw(100) : ?

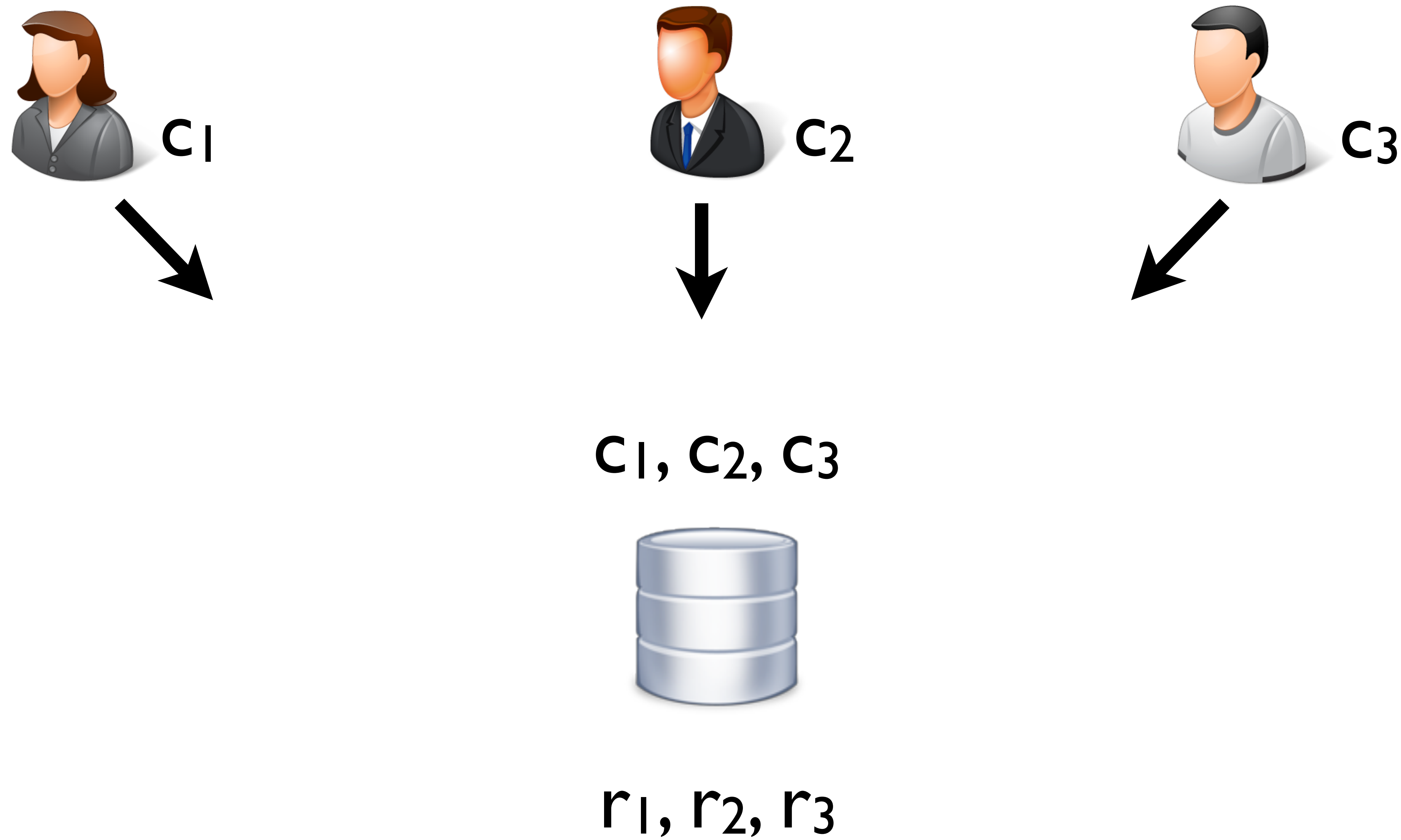
Sombody has to order commands

Strong consistency



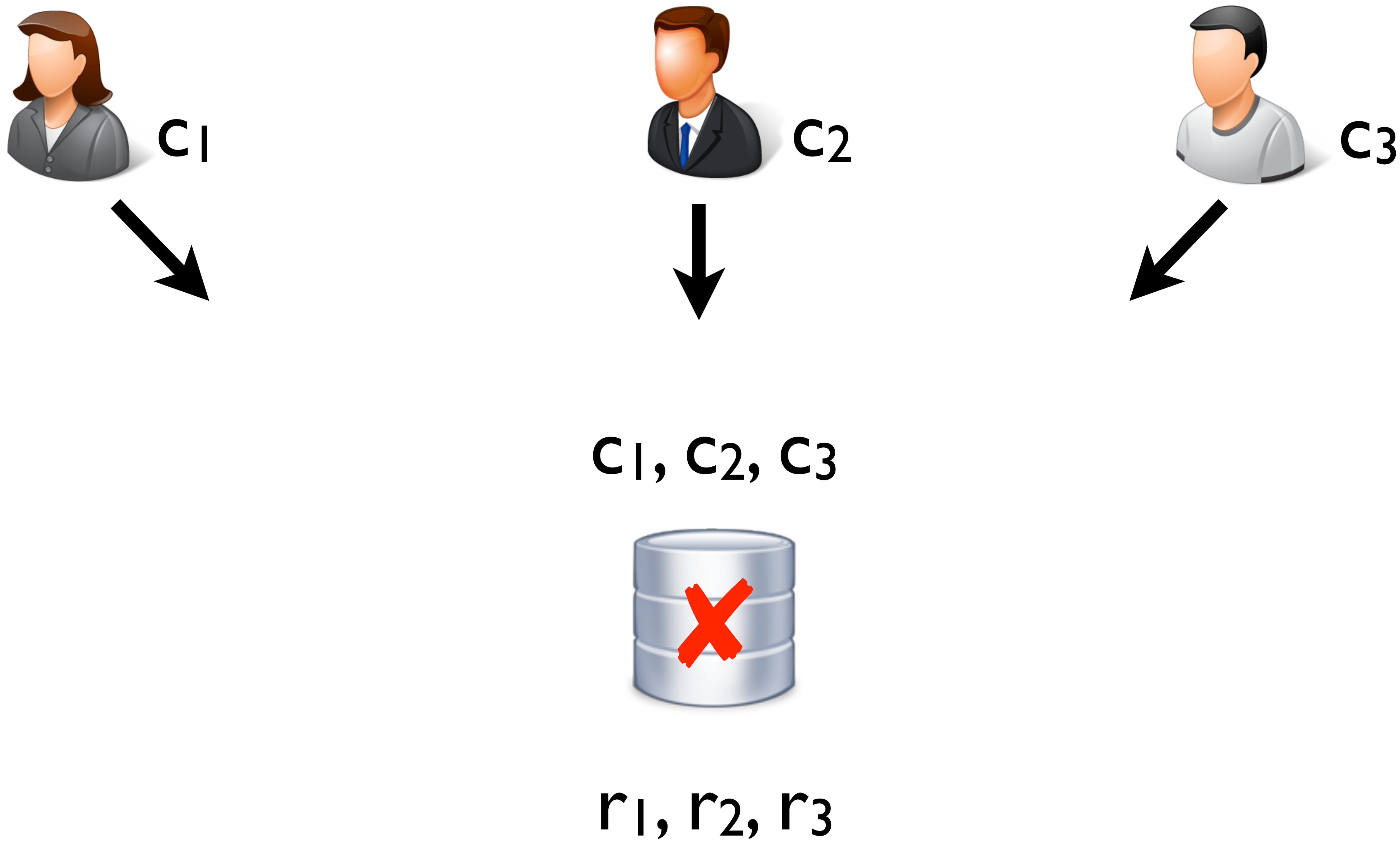
Single server, clients send commands to the server

Strong consistency



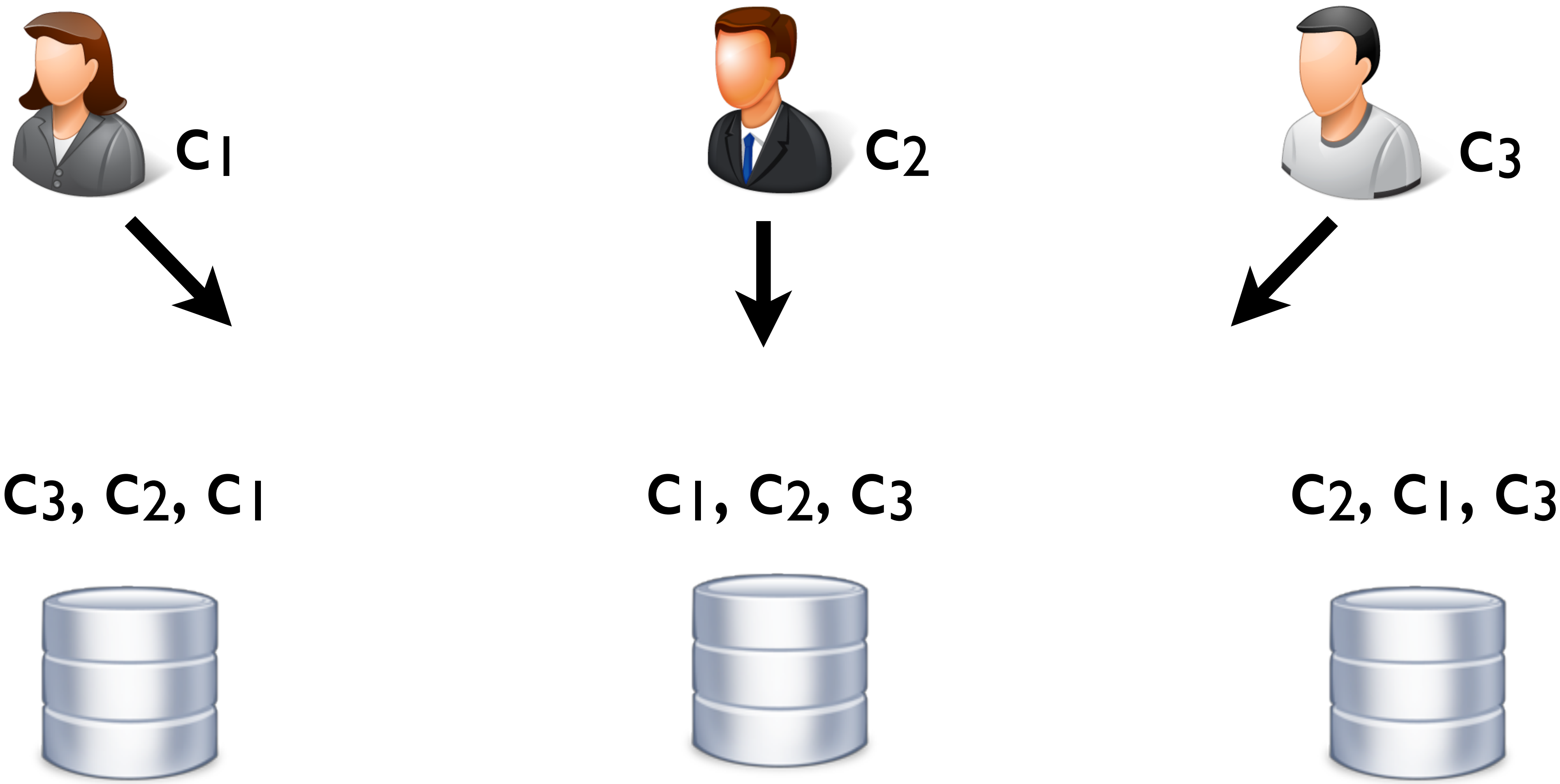
Server totally orders commands and computes the sequence of results

Strong consistency



Servers can crash! Need a fault-tolerant solution

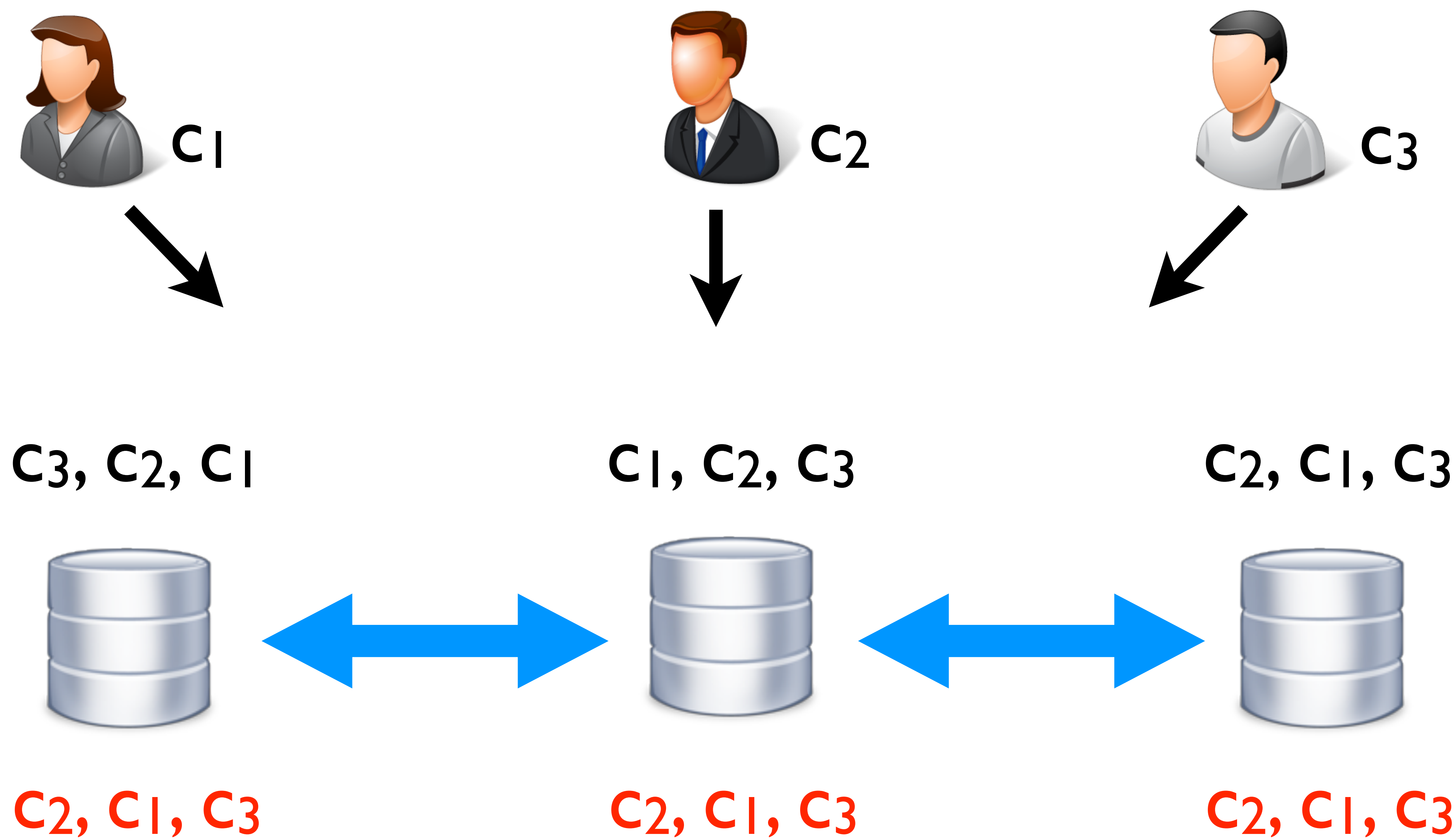
State machine replication



Clients send commands to all replicas

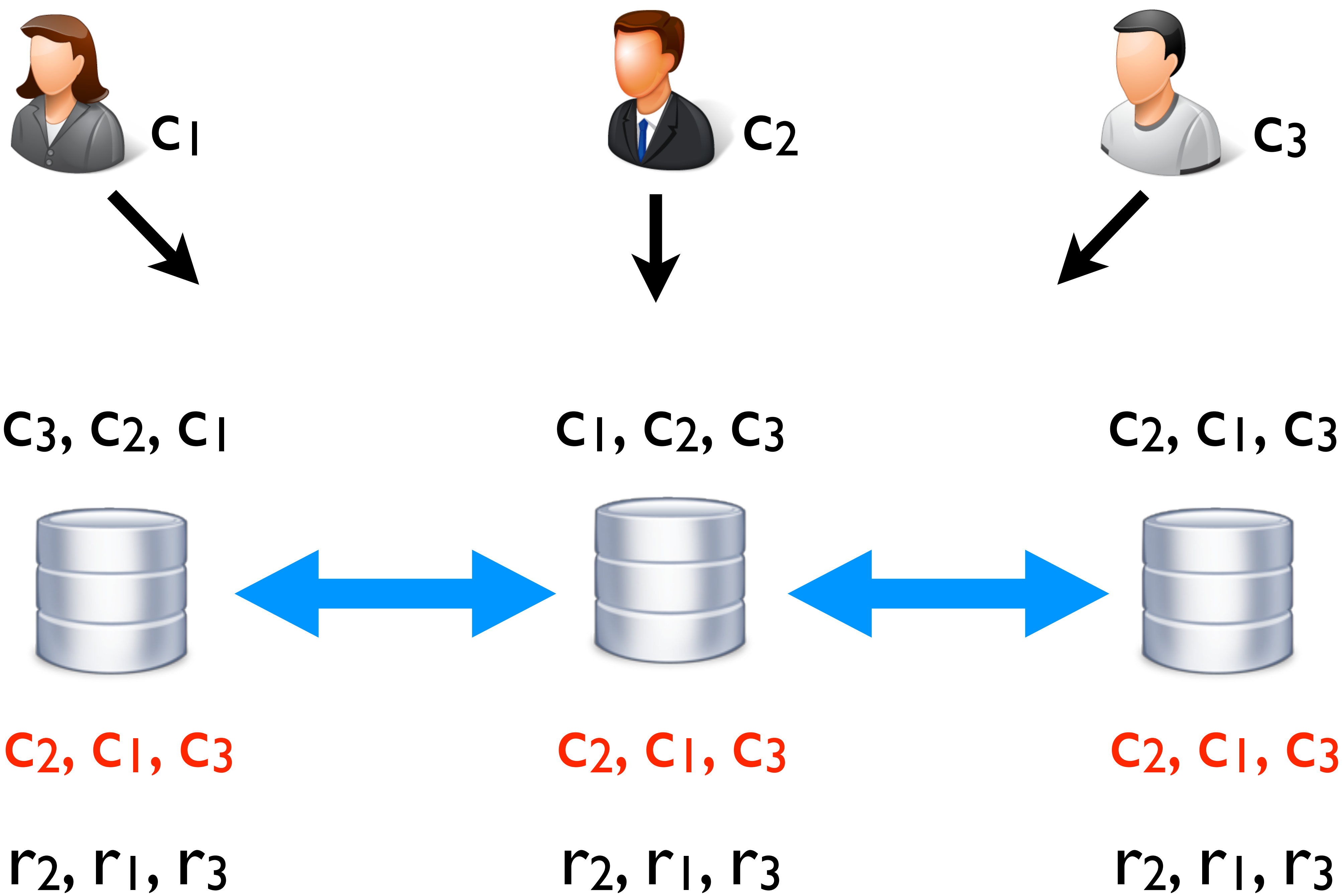
Replicas may receive commands in different orders

State machine replication



A distributed protocol totally order commands:
needs synchronisation

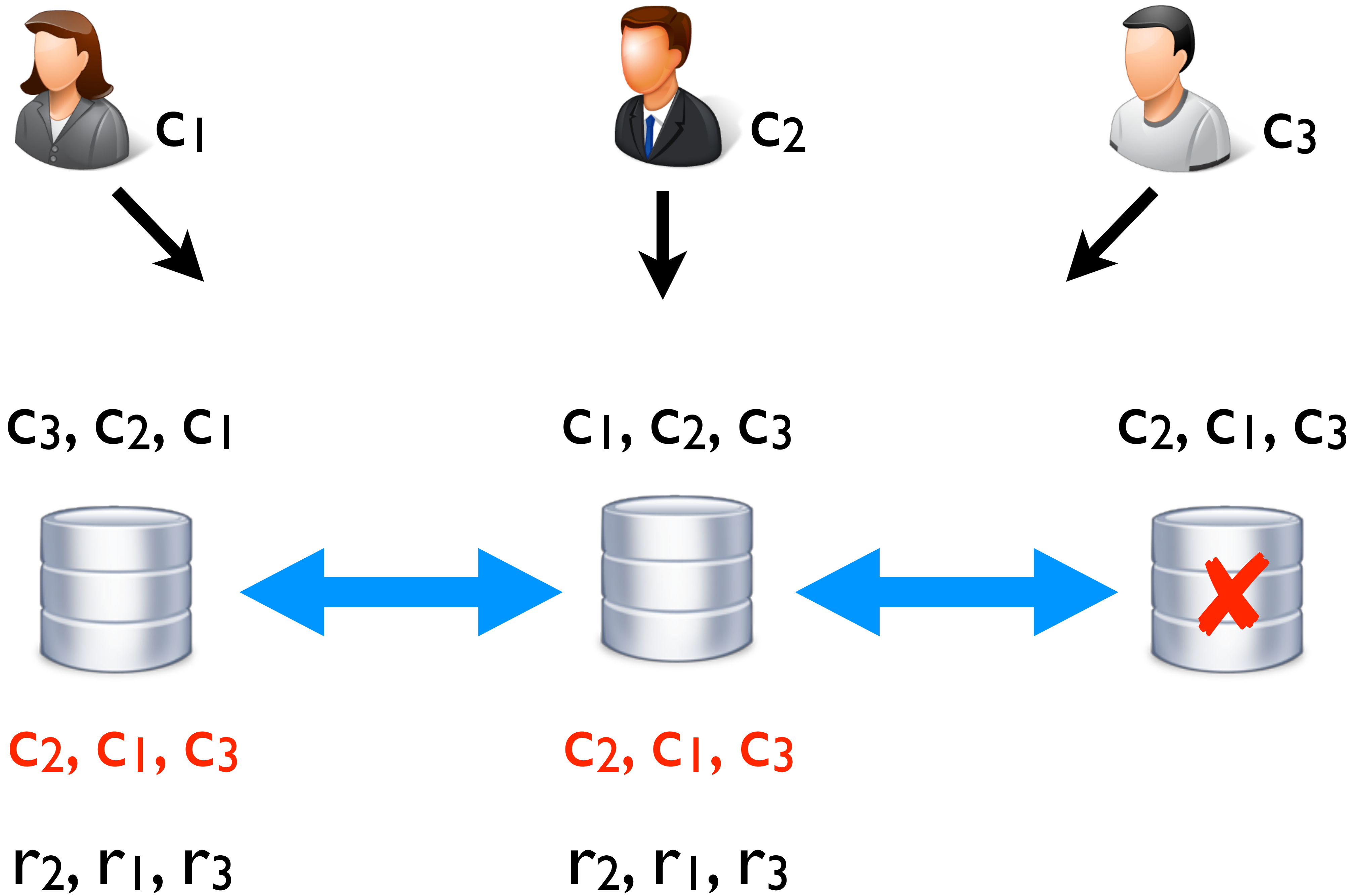
State machine replication



Operations are deterministic $\Longrightarrow$

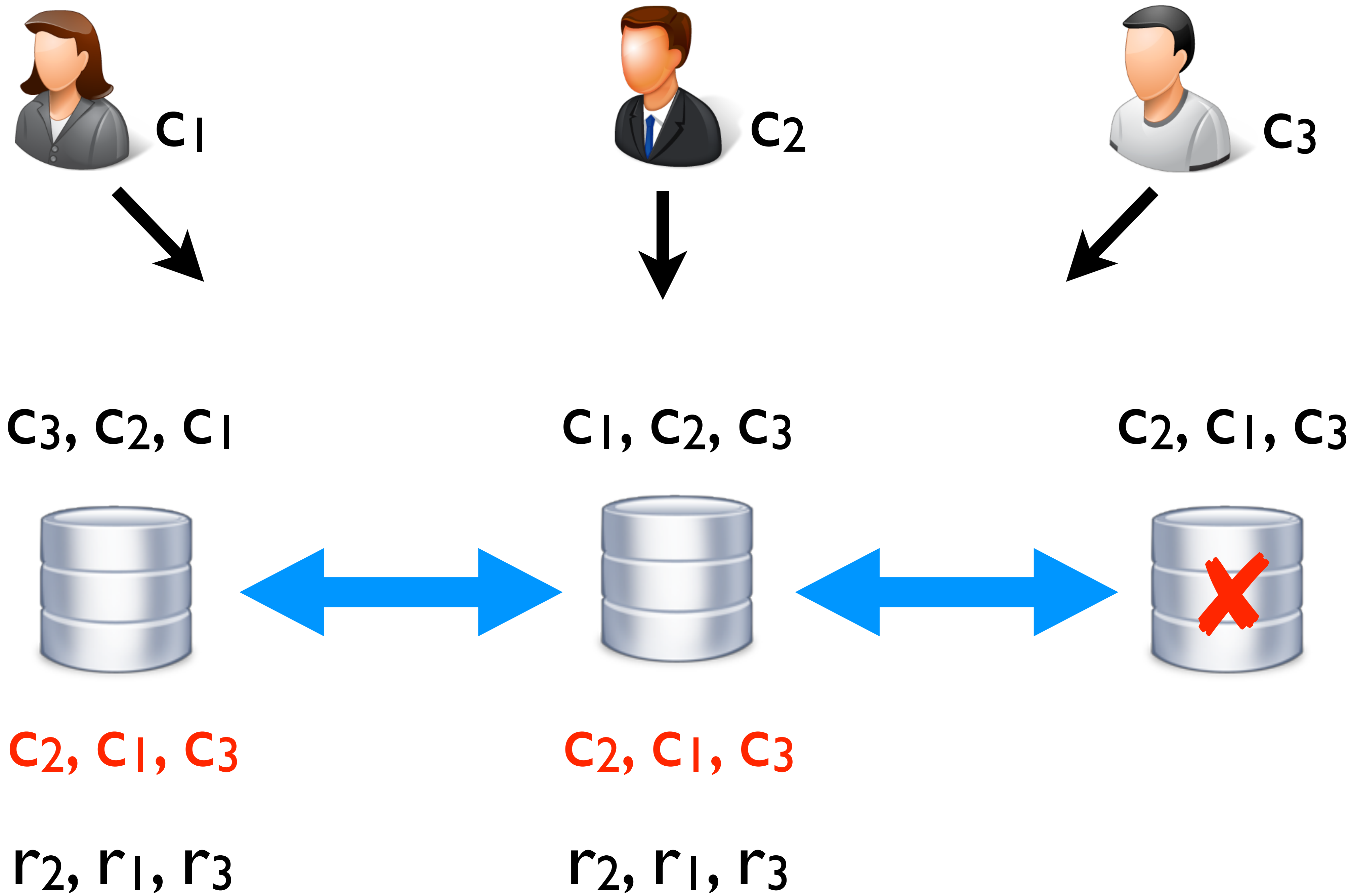
replicas compute the same sequence of results

State machine replication



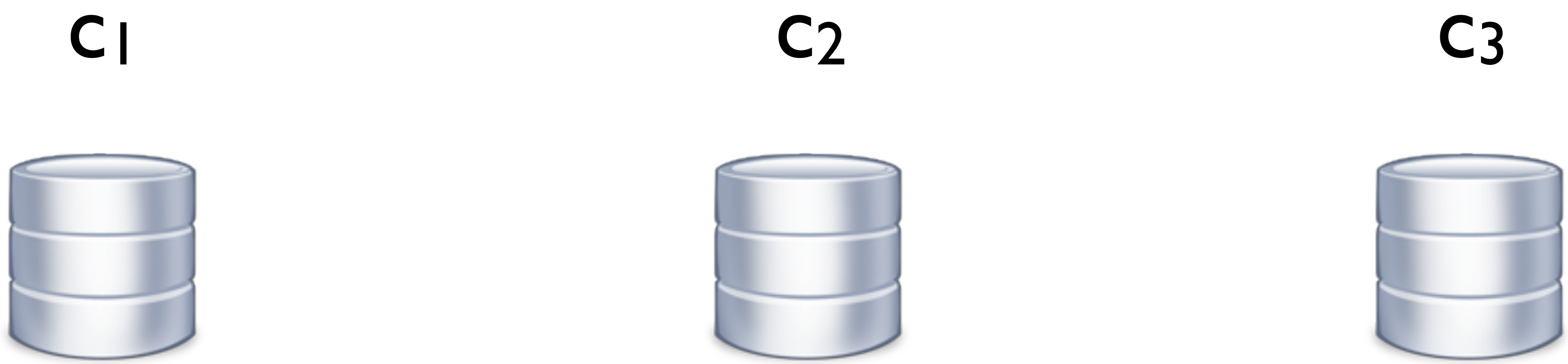
Implements sequential consistency (in fact, linearizability)

State machine replication



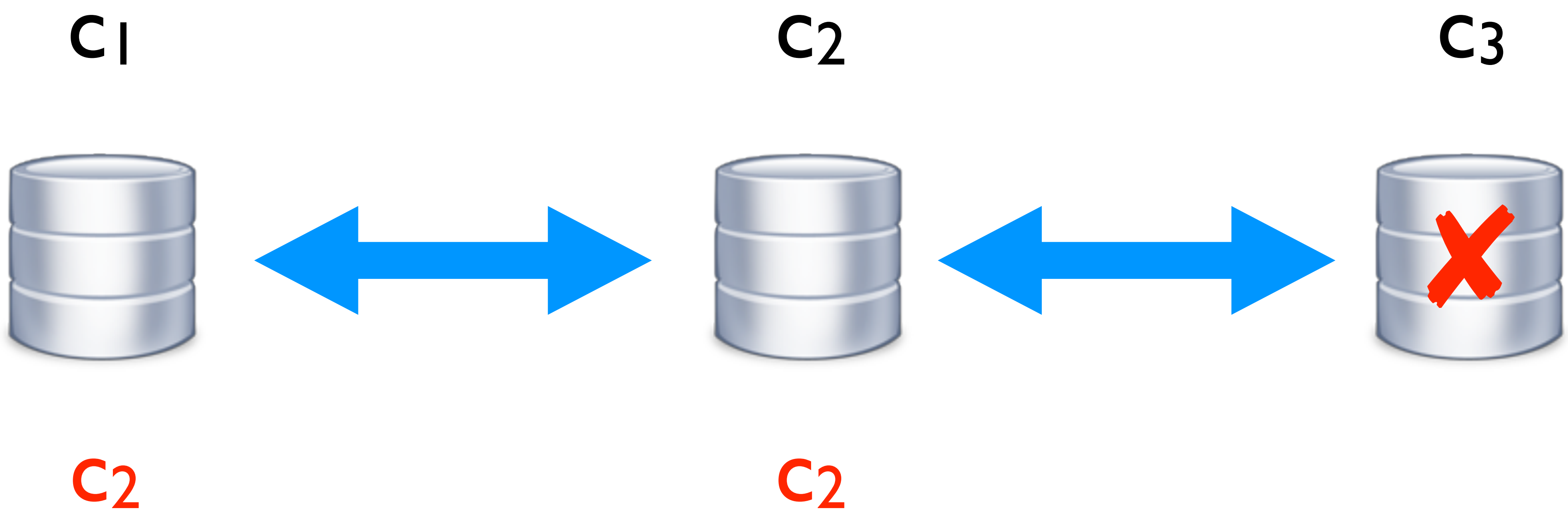
SMR requires solving a sequence of consensus instances:
agree on the next command to execute

Consensus



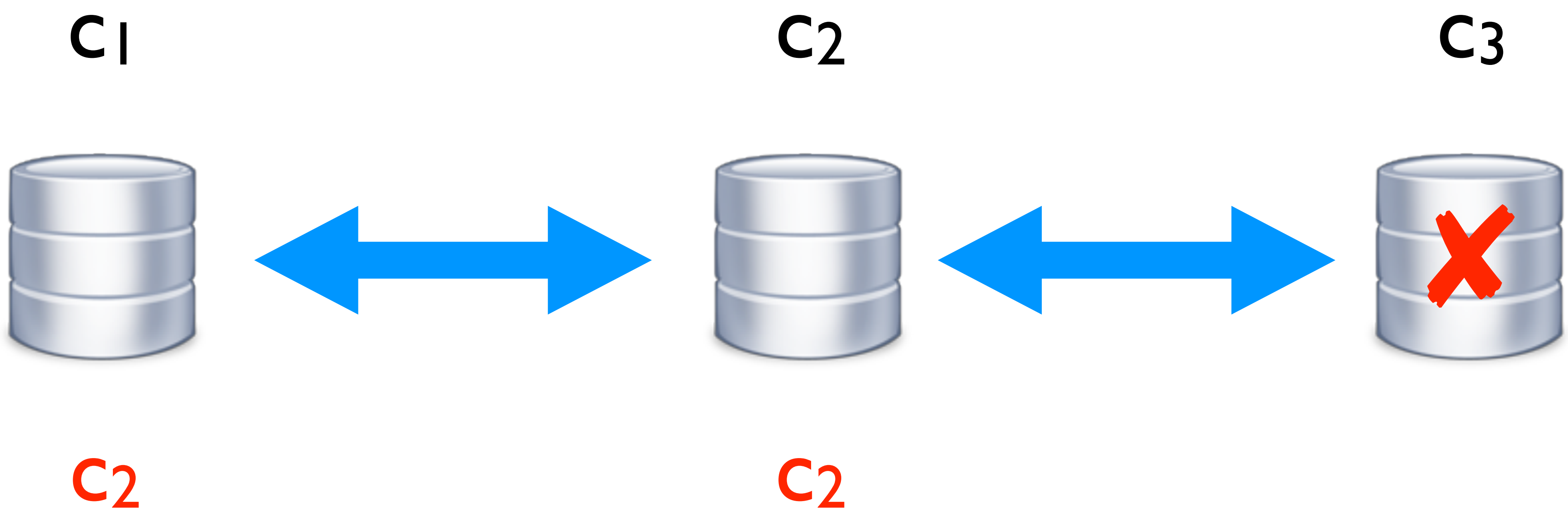
- Several nodes, which can crash
- Each proposes a value

Consensus



- Several nodes, which can crash
- Each proposes a value
- All non-crashed nodes agree on a single value

Consensus



- Challenge: asynchronous channels $\Longrightarrow$ can't tell a crashed node from a slow one!
- Assume only a minority of nodes can crash: a majority reach an agreement

The zoo of consensus protocols

- Viewstamped replication (1988)
- Paxos (1998)
- Disk Paxos (2003)
- Cheap Paxos (2004)
- Generalized Paxos (2004)
- Paxos Commit (2004)
- Fast Paxos (2006)
- Stoppable Paxos (2008)
- Mencius (2008)
- Vertical Paxos (2009)
- ZAB (2009)
- Ring Paxos (2010)
- Egalitarian Paxos (2013)
- Raft (2014)
- M2Paxos (2016)
- Flexible Paxos (2016)
- Caesar (2017)

The zoo of consensus protocols

- Viewstamped replication (1988)
- Paxos (1998)
- Disk Paxos (2003)
- Cheap Paxos (2004)
- Generalized Paxos (2004)
- Paxos Commit (2004)
- Fast Paxos (2006)
- Stoppable Paxos (2008)
- Mencius (2008)
- Vertical Paxos (2009)
- ZAB (2009)
- Ring Paxos (2010)
- Egalitarian Paxos (2013)
- Raft (2014)
- M2Paxos (2016)
- Flexible Paxos (2016)
- Caesar (2017)

The zoo of co

Complex protocols: constant fight for better performance

- Viewstamped replication (1988)
- Paxos (1998)
- Disk Paxos (2003)
- Cheap Paxos (2004)
- Generalized Paxos (2004)
- Paxos Commit (2004)
- Fast Paxos (2006)
- Stoppable Paxos (2008)
- Mencius (2008)
- Vertical Paxos (2009)
- ZAB (2009)
- Ring Paxos (2010)
- Egalitarian Paxos (2013)
- Raft (2014)
- M2Paxos (2016)
- Flexible Paxos (2016)
- Caesar (2017)

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state-machine approach to the design of distributed systems.

Categories and Subject Descriptors: C2.4 [**Computer-Communications Networks**]: Distributed Systems—*Network operating systems*; D4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; J.1 [**Administrative Data Processing**]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

This submission was recently discovered behind a filing cabinet in the *TOCS* editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate; even though the obscure ancient Paxon civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment. Indeed, some of the refinements the Paxons made to their protocol appear to be unknown in the systems literature.

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Paxos Made Simple

Leslie Lamport

Abstract

The Paxos algorithm, when presented in plain English, is very simple.

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Paxos Made Simple

Paxos Made Moderately Complex

ROBBERT VAN RENESSE and DENIZ ALTINBUKEN, Cornell University

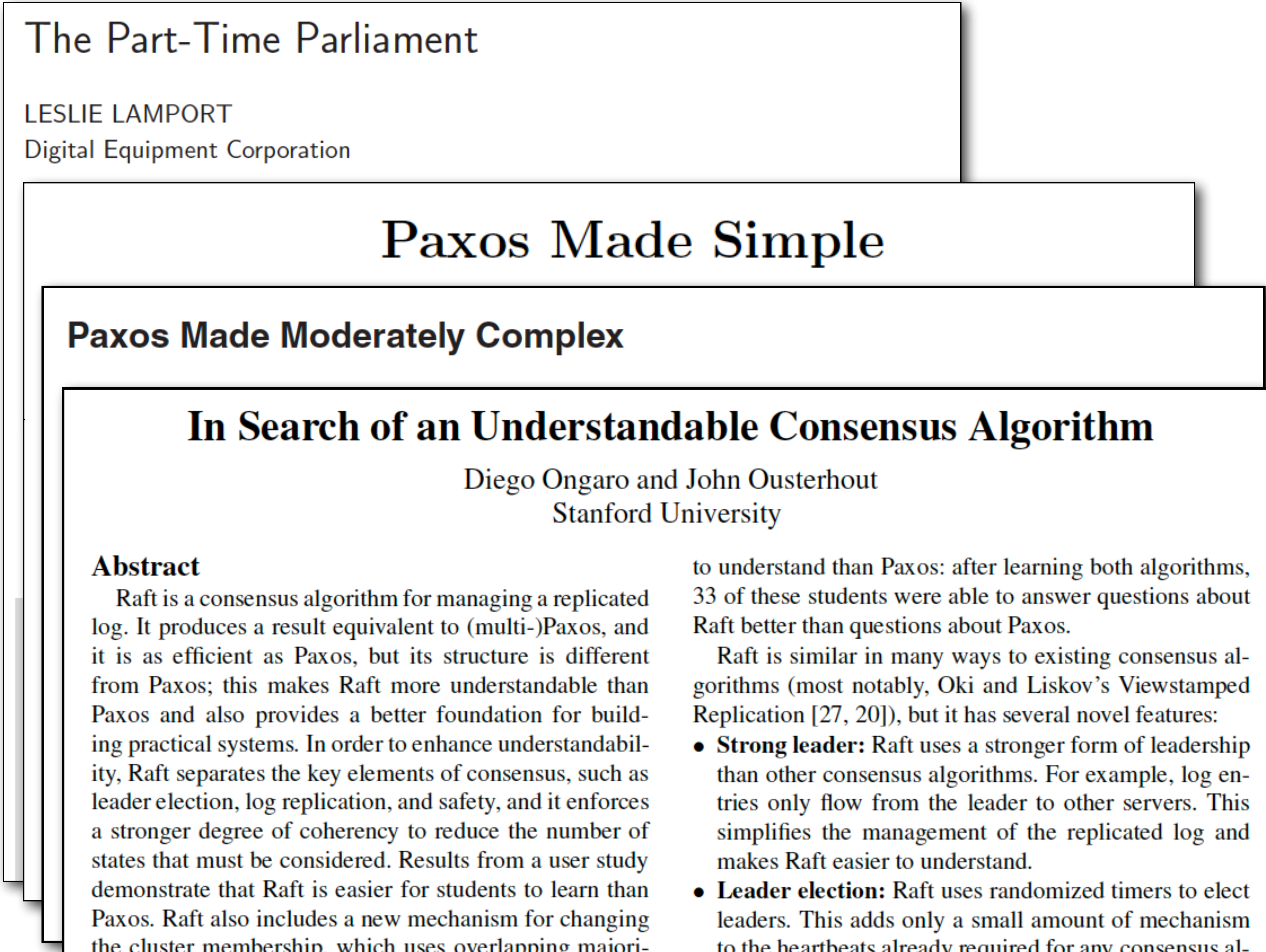
This article explains the full reconfigurable multidecree Paxos (or multi-Paxos) protocol. Paxos is by no means a simple protocol, even though it is based on relatively simple invariants. We provide pseudocode and explain it guided by invariants. We initially avoid optimizations that complicate comprehension. Next we discuss liveness, list various optimizations that make the protocol practical, and present variants of the protocol.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Network operating systems*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*

General Terms: Design, Reliability

Additional Key Words and Phrases: Replicated state machines, consensus, voting

ACM Reference Format:
Robbert van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. ACM Comput. Surv. 47, 3, Article 42 (February 2015), 36 pages.
DOI: http://dx.doi.org/10.1145/2673577



The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Paxos Made Simple

Paxos Made Moderately Complex

In Search of an Understandable Consensus Algorithm

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing

to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [27, 20]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Paxos Made Simple

Paxos Made Moderately Complex

In Search of an Understandable Consensus Algorithm

Abstract

Raft is a consensus algorithm for m
log. It produces a result equivalent t
it is as efficient as Paxos, but its s
from Paxos; this makes Raft more
Paxos and also provides a better f
ing practical systems. In order to enh
ity, Raft separates the key elements c
leader election, log replication, and s
a stronger degree of coherency to re
states that must be considered. Resu
demonstrate that Raft is easier for st
Paxos. Raft also includes a new mec
the cluster membership, which uses

Unfortunately, Paxos has two significant drawbacks. The first drawback is that Paxos is exceptionally difficult to understand. The full explanation [15] is notoriously opaque; few people succeed in understanding it, and only with great effort. As a result, there have been several attempts to explain Paxos in simpler terms [16, 20, 21]. These explanations focus on the single-decree subset, yet they are still challenging. In an informal survey of attendees at NSDI 2012, we found few people who were comfortable with Paxos, even among seasoned researchers. We struggled with Paxos ourselves; we were not able to understand the complete protocol until after reading several simplified explanations and designing our own alter-

Paxos Made Live - An Engineering Perspective (2006 Invited Talk)

Tushar Chandra, Robert Griesemer, and Joshua Redstone

Google Inc.

ABSTRACT

We describe our experience in building a fault-tolerant database using the Paxos consensus algorithm. Despite the existing literature in the field, building such a database proved to be non-trivial. We describe selected algorithmic and engineering problems encountered, and the solutions we found for them. Our measurements indicate that we have built a competitive system.

Categories and Subject Descriptors

D.4.5 [**Operating systems**]: Reliability—*Fault-tolerance*; B.4.5 [**Input/output and data communications**]: Reliability, Testing, and Fault-Tolerance—*Redundant design*

General Terms

Experimentation, Performance, Reliability

Keywords

Experiences, Fault-tolerance, Implementation, Paxos

database is just an example. As a result, the consensus problem has been studied extensively over the past two decades. There are several well-known consensus algorithms that operate within a multitude of settings and which tolerate a variety of failures. The Paxos consensus algorithm [8] has been discussed in the theoretical [16] and applied community [10, 11, 12] for over a decade.

We used the Paxos algorithm ("Paxos") as the base for a framework that implements a fault-tolerant log. We then relied on that framework to build a fault-tolerant database. Despite the existing literature on the subject, building a production system turned out to be a non-trivial task for a variety of reasons:

- While Paxos can be described with a page of pseudocode, our complete implementation contains several thousand lines of C++ code. The blow-up is not due simply to the fact that we used C++ instead of pseudo notation, nor because our code style may have been verbose. Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations – some published in the lit-

Paxos Made Live - An Engineering Perspective (2006 Invited Talk)

Tushar Chandra, Robert Griesemer, and Joshua Redstone

Google Inc.

ABSTRACT

We describe our experience in building a fault-
base using the Paxos consensus algorithm. D
isting literature in the field, building such a da
to be non-trivial. We describe selected algorit
gineering problems encountered, and the solut
for them. Our measurements indicate that we
competitive system.

Categories and Subject Descriptors

D.4.5 [**Operating systems**]: Reliability—*Fault-tolerance*; B.4.5 [**Input/output and data communications**]: Reliability, Testing, and Fault-Tolerance—*Redundant design*

General Terms

Experimentation, Performance, Reliability

Keywords

Experiences, Fault-tolerance, Implementation, Paxos

- There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol.

a framework that implements a fault-tolerant log. We then relied on that framework to build a fault-tolerant database. Despite the existing literature on the subject, building a production system turned out to be a non-trivial task for a variety of reasons:

- While Paxos can be described with a page of pseudo-code, our complete implementation contains several thousand lines of C++ code. The blow-up is not due simply to the fact that we used C++ instead of pseudo notation, nor because our code style may have been verbose. Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations – some published in the lit-

Paxos Made Live -
(200

Tushar Chandra, Rob

5.1 Handling disk corruption

Replicas witness disk corruption from time to time. A disk may be corrupted due to a media failure or due to an operator error (an operator may accidentally erase critical data). When a replica's disk is corrupted and it loses its persistent state, it may renege on promises it has made to other replicas in the past. This violates a key assumption in the Paxos algorithm. We use the following mechanism to address this problem [14].

Disk corruptions manifest themselves in two ways. Either file(s) contents may change or file(s) may become inaccessible. To detect the former, we store the checksum of the contents of each file in the file[2]. The latter may be indistinguishable from a new replica with an empty disk – we detect this case by having a new replica leave a marker in GFS after start-up. If this replica ever starts again with an empty disk, it will discover the GFS marker and indicate that it has a corrupted disk.

A replica with a corrupted disk rebuilds its state as follows. It participates in Paxos as a non-voting member; meaning that it uses the catch-up mechanism to catch up but does not respond with promise or acknowledgment messages. It remains in this state until it observes one complete instance of Paxos that was started after the replica started rebuilding its state. By waiting for the extra instance of Paxos, we ensure that this replica could not have reneged on an earlier promise.

ABSTRACT

We describe our experience in building a fault-tolera
base using the Paxos consensus algorithm. Despite
isting literature in the field, building such a databas
to be non-trivial. We describe selected algorithmic
gineering problems encountered, and the solutions v
for them. Our measurements indicate that we hav
competitive system.

Categories and Subject Descriptors

D.4.5 [**Operating systems**]: Reliability—*Fault-to*
B.4.5 [**Input/output and data communication**
ability, Testing, and Fault-Tolerance—*Redundant d*

General Terms

Experimentation, Performance, Reliability

Keywords

Experiences, Fault-tolerance, Implementation, Paxos

production-ready system involved implementing many features and optimizations – some published in the lit-

Paxos Made Live - (200

Tushar Chandra, Rob

ABSTRACT

We describe our experience in building a fault-tolera base using the Paxos consensus algorithm. Despite isting literature in the field, building such a databas to be non-trivial. We describe selected algorithmic gineering problems encountered, and the solutions v for them. Our measurements indicate that we hav competitive system.

Categories and Subject Descriptors

D.4.5 [**Operating systems**]: Reliability—*Fault-to* B.4.5 [**Input/output and data communication** ability, Testing, and Fault-Tolerance—*Redundant d*

General Terms

Experimentation, Performance, Reliability

Keywords

Experiences, Fault-tolerance, Implementation, Paxos

5.1 Handling disk corruption

Replicas witness disk corruption from time to time. A disk may be corrupted due to a media failure or due to an operator error (an operator may accidentally erase critical data). When a replica's disk is corrupted and it loses its persistent state, it may renege on promises it has made to other replicas in the past. This violates a key assumption in the Paxos algorithm. We use the following mechanism to address this problem [14].

Disk corruptions manifest themselves in two ways. Either file(s) contents may change or file(s) may become inaccessible. To detect the former, we store the checksum of the contents of each file in the file[2]. The latter may be indistinguishable from a new replica with an empty disk – we detect this case by having a new replica leave a marker in GFS a

empty

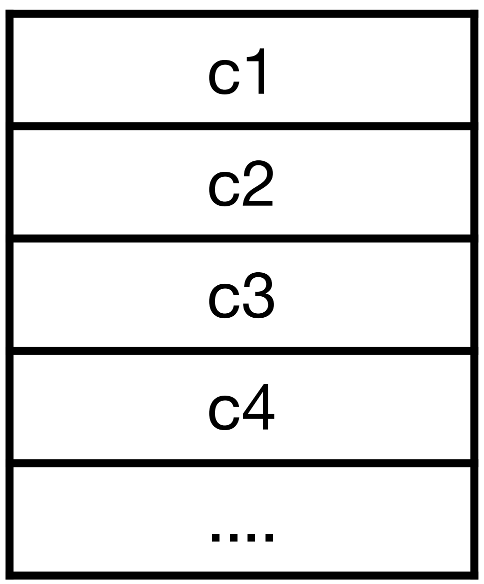
that it

Broken [Michael et al., DISC'16]

A replica with a corrupted disk rebuilds its state as follows. It participates in Paxos as a non-voting member; meaning that it uses the catch-up mechanism to catch up but does not respond with promise or acknowledgment messages. It remains in this state until it observes one complete instance of Paxos that was started after the replica started rebuilding its state. By waiting for the extra instance of Paxos, we ensure that this replica could not have reneged on an earlier promise.

production-ready system involved implementing many features and optimizations – some published in the lit-

Another application: blockchain



- Blockchain = using consensus to agree on a sequence of blocks in a ledger
- Tolerates malicious behaviour: some nodes may deviate from the protocol
- Many protocols descended from Paxos

Facebook's Libra pitches to be the future of money

Rory Cellan-Jones
Technology correspondent
@BBCRoryCJ

18 June 2019

It is a hugely ambitious - some might say megalomaniacal - project to create a new global currency. Facebook's David Marcus tells me it is about giving billions of people more freedom with money and "righting the many wrongs of the present system".

The message is this is not some little side project a small team at the Facebook's

BBC | Your account | News | Sport | Weather | Shop | Reel | Travel | More

NEWS

Home | Video | World | UK | Business | Tech | Science | Stories | Entertainment & Arts

Technology

Facebook's Libra pitches to be the future of mone

Rory C
Techno
@BBC

18 June 2019

It is a hugely am
a new global cu
billions of peop
of the present s

The message is

[PODC'19]

HotStuff: BFT Consensus with Linearity and Responsiveness

Maofan Yin
Cornell University
VMware Research

Dahlia Malkhi
VMware Research

Michael K. Reiter
UNC-Chapel Hill
VMware Research

Guy Golan Gueta
VMware Research

Ittai Abraham
VMware Research

ABSTRACT

We present HotStuff, a leader-based Byzantine fault-tolerant replication protocol for the partially synchronous model. Once network communication becomes synchronous, HotStuff enables a correct leader to drive the protocol to consensus at the pace of actual (vs. maximum) network delay—a property called *responsiveness*—and with communication complexity that is linear in the number of replicas. To our knowledge, HotStuff is the first partially synchronous BFT replication protocol exhibiting these combined properties. Its simplicity enables it to be further pipelined and simplified into a practical, concise protocol for building large-scale replication services.

CCS CONCEPTS

• **Software and its engineering** → **Software fault tolerance**; • **Security and privacy** → **Distributed systems security**.

KEYWORDS

Byzantine fault tolerance; consensus; responsiveness; scalability; blockchain

stabilization time (GST). In this model, $n \geq 3f + 1$ is required for non-faulty replicas to agree on the same commands in the same order (e.g., [12]) and progress can be ensured deterministically only after GST [27].

When BFT SMR protocols were originally conceived, a typical target system size was $n = 4$ or $n = 7$, deployed on a local-area network. However, the renewed interest in Byzantine fault-tolerance brought about by its application to blockchains now demands solutions that can scale to much larger n. In contrast to *permissionless* blockchains such as the one that supports Bitcoin, for example, so-called *permissioned* blockchains involve a fixed set of replicas that collectively maintain an ordered ledger of commands or, in other words, that support SMR. Despite their permissioned nature, numbers of replicas in the hundreds or even thousands are envisioned (e.g., [30, 42]). Additionally, their deployment to wide-area networks requires setting Δ to accommodate higher variability in communication delays.

The scaling challenge. Since the introduction of PBFT [20], the first practical BFT replication solution in the partial synchrony model, numerous BFT solutions were built around its core two-

BBC | Your account | News | Sport | Weather | Shop | Reel | Travel | Mor

NEWS

Home | Video | World | UK | Business | Tech | Science | Stories | Entertainment & Arts

Technology

Facebook's Libra pitches to be the future of mone

Rory C
Techno
@BBC

18 June 2019

It is a hugely am
a new global cu
billions of peop
of the present s

The message is

[PODC'19]

HotStuff: BFT Consensus with Linearity and Responsiveness

Maofan Yin, Cornell University, VMware Research

Dahlia Malkhi, VMware Research

Michael K. Reiter, UNC-Chapel Hill, VMware Research

Guy Golan Gueta, VMware Research

Ittai Abraham, VMware Research

ABSTRACT

We present HotStuff, a leader-based Byzantine fault-tolerant replication protocol for the partially synchronous model. Once network communication becomes synchronous, HotStuff enables a correct leader to drive the protocol to consensus at the pace of actual (vs. maximum) network delay—a property called *responsiveness*—and with communication complexity that is linear in the number of replicas. To our knowledge, HotStuff is the first partially synchronous BFT replication protocol exhibiting
Its simplicity enables it to be further pi
a practical, concise protocol for buildi
services.

CCS CONCEPTS

• **Software and its engineering** → **Sc**
Security and privacy → **Distributed**

KEYWORDS

Byzantine fault tolerance; consensus; responsiveness; scalability; blockchain

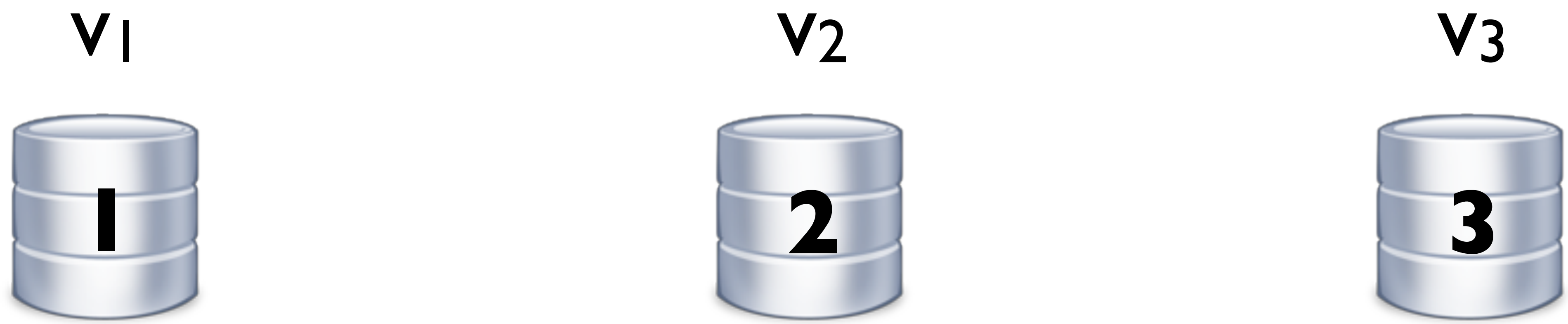
stabilization time (GST). In this model, $n \geq 3f + 1$ is required for non-faulty replicas to agree on the same commands in the same order (e.g., [12]) and progress can be ensured deterministically only after GST [27].

When BFT SMR protocols were originally conceived, a typical target system size was $n = 4$ or $n = 7$, deployed on a local-area network. However, the renewed interest in Byzantine fault-tolerance brought about by its application to blockchains now demands solutions that can scale to much larger n. In contrast to *permissionless*

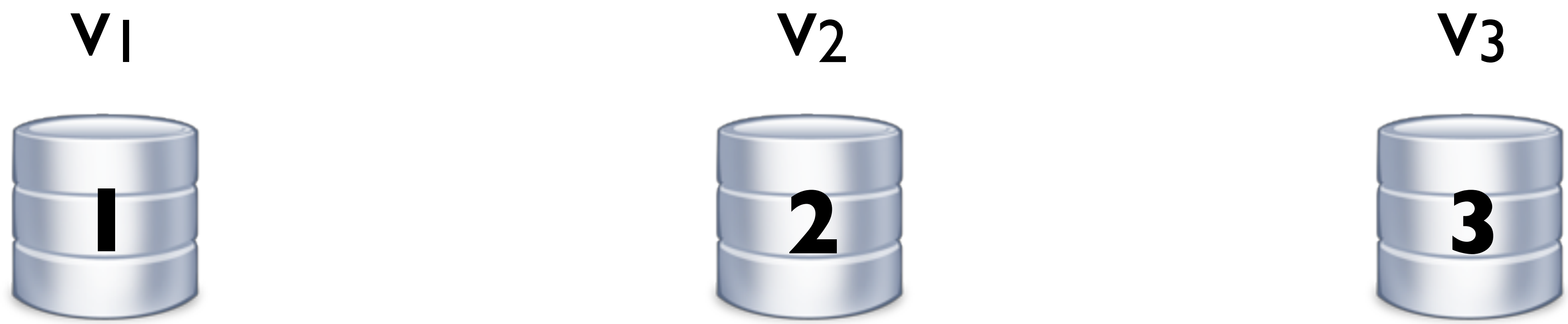
first practical BFT replication solution in the partial synchrony model, numerous BFT solutions were built around its core two-

ACKNOWLEDGMENTS

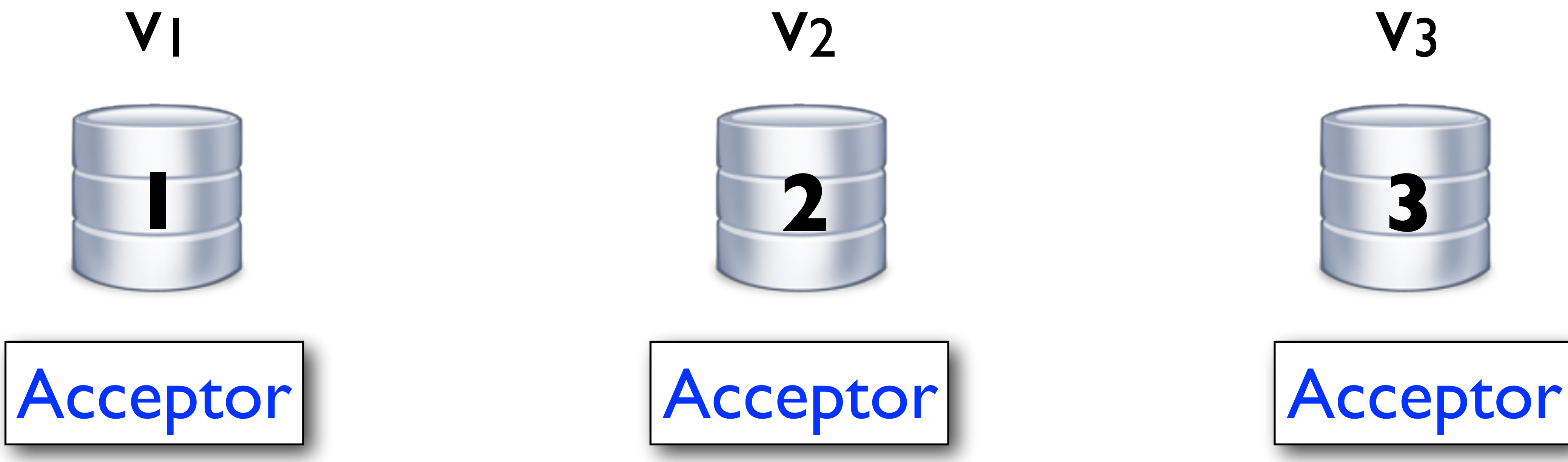
We are thankful to Mathieu Baudet, Avery Ching, George Danezis, François Garillot, Zekun Li, Ben Maurer, Kartik Nayak, Dmitri Perelman, and Ling Ren, for many deep discussions of HotStuff, and to Mathieu Baudet for exposing a subtle error in a previous version posted to the ArXiv of this manuscript.



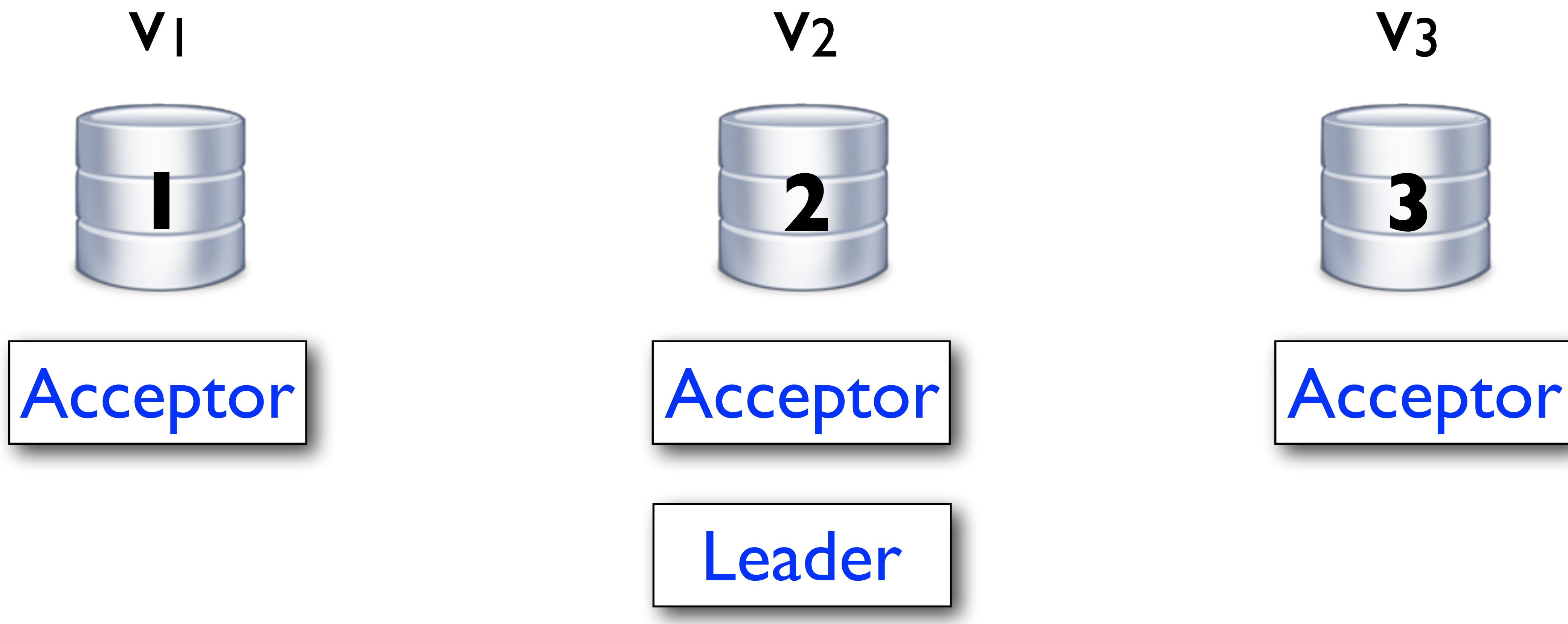
- 2f+1 nodes, at most f can crash
- Each node proposes a value
- All non-crashed nodes agree on a single value



v_1
v_2
v_3
1
2
3

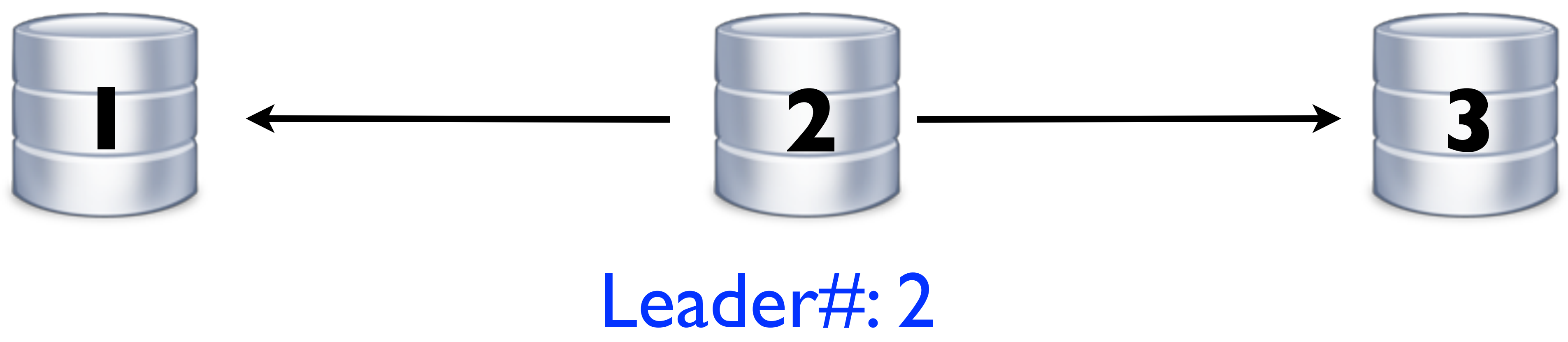


- Acceptors = members of parliament:
  can vote to accept a value, majority (quorum) wins

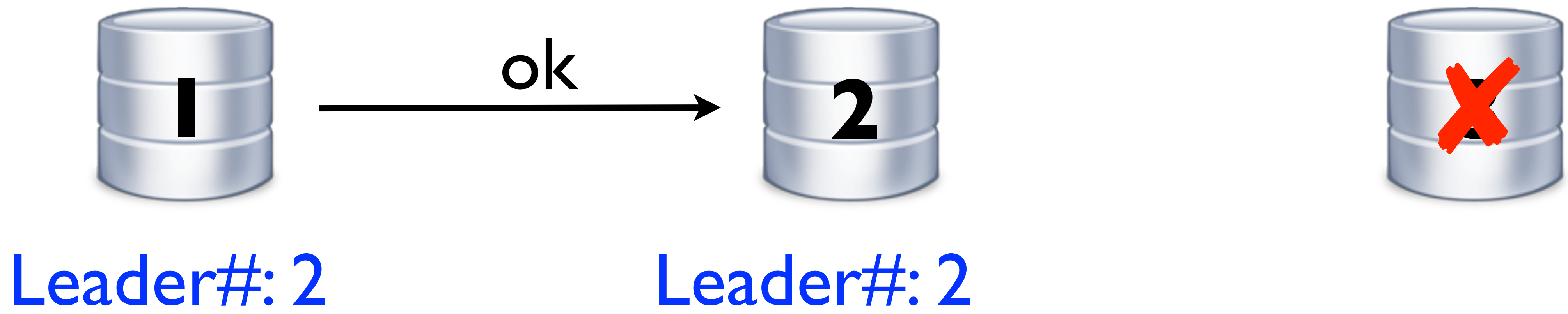


- Acceptors = members of parliament: can vote to accept a value, majority (quorum) wins
- Leader = parliament speaker: proposes its value to vote on
- Good for state-machine replication: can elect the leader once and get it to process multiple commands

- Phase 1: a prospective leader convinces a quorum of acceptors to accept its authority



- Phase 1: a prospective leader convinces a quorum of acceptors to accept its authority

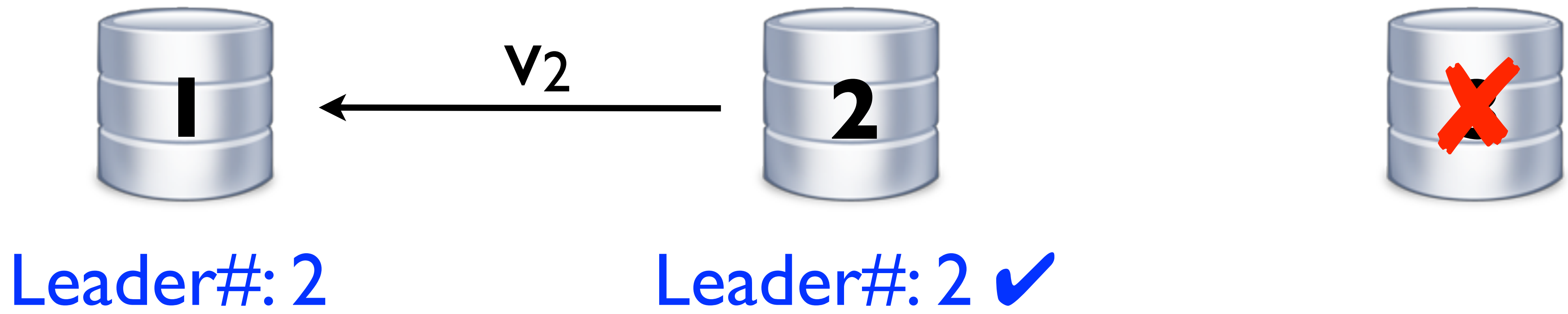


- Phase 1: a prospective leader convinces a quorum of acceptors to accept its authority

- Phase 1: a prospective leader convinces a quorum of acceptors to accept its authority

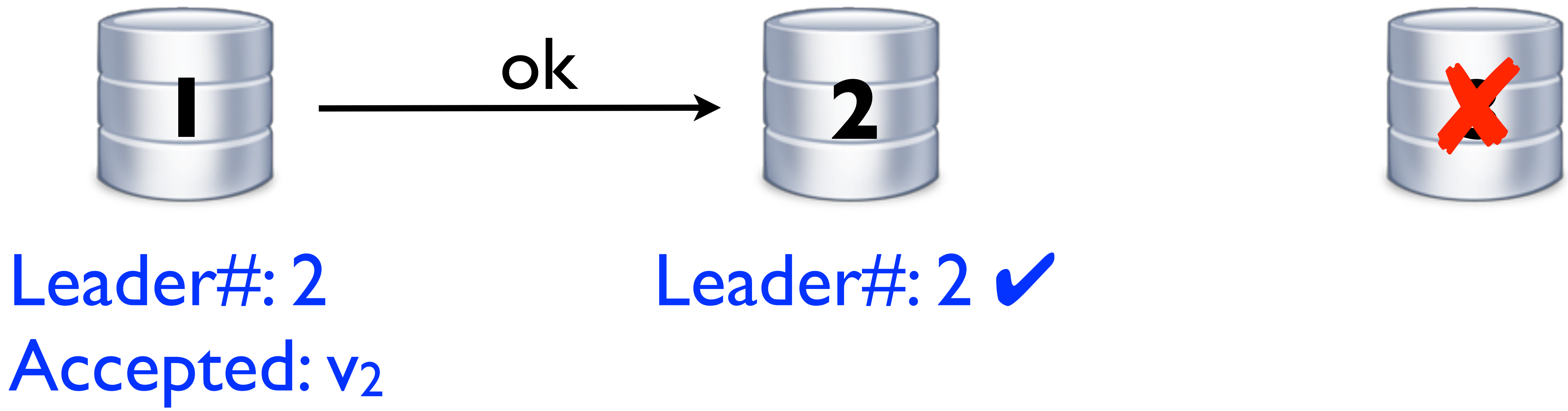
- Phase 1: a prospective leader convinces a quorum of acceptors to accept its authority

- Phase 2: the leader gets a quorum of acceptors to accept its value and replies to the client



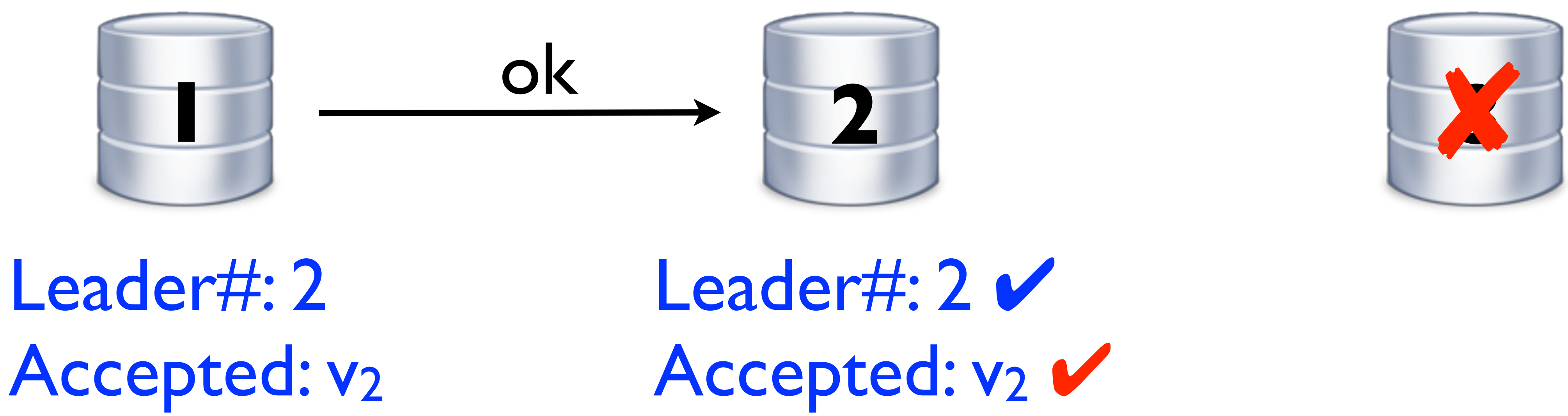
- Phase 1: a prospective leader convinces a quorum of acceptors to accept its authority

- Phase 2: the leader gets a quorum of acceptors to accept its value and replies to the client



- Phase 1: a prospective leader convinces a quorum of acceptors to accept its authority

- Phase 2: the leader gets a quorum of acceptors to accept its value and replies to the client



- Phase 1: a prospective leader convinces a quorum of acceptors to accept its authority

- Phase 2: the leader gets a quorum of acceptors to accept its value and replies to the client

Leader#: 2
Accepted: v_2

Leader#: 2 ✔
Accepted: v_2 ✔
Reply v_2 to client

- Phase 1: a prospective leader convinces a quorum of acceptors to accept its authority

- Phase 2: the leader gets a quorum of acceptors to accept its value and replies to the client

Leader#: 2
Accepted: v_2

Leader#: 2 ✔
Accepted: v_2 ✔
Reply v_2 to client

- Phase 1: a prospective leader convinces a quorum of acceptors to accept its authority

- Phase 2: the leader gets a quorum of acceptors to accept its value and replies to the client

Leader#: 3
Accepted: v_3

Leader#: 2 ✔
Accepted: v_2 ✔
Reply v_2 to client

Leader#: 3 ✔
Accepted: v_3 ✔
Reply v_3 to client

- Problem: node 3 may wake up, form a quorum of 1 and 3, and accept value v_3

Leader#: 3
Accepted: v_3

Leader#: 2 ✔
Accepted: v_2 ✔
Reply v_2 to client

Leader#: 3 ✔
Accepted: v_3 ✔
Reply v_3 to client

- Problem: node 3 may wake up, form a quorum of 1 and 3, and accept value v_3
- Need to ensure once a value is chosen by a quorum, it can't be changed
- Use ballot numbers to distinguish different votes: unique for each potential leader

Leader#: ?
Ballot#: 0
Accepted: ?

Leader#: ?
Ballot#: 0
Accepted: ?

Leader#: ?
Ballot#: 0
Accepted: ?

- Phase 1: a prospective leader choses a ballot b and convinces a quorum of acceptors to switch to b
- Acceptor switches only if it's current ballot is smaller

Leader#: ?
Ballot#: 0
Accepted: ?

Leader#: 2
Ballot#: b
Accepted: ?

Leader#: ?
Ballot#: 0
Accepted: ?

- Phase 1: a prospective leader choses a ballot b and convinces a quorum of acceptors to switch to b
- Acceptor switches only if it’s current ballot is smaller

Leader#: 2
Ballot#: b
Accepted: ?

Leader#: 2 ✔
Ballot#: b
Accepted: ?

Leader#: ?
Ballot#: 0
Accepted: ?

- Phase 1: a prospective leader choses a ballot b and convinces a quorum of acceptors to switch to b
- Acceptor switches only if it's current ballot is smaller

Leader#: 2	Leader#: 2 ✔	Leader#: ?
Ballot#: b	Ballot#: b	Ballot#: 0
Accepted: ?	Accepted: v_2@b	Accepted: ?

- Phase 2: the leader sends its value tagged with its ballot number
- Acceptor only accepts a value tagged with the ballot it is in

Server 1	Server 2	Server 3
Leader#: 2	Leader#: 2 ✔	Leader#: ?
Ballot#: b	Ballot#: b	Ballot#: 0
Accepted: v_2@b	Accepted: v_2@b	Accepted: ?

- Phase 2: the leader sends its value tagged with its ballot number
- Acceptor only accepts a value tagged with the ballot it is in

Leader#: 2	Leader#: 2 ✔	Leader#: ?
Ballot#: b	Ballot#: b	Ballot#: 0
Accepted: v_2@b	Accepted: v_2@b✔	Accepted: ?
	Reply v_2 to client	

- Phase 2: the leader sends its value tagged with its ballot number
- Acceptor only accepts a value tagged with the ballot it is in

Leader#: 2
Ballot#: b
Accepted: v_2@b

Leader#: 2 ✔
Ballot#: b
Accepted: v_2@b✔
Reply v_2 to client

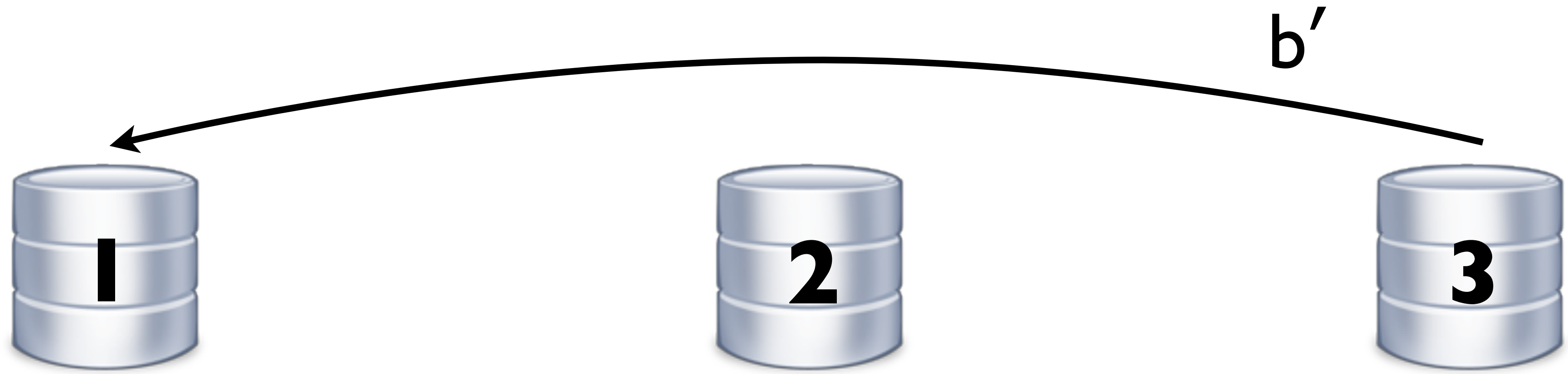
Leader#: ?
Ballot#: 0
Accepted: ?

Leader#: 2
Ballot#: b
Accepted: v_2@b

Leader#: 2 ✔
Ballot#: b
Accepted: v_2@b✔
Reply v_2 to client

Leader#: ?
Ballot#: 0
Accepted: ?

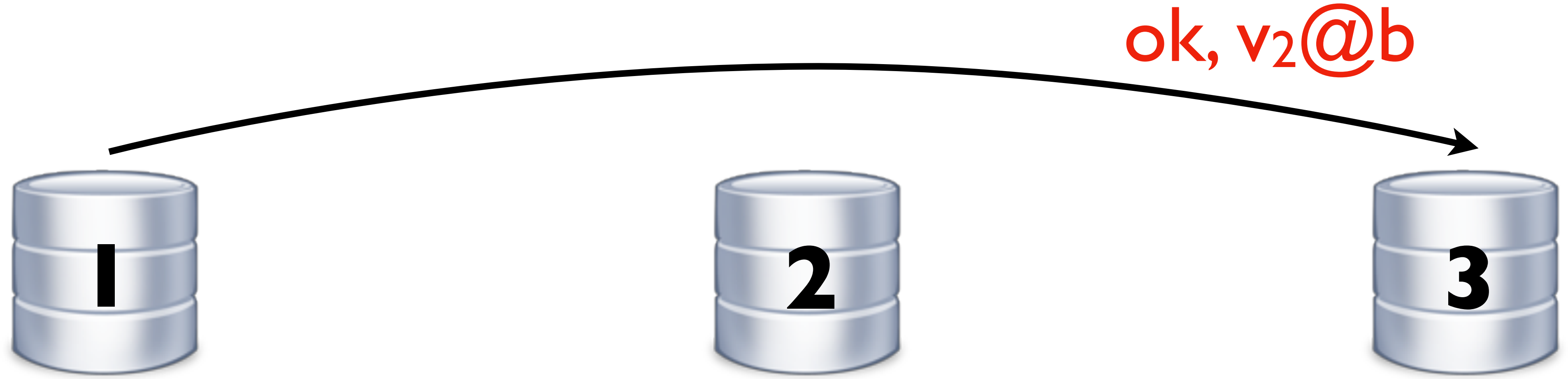
- Need to ensure once a value is chosen by a quorum, it can’t be changed
- Need do change Phase 1 to restrict which values can be proposed



Leader#: 2
Ballot#: b
Accepted: v_2@b

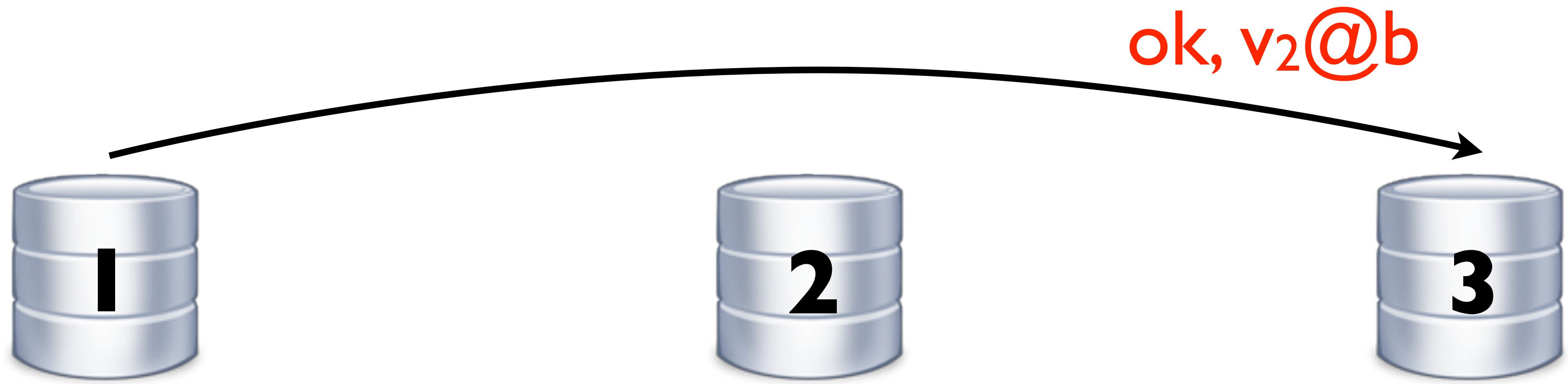
Leader#: 2 ✔
Ballot#: b
Accepted: v_2@b ✔
Reply v_2 to client

Leader#: 3
Ballot#: b′ > b
Accepted: ?



1	2	3
Leader#: 3	Leader#: 2 ✔	Leader#: 3
Ballot#: b′	Ballot#: b	Ballot#: b′ > b
Accepted: v_2@b	Accepted: v_2@b ✔	Accepted: ?
	Reply v_2 to client	

- Phase 1: acceptor sends to the prospective leader its value and the ballot it was accepted at
- If some acceptor has accepted a value, the leader proposes the value accepted at the highest ballot number

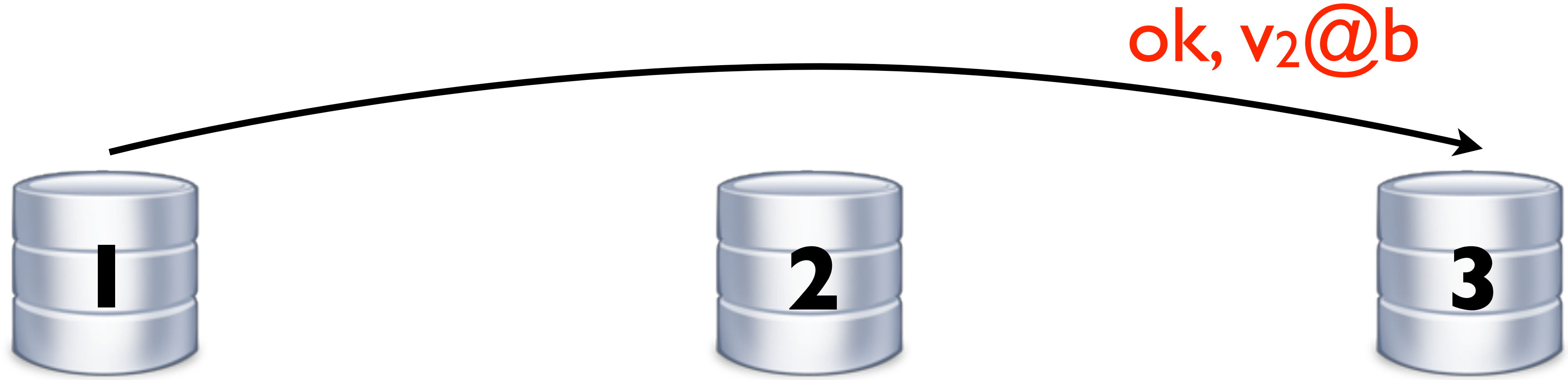


Leader#: 3
Ballot#: b′
Accepted: v_2@b

Leader#: 2 ✔
Ballot#: b
Accepted: v_2@b ✔
Reply v_2 to client

Leader#: 3
Ballot#: b′
Accepted: v_2@b′

- Phase 1: acceptor sends to the prospective leader its value and the ballot it was accepted at
- If some acceptor has accepted a value, the leader proposes the value accepted at the highest ballot number

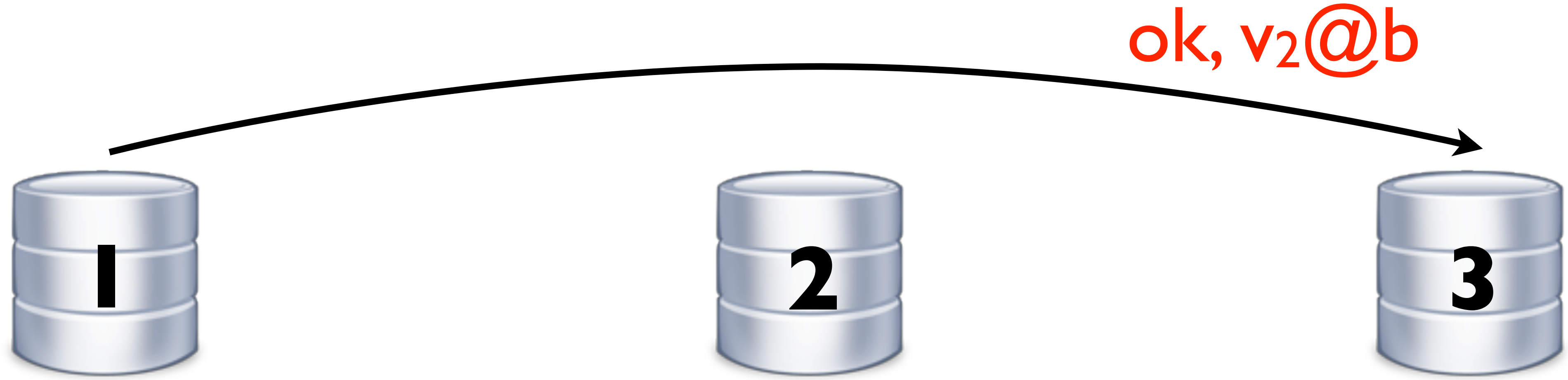


1	2	3
Leader#: 3	Leader#: 2 ✔	Leader#: 3
Ballot#: b′	Ballot#: b	Ballot#: b′
Accepted: v_2@b	Accepted: v_2@b ✔	Accepted: v_2@b′
	Reply v_2 to client	

- Phase 1: acceptor sends to the prospective leader its value and the ballot it was accepted at

- If some acceptor has accepted a value, the leader proposes the value accepted at the highest ballot number

- Ensures the value chosen will not be changed $\Longrightarrow$ nodes don't disagree about the chosen value



Leader#: 3	Leader#: 2 ✔	Leader#: 3
Ballot#: b'	Ballot#: b	Ballot#: b'
Accepted: v_2@b	Accepted: v_2@b ✔	Accepted: v_2@b'
	Reply v_2 to client	

> Key invariant: If a quorum Q accepted a value v at ballot b, then any leader of a ballot $b' > b$ will also propose v

- Ensures the value chosen will not be changed $\Longrightarrow$ nodes don't disagree about the chosen value

Proof of the key invariant

- Invariant: If a quorum Q accepted a value v at ballot b, then any leader of a ballot b′ > b may only propose v

Proof of the key invariant

- Invariant: If a quorum Q accepted a value v at ballot b, then any leader of a ballot b′ > b may only propose v
- Fix an execution of a protocol and assume that in this execution Q accepted v@b.

Proof of the key invariant

- Invariant: If a quorum Q accepted a value v at ballot b, then any leader of a ballot b′ > b may only propose v
- Fix an execution of a protocol and assume that in this execution Q accepted v@b.
- We prove by induction on b′ that: for any b′ > b, leader(b′) may only propose v.

Proof of the key invariant

- Invariant: If a quorum Q accepted a value v at ballot b, then any leader of a ballot $b' > b$ may only propose v
- Fix an execution of a protocol and assume that in this execution Q accepted v@b.
- We prove by induction on b' that: for any $b' > b$, leader(b') may only propose v.
- Consider $b' > b$ and assume leader(b'') may only propose v if $b < b'' < b'$. We prove that leader(b') may only propose v.

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$

- Q accepted v@b
- b′ > b
- leader(b″) may only propose v if b < b″ < b′

- leader(b′) gets support from a quorum Q′ before proposing

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$

- leader(b') gets support from a quorum Q' before proposing
- $Q \cap Q' \neq \varnothing \implies \exists$ process $p \in Q \cap Q'$ which both voted for leader(b') and accepted v@b

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$

- leader(b') gets support from a quorum Q' before proposing
- $Q \cap Q' \neq \varnothing \implies \exists$ process $p \in Q \cap Q'$ which both voted for leader(b') and accepted v@b
- p couldn't accept v@b after voting for leader(b'): after voting, p joins b' and rejects all messages with ballot $b < b'$

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$

- leader(b') gets support from a quorum Q' before proposing
- $Q \cap Q' \neq \varnothing \implies \exists$ process $p \in Q \cap Q'$ which both voted for leader(b') and accepted v@b
- p couldn't accept v@b after voting for leader(b'): after voting, p joins b' and rejects all messages with ballot $b < b'$
- p accepted v@b before voting for leader(b')

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$

- leader(b') gets support from a quorum Q' before proposing
- $Q \cap Q' \neq \varnothing \implies \exists$ process $p \in Q \cap Q'$ which both voted for leader(b') and accepted v@b
- p couldn't accept v@b after voting for leader(b'): after voting, p joins b' and rejects all messages with ballot $b < b'$
- p accepted v@b before voting for leader(b')
- p's ballot when voting for leader(b') is $b_p \geq b > 0$, and it will reply with v'@b_p for some value v'

- Q accepted $v@b$
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$

- leader(b') gets support from a quorum Q' before proposing
- $Q \cap Q' \neq \varnothing \implies \exists$ process $p \in Q \cap Q'$ which both voted for leader(b') and accepted $v@b$
- p couldn't accept $v@b$ after voting for leader(b'): after voting, p joins b' and rejects all messages with ballot $b < b'$
- p accepted $v@b$ before voting for leader(b')
- p's ballot when voting for leader(b') is $b_p \geq b > 0$, and it will reply with $v'@b_p$ for some value v'
- leader(b') can't propose its own value, has to pick one accepted at the highest ballot $b_{max} \geq b$ in the votes it got

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$
- $b_{max} \geq b$

$b_{max} = b$:

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$
- $b_{max} \geq b$

$b_{max} = b$:

- A leader makes a single proposal per ballot, and Q accepted v@b $\Longrightarrow$ any vote v'@b_{max} for leader(b') must have $v' = v$

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$
- $b_{max} \geq b$

$b_{max} = b$:

- A leader makes a single proposal per ballot, and Q accepted v@b $\Longrightarrow$ any vote $v'@b_{max}$ for leader(b') must have $v' = v$
- leader(b') has to choose v, QED.

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$
- $b_{max} \geq b$

$b_{max} > b$:

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$
- $b_{max} \geq b$

$b_{max} > b$:

- $b_{max} < b'$, since processes only vote for leaders of higher ballots

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$
- $b_{max} \geq b$

$b_{max} > b$:

- $b_{max} < b'$, since processes only vote for leaders of higher ballots
- By induction hypothesis leader(b_{max}) could only propose v

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$
- $b_{max} \geq b$

$b_{max} > b$:

- $b_{max} < b'$, since processes only vote for leaders of higher ballots
- By induction hypothesis leader(b_{max}) could only propose v
- Processes that accepted a value at b_{max} could only accept v

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$
- $b_{max} \geq b$

$b_{max} > b$:

- $b_{max} < b'$, since processes only vote for leaders of higher ballots
- By induction hypothesis leader(b_{max}) could only propose v
- Processes that accepted a value at b_{max} could only accept v
- Any vote v'@b_{max} for leader(b') must have $v' = v$

- Q accepted v@b
- $b' > b$
- leader(b'') may only propose v if $b < b'' < b'$
- $b_{max} \geq b$

$b_{max} > b$:

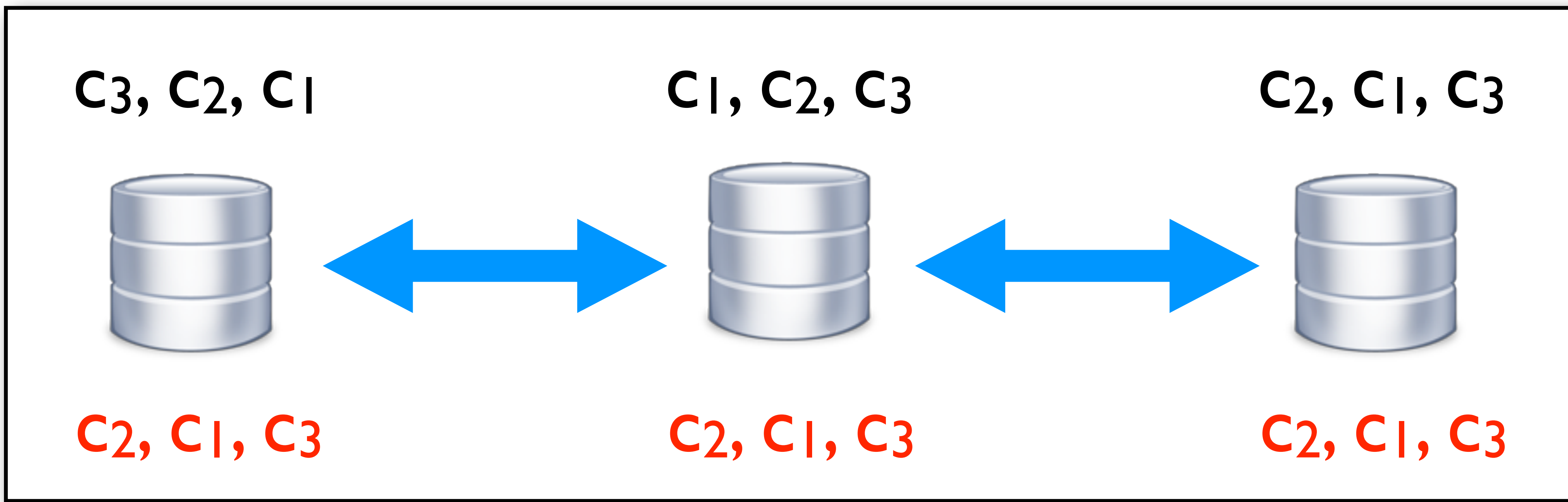
- $b_{max} < b'$, since processes only vote for leaders of higher ballots
- By induction hypothesis leader(b_{max}) could only propose v
- Processes that accepted a value at b_{max} could only accept v
- Any vote v'@b_{max} for leader(b') must have $v' = v$
- leader(b') has to choose v, QED.

Key invariant: If a quorum Q accepted a value v at ballot b, then any leader of a ballot b′ > b will also propose v

Ensures nodes don't disagree about the chosen value

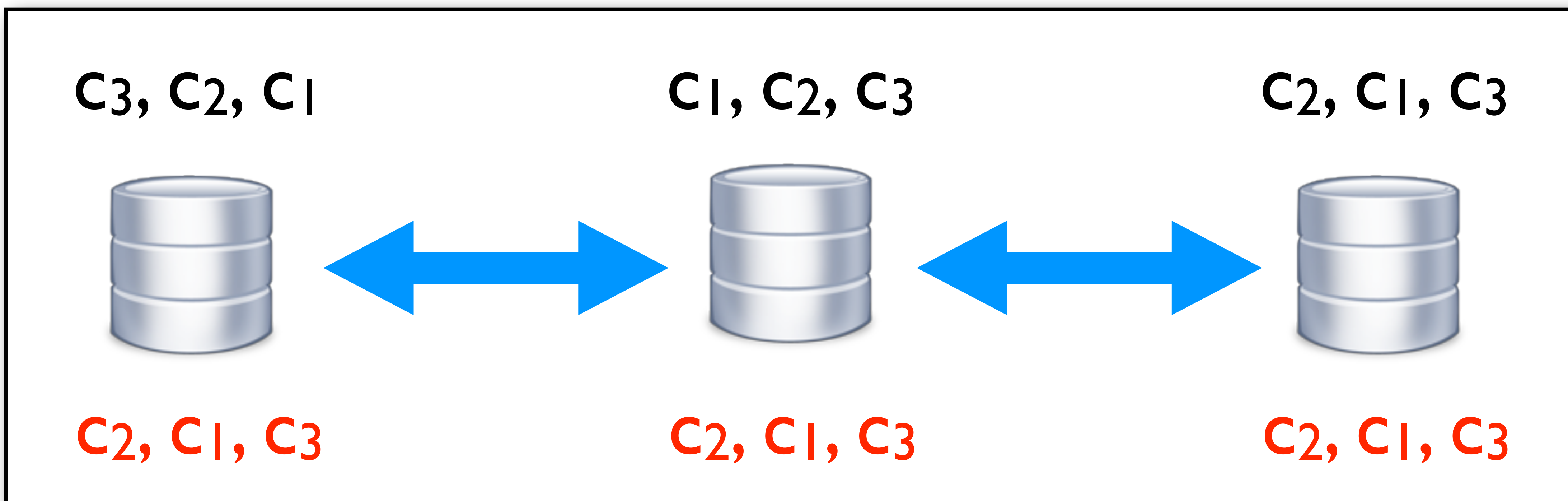
Multi-Paxos

State machine replication requires solving a sequence of consensus instances



Multi-Paxos

State machine replication requires solving a sequence of consensus instances



- Naive solution: execute a separate Paxos instance for each sequence element
- Multi-Paxos: execute Phase 1 once for multiple sequence elements

Paxos verification

- Lots of work on formally verifying Paxos-like protocols in theorem provers or semi-automatic systems
- Fully automatic verification is an open problem

The end

- Spectrum of data consistency models in distributed systems

- Downsides of weakening consistency can be mitigated by verification techniques and programming abstractions: replicated data types, transactions

- Proving correctness of consistency protocols is a verification challenge