Reasoning about Eventual Consistency and Replicated Data Types

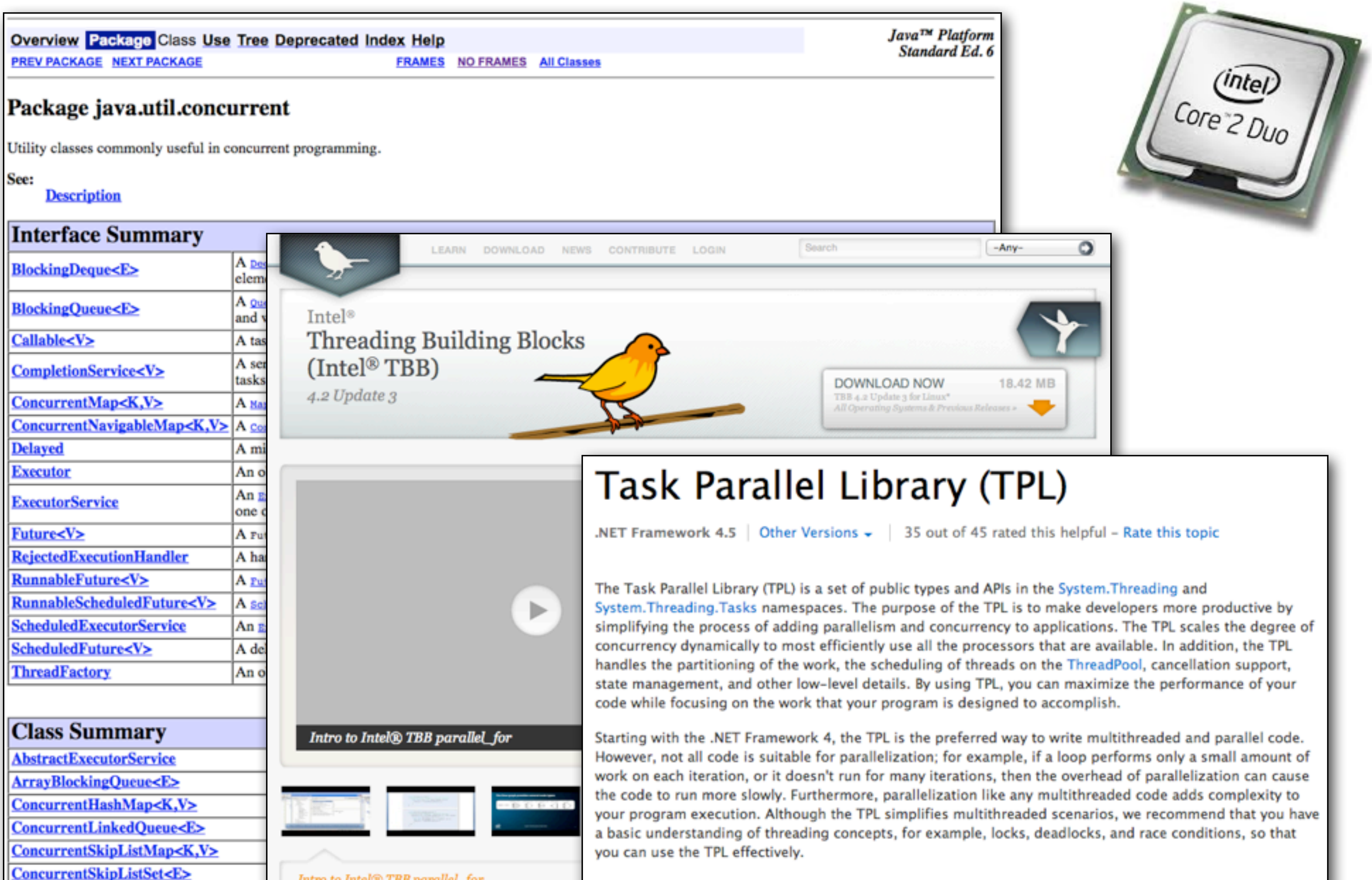
Alexey Gotsman

IMDEA Software Institute

MSR-IMDEA Collaboration Workshop, 04.04.14

Joint work with Sebastian Burckhardt (Microsoft Research), Hongseok Yang (Oxford) and Marek Zawirski (UPMC & INRIA)

Shared-memory concurrency



Package java.util.concurrent

Utility classes commonly useful in concurrent programming.

See:
Description

Interface Summary

- BlockingDeque<E>
- BlockingQueue<E>
- Callable<V>
- CompletionService<V>
- ConcurrentMap<K,V>
- ConcurrentNavigableMap<K,V>
- Delayed
- Executor
- ExecutorService
- Future<V>
- RejectedExecutionHandler
- RunnableFuture<V>
- RunnableScheduledFuture<V>
- ScheduledExecutorService
- ScheduledFuture<V>
- ThreadFactory

Class Summary

- AbstractExecutorService
- ArrayBlockingQueue<E>
- ConcurrentHashMap<K,V>
- ConcurrentLinkedQueue<E>
- ConcurrentSkipListMap<K,V>
- ConcurrentSkipListSet<E>

Intel®
Threading Building Blocks
(Intel® TBB)
4.2 Update 3

Intro to Intel® TBB parallel_for

Task Parallel Library (TPL)

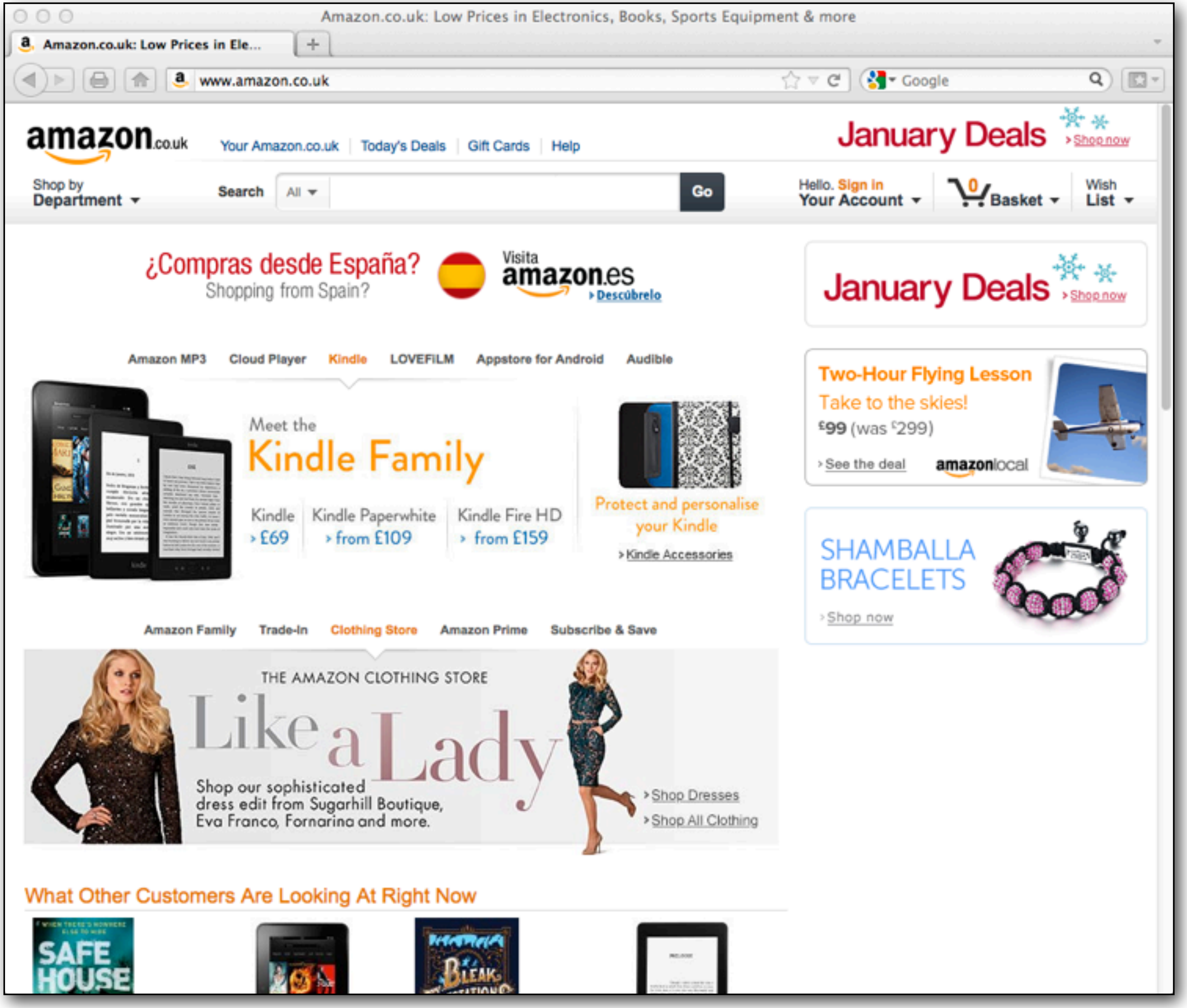
.NET Framework 4.5 | Other Versions | 35 out of 45 rated this helpful – Rate this topic

The Task Parallel Library (TPL) is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces. The purpose of the TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications. The TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available. In addition, the TPL handles the partitioning of the work, the scheduling of threads on the ThreadPool, cancellation support, state management, and other low-level details. By using TPL, you can maximize the performance of your code while focusing on the work that your program is designed to accomplish.

Starting with the .NET Framework 4, the TPL is the preferred way to write multithreaded and parallel code. However, not all code is suitable for parallelization; for example, if a loop performs only a small amount of work on each iteration, or it doesn't run for many iterations, then the overhead of parallelization can cause the code to run more slowly. Furthermore, parallelization like any multithreaded code adds complexity to your program execution. Although the TPL simplifies multithreaded scenarios, we recommend that you have a basic understanding of threading concepts, for example, locks, deadlocks, and race conditions, so that you can use the TPL effectively.

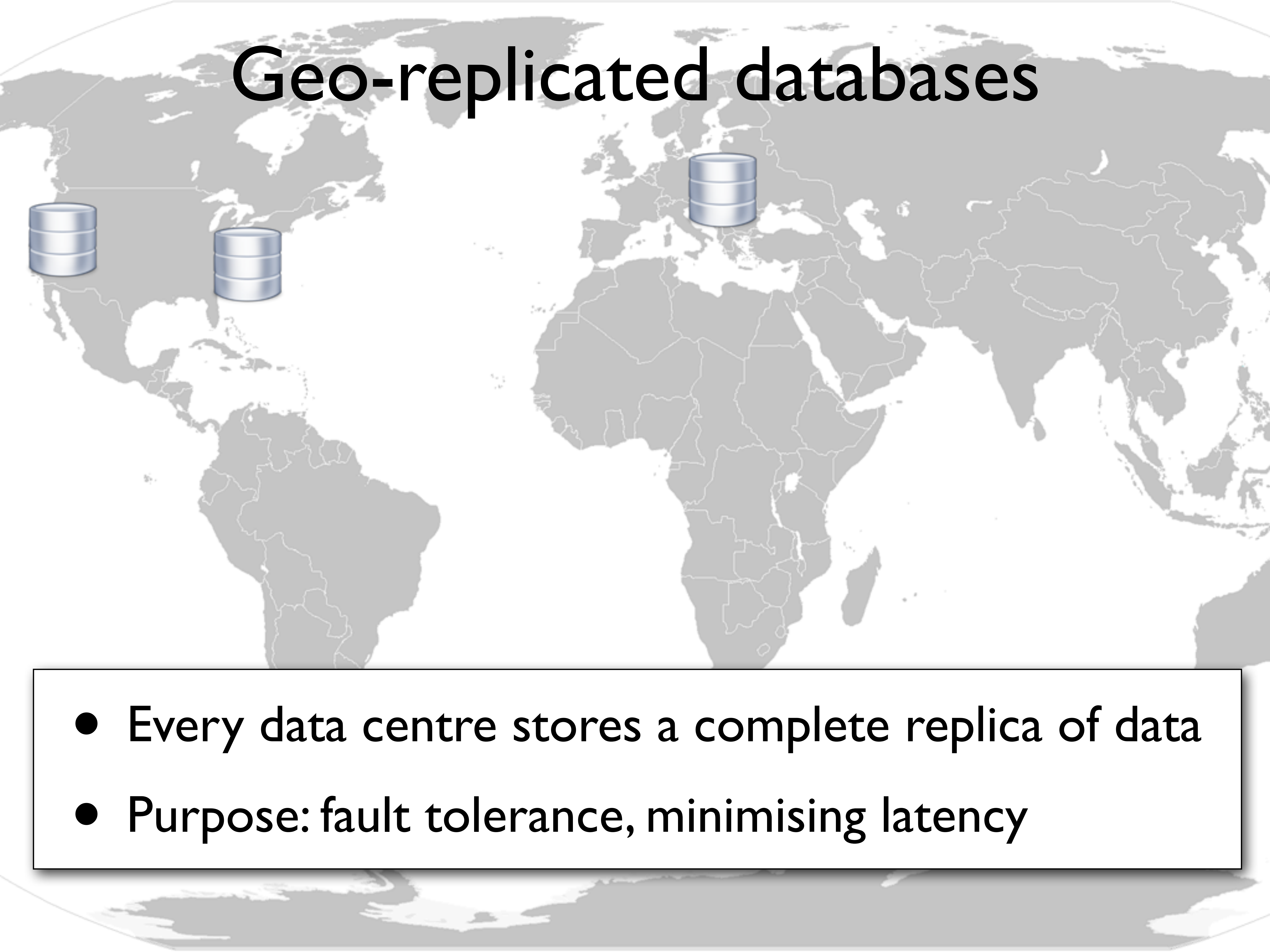
Distributed systems

Distributed systems



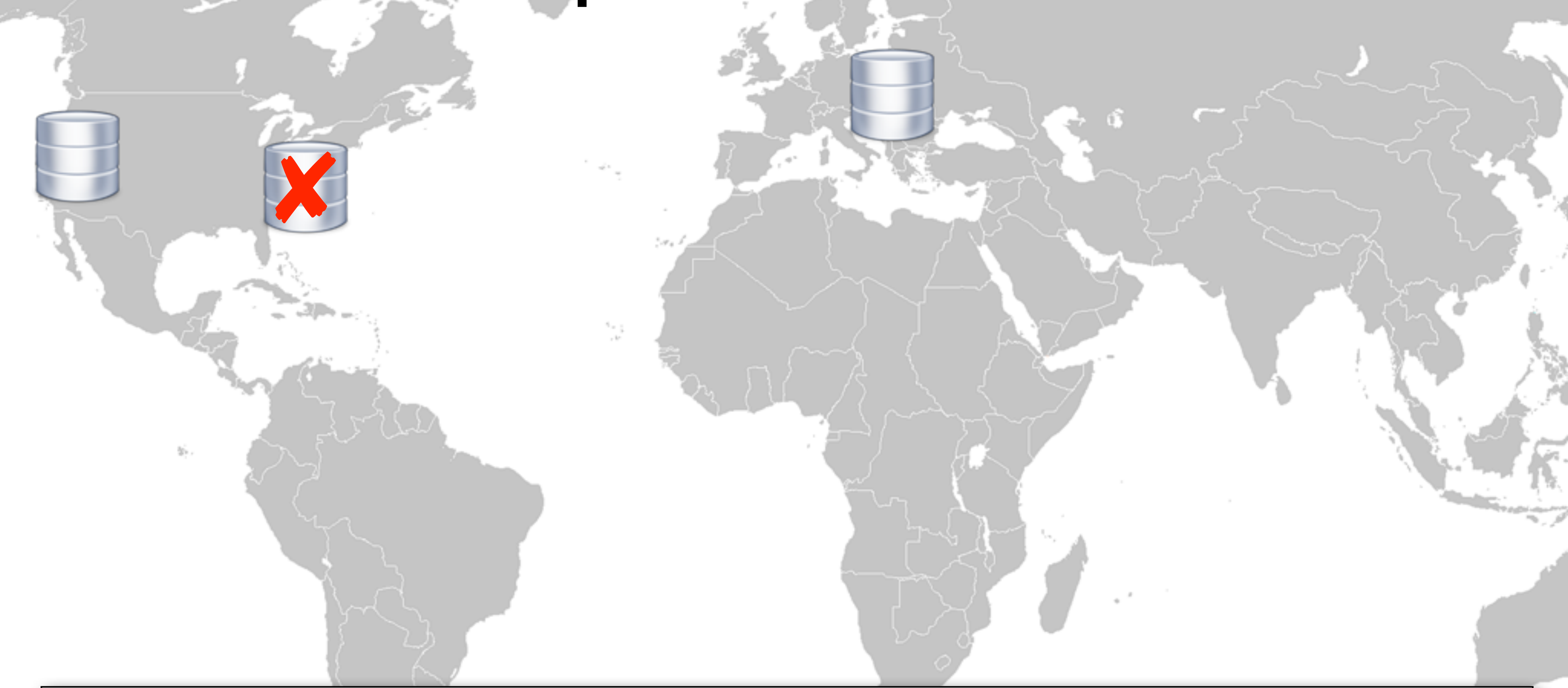
Shop our sophisticated dress edit from Sugarhill Boutique, Eva Franco, Fornarina and more.

Geo-replicated databases



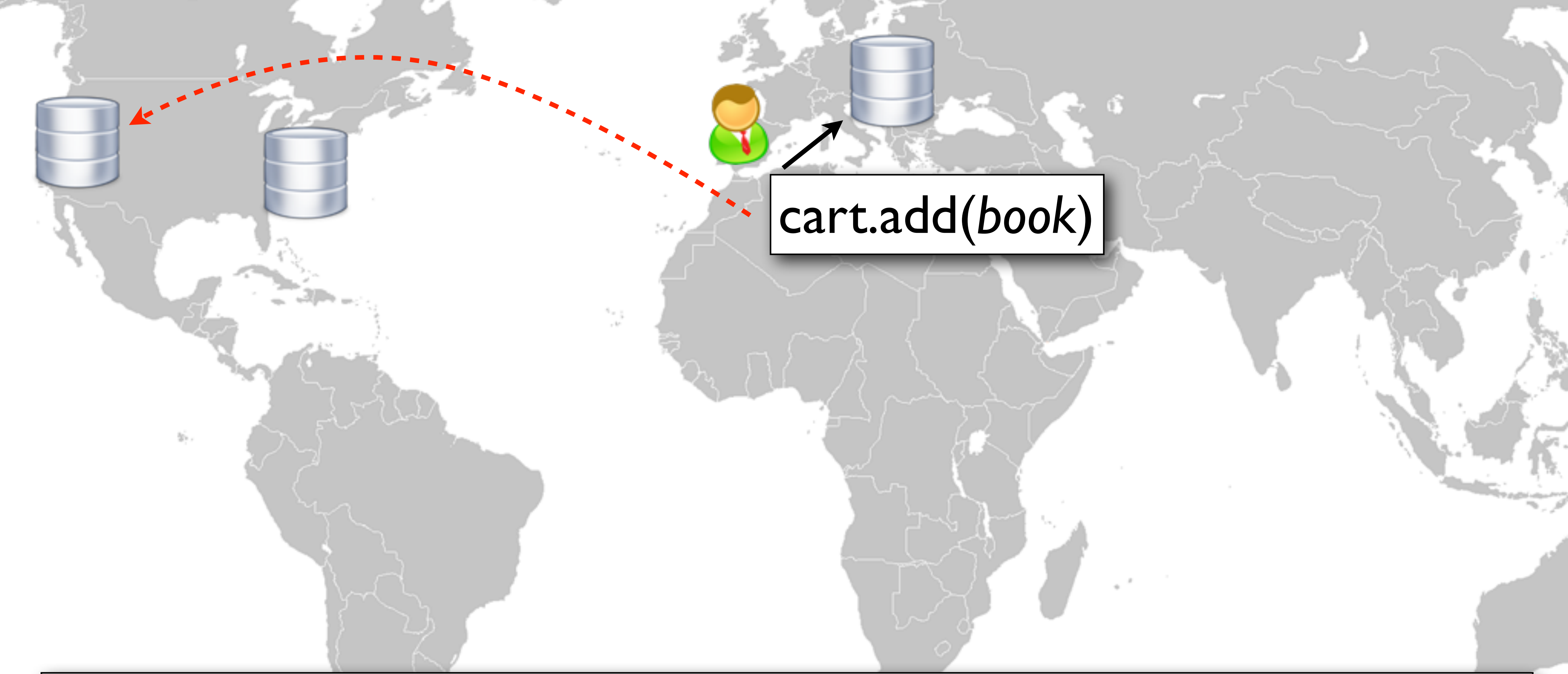
- Every data centre stores a complete replica of data
- Purpose: fault tolerance, minimising latency

Geo-replicated databases



- Every data centre stores a complete replica of data
- Purpose: fault tolerance, minimising latency

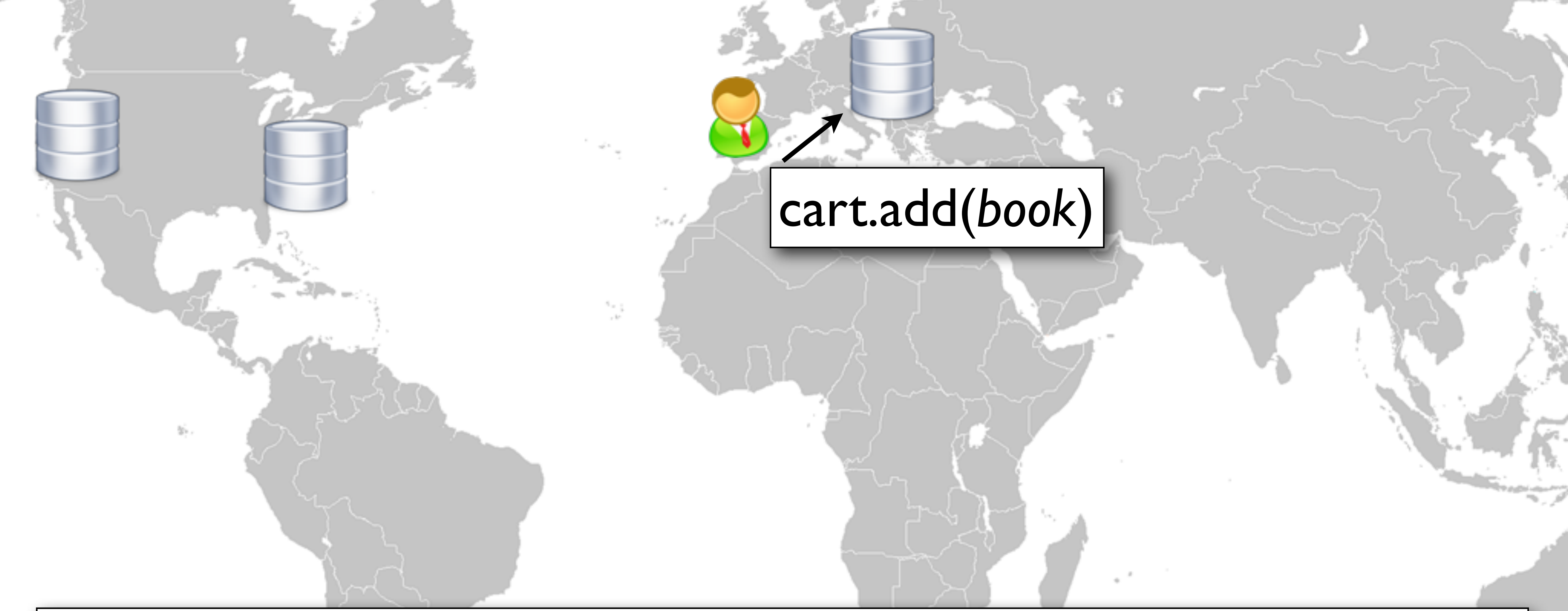
Geo-replicated databases



cart.add(*book*)

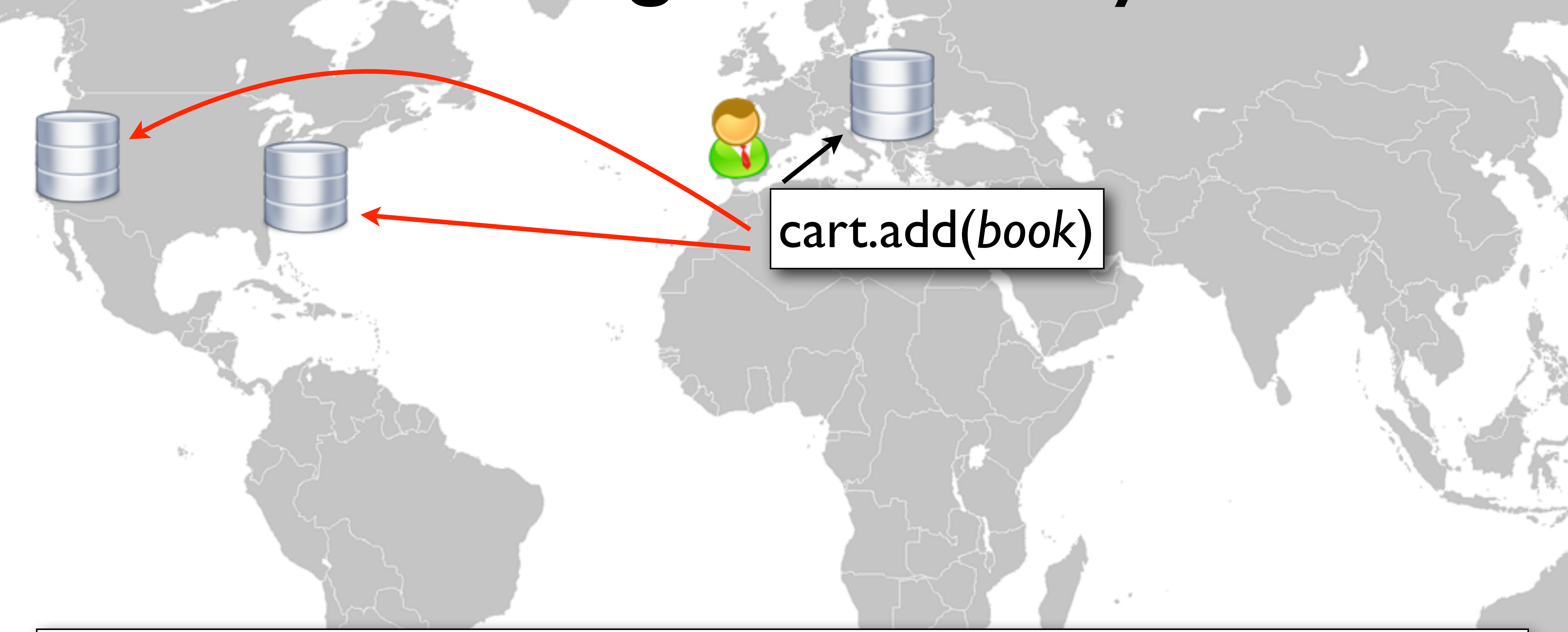
- Every data centre stores a complete replica of data
- Purpose: fault tolerance, minimising latency

Strong consistency



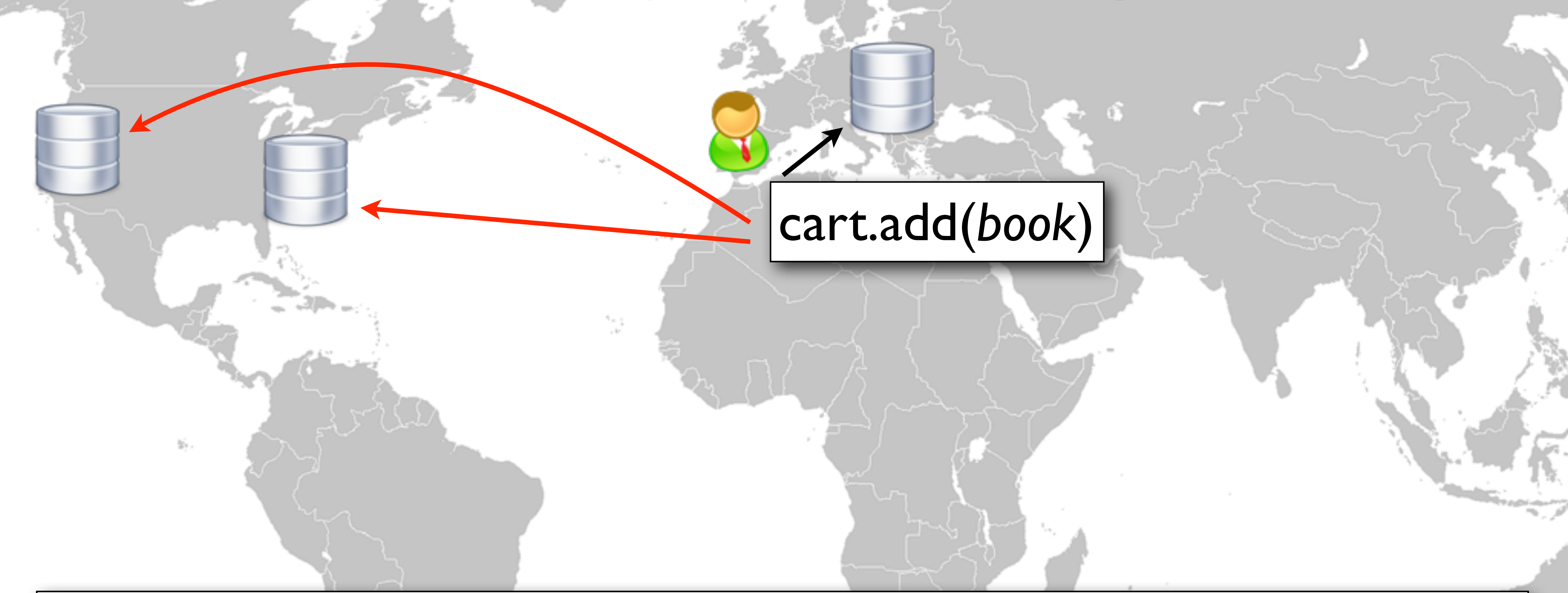
- Database behaves like a single replica
- Implementation: ensure replicas are in sync ➔ wait until other replicas get updated

Strong consistency



- Database behaves like a single replica
- Implementation: ensure replicas are in sync ➔ wait until other replicas get updated

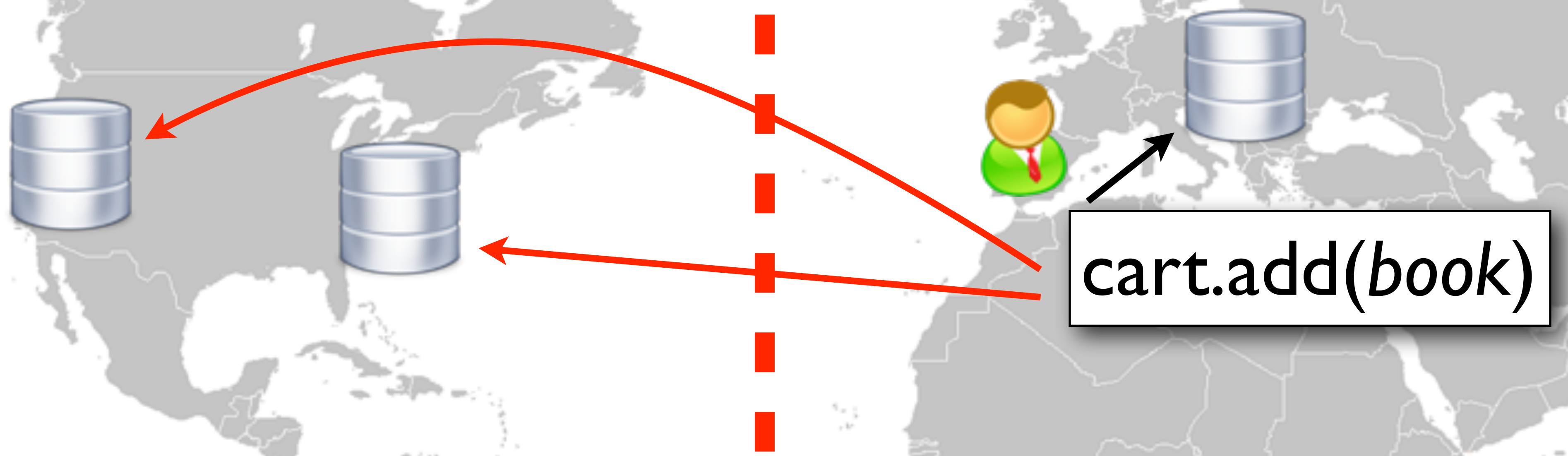
Strong consistency



cart.add(*book*)

- Problem: high latency, can't tolerate network partitions
- CAP theorem: impossible to get all of strong Consistency, Availability, Partition-tolerance

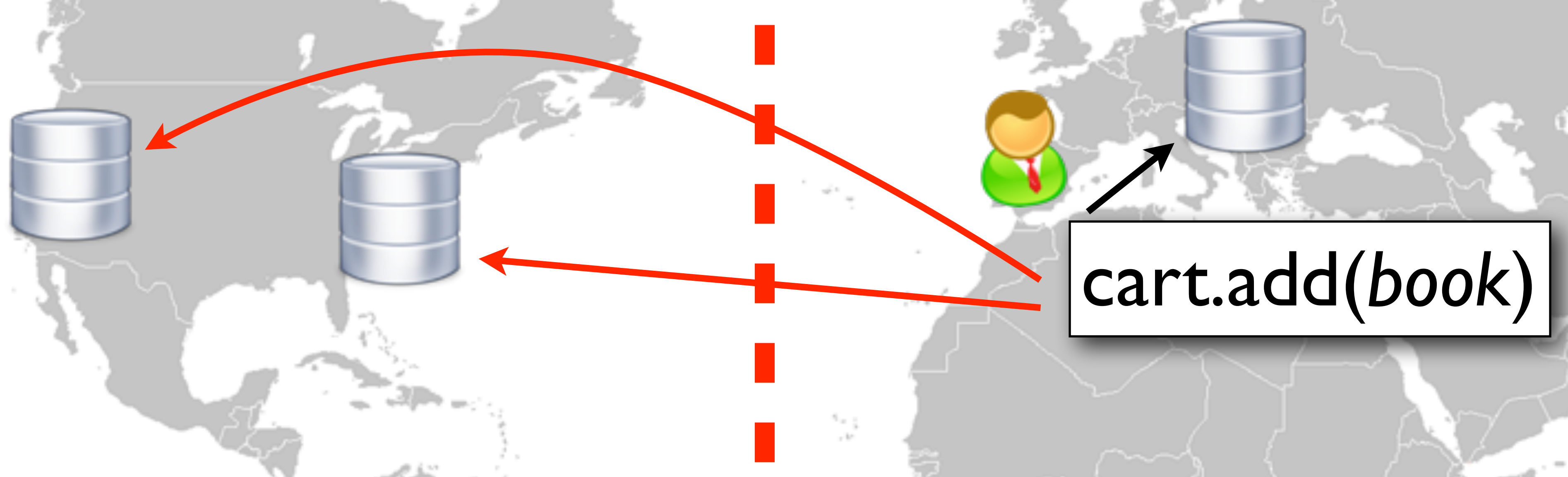
Strong consistency



cart.add(*book*)

- Problem: high latency, can’t tolerate network partitions
- CAP theorem: impossible to get all of strong Consistency, Availability, Partition-tolerance

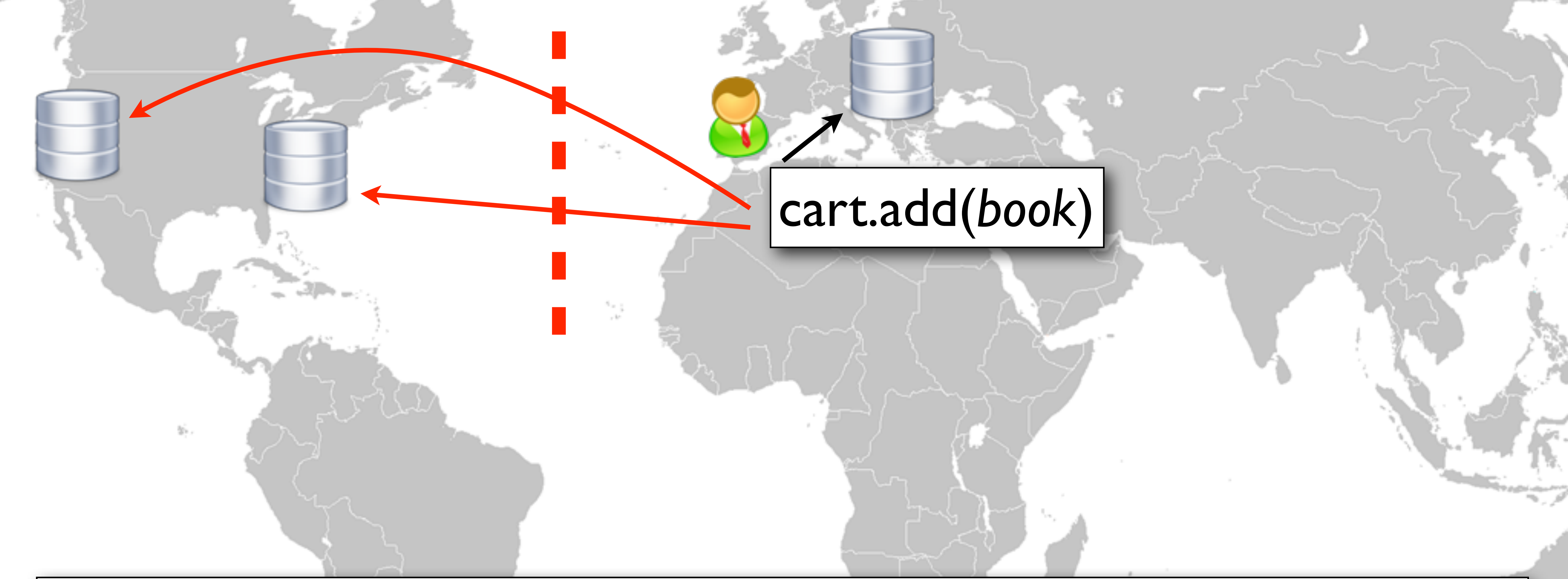
Strong consistency



cart.add(*book*)

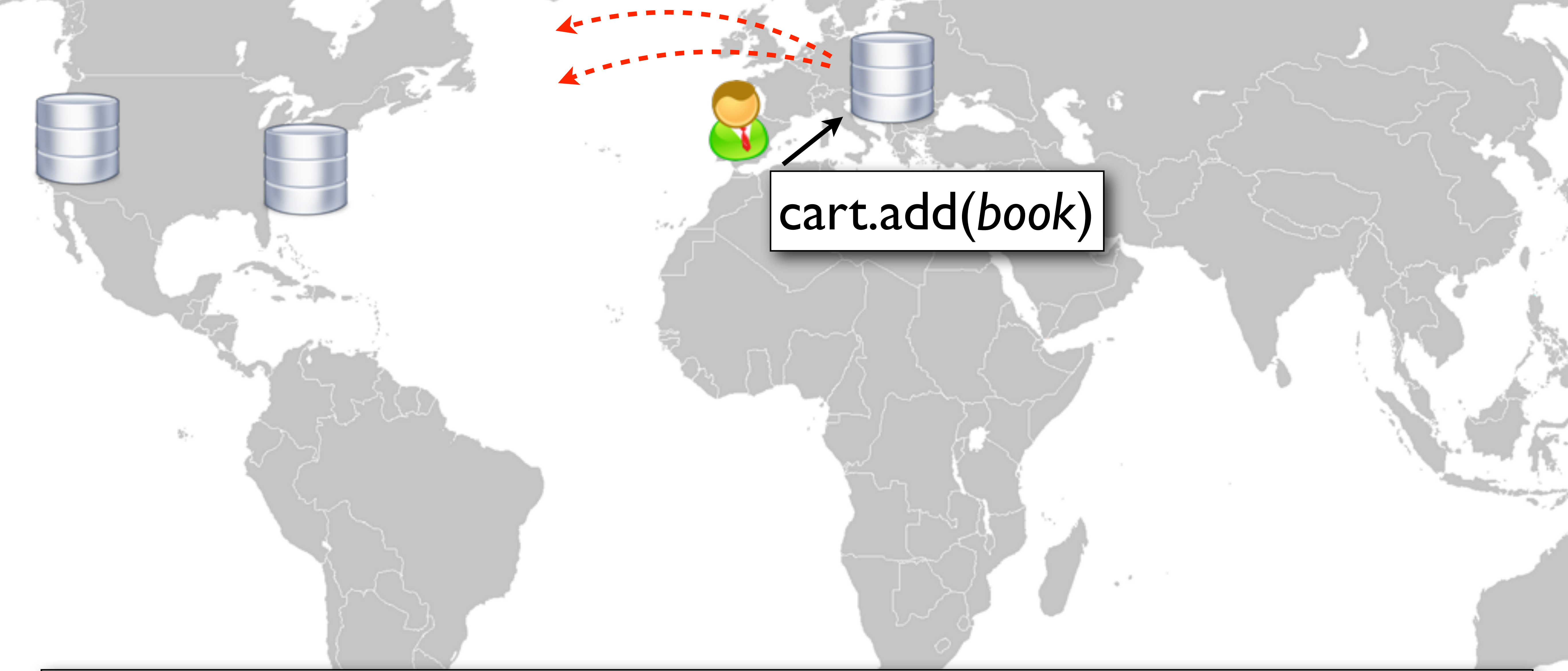
- Problem: high latency, can’t tolerate network partitions
- CAP theorem: impossible to get all of strong Consistency, Availability, Partition-tolerance

Strong consistency



- Problem: high latency, can't tolerate network partitions
- CAP theorem: impossible to get all of ~~strong Consistency~~, Availability, Partition-tolerance

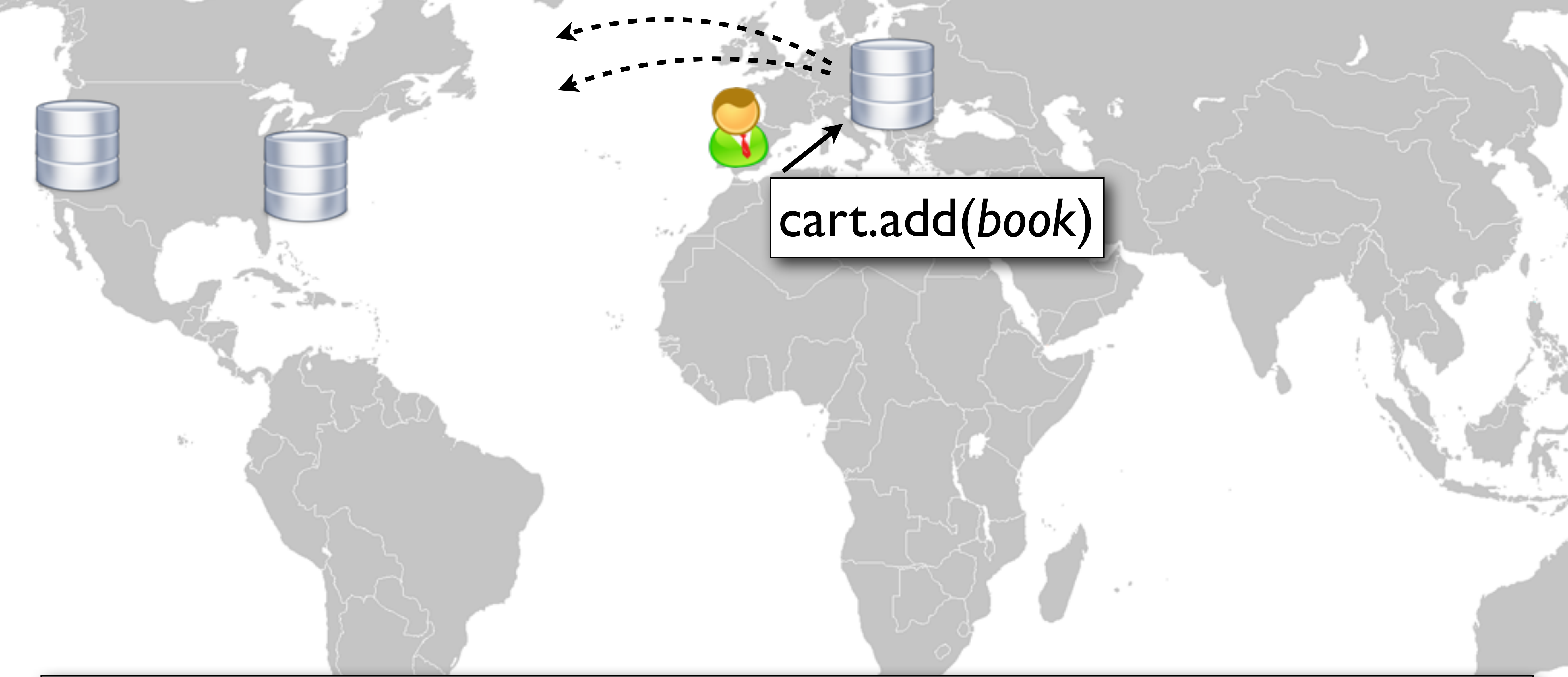
Weak consistency



cart.add(*book*)

- Update your replica now, propagate to others later
- Weak consistency: exhibits anomalies

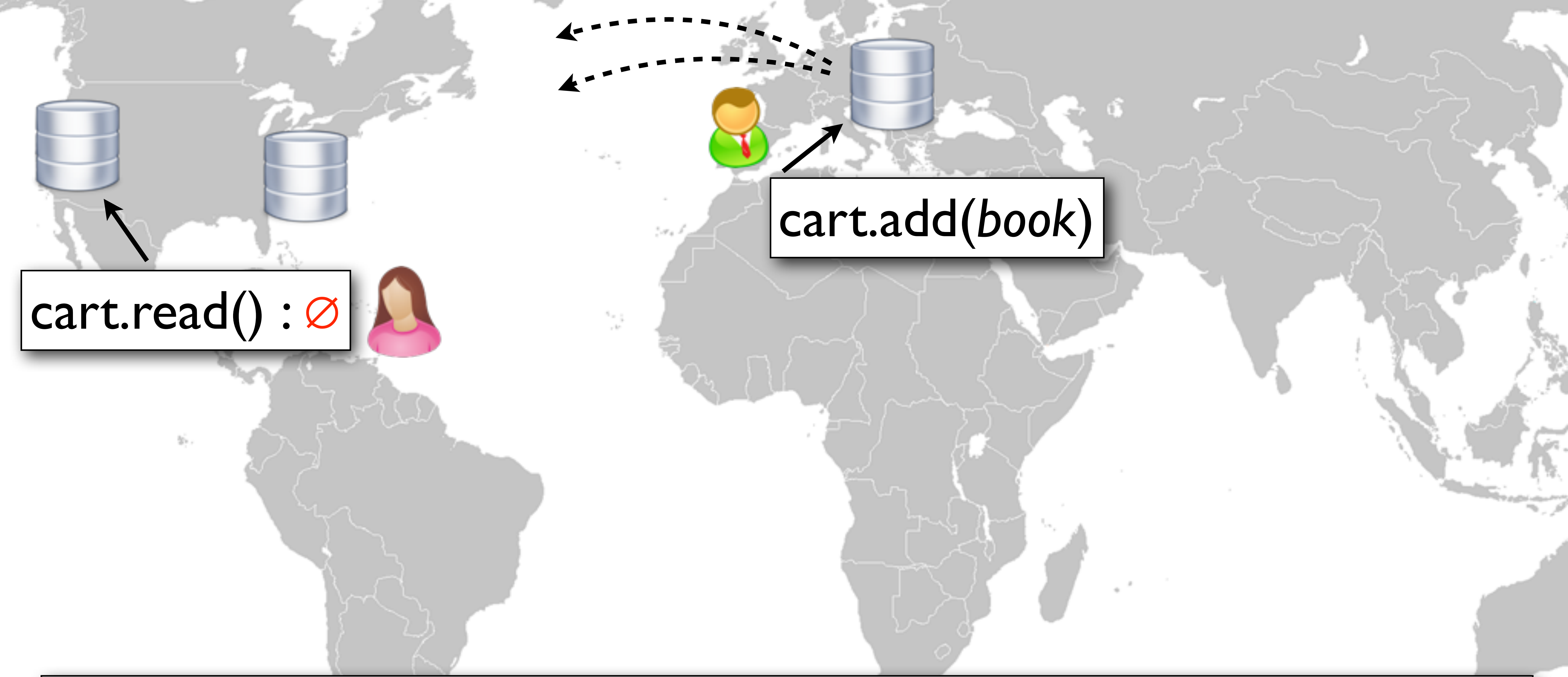
Weak consistency



cart.add(*book*)

- Update your replica now, propagate to others later
- Weak consistency: exhibits anomalies

Weak consistency

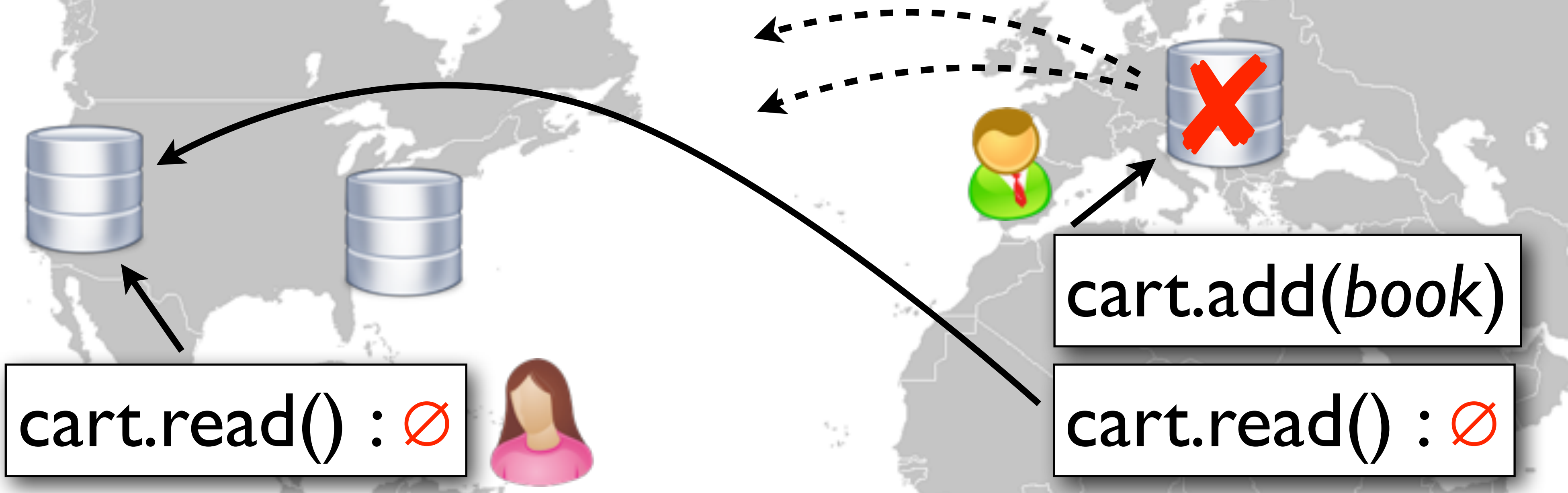


cart.read() : ∅

cart.add(*book*)

- Update your replica now, propagate to others later
- Weak consistency: exhibits anomalies

Weak consistency



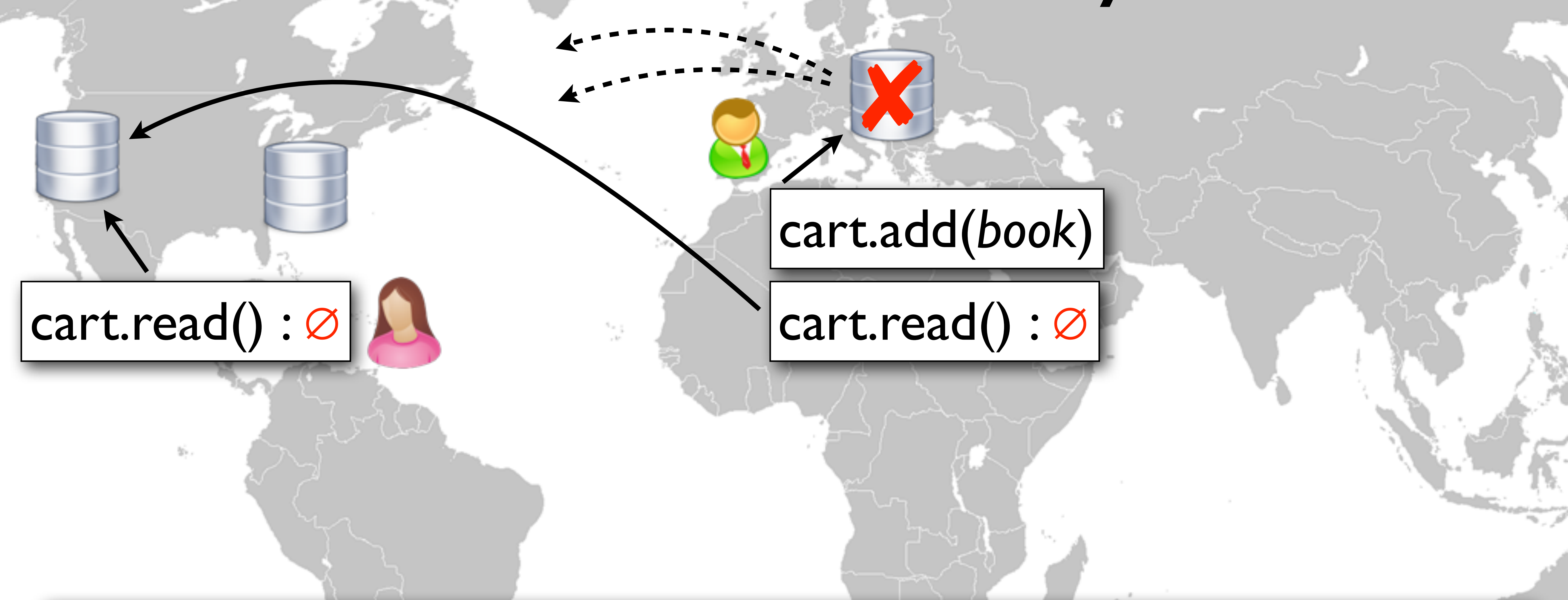
cart.add(*book*)

cart.read() : ∅

cart.read() : ∅

- Update your replica now, propagate to others later
- Weak consistency: exhibits anomalies

Weak consistency



cart.add(*book*)

cart.read() : ∅

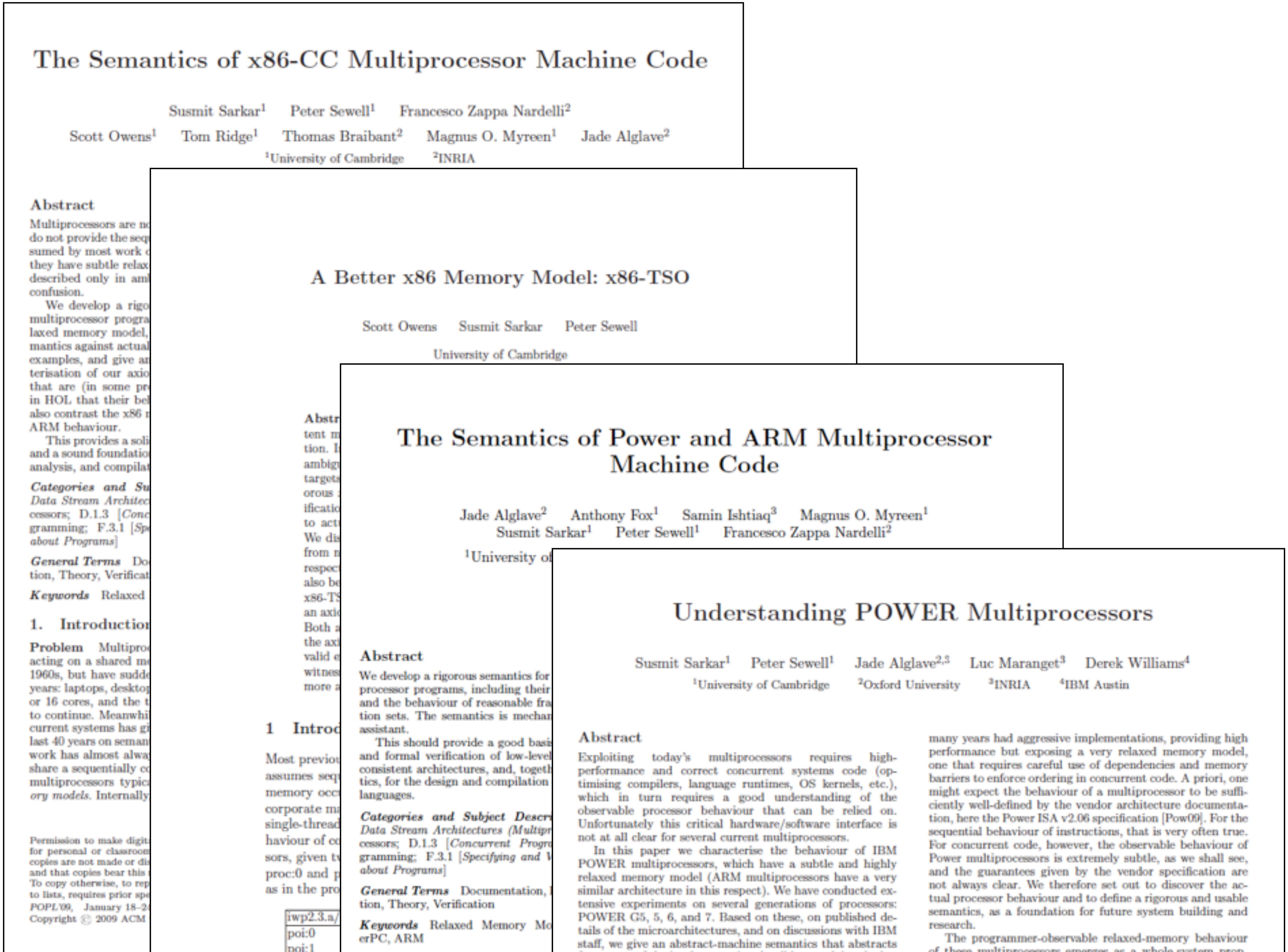
cart.read() : ∅

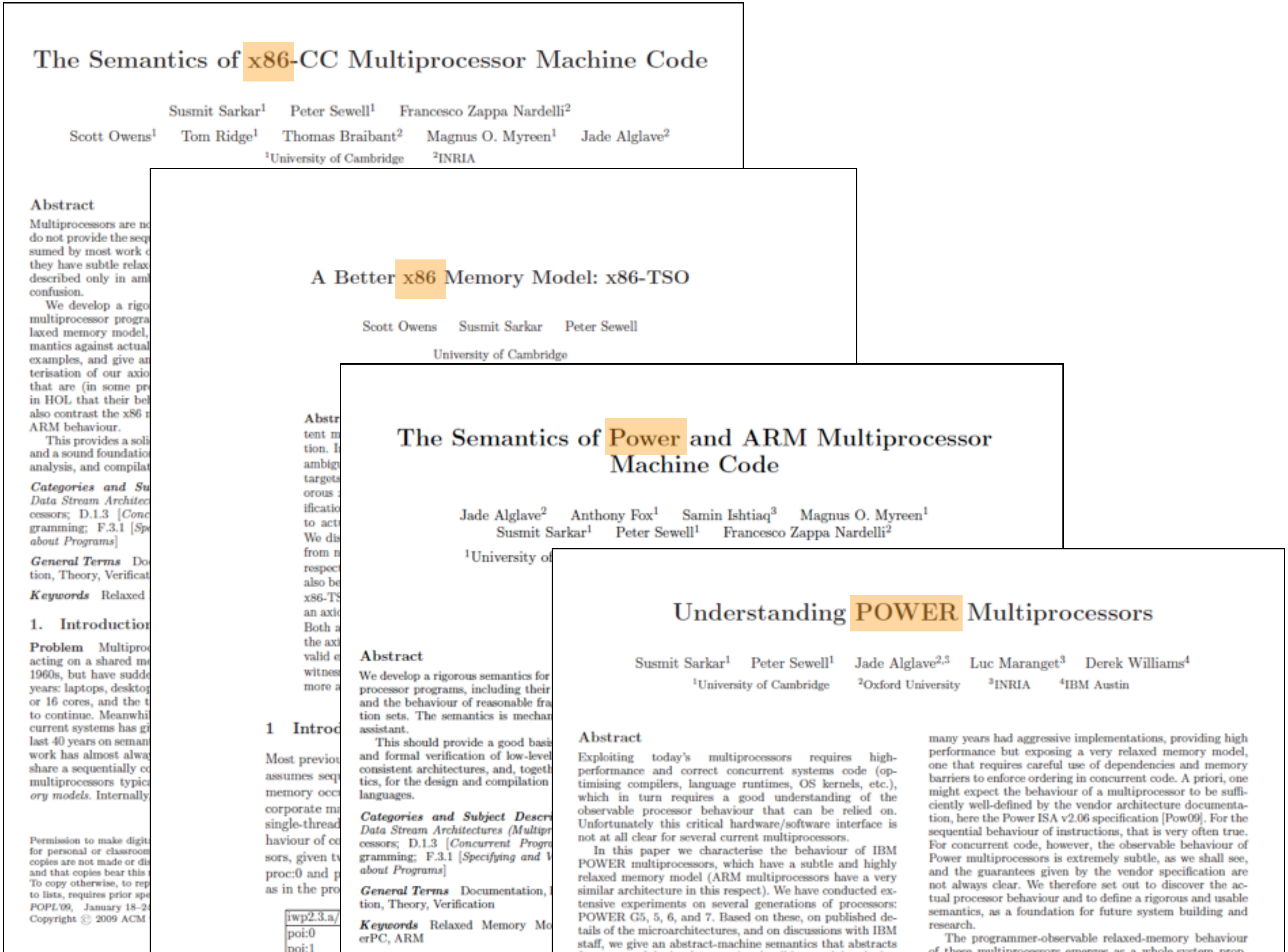
- Also an issue with mobile devices: operate when disconnected

Weak consistency

in shared memory

- Processors and programming languages don't provide strong consistency: weak memory models
- A multiprocessor is really a distributed system: cache-coherence protocol
- Hot topic now, but first had to define the memory models





The Semantics of x86-CC Multiprocessor Machine Code

Susmit Sarkar[1] Peter Sewell[1] Francesco Zappa Nardelli[2]
Scott Owens[1] Tom Ridge[1] Thomas Braibant[2] Magnus O. Myreen[1] Jade Alglave[2]
[1]University of Cambridge [2]INRIA

Abstract

Multiprocessors are no
do not provide the seq
sumed by most work c
they have subtle relax
described only in aml
confusion.
We develop a rigo
multiprocessor progra
laxed memory model,
mantics against actual
examples, and give an
terisation of our axio
that are (in some pr
in HOL that their bel
also contrast the x86 r
ARM behaviour.
This provides a soli
and a sound foundatio
analysis, and compilat

Categories and Su
Data Stream Architec
cessors; D.1.3 [*Conc*
gramming; F.3.1 [*Sp*
about Programs]

General Terms Do
tion, Theory, Verificat

Keywords Relaxed

1. Introduction

Problem Multipro
acting on a shared me
1960s, but have sudde
years: laptops, desktop
or 16 cores, and the t
to continue. Meanwhi
current systems has gi
last 40 years on seman
work has almost alwa
share a sequentially co
multiprocessors typic
ory models. Internally

Permission to make digit
for personal or classroom
copies are not made or dis
and that copies bear this
To copy otherwise, to rep
to lists, requires prior spe
POPL'09, January 18–2
Copyright © 2009 ACM

A Better x86 Memory Model: x86-TSO

Scott Owens Susmit Sarkar Peter Sewell
University of Cambridge

Abstr
tent m
tion. I
ambig
targets
orous
ificatio
to act
We dis
from n
respect
also be
x86-TS
an axi
Both a
the axi
valid e
witnes
more a

1 Introd

Most previou
assumes seq
memory occ
corporate m
single-thread
haviour of co
sors, given t
proc:0 and p
as in the pro

iwp2.3.a/
poi:0
poi:1

The Semantics of Power and ARM Multiprocessor Machine Code

Jade Alglave[2] Anthony Fox[1] Samin Ishtiaq[3] Magnus O. Myreen[1]
Susmit Sarkar[1] Peter Sewell[1] Francesco Zappa Nardelli[2]
[1]University of

Abstract

We develop a rigorous semantics for
processor programs, including their
and the behaviour of reasonable fra
tion sets. The semantics is mechan
assistant.
This should provide a good basis
and formal verification of low-level
consistent architectures, and, togeth
tics, for the design and compilation
languages.

Categories and Subject Descri
Data Stream Architectures (Multipr
cessors; D.1.3 [*Concurrent Progra*
gramming; F.3.1 [*Specifying and V*
about Programs]

General Terms Documentation,
tion, Theory, Verification

Keywords Relaxed Memory Mo
erPC, ARM

Understanding POWER Multiprocessors

Susmit Sarkar[1] Peter Sewell[1] Jade Alglave[2,3] Luc Maranget[3] Derek Williams[4]
[1]University of Cambridge [2]Oxford University [3]INRIA [4]IBM Austin

Abstract

Exploiting today's multiprocessors requires high-performance and correct concurrent systems code (optimising compilers, language runtimes, OS kernels, etc.), which in turn requires a good understanding of the observable processor behaviour that can be relied on. Unfortunately this critical hardware/software interface is not at all clear for several current multiprocessors.

In this paper we characterise the behaviour of IBM POWER multiprocessors, which have a subtle and highly relaxed memory model (ARM multiprocessors have a very similar architecture in this respect). We have conducted extensive experiments on several generations of processors: POWER G5, 5, 6, and 7. Based on these, on published details of the microarchitectures, and on discussions with IBM staff, we give an abstract-machine semantics that abstracts

many years had aggressive implementations, providing high performance but exposing a very relaxed memory model, one that requires careful use of dependencies and memory barriers to enforce ordering in concurrent code. A priori, one might expect the behaviour of a multiprocessor to be sufficiently well-defined by the vendor architecture documentation, here the Power ISA v2.06 specification [Pow09]. For the sequential behaviour of instructions, that is very often true. For concurrent code, however, the observable behaviour of Power multiprocessors is extremely subtle, as we shall see, and the guarantees given by the vendor specification are not always clear. We therefore set out to discover the actual processor behaviour and to define a rigorous and usable semantics, as a foundation for future system building and research.

The programmer-observable relaxed-memory behaviour

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-

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

But updates never stop!

So what does this tell database clients?

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-

50 shades of eventual consistency

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

Wyatt Lloyd[*], Michael J. Freedman[*], Michael Kaminsky[†], and David G. Andersen[‡]
[*]Princeton University, [†]Intel Labs, [‡]Carnegie Mellon University

Consistency-Based Service Level Agreements for Cloud Storage

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

Microsoft Research Silicon Valley
ersity

Eventually Consistent Transactions

Sebastian Burckhardt[1], Daan Leijen[1], Manuel Fähndrich[1], and Mooly Sagiv[2]

[1] Microsoft Research
[2] Tel-Aviv

Conflict-free Replicated Data Types [*]

Marc Shapiro[1,5], Nuno Preguiça[2,1], Carlos Baquero[3], and Marek Zawirski[1,4]

[1] INRIA, Paris, France
[2] CITI, Universidade Nova de Lisboa, Portugal
Portugal

50 shades of eventual consistency

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

Wyatt Lloyd[⋆], Michael J. Freedman[⋆], Michael Kaminsky[†], and David G. Andersen[‡]
[⋆]Princeton University, [†]Intel Labs, [‡]Carnegie Mellon University

Consistency-Based Service Level Agreements for Cloud Storage

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla,
Mahesh Balakrishnan, Marcos K. Aguilera, Hussam Abu-Libdeh[†]
Microsoft Research Silicon Valley
ersity

Eventually Consistent Transactions

Sebastian Burckhardt[1], Daan Leijen[1], Manuel Fähndrich[1], and Mooly Sagiv[2]
[1] Microsoft Research
[2] Tel-Aviv U

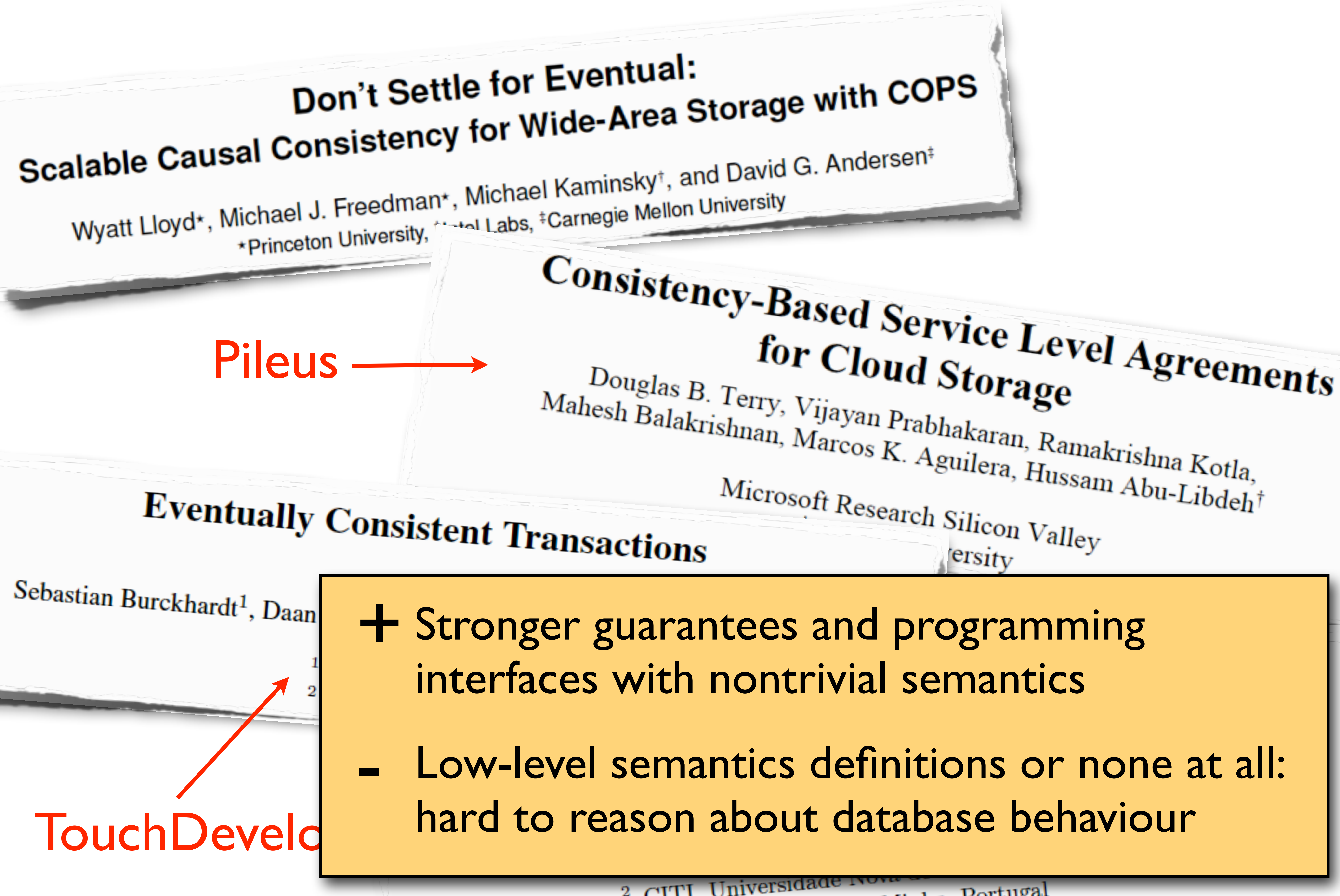
Conflict-free Replicated Data Types [⋆]

Marc Shapiro[1,5], Nuno Preguiça[2,1], Carlos Baquero[3], and Marek Zawirski[1,4]
[1] INRIA, Paris, France
[2] CITI, Universidade Nova de Lisboa, Portugal
Minho, Portugal

Pileus →

TouchDevelop ↗

50 shades of eventual consistency



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

Wyatt Lloyd[*], Michael J. Freedman[*], Michael Kaminsky[†], and David G. Andersen[‡]
[*]Princeton University, tel Labs, [‡]Carnegie Mellon University

Consistency-Based Service Level Agreements
for Cloud Storage

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla,
Mahesh Balakrishnan, Marcos K. Aguilera, Hussam Abu-Libdeh[†]
Microsoft Research Silicon Valley
ersity

Eventually Consistent Transactions

Sebastian Burckhardt[1], Daan
1
2
2 CITI, Universidade Nova
Portugal

Pileus →

TouchDevelo

- \+ Stronger guarantees and programming interfaces with nontrivial semantics
- \- Low-level semantics definitions or none at all: hard to reason about database behaviour

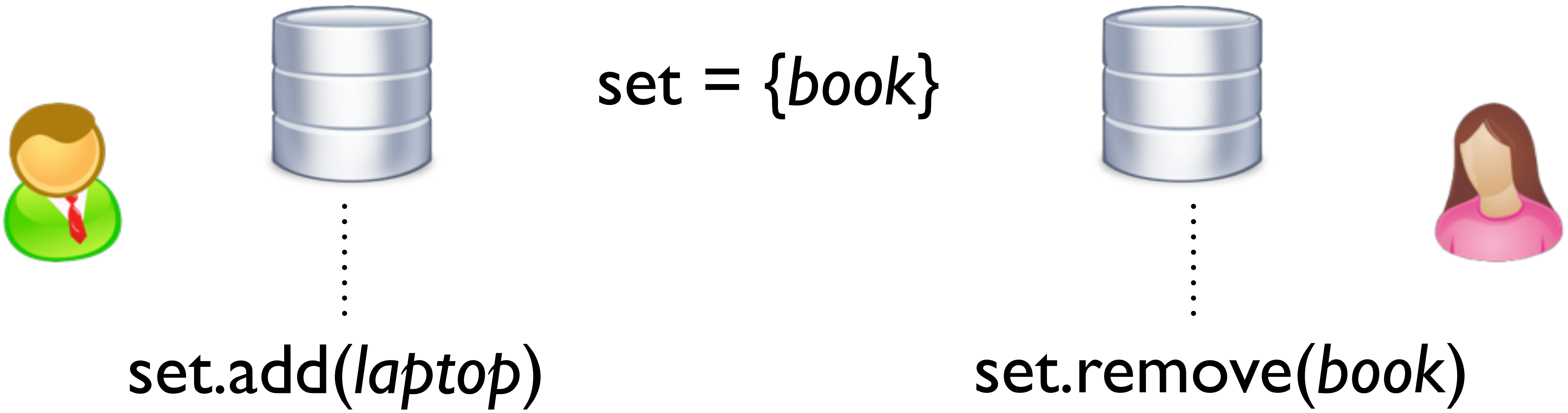
Key issues beyond ‘eventual’

If updates stop, replicas will eventually reach the same state

1. Which anomalies can we see before this?
   E.g., does a user always see his own actions?

2. Which state will replicas converge to?
   Users can make conflicting updates.
   How does the database resolve the conflicts?

Set ~ Shopping cart

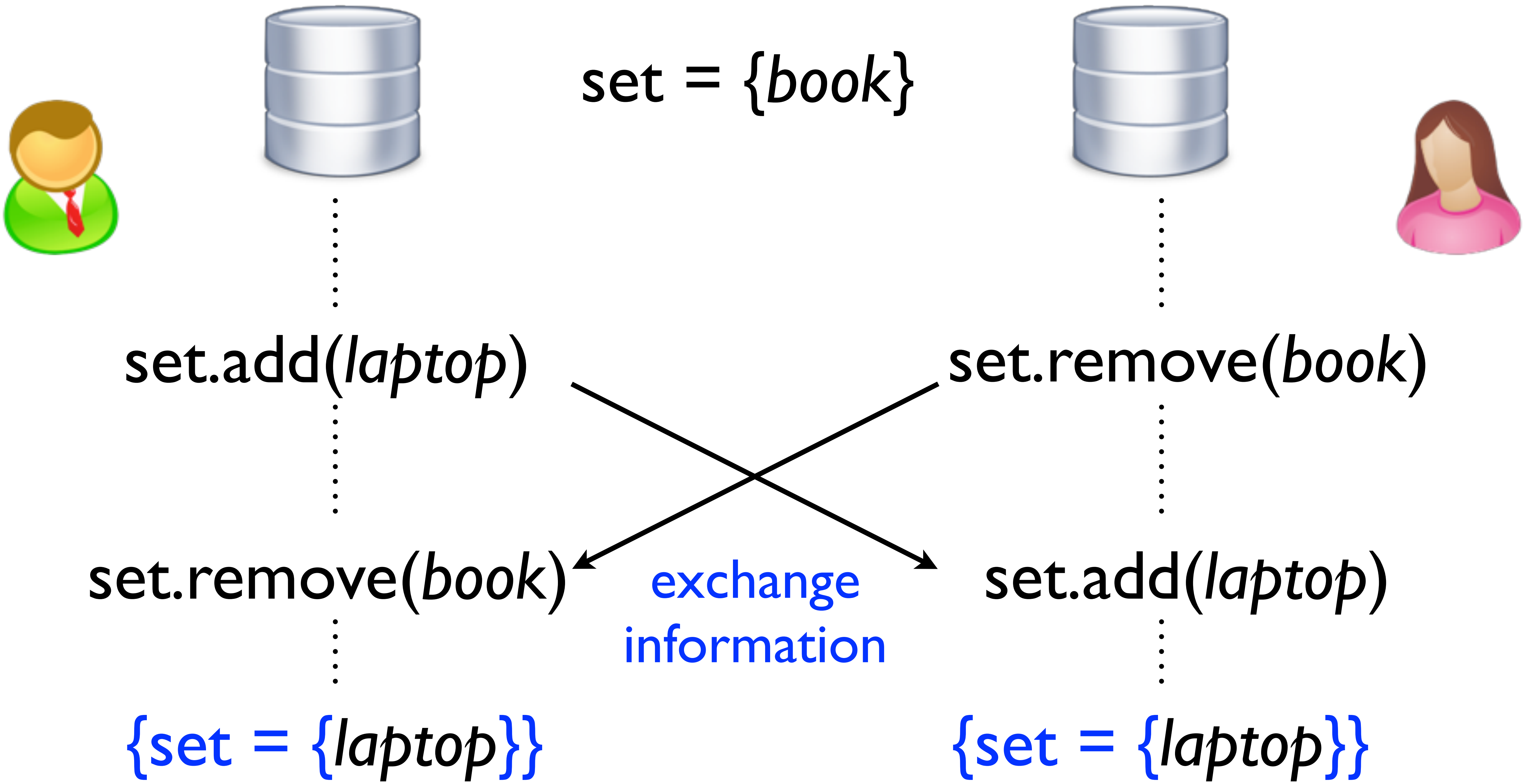


set = {*book*}

set.add(*laptop*)

set.remove(*book*)

Set ~ Shopping cart



set = {*book*}

set.add(*laptop*)

set.remove(*book*)

set.remove(*book*)

exchange information

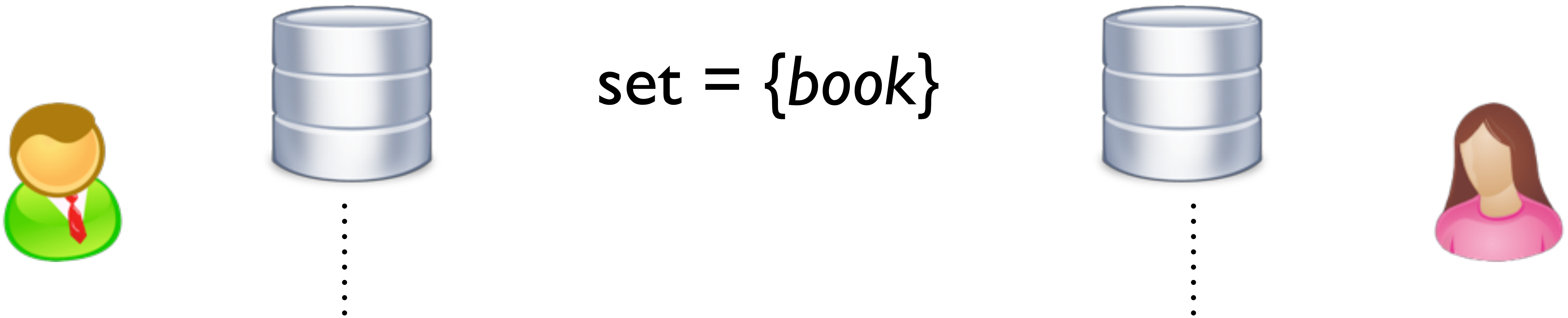
set.add(*laptop*)

{set = {*laptop*}}

{set = {*laptop*}}

Operations commute ➔ eventual consistency OK

Set ~ Shopping cart

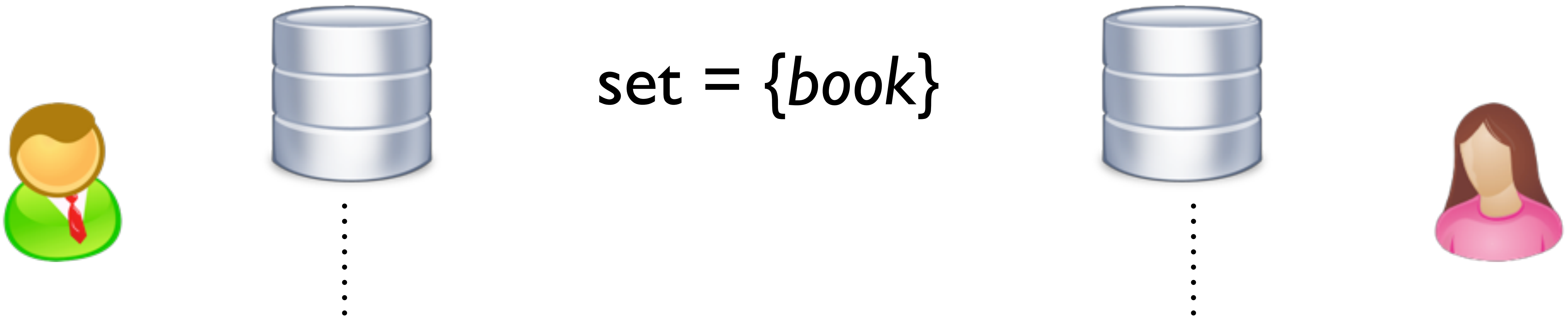


set = {*book*}

set.add(*book*)

set.remove(*book*)

Set ~ Shopping cart



set = {*book*}

set.add(*book*)

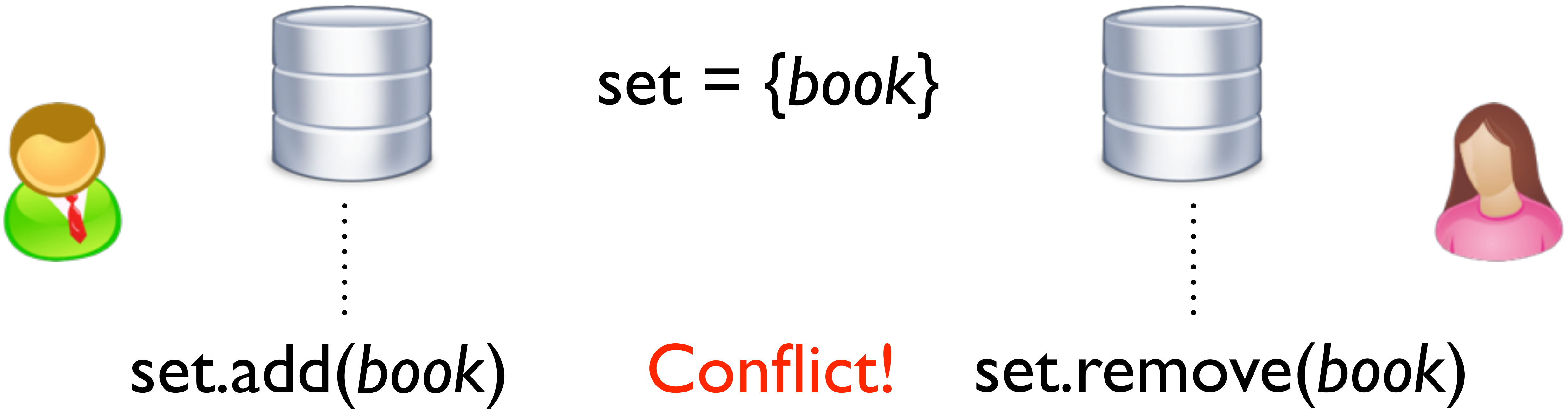
Conflict!

set.remove(*book*)

Should the remove cancel the concurrent add?

Depends on application requirements

Set ~ Shopping cart



set = {*book*}

set.add(*book*)

Conflict!

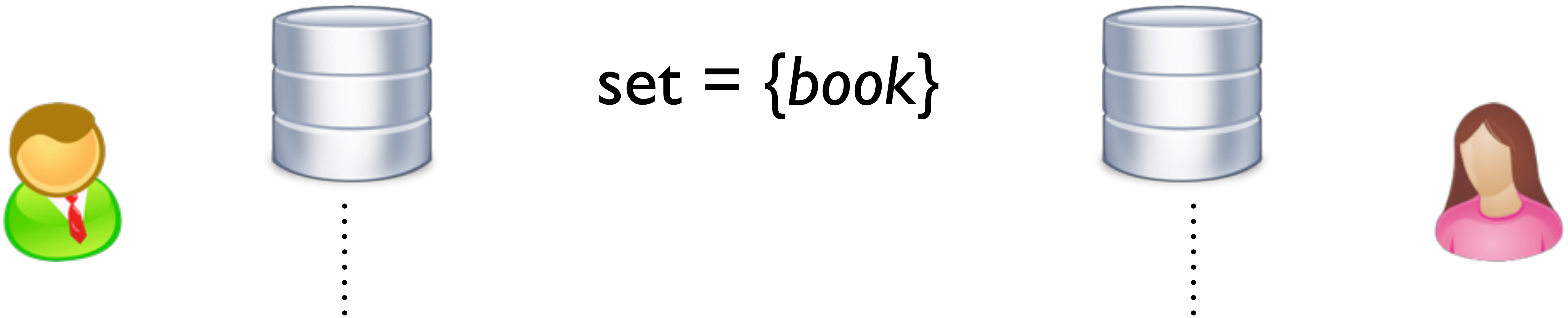
set.remove(*book*)

Remove wins: set = ∅

Add wins: set = {*book*}

Last writer wins: choose based on operation time-stamps

Set ~ Shopping cart



set = {*book*}

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

Remove wins: set = ∅

Add wins: set = {*book*}

Last writer wins: choose based on operation time-stamps

Replicated data types

aka CRDTs, cloud types

Object → Type → Conflict resolution policy

- Many data types: registers, counters, graphs, lists, file systems [Shapiro+ 2011]
- Nontrivial implementations

Replicated data types

aka CRDTs, cloud types

Object → Type → Conflict resolution policy

- Many data types: registers, counters, graphs, lists, file systems [Shapiro+ 2011]
- Nontrivial implementations

So far: implementation is your specification

Long-term goal

Use formal techniques to:

- Define the semantics of eventually consistent databases
- Develop tools for reasoning about their behaviour
- Improve programmability and efficiency

Results [POPL'14]

- Specification:
  - Conflict resolution ~ replicated data types
  - Anomalies

Results [POPL'14]

- Specification:
  - Conflict resolution ~ replicated data types
  - Anomalies
- Verification: framework for proving correctness of replicated data type implementations

Results [POPL'14]

- Specification:
  - Conflict resolution ~ replicated data types
  - Anomalies
- Verification: framework for proving correctness of replicated data type implementations
- Optimality:
  - Data types maintain metadata for conflict resolution
  - Method for proving lower bounds on metadata space requirements

Results [POPL'14]

- Specification:
  - Conflict resolution ~ replicated data types
  - Anomalies
- Verification: framework for proving correctness of replicated data type implementations
- Optimality:
  - Data types maintain metadata for conflict resolution
  - Method for proving lower bounds on metadata space requirements
- Applications to nontrivial data types

Results [POPL'14]

- Specification:
    - Conflict resolution ~ replicated data types
    - Anomalies

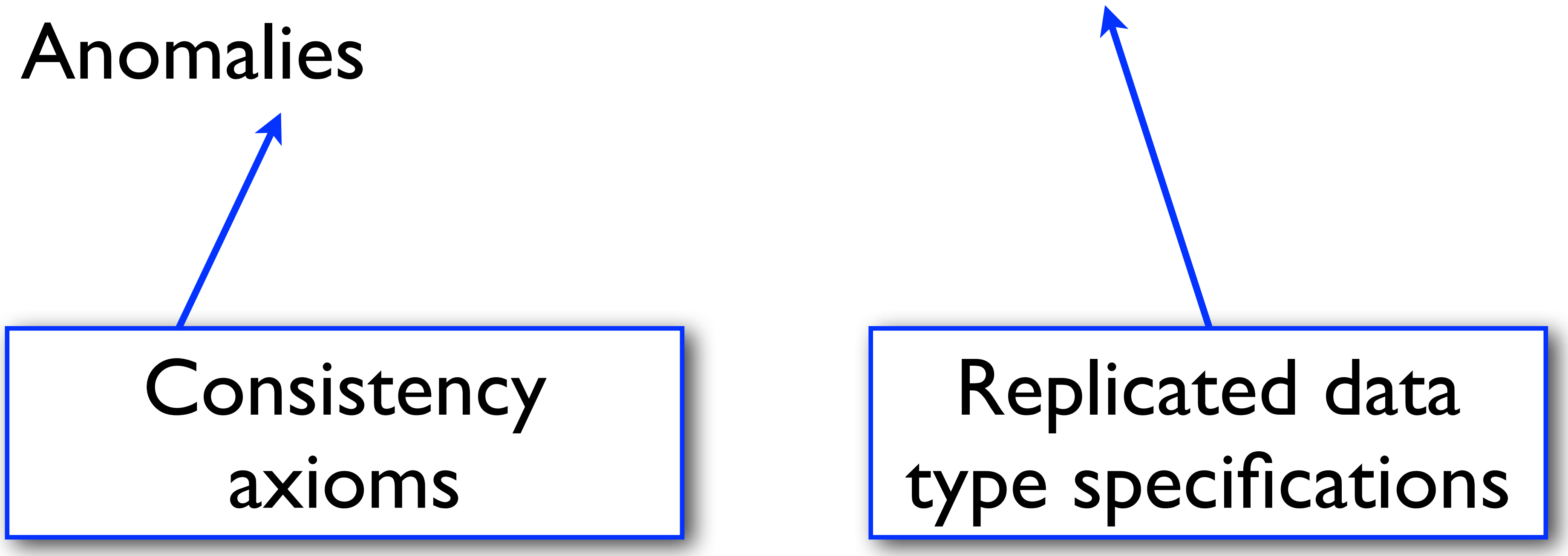
Results [POPL'14]

- Specification:
  - Conflict resolution ~ replicated data types
  - Anomalies

Replicated data type specifications

Results [POPL'14]

- Specification:
  - Conflict resolution ~ replicated data types
  - Anomalies



Consistency axioms

Replicated data type specifications

Sequential data type semantics

Strong consistency → operations are totally ordered:

set.add(*book*)

↓

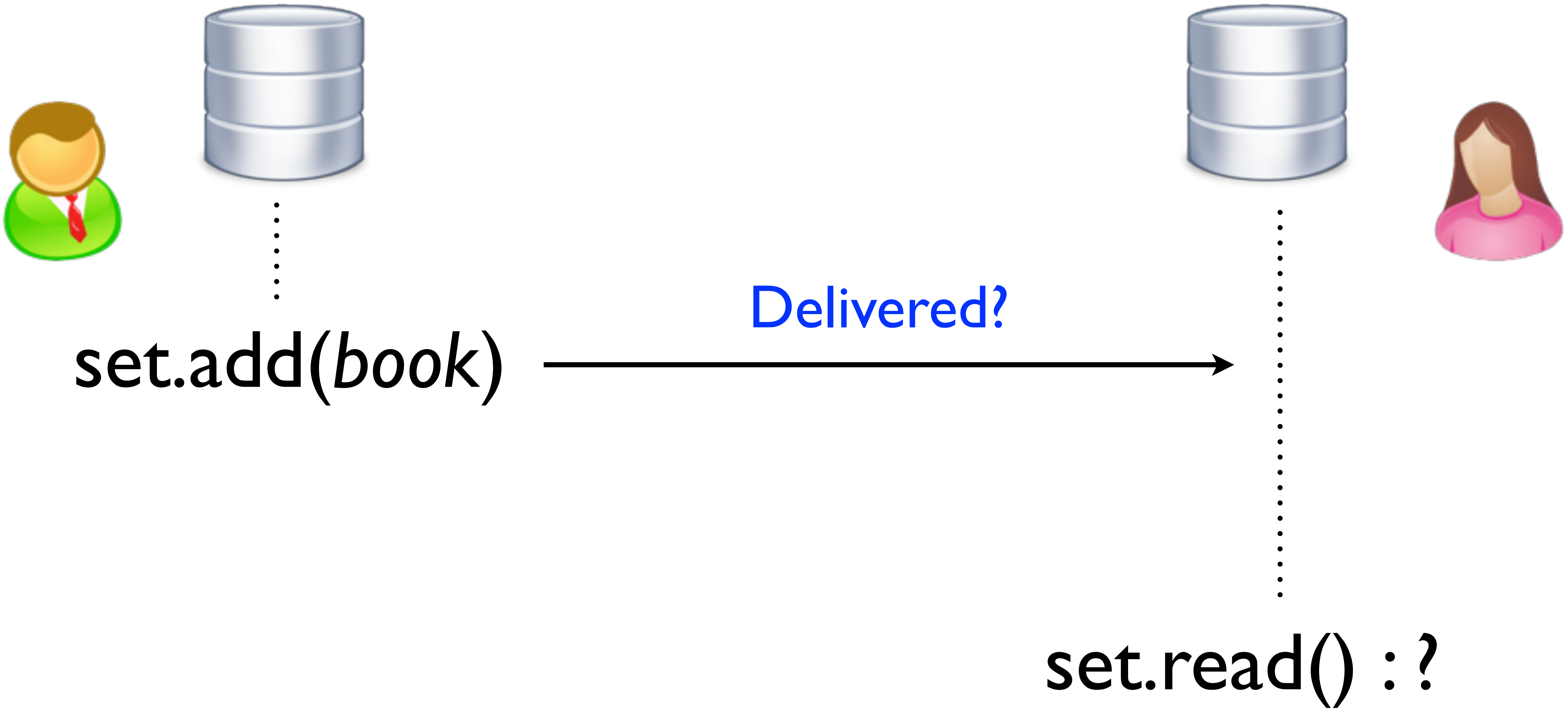
set.remove(*book*)

↓

set.read() : ∅

Compute the result by applying operations in sequence

Replicated data type semantics



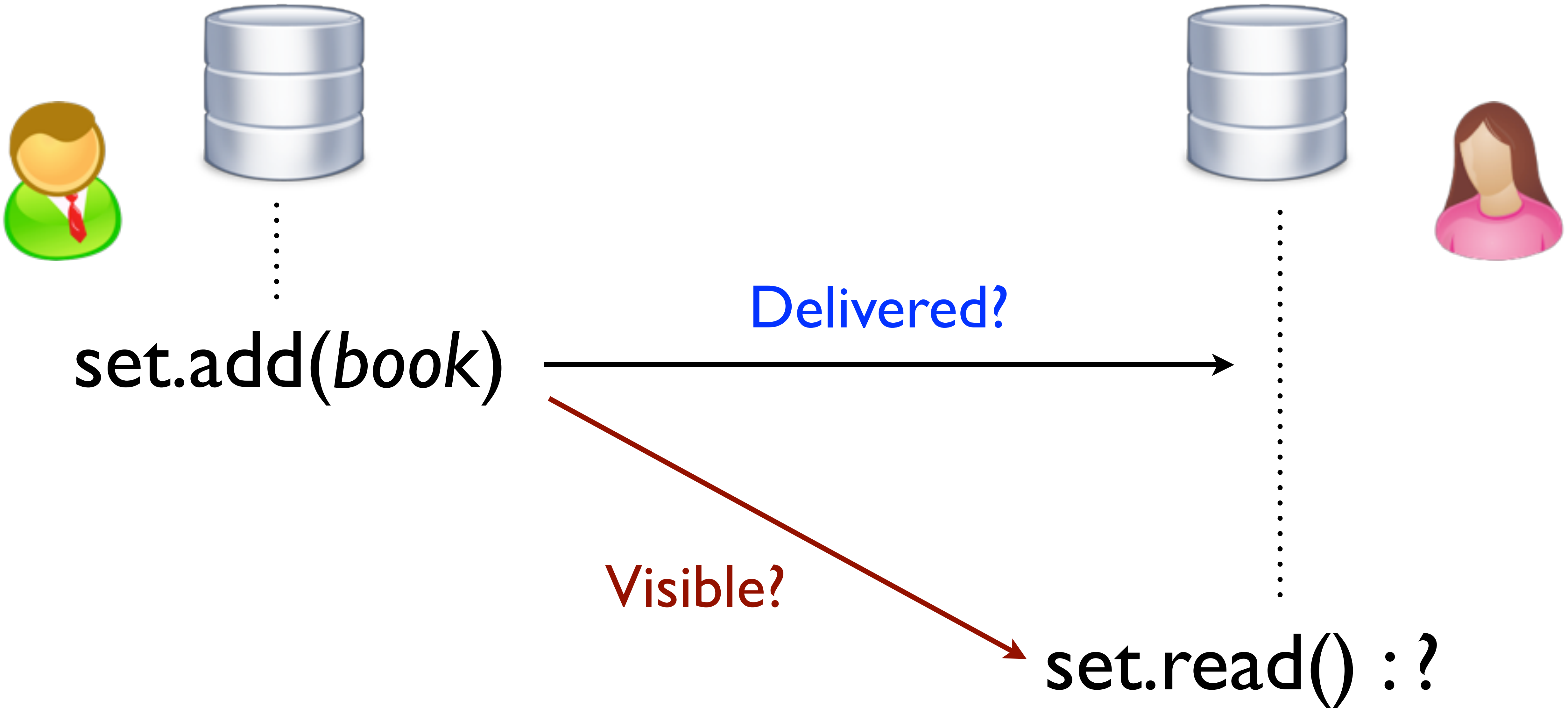
set.add(*book*)

Delivered?

set.read() : ?

Only updates that have been delivered to the replica performing the operation are important

Replicated data type semantics



set.add(*book*)

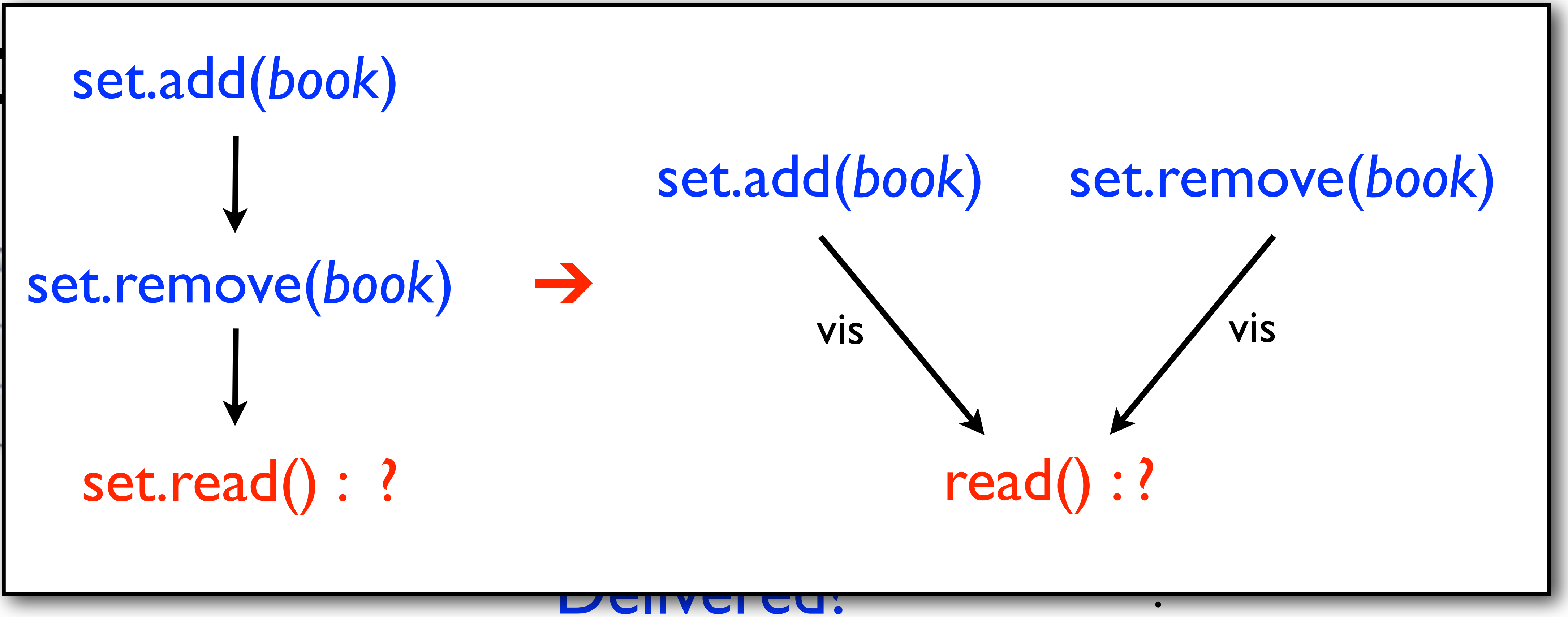
Delivered?

Visible?

set.read() : ?

Abstract by the visibility relation on operations (acyclic, ...)

Rep



set.add(*book*)

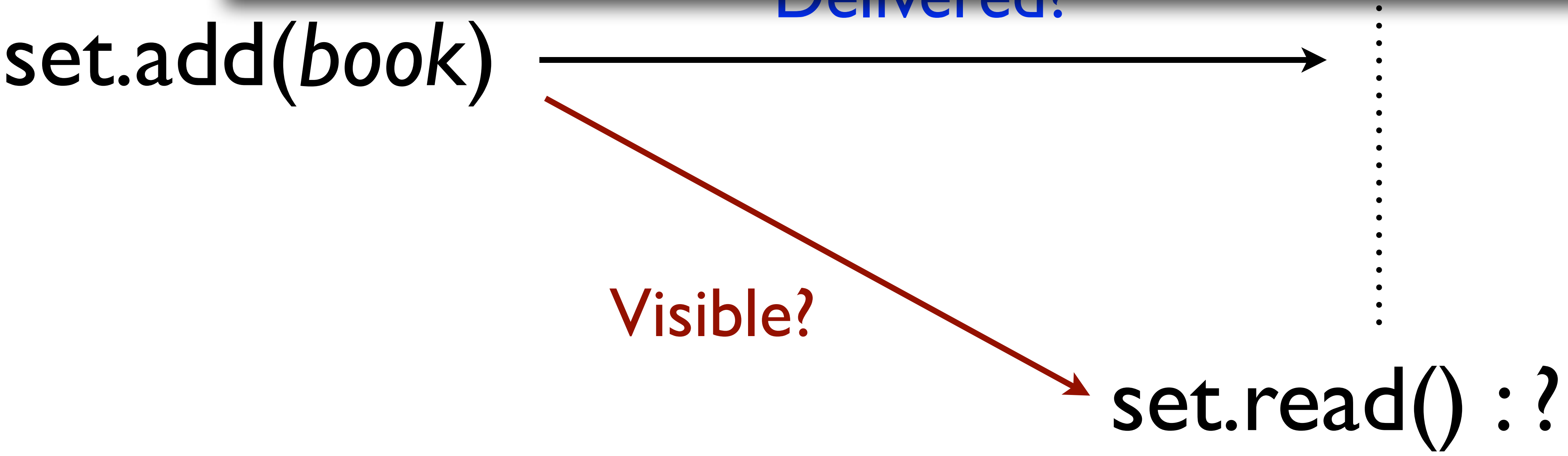
set.remove(*book*)

set.read() : ?

set.add(*book*)

set.remove(*book*)

read() : ?



Delivered?

set.add(*book*)

Visible?

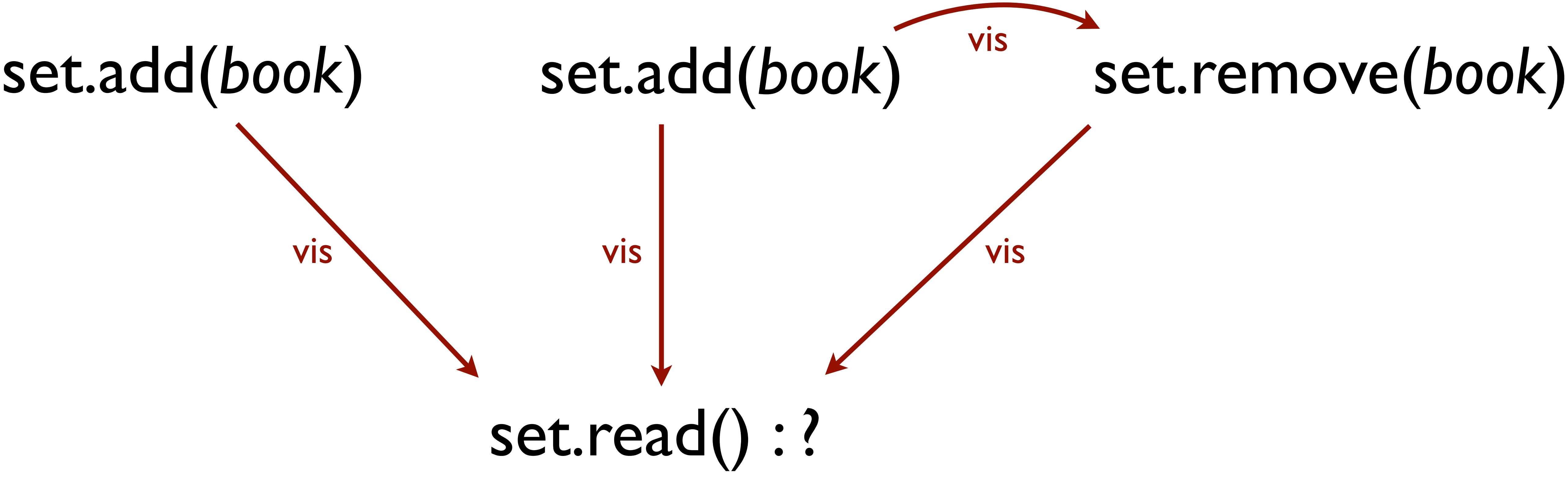
set.read() : ?

Abstract by the visibility relation on operations (acyclic, ...)

Replicated data type specification

F: context(op) → return value(op)

Context: all updates visible to the operation and the visibility relation between them + some other things



set.add(*book*)

set.add(*book*)

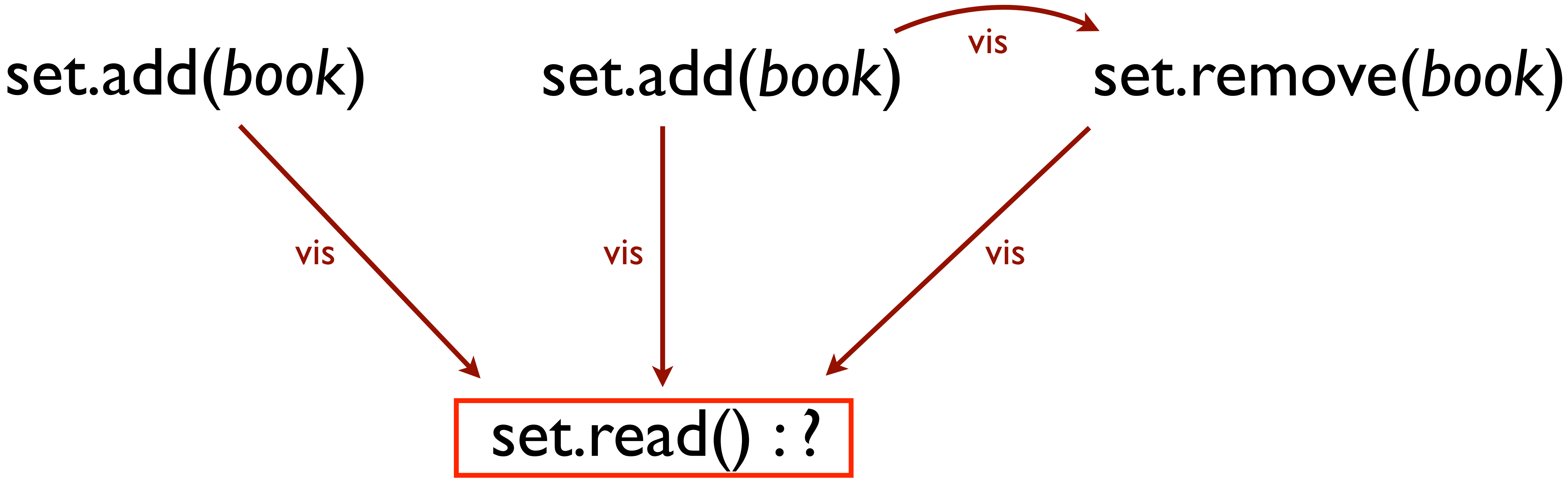
set.remove(*book*)

set.read() : ?

Replicated data type specification

F: context(op) → return value(op)

Context: all updates visible to the operation and the visibility relation between them + some other things



set.add(*book*)

set.add(*book*)

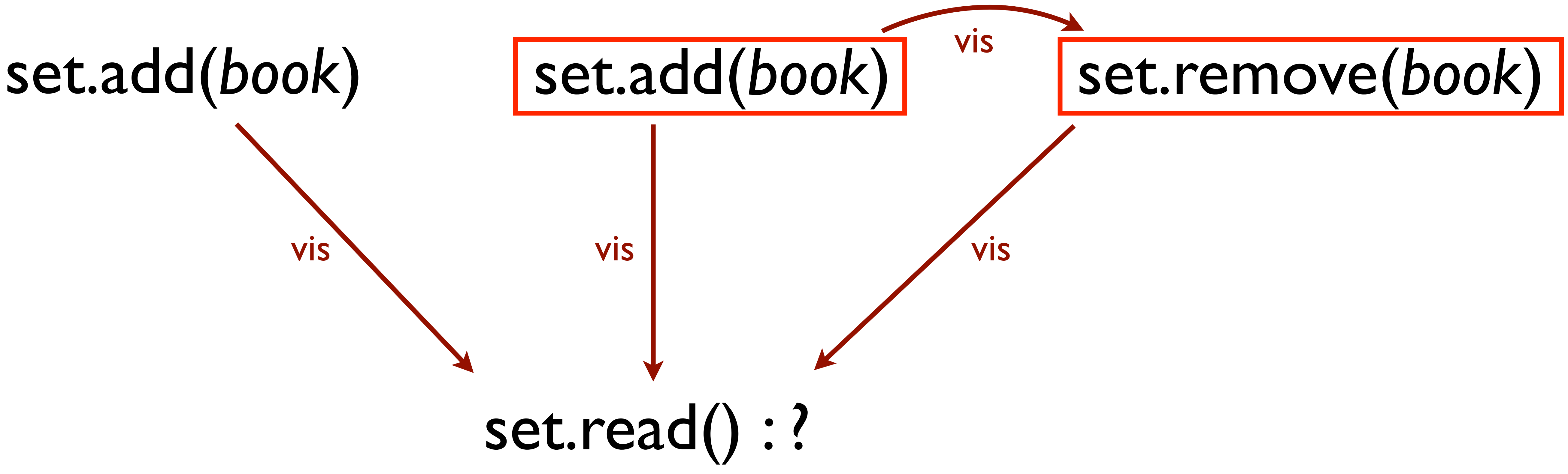
set.remove(*book*)

set.read() : ?

Replicated data type specification

F: context(op) → return value(op)

Context: all updates visible to the operation and the visibility relation between them + some other things



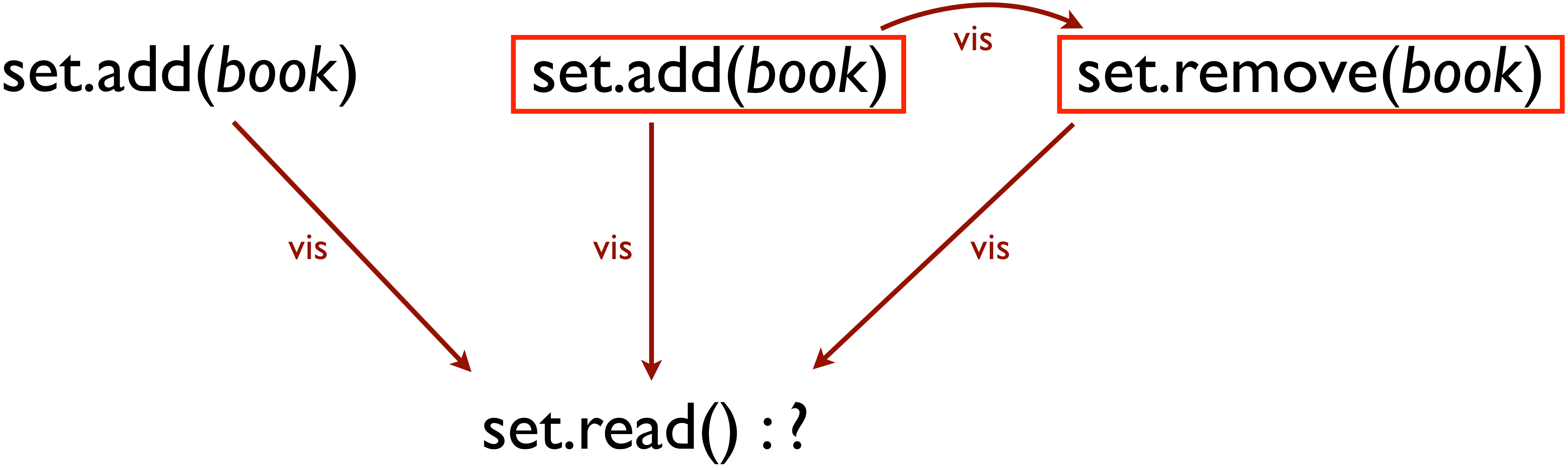
set.add(*book*)   set.add(*book*)   set.remove(*book*)

set.read() : ?

Add-wins set

F: context(op) → return value(op)

Context: all updates visible to the operation and the visibility relation between them + some other things



set.add(*book*)

set.add(*book*)

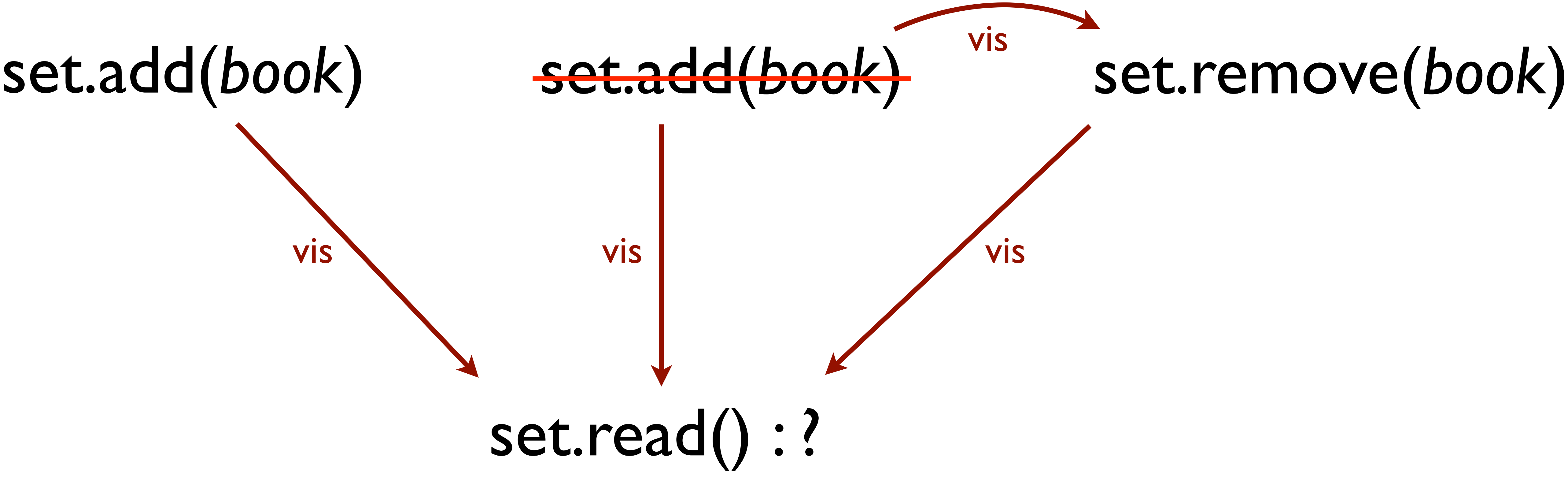
set.remove(*book*)

set.read() : ?

Add-wins set

F: context(op) → return value(op)

Context: all updates visible to the operation and the visibility relation between them + some other things



set.add(*book*)

~~set.add(*book*)~~

set.remove(*book*)

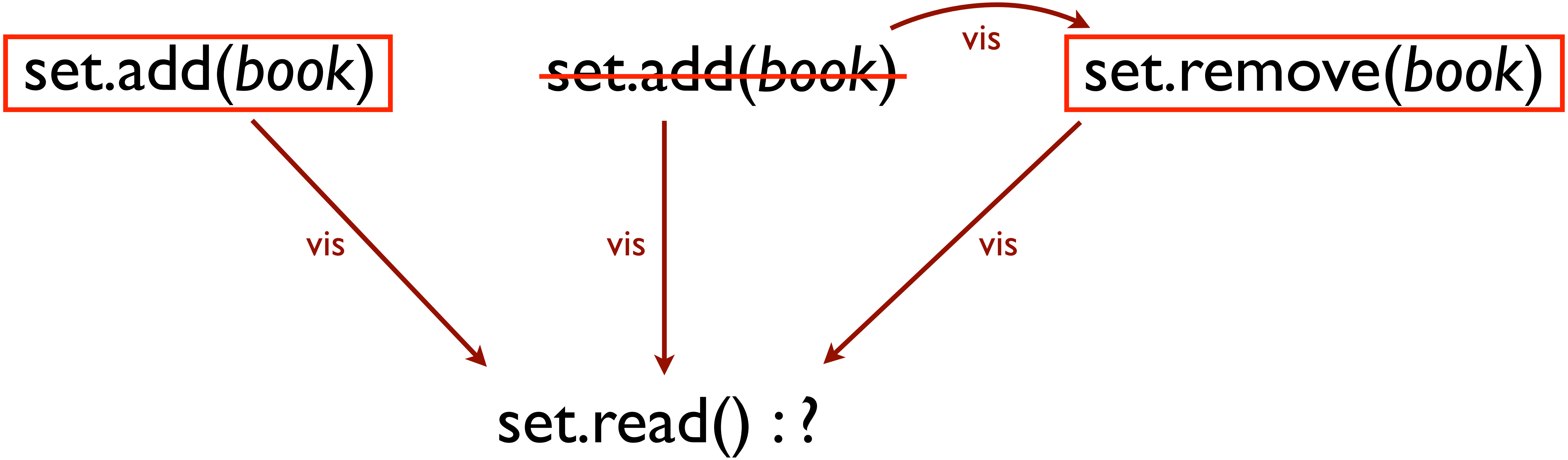
set.read() : ?

If you saw it, it's not a conflict

Add-wins set

F: context(op) → return value(op)

Context: all updates visible to the operation and the visibility relation between them + some other things



set.add(*book*)

~~set.add(*book*)~~

vis

set.remove(*book*)

vis

vis

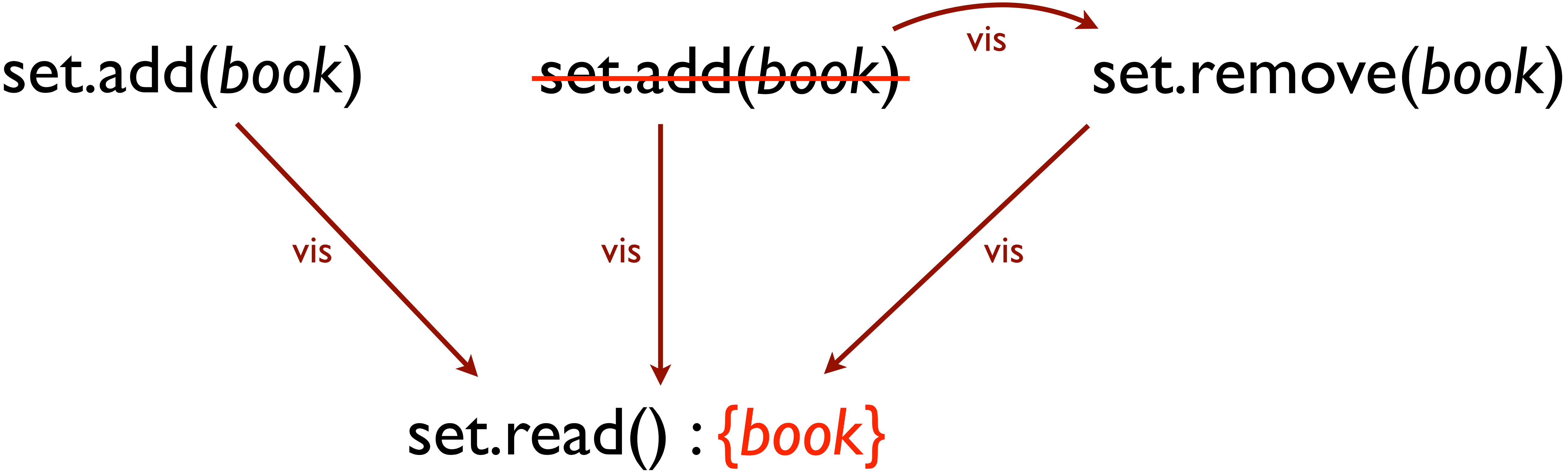
vis

set.read() : ?

Add-wins set

F: context(op) → return value(op)

Context: all updates visible to the operation and the visibility relation between them + some other things



set.add(*book*)

~~set.add(*book*)~~

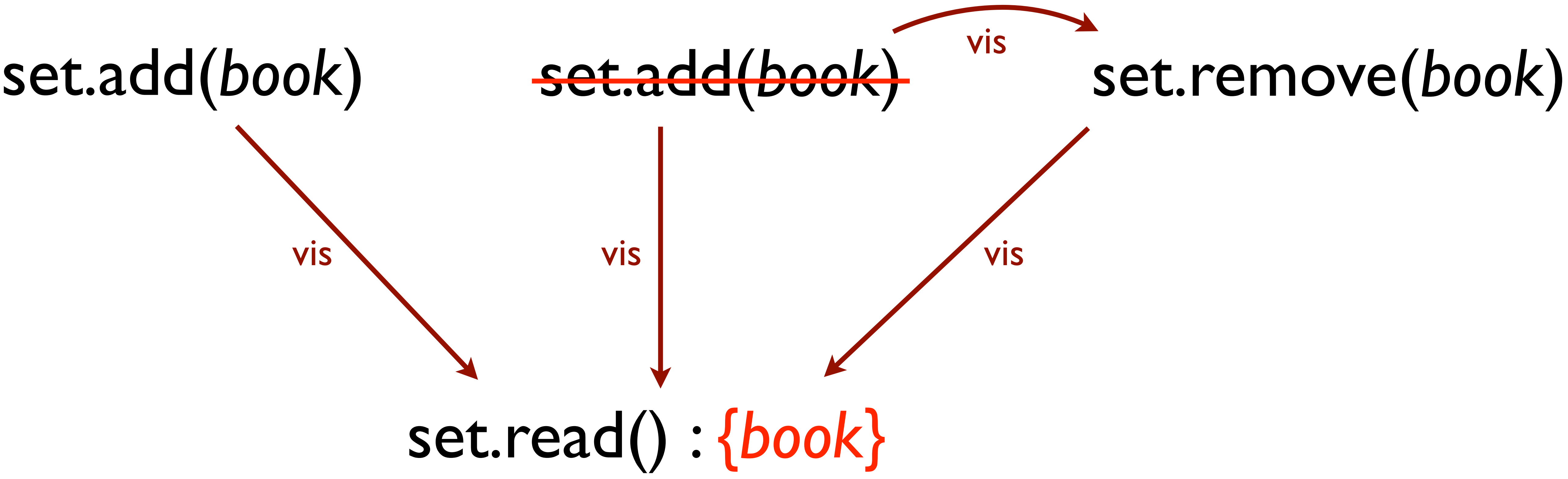
set.remove(*book*)

set.read() : {*book*}

Add-wins set

F: context(op) → return value(op)

Context: all updates visible to the operation and the visibility relation between them + some other things



set.add(*book*)

~~set.add(*book*)~~

set.remove(*book*)

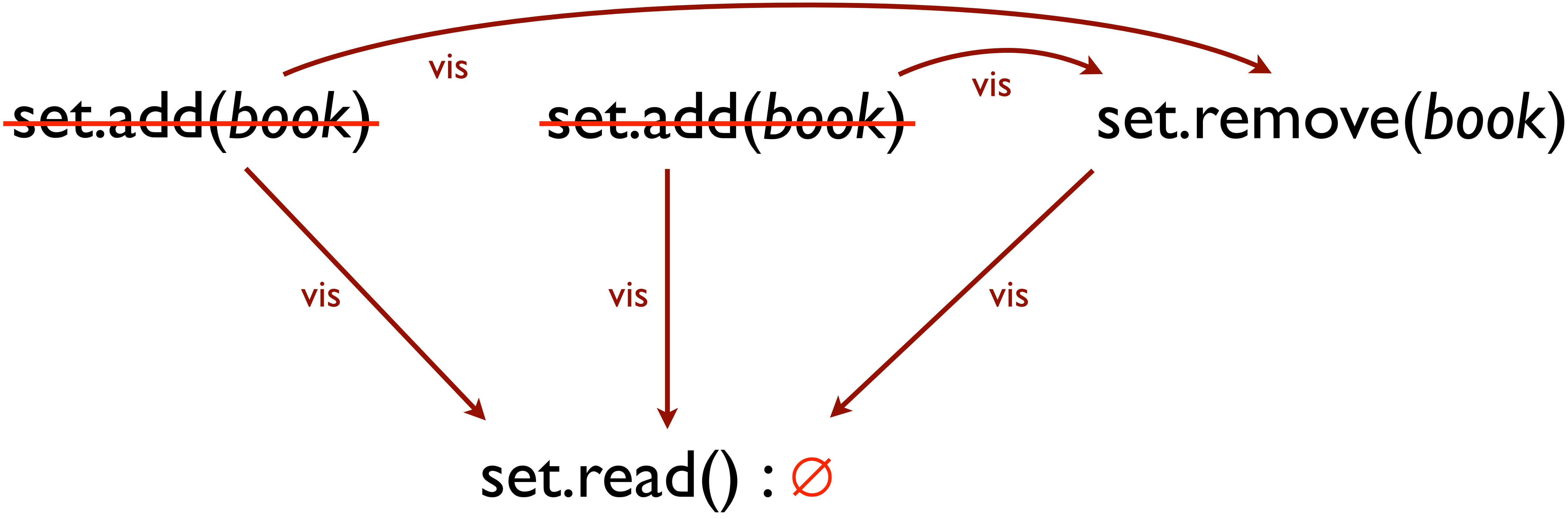
set.read() : {*book*}

F: cancel all adds seen by a remove

Add-wins set

F: context(op) → return value(op)

Context: all updates visible to the operation and the visibility relation between them + some other things



~~set.add(*book*)~~ ~~set.add(*book*)~~ set.remove(*book*)

set.read() : ∅

F: cancel all adds seen by a remove

Where does vis come from?

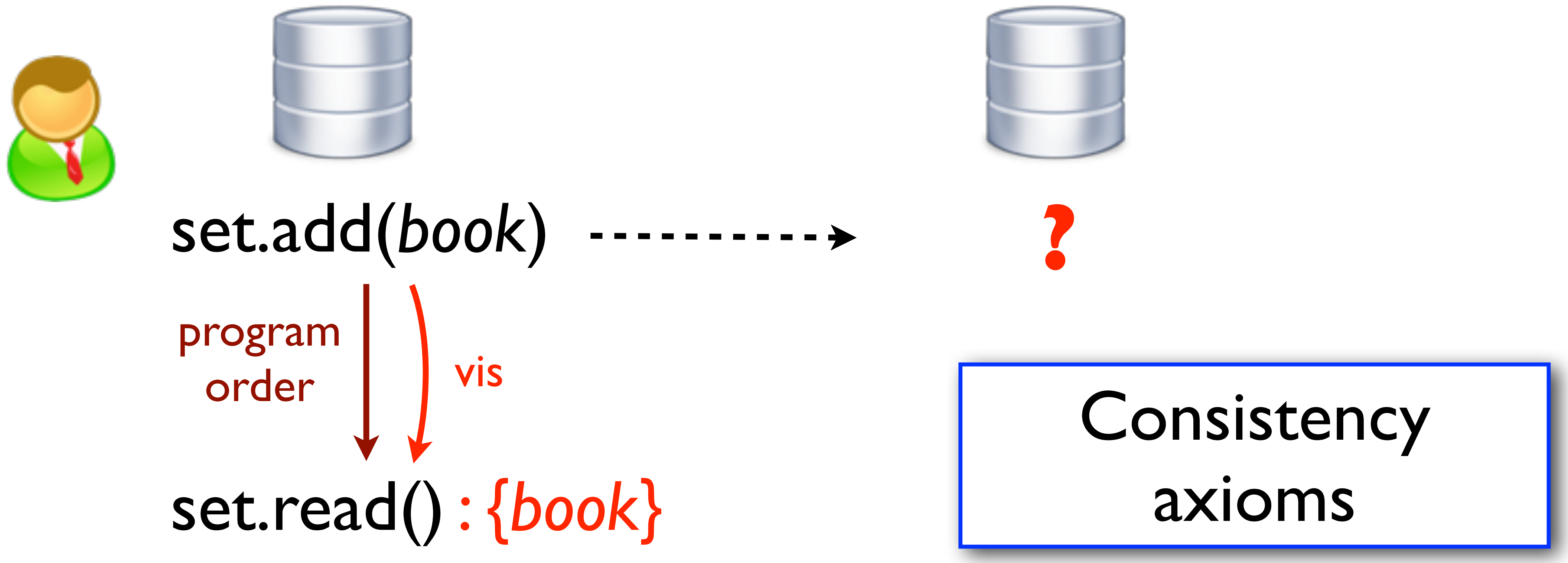
Almost arbitrary: little control over when updates are visible to other replicas

set.add(*book*)

?

Where does vis come from?

Almost arbitrary: little control over when updates are visible to other replicas



set.add(*book*) ?

program order

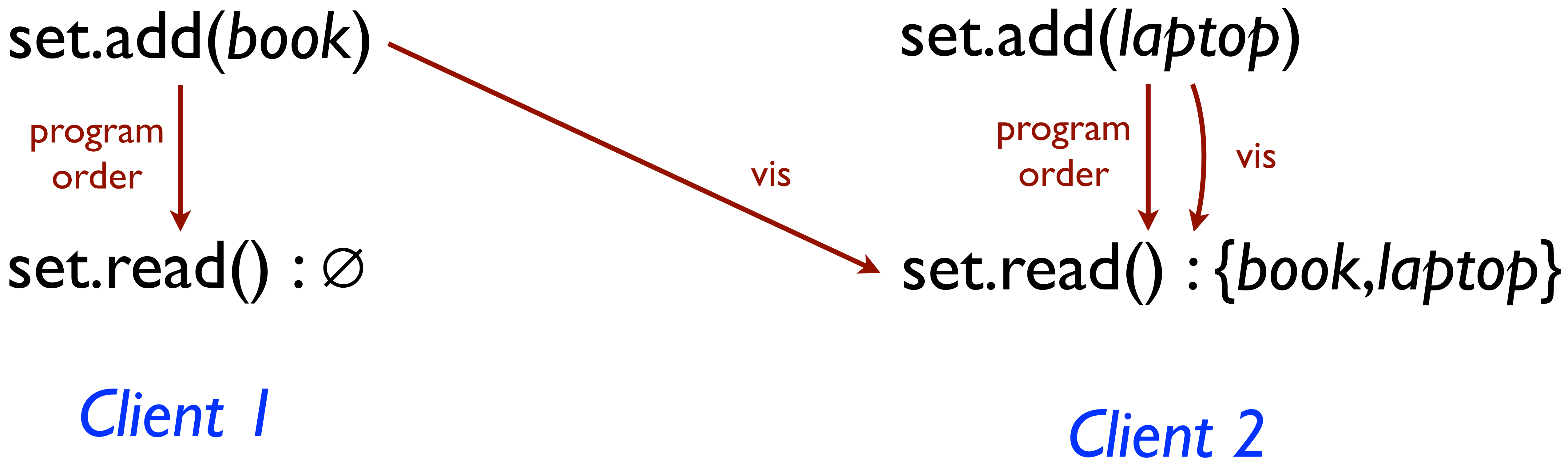
vis

set.read() : {*book*}

Consistency axioms

But may guarantee that they don't change unpredictably between operations = anomalies disallowed

Abstract executions: (E, po, vis)



set.add(*book*)

program order

set.read() : $\emptyset$

Client 1

vis

set.add(*laptop*)

program order

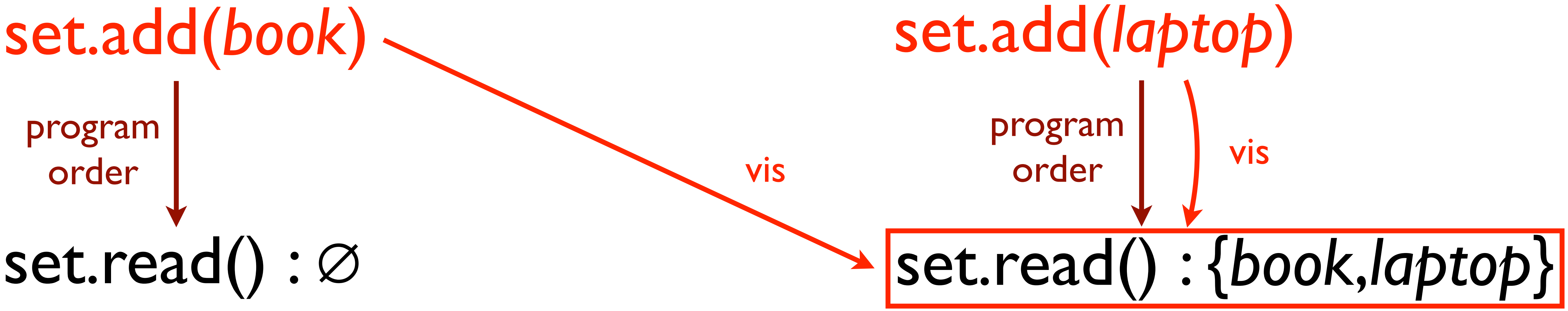
vis

set.read() : {*book*,*laptop*}

Client 2

- All operations in a database run, on all objects
- Operations grouped by clients and arranged in program order

Abstract executions: (E, po, vis)



set.add(*book*)

program order

set.read() : ∅

vis

set.add(*laptop*)

program order

vis

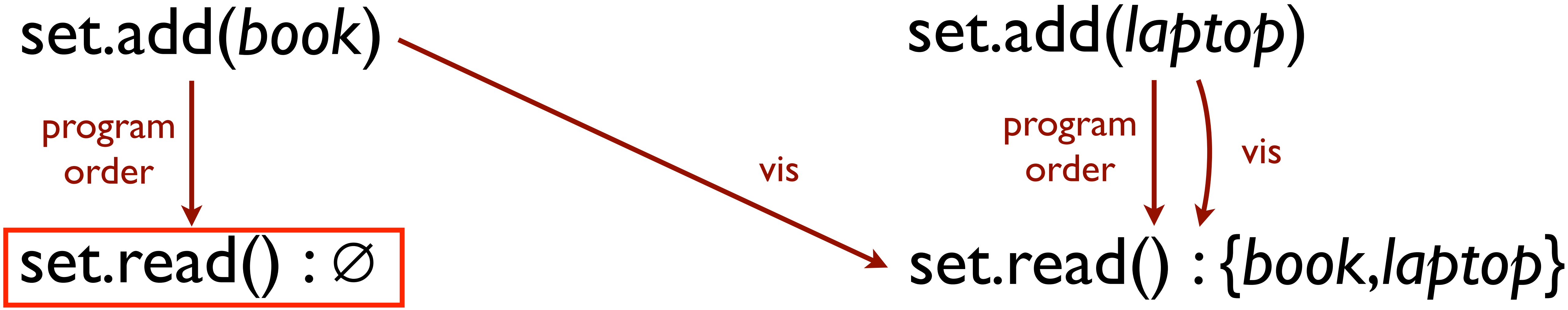
set.read() : {*book*,*laptop*}

Determines the context of every operation:

Context(op) = projection onto events visible to op

return value(op) = F(Context(op))

Abstract executions: (E, po, vis)



set.add(*book*)

program order

set.read() : ∅

vis

set.add(*laptop*)

program order

vis

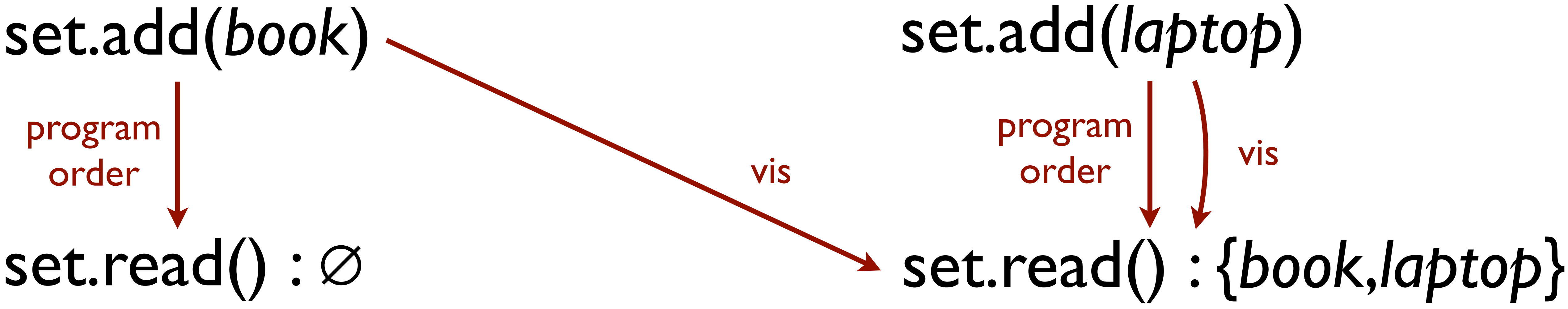
set.read() : {*book*,*laptop*}

Determines the context of every operation:

Context(op) = projection onto events visible to op

return value(op) = F(Context(op))

Consistency axioms



set.add(*book*)

set.add(*laptop*)

program order

vis

program order

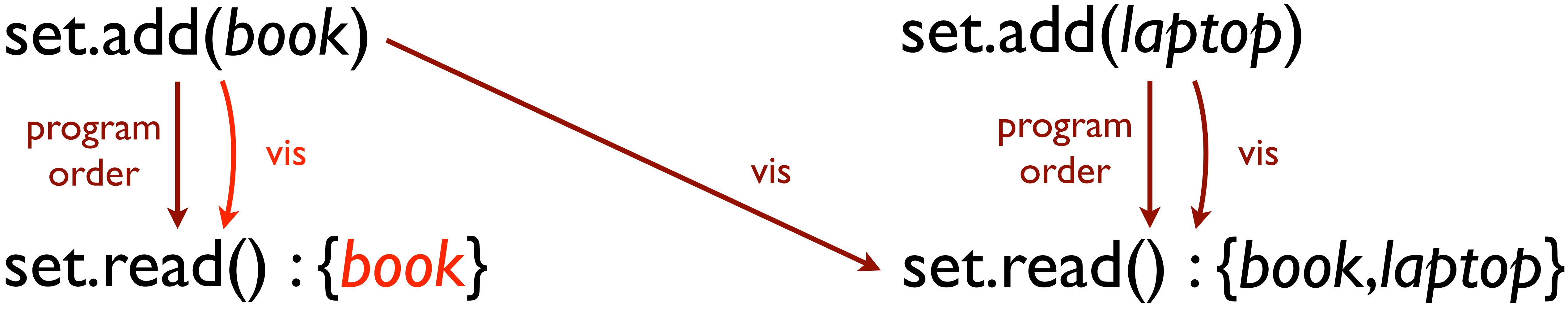
vis

set.read() : ∅

set.read() : {*book*,*laptop*}

- Consistency axioms disallow anomalies by constraining executions
- Read Your Writes: po ∩ same-object ⊆ vis
- Principle: strengthen consistency by mandating that more edges be included into vis

Consistency axioms


set.add(*book*)

program order

vis

set.read() : {*book*}

vis

set.add(*laptop*)

program order

vis

set.read() : {*book*,*laptop*}

- Consistency axioms disallow anomalies by constraining executions
- Read Your Writes: $\text{po} \cap \text{same-object} \subseteq \text{vis}$
- Principle: strengthen consistency by mandating that more edges be included into vis

Basic eventual consistency

Session guarantees

Per-object causal consistency

Causal consistency

Strong consistency

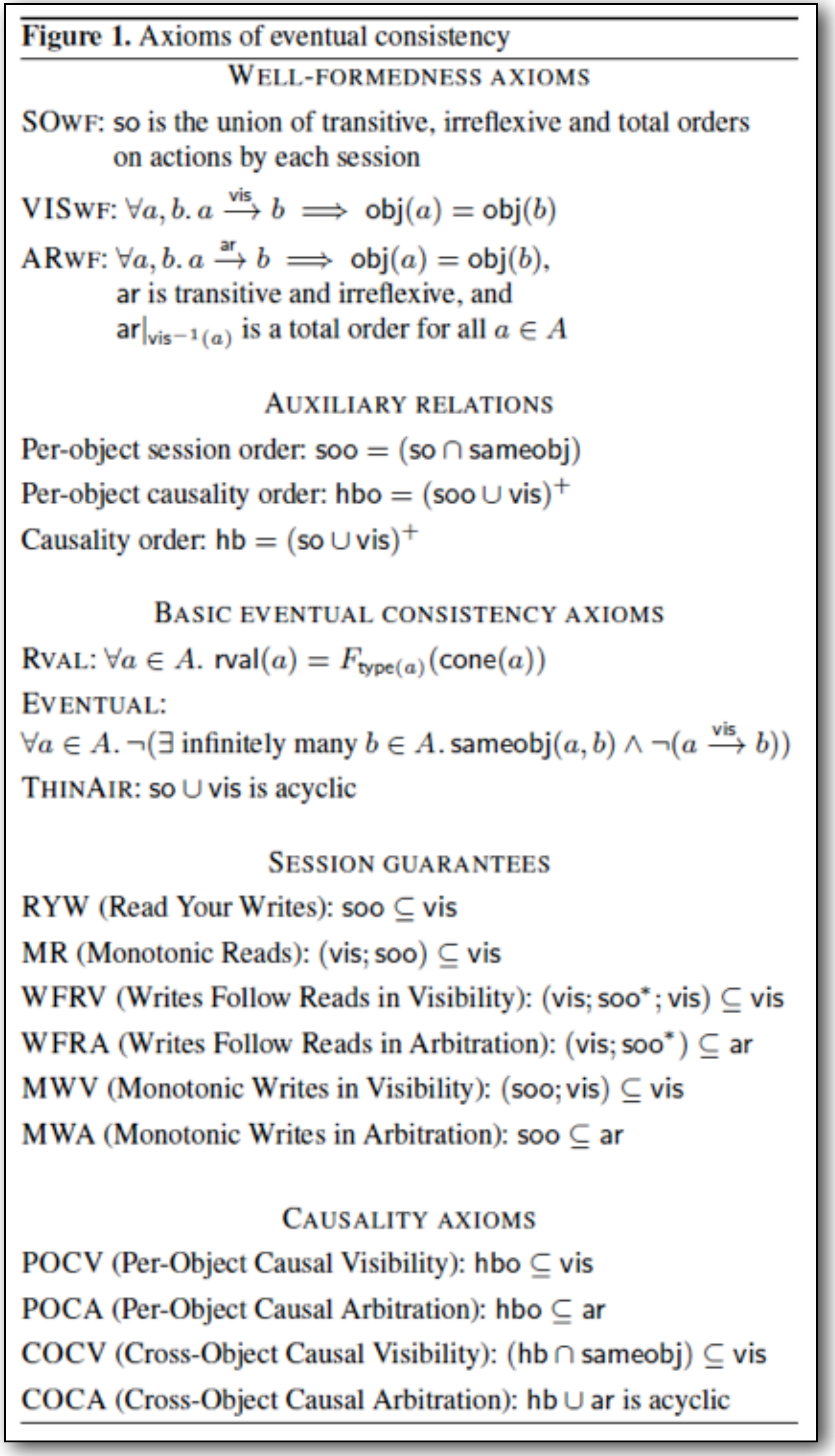


Figure 1. Axioms of eventual consistency

WELL-FORMEDNESS AXIOMS

SOWF: so is the union of transitive, irreflexive and total orders on actions by each session

VISWF: $\forall a, b.\, a \xrightarrow{\mathsf{vis}} b \implies \mathsf{obj}(a) = \mathsf{obj}(b)$

ARWF: $\forall a, b.\, a \xrightarrow{\mathsf{ar}} b \implies \mathsf{obj}(a) = \mathsf{obj}(b)$,
ar is transitive and irreflexive, and
$\mathsf{ar}|_{\mathsf{vis}^{-1}(a)}$ is a total order for all $a \in A$

AUXILIARY RELATIONS

Per-object session order: $\mathsf{soo} = (\mathsf{so} \cap \mathsf{sameobj})$

Per-object causality order: $\mathsf{hbo} = (\mathsf{soo} \cup \mathsf{vis})^{+}$

Causality order: $\mathsf{hb} = (\mathsf{so} \cup \mathsf{vis})^{+}$

BASIC EVENTUAL CONSISTENCY AXIOMS

RVAL: $\forall a \in A.\ \mathsf{rval}(a) = F_{\mathsf{type}(a)}(\mathsf{cone}(a))$

EVENTUAL:
$\forall a \in A.\ \neg(\exists \text{ infinitely many } b \in A.\ \mathsf{sameobj}(a, b) \wedge \neg(a \xrightarrow{\mathsf{vis}} b))$

THINAIR: $\mathsf{so} \cup \mathsf{vis}$ is acyclic

SESSION GUARANTEES

RYW (Read Your Writes): $\mathsf{soo} \subseteq \mathsf{vis}$

MR (Monotonic Reads): $(\mathsf{vis}; \mathsf{soo}) \subseteq \mathsf{vis}$

WFRV (Writes Follow Reads in Visibility): $(\mathsf{vis}; \mathsf{soo}^{*}; \mathsf{vis}) \subseteq \mathsf{vis}$

WFRA (Writes Follow Reads in Arbitration): $(\mathsf{vis}; \mathsf{soo}^{*}) \subseteq \mathsf{ar}$

MWV (Monotonic Writes in Visibility): $(\mathsf{soo}; \mathsf{vis}) \subseteq \mathsf{vis}$

MWA (Monotonic Writes in Arbitration): $\mathsf{soo} \subseteq \mathsf{ar}$

CAUSALITY AXIOMS

POCV (Per-Object Causal Visibility): $\mathsf{hbo} \subseteq \mathsf{vis}$

POCA (Per-Object Causal Arbitration): $\mathsf{hbo} \subseteq \mathsf{ar}$

COCV (Cross-Object Causal Visibility): $(\mathsf{hb} \cap \mathsf{sameobj}) \subseteq \mathsf{vis}$

COCA (Cross-Object Causal Arbitration): $\mathsf{hb} \cup \mathsf{ar}$ is acyclic

- Our specifications similar to weak memory model definitions
- Eventual consistency axioms for registers ≈ C/C++ memory model

Basic eventu

Session guarantees

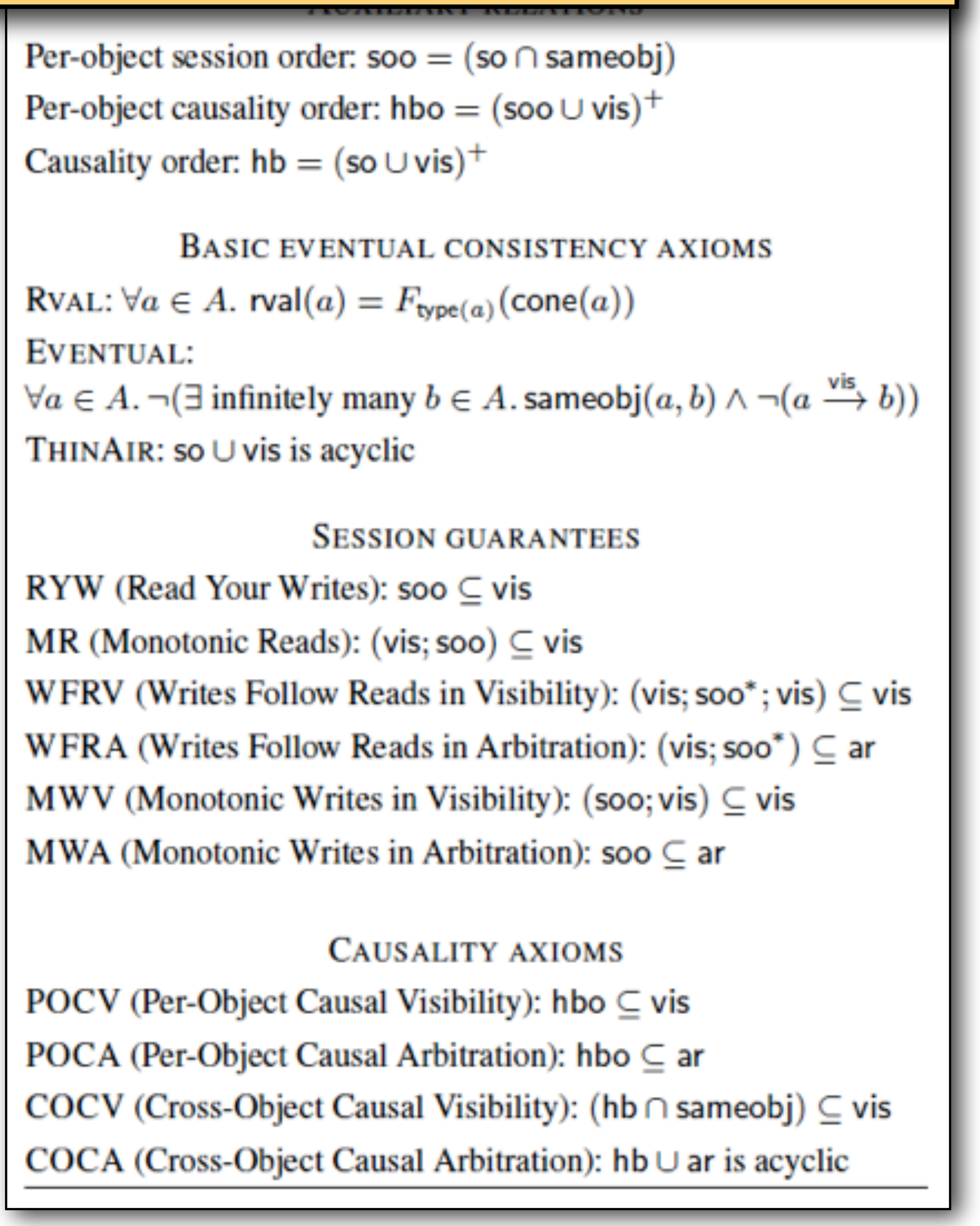
Per-object causal consistency

≈ 2011 C/C++ relaxed

Causal consistency

≈ 2011 C/C++ release/acquire

Strong consistency



Per-object session order: $\mathsf{soo} = (\mathsf{so} \cap \mathsf{sameobj})$

Per-object causality order: $\mathsf{hbo} = (\mathsf{soo} \cup \mathsf{vis})^+$

Causality order: $\mathsf{hb} = (\mathsf{so} \cup \mathsf{vis})^+$

BASIC EVENTUAL CONSISTENCY AXIOMS

RVAL: $\forall a \in A.\ \mathsf{rval}(a) = F_{\mathsf{type}(a)}(\mathsf{cone}(a))$

EVENTUAL:

$\forall a \in A.\ \neg(\exists \text{ infinitely many } b \in A.\ \mathsf{sameobj}(a, b) \wedge \neg(a \xrightarrow{\mathsf{vis}} b))$

THINAIR: $\mathsf{so} \cup \mathsf{vis}$ is acyclic

SESSION GUARANTEES

RYW (Read Your Writes): $\mathsf{soo} \subseteq \mathsf{vis}$

MR (Monotonic Reads): $(\mathsf{vis}; \mathsf{soo}) \subseteq \mathsf{vis}$

WFRV (Writes Follow Reads in Visibility): $(\mathsf{vis}; \mathsf{soo}^*; \mathsf{vis}) \subseteq \mathsf{vis}$

WFRA (Writes Follow Reads in Arbitration): $(\mathsf{vis}; \mathsf{soo}^*) \subseteq \mathsf{ar}$

MWV (Monotonic Writes in Visibility): $(\mathsf{soo}; \mathsf{vis}) \subseteq \mathsf{vis}$

MWA (Monotonic Writes in Arbitration): $\mathsf{soo} \subseteq \mathsf{ar}$

CAUSALITY AXIOMS

POCV (Per-Object Causal Visibility): $\mathsf{hbo} \subseteq \mathsf{vis}$

POCA (Per-Object Causal Arbitration): $\mathsf{hbo} \subseteq \mathsf{ar}$

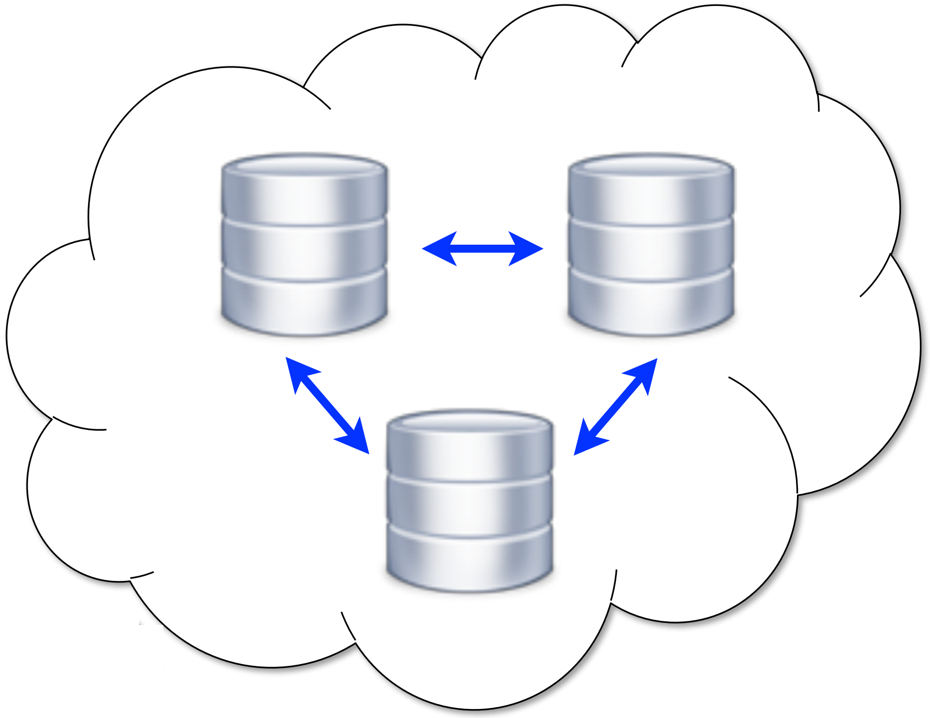
COCV (Cross-Object Causal Visibility): $(\mathsf{hb} \cap \mathsf{sameobj}) \subseteq \mathsf{vis}$

COCA (Cross-Object Causal Arbitration): $\mathsf{hb} \cup \mathsf{ar}$ is acyclic

Specification summary

Conflict resolution policies → Data type spec

Anomalies → Consistency axioms

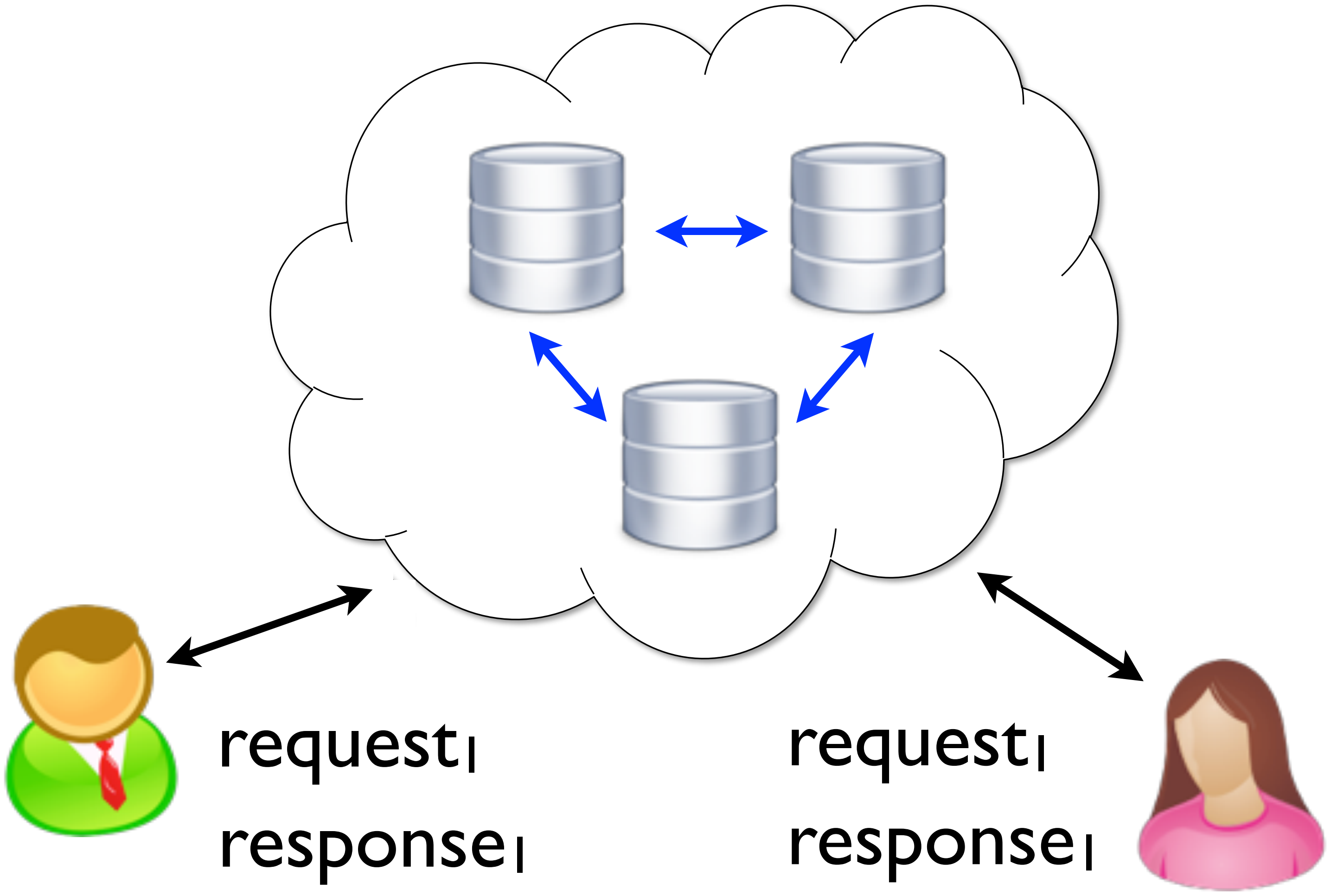


→ (E, po, vis)

Specification summary

Conflict resolution policies ➔ Data type spec

Anomalies ➔ Consistency axioms



➔ (E, po, vis)

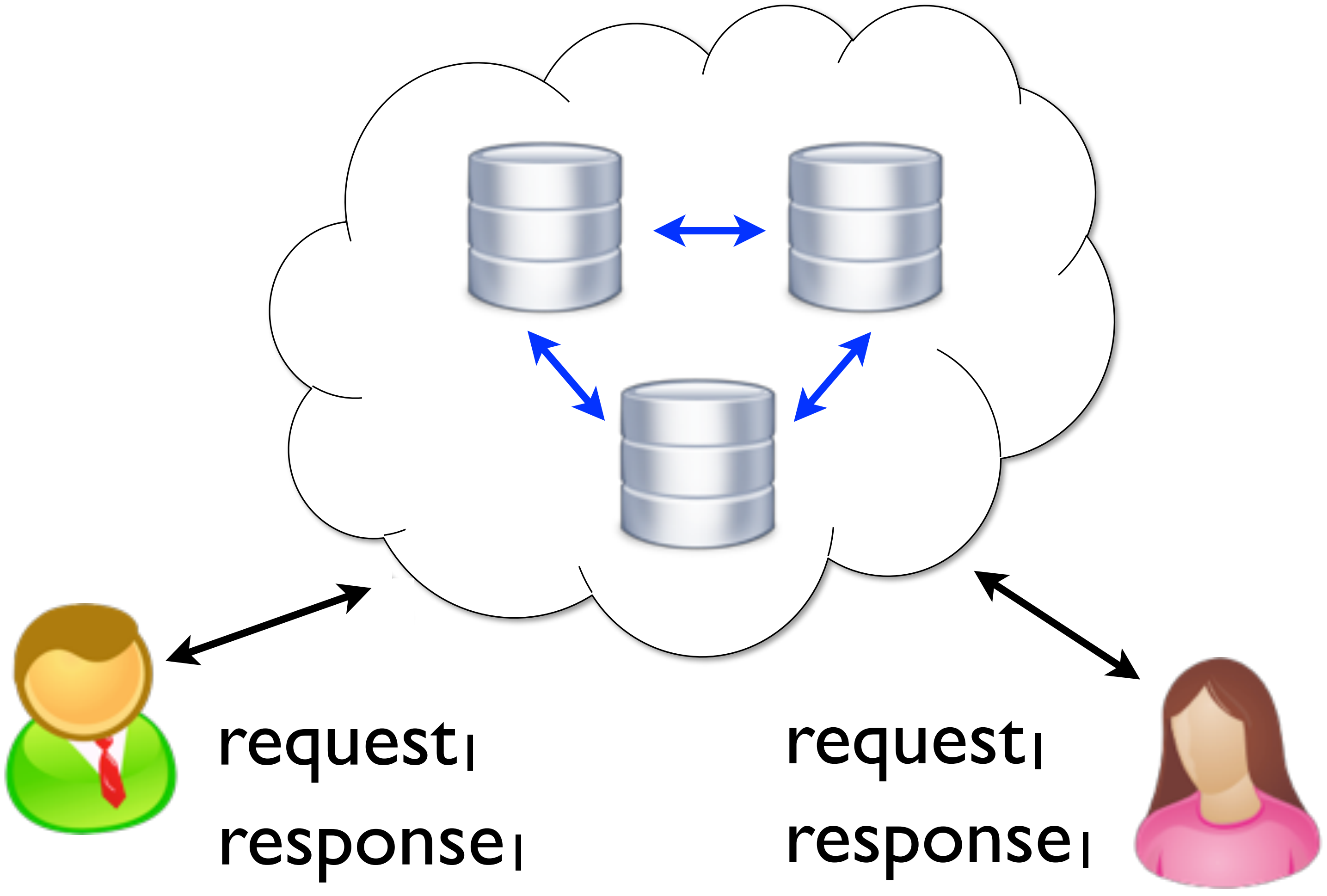
request$_1$
response$_1$
request$_2$
response$_2$
…

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

Specification summary

Conflict resolution policies → Data type spec

Anomalies → Consistency axioms



→ (E, po, vis)

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

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

Events (E, po) allowed iff ∃ execution (E, po, vis) satisfying data type specs and axioms

Specification summary

Conflict resolution policies ➔ Data type spec

Anomalies ➔ Consistency axioms

Quick & dirty proof of correspondence with algorithms used in systems [TR]

Specification summary

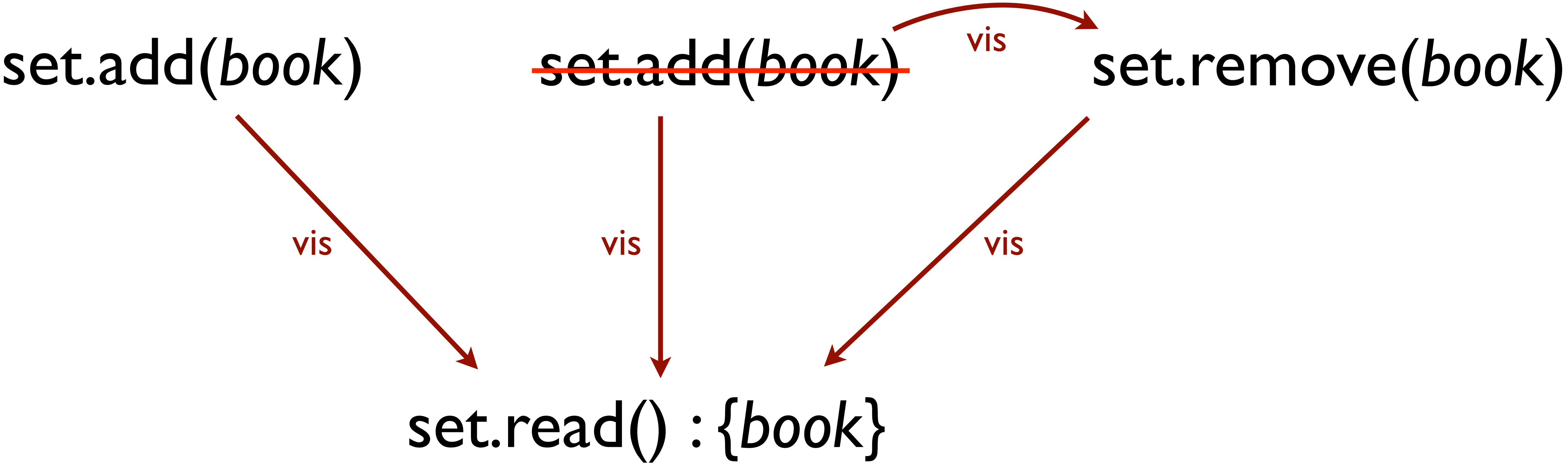
Conflict resolution policies → Data type spec

Anomalies → Consistency axioms

Quick & dirty proof of correspondence with algorithms used in systems [TR]

Verifying data type implementations

Naive add-wins set implementation



set.add(*book*)

~~set.add(*book*)~~

vis

set.remove(*book*)

vis

vis

vis

set.read() : {*book*}

Implementation challenge: remove behaves differently wrt different adds of the same element

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

set.add(*book*)

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

- Each add creates a new element instance: (element, unique instance id)

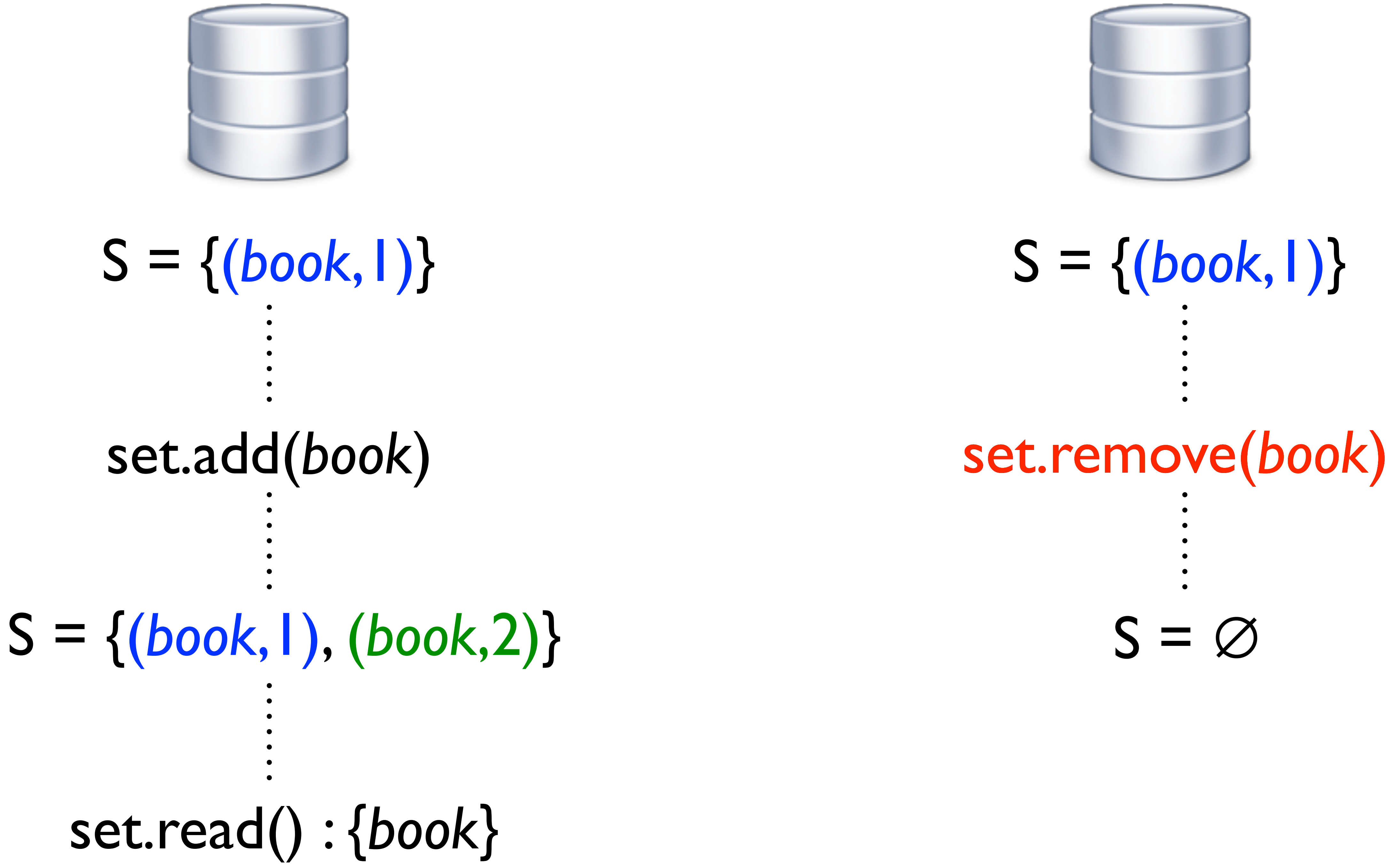
S = {(*book*,1)}

set.add(*book*)

S = {(*book*,1), (*book*,2)}

set.read() : {*book*}

- Each add creates a new element instance: (element, unique instance id)
- Instance ids ignored when reading the set



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

set.add(*book*)

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

set.read() : {*book*}

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

set.remove(*book*)

$S = \varnothing$

- Remove should remove all currently present instances of *book* from S

$S = \{(book, 1)\}, T = \emptyset$

set.add($book$)

$S = \{(book, 1), (book, 2)\}, T = \emptyset$

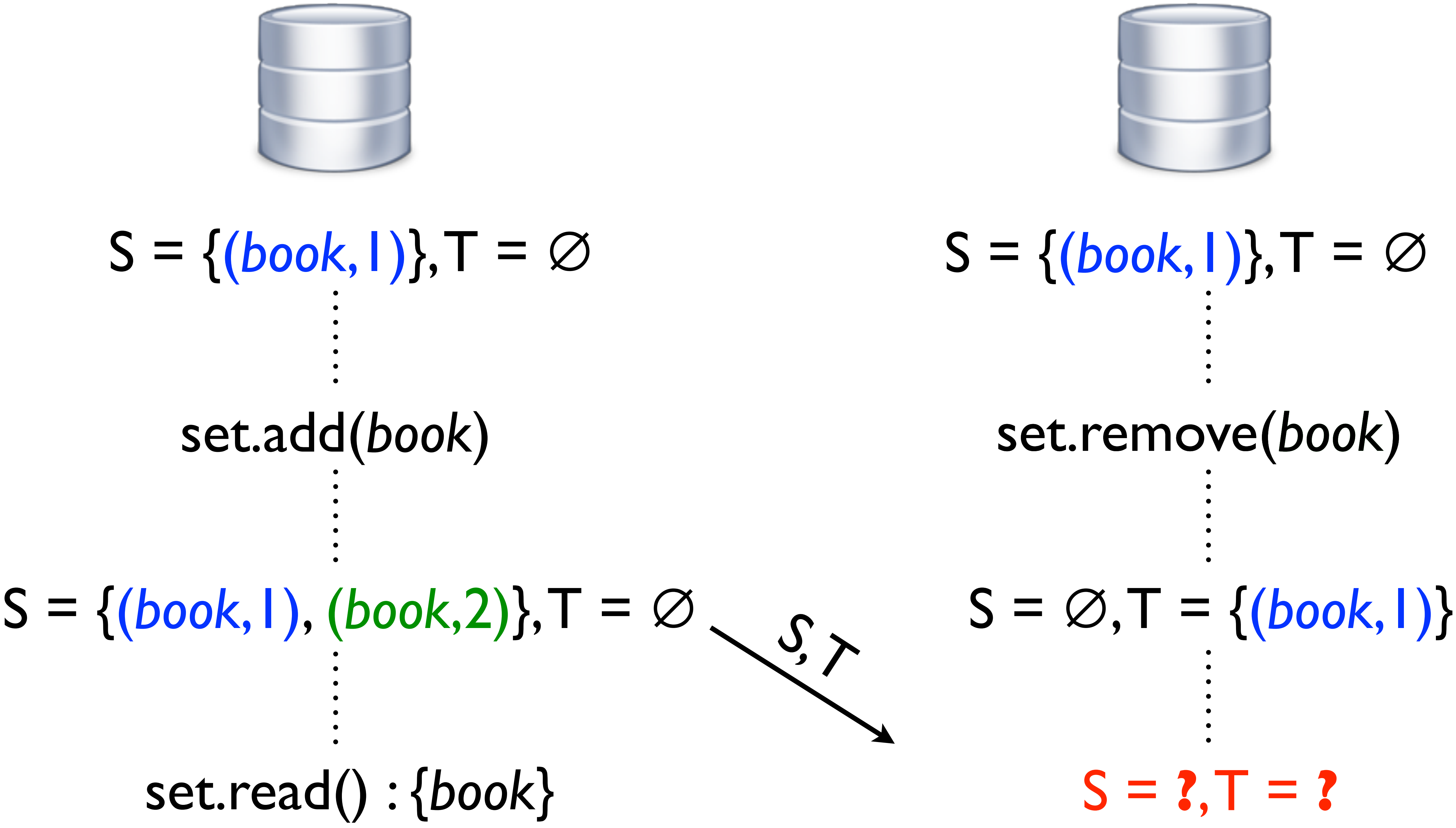
set.read() : $\{book\}$

$S = \{(book, 1)\}, T = \emptyset$

set.remove($book$)

$S = \emptyset, \boxed{T = \{(book, 1)\}}$

- But maintain the set of tombstones T: element instances removed
- Remove moves all instances of *book* in S to T



$S = \{(book, 1)\}, T = \emptyset$

set.add(*book*)

$S = \{(book, 1), (book, 2)\}, T = \emptyset$

set.read() : {*book*}

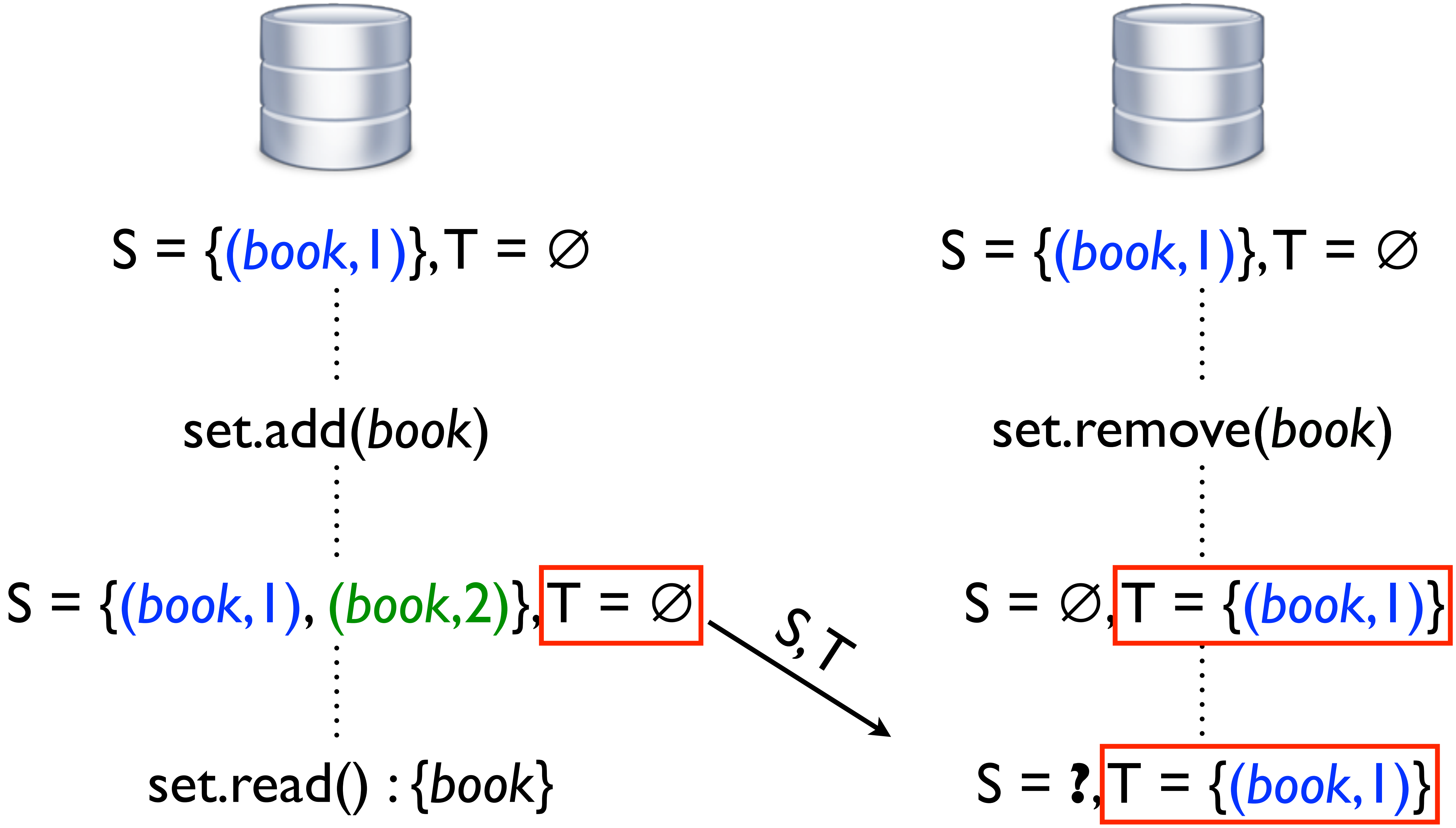
$S = \{(book, 1)\}, T = \emptyset$

set.remove(*book*)

$S = \emptyset, T = \{(book, 1)\}$

$S = ?, T = ?$

State-based implementation:
sends its state snapshot to other replicas



$S = \{(book, 1)\}, T = \emptyset$

set.add(*book*)

$S = \{(book, 1), (book, 2)\}, \boxed{T = \emptyset}$

set.read() : $\{book\}$

S, T

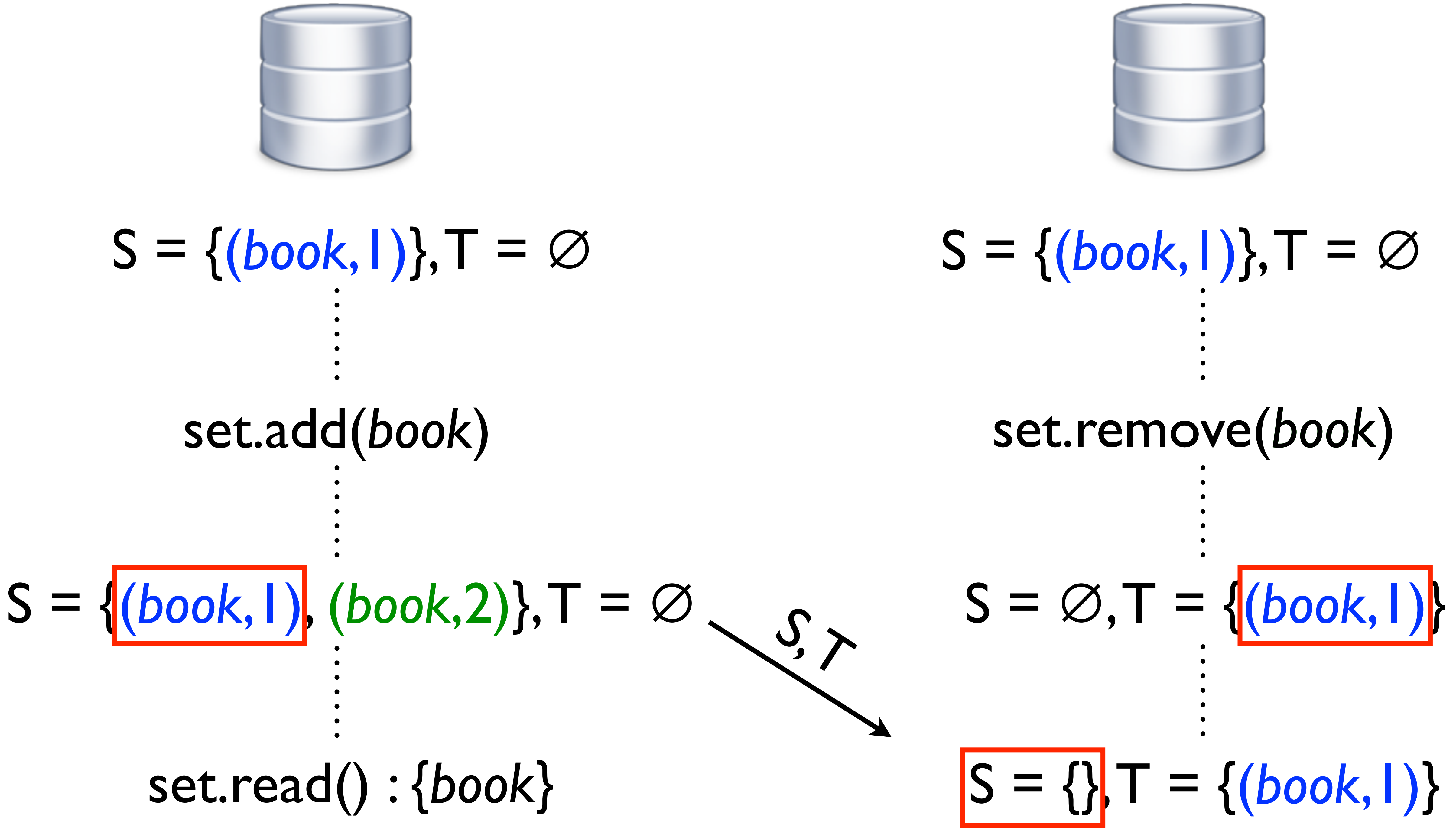
$S = \{(book, 1)\}, T = \emptyset$

set.remove(*book*)

$S = \emptyset, \boxed{T = \{(book, 1)\}}$

$S = \textbf{?}, \boxed{T = \{(book, 1)\}}$

State-based implementation:
sends its state snapshot to other replicas



$S = \{(\mathit{book}, 1)\}, T = \varnothing$

set.add(*book*)

$S = \{\boxed{(\mathit{book}, 1)}, (\mathit{book}, 2)\}, T = \varnothing$

set.read() : $\{\mathit{book}\}$

S, T

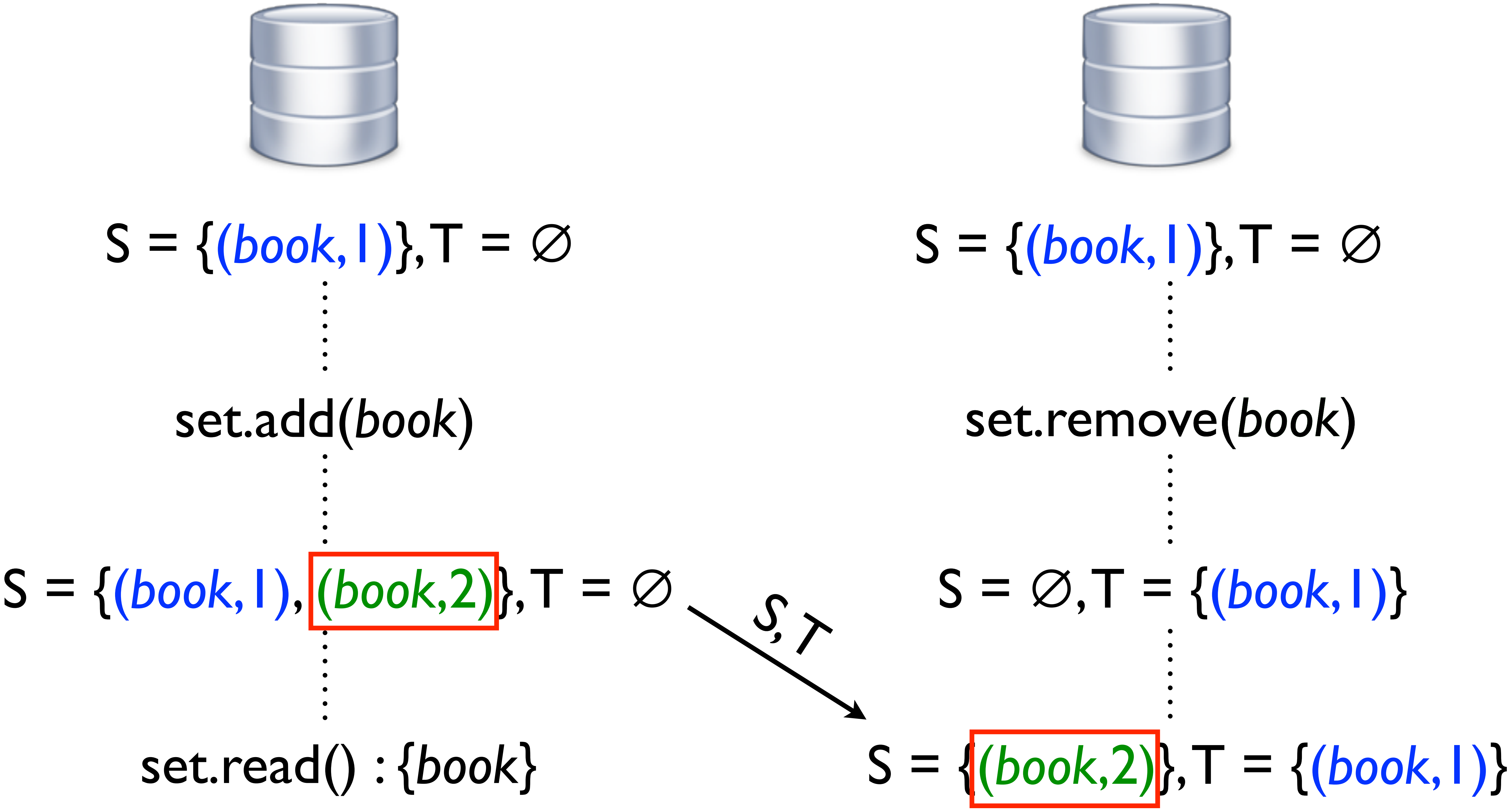
$S = \{(\mathit{book}, 1)\}, T = \varnothing$

set.remove(*book*)

$S = \varnothing, T = \{\boxed{(\mathit{book}, 1)}\}$

$\boxed{S = \{\}}, T = \{(\mathit{book}, 1)\}$

- Ignore arriving instances that are in T



$S = \{(book, 1)\}, T = \varnothing$

set.add(*book*)

$S = \{(book, 1), \boxed{(book, 2)}\}, T = \varnothing$

set.read() : {*book*}

S, *T*

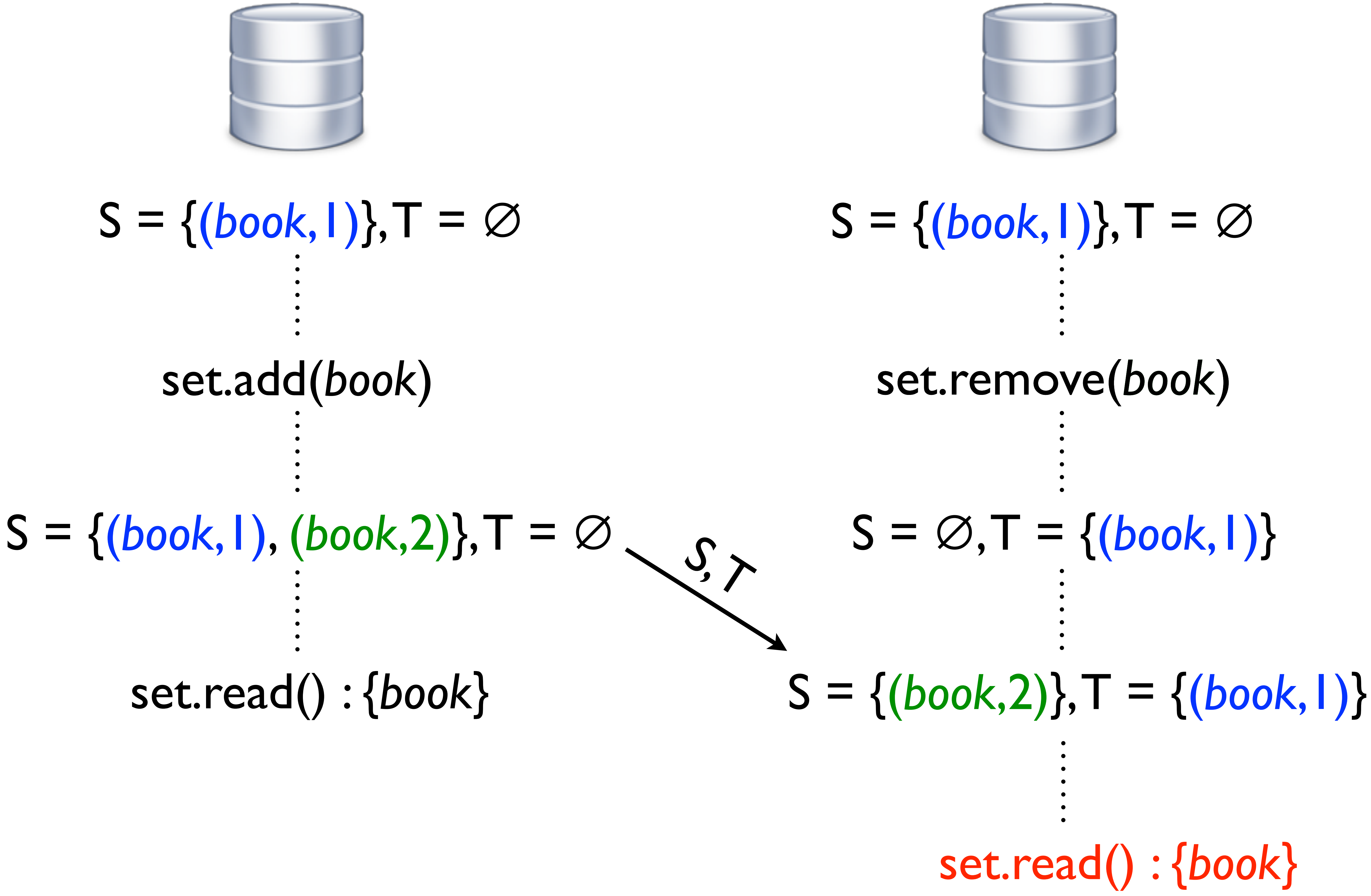
$S = \{(book, 1)\}, T = \varnothing$

set.remove(*book*)

$S = \varnothing, T = \{(book, 1)\}$

$S = \{\boxed{(book, 2)}\}, T = \{(book, 1)\}$

- Ignore arriving instances that are in T
- Add new arriving instances to S



$S = \{(book, 1)\}, T = \emptyset$

$set.add(book)$

$S = \{(book, 1), (book, 2)\}, T = \emptyset$

$set.read() : \{book\}$

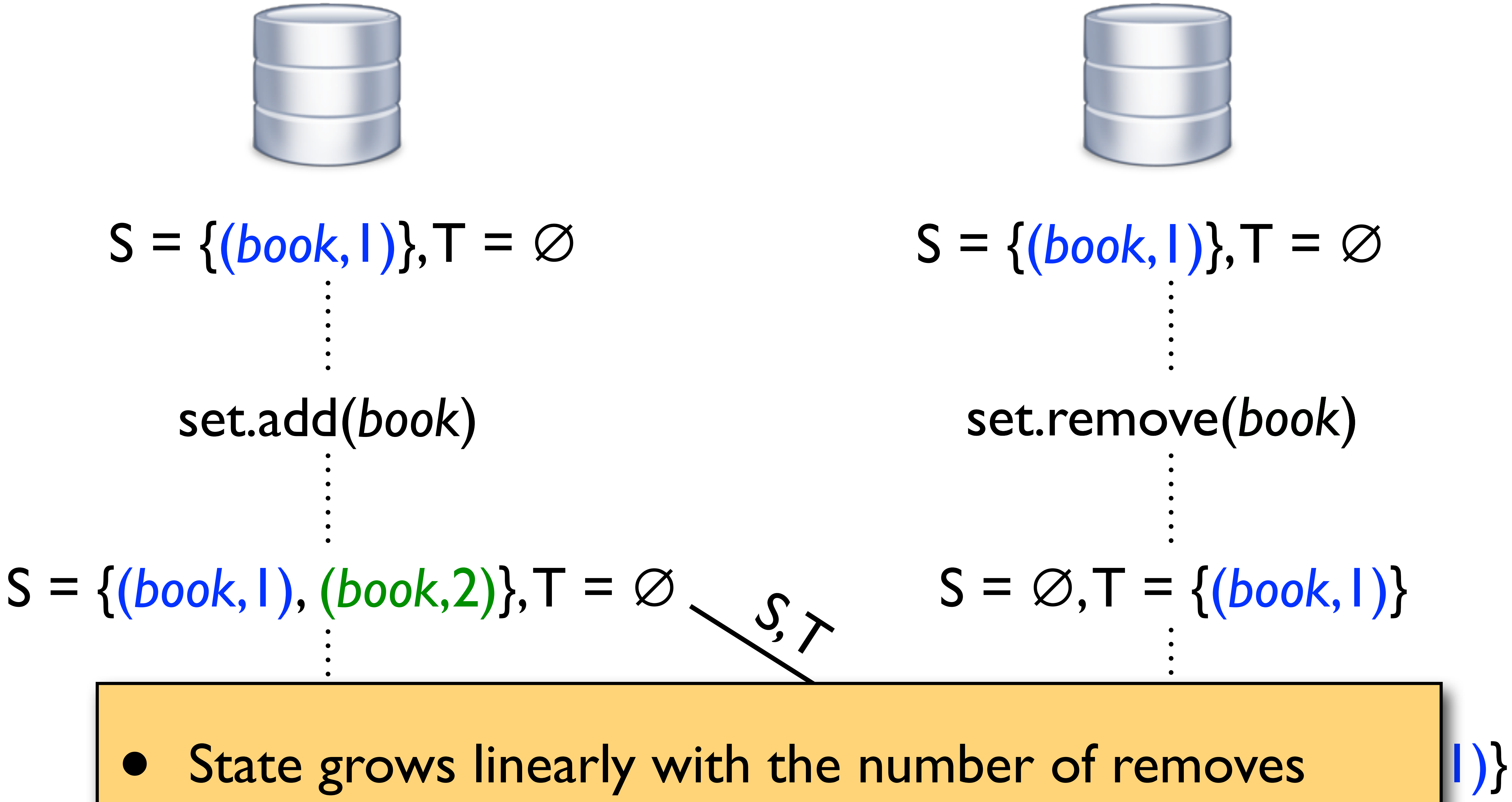
$S = \{(book, 1)\}, T = \emptyset$

$set.remove(book)$

$S = \emptyset, T = \{(book, 1)\}$

$S = \{(book, 2)\}, T = \{(book, 1)\}$

$set.read() : \{book\}$



$S = \{(\mathit{book}, 1)\}, T = \emptyset$

set.add(*book*)

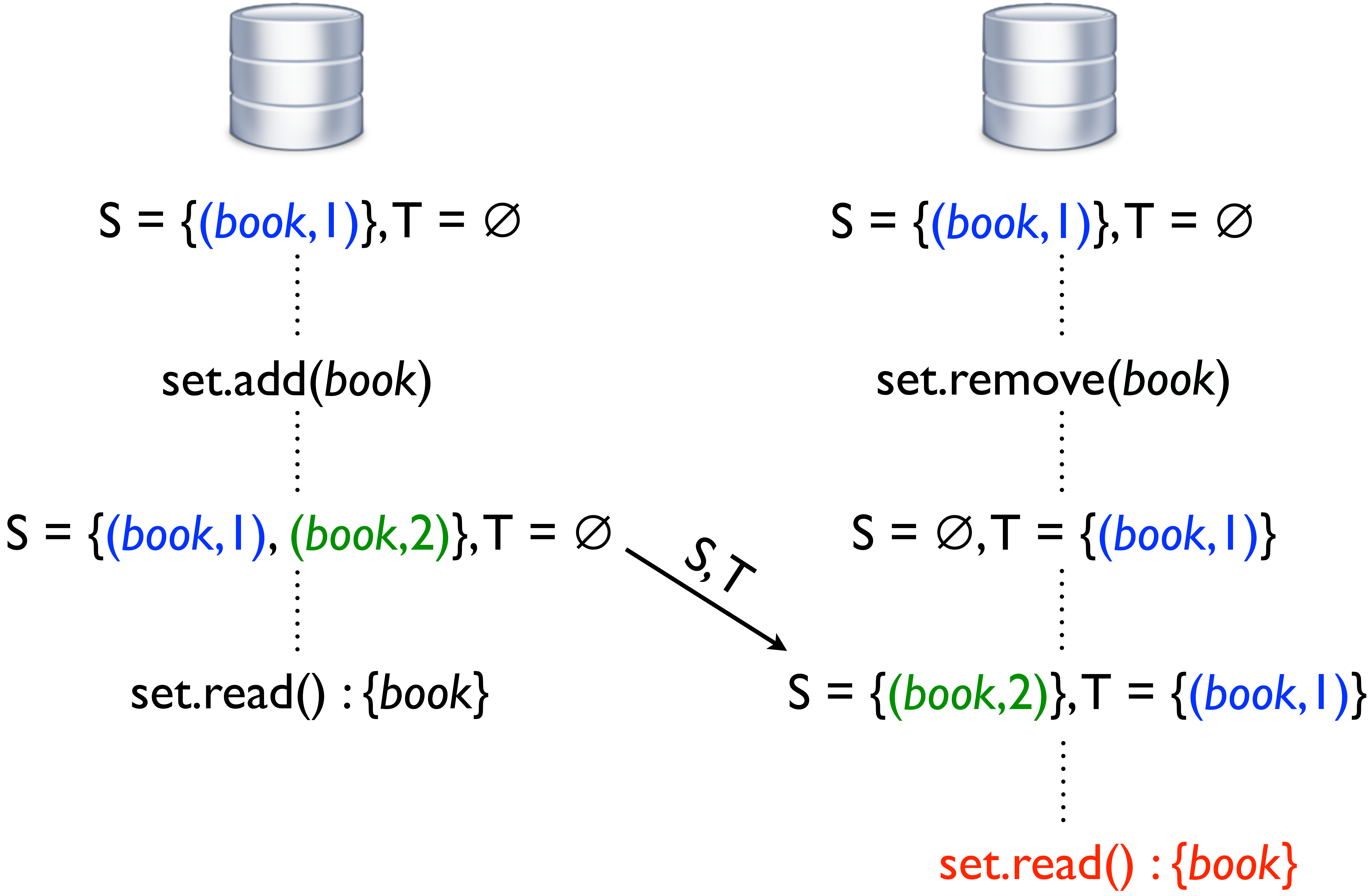
$S = \{(\mathit{book}, 1), (\mathit{book}, 2)\}, T = \emptyset$

$S = \{(\mathit{book}, 1)\}, T = \emptyset$

set.remove(*book*)

$S = \emptyset, T = \{(\mathit{book}, 1)\}$

- State grows linearly with the number of removes
- Realistic implementations represent T compactly: motivation for investigating space optimality
- We prove that space is Ω(log(number of operations))



$S = \{(book, 1)\}, T = \emptyset$

set.add(*book*)

$S = \{(book, 1), (book, 2)\}, T = \emptyset$

set.read() : {*book*}

S, *T*

$S = \{(book, 1)\}, T = \emptyset$

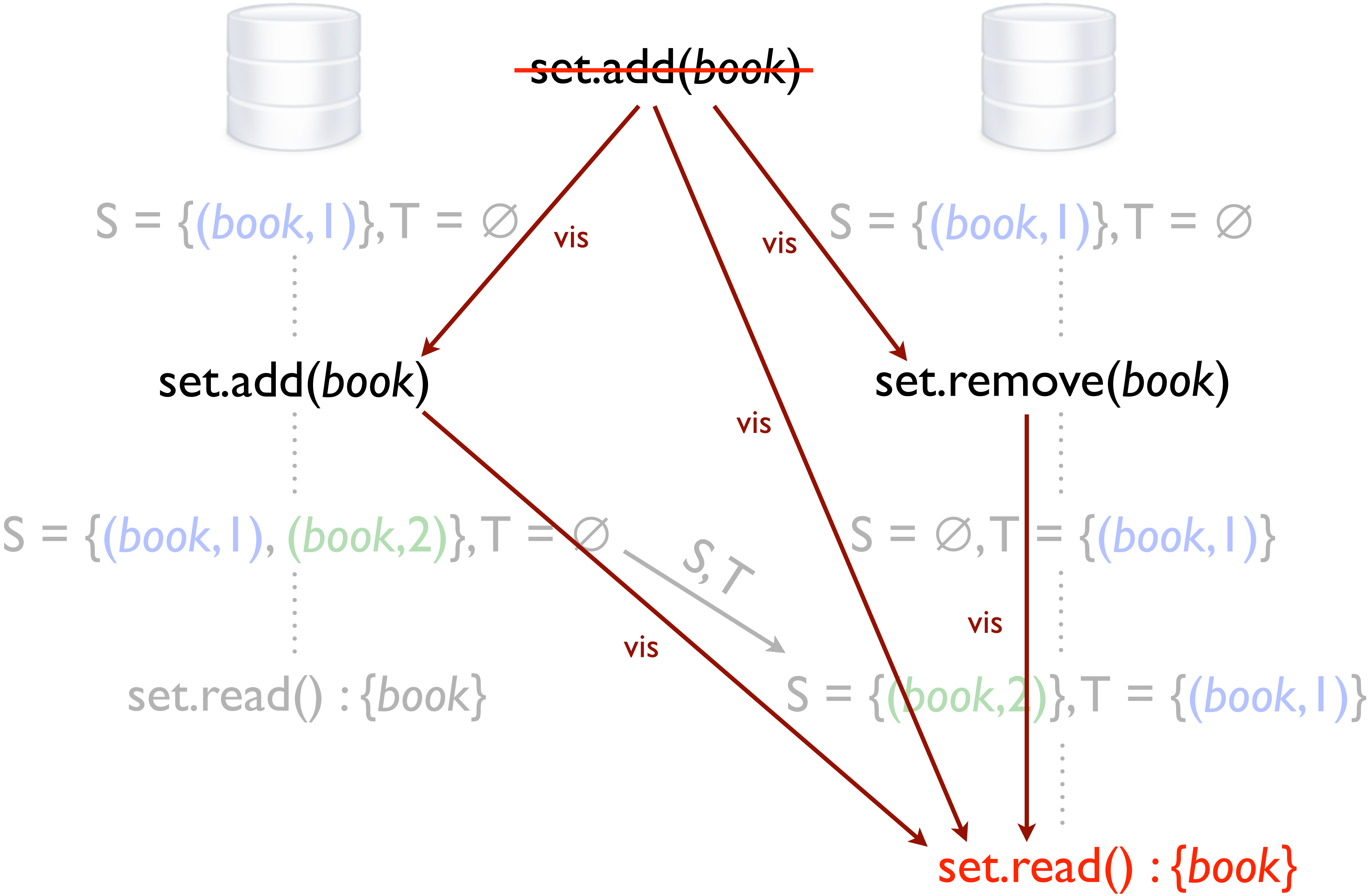
set.remove(*book*)

$S = \emptyset, T = \{(book, 1)\}$

$S = \{(book, 2)\}, T = \{(book, 1)\}$

set.read() : {*book*}

$\text{Impl} \models \text{F}$



~~set.add(*book*)~~

$S = \{(book, 1)\}, T = \varnothing$

$S = \{(book, 1)\}, T = \varnothing$

set.add(*book*)

set.remove(*book*)

$S = \{(book, 1), (book, 2)\}, T = \varnothing$

$S = \varnothing, T = \{(book, 1)\}$

set.read() : {*book*}

$S = \{(book, 2)\}, T = \{(book, 1)\}$

set.read() : {*book*}

$$\mathrm{Impl} \models \mathrm{F}$$

Data type correctness: Impl $\models$ F

- $\forall$ concrete execution of the implementation with any sequence of client operations
- $\exists$ corresponding abstract execution satisfying data type specifications and consistency axioms

Data type correctness: Impl $\models$ F

- $\forall$ concrete execution of the implementation with any sequence of client operations
- $\exists$ corresponding abstract execution satisfying data type specifications and consistency axioms

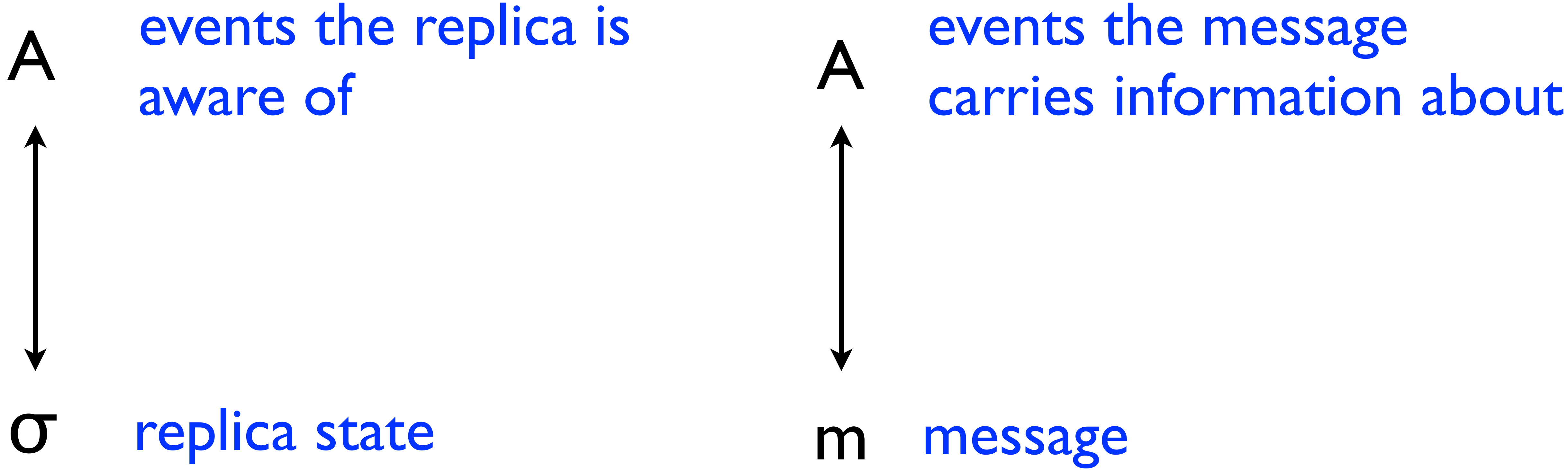
Data type correctness: Impl $\models$ F

- ∀ concrete execution of the implementation with any sequence of client operations
- ∃ corresponding abstract execution satisfying data type specifications and consistency axioms

- Requires reasoning about all replicas and interactions between them
- Want to modularise reasoning: construct the abstract execution from separate system configuration components

Replication-aware simulations

- Generalise simulation relations for abstract data types to replicated case
- Replica state or message associated with an abstract execution part describing events that led to it



A events the replica is aware of

σ replica state

A events the message carries information about

m message

Simulation for add-wins set

(S,T)

Set S: {(*book*,2), (*laptop*,3)}

Tombstones T: {(*book*,1)}

Simulation for add-wins set

(S,T) ↔ A

Set S: {(*book*,2), (*laptop*,3)}

Tombstones T: {(*book*,1)}

Simulation for add-wins set

(S,T)

Set S: {(*book*,2), (*laptop*,3)}

Tombstones T: {(*book*,1)}

$\leftrightarrow$

A

add(*book*)^{1} add(*laptop*)^{3}

add(*book*)^{2}

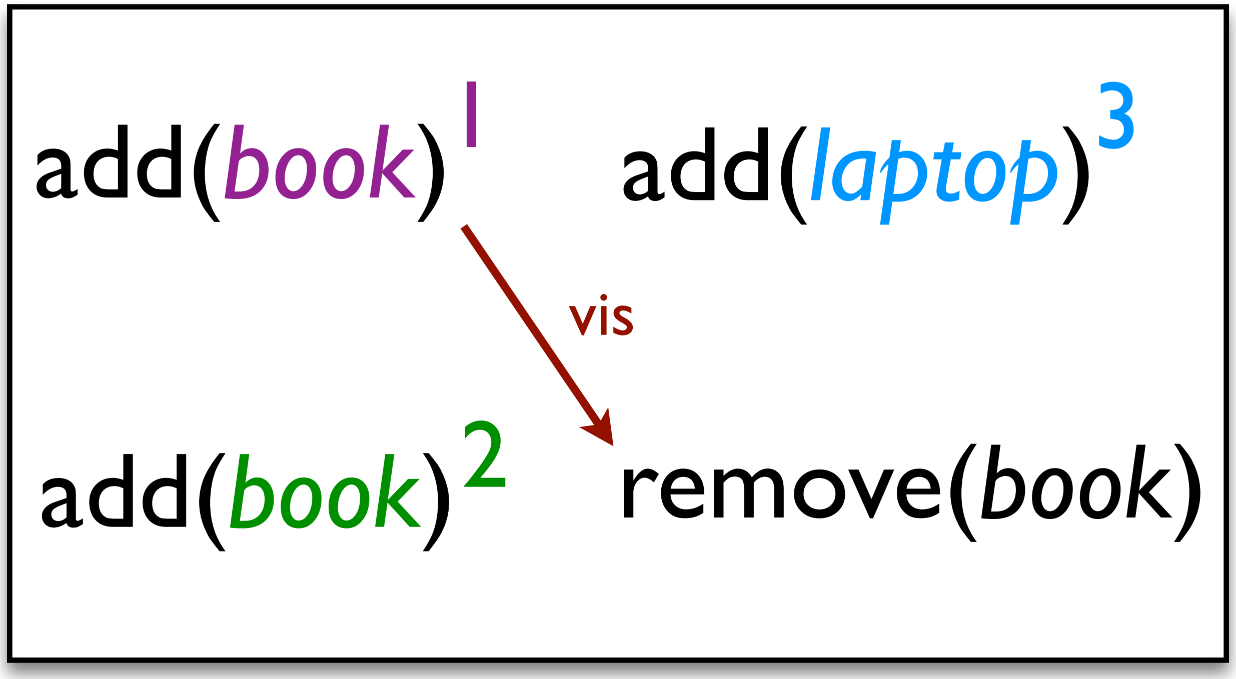
$$(\text{elt}, \text{id}) \in S \cup T \quad \leftrightarrow \quad \text{add}(\text{elt})^{\text{id}} \in A$$

Simulation for add-wins set

(S,T) ↔ A

Set S: {(*book*,2), (*laptop*,3)}

Tombstones T: $\boxed{\{(\textit{book},1)\}}$



add(*book*)^{1} add(*laptop*)^{3}

add(*book*)^{2} remove(*book*)

$$(\text{elt}, \text{id}) \in S \cup T \quad \longleftrightarrow \quad \text{add}(\text{elt})^{\text{id}} \in A$$

$$(\text{elt}, \text{id}) \in T \quad \longrightarrow \quad \text{remove}(\text{elt}) \xleftarrow{\text{vis}} \text{add}(\text{elt})^{\text{id}}$$

Simulation for add-wins set

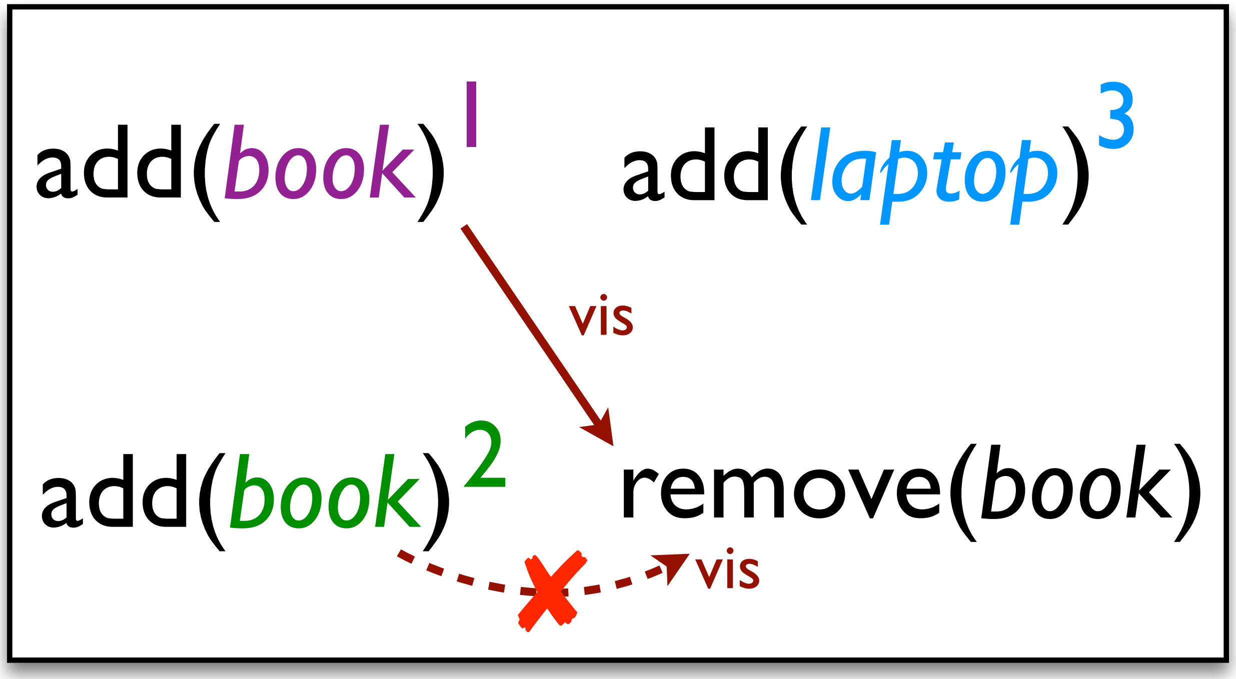
(S,T)

Set S: {(*book*,2), (*laptop*,3)}

Tombstones T: {(*book*,1)}

↔

A



$\mathrm{add}(\textit{book})^{1}$

$\mathrm{add}(\textit{laptop})^{3}$

vis

$\mathrm{add}(\textit{book})^{2}$

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

vis

$$(\mathrm{elt}, \mathrm{id}) \in S \cup T \quad \longleftrightarrow \quad \mathrm{add}(\mathrm{elt})^{\mathrm{id}} \in A$$

$$(\mathrm{elt}, \mathrm{id}) \in T \quad \longrightarrow \quad \mathrm{remove}(\mathrm{elt}) \xleftarrow{\text{vis}} \mathrm{add}(\mathrm{elt})^{\mathrm{id}}$$

$$(\mathrm{elt}, \mathrm{id}) \in S \quad \longrightarrow \quad \neg\, \mathrm{add}(\mathrm{elt})^{\mathrm{id}} \xrightarrow{\text{vis}} \mathrm{remove}(\mathrm{elt})$$

Proof obligations

- Relations are preserved during a system run
- Relations imply that the abstract execution satisfies the data type specification

Proof obligations

- Relations are preserved during a system run
- Relations imply that the abstract execution satisfies the data type specification

Executing an operation:

$$\sigma \xrightarrow[\mathrm{op}()\ :\ \mathrm{res}]{} \sigma'$$

Proof obligations

- Relations are preserved during a system run
- Relations imply that the abstract execution satisfies the data type specification

Executing an operation:

$$\begin{tikzcd}
A \arrow[d, leftrightarrow] & \\
\sigma \arrow[r, "\text{op() : res}"'] & \sigma'
\end{tikzcd}$$

Proof obligations

- Relations are preserved during a system run
- Relations imply that the abstract execution satisfies the data type specification

Executing an operation:

$$\begin{tikzcd}
A \arrow[r, dashed] \arrow[d, leftrightarrow] & A' \approx A + \{op\} \arrow[d, dashed, leftrightarrow] \\
\sigma \arrow[r, "op() : res"'] & \sigma'
\end{tikzcd}$$

Proof obligations

- Relations are preserved during a system run
- Relations imply that the abstract execution satisfies the data type specification

Executing an operation:

$$\begin{tikzcd}
A \arrow[r, dashed] \arrow[d, leftrightarrow] & A' \approx A + \{op\} \arrow[d, dashed, leftrightarrow] \\
\sigma \arrow[r, "op() : res"'] & \sigma'
\end{tikzcd}$$

And check

$res = F(Context_{A'}(op))$

Proof obligations

- Relations are preserved during a system run
- Relations imply that the abstract execution satisfies the data type specification

Executing an operation:

$$\begin{tikzcd}
A \arrow[r, dashed] \arrow[d, leftrightarrow] & A' \approx A + \{op\} \arrow[d, leftrightarrow, dashed] \\
\sigma \arrow[r, "op() : res"'] & \sigma'
\end{tikzcd}$$

And check
$res = F(\mathrm{Context}_{A'}(op))$

Modular: considers the state of a single replica

Receiving a message

$$(\sigma, m) \xrightarrow[\text{receive}(m)]{} \sigma'$$

Receiving a message

$$\begin{tikzcd}
(A, B) \arrow[d, leftrightarrow, shift right] \arrow[d, leftrightarrow, shift left] & \\
(\sigma, m) \arrow[r, "\text{receive}(m)"'] & \sigma'
\end{tikzcd}$$

Receiving a message

$$\begin{tikzcd}
(A, B) \arrow[r, dashed] \arrow[d, leftrightarrow, shift right] \arrow[d, leftrightarrow, shift left] & A' \arrow[d, leftrightarrow, dashed] \\
(\sigma, m) \arrow[r, "\text{receive}(m)"'] & \sigma'
\end{tikzcd}$$

Receiving a message

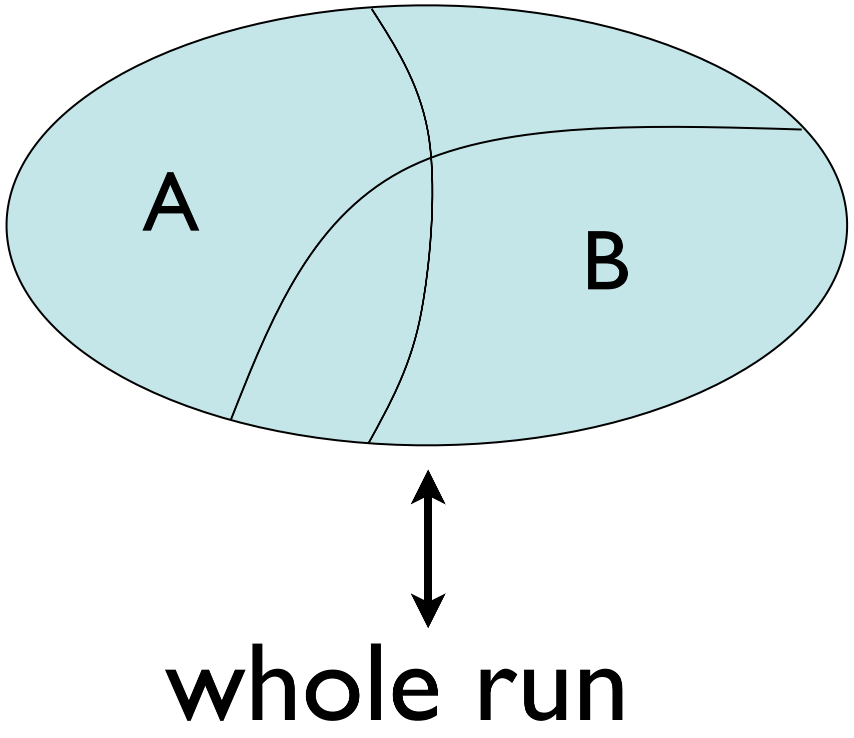
$$\begin{tikzcd}
(A, B) \arrow[r, dashed] \arrow[d, <->, shift right=2] \arrow[d, <->, shift left=2] & A' \arrow[d, <->, dashed] \\
(\sigma, m) \arrow[r, "\text{receive}(m)"'] & \sigma'
\end{tikzcd}$$

Good news: modular - consider the state of a single replica and a message

Bad news: modularity leads to incompleteness - loses required global information

Source of incompleteness

$$\begin{tikzcd}
(A, B) \arrow[r, dashed] \arrow[d, leftrightarrow, shift right] \arrow[d, leftrightarrow, shift left] & A' \arrow[d, dashed, leftrightarrow] \\
(\sigma, m) \arrow[r] & \sigma'
\end{tikzcd}$$



- A and B parts of the same abstract execution ➔ can be correlated by some invariants
  - *Visibility can't contradict on events common to A and B*
  - *Union of visibility relations in A and B itself a well-formed visibility relation ➔ acyclic*
- Simulation relations per-component ➔ don't give this

Solution: 2-stage verification

①

- Fix a class of data types implementations with similar messaging behaviour

  State-based: propagate information by sending full replica state

- Prove key global invariants non-modularly
- Unpleasant, but done once for the class

Solution: 2-stage verification

①
- Fix a class of data types implementations with similar messaging behaviour

  State-based: propagate information by sending full replica state
- Prove key global invariants non-modularly
- Unpleasant, but done once for the class

②
- For any implementation within the class
- Verify it modularly using replication-aware simulations while assuming the global invariants

Solution: 2-s

Technical details in the paper

①

- Fix a class of data types implementations with similar messaging behaviour

  State-based: propagate information by sending full replica state

- Prove key global invariants non-modularly
- Unpleasant, but done once for the class

②

- For any implementation within the class
- Verify it modularly using replication-aware simulations while assuming the global invariants

Summary

- First techniques for reasoning about eventual consistency and replicated data types
  - Specifying the intended semantics
  - Verifying replicated data type correctness
- Only the first step
  - Replicated data types only one system component
  - More work needed even for them: list data type, used for collaborative editing (Office Online, Google Docs)

Programming languages/verification vs distributed systems

- Put eventually consistent distributed systems onto the PL/verification agenda
- Usual paradigm: developing verification techniques
- But also: helping systems researchers design architectures and programming interfaces
  - Tricky to figure out semantics & implementation for complex interfaces: multiple consistency levels, transactions

Common ground: weak memory models

- Lot of recent work on weak memory
- Opportunity: apply weak memory technology to distributed systems

Common ground: weak memory models

- Lot of recent work on weak memory
- Opportunity: apply weak memory technology to distributed systems
- Processor and language models have very little known motivation
- Distributed systems are different: implemented algorithms motivate models