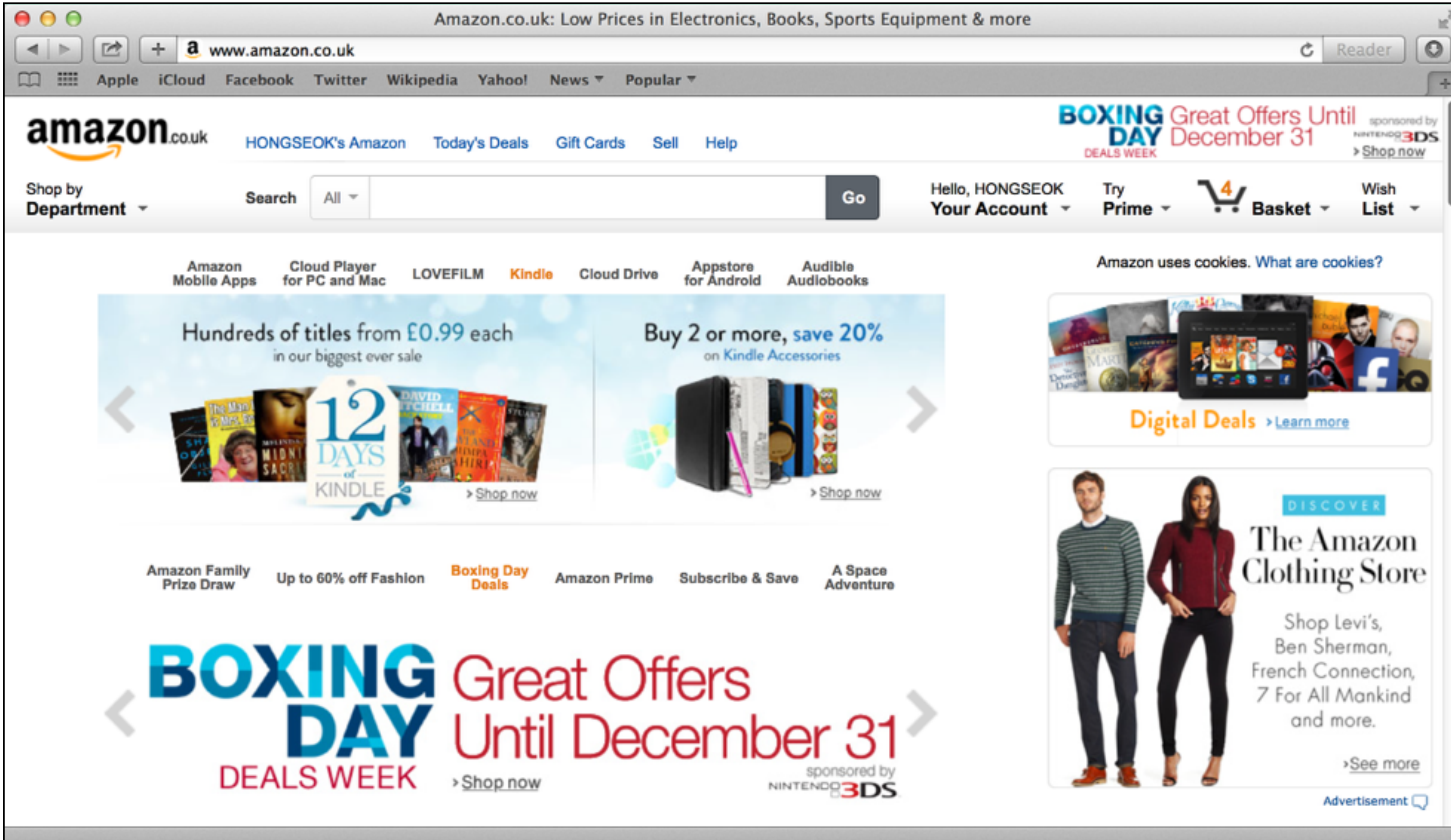
Replicated Data Types: Specification, Verification, Optimality

Hongseok Yang
University of Oxford

Joint work with Sebastian Burckhardt (Microsoft Research),
Alexey Gotsman (IMDEA) and Marek Zawirski (UPMC & INRIA)

What is it like writing programs running on top of weakly consistent distributed stores?


Amazon.co.uk: Low Prices in Electronics, Books, Sports Equipment & more
www.amazon.co.uk
Reader
Apple
iCloud
Facebook
Twitter
Wikipedia
Yahoo!
News
Popular
amazon.co.uk
HONGSEOK's Amazon
Today's Deals
Gift Cards
Sell
Help
BOXING DAY DEALS WEEK Great Offers Until December 31
sponsored by NINTENDO 3DS
Shop now
Shop by Department
Search
All
Go
Hello, HONGSEOK
Your Account
Try Prime
4
Basket
Wish List
Amazon Mobile Apps
Cloud Player for PC and Mac
LOVEFiLM
Kindle
Cloud Drive
Appstore for Android
Audible Audiobooks
Hundreds of titles from £0.99 each
in our biggest ever sale
12 DAYS of KINDLE
Shop now
Buy 2 or more, save 20%
on Kindle Accessories
Shop now
Amazon uses cookies. What are cookies?
Digital Deals
Learn more
Amazon Family Prize Draw
Up to 60% off Fashion
Boxing Day Deals
Amazon Prime
Subscribe & Save
A Space Adventure
BOXING DAY DEALS WEEK
Great Offers Until December 31
Shop now
sponsored by NINTENDO 3DS
DISCOVER
The Amazon Clothing Store
Shop Levi's, Ben Sherman, French Connection, 7 For All Mankind and more.
See more
Advertisement

Information about each customer

Shopping cart (Shapiro et al.)

Bob
in UK

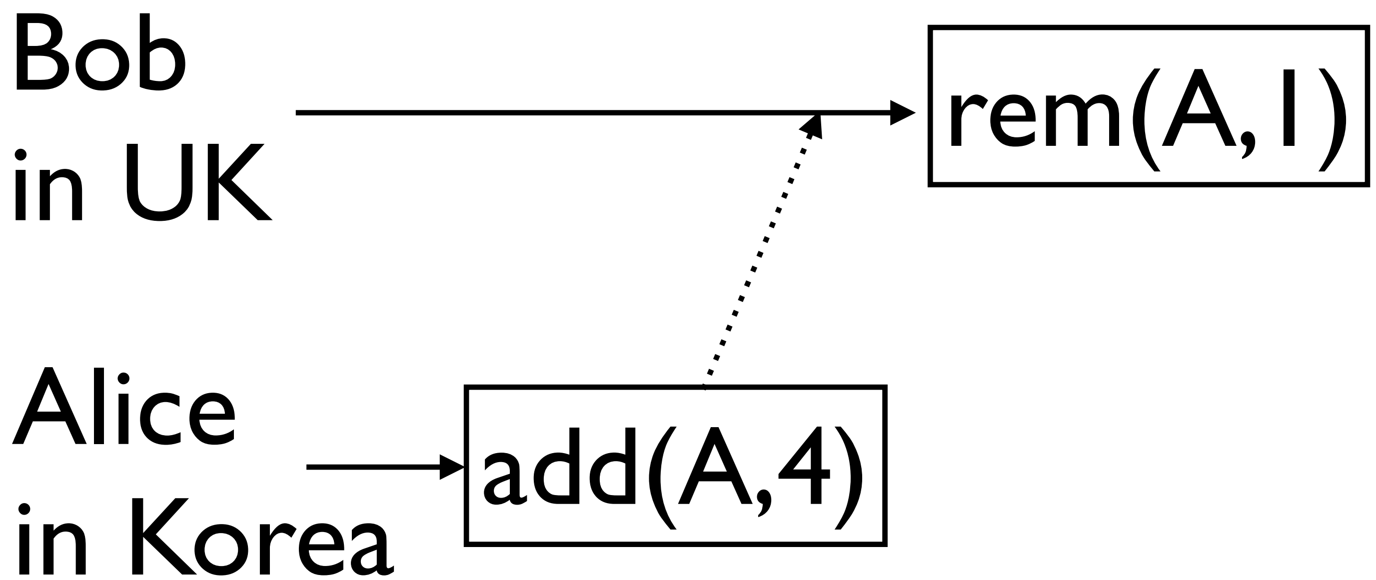
Alice
in Korea

Shopping cart (Shapiro et al.)

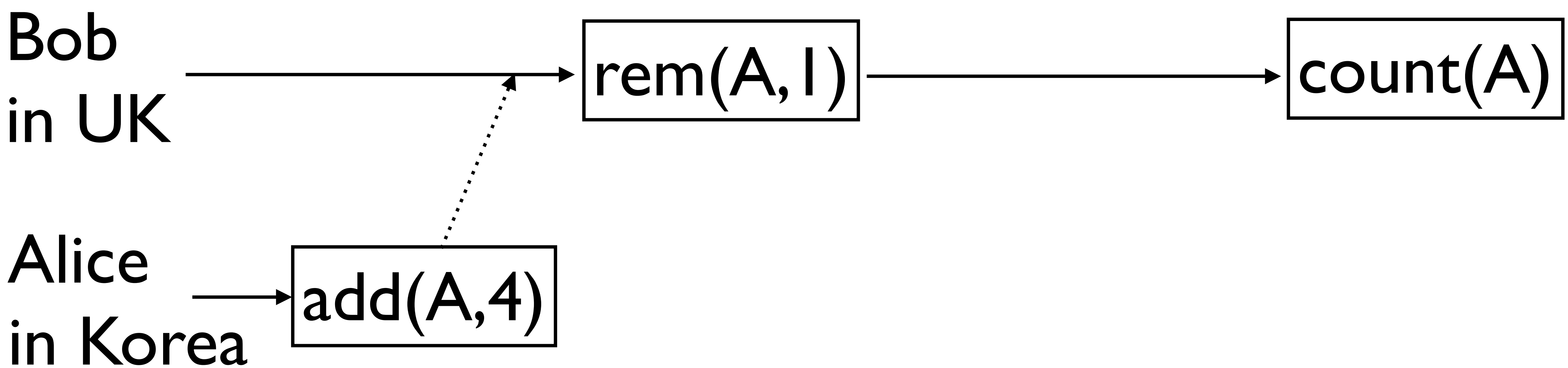
Bob
in UK

Alice
in Korea → $\boxed{\text{add(A,4)}}$

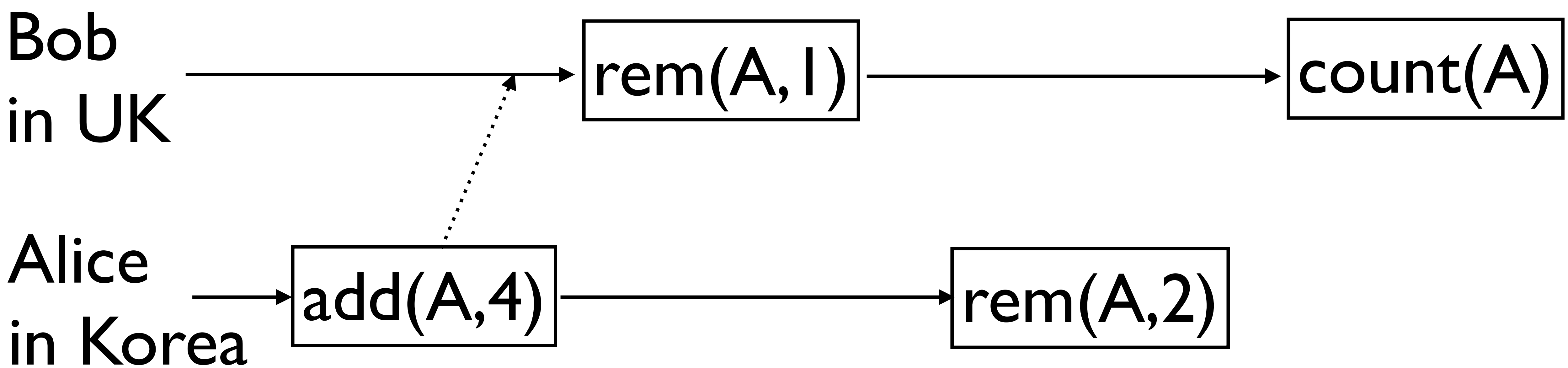
Shopping cart (Shapiro et al.)



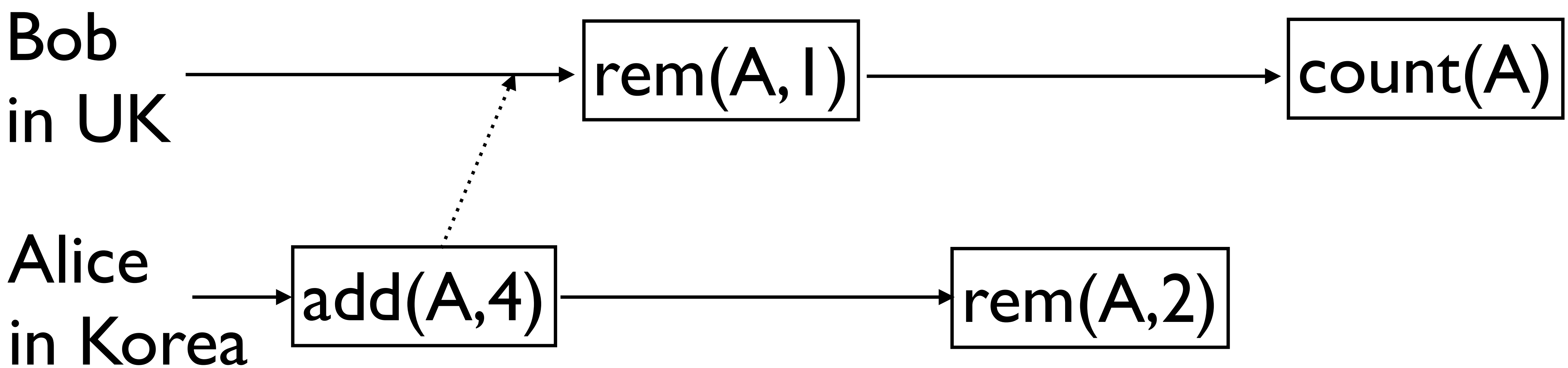
Shopping cart (Shapiro et al.)



Shopping cart (Shapiro et al.)

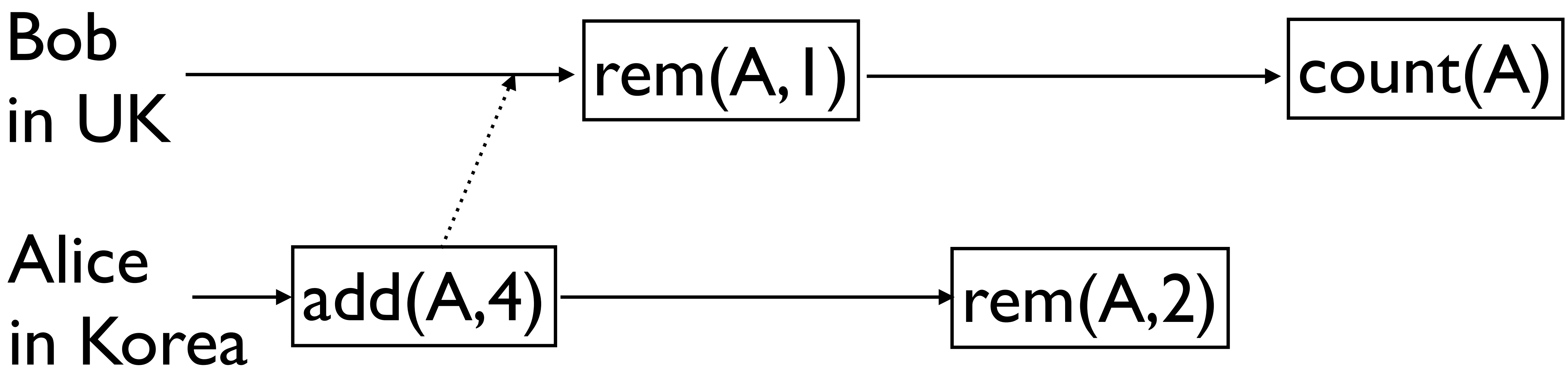


Shopping cart (Shapiro et al.)



[Q] What cannot be the result of count(A)?
(a) 1 (b) 3 (c) 5 (d) all possible

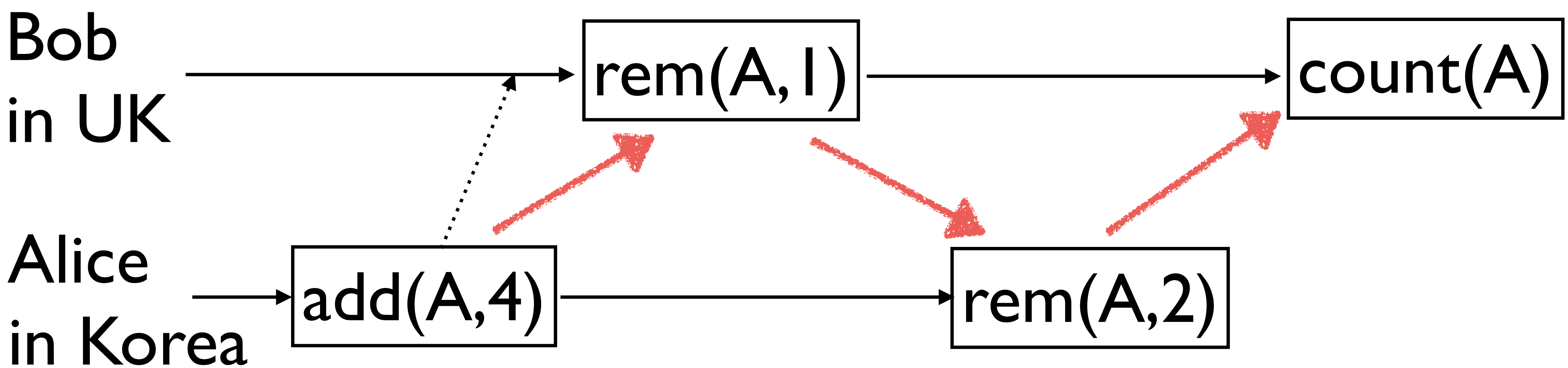
Shopping cart (Shapiro et al.)



[Q] What cannot be the result of count(A)?

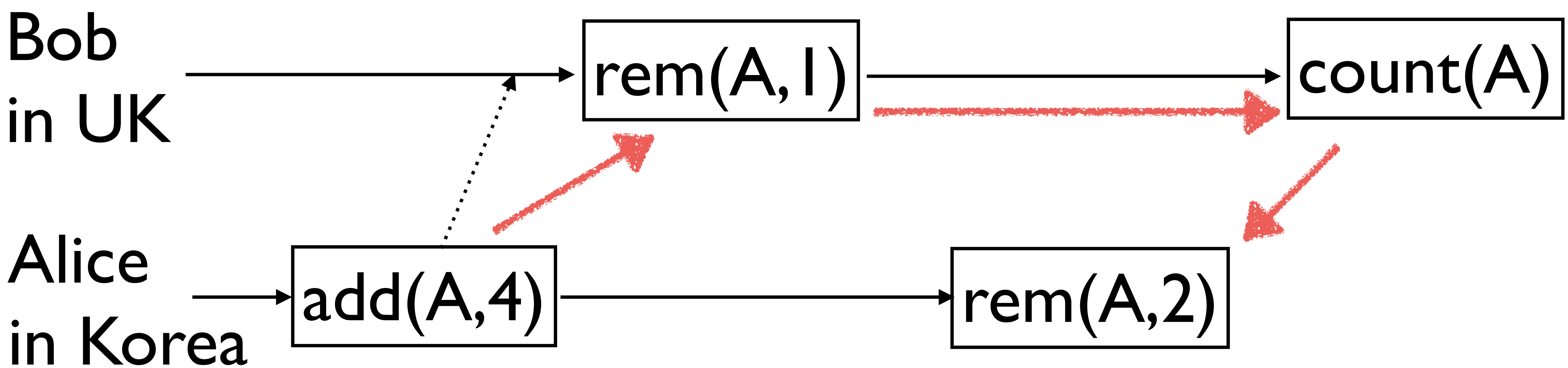
(a) 1 (b) 3 (c) 5 (d) all possible

Shopping cart (Shapiro et al.)



[Q] What cannot be the result of count(A)?
(a) 1 (b) 3 (c) 5 (d) all possible

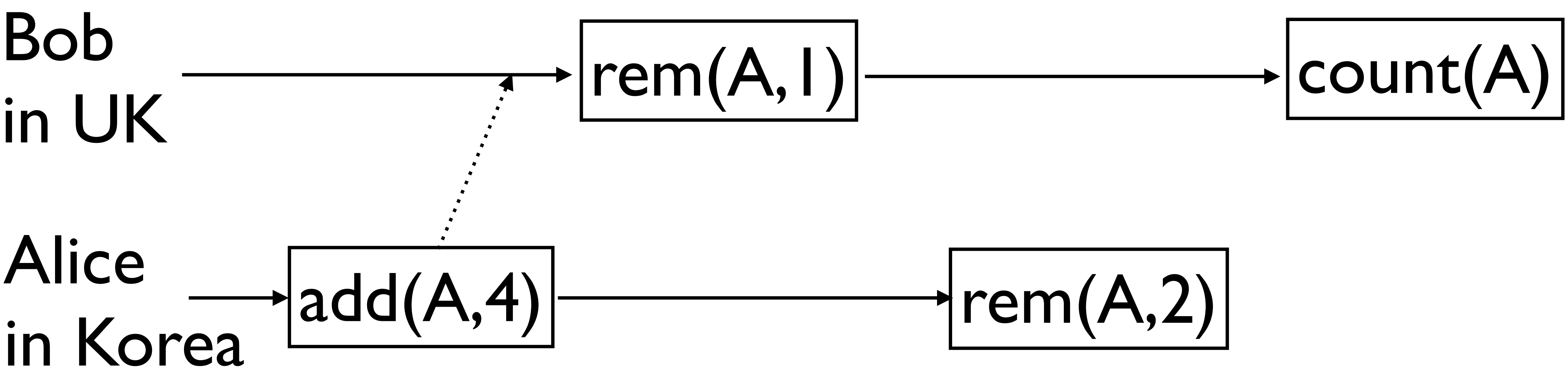
Shopping cart (Shapiro et al.)



[Q] What cannot be the result of count(A)?

(a) 1 (b) 3 (c) 5 (d) all possible

Shopping cart (Shapiro et al.)



[Q] What cannot be the result of count(A)?

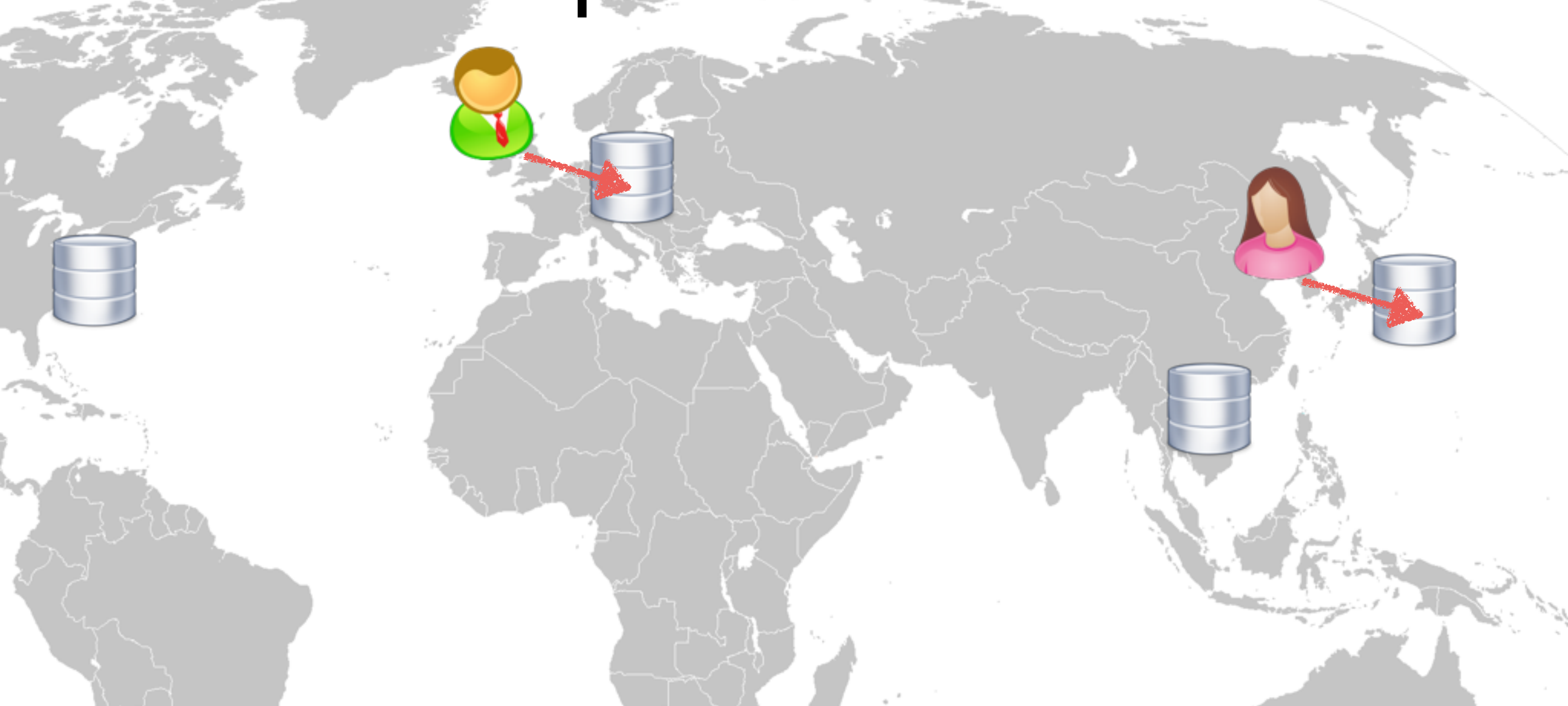
(a) 1 (b) 3 (c) 5 (d) all possible

Because of weak consistency of the store.

Geo-replicated databases

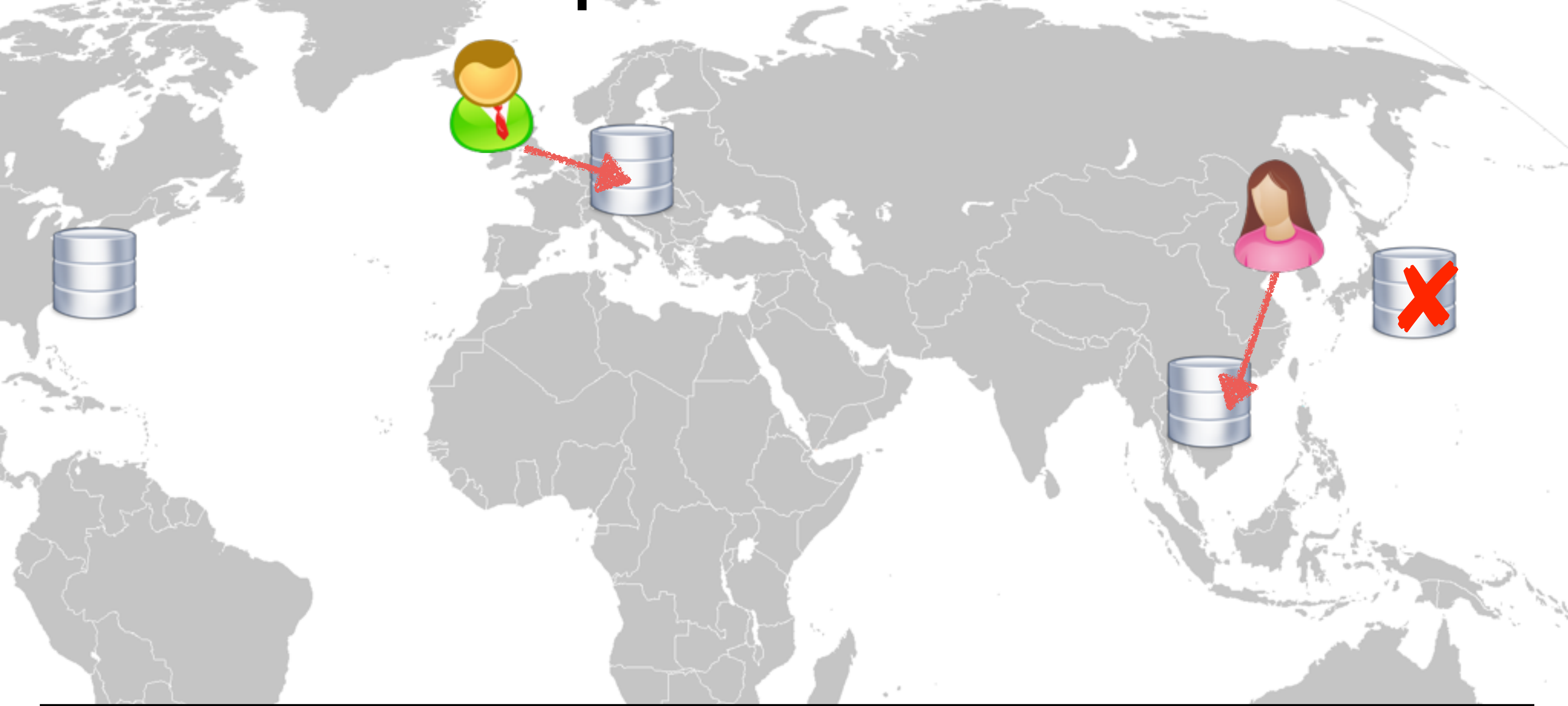
- Every data centre stores a complete replica of data
- Purpose: Minimising latency. Fault tolerance.

Geo-replicated databases



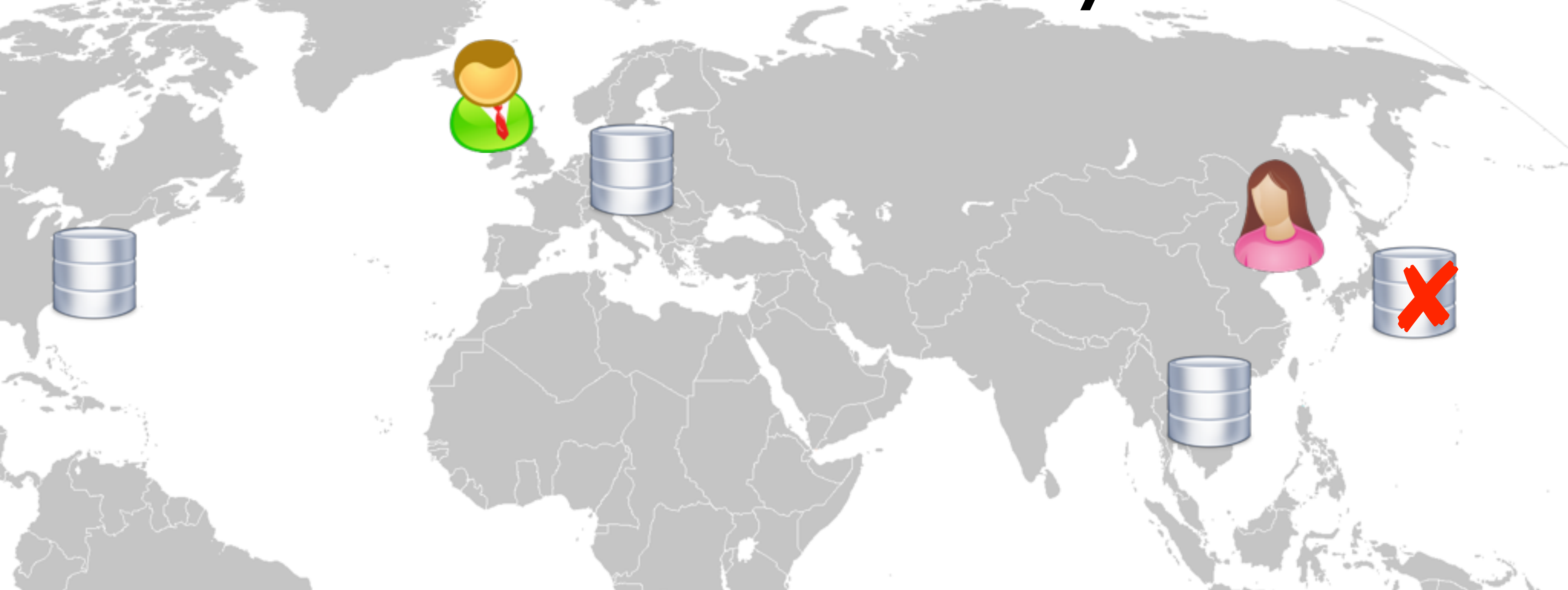
- Every data centre stores a complete replica of data
- Purpose: Minimising latency. Fault tolerance.

Geo-replicated databases



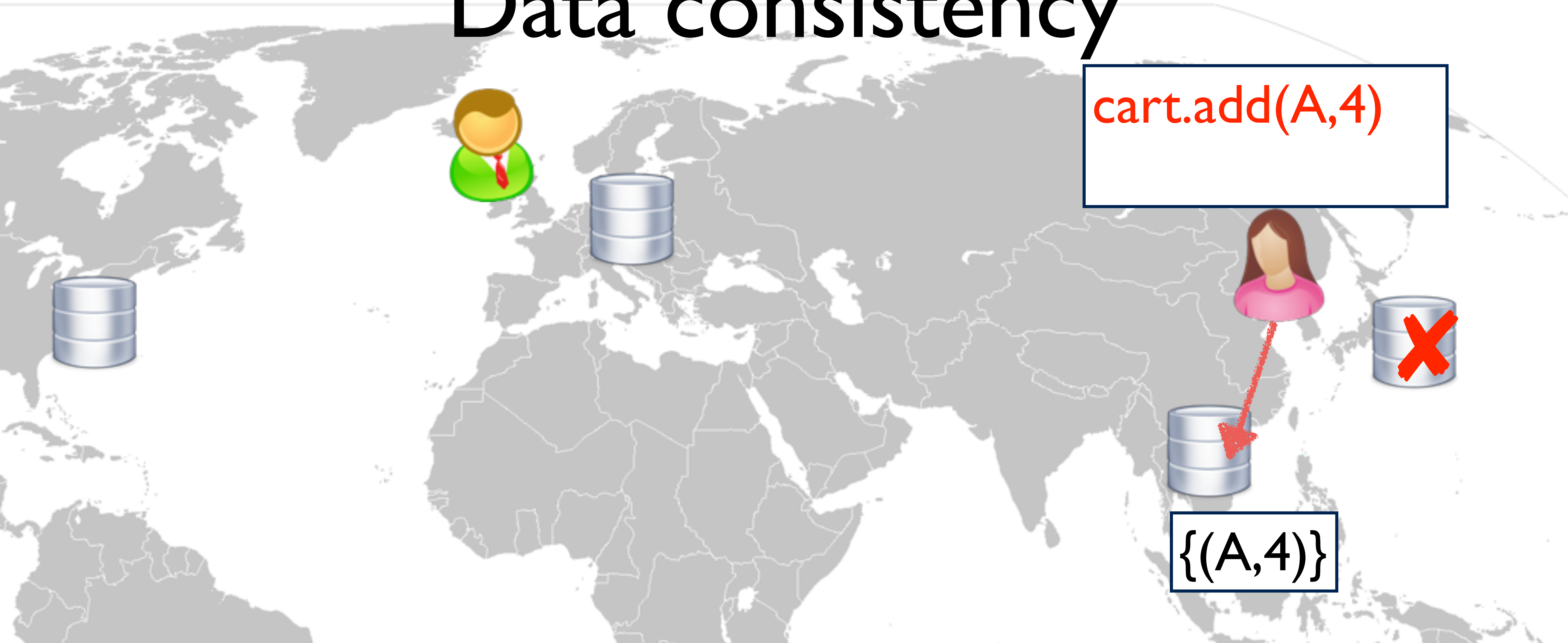
- Every data centre stores a complete replica of data
- Purpose: Minimising latency. Fault tolerance.

Data consistency



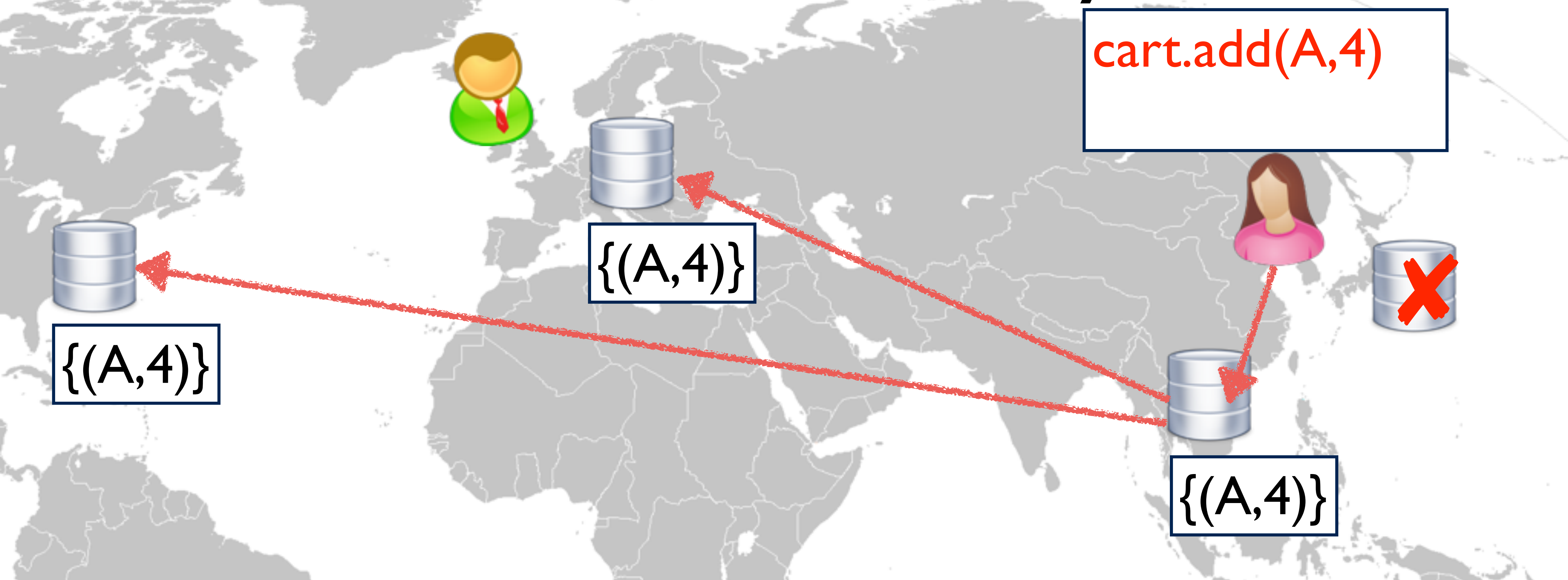
- Do data centre stores behave as if a single store?
- Strong consistency: Yes.
- Weak consistency: No.

Data consistency



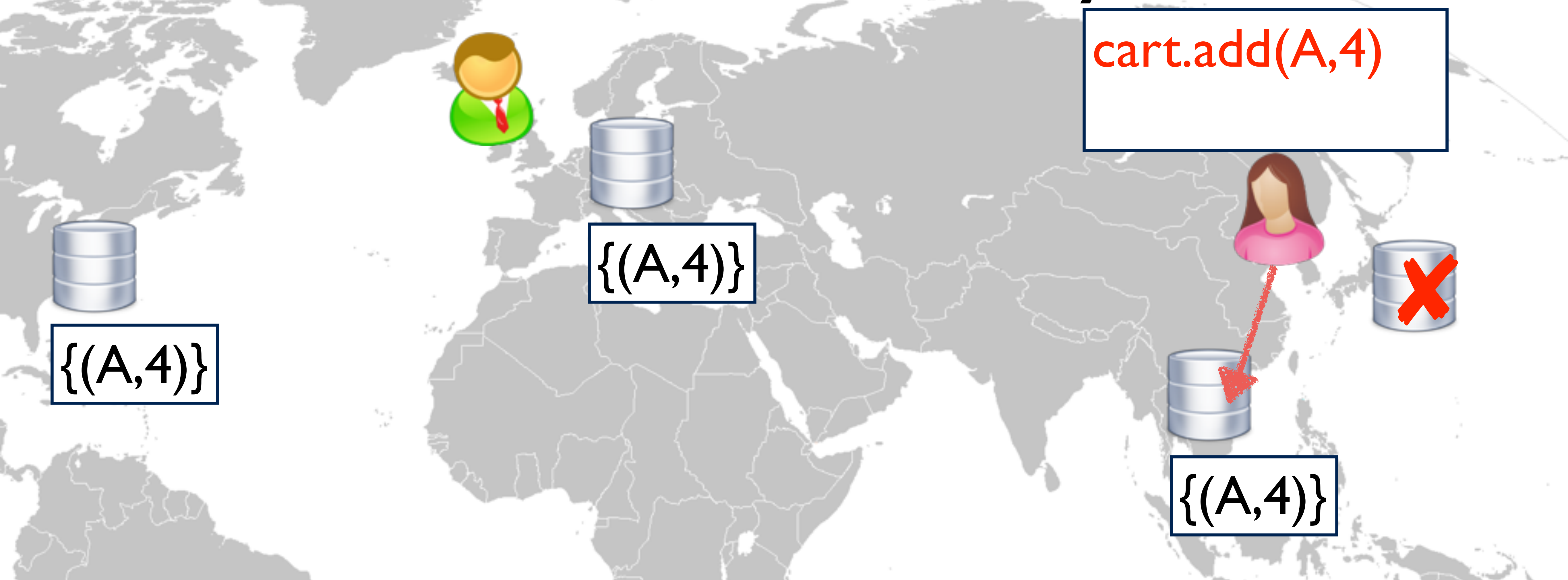
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



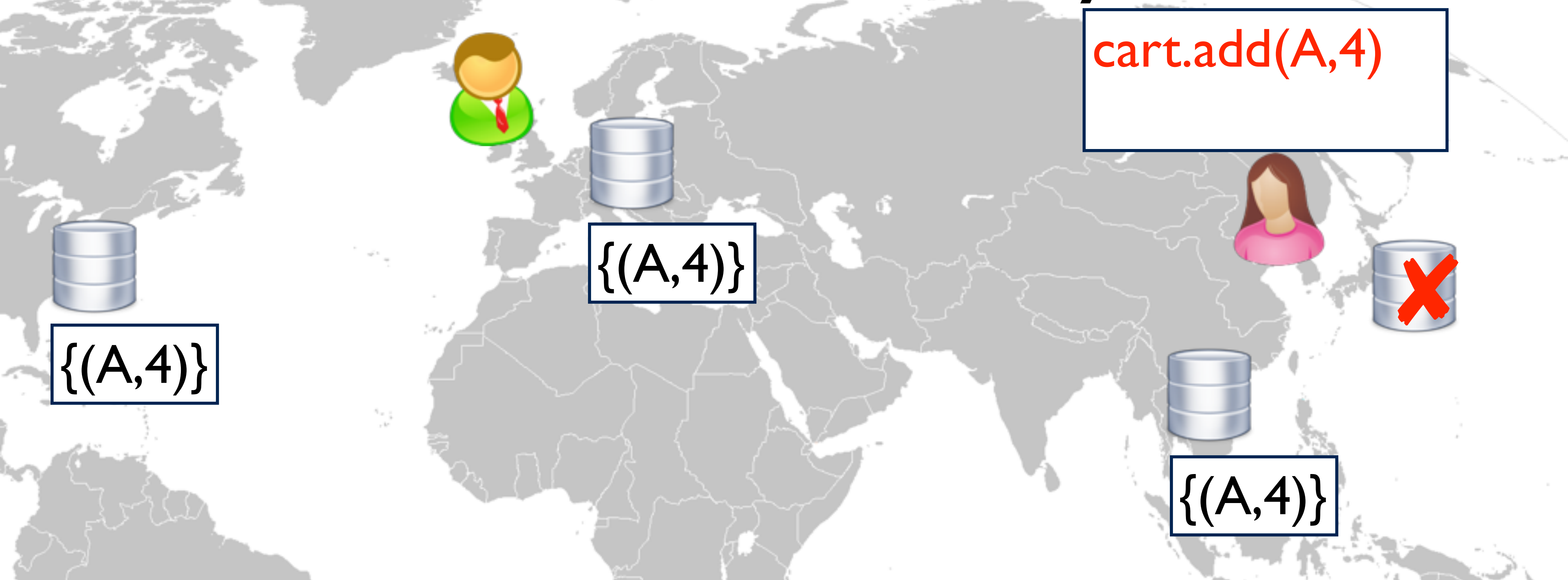
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



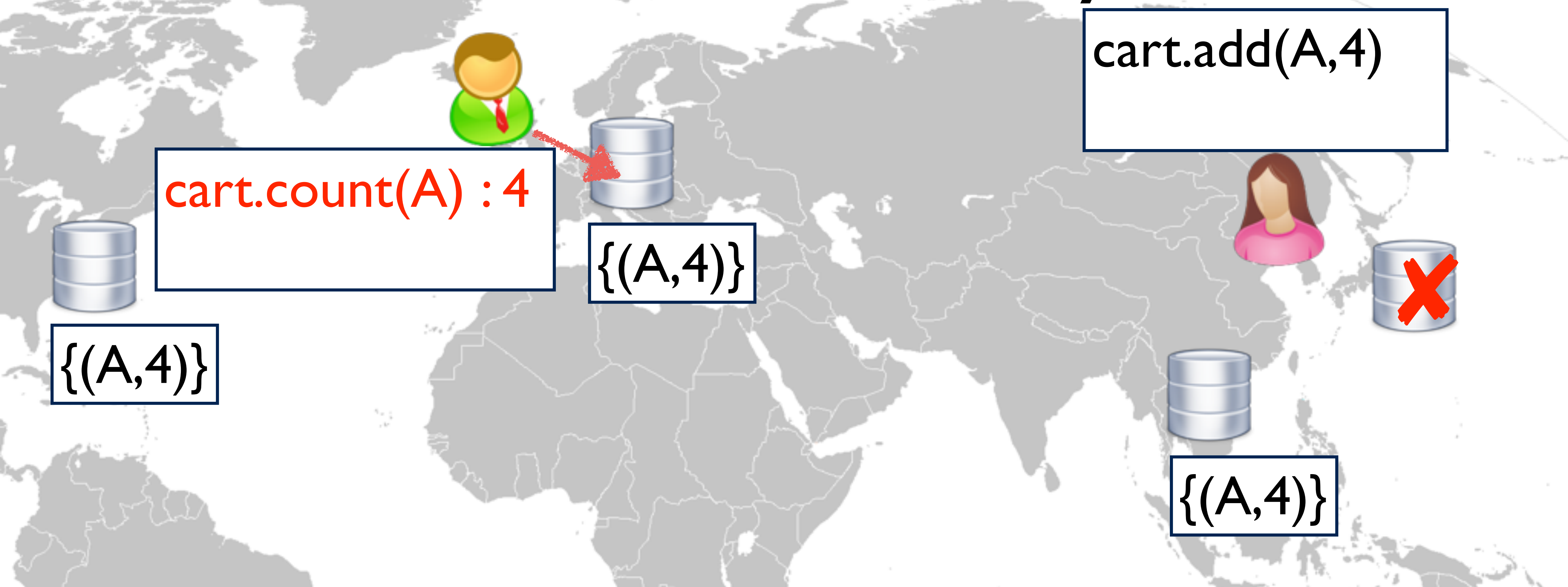
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



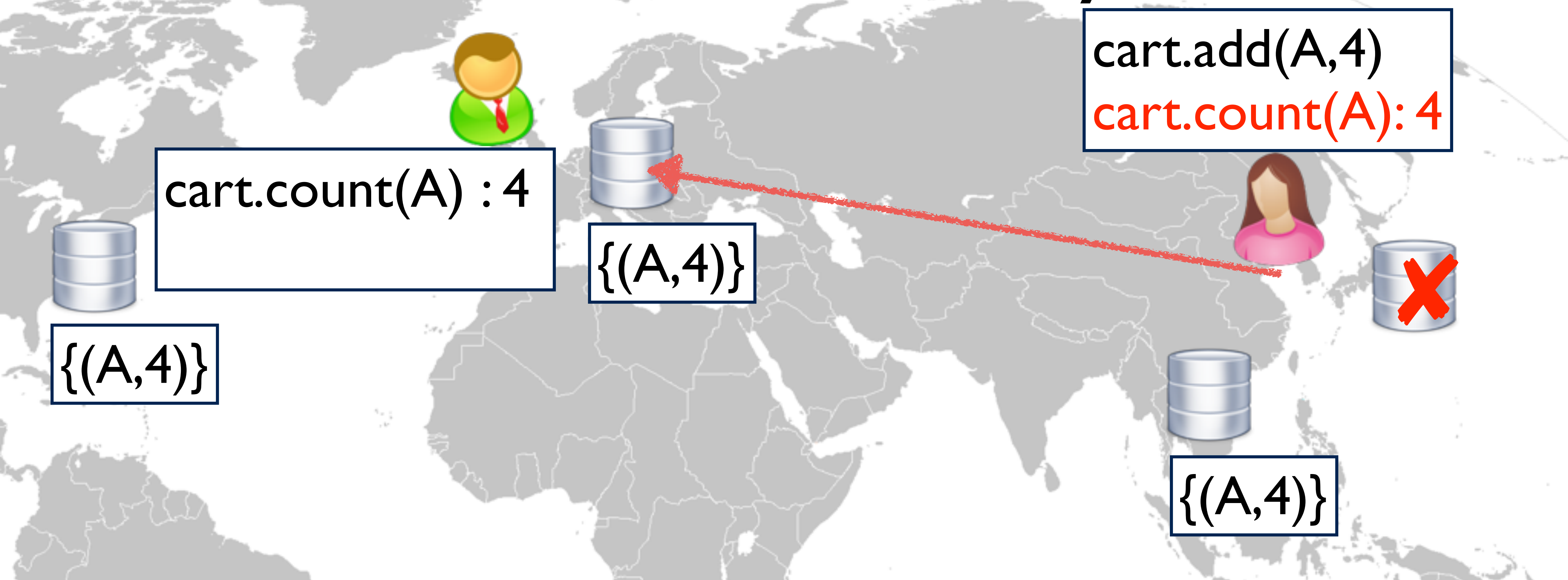
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



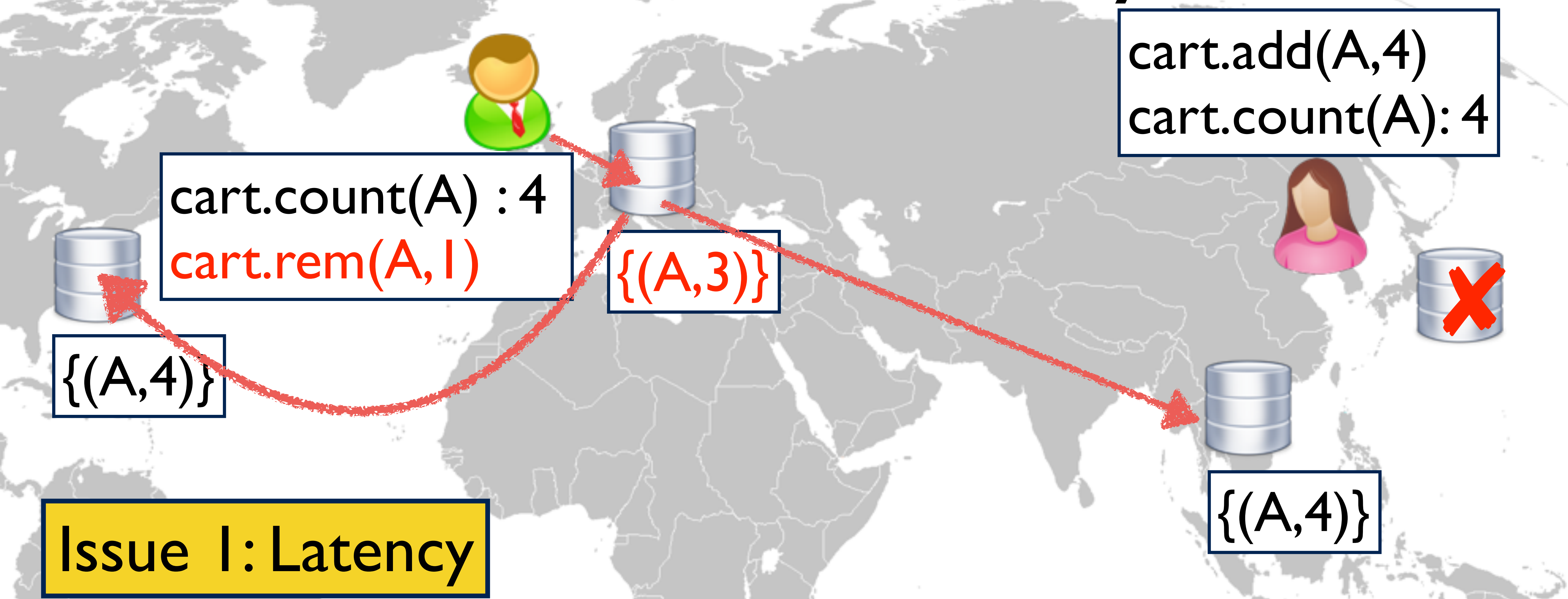
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



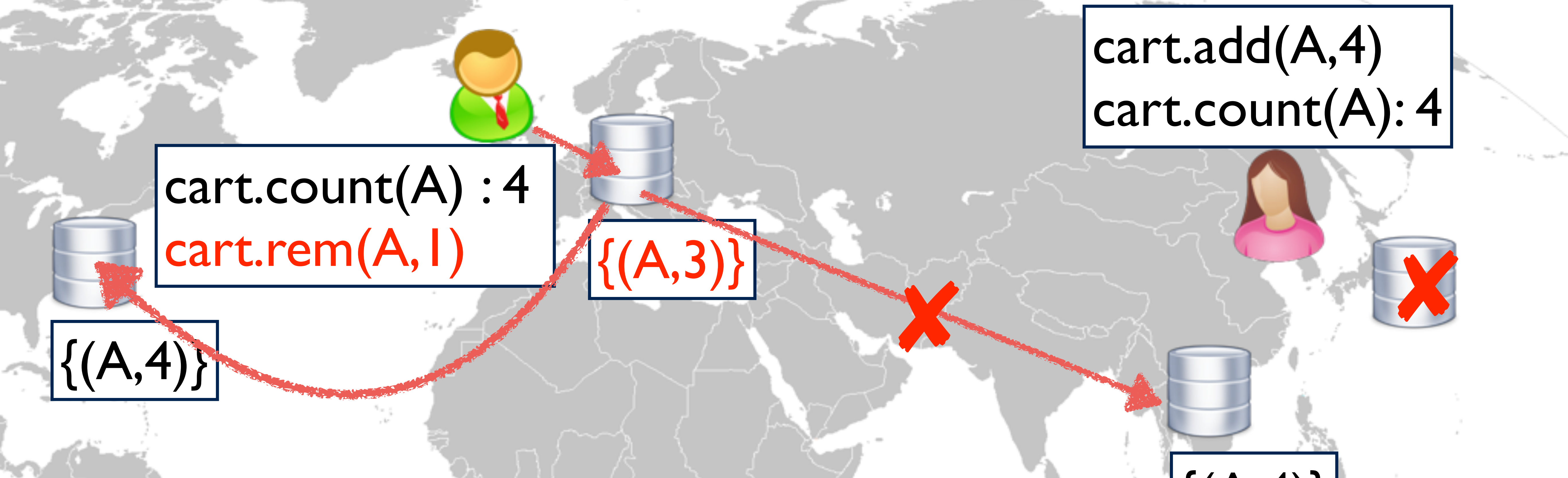
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

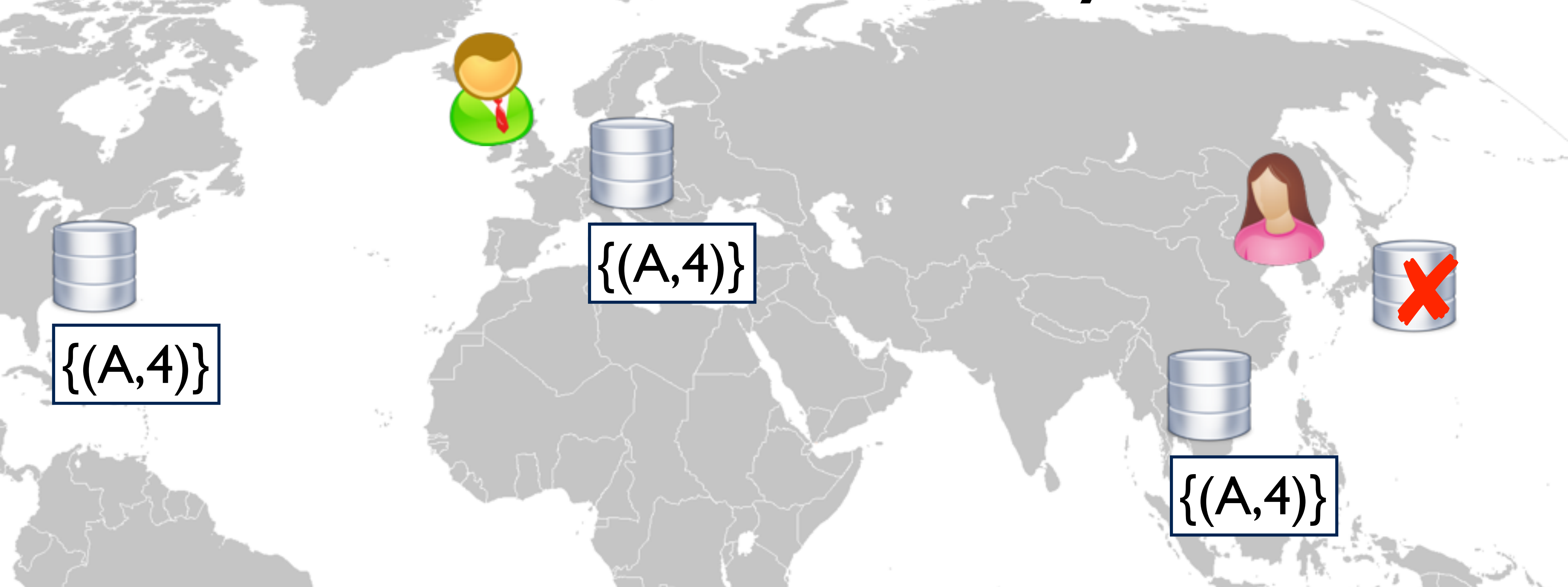
Data consistency



Issue 2: Cannot tolerate network partition

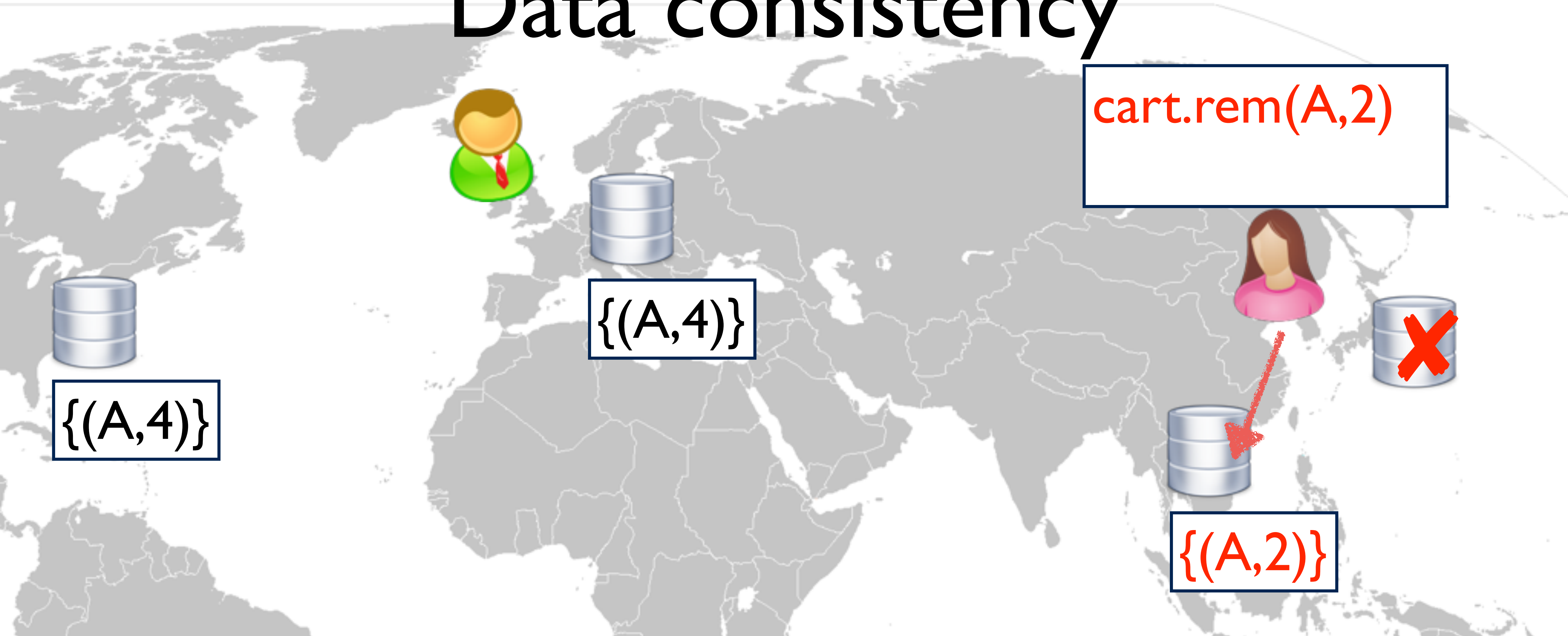
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



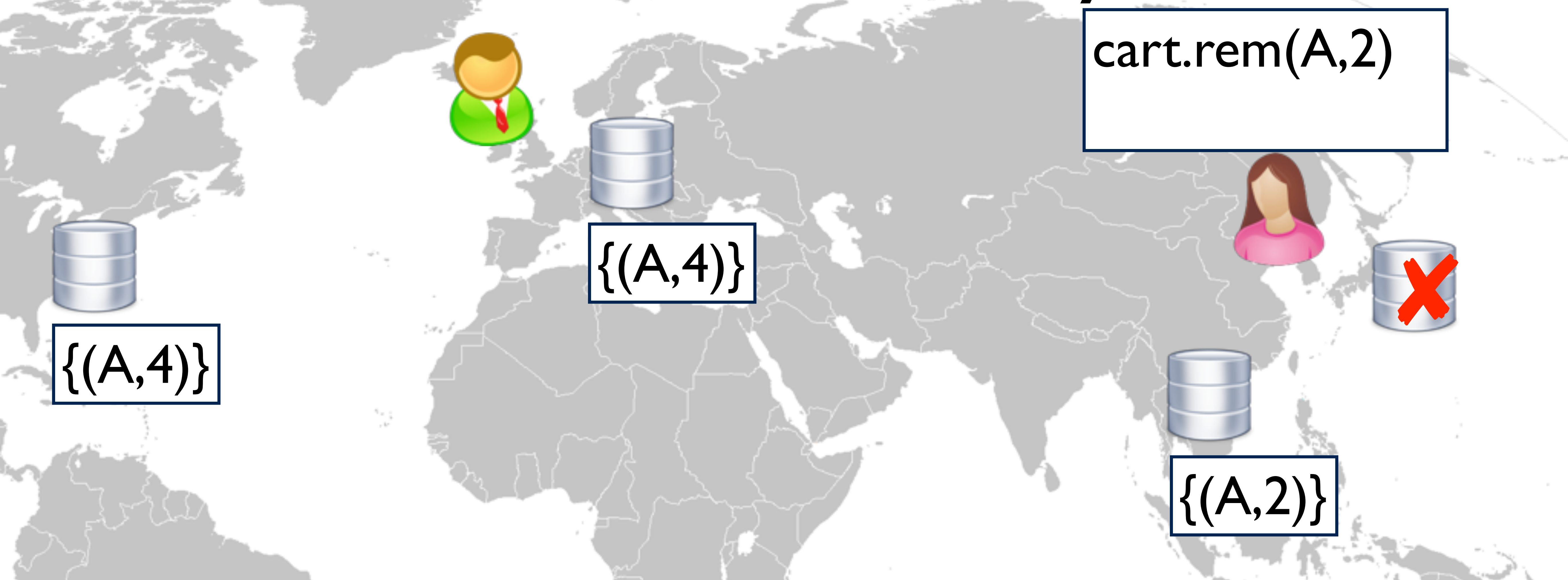
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

Data consistency



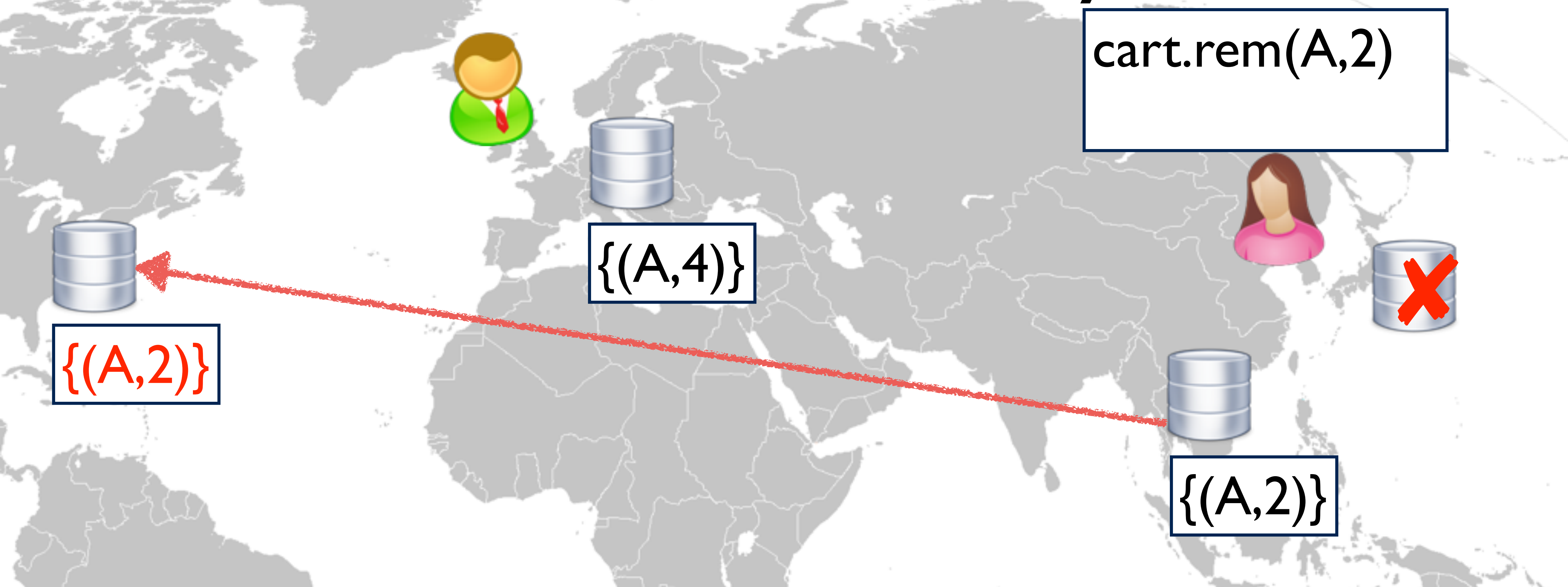
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

Data consistency



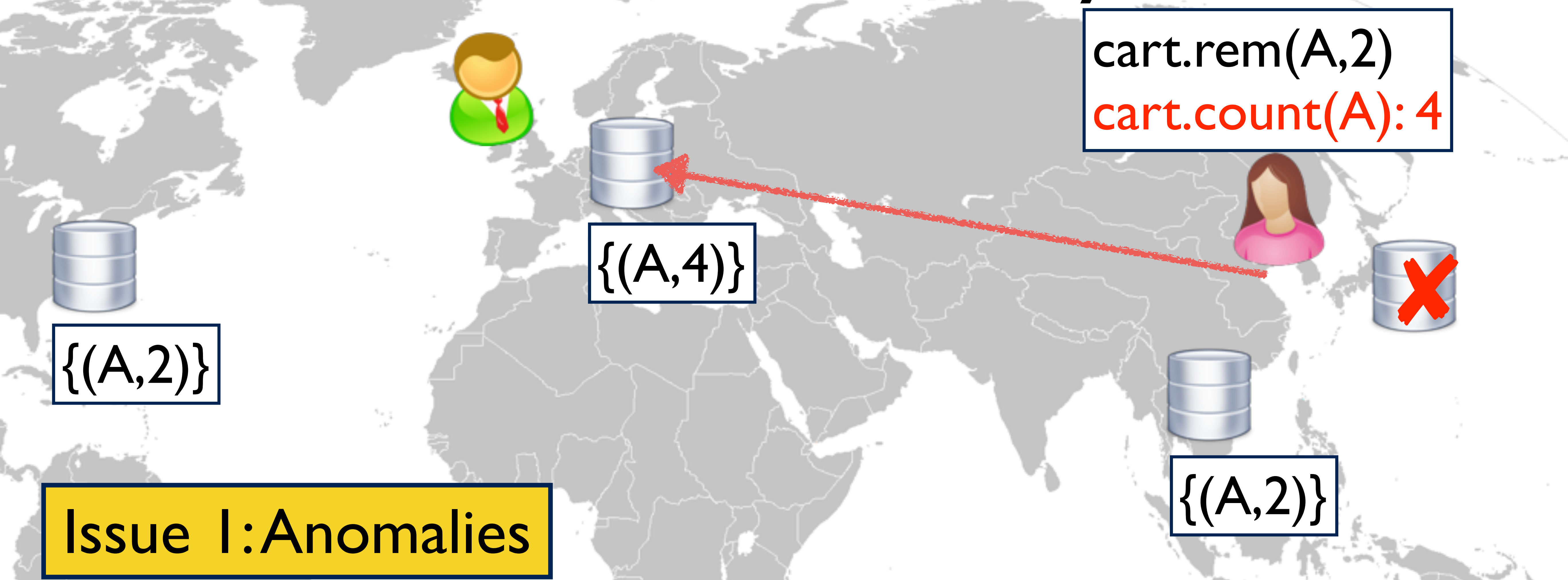
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

Data consistency



- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

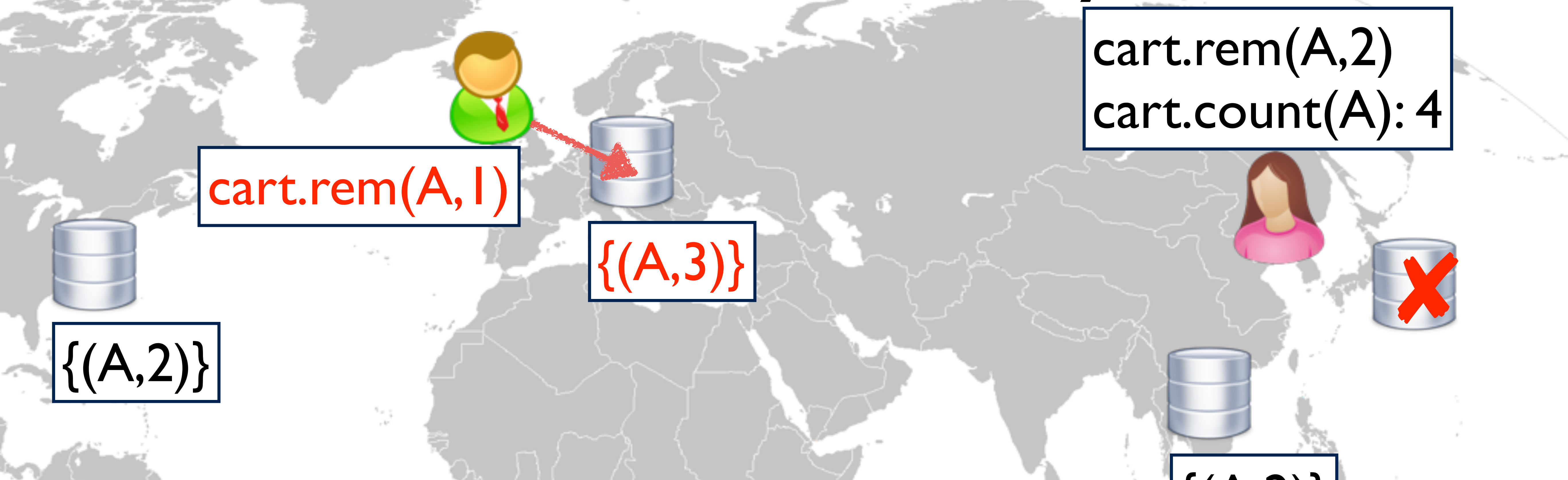
Data consistency



Issue 1: Anomalies

- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

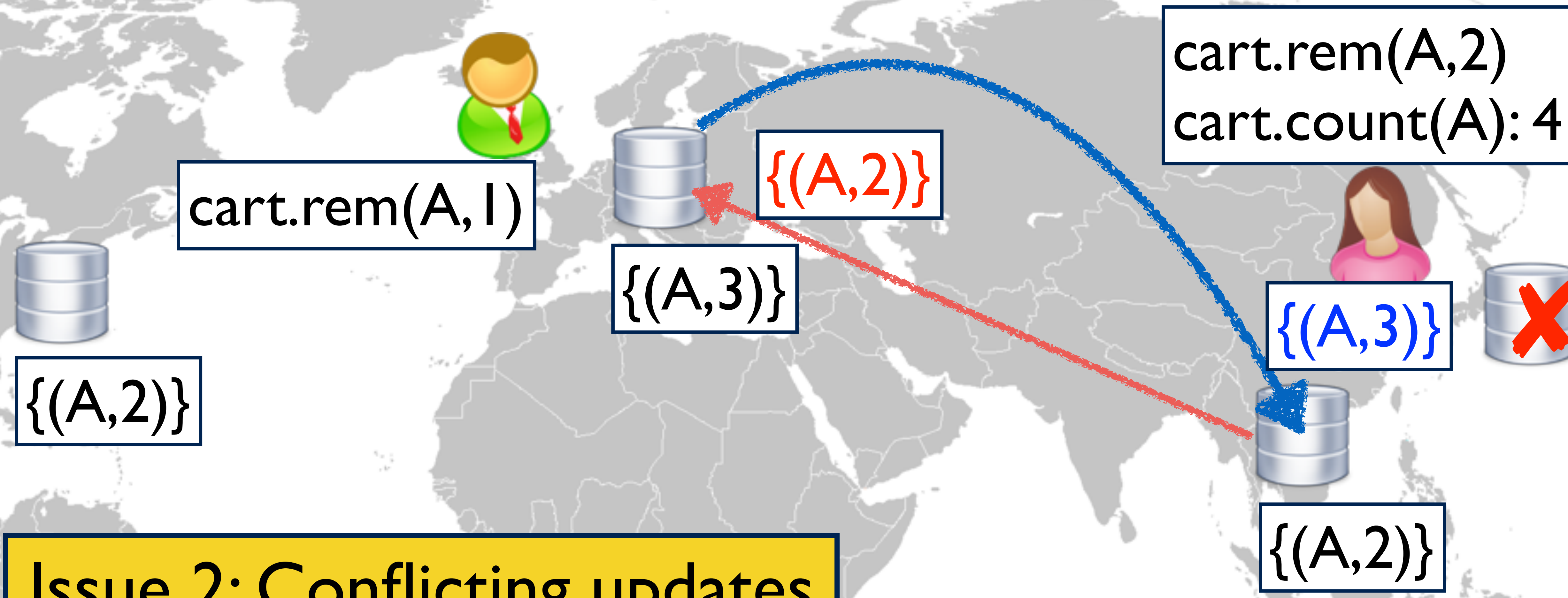
Data consistency



Issue 2: Conflicting updates

- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

Data consistency



Issue 2: Conflicting updates

- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

Popularity of weak consistency

- Dynamo/Simple DB (Amazon), Riak, Cassandra (Facebook/Twitter), CouchDB, Google Doc,etc
- Due to better responsiveness and availability.
- But created a programming challenge.

Shopping cart

- Map from book ids to natural numbers.
- Initially, constant-0 function.
- add : $\{0,\ldots,N\text{-}1\} \times \mathrm{Nat} \longrightarrow \mathrm{Unit}$
- rem : $\{0,\ldots,N\text{-}1\} \times \mathrm{Nat} \longrightarrow \mathrm{Unit}$
- count : $\{0,\ldots,N\text{-}1\} \longrightarrow \mathrm{Nat}$

Sequential data type

```
class Cart {
  val map = new Array[Nat](N)

  def add(b,n) = { val m = map(b); map(b) = m+n }
  def rem(b,n) = { val m = map(b); map(b) = max(m-n,0) }
  def count(b) = { val m = map(b); return m }
}
```

Sequential data type

```
class Cart {
  val map = new Array[Nat](N)

  def add(b,n) = { val m = map(b); map(b) = m+n }
  def rem(b,n) = { val m = map(b); map(b) = max(m-n,0) }
  def count(b) = { val m = map(b); return m }
}
```

Sequential data type

```
class Cart {
  val map = new Array[Nat](N)

  def add(b,n) = { val m = map(b); map(b) = m+n }
  def rem(b,n) = { val m = map(b); map(b) = max(m-n,0) }
  def count(b) = { val m = map(b); return m }
}
```

Sequential data type

```
class Cart {
  val map = new Array[Nat](N)

  def add(b,n) = { val m = map(b); map(b) = m+n }
  def rem(b,n) = { val m = map(b); map(b) = max(m-n,0) }
  def count(b) = { val m = map(b); return m }
}
```

Sequential data type

```
class Cart {
  val map = new Array[Nat](N)

  def add(b,n) = { val m = map(b); map(b) = m+n }
  def rem(b,n) = { val m = map(b); map(b) = max(m-n,0) }
  def count(b) = { val m = map(b); return m }
}
```

Replicated

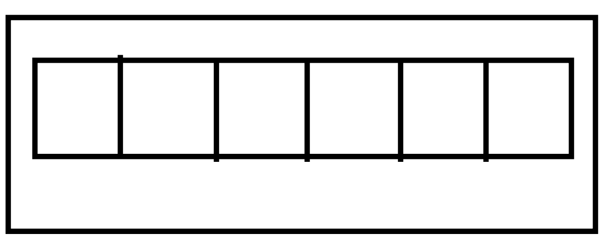
~~Sequential~~ data type

```
class Cart {
  val map = new Array[Nat](N)

  def add(b,n) = { val m = map(b); map(b) = m+n }
  def rem(b,n) = { val m = map(b); map(b) = max(m-n,0) }
  def count(b) = { val m = map(b); return m }
}
```

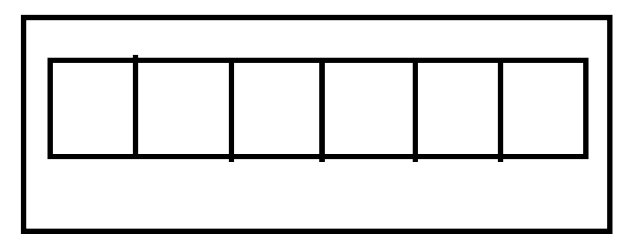
Replicated ~~Sequential~~ data type

```
class Cart {
  val map = new Array[Nat](N)

  def add(b,n) = { val m = map(b); map(b) = m+n }
  def rem(b,n) = { val m = map(b); map(b) = max(m-n,0) }
  def count(b) = { val m = map(b); return m }
}
```

cart



cart



Things to be added:

1. Asynchronous message sending & receiving.
2. Conflict detection and resolution.

Primitive replicated data types (AKA CRDT) [Shapiro+ 2011]

- Designed for weakly consistent replicated stores.
- Send & receive update messages asynchronously.
- Detect and resolve conflicts.
- Register, set, dag, graph, etc.
- Conceptually, implement a weak shared memory on top of a replicated store.

Multi-valued register

$$\text{set} : T \longrightarrow \text{Unit} \qquad \text{get} : \text{Unit} \longrightarrow \text{Set}[T]$$

- Implements a memory cell of type T.

Multi-valued register

$\text{set} : T \longrightarrow \text{Unit} \qquad \text{get} : \text{Unit} \longrightarrow \text{Set}[T]$

- Implements a memory cell of type T.
- Resolves conflicts by taking union.

Multi-valued register

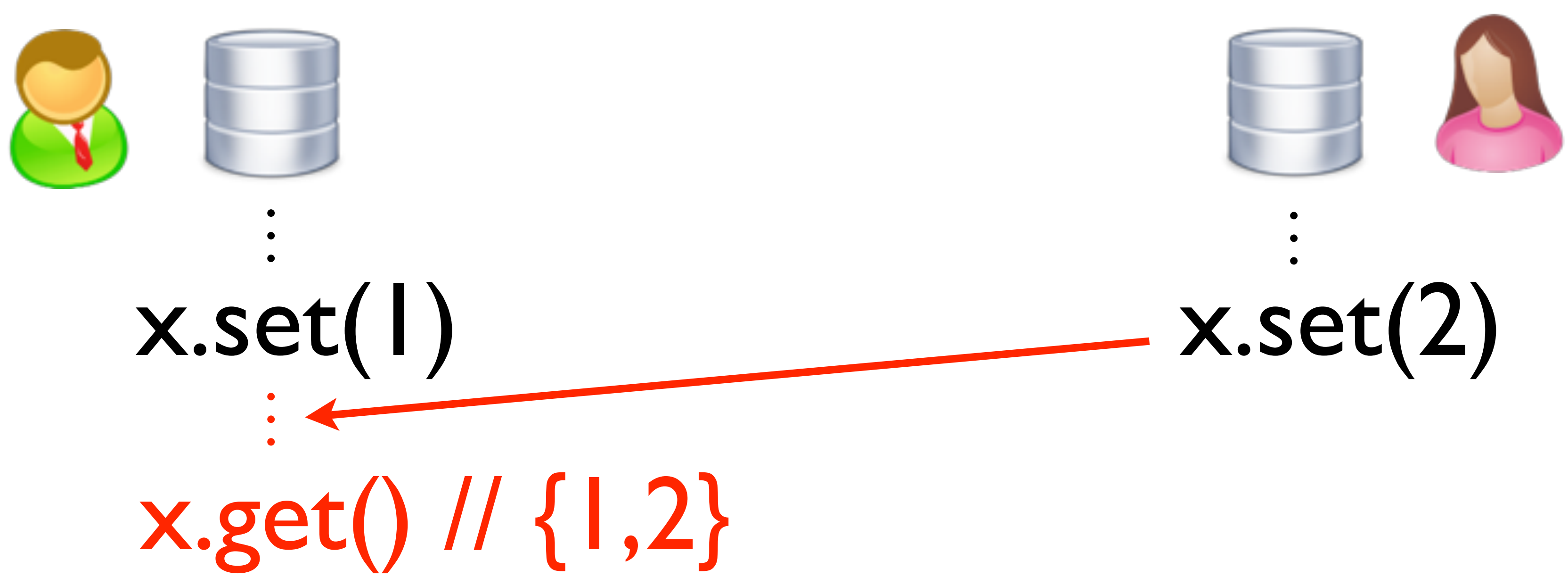
set : T $\longrightarrow$ Unit   get : Unit $\longrightarrow$ Set[T]

- Implements a memory cell of type T.
- Resolves conflicts by taking union.

Multi-valued register

set : T $\longrightarrow$ Unit   get : Unit $\longrightarrow$ Set[T]

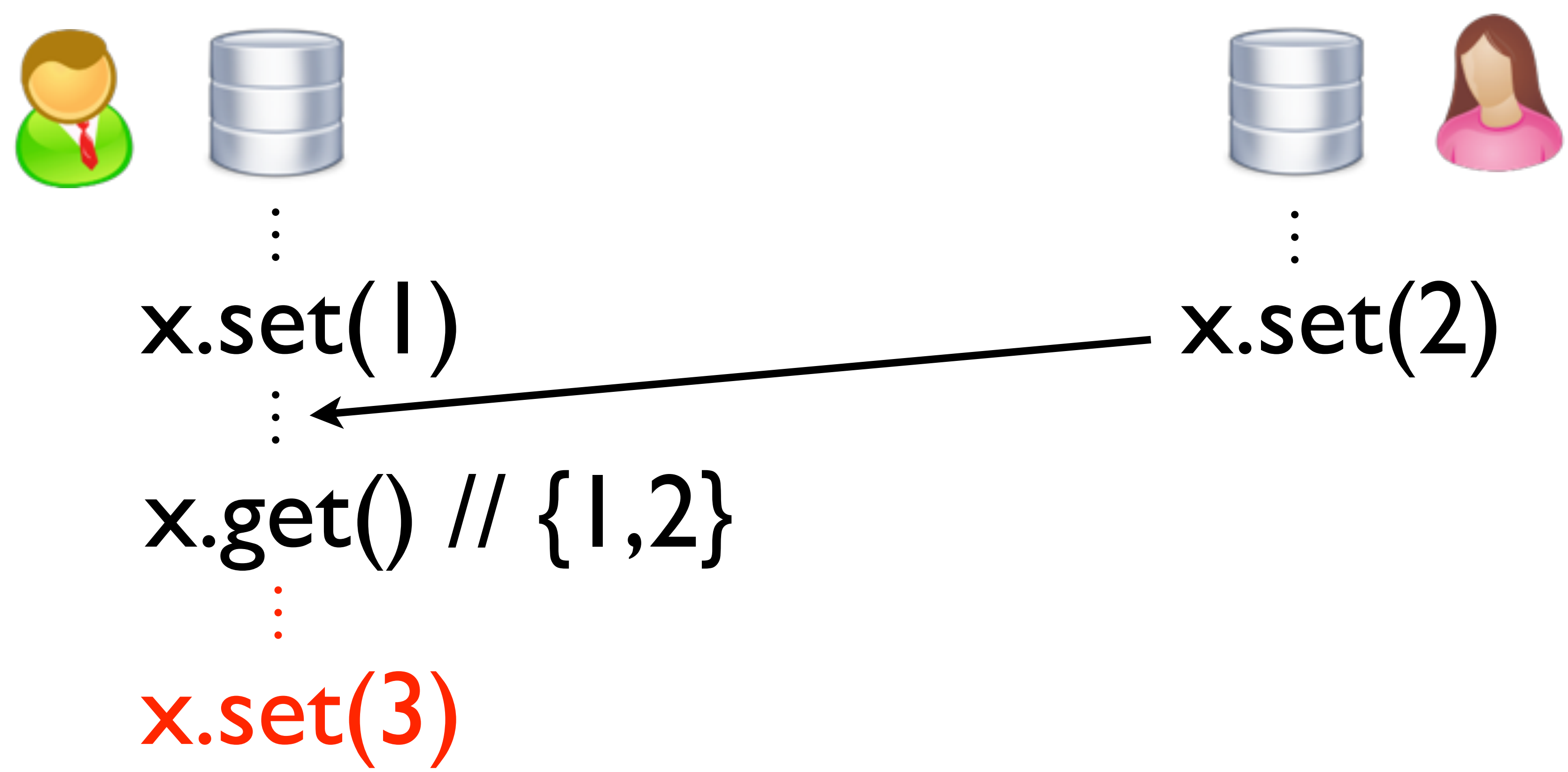
- Implements a memory cell of type T.
- Resolves conflicts by taking union.



Multi-valued register

$\text{set} : T \longrightarrow \text{Unit} \qquad \text{get} : \text{Unit} \longrightarrow \text{Set}[T]$

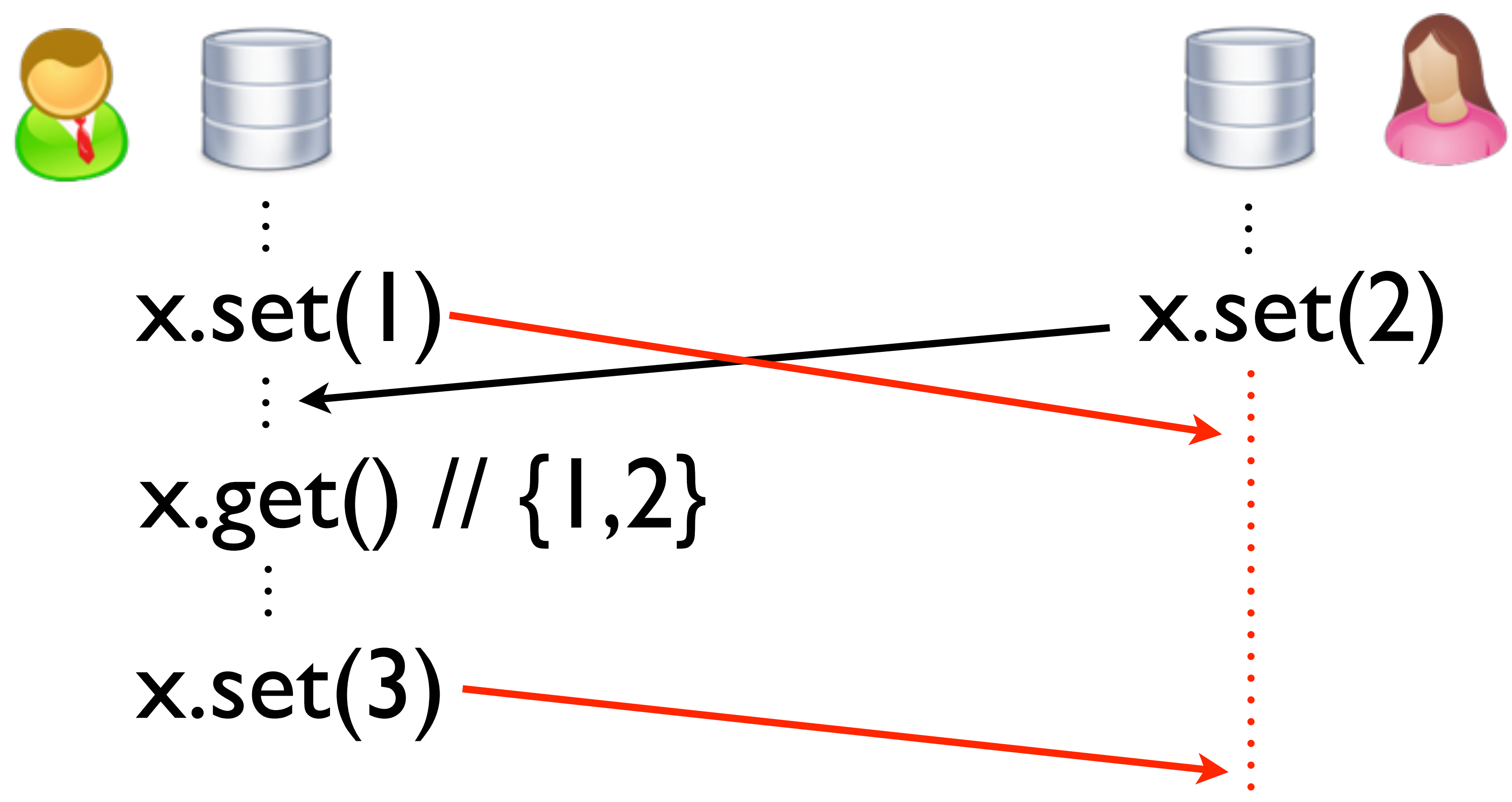
- Implements a memory cell of type T.
- Resolves conflicts by taking union.



Multi-valued register

$\text{set} : T \longrightarrow \text{Unit} \qquad \text{get} : \text{Unit} \longrightarrow \text{Set}[T]$

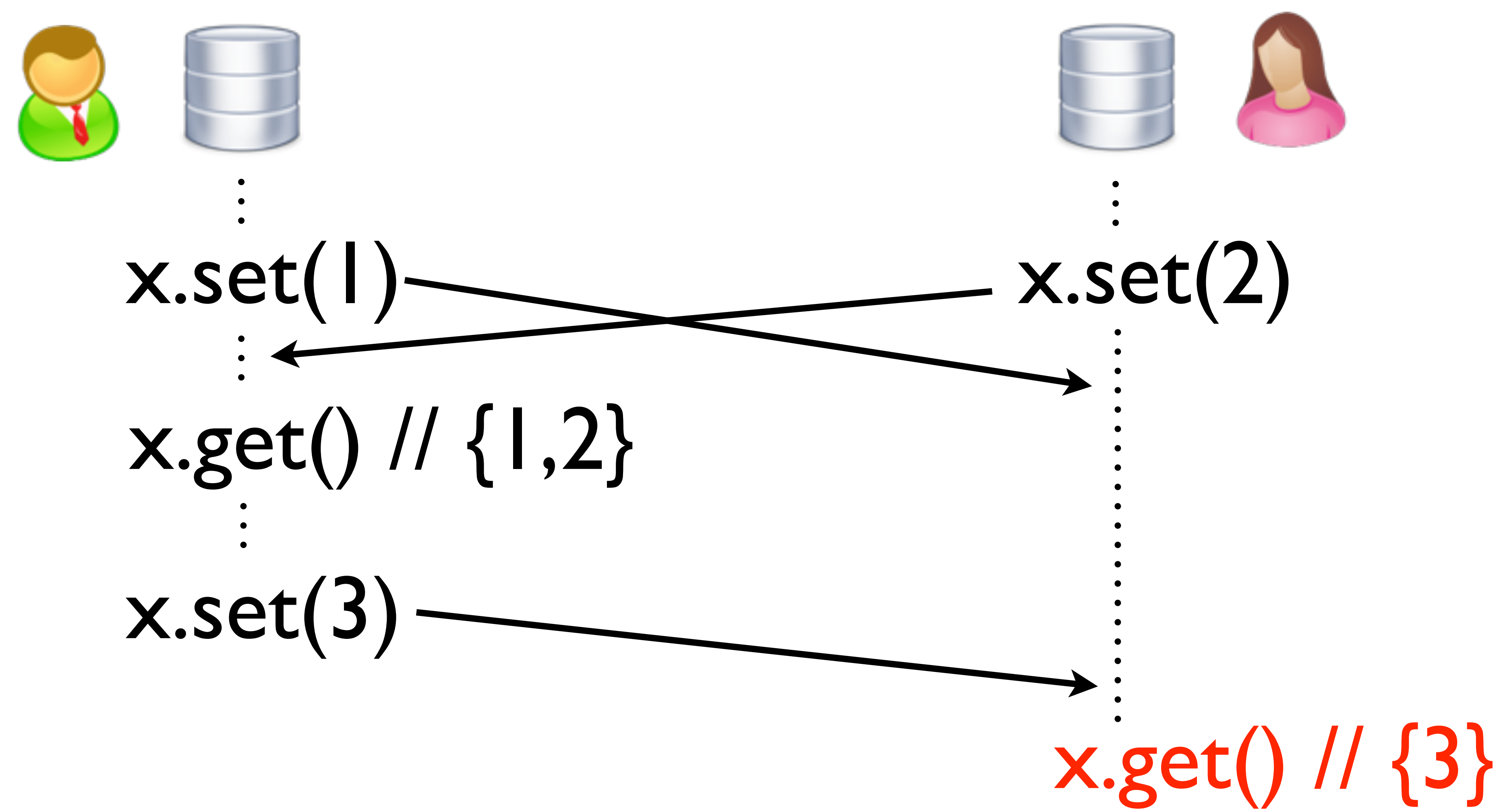
- Implements a memory cell of type T.
- Resolves conflicts by taking union.



Multi-valued register

set : T $\longrightarrow$ Unit   get : Unit $\longrightarrow$ Set[T]

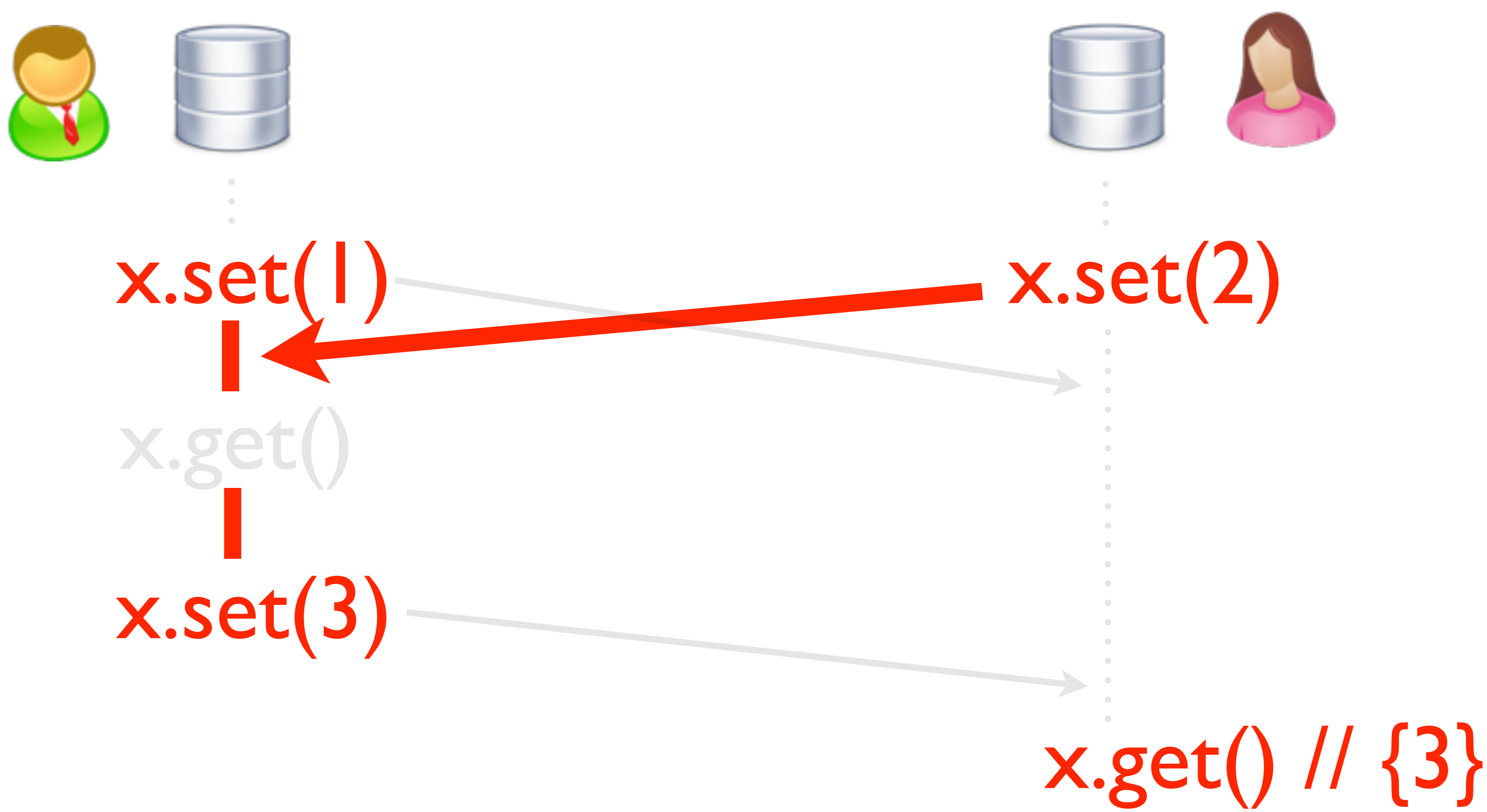
- Implements a memory cell of type T.
- Resolves conflicts by taking union.



Multi-valued register

set : T ⟶ Unit   get : Unit ⟶ Set[T]

- Implements a memory cell of type T.
- Resolves conflicts by taking union.



Shopping cart for replicated stores

```
class Cart {
  val map = new Array[Nat](N)


  def add(b,n) = {
    val m = map(b)
    map(b) = m+n
  }
  def rem(b,n) = {
    val m = map(b)
    map(b) = max(m-n,0)
  }
  def count(b) = {
    val m = map(b)
    return m
  }
}
```

Shopping cart for replicated stores

```
class Cart {
  val map = new Array[MVRegister[Nat]](N)
  for(i <- 0 until N) map(i) = new MVRegister[Nat]

  def add(b,n) = {
    val m = map(b).get()
    map(b).set(m+n)
  }
  def rem(b,n) = {
    val m = map(b).get()
    map(b).set(max(m-n,0))
  }
  def count(b) = {
    val m = map(b).get()
    return m
  }
}
```

1. Use MVRegister.

Shopping cart for replicated stores

```
class Cart {
  val map = new array[MVRegister[Nat]](N)
  for(i <- 0 until N) map(i) = new MVRegister[Nat]

  def add(b,n) = {
    val m = sum(map(b).get())
    map(b).set(m+n)
  }
  def rem(b,n) = {
    val m = sum(map(b).get())
    map(b).set(max(m-n,0))
  }
  def count(b) = {
    val m = sum(map(b).get())
    return m
  }
}
```

1. Use MVRegister.
2. Add conflict resolution.

Shopping cart for replicated stores

```
class Cart {
  val map = new array[MVRegister[Nat]](N)
  for(i <- 0 until N) map(i) = new MVRegister[Nat]

  def add(b,n) = {
    val m = max(map(b).get())
    map(b).set(m+n)
  }
  def rem(b,n) = {
    val m = max(map(b).get())
    map(b).set(max(m-n,0))
  }
  def count(b) = {
    val m = max(map(b).get())
    return m
  }
}
```

1. Use MVRegister.
2. Add conflict resolution.

Semantics of replicated data types

Dag[Op] = Set of Op-labelled finite dags

$$F : \text{Dag[Op]} \times \text{Op} \longrightarrow_{\text{partial}} \text{Value}$$

$$F_m\left(\begin{tikzcd} & \text{set}(2) \arrow[dl] \arrow[dd] \\ \text{set}(3) \arrow[dr] & \\ & \text{set}(4) \end{tikzcd} ,\quad \text{get}() \right) = \{4\}$$

Semantics of replicated data types

Dag[Op] = Set of Op-labelled finite dags

$$F : \mathrm{Dag[Op]} \times \mathrm{Op} \longrightarrow_{\mathrm{partial}} \mathrm{Value}$$

$$F_m\left(\begin{tikzcd} & \mathrm{set}(2) \arrow[dl] \arrow[dd] \\ \mathrm{set}(3) \arrow[dr] & \\ & \mathrm{set}(4) \end{tikzcd} ,\quad \mathrm{get}() \right) = \{4\}$$

Semantics of replicated data types

Dag[Op] = Set of Op-labelled finite dags

$$F : \mathrm{Dag}[\mathrm{Op}] \times \mathrm{Op} \longrightarrow_{\mathrm{partial}} \mathrm{Value}$$

$$F_m\left(\begin{tikzcd} & \mathrm{set}(2) \arrow[dl] \arrow[dd] \\ \mathrm{set}(3) \arrow[dr] & \\ & \mathrm{set}(4) \end{tikzcd} \;,\quad \mathrm{get}() \right) = \{4\}$$

Semantics of replicated data types

Dag[Op] = Set of Op-labelled finite dags

$$F : \text{Dag}[\text{Op}] \times \text{Op} \longrightarrow_{\text{partial}} \text{Value}$$

$$F_m\left(\begin{tikzcd} & \text{set}(2) \arrow[dl] \arrow[dd] \\ \text{set}(3) \arrow[dr] & \\ & \text{set}(4) \end{tikzcd} , \quad \text{get}() \right) = \{4\}$$

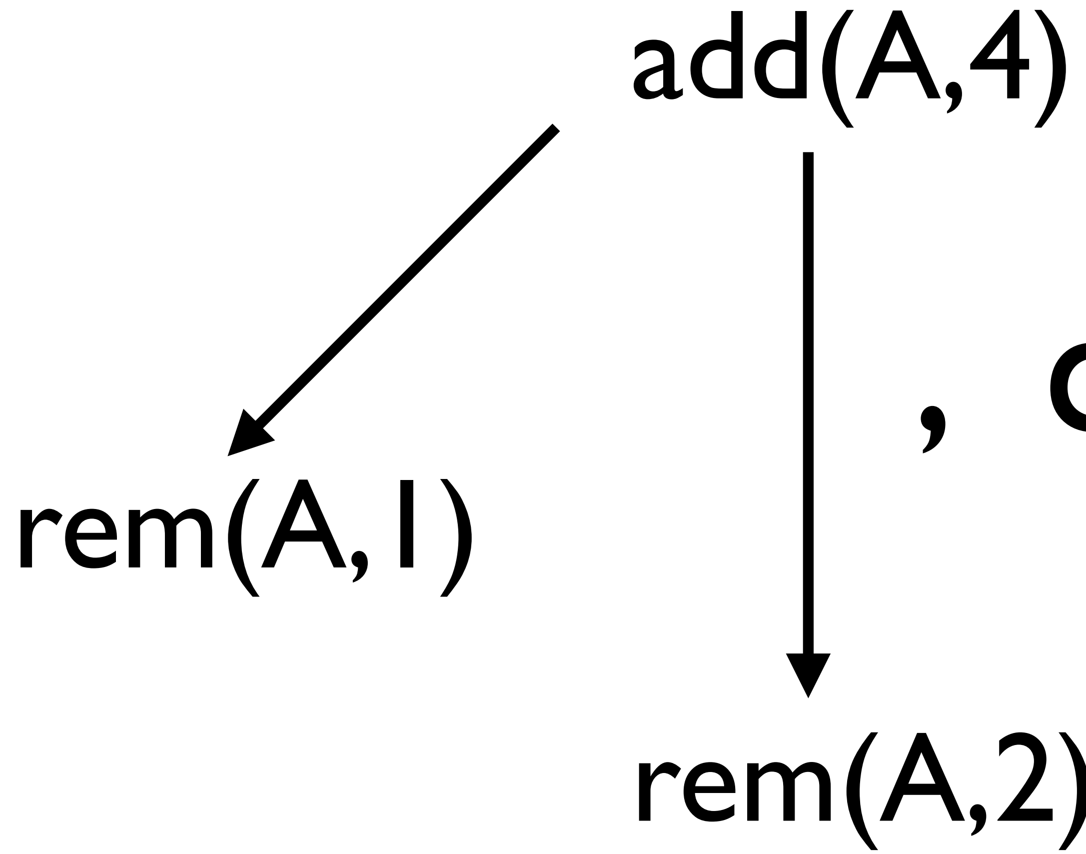
Semantics of replicated data types

Dag[Op] = Set of Op-labelled finite dags

$$F : \mathrm{Dag}[\mathrm{Op}] \times \mathrm{Op} \longrightarrow_{\mathrm{partial}} \mathrm{Value}$$

$$F_m\left(\begin{tikzcd} & \mathrm{set}(2) \arrow[dl] \arrow[dd] \\ \mathrm{set}(3) & \\ & \mathrm{set}(4) \end{tikzcd},\ \mathrm{get}()\right) = \{3,4\}$$

```
val map = new array[MVRegister[Nat]](N)
. . . . .
def add(b,n)={ val m = max(map(b).get()); map(b).set(m+n) }
def rem(b,n)={ val m = max(map(b).get()); map(b).set(max(m-n,0)) }
def count(b)={ val m = max(map(b).get()); return m }
```

$F_s($ 

$, \text{count}()) = ?$

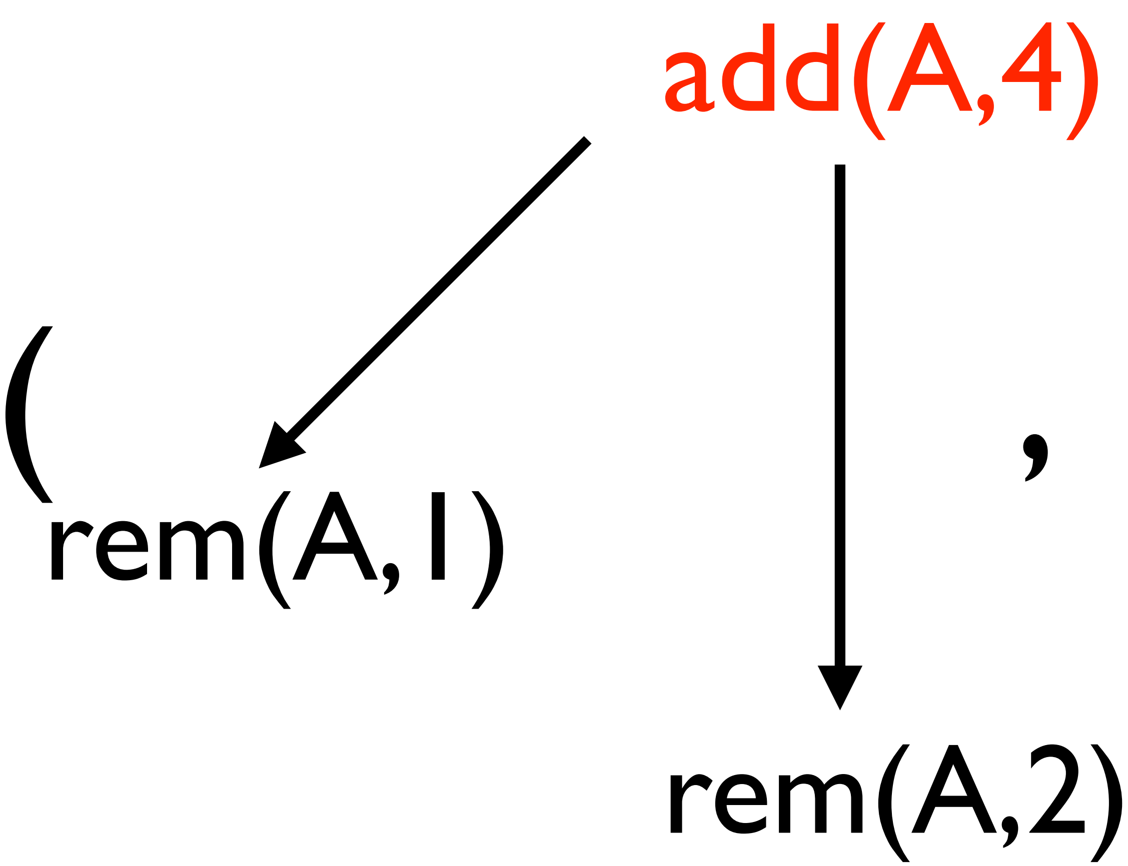
```
val map = new array[MVRegister[Nat]](N)
. . . . .
def add(b,n)={ val m = max(map(b).get()); map(b).set(m+n) }
def rem(b,n)={ val m = max(map(b).get()); map(b).set(max(m-n,0)) }
def count(b)={ val m = max(map(b).get()); return m }
```

$$F_s\left(\begin{tikzcd} & \text{add(A,4)} \arrow[dl] \arrow[d] \\ \text{rem(A,1)} & \text{rem(A,2)} \end{tikzcd},\ \text{count()}\right) = ?$$

```
val map = new array[MVRegister[Nat]](N)
. . . . .
def add(b,n)={ val m = max(map(b).get()); map(b).set(m+n) }
def rem(b,n)={ val m = max(map(b).get()); map(b).set(max(m-n,0)) }
def count(b)={ val m = max(map(b).get()); return m }
```

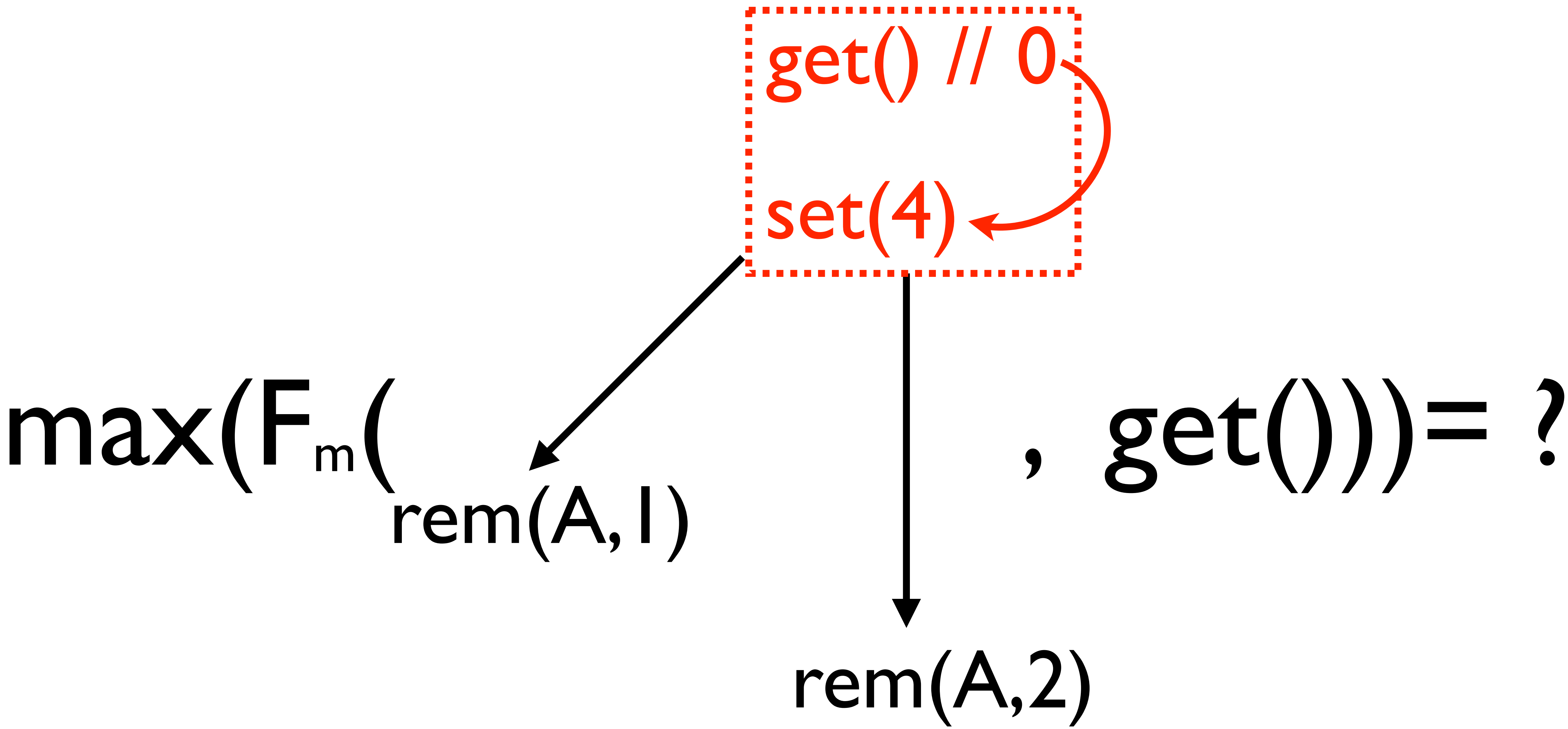
$$\max\left(F_m\left(\begin{tikzcd} & \mathrm{add}(A,4) \arrow[dl] \arrow[d] \\ \mathrm{rem}(A,1) & \mathrm{rem}(A,2) \end{tikzcd},\ \mathrm{get}()\right)\right) = ?$$

```
val map = new array[MVRegister[Nat]](N)
. . . . .
def add(b,n)={ val m = max(map(b).get()); map(b).set(m+n) }
def rem(b,n)={ val m = max(map(b).get()); map(b).set(max(m-n,0)) }
def count(b)={ val m = max(map(b).get()); return m }
```

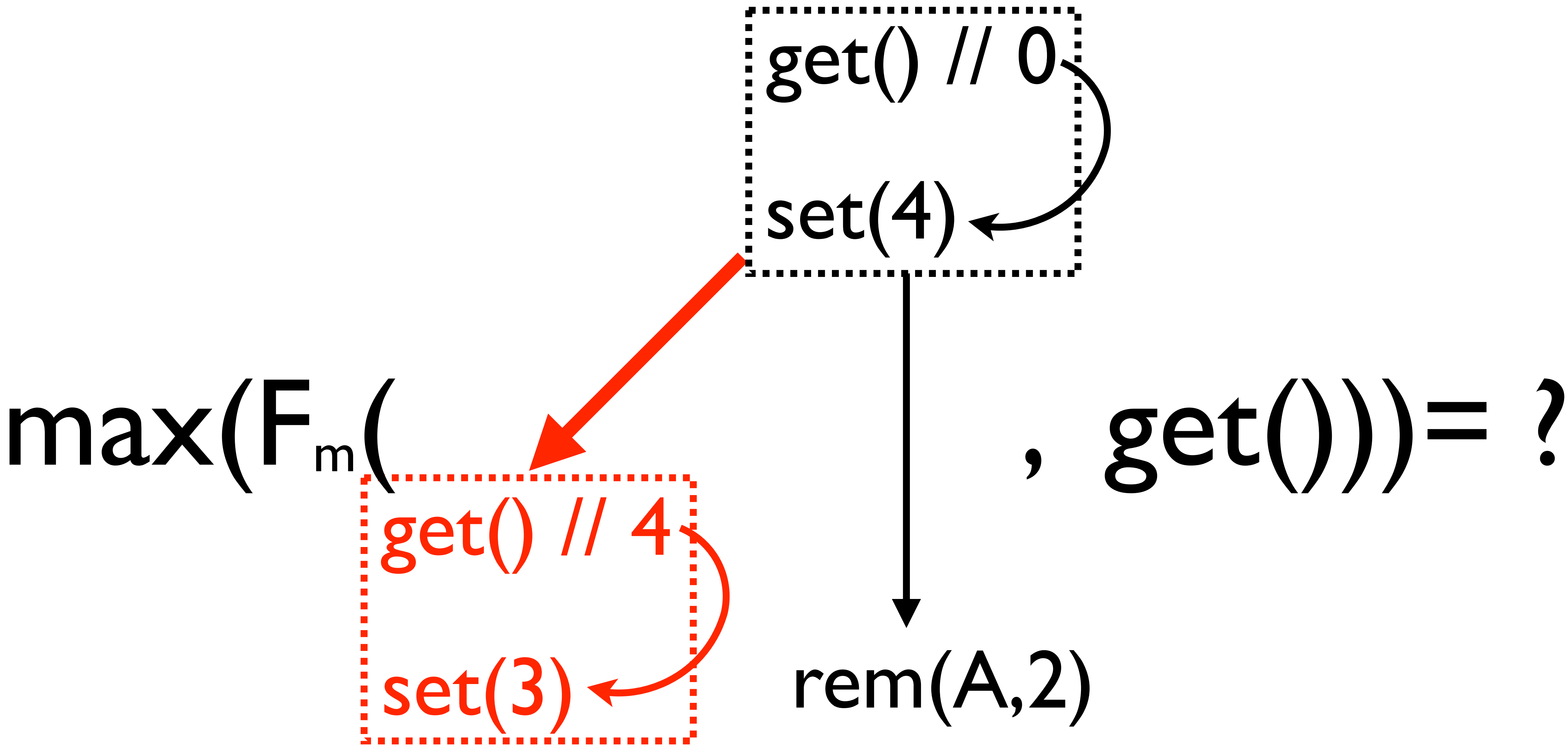
$\max(F_m($ 

$, \ get()))= ?$

```
val map = new array[MVRegister[Nat]](N)
. . . . .
def add(b,n)={ val m = max(map(b).get()); map(b).set(m+n) }
def rem(b,n)={ val m = max(map(b).get()); map(b).set(max(m-n,0)) }
def count(b)={ val m = max(map(b).get()); return m }
```

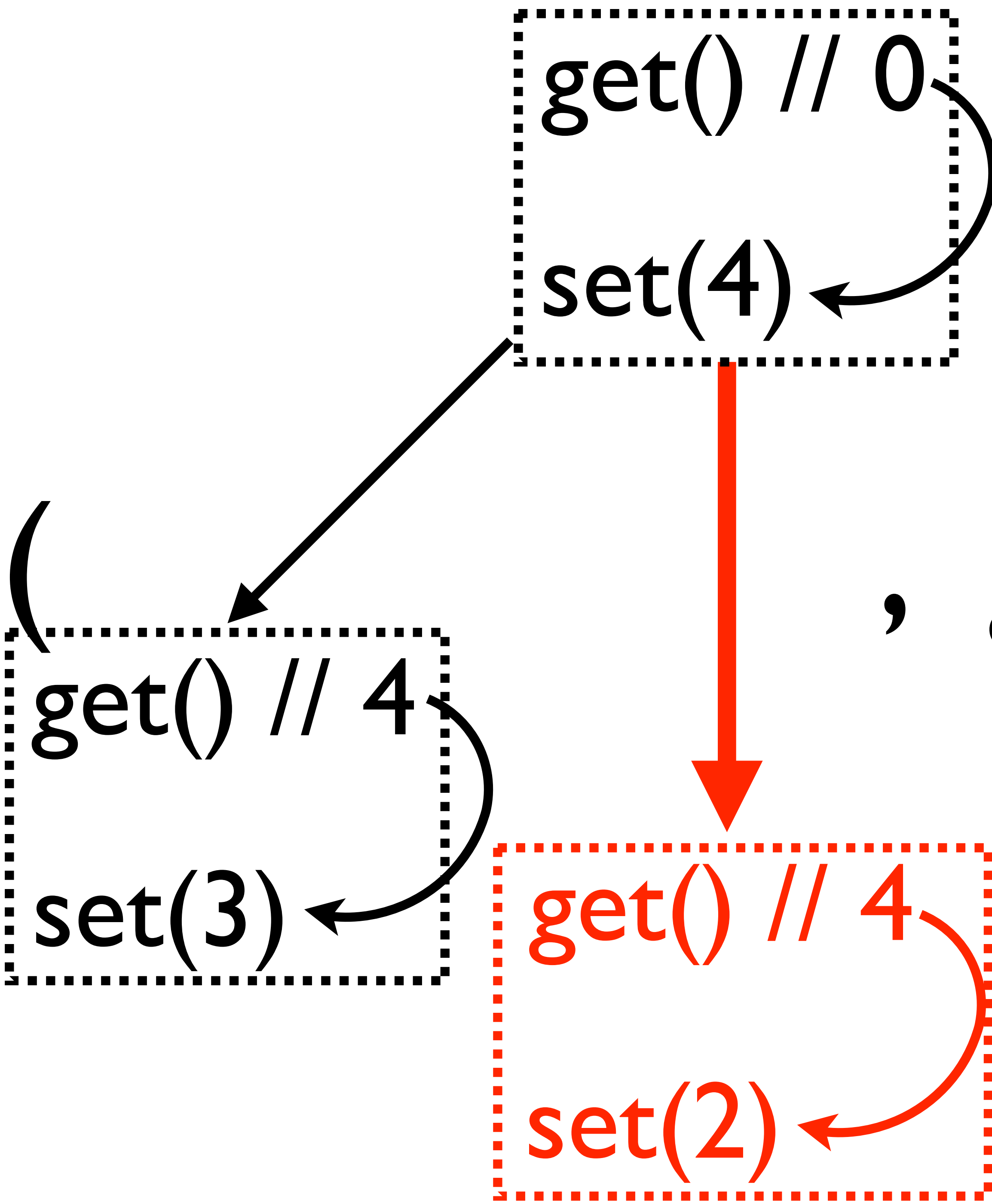


```
val map = new array[MVRegister[Nat]](N)
. . . . .
def add(b,n)={ val m = max(map(b).get()); map(b).set(m+n) }
def rem(b,n)={ val m = max(map(b).get()); map(b).set(max(m-n,0)) }
def count(b)={ val m = max(map(b).get()); return m }
```

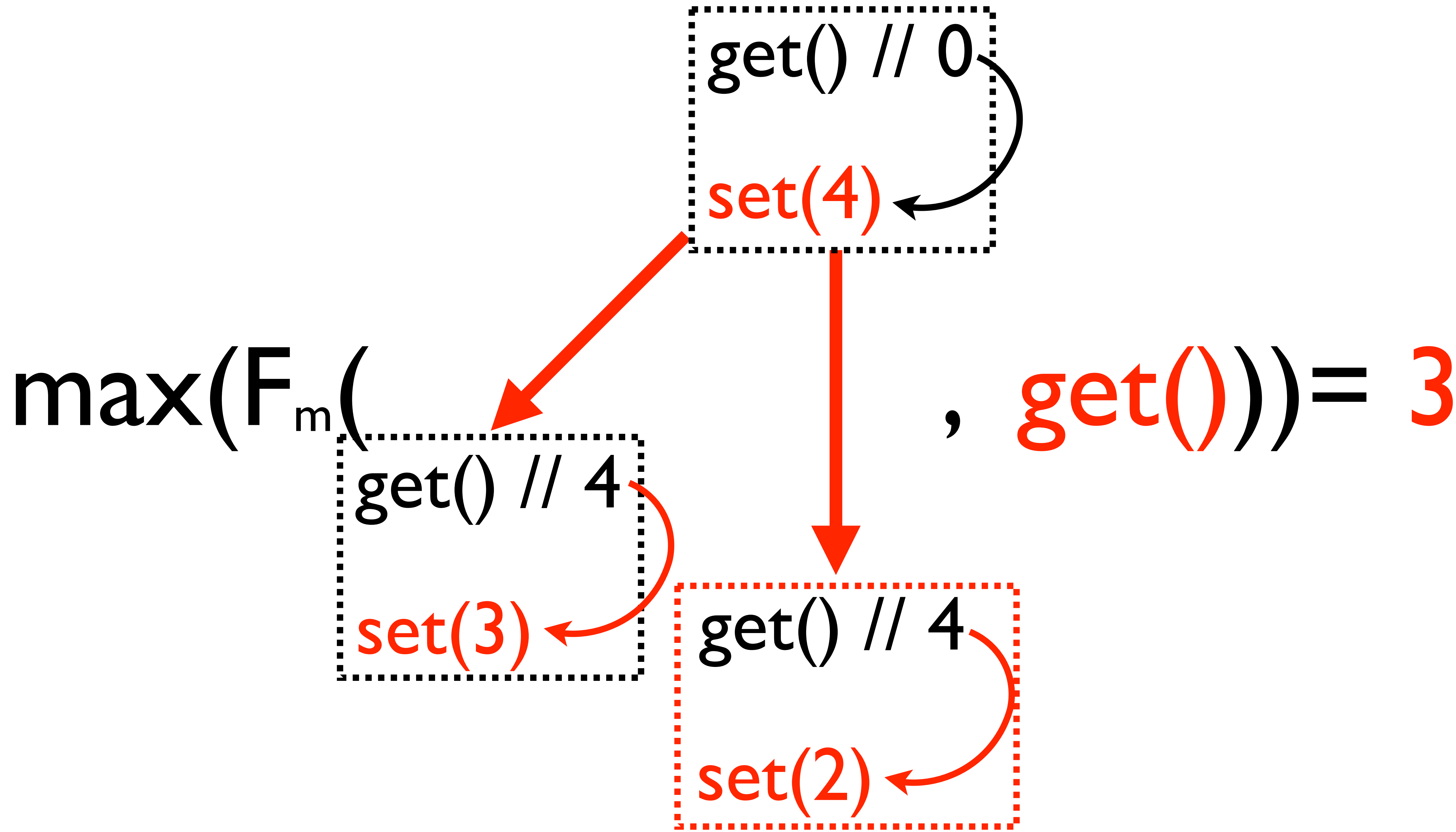


```
val map = new array[MVRegister[Nat]](N)
. . . . .
def add(b,n)={ val m = max(map(b).get()); map(b).set(m+n) }
def rem(b,n)={ val m = max(map(b).get()); map(b).set(max(m-n,0)) }
def count(b)={ val m = max(map(b).get()); return m }
```

$$\max(F_m(\quad\quad , \; get())) = \,?$$



```
val map = new array[MVRegister[Nat]](N)
. . . . .
def add(b,n)={ val m = max(map(b).get()); map(b).set(m+n) }
def rem(b,n)={ val m = max(map(b).get()); map(b).set(max(m-n,0)) }
def count(b)={ val m = max(map(b).get()); return m }
```



```
val map = new array[MVRegister[Nat]](N)
. . . . .
def add(b,n)={ val m = max(map(b).get()); map(b).set(m+n) }
def rem(b,n)={ val m = max(map(b).get()); map(b).set(max(m-n,0)) }
def count(b)={ val m = max(map(b).get()); return m }
```

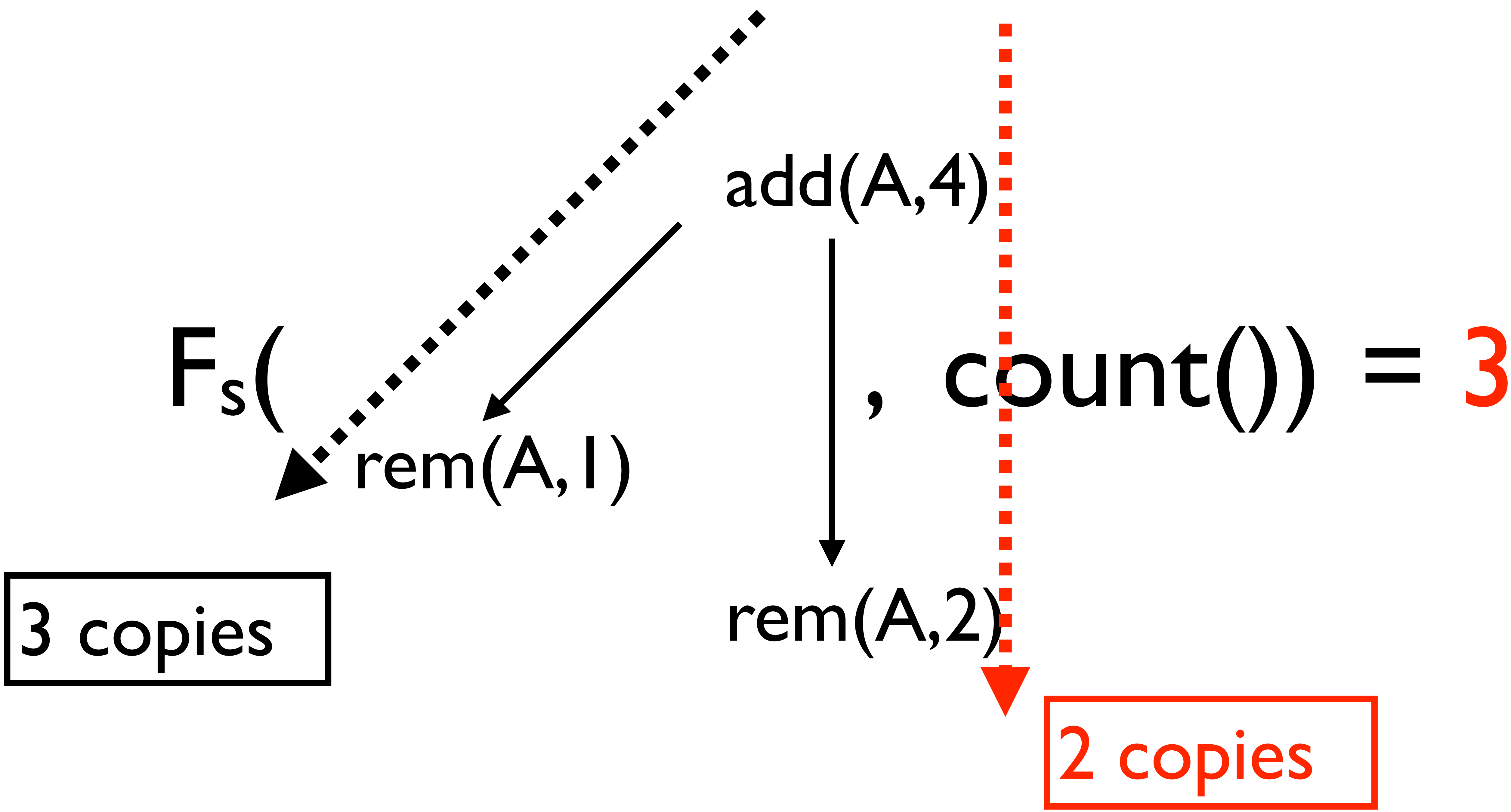
$$F_s\left(\begin{tikzcd} & \mathrm{add}(A,4) \arrow[dl] \arrow[d] \\ \mathrm{rem}(A,1) & \mathrm{rem}(A,2) \end{tikzcd},\ \mathrm{count}()\right) = 3$$

```
val map = new array[MVRegister[Nat]](N)
. . . . .
def add(b,n)={ val m = max(map(b).get()); map(b).set(m+n) }
def rem(b,n)={ val m = max(map(b).get()); map(b).set(max(m-n,0)) }
def count(b)={ val m = max(map(b).get()); return m }
```

$$F_s\left(\begin{tikzcd} & \mathrm{add}(A,4) \arrow[dl] \arrow[d] \\ \mathrm{rem}(A,1) & \mathrm{rem}(A,2) \end{tikzcd},\ \mathrm{count}()\right) = 3$$

3 copies

```
val map = new array[MVRegister[Nat]](N)
. . . . .
def add(b,n)={ val m = max(map(b).get()); map(b).set(m+n) }
def rem(b,n)={ val m = max(map(b).get()); map(b).set(max(m-n,0)) }
def count(b)={ val m = max(map(b).get()); return m }
```



Semantics

- Developed F-based semantics of distributed programs with weak transactions.
- It is equivalent to a more standard semantics.
- The details are in our tech report. Email me if you are interested.

Take-home message

Developing a good programming model for weakly consistent distributed stores is a big challenge.