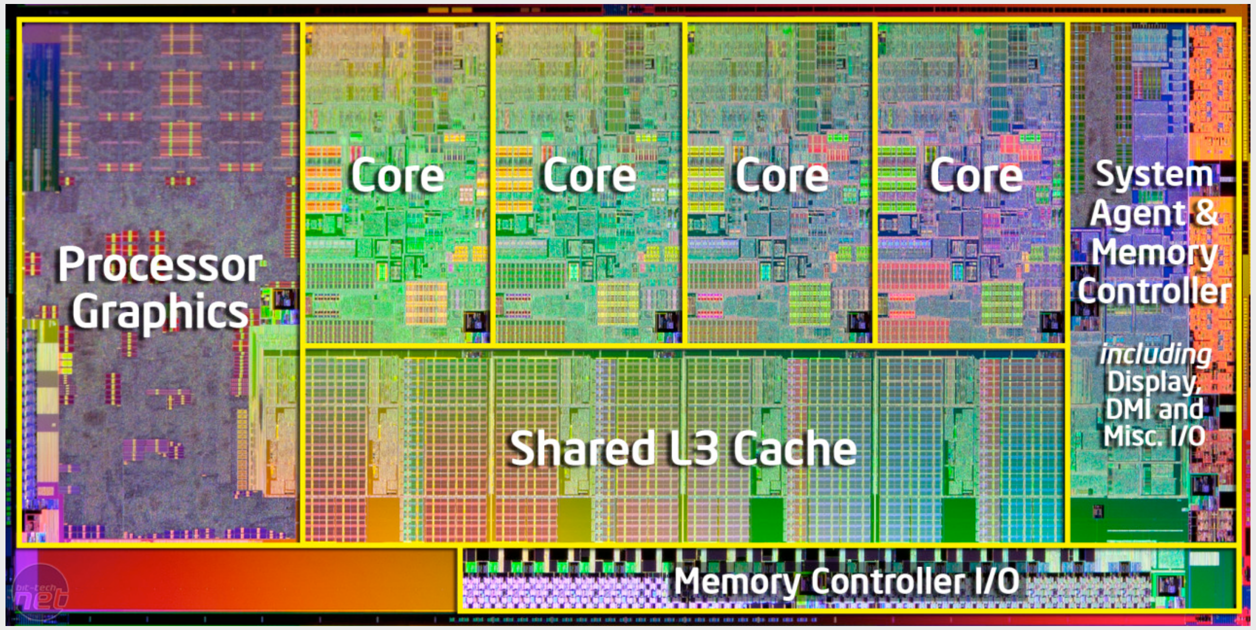
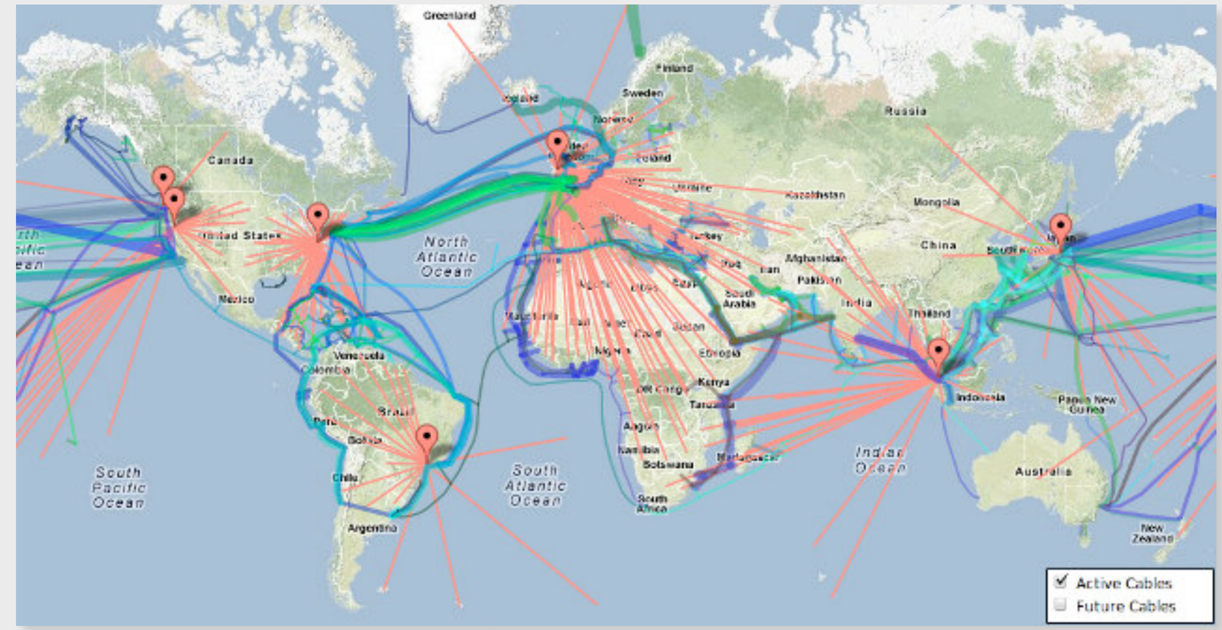
LVars for distributed programming
or, LVars and CRDTs `join` forces

Lindsey Kuper
Programming Systems Lab, Intel Labs

IFIP WG 2.8 Meeting
May 26, 2015





Shared-memory
parallel programming

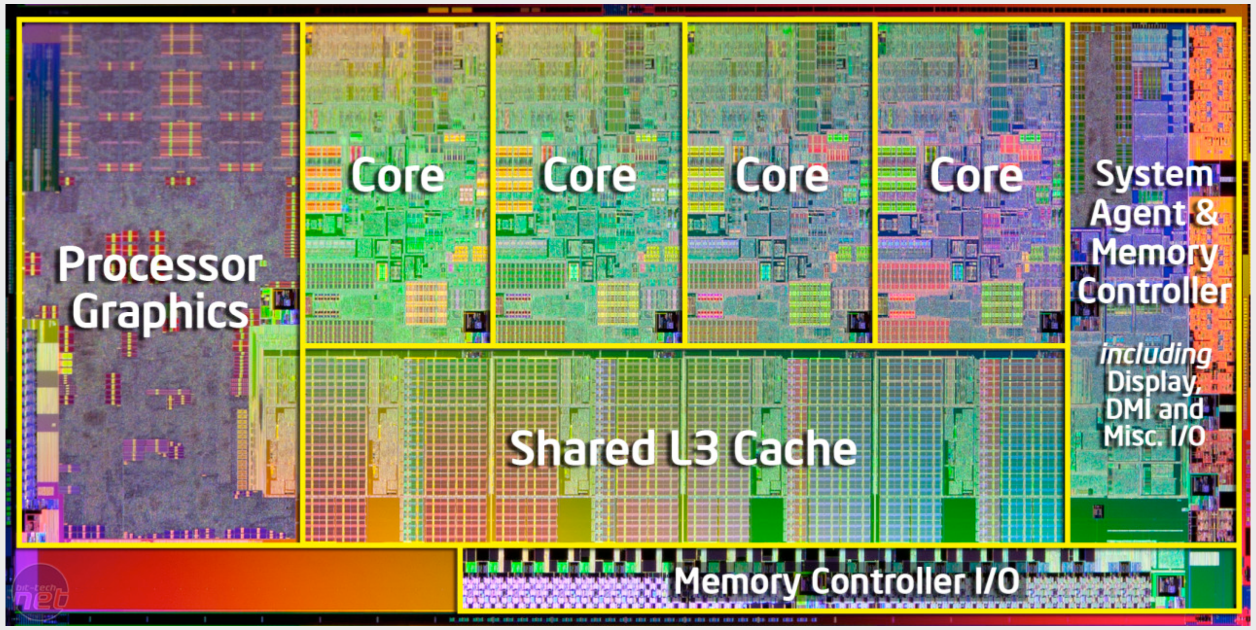
LVars

(Observable) determinism

Distributed
programming

CRDTs

(Eventual) consistency



Shared-memory
parallel programming

LVars

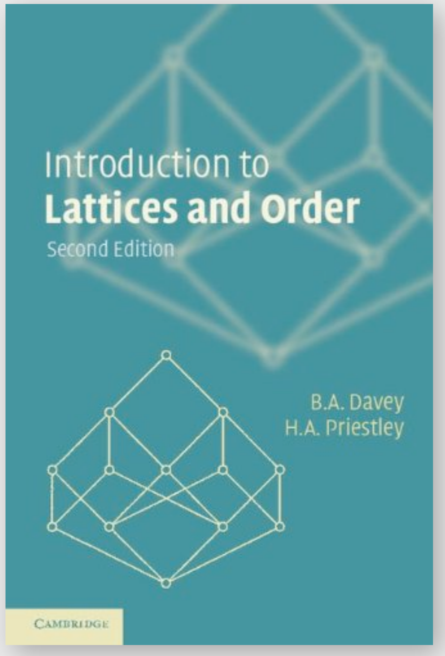
(Observable) determinism

```
data Item = Book | Shoes | ...
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```



```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

(What happens when we run this?)

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

IVars: single writes, blocking (but exact) reads
[Arvind *et al.*, 1989]

LVars: **commutative and inflationary** writes, blocking **threshold** reads

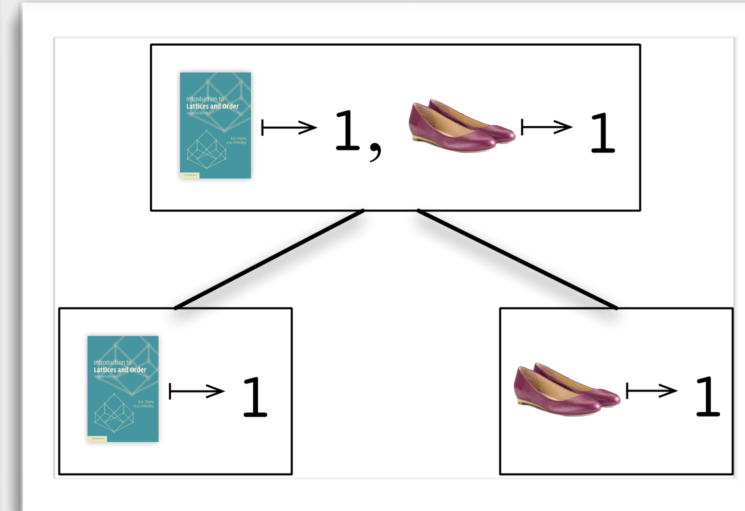
```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

IVars: single writes, blocking (but exact) reads
[Arvind *et al.*, 1989]

LVars: **commutative and inflationary** writes, blocking **threshold** reads

* actually a bounded join-semilattice

```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
       async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ())))
       res <- async (readIORef cart)
       wait res
```

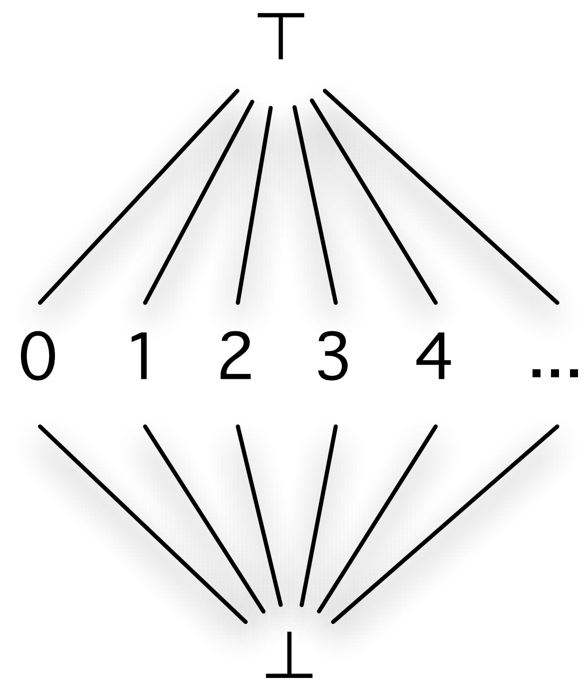


IVars: single writes, blocking (but exact) reads
[Arvind *et al.*, 1989]

LVars: **commutative and inflationary** writes, blocking **threshold** reads

* actually a bounded join-semilattice

`num`



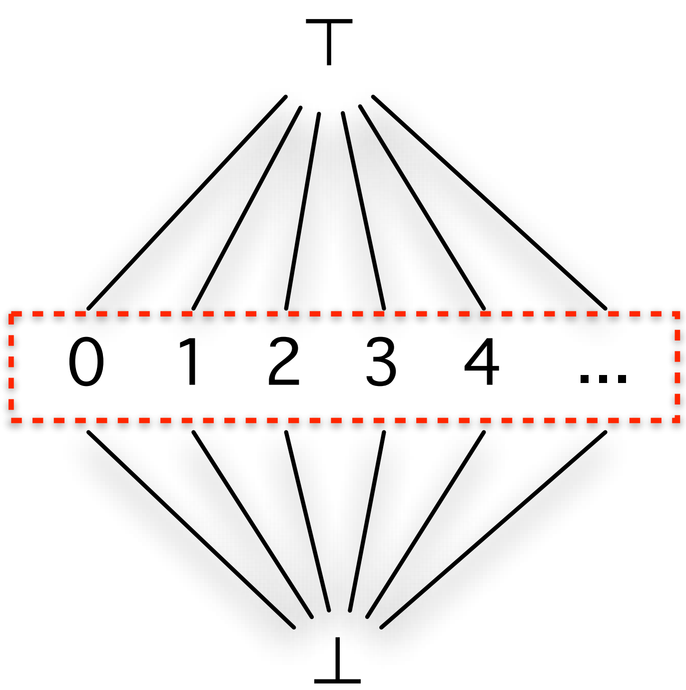
Raises an error, since $3 \sqcup 4 = \top$

```
do
  fork (put num 3)
  fork (put num 4)
```

Works fine, since $4 \sqcup 4 = 4$

```
do
  fork (put num 4)
  fork (put num 4)
```

num



Raises an error, since $3 \sqcup 4 = \top$

```
do
  fork (put num 3)
  fork (put num 4)
```

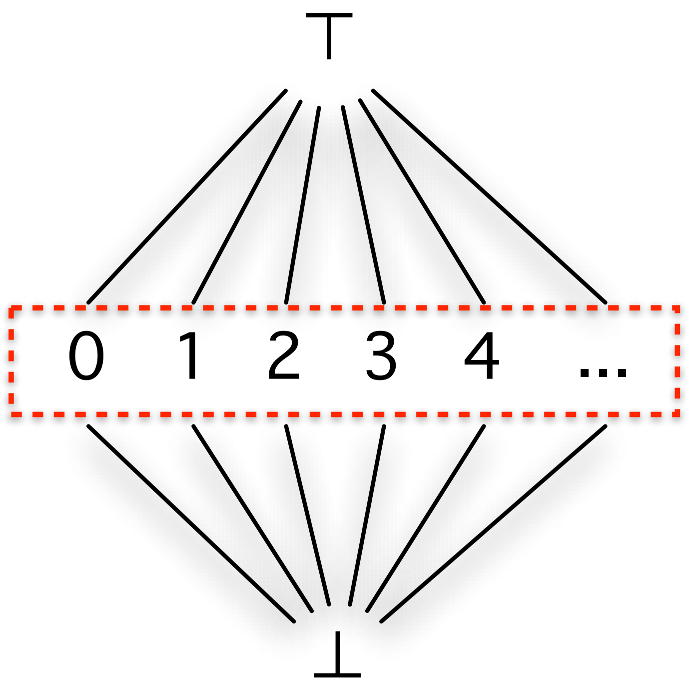
Works fine, since $4 \sqcup 4 = 4$

```
do
  fork (put num 4)
  fork (put num 4)
```

`get` blocks until threshold is reached

```
do
  fork (put num 4)
  get num
```

`num`



Data structure author's obligation:

threshold set elements must be **pairwise incompatible**

Raises an error, since $3 \sqcup 4 = \top$

```
do
  fork (put num 3)
  fork (put num 4)
```

Works fine, since $4 \sqcup 4 = 4$

```
do
  fork (put num 4)
  fork (put num 4)
```

`get` blocks until threshold is reached

```
do
  fork (put num 4)
  get num
```

`counter`

$\top$

$\vdots$

3

2

1

$\bot$

Works fine, since `incr`s commute

```
do
  fork (incr1 counter)
  fork (incr42 counter)
```

counter

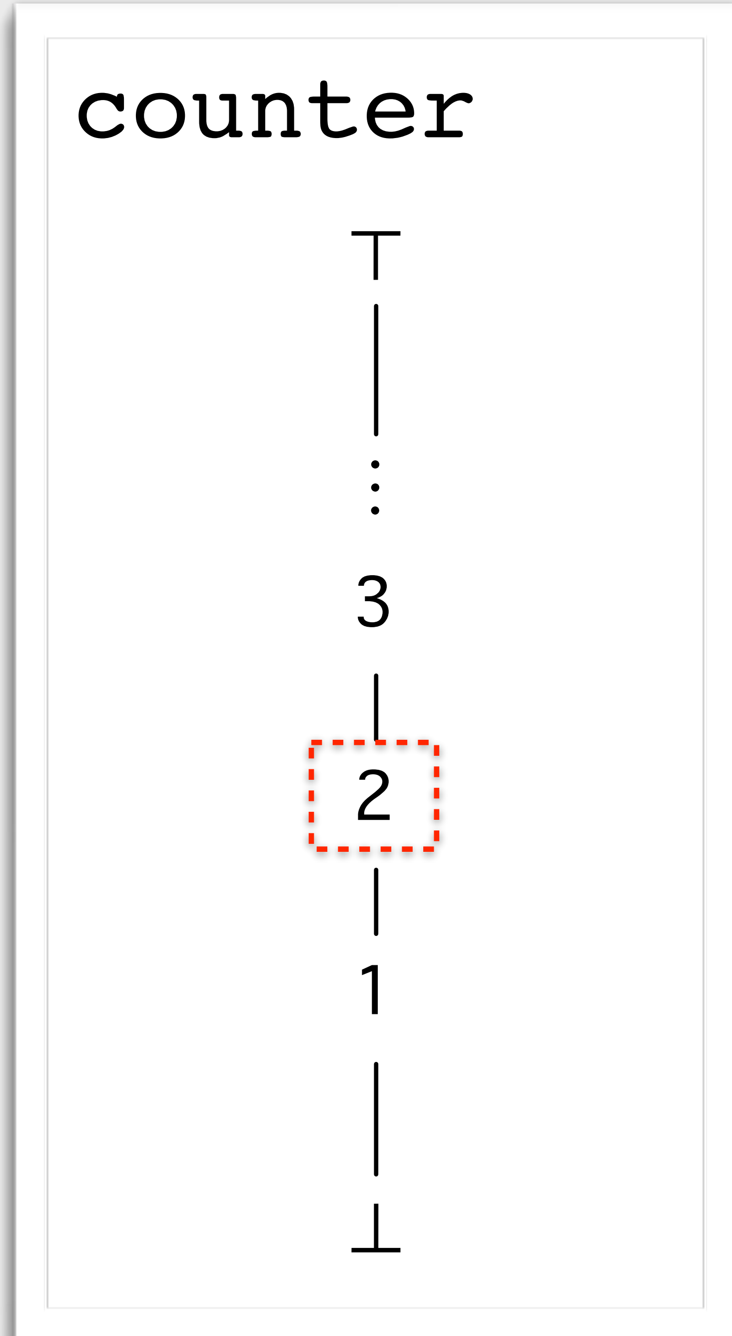
$\top$
|
⋮
3
|
2
|
1
|
$\bot$

Works fine, since `incr`s commute

```
do
  fork (incr1 counter)
  fork (incr42 counter)
```

`get` blocks until threshold is reached

```
do
  fork (incr1 counter)
  fork (incr42 counter)
  get counter 2
```



Works fine, since `incrs` commute

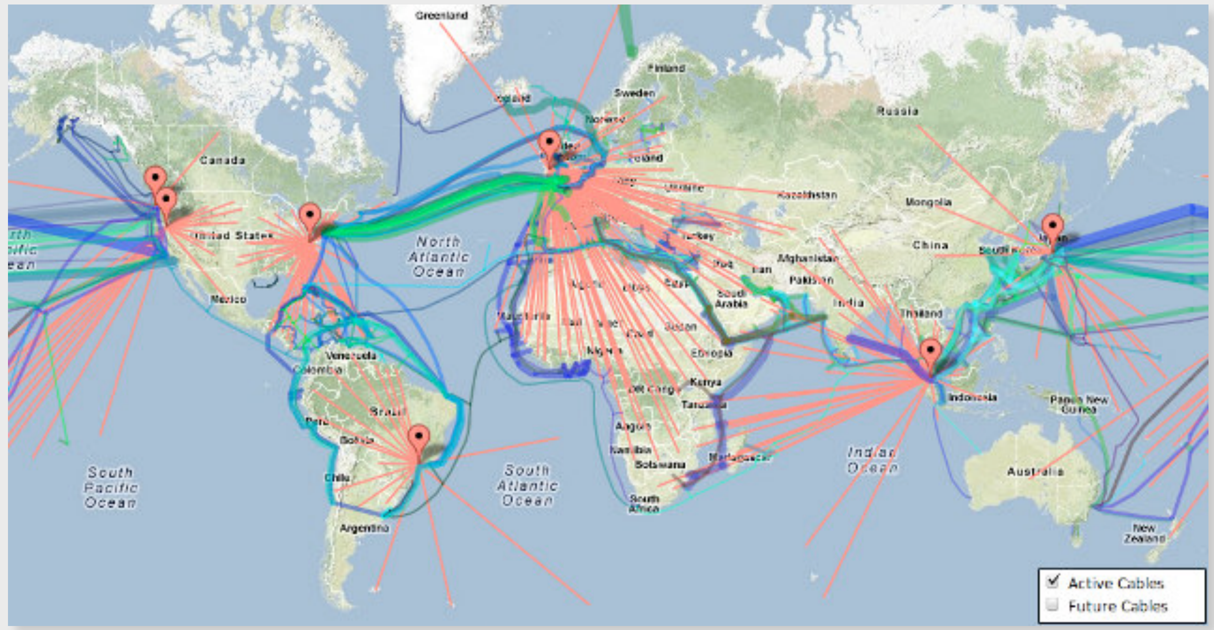
```
do
  fork (incr1 counter)
  fork (incr42 counter)
```

`get` blocks until threshold is reached

```
do
  fork (incr1 counter)
  fork (incr42 counter)
  get counter 2
```

unblocks when `counter` is at least 2

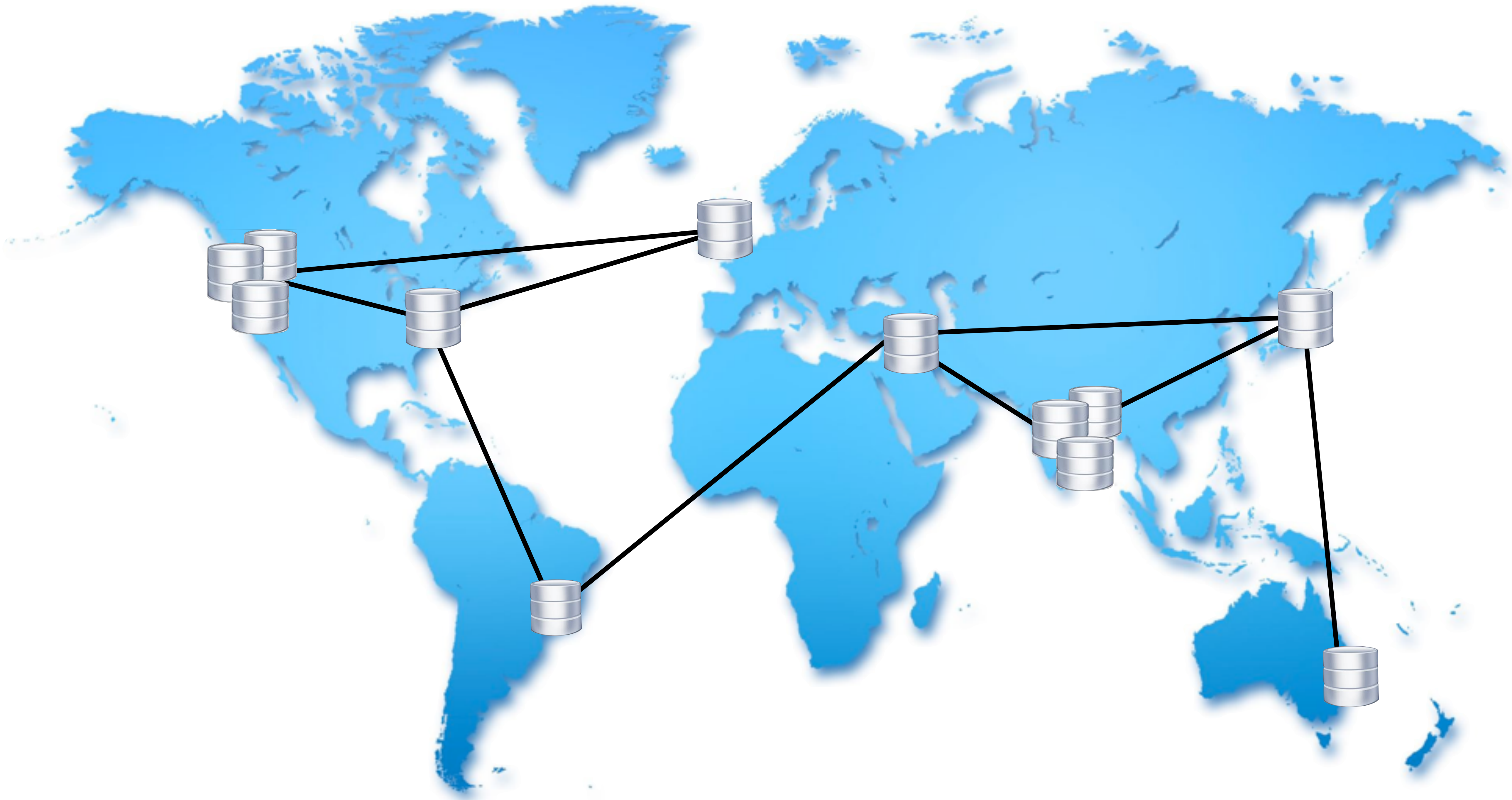
exact contents of `counter` **not observable**

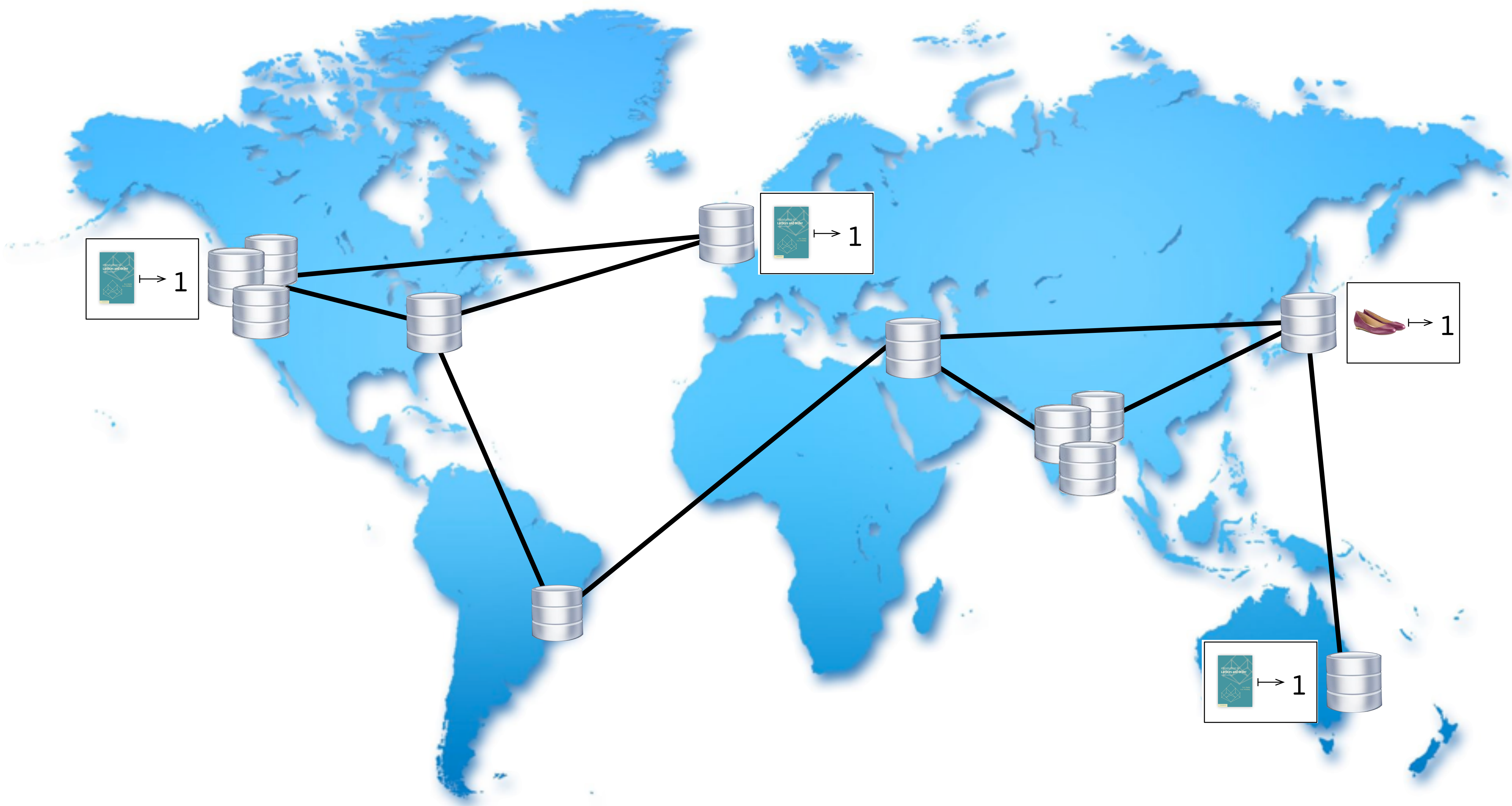


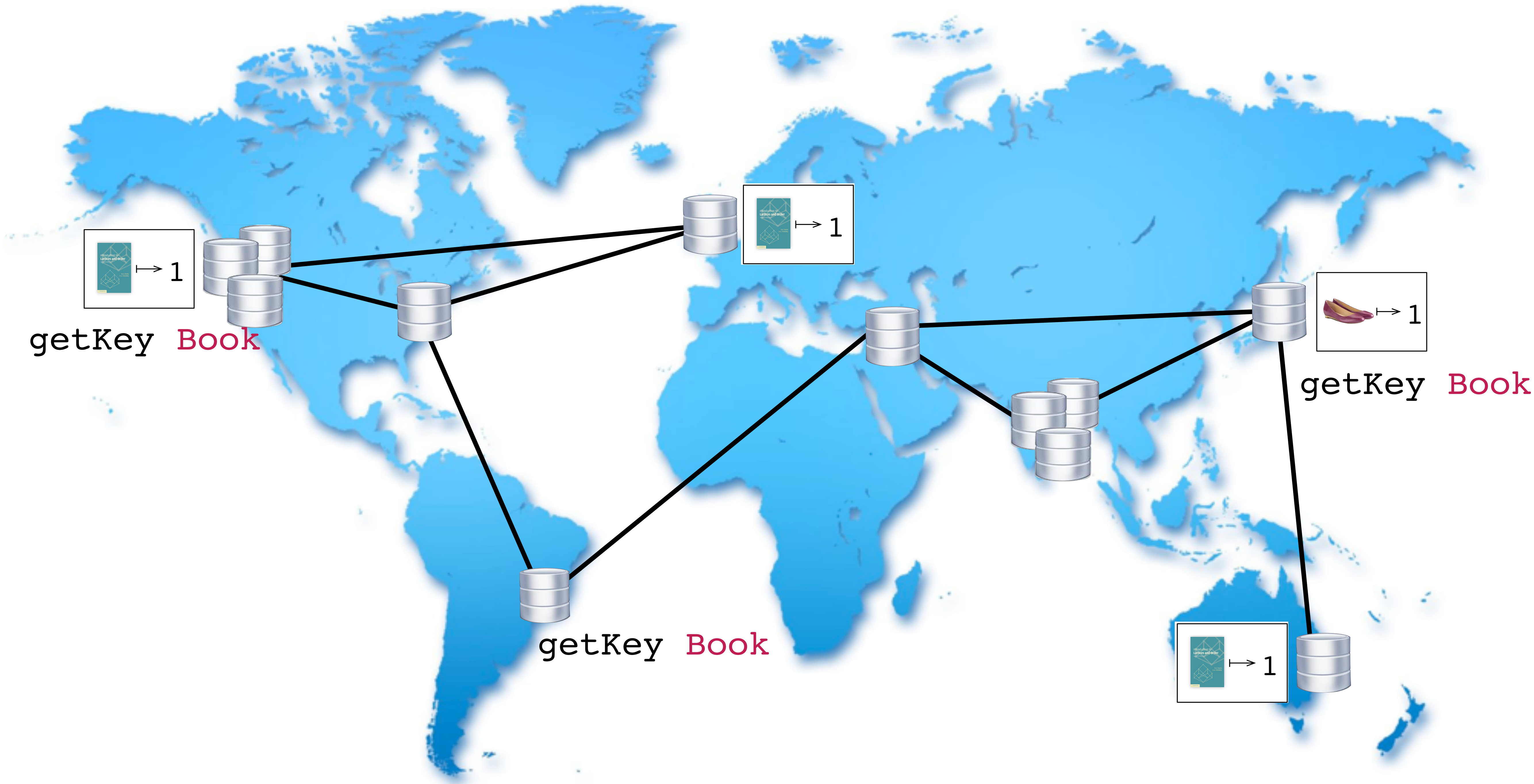
Distributed programming

CRDTs

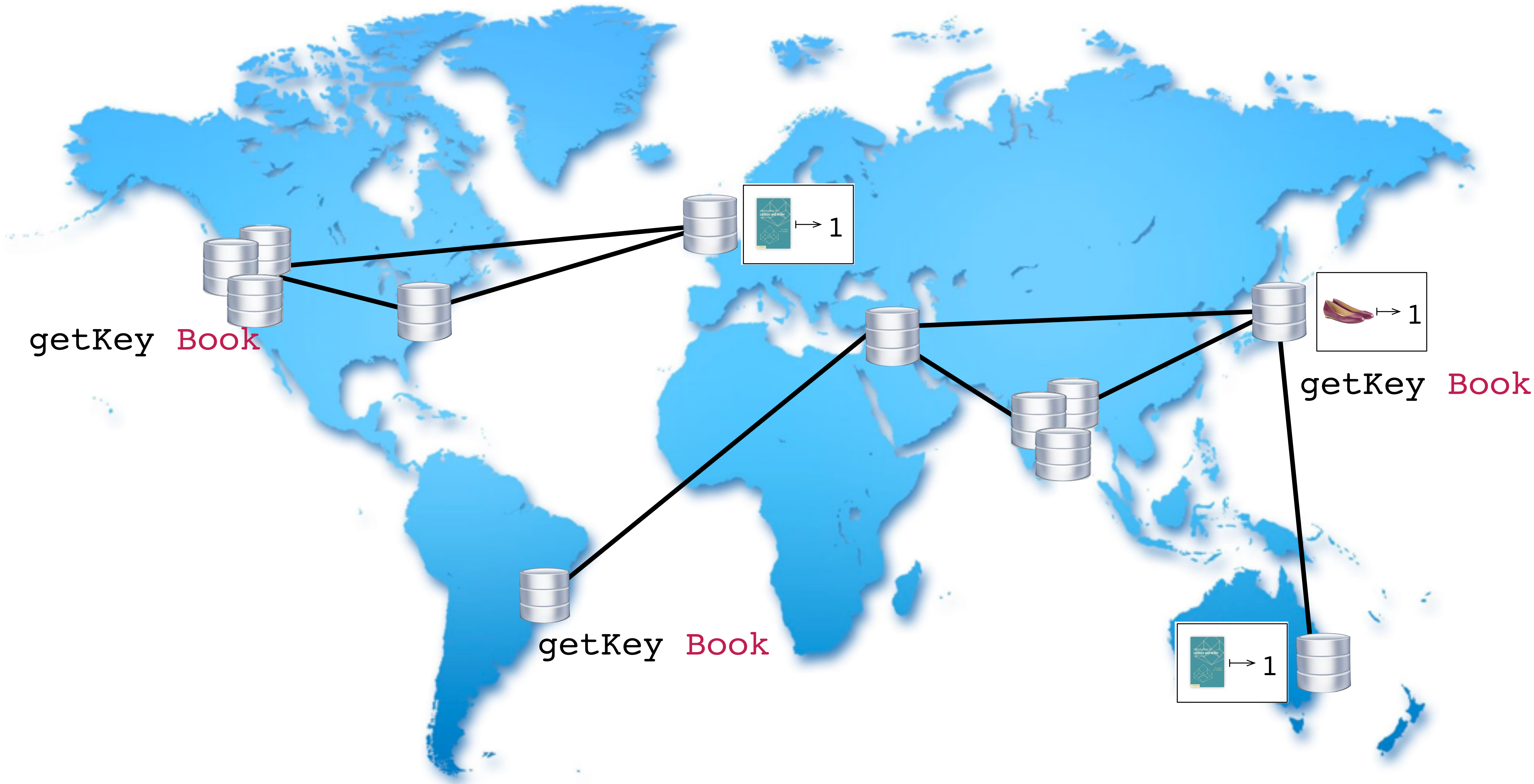
(Eventual) consistency




↦ 1
↦ 1
↦ 1
↦ 1



↦ 1
getKey Book
↦ 1
↦ 1
getKey Book
getKey Book
↦ 1



getKey Book
↦ 1
↦ 1
getKey Book
getKey Book
↦ 1

Replication requires us to trade off between:

Consistency (all replicas agree on the data)

Availability (all replicas can read or write at all times)

Partition tolerance (the system is robust to communication failure between replicas)

Replication requires us to trade off between:

Consistency (all replicas agree on the data)

Availability (all replicas can read or write at all times)

Partition tolerance (the system is robust to communication failure between replicas)

At most two of these properties hold of a given system [Brewer, 2000; Gilbert and Lynch, 2002]

Replication requires us to trade off between:

Consistency (all replicas agree on the data)

Availability (all replicas can read or write at all times)

Partition tolerance (the system is robust to communication failure between replicas)

At most two of these properties hold of a given system [Brewer, 2000; Gilbert and Lynch, 2002]

In large distributed systems, **network partitions are a given**, so we have to give up one of C or A

Replication requires us to trade off between:

Consistency (all replicas agree on the data)

Availability (all replicas can read or write at all times)

Partition tolerance (the system is robust to communication failure between replicas)

At most two of these properties hold of a given system [Brewer, 2000; Gilbert and Lynch, 2002]

In large distributed systems, **network partitions are a given**, so we have to give up one of C or A

But: **we should think of C, A, and P as more continuous than binary** [Brewer, 2012]

Replication requires us to trade off between:

Consistency (all replicas agree on the data)

Availability (all replicas can read or write at all times)

Partition tolerance (the system is robust to communication failure between replicas)

At most two of these properties hold of a given system [Brewer, 2000; Gilbert and Lynch, 2002]

In large distributed systems, **network partitions are a given**, so we have to give up one of C or A

But: **we should think of C, A, and P** as **more continuous than binary** [Brewer, 2012]

We can opt for **eventual consistency** [Vogels, 2009] since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart.

[DeCandia *et al*., 2007]

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]:

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as
S3), is probably the best known. This paper presents the
lementation of Dynamo, another highly available
ributed data store built for Amazon's platform.
to manage the state of services that have very
requirements and need tight control over the
een availability, consistency, cost-effectiveness and
Amazon's platform has a very diverse set of
different storage requirements. A select set of
ires a storage technology that is flexible enough
designers configure their data store appropriately
tradeoffs to achieve high availability and
rmance in the most cost effective manner.

services on Amazon's platform that only need
ess to a data store. For many services, such as
de best seller lists, shopping carts, customer
on management, sales rank, and product catalog,
ern of using a relational database would lead to
limit scale and availability. Dynamo provides a
key only interface to meet the requirements of

synthesis of well known techniques to achieve
vailability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

Conflict-free replicated data types

[Shapiro *et al.*, 2011]

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

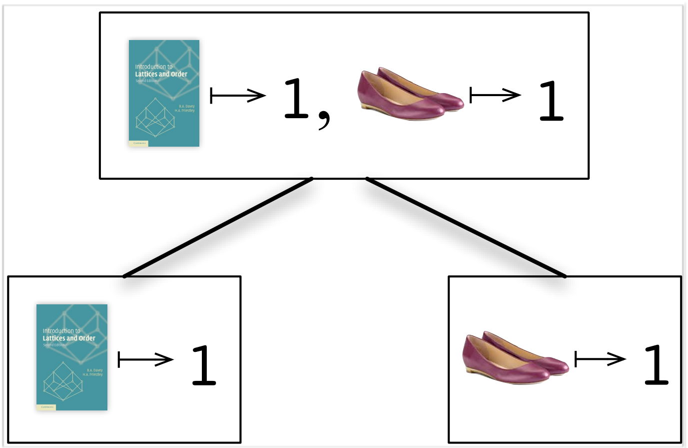
To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as ... S3) is probably the best known. This paper presents the ... ementation of Dynamo, another highly available ... ributed data store built for Amazon's platform. ... to manage the state of services that have very ... requirements and need tight control over the ... een availability, consistency, cost-effectiveness and ... Amazon's platform has a very diverse set of ... different storage requirements. A select set of ... ires a storage technology that is flexible enough ... designers configure their data store appropriately ... tradeoffs to achieve high availability and ... rmance in the most cost effective manner.

... services on Amazon's platform that only need ... ess to a data store. For many services, such as ... de best seller lists, shopping carts, customer ... on management, sales rank, and product catalog, ... ern of using a relational database would lead to ... limit scale and availability. Dynamo provides a ... key only interface to meet the requirements of ...

... synthesis of well known techniques to achieve ... vailability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

> since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart.

[DeCandia *et al.*, 2007]



Conflict-free replicated data types

[Shapiro *et al.*, 2011]

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]:

> since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as S3) is probably the best known. This paper presents the ...ementation of Dynamo, another highly available ...ributed data store built for Amazon's platform. ...to manage the state of services that have very ...requirements and need tight control over the ...een availability, consistency, cost-effectiveness and ...Amazon's platform has a very diverse set of ...different storage requirements. A select set of ...ires a storage technology that is flexible enough ...designers configure their data store appropriately ...tradeoffs to achieve high availability and ...rmance in the most cost effective manner.

...services on Amazon's platform that only need ...ess to a data store. For many services, such as ...de best seller lists, shopping carts, customer ...on management, sales rank, and product catalog, ...rn of using a relational database would lead to ...limit scale and availability. Dynamo provides a ...key only interface to meet the requirements of ...

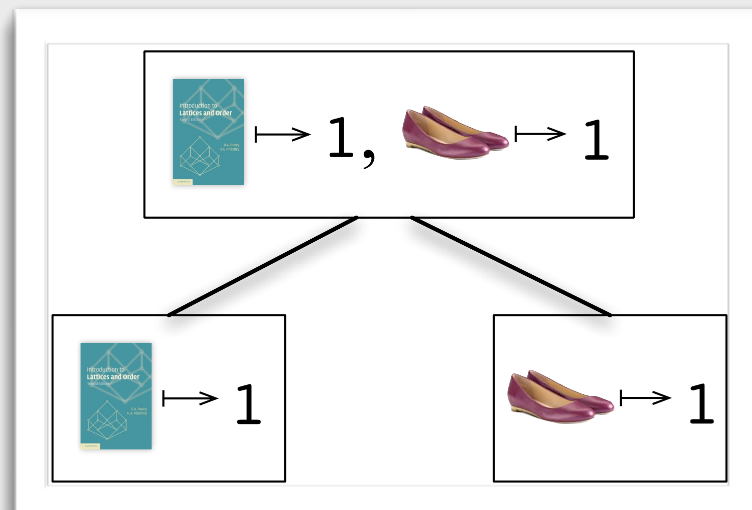
...synthesis of well known techniques to achieve ...vailability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

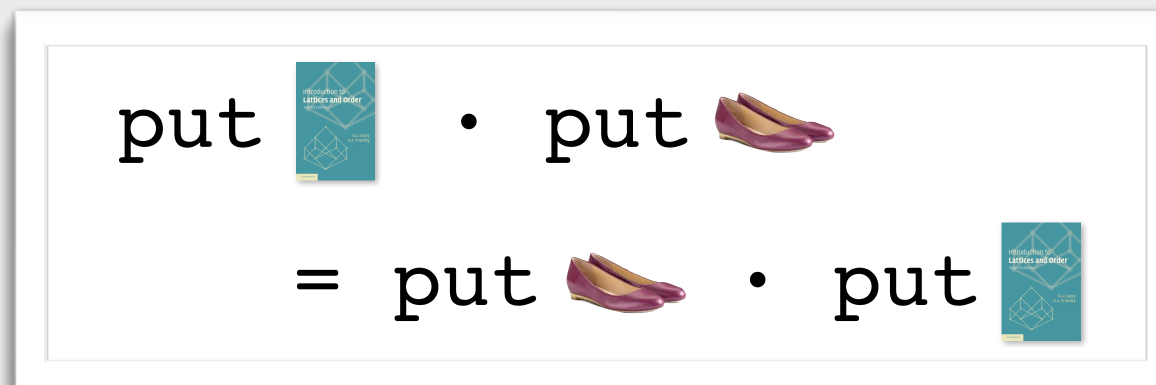
[DeCandia *et al.*, 2007]

Two flavors of CRDTs

"Convergent"
"state-based"
CvRDTs

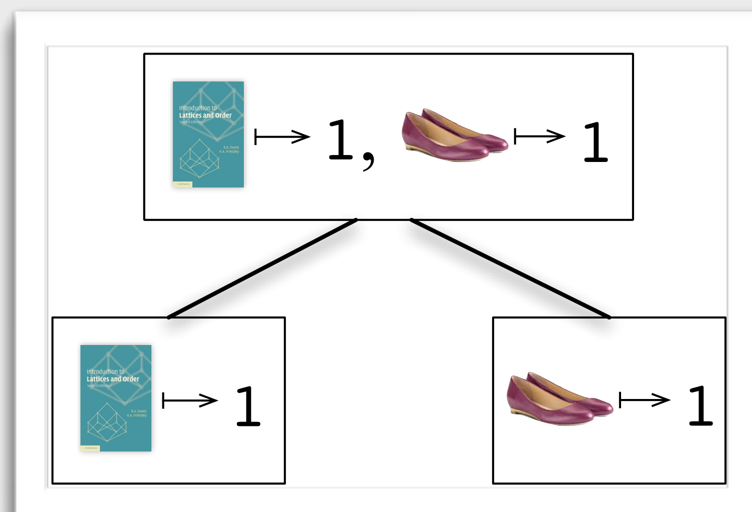
"Commutative"
"operation-based"
CmRDTs

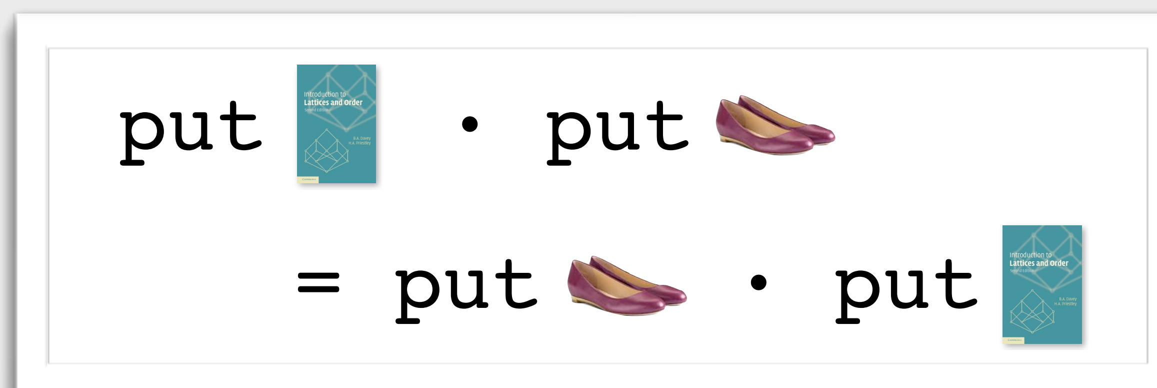




Two flavors of CRDTs

“Convergent”		“Commutative”
“state-based”	$\Leftrightarrow$	“operation-based”
CvRDTs	[Shapiro *et al.*, 2011]	CmRDTs





LVars *vs.* CvRDTs

LVars	CvRDTs
Threshold reads (deterministic)	Ordinary reads (non-deterministic)
Least-upper-bound writes (every write computes a lub)	Inflationary, commutative writes (only **replica merges** must be lubs)
Shared	Replicated!

LVars *vs.* CvRDTs

Threshold reads (deterministic)	Ordinary reads (non-deterministic)
Least-upper-bound writes (every write computes a lub)	Inflationary, commutative writes (only **replica merges** must be lubs)
Shared	Replicated!

So, to join forces:

- Generalize LVars to inflationary, commutative writes
  This gives us non-idempotent, **incrementable** counters
  (we were using them anyway…)
- Extend CvRDTs with threshold queries
  Systems in the wild (e.g., Amazon SimpleDB) already allow consistency choices at per-read granularity

LVars *vs.* CvRDTs

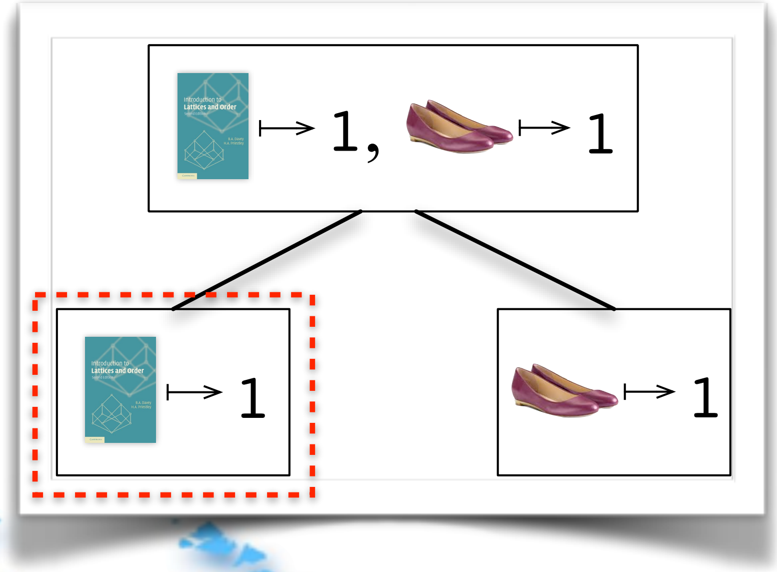
LVars	CvRDTs
Threshold reads (deterministic)	Ordinary reads (non-deterministic)
~~Least upper bound writes (every write computes a lub)~~	Inflationary, commutative writes (only **replica merges** must be lubs)
Shared	Replicated!

So, to join forces:

- Generalize LVars to inflationary, commutative writes
  This gives us non-idempotent, **incrementable** counters
  (we were using them anyway…)
- Extend CvRDTs with threshold queries
  Systems in the wild (e.g., Amazon SimpleDB) already allow consistency choices at per-read granularity

Deterministic **threshold queries** of CvRDTs:

Block only until a threshold element appears at **one** replica (that's all we need!)



Programming Systems Lab

(We're hiring! Email me!)