Causality for the Masses:

Offering Fresh Data, Low Latency, and High Throughput

Luís Rodrigues

Causality for the Masses:

Offering Fresh Data, Low Latency, and High Throughput

Manuel Bravo,
Luís Rodrigues, Chathuri Gunawardhana, Peter van Roy

Causal consistency

Limitations of Highly-Available Eventually-Consistent Data Stores

Hagit Attiya
Computer Science Department
Technion

Faith Ellen
Department of Computer Science
University of Toronto

Adam Morrison
Computer Science Department
Technion

ABSTRACT

Modern *replicated* data stores aim to provide high availability, by immediately responding to client requests, often by implementing objects that expose concurrency. Such objects, for example, multi-valued registers (MVRs), do not have sequential specifications. This paper explores a recent model for replicated data stores that can be used to precisely specify *causal consistency* for such objects, and liveness properties like *eventual consistency*, without revealing details of the underlying implementation. The model is used to prove the following results:

- An eventually consistent data store implementing MVRs cannot satisfy a consistency model strictly stronger than *observable causal consistency* (OCC). OCC is a model somewhat stronger than causal consistency, which captures executions in which client observations can use causality to infer concurrency of operations. This result holds under certain assumptions about the data store.
- Under the same assumptions, an eventually consistent and causally consistent replicated data store must send messages of unbounded size: If s objects are supported by n replicas, then, for every $k > 1$, there is an execution in which an $\Omega(\min\{n, s\}k)$-bit message is sent.

of data (i.e. accesses to data return without delay), and its *consistency*, while tolerating message delays. The *CAP theorem* [8, 18] demonstrates the difficulty of achieving this balance, showing that strong *Consistency* (i.e. *atomicity*) cannot be satisfied together with high *Availability* and *Partition tolerance*.

One aspect of data *consistency* is a safety property restricting the possible values observed by clients accessing different replicas. The set of possible executions is called a *consistency model*. For example, *causal consistency* [2] ensures that the causes of an operation are visible at a replica no later than the operation itself. (A precise definition appears in Section 3.) A smaller set of possible executions means that there is less uncertainty about the data. So, one consistency model is *strictly stronger* than another if its executions are a proper subset of the executions of the other.

Some weak consistency models can be trivially satisfied by never updating the data. Therefore, another aspect of data consistency is a liveness property, ensuring that updates are applied at all replicas. The designers of many systems, e.g., Dynamo [13] and Cassandra [1], opt for a very weak liveness property called, somewhat confusingly, *eventual consistency* [5, 9, 10, 29]. Eventual consistency only ensures that each replica eventually observes all updates to the object, a property also referred to as *update propagation* [11].

Strongest without compromising availability

Causal consistency

Key ingredient of several consistency criteria

Parallel Snapshot Isolation
[SOSP'11]

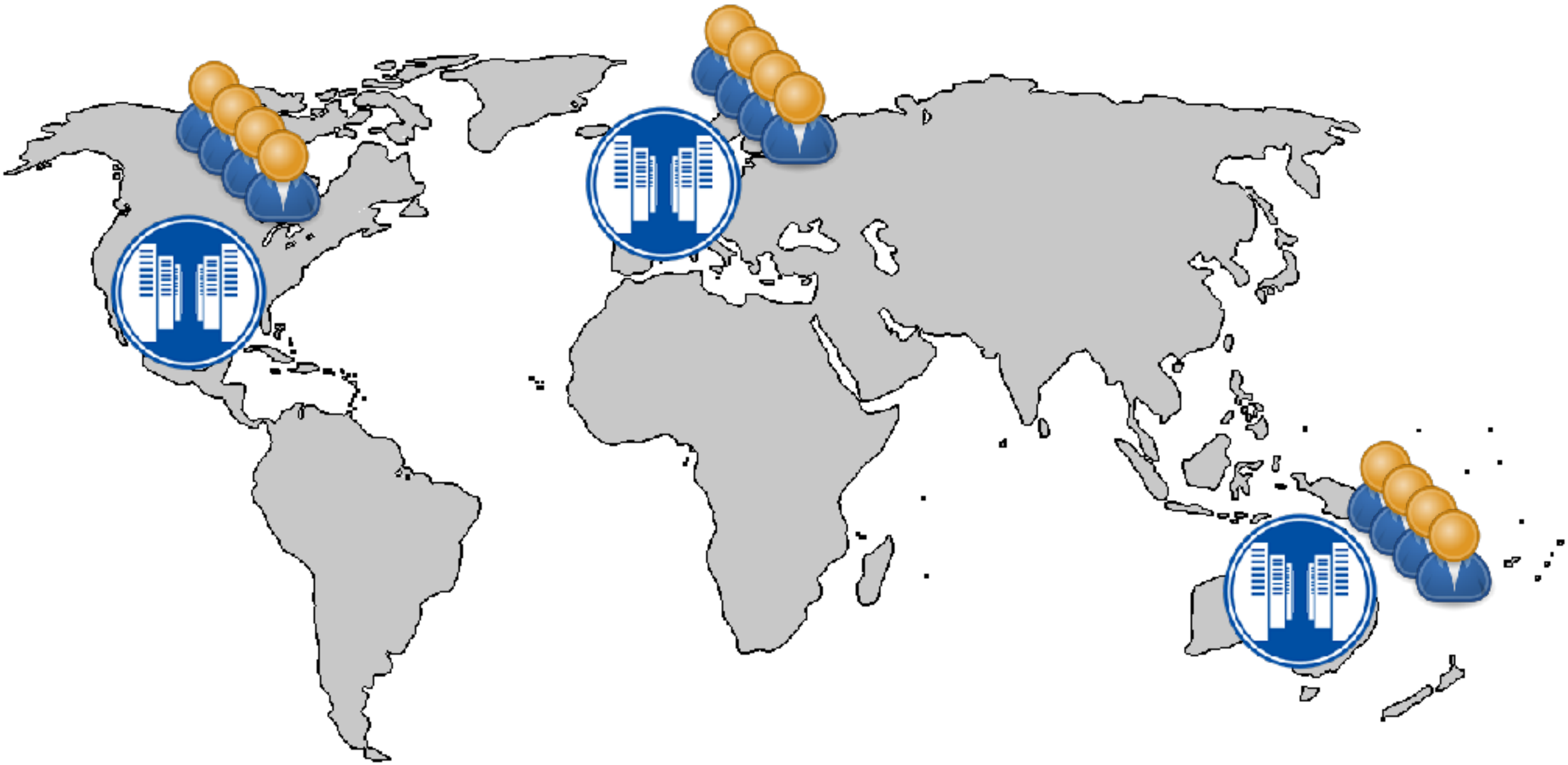
RedBlue Consistency
[OSDI'12]

Explicit Consistency
[EuroSys'15]

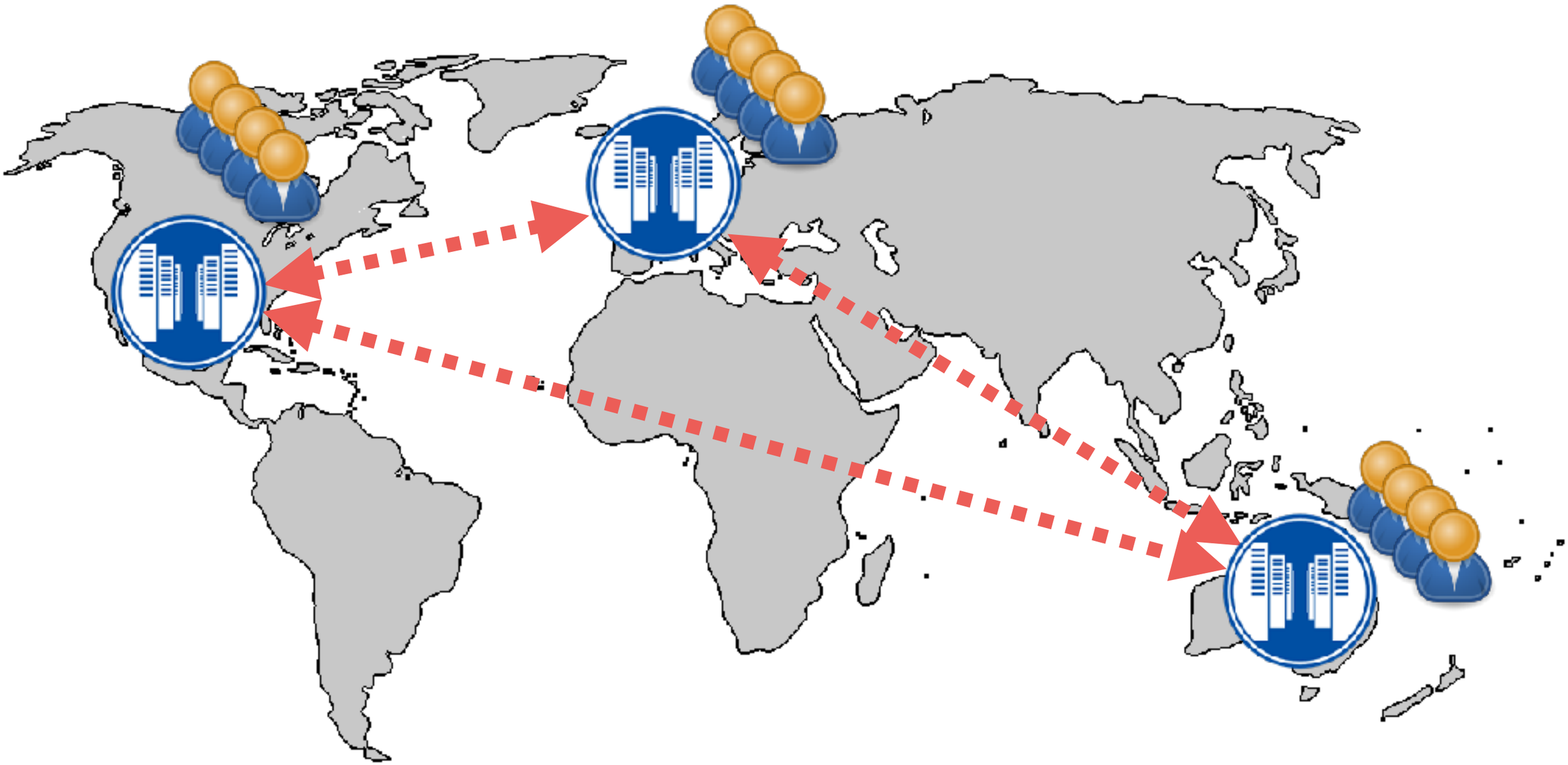
Session guarantees
[SOSP'97]

Causal consistency

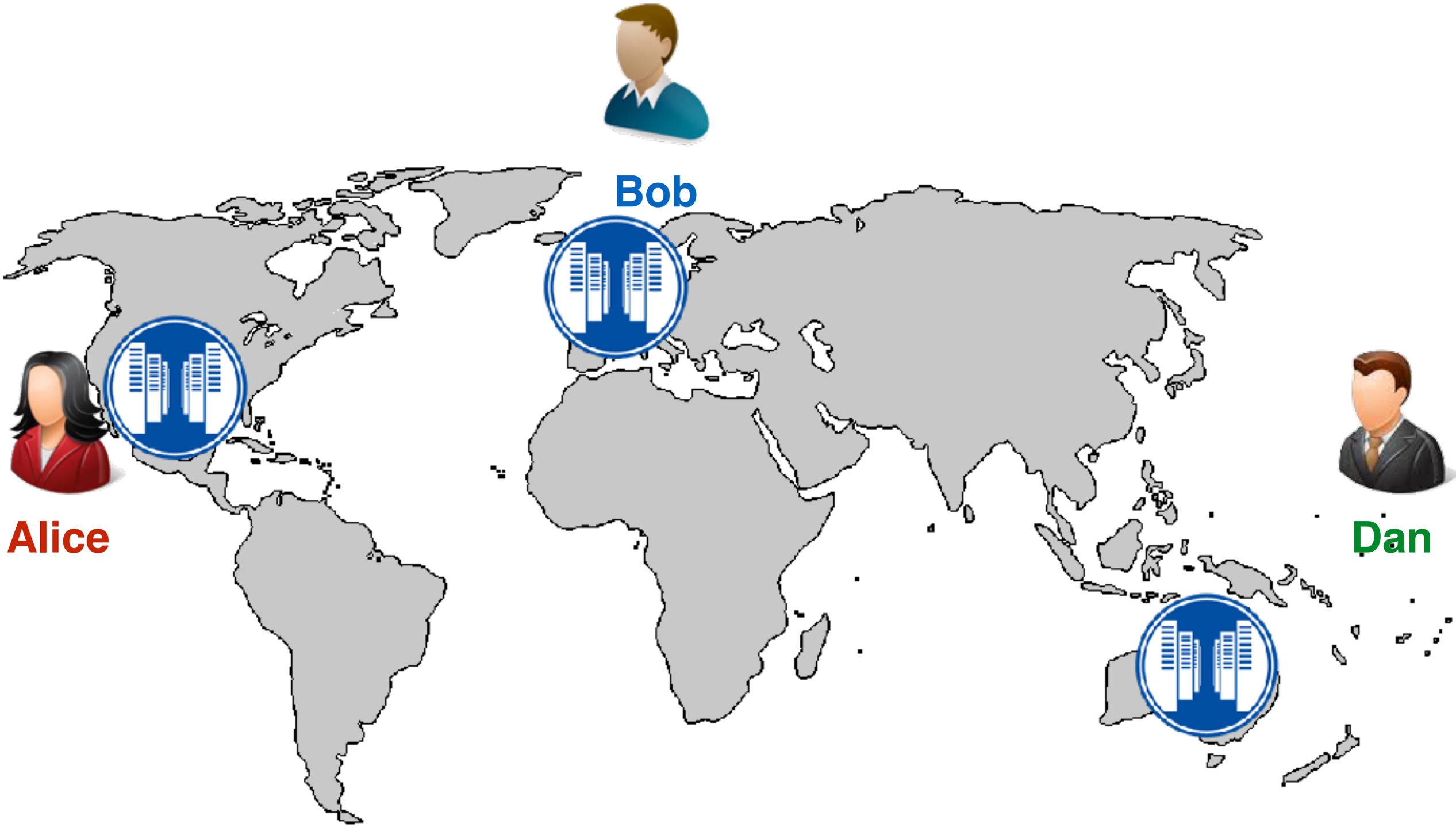
Causal consistency

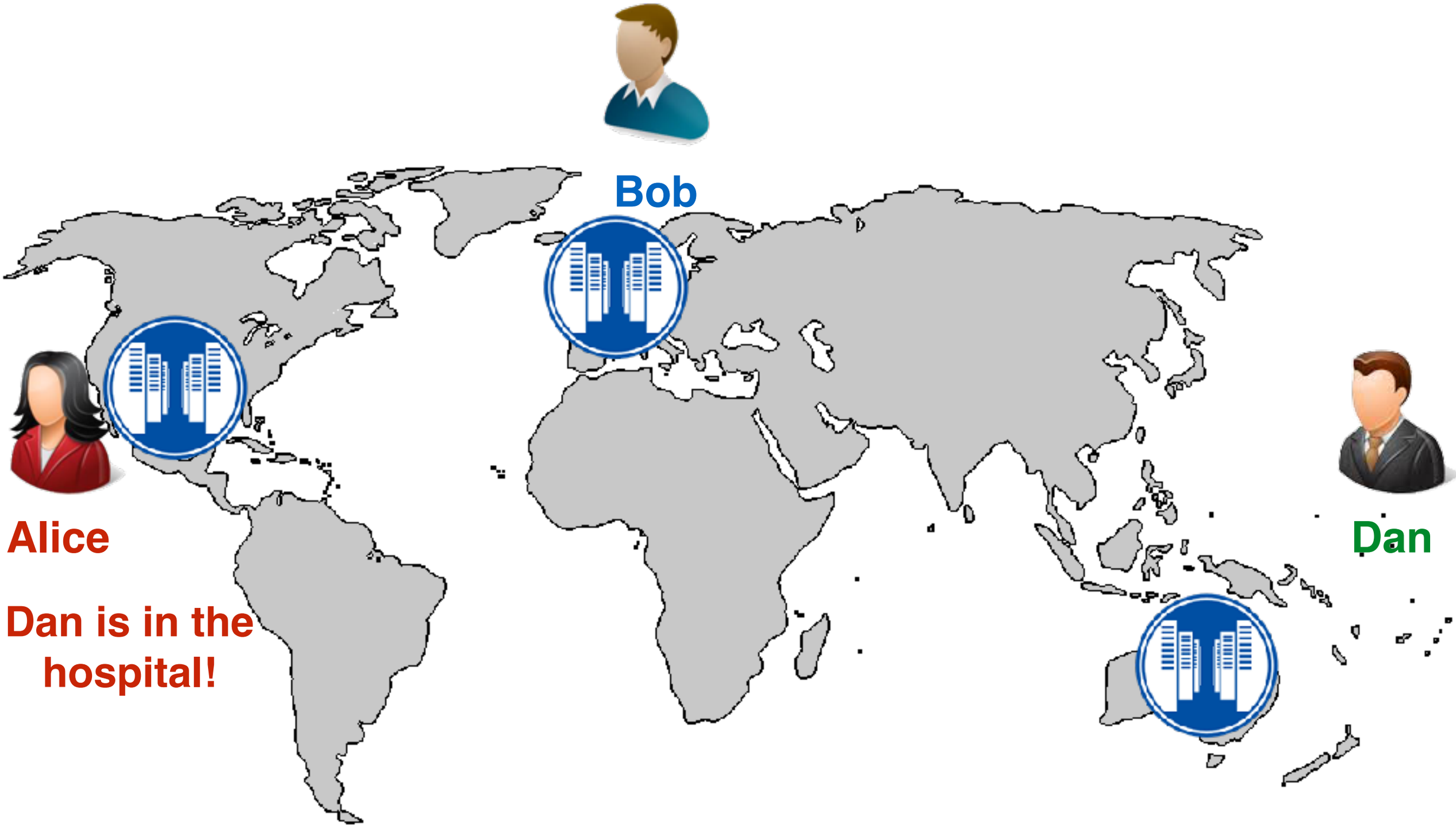


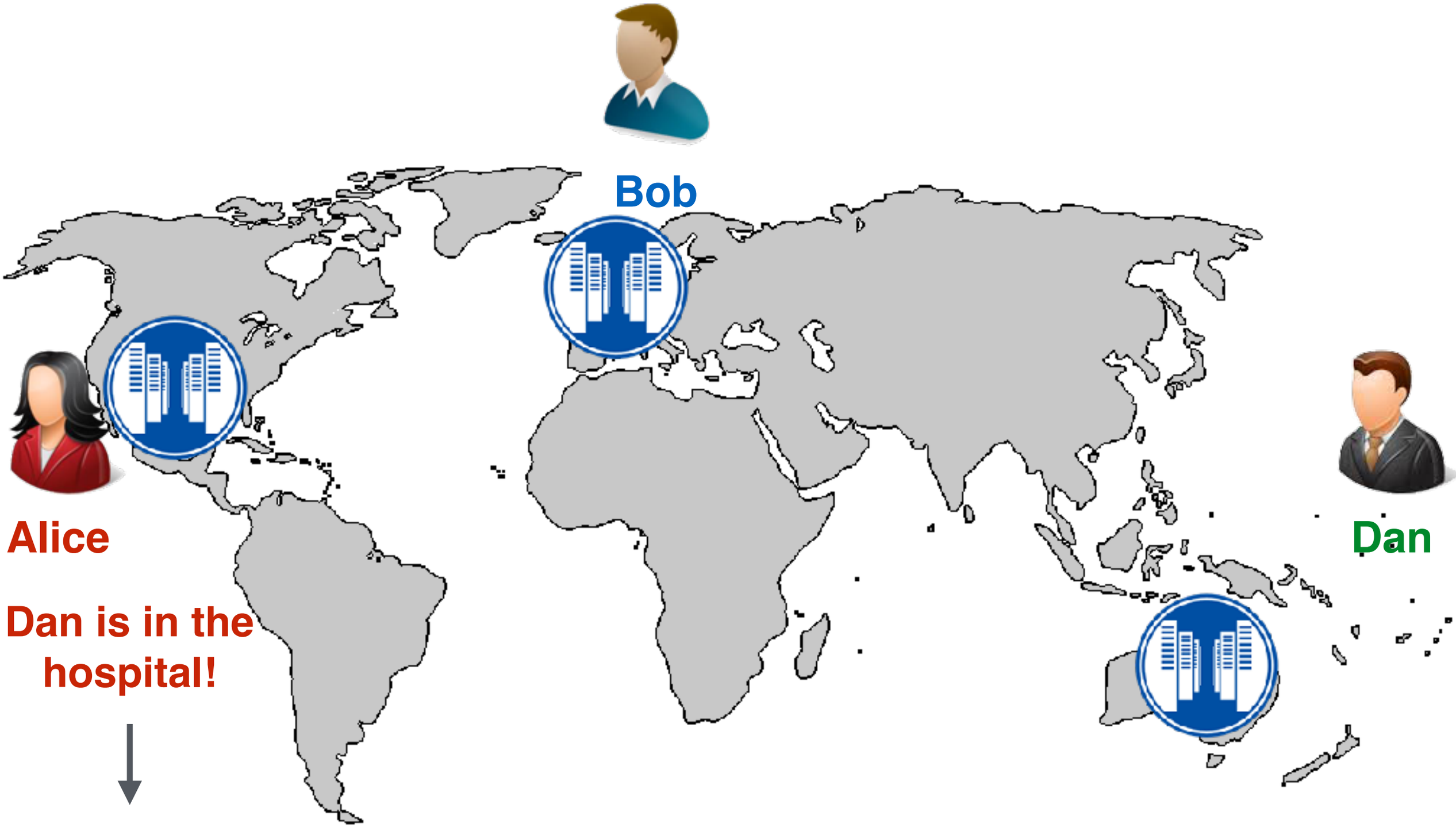
Causal consistency



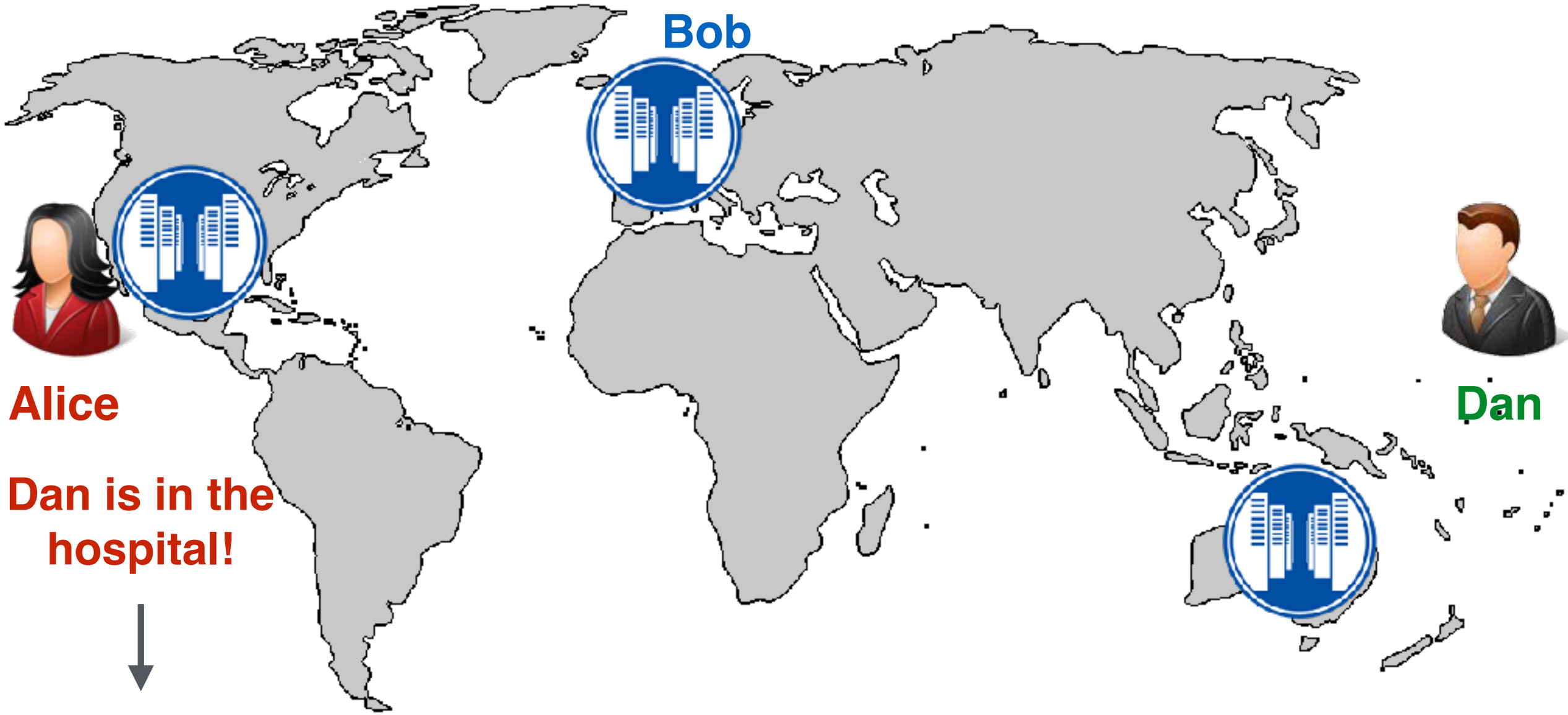
Causal consistency


Bob
Alice
Dan


Bob
Alice
Dan is in the hospital!
Dan

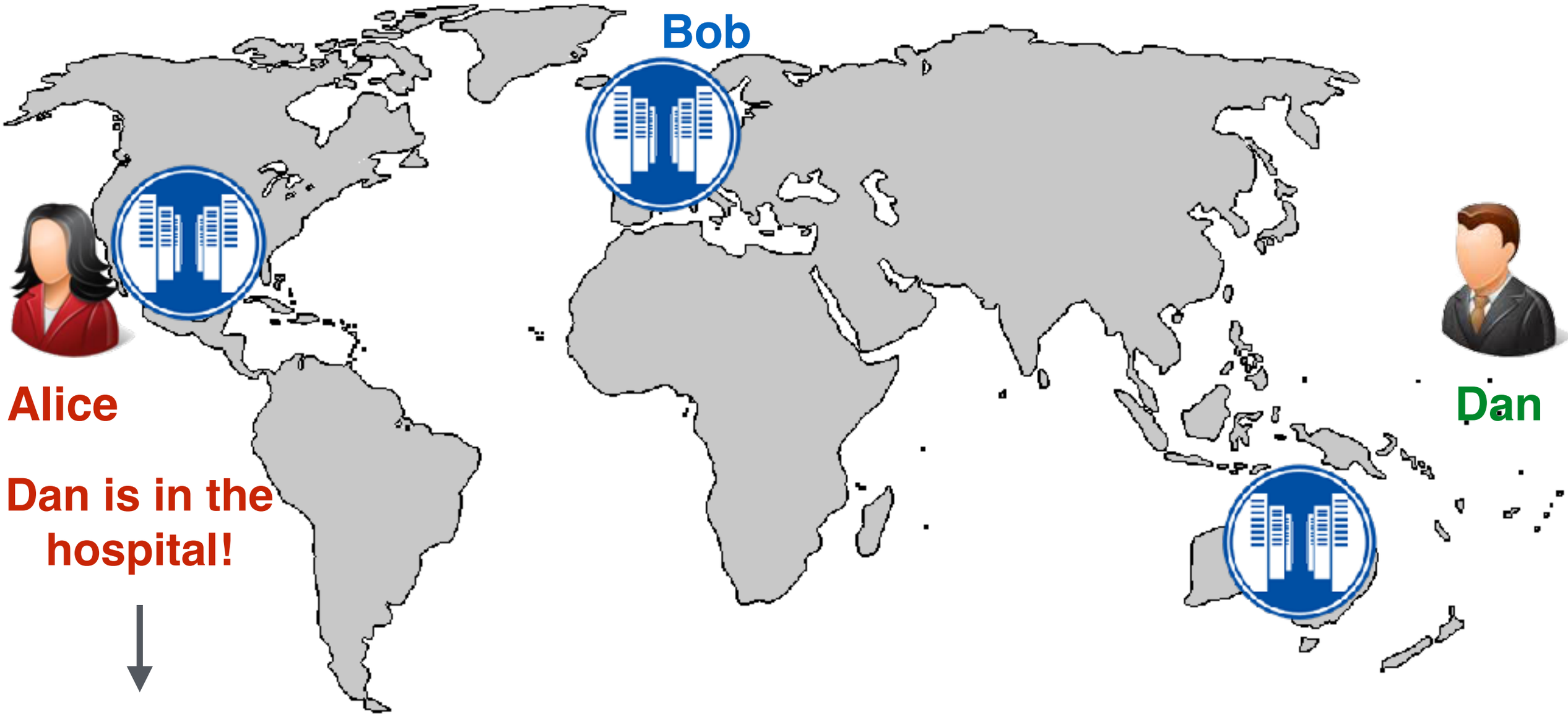

Bob
Alice
Dan is in the hospital!
Dan is ok!
Dan

Dan is in the hospital!

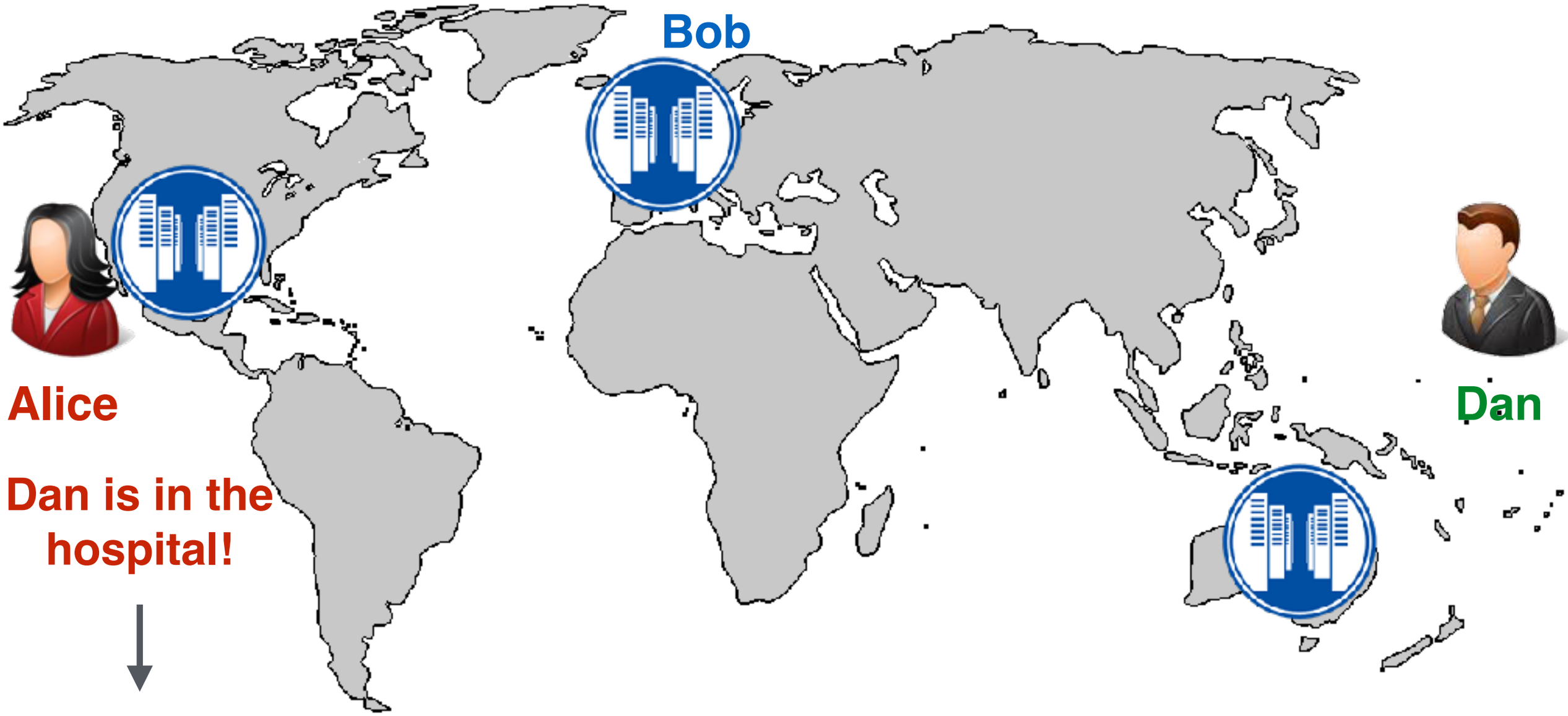


Dan is in the hospital!

Dan is ok!

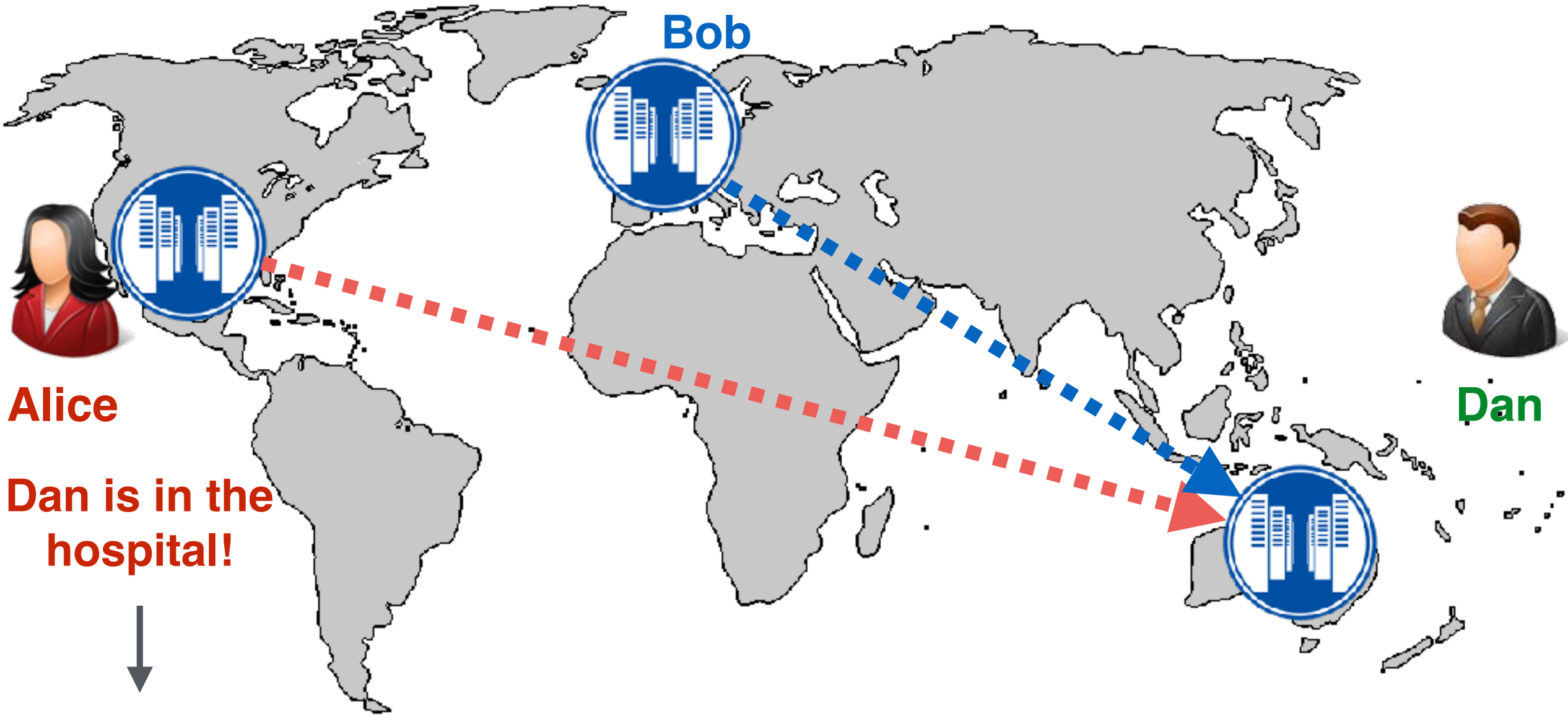


Dan is in the hospital! → Dan is ok! → That's great!



Dan is in the hospital! → Dan is ok! → That's great!

Bob



Alice

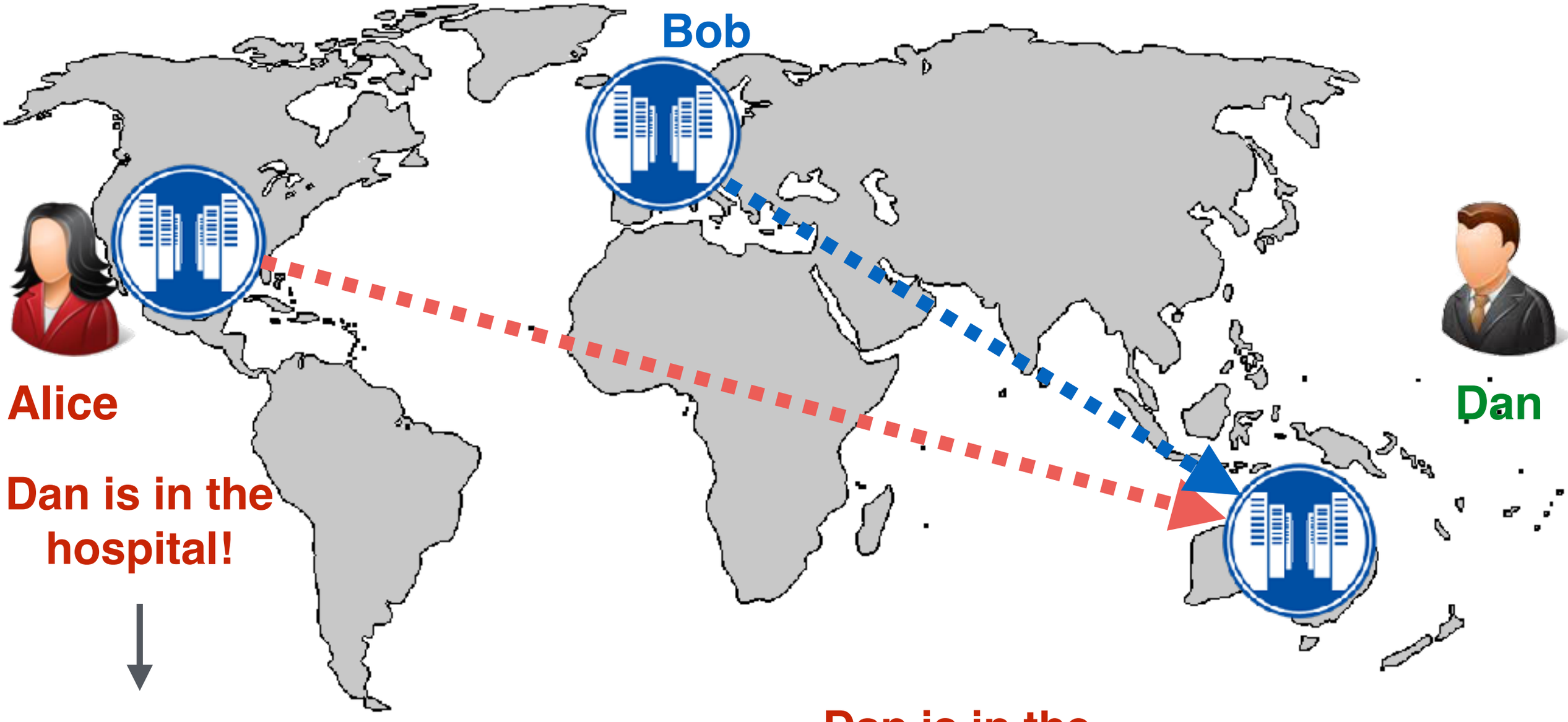
Dan is in the hospital!

↓

Dan is ok!

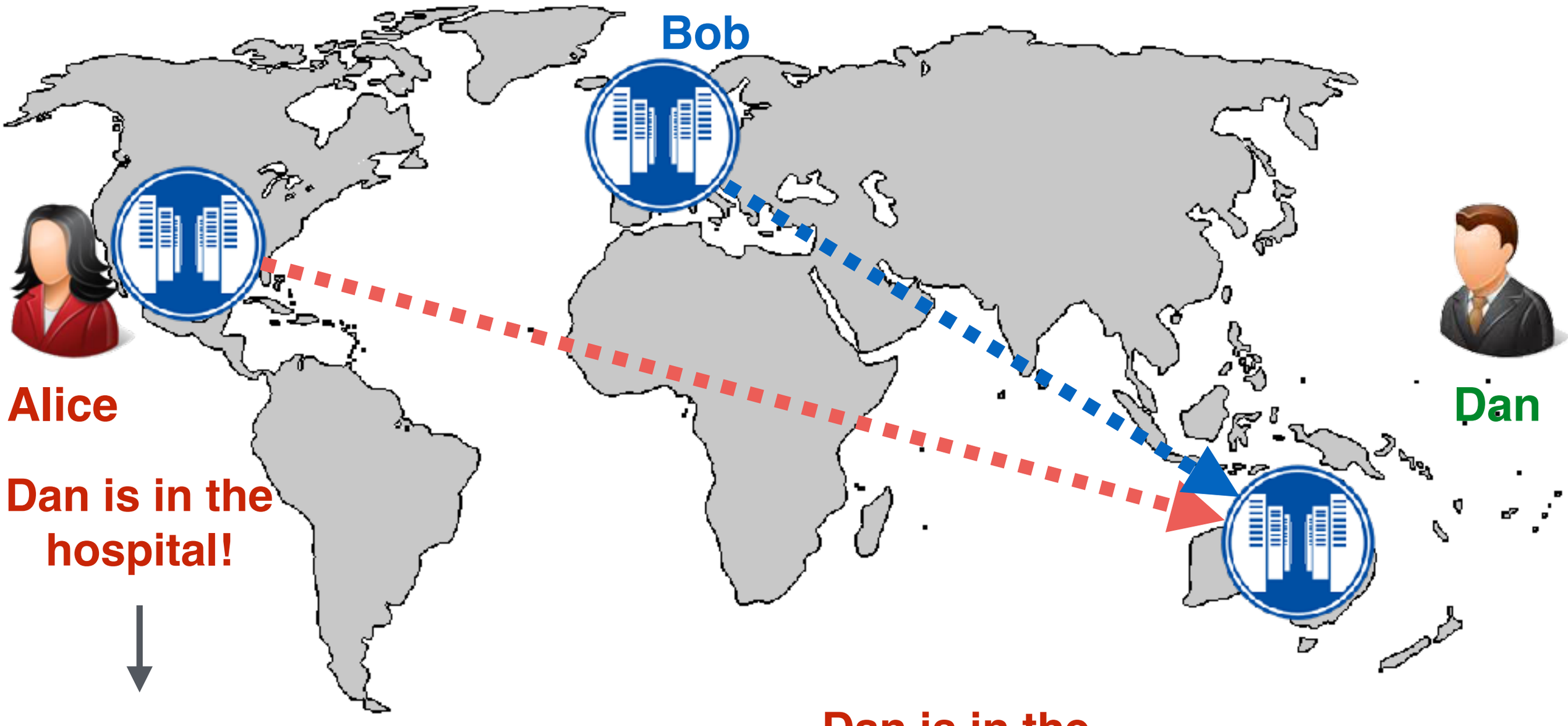
Dan

Dan is in the hospital! → Dan is ok! → That's great!



Dan is in the hospital! → Dan is ok! → That's great!

Bob



Alice

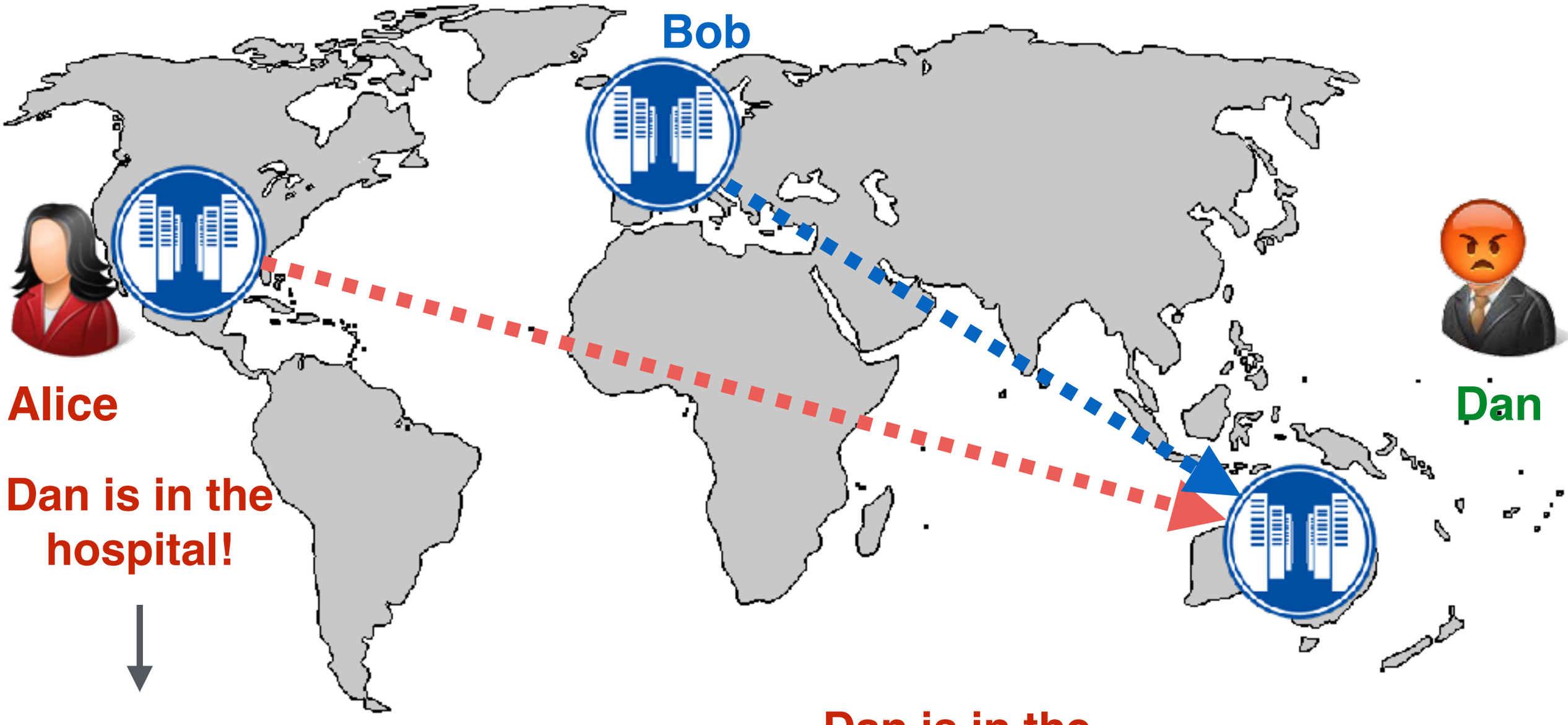
Dan is in the hospital!

Dan is ok!

Dan

Dan is in the hospital! → That's great!

Dan is in the hospital! → Dan is ok! → That's great!

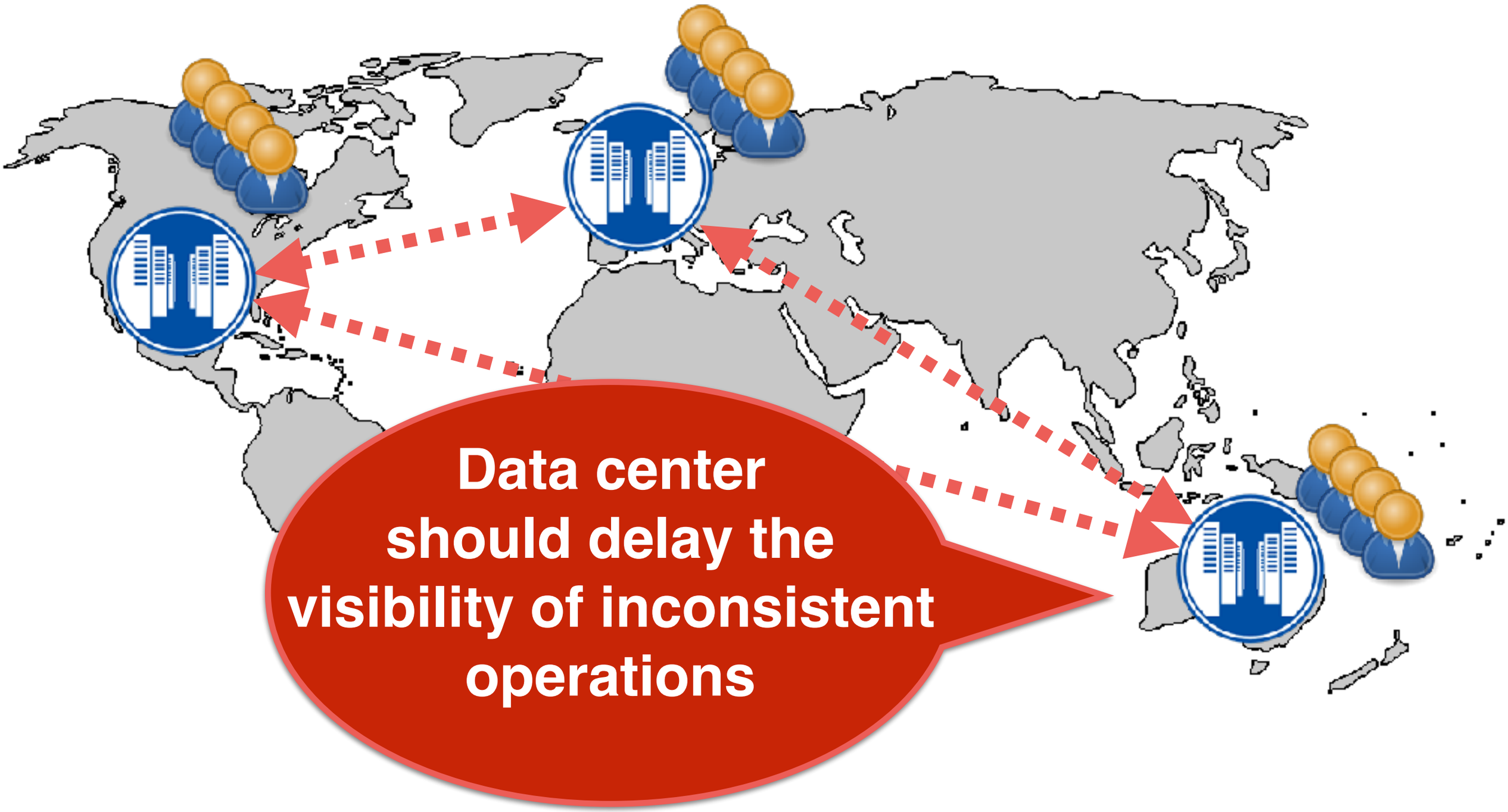


Dan is in the hospital! → Dan is ok!

Dan is in the hospital! → That's great!

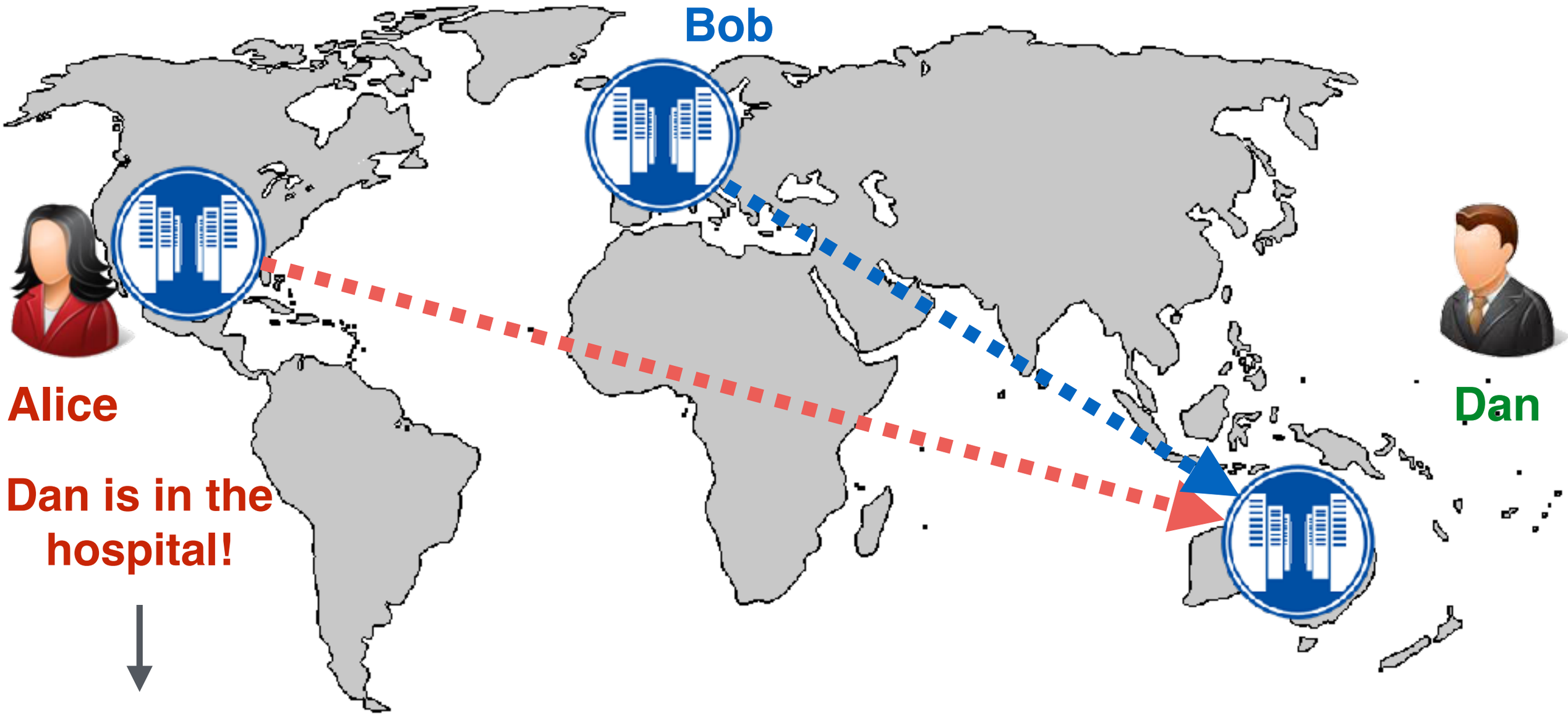
Causal consistency

Causal consistency



Dan is in the hospital! → Dan is ok! → That's great!

Bob



Alice

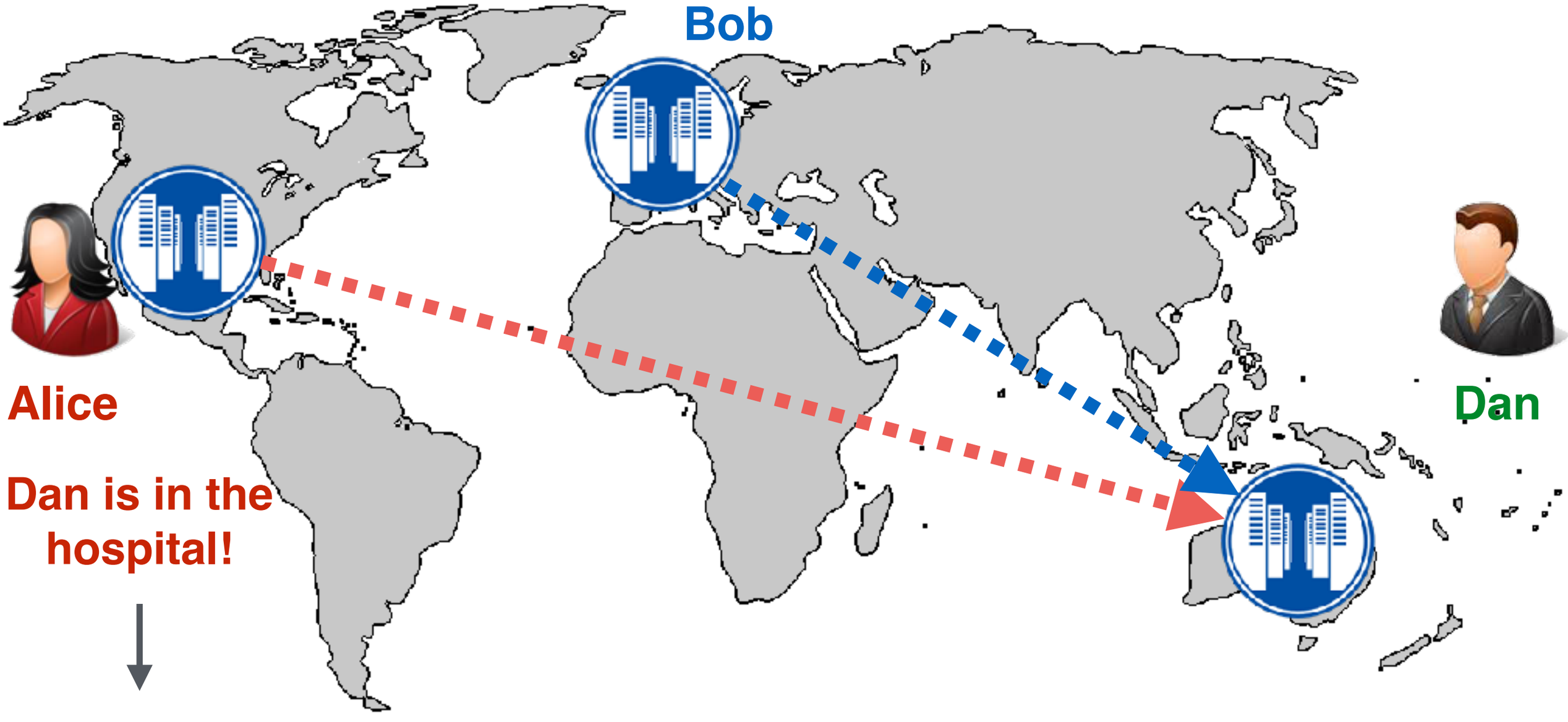
Dan is in the hospital!

Dan is ok!

Dan

Dan is in the hospital! → Dan is ok! → That's great!

Bob



Alice

Dan is in the hospital!

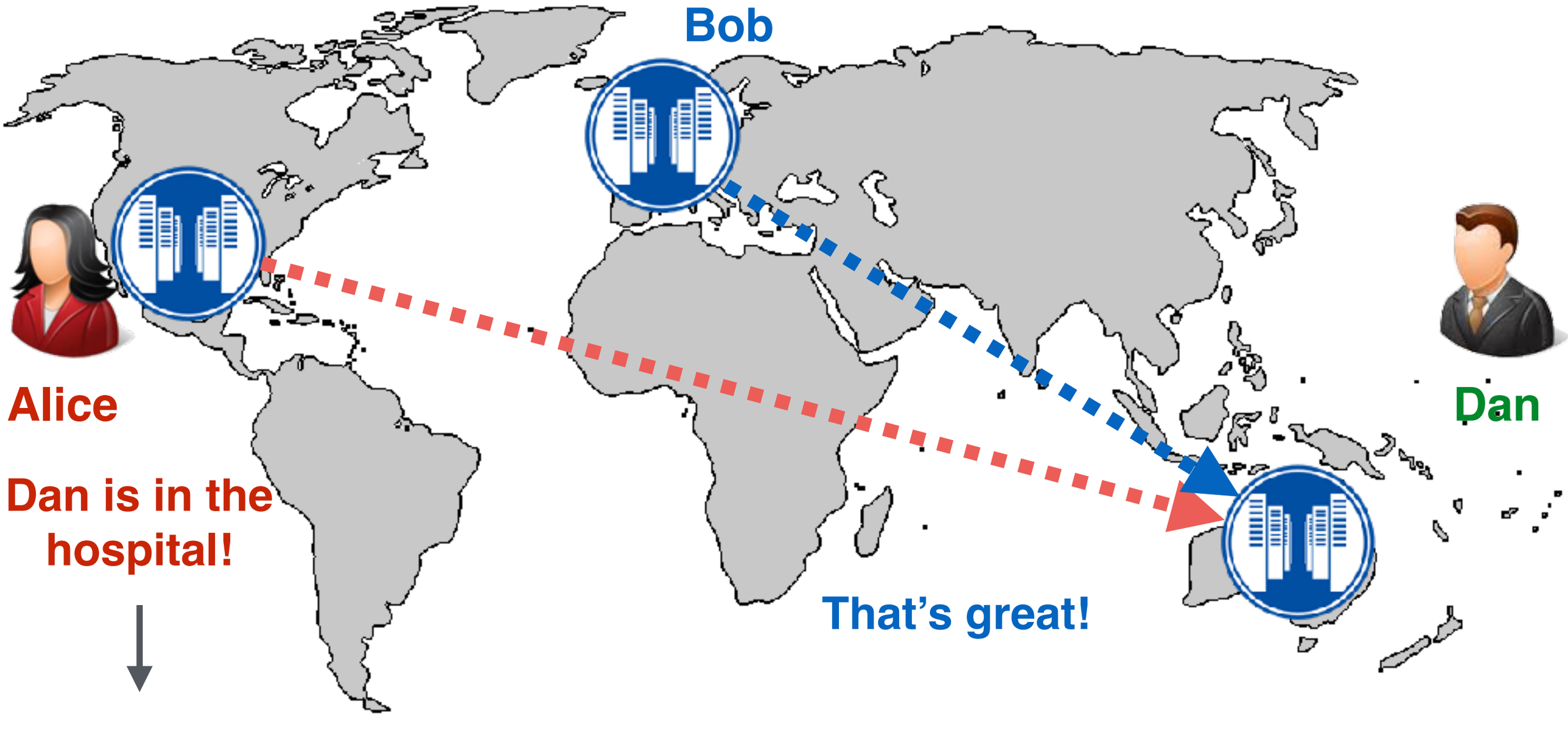
Dan is ok!

Dan

Dan is in the hospital!

Dan is in the hospital! → Dan is ok! → That's great!

Bob



Alice

Dan is in the hospital!

Dan is ok!

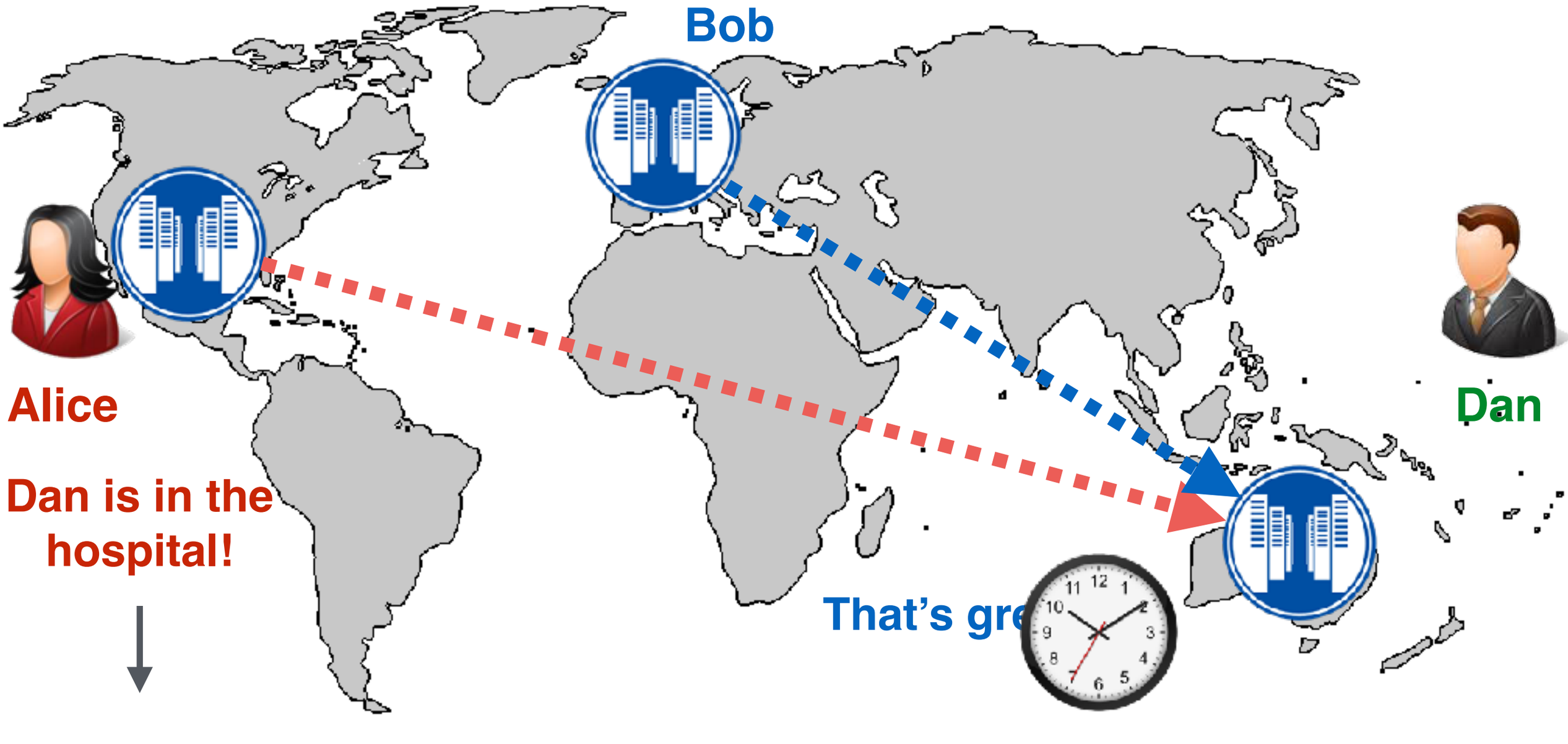
Dan is in the hospital!

That's great!

Dan

Dan is in the hospital! → Dan is ok! → That's great!

Bob



Alice

Dan is in the hospital!

Dan is ok!

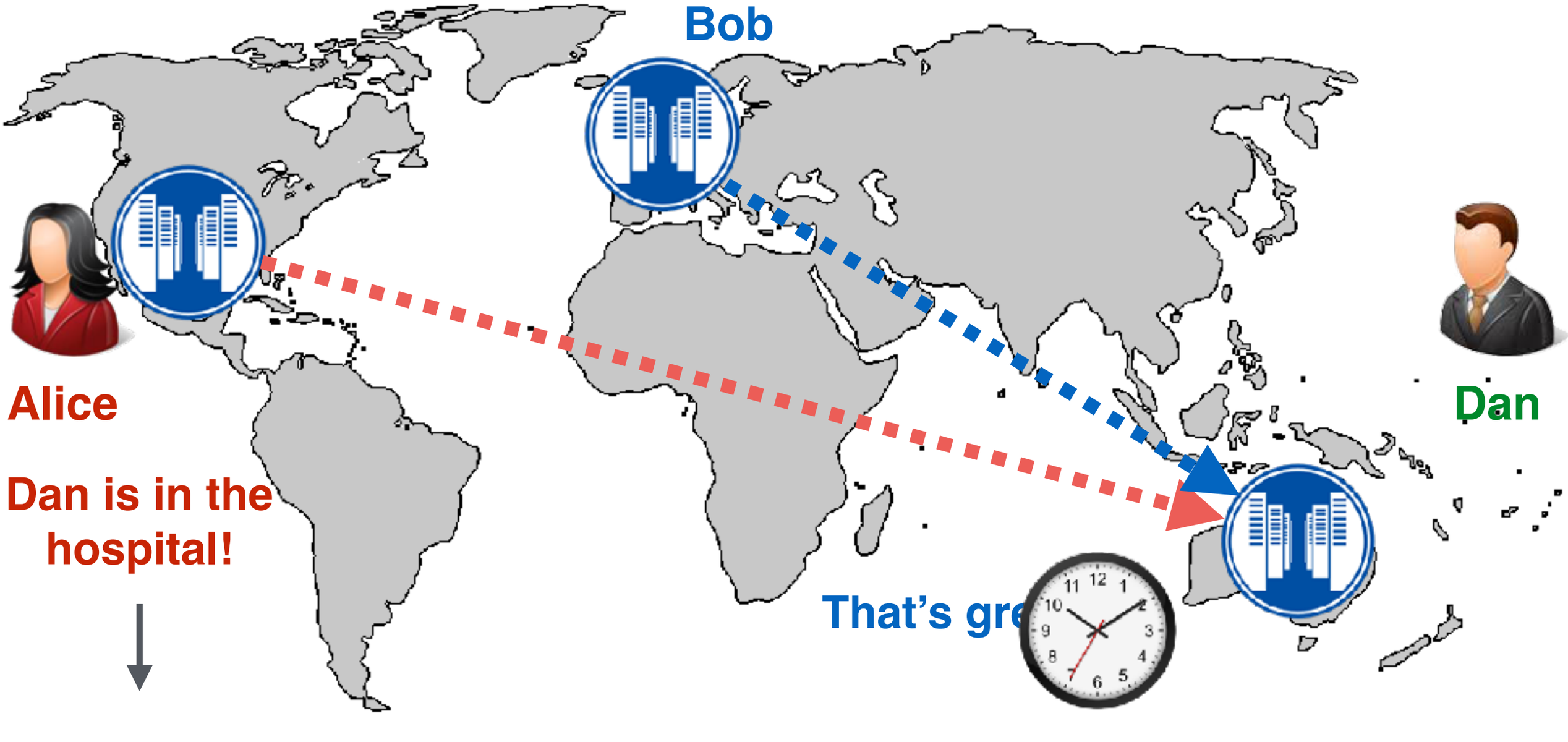
Dan is in the hospital!

That's gre

Dan

Dan is in the hospital! → Dan is ok! → That's great!

Bob



Alice

Dan

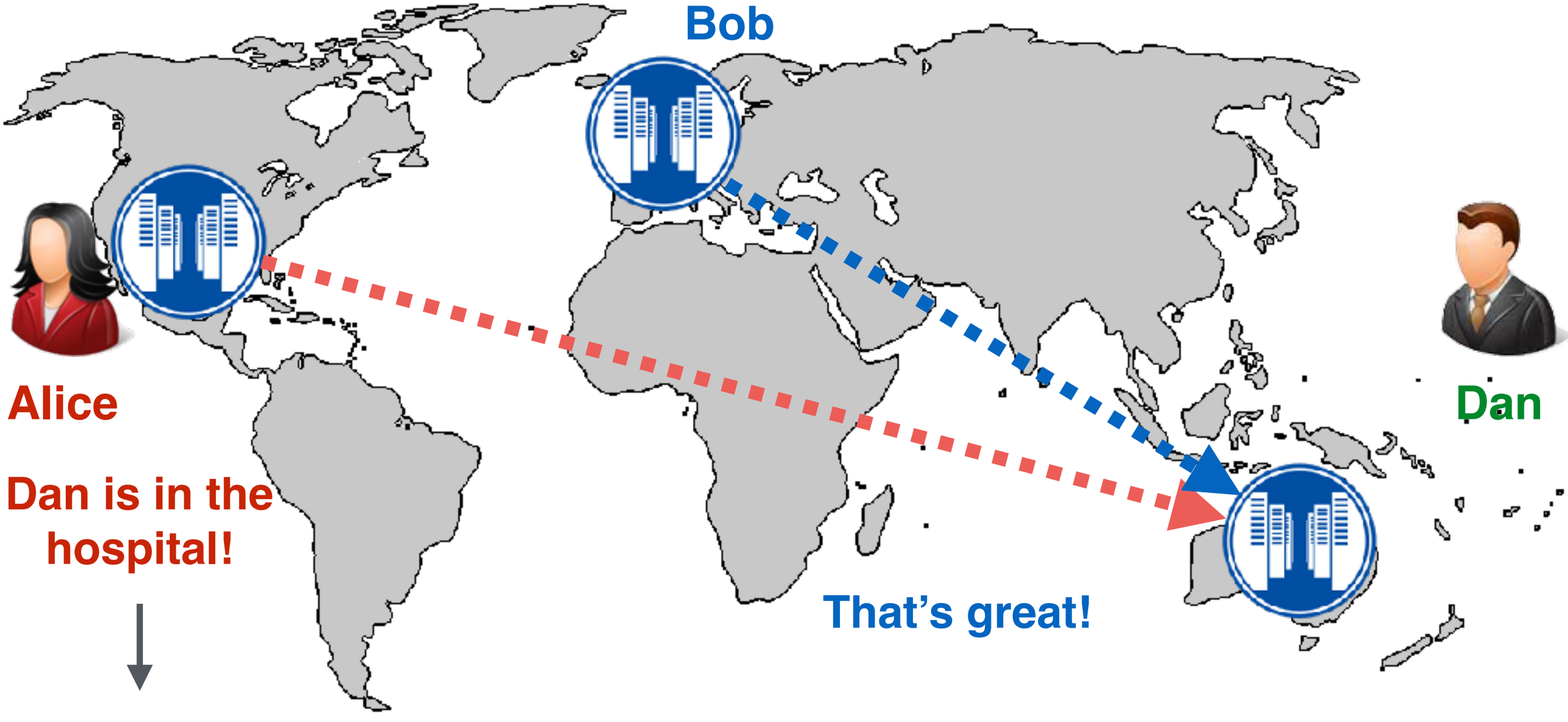
Dan is in the hospital!

Dan is ok!

That's gre

Dan is in the hospital! → Dan is ok!

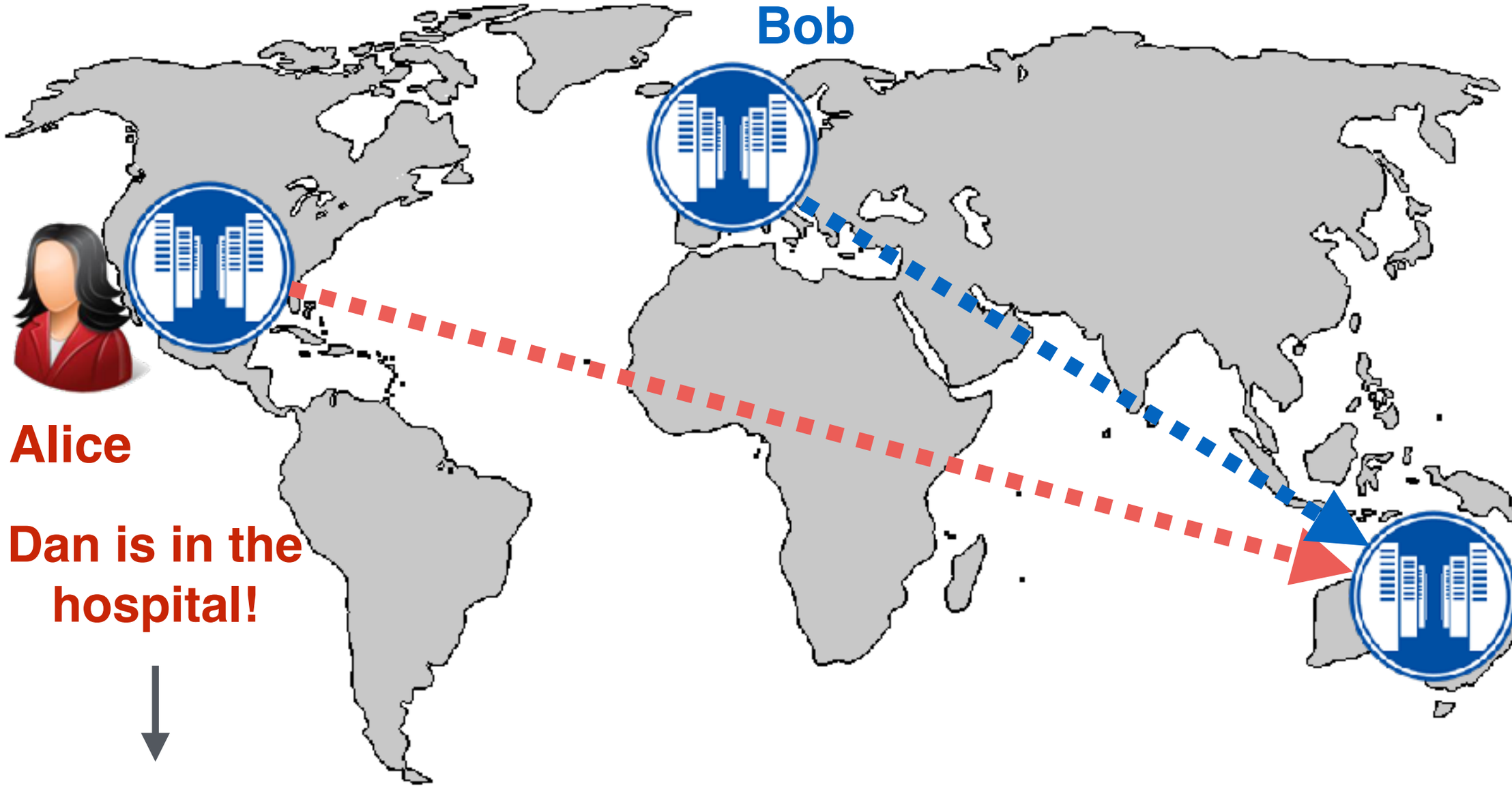
Dan is in the hospital! → Dan is ok! → That's great!



Dan is ok!

Dan is in the hospital! → Dan is ok!

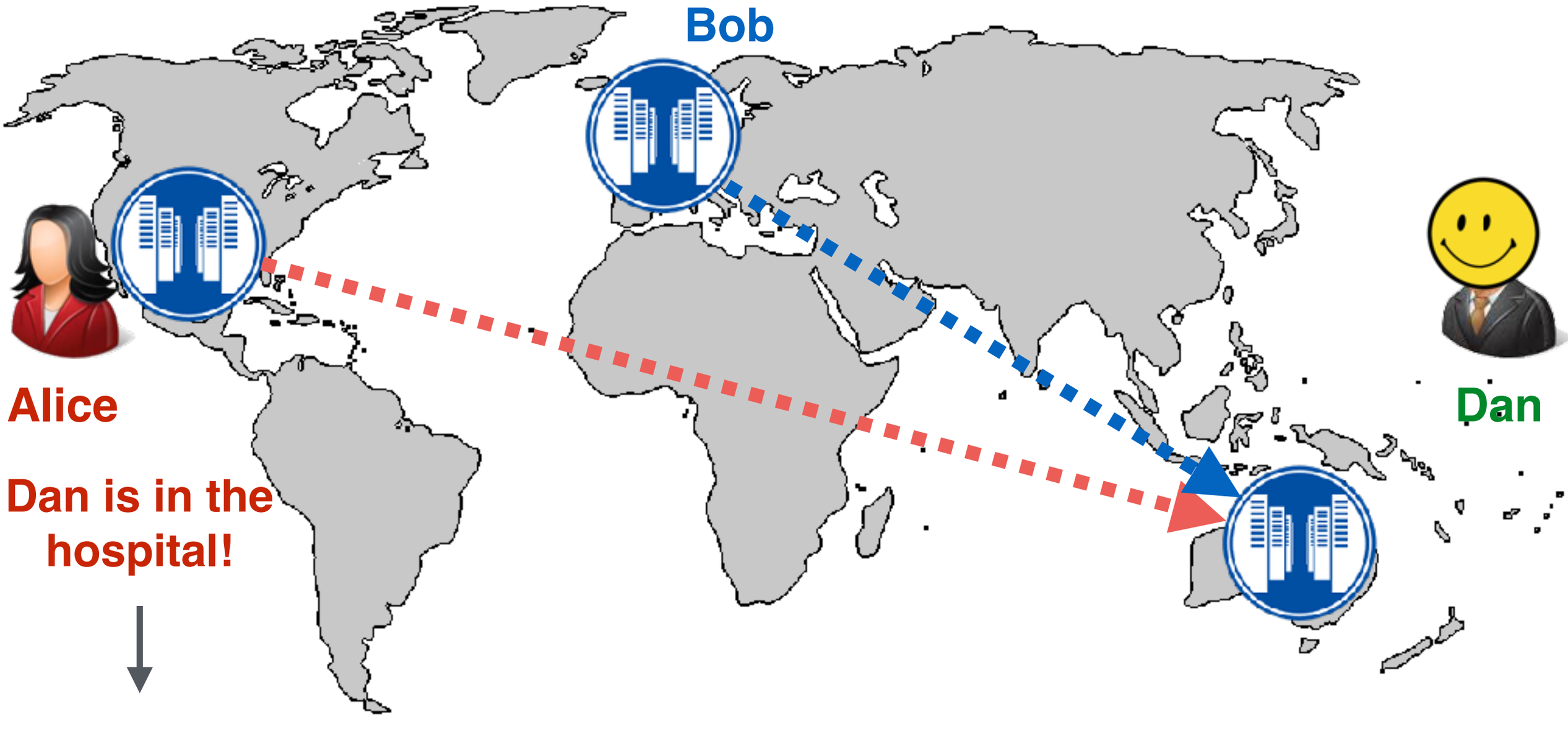
Dan is in the hospital! → Dan is ok! → That's great!



Dan is in the hospital! → Dan is ok! → That's great!

Dan is in the hospital! → Dan is ok! → That's great!

Bob



Alice

Dan is in the hospital!

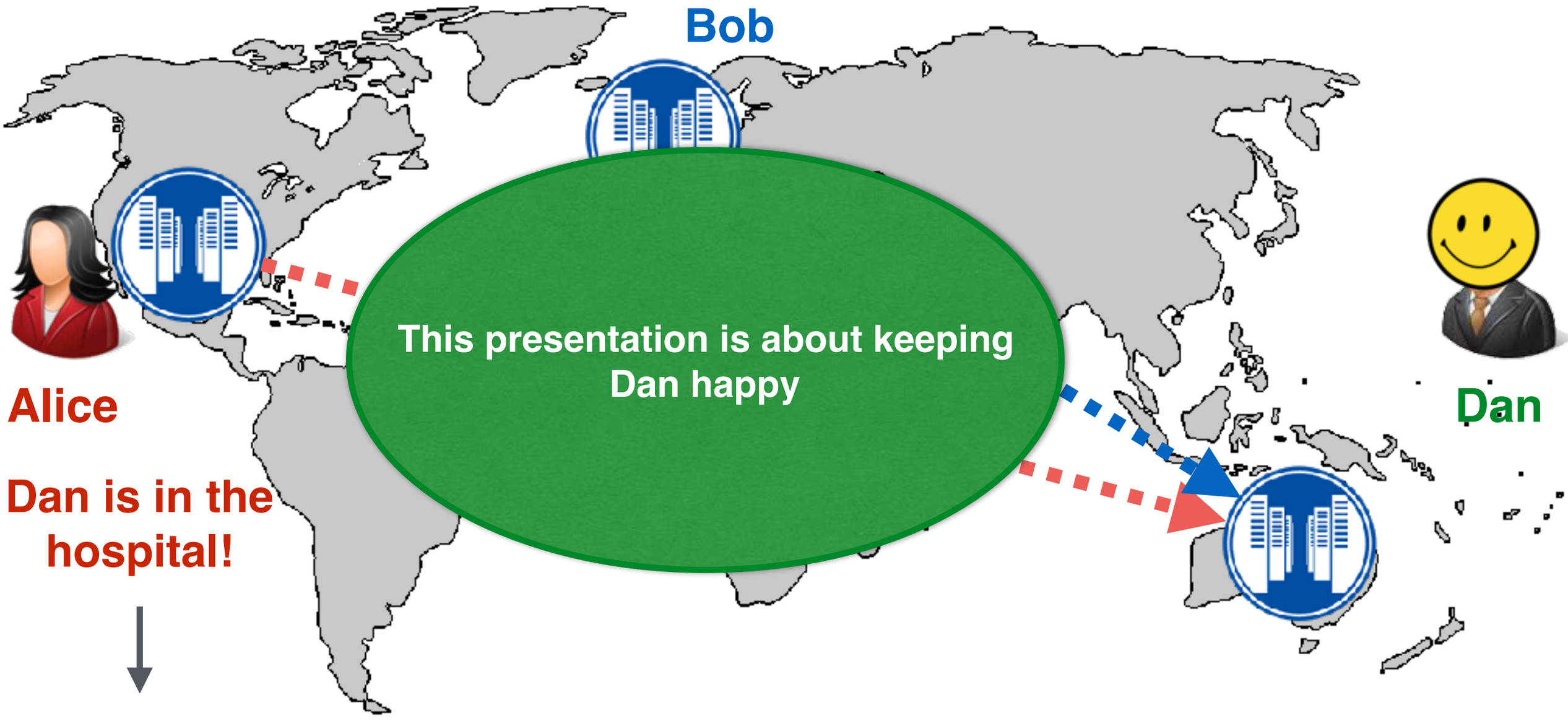
Dan is ok!

Dan

Dan is in the hospital! → Dan is ok! → That's great!

Dan is in the hospital! → Dan is ok! → That's great!

Bob



This presentation is about keeping Dan happy

Alice

Dan is in the hospital!

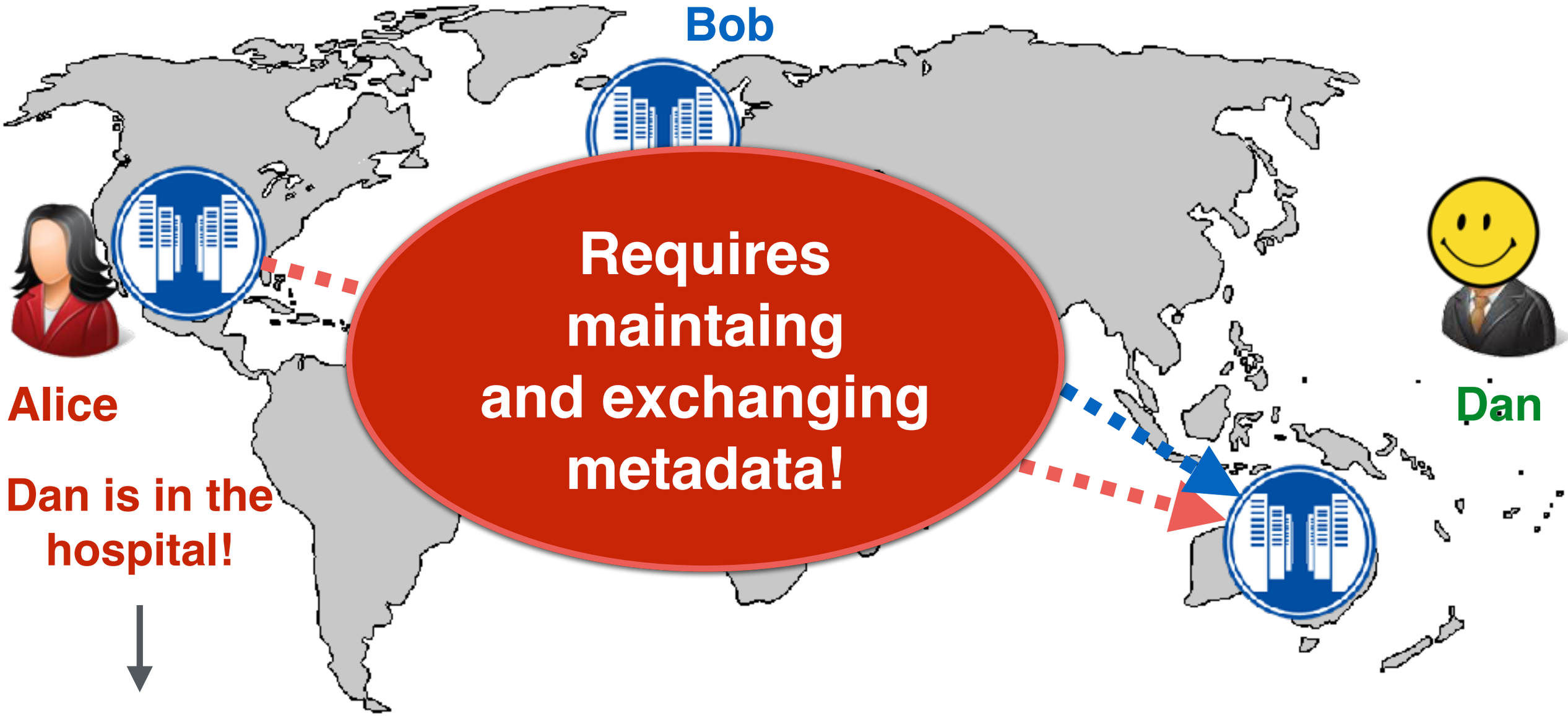
Dan is ok!

Dan

Dan is in the hospital! → Dan is ok! → That's great!

Dan is in the hospital! → Dan is ok! → That's great!

Bob



Requires maintaing and exchanging metadata!

Alice

Dan is in the hospital!

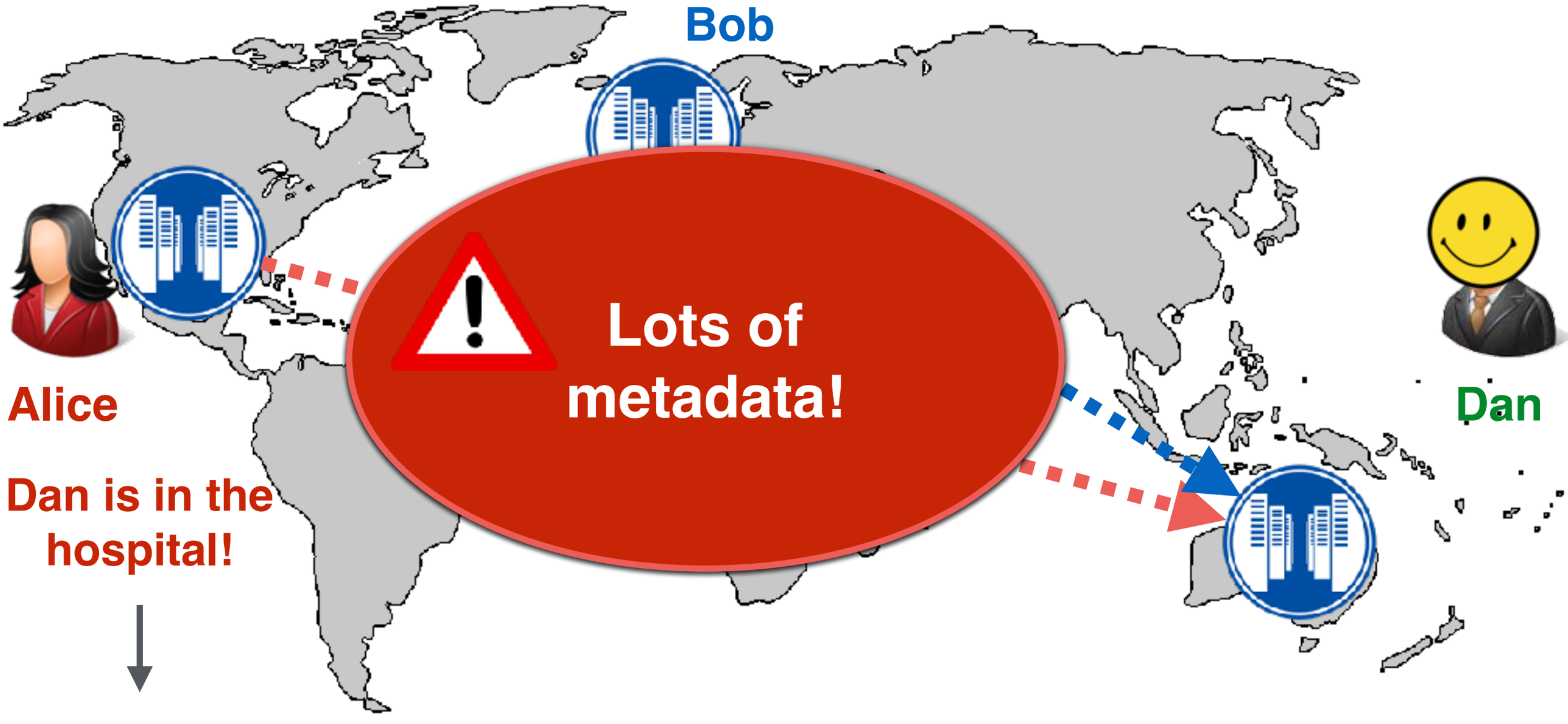
Dan is ok!

Dan

Dan is in the hospital! → Dan is ok! → That's great!

Dan is in the hospital! → Dan is ok! → That's great!

Bob



Lots of metadata!

Alice

Dan

Dan is in the hospital!

Dan is ok!

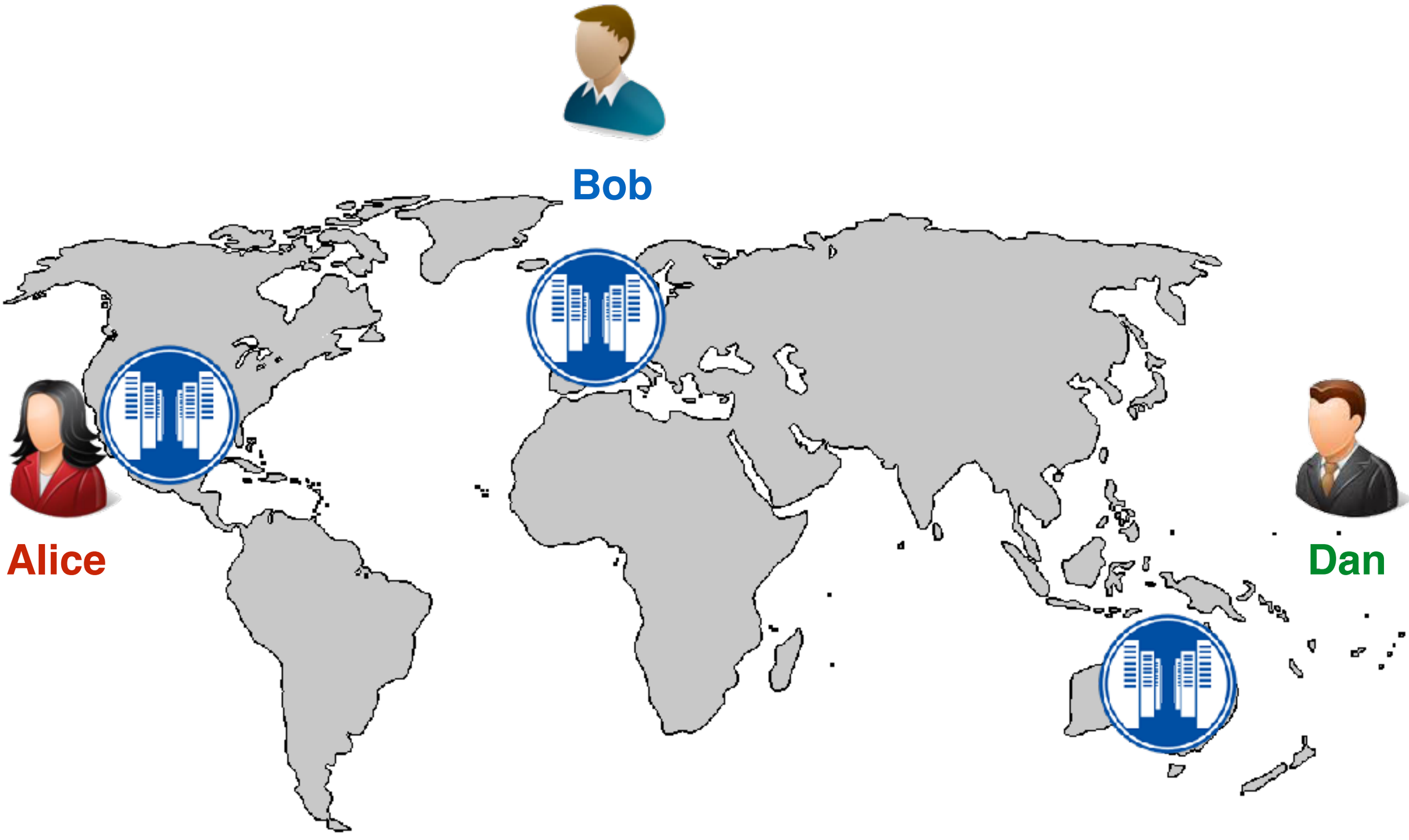
Dan is in the hospital! → Dan is ok! → That's great!

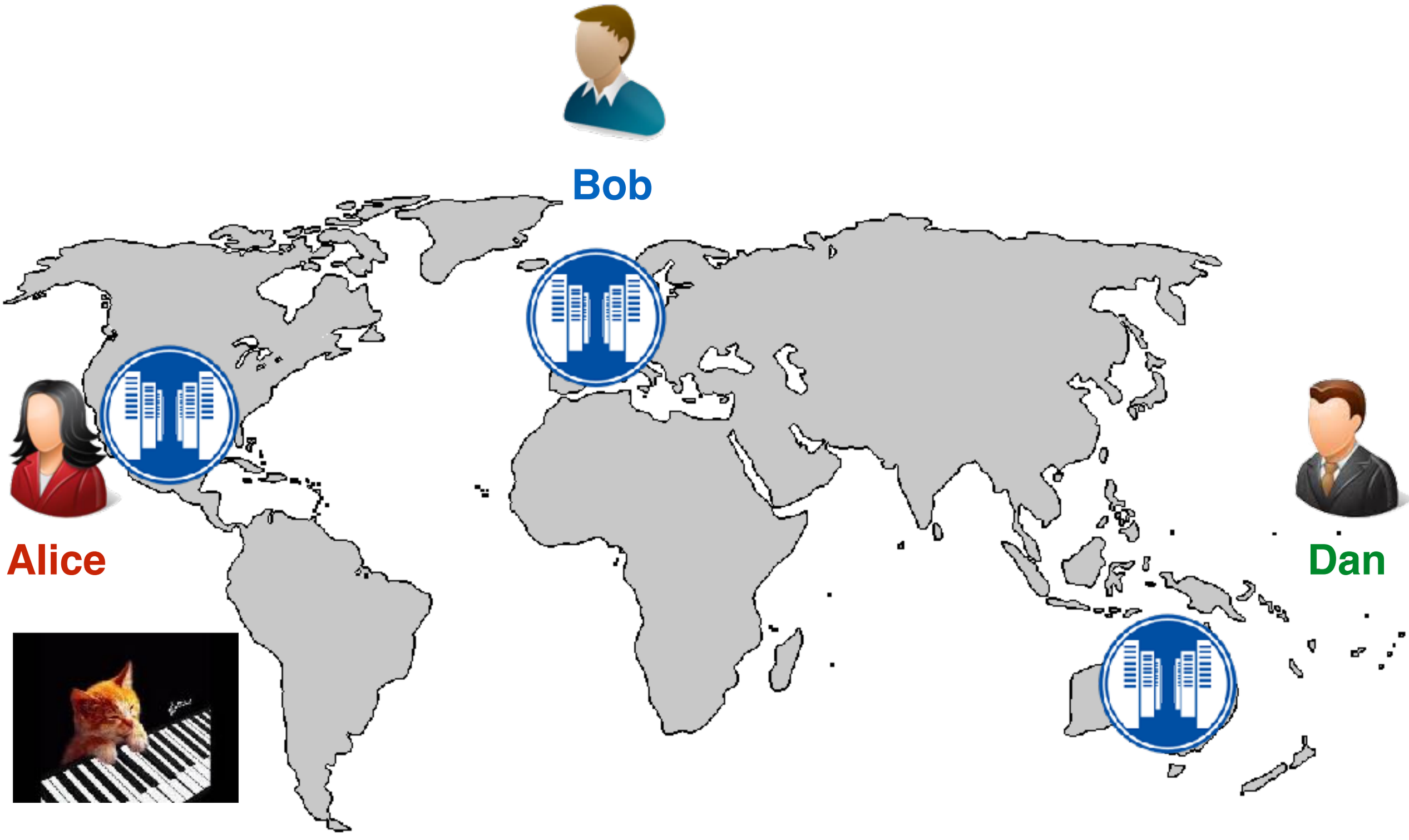
Causal consistency

Problem

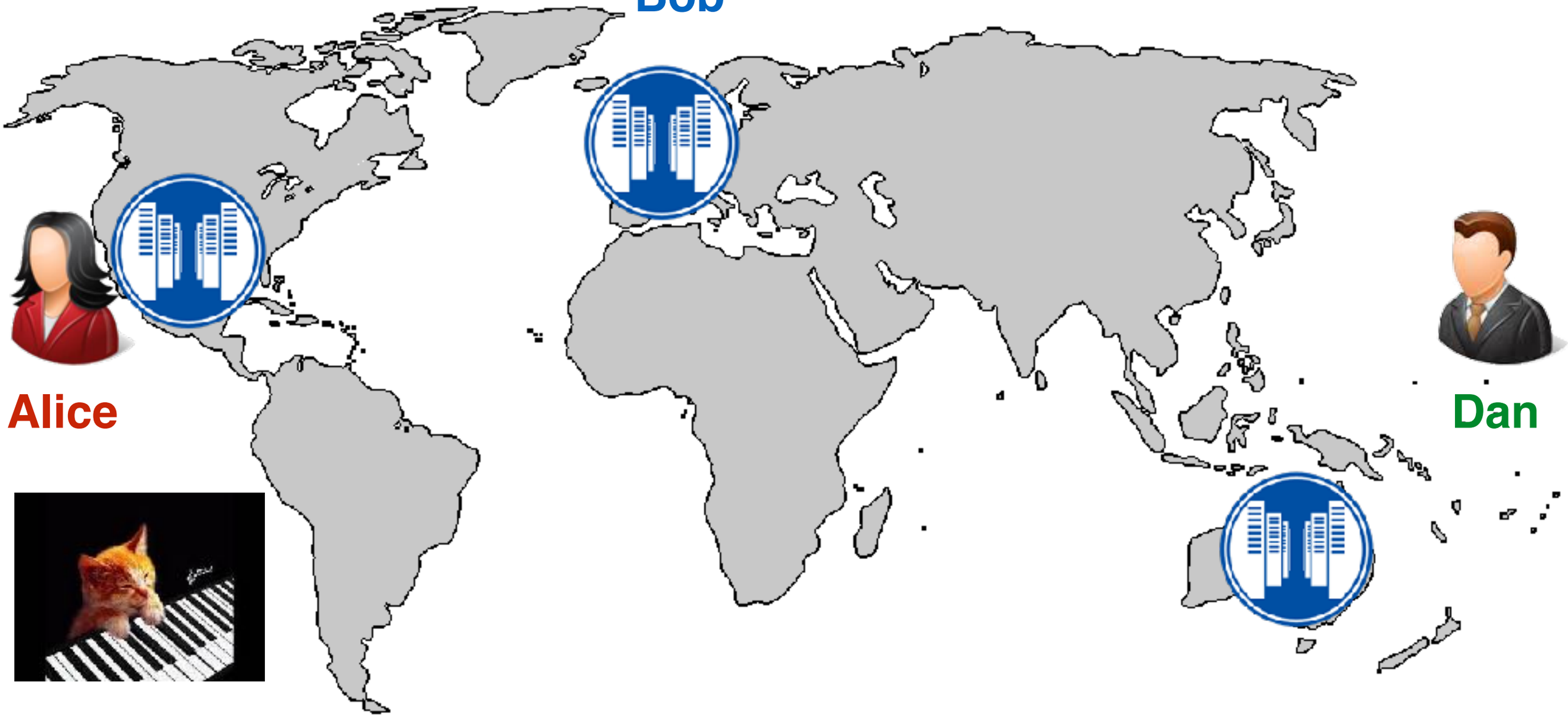
If causal dependencies are not accurately tracked metadata may generate false positives

False dependencies!


Bob
Alice
Dan


Bob
Alice
Dan

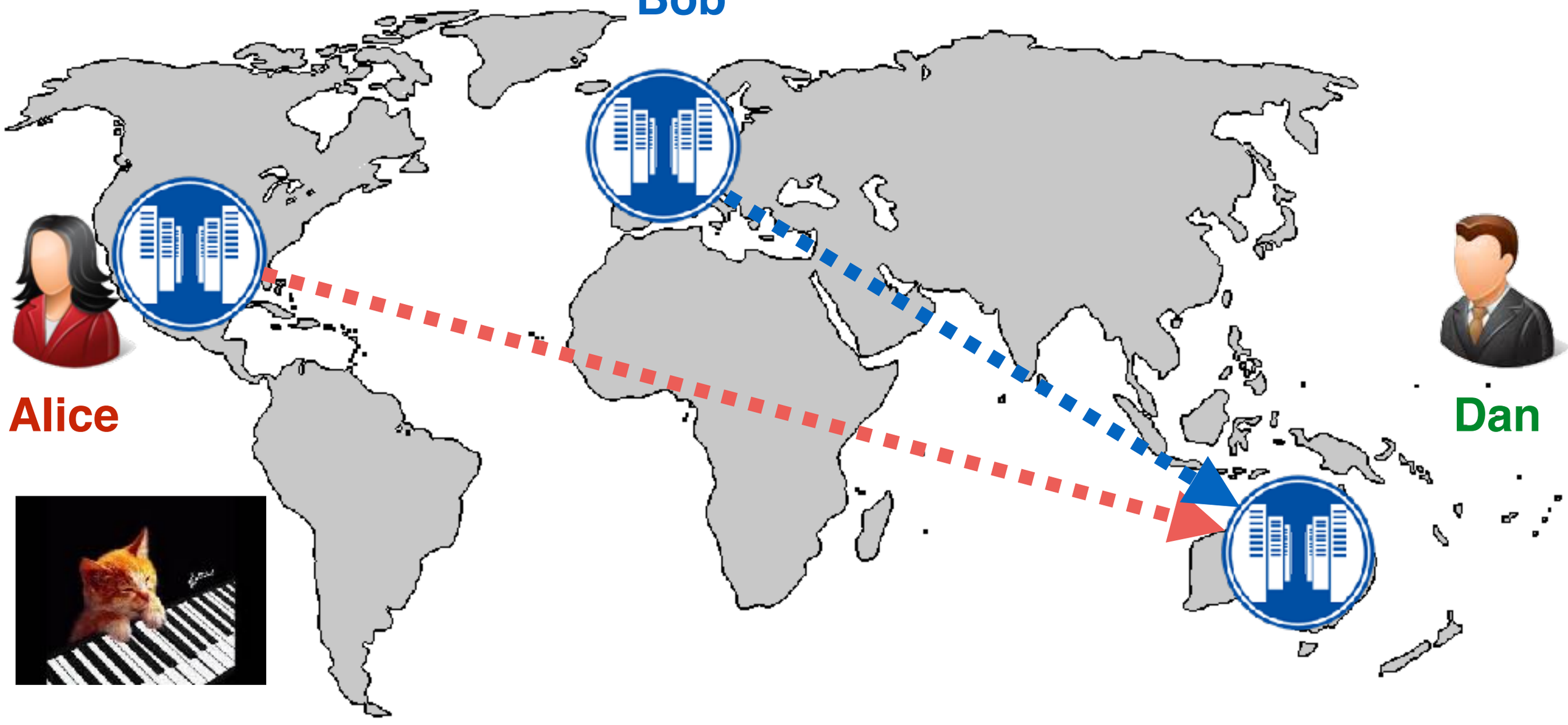
Bob



Alice

Dan

Bob



Alice

Dan

Bob

Alice

Dan

Bob

Alice

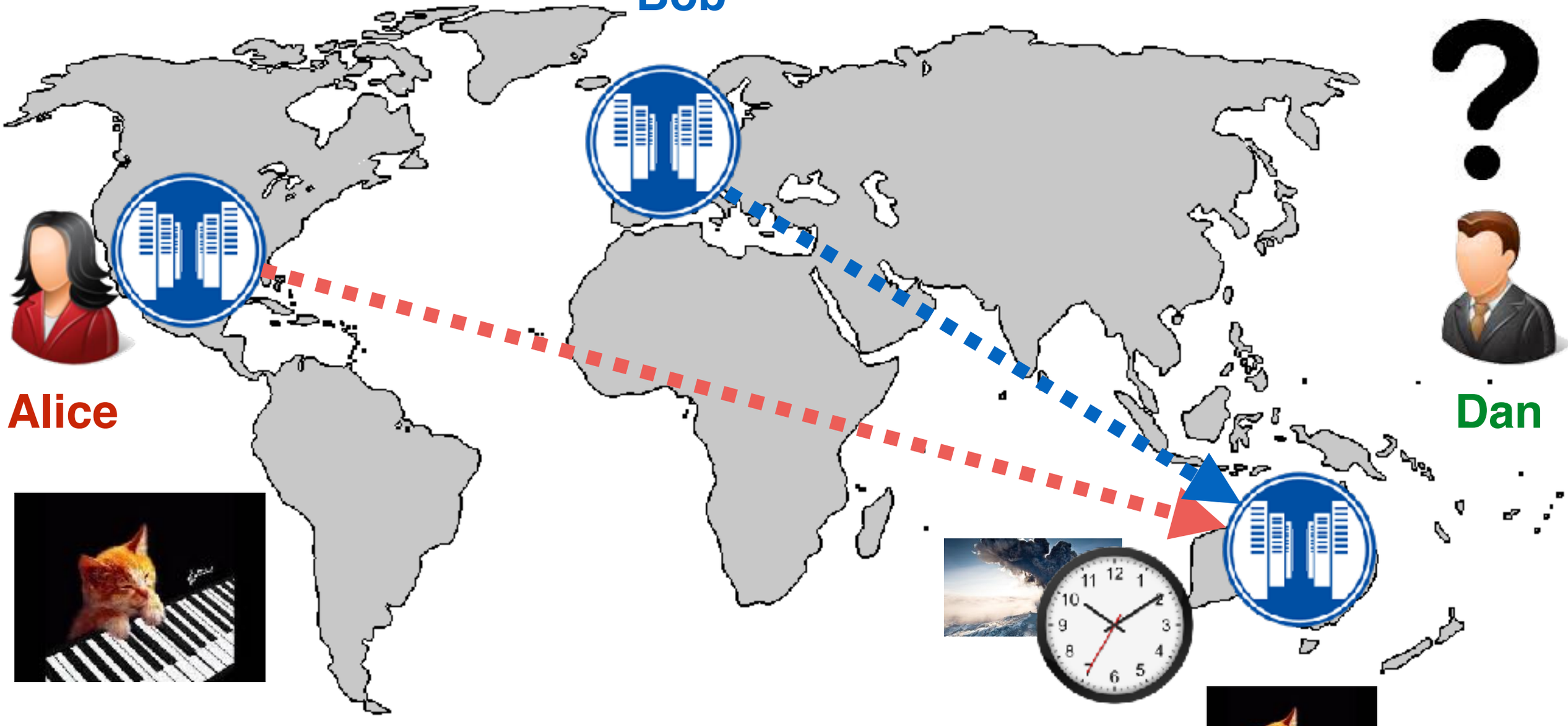
Dan

Bob

Alice

Dan

Bob



Bob

Alice

Dan

Metadata

more metadata

less metadata

Metadata

Matrix/vector clocks

more metadata

less metadata

Metadata

Matrix/vector clocks

more metadata

less metadata

One vector per item.
One entry in each vector per DC.

Metadata

Matrix/vector clocks

more metadata

less metadata

precise

expensive

Metadata

Matrix/vector clocks **Lamport's clocks**

more metadata less metadata

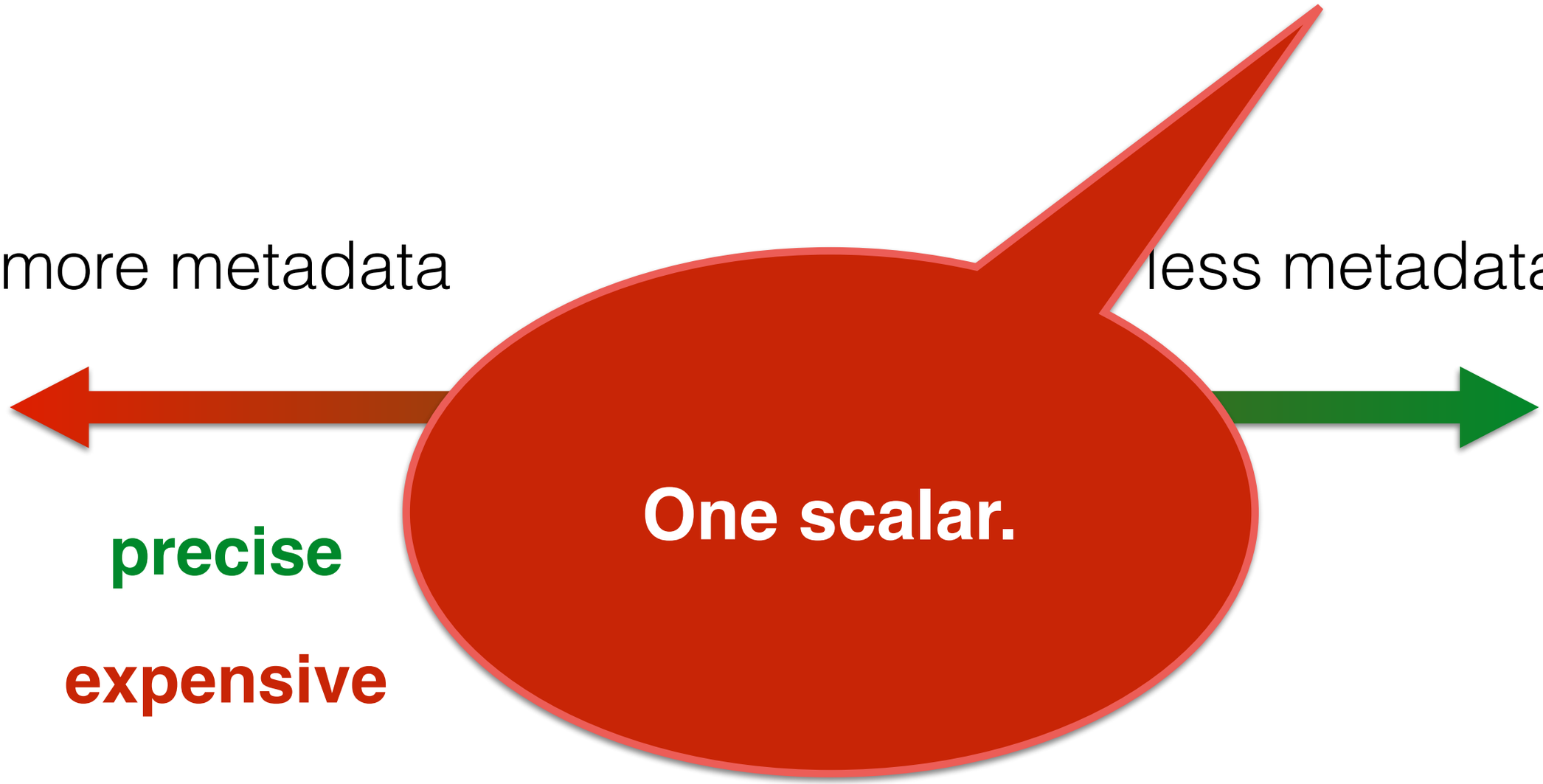
precise

expensive

Metadata

Matrix/vector clocks

Lamport's clocks



Metadata

Matrix/vector clocks — **Lamport's clocks**

more metadata — less metadata

precise — **false positives**

expensive — **cheap**

Problems of the previous state-of-the-art

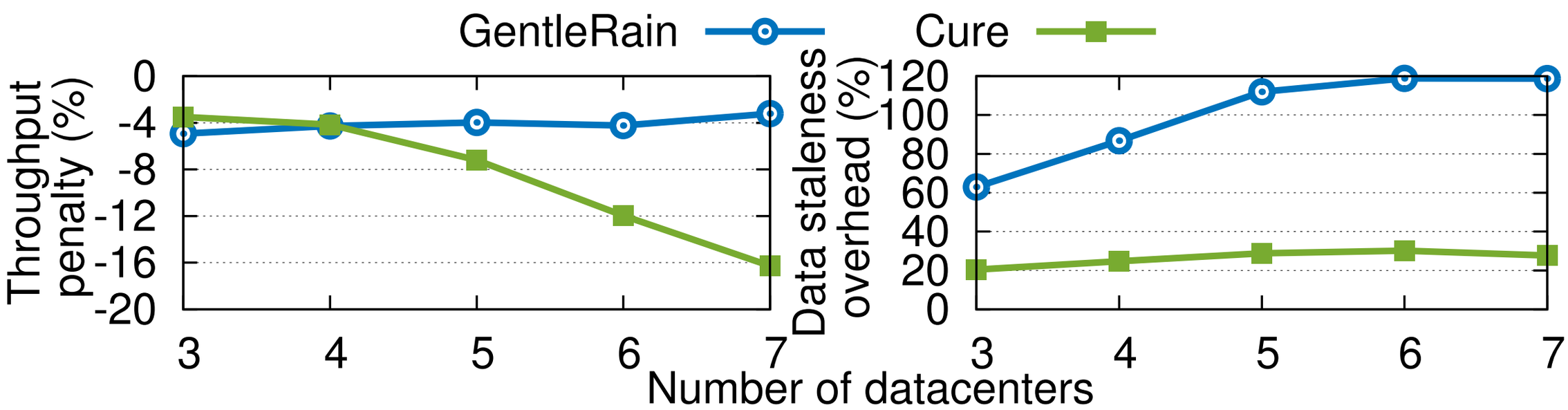
Throughput vs. data staleness tradeoff

GentleRain [SoCC' 14]: Optimizes throughput

Compresses metadata into a scalar

Cure [ICDCS' 16]: Optimizes data freshness

Relies on a vector clock with an entry per data center



Problems of the previous state-of-the-art

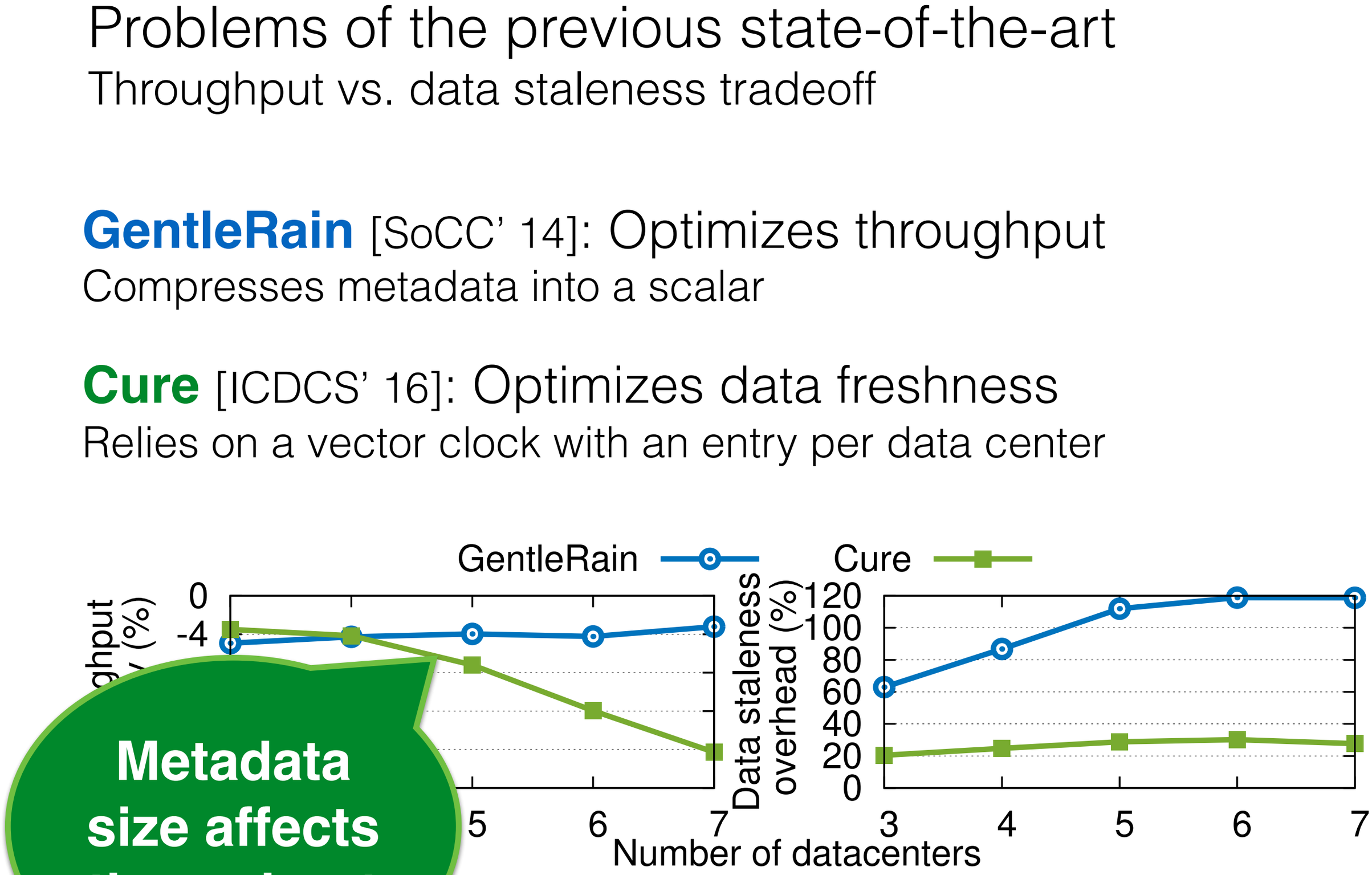
Throughput vs. data staleness tradeoff

GentleRain [SoCC' 14]: Optimizes throughput

Compresses metadata into a scalar

Cure [ICDCS' 16]: Optimizes data freshness

Relies on a vector clock with an entry per data center



Problems of the previous state-of-the-

Throughput vs. data staleness tradeoff

GentleRain [SoCC' 14]: Optimizes throughput

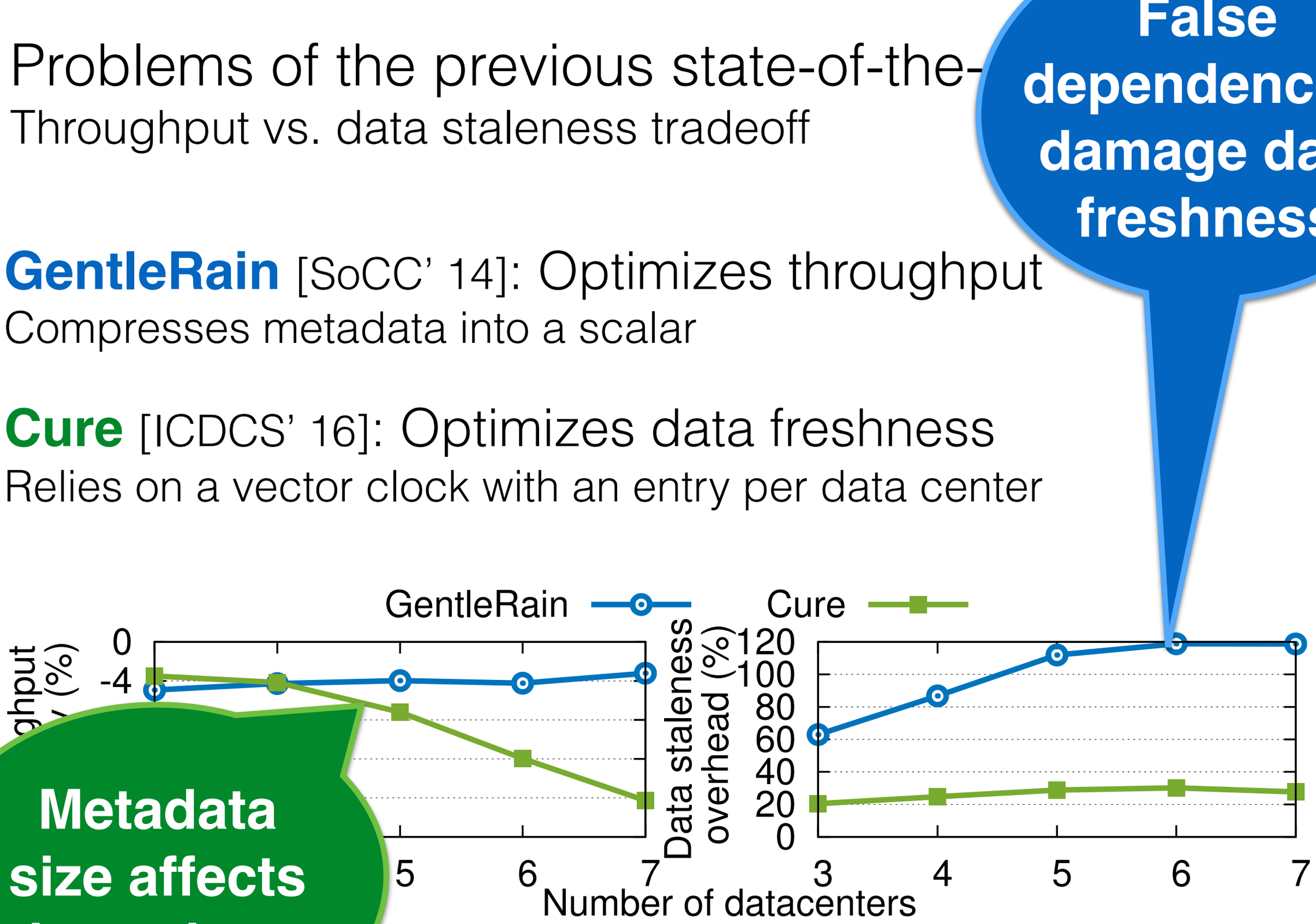
Compresses metadata into a scalar

Cure [ICDCS' 16]: Optimizes data freshness

Relies on a vector clock with an entry per data center

False dependencies damage data freshness

Metadata size affects throughput



Problems of the previous state-of-the-art

Throughput vs. data staleness tradeoff

Partial replication aggravates the problem

Visibility latencies have direct impact on client response latency

Our take on the problem

Saturn

+

Eunomia

Our take on the problem

Saturn

+

Eunomia

God in ancient Roman religion,
that become the god of time

Our take on the problem

+

Saturn

Eunomia

Greek goddess of law and legislation

Our take on the problem

Saturn

+

Eunomia

If this couple cannot fix the problem, nobody can…

Our take on the problem

Saturn

+

Eunomia

Our take on the problem

Saturn

Eunomia

Orders events across datacenters

Our take on the problem

Saturn

+

Eunomia

Orders events in each datacenter

Distributed metadata service

pluggable to existing geo-distributed data services

handles the dissemination of operations among data centers

Ensures that

clients always observe a causally consistent state

with a negligible performance overhead when compared to an eventually consistency system

key features

key features

Requires a **constant and small** amount of metadata
regardless of the system's scale (servers, partitions, and locations)

key features

Requires a **constant and small** am
regardless of the system's scale (server

to avoid impairing
throughput

key features

Requires a **constant and small** am
regardless of the system's scale (server

to avoid impairing
throughput

Mitigates the impact of **false dependencies**
by relying on a tree-based dissemination

key features

Requires a **constant and small** am
regardless of the system's scale (server

to avoid impairing
throughput

Mitigates the impact of **false depe**
by relying on a tree-based dissemination

to enhance
data freshness

key features

Requires a **constant and small** am
regardless of the system's scale (server

to avoid impairing
throughput

Mitigates the impact of **false depe**
by relying on a tree-based dissemination

to enhance
data freshness

Implements **genuine partial replication**
data centers only manage data and metadata of the items replicated locally

key features

Requires a **constant and small** am
regardless of the system's scale (server

to avoid impairing
throughput

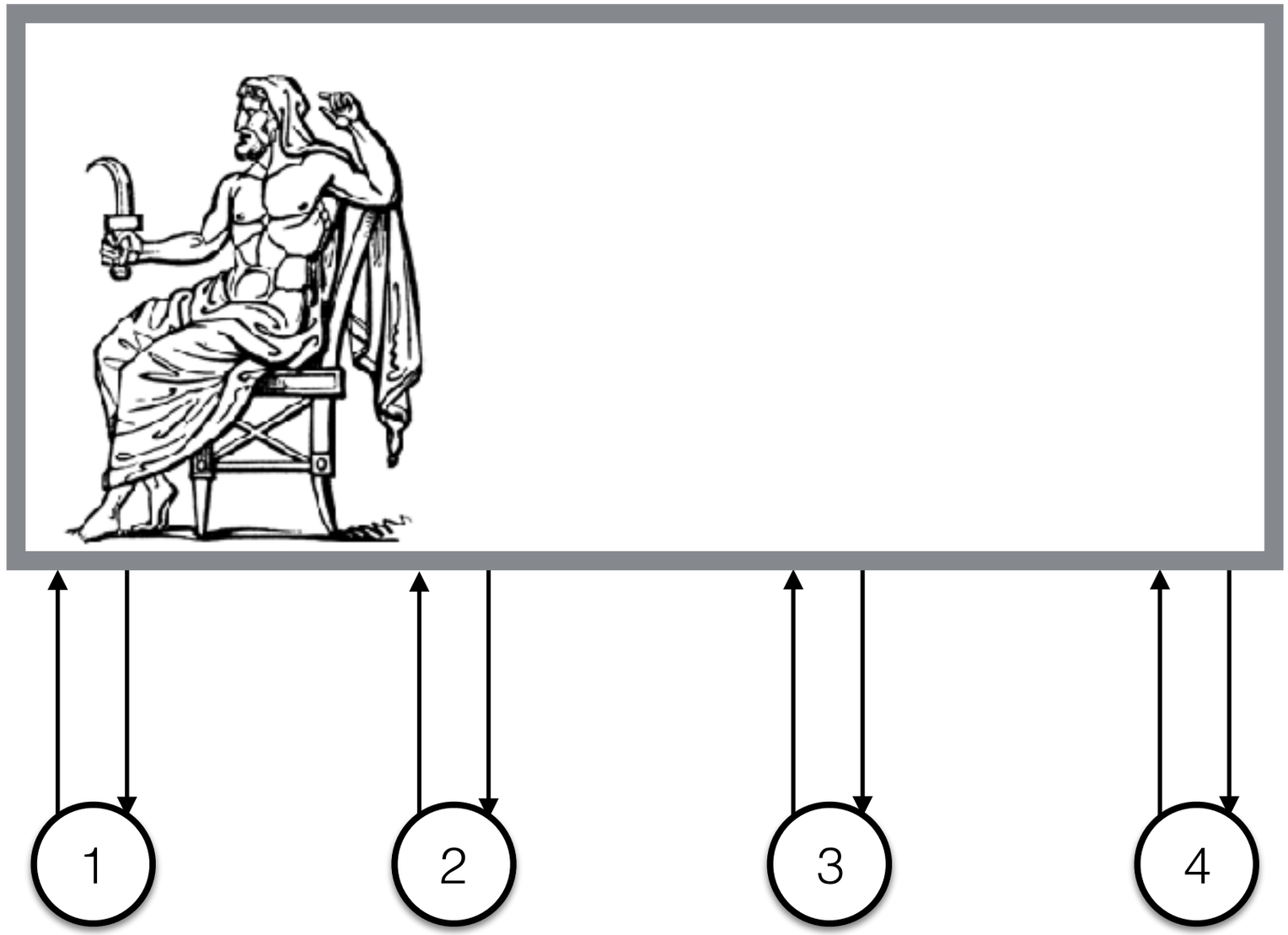
Mitigates the impact of **false depe**
by relying on a tree-based dissemination

to enhance
data freshness

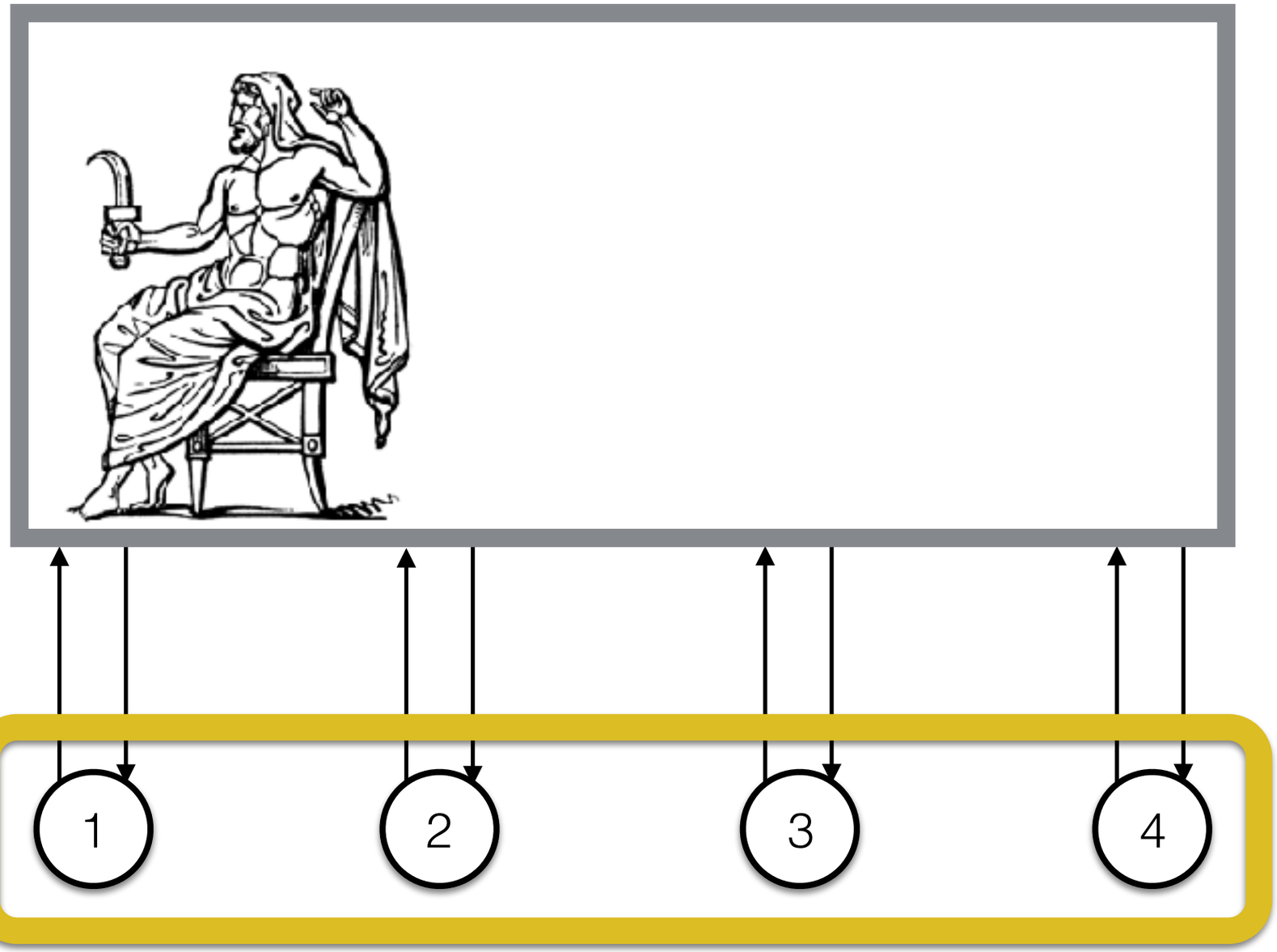
Implements **genuine partial replica**
data centers only manage data and meta
replicated locally

to take full advantage of
partial replication

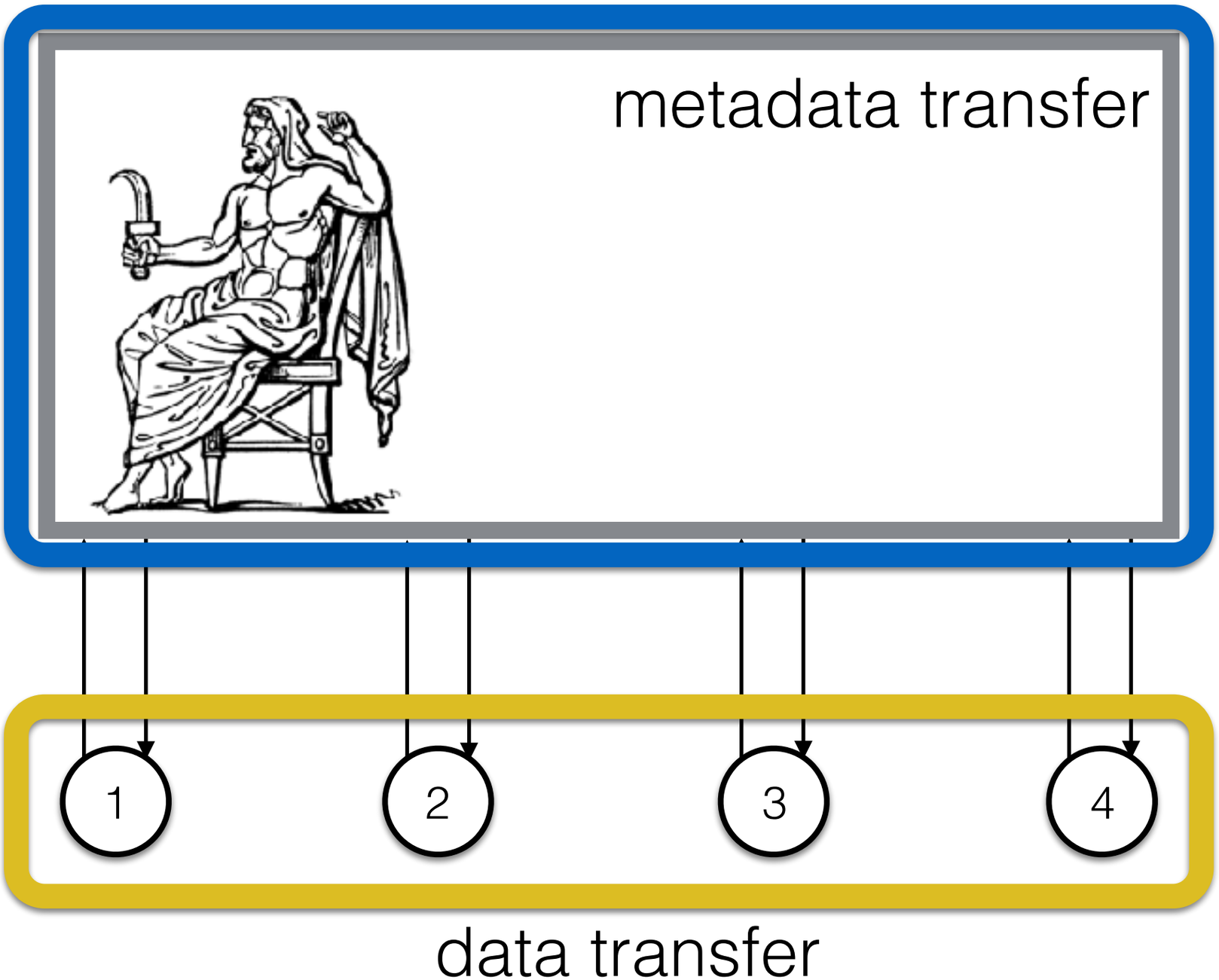
Decoupling data and metadata



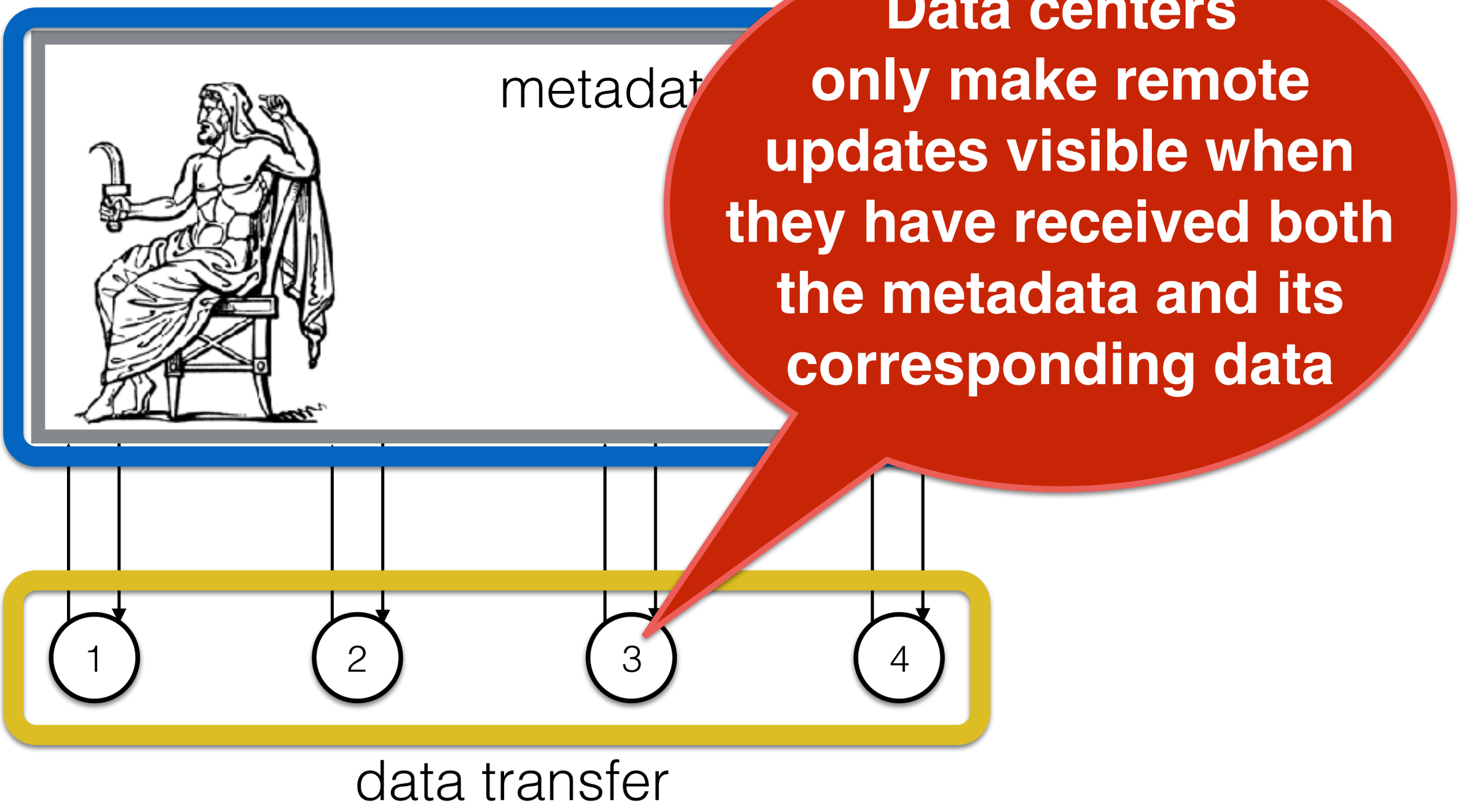
Decoupling data and metadata



Decoupling data and metadata



Decoupling data and metadata

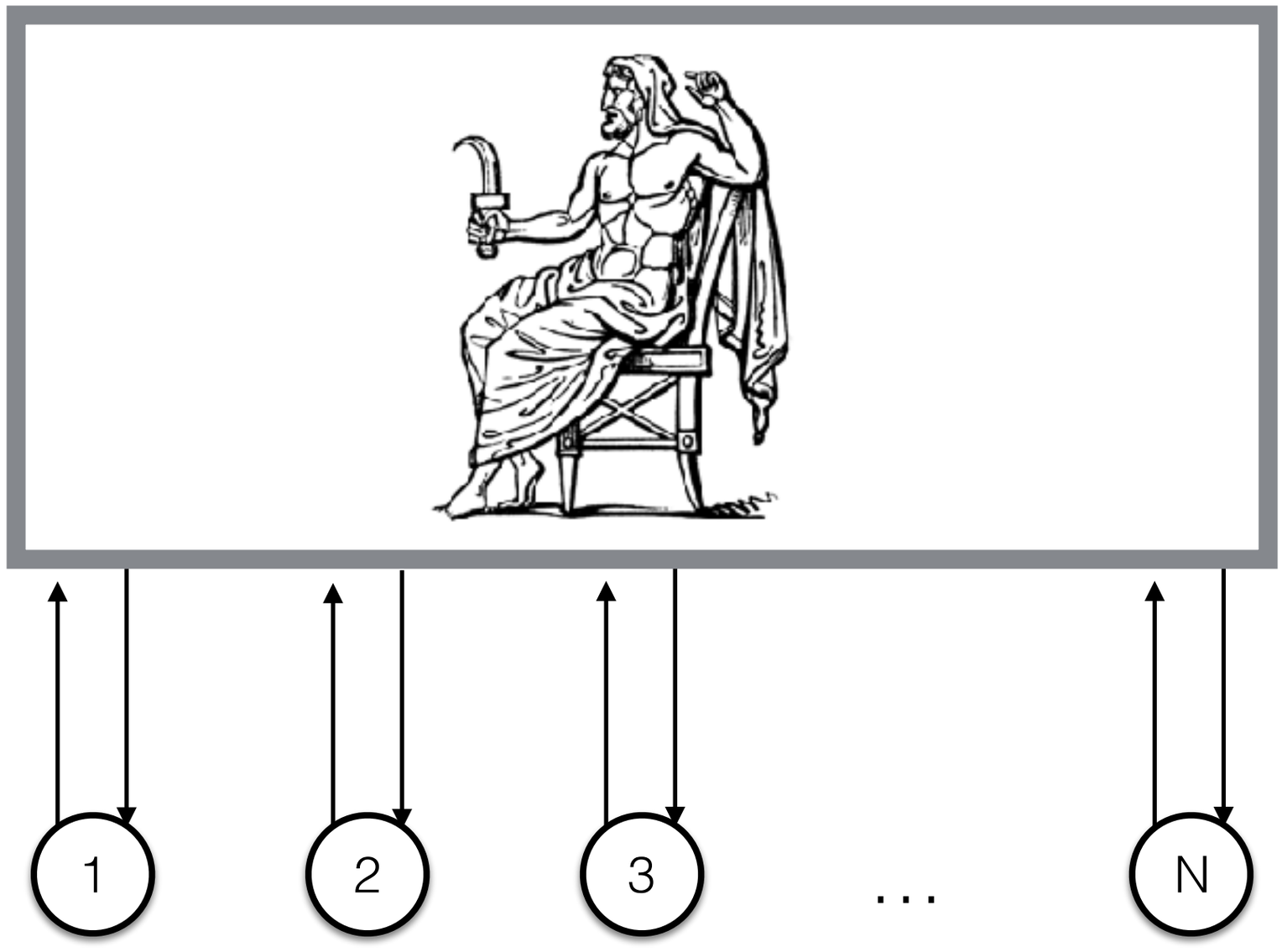


Example: write request

Example: write request

data

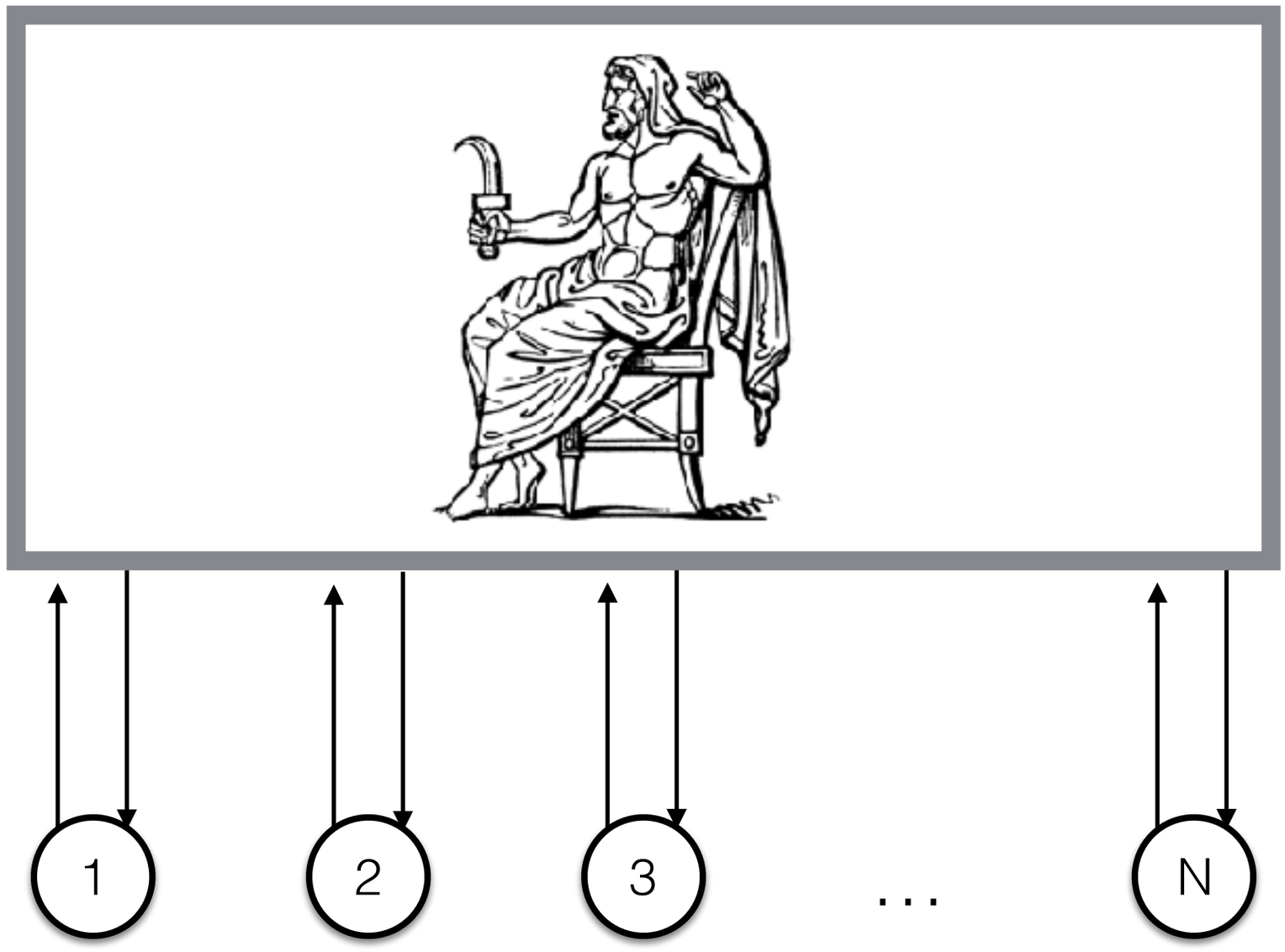
labels



Example: write request

data

labels



Client 1

1

2

3

...

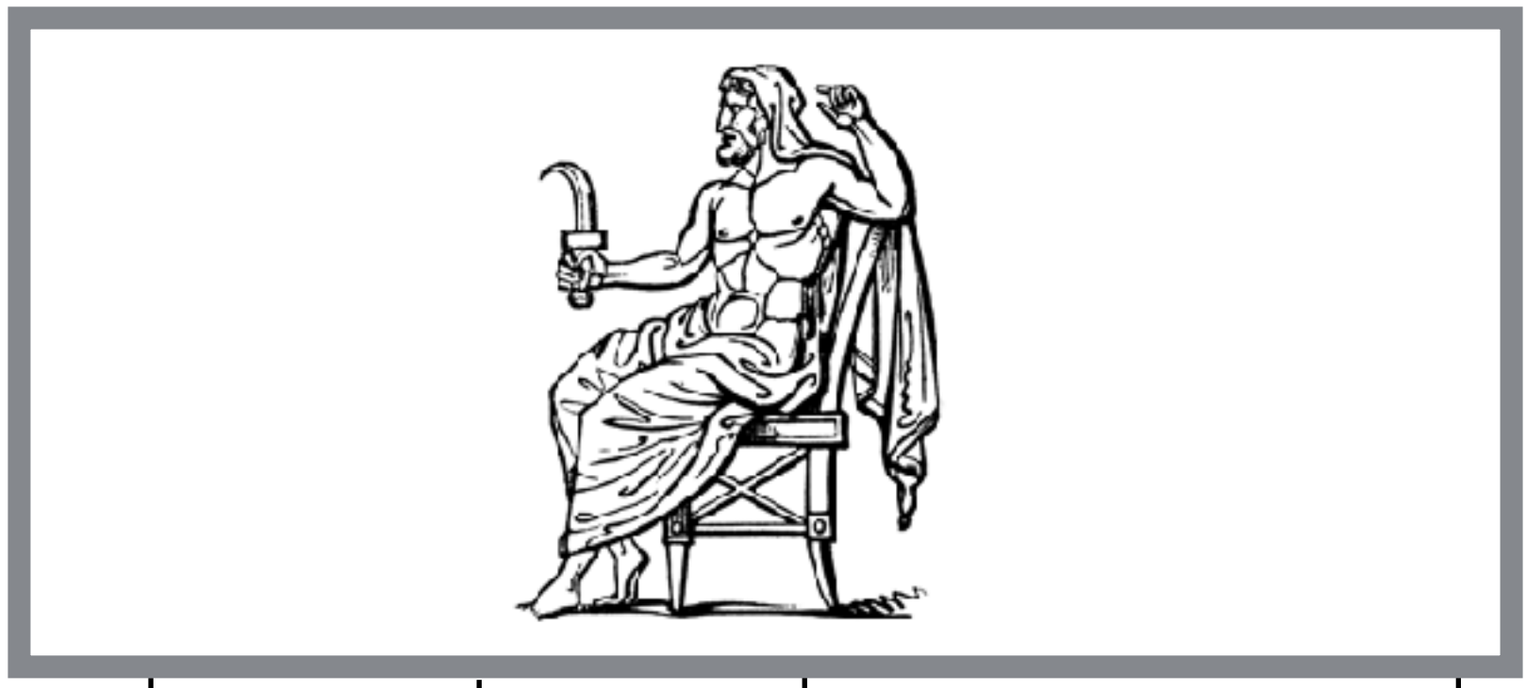
N

data centers

Example: write request

data

labels



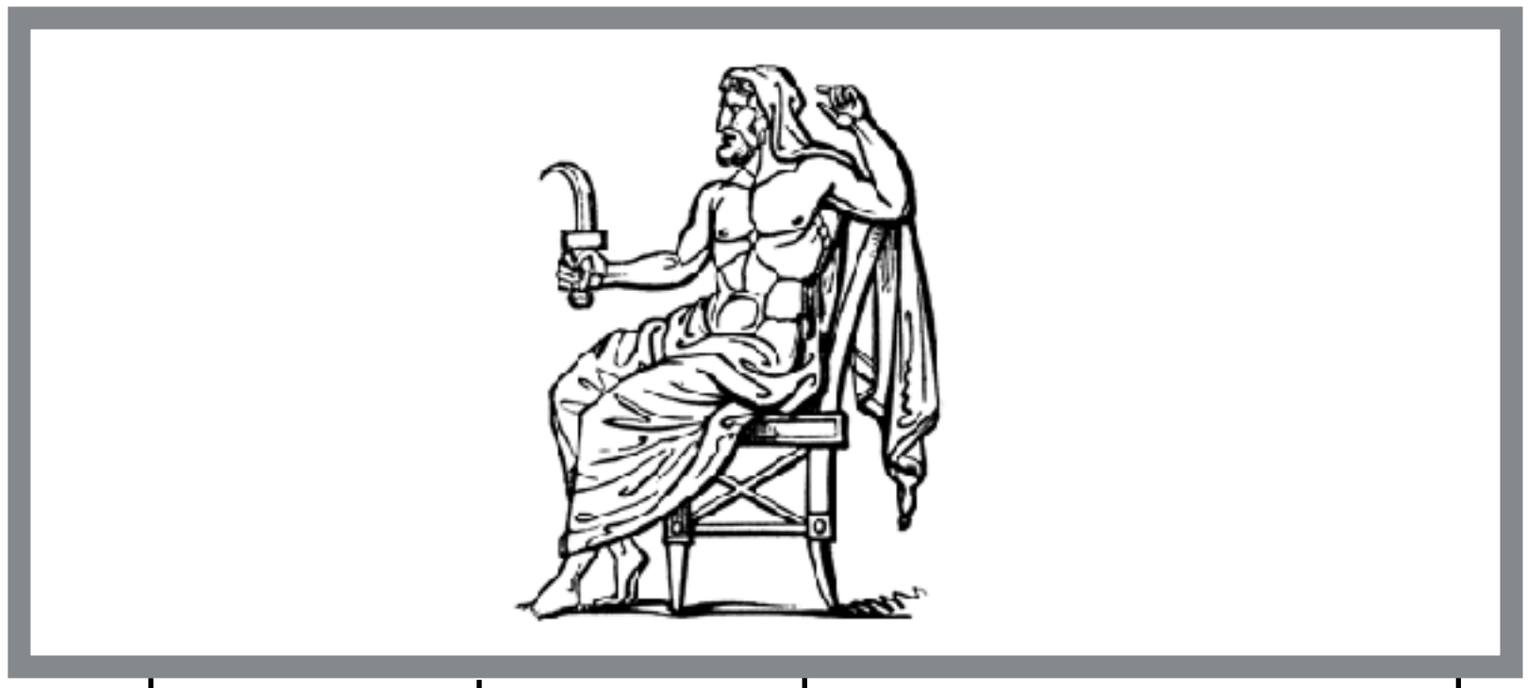
Client 1 — put(a_1) → 1 2 3 … N

data centers

Example: write request

data

labels



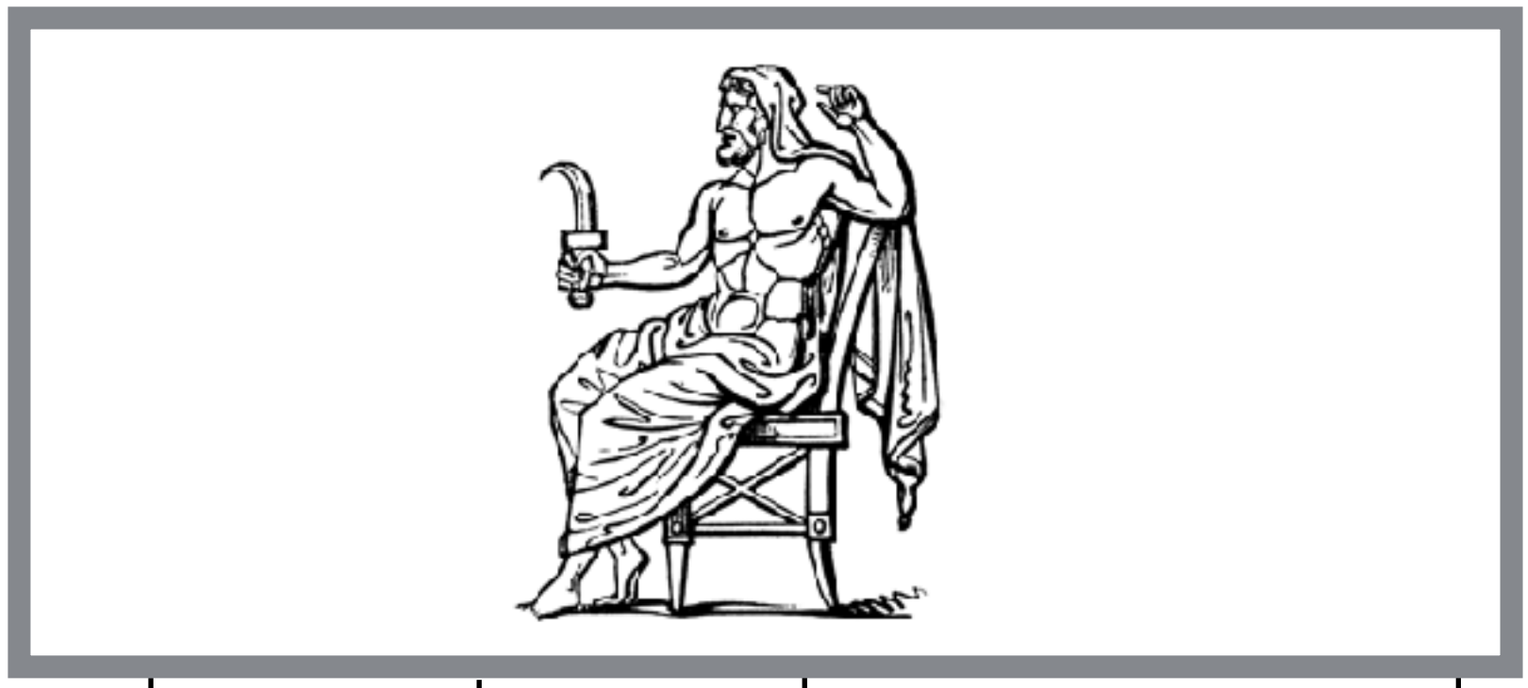
Client 1 — put(a_1) → 1 2 3 … N

data centers

Example: write request

data

labels



Client 1 — put(a_1) → 1

a_1

2

3

…

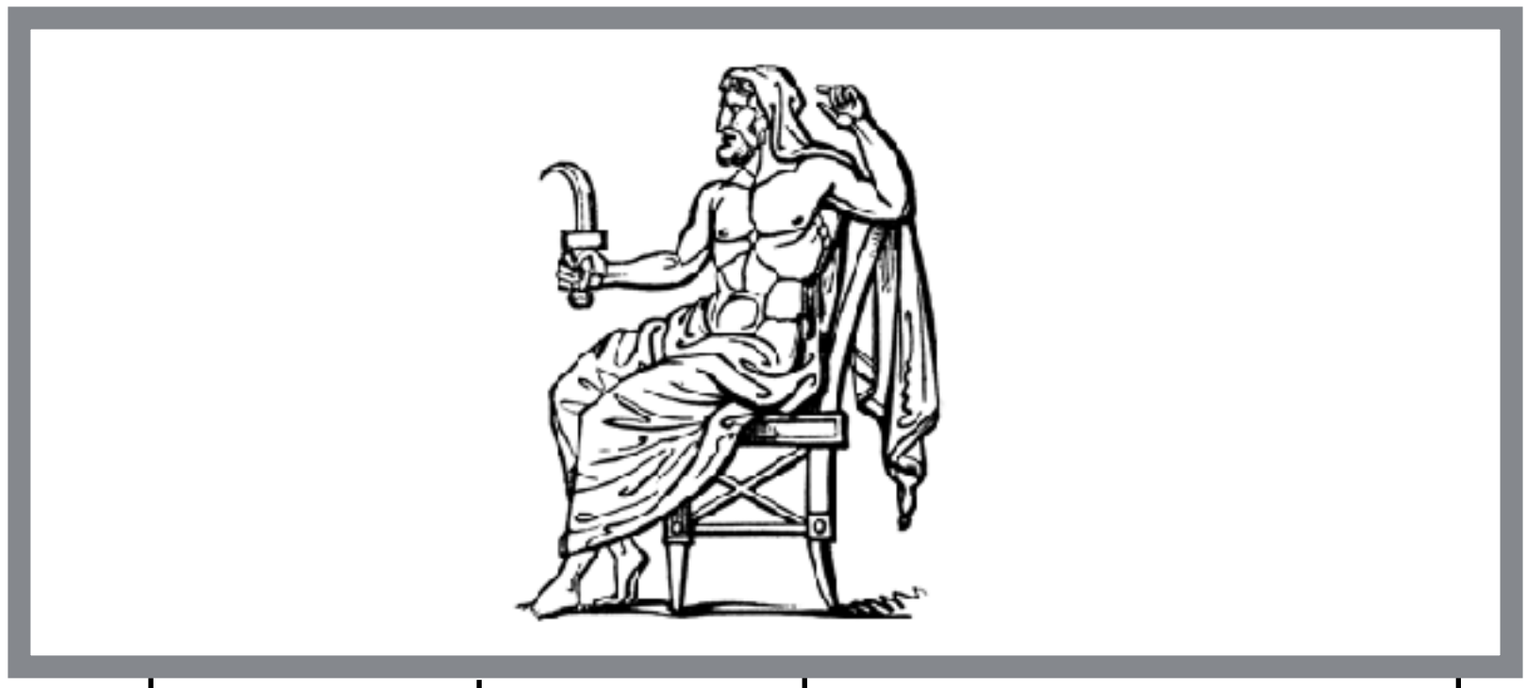
N

data centers

Example: write request

data

labels



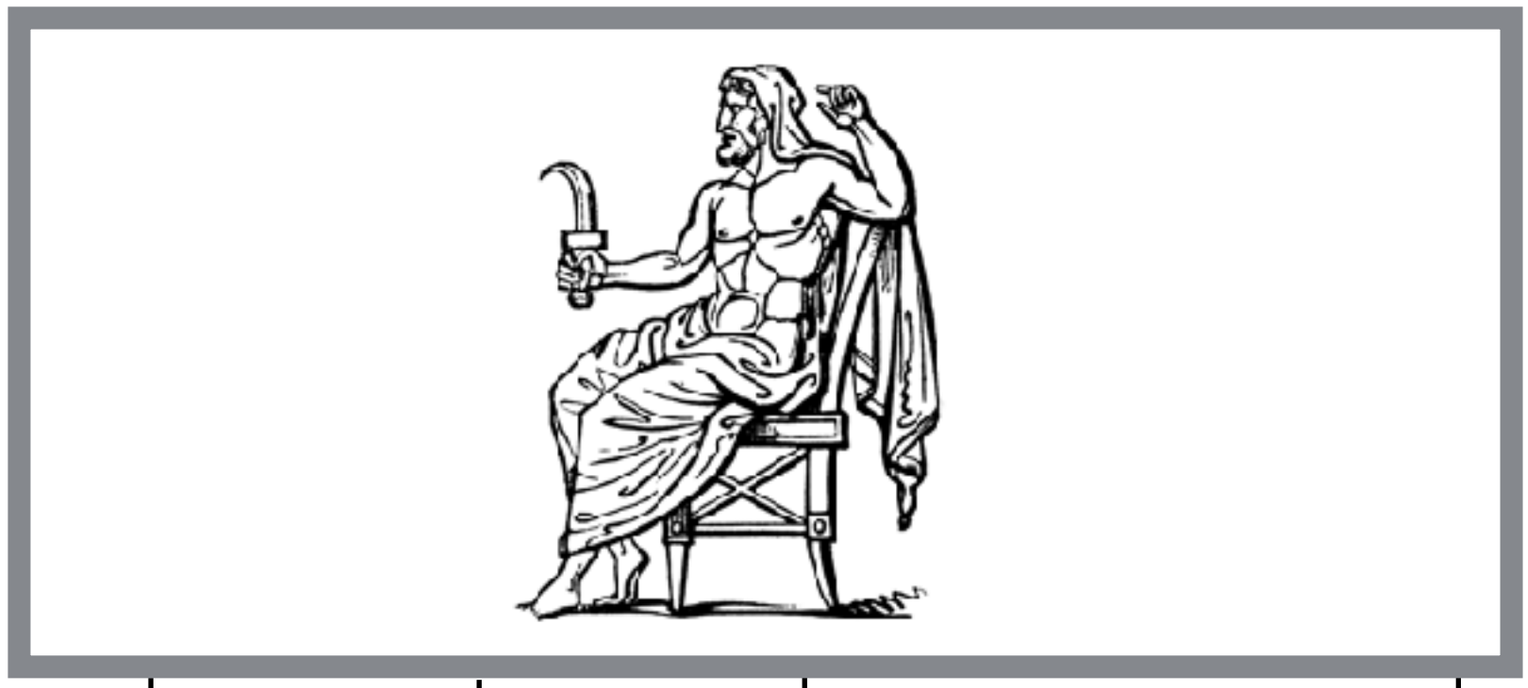
Client 1 — put(a_1) → 1 2 3 … N

data centers

Example: write request

data

labels



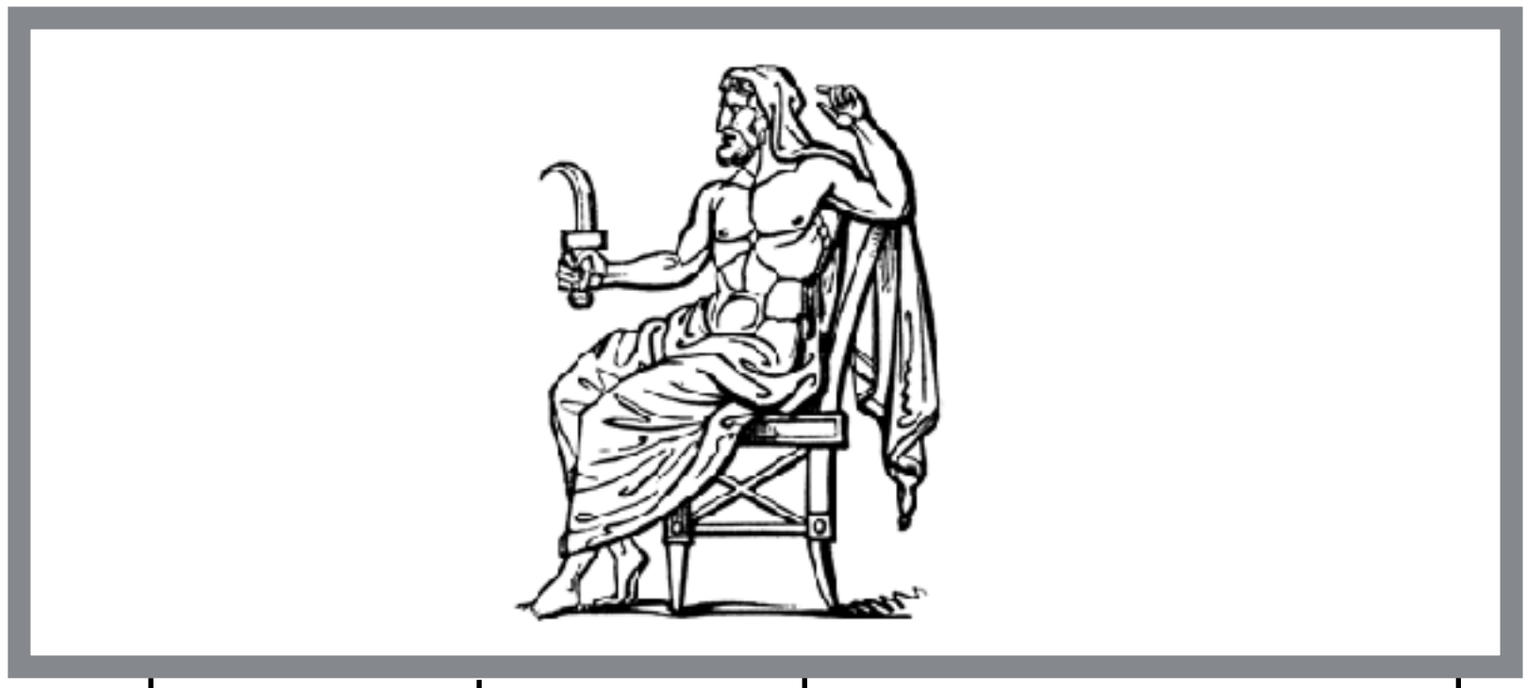
Client 1 — put(a_1) → 1 2 3 . . . N

data centers

Example: write request

data

labels



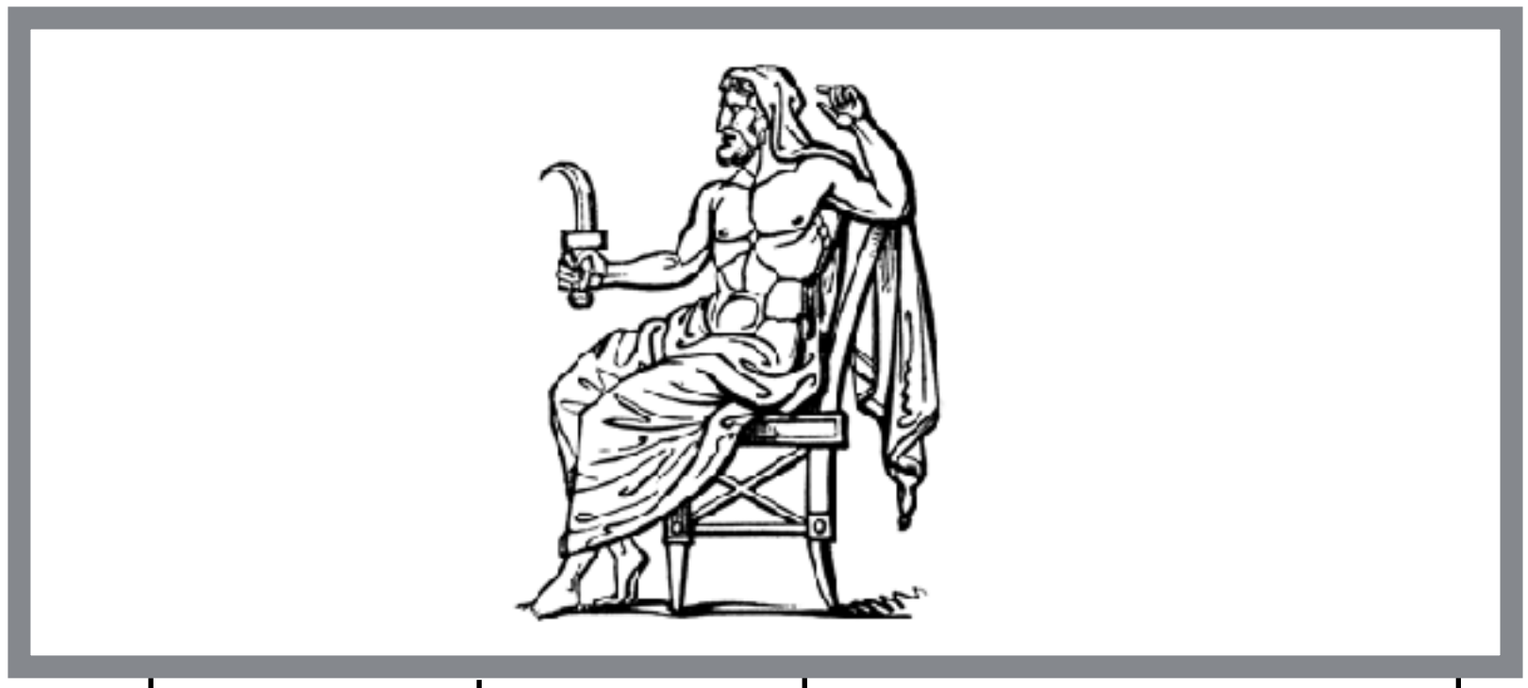
Client 1 — put(a_1) → 1 — 2 — 3 — … — N

data centers

Example: write request

data

labels



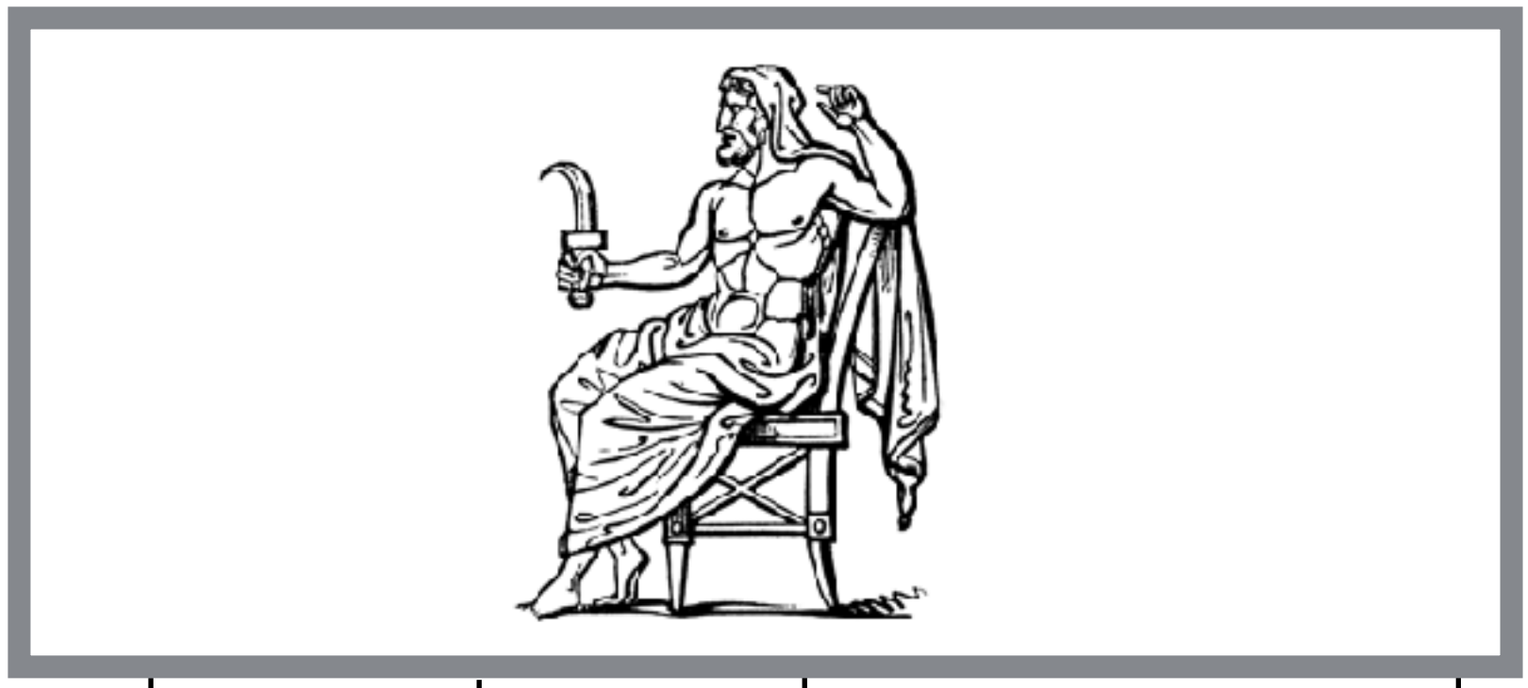
Client 1 — put(a_1) → 1 — 2 — 3 — … — N

data centers

Example: write request

data

labels



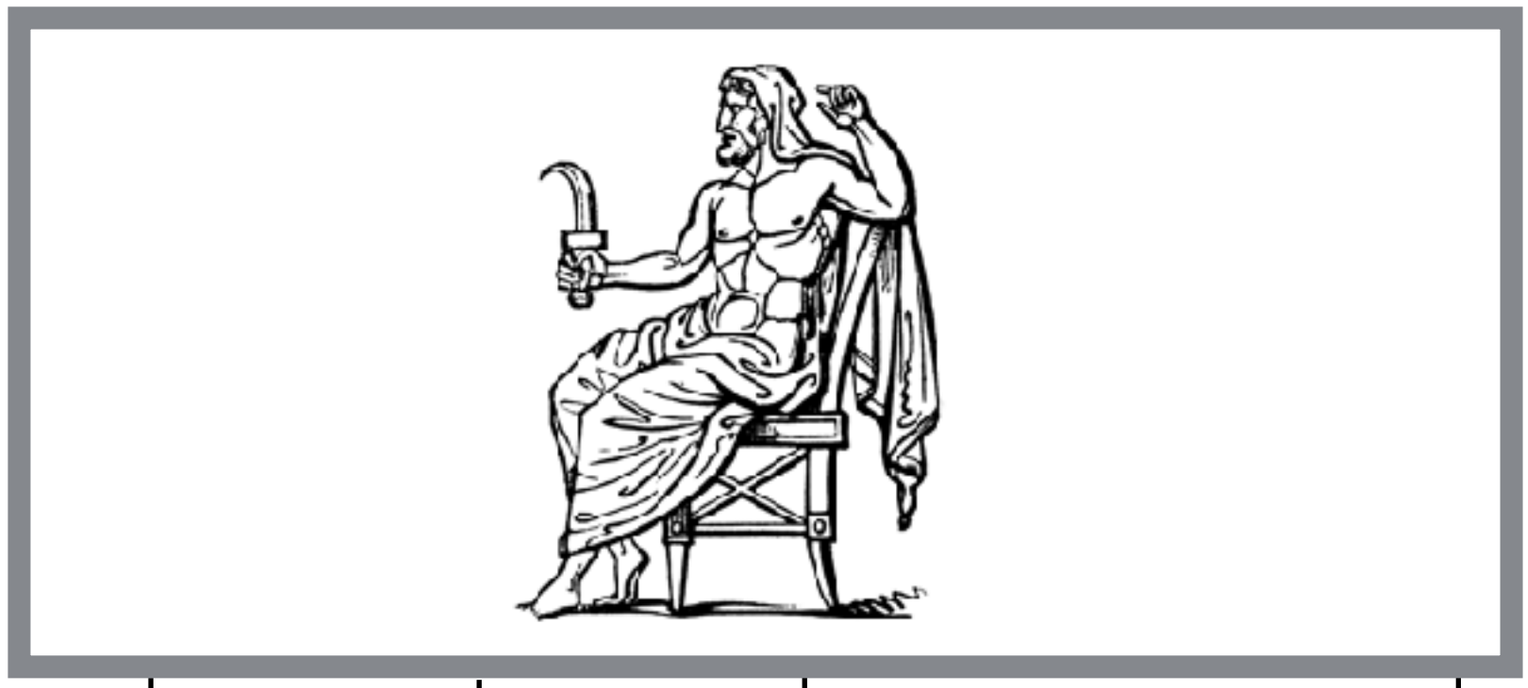
Client 1 — put(a_1) → 1 2 3 . . . N

data centers

Example: write request

data

labels



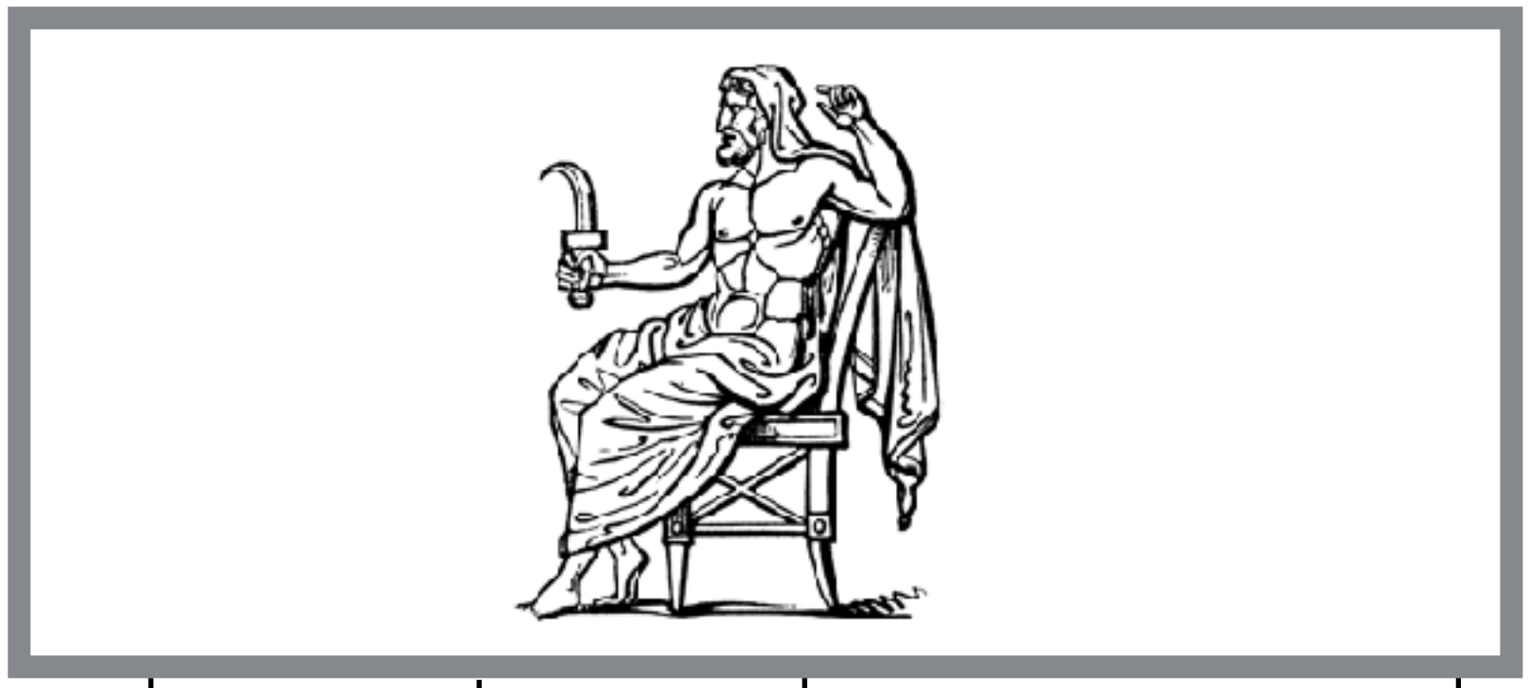
Client 1 — put(a_1) → 1 — 2 — 3 — … — N

data centers

Example: write request

data

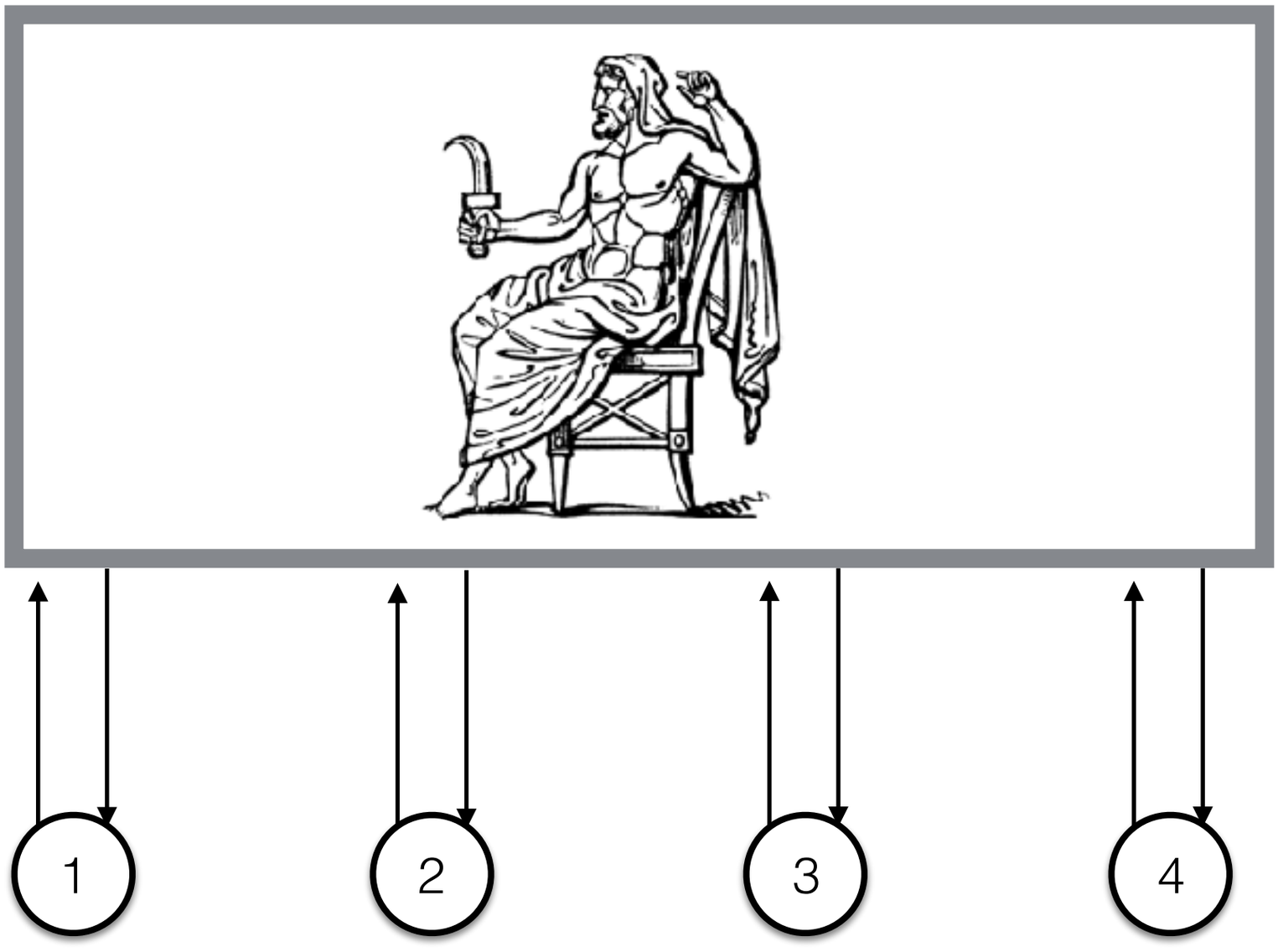
labels



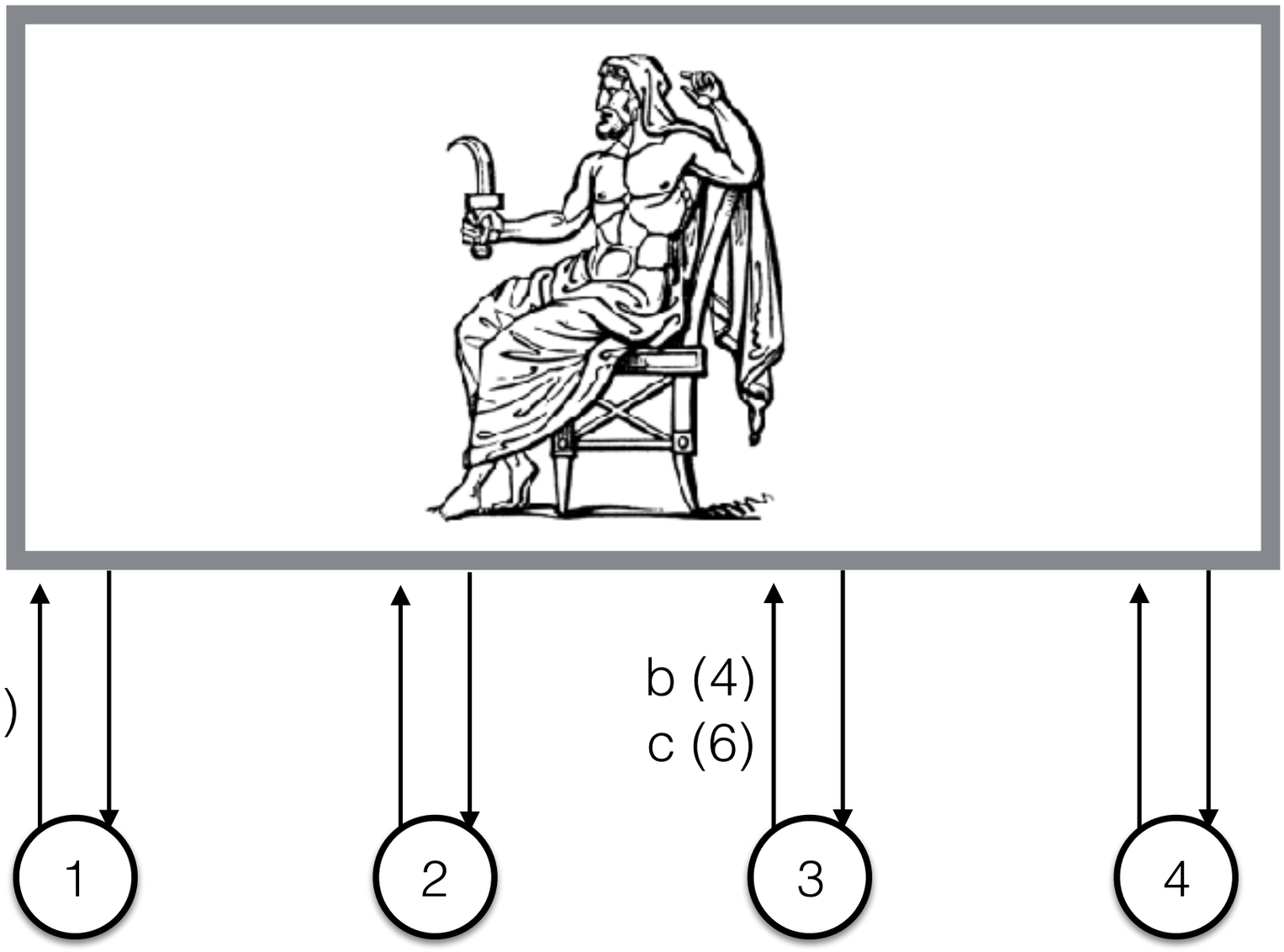
Client 1 — put(a_1) → 1 2 3 . . . N

data centers

Metadata propagation



Metadata propagation

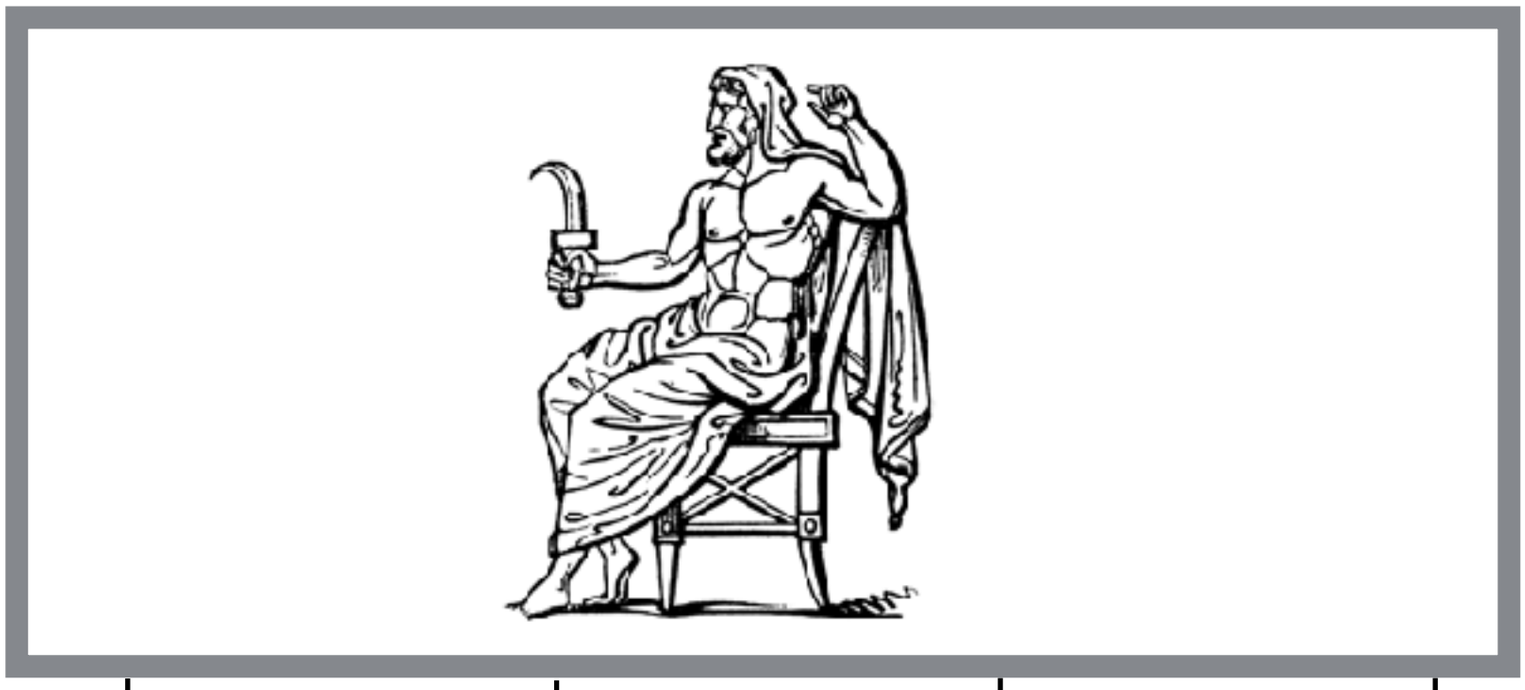


Metadata propagation

a → b, a → c

c causally depends on b

a concurrent to both b and c

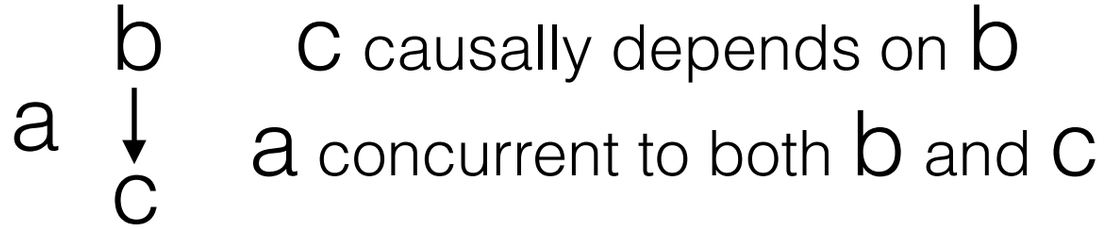


a (2)

b (4)
c (6)

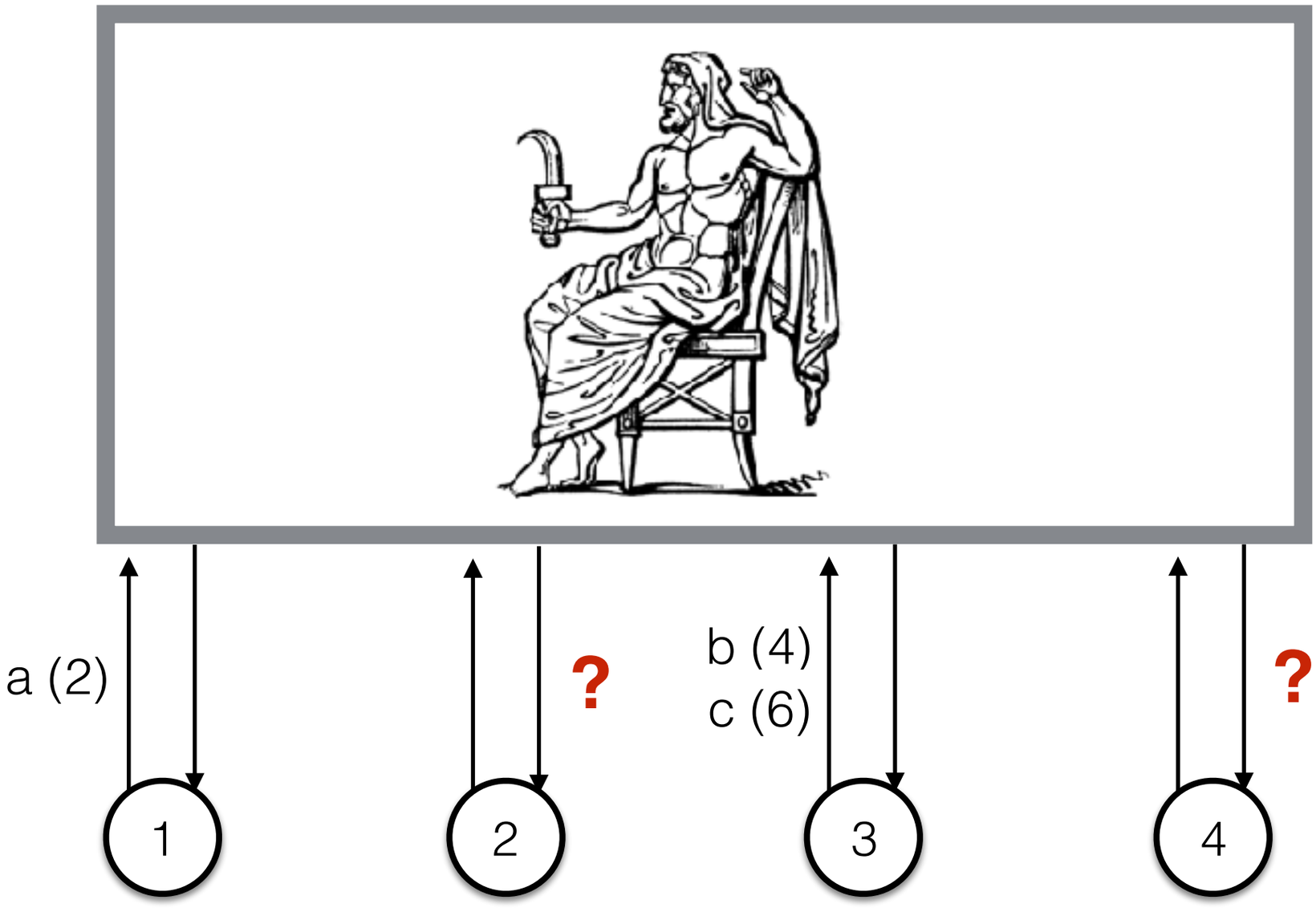
1 2 3 4

Metadata propagation



c causally depends on b

a concurrent to both b and c

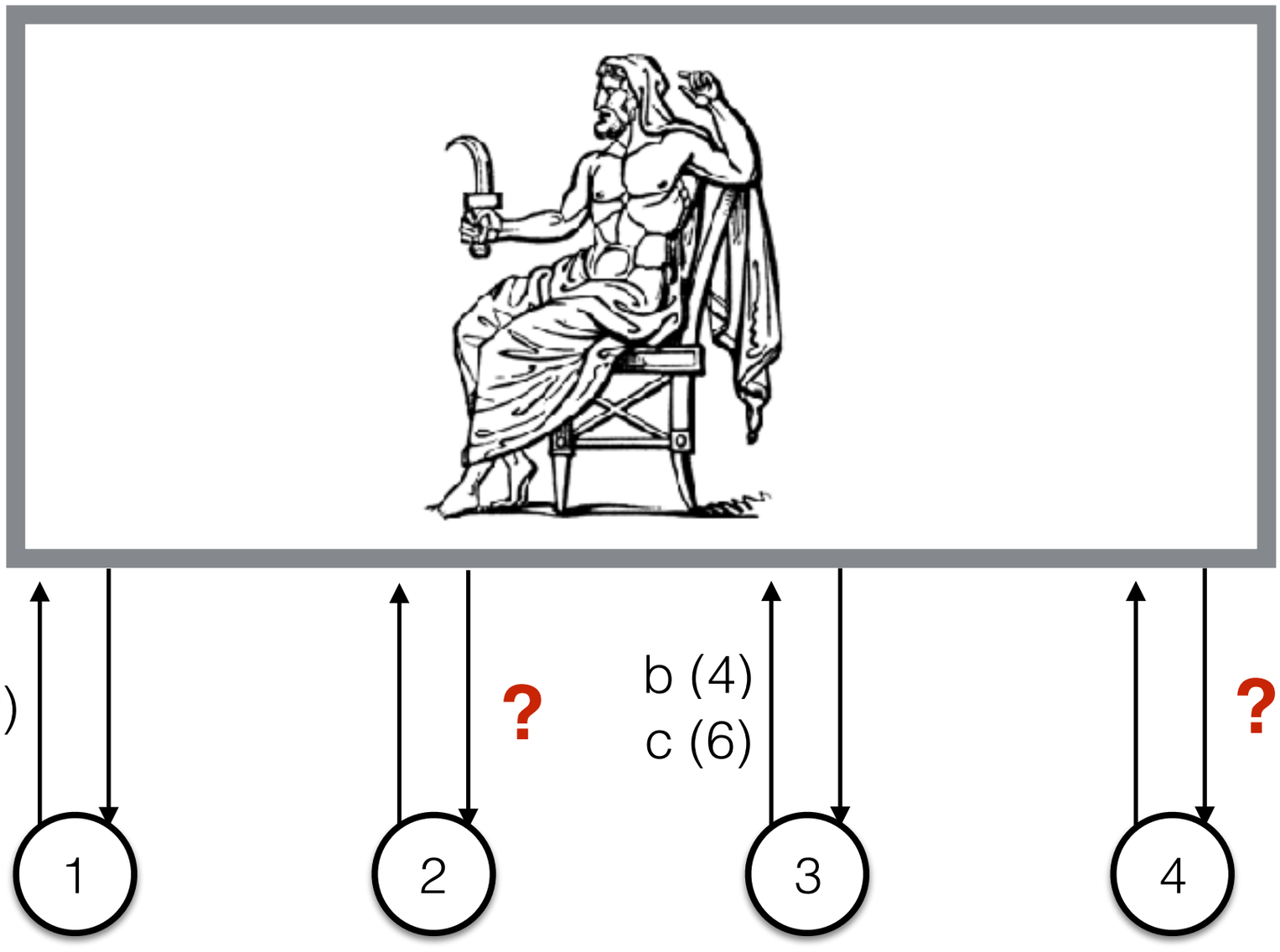


Metadata propagation

a → b, a → c

c causally depends on b

a concurrent to both b and c



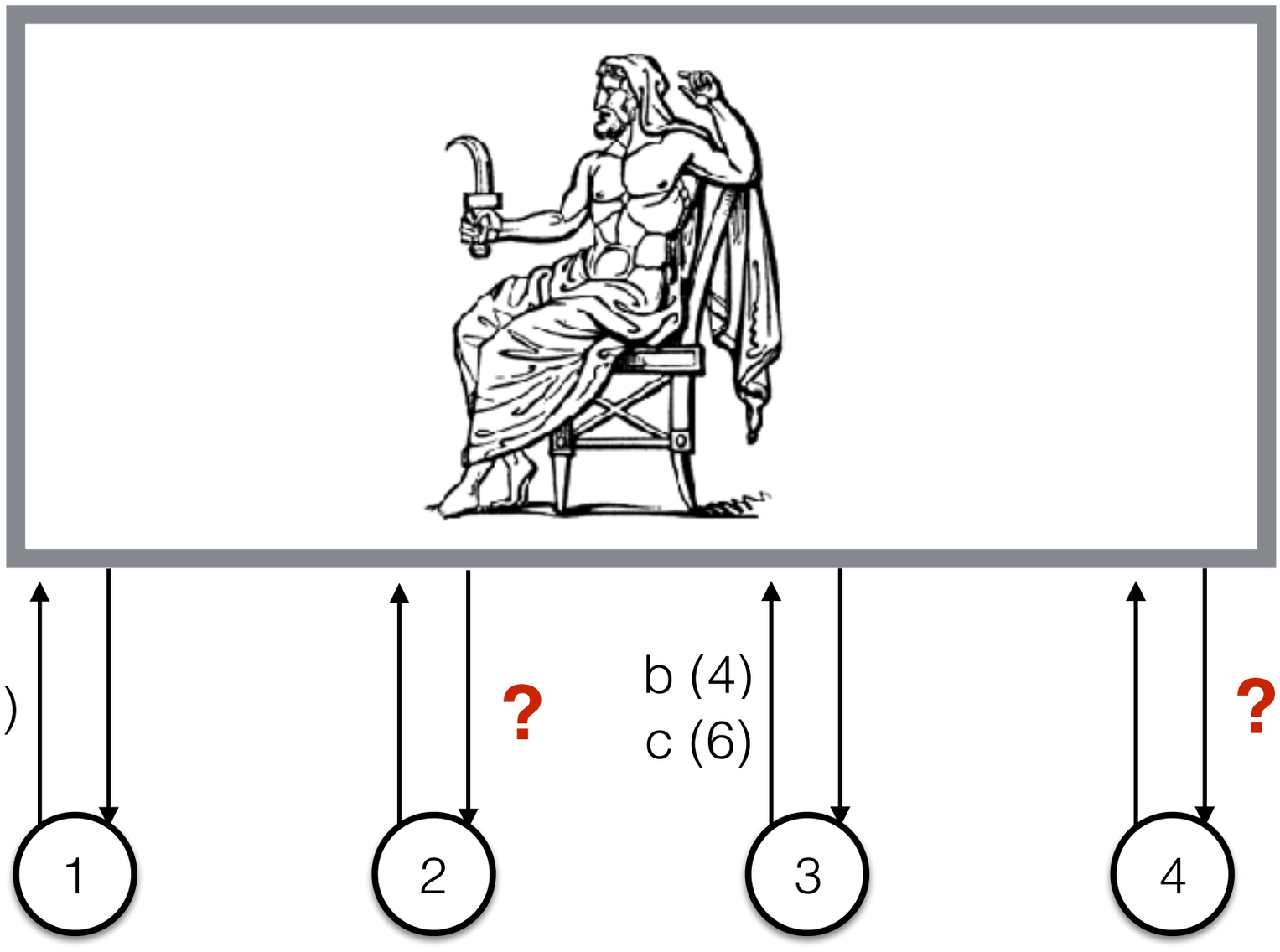
a b c ?

Metadata propagation

a → b, a → c

c causally depends on b

a concurrent to both b and c



a b c ?

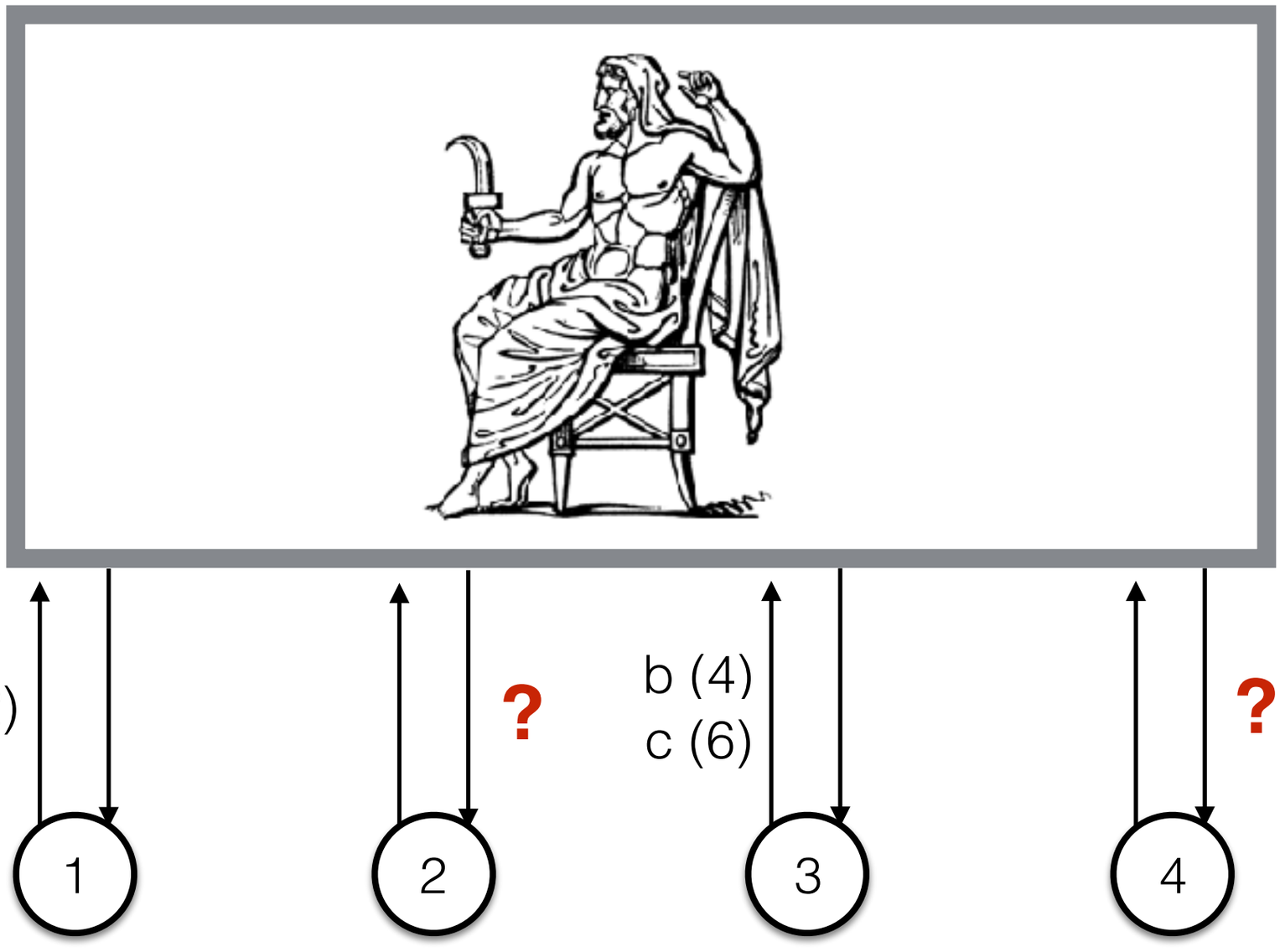
b a c ?

Metadata propagation

a → b, a → c

c causally depends on b

a concurrent to both b and c

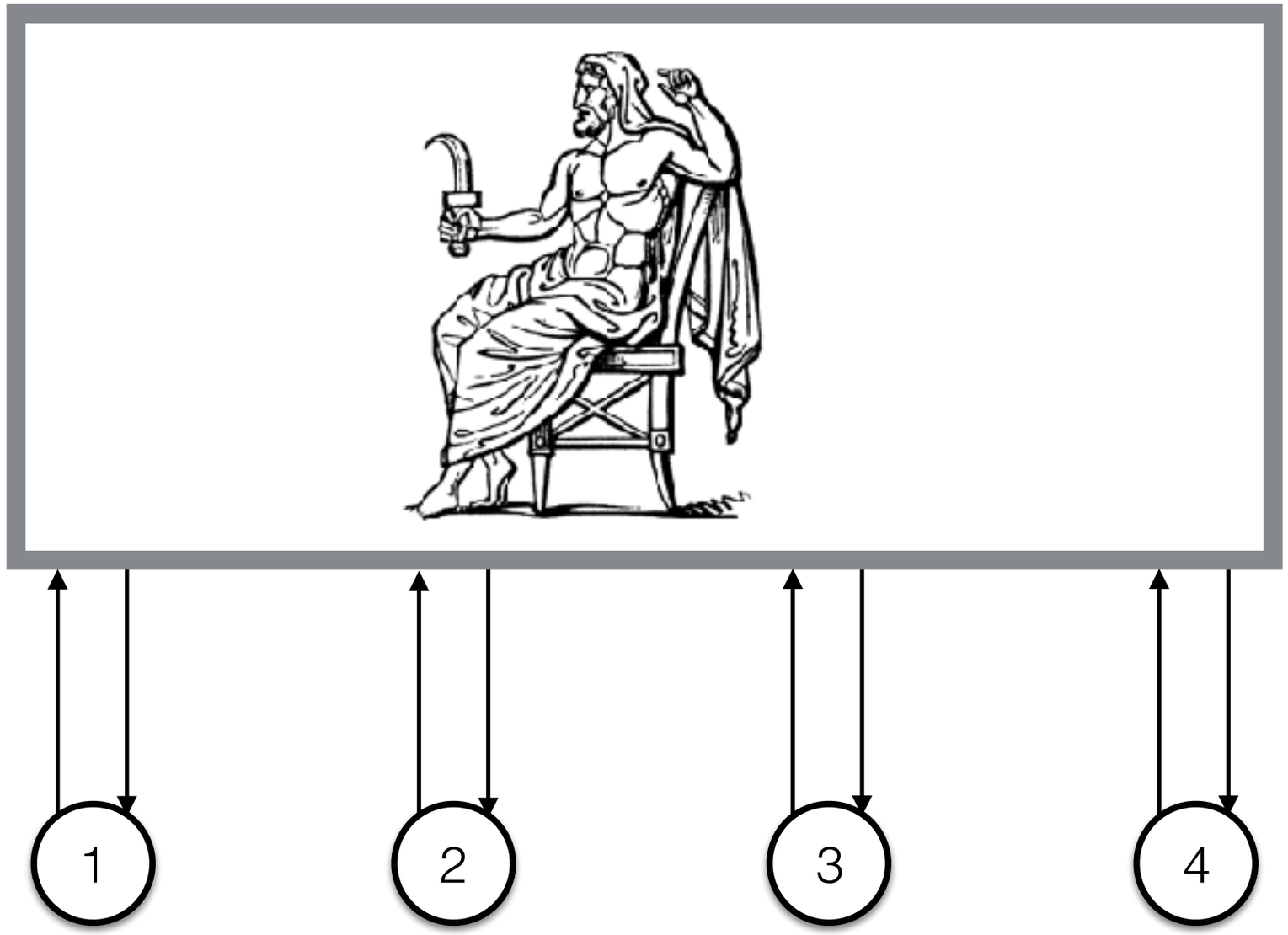


a b c ?

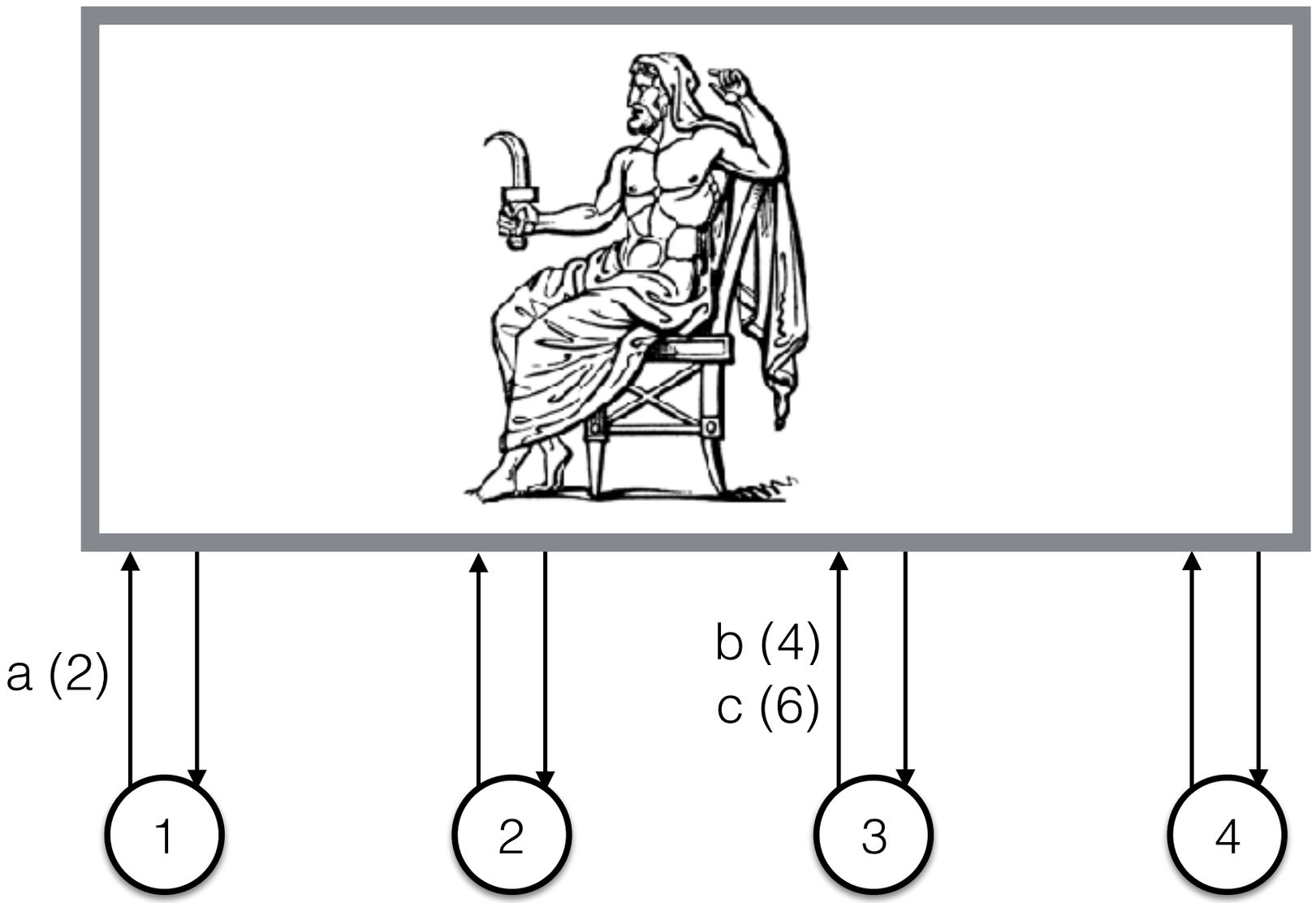
b a c ?

b c a ?

Metadata propagation



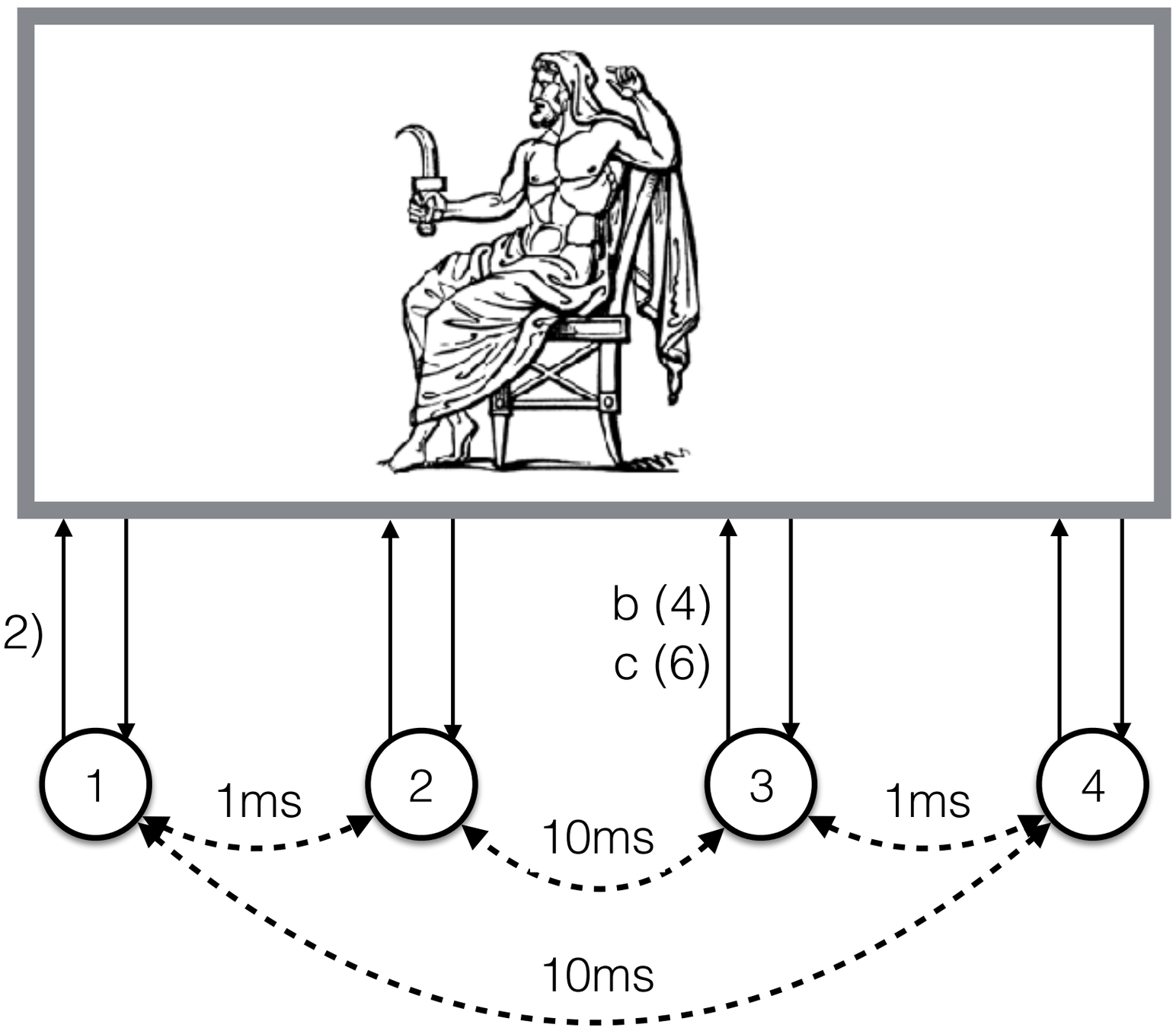
Metadata propagation



Metadata propagation

c causally depends on b

a concurrent to both b and c

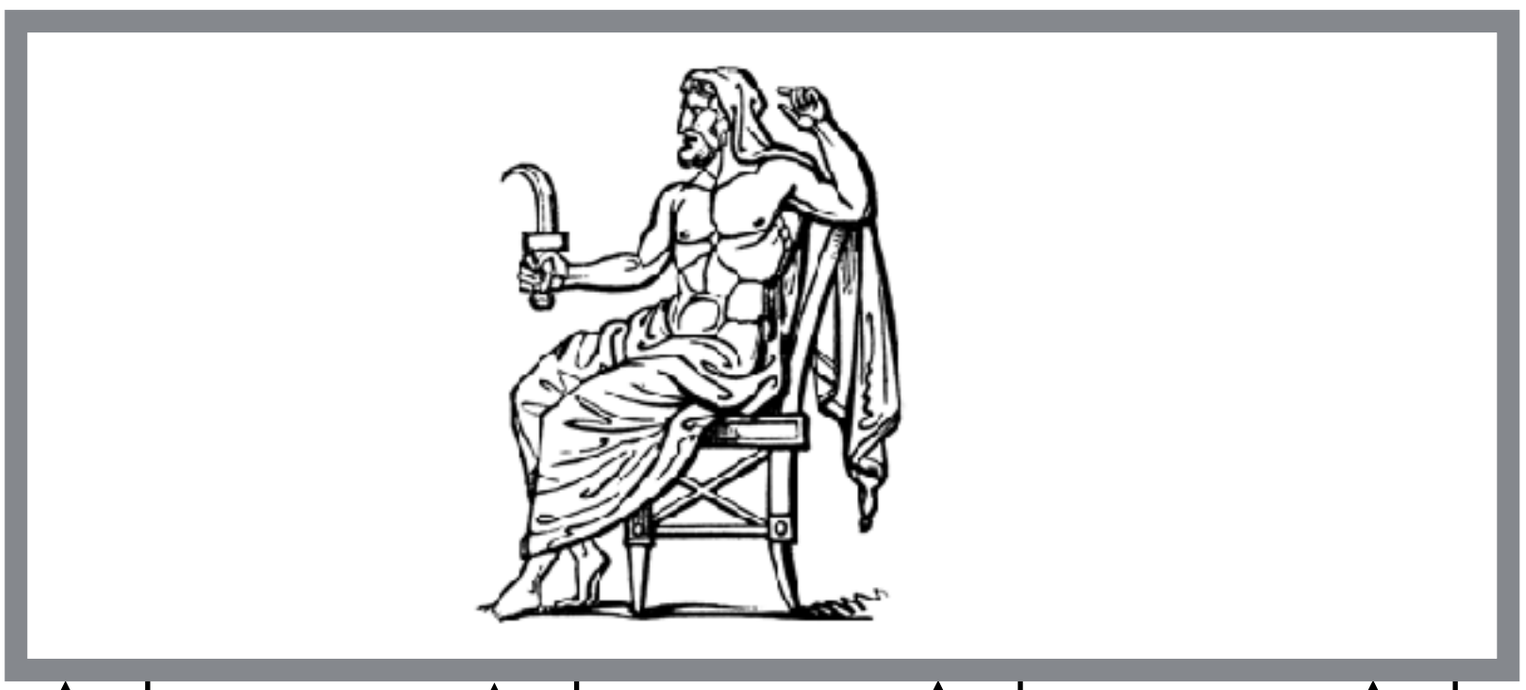


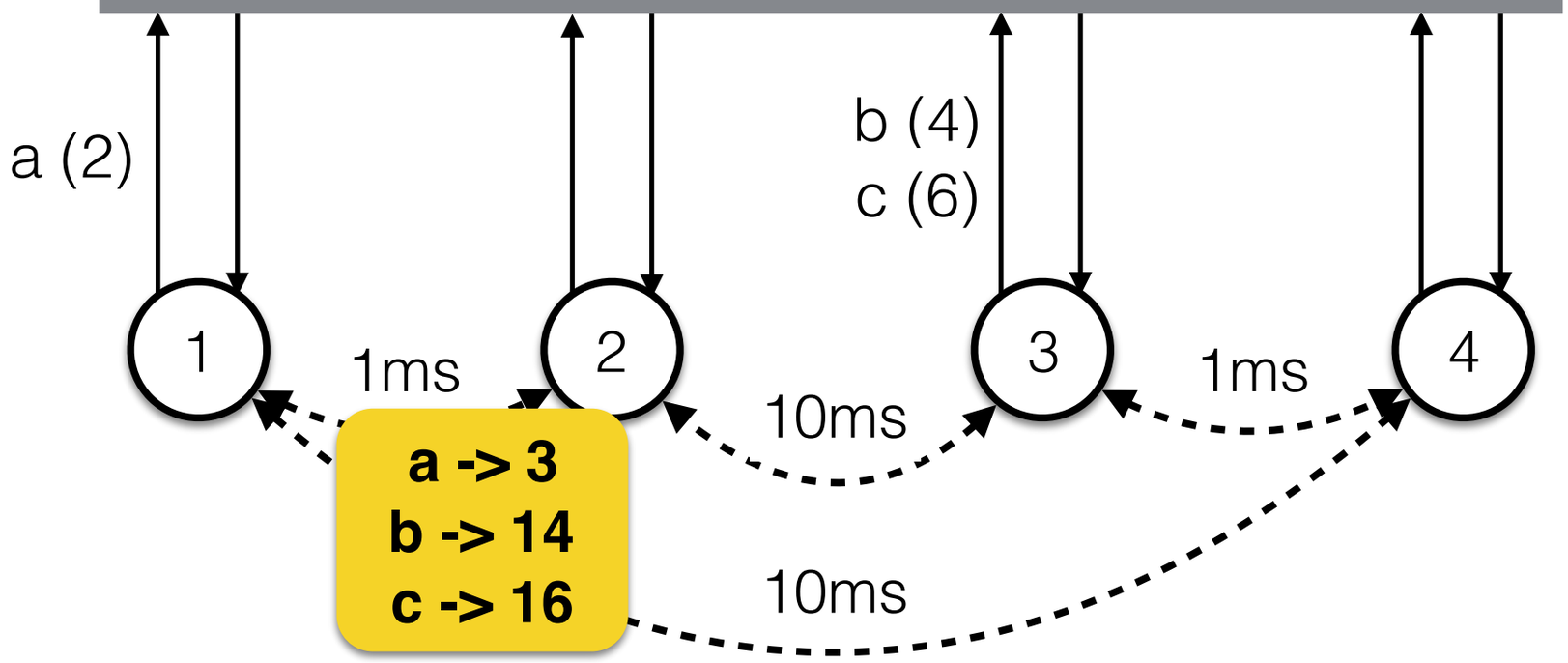
Metadata propagation

a → b, a → c

c causally depends on b

a concurrent to both b and c

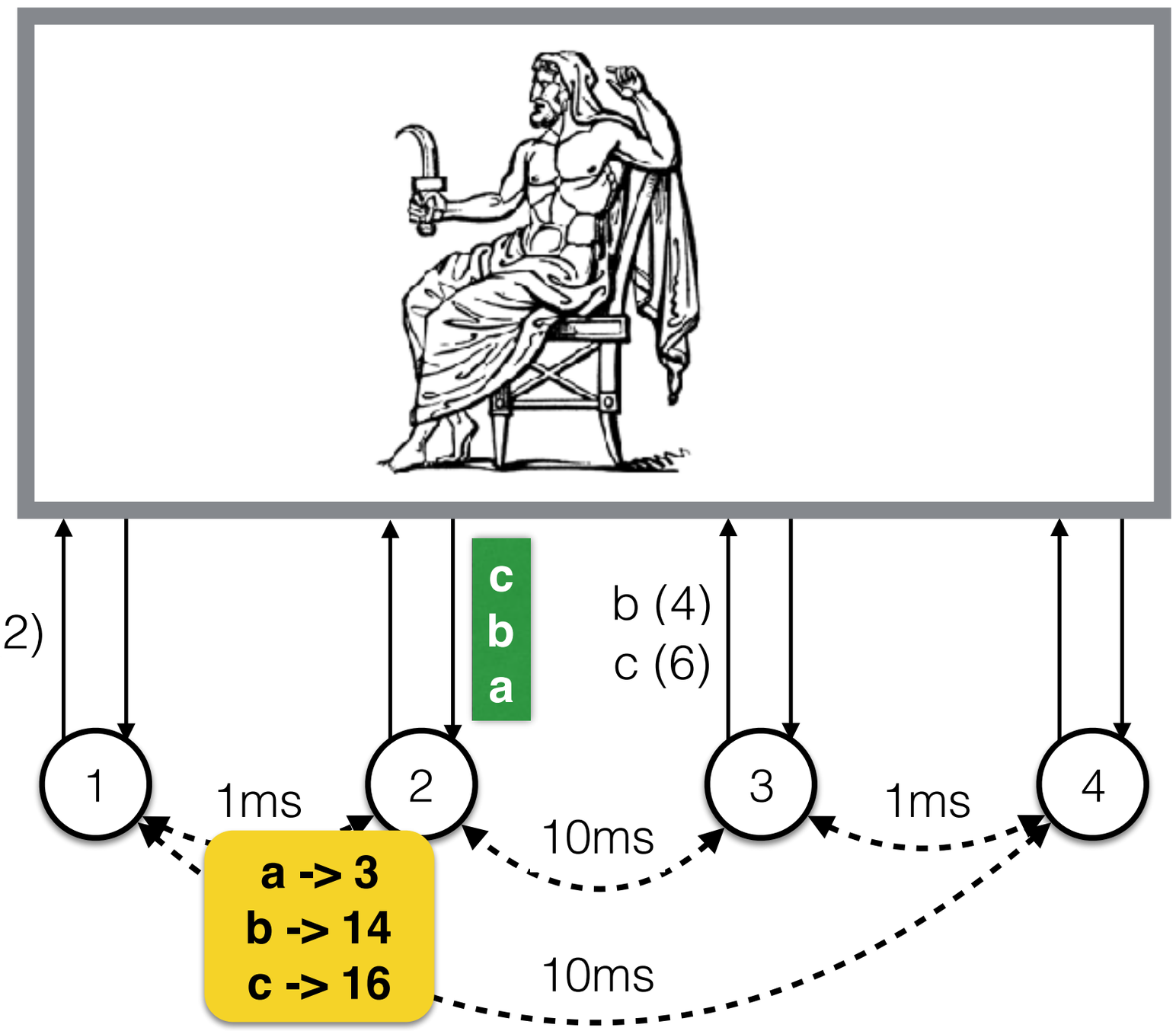




Metadata propagation

c causally depends on b

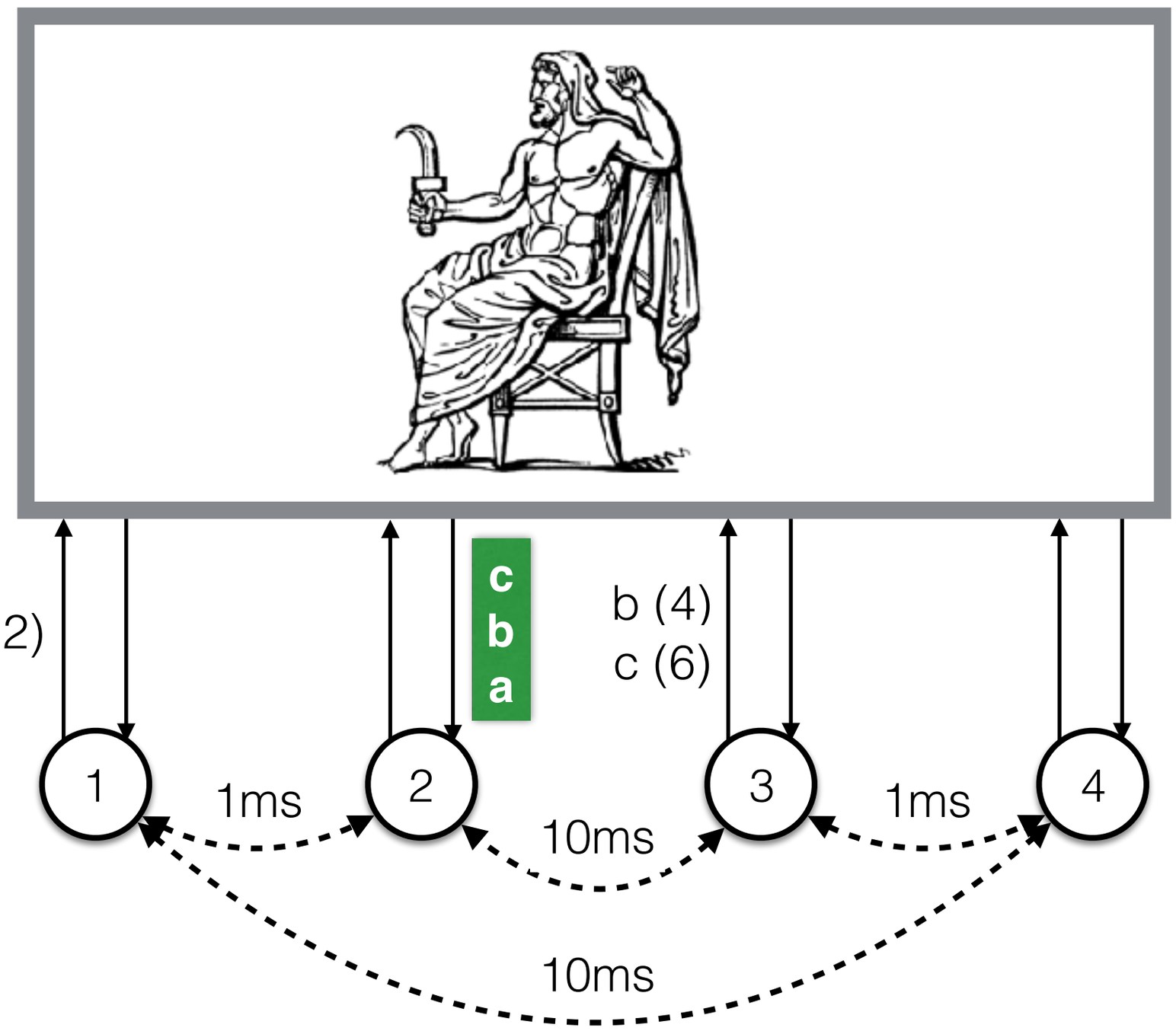
a concurrent to both b and c



Metadata propagation

c causally depends on b

a concurrent to both b and c



Metadata propagation

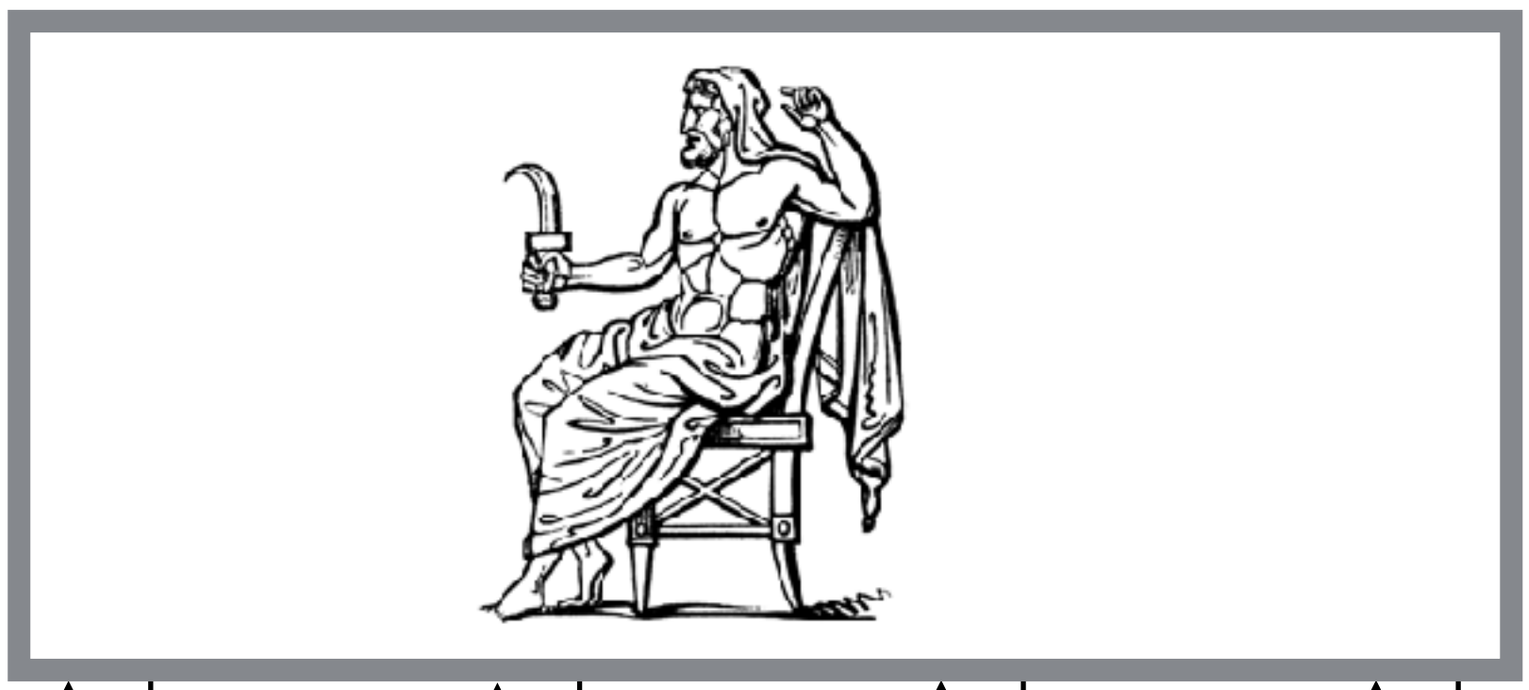
c causally depends on b

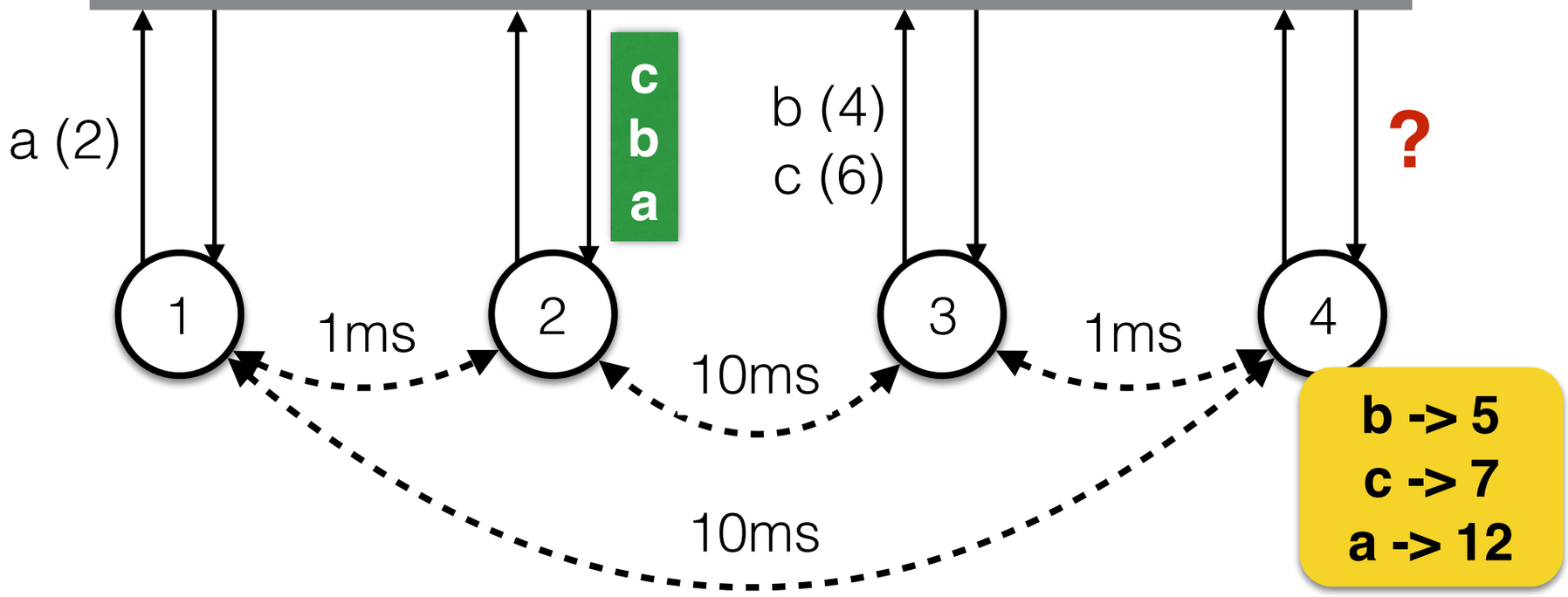
a concurrent to both b and c

Metadata propagation

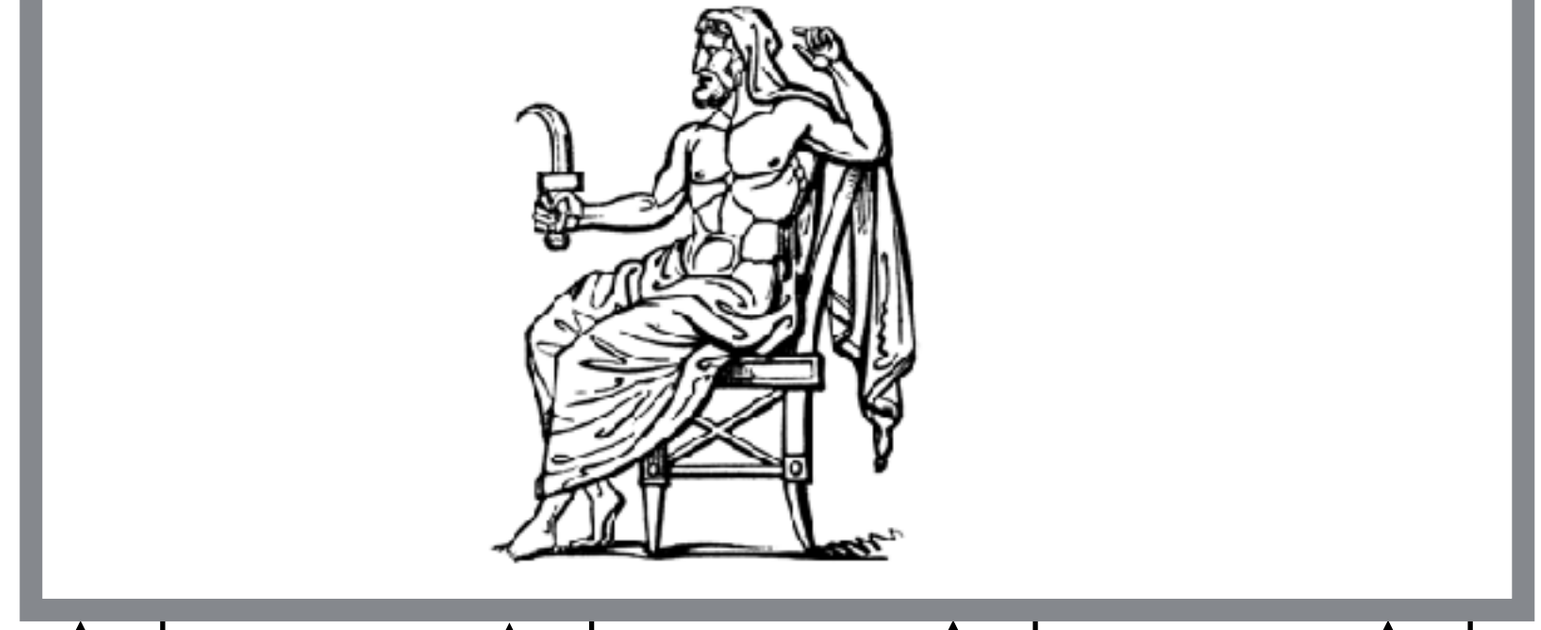
c causally depends on b

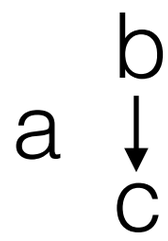
a concurrent to both b and c





Metadata propagation

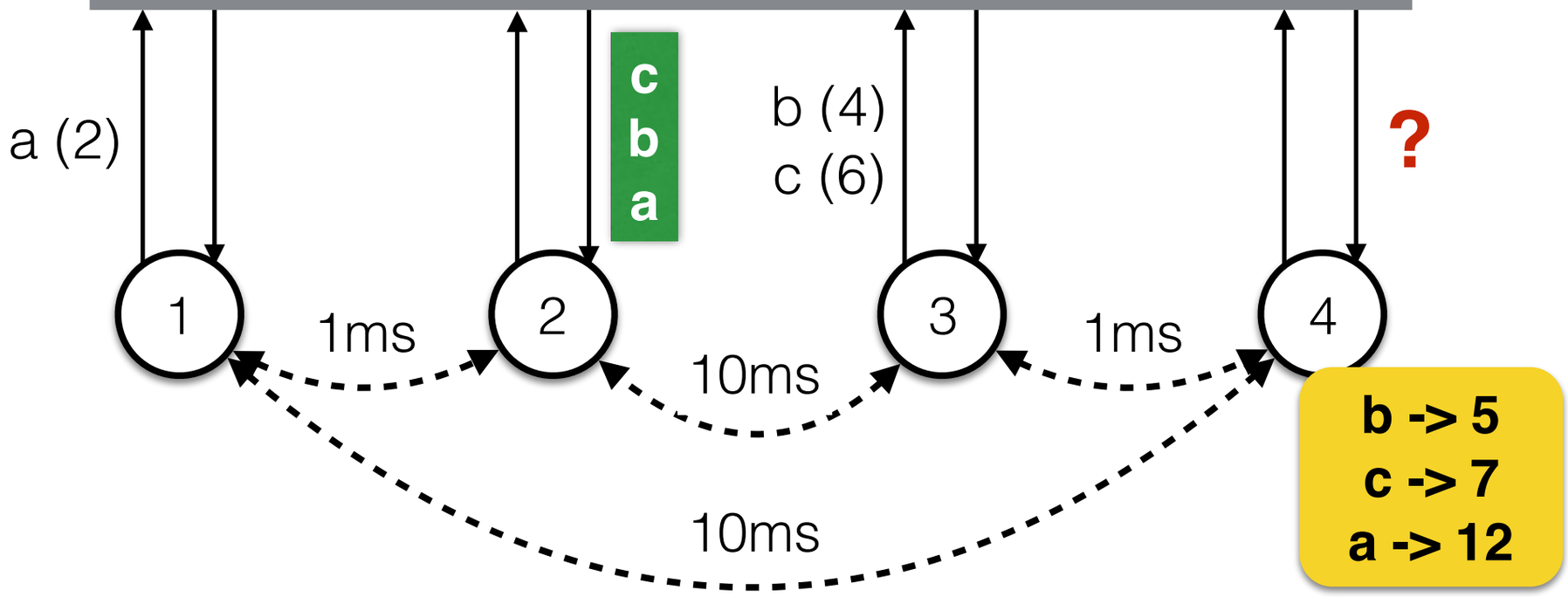




c causally depends on b

a concurrent to both b and c

c
b
a



Metadata propagation

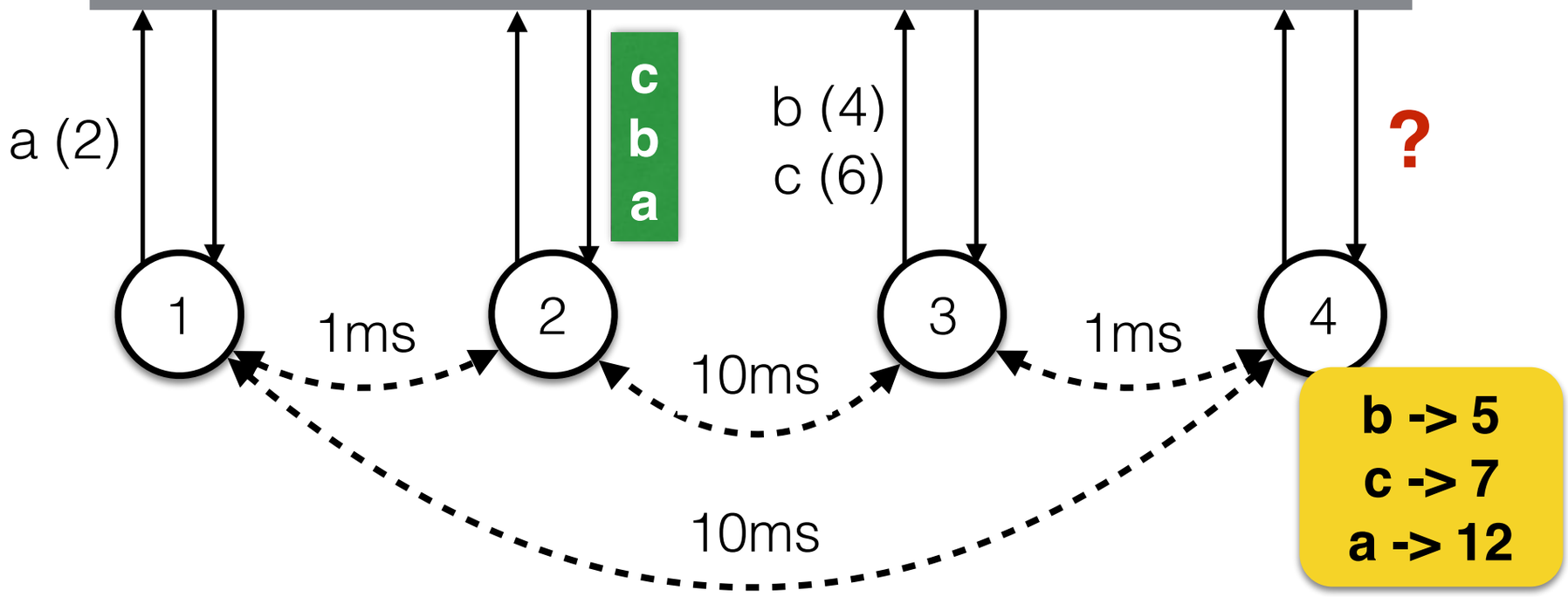
c causally depends on b

a concurrent to both b and c

delayed

c
b

a



Metadata propagation

a → b, a → c

c causally depends on b

a concurrent to both b and c

delayed

c
b
a

a -> 12
b -> 12
c -> 12

a (2)

c
b
a

b (4)
c (6)

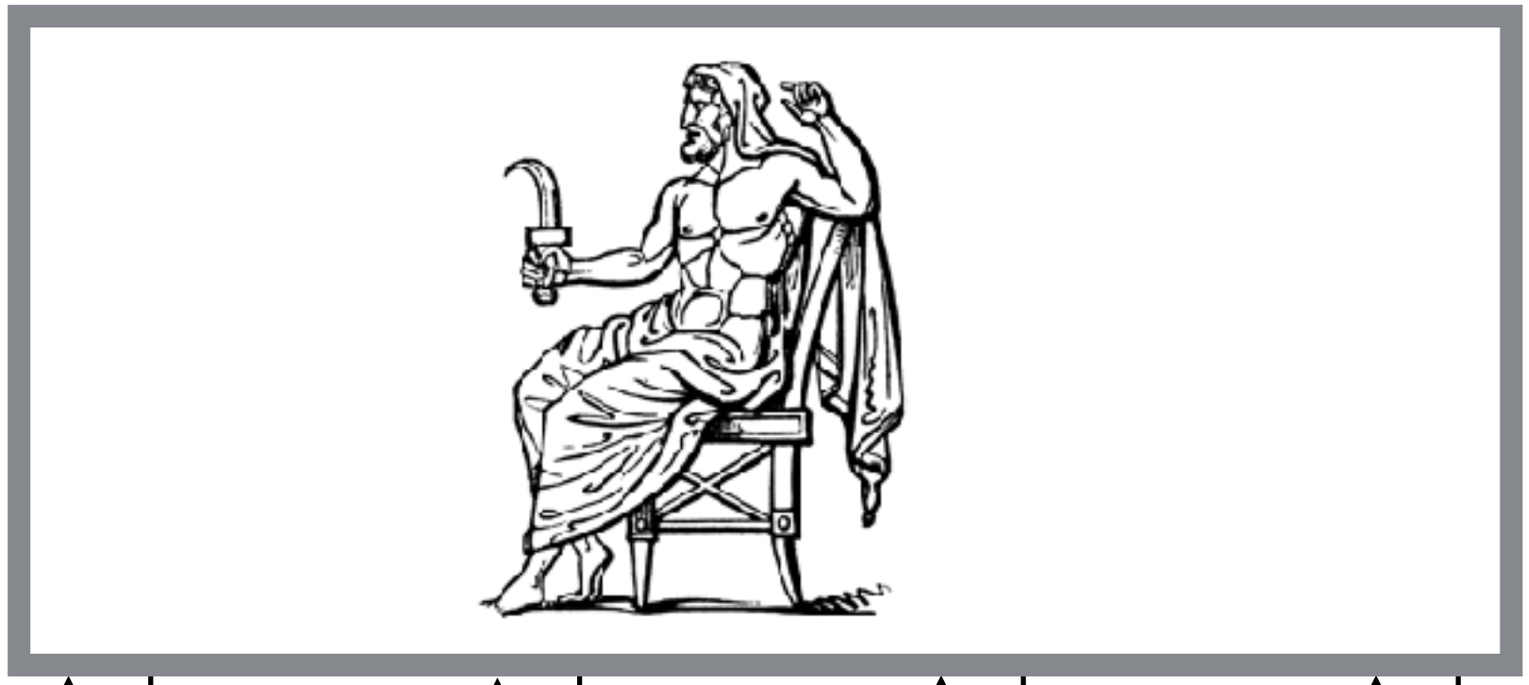
?

1 — 1ms — 2 — 10ms — 3 — 1ms — 4

10ms

b -> 5
c -> 7
a -> 12

Metadata propagation



a (2)

b (4)
c (6)

c
b
a

1ms

10ms

1ms

10ms

1

2

3

4

?

a
b
c

c causally depends on b

a concurrent to both b and c

delayed

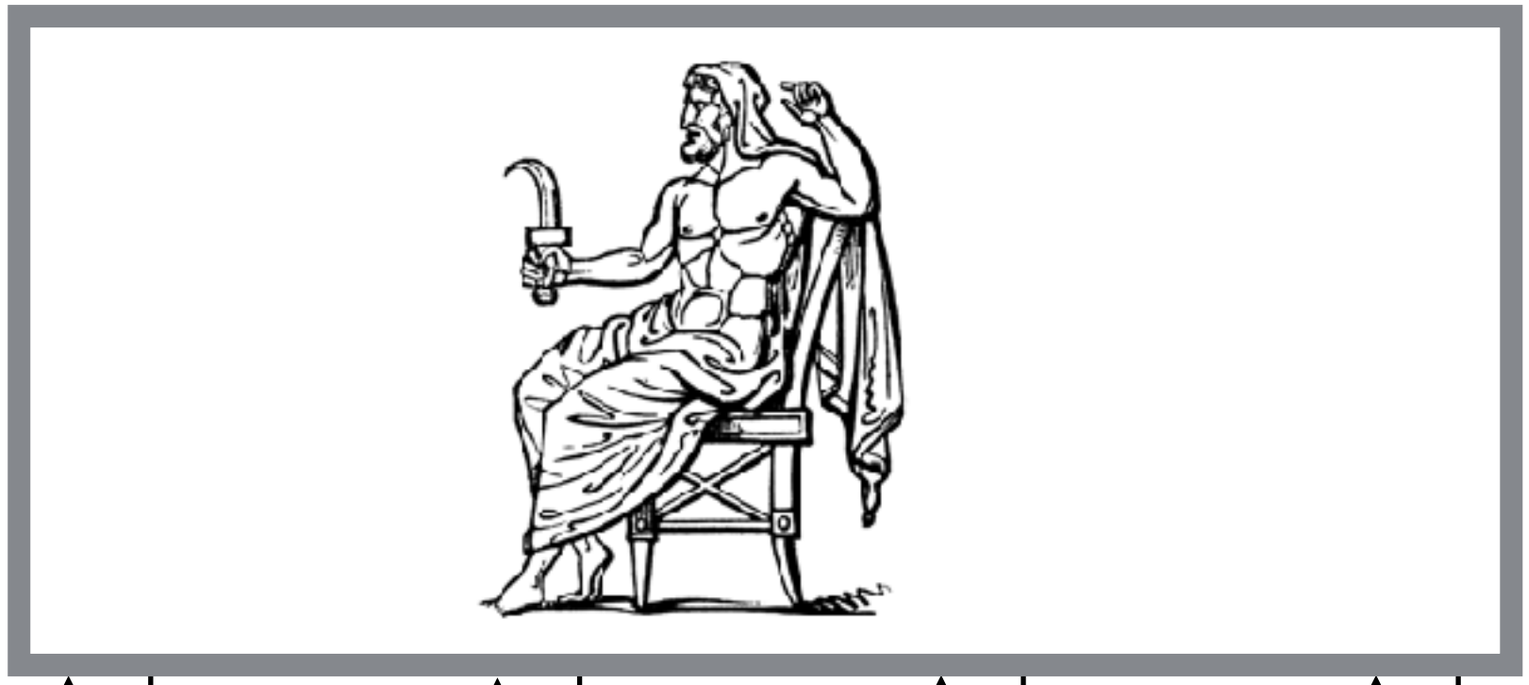
c
b
a

a -> 12
b -> 12
c -> 12

c
b
a

b -> 5
c -> 7
a -> 12

Metadata propagation



a (2)

b (4)
c (6)

c
b
a

1ms

10ms

1ms

10ms

1

2

3

4

delayed

c
b
a

a -> 12
b -> 12
c -> 12

delayed

c
a
b

b -> 5
c -> 7
a -> 12

?

b causally depends on a

c concurrent to both a and b

Metadata propagation

a → b, a → c

c causally depends on b

a concurrent to both b and c

delayed

c
b
a

a -> 12
b -> 12
c -> 12

b -> 5
a -> 12
c -> 12

delayed

c
a
b

c
b
a

a (2)

b (4)
c (6)

?

1

1ms

2

10ms

3

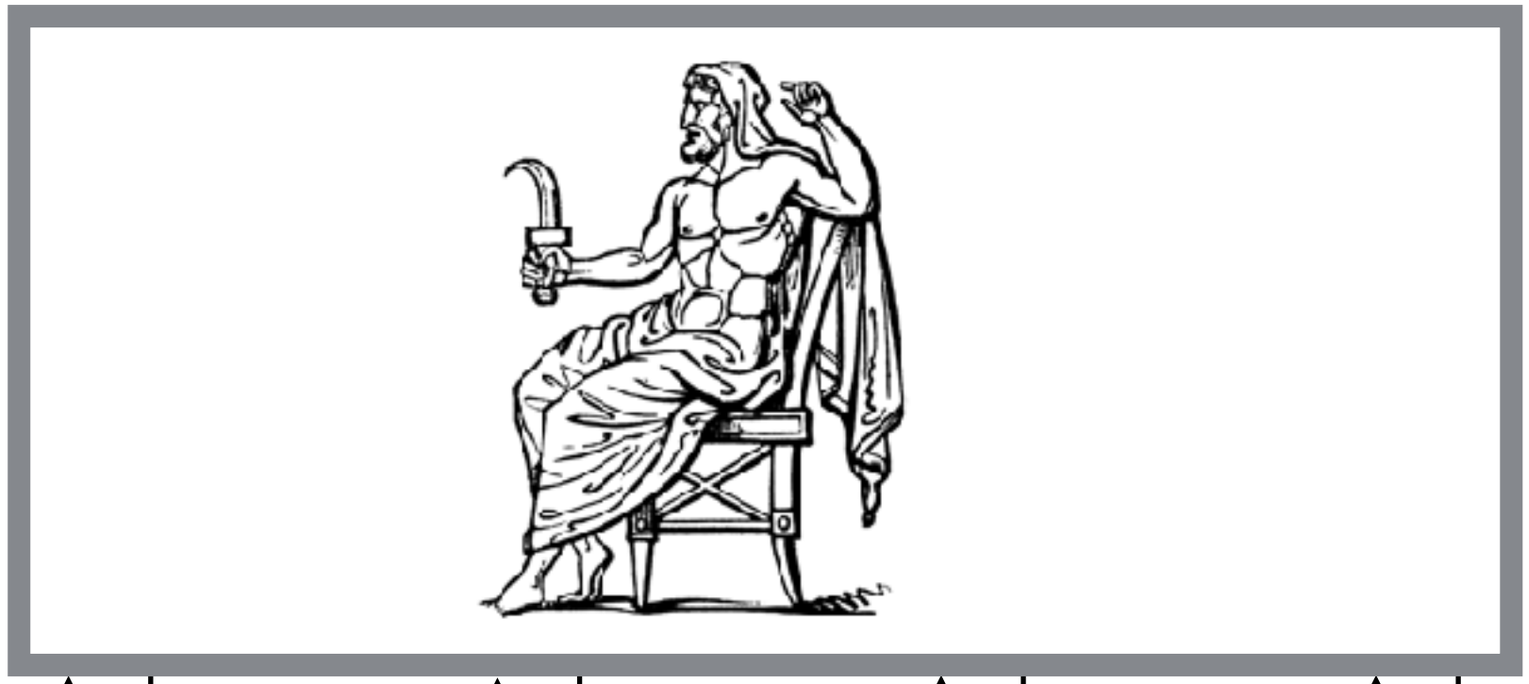
1ms

4

10ms

b -> 5
c -> 7
a -> 12

Metadata propagation



a (2)

b (4)
c (6)

?

1ms

10ms

10ms

1ms

1 2 3 4

c
b
a

b -> 5
a -> 12
c -> 12

b -> 5
c -> 7
a -> 12

a
c
b

a
b
c

a -> 12
b -> 12
c -> 12

delayed

c
b

a

delayed

c

a

b

a
c
b

a
b
c

a
b

c causally depends on b

a concurrent to both b and c

Metadata propagation

a → b, a → c

c causally depends on b

a concurrent to both b and c

a (2)

b (4)
c (6)

c
b
a

1ms

10ms

10ms

1

2

3

4

delayed

c
b

a

b -> 5
a -> 12
c -> 12

a
c
b

?

b -> 5
c -> 7
a -> 12

a -> 12
b -> 12
c -> 12

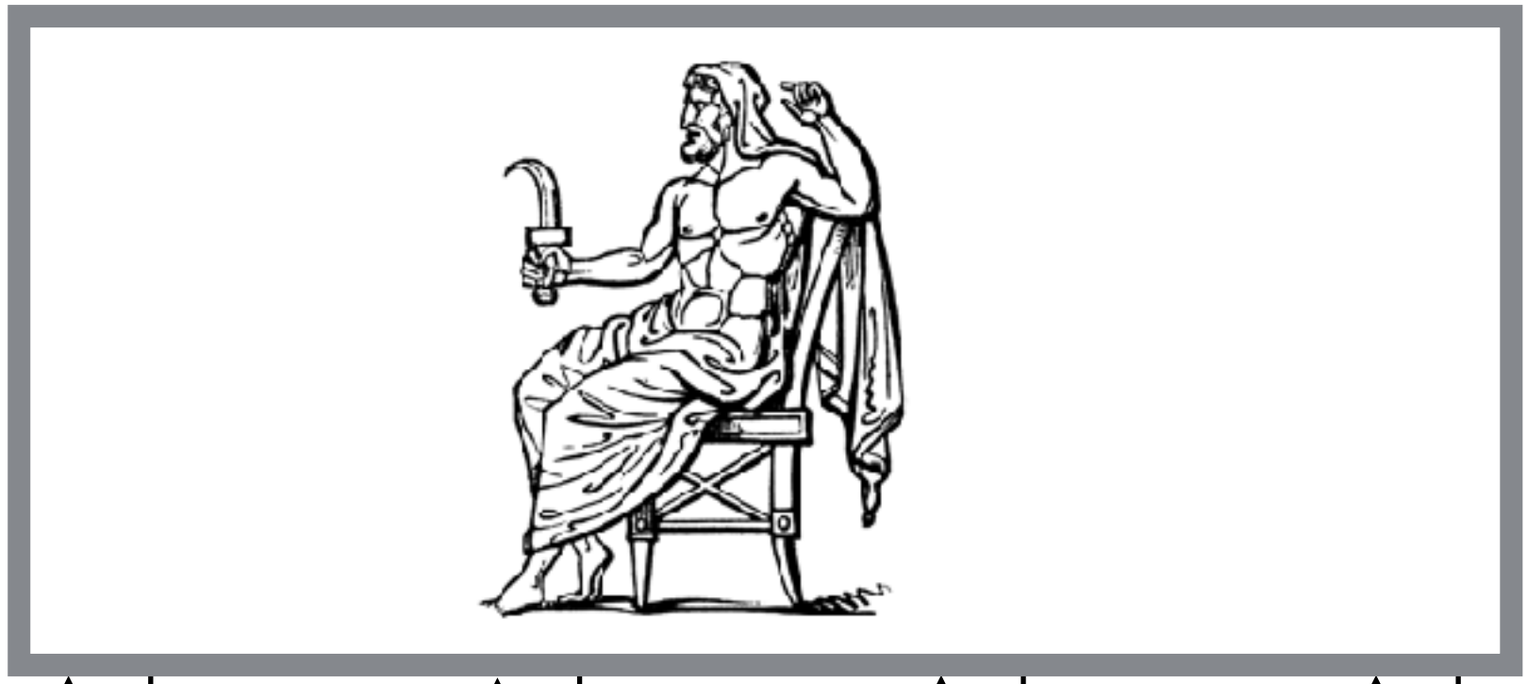
delayed

c

a
b

b -> 5
c -> 7
a -> 12

Metadata propagation



a (2)

b (4)
c (6)

1ms
10ms
10ms
1ms

1
2
3
4

c
b
a

c
b
a

a
c
b

a
c
b

b -> 5
a -> 12
c -> 12

b -> 5
c -> 7
a -> 12

a -> 12
b -> 12
c -> 12

b -> 5
c -> 7
a -> 12

a causally depends on b
a concurrent to both b and c

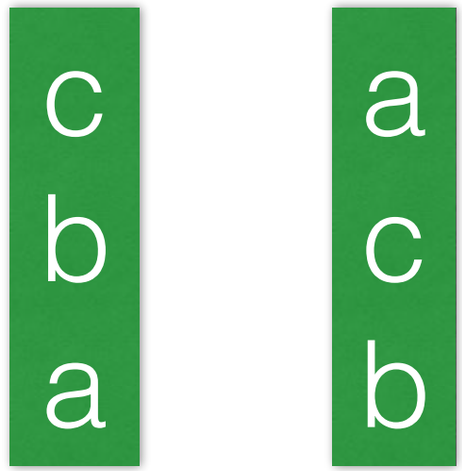
Metadata propagation

Causal consistency is a partial order

Saturn exploits this fact **by serving possibly different linear extensions** of the causal order to each data center

Each served serialization aims at optimising data freshness, and thus, reducing the impact of false dependencies

Metadata propagation



Causal consistency is a partial order

Saturn exploits this fact **by serving possibly different linear extensions** of the causal order to each data center

Each served serialization aims at optimising data freshness, and thus, reducing the impact of false dependencies

Metadata propagation: architecture

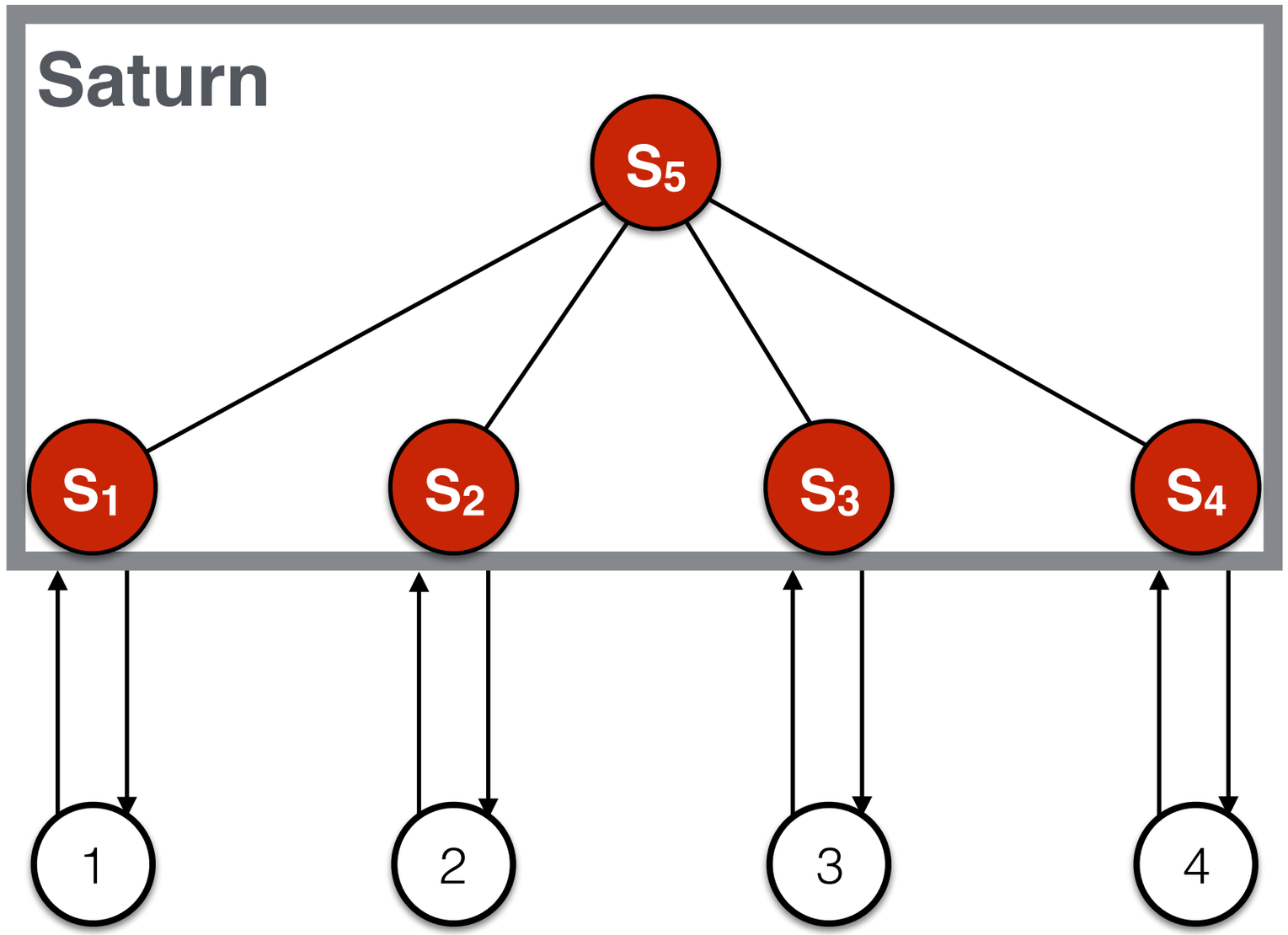
Saturn leverages a set of cooperating servers, namely **serializers**, forming a tree

Causal consistency is trivially enforced by the tree assuming

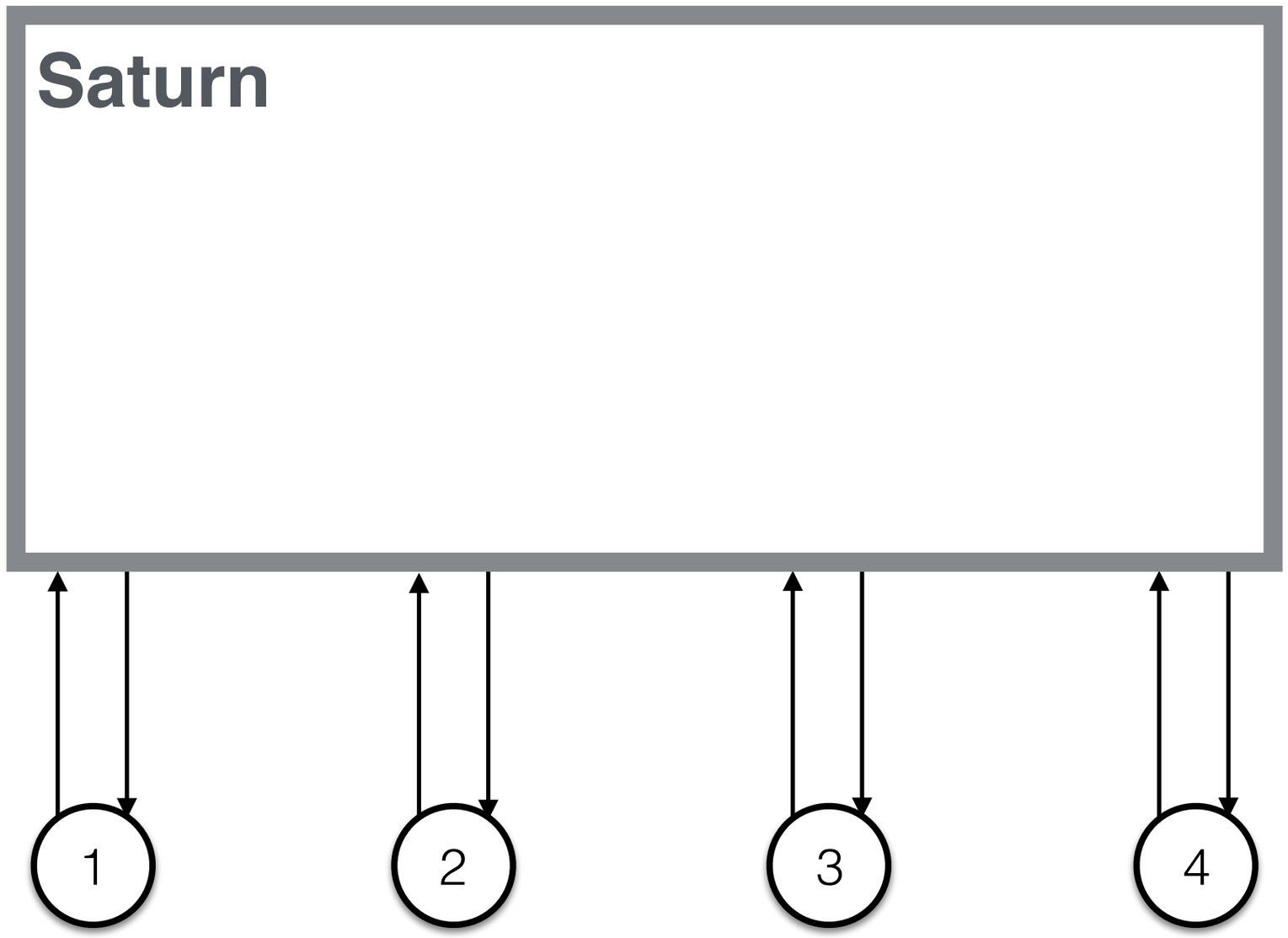
- FIFO links among serializers
- Serializers propagate metadata preserving the observed order

Serializers are geographically distributed to optimize data freshness

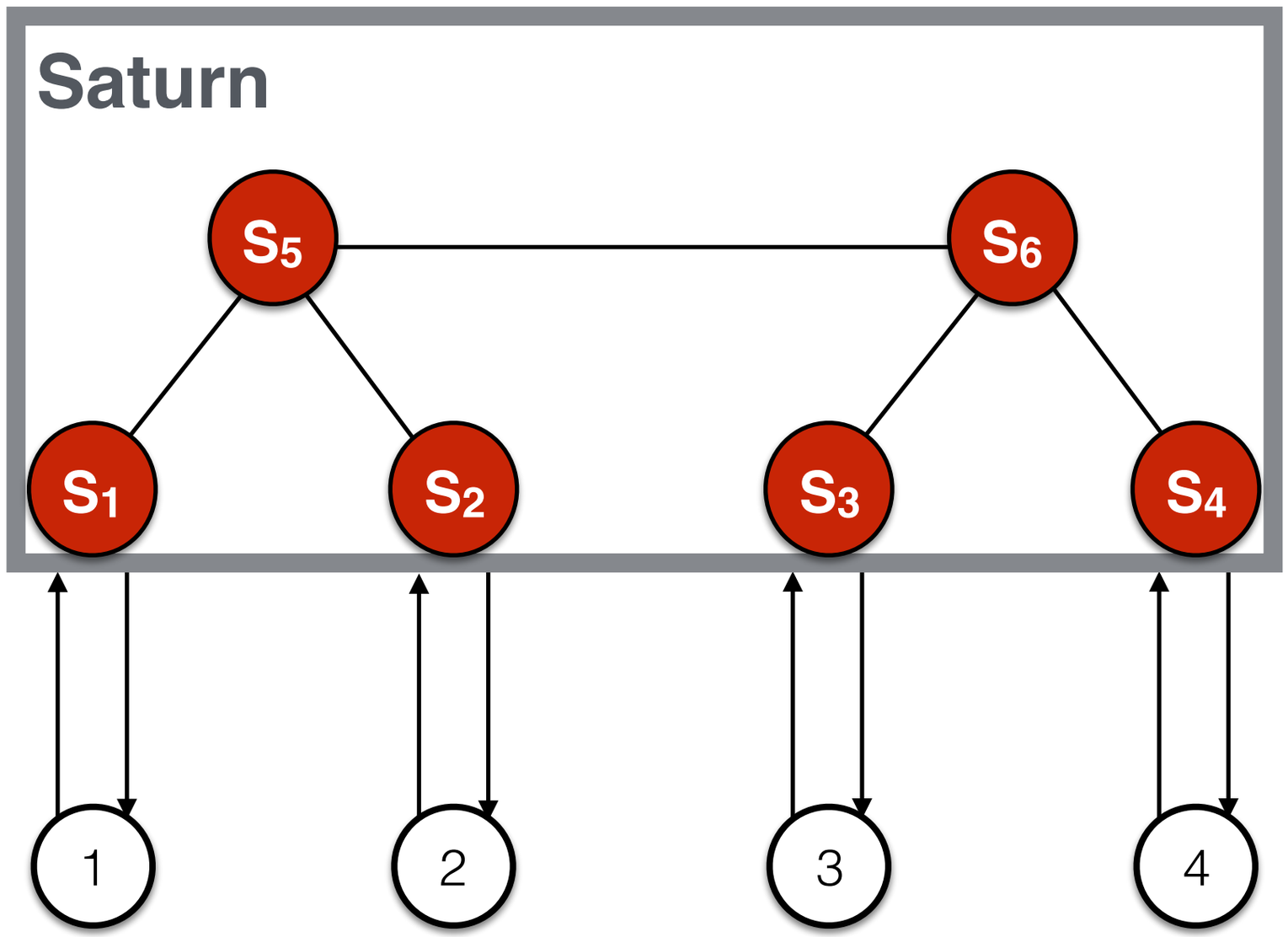
Metadata dissemination graph



Metadata dissemination graph



Metadata dissemination graph



Optimal dissemination graph

The goal is to build the tree such that metadata-paths latencies (through the tree) **match** data-paths

Weighted Minimal Mismatch

$$mismatch_{i,j} = |\Delta^M(i,j) - \Delta(i,j)|$$

$$min \sum_{\forall i,j \in V} c_{i,j} \cdot mismatch_{i,j}$$

Optimal dissemination graph

The goal is to build the tree such that metadata-paths latencies (through th data-paths

absolute difference between label-paths and data paths

Weighted Minimal Mismat

$$mismatch_{i,j} = |\Delta^M(i,j) - \Delta(i,j)|$$

$$min \sum_{\forall i,j \in V} c_{i,j} \cdot mismatch_{i,j}$$

Optimal dissemination graph

The goal is to build the tree such that metadata-paths latencies (through the tree) **match** data-paths

Weighted Minimal Mismatch

$$mismatch_{i,j} = |\Delta^M(i,j) -$$

$$min \sum_{\forall i,j \in V} c_{i,j} \cdot mismatch_{i,j}$$

minimize mismatch of busiest paths

Metadata propagation: building the tree

Finding the optimal tree is modelled as a **constraint optimization** problem

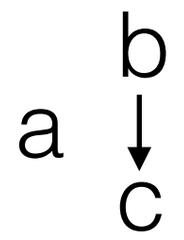
Input

Data-paths average latencies

Candidate locations for serializers (an latencies among them)

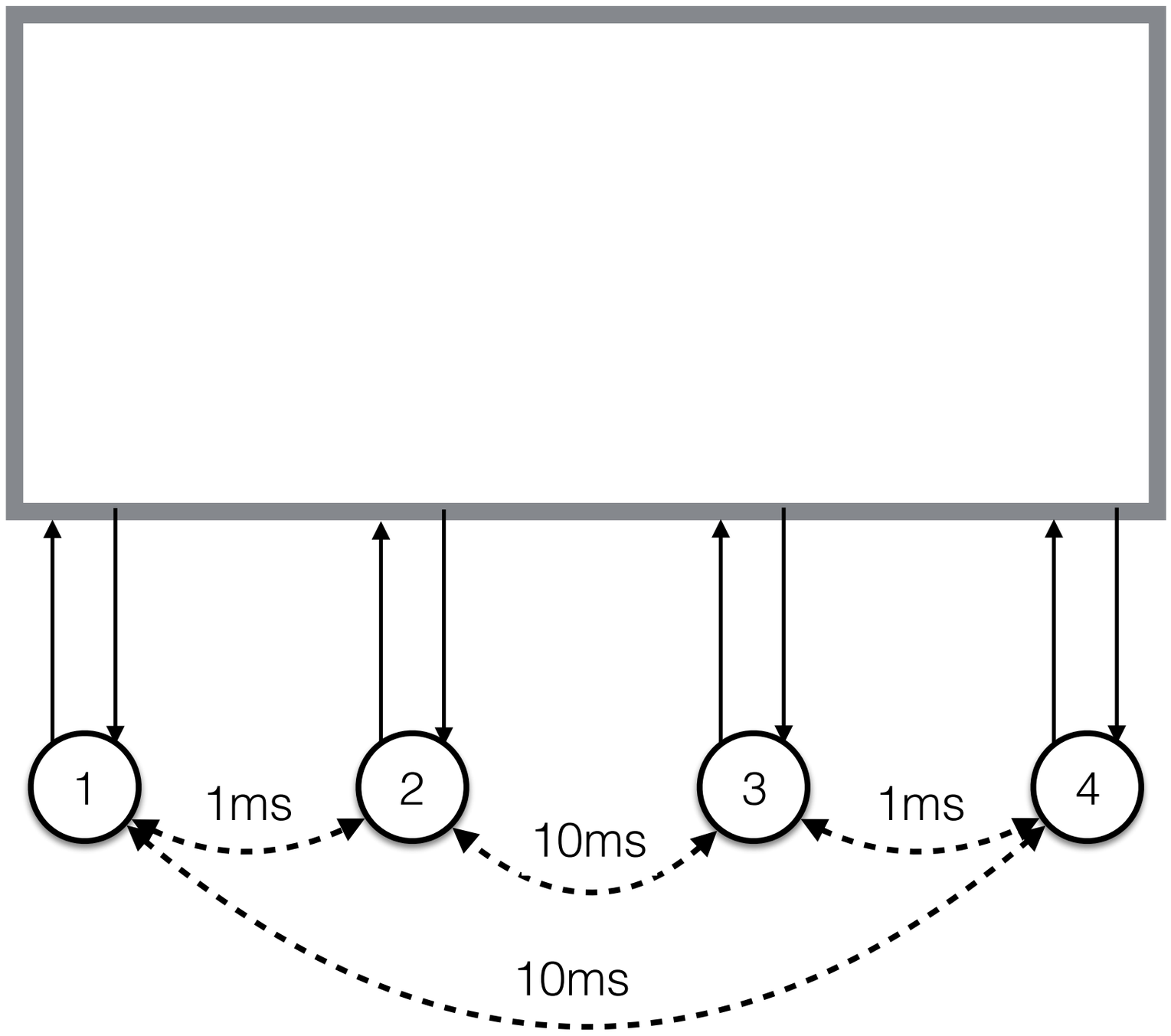
Access-patterns: to minimize the impact of mismatches

Example

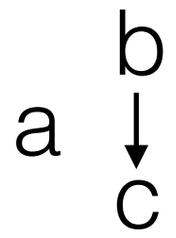


c causally depends on b

a concurrent to both b and c

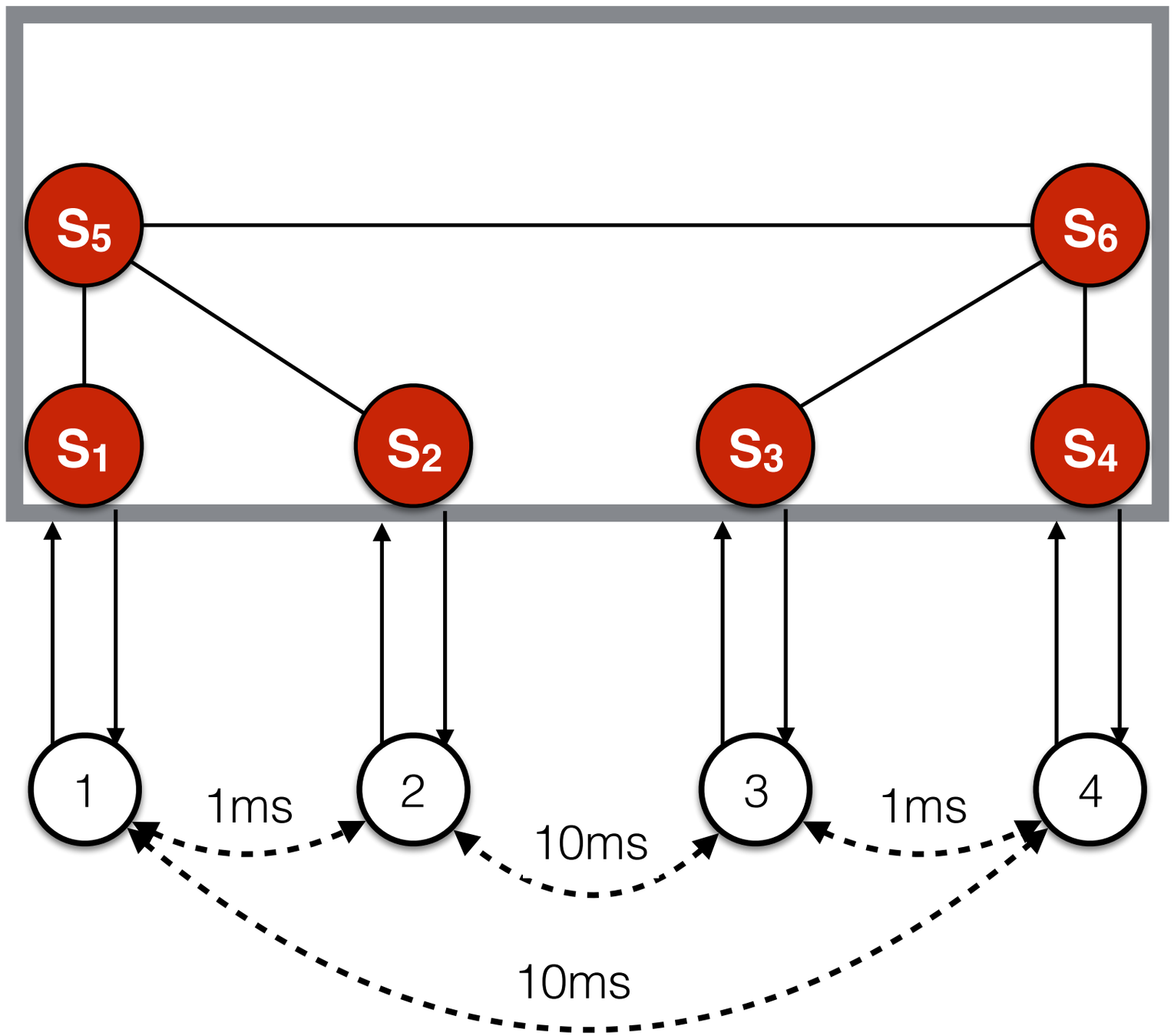


Example



c causally depends on b

a concurrent to both b and c

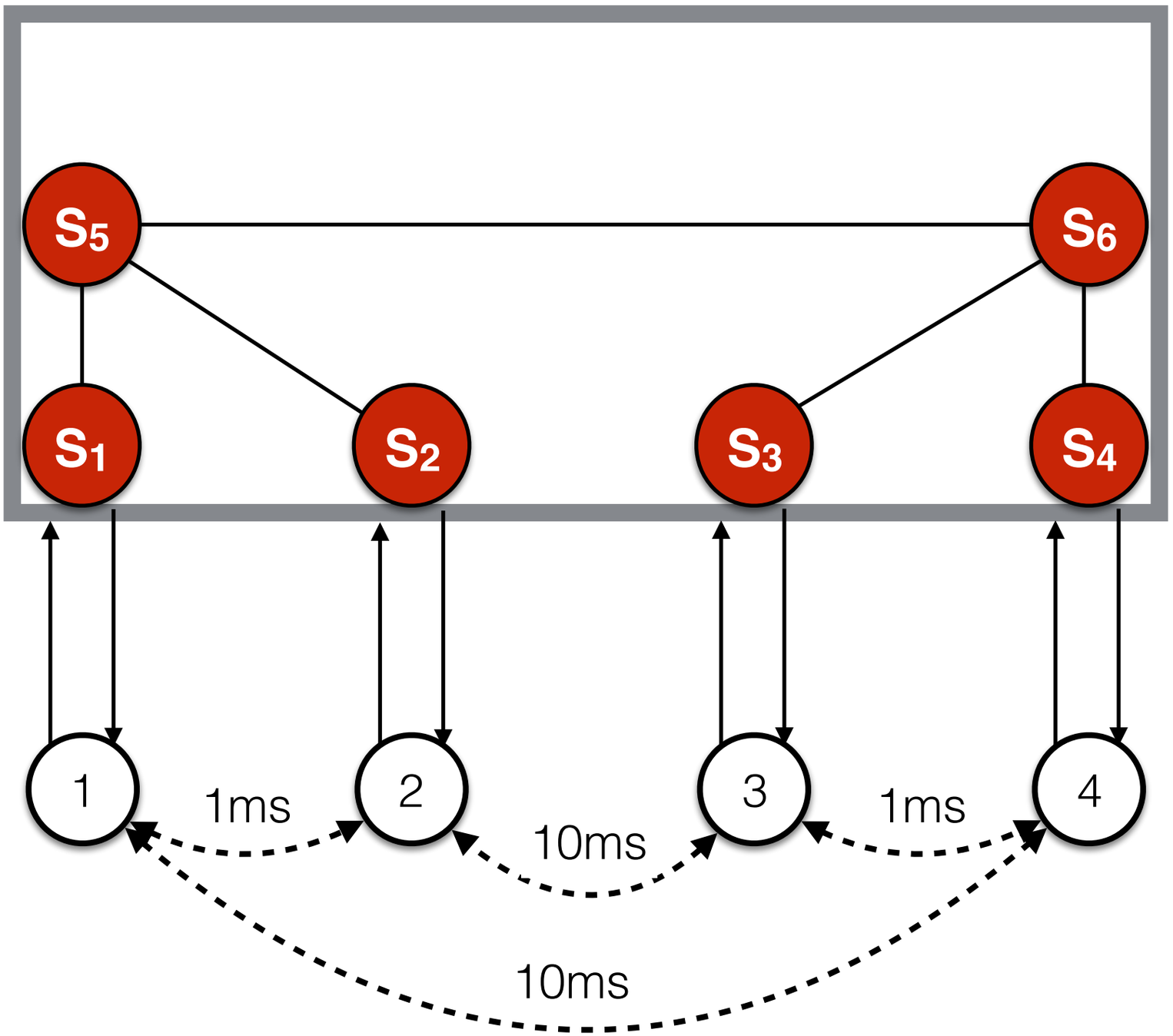


Example

a → c, b → c

c causally depends on b

a concurrent to both b and c



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

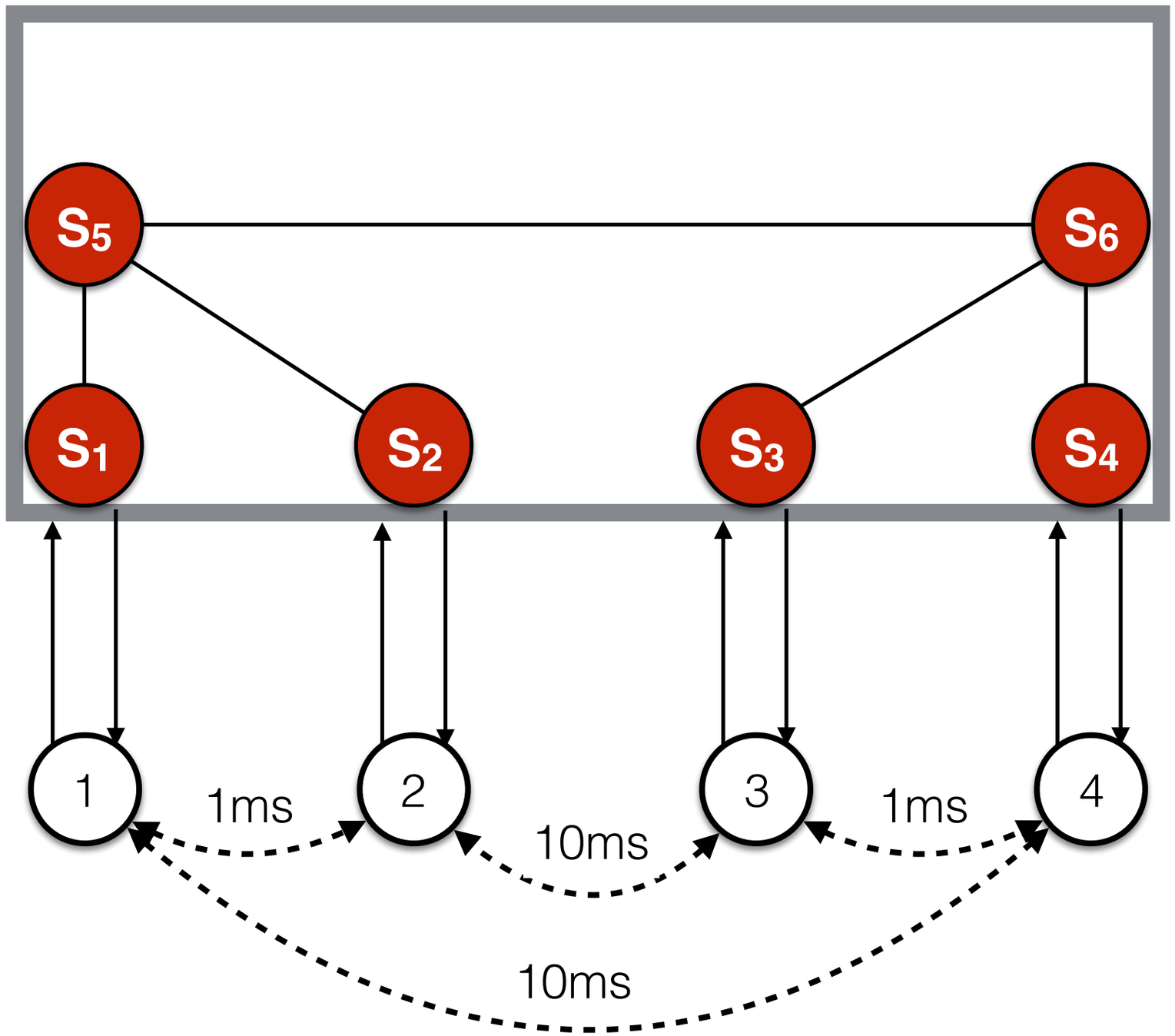
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

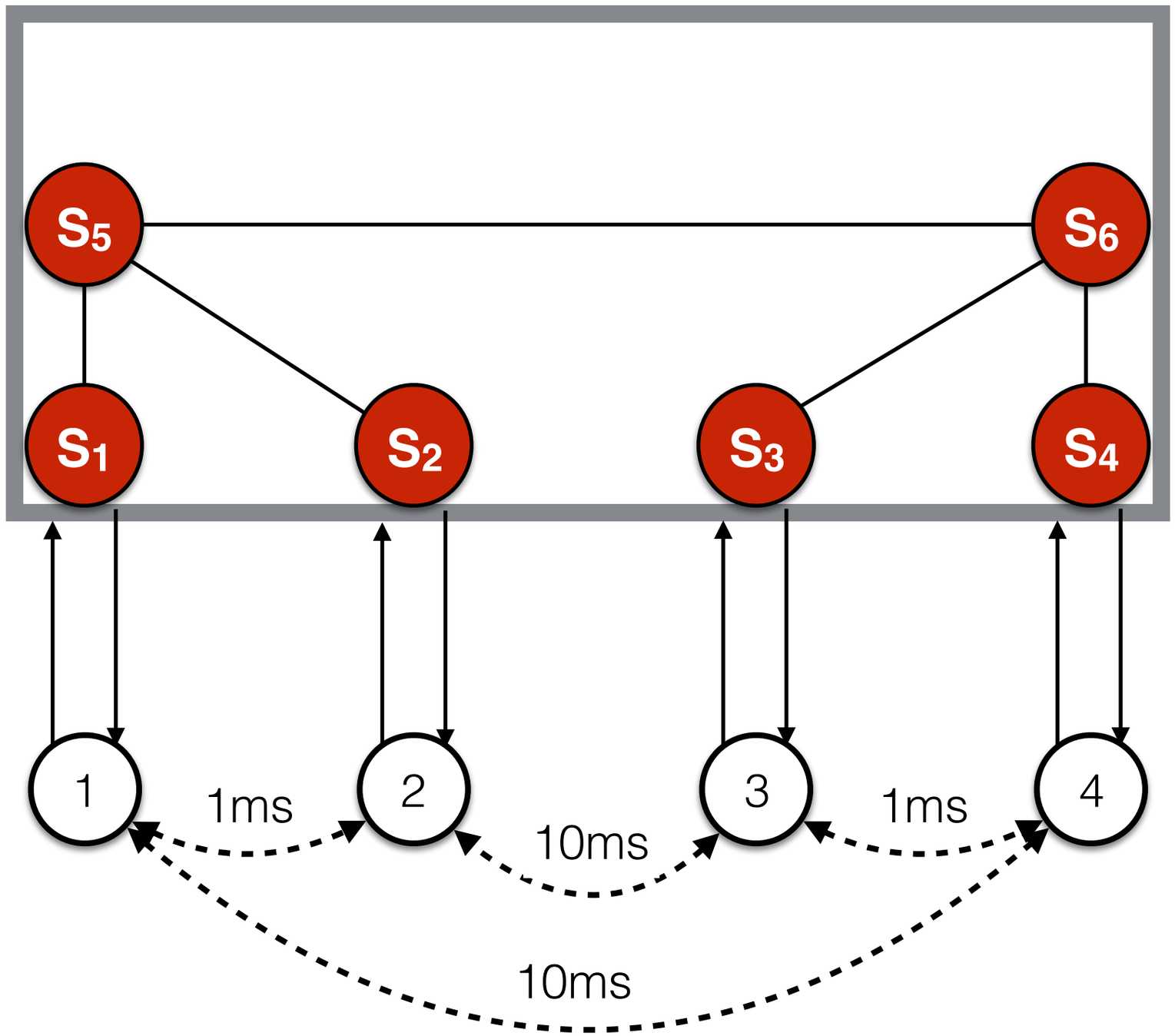
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 02



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

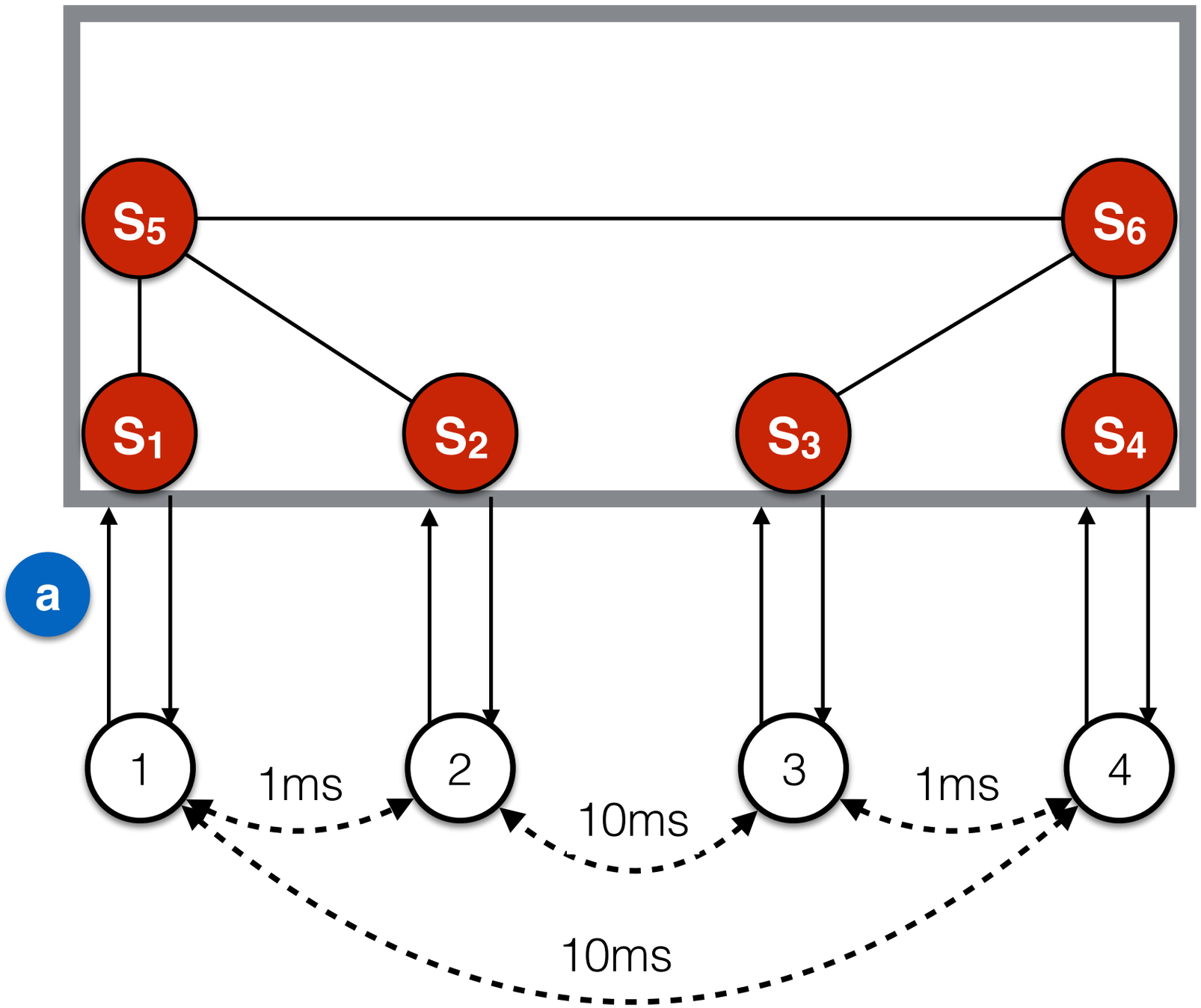
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 02



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

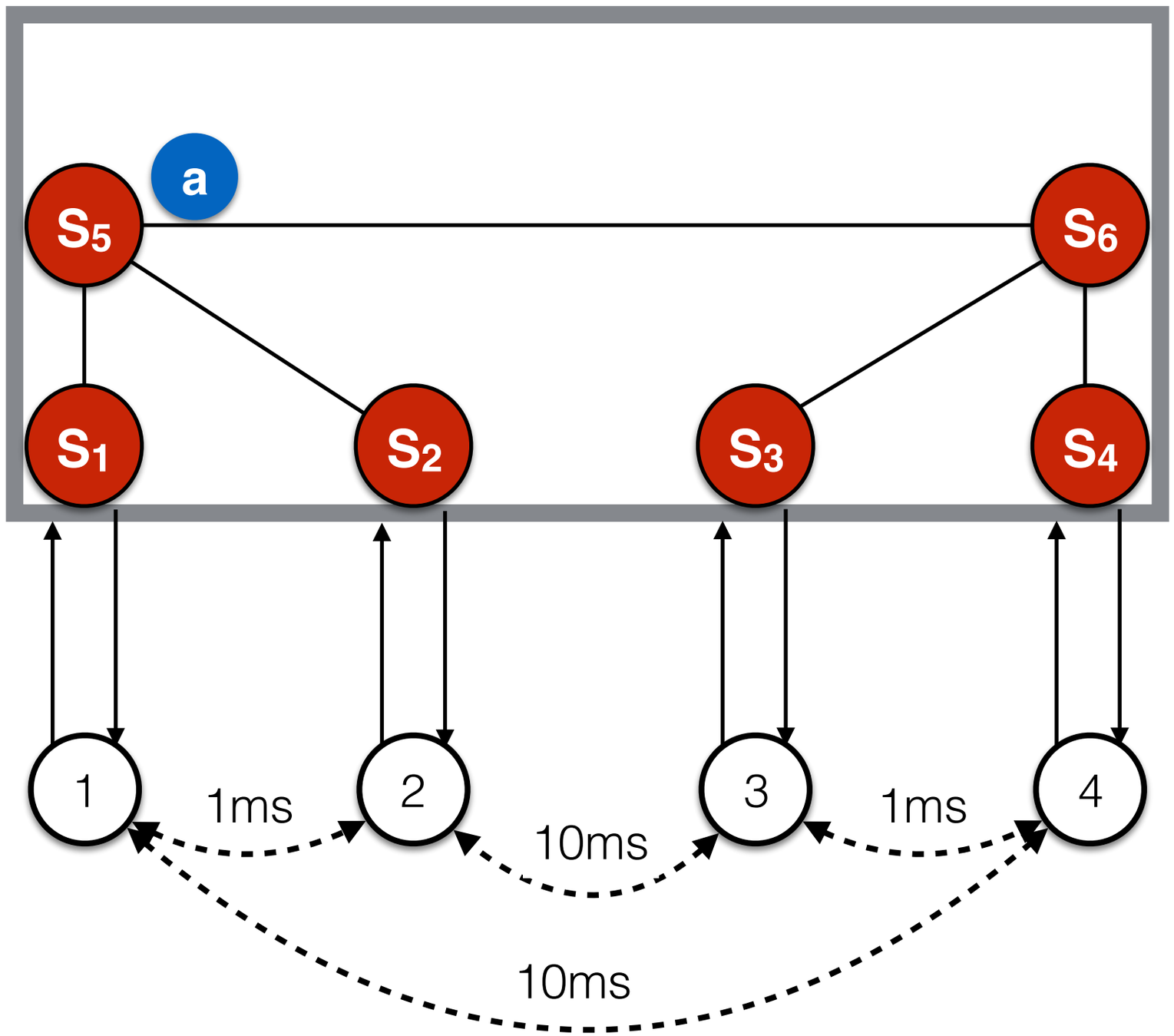
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 02



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

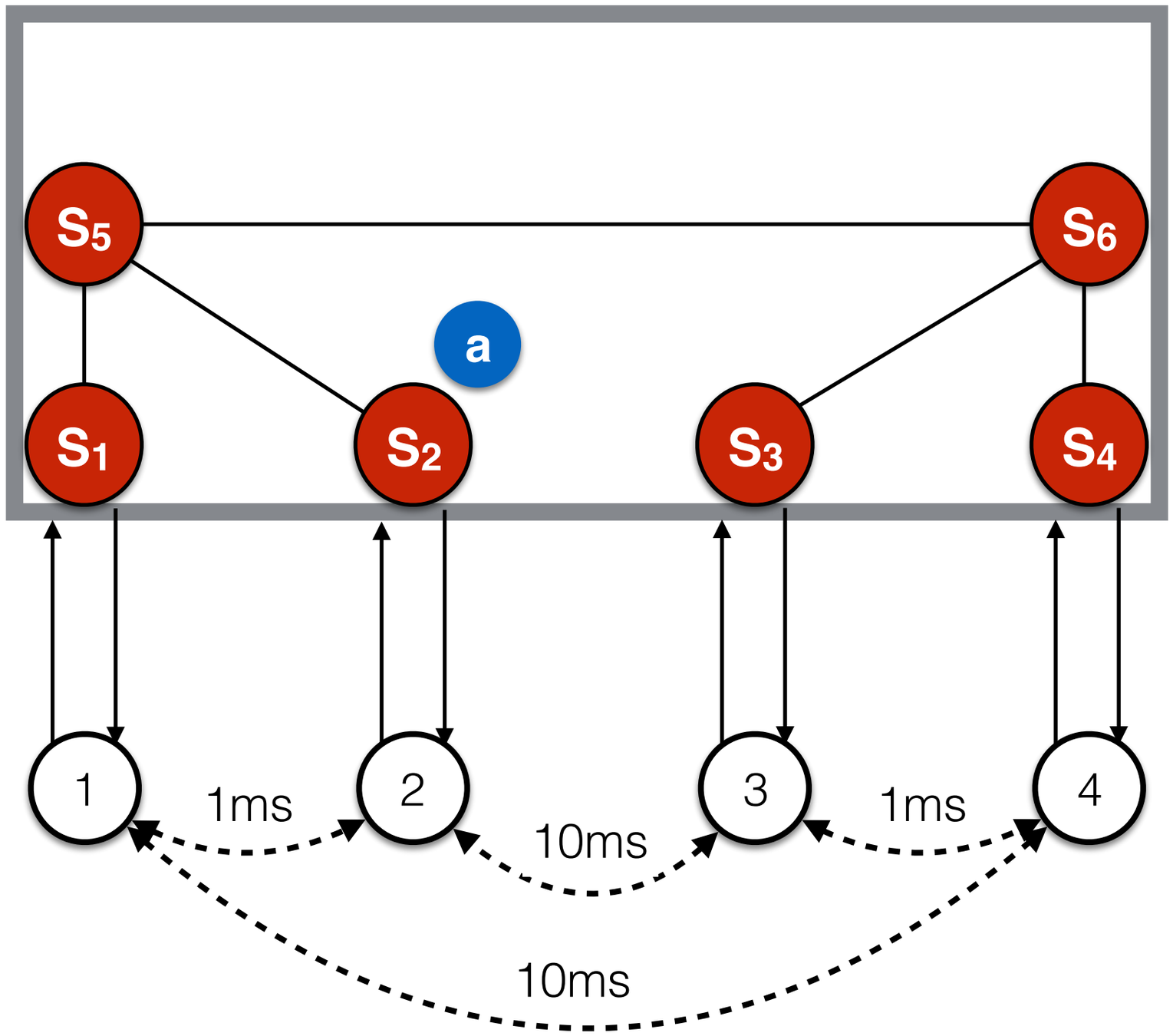
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 023



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

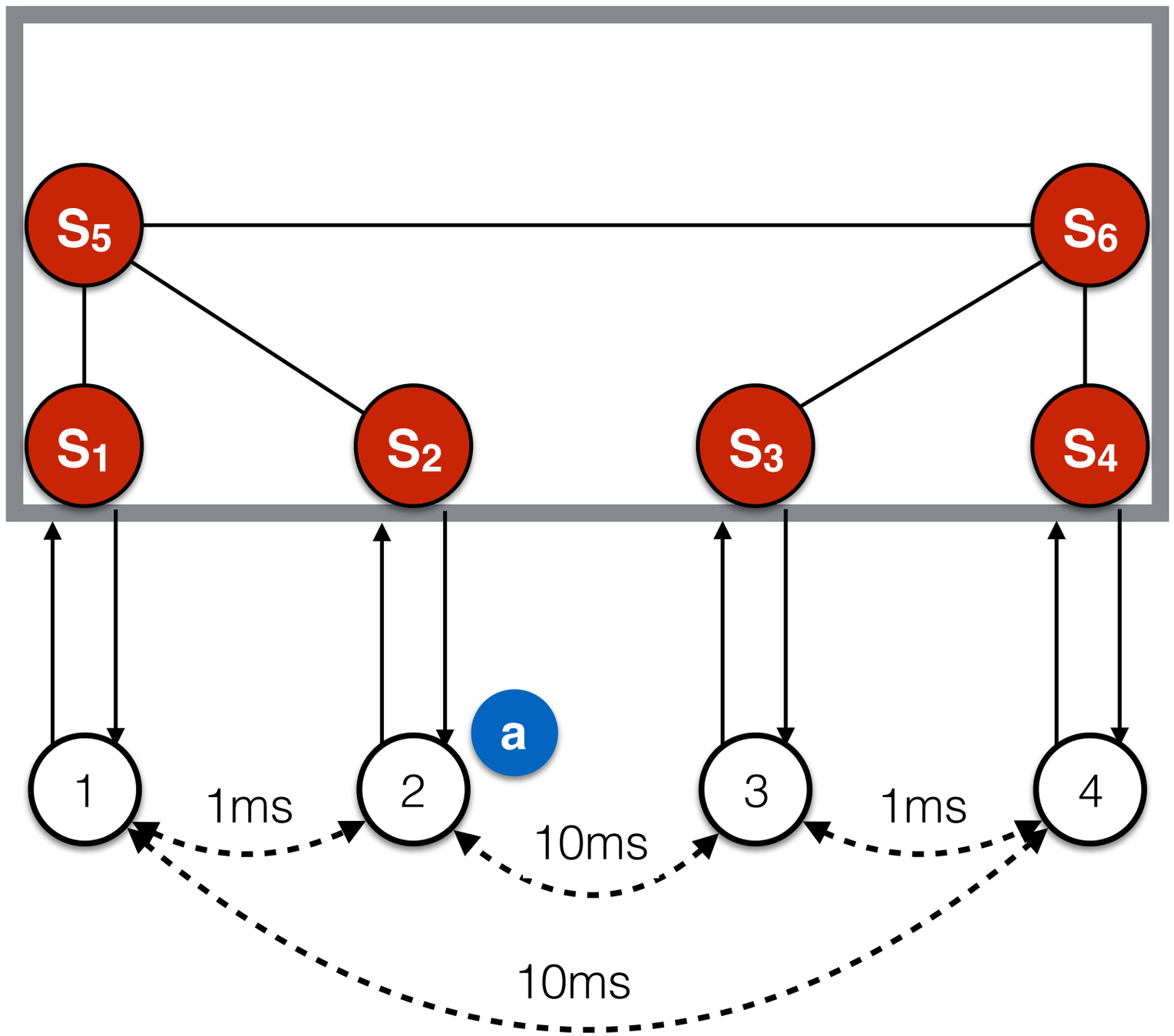
Example

Time: 0 2 3

a → b, a → c

c causally depends on b

a concurrent to both b and c



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

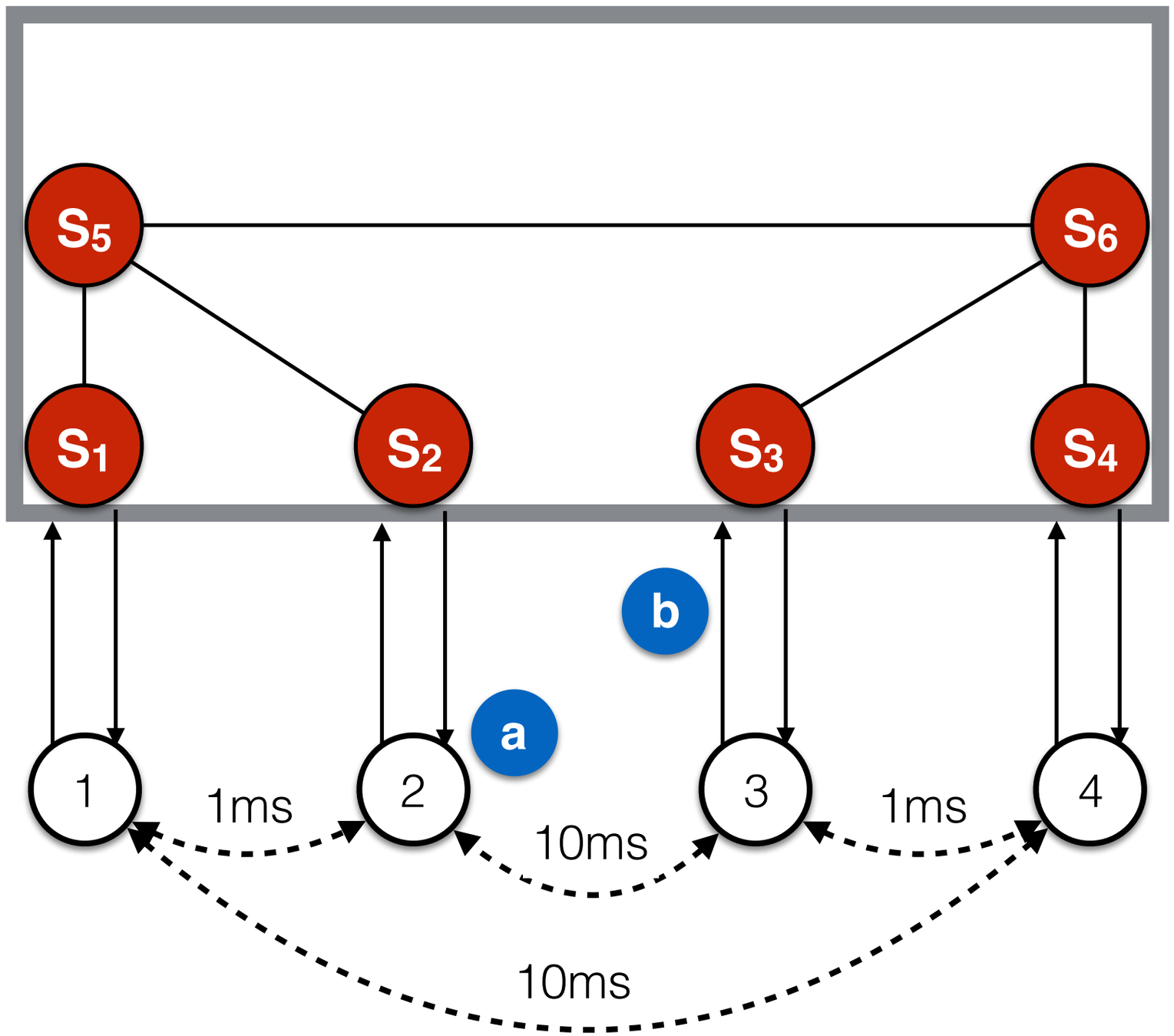
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

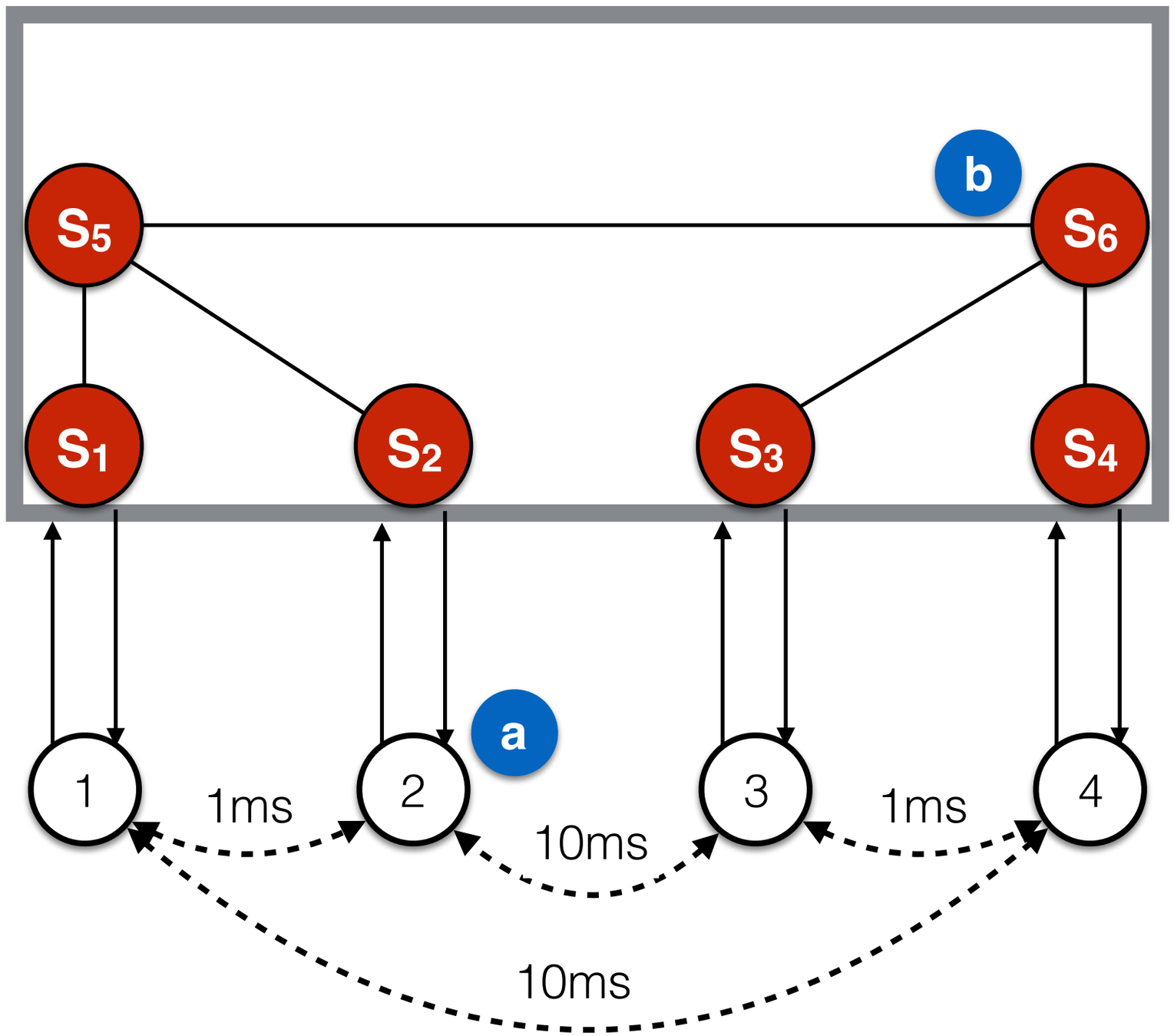
Example

Time: 0 2 3 4 5

a → b, a → c

c causally depends on b

a concurrent to both b and c



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

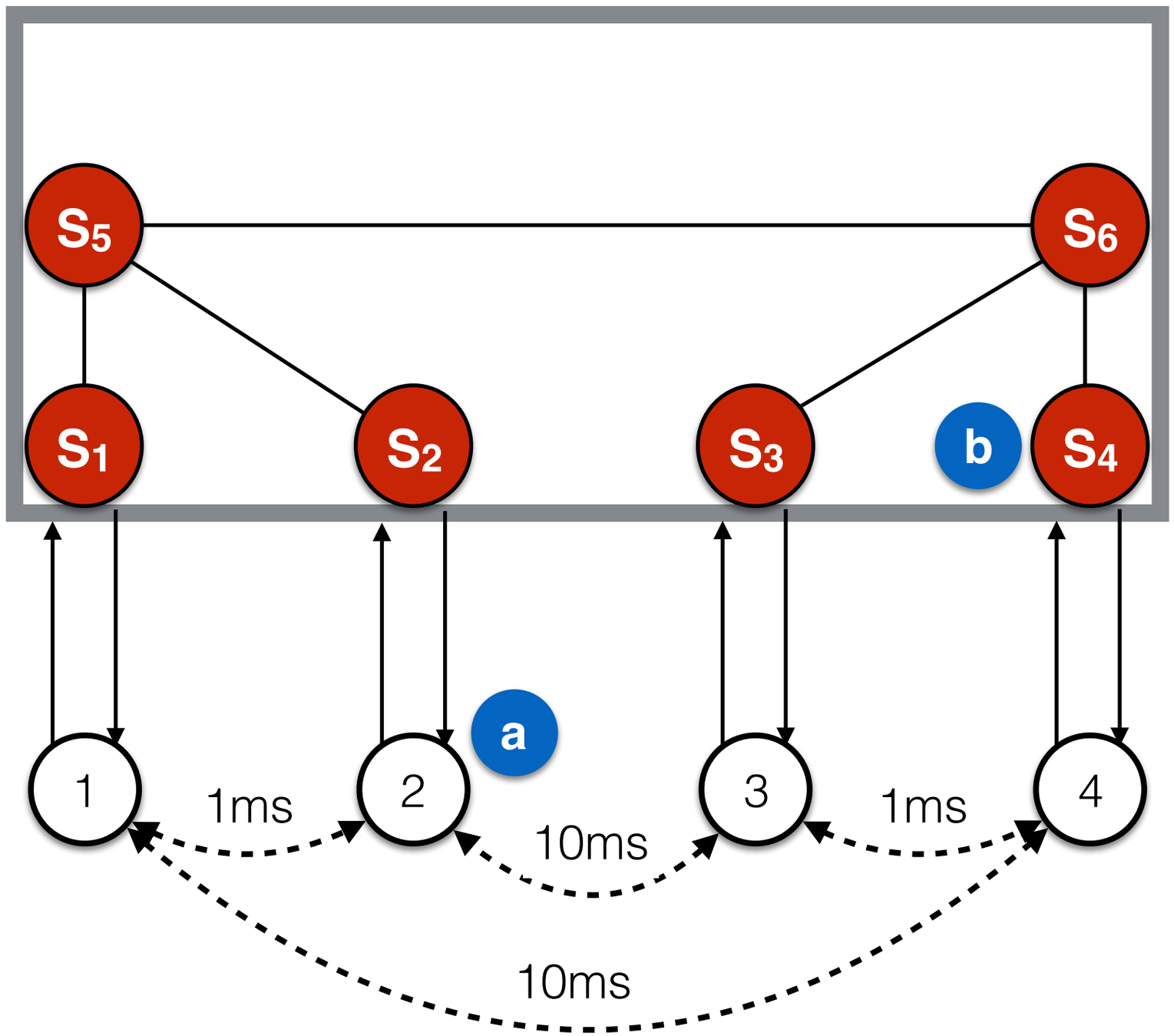
Example

Time: 0 2 3 4 5

a → b, a → c

c causally depends on b

a concurrent to both b and c



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

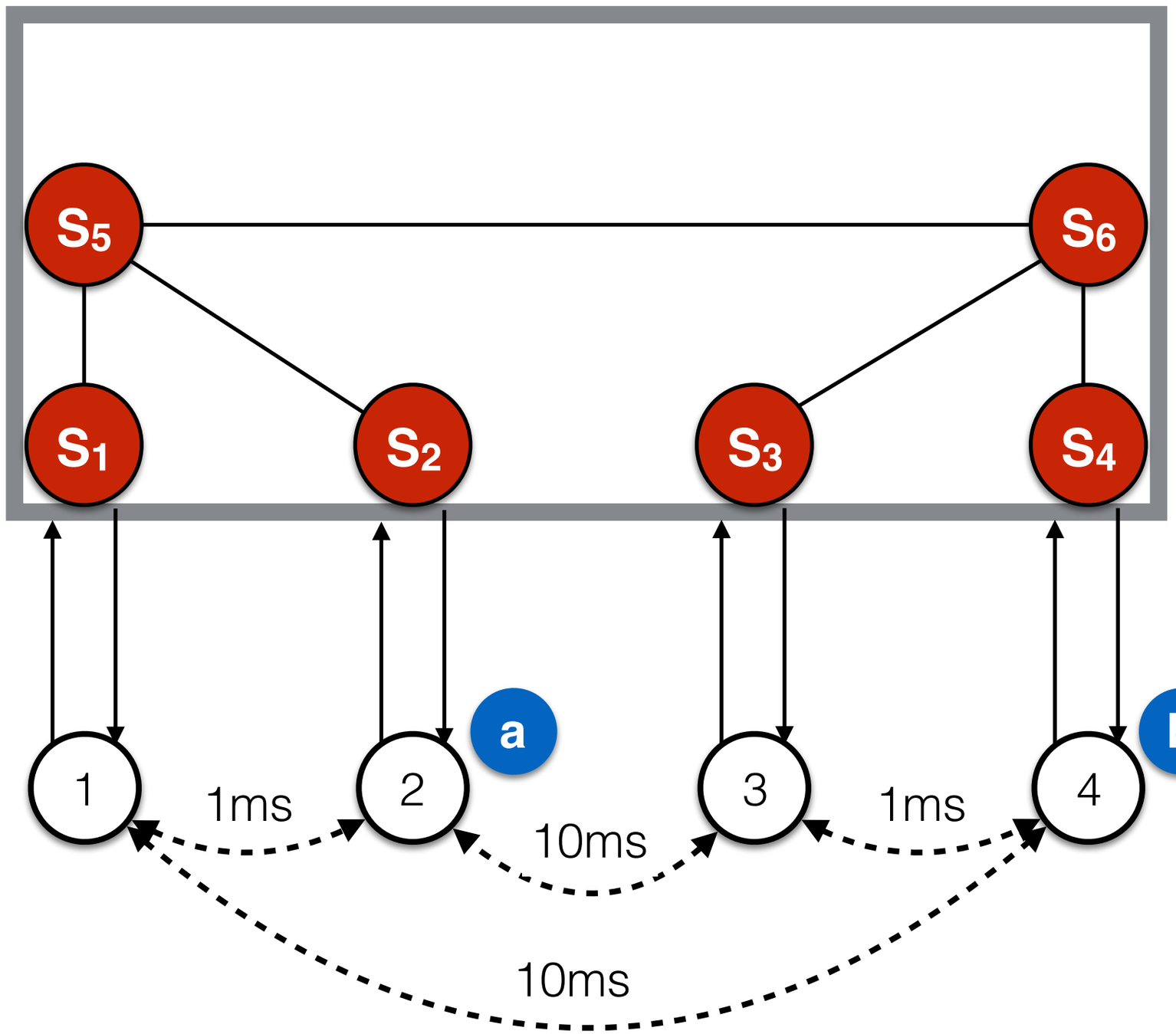
Example

Time: 0 2 3 4 5

a → b, a → c

c causally depends on b

a concurrent to both b and c



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

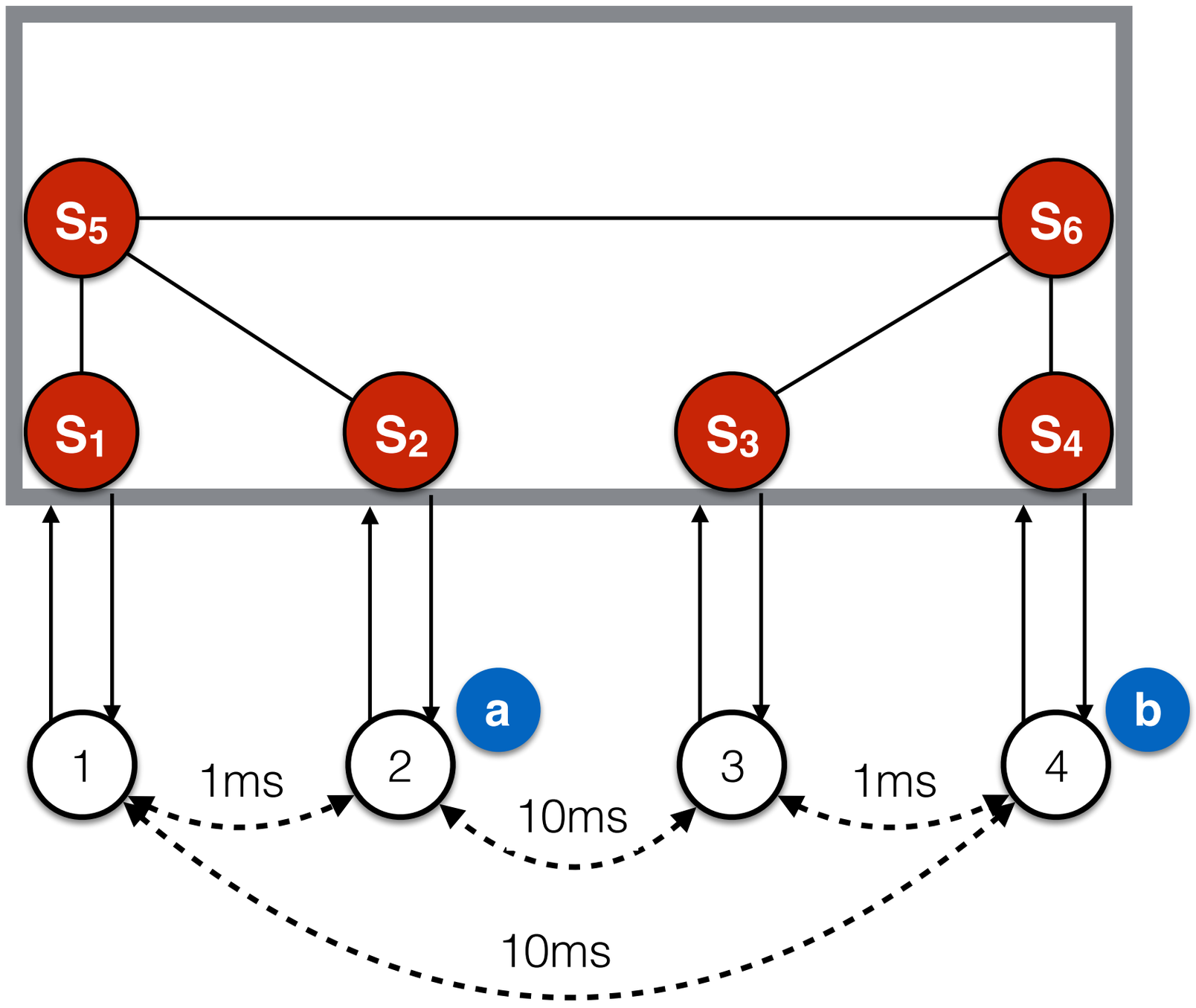
Example

Time: 0 2 3 4 5

a → b, a → c

c causally depends on b

a concurrent to both b and c



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

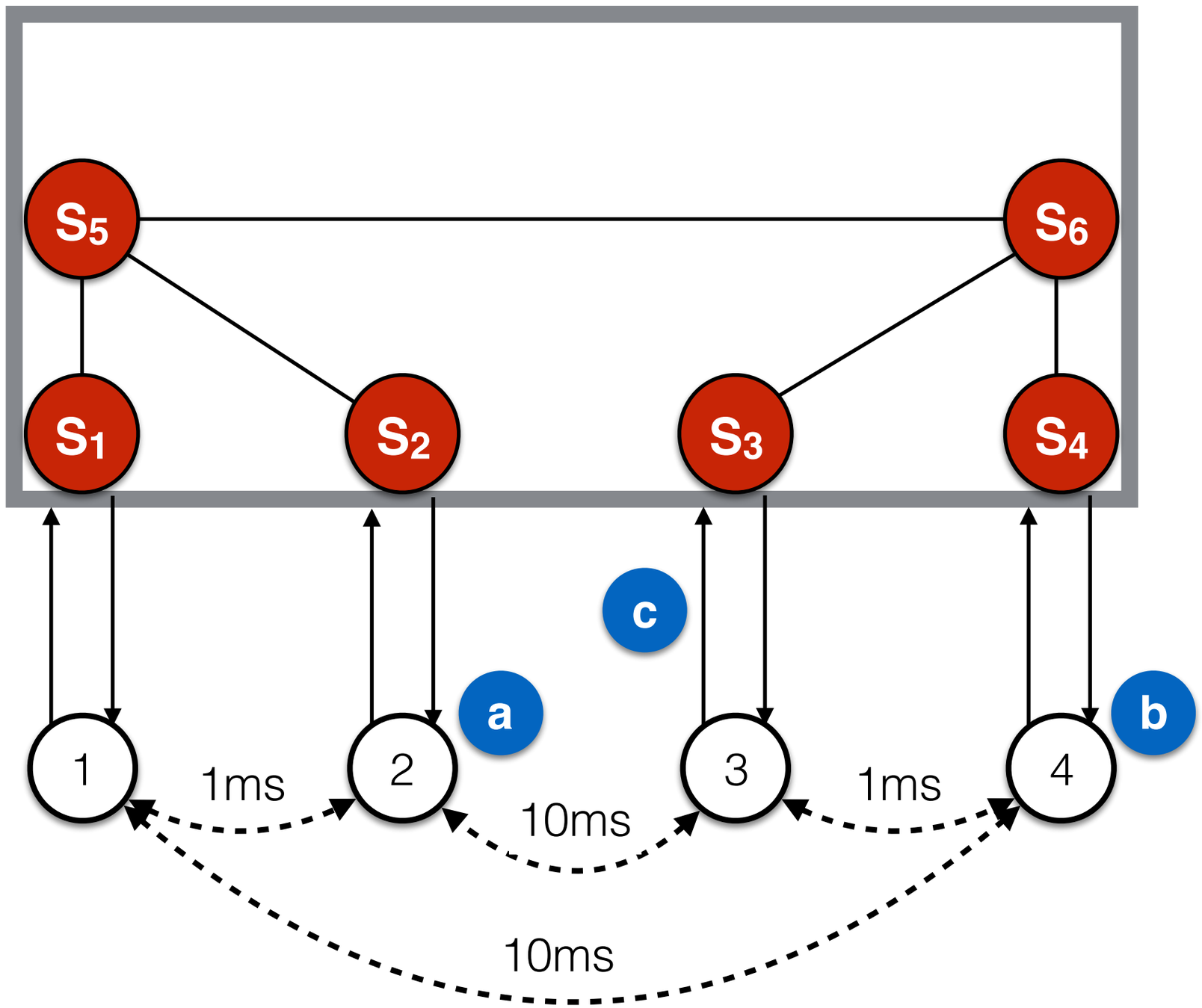
Example

Time: 0 2 3 4 5 6

a → b, a → c

c causally depends on b

a concurrent to both b and c



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

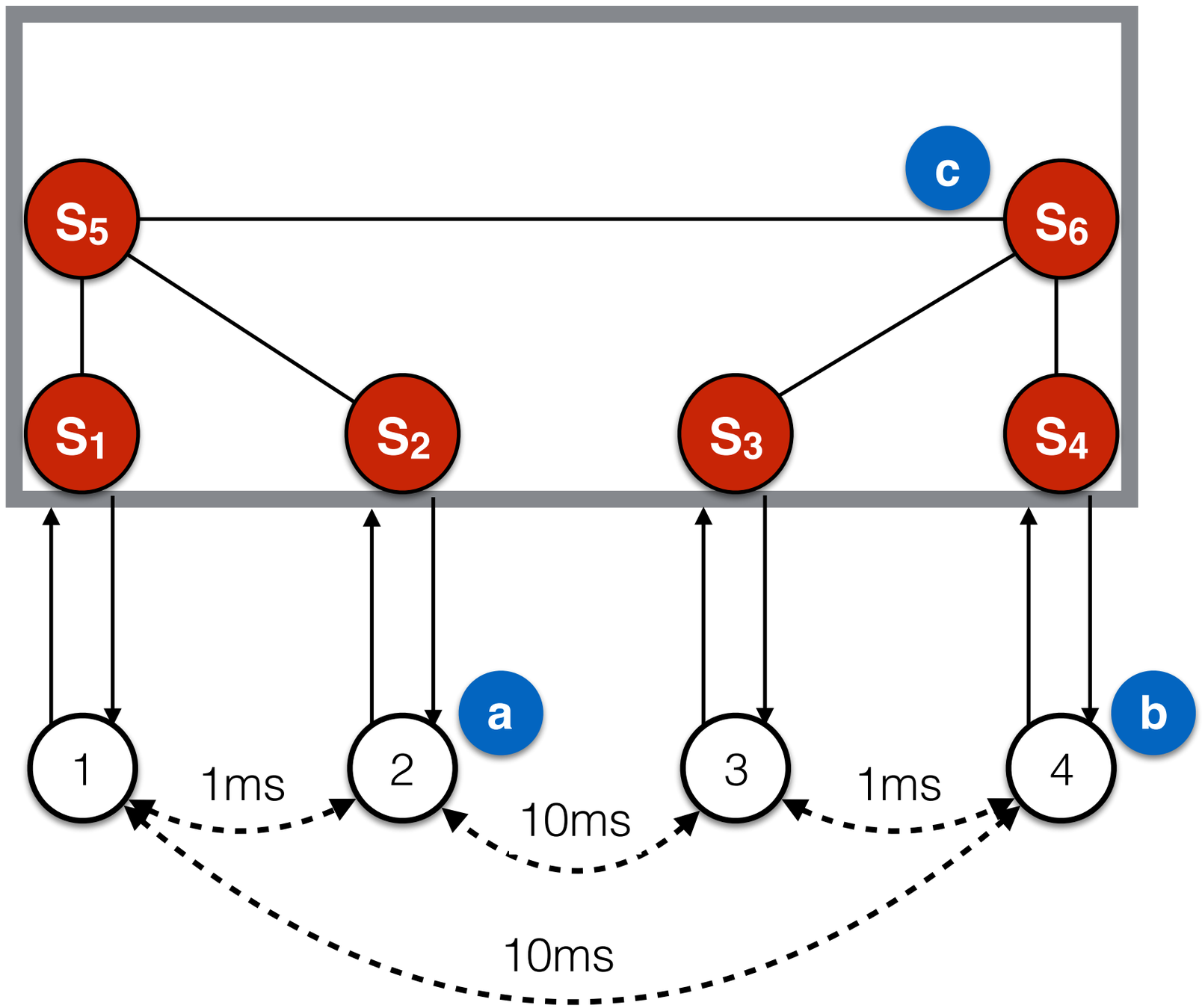
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

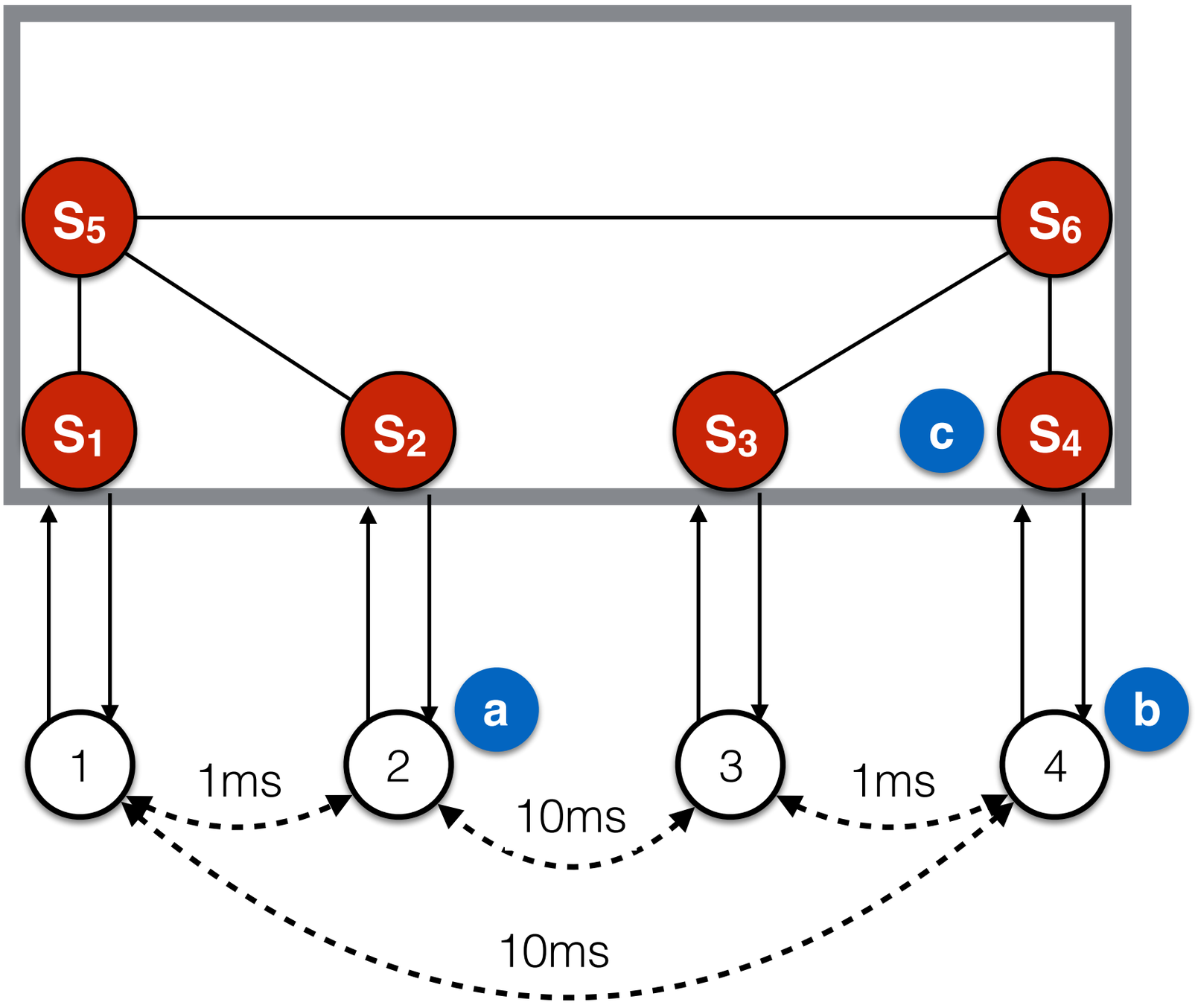
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

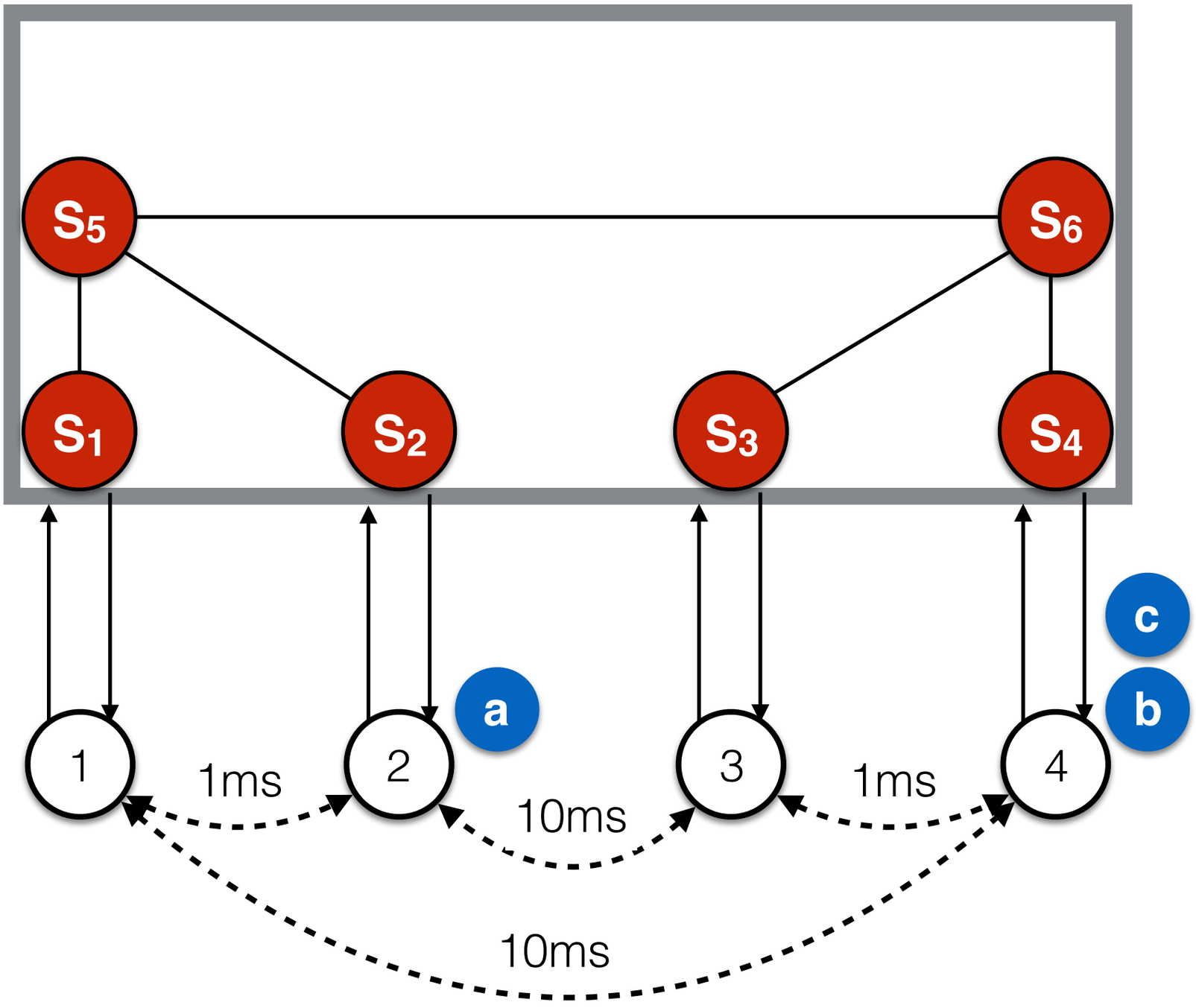
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

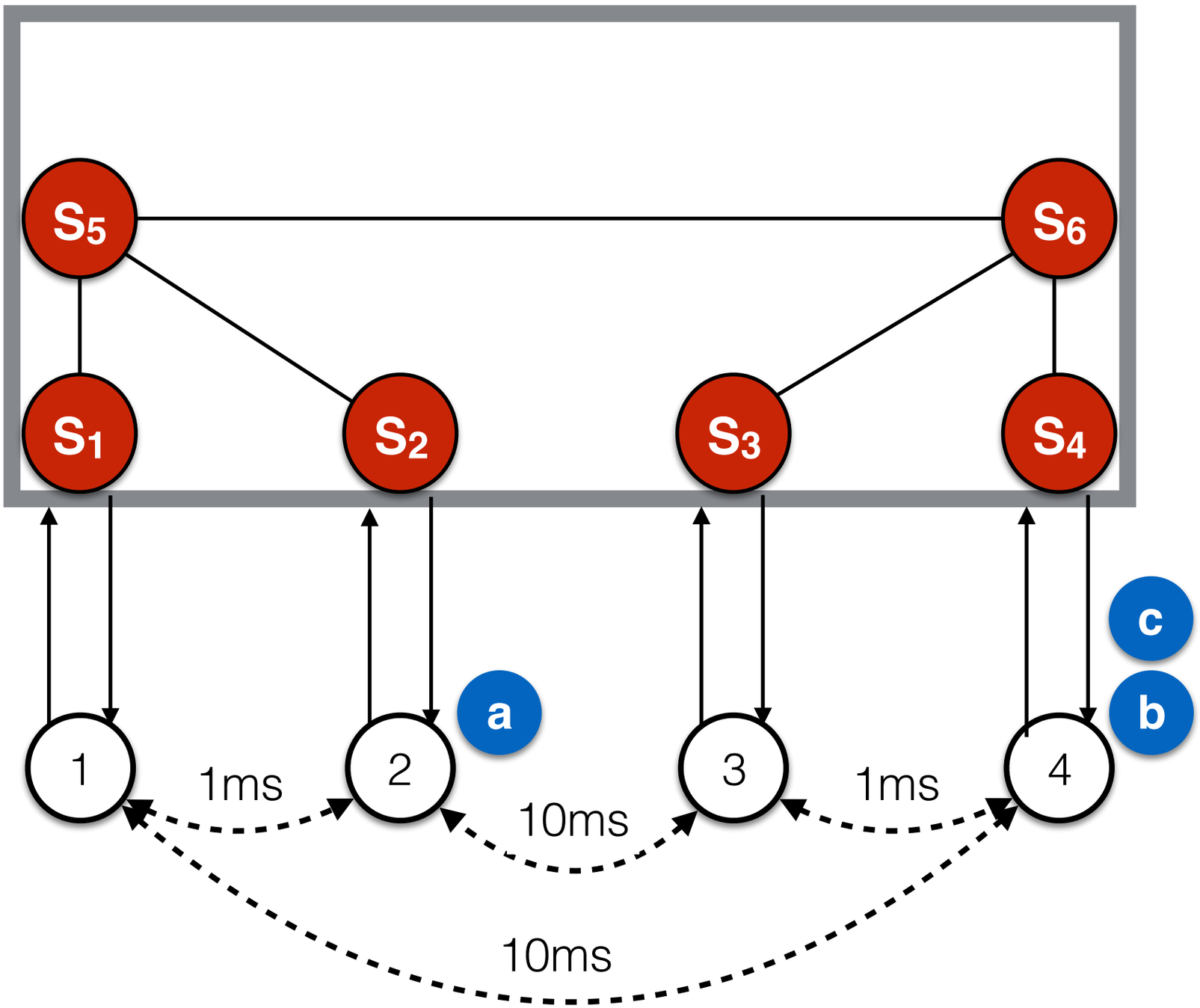
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

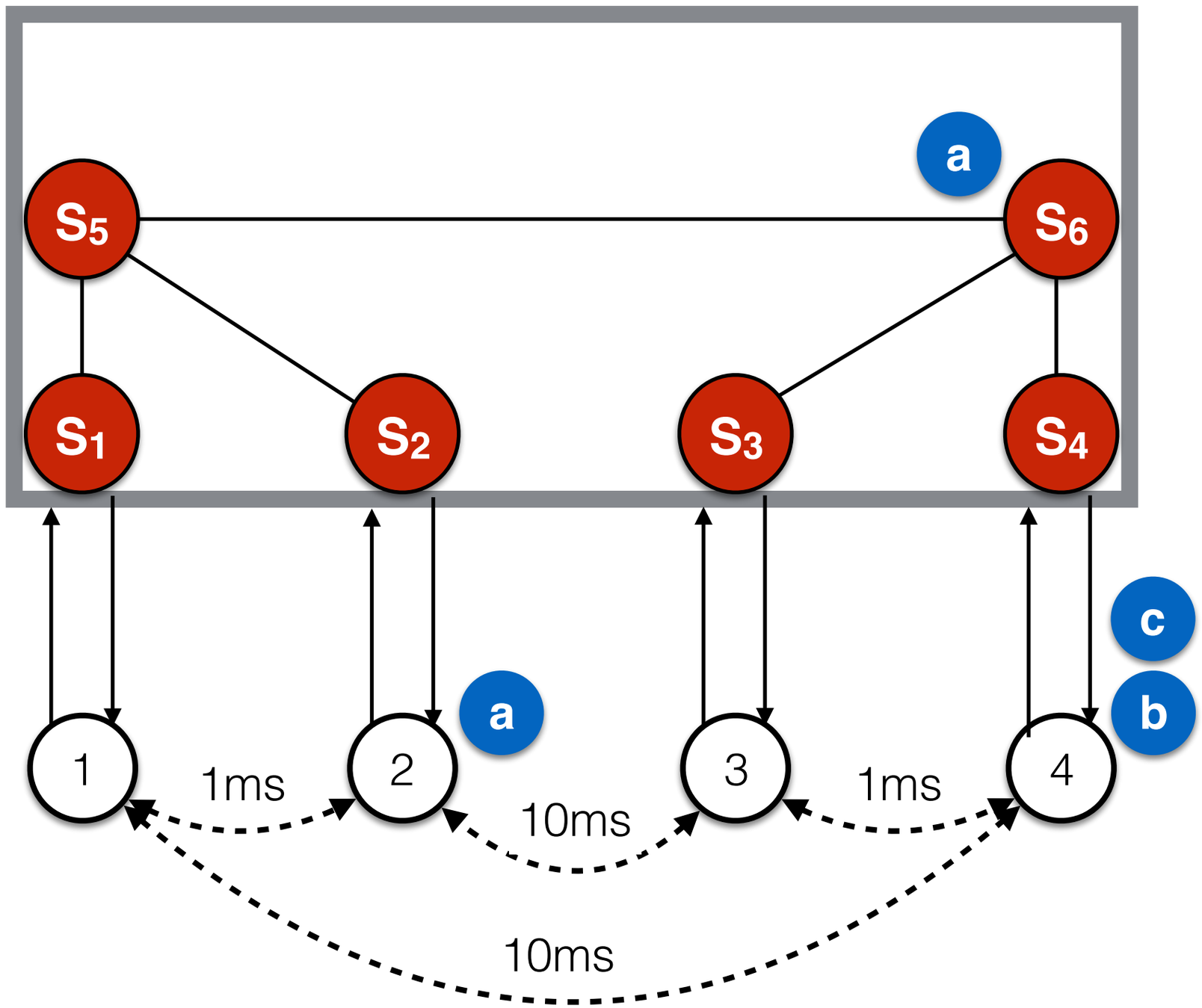
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7 …12



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

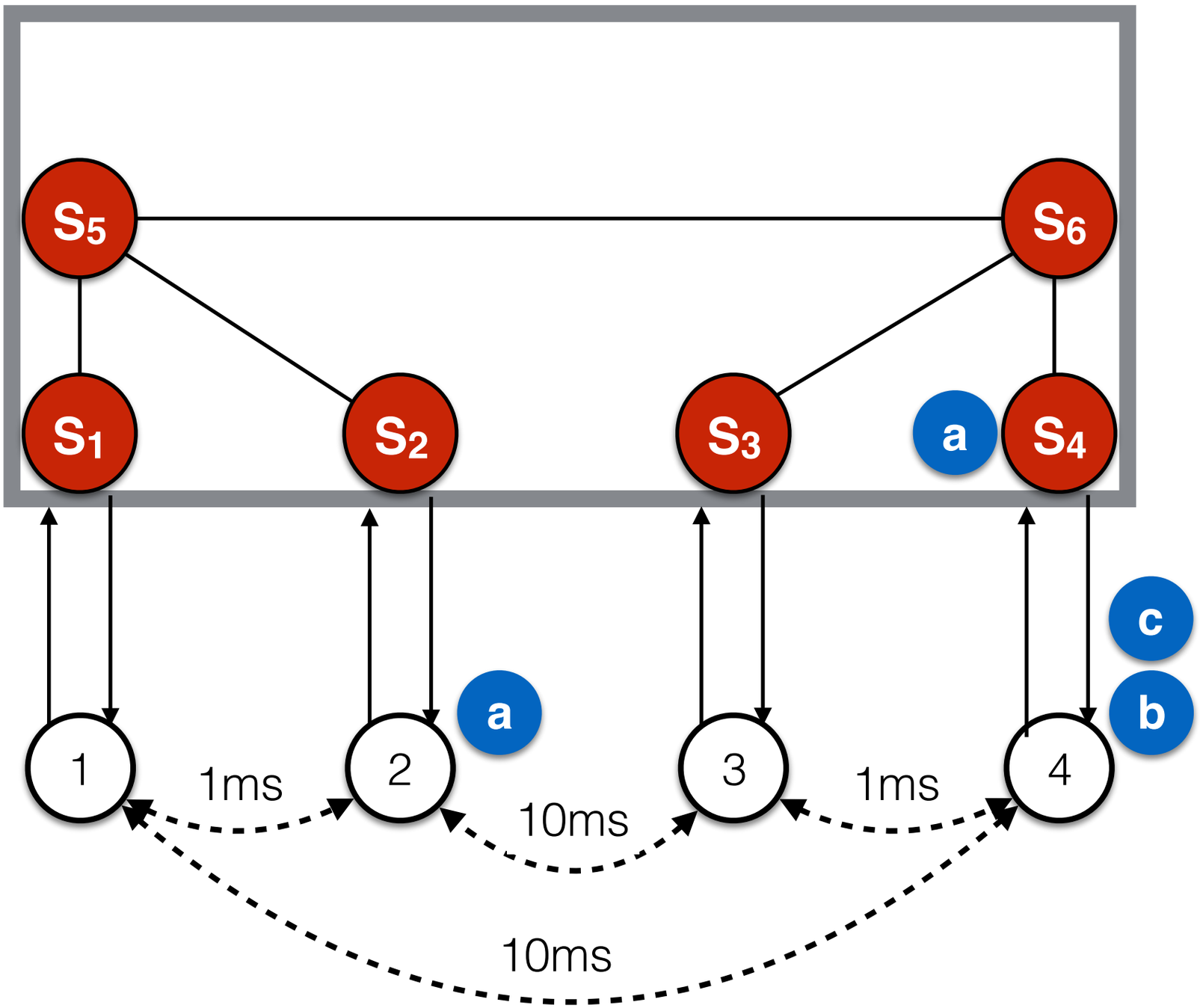
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7 …12



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

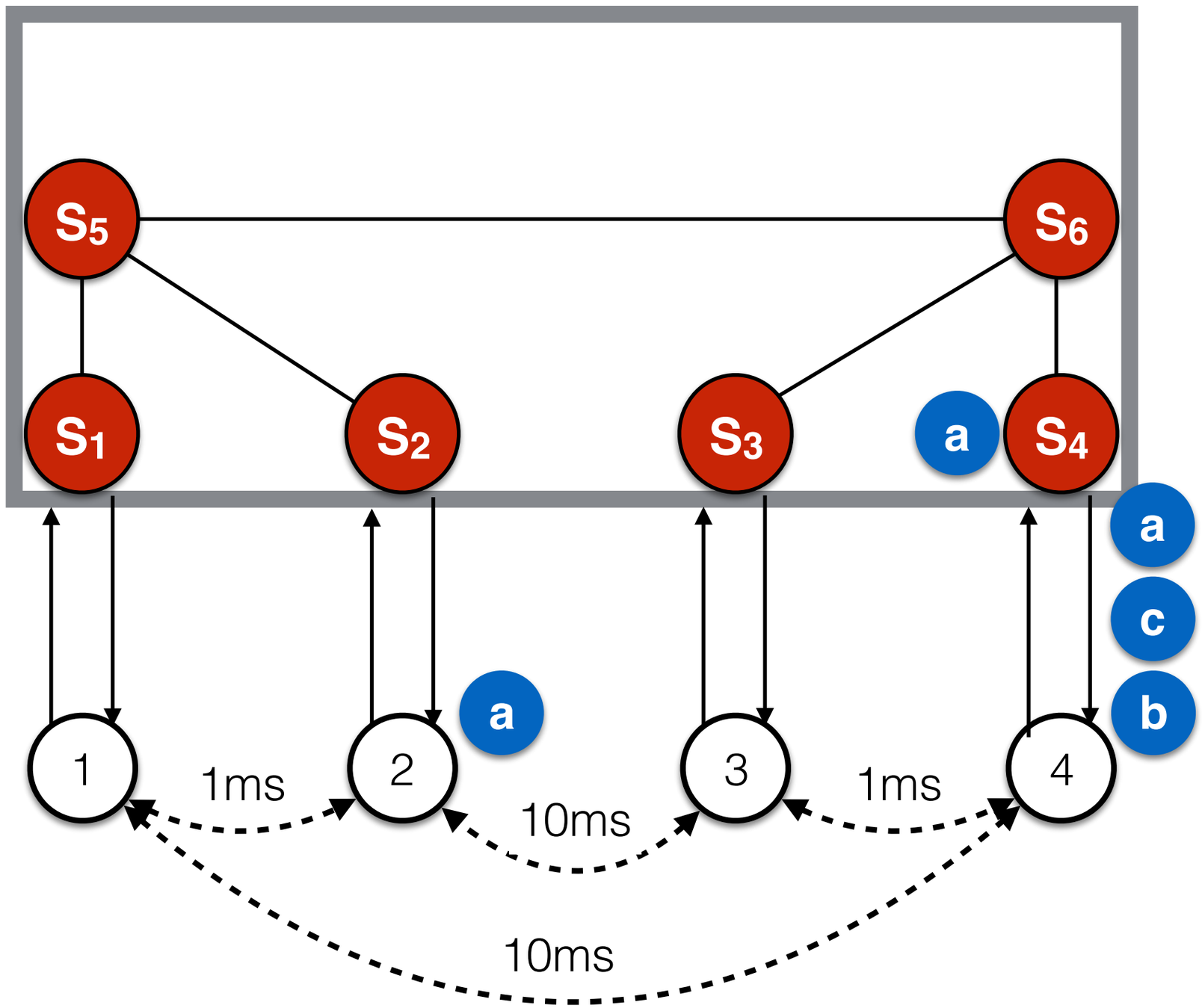
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7 …12



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

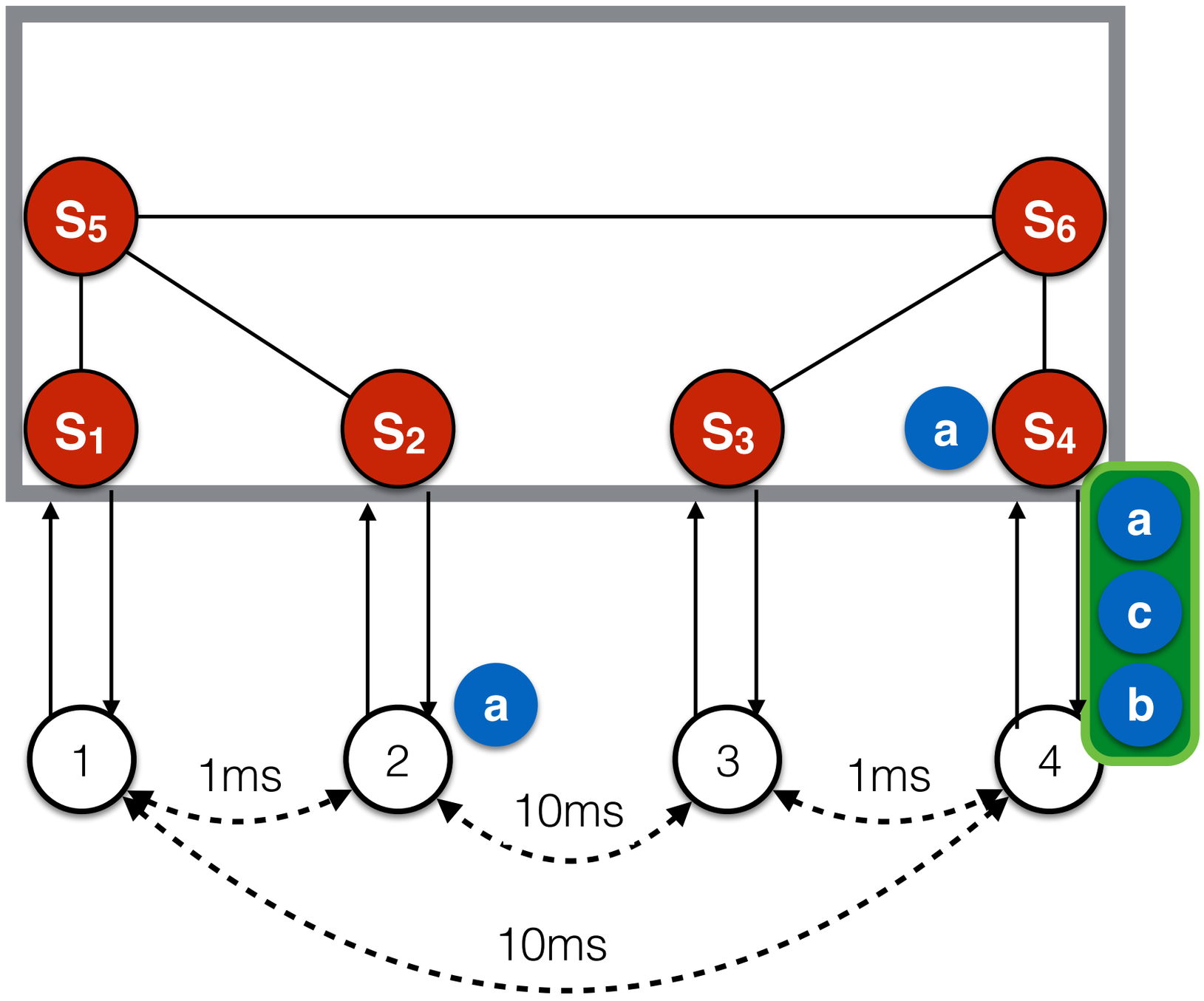
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7 …12



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

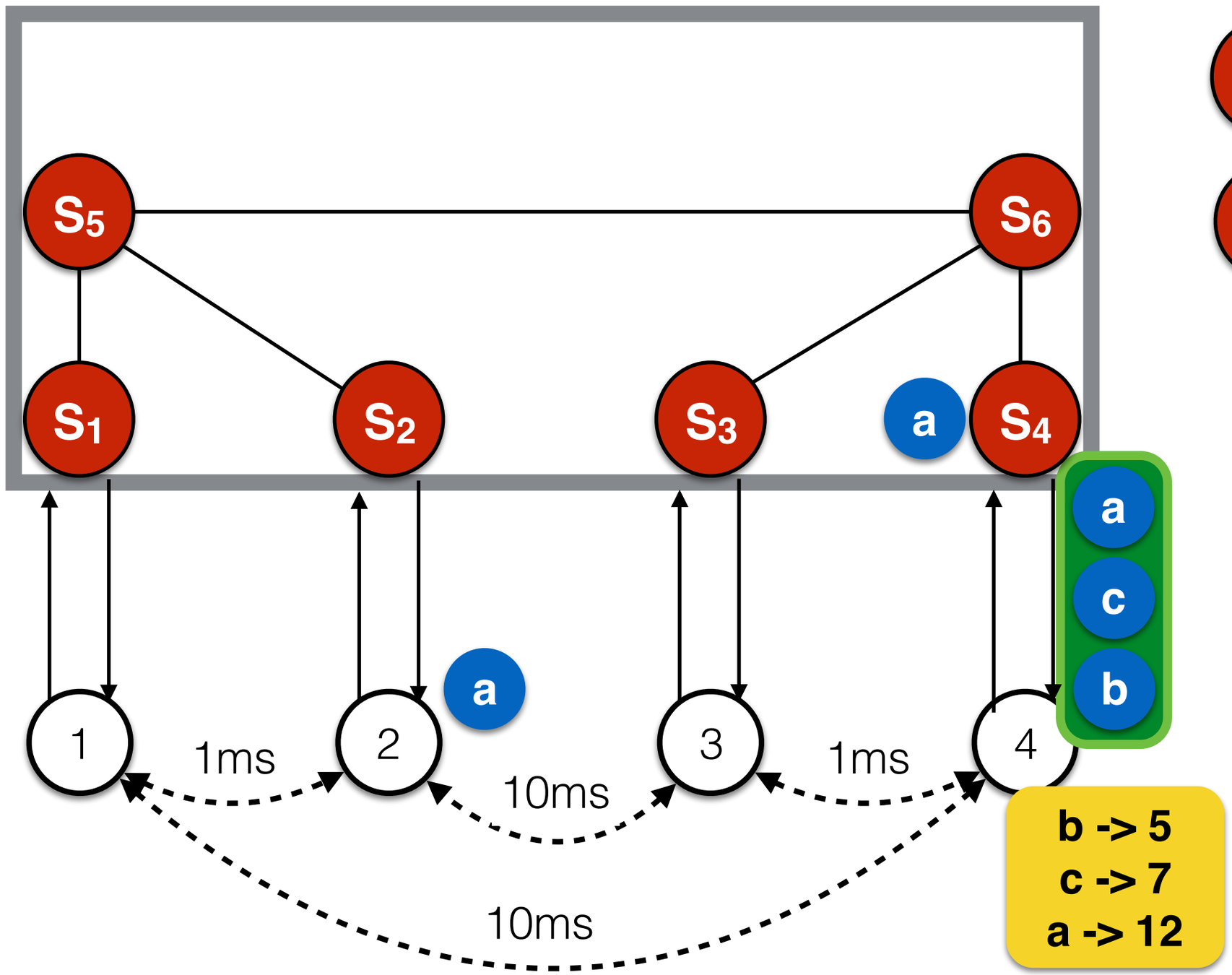
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7 …12



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

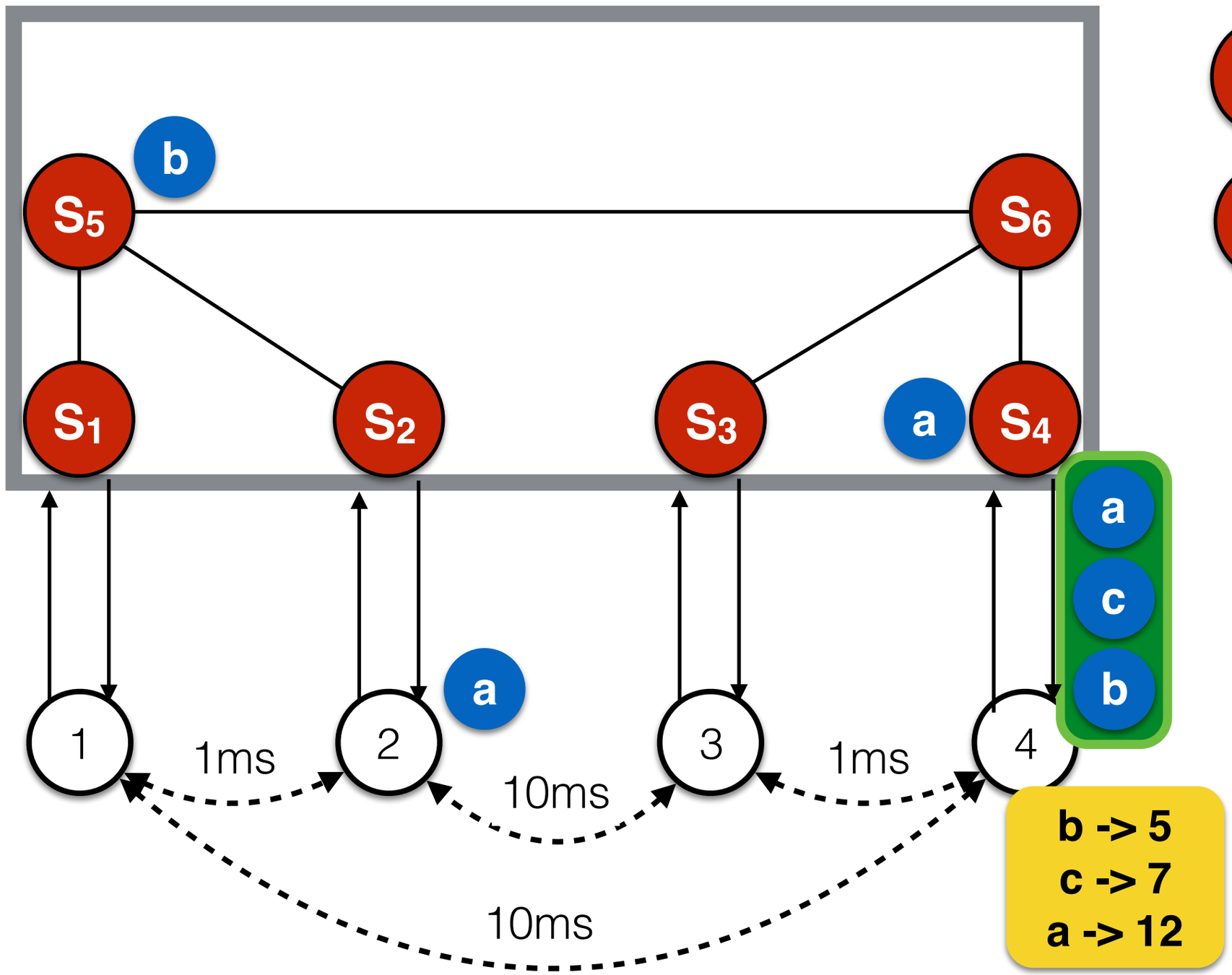
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7 …12 …14



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

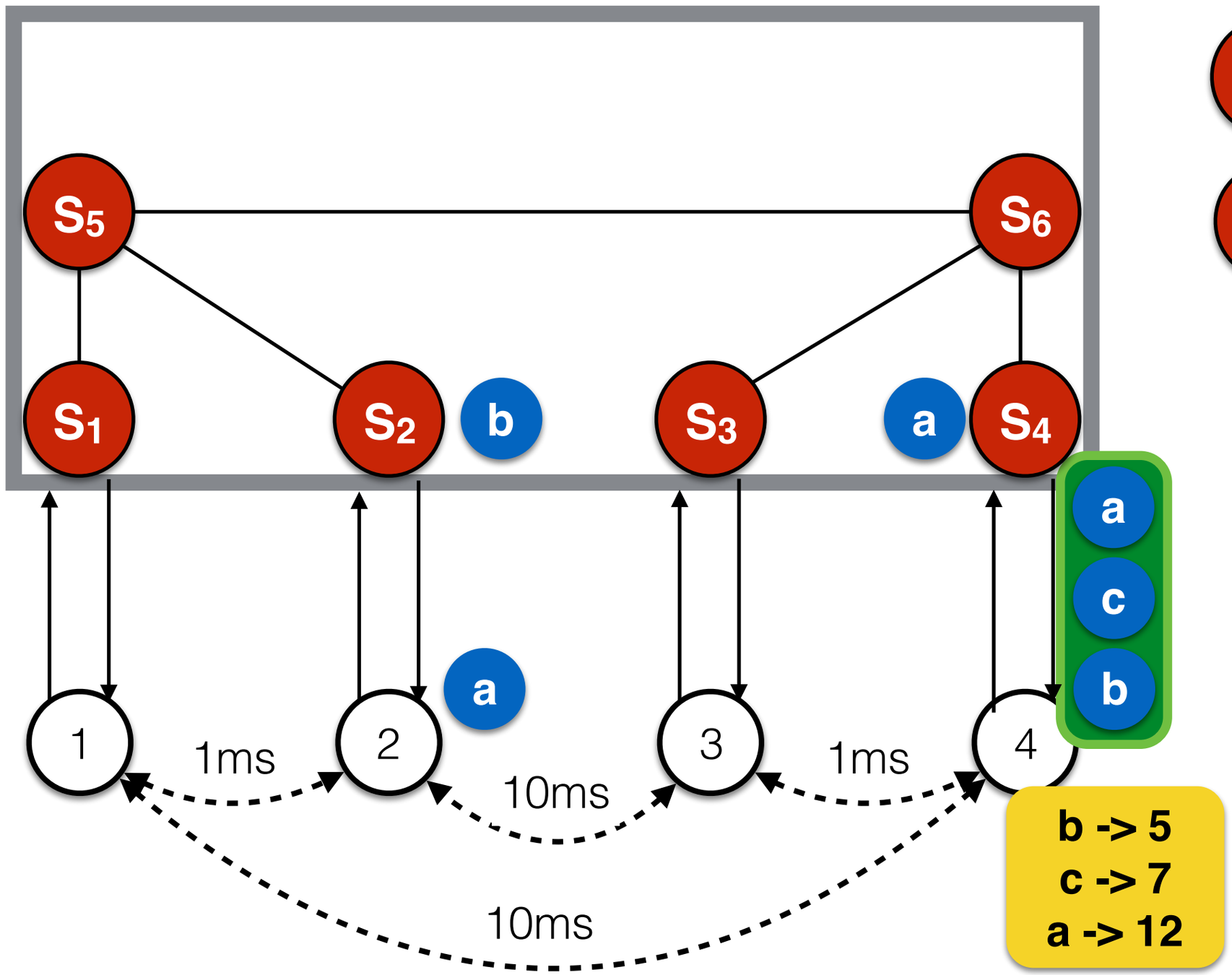
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7 …12 …14



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

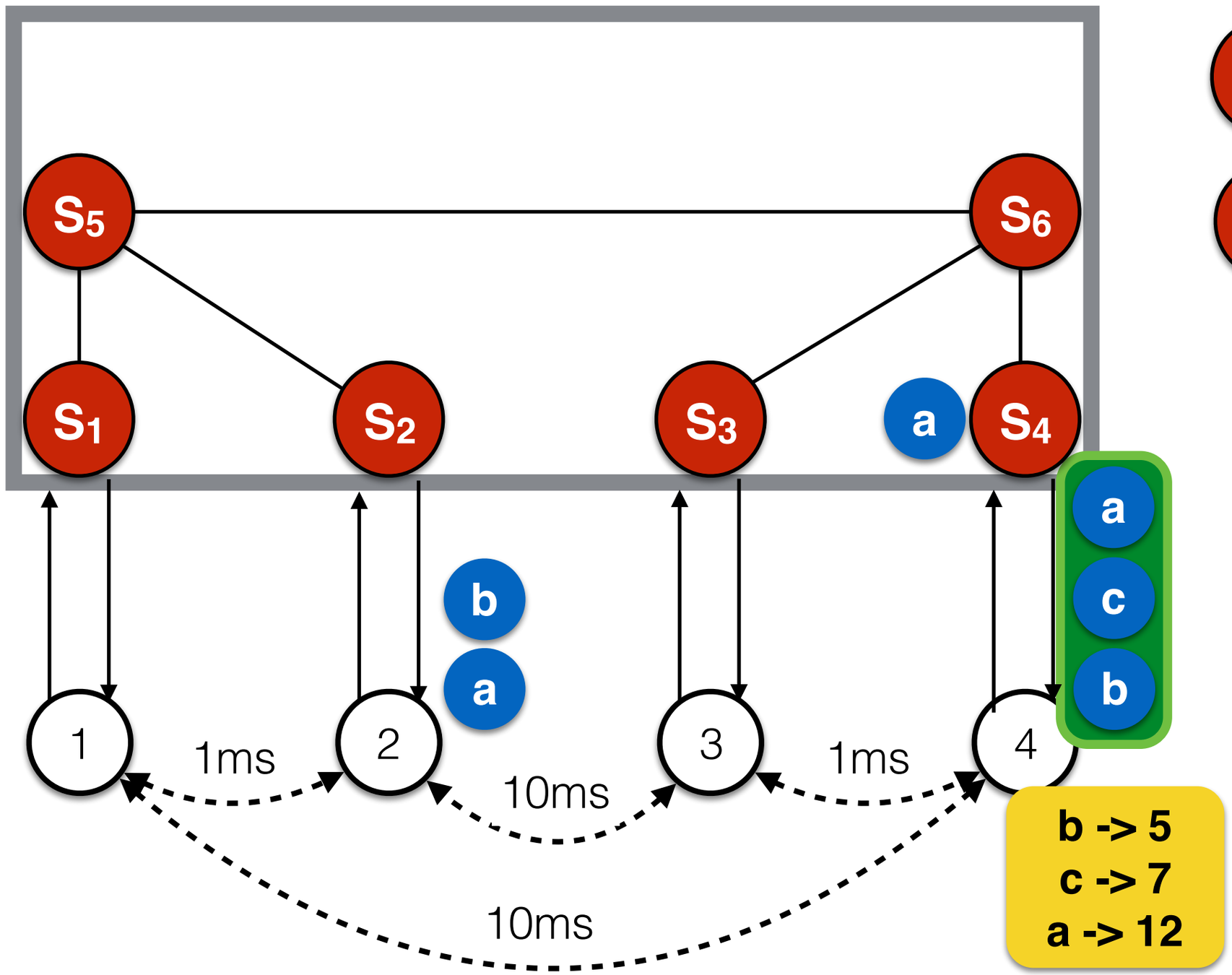
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7 …12 …14



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

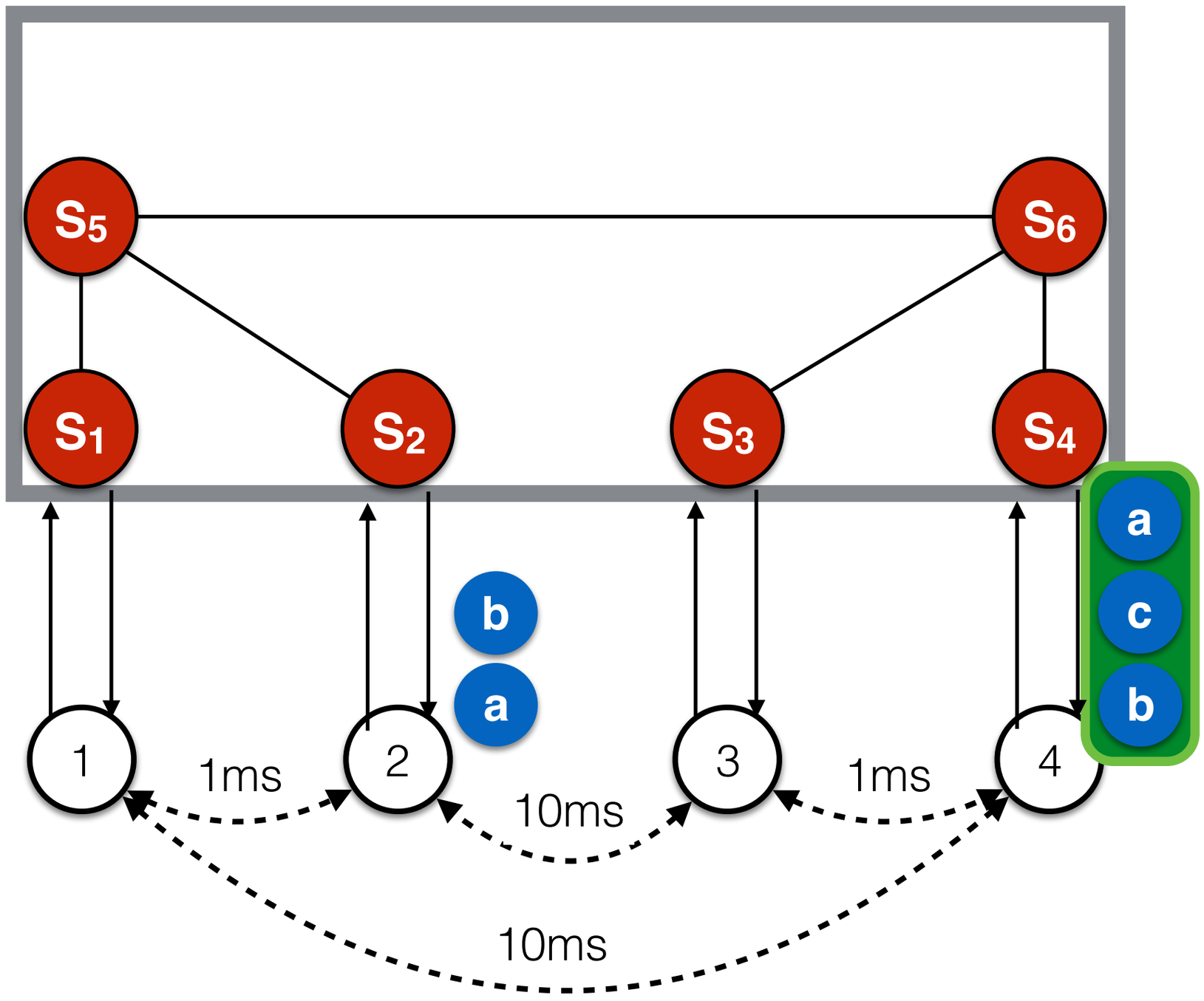
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7 …12 …14



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

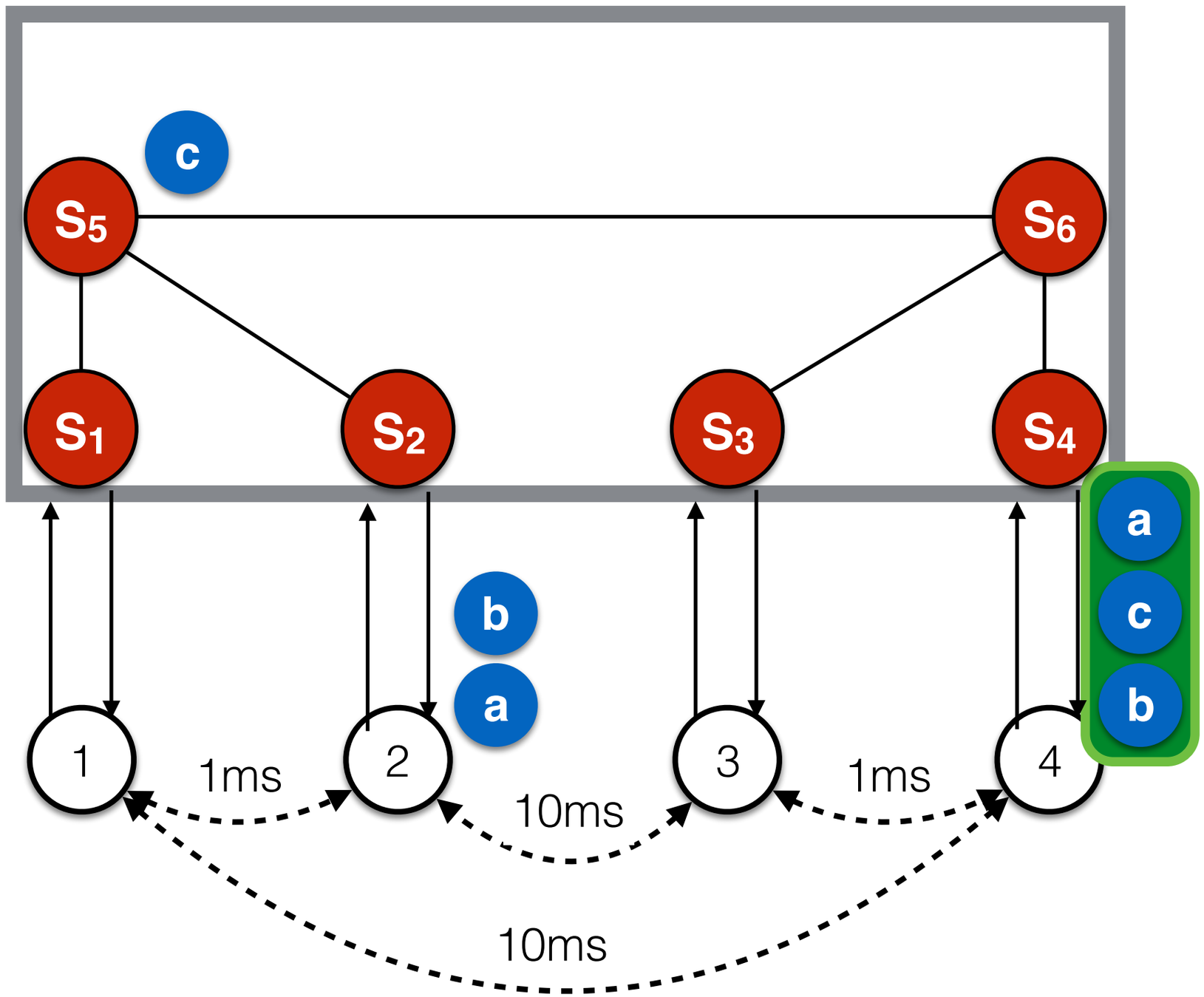
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7 …12 …14…16



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

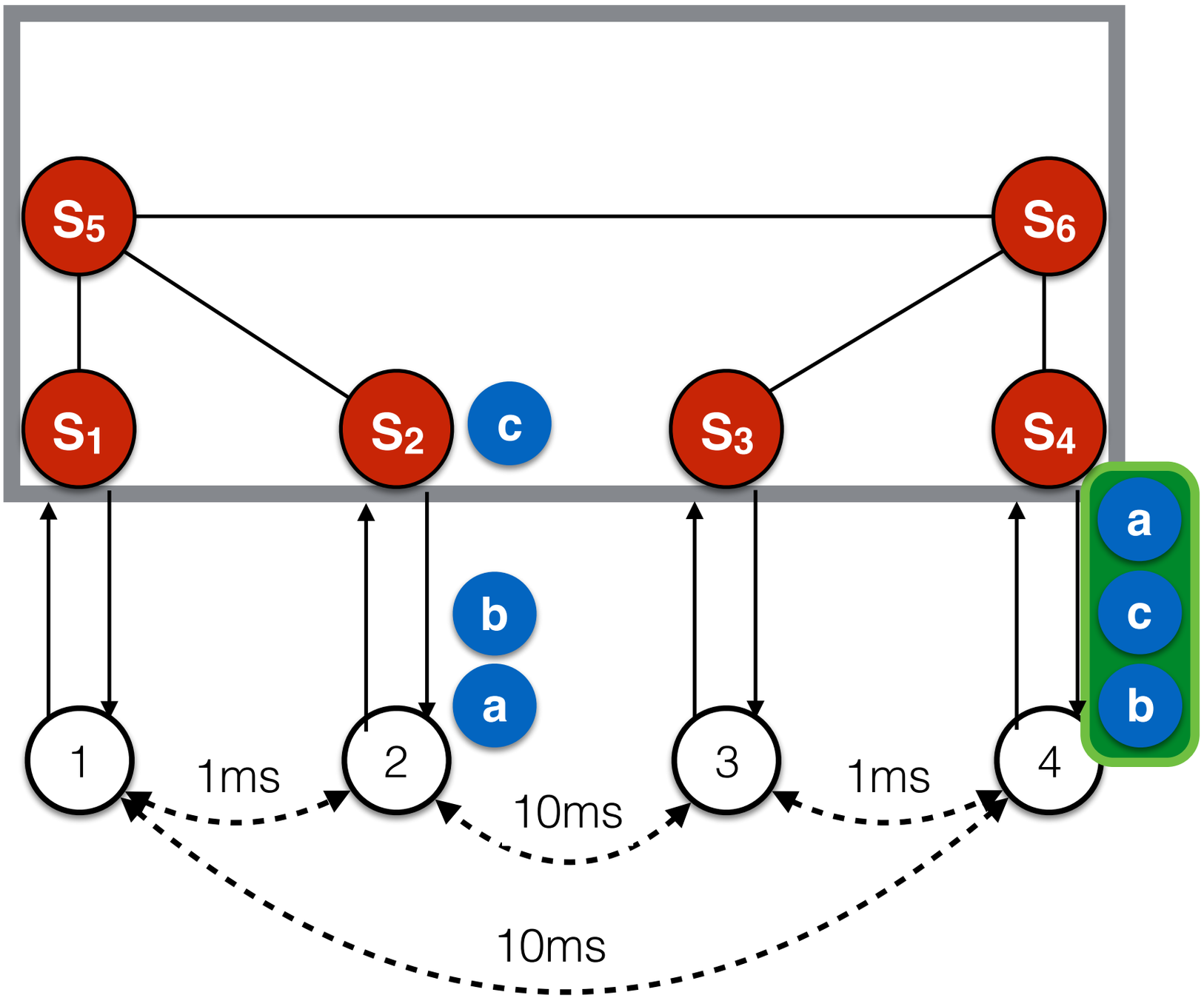
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7 …12 …14…16



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

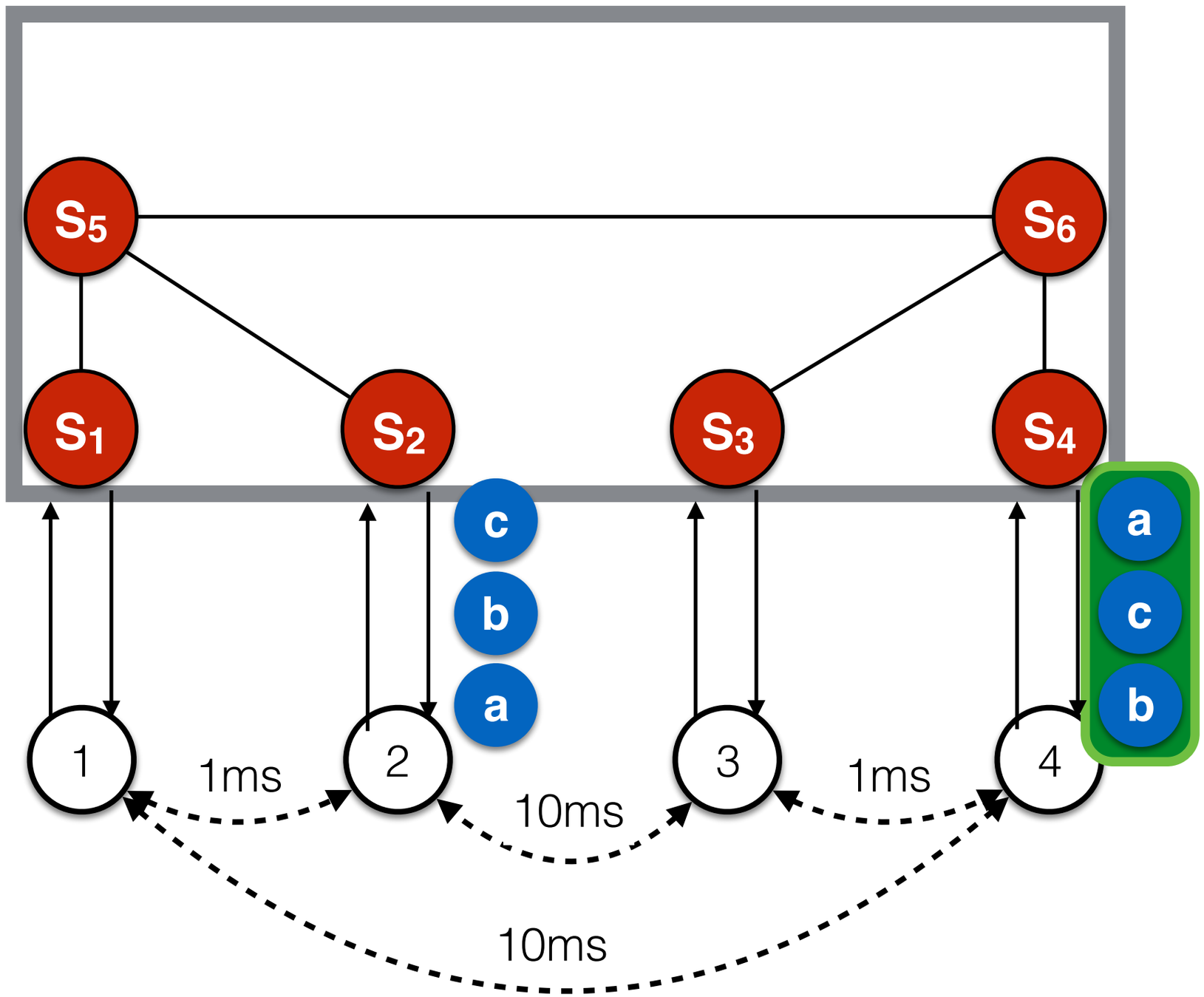
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7 …12 …14…16



S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

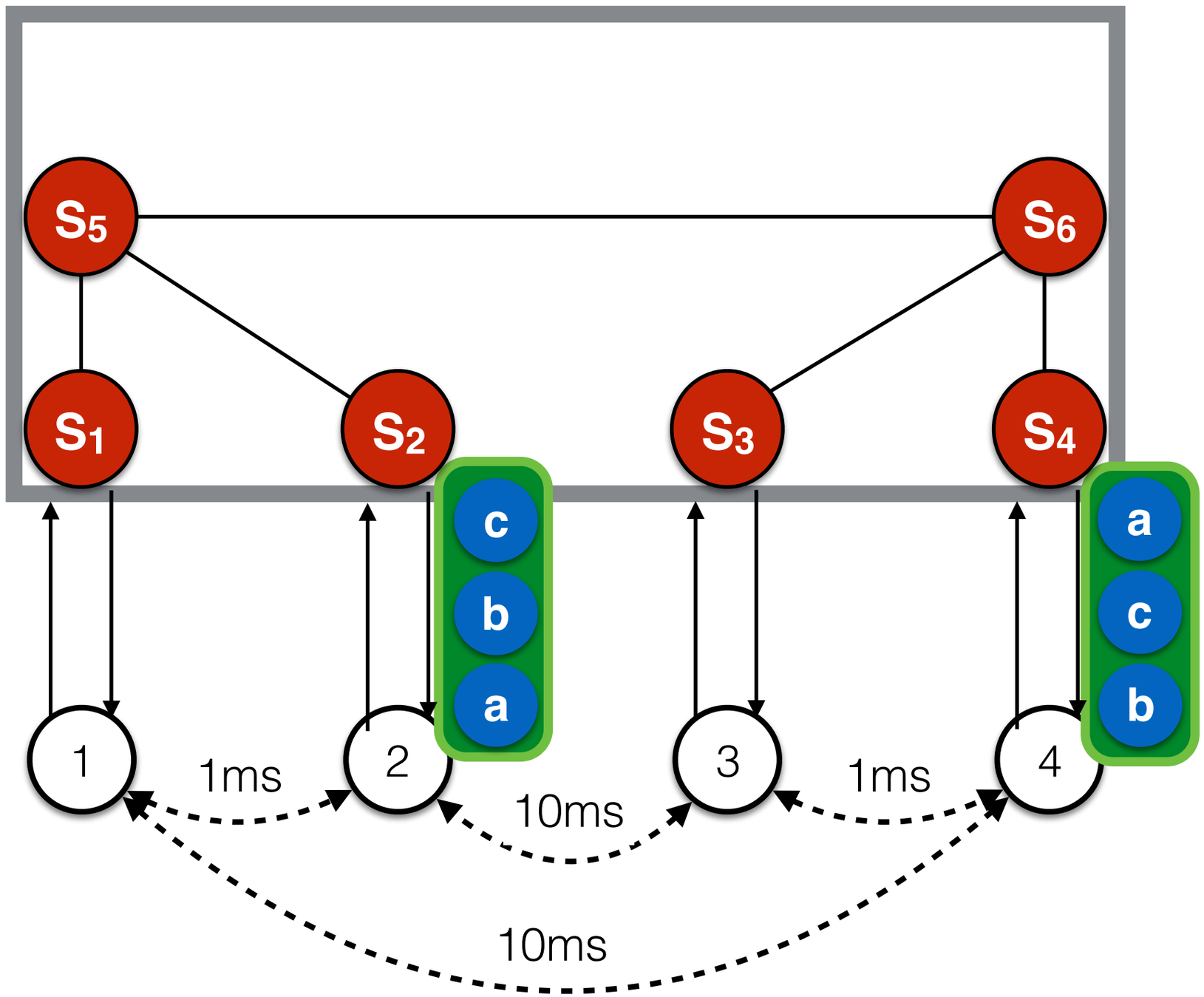
Example

a → b, a → c

c causally depends on b

a concurrent to both b and c

Time: 0 2 3 4 5 6 7 …12 …14…16

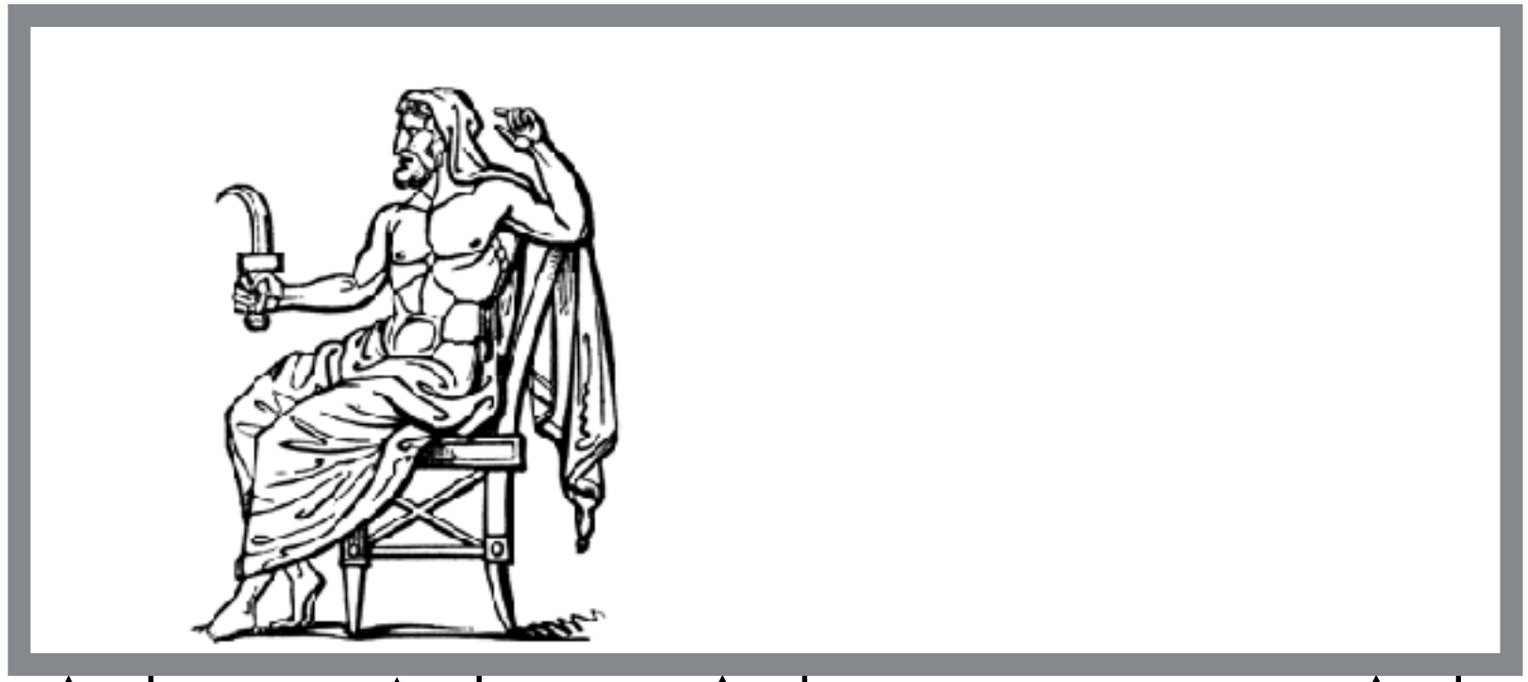


S5 @ dc1 (or dc2)

S6 @ dc4 (or dc3)

data

labels



Client 1 — put(a_1) → 1 — a_1

2

3

…

N

data centers

data

labels

Saturn requires each datacenter to output a SERIAL sequence of updates

Client 1 — put(a_1) → 1 — a_1

2

3

. . .

N

data centers

Eunomia

each datacenter has **many** machines

how to serialize updates performed on different machines **without loosing** performance?

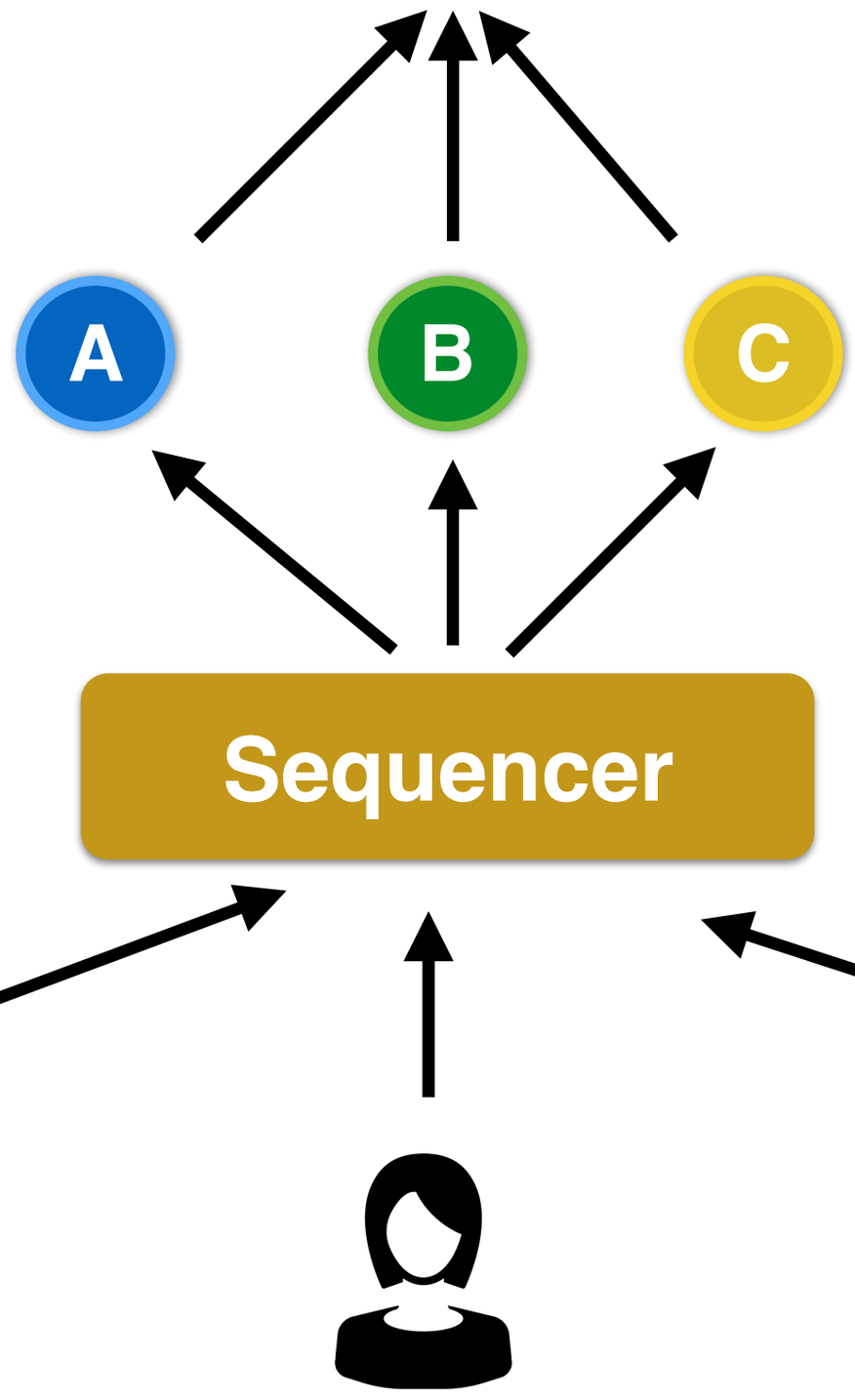
usually one relies on **sequencers** that limit concurrency

Eunomia

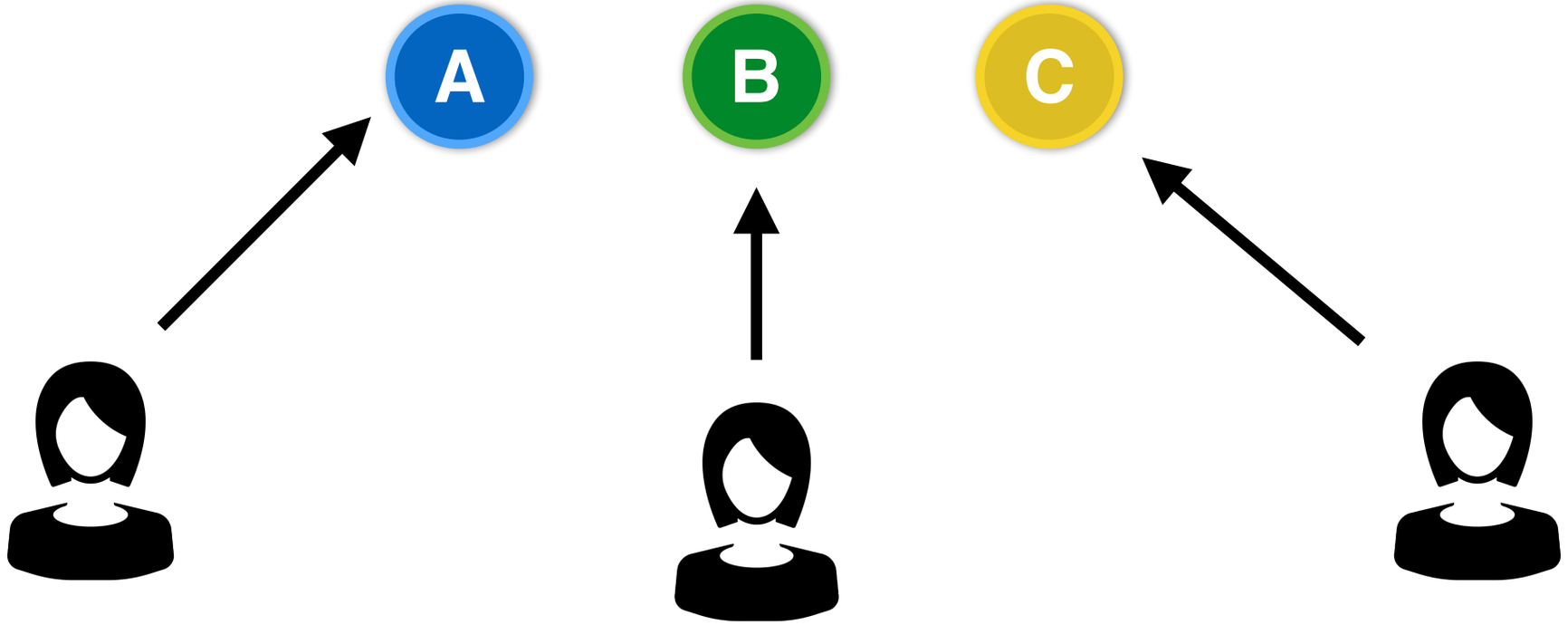
conceived to **replace sequencers** used to serialize updates in replicated storage systems

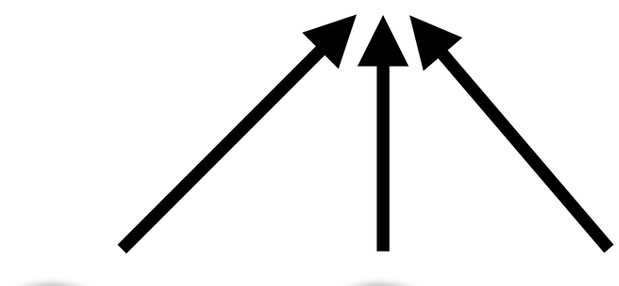
totally orders—consistently with causality—local updates, before shipping them to Saturn and other dcs

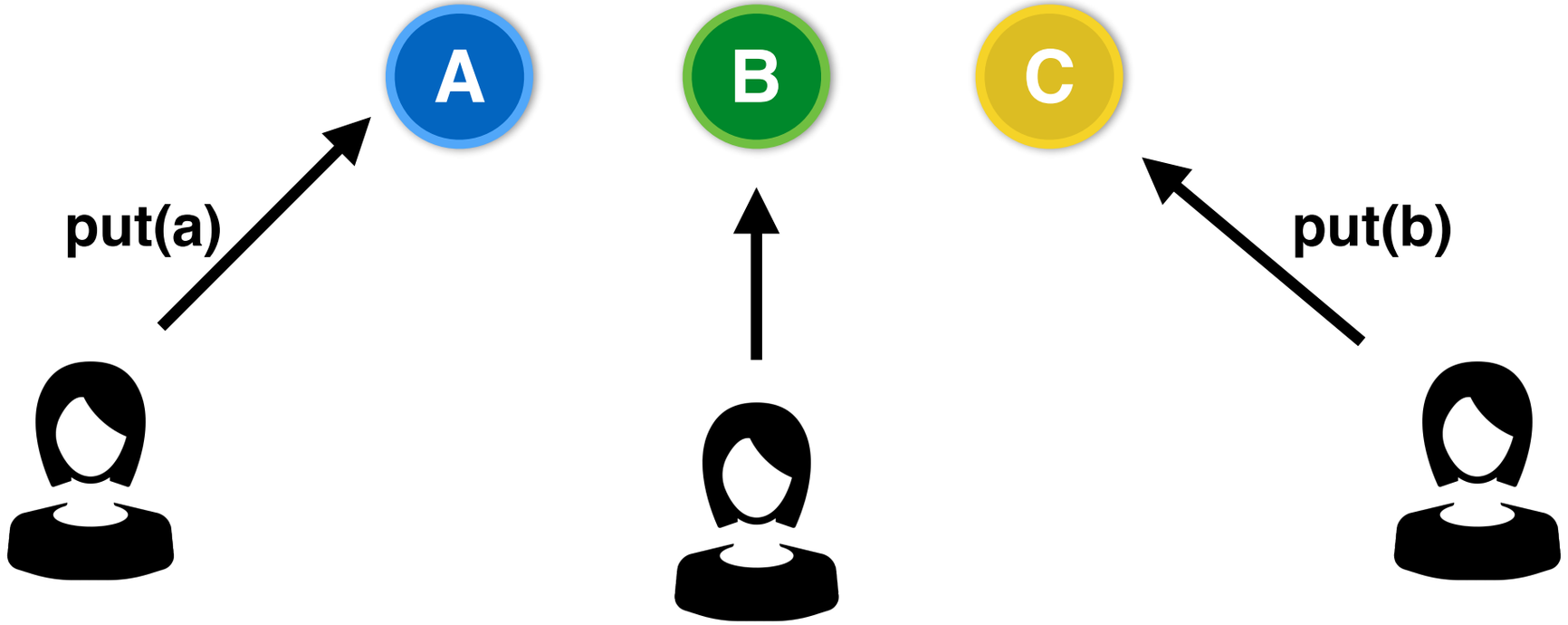
the ordering is done **in the background**, out of client's critical path

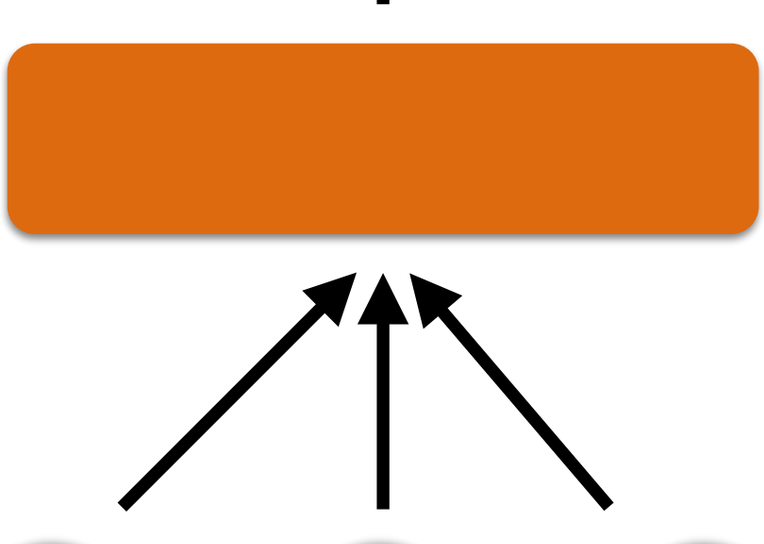

A
B
C
Sequencer

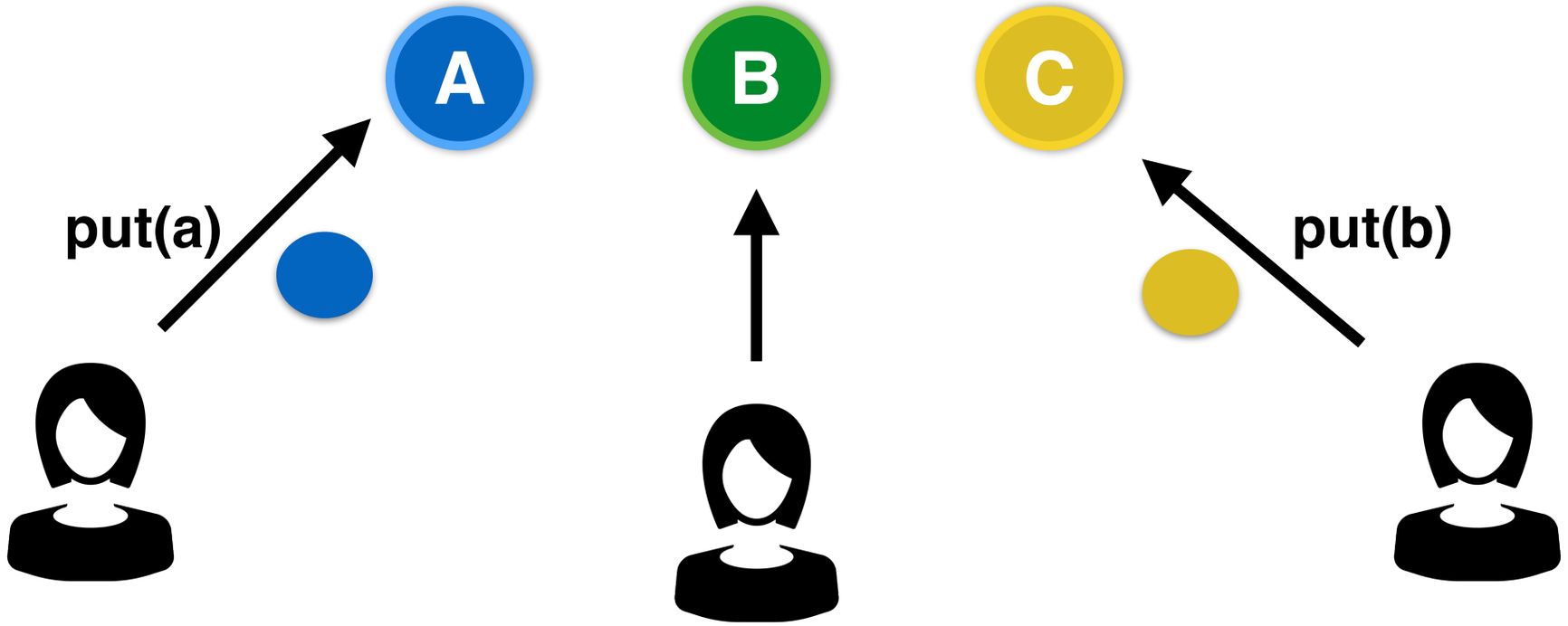
Eunomia

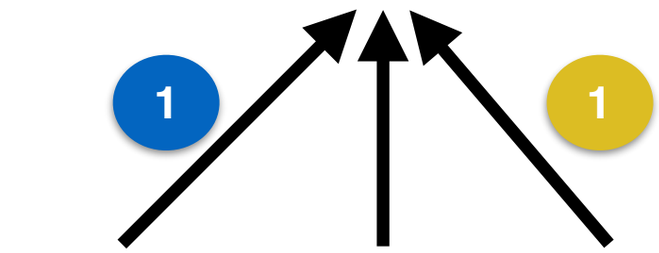


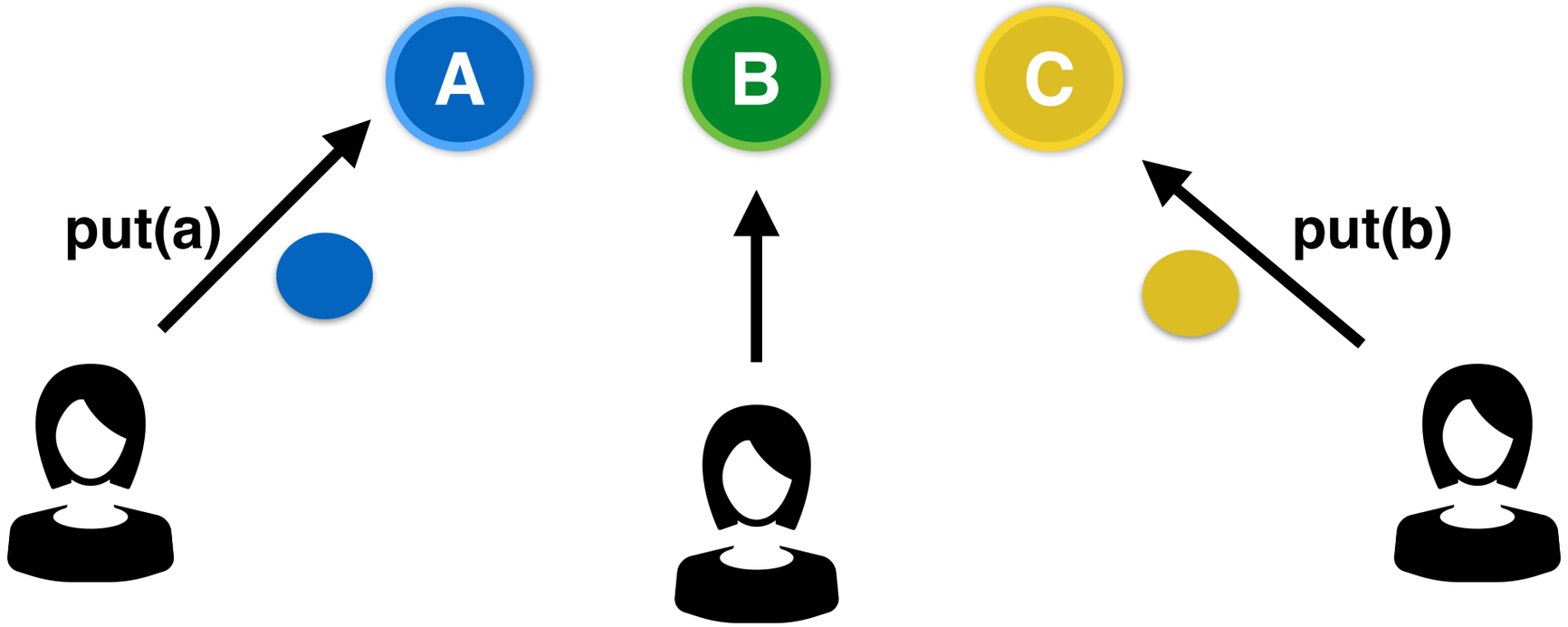



A
B
C
put(a)
put(b)

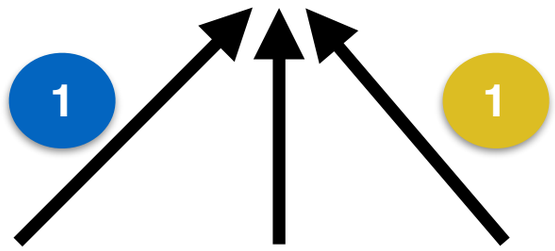


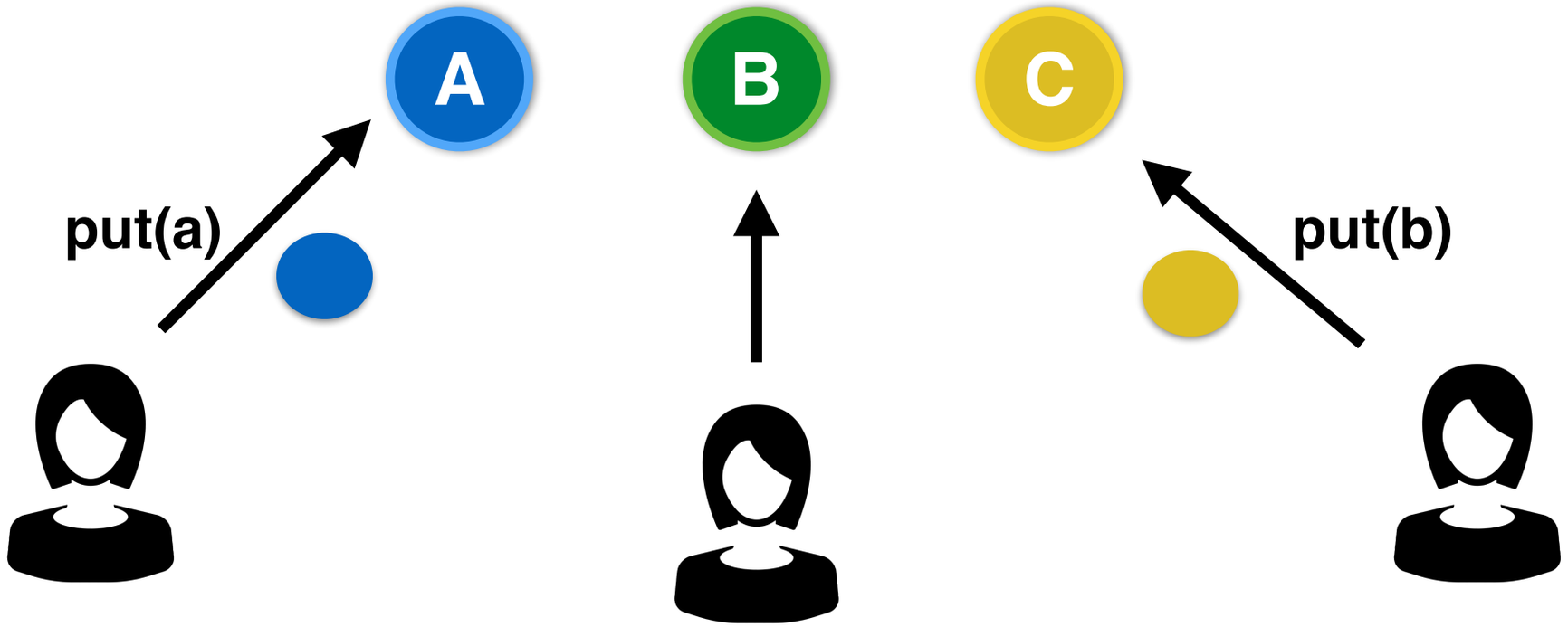

A
B
C
put(a)
put(b)


1
1

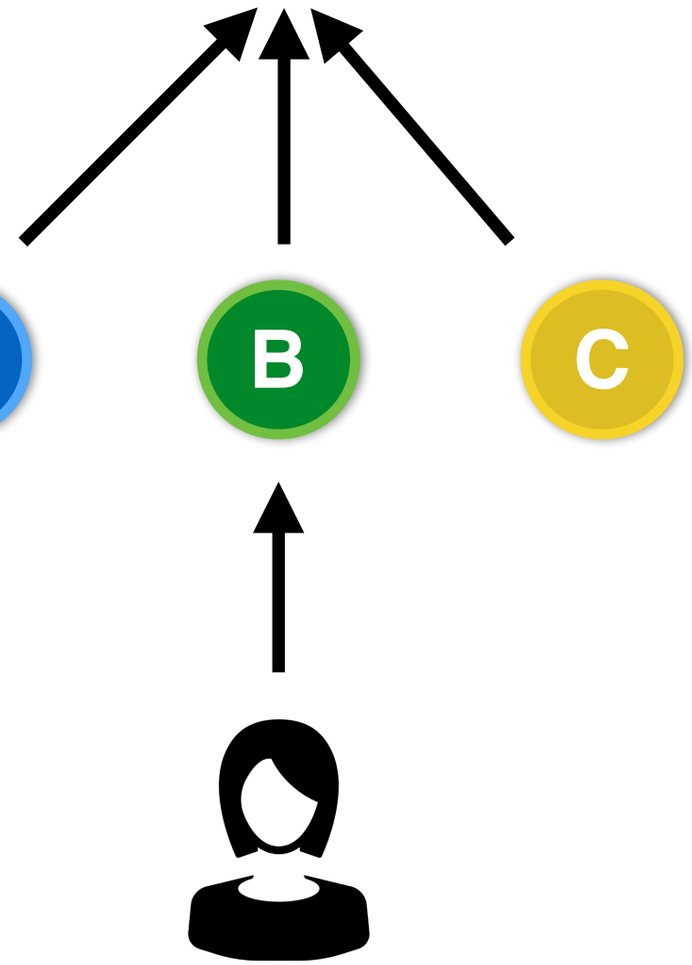

A
B
C
put(a)
put(b)

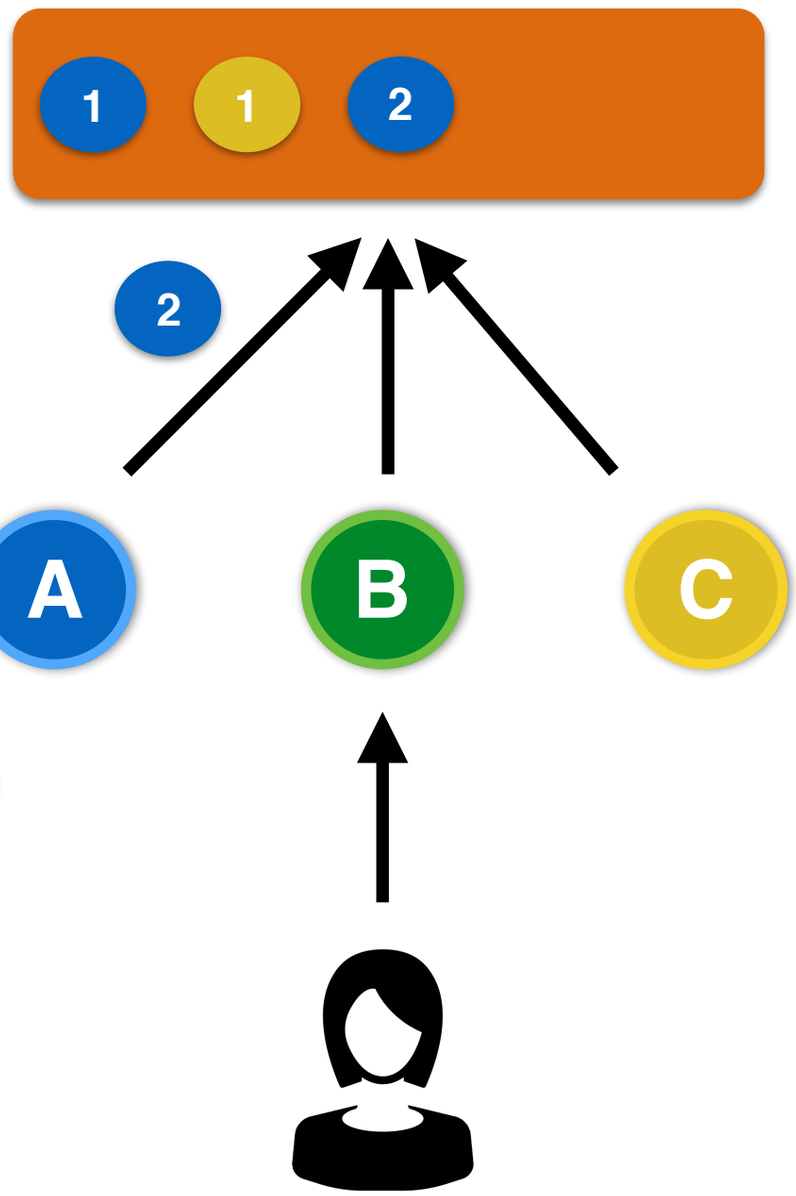

1
1


1
1

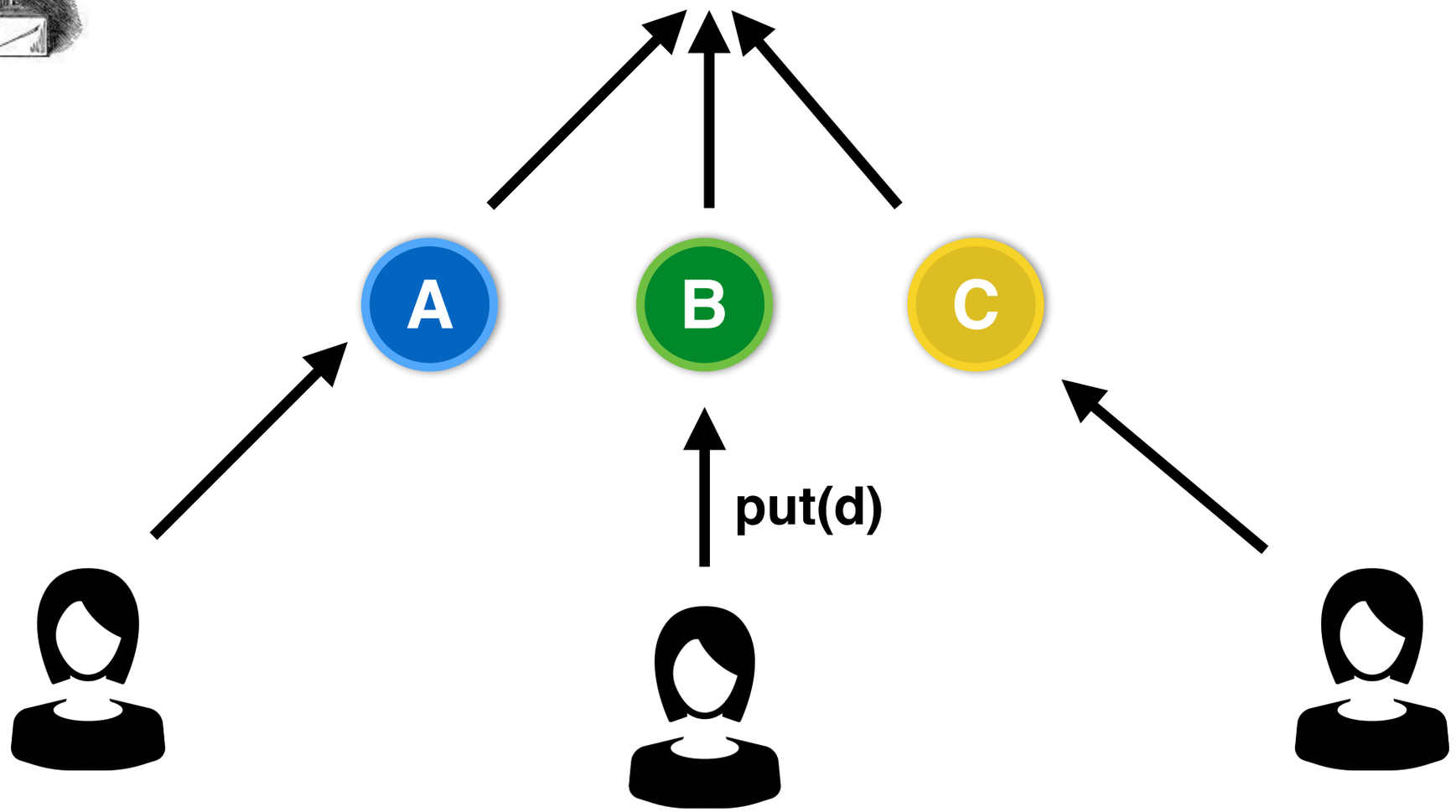

A
B
C
put(a)
put(b)


1
1

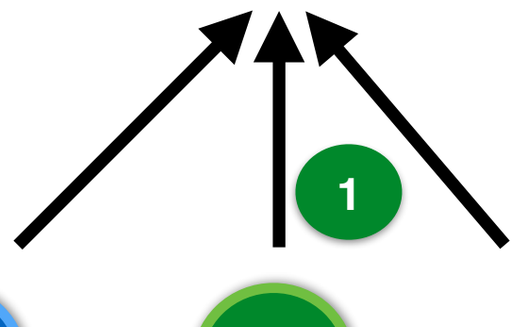

A
B
C
put(c)

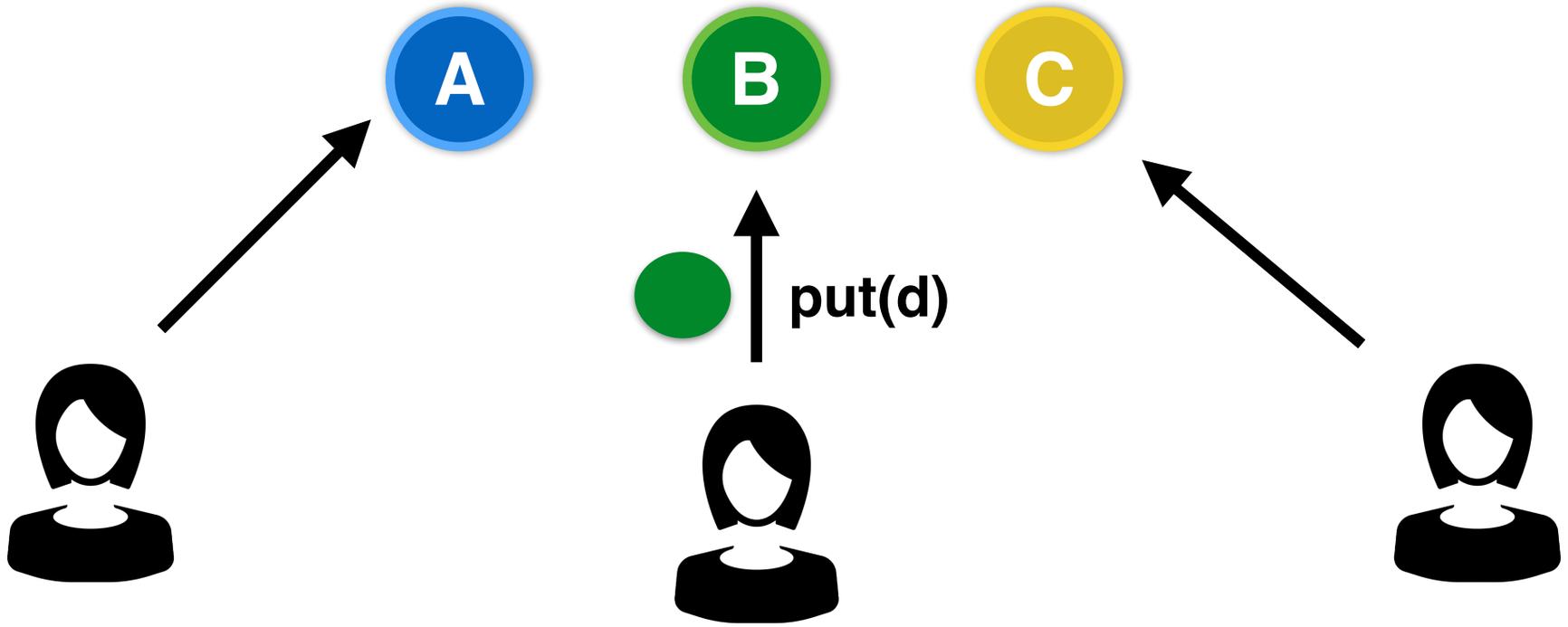

1
1
2
2
A
B
C
put(c)


1
1
2

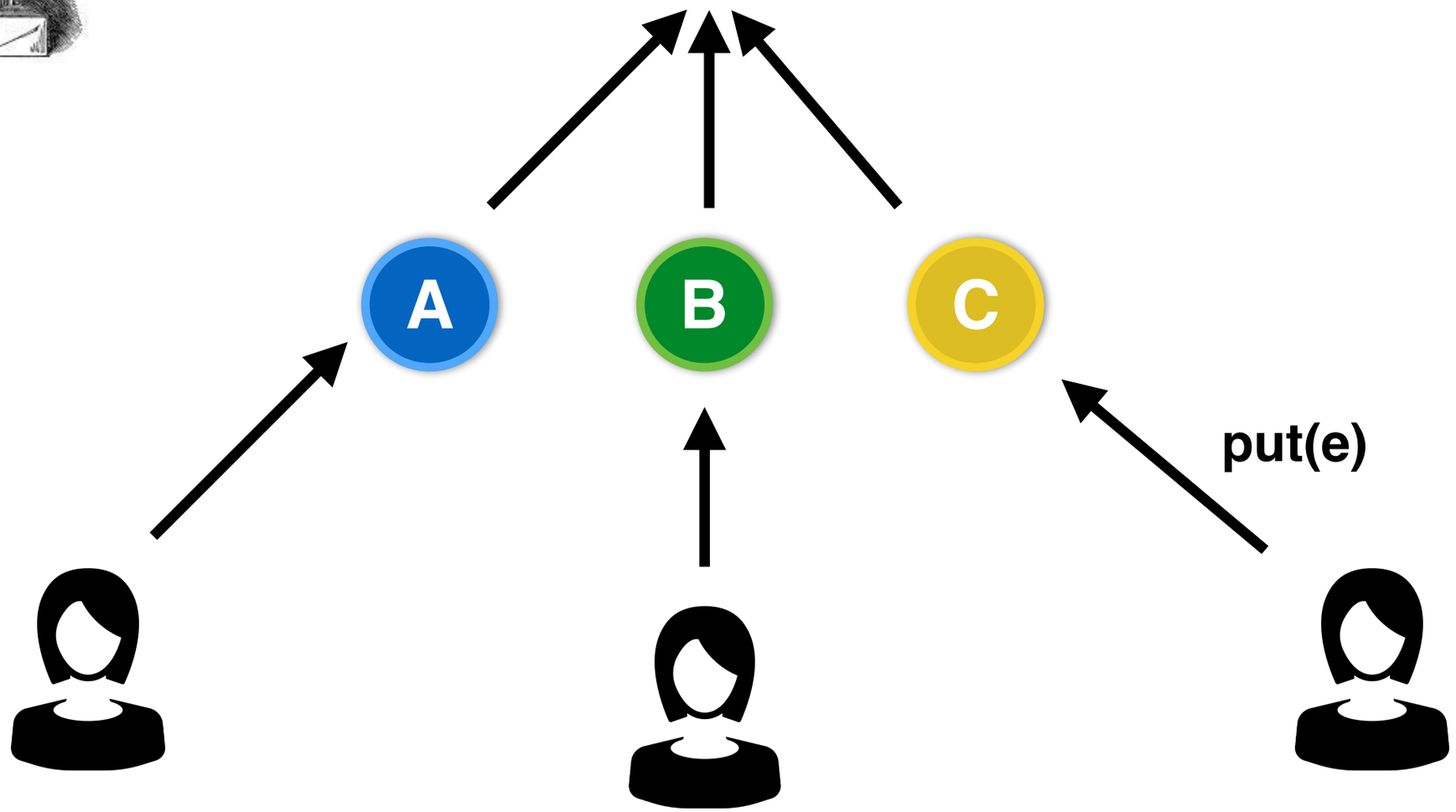

A
B
C
put(d)

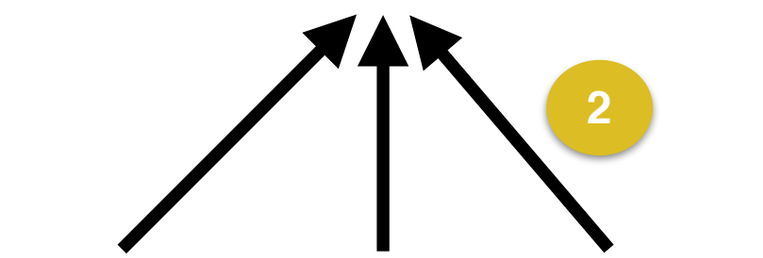

1
1
2
1

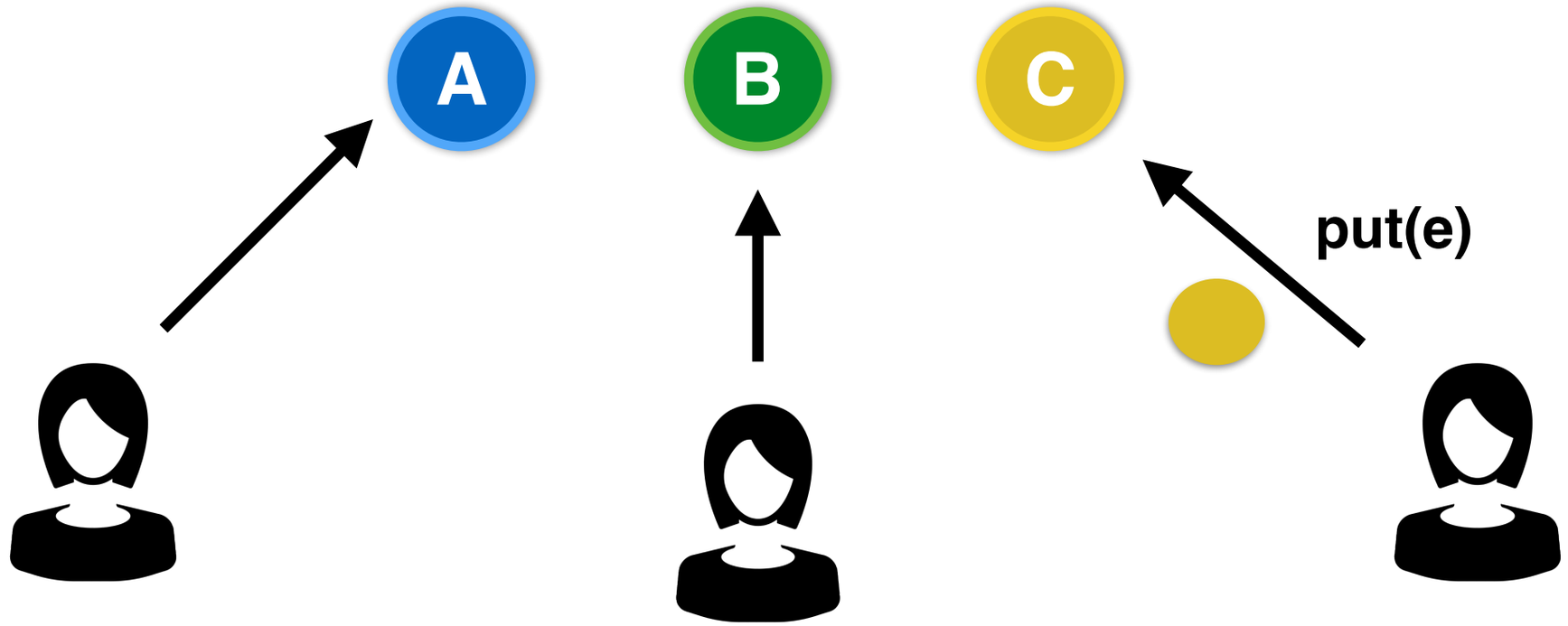

1


A
B
C
put(d)

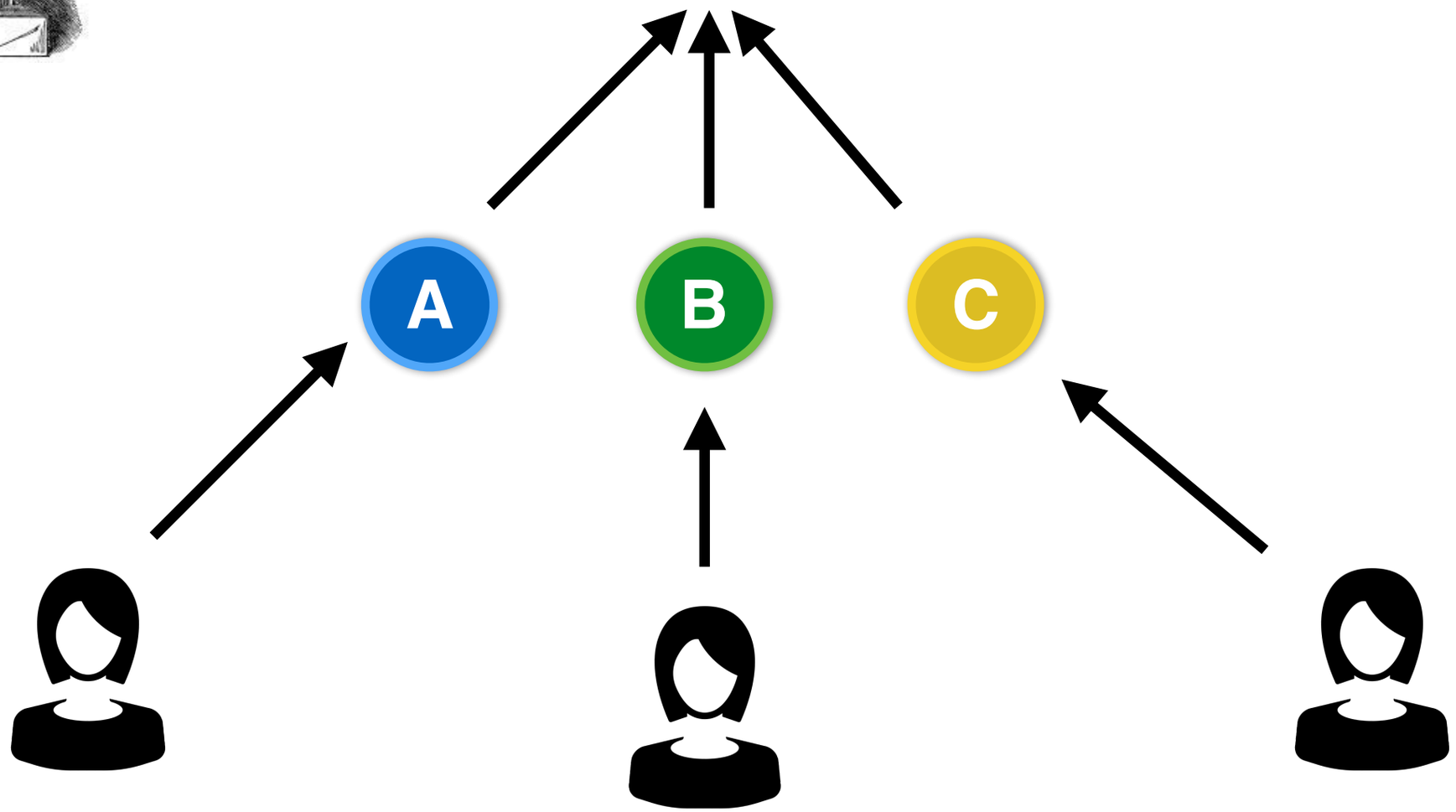

1
1
2
1

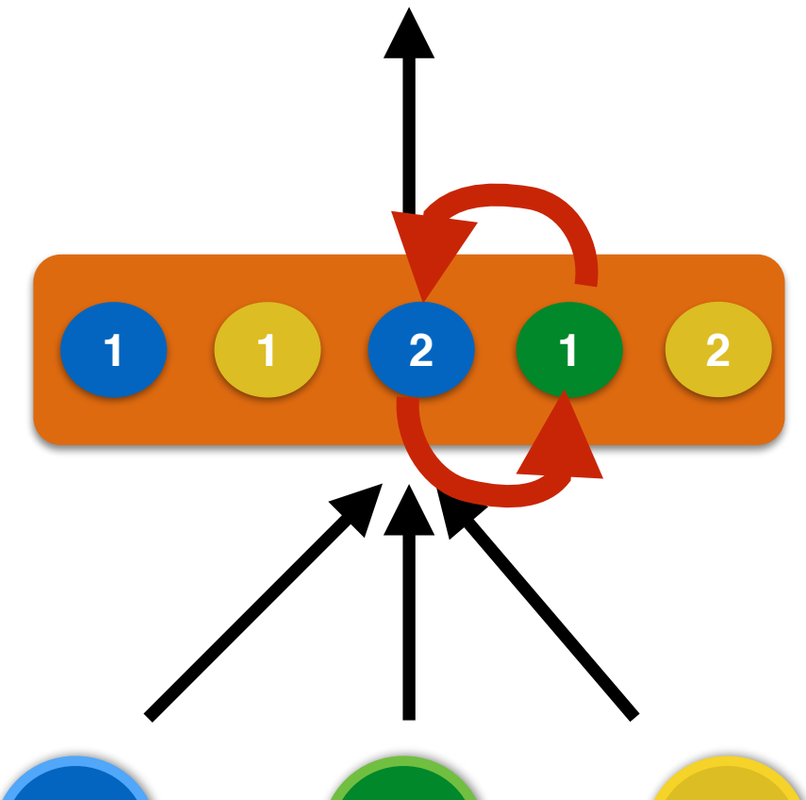

A
B
C
put(e)

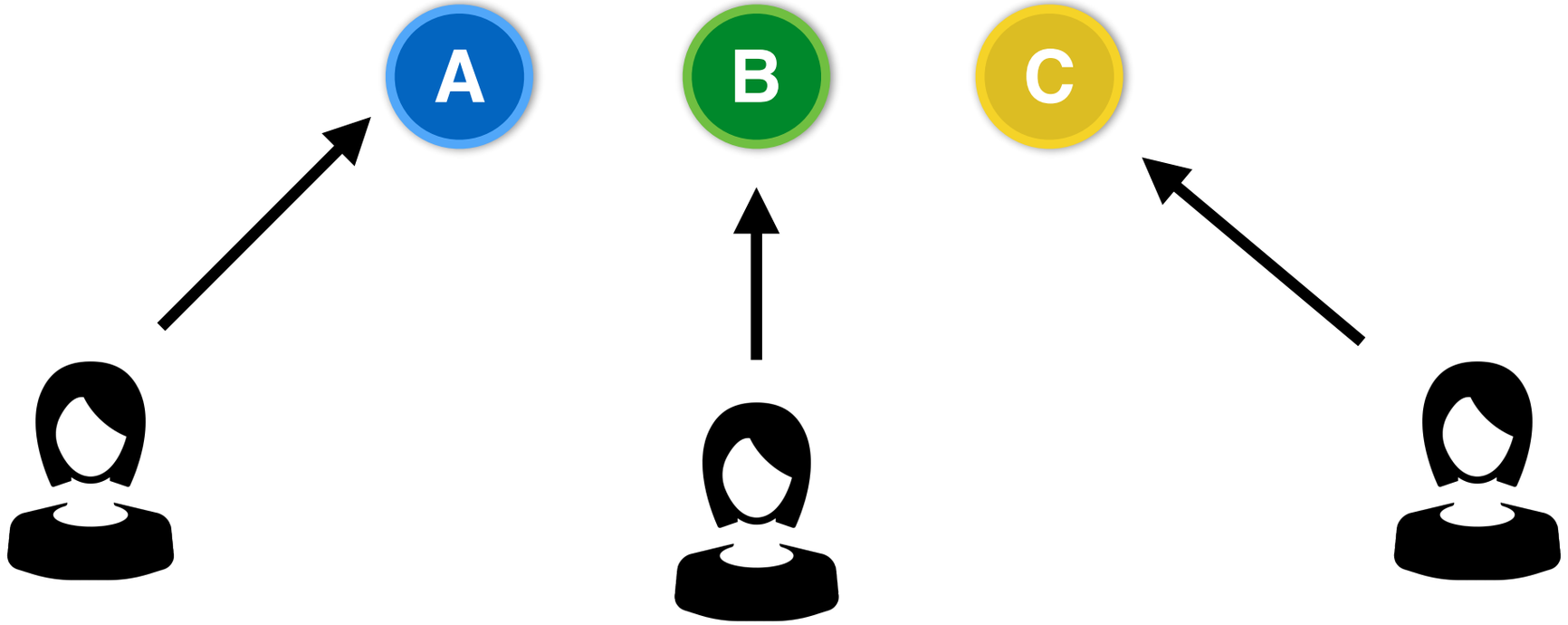

1
1
2
1
2
2

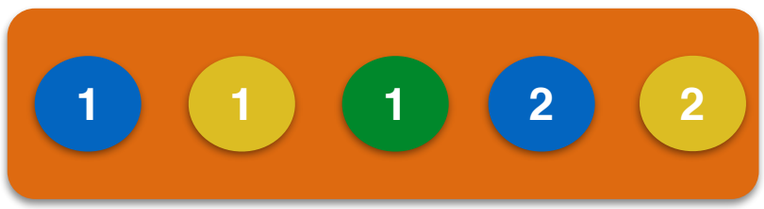

A
B
C
put(e)

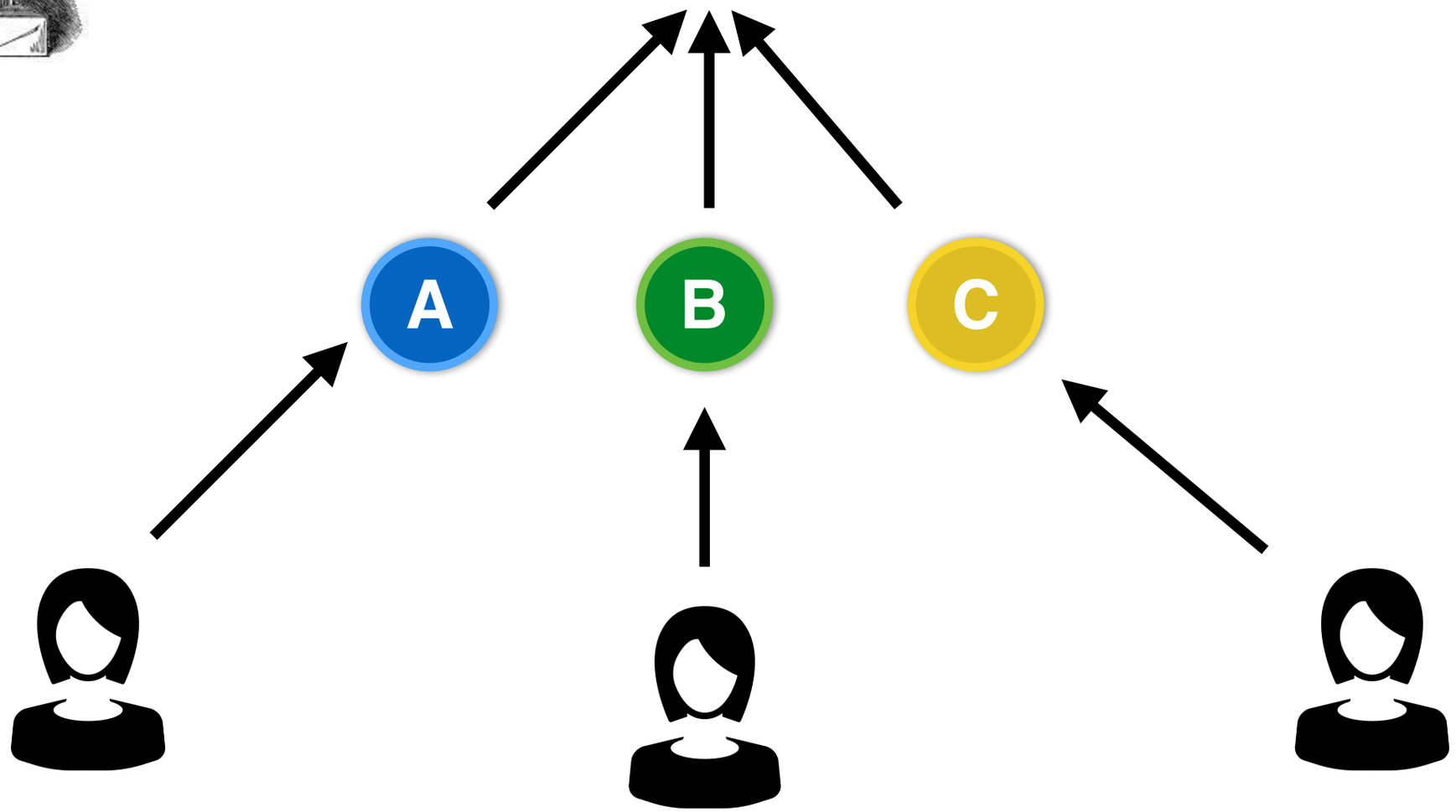

1
1
2
1
2

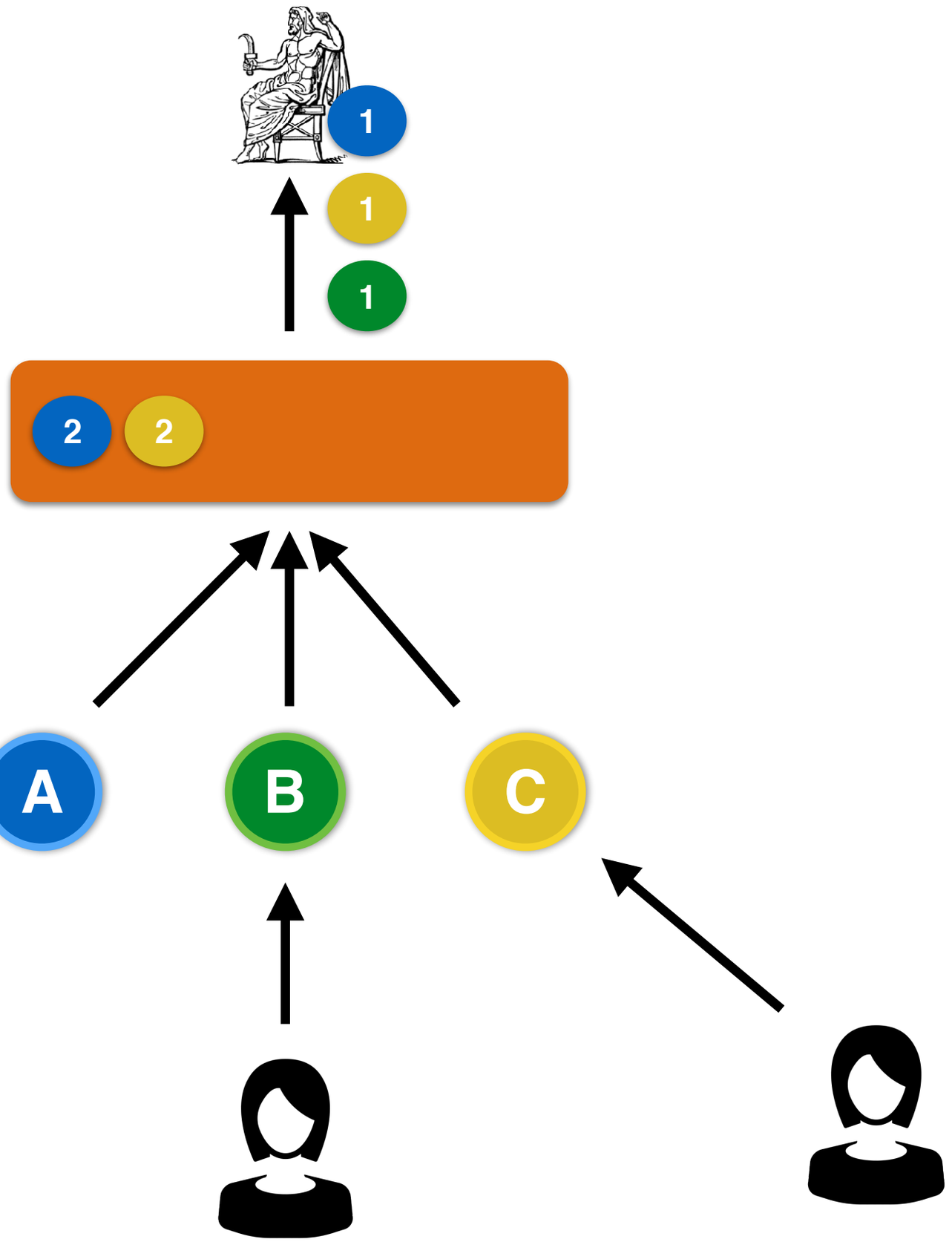

A
B
C


1
1
2
1
2


A
B
C


1
1
1
2
2


A
B
C


1
1
1
2
2
A
B
C

2

2

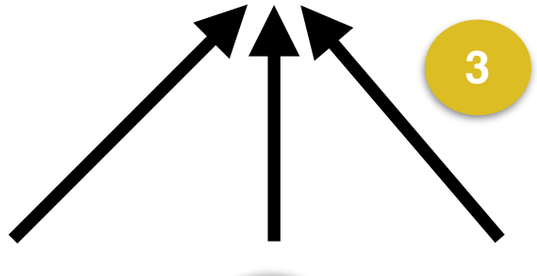
A

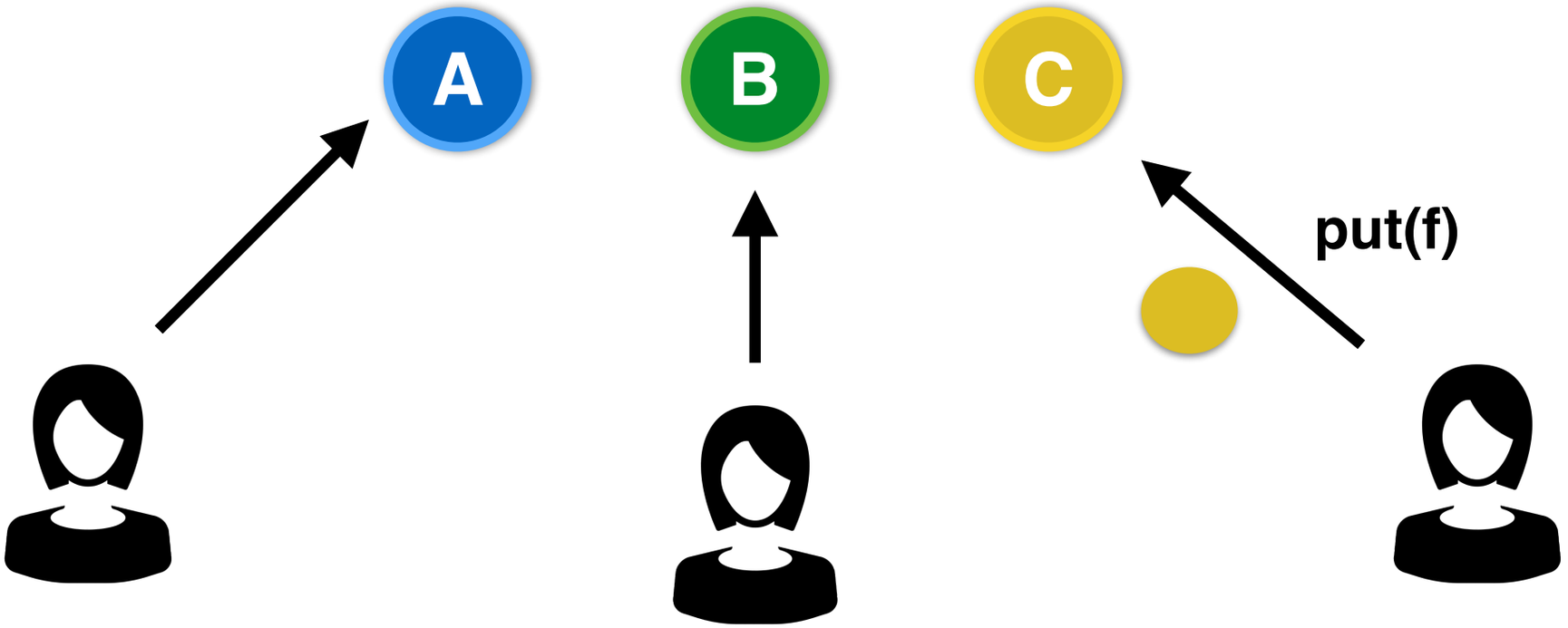
B

C

put(f)


2
2
3


3


A
B
C
put(f)

Evaluation

+

Evaluation: goals

- Can **Saturn+Eunomia** optimize both throughput and data freshness simultaneously?
- Why **Saturn's** genuine partial replication matters?
- How important is the configuration of **Saturn**?
- How **Eunomia** compares to sequencers?

Evaluation: goals

- Can **Saturn+Eunomia** optimize both throughput and data freshness simultaneously?

- Why **Saturn's** genuine partial replication matters?

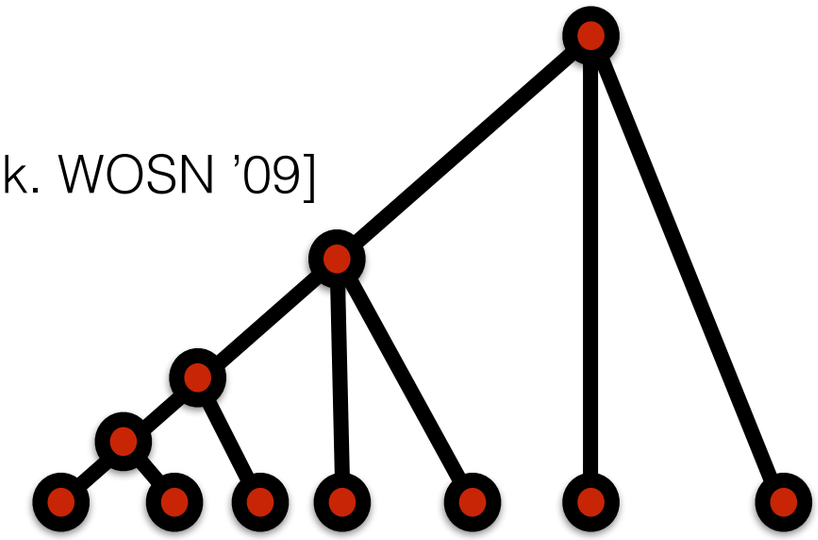
- How important is the configuration of **Saturn**?

- How **Eunomia** compares to sequencers?

Saturn vs state-of-the-art

Public Facebook dataset

[B. Viswanath et al. On the evolution of user interaction in Facebook. WOSN ’09]



Partitioning among 7 data centers

Replication factor min. 2 max. [2,5]

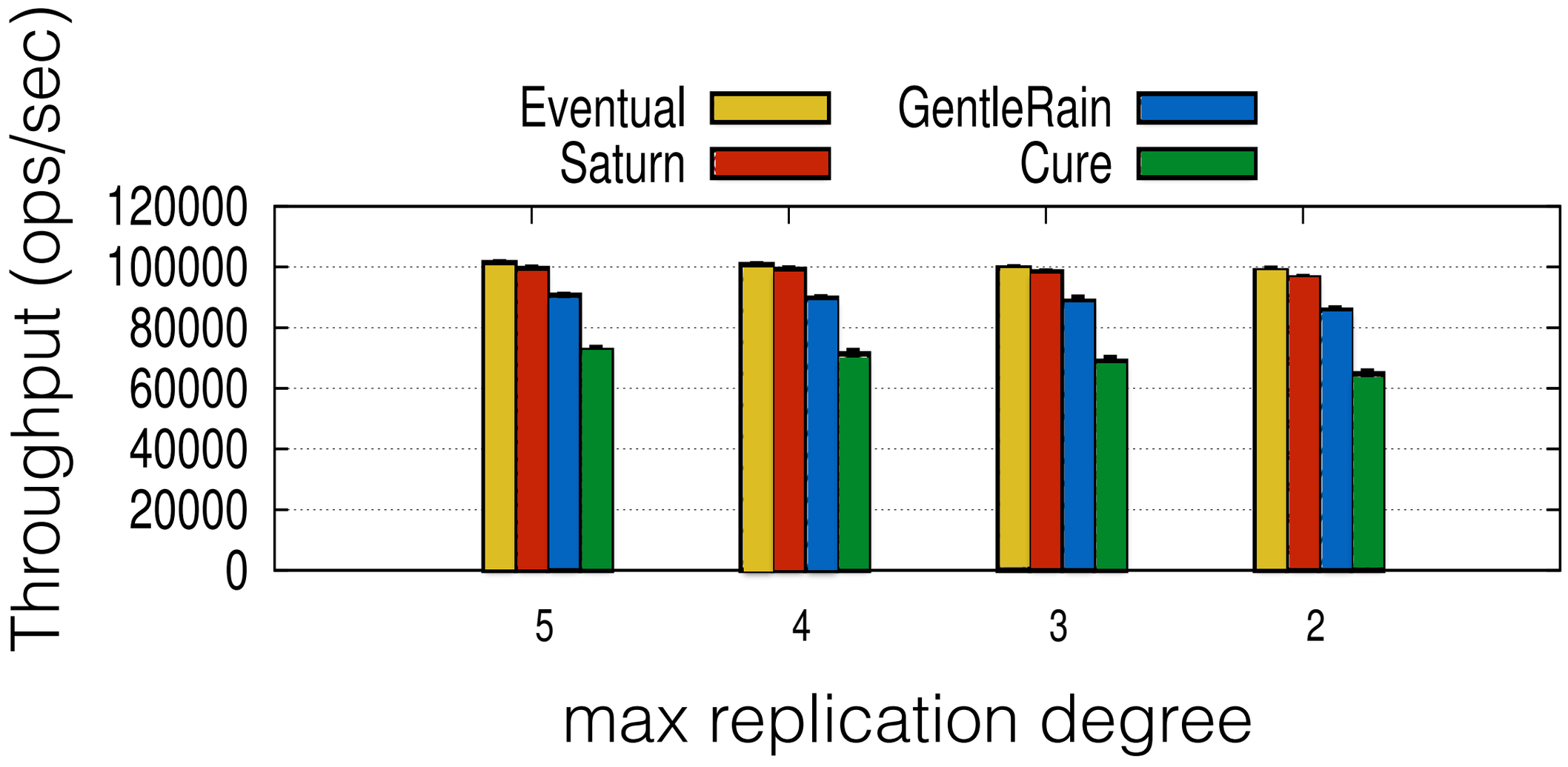
[J. M. Pujol et al. The little engine(s) that could: Scaling online social networks. SIGCOMM ’10]

Workload based on real social network workloads

[F. Benevenuto et al. Characterizing user behavior in online social networks. ICM ’09]

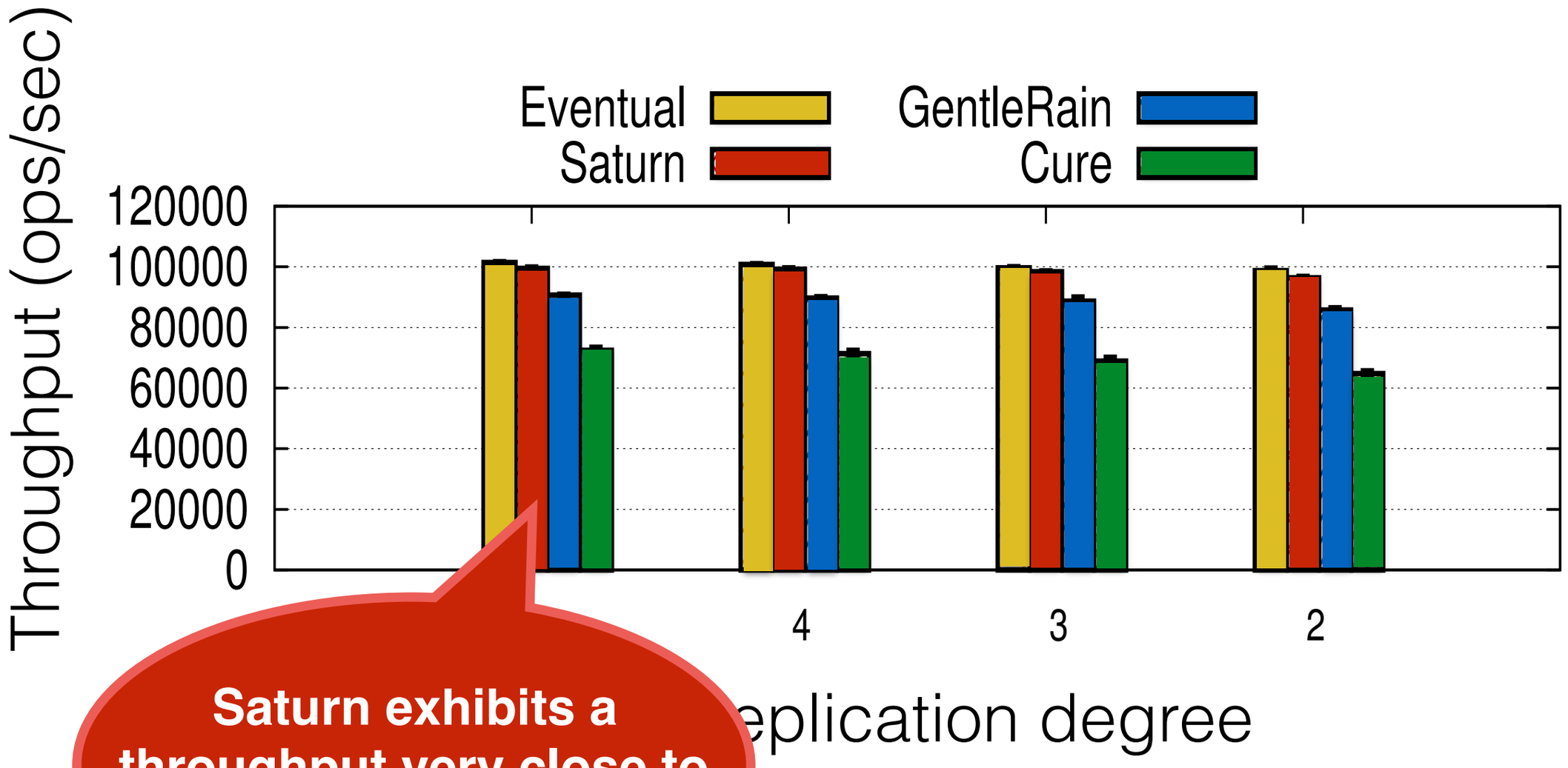
Saturn vs state-of-the-art

Throughput



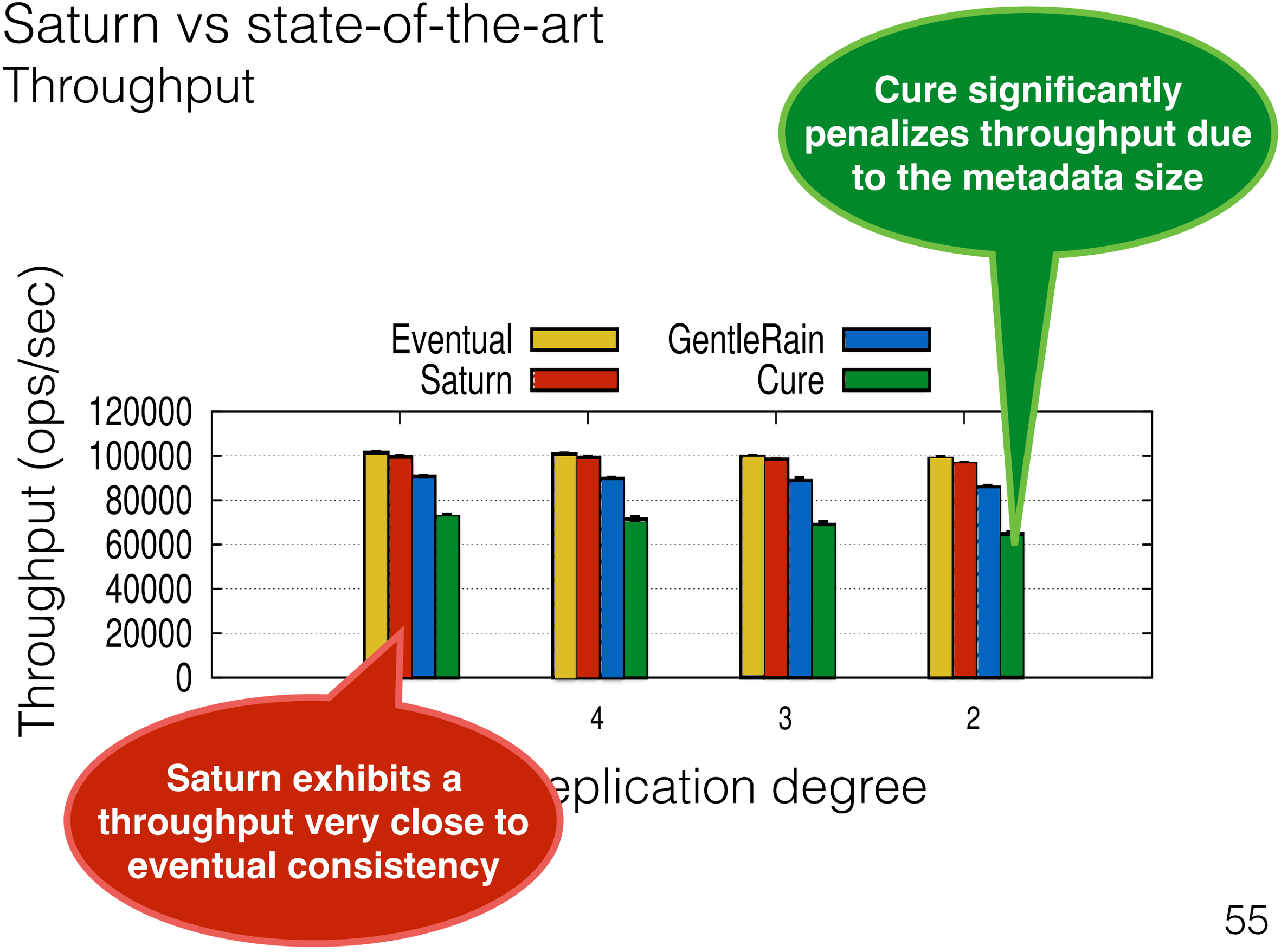
Saturn vs state-of-the-art

Throughput



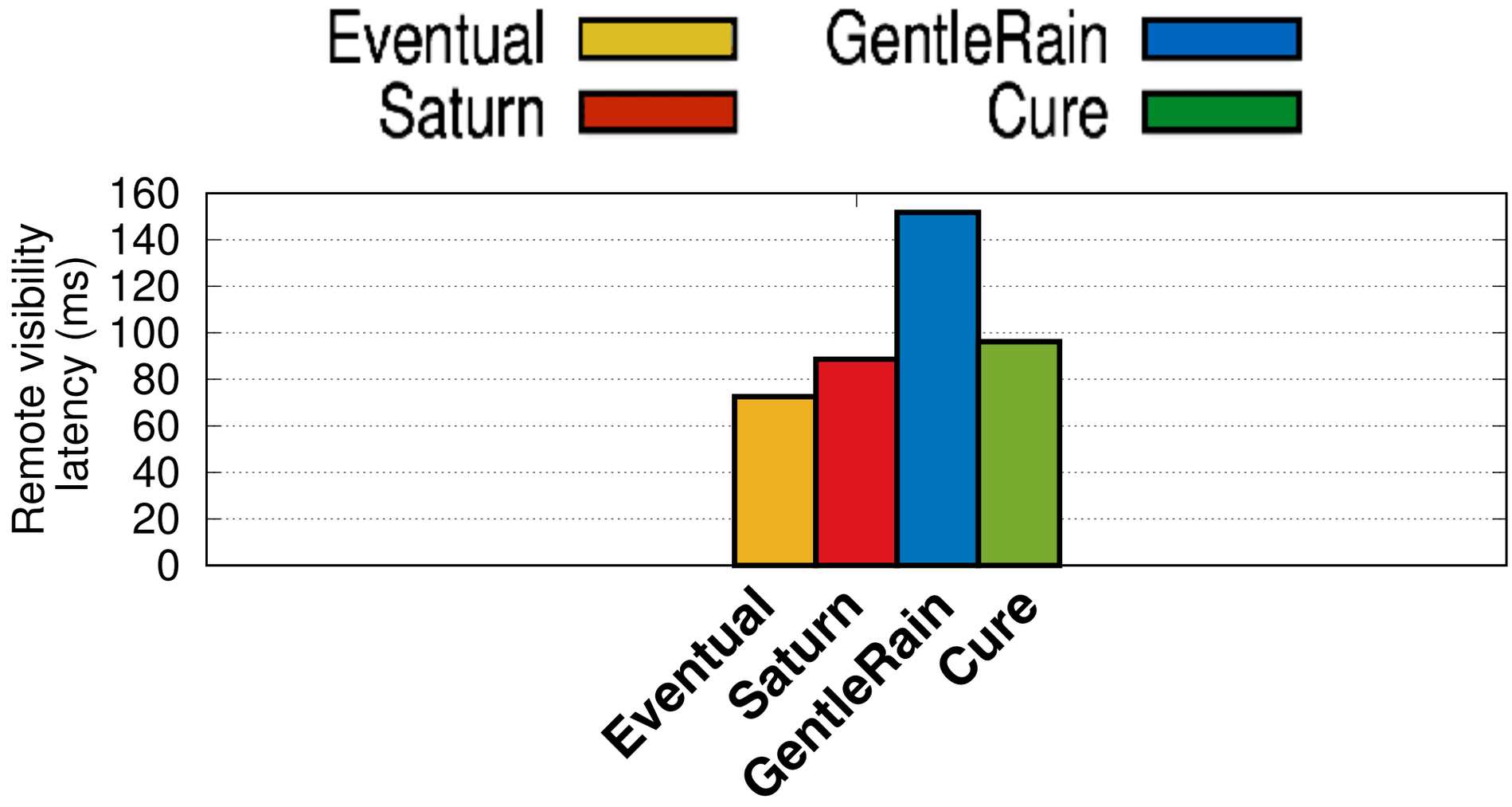
Saturn exhibits a throughput very close to eventual consistency

Saturn vs state-of-the-art
Throughput



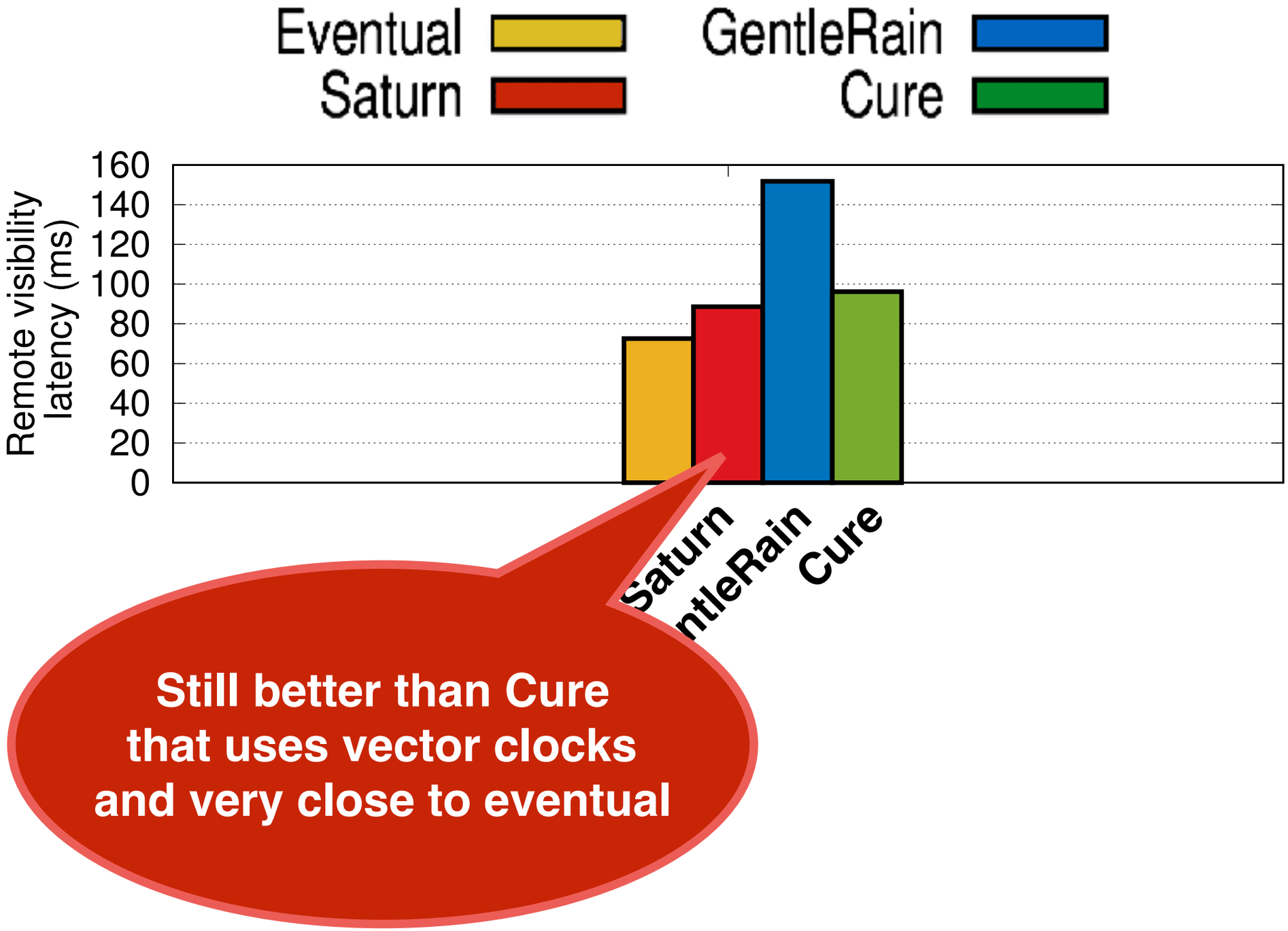
Saturn vs state-of-the-art

Data freshness: average



Saturn vs state-of-the-art

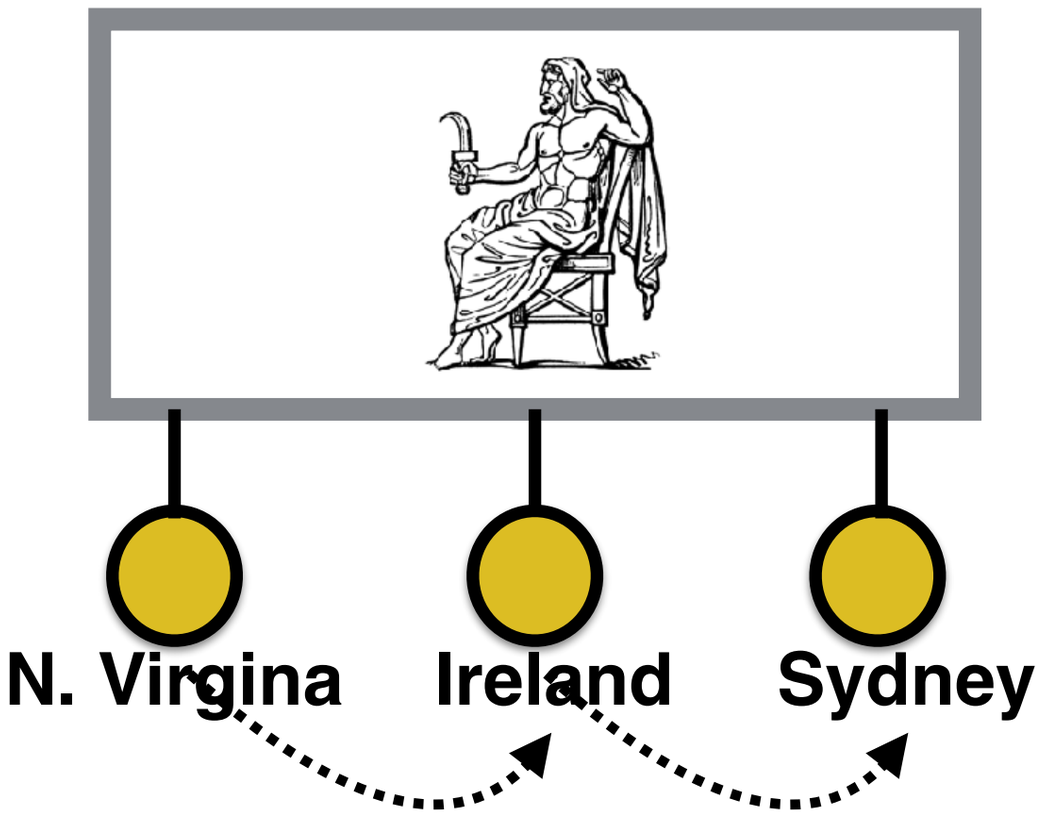
Data freshness: average



Evaluation: goals

- Can **Saturn+Eunomia** optimize both throughput and data freshness simultaneously?
- Why **Saturn's** genuine partial replication matters?
- How important is the configuration of **Saturn**?
- How **Eunomia** compares to sequencers?

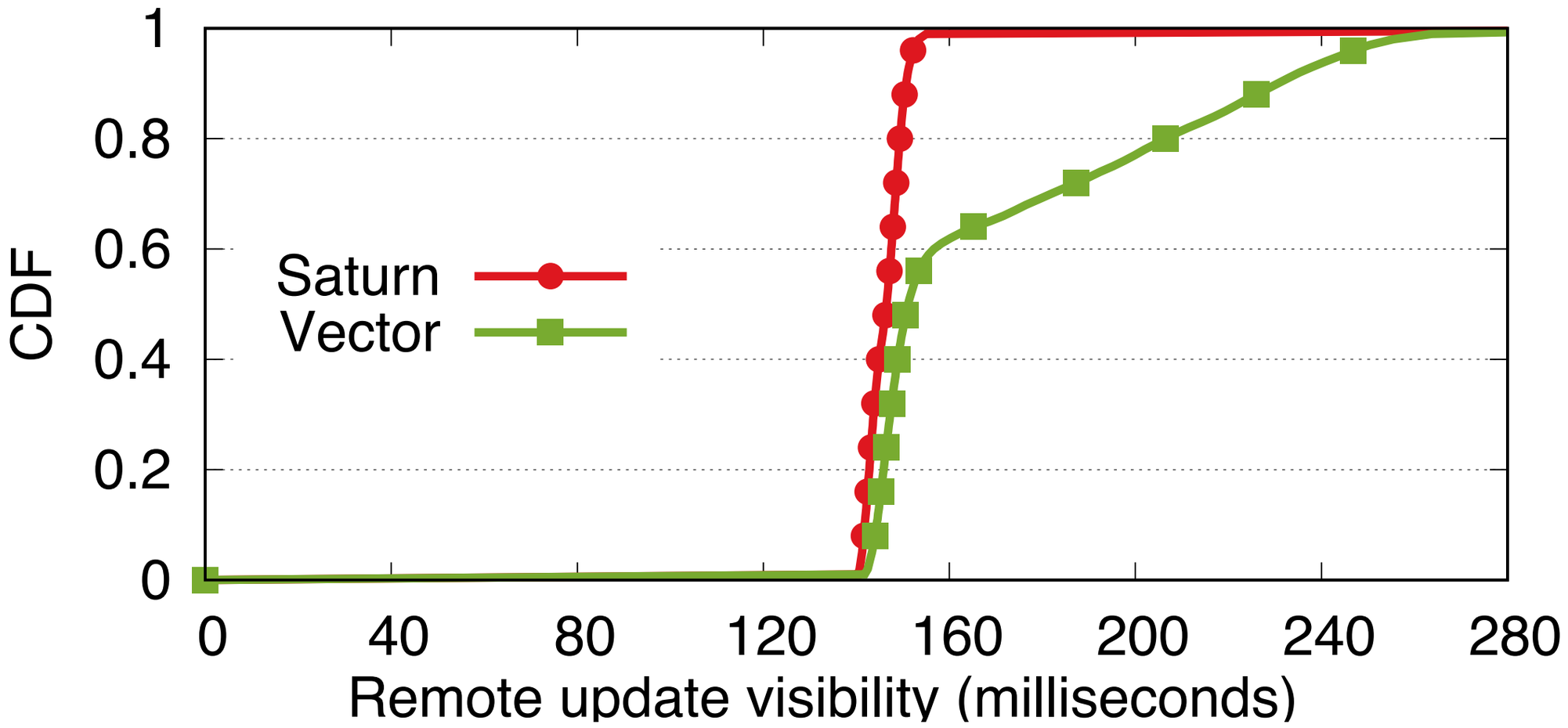
Saturn
Genuine partial replication matters



- Sydney receives updates from Ireland
- Ireland updates depend on N. Virginia updates
- Sydney **does not** replicate all N. Virginia data

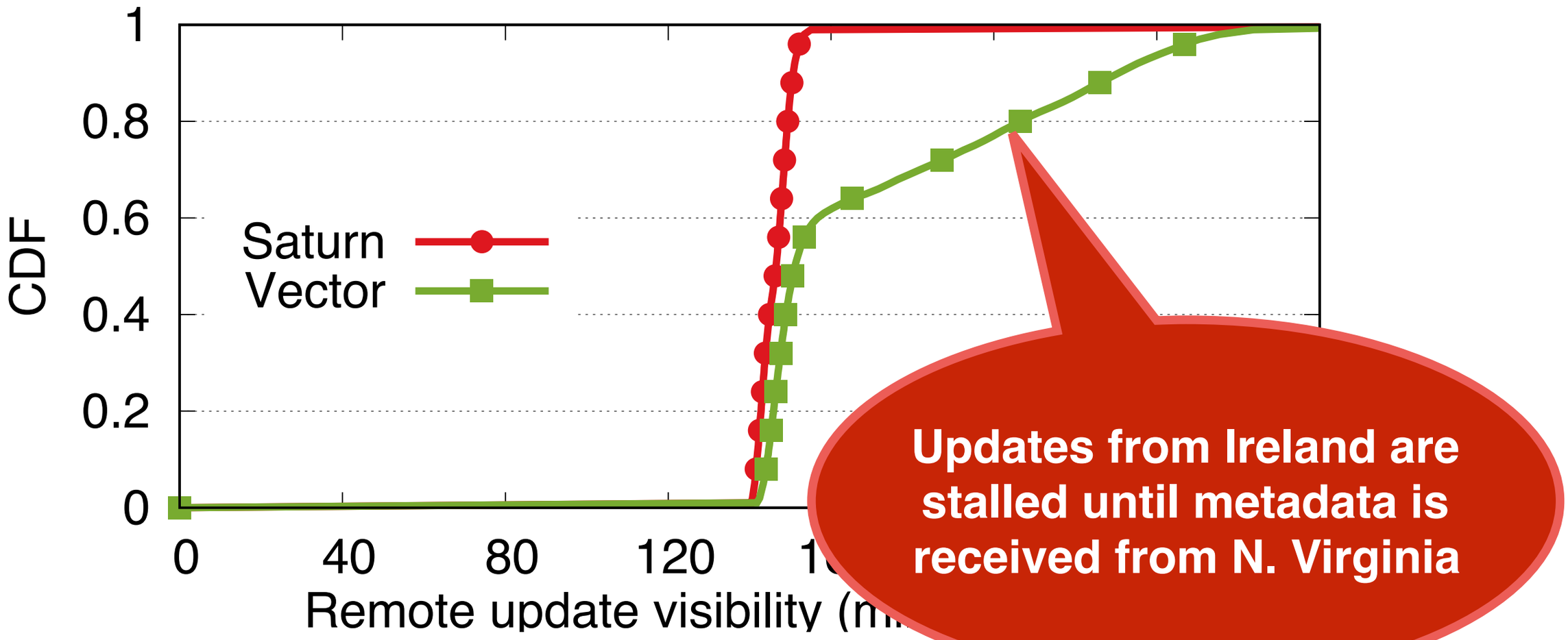
Saturn
Genuine partial replication matters

Latency of updates from Ireland when applied at Sydney



Saturn
Genuine partial replication matters

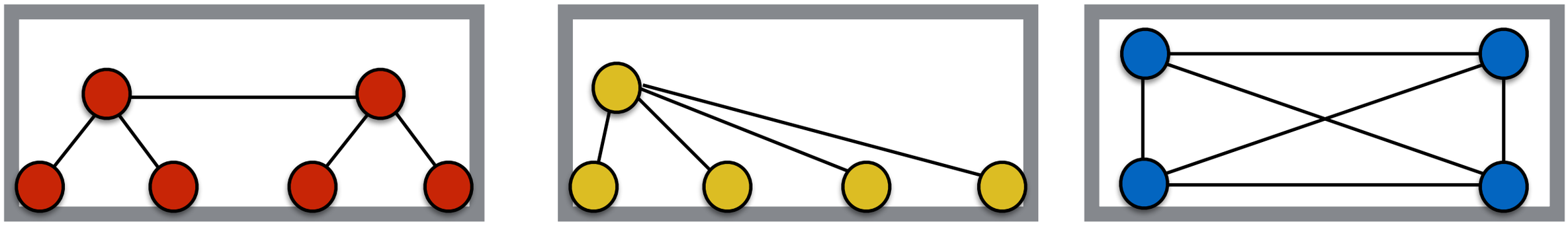
Latency of updates from Ireland when applied at Sydney



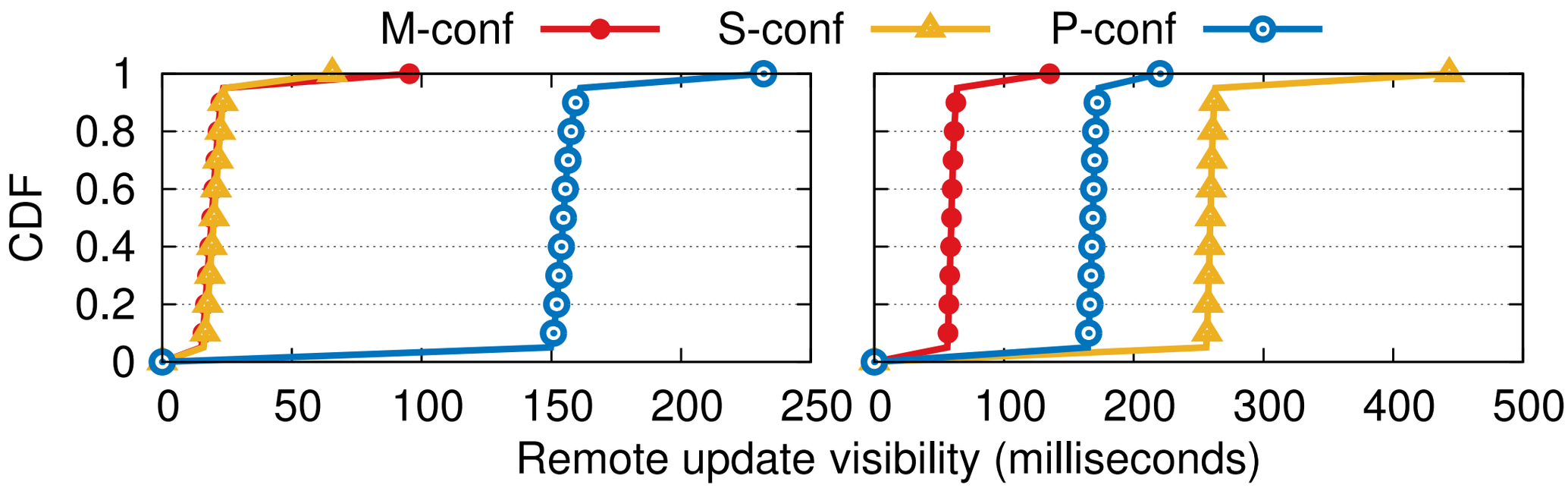
Evaluation: goals

- Can **Saturn+Eunomia** optimize both throughput and data freshness simultaneously?
- Why **Saturn's** genuine partial replication matters?
- How important is the configuration of **Saturn**?
- How **Eunomia** compares to sequencers?

Saturn configuration matters

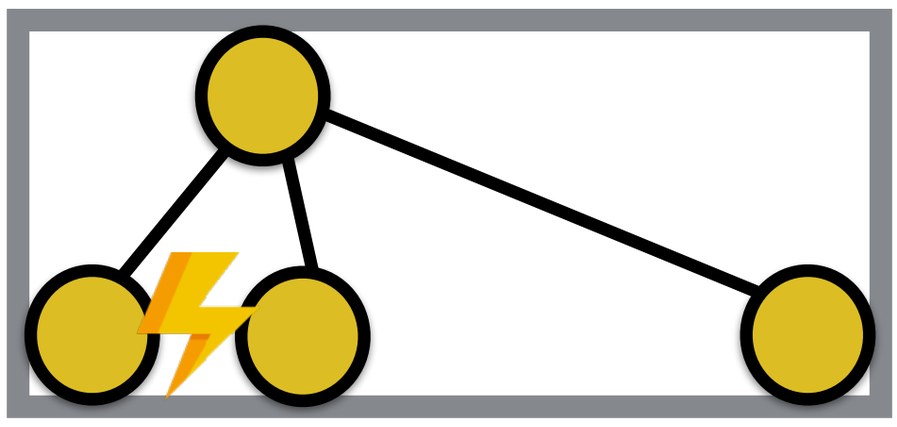


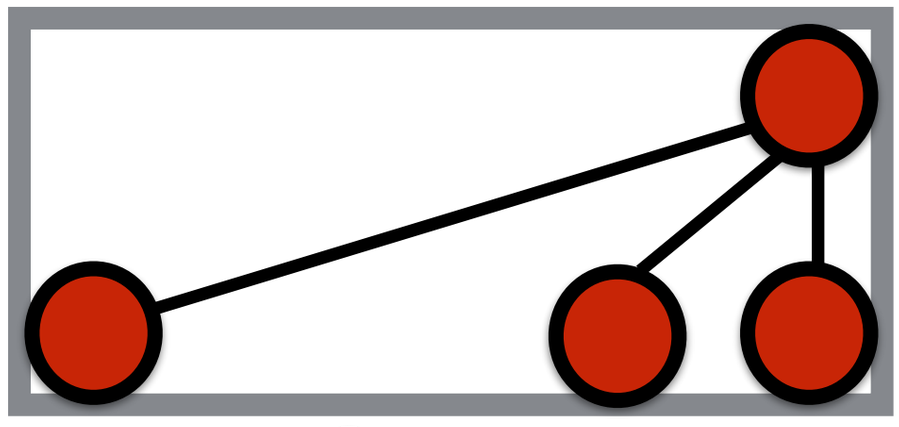
Ireland **Frankfurt** **Tokyo** **Sydney**

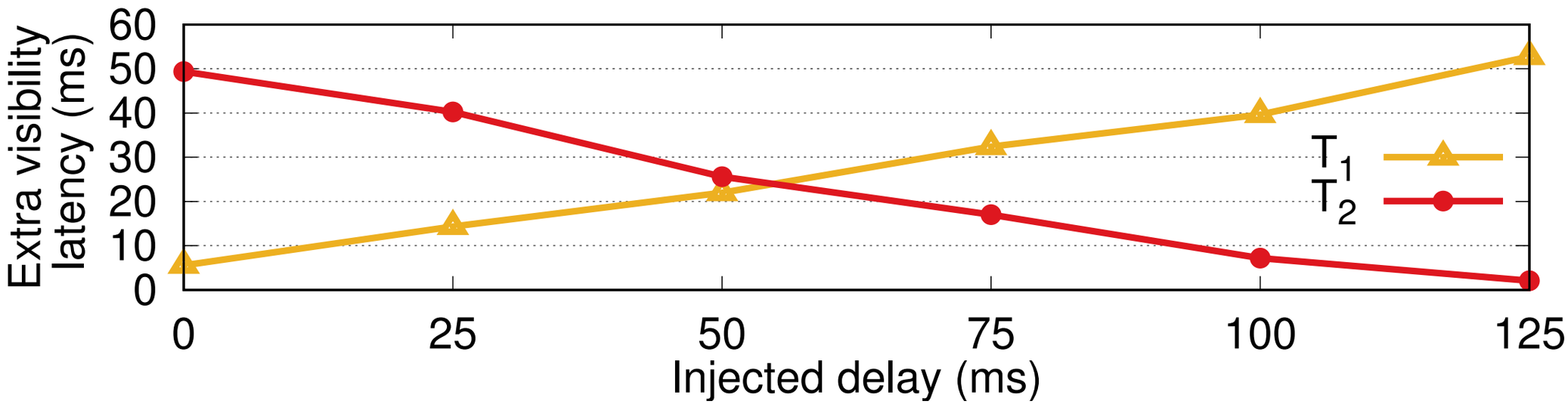


Ireland to Frankfurt **Tokyo to Sydney**

Saturn configuration matters





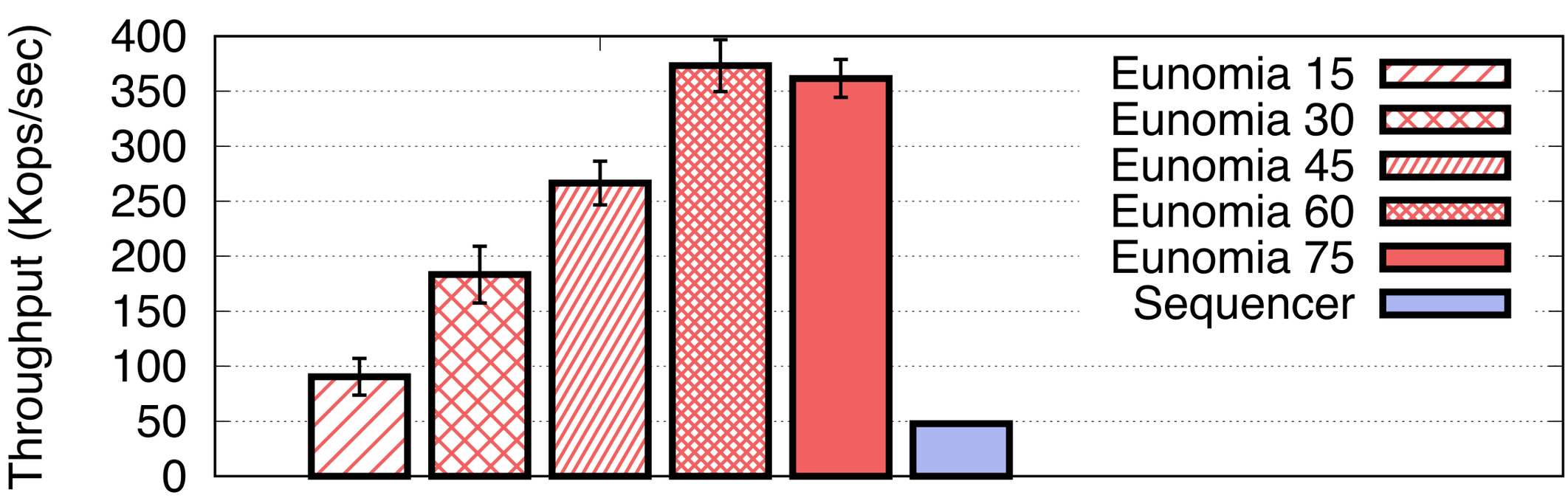


Evaluation: goals

- Can **Saturn+Eunomia** optimize both throughput and data freshness simultaneously?
- Why **Saturn's** genuine partial replication matters?
- How important is the configuration of **Saturn**?
- How **Eunomia** compares to sequencers?

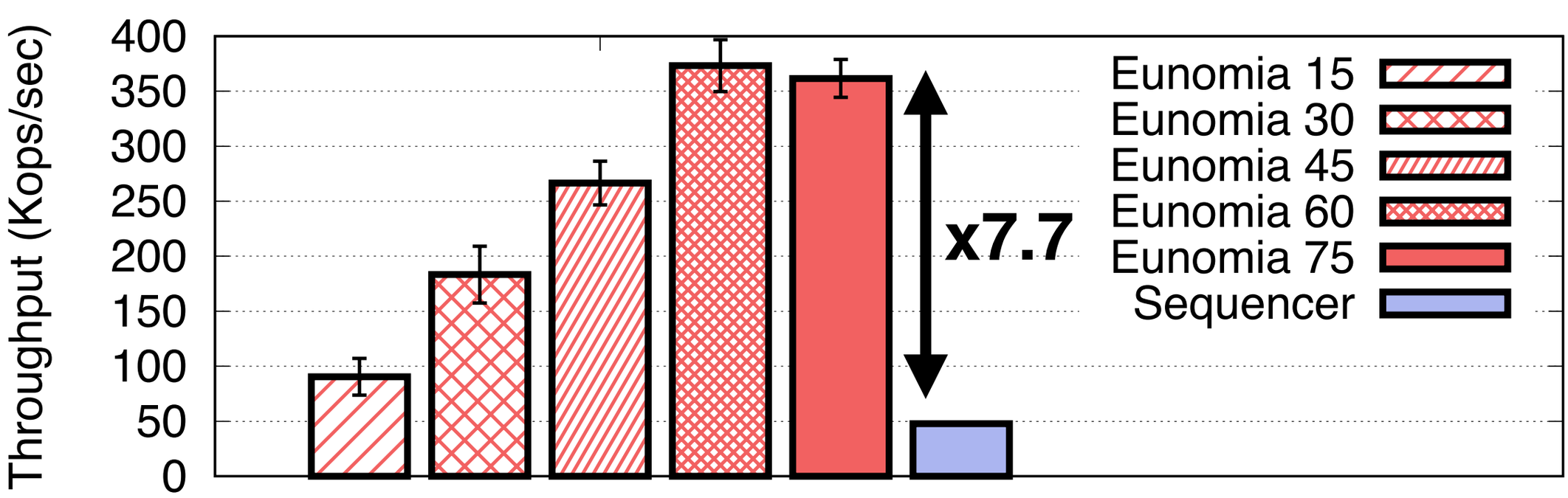
Evaluation

maximum throughput achievable by
Eunomia vs a classical **sequencer**



Evaluation

maximum throughput achievable by
Eunomia vs a classical **sequencer**



Other features…

- Support for **fault-tolerance**
- On-line **reconfiguration** protocols
- Efficient **migration** of clients among data centers
- Impact of **stragglers**
- More **experiments**

+

EUROSYS 2017

ATC 2017

Check the papers for more details.

Take-away messages

- It is possible to preserve concurrency inside a DC, and to capture causality with a single scalar, using an **ordering service that is out the the client's critical path**

- These scalar clocks can be propagated using **tree-based dissemination,** that can effectively mitigate the impact of false dependencies

- Combined, they allow the **optimisation of both throughput and data freshness**