

CONSISTENCY

It's about correctness!

SEQUENTIAL OBJECTS

Thanks to Maurice Herlihy
“The Art of Multiprocessor Programming”

- Each object has a **state**
 - ▶ Register: the value it stores
 - ▶ Queue: the sequence of objects it holds

SEQUENTIAL OBJECTS

Thanks to Maurice Herlihy
“The Art of Multiprocessor Programming”

- Each object has a **state**
 - ▶ Register: the value it stores
 - ▶ Queue: the sequence of objects it holds
- Each object has a set of **methods**
 - ▶ Register: Read/Write
 - ▶ Queue: Enq/Deq/Head

SEQUENTIAL SPECIFICATIONS

Thanks to Maurice Herlihy

- **If** (precondition)
 - ▶ the object is in such-and-such-state before method is called
- **Then** (postcondition)
 - ▶ the method will return a particular value
 - ▶ or throw a particular exception
- **and** (postcondition continued)
 - ▶ the object will be in some other state when method returns

PRE AND POST CONDITIONS FOR DEQ

Thanks to Maurice Herlihy

- Precondition

- ▶ Queue is non-empty



- Postcondition

- ▶ Returns first item in queue

- Postcondition

- ▶ Removes first item in queue

PRE AND POST CONDITIONS FOR DEQ

Thanks to Maurice Herlihy

- Precondition

- ▶ Queue is non-empty



- Postcondition

- ▶ Returns first item in queue ●

- Postcondition

- ▶ Removes first item in queue

PRE AND POST CONDITIONS FOR DEQ

Thanks to Maurice Herlihy

- Precondition

- ▶ Queue is empty



- Postcondition

- ▶ Throws Empty exception

- Postcondition

- ▶ Queue state unchanged

SEQUENTIAL SPECIFICATIONS ARE AWESOME

So is
Maurice Herlihy

- Interactions among methods captured by side-effects on object state
 - ▶ State **between** method calls is meaningful
- Documentation size linear in the number of methods
 - ▶ Separation of concerns: each method described in isolation
- Easy to add new methods
 - ▶ Without changing description of old methods

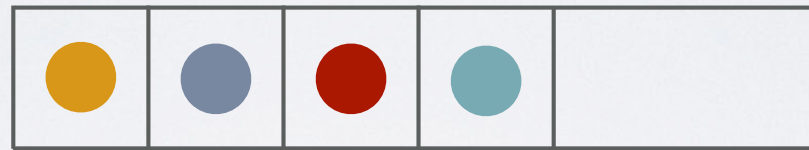
WHAT ABOUT CONCURRENT SPECIFICATIONS?

- Methods?
- Documentation?
- Adding new methods?

CONCURRENCY

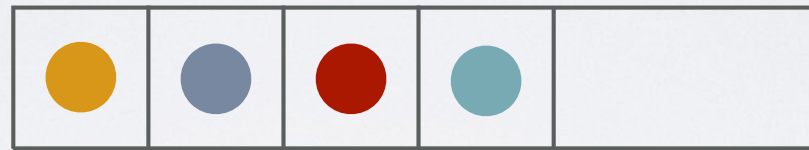


METHODS TAKE TIME



✓
 $Q.enq(\text{●})$





Q.enq(●)



void



Method call

METHODS TAKE TIME

- if you are Sequential
 - ▶ Really? Never noticed!
- ...but if you are Concurrent
 - ▶ Method call is not an event
 - ▶ Method call is an interval
 - ★ Concurrent method calls overlap! ★

WHAT DOES IT MEAN FOR CORRECTNESS?

- Sequential

- ▶ Object needs meaningful state only between method calls

- Concurrent

- ▶ Because method calls overlap, object may **never** be between method calls

WHAT DOES IT MEAN FOR CORRECTNESS?

- Sequential

- ▶ Each method described in isolation

- Concurrent

- ▶ Must consider **all possible interactions** between concurrent calls
 - What if two `enq()` overlap?
 - What if `enq()` and `deq()` overlap?

WHAT DOES IT MEAN FOR CORRECTNESS?

- Sequential
 - ▶ New methods do not affect existing methods
- Concurrent
 - ▶ Everything can potentially interact with everything else



A dramatic landscape photograph featuring a road that splits into two paths, leading through a vast field of golden wheat. The sky is filled with large, dark, and textured clouds, with a bright sun low on the horizon to the left, casting a warm glow. In the distance, dark silhouettes of hills or mountains are visible against the horizon line.

Distributed Systems

Databases

REGISTERS

- Sequential specification
 - ▶ A read returns the result of the latest completed write

REGISTERS

- Sequential specification
 - ▶ A read returns the result of the latest completed write
- What if reads and writes can be concurrent?

REGISTERS

- Sequential specification
 - ▶ A read returns the result of the latest completed write
- What if reads and writes can be concurrent?
 - ▶ A read **not concurrent with a write** returns the result of the latest completed write

REGISTERS

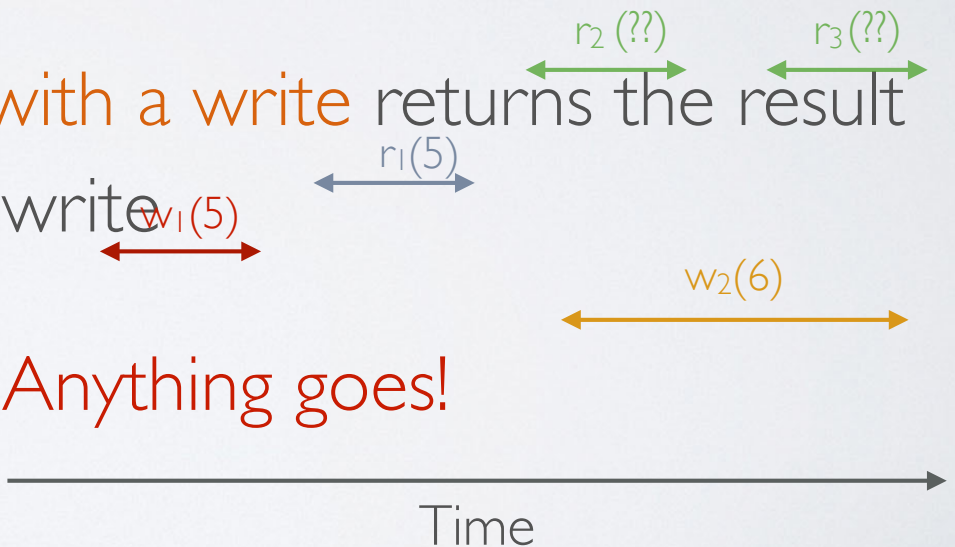
- Sequential specification
 - ▶ A read returns the result of the latest completed write
- What if reads and writes can be concurrent?
 - ▶ A read **not concurrent with a write** returns the result of the latest completed write
- And if they are concurrent?

SAFE REGISTERS

- Sequential specification
 - ▶ A read returns the result of the latest completed write
- What if reads and writes can be concurrent?
 - ▶ A read **not concurrent with a write** returns the result of the latest completed write
- And if they are concurrent? **Anything goes!**

SAFE REGISTERS

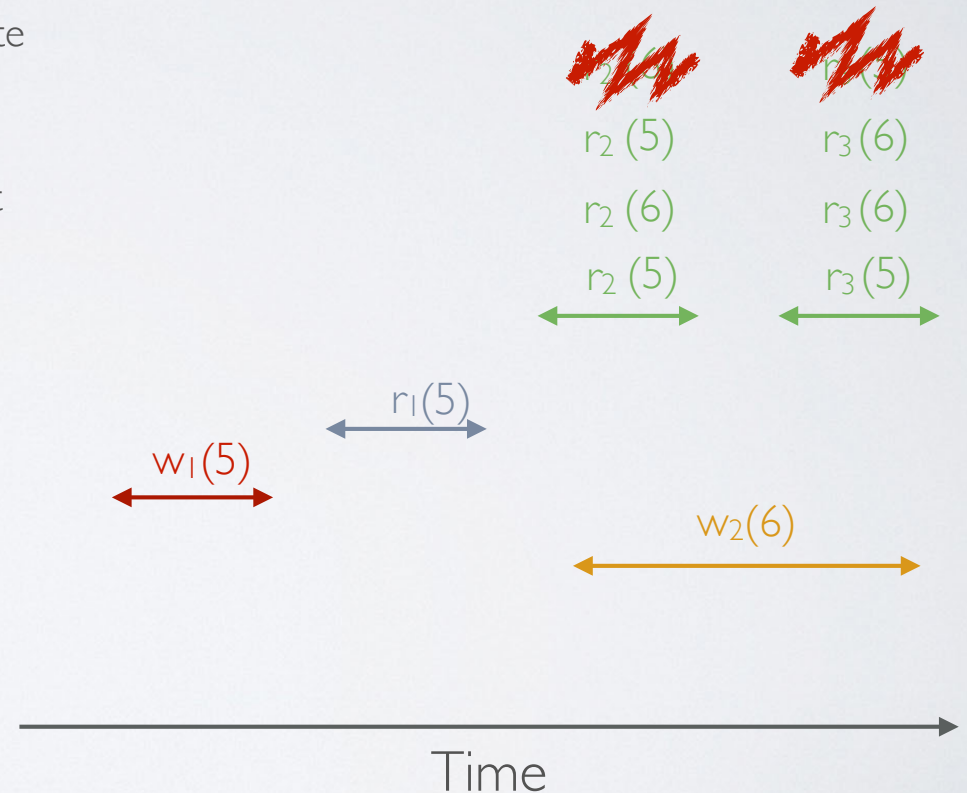
- Sequential specification
 - ▶ A read returns the result of the latest completed write
- What if reads and writes can be concurrent?
 - ▶ A read **not concurrent with a write** returns the result of the latest completed write
- And if they are concurrent? **Anything goes!**



REGULAR REGISTERS

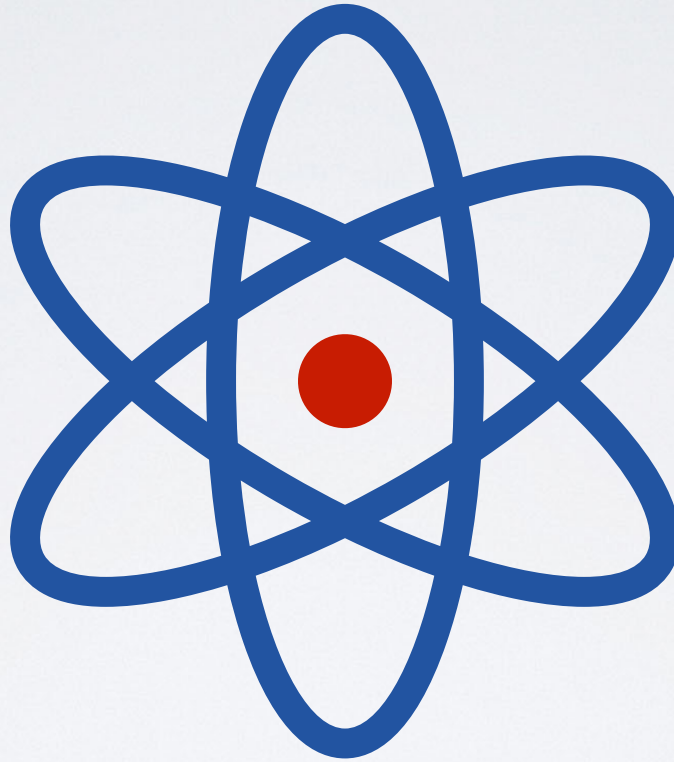
- Sequential specification
 - A read returns the result of the latest completed write
- What if reads and writes can be concurrent?
 - A read **not concurrent with a write** returns the result of the latest completed write
- And if they are concurrent?

A read overlapping with a write returns either the old or the new value!



CAN WE DO BETTER?

ATOMIC REGISTERS

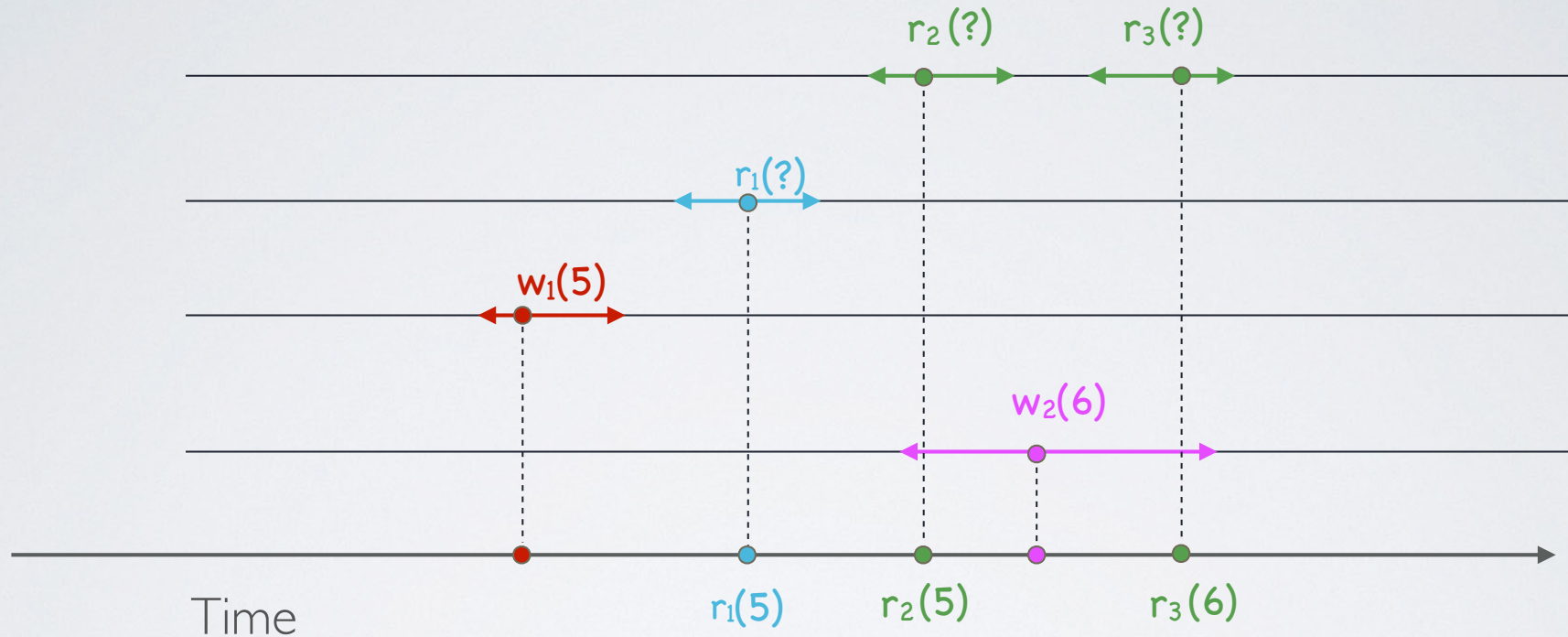


LINEARIZABILITY

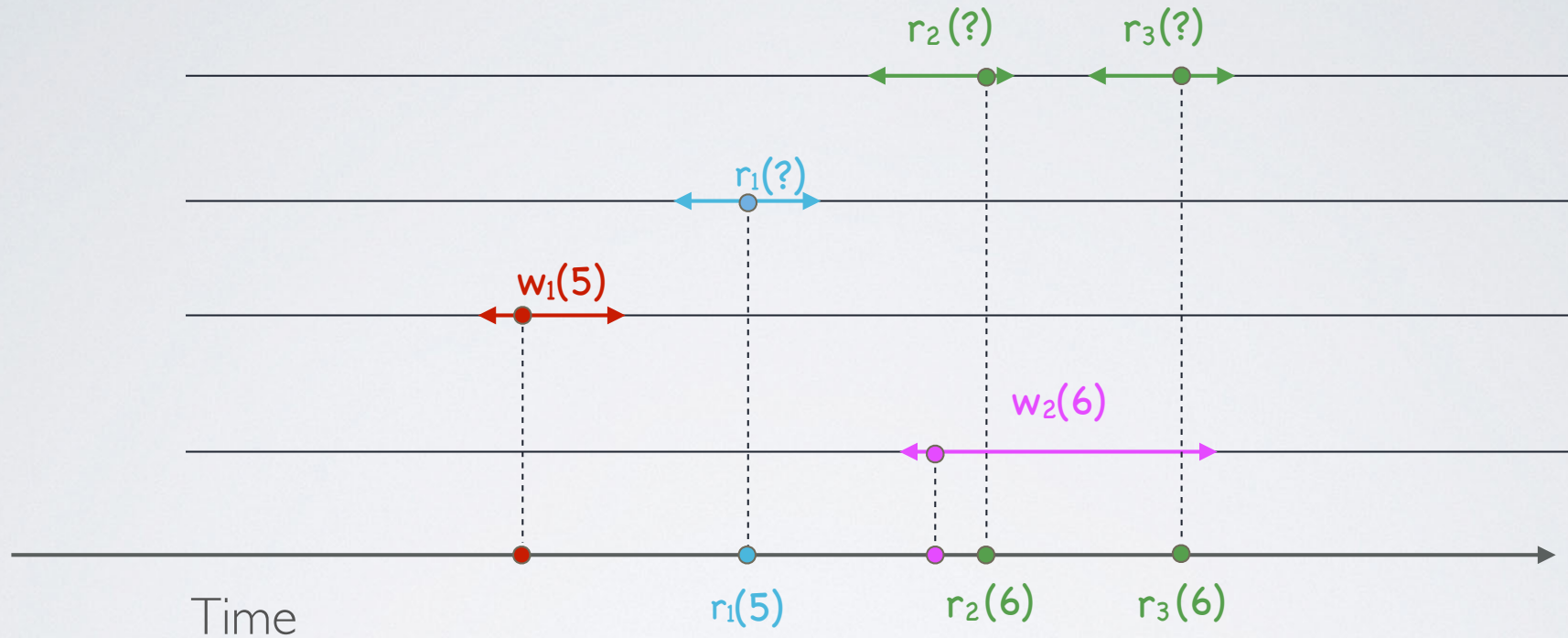
Herlihy & Wing '87

- Each method
 - ▶ Takes effect instantaneously
 - ▶ Between *invocation* and *response*
- Object is correct (*linearizable*) if this “sequential” behavior is correct (i.e., it meets the object’s sequential specification)
 - ▶ All executions of a linearizable object are linearizable

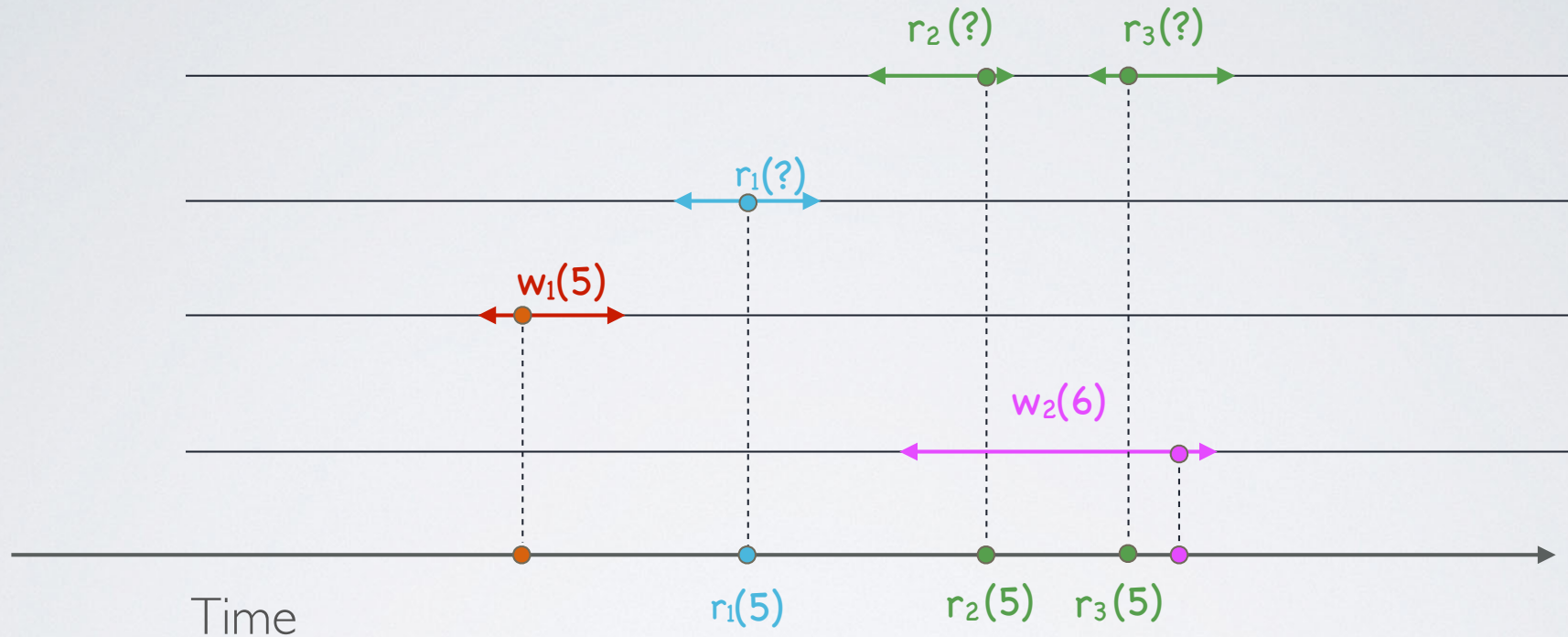
EXAMPLE: REGISTERS



EXAMPLE: REGISTERS

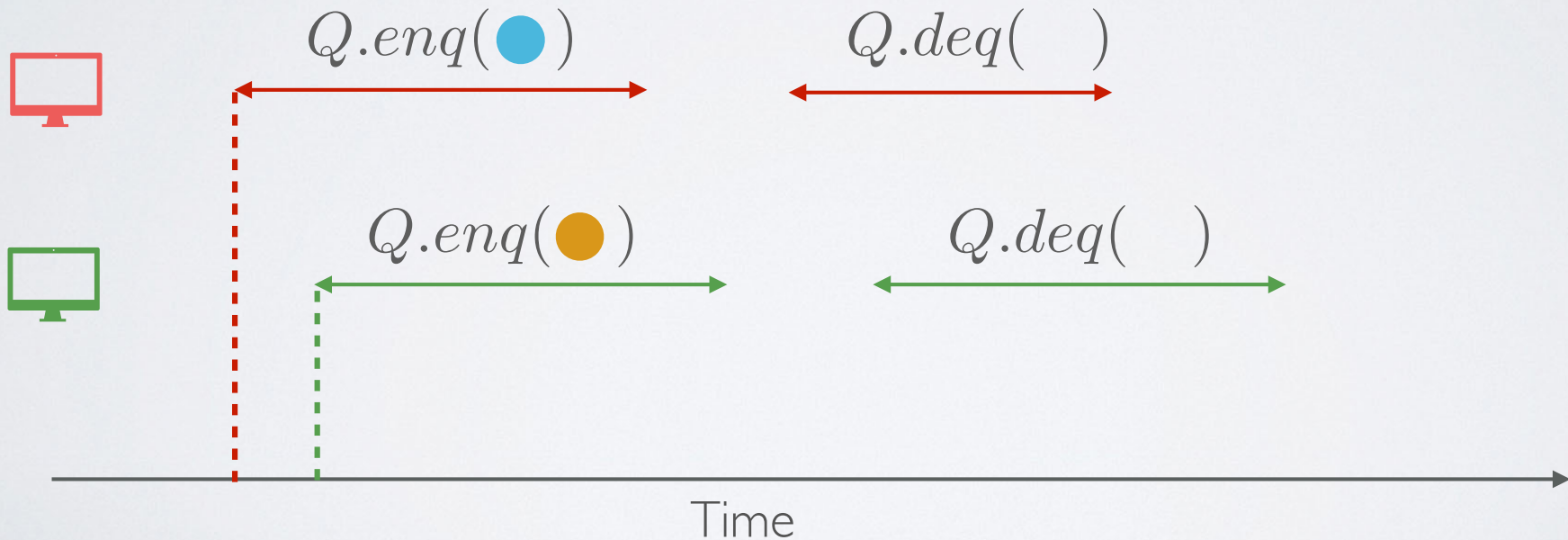
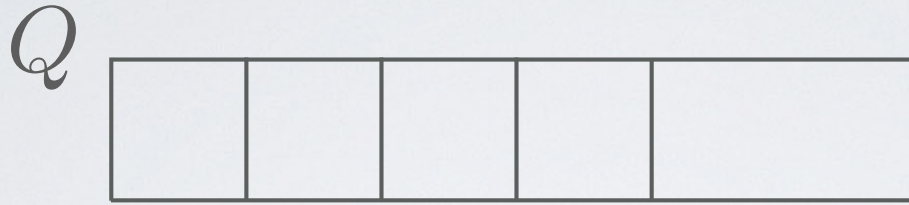


EXAMPLE: REGISTERS

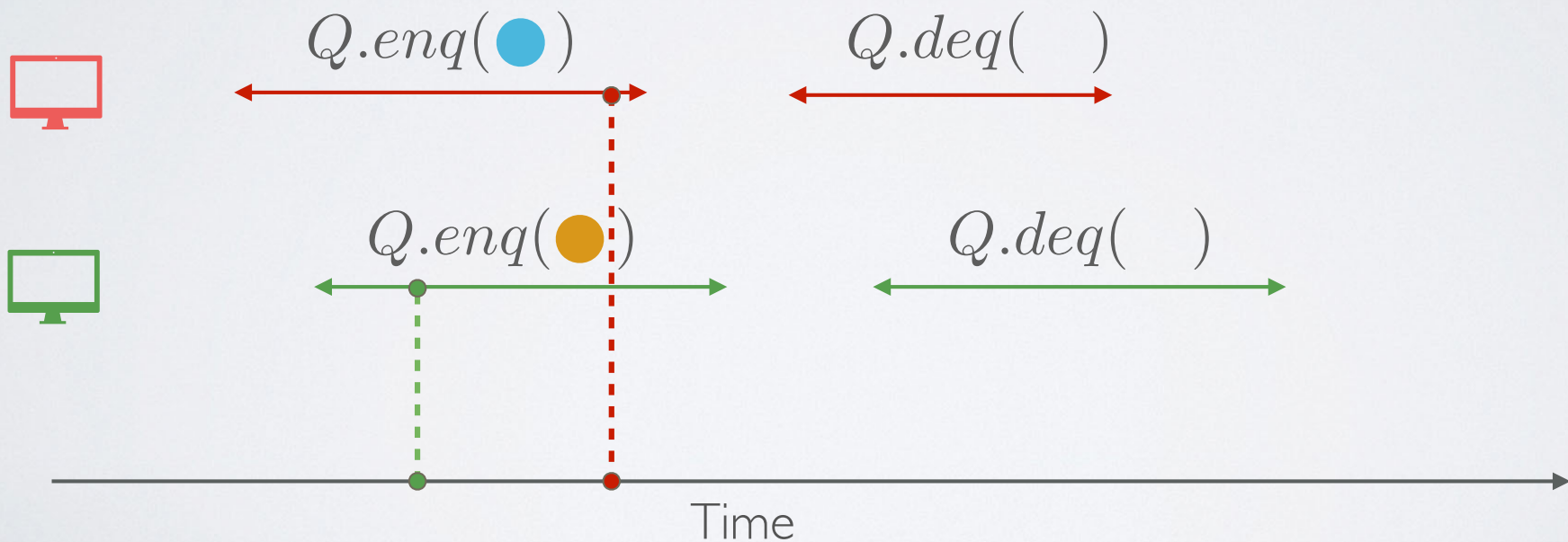
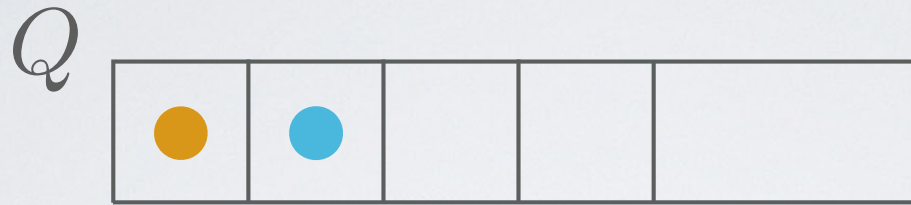


Linearizable registers are called **atomic**

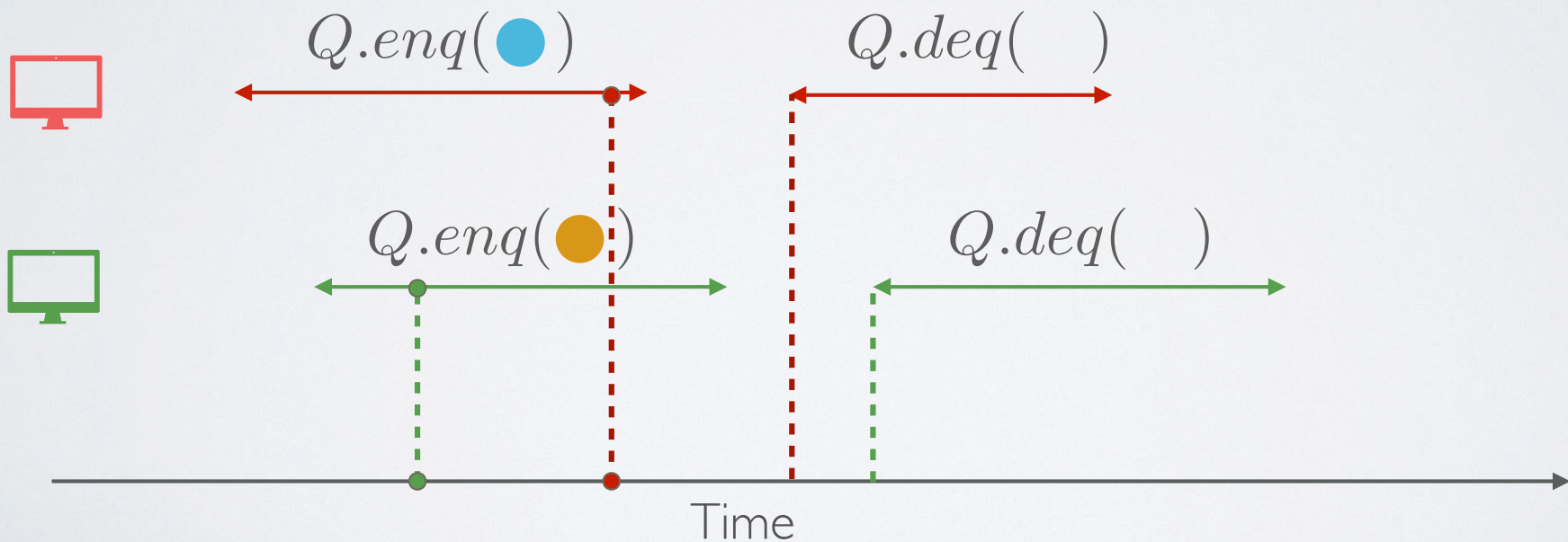
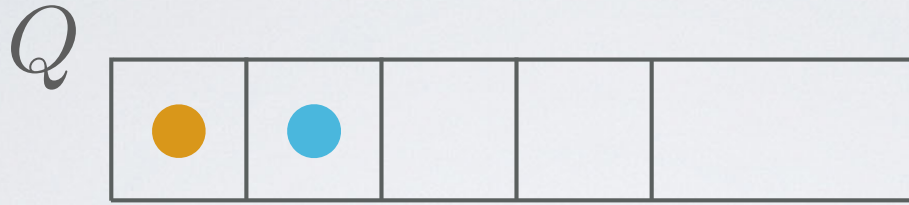
LINEARIZABLE QUEUE



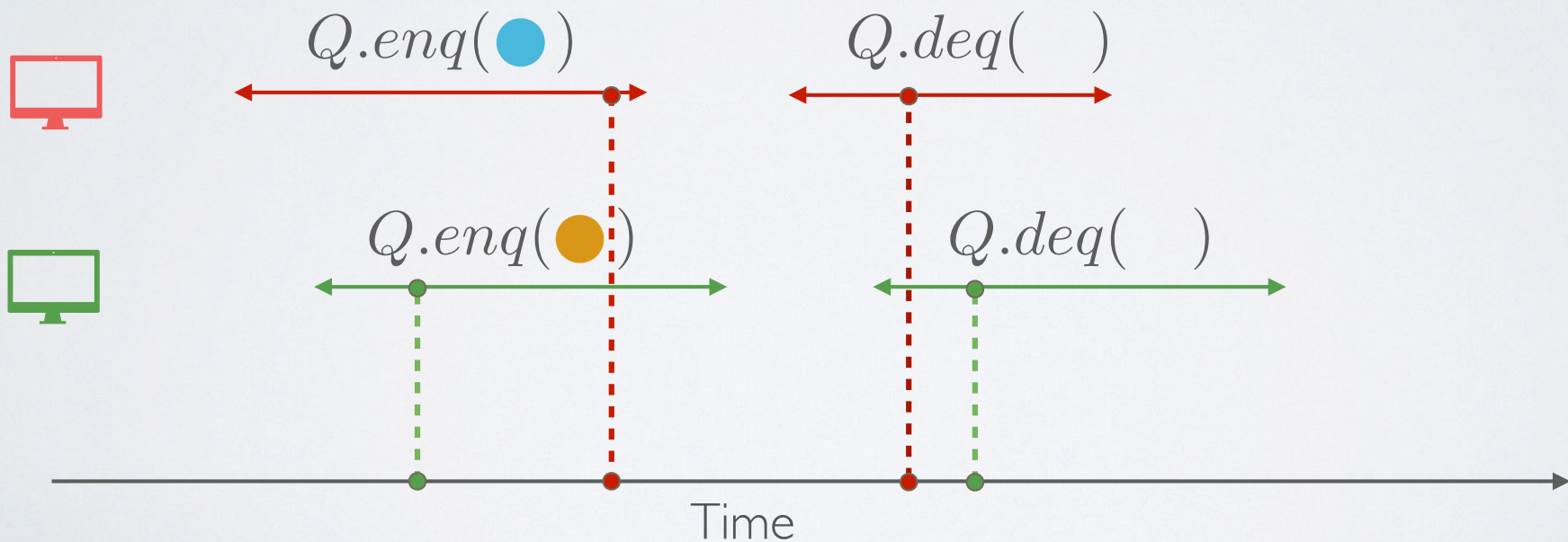
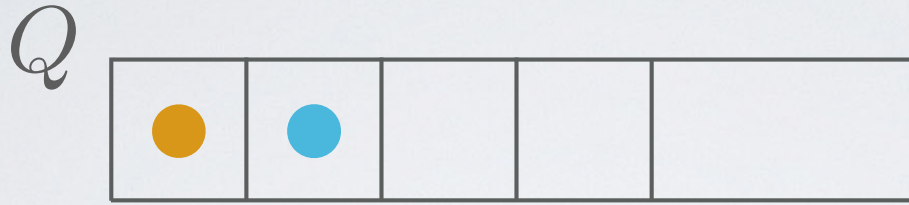
LINEARIZABLE QUEUE



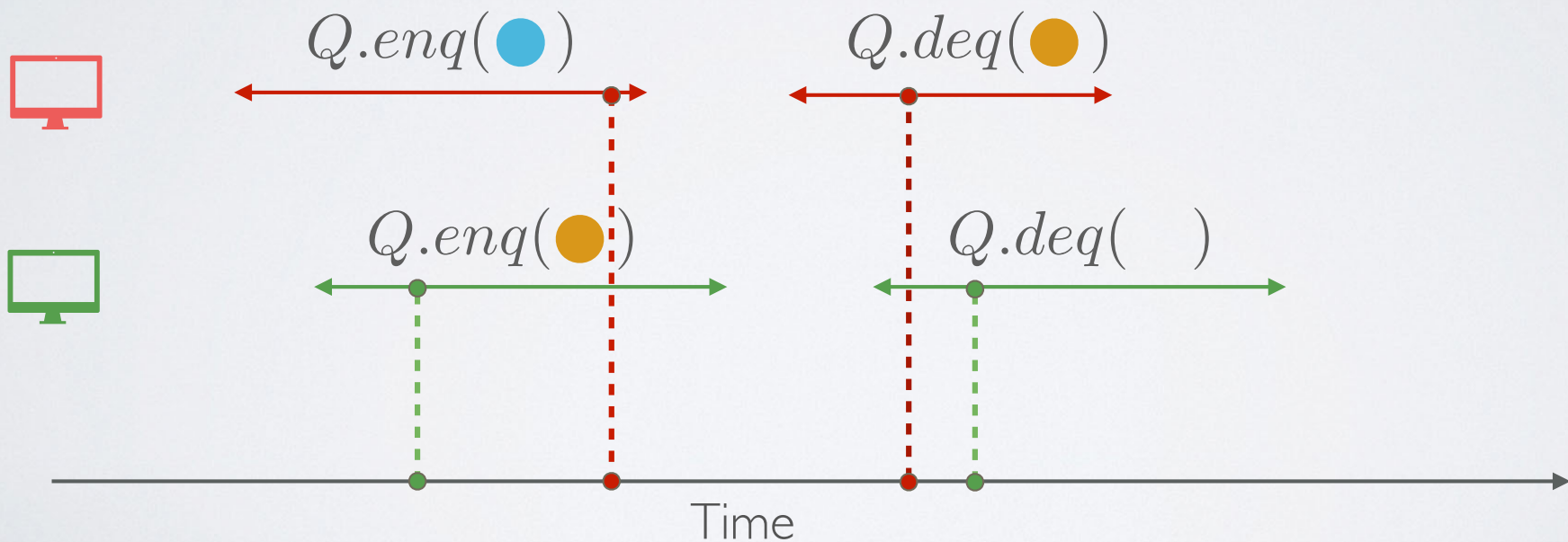
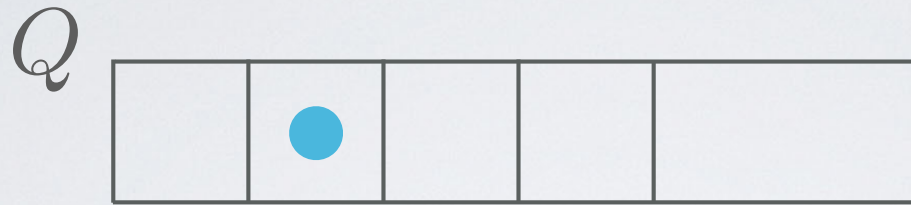
LINEARIZABLE QUEUE



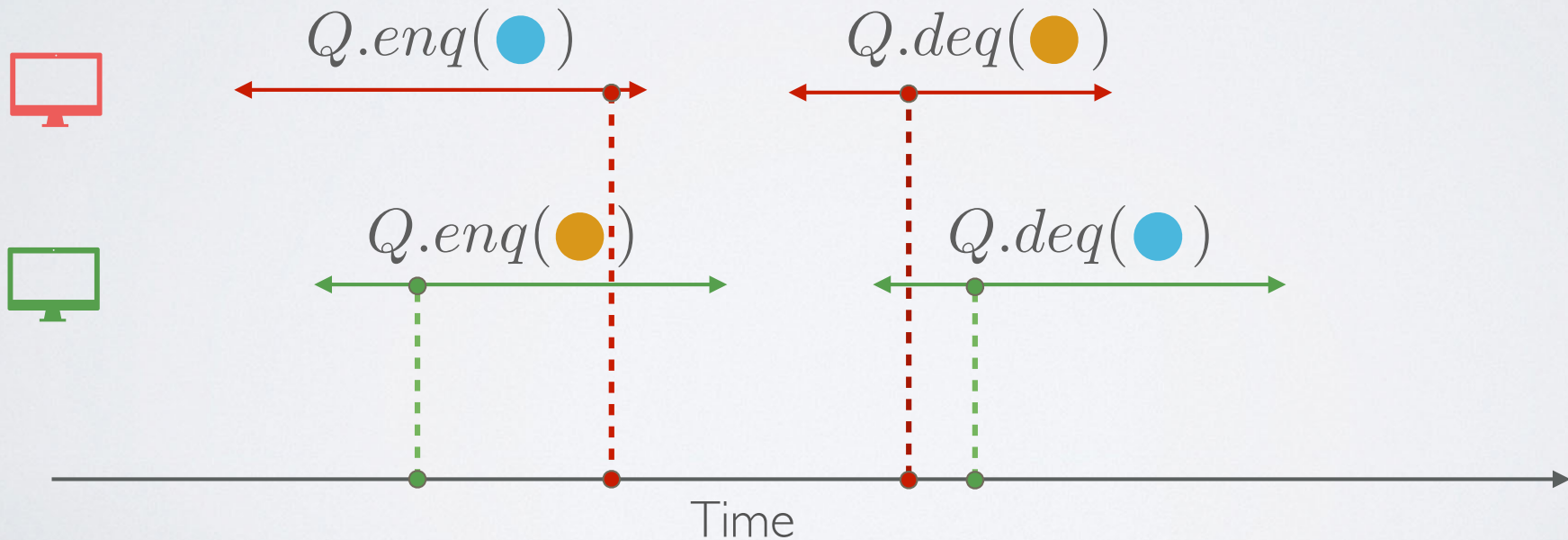
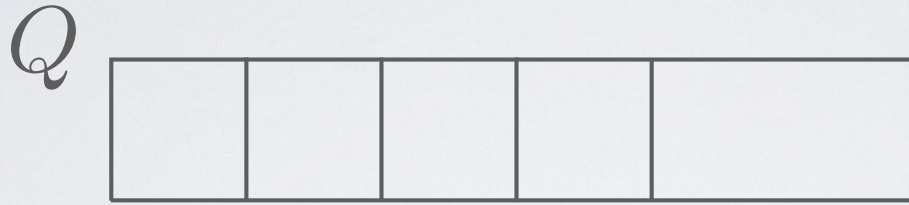
LINEARIZABLE QUEUE



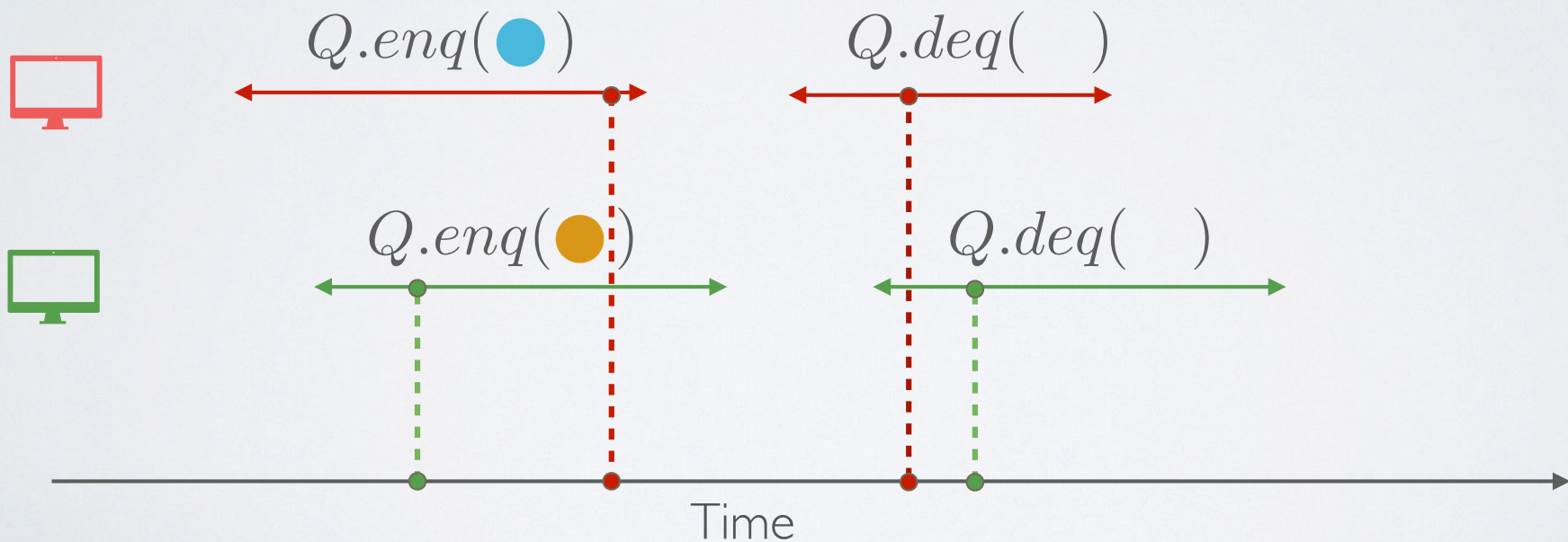
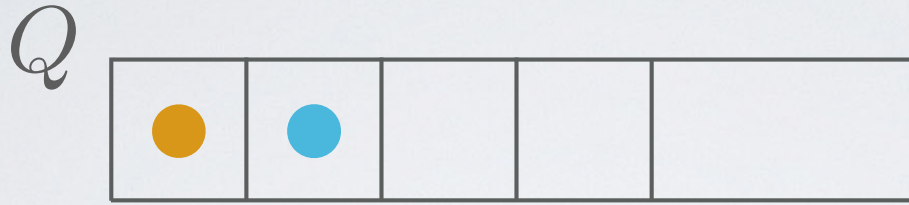
LINEARIZABLE QUEUE



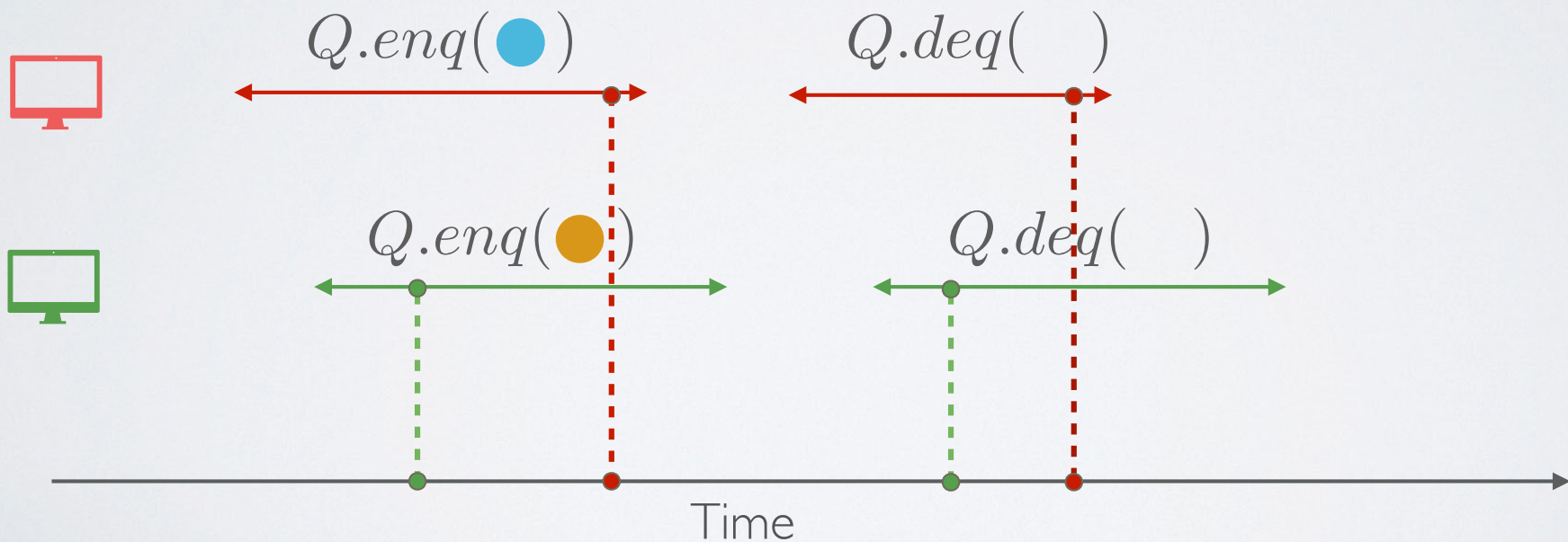
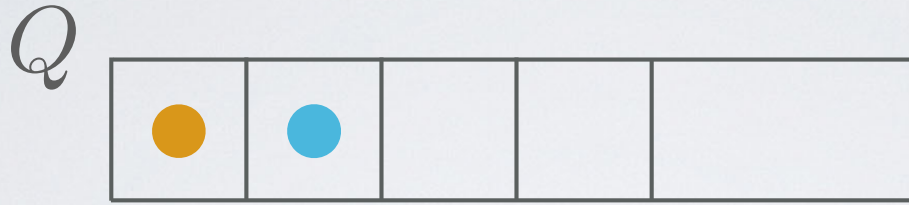
LINEARIZABLE QUEUE



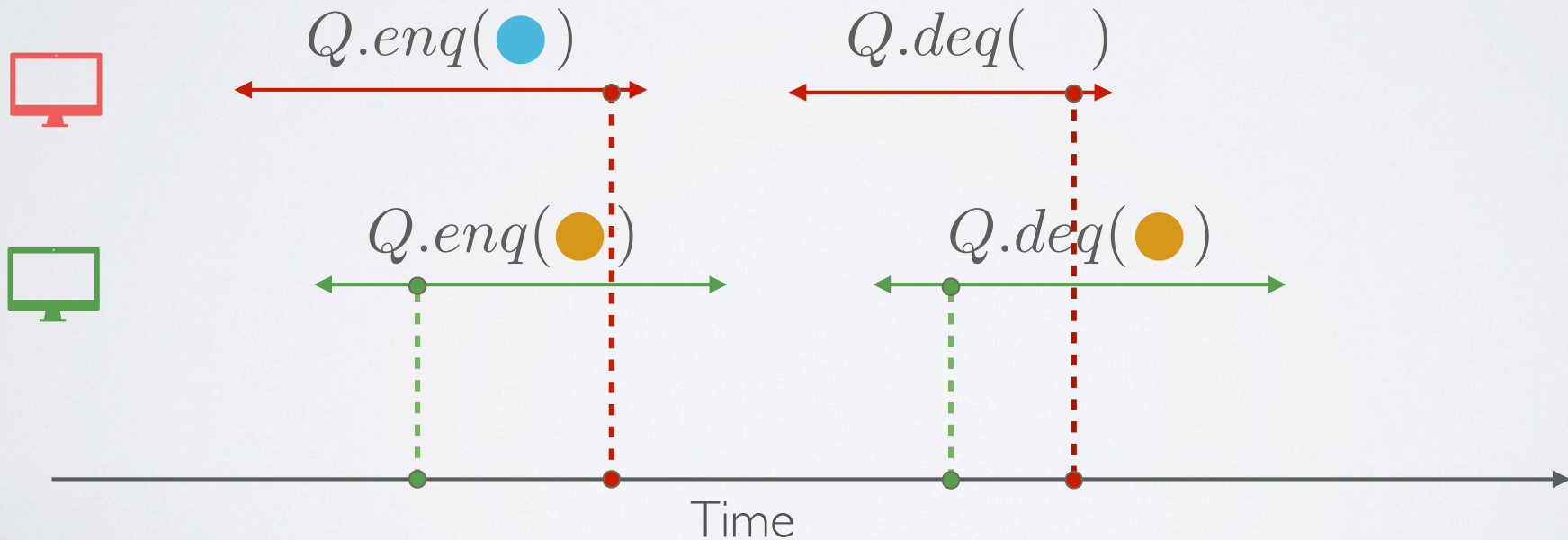
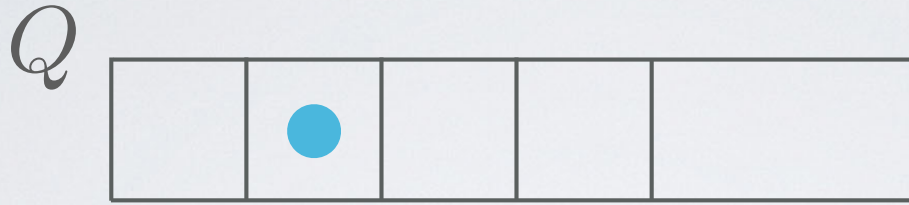
LINEARIZABLE QUEUE



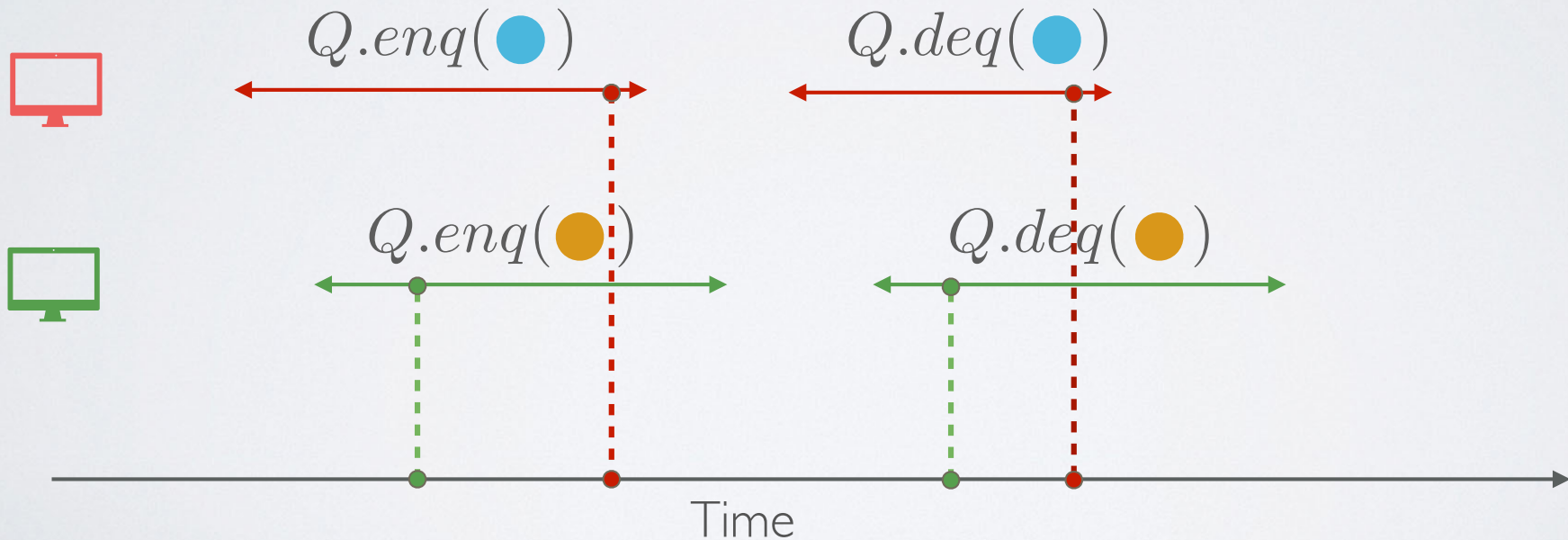
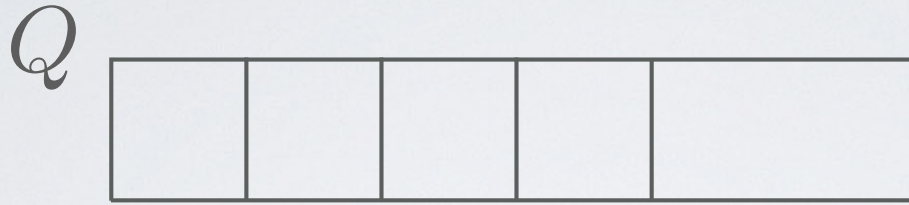
LINEARIZABLE QUEUE



LINEARIZABLE QUEUE



LINEARIZABLE QUEUE



An alternative...

SEQUENTIAL CONSISTENCY

(LAMPORT "HOW TO MAKE A MULTIPROCESSOR COMPUTER THAT CORRECTLY EXECUTES MULTIPROCESS PROGRAMS", IEEE TOC C-28,9 (SEPT. 1979), 690-691)

- A history H is **sequentially consistent** if there exists a permutation π of the operations in H such that
 - ▶ $\pi|_o$ respects the sequential specification of each object o
 - ▶ If the response for operation o_1 at p_i occurs in H before the invocation for operation o_2 at p_i , then o_1 appears before o_2 in π

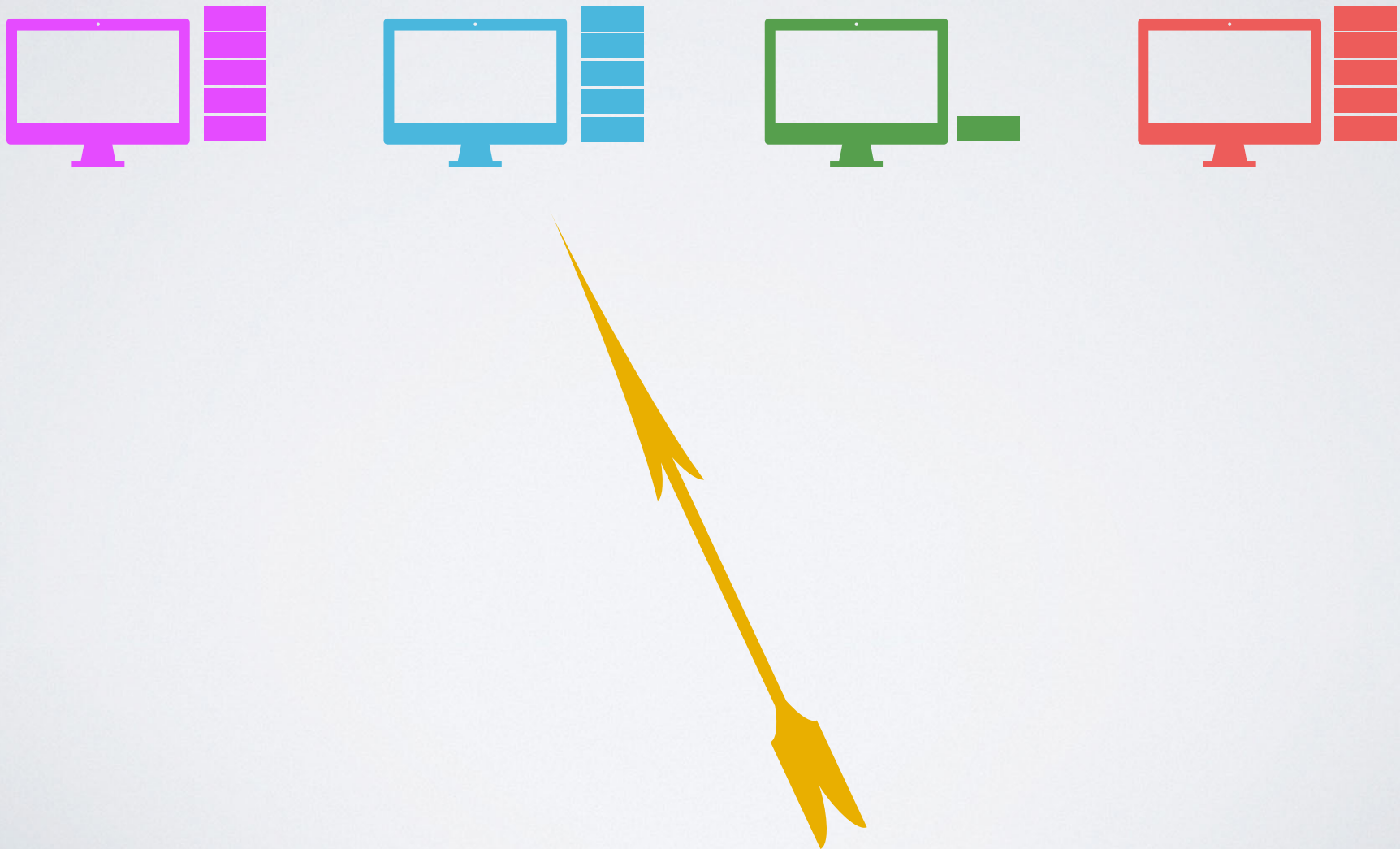
SEQUENTIAL CONSISTENCY

(LAMPORT "HOW TO MAKE A MULTIPROCESSOR COMPUTER THAT CORRECTLY EXECUTES MULTIPROCESS PROGRAMS", IEEE TOC C-28,9 (SEPT. 1979), 690-691)



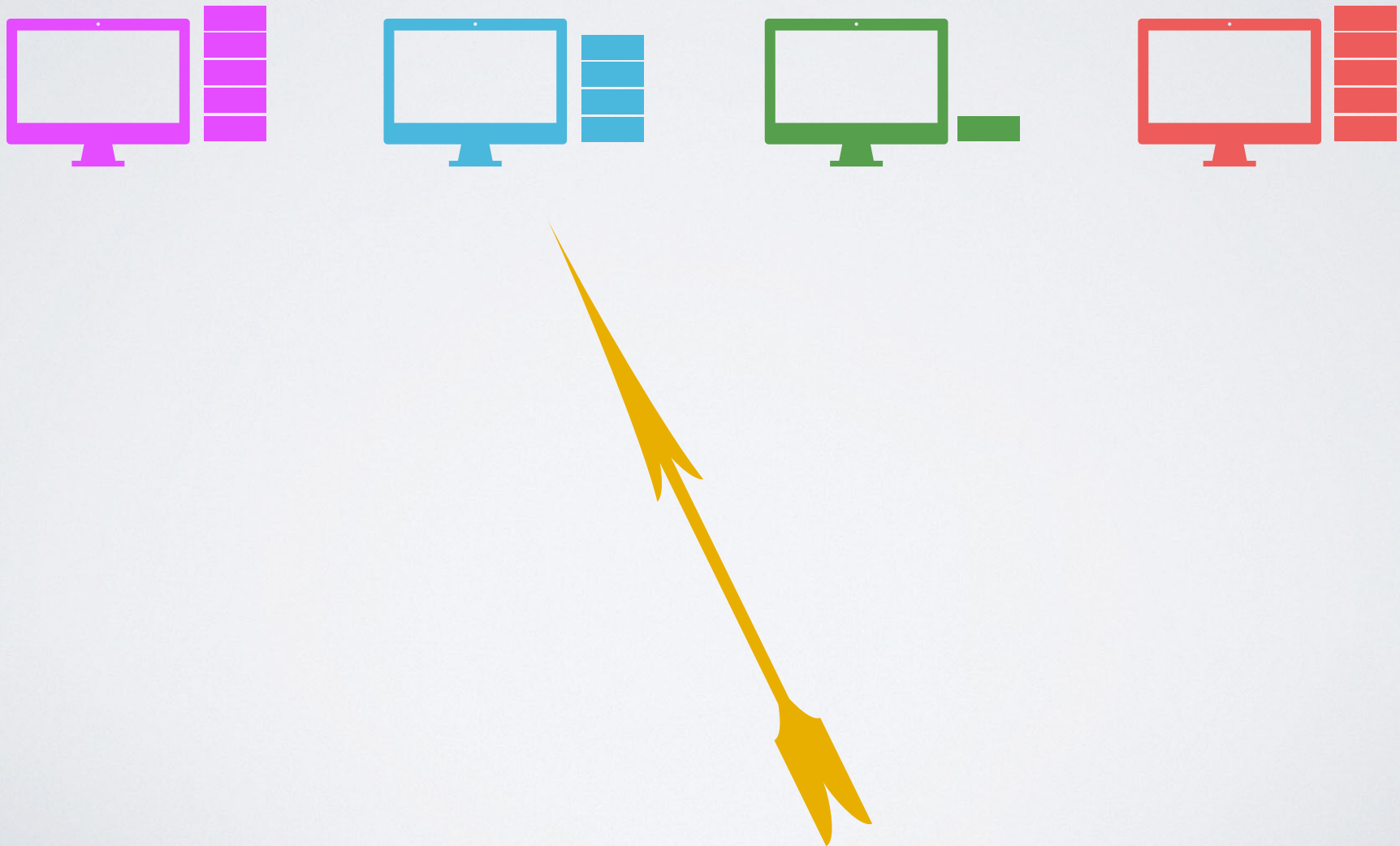
SEQUENTIAL CONSISTENCY

(LAMPORT "HOW TO MAKE A MULTIPROCESSOR COMPUTER THAT CORRECTLY EXECUTES MULTIPROCESS PROGRAMS", IEEE TOC C-28,9 (SEPT. 1979), 690-691)



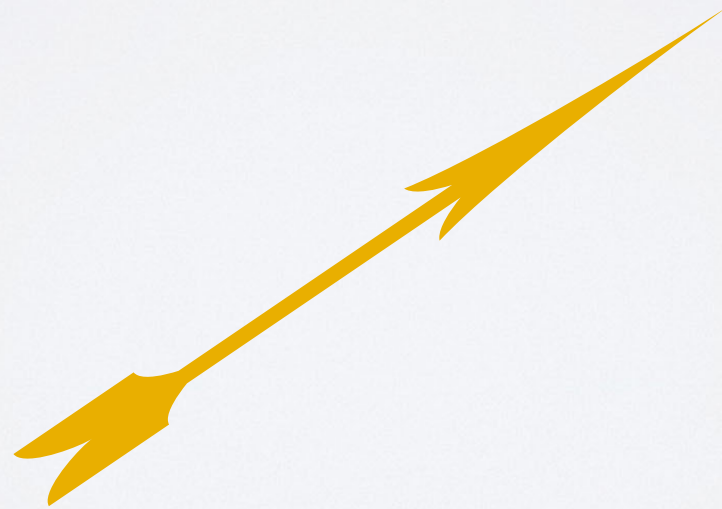
SEQUENTIAL CONSISTENCY

(LAMPORT "HOW TO MAKE A MULTIPROCESSOR COMPUTER THAT CORRECTLY EXECUTES MULTIPROCESS PROGRAMS", IEEE TOC C-28,9 (SEPT. 1979), 690-691)



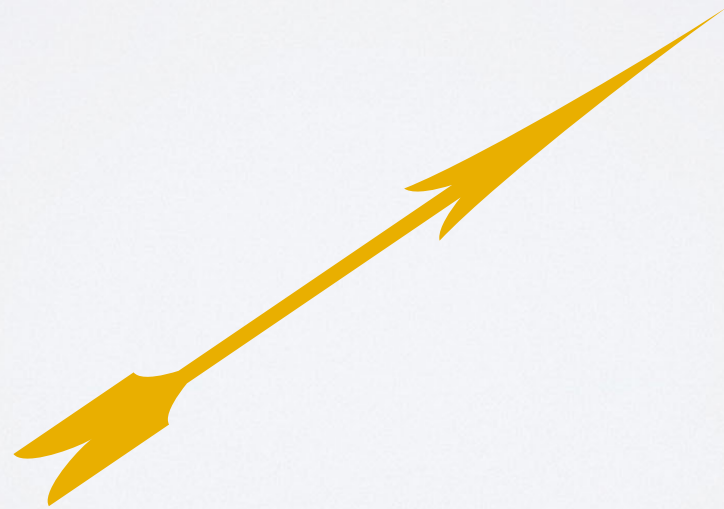
SEQUENTIAL CONSISTENCY

(LAMPORT "HOW TO MAKE A MULTIPROCESSOR COMPUTER THAT CORRECTLY EXECUTES MULTIPROCESS PROGRAMS", IEEE TOC C-28,9 (SEPT. 1979), 690-691)



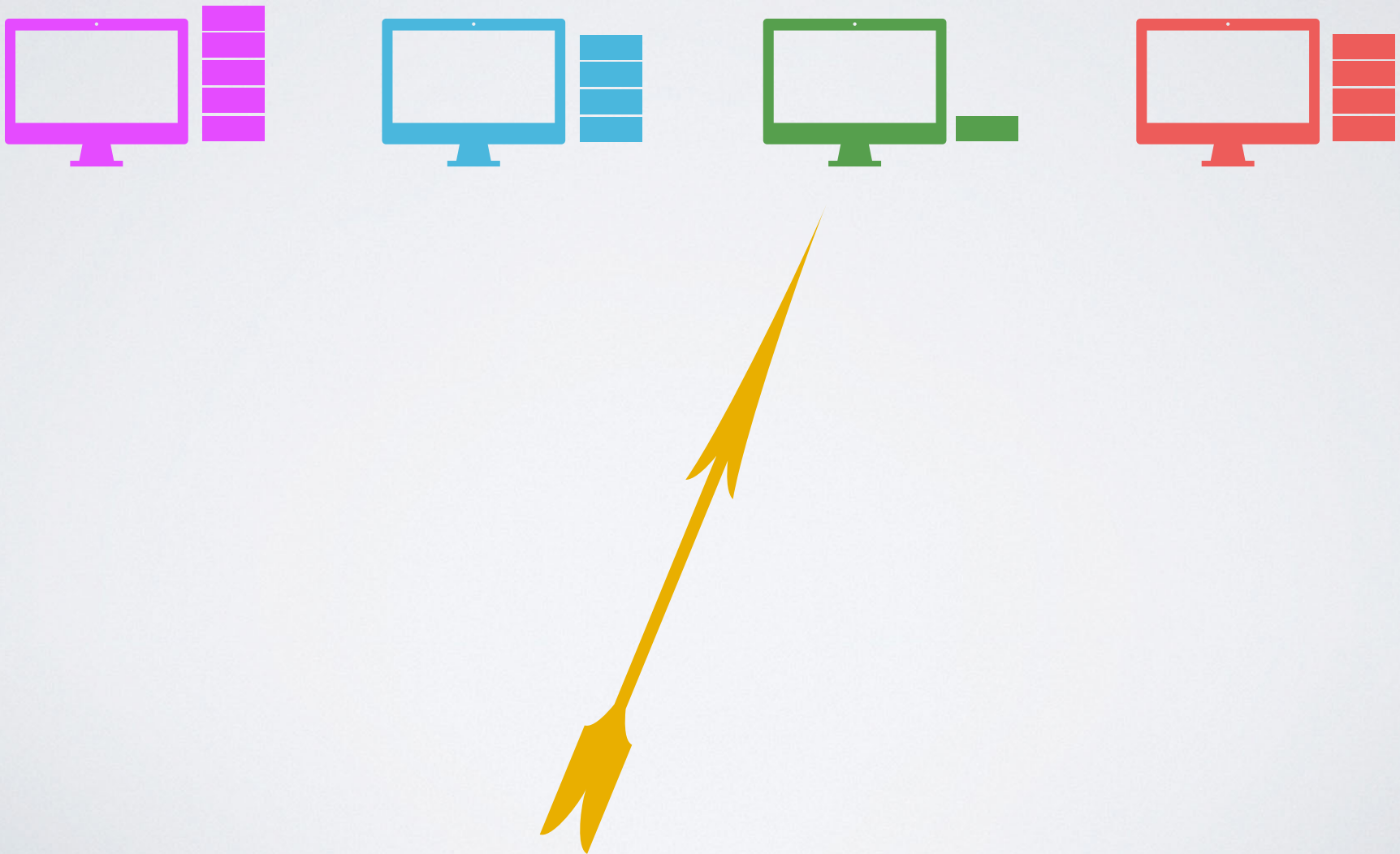
SEQUENTIAL CONSISTENCY

(LAMPORT "HOW TO MAKE A MULTIPROCESSOR COMPUTER THAT CORRECTLY EXECUTES MULTIPROCESS PROGRAMS", IEEE TOC C-28,9 (SEPT. 1979), 690-691)



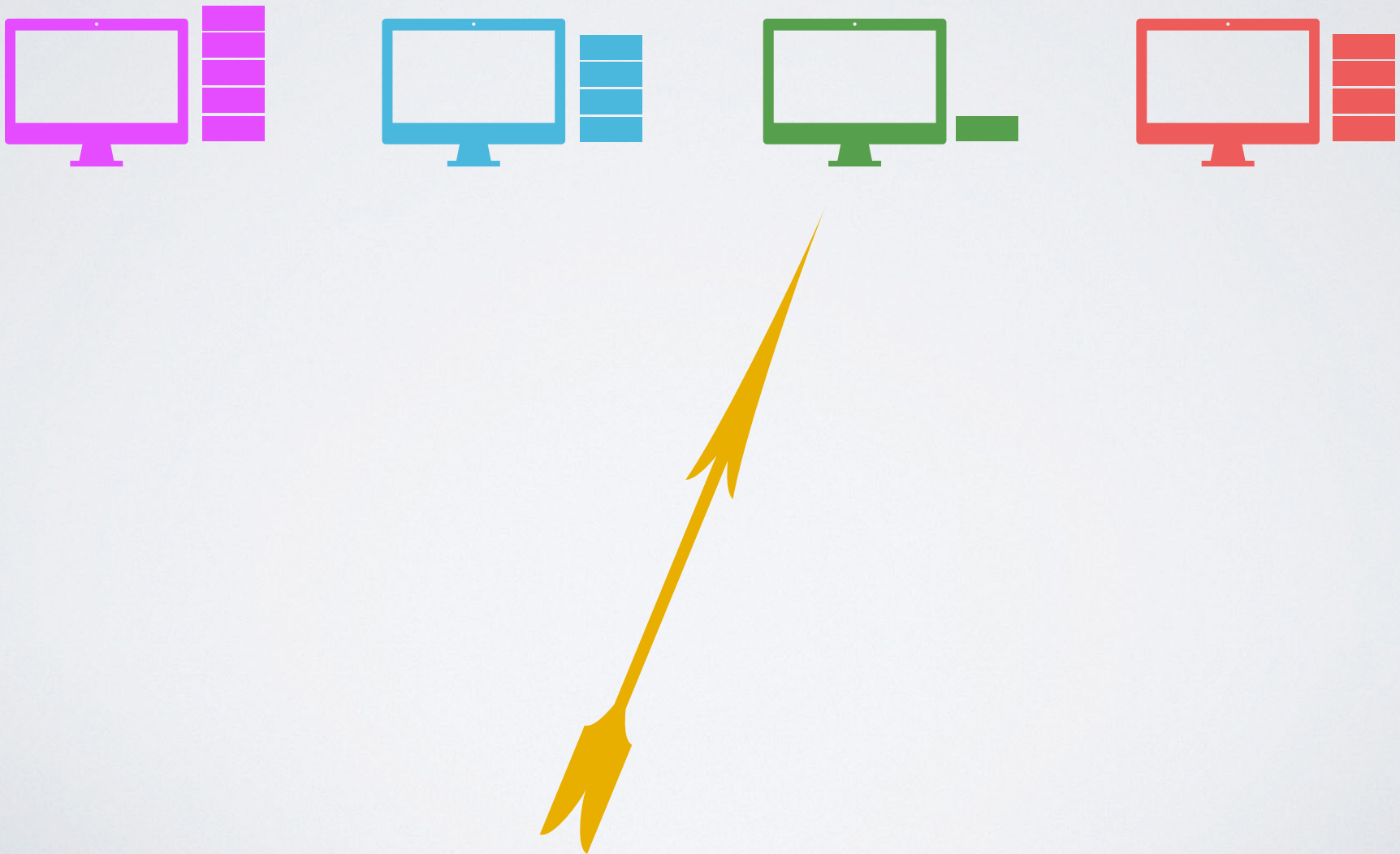
SEQUENTIAL CONSISTENCY

(LAMPORT "HOW TO MAKE A MULTIPROCESSOR COMPUTER THAT CORRECTLY EXECUTES MULTIPROCESS PROGRAMS", IEEE TOC C-28,9 (SEPT. 1979), 690-691)



SEQUENTIAL CONSISTENCY

(LAMPORT "HOW TO MAKE A MULTIPROCESSOR COMPUTER THAT CORRECTLY EXECUTES MULTIPROCESS PROGRAMS", IEEE TOC C-28,9 (SEPT. 1979), 690-691)



SEQUENTIAL CONSISTENCY

(LAMPORT "HOW TO MAKE A MULTIPROCESSOR COMPUTER THAT CORRECTLY EXECUTES MULTIPROCESS PROGRAMS", IEEE TOC C-28,9 (SEPT. 1979), 690-691)

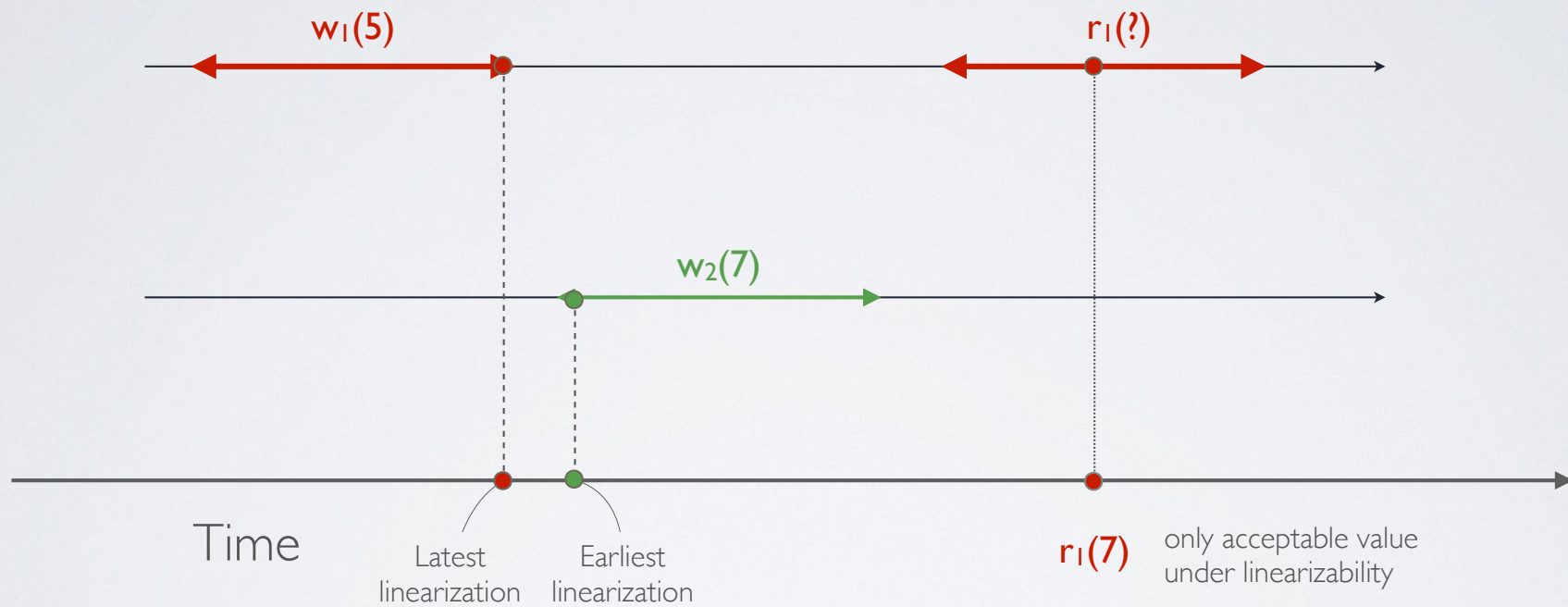


Temporal order
operations is
operations by

of non-overlapping
preserved only for
the same thread

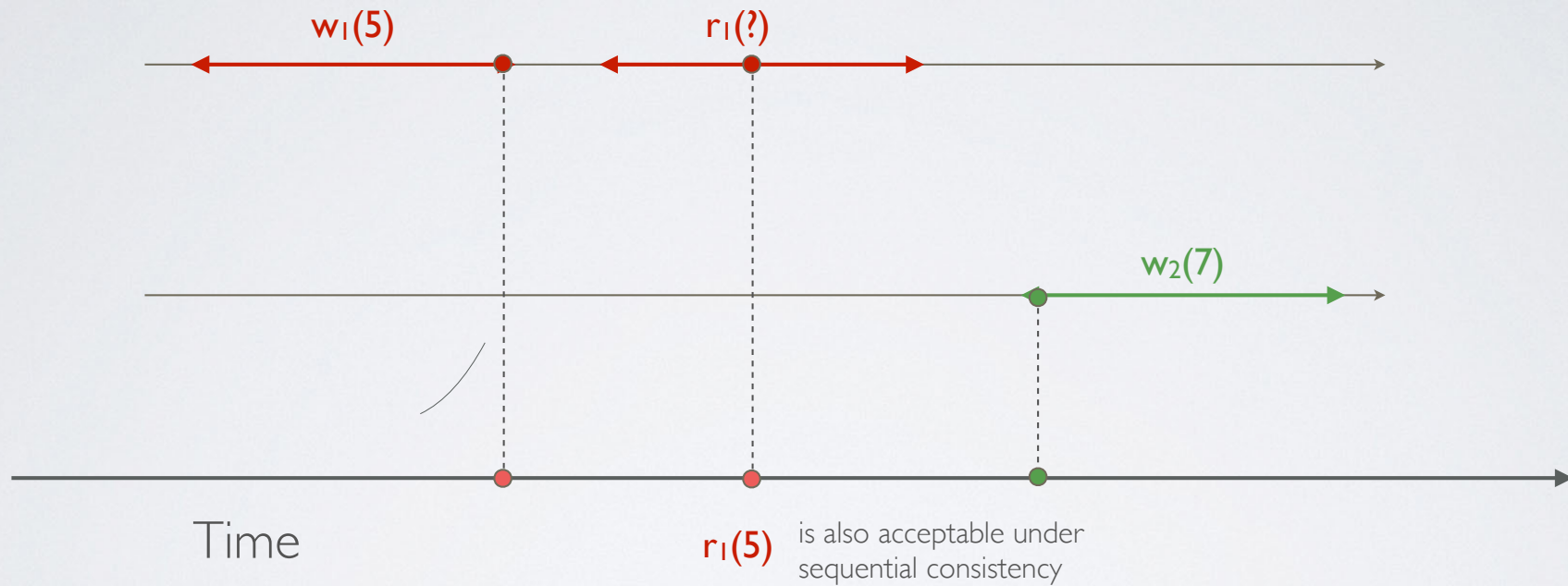


EXAMPLE



but under sequential consistency...

EXAMPLE



...this is a valid permutation of the same history

THINK GLOBAL, ACT LOCAL

- A property P of a concurrent system is

local

if the system satisfies P whenever each individual object satisfies P

- ▶ Given two objects o_1 and o_2 each satisfying P , the composite object $[o_1, o_2]$ satisfies P

LINEARIZABILITY IS A LOCAL PROPERTY

Theorem

A history H is linearizable iff

for each object o , H restricted to the operations in o is linearizable

- Because linearizability is a local property, objects can be implemented independently

What about
sequential consistency?

THE CASE OF THE FIFO QUEUE



$\leftarrow P.enq(\bullet) \rightarrow$ $\leftarrow Q.enq(\bullet) \rightarrow$ $\leftarrow P.deq(\bullet) \rightarrow$



$\leftarrow Q.enq(\bullet) \rightarrow$ $\leftarrow P.enq(\bullet) \rightarrow$ $\leftarrow Q.deq(\bullet) \rightarrow$



Time

THE CASE OF THE FIFO QUEUE



$\leftarrow P.enq(\text{yellow}) \rightarrow$ $\leftarrow P.enq(\text{green}) \rightarrow$ $\leftarrow P.deq(\text{yellow}) \rightarrow$



Time 

THE CASE OF THE FIFO QUEUE



$\leftarrow P.enq(\bullet) \rightarrow \quad \leftarrow Q.enq(\bullet) \rightarrow \quad \leftarrow P.deq(\bullet) \rightarrow$



$\leftarrow Q.enq(\bullet) \rightarrow \quad \leftarrow P.enq(\bullet) \rightarrow \quad \leftarrow Q.deq(\bullet) \rightarrow$



Time

THE CASE OF THE FIFO QUEUE



$\leftarrow Q.enq(\bullet) \rightarrow$ $\leftarrow Q.enq(\bullet) \rightarrow$ $\leftarrow Q.deq(\bullet) \rightarrow$

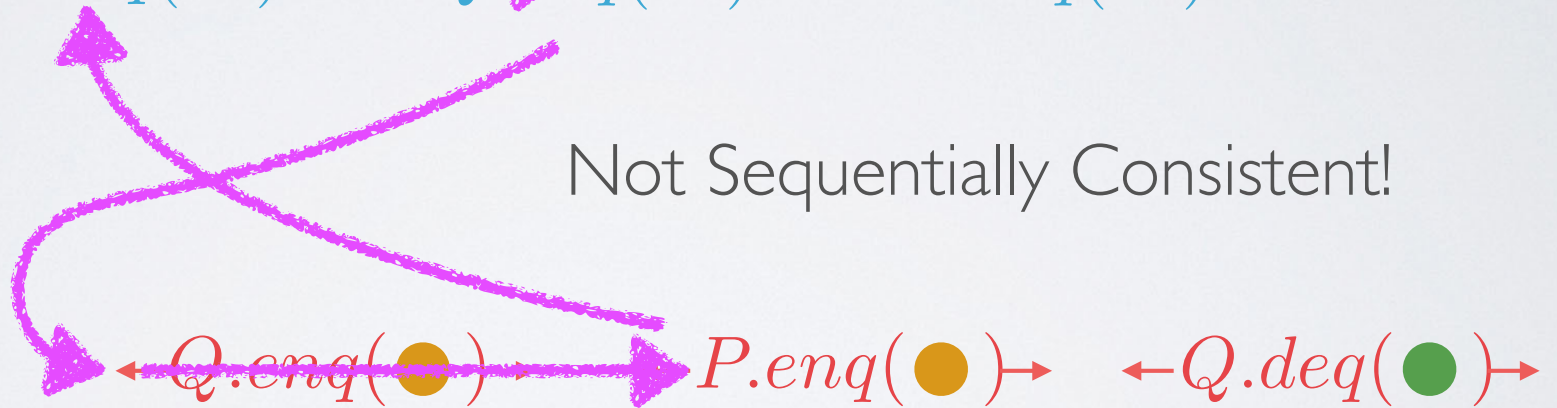


Time 

THE CASE OF THE FIFO QUEUE



$\leftarrow P.enq(\bullet)$ $\leftarrow Q.enq(\bullet)$ $\leftarrow P.deq(\bullet)$



Time

THEOREM

Sequential Consistency Is Not Composable

i.e., an execution involving a collection of sequentially consistent objects may not be sequentially consistent

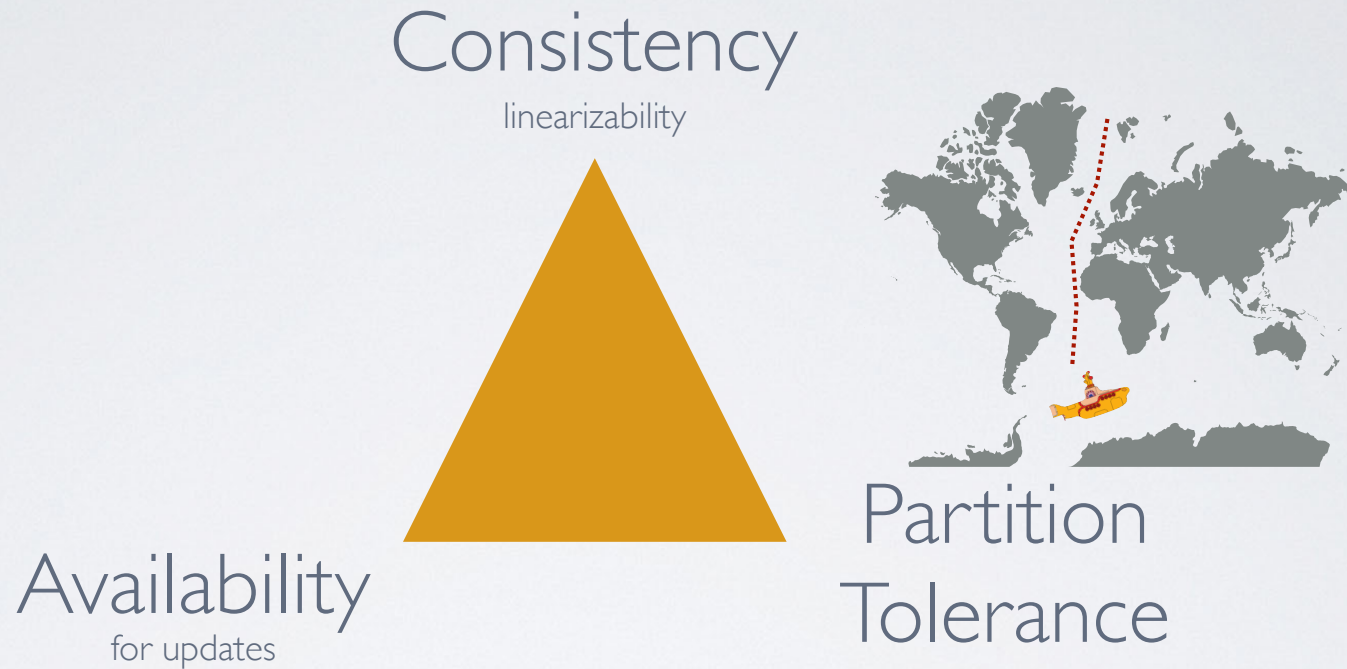
TAKING STOCK

- We can define **linearizability**, a strong, composable notion of correctness for concurrent objects
- We can use consensus to achieve it in a distributed setting

... BUT WHAT ABOUT GEO-REPLICATED SYSTEMS?



THE CAP DILEMMA



Eric Brewer's CAP “Theorem”

“You can have at most two of C, A, and P for any shared data system”

WHAT DOES $A \triangle C_P$ MEAN?

Werner Vogels, CTO Amazon

“An important observation is that in larger distributed-scale systems, network partitions are a given; therefore, **consistency and availability cannot be achieved at the same time.**”

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html



Farewell consistency, we hardly knew ye...



Weak
Consistency

A dramatic landscape photograph featuring a road that splits into two paths, leading through a vast field of golden wheat. The sky is filled with large, dark, and textured clouds, with a bright light source on the left side, creating a high-contrast scene. In the background, there are dark silhouettes of hills or mountains.

Distributed Systems

Databases

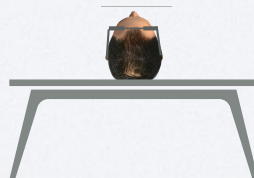
WHAT ABOUT DATABASES?

TRANSACTIONS TAKE TIME!

A CLASSIC HORROR STORY

Ease
of
Programming

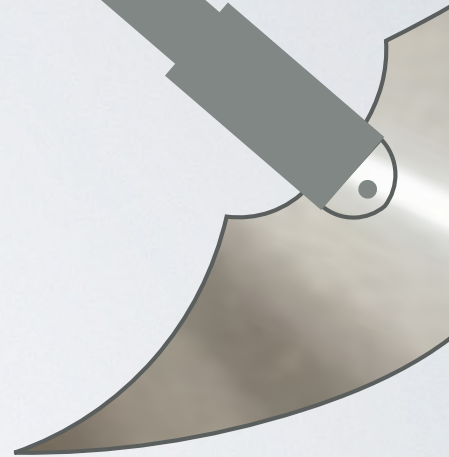
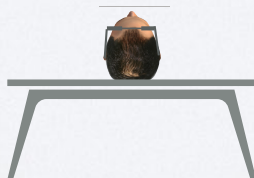
Performance



A CLASSIC HORROR STORY

Ease
of
Programming

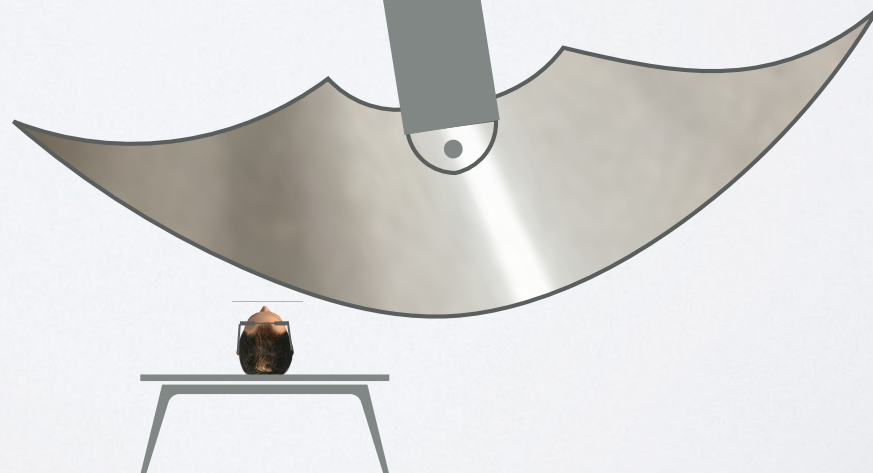
Performance



A CLASSIC HORROR STORY

Ease
of
Programming

Performance



THE FANTASTIC FOUR



Atomicity

Either all changes to the state happen, or none

Consistency

A transaction is a correct transformation of the state

Isolation

Regulates which states generated in executing T are visible to transactions executing concurrently to T

Durability

Once a transaction commits, its state changes survive failures

THE FANTASTIC FOUR



Isolation

Serializability

Though transactions execute concurrently, it appears to each T as if every other transaction executed either before or after T

Strict Serializability

The total order in which transactions are serialized is consistent with real time: if T1 commits before T2 starts, then T1 appears before T2 in the serial order.

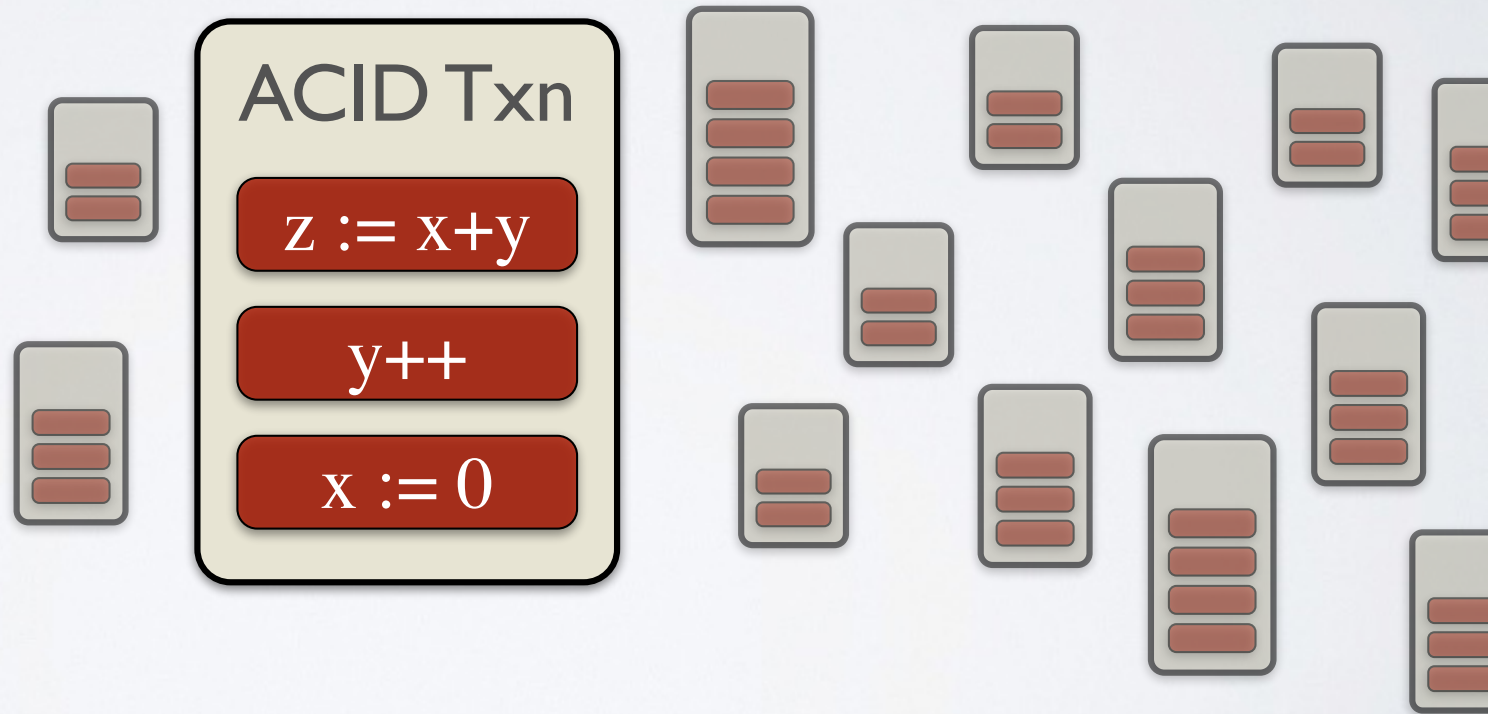
ACID TRANSACTIONS: SIMPLE AND POWERFUL

Atomicity

Consistency

Isolation

Durability



WEAKER FLAVORS OF ISOLATION

It-That-Shall-Not-Be-Named

dirty writes - transaction modifies item previously modified by undecided transaction

Read-Uncommitted

dirty reads: one transaction may see uncommitted state of another transaction

Read-Committed

no dirty reads or writes, but allows for **non-repeatable reads**

Repeatable Reads

non-repeatable range reads

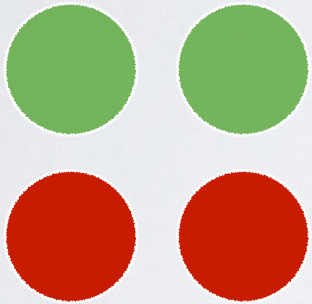
Snapshot Isolation

none of the above, but **write skew**



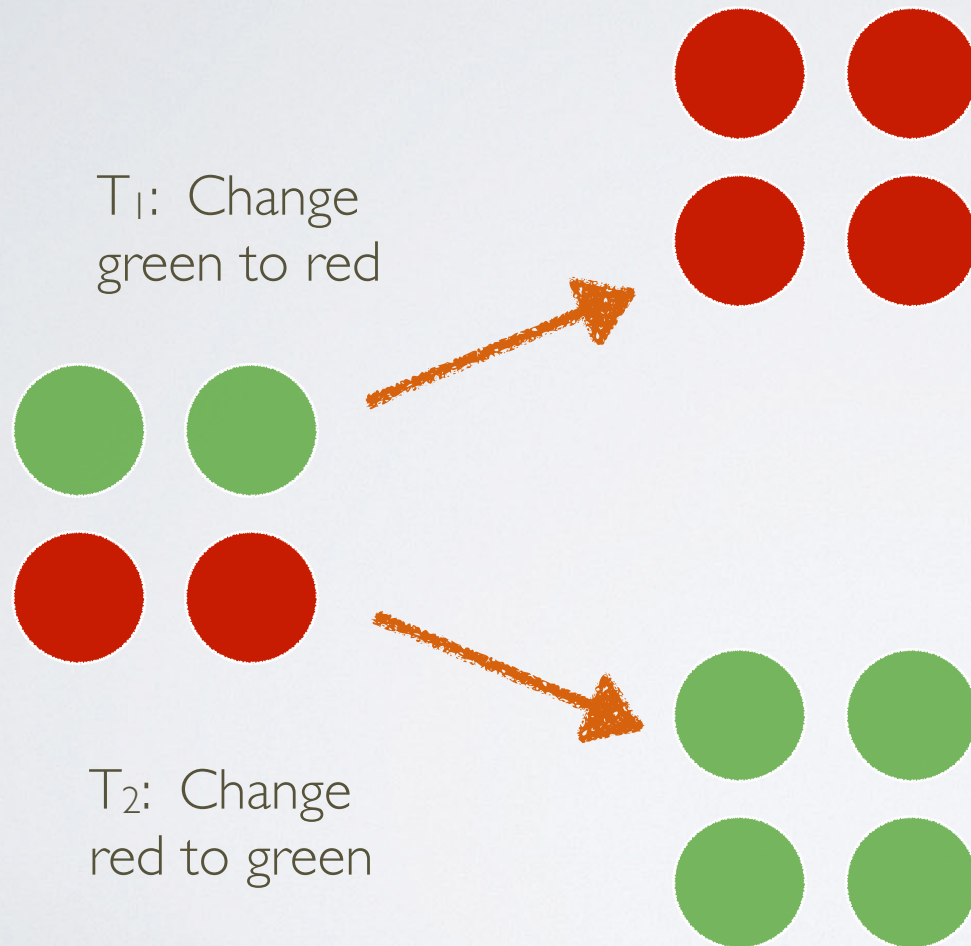
WRITE SKEW ANOMALY

T_1 : Change
green to red

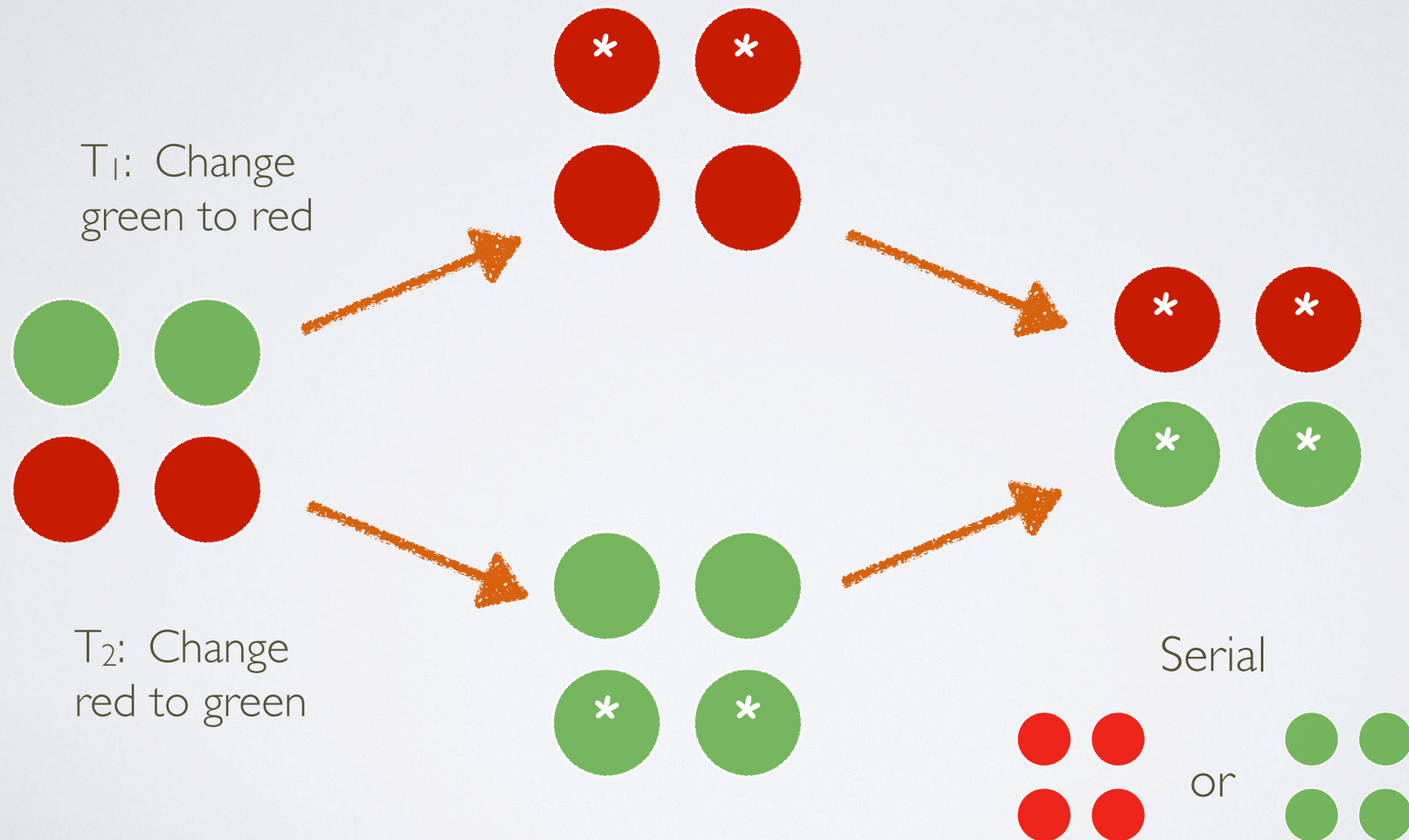


T_2 : Change
red to green

WRITE SKEW ANOMALY



WRITE SKEW ANOMALY



PERFORMANCE VIA WEAKER ISOLATION GUARANTEES

Database System	Default Isolation	Strongest Isolation
MySQL Cluster	Read Committed	Read Committed
SAP HANA	Read Committed	Snapshot Isolation
Google Spanner	Serializability	Serializability
VoltDB	Serializability	Serializability
Oracle 12C	Read Committed	Snapshot Isolation
MemSQL	Read Committed	Read Committed
SQL Server	Read Committed	Serializability
Postgres	Read Committed	Serializability



Weak
Consistency

EVENTUAL CONSISTENCY

(A.K.A. NO CONSISTENCY (RVR DIXIT))

- Replicas are guaranteed to **converge**
 - ▶ updates performed at one replica are eventually seen by all others
 - ▶ if no more updates, replicas eventually reach the same state

If no new updates are made to an object, eventually all accesses will return its last updated value

GEO-REPLICATED SYSTEMS

- Facebook, Twitter, Amazon aim for **ALPS**
 - ▶ **A**vailability
 - ▶ low **L**atency
 - ▶ **P**artition tolerance
 - ▶ **S**calability
- What about consistency?
 - ▶ Tension (you guessed it) between performance and ease of programming

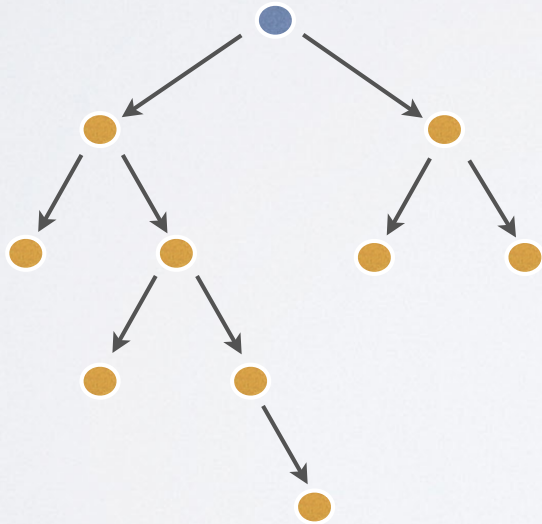


A faded, comic book-style illustration of three women. The woman on the left is shown in profile, holding a cigarette. The woman in the center has long brown hair and is wearing a green polka-dot top. The woman on the right is smiling and looking towards the center. The word "GOSSIP" is centered over the image in a large, black, sans-serif font.

GOSSIP

DATA DISSEMINATION

- Want efficiency, robustness, speed, scale
- Tree distribution is efficient...

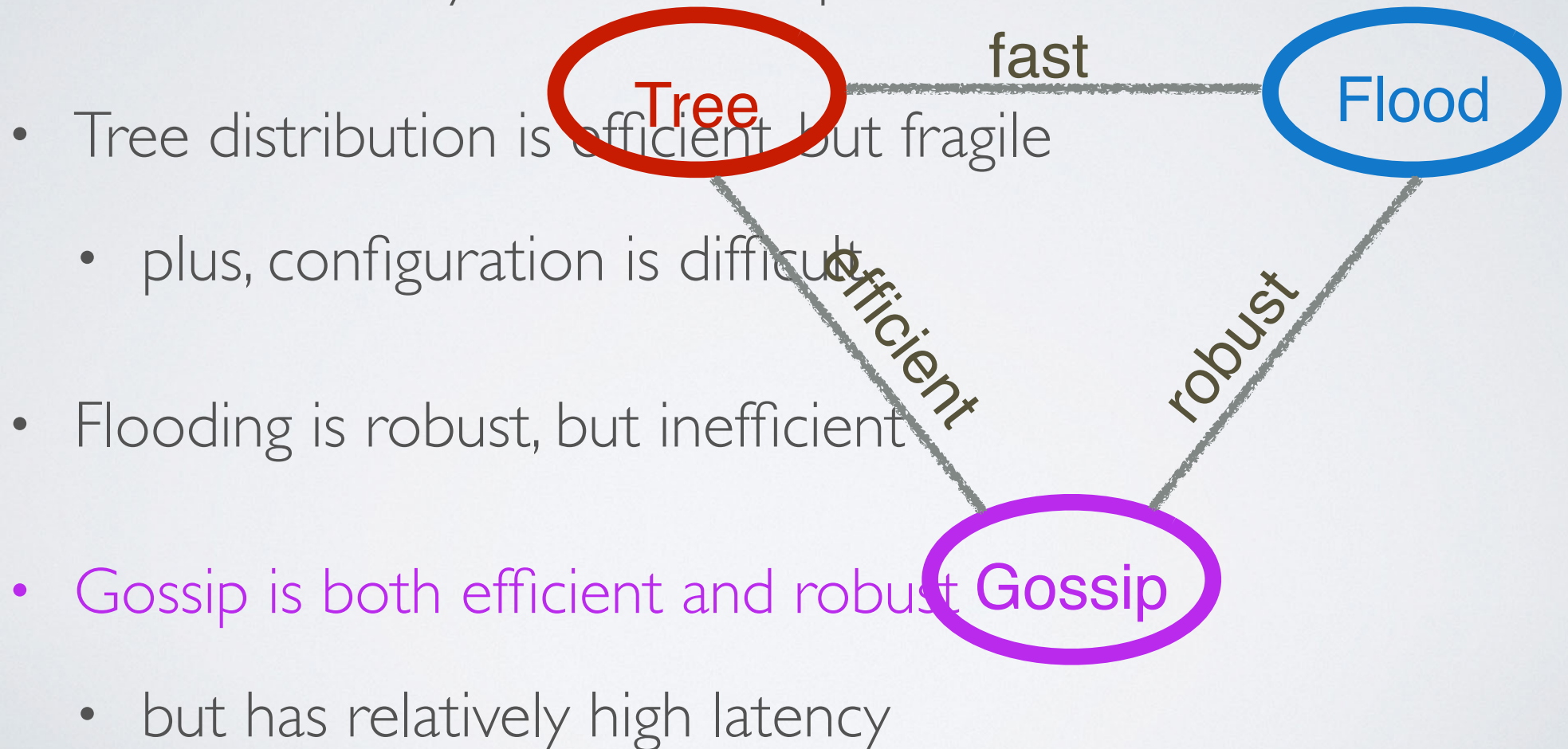


DATA DISSEMINATION

- Want efficiency, robustness, speed, scale
- Tree distribution is efficient, but fragile
 - plus, configuration is difficult

DATA DISSEMINATION

- Want efficiency, robustness, speed, scale



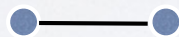
HISTORY

- Gossips and Telephones (Baker, Shostack 1972)
 - There are n ladies, each of whom knows some item of gossip not known to the others. They communicate by telephone, and whenever one lady calls another, they tell each other all that they know at the time. How many calls are required before each gossip knows everything?

$$n = 1$$

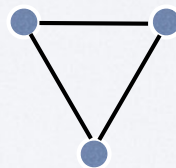
$$0$$

$$n = 2$$



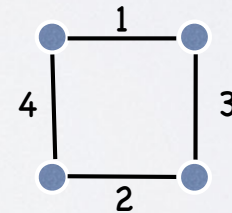
$$1$$

$$n = 3$$



$$3$$

$$n = 4$$



$$4$$

$$n > 4 : 2n - 4$$

HISTORY

- Gossips and Telephones (Baker, Shostack 1972)
- Grapevine/Clearinghouse Directory Service (Demers et al., Xerox Park, 1987)
- Refdbms (Golding, UCSC 1993)
- Bayou (Terry et al., Xerox PARC 1995)
- Bimodal Multicast (Birman et al., Cornell 1998)
- Astrolabe (Van Renesse, Cornell 1999)

GOSSIP PROTOCOL

Active Thread (P)

Forever

selectPeer(&Q)

selectToSend(&bufs)

sendTo(Q, bufs)

receiveFrom(Q, &bufr)

selectToKeep(cache, buf_r)

processData(cache)

Passive Thread (Q)

Forever

receiveFromAny(&P, &buf_r)

selectToSend(&bufs)

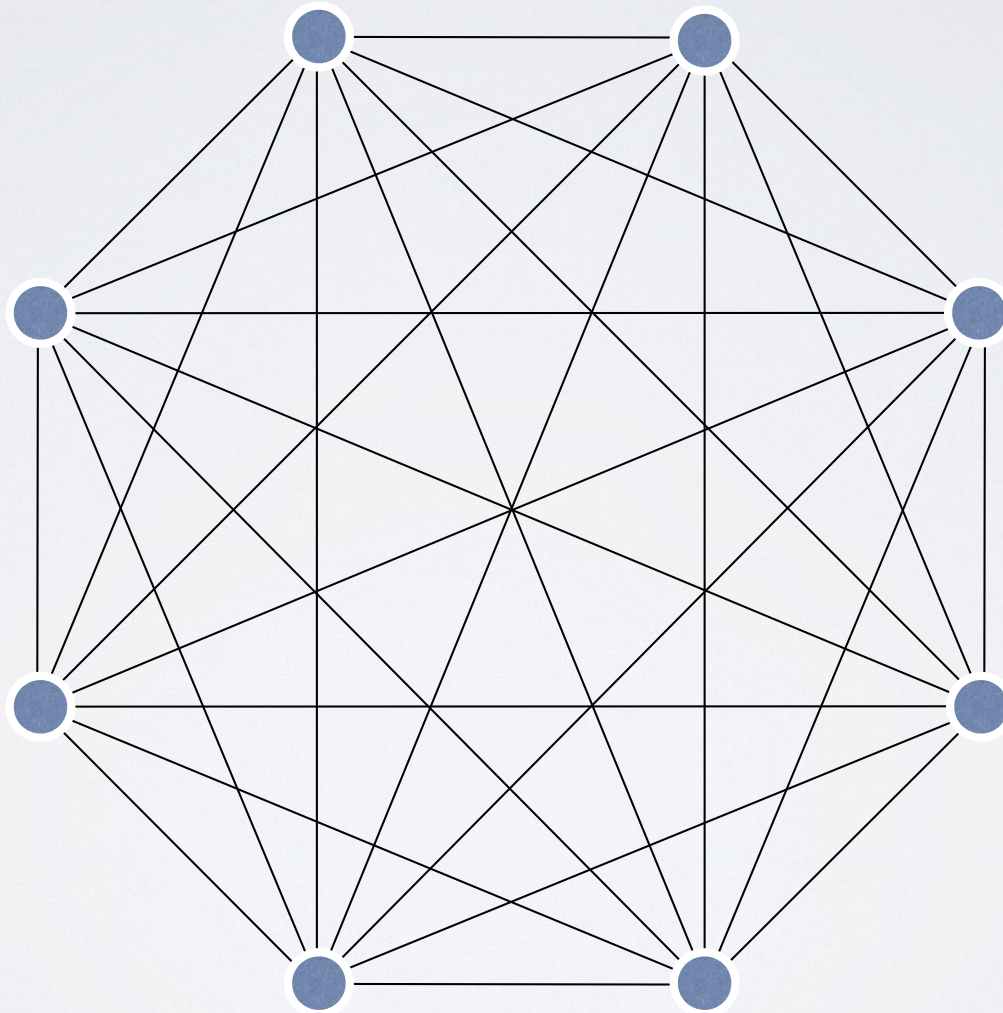
sendTo(P, bufs)

selectToKeep(cache, buf_r)

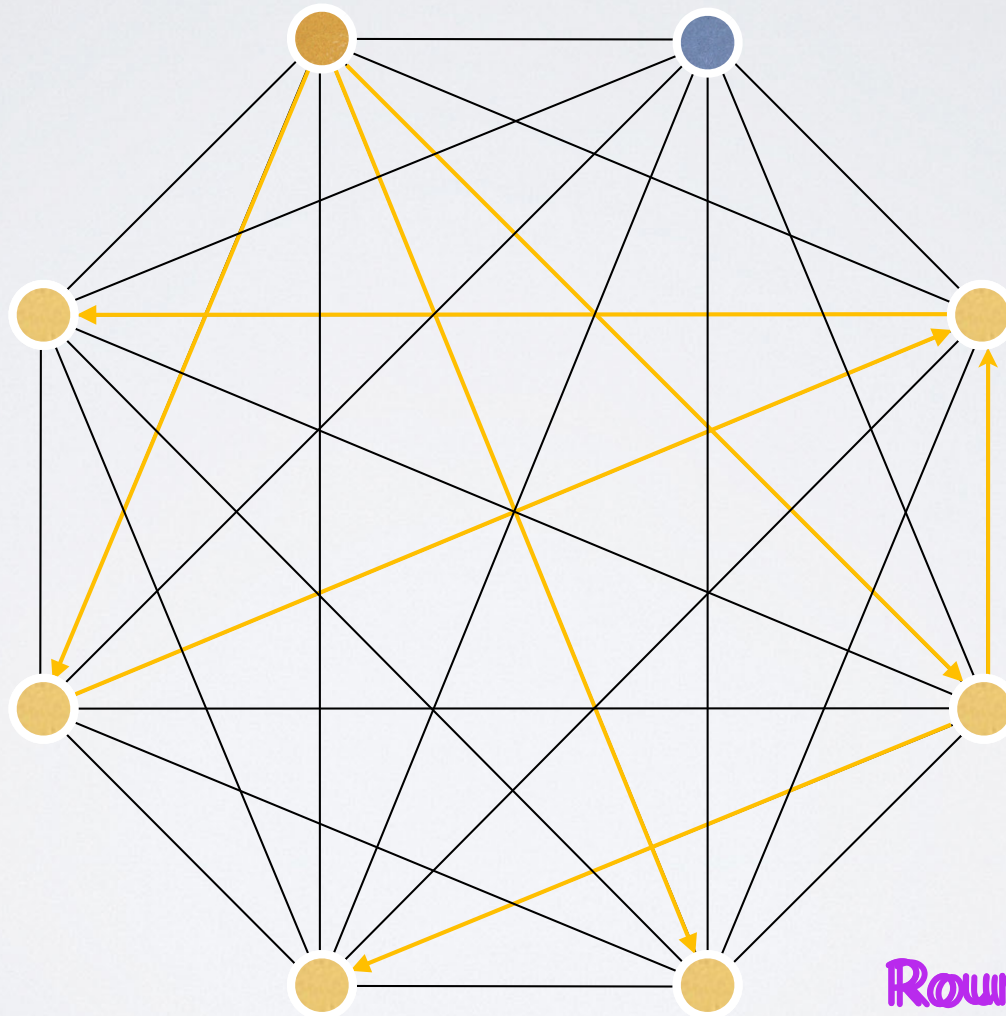
processData(cache)



PSST! PSST!

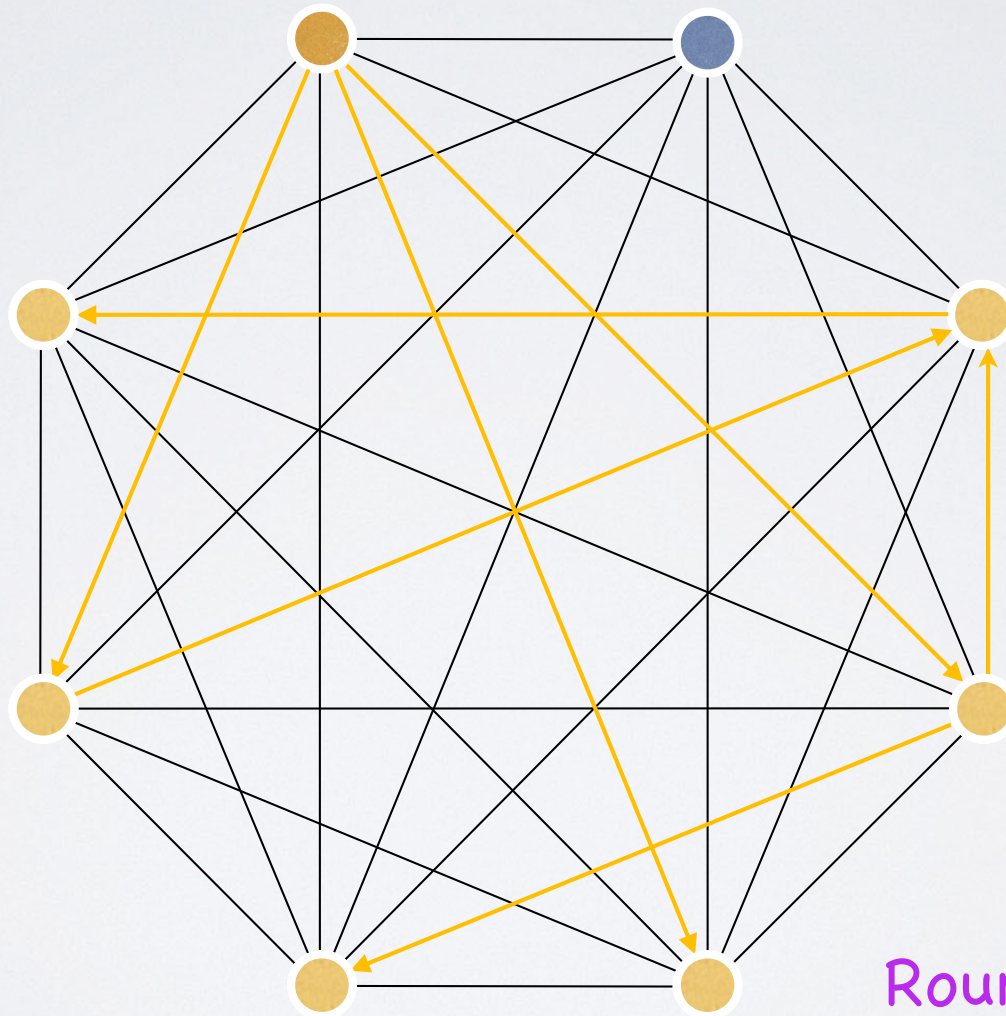


PSST! PSST!



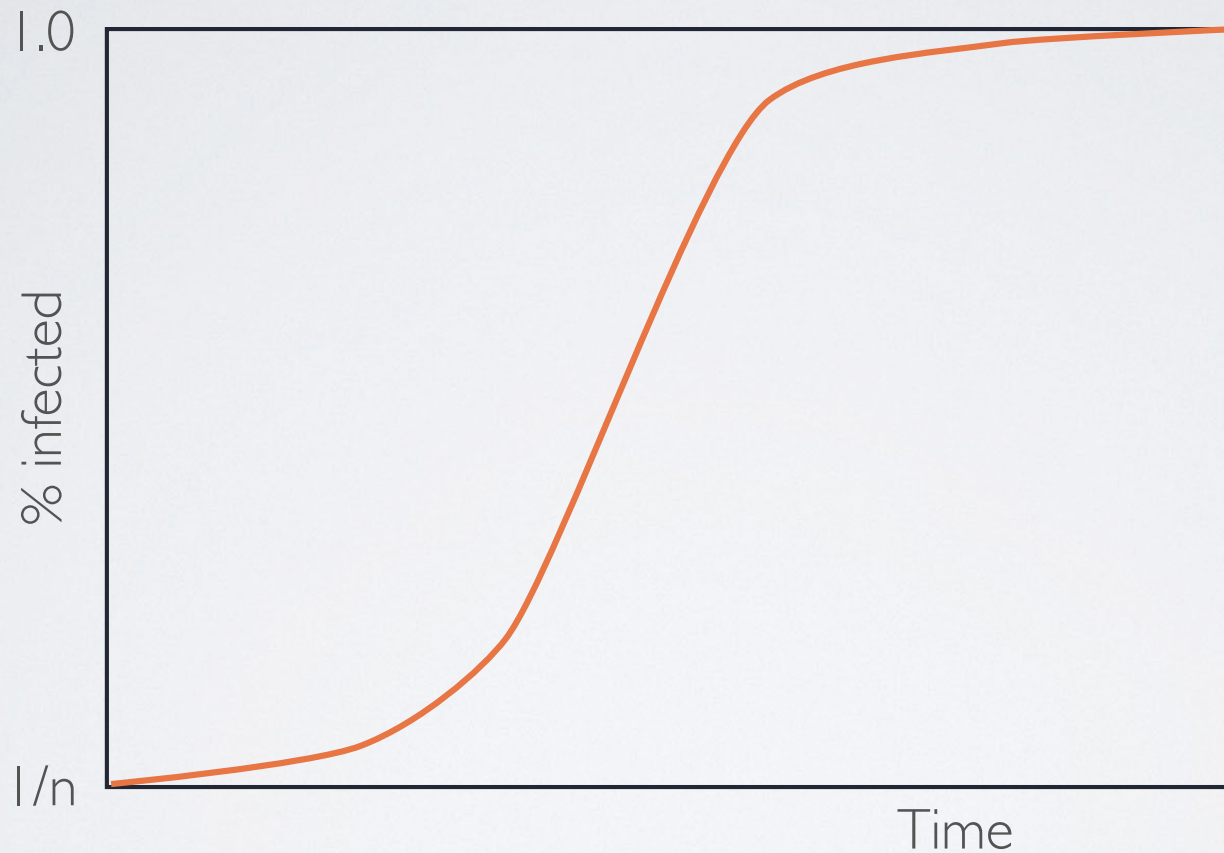
Round 3: 4

PSST! PSST!



Round 3: 7

GOSSIP PROPAGATION TIME



Time to complete “infection”: $O(\log n)$

ANTI-ENTROPY

- Gossip scheme with monotonic merge
 - Merges received state and local state such that new state includes both local and gossiped state
- Corresponding protocol: simple epidemic

HOW FAST DOES GOSSIP REALLY SPREAD?

- Epidemic theory (Bailey 1957)
 - fixed population of size n
 - homogeneous spreading
 - anybody can infect anyone else with equal probability
 - Assume
 - k members are already infected
 - infection occurs in rounds

UNCLEAN!

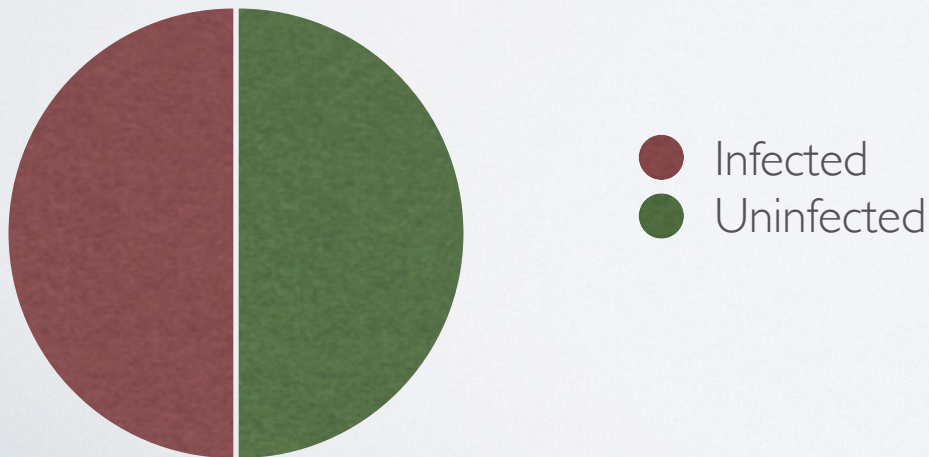
- What is the probability $P_{\text{infect}}(k, n)$ that a particular uninfected member is infected in a round if k are already infected?

$$\begin{aligned} P_{\text{infect}}(k, n) &= 1 - P(\text{nobody infects member}) \\ &= 1 - (1 - 1/n)^k \end{aligned}$$

$$\begin{aligned} E(\# \text{ of newly infected members}) &= \\ (n - k) \times P_{\text{infect}}(k, n) \end{aligned}$$

RATE OF SPREAD

- Two phases
 - Phase 1: $1 \rightarrow n/2$ - fraction of infected grows quickly
 - Phase 2: $n/2 \rightarrow n$ - fraction of uninfected declines quickly
- For large n , $P_{\text{infect}}(n/2, n) \approx 1 - (1 - e)^{0.5} \approx 0.4$



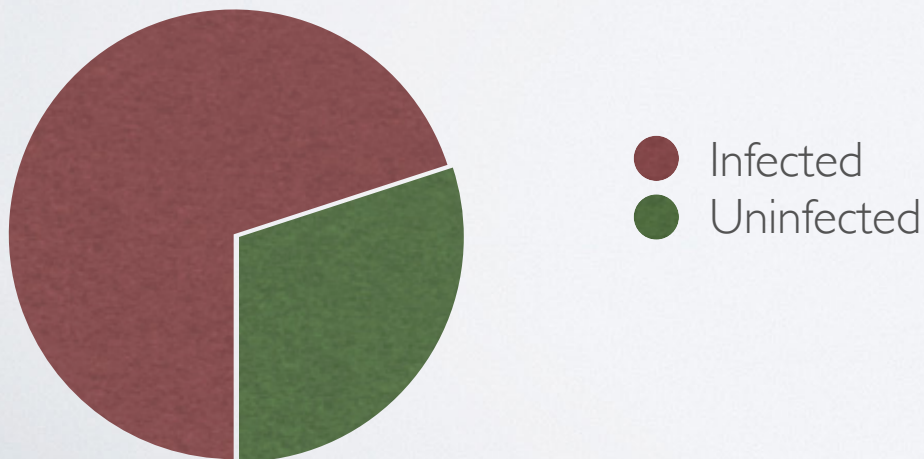
RATE OF SPREAD

- Two phases
 - Phase 1: $1 \rightarrow n/2$ - fraction of infected grows quickly
 - Phase 2: $n/2 \rightarrow n$ - fraction of uninfected declines quickly
- For large n , $P_{\text{infect}}(n/2, n) \approx 1 - (1 - e)^{0.5} \approx 0.4$



RATE OF SPREAD

- Two phases
 - Phase 1: $1 \rightarrow n/2$ - fraction of infected grows quickly
 - Phase 2: $n/2 \rightarrow n$ - fraction of uninfected declines quickly
- For large n , $P_{\text{infect}}(n/2, n) \approx 1 - (1 - e)^{0.5} \approx 0.4$



WHO'S USING EC?

- Domain Name Service (DNS)
- Facebook/Meta
- Amazon
- Twitter
- ...
- Bayou (1995)



BAYOU

Terry et al, SOSP '95

- Replicas keep ordered log of updates reflected in their state
- They gossip entries in their log
- If no more updates, replica logs (states) eventually converge
- But Bayou gives you more:



BAYOU

Terry et al, SOSP '95

- Replicas keep ordered log of updates reflected in their state
- They gossip entries in their log
- If no more updates, replica logs (states) eventually converge
- But Bayou gives you more:

“If the log of R_i contains an update first performed on R_j , then the log of R_i also contains all the writes accepted by R_j prior to w .”

If a replica sees an update w , it has seen all updates that causally precede w !

CAUSAL CONSISTENCY

Updates that are causally related should be seen by all replicas in the same order. Concurrent updates may be seen by different replicas in different orders (Hutto & Ahamad, 1990)

Two operations a and b are **causally related** ($a \rightarrow b$) if

1. The same client executes first a then b
2. b reads the value written by a
3. There exists an operation a' such that $a \rightarrow a'$ and $a' \rightarrow b$

WHY CAUSAL CONSISTENCY?



Prof. Natacha Crooks



1. Receives selfie (update 2) then defriend request (update 1)
2. Whoops.

1. meditates unspeakable crime
2. defriends me (update 1)
3. posts selfie (update 2) while engaging in unspeakable crime

SEQUENTIALLY CONSISTENT?



$\leftarrow w(x)a \rightarrow$

$\leftarrow w(x)c \rightarrow$

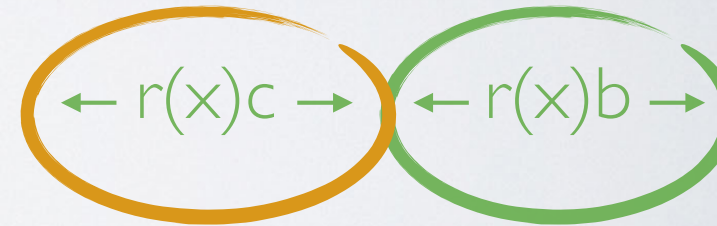


$\leftarrow r(x)a \rightarrow$

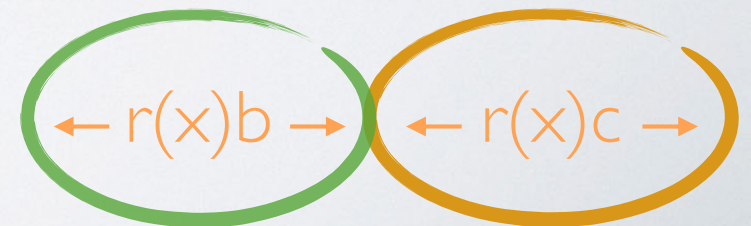
$\leftarrow w(x)b \rightarrow$



$\leftarrow r(x)a \rightarrow$



$\leftarrow r(x)a \rightarrow$



CAUSALLY CONSISTENT?



$\leftarrow w(x)a \rightarrow$ $\xrightarrow{\hspace{1cm}}$ $\leftarrow w(x)c \rightarrow$



$\leftarrow r(x)a \rightarrow$ $\xrightarrow{\hspace{1cm}}$ $\leftarrow w(x)b \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)c \rightarrow$ $\leftarrow r(x)b \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)b \rightarrow$ $\leftarrow r(x)c \rightarrow$

CAUSALLY CONSISTENT?



$w(x)a$ $r(x)a$ $w(x)b$ $w(x)c$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)c \rightarrow$ $\leftarrow r(x)b \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)b \rightarrow$ $\leftarrow r(x)c \rightarrow$

CAUSALLY CONSISTENT?



$\leftarrow w(x)a \rightarrow$

$\leftarrow w(x)c \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow w(x)b \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)c \rightarrow \quad \leftarrow r(x)b \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)b \rightarrow \quad \leftarrow r(x)c \rightarrow$

CAUSALLY CONSISTENT?



$\leftarrow r(x)a \rightarrow$



$w(x)a$ $r(x)a$ $w(x)c$ $r(x)c$ $w(x)b$ $r(x)b$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)b \rightarrow$ $\leftarrow r(x)c \rightarrow$

CAUSALLY CONSISTENT?



$\leftarrow w(x)a \rightarrow$

$\leftarrow w(x)c \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow w(x)b \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)c \rightarrow \quad \leftarrow r(x)b \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)b \rightarrow \quad \leftarrow r(x)c \rightarrow$

CAUSALLY CONSISTENT?



$\leftarrow r(x)a \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)c \rightarrow \quad \leftarrow r(x)b \rightarrow$



$w(x)a \quad r(x)a \quad w(x)b \quad r(x)b \quad w(x)c \quad r(x)c$

CAUSALLY CONSISTENT?



$\leftarrow w(x)a \rightarrow$

$\leftarrow w(x)c \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow w(x)b \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)c \rightarrow \quad \leftarrow r(x)b \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)b \rightarrow \quad \leftarrow r(x)c \rightarrow$

CAUSALLY CONSISTENT?



$\leftarrow w(x)a \rightarrow$



$\leftarrow w(x)c \rightarrow$



$\leftarrow r(x)a \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow w(x)b \rightarrow$

$\leftarrow r(x)c \rightarrow$

$\leftarrow r(x)b \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)b \rightarrow$

$\leftarrow r(x)c \rightarrow$

CAUSALLY CONSISTENT?



$\leftarrow r(x)a \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)c \rightarrow \quad \leftarrow r(x)b \rightarrow$



$w(x)a \quad r(x)a \quad w(x)b \quad r(x)b \quad w(x)c \quad r(x)c$

CAUSALLY CONSISTENT?



$\leftarrow w(x)a \rightarrow$

$\leftarrow w(x)c \rightarrow$



$\leftarrow r(x)a \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow w(x)b \rightarrow$

$\leftarrow r(x)c \rightarrow$

$\leftarrow r(x)b \rightarrow$



$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)b \rightarrow$

$\leftarrow r(x)c \rightarrow$

CAUSALLY CONSISTENT?



$\leftarrow r(x)a \rightarrow$

Does not meet the
sequential specification!



$w(x)a$ $r(x)a$ $w(x)b$ $w(x)c$ $r(x)c$ $r(x)b$



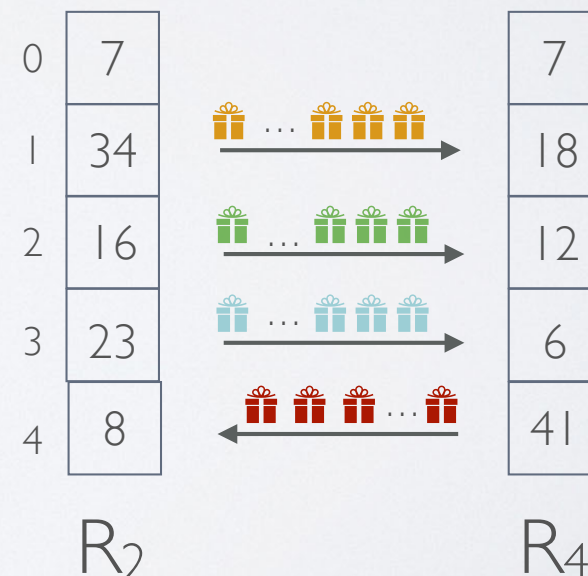
$\leftarrow r(x)a \rightarrow$

$\leftarrow r(x)b \rightarrow$ $\leftarrow r(x)c \rightarrow$

CAUSAL CONSISTENCY IN BAYOU

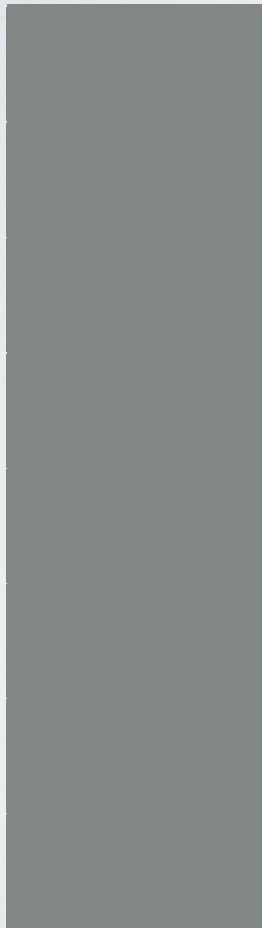
- When replica R_i receives an update from a client, it **logs** it by assigning to it a timestamp (logical time $_i$, i)
- Replicas learn which updates they need to exchange by comparing version vectors!

- Each replica R_i maintains a **version vector** $R_i.V[]$
 - ▶ $R_i.V[j] =$ highest timestamp of any write logged by R_j and known to R_i



SHARDING FOR SCALABILITY

Storage
Tier



DC₁

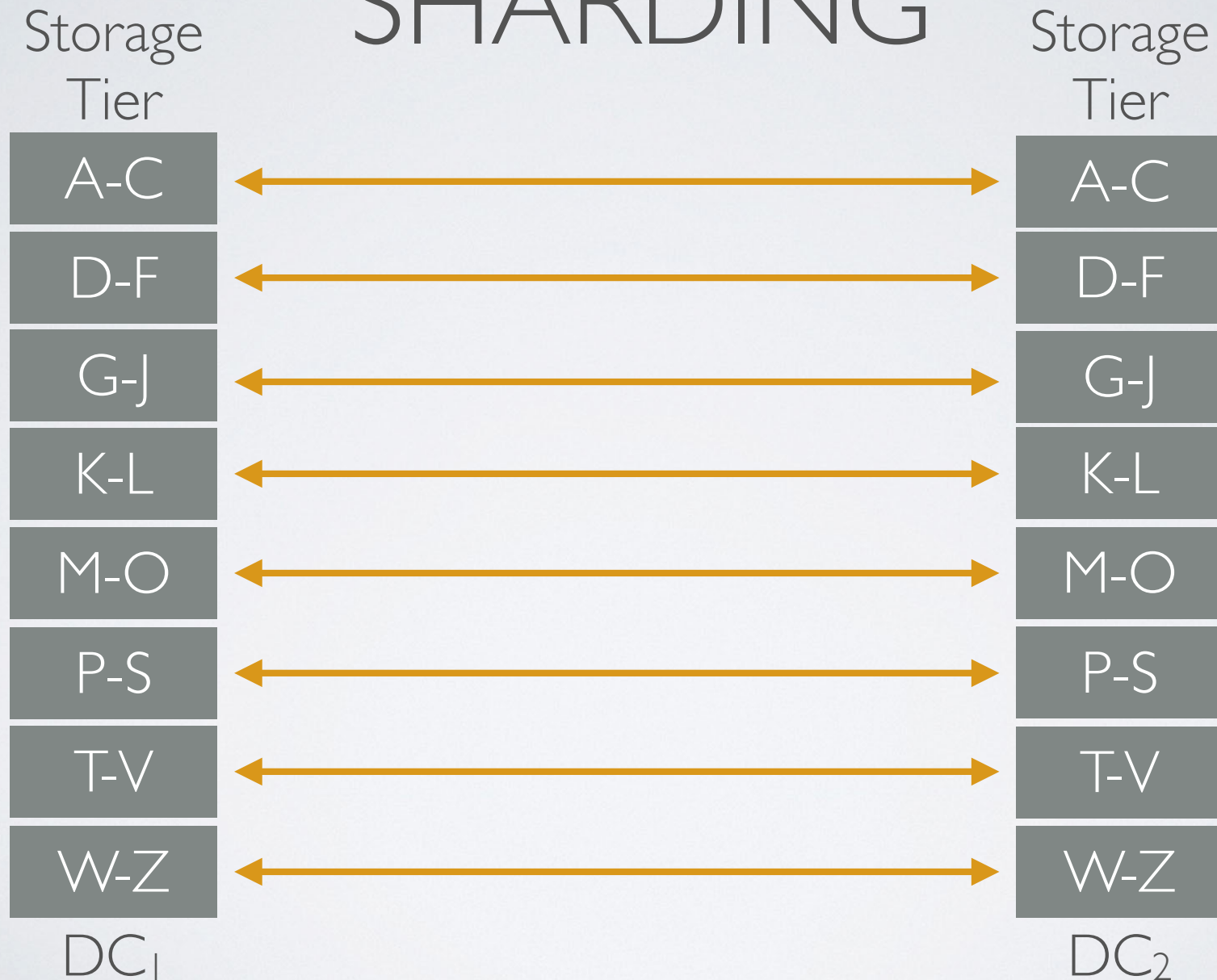
Storage
Tier



DC₂



SCALABILITY THROUGH SHARDING



BAYOU CAUSAL CONSISTENCY DOES NOT SCALE!

- Log requires **one serialization point** per DC, so DCs' states can converge
 - ✓ either causal dependencies only exist between keys stored on a single node
 - ✓ or some node must serialize across all nodes

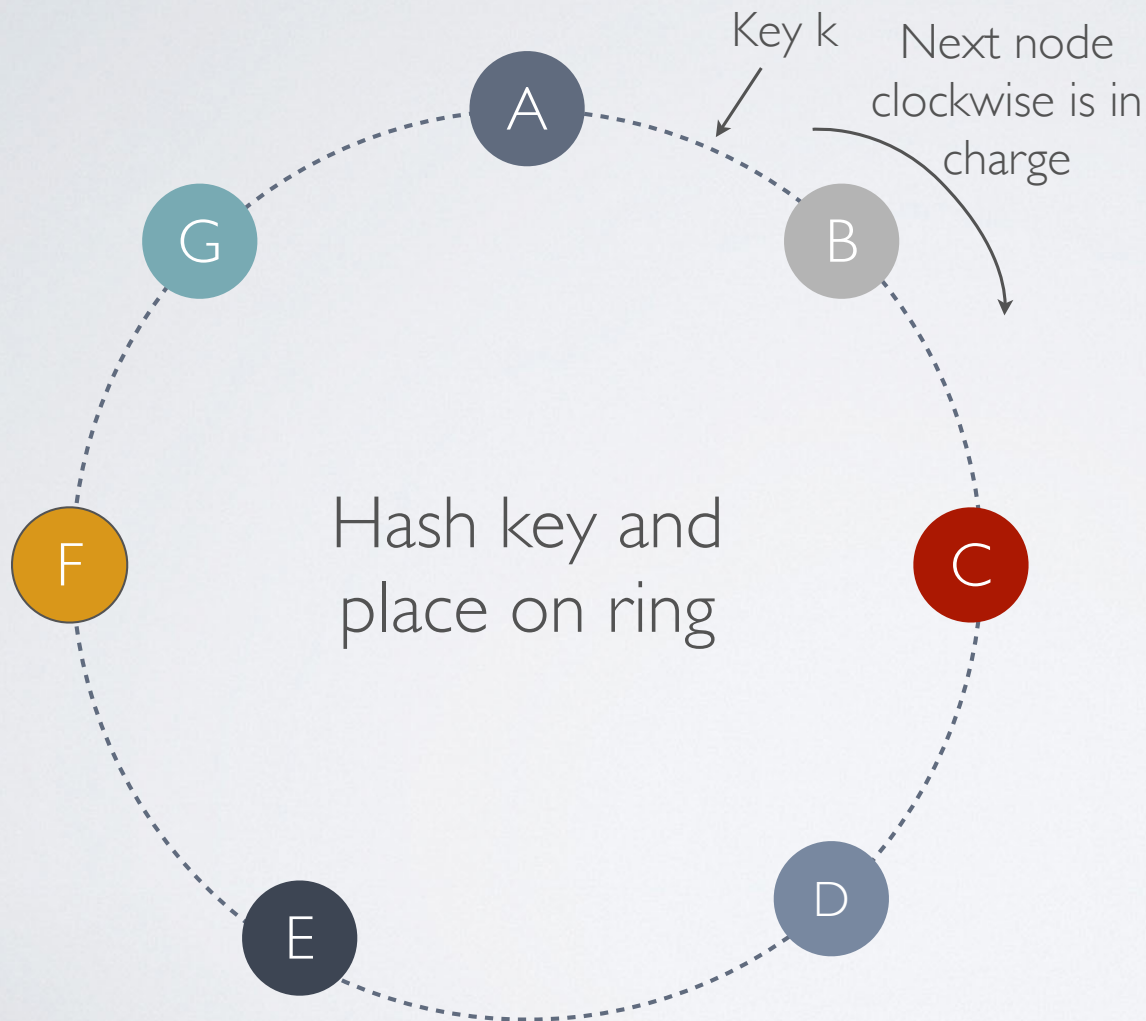
COPS: CLUSTER OF ORDER PRESERVING SERVERS

Loyd et al.,
SOSP '11

- Each datacenter is linearizable
 - local get/put operations on each shard are linearizable, and, since linearizability is composable, the whole system (within a DC) is linearizable
 - linearizability can be achieved with low latency and no fear of partitions
- Operations are replicated *across* datacenters asynchronously
 - as in Liskov's *Lazy Replication*



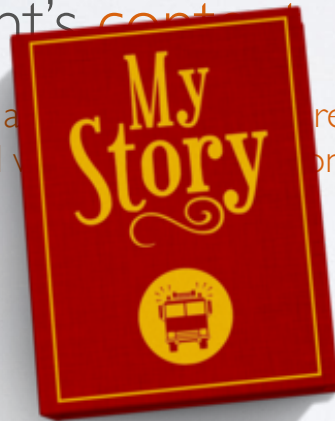
SHARDING THROUGH CONSISTENT HASHING



- Multiple servers replicate each shard inside a given DC (e.g., using chain replication)

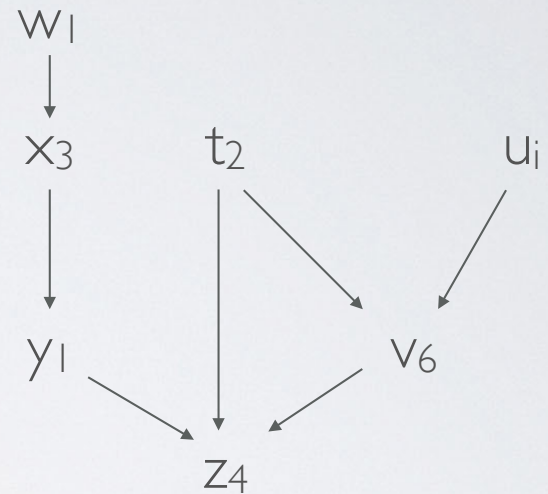
TOWARDS SCALABLE CAUSAL CONSISTENCY

- Replace serialization with distributed verification
- On get, returned $\langle \text{version}, \text{value} \rangle$ is stored in the client's context
 - In principle, context includes a log of all read or written in client's session and version!



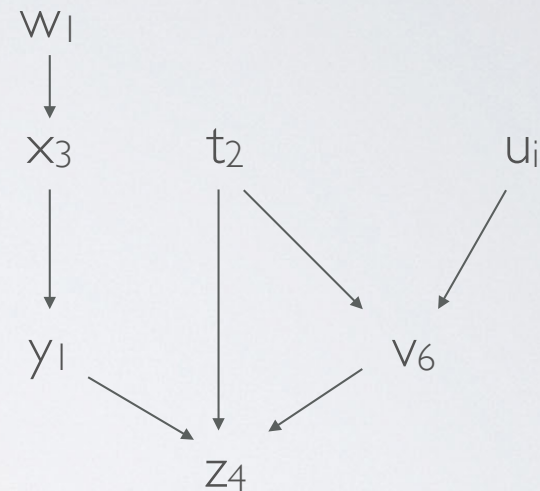
TOWARDS SCALABLE CAUSAL CONSISTENCY

- Replace serialization with **distributed verification**
- On get, returned $\langle \text{version}, \text{value} \rangle$ is stored in the client's **context**
 - In principle, context includes **all values previously read or written in client's session and what they depend on!**



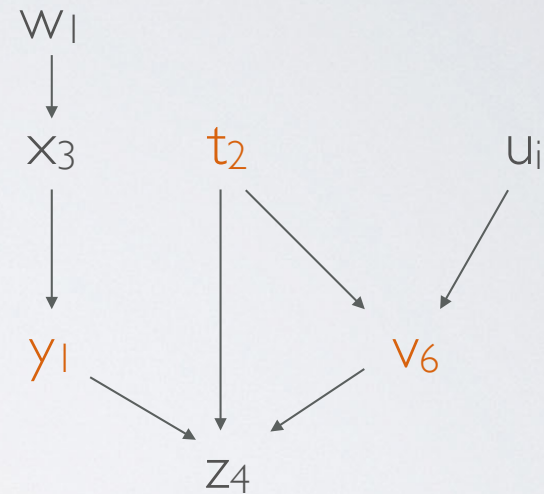
TOWARDS SCALABLE CAUSAL CONSISTENCY

- Replace serialization with **distributed verification**
- On get, returned $\langle \text{version}, \text{value} \rangle$ is stored in the client's **context**
 - In principle, context includes **all values previously read or written in client's session and what they depend on!**
- On a put, client includes (and replicates) its “nearest dependencies” from context...



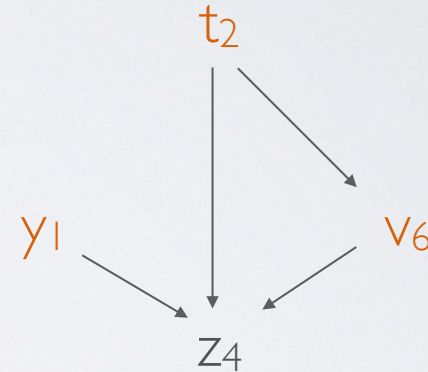
TOWARDS SCALABLE CAUSAL CONSISTENCY

- Replace serialization with **distributed verification**
- On get, returned $\langle \text{version}, \text{value} \rangle$ is stored in the client's **context**
 - In principle, context includes **all values previously read or written in client's session and what they depend on!**
- On a put, client includes (and replicates) its “nearest dependencies” from context...



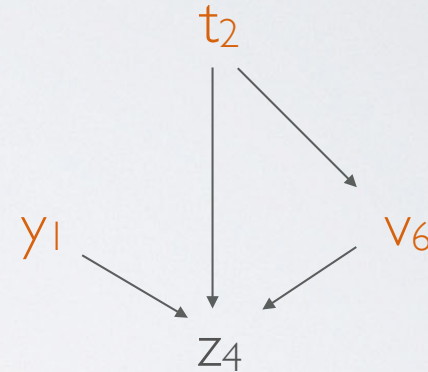
TOWARDS SCALABLE CAUSAL CONSISTENCY

- Replace serialization with distributed verification
- On get, returned $\langle \text{version}, \text{value} \rangle$ is stored in the client's context
 - In principle, context includes all values previously read or written in client's session and what they depend on!
- On a put, client includes (and replicates) its “nearest dependencies” from context... and resets context to the latest put



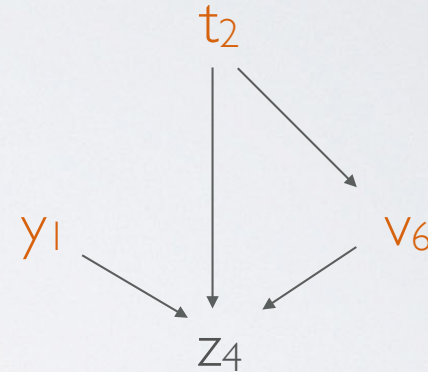
TOWARDS SCALABLE CAUSAL CONSISTENCY

- Replace serialization with distributed verification
- On get, returned $\langle \text{version}, \text{value} \rangle$ is stored in the client's context
 - In principle, context includes all values previously read or written in client's session and what they depend on!
- On a put, client includes (and replicates) its “nearest dependencies” from context... and resets context to the latest put



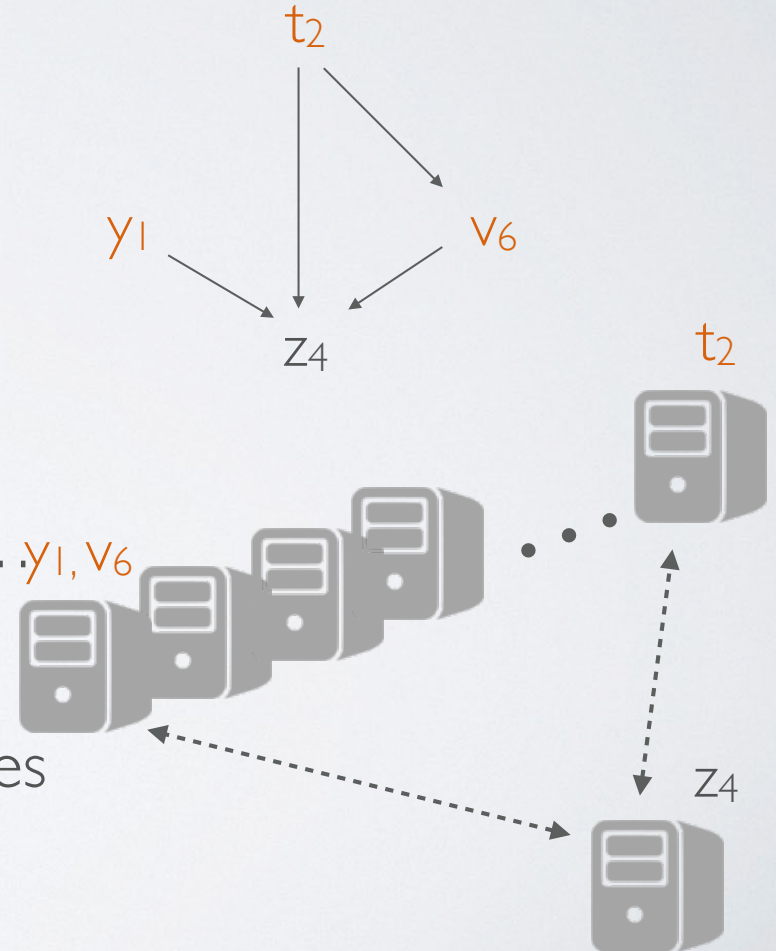
TOWARDS SCALABLE CAUSAL CONSISTENCY

- Replace serialization with **distributed verification**
- On get, returned $\langle \text{version}, \text{value} \rangle$ is stored in the client's **context**
 - In principle, context includes **all values previously read or written in client's session and what they depend on!**
- On a put, client includes (and replicates) its “nearest dependencies” from context... and resets context to the latest put
- Before applying z_4 , remote partition verifies nearest dependencies have already been applied



TOWARDS SCALABLE CAUSAL CONSISTENCY

- Replace serialization with **distributed verification**
- On get, returned $\langle \text{version}, \text{value} \rangle$ is stored in the client's **context**
 - In principle, context includes **all values previously read or written in client's session and what they depend on!**
- On a put, client includes (and replicates) its “nearest dependencies” from context... y_1, v_6 and resets context to the latest put
- Before applying z_4 , remote partition verifies nearest dependencies have already been applied



A dramatic landscape photograph featuring a road that splits into two paths, leading through a vast field of golden wheat. The sky is filled with large, dark, and textured clouds, with a bright light source on the left side. In the background, there are dark silhouettes of hills or mountains. The overall mood is contemplative and suggests a choice or a journey.

Distributed Systems

Databases

WEAKER FLAVORS OF ISOLATION

It-That-Shall-Not-Be-Named

dirty writes - transaction modifies item previously modified by undecided transaction

Read-Uncommitted

dirty reads: one transaction may see uncommitted state of another transaction

Read-Committed

no dirty reads or writes, but allows for **non-repeatable reads**

Repeatable Reads

non-repeatable range reads

Snapshot Isolation

none of the above, but **write skew**



BASE

Basically
Available,
Soft state,
Eventually consistent



BASE

Basically
Available,
Soft state,
Eventually consistent

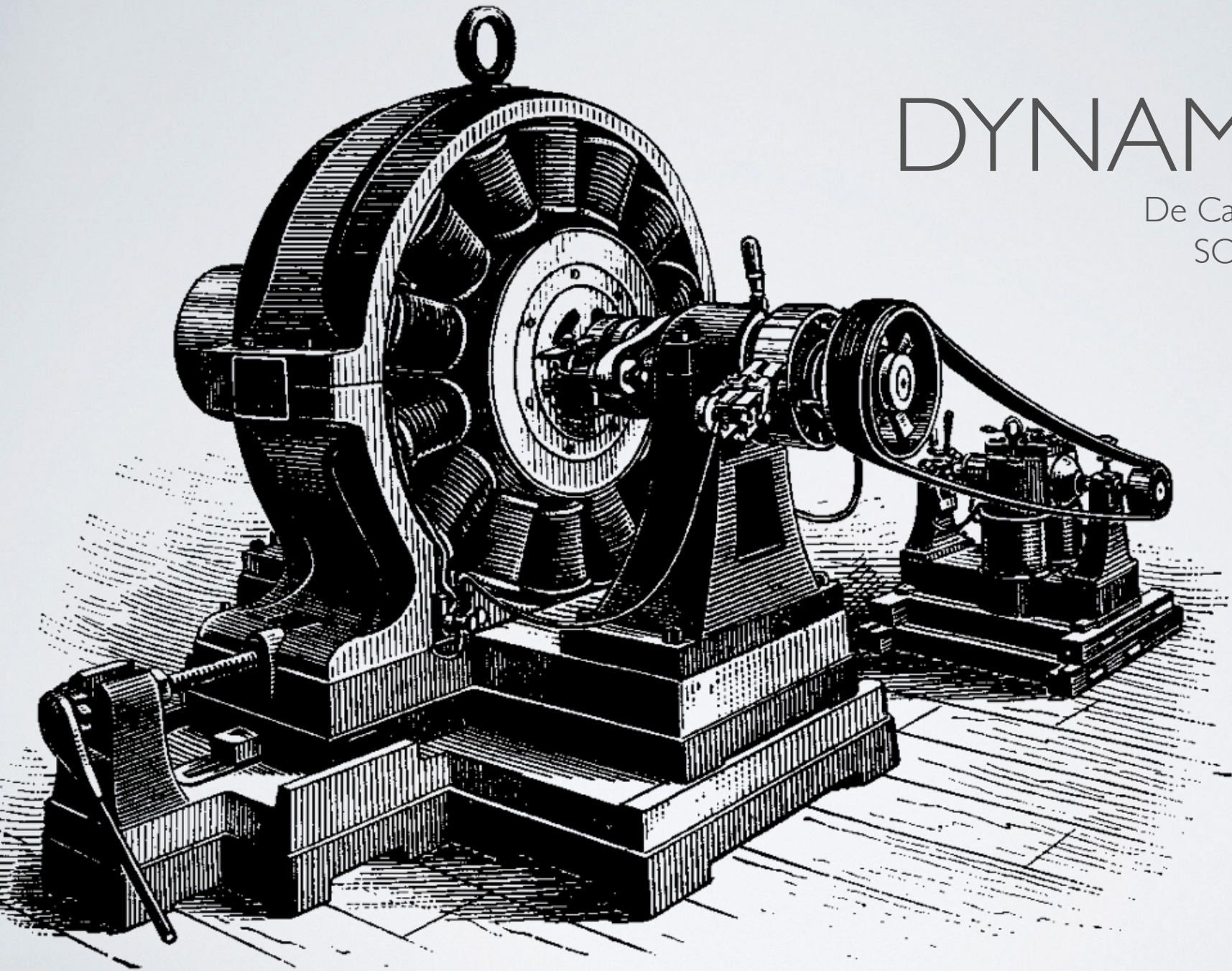
Custom code for better performance

Complexity gets quickly out of control




DYNAMO

De Candia et al.,
SOSP '07

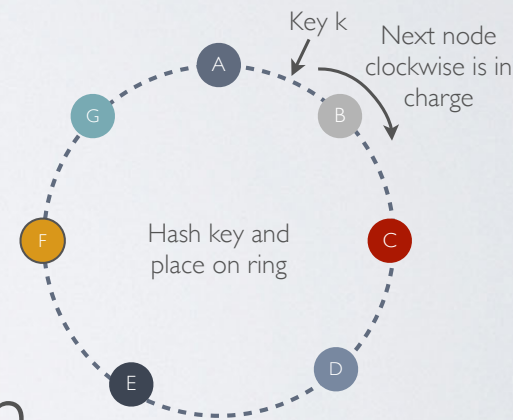


DYNAMO

- A highly available, distributed key-value store
 - `put(key, context, object); get(key)`
- Sacrifices consistency for “always on” availability 
- Supports some of Amazon key features (shopping cart)
- “Synthesis of well-known techniques to achieve scalability and availability”

CONSISTENCY?

- BASE
 - Basically Available, Soft State, Eventually Consistent
- Always writable
 - Can always write to shopping cart
 - Conflict resolution on reads
- Replication via Consistent Hashing
- Application-driven conflict resolution
 - conflict resolution on reads, not writes
 - merge conflicting shopping carts to never lose “Add Cart”



DATA VERSIONING

- A `put()` may return to its caller before the update has been applied to all replicas
- A `get()` may return many versions of the same object
- Objects have distinct version sub-histories that need to be reconciled!

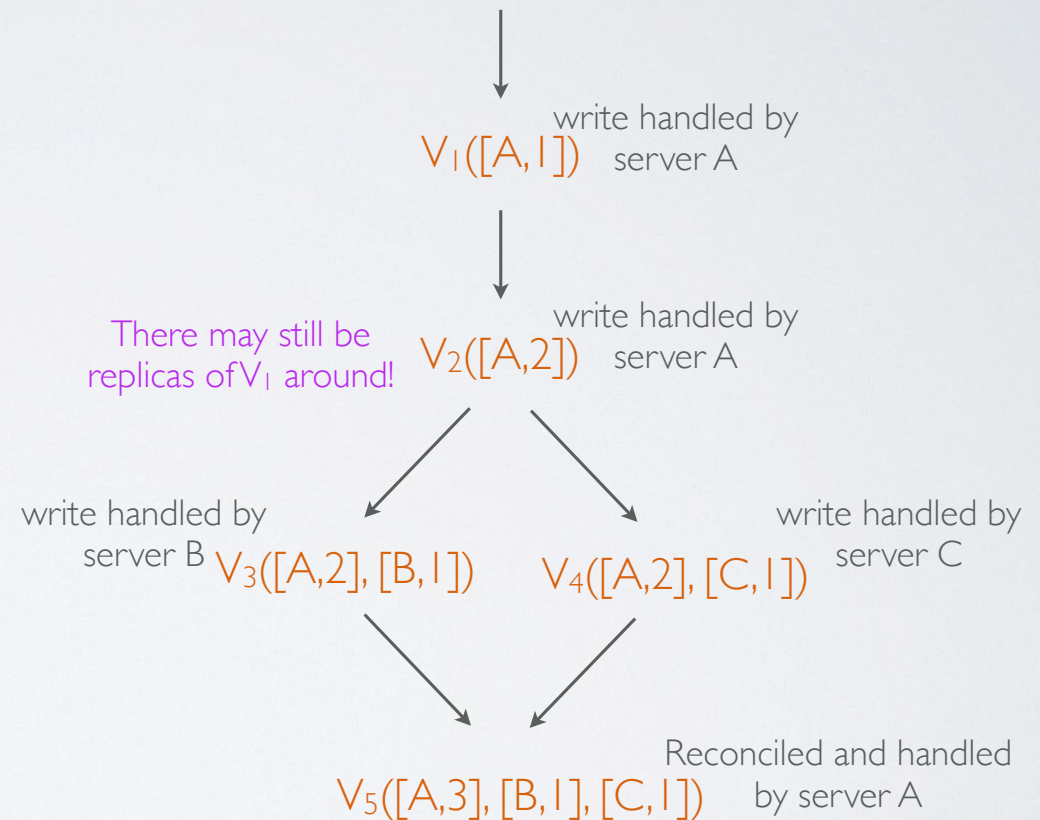
DATA VERSIONING

- A put() may return to its caller before the update has been applied to all replicas
- A get() may return many versions of the same object
- Objects have distinct version sub-histories that need to be reconciled!

Vector clocks!

VECTOR CLOCKS À LA DYNAMO

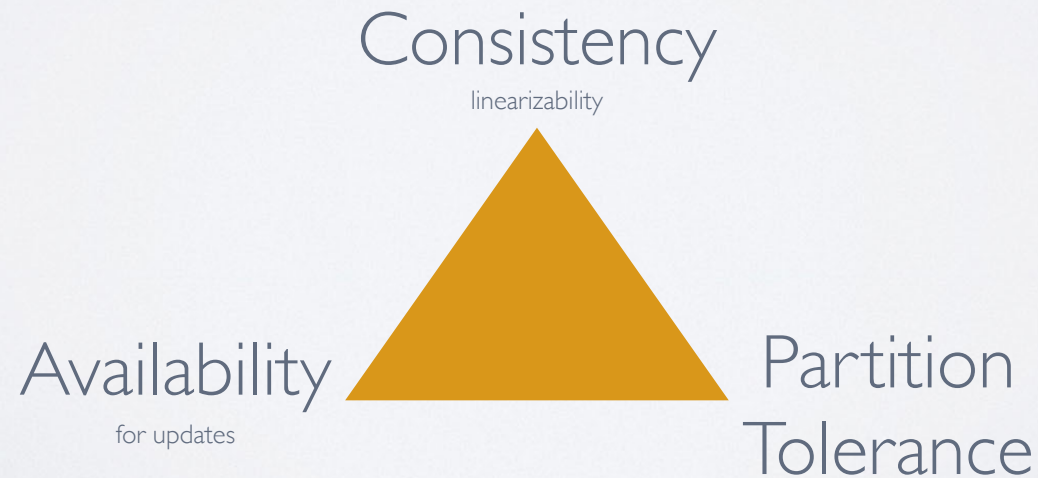
- List of (node, counter) pairs
- Each object version has one vector clock, which captures its genesis story
- Used to determine whether a version is subsumed
- If VC becomes too long, it is trimmed



QUASI-QUORUMS

- Each key is stored in a *preference list PL* of nodes
- `get()` and `put()` driven by two parameters
 - R: minimum number of replicas to read from
 - W: minimum number of replicas to write to
 - if $R+W > N$, we have a quorum system!
 -but latency dictated by slowest replica — typically R & W chosen so together they are fewer than N
- Sloppy Quorums
 - `put()` sends new version to top N reachable nodes in PL; declares success as soon as received W-1 responses
 - In case of failure, metadata specifies intended recipient; it will be notified later
 - always writeable!
 - `get()` retrieves from top N reachable nodes in PL; success as soon as it received R
 - if `get()` returns multiple versions, they are reconciled and a new corresponding version is written back

CAP THEOREM, REVISITED



THE RETURN OF THE “C”

- Last decade: a resurgence of strong consistency
 - Application code too complex and buggy without consistency support in DB
 - Better network availability makes C+P a more attractive choice, since availability loss is minimal
- CAP is not symmetric in its guarantees:
 - C+P can *guarantee* consistency...
 - ...but A+P can't *guarantee* availability (only a matter of degrees)

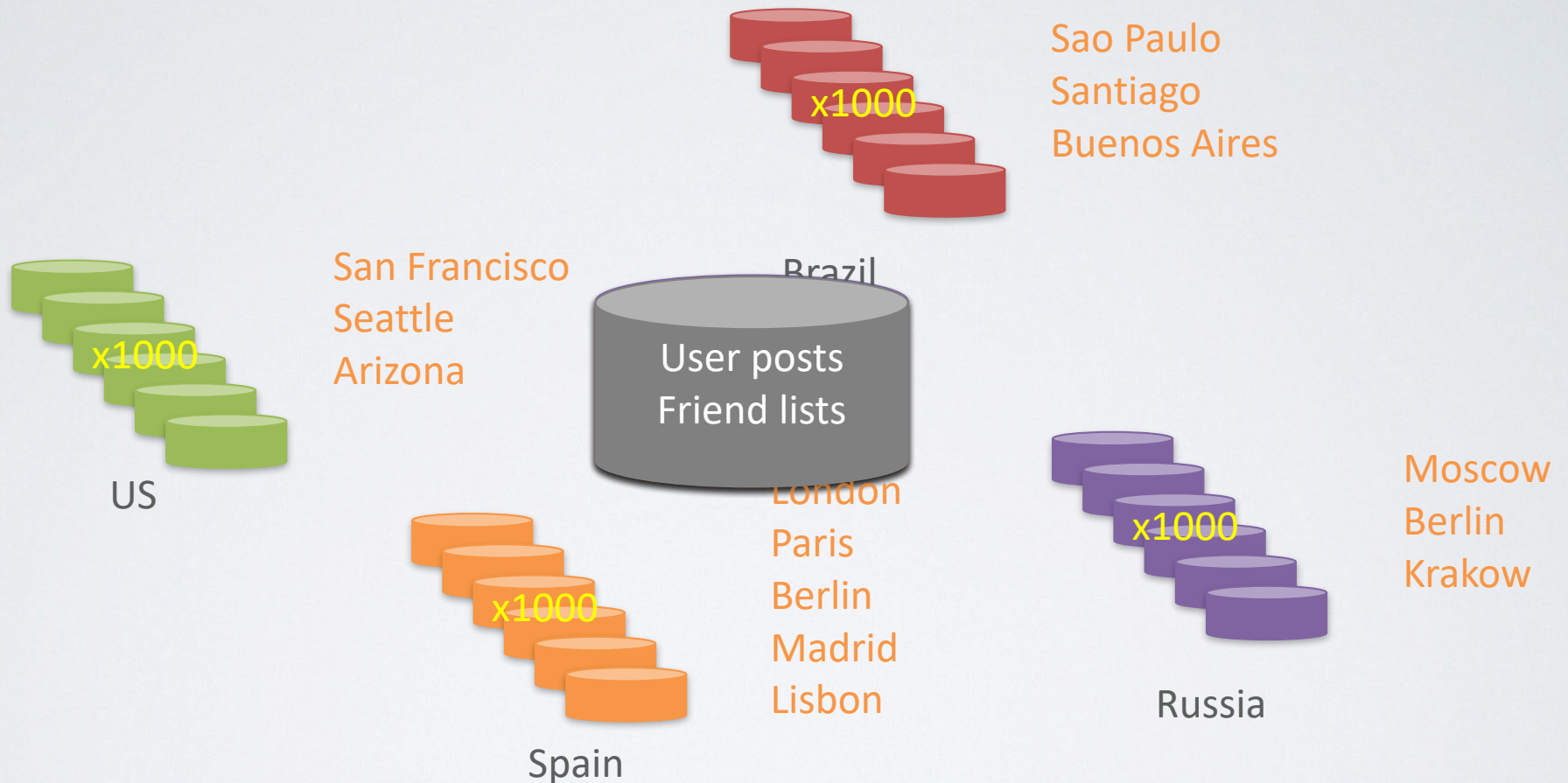
No system can guarantee 100% availability!

SPANNER

Corbett et al., OSDI 2012

- Distributed **multiversion** database
 - General purpose transactions
 - SQL / Semi-relational data model / Schemas / Tables
- Running in production
 - Storage for Google's ad data (!)

EXAMPLE: SOCIAL NETWORK



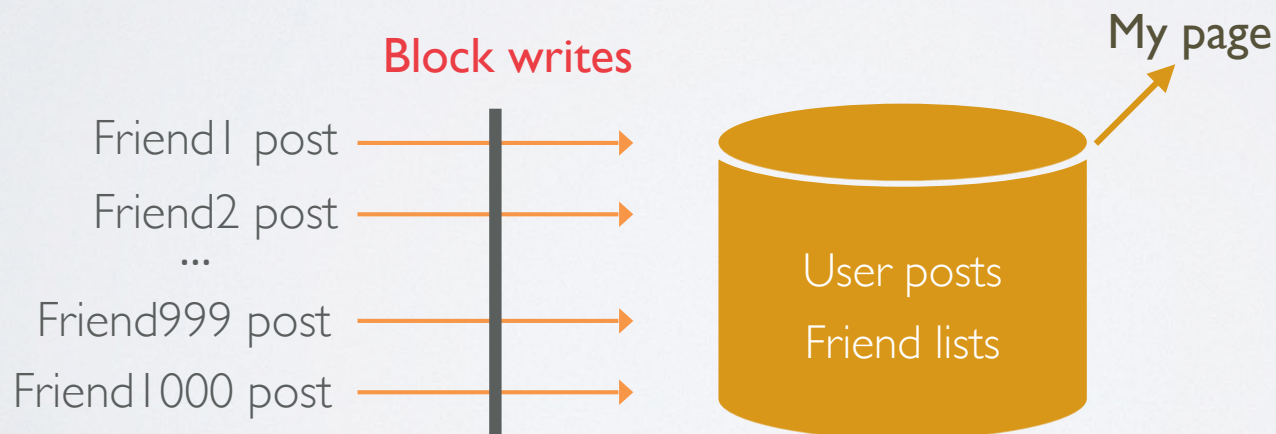
Figures adapted from [Wilson Hsieh and coauthors, OSDI 2012]

OVERVIEW

- Spanner offers:
 - Lock-free distributed read-only transactions
 - **External consistency** [strict serializability] for distributed transactions
 - Strictly serializable order consistent with real-time: transactions are “linearized” at their commit time
 - Integration of concurrency control, replication, and 2PC
- Enabling technology: **TrueTime**
 - exposes uncertainty about time by representing global time in terms of intervals

READ-ONLY TRANSACTIONS

- Suppose I want to generate a web page with my friends' recent posts
- From a single machine...



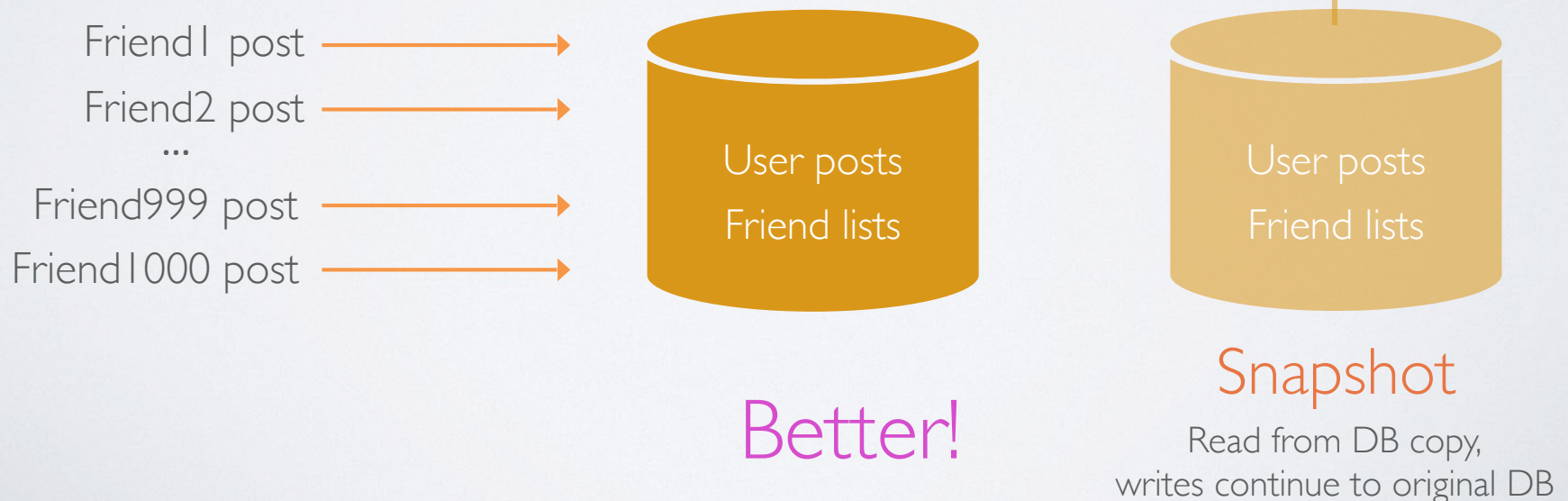
Read lock

Block all writes until
read has finished

We'd rather
not do that...

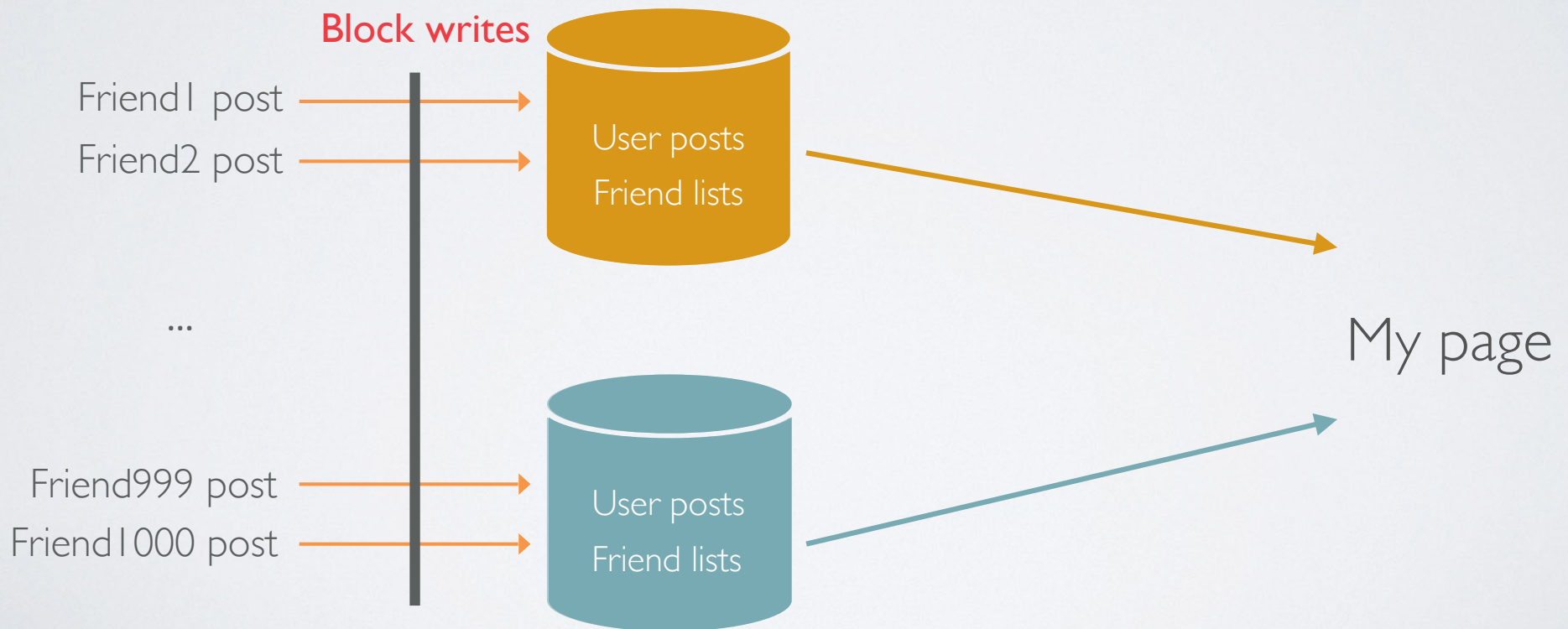
READ-ONLY TRANSACTIONS

- Suppose I want to generate a web page with my friends' recent posts
- From a single machine...



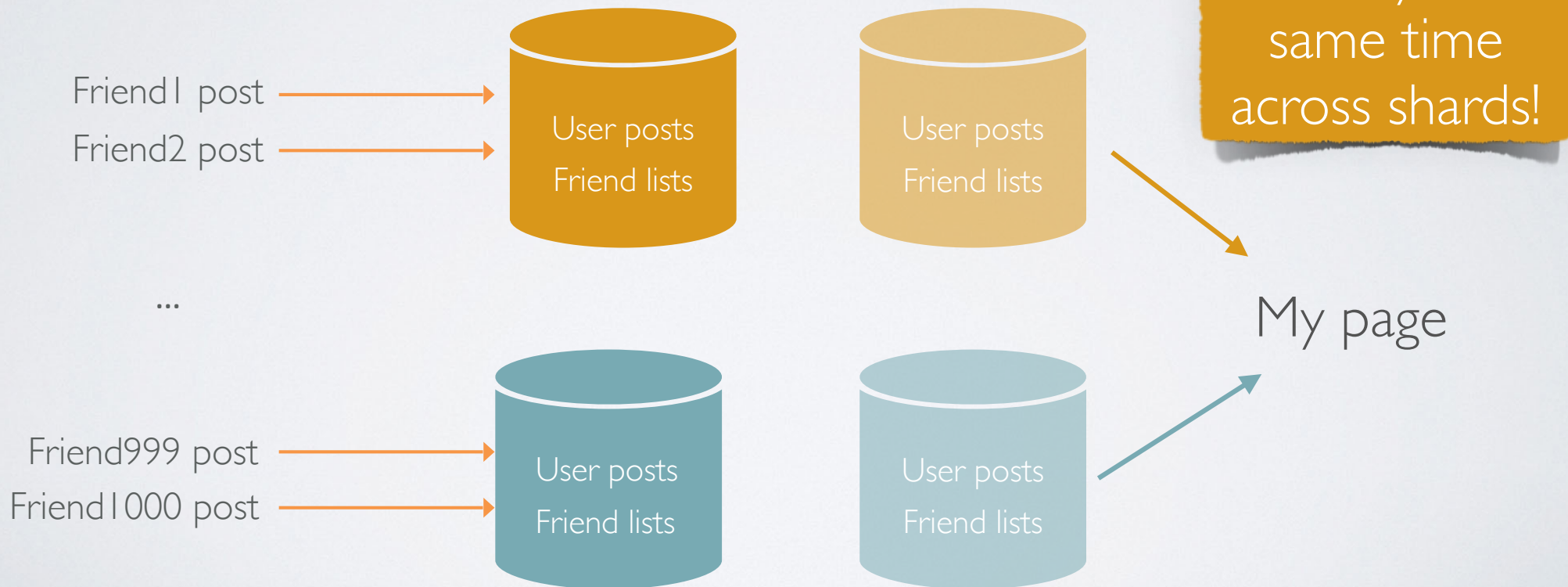
READ TRANSACTIONS

- From multiple machines...



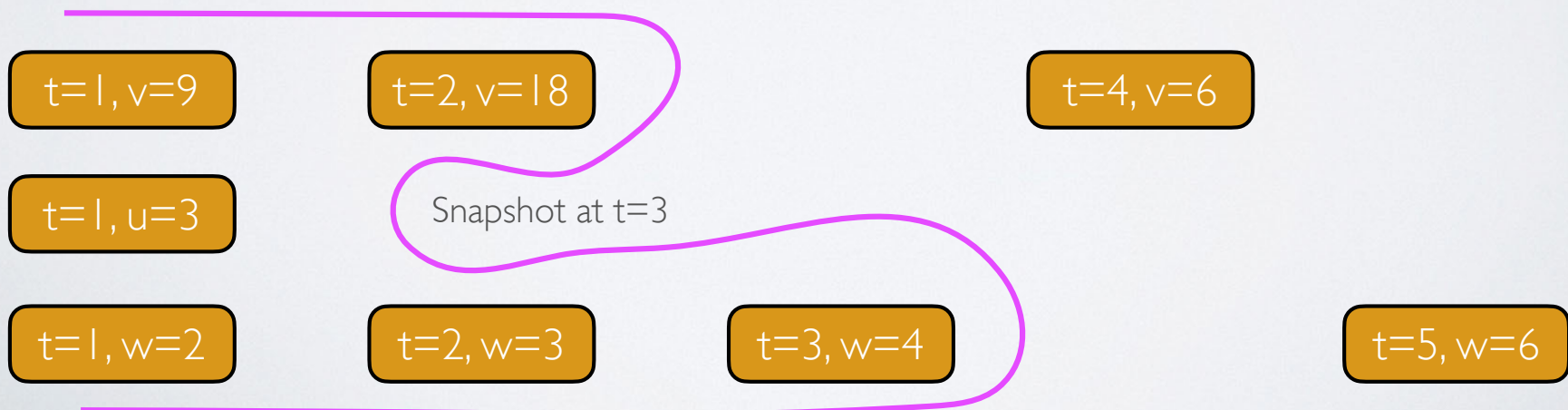
READ TRANSACTIONS

- From multiple machines...



IMPLEMENTING SNAPSHOT READS

- **Multi-version** concurrency control
 - At commit time, create new versions of modified objects as tuples (timestamp, value)
 - Retain old (timestamp, value) tuples
 - Snapshot: read latest tuples with timestamp $<$ now



HOW HARD IS TO SYNCHRONIZE CLOCKS?

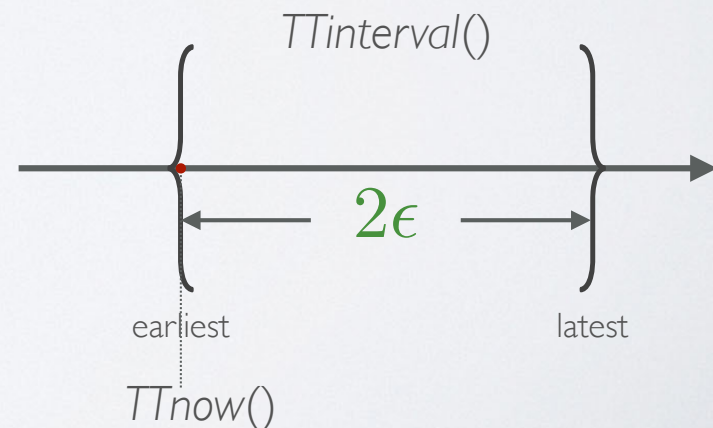
- Time synchronization error proportional to RTT!
- For geo-replicated data centers, that is in the hundreds of milliseconds!

TRUE TIME: TIME AS AN INTERVAL

- “Global wall-clock time” with bounded uncertainty

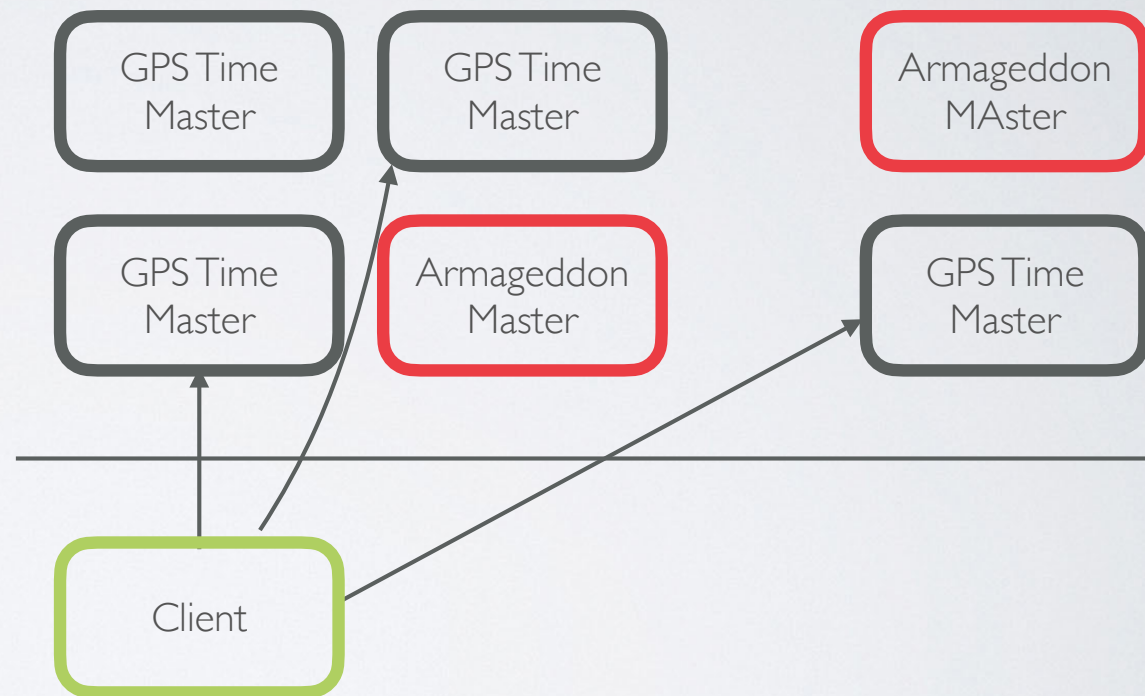
Method	Returns
<i>TT.now()</i>	<i>TTinterval</i> : [<i>earliest</i> , <i>latest</i>]
<i>TT.after(t)</i>	true if <i>t</i> has definitely passed
<i>TT.before(t)</i>	true if <i>t</i> has definitely not arrived

- The *TTinterval* returned by *TT.now()* is guaranteed to include the absolute time at which *TT.now()* was invoked
- Between synchronizations, ϵ grows from 1 to 7 ms (200 μ s/s drift, 30s poll, + 1 ms of communication delay from Time Master)



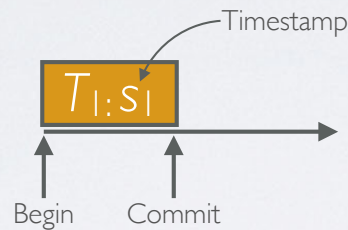
TIME-SYNCH IN SPANNER

- Each datacenter has a set of *time leader* (some with GPS antennas, others [the *Armageddon leaders*] with atomic clocks) and a *time follower daemon* per machine
- Each daemon uses a variant of *Marzullo's algorithm* (used also in NTP) to detect and reject liars, and synchronize to non liars



TRUE TIME AND STRICT SERIALIZABILITY

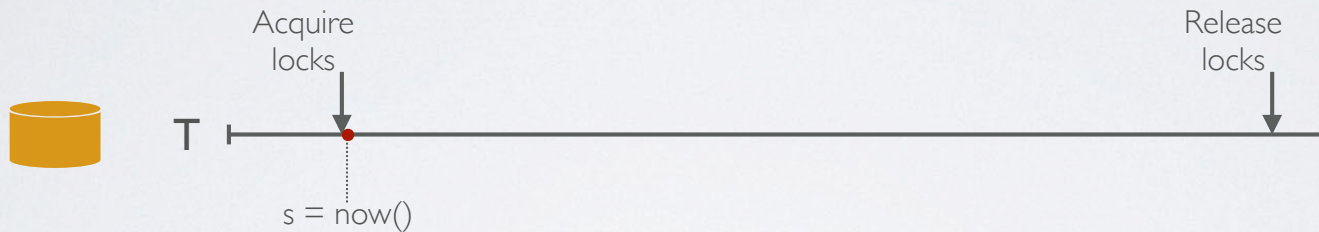
- Assign transactions a globally meaningful timestamp



- Commit transactions in timestamp order
- Ensure that if T_1 commits before T_2 begins, the $s_1 < s_2$
- Then reading a consistent state is easy:
 - to obtain a consistent snapshot at t , read state produced by transactions whose timestamp is at most t

HOW SHOULD WE PICK THE TIMESTAMP?

- Spanner uses strict two-phase locking for write transactions...
 - ...assign timestamp while locks are held!



- But how? Let

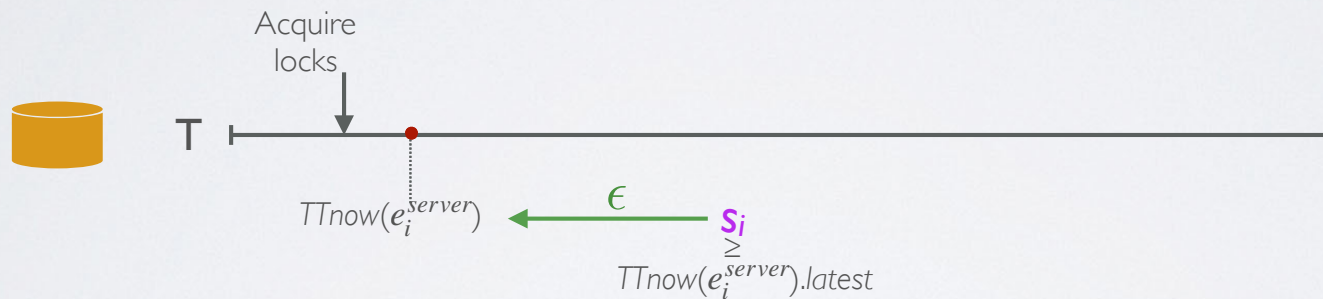
e_i^{begin} = T_i 's begin event e_i^{commit} = T_i 's commit event $t_{abs}(e)$ = e 's absolute time

- Strict serializability requires:

$$t_{abs}(e_1^{commit}) < t_{abs}(e_2^{begin}) \Rightarrow s_1 < s_2$$

BUT TIME IS AN INTERVAL...

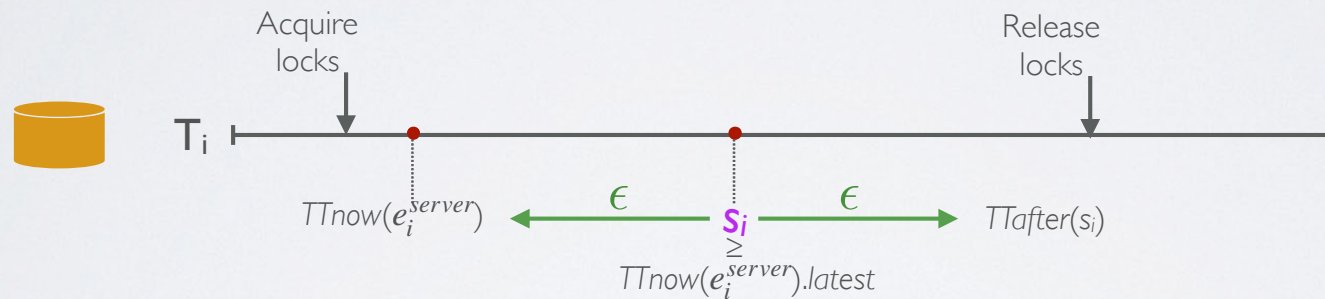
- Base T_i 's timestamp on result of $TTnow(e_i^{server})$, where e_i^{server} is the event corresponding to T_i 's coordinator receiving from the client the indication that it would like to commit T_i



- Choosing s_i at least as large $TTnow().latest$ ensures that s is largest than the result of any $TTnow()$ invoked **before** T_i begins

BUT TIME IS AN INTERVAL...

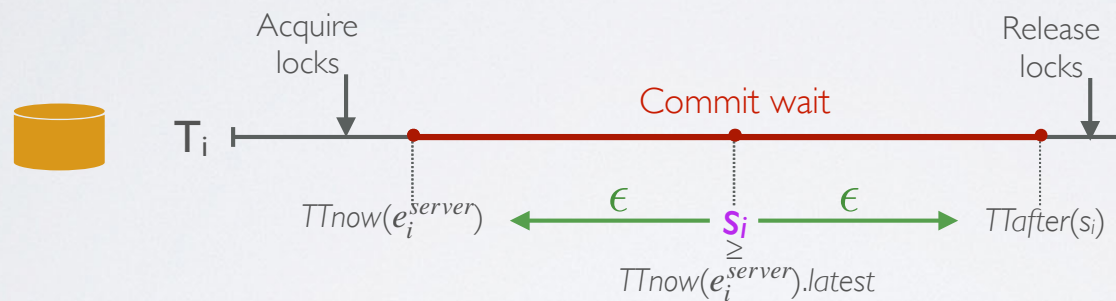
- Base T_i 's timestamp on result of $TTnow(e_i^{server})$, where e_i^{server} is the event corresponding to T_i 's coordinator receiving from the client the indication that it would like to commit T_i



- Choosing s_i at least as large $TTnow().latest$ ensures that s is largest than the result of any $TTnow()$ invoked before T_i begins

BUT TIME IS AN INTERVAL...

- Base T_i 's timestamp on result of $TTnow(e_i^{server})$, where e_i^{server} is the event corresponding to T_i 's coordinator receiving from the client the indication that it would like to commit T_i



- Choosing s_i at least as large $TTnow().latest$ ensures that s is largest than the result of any $TTnow()$ invoked before T_i begins
- Releasing locks and committing after $TTafter(s_i)$ ensures clients cannot see any data committed by T_i until s_i has definitely passed.

ENFORCING STRICT SERIALIZABILITY

$$t_{abs}(e_1^{commit}) < t_{abs}(e_2^{begin}) \Rightarrow s_1 < s_2$$

$$s_1 < t_{abs}(e_1^{commit}) \quad (\text{commit wait})$$

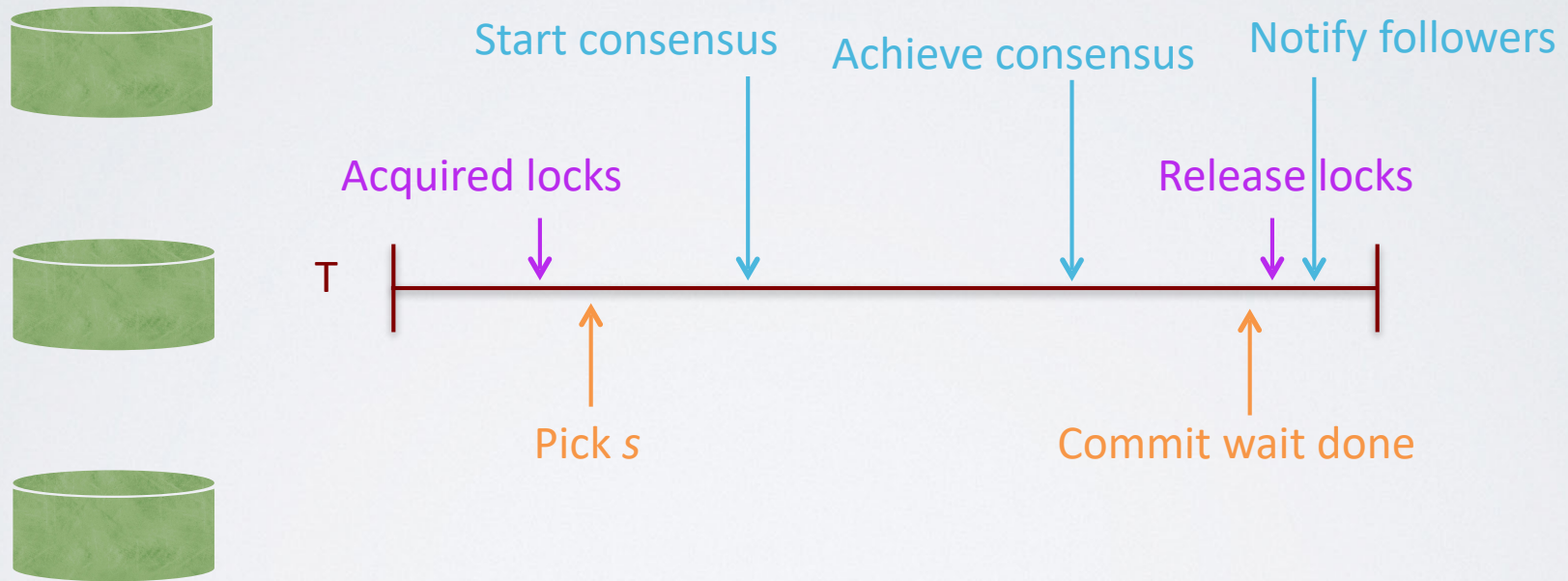
$$t_{abs}(e_1^{commit}) < t_{abs}(e_2^{begin}) \quad (\text{assumption})$$

$$t_{abs}(e_2^{begin}) \leq t_{abs}(e_2^{server}) \quad (\text{causality})$$

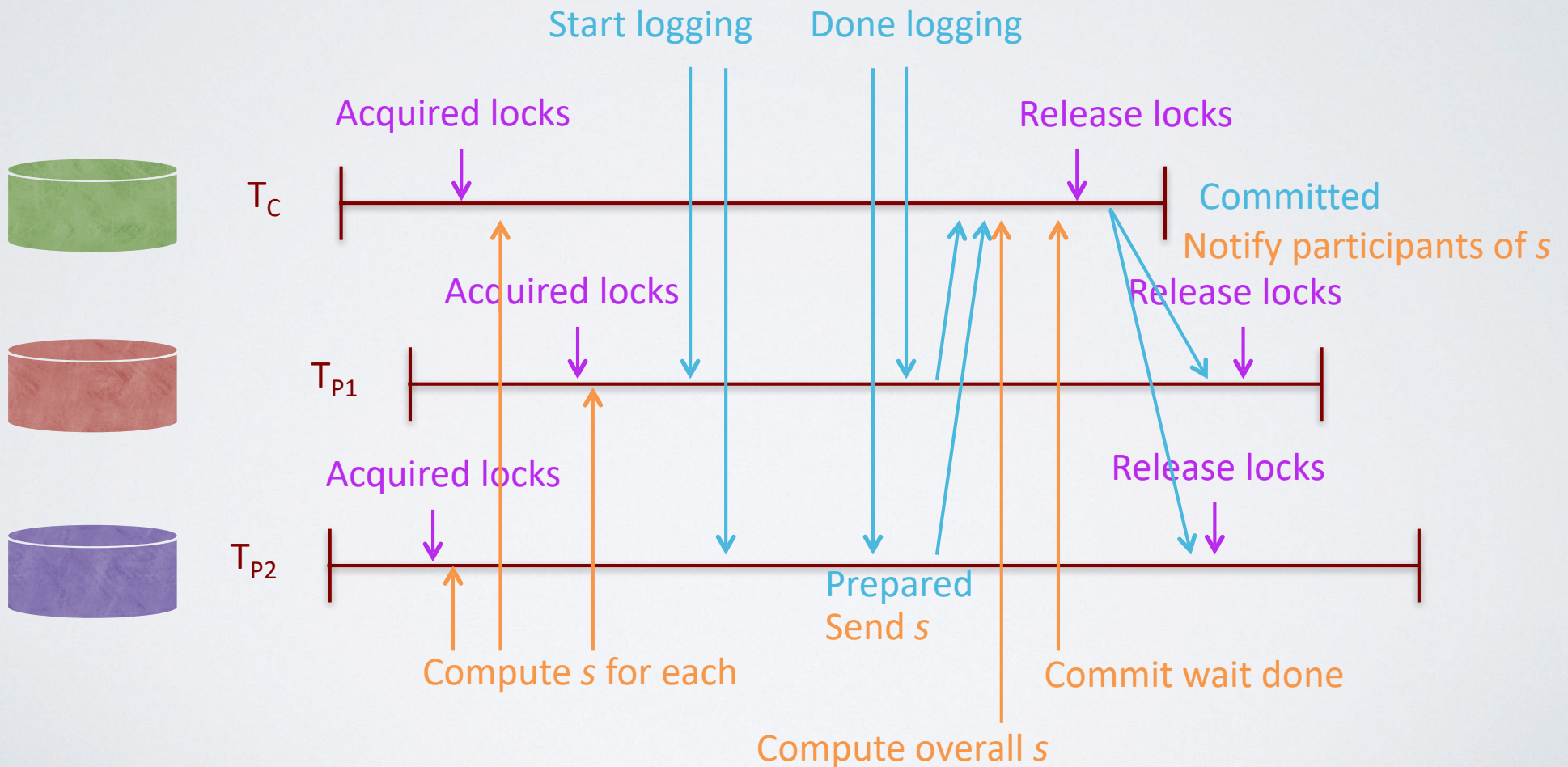
$$t_{abs}(e_2^{server}) \leq s_2 \quad (\text{start})$$

$$s_1 \leq s_2 \quad (\text{transitivity})$$

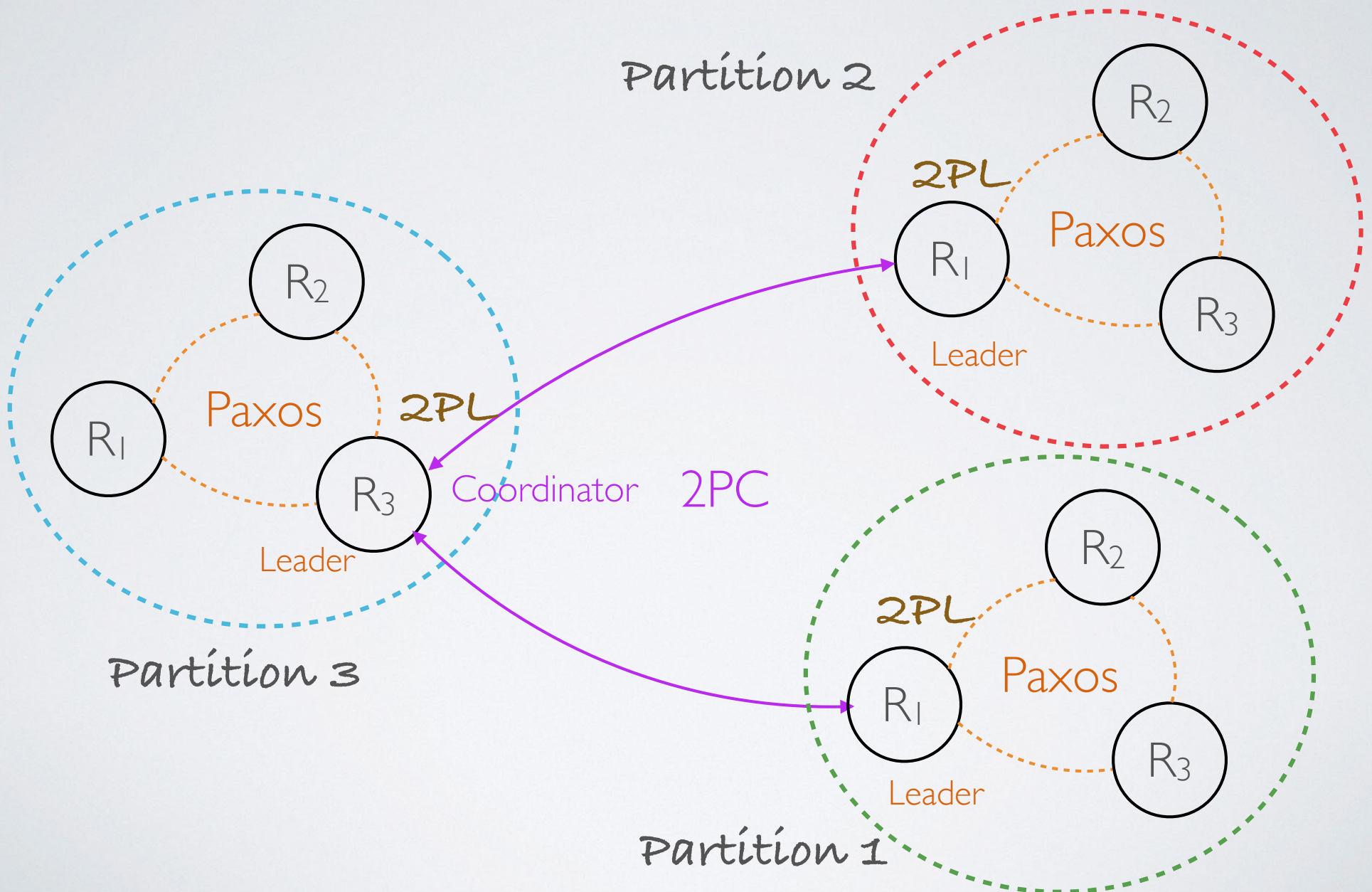
COMMIT WAIT AND REPLICATION



COMMIT WAIT AND 2-PHASE COMMIT



THE BIG PICTURE: RW



THE BIG PICTURE: RO

- Read from a snapshot read at time t
 - for scalability, go to any replica, not just the leader
- Use TrueTime to ensure that, no matter which replica we use, we are sure to see every version with timestamp at most t_{safe}
- Block reading until $t_{safe} \geq t$