Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

Step 2 p_i is waiting for VOTE-REQ from coordinator Exactly as in 2PC	Step 3 Coordinator is waiting for vote from participants
Step 4 p_i waits for PRECOMMIT	Step 5 Coordinator waits for ACKs
Step 6 p_i waits for COMMIT	

Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

Step 2 p_i is waiting for VOTE-REQ from coordinator Exactly as in 2PC	Step 3 Coordinator is waiting for vote from participants Exactly as in 2PC
Step 4 p_i waits for PRECOMMIT	Step 5 Coordinator waits for ACKs
Step 6 p_i waits for COMMIT	

Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

Step 2 p_i is waiting for VOTE-REQ from coordinator Exactly as in 2PC	Step 3 Coordinator is waiting for vote from participants Exactly as in 2PC
Step 4 p_i waits for PRECOMMIT	Step 5 Coordinator waits for ACKs Coordinator sends COMMIT
Step 6 p_i waits for COMMIT	

Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

Step 2 p_i is waiting for VOTE-REQ from coordinator Exactly as in 2PC	Step 3 Coordinator is waiting for vote from participants Exactly as in 2PC
Step 4 p_i waits for PRECOMMIT Run some Termination protocol	Step 5 Coordinator waits for ACKs Coordinator sends COMMIT
Step 6 p_i waits for COMMIT	

Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

Step 2 p_i is waiting for VOTE-REQ from coordinator Exactly as in 2PC	Step 3 Coordinator is waiting for vote from participants Exactly as in 2PC
Step 4 p_i waits for PRECOMMIT Run some Termination protocol	Step 5 Coordinator waits for ACKs Coordinator sends COMMIT
Step 6 p_i waits for COMMIT	Participant knows what is going to receive...

Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

Step 2 p_i is waiting for VOTE-REQ from coordinator Exactly as in 2PC	Step 3 Coordinator is waiting for vote from participants Exactly as in 2PC
Step 4 p_i waits for PRECOMMIT Run some Termination protocol	Step 5 Coordinator waits for ACKs Coordinator sends COMMIT
Step 6 p_i waits for COMMIT Run some Termination protocol	Participant knows what is going to receive... but NB property can be violated!

Termination protocol: Process states

At any time while running 3 PC, each participant can be in exactly one of these 4 states:

Aborted	Not voted, voted NO, received ABORT
Uncertain	Voted YES, not received PRECOMMIT
Committable	Received PRECOMMIT, not COMMIT
Committed	Received COMMIT

Not all states are compatible

	Aborted	Uncertain	Committable	Committed
Aborted	Y	Y	N	N
Uncertain	Y	Y	Y	N
Committable	N	Y	Y	Y
Committed	N	N	Y	Y

Termination protocol

- When p_i times out, it starts an election protocol to elect a new coordinator
- The new coordinator sends STATE-REQ to all processes that participated in the election
- The new coordinator collects the states and follows a termination rule

TR1. if some process decided ABORT, then
  decide ABORT
  send ABORT to all
  halt

TR2. if some process decided COMMIT, then
  decide COMMIT
  send COMMIT to all
  halt

TR3. if all processes that reported state are uncertain, then
  decide ABORT
  send ABORT to all
  halt

TR4. if some process is committable, but none committed, then
  send PRECOMMIT to uncertain processes
  wait for ACKs
  send COMMIT to all
  halt

Termination protocol and failures

Processes can fail while executing the termination protocol...

- if c times out on p, it can just ignore p
- if c fails, a new coordinator is elected and the protocol is restarted (election protocol to follow)
- total failures will need special care...

Recovering p

- if p fails before sending YES, decide ABORT
- if p fails after having decided, follow decision
- if p fails after voting YES but before receiving decision value
  - p asks other processes for help
  - 3PC is non blocking: p will receive a response with the decision
- if p has received PRECOMMIT
  - still needs to ask other processes (cannot just COMMIT)

Recovering p

- if p fails before sending YES, decide ABORT
- if p fails after having decided, follow decision
- if p fails after voting YES but before receiving decision value
  - p asks other processes for help
  - 3PC is non blocking: p will receive a response with the decision
- if p has received PRECOMMIT
  - still needs to ask other processes (cannot just COMMIT)

No need to log PRECOMMIT!?!

The election protocol

- Processes agree on linear ordering (e.g. by pid)
- Each p maintains set UP_p of all processes that p believes to be operational
- When p detects failure of c, it removes c from UP_p and chooses smallest q in UP_p to be new coordinator
- If $q = p$, then p is new coordinator
- Otherwise, p sends UR-ELECTED to q

A few observations

- What if p', which has not detected the failure of c, receives a STATE-REQ from q ?

A few observations

- What if p', which has not detected the failure of c, receives a STATE-REQ from q ?
  - it concludes that c must be faulty
  - it removes from $UP_{p'}$ every $q' < q$

A few observations

- What if p', which has not detected the failure of c, receives a STATE-REQ from q ?
  - it concludes that c must be faulty
  - it removes from $UP_{p'}$ every $q' < q$
- What if p' receives a STATE-REQ from c after it has changed the coordinator to q ?

A few observations

- What if p', which has not detected the failure of c, receives a STATE-REQ from q ?
  - it concludes that c must be faulty
  - it removes from $UP_{p'}$ every $q' < q$
- What if p' receives a STATE-REQ from c after it has changed the coordinator to q ?
  - p' ignores the request

Total failure

- Suppose p is the first process to recover, and that p is uncertain
- Can p decide ABORT?

Total failure

- Suppose p is the first process to recover, and that p is uncertain
- Can p decide ABORT?

Some processes could have decided COMMIT after p crashed!

Total failure

- Suppose p is the first process to recover, and that p is uncertain
- Can p decide ABORT?

  Some processes could have decided COMMIT after p crashed!

- p is blocked until some q recovers s.t. either
  - q can recover independently
  - q is the last process to fail–then q can simply invoke the termination protocol

Determining the last process to fail

- Suppose a set R of processes has recovered
- Does R contain the last process to fail?

Determining the last process to fail

- Suppose a set R of processes has recovered
- Does R contain the last process to fail?
  - the last process to fail is in the UP set of every process
  - so the last process to fail must be in

$$\bigcap_{p \in R} UP_p$$

Determining the last process to fail

- Suppose a set R of processes has recovered
- Does R contain the last process to fail?
  - the last process to fail is in the UP set of every process
  - so the last process to fail must be in

$$\bigcap_{p \in R} UP_p$$

R contains the last process to fail if

$$\bigcap_{p \in R} UP_p \subseteq R$$

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

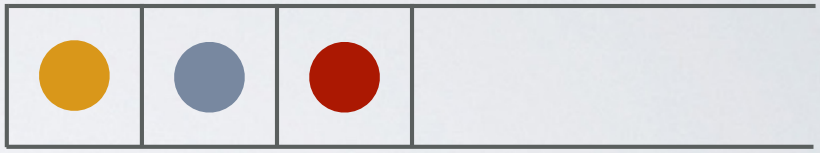
Thanks to Maurice Herlihy

- **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

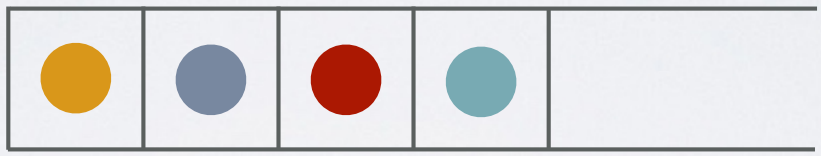


Pierre Franc Lamy (1911)
Young girl on a balcony



Carlo Carrà (1912)
Concurrency, Woman on a balcony

METHODS TAKE TIME

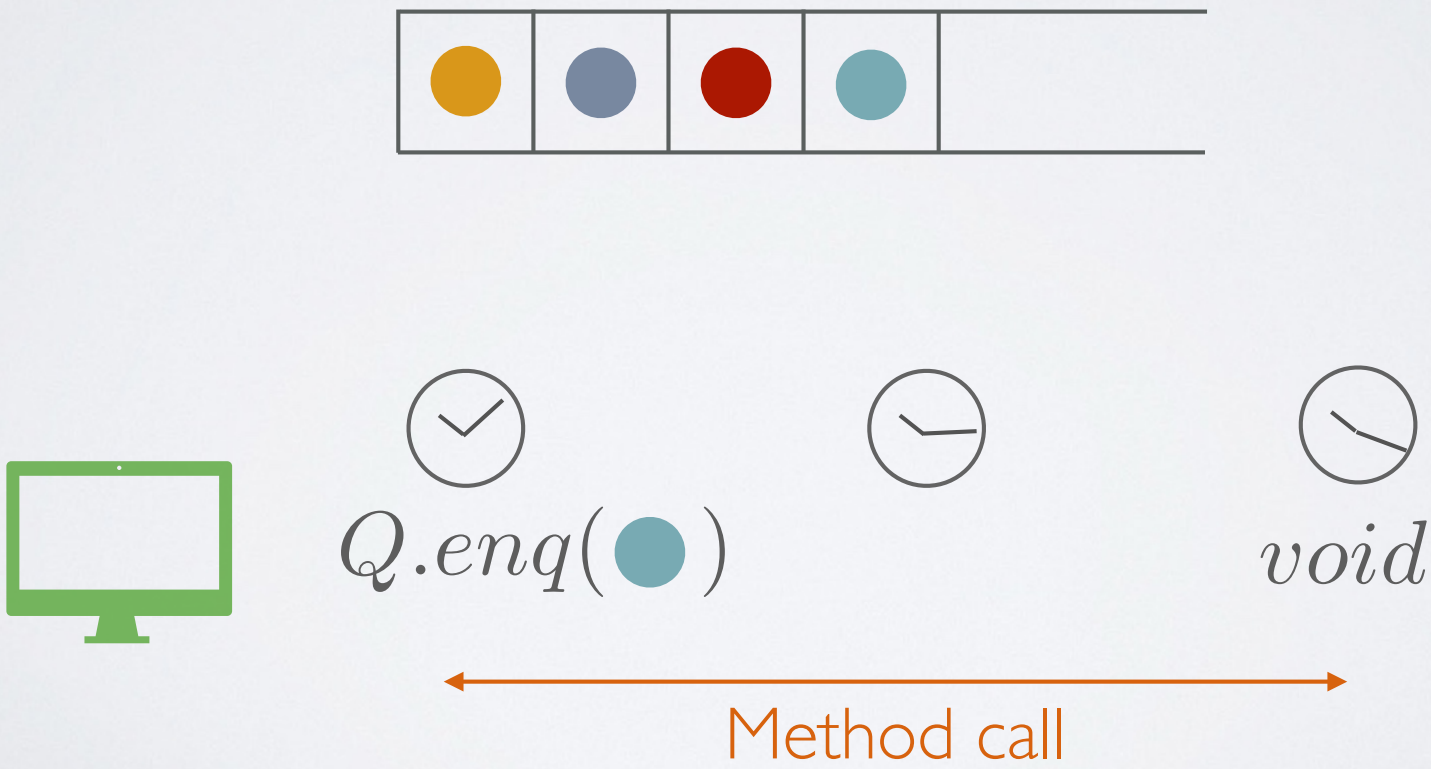








$Q.enq($  $)$


$Q.enq(\bullet)$
$void$
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




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

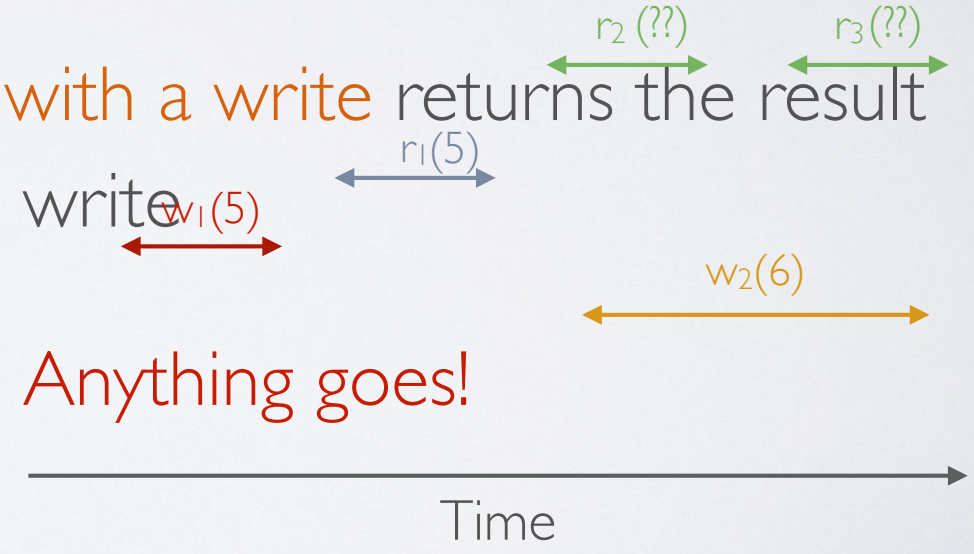
- Sequential specification
  - A read returns the result of the latest completed write
- What if reads and writes can be concurrent?
  - A read not concurrent with a write returns the result of the latest completed write
- And if they are concurrent?

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

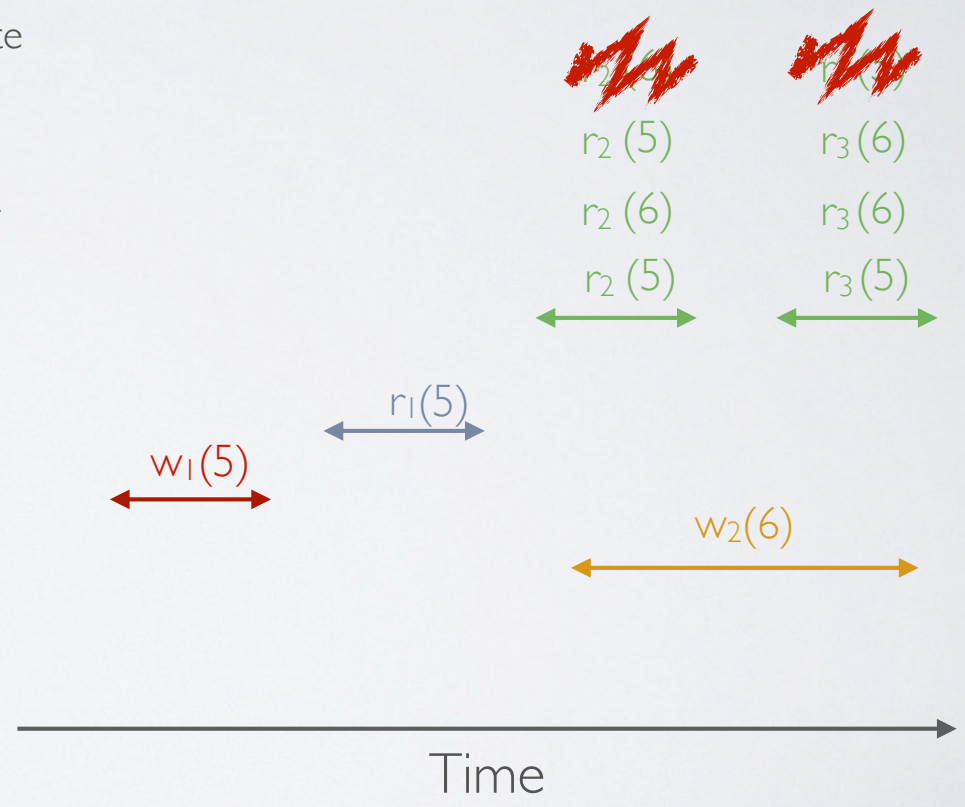
- Sequential specification
  - A read returns the result of the latest completed write
- What if reads and writes can be concurrent?
  - A read not concurrent with a write returns the result of the latest completed write
- And if they are concurrent? Anything goes!



REGULAR REGISTERS

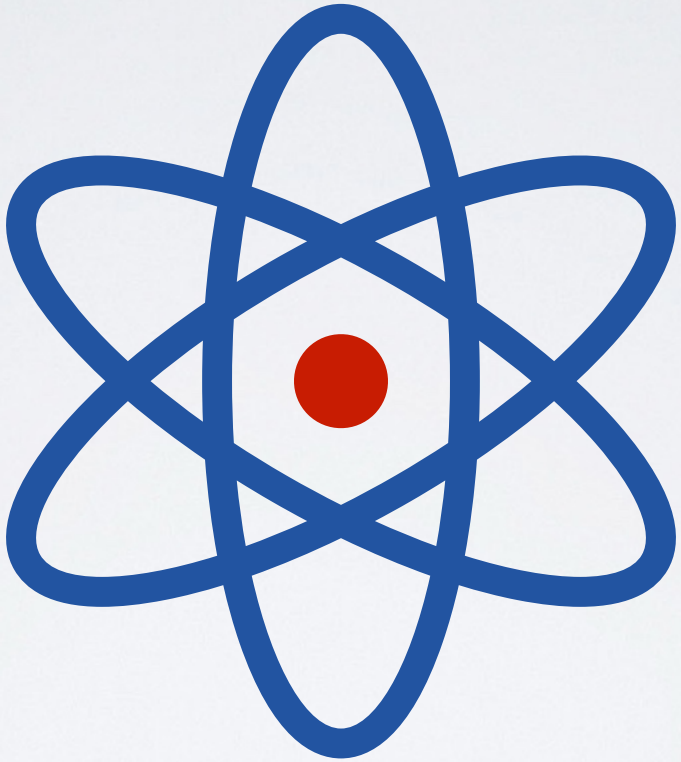
- Sequential specification
  - A read returns the result of the latest completed write
- What if reads and writes can be concurrent?
  - A read not concurrent with a write returns the result of the latest completed write
- And if they are concurrent?

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



CAN WE DO BETTER?

ATOMIC REGISTERS

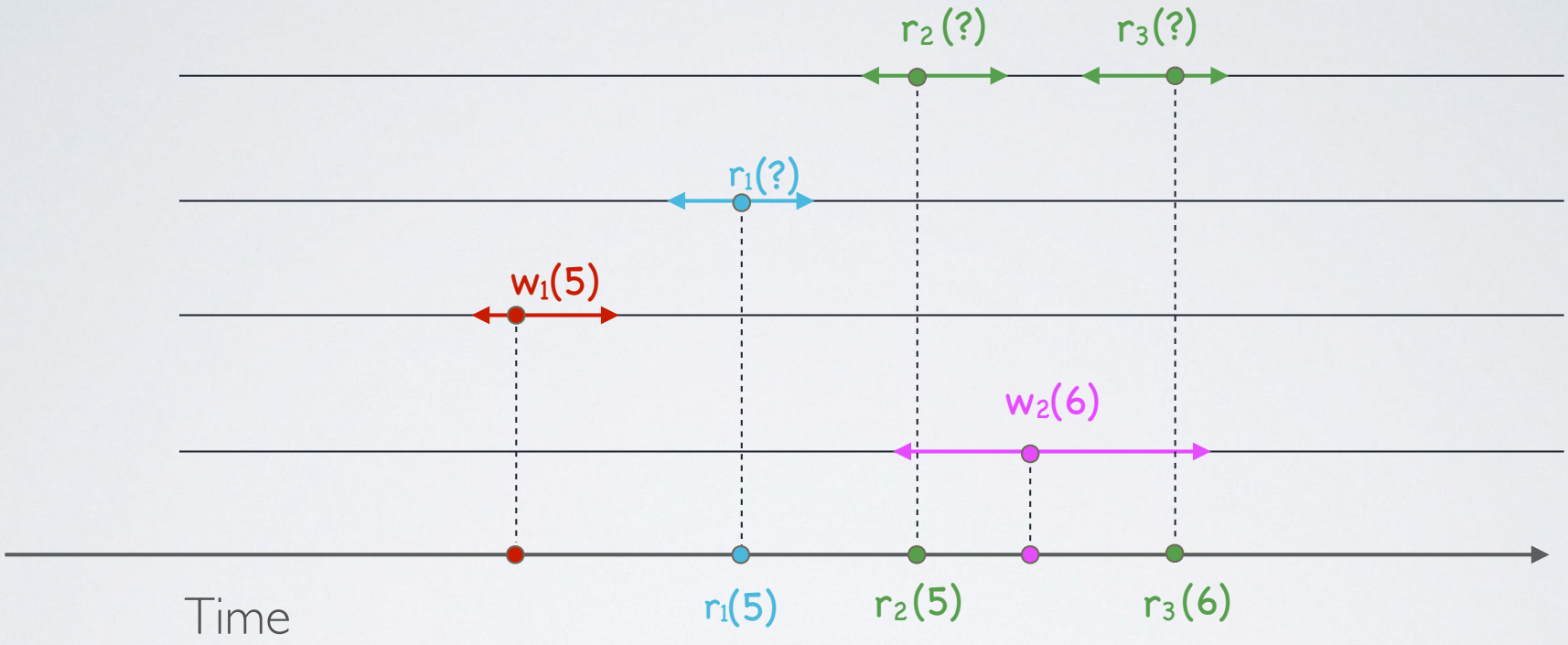


LINEARIZABILITY

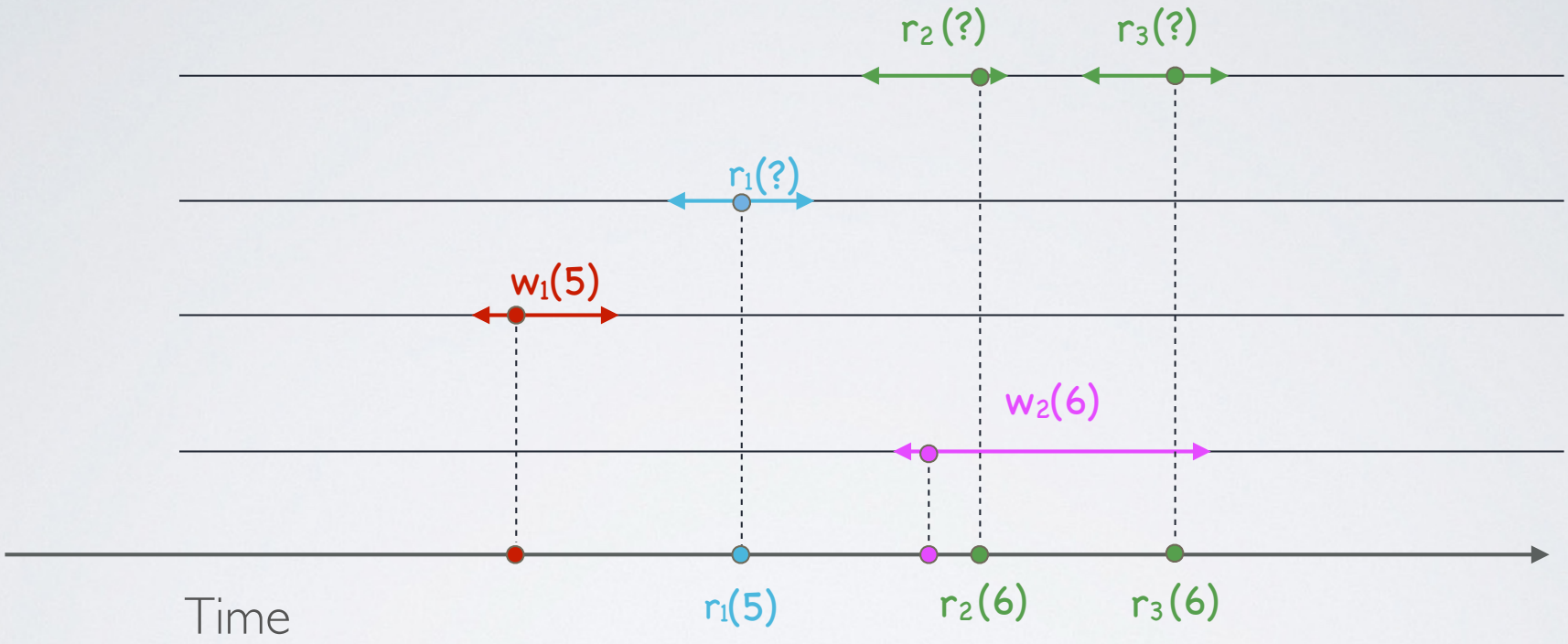
Herlihy & Wing '87

- 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

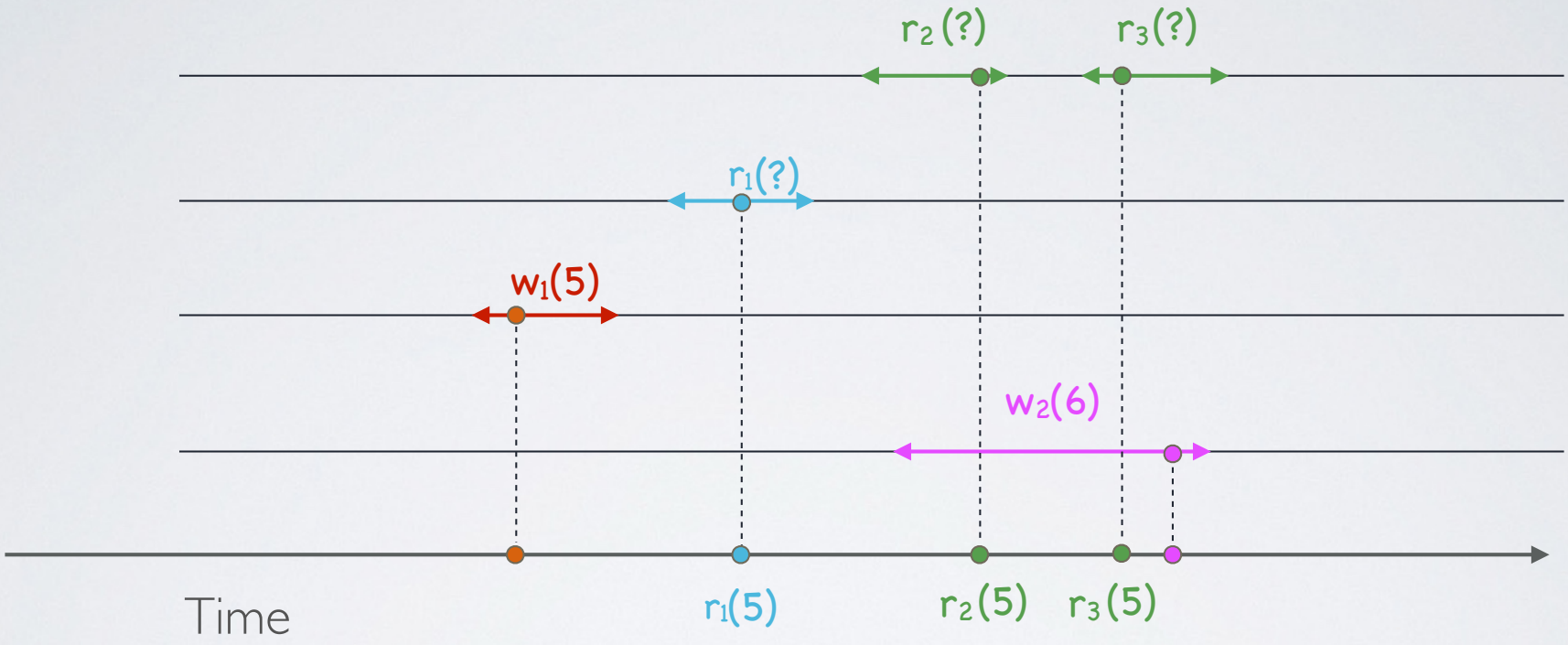
EXAMPLE: REGISTERS



EXAMPLE: REGISTERS

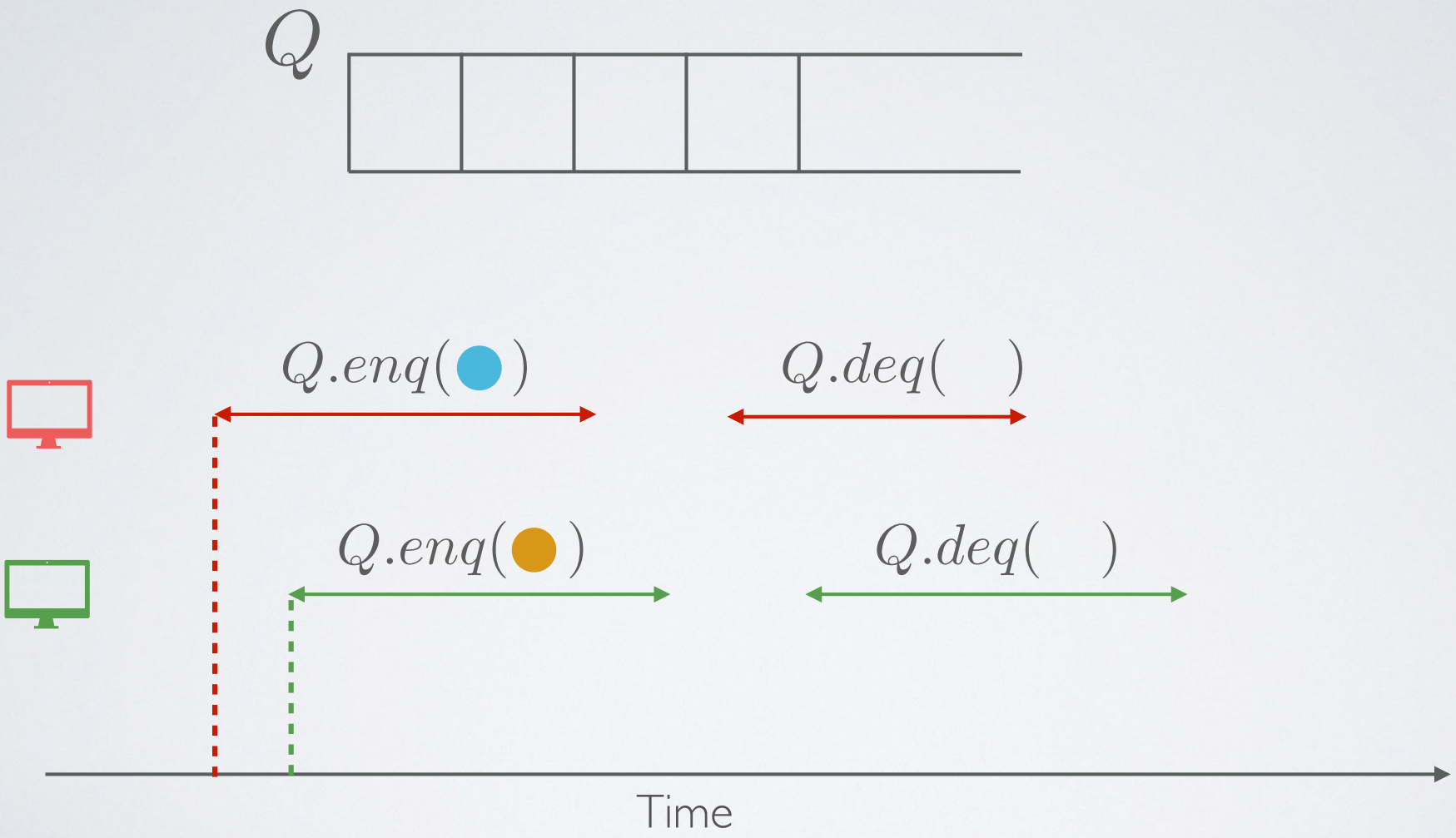


EXAMPLE: REGISTERS

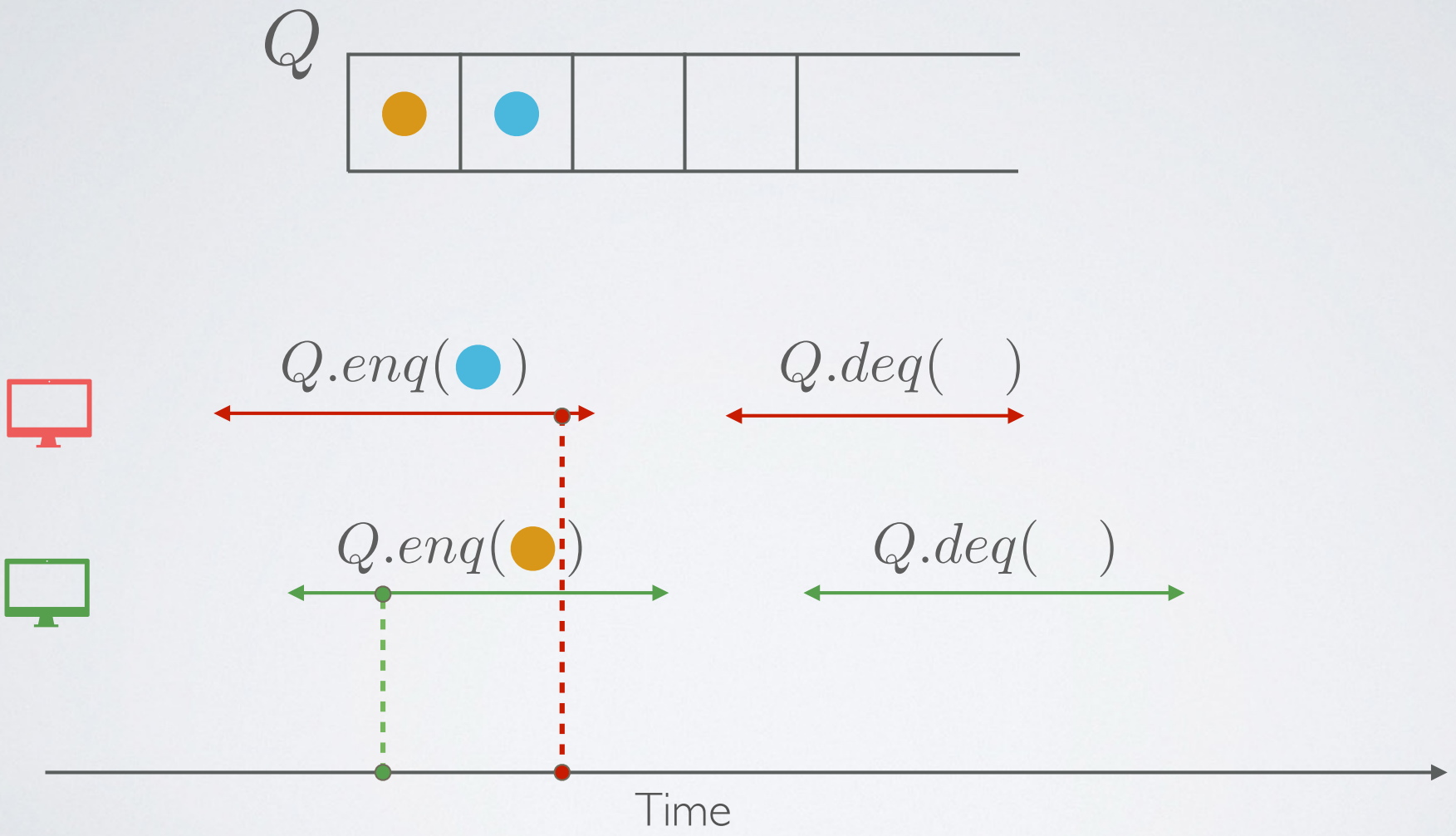


Linearizable registers are called **atomic**

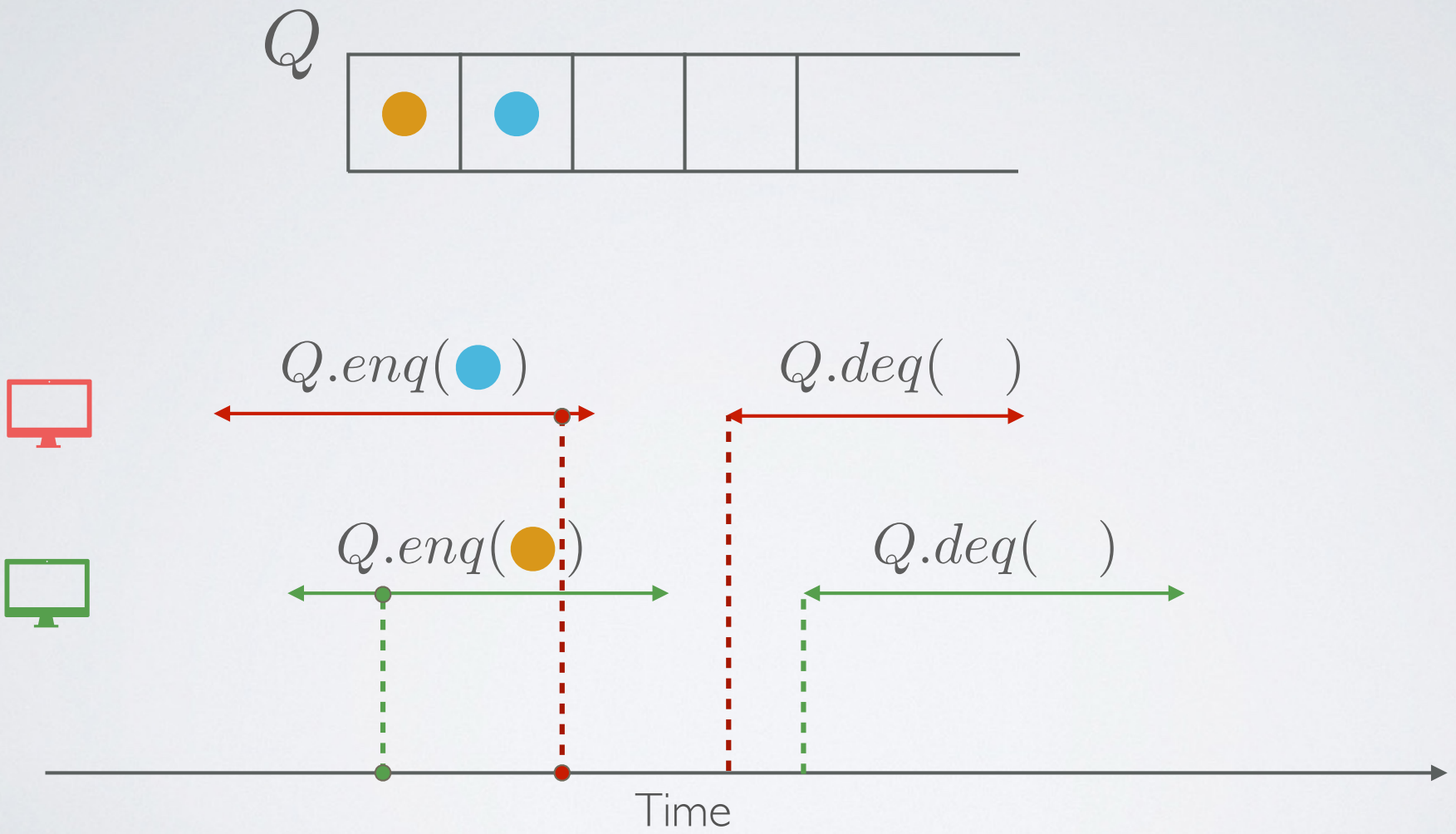
LINEARIZABLE QUEUE



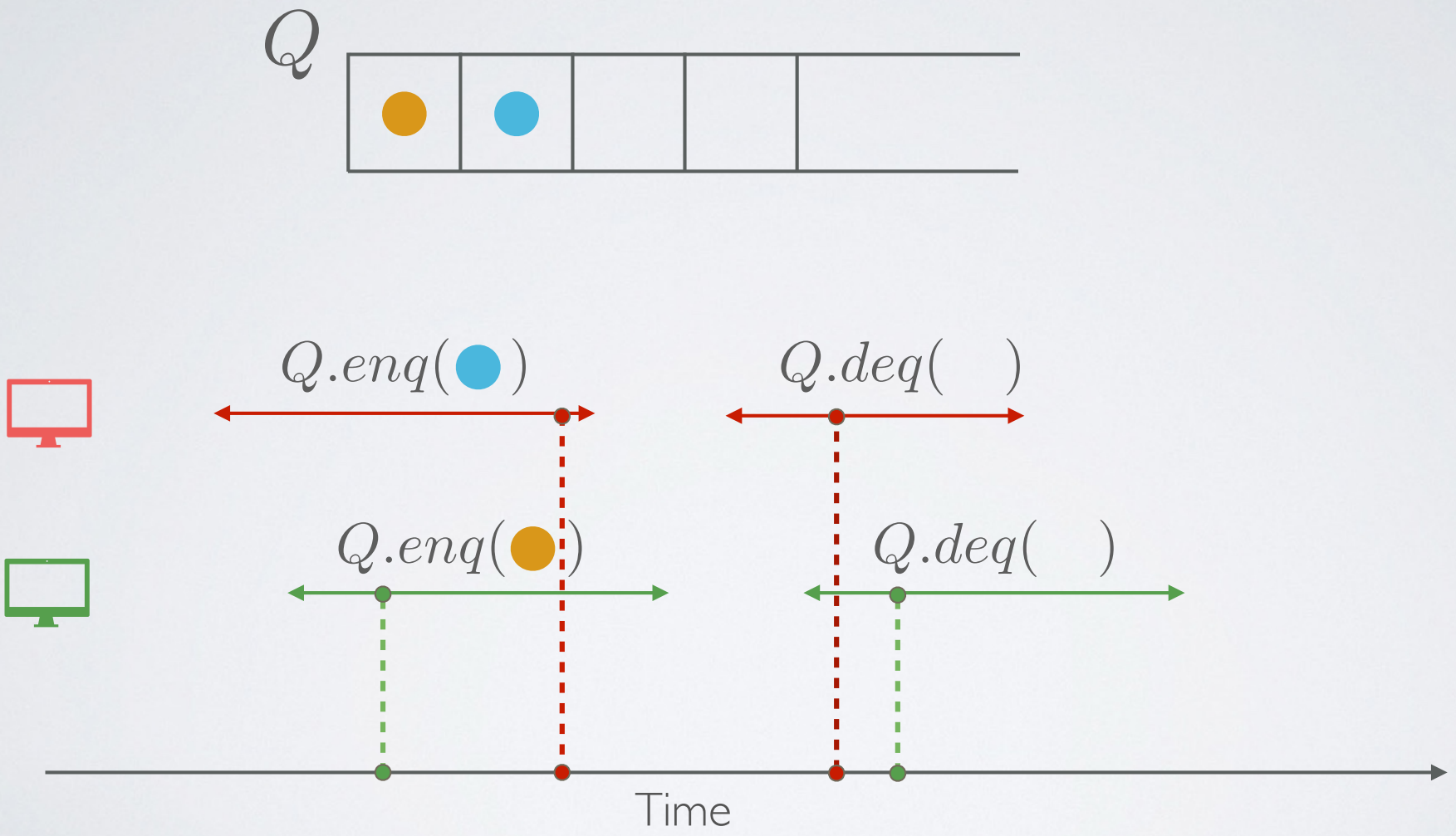
LINEARIZABLE QUEUE



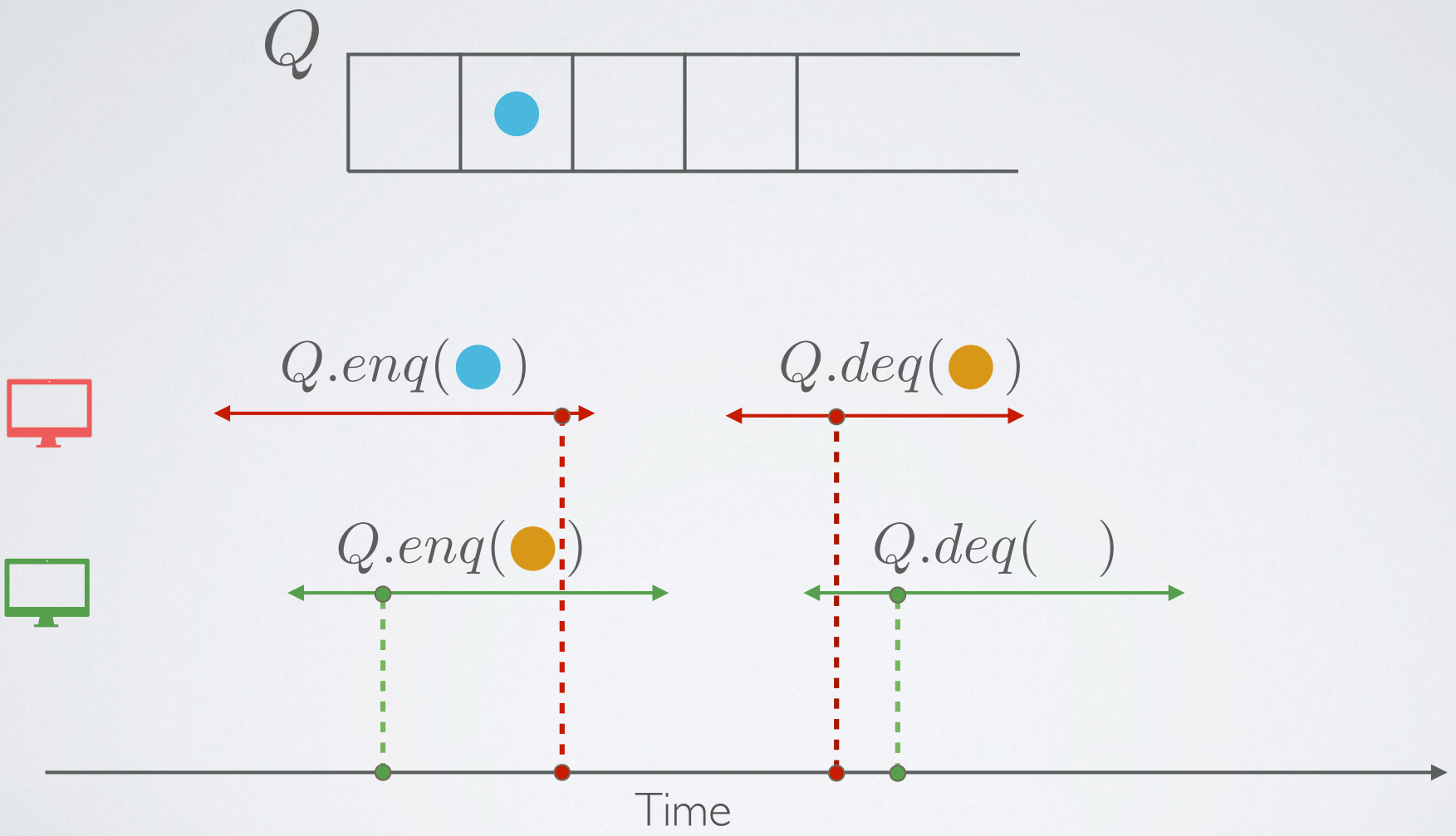
LINEARIZABLE QUEUE



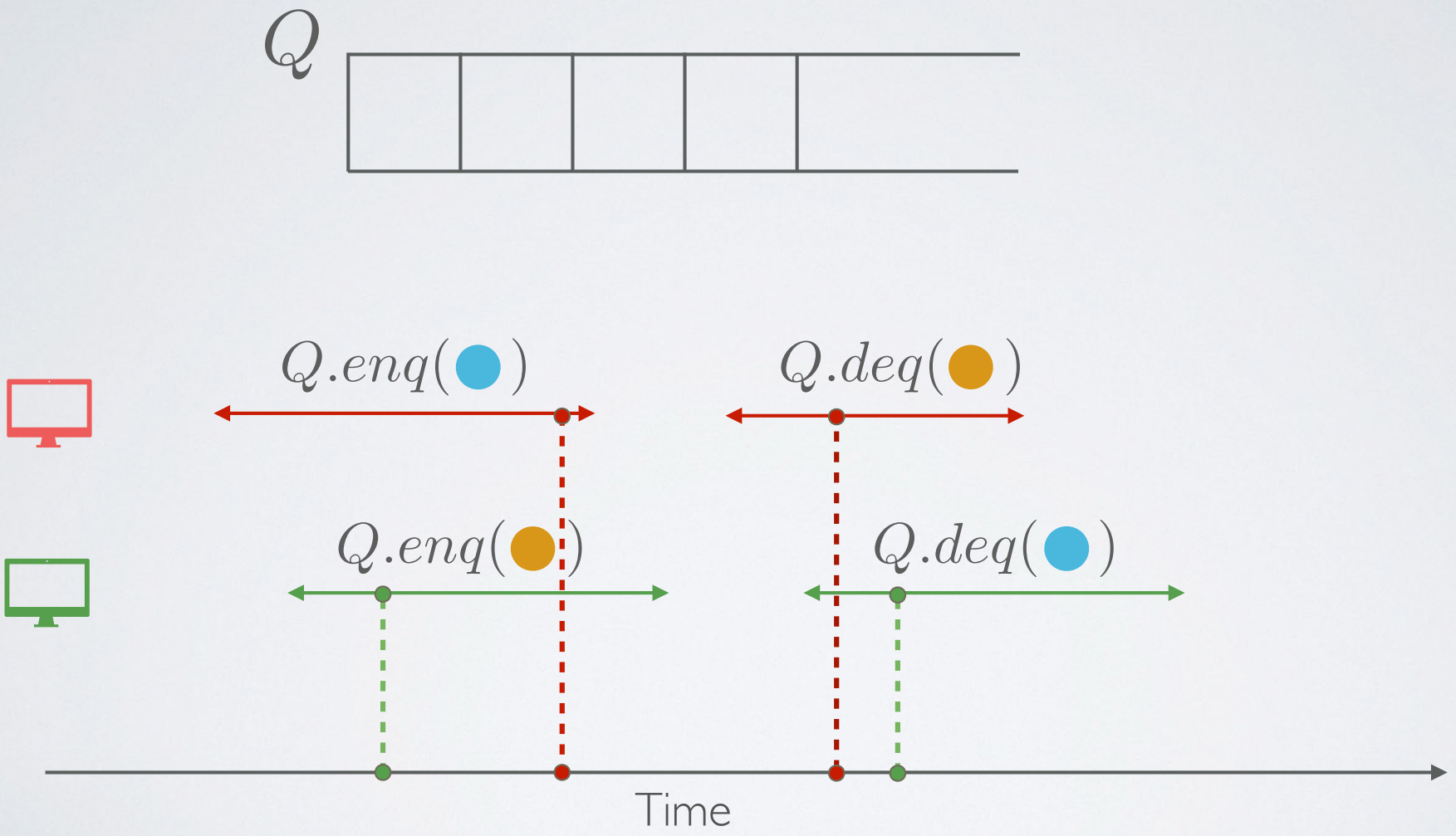
LINEARIZABLE QUEUE



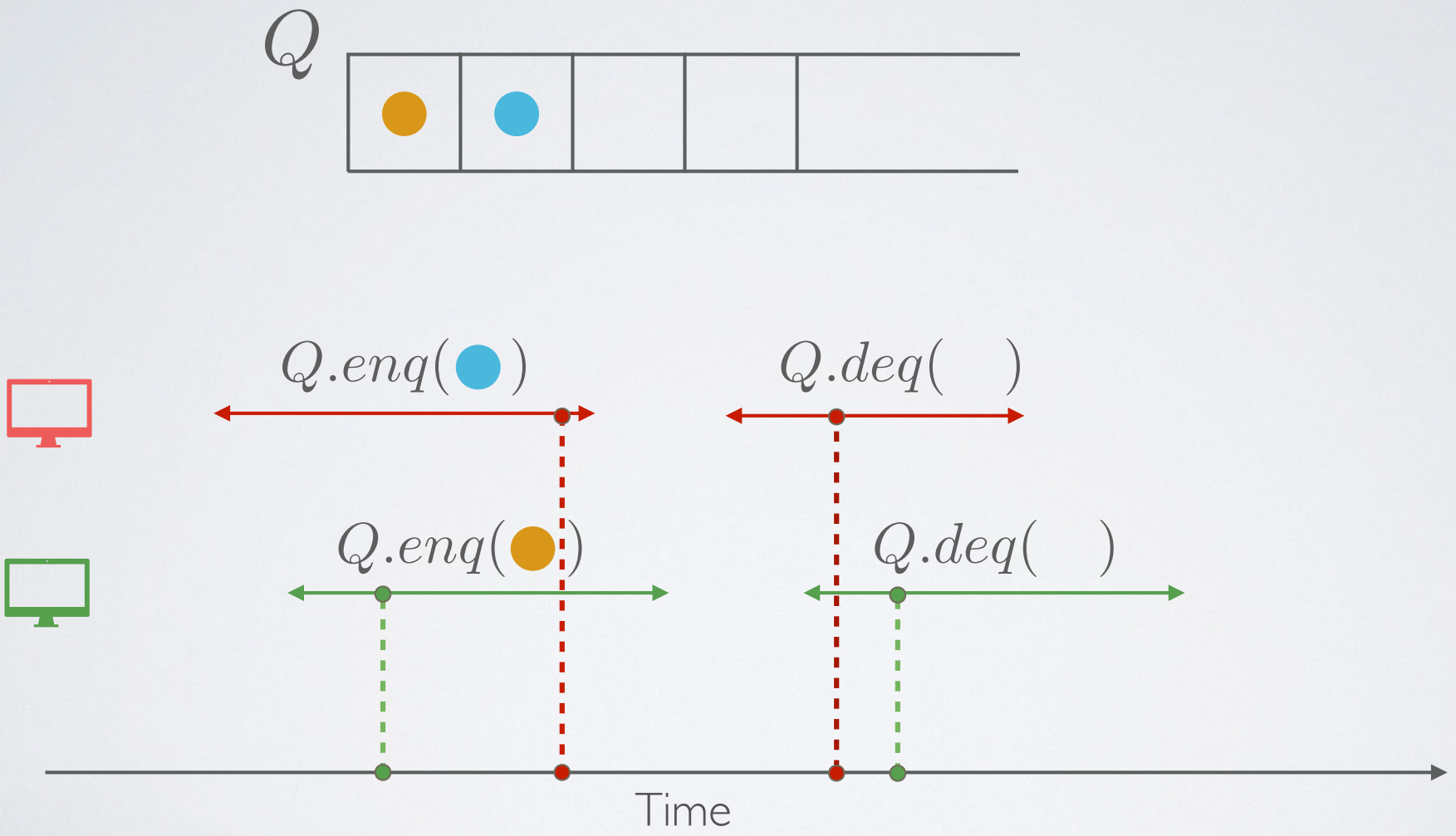
LINEARIZABLE QUEUE



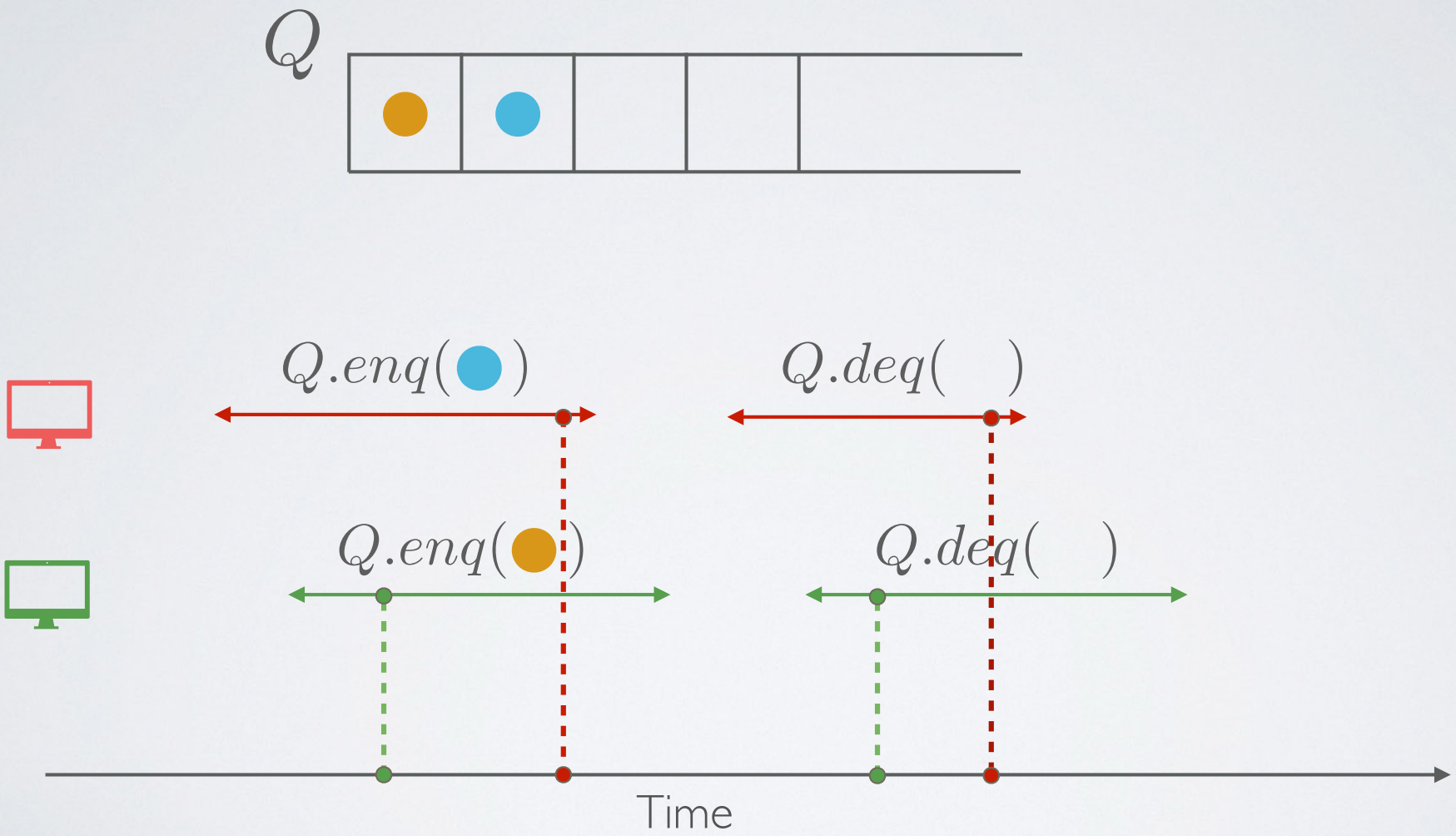
LINEARIZABLE QUEUE



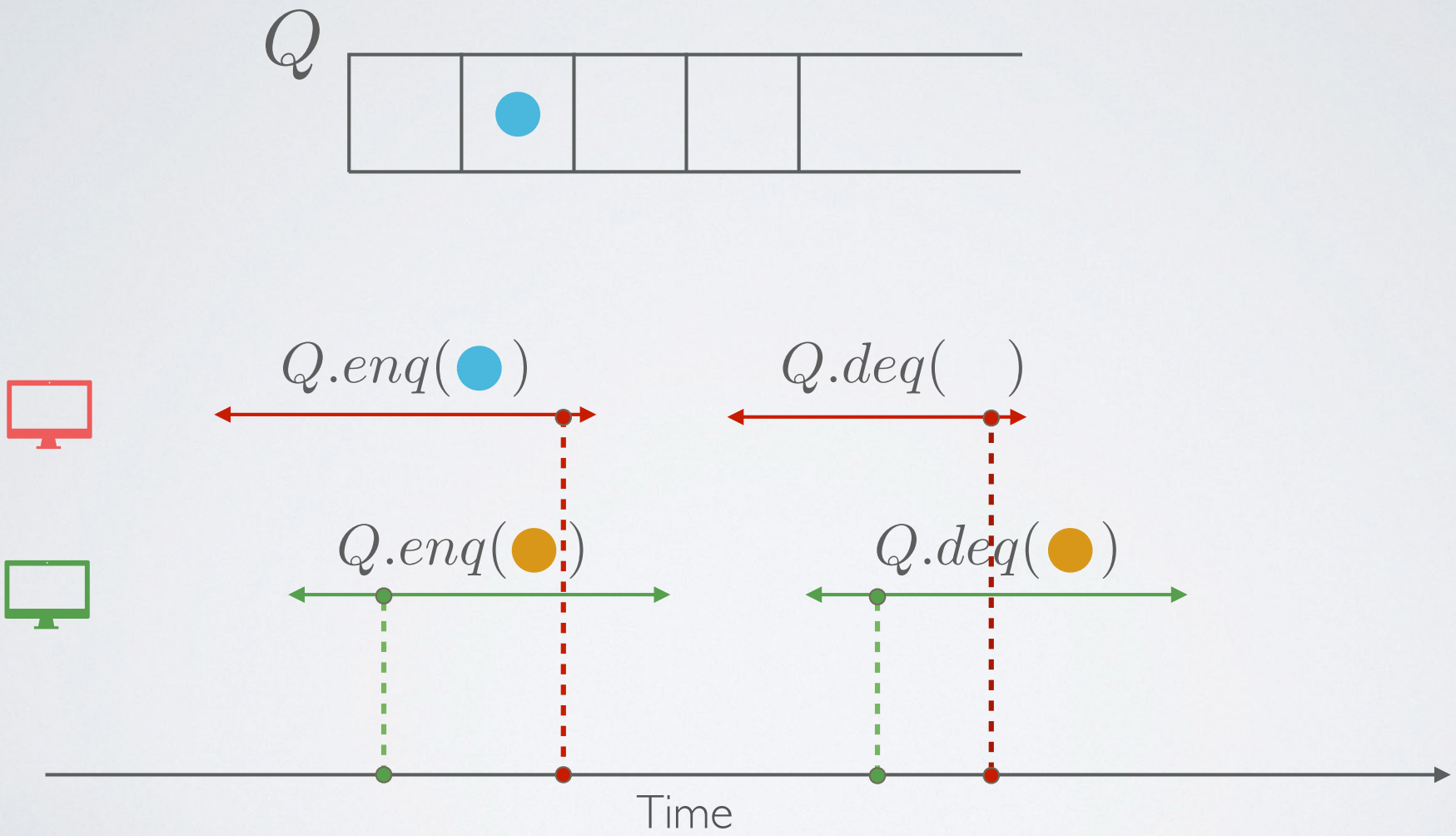
LINEARIZABLE QUEUE



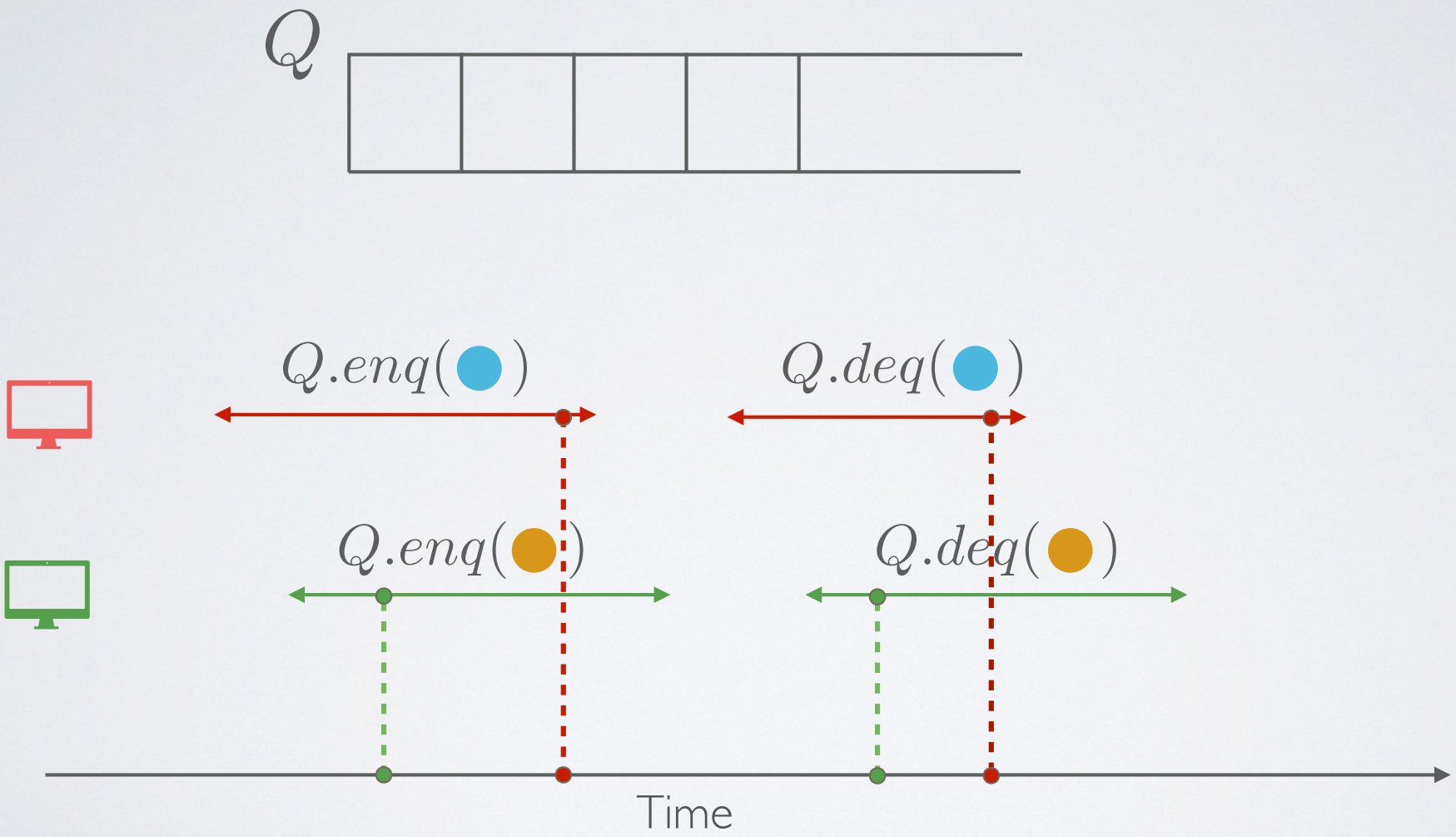
LINEARIZABLE QUEUE



LINEARIZABLE QUEUE



LINEARIZABLE QUEUE



An alternative…

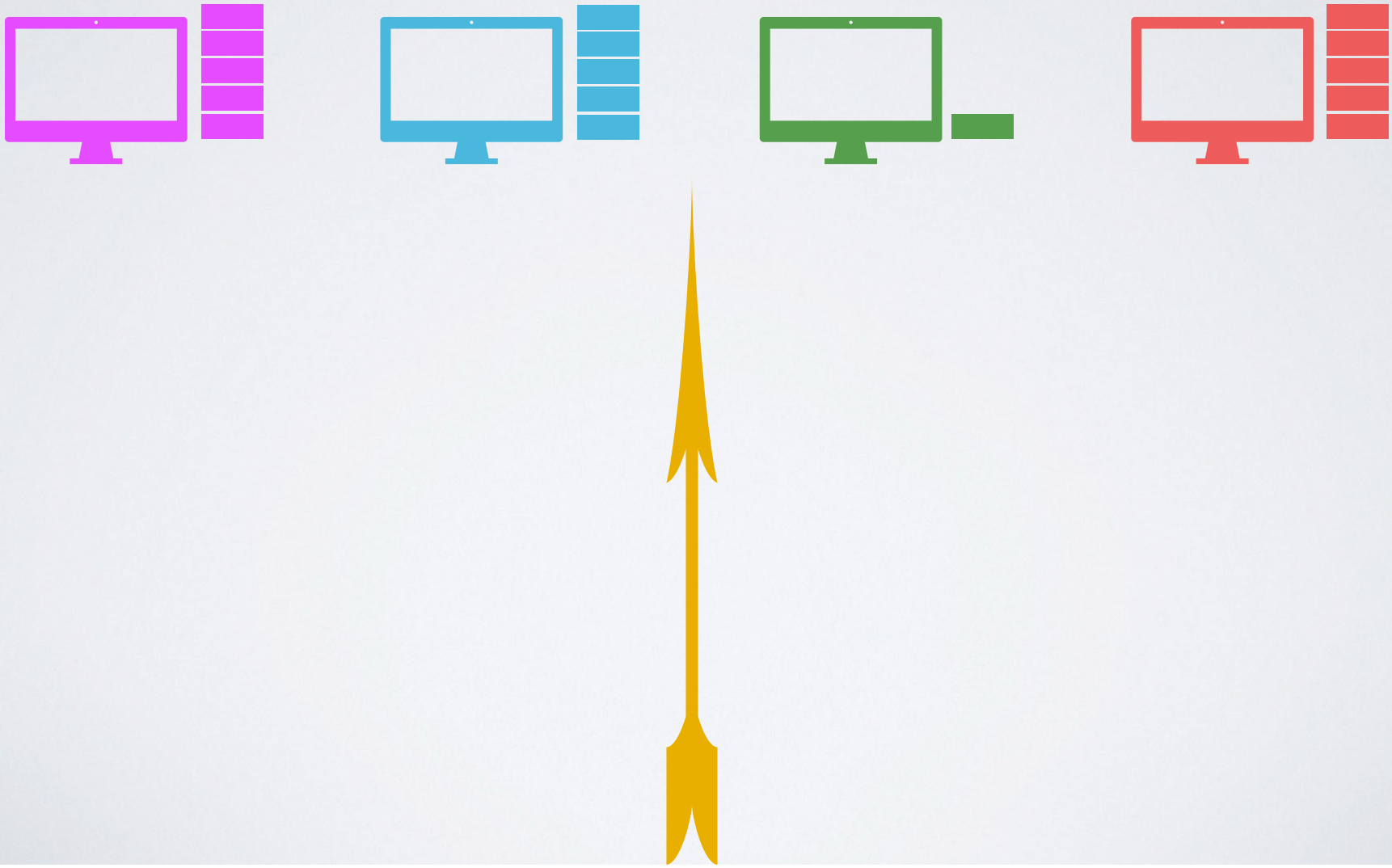
SEQUENTIAL CONSISTENCY

(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

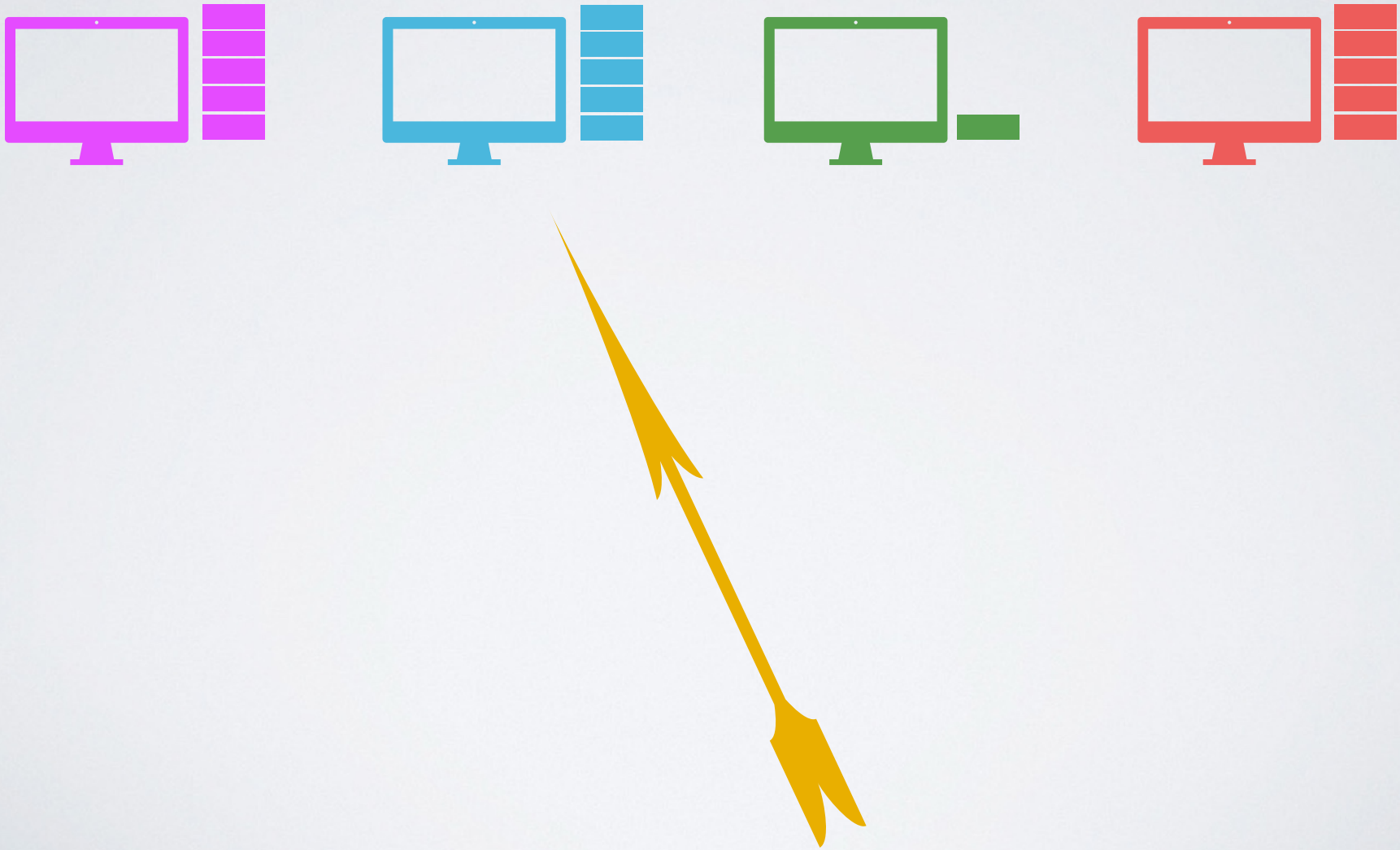
SEQUENTIAL CONSISTENCY

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



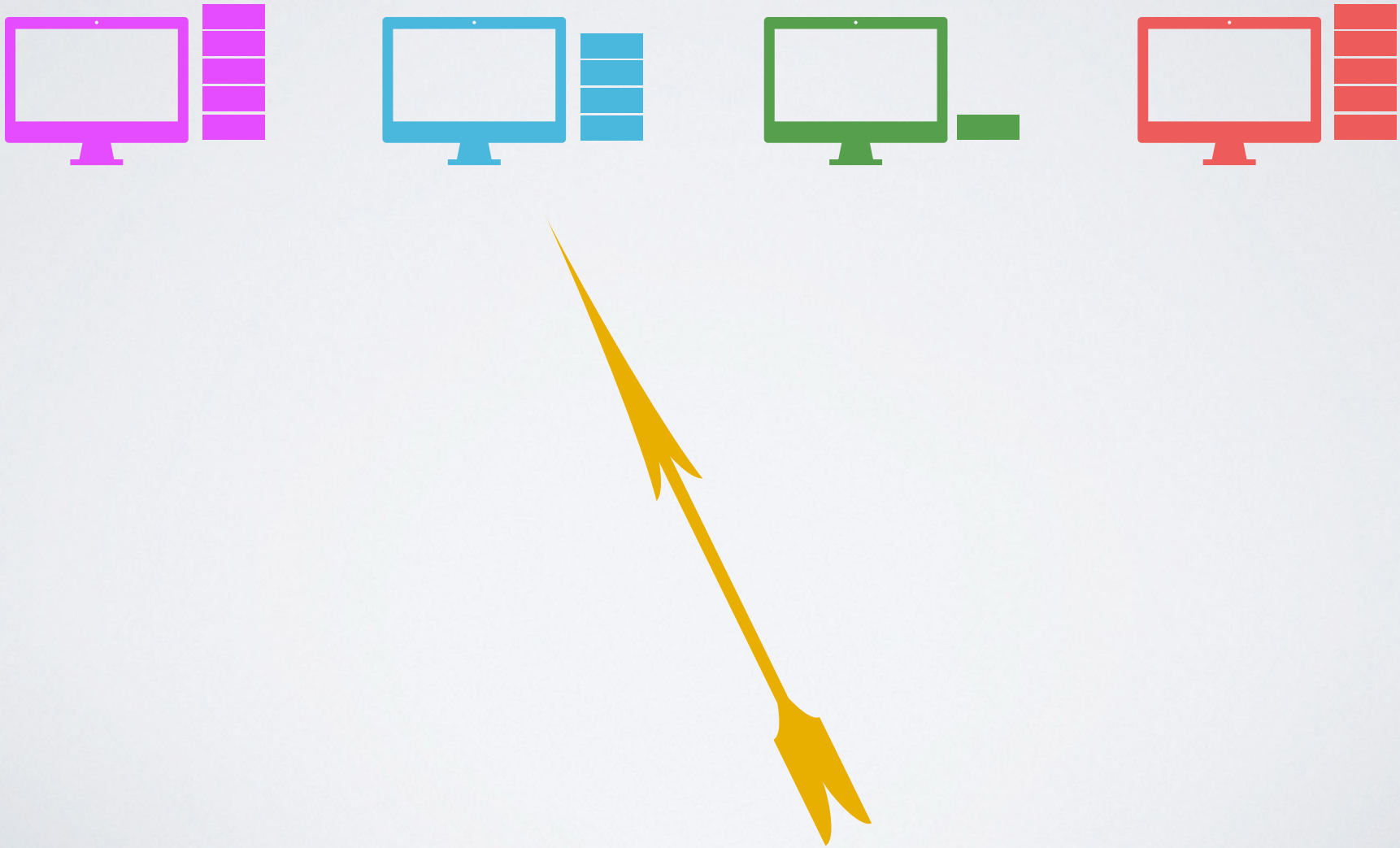
SEQUENTIAL CONSISTENCY

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



SEQUENTIAL CONSISTENCY

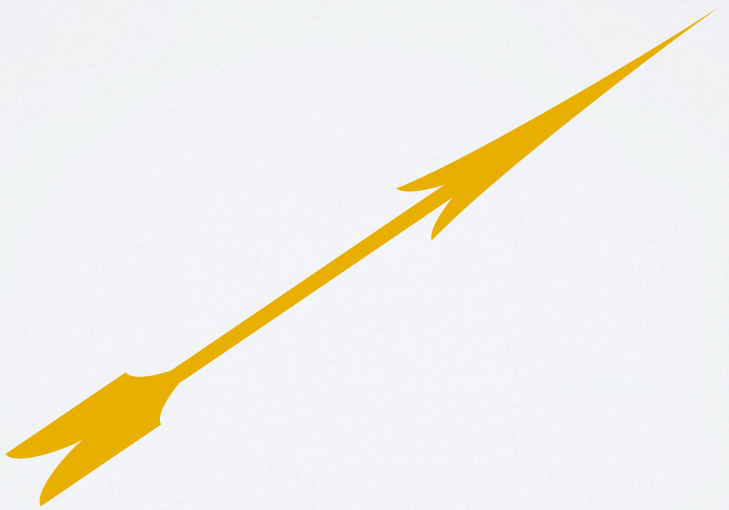
(LAMPORT "HOW TO MAKE A MULTIPROCESSOR COMPUTER THAT CORRECTLY EXECUTES MULTIPROCESS PROGRAMS", IEEE TOC C-28,9 (SEPT. 1979), 690-691)



SEQUENTIAL CONSISTENCY

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

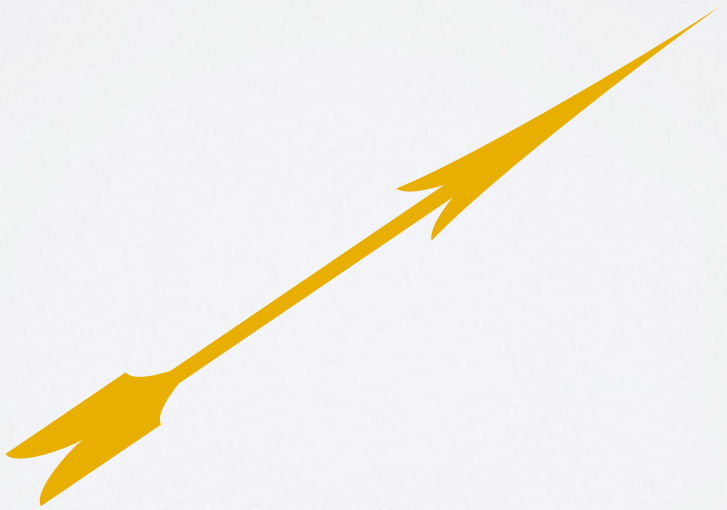




SEQUENTIAL CONSISTENCY

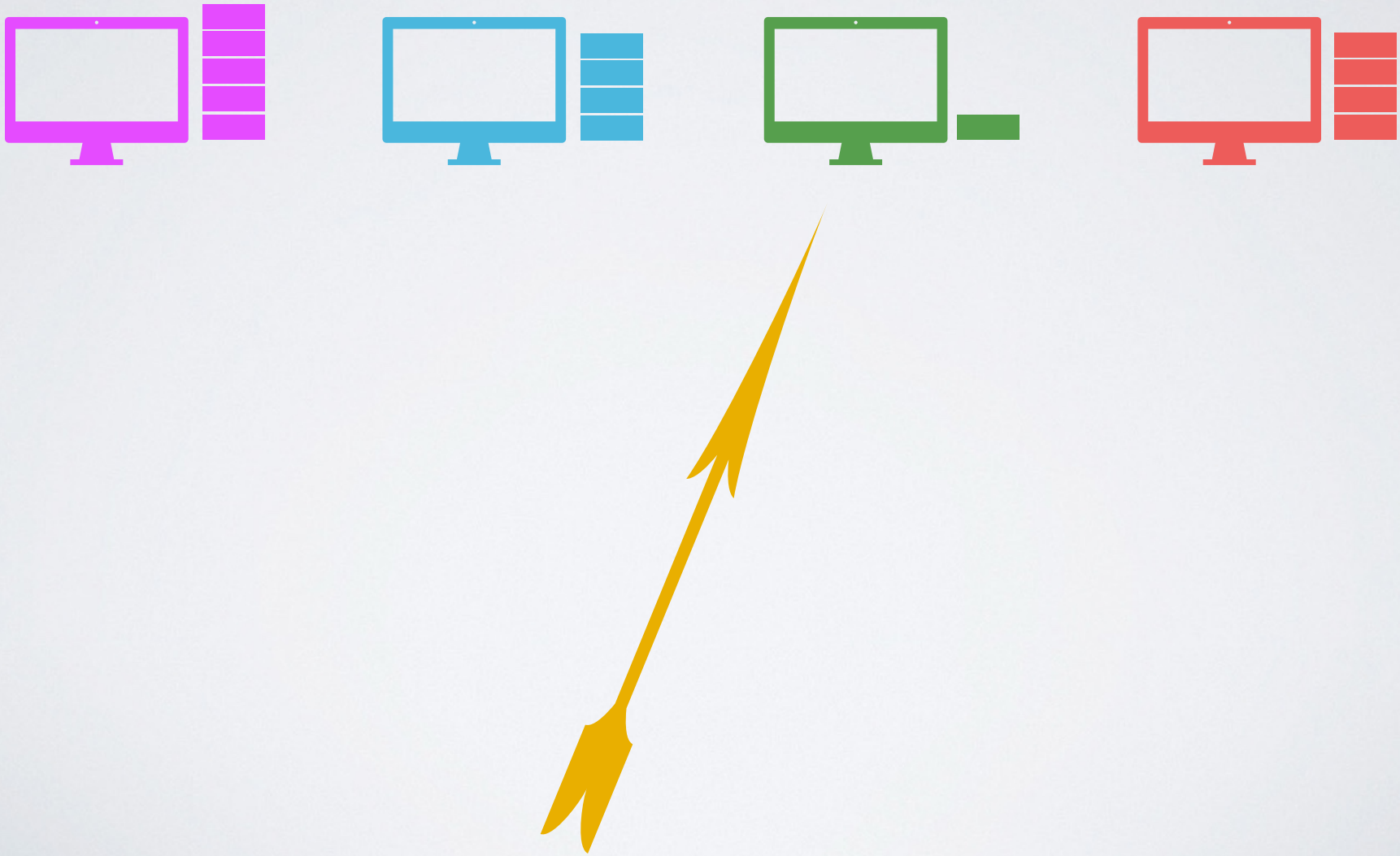
(LAMPORT "HOW TO MAKE A MULTIPROCESSOR COMPUTER THAT CORRECTLY EXECUTES MULTIPROCESS PROGRAMS", IEEE TOC C-28,9 (SEPT. 1979), 690-691)





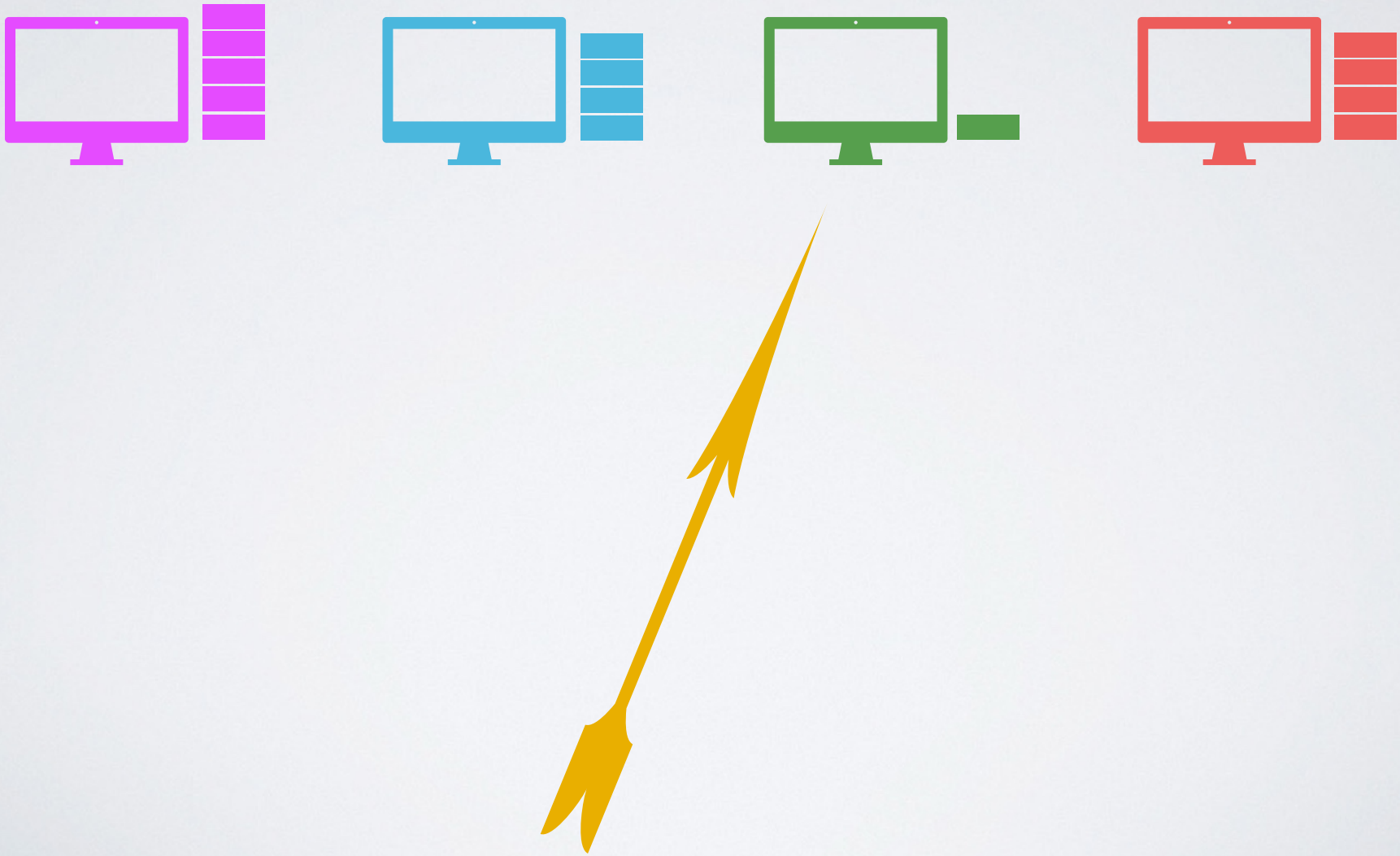
SEQUENTIAL CONSISTENCY

(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)



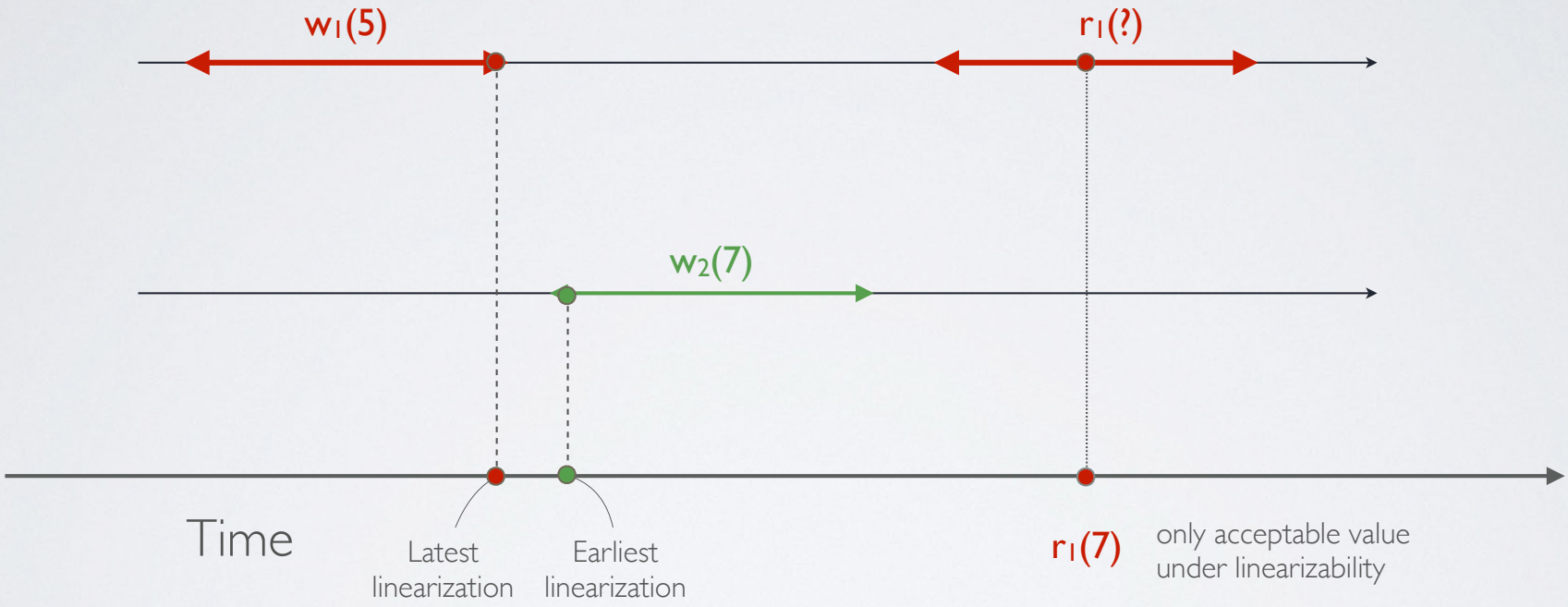
SEQUENTIAL CONSISTENCY

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



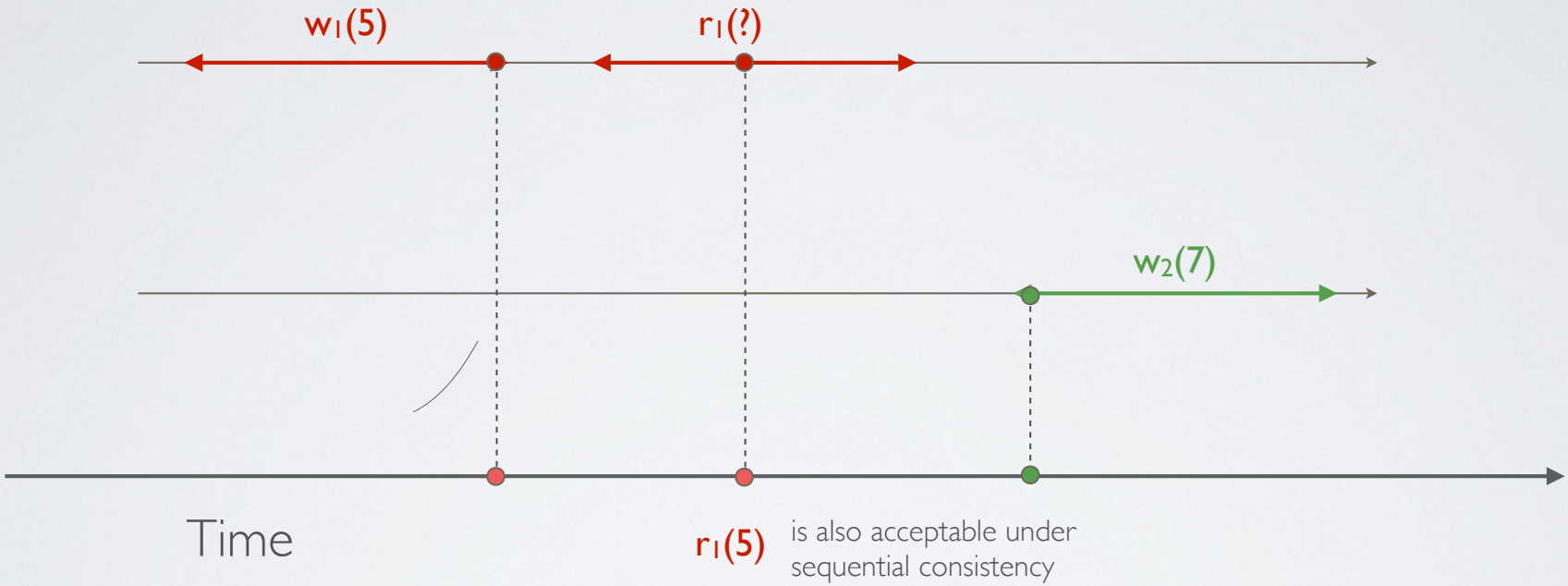
Temporal order of non-overlapping operations is preserved only for operations by the same thread

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



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



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

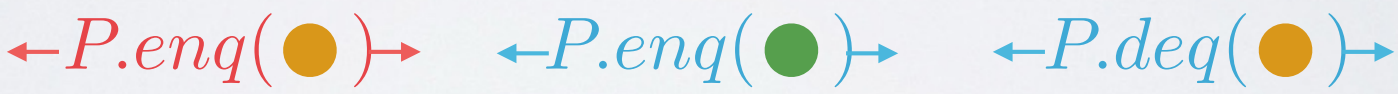
Time

THE CASE OF THE FIFO QUEUE



Queue *P* is sequentially consistent!







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



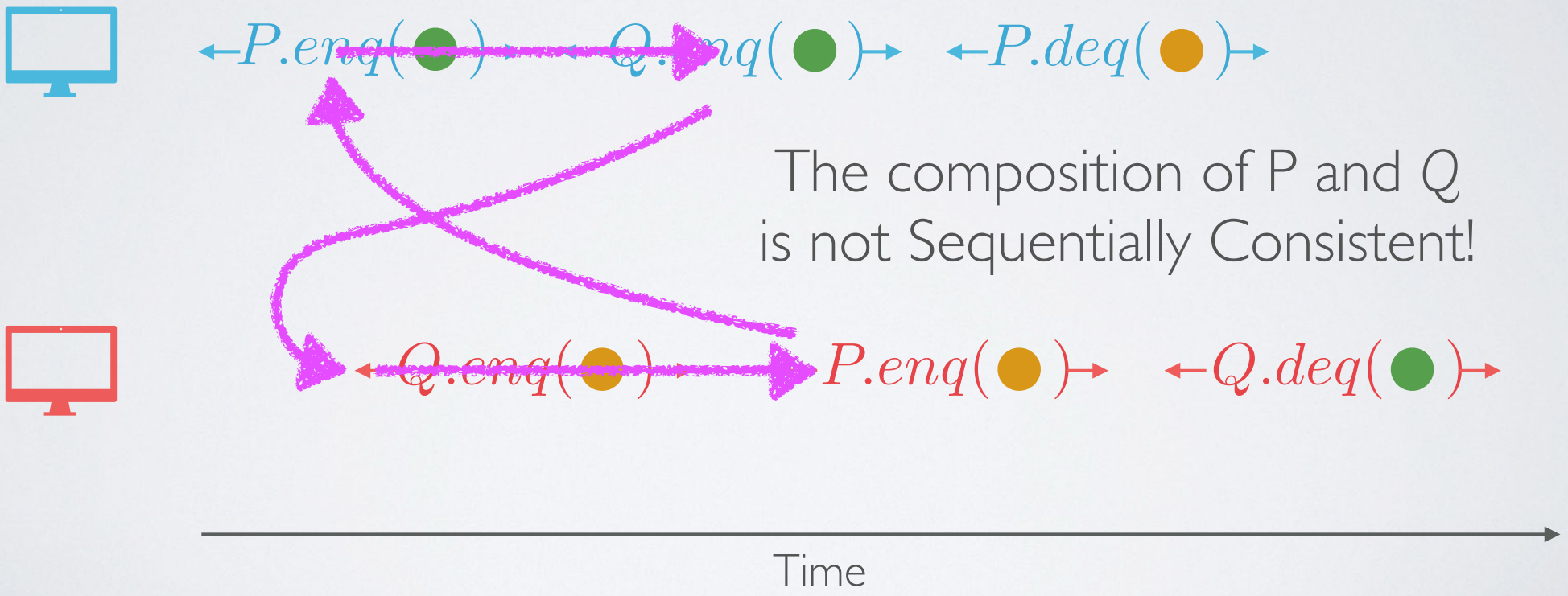
Queue Q is sequentially consistent!



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

Time

THE CASE OF THE FIFO QUEUE



The composition of P and *Q* is not Sequentially Consistent!

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

Consistency
linearizability



Availability
for updates

Partition
Tolerance

Eric Brewer's CAP "Theorem"

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

WHAT DOES C A 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


Distributed Systems
Databases

WHAT ABOUT DATABASES?

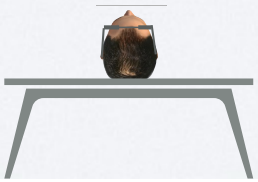
TRANSACTIONS TAKE TIME!

A CLASSIC HORROR STORY

Ease of Programming

Performance

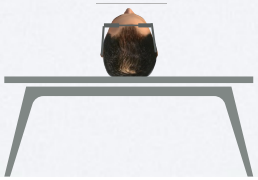




A CLASSIC HORROR STORY

Ease of Programming

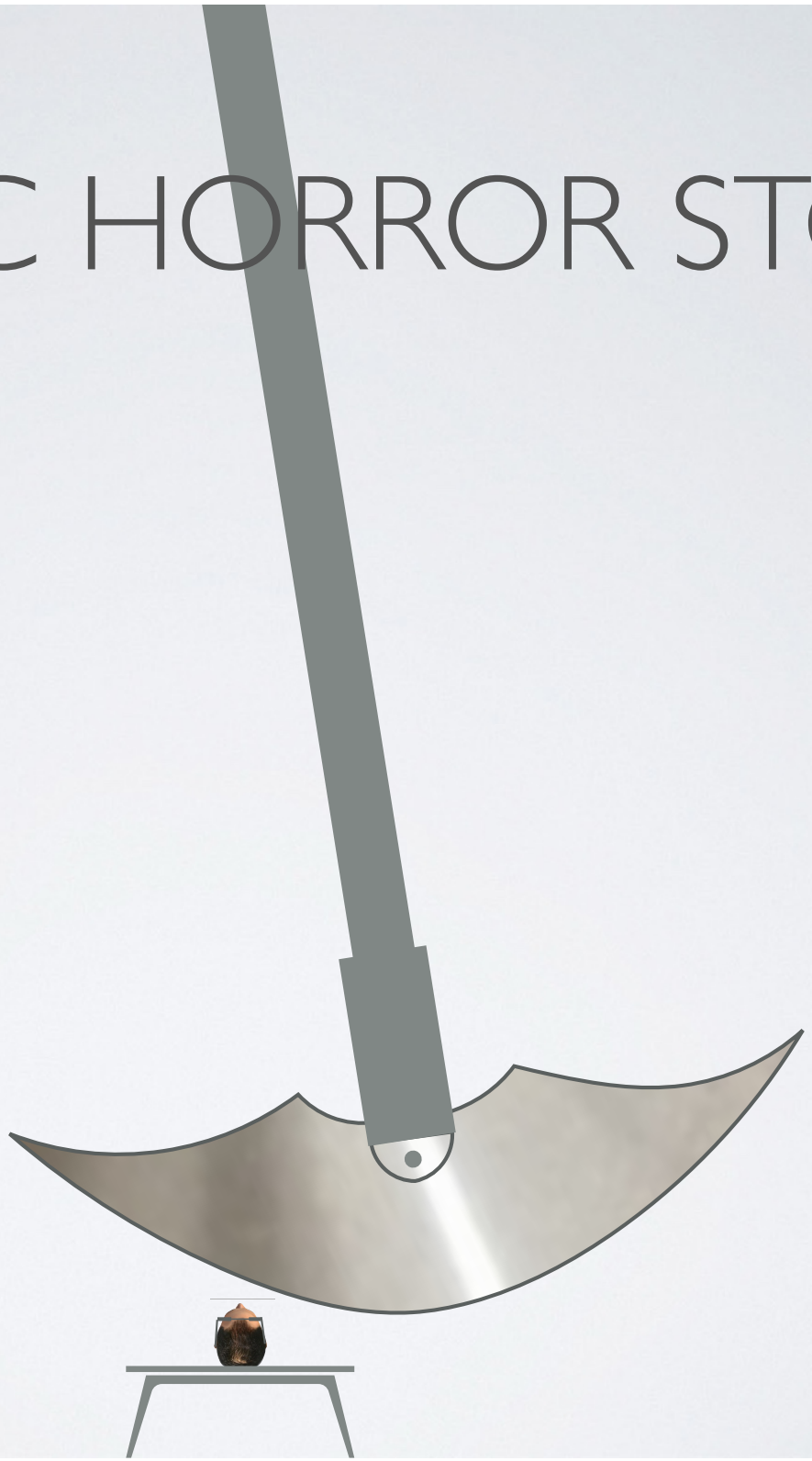
Performance



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.

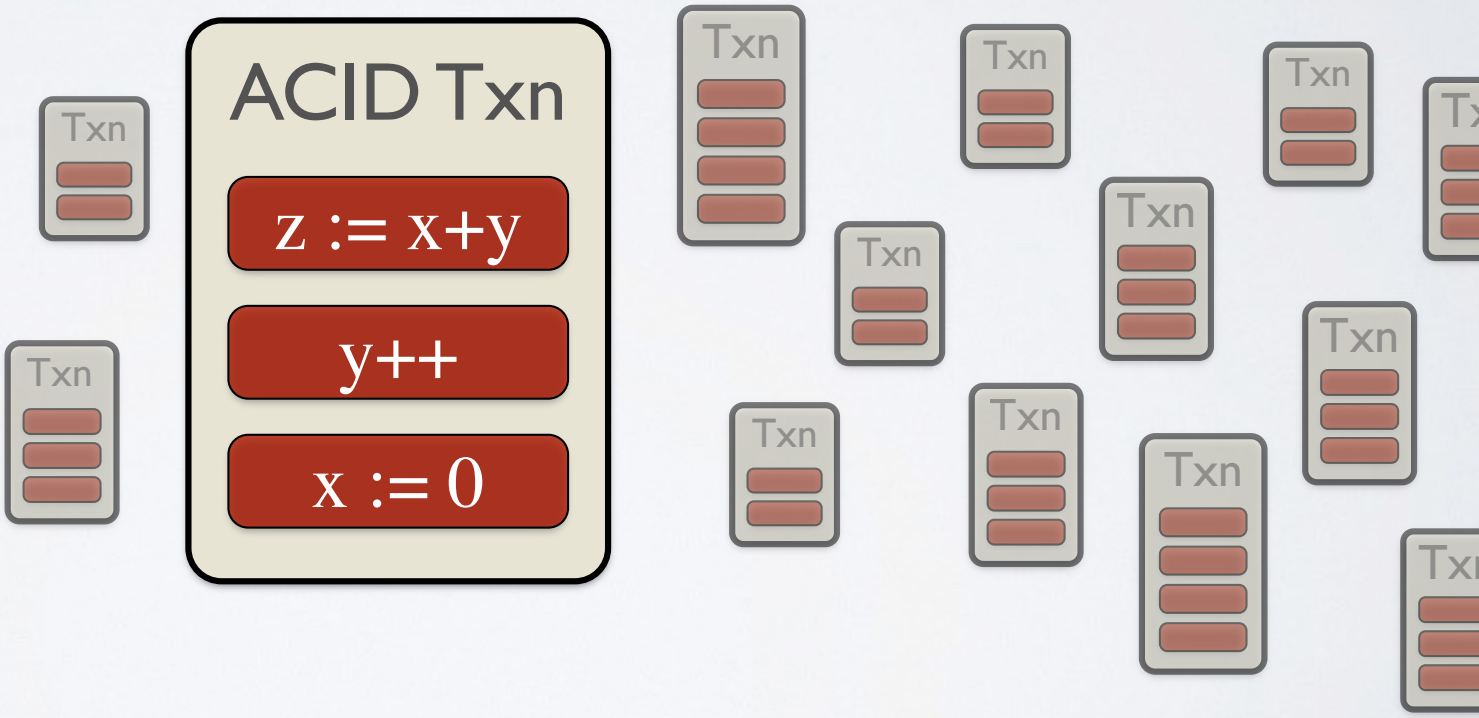
ACID TRANSACTIONS: SIMPLE AND POWERFUL

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

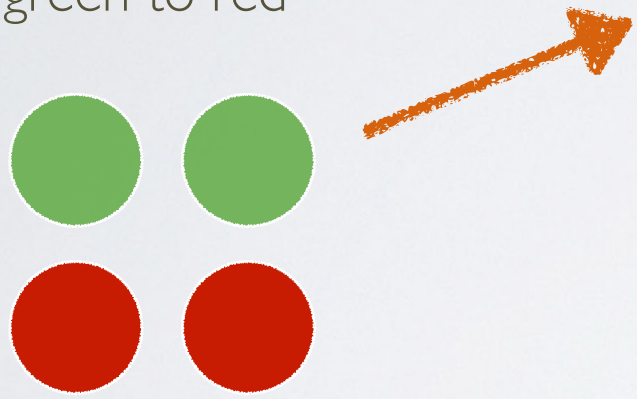
non-repeatable range reads

Snapshot Isolation

none of the above, but write skew

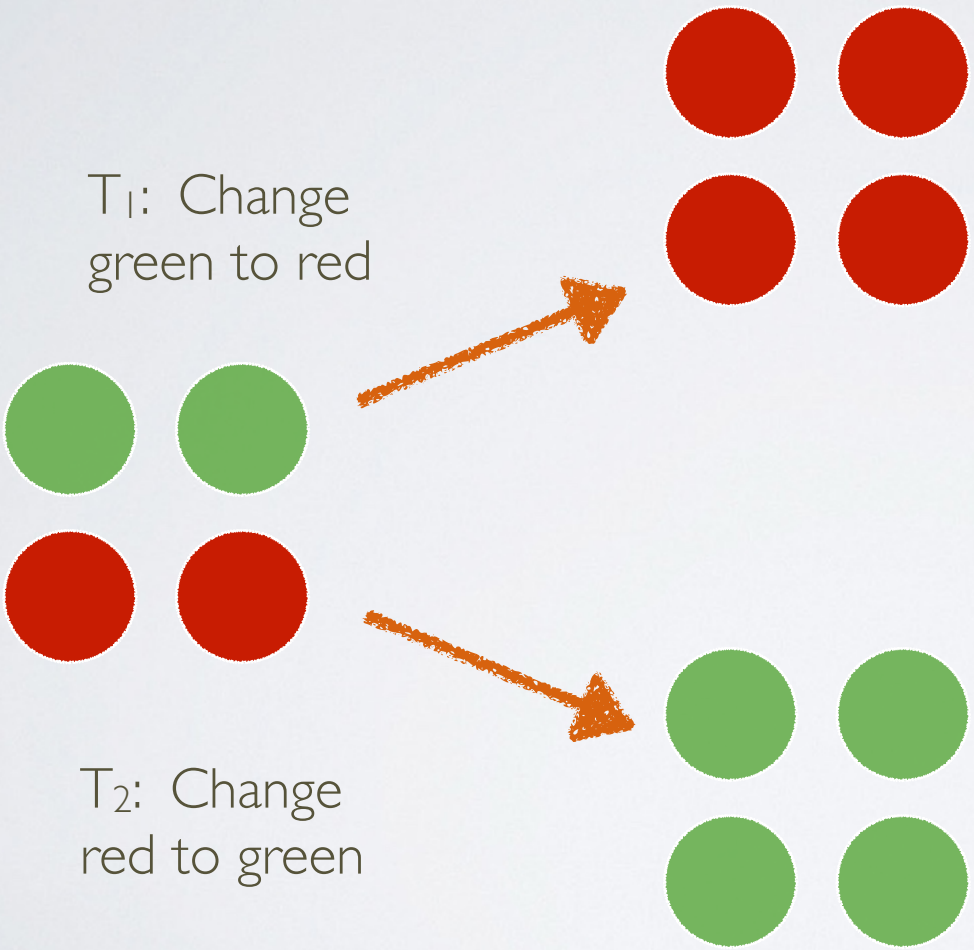
WRITE SKEW ANOMALY

T_1: Change green to red

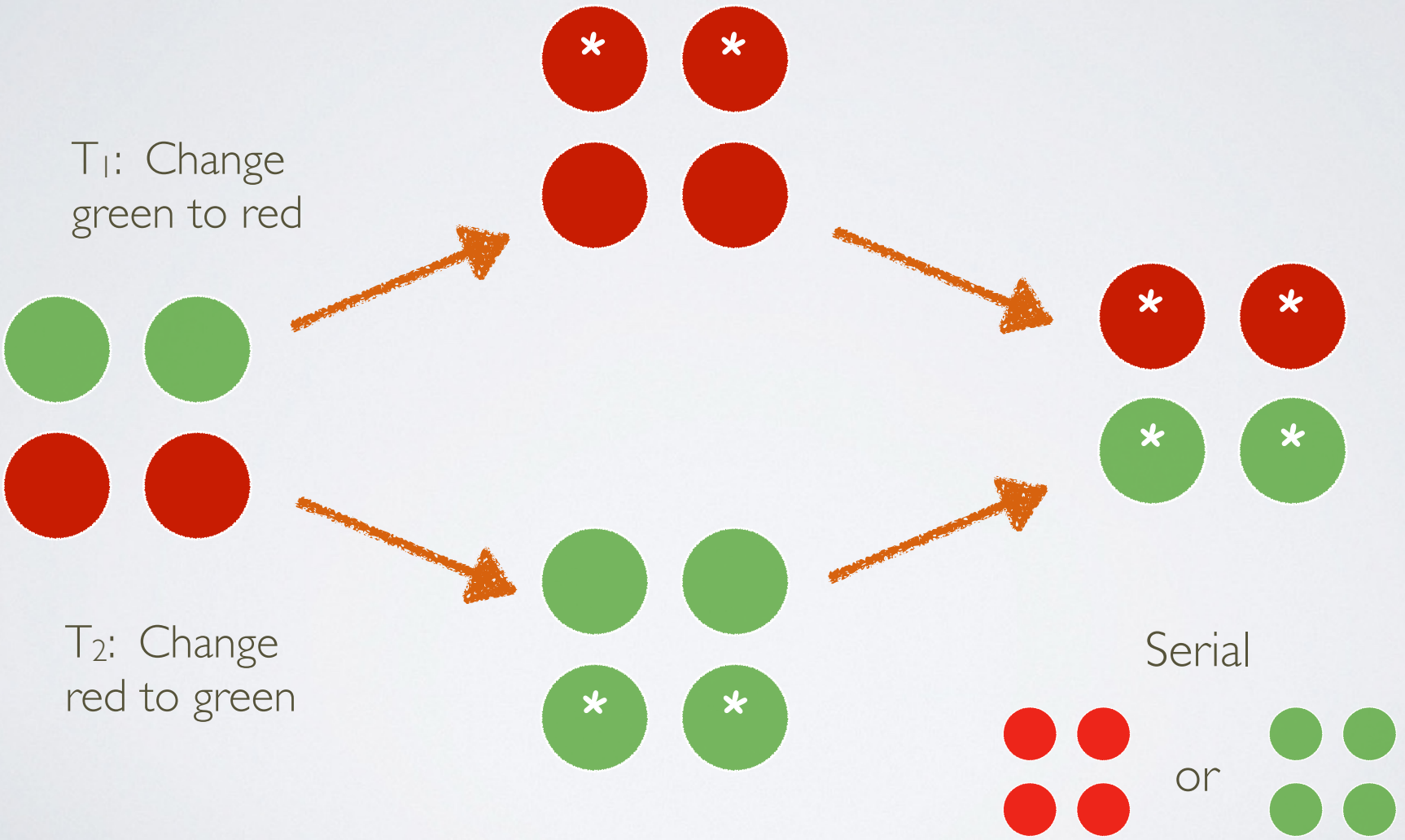


T_2: Change red to green

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

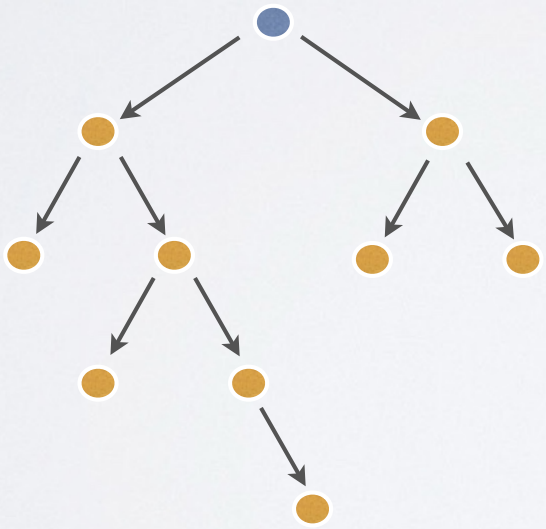
- Facebook, Twitter, Amazon aim for ALPS
  - Availability
  - low Latency
  - Partition tolerance
  - Scalability
- What about consistency?
  - Tension (you guessed it) between performance and ease of programming




GOSSIP

DATA DISSEMINATION

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

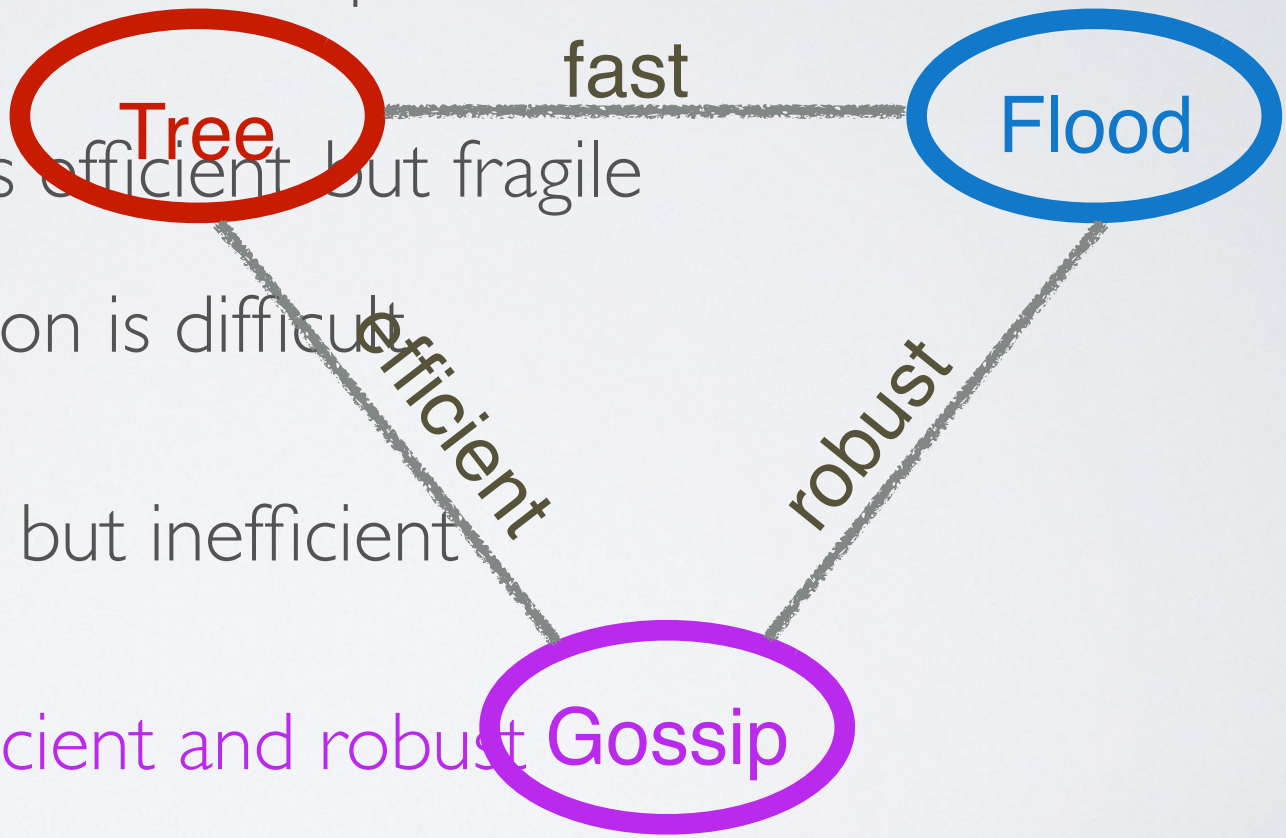


DATA DISSEMINATION

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

DATA DISSEMINATION

- Want efficiency, robustness, speed, scale
- Tree distribution is efficient but fragile
  - plus, configuration is difficult
- Flooding is robust, but inefficient
- Gossip is both efficient and robust
  - but has relatively high latency

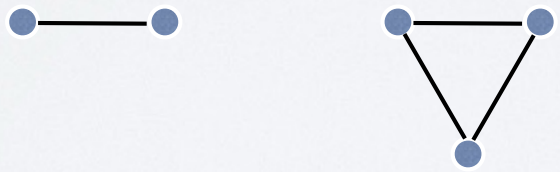


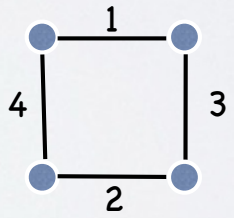
HISTORY

- Gossips and Telephones (Baker, Shostack 1972)

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

$n = 1$ $n = 2$ $n = 3$ $n = 4$





0 1 3 4