Distributed shared memory

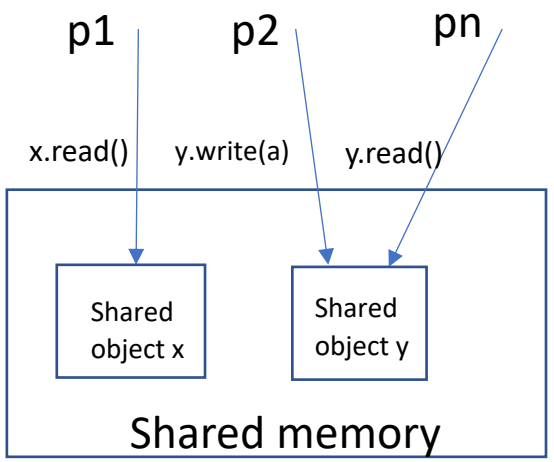
.

Emmanuelle Anceaume

emmanuelle.anceaume@irisa.fr

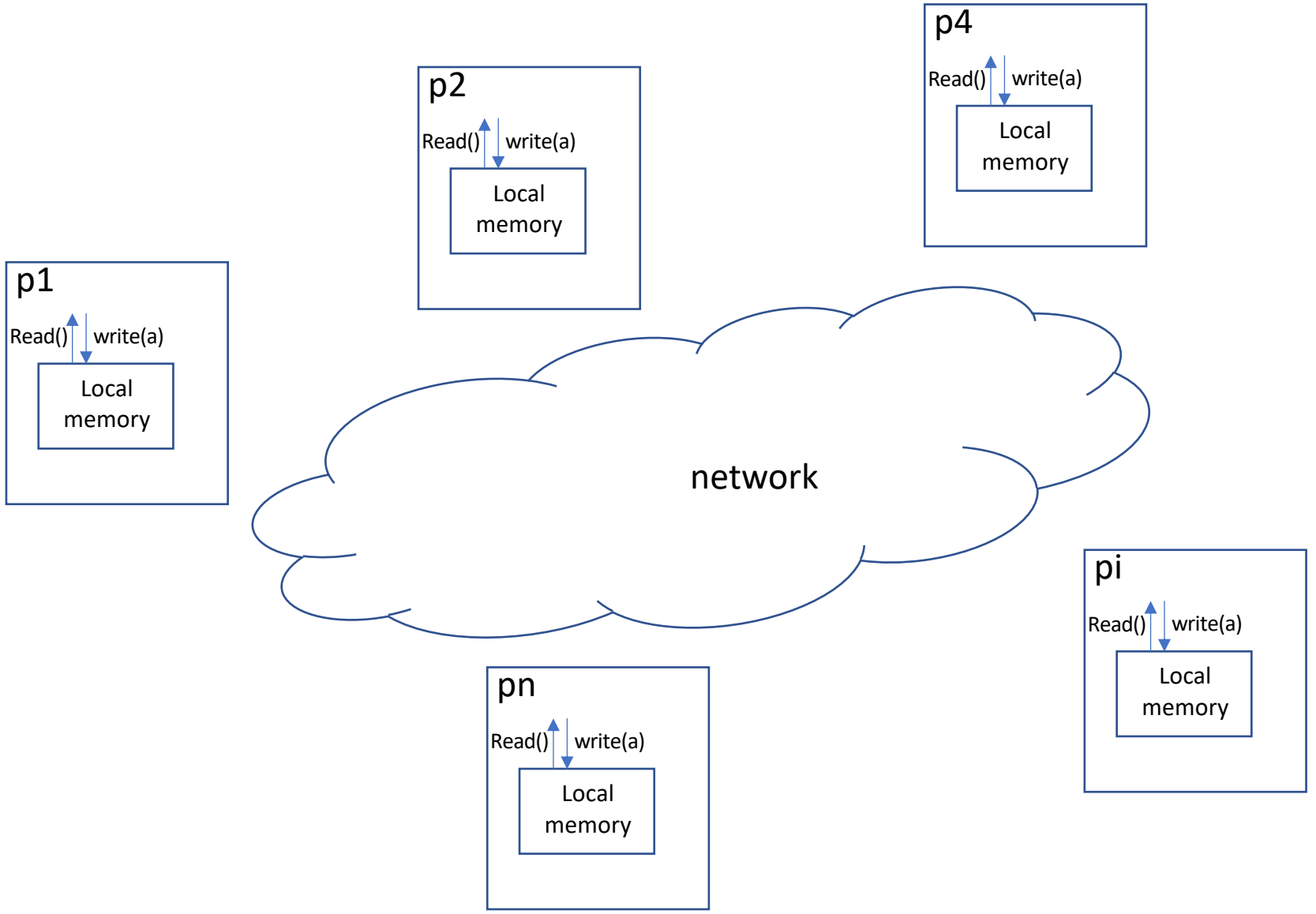
Distributed shared memory

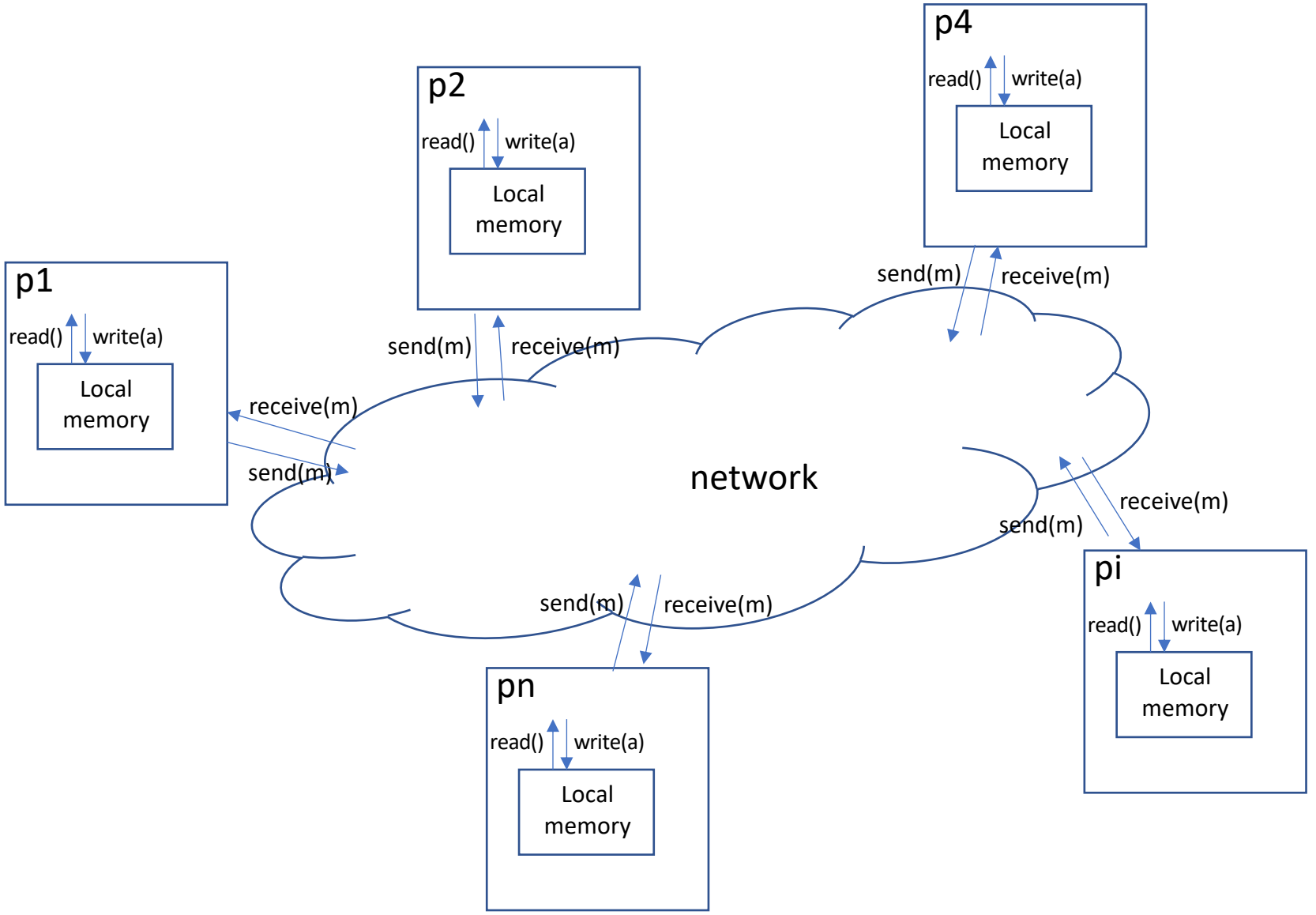
- We have seen in the previous lesson how processes can synchronize their activity (e.g. mutual exclusion) and communicate by accessing shared objects



Distributed shared memory

- The objective of this lesson is to simulate a shared memory, i.e., to provide the illusion to each node that it has access to a memory physically shared with all the other nodes
- This is achieved by having each node maintaining its own local memory consistent with the one of the other nodes by synchronizing the different updates by sending and receiving messages


p1
Read()
write(a)
Local memory
p2
Read()
write(a)
Local memory
p4
Read()
write(a)
Local memory
network
pn
Read()
write(a)
Local memory
pi
Read()
write(a)
Local memory


p1
read()
write(a)
Local memory
p2
read()
write(a)
Local memory
p4
read()
write(a)
Local memory
pi
read()
write(a)
Local memory
pn
read()
write(a)
Local memory
network
receive(m)
send(m)
send(m)
receive(m)
send(m)
receive(m)
send(m)
receive(m)
send(m)
receive(m)

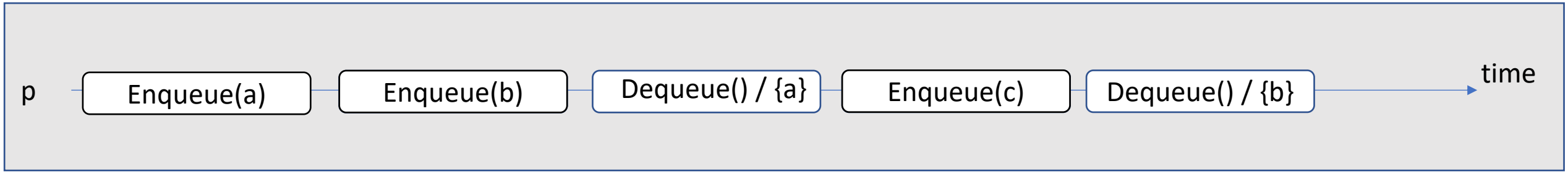
Distributed shared memory

- We start this lesson with the specification of two asynchronous shared memory to be simulated, introducing two notions of consistency:
  - Linearizability
  - Sequential consistency

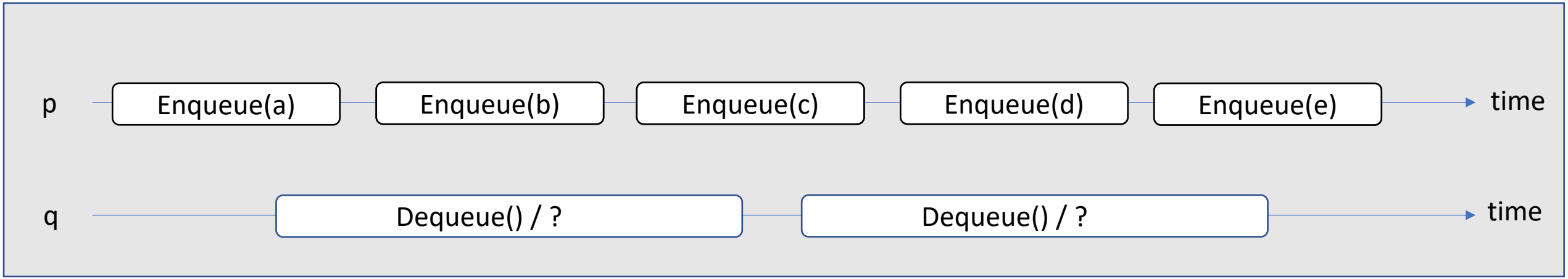
- Next we will present how to implement a distributed shared memory satisfying linearizability or sequential consistency

Consistency criteria

- A consistency criteria is a property that links a sequential specification and all the concurrent executions
- A strong consistency criteria can be seen as a way to look an execution so that it appears sequential and in accordance with the sequential specification of the accessed objects
- We will look at two strong consistency criteria:
  - Linearizability, sequential consistency



Sequential execution of a queue



Concurrent execution on a queue

The first figure shows a sequential execution of a system made of a single process accessing a queue. This execution respects the sequential specification of a queue.
The second figure represents a concurrent execution by p and q on a shared queue. Process p acts as a producer and process q acts as a consumer. Operations overlap.

The questions raised in this figure are what elements can be dequeued by process q to appear sequential.
The role of strong **consistency criteria** exactly address this question

Consistency criteria

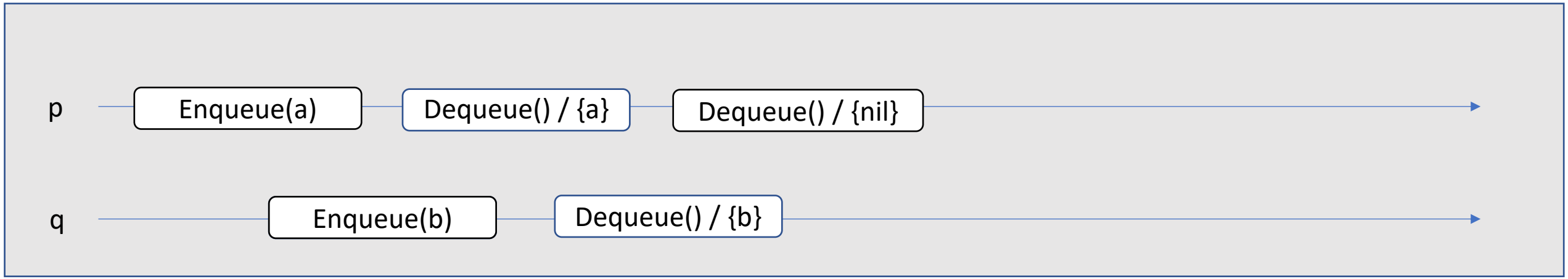
- We have seen that the sequential specification of an object is the set of admitted sequential executions
- A consistency criteria characterizes the set of executions which are admissible for a shared object
- The concurrent specification of a shared object is thus a set of admissible concurrent executions

Linearizability

- Formally, a complete history H is linearizable if there exists a permutation L of H such that
  1. H and L are equivalent (i.e. for any process p, H|p=L|p)
  2. L is a sequential history
  3. L is legal (i.e., for any object X of L, L|X respect the sequential specification of X)
  4. $->_H \subseteq ->_L$ (i.e., L respects the real-time order of none interleaving operations)

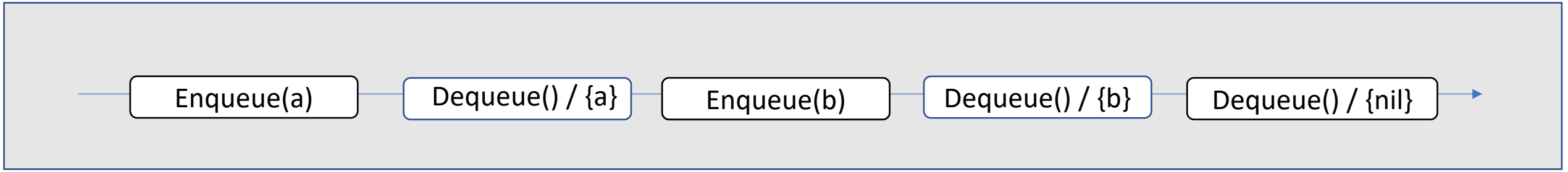
  L represents a history in which all the events of H are executed sequentially, such that the precedence on the operations of H is respected.

  L is called a linearization of H or a sequential witness of H

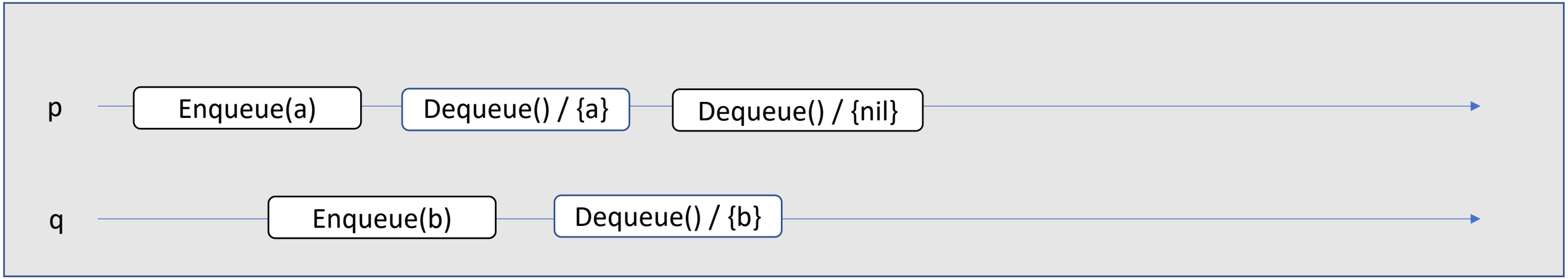


Concurrent execution

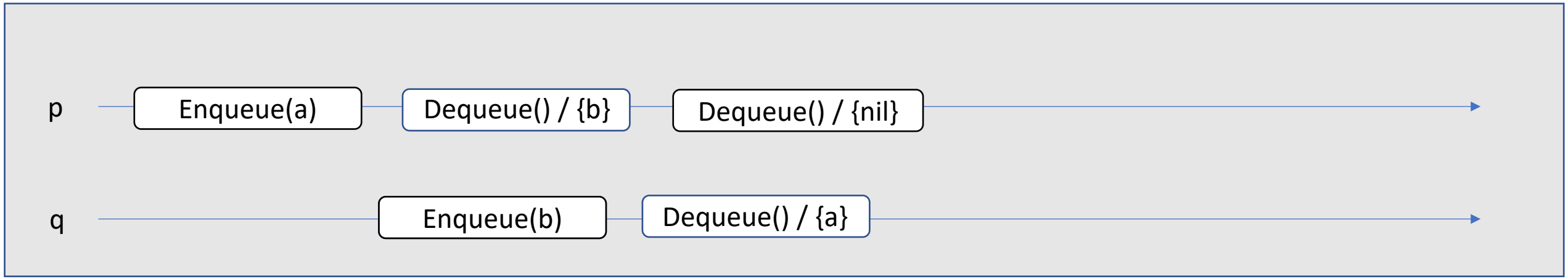
Is this concurrent execution linearizable ?



A possible permutation of the concurrent execution H

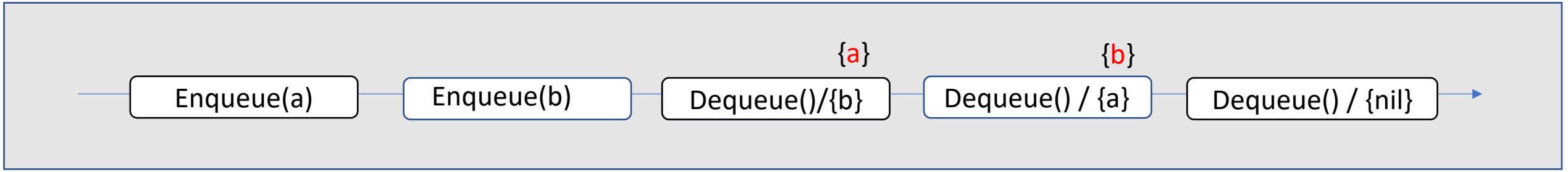


The concurrent execution H is linearizable

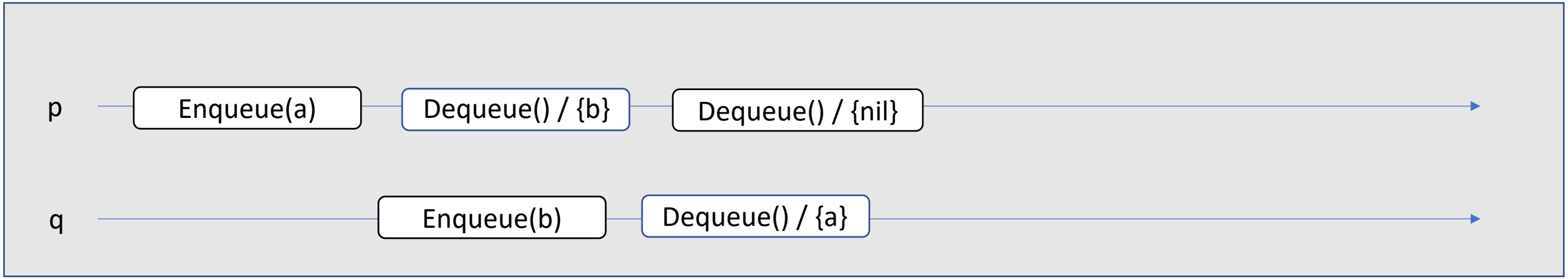


Concurrent execution H

Is this concurrent execution H linearizable ?



A possible permutation of the concurrent execution



Concurrent execution H

The real-time order of the events on all the processes is respected by the sequential execution
but the specification of the queue is not.
Thus H is not linearizable

Composability

- An important feature of linearizability is the fact that it is composable
- A property P is said to be composable (we also say local) if whenever P holds for each object of the system, P holds for the entire set of objects

  For each history H, we have $\forall X\ H|X \in P$ iff $H \in P$

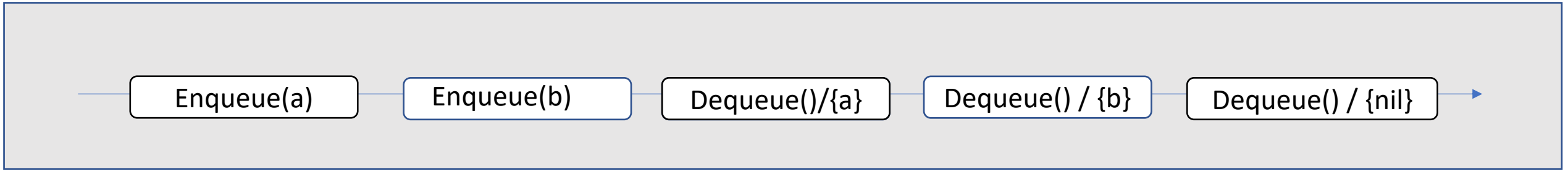
- Intuitively, composability enables us to derive the correctness of a composed system from the correctness of the components

Sequential consistency

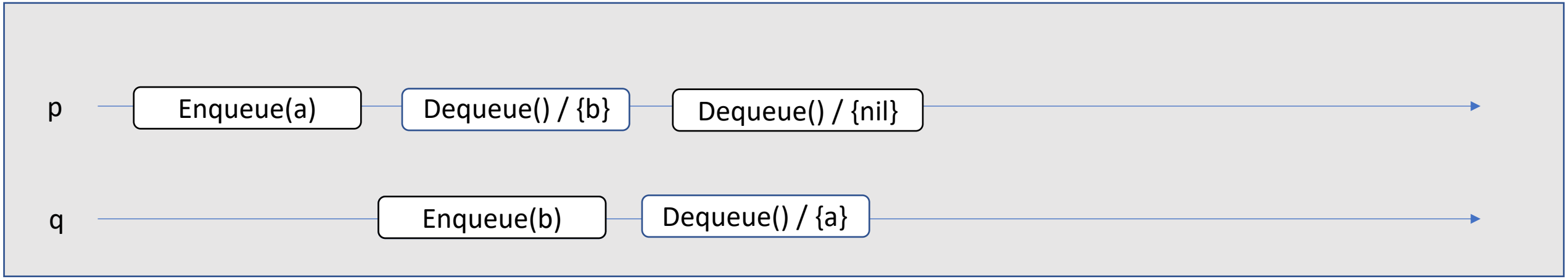
- It may happen that the real-time order at which events occur on different processes is not important
- The sequential consistency exploits this idea
- A history is sequentially consistent if there is a way to reorder the operations in the history that
  - (1) respects the sequential specification of the objects
  - (2) respects the ordering of operations on the same process

Sequential consistency

- More formally, a history H is sequentially consistent if there exists a permutation L of H such that
  1. H and L are equivalent (i.e. for any process p, H|p=L|p)
  2. L is a sequential history
  3. L is legal (i.e., for any object X of L, L|X respect the sequential specification of X)
  4. $->_{H|p} \subseteq ->_{L|p}$ (i.e., L respects the order of operations at process p)



A possible permutation of the concurrent execution



Concurrent execution H

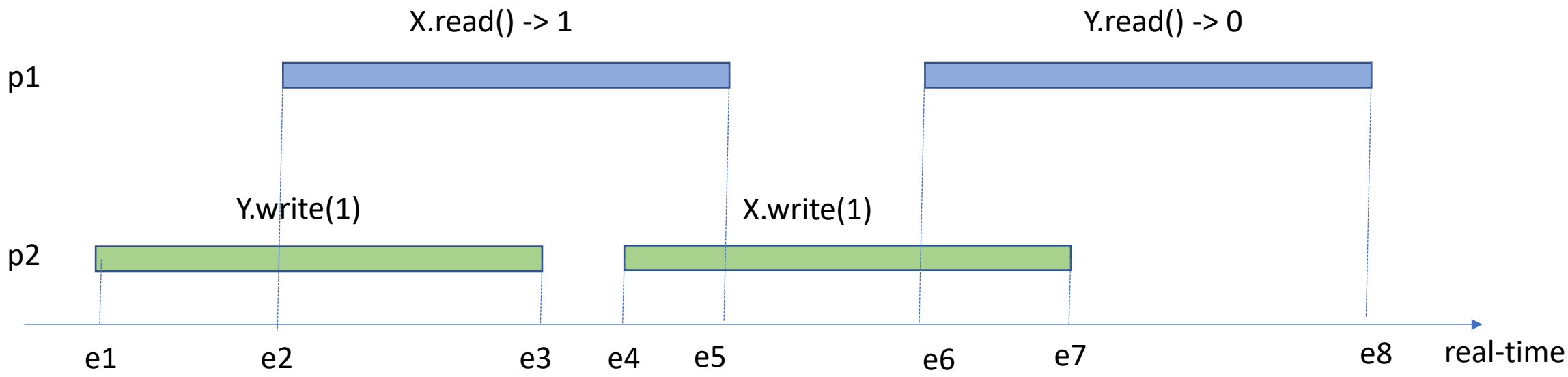
The real-time order of the events on each process is respected by the sequential execution and the specification of the queue is respected.
Thus H is sequentially consistent

Sequential consistency is not composable

- Sequential consistency has no requirement regarding the real-time order of operations that are invoked by different processes
- A major drawback of sequential consistency is that it is not a local property (it is not compositional)

A concurrent history H

Both registered X and Y are initialized to 0



Let us consider H|X { e2,e4,e5,e7} the projection of H on X.
Let S1={ e4,e7,e2,e5} be the witness history of H|X

1. H|X and S1 are equivalent
2. S1 is sequential
3. S1 is legal : it belongs to the specification of X

- S1 is sequentially consistent. Note that it H is also linearizable

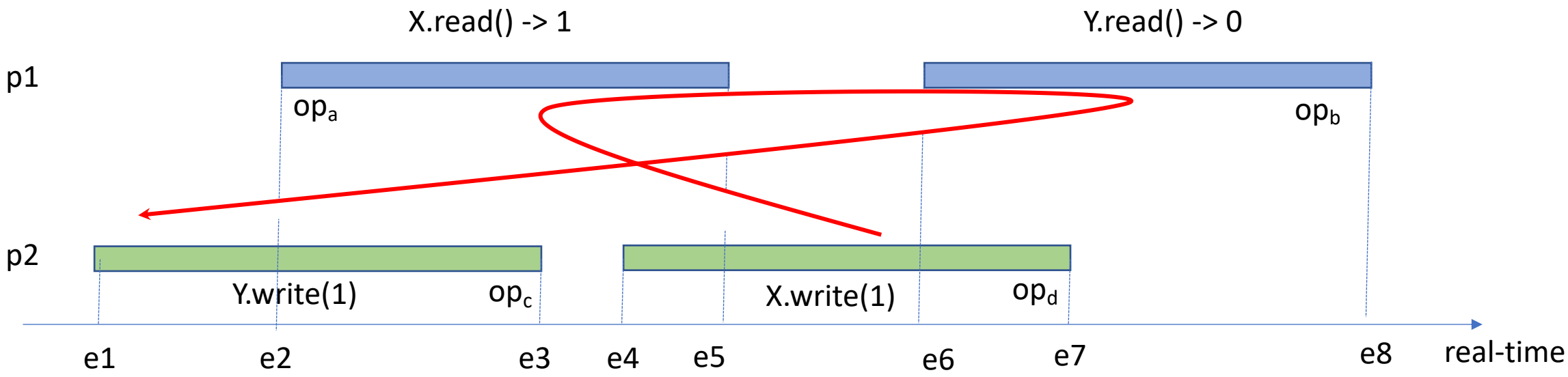
Let us consider H|Y { e1,e3,e6,e8} the projection of H on Y.
Let S2={ e6,e8,e1,e3} be the witness history of H|X

1. H|Y and S2 are equivalent
2. S2 is sequential
3. S2 is legal : it belongs to the specification of X

- S2 is sequentially consistent but not linearizable

A concurrent history H

Both registered X and Y are initialized to 0



Is H sequentially consistent ? Let S be a permutation of H

S should preserve the process order order

op_a -> op_b and op_c -> op_d

In every legal history S of H, op_d should precede op_a (specification of a register).

op_d ->$_H$ op_a

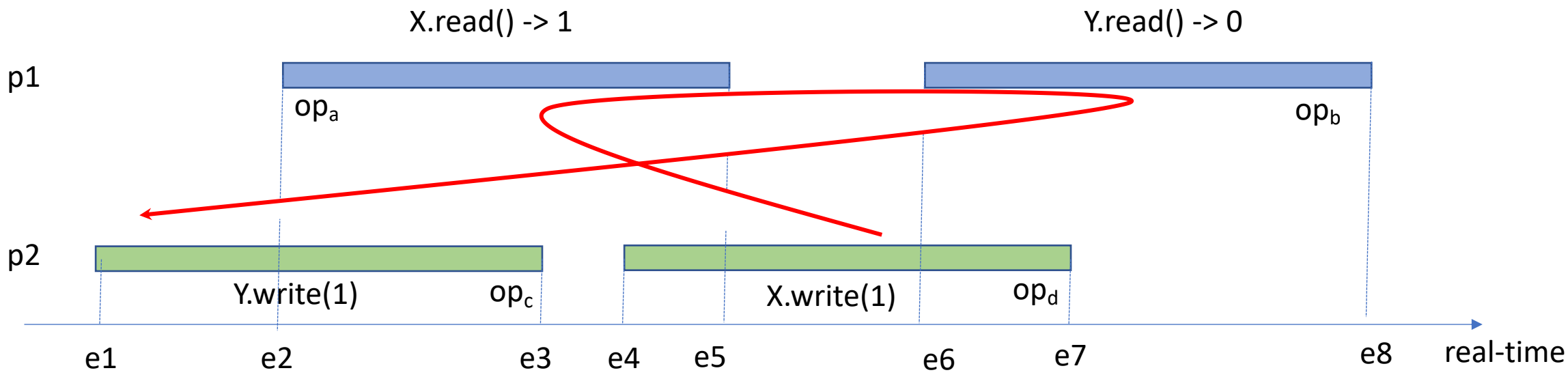
In every legal history S of H, op_b should precede op_c (specification of a register).

op_b ->$_H$ op_c

Thus op_a -> op_b ->$_H$ op_c -> op_d ->$_H$ op_a

A concurrent history H

Both registered X and Y are initialized to 0



Is H sequentially consistent ? Let S be a permutation of H

S should preserve the process order order

op_a -> op_b and op_c -> op_d

In every legal history S of H, op_d should precede op_a (specification of a register).

op_d ->$_H$ op_a

In every legal history S of H, op_b should precede op_c (specification of a register).

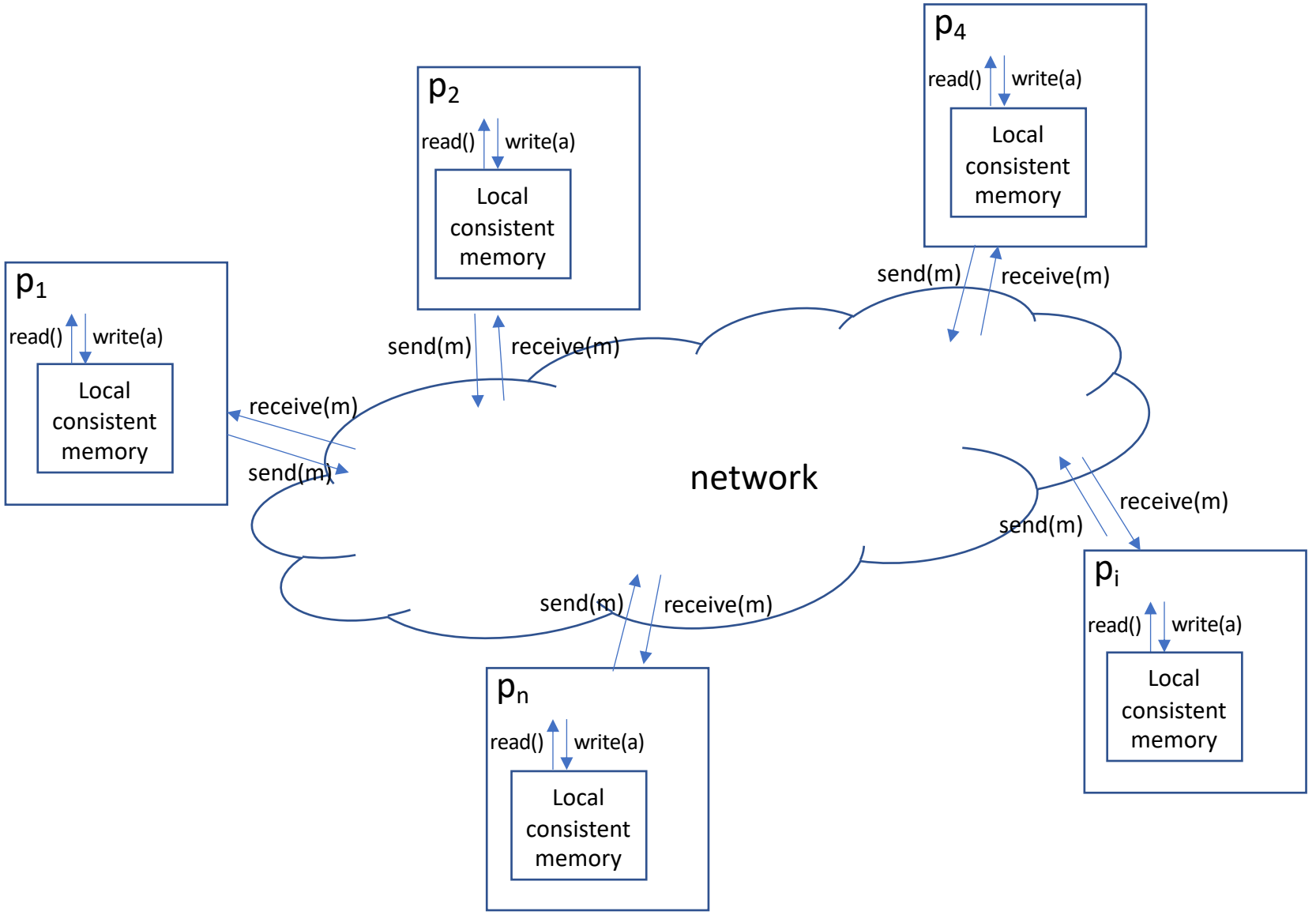
op_b ->$_H$ op_c

Thus op_a -> op_b ->$_H$ op_c -> op_d ->$_H$ op_a

Sequential consistency

- Linearizability is a strictly stronger condition than sequential consistency, i.e., every execution that is linearizable is also sequentially consistent, but the reverse is not true

- As we will see later this difference between sequential consistency and linearizability imposes a difference in the cost of implementing them.

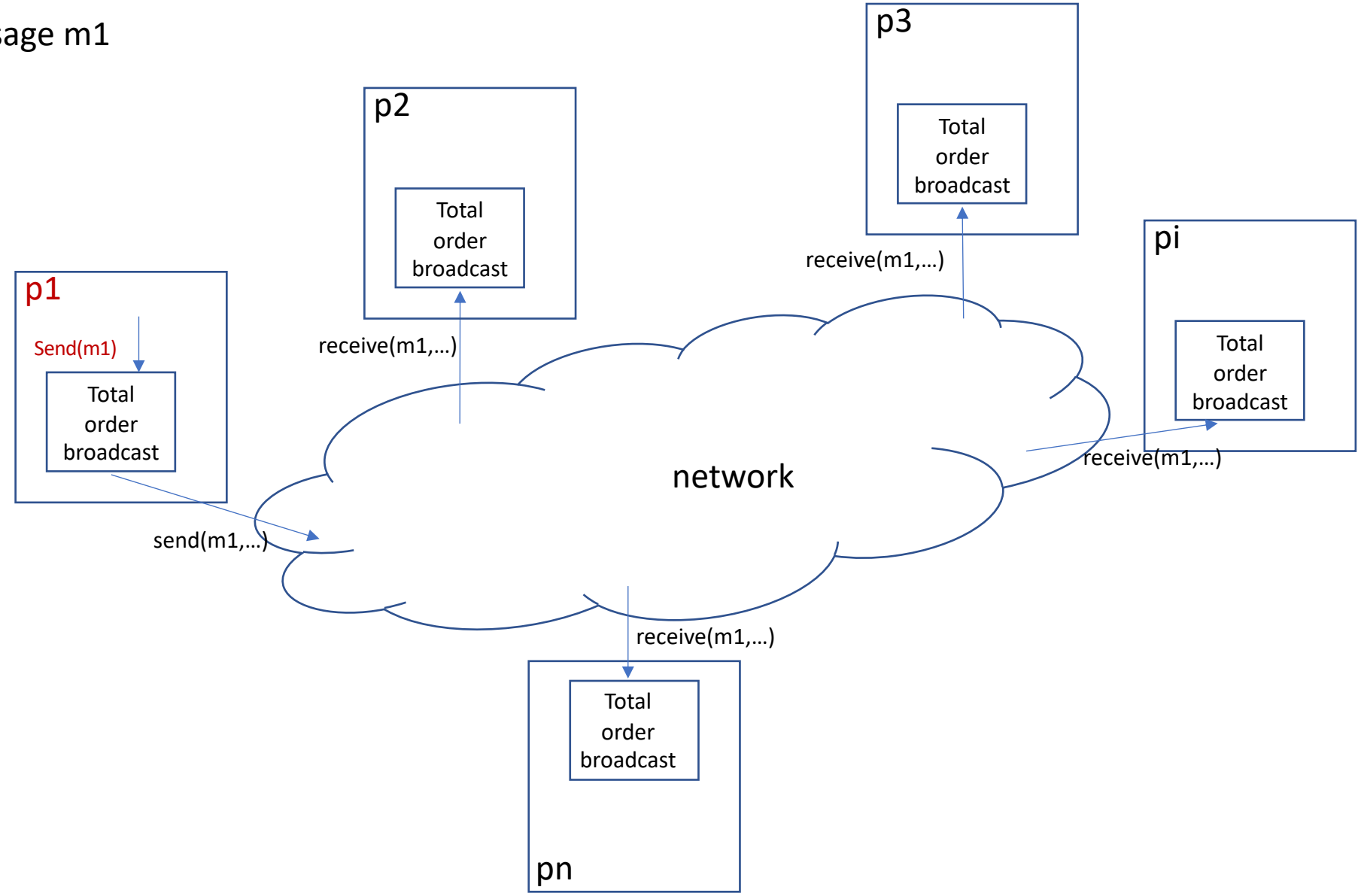

p1
read()
write(a)
Local consistent memory
p2
read()
write(a)
Local consistent memory
p4
read()
write(a)
Local consistent memory
pi
read()
write(a)
Local consistent memory
pn
read()
write(a)
Local consistent memory
network
send(m)
receive(m)
send(m)
receive(m)
send(m)
receive(m)
send(m)
receive(m)
send(m)
receive(m)

Implementing a distributed shared memory

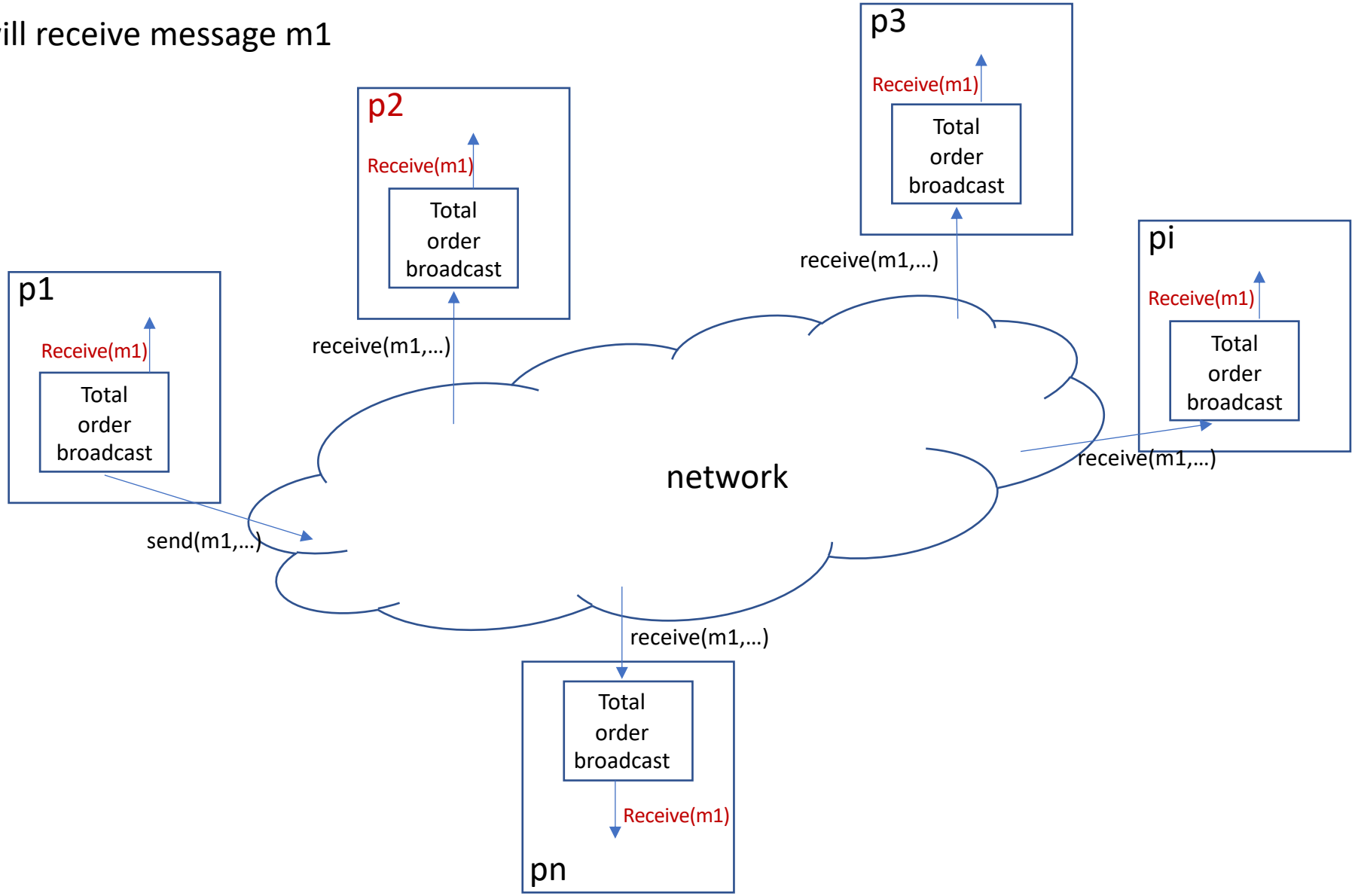
In this algorithm we suppose that all nodes have access to a totally ordered broadcast primitive

- All the nodes deliver (1) all the sent messages and (2) in the same order

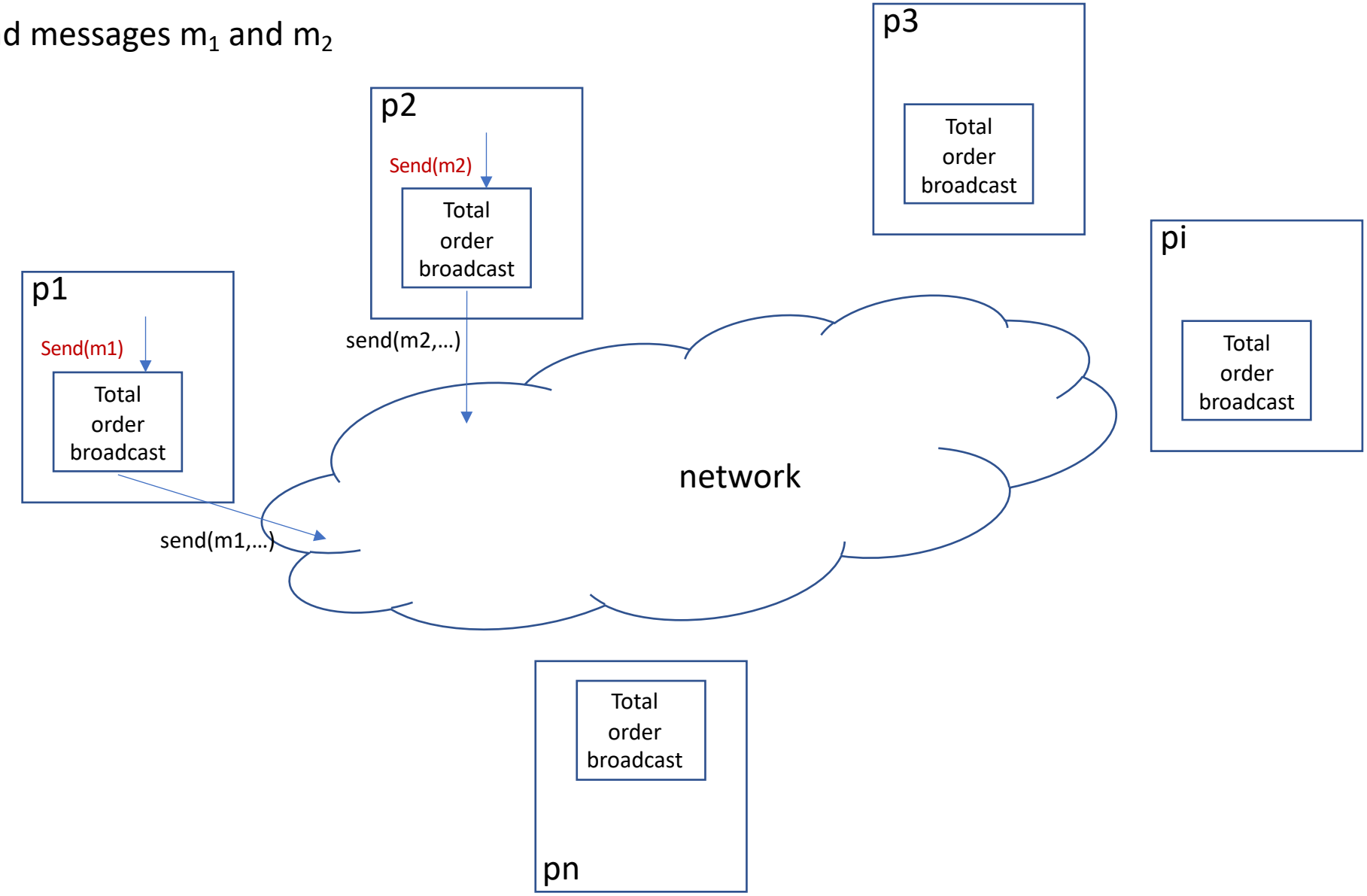
If p1 sends message m1



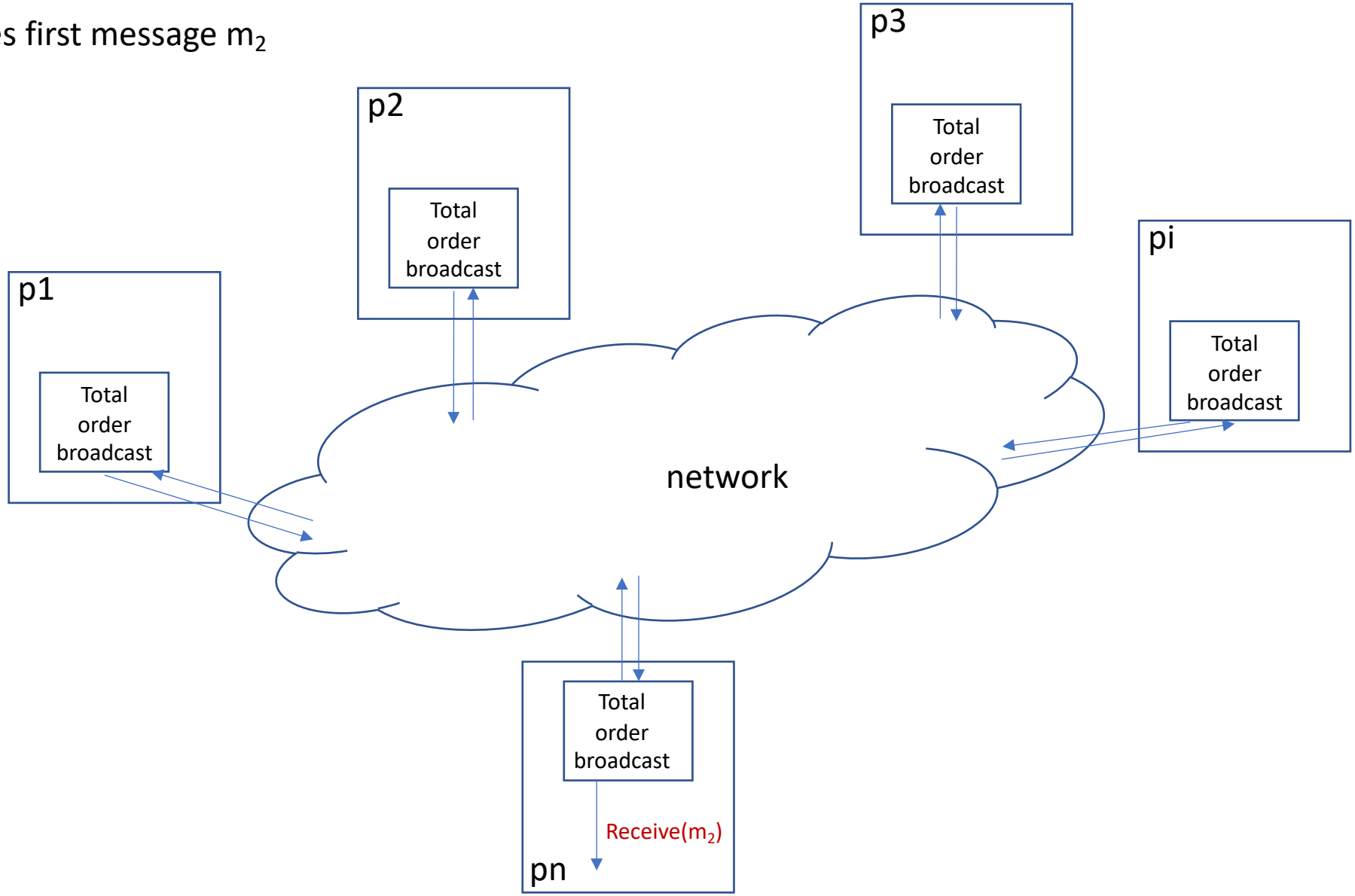
Then all nodes will receive message m1



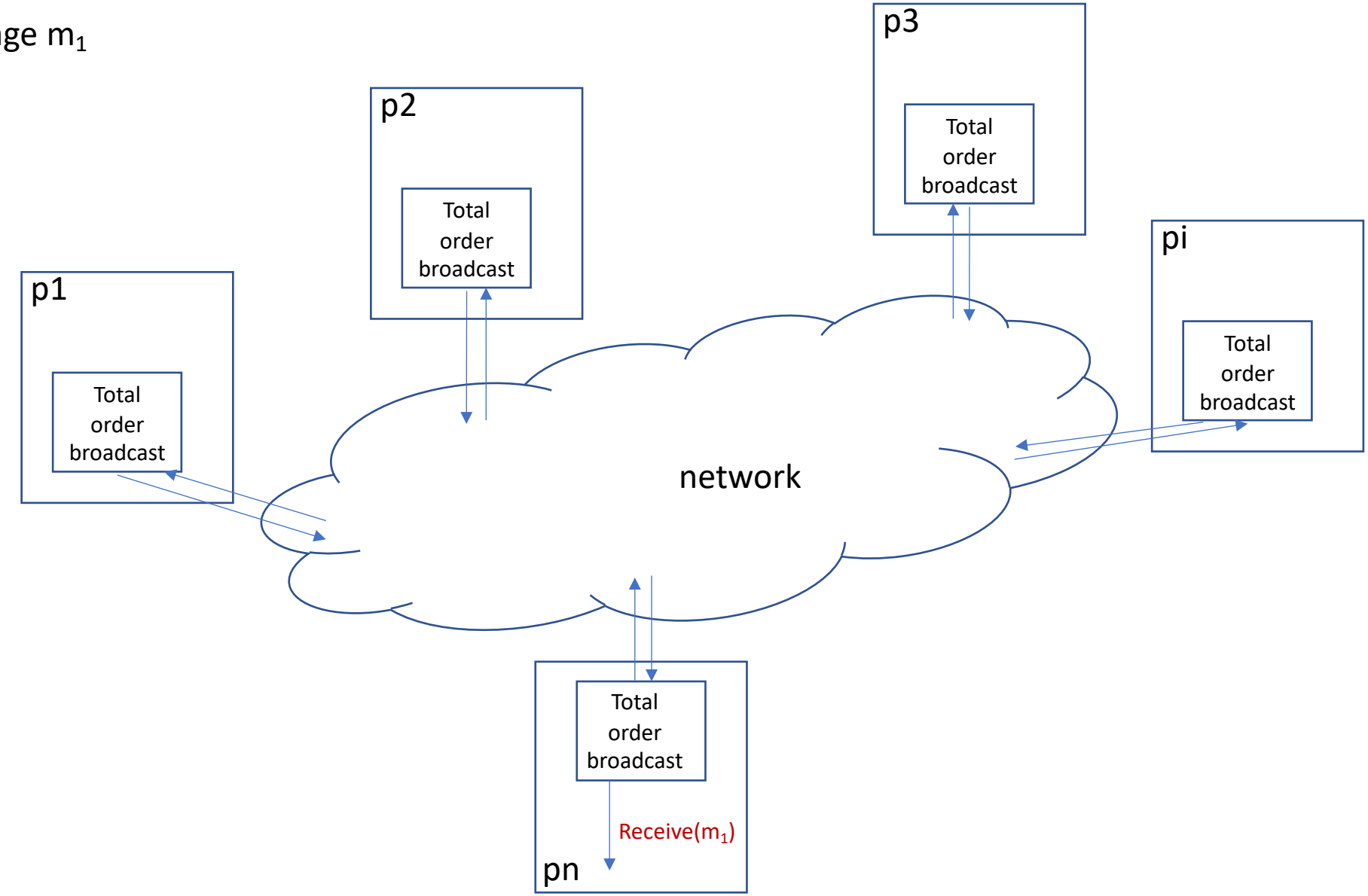
If p1 and p2 send messages m_1 and m_2



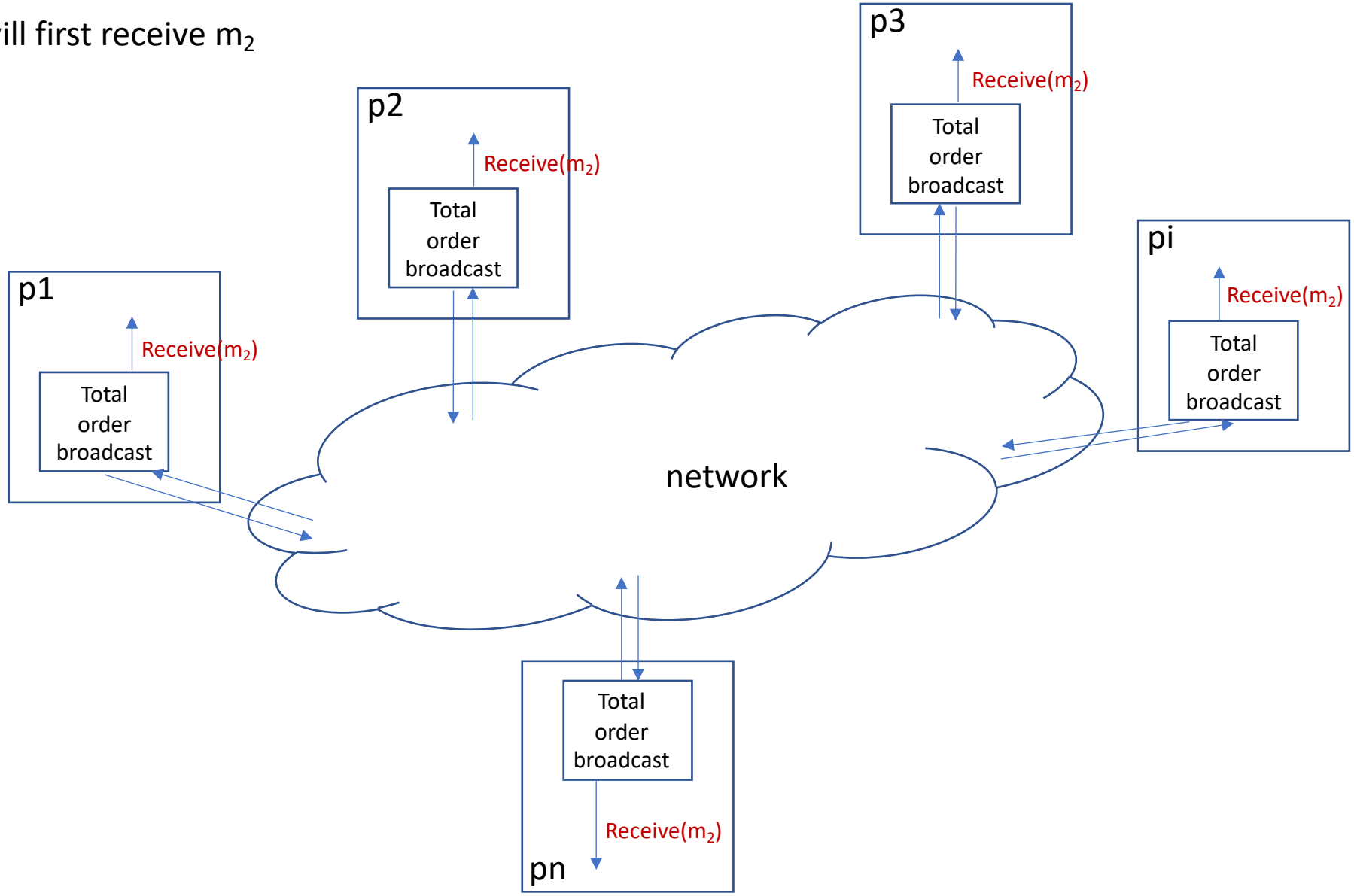
Then if p_i receives first message m_2



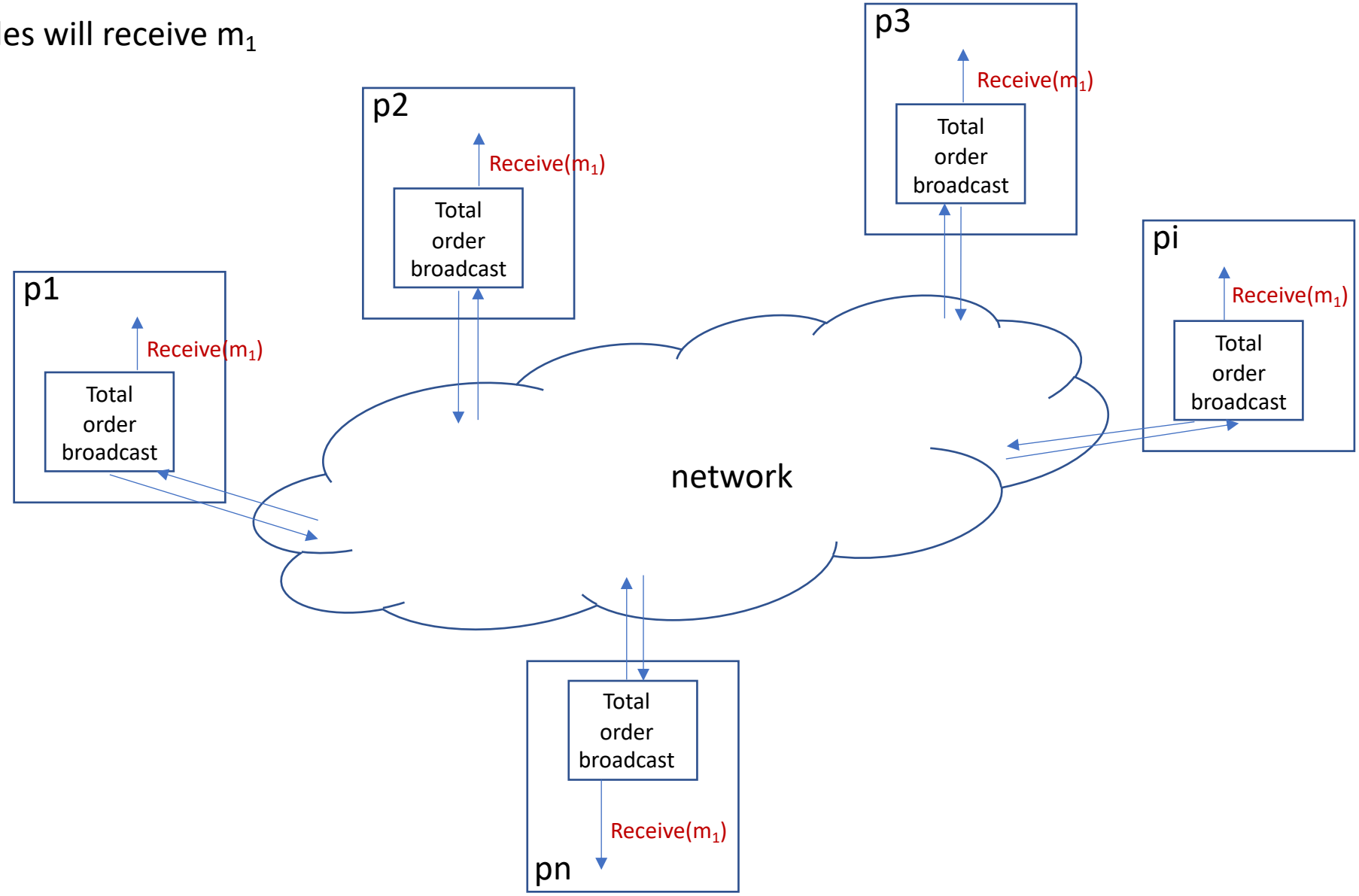
And then message m_1



Then all nodes will first receive m_2



And then all nodes will receive m_1



Implementing a distributed shared memory

- Suppose that the shared object to be implemented is a register X.
- X supports two operations: read() and write(v)
  - Invocation for a read is denoted by $read_i(x)$ and response is $return_i(x,v)$
  - Invocation for a write is denoted by $write_i(x,v)$ and response is $return_i(x,ack)$
- A sequence of operations is legal is the value returned by each read is the value written by the most recent preceding write

Implementing a distributed shared linearizable read/write object

- We now present an algorithm that provides linearizability
- Each node locally maintains its own memory and tries to maintain it consistent (in the sense of linearizability) with the one of the other nodes
- Each node communicates by reliably sending and receiving messages

Algorithm 1: Linearizable algorithm. Code executed by processor p_i

Initially:
copy[x] holds the initial value of the shared register x, for all x

1: **when** $x.read_i()$ occurs at p_i **do**
2: send $m=(i,x.read_i())$ to all processes
3: **wait until** receive m from p_i
4: **return** copy[x]

5: **when** $x.write_i(v)$ occurs at p_i **do**
6: send $m=(i,x.write_i(v))$ to all processes
7: **wait until** receive m from p_i
8: **return** ack(x)

9: **when** receive $m=(j,x.write(v))$ from p_j **do**
10: copy[x] := v

Implementing a shared linearizable read/write object

- Let us show that this algorithm simulates a linearizable shared memory
- Let H be any execution of the algorithm
- We show that top(H) is linearizable, where top(H) represents the operations of the shared objects
- Let L be a permutation of H such that all operations are ordered according to the total order of the broadcast messages.

1. We will show that L respects the semantic of the shared objects. Let X be a R/W object
2. We will show that L respects the real-time order of all the operations in H

Implementing a shared linearizable read/write object

1. We show that L respects the semantic of the shared objects.
   - Let X be a R/W object
   - L|X is the sequence of operations that access X
   - Since the broadcasts are totally ordered, every process receive messages for each operation on X in the same order, and manages its copy correctly.
   - Thus the sequential specification of X is respected
   - The same applies for all shared objects involved in the execution
2. We show that L respects the real-time order of all the operations in H
   - Suppose that in H $op1 <_H op2$
   - Then the broadcast of op1 has been received at its initiator before the broadcast of op2 by its initiator
   - Thus op2's broadcast is ordered after op1's broadcast
   - Thus $op1 <_L op2$

Implementing a shared linearizable read/write object

- This algorithm requires that every read() and write() operations to wait until the initiator receives its own broadcast message back.
- Let us try to optimize the algorithm. Indeed, since
  - No copies are changed when the broadcast message for a read() are received.
  - Why bothering to send this message ?
  - Why not just returning the value of the local copy of the object right away?

Algorithm 1: Linearizable algorithm. Code executed by processor p_i

Initially:
copy[x] holds the initial value of the shared register x, for all x

1: **when** $x.read_i()$ occurs at p_i **do**
2: ~~sends $m=(i,x.read_i())$ to all processes~~
3: ~~**wait until** receive m from p_i~~
4: **return** copy[x]

5: **when** $x.write_i(v)$ occurs at p_i **do**
6: sends $m=(i,x.write_i(v))$ to all processes
7: **wait until** receive m from p_i
8: **return** ack

9: **when** receive $m=(j,x.write(v))$ from p_j **do**
10: copy[x] := v

Algorithm 1: ~~Linearizable~~ algorithm. Code executed by processor p_i

Initially:
copy[x] holds the initial value of the shared register x, for all x

1: **when** x.read$_i$() occurs at p_i **do**
2: ~~sends m=(i,x.read$_i$()) to all processes~~
3: ~~**wait until** receive m from p_i~~
4: **return** copy[x]

5: **when** x.write$_i$(v) occurs at p_i **do**
6: sends m=(i,x.write$_i$(v)) to all processes
7: **wait until** receive m from p_i
8: **return** ack

9: **when** receive m=(j,x.write(v)) from p_j **do**
10: copy[x] := v

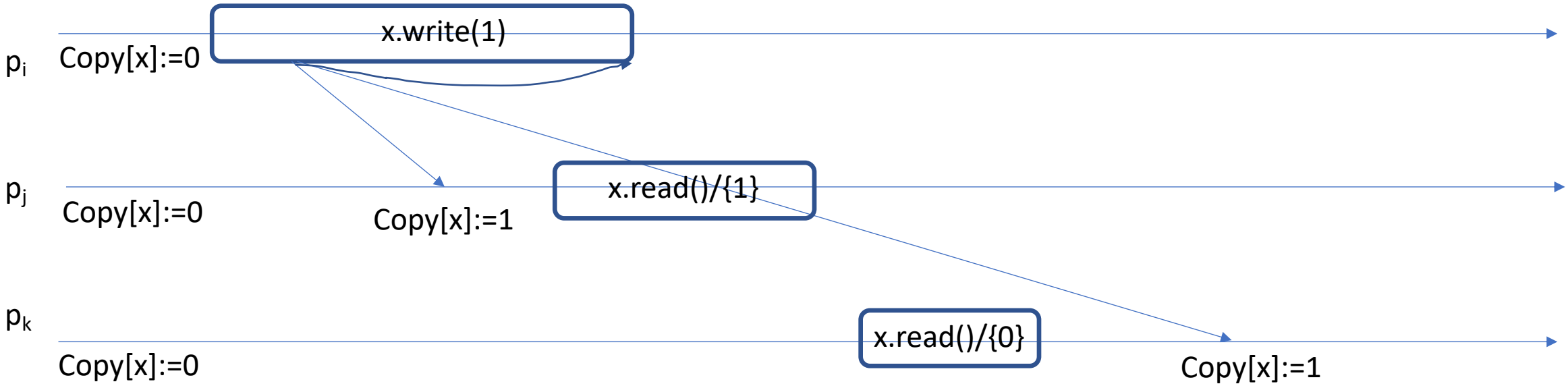
A possible execution…

- The problem is that the resulting algorithm does not guarantee linearizability.
- Consider an execution in which the initial value of X is 0
- An invocation at process p_i of a X.write(1) operation is triggered.
- Process p_i sends the operation to all the processes
- Process p_j receives this message from p_i, thus p_j updates its local copy
  $copy_j[x]:=1$
- Subsequently, p_j triggers a read() operation, and thus directly returns the new value 1

A possible execution …

- Consider now process p_k that receives p_i's message very late (at least after p_j has executed the read()/{1} operation
- Process p_k triggers a read() operation right after p_j has executed the read()/{1} operation. Since p_k has not received yet p_i's message, its read returns 0

Counter example



Any permutation L of the execution H must be such that
x.write(1) before x.read()/{1} (to respect the sequential specification of X)
and x.read()/{1} <H x.read()/{0} (to respect the real time ordering of H)

Thus no permutation of these 3 operations can both conform to the
read/write specification and respects the total order of all non-overlapping operations

We now see that this modified algorithm implements sequential consistency

Implementing a shared sequentially consistent read/write object

- As for Algorithm1, each process keeps a local copy of every object
- A read returns the local value of the copy immediately
  we say that reads are local
- When a write occurs at p_i, p_i sends the request to all processes and wait until it receives its own request to update its local copy of the object and to return ack
- When p_i receives a request for a write it updates its local value of x

Algorithm 2: Sequentially consistent local read algorithm. Code executed by processor p_i

Initially:
copy[x] holds the initial value of the shared register x, for all x

1: **when** $x.read_i()$ occurs at p_i **do**
2: **return** copy[x]

3: **when** $x.write_i(v)$ occurs at p_i **do**
4: sends $m=(i,x.write_i(v))$ to all processes

5: **when** receive $m=(j,x.write(v))$ from p_j **do**
6: copy[x] := v
7: if i=j then **return** ack(x)

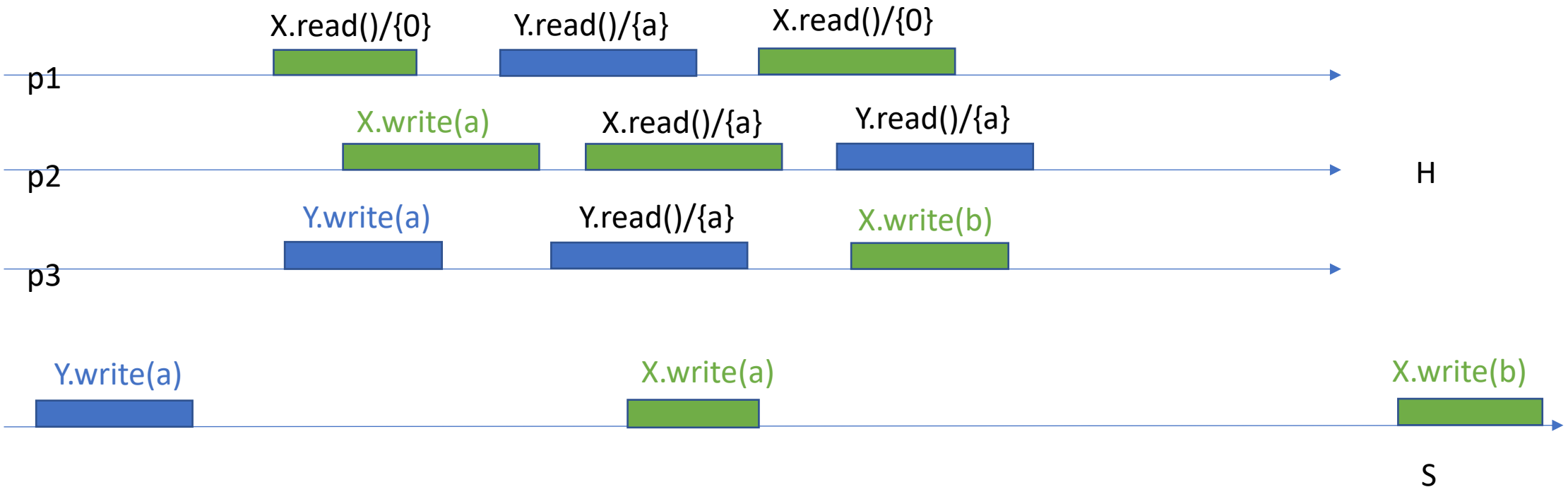
Proof that the algorithm implements a sequentially consistent shared memory

- Let us show that this algorithm simulates a sequentially consistent shared memory
- Let H be any execution of the algorithm
- We show that top(H) is sequentially consistent, where top(H) represents the operations of the shared objects
- Let S be a permutation of H

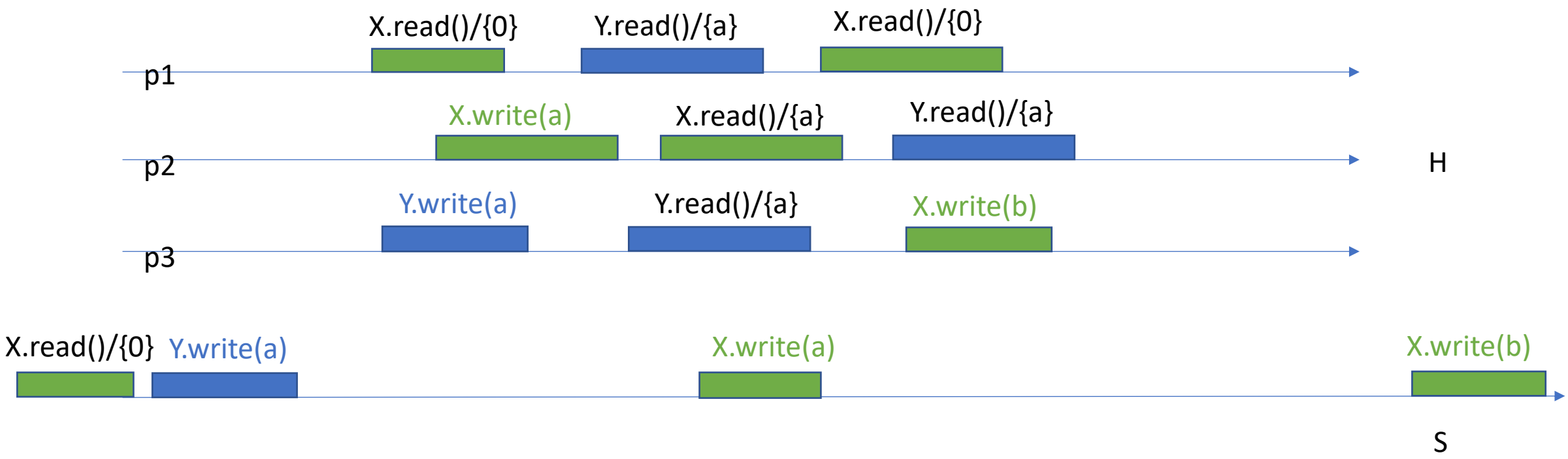
Proof that the algorithm implements a sequentially consistent shared memory

- Permutation S is constructed as follows:
- Write operations:
  - All write operations are ordered according to the total order of the broadcast messages
- Read operations:
  - We consider each read in order starting at the beginning of H
  - Read r by p_i on object X is placed immediately after the later of
    1. The previous operation in H for p_i (this operation can be a read, a write on any object), and
    2. The write w that caused the latest update of x on pi preceding the moment where r is executed

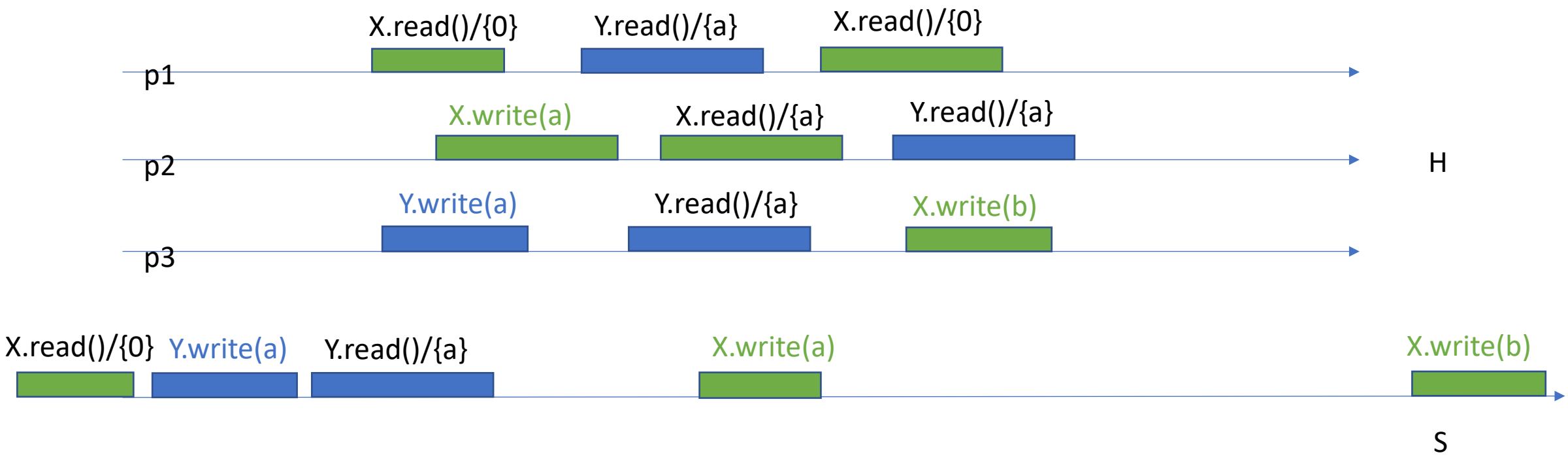
Proof that the algorithm implements a sequentially consistent shared memory



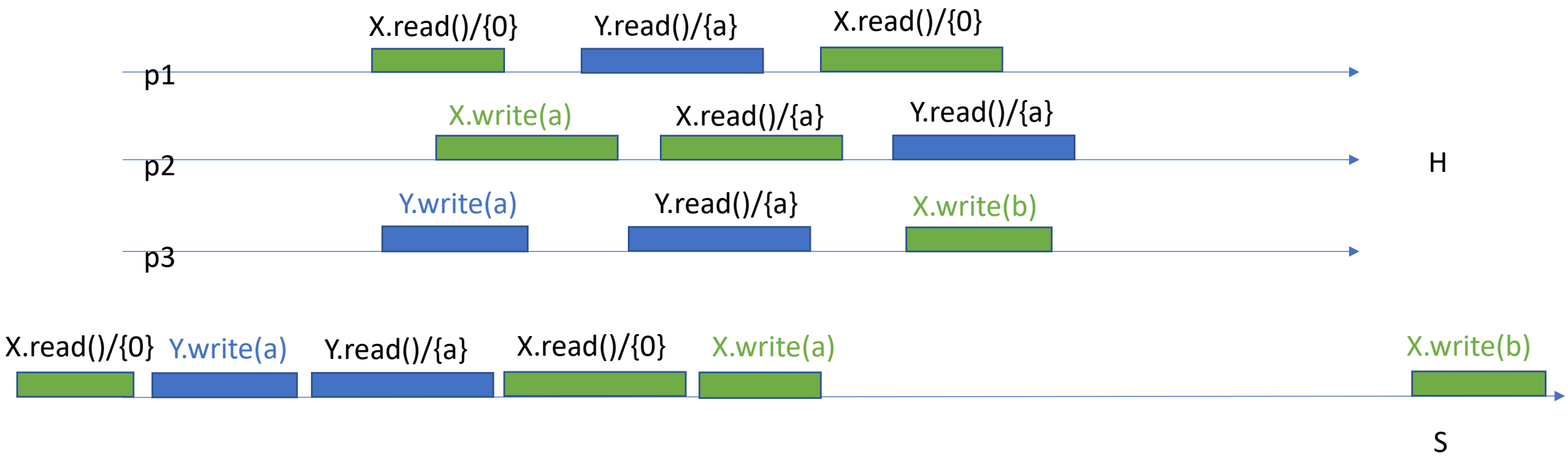
Proof that the algorithm implements a sequentially consistent shared memory



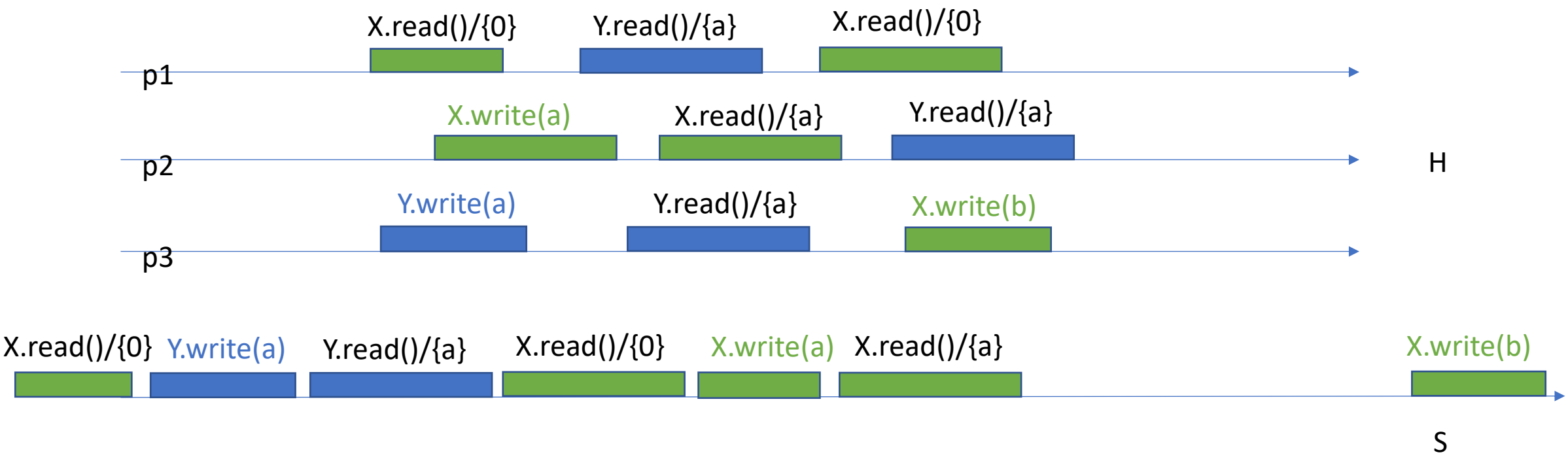
Proof that the algorithm implements a sequentially consistent shared memory



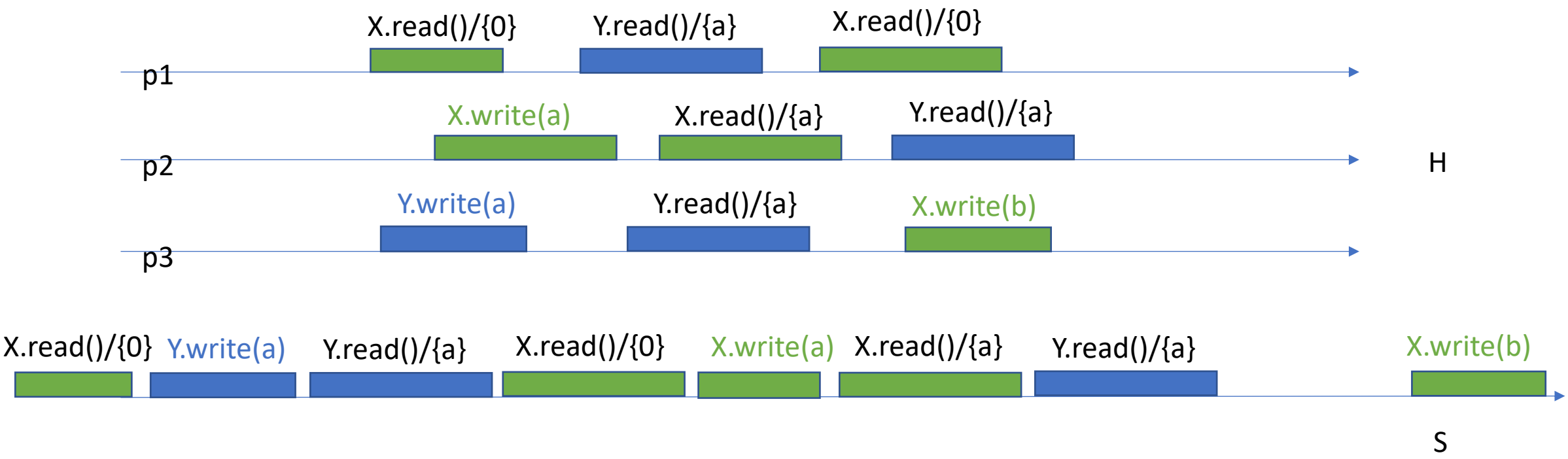
Proof that the algorithm implements a sequentially consistent shared memory



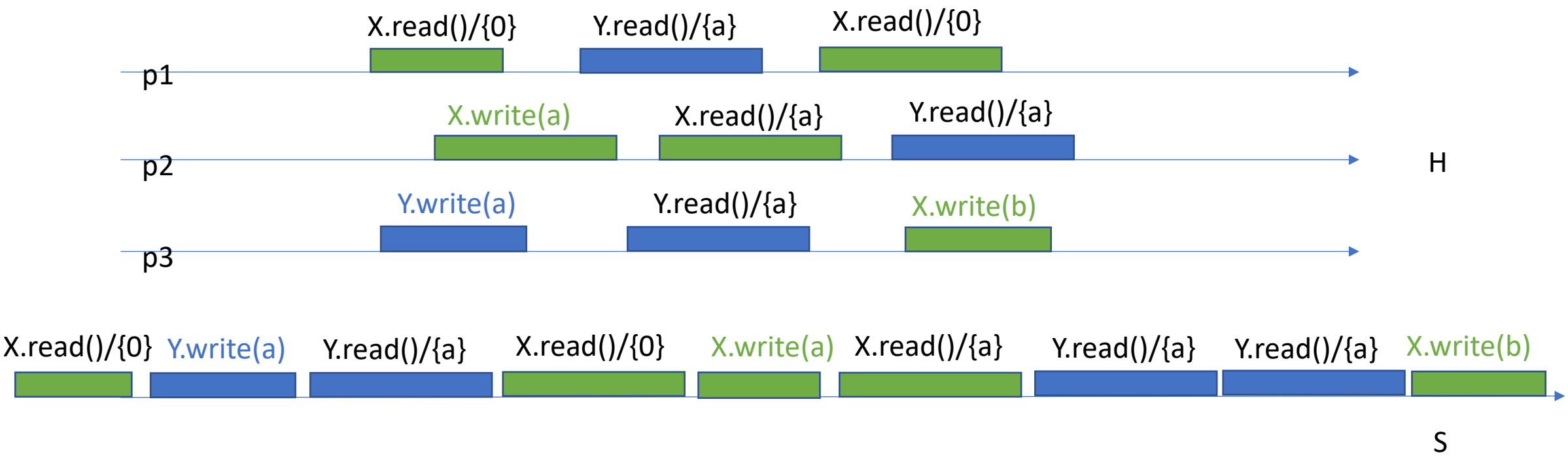
Proof that the algorithm implements a sequentially consistent shared memory



Proof that the algorithm implements a sequentially consistent shared memory



Proof that the algorithm implements a sequentially consistent shared memory



Proof that the algorithm implements a sequentially consistent shared memory

We must show that H and S are equivalent i.e. H|i = S|i for all proc i

- Fix some proc i
- The relative ordering of any two reads in H|i is the same as in S|i by definition of permutation L. The same applies for any two writes
- Suppose that w(v) $<_{H|i}$ r/{v}: By the definition of S, w(v) $<_S$ r()/{v}
- Suppose that r $<_{H|i}$ w:

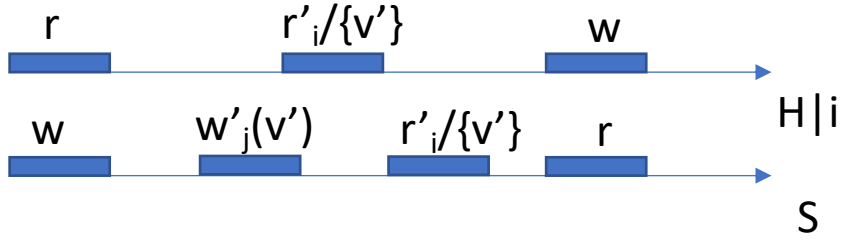
  Suppose in contradiction that w $<_S$ r

  So there must exist on pi a read r'/{v'} and a write w'(v') by pj such that

  1. (r' = r or r' $<_{H|i}$ r) and
  2. (w = w' or w < w') and
  3. w' = f(r')

  But in H|i we have r' $<_{H|i}$ w and by assumption w < w'.

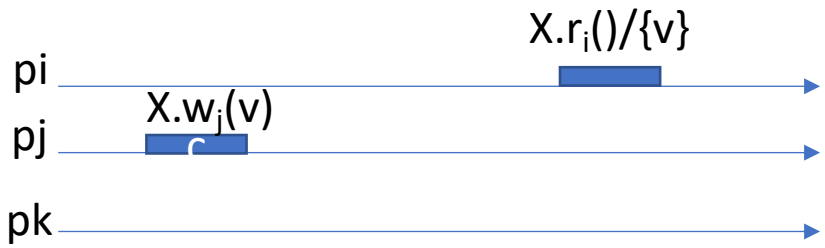
  Thus r' cannot see the update of w'. A contradiction

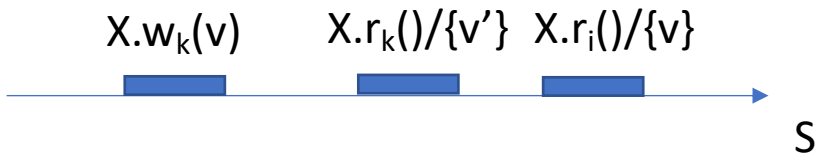


Proof that the algorithm implements a sequentially consistent shared memory

We must show that S is legal i.e. S|X respects the sequential specification of the shared object X

- Consider a read r by pi that reads v from object X in S
- Let w_j be the write that causes the latest update of x at pi that precedes r
- By definition of S r follows w on S
- We must show that there is not a write w'=X.w(v') such that w < w' < r in S
- Suppose in contradiction that w' exists

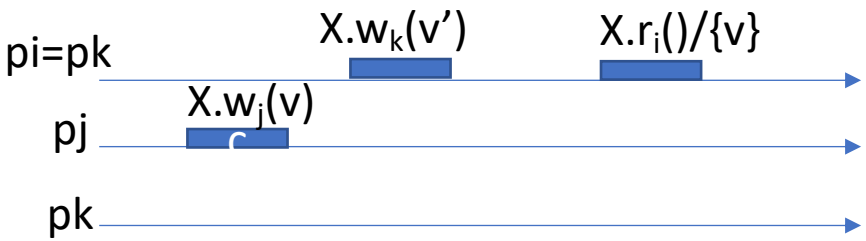


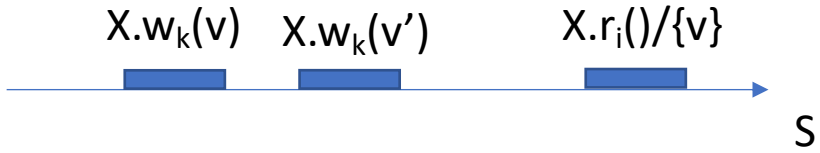


Proof that the algorithm implements a sequentially consistent shared memory

We must show that S is legal i.e. S|X respects the sequential specification of the shared object X

- Case 1: k = i
  Since S preserves the order of read and write operations at pi, w' precedes r in H
  Since w $<_H$ w', read r sees the update of w' and not the one of w.
  A contraction with the choice of w

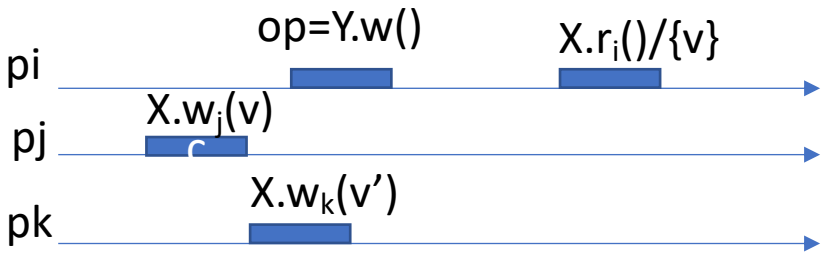


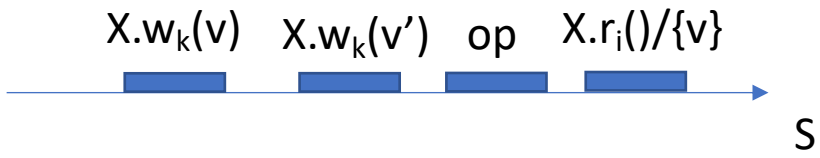


Proof that the algorithm implements a sequentially consistent shared memory

We must show that S is legal i.e. S|X respects the sequential specification of the shared object X

- Case 2: k ≠ i
  By the first condition on the construction of permutation S, there is some operation op in H|i such that in S op precedes r and follows w' (otherwise r would not follow w')
  case 2.1 Suppose that op is a write to some object Y.
  We have op $<_{H|i}$ r
  Since updates are done in the same order at all proc, the update for w' at pi occurs before op and thus before r. This contradicts the choice of w

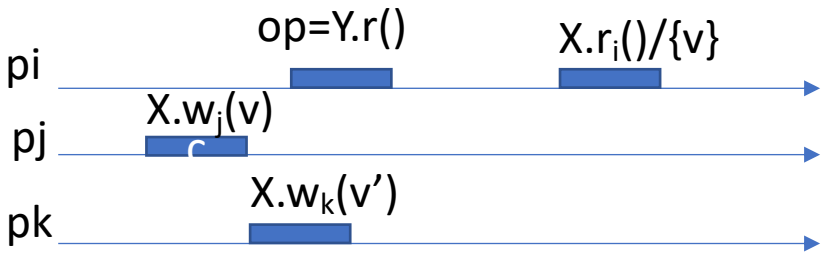


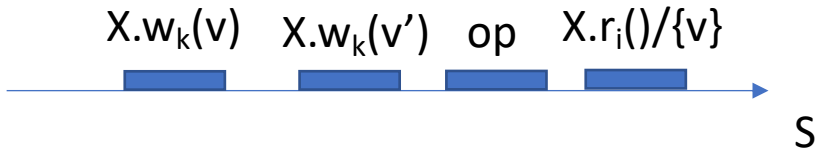


Proof that the algorithm implements a sequentially consistent shared memory

We must show that S is legal i.e. S|X respects the sequential specification of the shared object X

- Case 2: k ≠ i
  By the first condition on the construction of permutation S, there is some operation op in H|i such that in S op precedes r and follows w' (otherwise r would not follow w')
  case 2.2 Suppose that op is a read.
  By construction of S this read must read object X (otherwise op would not follow w'), and the update of X at pi due to w' is the latest one preceding op
  Since updates are done in the same total order, the value written by w' supersedes the value from w.
  Contraction with the choice of w





Bibliography

Hagit Attiya and Jennifer Welch’s book, “Distributed computing, Fundamentals, simulations, and advanced topics”, Wiley series.