

We have discussed three types of broadcast ordering: single source FIFO, totally ordered and causally ordered. In this section, we present several implementation of broadcast services.

The basic broadcast service is simple to implement on top of an asynchronous point-to-point message system with no failure. When a broadcast send occurs for message  $m$  at processor  $p_i$ ,  $p_i$  use underlying point-to-point message system to send a copy of  $m$  to all the processors. Once a processor receives the message from  $p_i$  over the underlying message system, it performs the broadcast receive event for  $m$  and  $i$ .

- Single-source FIFO ordering

Each processor assigns a sequence number to each of the message it broadcasts; the sequence number is incremented by one whenever a new message is broadcast. A recipient processes messages in order of the sequence number.

- Totally Ordered broadcast (an asymmetric algorithm)

All processors send message to a unique central processor who assigns sequence numbers and sends to all processors using the basic broadcast service. Processors perform the receives according to the sequence numbers. Thus the receives of the totally ordered broadcast service happen in the same order at all processors. The role of central processor can rotate via a token to spread the communication overhead.

- Total ordered (a symmetric algorithm)

This algorithm is based on assigning timestamps to messages. Also it assumes that the underlying communication system provides single-source FIFO broadcast.

Each processor maintains an timestamp. When a processor is supposed to broadcast a message, it tags the message with the current value of its timestamp before sending it out. Each processor also maintains a vector with estimate of the timestamps of all other processors. The  $p_i$ 's entry for processor  $p_j$  means  $p_i$  will never again receive a message from  $p_j$  with timestamp smaller than or equal to that value. Each processor maintains its own timestamp to be greater than or equal to its estimates of the timestamp of all the other processors.

## 14 Distributed shared memory

Distributed shared memory (DSM) is a model for interprocess communication that provides the illusion of a shared memory on top of a message passing system. In this model, processes running on separate

nodes can access a shared memory address space, provided by underlying DSM system, through familiar read and write operations.

A DSM is a simulation of an synchronous shared memory model by the asynchronous message passing model. We call the simulation program, which runs on top of the message system providing the illusion of shared memory, the *memory consistency system* (MCS). In Figure 48, the MCS consists of local MCS processes at each processor  $p_i$ , which use local memory and communication with each other using a message passing communication system

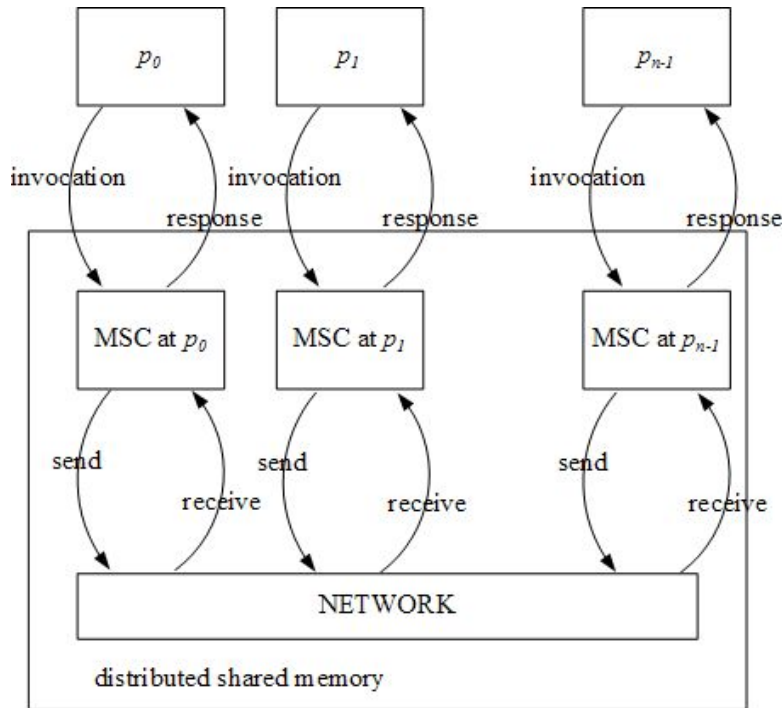


Figure 48: A memory consistency system

### 14.1 Linearizable shared memory

Every shared object is assumed to have a sequential specification, which indicates the desired behavior of the object in the absence of concurrency. The object supports *operations*, which are pairs of invocations and matching responses. A *sequential specification* consists of a set of operations, and a set of sequences of operations.

Let  $i$  be a processor and  $X$  be a shared object. Then we have operation:

operation	invocation	response
$\text{READ}_i(X, v)$	$\text{read}_i(X)$	$\text{return}_i(X, v)$
$\text{WRITE}_i(X, v)$	$\text{write}_i(X, v)$	$\text{ack}_i(X)$

A sequence of operation is *legal* if each READ returns the value of the most recent preceding WRITE. If there is no preceding WRITE, then returns the initial value.

For example, suppose  $\text{init} := 0$ . The sequence

$$\sigma = \text{READ}_i(X, 0) \text{WRITE}_j(X, 1) \text{WRITE}_i(X, 2) \text{READ}_j(X, 2)$$

is a legal operation sequence for object  $X$ .

The sequential specification of an object is the set of legal operation sequences for the object.

Now we can specify a linearizable shared memory communication system. Its inputs are invocations on the shared objects and its output are responses from the shared objects. In order for a sequence  $\sigma$  to be in the allowable set, the following properties must be satisfied:

1. Correct interaction

For each  $p_i$ ,  $\sigma|_i$  consists of alternating invocations and matching responses beginning with an invocation. This condition imposes constraints on the inputs. A processor has to wait for a response from one object before submitting an invocation on another object. It prohibits processors from pipelining operations on shared objects.

2. Liveness

Every invocation has a matching response.

3. Linearizability

There exists a permutation  $\pi$  of all the operations in  $\sigma$  such that 1) for each  $X$ ,  $\pi|_X$  is legal; and 2) if response of operation  $o_1$  occurs in  $\sigma$  before invocation of operation  $o_2$ , then  $o_1$  appears before  $o_2$  in  $\pi$ .

A sequence is linearizable if there is a way to reorder the operations in the sequence that 1) respects their sequential specification, and 2) respects the real time ordering of events among all the nodes.

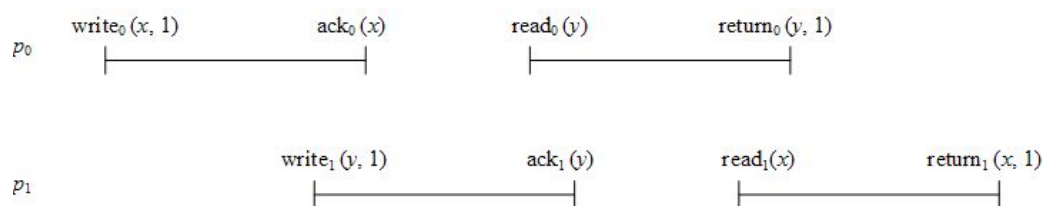


Figure 49: A linearizable execution  $\sigma_1$

For example, suppose we have two shared registers  $x$  and  $y$ , both initially 0, the sequence

$$\begin{aligned} \sigma_1 = & \text{write}_0(x, 1) \text{write}_1(y, 1) \text{ack}_0(x) \text{ack}_1(y) \\ & \text{read}_0(y) \text{read}_1(x) \text{return}_0(y, 1) \text{return}_1(x, 1) \end{aligned}$$

is linearizable (See Figure 49), since  $\pi_1 = w_0w_1r_0r_1$  is the desired permutation, where  $w_i$  indicates the complete write operation by  $p_i$  and  $r_i$  the complete read operation by  $p_i$ .

Suppose that  $r_0$  returns 0 instead of 1, the resulting sequence

$$\begin{aligned} \sigma_2 = & \text{write}_0(x, 1) \text{write}_1(y, 1) \text{ack}_0(x) \text{ack}_1(y) \\ & \text{read}_0(y) \text{read}_1(x) \text{return}_0(y, 0) \text{return}_1(x, 1) \end{aligned}$$

is not linearizable. In order to respect the semantics of  $y$ ,  $r_0$  must precede  $w_1$ . But this would violate the real time ordering, since  $w_1$  precedes  $r_0$  in  $\sigma_2$ .

## 14.2 Sequentially consistent shared memory

In many situations the relative order of events at different nodes is irrelevant. The consistency condition called sequential consistency exploits this idea. Formally, a sequence  $\sigma$  of invocations and responses (satisfying the same correct interaction and liveness properties as for linearizable shared memory) is sequentially consistent if there exists a permutation  $\pi$  of the operations in  $\sigma$  such that

1) for every object  $X$ ,  $\pi|X$  is legal, according to the sequential specification of  $X$ ; and 2) if the response for operation  $o_1$  at node  $p_i$  occurs in  $\sigma$  before the invocation for operation  $o_2$  at node  $p_i$ , then  $o_1$  appears before  $o_2$  in  $\pi$ .

The first condition, requiring the permutation to be legal, is the same as for linearizability. The second condition has been weakened; instead of having to preserving the order of all non-overlapping operations at all nodes, it is only required for operations at the same node. The second condition is equivalently written  $\forall i : \sigma|i = \pi|i$ .

The sequence  $\sigma_2$  is sequentially consistent;  $\pi = w_0r_0w_1r_1$  is the desired permutation. The read by  $p_0$  has been moved before the write by  $p_1$ .

Linearizability is a strictly stronger condition than sequential consistency, i.e., every sequence that is linearizable is also sequentially consistent, but the reverse is not true (for example  $\sigma_2$ ). This difference between sequential consistency and linearizability imposes a difference in the cost of implementing them. Weak condition is more efficient implemented by the underlined architecture while strong condition is better from programmer's perspective.

There exist sequences that are not sequentially consistent. For example, suppose  $r_1$  also returns 0 instead of 1, the sequence

$$\begin{aligned} \sigma_3 = & \text{write}_0(x, 1) \text{write}_1(y, 1) \text{ack}_0(x) \text{ack}_1(y) \\ & \text{read}_0(y) \text{read}_1(x) \text{return}_0(y, 0) \text{return}_1(x, 0) \end{aligned}$$

is not sequentially consistent. In order to respect the order the semantics of  $x$  and  $y$ ,  $r_0$  must precede  $w_1$  and  $r_1$  must precede  $w_0$ . In order to respect the order of events at  $p_1$ ,  $w_1$  must precede  $r_1$ . Yet by transitivity, these constraints would require  $r_0$  to precede  $w_0$ , which violates the order of events at  $p_0$ .

### 14.3 Algorithms for linearizability and sequential consistency

We use notion  $\text{tbc-send}_i(m)$  and  $\text{tbc-recv}_i(m)$  to mean broadcast send and receive which are totally ordered. Algorithm 19 satisfies both linearizability and sequential consistency. We prove that it satisfies linearizability, which implies sequential consistency as well.

---

**Algorithm 19** Algorithm satisfying Linearizability

---

*Code for processor  $p_i$ :*  
 $\text{copy}_i[X]$  contains the initial value of shared object  $X$   
when  $\text{read}_i(X)$  occurs  
     $\text{tbc-send}_i(\text{read}, X)$   
  
when  $\text{write}_i(X, v)$  occurs  
     $\text{tbc-send}_i(\text{write}, X, v)$   
  
when  $\text{tbc-recv}_i(\text{read}, X)$  from  $p_j$  occurs  
    if  $j = i$  then  $\text{return}_i(X, \text{copy}[x])$   
  
when  $\text{tbc-recv}_i(\text{write}, X, v)$  from  $p_j$  occurs  
     $\text{copy}[X] := 0$   
    if  $j = i$  then  $\text{ack}_i(X)$

---

**Theorem 14.1.** *Linearizable shared memory is simulated by the totally ordered broadcast system.*

*Proof.* Let  $\alpha$  be an admissible execution of Algorithm 19. Let  $\sigma$  be the sequence of invocations and responses to the MCS in  $\alpha$ . We must show that  $\sigma$  is linearizable.

Order the operation in  $\sigma$  according to the total order provided in  $\alpha$  for their broadcast, to create the desired permutation  $\pi$ .

We now check that  $\pi$  respects the semantics of the objects. Let  $x$  be a read/write object.  $\pi|x$  is the sequence of operations that access  $x$ . Since the broadcasts are totally ordered, every MCS process receives the message for the operations on  $x$  in the same order, which is the order of  $\pi$ , and messages its copy of  $x$  correctly.

We now check that  $\pi$  respects the real time ordering of operations in  $\alpha$ . Suppose operation  $o_1$  finishes in  $\alpha$  before operation  $o_2$  begins. Then  $o_1$ 's broadcast has been received at its initiator before  $o_2$ 's broadcast is sent by its initiator. Obviously  $o_2$ 's broadcast is ordered after  $o_1$ 's. Thus  $o_1$  appears in  $\pi$  before  $o_2$ .

□

Since the totally ordered broadcast system is simulated by the point-to-point message passing system, we have corollary:

**Theorem 14.2.** *The linearizable shared memory system is simulated by the point-to-point message passing system.*

The algorithm is still possible to be optimized. Notice that the broadcast message for a read does not cause any changes in local copies. Can we optimize by not sending this message? This would

- improve message complexity since no broadcast would be sent for a read invocation.
- improve time complexity since a read could return immediately with the value of the local copy.

The problem is that the resulting algorithm does not guarantee linearizability. Consider an execution in which the initial value of  $x$  is 0. Processor  $p_i$  has an invocation to write 1 to  $x$  and performs the tbc-send for the write. The broadcast message arrives at processor  $p_j$ , which updates its local copy of  $x$  to be 1, subsequently does a read on  $x$ , and returns the new value 1. Consider a third processor  $p_k$ , which has not yet received  $p_i$ 's broadcast message and still has its local copy of  $x$  equal to 0. Suppose that after  $p_j$ 's read but before receiving  $p_i$ 's broadcast message,  $p_k$  does a read on  $x$ , returning the old value 0. (See Figure 50) This is not linearizable, since, to satisfy the real-time order of non-overlapping operations,  $p_j$ 's read of 1 should precede  $p_k$ 's read of 0 in  $\pi$ , but this violates legality of object  $x$ .

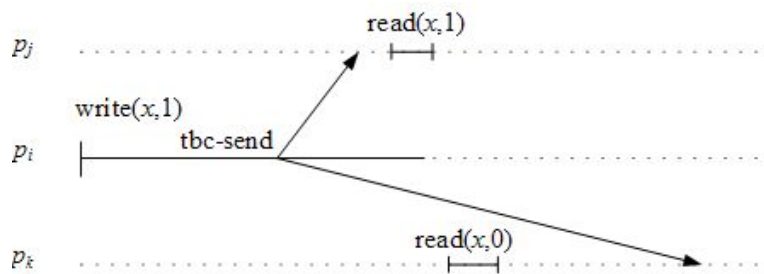


Figure 50: A non-linearizable execution