Course notes for Distributed Systems
Nitin Vaidya
March 14, 2016

Chapter 1

Distributed Shared Memory

1.1 Chapter Overview

In this chapter, we consider a distributed shared memory abstraction implemented using message-passing. That is, processes communicate via messages to implement an abstraction of shared memory. Distributed shared memory may be implemented in many different ways. Consider the two approaches below. In both approaches, multiple copies of each shared memory variable will be maintained.

- Approach 1: In the first approach, the replicas are stored in locations that are distinct from users that access the replicas. This is a common approach realized today using replicas stored at geographically distributed data centers. For instance, users of a social network may upload pictures, which can then be viewed by other users. These pictures are stored in multiple replicas, which provides two benefits. First, the latency of access can be reduced by allowing each user to access the replica close to them. Secondly, maintaining multiple replicas improves fault-tolerance. Even if one of the replicas becomes unavailable for some reason, the data is not entirely lost if another replica remains available. In this approach, the users themselves do not need to maintain a copy of the shared data locally. Any shared data that the users wants to view (Read) can be accessed from one of the replicas. Similarly, any updates to the data that the users may want to perform are achieved by modifying the replicas.

- Approach 2: In the second approach, we do not distinguish between the replicas and the users. *All* the users maintain a copy (or replica) of the shared data. Thus, when a user updates a shared data, the updates may be propagated to all the users.

On the one hand, the above two approaches are quite different. The number of replicas in Approach 1 may be quite small, the number of users may be very large. Maintaining a large number of replicas, as would be required in Approach 2, can be very expensive. On the other hand, for *defining* the notion of *consistency* of shared data, we do not necessarily have to distinguish between the two approaches. Secondly, in Approach 1, if we restrict each user to access all the shared data from a specific replica (say, the closest replica), then effectively the user population is partitioned based on the replicas that they *directly* access. Thus, each replica can be viewed as a "mega-user" that represents the collection of users that are associated with that replica. With this viewpoint then, Approach 1 becomes analogous to Approach 2.

For convenience of presentation, we will hereafter assume Approach 2, with the understanding that the discussion is also relevant for Approach 1. As noted above, maintaining multiple replicas has the potential benefit of improving latency of access, and also the possibility of improving availability of the shared data despite the failure of some the replicas (e.g., replica crash). In this chapter, however, we will *not* consider failure of the replicas.

1.2 Some Algorithms for Implementing Shared Memory

Let us consider n processes $p_1,\ p_2, \cdots,\ p_n$ that implement the distributed shared memory. Each process maintains a local copy of each shared variable. In this section, we first present three different algorithms for implementing distributed shared memory. Each algorithm specifies the actions taken when a process writes to a shared variable, and the actions taken when a process reads a shared variable.

To illustrate the behavior of the different algorithms, we consider the sequence of shared memory operations performed by three processes, p_1, p_2 and p_3 as shown in Figure 1.1.

p_1	p_2	p_3

`Write(X,2)`
`Read(X)`
`Read(X)`
	`Read(X)`
	`Read(X)`
		`Write(X,5)`

Figure 1.1: Example programs

Algorithm 1

In Algorithm 1, on a Read operation, each process simply reads its local copy of the desired shared memory location, and on a Write, performs an update locally, and sends the update messages to other processes, so that they can update their own local replicas accordingly.

When operation `Write(X,v)` is invoked at process p_i:

(step w1) Write v in the replica of variable X at p_i.

(step w2) Send message $Write(X, v)$ to each of the remaining processes (i.e., processes other than p_i).

(step w3) Return *Ack()* indicating completion of the `Write(X,v)` operation

When process p_i receives a message $Write(Y, w)$:

(step w4) process p_i writes value w in its local copy of shared variable Y.

When process p_i invokes operation `Read(X)`:

(step r1) read local copy of variable X at process p_i; suppose that the value read is v.

(step r2) return *Ack(X,v)* to p_i as the response to `Read(X)`, which also indicates the completion of the `Read(X)` operation.

The algorithm above is *event-driven*, with the pseudo-code specifying what happens when three events take place: (i) when a `Write` operation is invoked by some process, (ii) when some process receives a $Write$ message from another process, and (iii) when a `Read` operation is invoked by some process. For `Write` and `Read` both, the process that invokes the operation receives a response when the operation has been completed. For a `Write`, the process receives an $Ack()$ indicating the completion of the operation, whereas for a `Read`, the process receives $Ack(X, v)$ indicating that the Read operation on X has returned value v.

Figures are included in a separate file provided with this handout.

Consider Figure 1. In Figure 1, the timeline at the top is for process p_1, followed by processes p_2 and p_3. Also, white circles are used to depict the invocation of a Read or Write operation, and for the response (Ack) for such operations. The dark circles are used to indicate events such as local computation (including write to a local copy of a variable) and message send/receive events. In Figure 1, when process p_1 invokes the `Write(X,2)` operation, it writes 2 in its local copy of variable `X` (named X_1 in the figure), and then sends message $Write(X, 2)$ to the other processes, p_2 and p_3, as per steps w1 and w2 in the above algorithm. After sending these messages, the Write operation is complete, and the completion is signalled to process p_1 by an *Ack*() message, as per step w3. When processes p_2 and p_3 receive the $Write(X, 2)$ message from p_1, they write 2 in their local copy of variable `X`, as per step w4 above. Observe that the `Read` and `Write` operations both require a non-zero amount of time (this is the duration of time between invocation of an operation and its response). The delay in completing these operations may potentially be quite large if the DSM layer at p_i encounters a large scheduling delay while performing the steps described in the algorithm. In asynchronous systems, it is assumed that there is no finite bound on the time required to complete each operation.

For the `Read` operation, no inter-process communication is required in Algorithm 1. The value of the local copy of the desired shared variable is returned in response to a `Read` operation. For example, the `Read(Z)` operation at process p_2 will return value 2, which was previously written due to the `Write(Z,2)` operation invoked by process p_1.

Observe that Algorithm 1 has a peculiar behavior when multiple write operations are invoked by different processes. In Figure 1, process p_1 invokes `Write(X,5)` operation some time after process p_1 invokes `Write(X,2)`. As per step w1, process p_1 writes 2 to its local copy of variable X, and later writes 5 when it receives the $Write(X, 5)$ message from p_3. On the other hand, process p_3 writes 5 to its local copy of variable X, and later writes 2 to X when it receives message $Write(X, 2)$ from process p_1. Thus, the replicas at processes p_1 and p_3 have different values after the execution shown in the figure. Due to the different order in which the updates are performed at p_1 and p_2, while p_1's first `Read(X)` returns value 2, and its second `Read(X)` returns value 5, p_2's first `Read(X)` returns value 5, and its second `Read(X)` returns 2. Due to this phenomenon, p_1 and p_2 have an inconsistent view of the updates performed on shared variable X. In our example, there are only two Write operations performed. When there are more Write operations, it should be easy to see that each replica may go through a sequence of states that is different from the other replicas, and all the different replicas may never reach an identical state.

Last-writer-wins: One approach to ensure that all the replicas *eventually* become identical is to use a *timestamp* mechanism to determine which values are older. Suppose that each replica of variable X not only stores its value, but also the time at which the `Write` that resulted in that value was invoked. To implement this, the *Write* messages carry the time at which the corresponding `Write` operation is invoked. Suppose that the time at which `Write(X,2)` is invoked at p_1 is $t1$ (where $t1 > 0$), and the time at which `Write(X,5)` is invoked at p_3 is $t3$, where $t1 < t3$. The initial timestamp of all variables is 0. When p_1 invokes the Write, it updates its local copy of X because the local copy's timestamp 0 is smaller that $t1$. The timestamp is updated to $t1$ when value 2 is written to X. When p_1 receives the $Write$ message from $p3$, the timestamp of the local copy is $t1$, which is smaller than the timestamp received with the $Write$ message from p_3; hence the local copy of p_1 is updated to value 5 and timestamp $t2$. On the other hand, when $p2$ and $p3$ receive the $Write$ message from p_1, their local timestamps for X are already $t2$, which is greater than timestamp $t1$

received with the $Write$ message from p_1. Hence p_2 and p_3 do not write value 2 to their local copy of X on receiving the message from p_1 (i.e., p_2 and p_3 simply ignore the message from p_1). Thus, when all the replicas have received both the $Write$ messages, the final value of all the local copies of X will be identical (i.e., the values of local copies of X *eventually* become identical). The above approach for using timestamps is called the *last-writer-wins* rule.

Let us name the above modified Algorithm 1 as Algorithm 1T (or *Algorithm 1 with Timestamps*).

Algorithm 2

Algorithm 2 uses the same procedure for a Read operation as Algorithm 1. The procedure for `Write` in Algorithm 2 uses totally-ordered broadcast to deliver the new value of the written variable to all the replicas. By using totally-ordered broadcast to propagate new values to the replicas, Algorithm 2 ensures that all replicas receive all the updates in an identical order. Thus, this approach ensures that eventually all replicas will have an identical state, unlike the original Algorithm 1. Algorithm 2 also avoids violation of causality, unlike Algorithm 1T, although this may not be immediately apparent.

When process p_i invokes operation `Write(X,v)`:

(step w5) p_i performs a *totally-ordered multicast* of $Write(X, v)$.

(step w6) When totally-order multicast message $Write(X, v)$ is delivered to process p_i, write value v to the local copy of X at p_i.

(step w7) Return *Ack()* indicating completion of the `Write(X,v)` operation

When process p_i is delivered totally-ordered multicast message $Write(Y, w)$ for multicast performed by some other process p_j:

(step w8) Write value w to the local copy of Y at p_i.

When process p_i invokes operation `Read(X)`:

(step r1) Read local copy of variable X at process p_i; suppose that the value read is v.

(step r2) Return *Ack(X,v)* to p_i as the response to `Read(X)`, which also indicates the completion of the `Read(X)` operation.

The key distinction between Algorithms 1 and 2 is the use of totally-ordered broadcast to deliver $Write$ requests to the replicas. Figure 2 shows an execution that uses Algorithm 2. Although the operations performed in Figure 2 are identical to those in Figure 1, the outcomes are different. In particular, in Figure 2, observe that when processes p_1 and p_3 invoke `Write(X,2)` and `Write(X,5)`, respectively, they performs totally-ordered multicast of $Write(X, 2)$, and $Write(X, 5)$. Due to the use of total-ordering, all processes deliver the multicast messages in identical order, as shown in Figure 2. $Write(X, 5)$ is delivered first, followed by $Write(X, 2)$. Each process writes value 5 to its local copy of X when $Write(X, 5)$ is delivered, and then writes 2 when $Write(X, 2)$ is delivered. In particular, process p_1 updates X to 5 first because the multicast message from p_3 is delivered before p_1's own multicast message. Using totally-ordered multicast for the *Write* messages ensures that all replicas finish the execution with value 2 in X.

Algorithm 3

For a Write operation, Algorithm 3 uses the same procedure as Algorithm 2, but in Algorithm 3, the procedure for Read operations also uses totally-ordered broadcast. In Algorithm 3, total ordering is achieved over *all* operations, Reads and Writes both.

When process p_i invokes operation `Write(X,v)`:

(step w5) p_i performs a *totally-ordered multicast* of $Write(X, v)$.

(step w6) When totally-order multicast message $Write(X, v)$ is delivered to process p_i, write value v to the local copy of X at p_i.

(step w7) Return *Ack()* indicating completion of the `Write(X,v)` operation

When process p_i is delivered totally-ordered multicast message $Write(Y, w)$ for multicast performed by some other process p_j:

(step w8) Write value w to the local copy of Y at p_i.

When process p_i invokes operation `Read(X)`:

(step r3) p_i performs a *totally-ordered multicast* of *Read*(X).

(step r4) When totally-ordered multicast of *Read*(X) initiated by p_i is delivered to process p_i, read the value of local copy of X at p_i; suppose that the value read is v.

(step r5) Return *Ack(X,v)* to p_i as the response to `Read(X)`, which also indicates the completion of the `Read(X)` operation.

When process p_i is delivered totally-ordered multicast of $Read(Y)$ initiated by some other process p_j:

(step r6) No action is needed. Discard the message.

In Algorithm 3, Reads and Writes both result in totally-ordered broadcast. The behavior for Write is identical to that of Algorithm 2. For `Read(X)` operation invoked by process p_i, process p_i only reads its local copy of variable X after it is delivered its own totally-ordered multicast message $Read(X)$ (as per step r4). All other processes besides p_i simply ignore the multicast of $Read(X)$ by p_i, as per step r6. Figure 3 illustrates the execution of operations listed in Figure 1.1. Compare Figure 3 with Figure 2.

1.3 Consistency Models

A consistency model describes the constraints that must be satisfied by executions using shared memory. A distributed shared memory system is said to implement a certain consistency model if all executions using the DSM satisfy the desired consistency model. A general yet impractical method to define a consistency model is by using a set of executions, which we will call its *execution set*. An execution satisfies a certain consistency model if and only if the execution is in the *execution set* specified for that consistency model. In practice, this type of definition for a consistency model is not very useful, since it requires us to enumerate all the allowed executions. It is better to have a more compact way of defining a consistency model. As an alternative, we may use a DSM implementation itself as the definition of a consistency model. For instance, we may define a consistency model as the set of all executions that can occur with Algorithm 2 presented earlier. This results in a more compact specification since we only need to specify the algorithm, not the actual execution set. But this is still not the most desirable way to specify the consistency model since it does not really reveal anything interesting about the properties satisfied by the executions in the execution set. Therefore, the preferred approach is to define a consistency model using a set of desirable properties that must be satisfied by the executions, and then develop algorithms that ensure that all resulting executions will satisfy the desired properties. In fact, the DSM algorithms presented earlier implement some well-known consistency models to be defined soon.

In our illustrations of executions using the different DSM algorithms, we showed the times at which various operations were invoked and when they were completed (i.e., when their responses were received). We also showed the messages that were sent to implement the shared memory operations. These messages are implementation-dependent. For the purpose of defining consistency, the details of the implementations are not of interest. Only the sequence of operations performed, timing of invocation and response of each operations, and the returned values (for Reads) are taken into account when determining whether an execution satisfies a certain consistency model. With this in mind, Figures 4, 5 and 6 redraw the executions in Figures 1, 2 and 3, respectively. The redrawn figures show the invocations and responses for all the operations, but omit the internal details of the DSM implementation.

A large number of consistency models have been proposed so far. We will define four consistency models in this chapter:

- Linearizability (also called Atomic Consistency)
- Sequential consistency
- Causal consistency
- Eventual consistency

Before we can define these consistency models, we need to introduce some terminology.

A permutation of the operations in a given execution consists of a single sequence in which all the operations performed by all the processes are included, and the operations performed at any single process appear in the same order as in the given execution. For instance, let us consider the following permutation of the operations in Figure 5. To help identify the process that performed each operation, the process index is included as a subscript below. Also, the value returned by a read operation is included as the second parameter of the Read. Thus, Read(X) at process p_1 that returned value 5 is written as $Read_1(X, 5)$ below.

$$Write_1(X, 2),\ Write_3(X, 5),\ Read_1(X, 2),\ Read_2(X, 5),\ Read_2(X, 2),\ Read_1(X, 2)$$

In the execution in Figure 5, some of the operations overlap in time (for instance, $Write_1(X, 2)$ and $Write_3(X, 5)$), and yet in the above permutation we specify them in a single totally-ordered sequence.

Observe that the operations performed by p_1 appear in the above permutation in the same order as the order of operations at p_1 in the execution in Figure 5. In fact, this is true for all the processes. Hereafter we will only consider permutations that have this **per-process order-preserving property**.

Now let us consider whether the above permutation "makes sense". Specifically, if the operations were indeed to be performed – one at a time – in the order specified in the above permutation, could the various Reads indeed return the specified values? The answer is `no` for the above permutation, since $Read_1(X,2)$ returns 2, but the last Write prior to this Read in the permutation is $Write_3(X,5)$. We want to consider only those permutations that "make sense".

In particular, we will say that a **permutation is valid** if each Read operation in the permutation returns the value written by the most recent preceding write to the variable accessed by that Read (if no such Write exists, then the Read must return the initial value of the variable). While the above permutation is not valid, the permutation below (of the same set of operations in Figure 5) is valid.

$$Write_3(X,5),\ Read_2(X,5),\ Write_1(X,2),\ Read_2(X,2),\ Read_1(X,2),\ Read_1(X,2)$$

In fact, there is an alternate permutation for the operations that is also valid (as noted below, we only consider permutations that also preserve the order of operations at each individual process).

$$Write_3(X,5),\ Read_2(X,5),\ Write_1(X,2),\ Read_1(X,2),\ Read_2(X,2),\ Read_1(X,2)$$

We will say that a **permutation preserves the real-time order** of operations provided that the following holds true: if operation $op1$ finishes (i.e., its response is received) before operation $op2$ is invoked, then $op1$ must appear in the permutation before $op2$.

Does the last permutation above preserve the real-time order? The answer is **no**, because $Write_1(X,2)$ finishes at p_1 before $Read_2(X,5)$ is invoked at p_2, and yet $Read_2(X,5)$ appears before $Write_1(X,2)$ in the permutation. Is there any permutation of the execution in Figure 5 that is **per-process order-preserving**, **valid** and **real-time order-preserving**? You should convince yourself that no such permutation exists for the operations in Figure 5.

Now consider the execution in Figure 6. In this case, there is indeed a permutation that is **per-process order-preserving**, **valid** and **real-time order-preserving**, as shown below.

$$Write_3(X,5),\ Read_2(X,5),\ Write_1(X,2),\ Read_1(X,2),\ Read_2(X,2),\ Read_1(X,2)$$

In fact, this permutation is identical to the last permutation above for Figure 5. Why is it preserving real-timer order for Figure 6 when it was not achieving that property previously? This is because the timing of the operations is affected by the use of total-ordering of Read and Write in Algorithm 3 (on the other hand, Algorithm 2, on totally orders the Writes). In Figure 6, observe that $Write_1(X,2)$ and $Read_2(X,5)$ are overlapping in time – thus, the real-time order property does **not** dictate any particular order between them in the permutation. The real-time property only applies if one operation finishes before another is invoked, and does not apply to overlapping operations. In Figure 6, the two Writes are also overlapping, and similarly the last Read by process p_1 and p_2 are overlapping.

With this background, now we are ready to define two important consistency models. **In all the discussion here, it is assumed that each process can have only one pending operation**

at any time. For instance, a process may not invoke a Write operation, and invoke a Read operation before the response for that Write is received.

Linearizability

Definition 1 *Linearizability: An execution is said to be linearizable, if* **there exists** *a permutation of the operations in the execution that is* **per-process order-preserving***,* **valid** *and* **real-time order-preserving***.*

While every operation requires a non-zero amount of time, in an execution satisfying *linearizability*, each operation "appears to take effect" instantaneously. More precisely, an execution satisfying linearizablity is equivalent to another (hypothetical) execution that satisfies the following conditions:

1. For each operation, there exists a time instant – referred to as its *linearization point* – at which the operation takes effect instantaneously,
2. The linearization point for an operation occurs sometime between its invocation and response.
3. No two operations have their linearization points at the exact same time, and
4. The permutation of the operations in the order of their linearization points is valid.

In fact, one way to verify whether a given execution is linearization is to identify linearization points for the operations such that the above three conditions are satisfied. Figure 7 shows a choice of linearization points for the various operations that satisfies the above conditions. Observe that the order of the operations – according to the real-time at which the corresponding linearization points occur – is identical to the order of the operations in the permutation below (which we have seen earlier).

$$Write_3(X,5),\ Read_2(X,5),\ Write_1(X,2),\ Read_1(X,2),\ Read_2(X,2),\ Read_1(X,2)$$

The linearization points satisfying all the conditions above may not necessarily be unique. Figure 8 shows another choice of linearization points for the same execution that also satisfies all the above conditions. Also, not all possible choices of linearization points for a given execution may satisfy the above conditions. For the execution to be linearizable, we only need to be able to find one set of linearization points that satisfies the conditions.

For a given execution, if there **does not exist** any set of linearization points that satisfies all the four conditions above, then the execution is **not linearization**. You should convince yourself that the execution in Figure 5 is not linearizable.

Linearizability is the strongest of the consistency conditions we will consider. We will say that a consistency model C_1 is stronger than consistency model C_2 if every execution that satisfies C_1 also satisfies C_2. In other words, the *execution set* of C_1 is fully contained in the *execution set* of C_2. If C_1 is stronger than C_2, then C_2 is said to be the weaker consistency model.

Sequential Consistency

The sequential consistency model drops the real-time order-preserving requirement imposed by linearizability.

Definition 2 *Sequential consistency: An execution is said to be sequentially consistent if there exists a permutation of the operations in the execution that is* **per-process order-preserving** *and* **valid**.

Since linearizability also requires that the executions be per-process order-preserving and valid, in addition to requiring the real-time order-preserving property, *every linearizable execution is also sequentially consistent.* However, since linearizability imposes an additional constraint compared to sequential consistency, the converse is not true. That is, there exist executions that are sequentially consistent but not linearizable. Recall from our prior discussion that For the execution in Figure 5 the permutation below is per-process order-preserving and valid. Thus, the execution in Figure 5 satisfies sequential consistency.

$$Write_3(X,5),\ Read_2(X,5),\ Write_1(X,2),\ Read_1(X,2),\ Read_2(X,2),\ Read_1(X,2)$$

On the other hand, for the execution in Figure 4, there does not exist any permutation that is per-process order-preserving and valid. Hence that execution in Figure 4 does not satisfy sequential consistency.

1.4 Implementing Linearizability and Sequential Consistency

All executions resulting from shared memory implemented using Algorithm 2 satisfy sequential consistency. All executions resulting from shared memory implemented using Algorithm 3 satisfy linearizability. Algorithm 1 along with the *last-writer-wins* rule achieves eventual consistency.

1.5 Composability

Given an execution E, we define $E|x$ as the execution that includes only those operations in E for which shared memory location x is an argument. For instance, if the execution in Figure 9 is called E, then the execution in Figure 10 is $E|X$ ("E mod X"), and the execution in Figure 11 is $E|Y$.

The *linearizability* consistency model is *composable* in the following sense:

- An execution E is linearizable if and only if, for each shared memory location X, $E|X$ is linearizable.

Sequential consistency is not composable. For instance, Figure 12 shows execution F. Observe that F does not satisfy sequential consistency. However, $F|X$ and $F|Y$ both satisfy sequential consistency.

1.6 Happened-Before for Shared Memory

We first define *program-order* and *read-from* relations, and use them to define the *happened-before* relations for shared memory operations Read and Write.

Program Order: If a given process completes operation O_1 some time prior to invoking operation O_2, then O_1 is said to appear before O_2 in the program order. We will denote that O_1 appears before O_2 in program order as

$$O_1 < O_2.$$

Note that when $O_1 < O_2$, there may exist other operations at the same process that occur between O_1 and O_2. That is, O_1 and O_2 are not necessarily consecutive operations at a given process. For instance, in Figure 4, $Write_1(X, 2) < Read_1(X, 5)$ and $Read_2(X, 5) < Read_2(X, 2)$.

Reads-From: We will say that a `Read` operation R reads-from a `Write` W operation (on the same shared variable) provided that R returns the value written by W.

Since it is possible for different write operations to write the same value to the same variable, the above definition is somewhat ambiguous. To simplify the discussion, let us pretend that each write to a given variable writes a unique value (for instance, we can make this property true by adding a unique timestamp to each write operation). When R reads-from W, we will denote this by

$$W \dashrightarrow R$$

For instance, in Figure 4, $Write_3(X,5) \dashrightarrow Read_2(X,5)$.

Happened-Before for shared memory: If operation O_1 happened-before O_2, we will denote that as

$$O_1 \rightarrow O_2$$

This notation is identical to that for happened-before in message-passing systems. Three rules together define the happened-before relation.

- (Program order) If $O_1 < O_2$, then $O_1 \rightarrow O_2$. That is, of O_1 appears before O_2 in the program order at a certain process, then O_1 happened-before O_2.
- (Read-From) If O_1 is a Write and O_2 is a Read, and $O_1 \dashrightarrow O_2$, then $O_1 \rightarrow O_2$. That is, if a given Read operation reads-from a particular Write operation, then that Write operation happened-before the given Read.
- (Transitivity) If $O_1 \rightarrow O_2$ and $O_2 \rightarrow O_3$, then $O_1 \rightarrow O_3$.

For operations O_1 and O_2, if $O_1 \not\rightarrow O_2$ and $O_2 \not\rightarrow O_1$, then operations O_1 and O_2 are *concurrent*, which is denoted as $O_1 \| O_2$.

Consider Figure 4: due to the program-order rule, we have $Write_1(X,2) \rightarrow Read_1(X,5)$ and $Read_2(X,5) \rightarrow Read_2(X,2)$. Due to the read-from rule, we have $Write_3(X,5) \rightarrow Read_2(X,5)$. Finally, because $Write_3(X,5) \rightarrow Read_2(X,5)$ and $Read_2(X,5) \rightarrow Read_2(X,2)$, by transitivity, we have $Write_3(X,5) \rightarrow Read_2(X,2)$. Also, $Write_1(X,2) \| Read_2(X,5)$.

Permutations satisfying happened-before for shared memory: A permutation of operations in a given execution is said to satisfy the happened before relation provided that for any two operations op_1 and op_2 in the given execution, op_1 appears before op_2 in the permutation if ~~and only if~~ $op_1 \rightarrow op_2$.

For instance, the permutation below of the operations in Figure 9 satisfies the happened-before relation and preserves the per-process order (but it is not *valid*).

$$Write_1(X,2),\ Write_3(Y,5),\ Read_1(Y,0),\ Read_2(Y,5),\ Read_2(X,2),\ Read_1(X,2)$$

1.7 Causal Consistency

An execution E is said to satisfy *causal consistency* provided that the following property is true, **for each process** p that participates in the given execution.

- Consider an execution that includes only the following operations in E: all *write* operations at all processes (including p), and all *read* operations at process p (thus, we exclude read operations at processes other than p).

  Let us denote the above (reduced) execution by E_p.

- There exists a permutation of the events in E_p that is *valid* and satisfies the *happened-before* relation.

Consider execution F in Figure 12. For process p_1 (whose timeline appears at the top of the figure), the permutation below that satisfies the above requirements:

$$write_1(X, 2),\ Read_1(Y, 0),\ Write_2(Y, 5)$$

Similarly for process p_2 in execution F, the permutation below that satisfies the above requirements:

$$Write_2(Y, 5),\ Read_2(X, 0),\ Write_1(X, 2)$$

Hence execution F is *causally consistent.* However, as noted earlier, this execution is *not sequentially consistent.*

In the first permutation above, $Write_1(X, 2)$ appears before $Write_2(Y, 5)$. On the other hand, in the second permutation, the order is opposite. Both orders satisfy *happened-before* because $Write_1(X, 2)$ and $Write_2(Y, 5)$ are concurrent operations. Thus, under *causal consistency*, different processes are allowed to "see" these two writes in different order. Sequential consistency does *not* provide this flexibility.