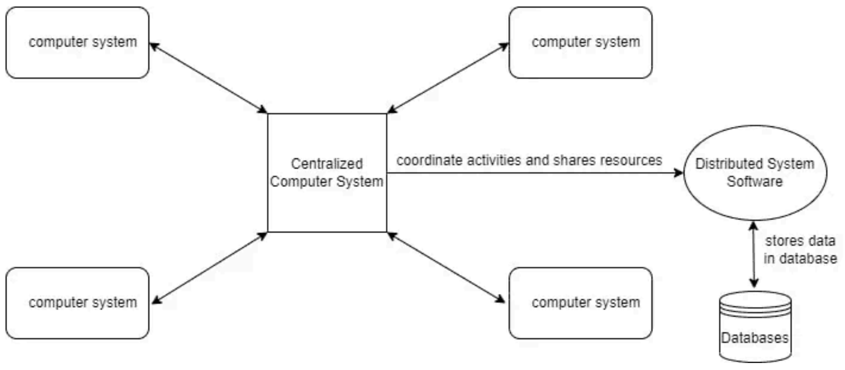
● Data-Centric Consistency Models

The Data Centric Consistency Model is a set of principles and protocols used in distributed systems to define how data is accessed and updated, ensuring that all nodes in the system agree on the state of shared data.

It primarily focuses on the consistency guarantees provided to clients regarding the data they access and manipulate. In a distributed system, data is stored and managed across multiple nodes that communicate and collaborate to provide services and respond to client requests.


computer system
computer system
Centralized Computer System
coordinate activities and shares resources
Distributed System Software
stores data in database
computer system
computer system
Databases

However, due to factors like network delays, node failures, and concurrent data updates, achieving consistency in such an environment can be challenging.

Types of Data Centric Consistency Model:

Data-centric consistency models are a set of principles and protocols that define how data is accessed and updated in a distributed system, ensuring that all nodes agree on the state of shared data.

These models primarily focus on the consistency guarantees provided to clients regarding the data they access and manipulate. Let's explore the most common data-centric consistency models in detail:

Sequential Consistency

The time axis is always drawn horizontally, with time increasing from left to right. The symbols

$$W_i(x)a \text{ and } R_i(x)b$$

mean that a write by process P; to data item x with the value a and a read from that item by Pi returning b have been done, respectively. We assume that each data item is initially NIL. When there is no confusion concerning which process is accessing data, we omit the index from the symbols W and R.

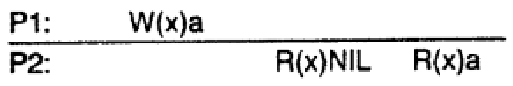


Figure 7-4. Behavior of two processes operating on the same data item. The horizontal axis is time.

As an example, in Fig. 7-4 PI does a write to a data item x, modifying its value to a. Note that, in principle, this operation WI (x)a is first performed on a copy of the data store that is local to PI, and is then subsequently propagated to the other local copies. In our example, P2 later reads the value NIL, and some time after that (from its local copy of the store). What we are seeing here is that it took some time to propagate the update of x to P2, which is perfectly acceptable.

In general, a data store is said to be sequentially consistent when it satisfies the following condition:

The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of-each individual process appear in this sequence in the order specified by its program.

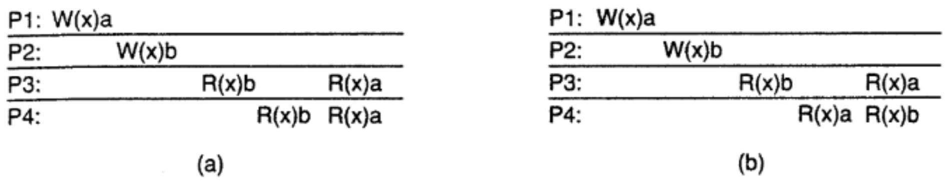
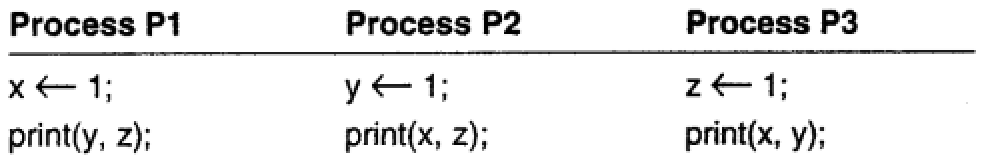


Figure 7-5. (a) A sequentially consistent data store. (b) A data store that is not sequentially consistent..

Consider four processes operating on the same data item x. In Fig. 7-5(a) process PI first performs W(x)a to x. Later (in absolute time), process P2 also performs a write operation,

by setting the value of x to b. However, both processes P3 and P4 first read value b, and later value a. In other words, the write operation of process P2 appears to have taken place before that of PI·

In contrast, Fig.7-5(b) violates sequential consistency because not all processes see the same interleaving of write operations. In particular, to process P3, it appears as if the data item has first been changed to b, and later to a. On the other hand, P4 will conclude that the final value is b.



Process P1	Process P2	Process P3
x ← 1;	y ← 1;	z ← 1;
print(y, z);	print(x, z);	print(x, y);

Figure 7-6. Three concurrently-executing processes.

To make the notion of sequential consistency more concrete, consider three concurrently-executing processes PI, P2, and P3, shown in Fig. 7-6 (Dubois et aI.,1988). The data items in this example are formed by the three integer variables x, y, and z, which are stored in a (possibly distributed) shared sequentially consistent data store. We assume that each variable is initialized to O. In this example, an assignment corresponds to a write operation, whereas a print statement corresponds to a simultaneous read operation of its two arguments. All statements are assumed to be indivisible.

Causal Consistency

The causal consistency model (Hutto and Ahamad, 1990) represents a weakening of sequential consistency in that it makes a distinction between events that are potentially causally related and those that are not. We already came across causality when discussing vector timestamps in the previous chapter. If event b is caused or influenced by an earlier event a, causality requires that everyone else first see a, then see b.

Consider a simple interaction by means of a distributed shared database. Suppose that process P, writes a data item x. Then P2 reads x and writes y. Here the reading of x and the writing of y are potentially causally related because the computation of y may have depended on the value of x as read by Pz (i.e., the value written by PI)' On the other hand, if two processes spontaneously and simultaneously write two different data items, these are not causally related. Operations that are not causally related are said to be concurrent. For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

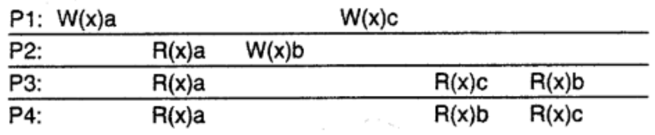


Figure 7-8. This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.

As an example of causal consistency, consider Fig. 7-8. Here we have an event sequence that is allowed with a causally-consistent store, but which is forbidden with a sequentially-consistent store or a strictly consistent store. The thing to note is that the writes Wz(x)b and WI (x)c are concurrent, so it is not required that all processes see them in the same order.

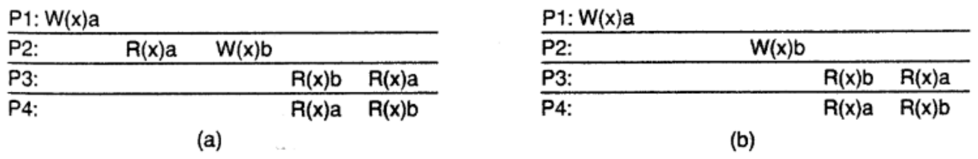


Figure 7-9. (a) A violation of a causally-consistent store. (b) A correct sequence of events in a causally-consistent store.

Now consider a second example. In Fig. 7-9(a) we have Wz(x)b potentially depending on WI (x)a because the b may be a result of a computation involving the value read by Rz(x)a. The two writes are causally related, so all processes must see them in the same order. Therefore, Fig. 7-9(a) is incorrect. On the other hand, in Fig. 7-9(b) the read has been removed, so WI (x)a and Wz(x)b are now concurrent writes. A causally-consistent store does not require concurrent writes to be globally ordered, so Fig.7-9(b) is correct. Note that Fig.7-9(b) reflects a situation that would not be acceptable for a sequentially consistent store.

Grouping Operations

Sequential and causal consistency are defined at the level of read and write operations. This level of granularity is for historical reasons: these models have initially been developed for shared-memory multiprocessor systems and were actually implemented at the hardware level.

The fine granularity of these consistency models in many cases did not match the granularity as provided by applications. What we see there is that concurrency between programs sharing data is generally kept under control through synchronization mechanisms for mutual exclusion and transactions. Effectively, what happens is that at the program level read and write operations are bracketed by the pair of operations ENTER_CS and LEAVE_CS where "CS" stands for critical section. As we explained in Chap. 6, the synchronization between processes takes place by means of these two operations. In terms of our distributed data store, this means that a process that has successfully executed ENTER_CS will ensure that the data in its local store is up to date. At that point, it can safely execute a series of read and write operations on that store, and subsequently wrap things up by calling LEAVE_CS.

In essence, what happens is that within a program the data that are operated on by a series of read and write operations are protected against concurrent accesses that would lead to seeing something else than the result of executing the series as a whole. Put differently, the bracketing turns the series of read and write operations into an atomically executed unit, thus raising the level of granularity.

In order to reach this point, we do need to have precise semantics concerning the operations ENTER_CS and LEAVE_CS. These semantics can be formulated in terms of shared synchronization variables. There are different ways to use these variables. We take the general approach in which each variable has some associated data, which could amount to the complete set of shared data. We adopt the convention that when a process enters its critical section it should acquire the relevant synchronization variables, and likewise when it leaves the critical section, it releases these variables. Note that the data in a process's critical section may be associated with different synchronization variables.

Each synchronization variable has a current owner, namely, the process that last acquired it. The owner may enter and exit critical sections repeatedly without having to send any messages on the network. A process not currently owning a synchronization variable but wanting to acquire it has to send a message to the current owner asking for ownership and the current values of the data associated with that synchronization variable. It is also possible for several processes to simultaneously own a synchronization variable in nonexclusive mode, meaning that they can read, but not write, the associated data.

We now demand that the following criteria are met:

1. An acquired access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.

2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.

3. After an exclusive mode access to a synchronization variable has been performed, any other process' next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

The first condition says that when a process does an acquire, the acquire may not complete (i.e., return control to the next statement) until all the guarded shared data have been brought up to date. In other words, at an acquire, all remote changes to the guarded data must be made visible.

The second condition says that before updating a shared data item, a process must enter a critical section in exclusive mode to make sure that no other process is trying to update the shared data at the same time.

The third condition says that if a process wants to enter a critical region in nonexclusive mode, it must first check with the owner of the synchronization variable guarding the critical region to fetch the most recent copies of the guarded shared data.

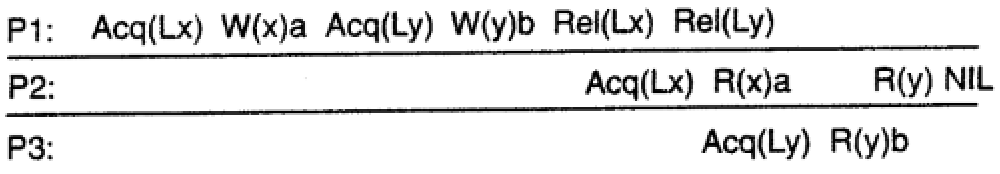

P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)
P2: Acq(Lx) R(x)a R(y) NIL
P3: Acq(Ly) R(y)b

Figure 7·10. A valid event sequence for entry consistency.

Fig. 7-10 shows an example of what is known as entry consistency. Instead of operating on the entire shared data, in this example we associate locks with each data item. In this case, P I does an acquire for x, changes x once, after which it also does an acquire for y. Process P2 does an acquire for x but not for Y'.so that it will read value a for x, but may read NIL for y. Because process P3 first does an acquire for y, it will read the value b when y is released by Pl'.