A consistency model is essentially a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly.

Normally, a process that performs a read operation on a data item, expects the operation to return a value that shows the results of the last write operation on that data.

In the absence of a global clock, it is difficult to define precisely which write operation is the last one.

As an alternative, we need to provide other definitions, leading to a range of consistency models.

Each model effectively restricts the values that a read operation on a data item can return.

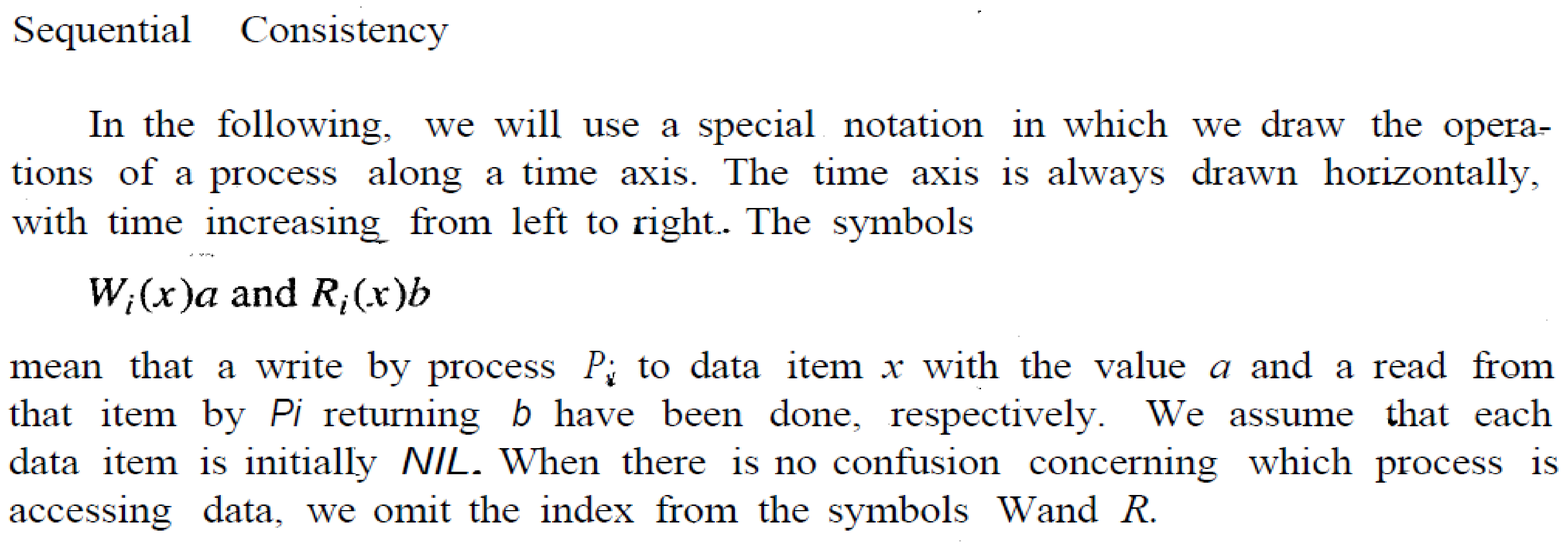
As is to be expected, the ones with major restrictions are easy to use, for example when developing applications, whereas those with minor restrictions are sometimes difficult.

The tradeoff is, of course, that the easy-to-use models do not perform nearly as well as the difficult ones. Such is life.

Strict consistency, also known as atomic consistency or linearizability, is a strong consistency model in distributed shared memory (DSM) systems. In a strictly consistent DSM system, every operation appears to take effect instantaneously at some point between its invocation and response, as if the operations occurred atomically in a global total order.
This model ensures that all processes observe a single, coherent view of shared memory at all times.
Example: Consider a distributed key-value store where multiple clients concurrently access shared data stored across multiple nodes. In a system that enforces strict consistency:

1. Client A sends a request to update a key-value pair (e.g., set key "X" to value "V1").
2. Client B sends a request to read the value associated with the same key.
3. Client B receives a response with the value "V1," indicating that the update from Client A has taken effect.

In this scenario, strict consistency guarantees that the read operation by Client B reflects the most recent update made by Client A, as if the update occurred instantaneously before the read operation. The system ensures that all processes observe the operations in a global total order, providing a consistent and coherent view of shared memory across all nodes.



Sequential Consistency

In the following, we will use a special notation in which we draw the operations of a process along a time axis. The time axis is always drawn horizontally, with time increasing from left to right. The symbols

$W_i(x)a$ and $R_i(x)b$

mean that a write by process P_i 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 Wand *R*.

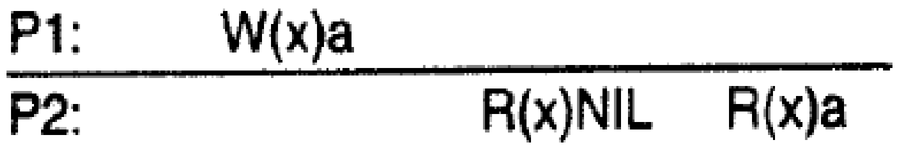
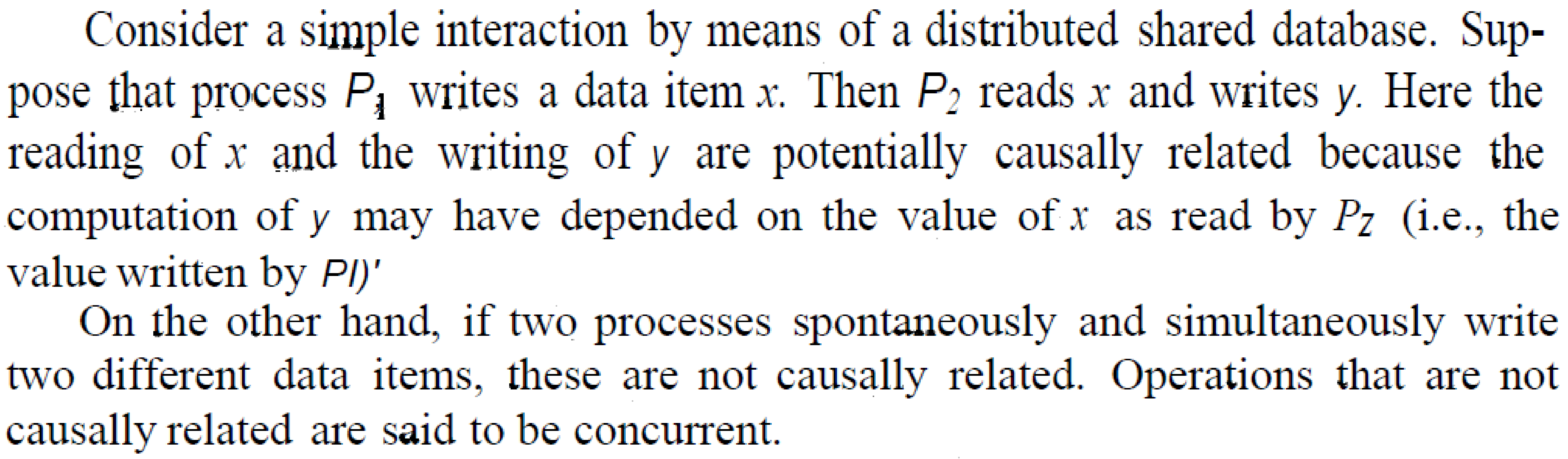


Figure 7-4. Behavior of two processes operating on the same data item. The horizontal axis is time.

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, P_2 later reads the value *NIL*, and some time after that *a* (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 P_2, which is perfectly acceptable.

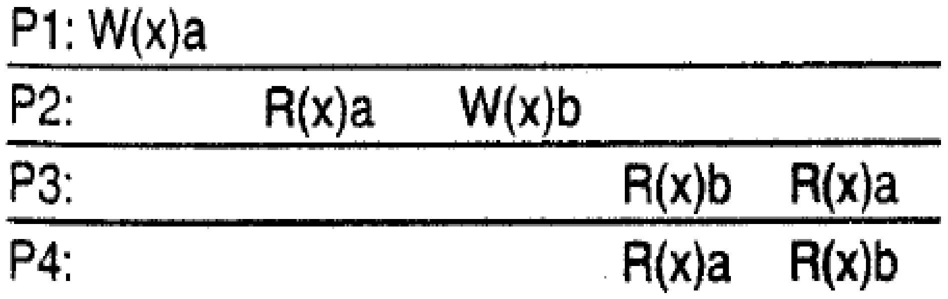
Causal Consistency



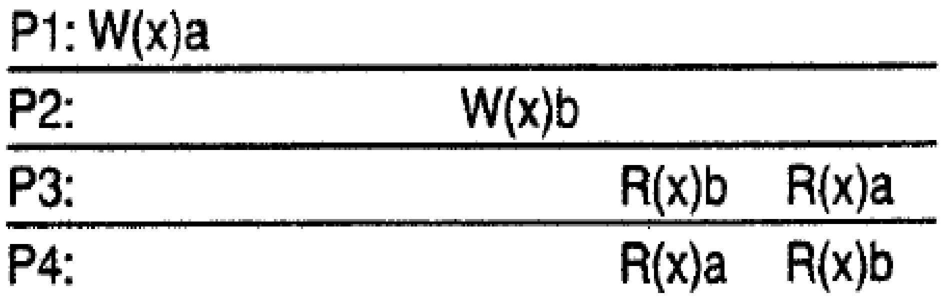
Consider a simple interaction by means of a distributed shared database. Suppose that process P_1 writes a data item x. Then P_2 reads x and writes y. Here the reading of x and the writing of y are potentially causally related because the computation of y may have depended on the value of x as read by P_Z (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.

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.



(a)



(b)

Problem Scenario: Achieving Strict Consistency in a Distributed Key-Value Store
Consider a distributed key-value store deployed across multiple nodes in a cloud environment. Clients interact with the key-value store to read and update data stored in the system. The goal is to ensure strict consistency, where all clients observe a single, coherent view of the shared data.

1. **Problem Description**:
    1. Multiple clients concurrently access the distributed key-value store to read and update key-value pairs.
    2. The system must ensure that all clients observe updates to the key-value pairs in a globally agreed-upon order, ensuring strict consistency.
    3. The distributed nature of the system and network delays pose challenges to maintaining strict consistency, as operations may arrive at different nodes in different orders.

Solution:

Implement Distributed Locking: Use distributed locking mechanisms to coordinate access to shared data and enforce a global total order of operations.

Sequentialize Access: Require clients to acquire a distributed lock before performing read or write operations on the key-value store. This ensures that only one client can modify the shared data at a time.

Use Consensus Protocols: Employ consensus protocols such as Paxos or Raft to agree on the order of operations across all nodes in the system.

Serialize Operations: Serialize read and write operations on the key-value store based on the agreed-upon order determined by the consensus protocol.

Guarantee Atomicity: Ensure that each operation appears to take effect instantaneously at some point between its invocation and response, providing the illusion of atomic consistency.

Implement Rollback Mechanisms: Introduce rollback mechanisms to revert operations that violate the agreed-upon order or consistency constraints.

Granularity in distributed shared memory (DSM) systems refers to the size or scope of data units that are shared among processes in a distributed computing environment. It determines the level of detail at which data is accessed, manipulated, and synchronized across multiple nodes in the system. Granularity plays a crucial role in the performance, scalability, and efficiency of DSM systems, as it affects communication overhead, synchronization complexity, and resource utilization.

There are typically two main levels of granularity in DSM systems:

1. **Coarse-Grained Granularity**:
    1. In coarse-grained DSM systems, large blocks of data are shared among processes, reducing the frequency of data accesses and synchronization operations.
    2. Data units at this level of granularity may include entire data structures, objects, or files, which are accessed and modified as a single entity.
    3. Coarse-grained granularity simplifies synchronization and reduces communication overhead, as fewer synchronization points are required to coordinate access to shared data.
    4. However, coarse-grained granularity may lead to increased contention for shared resources and reduced parallelism, especially in scenarios where multiple processes need to access different parts of the shared data concurrently.

2.Fine-Grained Granularity:

2. In fine-grained DSM systems, small units of data, such as individual variables, records, or elements of data structures, are shared among processes.
3. Data units at this level of granularity allow for more fine-grained control over data access and synchronization, enabling concurrent access to different parts of the shared data.
4. Fine-grained granularity increases parallelism and concurrency by allowing multiple processes to access and modify different parts of the shared data simultaneously.
5. However, fine-grained granularity may introduce higher communication overhead and synchronization complexity, as more frequent synchronization operations are required to coordinate access to fine-grained data units.

The choice of granularity in DSM systems depends on various factors, including the characteristics of the application, the communication patterns among processes, the nature of shared data, and the performance requirements of the system. Different applications may benefit from different levels of granularity based on their specific needs and constraints.

It's important to strike a balance between coarse-grained and fine-grained granularity to optimize performance, scalability, and efficiency in DSM systems. Hybrid approaches that combine elements of both coarse-grained and fine-grained granularity may also be employed to achieve the desired balance and meet the requirements of diverse applications.

Problem Scenario: Granularity Selection in a Distributed File Storage System
In a distributed file storage system deployed across multiple servers, clients access and manipulate files stored in the system. The challenge is to determine the appropriate granularity level for sharing files among servers to optimize performance, minimize communication overhead, and ensure efficient resource utilization.

Problem Description:

The distributed file storage system consists of multiple servers interconnected over a network, with each server responsible for storing and serving a portion of the files.

Clients access files stored in the system for read and write operations, and the granularity of file sharing affects the efficiency of data access and synchronization.

The system must strike a balance between coarse-grained and fine-grained granularity to optimize performance, scalability, and resource utilization.

Solution:

- Evaluate Access Patterns: Analyze the access patterns of clients to understand how files are accessed and manipulated in the system. Identify frequently accessed files and the nature of concurrent access by multiple clients.
- Determine Granularity Levels: Based on the analysis, determine the appropriate granularity levels for sharing files among servers. Consider factors such as file size, access frequency, and concurrency of access.

Coarse-Grained Granularity:

Store entire files on each server to minimize communication overhead and simplify data access. Clients can access and manipulate entire files without the need for fine-grained synchronization.

Suitable for large files or scenarios where files are accessed and modified as a whole, reducing the complexity of synchronization and resource contention.

Fine-Grained Granularity:

Divide files into smaller chunks or blocks and distribute them across multiple servers to increase parallelism and concurrency. Clients can access and modify individual blocks of files independently.

Suitable for scenarios where files are large and accessed by multiple clients concurrently, allowing for finer control over data access and synchronization.

Hybrid Granularity:

Employ a hybrid approach that combines elements of coarse-grained and fine-grained granularity based on the characteristics of files and access patterns.

Store frequently accessed files with coarse-grained granularity to minimize overhead, while using fine-grained granularity for large files with high concurrency requirements.

Q: What kind of consistency would you use to implement an electronic stock market? Explain your answer.
A: Causal consistency is probably enough. The issue is that reactions to changes in stock values should be consistent. Changes in stocks that are independent can be seen in different orders

Q: Explain in your own words what the main reason is for actually considering weak consistency models.
A: Weak consistency models come from the need to replicate for performance. However, efficient replication can be done only if we can avoid global synchronizations, which, in turn, can be achieved by loosening consistency constraints.