

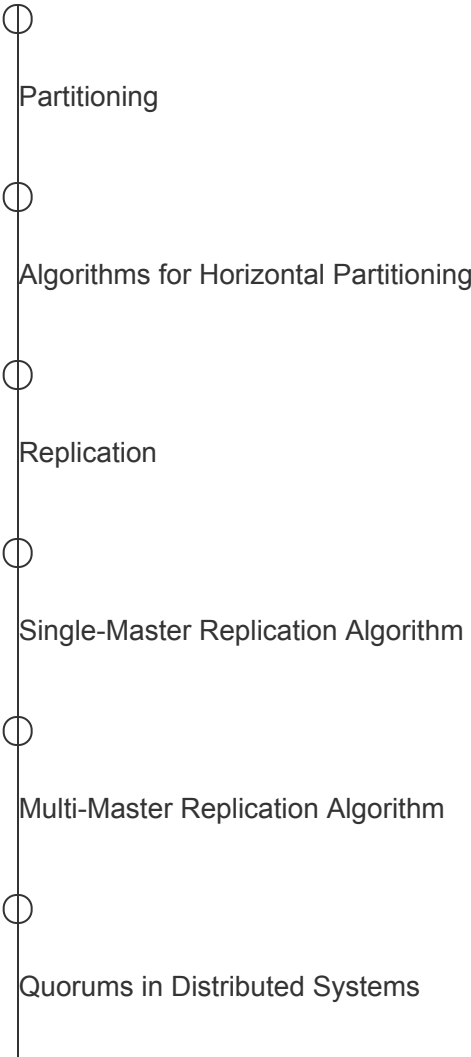
[Back To Module Home](#)


Distributed Systems

0% completed

Introduction to Distributed Systems

Basic Concepts and Theorems





	Safety Guarantees in Distributed Systems
	ACID Transactions
	The CAP Theorem
	Consistency Models
	CAP Theorem's Consistency Model
	Isolation Levels and Anomalies
	Prevention of Anomalies in Isolation Levels
	Consistency and Isolation
	Hierarchy of Models
	Why All the Formalities?
	Quiz

Conclusion

Mark Module as Completed

Consistency Models

In this lesson, we will learn the different forms of consistency.

We'll cover the following

- Consistency model
 - A strong consistency model
 - List of consistency models
 - Linearizability
 - Benefits
 - Sequential consistency
 - Example
 - Causal consistency
 - Example
 - Eventual consistency
 - Example

According to the CAP Theorem, consistency means that every successful read request will return the result of the most recent write. In fact, this is an oversimplification, because there are many different forms of consistency.

In this lesson, we introduce the forms of consistency that are most relevant to us.

To accurately define all these forms really, we need to build a formal model. This is usually the **consistency model**.

Consistency model#

The consistency model defines the set of execution histories that are valid in a system.

In layperson's terms, a model formally defines the behaviours that are possible in a distributed system.

Consistency models are extremely useful for many reasons:

- They help us formalise the behaviours of systems. Systems can then provide guarantees about their behaviour.
- Software engineers can confidently use a distributed system (i.e. a distributed database) in a way that does not violate any safety properties they care about.

In essence, software engineers can treat a distributed system as a black box that provides a set of properties. Moreover, they can do this without knowing of all the complexity the system internally assumes to provide these properties.

A strong consistency model#

We consider consistency model A stronger than model B when the first allows fewer histories. Alternatively, we say model A makes more assumptions about or poses more restrictions on, the system's possible behaviors.

Usually, the stronger the consistency model a system satisfies, the easier it is to build an application on top of it. This is because the developer can rely on stricter guarantees.

List of consistency models#

There are many different consistency models used across the modern system design field. We will focus on the most fundamental ones. These are the following:

- Linearizability
- Sequential Consistency
- Causal Consistency
- Eventual Consistency

Linearizability#

A system that supports the consistency model of **linearizability** is one where operations appear to be instantaneous to the external client. This means that they happen at a specific point—from the point the client invokes the operation, to the point the client receives the acknowledgment by the system the operation has been completed.

Furthermore, once an operation is complete and the acknowledgment is delivered to the client, it is visible to all other clients. This implies that if a client C2 invokes a read operation after a client C1 receives the completion of its write operation, C2 should see the result of this (or a subsequent) write operation. It may be obvious to some that operations are *instantaneous* and *visible* after they're completed.

In a centralized system, linearizability is obvious. The following illustration shows this.

Created with Fabric.js 3.6.6

There is a client and a single node centralized system

1 of 6

However, there is no such thing as instantaneity in a distributed system.

The following illustration shows why linearizability is not obvious in a distributed system.

Created with Fabric.js 3.6.6

A client is using a distributed system of three nodes, A, B, and C, which are asynchronous replicas, for each node X is initialized to 0

1 of 9

When we think of a distributed system as a single node, it seems obvious that every operation happens at a specific instant of time, and is immediately visible to everyone. However, when we think of a distributed system as a set of cooperating nodes, we realize that we should not take this for granted.

For instance, the system in the above illustration is not linearizable since $T_4 > T_3$. However, still, the second client won't observe the read because it has not yet propagated to the node that processes the read operation. The **non-linearizability** comes from the use of *asynchronous replication*.

When we use a *synchronous replication* technique, we make the system linearizable. However, that means that the first write operation takes longer until the new value has propagated to the rest of the nodes. Remember the latency-consistency trade-off from the PACELC theorem.

Benefits#

As a result of the above discussion, we realize that linearizability is a very powerful consistency model. It helps us treat complex distributed systems as much simpler, single-node datastores, and reason about our applications more efficiently. Moreover, by leveraging atomic instructions provided by hardware (such as CAS operations), we can build more sophisticated logic on top of distributed systems, such as mutexes, semaphores, counters, etc. This is not possible under weaker consistency models.

Sequential consistency#

Sequential consistency is a weaker consistency model, where operations are allowed to take effect before their invocation or after their completion.

As a result, it provides no real-time guarantees. However, operations from different clients have to be seen in the same order by all other clients, and operations of every single client preserve the order specified by its program (in this global order). This allows many more histories than linearizability, but still poses some constraints that can help real-life applications.

Example#

For example, in a social networking application, we usually do not care what's the ordering of posts between some of our friends. However, we still expect posts from a single friend to be displayed in the right order (i.e., the one they published them at). Following the same logic, we usually expect our friends' comments in a post to appear in the order that they submitted them. These are all properties that the sequential consistency model captures.

Causal consistency#

In some cases, we don't need to preserve the ordering specified by each client's program—as long as causally related operations are displayed in the right order. This is the **causal consistency** model, which requires that only operations that are causally related need to be seen in the same order by all the nodes.

Example#

Consider the same scenario as our previous comments example. We may want to display comments out of chronological order if it means that every comment is displayed after the comment it replies to. This is expected since there is a cause-and-effect relationship between a comment and the comments that constitute replies to it.

Thus, unlike in sequential consistency, the operations that are not causally related can be seen in different orders in the various clients of the system, without the need to maintain

the order of each client's program. Of course, to achieve that, each operation needs to contain some information that signals whether it depends on other operations or not. This does not need to at all be related to time and can be an application-specific property.

Causal consistency is a weaker consistency model that prevents a common class of unintuitive behaviors.

Eventual consistency#

There are still even simpler applications that do not have the notion of a cause-and-effect and require an even simpler consistency model. The **eventual consistency** model is beneficial here.

Example#

For instance, we could accept that the order of operations can be different between the multiple clients of the system, and reads do not need to return the latest write as long as the system eventually arrives at a stable state. In this state, if no more write operations are performed, read operations will return the same result. This is the model of eventual consistency.

It is one of the weakest forms of consistency since it does not really provide any guarantees around the perceived order of operations or the final state the system converges to.

It can still be a useful model for some applications, which do not require stronger assumptions or can detect and resolve inconsistencies at the application level.

Note that there are many more consistency models besides the ones we explained here.



Back

The CAP Theorem

Next

CAP Theorem's Consistency Model

Mark as Completed

Report an Issue