Distributed Key-Value Store

Siddhant Dilip Godshalwar

May 3, 2023

1 Introduction

Consistency models are critical in distributed systems as they ensure that the data stored in a distributed system is accurate, up-to-date, and reliable. A consistency model defines the guarantees provided by a distributed system regarding the order and visibility of updates to shared data across nodes. Different consistency models provide different trade-offs between consistency, availability, and partition tolerance.

In a distributed system, data may be replicated across multiple nodes for fault-tolerance and performance reasons. When updates are made to the data, these updates need to be propagated to all replicas in a consistent and timely manner to ensure that all nodes have the same view of the data. Consistency models ensure that all replicas see the same updates in the same order, preventing inconsistencies and conflicts.

In addition to ensuring data consistency, consistency models also help to ensure that distributed systems are scalable, fault-tolerant, and performant. By specifying the guarantees provided by a distributed system, consistency models enable developers to design and implement efficient and effective distributed systems that can handle large volumes of data, tolerate network failures and node crashes, and provide fast and reliable access to shared data.

Overall, consistency models are critical in ensuring the reliability, scalability, and performance of distributed systems, making them an essential consideration for anyone building or deploying distributed applications.

2 Implementation

2.1 Sequential Consistency

Sequential consistency is a property that ensures that the result of any execution of a concurrent system is equivalent to the result that would be obtained if the operations of all processes in the system were executed in some sequential order, without interleaving.

In other words, sequential consistency ensures that the behavior of a concurrent system is the same as if all operations were performed in a single sequence in some order. This property guarantees that the execution of concurrent operations produces a result that is consistent with the correct sequential execution of the same operations.

To achieve sequential consistency in a concurrent system, there are several techniques that can be used, such as locks, semaphores, and barriers. These techniques ensure that the execution of critical sections of code is serialized and that the results of concurrent operations are correctly synchronized and ordered.

However, achieving sequential consistency can come at the cost of reduced performance, as the serialization of operations can limit concurrency and introduce overhead. Therefore, in some cases, weaker consistency models, such as eventual consistency or relaxed consistency, may be used instead to achieve higher scalability and performance, at the cost of weaker guarantees of consistency.

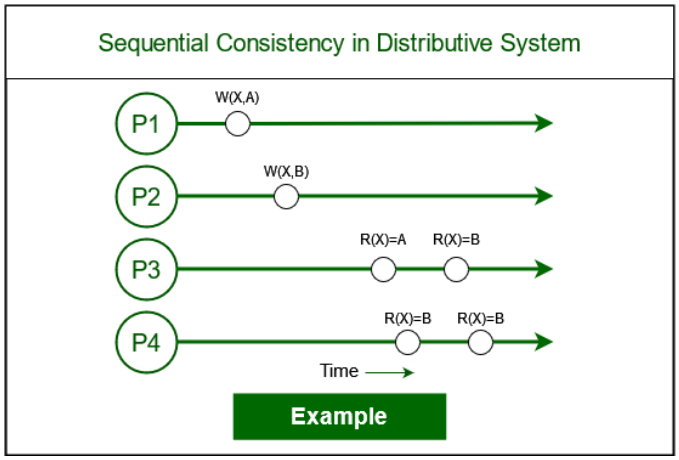

Figure 1: Sequential Consistency

2.2 Linearizable

Linearizability is a consistency model for concurrent systems that provides a strong guarantee of consistency. It is also known as atomic consistency, and it ensures that the system behaves as if there were a single copy of the data that is updated atomically.

In a linearizable system, all operations appear to execute instantaneously in a globally agreed-upon order, which is often referred to as the "linearization point". This means that the effects of a completed operation are visible to all other processes in the system immediately after the operation has been completed.

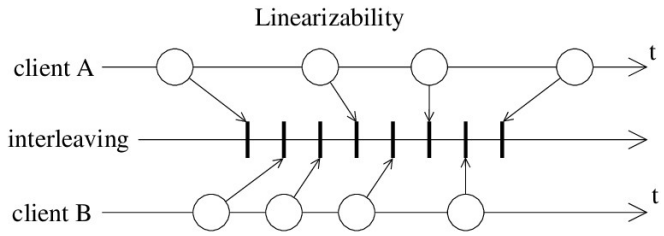

Figure 2: Linearizable

Linearizability is important in distributed systems, where data is replicated across multiple nodes, and multiple clients may access and update the data simultaneously. In such systems, linearizability ensures that updates to the replicated data are serialized and that the effects of concurrent updates are consistent with a serial order of execution.

To achieve linearizability in a distributed system, there are various techniques that can be used, such as consensus protocols, two-phase commit, and quorums. These techniques ensure that updates to the data are agreed upon by a quorum of nodes or processes and that conflicting updates are resolved in a consistent manner.

Overall, linearizability provides a strong guarantee of consistency in concurrent and distributed systems, but it can come at the cost of increased latency and reduced scalability, as updates to the data must be coordinated across multiple nodes or processes.

2.3 Eventual

Eventual consistency is a consistency model for distributed systems that provides weaker guarantees of consistency than linearizability but allows for higher scalability and availability.

In an eventually consistent system, updates to replicated data are allowed to propagate asynchronously across multiple nodes or processes, and there may be a period of time during which different

nodes or processes may see different versions of the data. However, over time, all nodes or processes will eventually converge to a consistent state.

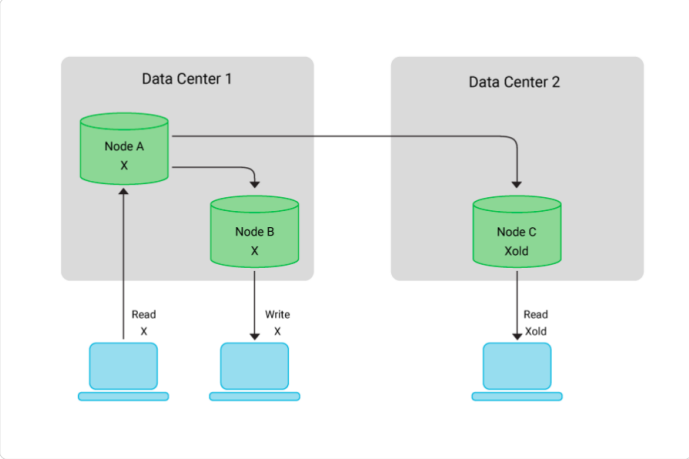

Figure 3: Eventual Consistency

Eventual consistency is based on the principle of "eventual convergence", which means that all nodes or processes will eventually see the same version of the data, provided that there are no new updates being made to the data. This means that in the presence of concurrent updates, different nodes or processes may see different versions of the data for a period of time, but eventually, all nodes or processes will converge to a consistent state.

To achieve eventual consistency in a distributed system, various techniques can be used, such as versioning, conflict resolution, and anti-entropy mechanisms. These techniques ensure that updates to the data are propagated across nodes or processes in a consistent manner and that conflicting updates are resolved in a way that is consistent with the intended semantics of the data.

Overall, eventual consistency provides weaker guarantees of consistency than linearizability but allows for higher scalability and availability in distributed systems. It is often used in systems where the cost of maintaining strong consistency is too high, such as in highly distributed or highly concurrent environments.

2.4 Causal Consistency

Causal consistency is a consistency model for distributed systems that provides a middle ground between the strong consistency of linearizability and the weaker consistency of eventual consistency. In a causally consistent system, updates to replicated data are ordered based on their causality, meaning that updates that are causally related must be observed in the same order by all nodes or processes in the system.

Causally related updates are those that are related by a cause-and-effect relationship, meaning that one update must have happened before the other in order for the updates to be causally related. For example, if two nodes in a system update different fields of the same record, the order in which the updates are propagated should be consistent with the causal relationship between the updates.

Causal consistency ensures that updates to replicated data are consistent with the intended causality of the data, and provides stronger guarantees of consistency than eventual consistency, while still allowing for some degree of asynchrony and replication latency.

To achieve causal consistency in a distributed system, various techniques can be used, such as vector clocks and dependency tracking. These techniques ensure that updates to the data are ordered based on their causal relationship, and that updates that are causally related are propagated in a consistent order across nodes or processes.

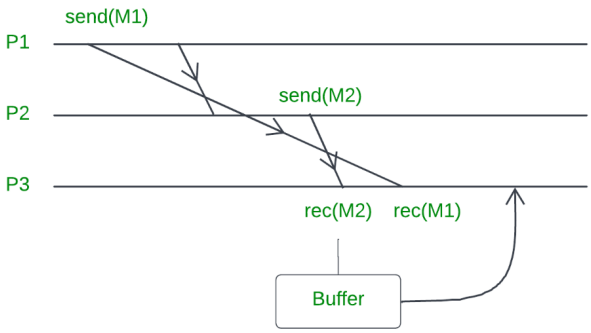

Figure 4: Causal Consistency

Overall, causal consistency provides a balance between strong consistency and scalability in distributed systems and is often used in systems where maintaining a strong causal relationship between updates is important, such as in financial systems, databases, and messaging systems.

3 Workflow

3.1 Steps to run the files:

- 1 Open the config.py file and set the number of replicas you want
- 2 Run Server.py and select the type of server you want
- 3 Run the Client.py file and select the type of client you want to be
- 4 Run multiple Client.py instances to replicate multiple clients

3.2 Output:

The output files genrated are the key value store of each individual replicas and primary in each folder named after the consistency used to achieve the key value store.

They are namely:

- 1 Sequential Consistency
- 2 Linearizable
- 3 Causal
- 4 Eventual

4 Testing

The testing screenshots for each consistency have been attached below

4.1 TestCase1:- Sequential Consistency

Clients:-2
Replicas:-3

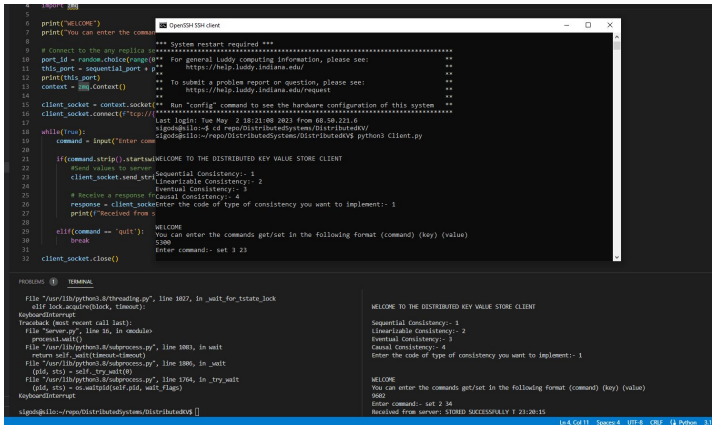

Figure 5: TestCase 1:- Sequential

4.2 TestCase2:- Linearizability Consistency

Clients:-2
Replicas:-3

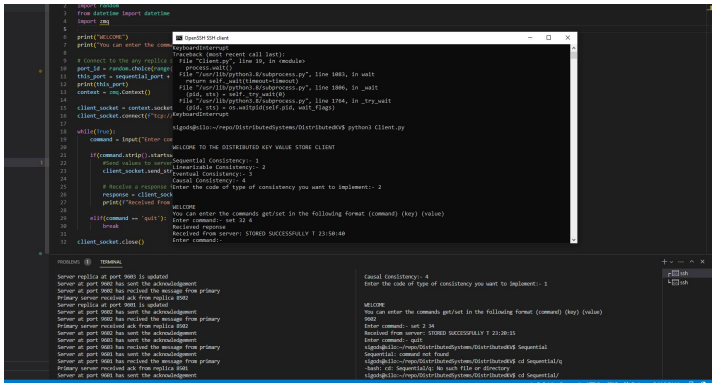

Figure 6: TestCase 2:- Linearizability

4.3 TestCase3:- Causal Consistency

Clients:-2
Replicas:-3

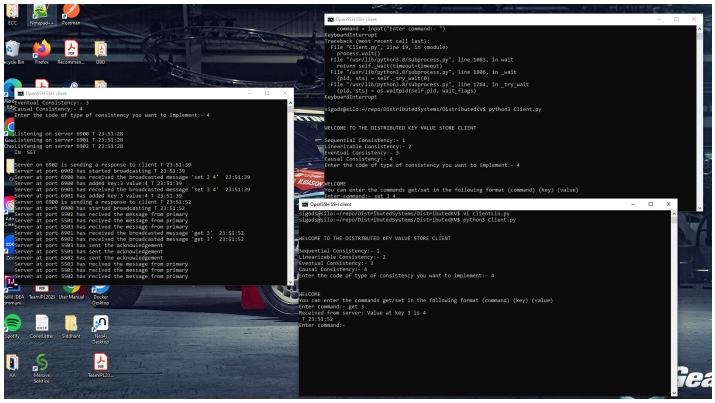

Figure 7: TestCase 3:- Causal Consistency

4.4 TestCase4:- Eventual Consistency

Clients:-2
Replicas:-3

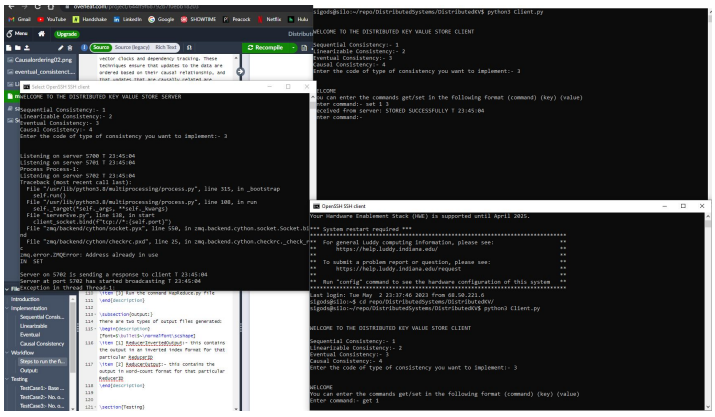

Figure 8: TestCase 4:- Eventual

5 Limitations and Assumptions

5.1 Limitation

The Limitations of this implementation is

- 1 I tried to make it as dynamic as possible but you can still find a bit of redundancy in the client code for rach individual consistency