



[Show Submission Credentials](#)

P3. Introduction to consistency models [Optional]

Consistency Models and Real World Examples

16 days 3 hours left

 Introduction to Consistency Models

 Understanding the Strong Consistency using an example

 Conclusion

Introduction to Consistency Models

Introduction to Consistency Models

Information

Learning Objectives

This primer will encompass the following learning objectives:

1. Describe and compare the various consistency models
2. Discuss the advantages and disadvantages of different consistency models
3. Identify the appropriate consistency model for different use cases
4. Explain the implementation of strong consistency via the request ACK protocol

In computer science, consistency models are used in distributed systems, such as what you will be developing in Project 3.3. A distributed system supports a given consistency model if operations on data storage follow specific rules. The data consistency model specifies a

contract between programmer and system, wherein the system guarantees that if the programmer follows the rules, memory will be consistent and the results of memory operations will be predictable.

Consistency Models in the Real World

In large globally distributed applications, the data for an application may be replicated across various locations across the globe. This is done for a number of reasons. Some of the reasons for doing this are listed below:

1. **Reliability/Availability:** Having a distributed data store ensures normal operation in case of single node failures.
2. **Performance:** Having multiple replicas for the application data can reduce the potential of one data store being overloaded with requests, which could become a potential bottleneck to the application.
3. **Latency:** Having multiple replicas for the same data can help in reducing access time for requests coming from different geographic locations, as the data can be served from the nearest datastore.

Most web-scale applications these days are globally distributed. For example, Facebook hosts its application servers and data across several data centers in order to reduce the latency and access time (which in turn increases usability of the application) for users from across different geographic locations. A Facebook user from anywhere in Europe may be served a page from the application running in their Dublin datacenter whereas the users from the west coast of the United States may be served from the application server hosted in Silicon Valley.

Having multiple copies of the same data across different datastores that are globally distributed for the same application raises several design questions. Some important questions for developers to ask are:

1. What is the target performance (in terms of throughput and latency) that users in different parts of the world should experience?
2. How many replicas should there be, and where should they be located?
3. To which replica should a particular read be directed, and based on what parameters?
4. To which replica should a particular write be directed, and based on what parameters?
5. How should the change caused by a single write be cascaded to all replicas?
6. What level of locking should be used to support a particular transaction?
7. How often does the data between the various datastore replicas need to be synchronized?

Different application requirements might require different levels of consistency in the replicas. In the next section, we describe the different levels of consistency that are possible in a distributed key-value store. Different levels of consistency can be employed, however, there is a trade-off between how fresh the data ought to be in the replicas and an application's performance requirements. For example, if an application deals with important information such as bank balances, then it may choose to employ strict or strong consistency, sacrificing performance in exchange for the guarantee that all replicas will be consistent at any point in time. You will encounter applications of different consistency levels.

An Example of Inconsistency

Consider the following setup with three datacenter nodes and three clients. Client 1 reads from Datacenter 1, Client 2 from Datacenter 2 and Client 3 from Datacenter 3. The data is replicated across the three datacenter instances. Any update occurring at a Datacenter node has to be reflected in the other two Datacenter nodes in order to keep the data consistent.

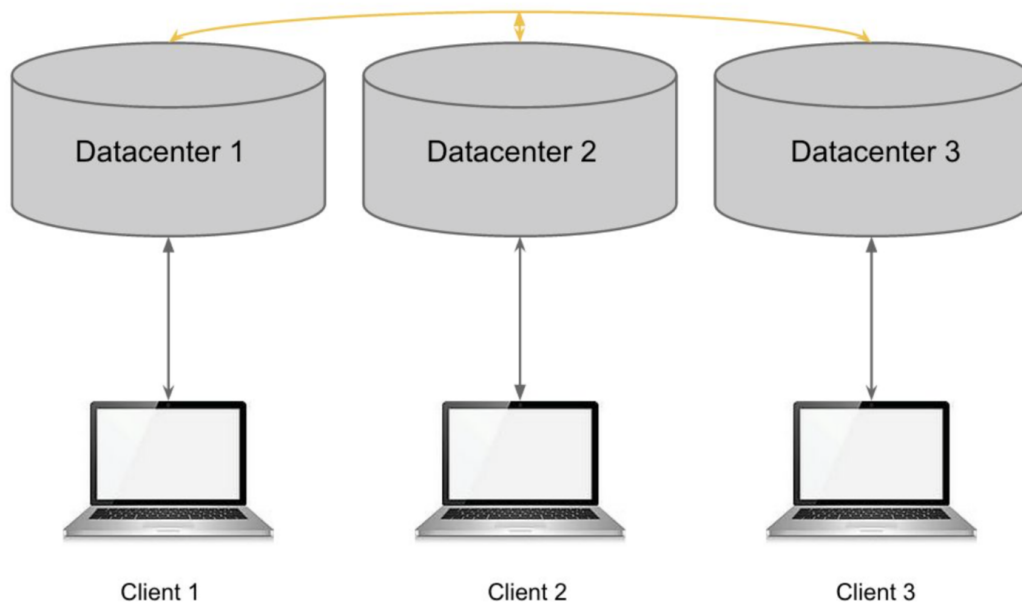


Figure 1: The architecture of the distributed datastore

Let a particular record be denoted by X . Its current value is 1 in all datacenters. Now, the following operations (shown in Figure 2) occur:

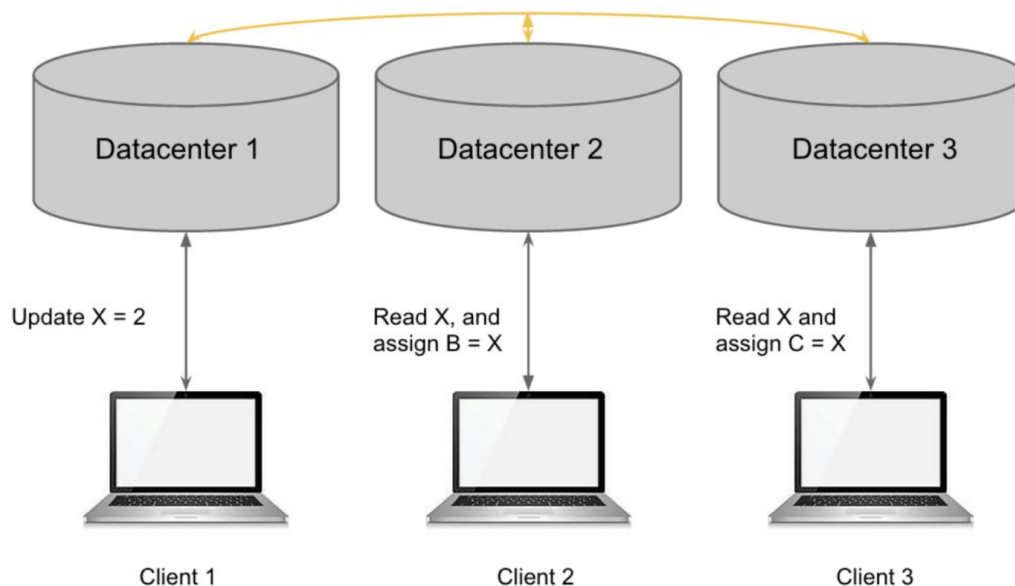


Figure 2: Possible inconsistency in different datacenters

Let's say that client 1 performs the update $X = 2$ at timestamp 1. Assuming that the two clients read X at timestamp 2 (i.e., after the update $X=2$ has been completed on datacenter 1), what are the values of B and C ? Turns out, B and C can be any of 1 or 2. The answer depends on the consistency model implemented by the system designers.

Consistency Models

Ordered from the strongest to the weakest, these are the various consistency models that we will be discussing:

1. **Strict**
2. **Strong (Linearizability)**
3. **Sequential**
4. **Causal**
5. **Eventual**

In the following sections, we will introduce the definition of each consistency model along with example scenarios. After the introduction, we will summarize the comparison between consistency models and their use cases.

Strict Consistency

In strict consistency, each operation must be stamped with **an absolute global time** and all operations must be **executed in the order of their timestamps**. Further, each read must get the latest written value. However, in a distributed system, it is hard to **synchronize clocks to be exactly equal**, and the time between instructions can be less than the time taken to communicate with the other processor(s)/node(s) to let them know about the issuing of an operation. Thus, strict consistency is hard to achieve on multiprocessors, let alone distributed datacenters. **We do not have any real world applications that implement strict consistency.**

Although Strong consistency is the next strongest form of consistency (after strict), we look at sequential consistency first, as it will make it easier to explain.

Sequential Consistency

Sequential consistency is achieved when **all operations were executed in some sequential order and all replicas see the operations in the exact same order**. Hence, all datacenters must see the operations in the same order, and any write occurring at a datacenter node must be instantly visible at the other datacenter nodes. To achieve this, operations for the object being updated must be locked until the updates have been made across all replicas.

As shown in the following figure, while the value of X is being updated at Datacenter 1, the clients reading from Datacenters 2 and 3 are locked from accessing X. They cannot read or write to object X, but they can read/write to other objects in those datacenters. Once the value of X has been updated across all datacenters, Clients 2 and 3 can read the latest updated value from their respective datacenters.

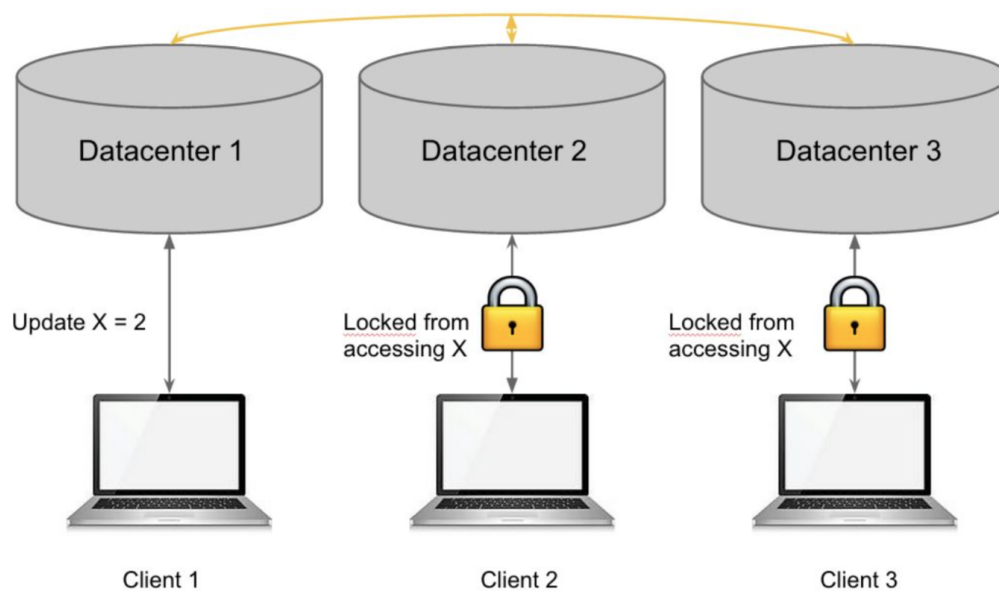


Figure 3: When an update is going on for an object X, the other clients are prevented from reading or writing the value of X across all data centers. This way, no client will receive stale data (old value of X)

Strong Consistency (Linearizability)

Strong consistency is sequential consistency with an additional requirement. In sequential consistency, every operation had a timestamp associated with it. This timestamp can be obtained through a vector clock (https://en.wikipedia.org/wiki/Vector_clock), **in which two timestamps can be concurrent and there may not be a global ordering**. The timestamps of two operations can be compared in order to determine whether one operation occurred before the other or if the two operations were concurrent. If two operations were concurrent, sequential consistency gives you the liberty to order those arbitrarily as long as the same order is seen by all nodes in the distributed system. In strong consistency, however, each operation **receives a global timestamp sometime during its execution (For example, when your server receives this operation)**, and all operations must be ordered according to their timestamps. This is in contrast to strict consistency in which **the timestamp is always the exact global time when the operation was issued**.

Thus, the key difference between strong consistency (linearizability) and sequential consistency is that, sequential consistency ensures a unique order of operations in order of timestamp with **only partial global ordering** because some operations can be concurrent, whereas strong consistency ensures operations in order of time with **full global ordering**. Thus, sequential consistency allows the system to interleave operations coming from different nodes as long as the order from each node is maintained. In linearizability, the interleaving of operations across clients is limited by the wall clock time.

Strong Consistency in the Real World: Bank Database

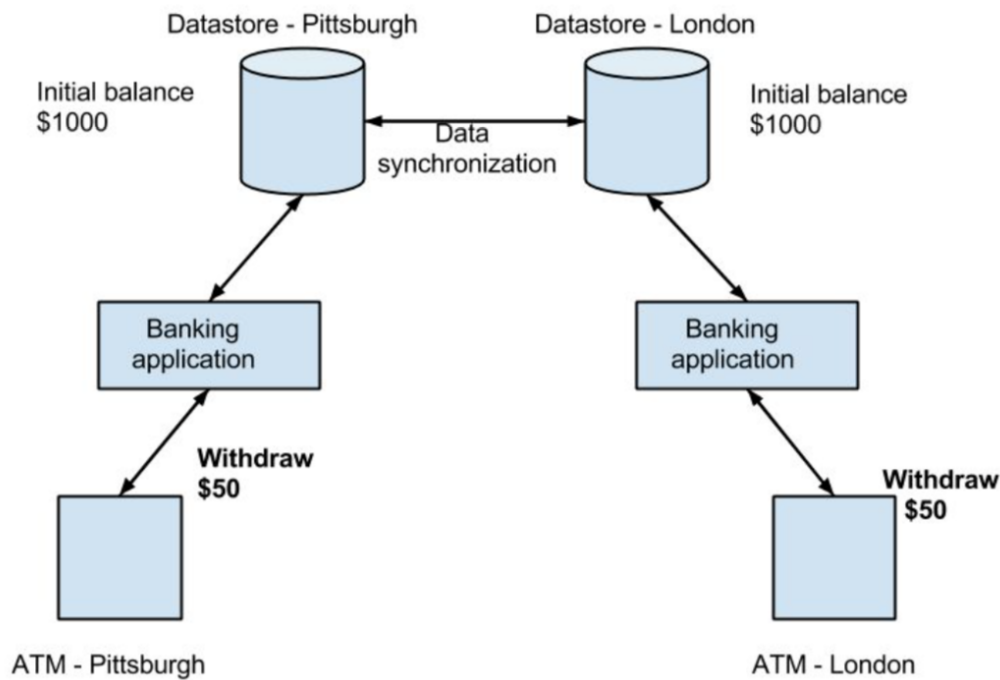


Figure 4: A banking system's datastore.

The diagram above shows a scenario where strong consistency needs to be enforced. Assume that a banking application has decided to distribute its datastores across various locations in order to serve its customers effectively.

In this scenario, let's say Joe and his wife Jane hold a shared account in the bank (which currently has \$1000 balance). If both of them try to withdraw money from the shared account simultaneously from two different locations as shown in the figure above, there is a possibility for an inconsistency as the two updates can deduct the amount (\$50) twice respectively in the replicas where the requests came, but not \$100 overall. This is not an acceptable situation for a banking application. In order to prevent this, the banking application must ensure strong consistency across its datastores as it ensures that only one operation can update all the replicas at a time.

If a user is currently trying to update the datastore (by withdrawal or deposit), the banking system should block all other users from updating this specific user's data in all its datastores. The next user is allowed to modify the datastore only once all the datastores are made consistent from the previous transaction. Although this is a feasible solution, it adds a lot of latency to each update. Not all applications might require strong consistency.

Causal Consistency

Causal consistency is a slightly weaker form compared to sequential consistency. Causal consistency refers to the level of consistency where the writes can be seen by different clients in different order, **except for causally related writes**. The order refers to the order of operations that are requested to the distributed datastore. Two writes are said to be "causally" related if they are dependent on each other in some way. For example, operation 1 is PUT(A, 2) and operation 2 is GET(A). These two operations can be causally dependent, since

the result of the second operation depends on the first. If two writes are causally related, then these writes need to be seen in the same order in all the clients. Replicas can be updated periodically or based on some predefined trigger. The causal consistency model provides much better performance over the strong consistency model, as all the datastores will not be blocked completely for each and every operation.

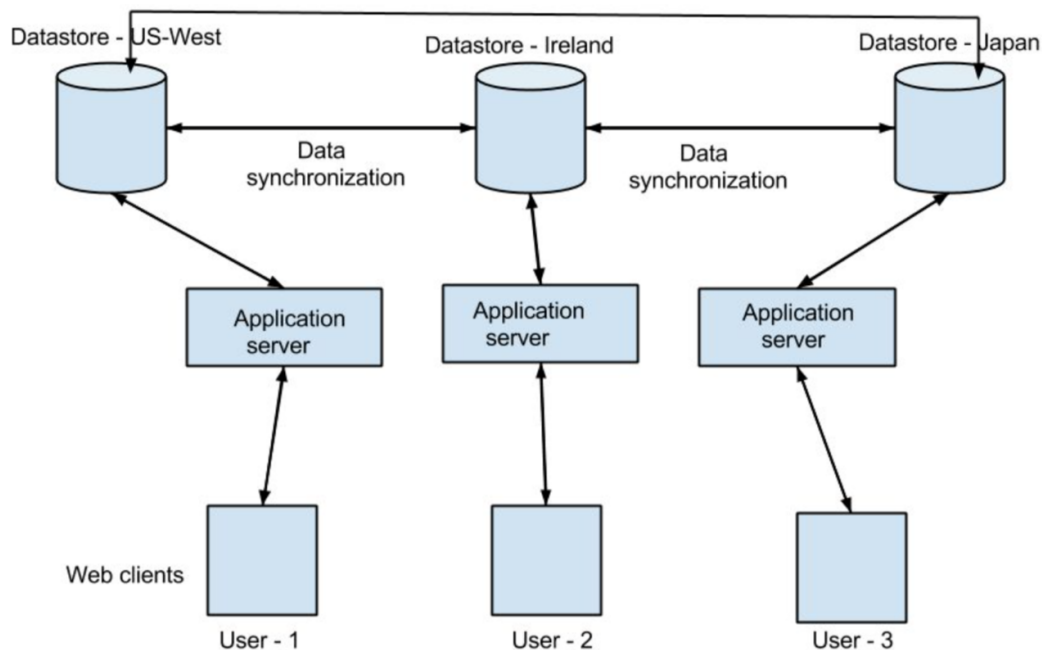


Figure 5: A social network's (InstaSnap) datastore

Causal Consistency in the Real World: Facebook Post Comments

Facebook is the largest social network in the world. People can share posts, make friends, upload photos and comment others' posts or comments. It stores substantial data related to user information. Some of this information might be required to be protected under strong consistency.

But for other kinds of social data, such as messages on posts, a weaker consistency model may be sufficient. For example, when people comment on others' post, they will tolerate the re-order of causally unrelated messages. But the reply comment should always be after the comment it replies to.

When user A makes a comment and user B replies to that comment, there is a causal order between the two comments. Not only A and B, but also all other people should see that A's post is prior to B's post. Unrelated posts, however, can be reordered since there is no causality between them.

Consider this stream of posts:

Bad luck! I was dropped at the wrong stop by the bus.
[a few minutes later] Lucky! An uber driver helped me out.
[reply from a friend] Nice driver!

It looks a little weird if what shows up on someone else's screen is:

Bad luck! I was dropped at the wrong stop by the bus.
[reply from a friend] Nice driver!

There are even better examples, widely used, when talking about access control:

[Block my boss from my post lists]
[Posts]: I hate my boss and my job.

Well, if causal consistency does not apply correctly here, you will get into a big trouble.

Consider the example of a distributed photo sharing social network (InstaSnap) application which employs causal consistency in its datastore. InstaSnap values low latency over data staleness. Different users using the social network may see different data. For example, if user 1 updates a particular photo on the datastore in USWest, user 2 may not get the update in the datastore in Japan at the end of the operation as the local datastore may not be updated immediately as in the case of strong consistency model. However, all the users should see the order of comments on the photo in the same order (Ordering for comments should be strictly ensured, and the update of comments coming from across the globe can be considered causally related). A user may see the photo with all the comments in the correct order. However, with the causal dependency defined by InstaSnap, a user will never see a photo with comments in a wrong order.

Eventual Consistency

Eventual consistency is one of the most widely used consistency models in the real world.

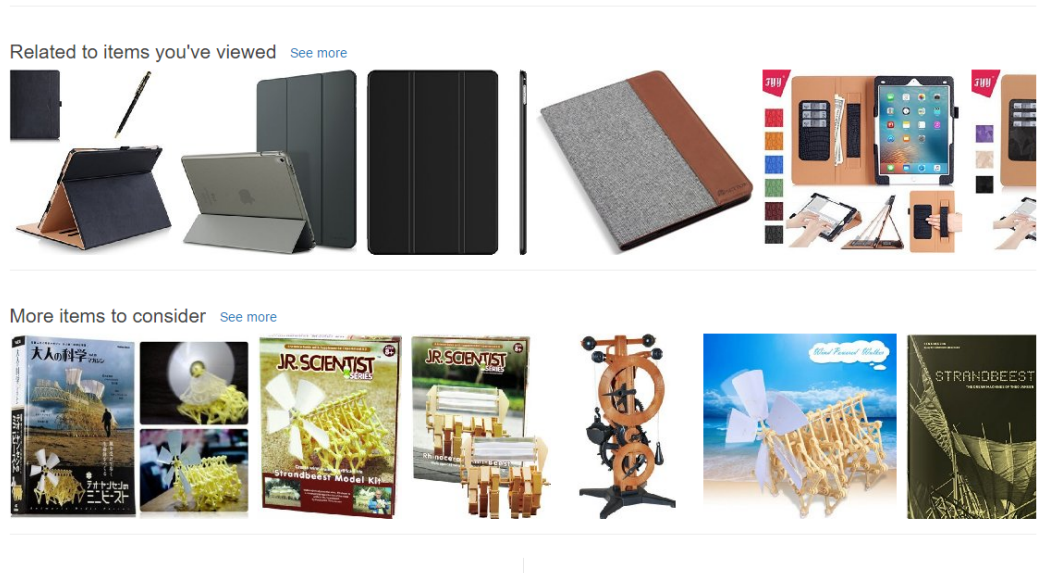
For some applications, even Causal Consistency is not necessary and eventual consistency is a consideration. The requirement to achieve eventual consistency is that **operations are executed as they are received by the datacenters**. This relaxes the requirement of having to lock all datastores when performing an operation, and further removes the requirement that each datacenter carries out each update request in the order they are received. While this provides higher throughput due to the lack of blocked requests, it comes at a cost of possible inconsistency amongst the replicas, especially during periods of high traffic. The only consistency guarantee that eventual consistency provides is that **after some period of time, and the operations have slowed down, the replicas will have the last updated value and be consistent again**.

As an example of eventual consistency, geo-replicated key-value stores each update in the order that write requests arrive at the replica, with only one requirement: the newly received write operation must have a greater timestamp than the write operation that updated the currently stored value. If the timestamp of a new write operation is less than the timestamp of the current stored value, then that means that the new operation contains a stale value, and should be discarded. This ensures that older write requests will not overwrite newer ones, and still provides the same high throughput benefit. This means that updates on a particular key may have the different ordering than updates on the same key at a different replica, due to delays in communication between regions. As a result, the clients who read the value in between these operations may read improperly ordered values or stale data. However, what eventual consistency can guarantee is given a long enough period of time of slow traffic, the latest value associated with each key will be consistent again because the replicas will never overwrite the latest value.

NOTE: It becomes crucial to decide the level of consistency in an application that uses a distributed datastore, because this will have direct implications on the performance and speed of the application accessing the data store. If your application requires that all clients read the same data at any point in time, then the system has to use a strong consistency

model. Using a strong consistency model can reduce throughput as the datastores need to be locked each time an update operation is performed. Using a causal or eventual consistency model can increase throughput of the operations, but at the cost of the clients receiving stale or unordered data.

Eventual Consistency in the Real World: (My) Amazon recommendations



Amazon is the largest online shopping company in the US, and based on large-scale machine learning algorithm, Amazon is able to provide precise and customized recommendations to the end users. The input data of this recommendation is mainly the items one user has bought or viewed previously.

It is impossible and also unnecessary to use any strong consistency model here, since we do not need the recommendation system to immediately update per new user behavior. We could tolerate delays and re-orders because these operations, during a small time period, do not influence the recommendation result in a time-sensitive manner, and we can revise our data and show the result to the user in the future. Sooner is usually better, however, not critical.

Here, **Amazon introduces an extra layer of cache. The cache implements the eventual consistency model.** There is no lock on updating this cache, and thus, Amazon is able to scale to selling millions of products while still generating pages containing recommendation at the same time.

The following video should refresh some of the concepts discussed above:

Consistency Models



Comparison Between Consistency Models and Use Cases

Strict Consistency

- Difference compared to the previous model: None. This model is the strongest model which defines the theoretical limits of consistency against which other models can be evaluated.
- Use Cases: Almost impossible to achieve since it's hard to have a globally synchronized clock in distributed systems.

Strong Consistency (Linearizability)

- Difference compared to the previous model: In the strict consistency model, **the timestamp is always the exact global time when the operation was issued by the client.** However, in the strong consistency model, **the timestamp is assigned when one of the servers receives the operation.**
- Use Cases: Bank account and flight ticket booking systems. When a ticket is booked, the system should prevent other users from buying it. Also, subsequent users are notified that the ticket is unavailable. In the following example, there are five operations ordered by their global timestamps.

Global Timestamp	T1	T2	T3	T4	T5
Operation	Check_Ticket(U1)	Check_Ticket(U2)	Buy_Ticket(U1)	Buy_Ticket(U2)	Check_Ticket(U3)

In this example, there are five operations here with different global timestamps. Both Check_Ticket(U1) and Check_Ticket(U2) will return true since ticket is available. However, since user1 purchases the ticket before user2, the system does not allow user2 to buy it(Buy_Ticket(U2)). Here are two examples that do not meet strong consistency: If Buy_Ticket(U2) were executed before Buy_Ticket(U1), user2 will probably own the ticket instead of user1. If Check_Ticket(U3) at T = 5 was processed before Buy_Ticket(U1), users3 would get the wrong information.

Sequential Consistency

- Difference compared to the previous model: **Two timestamps can be concurrent and there may not be a global ordering.** The timestamps of two operations can be compared in order to determine whether one operation occurred before the other or if the two operations were concurrent. If two operations were concurrent, sequential consistency orders those arbitrarily as long as the same order is seen by all nodes in the distributed system.
- Use Cases: Online game server, such as FPP (first person shooter) games. The server can decide which operation gets processed first, if two operations arrive at the same time.

However, all players should see the same order, meaning that all replicas have the same sequence of operations.

Causal Consistency

- Difference compared to the previous model: In sequential consistency, all replicas see the operations in the exact same order. However, operations can be seen in a different order in causal consistency if they are not causally related.
- Difference compared to **strong consistency**: Operations don't have a global ordering and they could be out of order on different servers. On the other hand, causal consistency has a much better performance than strong consistency since there is less locking/blocking. Locking/blocking is necessary where the sequence of operations have a causality relationship.
- Use Cases: Social media comments.

Eventual Consistency

- Difference compared to previous model: Replicas are inconsistent for a period of time.
- Use Cases: CDN (Content delivery network).

An Example of Consistency Models

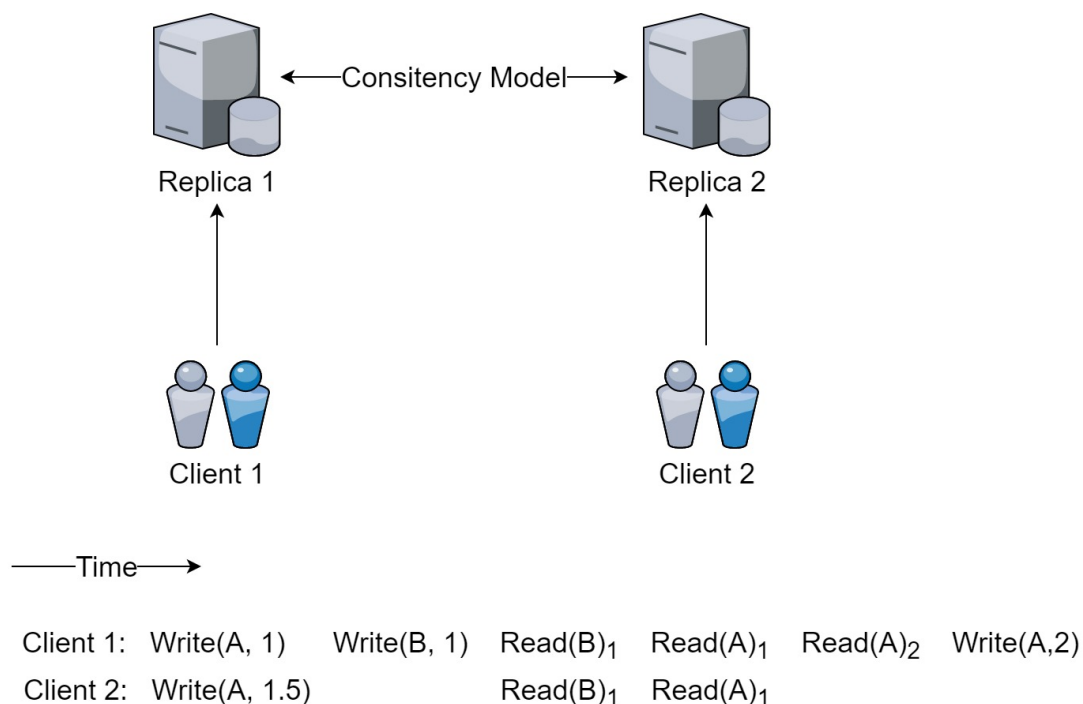


Figure 6: An Example of Consistency Models

In this example, there are two replicas in two different regions. Client1 and Client2 are sending requests to each of these replicas. They can send either read or write requests: write(A, 1) means that client is sending a request to set the value of A to 1, and read(A) means that client is reading the value of A. You can see the sequence of client sending requests to replicas ordered by time, and later we will show you, in different consistency models, the order of operations in these two replicas. Note that all reads are processed by a local replica, and we use Read(A)₁ and Read(A)₂ to distinguish between two consecutive read operations.

Strong Consistency

In Strong consistency, there is a global order of the operations. Here, we assumed that `write(A,1.5)` has a slightly larger global timestamp compared to `write(A,1)`.

Replica 1	Write(A,1) Write(A,1.5) Write(B,1) Read(B) ₁ Read(A) ₁ Read(A) ₂ Write(A,2)
Replica 2	Write(A,1) Write(A,1.5) Write(B,1) Read(B) ₁ Read(A) ₁ Write(A,2)

Figure 7: Strong Consistency Model Order of Operation

Since the strong consistency model requires a full global ordering, `write(A,1)` should be processed before `write(A, 1.5)` because `write(A,1)` has an earlier timestamp.

Sequential Consistency

Replica 1	Write(A,1) Write(A,1.5) Write(B,1) Read(B) ₁ Read(A) ₁ Read(A) ₂ Write(A,2)
Replica 2	Write(A,1) Write(A,1.5) Write(B,1) Read(B) ₁ Read(A) ₁ Write(A,2)

OR

Replica 1	Write(A,1.5) Write(A,1) Write(B,1) Read(B) ₁ Read(A) ₁ Read(A) ₂ Write(A,2)
Replica 2	Write(A,1.5) Write(A,1) Write(B,1) Read(B) ₁ Read(A) ₁ Write(A,2)

Figure 8: Sequential Consistency Model Order of Operation

In the sequential consistency model, **two operations can have the same timestamp and there may not be a full global ordering**. Let's say `Write(A,1)` and `Write(A, 1.5)` have the same logical timestamp. In this scenario, the server can arbitrarily determine the ordering between `write(A,1.5)` and `write(A,1)`, as long as **all replicas see the operations in the exact same order**. Here, both replicas see the same ordering: `write(A,1) write(A,1.5)` or `write(A,1.5) write(A,1)`.

Causal Consistency

Replica 1	Write(A,1) Write(A,1.5) Read(A) ₂ Read(A) ₁ Write(A,2) Write(B,1) Read(B) ₁
Replica 2	Write(A,1) Write(A,1.5) Read(A) ₁ Write(A,2) Write(B,1) Read(B) ₁

Figure 9: Causal Consistency Model Order of Operation

In this example, Write(A, 1) and Write(B,1) are not causally related. The ordering of these two operations does not affect the value going to be read by the client, as long as they happen before the read. Therefore, Write(A,1) and Write(B,1) can have random ordering, as long as they happen before their causally related reads.

Eventual Consistency

Replica 1	Write(A,1) Write(B,1) Read(B) ₁ Read(A) ₁ Read(A) ₂ Write(A,2)
Replica 2	Write(A,1.5) Read(B) ₁ Read(A) ₁

Figure 10: Eventual Consistency Model Order of Operation

In the eventual consistency model, operations are executed **immediately** as they arrive at the servers. There is no guarantee on when the replicas will be consistent again. However, the application programmer may decide “when” to synchronize two replicas, i.e., make two replicas consistent. For example, per application’s requirement, two replicas may need to synchronize every 5 minutes to make data across replicas consistent. Both replicas will have the last updated value of A as 2 eventually.

Understanding the Strong Consistency using an example

Understanding the Strong Consistency Implementation using an example

In this example, we will focus on **distributed key-value stores** which are located in different geographical locations, and store the data required for applications, often in-memory. Distributed key-value stores, especially when distributed to multiple servers globally, could reduce the latency between the servers and clients by reducing the geographical distance between the client and the closest server. These key-value stores are considered to be a type of NoSQL storage systems, as they do not have the full relational capabilities of systems such as MySQL. Key-value stores are also commonly used in caching systems as a way to store recent results for a given key. Unlike SQL, a key-value store supports two basic operations:

1. PUT requests, which "puts" value for a given key into the database.
2. GET requests, which "gets" the value associated with a key in the database.

Global Timestamp

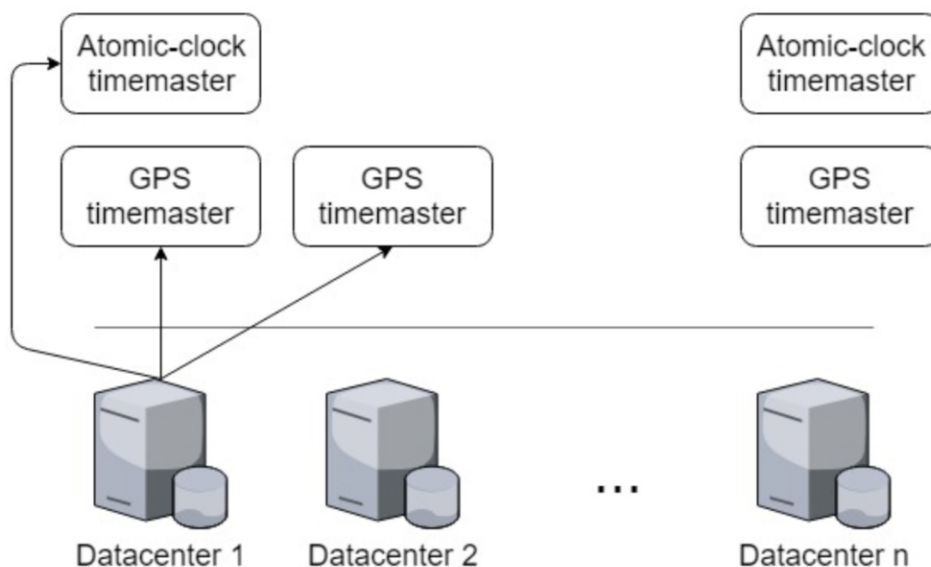
One of the challenges of distributed data stores under the strong consistency model is determining the global timestamp of each operation. This is crucial to order clients' requests from different regions. Imagine the following scenario, where a flight ticket booking company has two replicas in the U.S. East and U.S. West in order to improve the user experience. This means that requests issued by clients in U.S. East will be handled by the replica in the U.S. East, and others will be handled by the U.S. West replica. User A in Silicon Valley and user B in Pittsburgh try to buy the same airplane ticket. They click the purchase button at approximately the same time, and guess who can get the ticket? Despite other factors such as

network delays, what really matters is the timestamp assigned by the servers of these two requests. The request with a smaller timestamp indicates that it will be processed earlier than the other one and that user will be able to purchase the ticket.

Using the local timestamp is not a viable solution, as the replicas in U.S. East and U.S. West have their own coordinators (see definition below), the computer clock on these two machines are not guaranteed to be in sync with each other, and a request that arrives earlier in U.S East coordinator might be assigned a larger local timestamp (scheduled later) than another request arrives later at the U.S. West coordinator. Therefore, a **global timestamp** for each operation is needed to determine the global ordering.

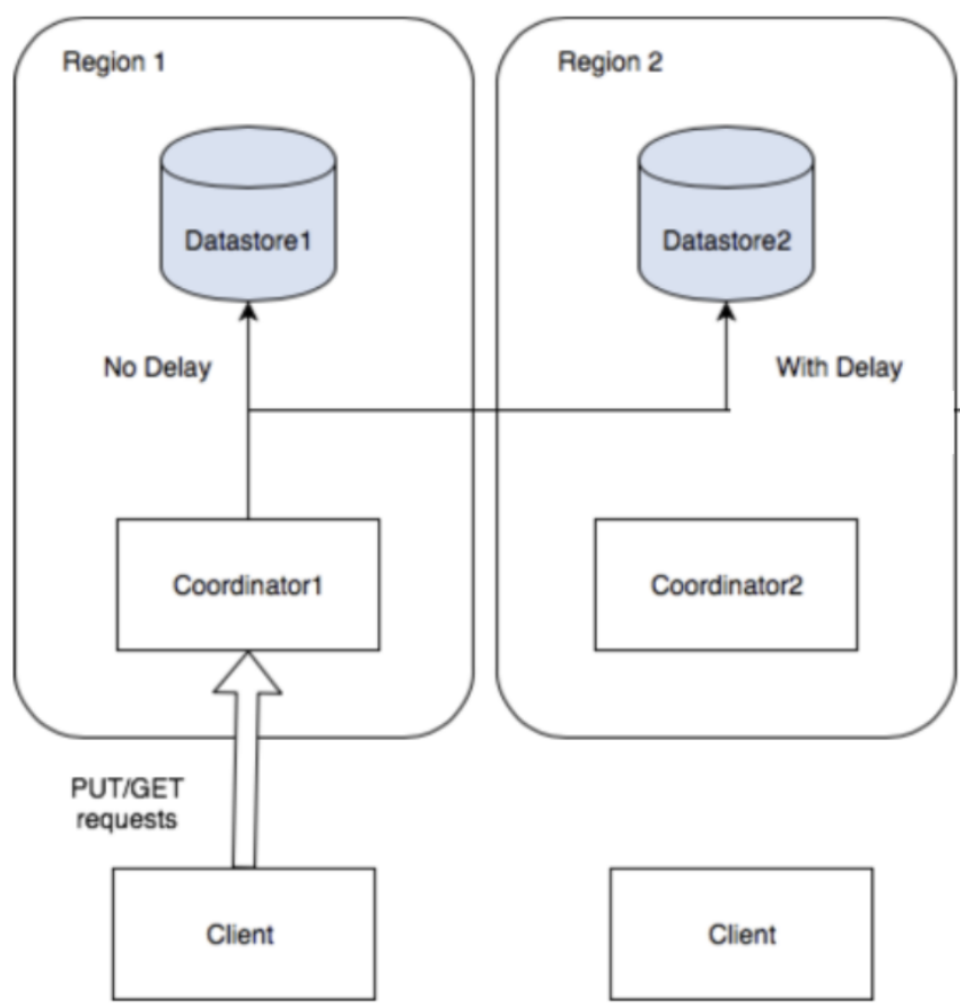
Real-life distributed systems such as Google Spanner (<https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/39966.pdf>) deploy a set of time master machines, which are equipped with GPS receivers (https://en.wikipedia.org/wiki/Global_Positioning_System) or Atomic clocks (https://en.wikipedia.org/wiki/Atomic_clock) in each regional data center. These machines are compared regularly to make sure they are synchronized with each other. Upon receiving a request, the regional coordinator asks multiple time masters for a global timestamp, which will make the system robust if there's any time-master failure. Then it will assign a global timestamp to the request.

Time masters in real-life distributed systems



Scenario

We will consider the use case where the data is replicated in two data stores, where each data store is in a different global region. The communication between each region of a data store is not instantaneous, and has a non-trivial delay. Also, because the distance between each data store is different, the delays are different. In this example, we will employ a coordinator for each data store in each region as shown below.



The definition of these instances are below:

Table 1: Instance definitions

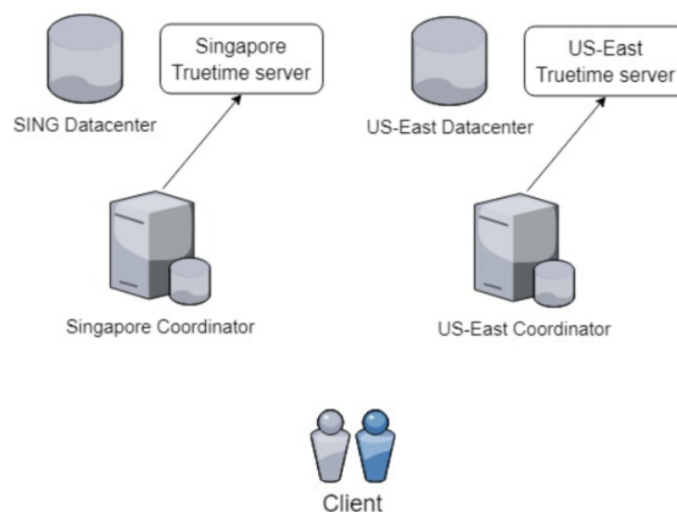
Coordinator	A Coordinator acts as a distributed frontend located close to the clients in that region. Clients interact with the coordinators by sending PUT and GET requests. The Coordinators route the incoming requests to the data store and maintain the required consistency for the application.
Data Store	A fast in-memory key-value data store. A data store supports the basic operations of PUT and GET requests to specific keys. Clients don't interact with the Data Store directly, but via Coordinators.
TrueTime Server	Maintains global time. Upon receiving a request, a coordinator asks the TrueTime Server for a global timestamp. TrueTime Server assigns a global timestamp to the request. Please see the detailed description below.
Clients	Clients who wish to perform GET or PUT request for specific keys on the distributed data store.

Request ACK protocol

One approach to enforce the **strong consistency** is Request-Ack protocol. This approach is based on the notion of a global virtual priority queue for the ordering of requests on the key level. The queue is ordered by the timestamp attached to the request by TrueTime Server. Each node maintains a copy of this global queue so that the replicas store data consistently. Let's go through an example of the Request-Ack protocol.

The client sends all requests to one of the coordinators in their geographical region. When a coordinator node receives a PUT request to update/insert a key, the coordinator sends a LOCK REQUEST message to every other coordinator node. The LOCK REQUEST message contains the timestamp of the PUT request. When the other coordinators receive this request, each coordinator adds the request to its priority queue and returns a LOCK ACK response to the initial coordinator. The initial coordinator pauses until it receives the LOCK ACK response from all other coordinators. This ensures that the same PUT request has been queued on all of the coordinators and that the queues are consistent at least with respect to this PUT request. Once the initial coordinator receives all of the LOCK ACK responses, the initial coordinator is sure that the request has been added in all other coordinator's queue. At this time, if the current request is not on the front of the priority queue, the requesting coordinator needs to wait for its turn. When the request is at the front of the queue, the coordinator is sure that the key has been locked on all the coordinators with respect to this request and it can start updating the key. The initial coordinator then sends a PUT REQUEST to all other coordinators. Once other coordinators receive the PUT REQUEST, they update the value of the key and send a PUT ACK back to the requesting coordinator. After executing the PUT REQUEST from requesting coordinator and sending the PUT ACK, these coordinators look into its priority queue and attempt to run the next request if available. The requesting coordinator will wait for all PUT ACK, and once it receives all PUT ACK it returns a response to the client that the PUT request is successful.

Let's learn the implementation using an example with a cross-region replicated data store with double replicas. The coordinators and data centers are in the following two regions (U.S. East and Singapore). If we were to follow the architecture of Google Spanner, here's what we will have: the Truetime server is similar to the time master, and Google calls the above timekeeping technology TrueTime. When a coordinator receives a request, it asks its regional Truetime server for a current timestamp and attaches it to the request.

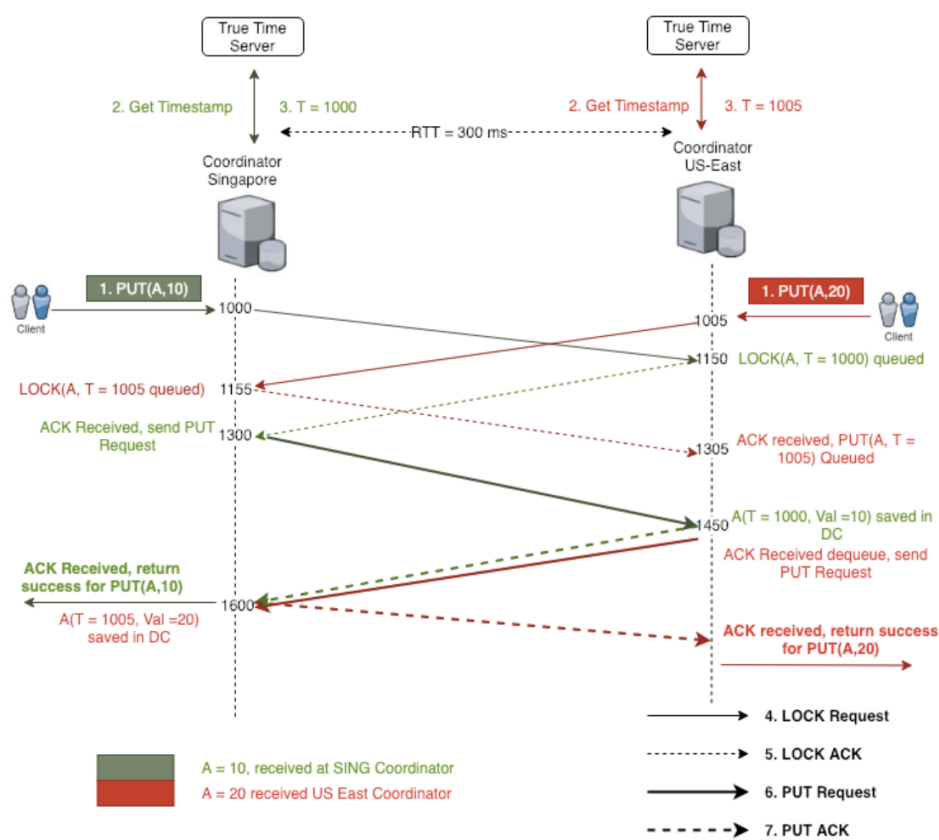


Also, the communication delays between these coordinators vary depending on which regions they are communicating with. **To simplify the problem**, we assume that the two regions that we are using have the following **fixed** communication delays:

Table 2: Communication Delay

Region	Communication Delay (RTT)	Region
us-east	300ms	Singapore

Let’s consider a case when a client in the Singapore region sends a PUT Request(A,10) (update key A with value 10) to the Singapore coordinator. With a slight time difference, another client in the us-east region sends another PUT request for the same key with value 20 (A, 20). The diagram below explains the concrete steps to complete these requests:



In this example, we have made the following assumptions: 1. There is no processing delay on any Coordinator, Data stores or Truetime Servers. However, in practical systems, this isn’t true. There is always some overhead involved in processing the request. 2. We are assuming the timeline starts at global timestamp 1000. In a practical scenario, it will be the actual timestamp when the request arrives at the TrueTime Server. Also, we are assuming that there is no delay while processing the request, which isn’t the case in practical scenarios. 3. We are assuming that there is no communication delay between the Truetime Server and the regional Coordinators.

Let’s try to understand the steps from the above timeline.

1. Client 1 sends a PUT(A, 10) request, which means put value 10 for key A, to the Singapore coordinator. The coordinator requests the TrueTime Server for a timestamp and receives 1000 as the timestamp.
2. The Singapore coordinator adds the request to its queue, and since its queue is empty, it sends the LOCK REQUEST to the us-east coordinator with timestamp 1000 and waits for the LOCK ACK from the us-east coordinator.
3. Client 2 sends a PUT(A, 20) request to the us-east coordinator. The us-east coordinator queries the TrueTime Server for timestamp and receives 1005 as the timestamp.
4. The us-east coordinator also adds the request to its queue and since its queue is empty, it sends the LOCK REQUEST to the Singapore coordinator with timestamp 1005 and waits for the LOCK ACK from Singapore coordinator.
5. The us-east coordinator receives the LOCK REQUEST at time 1150 (Half of the RTT between them) and adds the request to its queue. Here since the lock request is carrying the lower timestamp, it will be at the head of the queue. It then sends the LOCK ACK back to the Singapore coordinator confirming that the request has been queued successfully.
6. The Singapore coordinator receives the LOCK REQUEST from the us-east coordinator at time 1155. It adds the request to its queue but since the timestamp 1005 is higher than the timestamp in the queue, this request will be queued after the already present request. It sends back the LOCK ACK to the us-east coordinator.
7. At time 1300, the Singapore coordinator receives LOCK ACK from the us-east coordinator. This confirms that the key has been queued on the us-east coordinator. At this point, since it has already received all ACKS it checks its queue to see if the request with T=1000 is at the front of the queue. Since it's at the front of the queue, it is sure that the key has been locked in all coordinator's queue, it sends the PUT REQUEST to the us-east coordinator at time 1300 and also applies the update to its data center simultaneously.
8. At time 1305, the us-east coordinator receives ACK from the Singapore coordinator. At this time since its request with T=1005 is not in front of the queue, it doesn't perform any operation.
9. At time 1450, the PUT REQUEST from the Singapore coordinator arrives at the us-east coordinator. The us-east coordinator applies the update to its data center and replies with PUT ACK. It also removes the request with timestamp 1000 from its queue and checks if there are any other requests queued in the queue. Since the PUT request with T=1005 will be at the front of the queue at us-east coordinator, it resumes the request. It sends the PUT REQUEST to Singapore coordinator.
10. The Singapore coordinator receives the PUT ACK at time 1600, confirming that the update has been applied to all datacenters. It removes the request from the queue and returns the success message to the client. It then checks its queue for any further PUT requests, but the next request is LOCK, hence it doesn't perform any operation.
11. The Singapore coordinator receives the LOCK REQUEST from the us-east coordinator at time 1600. It checks its queue to see if this request is at the head of the queue. Since it is at the head of the queue, it applies the update to its data center and replies with a LOCK ACK to the us-east data center. It also removes the request from its queue.
12. The us-east coordinator receives the LOCK ACK at time 1750 and then returns success to client 2. It removes this request from its queue.

Note: Similarly, if a GET request arrives at any of the coordinators, the coordinator adds the request to its queue and checks if any other LOCK or PUT requests with the lowest timestamp are waiting. If no requests are waiting, it first retrieves the value from its data store and then

removes the request from its queue. Otherwise, it adds the GET request to its queue. Please note that a GET request is only blocked by any LOCK or PUT requests; if there are only GET requests pending, they can be served immediately.

In the real world, another complexity to handle is that the network may experience failures. To simplify the problem we assume that there are no failure in the network. There are solutions to handle these failures which are out of the scope of this primer.

Conclusion

Conclusion

Congratulation, you have completed the **Introduction to Consistency Models** primer!