CMPSCI 677 Operating Systems **Spring 2019**

Lecture 17: March 29

Lecturer: Prashant Shenoy *Scribe:* ***Justin Alvin***

17.1 Consistency and Replication

Replication in distributed systems involves making redundant copies of resources, such as data or process, while ensuring that all the copies are identical, to improve reliability, fault-tolerance and performance of the system.

Types of Replication:

- *Data replication*: when the same data is stored on multiple storage devices
- *Computation replication*: when the same computing task is available to be executed on multiple servers

17.1.1 Why Replicate?

- Performance

  When our system is replicated on n machines, we can use all of the resources available to those n machines to process requests. Conversely, if there is no replication and we just have a single machine, we are limited to the capacity of that machine.

  For example, if we have just one web-server, it would have a certain capacity - i.e. requests it can serve per second. After reaching the limit, it will get saturated. By replicating it on multiple servers, we can increase the capacity of our application, so that it can serve more requests per second.

- Reliability

  If one of the replicas in our distributed system goes down, we want at least one to still be up to server requests and ensure that data is still be available. For instance, in a distributed system if one of the database servers crashes, and we have a replicated copy of the same data on another database server, then the data is safe. We can point our system to the second replica of the database and continue to access the data without any problems.

17.1.2 Replication Issues

We need to consider what additional work we will need to do to achieve replication. Our ultimate goal is to maintain consistency. For example if one copy of a file is modified, we want all replicas to make the same update and remain consistent. This tends to be complicated and expensive.

- When to replicate:

  We can replicate on-demand or statically.

- How many replicas to create:

  In terms of fault tolerance, we need at least two copies to tolerate one crash. To tolerate k crashes, need $k + 1$ copies. In general, it depends on what you are trying to do. This is also true in terms of performance. We have to consider how many machines we need to service a given workload.

- Where are replicas located:

  This matters in a geo-distributed replication. For example, if we are building a game with replicated servers, we want users to be as close to a server as possible to reduce latency. So where replicas are located is very important in this system.

17.2 CAP Theorem

The CAP theorem states that it is impossible for a distributed system to simultaneously provide more than two out of the following three guarantees:

Consistency (C): If there are multiple copies of a replicated data item, they are in sync and appear as a single, up-to-date item.

Availability (A): If there are node faults, they are tolerated. In other words, if a node fails, other nodes can take over its job(s) and the system will not go down.

Partition-tolerance (P): If there is a network fault, the system continues to function. In this case, the network is down but there are still nodes up.

17.2.1 CAP Theorem Examples

Consistency + Availability: Single database, cluster database, LDAP, xFS. They assume that the network is reliable so that messages do not get lost.

Consistency + Partition-tolerance: distributed database, distributed locking. They assume that the coordinator doesn't fail.

Availability + Partition-tolerance: Coda, Web caching, DNS. DNS update can take up to a few days to propagate.

17.2.2 CAP Properties

We should evaluate why consistency is hard to achieve if there is a network partition. Consider a scenario in which we have two replicas. When we make an update on one replica, to make our system consistent, it has to tell the other replica that the change occurred. If network is down, the two replicas can't communicate, so consistency doesn't happen. This means that there is an inherent tension between consistency and network failures. We can make similar arguments for any pair of properties.

Question: How does partition tolerance relate to consistency?

Answer: They dont go with one another. If you have perfect consistency, you have to assume that the network is up. If you want to tolerate network partitions, you will lose consistency. But local instances can still serve requests, even if the connection goes down.

17.2.3 NoSQL Systems and CAP

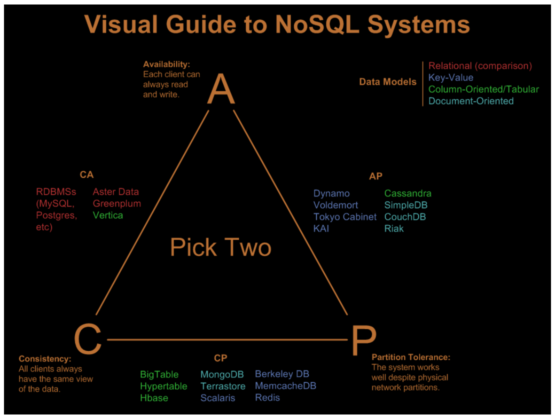


Figure 17.1: CAP in Database systems

These systems don't use a SQL database. Figure 17.1 shows some database systems and which properties they hold. Consistencies in the presence of network partitions are problematic.

The nodes in first half can talk to one another, and the nodes in second half can talk to one another, but the nodes from first half cannot talk to the nodes in second and there are clients able to talk to either one or both of those nodes.

In our case these are replicas. If there is an update on the node, that update can be propagate to other nodes, but since the network is partitioned it cannot communicate with the other nodes until the network is fixed. The system will be inconsistent if the messages are not flowing back and forth.

17.3 Object Replication

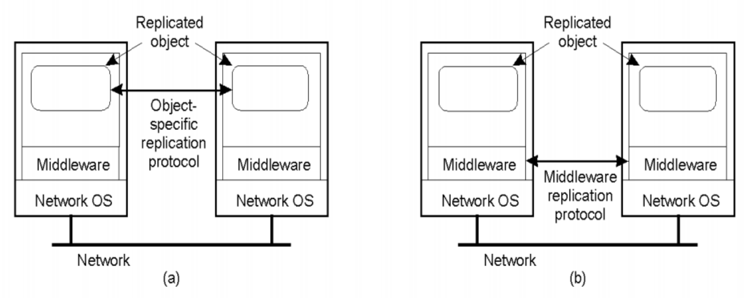


Figure 17.2: Two types of replication

Two ways to replicate (Figure 16.2):

1. Application does the replication and handles consistency (Figure 16.2(a)).

2. Middleware does the replication and handles consistency (Figure 16.2(b)).

17.3.1 Replication and Scaling

What kinds of consistency guarantees can we provide? We must first consider what kinds of overheads consistency imposes. Ideally we want to provide the strongest possible guarantee that we can still pay for.

For example, if file is replicated n times, when it is changed, we have to update all of the copies. Assume that R is the read request frequency and W is the write frequency. We must consider relative frequency of reads and writes.

- **Scenario 1**: Consider a file on a popular new site. This file may be read once every second but updated every 10 minutes, so $R >> W$. In this case, we are likely willing to pay the price for consistency because every write update will be read many times down the line.
- **Scenario 2**: On the other hand, consider a file on a blog that no one reads. The write frequency for this file may high but the read is not, so $W >> R$. To achieve consistency, we have do a lot of extra work, but get no benefit because there are very few reads that see updates.

17.4 Data-Centric Consistency Models

We can analyze different consistency models from the perspective of a data item, with the goal of always retrieving the most up-to-date version of the item.

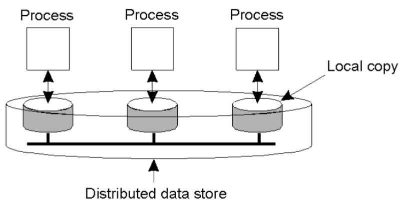


Figure 17.3: Data Centric Consistency Models

17.4.1 Strict Consistency

In strict consistency, all copies are always in sync. As soon as we make an update to a data item, we immediately update all its copies. In strict consistency, we will never see outdated info.

Strict consistency is very hard to implement. Once a copy is updated, messages must be sent to other replicas. The speed of this process is limited by the speed of light. So, if an update is made at time t and it takes ϵ time for the update message to get to machine M, it is possible for another request to reach machine M between time t and $\epsilon + t$. So we must account for network delays and assume that the message gets to each node in time, which is difficult to achieve in real system.

17.4.2 Sequential Consistency

Sequential consistency relaxes the requirements of strict consistency and is therefore weaker. If we make a sequence of operations on one machine, other replicas see them in some sequential order and execute them in such a way that preserves program order (how the initial machine actually executed them). We can decide on any interleaving so long as it preserves program order and everyone sees it in that order. We cannot have a scenario in which one node sees one order and another sees a different order.

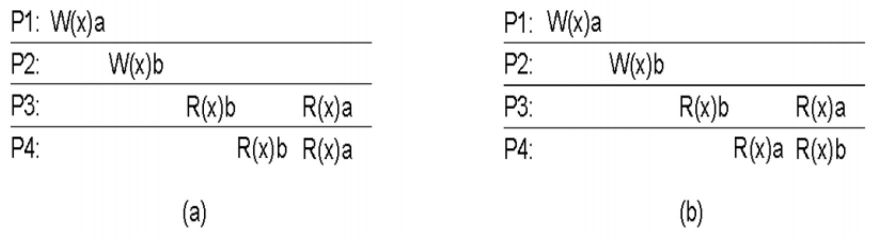


Figure 17.4: Sequential Consistency

In figure 17.4, lets say x is a web-page. Process $P1$ writes a to x and process $P2$ writes b to x. Process $P3$ reads x's value as b and then later reads it as a. If we had a global lock, we would know that $P1$ wrote it first and then $P2$. So, once $P3$ sees a it shouldn't see b. But since we do not have a synchronized clock, we don't really know if that is what has happened, because $P1$ and $P2$ did not communicate with each other and hence we don't know if they are concurrent events. So processes just agreed that $P1$'s write happened before $P2$'s write.

In 17.4(a) The processes agree on the order that $P2$ wrote before $P1$ and both $P3$ and $P4$ read in that order. Figure 17.4(b), $P3$ and $P4$ see in different orders and that is not allowed.

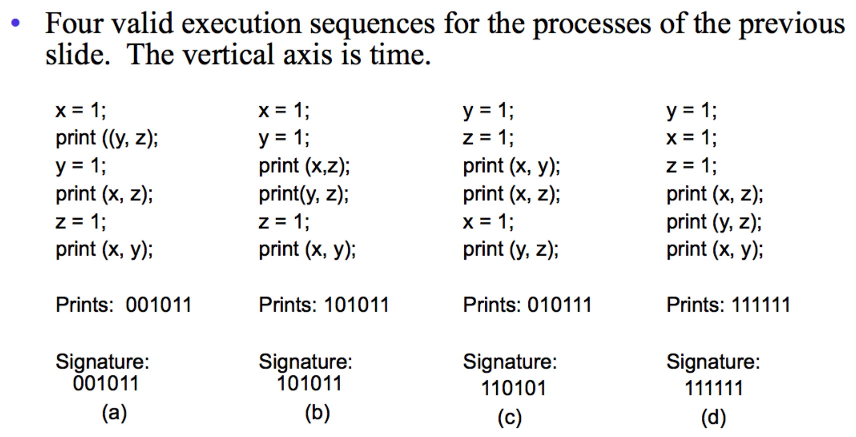
17.4.3 Linearizability

Linearizability assumes all of the requirements of sequential consistency but adds the requirement that if the timestamp of a write is less than another timestamp, interleaving has to preserve that order. So if two writes are related somehow, this must be preserved. Any interleaving is allowed so long as there is some relationship between timestamps. Linearizability is stronger than sequential consistency but still weaker than strict consistency.

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

Figure 17.5: Linearizability example

Figure 17.5 shows three processes. Each process writes one variable and reads variables written by the others. Thus, there is an implicit communication here about the ordering. The valid interleaving is shown in figure 17.6.



- Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

(a)	(b)	(c)	(d)
x = 1;	x = 1;	y = 1;	y = 1;
print ((y, z);	y = 1;	z = 1;	x = 1;
y = 1;	print (x,z);	print (x, y);	z = 1;
print (x, z);	print(y, z);	print (x, z);	print (x, z);
z = 1;	z = 1;	x = 1;	print (y, z);
print (x, y);	print (x, y);	print (y, z);	print (x, y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111
Signature: 001011	Signature: 101011	Signature: 110101	Signature: 111111

Figure 17.6: Valid interleaving for figure 17.5 conforming to linearizability

An invalid ordering would be when after assigning a value to a variable we still print a 0. Another scenario might be if we do not agree to program order.

17.4.4 Causal Consistency

In causal consistency, causally ordered writes must be seen in the same order by all processes. Causally ordered means that, for example, if P1 sends a message to P2, the operation order has to be preserved by interleaving because there was some communication. This doesn't happen in sequential consistency, which can always pick any order. In causal consistency, we can only pick any order if events are independent or concurrent.

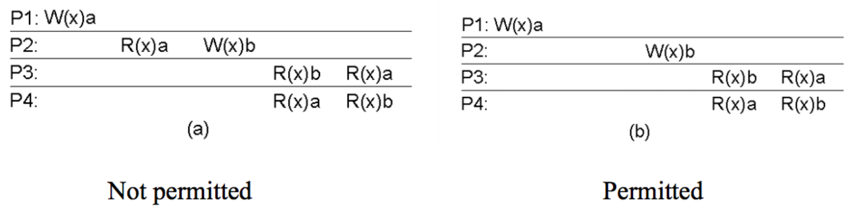


Figure 17.7: Causal Consistency

Question: Is causal consistency a stricter version of sequential consistency?

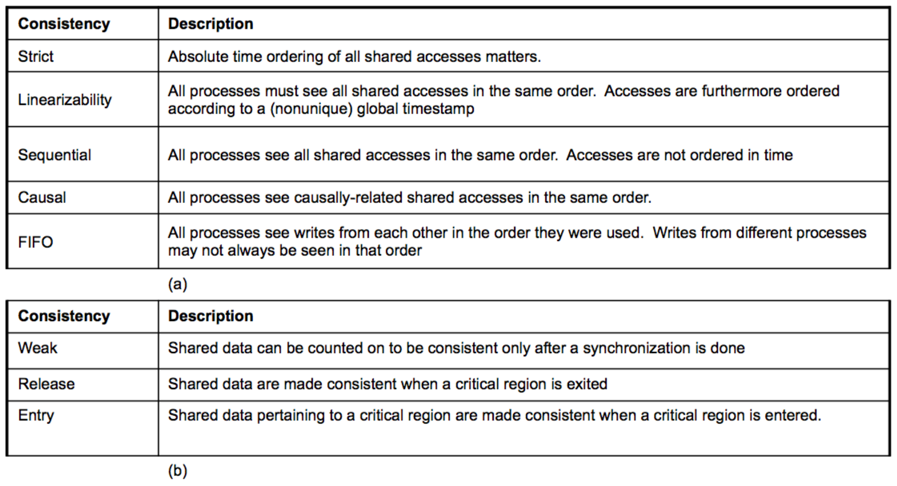
Answer: It is a weaker version because if writes are not ordered, you can actually have different orders.

17.4.5 FIFO Consistency

There are other consistency models, such as FIFO consistency, where we don't care about interleaving writes across machines. If one machine is making a series of updates, we respect that ordering on a per-machine basis. This is an even weaker version of other models.

17.4.6 Entry and Release Consistency

In entry consistency, we assume consistency at entry to critical sections but not after exit. Release consistency is the opposite, where shared data is made consistent after exiting a critical section.



Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

Figure 17.8: Consistencies (weaker as you go down)

Question: Is sequential consistency only seen from client's perspective?

Answer: Some machines are servers and some are clients. In sequential consistency, we don't make that distinction. So all machines will see the same order, whether they are clients or servers.

Question: Can you implement sequential consistency?

Answer: We are not talking about how to provide these guarantees, just the semantics. There are techniques to achieve sequential consistency, but they are fairly complicated.

Question: When we talk about the CAP theorem, which consistency model are we talking about?

Answer: We assume strict consistency for the CAP theorem.

17.5 Client-centric Consistency Models

Client-centric consistency models are written purely from the client perspective, without considering the server.

Monotonic Reads: Once a file is read, we can only see the same or a newer version, not an older version.

Monotonic Writes: Once a file changes, all replicas must be updated before any successive writes from the same process occur.

Read your writes: If we write something, we should always be able to read that update or something newer.

Writes follow reads: The writes after a read will occur on the same or more recent version of the data.

17.6 Eventual consistency

Eventual consistency is how many systems work. We make no guarantees and assume that everything gets propagated eventually. We dont consider how long this will take. The general principle is that if we stop making updates to files, eventually all of our replicas will converge. Our system will work exactly as expected so long as the user always accesses the same replica. Eventual consistency is easier to achieve and cheaper to implement than stricter forms of consistency.

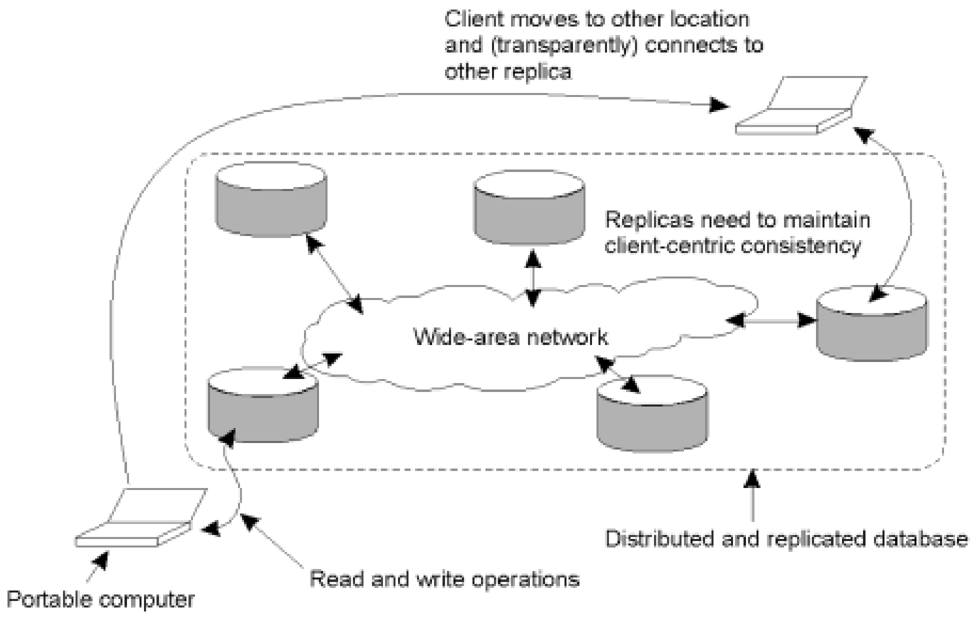

Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

Figure 17.8: Eventual Consistency

DNS is an example of a system that uses eventual consistency. We can see this if we change a password on one machine and then try to log in on another machine immediately. It is possible that the new password may not work yet but it will after a short period of time.

17.6.1 Gossip/Epidemic Protocols

Epidemic protocols model eventual consistency on the spread of infectious diseases. For example, if you have a cold and then shake someone's hand, you infect them. This is called **pairwise exchange**. Updates will be propagated in our system in the same way. Replicas connect and check for updates periodically, so updates slowly spread to all machines. This is how most systems work since we can tolerate some small delay in things being updated. Examples include Dropbox, GitHub or any kind of cloud storage that is synchronized across machines.

There is always a random chance that some server is not infected if it never talks to an infected server and thus always some small probability that an update doesn't propagate everywhere.

Anti-Entropy

Variant where server P picks server Q at random and exchanges updates via push or pull requests, or both.

A pure push-based approach does not spread updates quickly. If we just push, we may end up just talking to machines that have already seen the updates. Generally speaking, a push gets an update spread through most of the system quickly and a pull makes sure the few remaining uninfected machines see the update.

Rumor Mongering

Another variant where when P receives an update, it pushes it to Q. If Q has already received that update, we stop propagating the update with probability $\frac{1}{k}$, giving us a probabilistic backoff. Similar to gossiping, if an item is "hot", it will keep being spread but if it becomes "cold", we stop spreading it.