Chan and Golab: Consistency and Replication

Why should we replicate data?

Reliability: having multiple copies of data helps prevent data loss in the case of failures. 3 copies means that even if two fail, we've avoided data loss. Data is precious. No bank wants to forget how much money a customer has, and no advertisement company wants to lose the data that helps them target ads. Replicas / backups help ensure we don't lose anything.

Throughput: In principle, a client could read from multiple replicas in parallel.

Latency: having data in a data center geographically close to the client means they can avoid making a slow and costly request halfway across the world

Data stores (and the objects they contain) can be either local to a process, or remote - residing on another host.

Consistency Models

Replicating read-only data is fine and dandy. The issues arise from replicating mutable data. Due to network delays, perfectly synchronized replicas simply cannot exist.

As such, there are often times in which replicas disagree with each other as to the state of a certain object or document. Consistency models help us understand concurrent operations, and allow us to describe to what

extent replicas are "allowed" to disagree. There are a few consistency models, none of which are particularly better than another - they simply allow for different things.

Model 1: Sequential Consistency

A data store is sequentially consistent if:

a) the result of any execution is the same as if the operations by all processes were executed in some sequential order*

AND

b) the operations of each individual process appear in this sequence in the order specified by its program.

This is a fairly verbose description and it's not immediately clear what constraints it is trying to enforce. Perhaps it's better to talk about the exact properties sequential consistency gives us

1) The order (mentioned at *) contains all operations in the execution and nothing else
2) Each read in the order returns the value of the most recent write that happened in this order
3) If some operation O1 precedes a different operation O2 for a specific process, O1 must precede O2 in the entire order as well

Okay, so the properties are a little easier to understand. Let's do some examples to bring it home.

ex.
P1: $W_x(a)$
P2: $W_x(b)$
P3: $R_x(b)$ $R_x(a)$
P4: $R_x(b)$ $R_x(a)$

↑ process ids
↑ reads a from x

Our goal is to find an order that satisfies our three properties, we'll call this order T. What happens if we start with P1. $W_x(a)$?

$$T = \{ P1.W_x(a) \}$$

In property 2, each read must return the last write. The two reads we have that return 'a' are P3. $R_x(a)$ and P4. $R_x(a)$. However,

$$P_3.R_x(b) \xrightarrow{\text{MUST}} P3\,R_x(a)$$
$$P4.R_x(b) \xrightarrow[\text{MUST}]{} P4.R_x(a)$$
} property 3

But there's no way to read the value as 'b' when we've only written 'a'. As such, we can't start our order with P1. $W_x(a)$. So let's try P2. $W_x(b)$ first.

$$T = \{ P2.W_x(b) \}$$

Now, we can legally execute the first reads from P3 and P4, as they read out 'b'.

$$T = \{ P2.W_x(b),\ P4.R_x(b),\ P3.R_x(b) \}$$

Now that we've dealt with all the operations that write/read 'b', we can go back to 'a'. Obviously we can't read out 'a' if we haven't written it yet, so P1. Wx(a) must go next.

T = { P2. Wx(b), P4. Rx(b), P3. Rx(b), P1. Wx(a) }

We only have read operations that read out 'a' left, so we can insert them willy-nilly at the end.

T = { P2. Wx(b), P4. Rx(b), P3. Rx(b),
P1. Wx(a), P3. Rx(a), P4. Rx(a) }

This order satisfies all three properties and as such, the given execution is sequentially consistent. Let's do another.

ex.
P1: Wx(a)
P2: Wx(b)
P3: Rx(b) Rx(a)
P4: Rx(a) Rx(b)

Let's begin with precedent constraints

P3. Rx(b) → P3. Rx(a)
P4. Rx(a) → P4. Rx(b)
} property 3

P1. Wx(a) → P4. Rx(a), P3. Rx(a)
P2. Wx(b) → P3. Rx(b), P4. Rx(b)
} property 1

P1. Wx(a) → P2. Wx(b)
P2. Wx(b) → P1. Wx(a)

The last two lines should probably sound some alarm bells. But where do these constraints come from? P3 reads 'b', then 'a' from the same variable x. As such, it's implied that the following occurred:

$$\text{write } b \rightarrow P3.Rx(b) \rightarrow \text{write } a \rightarrow P3\ Rx(a)$$

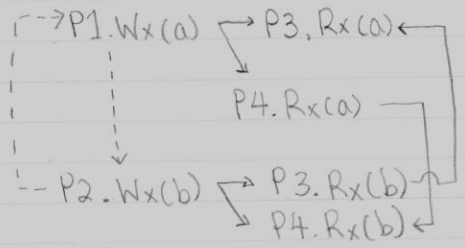
We only have one Wx(b), from P2. We only have one Wx(a), from P1. As such,

$$P2.Wx(b) \rightarrow P1.Wx(a)$$

But P4 reads 'a' then 'b'. Which would imply

$$P1.Wx(a) \rightarrow P2.Wx(b)$$

Obviously there's an issue here.



There's a cycle in our dependency graph. This means this execution is NOT sequentially consistent.

Casually, this is fairly simple to intuit given that one process reads 'a' → 'b' but another reads 'b' → 'a', but formally, showing the cycle in the dependency graph is the correct proof

Model 2: Causal Consistency

Again, a somewhat difficult to read description. A data store is causally consistent when

> Writes related by the "causally precedes" relation must be SEEN BY ALL PROCESSES in the same order. Concurrent writes may be seen in different orders in different processes.

But what exactly does causally precedes mean? Causally precedes encapsulates two different cases.

1) O1 causally precedes O2 if O1 occurs before O2 in the same process
2) O1 causally precedes O2 if O2 reads a value written by O1

Basically, we need to create an order for each process with the related operations. The properties we want to guarantee for a specific process's order T_i:

1) T_i contains all operations executed by P_i AND the writes of values read by P_i and nothing else
2) Each read returns the value of the most recent write
3) If O1 causally precedes O2 in the context of the entire process AND both operations appear in T_i, then O1 precedes O2 in T_i

We'll do an example, this time for the harder case first.

ex.
P1: Wx(a)
P2: Rx(a) Wx(b)
P3: Rx(b) Rx(a)
P4: Rx(a) Rx(b)

We begin by identifying constraints in the exact same way that we do for sequential consistency

reads from { P1. Wx(a) → P2. Rx(a), P3. Rx(a), P4. Rx(a)
P2. Wx(b) → P3. Rx(b), P4. Rx(b)

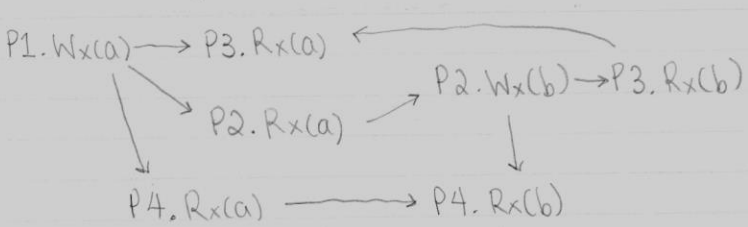
in-process order { P2. Rx(a) → P2. Wx(b)
P3. Rx(b) → P3. Rx(a)
P4. Rx(a) → P4. Rx(b)

order of writes to allow reads { P2. Wx(b) → P1. Wx(a)
P1. Wx(a) → P2. Wx(b)

These are the same constraints (as it's the same example as before) but we analyze them somewhat differently.

Again, we start with a graph, with vertices as the operations and edges as the constraints/dependencies

(only the first two)



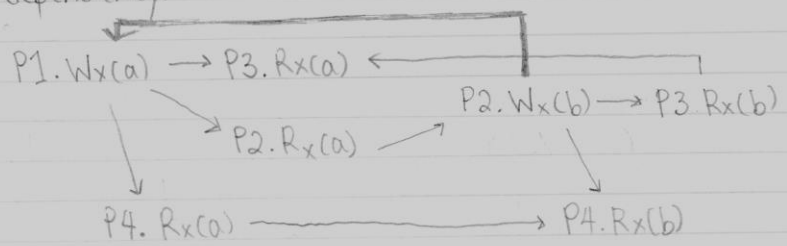
Here's the tricky part. We have to modify this graph for each process in the execution. We must add EXTRA edges based on the additional constraints on the order of writes with respect to the current process.

Let's go through them.

P1: P1 only contains a write, so no additional edges are needed. If the graph didn't already have a cycle, we'd keep going to attempt to disprove causal consistency.

P2: Aside from its own two operations, P2 needs P1's $W_x(a)$ as P2 reads 'a' out of x. This edge is already there, though, so nothing gets added for P2's subgraph.

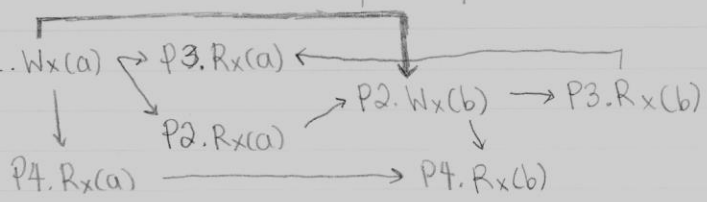
P3: Here's where things get spicy. Since P3 reads 'b' then 'a', we need the $P2.W_x(b) \rightarrow P1.W_x(a)$ dependency.



We can see this subgraph has a cycle*, so we can conclude the entire execution is not causally consistent. Let's do P4 to be sure, though

* $P1.W_x(a) \rightarrow P2.R_x(a) \rightarrow P2.W_x(b) \rightarrow P1.W_x(a)$

P4: P4 reads 'a' then 'b' which implies the P1.Wx(a) → P2.Wx(b) dependency.



This is actually fine. So we wouldn't be able to determine if this execution is causally consistent by inspecting P4.

Okay, now that we're done that we can do a nicer example.

ex.

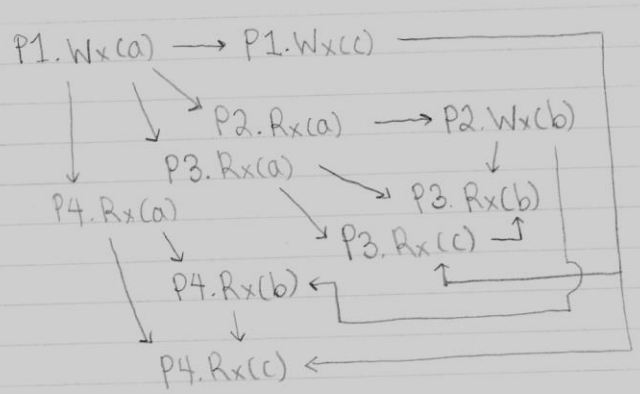
P1:	Wx(a)			Wx(c)	
P2:		Rx(a)	Wx(b)		
P3:		Rx(a)		Rx(c)	Rx(b)
P4:		Rx(a)		Rx(b)	Rx(c)

Constraints!

in-process order:
- P1.Wx(a) → P1.Wx(c)
- P2.Rx(a) → P2.Wx(b)
- P3.Rx(a) → P3.Rx(c), Rx(b)
- P3.Rx(c) → P3.Rx(b)
- P4.Rx(a) → P4.Rx(b), P4.Rx(c)
- P4.Rx(b) → P4.Rx(c)

read after write:
- P1.Wx(a) → P2.Rx(a), P3.Rx(a), P4.Rx(a)
- P2.Wx(b) → P3.Rx(b), P4.Rx(b)
- P1.Wx(c) → P3.Rx(c), P4.Rx(c)

Let's draw our dependency graph.



Despite how gross this looks, there aren't actually any cycles. Let's go through each process's view of events.

P1: P1 only has writes, so a proper ordering would be

$$T_1 : \{ P1.Wx(a), P1.Wx(c) \}$$

P2: P2 reads out 'a' from x, so all it needs is P1.Wx(a) before the read.

$$T_2 : \{ P1.Wx(a), P2.Rx(a), P2.Wx(b) \}$$

P3: P3 reads 'a', 'c', then 'b'. So we just have to shove our writes in that order, and it's all gucci

$$T_3 : \{ P1.Wx(a), P3.Rx(a), P1.Wx(c), P3.Rx(c), P2.Wx(b), P3.Rx(b) \}$$

P4: P4 reads 'a', 'b', then 'c'.

T_4: { P1.Wx(a), P4.Rx(a), P2.Wx(b)
P4.Rx(b), P1.Wx(c), P4.Rx(c) }

Since we can make legal orderings for all 4 processes, this execution is causally consistent.

Model 3: Linearizability

Unlike the other models, linearizability isn't restricted to just read and write operations. It also assumes all operations have defined start/finish times, ordered by a hypothetical global clock.

A datastore is linearizable when:

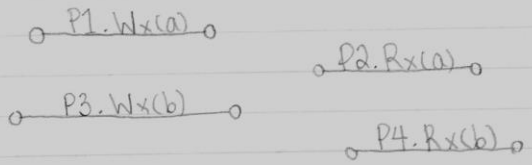
> The result of any execution is the same as if the operations were executed in an order such that O1 → O2 if O1 finishes before O2.

The three properties we can guarantee are familiar:

1) T contains all and only the operations in a given execution
2) Each read returns the value of the most recent write
3) If O1 finishes before O2 begins, O1 must precede O2 in T

Example time woo!

ex.



Damn, Daniel! Back at it again with the timing constraints!

$P_1.Wx(a) \rightarrow P_2.Rx(a)$ } read order
$P3.Wx(b) \rightarrow P4.Rx(b)$ }

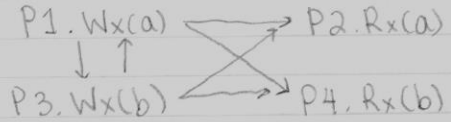
$P1.Wx(a) \rightarrow P2.Rx(a), P4.Rx(b)$ } finishes before
$P3.Wx(b) \rightarrow P2.Rx(a), P4.Rx(b)$ }

$P1.Wx(a) \rightarrow P3.Wx(b)$ because $P1.Wx(a) \rightarrow P4.Rx(b)$
$P3.Wx(b) \rightarrow P1.Wx(a)$ because $P3.Wx(b) \rightarrow P2.Rx(a)$

The last two, again, are the tricky ones. P1.Wx(a) precedes P4.Rx(b). Now, in order for P4 to read out 'b', a write must have happened between those two operations: P3.Wx(b).

The reverse is true: $P_3.Wx(b)$ precedes P2.Rx(a). As such, a write must have happened in between: P1.Wx(a).

As such, our dependency graph looks like this.



There's a cycle, and as such this execution is not linearizable. How about something that is?

Eventual Consistency (Kind of Model 4)

In the absence of new writes from clients, all servers will eventually hold the same data. This is a super weak consistency model.

This allows for observation of writes in an order that violates causal consistency or "happens before".

Things are not eventually consistent if there is a set of operations that go on indefinitely:

1) P3. Rx(a)
2) P4. Rx(b)
3) Repeat 1) and 2) forever

Session Guarantees

This is usually used with eventual consistency as eventual consistency is too weak. Session guarantees restrict what a single process can do in a single session

Some possible properties:

Monotonic reads - reads on x will return the same value OR a more recent value

Read your own write - any writes by process P will always be seen by successive reads on the same item by P.

Protocols

So great, now we have some ability to determine how to schedule operations in a way that makes sense. But once changes are made, how do we update our replicas with the new data? How do we prevent clashes from happening due to concurrency?

This is what replication protocols are for. These protocols let us decide who writes to who, and how to avoid concurrency issues and stale data.

Primary-Based Protocols

In this protocol, all updates are executed by a designated primary replica. Updates are then pushed to one or more backup replicas. These types can be classified as either

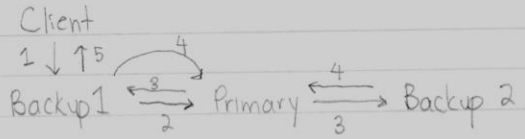
- remote write
  - primary replica is always the same machine and updated remotely by other servers, or
- local write
  - primary replica migrates from server to server, allowing backed up local updates to execute

If the primary replica fails, one of the backups has to take over.

Generally, protocols solve the concurrency issue by using locks (most notably, two-phase locking)

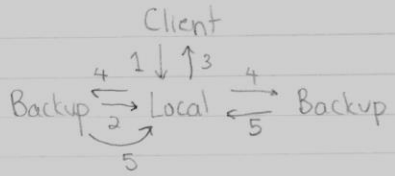
ex. Remote-write protocol

1) Client executes write request
2) Request gets forwarded to the primary
3) Primary tells all backups to update
4) Backups acknowledge update
5) Local server acknowledges write completion to client



ex. Local-write protocol

1) Client executes write request
2) Item to be written to is moved to the local server if needed
3) Write acknowledged to the client
4) Tell backups to update
5) Update acknowledged to local



Quorum-Based Protocols

Forcing all updates to go through the primary is nice because it lets us implement strong consistency models like sequential consistency and linearizability.

However, this comes at the price of performance bottlenecks and temporary availability loss if the primary fails. Moving an item to a local server to perform a write operation is essentially the same as simply locking the item - defeating concurrency.

Quorum-based protocols try to remedy this. All replicas can receive updates, but each update must be accepted by "enough" replicas - a write quorum. A read has a similarly enforced read quorum.

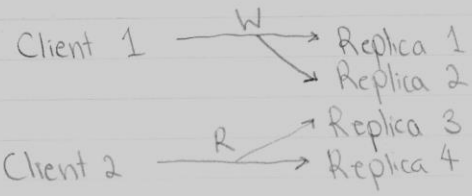
The exact number is given by the following property:

1) $N_R + N_W > N$

2) $N_W + N_W > N$

The first property states that the read quorum and write quorum must be, together, larger than the total number of replicas. Why is this?

Say we have two clients and four servers, and $N_R = 2$ and $N_W = 2$. Client 1 wants to write to A. Client 2 wants to read A afterward.



Despite Client 2's read coming after, it reads a stale value because its quorum consists of replicas that didn't get written to.

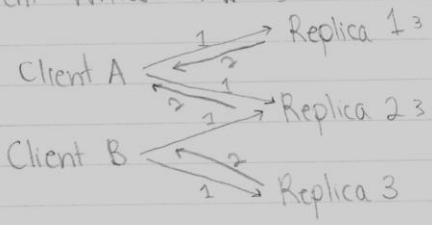
This tells us two useful things. First, quorums are usually implemented with an odd number of servers. Second, the first property is to help prevent read-write conflicts.

The second property (I'm sure you've guessed now) is to prevent write-write conflicts. This way, two separate writes cannot occur at the same time, and as such we won't be confused about what the freshest value is.

There's also no such thing as a read-read conflict, so we don't need a property for that.

Let's go through some examples to see how this works.

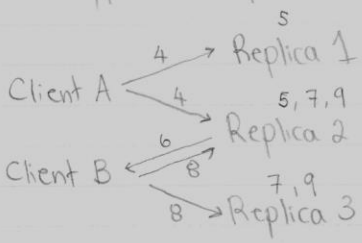
ex. Concurrent Writes ($N_W = 2$)



1) Both client A and client B want to perform a write on the same object. Both replica 1 and replica 3 can accept their respective write requests. However, replica 2 must make a decision on who to grant the write lock to. Generally, the network is enough to force an ordering, and as such, no actual decision making code is required.

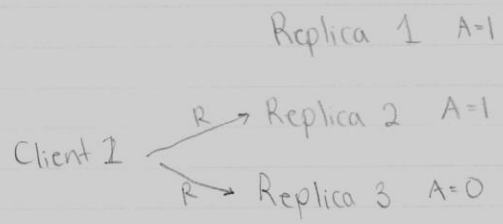
2) Locks are granted to A for 1, 2. Lock is granted to B for 3. It will have to wait until A's request has complete.

3) Updates are applied at replicas 1 and 2



4) Client A completes its update and tells replicas 1 and 2 to release the write locks.
5) Locks are released
6) Lock at replica 2 is then granted to B.
7) B's update has been applied, at replicas 2 and 3
8) Client B tells replicas 2 and 3 to release their locks.
9) Locks are released.

That's it! By using a combination of locks and the network, a quorum system can maintain consistency. Let's take a look at how disagreements between replicas are resolved



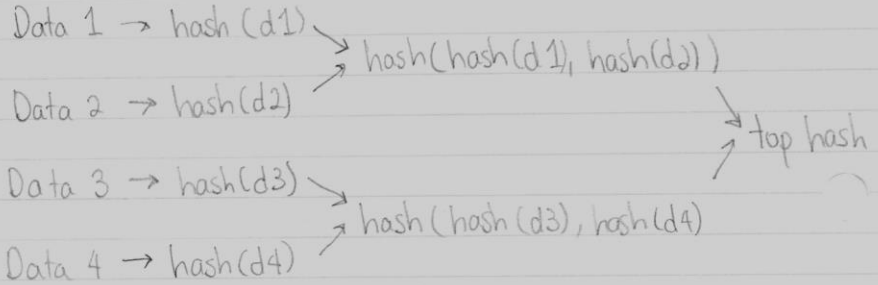
Client 1 properly acquires locks from replicas 2 and 3 for the object A. However, 2 and 3 think A has different values.

In this case, replicas 2 and 3 will compare the timestamps of the updates, and return the most recent value, and update the replica that has the stale value. This leads to eventual consistency.

Majority quorums like this allow for linearizability on individual data items.

Anti-entropy

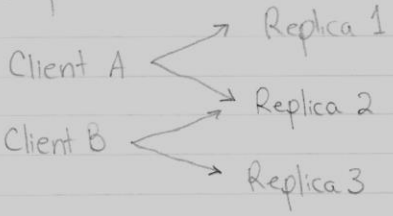
As the name might imply, anti-entropy is the process of reversing disorder - in other words, a way to re-synchronize stale servers. Each server constructs a hash tree (Merkle tree), a way to represent the data it stores without needing to compare each entry.



If the top hashes are different, a server will step down the tree in order to determine what entries differ, compare timestamps, and update data accordingly.

Quorums without Locks

If we want to avoid the overhead of locks, we can use a system similar to what Apache Cassandra uses. Instead of locks, this uses timestamps.



Again, writes are performed concurrently by A and B. Replicas 1 and 3 behave the same way as before, just without locks.

At replica 2, what happens is dependent on timestamps. Given that A's update happens at t_A, and B's happens at t_B, and $t_A < t_B$, one of two things can occur

1) A's update is applied, then B's

or

2) B's update is applied, then A's is silently dropped

After this, the replicas respond as normal to the clients. This system does not guarantee linearizability, and only recognizes the $N_R + N_W > N$ constraint, as write conflicts are no longer a thing to worry about.