brianstorti.com

A Primer on Database Replication

46-58 minutes

Get as PDF or ePub

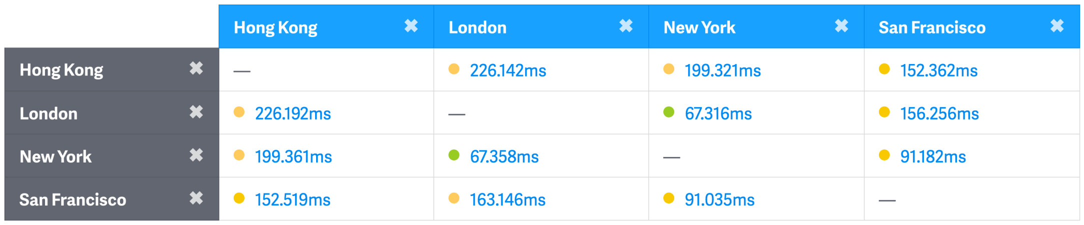
Replicating a database can make our applications faster and increase our tolerance to failures, but there are a lot of different options available and each one comes with a price tag. It’s hard to make the right choice if we do not understand how the tools we are using work, and what are the guarantees they provide (or, more importantly, do *not* provide), and that’s what I want to explore here.

Before we start, some background

AlphaSights has offices in North America, Europe and Asia and is rapidly expanding. People working in these 3 continents rely heavily on the tools that we build to do their job, so any performance issue has a big impact in their work. As the number of people using our systems increased our database started to feel the pressure. Initially we could just keep increasing our database server capacity, getting a more powerful machine, adding more RAM, and keep scaling vertically, but there is one problem that we cannot solve, unfortunately: The limit of the speed of light.

> Light travels with a speed of 299,792 km/s in a vacuum. Even if we assume that our requests are traveling at this speed, and that they travel in a straight line, it would still take 133ms for a round the world trip. In reality, our requests will be slower than the speed of light and there will be a lot of zig zag from one hop to another until

they reach their destination.



	Hong Kong ✖	London ✖	New York ✖	San Francisco ✖
Hong Kong ✖	—	● 226.142ms	● 199.321ms	● 152.362ms
London ✖	● 226.192ms	—	● 67.316ms	● 156.256ms
New York ✖	● 199.361ms	● 67.358ms	—	● 91.182ms
San Francisco ✖	● 152.519ms	● 163.146ms	● 91.035ms	—

No matter how quickly we can *execute* a query, if the database is in North America, the data still needs to travel all the way to Asia before people in that office can use it. It was clear that we had to make that data available somewhere closer to them, and so the quest began.

Researching all the available options and everything that is involved in a database replication setup can be overwhelming, there are literally decades of literature about the subject, and after you start digging it's hard to see the end.

I am by no means a replication expert, but during this process I learned a thing or two, and that's what I want to share here. This is not supposed to be an extensive resource to learn everything there is to know about replication, but hopefully it's a good starting point that you can use in your own journey. In the end of this article I will link to some great resources that can be helpful if you decide to learn more.

Sounds good? Cool, grab a cup of coffee and let's have fun.

First things first, the What and the Why

Just to make sure we are on the same page, let's define what replication is and describe the three main reasons why we might want it.

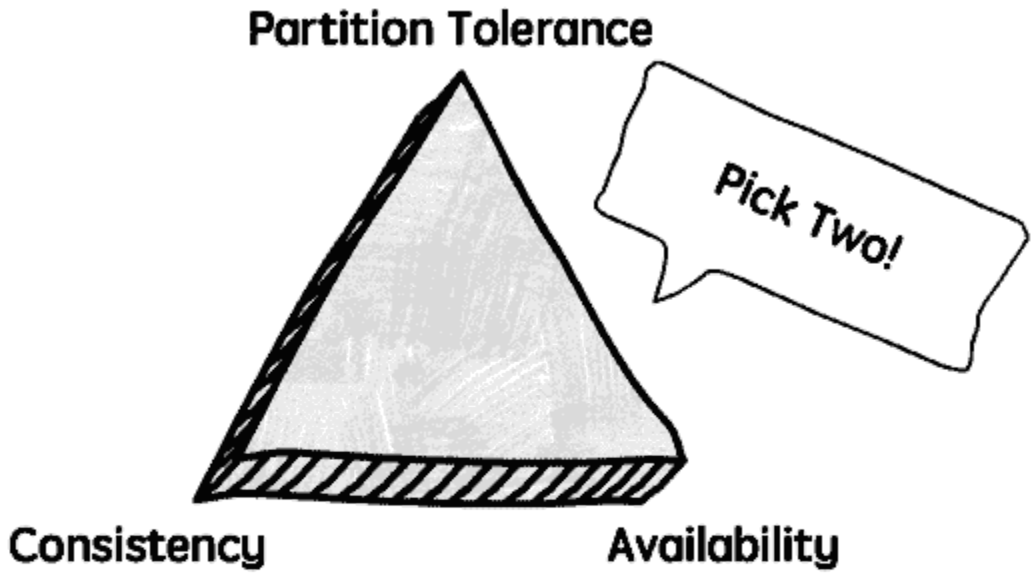
When we say we want to replicate something, it means we want to keep a copy of the same data in multiple places. In the case of databases, that can mean a copy of the entire database, which is the most common scenario, or just some parts of it (e.g. a set of tables). These multiple locations where we will keep the data are usually connected by a network, and that's the origin of most of our headaches, as you will see in a bit.

The reason for wanting that will be one or more of the following:

- You want to keep the data closer to your users so you can save the travel time. Remember, no matter how fast your database is, the data still needs to travel from the computer that started the request to the server where the database is, and then back again. You can optimize the heck out of your database, but you cannot optimize the laws of physics.
- You want to scale the number of machines serving requests. At some point a single server will not be able to handle the number of clients it needs to serve. In that case, having several databases with the same data helps you serve more clients. That's what we call scaling *horizontally* (as opposed to *vertically*, which means having a more powerful machine).
- You want to be safe in case of failures (that will happen). Imagine you have your data in single database server and that server catches fire, then what happens? I am sure you have some sort of backup (right?!), but your backup will a) take some time to be restored and b) probably be *at least* a couple of hours old. Not cool. Having a replica means you can just start sending your requests to this server while you are solving the fire situation, and maybe no one will even notice that something bad happened.

The obligatory CAP introduction

The CAP theorem was introduced by Eric Brewer in the year 2000, so it's not a new idea. The acronym stands for Consistency, Availability and Partition Tolerance, and it basically says that, given these 3 properties in a distributed system, you need to choose 2 of them (i.e. you cannot have all 3). In practice, it means you need to choose between consistency and availability when an inevitable partition happens. If this sounds confusing, let me briefly define what these 3 terms mean, and why I am even talking about this here.

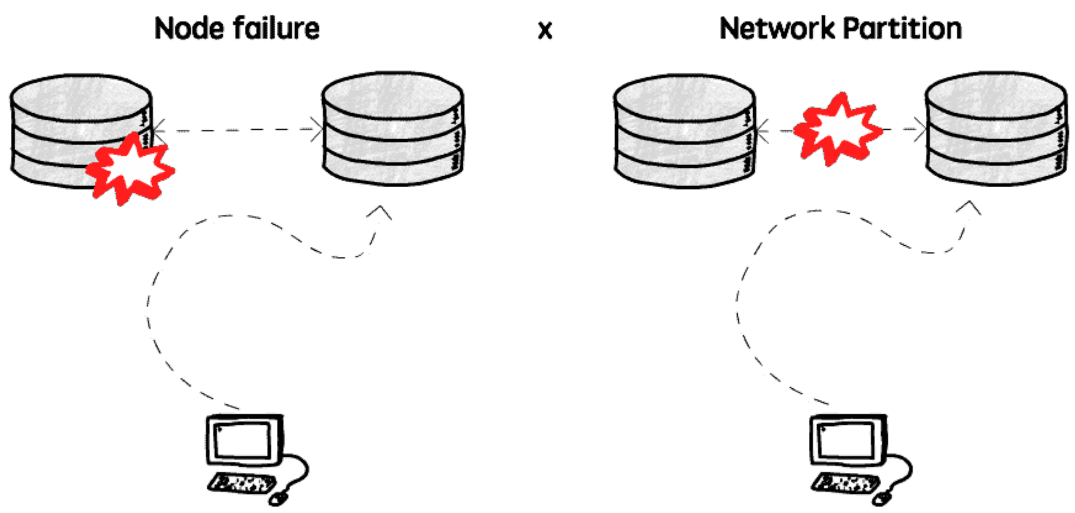


Consistency: In the CAP definition, consistency means that all the nodes in a cluster (e.g. all your database servers, leaders and replicas) see the same data at any given point in time. In practice, it means that if you query any of your database servers at the exact same time, you will get the same result back.

> Notice that this is completely unrelated to the 'Consistency' from the ACID properties.

Availability: It means that reads and writes will always succeed, even if we cannot guarantee that it will have the most recent data. In practice, it means that we will still be able to use one of our databases, even when it cannot talk to the others, and therefore might not have received the latest updates.

Partition Tolerance: This means that your system will continue working even if there is a network partition. A network partition means that the nodes in your cluster cannot talk to each other.



And why am I talking about this? Well, because depending on the route you take you will have different trade-offs, sometimes favoring consistency and sometimes availability.

How valuable the CAP theorem is in the distributed systems discussions is debatable, but I think it is useful to keep in mind that you are almost always trading consistency for availability (and vice-versa) when dealing with network partitions.

A word about latency

Latency is the time that a request is waiting to be handled (it's *latent*). Our goal is to have the lowest latency possible. Of course, even with a low latency we can still have a high *response time* (if a query takes a long time to run, for example), but that's a different problem.

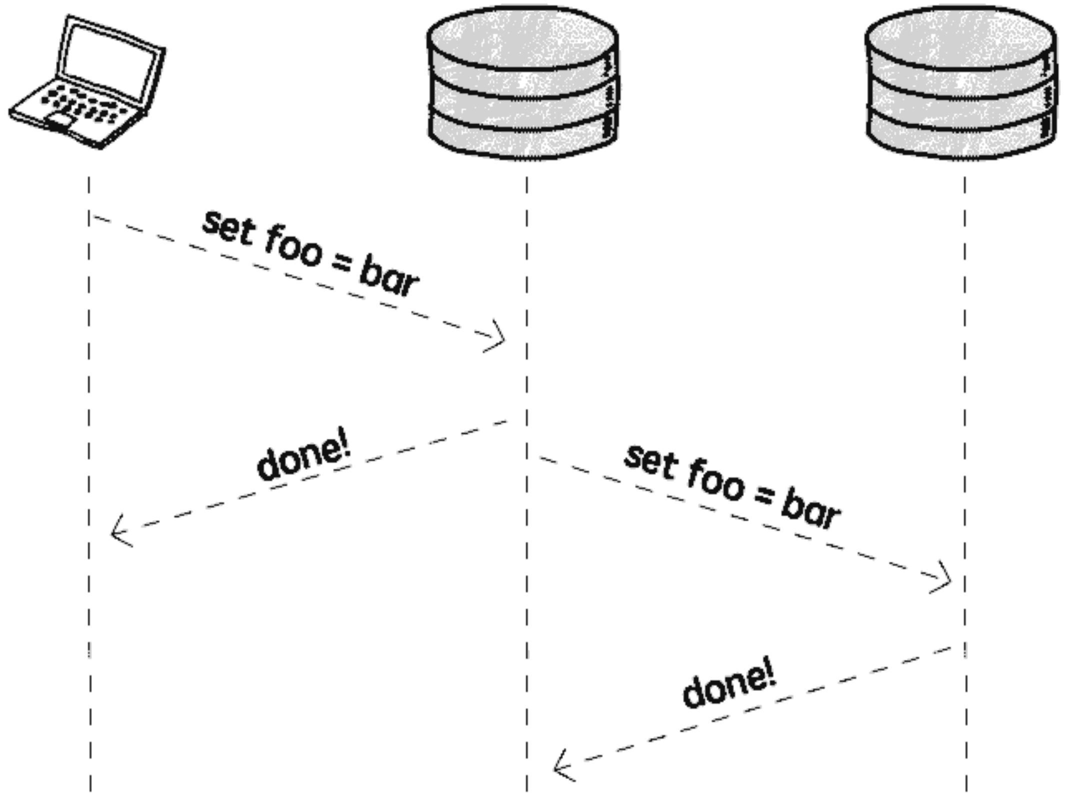
When we replicate our database we can decrease the latency by shortening the distance this request needs to travel and/or increasing our capacity, so the request doesn't need to wait before it can be handled due to a busy server.

I'm just mentioning this here because I think it's very important to be sure that the reason why we are experiencing high response times is really because the latency is high, otherwise we may be solving the wrong problem.

Asynchronous replication

When we talk about replication, we are basically saying that when I write some data in a given node A, this same data also needs to be written in node B (and maybe C and D and E and…), but we need to decide *how* this replication will happen, and what are the guarantees that we need. As always, it's all about trade-offs. Let's explore our options.

The first option is to be happy to send a confirmation back to the client as soon as the node that received the message has successfully written the data, and *then* send this message to the replicas (that may or may not be alive). It works somewhat like this:

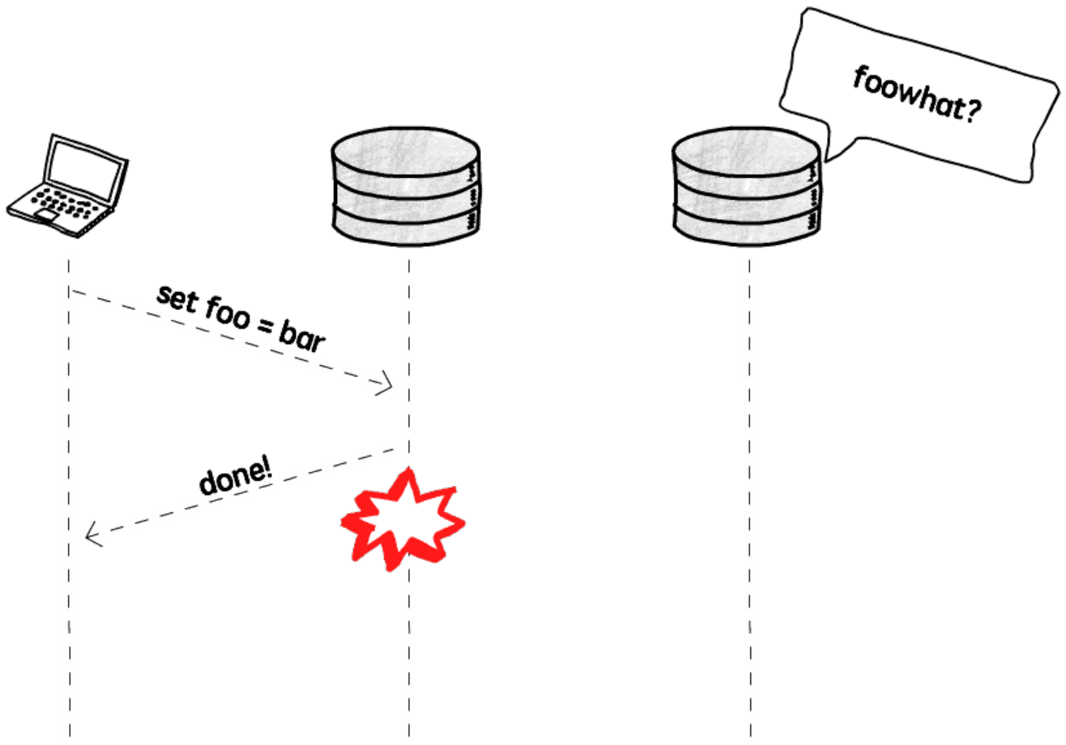


This looks great, we don't notice any performance impact as the replication happens in the background, after we already got a response, and if the replica is dead or slow we won't even notice it, as the data was already sent back to the client. Life is good.

There are (at least) two main issues with asynchronous replication.

The first is that we are weakening our durability guarantees, and the other is that we are exposed to replication lags. We will talk about replication lag later, let's focus on the durability issue first.

Our problem here is that if the node that received this write request fails before it can replicate this change to the replicas, the data is lost, even though we sent a confirmation to the client.



You may be asking yourself

> "But what are the chances of a failure happening right at THAT moment?!"

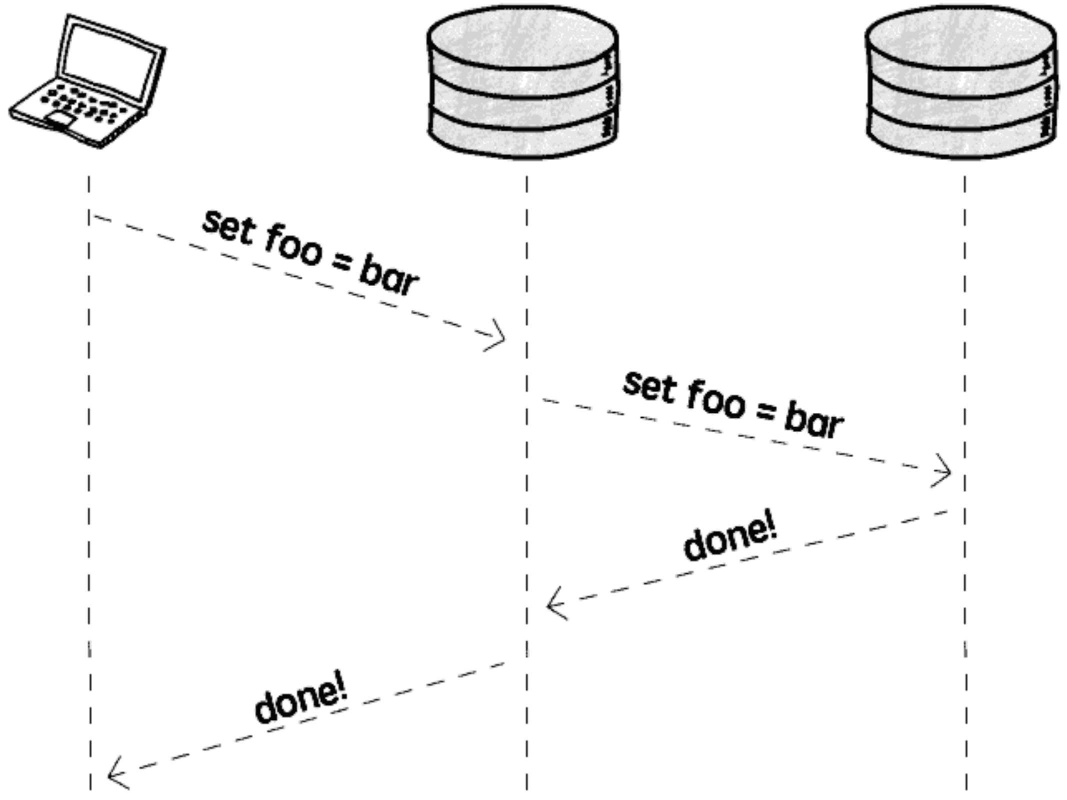
If that's the case, I'll suggest that you instead ask

> "What are the *consequences* if a failure happens at that moment?"

Yes, it may be totally fine to take the risk, but in the classic example of dealing with financial transactions, maybe it's better to pay the price to have stronger guarantees. But what is the price?

Synchronous replication

As you might expect, synchronous replication basically means that we will *first* replicate the data, and then send a confirmation to the client.



So when the client gets the confirmation we can be sure that the data is replicated and safe (well, it's never 100% safe, all of our data centers can, in theory, explode at the same time, but it's safe enough).

The price we need to pay is: Performance and availability.

The performance penalty is due to the fact that we need to *wait* for these - potentially - slow replicas to do their thing and send us a confirmation before we can tell the client that everything is going to be fine. As these replicas are usually distributed geographically, and potentially very far from each other, this takes more time than we would like to wait.

The second issue is availability. If one of the replicas (remember, we can have many!) is down or we cannot reach it for some reason, we simply cannot write any data. You should always plan for failures, and network partitions are more common than we imagine, so depending on *all* replicas being reachable to perform any write doesn't seem like a great idea to me (but maybe it is for your specific case).

Not 8, not 80

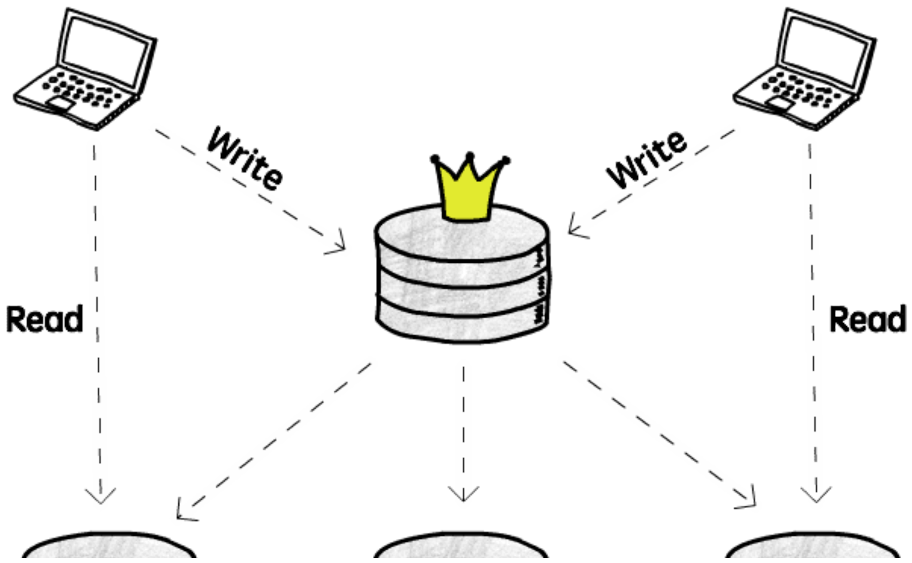
There's some middle ground. Some databases and replication tools allow us to define a number of followers to replicate synchronously, and the others just use the asynchronous approach. This is sometimes called *semi-synchronous replication*.

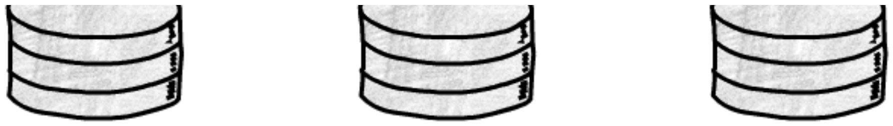
As an example, in `Postgres` you can define a configuration called `synchronous_standby_names` to specify which replicas will receive the updates synchronously, and the other replicas will just receive them asynchronously.

Single leader replication

The most common replication topology is to have a single leader, that then replicate the changes to all the followers.

In this setup, the clients always send writes (in the case of databases, `INSERT`, `UPDATE` and `DELETE` queries) to the leader, and never to a follower. These followers can, however, answer read queries.





The main benefit of having a single leader is that we avoid conflicts caused by concurrent writes. All the clients are writing to the same server, so the coordination is easier. If we instead allow clients to write to 2 different servers at the same time, we need to somehow resolve the conflict that will happen if they both try to change the same *object*, with different values (more on that later).

So, what are the problems that we need to keep in mind if we decide to go with the single leader approach? The first one is that we need to make sure that just one node is able to handle all the writes. Although we can split the read work across the entire cluster, all the writes are going to a single server, and if your application is very write-intensive that might be a problem. Keep in mind though, that most applications read a lot more data than they write, so you need to analyze if that’s really a problem for you.

Another problem is that you will need to pay the latency price on writes. Remember our colleagues in Asia? Well, when they want to update some data, that query will still need to travel the globe before they get a response.

Lastly, although this is not really a problem just for single leader replication, you need to think about what will happen when the leader node dies. Is the entire system going to stop working? Will it be available just for reads (from the replicas), but not for writes? Is there a process to *elect* a new leader (i.e. promoting one of the replicas to a leader status)? Is this election process automated or will it need someone to tell the system who is the new king in town?

At first glance it seems like the best approach is to just have an automatic failover strategy, that will elect a new leader and everything will keep working wonderfully. That, unfortunately, is easier said than done.

The challenges of an automatic failover

Let me list *some* of the challenges in implementing this automatic failover strategy.

The first question we need to ask is: How can we be sure that the leader is dead? And the answer is: We probably can't.

There are a billion things that can go wrong, and, like in any distributed system, it is impossible to distinguish a slow-to-answer from a dead node. Databases usually use a timeout to decide that (e.g. if I don't hear from you in 20 seconds you are dead to me!). That is usually good enough, but certainly not perfect. If you wait more, it is less likely that you will identify a node as dead by mistake, but it will also take more time start your failover process, and in the meantime your system is probably unusable. On the other hand, if you don't give it enough time you might start a failover process that was not necessary. So that is challenge number one.

Challenge number two: You need to decide who is the new leader. You have all these followers, living in an anarchy, and they need to somehow agree on who should be the new leader. For example, one relatively simple (at least conceptually) approach it to have a predefined successor node, that will assume the leader position when the original leader dies. Or you can choose the node that has the most recent update (e.g. the one that is closer to the leader), to minimize data loss. Any way you decide to choose the new leader, all the nodes still need to *agree* on that decision, and that's the hard part. This is known as a consensus problem, and can be quite tricky to get right.

Alright, you detected that the leader is really dead and selected a new leader, now you need to somehow tell the clients to start sending writes to this new leader, instead of the dead one. This is a *request routing* problem, and we can also approach it from several different angles. For example, you can allow clients to send writes to any node, and have these nodes redirect this request to the leader. Or you can have a *routing layer* that receives this messages and redirect them to the appropriate node.

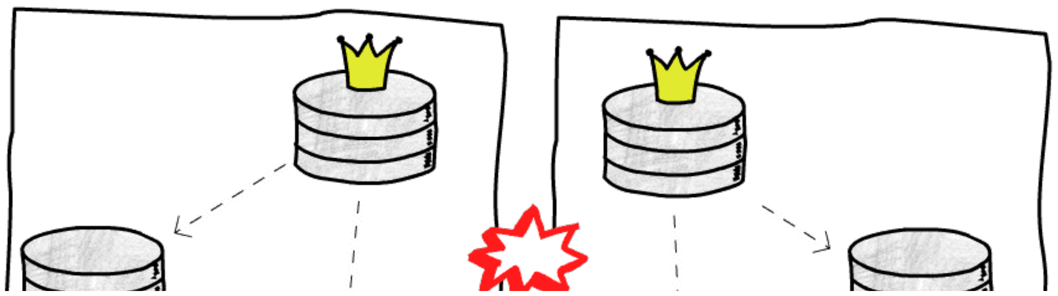
If you are using asynchronous replication, the new leader might not have all the data from the previous leader. In that case, if the old leader resurrects (maybe it was just a network glitch or a server restart) and the new leader received conflicting updates in the meantime, how do we handle these conflicts?
One common approach is to just discard these conflicts (using a last-write-win approach), but that can also be dangerous (take this Github issue (from 2012) as an example).

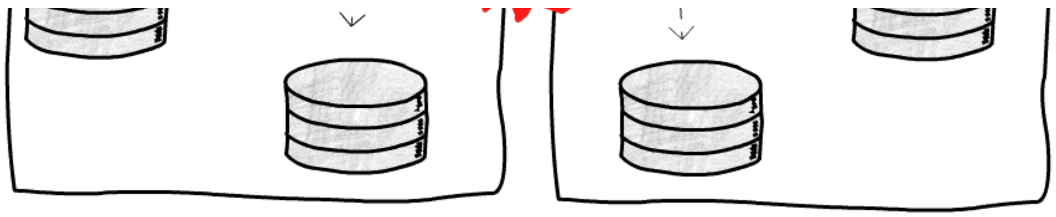
We can also have a funny (well, maybe it's not that funny when it happens in production) situation where the previous leader comes back up and thinks it is still the leader. That is called a *split brain*, and can lead to a weird situation.

If both leaders starts accepting writes and we are not ready to handle conflicts it is possible to lose data.

Some systems have fencing mechanisms that will force one node to shut down if it detects that there are multiple leaders. This approach is known by the great name `STONITH`, Shoot The Other Node In The Head.

> This is also what happens when there's a network partition and we end up with what appears to be two isolated clusters, each one with its own leader, as each part of this cluster cannot see the other, and therefore thinks they are all dead.



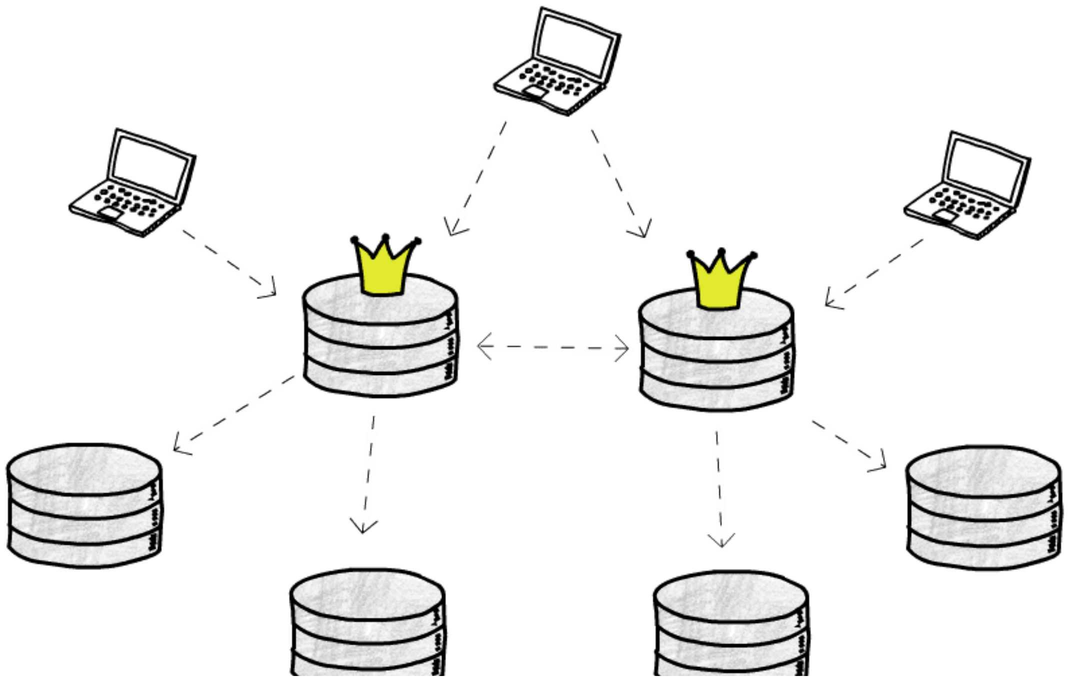


As you can see, automatic failovers are not simple. There are a lot of things to take into consideration, and for that reason sometimes it's better to have a human manually perform this procedure. Of course, if your leader database dies at 7pm and there's no one on-call, it might not be the best solution to wait until tomorrow morning, so, as always, trade-offs.

Multi leader replication

So, we talked a lot about single leader replication, now let's discuss an alternative, and also explore its own challenges and try to identify scenarios where it might make sense to use it.

The main reason to consider a multi leader approach is that is solves some of the problems that we face when we have just one leader node. Namely, we have more than one node handling writes, and these writes can be performed by databases that are closer to the clients.



If your application needs to handle a very high number of writes, it might make sense to split that work across multiple leaders. Also, if the latency price to write in a database that is very far is too high, you could have one leader in each location (for example, one in North America, one in Europe and another in Asia).

Another good use case is when you need to support offline clients, that might be writing to their own (leader) database, and these writes need to be synchronized with the rest of the databases once this client gets online again.

The main problem you will face with multiple leaders accepting writes is that you need some way to solve conflicts. For example, let's say you have a database constraint to ensure that your users' emails are unique. If two clients write to two different leaders that are not yet in sync, both writes will succeed in their respective leaders, but we will have problems when we try to replicate that data. Let's talk a bit more about these conflicts.

> Here we are assuming that these leaders replicate data asynchronously, that's why we can have conflicts. You could, in theory, have multi leader synchronous replication, but that doesn't really make a lot of sense, as you lose the main benefit of having leaders accepting writes independently, and might just use single leader replication instead. There are some projects, though, like PgCluster, that implement multi master synchronous replication, but they are mostly abandoned, and I will not talk about this type of replication here.

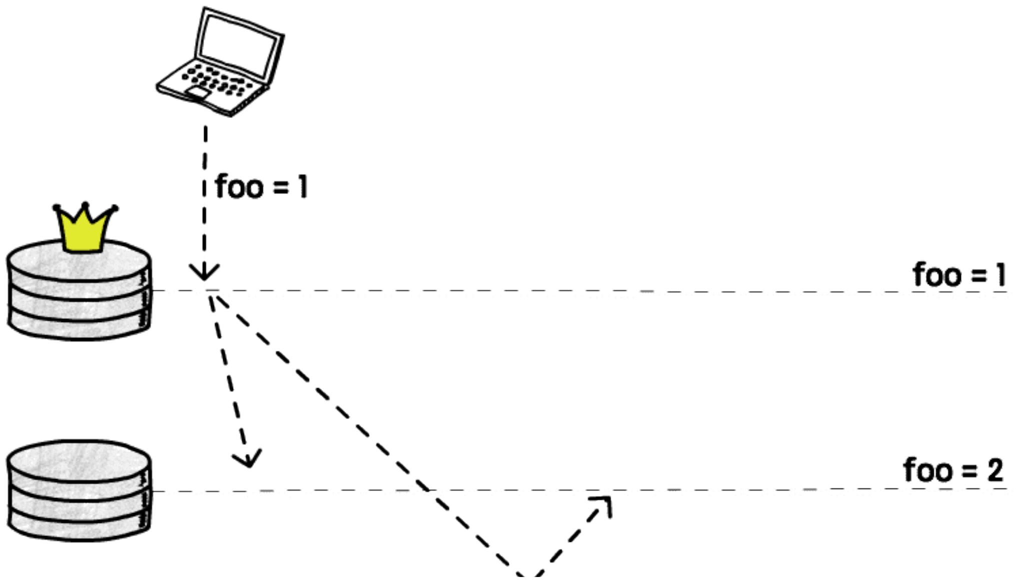
Dealing with conflicts

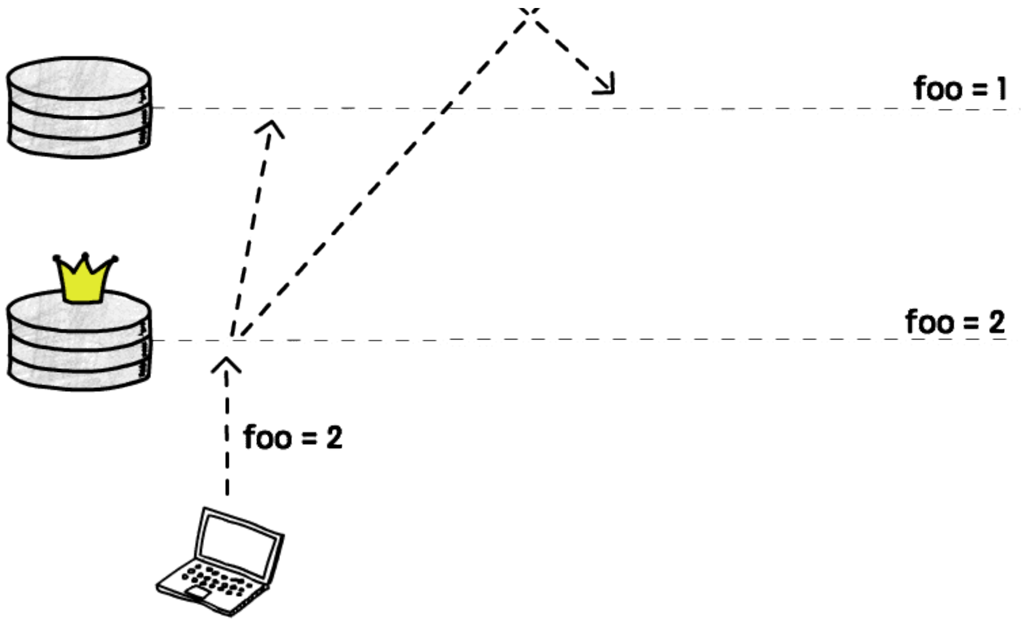
The easiest way to handle conflicts is to not have conflicts in the first place. Not everyone is lucky enough to be able to do that, but let's see how that could be achieved.

Let’s use as an example an application to manage the projects in your company. You can ensure that all the updates in the projects related to the American office are sent to the leader in North America, and all the European projects are written to the leader in Europe. This way you can avoid conflicts, as the writes to the same projects will be sent to the same leader. Also, if we assume that the clients updating these projects will probably be in their respective offices (e.g. people in the New York office will update the American projects, that will be sent to the leader in North America), we can ensure that they are accessing a database geographically close to them.

Of course, this is a very biased example, and not every application can “partition” its data in such an easy way, but it’s something to keep in mind. If that’s not your case, we need another way to make sure we end up in a consistent state.

We cannot let each node just apply the writes in the order that they see them, because a node `A` may first receive an update setting `foo=1` and then another update setting `foo=2`, while node `B` receive these updates in the opposite order (remember, these messages are going through the network and can arrive out of order), and if we just blindly apply them we would end up with `foo=2` on node `A` and `foo=1` on node `B`. Not good.





One common solution is to attach some sort of timestamp to each write, and then just apply the write with the highest value. This is called LWW (last write wins). As we discussed previously, with this approach we may lose data, but that's still very widely used.

> Just be aware that physical clocks are not reliable, and when using timestamps you will probably need at least some sort to clock synchronization, like NTP.

Another solution is to record these conflicts, and then write application code to allow the user to manually resolve them later. This may not be feasible in some cases, like in our previous example with the unique constraint for the email column. In other cases, though, it may be just a matter of showing two values and letting the user decide which one should be kept and which should be thrown away.

Lastly, some databases and replication tools allow us to write custom conflict resolution code. This code can be executed on write or on read time. For instance, when a conflict is detected a stored procedure can be called with the conflicting values and it decides what to do with them. This is a *on write* conflict resolution. Bucardo and BDR are example of tools that use this approach.

Other tools use a different approach, storing all the conflicting writes, and also returning all of them when a client tries to read that value. The client is then responsible for deciding what to do with those values, and write it back to the database. CouchDB, for example, does that.

There is also a relatively new family of data structures that provide automatic conflict resolution. They are called *Conflict-free replicated data type*, or *CRDT*, and to steal the wikipedia definition:

> CRDT is a data structure which can be replicated across multiple computers in a network, where the replicas can be updated independently and concurrently without coordination between the replicas, and where it is always mathematically possible to resolve inconsistencies which might result.

Unfortunately there are some limitations to where these data structured can be used (otherwise our lives would be too easy, right?), and as far as I know they are still not very widely used for conflict resolution in databases, although some CRDTs were implemented in Riak.

DDL replication

Handing DDLs (changes in the structure of the database, like adding/removing a column) can also be tricky in a multi leader scenario. It's, in some sense, also a conflict issue, we cannot change the database structure while other nodes are still writing to the old structure, so we usually need to get a global database lock, wait until all the pending replications take place, and then execute this DDL. In the meantime, all the writes will either be blocked or fail. Of course, the specific details will depend on the database or replication tool used, and some of them will not even try to replicate DDLs, so you need to somehow do that manually, and other tools will replicate *some* types of DDLs, but not other (for instance, DDLs that need to rewrite the entire table are forbidden in BDR).

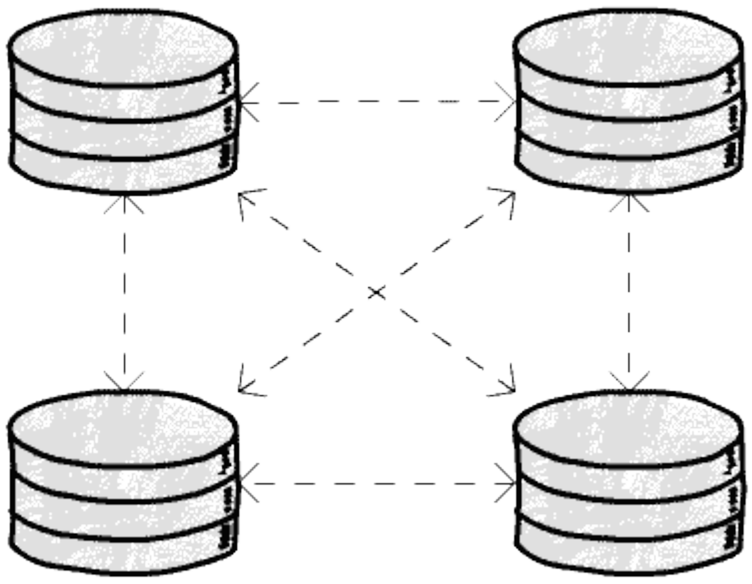
The point is, there is a lot more coordination involved in replicating DDLs when you have multiple leaders, so that's also something to keep in mind when considering this setup.

The topologies of a multi leader setup

There are several different kinds of topologies that we can use with multiple leaders. A topology defines the communication patterns between your nodes, and different ways to arrange your communication paths have different characteristics.

If you have only two leaders, there are not a lot of options: Node A sends updates to node B, and node B sends updates to node A. Things start to get more interesting when you have three or more leaders.

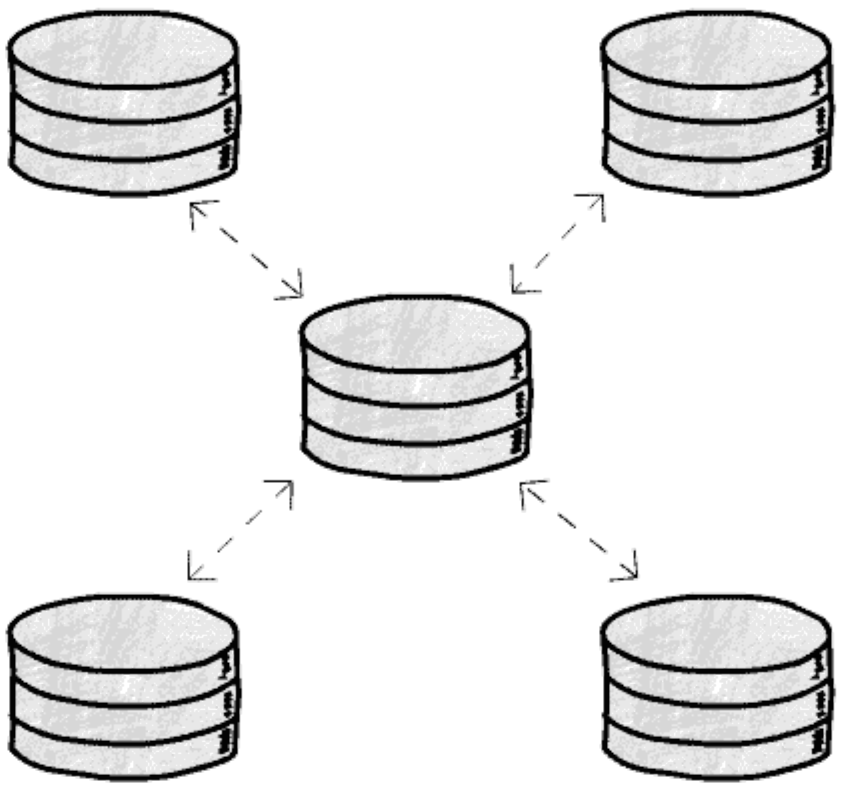
A common topology is to have each leader sending its updates to every other leader.



The main problem here is that messages can arrive out of order. For example, if a node A inserts a row, and then node B updates this row, but node C receives the update before the insert, we will have problems.

This is a *causality problem*, we need to make sure that all the nodes first process the insert event before processing the update event. There are different ways to solve this problem (for instance, using logical clocks), but the point is: You need to make sure your database or replication tool is actually handling this issue, or, in case it's not, be aware that this is a failure that can happen.

Another alternative is to use what some databases call the *star topology*.



In this case one node receives the updates and sends them to everyone else. With this topology we can avoid the causality problem but, on the other hand, introduce a single point of failure. If this central node dies, the replication will stop. It's high price to pay, in some cases.

Of course, these are just 2 examples, but the imagination is the limit for all the different topologies you can have, and there's no perfect answer, each one will have their pros and cons.

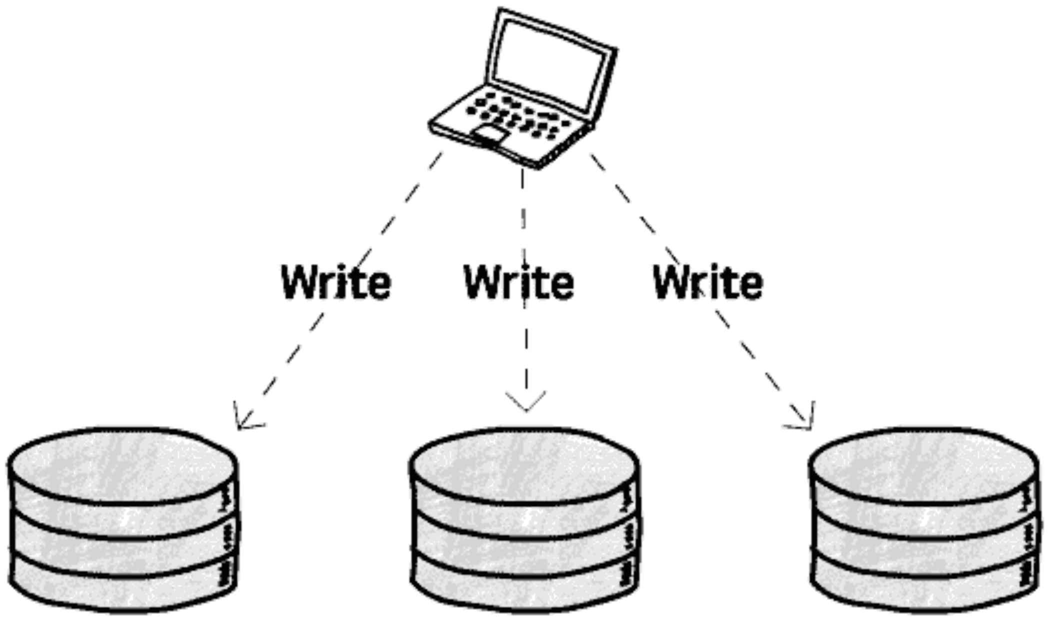
And yes, leaderless replication

Another idea that was popularized by Amazon's DynamoDB (although it first appeared some decades ago) is to simply have no leaders, every replica can accept writes (maybe it should be called leaderful?).

It seems like this is going to be a mess, doesn't it? If we had lots of conflicts to handle with a few leaders, imagine what will happen when writes are taking place everywhere. Chaos!

Well, it turns out these database folks are quite smart, and there are some clever ways to deal with this chaos.

The basic idea is that clients will send writes not only to one replica, but to several (or, in some cases, to all of them).

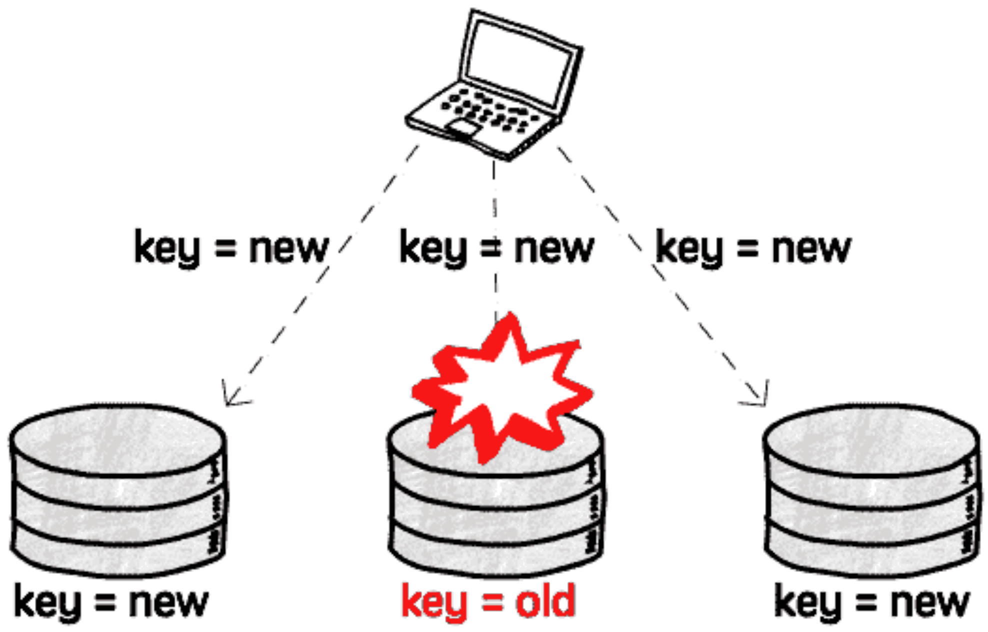


The client sends this write request concurrently to several replicas, and as soon as it gets a confirmation from some of them (we will talk about how many are "some" in a bit) it can consider that write a success and move on.

One advantage we have here is that we can tolerate node failures more easily. Think about what would happen in a scenario where we had to send a write to a single leader and for some reason that leader didn't respond. The write would fail and we would need to start a failover process to elect a new leader that could start receiving writes again. No leaders, no failover, and if you remember what we've talked about failovers, you can probably see why this can be a big deal.

But, again, there is no free lunch, so let's take a look at the price tag here.

What happens if, say, your write succeeds in 2 replicas, but fails in 1 (maybe that server was being rebooted when you sent the write request)?
You now have 2 replicas with the new value and 1 with the old value. Remember, these replicas are not talking to each other, there's no leader handling any kind of synchronization.



Now if you read from this replica, BOOM, you get stale data.

To deal with this problem, a client will not read data from one replica, but also send requests to several replicas concurrently (like it did for writes). The replicas then return their values, and also some kind of version number, that the clients can use to decide which value it should use, and which it should discard.

We still have a problem, though. One of the replicas still has the old value, and we need to somehow synchronize it with the rest of the replicas (after all, replication is the process of keeping the *same* data in several places).

There are usually two ways to do that: We can make the client responsible for this update, or we can have another process that is responsible just for finding differences in the data and fixing them.

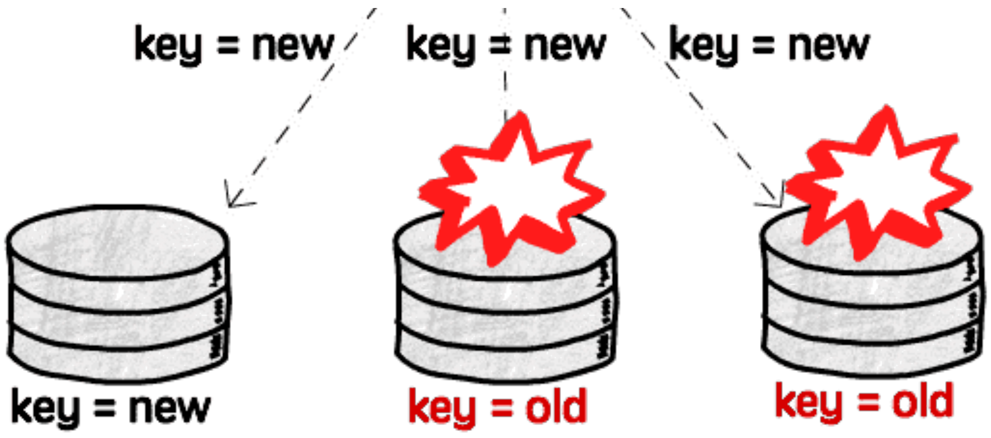
Making the client fix it is conceptually simple, when the client reads data from several nodes and detects that one of them is stale, it sends a write request with the correct value. This is usually called *read repair*.

The other solution, having a background process fixing the data, really depends on the database implementation, and there are several ways to do that, depending on how the data is stored. For example, `DynamoDB` has an *anti-entropy* process using Merkle trees.

Quorums

So we said we need to send the write/read requests to “some” replicas. There are good ways to define how many are enough, and what we are compromising (and gaining) if we decide to decrease this number.

Let’s first talk about the most obvious problematic scenario, when we require just one successful response to consider a value written, and also read from just one replica. From there we can expand the problem to more realistic scenarios.



As there is no synchronization between these replicas, we will read stale values every time we send a read request to a node other than the only one that succeeded.

Now let's imagine we have 5 nodes and require a successful write in 2 of them, and also read from 2. Well, we will have the exact same problem. If we write to nodes `A` and `B` and read from nodes `C` and `D`, we will always get stale data.

What we need is some way to guarantee that at least one of the nodes that we are reading from is a node that received the write, and that's what quorums are.

For example, if we have 5 replicas and require that 3 of them accept the write, and also read from 3 replicas, we can be sure that *at least* one of these replicas that we are reading from accepted the write and therefore has the most recent data. There's always an overlap.

Most databases allow us to configure how many replicas need to accept a write (`w`) and how many we want to read from (`r`). A good rule of thumb is to always have `w + r > number of replicas`.

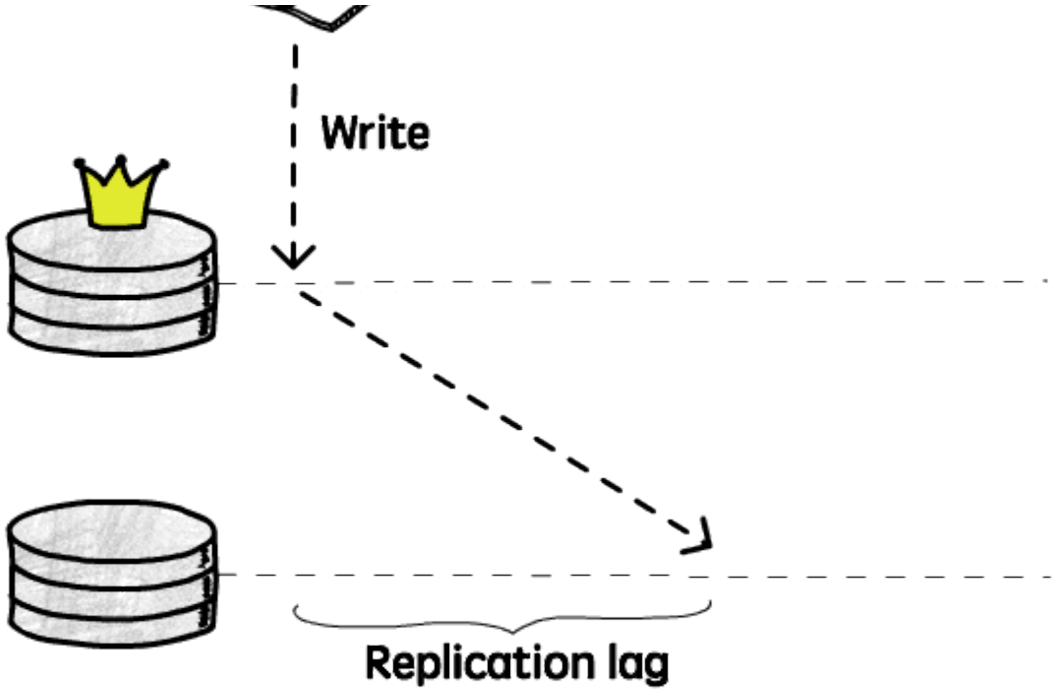
Now you can start playing with these numbers. For example, if your application writes to the database very rarely, but reads very frequently, maybe you can set `w = number of replicas` and `n = 1`. What that means is that writes need to be confirmed by every replica, but you can then read from just one of them as you are sure every replica has the latest value. Of course, you are then making your writes slower and less available, as just a single replica failure will prevent any write from happening, so you need to measure your specific needs and what is the right balance.

Replication lag

In a leader-based replication, as we have seen, writes need to be sent to a leader, but reads can be performed by any replica. When we have applications that are mostly reading from the database, and writing a lot less often (which is the most common case), it can be tempting to add many replicas to handle all these read requests, creating what can be called a *read scaling architecture*. Not only that, but we can have many replicas geographically close to our clients to also improve latency.

The more replicas we have, though, the harder it is to use synchronous replication, as the probability of one these many nodes being down when we need to replicate an update increases, and our availability decreases. The only feasible solution in this case is to use asynchronous replication, that is, we can still perform updates even if a node is not responding, and when this replica is back up it should catch up with the leader.

We've already discussed the benefits and challenges in using synchronous and asynchronous replication, so I'll not talk about that again, but assuming we are replicating updates asynchronously, we need to be aware of the problems we can have with the replication lag, or, in other words, the delay between the time an update is applied in the leader node and the time it's applied in a given replica.



If a client reads from this replica during this period, it will receive outdated information, because the latest update(s) were not applied yet. In other words, if you send the same query to 2 different server, you may get 2 different answers. As you may remember when we talked about the CAP theorem, this breaks the *consistency* guarantee. This is just temporary, though, eventually all the nodes replicas will get this update, and if you stop writing new data, they will all end up being identical. This is what we call *eventual consistency*.

In theory there is no limit for how long it will take to a replica to be consistent with its leader (the only guarantee we have is that *eventually* it will be), but in practice we usually expect this to happen fairly quickly, maybe in a couple of milliseconds.

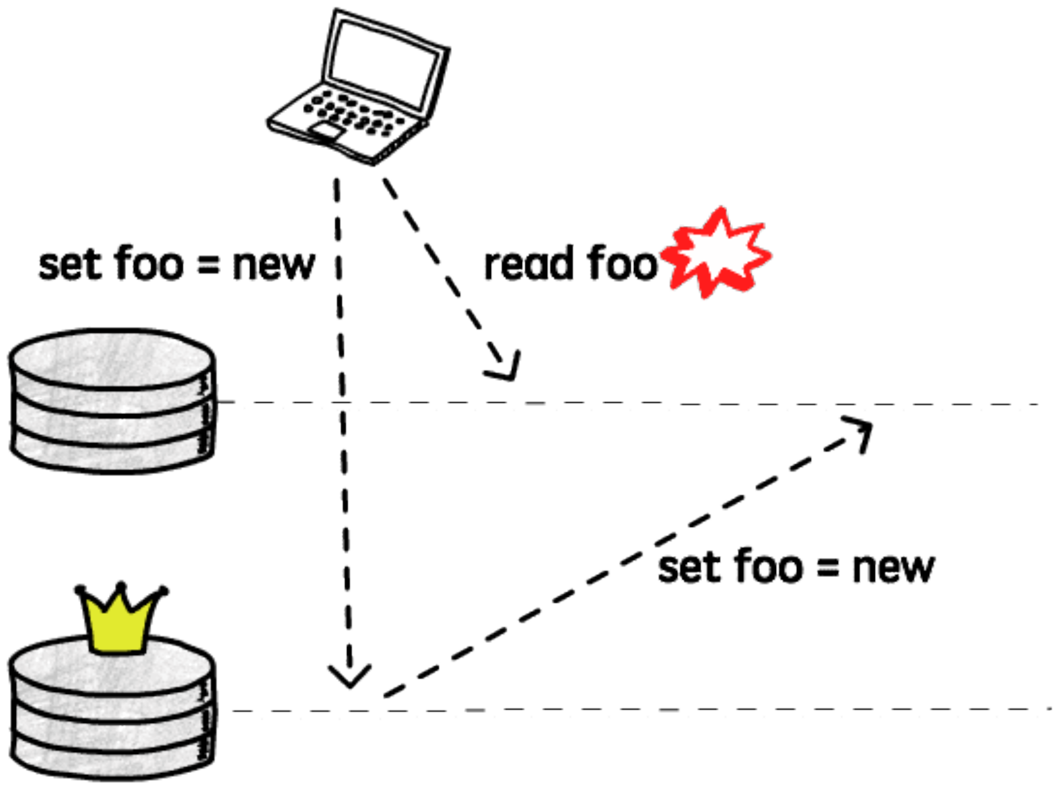
Unfortunately, we cannot expect that to always be the case, and we need to plan for the worst. Maybe the network is slow, or the server is operating near capacity and is not replicating the updates as fast as we'd except, and this replication lag can increase. Maybe it will increase a couple of seconds, maybe minutes. What happens then?

Well, the first step is to understand the guarantees we need to provide. For example, is it really a problem that, when facing an issue that increases the lag, it will take 30 seconds for your friend to be able to see that last cat picture you just posted on Facebook? Probably not.

In a lot of cases this replication lag, and eventual consistency in general, will not be a problem (after all, the physical world is eventually consistent), so let's focus on some cases where this *can* be an issue, and see some alternatives to handle them.

Read-your-writes consistency

The most common problem we can have with asynchronous replicas is when a client sends a write to the leader, and shortly after tries to read that same value from a replica. If this read happens before the leader had enough time to replicate the update, it will look like the write didn't actually work.



So, although it might not be a big issue if a client doesn't see other clients' updates right away, it's pretty bad if they don't see their own writes. This is what is called *read-your-writes consistency*, we want to make sure that a client never reads the database in a state it was before it performed a write.

Let's talk about some techniques that can be used to achieve this type of consistency.

A simple solution is to actually read from the leader when we are trying to read something that the user might have changed. For example, if we are implementing something like Twitter, we can read other people's timeline from a replica (as the user will not be able to write/change it), but when viewing their own timeline, read from the leader, to ensure we don't miss any update.

Now, if there are lots of things that can be changed by every user, that doesn't really work very well, as we would end up sending all the reads to the leader, defeating the whole purpose of having replicas, so in this case we need a different strategy.

Another technique that can be used is to track the timestamp of the last write request and for the next, say, 10 seconds, send all the read requests to the leader. Then you need to find the right balance here, because if there is a new write every 9 seconds you will also end up sending all of your reads to the leader. Also, you will probably want to monitor the replication lag to make sure replicas that fall more than 10 seconds behind stop receiving requests until they catch up.

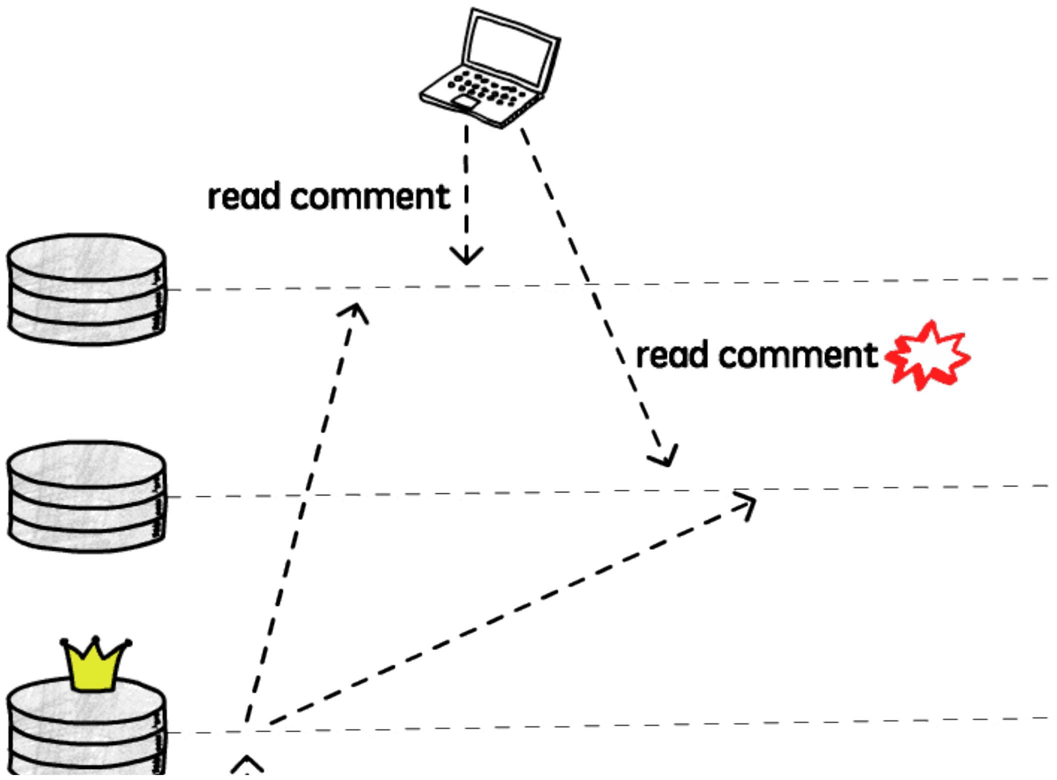
Then there are also more sophisticated ways to handle this, that requires more collaboration of your database. For example, Berkeley DB will generate a *commit token* when you write something to the leader. The client can then send this token to the replica it's trying to read from, and with this token the replica knows if it's current enough (i.e. if it has already applied that commit). If so, it can just serve that read without any problem, otherwise it can either block until it receives that update, and then answer the request, or it can just reject it, and the client can try another replica.

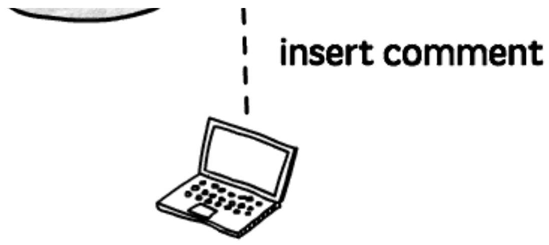
As always, there are no right answers here, and I am sure there lots of other techniques that can be used to work around this problem, but you need to know how your system will behave when facing a large replication lag and if *read-your-writes* is a guarantee you really need to provide, as there are databases and replication tools that will simply ignore this issue.

Monotonic Reads consistency

This is a fancy name to say that we don't want clients to see time moving backwards: If I read from a replica that has already applied commits 1, 2 and 3, I don't want my next read to go to a replica that only has commits 1 and 2.

Imagine, for example, that I'm reading the comments of a blog post. When I refresh the page to check if there's any new comment, what actually happens is that the last comment disappears, as if it was deleted. Then I refresh again, and it's back there. Very confusing.





Although you can still see stale data, what monotonic read guarantees is that if you make several reads to a given value, all the successive reads will be at least as recent as the previous one. Time never moves backwards.

The simplest way to achieve monotonic reads is to make each client send their read requests to the same replica. Different clients can still read from different replicas, but having a given client always (or at least for the duration of a session) connected to the same replica will ensure it never reads data from the past.

Another alternative is to have something similar to the commit token that we talked about in the *read-your-writes* discussion. Every time that a client reads from a replica it receives its latest commit token, that is then sent in the next read, that can go to another replica. This replica can then check this commit token to know if it's eligible to answer that query (i.e. if its own commit token is "greater" than the one received). If that's not the case, it can wait until more data is replicated before responding, or it can return an error.

Bounded Staleness consistency

This consistency guarantee means, as the name indicates, that there should be a limit on how stale the data we are reading is. For example, we may want to guarantee that clients will not read data that is more than 3 minutes old. Alternatively, this staleness can be defined in terms of number of missing updates, or anything that is meaningful the application.

Delayed replicas

We talked about replication lags, some of the problems that we can have when this lag increases too much, and how to deal with these problems, but sometimes we may actually *want* this lag. In other words, we want a *delayed replica*.

We will not really read (or write) from this replica, it will just sit there, lagging behind the leader, maybe by a couple of hours, while no one is using it. So, why would anyone want that?

Well, imagine that you release a new version of your application, and a bug introduced in this release starts deleting all the records from your `orders` table. You notice the issue and rollback this release, but the deleted data is gone. Your replicas are not very useful at this point, as all these deletions were already replicated and you have the same messy database replicated. You could start to restore a backup, but if you have a big database you probably won't have a backup running every couple of minutes, and the process to restore a database can take a lot of time.

That's were a delayed replica can save the day. Let's say you have a replica that is always 1 hour behind the leader. As long as you noticed the issue in less than 1 hour (as you probably will when your orders evaporate) you can just start using this replica and, although you will still probably lose some data, the damage could be a lot worse.

A replica will almost never replace a proper backup, but in some cases having a delayed replica can be extremely helpful (as the developer that shipped that bug can confirm).

Replication under the hood

We talked about several different replication setups, consistency guarantees, benefits and disadvantages of each approach. Now let's go one level below, and see how one node can actually send its data to another, after all, replication is all about copying bytes from one place to another, right?

Statement-based replication

Statement-based replication basically means that one node will send the same statements it received to its replicas. For example, if you send an `UPDATE foo = bar` statement to the leader, it will execute this update and send the same instruction to its replicas, that will also execute the update, hopefully getting to the same result.

Although this is a very simple solution, there are some things to be considered here. The main problem is that not every statement is deterministic, meaning that each time you execute them, you can get a different result. Think about functions like `CURRENT_TIME()` or `RANDOM()`, if you simply execute these functions twice in a row, you will get different results, so just letting each replica re-execute them would lead to inconsistent data.

Most databases and replication tools that use statement-based replication (e.g. `MySQL` before 5.1) will try to replace these nondeterministic function calls with fixed values to avoid these problems, but it’s hard to account for every case. For example, a user-defined function can be used, or a trigger can be called after an update, and it’s hard to guarantee determinism in these cases. `VoltDB`, for instance, uses logical replication but requires stored procedures to be deterministic.

Another important requirement is that we need to make sure that all transactions either commit or abort on every replica, so we don’t have a change being applied in some replicas and not in others.

Log Shipping replication

Most databases use a log (an append-only data structure) to provide durability and atomicity (from the *ACID* properties). Every change is first written to this log before being applied, so the database can recover in case of crashes during a write operation.

The log describes changes to the database at a very low level, describing, for example, which bytes were changed and where exactly in the disk. It's not meant to be read by humans, but machines can interpret them pretty efficiently.

The idea of log shipping replication is to transfer these log files to the replicas, that can then apply them to get the exact same result.

The main limitation that we have when shipping these logs is that, as it describes the changes at such a low level, we probably won't be able to replicate a log generated by a different version of the database, for example, as the way the data is physically stored may have changed.

Another issue is that we cannot use multi-master replication with log shipping, as there's no way to unify multiple logs into one, and if data is changing in multiple locations at the same time, that would be necessary.

This technique is used by `Postgres`' streaming replication and also to provide incremental backups and Point-in-Time Recovery.

This is also known as *physical replication*.

Row-based replication

Row-based, or *logical* replication, is kind of a mix of these two techniques. Instead of shipping the internal log (`WAL`), it uses a different log just for replication. This log can then be decoupled from the storage engine and therefore can be used, in most cases, to replica data across different database versions.

This row-based log will include enough information to uniquely identify a row, and a set of changes that need to be performed.

A benefit of using a row-based approach is that we can, for example, upgrade the database version with zero downtime. We can take one node down to upgrade it, and in the meantime the other replicas handle all the requests, and after it's back up, with the new version, we do the same thing for the other nodes.

The main disadvantage here when compared to a statement-based replication is that sometimes we need to log a lot more data. For example, if we want to replicate `UPDATE foo = bar`, and this update changes 100 rows, with the statement-based replication we would log just this simple `SQL`, while we would need to log all the 100 rows when using the row-based technique. In the same way, if you use an user-defined function that generates a lot of data, all that data needs to be logged, instead of just the function call.

`MySQL` for example, allows us to define a `MIXED` logging format, that will switch between statement and row-base replication, trying to use the best technique for each case.

Diving deeper

I hope this introduction to the different ideas and concepts behind database replication made you curious to learn more. If that's the case, here's the list of resources that I used (and am still using) in my own studies and can recommend:

- (Book) Designing Data-Intensive Applications
  This is one of the best books I've ever read. It covers lots of different topics related to distributed systems, and in the 5th Chapter the author focuses on replication. Although it's not specific for databases, most of the topics covered are applicable.
- (Book) Distributed Systems For Fun and Profit
  This is another book that is not really focused only on databases, but in distributed systems in general, but chapters 4 and 5 are focused on replication. It explains in a straightforward (and fun) way

a lot of important topics that were not covered in depth here.

- (Book) PostgreSQL Replication
  This book is, of course, focused on `PostgreSQL`, but it also presents some concepts that are applicable to other RDBMS. Also, it's a good way to see how these ideas can be applied in practice. It explains how to setup synchronous/asynchronous replication, WAL shipping, and use tools that use a variety of techniques (e.g. `Bucardo`, using triggers and BDR, using row-based replication that we discussed here).

- (Paper) Understanding Replication in Databases and Distributed Systems
  This paper compares the replication techniques discussed in the distributed systems and databases literatures. It first describes an abstract model and then examines how that applies to synchronous/asynchronous replication (that is called *lazy*/*eager* replication in the paper), and in a single-leader/leaderless setup (that is calls *active/passive* for distributed systems and *primary-copy/update- everywhere* for databases).

- (Paper) Dynamo: Amazon's Highly Available Key-value Store
  Dynamo is the key value data storage created by Amazon that popularized the idea of leaderless databases. This is the paper that explains (at least superficially) how it works. It's interesting to see how they handle conflicts (letting the clients resolve them) and also use some techniques not explored in this post, like sloppy quorums (as opposed to strict quorums) and hinted handoff to achieve more availability. The paper is 10 years old already and I'm sure a few things changed, but it's an interesting read anyway.

- (Paper) A Critique of the CAP Theorem
  This paper explains why the CAP theorem is not always the best tool to use when reasoning about distributed systems. The author explains the problems he sees with the CAP definitions (or lack of

definitions, in some cases). For example, *consistency* in CAP means a very specific kind of consistency (*linearizability*), but there's a whole spectrum of different consistency guarantees that are ignored.

- (Paper) Replicated Data Consistency Explained Through Baseball
  This paper describes six consistency guarantees and tries to define how each participant (scorekeeper, umpire, etc.) in a baseball game would require a different guarantee. What I think is interesting in this paper is that it shows that every consistency model can be useful depending on the situation, sometimes it's fine to live with eventual consistency, and sometimes you need stronger guarantees (e.g. read-your-writes, described above).
- (Paper) How VoltDB does Transactions
  There is a section in this paper that explains specifically how `VoltDB` handles replication. It's interesting to see a relatively new database using a kind of statement-based replication, and how they enforce determinism to avoid data inconsistencies.
- (Research report) Notes on Distributed Databases
  This research report from 79 (!) covers a lot of different topics, but the first chapter is dedicated to replication. It's amazing to see how little the problems that we face have changed, as the network constraints are still the same.
- (Article) Eventually Consistent
  This article, by Amazon's CTO Werner Vogels, is a very good introduction to what it means to be eventually consistency. He talks about the trade-offs that need to be made in order to achieve high availability in large scale systems (like the ones Amazon operates).
- (Article) Clocks and Synchronization
  This article explains why physical clocks are not reliable.The summary is: When building a distributed system, do not take it for granted that you can simply trust the clock on the machine that is

executing your code.

- (Article) CAP Twelve Years Later: How the “Rules” Have Changed
  This is an article written by Eric Brewer, the guy that first presented the CAP theorem in 2000, so it’s interesting to see what the author has to say 12 years later. Although the theorem can be useful to make us think about the trade-offs in a particular design decision, it can also be misleading in some cases.
- (Documentation) Berkeley DB Read-Your-Writes Consistency
  This section of the documentation explains how `Berkeley` achieves Read-Your-Writes consistency, using the commit token technique that we discussed here.
- (Documentation) VoltDB Database Replication
  This section of the documentation is dedicated to explain how VoltDB handles replication. It explains the two options available: single and multi-master (that they call one-way and two-way replication, respectively).

  Get PDF