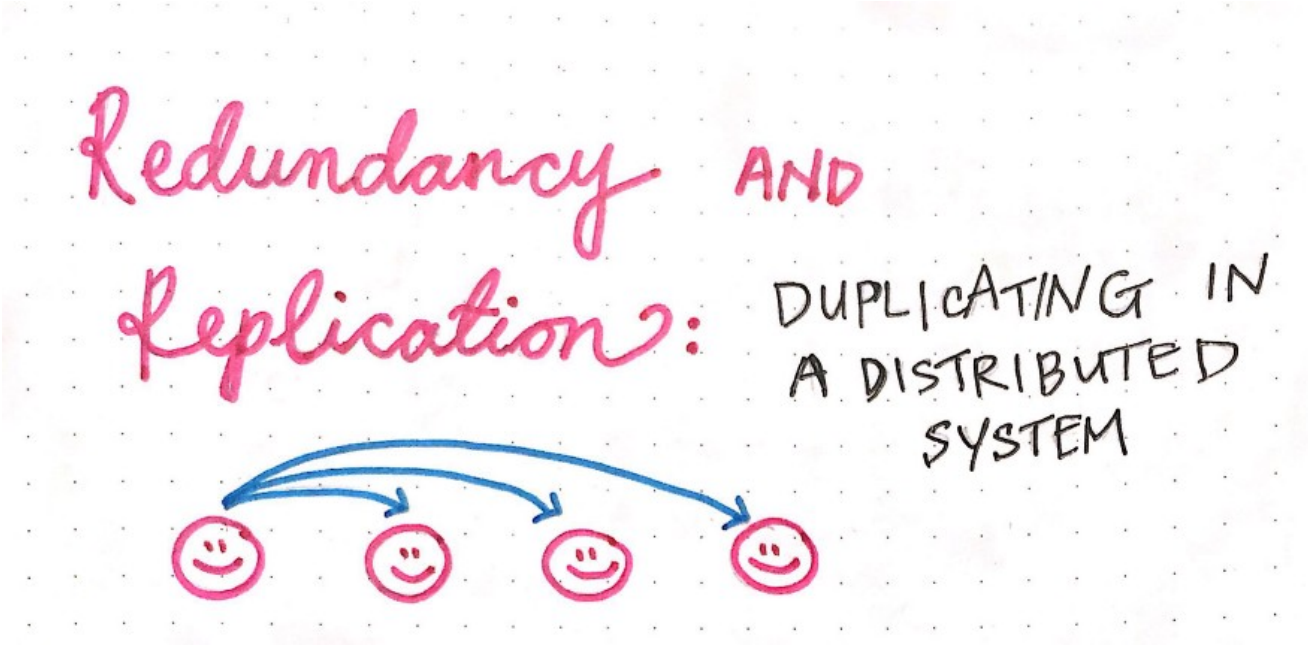
Redundancy and Replication: Duplicating in a Distributed System



Redundancy and replication: duplicating in a distributed system

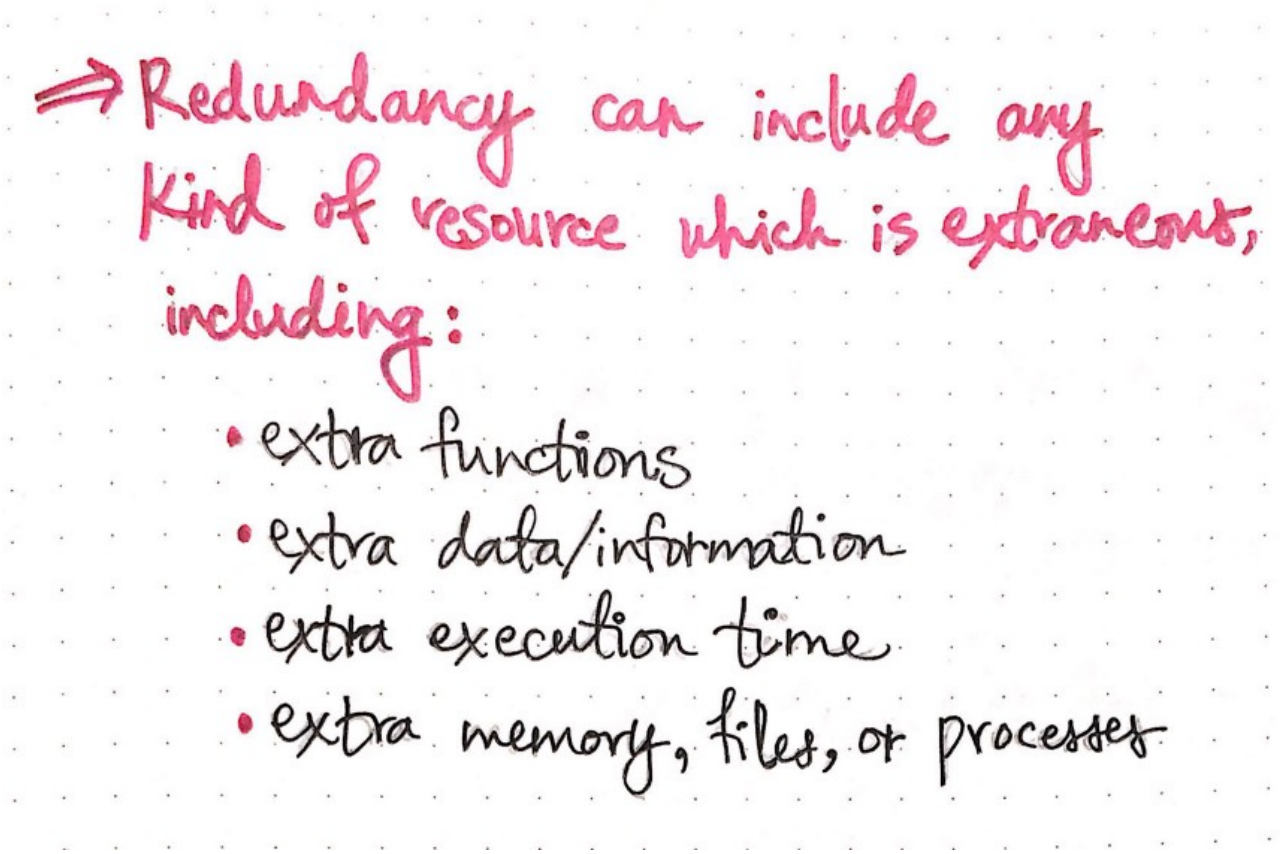
When it comes to programming, there are certain conventions, idioms, and principles that we run into and reference as a community quite often. One of those principles is the idea of “Don’t Repeat Yourself”, or DRY for short. I encountered this idea early on in my programming career, and it seemed pretty straightforward to me at the time: in order to maintain clean, concise code, it was important to ensure that one didn’t repeat the same lines or logic in our codebase.

But over the years of my career, I’ve learned and seen more, and realized that that repetition is not so cut and dry —no pun intended! Sometimes, it actually does make sense to repeat yourself at the risk of over-engineering or over-abstracting something unnecessarily. Sometimes, it makes sense to just duplicate a function or piece of logic and just “copypasta” it into another file.

And when it comes to distributed systems, it often makes a lot of sense to have many of one thing! In fact, as we start to look at duplication as a concept in distributed computing, the more obvious it becomes that repeating ourselves can actually be a huge benefit to our system as a whole. So let’s dig into duplication in a distributed system, and try to understand why we’d ever want to do it.

Repetition: sometimes a good thing?

When we talk about "repeating ourselves" in the the realm of distributed computing, we can mean many different things. Repetition of a line of code or a piece of logic usually indicates that we have more than one thing that isn't quite necessary. In a distributed system, we have a special term for this called ***redundancy***, which is all of the resources in a system that are not necessary for the system to actually function.

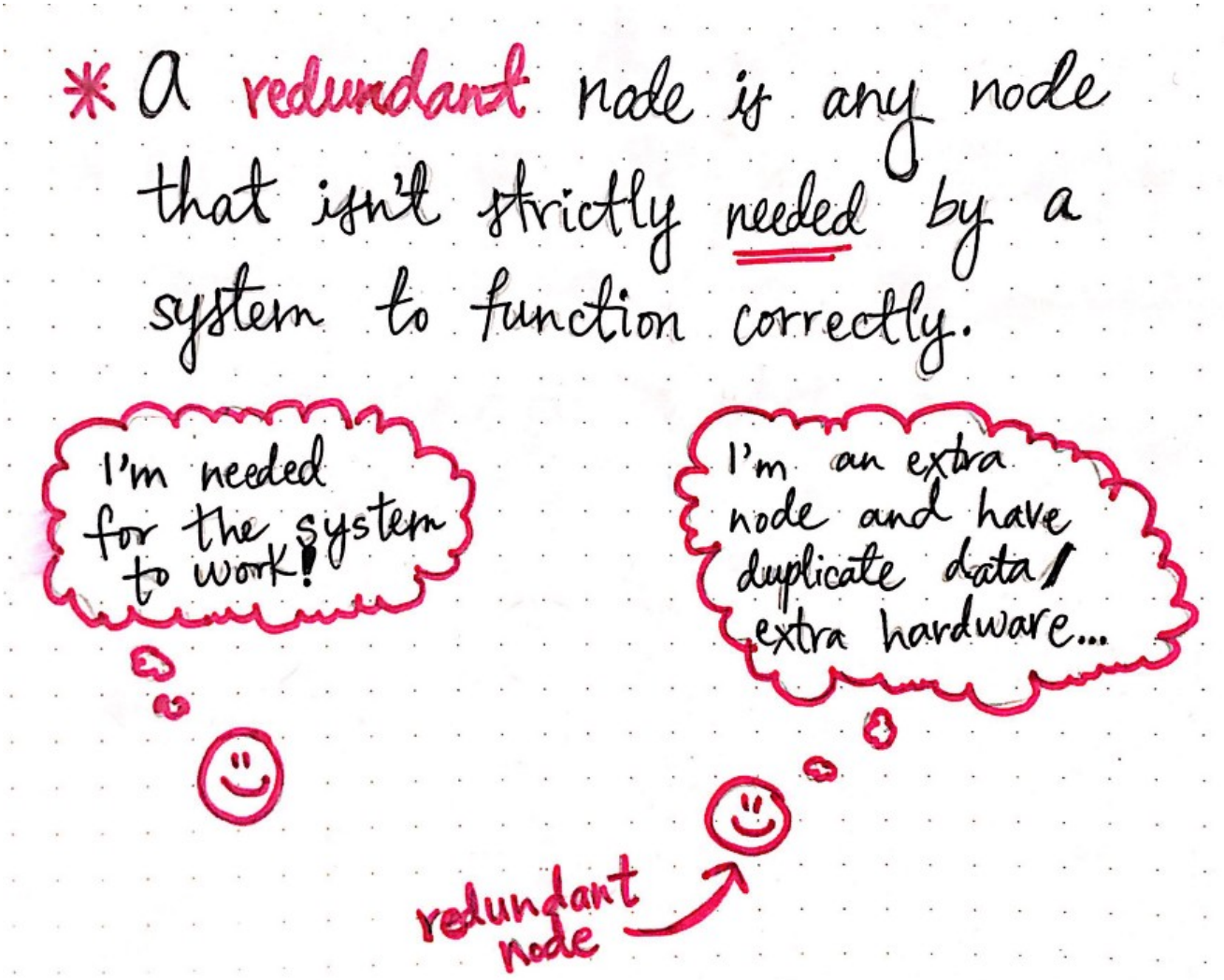


What can redundancy include?

However, in distributed computing, we could potentially have a whole lot of different resources, which means that we have many different things that could potentially be repeated! Thus, redundancy can encompass many different kinds of extraneous resources.

For example, repeated calls to the same function from the different places in the system, or duplicated references/variables to the same data are forms of redundancy. Similarly, duplicated execution time in a function as well as extra memory, files, or processes are also redundant, since the system doesn't actually need those duplicated resources to function correctly.

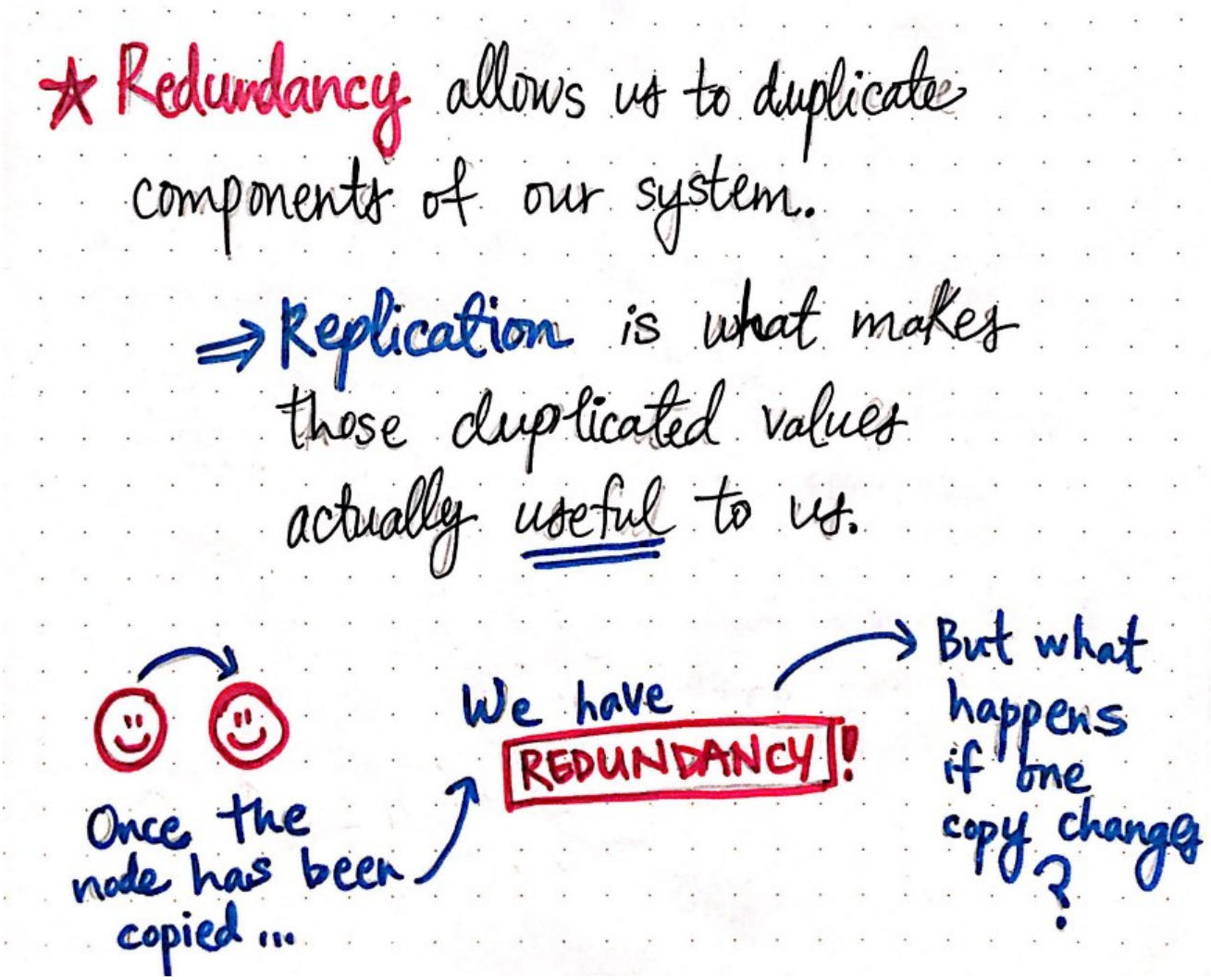
For the purposes of this series, we'll focus on redundancy in the context of extra memory, files or processes, which is often called ***structural redundancy***. In the context of a distributed system, some extra memory or file (read: a database) or an extra process all can be abstracted to a term that we're already familiar with: nodes.



Redundant nodes: a definition.

A redundant node is any node that isn't strictly necessary for the distributed system to function correctly. In other words, any node that adds to the bare minimum functionality of the system is extraneous, and we can therefore say that it is redundant.

Well, wait a minute — if we think about this a bit more deeply for a minute, we'll probably realize that we duplicate a lot of stuff in a distributed system! We've talked about adding nodes to a system pretty loosely in this series, but all of the nodes that we add can't all be *necessary* to the system, right? So, why did we even bother adding redundant nodes? Why would anyone want to be explicitly and intentionally redundant in their system? Well, in order to really answer that question, we need to move from the larger concept of redundancy to the more specific topic of replication.



Redundancy versus replication

While redundancy is what allows us to duplicate the components of our system, the duplication itself isn't really what is useful.

> Redundancy doesn't actually help us if a redundant node is out of sync with the node that it was copied *from*.

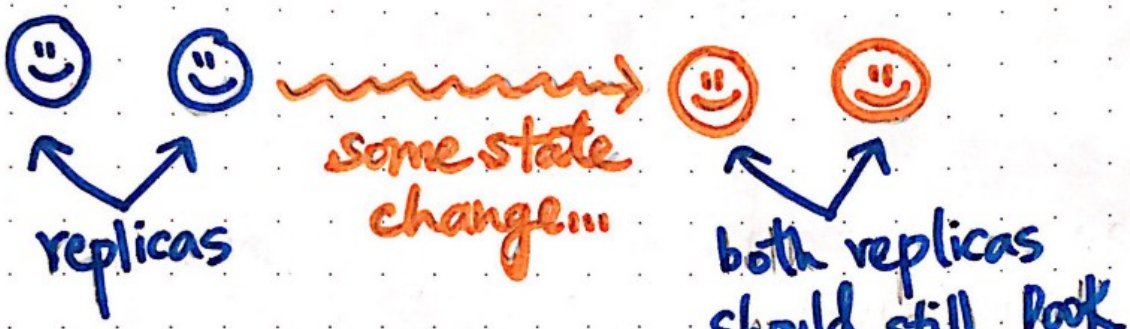
A redundant node on its own isn't all that helpful; sure, we have achieved redundancy, but who knows how long that redundant node will actually be functional for us in our system? What happens if one of the nodes changes in value or state? As it turns out, the answer to both of these questions is replication! When we replicate a node in our system, these duplicate copies actually become much more functional for us.

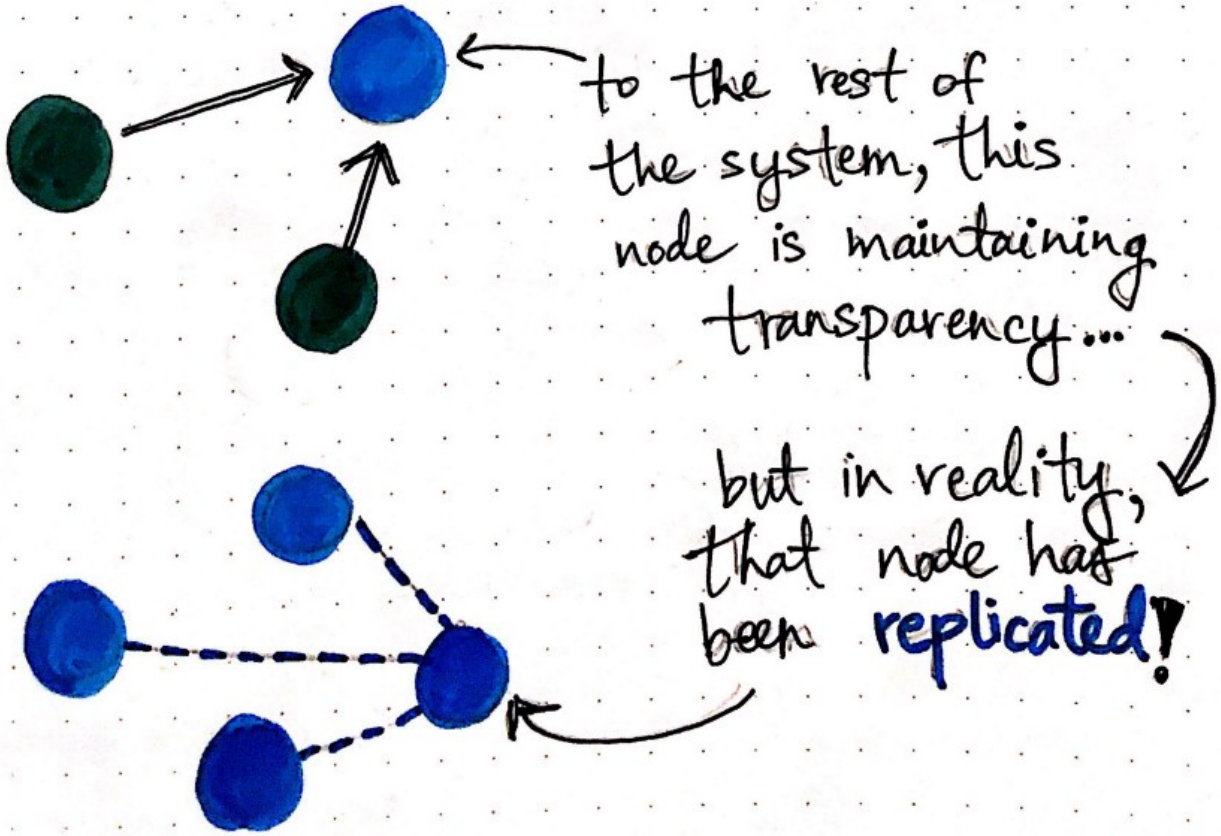
Duplication and maintaining state

Replication as a concept is much of like a subset of redundancy. Both of them involve creating redundant nodes in a system — nodes that aren't strictly necessary for the system to work correctly.

However, ***replication*** takes a redundant node one step further; it ensures that the redundant node (a ***replica***) is identical to all of its other copies. At first glance, this might seem simple and perhaps even obvious; we might even assume that a node that has been copied from somewhere else will be identical to the thing it was copied from, right?

* **Replication** is the process of creating a redundant node(s), and ensuring that it is identical to all other copies.





Replicated nodes: a definition

Well, not necessarily! In fact, let's say that we copied a node and created a redundant one. But later on, we change something about the original node — perhaps we change its state, some value that it contains, or some internal behavior. What would happen to the redundant node? Well…nothing!

If we created just a simple redundant node, there is nothing in place within our system that will ensure that the redundant node will 1) know that something changed and 2) will update itself or be updated by someone else to have the correct state.

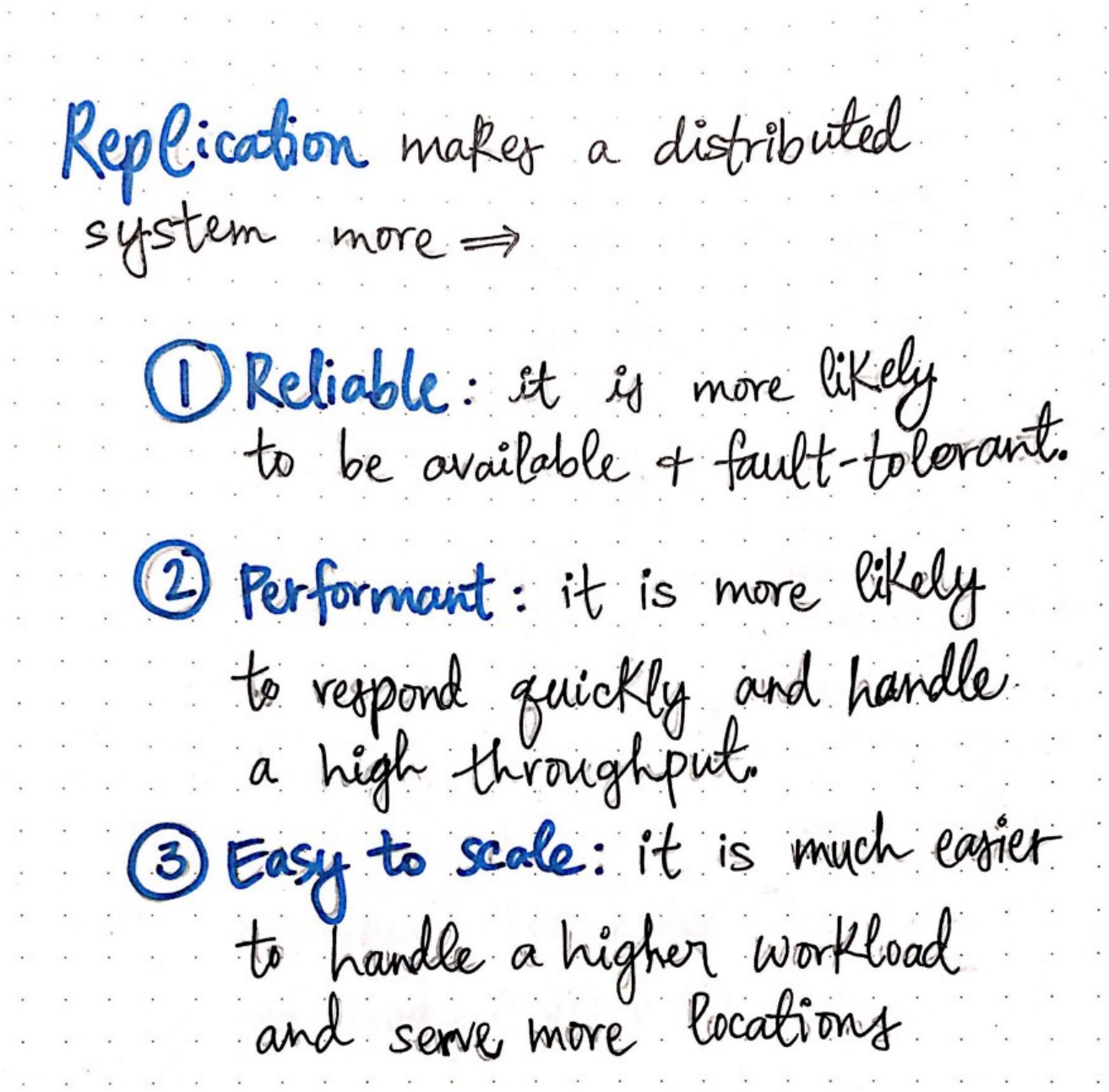
So perhaps this problem is not as obvious as we first thought! Replication attempts to solve this problem, because it ensures that all replicas of a node will be identical to one another, and will match the original node that they were replicated from. (Side note: the question of *how* to ensure this is no easy task, but we'll learn more about that later on in the series…)

Another important thing to note here is that replication comes hand in hand with ***transparency***, a concept that we're already familiar with. With each replica that is added to a system, the rest of the system should still function correctly, and should ideally be unaware of any replicas that have been created.

> If a replica is updated, the rest of the system shouldn't know about it, and if a node is replicated, the rest of the system shouldn't really care.

For example, if a web server or a database node is replicated, the end users of the distributed system ought not to have any idea that are interacting with a replica or the original node. (In fact they probably shouldn't have any idea that there even *is* a replicated node!)

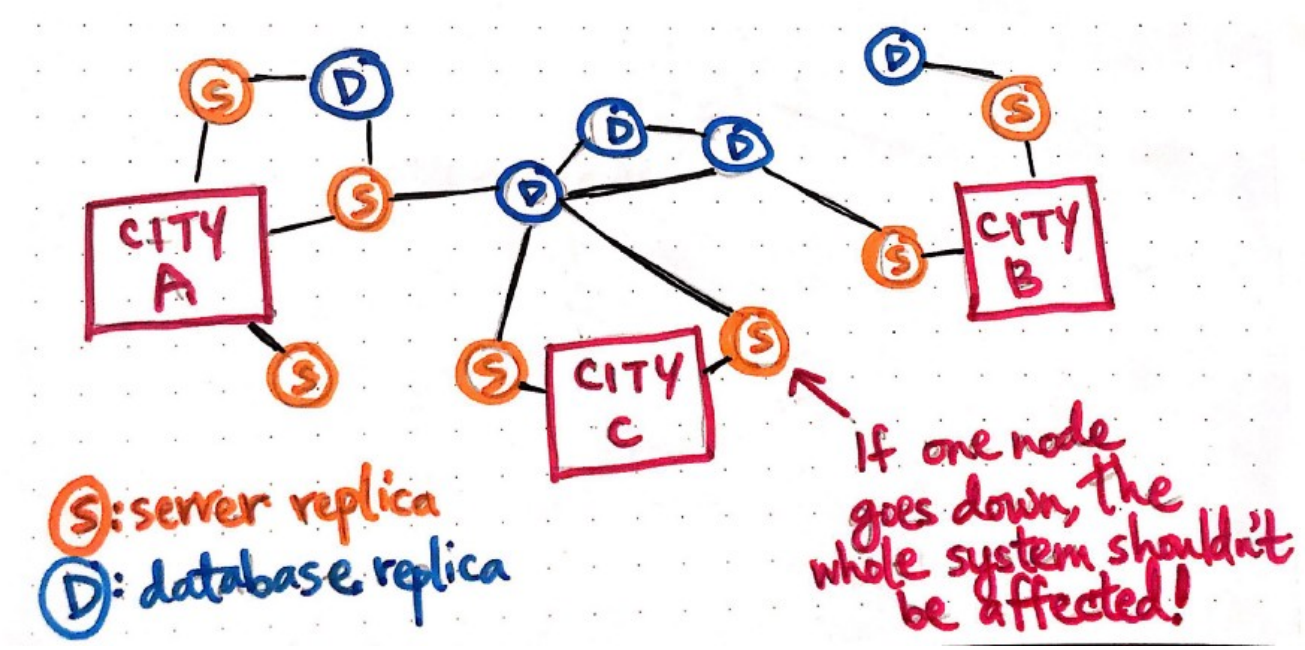
When this transparency is mostly maintained in a system, replication provides us with a whole lot of benefits:



Some benefits of using replication in a distributed system.

1. Replication of nodes makes our system is much more ***reliable.*** If one node goes down and we have already added replicas, it's much more likely for a replica to step in and do the job of the failing node. This makes our system more fault-tolerant. Having replicas in place makes our system more available as well, since overall there are more backup replicas to step in to take another node's place, which means that our system shouldn't experience too much downtime.
2. Replication of nodes also makes our system more ***performant***. With more replicas in place, we now have the ability to do more work, serve more requests, and process more data. This makes our system just generally faster, reduces latency, and allows the system to handle a high throughput of data that has to be processed/delivered.

3. Replication of nodes makes our system much ***easier to scale***. Scalability is a big plus point when it comes to building a distributed system, but it's not always easy to achieve. With more replicated nodes, it's is easier to scale a system by adding more databases, servers, or services as required by the demands on the system. It's also easier to geographically scale a system when we can replicate a node and place it in a different continent or country to help scale a high workload in a specific location.



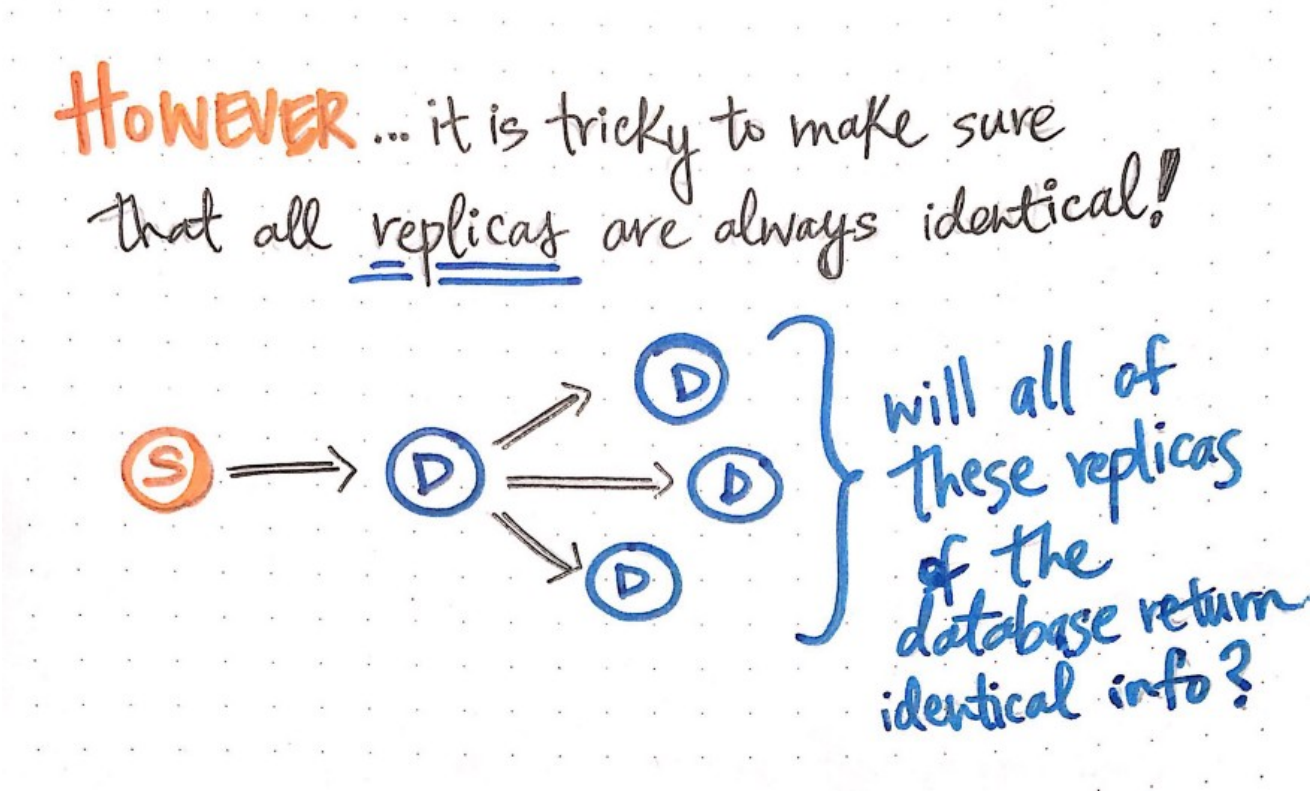
Reliability, performance, and scalability through replication

By this point, hopefully we're all on board with the idea of replication (right?). It's pretty cool and nearly every distributed system will take advantage and leverage this concept to help deal with issues of reliability, performance, and fault-tolerance. Whether in the form of workers, servers, databases, or services, we've probably each worked with replicas in our daily lives as developers! Sounds great, right?

Well, almost. The story doesn't quite end here. Replication is awesome, but it presents some unique challenges unto itself. I can't promise any solutions in this post (don't worry, we'll talk a lot more about this as the series goes on), but at the very least, I'll present you with some of the problems. 😉

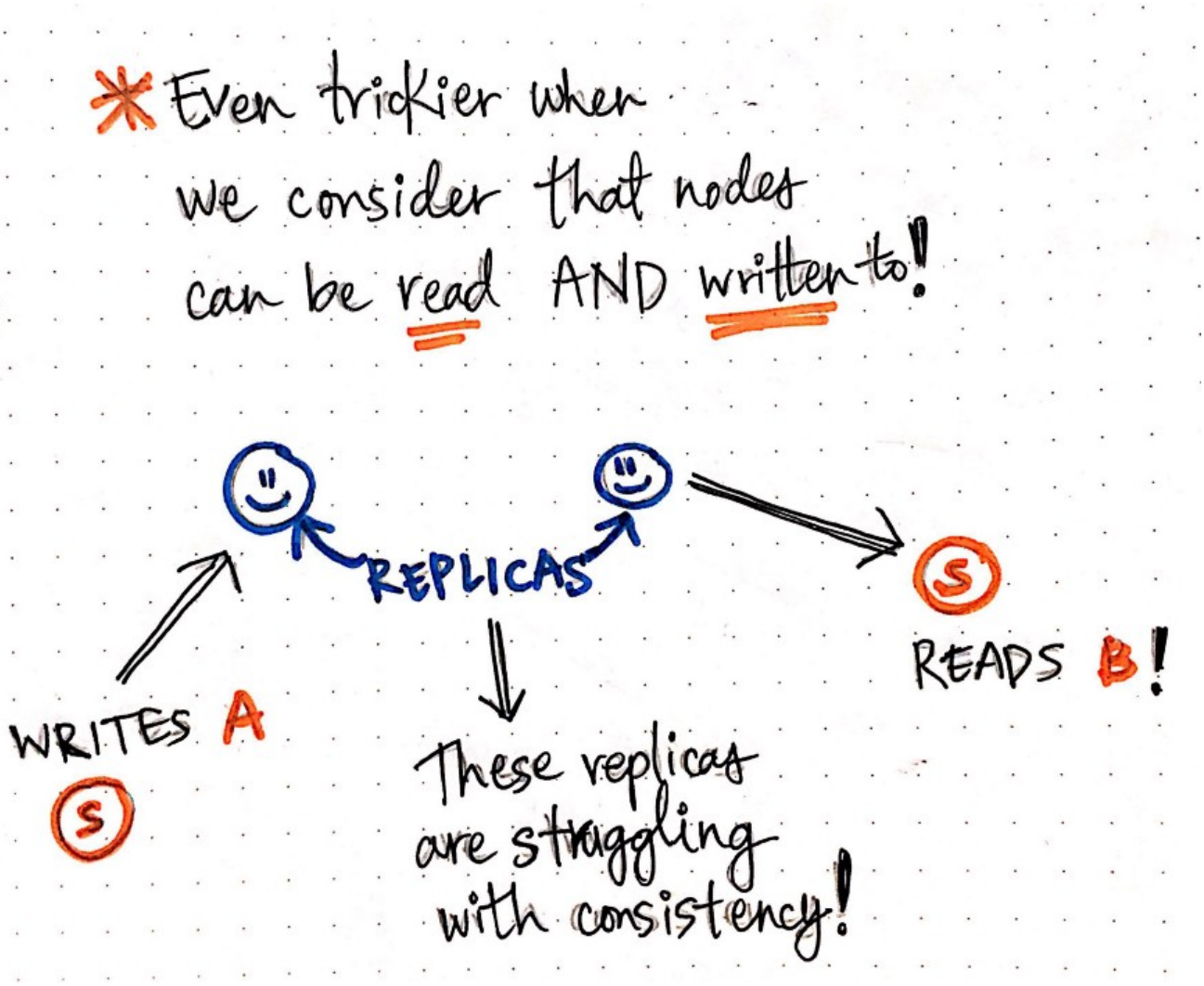
Consistent headaches

In addition to transparency, the concept of replication needs one other thing to really work well: ***consistency***. In order for a replica to actually be up-to-date with all the other replicas in the system, it needs to be consistent with the other copies. And this is not a trivial task!



What makes replication hard?

Now perhaps, at first glance, this doesn't seem too difficult. Maybe we could always have the replicas get updates from the original copy? Or maybe the original copy can tell the replicas whenever something else changes? However, this is tricker than it sounds.



It's a struggle to maintain consistency as replicas change state!

We learned earlier that a system must also be transparent for replication to work. Transparency in a system means that all the replicas and the original node must behave similarly, which means that a consumer of the system (like an end user or another node) could potentially write to one replica, while another consumer of the system could read from *another* replica!

So, we can't rely on just the original node anymore as our single source of truth when it comes to the state of our node(s). The question here is, of course, how do we tell these two replicas what just happened? The replica that is being read-from needs to know about the latest write; similarly, the replica that is being written-to needs to disperse information about that write.

This is exactly what makes replication so hard; we have to contend with consistency and figure out a model for how to ensure that all of our replicas. Thankfully, we'll cover that in much more detail later. For now, we just need to know that it's a problem that we'll have to deal with soon. sUntil then, feel free to replicate your little hearts away!