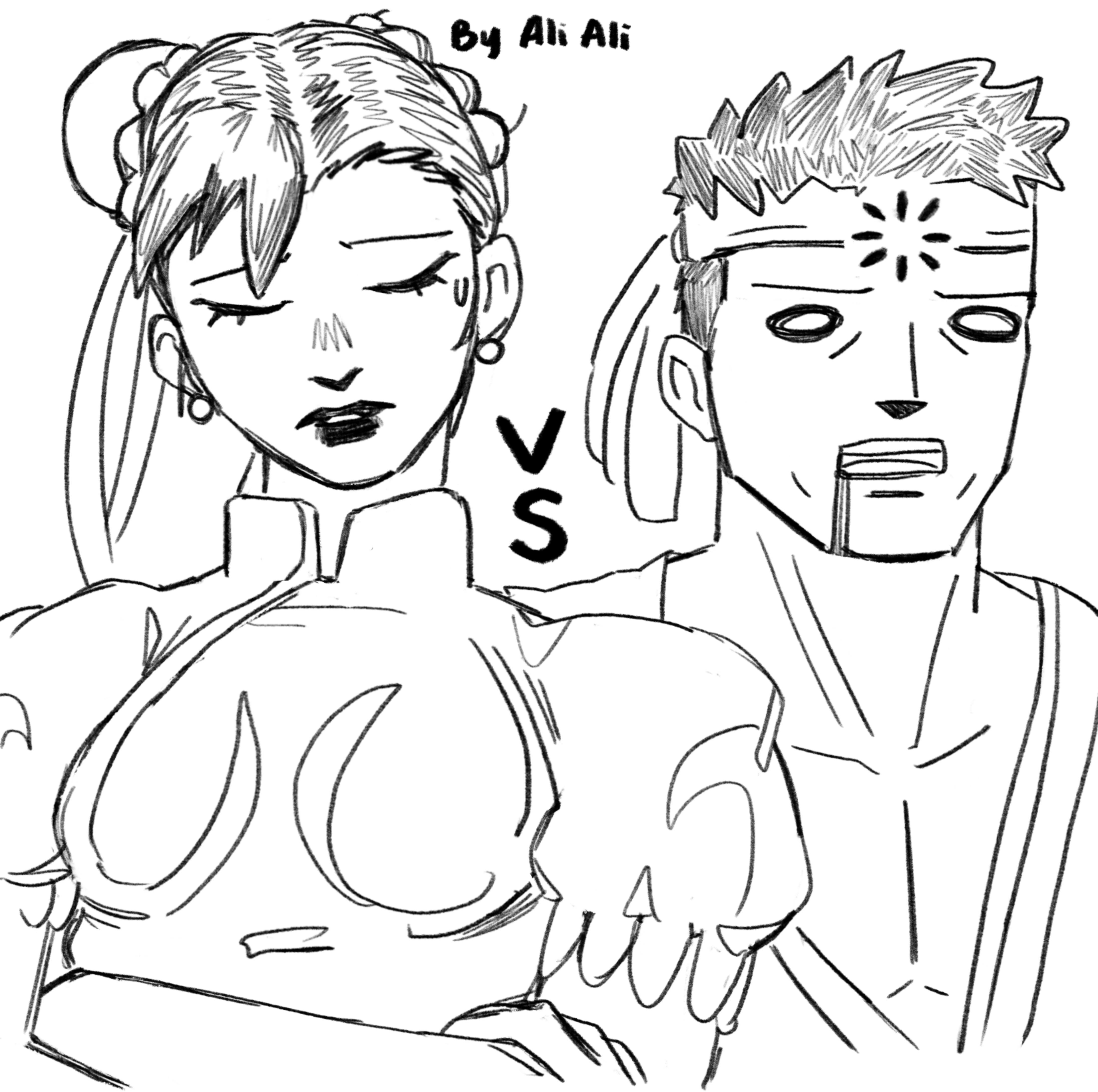


FIGHTING ⚡ ⚡ FAULTS IN DISTRIBUTED SYSTEMS

By Ali Ali



Small note: It was during the making of this that I realized just how important the study of distributed systems is in the context of fighting games (specifically how netcode is handled). Unfortunately, I didn't include any examples too involved in this zine for simplicity, but I highly recommend checking that topic out!

WHAT ARE DISTRIBUTED SYSTEMS?

Ever tried to access a website, game, or any other server-based application, and were met with it being unresponsive, slow, or outright malfunctioning?

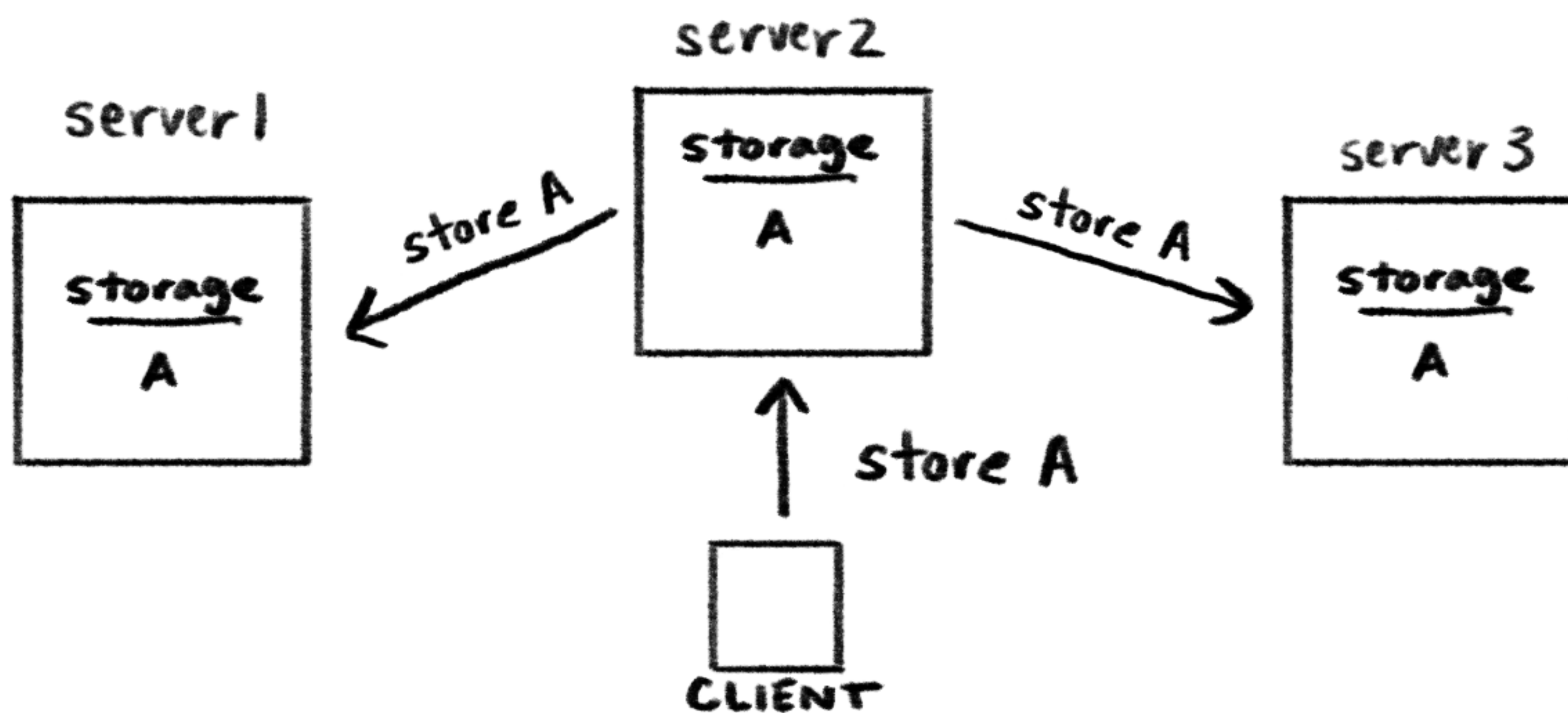


It makes you wonder...

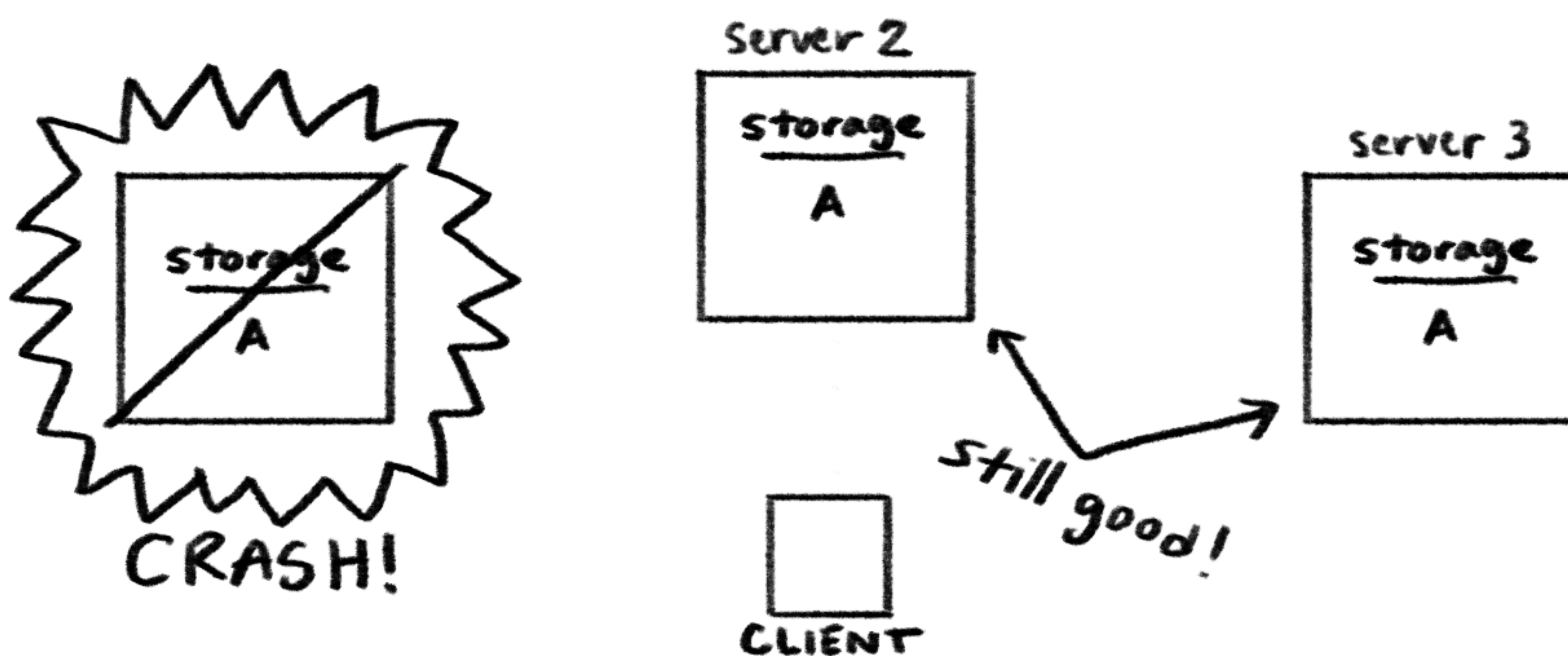


Good news! This already exists!
Dealing with these issues is what
distributed systems is for.

A distributed system is a system made up of independent components that communicate over a network with messages.



This practice solves a lot of real world problems, as if any of these components fail, they fail independently.



The example above demonstrates an environment where crashes are possible. However, this is not the only fault that can occur to a server.

In this zine, I hope to cover some of the important types of faults, as well as give an introduction to the techniques used in distributed systems that allows us to tolerate them.

SAFETY and LIVENESS PROPERTIES

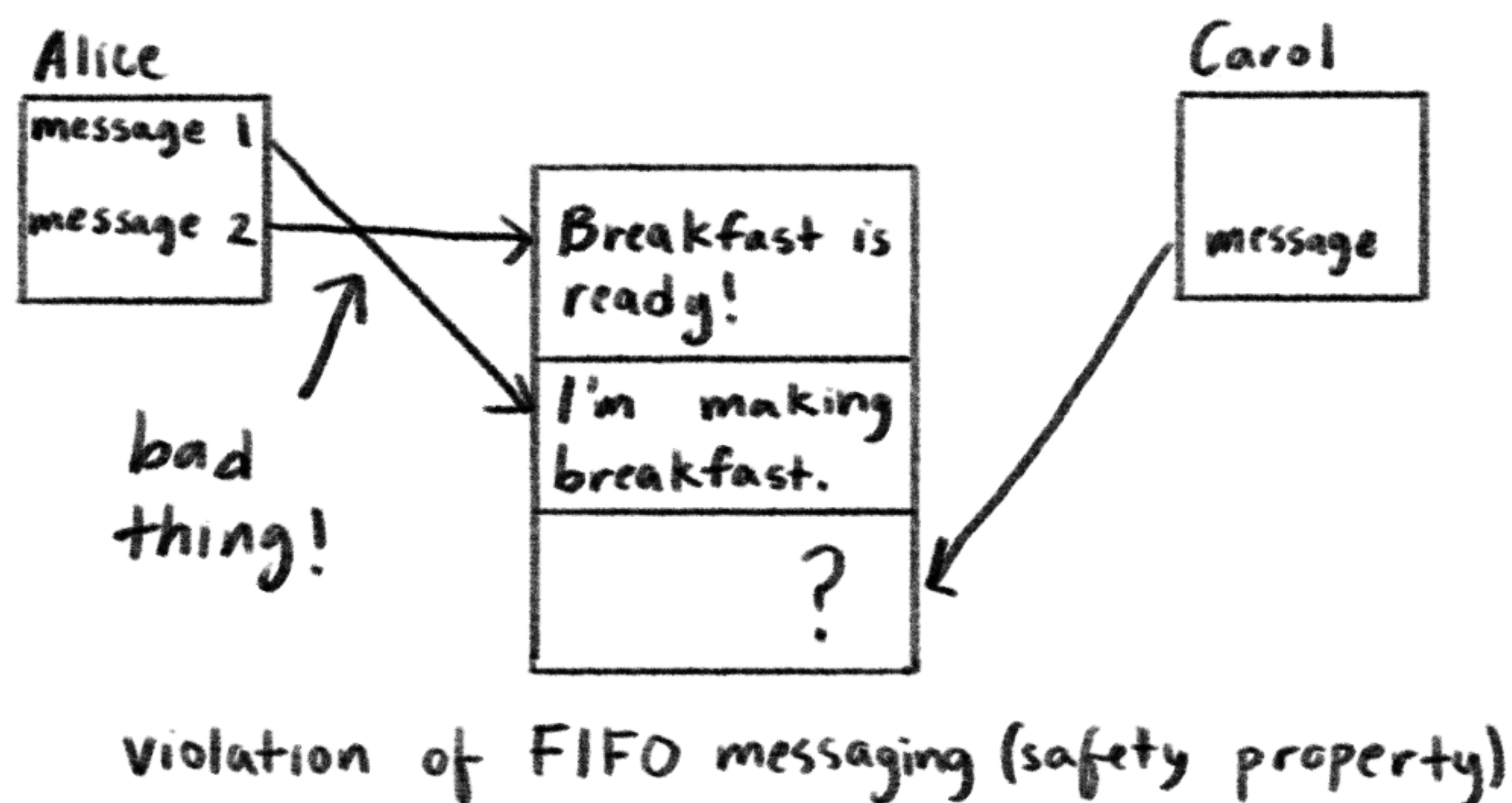
To understand faults, we should first understand safety and liveness properties. These properties help us define what the correct behavior of a system is. We can't know what's wrong if we don't know what's right, right?

Ensuring safety properties is ensuring that nothing "bad" will happen in how our program runs. They help us define which states are incorrect for our system.

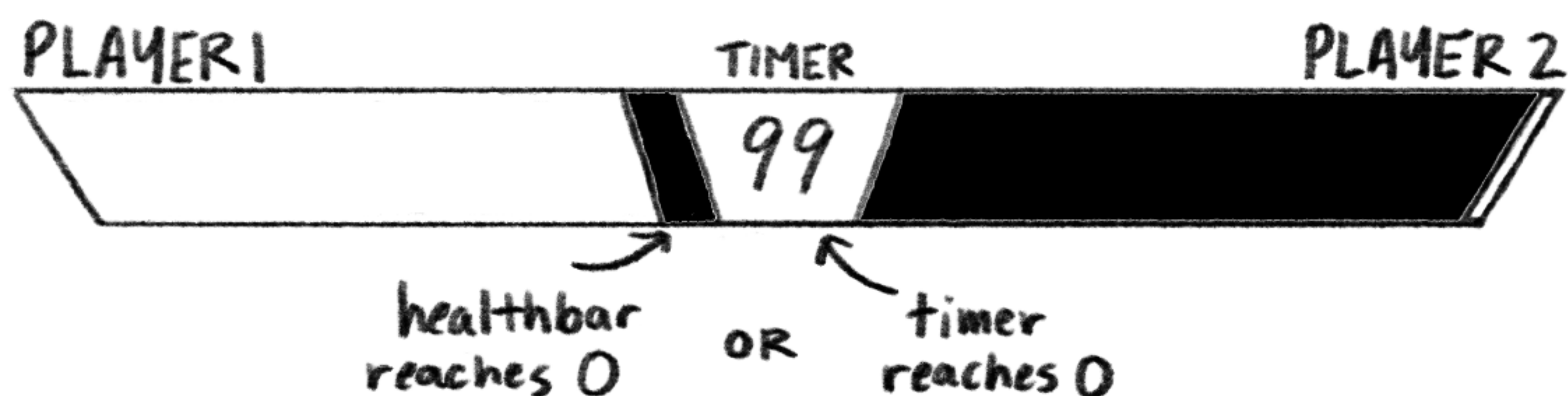


Here, we input a kick. However, for whatever reason, the player punches instead. Because our execution resulted in an incorrect state, it violated our safety property.

In the context of distributed systems, safety violations can include data corruption or a violation of consistency between message sends.



Liveness properties ensure that something “good” will eventually happen. For example, let’s make sure that a round of our fighting game will always end at some point.



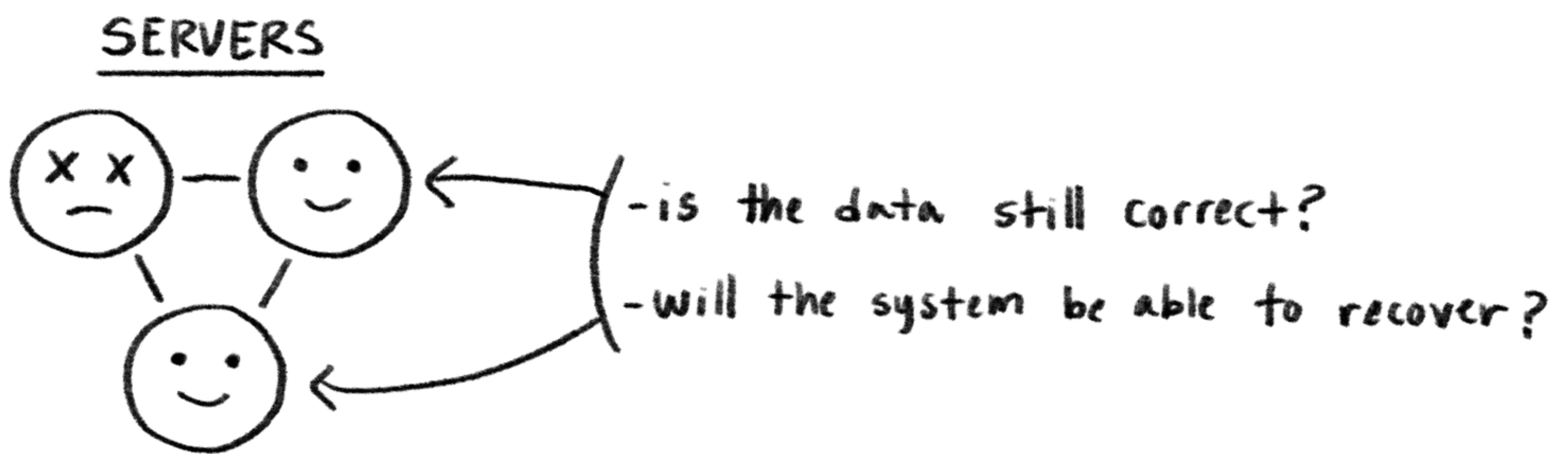
We can consider the match ending as a “good thing”, because it’s something we always want to eventually happen. Our timer handles this liveness property.

Similar logic can be applied to our programs; we might always want our executions to terminate, so we’d want to make sure there are no infinite loops, or anything else that might prevent this.

FAULT MODELS

Fault models provide us with a clear understanding of the different types of faults that can occur in our distributed system.

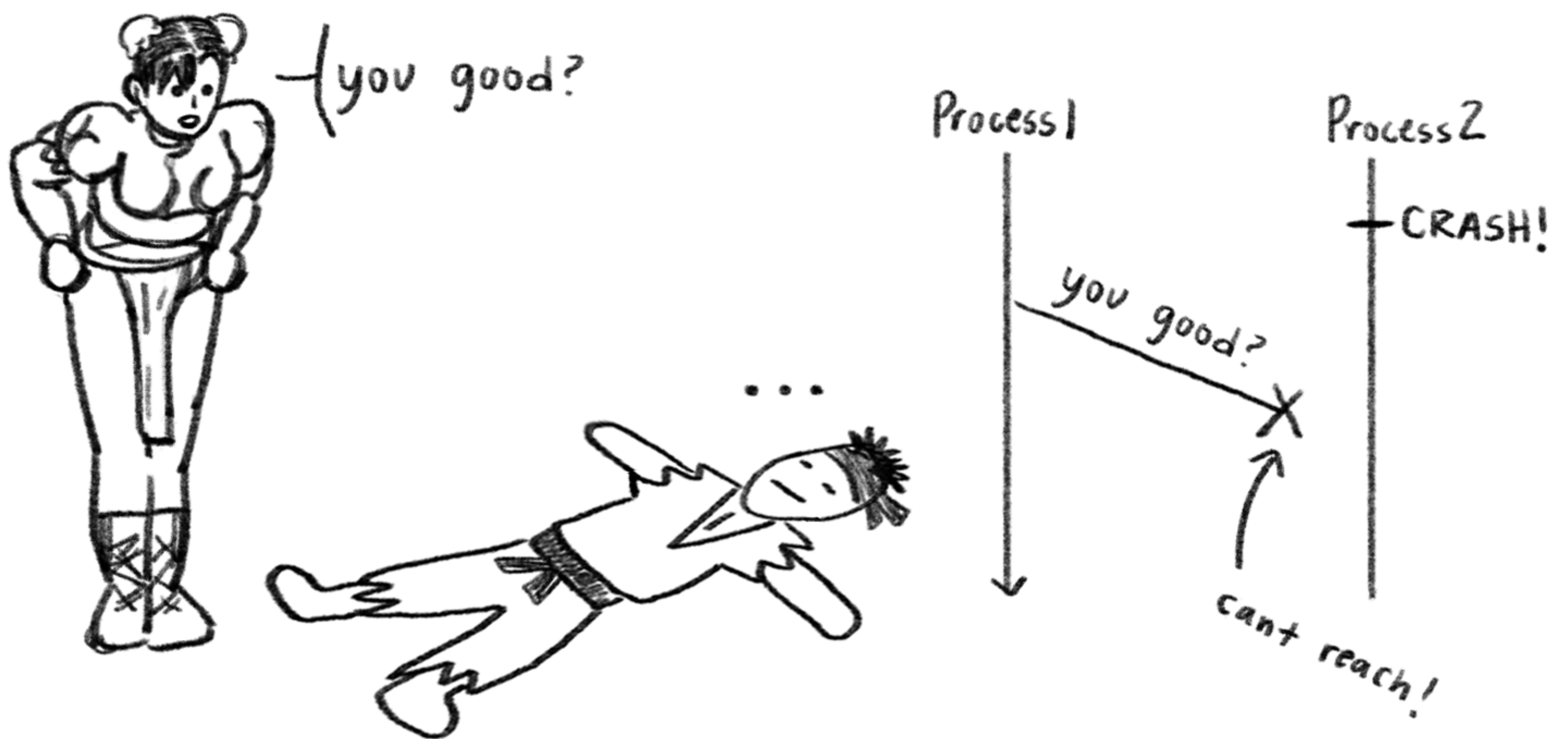
When we know what faults we might encounter, we can design our system with respect to them. We do this by ensuring our safety and liveness properties hold even if a fault occurs.



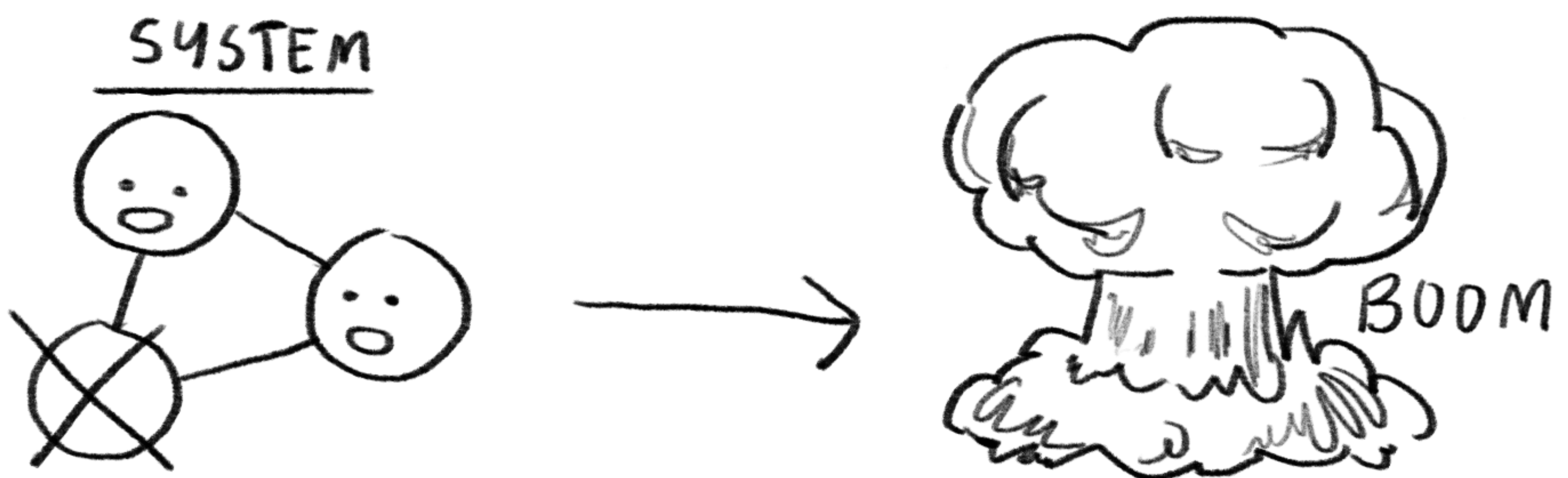
There are many different kinds of faults, and the classification of one fault may encapsulate another. The main classifications are crash faults, omission faults, timing faults, and Byzantine faults.

We'll begin by defining each one.

Crash faults - a crash fault occurs when a process fails by halting all operations. In this case, the process will no longer send or receive messages.

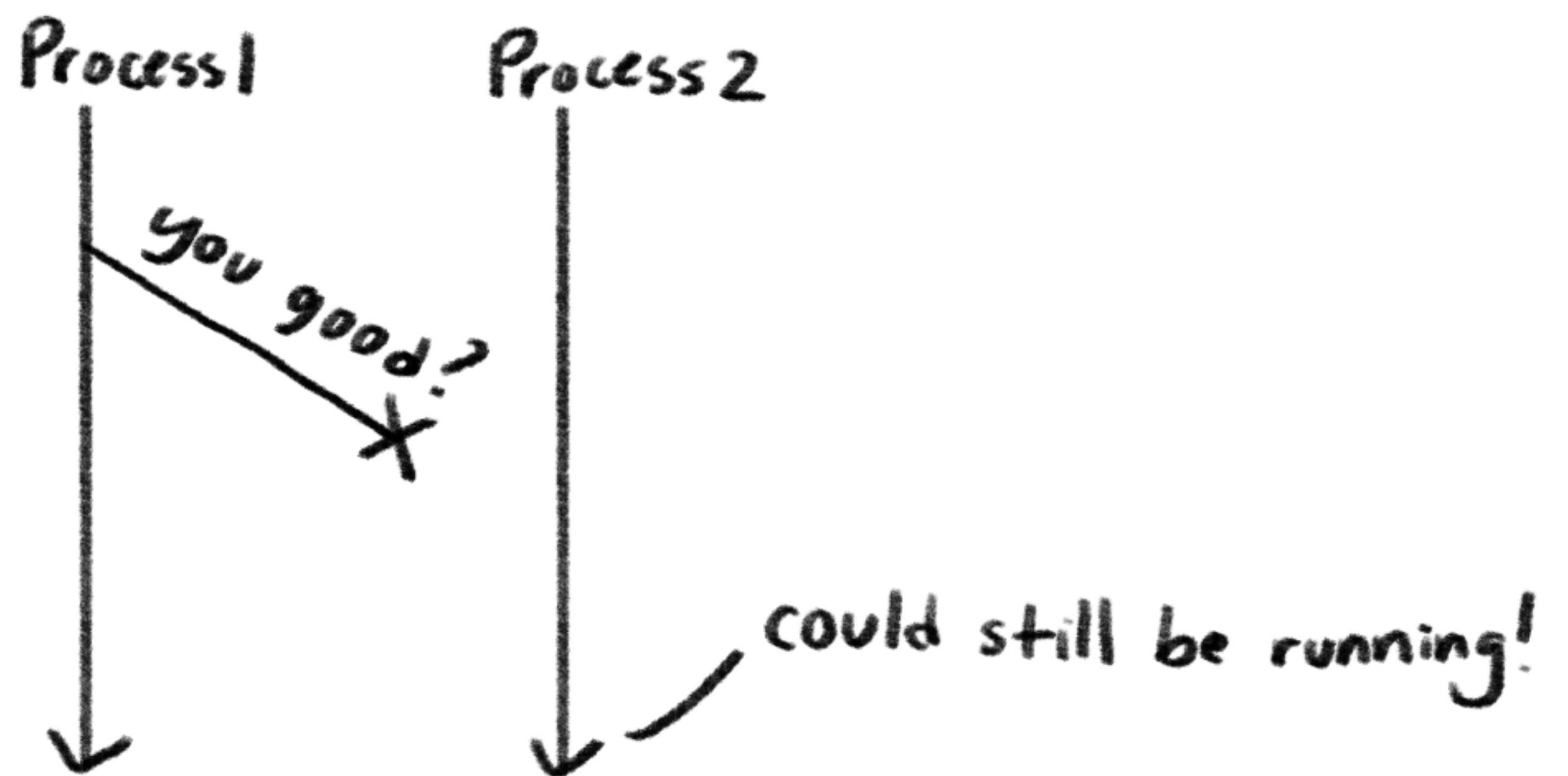


This is very bad for a distributed system!!!
Remember: our system is comprised of components that must communicate through messages!



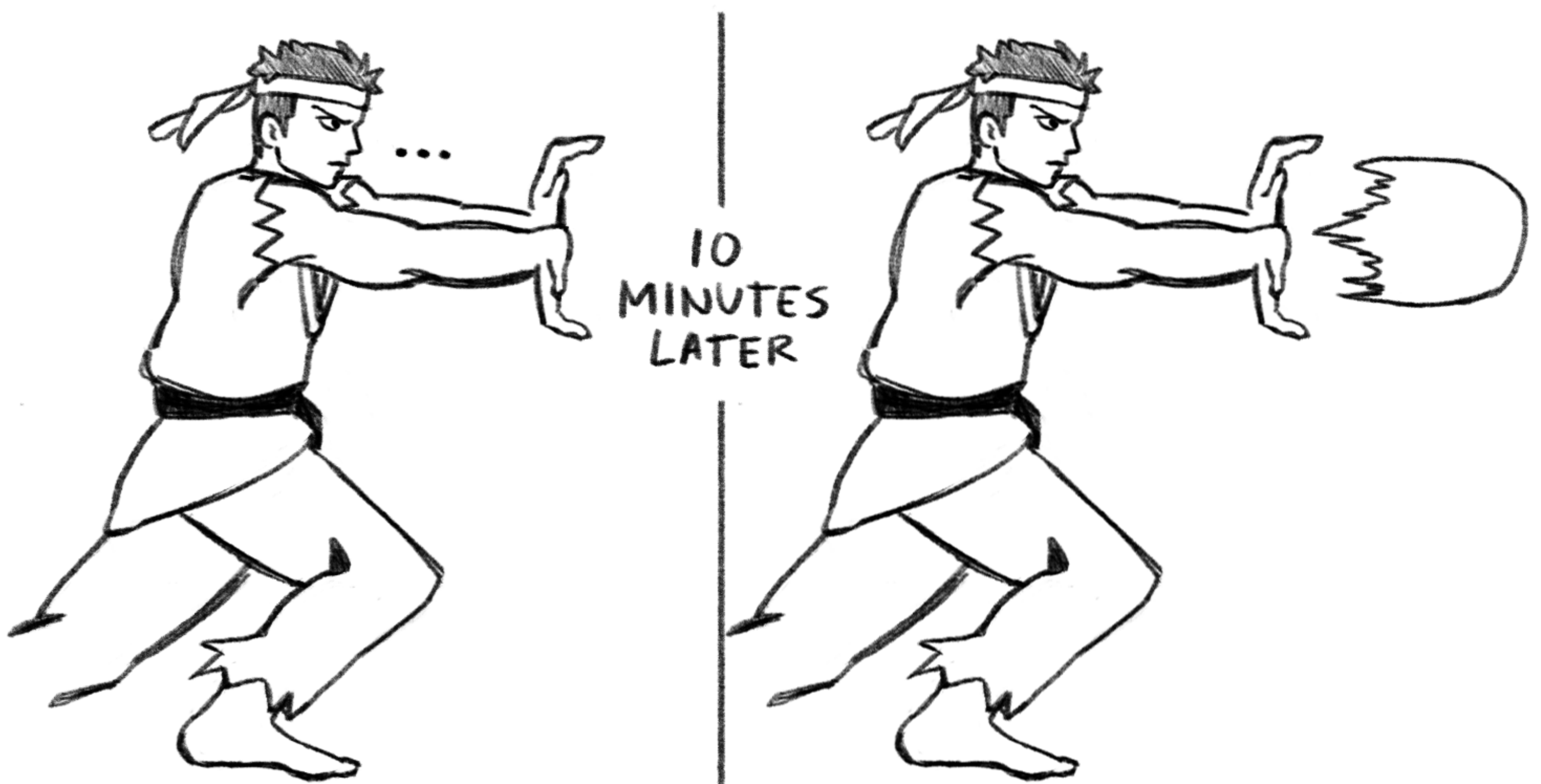
If we don't have a way to deal with these faults, it could ruin our whole system! But don't panic yet, we'll come back to this.

Omission faults - omission faults occur simply when a message is lost. In this case, a process just fails to send or receive a message.



Sounds pretty similar right? That's because crash faults actually count as omission faults. A message sent to a crashed server will always be omitted, so you can just think of it as an omission fault that will always occur so long as the server is down.

Timing faults - timing faults occur when an event occurs too early or too late. For example, a process might respond too slowly.



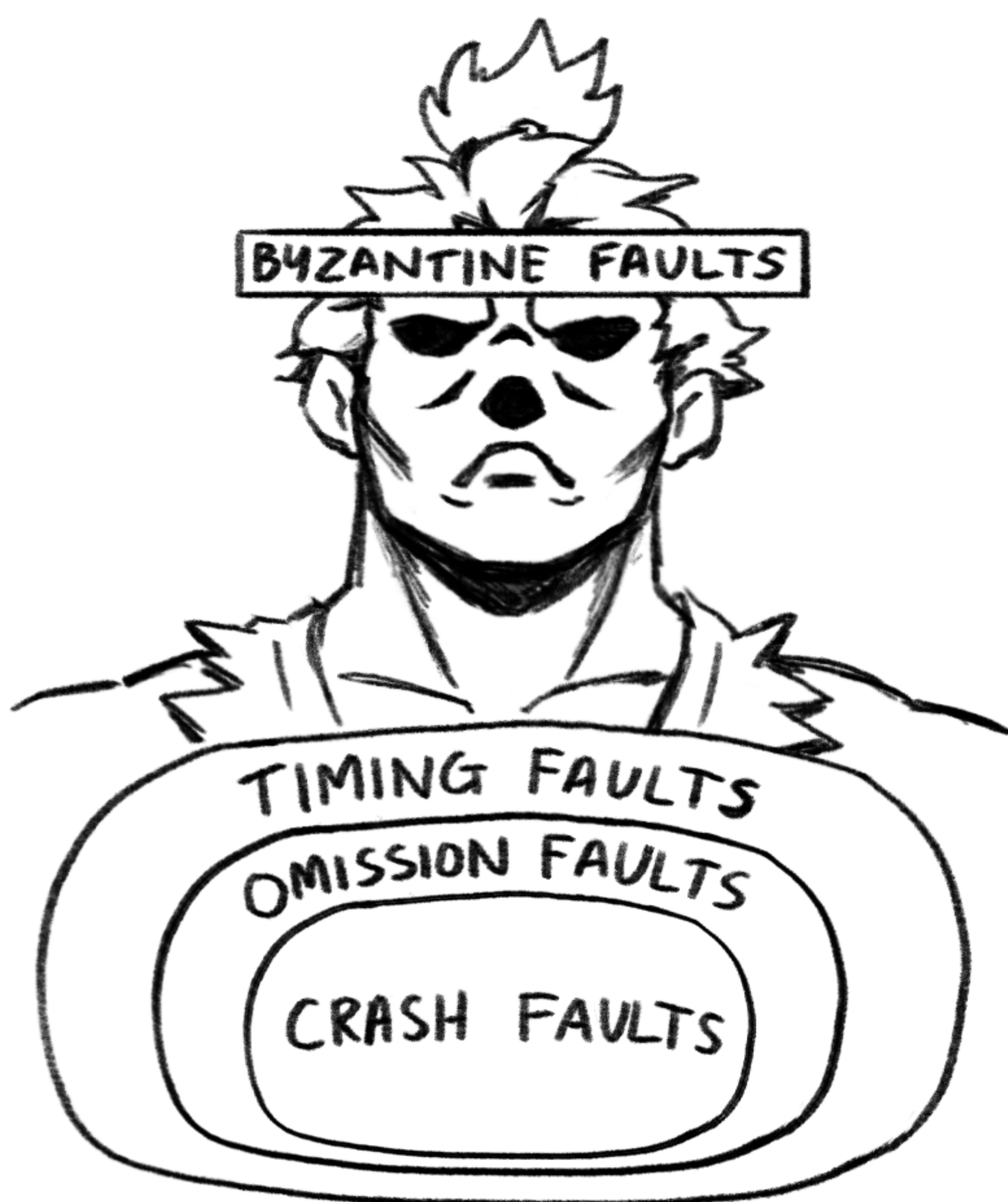
But wait... if crash faults count as omission faults, could it be that omission faults count as timing faults?

You probably weren't thinking that, but I took a class on this, so I already knew: Yes, they do!



Think of it this way: Because omitted messages are never received, they could be viewed as infinitely late messages.

And finally, a Byzantine fault is when a process behaves in an arbitrary or malicious way. This encompasses all of the previously covered faults.



That means if we can handle Byzantine faults, we can handle any fault!

(this is insanely difficult to do)

How do these affect our safety and liveness?

For the sake of clarity and conciseness, we'll focus on crash faults in this zine.

When a server crashes, the data on that server is lost. This is a "bad thing", so it violates our safety.



When a server crashes, any requests sent to that server will never be received, violating liveness. Receiving requests is a "good thing"!



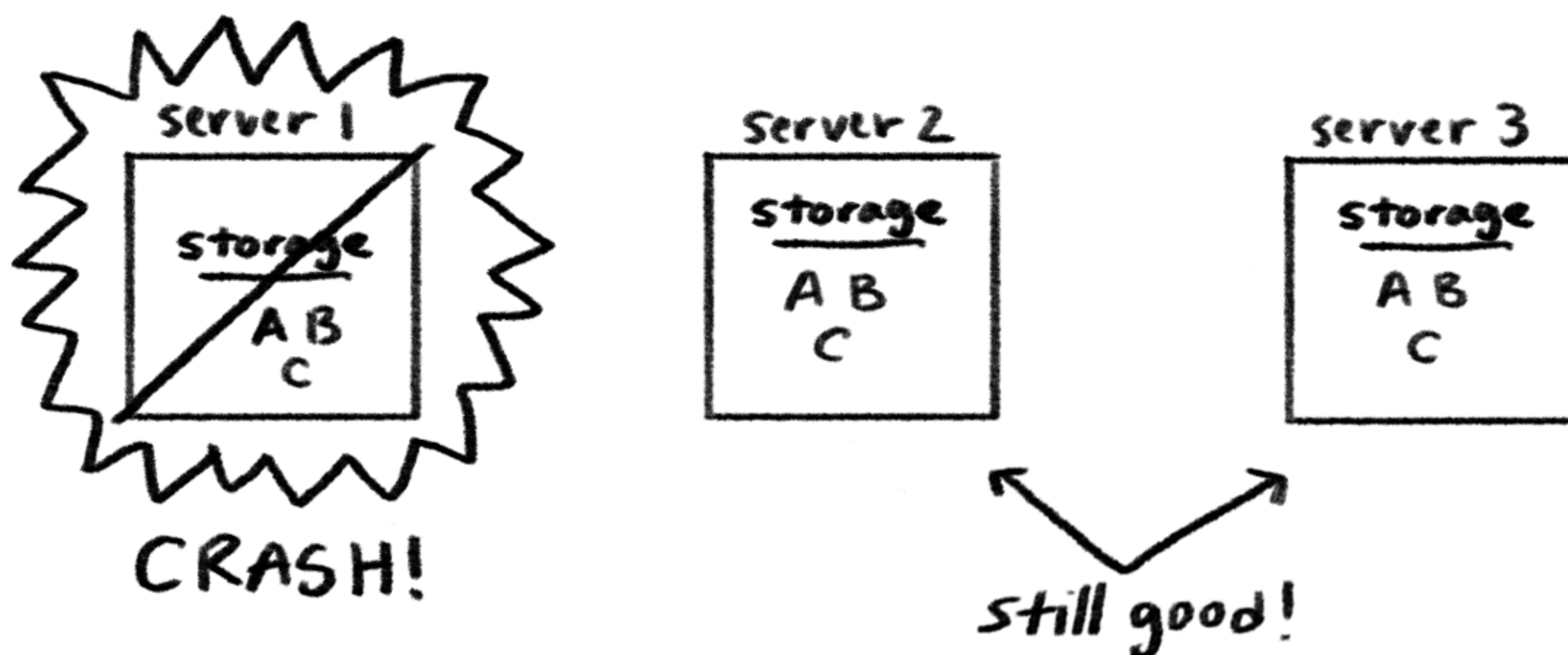
HOW TO FIGHT FAULTS

Now that we've covered the different types of faults, let's learn some techniques to tolerate them!

To help keep safety and liveness in the crash fault model, we can use...

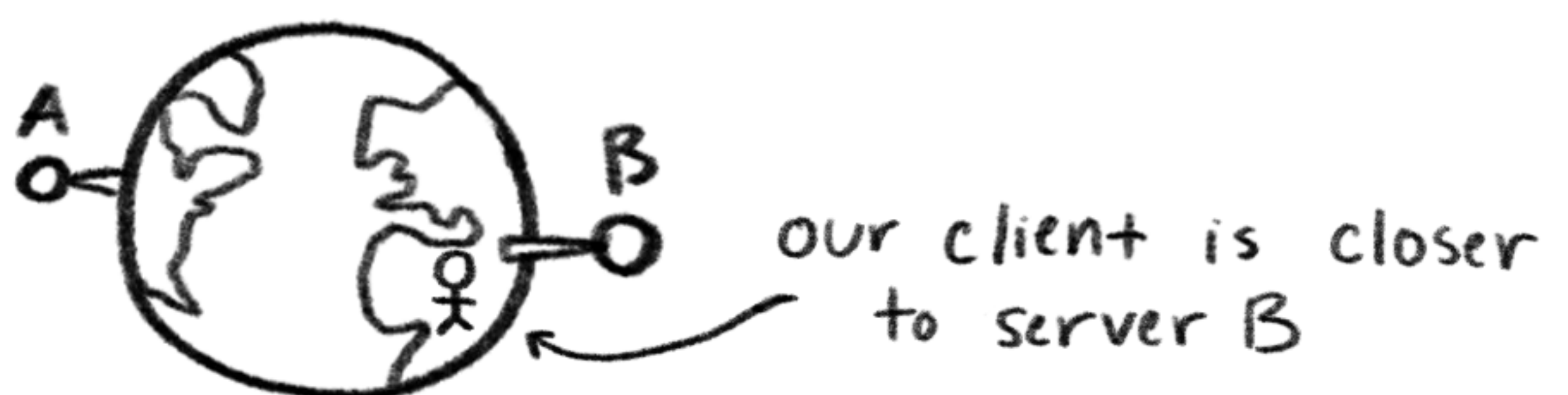
REPLICATION

Replication is the practice of making multiple copies of data. The backup servers we previously mentioned are an example of this.



Replication also helps us with:

-Data locality: keeping data close to clients that may need it



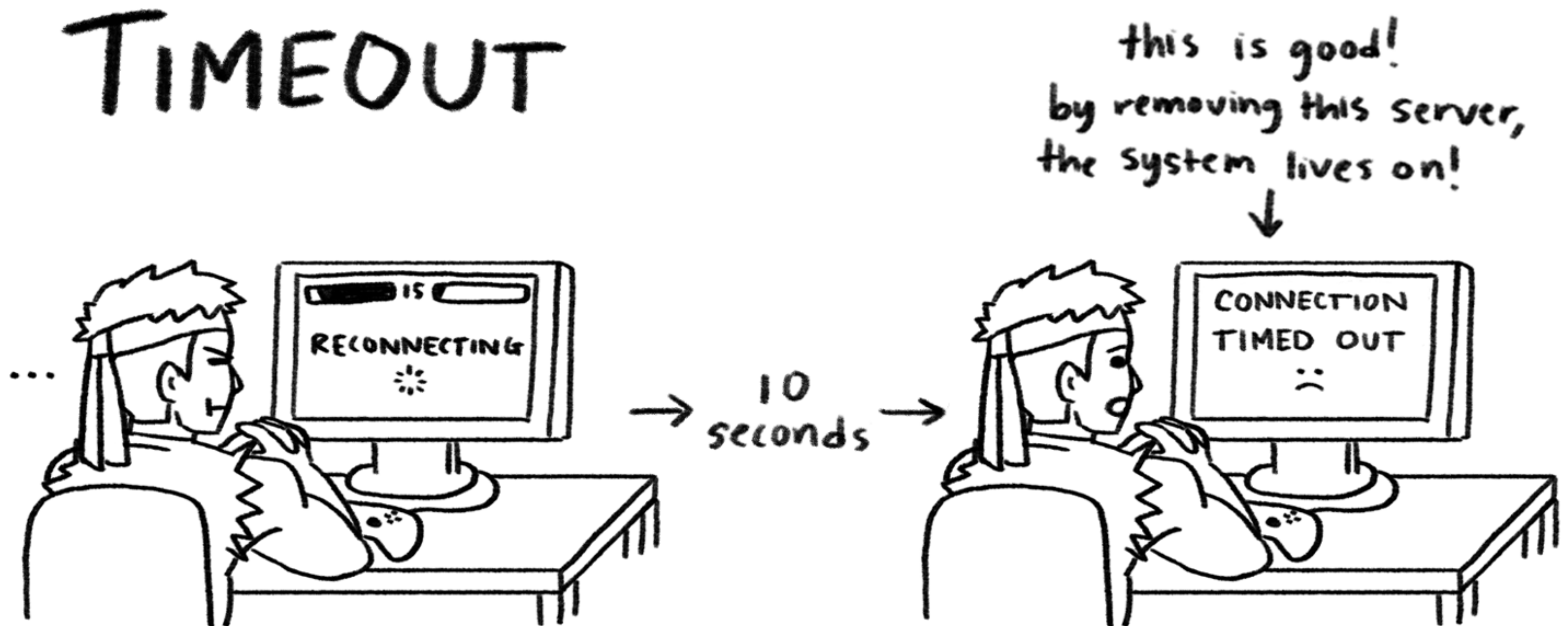
-Scalability: Dividing work between replicas to avoid bottlenecks

DOWN DETECTION

Down detection involves techniques used to detect when a server has become unreachable. Common ways we can implement this are with timeouts or a heartbeat mechanism.

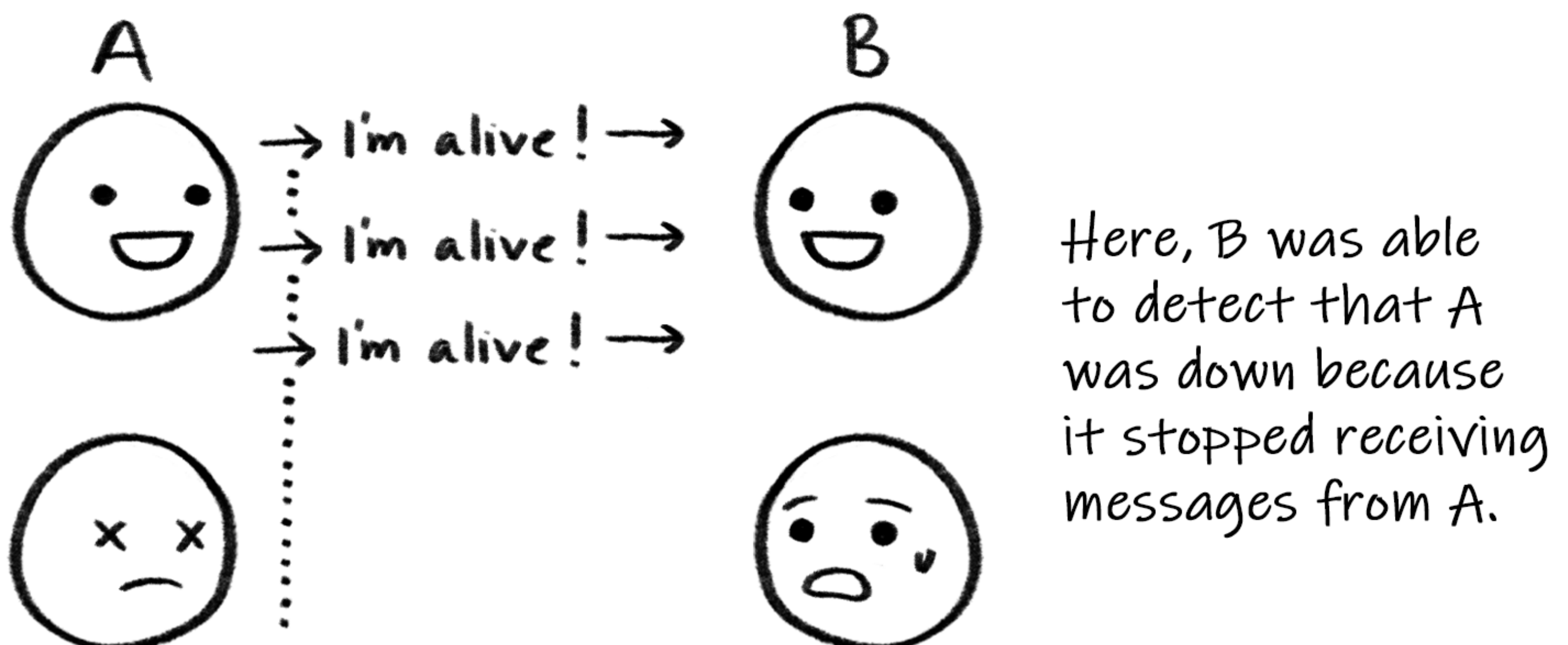
Timeouts set a time limit on a server's response.

TIMEOUT



Heartbeat mechanisms continuously send signals between servers to show that they are still alive.

HEARTBEAT



How do these help with safety and liveness?

Replication allows us to guard against data loss. When one replica crashes, we still have others to hold our data. This ensures safety!

Down detection allows us to learn when a server has crashed. If we are aware of this, we can do something about it, such as deleting the server so that our system can continue running past an unresponsive request. This ensures liveness!

Hooray! We now know some ways that our systems can fight faults!

PLAYER 2 DISCONNECTED
PLAYER 1 WINS





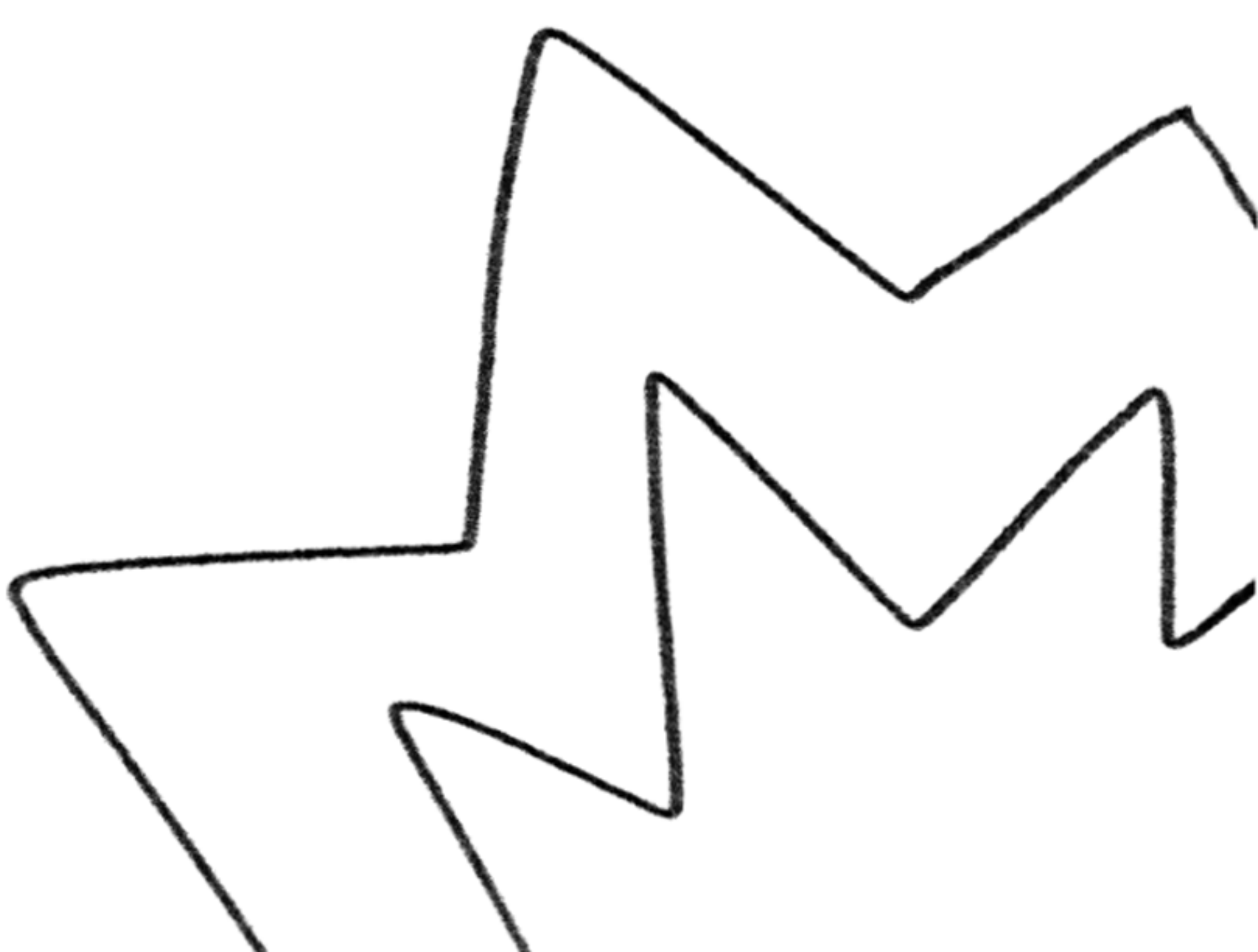

REFERENCES

The information in this zine was mostly from notes that I had from Professor Lindsey Kuper's Distributed Systems lectures. In fact, the theme of this zine was partly inspired by an example she made in class (liveness in terminating a match of a fighting game) which I used here.

However, here are some resources that I found helpful on the topic:

"Fault Tolerance in Distributed System." GeeksforGeeks, [geeksforgeeks.org/fault-tolerance-in-distributed-system/](https://www.geeksforgeeks.org/fault-tolerance-in-distributed-system/)

"Fault and Failure in Distributed Systems." Baeldung, <https://www.baeldung.com/cs/distributed-systems-fault-failure>



THANK YOU
FOR READING



For CSE138 w/ Prof. Lindsey Kuper
Ali Ali