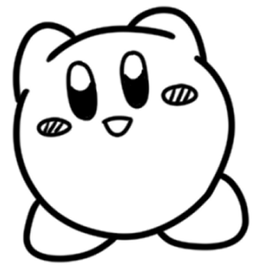
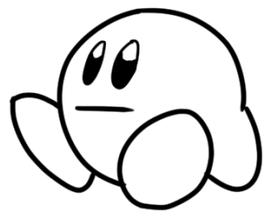
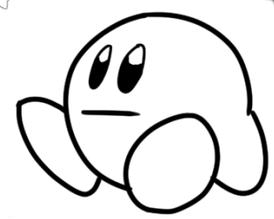
thanks ☺

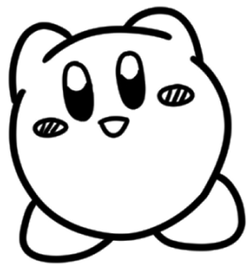
Replica Conflict Resolution

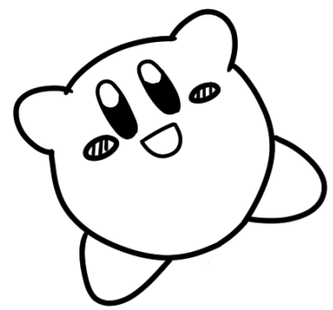


by Frank Pan









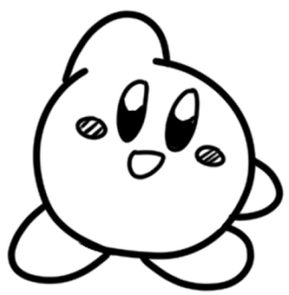


Table of Contents

References

1. lindsey's lectures ☺

2. peter's lectures ☺

3. "Gossiping in distributed systems"
   - Anne-Marie Kermarrec et. al

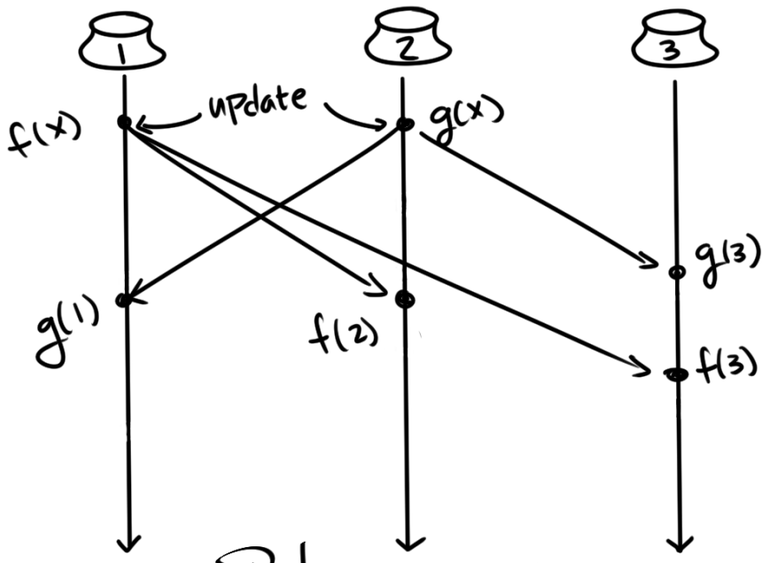
4. "The Promise & Limitations, of Gossip Protocols"
   - ken Birman

5. "A comprehensive study of Convergent & Commutative Replicated data types"
   - Marc Shapiro, et. al.

6. 128 fall friends ☺

Operation-Based CRDT's

-When a client fowards a request to a replica, updates to other replicas occur when the source replica sends the request operation to all other replicas. This uses an epidemic model to transmit updates just like state-based CRDT's



Phases

Prepare

- Modify your current state with the OP.

Effect

- Modify other replicas
- If delivered in causal order, concurrent updates must be commutative, else ALL updates have to.

Examples

Counters

- Each node updates, then propagate.

Simple!

Sets

- Add() is Union based & commutative so simply propogate add and it converges.
- remove() requires that add occur beforehand.

Things to Know

Process: a node in a distributed system capable of communicating with other processes (aka. Node)

Latency: time it takes for a message to travel

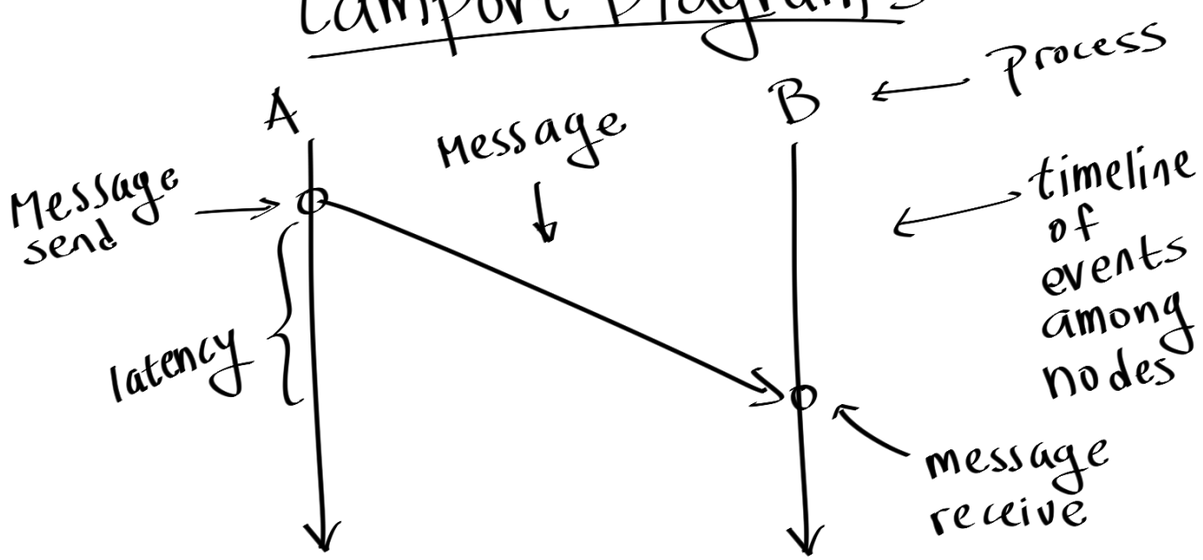
State: contents of registers/memory

Consistency: Every read receives the most recent write or an error

records: data type {key: value}

- writes set the value of key
- reads read the value of key

Lamport Diagrams

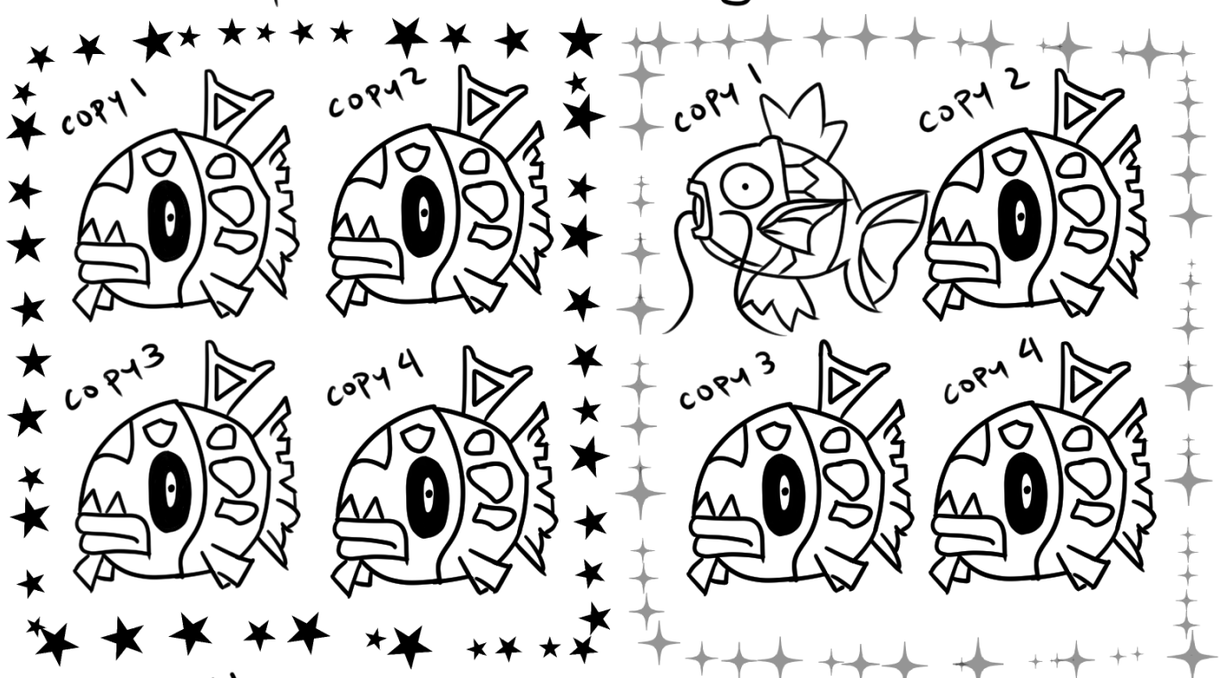


The Problem

- When you store data somewhere it's usually replicated and stored somewhere else. This ensures that if a singular server goes down, you should still be able to access your data.

So, what's the problem?

- Something happens and a replica of your data is gone/corrupted/different
- You try to access something and get bad data



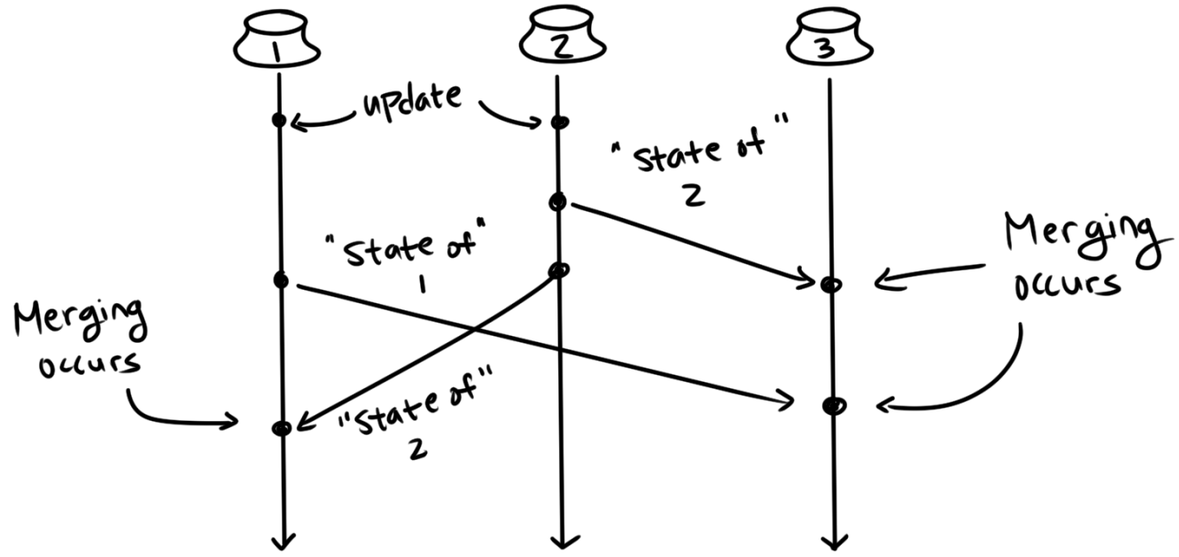
- Good!

- NOT GOOD :(

Let's begin by talking about replication and how that results in these Problems

State-based CRDT's

- When a client forwards a request to a replica, updates occur then they transmit states to other replicas. This essentially uses an epidemic model to transmit updates.



Merging

- The lifeblood of state-based CRDTs, we need to ensure
  1) Commutativity ($x \cdot y = y \cdot x$)
  2) Associativity ($(x \cdot y) \cdot z = x \cdot (y \cdot z)$)
  3) idempotency ($x \cdot x = x$)

  } essentially gives us 2 options. MAX & Union

Examples

Counters

- each node updates its own counter ONLY. Update the status of others using MAX when you receive their state.

Sets

- Simply union the sets to produce a new update.
- Deleted elements may reappear due to latency to propagate deletes (or crashes).

Conflict-free Replicated Data Types

Conflict-free Replicated data types (CRDTs) are basically data types that you can update concurrently and have all replicas eventually converge. We'll give a quick intro to two different models.

State-based

- upon an update, update the local state
- occasionally, a replica will send it's FULL state to another replica
- upon receiving a state, a process will MERGE with their state
- Replicas will converge eventually given these conditions.

1) states are partially ordered with a upper bound operation
2) updates are increasing
3) merges need to be idempotent, associative, & commutative.

→ More on this on p.11 →

Operation-based

- upon an update, update the local data, then broadcast the update to all other replicas
- What happens if two concurrent updates happen?
  - as long as the operations are commutative, replicas will converge as long as every update arrives at every replica.
  - operations are NOT idempotent.
- To ensure convergences:
  1) Reliable transmissions

→ More on this on p.12 →

So, let's talk about Replication

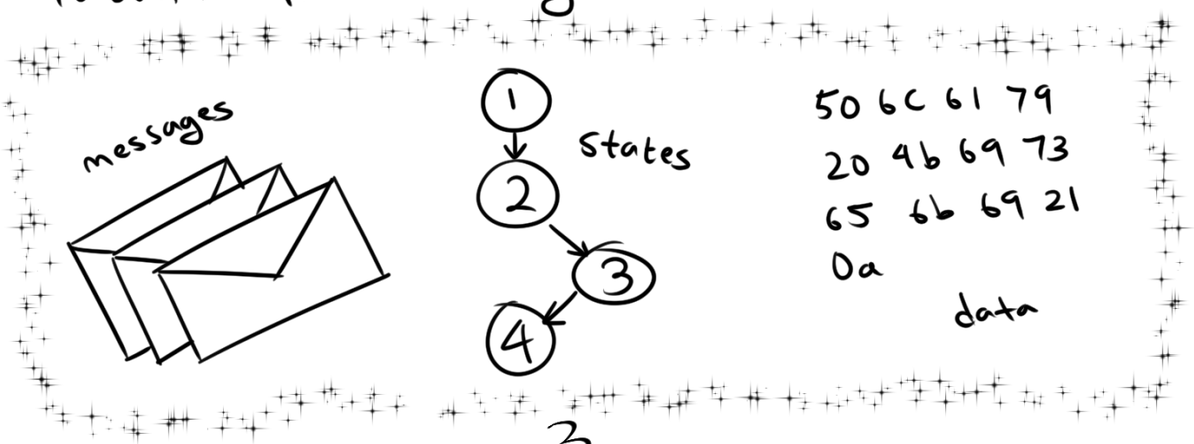
When we talk about distributed systems, we have to keep in mind that individual parts can die!

tldr: FAILURES HAPPEN

Don't be sad when data is gone!

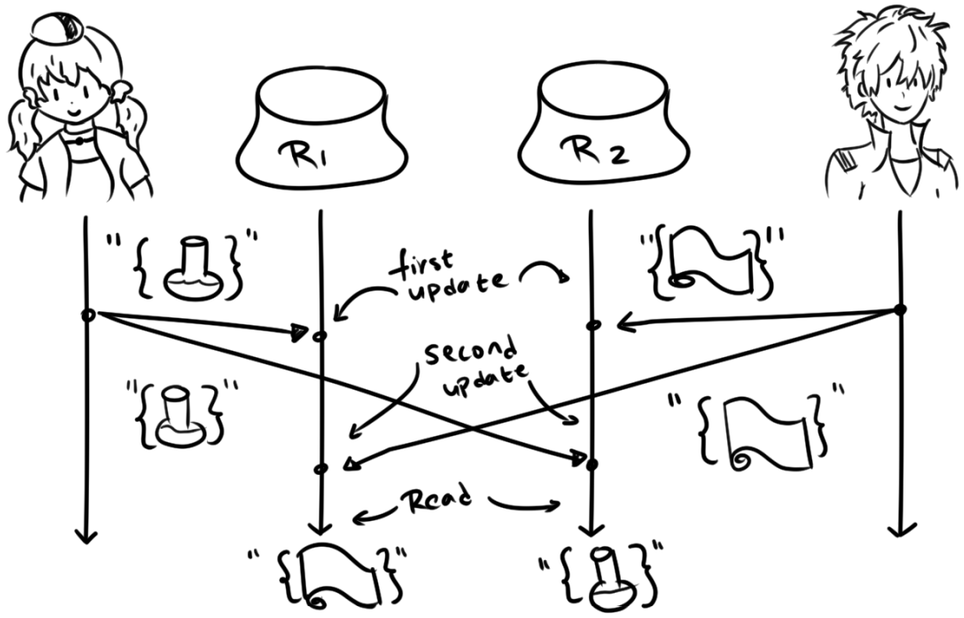
Be prepared!

- Replication is employed to guard against problems like crashes
- When you replicate data, you want to store them in separate locations.
  - locality matters so multiple processes can access data
- To sum it up, save messages, states, and data



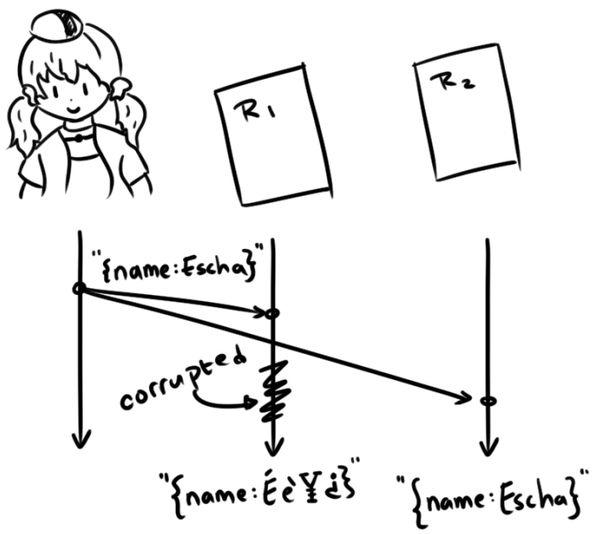
How do we get Replica Conflicts?

There are tons of scenarios that lead to a replica conflict so we're going to highlight a few.



- The messages to update the data do not get received in the same order as they were sent. This leads to replicas having different data despite getting the same messages.

- This is called a total order violation: If someone sends a message first then all replicas should process that request first before anything else.

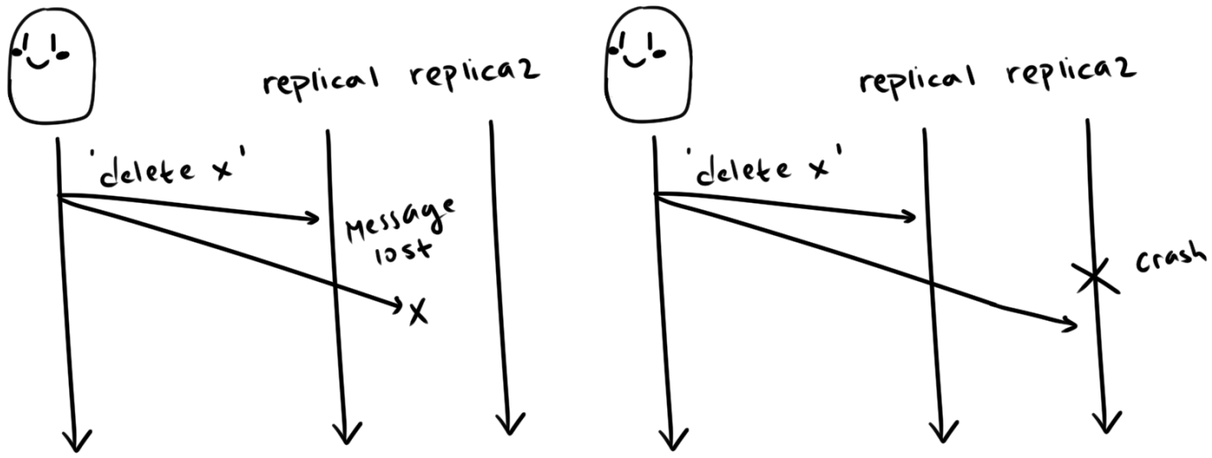


- Sometimes problems are completely out of your control
- For example, on the left, the data for a value got corrupted somehow which can lead to problems if you access it.
- These are called Byzantine faults
- Something went wrong but you do not know the exact reason why the fail occured and may give different info to different observers

Tombstones

When you delete something it's gone for sure right?

NO



- Distributed systems assume that failures happen. The two scenarios above illustrate possibilities where delete operations only go through on some replicas.

- What happens when a replica recovers and see that it has a record while the others don't?
  - it's possible it doesn't know the record was deleted and tries to send the data around.

What is a tombstone?

- Tombstones are essentially records of a deletion that tells everyone "This was deleted"
- Creation of a tombstone record doesn't mean the value is deleted. The tombstone takes the place of the original record.

{"RTZ":"EZ"} ⇒ {"RTZ": NULL}

How does it function?

- If a node receives a request pertaining to a node with a tombstone:
  1) ignore the request
  2) respond saying the node is deleted
- Tombstones are kept forever or once it has aged enough. This is up to the application to configure.

Gossip - continued

Let's divide gossip into 3 parts

1) Peer selection
2) Data exchange
3) Data processing

Peer Selection

- Goal is to select nodes among all LIVE nodes.
  - This must be uniformly selected
- Selection is based on the gossip model.
  - ex. if you want to disseminate information, select more peers.
  - ex. the peer sampling method gives each node a subset of nodes to choose from.

Data Selection

- Data doesn't have to be just stored values/states
- Heavily application based.

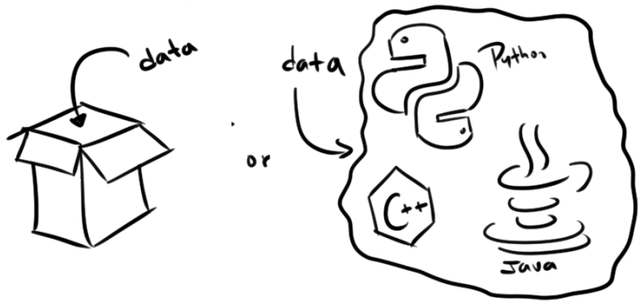
  ex Dissemination models either 'push' or send to n nodes or 'pull' where n nodes sends

  ex. Peer sampling simply sends their lists of peers to each other.

Data Processing

- like data selection, this is highly Application dependent!
- upon receiving data, do one of 3 things:

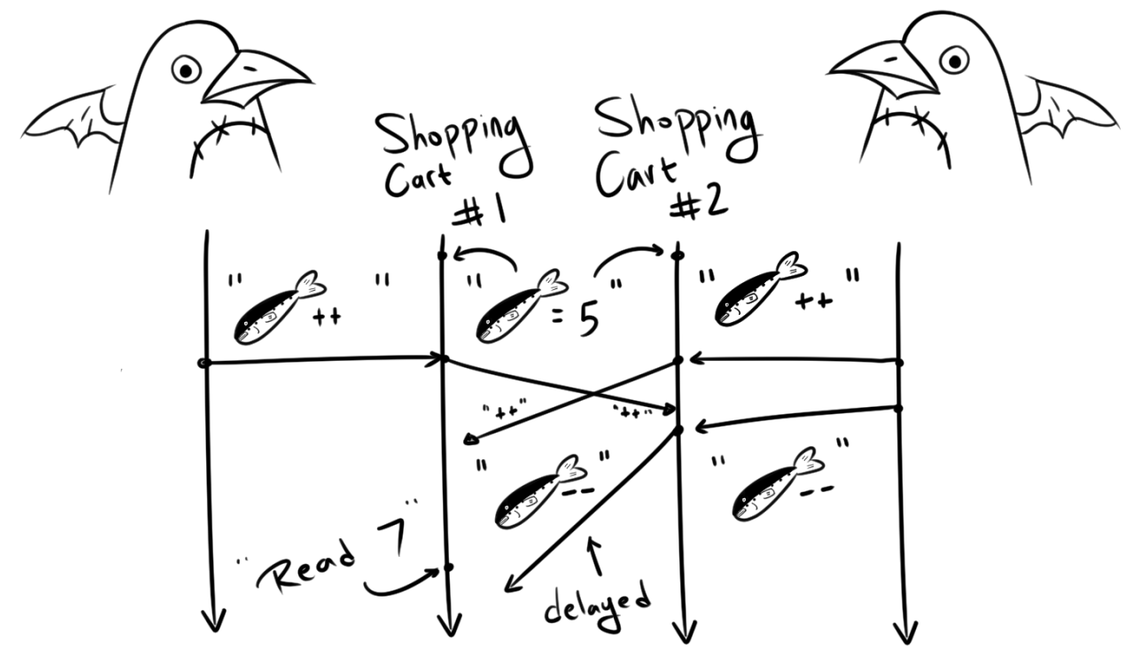
  1) keep data for future iterations
  2) use the data
  3) pass the data to a higher application layer



Miscellaneous

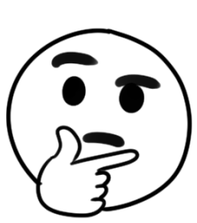
- Limitations of Gossip
  - Channels between nodes can only carry so much data. Increasing the number of events can lead to problems. Linear growth in events can overwhelm the system.
  - Byzantine faults that lead to incorrect data can defeat the purpose of gossip.
- Benefits of Gossip
  - convergent consistency!
  - Simple to implement
  - fault tolerance, especially network problems

Replica Conflict - continued

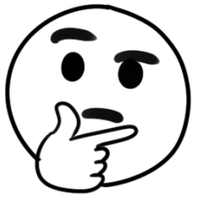


- Sometimes updates do not go through on all replicas before someone tries to access something
- This can lead to 2 or more replicas with different values
  - We can solve this through resolution on writes to a server or reads from the server.
  - Sometimes problems like this are just impossible to 100% prevent so you have to build your system around these events

SO, how do we select how to resolve these conflicts?



Depends on what you want for your system!



Methods to resolve Conflicts!

So, lets talk about how to help solve or at least mitigate replica conflicts. Here are a few protocols/concepts/etc. that we'll explain more about soon

#1) Gossip Protocol

- This protocol is a way for systems to be consistent. That is, if updates/writes stop arriving, all systems will all share the same state.
- A basic runthrough of the protocol goes like this:
  1) a node in the system picks a node at random
  2) both nodes exchange info
  3) repeat

#2) Tombstones

- When you delete something from a system, you know that locally the delete goes through. But in a distributed system environment? Who knows?
- Tombstones help guard against those problems

#3) Conflict-free Replicated Data Type

- Objects you can update without consensus OR synchronization. These objects will eventually converge given that all concurrent updates are commutative(!)

GOSSIP

Gossiping as a method to ensure eventual consistency has been relevant since the 1987 paper "Epidemic Algorithms for Replicated Database Maintenance." It's essentially a communication protocol modeled after epidemics.

Assumptions

1. We are dealing with a distributed system with near-continuous changes
2. There exists a connection between the nodes in the system.

Logic