Holochain Core Concepts

Welcome to Holochain Core Concepts! Here we'll introduce you to the basics of Holochain, a framework and network protocol for building secure distributed applications. Holochain is different from what you may be used to, but we'll go at a comfortable pace, building on things you already know.

Who is this introduction for?

We've written this introduction for programmers, CTOs, and other technically oriented people. It's a bit like a choose-your-own-adventure story.

If you're in a hurry, and want to find out if Holochain is a good fit for your project, you can just read the articles—or if you're in a *real* hurry, read the intro paragraph at the top and the key takeaways at the bottom.

If you want to dig deep, each article has links to further reading from the guidebook, API documentation, or web.

Ready? Let's get started

1. The Basics: What is Holochain?

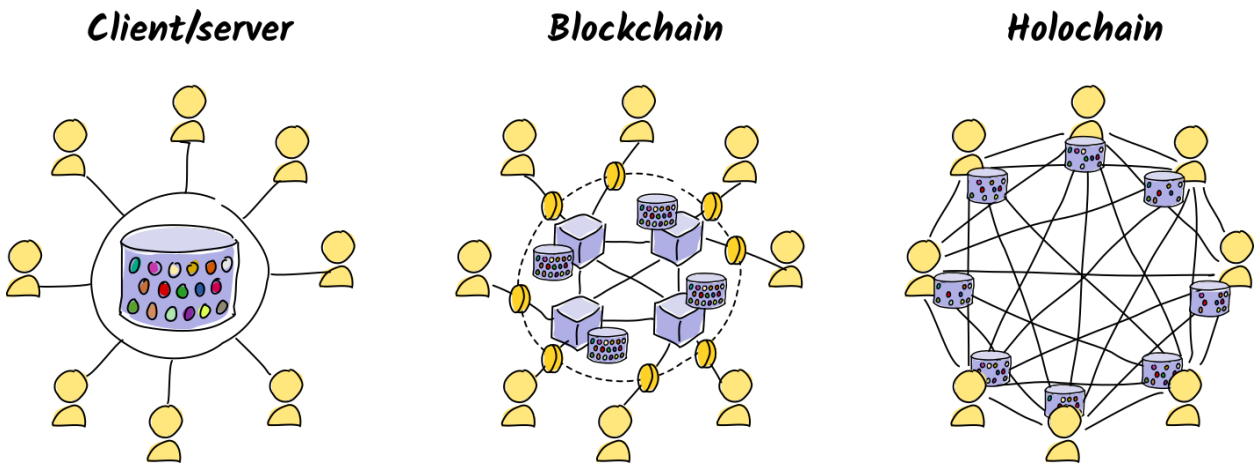
Holochain is an open-source development framework and networking protocol. It allows you to create **truly serverless applications** with high levels of **security, reliability, and performance**. Every user runs the application on their own device, creates and stores their own data, and talks directly to other users. The security of the application is supported by both cryptography and peer accountability.

What you'll learn

1. The problem with centralized architectures
2. Current solutions
3. How Holochain does things differently

Why it matters

You'll understand how Holochain is different from centralized architectures, and how this difference can make your applications more robust, secure, and affordable.



The problem with centralized architectures

We're all familiar with the client/server architecture. It's comfortable, allows for easy maintenance, and is well supported by a marketplace of service providers. It also makes it easy for you to control access rights and enforce business rules. As a focal point for processing and storage, however, it leaves your application vulnerable:

- It's an attractive target for hackers.
- Device and network outages or misconfiguration can take the app down.

- Hosting costs scale with traffic, making you a victim of your own success.
- Storing private user data increases your legal liability.

Current solutions

Cloud hosting platforms offer horizontal scaling. As your application grows, you add more virtual machines to increase availability. This is tricky to maintain, as you are responsible for updating your OS, provisioning new machines, and maintaining security.

Serverless computing liberates you from those details and lets you focus on the core of your application—just choose the ingredients, link them together with functions, and press play. However, it is just an abstraction; it still runs on rented hardware with recurring costs and central failure points.

Distributed computing efforts, like blockchain, attempt to solve these problems by creating a network of participants who all hold a public, global data set. Each user helps maintain the availability and integrity of the data. Centralized vulnerabilities are eliminated. However, it's costly to replicate, check, and come to consensus on the contents of the data set, sometimes by design. This hurts performance and creates waste. It also leads to centralization—participants are separated into 'full nodes' who have the computer power, reputation, or capital to participate, and 'light clients' who have to ask the full nodes to do things for them, often in exchange for fees. *To our thinking, this begins to sound like client/server all over again.*

How Holochain does things differently

Holochain approaches the problem from a different set of assumptions. Reality offers a great model—agents in the physical world interact with each other just fine without an absolute, ordered, total view of all events. We don't need a server or global public ledger.

We start with users, not servers or data, as the primary system component. The application is modeled from the user perspective, which we call **agent-centric** computing. Empowered by the Holochain runtime, each user runs their own copy of the back end code, controls their identity, and stores their own private and public data. An encrypted peer-to-peer network for each app means that users can find each other and communicate directly.

Then we ask what sort of data integrity guarantees people need in order to interact meaningfully and safely with one another. Half of the problem is already solved—because everyone has the 'rules of the game' in their copy of the code, they can verify that their peers are playing the game correctly just by looking at the data they create. On top of this, we add cryptographic proofs of authorship and tamper resistance.

This is Holochain's first pillar: **intrinsic data integrity**.

However, we're only halfway there. It's not particularly resilient; data can get lost when people go offline. It also makes everyone do a lot of their own work to find and validate data.

So we add another pillar: **peer witnessing**. Each piece of public data is witnessed, validated, and stored by a random selection of devices. Together, all cooperating participants detect modified or invalid data, spread evidence of corrupt actors or validators, and take steps to counteract threats.

These simple building blocks create something surprisingly robust—a multicellular social organism with a memory and an immune system. It mimics the way that biological systems have managed to thrive in the face of novel threats for millions of years.

The foundation of Holochain is simple, but the consequences of our design can lead to new challenges. However, most of the solutions can be found in the experiences of real life, which is already agent-centric. Additionally, some of the trickier problems of distributed computing are handled by Holochain itself at the 'subconscious' layer. All you need to do is think about your application logic and the validation rules for your data.

Key takeaways

- Traditional centralized architectures are easy to understand, maintain, and secure, but they create central points of failure.
- Holochain turns the architecture of applications inside-out—users are at the center of their online presence, in charge of their own identity, data, and processing.
- In a Holochain app, processing, storage, and security surface area are distributed across the entire network. This reduces central points of failure, bottlenecks, and attractive attack targets.
- The two pillars of application integrity are intrinsic data integrity and peer replication/validation.
- There is no single global database; data comes from many individual sources.
- Each user of an app also participates in building the app's infrastructure, supplying their own compute and storage resources and taking responsibility for validating and storing a small portion of other users' data.
- The whole is greater than the sum of its parts—many agents, playing by simple rules, combine to form a social organism that maintains its own health.

02. Application Architecture

Applications built with Holochain are highly **modular**. This makes it easy to share code and compose smaller pieces together into larger wholes. Each functional part of a Holochain

application, called a **DNA**, has its own business rules, isolated peer-to-peer network, and shared database.

What you'll learn

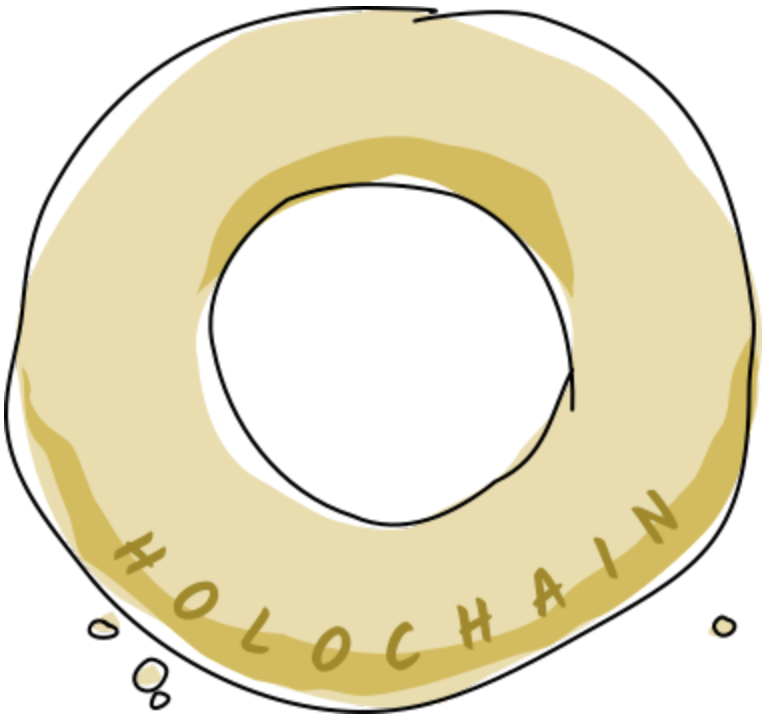
1. Agent-centric integrity: inside, outside, and in between
2. Layers of the application stack

Why it matters

A good understanding of the components of the tech stack will equip you to architect a well-structured, maintainable application. Because Holochain is probably different from what you're used to, it's good to shift your thinking early.

Agent-centric integrity: inside, outside, and in between

Before we talk about the tech stack, let's talk about donuts. That's a good way to start, isn't it?

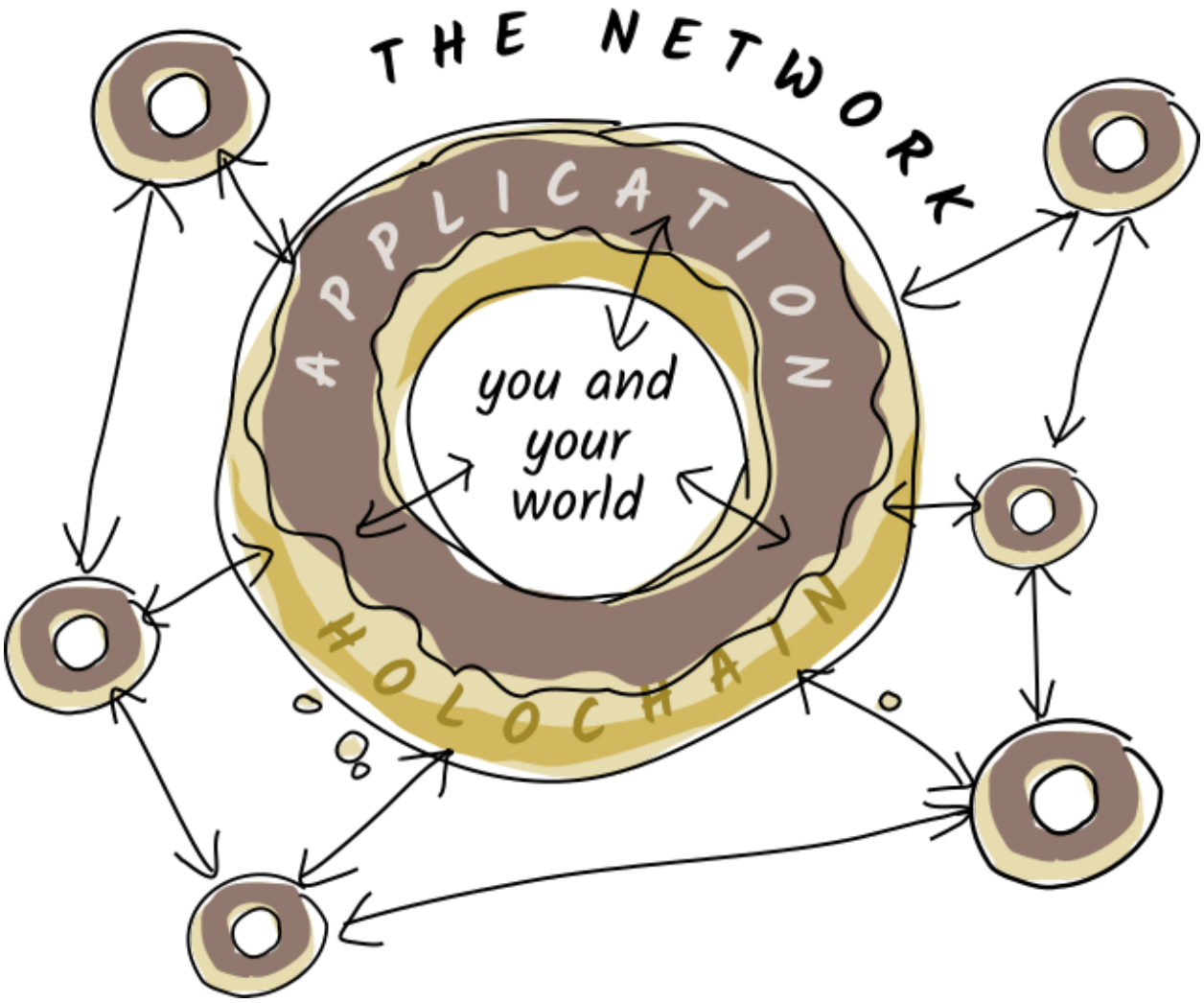


This is Holochain. Thanks to the magic of gluten, it has enough integrity to hold itself together. It separates the universe into two empty spaces—the hole and the space beyond.

On top of Holochain is your application. Each application has a different flavor.

Let’s put you inside the hole. You have agency—the power to receive information from your world and act on it. Together, your copy of the Holochain runtime and your application mediate

between you and Holochain land. Your application defines a set of functions that define all the valid ways you can interact with it, and Holochain exposes those functions to you.

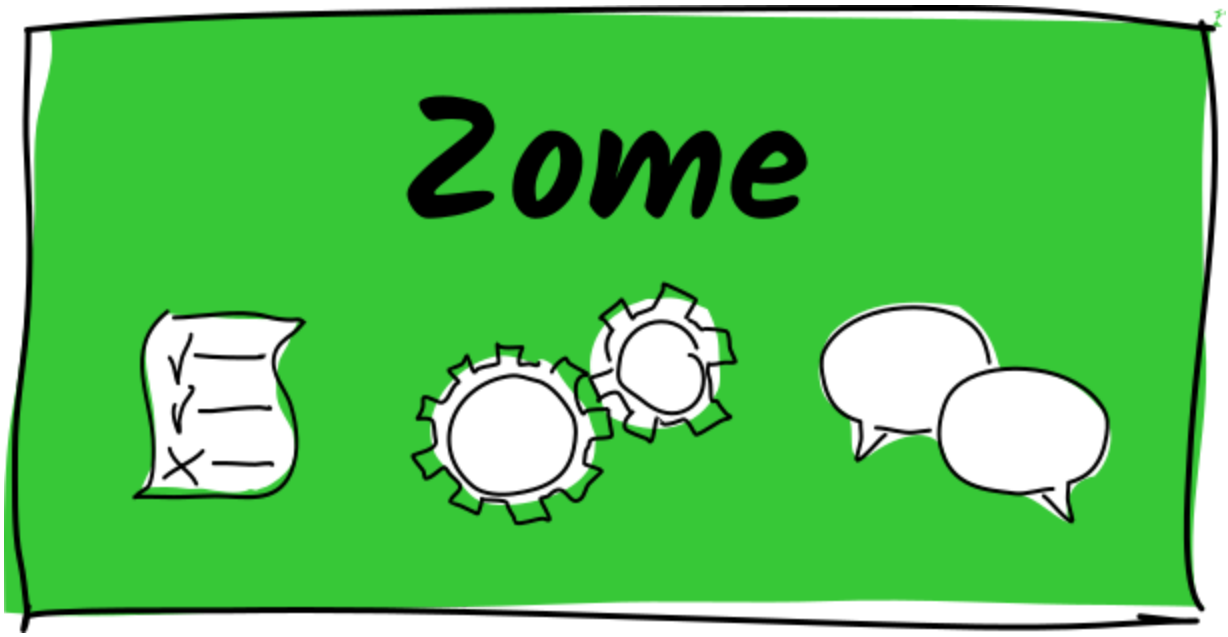


On the outside of the donut is a shared space; in it are the other people using the same application. Holochain mediates interactions with them too, shuttling information across space with the help of a computer network. Again, your app defines what it considers valid, on this side, through a set of rules that define what data should look like.

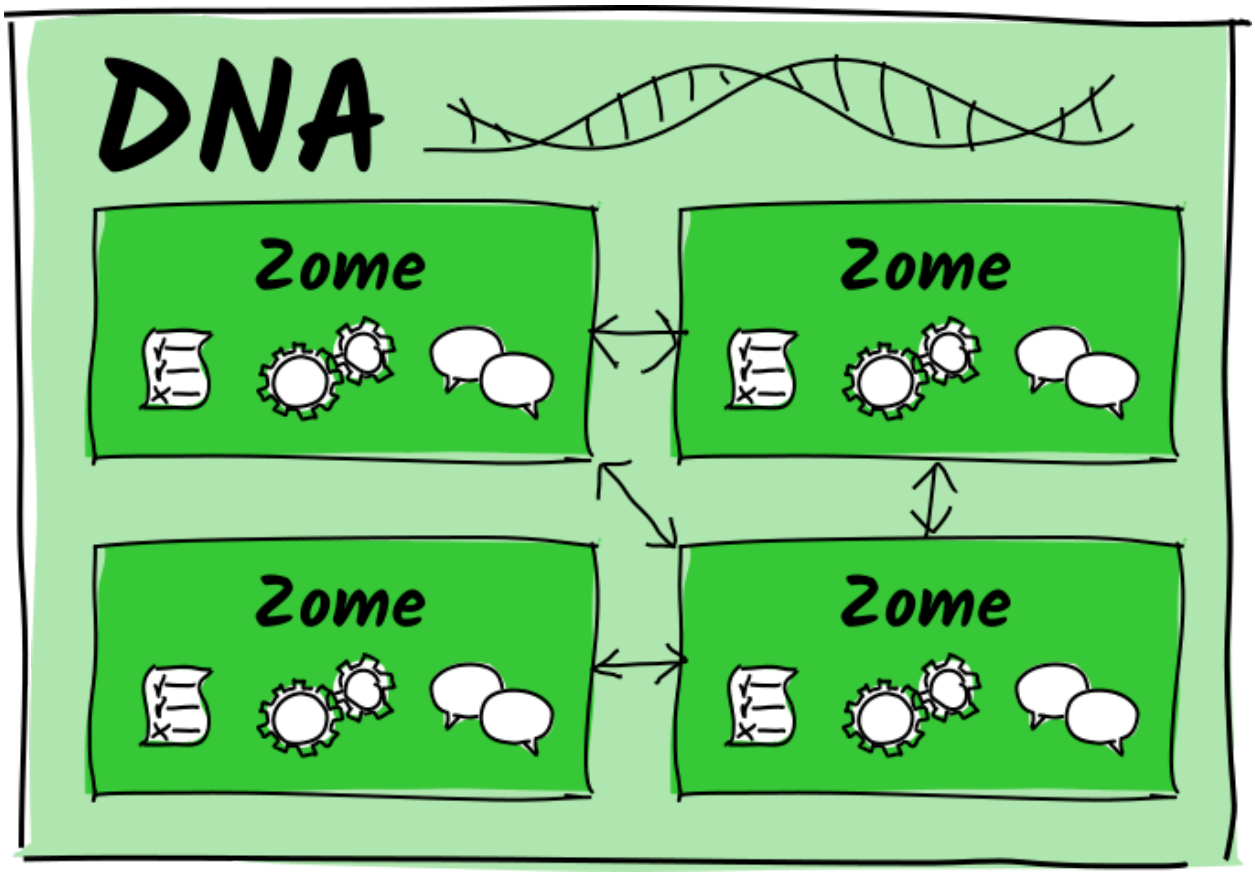
Holochain creates a ‘double membrane’ for each participant, bridging between their world and the digital space they share with others. It ensures the integrity of information passing through both the inside and the outside. This lets people safely do the things that are important to them, without having to depend on a central authority.

Layers of the application stack

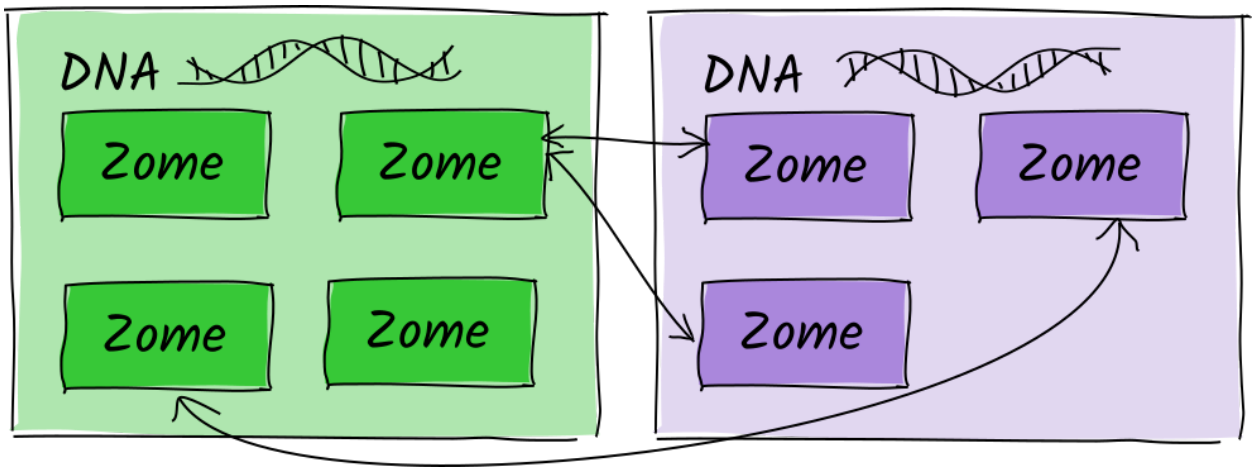
Now, let's get into the details of how a Holochain app is put together. Holochain apps (**hApps**) are made from loosely coupled components. Here's how they are built:

1. 

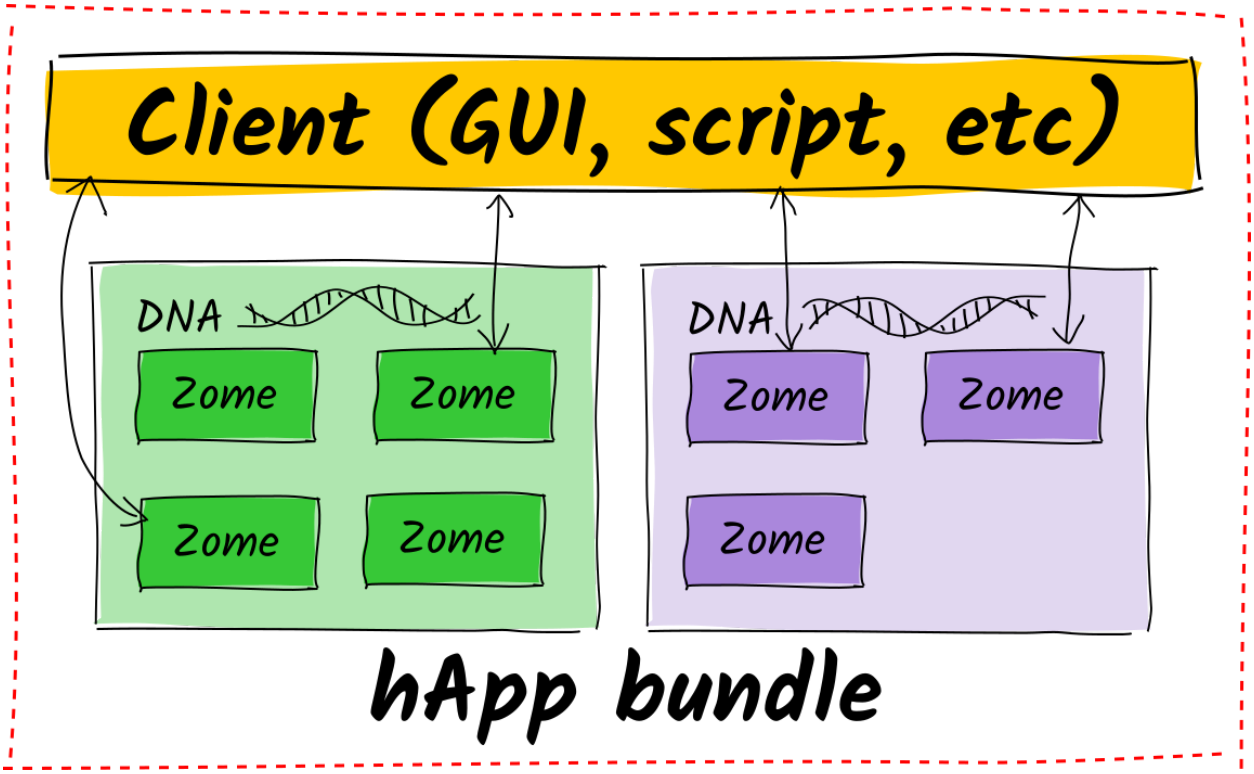
   A code module called a **zome** (short for chromosome) defines the core logic of your app. It exposes its public functions to the Holochain runtime. Some of these functions are required, such as validation functions for each type of data the zome defines. Other functions are invented by the developer and define the zome's public API.

2. 

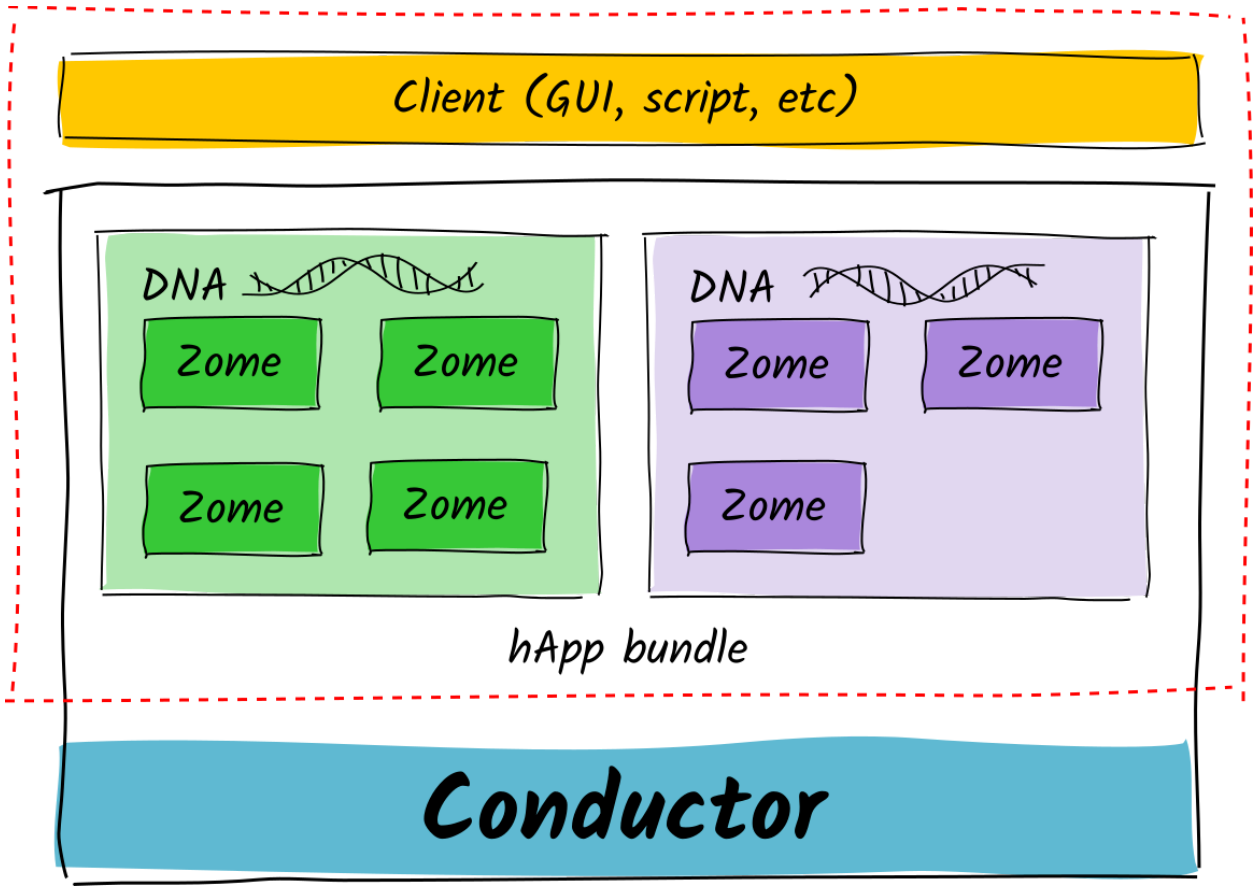
One or more zomes are combined into a **DNA** that defines the basic functionality and ‘rules of the game’ for a portion of an application’s functionality. You can think of it like a microservice. The running DNA instance is the user’s personal **agent**—every piece of data that it creates or message it sends, it does so from the perspective of the user.

3. 

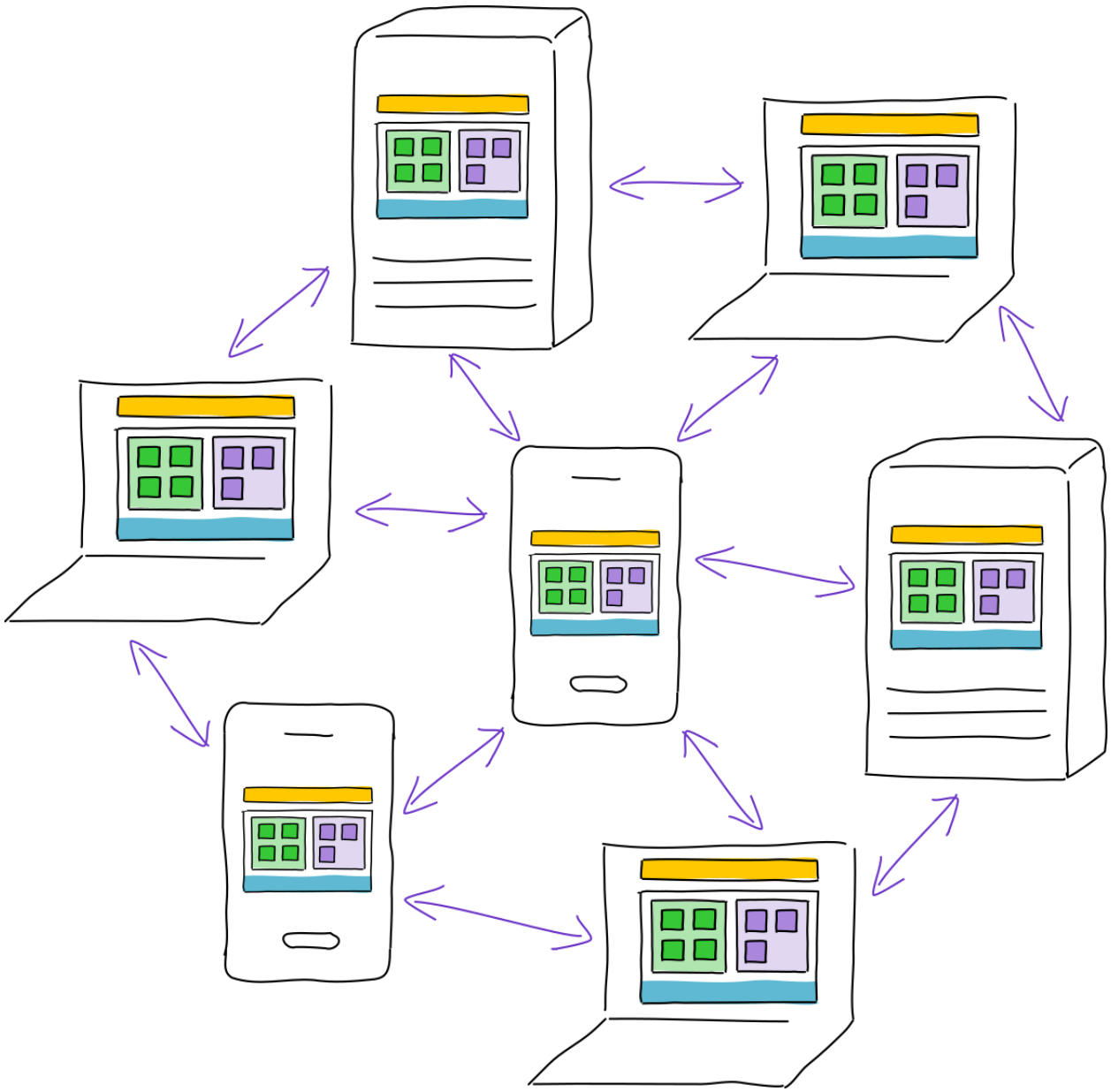
A user’s DNA instances can talk to each other’s APIs via **bridging**. This lets you compose them into a bundle of functionality that’s needed for a full-featured app.

4. 

   A **client** on the user's device, such as a GUI or utility script, talks to the DNAs' APIs via a lightweight remote procedure call (RPC) interface. The client is like the front end of a traditional app and can be written with whatever language, toolkit, or framework you like. This client and its DNAs make up a **hApp bundle**.

5. 

All DNAs are hosted in the Holochain runtime, called the **conductor**, which sandboxes and executes DNA code, manages data flow and storage, and handles connections between components of the stack. As with all the other components, it lives on each user's device.



6. Each conductor is a **node** in a peer-to-peer network of agents using the same app. Each DNA in the hApp has its own isolated peer-to-peer network and distributed data store. The conductor handles communication and data sharing between nodes.

Functional components and architectural layers both enjoy clean separation. You can combine, augment, or replace existing pieces. This gives you a lot of flexibility and empowers your users to take ownership of their experience.

Info

All functions in your DNA start with a fresh memory state which is cleared once the function is finished. The only place that persistent state is kept is in the user's personal data journal. If you've written applications with REST-friendly stacks like Django and PHP-FastCGI, or with function-as-a-service platforms like AWS Lambda, you're probably familiar with this pattern.

Key takeaways

You can see that Holochain is different from typical application stacks. Here's a summary:

- An application consists of a client and a hApp bundle (a collection of isolated services called DNAs which in turn are composed of code modules called zomes).
- Each user has their own copy of the client, hApp bundle, and Holochain runtime (conductor).
- The conductor sandboxes the DNA code, mediating all access to the device's resources, including networking and storage.
- All code is executed on behalf, and from the perspective, of the individual user.
- Users communicate and share data directly with one another rather than through a central server or blockchain validator network.
- Holochain is opinionated about data—it handles all storage and retrieval. (We'll learn about why and how in the next three articles.)
- DNA functions don't maintain any in-memory state between calls.
- Persistence logic and core business logic are mixed together in your DNA, because at its heart, Holochain is a framework for data validation. However, you usually don't need much code in your DNA—just enough to encode the 'rules of the game' for your application.
- As with microservices, Holochain lends itself to combining small, reusable components into large applications.

Learn more

- Holochain: Reinventing Applications
- The Holo vision: Serverless 2.0

03. The Source Chain: A Personal Data Store

Each user in a Holochain network creates and stores their own data in a journal called a **source chain**. Each journal entry is cryptographically signed by its author and is immutable once written (append-only).

What you'll learn

1. How agent identities are created
2. Where user data is stored
3. Detecting third-party tampering

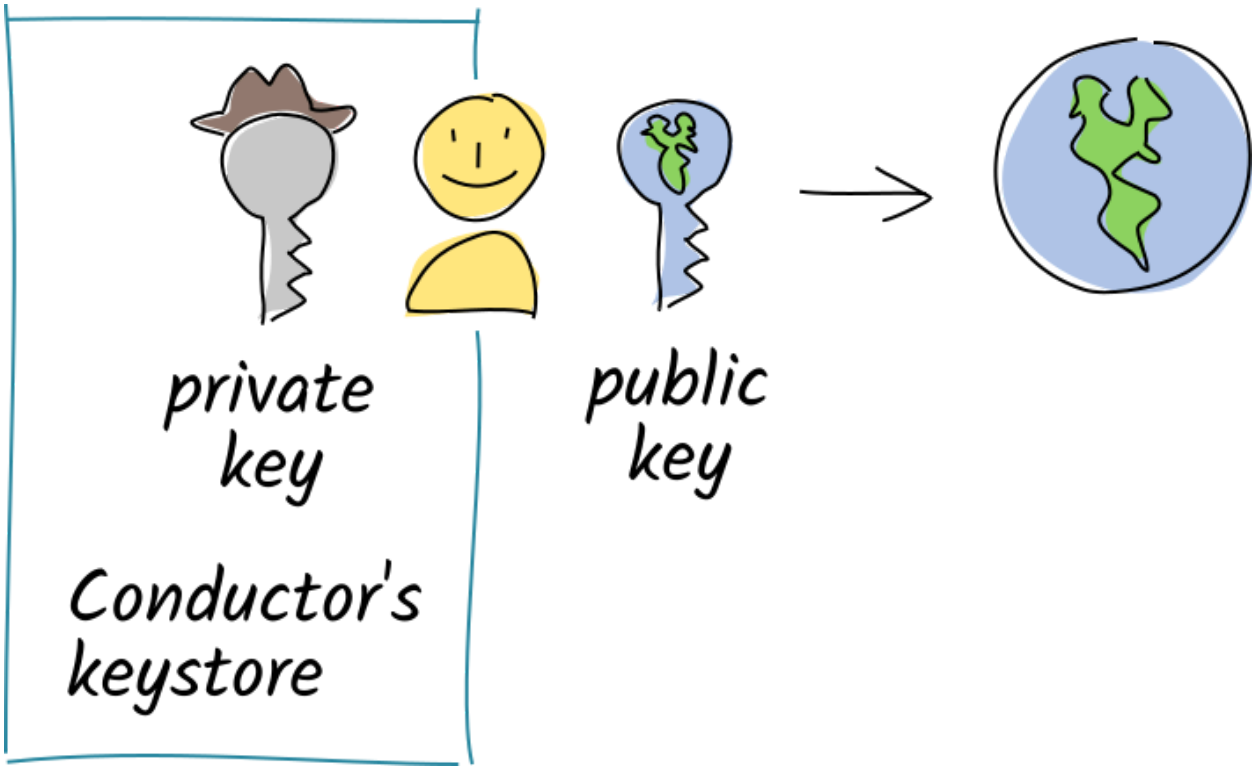
Why it matters

When you understand how agents and their data are represented, you have the foundational knowledge for creating an appropriate user experience that takes advantage of Holochain's agent-centric design.

Agent identity

Let's take a look at one single node and see what's happening from the user's perspective.

Back in the basics, we said that one of Holochain's pillars is 'intrinsic data integrity.' The first stone in this pillar is **public key cryptography**, which allows each user to create and authenticate their own identifier without a central password database. If you've ever used SSH, you're already familiar with this.



When you join a hApp's network, you create an identifier for yourself by generating a **public/private key pair**. This key pair does a few things for you:

- It gives you a unique ID in the app's network, which shows that you belong and lets others contact you directly.
- It allows you to prove that you authored the things you said you authored.
- It allows others to detect third-party tampering with your data.
- It allows others to encrypt and send you data that only you can decrypt.

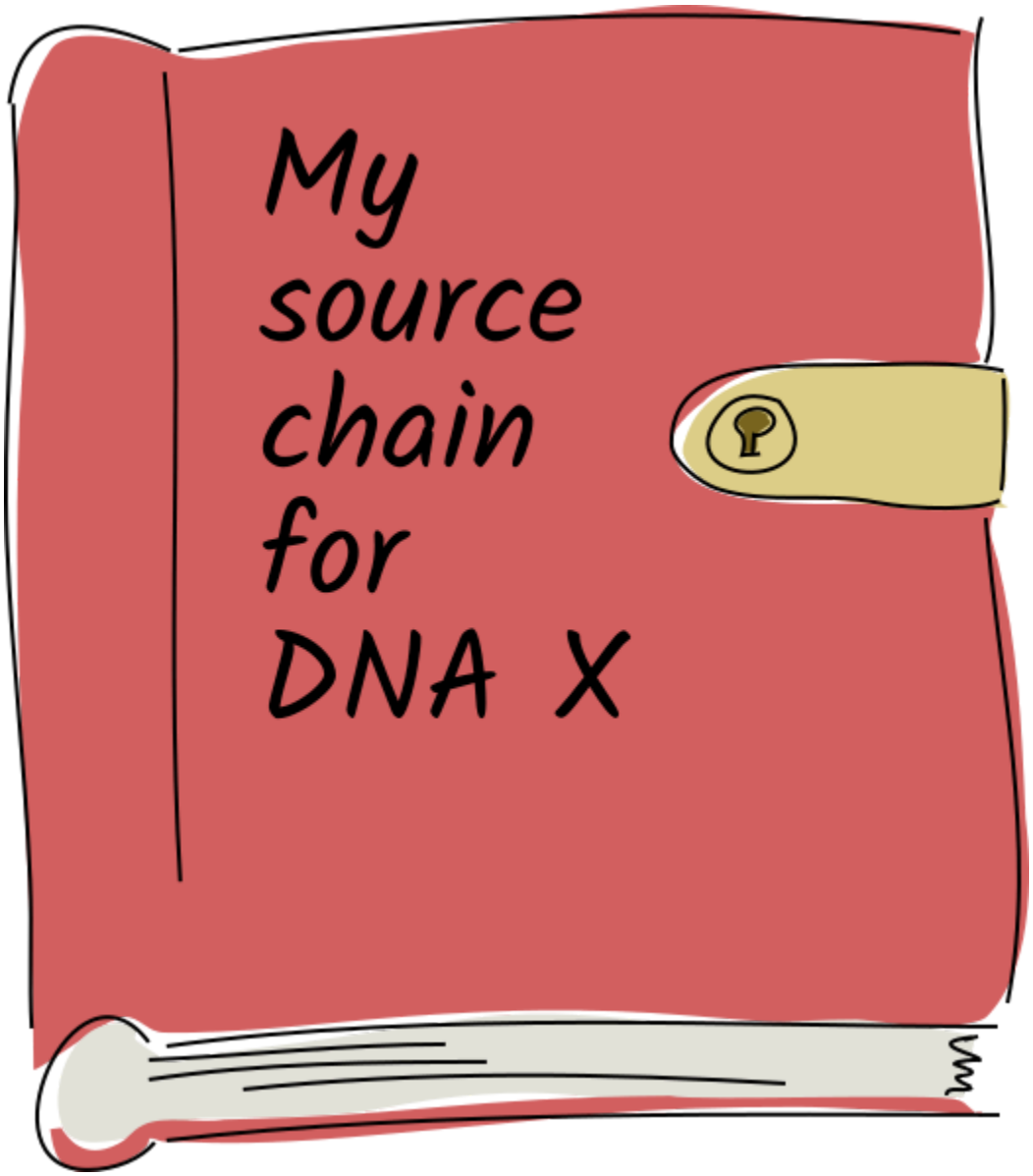
Your conductor generates and stores all your key pairs in an encrypted, password-protected file. This table shows how the public and private keys are used.

Private Key	Public Key
• Stays **secret** on your device	• **Shared** with all your peers on the network
• Acts like a **password**—only you have it, and it's necessary for proving ownership of your public key	• Acts like a **user ID**—uniquely identifies you to other users
• Acts like a **royal seal**—creates unforgeable, tamper-evident digital signatures on your data	• Allows others to **verify the integrity** of your signatures

• Acts like a **mailbox key**—opens messages sealed with your public key

• Acts like a **mail slot**—allows others to encrypt and send data that only you can unlock

Source chain: your own data store

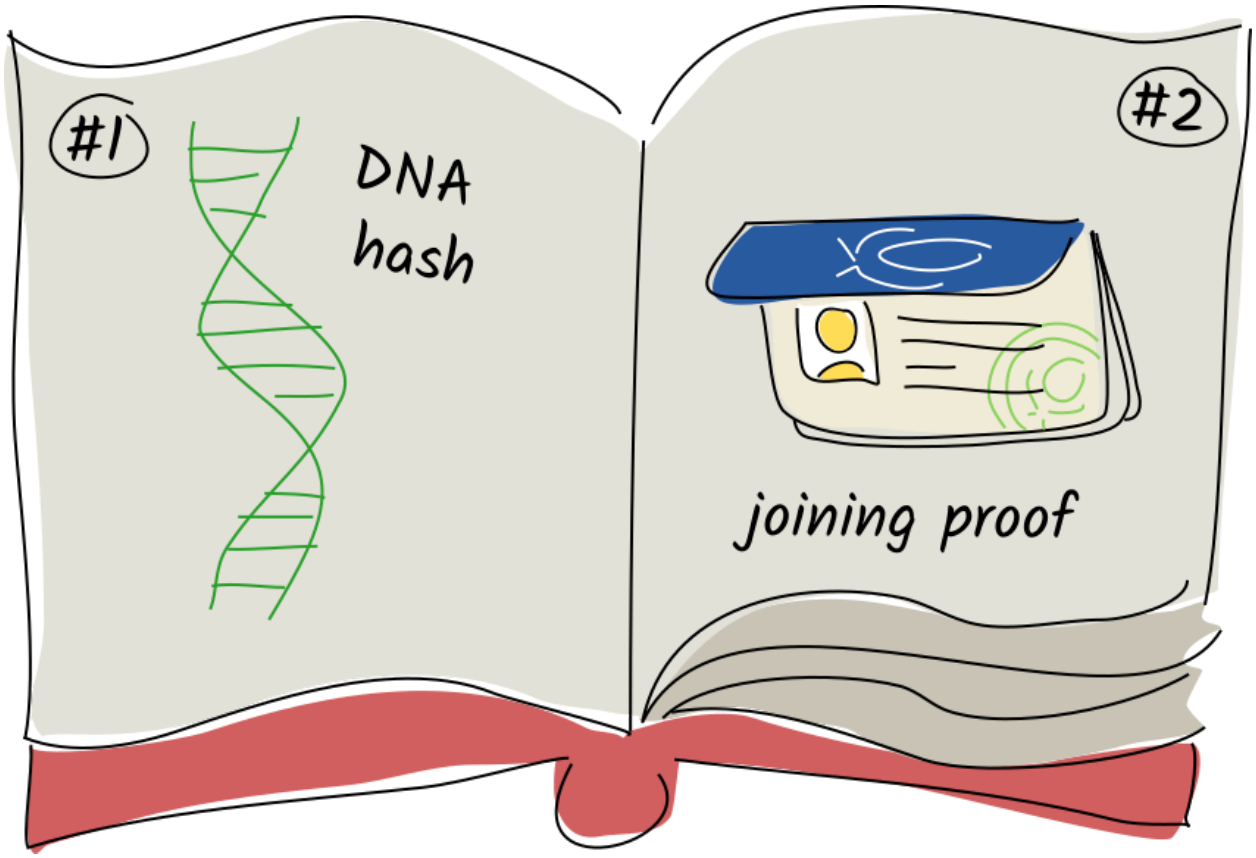


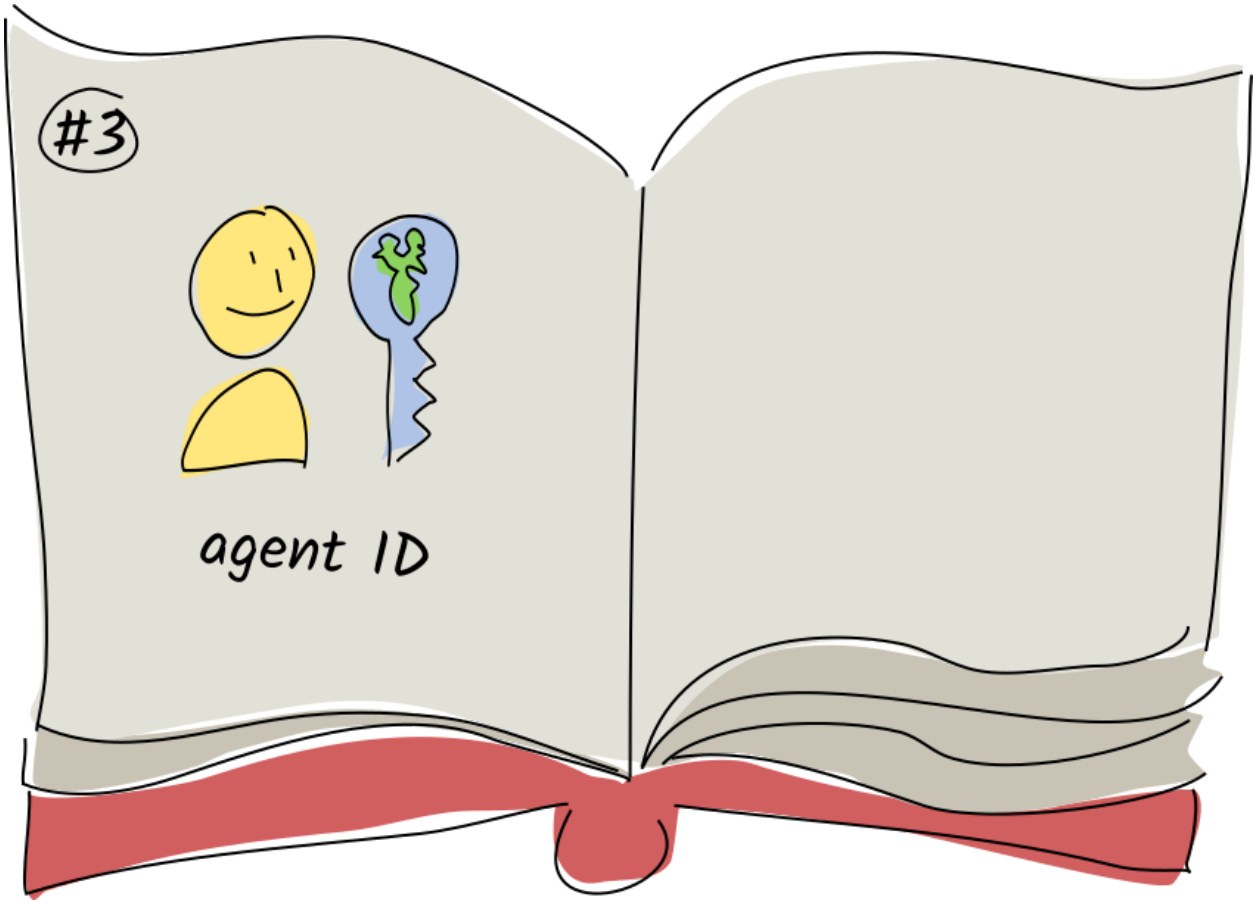
The next stone in the pillar is a chronological journal of every action that the user has performed in their copy of the app—changing their public key; creating, updating, or deleting public or private data, linking data together, and more. Only the user has the authority to write to it; it lives

on their device and each entry must be signed by their private key. This journal is called a **source chain** because every piece of data in an app has its source here.

The user's actions are stored in the source chain as **elements**, which consist of a **header** (we'll get to that later) and usually some sort of binary data (an **entry)**. Each entry has a **type** that distinguishes its purpose, similar to an object-oriented class or database table schema. As with a traditional database, you can define entry types for whatever you need to record—things (such as user profiles, messages, or pictures) or actions (such as chess moves, transactions, or votes).

This journal starts with three special system elements:


#1
DNA
hash
#2
joining proof



1. **The hash of the DNA**. Because the DNA's executable code constitutes the 'rules of the game' for everyone in the app, this element shows that your Holochain runtime has seen and agrees to abide by those rules.
2. **The agent's 'joining proof'**. When an agent tries to join this DNA's peer-to-peer network, it shares this entry with the existing peers, who check it and determine whether the agent should be allowed to join. Examples: an invite code, an employee ID signed by the HR department, or a proof of paid subscription fees.
3. **The agent ID**. This contains your public key as a record of your digital identity. The signatures on all subsequent entries must match this public key in order to be valid.

After this come the elements that record the user's actions. These include:

- The creation, modification, and deletion of **app entries** (user data)
- The creation or deletion of **links** between data

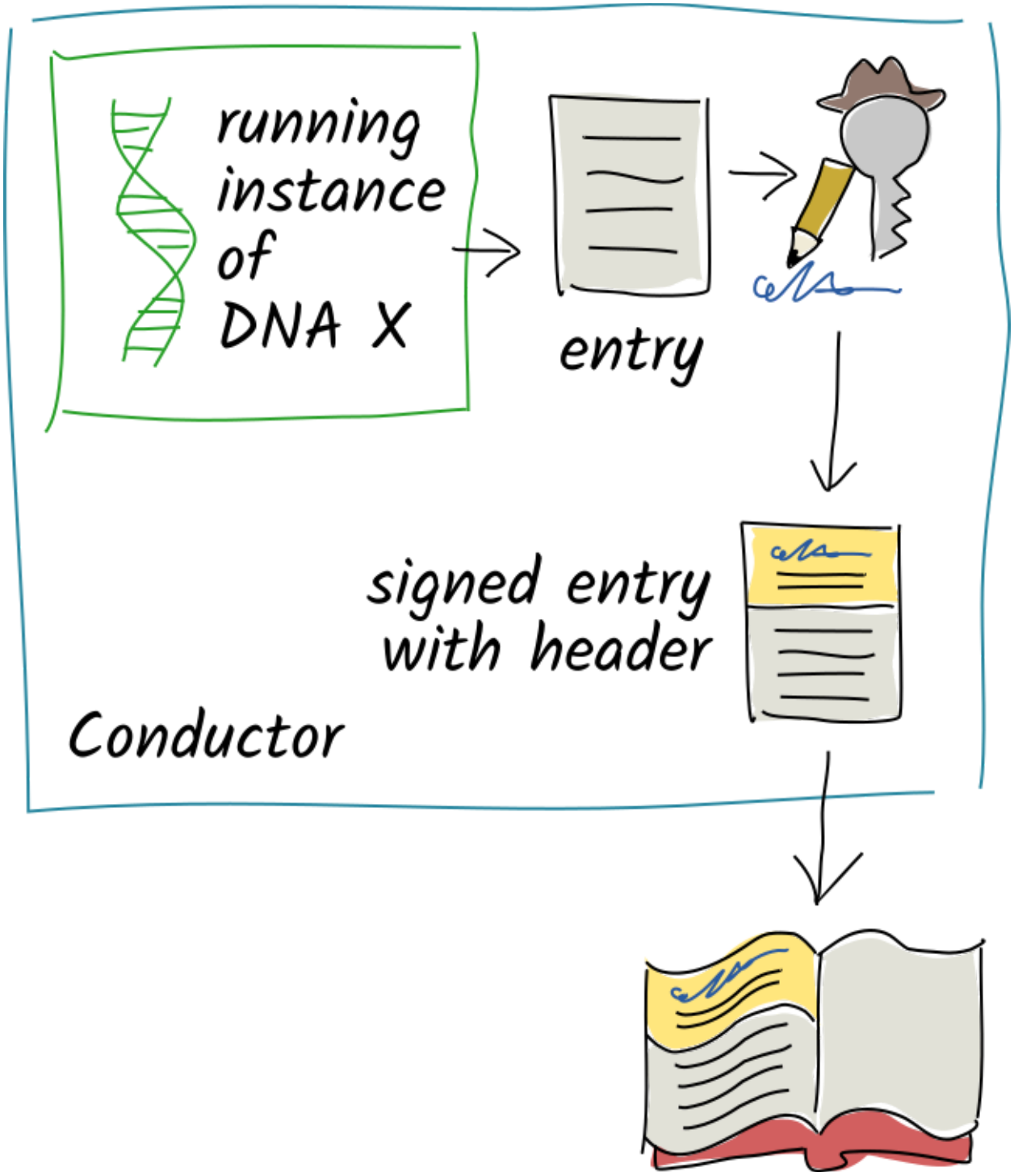
(Other special system actions show up too, but we'll get to those later.) As a developer, you define the format and **validation rules** for each type of app entry or link that your DNA deals with. An entry can contain any sort of binary data, but most of the time you'll want to give it

structure using some sort of serialization format. Our SDK gives you the tools to automatically convert from Rust data structures to MessagePack and back again.

An element on your source chain cannot be modified once it's been committed. This is important, because your source chain is a record of all the things you've done in the app, and your peers may need to check this record in order to validate that you're playing by the rules.

Detecting third-party tampering

If the integrity of your data is so important, what might happen if a third party tried to mess with it en route to your true love or business partner? The answer is, *not much*. Let's take a look at why.



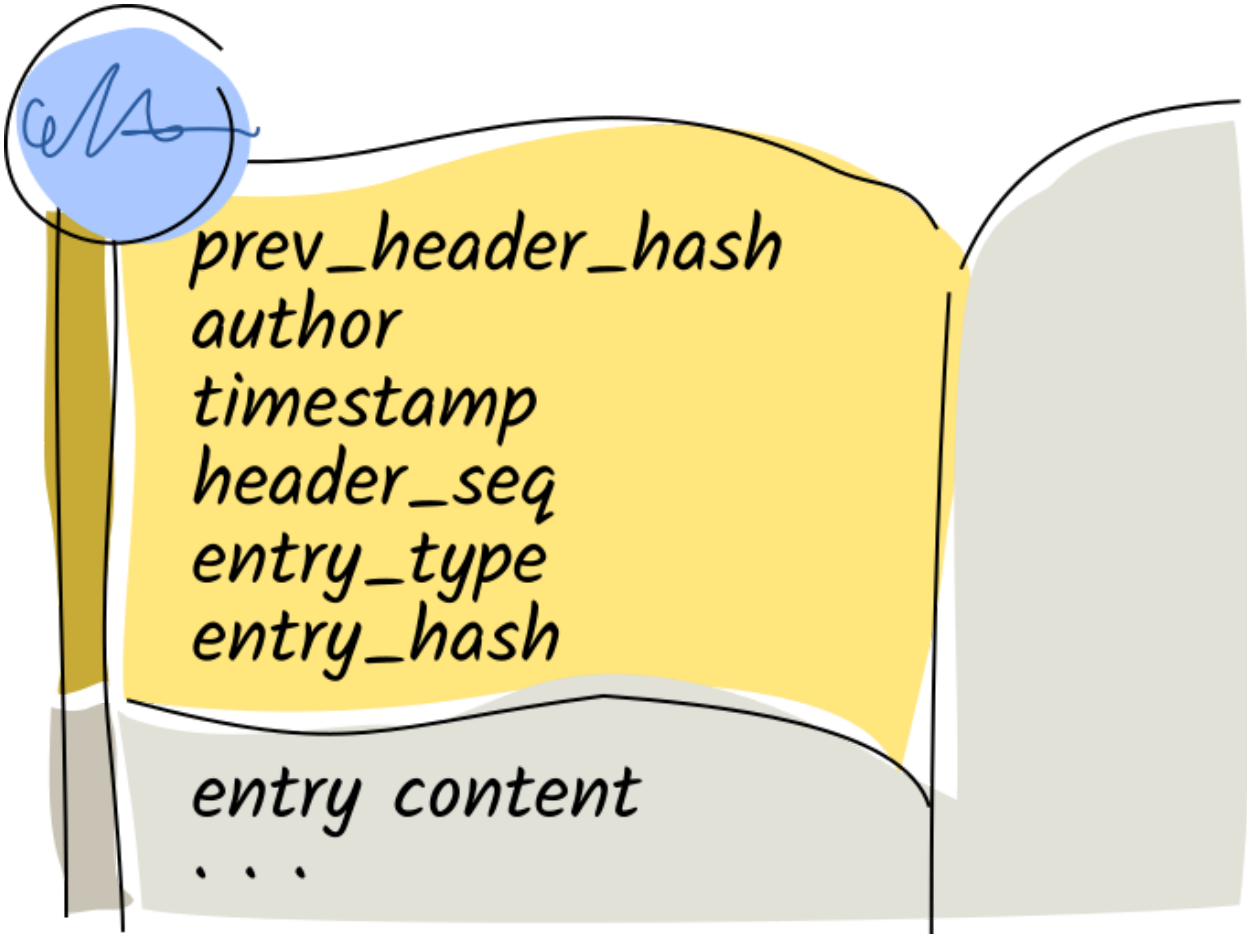
1. When a function in the DNA wants to record a user action, it creates an element containing the details of that action.
2. Then the conductor calls the DNA's validation function for that element. If it fails validation, it returns an error to the client.

3. If validation is successful, the conductor signs the element with the user's private key and writes it onto the end of their source chain.

Like a real-life signature, the signature proves that it was you who created the element. It's based on cryptographic math, so it's verifiable and impossible to forge. It's also only valid for the contents of that element—if a third party modifies even a single character of the signed data, the signature no longer matches. It's like a fingerprint for the data, combined with the author's own fingerprint.

This lets us detect man-in-the-middle attacks on *element data*, but it still doesn't tell us whether anyone has tampered with the *order of elements* in the source chain.

Let's take a closer look at the header. Along with the signature, it includes the hash of the previous header, a timestamp, and the entry's type.



Let's look even more closely at that first line in the header.

This hash is the unique cryptographic ‘fingerprint’ for the previous element’s data. This is what ensures the integrity of the entire source chain. Each element points back to its previous entry. With a paper journal, it’s obvious when someone’s ripped out a page, glued a new page in, or taped a sheet of paper over an existing page. This chain of hashes is the digital equivalent: if anyone so much as modifies a single character in an element, all subsequent elements will be invalidated.

This is great for helping people detect third-party tampering. But unfortunately anyone can modify their own source chain, regenerate the hashes and signatures, and create a perfectly valid, but wrong, alternate history for themselves. For some applications, this wouldn’t matter so much, but it gets quite serious when the source chain holds things like financial transactions or ballots. So how do we tackle this problem?

Holochain’s answer is simple—*somebody will notice*. More on that in the next concept!

Key takeaways

- Holochain apps do not use logins or password databases. Instead, users create their own digital identifiers as cryptographic public/private key pairs. These two keys together prove their possession of their online identity.
- Users share their public keys with other participants in the same app.
- Users prove authorship of their data via impossible-to-forge digital signatures created with their private key. Third-party data tampering is detected by using the author’s public key to verify the signature.
- The user’s source chain is a chronological record of all the data they’ve produced in an app. It lives on their device.
- Data is stored in the source chain as elements — headers and entries.
- Every entry has a type.

- The first three entries are called genesis entries, and are special system types that contain the DNA hash, the agent's joining proof, and their public key.
- Entries are just binary data, and MessagePack is a good way to give them structure.
- The source chain and all of its data is tamper-evident; validators can detect third-party attempts to modify it.

Learn more

- Guidebook: The local source chain
- dApp Planning: Crypto Basics
- Learn Cryptography: what are hash functions?
- Wikipedia: Hash chain
- Wikipedia: Public key cryptography
- Wikipedia: Man-in-the-middle attack

04. The DHT: A Shared, Distributed Graph Database

Agents share records of their actions, including any data meant to be shared with the group, in a **distributed hash table (DHT)**. This graph database provides redundancy and availability for shared data and gives the network the power to detect corruption.

What you'll learn

1. The downsides and risks of self-owned data
2. How public data is stored
3. How nodes store and retrieve data in a distributed database
4. What happens when the network goes down
5. How public data is validated

Why it matters

If you're unsure whether a distributed network can provide the same integrity, performance, and uptime guarantees as a server-based system under your control, this article will give you a better picture of how Holochain addresses these issues. You'll also have a better understanding of how user data is stored, which will help you think about how to design your persistence layer.

Self-owned data isn't enough

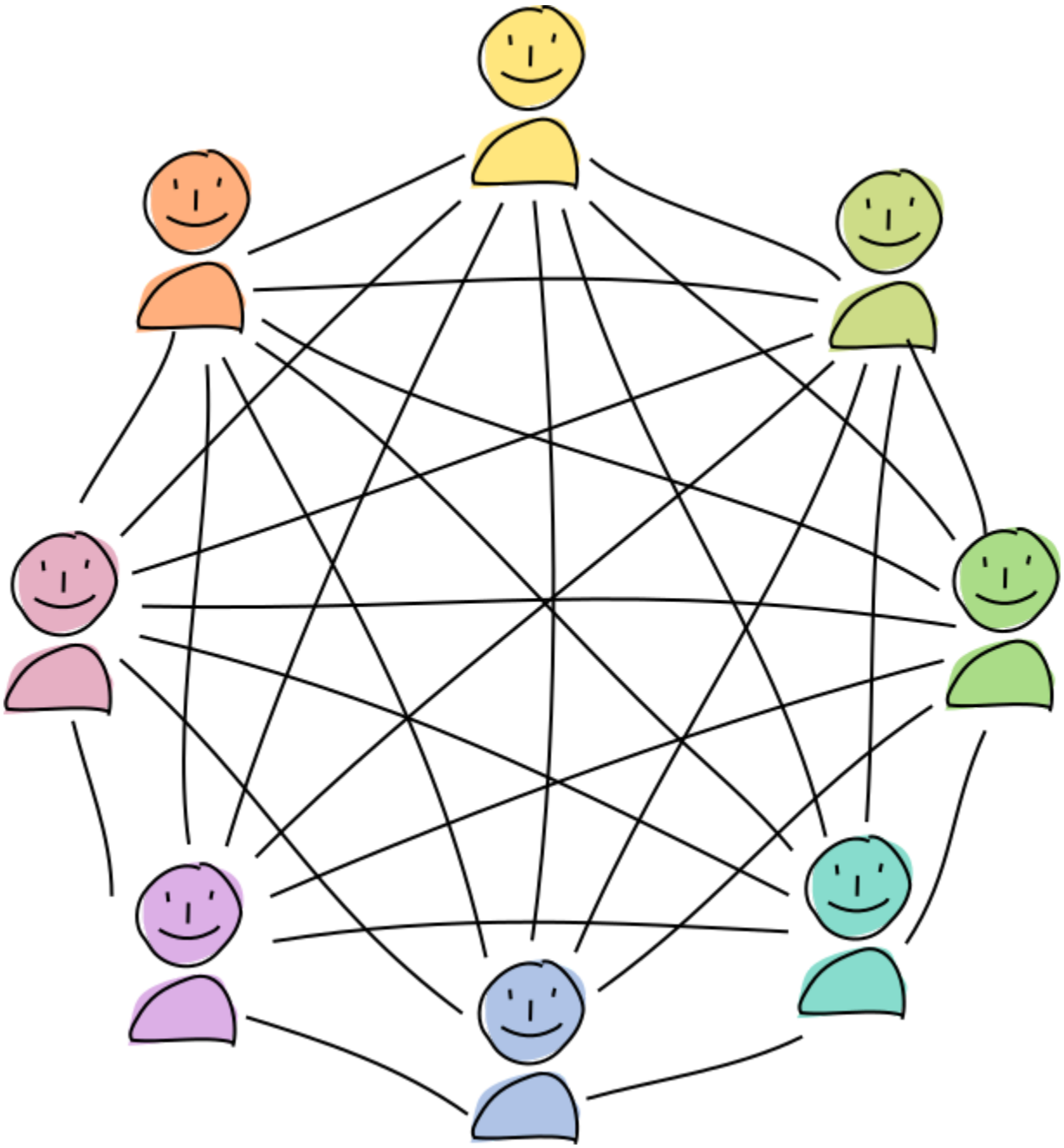
Let’s talk about your source chain again. It belongs to you, it lives in your device, and you can choose to keep it private.

However, the value of most apps comes from their ability to connect people to one another. Email, social media, and team collaboration tools wouldn’t be very useful if you kept all your work to yourself. Data that lives on your machine is also not very available—as soon as you go offline, nobody else can access it. Most users don’t want to run their own servers, so there needs to be a way to make public data stick around.

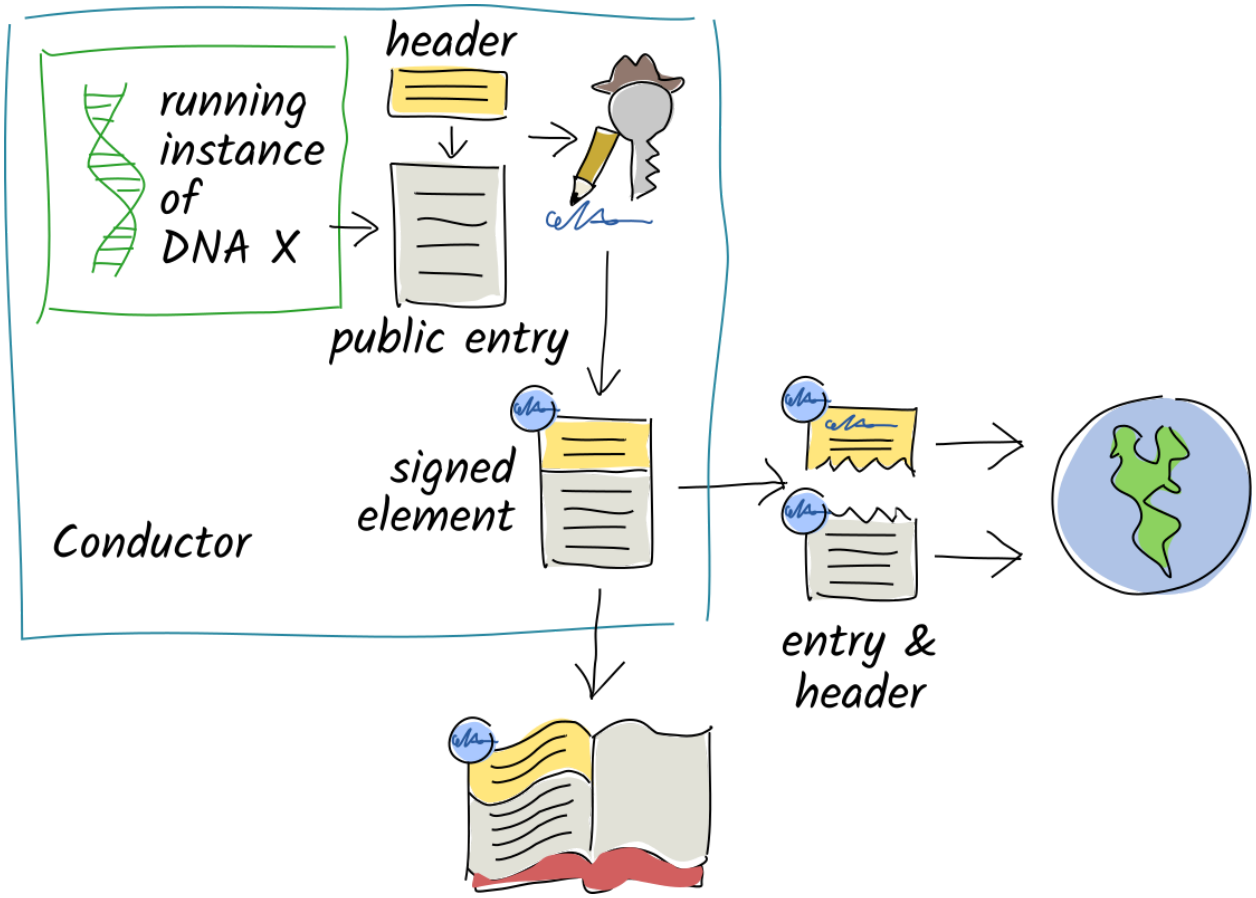
This is also the point where we run into integrity problems in a peer-to-peer system. When everybody is responsible for creating their own data, they can mess around with it any way they like. In the last article, we learned that the signed source chain is resistant to third-party tampering, but not to tampering by its owner. They could erase a transaction or a vote and make it look like it never happened.

And finally, in cases where the validity of a given piece of data depends on a bunch of other data, it can be costly for someone to audit someone else’s data before deciding to engage with them. Because other people have probably checked portions of that same data, it might be possible to save everyone a bit of work.

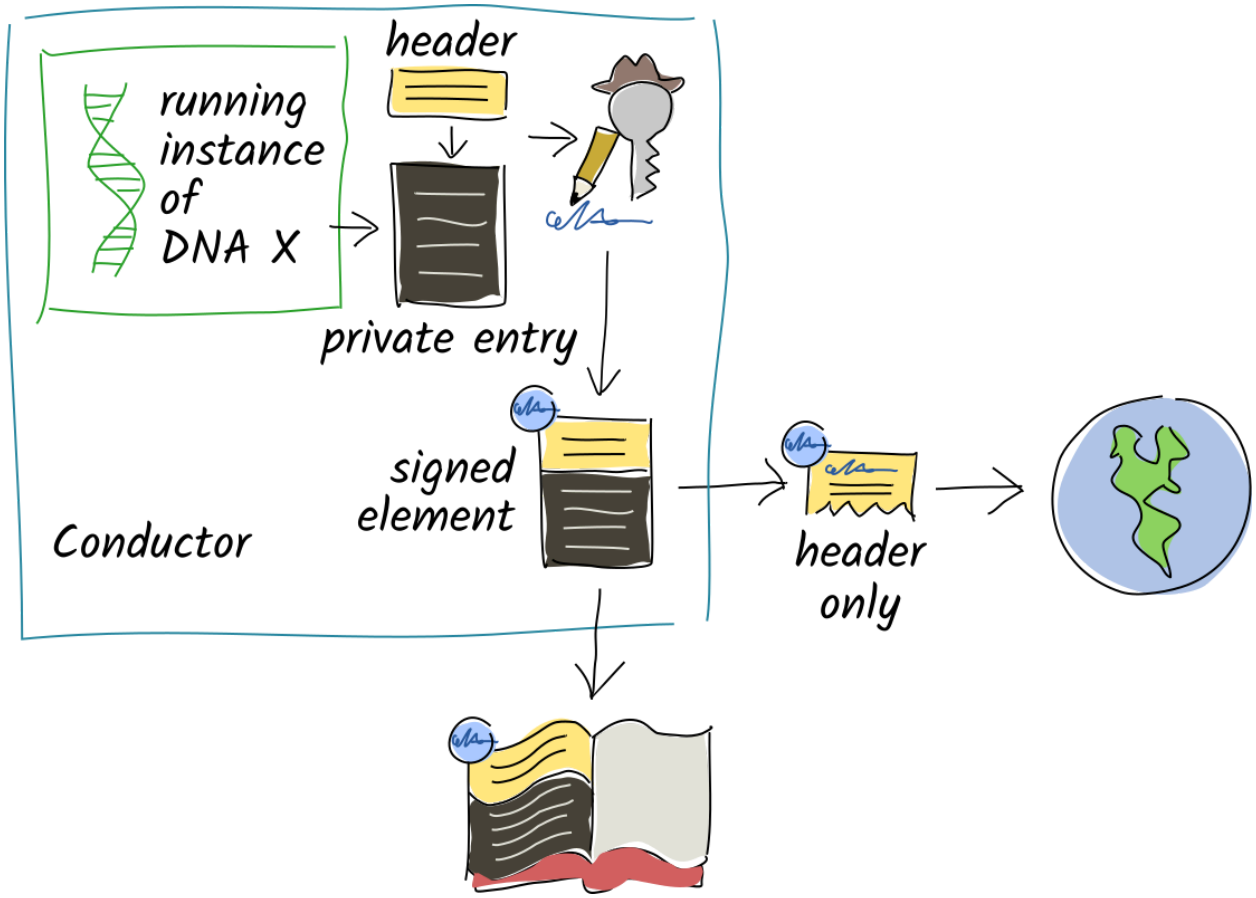
The distributed hash table, a public data store



In a Holochain network, you share your source chain headers and public entries with a random selection of your peers, who witness, validate, and hold copies of them.



When you commit an element with private data, the data stays in your source chain, but its header is still shared.



Every DNA has its own isolated, encrypted network of peers who **gossip** with one another about new peers, new data, invalid data, and network health.

This network holds a distributed database of all public data called a **distributed hash table (DHT)**, which is basically just a big key/value store. Each node is responsible for only a small **shard** of the DHT, so the burden of participation isn't painful for any one agent.

Finding peers and data in a distributed database

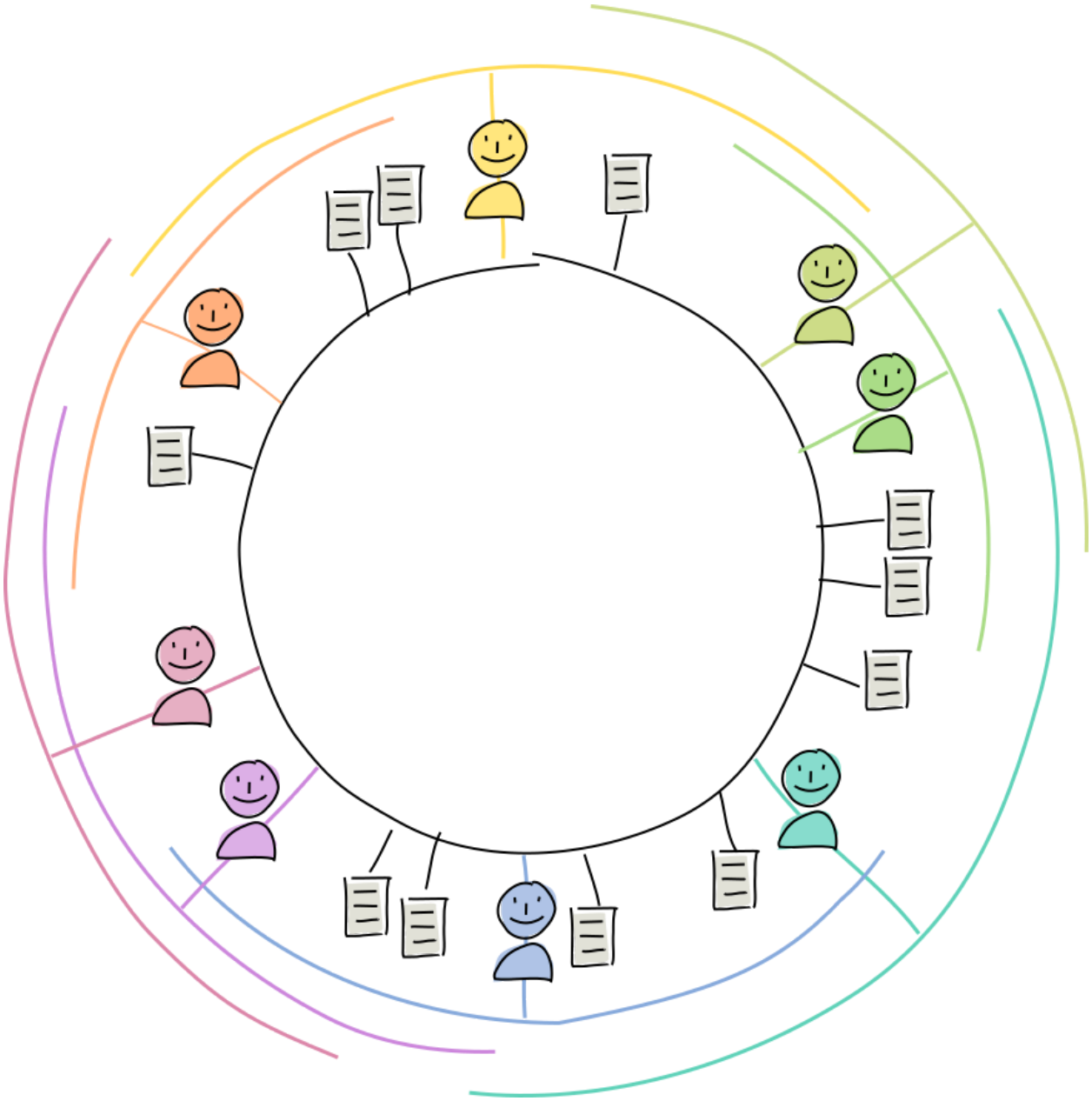
Databases that spread their data among a bunch of machines have a performance problem: unless you have time to talk to each node, it's very hard to find the data you're looking for. DHTs handle this by assigning data to nodes by ID. Here's how Holochain's implementation works:

1. Each agent gets their own DHT address, based on their public key. Each piece of data also gets its own DHT address, based on its hash. These addresses are just huge numbers, so they can be compared to each other. And because they're derived directly from the data they represent, they can't be forged or chosen at will.
2. Each agent chooses the size of the DHT neighborhood they want to be an authority for, as a range to the left and right of them in the DHT's address space. They commit to

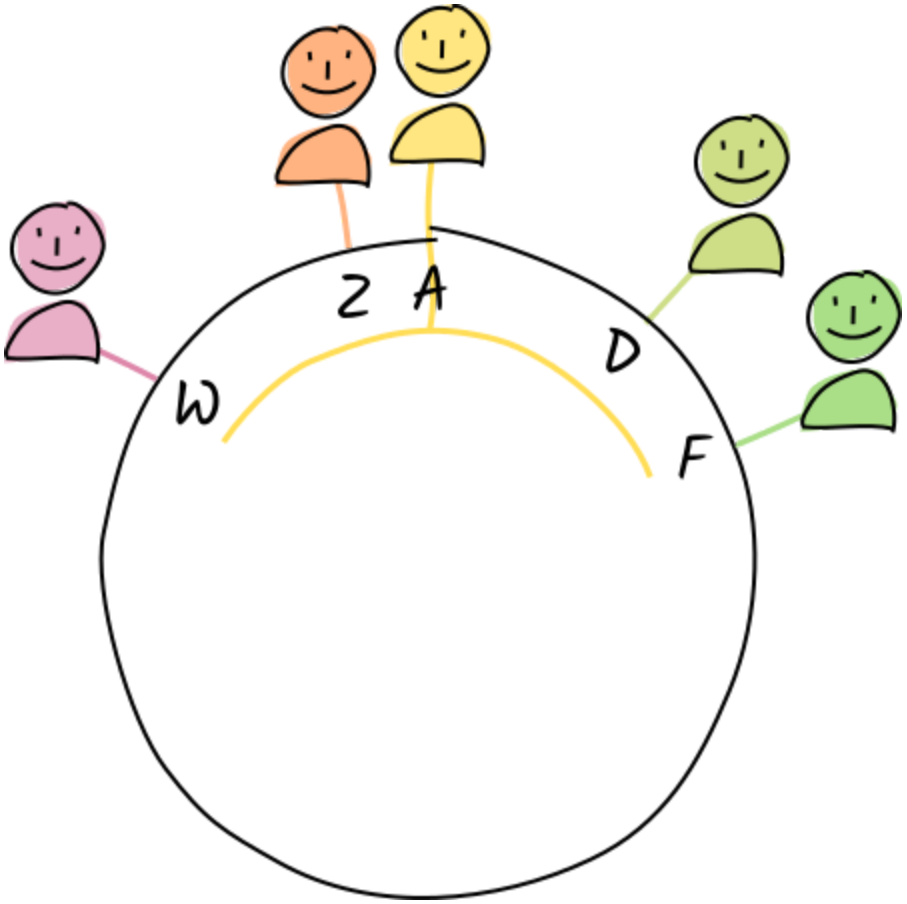
keeping track of all the peers and storing all the data whose addresses fall in that neighborhood.

3. When an agent wants to publish or retrieve a piece of data, they need to figure out who is most likely to be an authority for its address.
4. The agent looks in their own list of peers and compares their addresses against the data's address.
5. The agent sends their **publish** or **get** request message to the top candidates.
6. If any of those candidates are authorities for that address, they respond with the requested data or a confirmation of storage.
7. If none of the candidates are authorities, they look in their own lists of peers and suggest peers who are more likely to be authorities. The agent then contacts those peers and the cycle repeats until the true authorities are found.
8. If the request was a 'publish' request, the authorities 'gossip' the data to other authorities in their neighborhood to increase redundancy of the data.

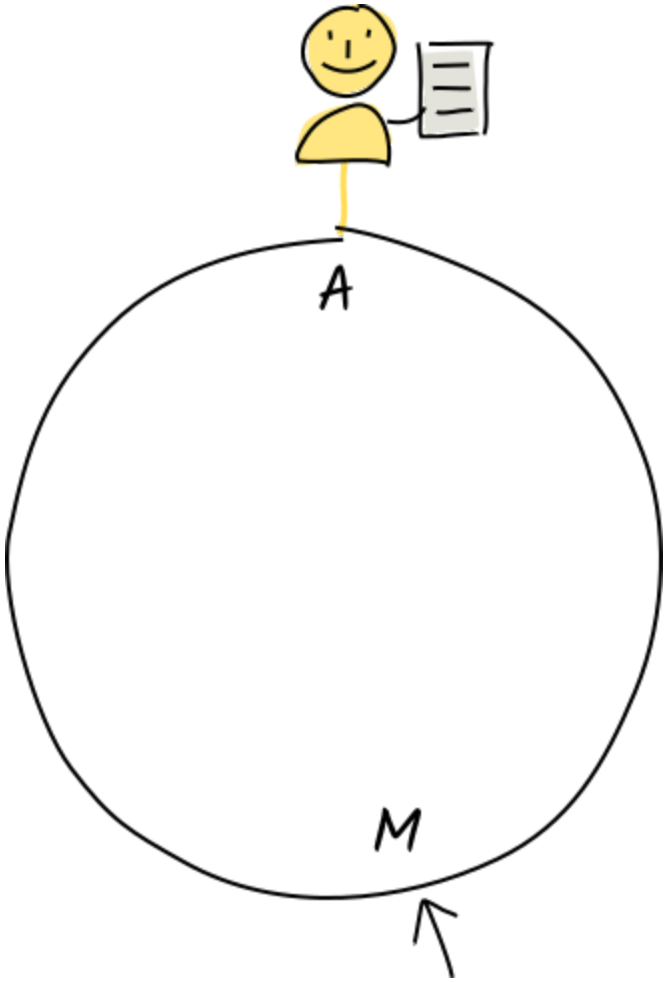
Every agent knows about its neighbors and a few faraway acquaintances. Using these connections, they can find any other agent in the DHT with just a few hops. This makes it fairly quick to find the right authorities for an address.



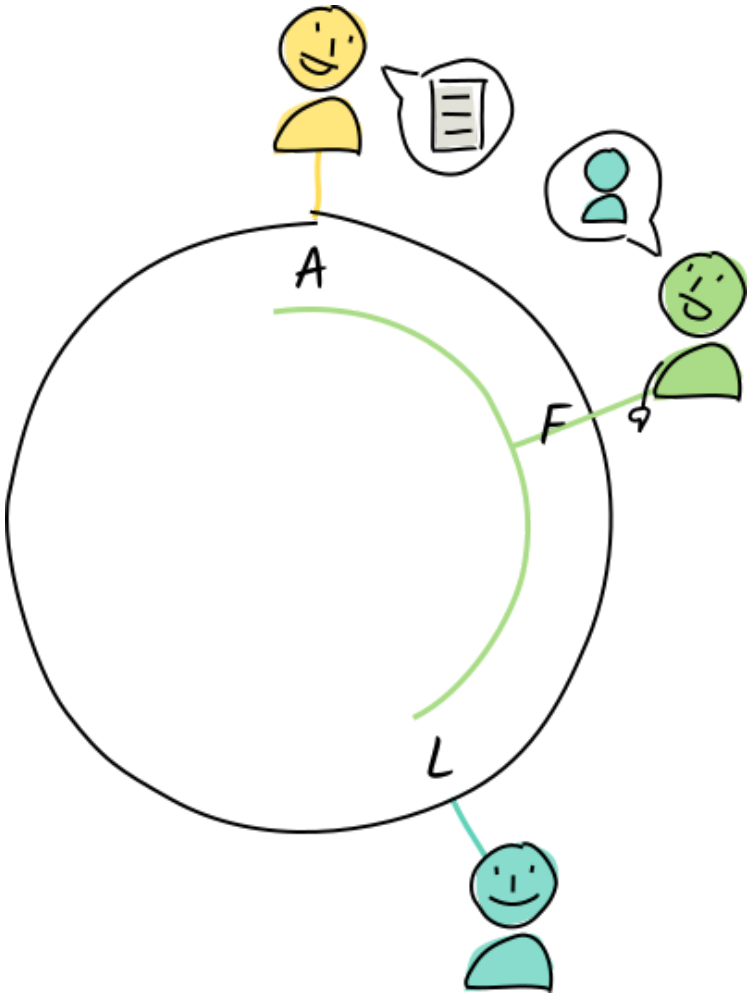
Let’s see how this works with a very small address space. Instead of public keys and hashes, we’re just going to use letters of the alphabet.



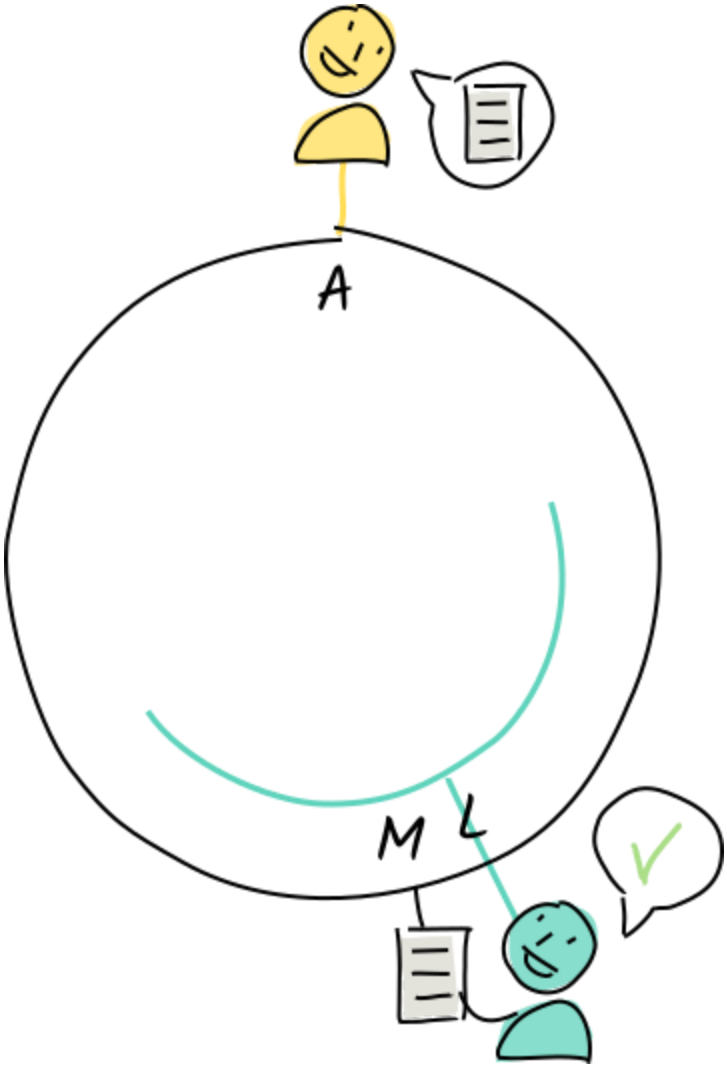
1. Alice lives at address A. Her neighbors to the left are Diana and Fred, and her neighbors to the right are Zoe and Walter.

2. 

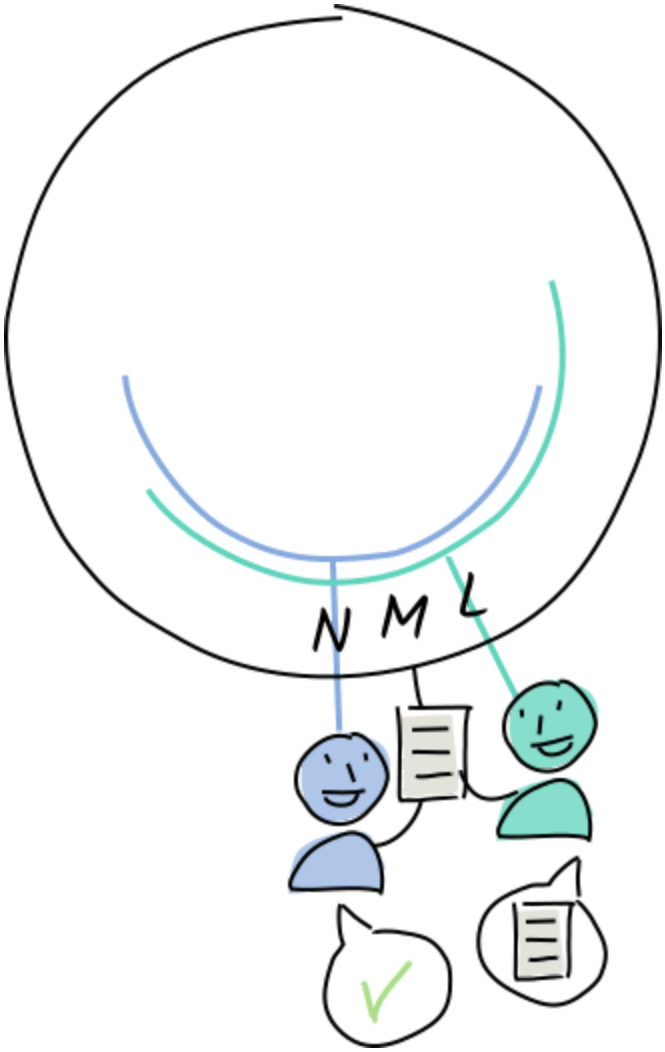
3. Alice creates an entry containing the word “molecule”, whose address is M.



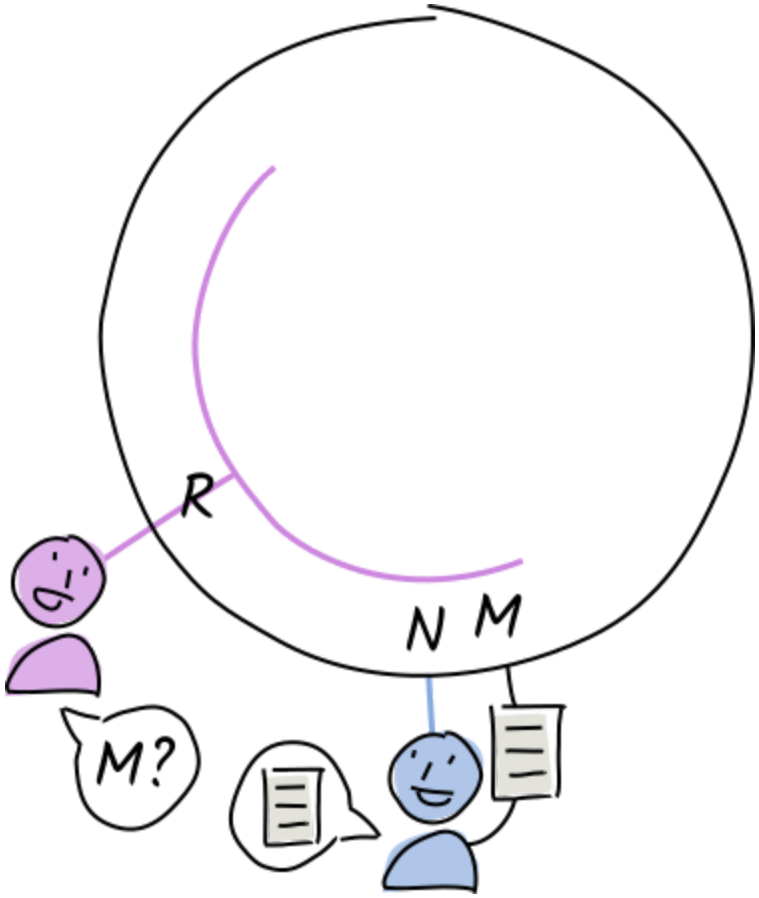
4. Of all of Alice's neighbors, Fred is closest to that address, so she asks him to store it. Fred hasn't claimed authority for that address, so he tells Alice about his neighbor Louise.



5. Alice shares the entry with Louise, who agrees to store it because her neighborhood covers M.

6. 

7. Louise shares it with her neighbor, Norman, in case she goes offline.

8. 

Rosie is a word collector who learns that an interesting new word lives at address is M. She asks her neighbor, Norman, if he has it. Louise has already given him a copy, so he delivers it to Rosie.

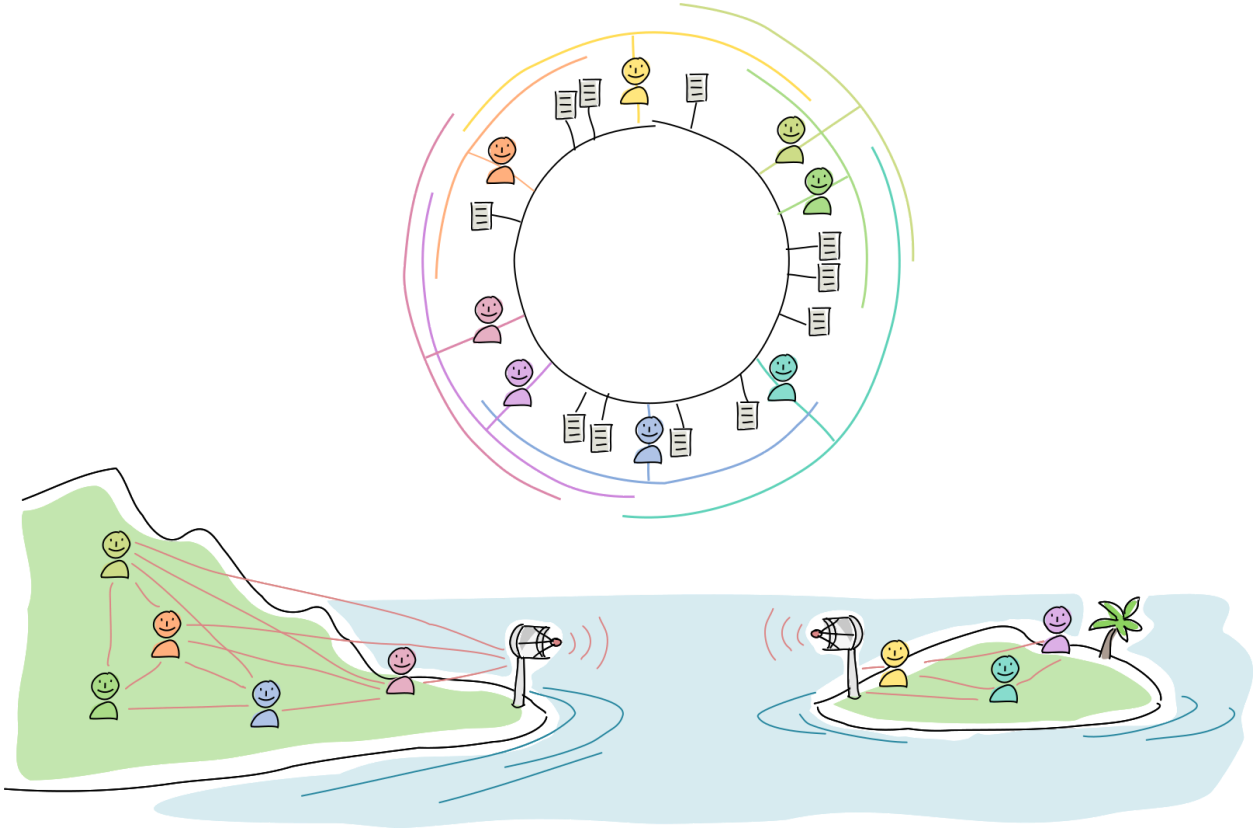
Resilience and availability

The DHT stores multiple redundant copies of each entry so that the information is available even when the author and a portion of the authorities are offline. This allows others to access it whenever they need to.

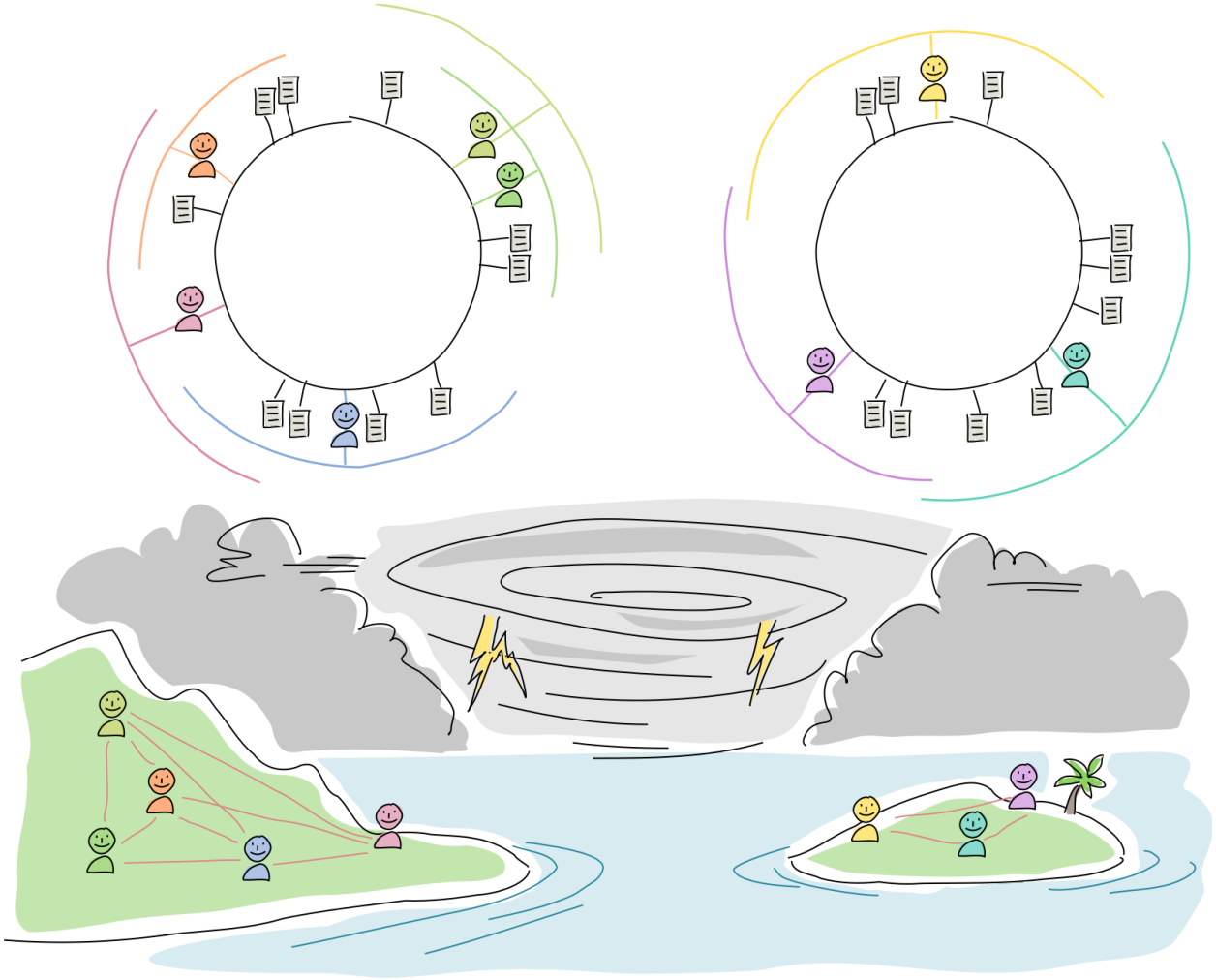
It also helps an application tolerate network disruptions. If your town experiences a natural disaster and is cut off from the internet, you and your neighbors can still use the app. The data you see might not be complete or up to date, but it's still accessible and you can still reach each other. You might even be able to use the app when you're completely offline.

The author of an app can specify the desired data redundancy level. This is called the **resilience factor**. Cooperating agents work hard to keep enough copies of each entry to satisfy the resilience factor. It should be set higher for apps that require higher security or better failure tolerance.

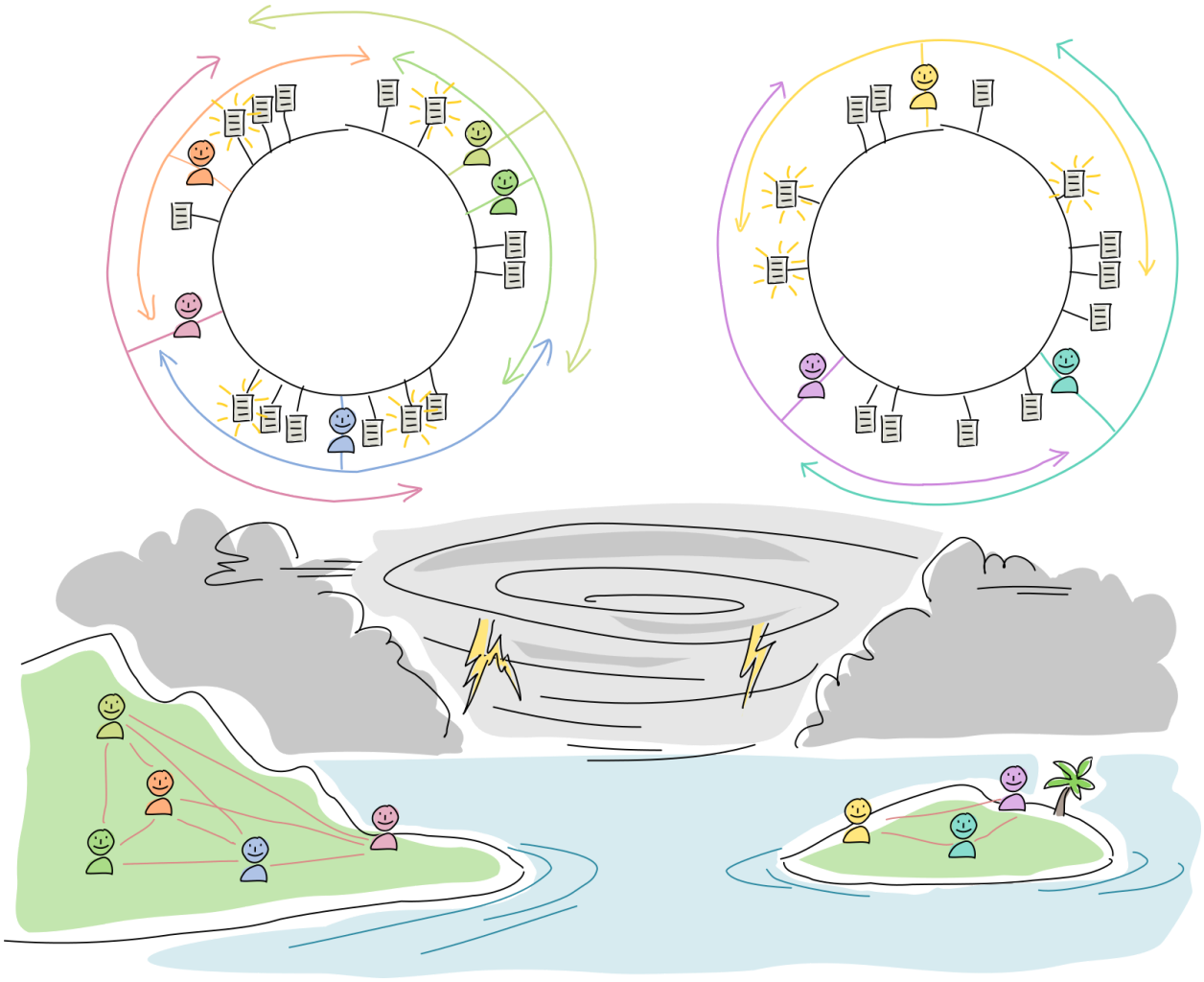
Let’s see how this plays out in the real world.



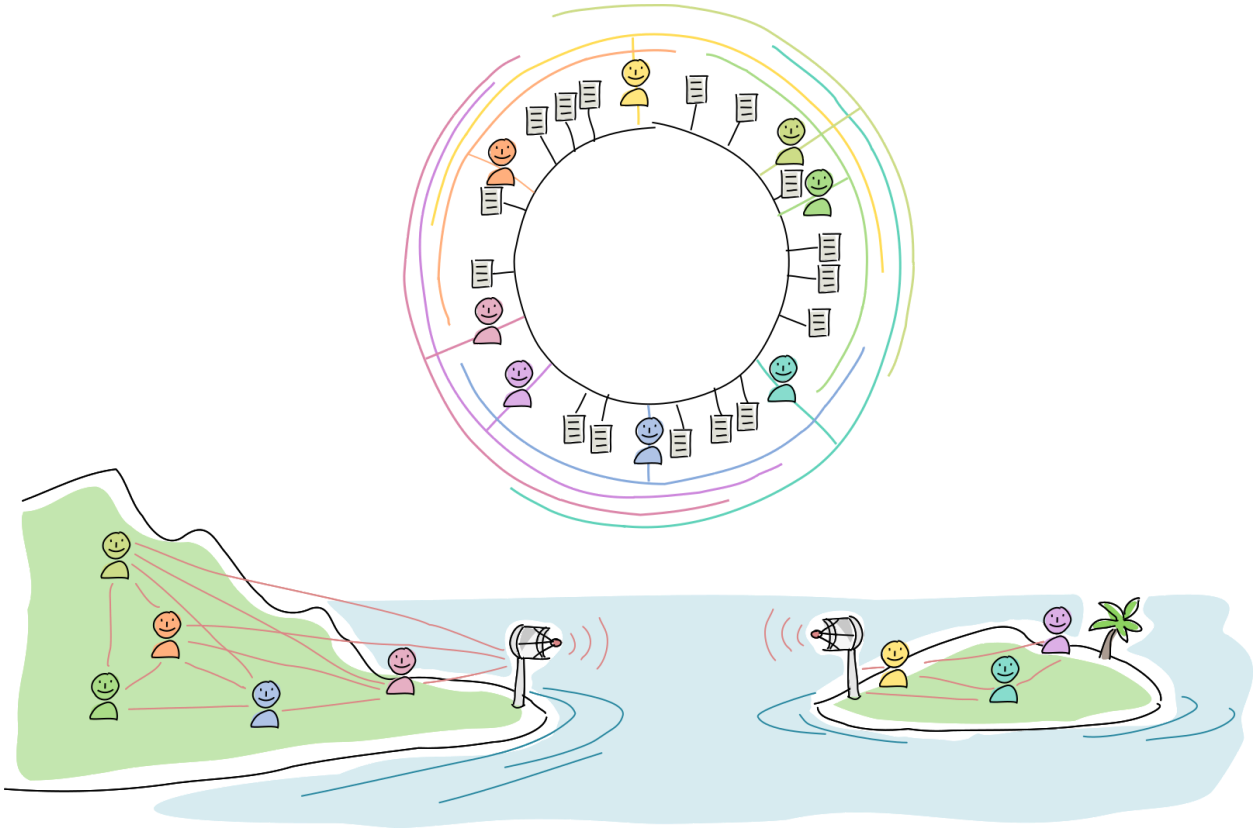
1. An island is connected to the mainland by a radio link. They communicate with each other using a Holochain app.



2. A hurricane blows through and wipes out both radio towers. The islanders can’t talk to the mainlanders, and vice versa, so some DHT neighbours are unreachable. But everyone can still talk to their *physical* neighbors. None of the data is lost, but not all of it is available to each side.



3. On both sides, all agents attempt to improve resilience by enlarging their DHT neighborhoods. Meanwhile, they operate as usual, talking with one another and creating new entries.



4. The radio towers are rebuilt, the network partition heals, and new data syncs up across the DHT.

A cloud of witnesses

When we laid out the basics of Holochain, we said that the second pillar of trust is **peer validation**. When a node is asked to store an entry, it doesn't *just* store it—it also checks it for validity. As the entry is passed to more nodes in its neighborhood, it gathers more signatures attesting to its validity.

There are up to three authorities for each source chain element.

- The **entry authority**, whose address is in the neighborhood of the entry's address, checks the contents of the entry data to make sure it's properly formed and conforms to the validation rules for the entry type. (Some elements, like link creations and deletions, don't have any entry data.)
- The **header authority**, whose address is in the neighborhood of the header's address, checks the contents of the header to make sure the previous header exists on the DHT and the header's sequence ID and timestamp are higher than that of the previous header.
- The **agent activity authority**, whose address is in the neighborhood of the element's author (that is, they're the author's neighbors), receives a copy of the header and checks

that it doesn't conflict with another header — that is, the author isn't trying to modify their history by publishing an alternate one.

All three authorities check the signature on the data they receive to make sure it hasn't been modified in transit and it belongs to the agent that claims to have authored it. If the signature check fails, the data is rejected.

If any other check fails, the authority marks the data as invalid and creates a **warrant**, signed evidence of the agent's invalid action. They then share that warrant with the author's agent activity authorities as a permanent record that they broke the rules. Others can use that warrant to justify ignoring the rule-breaker. This is what creates Holochain's immune system.

Using cryptographically random data (public keys and hashes) as addresses has a couple benefits. First, validator selection is impartial, resistant to collusion, and enforced by all honest participants. Second, the data load is also spread fairly evenly around the DHT.

The important thing is that the DHT *remembers what you've published*, so it's very hard to break the rules without getting caught.

Key takeaways

- Source chain entries can be private (kept on the user's device) or public (shared with the group). Source chain headers are always public. Participants share these with their peers in a distributed database called the DHT.
- A piece of data in a DHT is retrieved by its unique address, which is based on the data's hash.
- Each participant takes responsibility to be an authority for validating and storing a small portion of the public data in the DHT.
- Holochain's DHT is a validating DHT that remembers the validation result of existing entries. This speeds things up for everyone and allows the detection of bad actors.
- Holochain's DHT also detects agents' attempts to roll back their source chains and create alternate histories.
- Authority selection for a piece of data is random and enforced by all honest peers.
- News of bad actors is spread through warrants, which are special pieces of data that carry signed evidence of rule-breaking.
- An app can set a resilience factor, or expected level of DHT redundancy, for each entry type.
- A DHT tolerates network disruptions. It can keep operating as two separate networks and subsequently 'heal' when the network is repaired.

Learn more

- Wikipedia: Gossip protocol
- Wikipedia: Distributed hash table
- Wikipedia: Peer-to-peer

05. Links, Paths, and Anchors: Connecting DHT Data Together

Entries on the DHT are connected to one another via one-way **links**. They allow you to create a graph database on the DHT, making information easy to discover.

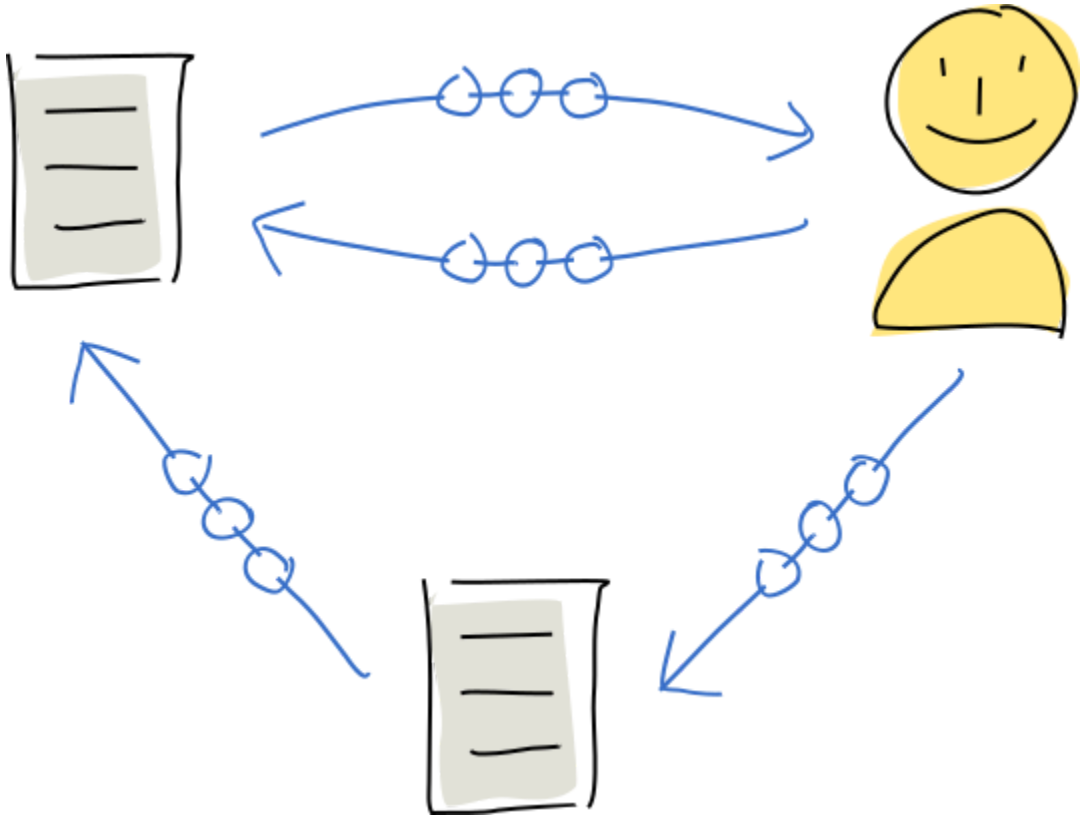
What you'll learn

1. Why it's hard to find data in a DHT
2. How turning it into a graph database makes it easy
3. Starting points you can use to discover data
4. How it looks in real life
5. The dangers of anchors and paths

Why it matters

DHTs and graph databases are different from familiar data stores, like relational databases, key/value stores, and document/object stores. Once you understand how they work, you'll be able to design a robust data model for your app that takes advantage of their strengths and avoids their weaknesses.

The difficulty of looking for data (especially when you don't know what you're looking for)



The use of numeric addresses based on data's content has one big advantage—you can ignore the physical location of data and ask for it by its content address. This means no broken URLs or nasty surprises about what the entry contains.

It does, however, make it hard to find the data you're looking for. Addresses are just random numbers, which doesn't give you much of a clue about the content that lives there. If all data were all on one machine, you could quickly scan through it for a particular substring. But that would get pretty slow on a distributed system, where the data set is spread out over many machines.

This creates a chicken-and-egg problem. In order to retrieve a piece of data, you need to know its address. But you only know the address if:

1. You already have the data and can calculate its address.
2. You find an address and know that there's interesting data there.

Option 1 is an exercise in pointlessness, because if you already have the data you don't need to retrieve it. So let's move on to option 2: how do you find the address of a piece of interesting data?

Links: creating a distributed graph database

Holochain lets you **link** any two addresses together. This lets you *connect known things to unknown things*, which then become known things that can link to more unknown things, and so on. Your app's DHT becomes a **graph database**.

A link is made up of three components:

- The **base** address, or the address from which it is linked. This is the 'known' thing.
- The **target** address, or the address it's linking to. This is the 'unknown' thing.
- An optional **tag** containing extra metadata about the nature of the relationship or target. You can put whatever you like into this tag, then retrieve it or filter on it when you're performing a link query.

Links are stored by the same authority that stores the data at the base address.

Starting points for traversing graphs

What sorts of 'known' things can you use as starting points?

First of all, people can just share addresses (both entry and agent addresses) with each other. Once you receive an address, it becomes a 'known' thing and you can check it for links. But the chicken-and-egg problem hasn't gone away, because you can't share something with someone if you don't know their address yet. You could send addresses by email or pigeon post, but that defeats the purpose of putting everyone together in one communication network.

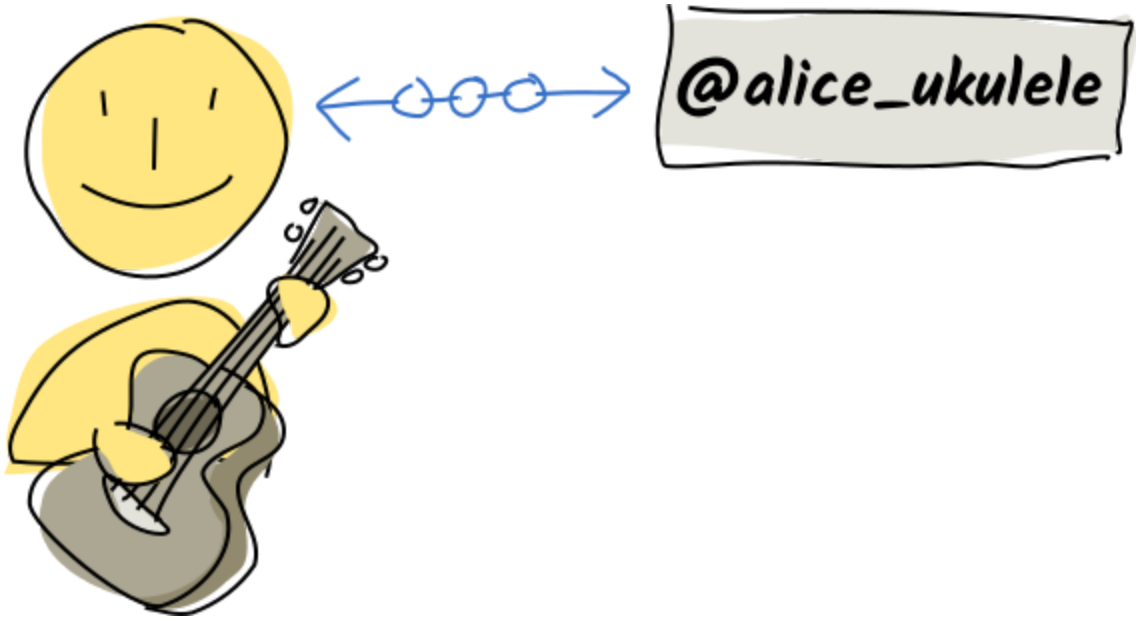
We've introduced **anchors** and **paths** as useful patterns for creating starting points on the DHT's graph database. They're just built from entries and links, like anything else in Holochain, but they're part of the SDK so you don't have to roll your own implementation.

Anchors are just small chunks of data, usually strings, that simply serve as things to attach links to. They can be easy for humans to share and type, such as usernames, or they can be baked into the application as constants.

Paths are just a generalization of anchors. While anchors are meant for simple roots of a graph such as "_all_users_", or two-level hierarchies such as "_all_users_.@alice", paths can be of any length. You can use them for product categories, search indexes, filesystem hierarchies, and more.

Case study: a music sharing app

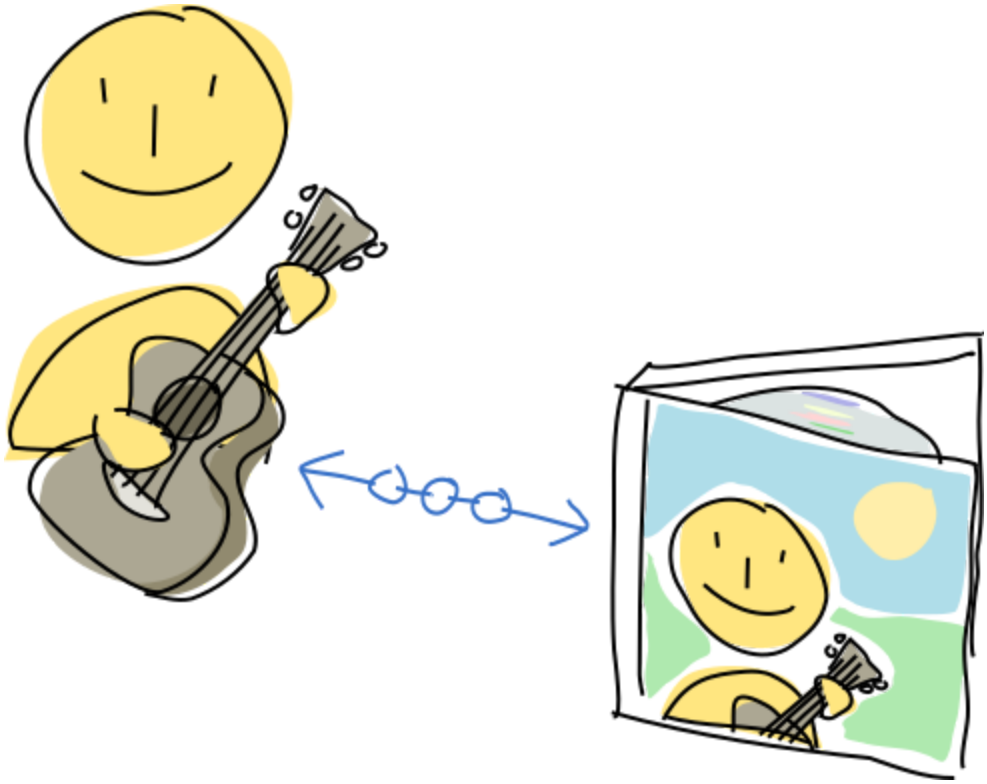
1.  Alice is a singer/songwriter who excels at the ukulele and wants to share her music with the world. She joins the app and chooses to register the username “@alice_ukulele”. She checks if it’s already been taken by calculating its address and looking for an existing username DHT entry with that address.

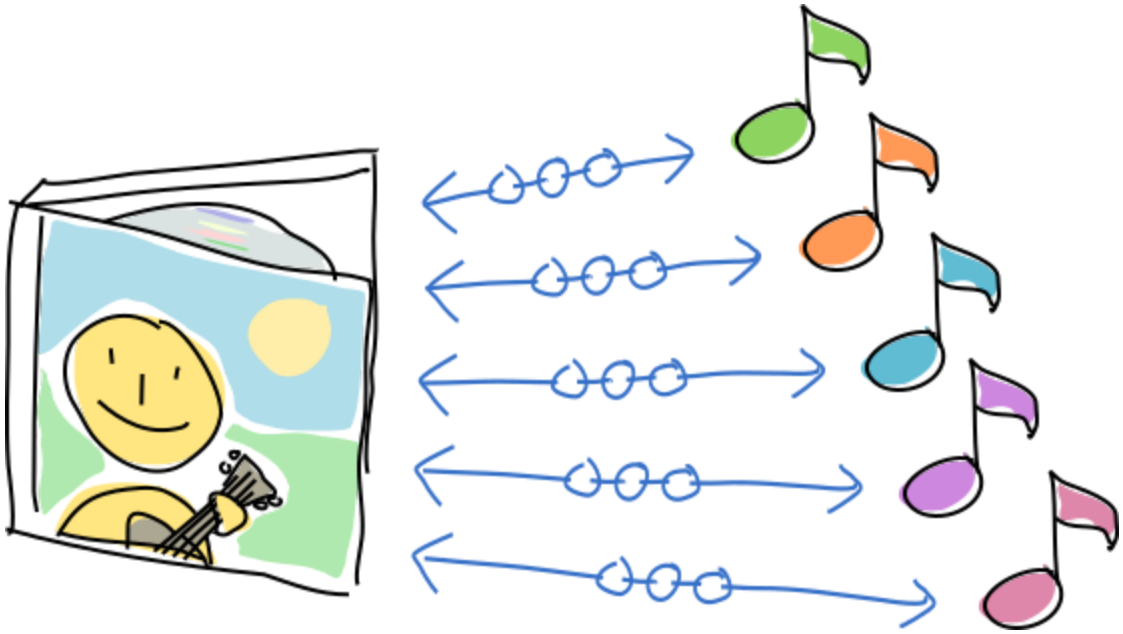
2. 

That entry doesn’t exist, so she publishes it and links it to her agent address. Now, users who know her username can find her agent address.

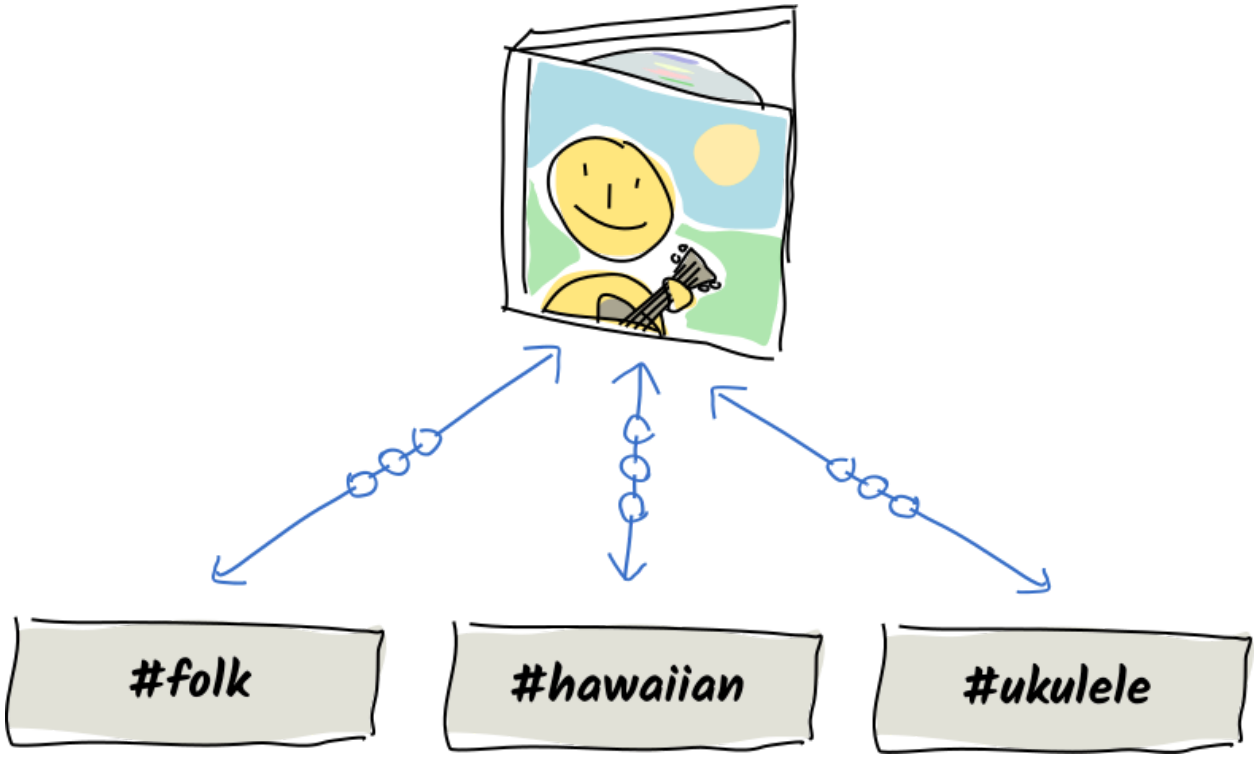
3. 

Alice wants to show up in the public directory of artists, so she links her username entry to the “_all_users_” anchor. This anchor already exists in the DHT and its value is hard-coded into the app. Anyone can query this anchor for a full list of usernames.

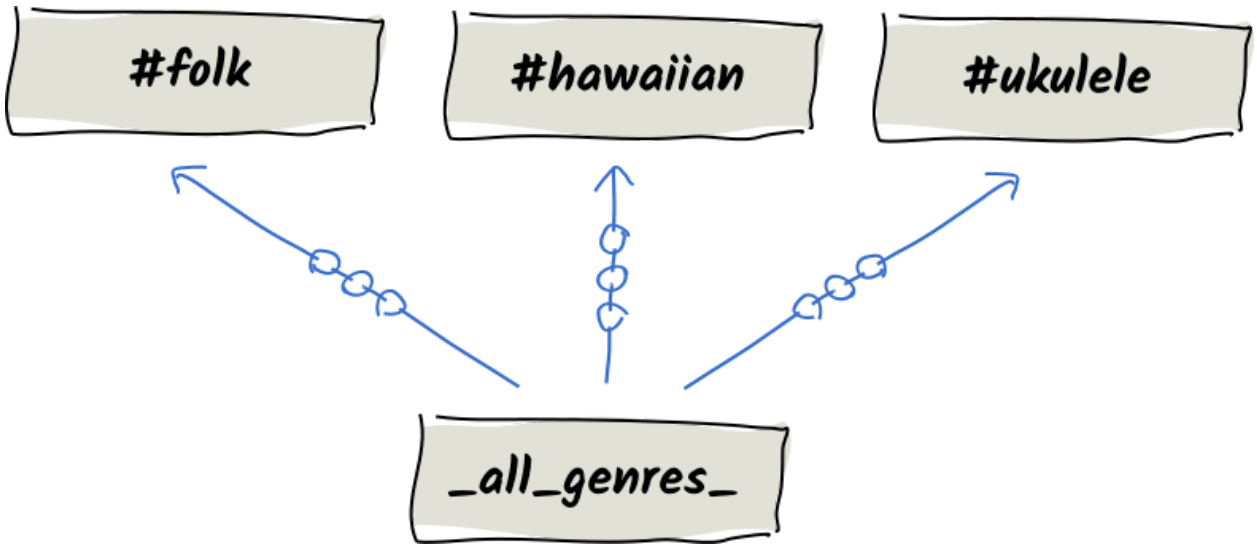
4. 
Alice creates an entry for her debut EP album and links it to her agent address. Now listeners who know her agent address can find the albums she’s published.

5. 

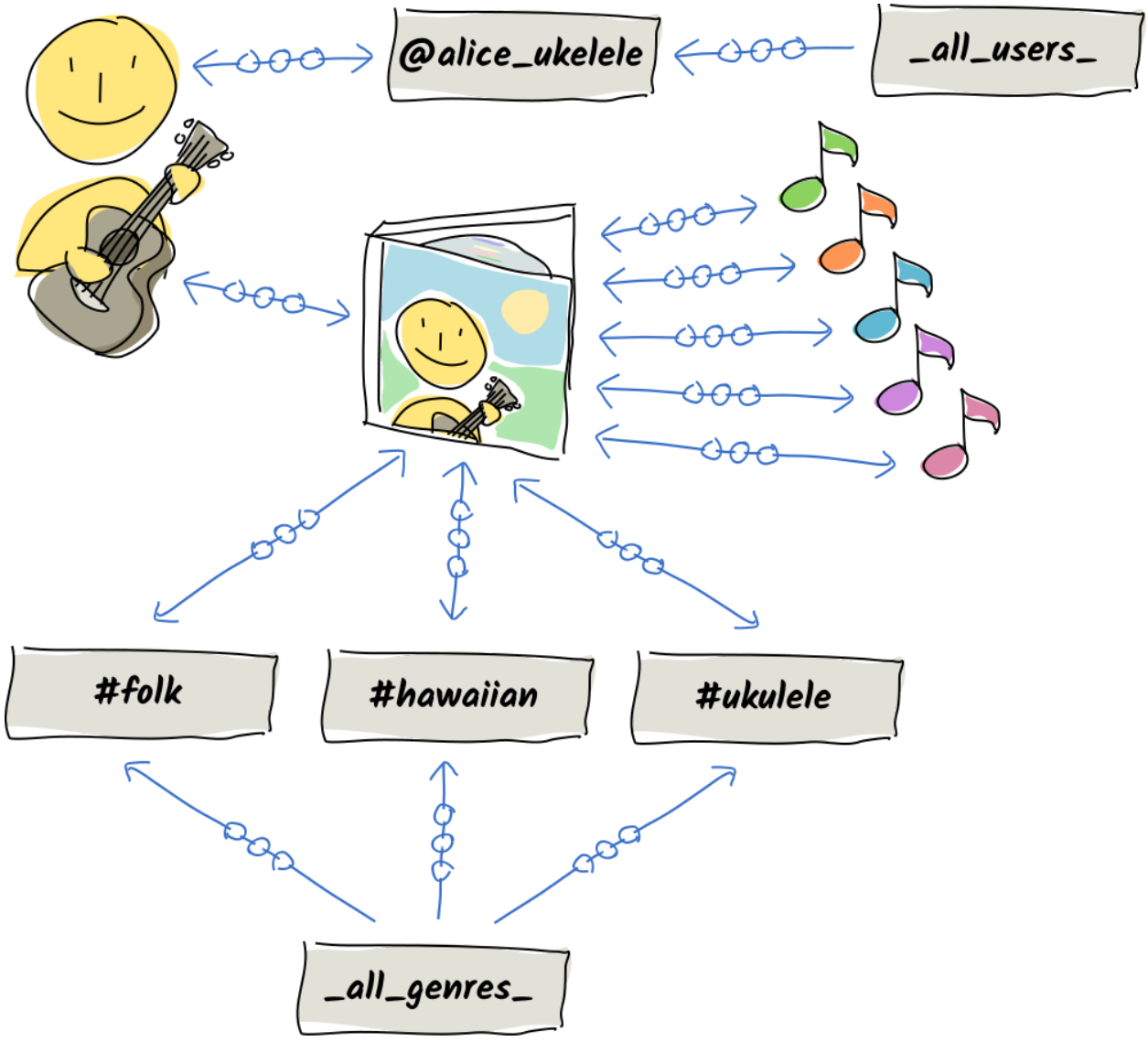
She uploads all the tracks and links them to the album.

6. 

She wants people to be able to find her album by genre, so she selects three applicable genre tags (they're anchors too) and links her album to them.

7. 

   Those genres are already linked to an “_all_genres_” anchor, another hard-coded constant. Listeners can query this anchor to get the full list of genres.

8. 

   Alice's entries, now linked to one another and other existing entries on the DHT, form a graph that allows listeners to discover her and her music from a number of different starting points.

Note

Take note of the arrowheads above; you'll see that most are bi-directional, which means they are actually two separate links going in opposite directions. Hard-coded anchors are the exception because they don't need to be discovered—they're baked into the app so it's easy to calculate their addresses.

The dangers of anchors and paths

Consider the above app once it's reached a million users. The poor authorities responsible for that "_all_users_" anchor will have to store a million links, serve a steady stream of get-links queries for anyone who's searching for a user, and serve a lot of data for each of those requests. This creates a 'hot spot' on the DHT.

We want to avoid this as much as possible by spreading the workload out. Sharding that one anchor by the letters in the username does that nicely: instead of one root anchor with millions of links, you get one root anchor with only 26 links, each of which has 26 links, and so on. As a side benefit, you get fast search indexing for autocomplete, because asking a peer to return a hundred links on the "_all_users_.@ali" anchor is a lot faster than asking them to return a million links on the "_all_users_" anchor.

The Holochain SDK has a simple sharding scheme that's useful in cases like the above. If you want to use a different strategy, such as sharding ISO-8601 timestamps, it's easy to build yourself. After all, anchors and paths are just a pattern, and you can implement that pattern any way you like.

Key takeaways

- It's not possible to do arbitrary queries on a DHT, because entries are scattered across many nodes and can only be retrieved by their addresses.
- Links allow you to connect a known address (the base) to an unknown address (the target) to create a graph database on the DHT.
- Links are stored by the authority responsible for their base's address.
- Links are one-way; you create a two-way relationship with a pair of links.
- Links can have an arbitrary string tag that lets you filter results or preload information about their targets.
- An anchor is an entry whose address is easy to calculate because its value is easy to discover, such as a username or hard-coded app constant.
- A path is like an anchor, but lets you specify hierarchical trees.
- Anchors and paths risk creating 'hot spots' on the DHT; this can be mitigated by sharding them into many sub-anchors and sub-paths.

Learn more

- Wikipedia: Graph database
- Wikipedia: Resource Description Framework, a standard for linking semantic data on the web

- Wikipedia: Linked data, an application of linking to the web

06. CRUD Actions: Modifying and Deleting Data

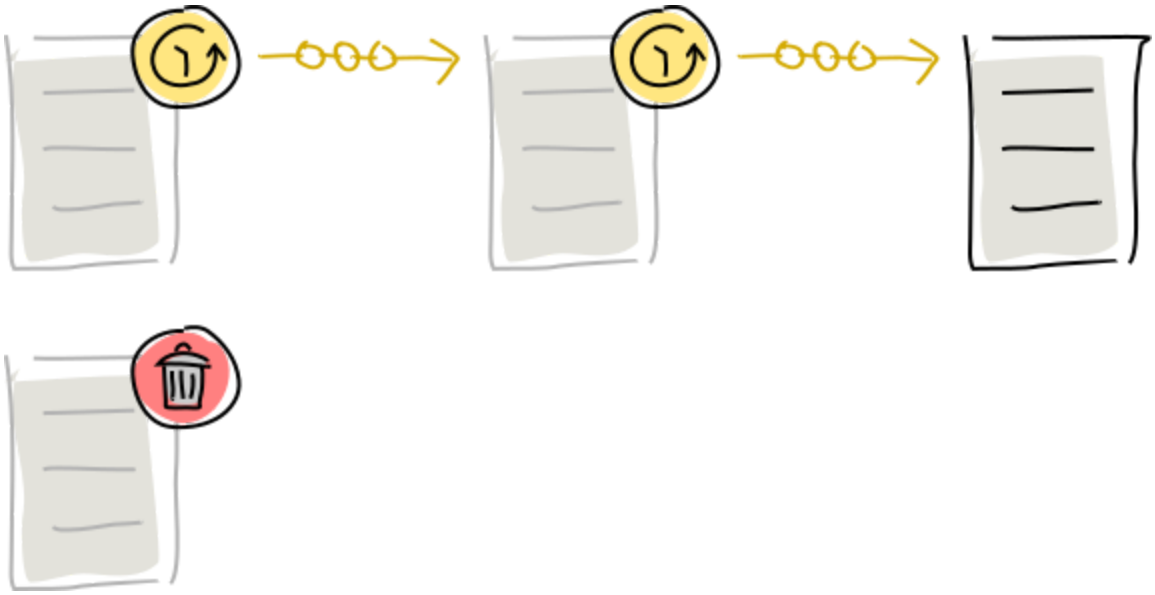
Holochain allows agents to mutate immutable data by publishing special **remove** and **update** actions to the DHT.

What you'll learn

1. Why you can't delete or modify DHT data in the usual way
2. How to simulate mutability in an immutable database
3. Addressing concerns about privacy and storage requirements

Why it matters

Immutable public data is a surprising feature of Holochain and many other distributed systems. It's important to understand the consequences in order to make informed design decisions that respect your users' privacy and storage space.



Public, immutable databases

Data in a Holochain app is immutable for a couple reasons:

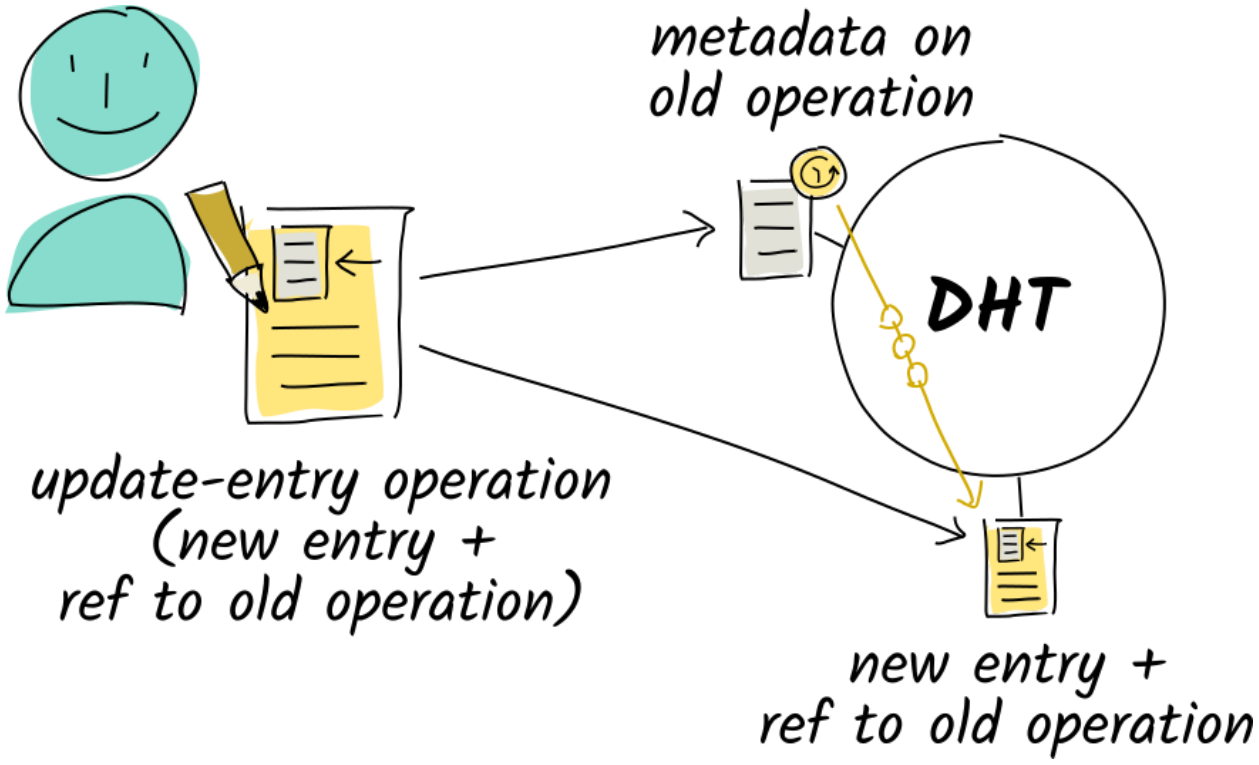
- Immutability of the source chain and DHT data means we can safely assume that data wasn't changed after being published.
- Immutability makes data syncing faster, simpler, and more reliable. Mutable distributed databases often require complex coordination protocols that reduce performance.

However, developers expect CRUD (create, read, update, delete) to be a basic feature of a database; it's an important part of most apps. So how do we do it on Holochain?

Simulating mutability

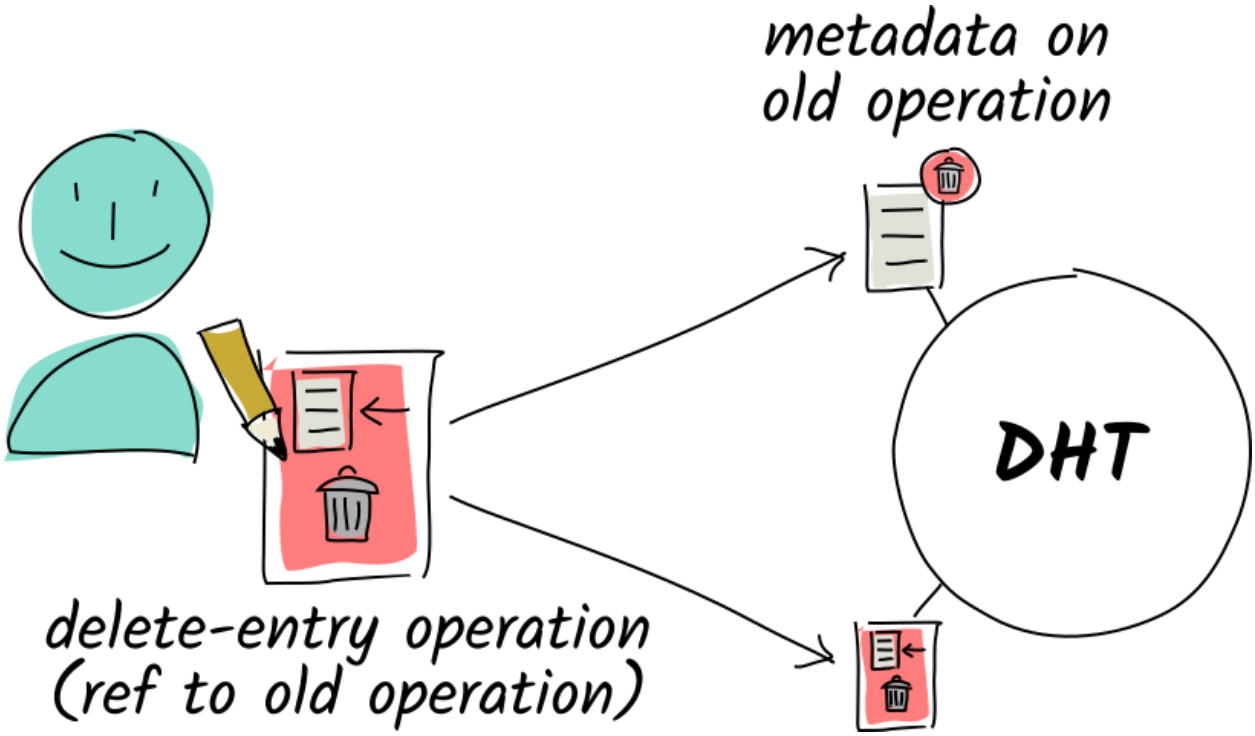
You might remember from a few pages back that we described each element as an 'action', not a thing. When you write an entry to your source chain, you're actually recording the *act of speaking it into existence*; we call it a **create-entry action**. (This is called event sourcing, if you're interested.)

Holochain also lets you speak the *act* of deleting something or replacing it with something new. The old data is left in place, new data appears, and a piece of metadata is added onto the old data to indicate its new status. Holochain features a few special action types to do this for you:



- An **update-entry action** is a type of new-entry action just like a create-entry action, but it references another new-entry action and instructs its DHT authorities to point to the new

one as a replacement. Multiple update-entry actions can be applied to any type of new-entry action.



- A **delete-entry action** instructs DHT authorities to mark an existing new-entry action as ‘dead’. If all of an entry’s new-entry actions are marked as dead, the entry itself is marked as dead too.
- A **create-link action** and a **delete-link action** do the same for links. (Note that there is no update-link action.)

You might notice that these modification actions apply to existing *new-entry actions*, not *entries.* All the DHT does is accumulate all these actions and present them to the application. This gives you some versatility in deciding how to manage multiple potentially conflicting contributions from many agents. It also prevents clashes between identical entries written by different authors at different times, such as Alice and Bob both writing the message “hello”. That entry exists in one place in the DHT, but it will have two new-entry actions attached to it, each of which can be updated or deleted independently.

You can ask the DHT for data in two different ways and get different results depending on the type of address you’re asking for:

	entry	**New-entry header**
get	Returns an **element** (entry/header pair) containing the entry data and the oldest live new-entry header that	Returns an **element** containing the entry data and the exact create or update header at this address.

	spoke this entry into existence. Returns an empty result If all new-entry headers have been marked as deleted.	Returns an empty result if this header has been marked as deleted.
get details	Returns the entry data along with all new-entry headers that spoke it into existence, any deletes on those headers, and any updates that point from this entry's new-entry headers or updates to new entries.	Returns an element along with all the deletes and updates on it.

The future: real deletion

In the future we intend to introduce two new actions:

- **Withdraw-header actions** will allow an agent to ask DHT authorities to forget an action they accidentally published.
- **Purge-entry actions** will allow anyone to ask DHT authorities to remove an entry regardless of who published it. This can be removed to scrub illegal or dangerous content, or comply with right-to-forget laws.

Handling privacy concerns and storage constraints

You've seen how deletes and updates don't actually remove data; they just add a piece of metadata that changes its status. Even with the future withdraw and purge actions, all they are is a polite request for other peers to remove data from their stores. This can break users' expectations. When you ask a central service to delete information you'd rather people not know, you're trusting the service to wipe it out completely—or at least stop displaying it. When your data is shared publicly, however, you can't control what other people do with it.

In a sense, this is true of anything you put on the internet. Even when a central database permanently deletes information, it can live on in caches, backups, screenshots, public archives, reading-list apps, and people's memories. Holochain just makes it easier to share and persist data. Privacy is all about creating friction around the sharing of data, so your responsibility as a designer is to create appropriate levels of friction. Here are some guidelines:

- Be careful about what the DNA returns to the UI. Your zome functions serve as the API to the DHT's data and can avoid retrieving deprecated data. This doesn't completely prevent people from accessing it, but it does make it harder.
- Design your UI to communicate the permanence of the information users publish so they can make responsible decisions.

Because data takes up space even when it's no longer live, be judicious about what you commit to the source chain and the DHT.

- For large objects that have a short life, consider storing data outside of the DHT in separate, short-lived DHTs, IPFS, Dat, a data store on the user's machine outside of Holochain, or even a centralized service.
- Commit large sets of small updates to entries in batches, or ignore the built-in update function and commit deltas instead.
- For large entries with frequent but small changes, break the content into 'chunks' that align with natural content boundaries such as paragraphs in text, regions in images, or scenes in videos.

Key takeaways

- All data in the source chain and DHT is immutable once it's written. Nothing is ever deleted.
- It's useful to be able to modify data, so Holochain offers 'delete' and 'update' functions for public data that mark old entries obsolete.
- This requires extra care to meet users' expectations because no data is ever deleted.
- In the future we intend to introduce actions that request the actual deletion of data.

07. Validation: Assuring Data Integrity

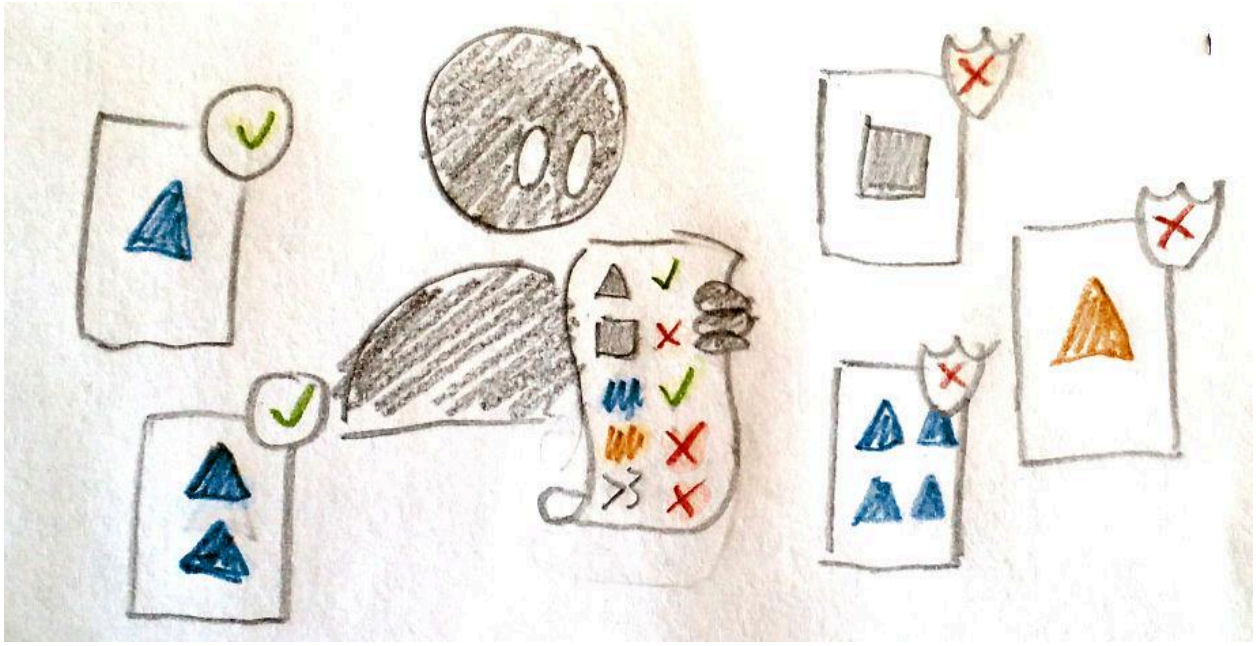
Holochain DNAs specify **validation rules** for every type of entry or link. This empowers agents to **check the integrity** of the data they see. When called upon to validate data, it allows them to identify corrupt peers and publish a **warrant** against them.

What you'll learn

1. Why validation matters
2. What happens when validation fails
3. What validation rules can be used for
4. How a validation rule is defined
5. What makes a good validation rule

Why it matters

Data validation rules are the core of a Holochain app. They deserve the bulk of your attention while you're writing your DNA.



Validation: the beating heart of Holochain

Let's review what we've covered so far.

1. Holochain is a framework for building apps that let peers directly share data without needing the protective oversight of a central server.
2. How does it do this? You may recall from the beginning of this series that Holochain's two pillars are **intrinsic data integrity** and **peer replication/validation**. The first pillar defines what valid data looks like, while the second pillar uses the rules of the first pillar to protect users and the whole network.
3. We said that each type of **app entry** can contain any sort of binary data whose correctness is determined by **validation rules** written into the application.
4. We then showed how **peer validators** use those rules to analyze entries and spread news about bad actors.

Holochain is the engine that allows peers to move data around, validate it, and take action based on validation results. **Your DNA is simply a collection of functions for creating data and validation rules for checking that data.** Those two things are critical to the success of your app because they define the membranes of safety between the user and the stuff she receives from others via the DHT. An agent's running DNA prevents them from creating invalid

data and protects them from accepting other people's invalid data. Well-designed validation rules protect everyone, while buggy validation rules leave them vulnerable.

Some entries can be computationally expensive to validate. In a currency app, for example, the validity of a transaction depends on the account balances of both transacting parties, which is the sum of all their prior transactions. The validity of each of those transactions depends on the account balance at the time, plus the validity of the account balance of the people they transacted with, and so on and so on. The data is deeply interconnected; you don't want the coffee shop to interrogate your entire transaction history and the histories of all your counterparties when you're trying to buy a cup of coffee and get to work.

The DHT offers a shortcut—it remembers the validation results of existing entries. You can ask the validators of the parties' previous transactions if they detected any problems. You can assume that they have done the same thing for the transaction prior to those and so on. As long as you trust a significant portion of your peers to be following the same rules as you, the validation result of the most recent entry 'proves' the validity of all the entries before it.

Validation flow: success and failure

A validation rule operates on an **element**, which you will remember is the *act of speaking something into existence.* It's used in two different scenarios, each with different consequences:

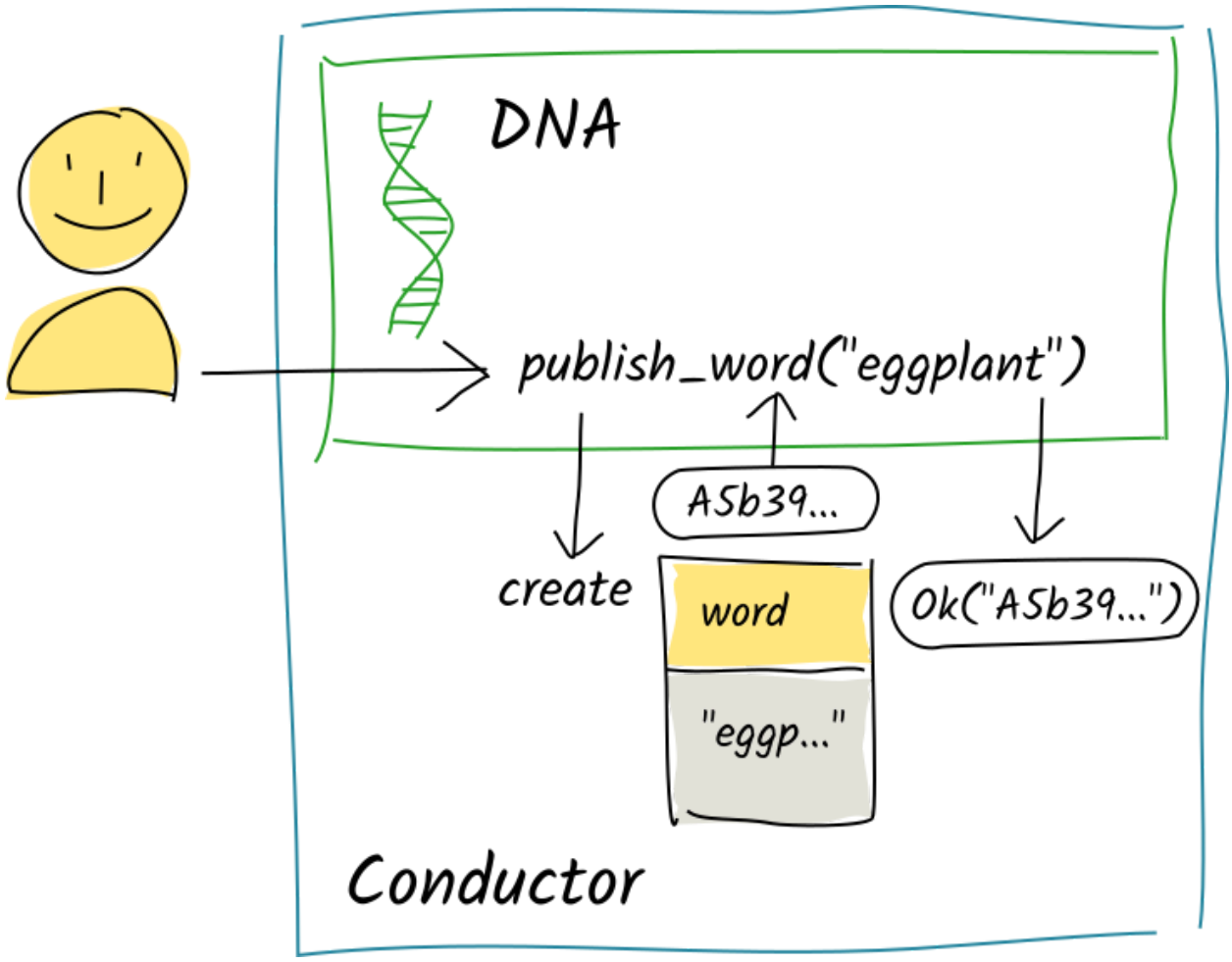
- When an agent first authors an element,
- When an authority receives an element for validation.

We'll carry on with the DHT illustrations from chapter 4, but let's add a simple validation rule: the "word" entry type has a validation rule that says that it can only contain one word.

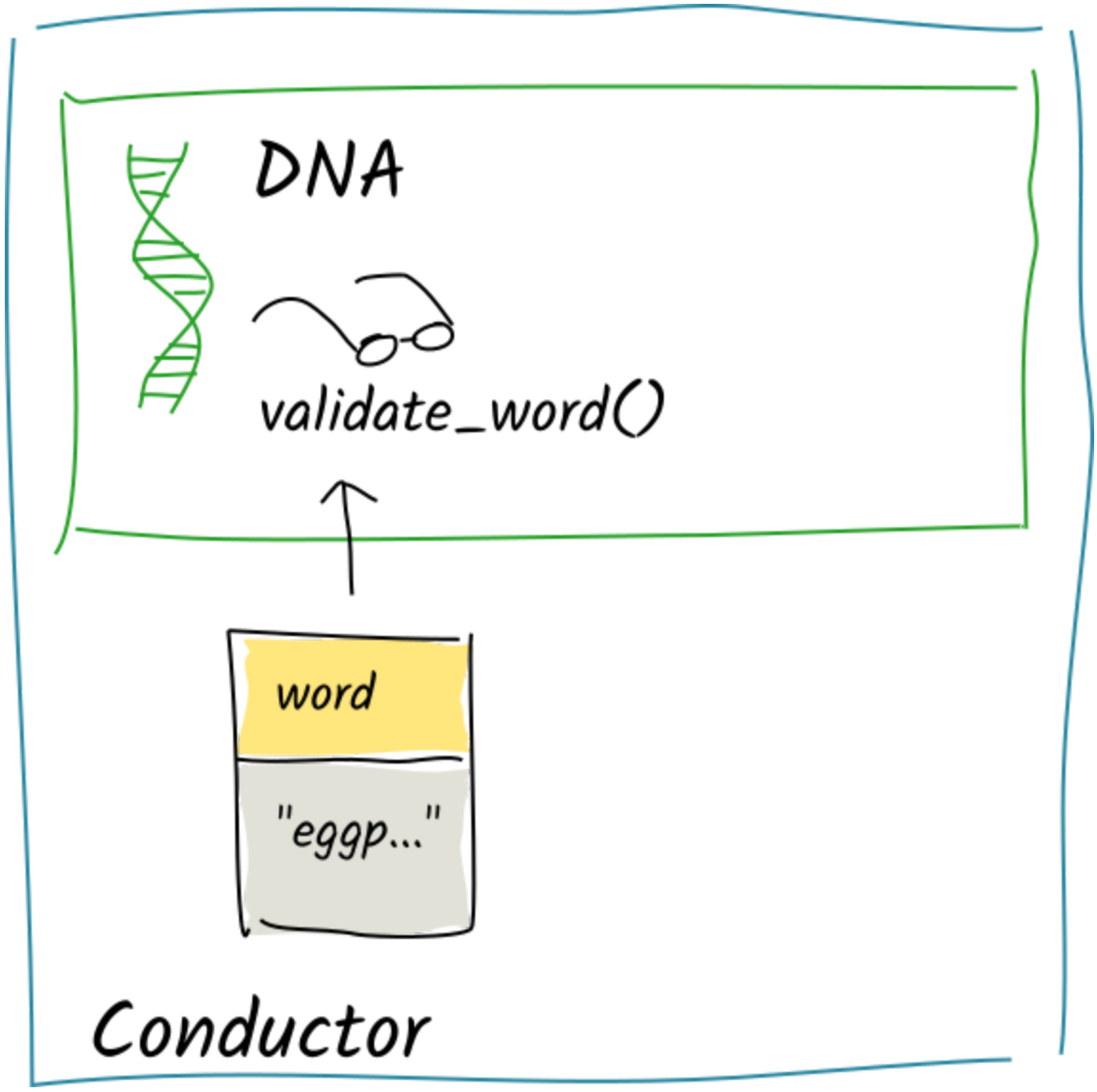
Authoring

When you **commit an entry**, your Holochain conductor is responsible for making sure you're playing by the rules. This protects you from publishing any invalid data that could make you look like a bad actor.

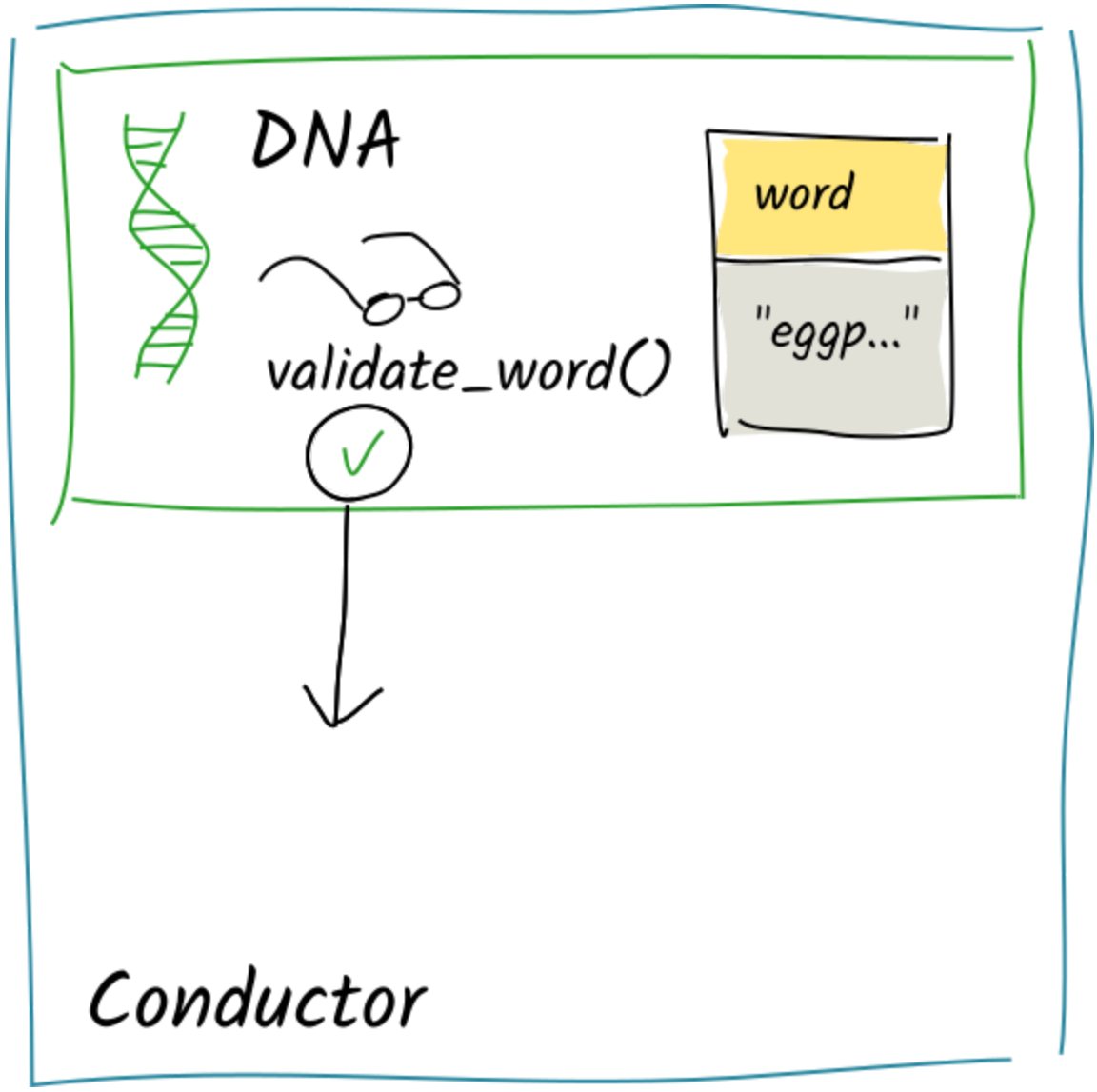
Valid entry



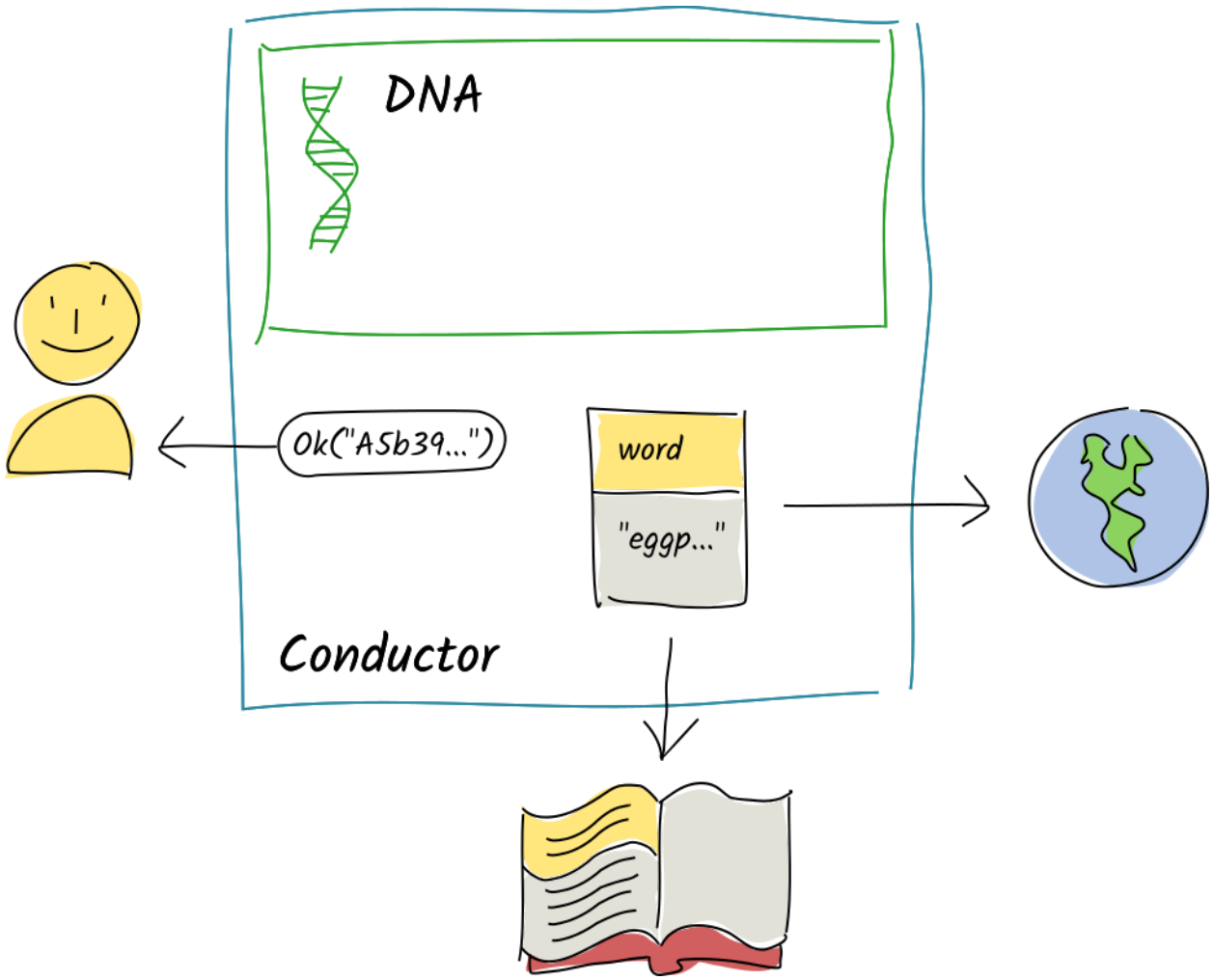
1. Alice calls the “publish_word” zome function with the string "eggplant". The function calls the “create_entry” host function to commit that word to her source chain. The conductor ‘stages’ the commit in the function’s scratch space and returns the creation action’s element hash to the “publish_word” function. The function continues executing and passes a return value back to the conductor, which holds onto it for now.

2. 

After the function has finished, Alice's conductor takes this element and calls the DNA's validation function for the "word" entry type.

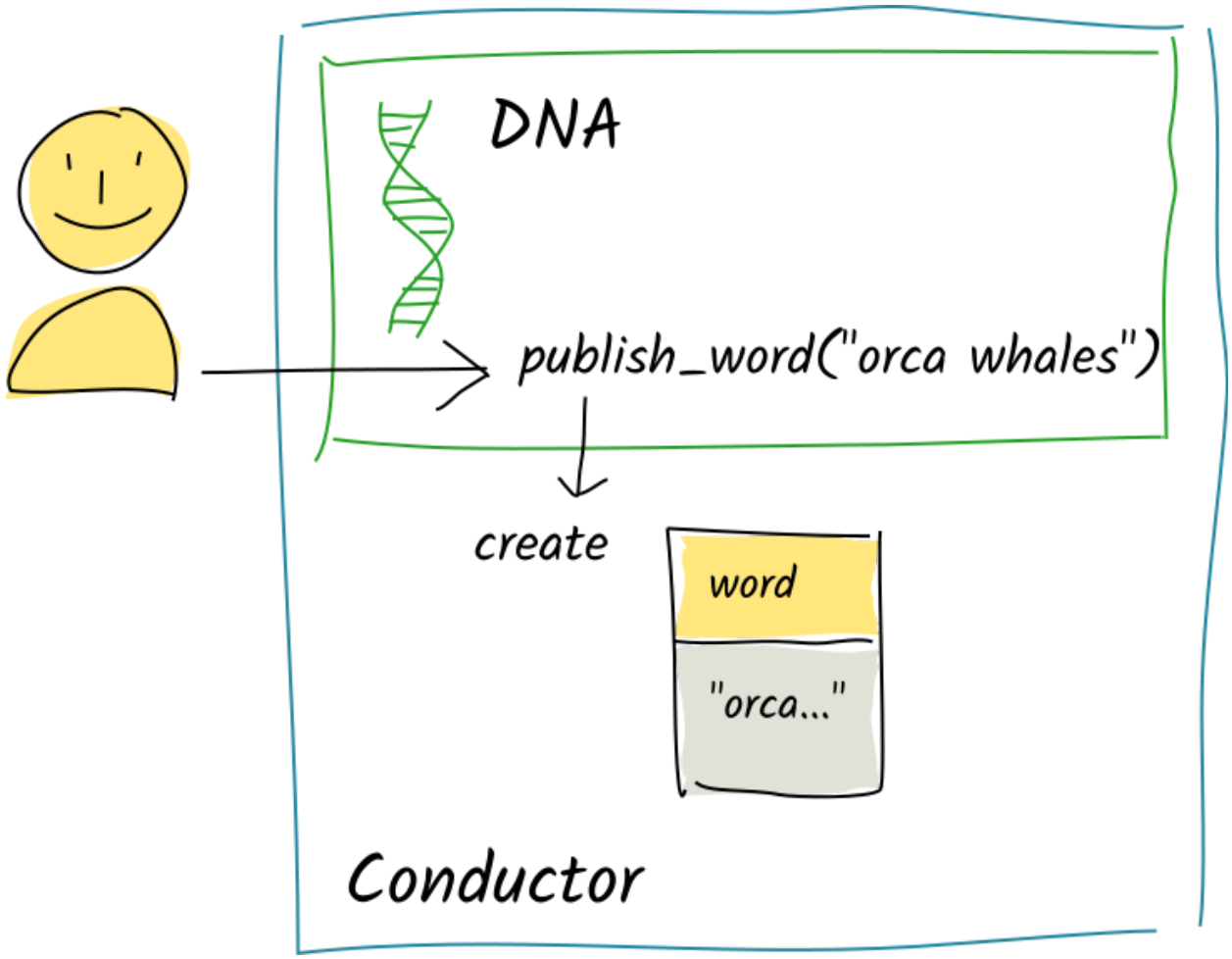
3. 

The validation function sees only one word, so the entry is valid. It returns “Valid”.

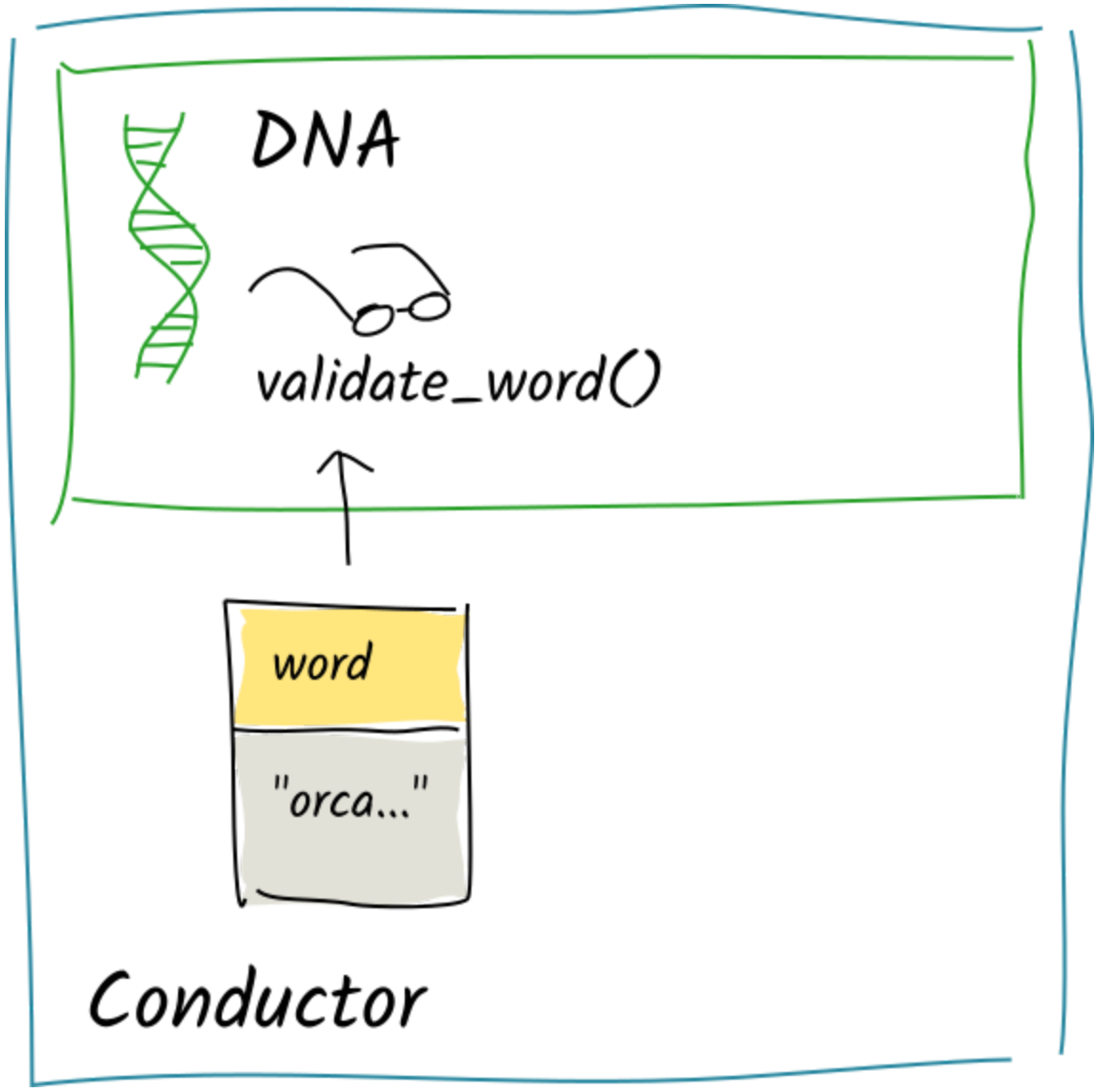


4. Her conductor commits the entry to her source chain, clears out the scratch space, and passes the “publish_word” function’s return value back to the client.

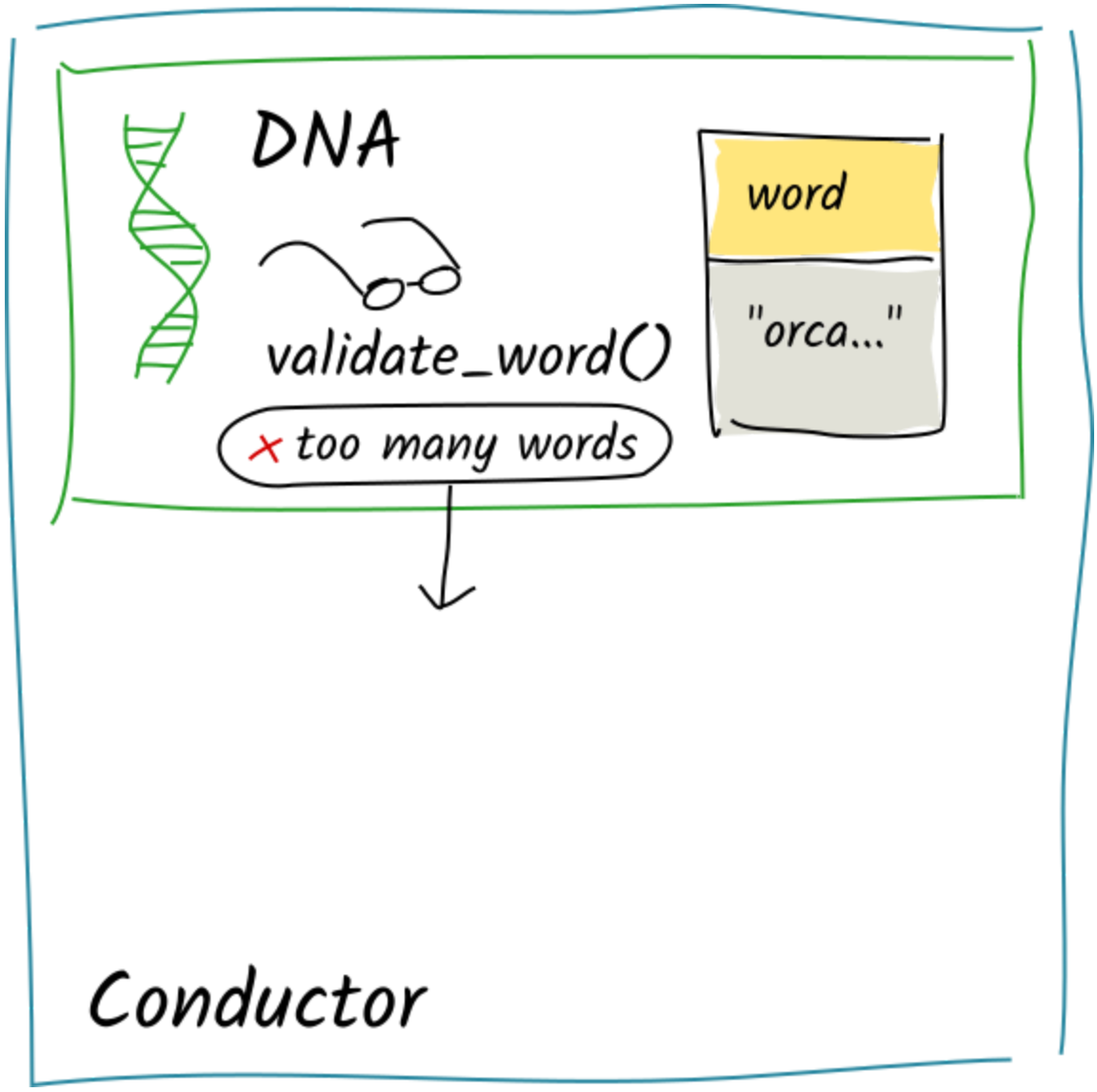
Invalid entry

1. 

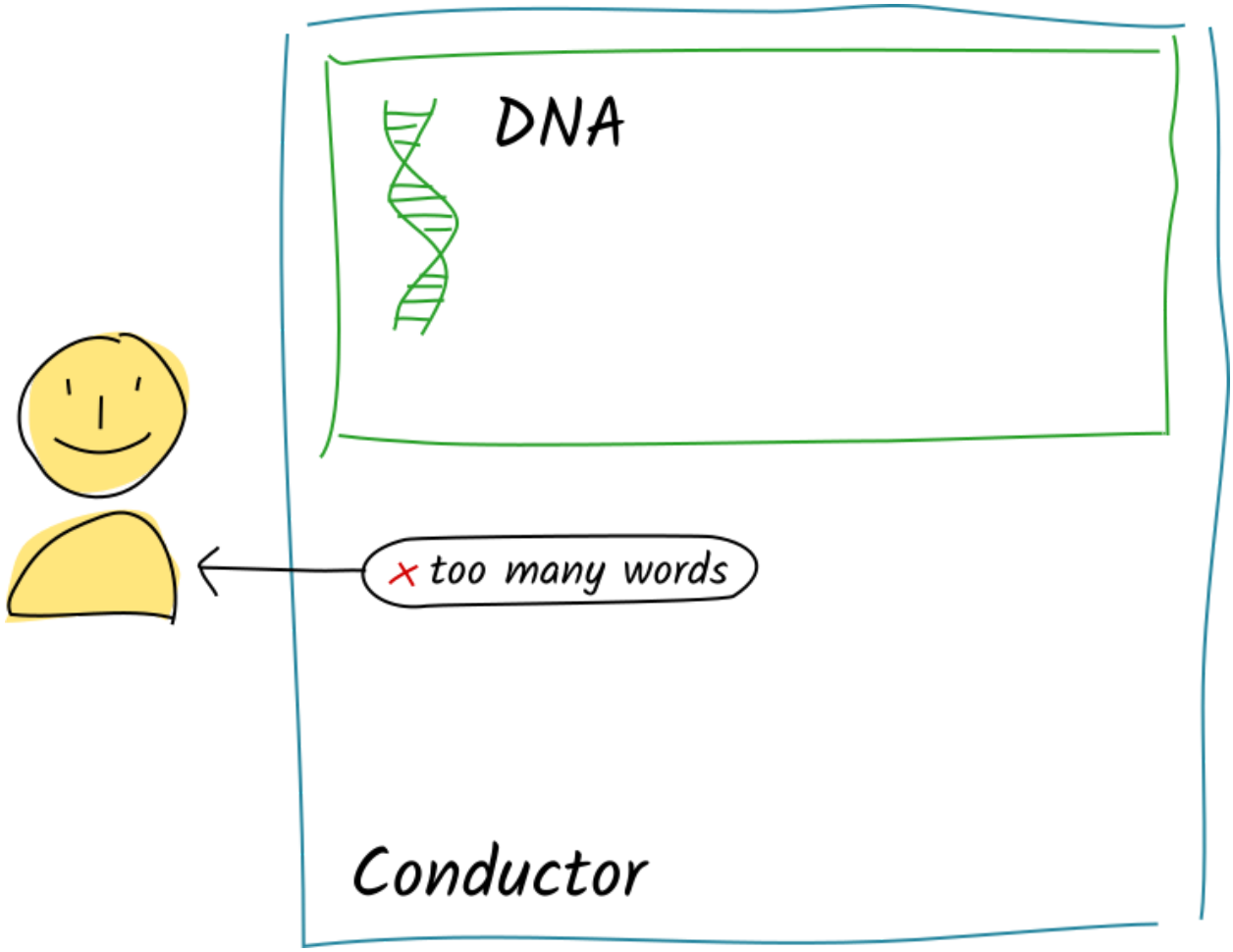
   Alice calls the same zome function with the string "orca whales". Again, the function calls “create_entry” and the commit is staged to the scratch space.

2. 

   Again, the conductor calls the validation function for the “word” entry type.

3. 

This time, the validation function sees two words. It returns Invalid("too many words").



4. Instead of committing the entry, the conductor passes this error message back to the client instead of whatever the “publish_word” function’s return value was.

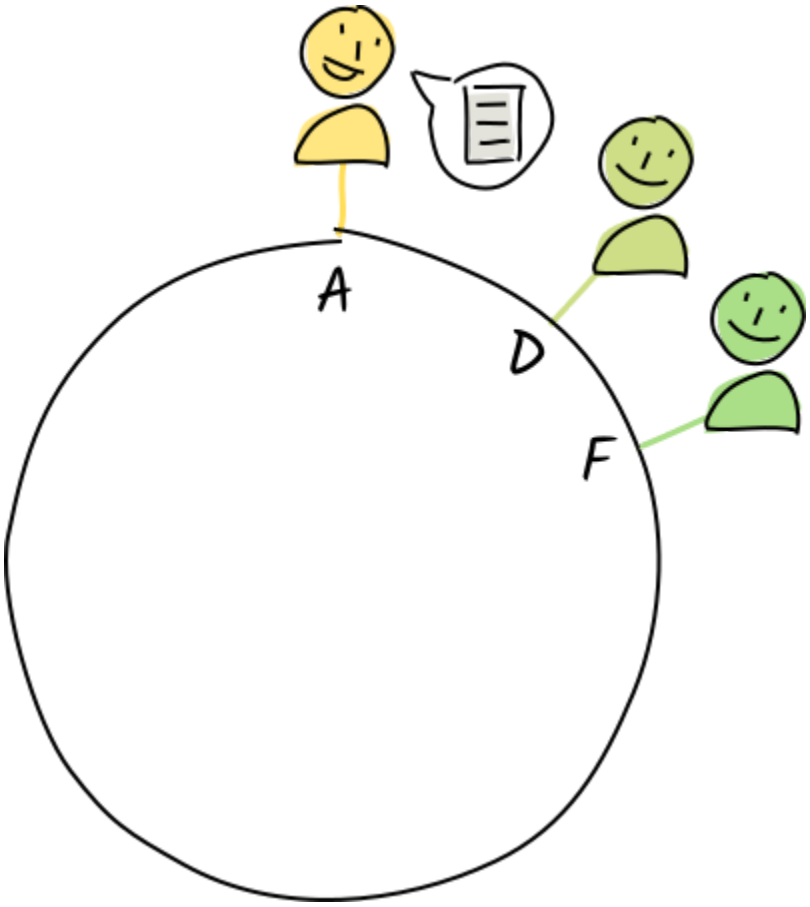
You can see that author-side validation is similar to how data validation works in a traditional client/server app: if something is wrong, the business logic rejects it and asks the user to fix their data.

Peer validation

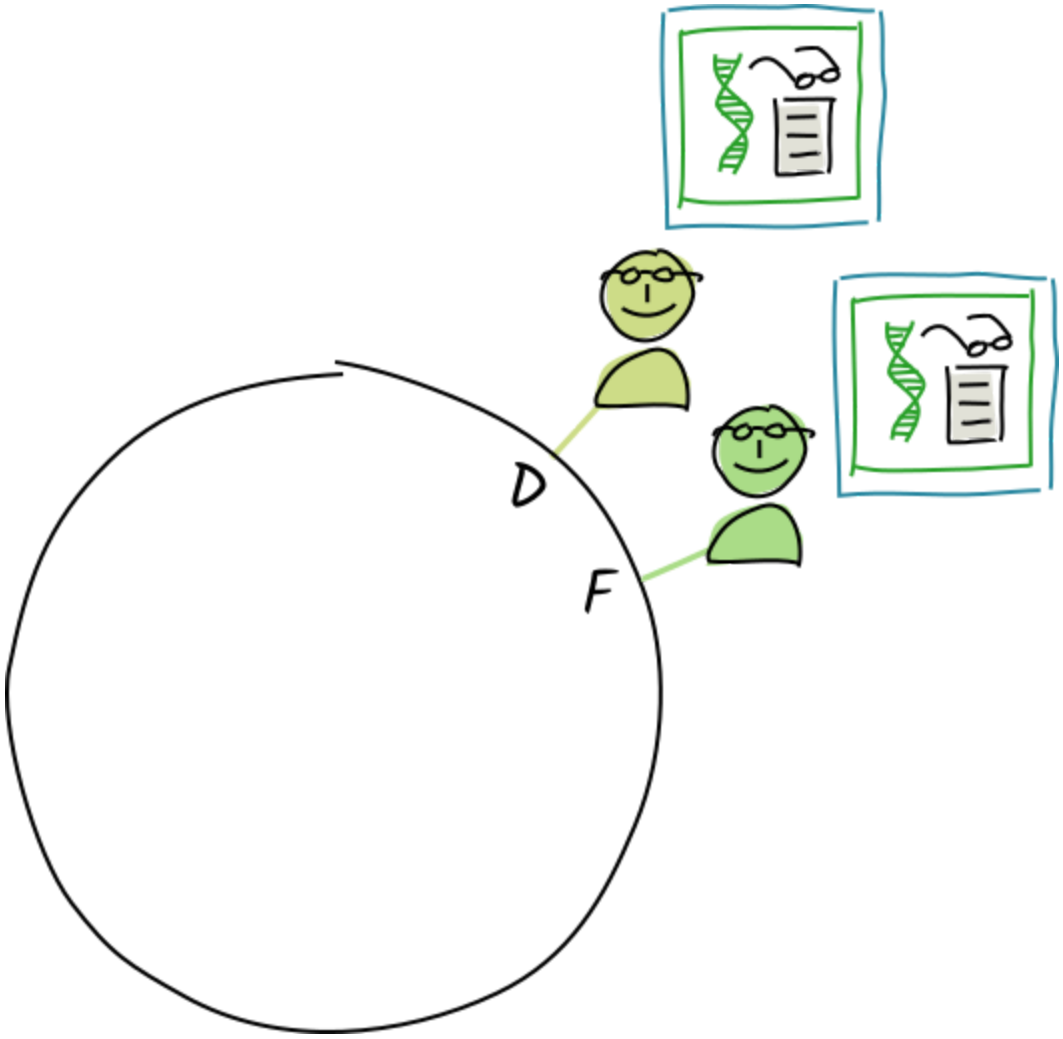
When an authority **receives an entry for validation**, the flow is very different. The authority doesn’t just assume that the author has already validated the data; they could easily have hacked their conductor to bypass validation rules. It’s the authority’s duty and right to treat every piece of data as suspect until they can personally verify it. Fortunately, they have their own copy of the validation rules.

Here are the two scenarios above from the perspective of the validators.

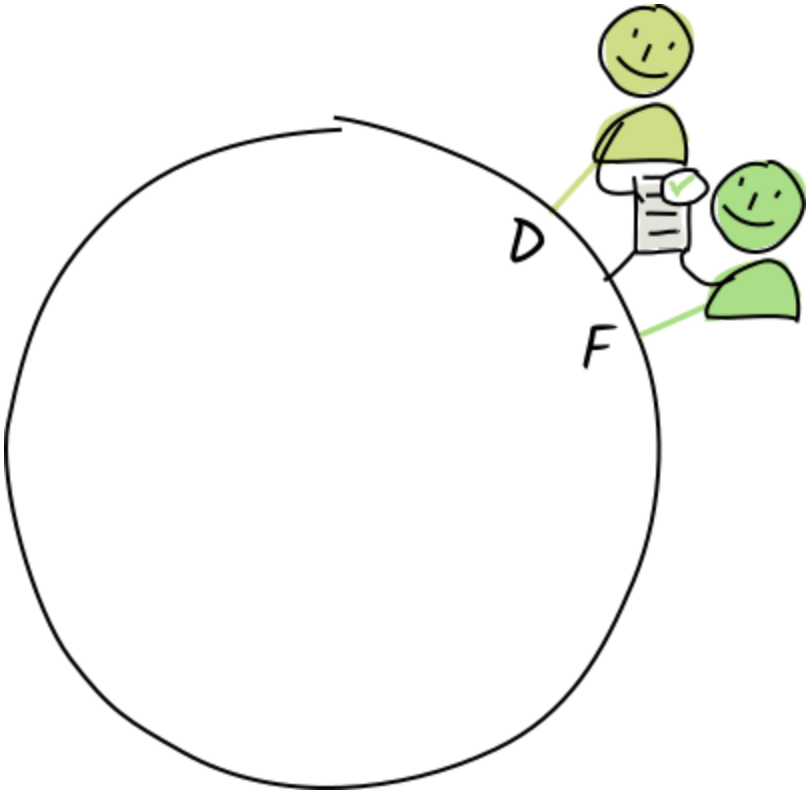
Valid entry



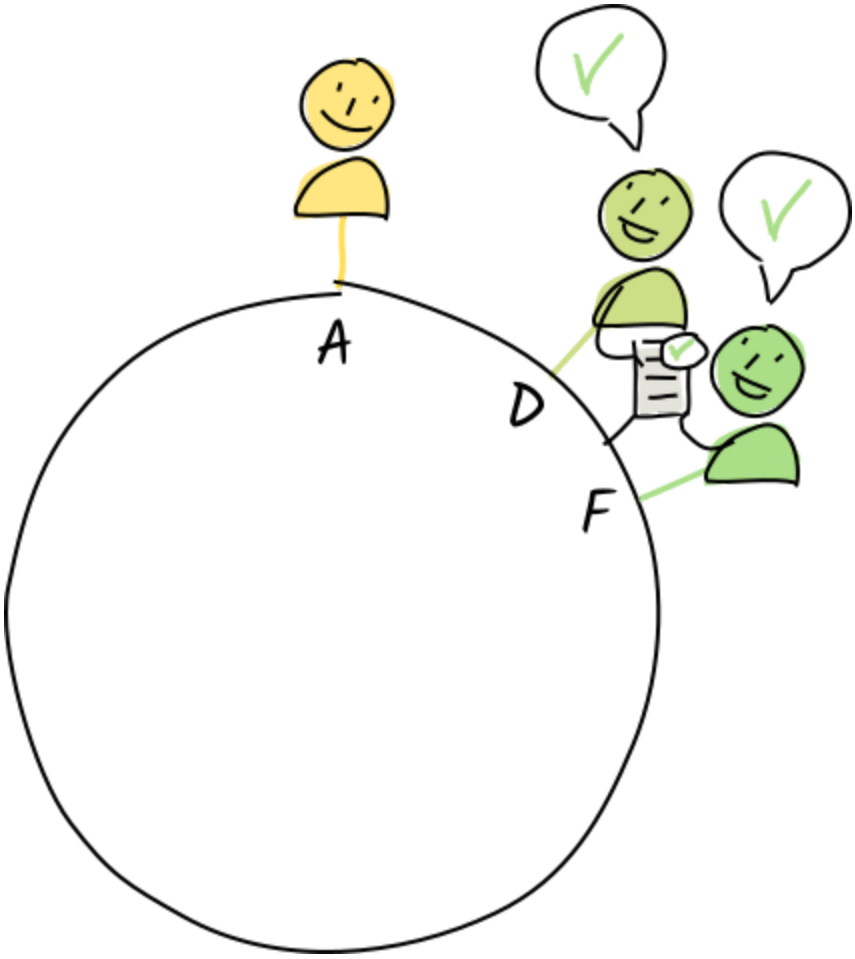
1. As authorities for the address “E”, Diana and Fred receive a copy of Alice’s "eggplant" entry for validation and storage.



2. Their conductors call the “word” entry type’s validation function.



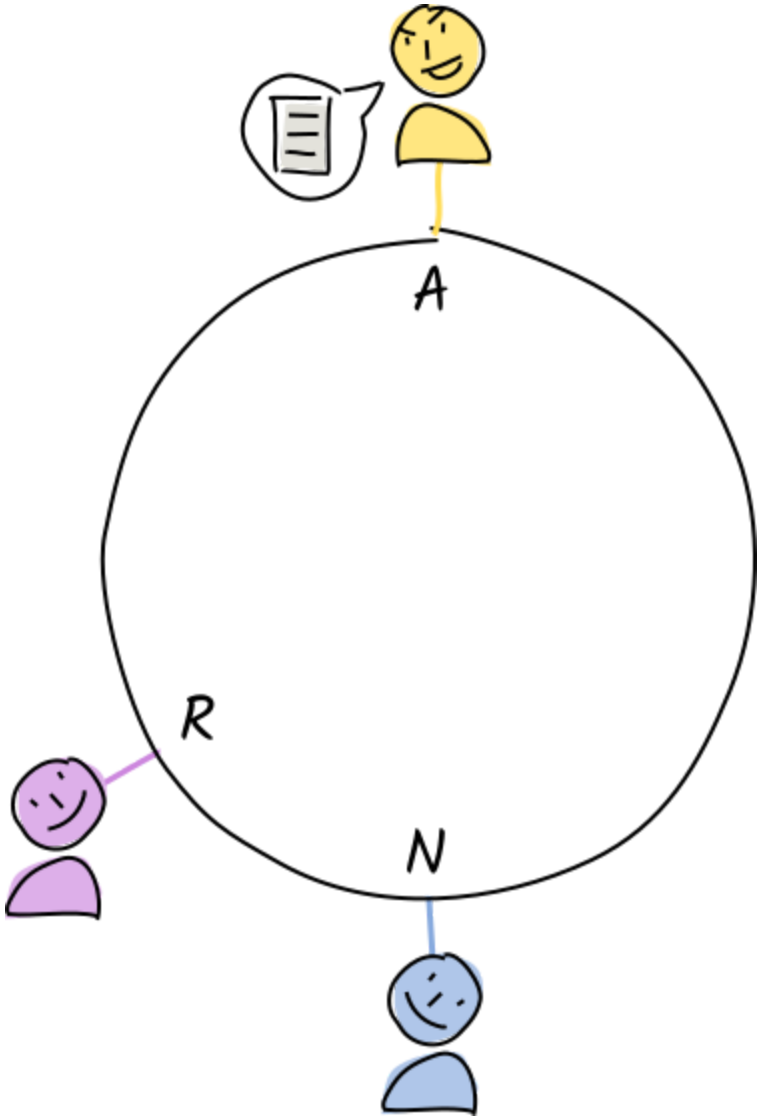
3. The entry is valid, so they store it in their personal shard of the DHT, along with their **validation receipts** attesting its validity.

4. 

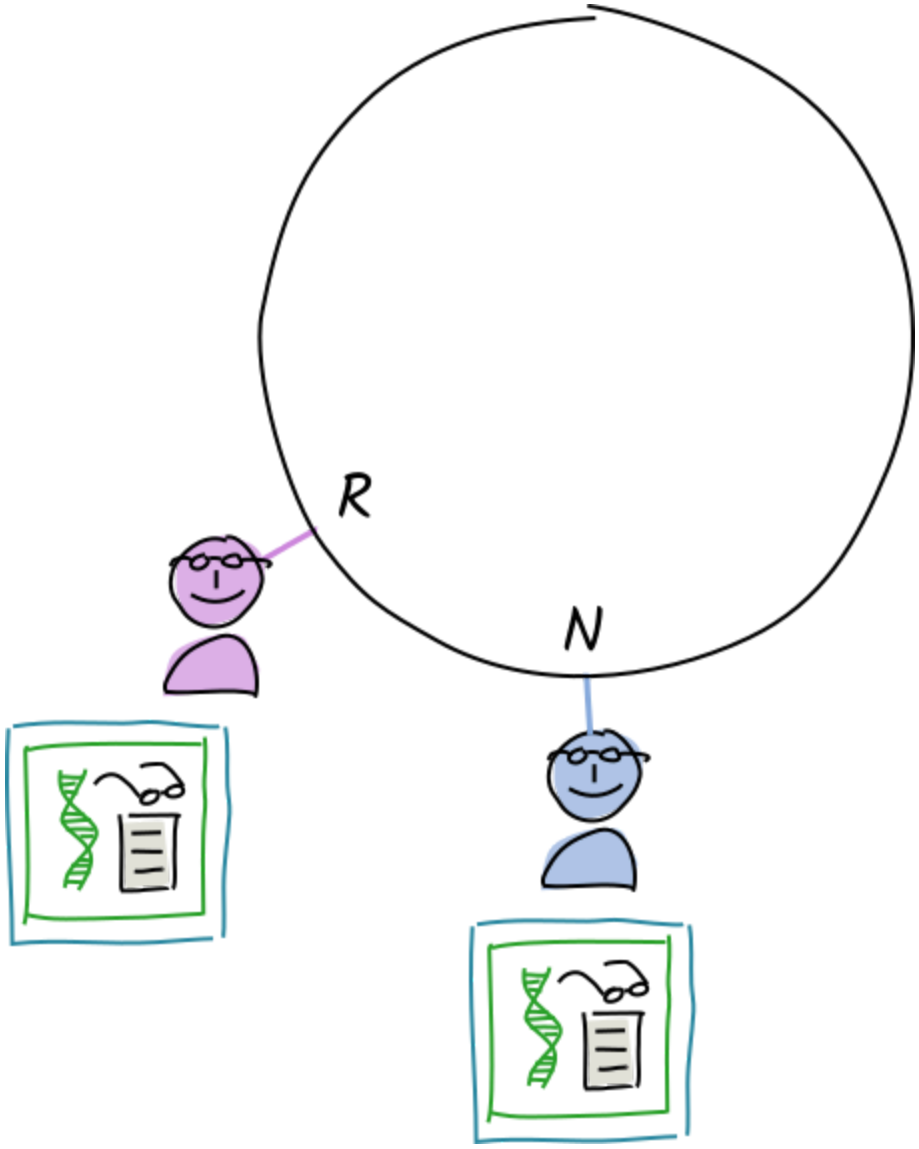
They both send a copy of their receipts back to Alice.

Invalid entry

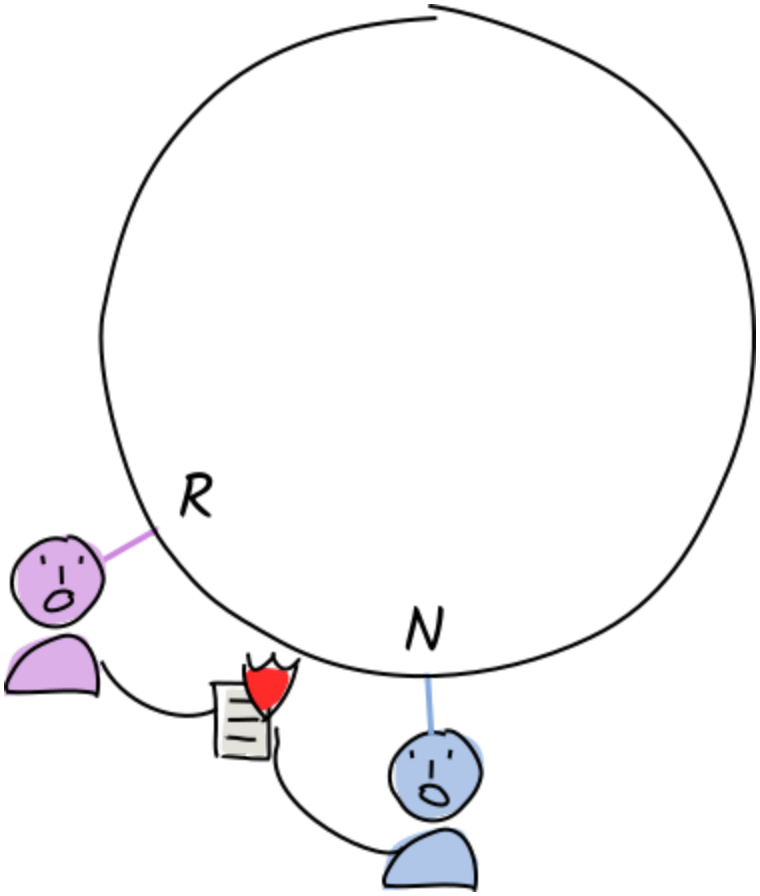
Let’s say Alice has taken off her guard rails—she’s hacked her Holochain software to bypass the validation rules.



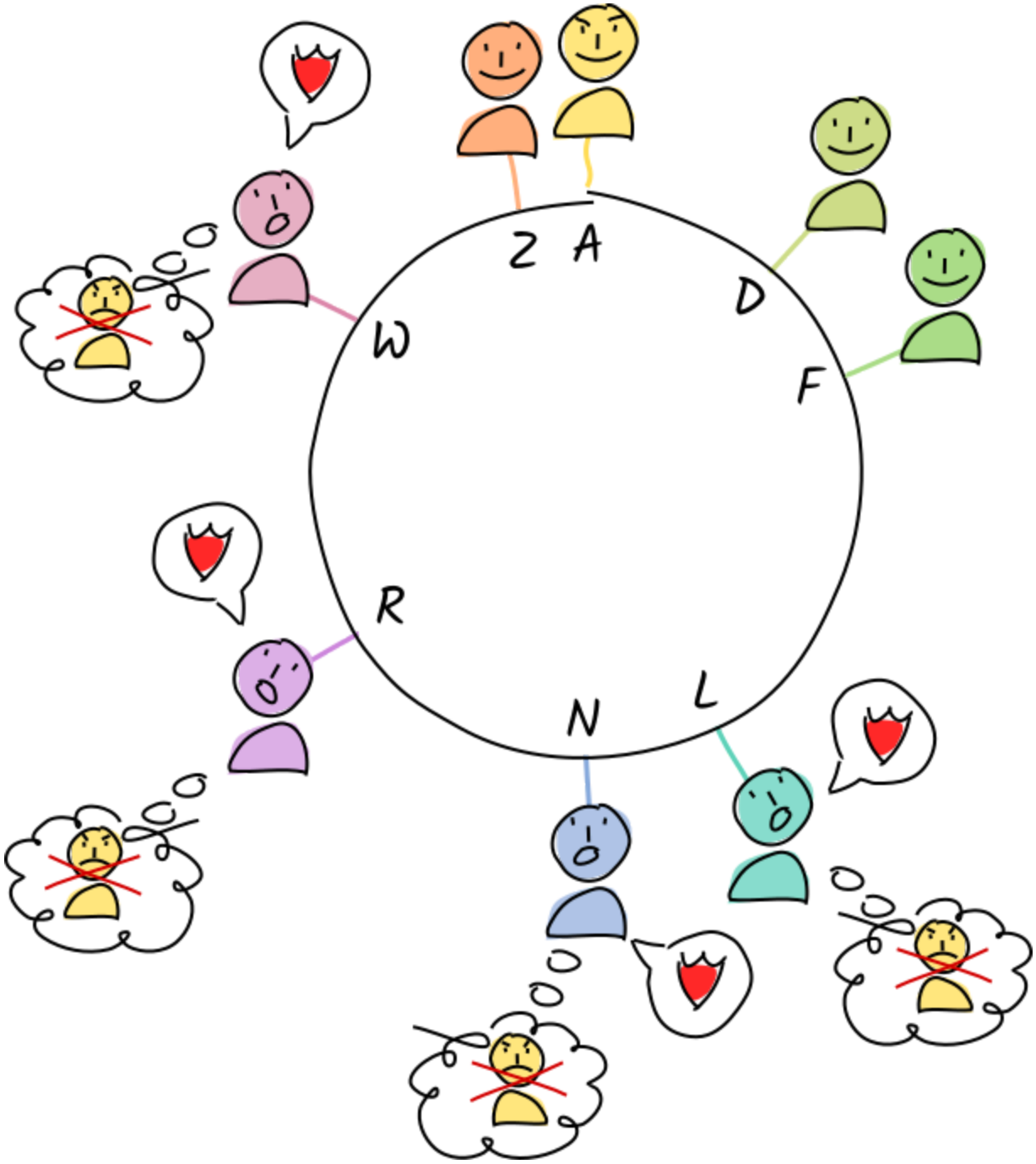
1. Norman and Rosie receive a copy of Alice’s "orca whales" entry.
2.



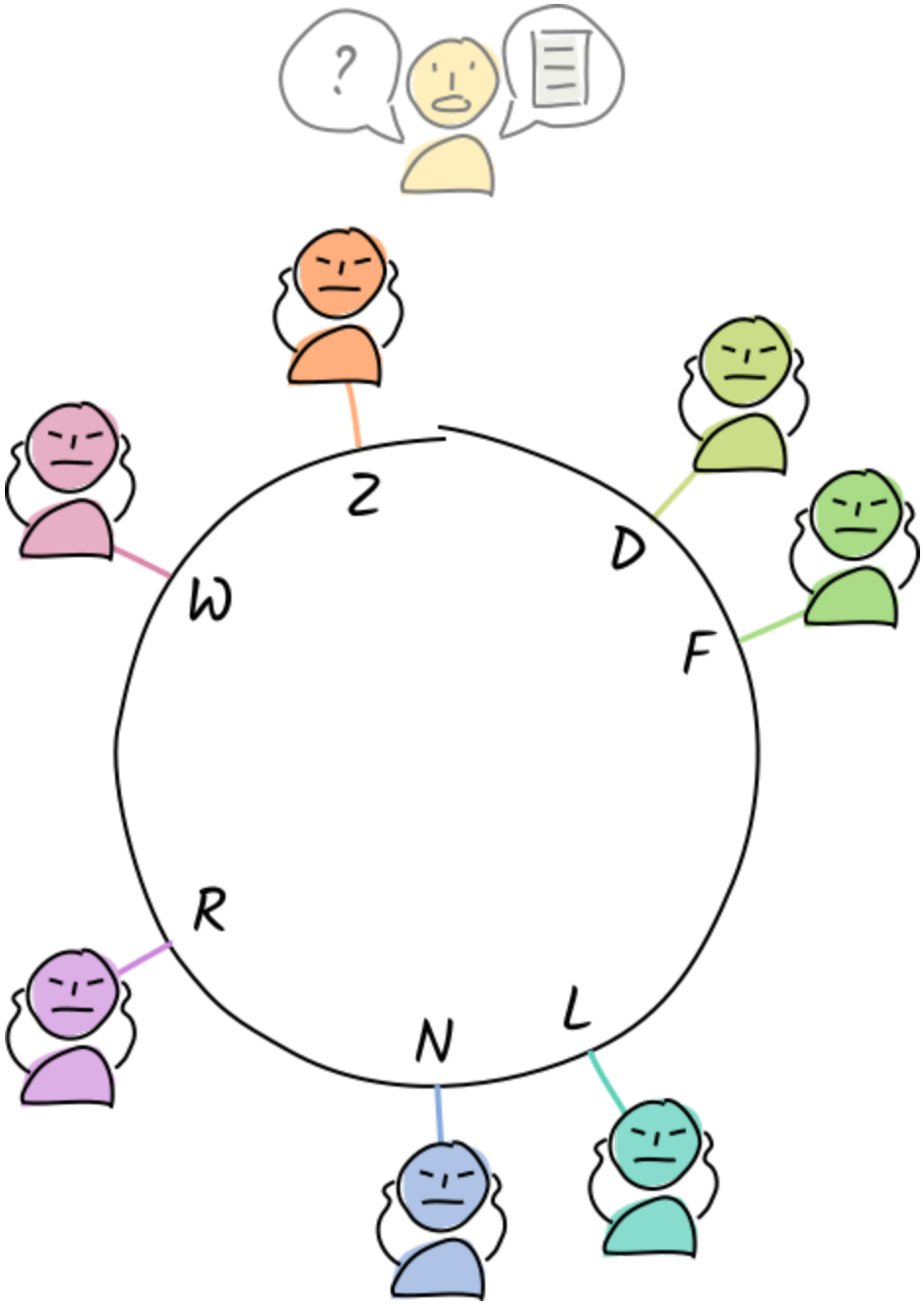
3. Their conductors call the validation function.

4. 

The entry is invalid. They create, sign, and store a **warrant** that contains the evidence (the invalid entry) along with their signatures.



5. Norman and Rosie share the warrant with Alice's agent ID authorities — that is, her DHT neighbors. Now anyone who wants to check up on her honesty can contact those authorities, ask for warrants, and choose to refuse contact with her.



6. Eventually, everyone knows that Alice is a ‘bad actor’ who has hacked her app. They all ignore her whenever she tries to talk to them, which effectively ejects her from the DHT.

Use cases for validation

The purpose of validation is to **empower a group of individuals to hold one another accountable to a shared set of rules**. This is a pretty abstract claim, so let's break it down into a few categories. With validation rules, you can define:

- **Access membranes**—validation functions for the **membrane proof entry** govern who's allowed to join a DNA's network and see its data.
- **The shape of valid data**—validation rules on **entry and link types that hold data** can check for properly structured data, upper/lower bounds on numbers, string lengths, non-empty fields, or correctly formatted content.
- **Rules of the game**—validation rules on **connected graph data** can make sure chess moves, transactions, property transfers, and votes are legitimate.
- **Privileges**—validation rules on **create, update, or remove actions** on entries and links can define who gets to do what.

How validation rules are defined

A validation rule is simply a callback function in your zome code that takes an element and the agent's context in which it was written, analyzes it, and returns a result. The context is called a **validation package**, and it consists of any prior source chain data necessary to validate the element. The required validation package contents are specified per entry type and can be one of:

- Nothing,
- **The full chain** up to the element,
- **All chain elements of the same type** up to the element,
- A **custom package** generated by a special function of your own creation.

Data can be retrieved from the DHT to support validation. Once it's done its work, the validation function can return one of three values:

- **Valid**,
- **Invalid**, with a description of what was invalid,
- **Unresolved dependencies**, with a list of the addresses of data it couldn't retrieve from the DHT.

All actions (create, update, delete) on all entry types (app entries, agent ID entries, and capability grants/claims) can have different validation functions, which follow a 'cascade' of specificity. Holochain calls all the matching validation functions it can find for an entry type, starting with the most specific, until one of them returns an error. The cascade for entries looks like this:

1. `validate_<action>_entry_<entry_type>` for specific actions on specific app entry types (e.g., `validate_update_entry_word` will be called for all update actions for entries of type 'word')
2. `validate_<action>_entry` for specific actions on all app entry types produced by the current zome
3. `validate_<action>` for specific actions on all app entry types produced by the current zome, as well as any system entry type produced by any zome in the DNA
4. `validate` for all actions on all app entry types produced by the current zome, as well as any system entry type produced by any zome in the DNA

For instance, you could have two validation functions for the "word" entry type:

- `validate_create_entry_word` checks that the entry only contains one word.
- `validate_create_entry` checks general write permissions for all app entry types, including the "word" type.

Links also have their own validation functions, `validate_create_link` and `validate_delete_link`, which are called for any links produced by the current zome. Because links don't have a type, if you want to apply different validation rules to different types of link, your function will have to switch on different kinds of link based on the base, target, and tag.

Guidelines for writing validation rules

A validation function **returns a yes/no result for a given element**, meant to ~~be~~ determine whether an agent is playing by the rules. If the function fails, it should be clear evidence that they've tampered with their DNA or conductor software. This means these rules aren't appropriate for soft things, like codes of conduct, which usually require human insight.

Validation rules should be **deterministic** and **pure** so that the result for a given element **doesn't change** based on who validated it, when they validated it, or what information was available to them at validation time. This means that the validation function **shouldn't rely on host context** such as the random number generator or the system time. It also means that it **shouldn't rely on any DHT state that might change over time**, but only on the addresses of data that *does exist*.

This means that care is required when an element depends on DHT data for validation. For instance:

- Rather than have a validation function check the status of something that could be updated or deleted (such as a changing list of banned words, or a privilege that can be revoked), set regular expiry intervals on this changeable data and let the author of a dependent element explicitly reference the currently active one at publish time.

- Rather than having a validation function query the links on a base, the author of the dependent element should reference the necessary links at publish time.
- Rather than having a validation function query the author headers on a dependency (such as an entry that requires a majority of voters before it's active), the author of the dependent element should reference the author headers it sees at publish time.

The common pattern here is that **it's the author's job to collect and explicitly reference the dependencies** that unambiguously prove the dependent element is valid. The validation function can then try to retrieve those entries and make decisions based on their contents and validity.

Because the authorities that hold an element's dependencies have probably already validated them, the authority trying to validate the dependent element doesn't need to revalidate them once it receives them — it can trust that they're valid unless it receives a warrant along with any of them.

If an authority can't retrieve the data for an element's dependencies at validation time, it should return their addresses as a list of **unresolved dependencies**. This puts the element into a not-yet-valid state, and the conductor will keep on trying to retrieve those dependencies until it either has them all or gives up.

Key takeaways

- Validation rules are the most important part of a Holochain DNA, as they define the core domain logic that comprises the 'rules of the game'.
- Validation supports intrinsic data integrity, which upholds the security of a Holochain app.
- Validation rules can cover an agent's entry into an application, the shape and content of data, rules for interaction, and write permissions.
- The result of a validation function is a clear yes/no result. It proves whether the author has hacked their software to produce invalid entries.
- If the validity of an entry depends on existing entries, existing validation results on that data can speed up the process.
- An author validates their own entries before committing them to their source chain or publishing them to the DHT.
- All public entries on the DHT are subject to third-party validation before they're stored. This validation uses the same rules that the author used at commit time.
- Validation rules are functions that analyze the content of an element and return either a success, an error message, or a list of dependencies that must be retrieved before the element can be validated.
- When an entry is found to be invalid, the validator creates and shares a warrant, which proves that the author has broken the rules of the app.
- Agents can use warrants as grounds for blocking communication with a corrupt agent.

- Validation functions should be deterministic and pure; once an element is deemed valid it should always be considered valid.
- Validation functions operate on *elements*, not *entries*.
- If an element's validity depends on other DHT data, it should reference those dependencies by address. The validation function should then try to retrieve those dependencies and return their addresses if they can't be retrieved; Holochain will then retry validation once the dependencies are retrieved.

Learn more

- Wikipedia: Deterministic algorithm
- Wikipedia: Pure function
- Wikipedia: Total function

08. Calls and Capabilities: Communicate With Other Components And Peers

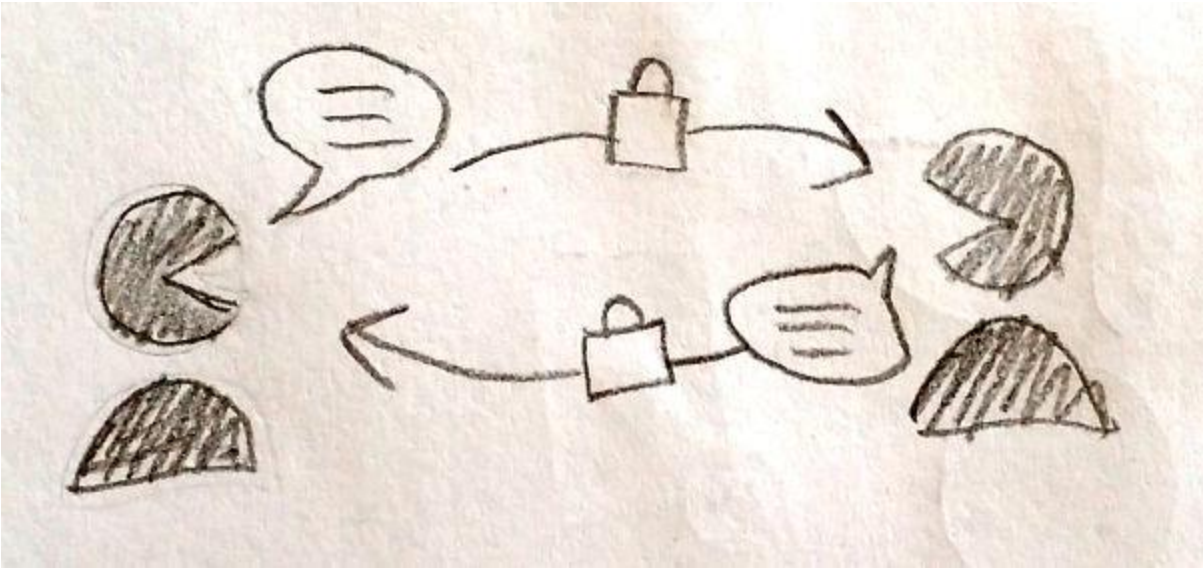
Application components can **call each other's zome functions**. On one agent's device, clients can call functions in cells, and cells in the same conductor can call each other's functions. Within one DHT, cells can call other agent's cells, allowing agents to delegate their agency to others.

What you'll learn

1. Who can call whose functions
2. How to secure functions against unauthorized use
3. How calls work

Why it matters

Web 2.0 flourished thanks in part to 'mashups', or publicly accessible APIs that allowed apps and services to use each other's data. Holochain enables a richer sharing of functionality and data between apps, anchoring the experience in the end-user's agency. This increases application development velocity and encourages the development of standard, shared functionality. Remote calls, on the other hand, allow agents in one app to interact privately without publishing any data to the DHT.



Client, inter-zome, bridge, and remote calls: who can call whom

There are four scenarios when a zome's functions might be accessed. In every scenario, they're just a remote procedure call. You might remember

Client call

Inter-zome call

You may remember from Application Architecture that zomes are separate libraries in one DNA. They don't have direct access to each other's functions, but they can still call them via the `call` host function.

Bridge call

A bridge call allows Alice's running cells on one machine to communicate with each other. This is useful for combining the functionality of multiple DNAs into one app. Because Holochain is centered around the agent, it makes more sense to say "Alice's app instances are talking to each other" than "app A is talking to app B".

Alice's UI can talk to any of her cells too, and it could certainly bear the responsibility for connecting them together. But it can't offer the same correctness guarantees that direct calls

between cells can offer: the source chains of each cell are ‘locked’ for the duration of the call, and the conductor provides the assurance that the code that cell A thinks cell B is running is actually what it’s running. This means you can better reason about the state of each instance, which is important for things like financial transactions.

Remote call

A remote call allows agents running the same DNA to call each other’s functions. When Bob’s cell makes a remote call to Alice’s cell, it’s Alice’s cell doing the work, which means that everything that happens — reads and writes, signals, and even calls to other cells — happens from her perspective. Essentially she’s delegating a bit of her agency to him.

Alice and Bob can use this to do all sorts of useful things:

- Alice can share off-the-record information with Bob, bypassing the DHT entirely. The connection between the two of them is end-to-end encrypted and doesn’t involve any hops through other peers.
- Bob can ask Alice for private data from her source chain.
- Bob can send Alice timely updates on information that matters to both of them, such as chat notifications, heartbeats, game moves, or edits on a shared document.
- Bob can also ask Alice for data from instances of other cells that she’s running and he isn’t — Alice acts as Bob’s bridge between the two DHTs.
- Alice and Bob can negotiate an agreement, such as a financial transaction or legal contract, because both of their source chains are ‘frozen’ at the moment of interaction.
- Alice can delegate certain tasks to Bob, such as publishing blog posts under her name (see the example below). She can even delegate agency that others have delegated to her.

How to secure functions against unauthorized use

At first sight, this seems pretty risky. Remote calls let someone exercise the full power of someone else’s agency, and bridge calls let one app access another’s functionality. And it would be risky — if there weren’t a security model in place.

Holochain uses a variation of capability-based security to protect an agent’s exposed zome functions. In this model, one agent is in complete control of a resource but can delegate control to another agent. Traditional capability-based security uses ‘capability tokens’, secrets that the resource controller generates and gives out to anyone they want to delegate their capabilities to. We’ve expanded that model a little bit:

- An **unrestricted** capability lets anybody call a function.

- A **transferable** capability means anybody who presents a valid capability token can call it (this is identical to traditional capability-based security).
- An **assigned** capability means that only agents with a valid capability token *and the right agent ID* can call a function.

In order for others to call one of their functions, the callee first has to grant access to that function. They do this by writing a **capability grant entry** to their source chain that specifies the function name, the access level, and any optional information needed to secure the grant (a random capability secret and/or a list of assignees). After that, Holochain will automatically check any incoming function call to make sure it matches an existing grant. When an agent wants to revoke or modify access, they simply delete or update that grant entry.

In order to use a transferable or assigned grant, a caller must have already received a capability secret, which they can then save to their own source chain as a **capability claim entry**. Any time they want to call a function, they retrieve this entry and pass the secret along with the function call.

There is one special case where capability tokens aren't needed: the implied **author** capability. If the agent ID of the caller and the callee match, such as with calls between zomes in a DNA or cells whose agent IDs are the same, no explicit capability grant is needed.

The lifecycle of a call

Here's an example of how remote calls might work in a blog app that allows people to publish under someone else's name.

Alice is the world's foremost authority on octopi who occasionally hires ghost writers to rewrite her articles in a more popular tone. Recently, she's hired Bob to write a few articles about octopus camouflage. They both use the same blog DNA, whose functions include:

1. "publish_post", which receives the text of a post, writes it to the user's source chain, and publishes it to the DHT
2. "receive_publish_post_permission", which takes a capability secret and a memo about why this permission is being granted, and records it as a capability claim on the source chain (this function is given an unrestricted capability at app install time, so anyone can call anyone else's)
3. "get_publish_post_permissions", which queries the source chain for all the capability claims that can be used on other agents' "publish_post" functions
4. "publish_as", which receives a post address and a capability claim address and remote-calls "publish_post"

First, Alice needs to let Bob publish posts under her name. Here's how she does it:

5. Alice grants Bob permission to call her “publish_post” function by committing an assigned capability grant to her source chain, consisting of Bob’s public key, a random secret, and the aforementioned function name.
6. Alice shares the secret with Bob by remote-calling his “receive_publish_post_permission” function along with the memo “write octopus camouflage articles for Alice”.
7. Bob’s conductor checks his source chain, finds the unrestricted grant, and runs the function. Inside the function, a capability claim entry is written to his chain consisting of the secret, Alice’s public key, the function name, and a **tag** consisting of the memo. This process doesn’t require any interaction from Bob, but just to be nice it sends a signal to his UI (we’ll talk about that later) to let him know Alice has granted him permission.

Bob has already been working on an article draft for Alice. Now that he’s received permission from Alice, it’s time to publish it. Bob clicks the ‘Publish’ button in his UI.

8. Bob’s UI calls his DNA’s “get_publish_post_permissions” function, then displays the ‘Publish As’ dialog box that shows the names of all the agents who’ve granted him permission to ghost-write for them, along with the memos that explained why he received those permissions.
9. Bob selects Alice’s and clicks ‘Publish’. It calls his DNA’s “publish_as” function with the post ID and capability claim address, which retrieves the post and the capability secret, then remote-calls Alice’s “publish_post” function.
10. Alice’s “publish_post” function returns a success message to Bob’s “publish_as” function, which returns the message to his UI.

Key takeaways

- Inter-zome calls let one zome call another zome’s functions within one agent’s DNA instance.
- Bridge calls let one of an agent’s DNA instances call another of their DNA instances, allowing sharing of functionality and data between apps.
- Remote calling lets one agent call functions in another agent’s DNA instance within the same DHT network.
- Remote calling sets up a direct, end-to-end encrypted channel between two agents to pass the input parameters and return value
- You call agents according to their agent IDs; Holochain resolves the ID to the agent’s IP address.
- Remote calling can be used for any data exchange that needs to be private, synchronous, temporary, or timely.
- Remote calling allows one agent to ‘delegate’ their agency to another.
- As with any function call, an inter-zome, bridge, or remote call blocks execution on the callerer’s side until they receive a response or the request times out.

- All calls are covered by capability-based security, which consists of a grant created by the callee, who shares a capability secret for the caller to use whenever they want to make a call.
- In Holochain's capability model, unrestricted and assigned capabilities allow for more or less permissivity than the traditional capability model.

9. Signals: Communicating without waiting for a response

As with any API, a DNA generally only receives function calls from the outside world and returns a response. But a DNA can also push **signals** to a listening UI on the agent's device, or another agent on the same DHT.

What you'll learn

1. When signals are useful
2. Local and remote signals
3. How reliable signals are

Why it matters

Signals avoid reliance on constant polling, making UIs much more responsive and performant.

When are signals useful?

Most connected applications are heavily interactive. You expect information from others as much as you create your own information. Chat messages appear, notification badges change, documents show real-time updates from collaborators, your opponent moves her knight, and so on. This liveness is the heartbeat of the modern app.

So far we've only described a DNA as just a collection of functions that you can call. And it is. You can make your app feel pretty responsive by having the UI regularly poll a function such as "get_messages_for_chat_channel", But this feels old-fashioned and can hurt performance.

Signals allow a running DNA instance to broadcast messages back to the UI. You would typically use this in actions that are *not* initiated by the UI, such as functions that are meant to be called remotely by other agents in the network, or by bridged DNA instances on the user's own machine.

Unlike a function call, a signal is just a message, so it doesn't expect a response. You can put whatever you like into that message — like most other payloads in Holochain, it's just bytes — and any listening component will receive it. From there they can open the

MessagePack-encoded envelope and interpret the payload any way they like. As with entry content and link tags, we recommend using MessagePack to serialize and deserialize signal payloads.

Local and remote signals

There are two kinds of signals. One goes to the user; the other goes to other agents on the DHT.

Local signals

When a UI or other client wants to keep up to date on something without initiating any action, they can listen for local signals on the same WebSocket connection they use to make zome and admin calls. The DNA can emit these signals as part of a zome function or other callback. For instance, in Calls and Capabilities Bob wants to ask Alice to give him permission to ghost-write on her behalf. Alice needs to know about his outstanding request, so the function that receives permission requests can emit a signal to her UI saying “please check this request and take action on it”.

Remote signals

Not every peer-to-peer interaction on the DHT needs a response. In cases where Bob doesn’t need to know whether his message was received, he can simply send a signal rather than making a remote call.

This is a lot faster too — if Bob needs to send a message to fifty people, with remote calls he’d have to make a call to each recipient, wait for a response or a timeout, and move on to the next. Remote signals, on the other hand, let you specify a number of recipients, send off the message, and move on with execution. The conductor will send it off to all recipients in parallel.

Are signals reliable?

Never use a signal for mission-critical message delivery, because your code has no way of knowing whether the message reached its destination. If you do use it to facilitate important activities, make sure there’s always a slower but more dependable way of accessing the information contained in the message. For remote signals, this could include Bob following up with Alice via remote calls; for local signals, Bob could persist the message to the DHT and Alice could periodically poll for new messages.