Holochain

Distributed Coordination by Scaled Consent, not Global Consensus

Eric Harris-Braun, Arthur Brock, and Paul d'Aoust

(Dated: 2024-11-08 — v2.0)

We present a generalized system for large-scale distributed coordination that does not rely on global consensus, explicating the problem instead through the lens of scaling consent. After establishing initial axioms, we illustrate their application for coordination in games, then provide a formal outline of the necessary integrity guarantees and functional components needed to deliver those guarantees. Finally, we provide a fairly detailed, high-level, technical specification for our operating implementation for scaling consent.

Introduction

Preamble – A Focus on Practice, Not Just Theory

The original/alpha version of the Holochain white paper[1] took a formal approach to modeling generalized distributed computation schemes and contrasted Holochain's approach with blockchain-based systems. It also provided formal reasoning for the scaling and performance benefits of Holochain's agent-centric approach.

When dealing with distributed systems, however, the application of logical models and formal proofs are often deceiving. This stems from how easy it is to define sets and conditions which are logically solid in theory but fundamentally impossible and unintelligible in practice. Since our primary intent with Holochain is to provide a highly functional and scalable framework for sophisticated decentralized coordination, our focus must be on what is practicable, and resist the pull of the purely conceptual which frequently steers builders into unwieldy architectures.

Note how easy it is to reference a theoretical set like "all living persons" or "all nodes in the network." But in reality, it is impossible to identify that set in the physical world. Even if one could eliminate the fuzzy boundaries in the meaning of "persons" or "living," there is no way to discover and record the information quickly enough about who is dying, being resuscitated, and being born to construct the actual data set. Likewise, no single agent on a network can determine with certainty which nodes have come online, gone offline, or have declared themselves as new nodes. Also, since network partitions are a real, at any moment, one must question which partition is considered "the network," and how to enable a single node or group of nodes to continue operating appropriately even when no longer connected to the main network.

The initial example should be a comparatively easy data set to work with, since it changes relatively slowly. Typically each person undergoes a state change only twice in their life (when they become a living person, and when they cease to be one). However, the real-world use-cases that modern distributed tooling needs to handle involve data sets with much more rapid and complex changes. A more apt logical construct might be "all people with just one foot on the ground". Membership in this set changes quite rapidly – around 1/2 billion times per second[2].

It should be obvious there is no practical way to work with that data set without requiring actions that either break reality (like freezing time) or asserting a god-like, omniscient being, who not only has instantaneous access to all knowledge (requiring information propagation faster than the speed of light), but also has infinite attention to all states (likely requiring infinite energy). However, since current computing architectures are bound by laws of physics, we should avoid the temptation of injecting such impossible constructs into our theoretical models. A proof that involves simple logical concepts which cannot be reliably worked with in practice is not much of a proof at all.

"Global state" and strategies for consensus about it are exactly one of these dodgy constructs which are easy concepts, but involve a drastic reduction of complexity, agency, and degrees of freedom to reflect a small subset of events happening in physical reality. Yet most current distributed systems undertake the expensive task of having each node construct and maintain this unwieldy global fiction. So, for example, although many blockchains run on tens of thousands of processors, they advance in lockstep as if a single-threaded process, and they are only reliable for very simple world models, like moving tokens (subtracting from a number in one address and adding it to the number in another address).

"State" within the local computing context is likely rooted in the concept of the Turing Tape[3] or Von Neumann lin-

[1] https://holochain.org/documents/holochain-white-paper-alpha.pdf

[2] *Globally, The Average Person Walks About 5,000 Steps Per Day*, American Institute of Cancer Research, 2017 https://www.aicr.org/news/globally-the-average-person-walks-about-5000-steps-per-day/.

[3] See https://en.wikipedia.org/wiki/Turing_machine.

ear memory address architecture[4] which assume a single tape head or system bus for making changes to the single local memory space where changes are stored. With the introduction of multi-core processors, programmers encountered the myriad problems of having multiple agents (CPUs) operating on just one shared local state. They developed various strategies to enforce memory safety for concurrent local operations. So, in distributed computing, people extrapolated these local strategies and starting inventing some new ones, **still in the attempt to manage one single state across many physical machines.** The assumption of the need to sustain this simple logical concept of managing one global state persisted, even when that concept was mapped onto a physical topology which made it fundamentally unknowable in practice.

Early influential works in decentralized computing (such as the Byzantine Generals Problem[5]) may have also set such expectations. Those papers were written in the context of reaching consensus in finite control systems where there was a known number of sensors and states, and the goal was to reach a unified decision (like nine generals deciding a time for all to attack). Therefore, to be Byzantine Fault Tolerant, seems simply that a system is tolerant of the kinds of faults introduced by the generals problem (messages that are corrupted, misordered, or lost and generals/nodes that are malicious), but most distributed systems have assumed that global consensus must be constructed and maintained in order to reach a unified outcome. In this paper, we will detail some more efficient paths to enable agents to act without a construct of global consensus at all, yet still have strong guarantees that even when nodes act in partitioned groups or individually, they will reach a unified outcome.

So, while the formalizations from the original Holochain white paper are still valid in theory, this white paper is more concerned with addressing what works in practice. We will start by stating our underlying assumptions as axioms – each of which correlates to architectural properties of Holochain. And we will take special care not to make grand, categorical statements which cannot be implemented inside the constraints of computational systems bounded by the laws of physics.

[4] See https://en.wikipedia.org/wiki/Von_Neumann_architecture.

[5] Some readers may come to the problems of distributed coordination from the framework laid out by the literature on Byzantine Fault Tolerance such as *The Byzantine Generals Problem*, Leslie Lamport, Robert Shostak, and Marshall Pease https://lamport.azurewebsites.net/pubs/byz.pdf, and *Reaching Agreement in the Presence of Faults*, Marshall Pease, Robert Shostak, and Leslie Lamport https://dl.acm.org/doi/pdf/10.1145/322186.322188. For a story elucidating our frame that isn't about generals coordinating an attack, please see our paper *The Players of Ludos: Beyond Byzantium* https://holochain.org/documents/holochain-players-of-ludos.pdf.

Axioms – Our Underlying Basis for Coordination

Here we spell out the assumptions upon which we have built our approach to address the challenges of decentralized coordination.

First, let us clarify what we mean by coordination. Our goals for coordination are:

- To enable a group to establish ground rules which form the context needed for coordination,
- To enable agents in the group to take effective or correct action inside that context,
- To protect agents and the group from incorrect actions any agent may take.

Axioms for Multi-agent Coordination Through Scaled Consent

1. Agency is defined by the ability to take individual action.
2. "State" is persisted data that an agent can access through introspection.
3. It is easy to agree on a starting state; therefore it is easy to establish ground rules for coordination up front.
4. It is hard to maintain a unified, dynamic, shared state across a network, because of the constraints of physics.
5. Since only local time is knowable, non-local ordering is constructed by explicit reference.
6. Agents always act based only on their state; that is, data they can access locally.
7. Incorrect actions taken by an agent must harm only themself.
8. Long-term coordination must include the ability to orchestrate changes to ground rules.

Detailed Axioms and Architectural Consequences

The aforementioned axioms have affected the design of Holochain in the following ways.

Agency is defined by the ability to take individual action: Each agent is the sole authority for changing their state; the corollary of this is that an agent *cannot* change other agents' states. Since Holochain uses cryptography to eliminate many types of faults, this primarily means constructing a public/private key pair and using it to sign state changes recorded on an append-only log of the agent's actions. The log contains only actions of this agent, and writing to it (changing their own state), then sharing their state changes, is essentially the only authority (in terms of authorship) an agent has.

"State" is persisted data, local to an agent, that the agent can access through introspection: Because each agent is the sole author of their state[6], agents

[6] The "state" of a Holochain app does not generally include

interact with their world by sharing their own state changes and discovering, querying, and integrating state changes from other agents. Agents must be able to safely do so regardless of whether the peer delivering such data is its original author. Once an agent holds the data (whether because they authored it or received via networked communication) it is now part of their introspectable state. To act on such data, the agent still must verify whether it is true/false, complete/incomplete, authentic/forged, isolated/connected, reliable/suspect, etc.

It is easy to agree on a starting state; therefore it is easy to establish ground rules for coordination up front: The very first entry in an agent's state log for an app is the hashed reference to the code which establishes the grammar of coordination for that app. This code defines data structures, linking patterns, validation rules, business logic, and roles which are used and mutually enforced by all participating agents. The hash of this first entry defines the space and methods of coordination – agents only need to coordinate with other agents who "speak the same language". This establishes intentional partitions between networks in support of scalability, because an agent does not need to interact with all agents running Holochain apps, only the agents operating under the same ground rules. This simplifies and focuses overhead for validation and coordination.

It is hard to maintain a unified, dynamic, shared state across a network, because of the constraints of physics: In a distributed and open system, which enables autonomous agents to make valid changes to their state and freely associate with other agents in order to communicate those state changes to them, one cannot expect any one agent to know the state of all other agents across whatever partitions they may be in and whatever realtime changes they may be making. Such an assumption requires either centralization or omniscience. However, it is feasible to ensure strong eventual consistency[7], so that when any agents interact with and integrate the same data, all will converge to matching conclusions about the validity of any state change it represents.

Since only local time is knowable, non-local ordering is constructed by explicit reference: In the physical universe, entities experience time only as a local phenomenon; any awareness of other entities' experience of time comes from direct interaction with them. Thus, "global time" as a total ordering of causal events does not exist and entities' interactions form only partial orderings.

In Holochain, "direct interaction" comes in the form of explicit hash references to other data. The first simple structure for constructing order by reference is that each agent's action log is structured as an unbroken hash chain, with every new action containing the hash of the prior action. (Timestamps are also included, but are understood to be an expression of local time only.) When agents make state changes that rely on their prior state, the chain automatically provides explicit sequence. When an agent's action has logical dependencies on another agent's data, they must reference the hash of those action(s) to provide explicit ordering. In almost every application, there is no need to construct absolute time or sequencing to guarantee correct action; the only applications that absolutely require this are ones that deal with rivalrous state data, that is, exclusive ownership of a resource. If the problem cannot be restructured to eliminate all rivalrous state data, Holochain provides tools to implement conflict resolution or micro-consensus for that small subset of data for which it remains useful.

Agents always act based on their state; that is, data they can access locally: Since agents must act on what they know, they should be able to act *as soon* as they have have whatever local knowledge they need to take an action, with the assurance that any other nodes validating their action will reach the same conclusion. There is no reason to wait for other agents to reach a state *unless that is the confidence threshold required* to take that particular action. It is possible to architect agent-centric solutions to most decentralized problems which are many orders of magnitude more efficient than managing global, data-centric consensus. For example, this even includes building cryptocurrencies structured as peer-to-peer accounting instead of global tracking of token ownership, enabling the transacting agents (the only ones who are changing their states) to validate each other's actions and countersign a transaction independent from the rest of the network, who will validate it when they receive it after it is done.

Incorrect actions taken by an agent must harm only themself: We mentioned in the goals of collaboration that incorrect actions must not harm other agents or the integrity of the group. This is accomplished via the validation that occurs when data is persisted in the network. When a node receives a data element which it is supposed to store and serve as part of the architecture of global visibility into local state changes, they must validate it before integrating and serving it to others. For the previous example of a cryptocurrency, if the sender did not have enough credits in their chain state for the amount they are sending, the transaction would fail validation. The validating agents mark this action invalid, add both parties who signed the transaction to their blocked peers list, and publish a "warrant" letting others know about the corrupt nodes so other agents can block them. These warrants function as an immune system which protects individuals and the group from malicious or corrupt actors. The agents did not need to be prevented from taking the bad action, because they only changed their

---

ephemeral, non-persisted data such as what network sessions with other agents one may currently have open, although Holochain itself uses that kind of data to drive coordination.

[7] *Conflict-free Replicated Data Types*, Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski https://pages.lip6.fr/Marc.Shapiro/papers/RR-7687.pdf.

own states, and the bad action becomes the proof needed for the warrant to protect others from it.

Long-term coordination must include the ability to orchestrate changes to ground rules: Coordination cannot be effective without comprehension of the real-world context within which it is happening. However, agents cannot fully comprehend their context at first; understanding comes with interaction over time. And as the agents interact with their context, they may need to evolve their understanding, as they encounter new new situations which were not comprehended when the current ground rules were established. Finally, the act of interacting with a context changes the context itself. Any "grammar" in which ground rules are written must be expressive enough to write rules that address the needs of the problem domain as well as a capacity to evolve rules in response to changing context. In Holochain, the ground rules for a group are expressed in executable code. Its tools include an affordance for agents to migrate to a new group with a new set of rules, as well as the ability for an agent to "bridge" their presence in two different groups via cross-space function calls.

Building a distributed coordination framework starting from these axioms results in a system that empowers agents to take independent and autonomous action with partial information as soon as they have whatever they need to ensure it is a correct action. This constitutes a significant departure from the frame of thinking that Byzantine Fault Tolerance traditionally assumes: that complete consensus must be constructed *before* an agent can act.

From Global Consensus to Scaled Consent

We start from the understanding that networks, social spaces, and decentralized activities are inherently uncertain. Thus, coordination is never about absolute certainty; rather, it is about the capacity to remove sufficient uncertainty to provide confidence for action, which is always contextual. In distributed systems, it is absolutely fundamental to understand that every action taken by an agent happens because that agent has crossed a confidence threshold of some sort – from its own point of view, the action is appropriate to take.

Fault Tolerance and Reducing Uncertainty

Like blockchain and other cryptographic systems, Holochain starts the path of establishing confidence by leveraging cryptography to reduce uncertainties which remove most of the sources of Byzantine faults:

- **Corrupt Messages:** Data is retrieved by cryptographic hash which makes records self-validating, ensuring they are as requested and remain uncorrupted by checking data received against requested hash. Network messages are also cryptographically signed.
- **Misordered Messages:** Each agent writes their actions to a local append-only cryptographic hash chain, and must make explicit references to the hashes of any other agent's actions which one of their actions logically relies on, thus establishing indisputable ordering of data.
- **Lost Messages:** If the validity of any action relies on prior data, there will be either missing hash references or chain links which can be explicitly retrieved or have validation paused until available.
- **Forged Messages or Actions:** Each action is signed in sequence to its author's local hash chain. The public signing key is the same as the agent's address on the network. Hence all actions or messages are self-validating with respect to identity of author.
- **Malicious Actors/Actions:** Actions are validated based on local state established by the sequence of actions in an agent's hash chain, plus any actions included by reference. This enables every peer who is responsible for performing such validation to reach the same deterministic conclusion regarding validity.

These strategies help reduce sources of uncertainty; however, when it comes to concerns related to "consensus," still admit actions which individually pass validation but conflict with each other in content, substance, or perspective.

Increasing Gradients of Certainty

Given the above, we propose a very simple approach to creating tooling capacity for building increasing certainty: **enable validated global visibility, on demand, of local state**. In this approach, we distinguish between *authorship*, which is about local state changes initiated by agents, and *responsibility*, which is about distributing the workload of validating and serving records of local state changes across the participants in the network. This approach requires that we:

1. Ensure that all agents can *reliably* see what's going on; i.e., offer a framework for adding to and querying a collectively held database in which there is a minimum or "floor" of certainty regarding the contents and authorship of data even in the presence of an unbounded number of adversaries.
2. Ensure that all agents know the "ground-rules"; i.e., offer a framework for composing many small units of social agreement in which players can add elements of deterministic certainty into their interactions, yielding an appropriate level of certainty ranging from arbitrarily low to arbitrarily high.

The first point we deliver through various types of **Intrinsic Data Integrity**. We use a number of cryptographic methods to create self-proving data of various types:

- **Provenance**: An agent's network address is their public key. Thus, when interacting with agents it's possible to have deterministic confidence in whom one is interacting with because there is no identity

layer between network locations subject to attack surface. I.e., unlike a web address, you don't need a domain name certificate associated with the domain name to become confident of "whom" you are talking to.

- **Signatures**: Because provenance is a public key, it's also easy to create self-proving authenticity. All messages sent, and all data committed to chains, is signed by agents using their public key. Thus any agent can immediately, and with high confidence, verify the authenticity of messages and data.
- **Hashes**: All shared data in a Holochain application is addressed by its hash. Thus, when retrieving data it's possible to have deterministic confidence that it hasn't been tampered with by whoever was storing or relaying it.
- **Monotonicity**: The system is both structurally and logically monotonic. Structurally, local state is append-only and shared state can only grow. Data can be marked as deleted, but it is never actually removed from the history of the agent who authored it. Logically, once any peer has identified that a state change is invalid, no peers should identify it as valid.
- **Common genesis**: The validation rules and joining criteria of an application are the first entries in every agent's chain. This provides a mechanism for self-proving, shared ground rules. Any agent can examine the chain of any other agent all the way back to the source and thus have high confidence that they have actually committed to play by the same rules.

Building upon this fundament, we deliver the second point through the ability to compose various types of **Validation Rules**. Validation Rules create certainty in the following dimensions, with some examples:

- **Content**: a string does not exceed a maximum length
- **Structure**: an entry consists of a certain set of types of data[8]
- **Sequence**: someone cannot spend credits they have not already received
- **Process**: a transaction must be approved and signed by a notary
- **Behavior**: one does not take an action more frequently than a certain rate
- **Dependency**: an editor can only make changes if another agent has given them prior authorization

The two domains of Intrinsic Data Integrity and Validation Rules, and their component dimensionality, amounts to what we might call a "Geometry of Certainty". Inside the clarity of such a geometry, the gradients of certainty become both much more visible, and much easier to build for appropriately. Thus it provides a context of agents being able to scale up their consent to play together in ways that meet their safety requirements. This is why we call our approach "Scaling Consent." It is what enables coherent collaborative action without first achieving global consensus.

Scaling Coherence across Consenting Agents

The concept of **social coherence** may be the single most important design goal for Holochain applications – to provide a simple and consistent means of mutually enforcing shared ground rules appropriate to a social context. Some applications may require stricter validation because they contain high-value data with weak trust relationships between peers. Other apps which hold informal data or have higher relational trust between agents may be significantly less strict. Part of Holochain's scalability comes from the ability to implement appropriate coherence for each application's context. To illustrate appropriate social coherence, the existence of and resolution of conflicts in rivalrous data provides some clear examples.

Consider a social microblogging application built on Holochain. Since the precise global ordering of most actions is not vital, there is no reason to undertake the coordination overhead of global consensus for each post, like, follow, unfollow, reply, etc. Instead, simple causal ordering, in which data explicitly refers to its logical predecessors, will suffice for almost all actions.

If this application, unaware of total global ordering, ran at the scale of X (formerly Twitter) with hundreds of millions of daily users, a serious network partition[9] such as an extended loss in intercontinental connectivity would not cause a change in functionality for users, beyond being unable to see new posts from the far side of the partition. Old data would be accessible in the near side of the partition, and everything would keep functioning for both hemispheres. If this continued for a week, and the partition was resolved, all the data from both hemispheres would merge gracefully except for one possible source of conflict – new username registrations – because they are the only rivalrous data in such an application.

Now, a given group's rules for social coherence may not require username registrations to be unique across all participants. Systems that refer to participants by a random

[8] While Per Martin-Löf demonstrated (see https://en.wikipedia.org/wiki/Intuitionistic_type_theory) that values can be unified with classical types into a single dependent type theory (see https://en.wikipedia.org/wiki/Dependent_type), thus showing that content and structure can be equivalent and share a single calculus, here we distinguish the two in order to speak a language that is more familiar to programmers.

[9] It should be noted that communication latency induces conditions equivalent to a network partition, differing only in scope; therefore, there is still a risk of conflicting username registrations even in an unpartitioned network.

unique key, allowing participants to identify themselves and others by assigning non-unique "petnames"[10] (personally meaningful identifiers) to those keys, are proven to be usable in cases such as Signal Messenger and Secure Scuttlebutt[11].

But let us assume that users of this application demand unique usernames in the manner of X. It could employ one of a number of strategies for resolving or preventing conflicts:

1. Users could timestamp username registrations using a neutral, trusted timestamping strategy such as Roughtime[12], and the application would automatically resolve a conflict by favoring the earliest registration time.
2. Conflicts could be permitted, but upon detection, a social resolution procedure could be engaged, possibly processing through multiple stages if less costly strategies fail. Assisted by logical or cryptographic algorithms, this procedure could take forms such as:
    - A relationship-building approach, in which contestants are encouraged to sort it out amongst themselves, ending in one or both parties releasing their registration,
    - Awarding the username to the participant with the highest reputation or social capital, or
    - An auction.
3. Similar to client/server or blockchain approaches, a set of one or more witnesses can be elected to approve all name registrations and ensure there are no conflicts. While this approach prevents conflicts from happening, it requires a majority of a known set of witnesses, and any partition which contains a minority will not be able to process registrations.

Each one of these strategies achieves the same outcome while embodying very different patterns of social coherence; and in each case, there was no resolution overhead expended on all the non-rivalrous forms of data.

Another commonly used example of the rivalrous data problem is a "double-spend" attempt in a cryptocurrency. This involves fooling two separate parties into receiving units from the same pool of currency, such that the same units are sent twice, thus artificially expanding one's outflow of currency beyond what should be possible. Each transaction is valid in isolation, but conflicts with the other.

Cryptocurrencies in Holochain are typically implemented as accounting records stored in the histories of individual agents' hash chains – If Alice sends Bob 1 million credits of a currency, only Alice and Bob are changing their states, so they can move forward with the transaction as soon as they are confident the other party has committed to do so, and that all parties are in a position to do so validly.

Accordingly, the double-spend problem presents differently. It involves Alice "forking" her own hash chain: if Alice tries to send her 1 million credits to Bob and Carol simultaneously, then each of those transaction records in her chain will point to the same parent record hash. Each looks valid on its own, but taken together they demonstrate an invalid chain fork.

The protections an application implements against this kind of forking may be very different based on the social context and purpose of the application. At its most basic, the application's shared database (described in requirement 1 under Increasing Gradients of Certainty) acts as an "ambient witness" to all transactions. This allows agents to see each other's past behavior, including whether they have forked their chains.

An application which assumes high network uptime, low latency, and low risk of partitions might prevent this by requiring a time delay between Alice's promise of funds and Bob or Carol's acceptance, giving them time to check the shared database for proof that Alice has published a promise to them, and only them. Upon detection of the fork neither will accept the funds.

However, if this currency were designed to work in regions with unreliable network connectivity but strong, long-term trust relationships between members, it may not require such protections. This would increase the risk of Alice forking her chain, but it could provide a way for double-spends to be remedied after the fact. If both Bob and Carol know Alice, where her business is, or where she lives, there are social repercussions to cheating, and Alice will have an incentive to fix her public record of trying to cheat people, for instance cancelling one branch of the fork and eventually delivering a new valid payment to the recipient who had received payment via the cancelled branch.

The above examples illustrate how the demand for appropriate social coherence drives an application's approach to selecting from affordances that Holochain provides to resolve conflicts and reach unified outcomes. They also demonstrate how coordination overhead becomes unnecessarily high if all non-rivalrous data is treated as rivalrous, and how forcing conflict resolution into a single costly pattern should not apply to all data nor in all social contexts. Agentic assessment of the social context, and mutual enforcement of only the necessary rules for coherence, enables agents to act as soon as their certainty threshold is reached. This is always true, whether it is reached through centralized coordination, a Byzantine Generals' Problem approach, or blockchain consensus algorithms.

---

[10] *An Introduction to Petname Systems*, Marc Stiegler, 2005 http://www.skyhunter.com/marcs/petnames/IntroPetNames.html.

[11] See https://signal.org and https://scuttlebutt.nz.

[12] Roughtime IETF draft, W Ladd, Akamai Technologies, M Dansarie, Netnod, 2024 https://datatracker.ietf.org/doc/draft-ietf-ntp-roughtime/.

Conclusion

While our axioms may seem obvious to those familiar with distributed and agent-based systems, they yield surprising and often-overlooked consequences when taken to their logical conclusions in the design of a practical distributed system. As we have seen, such a system is likely to be more efficient than a consensus-based system of equivalent functionality in terms of computation, communication, and storage. It is also likely to be more respectful of the agency of individuals and the group than either consensus or centralized systems: there is an underlying theme in these axioms, that of full agency constrained by the obligation to respect the agency of others (and indeed the inability to override their agency).

As we have also argued, and as other authors formally prove[13], such freedom need not compromise the technical or social integrity needed to take confident action. There is a broad space of design possibilities that allow groups to embody non-coercive, highly coherent, contextually appropriate patterns of coordination even in the presence of malicious actors. In the remainder of this paper, we will explore how Holochain's design realizes the expressivity necessary to build these patterns.

Holochain Design Overview: A Game Play Metaphor

It may help to understand the design of Holochain through a well-known pattern of agentic collaboration: playing games.

Playing Games

People define the rules of a *Game* they want to play together. As *Players* join the *Game* and start playing, they append new *Records* of their play to their own *Action* history. Then they update the *Game Board* by sharing the *Records* of their *Actions*[14] with other players.

The first requirement to create social coherence is ensuring that people are playing the same game; therefore, the very first record in every Agent's history is the rules of the game by which they agree to play. Obviously, Players are not in the same game or able to use the same Game Board if they don't start with the same rule set. These rules are the actual computer code that is executed in running the Game, making moves and validating the Actions of all Players.

[13] *Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases*, Martin Kleppmann and Heidi Howard, 2020 https://arxiv.org/pdf/2012.00472.

[14] You can think of this somewhat like correspondence chess, but with substantially more formality.

System Description

We can describe the system as Agents, who play Games together by taking Actions, the Records of which are held in a distributed Ledger that is built by sharing these Records over a Network with other Agents. We capitalize terms that comprise the ontological units of the system, and which are formally described in the later sections.

Agents

Agents have these properties:

1. Agents are the only source of Actions in the system, thus Agents are the source of agency. All such Actions are uniquely identifiable as to which Agent took them; i.e., all Actions are signed by public-key cryptography (see Actions below).
2. Agents are uniquely addressable by other Agents.
3. An Agent's address is its public key.
4. Agents share Records of the Actions they take with other Agents through distributed storage so that those Records can be retrieved by other Agents reliably.
5. Agents validate received Actions before storing or sharing them.
6. Agents respond to requests for stored information.
7. Agents can send messages with arbitrary content directly to other Agents.

Games

Games have these properties:

1. A Game consists of an Integrity specification with these parts:
    1. A deterministic description of the types of data that are used to record some "play" in the game. Such data is called an Entry, where the act of generating such data is called an Action, which is also recorded. Note: both types of data, the content of the play (Entry) and the meta-data about the play (Action), when taken together, are called a Record.
    2. A deterministic description of of the types of relations between Entries or Actions. Such a relation is called a Link.
    3. A deterministic description of a properly formed Membrane Proof, a credential that grants an Agent permission to join a Game. This description may not be able to fully validate a Membrane Proof if its validity depends on data elsewhere on the Game Board, as the Agent's Membrane Proof is checked against this description before they join.
    4. A deterministic description of the validity of an Action and any Entry, Link, or Membrane

Proof data it contains, in the context in which it is produced. This may include rules about the contents of data fields, the author's current state (for instance, whether a game move is allowed given their history), presence of dependencies (such as moves by their opponent or certificates permitting them to play a certain role), and any other rules that may be expressed deterministically given the context available to the Action.

2. Along with the Integrity specification, a Game also consists of a Coordination specification. This specification contains instruction sets that wrap the reading, writing, and modification of Actions into function call units and thus serve as an API to the Game. For example, for a blogging "Game" one such function call might be `create_post` which takes a number of Actions that atomically write a number of Records to the Agent's Source Chain, which include a Create-Entry Action for the post as well as Create-Link Actions relating the post to other Entries such as a "category" Entry (see below for definitions of Actions and Source Chain).
3. Each instance of the Game is played on its own Game Board which exists as a unique and independent network of Agents playing that Game. The consequence of this is that Games cannot interact with each other directly, as all action in the system is only taken by Agents. Note that Games can be composed together, but only by groups of Agents all playing across multiple games. This at first may seem like a weakness, but it's part of a key design decision that contributes to the system's overall design goals of evolvability. Essentially this creates the pattern of game-within-a-game. For example a chess tournament is really two games: the game of "chess", and the game of "tournament". Composition thus happens at the edges of games, through the agency of the Agents who are playing both Games, which allows each game to remain coherent on its own terms.

In keeping with the metaphor of Game, we also refer to the Integrity specification as the Validation Rules of the Game.

We also refer to Integrity specification of a Game as its DNA because this evokes the pattern of all the "cells" in the social "body" as being built out of the same instruction set, thus being the ground of self for that social body. We also call an Integrity or Coordination specification generically by the name Zomes (short for chromosomes) as they also function as composable units for building larger bodies of "genetic code" for said body.

Actions (and Entries and Records)

Actions have these properties:

1. An Action has cryptographic provenance in that it is signed by the Agent that took the Action.
2. Actions are Recorded in a monotonically temporally increasing hash-chain by the Agent that takes the Action. We refer to this as a hash-chain because each Action includes in it the hash of the previous Action, thus creating a tamper-proof Action history.
3. Actions are addressed by the hash of the Action.
4. There are a number of Action types:
    1. `Dna`: An action which contains the hash of the DNA's code, demonstrating that the Agent possesses a copy of the rules of the Game and agrees to abide by them.
    2. `AgentValidationPkg`: An action which presents an Agent's Membrane Proof, the data that proves that they have permission to join a Game.
    3. `Create`: An Action for adding new Game-specific content. We call such content an Entry. The Entry may be declared as public, and will thus be published by the Agent to the network, or declared as private, in which case publishing is limited to just the Action data and not the content. Entries are addressed by their hash, and thus for Create Actions, this Entry hash is included in the Action. Thus sometimes the Action may be understood as "meta-data" where the Entry is understood as "data"[15]. Note that Actions and Entries are thus independently addressable and retrievable. This is a valuable property of the system. Note also that many actions (for example ones taken by different agents) may create the exact same Entry; e.g., a hash-tag may be its own entry but be created by different people.
    4. `Update`: An Action which adds new Game-specific content onto the chain that is intended to update an existing entry creation Action (either a Create or an Update).
    5. `Delete`: An Action which indicates a previous entry creation Action should be considered deleted.

[15] In many cryptographic systems hash-chains are thought of as having "headers" and "entries". Historically in Holochain development we also used that nomenclature, but realized that the name "header" comes from an implementation detail in building hash chains. Ontologically what's actually happening is that in building such intrinsic integrity data structures, not only must we record the "content" of what is to be written, but we must also record data about the act of writing itself; i.e., who is doing the writing, when they did so, and what they previously wrote. Thus, in keeping with the core ontology of agent-centricity, we switched to using the term "Action" instead of "header", but we retain the name Entry for that which is written.

6. `CreateLink`: An Action that unidirectionally links one hash to another.
7. `DeleteLink`: An Action that indicates a previous link Action should be considered deleted.
8. `InitZomesComplete`: An action which announces that the Agent has completed any necessary preparations and is ready to play the Game.
9. `OpenChain`: An action which indicates that an Agent is continuing their participation in this Game from another Source Chain or an entirely different Game.
10. `CloseChain`: An action which indicates that an Agent is ending their participation in this Game on this Source Chain, and may be continuing their participation in another Source Chain or an entirely different Game.

5. A Record is just a name for both an Action and, when applicable, its Entry, taken together. As an implementation detail, note that for actions other than Create and Update we don't need to address the content of the Action separately, in which case the Record contains no Entry and we simply retrieve all the data we need from the recorded Action.
6. Subsets of Agents can mutually consent to a single Action by atomically recording the Action in their history through Countersigning. Countersigning can also be seen as an affordance in the system for "micro-consensus" when that is necessary.

The Distributed Ledger

The Ledger, when seen systemically as a whole, consists of the collection of all Records of Actions and their Entries in a Game that have been taken by all the Agents together. The Ledger is stored in two distinct forms:

1. As self-recorded Source Chains of each of the Agent's Actions.
2. As a Graphing Distributed Hash Table that results from the sharing and validation of these Actions across Agents, collectively taking responsibility for validating and storing portions of the data.

The first form ensures the integrity of all data stored in the network because it creates the coherence of provenance and ordering of local state. The second form ensures the validity and visibility of that data.

Note, there is never a point or place where a canonical copy of the entire state of the ledger exists. It is always distributed, either as the Source Chain of Actions taken by a single agent, or broken into parts and stored after validation by other participating Agents in the system. An Agent may elect to take responsibility for validating and storing the entire contents of the Ledger, but as Holochain is an eventually consistent system, their copy can never be said to be canonical.

The Ledger as Local State: Source Chain An Agent's Source Chain for a Game consists of a hash chain of Records of Actions taken by the Agent in accordance with the validation rules of that Game.

A Record consists of an Action which holds context and points to an Entry which is the content of the Action. The context information held by the Action includes:

1. The Action type (e.g. create/update/delete/link, etc)
2. A time-stamp
3. The hash of the previous Action (to create the chain)
4. The sequence index of the Action in the chain
5. The hash of the Entry

The first few Records of every Source Chain - called Genesis Records - create a "common ground" for all the agents "playing" a Game to be able to verify the Game and its "players" as follows:

1. The first Record always contains the full Validation Rules of the Game, and is hence referred to as the DNA. It's what makes each Game unique, and, as part of validation, always allows Agents to check that other Agents are playing the same Game.
2. The second Record is a Game-specific Membrane Proof, which allows Games to create Validation rules for permissioned access to a Game.
3. The third Record is the Agent's address, i.e. its public key.
4. The final Genesis Records are any Game-specific Records added during Genesis, followed by an `InitZomesComplete` Record indicating the end of the Genesis Records.

All subsequent Records in the Source Chain are simply the Actions taken by that Agent. Note that Source Chains may end with a Closing Record which points to an opening record in a new Game.

The Ledger as Validated Shared State: Graphing DHT After Agents record the Actions they take to their Source Chains, they Publish these Actions to other Agents on the Network. Agents receiving published data validate it and make it available to other agents to query, thus creating a distributed database. Because all retrieval requests are keyed on the hashes of Actions or Entries, we describe this database as a Distributed Hash Table (DHT). Because such content-addressable stores create sparse spaces in which discovery is prohibitively expensive, we have extended the usual Put/Get operators of a DHT to include linking hashes to other hashes, thus creating a Graphing DHT.

As a distributed database, the DHT may be understood as a topological transform of many Agents' Source Chain states into a form that makes that data retrievable by all

the other Agents for different purposes. These purposes include:

1. Retrieval of Agent's Actions and created Entries
2. Confirmation of "good behavior" by retrieving an Agent's activity history which is used to verify that agents haven't forked their chains
3. Retrieval of link information
4. Retrieval of validation receipts

To achieve this end, we take advantage of the fact that an Agent's public key (which serves as its address) is in the same numeric space as the hashes of the data that we want to store and retrieve. Using this property, we can create a mapping between Agents and the portions of the overall data they are responsible for holding, by using a nearness algorithm between the Agent's public key and the address of the data to be stored. Agents that are "close" to a given piece of data are responsible to store it and are said to comprise a Neighborhood for that data. Hashing creates an essentially random distribution of which data will be stored with which Agents. The degree of redundancy of how many Agents should store copies of data is a per-Game parameter.

Agents periodically gossip with other Agents in their Neighborhood about the published data they've received, validating and updating their Records accordingly. This gossip ensures that eventually all Agents querying a Neighborhood for information will receive the same information. Furthermore it creates a social space for detecting bad actors. Because all gossiped data can intrinsically be validated, any Agents who cheat, including by changing their (or other's) histories, will be found out, and because all data includes Provenance, any bad actors can be definitively identified and ejected from the system.

See the Formal Design Elements section for more information on how Agents convert Source Chain data to operations that are published into the collectively stored data on the DHT, and how this works to provide eventual consistency, and the sections in System Correctness for details on detection of malicious actors.

System Correctness: Confidence

In the frame of the Byzantine Generals Problem, the correctness of a distributed coordination system is analyzed through the lens of "fault tolerance". In our frame we take on a broader scope and address the question of the many kinds of confidence necessary for a system's adoption and continued use. We identify and address the following dimensions of confidence:

1. **Fault Tolerance:** the system's resilience to external perturbations, both malicious and natural. Intrinsic integrity.
2. **Completeness/Fit:** the system's *a priori* design elements that demonstrate fitness for purpose. We demonstrate this by describing how Holochain addresses multi-agent reality binding, scalability, and shared-state finality.
3. **Security:** the system's ability to cope with intentional disruption by malicious action, beyond mere detection of faults.
4. **Evolvability:** the system's inherent architectural affordances for increasing confidence over time, especially based on data from failures of confidence in the above dimensions.

Our claim is that if all of these dimensions are sufficiently addressed, then the system takes on the properties of antifragility; that is, it becomes more resilient and coherent in the presence of perturbations rather than less.

Fault Tolerance

In distributed systems much has been written about Fault Tolerance especially to those faults known as "Byzantine" faults. These faults might be caused by either random chance or by malicious action. For aspects of failures in system confidence that arise purely from malicious action, see the Security section.

1. **Faults from unknown data provenance:** Because all data transmitted in the system is generated and cryptographically signed by Agents, and those signatures are also included in the hash-chains, it is always possible to verify any datum's provenance. Thus, faults from intentional or accidental impostors is not possible. The system cannot prevent malicious or incautious actors from stealing or revealing private keys, however, although it does include affordances to deal with these eventualities. These are discussed under Human Error.
2. **Faults from data corruptibility in transmission and storage:** Because all state data is stored along with a cryptographic hash of that data, and because all data is addressed and retrieved by that hash and can be compared against the hash, the only possible fault is that the corruption resulted in data that has the same hash. For Blake2b-256 hashing (which is what we use), this is known to be a vanishingly small possibility for both intentional and unintentional data corruption.[^corruption] Furthermore, because all data is stored as hash-chains, it is not possible for portions of data to be retroactively changed. Agents' Source Chains thus become immutable append-only event logs.

   One possible malicious act that an Agent can take is to roll back their chain to some point and start publishing different data from that point forward. But because the publishing protocol requires Agents to also publish all of their Actions to the neighborhood of their own public key, any Actions that lead to a forked chain will be easily and immediately

detected by simply detecting more than one action linked to the same previous action.

It is also possible to unintentionally rollback one's chain. Imagine a setting where a hard-drive corruption leads to a restore from an outdated backup. If a user starts adding to their chain from that state, it will appear as a rollback and fork to validators.

Holochain adds an affordance for such situations in which a good-faith actor can add a Record repudiating such an unintentional chain fork.

3. **Faults from temporal indeterminacy:** In general these faults do not apply to the system described here because it only relies on temporality where it is known that one can rely on it; i.e., when recording Actions that take place locally as experienced by an Agent. As these temporally recorded Actions are shared into the space in which nodes may receive messages in an unpredictable order, the system still guarantees eventual consistency (though not uniform global state) because of the intrinsic integrity of recorded hash-chains and deterministic validation. Additionally, see the Multi-agent reality binding (Countersigning) section for more details on how some of the use cases addressed by consensus systems are handled in this system.

Completeness/Fit

Multi-agent reality binding (Countersigning)

The addition of the single feature of Countersigning to Holochain enables our eventually consistent framework to provide most of the consensus assurances people seek from decentralized systems. Countersigning provides the capacity for specific groups of agents to mutually sign a single state-change on all their respective source-chains. It makes the deterministic validity of a single Entry require the cryptographic signatures of multiple agents instead of just one. Furthermore any slow-downs necessary to add coordinated countersigned entries are not just localized to the DNA involved, they are also localized to just the parties involved. The same parties can continue to interact in other DNAs.

The following are common use cases for countersigning:

- **Multi-Agent State Changes:** Some applications require changes that affect multiple agents simultaneously. Consider the transfer of a deed or tracking a chain of custody, where Alice transfers ownership or custody of something to Bob and they want to produce an **atomic change across both of their source chains**. We must be able to prevent indeterminate states like Alice committing a change releasing an item without Bob having taken possession yet, or Bob committing an entry acknowledging possession while Alice's release fails to commit. Holochain provides a countersigning process for multiple agents to momentarily lock their chains while they negotiate one matching entry that each one commits to their chain. An entry which has roles for multiple signers requires signed chain Actions from each counterparty to enter the validation process. This ensures no party's state changes unless every party's state changes.
- **Cryptocurrencies Based on P2P Accounting:** Extending the previous example, if Alice wants to transfer 100 units of a currency to Bob, they can both sign a single entry where Alice is in the spender role, and Bob the receiver. This provides similar guarantees as familiar double-entry accounting, ensuring changes happen to both accounts simultaneously. Someone's balance can be easily computed by replaying the transactions on their source chain, and both signing parties can be held accountable for any fraudulent transfers that break the data integrity rules of the currency application. There's no need for global time of transactions when each is clearly ordered by its sequence in the chains of the only accounts affected by the change.
- **Witnessed Authoritative Sequence:** Some applications may require an authoritative sequence of changes to a specific data type. Consider changes to membership of a group of administrators, where Carol and David are both members of the group, and Carol commits a change which removes David from the group, and David commits a change which removes Carol. With no global time clock to trust, whose change wins? An application can set up a small pool of N witnesses and configure any change to be the result of a countersigning session that requires M optional witnesses (where M > 50% of N). Whichever action the witnesses sign first would prevent the other action from being signed, because either Carol or David would have been successfully removed and would no longer be authorized participate in a countersigning session to remove the other.
- **Exclusive Control of Rivalrous Data:** Another common need for an authoritative time sequence involves determining control of rivalrous data such as name registrations. Using M of N signing from a witness pool makes it easy to require witnessing for only rivalrous data types, and forgo the overhead of witnessing for all other data. For example, a Twitter-like app would not need witnessing for tweets, follows, unfollows, likes, replies, etc, only for registration of new usernames and for name changes. This preserves the freedom for low-overhead and easy scaling by not forcing consensus to be managed on non-rivalrous data (which typically comprises the majority of the data in web apps).
- **Generalized Micro-Consensus: Entwined multi-agent state change:** Even though Holochain is agent-centric and designed to make

only local state changes, the countersigning process may be seen as an implementation of Byzantine consensus applied to specific data elements or situations. Contextual countersigning is exactly what circumvents the need for global consensus in Holochain applications.

Scaling

Holochain's architecture is specifically designed to maintain resilience and performance as both the number of users and interactions increase. Key factors contributing to its scaling capabilities include:

1. **Agent-centric approach:** Unlike traditional blockchain systems, which require global consensus before progressing, Holochain adopts an agent-centric approach where changes made to an agent's state become authoritative once stored on their chain, signed, and communicated to others via the DHT. As a result, agents are able to initiate actions without delay and in parallel to other agents initiating their own actions.
2. **Bottleneck-Free Sharded DHT:** Holochain's DHT is sharded, meaning that each node only stores a fraction of the total data, reducing the storage and computational requirements for each participant. At the same time, the storage of content with agents whose public key is "near" the hash of each Action or Entry, in combination with the use of Linking metadata attached to such hashes, transforms the DHT into a graphing DHT in which data discovery is simple in spite of the sparseness of the address space. When the agents responsible for validating a particular state change receive an authoring agent's proposed state change, they are able to:
    1. Request information from others in the DHT regarding the prior state of the authoring agent (where relevant), and
    2. Make use of their own copy of the app's validation rules to deterministically validate the change.

    While that agent and its validating peers are engaged with the creation and validation of a particular change to the state of the authors chain, in parallel, other agents are able to author state changes to their own chain and have these validated by the validating peers for each of those changes. This bottle-neck free architecture allows users to continue interacting with the system without waiting for global agreement.

    With singular actions by any particular agent (and the validation of those actions by a small number of other agents) able to occur simultaneously with singular actions by other agents as well as countersigned actions by particular groups of agents, he network is not updating state globally (as blockchains typically do) but is instead creating, validating, storing, and serving changes of the state of particular agents in parallel.
3. **Multiple networks**: In Holochain, each application (DNA) operates on its own independent network, effectively isolating the performance of individual apps. This prevents a high-traffic, data-heavy, or processing-heavy app from affecting the performance of other lighter apps within the ecosystem. Participants are able to decide for themselves which applications they want to participate in.
4. **Order of Complexity**: "Big O" notation is usually only applied to local computation based on handling `n` number of inputs. However, we may consider a new type of O-notation for decentralized systems which includes two inputs, `n` as the number transactions/inputs/actions, and `m` as the number of nodes/peers/agents/users, as a way of expressing the time complexity for both an individual node and for the aggregate power of the entire network of nodes. Most blockchains are some variant of $\mathcal{O}(n^2 * m)$ in their order of complexity. Every node must gossip and validate all state changes. However, Holochain retains a constant $\mathcal{O}(\frac{log(n)}{m})$ complexity for any network larger than a given size R, where R is the sharding threshold. As the number of nodes in the network grows, each node performs a static workload irrespective of network size; or expressed inversely, a smaller portion of the total network workload.

Shared-state Finality

Many blockchains approximate chain finality by assuming that the "longest chain wins." That strategy does not translate well to agent-centric chains, which are simply histories of an agent's actions. While there is no concern about forking global state because a Holochain app doesn't have one, we can imagine a situation where Alice and Bob have countersigned a transaction, then Alice forks her source chain by later committing an Action to an earlier sequence position in her chain. If the timestamp of this new, conflicting Action precedes the timestamp of the transaction with Bob, it could appear that Bob had knowingly participated in a transaction with a malicious actor, putting his own integrity in question. This can even happen non-maliciously when someone suffers data loss and restores from a backup after having made changes that were not included in the backup. While the initial beta version of Holochain does not offer fork finality protections for source chains, later versions will incorporate "meta-data hardening" which enables gossiping peers to tentatively solidify a state of affairs when they see that gossip for a time window has calmed and neighbors have converged on the same state. After this settling period (which might be set to something between 5 to 15 minutes) any later changes which would produce a conflict

(such as forking a chain) can be rejected, preserving the legitimacy of state changes that were made in good faith.

Security

The system's resilience to intentional gaming and disruption by malicious actors will be covered in depth in future papers, but here we provide an overview.

Many factors contribute to a system's ability to live up to the varying safety and security requirements of its users. In general, the approach taken in Holochain is to provide affordances that take into account the many types of real-world costs that result from adding security and safety to systems such that application developers can match the trade-offs of those costs to their application context. The integrity guarantees listed in the formal system description detail the fundamental data safety that Holochain applications provide. Some other important facets of system security and safety come from:

1. Gating access to functions that change local state, for which Holochain provides a unified and flexible Object Capabilities model
2. Detecting and blocking participation of bad actors, including attempts to flood a DHT with otherwise valid data, for which Holochain provides the affordances of validation and warranting.
3. Protection from attack categories
4. Resilience to human error

Gating Access via Cryptographic Object Capabilities

To use a Holochain application, end-users must trigger Zome Calls that effect local state changes on their Source Chains. Additionally, Zome Functions can make calls to other Zome Functions on remote nodes in the same app, or to other DNAs running on the same Conductor. All of these calls must happen in the context of some kind of permissioning system. Holochain's security model for calls is based on the Object-capability[16] security model, but augmented for a distributed cryptographic context in which we use cryptographic signatures to prove the necessary agency for taking action.

Access is thus mediated by Capability Grants of four types:

- **Author**: only the agent owning the source change can make the zome call
- **Assigned**: only the specified public key holders can make the zome call, as verified by a signature on the function call payload
- **Transferrable**: anybody with the given secret can make the zome call
- **Unrestricted**: anybody can make the zome call (no secret nor proof of authorized key needed to use this capability)

All zome calls must be signed and also take a required capability claim parameter that MUST be checked by the system for making the call. Agents record capability grants on their source chains and distribute their corresponding secrets as applicable according to the application's needs. Receivers of secrets can record them as private capability claim entries on their chains for later lookup and use. The "agent" type grant is just the agent's public key.

Validation & Warranting

We have already covered how Holochain's agent-centric validation and intrinsic data integrity provides security from malicious actors trying to introduce invalid or incorrect information into an Application's network, as every agent can deterministically verify data and thus secure itself. It is also important, however, to be able to eject malicious actors from network participation who generate or propagate invalid data, so as to proactively secure the network against the resource drain that future such actions from those actors may incur.

As agents publish their actions to the DHT, other agents serve as validators. When validation passes, they send a validation receipt back to the authoring agent, so they know the network has seen and stored their data. When validation fails, they send a negative validation receipt, known as a warrant, back to the author and their neighbors so the system can propagate these provably invalid attempted actions. This also flags the offending agent as corrupted or malicious so that other nodes can block them and stop interacting with the offending agent. Every node can confirm the warrant for themselves, as it is justified by the shared deterministic validation rules, of which all agents have a copy.

This enables a dynamic whereby any single honest agent can detect and report any invalid actions. So instead of needing a majority consensus to establish reliability of data (an "N/2 of N" trust model), Holochain enables "one good apple to heal the bunch" with a "1 of N" trust model for any data you acquire from agents on the network.

For even stricter situations, apps can achieve a "0 of N" trust model, where no external agents need to be trusted, because nodes can always validate data for themselves, independent of what any other nodes say.

Security from Attack Categories

Consensus Attacks This whole category of attack starts from the assumption that consensus is required for distributed systems. Because Holochain doesn't start from that assumption, the attack category really doesn't apply, but it's worth mentioning because there are a number of attacks on blockchain which threaten confidence in the reliability of the chain data through collusion between

[16] See https://en.wikipedia.org/wiki/Object-capability/_model

some majority of nodes. The usual thinking is that it takes a large number of nodes and massive amounts of computing power or financial incentives to prevent undue hijacking of consensus. However, since Holochain's data coherence doesn't derive from all nodes awaiting consensus, but rather on deterministic validation, nobody ever needs to trust a consensus lottery.

Sybil Attacks Since Holochain does not rely on any kind of majority consensus, it is already less vulnerable to Sybil Attacks, the creation of many fake colluding accounts which are typically used to overwhelm consensus of honest agents. And since Holochain enables "1 of N" and even "0 of N" trust models, Sybils cannot entirely overwhelm honest agents' ability to determine the validity of data.

Additionally, since Holochain is a heterogeneous environment in which every app operates on its own isolated network, a Sybil Attack can only be attempted on a single app's network at a time. For each app, an appropriate membrane can be defined on a spectrum from very open and permissive to closed and strict by defining validation rules on a Membrane Proof.

A membrane proof is passed in during the installation process of an agent's instance of the app, so that the proof can be committed to the agent's chain just ahead of their public key. An agent's public key acts as their address in that application's DHT network, and is created during the genesis process in order to join the network. Other agents can confirm whether an agent may join by validating the membership proof.

A large variety of membrane proofs is possible, ranging from none at all, loose social triangulation, or an invitation from any current user, to stricter invitation lists, proof-of-work requirements, or a kind of proof-of-stake showing the agent possesses and has staked some value which they lose if their account gets warranted.

We generally suggest that applications may want to enforce some kind of membrane against Sybils, not because consensus or data integrity are at risk but because carrying a lot of Sybils makes unnecessary work for honest agents running an application. We cover more about this in the next section.

Denial-of-Service Attacks Holochain is not systemically subject to denial-of-service attacks because there is no central point to attack. Because each application is its own network, attackers would have to flood every agent of every application to carry out a systemic denial-of-service attack; to do that would require knowing who all those agents are, which is also not recorded in one single place. One point of vulnerability is the bootstrap servers for an application. But this is not a systemic vulnerability, as each application can designate its own bootstrap server, and they can also be arbitrarily hardened against denial-of-service to suit the needs of the application.

Eclipse Attacks An Eclipse Attack happens when a newly joining node is prevented from ever joining the main/honest network because it initially connects to a dishonest node which only ever shares information about other colluding dishonest nodes. This attack is specific to gossip-based peer-to-peer networks such as Bitcoin, Holochain, and DHTs like IPFS.

Holochain apps bypass the risk of an Eclipse Attacks by providing an address for a bootstrap service which ensures your first connection is to a trusted or honest peer. If an application fails to provide a bootstrap service, nodes will try connecting via https://bootstrap.holochain.org which provides initial trusted peers, if those have been specified. If not specified, the default bootstrap service provides random access to any and all peers using the app, which at least ensures nodes cannot be partitioned from honest peers.

Application developers can take steps to further protect their users by providing in-app methods of exchanging signed pings with known nodes (such as a progenitor, migrator, notary, witness, or initial admin node) so a node can ensure it is not partitioned from the real network.

Human Error There are some aspects of security, especially those of human error, that all systems are subject to. People will still lose their keys, use weak passwords, get computer viruses, etc. But, crucially, in the realm of "System Correctness" and "confidence," the question that needs addressing is how the system interfaces with mechanisms to mitigate against human error. Holochain provides significant tooling to support key management in the form of its core Distributed Public Key Infrastructure (DPKI) and DeepKey app built on that infrastructure. Among other things, this tooling provides assistance in managing keys, managing revocation methods, and reclaiming control of applications when keys or devices have become compromised.

A definition and specification of a DPKI system is outside of the scope of this paper; see the DeepKey design specification[17] for a more thorough exploration.

Evolvability

For large-scale systems to work well over time, we contend that specific architectural elements and affordances make a significant difference in their capacity to evolve while maintaining overall coherence as they do so:

1. **Subsidiarity:** From the Wikipedia definition:

   > Subsidiarity is a principle of social organization that holds that social and political issues should be dealt with at the most immediate (or local) level that is consistent with their resolution.

   Subsidiarity enhances evolvability because it insulates the whole system from too much change, while simultaneously allowing change where it is needed.

[17] See https://github.com/holochain/deepkey/blob/main/docs/2023/README.m

Architecturally, however, subsidiarity is not easy to implement because it is rarely immediately obvious what level of any system is consistent with an issue's resolution.

In Holochain, the principle of subsidiarity is embodied in many ways, but crucially in the architecture of app instances having fully separate DNAs running on their own separate networks, each also having clear and differentiable Integrity and Coordination specifications. This creates very clear loci of change, both at the level of when the integrity rules of a DNA need to change, and at the level of how one interacts with a DNA. This allows applications to evolve exactly in the necessary area by updating only the DNA and DNA portion necessary for changing the specific functionality that needs evolving.

2. **Grammatic**[18] **composability:** Highly evolvable systems are built of grammatic elements that compose well with each other both "horizontally", which is the building of a vocabulary that fills out a given grammar, and "vertically" which is the creation of new grammars out of expressions of a lower level grammar. There is much more that can be said about grammatics and evolvability, but that is out of scope for this paper. However, we contend that the system as described above lives up to these criteria of having powerful grammatical elements that compose well as described. DNAs are essentially API definitions that can be used to create a large array of micro-services that can be assembled into small applications. Applications themselves can be assembled at the User Interface level. A number of frameworks in the Holochain ecosystem are already building off of this deep capacity for evolvability that is built into the system's architecture[19].

Holochain Formal Design Elements

Now we turn to a more formal and detailed presentation of the Holochain system, including assumptions, architecture, integrity guarantees, and formal state model.

Purpose of this Section: To provide an understanding of the functional requirements of Holochain and specify a technical implementation of the cryptographic state transitions and application processes that enforce Holochain's integrity guarantees.

Definition of Foundational Principles

- **Cryptography:** Holochain's integrity guarantees are largely enabled by cryptography. It is used in three main ways.
  - **Hashes:** Data is uniquely identified by its hash, which is the key used to retrieve the data from a Content Addressable Store.
  - **Signing:** Origination of data (for all storage and network communications) is verified by signing a hash with a private key.
  - **Encryption:** Data is encrypted at rest and on the wire throughout the system.
- **Agency:** Holochain is agent-centric. Each and every state change is a result of:
  1. A record of an agent's action,
  2. signed by the authoring agent,
  3. linearly sequenced and timestamped
  4. to their local source chain.

  Each agent is the sole authority for managing its local state (by virtue of controlling their private key required for signing new actions to their source chain).
- **Accountability:** Holochain is also socio-centric. Each Holochain application defines its set of mutually enforced data integrity rules. Every local state change gets validated by other agents to ensure that it adheres to the rules of that application. Peers also enforce limits on publishing rates, protect against network integrity threats, and can ban rule-breakers by a process we call *warranting*.
- **Data:** Unlike some other decentralized approaches, in Holochain, data does not have first-order, independent, ontological existence. Every record in the shared DHT network space MUST CARRY its provenance from a local source chain as described below.
- **Provenance:** Each record created in a Holochain application starts as an action pair on someone's local source chain. As such, even when published to the shared DHT, records must carry the associated public key and signature of the agent who created it. This means every piece of data carries meta-information about where that data came from (who created it, and in what sequence on the their chain). Note: In other hash-chain based systems Holochain's "actions" are often called "headers," which link to the previous headers to create the chain. In Holochain, while the action does establish

[18] We use the term "grammatic" as a way to generalize from the usual understanding of grammar which is linguistic. Where grammar is often understood to be limited to language, grammatics points to the pattern of creating templates with classes of items that can fill slots in those templates. This pattern can be used for creating "grammars" of social interaction, "grammars" of physical structures (we would call Christopher Alexander's "A Pattern Language" for architecture an example of grammatics), and so on.

[19] A number of projects in the Holochain ecosystem are already exhibiting this characteristic of evolvability, such as The Weave / Moss (see https://theweave.social), Ad4m (https://ad4m.dev/), Memetic Activation Platform (see https://github.com/evomimic/we-all-map/wiki/MAP-Overview).

temporal order, its core function is to record an act of agency, that of "speaking" data into existence.

- **State:** State changes in Holochain are local (signed to a local *Source Chain*) and then information about having created a local state change is shared publicly on the DHT. This allows global visibility of local state changes, without a need to manage consensus about a global state, because there is truly no such thing as global state in a system that allows massive, simultaneous, decentralized change.
- **Time:** There is no global time nor global absolute sequence of events in Holochain either. No global time is needed for local state changes, and since each local change is stored in a hash chain, we get a clear, immutable, sequence of actions tagged with local timestamps. (Note: For apps that need some kind of time proof to interface with the outside world (e.g. token or certificate expiration timestamps) we plan to provide a time proof service that replaces the need for centrally trusted timeservers.)

System Architecture Overview

In Holochain every app defines a distinct, peer-to-peer, encrypted network where one set of rules is mutually enforced by all users. This network consists of the peers running the app, who participate in routing messages to each other and validating and storing redundant copies of the application's database.

Holochain operates different subsystems, each of which functions on separate workflows and change models. Even though Holochain functions as a common underlying database on the back-end, the workflows in each subsystem each have different input channels which trigger different transformational processes. Each workflow has distinct structural bottlenecks and security constraints, which necessitates that execution of workflows is parallelized across subsystems, and sometimes within a subsystem.

1. **Local Agent State:** Represented as changes to an agent's state by signing new records with their private key, and committing them to a local hash chain of their action history called a Source Chain. Initial chain genesis happens upon installation/activation, and all following changes result from "zome calls" into the app code.
2. **Global Visibility of Local State Changes:** After data has been signed to a Source Chain it gets published to a Graphing DHT (Distributed Hash Table) where it is validated by the peers who will store and serve it. The DHT is continually balanced and healed by gossip among the peers.
3. **Network Protocols:** Holochain instantiates the execution of app DNA on each node under the agency identified by the public key, transforming code into a collective networked organism. An agent's public key ***is*** their network address, and is used as the to/from target for remote zome calls, signals, publishing, and gossip. Holochain is transport-agnostic, and can operate on any network transport protocol which a node has installed for routing, bootstrapping, or proxying connections through NAT and firewalls.
4. **Distributed Application:** Apps are compiled and distributed into WebAssembly (WASM) code bundles which we call a DNA. Data integrity is enforced by the validation defined in an app's DNA, which is composed of data structures, functions, and callbacks packaged in Zomes (short for chromosome) which function as reusable modules. DNAs are coupled with an Agent's public key and activated or instantiated into a Cell. Installation and activation status of these bundles is managed by a runtime container.

Some notes on terminology

Biological Language We have chosen biological language to ground us in the pattern of collective distributed coherence that we observe in biological organisms. This is a pattern in which the agents that compose an organism (cells) all start with the same ground rules (DNA). Every agent has a copy of the rules that *all* the other agents are playing by, clearly identifying membership in the collective self based on matching DNA.

This is true of all Holochain DNAs, which can also be combined together to create a multi-DNA application (with each DNA functioning like a distinct micro-service in a more complex application). In a hApp bundle, each DNA file is the complete set of integrity zomes (WASM) and settings whose hash also becomes the first genesis entry in the agent's source chain. Therefore, if the DNA hash in your first chain record does not match mine, we are not cells of the same network organism. A "zome" is a code module, which functions as the basic compositional unit for assembling the complete set of an application's DNA.

When a DNA is instantiated along with a public/private key pair, it becomes a "cell" which is identified by the combination of the DNA hash and the public key.

Students of biology may recognize ways that our language doesn't fully mesh with their expectations. Please forgive any imprecision with understanding of our intent to build better language for the nature of distributed computing that more closely matches biology than typical mechanistic models.

The Conductor Much of the discussion below is from the perspective of a single DNA, which is the core unit in Holochain that provides a set of integrity guarantees for binding agents together into a single social context. However, Holochain can also be seen as micro-service provider, with each DNA providing one micro-service. From this perspective, a Holochain node is a running process that manages many connections to many DNAs simultaneously, from user interfaces initiating actions,

from other nodes sharing a subset of identical DNAs, and from cells within the same node sharing the same agent ID but bound to different DNAs. Thus, we call a Holochain node the **Conductor** as it manages the information flows from "outside" (UI calls and calls from other local cells) and from "inside" (network interactions) as they flow into and out of the may DNA instances running code. This term was chosen as it suggests the feel of musical coordination of a group, as well as the conduit of an electrical flow. Please see the Implementation Spec (Appendix A) for a more detailed on how a complete Holochain Conductor must be built.

Integrity Guarantees

Within the context of the Basic Assumptions and the System Architecture both described above, the Holochain system makes the following specific integrity guarantees for a given Holochain DNA and network:

1. **State:** Agents' actions are unambiguously ordered from any given action back to genesis, unforgeable, non-repudiable, and immutable (accomplished via local hash chains called a Source Chain, because all data within the network is sourced from these chains.)
2. **Self-Validating Data:** Because all DHT data is stored at the hash of its content, if the data returned from a request does not hash to the address you requested, you know you've received altered data.
3. **Self-Validating Keys:** Agents declare their address on the network as their public key, and key rotation is subject to rules defined by the agent and enforced by their peers. Peers can confirm any published data or remote call is valid by checking the signature using the from address as the public key.
4. **Termination of Execution:** No node can be coerced into infinite loops by non-terminating application code in either remote zome call or validation callbacks. Holochain uses WASM metering to guarantee a maximum execution budget to address the the Halting Problem.
5. **Deterministic Validation:** Ensure that only deterministic behaviors (ones that will always get the same result no matter who calls them on what computer) are available in validation functions. An interim result of "missing dependency" is also acceptable, but final evaluation of valid/invalid status for each datum must be consistent across all nodes and all time spans.
6. **Strong Eventual Consistency:** Despite network partitions, all nodes who are authorities for a given DHT address (or become one at any point) will eventually converge to the same state for data at that address. This is ensured by the DHT functioning as a conflict-free replicated data type (CRDT).
7. **"0 of N" Trust Model:** Holochain is immune to "majority attacks" because any node can always validate data for themselves independent of what any other nodes say.[20]
8. **Data Model Scalability:** Because of the overlapping sharding scheme of DHT storage and validation, the total computing power and overall throughput for an application scales linearly as more users join the app.
9. **Atomic Zome Calls:** Multiple writes in a single zome call will all be committed in a single SQL transaction or all fail together. If they fail the zome call, they will report an error to the caller and the writes will be rolled back.

Source Chain: Formal State Model

Data in a Holochain application is created by agents changing their local state. This state is stored as an append-only hash chain. Only state changes originated by that agent (or state changes that they are party to in a multi-agent action) are stored to their chain. Source Chains are NOT a representation of global state or changes that others are originating, but only a sequential history of local state changes authored by one agent.

The structure of a Source Chain is that of a hash chain which uses headers (called "actions" in Holochain terms) to connect a series of entries. Each record in the chain is a two-element tuple, containing the action and the entry (if applicable for the action type).

Since the action contains the prior action hash and current entry hash (if applicable), each record is a tamper-proof atomic data element. Additionally, in practice a record is always transmitted along with a signature on the action's hash, signed by the private complement of the public key in the action. This means that anyone can hash the entry content to make sure it hasn't been tampered with, and they can hash the action data and compare the accompanying signature on that hash to ensure it matches the author's public key. The action's chain sequence and monotonic timestamp properties provide further immutable reinforcement of logical chain ordering.

Data in Holochain is kept in Content Addressable Stores which are key-value stores where the key is the hash of the content. This makes all content self-validating, whether served locally or remotely over the DHT. Data can be retrieved by the action hash (synonymous with record hash) or the entry hash.

The code that comprises a Holochain application is categorized into two different types of zomes:

1. **Integrity Zomes** which provide the immutable portion of the app's code that:
   - identifies the types of entries and links that may be committed in the app,

[20] See this Levels of Trust Diagram https://miro.medium.com/max/1248/0*k3o00pQovnOWRwtA.

- defines the structure of data entries, and
- defines the validation code each node runs for each type of operation that intends to add to state at a given DHT address.

2. **Coordinator Zomes**, the set of which can be removed from or added to while an app is live, and which contain various create, read, update, and delete (CRUD) operations for entries and links, functions related to following graph links and querying collections of data on the DHT, and any auxillary functionality someone wants to bundle in their application.

Each application running on Holochain is uniquely identified by a DNA hash of the integrity zome code, after being compiled to Web Assembly (WASM) and bundled with additional settings and properties required for that app.

Application Note: Multiple DNA-level apps can be bundled together like interoperating micro-services in a larger Holochain Application (hApp), but the locus of data integrity and enforcement remains at the single DNA level, so we will stay focused on that within this document.

There are three main types of Zome functions:

1. (z_f) zome functions which do not alter state.
2. (Z_f) that can be called to produce state changes, as well as the
3. Validation Rules (V_R) for enforcing data integrity of any such state changes (additions, modifications, or deletions of data).

$$z_{f_1} \dots z_{f_x} \in \text{Coordinator Zomes}$$
$$Z_{f_1} \dots Z_{f_x} \in \text{Coordinator Zomes}$$
$$V_{R_1} \dots V_{R_x} \in \text{Integrity Zomes}$$

Note about Functions: Most functionality does not need to be in the immutable, mutually enforced rules included in the DNA hash (Integrity Zomes); only the functionality which validates data ((V_R)) does. In practice, including code that does not contribute to data validation ((z_f), (Z_f)) in the integrity zome creates a brittle DNA that is difficult to update when bugs are repaired or functionality needs to be introduced or retired.

The first record in each agent's source chain contains the DNA hash. This initial record is what demonstrates that each agent possessed, at installation time, identical and complete copies of the the rules by which they intend to manage and mutually enforce all state changes. If a source chain begins with a different DNA hash, then its agent is in a different network playing by a different set of rules.

Genesis: The genesis process for each agent creates three initial entries.

1. The hash of the DNA is stored in the first chain record with action C_0 like this:

$$C_0 = WASM \begin{Bmatrix} a_{DNA} \\ e_{DNA} \end{Bmatrix}$$

2. Followed by a "Membrane Proof" which other nodes can use to validate whether the agent is allowed to join the application network. It can be left empty if the application membrane is completely open and it doesn't check or use proofs of membership.

$$C_1 = \begin{Bmatrix} a_{mp} \\ e_{mp} \end{Bmatrix}$$

3. And finally the agent's Public Key that they have generated, which also becomes their address on the network and DHT. Keys are the only entry type for which the hash algorithm is equality (meaning the hash of a key is the key itself, so it cannot contain any content other than the public key).

$$C_2 = \begin{Bmatrix} a_K \\ e_K \end{Bmatrix}$$

Initialization: After genesis, DNAs may have also provided initialization functions which are all executed the first time an inbound zome call is received and run. This delay in initialization is to allow time for the application to have joined and been validated into the network, just in case initialization functions may need to retrieve some data from the network.

Initialization functions may write entries to the chain, send messages, or perform any variety of actions, but after all coordinator zomes' initialization functions (according to the order they were bundled together) have successfully completed their initializations, an InitZomesComplete action is written to the source chain, so that it will not re-attempt initialization, thus preventing any redundant side-effects.

Ongoing Operation via Calls to Zome Functions: All changes following genesis and initialization occur by Zome call to a function contained in a Coordinator Zome in the following form:

$$Z_c = \{Z_f, Params, CapTokenSecret\}$$

Where Z_f is the Zome function being called, $Params$ are the parameters passed to that Zome function, and $CapTokenSecret$ references the capability token which explicitly grants the calling agent the permission to call that function.

Based on the interface connection and state when the Zome call is received we construct a context which provides additional necessary parameters to validate state transformation:

$$Context(Z_c) = \{Provenance, C_n\}$$

Provenance contains the public key of the caller along with their cryptographic signature of the call as proof that it originated from the agent controlling the associated private key.

C_n is the Source Chain's latest action at the time we begin processing the zome call. The Zome call sees (and potentially builds upon) a snapshot of this state through its lifetime, and validation functions will all be called "as at" this state. Since multiple simultaneous zome calls might be made, tracking the "as at" enables detection of another call having successfully changed the state of the chain before this call completed its execution, at which point any actions built upon the now-obsolete state may need to be reapplied to and validated on the new state.

Zome Calls & Changing Local State First, Holochain's "subconscious" security system confirms the *CapTokenSecret* permits the agent identified by the *Provenance* to call the targeted function. It returns a failure if not. Otherwise it proceeds to further check if the function was explicitly permitted by the referenced capability token.

Note on Permissions: Capability tokens function similarly to API keys. Cap token grants are explicitly saved as private entries on the granting agent's source chain and contain a secret used to call them. Cap token claims containing the secret are saved on the calling agent's chain so they can be used later to make calls that execute the capabilities that have been granted.

If the Zome call is one which alters local state (distinct from a call that just reads from the chain or DHT state), we must construct a bundle of state changes that will attempt to be appended to the source chain in an atomic commit:

$$\Delta_C(C_n, Z_c) = \begin{Bmatrix} a_{\prime} & a_{\prime\prime} & \dots & a_x \\ e_{\prime} & e_{\prime\prime} & \dots & e_x \end{Bmatrix}$$

where a Chain is composed of paired actions, a_x, and entries, e_x.

The next chain state is obtained by appending the changes produced by a zome call to the state of the chain at that point.

$$C_n = C_n + \Delta_C(C_n, Z_c)$$

If the validation rules pass for these state changes **and the current top of chain is still in state** C_n then the transaction is committed to the persistent store, and the chain is updated as follows:

$$C_n = \begin{Bmatrix} a_{DNA} & \dots & a_n \\ e_{DNA} & \dots & e_n \end{Bmatrix}$$

If the validation rules fail, the deltas will be rejected with an error. Also, if the chain state has changed from C_n, we can:

1. return an error (e.g. "Chain head has moved"),
2. commit anyway, restarting the validation process at a new "as at" C'_n if the commit is identified as "stateless" in terms of validation dependencies (e.g., a tweet generally isn't valid or invalid because of prior tweets/state changes). We refer to any application entry types that can be committed this way as allowing "relaxed chain ordering".

Note about Action/Entry Pairs: This paired structure of the source chain holds true for all application data. However, certain action types defined by the system, whose entry payloads are small or require metadata that is additional to primary entry content, integrate what would be entry content as ***additional fields inside the action*** *instead of creating a separate entry which would add unnecessary gossip on the DHT. These types are identified and described in Appendix A, Implementation.*

Countersigning

So far we have discussed individual agents taking Actions and recording them on their Source Chains. It is also desireable for subsets of agents to mutually consent to a single Action by atomically recording the Action to their chains. We achieve this through a process of Countersigning, whereby a session is initiated during which the subset of agents builds an Action that all participating agents sign, and during which all agents promise one another that they will not take some other action in the meantime.

There are two ways of managing the countersigning process:

1. Assigned completion: where one preselected agent (whom we call the Enzyme) acts as a coordinator for ensuring completion of a signing session.
2. Randomized completion: where any agent in the neighborhood of the Entry address (which is cryptographically pseudorandom and is computed on data contributed by each counterparty) can report completion.

Additionally there are two contexts for making these atomic changes across multiple chains:

1. When the change is about parties who are accountable to the change, i.e., their role is structurally part of the state change, as in spender/receiver of a mutual credit transaction
2. When the change simply requires witnessing by M of N parties, i.e., all that's needed is a "majority" of a group to agree on the atomicity. This allows a kind of "micro-consensus" to be implemented in parts of

an application. It's an affordance for applications to implement a set of "super-nodes" that manage a small bit of consensus. Note that in our current implementation, M of N countersigning always uses an Enzyme to manage the session completion.

Countersigning Constraints

1. All actions must be signed together; one action is not enough to validate an atomic state change across multiple chains.
   - All parties must be able to confirm the validity of each others's participation in that state change (meaning each chain is in a valid state to participate in the role/capacity which they are engaging – e.g., a spender has the credits they're spending at that point in their chain).
2. The moment the enzyme or random session completer agent holds and broadcasts all the signed and valid actions, then everyone is committed.
3. It should not be possible for a participant to withhold and/or corrupt data and damage/fork/freeze another participant's source chain.
4. It should not require many network fetches to calculate state changes based on countersignatures (i.e., it should be possible to `get` a unified logical unit – that is, multiple actions on a single entry hash address on the DHT).
5. Participants can NOT move their chain forward without a provable completion of the process, and there IS a completion of the process in a reasonable time frame
   - The countersigning process should work as closely as possible to the standard single-agent "agent-centric network rejection of unwanted realities": anyone who moves forward before the process has timed out or completed, or anyone who tries to submit completion outside of timeouts, will be detected as a bad fork.

Countersigning Flow Here is a high-level summary of how a countersigning session flows:

0. Alice sends a **preflight request** to Bob, Carol, etc, via a remote call.
   - The preflight request includes all information required to negotiate the session with the entry itself, for example:
     - Entry hash: What data are we agreeing to countersign? (The contents of the entry are often negotiated beforehand and communicated to all parties separately, although the app data field described below can also be used for this purpose.)
     - Action base: What type is the entry we'll be countersigning, and will it be a Create or an Update?
     - Update/delete references: what are we agreeing to modify?
     - Session times: Will I be able to accept the session start time, or will it cause my chain to be invalid? Am I willing to freeze my chain for this long?
     - The agents and roles: Are these the parties I expected to be signing with?
     - App data: can point to necessary dependencies or, if the contents of the entry to be countersigned are small, the entry itself.
1. If the other parties accept, they freeze their chains and each return a **preflight response** to Alice. It contains:
   - The original request.
   - The state of the party's source chain "as at" the time they froze it.
   - Their signature on the above two fields.
2. Alice builds a session data package that contains the preflight request along with the source chain states and signatures of all consenting parties, and sends it to them.
3. Each party builds and commits an action that writes the countersigned entry (including the contents of the session data package and the entry data itself) to their source chains. At this point, unsigned actions are created for themselves and every other party and full record validation is run against each action, as though they were authoring as that agent.
4. After everything validates, each agent signs and sends their action to the session completer – either the enzyme (if one was elected) or the entry's DHT neighborhood.
5. The session completer reveals all the signed actions as a complete set, sending it back to all parties.
6. Each signer can check for themselves that the set is valid simply by comparing against the session entry and preflight info. They do not have to rerun validation; they only need to check signatures, integrity, and completeness of the action set data.
7. All counterparties now proceed to write the completed action to their source chain and publish its data to the DHT.
8. The DHT authorities validate and store the action and entry data as normal.

Graph DHT: Formal State Model

Holochain performs a topological transform on the set of the various agents' source chains into a content-addressable graph database (graph DHT or GDHT) sharded across many nodes who each function as authoritative sources for retrieving certain data.

Fundamental Write Constraint: The DHT can never be "written" to directly. All data enters the DHT **only** by having been committed to an agent's source chain and then being **transformed** from validated local chain state into the elements (DHT operations) required for GDHT representation and lookup.

Structure of GDHT data: The DHT is a content-addressable space where each piece of content is found at the address which is the hash of its content. In addition,

any address in the DHT can have metadata attached to it. This metadata is not part of the content being hashed.

Note about hashing: Holochain uses 256-bit Blake2b hashes with the exception of one entry type, AgentPubKey, which is a 256-bit Ed25519 public key and its hash function is simply the identity function. In other words, ***the content of the AgentPubKey is identical to its hash****. This preserves content-addressability but also enables agent keys to function as self-proving identifiers for network transport and cryptographic functions like signing or encryption.*

DHT Addresses: Both Actions and Entries from source chains can be retrieved from the DHT by either the ActionHash or EntryHash. The DHT `get()` function call returns a Record, a tuple containing the most relevant action/entry pair. Structurally, Actions "contain" their referenced entries so that pairing is obvious when a Record is retrieved by ActionHash. However, Actions are also attached as metadata at an EntryHash, and there could be many Actions which have created the same Entry content. A `get()` function called by EntryHash returns the oldest undeleted Action in the pair, while a `get_details()` function call on an EntryHash returns all of the Actions.

Agent Addresses & Agent Activity: Technically an AgentPubKey functions as both a content address (which is never really used because performing a `get()` on the key just returns the key itself) and a network address to send communications to that agents. But in addition to the content of the key stored on the DHT is metadata about that agent's chain activity. In other words, a `get_agent_activity()` request retrieves metadata about their chain records and chain status.

Formally, the entire GDHT is represented as a set of 'basis hashes' b_{c_x}, or addresses where both content c and metadata m may be stored:

$$GDHT = \{d_1, \ldots, d_n\}$$

The data at a basis hash can consist of content and/or metadata:

$$d_{b_{c_x}} = (c_x, M)$$

A basis hash is the hash of the content stored at the address:

$$b_{c_x} = hash(c_x)$$

The total set of content represented by the GDHT consists of entries E, actions A, and external content T (where the addresses can still store metadata and be used as references, but the content is not stored in the DHT):

$$E = \{e_1, \ldots, e_n\}$$
$$A = \{a_1, \ldots, a_n\}$$
$$T = \{t_1, \ldots, t_n\}$$
$$C = E \bigsqcup A \bigsqcup T$$

An address can hold a set of metadata:

$$M = \{m_1, \ldots, m_n\}$$
$$m_x = \text{metadata}$$

There may be arbitrary types of metadata. For instance, every instance of entry content e has a set of creation actions A_e associated with it:

$$\forall e \; M_{context} = \{a_{e1}, \ldots, a_{en}\}$$

And any address may have a set of links pointing to other addresses, each of which is a tuple of its type, an arbitrary tag, and a reference to the target address b_{c_T}:

$$M_{link} = \{link_1, \ldots, link_n\}$$
$$\exists c_T \; link = (type, tag, b_{c_T})$$

For links, we refer to an address with link metadata as a **Base** and the address that the link points to as a **Target**. The link can also be typed and have an optional *tag* containing arbitrary content.

Topological Transform Operations: A source chain is a hash chain of actions with entries, but these are transformed into DHT operations which ask DHT nodes to perform certain validation and storage tasks on the content and metadata at the address, because we are transforming or projecting from authorship history to a distributed graph. Chain entries and actions are straightforwardly stored in the graph as nodes, as C at their hash in the DHT, but more sophisticated operations are also performed on existing DHT entries. For example, when updating/deleting entries, or adding/removing links, additional metadata is registered in parts of the DHT to properly weave a graph.

Graph Transformation

While source chain entries and actions contain all the information needed to construct a graphing DHT, the data must be restructured from a linear local chain under single authority and location, to a graph across many nodes (where a node is an address or hash, optionally with content) with many authorities taking responsibility for redundantly storing content and metadata for the entire range of nodes. In this section we focus only on

the transformation from source chain to DHT. The next section will focus on the election of authoritative sources for data.

The linking/graphing aspects must be constructed from the state changes committed to source chains.

The process from an agent's action to changed DHT state is as follows:

1. An action produces a **source chain record** detailing the nature of the action, including the context in which it was taken (author and current source chain state).
2. The source chain record is transformed to **DHT operations**, each of which has a **basis hash** that it applies to.
3. The author sends these DHT operations to the respective neighborhoods of their basis hashes, where peers who have assumed authority for the basis hashes **integrate** them into an updated state for the data at those basis hashes.

The following table shows how each action (which gets stored on the author's source chain as a record) is transformed into multiple DHT operations. Remember an operation corresponds with a way that the DHT state needs to be manipulated.

For viable eventual consistency in a gossipped DHT, all actions must be idempotent (where a second application of an operation will not result in a changed state) and additive/monotonic:

- The deletion of an entry creation action and its corresponding entry doesn't actually delete the entry; it *marks* the action as deleted. At the entry basis hash, the delete action becomes part of a CRDT-style "tombstone set", and a set difference is taken between the entry creation actions A_c and entry deletion actions A_d that reference at the entry's basis hash to determine which creation actions are still 'live' ($A_{c_l} = A_c - A_d$). Eventually the entry itself is considered deleted when $A_c - A_d = \varnothing$.
- The removal of a link adds the removal action to a tombstone set at the link's base address in a similar fashion, subtracting the link removal actions from the link creation actions they reference to determine the set of live links.
- Updating an entry creation action and its corresponding entry doesn't change the content in place; it adds a link to the original action and entry pointing to their replacements. One entry creation action may validly have many updates, which may or may not be seen by the application's logic as a conflict in need of resolution.

The transformation of an action is followed by sending the operation to specific DHT basis hashes, instructing the agents claiming authority for a range of address space covering those basis hashes, to validate and store (integrate) the operations into their respective portions of the DHT store. Because the DHT is a **graph** database, what is added is either a node or an edge. A node is a basis hash in the DHT, while an edge is part of the addressable content or metadata stored at a node.

Here is a legend of labels and symbols used in the diagrams:

- The large, gray, rounded rectangle on the left of each row represents the agent k currently making an action, and encompasses the data they produce.
- A label styled as do_x() is the function representing the action being taken by the agent k. It yields a record of the action, which is saved to the source chain.
- a_n is the action that records the action. It is represented by a square.
- a_{n-1} is the action immediately preceding the action currently being recorded.
- E is action-specific data which is contained in a separate *entry* which has its own home in the DHT. It is represented by a circle.
- $e : \{\dots\}$ is action-specific data which performs an operation on prior content. Such data exists wholly within the action of the record of the action.
- Overlapping shapes (primarily square actions and circular entries) represent data that travels together and can be seen as a single unit for the purpose of defining what exists at a given basis hash. In the case of an entry basis hash, where multiple actions authoring the same entry may exist, each entry/action pair can be seen as its own unit, or alternatively the content at that address can be seen as a superposition of multiple entry/action pairs.
- k is the public key of the agent taking an action.
- $\rightarrow$ is a graph edge pointing to the hash of other content on the DHT.
- C_B and C_T are a link base and target, the basis hashes of previously existing content. Any addressable content can be the base and target of a link. These are represented by blobs.
- Blue arrows are graph edges.
- a_p and E_p are the previously existing content which a graph edge $\rightarrow$ references, when the reference may *only* pertain to a action or an entry, respectively.
- A label styled as RegisterX is a DHT operation that adds metadata to a basis hash. A label styled as StoreX is a DHT transform that adds addressable content to a DHT basis hash. The payload of an operation is contained in a gray triangle.
- Basis hashes are represented as b_x in black circles, in which the subscript x represents the kind of addressable content stored at that basis hash. For instance, b_k is the basis hash of the author k's agent ID entry; that is, their public key.
- A stack of rounded rectangles represents the neighborhood of the basis hash being manipulated, in which multiple peers may be assuming authority for the same hash.

- Gray arrows represent the transformation or movement of data.
- Data attached to a basis hash by a line is metadata, while data overlapping a basis hash is primary content.
- A green slash indicates existing data that has been replaced by an update. A green arrow leads from the update action to the data it replaces.
- A red X indicates existing data that has been *tombstoned*; that is. it is marked as dead. A red arrow leads from the delete action to the data it tombstones.

Authority Election In the case of source chain entries (and actions), it is fairly obvious that the author who created them is the **author**itative source. But part of translating from a series of local chain states to a resilient global data store involves identifying which nodes in the network become the responsible authorities for holding which DHT content.

Most existing DHT frameworks simply have nodes volunteer to hold specific content, and then use the DHT as a tracking layer to map content hashes to the nodes holding the content. But this allows content to disappear arbitrarily and also creates imbalanced dynamics between people who consume content and people who serve it (leechers & seeders). Since Holochain is designed to function more like a distributed database than a content distribution network, it needs to ensure resilience and permanence of data elements, as well as load balancing and reasonable performance, on a network where nodes are likely coming online and going offline frequently.

As such, Holochain doesn't rely on nodes to volunteer to hold specific entries, but rather to volunteer aggregate capacity (e.g., holding 100MB of data rather than arbitrarily chosen entries). Authoring nodes are responsible for publishing entries from their local DHT instance to other nodes on the network (**authorities**) who will become responsible for serving that data.

Like most DHT architectures, Holochain uses a "nearness" algorithm to compute the "distance" between the key of a piece of data and the key of a peer holding the data; in our case, between the 256-bit Blake2b basis hash of the data or metadata to be stored and the 256-bit Ed25519 public key (network address) of nodes. Basically, it is the responsibility of the nodes *nearest* a basis hash to store data and metadata for it, within an "arc" of authority of their choosing.

Holochain's validating, graphing, gossiping DHT implementation is called **rrDHT**.

rrDHT is designed with a few performance requirements/characteristics in mind.

1. It must have a compact and computationally simple representational model for identifying which nodes are responsible for which content, and which nodes actually hold which content. (A **"world model"** of what is where.)
2. It must have **lookup speeds** at least as fast as Kademlia's binary trees ($\mathcal{O}(n \log n)$). Current testing shows an average of 3 hops/queries to reach an authority with the data.
3. It must be **adjustable** to be both resilient and performant across many DHT compositional make-ups (reliability of nodes, different network topologies, high/low usage volumes, etc.)

World Model: The network location space is a circle comprising the range of unsigned 32-bit numbers, in which the location 0 is adjacent to the location $2^{32} - 1$. It can

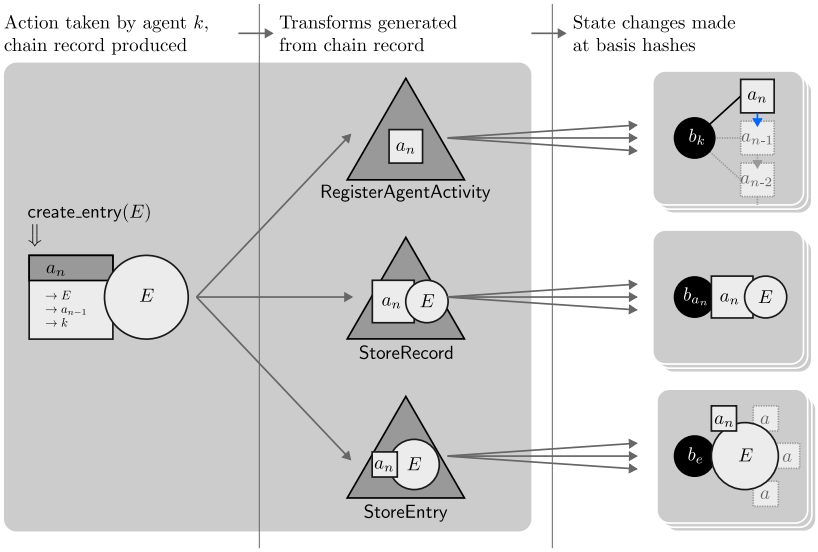


FIG. 1. Operations and state changes produced by `create` action

Text is entered as Latex on the 'latex' layer, then you run the Extensions
Note: text sizes and locations can be changed on the rendered Latex layer

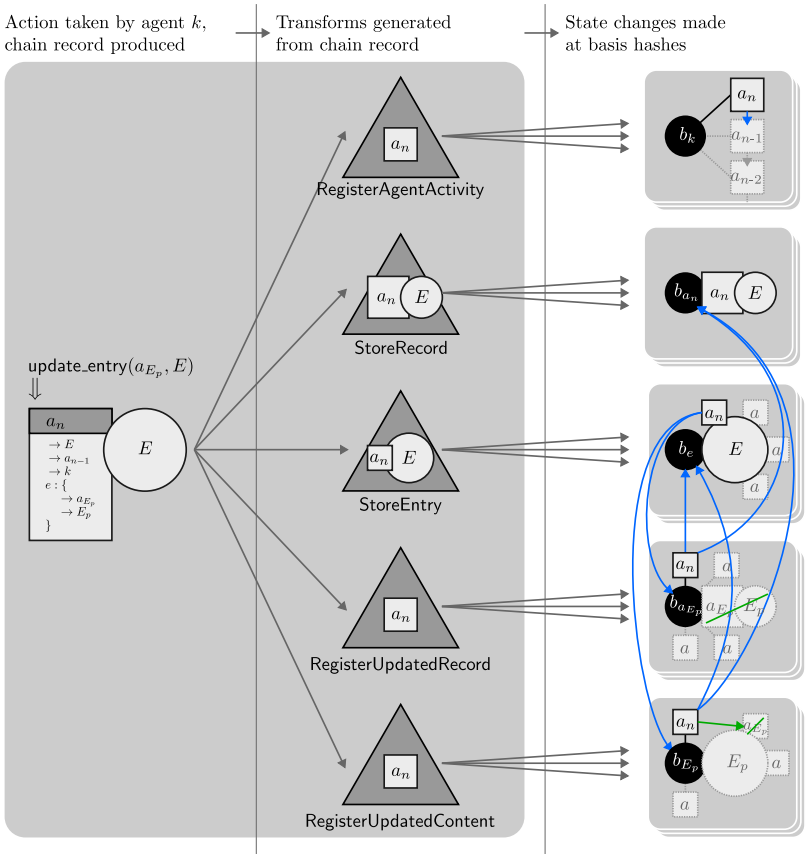


FIG. 2. Operations and state changes produced by `update` action

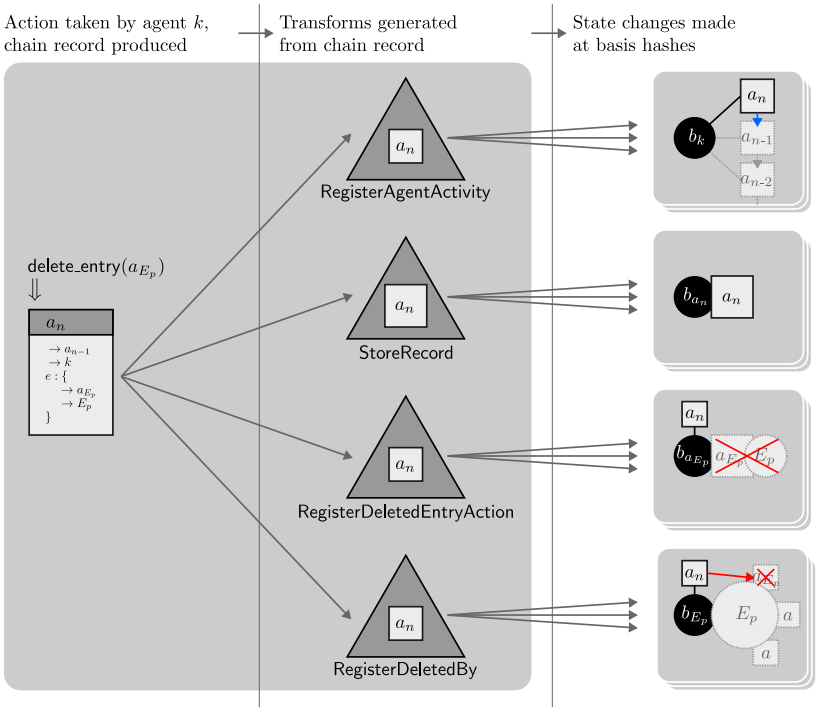


FIG. 3. Operations and state changes produced by `delete` action

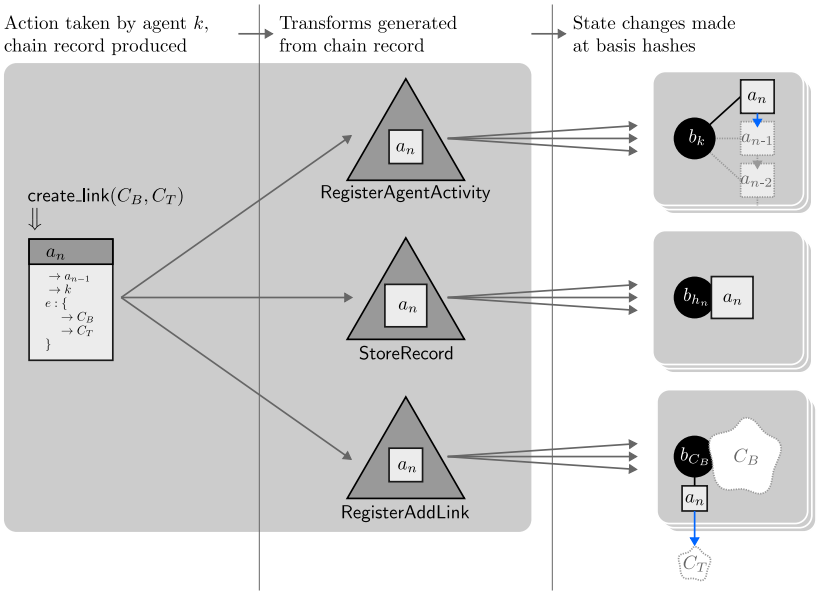


FIG. 4. Operations and state changes produced by `create_link` action

be more precisely defined as:

$$L : \mathbb{Z} \mod 2^{32}$$

Defining this in terms of modulo arithmetic has an important consequence for routing a publish or query request to the correct agent, which we will explain later.

The larger 256-bit address space of the DHT, consisting of 256-bit "basis hashes" B (Blake2b-256 hashes of addressable content C, which as previously defined includes Ed25519 public keys of agents K), is mapped to the smaller network location space via a function:

$$\mathsf{map\_to\_loc} : B \to L$$

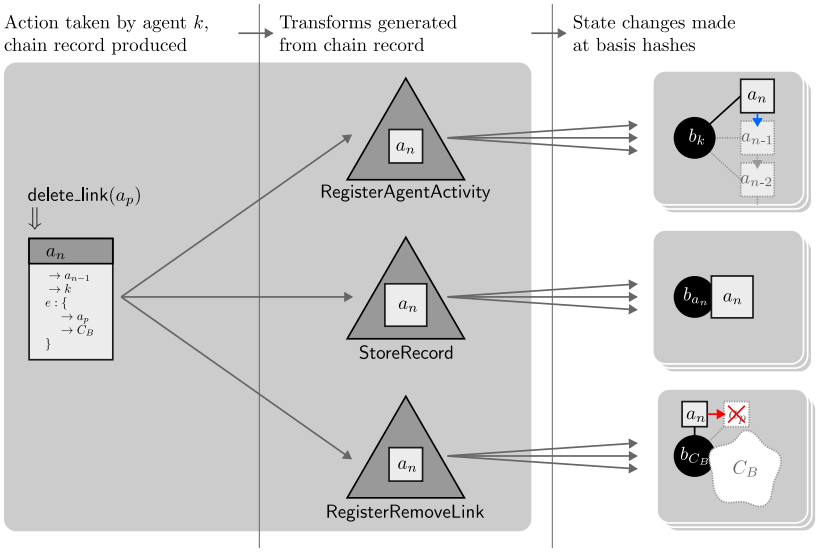


FIG. 5. Operations and state changes produced by `delete_link` action

which is the XOR of 8 × 32-bit segments of the hashes. At the storage level, the original address is still used for addressing content and metadata, so collisions in the smaller space are not a concern.

Using the sets B and L to denote all basis addresses and network locations in the DHT, respectively:

$$|L| = 2^{32}$$
$$|B| = 2^{256}$$
$$\therefore |B| = |L| \cdot 2^{256-32}$$

Each agent has a network location l_k in this 32-bit space as well as an arc size s_{arc} indicating how large an arc of the location circle they are claiming authority for. The storage arc ARC_{l_k} defines the range of basis hashes for which a node claims authority. The arc spreads clockwise from l_k.

$$ARC_{l_k} : \{l_k, \ldots, l_k + s_{arc}\}$$

As a consequence of modulo arithmetic, agents close to 2^{32} may end up claiming authority for data at and beyond 0; for example, if the network location for an agent's public key k is $2^{32} - 2$ and their arc of authority is 20, the arc extends to network location 18:

$$(2^{32} - 2 + 20) \mod 2^{32} = 18$$

A node can rapidly resolve any basis hash b to the most likely candidate for an authority p_{best} by comparing the basis hash's network location l_b to all the peers they know about (the set L_P, or their "peer table") using the following algorithm (expressed in pseudocode):

```
p_best = L_P
    .sort_ascending_by(l_p -> (l_b - l_p) mod 2^32)
    .first()
```

It is then determined whether the peer is indeed an authority for l_b, either by relying on locally cached knowledge of their arc or asking them directly. At this point, if the peer is determined to not claim authority, the next less likely candidate may be chosen, on the hope that their arc is larger, or the most likely candidate is asked if they know of a *more* likely candidate. They are in an advantageous place to do so, as agents' peer tables are naturally biased toward peers that are near to them in the network location space.

Network Location Quantization Additionally, arcs are subjected to **quantization** which splits the network location space L into disjoint subsets of a given size s_q, and to which the starting arc boundary k and arc size s_{arc} are also snapped. The quantized arc is then fully represented by three numbers: the quantized chunk size s_q, the number of chunks until the start boundary k_q, and the number of chunks from start to end n_q.

Peers also quantize the time dimension such that the size of chunks of time increase quadratically as the dimension extends into the past.

The spaces of network locations and time form two dimensions of a coordinate space, and each operation can be mapped to a point in this space using the network location of its basis hash as the x coordinate and its authoring time as the y coordinate.

When the coordinate space is quantized, it forms a grid. Each agent holds a finite region of this grid, bounded by their quantized arc, and the total set of held operations within each grid cell is fingerprinted using a lossy algorithm (such as the XOR of the hashes of all the operations whose coordinates fall within the cell).

When two peers attempt to synchronize the held sets of operations for the intersection of their two address spaces $ARC_{l_{k_a}} \cap ARC_{l_{k_b}}$, they can then simply compare their respective fingerprints of each cell within that intersection. If the fingerprints do not match, they exchange and compare the entire list of operation hashes they each hold. This allows peers to more quickly compare and synchronize regions of shared authority, and the quadratic nature of quantum sizes in the time dimension allows them to prioritize syncing of newer, more rapidly changing data, by comparing more fingerprints from smaller time regions for newer data, and fewer fingerprints over larger time regions for older data.

DHT Communication Protocols So far we have described the topological transformation between the agentic holding of state on a source chain into the shape of the shared data for making that state visible via the DHT. Additionally we have described an addressing scheme that makes data deterministically retrievable. Now we must describe the communication protocols that allow agents to come and go on the network while still maintaining the necessary data redundancy.

Peers conduct all communication with each other using **messages** of various classes and types. There are two levels of abstraction for messages; the lower level establishes peer connections in discrete **network spaces** and defines basic messages for maintaining DHT state and sending arbitrary application messages, while the higher level adapts these basic message types to implement Holochain-specific features.

There are two classes of messages, both of which are non-blocking; that is, they are sent asynchronously and don't monopolize the peer connection while waiting for a response:

- **Notify** messages are "fire-and-forget"; that is, they don't anticipate a response from the receiver.
- **Request** messages are wrapped in an 'envelope' that has a sequence ID, and anticipate a corresponding **response** message with the same sequence ID from the receiver.

Basic Message Types These message types exist at the lower level.

- **Notify** message types
  - **Broadcast** sends a message of one of the following types:
    - **User** contains arbitrary, application-level data. Here, the application in question is Holochain rather than a specific hApp.
    - **AgentInfo** advertises an agent's current storage arc and network transport addresses.
    - **Publish** advertises that one or more DHT operations are available for retrieval. An arbitrary **context** value indicates the publishing context, which in practice is a bit field that indicates whether it's being published as part of a countersigning session and whether a validation receipt is needed.
  - **DelegateBroadcast** sends a broadcast, but rather than expecting the receiver to do something with it, it expects them to broadcast it in turn to the peers in their DHT neighborhood.
  - **FetchOp** requests the data for one or more DHT operations, usually as a follow-up from receiving a **Publish** broadcast message or **MissingOpHashes** gossip message advertising that such operations are available. While it is strictly a notify-class message, it functions similarly to a request-class message in that it anticipates a response in the form of a **PushOpData** message.
  - **PeerUnsolicited** is similar to **PeerQueryResp** below, but is initiated by a node without being prompted.
  - **PushOpData** sends the data for one or more DHT operations as a response to a **FetchOp** message. Each op optionally includes the quantized region it belongs to if it's being pushed as part of a historical sync.
  - **Gossip** is a container for messages implementing various gossip strategies among nodes who share authority for portions of the DHT's network location space.
- **Request** message types
  - **Call** and **CallResp** allow a peer to make an arbitrary, application-level function call to another peer and receive data in response. As with broadcast, the application in question is Holochain.
  - **PeerGet** and **PeerGetResp** allow a peer to ask another peer if they know about a specific agent. The response contains the same data as an **AgentInfo** message.
  - **PeerQuery** and **PeerQueryResp** allow a peer to ask another peer if they know of any agents who are currently claiming authority for a given 32-bit network location. The response contains zero or more **AgentInfo**s.

DHT data is synchronized between peers two stages:

1. A node sends a peer the hashes of the DHT operations they have available. This can happen via publish, where the initiator is creating new operations, or via gossip, where the initiator and remote peer engage in one or more rounds of comparing the operations they respectively hold for a shared arc of the network location space.
2. The remote peer 'fetches' the data for the operations they need but do not have.

Holochain-Specific Message Types The following Holochain-specific message types are implemented using the preceding basic message types. Unless otherwise

noted, the following messages follow a call-and-response pattern using **Call** and **CallResp**.

- An agent uses **CallRemote** to attempt a remote procedure call (RPC) to a zome function in another peer's cell.
- When an authority has finished validating DHT operations as a consequence of receiving a publish message, they send a **ValidationReceipts** message to the publisher. This tells the publisher that the authority has received the data, deemed it to be valid, and is now holding it for serving to other peers. This message uses the **User** broadcast message type.
- **Get** requests the addressable content stored at the given basis hash.
- **GetMeta** requests all metadata stored at the given basis hash.
- **GetLinks** requests only link metadata of a certain type at the given basis hash, optionally with a filter predicate.
- **CountLinks** is similar to GetLinks, but only requests the count of all links matching the type and filter predicate.
- **GetAgentActivity** requests all or a portion of the 'agent activity' metadata for the given agent ID, which includes source chain actions, chain status (whether it has been forked), and any outstanding warrants collected for that agent (see the following section for a description of warrants).
- **MustGetAgentActivity** requests only the portion of the agent activity metadata that can be guaranteed to be unchanging (if it exists) regardless of the current state at the agent's basis hash — that is, a contiguous sequence of source chain actions, notwithstanding any contiguous sequence that may exist in a fork of that agent's chain.
- There are three message types used in negotiating a countersigning session, all of which use the **User** broadcast message:
  - Counterparties use **CountersigningSessionNegotiation**, with a subtype of **EnzymePush**, to send their signed **Create** or **Update** action to the designated facilitator of the session (the Enzyme) when such an agent has been elected.
  - When an Enzyme has not been elected, counterparties instead use **PublishCountersign** to send their action to the neighborhood of the basis hash of the **StoreEntry** DHT operation that they will eventually produce if countersigning succeeds.
  - When authorities have received a PublishCountersign message from all expected counterparties, they then send the complete list of signed actions to all parties using **CountersigningSessionNegotiation** with a subtype of **AuthorityResponse**.

Fast Push vs. Slow Heal It is important to underscore the dual way in which data is propagated around the DHT, and the rationale for designing Holochain in such a way.

When data is initially created with the intention of persisting it in the DHT, it is sent to the neighborhoods of the appropriate authorities using a **fast push** strategy. This is the **Publish** broadcast message described above, in which the creator of the data takes responsibility for making sure it reaches a sufficient number of authorities to ensure resilience and availability of the data. The creator then attempts to re-send the Publish message to more authorities until they have received a satisfactory number of **ValidationReceipts** in response. (In practice, the publisher uses a combination of **Broadcast** and **DelegateBroadcast**, the latter message type reducing the burden on the publisher, who is unlikely to know of as many peers in the DHT operation's neighborhood as the authorities do, and who may intend to go offline before they have received a satisfactory number of validation receipts.)

After data has been created and has 'saturated' the neighborhood of the data's basis hashes, however, ongoing maintenance is required to keep the data alive as authorities leave and join the network. This is done using a **slow heal** strategy, in which authorities in the same neighborhood regularly initiate **gossip rounds**[21] to check each other's stores for new data.

Additionally, gossip is split into **recent** and **historical** gossip, wherein peers attempt to sync data that is younger than a certain threshold (for instance, five minutes) using a diffing strategy (a Bloom filter) that results in fewer unnecessary deltas being transferred, while data that is older than this threshold can afford to use a strategy with more noisy diffs (time/space quantization).

This multi-tiered strategy is chosen for Holochain because of the observation that, in a typical application, the set of data created recently changes more often than the set of data created further in the past. In fact, as long as peers are synchronizing frequently, the latter set should only change when a partial or full network partition is resolved.

A secondary concern is that, for many applications such as social media, digital currencies, or telemetry, historical data is less relevant and accessed less frequency than recent data. Any discrepancy between two peers' views of the total data set can often in practice be tolerated.

Hence, this approach favors freshness of recent data so that it becomes available to all peers in a timely fashion

[21] While we use the term 'gossip' exclusively for the slow-heal strategy, both fast-push and slow-heal can be considered a gossip protocol (see https://en.wikipedia.org/wiki/Gossip_protocol), as in both strategies a piece of data is initially communicated to a small number of peers who then communicate it to a larger number of their peers.

expected of modern networked applications, while resolution of discrepancies in historical data is treated as a maintenance concern.

Security & Safety

Many factors contribute to a system's ability to live up to the varying safety and security requirements of its users. In general, the approach taken in Holochain is to provide affordances that take into account the many types of real-world costs that result from adding security and safety to systems such that application developers can match the trade-offs of those costs to their application context. The integrity guarantees listed in prior sections detail the fundamental data safety that Holochain applications provide. Two other important facets of system security and safety come from:

1. Gating access to functions that change state, for which Holochain provides a unified and flexible Object Capabilities model.
2. Detecting and blocking participation of bad actors, for which Holochain provides the affordances of validation, warranting, and blocking.

Cryptographic Object Capabilities

To use a Holochain application, end-users must trigger zome calls that effect local state changes on their Source Chains. Additionally, zome functions can make calls to other zome functions on remote nodes in the same application network, or to other cells running on the same conductor. All of these calls must happen in the context of some kind of permissioning system. Holochain's security model for calls is based on the Object-capability security model[22], but augmented for a distributed cryptographic context in which we use cryptographic signatures to further prove the necessary agency for taking action and create an additional defense against undesired capability leakage.

Access is thus mediated by Capability Grants of four types:

- **Author**: Only the agent owning the source change can make the zome call. This capability is granted to all zome functions.
- **Assigned**: Only the named agent(s) with the given capability secret can make the zome call.
- **Transferrable**: Anybody with the given capability secret can make the zome call. This is equivalent to the common definition of object-capabilities.
- **Unrestricted**: Anybody can make the zome call (no secret nor proof of authorized key needed to use this capability).

All zome calls must be signed and supply a required capability claim argument that MUST be checked by the system receiving the call. Agents record capability grants on their source chains and distribute their associated secrets as necessary according to the application's needs. Receivers of secrets can record them as claims (usually as a private entry) on their chains for later lookup. The "agent" type grant is just the agent's public key.

Warrants

We take that, by definition, in a fully distributed system, there is no way for a single agent to control the actions of other agents that comprise the system; i.e., what makes an agent an agent is its ability to act independently. This creates a challenge: How do agents deal with "bad-actor" agents, as they cannot be controlled by another party?

In Holochain "bad-action" is defined by attempts by agents to act in a way inconsistent with a DNA's validation rules. Because a DNA's network ID is defined by the hash of its integrity bundle (which includes both data structures and the deterministic validation rules) we can know that every agent in a network started with the same rules, and thus can deterministically run those rules to determine if any action fails validation. (Note that some validation rules reveal bad actions not just in structure or content of data committed, but also bad behavior. For example, validating timestamps over contiguous sequences of Actions enables detection of and protection against spam and denial-of-service attacks. Holochain has its own base validation rules as well; for instance, a source chain must never 'fork', so the presence of two parallel branching points from one prior source chain record is considered a bad-action.)

Once a bad-action has been identified via a validation failure, it is considered to be unambiguously a consequence of malicious intent. The only way invalid data can be published is by intentionally circumventing the validation process on the author's device when committing to chain.

Each Warrant must be self-proving. It must flag the agent being warranted as a bad actor and include references to set of actions which fail to validate. This might be, for example, a single signed Action that fails validation, or it might be a set of Actions that are issued consecutively which exceed spam rate limits, or a set of Actions that are issued concurrently which cause the agent's chain to fork.

Upon receipt of a Warrant, a node must take three actions:

1. **Determine who is the bad actor.** For any Warrant, someone either performed a bad action, or someone created a false report of bad action. So a node must validate the referenced actions. If they fail validation, then the reported agent is the bad actor. If the actions pass validation, then the Warrant author is the bad actor.
2. **Block the bad actor.** Add either the warranted agent or the Warrant author to the validating node's peer block list. This node will no longer interact

[22] See https://en.wikipedia.org/wiki/Object-capability_model.

with bad actor, and will reject any connection attempts from that agent.
3. **Report it to the bad actor's Agent Activity Authorities.** Because nodes expect to be able to find out if an agent is warranted by asking its neighbors who validate its chain activity, those neighbors must be notified of any warrants.

There is no global blocking of a bad actor. Each agent must confirm for themselves whom to block. Warrants and blocking, taken together, enable the network to defend itself from bad actors while preserving individual agency in the warranting process.

Note: Beyond Warrants, blocking can also theoretically be used by apps or agents for whatever reason the application logic or node owner may have to refuse to participate with a node. It allows for local, voluntary self-defense against whatever nodes someone might interpret as malicious, or simply ending communication with peers that are no longer relevant (e.g., a terminated employee).

Cross-DNA Composability

Holochain is designed to be used to build micro-services that can be assembled into applications. We expect DNAs to be written that assume the existence of other long-running DNAs and make calls to them via the agency of a user having installed both DNAs on their node. The Capabilities security model described above makes sure this kind of calling is safe and can only happen when permissions to do so have been explicitly granted in a given context. The HDK `call` function provides an affordance to allow specification of the DNA by hash when making the call, so the Holochain node can make a zome call to that DNA and return the result to the calling node.

Holochain Implementation

Given the above formal description of our local state model (Source Chain) and shared data model (Graph DHT) we can now present a high-level implementation specification of different components of the Holochain architecture. The components include:

- App Virtual Machine (Ribosome)
- State Management (Workflows)
- P2P Networking (Kitsune and Holochain P2P)
- Application Interface (Conductor API)
- Secure Private Key Management (lair-keystore)

Please see the Implementation Spec (Appendix A) for details.

Conclusion

We have described an approach to distributed systems design that achieves increasing capacities of social coordination and coherence without requiring the bottlenecks of global consensus, which delivers on the promise of massively scalable and secure distributed applications fit for heterogeneous contexts. This approach has been fully demonstrated. Appendix A provides a complete implementation spec.

Appendix A: Holochain Implementation Spec v0.3.0 Beta

So far we have described the necessary components of a scalable coordination and collaboration system. We have built an "industrial strength" implementation of this pattern suitable for real-world deployment, under the name Holochain. Here we describe the technical implementation details that achieve the various requirements described above.

This specification assumes that the reader has understood context and background provided in the Holochain Formalization.

Given the formal description from that document of our local state model (Source Chain) and shared data model (Graph DHT) we can now present a high-level implementation specification of the different components of the Holochain architecture:

- App Virtual Machine (Ribosome)
- Workflows
- P2P Networking (Kitsune)
- The Conductor
- Secure Private Key Management (lair-keystore)

Note on code fidelity: The code in this appendix may diverge somewhat from the actual implementation, partially because the implementation may change and partially to make the intent of the following code clearer and simpler. For instance, specialized value types that are merely wrappers around a vector of bytes are frequently replaced with `Vec<u8>`.

Ribosome: The Application "Virtual Machine"

We use the term **Ribosome** to the name of part of the Holochain system that runs the DNA's application code. Abstractly, a Ribosome could be built for any programming language as long as it's possible to deterministically hash and run the code of the DNA's Integrity Zome such that all agents who possess the same hash can rely on the validation routines and structure described by that Integrity Zome operating identically for all. (In our implementation we use WebAssembly (WASM) for DNA code, and Wasmer[23] as the runtime that executes it.)

The Ribosome, as an application host, must expose a minimal set of functions to guest applications to allow them to access Holochain functionality, and it should expect that guest applications implement a minimal set of callbacks that allow the guest to define its entry types, link types, validation functions, and lifecycle hooks for both Integrity and Coordinator Zomes. We will call this set of provisions and expectations the Ribosome Host API.

Additionally, it is advantageous to provide software development kits (SDKs) to facilitate the rapid development of Integrity and Coordinator Zomes that consume the Ribosome's host functions and provide the callbacks it expects.

In our implementation we provide SDKs for Integrity and Coordinator Zomes written in the Rust programming language[24] as Rust crates: the Holochain Deterministic Integrity (HDI) crate[25] facilitates the development of Integrity Zomes, while the Holochain Development Kit (HDK) crate[26] facilitates the development of Coordinator Zomes.

Ribosome/Zome Interop ABI

Because WebAssembly code can only interface with its host system via function calls that pass simple numeric scalars, an application binary interface (ABI) must be defined to pass rich data between the Ribosome host and the zome guest.

The host and guest expose their functionality via named functions, and the input and output data of these functions (a single argument and a return value) are passed as a tuple of a shared memory pointer and a length. This tuple is a reference to the serialized data that makes up the actual input or output data.

The caller is responsible for serializing the expected function argument and storing it in a shared memory location in the WebAssembly virtual machine instance, then passing the location and length to the callee.

The callee then accesses the data at the given location, attempts to deserialize it, and operates on the deserialized result.

The same procedure is followed for the function's return value, with the role of the caller and callee reversed.

---

[23] See https://wasmer.io/.
[24] See https://rust-lang.org.
[25] See https://docs.rs/hdi/.
[26] See https://docs.rs/hdk/.

Because errors may occur when the callee attempts to access and deserialize its argument data, the callee MUST return (or rather, serialize, store, and return the address and length of) a Rust `Result<T, WasmError>` value, where `WasmError` is a struct of this type:

```
struct WasmError {
    file: String,
    line: u32,
    error: WasmErrorInner,
}

enum WasmErrorInner {
    PointerMap,
    Deserialize(Vec<u8>),
    Serialize(SerializedBytesError),
    ErrorWhileError,
    Memory,
    Guest(String),
    Host(String),
    HostShortCircuit(Vec<u8>),
    Compile(String),
    CallError(String),
    UninitializedSerializedModuleCache,
}
```

The type `Result<T, WasmError>` is aliased to `ExternResult<T>` for convenience, and will be referred to as such in examples below.

Our implementation provides a `wasm_error!` macro for the guest that simplifies the construction of an error result with the correct file and line number, along with a `WasmErrorInner::Guest` containing an application-defined error string.

Our implementation also provides various macros to abstract over the mechanics of this process, wrapping host functions and guest callbacks, automatically performing the work of retrieving/deserializing and serializing/storing input and output data, and presenting more ergonomic function signatures (in the case of host functions) or allowing application developers to write more ergonomic function signatures (in the case of guest functions). In particular, the `#[hdk_extern]` procedural macro, when applied to a guest function, handles the conversion of the bytes stored in the memory to a map of arguments, passes those arguments, and handles the conversion of the return value to bytes stored in memory.

Hereafter, our examples of host and guest functions will assume the use of ergonomic function signatures.

Handling Guest Functions

For any guest function, the Ribosome MUST prepare a context which includes the list of host functions which may be called by the given type of function:

- Guest functions which are only intended to establish valid entry and link types (`entry_defs` and `link_types`) MUST NOT be given access to any host functions.
- Guest functions which are expected to give a repeatable result for the input arguments (`validate`) MUST NOT be given access to host functions whose return values vary by context.
- Guest functions which are expected to not change source chain state (`validate`, `genesis_self_check`, `post_commit`) MUST NOT be given access to host functions which change state.

For any guest functions which are permitted to change source chain state (`init`, `recv_remote_signal`, zome functions, and scheduled functions), the Ribosome MUST:

1. Prepare a context which includes the aforementioned host function access, as well as the current source chain state and a temporary "scratch space" into which to write new source chain state changes.
2. Check the state of the source chain; if it does not contain an `InitZomesComplete` action, run the `init` callback and remember any state changes in the scratch space.
3. If no `init` callbacks fail, proceed to call the guest function, remembering any state changes in the scratch space.
4. Transform the state changes in the scratch space into DHT operations.
5. Attempt to validate the DHT operations.

6. If all the DHT operations are valid, persist the Actions in the scratch space to the source chain.
7. If the called function was a zome function, return the zome function call's return value to the caller.
8. Spawn the `post_commit` callback in the same Coordinator Zome as the called guest function and attempt to publish the DHT operations to the DHT.

State changes in a scratch space MUST be committed atomically to the source chain; that is, all of them MUST be written or fail as a batch.

HDI

The Holochain Deterministic Integrity (HDI) component of the Holochain architecture comprises the functions and capacities that are made available to app developers for building their Integrity Zomes.

Integrity Zomes provide the immutable portion of the app's code that:

- identifies the types of entries and links able to be committed in the app,
- defines the structure of data entries, and
- defines the validation code each node runs for DHT operations produced by actions to create, update, and delete the aforementioned entry types, as well as for a small number of system types.

The following data structures, functions and callbacks are necessary and sufficient to implement an HDI:
Core Holochain Data Types

The `Action` Data Type All actions MUST contain the following data elements (with the exception of the `Dna` action which, because it indicates the creation of the first chain entry, does not include the `action_seq` nor `prev_action` data elements):

```
{
    author: AgentHash,
    timestamp: Timestamp,
    action_seq: u32,
    prev_action: ActionHash,
    ...
}
```

Additionally, the HDI MUST provide a signed wrapper data structure that allows integrity checking in validation:

```
struct Signed<T>
where T: serde::Serialize {
    signature: Signature,
    data: T,
}

// A signature is an Ed25519 public-key signature.
struct Signature([u8; 64]);
```

Implementation detail: Theoretically all actions could point via a hash to an entry that would contain the "content" of that action. But because many of the different actions entries are system-defined, and they thus have a known structure, we can reduce unnecessary data elements and gossip by embedding the entry data for system-defined entry types right in the action itself. However, for application-defined entry types, because the structure of the entry is not known at compile time for Holochain, the entry data must be in a separate data structure. Additionally there are a few system entry types (see below) that must be independently retrievable from the DHT, and thus have their own separate system-defined variant of the `Entry` enum type.

Many, though not all, actions comprise intentions to create, read, update, or delete (CRUD), data on the DHT. The action types and their additional data fields necessary are:

- `Dna`: indicates the DNA hash of the validation rules by which the data in this source chain agrees to abide.

  ```
  struct Dna {
      ...
      hash: DNAHash,
  }
  ```

- `AgentValidationPkg`: indicates the creation of an entry holding the information necessary for nodes to confirm whether an agent is allowed to participate in this DNA. This entry is contained in the action struct.

```
struct AgentValidationPkg {
    ...
    membrane_proof: Option<SerializedBytes>
}
```

- `InitZomesComplete`: indicates the creation of the final genesis entry that marks that all zome init functions have successfully completed (see the HDK section for details), and the chain is ready for commits. Requires no additional data.
- `Create`: indicates the creation of an application-defined entry, or a system-defined entry that needs to exist as content-addressed data.

```
struct Create {
    ...
    entry_type: EntryType,
    entry_hash: EntryHash,
}

// See the section on Entries for the definition of `EntryType`.
```

- `Update`: Mark an existing entry and its creation action as updated by itself. In addition to referencing the new entry, the action data points to the old action and its entry. As this is an entry creation action like `Create`, it shares many of the same fields.

```
struct Update {
    ...
    original_action_address: ActionHash,
    original_entry_address: EntryHash,
    entry_type: EntryType,
    entry_hash: EntryHash,
}
```

- `Delete`: Marks an existing entry and its creation action as deleted. The entry containing the hashes of the action and entry to be deleted are contained in the action struct.

```
struct Delete {
    ...
    deletes_address: ActionHash,
    deletes_entry_address: EntryHash,
}
```

- `CreateLink`: Indicates the creation of a link.

```
struct CreateLink {
    ...
    base_address: AnyLinkableHash,
    target_address: AnyLinkableHash,
    zome_index: u8,
    link_type: u8,
    tag: Vec<u8>,
}
```

- `DeleteLink`: Indicates the marking of an existing link creation action as deleted.

```
struct DeleteLink {
    ...
    base_address: AnyLinkableHash,
    link_add_address: ActionHash,
}
```

- `CloseChain`: indicates the creation of a final chain entry with data about a new DNA version to migrate to.

```
struct CloseChain {
    ...
```

```
    new_dna_hash: DnaHash,
}
```

- `OpenChain`: indicates the creation of an entry with data for migrating from a previous DNA version.

```
struct OpenChain {
    ...
    prev_dna_hash: DnaHash,
}
```

All of the CRUD actions MUST include data to implement rate-limiting so as to prevent malicious network actions. In our implementation, all CRUD actions have a `weight` field of the following type:

```
struct RateWeight {
    bucket_id: u8,
    units: u8,
}
```

An application may specify an arbitrary number of rate limiting 'buckets', which can be 'filled' by CRUD actions until they reach their capacity, after which point any further attempts to record an action to the Source Chain will fail until the bucket has drained sufficiently. Each bucket has a specified capacity and drain rate, which the Integrity Zome may specify using a `rate_limits` callback.

The Integrity Zome may also weigh a given CRUD action using a `weigh` callback, which allows both the author and the validating authority to deterministically assign a weight to an action.

Note: This feature is not completed in the current implementation.

The `Entry` Data Type There are four main entry types, defined in an `EntryType` enum:

```
enum EntryType {
    AgentPubKey,
    App(AppEntryDef),
    CapClaim,
    CapGrant,
}
```

There is also an `Entry` enum that holds the entry data itself, with five variants that correspond to the four entry types:

```
enum Entry {
    Agent(AgentHash),
    App(SerializedBytes),
    CounterSign(CounterSigningSessionData, SerializedBytes),
    CapClaim(CapClaim),
    CapGrant(ZomeCallCapGrant),
}
```

(Note that the `App` and `CounterSign` variants are both intended for application-defined entries.)

- `AgentPubKey` is used in the second genesis record of the source chain, a `Create` action that publishes the source chain author's public key to the DHT for identification and verification of authorship.
- `App` indicates that the entry data contains arbitrary application data of a given entry type belonging to a given integrity zome:

```
struct AppEntryDef {
    entry_index: u8,
    zome_index: u8,
    visibility: EntryVisibility,
}

struct EntryVisibility {
    Public,
    Private,
}
```

Its entry data can be of either `Entry::App` or `Entry::CounterSign`, where the inner data is an arbitrary vector of bytes (typically a serialized data structure). If the data is `Entry::CounterSign`, the bytes are accompanied by a struct that gives the details of the countersigning session (this struct will be dealt with in the Countersigning section).

Note that in both these cases the data is stored using a serialization that is declared by the `entry_defs()` function of the HDI.

- `CapClaim` indicates that the entry data contains the details of a granted capability that are necessary to exercise such capability:

```
struct CapClaim {
    tag: String,
    grantor: AgentHash,
    secret: CapSecret,
}
```

- `CapGrant` indicates that the entry data contains the details of a capability grant in the following enum and the types upon which it depends:

```
struct ZomeCallCapGrant {
    tag: String,
    access: CapAccess,
    functions: GrantedFunctions,
}

enum CapAccess {
    Unrestricted,
    Transferable {
        secret: [u8; 64],
    },
    Assigned {
        secret: [u8; 64],
        assignees: BTreeSet<AgentHash>,
    },
}

enum GrantedFunctions {
    All,
    Listed(BTreeSet<(ZomeName, FunctionName), Global>),
}

struct ZomeName(str);

struct FunctionName(str);
```

The `Record` Data Type A record is just a wrapper for an `Action` and an `Entry`. Because an entry may not be present in all contexts or for all action types, the `RecordEntry` enum wraps the possible entry data in an appropriate status.

```
struct Record {
    action: SignedHashed<Action>,
    entry: RecordEntry,
}

enum RecordEntry {
    Present(Entry),
    Hidden,
    NA,
    NotStored,
}
```

Links A `CreateLink` action completely contains the relational graph information, which would be considered the link's entry data if it were to have a separate entry. Note that links are typed for performance purposes, such that when requesting links they can be retrieved by type. Additionally links have tags that can be used as arbitrary labels on-graph as per the application's needs. The `zome_index` is necessary so that the system can find and dispatch the correct validation routines for that link, as a DNA may have multiple integrity zomes.

```
struct Link {
    base_address: AnyLinkableHash,
    target_address: AnyLinkableHash,
    zome_index: ZomeIndex,
    link_type: LinkType,
    tag: LinkTag,
}

struct LinkTag(Vec<u8>);
```

Comparing this structure to a Resource Description Framework (RDF) triple:

- The `base_address` is the subject.
- The `target_address` is the object.
- The `zome_index`, `link_type`, and `tag` as a tuple are the predicate.

The `Op` Data Type The `Op` types that hold the chain entry data that is published to different portions of the DHT (formally described in the Graph Transformation section of Formal Design Elements) are listed below. The integrity zome defines a validation callback for the entry and link types it defines, and is called with an `Op` enum variant as its single parameter, which indicates the DHT perspective from which to validate the data. Each variant holds a struct containing the DHT operation payload:

- `StoreRecord`: executed by the record (action) authorities to store data. It contains the record to be validated, including the entry if it is public.

  ```
  struct StoreRecord {
      record: Record,
  }
  ```

- `StoreEntry`: executed by the entry authorities to store data for any entry creation action, if the entry is public. It contains both the entry and the action in a struct similar to `Record`, with the exception that the `entry` field is always populated.

  ```
  struct StoreEntry {
      action: SignedHashed<EntryCreationAction>,
      entry: Entry,
  }

  // The following variants hold the corresponding Action struct.
  enum EntryCreationAction {
      Create(Create),
      Update(Update),
  }
  ```

- `RegisterUpdate`: executed by both the entry and record authorities for the *old* data to store metadata pointing to the *new* data. This op collapses both the `RegisterUpdatedRecord` and `RegisterUpdatedContent` operations into one for simplicity. It contains the update action as well as the entry, if it is public.

  ```
  struct RegisterUpdate {
      update: SignedHashed<Update>,
      new_entry: Option<Entry>,
  }
  ```

- `RegisterDelete`: executed by the entry authorities for the *old* entry creation and its entry to store metadata that tombstones the data. This opp collapses both the `RegisterDeletedEntryAction` and `RegisterDeletedBy` operations into one. It contains only the delete action.

```
struct RegisterDelete {
    delete: SignedHashed<Delete>,
}
```

- `RegisterAgentActivity`: executed by agent activity authorities (the peers responsible for the author's AgentID entry) to validate the action in context of the author's entire source chain. At the application developer's discretion, this operation can also contain the entry data.

```
struct RegisterAgentActivity {
    action: SignedHashed<Action>,
    cached_entry: Option<Entry>,
}
```

- `RegisterCreateLink`: executed by the authorities for the link's base address to store link metadata.

```
struct RegisterCreateLink {
    create_link: SignedHashed<CreateLink>,
}
```

- `RegisterDeleteLink`: executed by the authorities for the link's base address to store metadata that tombstones the link.

```
struct RegisterDeleteLink {
    delete_link: SignedHashed<DeleteLink>,
    create_link: CreateLink,
}
```

Hash Data Structures Holochain relies on being able to distinguish and use hashes of the various Holochain fundamental data types. The following hash types must exist:

- `ActionHash`: The Blake2b-256 hash of a serialized `Action` variant, used for DHT addressing.
- `AgentHash`: The Ed25519 public key of an agent, used for referencing the agent.
- `DhtOpHash`: The Blake2b-256 hash of a serialized `DhtOp` variant, used for comparing lists of held operations during syncing between authorities.
- `DnaHash`: The hash of all the integrity zomes and associated modifiers, when serialized in a consistent manner.
- `EntryHash`: The hash of the bytes of a `Entry` variant, according to the hashing rules of that variant (the Blake2b-256 hash of the serialized variant in all cases except `Entry::Agent`, which is the public key). Used for DHT addressing.
- `ExternalHash`: This type is used for creating links in the graph DHT to entities that are not actually stored in the DHT. It is simply an arbitrary 32 bytes.
- `WasmHash`: The Blake2b-256 hash of the WebAssembly bytecode of a zome, used by the Ribosome to look up and call zomes.

Furthermore, there are two composite hash types, which are unions of two or more of the preceding hash types:

- `AnyDhtHash`, the enum of `EntryHash` and `ActionHash`, is the union of all 'real' addressable content on the DHT; that is, content that can actually be written.
- `AnyLinkableHash`, the enum of `EntryHash`, `ActionHash`, and `ExternalHash`, is the union of all real and imaginary addressable content on the DHT; that is, it includes external hashes.

All of these hash types are derived from a generic struct, `HoloHash<T>`, which holds the three-byte hash type signifier and the 32 bytes of the hash (the 'core' of the hash), along with the 4-byte network location. For those hash types that are the basis of addressable content (`AnyDhtHash`), the hash alone is sufficient to uniquely identify a DHT basis from which a network location can be computed, while the type signifier ensures type safety in all struct fields and enum variant values that reference the hash. The four-byte network location is computed from the hash core and stored along with the preceding 36 bytes as a matter of convenience.

The three-byte type signifiers are as follows:

Type	Hexadecimal	Base64
`ActionHash`	`0x842924`	`hCkk`
`AgentHash`	`0x842024`	`hCAk`
`DhtOpHash`	`0x842424`	`hCQk`
`DnaHash`	`0x842d24`	`hC0k`
`EntryHash`	`0x842124`	`hCEk`

Type	Hexadecimal	Base64
`ExternalHash`	`0x842f24`	`hC8k`
`WasmHash`	`0x842a24`	`hCok`

Application Type Definition Callbacks In order for the Ribosome to successfully dispatch validation to the correct integrity zome, each integrity zome in a DNA should register the entry and link types it is responsible for validating. The HDI MUST allow the integrity zome to implement the following functions:

- `entry_defs(()) -> ExternResult<EntryDefsCallbackResult>`: Called to declare the type and structure of the application's entry types. The return value is:

  ```
  enum EntryDefsCallbackResult {
      Defs(EntryDefs),
  }

  struct EntryDefs(Vec<EntryDef>);

  struct EntryDef {
      id: EntryDefId,
      visibility: EntryVisibility,
      required_validations: u8,
      cache_at_agent_activity: bool,
  }

  enum EntryDefId {
      App(str),
      CapClaim,
      CapGrant,
  }
  ```

  This function can be automatically generated using the `#[hdk_entry_types]` procedural macro on an enum of variants that each hold a type that can be serialized and deserialized.

- `link_types(()) -> ExternResult<Vec<u8>>`: called to declare the link types that will be used by the application. This function can be automatically generated using the `#[hdk_link_types]` procedural macro on an enum of all link types.

Note: In our implementation these functions are automatically generated by Rust macros. This gives us the benefit of consistent, strongly typed entry and link types from the point of definition to the point of use. Thus it's very easy to assure that any application data that is being stored adheres to the entry and link type declarations.
Functions Necessary for Application Validation The HDI MUST allow for hApp developers to specify a `validate(Op) -> ExternResult<ValidateCallbackResult>` callback function for each integrity zome. This callback is called by the Ribosome in the correct context for the Op as described above in the graph DHT formalization, so that the data associated with the `Op` will only be stored if it meets the validation criteria.

The HDI MUST also allow for hApp developers to specify a `genesis_self_check(GenesisSelfCheckData) -> ExternResult<ValidateCallbackResult>` callback for each integrity zome. This callback is called by the Ribosome *before* attempting to join a network, to perform sanity checks on the genesis records. This callback is limited in its ability to validate genesis data, because it MUST NOT be able to make network calls. Nevertheless, it is useful to prevent a class of errors such as incorrect user entry of membrane proofs from inadvertently banning a new agent from the network. The input payload is defined as:

```
struct GenesisSelfCheckData {
    membrane_proof: Option<SerializedBytes>,
    agent_key: AgentHash,
}
```

The HDI MUST provide the following functions for application authors to retrieve dependencies in validation:

- `must_get_agent_activity(AgentPubKey, ChainFilter) -> ExternResult<Vec<RegisterAgentActivity>>`: This function allows for deterministic validation of chain activity by making a hash-bounded range of an agent's chain into a dependency for something that is being validated. The second parameter is defined as:

```
struct ChainFilter {
    chain_top: ActionHash,
    filters: ChainFilters,
    include_cached_entries: bool
}

enum ChainFilters {
    ToGenesis,
    Take(u32),
    Until(HashSet<ActionHash>),
    Both(u32, HashSet<ActionHash>),
}
```

The vector element type in the return value is defined as:

```
struct RegisterAgentActivity {
    action: SignedHashed<Action>,
    cached_entry: Option<Entry>,
}
```

- `must_get_action(ActionHash) -> ExternResult<SignedHashed<Action>>`: Get the `Action` at a given action hash, along with its author's signature.
- `must_get_entry(EntryHash) -> ExternResult<HoloHashed<Entry>>`: Get the `Entry` at a given hash.
- `must_get_valid_record(ActionHash) -> ExternResult<Record>`: Attempt to get a *valid* `Record` at a given action hash; if the record is marked as invalid by any contacted authorities, the function will fail.

The HDI MUST implement two hashing functions that calculate the hashes of `Action`s and `Entry`s so that hash values can be confirmed in validation routines.

- `hash_action(Action) -> ActionHash`
- `hash_entry(Entry) -> EntryHash`

The HDI MUST implement two introspection functions that return data about the DNA's definition and context that may be necessary for validation:

- `dna_info() -> ExternResult<DnaInfo>`: returns information about the DNA:

```
struct DnaInfo {
    name: String,
    hash: DnaHash,
    modifiers: DnaModifiers,
    zome_names: Vec<ZomeName>,
}

struct DnaModifiers {
    network_seed: String,
    properties: SerializedBytes,
    origin_time: Timestamp,
    quantum_time: Duration,
}
```

- `zome_info() -> ExternResult<ZomeInfo>`: returns information about the integrity zome:

```
struct ZomeInfo {
    name: ZomeName,
    id: ZomeIndex,
    properties: SerializedBytes,
    entry_defs: EntryDefs,
    extern_fns: Vec<FunctionName>,
    zome_types: ScopedZomeTypesSet,
}

struct ZomeIndex(u8);
```

```
struct ScopedZomeTypesSet {
    entries: Vec<(ZomeIndex, Vec<EntryDefIndex>)>,
    links: Vec<(ZomeIndex, Vec<LinkType>)>,
}

struct EntryDefIndex(u8);

struct LinkType(u8);
```

Note: `properties` consists of known application-specified data that is specified at install time (both at the DNA and zome levels) that may be necessary for validation or any other application-defined purpose. Properties are included when hashing the DNA source code, thus allowing parametrized DNAs and zomes.

The HDI MUST implement a function that validation code can use to verify cryptographic signatures:

- `verify_signature<I>(AgentPubKey, Signature, I) -> ExternResult<bool> where I: Serialize`: Checks the validity of a signature (a `Vec<u8>` of bytes) upon the data it signs (any type that implements the `Serialize` trait, allowing it to be reproducibly converted into a vector of bytes, against the public key of the agent that is claimed to have signed it.

HDK

The HDK contains all the functions and callbacks needed for Holochain application developers to build their Coordination Zomes. Note that the HDK is a superset of the HDI. Thus all of the functions and data types available in the HDI are also available in the HDK.

Initialization The HDK MUST allow application developers to define an `init() -> ExternResult<InitCallbackResult>` callback in each coordinator zome. All `init` callbacks in all coordinator zomes MUST complete successfully, and an `InitZomesComplete` action MUST be written to a cell's source chain, before zome functions (see following section) may be called. Implementations SHOULD allow this to happen lazily; that is, a zome function is permitted to be called before initialization, and the call zome workflow runs the initialization workflow in-process if `InitZomesComplete` does not exist on the source chain yet.

The return value of the callback is defined as:

```
enum InitCallbackResult {
    Pass,
    Fail(String),
    UnresolvedDependencies(UnresolvedDependencies),
}
```

If the return value of all `init` callbacks is `Pass`, the actions in the scratch space prepared for the initialization workflow are written to the source chain, followed by the `InitZomesComplete` action, and execution of zome functions in the cell may proceed.

If the return value of at least one `init` callback is `Fail`, the cell is put into a permanently disabled state.

If the return value of at least one `init` callback is `UnresolvedDependencies`, the scratch space prepared for the initialization workflow is discarded, and the initialization workflow will be attempted upon next zome function call. This permits a cell to gracefully handle a temporary poorly connected state on cell instantiation.

Arbitrary API Functions (Zome Functions) The HDK MUST allow application developers to define and expose functions in their Coordinator Zomes with arbitrary names, input payloads, and return payloads that serve as the application's API. While the content of the return payload of these functions may be arbitrary data, it MUST be wrapped in a `Result<T, WasmError>`, where `T` is the return payload.

As function calls across the host/guest interface only deal with arbitrary bytes stored in memory address ranges, the HDK SHOULD provide an abstraction to allow developers to define functions in a more natural manner, with typed input and return payloads. We have provided a `#[hdk_extern]` procedural macro that facilitates this abstraction, wrapping the following function definition with the necessary machinery to load and deserialize the input data and serialize and store the return data.

The Conductor MUST also receive calls to these zome functions, enforce capability restrictions, dispatch the call to the correct WASM module, and handle side effects, error conditions, and the called function's return value. These calls MAY come from external clients, other zomes in the same cell, other cells in the same application, or other agents in the same DHT.

Post-Commit Callback The HDK MUST allow application developers to define a `post_commit(Vec<SignedAction>) -> ExternResult<()>` callback in their Coordinator Zomes which receives a sequence of Actions committed to the source chain. The purpose of this callback is to provide a way of triggering follow-up activities when an atomic commit has definitively succeeded in persisting new Actions.

The Conductor MUST call this callback with all the Actions successfully committed in any guest function that is permitted to persist state changes to the source chain. The Conductor MUST NOT permit this callback to make further state changes, but it MAY allow it to access any other host functions, including calling or scheduling other functions which may make state changes in their own call contexts.

Chain Operations The HDK MUST implement the following functions that create source chain entries:

- `create(CreateInput) -> ExternResult<ActionHash>`: Records the creation of a new application entry. The `CreateInput` parameter is defined as:

  ```
  struct CreateInput {
      entry_location: EntryDefLocation,
      entry_visibility: EntryVisibility,
      entry: Entry,
      chain_top_ordering: ChainTopOrdering,
  }

  enum EntryDefLocation {
      App(AppEntryDefLocation),
      CapClaim,
      CapGrant,
  }

  struct AppEntryDefLocation {
      zome_index: ZomeIndex,
      entry_def_index: EntryDefIndex,
  }

  enum ChainTopOrdering {
      Relaxed,
      Strict,
  }
  ```

  The `EntryVisibility` parameter specifies whether the entry is private or should be published to the DHT, and the `ChainTopOrdering` parameter specifies whether the call should fail if some other zome call with chain creation actions completes before this one, or whether it's ok to automatically replay the re-write the action on top of any such chain entries.

  In our implementation, the `create` function accepts any value that can be converted to a `CreateInput`, allowing most of these fields to be populated by data that was generated by the `#[hdk_entry_types]` macro and other helpers. This is accompanied by convenience functions for `create` that accept app entries, capability grants, or capability claims.

- `update(UpdateInput) -> ExternResult<ActionHash>`: Records the marking of an existing entry and its creation action as updated. Requires the `ActionHash` that created the original entry to be provided. The `UpdateInput` parameter is defined as:

  ```
  struct UpdateInput {
      original_action_address: ActionHash,
      entry: Entry,
      chain_top_ordering: ChainTopOrdering,
  }
  ```

  Many fields necessary for `create` are unnecessary for `update`, as the new entry is expected to match the entry type and visibility of the original. Similar to `create`, in our implementation there are convenience functions to help with constructing `UpdateInputs` for app entries and capability grants.

- `delete(DeleteInput) -> ExternResult<ActionHash>`: Records the marking of an entry and its creation action as deleted. The `DeleteInput` parameter is defined as:

```
struct DeleteInput {
    deletes_action_hash: ActionHash,
    chain_top_ordering: ChainTopOrdering,
}
```

- `create_link(AnyLinkableHash, AnyLinkableHash, ScopedLinkType, LinkTag) -> ExternResult<ActionHash>`: Records the creation of a link of the given `ScopedLinkType` between the hashes supplied in the first and second arguments, treating the first hash as the base and the second as the target. The fourth `LinkTag` parameter is a struct containing a `Vec<u8>` of arbitrary application bytes.
- `delete_link(ActionHash) -> ExternResult<ActionHash>`: Records the marking of a link creation action as deleted, taking the original link creation action's hash as its input.
- `query(ChainQueryFilter) -> ExternResult<Vec<Record>>`: search the agent's local source chain according to a query filter returning the `Record`s that match. The `ChainQueryFilter` parameter is defined as:

```
struct ChainQueryFilter {
    sequence_range: ChainQueryFilterRange,
    entry_type: Option<Vec<EntryType>>,
    entry_hashes: Option<HashSet<EntryHash>>,
    action_type: Option<Vec<ActionType>>,
    include_entries: bool,
    order_descending: bool,
}

enum ChainQueryFilterRange {
    // Retrieve all chain actions.
    Unbounded,
    // Retrieve all chain actions between two indexes, inclusive.
    ActionSeqRange(u32, u32),
    // Retrieve all chain actions between two hashes, inclusive.
    ActionHashRange(ActionHash, ActionHash),
    // Retrieve the n chain actions up to and including the given hash.
    ActionHashTerminated(ActionHash, u32),
}
```

Capabilities Management The HDK includes convenience functions over `create`, `update`, and `delete` for operating on capability grants and claims:

- `create_cap_grant(ZomeCallCapGrant) -> ExternResult<ActionHash>`
- `create_cap_claim(CapClaim) -> ExternResult<ActionHash>`
- `update_cap_grant(ActionHash, ZomeCallCapGrant) -> ExternResult<ActionHash>`
- `delete_cap_grant(ActionHash) -> ExternResult<ActionHash>`

In addition to these, a function is provided for securely generating capability secrets:

- `generate_cap_secret() -> ExternResult<[u8; 64]>`

It is the application's responsibility to retrieve a stored capability claim using a host function such as `query` and supply it along with a remote call to another agent. As the Conductor at the receiver agent automatically checks and enforces capability claims supplied with remote call payloads, there is no need to retrieve and check a grant against a claim.

DHT Data Retrieval

- `get(AnyDhtHash, GetOptions) -> ExternResult<Option<Record>>`: Retrieve a `Record` from the DHT by its `EntryHash` or `ActionHash`. The content of the record return is dependent on the type of hash supplied:
  - If the hash is an `Entry` hash, the authority will return the `entry` content paired with its oldest-timestamped `Action`.
  - If the hash is an `Action` hash, the authority will return the specified action.

  The `GetOptions` parameter is defined as:

```
struct GetOptions {
    strategy: GetStrategy,
}
```

```
enum GetStrategy {
    Network,
    Local,
}
```

If `strategy` is `GetStrategy::Network`, the request will always go to other DHT authorities, unless the the requestor is an authority for that basis hash themselves. If `strategy` is `GetStrategy::Local`, the request will always favor the requestor's local cache and will return nothing if the data is not cached.

- `get_details(AnyDhtHash, GetOptions) -> ExternResult<Option<Details>>`: Retrieve all of the addressable data and metadata at a basis hash. The return value is a variant of the following enum, depending on the data stored at the hash:

```
enum Details {
    Record(RecordDetails),
    Entry(EntryDetails),
}

struct RecordDetails {
    record: Record,
    validation_status: ValidationStatus,
    deletes: Vec<SignedHashed<Action>>,
    updates: Vec<SignedHashed<Action>>,
}

enum ValidationStatus {
    // The `StoreRecord` operation is valid.
    Valid,
    // The `StoreRecord` operation is invalid.
    Rejected,
    // Could not validate due to missing data or dependencies, or an
    // exhausted WASM execution budget.
    Abandoned,
    // The action has been withdrawn by its author.
    Withdrawn,
}

struct EntryDetails {
    entry: Entry,
    actions: Vec<SignedHashed<Action>>,
    rejected_actions: Vec<SignedHashed<Action>>,
    deletes: Vec<SignedHashed<Action>>,
    updates: Vec<SignedHashed<Action>>,
    entry_dht_status: EntryDhtStatus,
}

enum EntryDhtStatus {
    // At least one `StoreEntry` operation associated with the entry is
    // valid, and at least one entry creation action associated with it has
    // not been deleted.
    Live,
    // All entry creation actions associated with the entry have been marked
    // as deleted.
    Dead,
    // All `StoreEntry` operations are waiting validation.
    Pending,
    // All `StoreEntry` operations associated with the entry are invalid.
    Rejected,
    // All attempts to validate all `StoreEntry` operations associated with
    // the entry have been abandoned.
```

```
    Abandoned,
    // All entry creation actions associated with the entry have been
    // withdrawn their authors.
    Withdrawn,
    // The entry data has been purged.
    Purged,
}
```

- `get_links(GetLinksInput) -> ExternResult<Vec<Link>>`: Retrieve a list of links that have been placed on any base hash on the DHT, optionally filtering by the links' types and/or tags. The returned list contains only live links; that is, it excludes the links that have `DeleteLink` actions associated with them. The `GetLinksInput` parameter is defined as:

```
struct GetLinksInput {
    base_address: AnyLinkableHash,
    link_type: LinkTypeFilter,
    get_options: GetOptions,
    tag_prefix: Option<Vec<u8>>,
    after: Option<Timestamp>,
    before: Option<Timestamp>,
    author: Option<AgentHash>,
}

enum LinkTypeFilter {
    // One link type
    Types(Vec<(ZomeIndex, Vec<LinkType>)>),
    // All link types from the given integrity zome
    Dependencies(Vec<ZomeIndex>),
}
```

- `get_link_details(AnyLinkableHash, LinkTypeFilter, Option<LinkTag>, GetOptions) -> ExternResult<LinkDetails>`: Retrieve the link creation *and* deletion actions at a base. The return value is defined as:

```
struct LinkDetails(Vec<(SignedActionHashed, Vec<SignedActionHashed>)>);
```

  where each element in the vector is a `CreateLink` action paired with a vector of any `DeleteLink` actions that apply to it.

- `count_links(LinkQuery) -> ExternResult<usize>`: Retrieve only the count of live links matching the link query.

- `get_agent_activity(AgentPubKey, ChainQueryFilter, ActivityRequest) -> ExternResult<AgentActivity>`: Retrieve the activity of an agent from the agent's neighbors on the DHT. This functions similar to `query`, but operates on the source chain of an agent *other* than the requestor. The `ActivityRequest` parameter is defined as:

```
enum ActivityRequest {
    Status,
    Full,
}
```

  The `AgentActivity` return value is defined as:

```
struct AgentActivity {
    valid_activity: Vec<(u32, ActionHash)>,
    rejected_activity: Vec<(u32, ActionHash)>,
    status: ChainStatus,
    highest_observed: Option<(u32, ActionHash)>,
    warrants: Vec<Warrant>,
}

enum ChainStatus {
    Empty,
```

```
    Valid(ChainHead),
    Forked(ChainFork),
    Invalid(ChainHead),
}

struct ChainHead {
    action_seq: u32,
    hash: ActionHash,
}

struct ChainFork {
    fork_seq: u32,
    first_action: ActionHash,
    second_action: ActionHash,
}
```

Depending on the value of the `ActivityRequest` argument, `status` may be the only populated field.

- `get_validation_receipts(GetValidationReceiptsInput) -> ExternResult<Vec<ValidationReceiptSet>>`: Retrieve information about how 'persisted' the DHT operations for an Action are. This is meant to provide end-user feedback on whether an agent's authored data can easily be retrieved by other peers. The input argument is defined as:

```
struct GetValidationReceiptsInput {
    action_hash: ActionHash,
}
```

The return value is defined as a vector of:

```
struct ValidationReceiptSet {
    // The DHT operation hash that this receipt is for.
    op_hash: DhtOpHash,
    // The type of the op that was validated. This represents the underlying
    // operation type and does not map one-for-one to the `Op` type used in
    // validation.
    op_type: String,
    // Whether this op has received the required number of receipts.
    receipts_complete: bool,
    // The validation receipts for this op.
    receipts: Vec<ValidationReceiptInfo>,
}
```

Introspection

- `agent_info() -> ExternResult<AgentInfo>`: Get information about oneself (that is, the agent currently executing the zome function) and one's source chain, where the return value is defined as:

```
struct AgentInfo {
    agent_initial_pubkey: AgentHash,
    agent_latest_pubkey: AgentHash,
    chain_head: (ActionHash, u32, Timestamp),
}
```

The initial and latest public key may vary throughout the life of the source chain, as an `AgentPubKey` is an entry which may be updated like other entries. Updating a key entry is normally handled through a DPKI implementation (see Human Error section of System Correctness: Confidence).

- `call_info() -> ExternResult<CallInfo>`: Get contextual information about the current zome call, where the return value is defined as:

```
struct CallInfo {
    provenance: AgentHash,
    function_name: FunctionName,
    // A snapshot of the source chain state at zome call time.
    as_at: (ActionHash, u32, Timestamp),
```

```
    // The capability grant under which the call is permitted.
    cap_grant: CapGrant,
}
```

- `dna_info() -> ExternResult<DnaInfo>` (see HDI)
- `zome_info() -> ExternResult<ZomeInfo>` (see HDI)

Modularization and Composition Zomes are intended to be units of composition for application developers. Thus zome functions MUST be able to make calls to other zome functions, either in the same zome or in other zomes or even DNAs:

- `call<I>(CallTargetCell, ZomeName, FunctionName, Option<CapSecret>, I) -> ZomeCallResponse where I: Serialize`: Call a zome function in a local cell, supplying a capability and a payload containing the argument to the receiver. The `CallTargetCell` parameter is defined as:

```
enum CallTargetCell {
    // Call a function in another cell by its unique conductor-local ID, a
    // tuple of DNA hash and agent public key.
    OtherCell(CellId),
    // Call a function in another cell by the role name specified in the app
    // manifest. This role name may be qualified to a specific clone of the
    // DNA that fills the role by appending a dot and the clone's index.
    OtherRole(String),
    // Call a function in the same cell.
    Local,
}

struct CellId(DnaHash, AgentPubKey);
```

Clone Management The HDK SHOULD implement the ability for cells to modify the running App by adding, enabling, and disabling clones of existing DNA.

- `create_clone_cell(CreateCloneCellInput) -> ExternResult<ClonedCell>`: Create a clone of an existing DNA installed with the App, specifying new modifiers and optionally a membrane proof. The input parameter is defined as:

```
struct CreateCloneCellInput {
    // The ID of the cell to clone.
    cell_id: CellId,

    // Modifiers to set for the new cell. At least one of the modifiers must
    // be set to obtain a distinct hash for the clone cell's DNA.
    modifiers: DnaModifiersOpt<YamlProperties>,
    // Optionally set a proof of membership for the clone cell.
    membrane_proof: Option<MembraneProof>,
    // Optionally a name for the DNA clone.
    name: Option<String>,
}

struct DnaModifiersOpt<P> {
    network_seed: Option<String>,
    properties: Option<P>,
    origin_time: Option<Timestamp>,
    // The smallest size of time regions for historical gossip.
    quantum_time: Option<Duration>,
}

type MembraneProof = SerializedBytes;
```

Implementations MUST NOT enable the clone cell until `enable_clone_cell` is subsequently called.

The return value is defined as:

```
struct ClonedCell {
```

```
    cell_id: CellId,
    // A conductor-local clone identifier.
    clone_id: CloneId,
    // The hash of the DNA that this cell was instantiated from.
    original_dna_hash: DnaHash,
    // The DNA modifiers that were used to instantiate this clone cell.
    dna_modifiers: DnaModifiers,
    // The name the cell was instantiated with.
    name: String,
    // Whether or not the cell is running.
    enabled: bool,
}
```

- `disable_clone_cell(DisableCloneCellInput) -> ExternResult<()>`: Disable an active clone cell in the current app. The input is defined as:

```
struct DisableCloneCellInput {
    clone_cell_id: CloneCellId,
}

enum CloneCellId {
    // Clone ID consisting of role name and clone index.
    CloneId(CloneId),
    // Cell id consisting of DNA hash and agent key.
    CellId(CellId),
}

// A conductor-local unique identifier for a clone, consisting of the role
// name from the app manifest and a clone index, delimited by a dot.
struct CloneID(String);
```

- `enable_clone_cell(EnableCloneCellInput) -> ExternResult<ClonedCell>`: Enable a cloned cell in the current app. The input is defined as:

```
struct EnableCloneCellInput {
    clone_cell_id: CloneCellId,
}
```

- `delete_clone_cell(DeleteCloneCellInput) -> ExternResult<()>`: Delete an existing clone cell in the current app. The input is defined as:

```
struct DeleteCloneCellInput {
    clone_cell_id: CloneCellId,
}
```

Scheduling The HDK SHOULD implement the ability for zome calls to be scheduled for calling in the future, which allows for important application functionality like automatic retries.

- `schedule(str) -> ExternResult<()>`: Schedule a function for calling on the next iteration of the conductor's scheduler loop, and thereafter on a schedule defined by the called function. To be schedulable, a function must have the signature `(Schedule) -> Option<Schedule>`, receiving the schedule on which it was called and returning the schedule (if any) on which it wishes to continue to be called. A `Schedule` is defined as:

```
enum Schedule {
    Persisted(String),
    Ephemeral(Duration),
}
```

  Where the value of `Persisted` is a UNIX crontab entry and the value of `Ephemeral` is a duration until the next time. Persisted schedules survive conductor restarts and unrecoverable errors, while ephemeral schedules will not. If `None` is returned instead of `Some(Schedule)`, the function will be unscheduled.

  A scheduled function MUST also be **infallible**; that is, it must be marked with the macro `#[hdk_extern(infallible)]` and return an `Option<Schedule>` rather than an `ExternResult<Option<Schedule>>`. This is because there is no opportunity for user interaction with the result of a scheduled function.

P2P Interaction Agents MUST be able to communicate directly with other agents. They do so simply by making zome calls to them. Holochain systems MUST make this possible by sending a call requests over the network and awaiting a response. For performance reasons the HDK SHOULD also make possible sending of best-effort in parallel signals for which no return result is awaited.

- `call_remote<I>(AgentPubKey, ZomeName, FunctionName, Option<CapSecret>, I) -> ExternResult<ZomeCallResponse> where I: Serialize`: Call a zome function on a target agent and zome, supplying a capability secret and an arguments payload. The return value is defined as:

```
enum ZomeCallResponse {
    Ok(ExternIO),
    Unauthorized(ZomeCallAuthorization, CellId, ZomeName, FunctionName, AgentHash),
    NetworkError(String),
    CountersigningSession(String),
}

enum ZomeCallAuthorization {
    Authorized,
    BadSignature,
    BadCapGrant,
    BadNonce(String),
    BlockedProvenance,
}
```

- `send_remote_signal<I>(Vec<AgentPubKey>, I) -> ExternResult<()> where I: Serialize`: Send a best-effort signal to a list of agents. Implementations SHOULD provide this function, SHOULD implement it by convention as a workflow that sends messages to the receivers as a remote call to a zome function with the signature `recv_remote_signal(SerializedBytes) -> ExternResult<()>` in the same coordinator zome as the function that calls this host function, and MUST NOT await responses from the receivers. Implementations MUST spawn a separate thread to send the signals in order to avoid blocking execution of the rest of the zome function call.

Countersigning In order to safely facilitate the peer interaction necessary to complete a countersigning among multiple agents, the Ribosome and HDK MUST implement the following functions:

- `accept_countersigning_preflight_request(PreflightRequest) -> ExternResult<PreflightRequestAcceptance>`: Lock the local chain to commence a countersigning session. The `PreflightRequestAcceptance` MUST be sent back to the session initiator so that the corresponding entry can be built for everyone to sign. This function MUST be called by every signer in the signing session. The details of how are left to the application developer (although concurrent remote calls are probably the simplest mechanism to distribute and accept preflight requests before the session times out). The preflight request is defined as (see discussion above on countersigning):

```
struct PreflightRequest {
    // The hash of the app entry, as if it were not countersigned. The final
    // entry hash will include the countersigning session data.
    app_entry_hash: EntryHash,
    // The agents that are participating in this countersignature session.
    signing_agents: Vec<(AgentHash, Vec<Role>)>,
    // The optional additional M of N signers. If there are additional
    // signers then M MUST be the majority of N. If there are additional
    // signers then the enzyme MUST be used and is the first signer in BOTH
    // signing_agents and optional_signing_agents.
    optional_signing_agents: Vec<(AgentHash, Vec<Role>)>,
    // The M in the M of N signers. M MUST be strictly greater than than
    // N / 2 and NOT larger than N.
    minimum_optional_signing_agents: u8,
    // The first signing agent (index 0) is acting as an enzyme. If true AND
    // optional_signing_agents are set then the first agent MUST be the same
    // in both signing_agents and optional_signing_agents.
    enzymatic: bool,
    // The window in which countersigning must complete. Session actions
    // MUST all have the same timestamp, which is the session offset.
```

```
    session_times: CounterSigningSessionTimes,
    // The action information that is shared by all agents. Contents depend
    // on the action type, create, update, etc.
    action_base: ActionBase,
    // Optional arbitrary bytes that can be agreed to.
    preflight_bytes: PreflightBytes,
}

struct CounterSigningSessionTimes {
    start: Timestamp,
    end: Timestamp,
}

enum ActionBase {
    Create(CreateBase),
    Update(UpdateBase),
}

struct CreateBase {
    entry_type: EntryType,
}

struct UpdateBase {
    original_action_address: ActionHash,
    original_entry_address: EntryHash,
    entry_type: EntryType,
}

// An arbitrary application-defined role in a session.
struct Role(u8);
```

The return value is defined as:

```
enum PreflightRequestAcceptance {
    Accepted(PreflightResponse),
    UnacceptableFutureStart,
    UnacceptableAgentNotFound,
    Invalid(String),
}

struct PreflightResponse {
    request: PreflightRequest,
    agent_state: CounterSigningAgentState,
    signature: Signature,
}

struct CounterSigningAgentState {
    // The index of the agent in the preflight request agent vector.
    agent_index: u8,
    // The current (frozen) top of the agent's local chain.
    chain_top: ActionHash,
    // The action sequence of the agent's chain top.
    action_seq: u32,
}
```

- `session_times_from_millis(u64) -> ExternResult<CounterSigningSessionTimes>`: Create the session times that are included in the `PreflightRequest` and bound the countersigning session temporally. This function returns a session start timestamp is "now" from the perspective of the system clock of the session initiator calling this function, and a session end timestamp that is "now" plus the given number of milliseconds. The countersigning parties will check these times against their own perspectives of "now" as part of accepting

the preflight request, so all system clocks need to be roughly aligned, and the ambient network latency must fit comfortably within the session duration.

Cryptography The HDK MUST provide mechanisms for agents to sign and check the signatures of data. It SHOULD provide mechanisms to encrypt and decrypt data and return pseudo-random data:

- `sign<D>(AgentPubKey, D) -> ExternResult<Signature> where D: Serialize`: Given a public key, request from the key-management system a signature for the given data by the corresponding private key.
- `verify_signature<I>(AgentPubKey, Signature, I) -> ExternResult<bool> where I: Serialize`: (see HDI)
- `x_salsa20_poly1305_shared_secret_create_random(Option<XSalsa20Poly1305KeyRef>) -> ExternResult<XSalsa20Poly1305KeyRef>`: Generate a secure random shared secret suitable for encrypting and decrypting messages using NaCl's secretbox[27] encryption algorithm, and store it in the key-management system. An optional key reference ID may be given; if this ID already exists in the key-management system, an error will be returned. If no ID is given, one will be generated and returned. The key reference is defined as:

  ```
  struct XSalsa20Poly1305KeyRef(u8);
  ```

- `x_salsa20_poly1305_encrypt(XSalsa20Poly1305KeyRef, Vec<u8>) -> ExternResult<XSalsa20Poly1305EncryptedD` Given a reference to a symmetric encryption key stored in the key-management service, request the encryption of the given bytes with the key. The return value is defined as:

  ```
  struct XSalsa20Poly1305EncryptedData {
      nonce: [u8; 24],
      encrypted_data: Vec<u8>,
  }
  ```

- `x_salsa20_poly1305_decrypt(XSalsa20Poly1305KeyRef, XSalsa20Poly1305EncryptedData) -> ExternResult<Option<Vec<u8>>>`: Given a reference to a symmetric encryption key, request the decryption of the given bytes with the key.
- `create_x25519_keypair() -> ExternResult<X25519PubKey>`: Create an X25519 key pair suitable for encrypting and decrypting messages using NaCl's box[28] algorithm, and store it in the key-management service. The return value is defined as:

  ```
  struct X25519PubKey([u8; 32]);
  ```

- `x_25519_x_salsa20_poly1305_encrypt(X25519PubKey, X25519PubKey, Vec<u8>) -> ExternResult<XSalsa20Poly1305EncryptedData>`: Given X25519 public keys for the sender and recipient, attempt to encrypt the given bytes via the box algorithm using the sender's private key stored in the key-management service and the receiver's public key.
- `x_25519_x_salsa20_poly1305_decrypt(X25519PubKey, X25519PubKey, Vec<u8>) -> ExternResult<XSalsa20Poly1305EncryptedData>`: Given X25519 public keys for the recipient and sender, attempt to decrypt the given bytes via the box algorithm using the sender's public key and the receiver's private key stored in the key-management service.
- `ed_25519_x_salsa20_poly1305_encrypt(AgentPubKey, AgentPubKey, XSalsa20Poly1305Data) -> ExternResult<XSalsa20Poly1305EncryptedData>`: Attempt to encrypt a message using the box algorithm, converting the Ed25519 signing keys of the sender and recipient agents into X25519 encryption keys. This procedure is not recommended[29] by the developers of libsodium, the NaCl implementation used by Holochain.
- `ed_25519_x_salsa20_poly1305_decrypt(AgentHash, AgentHash, XSalsa20Poly1305EncryptedData) -> ExternResult<XSalsa20Poly1305Data>`: Attempt to decrypt a message using the box algorithm, converting the Ed25519 signing keys of the recipient and sender agents into X22519 encryption keys. This procedure is not recommended by the developers of libsodium, the NaCl implementation used by Holochain.

User Notification The HDK SHOULD provide a way for zome code to notify the application user of events. To start with we have implemented a system where signals can be emitted from a zome:

- `emit_signal<I>(I) -> ExternResult<()> where I: Serialize`: Emit the bytes as a signal to listening clients.

---

[27] See https://nacl.cr.yp.to/secretbox.html.
[28] See https://nacl.cr.yp.to/box.html.
[29] See https://doc.libsodium.org/quickstart#how-can-i-sign-and-encrypt-using-the-same-key-pair.

Anchors and Paths A content-addressable store, accessible only by the hashes of stored items, is difficult to search because of the sparse nature of the hashes. Holochain's graph DHT makes it much easier to retrieve related information via the affordance of links that can be retrieved from a given hash address. A powerful pattern that can be built on top of links is what we call anchors and, more generally, paths. These patterns rely on the idea of starting from a known hash value that all parties can compute, and placing links from that hash to relevant entries. So, for example, one could take the hash of the string `#funnycats` and add links on that hash to all posts in a social media app that contain that hashtag. This pattern, the anchor pattern, affords the discovery of arbitrary collections or indexes of content-addressed data. The path pattern simply generalizes this to creating an arbitrary hierarchical tree of known values off of which to create links in the DHT.

A note about efficiency: Because every attempt to create an entry or link results in another record that needs to be validated and stored, implementations of this pattern SHOULD attempt to be idempotent when creating anchors or tags; that is, they should check for the prior existence of the links and entries that would be created before attempting to create them. It is both semantically and practically appropriate to hash the anchor or path string in-memory and wrap it in an `ExternalHash` for link bases and targets, as this avoids the the overhead of creating an entry, and the hash, which exists only in memory, can truly be said to be external to the DHT.

Anchors The HDK MAY provide functions to compute hashes from, and attach links to, known strings using the anchor pattern, which creates a two-level hierarchy of anchor types and anchors from which to link entries:

- `anchor(ScopedLinkType, String, String) -> ExternResult<EntryHash>`: Create an anchor type and/or anchor, linking from the 'root' anchor to the anchor type, and from the type to the anchor (if given). Return the anchor's hash.
- `list_anchor_type_addresses(ScopedLinkType) -> ExternResult<Vec<AnyLinkableHash>>`: Retrieve the hashes of all anchor types created in the DHT. This permits ad-hoc runtime creation and discovery of anchor types.
- `list_anchor_addresses(LinkType, String) -> ExternResult<Vec<AnyLinkableHash>>`: Retrieve the hashes of all anchors for a given type.

Paths The HDK MAY provide functions to compute hashes from, and attach links to, known strings using the path pattern, which affords an arbitrary hierarchy of known hashes off of which to link entries:

```
struct Path(Vec<Component>);

struct Component(Vec<u8>);

struct TypedPath {
    link_type: ScopedLinkType,
    path: Path,
}
```

- `root_hash() -> ExternResult<AnyLinkableHash>`: Compute and return the root hash of the path hierarchy, from which one can search for any previously registered paths; e.g. `path_children(path_root())` will find all top-level paths. The bytes that make up the root node SHOULD be reasonably unique and well-known in order to avoid clashes with application data; our implementation uses the bytes `[0x00, 0x01]`.
- `Path::path_entry_hash() -> ExternResult<EntryHash>`: Return the hash of a given path, which can then be used to search for items linked from that part of the path tree. Note that, in our implementation, entries are generated in memory and hashed but not recorded to the DHT.
- `TypedPath::ensure() -> ExternResult<()>`: Create links for every component of the path, if they do not already exist. This method SHOULD attempt to be idempotent.
- `TypedPath::exists() -> ExternResult<bool>`: Look for the existence in the DHT of all the path's components, and return true if all components exist.
- `TypedPath::children() -> ExternResult<Vec<Link>>`: Retrieve the links to the path's direct descendants. Note that these are *not* links to app-defined data but to nodes in the path hierarchy. App-defined data is expected to be linked to and retrieved from the path node's hash via the HDK's `create_link` and `get_links` functions.
- `TypedPath::children_details() -> ExternResult<Vec<LinkDetails>>`: Retrieve details about the links to the path's direct descendants. This is equivalent to the HDK's `get_link_details` function.

- `TypedPath::children_paths() -> ExternResult<Vec<TypedPath>>`: Retrieve the path's direct descendant nodes in the hierarchy as `TypedPath` values.

State Management via Workflows

The previous section describes the functions exposed to, and callable from, DNA code, such that developers can implement the integrity of a DNA (its structure and validation rules) and the functions that can be called on that integrity for authoring source chain entries and coherently retrieving that information from the application's DHT. This section describes the implementation requirements for recording and storing all aspects of Holochain's state. This includes agents' source-chain entries, the portion of the DHT data a node is holding, configuration data, caches, etc.

Ontology of Workflows

While a properly defined and implemented Holochain system must necessarily be robust enough to handle data from an incorrectly operating peer, it is nevertheless a more productive experience for everyone if all nodes in a network change their states according to the same process. There are also cases in which an incorrect implementation may result in unrecoverable corruption to state.

Hence, we must define an **ontology of workflows**. A Workflow is defined ontologically as a process which:

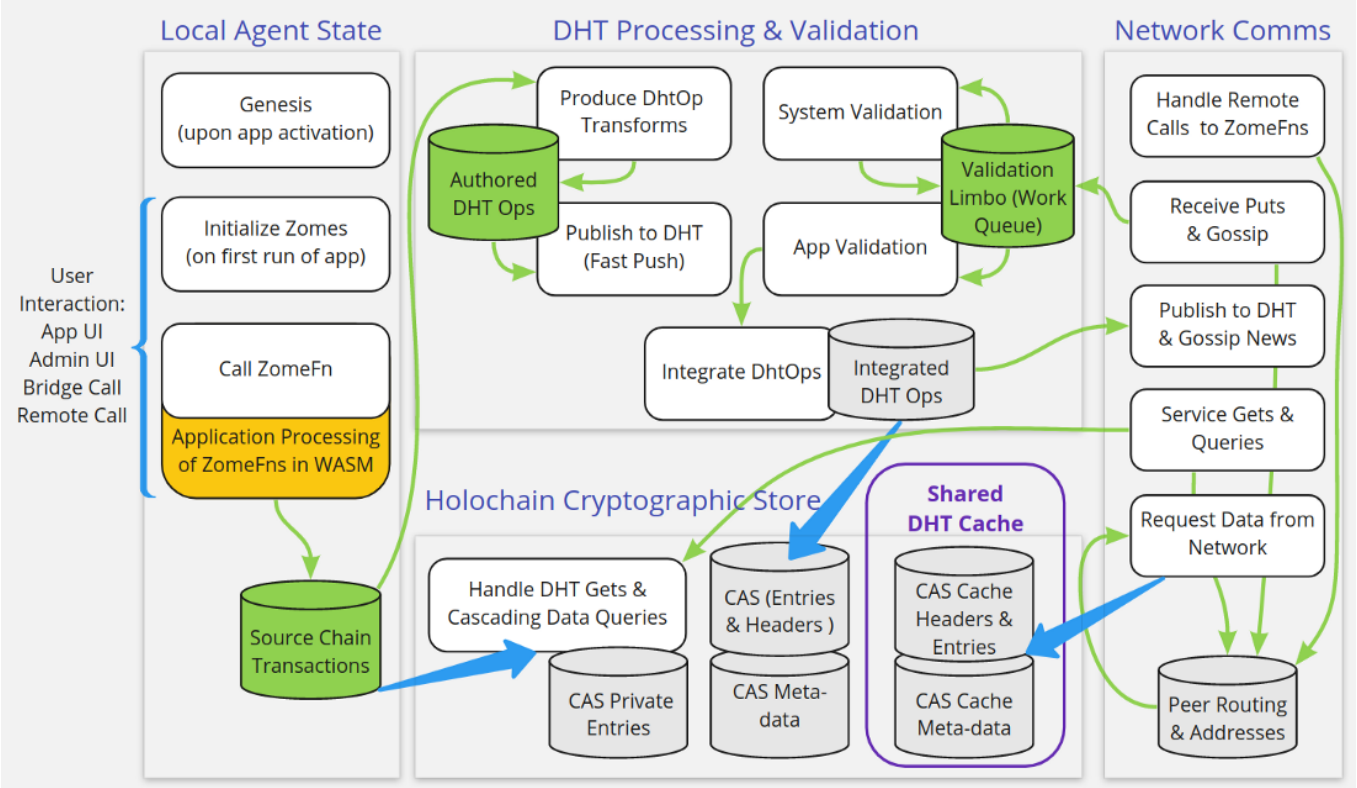
1. Accesses and potentially changes Holochain state,
2. Receives an ephemeral input context necessary to do its job,
3. Optionally triggers other workflows to follow up on the newly changed state, potentially including another iteration of itself, and
4. Optionally returns a value which can be passed to a waiting receiver.

It is important to note that Workflows are reifications of the inherent physics of Holochain; that is, the concept of a Workflow is demanded by the kinds of state changes a Holochain implementation is expected to make.

The properties which hold for all Workflows are:

- A Workflow MUST operate only on an aspect of local Holochain state, and MUST NOT make assumptions about the value of any aspects of Holochain state it does not operate on, whether local or remote.
- A Workflow MUST NOT leave the state it operates on in a corrupt condition it fails for any reason, whether the failure is expected (such as validation failure) or unexpected (such as hardware malfunction). This means that it MUST either make an atomic and valid state change or make no state change at all.
  - Corollary: a Workflow MUST treat the Holochain state upon which it operates as the ultimate source of truth about itself, which means that any other state it builds up during execution MUST be treated as incidental and disposable; that is, it MUST able to successfully recover from a failure and correctly change cryptographic state even if incidental state is lost.
- A Workflow MUST have direct access to the state it is manipulating so that it may observe it immediately before changing it, to avoid race conditions between Workflows that operate on the same state.
- A Workflow MUST operate on only one aspect of Holochain state, an aspect being defined as a portion of state which can be changed independently of other aspects.
- A change to Holochain state MUST be expressed monotonically. (This is merely a restatement of the fact that all changes of Holochain state are by nature monotonic.)
- If a Workflow operates on a contentious aspect of state, it MUST either:
  - Be a singleton (that is, only one instance of the Workflow is permitted to run at any time), or
  - Be permitted to run concurrently with another instance of itself and:
    1. Take a snapshot of the current value of the state when it begins to build a state change to be written,
    2. Check the current value of the state immediately before attempting to write a change, and
    3. Discard its attempted state change if the value of the state is now different from the snapshot.

We intend to publish an addendum which enumerates the necessary workflows, the aspects of Holochain state upon which they operate, and the ways in which they operate. In the meantime, the following diagram is a simplified overview.



Shared Data (rrDHT)

In this section we detail some important implementation details of Holochain's graph DHT.

DHT Operations

Structure of DhtOps You can think of a DHT operation as having this sort of grammar:

$BasisHash, OperationType, Payload$

Where:

- *BasisHash* is the address to which an operation is being applied.
- *OperationType* is the type of operation a node is responsible for performing.
- *Payload* is the self-proving structure which contains the data needed to perform the operation. In all cases this includes the `Action`; it may also include the `Entry` if such a thing exists for the action type and if it is required to validate and perform the operation.

The technical implementation below of the human-friendly grammar above compresses and drops unnecessary items where possible. There are a couple of *OperationType* where we can drop the entry (but never the action); in these cases we can reduce all the data down to `Action` + an *OperationType* enum struct which usually contains the entry.

The basis hash (or hash neighborhood we're sending the operation to) can be derived from the payload using the `dht_basis` function outlined below.

```
enum DhtOp {
    ChainOp(ChainOp),
    WarrantOp(WarrantOp),
}

impl DhtOp {
    fn dht_basis(self) -> AnyLinkableHash {
    match self {
        Self::ChainOp(op) => op.dht_basis(),
        Self::WarrantOp(op) => op.dht_basis(),
    }
```

```
  }
}

// Ops that start with `Store` store new addressable content at the basis hash.
// Ops starting with `Register` attach metadata to the basis hash.
enum ChainOp {
    StoreRecord(Signature, Record, RecordEntry),
    StoreEntry(Signature, NewEntryAction, Entry),
    RegisterAgentActivity(Signature, Action),
    RegisterUpdatedContent(Signature, action::Update, RecordEntry),
    RegisterUpdatedRecord(Signature, action::Update, RecordEntry),
    RegisterDeletedBy(Signature, action::Delete),
    RegisterDeletedEntryAction(Signature, action::Delete),
    RegisterAddLink(Signature, action::CreateLink),
    RegisterRemoveLink(Signature, action::DeleteLink),
}

impl ChainOp {
    fn dht_basis(self) -> AnyLinkableHash {
        match self {
            StoreRecord(_, action, _) => hash(action),
            StoreEntry(_, action, _) => hash(action.entry),
            RegisterAgentActivity(_, action) => header.author(),
            RegisterUpdatedContent(_, action, _) => action.original_entry_address,
            RegisterUpdatedRecord(_, action, _) => action.original_action_address,
            RegisterDeletedBy(_, action) => action.deletes_address,
            RegisterDeletedEntryAction(_, action) => action.deletes_entry_address,
            RegisterAddLink(_, action) => action.base_address,
            RegisterRemoveLink(_, action) => action.base_address,
        }
    }
}

struct WarrantOp(Signed<Warrant>);

struct Warrant {
    proof: WarrantProof,
    // The author of the warrant.
    author: AgentHash,
    timestamp: Timestamp,
}

enum WarrantProof {
    ChainIntegrity(ChainIntegrityWarrant),
}

impl WarrantProof {
    fn dht_basis(self) -> AnyLinkableHash {
        self.action_author()
    }

    fn action_author(self) -> AgentPubKey {
        match self {
            Self::ChainIntegrity(w) => match w {
                ChainIntegrityWarrant::InvalidChainOp { action_author, .. } => action_author,
                ChainIntegrityWarrant::ChainFork { chain_author, .. } => chain_author,
            },
        }
```

```
    }
}

enum ChainIntegrityWarrant {
    InvalidChainOp {
        action_author: AgentHash,
        action: (ActionHash, Signature),
        validation_type: ValidationType,
    },
    ChainFork {
        chain_author: AgentHash,
        action_pair: ((ActionHash, Signature), (ActionHash, Signature)),
    },
}
```

Uniquely Hashing Dht Operations When items are gossiped/published to us, we SHOULD be able to quickly check:

1. Do we consider ourselves an authority for this basis hash?
2. Have we integrated it yet?

and quickly take appropriate action.

To facilitate this, implementations MUST define a reproducible way of hashing DHT operations. The following code outlines the minimal necessary contents to create the correct operation hash. The basic procedure for all operations is:

1. Drop all data from the operation except the action.
2. Wrap the action in a variant of a simplified enum representing the minimal data needed to uniquely identify the operation, thus allowing it to be distinguished from other operations derived from the same action.
3. Serialize and hash the simplified value.

```
// Parallels each variant in the `ChainOp` enum, only retaining the minimal data
// needed to produce a unique operation hash.
enum ChainOpUniqueForm {
    StoreRecord(Action),
    StoreEntry(NewEntryAction),
    RegisterAgentActivity(Action),
    RegisterUpdatedContent(action::Update),
    RegisterUpdatedRecord(action::Update),
    RegisterDeletedBy(action::Delete),
    RegisterDeletedEntryAction(action::Delete),
    RegisterAddLink(action::CreateLink),
    RegisterRemoveLink(action::DeleteLink),
}

// Conversion implementation for all the types involved in a `DhtOp`.

impl ChainOp {
    fn as_unique_form(self) -> ChainOpUniqueForm {
        match self {
        Self::StoreRecord(_, action, _) => ChainOpUniqueForm::StoreRecord(action),
        Self::StoreEntry(_, action, _) => ChainOpUniqueForm::StoreEntry(action),
        Self::RegisterAgentActivity(_, action) => {
            ChainOpUniqueForm::RegisterAgentActivity(action)
        }
        Self::RegisterUpdatedContent(_, action, _) => {
            ChainOpUniqueForm::RegisterUpdatedContent(action)
        }
        Self::RegisterUpdatedRecord(_, action, _) => {
            ChainOpUniqueForm::RegisterUpdatedRecord(action)
        }
        Self::RegisterDeletedBy(_, action) => ChainOpUniqueForm::RegisterDeletedBy(action),
        Self::RegisterDeletedEntryAction(_, action) => {
```

```
            ChainOpUniqueForm::RegisterDeletedEntryAction(action)
        }
        Self::RegisterAddLink(_, action) => ChainOpUniqueForm::RegisterAddLink(action),
        Self::RegisterRemoveLink(_, action) => ChainOpUniqueForm::RegisterRemoveLink(action),
        }
    }
}

trait HashableContent {
    type HashType: HashType;

    fn hash_type(self) -> Self::HashType;

    fn hashable_content(self) -> HashableContentBytes;
}

impl HashableContent for DhtOp {
    type HashType = hash_type::DhtOp;

    fn hash_type(self) -> Self::HashType {
        hash_type::DhtOp
    }

    fn hashable_content(self) -> HashableContentBytes {
        match self {
            DhtOp::ChainOp(op) => op.hashable_content(),
            DhtOp::WarrantOp(op) => op.hashable_content(),
        }
    }
}

impl HashableContent for ChainOp {
    type HashType = hash_type::DhtOp;

    fn hash_type(self) -> Self::HashType {
        hash_type::DhtOp
    }

    fn hashable_content(self) -> HashableContentBytes {
        HashableContentBytes::Content(
            self.as_unique_form().try_into()
        )
    }
}

impl HashableContent for WarrantOp {
    type HashType = hash_type::DhtOp;

    fn hash_type(&self) -> Self::HashType {
        hash_type::DhtOp
    }

    fn hashable_content(&self) -> HashableContentBytes {
        self.warrant().hashable_content()
    }
}

impl HashableContent for Warrant {
```

```
    type HashType = holo_hash::hash_type::Warrant;

    fn hash_type(&self) -> Self::HashType {
        Self::HashType::new()
    }

    fn hashable_content(&self) -> HashableContentBytes {
        HashableContentBytes::Content(self.try_into())
    }
}
```

Changing States of DHT Content

As a simple accumulation of data (DHT operations) attached to their respective basis addresses, a Holochain DHT exhibits a logical monotonicity[30]. The natural consequence of this property is that any two peers who receive the same set of DHT operations will arrive at the same database state without need of a coordination protocol.

While the monotonic accumulation of operations is the most fundamental truth about the nature of DHT data, it is nevertheless important for the goal of ensuring Holochain's fitness for application development that we give the operations further meaning. This happens at two levels:

- **Data and metadata**: The immediate result of applying an operation to a basis address is that data or metadata is now available for querying at that basis address. This takes the form of:
  - A record as primary data,
  - An entry and its set of creation actions as primary data, presented as the Cartesian product $\{e\}\times\{h_1,\ldots,h_n\}$,
  - Record updates and deletes as metadata,
  - Link creations and deletions as metadata,
  - Agent activity as a tree of metadata,
  - Validation status as metadata, and
  - Warrants as metadata.
- **CRUD**: The total set of metadata on a basis address can always be accessed and interpreted as the application developer sees fit (see `get_details` in the DHT Data Retrieval section of this appendix), but certain opinionated interpretations of that set are useful to provide as defaults[31]:
  - The set difference between all record creates/updates and deletes that refer to them can be accessed as a "tombstone" set that yields the list of non-deleted records, the liveness of an entry or record, or the earliest live non-deleted record for an entry (see `get` in the DHT Data Retrieval section).
  - The set difference between all link creates and link deletes that refer to them can be accessed as a tombstone set that yields the list of non-deleted links (see `get_links` in the DHT Data Retrieval section).

Validation and Liveness on the DHT The first task before changing the DHT to include a new piece of data is to validate the operation according to both system-level and application-specific rules. Additionally, an operation MUST be accompanied by a valid provenance signature that matches the public key of its author.

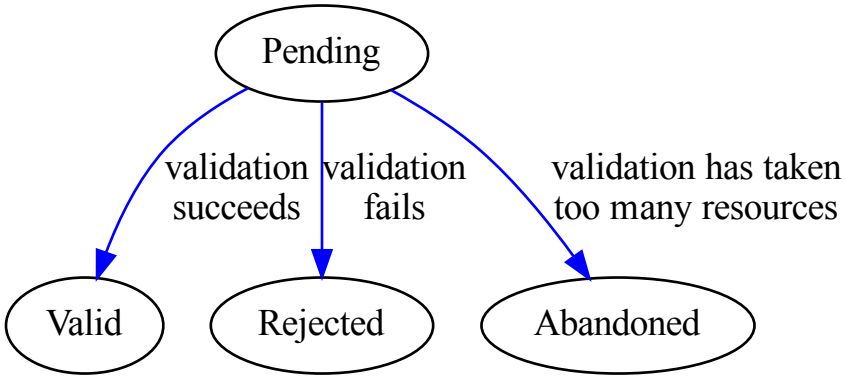
DHT operations whose validation process has been abandoned are not gossiped. There are two reasons to abandon validation. Both have to do with consuming too much resources.

1. It has stayed in our validation queue too long without being able to resolve dependencies.
2. The app validation code used more resources (CPU, memory, bandwidth) than we allocate for validation. This lets us address the halting problem of validation with infinite loops.

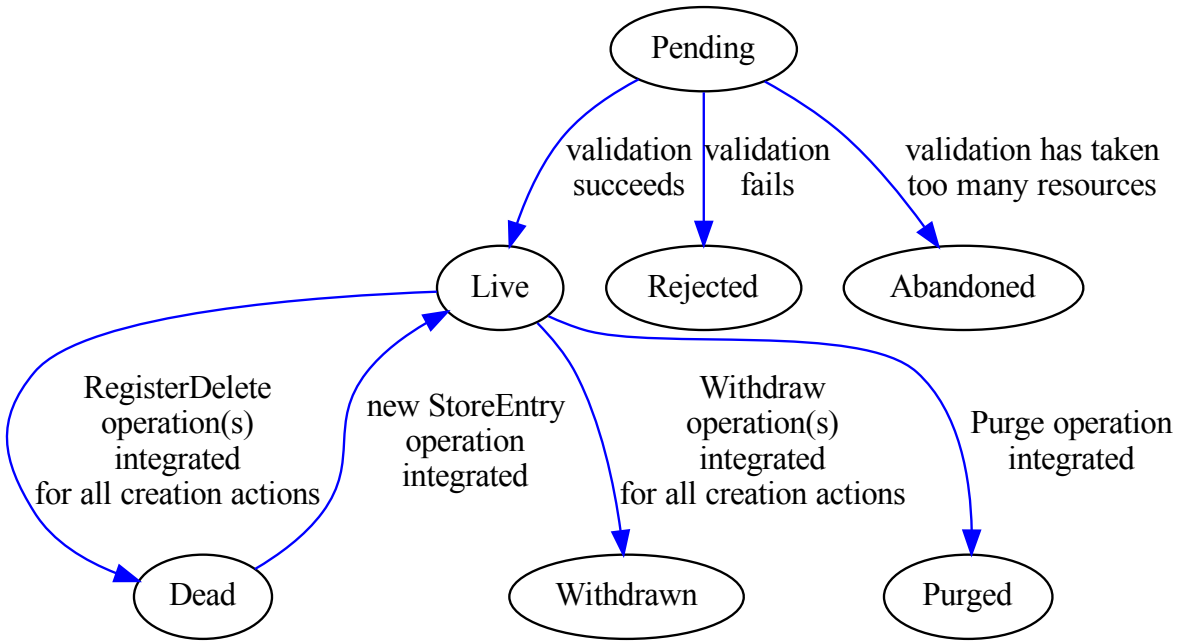
---

[30] *Keeping CALM: When Distributed Consistency is Easy*, Joseph M Hellerstein and Peter Alvaro https://arxiv.org/abs/1901.01930.

[31] While this interpretation indicates that the set of metadata can be validly seen as operations in a simple operation-based conflict-free replicated data type (CRDT) (see https://crdt.tech), we have chosen not to use this term in order to avoid overlaying of preconceptions formed by more capable CRDTs.



Entry Liveness Status The 'liveness' status of an Entry at its DHT basis is changed in the following ways:



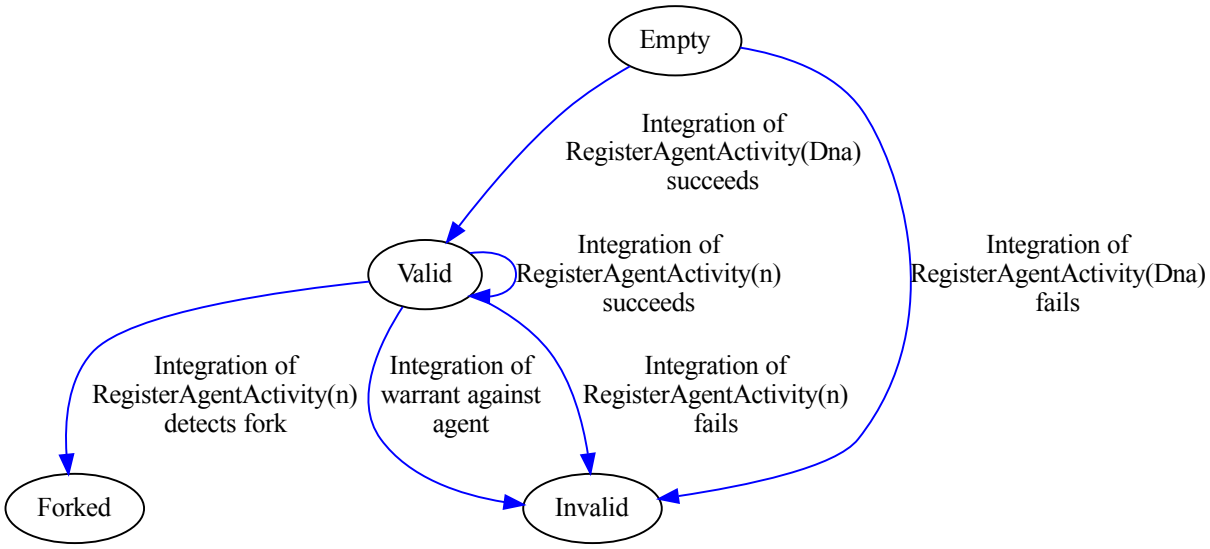
An Entry is considered `Dead` when ALL of the valid creation Actions which created it have been marked as deleted by valid deletion Actions; that is, `Live` entails a non-empty result of a set difference between the creation Action hashes and the `deletes_address` field of the deletion Action hashes stored at the entry's basis.

`Withdrawn` and `Purged` are placeholders for possible future features:

- `Withdrawn`: The record has been marked by its author as such, usually to correct an error (such as accidental forking of their chain after an incomplete restoration from a backup). The same set difference rules apply to `Live`/`Withdrawn` as to `Live`/`Dead`.
- `Purged`: The addressable content has been erased from the CAS database, possibly by collective agreement to drop it – e.g., for things that may be illegal or unacceptable to hold (e.g., child pornography).

The process of changing data to these two states is unimplemented.
Action Liveness Status An Action is considered `Dead` only after a `RegisterDeletedBy` operation which references the Action's has has been integrated at the Action's basis.
Link Liveness Status A link is considered `Dead` only after at least one `RegisterDeleteLink` operation which references the `CreateLink` action has been integrated at the link base's basis.
Agent Status An Agent's status, which can be retrieved from the Agent ID basis (that is, the Agent's public key), is a composite of:

- Liveness of `AgentID` Entry, according to the above rules defined in Entry Liveness Status
- Validity of every Source Chain action (that is, whether all `RegisterAgentActivity` operations are valid)
- Linearity of Source Chain (that is, whether there are any branches in the Source chain, also determined during integration of `RegisterAgentActivity` operations)
- Presence of valid Warrants received from other authorities via `WarrantOp` DHT operations



P2P Networking

A robust networking implementation for Holochain involves three layers:

1. The Holochain P2P networking layer, which is designed around the peer-to-peer communication needs of agents in a DNA and the building of the DNA's graph DHT,
2. An underlying P2P layer that handles the fact that a Holochain node will be managing communication on behalf of potentially multiple agents in multiple networks, and will be connecting with other nodes, any of which may be running non-overlapping sets of DNAs, and
3. A transport-level layer that supplies and interprets transport.

Thus, from a networking perspective, there is the view of a single DNA (which is its own network), in which more than one local agent may be participating, but there is also the view of an agent belonging to many DNAs at the same time.

Because the same DHT patterns that work at the level of a Holochain DNA sharing storage of application data also work to solve the problem of a shared database holding updates to a routing table of peer addresses, we have implemented a generalized P2P DHT solution and built the higher-level Holochain P2P networking needs on top of that lower level. Below we describe the high-level requirements and network messages for Holochain, followed by the lower-level requirements and network messages that carry the higher-level ones.

High-Level Networking (Holochain P2P)

There is a number of network messages that are sent and handled as a direct result of HDK functions or callbacks being executed in a zome call. These calls are all directed at specific agents in a DNA, either because they are explicitly targeted in the call (e.g., `CallRemote`) or because the agent has been determined to be responsible for holding data

on the DHT. And in most cases, these message types expect a response. Hence, all Holochain message types are implemented as the lower-level `Call` and `CallResp` message pairs, with the exception of `ValidationReceipts` and `CountersigningSessionNegotiation`, which are implemented as lower-level `Broadcast` messages.

Note that the `ValidationReceipts` message is sent back to an authoring agent as a result of a node validating a `DhtOp`, and `Publish` messages are sent by an author node as a result of committing any chain action and transforming it into `DhtOp`s.

The following messages types MUST be implemented. In our implementation, they are all defined as variants of a `WireMessage` enum which are wrapped in lower-level Kitsune messages before being serialized and sent via the network transport implementation. The message payloads are defined as unnamed structs which comprise the data portion of the enum variants.

- `CallRemote`: Call a zome function in a remote cell in the same DHT network, supplying a valid capability secret if required.
  - **Notes**: On the remote side, implementations MUST enforce permissions:
    * They MUST check all the active capability grants for the function being called against the capability claim being exercised.
    * They MUST check the `nonce` and `expires_at` field in order to detect replay attacks and MUST reject the call if the nonce has been seen for the same agent and/or the `expires_at` timestamp has passed.
    * They MUST also check that the signature is valid for the supplied `from_agent` public key and the call data.

    On the sending side, implementations MUST generate a nonce that is sufficiently unguessable, and an expiry time that is sufficiently short, to effectively thwart replay attacks while also avoiding spurious timeout failures.

    As a performance optimization, implementations MUST also implement a `CallRemoteMulti` message type which provide a workflow to send the same `CallRemote` message to multiple remote cells in parallel. This message type differs in these ways:

    * The `signature` field is removed.
    * The `to_agent` field is replaced with a `to_agents: Vec<(Signature, AgentPubKey)>`, wherein each signature is valid for a serialized `ZomeCallUnsigned` struct with the given `AgentPubKe` in the `to_agent` field.

    Implementations MUST respond with the function call's return value, if the function call was allowed and successful, or an error.

    Implementations SHOULD terminate function execution on the remote node if it has exhausted an execution cost limit, to prevent denial-of-service attacks against the receiver.
  - **Payload**: The payload is defined as:

```
{
    zome_name: ZomeName,
    fn_name: FunctionName,
    from_agent: AgentPubKey,
    signature: Signature,
    to_agent: AgentPubKey,
    cap_secret: Option<CapSecret>,
    data: Vec<u8>,
    nonce: [u8; 32],
    expires_at: Timestamp,
}
```

    The zome function payload is serialized into a `Vec<u8>` in the `data` field. The signature is generated by copying the above fields into the following struct, serializing it, and signing the hash of the serialized bytes.publish

```
struct ZomeCallUnsigned {
    provenance: AgentPubKey,
    cell_id: CellId,
    zome_name: ZomeName,
```

```
    fn_name: FunctionName,
    cap_secret: Option<CapSecret>,
    payload: ExternIO,
    nonce: Nonce256Bits,
    expires_at: Timestamp,
}
```

  - **Response**: The expected response is a `ZomeCallResponse`, which is defined above in the HDK section.

- `ValidationReceipts`: Send validation receipts to the node that authored the DHT operations to which the receipts apply, as a result of integrating published operations. publish
  - **Payload**: The payload is defined as:

```
ValidationReceipts {
    receipts: Vec<SignedValidationReceipt>,
}
```

    A receipt is defined as:

```
struct SignedValidationReceipt {
    receipt: ValidationReceipt,
    // Because multiple agents on the remote node
    // may claim authority for the same DHT basis hash,
    // this field MUST be plural.
    validators_signatures: Vec<Signature>,
}

struct ValidationReceipt {
    // The hash of the DHT operation to which this receipt applies.
    dht_op_hash: DhtOpHash,
    // The result of validating the operation.
    validation_status: ValidationStatus,publish
    // The remote agents who have validated the operation.
    // As with `validators_signatures` above,
    // this field MUST be plural.
    validators: Vec<AgentPubKey>,
    // The time when the operation was integrated.
    when_integrated: Timestamp,
}
```

- `Get`: Request addressable content and/or metadata stored at the basis hash.
  - **Notes**: At the receiver side, implementations MUST return only integrated data unless pending data has been requested.
  - **Payload**: The payload is defined as:

```
{
    dht_hash: AnyDhtHash,
    options: GetOptions,
}

struct GetOptions {
    request_type: GetRequest,
}

enum GetRequest {
    // Get integrated content and metadata.
    All,
    // Get only addressable content.
    Content,
    // Get only metadata.
    Metadata,
```

```
    // Get content even if it hasn't been integrated.
    Pending,
}
```

– **Response**: The expected response is defined as:

```
enum WireOps {
    Entry(WireEntryOps),
    Record(WireRecordOps),
    // A warrant in place of data in the case that the data is invalid.
    Warrant(WarrantOp),
}

struct WireEntryOps {
    // Any actions that created this entry.
    creates: Vec<Judged<WireNewEntryAction>>,
    // Any deletes that deleted this entry's creation actions.
    deletes: Vec<Judged<WireDelete>>,
    // Any updates on this entry's creation actions pointing to new
    // entries. This is different from updates that created this entry
    // listed in the `creates` field.
    updates: Vec<Judged<WireUpdateRelationship>>,
    // The entry data shared across all actions.
    entry: Option<EntryData>,
}

// Data with an optional validation status.
struct Judged<T> {
    // The data that the status applies to.
    data: T,
    // The validation status of the data.
    status: Option<ValidationStatus>,
}

enum WireNewEntryAction {
    Create(WireCreate),
    Update(WireUpdate),
}

// The following are compact representations of similarly named Action
// structs. They do not need the entry type and hash, as these can be
// derived from `WireEntryOps::entry`.
struct WireCreate {
    timestamp: Timestamp,
    author: AgentPubKey,
    action_seq: u32,
    prev_action: ActionHash,
    signature: Signature,
    weight: EntryRateWeight,
}

struct WireUpdate {
    timestamp: Timestamp,
    author: AgentPubKey,
    action_seq: u32,
    prev_action: ActionHash,
    original_entry_address: EntryHash,
    original_action_address: ActionHash,
    signature: Signature,
```

```
    weight: EntryRateWeight,
}

struct WireDelete {
    delete: Delete,
    signature: Signature,
}

struct WireUpdateRelationship {
    timestamp: holochain_zome_types::timestamp::Timestamp,
    author: AgentPubKey,
    action_seq: u32,
    prev_action: ActionHash,
    original_action_address: ActionHash,
    new_entry_address: EntryHash,
    new_entry_type: EntryType,
    signature: Signature,
    weight: EntryRateWeight,
}

// Entry data shared across all CRUD actions.
struct EntryData {
    entry: Entry,
    entry_type: EntryType,
}

struct WireRecordOps {
    // The action this request was for.
    action: Option<Judged<SignedAction>>,
    // Any deletes on the action.
    deletes: Vec<Judged<WireDelete>>,
    // Any updates on the action.
    updates: Vec<Judged<WireUpdateRelationship>>,
    // The entry if there is one.
    entry: Option<Entry>,
}
```

- `GetMeta`: Request all metadata stored at the given basis hash.
  - **Payload**: The payload is defined as:

    ```
    {
        dht_hash: AnyDhtHash,
    }
    ```

  - **Response**: The response is defined as:

    ```
    struct MetadataSet {
        // Actions that created or updated an entry. These are the actions
        // that show the entry exists.
        actions: BTreeSet<TimedActionHash>,
        invalid_actions: BTreeSet<TimedActionHash>,
        deletes: BTreeSet<TimedActionHash>,
        updates: BTreeSet<TimedActionHash>,
        // The status of an entry from an authority. This is simply a faster
        // way of determining if there are any live actions on an entry.
        // If the basis hash is not for an entry, this will be empty.
        entry_dht_status: Option<EntryDhtStatus>,
    }
    ```

- `GetLinks`: Request link creation and deletion actions stored at the basis hash, optionally filtered by a query

predicate.

  - **Payload**: The payload is defined as:

```
{
    query: WireLinkQuery,
}

struct WireLinkQuery {
    base: AnyLinkableHash,
    link_type: LinkTypeFilter,
    tag_prefix: Option<LinkTag>,
    before: Option<Timestamp>,
    after: Option<Timestamp>,
    author: Option<AgentPubKey>,
}

enum LinkTypeFilter {
    Types(Vec<(ZomeIndex, Vec<LinkType>)>),
    Dependencies(Vec<ZomeIndex>),
}
```

  - **Response**: The response is defined as:

```
struct WireLinkOps {
    // Link creation actions that match the query.
    creates: Vec<WireCreateLink>,
    // Link deletion actions that match the query.
    deletes: Vec<WireDeleteLink>,
}
```

- `CountLinks`: Request only the *count* of *live* link creation actions, optionally filtered by the query predicate.
  - **Payload**: The payload is defined as:

```
{
    query: WireLinkQuery,
}
```

  - **Response**: The response is defined as a `Vec<ActionHash>`, where the included action hashes are non-tombstoned link creation actions matching the query.

- `GetAgentActivity`: Request information about the given agent's activity, optionally filtered by the given predicate, which is defined above in the HDK section, and including or excluding data specified by the options,
  - **Payload**: The payload is defined as:

```
{
    agent: AgentPubKey,
    query: ChainQueryFilter,
    options: GetActivityOptions,
}

struct GetActivityOptions {
    // Include the agent activity actions in the response.
    include_valid_activity: bool,
    // Also include any rejected actions in the response.
    include_rejected_activity: bool,
    // Include warrants in the response.
    include_warrants: bool,
    // Include the full signed actions and hashes in the response
    // instead of just the hashes.
    include_full_actions: bool,
}
```

  - **Response**: The response is defined as `AgentActivityResponse<ActionHash>`, where `AgentActivityResponse<T>` is defined as:

```
struct AgentActivityResponse<T = SignedActionHashed> {
    agent: AgentPubKey,
    valid_activity: ChainItems<T>,
    rejected_activity: ChainItems<T>,
    status: ChainStatus,
    highest_observed: Option<HighestObserved>,
    warrants: Vec<Warrant>,
}

enum ChainItems<T = SignedActionHashed> {
    Full(Vec<T>),
    Hashes(Vec<(u32, ActionHash)>),
    // In the case that the value of `GetActivityOptions` specified that
    // the given type of agent activity was not wanted.
    NotRequested,
}

struct HighestObserved {
    // The highest sequence number observed.
    action_seq: u32,
    // Hash(es) of any action(s) claiming to be at this action sequence.
    // Any vector with a cardinality > 1 indicates a forked chain, which
    // will be corroborated by the information contained in
    // `AgentActivityResponse<T>::status` and
    // `AgentActivityResponse<T>::warrants`.
    hash: Vec<ActionHash>,
}
```

- `MustGetAgentActivity`: Request a contiguous sequence of agent activity actions for the given agent, bounded by the specified `ChainFilter`, which is defined above in the HDK section.
  - **Payload**: The payload is defined as:

```
{
    agent: AgentPubKey,
    filter: ChainFilter,
}
```

  - **Response**: The response is defined as:

```
enum MustGetAgentActivityResponse {
    // The activity was found.
    Activity {
        // The actions performed by the agent.
        activity: Vec<RegisterAgentActivity>,
        // Any warrants issued to the agent for this activity.
        warrants: Vec<WarrantOp>,
    },
    // The requested chain range was incomplete.
    IncompleteChain,
    // The requested chain top was not found in the chain.
    ChainTopNotFound(ActionHash),
    // The filter produces an empty range.
    EmptyRange,
}
```

- `CountersigningSessionNegotiation`: Negotiate a step in the countersigning process.
  - **Payload**: The payload is defined as:

```
CountersigningSessionNegotiation {
    message: CountersigningSessionNegotiationMessage,
}

enum CountersigningSessionNegotiationMessage {
    // Sent by a `StoreEntry` authority or enzyme after they have
    // collected signed actions from all counterparties; the author (the
    // receiver of the message) may now safely proceed to commit their
    // own countersigning entry creation action.
    AuthorityResponse(Vec<SignedAction>),
    // Sent by a counterparty to the designated enzyme when they have
    // determined that the countersigned entry creation action is valid
    // from the perspective of all counterparties and they intend to
    // commit their own entry creation action once they have received
    // all signatures from the enzyme. The `DhtOp` payload is a
    // `StoreEntry`.
    EnzymePush(DhtOp),
}
```

- `PublishCountersign`: Publish a countersigned DHT operation to a DHT authority. This happens in two steps:
  1. When a counterparty has received preflight acceptances from all participating counterparties, and the countersigning session is not being managed by an enzyme, they proceed to register their intent to commit the countersigned entry creation action by publishing the `StoreEntry` operation with the `is_action_author` flag set to `true`. In this scenario, the authority is acting as a reliable witness to all counterparties' signatures and MUST send a `CountersigningSessionNegotiation(CountersigningSessionNegotiationMessage::AuthorityResponse)` message to all counterparties if, from their perspective, all operations have been received within the session time window.
  2. When a counterparty has received all signed actions from all other participating counterparties, they publish a `RegisterAgentActivity` operation for each of their counterparties' signed actions with the `is_action_author` flag set to `false`. Note that the DHT operation is signed by the counterparty sending the message, while the enclosed action is signed by the counterparty that authored the action. Also note that this MUST also be accompanied by regular `Publish` messages for the DHT operations produced from this counterparty's entry creation action.

  – **Notes**: On the receiver side, this is handled as an incoming publish message.

  – **Payload**: The payload is defined as:

```
{
    is_action_author: bool,
    op: DhtOp,
}
```

Low-Level Networking (Kitsune P2P)

Kitsune is a P2P library for implementing distributed application messaging needs that delivers dynamic peer address discovery and message routing. It also delivers the necessary affordances for distributed applications to implement sharded DHTs as a content-addressable store, as it groups its messages into `KitsuneSpace`s (which correspond to Holochain's DNA addresses) and `KitsuneAgent`s which are, as in Holochain, the public keys of the agents participating in the space. Kitsune handles the mapping of the `KitsuneAgent` address space to network transport addresses.

Kitsune assumes an "implementor" that defines its own higher-level custom message types, and manages persistent storage, and handles key management and message signing. Kitsune sends events to the implementor to retrieve data, and receives messages from the implementor to be delivered to other Kitsune nodes.

Thus, Holochain implements both its node-to-node messaging and its graph DHT on top of the capabilities provided by Kitsune.

Message Classes and Types Kitsune has two message classes:

- **Notify**: Optimistically send a message without listening for a response.
- **Request**: Send a message with a nonce, and expect a response with a matching nonce.

Messages of both of these classes are sent asynchronously; Request is simply a pattern of pairing two messages by means of a nonce.

These are the message types that MUST be implemented. They are all defined as variants of a `Wire` enum, which are serialized and sent via the network transport layer.

- `Failure`: Notify a peer of failure, as a response to a received message that couldn't be handled.
  - **Payload**: THe payload is defined as:

    ```
    {
        reason: String,
    }
    ```

- `Call`: Make a remote procedure call (RPC) to a remote peer.
  - **Payload**: The payload is defined as:

    ```
    Call {
        space: KitsuneSpace,
        to_agent: KitsuneAgent,
        data: Vec<u8>,
    }

    struct KitsuneSpace(Vec<u8>);

    struct KitsuneAgent(Vec<u8>);
    ```

    The `space` and `to_agent` arguments map at the Holochain layer to DNA hash and agent public key. The `data` argument holds the input that will be passed to the remote function.

- `CallResp`: Respond to a `Call` message with the output of the called function.
  - **Payload**: The payload is defined as a `Vec<u8>`.

- `Broadcast`: Broadcast a message using Notify.
  - **Payload**: The payload is defined as:

    ```
    {
        space: KitsuneSpace,
        to_agent: KitsuneAgent,
        data: BroadcastData,
    }

    enum BroadcastData {
        // Broadcast arbitrary data.
        User(Vec<u8>),
        // Broadcast one's own agent info.
        AgentInfo(AgentInfoSigned),
        // Announce that one or more DHT operations have been published for
        // which the receiver is believed to be an authority; it is expected
        // that they will follow up by sending `FetchOp` messages to request
        // the operations. Because the remote node may claim authority for a
        // range of basis hashes, multiple operations MUST be permitted to
        // be announced in one message.
        Publish {
            source: KitsuneAgent,
            op_hash_list: Vec<RoughSized<KitsuneOpHash>>,
            context: FetchContext,
        },
    }

    struct AgentInfoSigned {
        space: KitsuneSpace,
    ```

```
    agent: KitsuneAgent,
    storage_arq: Arq,
    url_list: Vec<url2::Url2>,
    signed_at_ms: u64,
    expires_at_ms: u64,
    signature: KitsuneSignature,
    encoded_bytes: [u8],
}

// Description of a network location arc over which an agent claims
// authority.
struct Arq {
    start: DhtLocation,
    // The size of chunks for this arc, as 2^power * 4096.
    power: u8,
    // The number of chunks in this arc. Hence, the arc size in terms of
    // network location space is power * count.
    count: u32,
}

// Network locations wrap at the bounds of u32.
struct DhtLocation(Wrapping<u32>);

// Represents a public key signature.
struct KitsuneSignature(Vec<u8>);

// Convey the rough size of the data behind a hash.
struct RoughSized<T> {
    // The hash of the data to be rough-sized.
    data: T,
    // The approximate size of the data.
    size: Option<RoughInt>,
}

// Positive numbers are an exact size; negative numbers represent a size
// of roughly -x * 4096.
struct RoughInt(i16);

// Represents a DHT operation hash.
struct KitsuneOpHash(Vec<u8>);

// Arbitrary context identifier.
struct FetchContext(u32);
```

- `DelegateBroadcast`: Broadcast a message to peers covering a basis hash, requesting receivers broadcast to peers in the same neighborhood.
  - **Payload**: The payload is defined as:

```
{
    space: KitsuneSpace,
    // The DHT basis hash to target.
    basis: KitsuneBasis,
    to_agent: KitsuneAgent,
    mod_idx: u32,
    mod_cnt: u32,
    data: BroadcastData,
}

struct KitsuneBasis(Vec<u8>);
```

The `mod_cnt` and `mod_idx` fields fields define the scope of the broadcast. Receivers MUST modulo the network locations of candidate authorities in their own peer tables by `mod_cnt`, only re-broadcasting to a peer if the modulo matches `mod_idx`. This avoids two nodes sending the same broadcast message to the same peer.

The `data` argument is the data to be passed on to the neighborhood, and is defined above under `Broadcast`.

- `Gossip`: Negotiate gossiping of DHT operations, with an opaque data block to be interpreted by a gossip implementation.
  - **Notes**: Kitsune handles gossip per Space (which maps to a DNA at the higher Holochain layer) rather than a cell. This message uses the Notify strategy.
  - **Payload**: The payload is defined as:

```
{
    space: KitsuneSpace,
    data: Vec<u8>,
    module: GossipModuleType,
}

// Currently implemented gossip strategies.
enum GossipModuleType {
    // Recent gossip deals with DHT data with a recent timestamp.
    ShardedRecent,
    // Historical gossip deals with data whose timestamp is older than
    // the recent gossip threshold.
    ShardedHistorical,
}
```

    The structure of the messages that appear in the `data` argument are documented in the following section on Gossip.

- `PeerGet`: Ask a remote node if they know about a specific agent.
  - **Payload**: The payload is defined as:

```
{
    space: KitsuneSpace,
    agent: KitsuneAgent,
}
```

- `PeerGetResp`: Respond to a `PeerGet` with information about the requested agent.
  - **Payload**: The payload is defined as:

```
{
    agent_info_signed: AgentInfoSigned,
}
```

- `PeerQuery`:Query a remote node for peers holding or nearest to holding a `u32` network location.
  - **Payload**: The payload is defined as:

```
{
    space: KitsuneSpace,
    basis_loc: DhtLocation,
}
```

- `PeerQueryResp`: Respond to a `PeerQuery`.
  - **Payload**: The payload is defined as:

```
{
    peer_list: Vec<AgentInfoSigned>,
}
```

- `PeerUnsolicited`: Send peer information without being asked to. Notably, a node may want to send their own peer info to prevent being inadvertently blocked.

  - **Payload**: The payload is defined as:

```
{
    peer_list: Vec<AgentInfoSigned>,
}
```

- `FetchOp`: Request DHT operation data which a node claims to hold.
  - **Notes**: This and its response `PushOpData` transfer the actual data which is validated and integrated at basis hashes for which a node is an authority. As an optimization, a node can request data for multiple Spaces which they believe the remote node has in common with them. The `FetchKey` type is defined as:
  - **Payload**: The payload is defined as:

```
{
    fetch_list: Vec<(KitsuneSpace, Vec<FetchKey>)>,
}

enum FetchKey {
    Op(KitsuneOpHash),
}
```

- `PushOpData`: Send requested DHT operation data in response to `FetchOp`.
  - **Payload**: The payload is defined as:

```
{
    op_data_list: Vec<(KitsuneSpace, Vec<PushOpItem>)>,
}

struct PushOpItem {
    op_data: Vec<u8>,
    region: Option<(RegionCoords, bool)>,
}

struct RegionCoords {
    space: Segment,
    time: Segment,
}

struct Segment {
    power: u8,
    offset: u32,
}
```

- `MetricExchange`: Exchange availability information about one or more peers.
  - **Notes**: Implementations SHOULD use this data to rebalance the arc of DHT basis hashes for which they claim authority, in order to ensure adequate availability for all basis hashes.
  - **Payload**: The payload is defined as:

```
{
    space: KitsuneSpace,
    msgs: Vec<MetricExchangeMsg>,
}

enum MetricExchangeMsg {
    V1UniBlast {
        extrap_cov_f32_le: Vec<u8>,
    },
    UnknownMessage,
}
```

Gossip Kitsune MUST provide a way for the DHT data to be gossiped among peers in a space. We assume that there will be many gossip implementations that are added over time.

Any gossip algorithm to be used in the Holochain context MUST be able to handle the following constraints:

1. A new node coming on to the network, or one returning to the network after a significant changes have occurred on the DHT, SHOULD be able to quickly but incrementally synchronize to a state of holding the correct data that it is deemed to be an authority for, while balancing bandwidth limitations of the network it is part of. This requires that the system be resilient to asymmetric upload and download speeds that will vary across peers, and indicates that nodes that believe they are out of sync with their peers release their authority, and incrementally and conservatively increase it.
2. Gossiping SHOULD minimize total consumed resources; e.g., by re-transmitting as little data as possible, dynamically adjusting gossip round frequency to levels of activity in the network, or backing off gossip when a peer exhibits backpressure.
3. Gossip SHOULD prioritize the synchronization of data that is more likely to be in demand; for many common application scenarios, this means that more recently published data should be synchronized sooner and more frequently.
4. Gossip SHOULD dynamically adapt to changing realities of authority coverage for all basis hashes by communicating individual peer coverage regularly, allowing nodes with sufficient excess capacity to assume greater coverage to compensate for regions with poor coverage, either because peers go offline or their excess capacity is limited.

We have developed a hybrid gossip implementation that separates DHT operations into "recent" and "historical", with recent gossip using a Bloom filter and historical gossip using a novel "quantized gossip" algorithm that efficiently shards and redistributes data as nodes come and go on the network. While the full description of that implementation is beyond the scope of this document, we will document the messages that nodes pass:

- `Initiate`: Propose a gossip round, specifying one or more arcs of the location space for which the initiator is an authority.
  - **Notes**: A gossip round MAY cover more than one agent within a given Space on the initiator's device.
  - **Payload**: The payload is defined as:

```
{
    intervals: Vec<Arq>,
    // Disambiguates gossip rounds initiated in parallel.
    id: u32,
    agent_list: Vec<AgentInfoSigned>,
}
```

- `Accept`: Respond to an `Initiate`, specifying the arcs for which the agents in the acceptor's conductor are authorities.
  - **Notes**: The gossip round, as it goes forward, will concern network locations that are the *set intersection* of all the network locations covered by all the arcs of both the initiator and the acceptor.
  - **Payload**: The payload is defined as:

```
{
    intervals: Vec<Arq>,
    agent_list: Vec<AgentInfoSigned>,
}
```

- `Agents`: Send a Bloom filter of the public keys of all the agents for which a peer is storing `AgentInfo` data.
  - **Notes**: The recipient is expected to compare this value against the Bloom filter value for their own held `AgentInfo` data, and respond with a `MissingAgents` message. As this uses a Bloom filter, peers may require a few rounds of exchanges before they converge on identical values and are finished synchronizing `AgentInfo` data. Implementations SHOULD retry on a loop until this condition is satisfied.
  - **Payload**: The payload is defined as:

```
{
    filter: Option<(usize, Vec<u8>)>,
}
```

- `MissingAgents`: Respond to `Agents`, supplying the `AgentInfo`s for all the agents that did not appear to be included in the Bloom filter.
  - **Payload**: The payload is defined as:

```
{
    agents: Vec<AgentInfoSigned>,
}
```

- `OpBloom`: Send a Bloom filter of the hashes of all the *recent* DHT operations which a peer is holding.
  - **Notes**: As with **Agents**, the recipient compares this value with their own held recent DHT operations and responds with a **MissingOpHashes** message, and the exchange SHOULD be repeated until peers are synchronized.
  - **Payload**: The payload is defined as:

```
{
    // The Bloom filter value.
    missing_hashes: EncodedTimedBloomFilter,
    // Is this the last Bloom message to be sent?
    finished: 1,
}

enum EncodedTimedBloomFilter {
    // I have no overlap with your agents; please don't send any ops.
    NoOverlap,
    // I have overlap and I have no hashes; please send all your ops.
    MissingAllHashes {
        // The time window that we are missing hashes for.
        time_window: Range<Timestamp>,
    },
    // I have overlap and I have some hashes; please send any missing
    // ops.
    HaveHashes {
        // The encoded Bloom filter.
        filter: Vec<u8>,
        // The time window these hashes are for.
        time_window: Range<Timestamp>,
    },
}
```

- `OpRegions`: Send a map of quantized region coordinates to XOR fingerprints of the set of DHT operations the sender is holding for that region.
  - **Notes**: Its purpose is to quickly communicate information about the infrequently changing set of *historical* DHT operations which the sender holds for comparison and synchronization. The recipient is expected to send the DHT operation hashes for all mismatched regions via `MissingOpHashes`.
  - **Payload**: The payload is defined as:

```
{
    region_set: RegionSetLtcs
}

struct RegionSetLtcs {
    // The generator for the coordinates.
    coords: RegionCoordSetLtcs,
    // The outermost vec corresponds to arqs in the `ArqSet`; the
    // middle vecs correspond to the spatial segments per arq; the
    // innermost vecs are the time segments per arq.
    data: Vec<Vec<Vec<RegionData>>>,
}
```

```
struct RegionCoordSetLtcs {
    times: TelescopingTimes,
    arq_set: ArqSet,
}

struct TelescopingTimes {
    time: TimeQuantum,
    // MUST be equal to or more recent than the DNA's `origin_time`
    // property.
    limit: Option<u32>,
}

struct TimeQuantum(u32);

struct RegionData {
    // The XOR of the hashes of all operations found in this region.
    hash: Hash32,
    // The total size of operation data contained in this region.
    size: u32,
    // The number of operations in this region.
    count: u32,
}
```

- `MissingOpHashes`: Respond to an `OpBloom` or `OpRegions` message with a list of DHT operation hashes which don't match the sender's Bloom filter.
  - **Notes**: If the list is large, it can be chunked into multiple messages, and the `finished` property in each message indicates whether more chunks will be sent. (Implementations SHOULD automatically send the next chunk without being asked to.) After this, the recipient of this message is expected to retrieve DHT operation data from the sender, not via gossip but via the `FetchOp` notify message.
  - **Payload**: The payload is defined as:

```
{
    ops: Vec<HashSized>,
    finished: bool;
}
```

- `Error { message: String, }`, `Busy`, `NoAgents`, `AlreadyInProgress`: Sent by the receiver of a gossip message if they're unable to satisfy the sender's request for the specified reason.

The Conductor

A Holochain Conductor manages running Holochain applications, which consist of logically related DNAs operating under a single agency. Thus a conductor MUST be able to interpret an application bundle format and instantiate Cells from that format. The bundle format SHOULD store its manifests in a readable format such as YAML, and SHOULD be capable of storing arbitrary resources as streams of bytes. Additionally a Conductor SHOULD cache DNA definitions and WASMs provided (along with WASM instructions compiled to machine instructions, if supported by the architecture) so as to decrease installation time of other instances of the same DNA and not store multiple copies of the same DNA.

Bundle Formats

Holochain implementations must be able to load Holochain applications that have been serialized, either to disk or for transmission over a network. Holochain uses a bundling format that allows for specification of properties along with other resources in a manifest that can include recursively bundled elements of the same general bundling format but adapted for different component types. The bundling format can also store the resources themselves within the same file; any of the sub-bundles can be specified by "location", which may be specified to be in the same bundle, in a separate file, or at a network address. Thus we have Zomes, DNAs, Apps, UIs, and WebApps that can all be wrapped

up in a single bundle, or can reference components stored elsewhere.[32] The manifests for each of the type of bundles that MUST be implemented are specified as follows:

DNA Bundle Manifest A DNA bundle manifest specifies the components that are critical to the operation of the DNA and affect its hash (the `IntegrityManifest` property) as well as the components that are supplied to facilitate the operation of a cell (the `CoordinatorManifest` property).

```
struct DnaManifestV1 {
    // A user-facing label for the DNA.
    name: String,
    integrity: IntegrityManifest,
    coordinator: CoordinatorManifest,
    // A list of ancestors of this DNA, used for satisfying dependencies on
    // prior versions of this DNA. The application's Coordinator interface is
    // expected to be compatible across the list of ancestors.
    lineage: Vec<DnaHashB64>,
}

struct IntegrityManifest {
    // A network seed for uniquifying this DNA.
    network_seed: Option<Vec<u8>>,

    // Any arbitrary application properties can be included in this object.
    // They may be accessed by DNA code to affect runtime behavior.
    properties: Option<YamlProperties>,

    // The time used to denote the origin of the network, used to calculate time
    // windows during gossip. All Action timestamps must come after this time.
    origin_time: HumanTimestamp,

    // An array of integrity zome manifests associated with the DNA. The order
    // is significant: it determines initialization order and affects the DNA
    // hash.
    zomes: Vec<ZomeManifest>,
}

struct CoordinatorManifest {
    // Coordinator zomes to install with this DNA.
    zomes: Vec<ZomeManifest>,
}

struct ZomeManifest {
    // A user-facing label for the zome.
    name: ZomeName,

    // The hash of the WebAssembly bytecode which defines this zome.
    hash: Option<WasmHashB64>,

    // The location of the wasm for this zome.
    location: Location,

    // The integrity zomes this zome depends on. The order of these MUST match
    // the order the types are used in the zome.
    dependencies: Option<Vec<ZomeName>>,
}
```

[32] The "meta bundle" format can be seen here: https://github.com/holochain/holochain/tree/develop/crates/mr_bundle.

```
enum Location {
    Bundled(PathBuf),

    // Get the file from the local filesystem (not bundled).
    Path(PathBuf),

    // Get the file from a URL.
    Url(String),
}
```

App Bundle Manifest An `AppBundle` combines together a set of DNAs paired with "Role" identifiers and instructions for how/when the Conductor should instantiate DNAs to make cells in the bundle. The "role" of DNA is useful for application developers to be able to specify a DNA by a semantically accessible name rather than just its hash. This also allows for "late-binding" as DNAs may be used in different ways in applications, and thus we can think of the DNA's name by the role it plays in a given application.

There is a number of ways that application developers MUST be able to specify conditions under which DNAs are instantiated into Cells in the Conductor:

- The basic use case is simply that a DNA is expected to be instantiated as a Cell. There MUST be an option to defer instantiation of the installed DNA until a later time, thus implementing a "lazy loading" strategy.
- There is a number of use cases where a Holochain application will also expect a Cell of a given DNA to already have instantiated and relies on this behavior, and fail otherwise. Thus, there MUST be a provisioning option to specify this use case. There also SHOULD be a way of signalling to the conductor that the dependency SHOULD NOT be disabled or uninstalled until the dependent app is uninstalled.
- Holochain Conductors MUST also implement a "cloning" mechanism to allow applications to dynamically create new Cells from an existing DNA via the App interface (see Conductor API below). Cloned cells are intended to be used for such use cases as adding private workspaces to apps where only a specific set of agents are allowed to join the DHT of that DNA, such as private channels; or for creating DHTs that have temporary life-spans in app, like logs that get rotated. DNAs that are expected to be cloned MUST be specified as such in the DNA Bundle so that the Conductor can have cached and readied the WASM code for that DNA.
- Finally, Conductors MUST provide a way for an App to be installed without supplying membrane proofs and instantiating Cells, in cases where membrane proof values are dependent on the agent's public key which is generated at application installation time. This MUST be accompanied by a method of supplying those membrane proofs when they become available. (Note that this method of deferred instantiation is distinct from the deferred option for the preceding strategies in two ways: first, its purpose is to enable an instantiation process which requires information that isn't available until after installation rather than to enable lazy loading, and second, the Cells are instantiated but not active.)

```
struct AppManifestV1 {
    // User-facing name of the App. This may be used as the `installed_app_id`
    // in the Admin API.
    name: String,

    // User-facing description of the app.
    description: Option<String>,

    // The roles that need to be filled (by DNAs) for this app.
    roles: Vec<AppRoleManifest>,

    // If true, the app should be installed without needing to specify membrane
    // proofs. The app's cells will be in an incompletely instantiated state
    // until membrane proofs are supplied for each.
    membrane_proofs_deferred: bool,
}

struct AppRoleManifest {
    // The ID which will be used to refer to:
    // * this role,
    // * the DNA which fills it,
    // * and the cell(s) created from that DNA
```

```
    name: RoleName,

    // Determines if, how, and when a Cell will be provisioned.
    provisioning: Option<CellProvisioning>,

    // The location of the DNA bundle resource, and options to modify it before
    // instantiating in a Cell.
    dna: AppRoleDnaManifest,
}

type RoleName = String;

enum CellProvisioning {
    // Always create a new Cell when installing this App.
    Create { deferred: bool },

    // Require that a Cell be already installed which matches the DNA
    // `installed_hash` spec, and which has an Agent that's associated with
    // this App's agent via DPKI. If no such Cell exists, *app installation MUST
    // fail*. The `protected` flag indicates that the Conductor SHOULD NOT allow
    // the dependency to be disabled or uninstalled until all cells using this
    // DNA are uninstalled.
    UseExisting { protected: bool },

    // Install or locate the DNA, but do not instantiate a Cell for it. Clones
    // may be instantiated later. This requires that `clone_limit` > 0.
    CloneOnly,
}

struct AppRoleDnaManifest {
    // Where to find this DNA.
    location: Option<Location>,

    // Optional default modifier values, which override those found in the DNA
    // manifest and may be overridden during installation.
    modifiers: DnaModifiersOpt<YamlProperties>,

    // The expected hash of the DNA's integrity manifest. If specified,
    // installation MUST fail if the hash does not match this. Also allows this
    // DNA to be targeted as a dependency in `AppRoleManifest`s that specify
    // `UseExisting` or `CreateIfNotExists` provisioning strategies.
    installed_hash: Option<DnaHashB64>,

    // Allow up to this many "clones" to be created at runtime.
    clone_limit: u32,
}
```

WebApp Bundle A `WebAppBundle` combines together a specific user interface together with an `AppBundle` as follows:

```
struct WebAppManifestV1 {
    // Name of the App. This may be used as the `installed_app_id`.
    name: String,

    // Web UI used for this app, packaged in a .zip file.
    ui: Location,

    // The AppBundle location.
    happ_manifest: Location,
}
```

API

A Holochain Conductor MUST provide access for user action through an Admin API to manage Apps and DNAs (install/uninstall, enable/disable, etc) and through an App API to make zome calls to specific DNAs in specific Apps, create cloned DNAs, supply deferred membrane proofs, and introspect the App. In our implementation, these API is defined as a library so that these calls can be made in-process, but they are also implemented over a WebSocket interface so they can be called by external processes.

In the WebSocket implementation of this API, requests and responses are wrapped in an "envelope" format that contains a nonce to match requests with response, then serialized and sent as WebSocket messages. The request message types are defined as variants of an `AdminRequest` or `AppRequest` enum, as are their corresponding responses (`AdminResponse` and `AppResponse` respectively). Hence, in the API definitions below, the enum name of the function name or return value type is implied.

Both response enums MUST define an `Error(e)` variant to communicate error conditions, where `e` is a variant of the enum:

```
enum ExternalApiWireError {
    // Any internal error.
    InternalError(String),
    // The input to the API failed to deserialize.
    Deserialization(String),
    // The DNA path provided was invalid.
    DnaReadError(String),
    // There was an error in the ribosome.
    RibosomeError(String),
    // Error activating app.
    ActivateApp(String),
    // The zome call is unauthorized.
    ZomeCallUnauthorized(String),
    // A countersigning session has failed.
    CountersigningSessionError(String),
}
```

Admin API Below is a list of the Admin API functions that MUST be implemented along with any details of function arguments and return values, as well as any contextual notes on functional constraints or other necessary implementation details.

For error conditions, the `AppResponse::Error(e)` variant MUST be used, where `e` is a variant of the `ExternalApiWireError` enum.

- `AddAdminInterfaces(Vec<AdminInterfaceConfig>) -> AdminInterfacesAdded`: Set up and register one or more new admin interfaces as specified by a list of configurations.
  - **Arguments**: The `AdminInterfaceConfig` SHOULD be a generalized data structure to allow creation of an interface of whatever types are contextually appropriate for the system on which the conductor runs:

    ```
    struct AdminInterfaceConfig {
        driver: InterfaceDriver,
    }

    enum InterfaceDriver {
        Websocket {
            port: u16,
            // The allowed values of the `Origin` HTTP header.
            allowed_origins: AllowedOrigins,
        }
    }

    enum AllowedOrigins {
        Any,
        Origins(HashSet<String>),
    }
    ```

- `RegisterDna(RegisterDnaPayload) -> DnaRegistered(DnaHash)` : Install a DNA for later use in an App.
  - **Notes**: This call MUST store the given DNA into the Holochain DNA database. This call exists separately from `InstallApp` to support the use case of adding a DNA into a conductor's DNA database once, such that the transpilation of WASM to machine code happens only once and gets cached in the conductor's WASM store.
  - **Arguments**: A struct of the following type:

```
struct RegisterDnaPayload {
    // Override the DNA modifiers specified in the app and/or DNA bundle
    // manifest(s).
    modifiers: DnaModifiersOpt<YamlProperties>,
    source: DnaSource,
}

enum DnaSource {
    Path(PathBuf),
    Bundle(DnaBundle),
    // Register the DNA from an existing DNA registered via a prior
    // `RegisterDna` call or an `InstallApp` call.
    Hash(DnaHash),
}
```

  - **Return value**: If the DNA cannot be located at the specified path, `AdminResponse::Error(ExternalApiWireError::DnaReadError(s))` MUST be returned, where `s` is an error message to be used for troubleshooting.
- `GetDnaDefinition(DnaHash) -> DnaDefinitionReturned(DnaHash)`: Get the definition of a DNA.
  - **Return Value**: This function MUST return all of the data that specifies a DNA as installed as follows:

```
struct DnaDef {
    name: String,
    modifiers: DnaModifiers,
    integrity_zomes: Vec<ZomeName>,
    coordinator_zomes: Vec<ZomeName>,
    lineage: HashSet<DnaHash>,
}
```

- `UpdateCoordinators(UpdateCoordinatorsPayload) -> CoordinatorsUpdated`: Update coordinator zomes for an already installed DNA.
  - **Notes**: This call MUST replace any installed coordinator zomes with the same zome name. If the zome name doesn't exist then the coordinator zome MUST be appended to the current list of coordinator zomes.
  - **Arguments**: A struct defined as:

```
struct UpdateCoordinatorsPayload {
    dna_hash: DnaHash,
    source: CoordinatorSource,
}

enum CoordinatorSource {
    // Load coordinators from a bundle file.
    Path(PathBuf),
    Bundle(Bundle<Vec<ZomeManifest>>),
}
```

- `InstallApp(InstallAppPayload) -> AppInstalled(AppInfo)`: Install an app using an `AppBundle`.
  - **Notes**: An app is intended for use by one and only one Agent, and for that reason it takes an `AgentPubKey` and instantiates all the DNAs bound to that `AgentPubKey` as new Cells. The new app should not be enabled automatically after installation, and instead must explicitly be enabled by calling `EnableApp`.
  - **Arguments**: `InstallAppPayload` is defined as:

```
struct InstallAppPayload {
    source: AppBundleSource,

    // The agent to use when creating Cells for this App.
    agent_key: AgentPubKey,

    // The unique identifier for an installed app in this conductor.
    // If not specified, it will be derived from the app name in the
    // bundle manifest.
    installed_app_id: Option<String>,

    // Optional proof-of-membrane-membership data for any cells that
    // require it, keyed by the `RoleName` specified in the app bundle
    // manifest.
    membrane_proofs: HashMap<RoleName, MembraneProof>,

    // Optional: overwrites all network seeds for all DNAs of Cells
    // created by this app. This does not affect cells provisioned by
    // the `UseExisting` strategy.
    network_seed: Option<Vec<u8>>,

    // If app installation fails due to genesis failure, normally the
    // app will be immediately uninstalled. When this flag is set, the
    // app is left installed with empty cells intact. This can be useful
    // for using `GraftRecordsOntoSourceChain` or diagnostics.
    ignore_genesis_failure: bool,
}
```

– **Return Value**: The returned value MUST contain the `AppInfo` data structure (which is also retrievable after installation via the `GetAppInfo` API), and is defined as:

```
struct AppInfo {
    installed_app_id: String,
    cell_info: HashMap<RoleName, Vec<CellInfo>>,
    status: AppInfoStatus,
}

enum CellInfo {
    // Cell provisioned at app installation as defined in the bundle.
    Provisioned(ProvisionedCell),
    // Cell created at runtime by cloning a DNA.
    Cloned(ClonedCell),
    // Potential cell with deferred installation as defined in the
    // bundle.
    Stem(StemCell),
}

struct ProvisionedCell {
    cell_id: CellId,
    dna_modifiers: DnaModifiers,
    name: String,
}

struct StemCell {
    // The hash of the DNA that this cell will be instantiated from.
    original_dna_hash: DnaHash,
    // The DNA modifiers that will be used when instantiating the cell.
    dna_modifiers: DnaModifiers,
    // An optional name to override the cell's bundle name when
```

```
    // instantiating.
    name: Option<String>,
}

enum AppInfoStatus {
    // The app is paused due to a recoverable error. There is no way to
    // manually pause an app.
    Paused { reason: PausedAppReason },
    // The app is disabled, and may be restartable depending on the
    // reason.
    Disabled { reason: DisabledAppReason },
    Running,
    AwaitingMemproofs,
}

enum PausedAppReason {
    Error(String);
}

enum DisabledAppReason {
    // The app is freshly installed, and has not been started yet.
    NeverStarted,
    // The app is fully installed and deferred memproofs have been
    // provided by the UI, but the app has not been started yet.
    NotStartedAfterProvidingMemproofs,
    // The app has been disabled manually by the user via an admin
    // interface.
    User,
    // The app has been disabled due to an unrecoverable error.
    Error(String),
}
```

- `UninstallApp { installed_app_id: InstalledAppId } -> AppUninstalled` : Uninstall the app specified by the argument `installed_app_id` from the conductor.
  - **Notes**: The app MUST be removed from the list of installed apps, and any cells which were referenced only by this app MUST be disabled and removed, clearing up any persisted data. Cells which are still referenced by other installed apps MUST NOT be removed.
- `ListDnas -> DnasListed(Vec<DnaHash>)` : List the hashes of all installed DNAs.
- `GenerateAgentPubKey -> AgentPubKeyGenerated(AgentPubKey)` : Generate a new Ed25519 key pair.
  - **Notes**: This call MUST cause a new key pair to be added to the key store and return the public part of that key to the caller. This public key is intended to be used later when installing an App, as a Cell represents the agency of an agent within the space created by a DNA, and that agency comes from the power to sign data with a private key.
- `ListCellIds -> CellIdsListed<Vec<CellId>>`: List all the cell IDs in the conductor.
- `ListApps { status_filter: Option<AppStatusFilter> } -> AppsListed(Vec<AppInfo>)`: List the apps and their information that are installed in the conductor.
  - **Notes**: If `status_filter` is `Some(_)`, it MUST return only the apps with the specified status.
  - **Arguments**: The value of `status_filter` is defined as:

```
enum AppStatusFilter {
    // Filter on apps which are Enabled, which can include both Running
    // and Paused apps.
    Enabled,
    // Filter only on apps which are Disabled.
    Disabled,
    // Filter on apps which are currently Running (meaning they are also
```

```
        // Enabled).
        Running,
        // Filter on apps which are Stopped, i.e. not Running. This includes
        // apps in the Disabled status, as well as the Paused status.
        Stopped,
        // Filter only on Paused apps.
        Paused,
    }
```

- `EnableApp { installed_app_id: InstalledAppId } -> AppEnabled { app: AppInfo, errors: Vec<(CellId, String)> }`: Change the specified app from a disabled to an enabled state in the conductor.
  - **Notes**: Once an app is enabled, zome functions of all the Cells associated with the App that have a `Create` or `CreateIfNotExists` provisioning strategy MUST immediately be callable. Previously enabled Applications MUST also be loaded and enabled automatically on any reboot of the conductor.
  - **Return value**: If the attempt to enable the app was successful, `AdminResponse::Error(ExternalApiWireError::ActivateApp(s))` MUST be returned, where `s` is an error message to be used for troubleshooting purposes.
- `DisableApp { installed_app_id: InstalledAppId } -> AppDisabled`: Changes the specified app from an enabled to a disabled state in the conductor.
  - **Notes**: When an app is disabled, calls to zome functions of all the Cells associated with the App MUST fail, and the app MUST not be loaded on a reboot of the conductor. Note if cells are associated with more than one app, they MUST not be disabled unless all of the other apps using the same cells have also been disabled.
- `AttachAppInterface { port: Option<u16>, allowed_origins: AllowedOrigins, installed_app_id: Option<InstalledAppID> } -> AppInterfaceAttached { port: u16 }`: Open up a new WebSocket interface for processing `AppRequest`s.
  - **Notes**: All active apps, or the app specified by `installed_app_id`, if active, MUST be callable via the attached app interface. If an app is specified, all other apps MUST NOT be callable via the attached app interface. If the `allowed_origins` argument is not `Any`, the Conductor MUST reject any connection attempts supplying an HTTP `Origin` header value not in the list. Optionally a `port` parameter MAY be passed to this request. If it is `None`, a free port SHOULD be chosen by the conductor. The response MUST contain the port chosen by the conductor if `None` was passed.
  - **Arguments**: The `allowed_origins` field is a value of the type:

    ```
    enum AllowedOrigins {
        Any,
        Origins(HashSet<String>),
    }
    ```
- `ListAppInterfaces -> AppInterfacesListed(Vec<AppInterfaceInfo>)`: List all the app interfaces currently attached with `AttachAppInterface`, which is a list of WebSocket ports that can process `AppRequest()`s.
  - **Return value**: The app interface info is defined as:

    ```
    struct AppInterfaceInfo {
        port: u16,
        allowed_origins: AllowedOrigins,
        installed_app_id: Option<InstalledAppId>,
    }
    ```
- **Debugging and introspection dumps**: The following functions are for dumping data about the state of the Conductor. Implementations MAY implement these functions; there is no standard for what they return, other than that they SHOULD be self-describing JSON blobs of useful information that can be parsed by diagnostic tools.
  - `DumpState { cell_id: CellId } -> StateDumped(String)`: Dump the state of the cell specified by the argument `cell_id`, including its chain.

- `DumpConductorState -> ConductorStateDumped(String)`: Dump the configured state of the Conductor, including the in-memory representation and the persisted state, as JSON. State to include MAY include status of Applications and Cells, networking configuration, and app interfaces.
- `DumpFullState { cell_id: CellId, dht_ops_cursor: Option<u64> } -> FullStateDumped(FullStateDump)`: Dump the full state of the specified Cell, including its chain, the list of known peers, and the contents of the DHT shard for which it has claimed authority.
  - **Notes**: The full state including the DHT shard can be quite large.
  - **Arguments**: The database cursor of the last-seen DHT operation row can be supplied in the `dht_ops_cursor` field to dump only unseen state. If specified, the call MUST NOT return DHT operation data from this row and earlier.
  - **Return value**: Unlike other dump functions, this one has some explicit structure defined by Rust types, taking the form:

```
struct FullStateDump {
    // Information from the Kitsune networking layer about the
    // agent, the DHT space, and their known peers.
    peer_dump: P2pAgentsDump,
    // The cell's source chain.
    source_chain_dump: SourceChainDump,
    // The dump of the DHT shard for which the agent is responsible.
    integration_dump: FullIntegrationStateDump,
}

struct P2pAgentsDump {
    // Information about this agent's cell.
    this_agent_info: Option<AgentInfoDump>,
    // Information about this DNA itself at the level of Kitsune
    // networking.
    this_dna: Option<(DnaHash, KitsuneSpace)>,
    // Information about this agent at the level of Kitsune
    // networking.
    this_agent: Option<(AgentPubKey, KitsuneAgent)>,
    // Information about the agent's known peers.
    peers: Vec<AgentInfoDump>,
}

// Agent info dump with the agent, space, signed timestamp, and
// expiry of last self-announced info, printed in a pretty way.
struct AgentInfoDump {
    kitsune_agent: KitsuneAgent,
    kitsune_space: KitsuneSpace,
    dump: String,
}

struct SourceChainDump {
    records: Vec<SourceChainDumpRecord>,
    published_ops_count: usize,
}

struct SourceChainDumpRecord {
    signature: Signature,
    action_address: ActionHash,
    action: Action,
    entry: Option<Entry>,
}

struct FullIntegrationStateDump {
```

```
        // Ops in validation limbo awaiting sys or app validation.
        validation_limbo: Vec<DhtOp>,
        // Ops waiting to be integrated.
        integration_limbo: Vec<DhtOp>,
        // Ops that are integrated. This includes rejected ops.
        integrated: Vec<DhtOp>,
        // Database row ID for the latest DhtOp that we have seen.
        // Useful for subsequent calls to `FullStateDump` to return only
        // what they haven't seen.
        dht_ops_cursor: u64,
    }
```

  - `DumpNetworkMetrics { dna_hash: Option<DnaHash> } -> NetworkMetricsDumped(String)`: Dump the network metrics tracked by Kitsune.
    * **Arguments**: If the `dna_hash` argument is supplied, the call MUST limit the metrics dumped to a single DNA hash space.
  - `DumpNetworkStats -> NetworkStatsDumped(String)`: Dump network statistics from the back-end networking library. This library operates on a lower level than Kitsune and Holochain P2P, translating the P2P messages into protocol communications in a form appropriate for the physical layer. Our implementation currently includes a WebRTC library.

- `AddAgentInfo { agent_infos: Vec<AgentInfoSigned> } -> AgentInfoAdded`: Add a list of agents to this conductor's peer store.
  - **Notes**: Implementations MAY implement this function. It is intended as a way of shortcutting peer discovery and is useful for testing. It is also intended for use cases in which it is important for agent existence to be transmitted out-of-band.

- `GetAgentInfo { dna_hash: Option<DnaHash> } -> AgentInfoReturned(Vec<AgentInfoSigned>)`: Request information about the agents in this Conductor's peer store; that is, the peers that this Conductor knows about.
  - **Notes**: Implementations MAY implement this function. It is useful for testing across networks. It is also intended for use cases in which it is important for peer info to be transmitted out-of-band.
  - **Arguments**: If supplied, the `dna_hash` argument MUST constrain the results to the peers of the specified DNA.

- `GraftRecords { cell_id: CellId, validate: bool, records: Vec<Record> } -> RecordsGrafted`: "Graft" `Record`s onto the source chain of the specified `CellId`.
  - **Notes**: Implementations MAY implement this function. This admin call is provided for the purposes of restoring chains from backup. All records must be authored and signed by the same agent; if they are not, the call MUST fail. Caution must be exercised to avoid creating source chain forks, which will occur if the chains in the Conductor store and the new records supplied in this call diverge and have had their `RegisterAgentActivity` operations already published.
  - **Arguments**:
    * If `validate` is `true`, then the records MUST be validated before insertion. If `validate` is `false`, then records MUST be inserted as-is.
    * Records provided are expected to form a valid chain segment (ascending sequence numbers and valid `prev_action` references). If the first record contains a `prev_action` which matches an existing record, then the new records MUST be "grafted" onto the existing chain at that point, and any other records following that point which do not match the new records MUST be discarded. See the note above about the risk of source chain forks when using this call.
    * If the DNA whose hash is referenced in the `cell_id` argument is not already installed on this conductor, the call MUST fail.

- `GrantZomeCallCapability(GrantZomeCallCapabilityPayload) -> ZomeCallCapabilityGranted`: Attempt to store a capability grant on the source chain of the specified cell, so that a client may make zome calls to that cell.
  - **Notes**: Callers SHOULD construct a grant that uses the strongest security compatible with the use case; if a client is able to construct and store an Ed25519 key pair and use it to sign zome call payloads, a grant

using `CapAccess::Assigned` with the client's public key SHOULD be favored.

  - **Arguments**: The payload is defined as:

```
struct GrantZomeCallCapabilityPayload {
    // Cell for which to authorize the capability.
    cell_id: CellId,
    // Specifies the capability, consisting of zomes and functions to
    // allow signing for as well as access level, secret and assignees.
    cap_grant: ZomeCallCapGrant,
}
```

- `DeleteCloneCell(DeleteCloneCellPayload) -> CloneCellDeleted`: Delete a disabled cloned cell.
  - **Notes**: The conductor MUST return an error if the specified cell cannot be disabled.
  - **Arguments**: The payload is defined as:

```
struct DeleteCloneCellPayload {
    app_id: InstalledAppId,
    clone_cell_id: CloneCellID,
}
```

- `GetStorageInfo -> StorageInfoReturned(StorageInfo)`: Request storage space consumed by the Conductor.
  - **Notes**: Implementations MAY implement this function to allow resource consumption to be displayed. If implemented, all runtime resources consumption MUST be reported.
  - **Return Value**: Storage consumption info, defined as:

```
struct StorageInfo {
    blobs: Vec<StorageBlob>,
}

enum StorageBlob {
    Dna(DnaStorageInfo),
}

// All sizes are in bytes. Fields ending with `_on_disk` contain the
// actual file size, inclusive of allocated but empty space in the file.
// All other fields contain the space taken up by actual data.
struct DnaStorageInfo {
    // The size of the source chain data.
    authored_data_size: usize,
    authored_data_size_on_disk: usize,
    // The size of the DHT shard data for which all local cells are
    // authorities.
    dht_data_size: usize,
    dht_data_size_on_disk: usize,
    // The size of retrieved DHT data for which local cells are not
    // authorities.
    cache_data_size: usize,
    cache_data_size_on_disk: usize,
    // The ID of the app to which the above data applies.
    used_by: Vec<InstalledAppId>,
}
```

- `IssueAppAuthenticationToken(IssueAppAuthenticationTokenPayload) -> AppAuthenticationTokenIssued(AppAut`
  Request an authentication token for use by a client that wishes to connect to the app WebSocket.
  - **Notes**: Implementations MUST expect a client to supply a valid token in the initial HTTP request that establishes the WebSocket connection, and MUST reject connection attempts that do not supply a valid token. An invalid token is defined as either one that was never issued or one that is no longer usable. The latter happens in four different cases:

    * The token had an expiry set and the expiry timeout hsa passed,
    * The token was single-use and has been used once,
    * The token was revoked,
    * The conductor has been restarted since the token was issued (implementations MAY implement this case).

  Implementations MUST bind the WebSocket connection to the app for which the token was issued, excluding the possibility of a client accessing the functionality, status, and data of an app other than the one the token is bound to. Implementations SHOULD NOT terminate an established WebSocket connection once the token has expired; the expiry is to be enforced at connection establishment time.

  – **Arguments**: The payload is defined as:

```
struct IssueAppAuthenticationTokenPayload {
    // The app to bind the token to.
    installed_app_id: InstalledAppID,
    // MAY be set to a reasonable default such as 30 seconds if not
    // specified; MUST NOT expire if set to 0.
    expiry_seconds: u64,
    // MAY default to true.
    single_use: bool,
}
```

  – **Return type**: The payload is defined as:

```
struct AppAuthenticationTokenIssued {
    token: Vec<u8>,
    expires_at: Option<Timestamp>,
}
```

    The generated token MUST be unguessable; that is, it MUST be sufficiently strong to thwart brute-force attempts and sufficiently random to thwart educated guesses.

- `RevokeAppAuthenticationToken(AppAuthenticationToken) -> AppAuthenticationTokenRevoked`: Revoke a previously issued app interface authentication token.

  – **Notes**: Implementations MUST reject all WebSocket connection attempts using this token after the call has completed.

- `GetCompatibleCells(DnaHash) -> CompatibleCellsReturned(BTreeSet<(InstalledAppId, BTreeSet<CellId>)>)`: Find installed cells which use a DNA that is forward-compatible with the given DNA hash, as defined in the contents of the `lineage` field in the DNA manifest.

  – **Notes**: Implementations SHOULD search DNAs installed by all applications, as well as DNAs installed ad-hoc via `RegisterDna`.

App API An App interface MUST expose the following API for all the apps to which it is bound. However, it MUST also enforce the use of valid `Origin` headers and authentication tokens for each WebSocket connection establishment attempt, and MUST bind the connection to the app for which the token was issued.

As with the Admin API, the following are expressed as variants of an `AppRequest` enum and a corresponding `AppResponse` enum.

For error conditions, the `AppResponse::Error(e)` variant MUST be used, where `e` is a variant of the following enum:

```
enum ExternalApiWireError {
    // Any internal error.
    InternalError(String),
    // The input to the API failed to deserialize.
    Deserialization(String),
    // The DNA path provided was invalid.
    DnaReadError(String),
    // There was an error in the ribosome.
    RibosomeError(String),
    // Error activating app.
    ActivateApp(String),
```

```
    // The zome call is unauthorized.
    ZomeCallUnauthorized(String),
    // A countersigning session has failed.
    CountersigningSessionError(String),
}
```

- `GetAppInfo -> AppInfoReturned(Option<AppInfo>)`: Get info about the app, including info about each cell instantiated by this app. See above for the definition of `AppInfo`.
- `CallZome(ZomeCall) -> ZomeCalled(ExternIO)`: Call a zome function.
  - **Notes**: Implementations MUST enforce a valid capability for the function being called. This means that if the function is covered by a transferrable or assigned grant, the secret MUST be provided and valid; and if the function is covered by an assigned grant, the provenance MUST be valid. Regardless of the grant's access type, implementations MUST enforce that the provided signature matches the provided provenance. Implementations also MUST prevent replay attacks by rejecting a call that supplies a nonce that has been seen before or an expiry timestamp that has passed. Finally, the provenance (source) of the call MUST match the signature.
  - **Arguments**: The payload is defined as:

    ```
    struct ZomeCall {
        // The ID of the cell containing the zome to be called.
        cell_id: CellId,
        // The zome containing the function to be called.
        zome_name: ZomeName,
        // The name of the zome function to call.
        fn_name: FunctionName,
        // The serialized data to pass as an argument to the zome function
        // call.
        payload: ExternIO,
        // The secret necessary for exercising a claim against the granted
        // capability, if the capability is `CapAccess::Transferable` or
        // `CapAccess::Assigned`.
        cap_secret: Option<CapSecret>,
        provenance: AgentPubKey,
        // The signature on a serialized `ZomeCallUnsigned` struct with the same field values as this
        signature: Signature,
        nonce: Nonce256Bits,
        expires_at: Timestamp,
    }
    ```

    The `payload` property is a MsgPack-encoded data structure provided to the zome function. This structure MUST be matched against the parameter defined by the zome function, and the zome function MUST return a serialization error if it fails.
  - **Return Value**: The payload MUST be `AppResponse::ZomeCalled` containing a MsgPack serialization of the zome function's return value if successful, or `AppResponse::Error` containing one of the following errors:
    - For unauthorized zome calls, `ExternalApiWireError::ZomeCallUnauthorized(s)`, where `s` is a message that describes why the call was unauthorized.
    - For zome calls that attempt to initiate, process, or commit a countersigned entry, `ExternalApiWireError::CountersigningSessionError(s)`, where `s` is a message that describes the nature of the failure.
    - For all other errors, including errors returned by the zome function itself, `ExternalApiWireError::InternalError(s)`, where `s` describes the nature of the error.
- `CreateCloneCell(CreateCloneCellPayload) -> CloneCellCreated(ClonedCell)`: Clone a DNA, thus creating a new `Cell`.
  - **Notes:** This call specifies a DNA to clone by its `role_id` as specified in the app bundle manifest. The function MUST register a new DNA with a unique ID and the specified modifiers, create a new cell from this cloned DNA, and add the cell to the specified app. If at least one modifier is not distinct from the

original DNA, or the act of cloning would result in a clone with the same DNA hash as an existing cell in the app, the call MUST fail.

- **Arguments**: The payload is defined as:

```
struct CreateCloneCellPayload {
    // The DNA to clone, by role name.
    role_name: RoleName,
    // Modifiers to set for the new cell.
    // At least one of the modifiers must be set to obtain a distinct
    // hash for the clone cell's DNA.
    modifiers: DnaModifiersOpt<YamlProperties>,
    // Optionally set a proof of membership for the clone cell.
    membrane_proof: Option<MembraneProof>,
    // Optionally set a human-readable name for the DNA clone.
    name: Option<String>,
}
```

- **Return value**: The payload is defined as:

```
struct ClonedCell {
    cell_id: CellId,
    // A conductor-local clone identifier.
    clone_id: CloneId,
    original_dna_hash: DnaHash,
    // The DNA modifiers that were used to instantiate this clone cell.
    dna_modifiers: DnaModifiers,
    // The name the cell was instantiated with.
    name: String,
    // Whether or not the cell is running.
    enabled: bool,
}
```

- `DisableCloneCell(DisableCloneCellPayload) -> CloneCellDisabled`: Disable a clone cell.
  - **Notes:** When the clone cell exists, it is disabled, after which any zome calls made to the cell MUST fail and functions scheduled by the cell MUST be unscheduled. Additionally, any API calls that return `AppInfo` should show a disabled status for the given cell. If the cell doesn't exist or is already disabled, the call MUST be treated as a no-op. Deleting a cloned cell can only be done from the Admin API, and cells MUST be disabled before they can be deleted.
  - **Arguments**: The payload is defined as:

```
struct DisableCloneCellPayload {
    clone_cell_id: CloneCellId,
}
```

- `EnableCloneCell(EnableCloneCellPayload) -> CloneCellEnabled(ClonedCell)`: Enabled a clone cell that was previously disabled or not yet enabled.
  - **Notes:** When the clone cell exists, it MUST be enabled, after which any zome calls made to the cell MUST be attempted. Additionally any API functions that return `AppInfo` should show an enabled status for the given cell. If the cell doesn't exist, the call MUST be treated as a no-op.
  - **Arguments:** The payload is defined as:

```
struct EnableCloneCellPayload {
    clone_cell_id: CloneCellId,
}
```

- `GetNetworkInfo(NetworkInfoRequestPayload) -> NetworkInfoReturned(Vec<NetworkInfo>)`: Get information about networking processes.
  - **Arguments**: The payload is defined as:

```
struct NetworkInfoRequestPayload {
    // Get gossip info for these DNAs.
```

```
    // Implementations MUST restrict results to DNAs that are part of
    // the app.
    dnas: Vec<DnaHash>,
    // Timestamp in milliseconds since which received amount of bytes
    // from peers will be returned. Defaults to UNIX_EPOCH.
    last_time_queried: Option<Timestamp>,
}
```

  - **Return value**: The payload is defined as:

```
struct NetworkInfo {
    fetch_pool_info: FetchPoolInfo,
    current_number_of_peers: u32,
    arc_size: f64,
    total_network_peers: u32,
    bytes_since_last_time_queried: u64,
    completed_rounds_since_last_time_queried: u32,
}

struct FetchPoolInfo {
    // Total number of bytes expected to be received through fetches.
    op_bytes_to_fetch: usize,
    // Total number of ops expected to be received through fetches.
    num_ops_to_fetch: usize,
}
```

- `ListWasmHostFunctions -> ListWasmHostFunctions(Vec<String>)`: List all the host functions supported by this conductor and callable by WASM guests.
- `ProvideMembraneProofs(HashMap<RoleName, MembraneProof) -> Ok`: Provide the deferred membrane proofs that the app is awaiting.
  - **Arguments**: The input is supplied as a mapping of role names to the corresponding membrane proofs.
  - **Return value**: Implementations MUST return `AppResponse::Error` with an informative message if the application is already enabled.
- `EnableApp -> Ok`: Enable an app which has been awaiting, and has received, deferred membrane proofs.
  - **Notes**: If the app is awaiting deferred membrane proofs, implementations MUST NOT allow an app to be enabled until the membrane proofs has been provided.
  - **Return value**: If this call is attempted on an already running app or an app that is still awaiting membrane proofs, implementations MUST return `AppResponse::Error` with an informative message.