Holochain

scalable agent-centric distributed computing

PRE-RELEASE DRAFT: October, 2017

Eric Harris-Braun, Nicolas Luck, Arthur Brock[1]

[1] *Ceptr, LLC*

ABSTRACT : We present a scalable, evolvable, agent-centric distributed computing platform. We use a formalism to characterize distributed systems, show how it applies to some existing distributed systems and demonstrate the benefits of shifting from a data-centric to an agent-centric model. We present a detailed formal specification of the Holochain system, along with an analysis of its systemic integrity, capacity for evolution, total system computational complexity, implications for use-cases, and current implementation status.

I. INTRODUCTION

Distributed computing platforms have achieved a new level of viability with the advent of two foundational cryptographic tools: secure hashing algorithms, and public-key encryption. These have provided solutions to key problems in distributed computing: verifiable, tamper-proof data for sharing state across nodes in the distributed system, and confirmation of data provenance via digital signature algorithms. The former is achieved by hash-chains, where monotonic data-stores are rendered intrinsically tamper-proof (and thus confidently sharable across nodes) by including hashes of previous entries in subsequent entries. The latter is achieved by combining cryptographic encryption of hashes of data and using the public keys themselves as the addresses of agents, thus allowing other agents in the system to mathematically verify the data's source.

Though hash-chains help solve the problem of independently acting agents reliably sharing state, we see two very different approaches in their use which have deep systemic consequences. These approaches are demonstrated by two of today's canonical distributed systems:

1. git[1]: In git all nodes can update their hash-chains as they see fit. The degree of overlapping shared state of chain entries (known as commit objects) across all nodes is not managed by git but rather explicitly by action of the agent making pull requests and doing merges. We call this approach ***agent-centric*** because of its focus on allowing nodes to share independently evolving data realities.

2. Bitcoin[2]: In Bitcoin (and blockchain in general), the "problem" is understood to be that of figuring out how to choose one block of transactions among the many variants being experienced by the mining nodes (as they collect transactions from clients in different orders), and committing that single variant to the single globally shared chain. We call this approach ***data-centric*** because of its focus on creating a single shared data reality among all nodes.

We claim that this fundamental original stance results directly in the two most significant limitations of the blockchain: scalability and evolvability. These limitations are widely known [3] and many solutions have been offered [4]. Holochain offers a way forward by directly addressing the root data-centric assumptions of the blockchain approach.

II. PRIOR WORK

This paper builds largely on recent work in cryptographic distributed systems and distributed hash tables and multi-agent systems.

Ethereum: Wood [EIP-150], DHT: [Kademlia] Benet [IPFS]

dicussion and more references here

III. DISTRIBUTED SYSTEMS

A. Formalism

We define a simple generalized model of a distributed system Ω using hash-chains as follows:

1. Let N be the set of elements $\{n_1, n_2, \ldots n_n\}$ participating in the system. Call the elements of N ***nodes*** or ***agents***.

2. Let each node n consist of a set S_n with elements $\{\sigma_1, \sigma_2, \ldots\}$. Call the elements of S_n the ***state*** of node n. For the purposes of this paper we assume $\forall \sigma_x \in S_n : \sigma_x = \{\mathcal{X}_x, D_x\}$ with $\mathcal{X}_x$ being a ***hash-chain*** and D a set of non-hash chain ***data elements***.

[1] https://git-scm.com/about

[2] https://bitcoin.org/bitcoin.pdf

[3] add various sources

[4] more footnotes here

3. Let $\tau(\sigma_i, t) = \sigma_{i+1}$ where t is arbitrary input which τ transforms to produce σ_{i+1}. Call τ a ***state transition*** function. Call t a ***transaction***.

4. Let $V(t, v)$ be a function that takes t, along with extra validation data v, verifies the validity of t and only if valid calls a transition function for t. Call V a ***validation*** function.

5. Let H be a cryptographically secure hash function.

6. Let $\tau_{\mathcal{X}}(\mathcal{X}_i, e) = \mathcal{X}_{i+1}$ where

$$\begin{aligned} \mathcal{X}_{i+1} &= \mathcal{X}_i \cup \{x_{i+1}\} \\ &= \{x_1, \ldots, x_i, x_{i+1}\} \end{aligned} \tag{3.1}$$

   with

$$\begin{aligned} x_{i+1} &= \{h, e\} \\ h &= \{H(e), H(x_a, \ldots, x_b) | a, b \leq i\} \end{aligned} \tag{3.2}$$

   Call h a ***header*** and note how the sequence of headers creates a chain (tree, in the general case) by linking each header to the previous header(s) and the entry.

7. Let there be functions $\tau_{\mathcal{X}}$ and τ_{D} in Ω for transforming S

8. Let $I_n(t)$ be a function that takes a transaction t, evaluates it using a function V_x appropriate to the transaction type, and if valid uses τ_x to transform S. Call I the ***input*** or ***stimulus*** function.

9. Let $P_n(x)$ be a function that can create transactions t and trigger functions V_x and τ_x, and P itself is triggered by state changes or the passage of time. Call P the ***processing*** function.

10. Let C be a channel that allows all nodes in N to communicate and over which each node has a unique address A_n. Call C and the nodes that communicate on it the ***network***

11. Let $E_n(i)$ be a function that changes functions V, I, P. Call E the ***evolution*** function.

Explanation: this formalism allows us to model separately key aspects of agents.

First we separate the agent's state into a cryptographically secured hash-chain part $\mathcal{X}$ and another part that holds arbitrary data D. Then we split the process of updating the state into two steps: 1) the validation of new transactions t through the validation function $V_x(t, v)$, and 2) the actual change of internal state S (as either $\mathcal{X}$ or D) through the according state transition function τ_x. Finally, we distinguish between 1) state transitions triggered by external events, stimuli, received through $I_n(t)$, and 2) a node's internal processing $P_n(x)$ that also results in calling V_X and τ_x with an internally created transaction.

We define some key properties of distributed systems:

1. Call a set of nodes in N for which any of the functions T, V, P and E have the properties of being both reliably known, and known to be identical for that set of nodes: ***trusted*** nodes with respect to the functions so known.

2. Call a channel C with the property that messages in transit can be trusted to arrive exactly as sent: ***secure***.

3. Call a channel C on which the address A_n of a node n is $A_n = H(pk_n)$, where pk_n is the public key of the node n, and on which all messages include a digital signature of the message signed by sender: ***authenticated***

4. Call a data element that is accessible by its hash ***content addressable***.

For the purposes of this paper we assume untrusted nodes, i.e. independently acting agents solely under their own control, and an insecure channel. We do this because the very *raison d'être* of the cryptographic tools mentioned above is to allow individual nodes to trust the whole system under this assumption. The cryptography immediately makes visible in the state data when any other node in the system uses a version of the functions different from itself. This property is often referred to as a ***trustless*** system. However, because it simply means that the locus of trust has been shifted to the state data, rather than other nodes, we refer to it as systemic reliance on ***intrinsic data integrity***. See IV C for a detailed discussion on trust in distributed systems.

B. Data-Centric and Agent-Centric Systems

Using this definition, Bitcoin can be understood as that system $\Omega_{\mathrm{bitcoin}}$where:

1. $\forall n, m \in N : \mathcal{X}_n \stackrel{!}{=} \mathcal{X}_m$

2. $V_n(e, v)$ e is a block and v is the output from the "proof-of-work" hash-crack algorithm, and V_n confirms the validity of v, the structure and validity of e according to the double-spend rules[5].

3. $I_n(t, n)$ accepts transactions from clients and adds them to D_n (the *mempool*) to build a block for later use in triggering $V()$

4. $P_n(i)$ is the *mining* process including the "proof-of-work" algorithm and composes with $V_{\mathrm{C}}()$ and τ_{C} when the hash is cracked.

[5] pointer here

5. $E_n(i)$ is not formally defined but can be mapped informally to a decision by humans operating the nodes to install new versions of the Bitcoin software.

The first point establishes the central aspect of Bitcoin's (and blockchain applications' in general) strategy for solving or avoiding problems otherwise encountered in decentralized systems, and that is by trying to maintain a network state in which all nodes **should** have the same (local) chain.

By contrast, for Ω_{git}there is no such constraint on any $\mathcal{X}_n$, $\mathcal{X}_m$ in nodes n and m matching, also $V_{\text{C}}(e, v)$ only checks the structural validity of e as a commit object not it's content, and $I_n(t)$ can be understood to be the set git commands available to the user, and τ_{C} the function that adds a commit object and τ_{D} the function that adds code to the `index` triggered by `add`. E is, similarly to Ω_{bitcoin}, not formally defined for Ω_{git}. more detailed comparison with git?

Our model of a distributed system makes one thing very clear about the difference between $\Omega_{\text{bitcoin}}$and Ω_{git}. As the size of $\mathcal{X}_n$ grows, necessarily all nodes of Ω_{bitcoin}must grow in size, whereas this is not necessarily the case for Ω_{git}. Though this seems like a clear limitation, it comes as a direct consequence of the constraint of $\forall n, m \in N : \mathcal{X}_n \stackrel{!}{=} \mathcal{X}_m$. Interestingly this is actually considered Bitcoin's strength as it defines the heart of "consensus" in the blockchain world, i.e. a limitation and method to the achieve sameness of state of all nodes.

It's not surprising that a data-centric approach was used for Bitcoin. This comes from the fact that its stated intent was to create digitally transferable "coins", i.e. to model in a distributed digital system that property of matter known as location. On centralized computer systems this doesn't even appear as a problem, because centralized systems have been designed to allow us to think from a data-centric perspective. They allow us to believe in a kind of data objectivity, as if data exists, like a physical object sitting someplace having a location. They allow us to think in terms of an absolute frame - as if there *is* a correct truth about data and/or time sequence, and suggests that "consensus" should converge on this truth. In fact, this is not a property of information. Data exists always from the vantage point of an observer. It is this fact that makes digitally transferable "coins" a hard problem in distributed systems where there is always more than one vantage point.

In the distributed world events don't happen in the same sequence for all observers. For Blockchain specifically this is the heart of the matter: choosing which block from all the nodes receiving transactions in different orders, to use for the the "consensus," i.e. what single vantage point for enforce on all nodes. Blockchains don't record a universal ordering of events – they manufacture a single authoritative ordering of events, by stringing together a tiny fragment of local vantage points into one global record that has passed validation rules.

The use of the word consensus seems at best dubious as a description of a systemic requirement that all nodes carry identical values of $\mathcal{X}_n$. Especially when the algorithm for ensuring that sameness is essentially a digital lottery powered by expensive computation that's primary design feature is to randomize which node gets to run V_n such that no node has preference to which e gets added to $\mathcal{X}_n$.

Consensus as normally used implies deliberation with regard to differences and work on crafting a perspective that holds for all parties, rather than simply selecting one party's dataset at random.

A more accurate term for the algorithm would be "proof-of-luck" and for the process itself simply sameness, not consensus. If you start from a data-centric view point, which naturally throws out the "experience" of all agents in favor of just one, it's much harder to design them to engage in processes that actually have the real-world properties of consensus. If the constraint of keeping all nodes' states the same were adopted consciously as a fit for a specific purpose, this would not be particularly problematic. Unfortunately the legacy of this data-centric view point has been held mostly unconsciously and is adopted by more generalized distributed computing systems, for which the intent doesn't specifically include the need to model "digital matter" with universally absolute location. While having the advantages of simplicity it also immediately transfers to them the scalability issues, but worse, it makes it hard to take advantages inherent in agent-centric approach.

C. Systemic Evolvability

Neither of the systems discussed so far address the question of evolvability directly in-system. The evolution function E is not formalized and left to humans operating the nodes. This has the very interesting side-effect of, in practice, obviating the very decentralization aimed for by the design of these systems and relegating it squarely back in traditional practices [DUPONT]. For truly successful distributed systems, this cannot be left out of the system. We discuss evolvability more fully in Section V

IV. GENERALIZED DISTRIBUTED COMPUTATION

The previous section described a general formalism for distributed systems, and compared git to Bitcoin as an example of an agent-centric vs. a data-centric distributed system. Neither of these systems, however provides generalized computation in the sense of being a framework for writing computer programs or creating applications. So, lets add the following constraints to formalism III A as follows:

1. With respect to a machine M some values of S_n can be interpreted as: executable code, and the results

of code execution, and may be accessible to M and the code. Call such values the ***machine state***.

2. $\exists t$ and nodes n such that $I_n(t)$ will trigger execution of that code. Call such transaction values ***calls***.

A. Ethereum

Ethereum[6] provides the current premier example of generalized distributed computing using the blockchain model. It lives up to the constraints listed above as described by Wood [EIP-150] where the bulk of the paper can be understood as a specification of a validation function $V_n()$ and the described state transition function $\sigma_{t+1} \equiv \Upsilon(\sigma, T)$ as a specification of how constraints above are met. Unfortunately the data-centric legacy inherent in Ethereum is immediately observable in its high compute cost[7] and difficulty in scaling[8].

flesh out more exactly how this is a data-centric approach to the constraints above

B. Holochain

We now proceed to describe an agent-centric distributed generalized computing system, where nodes can still confidently participate in the system as whole even though they are not constrained to maintaining the same chain state as all other nodes.

In broad strokes: A Holochain is a unique source chain for every agent, paired with a validating, monotonic, sharded, distributed hash table (DHT) where every node enforces validation rules on data in the DHT as well as verifying provenance of data from the source chains where it originated.

Using our formalism, a Holochain based application $\Omega_{\text{hc}}$is defined as:

1. Call $\mathcal{X}_n$ the ***source chain*** of n.

2. Let M be a virtual machine used to execute code.

3. Let the initial entry of all $\mathcal{X}_n$ in N be identical and consist in the set $\text{DNA}\{e_1, e_2, \dots, f_1, f_2, \dots, p_1, p_2, \dots\}$ where e_x are definitions of entry types that can be added to the chain, f_x are functions defined as executable on M (which we also refer to as the set $F_{\text{app}} = \{app_1, app_2, \dots\}$) and p_x are various system properties defined later.

4. Let ι_n be the second entry of all $\mathcal{X}_n$ and be set of the form $\{p, i\}$ where p is the public key, and i is identifying information appropriate to the use of this particular Ω_{hc}. Note that though this entry is of the same format for all $\mathcal{X}_n$ it's content is not the same. Call this entry the ***agent identity*** entry.

5. $\forall e_x \in DNA$ let there be an $app_x \in F_{\text{app}}$ which can be used to validate transactions that involve entries of type e_x. Call this set F_{v} or the ***application validation functions***.

6. Let there be a function $V_{\text{sys}}(ex, e, v)$ which checks that e is of the form specified by the entry definition for $e_x \in$ DNA. Call this function the ***system entry validation function***

7. Let the overall validation function $V(e, v) \equiv \bigvee_x F_{\text{v}}(e_x)(v) \wedge V_{\text{sys}}(e_x, e, v)$.

8. Let F_{I} be a subset of F_{app} distinct from F_{v} such that $\forall f_x(t) \in F_{\text{I}}$ there exists a t to $I(t)$ that will trigger $f_x(t)$. Call the functions in F_{I} the ***exposed functions***.

9. Call any functions in F_{app} not in F_{v} or F_{I} ***internal functions*** and allow them to be called by other functions.

10. Let the channel C be ***authenticated***

11. Let DHT define a distributed hash table on an authenticated channel as follows:

    (a) Let Δ be a set $\{\delta_1, \delta_2, \dots\}$ where δ_x is a set $\{key, value\}$ where key is always the hash $H(value)$ of $value$. Call Δ the ***DHT state***. need to specify that δs are unique or is that totally obvious?

    (b) Let F_{DHT} be the set of functions $\{dht_{\text{put}}, dht_{\text{get}}\}$ where:

        i. $dht_{\text{put}}(\delta_{\text{key,value}})$ adds $\delta_{\text{key,value}}$ to Δ
        ii. $dht_{\text{get}}(key) = value$ of $\delta_{\text{key,value}}$ in Δ

    (c) Assume $x, y \in N$ and $\delta_i \in \Delta_x$ but $\delta_i \notin \Delta_y$. Allow that when y calls $dht_{\text{get}}(key)$ δ_i will be retrieved from x over channel X and added to Δ_y.

    DHT are sufficiently mature that there are a number of ways to ensure property 11c. For our implementation we use [Kademlia]. Should this really be S/Kademlia?

12. Let DHT_{hc} augment DHT as follows:

    (a) Let q and r be parameters of DHT_{hc} that are to be set dependent on the characteristics deemed beneficial for the given application. Call q the ***neighborhood size*** and r the ***redundancy factor***.

[6] https://github.com/ethereum/wiki/wiki/White-Paper
[7] link to that article
[8] anther link

(b) $\forall \delta_{\text{key,value}} \in \Delta$ constrain *value* to be an entry type as defined in DNA. We do this by ensuring that any function call $dht_x(y)$ which would modify Δ does so $\iff F_{\text{v}}(y)$ indicates that y is valid.

(c) Enforce that all elements of Δ only be changed monotonically, that is, elements δ can only be added to Δ not removed.

(d) Include in F_{DHT} the functions defined in IX

(e) Allow the sets $\delta \in \Delta$ to also include more elements as defined in IX

(f) Let $d(x, y)$ be a *symmetric* and *unidirectional* distance metric within the hash space defined by H, as for example the XOR metric defined in [Kademlia]. Note that this metric can be applied between entries and nodes alike since the addresses of both are values of the same hash function H (i.e. $\delta_{key} = H(\delta_{value})$ and $A_n = H(pk_n)$).

(g) Let $\nu(A_n, q) = \psi_n$ be the set of q nodes $\{n_1, n_2, \ldots, n_q\}$ closest to n, so that $\forall n_i \in \psi_n, n_0 \notin \psi_n$:

$$d(A_n, A_{n_0}) > d(A_n, A_{n_i}) \\ \wedge |\psi_n| = q. \tag{4.1}$$

Call ψ_n a ***neighborhood*** of size q in N around n. Enforce that nodes maintain a reasonable up-to date representation of their neighborhood (of application specific size q) as peers appear and disappear from the network.

(h) Enforce that every node n shares its elements in Δ_n with all nodes in its neighborhood ψ_n. Call this sharing ***gossip***.

(i) Allow every node n to discard every $\delta_x \in \Delta_n$ if the number of closer (with regards to $d(x, y)$) nodes

$$\rho_x = |\{n_i | d(A_{n_i}, \delta_{x,key}) < d(A_n, \delta_{x,key}\}| \tag{4.2}$$

is greater than the ***redundancy factor*** r. Note that this results in the network adapting to changes in topology and DHT state migrations by regulating the number of network-wide redundant copies of all $\delta_i \in \Delta$ to match r. This implies the existence of r partitions of N called ***shards*** that each (among all their nodes $N_{Shard_i} \subset N$) hold a complete redundant copy of Δ :

$$\bigcup_{n \in N_{Shard_i}} \Delta_n = \Delta. \tag{4.3}$$

One sentence about about the shards being holographic.

Call DHT_{hc} a ***validating***, ***monotonic***, ***sharded*** DHT.

13. $\forall n \in N$ assume n implements DHT_{hc}, that is: Δ is a subset of D (the non hash-chain state data), and F_{DHT} are available to n, though note that these functions are NOT directly available to the functions F_{app} defined in DNA.

14. Let F_{sys} be the set of functions $\{sys_{\text{commit}}, sys_{\text{get}}, \ldots\}$ where:

    (a) $sys_{\text{commit}}(e)$ uses the system validation function $V(e, v)$ to add e to $\mathcal{X}$, and if successful calls $dht_{\text{put}}(H(e), e)$.

    (b) $sys_{\text{get}}(k) = dht_{\text{get}}(k)$

    (c) ... functions defined in X

15. Allow the functions in F_{app} defined in the DNA to call the functions in F_{sys}.

16. Let m be an arbitrary message. Include in F_{sys} the function $sys_{\text{send}}(A_{\text{to}}, m)$ which when called on n_{from} will trigger the function $app_{\text{receive}}(A_{\text{from}}, m)$ in the DNA on the node n_{to}. Call this mechanism ***node-to-node messaging***.

17. Allow that the definition of entries in DNA can mark entry types as ***private***. Enforce that if an entry σ_x is of such a type then $\sigma_x \notin \Delta$. Note however that entries of such type can be sent as node-to-node messages.

18. Let the system processing function $P(i)$ TODO

C. Systemic Integrity Through Validation

The appeal of the data-centric approach to distributed computing comes from the fact that if you can prove that all nodes reliably have the same data then that provides strong general basis from which to prove the integrity of the system as a whole. In the case of Bitcoin, the $\mathcal{X}$ holds the transactions and the unspent transaction outputs which allows nodes to verify future transactions against double-spend. In the case of Ethereum, $\mathcal{X}$ holds essentially pointers to machine state better one-sentence wording for Ethereum. Proving the consistency across all nodes of those data sets is fundamental to the integrity of those systems.

However, because we have started with the assumption (see III A) of distributed systems of independently acting agents, any *proof* of $\forall n, m \in N : \mathcal{X}_n \stackrel{!}{=} \mathcal{X}_m$ in blockchain based system is better understood as a *choice* (hence our use of the $\stackrel{!}{=}$), in that nodes use their agency to decide when to stop interacting with other nodes based on detecting that the $\mathcal{X}$ state no longer matches. This might also be called "proof by enforcement," and is also appropriately known as a ***fork*** because essentially it results in partitioning of the network.

The heart of the matter has to do with the trust any single agent has is in the system. In [EIP-150] Section 1.1 (Driving Factors) we read:

> Overall, I wish to provide a system such that users can be guaranteed that no matter with which other individuals, systems or organisations they interact, they can do so with absolute confidence in the possible outcomes and how those outcomes might come about.

The idea of "absolute confidence" here seems important, and we attempt to understand it more formally and generally for distributed systems.

1. Let Ψ_α be a measure of the confidence an agent has in various aspects of the system it participates in, where $0 \leq \Psi \leq 1$ and 0 represents no confidence and 1 represents absolute confidence.

2. Let $R_n = \{\alpha_1, \alpha_2, \ldots\ldots\}$ define a set of aspects about the system with which an agent $n \in N$ measures confidence. Call R_n the ***requirements*** of n with respect to Ω.

3. Let $\varepsilon_n(\alpha)$ be a thresholding function for node $n \in N$ with respect to α such that when $\Psi_\alpha < \varepsilon(\alpha)$ then n will either stop participating in the system, or reject the participation of others (resulting in a fork).

4. Let R_A and Let R_C be partitions of R where

$$\begin{aligned} \forall \alpha \in R_A : \varepsilon(\alpha) = 1 \\ \forall \alpha \in R_C : \varepsilon(\alpha) < 1 \end{aligned} \tag{4.4}$$

   so any value of $\Psi \neq 1$ is rejected in R_A and any value $\Psi < \varepsilon(\alpha)$ is rejected in R_C. Call R_A the ***absolute requirements*** and R_C the ***considered requirements***.

So we formally separated system characteristics that we have absolute confidence in (R_A) from those we only have considered confidence in (R_C). Still unclear is how to measure a concrete confidence level Ψ_α. In real-world contexts and for real-world decisions, this seems to be a soft criteria mainly dependent on an (human) agent's vantage point, set of data at hand and maybe even intuition. In order to comprehend this concept objectively and akin to the notion conveyed by Woods in the quote above, we proceed by defining the measure of confidence of an aspect α as the conditional probability of it being the case in a given context:

$$\Psi_\alpha \equiv \mathcal{P}(\alpha|\mathcal{C}) \tag{4.5}$$

where the context $\mathcal{C}$ models all other information available to the agent, including basic and intuitive assumptions.

Consider the fundamental example of cryptographically signed messages with asymetric keys as applied throughout the field of cryptographic systems (basically what coins the term crypto-currency). A core aspect to evaluate is: $\alpha_{signature}$ because it provides us with the ability to *know with certainty* that a given message's real author $Author_{real}$ is the is the same agent indicated solely via locally available data in the message's meta information through the cryptographic signature $Author_{local}$. We get this confidence because it is presumeably *very hard* for any agent not in possession of the private key to create a valid signature for a given message.

$$\alpha_{signature} \equiv Author_{real} = Author_{local} \tag{4.6}$$

The appeal of this aspect is that we can check authorship locally, i.e. without the need of a 3rd party or direct trusted communication channel to the real author. But, the confidence in this aspect of a certain cryptographic system depends on the context $\mathcal{C}$.

$$\Psi_{signature} = \mathcal{P}(Author_{real} = Author_{local}|\mathcal{C}) \tag{4.7}$$

If we constrain the context to remove the possiblity of an adversary gaining access to an agent's private key and also exclude the possible (future) existence of computing devices or algorithms that could easily calculate or brute force the key, we might then assign a (constructed) confidence level of 1, i.e. "absolute confidence". Without such constraints on $\mathcal{C}$ we must admit that $\Psi_{signature} < 1$, which real world events, for instance the Mt.Gox hack from 2014 [**?**], make clear.

Our motivation to describe these relationships in such detail is to point out that any set R_A of *absolute requirements* can't reach beyond trivial statements - statements about the local state of the agent itself. This is akin to Descarte's way of arriving at his famous statement *cogito ergo sum* after questioning the confidence in every thought along the way. **The only aspect any agent can have candid absolute confidence in is that it perceives a certain thing to be in a certain way or that it has chosen to act in a certain way,** i.e. that it's functions τ_x, V_x, I_n and P_n led to its state being mutated to the current state S_n. This is the heart of the agent-centric outlook, and what we claim must always be taken into account in the design of decentralized multi-agent systems, as it shows that any aspect of the system as a whole that includes assumptions about other agents and non-local events must be in R_C, i.e. have an a priori confidence of $\Psi < 1$. Facing this truth about multi-agent systems, we find little value in trying to force an absolute truth $\forall n, m \in N : \mathcal{X}_n \stackrel{!}{=} \mathcal{X}_m$ and we frame the problem:

> We wish to provide means by which decentralized multi-agent systems can be built so that
>
> 1. fit-for-purpose solutions can be applied in order to optimize for application contextualized confidences Ψ_α
> 2. violation of any threshold $\varepsilon(\alpha)$ through the actions of other agents can be detected and managed by any agent, such that
> 3. the system integrity is maintained at any point in

time or, when not, there is a path to regain it (see V).

We perceive the agent-centric solution to these requirements to be the holographic management of system-integrity within every agent/node of the system through application specific validation routines. These sets of validation rules is what is at the heart of every decentralized application. They have to be very different for very different applications. Every agent carefully keeps track of their representation of that part of reality that is of importance to them - within the context of that application.

For example, consider two different contexts for transactions:

1. delivery of an email message and we are trying to validate it as spam or not
2. commit of monetary transaction where we are trying to validate it against double-spend

These contexts have different consequences that an agent may wish to evaluate differently and may be willing to attach more or less resources to in validating. One of the key design elements of Holochain was to allow such validation functions to be set contextually per application, and expose these contexts explicitly. It is conceivable of a Holochain application that deliberately makes choices in it's validation functions to implement either all or partial characteristics of Blockchains. Holochain, therefore, can be understood as a framework that opens up a spectrum of decentralized application architectures in which Blockchain happens to be one, because it is unscalable if applied to all contexts.

D. Foundational Properties for Systemic Integrity

1. Intrinsic Data Integrity TODO
2. CALM & Logical Monotonicity TODO
3. Membranes TODO
4. Provenance TODO
5. ... TODO

E. Patterns of Trust Management

Tools in Holochain available to app developers for use in Considered Requirements, some of which are also used at the system level an globally parameterized for an application.

1. Countersigning TODO
2. Notaries TODO – "The network is the notary."
3. Publish Headers e.g. for chain-rollback detection
4. Source-chain examination. TODO
5. Blocked-lists. e.g. DDOS, spam, etc
6. ... more here...

F. Bridging

Here we discuss connection of services through agency. Ramifications and benefits.

V. EVOLVABILITY

As stated in section IV C, a key aspect of systemic integrity, is the capacity of that system to evolve. In this section we discuss the aspects of Holochain's design that take steps toward increasing evolvability by explicitly introducing governance mechanisms into the system which take into account application to versioning in-system.

1. TODO: DNA as first entry, we know what game we're play
2. TODO: Governance within the system: pluggable governance
3. TODO: Stage 1: Closing/Redirect entries
4. TODO: Stage 2: DNA revocation/rewrite in-chain

VI. COMPLEXITY IN DISTRIBUTED SYSTEMS

In this section we discuss the complexity of our proposed architecture for decentralised systems and compare it to the increasingly adopted Blockchain pattern.

Formally describing the complexity of decentralized multi-agent systems is a non-trivial task for which more complex approaches have been suggested ([Marir2014]). This might be the reason why there happens to be unclarity and misunderstandings within communities discussing complexity and scalability of Bitcoin for example [Bitcoin Reddit].

In order to be able to have a ball-park comparison between our approach and the current status quo in decentralized application architecture, we proceed by modeling the worst-case time complexity both for a single node $\Omega_{SystemNode}$ as well as for the whole system Ω_{System} and both as functions of the number of state transitions (i.e. transactions) n and the number of nodes in the system m.

A. Bitcoin

Let $\Omega_{Bitcoin}$ be the Bitcoin network, n be the number of transactions and m be the number full validating nodes (i.e. *miners*[9]) within $\Omega_{Bitcoin}$.

For every new transaction being issued, any given node will have to check the transaction's signature (among other checks, see. [BitcoinWiki]) and especially check if this transaction's output is not used in any other transaction to reject double-spendings, resulting in a time complexity of

$$c + n \tag{6.1}$$

per transaction. The time complexity in big-O notation per node as a function of the number of transactions is therefore

$$\Omega_{BitcoinNode} \in O(n^2). \tag{6.2}$$

The complexity handled by one Bitcoin node does not [10] depend on m the number of total nodes of the system. But since every node has to validate exactly the same set of transactions, the system's time complexity as a function of number of transactions and number of nodes results as

$$\Omega_{Bitcoin} \in O(n^2 m). \tag{6.3}$$

Note that this quadratic time complexity of Bitcoin's transaction validation process is what creates its main bottleneck as this reduces the network's gossip bandwidth since every node has to validate every transaction before passing it along. In order to still have an average transaction at least flood through 90% of the network, block size and time can't be pushed beyond 4MB and 12s respectively, according to [Croman et al 16].

B. Ethereum

Let $\Omega_{Ethereum}$ be the Ethereum main network, n be the number of transactions and m the number of full-clients within in the network.

The time complexity of processing a single transaction on a single node is a function of the code which's execution is being triggered by the given transaction plus a constant:

$$c + f_{tx_i}(n, m). \tag{6.4}$$

Similarly to Bitcoin and as a result of the Blockchain design decision to maintain one single state ($\forall n, m \in N : \mathcal{X}_n \stackrel{!}{=} \mathcal{X}_m$, *"This is to be avoided at all costs as the uncertainty that would ensue would likely kill all confidence in the entire system."* [EIP-150]), every node has to process every transaction being sent resulting in a time complexity per node as

$$c + \sum_{i=0}^{n} f_{tx_i}(n, m) \tag{6.5}$$

resulting in

$$\Omega_{EthereumNode} \in O(n \cdot f_{tx_i}(n, m)) \tag{6.6}$$

whereas the complexity $f_{tx_i}(n, m)$ of the code being run by Ethereum is typically held simple since execution has to be payed for in gas and is due to other restrictions such as the *block gas limit*. In other words, because of the complexity $f_{tx_i}(n, m)$ being burdened upon all nodes of the system, other systemic properties have to keep users from running complex code on Ethereum so as to not bump into the limits of the network.

Again, since every node has to process the same set of all transactions, the time complexity of the whole system then is that of one node multiplied by m:

$$\Omega_{Ethereum} \in O(nm \cdot f_{tx_i}(n, m)). \tag{6.7}$$

C. Blockchain

Both examples of Blockchain systems above do need a non-trivial computational overhead to work at all: the proof-of-work, hash-crack process also called *mining*. Since this overhead is not a function of either the number of transactions nor directly of the number of nodes, it is often omitted in complexity analysis. With the total energy consumption of all Bitcoin miners today being greater than the country of Iceland [Coppock17], neglecting the complexity of Blockchain's consensus algorithm seems like a silly mistake.

Blockchains set the block time, the average time between two blocks, as a fixed parameter that the system keeps in homeostasis by adjusting the hash-crack's difficulty according to the network's total hash-rate. For a given network with a given set of mining nodes and a given total hash-rate, the complexity of the hash-crack is constant. But as the system grows and more miners come on-line, which increases the networks total hash-rate, the difficulty needs to increase in order to keep the average block time constant.

With this approach, the benefit of a higher total hash-rate x_{HR} is an increased difficulty of an adversary to influence the system by creating biased blocks (which would render this party able to do double-spend attacks). That is why Blockchains have to subsidize mining, depending on a high x_{HR} as to make it economically im-

[9] For the sake of simplicity and focusing on a lower bound of the system's complexity, we are neglecting all nodes that are not crucial for the operation of the network, such as light-clients and clients not involved in the process of validation

[10] not inherently - that is more participants will result in more transactions but we model both values as separate parameters

possible for an attacker to overpower the trusted miners.

So, there is a direct relationship between the network's total trusted hash-rate and its level of security against mining power attacks. This means that the confidence $\Psi_{Blockchain}$ any agent can have in the integrity of the system is a function of the system's hash-rate x_{HR}, more precisely the cost/work $cost(x_{HR})$ needed to provide it. Looking only at a certain transaction t and given any hacker acts economically rational only, the confidence in t being added to all $\mathcal{X}_n$ has an upper bound in

$$\Psi_{Blockchain}(t) < min\left(1, \frac{cost(x_{HR})}{value(t)}\right) \tag{6.8}$$

In order to keep this confidence unconstrained by the mining process and therefore the architecture of Blockchain itself, $cost(x_{HR})$ (which includes the setup of mining hardware as well as the energy consumption) has to grow linearly with the value exchanged within the system.

D. Holochain

Let Ω_{HC} be a given Holochain system and let n be the sum of all public[11] (i.e. *put* to the DHT) state transitions (*transactions*) all agents in Ω_{HC} trigger in total and let m be the number of agents (= nodes) in the system.

Putting a new entry to the DHT involves finding a node that is responsible for holding that specific entry, which in our case according to [Kademlia] has a time complexity of

$$c + \lceil log(m) \rceil. \tag{6.9}$$

After receiving the state transition data, this node will gossip with its q neighbors which will result in r copies of this state transition entry being stored throughout the system - on r different nodes. Each of these nodes has to validate this entry which is an application specific logic of which the complexity we shall call $v(n, m)$.

Combined, this results in a system-wide complexity per state transition as given with

$$\underbrace{c + \lceil log(m) \rceil}_{DHTlookup} + q + r \cdot \underbrace{v(n, m)}_{validation} \tag{6.10}$$

which implies the following whole system complexity in O-notation

$$\Omega_{Holochain} \in O(n \cdot (log(m) + v(n, m)) \tag{6.11}$$

Now, this is the overall system complexity. In order to enable comparison, we reason that in the case of Holochain without loss of generality (i.e. dependent on the specific Holochain application) the load of the whole system is shared equally by all nodes. Without further assumptions, for any given state transition the probability of it originating at a certain node is $\frac{1}{m}$, so the term for the lookup complexity needs to be divided by m to describe the average lookup complexity per node. Other than in Blockchain systems where every node has to see every transaction, for the vast majority of state transitions one particular node is not involved at all. The stochastic closeness of the node's public key's hash with the entry's hash is what triggers the node's involvement. We assume the hash function H to show a uniform distribution of hash values which results in the probability of a certain node being one of the r nodes that cannot discard this entry to be $\frac{1}{m}$ times r. The average time complexity being handled by an average node then is

$$\Omega_{HolochainNode} \in O\left(\frac{n}{m} \cdot (log(m) + v(n, m))\right). \tag{6.12}$$

Note that the factor $\frac{n}{m}$ represents the average number of state transactions per node (i.e. the load per node) and that though this is a highly application specific value, it is an a priori expected lower bound since nodes have to process at least the state transitions they produce themselves.

The only overhead that is added by the architecture of this decentralized system is the node look-up with its complexity of $log(m)$.

The unknown and also application specific complexity $v(n, m)$ of the validation routines is what could drive up the whole system's complexity still. And indeed it is conceivable to think of Holochain applications with a lot of complexity within their validation routines. It is basically possible to mimic Blockchain's consensus validation requirement by enforcing that a validating node communicates with all other nodes before adding an entry to the DHT. It could as well only be half of all nodes. And there surely is a host of applications with only little complexity - or specific state transitions within an application that involve only little complexity. *In a Holochain app one can put the complexity where it is needed and keep the rest of the system fast and scalable.*

In section VII we proceed by providing real-world use cases and showing how non-trivial Holochain applications can be built that get along with a validation complexity of $O(1)$, resulting in a total time complexity per node in $O(log(m))$ and a high enough confidence in integrity without introducing proof-of-work at all.

VII. USE CASES

Now we present a few use cases of applications built on Holochain, considering the context of the use case and how it affects both complexity and evaluation of integrity

[11] private (see:17) state transitions, i.e. that are confined to a local $\mathcal{X}_n$, are completely within the scope of a node's agency and don't affect other parts of the system directly and can therefore be omitted for the complexity analysis of Ω_{HC} as a distributed system

and thus validation design.

A. Social Media

Consider a simple implementation of micro-blogging using Holochain where:

1. $F_{\mathrm{I}} = \{f_{\mathrm{post}}(text, node), f_{\mathrm{follow}}(node), f_{\mathrm{read}}(text)\}$ and
2. $F_{\mathrm{V}} = \{f_{\mathrm{isOriginator}}\}$

describe $O(1)$ complexity

B. Identity

DPKI

C. Money

mutual-credit vs. coins where the complexity of the transaction is higher, complexity may be $O(n^2)$ or $O(log(n))$ see holo currency white paper: [**?**]

VIII. IMPLEMENTATION

At the time of this writing we have a fully operational implementation of system as described in this paper, that includes two separate virtual machines for writing DNA functions in JavaScript, or Lisp, along with proof-of-concept implementations of a number of applications including a twitter clone, a slack-like chat system, DPKI, and a set mix-in libraries useful for building applications.

1. 30k+ lines of go code.
2. DHT: customized version of libp2p/ipfs's kademlia implementation.
3. Network Transport: libp2p including end-to-end encryption.
4. Javascript Virtual Machine: otto `https://github.com/robertkrimen/otto`
5. Lisp Virtual Machines: zygomys `https://github.com/glycerine/zygomys`

benchmarking tests
scalability tests

IX. APPENDIX I: $DHT_{\mathbf{hc}}$

1. $dht_{\mathrm{putLink}}(base, link, tag)$ where *base* and *link* are keys and where *tag* is an arbitrary string, which associates the tuple {link,tag} with the key *base*.
2. $dht_{\mathrm{getLinks}}(base, tag)$ where *base* is a key keys and where *tag* is an arbitrary string, which returns the set of links on *base* identified by *tag*.
3. $dht_{\mathrm{mod}}(key, newkey)$ where *key* and *newkey* are keys, which adds *newkey* as a modifier of $\sigma_{\mathrm{key}} \in \Delta$ and calls $dht_{\mathrm{putLink}}(key, newkey, \text{“}replacedby\text{”})$
4. $dht_{\mathrm{del}}(key)$ where *key* is a key, and marks $\sigma_{\mathrm{key}} \in \Delta$ as deleted.
5. modification to dht_{get} re mod & del

X. APPENDIX II: F_{sys}

1. all the other sys functions...

ACKNOWLEDGMENTS

We thank Steve Sawin for his review of this paper, LaTeX 2_{ε} support and so much more.....

XI. LUMBER

This is stuff that may or may not get used

1. *Informal description*

I'm not sure we should even do this informal description, cuz that exists elsewhere. All Holochain installations segment the computing space by application. Each application should be thought of as single distributed computing instance operating on a separate network from other applications. For each application, Holochain installations maintain separate nodes and state and communicate over separate isolated channels. Holochain applications can be connected, but only by external agents connecting one to another. This will be explained further in IV F. Each node participating in a Holochain application maintains a hash-chain. The first entry in the chain of all nodes is identical, and we call the DNA. The DNA consists in the entry type definitions, executable source code, and property definitions, and most importantly validation rules that define that application. Nodes also participate in operating a distributed hash table together.

When external agents wish to initiate a transaction on a node they control, they call an "exposed function" which is subset of the executable source code that's been

defined as part of the application's "API." These functions calls will result in changing state of the nodes only through adding one of the defined entry types to that node's local chain. Adding data to a source chain has the side-effect of doing a DHT `put`operation for that entry, where the key is the same hash of the entry used for building the hash-chain. Thus entries are accessible to all other nodes on the network via a DHT `get`of that hash. However, all nodes receiving a `put`must first validate it by verify it with the source node.

[DUPONT] Quinn DuPont. *Experiments in Algorithmic Governance: A history and ethnography of The DAO, a failed Decentralized Autonomous Organization* `http://www.iqdupont.com/assets/documents/DUPONT-2017-Preprint-Algorithmic-Governance.pdf`

[EIP-150] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger.* `http://yellowpaper.io/`

[Kademlia] Petar Maymounkov and David Mazieres *Kademlia: A Peer-to-peer Information System Base on the XOR Metric* `https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf`

[Zhang13] Zhang, H., Wen, Y., Xie, H., Yu, N. *Distributed Hash Table Theory, Platforms and Applications*

[Croman et al 16] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gn Sirer, Dawn Song, Roger Wattenhofer, *On Scaling Blockchains*, Financial Cryptography and Data Security, Springer Verlag 2016

[Bitcoin Reddit] /u/mike_hearn, /u/awemany, /u/nullc et al. `https://www.reddit.com/r/Bitcoin/comments/3a5f1v/mike_hearn_on_those_who_want_all_scaling_to_be/csa7exw/?context=3&st=j8jfak3q&sh=6e445294` Reddit discussion 2015

[Marir2014] Marir, Toufik and Mokhati, Farid and Bouchelaghem-Seridi, Hassina and Tamrabet, Zouheyr", *Complexity Measurement of Multi-Agent Systems"*, Multiagent System Technologies: 12th German Conference, MATES 2014, Stuttgart, Germany, September 23-25, 2014. Proceedings, Springer International Publishing 2014 `https://doi.org/10.1007/978-3-319-11584-9_13`

[Coppock17] Mark Coppock *THE WORLDS CRYPTOCURRENCY MINING USES MORE ELECTRICITY THAN ICELAND* `https://www.digitaltrends.com/computing/bitcoin-ethereum-mining-use-significant-electrical-power/`

[BitcoinWiki] *Bitcoin Protocol* `https://en.bitcoin.it/wiki/Protocol_rules#.22tx.22_messages` Bitcoin Wiki

[IPFS] Juan Benet *IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)* `https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4kipfs.draft3.pdf`