Holochain

Scalable agent-centric distributed computing
ALPHA – 2/15/2018

Eric Harris-Braun,[1] Nicolas Luck,[1] and Arthur Brock[1]
[1] *Ceptr, LLC*

ABSTRACT : We present a scalable, 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 most significant limitation of the blockchain: scalability. This limitation is widely known and many solutions have been offered. 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]

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 $\{n_1, n_2, \dots n_n\}$ 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 $\{\sigma_1, \sigma_2, \dots\}$. Call the elements of S_n the ***state*** of node n. For the purposes of this paper we assume $\forall \sigma_i \in S_n : \sigma_i = \{\mathcal{X}_i, D_i\}$ with $\mathcal{X}_i$ being a ***hash-chain*** and D a set of non-hash chain ***data elements***.

3. Let H be a cryptographically secure hash function.

4. Let there be a ***state transition function***:

$$\tau(\sigma_i, t) = (\tau_{\mathcal{X}}(\mathcal{X}_i, t), \tau_D(D_i, t)) \tag{3.1}$$

where:

[1] https://git-scm.com/about
[2] https://bitcoin.org/bitcoin.pdf

(a) $\tau_{\mathcal{X}}(\mathcal{X}_i, t) = \mathcal{X}_{i+1}$ where

$$\begin{aligned}\mathcal{X}_{i+1} &= \mathcal{X}_i \cup \{x_{i+1}\} \\ &= \{x_1, \ldots, x_i, x_{i+1}\}\end{aligned} \tag{3.2}$$

with

$$\begin{aligned}x_{i+1} &= \{h, t\} \\ h &= \{H(t), y\} \\ y &= \{H(x_j) | j < i\}\end{aligned} \tag{3.3}$$

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 transaction.

(b) $D_{\mathrm{i+1}} = \tau_{\mathrm{D}}(\sigma_i, t)$

5. 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.

6. Let $I(t)$ be a function that takes a transaction t, evaluates it using a function V, and if valid, uses τ to transform S. Call I the ***input*** or ***stimulus*** function.

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

8. 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***.

9. Let $E(i)$ 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 $V(t, v)$, and 2) the actual change of internal state S (as either $\mathcal{X}$ or D) through the state transition functions $\tau_{\mathcal{X}}$ and τ_D. Finally, we distinguish between 1) state transitions triggered by external events, stimuli, received through $I(t)$, and 2) a node's internal processing $P(x)$ that also results in calling V and τ 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 also 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. $\forall n, m \in N : \mathcal{X}_n \stackrel{!}{=} \mathcal{X}_m$ where $\stackrel{!}{=}$ means *is enforced*.

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

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

4. $P(i)$ is the *mining* process including the "proof-of-work" algorithm and composes with $V()$ and $\tau_{\mathcal{X}}$ when the hash is cracked.

5. $E(i)$ 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, as git's core intent is to allow different agents act autonomously and divergently on a shared code-base, which would be impossible if the states always had to match.

Through the lens of the formalism some other aspects of Ω_{git}can be understood as follows:

1. the validation function $V(e, v)$ by default only checks the structural validity of e as a commit object not it's content (though note that git does also support signing of commits which is also part of the validation)

2. the stimulus function $I(t)$ for Ω_{git} consists of the set of git commands available to the user

3. the state transition function $\tau_{\mathcal{X}}$ is the internal git function that adds a commit object and τ_{D} is the git function that adds code to the `index` triggered by `add`

4. E is, similarly to Ω_{bitcoin}, not formally defined for Ω_{git}.

We leave a more in depth application of the formalism to Ω_{git} as an exercise for the reader, however we underscore that the core difference between Ω_{bitcoin} and Ω_{git} lies in the former's constraint of $\forall n, m \in N : \mathcal{X}_n \stackrel{!}{=} \mathcal{X}_m$. One direct consequence of this for Ω_{bitcoin} is that 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} and in it lies the core of Bitcoin's scalability issues.

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 which consist entirely of multiple vantage points by definition.

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 "consensus," i.e., what single vantage point to 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 of which the 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 $\mathcal{X}_n$.

The term 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. In contrast, as a more agent-centric distributed system, git's *merge* command provides for a processes more recognizable as consensus, however it's not automated.

Perhaps a more accurate term for the hash-crack algorithm applied in Ω_{bitcoin} would be "proof-of-luck" and for the process itself simply sameness, not consensus. If you start from a data-centric viewpoint, 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 viewpoint 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 conceptual simplicity, it also immediately creates scalability issues, but worse, it makes it hard to take advantages inherent in the agent-centric approach.

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 they 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[3] provides the current premier example of generalized distributed computing using the Blockchain model. The Ethereum approach comes from an ontology of replicating the data certainty of single physical computer, on top of the stratum of a bunch of distributed nodes using the blockchain strategy of creating a single data reality in a cryptographic chain, but committing computations, instead of just monetary transactions as in bitcoin, into the blocks.

This approach does live up to the constraints listed above as described by Wood [EIP-150] where the bulk of that 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 inherited by Ethereum from the blockchain model, is immediately observable in its high compute cost and difficulty in scaling.

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 application consists of a network of agents maintaining a unique source chain of their transactions, paired with a shared space implemented as a validating, monotonic, sharded, distributed hash table (DHT) where every node enforces validation rules on that data in the DHT as well as providing 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 system properties which among other things declare the expected operating parameters of the application being specificed. For example the resilience factor as defined below is set as one such property.
4. Let ι_n be the second entry of all $\mathcal{X}_n$ and be a 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 $e_x \in \text{DNA}$. 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***.
    (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 current alpha version we use a modified version of [Kademlia] as implemented in [LibP2P].
12. Let DHT_{hc} augment DHT as follows:
    (a) $\forall \delta_{\text{key,value}} \in \Delta$ constrain $value$ to be of an entry type as defined in DNA. Furthmore, enforce that any function call $dht_x(y)$ which modifies Δ also uses $F_{\text{v}}(y)$ to validate y and records whether it is valid. Note that this validation phase may include contacting the source nodes involved in generating y to gather more information about the context of the transaction, see IV C 2.

[3] https://github.com/ethereum/wiki/wiki/White-Paper

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

(c) Include in F_{DHT} functions to add and retreive meta-data to elements that point to other elements with tags, thus creating a graph relations between the elments.

(d) 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 $A_n = H(pk_n)$).

(e) Let r be a parameter of DHT_{hc} to be set dependent on the characteristics deemed beneficial for maintaining multiple copies of entries in the DHT for the given application. Call r the ***resilience factor***.

(f) Allow that each node can maintain a set $M = \{m_n, \dots\}$ of metrics m_n about other nodes, where each m_n contains both a node's direct experience of n with respect to that metric, as well as the experience of other nodes of n. Enforce that one such metric kept is •uptime which keeps track of the percentage of time a node is experienced to be available. Call the process of nodes sharing these metrics ***gossip*** and refer to IV C 3 for details.

(g) Enforce that $\forall \delta \in \Delta_n$ each node n maintains a set $V_\delta = \{n_1, \dots, n_q\}$ of q closest nodes to δ as seen from n, which are *expected by* n to also hold δ. Resiliency is maintained by taking into account node uptimes and choosing the value of q so that:

$$\sum_{i=0}^{q} uptime(n_i) \geq r \tag{4.1}$$

whith $uptime(n) \in [0, 1]$.
Call the union of such sets V_δ, from a given node's perspective, the ***overlap list*** and also note that $q \geq r$.

(h) Allow every node n to discard every $\delta_x \in \Delta_n$ if the number of closer (with regards to $d(x, y)$) nodes is greater than q (i.e. if other nodes are able to construct their V_δ sets without including n, which in turn means there are enough other nodes responsible for holding δ in their Δ_m to have the system meet the resilience set by r even without n participating in storing δ). 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 according to node uptime.

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 $\{sys_{\text{commit}}, sys_{\text{get}}, \dots\}$ 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) Note: there may be additional system functions for a number of other purpose, such as content encryption, decryption, signature validation, etc that F_{app} may pragmatically want to rely on.

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 $P(i)$ be a set of functions in F_{app} to be registered in the system as callbacks based on various criteria, e.g. notification of rejected puts to the DHT, passage of time, etc.

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 what ammounts to pointers to machine state. 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 a

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 organizations 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 $0 \leq \Psi \leq 1$, 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

$$\begin{aligned} \forall \alpha \in R_A : \varepsilon(\alpha) = 1 \\ \forall \alpha \in R_C : \varepsilon(\alpha) < 1 \end{aligned} \tag{4.2}$$

   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 have 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, confidence is mainly dependent on an (human) agent's vantage point, set of data at hand, and maybe even intuition. Thus we find it more adequate to call it a soft criteria. In order to comprehend this concept objectively and relate it 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:

$$\Psi_\alpha \equiv \mathcal{P}(\alpha|\mathcal{C}) \tag{4.3}$$

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). The central aspect in this context we call $\alpha_{signature}$ which provides us with the ability to *know with certainty* that a given message's real author $Author_{real}$ is the same agent indicated solely via locally available data in the message's meta information through the cryptographic signature $Author_{local}$. We gain this confidence because we deem it *very hard* for any agent not in possession of the private key to create a valid signature for a given message.

$$\alpha_{signature} \equiv Author_{real} = Author_{local} \tag{4.4}$$

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 $\mathcal{C}$:

$$\Psi_{signature} = \mathcal{P}(Author_{real} = Author_{local}|\mathcal{C}) \tag{4.5}$$

If we constrain the context to remove the possibility 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[4], make clear.

We aim to describe these relationships in such detail in order to point out that any set R_A of *absolute requirements* can't reach beyond trivial statements - statements about the content and integrity of the local state of the agent itself. Following Descarte's way of questioning the confidence in every thought, we project his famous statement *cogito ergo sum* into the reference frame of multi-agent systems by stating: **Agents can only have honest confidence in the fact that they perceive a certain stimulus to be present and whether any particular abstract a priori model matches that stimulus without contradiction,** i.e., that an agent sees a certain piece of data and that it *is possible to interpret it in a certain way.* Every conclusion being drawn a posteriori through the application of sophisticated models of the context is dependent on assumptions about the context that are inherent to the model. This is the heart of the agent-centric outlook, and what we claim must always be taken into account

[4] "Most or all of the missing bitcoins were stolen straight out of the Mt. Gox hot wallet over time, beginning in late 2011" [Nilsson15]

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 instead frame the problem as:

> We wish to provide generalized 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.

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 lie at the heart of every decentralized application, and they vary across applications according to context. Every agent carefully keeps track of their representation of that portion of reality that is of importance to them - within the context of a given application that has to manage the trade-off between having high confidence thresholds $\varepsilon(\alpha)$ and a low need for resources and complexity.

For example, consider two different use cases of transactions:

1. receipt of an email message where we are trying to validate it as spam or not and
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 expend differing levels of resources to validate. We designed Holochain to allow such validation functions to be set contextually per application and expose these contexts explicitly. Thus, one could conceivably build a Holochain application that deliberately makes choices in its 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 specific instance at one end of this spectrum.

In the following sections we will show what categories of validation algorithms exist and how these can be stacked on top of each other in order to build decentralized systems that are able to maintain integrity without introducing an absolute truth every agent would be forced to accept or consider.

1. Intrinsic Data Integrity

Every application but the most low-level routines utilize non-trivial, structured data types. Structured implies the existence of a model describing how to interpret raw bits as an instance of a type and how pieces of the structure relate to each other. Often, this includes certain assumptions about the set of possible values. Certain value combinations might not be meaningful or violate the intrinsic integrity of this data type.

Consider the example of a cryptographically signed message $m = \{body, signature, author\}$, where $author$ is given in the form of their public key. This data type conveys the assumption that the three elements *body*, *signature* and *author* correspond to each other as constrained by the cryptographic algorithm that is assumed to be determined through the definition of this type. The intrinsic data integrity of a given instance can be validated just by looking at the data itself and checking the signature by applying the cryptographic algorithm that constitutes the central part of the type's a priori model. The validation yields a result $\in \{true, false\}$ which means that the confidence in the intrinsic data integrity is absolute, i.e. $\Psi_{intrinsic} = 1$.

Generally, **we define the intrinsic data integrity** of a transaction type ϕ as an aspect $\alpha_{\phi,intrinsic} \in R_A$, expressed through the existence of a deterministic and local validation function $V_\alpha(t)$ for transactions $t \in \phi$ that does not depend on any other inputs but t itself.

Note how the intrinsic data integrity of the message example above does not make any assumptions about any message's real author, as the aspect $\alpha_{signature}$ from the previous section does. With this definition, we focus on aspects that don't make any claims about system properties non-local to the agent under consideration, which roots the sequence of inferences that constitutes the validity and therefore confidence of a system's high-level aspects and integrity in consistent environmental inputs.

2. Membranes & Provenance

Distributed systems must rely on mechanisms to restrict participation by nodes in processes that without such restriction would compromise systemic integrity. Systems where the restrictions are based on the nodes' identity, whether that be as declared by type or authority, or collected from the history of the nodes' behaviors, are know as ***permissioned*** [Swanson15]. Systems where these restrictions are not based on properties of the nodes themselves are known as ***permissionless***. In permissionless multi-agent systems, a principle threat to systemic integrity comes from *Sybil-Attacks* [Douceur02], where an adversary tries to overcome the system's validation rules by spawning a large number of compromised nodes.

However, for both permissioned and permissionless

systems, mechanisms exists to gate participation. Formally:

Let $M(n, \phi, z)$ be a binary function that evaluates whether transactions of type ϕ submitted by $n \in N$ are to be accepted, and where z is any arbitrary extra information needed to make that evaluation. Call M the ***membrane*** function, and note that it will be a component of the validation function $V(t, v)$ from the initial formalism5.

In the case of Ω_{bitcoin} and $\Omega_{ethereum}$, M ignores the value of n and makes its determination solely on whether z demonstrates the "proof" in proof-of-X be it *work* or *stake* which is a sufficient gating to protect against Sybil-Attacks.

Giving up the data-centric fallacy of forcing one absolute truth $\forall n, m \in N : \mathcal{X}_n \stackrel{!}{=} \mathcal{X}_m$ reveals that we can't discard transaction provenance. Agent-centric distributed systems instead must rely on two central facts about data:

1. it originates from a source and
2. its historical sequence is local to that source.

For this reason, Ω_{hc} splits the system state data into two parts:

1. each node is responsible to maintain its own entire $\mathcal{X}_n$ or ***source chain*** and be ready to confirm that state to other nodes when asked and
2. all nodes are responsible to share portions of other nodes' transactions and those transactions' meta data in their **DHT shard** - meta data includes validity status, source, and optionally the source's chain headers which provide historical sequence.

Thus, the DHT provides distributed access to others' transactions and their evaluations of the validity of those transactions. This resembles how knowledge gets constructed within social fields and through interaction with others, as described by the sociological theory of *social constructivism*.

The properties of the DHT in conjunction with the hash function provide us with a deterministically defined set of nodes, i.e., a neighborhood for every transaction. One cannot easily construct a transaction such that it lands in a given neighborhood. Formally:

$$\begin{aligned} \forall t \in \Delta : \exists \eta : \mathcal{H} \to N^r \\ \eta(H(t)) = (n_1, n_2, \ldots, n_r) \end{aligned} \tag{4.6}$$

where the function η maps from the range $\mathcal{H}$ of the hash function H to the r nodes that keep the r redundant shards of the given transaction t (see 12h).

Having the list of nodes $\eta(H(t))$ allows an agent to compare third-party viewpoints regarding t, with its own and that of the transaction's source(s). The randomization of the hash function H ensures that those viewpoints represent an unbiased sample. r can be adjusted depending on the application's constraints and the chosen trade-off between costs and system integrity. These properties provide sufficient infrastructure to create system integrity by detecting nodes that don't play by the rules - like changing the history or content of their source chain. Additionally tooling appropriate for different contexts, including ones where detailed analysis of source chain history may be required, including:

1. Countersigning
2. Notaries (random agents on the network can be selected as such)
3. Published header examination
4. Source-chain examination.
5. Blocked-lists
6. etc..

Depending on the application's domain, neighborhoods could become vulnerable to Sybil-Attacks because a sufficiently large percentage of compromised nodes could introduce bias into the sample used by an agent to evaluate a given transaction. Holochain allows applications to handle Sybil-Attacks through domain specific membrane functions. Because we chose to inherently model agency within the system, permission can be granted or declined in a programmatic and decentralized manner thus allowing applications to appropriately land on the spectrum between permissioned and permissionless.

In appendix A, we provide some membrane schemes that can be chosen either for the outer membrane of that application that nodes have to cross in order to talk to any other node within the application or for any secondary membrane inside the application. That latter means that nodes could join permissionless and participate in aspects of the application that are not integrity critical without further condition but need to provide certain criteria in order to pass the membrane into application crucial validation.

Thus, Holochain applications maintain systemic integrity without introducing consensus and therefore (computationally expensive) absolute truth because 1) any single node uses provenance to independently verify any single transaction with the sources involved in that transaction and 2) because each Holochain application runs independently of all others, they are inherently permissioned by application specific rules for joining and continuing participation in that application's network. These both provide the benefit that any given Holochain application can tune the expense of that validation to a contextually appropriate level.

3. Gossip & World Model

So far, we have focused on those parts of the validation function V used to verify elements of $\mathcal{X}$. However, maintaining system integrity in distributed systems also requires that nodes have mechanisms sharing information about nodes that have broken the validation rules so that they can be excluded from participation. There exist, additionally, forms of bad-acting that do not live in the content of a transaction but in the patterns of transacting that are detrimental to the system, for example, denial of service attacks.

Holochain uses gossip for nodes to share information about their own experience of the behavior of other nodes. Informally we call this information the node's ***world model***. In this section we describe the nature of Holochain's gossip protocols and how they build and maintain a node's world model.

In 12e we described one such part of the world model, the *uptime* metric and how it is used for maintaining redundant copies of entries. In IV C 2 we defined a membrane function that determines if a node shall accept a transaction and allowed that function to take arbitrary data z. The main source of that data comes from this world model.

More formally:

1. Recall that each node maintains a set M of metrics m about other nodes it knows about. Note that in terms of our formalism, this world model is part of each node's non-chain state data D.

2. Let m be a tuple of tuples: $((\mu, c)_{\text{self}}, (\mu, c)_{\text{others}})$ which record an experience μ of a node with respect to a given metric and a confidence c of that experience, both as directly experienced or as "hearsay" received from other nodes.

3. Allow a class of entries stored in $\mathcal{X}_n$ be used also as a metric m_w which act as a signed declaration of the experience of n regarding some other node. Call such entries ***warrants***. These warrants allow us to use the standard tooling of Holochain to make provenance based, verifiable claims about other nodes in the network, which propagate orthogonally from the usual DHT methods, via gossip to nodes that need to "hear" about these claims so as to make decisions about interacting with nodes.

4. $\forall m \in M$ let the function $G_{\text{with}}(m)$ return a set of nodes important for a node to gossip **with** defined by a probabilistic weighting that information received from those nodes will result in changing m_{other}.

5. $\forall m \in M$ let the function $G_{\text{about}}(m)$ return a set of nodes important for a node to gossip **about** defined by the properties of m.

6. Define subsets of $G_{\text{with}}(m)$ according to a correlation with what it means to have low vs. high confidence value c:

   (a) **Pull**: consisting of nodes about which a low confidence means a need for more frequent gossip to raise a node's confidence. Such nodes would include those for which, with respect to the given node, hold its published entries, hold entries it is also responsible for holding, are close the then node (i.e. in its lowest k-bucket), and which it relies on for routing (i.e. a subset of each k-bucket)

   (b) **Push**: consisting of nodes about which a high confidence implies a need for more frequent gossip to spread the information about that node. Such nodes would include ones for which a given node has high confidence is a bad actor, i.e. it has directly experienced bad acting, or has received bad actor gossip from nodes that it has high confidence in being able to make that bad actor evaluation.

The computational costs of gossip depend on the set of metrics that a particular application needs to keep track of to maintain system integrity. For an application with a very strong membership membrane perhaps only *uptime* metrics are necessary to gossip about to balance resilience. But this too may depend on a priori knowledge of the nodes involved in the application. Applications with very loose membership membranes may have a substantial number of metrics and complex membrane functions using those metrics which may require substantial compute effort. The Holochain design intentionally leaves these parameters only loosely specified so that applications can be built fit for purpose.

V. COMPLEXITY IN DISTRIBUTED SYSTEMS

In this section we discuss the complexity of our proposed architecture for decentralized 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*[5]) 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

$$c + n \tag{5.1}$$

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

$$\Omega_{BitcoinNode} \in O(n^2) \tag{5.2}$$

The complexity handled by one Bitcoin node does not [6] 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

$$\Omega_{Bitcoin} \in O(n^2 m) \tag{5.3}$$

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 that has its execution being triggered by the given transaction plus a constant:

$$c + f_{tx_i}(n, m) \tag{5.4}$$

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

$$c + \sum_{i=0}^{n} f_{tx_i}(n, m) \tag{5.5}$$

that is

$$\Omega_{EthereumNode} \in O(n \cdot f_{avg}(n, m)) \tag{5.6}$$

whereas users are incentivized to hold the average complexity $f_{avg}(n, m)$ of the code being run by Ethereum small since execution has to be payed for in gas and which is due to restrictions such as the *block gas limit*. In other words, because of the complexity $\sum_{i=0}^{n} f_{tx_i}(n, m)$ 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 network's limits.

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:

$$\Omega_{Ethereum} \in O(nm \cdot f_{tx_i}(n, m)) \tag{5.7}$$

C. Blockchain

Both examples of Blockchain systems above do need a non-trivial computational overhead in order 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-

[5] 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

[6] 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}, and 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 rationally only, the confidence in t being added to all $\mathcal{X}_n$ has an upper bound in

$$\Psi_{Blockchain}(t) < min\left(1, \frac{cost(x_{HR})}{value(t)}\right) \tag{5.8}$$

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, let n be the sum of all public[7] (i.e., *put* to the DHT) state transitions (*transactions*), let 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

$$c + \lceil log(m) \rceil. \tag{5.9}$$

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

$$\underbrace{c + \lceil log(m) \rceil}_{DHTlookup} + q + r \cdot \underbrace{v(n,m)}_{validation} \tag{5.10}$$

which implies the following whole system complexity in O-notation

$$\Omega_{Holochain} \in O(n \cdot (log(m) + v(n,m)) \tag{5.11}$$

[7] 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

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

$$\Omega_{HolochainNode} \in O\left(\frac{n}{m} \cdot (log(m) + v(n,m))\right) \tag{5.12}$$

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.*

VI. 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`.

Additionally we have created a benchmarking suite to examine the processing, bandwidth and storage used in various scenarios, and compared these with Ethereum applications in similar scenarios. These can be seen here: `https://github.com/holochain/benchmarks`

We have yet to implement scalability tests for large scale applications, but it is in our roadmap.

Appendix A: Membranes

- *Invitation*
  One of the most natural approaches for membrane crossing in a space in which agents provide identity is to rely on invitation by agents that are already in the membrane. This could be invitation:
  - by anyone
  - by an admin (that could either be set in the application's DNA or a variable shared within the DHT - both could be mutable or constant)
  - by multiple users (applying social triangulation)

- *Proof-of-Identity / Reputation*
  Given the presence of other applications/chains, these can be used to attach the identity and its reputation in that chain to the agent that wants to join. Since this seems to be a crucial pillar of the ecosystem of Holochain applications, we plan to deliver a system-level application called DPKI (distributed public key infrastructure) that will function as the main identity and reputation platform. A prototype of this app was already developed prior to the writing of this paper.

- *Proof-of-Presence*
  Use of notarized national documents/passports/identity cards within the agent entry (second entry in $\mathcal{X}$).

- *Proof-of-Service*
  Cryptographic proof of delivery of a service / hosting of an application. We intend to leverage this technique with our distributed cloud hosting application **Holo**, which we will build on top of Holochain. See our Holo Hosting white paper for more detail [HoloHosting].
- *Proof-of-Work*
  If the application's requirement is not anonymity, other than the cryptographic hash-cracking work applied in most of the Blockchains, this could also be useful work that new members are asked to contribute to the community or a puzzle to proof domain knowledge. Examples are:
  - Test for knowledge about local maps to proof citizenship
  - DNA sequencing
  - Protein folding
  - SETI
  - Publication of scientific article

- *Proof-of-Stake / Payment*
  Deposit or payment to have agent certified.

- *Immune System*
  Blacklisting of nodes that don't play by the application rules.

ACKNOWLEDGMENTS

We thank Jonathan Paprocki for his care in editing and supporting the mathematical formalizations, and Steve Sawin for his review of this paper, LaTeX 2εsupport, and moral support.

[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 Gün 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 WORLD'S CRYPTOCURRENCY MINING USES MORE ELECTRICITY THAN ICELAND* https://www.digitaltrends.com/computing/bitcoin-ethereum-mining-use-significant-electrical-power/ a

[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

[LibP2P] Juan Benet, David Dias *libp2p Specification* https://github.com/libp2p/specs

[Oxford] Oxford Online dictionary https://en.oxforddictionaries.com/definition/provenance

[Douceur02] Douceur, John R. (2002). "The Sybil Attack" https://www.microsoft.com/en-us/research/publication/the-sybil-attack/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F74220%2Fiptps2002.pdf International workshop on Peer-To-Peer Systems. Retrieved 23 April 2016.

[HoloCurrency] Arthur Brock and Eric Harris-Braun 2017 *Holo: Cryptocurrency Infrastructure for Global Scale and Stable Value* https://holo.host/holo-currency-wp/

[Nilsson15] Nilsson, Kim (19 April 2015). *The missing MtGox bitcoins". Retrieved 10 December 2015.* http://blog.wizsec.jp/2015/04/the-missing-mtgox-bitcoins.html

[Swanson15] Tim Swanson *Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems* April 6, 2015 https://pdfs.semanticscholar.org/f3a2/2daa64fc82fcda47e86ac50d555ffc24b8c7.pdf

[HoloHosting] Arthur Brock et al, 2017 *Holo Green Paper* https://files.holo.host/2018/03/Holo-Green-Paper.pdf