STATUS DIGEST # 1: ARCHITECTURE

BlockScience

August 24, 2020

OVERVIEW

This foundational documentation digest covers the contextual hierarchy of the STATUS network as a cryptoeconomic system, providing a high level architectural description of four network components: the human/machine socioeconomic actors (or *entities*), the device *(app)lications* that entities use as interfaces, the peer-to-peer network of hardware/software *clients*, and the *smart contract* automation infrastructure.

Contents

1 Architecture Overview

The STATUS network architecture is comprised of four interconnected components. These components are a way of both 1) understanding the network from a goal-driven point of view, in which component characteristics are connected to the overall goals of the network, and 2) breaking down the network from a functional point of view, indicating how different members of each component interact with each other and with other components. The former is most important for understanding *why* the network is constructed the way it is, while the latter is required for understanding *how* the network operates, and consequently provides guidance for its eventual implementation.

There are several different methodologies that currently exist which can be leveraged to undertake this distillation of a complex, interconnected network such as the STATUS network into its four components. One of the most popular methodologies is **network science** (see e.g. Hexmoor 2015), in which a distinction is made between *type* and *relation* in the use of **Nodes** and **Edges**, respectively. Another methodology, related but developed independently, relies upon function to break a complex system into *subsystems*. This approach is undertaken largely (but not exclusively) under the auspices of **system engineering** (see e.g. Wasson 2015; Liu 2015). In what follows, both network science and system engineering will be applied, to place the network architecture into the context that is defined by the requirements of the STATUS network. This prevents the architectural structure from arising due to an arbitrary classification—instead, it is a natural consequence of the desired *system goals*. By contextualizing the breakdown into components, and classifying the resulting component characteristics, a construction is laid down that is followed in the succeeding Documentation Digests. These digests will address the behavioral assumptions that underlie each component, outline the methodology for a computational abstraction of the network (**cadCAD**), and provide the scaffolding for the use of this abstraction as a decision support system (DSS) alongside the real-time operation of the STATUS network.

This digest is organized as follows. In Section 2 an overview of the architectural components is provided, while in Section 3 the network model supporting and further characterizing the components is described. Section 4 then presents the system goals and associated key performance indicators of the STATUS network, and connects them to the design of the two most important components (the **Client** and the **Contract**), while at the same time setting the stage for the more detailed description of the cryptoeconomic incentive system to be provided in the second digest.

2 Architecture Components

2.1 Entities: Users of the Network

At its heart the STATUS network is designed to facilitate the transfer of information between final users. These users form a natural category in their own right, as they are:

- the *raison d'être* for the network, in the sense that the fulfillment of network objectives is predicated upon its use by users;

- the source of *monetizing* the network, as their desire for information is represented by their willingness to pay for network services; and
- the *initiators* and final *recipients* of information flows, as they both create information to disseminate and consume information at one of possibly many transfer endpoints.

In both network science and system engineering, such users are denoted as *actors* or (as in what follows) ***Entities***. An **Entity** is endowed with the following characteristics:

1. a unique *identity*, allowing the association of both information and value flows to an **Entity** to be made;
2. a set of *interfaces*, known as ***Apps***, allowing access to the STATUS network; and
3. a set of *behaviors*, describing how an **Entity** acts and reacts within the context of network activities.

Each of these characteristics can be modeled from the perspective of network science, allocating an **Entity** the role of participant in a network, with its behavior(s) and interface(s) dictating how they form relations between and among other **Entities**, and with other network participants. This formalization will be described in Section 3, with the exception of an **Entity**'s behaviors, which will be discussed briefly in Section 4 within the context of system goals (and further articulated in the second Documentation Digest, hereafter "Documentation Digest #2").

For this component and the succeeding components outlined in Sections 2.2 - 2.4, an accompanying figure provides a visual breakdown of the overall characteristics as described above: for example, the visual breakdown for the **Entity** component is given in Figure 1, p. 5. The figures provide a sub-component breakdown, indicating which characteristics are required *a priori* (such as an identity), and which are optional in the sense that they may be added (or removed) as the network evolves (such as various behaviors and/or actions). In addition, in the interest of brevity the possible *relationships* between the displayed component and other components is also shown—these relationships are described more fully in Section 3, and so these figures may be referenced again when that section is reached.

2.2 Apps: Gateways to the Network

While **Entities** are the final users of the network, and the source of its value, they cannot interact with the computational network without one or more interfaces. Such interfaces are (as of this writing) usually classified as *mobile* or *laptop/desktop* computer programs that engage with the network using the Internet as the low-level communications layer. Such programs, or ***Apps***, possess the following characteristics that allow an **Entity** to engage with other **Entities** in information and value exchange:

1. a unique *identity*, allowing the association of an **App** to an **Entity**;
2. *access* to an on-chain cryptographic **Account**, associated with one or more **Entities**, that permits value to flow in response to information flows; and
3. *access* to a communications and computation network, which implements the actual information flows and may also provide accounting services for associated value flows (see ***Clients***, below).

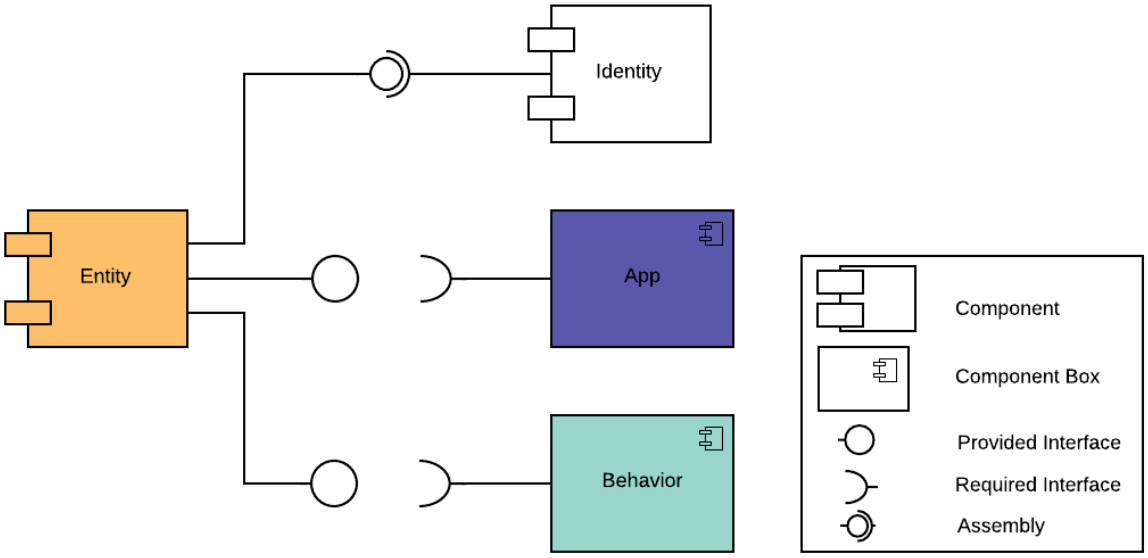

Figure 1: **Entity** Component Characteristic and Relationship Breakdown

Apps rely heavily in their implementation upon the hardware specifications of the device upon which they are built, and hence their explicit characterization will be omitted in what follows (apart from a brief discussion concerning the goals of the system in Section 4). The visual breakdown of the **App** component is displayed in Figure 2.

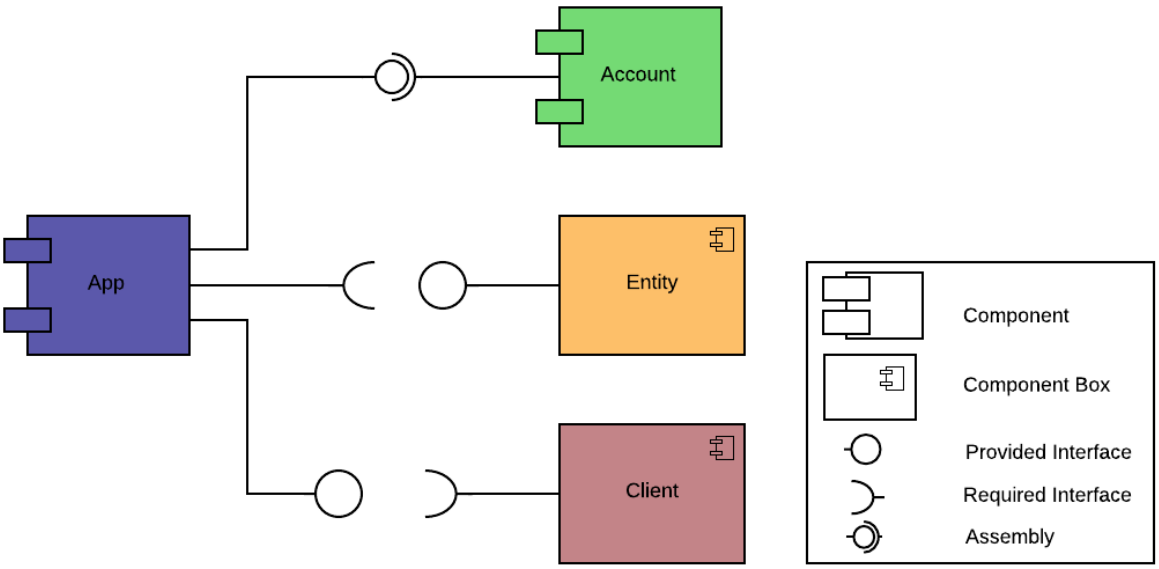

Figure 2: **App** Component Characteristic and Relationship Breakdown

2.3 Clients: Processors of the Network

When an **Entity** uses an **App** to initiate or receive an information flow, it utilizes the **App**'s access to the p2p network that handles information. Access is granted to a computational information processing subsystem, called a ***Client***, that facilitates the flow of information and may handle low-level value flow accounting. A **Client** is the most primitive decision-making unit in the STATUS network, and has the following characteristics:

1. a unique *identity*, allowing the association of one or more **Clients** to one or more **Apps**;
2. a set of *actions*, allowing information flows to be routed according to the requirements specified by an **App**'s **Entity**; and
3. a set of *behaviors*, specifying which action(s) to take in response to local and global network conditions.

Given its primal status in the p2p network, a **Client** is also an intuitively appealing class of network participant, and is assumed to interact only with other **Clients** and one or more **Apps**. Thus, there is a natural hierarchy between **Entities**, **Apps** and **Clients** that defines how information processing requests originate, propagate, and are processed 'downstream', before arriving at a **Client** where information is stored, value is transferred, and notification is propagated 'upstream'. A brief overview of **Client** incentives (and potential behaviors) is outlined in Section 4.3, while **Client** actions are discussed in greater detail in Section 4.4. In addition, the 'primacy' of the **Client** in the peer-to-peer (hereafter, p2p) network allows it to be more thoroughly described as part of the network model given in Section 3.

The visual breakdown of the **Client** component is given in Figure 3.

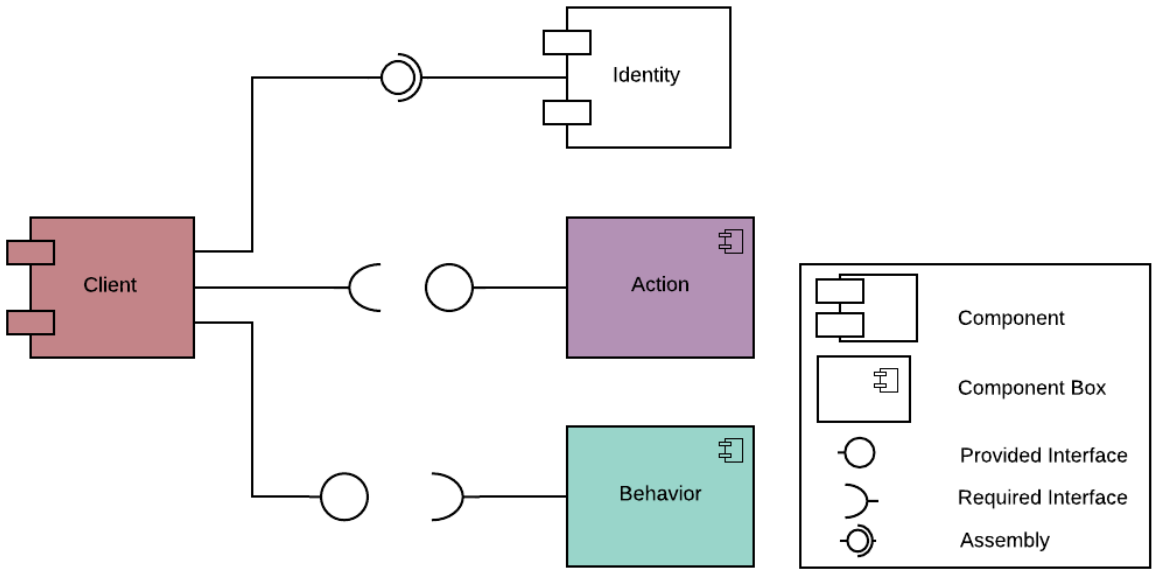

Figure 3: **Client** Component Characteristic and Relationship Breakdown

2.4 Smart Contracts: Trust in the Network

Information is communicated through the network in response to value flows between **Entities** (facilitated by **Apps** and their connection to **Clients**) and between **Clients**, facilitated by their actions and behaviors. Although in general no physical implementation of a p2p information and payments network is 100% error-free, it is critical to distinguish between deliberate activity that subverts the system's goals from events that takes place randomly (such as network outages or hardware failures). Deliberate subversion of system goals may take place to expropriate value flows (e.g. acquire the network's token costlessly), to disrupt the network for idiosyncratic benefit (e.g. "for the lulz"), or both. When the set of deliberately subversive actions have been identified during the design process, the system as a whole must be *engineered*, to:

- reduce the *set* of feasibly damaging activities to be as small as possible (*securitization*); and
- reduce the *payoff* of feasibly damaging activities to be as small as possible (*incentivization*).

Much is known regarding the implementation of cryptography to help secure a network and prevent certain activities from being undertaken. Moreover, cryptography also introduces the side-effect that claimed sources of value are demonstrably provable, allowing participants to *condition* upon the existence of value sources in their decision-making. This **trustworthiness** is further facilitated by *automating* such decision-making, providing a set of behaviors that can themselves be provably demonstrated to exist within a distributed ledger technology (DLT) such as blockchain. By facilitating trust in both value sources and in their manipulation via automated decision-making, a DLT-based value infrastructure can be developed that provides the proper, "cryptoeconomic", incentives for network participants to refrain from feasibly damaging activities, because they are unprofitable.

The apotheosis of such automation in a DLT infrastructure is the *(Smart) **Contract***, which is computer code that is provably demonstrated to exist at a DLT's database address. The **Contract** processes value flows in response to (local or global) network conditions, and has the following characteristics:

1. a unique *identity*, allowing **Apps** and/or **Clients** to interact with (and hence respond to) the **Contract**;
2. a set of *actions*, allowing value transfers to and from DLT addresses; and
3. a set of *behaviors*, specifying which action(s) to take in response to local and global network conditions;

In addition, the **Contract** has a *public visibility*: its identity, actions and behaviors are publicly viewable and auditable from the associated DLT address. Finally, the **Contract** responds to *governance*: while **Entities**, **Apps** and **Clients** are free to enter or exit the network at will, the **Contract** may only change its internal characteristics in response to the STATUS network's governance implementation.

As the heart of the **Contract** is its role in incentivizing behavior of network participants, it will be discussed more fully in Documentation Digest #2—a brief overview of its incentivization role is presented in Section 4.5. The visual breakdown of the **Contract** component is displayed in Figure 4, p. 8.

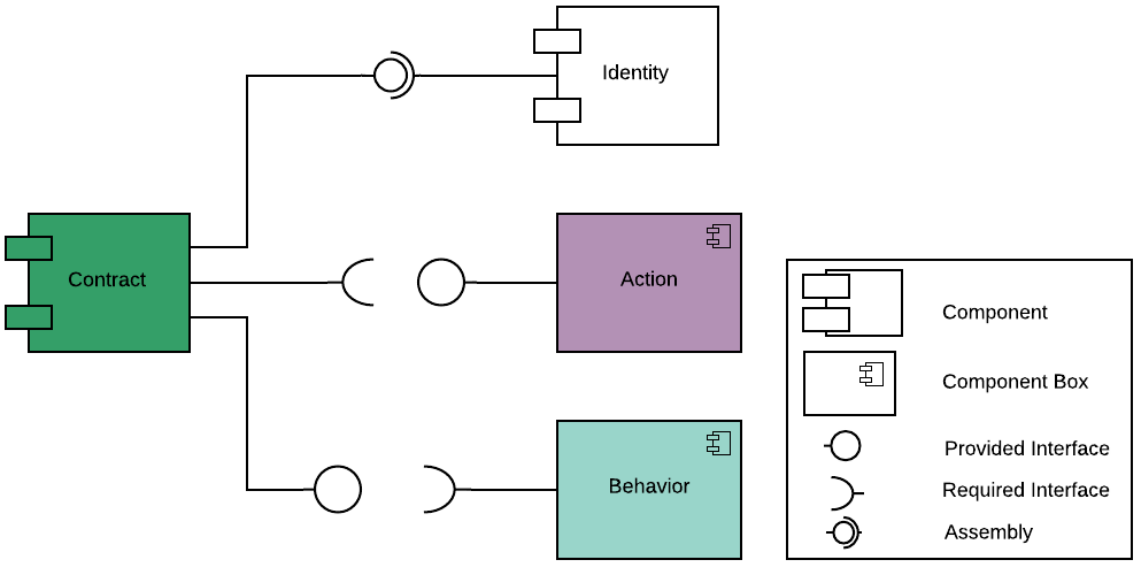

Figure 4: **Contract** Component Characteristic and Relationship Breakdown

3 The Network Model

The four components of the STATUS network introduced in the previous section correspond to distinct network participants, that take actions in response to conditions using behaviors. As such, a participant may be considered a *vertex* in a formal network model. Corresponding to these vertices are multiple types of *edges* between vertices. Edges define the types of relationships that vertices may have with one another. Together, these multiple types of vertices and edges create a *multigraph* network, where vertices may have multiple relationships (edges) between them. For example, an **App** may have both an information transfer relationship with a **Client**, when transmitting a **Message** for storage, and a value transfer relationship, indicating which **Account** should be debited to fund the **Message** transmission. **Clients** may have many such relationships between them, as well as multiple relationships with the **Contract** component (itself perhaps distributed among many sub-components).

For clarity in what follows we will sometimes focus upon one or another (set of) types for both vertices and edges. For example, when considering message passing between vertices, the edges are the communications pathways for the transmission of e.g. file requests, and the primary concern is the **routing** of messages between vertices. Moreover, since messages may contain files, the transmission of (and proof of storage of) files is accomplished along both communication pathway edges *and* edges between vertices that account for the balance of tokens that are promised—and, upon a successful storage of a file, delivered—from one vertex to another. Both of these aspects of the network (communication flows and financial flows) are discussed in further detail below.

3.1 The Network as a Cryptoeconomic System

The various aspects of the network are first delineated by vertex type, according to their function. The resulting network landscape (or 'topology') is a typical example of a *cryptoeconomic* network, in which value and (usually) information flows are secured by cryptography. The network is also *multilayered* (or multigraph), in the sense that multiple types of vertices can connect to each other via multiple types of edges.

Cryptoeconomic networks are generally sensitive to:

1. the **scale** of the network, in that the number of relationships (edges) between vertices quickly dwarfs the the number of vertices, and
2. the **complexity** of the network, in that the complicated interactions between network vertices (or between sub-networks), or the network and the outside world (its environment), can result in unexpected *emergent* properties.

In practice, the combination of scale effects and complex emergent phenomena prevent a simple closed-form analysis of the resulting network, as might be the case in a network with (say) one type of vertex and one type of edge. Instead, a classification (or *taxonomy*) of the participants in the network, as well as their relationships, provides the breakdown of the complex network into its respective components. Section 3.2 describes the classification of network participants, while Section 3.3 provides a high-level overview of the relationships between them (for the **Client** p2p network this will be further discussed in Sections 4.3 and 4.4). A brief description of the p2p local network and how **Clients** interact with each other is then given in Section 3.4.

3.2 The Vertices: Network Participants

The **Entity**, **App** and **Client** network participants are formalized as the vertices of an underlying cryptoeconomic system, with two general vertex types: **Entity**, and (for **Apps** and **Clients**) **Node**. In addition, there is a special vertex type, **Account**, that acts as the focal point for value transfers, and as the location for algorithmic incentivization via the Smart **Contract**. Although **Accounts** could also be subsumed into the architectural structure of **Apps** or even **Entities** (as it may be a requirement that an **Entity** control one or more **Accounts** as a prerequisite for network participation), in practice considering **Accounts** as a vertex type allows incentivization via the **Contract** component to be better understood (as described further in Section 4.5). The **Contract** component can then be viewed either as a type of **Account**, when discussing the algorithmic Smart Contract itself, or as a 'container' for **Accounts**, when discussing how incentive structures help regulate value transfers.

Vertex 1: Entity

An **Entity** vertex represents the unique identity of an off-chain person or organization. Its attributes include an *action set* A_E, indicating how an **Entity** can interact with the rest of the network, and a *behavior* $\mathcal{B}_E$, which is a set of mappings connecting what an **Entity** might know with its action.

Vertex 2: Account

An **Account** is a cryptographically-supported on-chain address, that is controllable using a private key in an encryption mechanism. It is generally assumed that the private key is controlled by one or (in the case of multi-signature mechanisms) more **Entities**. The **Contract** component, being a Smart Contract, is this type of vertex. The set of all **Account** vertices will be denoted A in what follows.

Vertex 3: Node

A **Node** is a hardware or software member of a p2p network that executes lower-level exchanges of information and facilitates transfers of value. Both **Apps** and **Clients** are considered **Nodes**. **Nodes** are the main non-**Contract** computation and problem-solving vertex type in the network, and both message routing and file storage take place at the **Node** level. The set of **Nodes** will be denoted V.

3.3 The Edges: Network Relationships

Entities, **Accounts** and **Nodes** interact via *relationships*, which are formalized as the edges in this multigraph network. There are six different types of edges, corresponding to the relationships these vertices have with each other in the network. Each of these edges is a *directed* edge, in a 'From -> To' relationship.

Edge 1: Entity -> Entity

An **Entity** has a relationship (off-chain) with another **Entity**. The **Entity**–**Entity** relationship is part of the off-chain *socioeconomic network*. It is the environment representing people and organizations that are connected to each other outside of the STATUS network.

Edge 2: Entity -> Account

An **Entity** may control the private cryptographic key(s) of an **Account**. The **Entity**–**Account** relationship is part of the *financial network*. It is the network where transfers of tokens are performed by **Entities** controlling those tokens. (Note that under this interpretation, STATUS may itself be viewed as an **Entity** that controls the **Contract** component as an **Account**.)

Edge 3: Entity -> Node

An **Entity** interfaces with one or more **Apps**, which are a type of **Node**. The **Entity**–**Node** relationship ultimately dictates who *controls* the **Nodes** (both **Apps** and **Clients**) that participate in the STATUS network as part of the *computation and communication network*.

Edge 4: Account -> Account

An **Account** transfers funds to another **Account**. The **Account**–**Account** relationship is the means by which funds are transferred within the financial network.

Edge 5: Node -> Account

A **Node** (usually as an **App**) transfers value to and receives value from an **Account**. The **Node**–**Account** relationship is part of the financial network. A **Node** can receive incentive rewards (resp. penalties) in an **Account** following actions that do (resp. do not) support system goals.

Edge 6: Node -> Node

A **Node** is called a *peer* of another **Node** if there exists a directed edge from the first **Node** to the second **Node**. Note that as **Apps** are assumed to interact only indirectly with each other via the computation and communication network mediated through **Clients**, the term peer generally applies only when one **Client** has a directed edge with another **Client**. A **Node**–**Node** relationship is the underlying fundamental 'unit' for the computation and communication network.

The prototypical **network schematic** representing the above network characterization is given in Figure 5 (p. 12). This figure provides the breakdown of network participants by component, and also introduces generic value and information flows as arrows between participants. For the purposes of this digest, the figure is meant to be illustrative only—in further digests this figure will be refined to include more detail about relationships (edges), showing how the incentive structure's value flows support the required information flows as **Messages** and **Files** pass between participants.

3.4 Local Network Structure

The designer of the network—hereafter the *systems designer*—generally examines global, network-wide information to optimize the structure of the network as a whole. By contrast, the individual **Nodes** in a computation and communication network are aware only of their immediate surroundings, *as specified by their set of relationships*, or edges. (Recall from Edge 6 above that if one **Node** has a directed edge from it to another Node, then that other **Node** is called a peer.)

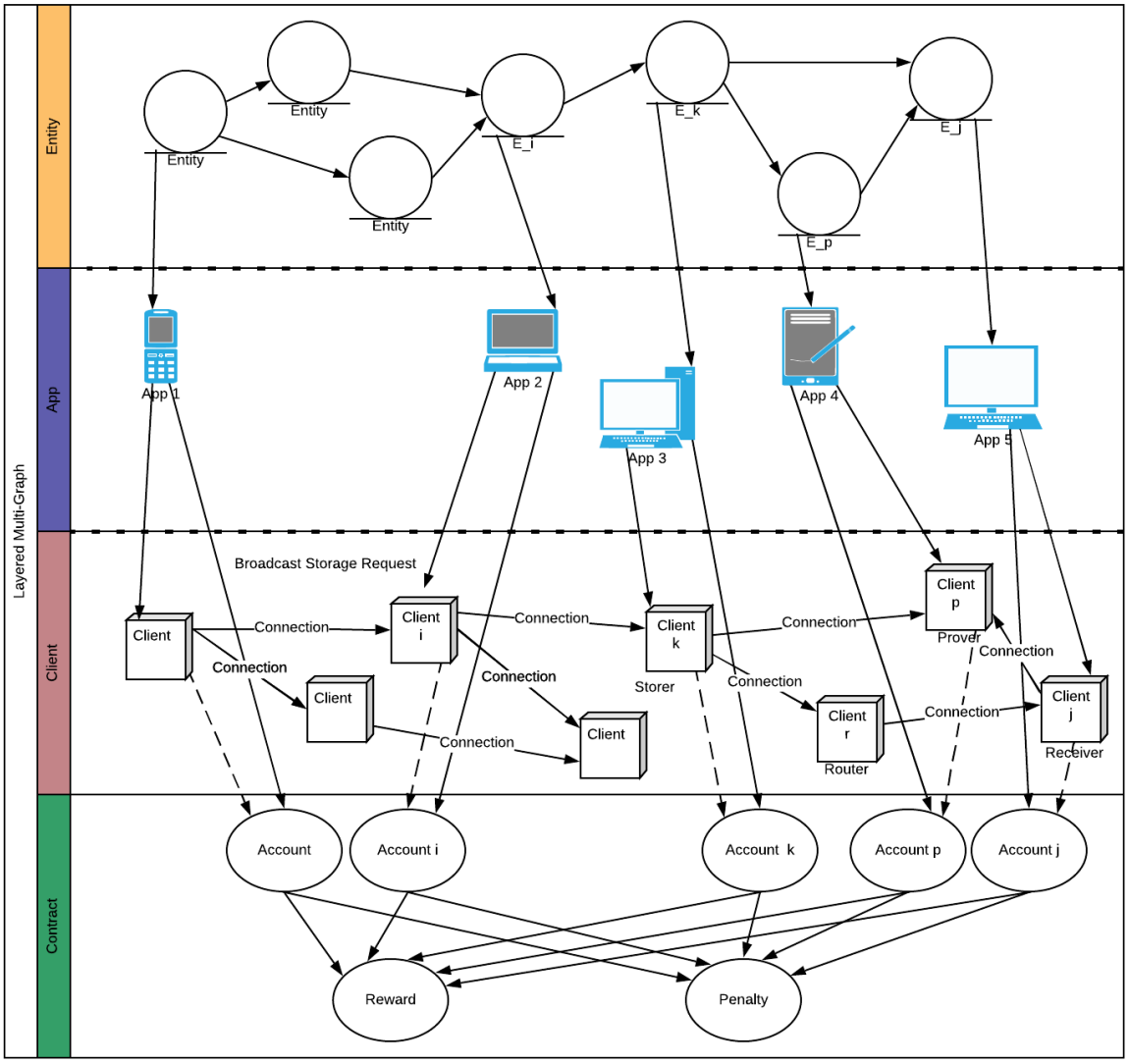

Figure 5: Network Schematic By Component (Generic Edges)

Node Characteristic: Neighborhood

The set of all peers for a Node at time t is a Node's *neighborhood* N_t, and defines the awareness of a Node of its surroundings.

If a **Node** is a **Client** then its neighborhood can include both other **Clients** and also one or more **Apps**, acting as the gateways for **Entities** to interact with the network. In addition, a **Client** can also interact with the **Contract** component (and vice-versa). The neighborhood of an **App** may contain one or more **Clients**.[1]

As defined, a **Client** may be a peer of another **Client** without a reciprocal relationship—but a discovery-based p2p network will usually allow the peer relationship to be reciprocated. For example, if a **Client** is attempting to expand its knowledge of the network's structure by 1) broadcasting a 'ping' to the entire network and then 2) adding ping respondents to its neighborhood, it is assumed that the arrival of such a ping will allow, in turn, the recipient to add the originator to *its* neighborhood. This means that the sub-network comprised of **Clients** and their peer network edges (cf. Edge 6, p. 11) is an *undirected* network when the edges define a knowledge relationship based upon identity: if one **Client** "knows about" another **Client**, then that knowledge is reciprocal.

Along the same lines, the sub-networks of **Entities** and **Apps**, or of **Apps** and **Clients**, also contain undirected edges, when an edge embodies the "knows about" relationship. This allows, for example, an **Entity** to instruct an **App** to send a **Message** via the **Client** network, and to pass on **Messages** returned by **Clients** via its **App** to the **Entity**. By contrast, the same networks may contain *directed* edges, when an edge embodies a "gives permission" relationship. An **Entity** may give permission to an **App** to initiate a value transfer from an **Account** on behalf of an **Entity**, but a **Client** will not be able to initiate such a transfer by giving permission to an **App**, as a **Client** does not have the "gives permission" relationship with an **App**.

The network science approach presented in this section allows the STATUS network to be disentangled into components that interact via communication, computation and value transfer pathways. To provide further structure requires a set of goals that the network is designed to support. Section 4 discusses how system goals influence architecture design, and discusses the resulting design decisions for (in particular) the **Client** and **Contract** components.

4 System Goals and Architecture Design

The *system goals* of the STATUS network specify what the network is to accomplish. They include abstract desirables such as "the network should be available" and "a message sent should be received" to specific requirements, such as "the network should have an uptime of 99.95%" or "a message sent should be routed to its recipient in the shortest hop count possible". As system goals become more precisely articulated, the system architecture that supports them becomes more and more clearly defined. This is a crucial step in the system engineering process, in which *a priori* system design decisions are reduced to the extent possible, and design proceeds only after system goals, and their associated *requirements*, have been laid out.

[1] Note that although an **App** interfaces with an **Entity**, it is not a part of the **App**'s p2p neighborhood.

4.1 System Goals

For this project the system goals have been articulated within the context of the *cryptoeconomics* of the network. This context involves how network participants are incentivized with a reward/penalty system to take actions that **support** at least ***one*** system goal, and directly **contradict** (or circumvent) ***no*** system goals. As such, the resulting architecture cannot serve as an explicit technical implementation of the STATUS network, but rather as an abstraction that indicates whether or not the resulting flows of information and value allow the cryptoeconomic system goals to be sustained. By carefully specifying these system goals from within the cryptoeconomic context, it is then possible to see why an architectural design is ultimately selected.

The STATUS network architecture is designed to support the following **system goals**, in which it is taken as given that an **Entity** wishes to send a **Message** to another **Entity**:

1. A **Client** should prefer to route a **Message** to its final recipient as quickly and cheaply as possible;
2. A **Client** should prefer to store a **Message**, and allow a **Message** to be retrieved, if it is possible to do so;
3. Opportunities to "game the system", by manipulating **Client** decentralized incentives for idiosyncratic benefit, should be too expensive to exploit;
4. Value transfer (e.g. compensation for **Message** routing, and reward/penalty incentive payments) should be trustworthy, and explicit enforcement of incentives should be undertaken as infrequently as possible.

4.2 Key Performance Indicators

As a preliminary step it is clear that two of the four architectural components described above, the **Entity** component and the **App** component, are foundational in the sense that *any* system goal presupposes their existence. These foundational roles are not a limitation on how **Entities** and **Apps** are implemented—for example, **Entities** may be human actors or other computing devices, while **Apps** may be e.g. computer desktop machines or mobile devices—but they do require that the *performance* of the network be measured according to how these roles are supported by the system goals. Since the reason for routing a **Message** between **Clients** is, in the end, so that one **Entity** may **Message** another **Entity**, their satisfaction makes up one of the overarching Key Performance Indicators (KPIs) that the architectural design must assist (rather than hinder).

KPIs are a "refinement" of the overall system goals, in that they can be stated in such as way as to provide quantitative feedback regarding the success or failure of supporting one or more system goals. The most straightforward method of introducing a *quantifiable* measurement of trustworthy communication is to specify what it is that **Entities** will require as a minimum quality level, below which they will find alternative communication channels preferable. In the case of **File** communication, perhaps the most important specification is that **Files** are to be available with the smallest delay possible. Thus we can specify two **performance metrics** that can be used to assess whether or not the overarching goal is being met by a particular architecture:

KPI 1: File Persistence

The (global) File persistence measure is the fraction of all Files in the system which are still available at a particular point in time.

Clients do not have a complete view of the entire network—thus, the performance metric of File persistence is a global measure, which might in principle be difficult to estimate in a decentralized environment. By virtue of the following performance metric, File persistence becomes *automatically* satisfied: in an optimally-tuned system, there are no lost Files.

KPI 2: Expected Waiting Time

The (local) expected waiting time is a Client's **best guess** of how long it must wait until their request for a File is fulfilled.

When a single Client is computing its expected waiting time, then the time is *local* to that Client and expresses a *statistic*, covering what that Client knows about its surroundings (its neighborhood).[2] By expressing KPI 2 in terms of an *objective*, e.g. "**minimize** the expected waiting time across all File requests", the performance measure can be used to achieve system goals 1 and 2 (p. 14). System goal 1 is fulfilled directly from such a minimization criterion; moreover, system goal 2 is fulfilled indirectly, as minimization of the expected waiting time fulfills KPI 1 at the same time, because an unavailable File carries with it an infinite waiting time for a File request.

Thus, the architecture is designed to minimize the expected waiting time across all File requests, and we may characterize this as a network *requirement*. A second requirement is to reduce the **resource cost** of storing and transferring a File from its source to its requester. This resource cost can include e.g. permanent media storage, memory cache storage, up/download transmission costs, etc. It is naturally difficult (or impossible) to envision a network where Clients always know the individual resource attributes of other Clients they could potentially forward File requests (or Files). Instead, a (noisy) proxy for the resource cost is the number of Clients in a route, under the assumption that a longer route is (through transmission costs if nothing else) more costly. At the local level, a Client will try to estimate how long a route is required to fulfill a File request:

KPI 3: Expected Route Distance

The (local) expected route distance is a Client's **best guess** of how many Clients make up the route to serve its File request.

By introducing the objective "**minimize** the expected route distance to serve a File request", KPI 3 acts to support system goals 1 and 3: system goal 1 is supported by virtue of an efficient route minimizing resource costs (and

[2] This can be aggregated across Clients to create a *global* expected waiting time, but in practice it is the local information that will determine the success of a File transfer.

potentially transmission time), while system goal 3 is supported by ensuring that resource costs are not inflated by introducing e.g. vacuous self-serving routes from fictitious **File** requests. It should also be noted that because system goal 3 relies upon an incentive structure that supports system goal 4, KPI 3 heavily influences **Contract** design as well. Coupled with the minimum expected waiting time requirement, then, the minimum expected route distance requirement serves to define the **Client** and **Contract** architecture necessary to support all system goals (1 - 4).

Note that while KPIs are vital for the design of the **Client** and **Contract** components, they are less applicable for the design of the **App** component. For example, the act of breaking up the final "User" of the network into **Entity** and **App** components allows the *user experience* to be designed at the **App** level, influencing the satisfaction of the **Entity** as they use the **App** to interface with the network. This "KPI" is beyond the scope of the p2p network architectural design problem. In addition, behavioral assumptions answering such questions as "How much would an **Entity** be willing to pay for a **Message**?" can be contained at the **Entity** level, while assumptions answering such implementation questions as "How does an **Entity** pay for a **Message**?" can be restricted to the **App** level (and its own **Client** interface).[3]

Thus, the main architectural design considerations in light of the system goals outlined above rest with the **Client** and **Contract** components, taking as given the existence of the **Entity** and **App** components in their foundational roles. Section 4.3 first provides an overview of the **Client**'s information partition, noting the utility in making a distinction between local and global information. Section 4.4 then provides further detail covering the characteristics of a **Client**, and how it interacts with the network in order to process **Messages** and **Files** from source to destination. Finally, Section 4.5 concludes with a brief overview of the **Contract** component—as this component is built around the provision of an incentive mechanism for **Clients**, this overview section leads naturally to the content of Documentation Digest #2, which will cover the cryptoeconomic incentive system from a Computational Social Science viewpoint.

4.3 Client Design and Information

A **Client** is a **Node** in the p2p network, routing **Messages** and storing (and retrieving) data. It is *aware* of its neighborhood N_t, which is comprised of

1. an **App**, responsible for conveying information to and from an **Entity**; and
2. one or more **Clients**, the identities of which are stored when **Messages** are received, or a ping is broadcast and responded to.

In addition, a **Client** is always aware of the **Contract** component, as it is publicly available and helps govern the messaging *reward structure*.

A **Client**'s awareness is thus divided into *local* and *global* information: **Apps** and other **Clients** interact locally, while the **Contract** component(s) interact at the global level. Although it may appear that global information, handled at

[3]This does not mean that architectural design has *nothing* to say regarding the structure of the **Entity** and **App** components—rather, they can be treated separately from the **Client** and **Contract** components because, conditional upon these components fulfilling the system goals, they may be specified using the KPIs most naturally applying to the foundational layer (such as **App** look and feel, **Account** access, etc.).

the **Contract** level, could become a centralized form of control, in practice the **Contract** level does nothing more than provide an auditable, trustworthy mechanism to verify the reward and penalty mechanisms that are implemented there (fulfilling system goal 4). It is important that this mechanism be publicly available, so that every participant in the network is aware of the "rules of the game"—for example, the reward mechanism for successfully routing a **Message** from an originating **Client** to a recipient **Client**, or for storing a **Message**, should be public, so that **Clients** can make informed decisions regarding their preferences on routing (system goals 1 and 2). Similarly, if a **Client** appears to be "self-routing", i.e. fulfilling vacuous requests on its own behalf (or on behalf of an **Entity** via an **App**), the penalty for doing so should be "on-chain" in order for that penalty to incentivize a **Client** to refrain from such behavior (supporting system goals 3 and 4).

By contrast, local information can inform a **Client** if another **Client** has reneged on a promise to route a **Message**, and what a **Client** has promised (if anything) in return for routing (or storing) a **Message**. It is important to emphasize here that there is little in the way of *enforcement* that is implemented between **Clients** via the network protocol—the protocol incentivizes *participation* in the network (since the protocol must be implemented for participation), but does not actively reward or penalize particular actions. The protocol is meant to handle low-level communication with (ideally) high throughput, and is optimized for the goal of routing **Messages** efficiently, while allowing a **Client** to identify and update their neighborhood of **Clients** and **Apps**. Thus, the **Client** has the freedom to select how they will use the protocol to respond to local events, without being constrained by the protocol in which actions they select.

4.4 Client Design and Network Characteristics

A **Client**'s actions are dictated by the network protocol and act to support the system goals (p. 14) of efficient **Message** transfer (goals 1 - 2) and cryptoeconomic incentivization (goals 3 - 4). These actions are based upon a minimal set of intrinsic *characteristics*, since **Clients** are hardware or software **Node** implementations of communication and computation units. These intrinsic characteristics, in addition to the **Client** neighborhood N_t (cf. Section 3.4) may naturally differ between **Nodes** (since the network is heterogeneous and decentralized):

Node Characteristic: Storage Capacity

A **Client** identified by an *index* i in a set of **Clients** V ("$i \in V$") has an idiosyncratic *storage capacity* $C_i \geq 0$, representing the amount of permanent storage space available to it.

Associated **cadCAD** variable: `storage`

Node Characteristic: File Cache

A **Client** $i \in V$ has an idiosyncratic *file cache* X_i^t, which contains all **Files** the **Client** has decided to store up to time t.

Associated **cadCAD** variable: `file_list`

Node Characteristic: Account

A **Client** $i \in V$ may have an associated **Account** $a_i \in A$, that is identified by the **App** using the **Client** as an interface.[a]

Associated **cadCAD** variable: `wallet`

[a] The ultimate controller of the **Account** is, of course, an **Entity** which uses the **App** as an interface, but for simplicity we assume that the **Client** is directly associated with an **Entity**'s **Account** to facilitate value transfers, settlement etc.

By virtue of its decentralization, every **Client** maintains its own local *state*, i.e. its idiosyncratic information about itself and its surroundings as formalized in its neighborhood. The local state is used by the **Client** to make decisions about **Messages** that are incoming, as well as to assess the contents of such **Messages**. Although until now we have used "**Message**" informally, to describe a unit of information transferred within the network, it is useful to formalize both a **Message** and one of its possible components, a **File**, to describe the **Client** architecture.

4.4.1 Client Capacity and Files

In order to fix the mapping between a **Client**'s local state and its decisions, it is important first to specify what is meant by "**File**", and then build the processes and decisions that a **Client** can take with regard to its request, its disposition, storage, etc. The definition of a **File** is *global* to the network, so that every **Client** agrees upon the characteristics of a **File** and agrees upon the syntax required for e.g. demanding or supplying a **File**. Thus, the **File** primitive is part of the global, or systems-level definition of the network.

File: Definition

A **File** f is a piece of information, or *datum*, possessing an identity $h(f)$, where $h(\cdot)$ is a cryptographic hash function. Every **File** has a *size* $s_f > 0$.

Here it is assumed that a **File** is *atomic*, i.e. it cannot be further broken down into sub-units, each of which may be considered a **File**. This is without loss of generality—in practice, a requested **File** may be broken into 'chunks' or 'blobs', which are routed separately from the sender to the receiver. In that context, a "**File**" as defined here would correspond to the smallest atomic sub-unit.

If a **File** is sent to a **Client** i, that **Client** must have enough storage capacity to store the **File**. Recall (cf. Section 4.4) that X_i^t is the file cache at time t. If a set of new **Files** arrives in between time t and $t+1$, the largest possible set of files to store for the **Client** is the union of F and X_i^t. From this set, the **Client** selects (according to some criteria, discussed below) a subset to store at time $t+1$. This subset must respect the storage constraints of the **Client**:

Constraint: (Local) Capacity

Given a Client i's file cache X_i^t in period t, and an arrival of incoming Files F, a Client selects its file cache for $t+1$, X_i^{t+1}, from $F \cup X_i^t$. The selected file cache X_i^{t+1} is subject to the constraint:

$$\sum_{f \in \{F \cup X_i^t\}} s_f \leq C_i. \tag{C1}$$

Associated **cadCAD** variable: `size`

Constraint (C1) says that a Client will not duplicate storage of an incoming File, if the File is already in their file cache X_i^t. In addition, the total storage size of all Files stored cannot exceed the storage capacity of the Client.

4.4.2 Client Bandwidth and Messaging

When a Client requests a File, or handles a request for a File from another Client, they participate in the creation or receipt of a "Message". A Message is the communication unit that travels along edges within a Client's p2p neighborhood, and like Files can be formally defined in a way that all Nodes agree upon:

Message: Definition

A Message m is a *data structure*, comprised of one or more elements. Elements may include Files. The size of a message is $s_m > 0$.

Messages can be transmitted from a Client provided that the capacity of the upload communications channel allocated to (or possible for) the Client is not exceeded. Similarly, a Client can receive a Message if their download communications channel capacity is not exceeded:

Message: Bandwidth

Every Client i in the set of Clients V has an idiosyncratic *upload bandwidth* B_i^u, and an idiosyncratic *download bandwidth* B_i^d. Without loss of generality bandwidth may be measured in units of bits-per-second (bps).

Upload bandwidth, associated **cadCAD** variable: `out_bandwidth`

Download bandwidth, associated **cadCAD** variable: `in_bandwidth`

Bandwidth is thus a *local constraint*: given the hardware or software implementations of Clients, each brings with it a particular constraint on how many Messages it can send and receive:

Constraint: (Local) Bandwidth

In a given time interval, each node $i \in V$ may transmit a set of messages M_i only if

$$\sum_{m \in M_i} s_m \leq B_i^u, \tag{C2}$$

and may receive a set of messages R_i only if

$$\sum_{m \in R_i} s_m \leq B_i^d. \tag{C3}$$

For simplicity, if it is ever the case that **Constraint** (C3) is violated for any **Client** i, this is interpreted as a Distributed-Denial-of-Service (DDoS) attack, and none of the **Messages** in R_i are received by i. It is assumed that the upload **Constraint** (C2) is never violated.

The primary purpose of **Messages** is to request **Files** from other **Clients**.

Message: File Request

A **Client** $i \in V$ may send a **Message** to any other **Client** in its neighborhood N_t^i *requesting* a **File** f with size s_f.

A File Request is a **cadCAD** system `Event`.

When a **Message** includes a **File** request, it may be that the **Client** sending the **Message** also includes an incentive for other **Clients** to participate in the receipt of the **File** (such as finding an optimal route in the network, cf. Section 4.4.3 below):

Message: Payment

A **Message** Payment ρ_m^i may be included in a **Message** m that contains a **File** Request from **Client** $i \in V$. The Payment is a promise from the **Client** sending m to pay ρ_m^i in the event that the **File** is received.

Node Characteristic: Message History

A **Client** remembers the list of all **Messages** that have been routed to (or through) it.

Associated **cadCAD** variable: `message_list`

4.4.3 Client Communication and Request Routing[4]

When a Client makes a request for a File, that request—encapsulated in a Message—is initially transmitted only to the Client's immediate neighborhood (recall that a Client knows nothing about the global topology of the network, and can only send a request to its peers). Although it may be the case that a Client's peer has the File the Client is requesting, in general the File may be one or more Clients, or "hops", away from the requesting Client.

Thus, the system must be designed so that a Client's File request is *routed* to the Client that can serve the request, and that every Client in the chain to and from the serving Client is properly incentivized to provide the most efficient path possible. In order to support the KPIs relevant to routing (cf. KPIs 2 and 3, p. 15), it is necessary to provide a minimal scaffolding of how a Client routes a File request, and how that request creates a path of Clients that can be verified in a trustworthy fashion.

Routing: Initiating Node

There is an initial Client $i \in V$ that initiates the File Request. This Client has an associated Account address a_i, which is used to cryptographically sign the Request. This may or may not be the Client that is Demanding the File.

Routing: Receiving Node

There is a recipient Client $j \in V$ that demanded the File. This Client has an associated Account address a_j. This Client may or may not use its Account address to cryptographically sign a **Proof** that the Request has been fulfilled. If not, then another Client p has provided the **Proof**, but this Client is not considered to be a part of the File Request.

Routing: File Location

The location of the File being requested is its identity $h(f)$ (cf. File Definition, p. 18).

Routing: Payment Offer

The receiving Client j offers an amount q in SNT as payment for the fulfillment of the File Request. This is an *ex ante* offer in the sense that no collateral or escrow lock is assumed.

[4]There is a parallel workflow for File storage as outlined in the original Mathematical Specification from Block Science—as the File request routing workflow sufficiently demonstrates the information flow aspect of the STATUS network, the storage workflow description has been omitted from this digest in the interest of brevity.

Routing: Deadline

The block height at the time of **Client** i's initiating **File** Request is b_0. **Client** j specifies a *deadline* block height $b > b_0$ by which time the **File** must be received, in order for the payment q to be transferred.

Routing: Task

A routing task begins with a **File** request **Message** and ends with a **File** receive **Message**. The task is constructed as a data structure, using the initiating **Client**'s **Account** address, the receiving **Client**'s **Account** address, the **File**'s identity, the payment offer, and the block height deadline and initial time:

$$\mathcal{D} := \langle a_i, a_j, h(f), q, b_0, b \rangle. \tag{R6}$$

Routing: Path

A *path* is a sequence of **Clients** $\{n_i, \ldots, n_k\}$, from a **Client** $i \in V$ to another **Client** $k \in V$, such that each **Client** in the sequence belongs to the neighborhood of the previous **Client**.

A path may also be thought of as the **Edges** connecting the **Clients** in the sequence, such that 1) **Messages** such as **File** requests follow a path, and 2) **Files** provided in response to **File** requests (also contained in **Messages**) are received. Viewed this way, **Messages** are "passed" from **Client** to **Client** in much the same way as a 'bucket brigade' passes water from a distant well to a burning building. Each member of the brigade knows their immediate neighbors, and can attest, or prove, that they 1) have received a bucket from their immediate predecessor, and 2) have passed on a bucket to their immediate successor. If every member is (presumably after the fire has been extinguished) surveyed and their proofs collected, then globally one can say that buckets have been *proved* to have followed the brigade from well to building.

Naturally, in this analogy there is only one possible 'path' for buckets to follow—but in a general network this will usually not be the case, and there will be many paths available to a **Message**. Nevertheless, it will be possible to 'survey' the **Clients** participating in a routing task so that each **Client** can prove receipt of a **Message** using the block height information contained within the task.

Routing: Trace

A *trace* is a path that is provable, i.e. every **Message** passed along an **Edge** between a **Client** and its immediate successor (in its neighborhood) along the path can be proven to have taken that **Edge**.

The existence—but not necessarily the *use*[5]—of a trace is a **necessary condition** to allow a reward (and penalty) system to be implemented in the network. Without the ability to prove that a **Message** has followed a particular route, it is not possible to provide incentives for **Clients** to attempt to route a **Message** that arrives as efficiently as possible. At the same time, it is not the case that a trace is **sufficient** to ensure that **Clients** always efficiently route **Messages**: there need to be additional safeguards in place to ensure that **Clients** do not select (and prove) routes that are inefficient or self-serving and thereby act against the collective goal of the network. Creating an incentive system with both necessary and sufficient conditions, given the network architecture provided in this Digest, is the subject of Documentation Digest #2.

4.5 Contract Design and Incentivization

The **Contract** component acts to incentivize behaviors that a **Client** can undertake, reinforcing the system goals (p. 14) that directly involve incentivization (goals 3 - 4). As previously mentioned, it is important that the **Contract** component be publicly viewable at the global level, because it must be *common knowledge* that the incentive structures (i.e. the Smart Contract algorithms) are available for everyone to inspect. Without such a common knowledge assumption (see e.g. Aumann 1976, for a technical discussion and analysis) **Clients** will not be able to agree upon which incentives are available, and may thus be unable to effectively coordinate their actions to support the system goals.

A full characterization of the **Contract** component, including its foundation upon Computational Social Science and incentive theory and its relationship with the emergence of local (p2p) cooperation, will be presented in Documentation Digest #2. In this and the following subsections, only a brief schematic of the **Contract** actions and behaviors will be outlined.

4.5.1 Contract Actions

The degree to which the **Contract** component can actively incentivize behavior depends upon the degree to which it is involved in the settlement process following the (successful or unsuccessful) transfer of a **Message**, or storage of a **File**. Because on-chain activities can lead to potentially unacceptably long transactions and/or settlement times, the **Contract** component is *nominally passive*, but *potentially active*. In other words, there exist actions that a **Contract** component can take which by their existence help incentivize behavior, but are never (or rarely) executed. They are (in a sense) a 'show of force' that keeps **Clients** from attempting to take actions that benefit themselves at the expense of the systems goals of the network.

Although this "speak softly, and carry a big stick"[6] approach allows for direct value transfers to take place between **Nodes** (such as **Apps** or **Clients**), it is often more natural to consider **Accounts** as being placed within the **Contract** component (cf. Figure 5). This provides an interpretation of value settlement occurring 'as if' the **Contract** component were actively participating, which is intuitively appealing for two reasons. First, this provides a way of encapsulating

[5] Cf. Section 4.5.1.
[6] T. Roosevelt (1900), attributed to a West African proverb; see e.g. Wikipedia.

the financial network, and hence value transfers, into a hierarchy of information. The 'coarsest' information partition is at the global information level, where the **Contract** acts as a publicly auditable information source, while the 'finest' information partition is at the local information level, where **Clients** and (as a consequence) **Apps** transfer value according to p2p local relationships. Thus, **Accounts** are initially contained at the coarsest information level (with the **Contract** component as the 'container'), while local interactions between **Nodes** may still feature value transfers between **Accounts**.

Second, placing **Accounts** at the same level as the **Contract** allows one to view the **Contract** as a "super" **Account** in its own right. This interpretation is particularly important for *governance*, as it is assumed that updating algorithmic code within a Smart **Contract** will require replacing (or subsuming) existing code at a particular **Account** address. In this way, a **Contract** component both *acts as* an **Account**, and also *acts on* other **Accounts**, as needed.

With this interpretation in mind, the following actions are considered to be available to the **Contract** component, as both an **Account** and as a container of **Accounts**:

Contract: Verification

A **Contract** may *verify* that a route, registered with the **Contract** by a **Client**, is a trace. A **Contract** may store a hash of this verification in a publicly accessible DLT address.

Contract: Reward

A **Contract** may *reward* one or more **Clients** in a route, by providing SNT to an **Account** in proportion to their participation in the route (e.g. by dividing a lump sum between all participating **Clients**, or by another allocation mechanism).

Contract: Penalty

A **Contract** may *penalize* one or more **Clients** by removing SNT from an **Account**, for behavior that runs directly against the defined system goals, whether or not that **Client** is participating in a route. Examples of such behavior include attempted **Client** self-rewarding, non-payment of promised funds in return for **Message** routing and/or **File** storage, or other acts of malfeasance.

4.5.2 Contract Behavior

The **Contract** component is designed to facilitate the computation and communication network, and the financial network, with only rare intervention. Its actions thus embody a *threat* of intervention, in much the same way that a personal income tax audit from a tax authority embodies a threat of government intervention in the event of suspected

tax fraud. The full characterization of the **Contract** incentivization structure, including its support of an emergent cooperative outcome between **Clients** using their local information, will be presented in Documentation Digest #2.

References

[1] Robert J. Aumann. "Agreeing to Disagree". In: *The Annals of Statistics* 4.6 (1976), pp. 1236–1239. URL: http://www.jstor.org/stable/2958591.

[2] Henry Hexmoor. *Computational Network Science: An Algorithmic Approach*. Elsevier (Morgan Kaufmann), 2015.

[3] Dahai Liu. *Systems Engineering*. CRC Press, 2015.

[4] Charles S. Wasson. *System Engineering Analysis, Design, and Development: Concepts, Principles, and Practices, 2nd Ed*. Wiley, 2015.