GILDAGE: Distributed Architecture for Smart Contract Trusted Third Party Services

Apurva Shah
July 29, 2018 (v1.0)

Abstract

Oracles are the mechanism by which smart contracts communicate with off-chain resources. Instead of using dedicated oracles, platforms reduce development and infrastructure costs and lead to more standardized and robust services. Existing oracle platforms either notarize messages or provide proof of authenticity for data flow. The important issue of transposing blockchain and off-chain identities remains unaddressed. This is critical for securely accessing key third party services such as financial institutions or medical records while complying with confidentiality and privacy regulations.

We propose a novel platform called Gildage. We discuss how a distributed architecture without any central authority and Shamir's shared secrets can solve message authenticity and identity transposition in a unified way. We cover detailed cryptographic steps and analyze the system architecture from the standpoint of security and privacy. We describe the broader Gildage ecosystem including distributed architecture governance, self-sustaining platform monetization and service discovery for smart contract developers. Also, we propose using Gild utility tokens for generating liquidity to create the platform infrastructure and incubate smart contract startups that build on the platform.

Contents

1. Introduction

Blockchain technology has enabled trustless networks such as Ethereum [BUTE 2013] at a critical operational scale reminiscent of the early days of internet and mobile. Peer to peer transactions based on implicit confidence in the cryptographically secure network rather than through trusted intermediaries or centralized platforms opens up numerous opportunities for reducing cost and latency of existing products and services [NAKA 2008].

Smart contracts [SZAB 1997] [FLOO 2017] combine trustless transactions with workflow automation and business logic in an open and generalized way. This development in conjunction with the critical mass of public blockchain networks has set the stage for massive disruptive innovation in both consumer and business applications. If network effects hold true, the adoption of these new social and business models will rise at an exponential rate.

One significant limitation of blockchains and smart contracts running on them is that they are closed systems. As such, they are able to handle network addresses, on-chain transactions and tokenized assets in an elegant way. Unfortunately, incorporating externalities in the form of real world identities, events and assets is far more problematic. For public blockchains to realize their full potential, it is critical to solve the problem of combining trustless transactions with trusted external services while maintaining the flexibility and extensibility inherent to smart contracts.

We begin by surveying current approaches to building *oracles*, as these external services are often called in section (2). We propose a distributed service architecture called *Gildage* that attempts to solve this problem in a robust and generalized way in sub-section (3.a) - (3.d). We evaluate the architecture from the standpoint of security and regulatory considerations in sub-section (3.e) & (3.f). In keeping with the trustless, open ideology of blockchains, we present our service architecture within the context of an open service ecosystem that does not require a central authority in section (4). We present an example smart contract lifecycle to make the ideas concrete in section (5). Finally, we discuss how *Gild* tokens could generate liquidity not just to fund the service infrastructure but also to incubate new applications within the ecosystem in section (6).

2. Oracle Survey

Entities that harvest smart contract requests from the blockchain and connect to external APIs and trusted third party (TTP) services to bring in external data and connectivity from the off-chain world are called oracles. Oracles can be inbound to handle off-chain events and data input or outbound to notify state changes or to initiate transactions in off-chain services. Their off-chain connectivity can be to software services & APIs or even directly to hardware sensors or IoT networks. In terms of architecture, oracles fall in one of two service classes: TTP and consensus networks.

2.a TTP

In its simplest form (Figure 1), a dedicated server polls the blockchain for requests from one or more contracts. Requests are handled by first transposing network IDs to service IDs, for example Ether addresses to bank account numbers. The request is than reconstituted as an API call or microservice request and passed on to the actual backend service provider. When the backend service returns asynchronously with acknowledgment or query results, they are once again transposed back to blockchain network IDs and then sent to the smart contract via on-chain transactions using the network ID of the service itself.

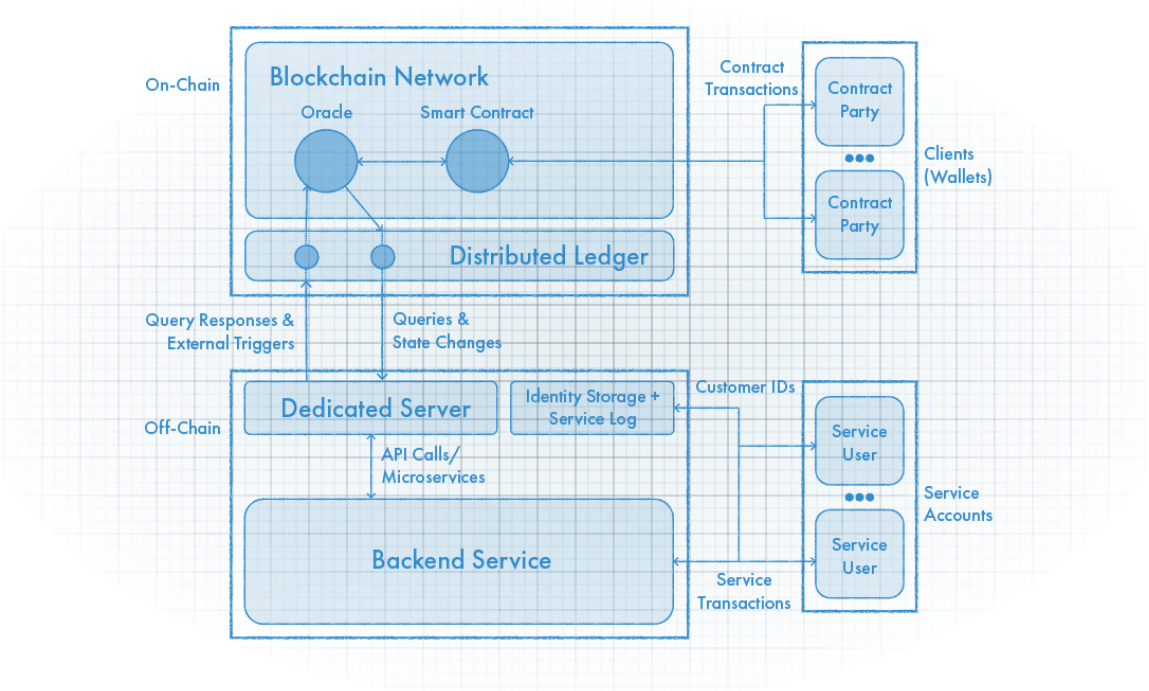

Figure 1: TTP Oracle architecture with dedicated server

Most smart contract external connectivity currently falls in this class. While they suffer from duplicative infrastructure and code, they are simple, flexible and highly controllable. They do suffer from an inherent problem related to confidentiality. Given the transparent nature of public blockchains, network IDs are visible and traceable back to every transaction since the start of the chain. If corresponding real world identities were discoverable, that would be a serious breach of confidentiality. In a dedicated oracle, at the very least, the TTP service would have to store the relationship between network ID and its corresponding service ID. That in turn creates a serious vulnerability for maintaining anonymity of blockchain identities.

A variation on the dedicated oracle service involves inserting an intermediary between the on-chain smart contract and off-chain TTP service provider to turn the on-chain to off-chain interface into a platform (Figure 2). In addition to reducing development and infrastructure costs, this approach leads to more standardized and robust services. The intermediate platform can notarize or otherwise provide proof of authenticity for data flowing through it [ORAC 2017]. In current approaches, authenticity is ensured through software algorithms [TLSN 2014] and/or hardware enclaves [ZANG 2016].

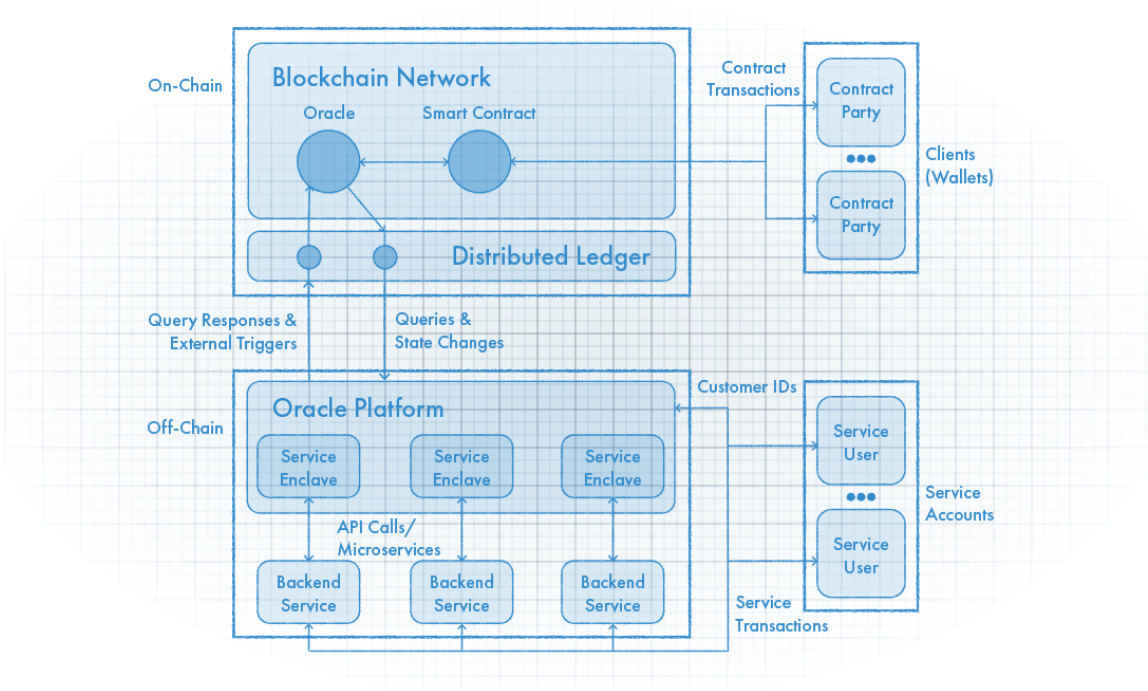

Figure 2: TTP Oracle architecture with intermediate platform

2.b Consensus Networks

In keeping with the blockchain philosophy of distributed trustless networks, oracles can aggregate multiple data sources to prevent manipulation and improve the quality of information supplied to smart contracts. An interesting application of this technique is in building prediction markets [PETE 2018] [KÖPP 2018]. For a small number of inputs and simple aggregation, the on-chain Oracle contract can simply accept multiple inputs over a prescribed time window and formulate the consensus response. This has the advantage of being completely auditable in terms of both the inputs recorded in the immutable ledger as well as the smart contract logic used for aggregation. In situations involving many more input sources or more complex aggregation schemes, figure (3) shows how Oracle platforms can be responsible for aggregating inputs on their own or in conjunction with the on-chain Oracle contracts [ELLI 2017] [OCHN 2017].

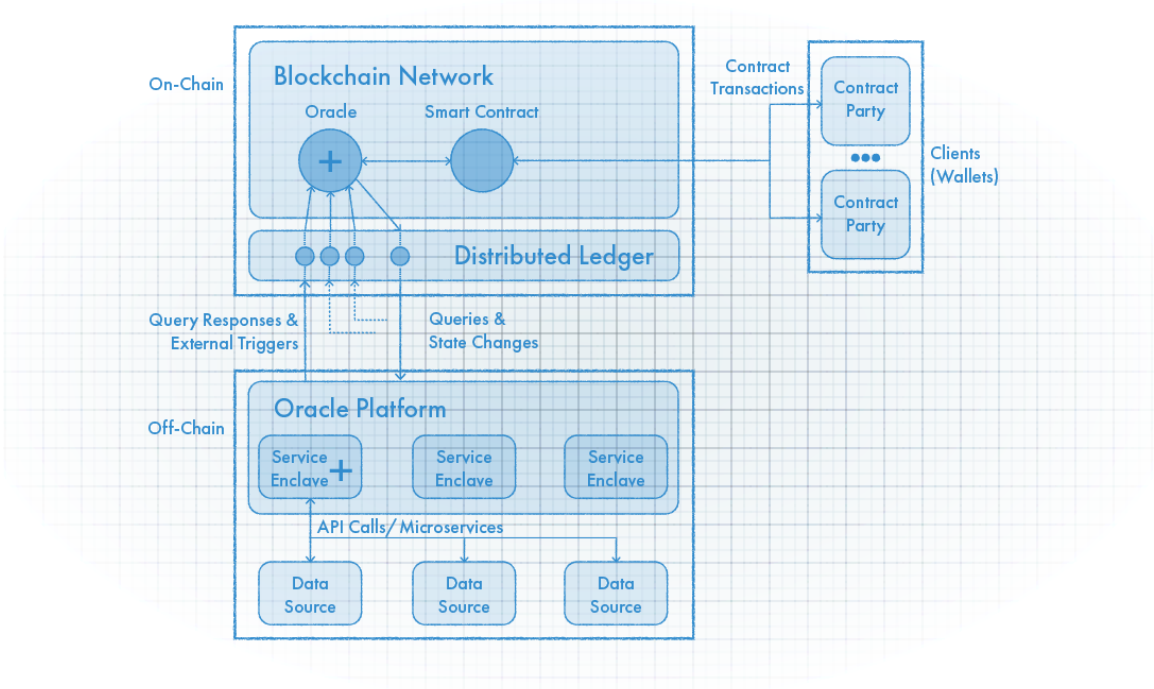

Figure 3: Consensus Oracle architecture with on & off-chain aggregation

3. Platform Architecture

Gildage connects on-chain smart contracts to one or more TTP services. We do this in a highly flexible way and without limitation on the type of contracts or class of services being incorporated. Unlike other existing or proposed oracle platforms discussed in section (2.a), we leverage a

distributed architecture with non-colluding participants to ensure authenticity and robustness of requests flowing through the platform. Furthermore, we address the critical problem of transposing blockchain network IDs to real world service user IDs without any centralization of either the mapping process or storage of relationship data.

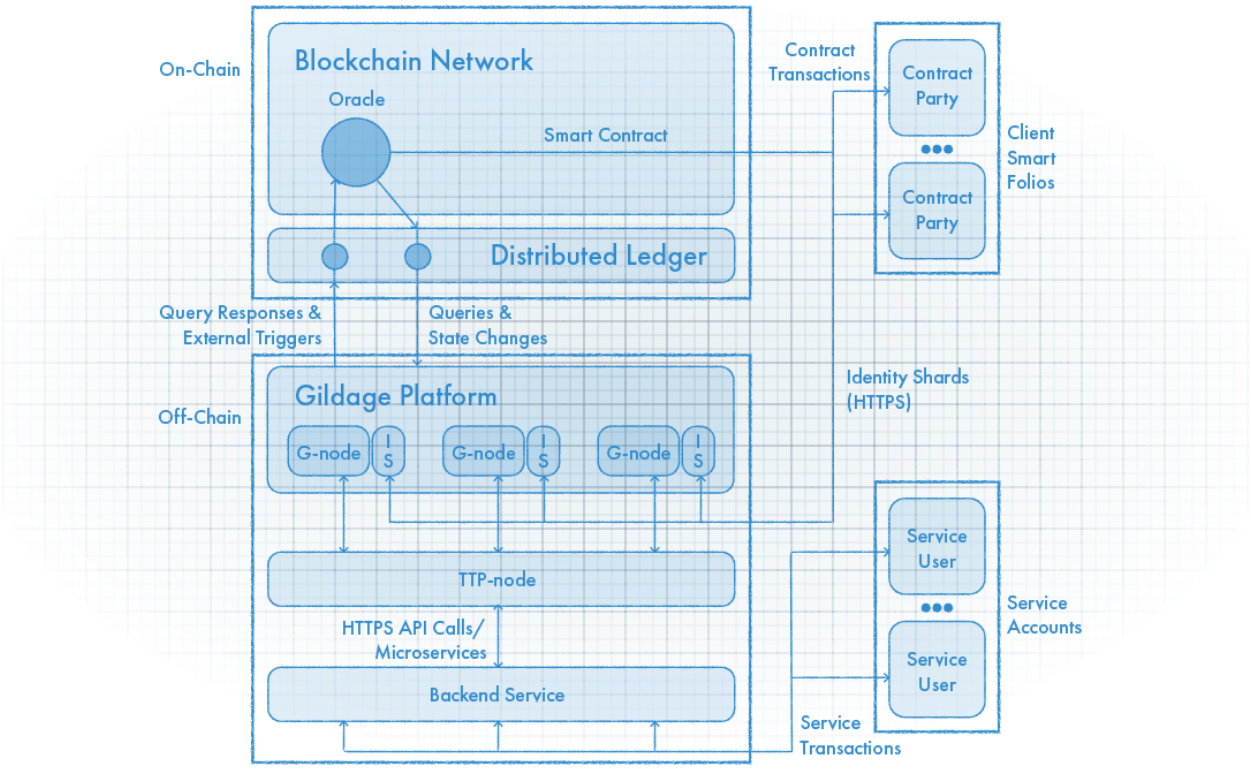

Figure 4: Gildage Platform Architecture

3.a G-nodes

G-node servers are the core processing and storage units of the platform. They intermediate messages between on-chain smart contracts and off-chain TTP service nodes. A smart contract (C_a) initiates a request (E) to TTP (S_a) by writing the event and related request data to the blockchain ledger. All active g-nodes (G_1-G_n) continuously poll newly mined blocks for events. Of these *m* nodes have the require secret to transpose the message and will therefore act on E. They reconstitute the request by replacing the blockchain transaction hash with a unique Gildage identifier hash [DAME 1999]. Any network addresses in the request are also transposed as discussed further in section (3.b). The reconstituted message is than passed to S_a.

G-nodes communicate with TTP-nodes over secure TCP/IP sockets with minimal latency. In the case of the message E, S_a will receive requests with the same Gildage identifier hash from one or more g-nodes. Integrity of the request is ensured by setting a critical threshold of g-nodes (*t*) that must convey the same request. This scheme also ensures redundancy since not all G_n nodes are required to participate. An appropriate value for *t* that balances security and robustness is discussed further in section (3.e).

All g-nodes have a unique blockchain network ID-IP address pair registered with the Gildage g-node smart contract as detailed in section (4.a). This node table will be used by TTP-nodes to ensure authenticity of the g-nodes.

3.b Identity Shards

One of the key challenges that the Gildage platform addresses is converting between blockchain network IDs and real world service user IDs without any centralized server or data store. Any centralization of this function or persistent storage of this identity linkage would create a significant security vulnerability and undermine anonymity, which along with immutability, are the supporting pillars of blockchain transparency and foundational to a trustless network.

We combine Shamir's threshold secret sharing cryptographic scheme [SHAM 1979] with the distributed g-node architecture to solve identity transposition. The client's smart folio, discussed further in section (3.d) is the only place from which the identity pair originates. This is similar to blockchain software and hardware wallets that hold private keys for network participants [KROM 2016]. Identity pairs consisting of blockchain network address, N_a, and service account ID, S_i, are split into bidirectional secrets we call *identity shards*. These are stored securely on a randomly selected set of *m* g-nodes within their persistent store (IS) as shown in figure (4).

Details for each key cryptographic step are as follows.

(i) Adding contract C_a, relying on service T_a, to client folio with blockchain address N_a and service account ID S_i.

Step 1. KN = Random(256-bytes)
KS = Random(256-bytes)

Step 2. $N_a^{'}$ = Encrypt(KN, N_a)
$S_i^{'}$ = Encrypt(KS, S_i)

Step 3. $[KN]_m^1$ = ShamirSplit(KN, m, t), t: threshold number for reconstructing K
$[KS]_m^1$ = ShamirSplit(KS, m, t)

Step 4. $[G]_m^1$.map(x ⇒ Store({C_a, T_a, S_i, $KN_x^{'}$, $N_a^{'}$}))
$[G]_m^1$.map(x ⇒ Store({C_a, T_a, N_a, $KS_x^{'}$, $S_i^{'}$}))
G-nodes 1..m are picked at random by the client folio.

(ii) Process event E on g-node G_x.

Step 1. Read from published block: E→{C_a, *eid*, T_a, R→ (N_a, …)}
Where *eid* is a unique ID for E and N_a is one of the parameters to request R intended for TTP service T_a.

Step 2. *gid* = hash(*eid*)

Step 3. Lookup(C_a, T_a, N_a) = ($KS_x^{'}$, $S_i^{'}$)

Repeat Step 3 for any additional blockchain identity parameters of R.

Step 4. Message(T_a , R→{*gid*, ($KS_x^{'}$, $S_i^{'}$), …})

(iii) Process request R on TTP-node T_a.

Step 1. Read requests from socket: $[R]_p^1$ →[x ⇒ {*gid*, ($KS_x^{'}$, $S_i^{'}$), …}]$_p^1$ where $1 \le p \le m$

Step 2. KS = Recombine[x ⇒ KS_x]$_p^1$ provided $p > t$

Step 3. S_i = Decrypt(KS, $S_i^{'}$)

Step 4. Process R($S_{i, \ldots}$) = $R^{'}$ →{*gid*, $S_{i, \ldots}$}

Note that Step 4 is asynchronous and could be performed either on or off TTP-node T_a.

Step 5. $[G]_m^1$.map(x ⇒ Message(G_x, $R^{'}$))

(iv) Process response $R^{'}$ from T_a on g-node G_x.

Step 1. Read from socket: $R^{'}$ →{*gid*, $S_{i, \ldots}$}

Step 2. $[E]_{pending}$.map(y ⇒ hash(*eid*) = *gid*? C_a = E_y{C_a}, *eid* = E_y{*eid*})

If T_a does not respond to E within specified time, G_x sends timeout error to C_a. This prevents $[E]_{pending}$ from growing indefinitely.

Step 3. Lookup(C_a, T_a, S_i) = ($KN_x^{'}$, $N_a^{'}$)

Repeat Step 3 for any additional service identity parameters of $R^{'}$.

Step 4. TXSend(C_a , TX→{*eid*, T_a, $R^{'}$ →{($KN_x^{'}$, $N_a^{'}$), …}}

(v) Process response TX in blockchain contract C_a.

Step 1. Buffer transactions: $[TX]_p^1$ →[x ⇒ {*eid*, T_a, $R^{'}$ →{($KN_x^{'}$, $N_a^{'}$), …}}] where $1 \le p \le m$

Step 2. KN = Recombine[x ⇒ KN_x]$_p^1$ provided $p > t$

Step 3. N_a = Decrypt(KN, $N_a^{'}$)

Step 4. Process TX→{*eid*, T_a, $R' \rightarrow (N_a, \ldots)$}

(i) need only be performed when N_a adds contract C_a to their folio. However, for enhanced security and data persistence, it can be repeated to refresh the shared secrets stored on the g-nodes. (ii)-(v) are repeated for every request originating from C_a. They can also be reversed starting with (iii) in case the TTP service wants to proactively notify C_a of an external trigger.

3.c TTP-nodes

TTP-node servers can provide internal services, act as micro-service interfaces to an external platform [HASS 2016] or connect to enterprise servers using the widely supported and secure HTTP protocol. They are entirely managed by the TTP service provider and beyond complying with Gildage's socket protocol requirements, their architecture specification and scalability is entirely at the discretion of the service provider.

Internal services can be computationally simple processes and without significant external data requirement. Examples are random number generators, timers and message forwarding services. External services can be arbitrarily complex, for example, banking and payment APIs for money movement or IoT device swarms [WEIS 1991].

All TTP services will have a unique blockchain network ID–IP address pair registered with the Gildage TTP-node smart contract as detailed in section (4.b). This service registry will be used by g-nodes to ensure authenticity of the TTP-nodes they connect with over secure sockets.

3.d Smart Folios

We propose an extension to the typical crypto-currency wallets used for storing, sending, and receiving crypto-tokens [WALL 2018] [MMSK 2018] called *smart folios*. A smart folio is also capable of identity management, operating on and viewing transaction history for smart contracts in which the wallet owner is participating. We show the basic layout of a smart folio in figure (5).

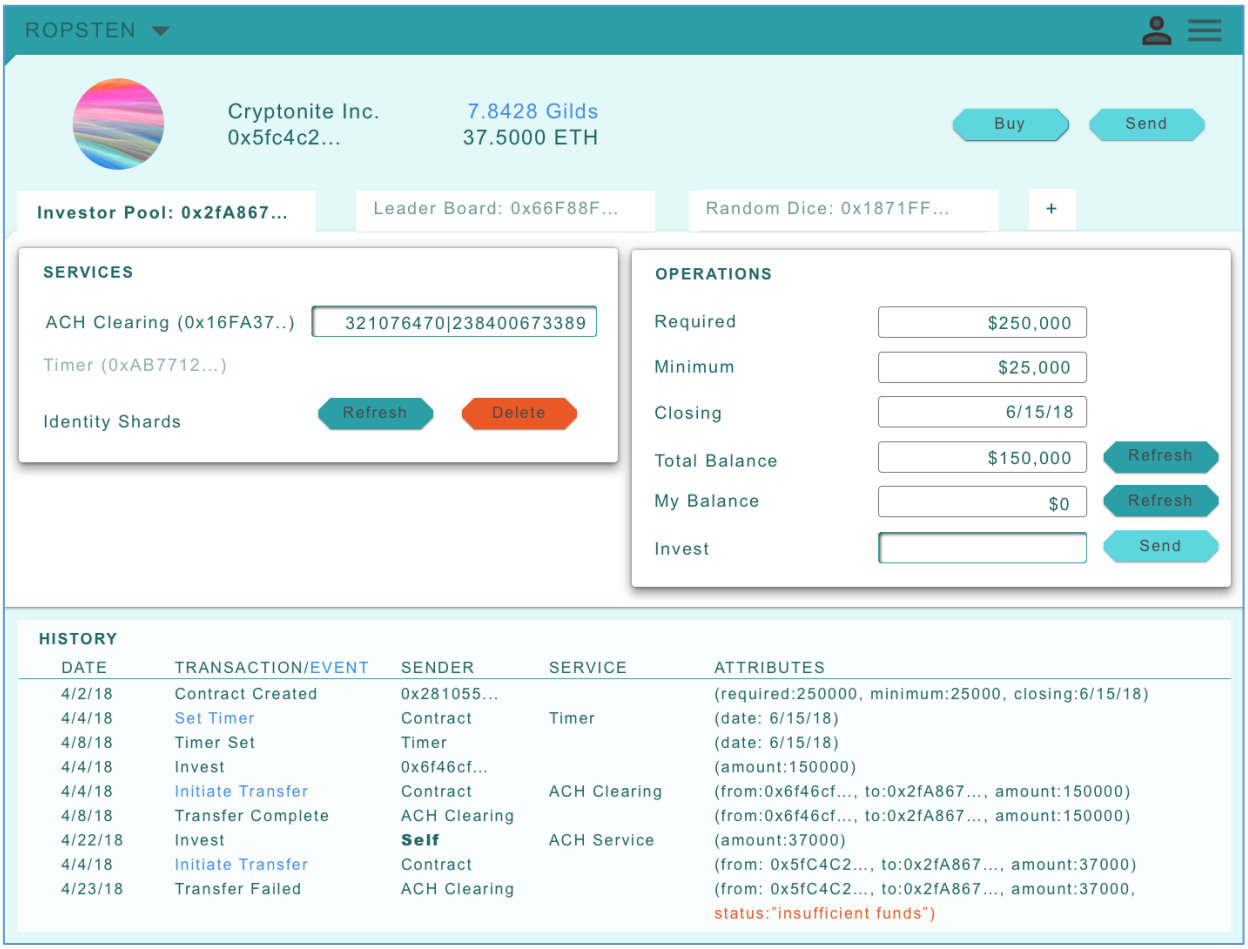

Figure 5: Smart Folio

As with crypto-currency wallets, smart folios can be implemented as standalone applications, browser extensions or with hardware for added security.

3.e Security Considerations

We first discuss two primary security considerations – maintaining integrity of the messages that are handled by the platform and ensuring that on-chain and off-chain identity relationship are not compromised. We are able to leverage the distributed architecture in order to solve both concerns in a unified way.

In all the network has n g-nodes. Of these, a client's identity secret is split between m g-nodes selected at random. Threshold t of subset m g-nodes must convey the untampered transposed message to a TTP-node or smart contract. t is set to $(m - 1)/2 + 1$, where m is ideally an odd number. In

other words, more than 50% of g-nodes in possession of the identity secret. By statistical sampling and assuming the number of g-nodes is sufficiently large, we argue that over 50% of g-nodes must collude in order to manipulate message content or decrypt the user's identity pair. In order to dissuade majority attacks, we use naive Proof of Stake (PoS) [BUTE 2014] as discussed in g-node governance described in section (4.a).

Next we analyze how blockchain network address and TCP/IP identity of the g-nodes and TTP-nodes is verified. Client folios and TTP-nodes confirm g-node identity using the g-node table maintained by the Gildage g-nodes smart contract. Since changes to this g-node table are governed by the published smart contract described in section (4.a) any past or proposed changes are publicly auditable. Similarly, identity of TTP-nodes is confirmed by checking with the registry maintained by the Gildage TTP-node smart contract outlined in section (4.b). While it does not have the same governance mechanism as the g-node smart contract, the blockchain network address of the TTP service is known and has to be bound to the end user smart contract in which the clients are participating. In other words, the TTP service identity can be considered bona fide provided the end user smart contract has not been compromised.

Finally, we consider network traffic security. All communication between client folios and g-nodes such as distributing identity shards is done over secure HTTPS. Likewise, all messages between g-nodes and TTP-nodes use asymmetric encryption with the target's public key.

3.f Privacy Considerations

Confidentiality and privacy regulations [GDPR 2016] pose a significant challenge to transparency and immutability necessary for public blockchains. For example, the "right to be forgotten" is impossible to comply with once personal data leaks onto the public blockchain. On the other hand, for smart contracts to have broad applicability they need to be able to trigger actions and respond to events in the real world. The Gildage platform provides a secure path to transpose on-chain and off-chain identities without constraints on the contracts or TTP services they access. The end client can simply break all connections between their on-chain and off-chain identity by deleting identity shards for any contract in which they are participating. In case of rogue g-nodes that hold on to deleted identity shards, collusion would be required with majority of g-nodes as discussed

in section (3.e) to unmask the client's identity and any personal data associated with it.

4. Operating Ecosystem

In section (3) we provided a detailed overview of the Gildage system architecture. In this section we will cover the rest of the ecosystem including the smart contract that establishes g-node governance rules and manages the g-node table in section (4.a). Gildage is a metered service with g-nodes being paid in utility tokens, called *Gilds*, for every message they handle. Client contracts must place Gilds in the client contract escrow as described in section (4.c) before they can access services. TTP services can specify their own service charge which is added to the platform service charge when funds are withdrawn from the client contract escrow. Alternatively, given that TTP services are often tied to real world relationships between clients and service providers, such as subscriptions or bank accounts, TTP services may choose to underwrite charges on behalf of client contracts. These details are specified in the smart contract for TTP services in section (4.b). Finally, in section (4.d) we present a service directory that smart contract developers can use to search for TTP services.

4.a G-node Governance

To be future proof, distributed architectures must be set up with appropriate governance in place. For Gildage platform this is provided by the g-node smart contract (GNSC). Central to GNSC is the g-node table (GNT) which contains all active g-nodes comprising the platform. It contains the blockchain network address and IP address of the g-node server. Only network addresses listed in the GNT are allowed to operate on the GNSC.

Here are some key governance aspects of GNSC:

- Bootstrap GNT with 3 nodes.
- The number of slots in GNT is proportional to the platform's message traffic. As traffic rises, new slots are opened in pairs so that the total number of nodes always remains odd.
- Operators can apply to host g-nodes by entering a lottery which requires Gilds to be deposited. When open slots become available, the lottery randomly picks from all applicants.

- Key statistics related to each g-node such as peak message load, average message processing time, message rejection ratio, etc. are recorded in GNSC at regular intervals.
- G-nodes can be removed from GNT in two ways – if they handle less than $1/n$ of platform traffic with some margin for error, as measured by their Gild payment over a defined period or if $2/3^{rd}$ g-nodes vote to have them ejected. The former culls non-performing nodes and latter is a form of peer penalty and useful in case a g-node operator is not managing their server securely or otherwise violating the basic operating principles of the platform.

4.b TTP Registry

The TTP smart contract (TTPSC) has the following functions – maintain a registry of services (TTPR), instructions for service payments, an optional Gild escrow, and logging service performance data. Any third party service can register with the contract by simply providing a blockchain network address and a server IP address for the TTP-node. Client contracts can issue service request events provided the TTP address being accessed can be found in TTPR. This look up also provides the IP address for the TTP server that the g-node will forward the transposed request to. At periodic intervals, g-nodes send service performance data to TTPSC for all TTP-nodes they have interacted with. This includes details such as number of messages handled, acknowledgement time outs, etc.

4.c Client Contract Escrow

The platform never charges actual client parties that participate in client contracts. Therefore, the client smart contract (CSC) holds Gilds for client contracts that want to access paid TTPs. Before processing event E, g-node will check T_a's payment instructions in TTPSC. If T_a is a paid service, its charges will be added to the platform charge and the necessary balance withdrawn from CSC escrow. However, if T_a is choosing to underwrite platform charges, the same will be taken from TTPSC escrow. Any request to withdraw funds from either escrows are queued until after escrow obligations are settled at regular intervals such as hourly or daily.

4.d Service Directory

As explained in section (4.b) TTPSC holds public performance data related to each service aggregated from g-nodes. Any interested provider can make this data available in the form of a searchable directory. Additional information such as service description, charges, user ratings and comments can be used to make the directory more comprehensive. For smart contract developers this is an invaluable resource and facilitates TTP discovery in a systematic way.

5. Smart Contract Lifecycle

In this section we tie together the Gildage system architecture described in section (3) and ecosystem discussed in section (4) by using a simple client contract example.

InvestorPool is a smart contract that allows the contract owner, typically a commercial bank or investment fund we refer to as *Bank*, to setup an escrow on behalf of a startup we will call *NewCo*. The goal is to raise a specified amount from multiple investors by a predetermined closing date. On closing, a successful raise would see the escrow amount forwarded to NewCo, otherwise funds are returned back to investors. We assume that Bank will be paying for all service and platform charges as the escrow manager.

Solidity [SOLI 2017] code for InvestorPool as a contract state machine is presented in Appendix A. It inherits from GildageContract that implements binding with TTPSC and Shamir secret thresholding. Code is annotated to match figures (6), (7), (8) & (9) that cover the states of the contract lifecycle.

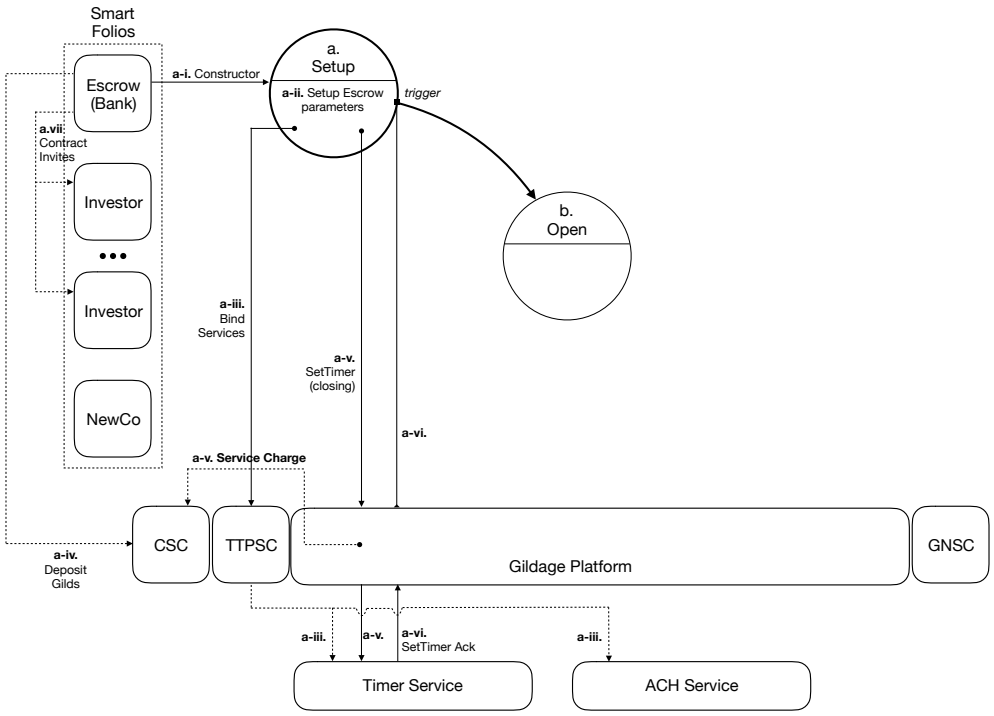

Figure 6: InvestorPool, Setup State

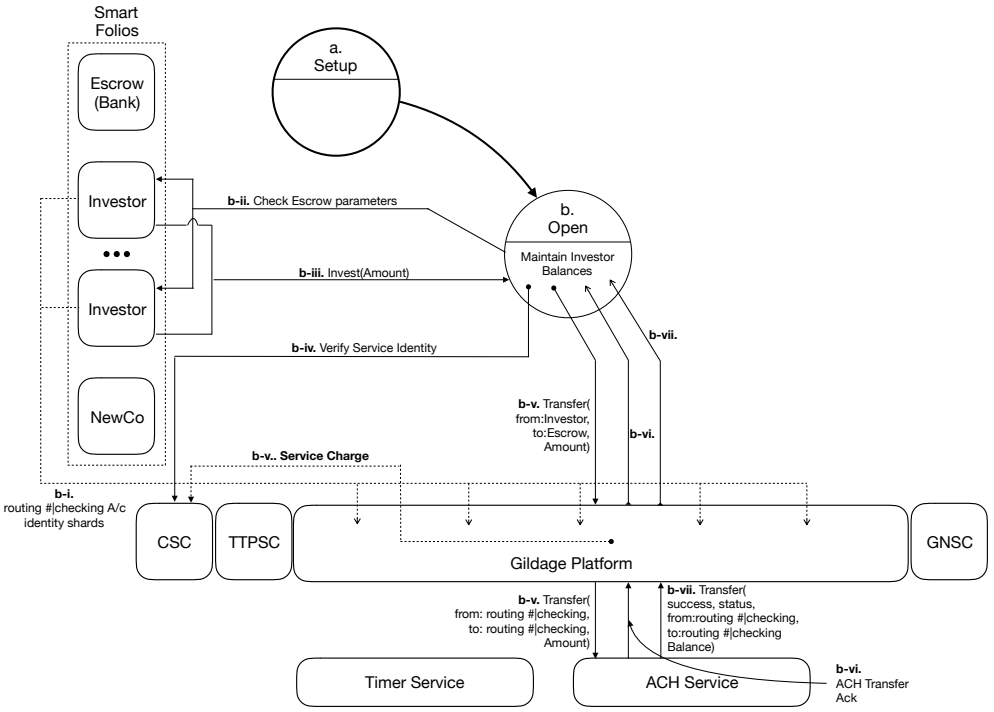

Figure 7: InvestorPool, Open State

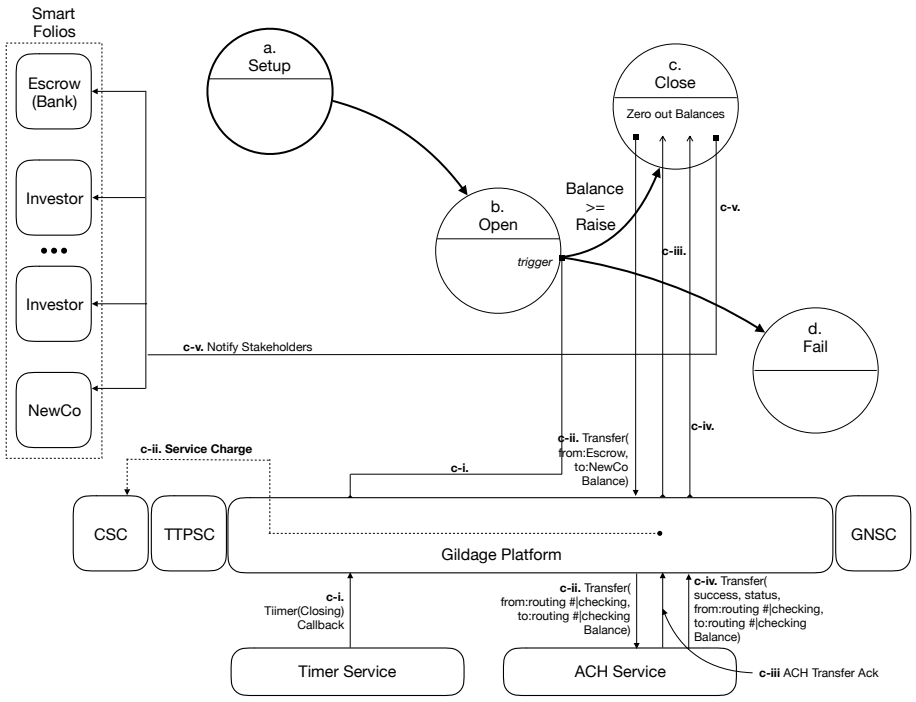

Figure 8: InvestorPool, Close State

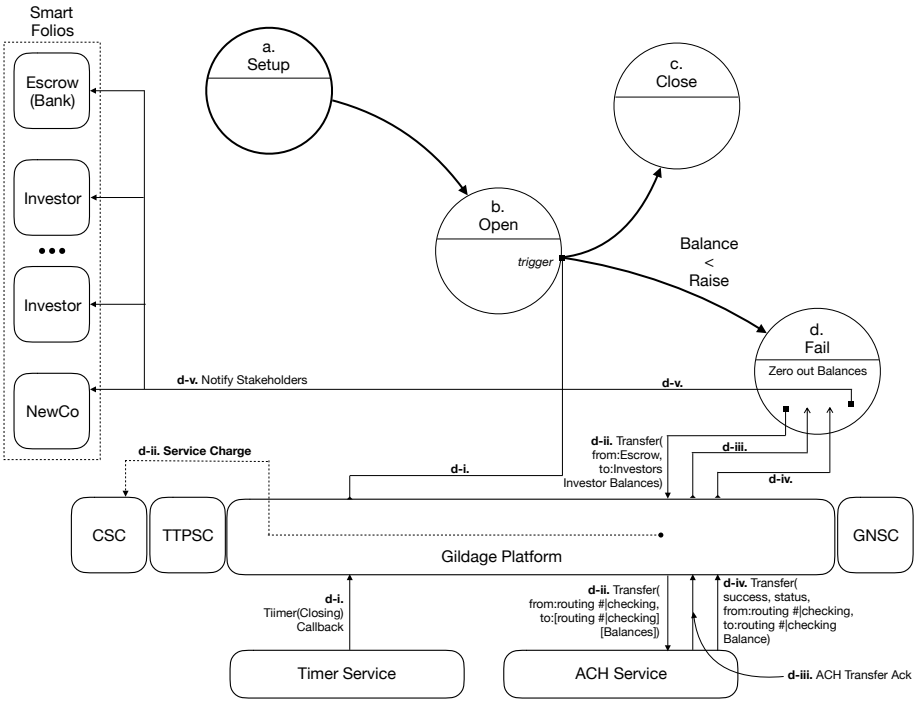

Figure 9: InvestorPool, Fail State

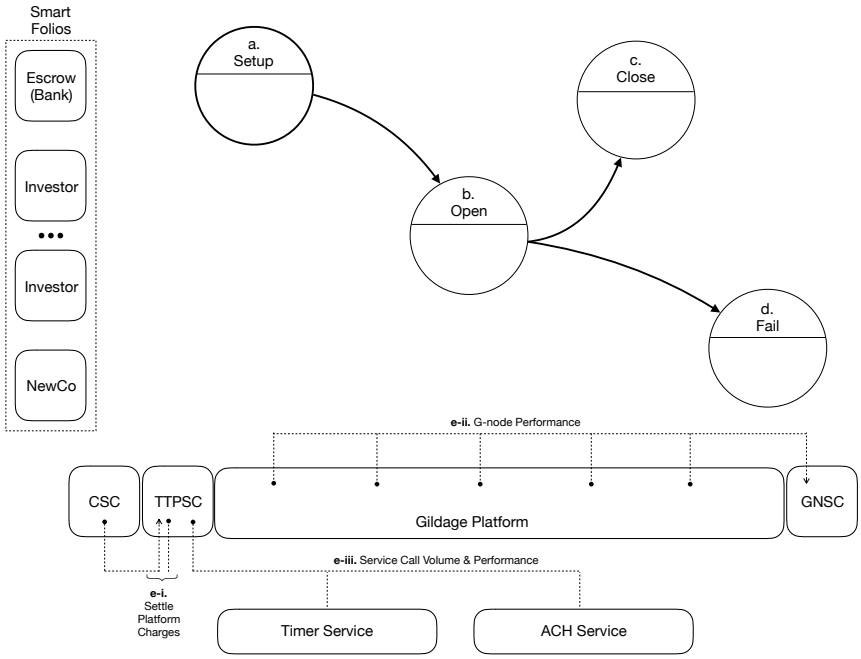

Figure 10: Gildage ecosystem Periodic Tasks

In addition to contract processing the ecosystem would perform the following additional functions that are also illustrated in figure 10.

- At regular intervals CSC and TTPSC settle escrow obligations by looking at how many messages have been handled.
- Periodic data related to all n g-nodes is recorded in GNSC.
- Periodic data related to T_{ACH}, T_{TIMER} and T_{EMAIL} is recorded in TPPSC.

While we have discussed InvestorPool smart contract in detail, the general mechanism applies to a wide variety of business logic and social situations. For example, in a social networking context, customers could ask a TTP to undertake the task of identity verification using conventional means, such as government issued identification, without having to reveal their paired network ID. Another example would be for in-game purchases. Customers could channel micropayments on behalf of their game identity via a smart contract wallet without having to reveal their network ID to the game developer.

In an enterprise setting for a permissioned blockchain, individual g-nodes could be under the control of co-signers and threshold for Shamir secret sharing can be set based on corporate governance policies. Another scenario could have g-nodes explicitly co-insure external requests issued by a smart contract, with risk and commission being distributed based on the

number of g-nodes that agree to sign onto the request. For example, imagine a supply chain contract that orders parts from a supplier. G-nodes that sign the purchase request and forward to the supplier would also be responsible for insuring against delays and product quality.

6. Infrastructure & Incubation Liquidity

The Gildage platform is fueled by utility tokens called *Gilds*. Just like public blockchain miners, g-node operators charge for message handling and storing identity shards. However, Gildage relies on Proof of Stake (PoS) instead of Proof of Work (PoW) and requires g-node operators to deposit Gilds with GNSC before they are added to the GNT. So, percentage of Gilds must be set aside for these PoS deposits.

Gilds will initially be distributed as follows:

Assignee	**Share**	**Notes**
Platform Founders & Development Team	25%	Compensation for platform creation and upkeep
Investors	25%	Private placement to inject initial liquidity
G-node operators	20%	Reserved for PoS deposit. Purchased at market rate by operators
Service Liquidity	20%	Utility tokens in circulation
Incubation Liquidity	10%	Invest in smart contract startup projects via token exchange

New Gilds can only be generated if Gild amounts in TTPSC or CSC escrows fall below a defined threshold due to increase in message traffic. All new Gild issuance must be done in proportion to the shares listed above and based on current Gild holders. This ensures that original platform stakeholders benefit from the platform's continued growth even as token liquidity in TTPSC and CSC escrows is restored.

Incubation liquidity requires some additional discussion. We want to encourage smart contract projects and services to adopt Gildage. In addition to providing a well-functioning platform, we invest in startup projects via token exchanges. Gilds assigned to incubation and any tokens they invest in are held in a separate smart contract with its own governance rules (GISC) that allows all outstanding Gilds to vote. This consensus mechanism allows all ecosystem stake holders to have a say in investment

decisions and establish an exchange price for incubation investments. Third party tokens are be gradually liquidated on a predetermined timetable and used to repurchase Gilds for future investments.

7. Conclusion

After surveying existing oracle platforms, we propose a novel approach for handling smart contract externalities called Gildage. We discussed how it ensures message authenticity and secure handling of identity pairs using a distributed architecture and Shamir's shared secrets. We provided examples and details on how they operate within the Gildage ecosystem. We covered payment mechanics that allow g-nodes to earn metered charges from client contracts and TTPs thus making the platform self-sufficient. We outlined how Gild utility tokens will provide necessary liquidity to build the platform infrastructure, set aside PoS reserves for g-node operators and incubate smart contract startups that choose to build on the platform.

In keeping with the spirit of the best public blockchain projects, our distributed architecture has no central authority and operates with consensus based governance. This means that the Gildage platform can truly be setup as a shared infrastructure for on-chain—off-chain integration in all conceivable smart contract and service provider scenarios.

We believe this is just a nascent beginning given the broad applicability of smart contracts. Our approach is pragmatic and keeps its focus on consumer realities, flexible business relationships, and innovative payment mechanisms rather than taking a singular approach to disintermediation and trustless networks. In most social contexts we routinely rely on and benefit from trusted intermediaries such as healthcare services, financial institutions, and government entities. Therefore, leveraging both the strengths of trustless networks and trusted third party services is ultimately in the best interest of the end customer.

References

[BUTE 2013] V. Buterin et al. Ethereum white paper.

[BUTE 2014] V. Buterin. "On Stake".

[DAME 1999] J. Damen, V. Rijmen. "AES Proposal: Rijndael"

[ELLI 2017] S. Ellis, A. Jules, S. Nazarov. "ChainLink: A Decentralized Oracle Network".

[FLOO 2017] M. Flood, O. Goodenough. "Contract as Automaton: The Computational Representation of Financial Agreements". Working Paper, Office of Financial Research.

[GDPR 2016] GDPR Portal.

[HASS 2016] W. Hasselbring. "Microservices for Scalability". Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, pp. 133-134.

[KÖPP 2018] Martin Köppelmann et al. Gnosis white paper.

[KROM 2016] K. Krombholz, A. Judmayer, M. Gusenbauer, E. Weippl. "The Other Side of the Coin: User Experiences with Bitcoin Security and Privacy". Proceedings of the Financial Cryptography and Data Security Conference 2016.

[LARI 2018] D. Larimer et al. EOS.io white paper (v2).

[MMSK 2018] Kumavis, et al. 2018. MetaMask website.

[NAKA 2008] S. Nakamoto. "Bitcoin: A peer-to-peer electronic cash system".

[OCHN 2017] Oraclechain white paper.

[ORAC 2017] Oracalize.it documentation.

[PETE 2018] J. Peterson et al. 2018. “Augur: A Decentralized Oracle and Prediction Market Platform”.

[SHAM 1979] A. Shamir, Magazine. “How to share a secret”. Communications of the ACM CACM, Volume 22 Issue 11, pp. 612-613.

[SOLI 2017] Solidity Documentation.

[SZAB 1997] N. Szabo. “Securing Relationships on Public Networks”. First Monday Journal, Volume 2, Number 9.

[TLSN 2014] “TLSNotary - a mechanism for independently audited https sessions”.

[WALL 2018] Wikipedia contributors. Cryptocurrency wallet. Wikipedia, The Free Encyclopedia. Retrieved April 28, 2018.

[WEIS 1991] M. Weiser. "The Computer for the Twenty-First Century," Scientific American, pp. 94-10.

[ZANG 2016] F. Zang et al. 2016. “Town Crier: An Authenticated Data Feed for Smart Contracts”. Proceedings of the 2016 ACM SIGSAC Conference, pp. 270-282.

Appendix A: InvestorPool Smart Contract

Code annotated based on message flow from figures (6), (7), (8) & (9). Some necessary error checking left out for brevity.

```
contract InvestorPool is GildageContract {
    enum Stages {
        Setup,
        Open,
        Close,
        Fail
    }
    Stages public stage = Stages.Setup;

    // (b-ii) public escrow information that can be inspected by investors
    public address bank, newco;
    public uint raiseAmount, minAmount, closing, currentBalance;
    public mapping(address => uint) investorBalance;
    // private
    mapping(uint => address) investorIndex;
    uint investorCount;
    address timerService, ACHService;

    // Events will be picked up by g-nodes
    event SetTimer(address _service, uint _time);
    event RequestACHTransfer(address _service, address _from, address _to, uint amount);

    // (a-i) Constructor
    InvestorPool(address _newco, uint _raiseAmount, uint _minAmount, uint _duration) public {
        // (a-ii) Setup escrow parameters
        bank = msg.sender; // contract owner
        newco = _newco; // entity raising investment
        raiseAmount = _raiseAmount;
        minAmount = _minAmount; // minimum required to invest
        closing = now + _duration; // in days
        currentBalance = 0;
        investorCount = 0;

        // (a-iii) Bind Services
        timerService = GildageContract.bindService("TIMER___"); // Service handles are 8 chars
        ACHService = GildageContract.bindService("ACH_XFER");

        SetTimer(timerService, closing); // (a-v)
    }

    modifier atStage(Stages _stage) {
        require(
```

```
        stage == _stage,
        "Invalid state"
    );
    _;
}

modifier fromService(address boundService) {
    require(
        _service == boundService,
        "Invalid service",
    )
}

// g-nodes call function based on event type specified by TTP and pass necessary attributes
// a-vi. SetTimer Ack
function timerSet(uint _eid, address _service, _closing)
public atStage(Stages.Setup) fromService(timerService) {
    require(_closing == closing);
    if (GildageContract.verifyResponse(_eid, _service, _closing)) {
        stage = Stages.Open; // trigger state change
    } else {
        // Require threshold number of responses to accept timer ack.
        // Contract can specify threshold to balance cost with reliability.
    }
}

// (b-iii)
function invest(uint _amount) public atStage(Stages.Open) {
    require(_amount > minAmount);
    // (b-iv) Ensure service identity info provided in smart folio
    GildageContract.verifyIdentity(msg.sender);
    RequestACHTransfer(ACHService, msg.sender, bank, _amount); // (b-v)
}

// (c-i) & (d-i)
function timerCallback(uint _eid, address _service)
public atStage(Stages.Open) from(timerService) {
    if (GildageContract.verifyResponse(uint _eid, address _service)) {
        if (currentBalance >= raiseAmount) {
            stage = Stages.Closed; // trigger state change
            // (c-ii) pay out NewCo
            RequestACHTransfer(ACHService, bank, newco, currentBalance);
        } else {
            stage = Stages.Failed; // trigger state change
            for(uint i=0; i < investorCount; i++) { // loop over investors
                address investor = investorIndex[i];
                // (d-ii) payback investors
                RequestACHTransfer(ACHService, bank, investor, investorBalance[investor]);
```

```
            }
        }
}

// ACH transfer callback for (b-vii), (c-iv), (d-iv)
function transferComplete(uint _eid, address _service, bool success, string status,
                          uint256 _fromKeySecret, address _fromEncrypted,
                          uint256 _toKeySecret, address _toEncrypted,
                          uint _amount) public from(ACHService) {
    address[2] resolveAddress =
        GildageContract.verifyResponse(_eid, _service, success, status,
                                       _fromKeySecret, _fromEncrypted,
                                       _toKeySecret, _toEncrypted,
                                       _amount);
    if (resolveAddress) { // only set once threshold responses have been received
        address from = resolveAddress[0];
        address to = resolveAddress[1];
        assert(stage == Stages.Open || stage == Stages.Close || stage == Stages.Fail);
        if (stage == Stages.Open) { // (b-vii) funding from investor
            require(to == bank);
            if (_success) {
                if (!investorBalance[from]) {  // register new investor
                    investorIndex[investorCount] = from;
                    investorCount++;
                }
                investorBalance[from] += _amount;
                currentBalance += amount;
            } else {
                // Use email or texting service to notify investor of ACH failure
            }
        } else if (stage == Stages.Close) { // (c-iv) pay our NewCo
            require(from == bank);
            require(to == newco);
            if (_success) {
                require(currentBalance >= _amount);
                currentBalance -= _amount; // zero balance
                // Maintain investor balances as immutable record
                // (c-v) Use email/text service to notify all stakeholders of completed round
            } else {
                // Use email/text service to notify Bank of ACH failure
            }
        } else if (stage == Stages.Fail) { // (d-iv) payback investors
            require(from == bank);
            require(investorBalance[to].isValue == true);
            if (_success) {
                require(investorBalance[to] >= _amount);
                investorBalance[to] -= _amount;
                currentBalance -= _amount;
```

```
                    if (currentBalance <= 0) { // zero balance
                        // All investors have been paid back
                        // (d-v) Use email/text service to notify
                        // all stakeholders of failed round
                    }
                } else {
                    // Use email/text service to notify Bank of ACH failure
                }
            }
        }
    }
}
```