Received: 30 October 2018 | Revised: 4 February 2019 | Accepted: 7 February 2019
DOI: 10.1002/cpe.5228

WILEY

SPECIAL ISSUE PAPER

Analyzing smart contract interactions and contract level state consensus

Yao-Chieh Hu[1] | Ting-Ting Lee[1] | Dimitris Chatzopoulos[1] | Pan Hui[1,2]

[1]Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Sai Kung, Hong Kong
[2]Computer Science Department, University of Helsinki, Helsinki, Finland

Correspondence
Yao-Chieh Hu, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong.
Email: yhuag@connect.ust.hk

Present Address
Yao-Chieh Hu, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong.

Summary

Although the primary function of distributed ledgers is to store data related to users' interactions, their capabilities allow them to offer more sophisticated functionalities. Advances in blockchain technologies introduced *smart contracts*, software programs that define immutable rules as functions stored on the blockchain and can be executed on demand. Smart contracts can interact not only with users but also with each other via message exchange. We compare existing smart contract interactions, and develop an architecture for asynchronous state consensus, a novel type of smart contract interaction required in applications but had rarely been addressed. The proposed architecture is composed of two types of smart contracts, ie, Custodian and Client. Client smart contracts serve as network participants reaching a particular consensus collectively by forming a cluster and issuing votes towards a final state agreement. Custodian smart contracts serve as the arbiters that aggregate and calculate voting results as the finalized state consensus that is shared across the network. To test the feasibility of our proposal, we conduct experiments on the consensus reaching latency and the scalability under different network configurations with standardized Amazon Web Service instances. Lastly, we discuss the robustness our proposal concerning Byzantine Fault tolerance and list possible applications. In the gaming industry, an ERC721 smart contract does not allow contrasting structural features between individual tokens, yet only minor value-level differences. The proposed solution can address the need for character diversity that characters can be created and attached to a gaming smart contract after deployment, which enables fine distinction between characters. The proposal can also achieve sharing states across smart contracts, such as the jackpot, which renovates the flexibility of blockchain gaming.

KEYWORDS

Byzantine fault tolerance, decentralized Ledgers, distributed consensus, smart contracts

1 | INTRODUCTION

The advent of Bitcoin, by Satoshi Nakamoto in 2008, introduced blockchain technologies.[1] Each native blockchain employs consensus algorithms and incentive mechanisms to maintain a decentralized ledger storing transactions between its users in the network. The decentralized ledgers are state machines that apply ordering transactions to move from one state to another. By the adoption of hash pointers and Merkle trees,[2] the ledger is guaranteed to keep its transactions from being tampered once they were published to the network. Its novelties lie in the trustless relationship between all network participants for reaching a concluded state of the shared ledger, which is determined by a distributed consensus endorsed by cryptographic signatures. The blockchain developed for Bitcoin is specifically designed to store monetary exchange transactions, which offers additional scripting capabilities that create the potential of programmable money. However, the scripting capacity of Bitcoin is limited to only a few practical applications, such as colored coins.[3] Ethereum[4] was the first proposal that provided an open blockchain platform that allows anyone to build and use decentralized applications (DApps). It employs a virtual machine called *Ethereum Virtual Machine* (EVM) that empowers a runtime environment for programs with general purpose computations. Programs are named *smart contracts*, written in Turing Complete programming languages. Examples of programming languages for the development of smart contracts are *Solidity*, Serpent, Viper, Chain, and others. Solidity is the most popular language and the one we use in this work.

Although blockchain has a well-defined mechanism for reaching consensus across multiple network participants for the shared ledger, the smart contracts themselves are deficient in the consensus architecture. Smart contracts are inclined to be governed by centralized parties, and therefore, they could be manipulated autocratically to hinder accessibility and data integrity. In particular, in current implementations of decentralized applications that utilize smart contracts, the fairness of the system is still heavily dependent on a few centralized identities even if the original design of smart contracts is to employ decentralization. The threat of centralization raises the issue of data authentication on smart contracts, which is aggravated when the data is contingent on a single external source of input. This leads to a more immediate concern since the data stored on the blockchain is often considered as accurate historical records to be shared among millions of participants in the network. The problem described above is known as *The Oracle problem* since the smart contract dedicated to external data importation is named as the *Oracle*. The Oracle problem has been actively debated among Bitcoin developers since 2015.[5] Centralization also exist in the token economy since tokens in Ethereum can be developed through smart contracts to adapt to various applications, and one single smart contract often has control over all its tokens. Compromising the particular smart contract can lead to critical loss, which can be as significant as the total value of the corresponding token.

The problem of centralization can be solved by leveraging the wisdom of crowds, where a *consensus* determined by the majority of the participants can be shared as a universal value across the network. The consensus can be the triggering condition of specific functions in smart contracts. In this design, a group of smart contracts can form a decentralized autonomous organization where they interact with each other based on the shared value determined collectively. Consensus between multiple participants can be reached through asynchronous voting. Information providers vote for the value they collect on a given topic of consensus at any time; an information aggregator calculates and finalizes the voting result once the number of votes is sufficient in comparison with total network participants. This mechanism demonstrates the built-in benefit of preventing a few participants acting maliciously or being unreliable from jeopardizing the network. Traditionally, a particular individual plays the role of the information aggregator. In contrast, this paper proposes a mechanism that assigns the role of the information aggregator to a smart contract, which is ownerless and self-managed. Privileges to specific functionalities such as intervening the voting campaign or mutating the finalized voting result will no longer exist in the smart contract implementation. Moreover, consensuses are updatable over time and have their *states* based on the time they are finalized. The methodology to reach the *state consensus* uncompromisingly and efficiently is then the primary focus to be discussed further in this work.

We develop two types of smart contracts, ie, *Custodian* contract as the information aggregator and *Client* contract as the information provider. For every consensus, voting campaigns can be finalized at different timestamps. Each voting campaign is assigned a corresponding sequence number, calculated by the hash of the previous voting campaign sequence and its finalized consensus. The Custodian contract keeps track of all the previously finalized consensuses, while the Client contract keeps only the latest voting campaign sequence and the most recently finalized consensus.

To summarize, this paper proposes an asynchronous state consensus reaching mechanism where multiple smart contracts with different sources of information can reach consensuses and share it as a common variable. This paper also extends the evaluation techniques and environmental configurations employed by a previous work of Hu et al[6] addressing hierarchical interactions between smart contracts and the corresponding performances across a few Ethereum testing networks.

Contributions

1. We present the first experimental exploration in the research space of smart contracts interaction to the best of our knowledge.
2. We propose a conceptual architecture of smart contracts reaching consensus with each other within the network in an asynchronous manner.
3. We develop two types of smart contracts and their interfaces using the Solidity programming language to realize the proposed architecture.
4. We deploy the developed contracts on Amazon Web Service instances in order to evaluate the practicability of our mechanism by measuring the consensus reaching latency and the gas usage of smart contracts execution.

The rest of this paper is organized as follows. In Section 2, we discuss the related work as a comparative analysis of existing smart contract interactions. In Section 3, we list the existing types of interactions between smart contracts. In Section 4, we introduce the oracle problem. In Section 5, we present the proposed architecture. In Section 6, we explain the findings of our measurements. In Section 7, we discuss our designs and concerns, and in Section 8, we conclude this paper.

2 | RELATED WORK

Significant efforts have been put in the development of decentralized ledger networks that enable Turing-complete smart contract executions and standards, which include the omnipotence of Ethereum network,[4,7] legal foundations of Ricardian contract,[8] and consensus and scalability trade-off negotiation.[9] However, few have addressed the interactions between two and more smart contracts. A multitude of decentralized applications have leveraged the characteristics of smart contracts to showcase utilizations in practice. A privacy-advocated voting scheme designed by McCorry et al escalates the secrecy level with a self-tallying protocol on Ethereum smart contract.[10] Trust degradation and integrity breach on the clinical trial data can be warded off by smart contract enforcement as addressed by Nugent et al when adopting immutability on the historical data.[11] In the 0x protocol, smart contracts facilitate low friction trading of *ERC20*[*] tokens to empower an autonomous decentralized exchange.[12] Basic Attention Token solves the middleman problem of digital advertisement in a smart contract based approach. Privacy violations can be prevented by utilizing the transparency and tamper-resistant characteristics of smart contracts on the blockchain.[13] CryptoKitties, transforming the blockchain technology in the form of games, designs smart contracts for its token as traditional collectibles of ownership for each token is introduced, and the sales auction mechanism also presents a new concept of automating the token selling process.[14] Status, an Ethereum-enabled mobile client, fuels the peer-to-peer social network with a standardized utility token that revolves around a deployed smart contract.[15] To extract deeper insights, the following section summarizes smart contract interactions into various categories and presents a comparative analysis. Then, a survey of existing approaches to reach state consensus from external data at smart contract level is discussed.

3 | TYPES OF SMART CONTRACT INTERACTIONS

3.1 | Modularization

Smart contracts prone to rely on a few modules of features with segregated logic when growing to a certain scale of complexity. A reusable structure saves gas consumption when issuing transactions on the Ethereum network and reduces the repeatability of code by rendering modules for dedicated functions. Common methods to achieve such functionality includes *gating*, *multiple inheritance*, and *library*.[†] With the modular approach, additional characteristics can be introduced to the smart contract logic to achieve flexible extension.

Gating: Gating is to create internal switches inside a core smart contract to enable the possibilities of switching on and off specific functionalities or traits after contract deployment. The core contract stows all the modularized traits, default to be switched off, which can be called externally to trigger on. The comparative advantage over traditional smart contracts convention is that all functionalities are accessible to be enabled and disabled on the fly, regardless of the contract deployment time. The shortage includes that the core contract can consume an excessive amount of gas during deployment since all possible traits, indifferent to their usage frequency, have to be included when deploying.

Multiple Inheritance: Multiple inheritance allows smart contracts to inherit codes instead of states, from one or multiple parent contracts. One possible scheme is to inherit functionalities from single-purpose parent contracts to extend modularization potential of smart contracts. The inheritance relationship is specified before deployment, and thus prevents further changes after deployment on module addition or removal intentions. In addition, modularized multiple inheritance brings the benefits of enhanced readability on smart contracts' code as modules are pluggable. The major limitation, however, is that no accessibility and functionalities can be updated after the deployment of the smart contract.

Library: These smart contracts can be organized and modularized into various user-defined libraries, in which the library stores all the modules and groups them according to their major application domain. For instance, *SafeMath*, one of the most well-known libraries in Ethereum devised by OpenZeppelin,[16] provides a sample implementation of the most commonly used arithmetic operations, which prevents unexpected behaviors by checking for underflows and overflows when performing calculations.

3.2 | External function call by function signature

In the context of utilizing solidity library calls, it is necessary for the caller contract to be aware of the exact function signature of the callee contract. This reduces the flexibility of caller contracts by rooting out the possibility for callee contracts to be deployed after the deployment of the caller contract, and thus freezes the order of the caller-callee deployment. To address this issue, two primitive functions in the Solidity language, `call()` and `delegate call()`, decouple the caller and callee by enabling external function calls given the desired function signatures, and make the fixed order of deployment unnecessary.

`call()`: The exact step to call a target function externally, from a separate smart contract that is not the implementer of the target function, is to first obtain the callee's contract address and the signature of the target function. In this case, the callee contract needs to be instantiated before its function call. As expected, the data storage of the callee smart contract can be modified accordingly, while the caller's data storage remains unchanged.

`delegate call()`: There is one major difference between a call and a delegate call. A call can affect the storage of the callee contract but not the storage of the caller contract. A delegate call, on the other hand, can only affect the storage of the caller contract. To

[*]The ERC20 token standard describes the functions and events that an Ethereum token contract has to implement.
[†]Reference code is provided here: https://github.com/BlockChain-UROP/smart-contract-interactions

witness the differences between the two operations, smart contracts have to define a variable with the same data type and name. For instance, if a caller contract A calls a function of contract B that changes a variable, `ans`, when both contracts have that variable defined beforehand, only A contract's variable `ans` will be modified.

3.3 | Interface

Interface, as its name suggests, is a device or program for connecting two items of hardware or software so that they can be operated jointly or communicate with each other. Smart contracts can also have their own interfaces, which contain full specification of the variables and function signatures. As more and more interfaces have been developed and people start to be curious about the actual interface implementing contract code and its smart contract deployment address, ERC820[17] was proposed as a standard that defines a universal registry smart contract where any address, either it is a smart contract or a regular account, can register the interface it implements and the smart contract responsible for its implementation. In addition, in order to ensure consistency for the interface contract address, this standard leverages the Elliptic Curve Digital Signature Algorithm, ECDSA, signature properties to produce fixed contract address across various kinds of blockchain networks given predefined variables in the contract deployment raw transaction.

3.4 | Operator

Corresponding to the design of the Ethereum ERC777 protocol,[18] operator is defined as a particular address, either a smart contract or a normal account, which is authorized to manage the funds of one or more other accounts. The advantage is to combine the conventional *approve* and *transferFrom* methods into a single *transfer* with an arbitrary amount of balance for the authorized operator. Specifically, by executing *approve* and *transferFrom*, each approval can only allow a fixed amount of balances to be transferred. On the contrary, by the authorization granted for operators in ERC777, an address can gain permanent or temporary control over one or more accounts to manage their balances. The design is especially useful when users want a cryptocurrency exchange or a fund manager to take over their account for a certain time span. Once approved, the user can be free from being asked to approve every single transfer transaction.

3.5 | Upgradeable smart contract

Although blockchain is immutable in terms of each broadcasted transaction and thus allows consensus to be reached for every node in the network, the biggest disadvantage is that the source code of smart contracts after being deployed cannot be changed. ZeppelinOS[19] has come to rescue as one of the most comprehensive upgradeable smart contract architecture in current blockchain space. In specific, it introduces a new "proxy" smart contract that manages the implementation of the original smart contract and its updated versions. Upon each function call, the proxy smart contract utilizes *delegate call* to execute the function code by referencing the contract implementation of the latest updated version. This approach keeps the contract address of the proxy smart contract the same and allows the contract executor to perform the desired operation without the hectic of re-instantiating a new smart contract upon every update.

3.6 | Oracle smart contract

In this work, we differ by the presented literature and interactions discussed above by focusing on the asynchronous state consensus reaching among smart contracts. The existing solutions and how it is different from our proposed architecture is illustrated in the following section.

4 | FROM THE ORACLE PROBLEM TO STATE CONSENSUS

4.1 | Oracle Client smart contract

Client contracts are allowed to be any form that equipped with any possible functionality. One special type of Client contract is the Oracle Client, which aims at absorbing external information from the outside world to bridge up the connection between Ethereum and real-world information. The use case can range from IoT devices and sensors that continuously collect data for landslide detection to multiple mobile devices that collaboratively work to locate a peer's exact location. A famous problem that remains unsolvable to date is the Oracle Problem. The Oracle Problem states that, in order to improve the efficiency of data collection via oracle, the system requires to rely on single source of input, which jeopardizes decentralization of the blockchain.

4.2 | The Oracle Problem

The Oracle Problem is the dilemma that when retrieving external data from the outside world, there is a choice to made between efficiency and decentralization. Because of the fact that smart contracts are locked in the blockchain without an access to the external data, oracle is designed to be the IO gateway that can collect data from outside. It is usual for the smart contracts to rely on a single oracle as the channel of information gathering, but the problem is that the architecture will be in ruin if the only oracle turns out to be malicious or faulty. There seemed to be no easier way but only to use several oracles simultaneously, whereas it is very time expensive for an asynchronous external inputs IO to reach on an agreement and forward the information. The trade-off of the Oracle Problem is between the efficiency and the decentralized security, to sacrifice one for another.

As for the actual negative impact of the oracle problem on real-life applications, decentralized exchange can be one of the cases. Since the smart contract of the decentralized exchange requires accurate exchange rate from the market, relying on a single source of centralized exchange of the outside can create bias on the rate. In the case that the centralized exchange modulates the price, the decentralized exchange will subsequently be influenced by the artificial interference. Similar effect can take place on smart contracts that have major dependence on the accuracy of the external data, which can be led to digesting bias results and put the entire system in jeopardy.

4.3 | State consensus

This paper solved the problem by designating each of the Clients as the oracle who collect data from an external source. The agreement can be reached within a time-off duration to preserve the liveness of the network. More importantly, the Client oracles can do more than collect, agree on, and forward the data. By reaching the consensus in a distributed way, the oracle can share common states that originate from their decisions and votes. Therefore, the Oracle Problem is solved by the distributed multiplex state consensus, which can further bolster the network for the possibility of common state sharing.

4.4 | Existing solutions

Present solutions that tackle the Oracle Problem include Augur,[20] Astraea,[21] Oraclize,[22] and ChainLink.[23] However, these approaches have the following limitations when it comes to practicability.

Augur creates an oracle for participants to stake coin on a predictive market at risk, in the expectation of a reward payoff when the provided value is a correct prediction, determined by the majority. Participants are free to decide the amount of coins staked with their provided data. Although the monetary staking mechanism allows the network to be intrinsically immune to Sybil attacks,[24] this mechanism may lead to a situation that the participants only report the value of the external data the same as the majority in order to claim the reward. In addition, the participants are obligated to keep reporting values in the network to avoid the penalty of inactiveness, which downgrades the participation freedom and involves probable malevolent attempts during the dispute period.

Astraea addresses a general-purpose decentralized oracle to reduce the complexity and abridge the protracted procedures that used to be employed in agreement reaching between multiple sources of knowledge. A polarization of propositions could develop when malignant submitter perennially presents biased propositions, breaking the Nash equilibrium,[25] which preserves both clans of the voters and certifiers to remain truthful informative.

Oraclize adopts the TLSNotary[26] proof that serves cryptographically verifiable information by associating the authentication to the Transport Layer Security (TLS) associated with the Hypertext Transfer Protocol Secure (HTTPS) protocol. A single-point-of-failure may form due to that the content of the website can be tampered and the integrity no longer holds to guarantee the correctness of the data brought in the blockchain.

ChainLink solves the problem of single-point-of-failure by realizing the concept of decentralized oracle, which collects data from multiple sources instead of one. However, it has limited functionalities to interact with the blockchain other than providing data to the aggregator oracle smart contract, since the data collector in this case is a node run by traditional servers.

4.5 | Differences with proposed solution

In summary, to avoid having to rely on any third party to maintain the validity and integrity of the external data and to allow more complex logic for the external data collector other than providing data, this work proposes a solution that uses smart contracts as the external data collector, which will be explained in detail in the following section.

5 | SYSTEM ARCHITECTURE

Consensus between multiple participants with disparate information in a network can be reached by mechanisms such as voting. Two types of smart contracts, Custodian contract and Client contract, are devised for this mechanism, in which the Custodian contract plays the role of a

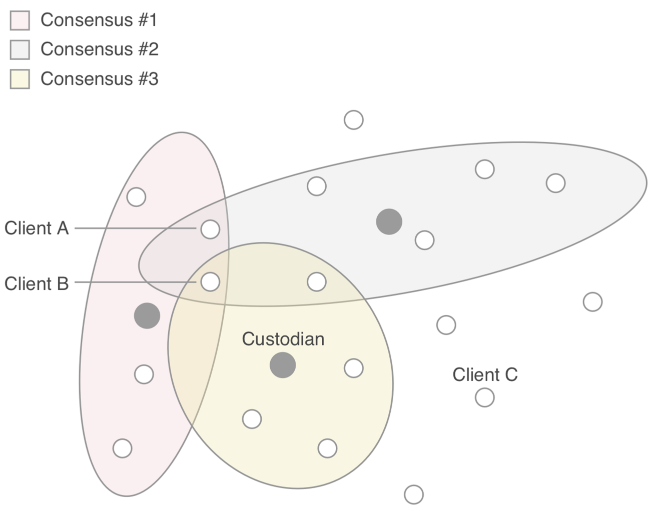

FIGURE 1 Multiple consensus reached across clusters of clients

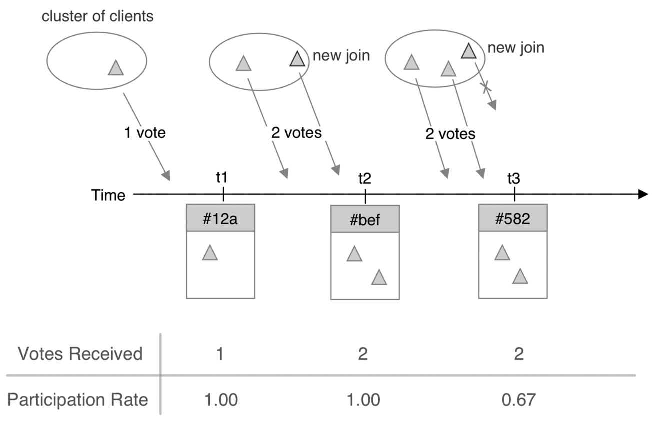

FIGURE 2 Timeline of a voting campaign

passive information aggregator while the Client contracts are the sources of information. This passive mechanism stands in contrast to some existing information aggregators who aggressively absorb data from distributed sources. In specific, a group of information providers want to reach an agreement on a particular topic. Rather than direct communication in pairs, as in a peer-to-peer network, they confide their piece of information to an individual information aggregator that helps finalize the outcome, which is prone to corroborate the majority.

This paper designates Client contracts as information providers, and Custodian contracts as information aggregators. For each topic on the way of agreement reaching, a group of Client contracts constitute a *cluster* and are able to send information across to a dedicated Custodian contract as votes. Custodian contracts then aggregate the votes and evaluate the outcome, without any possibility of tampering and manipulation externally (from outside of the contract). Figure 1 depicts the network where multiple consensuses can be reached among each devoted cluster of Client contracts, with a number of associated Custodian contracts. It is demonstrated that a Client contract can take part in voting campaigns across different consensuses. For instance, Client A votes for two distinct consensuses, while Client B votes for three. Likewise, information providers can choose not to participate in the voting campaign, such as the case of Client C.

In reaching a consensus by a distributed voting mechanism, the voting operation of each information provider must be well-defined and unified to be aggregated hazardlessly by the information collector. Sequential stages of the consensus reached must also be preserved to ensure the correctness of the final state and its history. *Voters*, namely, the information providers, have the freedom to choose whether to participate in a particular voting campaign or not. The final consensus reached would be publicly visible to all the network participants including the non-voters. A typical consensus-reaching voting activity can be illustrated as follows: A Custodian contract is created whenever a new topic of consensus is expecting to be reached. When a Client contract first issues a vote to the Custodian contract, it becomes a voter to the particular Custodian, and a voting campaign is launched subsequently. Each voting campaign is launched with a unique sequence number calculated by hashing the concatenation of previous voting campaign sequence and its finalized consensus. The generated sequence number for an individual voting campaign is fixed and read-only for both the Custodian and the Client contracts. Past voting result histories are recorded permanently and can be retrieved by the associated sequence number. This design further enables the verification on the current consensus based on past consensus history. A voting campaign is concluded when the participation rate reaches a system-defined threshold. The threshold mechanism demonstrates that the system is inherently Byzantine fault tolerant, and is immune to collaborative conspiratorial attempts across clans of offenders bounded by a third of the total participation. The network is presumed to accord with this assumption. As demonstrated in Figure 2, voting campaign ends

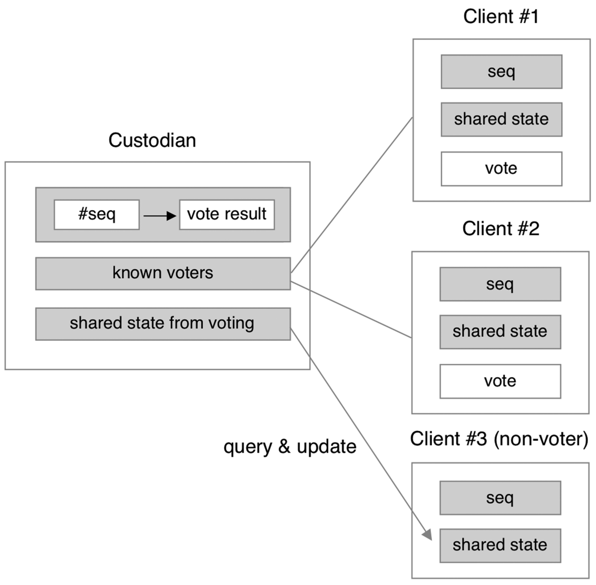

FIGURE 3 Relationship between the custodian and the client smart contracts

and a consensus is reached when the total number of votes received by the dedicated Custodian contract passes a predefined threshold of total voters. The sequence number for tracking each voting campaign is then recalculated and updated after the end of a voting campaign. This implies that, for each topic of consensus, there is only one active voting campaign at a time.

If there is another voter that wants to change the current consensus by issuing a new vote, a voting campaign will be launched again correspondingly and ended when there are sufficient amount of votes. Since there is no upper bound on the number of voting campaigns launched sequentially, the consensus can always be re-voted and updated dynamically by launching on demand voting campaigns when some voter has a different opinion on the currently agreed consensus. For a consensus-reaching Custodian contract that concludes a voting campaign by reaching a sufficient amount of votes over total voters, voters are known Client contracts that have issued a vote to the target Custodian contract. For each Custodian contract, a Client contract can either be unknown or known as a voter. By default, Client contracts are unknown voters to all Custodian contracts before the issuance of a vote. After the first vote has been issued, the Client contract is automatically registered as a known voter on the Custodian contract. Total number of unique voters is being tracked by the Custodian contract to determine when to finalize a voting campaign. Once there is enough number of votes, the Custodian contract aggregates all voters' vote and stores the calculated result as a finalized consensus. Client contracts, regardless of being voters or not, can then query the final state and perform predefined actions based on the resulting state. Each Client contract only has a single vote on each unique sequence of the voting campaign. A voter can be anyone that has inherited a Client contract interface at any given time, and custodian contracts can be created on demand when a new consensus is expected to be reached by a cluster of client contracts. There is no preassumption on which type of contract should be created first in order to reach a consensus. It is worth mentioning that the current design of the Client and the Custodian smart contracts does not guarantee the anonymity of the voters. Zero-knowledge proofs offer a solution to this problem but increase substantially the complexity of our proposal and the gas needs. The development of efficient Client and Custodian smart contracts that can guarantee voters' anonymity is part of our future work. Another point worth noting is that, in certain scenarios, one or multiple voters will not be authorized to participate in specific voting events; an authorization mechanism is therefore needed. This paper considers a membership list that can keep track of all the voters that are authorized to cast the vote in Section 7.3. A more comprehensive approach of authorization management is a future work.

In summary, as shown in Figure 3, Custodian contracts each represent one specific topic of consensus and store the history of voting results indexed by their sequence numbers; a list of known voters consists of Client contracts and the latest finalized state consensus. Client contracts preserve the latest voting campaign sequence number to query the last finalized consensus.

5.1 | Smart contract interface

Two smart contracts interfaces, named *ICustodian* and *IClient*, are developed to realize the conceptual architecture[‡]:

[‡] Reference Code Base: https://github.com/BlockChain-UROP/Smart-Contract-Hierarchical-Interactions-on-State-Consensus

```
contract ICustodian {

    uint8 public THRESHOLD_OF_PARTICIPANTS;

    uint256 public numOfTotalVoterClients;
    bytes32 public newly_opened_seq;
    bytes32 public last_finalized_seq;

    mapping (bytes32 => uint256) public votesCountOnCamp;
    mapping (bytes32 => uint8) public finalResultOnCamp;
    mapping (bytes32 => int) public currVotesBalanceOnCamp;
    mapping (address => bool) public clientIsKnown;
    mapping (address => mapping (bytes32 => bool)) clientHasVotedOnSeq;
    mapping (bytes32 => bool) public campHasFinished;

    function acceptVote(bytes32 _seq, bool _value) public;

    event VoteCampFinished(
        bytes32 seq,
        uint8 finalResult
    );
}
```

```
contract IClient {

    bytes32 public seq;

    function vote(address _custodianAddr, bool _value) public;
    function queryFinalState(address _custodianAddr) public view returns (uint8);
    function _syncToNewlyOpenedSeq(address _custodianAddr) private;
}
```

Algorithm 1 Accept votes and calculate voting result

```
1:  function AcceptVote(seq, val)
2:      $client \leftarrow$ Transaction Issuer
3:      if $voted(client, seq)$ then
4:          return false
5:      else
6:          $voted(client,seq) \leftarrow true$
7:      end if
8:      if not $clientIsKnown(client)$ then
9:          $clientIsKnown(client) \leftarrow true$
10:         $totalVoters \leftarrow totalVoters + 1$
11:     end if
12:     $votes(seq) \leftarrow votes(seq) + 1$
13:     if $value$ then
14:         $balances(seq) \leftarrow balances(seq) + 1$
15:     else
16:         $balances(seq) \leftarrow balances(seq) - 1$
17:     end if
18:     if $totalVotes(seq) >= THRESHOLD * votes(seq)$ then
19:         if $balances(seq) > 0$ then
20:             $result \leftarrow 1$
21:         else if $balances(seq) < 0$ then
22:             $result \leftarrow 2$
23:         else
24:             $result \leftarrow 0$
25:         end if
26:         $finished(seq) \leftarrow true$
27:         $lastSequence \leftarrow seq$
28:         $newSequence \leftarrow KECCAK256(seq, result)$
29:         return $result$
30:     end if
31: end function
```

The following section provides a more detailed definition of the two types of smart contracts.

5.1.1 | Custodian

A Custodian Smart Contract can perform the following four actions:

1. Receive and aggregate votes collected from Client contracts.
2. Generate sequence number for the newly launched voting campaign.
3. Maintain and update a list of known Clients upon receiving every vote.
4. Finalize a voting campaign once the number of votes reaches a predefined participation threshold.

5.1.2 | Client

A Client Smart Contract can perform the following three actions:

1. Cast a vote to a Custodian contract for reaching a consensus.
2. Query the lastly finalized voting result for a specific consensus given its Custodian contract deployment address.
3. Sync its voting campaign sequence number with a specific Custodian contract.

Client smart contracts can be further classified into two types, voter and participant, on whether a vote has been issued for the latest voting campaign or not.

6 | EXPERIMENTAL ANALYSIS

This paper addressed and provided an implementation of a minimum viable version of the Custodian-Client protocol to achieve contract-level asynchronous state consensus, with around 130 lines of smart contract codes in Solidity.[§] In this section, the experiments examine how the architecture reacts to the variation of the participation number, the consensus concurrency, and the alternation of the thresholds on participation, with respect to effective latency and scalability. Each of the experiment is repeated in the same environment for 20 trials. The standard deviations are demonstrated as a capped pole in every figure to quantify the variation of the data.

Three categories of experiments have been conducted. In the first experiment, this paper tests the resulted latency across various numbers of participation ranging from 10 to 300 Clients, as they are reaching a single agreement on a Custodian consensus. The second experiment verifies the latency for a fixed set of Clients reaching consensus concurrently across different numbers of Custodians, from 1 to 20 Custodians. The third experiment records the latency differences across variations of participation threshold, from 5% to 100%, when a fixed set of Clients are reaching a single consensus at a designated Custodian. The experiments should examine the effectiveness of the protocol in respect of the latency, as well as the scalability of the network with regard to the divergence of the numbers of Clients and Custodians, and the rate of the effectuation threshold.

6.1 | Setup

To provide a replicable examination on our protocol implementation, the experiments are conducted on Amazon Web Service EC2 instances. This paper adopts instances with the type *t2.large* with two virtual CPU core accommodating two processes and bandwidth of up to 500 Mbps. The experiments have displayed that the actual utilization of our protocol execution is only about 34.3 Kbps on average, given that each experiment occupies respectively around 45 Kbps, 26 Kbps, and 32 Kbps. All the experiments are attached to a ganache private blockchain[¶] running alongside, which simulates the actual Ethereum mainnet blockchain, yet with higher throughput and TPS (Transaction Per Second) to expedite the experiment progresses.

6.2 | Environmental limitations

Owing to the fact that the experiments are conducted on the private blockchain on the EC2 instance, the latency result from the experiments is not supposed to be replicable when the experiments are mirrored on the main Ethereum network and other testing networks. However, the result could indicate associations between factors and the latency when such a relationship is driven by the contract setup instead of the testing environment.

[§]Reference Code Base: https://github.com/BlockChain-UROP/Smart-Contract-Hierarchical-Interactions-on-State-Consensus
[¶]a tool from Truffle Suite to configure and initiate a local blockchain for Ethereum development

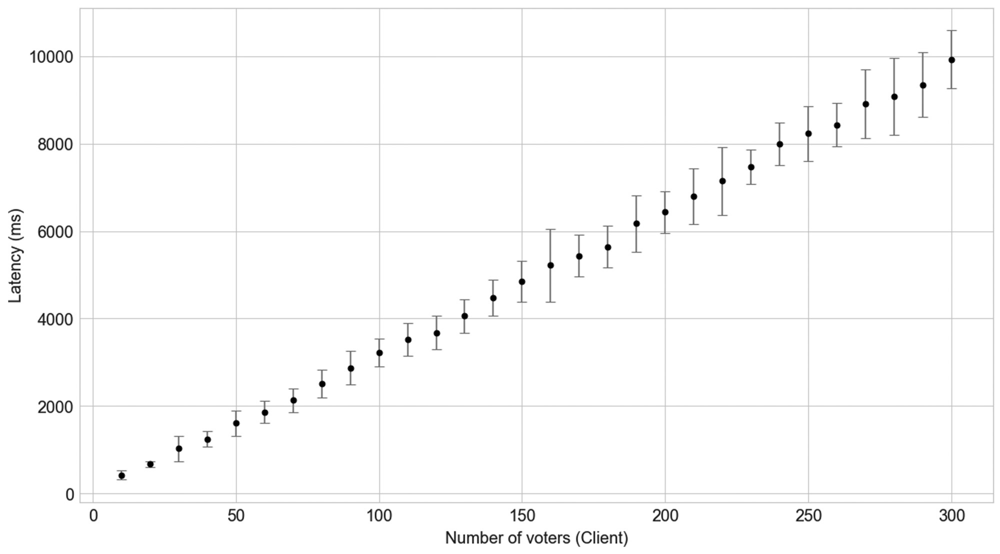

FIGURE 4 The latency test on various numbers of voters

6.3 | Measurements

Let N denote the size of one Client cluster, which is equivalent to the number of Clients reaching the same consensus at a designated Custodian contract. Let M denote the total number of consensuses to be reached at the same time, namely, the number of Custodian contracts. Let T denote the threshold that the participation rate needs to surpass so as to effectuate this particular campaign. For simplicity at the initialization of each experiment, all the network participants are configured as known-voters to the designated Custodian contract. It avoids the case that the voter base might be extended when unknown voters are joining in, which should stay constant across trials. The following experiment sections demonstrate the extent that the consensus latency can vary with respect to N, M, and T, while either two are fixed.

6.3.1 | Experiment 1: latency measurement on voter base sizes

The aim of experiment (1) is to discuss the relationship between the quantity of the voter bases, which is the number of Client contracts participated, and the resulted latency for an individual consensus to be reached upon. As shown in Figure 4, the x-axis is the total number of voters participated, and the y-axis indicates the latency. The solid dot stands as the average of all 20 trials of experiments, and the thin vertical lines depict the standard deviation of each set of trials at a particular amount of Client contracts participation.

In this experiment, the latency presents a stable linear growth as the voter base continuously increases by the step of 10. The standard deviation gradually increased as the latency goes up, yet still considered minor as the upward tendency of latency is discernible. The regression line of the data points has a slope of about 33, showcasing that, whenever a new Client has been added in the cluster to reach the next round of consensus, the latency has high chance to go up by a couple milliseconds. The inclination demonstrates that there is a minor but recognizable cost for each marginal participation of any Client, and can testify that the efficiency downgrading can take place, when the number of a cluster unstoppable grow to a number that is high enough to make the latency observable. Therefore, this paper suggests designs that can break down sophisticated consensuses into limited and modularized clusters of Clients, with primary consideration of performance and flexibility within the network.

The reason of the incremental inclination can be explained that, as the number of the Client number increased, the effectuation threshold rises up along with the voter base. This says that a particular Custodian contract is forced to wait longer until the total votes have reached the line of the threshold; otherwise, this campaign can never be effectuated. Specifically, when the Client number is as less as 10 voters, the designated Custodian must wait for 6 votes, as the threshold is set to default at 60%, to be received before finalizing the campaign and moving on to the next campaign. However, in the case of 300 Client contracts, the Custodian must delay for a sufficiently long span of time until all the 180 votes have arrived, so as to conclude the campaign and to produce the result. As the votes might arrive by anytime, according to the network stability and congestion rate, the resulted latency implies an incremental trend as the amount of votes to wait for are surging.

6.3.2 | Experiment 2: latency impact on concurrent consensuses

This experiment (2) has a focus on discovering the possibility of prompting a clusters of Clients to reach different consensuses at the same moment in a concurrent manner. The latency is recorded, as the number of consensuses to reach is increasing from 1 to 20. Each of the Client contract votes for those Custodian contracts asynchronously and concurrently to reach consensus on different topics. The x-axis is the number of the Custodian contracts, which each of them is responsible for a consensus topic, while the y-axis represents the latency yielded. There are a fixed set of Client contracts, with the number of 100, participating in all the consensus topics.

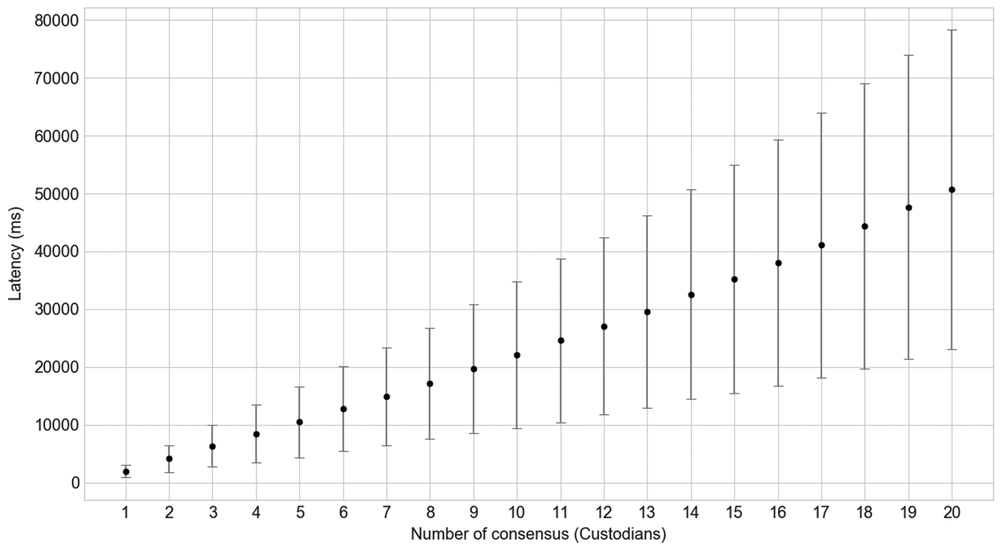

FIGURE 5 The latency test on numbers of consensus reached concurrently

From the indication of Figure 5, it is evident that it takes longer latency for the same set of Client contracts to reach a group of consensus topics at a larger number. Due to the fact that each of the Clients, or voters, is participating independently and simultaneously, the votes are cast to a specific Custodian, or consensus topic, at the same time. As discussed in the experiment (1), the latency for a Custodian contract to conclude a campaign majorly relies on the amount of votes to be received. In this case, since the adequate amount of votes to effectuate a campaign are fixed at 60 votes, which the total voter base is set to be 100 voters and the threshold is 60%, the latency for each of Custodian to finalize a campaign should be similar. This upward inclination is due to the restriction in the internal voting logic, which allows the client to vote for only one consensus at a time. This design guarantees every voting action is atomic and will be completed in a Client contract without manipulation from unexpected impact. Therefore, in order to vote for multiple consensus topics, a Client needs to iterate through all the Custodians and casts votes on one consensus at a time. The iteration accumulates the latency of each atomic vote actions up to be the increasing number of latency, as the vote actions to be executed are climbing.

This showcases the scalability discussion of the protocol design. In a network of a lot of clusters of Clients and Custodians, the latency produced for reaching multiple topics concurrently will rely on the average number of the consensus topics per Client and the average size of a cluster.

6.3.3 | Experiment 3: latency impact on threshold of votes

Experiment (3) has the purpose of addressing the impact that could be imposed on the latency by the adjustment of the effectuation threshold for a campaign's validity. The x-axis brings up the threshold from 25% to 100% by a step of 5%, and the y-axis shows the latency for a single

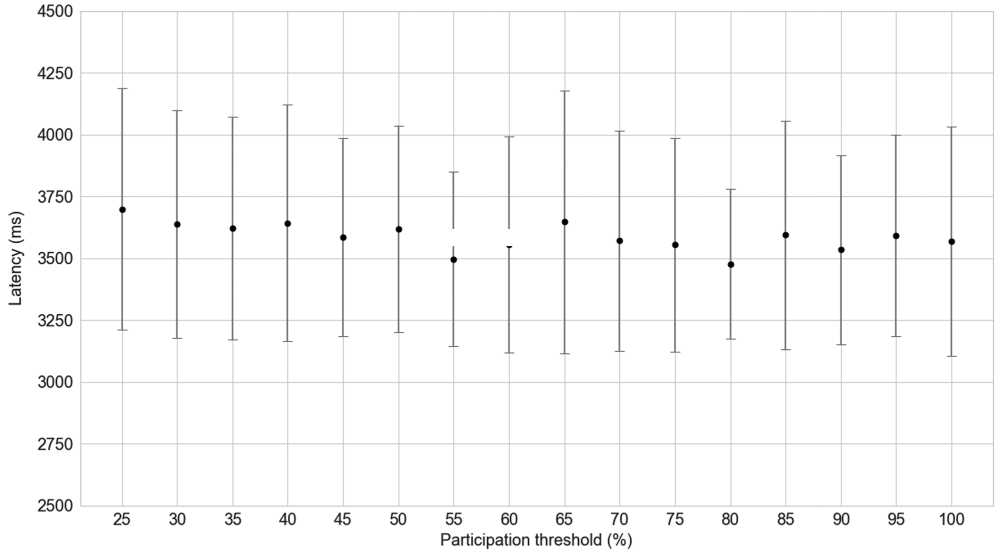

FIGURE 6 The latency test on various percentages of participation threshold

agreement to be reached upon. A fixed set of 100 Client contracts have been deployed and dedicated for these experiments. Each data point of average has 20 trials of experiments, which displays along with the calculated standard deviation on the figure.

From Figure 6, it is conspicuous that the latency tends to remain constant as the threshold grows. The reason is derived from the approach of casting votes. As detected in the network, all the Clients cast votes simultaneously, which will arrive at a designated Custodian at the same time. The Custodian will then digest the votes and compute whether the received votes have surpassed the threshold. Since the examination makes every client vote, the actual participation rate will always approach 100%, and thus, the Custodian will consider the campaign as effectuated right at the moment of evaluation, causing no divergences of latency across various thresholds.

This feature can provide scalability, in which the diversity of the threshold cannot influence the latency for consensus reaching, but only the to quantify how large the participation rate is accepted to be the effectuation threshold of campaign validity.

7 | DISCUSSION ON CONCERNS AND DESIGN EXTENSIONS

7.1 | Synchronization after a period of offline

In a distributed network, it cannot be guaranteed that all participants will stay responsive and synchronized perennially. Cases such as unstable network connection, functional failure, faulty peers, temporary isolation, and network congestion can hamper participants from being synchronized with the network. If the network protocol does not address this problem properly, haphazardly ripping off from the network can be painful for a participant to catch up when it is back online.

To tackle this issue, the proposed architecture ensures that each Client contract can enter and exit the network at wish, and can always sync up with the latest state of the network when recovering from a period of offline. As demonstrated in Figure 7, Client C can disconnect and catch up with the network later on seamlessly.

More specifically, in the Normal Case of Figure 7, Client A, B, and C act as ordinary voters at campaign with sequence number#12a, while Client D participates as a reticent outsider who casts no votes yet only queries the result of campaign #12a when finalized. As design, when a new vote is cast after the finalization of campaign #12a, the next campaign with a new sequence number #bef is initiated. In the Dropoff Case, Client A and Client B act as normal users, and Client C is disconnected from the network after campaign #12a. After the initiation of campaign #582, as shown in the Sync-up Case, Client C rejoins the network and casts a vote in campaign #582. The network simultaneously updates Client C with the finalized state of the latest voting campaign.

7.2 | Perpetual growth on voter base

Owing to the fact that the Custodian contract is designed to welcome as many new Client contracts to join the network as possible, the voter base, which consists of Client contracts, will continue to grow over time. Therefore, the level of difficulty for a voting campaign to receive sufficient votes over a determined threshold will only be increasing. Demonstrated in Figure 8, a voting campaign may never be finalized due to insufficient votes if the voter count remains large even when some voters leave the network permanently, no longer being functional, or vote

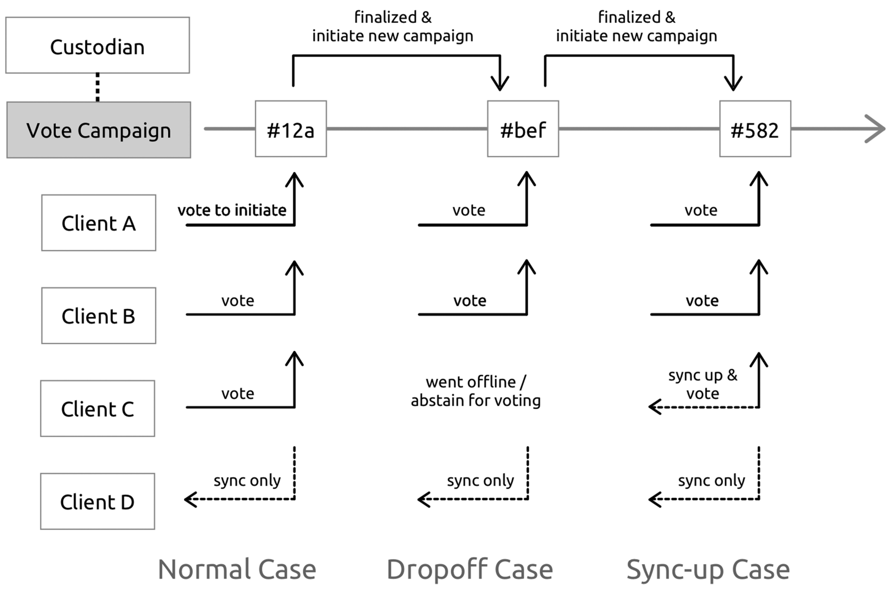

FIGURE 7 Normal case of a voting campaign

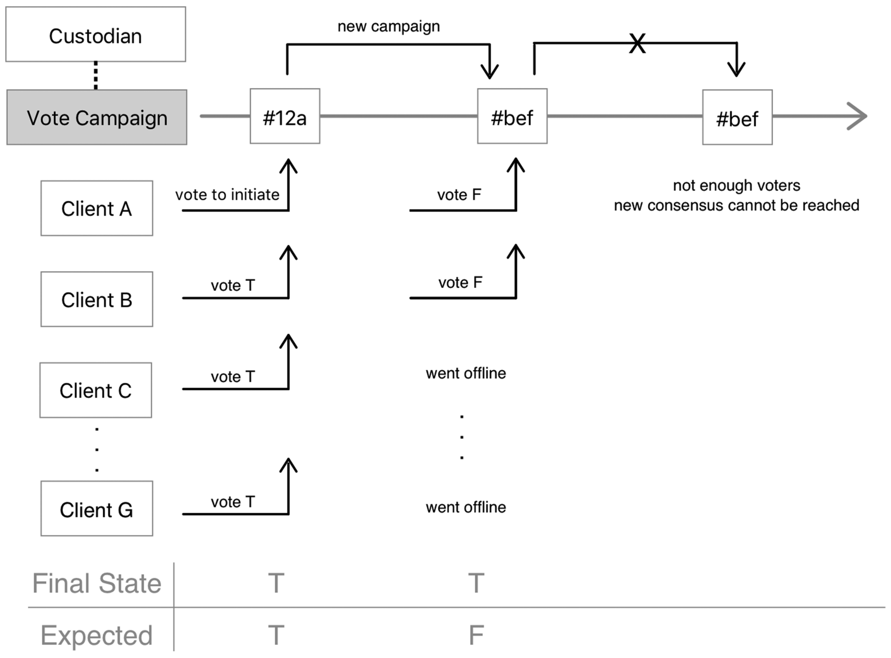

FIGURE 8 Insufficient votes in a voting campaign

with a extremely low frequency. To solve the problem, this paper proposes an algorithm to dynamically adapt to the changes in the total number of active voters to guarantee the liveness of the network.

To ensure the liveness of consensus reaching on the network, the expected voter base at time t, V_t, the determined threshold T, and the actual number of participants at time t, N_t are required to fulfill the following condition:

$$N_t \geq V_t T.$$

One approach is to extend V_t whenever an unknown voter joins the network. In this case, the definition of V_t can be described as

$$V_t := \max\{N_0, N_1, \ldots, N_{t-2}, N_{t-1}\}.$$

The design is the most intuitive implementation of the voter base approximation. Nonetheless, problems may arise as malicious or unreliable voters may leave the network and cause the expected voter base, V_t, to be much larger than the actual active voters, N_t. As V_t will only be increasing over time in this design, the condition $N_t \geq V_t T$ will only become harder and harder to meet. This implies that the liveness of the network cannot be guaranteed in the future at some time point t.

Another approach is to adjust V_t in accordance with the previous number of participants. The naive definition of V_t can be delineated as

$$V_t := N_{t-1}.$$

It is intuitive to take the last number of the participation as the expected voter base for the next round of the voting campaign. However, as the network might be exposed to Sybil attack and significant fluctuation of active participation, the resulted V_t is inclined to be unstable and unable to approximate the actual number of participants for the upcoming rounds of campaigns. The consequence can obstruct the consensus reaching in the case that participation number of round $t - 1$ pumped up to acme, whereas the participation at time t slump to the bottom. Therefore, a more eclectic approach is to consider multiple historical numbers of participants simultaneously by taking their average value. The V_t can be defined as the following:

$$V_t := \frac{\sum_{i=0}^{k-1} N_{t+i-k}}{k}.$$

The design can be implemented with a pipe of length k that aggregates and averages the latest k number of participants, disregarding any outdated round of campaign. This design is considered to be feasible for practical applications, since that any severe change of the participation number has a sufficient but confined interference on the expected voter base V_t. The liveness of the network can be achieved in a known bound of time that is contingent on the length of the pipe, k. The proof is intuitive that any large value will be abandoned at least after k rounds of campaigns, which may compromise the liveness of the network.

7.3 | Sybil attack and counteract

In the previous section, the paper discussed that it is possible to let the voter base continuously increase without a restriction, and lead to the oversize of the base that prevents the regular active voters from crossing the threshold to reach consensus. Aside from the normal case, it is

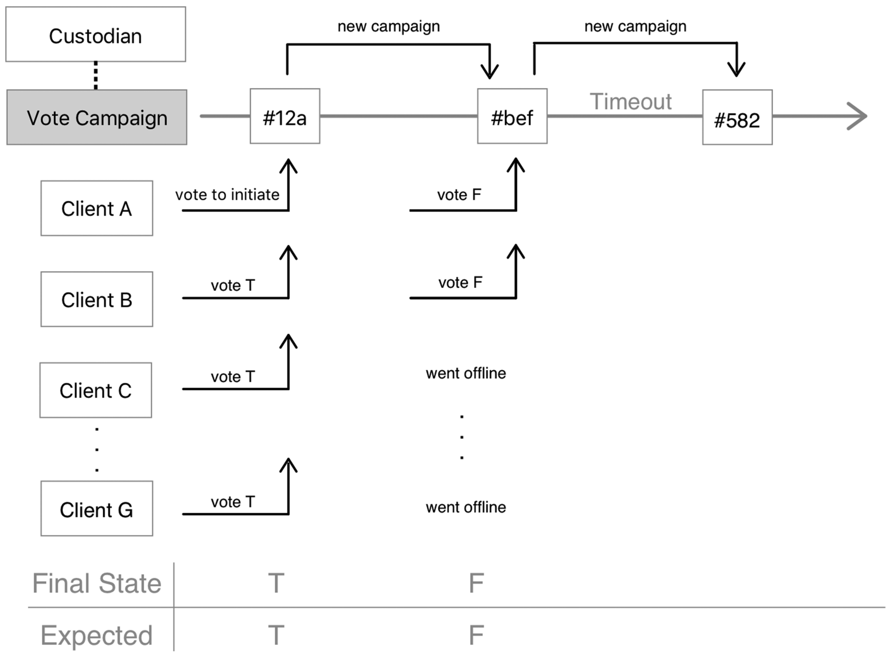

FIGURE 9 Timeout case in campaign finalization

possible that there are some adversaries in the network who aims at hampering the network from reaching agreements. The adversary can launch Sybil attack to create a significant number of identities, namely, Client contract, which is leveraged to stretch the voter base to a certain extent which ensures that the regular voters can never reach consensus.

To mitigate the effect, the aforementioned pipe average method can be adopted to solve the problem swiftly, yet if the adversary constantly generates lots of identities, the sum of the regular non-faulty voters can never drop. Other than the pipe method, this paper suggests another potential method to raise the threshold cost of the identity creation to sway the adversary from launching a costly attack. Specifically, the Custodian smart contract curates a membership list, where only the Client contract that paid the membership fee can be obligated with the rights to vote. To realize the membership, the Custodian smart contract requests ether as stack and refunds when the Client contract relinquishes the voting right. An alternative is to bolster a more mature token economy within the network. Particularly, the token economy mimics Ethereum's Casper PoS by asking token stacking for each vote and forfeiting the stacked token from malicious voters.

7.4 | Deterministic consensus by timeout mechanism

In the case that the participation rate for a proposed voting campaign cannot reach the threshold for decision finalization, ie, Figure 9, this paper proposes the method to timeout the agreement reaching procedure to ensure the network can move on to the next campaign without being blocked. The timeout has a default value that will be assigned to each of the proposed campaign, if not set by the first voter of the campaigns. In particular, a campaign is initiated by the finalization of the latest closed campaign, which has no voters at the very beginning. The campaign updates the Custodian contract with a specified sequence number as the newly opened campaign, which accepts any number of Client contract to be the voters. The new campaign with no voter remains in the condition of frozen timer, which will be unfrozen once a voter joins to vote at the campaign. No matter how insignificant the number of the participants has taken part in the campaign, the campaign can always timeout to proceed to the next campaign, in the case that the participation rate never reaches the threshold.

7.5 | Name server

Since it is required to provide a designated Custodian deployment address for a consensus to be reached by a particular group of Clients, the needs of a mapping between human-readable string and unreadable address has been considered. This paper proposed an approach to utilize a name server, similar to the design of DNS and Etheruem's ENS, to provide accessing a specific Custodian in a convenient manner (Figure 10).

7.5.1 | Name server smart contract

A smart contract is dedicated to storing all the mappings of the human-readable names and intents to the unreadable address. It provides accessibility to all the Client contracts that want to participate in the consensus reaching within a specified cluster with a corresponding Custodian contract. The workflow begins when a Client contract takes initiative to start a new consensus reaching cluster. The Client contract is required to

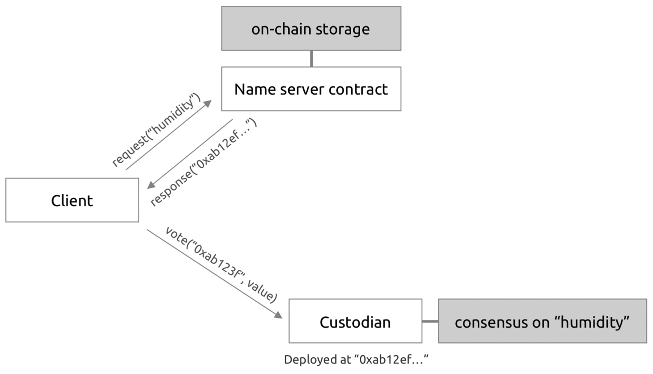

FIGURE 10 Name server architecture

register an arbitrary name at the Name Server smart contract with the deployment address of a new Custodian contract instance. Once the other Client contract queries the exact given arbitrary name, they are aware of the deployment address of the Custodian contract, the place where agreements will be reached. The Client contracts can vote to initiate and participate in campaign at that address for any round of state consensus.

7.6 | Applications to decentralized oracle

Decentralized oracles have been used to guarantee the integrity of the data that is originated from the outside space of the blockchain. Through a single oracle smart contract, it can be decided whether the input information is agreed among the majority of the contributors, which are the nodes themselves. As for the proposal, rather than depicting an alternative to substitute the conventional decentralized oracle at node-layer, this paper demonstrates an extension that is at the contract-layer. The major contribution addressed is creating an oracle machine that aggregates diverse information from multiple sources of smart contracts, not nodes, to reach a sharing state across multiple smart contracts.

It can be argued that the smart contract layer consensus is unnecessary. However, since a single oracle smart contract can still bear the risk of under control of a designated authority, which sacrifices the decentralizing goal of blockchain, a disintermediation within a network of smart contract is therefore in need. The contribution of the paper is to described a proposal that can benefit the established decentralized oracle to increase the scope of decentralization to the a thorough disintermediation, rooting out any potential of authority involvements.

7.7 | Applications to gaming complexity

7.7.1 | Non-fungible token

Non-fungible Token (NFT) is a type of token on Ethereum that makes one token disparate from others, and thus unique. In the recent past, Ethereum tokens are mostly indifferent, such as ERC20 tokens, which are widely used as a currency for payment, but hard to be applied to the gaming industry, as the assets in games are usually distinct. ERC721 is one of the first non-fungible tokens that solves the quest of setting Ethereum tokens apart from a bag of the rest. The implementation is to utilize a *Struct* definition in the smart contract to make every single token a *struct* instance. From that forth, each NFT token can store information that is pre-defined in the struct instance, so as to tell the differences between any two. The game industry starts to take into the NFT design, which is good for games such as card and asset collection, static character information storage, and some basic yet limited interactions between assets.

However, there are many features that even an NFT cannot capture, because of its lack of the following possibilities: (1) All NFT tokens in a single smart contract have the exact same interface and implementations. It is hard to include multiple types of NFT tokens in one smart contract; (2) NFT tokens can only interact with each other by the pre-defined rules and functions in the smart contract who created them; (3) Once the core smart contract, who manages all the NFT tokens, is compromised, the problem can cause enormous loss as the risk is concentrated; (4) The creator of the NFT token smart contract possesses excessive control over all the NFT tokens; (5) Players who own the NFTs cannot use their NFTs to vote for agreement reaching activities such as a game design change. The following section breaks down the aforementioned difficulties and proposes a solution, with the asynchronous state consensus reaching mechanism introduced in this paper, named the Non-Dominant Token (NDT).

7.7.2 | Non-dominant token

Evolving the asset management architecture, without a centralized *Struct* instance implementation as NFT, this paper leverages the autonomous characteristics of smart contracts instances to solve the aforementioned challenges. Each digital asset is now stored in the form of a smart contract, which can have complete control over itself, and is able to directly interact with other digital assets via a compatible interface. The NDT can have totally dissimilar designs and implementations from each other, and is no longer supervised by a centralized smart contract as in ERC721. In practice, NDTs can be applied to general gaming applications that require distinct functionalities of different characters and more interactive communications between each other.

Compared to ERC721, namely, Non-Fungible Token, NDT manages instances in the form of smart contract instead of *Struct*. This provides flexibility as smart contracts are independent of each other, yet connected through the shared state consensus managed by a Custodian contract. Loss from attack is also localized since the stakes of attacks are restrained in one single Client contract. The mechanism can be made compatible with ERC20 or other standards without restrictions as token instances can be in any form of smart contracts.

8 | CONCLUSION AND FUTURE WORK

In this paper, we have examined in depth the interactions between smart contracts by analyzing all the potential types and providing reference implementations towards sustainable distributed applications. This paper subtly modifies the roles of the Custodian smart contract and the Client smart contract, as introduced in the work of Hu et al[6] to bolster a design that allows multiple state consensus to be reached concurrently in a network with participants that can enter and exit at any time. A wide range of discussions has been covered the potentials for decentralized oracles, tokens in interactive games, and features such as timeout, name server, and the threshold of participation. Comprehensive comparison and introductions were included for myriad types of smart contract interactions. This paper now provides a suggested interface for a laconic design for any systems involving contract-level consensus and oracles.

The future work will have the focus on potential application to an improved version of the non-fungible token standard in games to address the needs of the interactions between different kinds of characters. The architecture can be further standardized as a communicative interface across various smart contracts with an aim of common state sharing and consensus reaching at the smart contract level. A more comprehensive discussion can be redressed to cover the possible solution to decentralized oracles and disintermediation.

ACKNOWLEDGMENT

We would like to express our gratitude to the Chungsang Tom Lam for comments that greatly improved the manuscript.

ORCID

Dimitris Chatzopoulos https://orcid.org/0000-0002-4765-5085

REFERENCES

1. Nakamoto S. Bitcoin: a peer-to-peer electronic cash system. 2008. https://bitcoin.org
2. Becker G. Merkle Signature schemes, merkle trees and their cryptanalysis. Technical Report. Bochum, Germany: Ruhr University Bochum; 2008.
3. Rosenfeld M. *Overview of Colored Coins*. White Paper. San Francisco, CA: Bitcoil; 2012. https://bitcoil.co.il
4. Wood G. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. White Paper. Zug, Switzerland: Ethereum; 2014.
5. Forum BR. Bitcoin Reddit Forum: The Oracle Problem; https://www.reddit.com/r/Bitcoin/comments/2p78kd/the_oracle_problem/; 2015.
6. Hu YC, Lee TT, Chatzopoulos D, Hui P. Hierarchical interactions between Ethereum smart contracts across testnets. In: Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems; 2018; Munich, Germany.
7. Buterin V. *A Next-Generation Smart Contract and Decentralized Application Platform*. White Paper. Zug, Switzerland: Ethereum; 2014.
8. Clack CD, Bakshi VA, Braine L. Smart contract templates: foundations, design landscape and research directions. 2016. arXiv preprint arXiv:1608.00771.
9. Vukolić M. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In: Camenisch J, Kesdoğan D, eds. *Open Problems in Network Security*. Cham, Switzerland: Springer; 2015:112-125.
10. McCorry P, Shahandashti SF, Hao F. A Smart Contract for Boardroom Voting with Maximum Voter Privacy. In: Kiayias A, ed. *Financial Cryptography and Data Security*. Cham, Switzerland: Springer; 2017:357-375.
11. Nugent T, Upton D, Cimpoesu M. Improving data transparency in clinical trials using blockchain smart contracts. *F1000Research*. 2016;5.
12. Warren W, Bandeali A. *0X: An Open Protocol for Decentralized Exchange on the Ethereum Blockchain*. White Paper. San Francisco, CA: 0x; 2017.
13. Brave Software. *Basic Attention Token (BAT): Blockchain-based Digital Advertising*. White Paper. Basic Attention Token (BAT) Blockchain Based Digital Advertising. https://basicattentiontoken.org; 2018.
14. Cryptokitties. *Cryptokitties: Collectible and Breedable Cats Empowered by Blockchain Technology*. White Paper. Vancouver, UK: Cryptokitties; 2017.
15. *The Status Network: A strategy towards mass adoption of Ethereum*. https://status.im; 2017.
16. OpenZeppelin: An open source library that contains best practices for smart contract development. https://github.com/OpenZeppelin/openzeppelin-solidity; 2016.
17. Baylina J. ERC820: Pseudo-introspection Registry Contract. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-820.md; 2018.
18. Dafflon J. ERC777: A token standard that allows operators to transfer tokens on behalf of another address. https://eips.ethereum.org/EIPS/eip-777; 2018.
19. ZeppelinOS: A platform to develop, deploy and operate upgradeable smart contracts on Ethereum and every other EVMand eWASM-powered blockchain. https://zeppelinos.org; 2018.
20. Peterson J, Krug J. *Augur: A decentralized, open-source platform for prediction markets*. White Paper. Augur; 2015.
21. Adler J, Berryhill R, Veneris A, Poulos Z, Veira N, Kastania A. ASTRAEA: A decentralized blockchain oracle. 2018. arXiv preprint arXiv:1808.00528

22. Oraclize: Data carrier for decentralized applications. http://www.oraclize.it
23. Ellis S. *A Decentralized Oracle Network Steve Ellis, Ari Juels, and Sergey Nazarov*; 2017.
24. Bentov I, Gabizon A, Mizrahi A. Cryptocurrencies without proof of work. In: Clark J, Meiklejohn S, Ryan P, Wallach D, Brenner M, Rohloff K, eds. *Financial Cryptography and Data Security*. Cham, Switzerland: Springer; 2016:142-157.
25. Gibbons R. *A Primer in Game Theory*. Upper Saddle River, NJ: Prentice Hall; 1992.
26. TLSNotary - a mechanism for independently audited https sessions. 2014.

How to cite this article: Hu Y-C, Lee T-T, Chatzopoulos D, Hui P. Analyzing smart contract interactions and contract level state consensus. *Concurrency Computat Pract Exper*. 2020;32:e5228. https://doi.org/10.1002/cpe.5228