Smart Contract Design of Ultrain

Version 1.0

Introduction

Smart contract is a protocol that transmits, verifies, and executes contracts in a purely electronic way. It allows trustful transactions to be carried out without third-party's supervision and the transactions bear the characteristics of being untraceable and irreversible. The concept of smart contract was first introduced by Nick Szabo in 1994 but didn't see widespread adoption at that time due to the lack of trustful execution environment until the introduction of blockchain technologies. In recent years the increasing popularity of blockchain has fueled the development of smart contract related technologies, especially those popular public blockchain projects like Ethereum and EOS have greatly advanced the research for smart contract and pushed it towards the direction of large-scale business adoption.

But we also realized that the status quo of blockchain and smart contract design still have some serious problems which make it very hard for supporting large-scale business use cases: 1. Unfriendly to smart contract developers: e.g. brand-new programming languages can't leverage developers' existing skills and experience; low productivity due to the lack of developing tools/SDK; 2. Poor interacting and representing capability of smart contract: e.g. the updating process of smart contract is not flexible which makes the development and deployment cycles laggy; strong coupling of contract running cycles with consensus cycles puts over-constrains on the allowed running time of smart contract which compromises the practical usefulness of alleged Turing-complete smart contract. 3. Poor runtime performance: e.g. blockchain system achieves its high level of security and decentralization through massive computation and storage redundancy, which in turn drags down the network's overall runtime efficiency.

At Ultrain, after a long-time contemplation and exploration for the abovementioned issues with current smart contract system, we have gained invaluable insights and built our own creative solutions. In this paper we will discuss in details the following topics: "Flexible Smart Contract Interaction", "Dynamic-and-Adaptive Sharding" and "Friendly Developing Environment for Smart Contract". With these features we are building a holistic solution for smart contract system that is friendly for developers, rich in expressing capability, highly-efficient in runtime performance and secure. And it forms the foundation for supporting Ultrain's grand vision for creating a "programmable business society".

Flexible Smart Contract Interaction

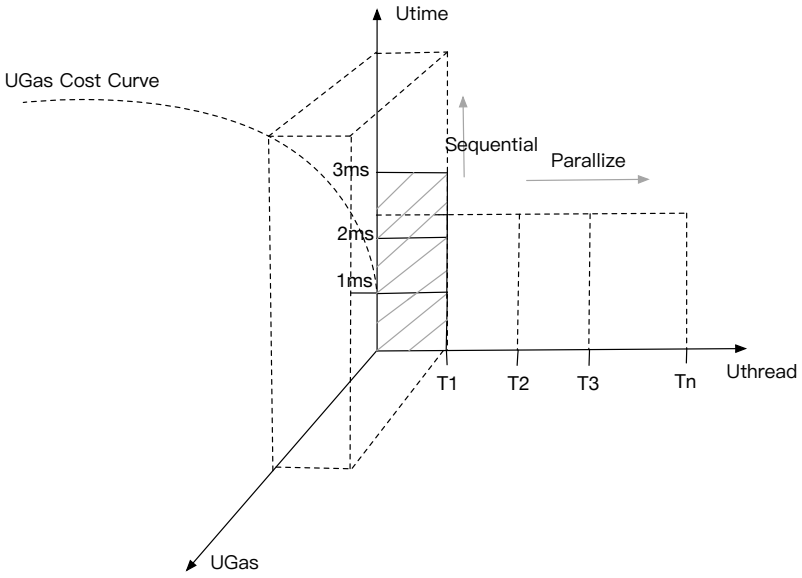
1. Quota management for smart contract's running time

Before getting into the details of quota management for smart contract' s running time, we first give a brief introduction of the concept of transaction. Transactions drive the running of smart contract in blockchian network, and the the output of smart contract relates to the final confirmation of transactions.

Lets first take a look at Ethereum. Ethereum has only one type of transaction which does not have lifecycle management. Transaction triggers the synchronous running of smart contract. The allowed running time of a smart contract is constrained by the Gas value provided in the transaction. If the contract does not finish before the provided Gas is used up, the contract will be forced to stop, any applied state change will be rolled back and the transaction is marked as failed. When integrating Turing-Complete smart contract with periodical consensus algorithm, one has to face the Halting Problem and the Gas design in Ethereum is considered a workaround for that.

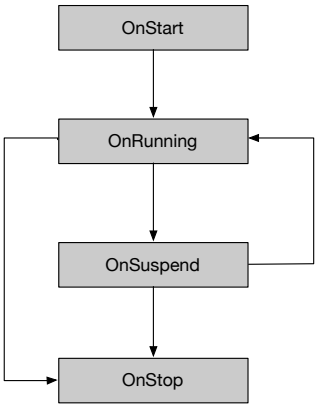
Secondly we take a look at EOS' s solution. The major difference between EOS and Ethereum is that for solving the Halting Problem, EOS sets a hard limit on the allowed smart contract running time to guarantee that the running of contracts aligns with the consensus cycles. Another difference is that EOS introduces a new type of transaction called Deferred Transaction. It allows transactions/smart contracts to be scheduled in future consensus cycles, which in certain sense simulates the case of a long-running contract. But the programming paradigm promoted by this design also imposes huge burden on smart contract developers, e.g. long-running contracts have to be logically divided into several sub-contracts that can each fit into the consensus cycles; and also one major drawback of this solution of EOS is that the system does not guarantee the execution of the scheduled deferred transactions.

It is clear that both the Gas and Deferred Transaction designs are aiming to solve the Halting Problem when there is practical demand of smart contracts running cross several consensus cycles. But neither solution is totally satisfactory. Now we introduce the Ultrain solution. There are two types of transactions in the Ultrain network: the first one, also the default one, is Short-Range Transaction, the smart contract triggered by which is expected to be finished in a short range of time, and the system imposes a 1ms limit on its running time; if the 1ms limit is exceeded, the transaction will be marked as failed and the system state will be rolled back. This type of transaction fits nicely into one consensus cycle so both the single-node parallelization and the network-wide sharding can be applied for boosting the system throughput. We will give detailed explanation in the following "Dynamic-and-Adaptive Sharding" chapter. Another type of transaction is Long-Range Transaction, which carries a smart contract that is expected to encode complex application logic that can not be finished in a short period of time. The design rational is that for accommodating the practical demand of long-running contracts from application developers, we allow the kind of contract execution mode that is without the 1ms limit but instead rating the resource limit through economical cost. We designed an exponential cost function between contract running time and the charged UGas to discourage the potential long-running contracts (maliciously) over-consuming computational resources. The following diagram depicts the running dynamics of Ultrain smart contracts:

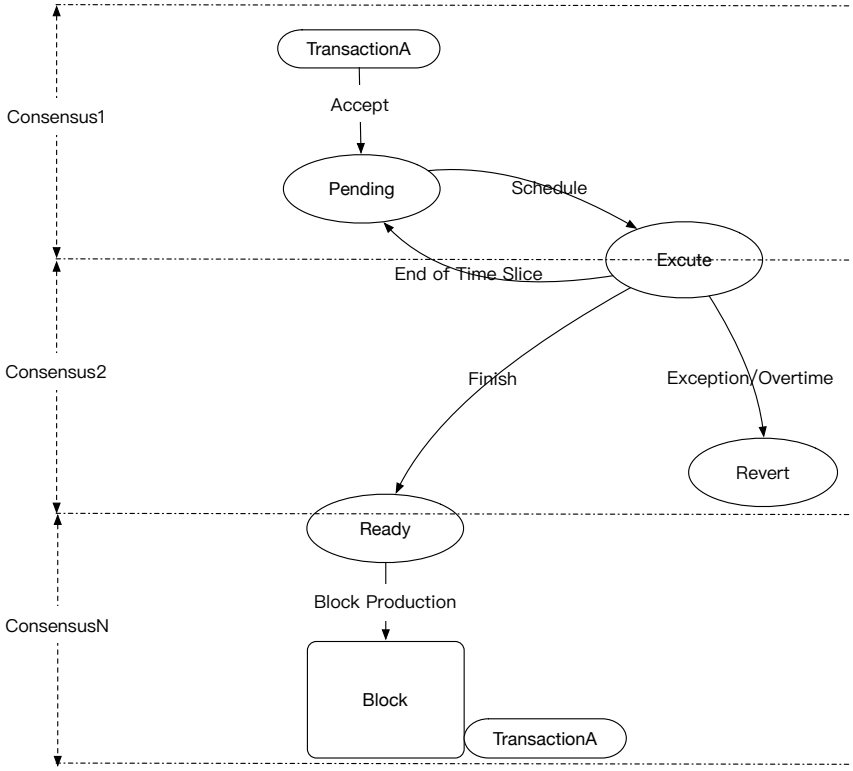


Above we designed the quota management for smart contract running time. But for this design to work stably and performant under the environment of a totally open public blockchain network, we must first solve the key problem: how to make sure at least 2/3 of all the participating nodes can reach the consensus of the world state for each block production. First we have to provide deterministic input elements that support the contract execution, e.g. the timestamp and data source. There are some good designs in the blockchain technology community we can refer to. For timestamp, we provide the system API based on the block timestamp, i.e. the blockchain itself can become a timestamp server and the timestamp for each block can be queried so that all the participating nodes can agree to the same deterministic timestamp source. With the deterministic agreement on input elements and running environment, the next step is to design a creative contract lifecycle management algorithm that can handle mixed Short-Range and Long-Range transactions. For the Long-Range transactions, we introduced the state management showed in the following diagram to decouple contract running cycles from consensus cycles (i.e. allowing contract running to cross consensus cycles) so that the consensus of the eventual contract state can be reached.

We introduce the following four contract lifecycles into our runtime contract management: **OnStart**, **OnRunning**, **OnSuspend** and **OnStop**.



Now lets assume there is a Long-Range Transaction called **TransactionA**, its execution workflow is as follows: when it is first accepted, its state is marked as **Pending** and put into the queue with other Long-Range Transactions waiting to be processed, i.e. **TransactionA** ∈ LRT （LRT = $\{\mathrm{lrt}_1, \mathrm{lrt}_2, \mathrm{lrt}_3 \ldots \mathrm{lrt}_N\}$）. A fixed period of time is allocated before each consensus cycle for processing the **LRT** set (time-sharing is deployed so that each transaction gets chance to be executed). Transaction triggered smart contract gets started and enters into **OnStart** phase. The major job of **OnStart** phase is to lock down all the world state that is related to the accessible accounts in **TransactionA** and marks **TransactionA** as in state **Excute**. Then the contract moves into **OnRunning** phase. **OnRunning** is when the contract' s major logic happens and since the execution time of contracts varies, if it can not finish within the allocated time slice then the contract will enter into **OnSuspend** phase. **OnSuspend** sets breakpoint on current execution environment and puts execution back into **Pending**. When the next time slice arrives, the contract execution resumes until the execution finishes normally or the **UGas** is used up, then the contract enters **OnStop** phase. If contract execution finishes normally then **TransactionA** is marked as **Ready** and waits to be confirmed in the next consensus cycle, or if there is execution exception due to insufficient **UGas** then **OnStop** will roll back any state change and notifies the sender of **TransactionA** of its failure; The last step of **OnStop** is always unlocking all the locked down parts of the world state.



As we can see the execution of Long-Range Transaction locks down the world state of its related accounts during its whole lifecycle, even when the contract is in **Pending** mode waiting to be scheduled. During the lockdown, all other conflicting contracts, be it Short-Range Transaction or Long-Range Transaction, will have to wait until the world state to be unlocked. The execution of Long-Range Transaction is indeed very costly, not only in terms

of the running contract's own resource consumption, but also the opportunity cost of all the other de-scheduled conflicting contracts. That justifies the design rational behind the exponential relationship between contract running time and UGas charge. We believe this model is superior to EOS's Deferred Transaction model in the following aspects: we provide semantic guarantee of the long-running contract, whereas EOS's Deferred Transaction's indeterministic promise on scheduling and execution leads to very complex exception handling design and puts too much burden on developers; while under our model, consensus layer is transparent to contract developers, they only need to worry about how to simplify application logic to reduce UGas cost, unlike EOS developers who have to design contract series to align with consensus cycles.

2. Contract Update Mechanism

We borrow the design experience of process management in traditional Operating System for our contract update mechanism: within operating system different processes have different importance so they are managed and scheduled by different priorities. Blockchain network is intended to support the deployment of massive smart contracts all serving various purposes and bearing various characteristics: some smart contracts emphasize more on its "contract" characteristics while the participating parties believe in "code is law", i.e. they believe that the contract itself is not changeable or revocable once deployed; some smart contracts are more like traditional applications that are built on top of the blockchain operating system and they are intended to provide update-to-date services to customers so the key feature here in demand is the contract's business logic can be updated frequently. With all these different and sometimes conflicting demands, there is no "one size fits all" design. So in Ultrain world we have designed three different types of contract for serving different scenarios: general, protected and restrict contract:

- **general contract:** contract owner can update the contract code while retaining the same contract address, mainly for the scenario where the "service" characteristics is more in demand.
- **protected contract:** contract owner can propose the contract update, but only when all or part of the designated users agree on the proposal can the update go through, mainly for the scenarios when multi parties are collaborating.
- **restrict contract:** once deployed, no one including the contract owner can update the contract, which is the smart contract in its most restrictive meaning. It is suitable for most blockchain applications' scenarios and is also the default type of Ultrain smart contract.

Dynamic-and-Adaptive Sharding

The most straightforward computational model for smart contract is that all the participating nodes keep all the accounts' and contracts' state (so called world state) and execute all the same smart contracts sequentially. This vanilla model ensures a high degree of the whole network's Security and Decentralization at the cost of massive computation and storage redundancy. Hence the system loses its Scalability and the computational power

of one single node could becomes the bottleneck of the whole network' s throughtput. Tradeoff is a recurring theme in solving most computer science problems, e.g. space VS time, and a well-designed blockchain system should also thrive to achieve a balance point within the trilemma of Scalability, Security and Decentralization. We are glad to see that the blockchain community has put massive effort into improving the blockchain network' s Scalability, e.g. sharding and sidechain technologies can significantly improve the netwrok' s TPS (transactions per second). These technologies share the same nature that it improves the Scalability at the cost of compromising Security and Decentralization to a certain extend. We should be aware of that when we are crunching high TPS number, we don' t really have a quantitative way of measuring the lose in the aspects of Security and Decentralization. Using TPS as the one and only criteria when comparing different blockchain systems is unfair. Ultrain' s contributions to this area are: 1) first we designed a state-of-the-art computation sharding architecture that is smart contract friendly and highly-performant in throughput. 2) based on the smart contract computation sharding algorithm, we propose a dynamic and adaptive sharding strategy that allows the network to dynamically change its sharding configuration according to the current network condition, which enables the flexible and dynamic anchoring in the trilemma. We call this technique Dynamic-and-Adaptive Sharding. Next we will give details of the above two innovations.

Sharding holds the same philosophy as the classic Divide-and-Conquer algorithm in computer science. We divide the whole network into many small sub-networks called shards. Each shard is assigned part of the all transactions and contracts for execution and then all the results are merged together. The sharding of computational power is obvious in this model, but what' s unclear is whether storage is sharded – that is whether each shard shares the same whole world state or only keeps a copy of part of the whole world state. In our design, the storage is not sharded (i.e. each node holds the full copy of the whole world state). The reason is that storage sharding inevitably introduces the overhead of cross-shard communication and whether this overhead is manageable is still not proved even in the scenario of simple money transfer, let alone the scenario of transactions carrying complex smart contracts. We believe that the status quo of storage sharding research is not mature enough to provide a practical solution yet. So all the following discussions are limited only to computation sharding.

Lets first take a look at the case of simple money transfer. Our design principle is that we want a sharding configuration without the need of cross-shard communication, so that every transaction can be independently verified within the shard it is assigned to. We denote the set of all the accounts in the blockchain network as A, and T, of size C, is the set of transactions waiting to be verified:

$$T_i = (s_i, r_i) \ where \ i \in C, s_i \in A, r_i \in A$$

represents a money transfer transaction from account s_i to account r_i, and we define an acyclic graph:

$$G = (V, E), \ where \ V = s \cup r, \ E = \{(s_i \ r_i) : i = 1,2 \dots C\}$$

With $O(|V| + |E|)$ time complexity we can find the connected components of this acyclic graph. These connected components form the basis for our sharding algorithm. As long as we can promise that transactions within the same connected component are assigned to the same shard, there would be no cross-shard communication overhead. The details of connected components based sharding algorithm will be discussed later.

Now we look at the more complex case of smart contract execution. Unlike the case of simple money transfer where before the transaction verification we already know the two involved accounts, smart contracts can dynamically call into other smart contracts which means the overall involved smart contracts and their related accounts/data can only be known at runtime, making it impossible to reuse the connected components based sharding algorithm designed for the simple money transfer cases. To solve this problem, one straightforward idea is to get developers involved – for a contract, the developer can statically declare a list of all the other contracts it can directly invoke, then through static code analysis we can know in advance all the possible contracts that might be invoked (through recursion), we call these contracts the invocation set. The invocation set forms a basis for smart contract sharding. The major drawback of this design is that it loses the smart contract' s dynamic updating capability. Suppose that we have a contract calling chain like ContractA → ContractB → ··· → ContractY → ContractZ, if we want to update the code for ContractZ, re-deploying ContractZ will change its contract address (here we are assuming the the contract is a **restrict** contract that re-deployment results in contract address change), and the address change will turns into a chain reaction causing all the contracts in the calling chain to be re-deployed. Image that a popular library contract decides to push out an update and all the contracts relying on it (directly or indirectly) need to be re-deployed. This is not acceptable in any serious engineering practice. The right way of updating that we want to see it happen, and which is also the common practice widely accepted by the community and developers, is: ContractY keeps a handler pointing to the address of ContractZ, and when ContractZ changes its address due to code update, one only needs to update the handler in ContractY to point to the new address of ContractZ, this way no other contracts along the calling chain get affected.

To achieve that, our approach is like this: by monitoring the read and write of a special data structure that represents the smart contract object, we can statically derive the invocation set at any time point without compromising the desired contract updating mechanism. First we give the following definitions for later discussions:

1. Assume in our smart contract programming language ContractInstance is used for representing a contract object.
2. Contract invocation has only one syntax form: e.g "Call ContractInstance" will invoke the contract whose address is held in the ContractInstance.
3. If the execution of a smart contract has any creation/destroy/update operation, we call this a TC-DIRTY operation, otherwise we call it a TC-CLEAN operation.
4. For a transaction triggering smart contracts, there is a special flag needs to be set. If the sender of the transaction expect the execution will include TC-DIRTY operations, he/she sets the transaction flag as TC-SHOULD-DIRTY, otherwise the sender sets the flag as TC-SHOULD-CLEAN.
5. If and only if a TC-SHOULD-CLEAN transaction triggers a TC-DIRTY operation during

execution, we call it a TC-BREAK condition.

6. Suppose now there is a set of N contracts waiting for execution called $T = \{t_1, t_2 \dots t_N\}$, of which there are NC contracts marked as TC-SHOULD-CLEAN and the set is called $TC = \{tc_1, tc_2 \dots tc_{NC}\}$, and of which there are ND contracts marked as TC-SHOULD-DIRTY and the set is called $TD = \{td_1, td_2 \dots td_{ND}\}$
7. We define the mapping $F\_LIST\_ONE(t)$ as the set of contracts pointed to by all the ContractInstance in contract t (including t itself)
8. We define the mapping $F\_LIST\_ALL(t)$ as:
   8.1 $TEMP = F\_LIST\_ONE(t)$
   8.2 $TEMP\_EXPAND = \{F\_LIST\_ONE(tmp_1) \cup F\_LIST\_ONE(tmp_2) \dots : tmp_i \in TEMP\}$
   8.3 If $TEMP == TEMP\_EXPAND$, then $F\_LIST\_ALL(t) = TEMP$, and exit
   8.4 $TEMP = TEMP\_EXPAND$, goto 8.2

In summary, $F\_LIST\_ALL(t)$ represents the set of all the directly and indirectly accessible contracts by the TC-SHOULD-CLEAN contract t under no TC-BREAK condition.

Next we describe the condition for smart contract sharding:
For any two TC-SHOULD-CLEAN contracts tc_i and tc_j, if $F\_LIST\_ALL(tc_i)$ and $F\_LIST\_ALL(tc_j)$ are disjoint, then the 2 contracts can be assigned to different shards for execution since they are reading and writing totally different state space.

1. We define an acyclic graph $G = (V, E)$, where

$$V = \{F\_LIST\_ALL(tc_1) \cup F\_LIST\_ALL(tc_2) \cup \dots \cup F\_LIST\_ALL(tc_{NC})\}$$
$$E_{tc_i} = \{all\ edges\ of\ Complete\ Graph\ of\ F\_LIST\_ALL(tc_i)\}$$
$$E = \{E_{tc_1} \cup E_{tc_2} \cup \dots \cup E_{tc_{NC}}\}$$

Also with $O(|V| + |E|)$ time complexity we can find the connected components of this acyclic graph. These connected components form the basis for our sharding algorithm. As long as we can promise that smart contracts within the same connected component are assigned to the same shard, there would be no cross-shard communication overhead.

2. As for smart contract t marked as TS-SHOULD-DIRTY, since it can dynamically change the elements in set $F\_LIST\_ALL(t)$, it can not be sharded correctly. One simple approach is to put these kind of transactions into an independent queue and run the execution in a non-sharded way at a certain time point. Since the flag of TS-SHOULD-DIRTY means contract update, it should have relatively low occurrence, so this kind of special treatment should not have big impact on the network' s long-term performance.
3. One exception is when TC-BREAK condition happens which will break the precondition of sharding and the system should roll back any state change and fall back to non-sharding mode. We should use tools like static cod analysis and economical penalty for discovering this condition and help developers to reduce the probability of such occurrence.

Up until now, for both the scenarios of simple money transfer and complex smart contract execution, we are able to derive the graph based connected components; for

transactions/contracts that fall into the same connected component, they must execute sequentially so as to reach a deterministic final world state since they might read/write the same part of the world state; as for transactions/contracts that fall into different connected components, they can run concurrently without interfering with each other. Connected components are the foundation for divider-and-conquer. Suppose we put transactions that are within the same shard into a transaction group, and call all the groups the transaction group set:

$$TG = \{TG_1, TG_2, \dots, TG_m\},$$

theoretically we can do up to m-ways concurrent computation. Next we talk about the two cases of single-node concurrency and multi-node sharding.

For the case of single-node concurrency, the problem we are facing is how to utilize the single-node' s hardware ability, e.g. multiple-core and multiple-hyperthread for best concurrent computation. Suppose the hardware has maximal N concurrent computation units, we call them

$$\text{CORE} = \{CORE_1, CORE_2 \dots CORE_N\}$$

Then the problem is how to distribute the m transaction groups onto the N computation units for execution. For simple transaction like money transfer, the execution time complexity of TG_i is proportional to the number of transactions in the group. If we denote the total complexity of all the transactions assigned to $CORE_i$ as $S(CORE_i)$, then the distribution can be thought of solving the min-max problem: $min\frac{max}{i=1\dots N}\ S(CORE_i)$. As for smart contract execution, since its time complexity can' t be statically inferred, doing a round-robin algorithm on the N computation units is a simple yet effective solution.

As for the case of multi-node sharding, lets denote all the available nodes in the network as

$$Node = \{Node_1, Node_2, \dots, Node_{N\_NUM}\}$$

And the number of shards is $SN \in [1, N\_NUM]$. Unlike the single-node case where N is proportional to the hardware' s concurrency capability, the value of SN essentially maps to an anchor point in the triangle formed by Security, Scalability and Decentralization, and reflects a balance choice under the trilemma. The key idea of our design is that this balance point is not fixed but rather dynamically adjustable. Suppose currently the network has the sharding configuration

$$w_{ij} = \begin{cases} 1 & if\ TG_i\ is\ assigned\ to\ shard_j \\ 0 & otherwise \end{cases}$$

$$s_{ij} = \begin{cases} 1 & if Node_i\ is\ assigned\ to\ shard_j \\ 0 & otherwise \end{cases}$$

The utility function for the sharding algorithm is

$$\frac{min}{N\_NUM, SN, w, s}(SecurityLoss + \alpha * ScalabilityLoss + \beta * DecentralizationLoss)$$

Under the most complicated model assumption, all of $SecurityLoss$, $ScalabilityLoss$ and $DecentralizationLoss$ are the functions of (N_NUM, SN, w, s), but that would make the model intractable, so in practice we can make the following simplification (Here we omit N_NUM, assuming its impact is always represented through SN):

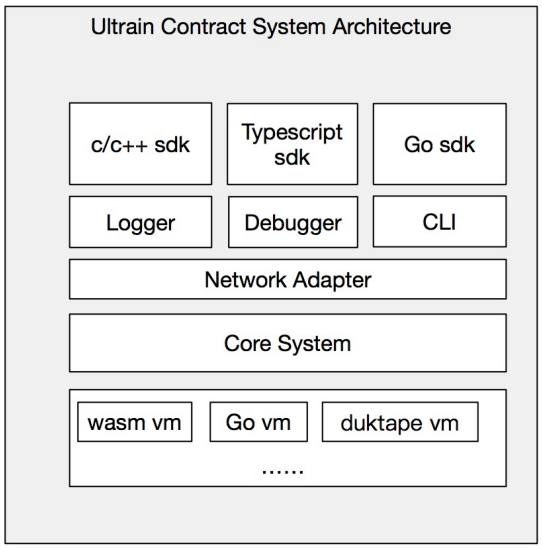
$$SecurityLoss \sim (SN)$$
$$ScalabilityLoss \sim (SN, w)$$
$$DecentralizationLoss \sim (SN, s)$$

Intuitively $SecurityLoss$ is mostly affected by the number of shards SN, e.g. in the extreme case where SN=1, we have the strongest security level which equals to no sharding at all. $ScalabilityLoss$ depends on both the number of shards and how the computation workload is distributed among those shards. $DecentralizationLoss$ then depends on the number of shards and also how the participating nodes are assigned to which shard. It is worth pointing out that the core part of our design is the introduction of the two weight items α and β, while the concreate forms of those $Loss$ functions are more of implementation details. Once the concreate forms of $Loss$ functions are set, how to adjust weight items α and β reflects how we balance the trilemma problem. We believe that the value of α and β should be dynamic so as to be adaptive to the current network condition and be aligned with the the community's expectation for how the network should be evolving. For example, if there is network congestion happening, we might want to increase the weight of α, i.e. meaning a higher degree of sharding and hence the concurrency, to increase the network throughput; or if there is a sign of increasing number of malicious nodes threatening the security assumption of our consensus algorithm, then we might want a smaller α for the time being, i.e. a weaker sharding configuration (each shard will have more participating nodes) to defense the potential malicious attack; or if the network sees the sign of over-centralization, we might want to increase the weight of β so as to achieve a better nodes-over-shards distribution that's more decentralized. The process of dynamic adjustment of α and β cloud be achieved by voting through network goverance, or a smart contract that constantly monitors the network conditions. But the key remains in the dynamic nature of α and β. That's why we call it the Dynamic-and-Adaptive Sharding.

Friendly Developing Environment for Smart Contract

For supporting the grand vision of "programmable society", Ultrain blockchain platform provides comprehensive supporting tools including docs & tutorials, SDK, debugging tools, deployment tools, wallet, explorer and etc, all of which streamlines the smart contract developing cycles on Ultrain.

Following is the architecture of Ultrain smart contract system:



Programming language and SDK: Ultrain supports several popular high-level programming languages for developing smart contracts: C/C++, TypeScript/JavaSscript and GoLang. Compared with Solidity from Ethereum, high-level programming languages have more flexibility, more matured language syntax, lower learning cost, and easier to write secure and audit-friendly smart contracts. And high-level programming languages also have better support for modern programming design pattern which increases developer's productivity.

Logger System: Ultrain platform provides logging output (without persistent storage). Contract developers can configure the logging output's format and address, e.g. POST logs can be redirected to specified URL address. There is no real-time promise in logging system.

Debugger System: Under the debug mode, smart contract can run without coupling with consensus algorithms. And also the gas constrains, run time quota constrains and system resource (e.g. IO/RAM) limit rating constrains can also be lifted for developing convenience.

1. For easier debugging, the platform provides usc-mocker tools that can simulate a local node running environment. Both GUI and CLI modes are provided. After mock environment is setup, it can generate test accounts with pre-allocated testing tokens;
2. We also provide a local wallet tool, which can be started by "usc wallet" command. It has full support for account related operations: account creation, account switch, money transfer between accounts and etc.
3. Ultrain platform provides the MyUltrainWallet tool for convenient wallet management and contract deployment and invocation; it supports both production and test environment and can also easily switch between the two. When deploying the contract, it will automatically calculate the desired UGas and check whether the contract can run

successfully. For contract invocation, all the callable methods can be queried and directly invoked.

CLI tools. The framework tool usc-cli provides a rich set of basic operations like init, compile, migrate, test and wallet. Through these operations, developers can choose contract templates, built-in syntax check, code formatting, unit test and report test coverage stats. It also supports automatic contract generation.

Network Adaptor. Ultrain platform provides test network environment. The test network can be configured to have strong consistency which can simulate the average network condition of the production environment. Through network adaptor one can switch between test and production network easily. The platform also provides ultrain-explorer which has interface for querying blocks, accounts, transactions and APIs. The online version of ultrain-explorer will also have developer community integrated for easy communication between developers.

Multi-Virtual Machine Support. Connecting to the Ultrain mainnet, different sidechains can choose to adopt different virtual machines to serve different application needs under various complex real world scenarios.

Ultrain Consensus yellow paper

June 17, 2018

Contents

Abstract

Ultrain targets at The purpose of Ultrain is to improve the performance of the underlying blockchain through innovations such as sharding and parallelism without loss of security. At the same time, through the reflection on the upper level smart contracts and applications, Ultrain redefines the application architecture.

1 Foreword

For the blockchain application scenario, the consensus algorithm needs to have the following features

- Throughput: TPS(Transaction per second) means that the blockchain system can handle the number of transactions per second. In order to support global users and multiple applications, TPS must be high enough, otherwise it will cause long transaction delay, such as Ethereum's network congestion caused by crytokitties.
- Latency: The time from the submission of one transaction to the final execution result. In the PoW network, because of the possibility of fork, it is generally necessary to wait for several blocks. In the BFT consensus system, since all nodes make consensus before building the next block, the response speed is the same as the block production time.
- Scalability: The more nodes participate in, the more decentralized the system security is. But for traditional BFT systems, the more nodes there are, the greater the communication and computational overhead is. Algorand proposed to randomly select some nodes using Verifiable Random Function to form a constant amount of committee members, which ensures that regardless of the total number of nodes, the consensus latency will not change.
- Security: For BFT consensus, security includes consistence, liveness, and fairness. Consistence means that all honest nodes in the system will eventually reach the same state. Liveness requires the system to converge to a certain state in any situation, and can continue to accept transactions. Fairness is that for users of the system, any legitimate transaction will not be rejected. For systems based on PoW consensus, the degree of security is related to the honest computing power. For systems based on BFT consensus, the more participating nodes, the more malicious nodes that can be accommodated as long as malicious nodes are less than 1/3.

For traditional PBFT[1, 2] system, it is assumed that the network is in a weakly synchronized state, and the liveness is guaranteed by timeout and view change. However, because the leader is deterministically switched to the pre-defined next node, and view change takes a long time, leaders may suffer from continuous attacks and the system is stuck at view change forever.

Regarding this, Tendermint[3] proposed that after each phase of consensus, the leader is deterministically switched to the next node, therefore avoiding malicious leaders and costly view change procedure, since each leader only occupies fixed time window and phases. However, this still cannot avoid the predictability of leader selection, resulting in sequential attacks, and the consensus result in each phase is empty. DPoS systems such as EOS, suffers from similar problem, since all proposers are predictable and attract attacks.

Honey Badger[4] proposed a protocol based on RBC (Reliable Broadcast) and BA (Binary Agreement) based on asynchronous time setting. The core is that any node can propose a consensus message that is reliably propagated to all nodes through RBC, and in order to reduce the broadcast bandwidth, the message is divided into multiple shares by erasure coding. At the same time, all nodes interact with each other through BA protocol. The execution of BA protocol has exponential probability of converging to 1 or 0, indicating acceptance or rejection of the proposed message. Each phase, all nodes propose transactions and make consensus in parallel, and finally get a subset of the

transaction as a consensus result. Therefore, an attack on a single node will not cause the entire network to crash, and only affect the performance of the network. The disadvantage of this protocol is that it needs to predetermine the collection of nodes. Nodes cannot enter and exit freely, and large-scale nodes cannot be supported.

In order to solve the problem of free entrance and exit of nodes on permissionless blockchain, Algorand[5] proposed to combine VRF(Verifiable Random Function)[6] and threshold voting with BA. Regardless of the number of nodes, a certain number of nodes are randomly selected based on the number of stakes held, and then the nodes vote transactions through the BA, and a consensus is reached on the transactions issued by the node with the highest priority (the VRF random number is the lowest). In order to improve security, each step of the consensus selects a new set of nodes, so that the attacker can not predict the next attack target. The disadvantage is that it cannot achieve high throughput with low bandwidth.

Dfinity[7] combined random beacon and Notaries to make each phase of members randomly selected. However, a potential economic game theory is that the threshold signature can be predicted by the collusion of multiple members. Collusion members can collaboratively calculate the group private key and quickly predict the next phase of random numbers. By DOS attacks on the proposers of the next phase, the conspirators can undermine the fairness of the network.

1.1 Contribution

Ultrain consensus has the following features

- Tamper-proof hardware fingerprints which is based on computing power score and historical behaviors rating, to guarantee that the participating nodes have high performance and good reputation.
- Improved efficiency of consensus by utilizing the parallelism in the Byzantine consensus process, and improved robustness against malicious proposers
- Optimized cryptographic primitives using lightweight cryptography [8, 9] to reduce the computation overhead
- Divided message through redundant encoding [10], instead of message broadcasting, which ensures that nodes can use the limited network bandwidth to broadcast maximum quantity of messages and increase the consensus throughput rate.
- Threshold signature [11] to group multiple nodes into a group, for better robustness, fairness and response speed against a single node.
- Threshold encryption [12] to ensure that each node cannot know the content of the message until it receives enough shares, for unbiased delivery of consensus messages.

2 Glossary

2.1 Signature Algorithm

Ultrain chooses ED25519[13] based on the following considerations

- ED25519 is an open source cryptographic algorithm proposed by Daniel Bernstein. After numerous cryptographic expert examinations, it was finally selected into the RFC7748 standard. It was also considered as an improvement in Bitcoin and Ethereum.

- ED25519's twisted Edwards curve is a relatively large performance improvement over the NIST P-256 curve's Weistrass curve, with high verification speed, which reduces nodes' computation overhead.
- ED25519 has available efficient and secure open source algorithm implementations, and the assembly implementation supported by the Intel AVX instruction set.

In the following mathematical expression,

- signature is denoted as: $SIG_{sk}(m) = ED25519_{sig}(sk, m)$
- verification is denoted as: $VER_{pk}(SIG_{sk}(m), m) = ED25519_{ver}(pk, SIG_{sk}(m), m)$

Where (sk, pk) is a pair of private key and public key.

2.2 Hash Function

Ultrain takes SHA256 as the hash function based on the following considerations:

- Although the SHA3 standard has been proposed, currently there is no apparent attack against SHA256.
- SHA256 performance is relatively high compared to SHA3, and there are many stable open source implementations.
- SHA256 has available hardware accelerators than SHA3 or other digest algorithms.

In the following mathematical expression, hash function is denoted as: $SHA(m) : hash = SHA256(m)$

2.3 Verifiable Random Functions

Verifiable random function requires the following properties:

- Deterministic, otherwise the node can keep trying new random numbers until the VRF's result is beneficial to itself.
- Non-malleable, once the signature result can be transformed into a different but valid one, the node can modify the result to achieve the desired result.

In the following mathematical expression, VRF is denoted as: $proof = VRF_{sk}(seed)$

2.4 Quantum Resistance

Quantum computers have a major impact on asymmetric algorithms and key exchanges in cryptography. For example, the problem of large number decomposition which RSA is based on can be efficiently solved using Schor's algorithm, and the same as elliptic curve discrete logarithm (eg, GEECM). However, it has less effect on symmetric algorithms and one-way functions, which only needs to double the security level. At present, Ultrain doesn't consider quantum-resistant algorithms, based on the following considerations:

- The development of quantum computers still has a considerable length of time [14]. Some of the key issues are the superposition of multiple quantums and quantum error correction.
- The current standards for quantum-resistant algorithms such as key exchange and asymmetric algorithms are not yet mature, and NIST's first phase of competition has just been completed.

- Quantum-resistant algorithms currently are based on LWE[15] or code[16]. The efficiency is generally low, and the quantitative assessment of security is currently inconclusive (such as more efficient CVP and SVP solutions [17]).
- Algorithms based on traditional difficulty problems, such as threshold signatures, threshold encryptions, ABE[18]/IBE[19] and zero-knowledge proof[20], are difficult or inefficient to construct with quantum-resistant problems.
- The migration to quantum-resistant wallets is not difficult and can be done at the dawn of quantum computers. Merkle tree, certificates, etc can be migrated with zero knowledge proofs.

2.5 Block

The block consists of the following structure

- block height, abbreviated as bh
- The header of the previous block, denoted as $hprev_{bh}$.
- the merkle root of transactions, denoted as $hroot_{bh}$.
- The merkle root of the world state after execution, denoted as $hstate_{bh}$.
- The merkle root of proposers, denoted as $hProposer_{bh}$.

The overall block format is

$$(bh, hprev_{bh}, hroot_{bh}, hstate_{bh}, hProposer_{bh}) \tag{1}$$

Considering that the Genesis block is special, we define it as

$$(0, 0, hroot_0, hstate_0, 0) \tag{2}$$

$hstate_0$ contains accounts and stake distribution of initial members.

2.6 Node Identity

During the operation of the blockchain system, any node belongs to one of the following roles:

- Proposer: propose transaction batches to be packed into the next block
- Voter: Confirm the transaction and vote to approve
- Listener: Help deliver messages

All nodes independently produce the next block.

2.7 Message Type

During the consensus procedure, there are two types of messages:

- PROPOSE: Proposers(each with a key pair (skp, pkp)) propose transactions to be packed in the next block, in the format of

$$(PROPOSE, shard, txs, bh, hprev_{bh}, phase, txshash, proof_{pkp}, SIG_{skp}, pkp) \tag{3}$$

where $shard$ is the shard which the proposer belongs to, txs is a transaction batch, $txshash = SHA(txs)$, $SIG_{skp} = SIG_{skp}(shard||hprev_{bh}||bh||phase||txshash)$, and $proof_{pkp}$ is the proof that the Proposer is selected.

- ECHO: Voters(each with a key pair (skv, pkv)) vote for proposed transaction batches, in the format of

$$(ECHO, shard, bh, phase, txshash, proof_{pkp}, SIG_{skp}, pkp, proof_{pkv}, SIG_{skv}, pkv) \quad (4)$$

where $SIG_{skv} = SIG_{skv}(bh||phase||txshash)$ is the vote. $proof_{pkv}$ is the proof that the Voter is selected. The rest are the same as above.

3 Architecture

Each phase of Ultrain consensus is divided into the following steps

1. Based on hardware fingerprints and stakes, use verifiable random function (VRF) to select a certain number of Proposers and Voters.
2. Proposers propose transaction batches that need to be packed in the block. Voters confirms with each other for which transaction batch to accept.
3. Nodes executes the transaction batch and then use VRF to select a new set of Voters to synchronize the execution results with BA (binary agreement) protocol.
4. Nodes pack the transaction batch and the post-execution state independently into the next block, then start consensus for the next block.

In order to achieve parallel execution and reduce node resource consumption, Ultrain uses sharding technology to randomly split nodes into different shards and cross-shards so that when nodes processes its shard, it asynchronously accepts the status from other shards, and finally it will packs information from all shards into one block.

3.1 Identity Layer

Each node that participates in consensus has an onchain account with a key pair (sk, pk). The probability that a node is selected as a consensus committee member depends on the combined score of the following three ratings.

- Stake on the chain. All assets corresponding to the public key can be queried on the blockchain, so the percentage of stakes held is w_{stake}. Some nodes may hold a certain amount of stakes, but they are reluctant to participate in consensus, so they can delegate to other reliable hardware providers, similar to DPOS. Besides, a node holding a large number of stakes may distribute stakes among multiple accounts, which will not affect its selected probability, because the probability of selection of multiple accounts is proportional to total number of stakes (assuming that the hardware device of all accounts are the same). This score is set to w_{stake}.
- Hardware features. For each device voluntarily participating in the consensus, the hardware characteristic score can be obtained. This score may have multiple dimensions, such as CPU/memory/disk, which are suitable for different roles in the committee. This score is set to $w_{hardware}$.
- Reputation. According to the node account's age (block height) of the stake, the historical participation record, the system automatically calculates the reputation. The score is set to $w_{reputation}$.

Each public key corresponds to a weight of $w = r_1 * w_{stake} + r_2 * w_{hardware} + r_3 * w_{reputation}$, where r_1, r_2, r_3 are system constants used to set each weight ratio. Other weight calculation method are also under examination.

3.2 VRF Layer

Ultrain chooses private random number generation, rather than public methods as Dfinity or Randao, for the following reasons.

- Public selection with static threshold signatures, if enough nodes collude, or a user who controls a large number of stakes entries, the next phase of random numbers can be predicted. Besides, all selected roles are publicly revealed, so there is a time window for attackers.
- This method has a relatively high delay, since a sufficient number of node responses are required to determine the next random number. The network traffic may be a problem.
- The threshold based method uses BLS algorithm based on the elliptic curve pairing, which is relatively slow compared with the general elliptic curve operations such as point multiplication. RSA based threshold signatures requires node interactions.

In order to ensure the correctness of VRF layer, Ultrain's VRF function is designed to meet the following characteristics:

- Each node that holds a stake amount of s, through the VRF election mechanism, has a probability $p(s)$ of being selected. VRF generates deterministic and unpredictable (same effect as random oracle access) proofs for such election procedure.
- For any number s_x, s_y, $p(s_x) + p(s_y) = p(s_x + s_y)$. This is designed to ensure fairness and prevent sybil attacks, which means that a stake holder cannot obtain a greater probability of being selected by splitting into multiple identities that hold a small amount of stakes. This also prevents that people with less than 1/3 stakes collaborate to obtain more than 1/3 of the voting probability, which affects the security of the system.
- The system can control the number of selected nodes through static or dynamic parameter settings.

A straightforward example is $p(s) = B^{-1}(r; s, \lambda)$, where s is the number of stakes held by the node and $\lambda \in [0, 1]$ is the ratio of selected members, $r \in [0, 1]$ is a random number, and B is a binomial distribution. Assuming r is a uniformly distributed, the expectation $E(p(s)) = s * \lambda$.

Since $B(v_1 : s_1, \lambda) + B(v_2 : s_2, \lambda) = B(v_1 + v_2 : s_1 + s_2, \lambda)$, if the stake s is splitted into two shares s_1 and s_2, the expected voting rights are $E(p(s_1)) = s_1 * \lambda$ and $E(p(s_2)) = s_2 * \lambda$ respectively, $s_1 * \lambda + s_2 * \lambda = s * \lambda$, so it is the same as that a node with the stake s being sampled individually.

3.3 Consensus Layer

3.3.1 Sharding

Considering the consensus process, except for PROPOSE message, ECHO message is small, but it takes time to propagate. In order to make full use of bandwidth, Ultrain consensus has 16 shards running at the same time, and each shard contains different transaction types or transactions from different accounts. Information from all shards are aggregated into one block at the end of consensus phases.

- For a node that only participates in $shard_i$, it may not be able to verify the status of other shard. In this case, it only needs to monitor the messages in the network and accept the other shard's concordal execution results.

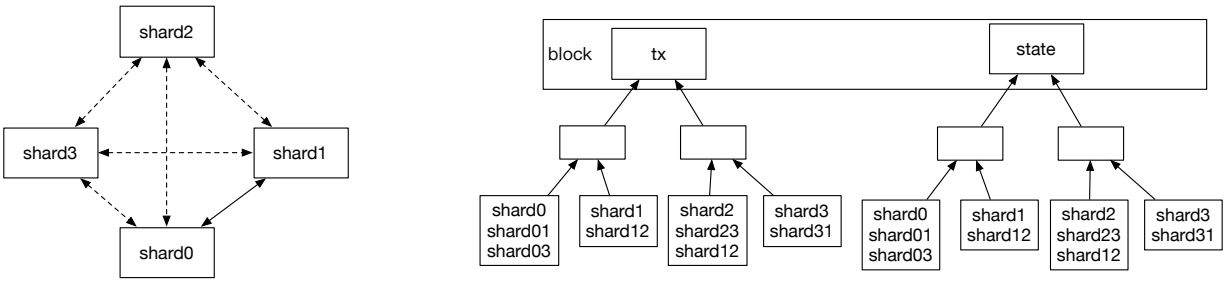


Figure 1: store and execute shard

- Of course, for contract execution that requires two shards, we allocate the C_N^2 cross-span shard execution transaction evenly to different shards in the network, so each shard executes two shards and C_N^2/N cross-shard's transaction, such that the node needs to store approximately $((C_N^2)/N + N)/N$ shard states.
- A transaction is not allowed to cross multiple shards. Even if supported, such transactions need to be completed in multiple consensus cycles, and each consensus cycle completes a cross shard operation. For ease of use, basic contracts should be included in each shard, and cross shard transactions are expensive and called as less as possible.

Nodes are randomly assigned to a shard. If shards are freely selected, a node holding a small number of stakes may concentrate on a shard[21] and easily gains control over it.

The behavior in one shard is described as follows.

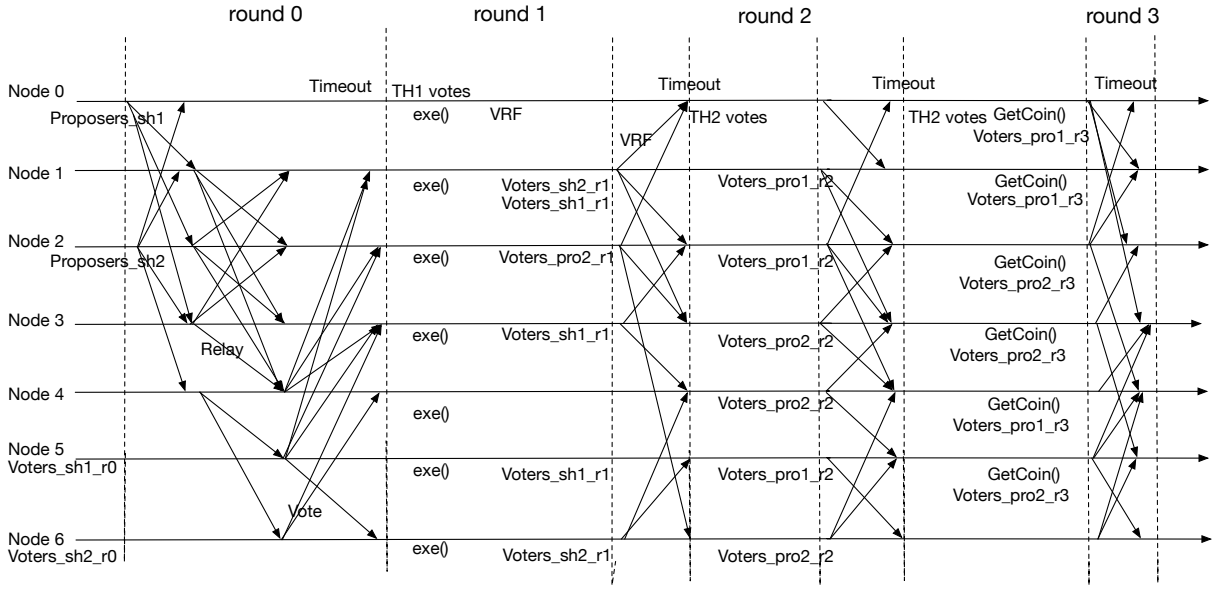


Figure 2: consensus overview

The behavior of nodes in one shard is described as follows.

- Propose phase (BA0). Based on the previous block header, each node independently determines whether it is selected as Proposer or Voter. This phase ensures that the proposed batch transactions are delivered to the majority of nodes.
  - Proposer broadcasts PROPOSE with batched transactions and ECHO with transaction hash to speed up the propagation. To minimize bandwidth consumption, PROPOSE

messages are encoded with erasure coding and sent to multiple nodes. Each Proposer can only broadcast once, and multiple broadcasts are considered as malicious nodes.

  - Voter waits for T_{PROP} time, and checks all received PROPOSE messages, then votes for the one with minimum proof in the format of ECHO message.
  - After receiving TH_1 messages, Voter broadcasts ECHO messages even if it does not receive a PROPOSE message, and ask for corresponding PROPOSE from neighbors.
  - All nodes finish this phase after receiving TH_2 ECHO or $TIMEOUT$.

- Execute phase. Each node executes based on the current state $state_1$ and the output of its selected transaction batch, and gets the output state $state_2$.

- BA phase, This phase ensures that honest nodes agree on whether or not to accept the execution result.

  - All nodes independently determine whether it is selected as Voter. If so, voters broadcasts an ECHO message with the digest of $state_2$.
  - Nodes finish this phase after receiving TH_2 ECHO, or timeout and return empty result.
  - If the last phase times out, a new phase begins, each node re-judges whether it is a voter. If so, determine whether to raise an ECHO message containing the digest of $state_2$ or an empty message based on CommonCoin.
  - All nodes finish this phase after receiving TH_2 ECHO. Otherwise, continue to use CommonCoin to determine the next phase's ECHO message content.
  - All nodes terminate the BA with empty message, if the maximum BA phases is reached. All nodes roll back to $state_1$.

- Produce block phase, all nodes combine its shard's execution result $state_2$ (or $state_1$ if timeout) with other shards's results to produce one block.

Throughout the consensus process, Listeners will receive the same message as Voter and Proposer, so in addition to relaying messages, Listener will also check, process messages and produce blocks.

The specific algorithm is described as follows:

Algorithm 2: Ultrain node operation

Data: Initial stake distribution Istake, member list Imember
Result: blockchain status

Initialization
  $bh = 1$ /* initial block height */
  $state_{bh} = (Istake, Imember)$ /* initial state and block */
  $N_{shard} = 16$ /* shard number */
  $(sk_i, pk_i) = KeyGen()$ /* Node Identity */
  $queue_{txs} = set_{EMPTY}$ /* initial transaction queue is empty */
Upon incoming transaction tx
  /* Check the signature and format of the transaction message */
  if *check(tx)* **then**
    $queue_{txs} \cup = tx$
Upon start consensus
  /* last block */
  $bh++$;
  $phase = 0$;
  /* Propose phase */
  $res, hbatchtx, batchtx =$
  $phaseops(PROPOSE, hblock_{bh-1}, shard, queue_{shard1}, block_{r-1}, phase)$
  /* BA */
  $phase++$;
  if $res == TIMEOUT$ **then**
    /* If timeout and insufficient votes collected in the last phase, propose empty message */
    $res = phaseops(ECHO, shard, set_{EMPTY}, block_{bh-1}, phase)$
  else
    /* Save the previous state $state_{bh-1}$ */
    $save(state_{bh-1})$
    /* Execute the batch batchx transaction, get a new state, calculate the summary as $hstate_{bh}$ */
    $hstate_{bh} = exe(state_{bh-1}, batchtx)$ exe(batchtx)
    $res = phaseops(ECHO, shard, hbatchtx, block_{bh-1}, phase)$
  while $res == TIMEOUT \& phase < MAX_r$ **do**
    /* at most $MAX_r = 7$ phases */
    $phase++$
    $CC = CommonCoin()$
    /* take the lowest bit of CC to decide the next step */
    if *LSB(CC)==1* **then**
      $res = phaseops(ECHO, shard, hbatchtx, block_{bh-1}, phase)$
    else
      $res = phaseops(ECHO, shard, set_{EMPTY}, block_{bh-1}, phase)$
  if $res == TIMEOUT$ **then**
    roll back to $state_{bh-1}$
  /* collects tx root and state root for other shards, builds blocks */
  $build\_block(state_{bh}, hstate_{othershards}, hbatchtx_{othershards})$

The response of each node to the message is as follows:

Algorithm 3: Ultrain node phase operation *phaseops*

Data: *Type*, *shard*, transaction batch *batchtx*, block summary $block_{bh-1}$, *phase*
Result: return value res, transaction batch summary hbatchtx, transaction batchatchx

initialization
  $set_{ECHO} = emtpy$ `/* response hbatchtx */`
  $set_{txs} = empty$ `/* HASH(batchtxs) : batchtxs */`
  $sent_{ECHO} = empty$ `/* sent ECHO HASH(batchtxs) : batchtxs */`
Upon Start
  start timing
  $proof = VRF_{sk}(block_{bh-1} \parallel phase \parallel role)$
  `/* judged which shard is selected by the lower 4 bits of proof */`
  $shard = proof[3:0]$
  `/* Selected or not */`
  if $proof[193:256] < score$ **then**
    $sig = SIG_{sk}(block_{bh-1}, phase, hbatchtxs)$
    if $phase! = 0$ `/* BA */`
    then
      broadcast (ECHO, *shard*, height, phase, batchtxs, proof, sig, pk)
      break
    hbatchtxs = SHA(batchtxs)
    broadcast (PROPOSE, *shard*, height, phase, hbatchtxs, batchtxs, proof, sig, pk)
    `/* Proposer is also voter by default, sending ECHO to accelerate message propagation */`
    broadcast (ECHO, *shard*, height, phase, hbatchtxs, proof,sig, pk)
Upon receive PROPOSE
  if *!check(PROPOSE))* **then**
    break
  if $PROPOSE.proof > MIN_{proof}$ **then**
    break `/* received better PROPOSE */`
  `/* Have not broadcasted ECHO */`
  broadcast ECHO
Upon receive ECHO
  if *check(ECHO)* **then**
    $set_{ECHO}[ECHO.hbatchtxs] \cup = ECHO.pk$
    if $count(set_{ECHO}[ECHO.hbatchtxs]) == TH_2$ **then**
      if $phase! = 0 \parallel ECHO.hbatchtxs \in index(set_{txs})$ **then**
        return hbatchtxs
      `/* wait for some time to get the batchtxs */`
    if $count(set_{ECHO}[ECHO.hbatchtxs]) == TH_1$ **then**
      if *not* $ECHO.hbatchtxs \in sent_{ECHO}$ **then**
        add_sig_then_send(ECHO, ECHO={ECHO, hbatchtxs, sig_{msg}, proof, pk)}, sk, pk)
        $sent_{ECHO} \cup = ECHO.hbatchtxs$
Upon TIMEOUT
  return TIMEOUT

Algorithm 4: messaging

Data: msg_{in}, sk, pk

1 /* add node signature to original message */

2 $broadcast(Type, msg_{in}.msg, msg_{in}.sig \cup SIG_{sk}(msg_{in}.msg), msg_{in}.proof, msg.pk \cup pk)$

Algorithm 5: message verification $check(msg)$

Data: $msg_{in}, sk, pk, block_{prev}, shard, role$

Result: verification result res

1 **if** $msg_{in}.proof[193:256] >= score$ **then**

2 | Return 0

3 **if** $!VER_{msg_{in}.pk}(proof, block_{prev} \parallel shard \parallel role)$ **then**

4 | return 0

5 **if** $!VER_{msg_{in}.pk}(msg_{in}.sig, msg_{in}.msg)$ **then**

6 | return 0

7 return 1

The preservation of certificates has a greater impact on the synchronization of newly added nodes. We consider using the following approach:

- Aggregate signatures. Save the signatures of multiple messages in a storage space of $O(1)$ aggregation method [22] such as BLS.
- Using DAG method, any node must verify existing two signatures. Then as long as the DAG's root is locked, the certificate chain can be determined.

In addition, in order to achieve sharding, the underlying p2p network also needs to be modified. In our protocol, the node only needs to receive messages from the selected shard and asynchronously accept the consensus results of other shards.

3.3.2 Transaction confirmation

Traditional consensus systems perform executions in the following ways:

- (tx execution + proposal + tx verification + BA), the delay is relatively large, because the verification of the execution result is equivalent to repeated execution, and the total delay contains two execution processes. The advantage is that the blockchain can avoid packing illegal transactions into the block. PoW-based public chain generally adopts similar form.
- (tx proposal + BA+tx execution) avoids packaged illegal transactions into the block as well. However, there is a stronger demand for the determinism of smart contract execution.
- (tx partial execution + consensus ordering + consensus state verification and transfer) This is a hyperledger fabric approach. The advantage is that the non-deterministic part is placed before the consensus and the deterministic verification signature and status transfer are executed concurrently. The disadvantage is that nodes do not verify the execution process and rely on signature verification by parties affected by the state transfer.

In order to reduce the delay of the node, we consider the following ways:

- (multiple phases of tx consensus + one phase of execution results consensus). This may allow multiple transactions to be executed in batches with a certain degree of efficiency improvement. The honest client only needs to receive the receipt that its transaction has been packed into the block, and have enough confidence in the execution result, without having to wait until the execution result is packed into blocks.

- (tx Consensus + Consensus of previous execution results). Consensus and execution of last phase consensus are independent and run in parallel. The disadvantage is that it may pack illegal transactions into blocks and wastes block storage.
- (tx proposal + voter executes + voter to execute result hash response + vote on execution result). The delay can be minimized. The advantage is that illegal transactions can be avoided in the block while identifying which transaction batches are not deterministically executed because these transaction batches do not collect enough votes. From an economic game standpoint, proposers will not intentionally submit a non-deterministic process in order to obtain profits.

Ultrain consensus adopts the third approach, considering the design of the upper-level smart contract.

3.4 new node

New nodes need to go through the following steps:

- Observe and receive messages that the network is spreading, determine the block height and consensus phases, and the current block consensus results. To prevent attacks by malicious nodes through network control, new nodes need to listen to consensus messages from multiple network addresses.
- Based on the current block, pulls the historical block from other nodes and verifies the correctness of the chain. Considering the unidirectionality of the chain, new nodes can be confident in the correctness of the chain as long as the header is correct.

4 Security Analysis

4.1 Theorem

Theorem 4.1 (FLP Impossibility[23]) *In a fully asynchronous system, within bounded time, no consensus protocol is totally correct in spite of one fault.*

Theorem 4.2 (Fault tolerance upper bound[24]) *There is an BFT algorithm, for any m malicious nodes, if there are total more than $3m$ nodes.*

Theorem 4.3 (CAP Theorem[25]) *It is impossible in the asynchronous network model to implement a read/write data object that guarantees Availability and Atomic consistency in in all fair executions (including those in which messages are lost).*

4.2 Assumptions

Assumption 4.4 (Network Connectivity) *Any two peer to peer communication will be done through the gossip protocol to ensure that the message reaches with overwhelming probability within a certain time window. At the same time, the origin of the message will not be easily identified and attacked.*

Assumption 4.5 (Malicious Nodes) *A node can send any malicious message or conspire or reject to response.*

Assumption 4.6 (VRF algorithm) *The VRF algorithm guarantees that a stake with a ratio of x obtains a voting power that is proportional to x.*

Assumption 4.7 (Local clock) *All local clocks run at the same speed with a certain amount of jitter and skew. Each node determines when to enter the next phase based on depends on local clock and messages. The local clock may be coordinatted with trusted network clock service occasionally to make the system more robust.*

4.3 Corollaries

Although Byzantine consensus can guarantee the ultimate consistency of the entire network system, because Ultrain is running on the public network, the situation is rather special. In order to meet the scalability requirements, use (committee+threshold) method instead of the traditional message that every node receives all other nodes's messages via p2p communication. The transmission of information between nodes is not p2p transmission, but through the gossip network.

A malicious node may attack in the following two ways:

- Case1: As a Proposer, send a PROPOSE message to one part of the network, delay sending it to another node, or not send it at all.
- Case2: As a Voter, send an ECHO message to one part of the network, delay sending to another node, or not send at all.

Corollary 4.7.1 *Case1 and Case2 will not affect the consistency and livness of the chain.*

Proof: VRF guarantees the unpredictability and randomness of voters' geographical and identity distribution, so if the PROPOSE message does not propagate over the network evenly above a threshold, most nodes will not be able to receive sufficient ECHO messages within a timeout window, resulting in an empty transaction batch and enter the second phase; for a small number of nodes that can receive ECHO messages within the timeout, in the second phase will not be able to get enough ECHO messages corresponding to the transaction hash, and will terminate with empty message. So all the nodes will end with an empty transaction batch. The consistency of the network is guaranteed. In the same way, if the Proposer broadcasts different PROPOSE messages to different network partitions, it will cause no transaction batch to receive sufficient ECHO messages. Finally, all nodes end with empty transaction batches. Besides, VRF will assign enough Proposers randomly to different shards, so as to ensure that not all Proposers are malicious nodes with a high probability and the system will eventually proceed.

Corollary 4.7.2 *In a strongly synchronized network, the Ultrain consensus is completed after the second phase; In semi-synchronous network, Ultrain needs multiple phases to converge with overwhelming probability.*

Proof: In a strongly synchronized network, any node can receive messages from other nodes within the timeout period. Based on the assumption that the number of malicious votes in the network will not exceed 1/3, all nodes in the first phase can receive all consensus proposals, whether sent by malicious or honest nodes. In the second phase of the process, only the honest node's proposal will receive enough votes and all nodes will perform deterministically and achieve the same result. In a semi-synchronous network, considering the randomized BA algorithm ends with high probability with the increase of phases, even if the attackers try to mislead by sending malicious messages, all nodes can converge to a consistent result with a large probability within finite phases.

Corollary 4.7.3 *Ultrain can defend against sybil attacks.*

Sybil attack is mainly the use of systemic unfairness to seek profit. Ultrain uses VRF functions to ensure that even if the user is split into multiple pseudo-identities, it cannot generate voting rights that exceed the total holding stakes, so sybil attacks cannot work. For attacks that use multiple pseudonyms to construct multiple pseudo-identities that correspond to a real identity, Ultrain uses hardware fingerprinting to uniquely label the stake holders on the network.

Corollary 4.7.4 *Ultrain can defend against the Selfish-mining attack*

The BFT system guarantees the consistency of all nodes. Therefore, even if an external node generates a chain independently, it will not be accepted by the network honest node and thus cannot affect the consistency of the system.

Corollary 4.7.5 *Ultrain can withstand Nothing-at-stake attacks.*

The voting ability of each stake holder is proportional to the stake, and less than 1/3 of the holders will not be able to produce a fork. Due to the uniqueness of the hardware fingerprint, Ultrain will also impose a certain percentage of penalty on the double-voting nodes.

For DoS and DDoS attacks, DDOS makes the attacker's server not work by consuming resources, and DOS directly exploits the loopholes. Even if there is a certain bug in the consensus code, it is difficult for the system to be attacked and crashed, because the cost of attack is $o(n^2)$, so the cost of attacking multiple nodes at the same time is very high. Besides, based on gossip, it is more difficult to get IP address revealed for attack. If the consensus coding is running in the sandbox, the signature is on the outside, or SGX, even if the implementation has a bug that does not affect the security of a single node, such as stakes being spent; the account should include different vote key and wallet key; sandbox will not let other modules of the system be contaminated, so we need to isolate the user's wallet and consensus code, especially for PoS scenarios, can not use the same account. If enough time window to start quickly, the entire system can recover quickly.

Besides, network partition will cause the availability and consistency of the blockchain system not satisfied at the same time. In general, blockchain systems choose to halt or fork (for example, Bitcoin). In order to discover the partition in time, our VRF function considers sampling in different regions, such as requiring that the voter must have a certain proportion in different countries, such as China, Europe and US.

5 Summary

This document describes Ultrain's consensus algorithm. In the future, more PoS and PoW details may be added as supplements to increase the security level. Besides, to assist sharding, receipts and gas mechanism will also be taken into consideration.

References

[1] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[3] Jae Kwon. Tendermint: Consensus without mining. *URL http://tendermint. com/docs/tendermint {_} v04. pdf*, 2014.

[4] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.

[5] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. SOSP, 2017.

[6] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 120–130. IEEE, 1999.

[7] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series consensus system, 2018.

[8] Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varıcı, and Ingrid Verbauwhede. Spongent: A lightweight hash function. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 312–325. Springer, 2011.

[9] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A lightweight hash. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 1–15. Springer, 2010.

[10] Yong Wang, Sushant Jain, Margaret Martonosi, and Kevin Fall. Erasure-coding based routing for opportunistic networks. In *Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, pages 229–236. ACM, 2005.

[11] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2003.

[12] Dan Boneh, Xavier Boyen, and Shai Halevi. Chosen ciphertext secure public key threshold encryption without random oracles. In *Cryptographers' Track at the RSA Conference*, pages 226–243. Springer, 2006.

[13] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[14] Scott Aaronson. The limits of quantum computers. *Scientific American*, 298(3):62–69, 2008.

[15] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange-a new hope. In *USENIX Security Symposium*, pages 327–343, 2016.

[16] Pierre-Louis Cayrel and Mohammed Meziani. Post-quantum cryptography: code-based signatures. In *Advances in Computer Science and Information Technology*, pages 82–99. Springer, 2010.

[17] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In *Cryptographers' Track at the RSA Conference*, pages 319–339. Springer, 2011.

[18] Brent Waters. Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization. In *International Workshop on Public Key Cryptography*, pages 53–70. Springer, 2011.

[19] Dan Boneh and Matthew Franklin. Identity-based encryption from the weil pairing. *SIAM journal on computing*, 32(3):586–615, 2003.

[20] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 238–252. IEEE, 2013.

[21] Sharding attack. http://www.mangoresearch.co/1-shard-attack-explained-ethereum-sharding-contd/.

[22] Dan Boneh, Craig Gentry, Ben Lynn, Hovav Shacham, et al. A survey of two signature aggregation techniques. *RSA cryptobytes*, 6(2):1–10, 2003.

[23] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[24] Leslie Lamport. The weak byzantine generals problem. *Journal of the ACM (JACM)*, 30(3):668–676, 1983.

[25] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.