ONTOLOGY SHARDING

DRAFT v0.1

ONTOLOGY RESEARCH TEAM
RESEARCH@ONT.IO

Abstract. The performance of blockchain has become a great concern. How to make blockchain networks scalable is one of the hot spots in current blockchain research. We propose Ontology Sharding protocol, which adopts a hierarchical network architecture. In Ontology Sharding, shards communicate with each other through reliable message queues, and the verifiability of messages between shards is achieved with Merkle proof. At the same time, based on Multi-Version Commitment Ordering, the transactionality of cross-shard smart contract transactions is realized, which provides a strong and consistent transaction model.

Keywords. Blockchain; sharding; cross-shard communication; cross-shard transactionality

1. Introduction

Blockchain and smart contract platforms are becoming a new generation of globally shared computing platforms. Although currently the most prominent application of blockchain is cryptocurrency, people have started building new generation decentralized applications based on blockchain.

Along with an increase of daily transactions and decentralized applications, the scalability issue of current blockchains has emerged. Before being a decentralized global computer, the blockchain platform should have competitive transaction processing performance. Visa is currently the fastest known global payment network, capable of processing approximately 2,400 payments per second. In contrast, the average transaction speed of Bitcoin [19] is about 7 Tx/s, and the average transaction speed of Ethereum [11] is about 20 Tx/s. Both of them are far lower than the expected performance. This problem has seriously hampered the possibility that the blockchain platform become a mainstream payment solution in the near future.

The scalability problems of current blockchain platforms are mainly revealed in the following three aspects:

(1) Block generation. A blockchain is very similar to a singly linked list, each block in the blockchain has a hash of its previous block, which can be thought of as a pointer to the previous block. In current mainstream blockchains, it normally reaches the consensus on only one block in each consensus round and only a few thousand transaction requests can be saved in one block. To achieve a high transaction performance, blocks need to be consensused in a very short time slot. Reaching the consensus on multiple blocks per second, such as ten blocks per second, is a big challenge for blockchain networks on a global scale. The main obstacle of this challenge stems from the global geographical distance from the limitations of message propagation delays on the network.

(2) Data capacity of a block. Currently, in all blockchain protocols, each node has to store all states (account balance, contract code, storage, etc.) and processes all transactions. This provides a lot of security, but it greatly limits scalability, that is, blockchains cannot handle more transactions than a single node. In addition, the blockchain must maintain its own functionalities, such as tamper-proof and traceability, then the data that each node needs to store will continue to increase. Therefore, the blockchain nodes will face higher and higher data storage pressure.

(3) Cost of using blockchain network. With the increase of users and applications of blockchain, blockchain nodes need to process more transactions in network and save more blockchain state data. All transaction requests require fees, and blockchain nodes tend to prioritize transactions that pay higher fees. consequently, during the peak hours, if transactions need to be verified on time, it has to pay higher

fee. This also restricts the use of current blockchain networks.

Blockchain developers have done lots of research and attempts to overcome the above challenges and improve the scalability of blockchains. For example, Bitcoin tries to improve performance by enlarging block size and EOS [10] improves consensus efficiency and mitigates storage pressure through partial centralization.

High scalability of blockchain should make no compromises on decentralization and security. Scalability solutions with off-chain approaches may bring their own security risks into the blockchain system. From the perspective of ensuring security of blockchain applications [9] and the development of blockchain architecture in the long run, blockchain scalability should be implemented at the base layer.

Sharding is regarded as a powerful blockchain scalability solutions. There exist two kinds of sharding systems, that is, state sharding and transaction processing sharding.

State sharding is to shard data of a blockchain, including smart contract state data and blocks. When updating blockchain data, only a subset of shard nodes need to update their data according to the sharding algorithm. State sharding provides storage scalability for blockchains. If the blockchain sharding design only supports state sharding, it can only provide scalability of dealing with asset transactions [22]. However, in the case of smart contract transactions, it has to ensure that state data of a smart contract is saved in the same shard, otherwise cross-sharding transactions will be required to ensure the consistency of the smart contract state data.

Transaction processing sharding means that different transactions can be processed in different shards concurrently, and each shard may have its own independent consensus protocol. Transaction sharding provides consensus scalability for blockchains. Transaction sharding can be achieved through smart contract sharding, i.e., the smart contract is run in shard, and all transactions that call this smart contract need to be submitted to this shard. Using transaction sharding, the efficiency of transaction processing in entire blockchain network can be scaled out.

To achieve continuous scalability, it should support scale-out. There are two ways to horizontally scale a shard, one is to split one shard into two shards, and the other is to build child shards based on current shards.

Both of these two methods can achieve scalability of shards, but they bring certain limitations. The first way makes all the shards at the same level, that is, all shards are based on the main chain, which can greatly guarantee the security of shards. However, some smart contracts deployed in the original shard should be split according to their functionalities. The split smart contracts have to be migrated to new shards and will only be able to access resources of the original shard by means of cross-shard communication. Excessive cross-sharding communication will greatly reduce the scalability of the shard. The second way is to build a hierarchical sharding network. In this way, new shards can be built based on the original shard, and smart contracts in new shards will be able to directly access resources of the original shard. In addition, in this hierarchical sharding method, shards can be configured to use consensus protocols different from that of original shard.

1.1. **Overview of Ontology Sharding.**

Ontology network implements blockchain scalability with sharding. The goal of Ontology Sharding includes:

- Performance should be linearly scalable;
- Transactions inside a shard are only processed by nodes in this shard;
- Atomic cross-shard transaction processing;
- Uniformed incentive mechanism in all shards.

The organization of Ontology Sharding network is shown in Figure-1. The root shard is Ontology main chain, which is responsible for managing all shards at the first layer. Every shard can have its own child shards. Shards can communicate with sibling shards directly.

Since most services in Ontology are implemented with smart contracts, Ontology Sharding uses smart contracts as the basic sharding unit, while supporting transaction sharding and state sharding. The smart contracts in Ontology Sharding network run in one shard, and the transaction and state data for the smart contract will only be processed and saved in this shard.

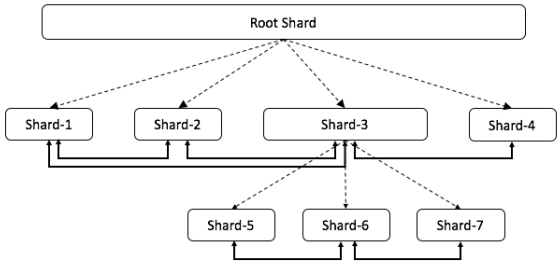

FIGURE 1. Ontology Sharding

The Ontology Sharding network uses Verifiable Byzantine Fault Tolerance with Randomness (VBFT) consensus algorithm as Ontology main chain. VBFT combines Proof-of-Stake (PoS), Verifiable Random Function (VRF) and Byzantine Fault Tolerance (BFT). It guarantees randomness and fairness of consensus group generation through VRF, while ensuring fast state finality.

The block in a shard contains link information to its parent shard block and message queue information with its sibling shards. The block relationship between shard and parent shard is shown in Figure-2.

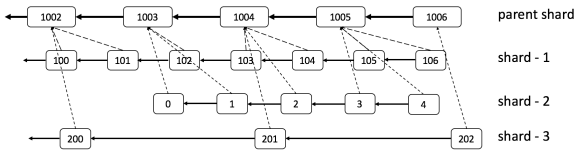

FIGURE 2. Blockchains of Ontology shards

As can be seen from Figure-2, although blocks of shards include block information of their parent shard, consensus performance of the shard is not bound to the parent shard. In addition, transactions in a shard will only get consensus within this shard, that is, transaction processing performance of a shard depends only on the node performance in this shard and the network performance of this shard. This ensures linearly scalability of Ontology sharing network.

1.2. **Organization.** The rest of the paper is organized as follows. In Section 2, we review some basic concepts and assumptions which are necessary for the understanding of the rest paper. In Section 3, we present some definitions and models of our sharding system. Section 4 introduces the management of shards, including shard creation, activation and deactivation. Section 5 describes the communication system among shards. We present cross-shard transaction processing of Ontology Sharding and give Scalability analysis of the cross-sharding transaction processing in Section 6. Cross-shard transaction fee processing is proposed in Section 7. We conclude the paper in Section 8.

2. Preliminaries

We begin by defining some notations that will be used throughout this paper. Denote by $\mathcal{S}$ a shard with identity number s. Denote by $\mathcal{S}_p$ the parent shard with identity number s_p of $\mathcal{S}$. For root shard, its parent shard is nil. The parent shard $\mathcal{S}_p$ ensures that all its sub-shards have different identities. We denote the i-th node by p_i and denote the i-th block by $B_i^{\mathcal{S}}$ in shard $\mathcal{S}$ respectively. When it is clear from the context, we omit the subscript i or the superscript $\mathcal{S}$. Node p_i has a key pair $\langle pk_i, sk_i \rangle$, where pk_i is the public key and sk_i is the corresponding private key. All messages sent by node must be signed by its private key, other nodes then can verify the message signature with the corresponding public key. Normally, these two notations p_i and pk_i are often used interchangeably to denote a node. Let $\mathtt{H}(\cdot)$ be a cryptographic hash function. Let $\mathtt{MR}(\cdot)$ be a Merkle tree based hash function.

2.1. **Consensus Safety Requirement.** In a PoS-based consensus algorithm, a consensus message msg_i^p for the i-th block B_i from consensus node p consists of B_i and p's signature on B_i, namely $msg_i^p = \langle B_i, \sigma_i^p \rangle$. The block path of node p from block s to block t is $path_{p,s,t} = \bigcup_{i=s}^{t} msg_{p,i}$. We then can define all consensus messages M_p sent by node p as:

$$M_p = \bigcup_{s,t} path_{p,s,t}.$$

Consensus message m' of node p satisfied safety requirement iff *either* of the following restrictions is satisfied:

- The block path selected in m' is prefixed with the candidate block path of a consensus message in M_p,

- Or, the intersection of the candidate block path selected in m' and the candidate block path of any consensus message in M_p is empty.

If a node violates the above safety requirements, it will be identified as a malicious node.

2.2. **Commitment Ordering.** Cross-shard transaction processing, similar as distributed transaction process in databases, in addition to snapshot isolation (SI) [7], should consider how to deal with commit order between conflicted transactions, that is, commitment ordering. Without loss of generality, transactions are processed in the following steps:

- Transaction-Begin;
- Processing operations in Transaction;
- Transaction-Commit.

Transaction-Begin can be merged with the first operation in transaction.

There are two ways to determine the order between conflicted transactions [17]: one is sorting transactions by transaction's Transaction-Begin order, the other is sort transactions by transaction's Transaction-Commit order.

If the commitment order is defined based on Transaction-Begin, it can be implemented with two-phase-lock (2PL) [24]. Transaction processing will lock all resources needed in the step of Transaction-Begin. This method is simple to implement, but resource locking time is longer. Since resources that are dependent by transaction cannot be predicted in advance, and all resources of the target smart contract need to be locked. During the lockout, all other transaction cannot access the locked smart contract. The optimization space of this method is relatively small, and impact on performance is relatively large [2].

If commitment order is defined based on Transaction-Commit, it can be implemented with Multi-Version snapshot and two-phase-commit(2PC) [12]. When processing the Transaction-Begin request, one snapshot of shard KVStore for the transaction is created, the subsequent Transaction-Read will read data from the snapshot, and the Transaction-Write will store updated data to the WriteSet cache of the transaction. At the end of the transaction execution, transaction commit is done with the two-phase-commit (2PC) protocol.

Compared with Transaction-Begin based ordering, ordering based on Transaction-Commit is more complicated, but resource locking time is much shorter and there is a larger space for optimization. For example, during the execution of a transaction, it can be detected in advance whether the current transaction has been conflicted with other transactions, so that conflicted transactions can be aborted in advance. In addition, because the resources are locked in final commit phase, if underlying kvstore supports fine-grained locking, only necessary data will be locked during transaction committing, thereby improving transaction concurrency.

2.3. **SSER.** In a database, the strongest consistency criteria of transaction processing model is strict serialized consistency criteria.

Definition 2.1 (SSER)**.** Strict serializability (SSER) [5, 24, 6] is a transactional model: operations, usually termed "transactions", can involve several primitive operations performed in order. SSER guarantees that operations take place atomically, namely, a transaction's sub-operations do not appear to interleave with sub-operations from other transactions.

Transaction processing in most of current non-sharding blockchains, such as Ethereum and EOS, are based on the synchronization model, which satisfies the following criteria:

- Atomicity. A transaction is either executed or failed. If the transaction fails, any intermediate results in the execution of the smart contract will not have any effect on the execution of other transactions.
- All transactions are executed in order of transactions in the block.
- All transactions are executed synchronously, that is, all smart contract calls are processed sequentially, and one transaction is processed before another transaction is processed.

Obviously, the transaction model of current blockchains satisfies SSER.

2.4. **Scalability Properties.** In [4], following the analysis of the PSI [21] model, four properties essential to scalability of distributed transaction processing are

defined, including Wait Free Queries (WFQ), Genuine Partial Replication (GPR), Minimal Commitment Synchronization (MCS), and Forward Freshness (ForwFresh).

Definition 2.2 (WFQ)**.** WFQ says, if there are only read operations in a transaction, then its execution does not have to wait for any other concurrent transactions to commit.

Definition 2.3 (GDR)**.** If both of the following restrictions are satisfied, we say GDR is satisfied.

- State data is partially replicated in the sharding network, that is, a subset of the data is saved in each node.
- Transaction communicates only with the replicas that store some object accessed in the transaction. Nodes participate in transaction execution when necessary.

Definition 2.4 (MCS)**.** Transaction T_i waits for transaction T_j only if T_i and T_j write-conflict.

Definition 2.5 (ForwFresh)**.** Under the premise of satisfying Snapshot Isolation, transaction T is allowed to read data that other transactions commit after T starts (reading an object version that committed after the start of the transaction).

Although these properties are defined in the distributed database domain, cross-sharding transactions in the blockchain network are semantically similar with cross-shard transactions in distributed databases. We shall analyze the scalability of Ontology Sharding algorithm based on these four properties.

3. System Definitions and Models

In this section, we shall give some definitions. Besides, we shall give system model, network model, security model, economic model and transaction model.

3.1. Formal Definitions.

We now introduce definitions for Ontology Sharding.

Definition 3.1 (Shard)**.** Ontology shard consists of nine components,

$$\begin{aligned}\mathcal{S} \quad := \quad & \langle s, s_p, state, peers, stakes, channels, \\ & contracts, kvstore, transactions\rangle,\end{aligned}$$

where s is the identity number corresponding to $\mathcal{S}$, and assigned by its parent shard $\mathcal{S}_p$ of identity s_p. *state* is the state of $\mathcal{S}$, and should be *init*, *active* or *archived*. *peers* keeps a list of all the nodes participating in $\mathcal{S}$. *stakes* records the total number staking by each node in $\mathcal{S}$. *channels* is for communication with other shards. Different shards communicate through separate channels.

contracts records all smart contracts deployed in $\mathcal{S}$. *kvstore* records the state data of shards, including the status data of all smart contracts in $\mathcal{S}$. *kvstore* provides four data operation methods, *get*/*put*/*snapshot*/*commit*, and supports multi-version control. *get* and *put* are the data read and update interfaces respectively. *snapshot* creates a read-only snapshot for the current *kvstore*. *commit* will update the batch of data to the current *snapshot* based on a snapshot. If the updated data in the batch is in the snapshot version and inconsistent with the latest version in kvstore, the commit will fail. *transactions* records all transactions in $\mathcal{S}$.

Definition 3.2 (Block)**.** The i-th block of shard $\mathcal{S}$ consisted of three parts,

$$B_i^{\mathcal{S}} := \langle \texttt{Hdr}_i^{\mathcal{S}}, \texttt{blkShardMsg}, \texttt{txs}\rangle,$$

where $\texttt{Hdr}_i^{\mathcal{S}}$ is the block header of B_i^s, `blkShardMsg` includes the cross-shard messages sent from other shards to $\mathcal{S}$, and `txs` include transactions requested in shard $\mathcal{S}$.

Definition 3.3 (Block Header)**.** The header of block $B_i^{\mathcal{S}}$, denoted by $\texttt{Hdr}_i^{\mathcal{S}}$, includes current block number i, block hash of the previous block $\texttt{H}(B_{i-1}^{\mathcal{S}})$, block hash of the current block $\texttt{H}(B_i^{\mathcal{S}})$, block number of parent shard j, and the corresponding block hash $\texttt{H}(B_j^{\mathcal{S}_p})$, Merkle root of the share message request queue and response queue, and the metadata information of the block `bM`, such as the VRF value of $B_i^{\mathcal{S}}$ and the configuration of the shard. Namely,

$$\begin{aligned}\texttt{Hdr}_i^{\mathcal{S}} \quad := \quad & \langle i, \texttt{H}(B_{i-1}^{\mathcal{S}}), \texttt{H}(B_i^{\mathcal{S}}), j, \texttt{H}(B_j^{\mathcal{S}_p}), \\ & \texttt{MR}(Req), \{\texttt{MR}(Rsp)\}, \texttt{bM}\rangle,\end{aligned}$$

where $\texttt{MR}(Req)$ stores the Merkle root in the cross-shard message queue sent from $\mathcal{S}$ to other shards. $\{\texttt{MR}(Rsp)\}$ stores the Merkle root of the cross-shard message sent from other shards to $\mathcal{S}$.

For the sake of security, it requires that

$$\texttt{Hdr}_i^{\mathcal{S}}.j \geq \texttt{Hdr}_{i-1}^{\mathcal{S}}.j.$$

Definition 3.4 (Smart Contract)**.** Smart contract $\mathcal{C}_c^{\mathcal{S}}$ deployed in shard $\mathcal{S}$ includes three parts,

$$\mathcal{C}_c^{\mathcal{S}} := \langle \mathcal{S}, c, ops \rangle,$$

where c is the identity number assigned by shard $\mathcal{S}$. It is worth noting that this identity number is unique in shard $\mathcal{S}$, and usually, we let c be the hashing result of metadata of $\mathcal{C}_c^{\mathcal{S}}$.

$ops = [\{get, put, localInvoke, remoteInvoke\}^*]$ is the operation sequence defined in $\mathcal{C}$, where get/put is to read and update data from $kvstore$ of $\mathcal{S}$, $localInvoke$ is to call other smart contracts in $\mathcal{S}$, and $remoteInvoke$ is to call smart contracts in other shards.

Definition 3.5 (Transaction)**.** The Transaction in Ontology shard consists of eight components:

$$\begin{aligned} T \quad := \quad & \langle \mathcal{S}, c, hash, sender, args, \\ & feeAmount, \{E_T^{\mathcal{S}}\} \rangle, \end{aligned}$$

where $\mathcal{S}$ is shard where T is sent to, c is the identity number of the smart contract where T is requesting to. $hash$ is the hash value of T's payload, $sender$ is the account address of T's sender. $sender$ pays the fee for T, $feeAmount$ is the limit for transaction payment. $\{E_T^{\mathcal{S}}\}$ records T's execution state in every involved shard.

Definition 3.6 (Transaction Execution State)**.** Transaction Execution State $E_T^{\mathcal{S}}$ in shard $\mathcal{S}$ includes five parts,

$$\begin{aligned} E_T^{\mathcal{S}} \quad &:= \quad \langle snapshot, readset, writeset, shards, O_T^{\mathcal{S}} \rangle, \\ O_T^{\mathcal{S}} \quad &:= \quad \langle result, fee \rangle, \end{aligned}$$

where $snapshot$ is the $kvstore$ snapshot created when the transaction execution started, $readset$ records all read operations in T's execution, $writeset$ caches all write operations in T's execution, $shards$ keeps ID numbers of all shards which are involved in processing of T.

$O_T^{\mathcal{S}}$ is the transaction T's output from shard $\mathcal{S}$, where fee keeps paid fees to shard $\mathcal{S}$ for T and $result$ is the execution result. The output of transaction T is O_T, which is defined as:

$$O_T \quad := \quad \bigcup_{\mathcal{S}} O_T^{\mathcal{S}}$$

$\mathbb{S}_T$ is all shards which participated in processing of T, and can be defined as:

$$\mathbb{S}_T \quad := \quad \bigcup_{\mathcal{S}} E_T^{\mathcal{S}}.shards$$

Definition 3.7 (Channel)**.** Channel is a bidirectional communication system between two shards in Ontology network. It consist of two ordered message queues. Shard $\mathcal{S}$ keeps a separate channel $\mathcal{CH}_{\mathcal{S}\leftrightarrow\mathcal{S}'}$ for every $\mathcal{S}'$,

$$\mathcal{CH}_{\mathcal{S}\leftrightarrow\mathcal{S}'} := \langle \mathcal{Q}_{\mathcal{S}\to\mathcal{S}'}, \mathcal{Q}_{\mathcal{S}'\to\mathcal{S}} \rangle,$$

where $\mathcal{Q}$ is a one-way message queue based on Merkle tree and $\mathcal{Q}_{\mathcal{S}\to\mathcal{S}'}$ and $\mathcal{Q}_{\mathcal{S}'\to\mathcal{S}}$ respectively store request messages from $\mathcal{S}$ to $\mathcal{S}'$ and response messages from $\mathcal{S}'$ to $\mathcal{S}$.

Definition 3.8 (Message Queue)**.** Message Queue $\mathcal{Q}_\ell^{\mathcal{S}\to\mathcal{S}'}$ of length ℓ is a one-way message queue based on Merkle tree,

$$\mathcal{Q}_\ell^{\mathcal{S}\to\mathcal{S}'} := \{\texttt{Msg}_i^{\mathcal{S}\to\mathcal{S}'}\}.$$

Message queue provides three interfaces: GetRoot, GetMsgProof, VerifyMsg.

Definition 3.9 (Cross-Sharding Message)**.** A cross-sharding message $\texttt{Msg}_i^{\mathcal{S}\to\mathcal{S}'}$ consists of $\texttt{MsgHdr}_i^{\mathcal{S}\to\mathcal{S}'}$ and $\texttt{MsgBody}_i^{\mathcal{S}\to\mathcal{S}'}$, namely

$$\begin{aligned} \texttt{Msg}_i^{\mathcal{S}\to\mathcal{S}'} \quad &:= \quad \langle \texttt{MsgHdr}_i^{\mathcal{S}\to\mathcal{S}'}, \texttt{MsgBody}_i^{\mathcal{S}\to\mathcal{S}'} \rangle, \\ \texttt{MsgHdr}_i^{\mathcal{S}\to\mathcal{S}'} \quad &:= \quad \langle i, \mathcal{S}, \mathcal{S}', n, \texttt{MsgHdr}_{i-1}^{\mathcal{S}\to\mathcal{S}'}.n, \\ & \qquad \texttt{MR}(\mathcal{Q}_{i-1}^{\mathcal{S}\to\mathcal{S}'} \cup \texttt{Msg}_i^{\mathcal{S}\to\mathcal{S}'}), \\ & \qquad \texttt{MR}(\mathcal{Q}_{i-1}^{\mathcal{S}\to\mathcal{S}'}) \rangle, \end{aligned}$$

where $\mathcal{S}$ is source shard where the message is sent out, $\mathcal{S}'$ is the target shard. n is the corresponding block number in $\mathcal{S}$ that generates the message.

$\texttt{MsgBody}_i^{\mathcal{S}\to\mathcal{S}'}$ is either request message or response message to $\mathcal{S}'$. $\texttt{MsgBody}_{\texttt{req}}$ is the request message for $\mathcal{S}'$ generated in the n-th block of shard $\mathcal{S}$, $\texttt{MsgBody}_{\texttt{rsp}}$ is the response message for $\mathcal{S}'$ generated in the block.

$$\begin{aligned} \texttt{MsgBody}_{\texttt{req}} \quad &:= \quad \{\langle tx, args, feeAmount \rangle\}, \\ \texttt{MsgBody}_{\texttt{rsp}} \quad &:= \quad \{\langle tx, O_{tx}^{\mathcal{S}} \rangle\}, \end{aligned}$$

where tx is the transaction where the message is generated for, $args$ is arguments associated the request, $feeAmount$ is the fee limit for the request processing on remote shard, $O_{tx}^{\mathcal{S}}$ is the output of processing tx from $\mathcal{S}$.

3.2. **System Model.** Currently, Ontology main chain uses VBFT consensus, a PoS-based consensus, and each node participates through staking. VBFT consensus adopt a two-tier network architecture. In Ontology Network, assuming that there are $\mathcal{N}$ nodes that are responsible for processing transactions and collaborate to maintain the state of Ontology main chain. A Node in the network should be assigned to one of two roles, namely candidate node or consensus node. The role of one node is adjusted based on its stake in the beginning each consensus epoch. In each epoch, a certain number of nodes are randomly selected by VRF [20, 23, 26], and consensus decision-making is achieved through BFT [8] among the selected consensus nodes. The governance contract is responsible for managing nodes in network.

Ontology Sharding takes a hierarchical solution, which is illustrated in Figure 3. We regard Ontology main chain as the root shard, each shard can be built based on its parent shard, where the parent shard can be the root shard (main chain), or another shard. Each Node maintains a set of shards this node joined. If a node is going to participate in some shard $\mathcal{S}$, it has to first participate in the parent shard of $\mathcal{S}$. In the current design, Ontology shards still use VBFT consensus protocol, and pledged stakes of each shard are managed by the Shard Management Contract (SMC) in its parent shard.

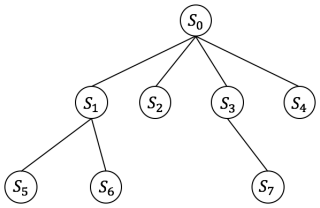

FIGURE 3. Hierarchical Sharding

3.3. **Network Model.** Like many blockchain networks, Ontology main chain is built on a P2P network in a synchronous network model, that is, honest nodes can guarantee the completion of message communication within the known maximum delay.

In Ontology Sharding network, each shard establishes its own sub-network based on the P2P network and can have its own maximum delay. The sub-networks of shards are independent of each other, that is, messages of a shard are only transmitted between nodes of the shard, and node p will not receive shard messages in $\mathcal{S}$ if p is not on $\mathcal{S}$'s peers list.

3.4. **Security Model.** In a PoS-based consensus mechanism, let W_t and W_e be the total nodes staking of the entire network W_e and the total honest nodes staking respectively. Let β be consensus security factor. Consensus safety requires $W_e > W_t \cdot (1 - \beta)$.

The consensus security of Ontology main chain depends basically on the two parameters: W_t and β. Consensus parameter β is set when the blockchain is created. Once β is determined, consensus safety of the blockchain depends on W_t, which is the sum of the stakes of all the nodes in the main chain.

Ontology Sharding network uses the same consensus mechanism as Ontology main chain. Nodes participate in a shard by staking. The security of a shard depends on the amount of stakes pledged for this shard. In Ontology Sharding network, it adds a new parameter γ, which defines the proportion of the number of stakes that nodes can pledge on children shards. The security of Ontology Sharding network requires:

$$\begin{cases} \gamma < & 50\% \\ \beta_{\mathcal{S}_p} \leq & \beta_{\mathcal{S}} \\ W_{\mathcal{S}_p} > & W_{\mathcal{S}} \end{cases}$$

In Ontology network, all the stake contracts are deployed in the main chain (root shard). If node p wants to increase its stakes in some shard $\mathcal{S}$, it must firstly increase its stakes in root shard and then increase its stakes in ancient shards of $\mathcal{S}$ sequentially. Therefore, if $\mathcal{S}_{ancient}$ is the ancient shard of $\mathcal{S}$, the stake in $\mathcal{S}_{ancient}$ is necessarily greater than the stake number in $\mathcal{S}$, that is, the security of $\mathcal{S}_{ancient}$ is necessarily higher than the security of $\mathcal{S}$.

3.5. **Economic Model.** The Ontology network uses a dual-token model with the ONT and ONG tokens, where ONT and ONG are respectively used for governance and economic incentives.

Fees for all transactions in Ontology Sharding network are paid based on ONG. It is worth noting that using ONG as a unified gas token would benefit Ontology Sharding network. On one hand, it encourages

nodes to participate in the transaction processing and state maintenance of shards. On the other hand, it contributes to the security of Ontology shards.

3.6. **Transaction Model.** Ontology transaction execution process also satisfies SSER as other ones. In Ontology Sharding network, cross-shard transactions are processed based on an asynchronous model, but transactions inside one shard are still processed synchronously.

Synchronous cross-shard transaction processing will make a shard pause to wait responses of another shards, and then will cause the entire sharding network to suspend. Finally, it will bring great security risks to all shards. Therefore, the synchronization model is not applicable for transactions across shards. In asynchronous mode, the life cycle of a transaction can span multiple blocks, completing the commit of the transaction in its last block.

This transaction model also greatly simplifies the development of smart contracts. Therefore, for the development of smart contracts, Ontology Sharding network provides transactional guarantees for cross-shard transactions while ensuring:

- Atomicity of transactions;
- Transactions inside a shard are executed in the order of transactions in blocks. For cross-shard transactions, the commit request order is taken as the commit order;
- Transactions inside a shard are executed synchronously. Cross-sharding transactions are executed based on an asynchronous model, and the commit of transactions is synchronous;
- State data of all sync-nodes on shards are consistent;
- Block synchronization in a shard does not need to synchronize any data from other shards;
- The processing results of cross-shard transactions can be verified and keep consistent across shards.

In addition, using the same consistency model for across-shard transactions greatly simplifies the development of smart contracts and is also a fundamental requirement for blockchain network fragmentation.

4. Management of Shards

The management of shards includes creation of shards, how nodes join and exit shards.

4.1. **Shard Creation.** Ontology shard is managed by the parent shard $\mathcal{S}_p$ Shard Management Contract (SMC). The creation request of shard $\mathcal{S}$ is defined as:

$$\begin{aligned} T_{create}.args \quad := \quad & \langle param_{net}, param_{stake}, \\ & param_{consensus}, param_{gov}, param_{fee} \rangle \end{aligned}$$

which respectively corresponds to network rules of the new shard, the consensus algorithm and parameters, the governance rules, the transaction fee configuration. The shard startup and shutdown criterion are also defined in $param_{net}$ and $param_{stake}$. After receiving T_{create} in $\mathcal{S}_p$, the SMC contract in $\mathcal{S}_p$ first verifies the validity of parameters. If all parameters are verified, the SMC contract will assign s to $\mathcal{S}$, set state of $\mathcal{S}$ as *init*, and save configuration information of $\mathcal{S}$ in the SMC contract. The newly created shard is

$$\begin{aligned} \mathcal{S} \quad := \quad & \langle s, S_p.s, state = init, \\ & peers = \emptyset, stakes = \emptyset, channels = \emptyset, \\ & contracts = \emptyset, kvstore = \emptyset, transactions = \emptyset \rangle \end{aligned}$$

4.2. **How Peers Join Shard.** Ontology node p of shard $\mathcal{S}_p$ can request to join shard $\mathcal{S}$ by sending T_{join} to SMC contract of $\mathcal{S}_p$.

$$T_{join}.args := \langle \mathcal{S}.s, pk_p, stake_p \rangle$$

The SMC contract in $\mathcal{S}_p$ will first verify if p meets the requirements specified in T_{create} and then add the $T_{join}.pk_p$ and $T_{join}.stake_p$ to $\mathcal{S}.peers$ and $\mathcal{S}.stakes$. If $\mathcal{S}$ satisfies its startup requirement, SMC contract on $\mathcal{S}_p$ sets $\mathcal{S}.state$ to *active* and creates genesis block $B_0^{\mathcal{S}}$ according to the configuration parameters of $\mathcal{S}$.

Since nodes in $\mathcal{S}$ are also in $\mathcal{S}_p$, the shard state update will be notified automatically in $\mathcal{S}$. With $B_0^{\mathcal{S}}$, shard $\mathcal{S}$ starts, and nodes in $\mathcal{S}$ start processing transactions in $\mathcal{S}$. At this point, shard $\mathcal{S}$ is instantiated successfully.

4.3. **How Peers Exit Shard.** Ontology node p in $\mathcal{S}$ can apply for exiting from $\mathcal{S}$ at any time.

If $\mathcal{S}.state$ is in *init* or *archived* state, the exit request should be sent to SMC contract of parent shard

$\mathcal{S}_p$. The SMC smart contract in $\mathcal{S}_p$ will process the exit request and return the stake which p pledged in the SMC.

If $\mathcal{S}.state$ is in *active* state, the exit request should be sent to the governance contract in $\mathcal{S}$. Usually, after one consensus epoch, node p will exit from the governance contract of $\mathcal{S}$, and then SMC contract of $\mathcal{S}_p$ is notified. Finally, SMC contract of $\mathcal{S}_p$ set node p exited from shard $\mathcal{S}$, and return its stakes.

After some nodes exit shard, if number of nodes in the shard or amount of stakes pledged for the shard do not meet the parameter requirements of T_{create}, the state of shard will be changed from *active* to *archived*. Shards with *archived* state stop processing any transactions, and all state data is read-only. Smart contracts in *archived* shards can be migrated to other *active* shards.

The state transition diagram of shard is shown in Figure-4.

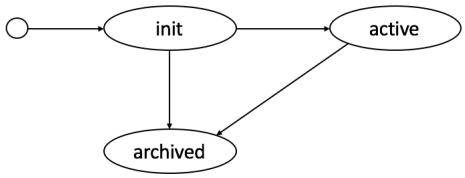

FIGURE 4. States of Ontology shard

5. COMMUNICATION AMONG SHARDS

In Ontology Sharding network, smart contracts are deployed inside of shard, and state data of smart contracts is stored in kvstore of the corresponding shard. If smart contract $\mathcal{C}$ in shard $\mathcal{S}$ does not call smart contracts in other shards or is called by smart contracts of other shards, all its processing will be inside shard $\mathcal{S}$. Ontology shards can scale out by deploying different smart contracts into separated shards, while maintaining smart contract transaction processing performance. However, separated shards also make smart contracts in Ontology network separated from each other. As types of services provided by smart contracts increase, smart contracts in one shard will inevitably require services or data from smart contracts in another shard, which necessitates cross-shard communication.

Shards in Ontology network communicate through channels, a mechanism of serverless asynchronous communication. As defined in section 3.1, shard $\mathtt{S}$ communicate with $\mathtt{S}'$ through $\mathcal{CH}_{\mathcal{S}\leftrightarrow\mathcal{S}'}$, which is composed by two unidirectionally Merklized message queue, $\mathcal{Q}_{\mathcal{S}\rightarrow\mathcal{S}'}$ and $\mathcal{Q}_{\mathcal{S}'\rightarrow\mathcal{S}}$. As shown in Figure-5:

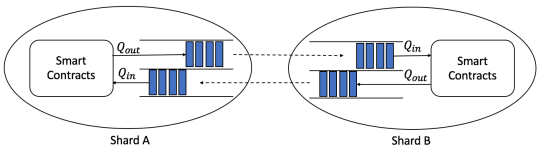

FIGURE 5. Communication Channels of Ontology shards

In Ontology shard definition, shard message channel is part of shard state and is maintained by all nodes in the shard. Smart contracts in shard $\mathcal{S}$ can call the interface of $\mathcal{CH}_{\mathcal{S}\leftrightarrow\mathcal{S}'}$ to send or reply to message requests to shard $\mathcal{S}'$. Similarly, all messages received in $\mathcal{Q}_{\mathcal{S}'\rightarrow\mathcal{S}}$ will be sent to their target smart contracts in shard $\mathcal{S}$ for processing.

When smart contract $\mathcal{C}^{\mathcal{S}}$ processes transaction T_t in block $B_i^{\mathcal{S}}$, $\mathcal{C}^{\mathcal{S}}$ needs to send a cross-shard message to shard $\mathcal{S}'$. The procedure is:

(1) The cross-shard transaction $T_{T_t,req}$ is built by system service of $\mathcal{S}$ based on parameters provided by $\mathcal{C}^{\mathcal{S}}$.

(2) $T_{T_t,req}$ is stored in one internal temporary queue of $\mathcal{S}$.

(3) After all transactions in $B_i^{\mathcal{S}}$ are processed, system service of $\mathcal{S}$ builds a cross-sharding message $\mathtt{Msg}_j^{\mathcal{S}\rightarrow\mathcal{S}'}$ including all cross-shard transactions $T_{\cdot,req}$ in its internal temporary queue, and adds $\mathtt{Msg}_j^{\mathcal{S}\rightarrow\mathcal{S}'}$ to $\mathcal{Q}_{\mathcal{S}\rightarrow\mathcal{S}'}$. $\mathtt{MsgHdr}_j^{\mathcal{S}\rightarrow\mathcal{S}'}$ contains the updated Merkle root of $\mathcal{Q}_{\mathcal{S}\rightarrow\mathcal{S}'}$.

(4) Updated Merkle root of $\mathcal{Q}_{\mathcal{S}\rightarrow\mathcal{S}'}$ is saved in the block header of next block and gets consensused in $\mathcal{S}$.

(5) After consensus completed, nodes in $\mathcal{S}$ confirm the validity of $\mathtt{Msg}_j^{\mathcal{S}\rightarrow\mathcal{S}'}$ in $\mathcal{Q}_{\mathcal{S}\rightarrow\mathcal{S}'}$, construct $T_{fwd_j}^{\mathcal{S}\rightarrow\mathcal{S}'}$ transaction which contains $\mathtt{Msg}_j^{\mathcal{S}\rightarrow\mathcal{S}'}$,

and sent it to $\mathcal{S}'$. To ensure validity of $\texttt{Msg}_j^{\mathcal{S}\to\mathcal{S}'}$, $T_{fwd_j}^{\mathcal{S}\to\mathcal{S}'}$ includes the Merkle proof of $\mathcal{Q}_{\mathcal{S}\to\mathcal{S}'}$.

(6) Like one normal transaction in $\mathcal{S}'$, $T_{fwd_j}^{\mathcal{S}\to\mathcal{S}'}$ is consensused by one block $B_m^{\mathcal{S}'}$.

(7) When processing $T_{fwd_j}^{\mathcal{S}\to\mathcal{S}'}$ in $B_m^{\mathcal{S}'}$, system contract of $\mathcal{S}'$ is invoked to process it.

(8) System contract of $\mathcal{S}'$ verifies $T_{fwd_j}^{\mathcal{S}\to\mathcal{S}'}$, and then puts $\texttt{Msg}_j^{\mathcal{S}\to\mathcal{S}'}$ in $T_{fwd_j}^{\mathcal{S}\to\mathcal{S}'}$ into $\mathcal{Q}_{\mathcal{S}\to\mathcal{S}'}$ of $\mathcal{CH}_{\mathcal{S}'\leftrightarrow\mathcal{S}}$.

(9) While queue $\mathcal{CH}_{\mathcal{S}'\leftrightarrow\mathcal{S}}.\mathcal{Q}_{\mathcal{S}\to\mathcal{S}'}$ is not empty, system contract of $\mathcal{S}'$ keeps getting cross-sharding messages from queue, and processing them in the same order of queue.

5.1. **Message Ordering in Queue.** As shown in above procedure, all cross-sharding transactions $T_{\cdot,req}$ generated in $B_i^{\mathcal{S}}$ are included in one cross-sharding message $\texttt{Msg}_j^{\mathcal{S}\to\mathcal{S}'}$, that is, there is only one cross-sharding message constructed per block. If there is no new cross-sharding transaction from block $B_i^{\mathcal{S}}$, there will no cross-sharding message for it. Similar as block hash pointers in blockchain, $\texttt{MsgHdr}_i^{\mathcal{S}\to\mathcal{S}'}$ contains the corresponding block number of $\texttt{MsgHdr}_{i-1}^{\mathcal{S}\to\mathcal{S}'}$. The following restriction should be satisfied,

$$\texttt{MsgHdr}_i^{\mathcal{S}\to\mathcal{S}'}.n > \texttt{MsgHdr}_{i-1}^{\mathcal{S}\to\mathcal{S}'}.n.$$

So $\mathcal{Q}_{\mathcal{S}\to\mathcal{S}'}$ can also be considered as a chain. When cross-sharding message $\texttt{Msg}_i^{\mathcal{S}\to\mathcal{S}'}$ is received at $\mathcal{S}'$, its block number and Merkle root hash in $\texttt{MsgHdr}_i^{\mathcal{S}\to\mathcal{S}'}$ is firstly verified before crossing sharding transaction $\{T_{\cdot,req}\}$ in $\texttt{MsgBody}_i^{\mathcal{S}\to\mathcal{S}'}$ are processed.

5.2. **Safe Delivery of Message in Queue.** As shown in definition of $\mathcal{Q}_{\mathcal{S}\to\mathcal{S}'}$, message queues are constructed with Merkle tree. After cross sharding message is added to the queue, new queue's Merkle root, $\texttt{MR}(\mathcal{Q}_{i-1}^{\mathcal{S}\to\mathcal{S}'} \cup \texttt{Msg}_i^{\mathcal{S}\to\mathcal{S}'})$, is consensused by subsequent blocks of $\mathcal{S}$. Therefore, based on consensus safety, $\mathcal{Q}_{\mathcal{S}\to\mathcal{S}'}$ is safe in source shard $\mathcal{S}$.

In target shard $\mathcal{S}'$, because Merkle proof of $\texttt{Msg}_i^{\mathcal{S}\to\mathcal{S}'}$ if contained in transaction $T_{fwd_j}^{\mathcal{S}\to\mathcal{S}'}$, nodes in $\mathcal{S}'$ can verify $\texttt{Msg}_i^{\mathcal{S}\to\mathcal{S}'}$ with block headers of $B^{\mathcal{S}}$ and previous cross-sharding messages in $\mathcal{CH}_{\mathcal{S}'\leftrightarrow\mathcal{S}}.\mathcal{Q}_{\mathcal{S}\to\mathcal{S}'}$. Therefore, $\mathcal{Q}_{\mathcal{S}\to\mathcal{S}'}$ is also safe in target shard $\mathcal{S}'$.

6. Cross-Shard Transaction Processing

In Ontology Sharding network, there are two types of cross-shard transaction operations: $NotificationCall$ and $TransactionCall$.

6.1. **Cross-Shard Contract Addressing.** Cross-shard transactions will send requests to smart contracts in other shards, but smart contracts can be migrated among shards in Ontology network. Cross-shard Smart Contract Management Contracts (CCMC) of parent shards provides smart contract addressing services for its children shards. CCMC runs in parent shards, and all smart contracts in its children shards which support cross-chain invocation need to be registered in its CCMC contract. Since all nodes in child shard are also nodes of parent shard, state updates of CCMC in parent shards can immediately be applied to nodes in sharding network. So shards in the network can always get latest address where the target smart contract is.

In addition to providing cross-shard smart contract addressing services, CCMC contracts are also responsible for:

- Detecting cycle dependencies among across-shard smart contracts, preventing deadlocks when processing cross-shard transactions,
- Migrating smart contracts across shards.

Cycle dependency detection is implemented as follows: If smart contract $\mathcal{C}$ in shard $\mathcal{S}$ is going to serve transactions from other shards, it firstly needs to be registered to the CCMC of parent shard $\mathcal{S}'$ and provide $(\mathcal{C}.s, \{\mathcal{C}.dependence_i\})$, where all smart contracts registered in the CCMC that $\mathcal{C}$ depends on should be set in $\{\mathcal{C}.dependence_i\}$. If $\{\mathcal{C}.dependence_i\}$ is not empty, CCMC in $\mathcal{S}'$ will check if there is any circular dependency formed among $\mathcal{C}$ and $\{\mathcal{C}.dependence_i\}$. If there is, registration of $\mathcal{C}$ will failed. Smart contract $\mathcal{C}$ is able to be invoked from other sibling shards after it is registered successfully.

6.2. **Notification Call in Shards.** Invoking smart contract with $NotificationCall$ is similar with in-shard smart contracts invocation. With $NotificationCall$, smart contract $\mathcal{C}$ in shard $\mathcal{S}$ is able to invoke other smart contract $\mathcal{C}'$ in shard $\mathcal{S}'$ asynchronously if $\mathcal{C}'$ has been registered successfully in the CCMC of the parent shard

$\mathcal{S}'$. The implementation of $NotificationCall$ is basically constructing transaction T with arguments provided by $\mathcal{C}$ and forwarding T to $\mathcal{S}'$ with $\mathcal{CH}_{\mathcal{S}\leftrightarrow\mathcal{S}'}$. Cross-shard smart contract invocation with $NotificationCall$ does not provide transactionality and will not receive response from $\mathcal{C}'$, and just forwarding transactions from $\mathcal{C}$ in $\mathcal{S}$ to $\mathcal{C}'$ in $\mathcal{S}'$.

Fee of $NotificationCall$ consists of two parts: the basic fee and the incremental fee.

$$F_{NotificationCall} = F_{source} + F_{target_{basic}} + F_{target_{incr}}$$

The basic fee, including F_{source} and $F_{target_{Basic}}$, is paid in the originating shard $\mathcal{S}$. If more fee is needed to paid to the target shard $\mathcal{S}'$, the incremental portion $F_{target_{incr}}$ will be charged in the target shard.

If the sender of T does not have an account or has insufficient balance in the target shard $\mathcal{S}'$, the execution of T in $\mathcal{S}'$ will fail, while smart contract $\mathcal{C}$ in $\mathcal{S}$ can not know this.

6.3. **Transaction Call in Shards.** Invoking smart contracts in a remote shard with $TransactionCall$ can guarantee the atomicity of multiple smart contract executions across different shards.

Ontology Sharding implements cross-shard TransactionCall based on Multi-Version Commitment Ordering (MVCO) [18, 25]. As mentioned earlier, operations of one transaction in the sharding network can be decomposed of four types: $read$, $write$, $localInvoke$, and $remoteInvoke$, where $read/write/localInvoke$ are operations executed internally in shard, and remoteInvoke will trigger cross-shard $TransactionCall$.

The implementation of MVCO on Ontology Sharding is shown in Algorithm-1, Algorithm-2 and Algorithm-3. When processing transaction T in shard $\mathcal{S}$, one snapshot of shard $kvstore$ will be created for T firstly, $T.snapshot = \mathcal{S}.newSnapshot()$. During the execution of transaction T, $read$ operations in T is to read data from $T.snapshot$ and add read result to $T.readset$; $write$ operations is to save data updates to $T.writeset$, $localInvoke$ operation is to execute the called smart contract based on $T.snapshot$; $remoteInvoke$ operation will make T enter $wait$ state in shard $\mathcal{S}$, build one cross-shard $TransactionCall$ T_{req} for T, and send T_{req} to target shard $\mathcal{S}'$ through $\mathcal{CH}_{\mathcal{S}\leftrightarrow\mathcal{S}'}$. After $\mathcal{S}$ receives the response T_{rsp} of T_{req} from $\mathcal{S}'$, transaction T will continue to execute. After all operations in T are completed, T's transaction result, $\mathcal{O}_T$, will be committed in shard $\mathcal{S}$ and $\mathcal{S}'$ through a 2PC protocol.

The transaction execution state transition in a shard is shown in Figure-6.

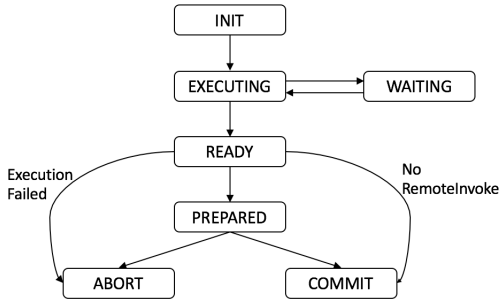

FIGURE 6. States of Cross-Shard Transaction

Algorithm 1 read and write

```
1:  function READ(T, k)
2:      v ← ∅
3:      if ⟨k, v⟩ ∈ T.writeset then
4:          return v
5:      end if
6:      if ⟨k, v, −⟩ ∈ T.readset then
7:          return v
8:      end if
9:      ⟨v, ver⟩ ← getValue(T.snapshot, k)
10:     T.readset ← T.readset ∪ {⟨k, v, ver⟩}
11:     return v
12: end function

13: function WRITE(T, k, v)
14:     T.writeset ← T.writeset \ {⟨k, −⟩} ∪ {⟨k, v⟩}
15: end function
```

After the execution of the transaction, the following 2PC protocol is executed to complete the commitment of transaction T. The algorithm of transaction commitment is shown in Algorithm-4, Algorithm-5, Algorithm-6 and Algorithm-7.

If $T.result = \top$, T will send $Prepare_T$ message to all shards in $\mathbb{S}_T$. $\mathbb{S}_T = \{\mathcal{S}_i\}$. Upon received the

Algorithm 2 LocalInvoke

```
1: function LocalInvoke(T, S, C, args)
2:     if C ∉ S.contracts then
3:         return ∅
4:     end if
5:     if T.snapshot is nil then
6:         T.snapshot ← S.kvstore.snapshot()
7:         T.shards ← {S}
8:     end if
9:     ops ← getOps(T, C)
10:    ops.args ← args
11:    for all op ∈ ops do
12:        if op = READ then
13:            op.value ← Read(T, op.key)
14:        else if op = WRITE then
15:            Write(T, op.key, op.value)
16:        else
17:            if op.shard = S.s then
18:                op.value ← LocalInvoke(T, S,
                               op.contract, op.args)
19:            else
20:                op.value ← RemoteInvoke(T,
                       op.shard, op.contract, op.args)
21:            end if
22:        end if
23:    end for
24:    return op.value
25: end function
```

Algorithm 3 RemoteInvoke

```
1: procedure upon receive RemoteInvoke(T, S',
                C', args)          ▷ RemoteInvoke from S
2:     result ← ∅
3:     T' ← t : {t} ∈ S.transactions ∧ t.hash = T.hash
4:     if T' is nil then
5:         T' ← S'.newtransaction(T)
6:         S.transactions ← S.transactions ∪ {T'}
7:     end if
8:     results ← LocalInvoke(T', S', C', args)
9:     S' send response {result, T'.shards} to S
10: end procedure

11: procedure upon receive RemoteResponse(T, S,
                rsp)                ▷ response from S'
12:    T.shards ← T.shards ∪ rsp.shards
13:    LocalInvoke(T, S, T.contract, rsp.result)
14: end procedure
```

$Prepare_T$ message, shard $\mathcal{S}_i$ firstly verifies the validity of the state of T in $\mathcal{S}_i$, then tries to obtain exclusive locks of all resources in $E_T^{\mathcal{S}_i}.writeset$, and shared locks of all resources in $E_T^{\mathcal{S}_i}.readset$, and then validates $E_T^{\mathcal{S}_i}.reaset$ by checking if there is other transaction which has updated resources in $E_T^{\mathcal{S}_i}.readset$. If verifications passed, shard $\mathcal{S}_i$ responses to shard $\mathcal{S}$ with one $Prepared_{i,commit}$ message. If the verification failed, shard $\mathcal{S}_i$ releases all resources acquired by T in shard $\mathcal{S}_i$, and responses with one $Prepared_{i,abort}$ message.

After shard $\mathcal{S}$ received *Prepared* messages of all the shards in $\mathbb{S}_T$, it determines whether transaction T can be committed. If any shard in $\mathbb{S}_T$ returns $Prepared_{i,abort}$, shard $\mathcal{S}$ will abort transaction T, and send $Abort_T$ message to all shards in $\mathbb{S}_T$. If all shards in $\mathbb{S}_T$ return $Prepared_{i,commit}$, the decision will be to commit. Shard $\mathcal{S}$ will send a $Commit_T$ message to all shards with reliable message queues.

On received the $Commit_T$ message, shard $\mathcal{S}_i$ commits $E_T^{\mathcal{S}_i}.writeset$ to $\mathcal{S}_i.kvstore$. If the $Abort_T$ message is received, shard $\mathcal{S}_i$ will discard $E_T^{\mathcal{S}_i}.writeset$, and record the state of T as "Abort", then release all resources that T held in shard $\mathcal{S}_i$. After all shard in $\mathbb{S}_T$ completed their transaction commitment or aborting, transaction T is completed.

6.4. **Scalability Analysis of Cross-Shard Transaction Processing.** We now analyze the scalability of our cross-shard transaction processing.

6.4.1. *WFQ.* In above Ontology Sharding algorithm, execution of transaction T is based on its isolated snapshot. *Read* operations in T read data from its writeset and snapshot without acquiring any lock, so read operations in Ontology Sharding algorithm are wait free. In Ontology Sharding algorithm, cross-shard transaction T is read-only, if it has not write operations, and all its localInvoke and remoteInvoke also only processed read operations. It's obvious that $T.writeset$ is empty in all shards if T is read-only, so T does not need

Algorithm 4 commit

```
1: function COMMIT(T, S)
2:     outcome ← ⊤
3:     if T.writeset = ∅ ∧ T.shards = {S} then
4:         return ∅
5:     end if
6:     Send Prepare(T, S) to S′   (∀ S′ ∈ T.shards)
7:     for all S′ ∈ T.shards do
8:         Wait to receive Prepared(T, S′, result) from
           S′ or timeout
9:         if result = ⊥ then
10:            outcome ← ⊥
11:            break
12:        end if
13:    end for
14:    Send Decide(T, S′, outcome) to S′
       (∀S′ ∈ T.shards)
15: end function
```

Algorithm 5 prepare

```
1: procedure UPON RECEIVE PREPARE(T, S′)
                                   ▷ prepare from S
2:     T′ ← t : {t} ∈ S′.transactions∧t.hash = T.hash
3:     if T′ is nil then
4:         return Send Prepared(T, S′, ⊥)
5:     end if
6:     if T′.tryExclLock(T′.writeset) is failed then
7:         return Send Prepared(T, S′, ⊥)
8:     end if
9:     if T′.trySharedLock(T′.readset) is failed then
10:        T′.exclUnlock(T′.writeset)
11:        return Send Prepared(T, S′, ⊥)
12:    end if
13:    outcome ← validate(T′, S′, T′.readset)
14:    if outcome = ⊥ then
15:        T′.sharedUnlock(T′.readset)
16:        T′.exclUnlock(T′.writeset)
17:    end if
18:    return Send Prepared(T, S′, outcome)
19: end procedure
```

to commit. Therefore, Ontology Sharding algorithm satisfies WFQ.

Algorithm 6 decide

```
1: procedure UPON RECEIVE DECIDE(T, S′, outcome)
                                   ▷ decide from S
2:     T′ ← t : {t} ∈ S′.transactions∧t.hash = T.hash
3:     if T′ is nil then
4:         return
5:     end if
6:     if outcome = ⊤ then
7:         for all ⟨k, v⟩ ∈ T′.writeset do
8:             ver ← S′.getVer(k) + 1
9:             S′.kvstore.put(⟨k, v, ver⟩)
10:        end for
11:    end if
12:    if T′holdslocks then
13:        T′.sharedUnlock(T′.readset)
14:        T′.exclUnlock(T′.writeset)
15:    end if
16: end procedure
```

Algorithm 7 validate

```
1: function VALIDATE(T, S, readset)
2:     for all ⟨k, −, ver⟩ ∈ readset do
3:         if S.getLatestVersion(k) > ver then
4:             return ⊥
5:         end if
6:     end for
7:     return ⊤
8: end function
```

6.4.2. *GPR.* In Ontology Sharding network, state data is bound with smart contract. If smart contract $\mathcal{C}$ is in shard $\mathcal{S}$, state data of $\mathcal{C}$ is only accessible from shard $\mathcal{S}$. When processing cross-shard transaction T, only smart contracts which are directly or indirectly invoked by T will participate. Therefore, only the shards that have contracts invoked by T participate in its processing. So partial replication can be satisfied. In Ontology Sharding, independent transactions in different shards are able to be executed concurrently. But if they are in the same shard, due to the serializability property of blockchain, they will have to be executed according to their order in the block. Therefore, cross-sharding transaction processing in Ontology Sharding satisfies GPR.

6.4.3. *MCS.* In Ontology Sharding algorithm, two transactions are conflict if their writeset is overlapped. Ontology Sharding transaction only locks the underlying data during the commit phase. The granularity of the commit phase lock is in unit of smart contract, that is, only two transactions need to wait for each other when they conflict on the level of smart contract. Therefore, Ontology Sharding algorithm satisfies MCS.

6.4.4. *ForwFresh.* In the current Ontology Sharding algorithm, snapshot of transaction T in shard $\mathcal{S}$ is created when the first operation of transaction T is performed on shard $\mathcal{S}$. After that, all read operations of T in $\mathcal{S}$ are based on this snapshot. So ForwFresh is not satisfied currently.

As can be seen from the above, Ontology Sharding algorithm satisfies most of the scalable distributed transaction processing properties.

7. Cross-Shard Transaction Fee Processing

The ONT/ONG assets are all managed on Ontology main chain and ONG settlements can only be finalized on the main chain. Fees of all transactions in Ontology Sharding network are settled based on ONG.

As defined in transaction T, $T.feeAmount$ is the max fee amount a user can pay for T. Fee F_T for transaction T is paid by the user who initiated T.

$$\begin{aligned} F_T &:= \sum_{\mathcal{S}} T.E_T^{\mathcal{S}}.fee \\ F_T &\leq T.feeAmount \end{aligned}$$

If T is one transaction inside of shard $\mathcal{S}$, F_T will be paid to the consensus governance contract of $\mathcal{S}$. For security, all transaction fees are accumulated in the consensus governance contract, and are paid to nodes which participate in $\mathcal{S}$ after one consensus epoch. If T is a cross-shard transaction which is initiated in one shard $\mathcal{S}$ and called smart contracts in the other shard $\mathcal{S}'$, F_T will be split into $F_{T,\mathcal{S}}$ and $F_{T,S'}$,

$$F_T = T.E_T^{\mathcal{S}}.fee + T.E_T^{\mathcal{S}'}.fee$$

$T.E_T^{\mathcal{S}}.fee$ will be paid to shard $\mathcal{S}$, $T.E_T^{\mathcal{S}'}.fee$ will be paid to shard $\mathcal{S}'$. The details are explained in the following sections.

7.1. **Fee management in shard.** If a user wants to invoke a smart contract in a shard, he needs to prepay ONG to the shard. The prepaid ONG will be mortgaged in the SMC contract of parent shard, and SMC will notify the shard to update user's prepaid ONG balance on the shard.

In the shard, the shard's system contract bookkeeps the total ONG amount the user paid for his transactions and generates corresponding receipts for each payment. All receipts from one user are managed through one Merkle tree. Based on Merkle roots of user receipt trees, another receipt Merkle tree for the shard is constructed. At the end of every consensus epoch, the shard system contract sends the total sum of all transaction fees in the epoch and its corresponding receipt Merkle root to the parent shard. SMC contracts in the parent shard will validate the Merkle root and pay ONG to nodes in the shard.

7.2. **Redeeming Stakes.** If a user wants to redeem his prepaid ONG on a shard, he/she should send the extraction request to the system contract on the shard. The shard will reduce the balance of the user's prepayment in the shard and notify SMC in the parent shard. SMC contract in parent shard transfers ONG to user account in parent shard.

7.3. **Main chain contract calls in shard.** Smart contracts in the shard can be invoked from the main chain if they have been registered to CCMC. For example, smart contract $\mathcal{C}$ in the main chain S_0 is invoked by transaction T. $T.feeAmount$ is the fee limit user specified for T. First, transaction T was processed in main chain and invoked remote start contract $\mathcal{C}'$ which is in shard $\mathcal{S}'$. At this point, F_1 amount of fee has been consumed on main chain.

When invoking $\mathcal{C}'$, cross-shard transaction T' is constructed. $T'.feeAmount = T.feeAmount - F_1$. T' will be processed in shard $\mathcal{S}'$ by the method described in Section 6.3, and the processing response is returned to main chain via message queue. Assuming F_2 amount of fee is consumed on $\mathcal{S}'$, it should be paid from main chain to $\mathcal{S}'$ for T', and its receipt proof is included in the response message. $F_2 < T'.feeAmount$. Before completed, transaction T may require some further processing in main chain, and consumed F_3 amount of

fee. When transaction T is complete, fee of T is settled on main chain.

$$\begin{aligned} F_T &= F_1 + F_2 + F_3 \\ T.E_T^{\mathcal{S}_0}.fee &= F_1 + F_3 \\ T.E_T^{\mathcal{S}'}.fee &= F_2 \end{aligned}$$

The F_1+F_3 amount of fee is paid to governance contract of main chain, F_2 amount of the fee is paid to the SMC contract of main chain for $\mathcal{S}'$.

Subsequently, when the consensus epoch of shard $\mathcal{S}'$ is complete, SMC pays the received ONG to nodes in shard $\mathcal{S}'$.

7.4. **Shard contract calls other shards.** Similar to the previous, smart contract $\mathcal{C}$ in shard $\mathcal{S}$ is invoked by transaction T with fee limit $T.feeAmount$. If T invokes a contract in the main chain, its transaction fee is settled as follows:

Assuming F_1 amount of fee consumed in $\mathcal{S}$ firstly, F_2 amount of fee consumed by cross-shard transaction on the main chain, then F_3 amount of fee consumed in $\mathcal{S}$ to complete all left processing of T. When transaction T is complete, the fee of T is settled on shard $\mathcal{S}$. On shard $\mathcal{S}$, the F_1+F_3 amount of fee is paid to the shard governance contract and the ONG account balance of user $T.sender$ is reduced by F_2. On main chain, F_2 amount of fee is paid from $\mathcal{S}$ to the governance contract of main chain by main chain SMC contract.

If T invokes a contract in other shard $\mathcal{S}'$, there's some difference in transaction fee settlement. Similar to the above example, transaction fee F_T of T includes F_1, F_2, and F_3. The $F_1 + F_3$ amount of fee is paid to the shard governance contract of $\mathcal{S}$. For F_2, the ONG account balance of user $T.sender$ is reduced by F_2 on shard $\mathcal{S}$, governance contract of $\mathcal{S}'$ requires F_2 amount of ONG from $\mathcal{S}$ on main chain with the transaction receipt as proof. Main chain SMC contract will transfer F_2 amount of ONG which $T.sender$ had pledged for $\mathcal{S}$ to $\mathcal{S}'$.

8. Conclusions

we have introduced the design of Ontology Sharding. The Ontology sharding network adopts hierarchical sharding architecture to achieve large-scale network expansion. Performance of shard is determined only by performance of nodes in the shard.

Key features of Ontology Sharding include reliable communication channels between shards, Multi-Version Commitment Ordering (MVCO) based cross-shard transaction model, and its governance model.

Shards in Ontology network communicate with each other through reliable channels. With a Merkle tree-based message queue, all messages between shards are order guaranteed and verifiable. Ontology sharding supports transactionality for both intra-shard transaction and cross-shard transaction. For intra-shard transaction, the classical synchronous transaction model in single-chain is used. Cross-shard transaction processing adopts the MVCO model, and achieves WFQ, GPR, and MCS.

There are many improvements that can be made to the current design of Ontology Sharding. The minimum granularity of Ontology Sharding is smart contract. When one smart contract is relied on by many other smart contracts, it will cause excessive cross-sharding transactions, which may affect performance of sharding network. We will continue to study how to reduce the granularity of sharding. On the security of shards, Ontology shards are currently based on the ONT token. In the next phase, we will continue to study how to ensure the security of shards if they are based on OEP-4 tokens.

References

[1] The ethereum foundation. sharding roadmap. *https://github.com/ethereum/wiki/wiki/Sharding-roadmap*.

[2] Daniel J. Abadi. Consistency tradeoffs in modern distributed database system design. *Computer*, 2012.

[3] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. 2017.

[4] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. *SRDS*, 2013.

[5] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O' Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *Int. Conf. on Management of Data*, 1995.

[6] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. Concurrency control and recovery in database systems. *Addison-Wesley*, 1987.

[7] Michael J. Cahill, Uwe Rohm, and Alan D. Fekete. Serializable isolation for snapshot databases. *Management of Data*, 2008.
[8] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. *OSDI*, 1999.
[9] John R. Douceur. The sybil attack. 2002.
[10] Larimer Daniel et al. EOS.IO technical white paper v2. 2017.
[11] Wood Gavin. Ethereum: A secure decentralised generalised transaction ledger. 2014.
[12] Rachid Guerraoui and Andre Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 2001.
[13] Poon Joseph and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. 2017.
[14] Poon Joseph and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. 2016.
[15] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. 2016.
[16] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger. 2017.
[17] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. Score: a scalable one-copy serializable partial replication protocol. *Middleware*, 2012.
[18] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luís Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. *International Conference on Distributed Computing Systems*, 2012.
[19] Nakamoto Santosh. Bitcoin: A peer-to-peer electronic cash system. 2008.
[20] Micali Silvio, Michael Rabin, and Salil Vadhan. Verifiable random functions. *Foundations of Computer Science*, 1999.
[21] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. *SOSP*, 2011.
[22] The Zilliqa Team. The zilliqa technical whitepaper. 2017.
[23] Hanke Timo, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. 2018.
[24] Gerhard Weikum and Gottfried Vossen. Transactional information systems. *Elsevier*, 2001.
[25] Raz Yoav. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. *VLDB*, 1992.
[26] Gilad Yossi, RotemHemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. 2017.

Appendix A. Related Work

A.1. **Side Chain.** Segregation Witness (SegWit) is a scaling solution first implemented in the Bitcoin network, mainly used to increase transaction speeds. The SegWit protocol attempts to improve performance of Bitcoin from two aspects, first increasing the maximum block size, increasing the block size limit to 4 MB, and then placing the witness information in each transaction outside the block. This reduces the size of each transaction in the block, allowing a single block to hold records for more than 8,000 transactions.

Lightning Network [14] is a scaling solution for both transaction costs and speeds. It realizes real-time transactions through micro-payment channels, and at the same time guarantees the revocability of disputed transactions by prepaying the fund pool. However, the UTXO asset exchange service can only be provided in the lightning network. In addition, the transactions in the lightning network are 'unconfirmed' bitcoin network transactions, relying on the fund pool in the payment channel to ensure the security of the transaction.

Ethereum's Plasma [13] is a sidechain solution based on Ethereum. Plasma locks Ethereum's assets in the Ethereum smart contract and generates the same amount of assets in the Plasma side chain. Subsequent asset-related transaction processing will be done in Plasma. All transactions in Plasma can be submitted to Ethereum by Merkle proof, so as to ensure the security of the Plasma side chain. Transaction processing is offloaded to the side chain to achieve performance scalability of Ethereum.

A.2. **Sharding.** Sharding is a database partition that divides larger databases into smaller data fragments to make the data management faster and easier. Although sharding has been an significant part of traditional database technology for many years, its implementation in blockchain is still in research [1, 15] because traditional blockchains require all nodes to carry all the data on the blockchain to ensure processing of transactions verified.

OmniLedger and Chainspace are BFT-based sharding proposals. OmniLedger [16] used a bias-resistant representative shards that process transactions, and introduced an efficient cross-shard commit protocol that automatically handles transactions affecting multiple shards. In Chainspace [3], state and execution of transactions are sharded into object, and cross-shard consistency is guaranteed by S-ABC, a distributed commit protocol. Compared with them, Ontology Sharding is a PoS-based sharding design, which is more scalable and secure in an ultra-large-scale network. For cross-shard transactions, besides atomicity, Ontology Sharding also provides fee settlement and contract governance.