# ONTOLOGY SHARDING
## DRAFT v0.1

ONTOLOGY RESEARCH TEAM
RESEARCH@ONT.IO

Abstract. The performance of blockchain has become a big concern. How to make current blockchain networks scalability is one of hot spots in the current blockchain research. We propose Ontology blockchain sharding protocol, which adopts a hierarchical network architecture to support an ultra-large-scale network through multi-layer network sharding. In Ontology sharding system, shards communicate with each other through reliable message queues, and the verifiability of messages between shards is realized by Merkel proof. At the same time, based on MVCO, the transactionality of cross-shard transactions is realized, which provides a strong and consistent transaction model for cross-shard smart contracts.

**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. At the same time, along with an increase of daily transactions and decentralized applications, the scalability issue of current blockchains has emerged.

For instance, before making a decentralized global computer, transaction processing performance of blockchain should have competitive strength. 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, which is far lower than target performance. This problem has seriously hampered possibility that the blockchain will become a mainstream payment solution in the near future.

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

(1) How block is generated. Current blockchains are mainly organized by hashing a single linked list. The current mainstream consensus algorithms only complete the consensus of one block in each round of consensus. Only a few thousand transaction requests can be saved in one block. To realize the matching transaction performance of Visa, blocks need to be consensused every second. The consensus of ten blocks per second is a big challenge for blockchain networks of global scale. Moreover, 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 stores all states (account balance, contract code, storage, etc.) and processes all transactions. This provides a lot of security, but it greatly limits scalability: blockchains cannot handle more transactions than a single node. In addition, as time goes by, the blockchain needs to maintain its own history, such as non-governable, traceable, and other functionalities. The data that each node needs to store will continue to increase, and 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. However, the operation cost of blockchain nodes is not cheap. All transaction requests require fees, and blockchain nodes tend to prioritize transactions that pay higher fees.

> That is, during the peak hours, if transactions need to be verified on time, higher fee have to be paid. This also restricts the use of current blockchain networks.

Blockchain developers have done lots of research and attempts to overcome the above challenges. For example, Bitcoin tries to improve performance by enlarging block size and EOS [10] improves efficiency and consensus nodes 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.

Blockchain state sharding is to shard data of a blockchain, including smart contract state data and blocks. When updating blockchain date, only a subset of shard nodes need to update their data according to the sharding algorithm. State data sharding provides storage scalability for blockchain. If the blockchain sharding design only supports state sharding, it can only provide scalability when dealing with asset transactions based on the randomness assumption of transaction addresses [22]. However, in the case of smart contract scenarios, it has to ensure that the 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 runs an independent consensus algorithm. Transaction sharding provides consensus scalability for blockchains. Transaction sharding is usually is achieved through smart contract sharding, i.e., the smart contract is ran in the shard, and all transactions that call this smart contract will only be submitted to this shard. Only with transaction sharding, the efficiency of transaction processing in the entire blockchain network can be scaled out.

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

- Performance of shard should be linearly scalable,
- Ontology shard network scale should achieve linear expansion,
- In-shard transactions are only processed on the shard nodes,
- Supporting Atomic cross-shard transaction processing,
- Uniformed incentive mechanism in all shards.

The organization of the Ontology sharding network is shown in Figure-1. The root shard is the current Ontology main chain, and child shards are created based on the Ontology root shard. Root shard is responsible for managing all the shards and shards can communicate with each other directly.

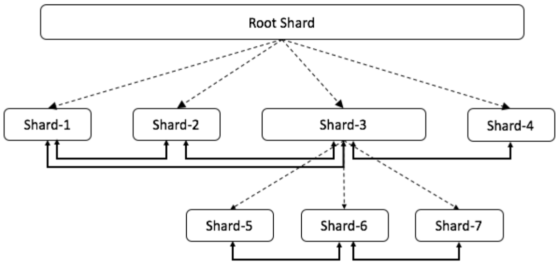

Figure 1. Ontology sharding

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 the Ontology shard network run in one shard (the main chain is the root shard), and the transaction and status data for the smart contract will only be processed and saved in the shard.

The Ontology shard network uses the Verifiable Byzantine Fault Tolerance with Randomness (VBFT) algorithm in the Ontology main chain. VBFT is a consensus algorithm combining Proof-of-Stake (PoS), Verifable Random Function (VRF), and Byzantine Fault

Tolerance (BFT). It guarantees the randomness and fairness of consensus group generation through VRF, while ensuring fast state finality.

The block in a shard adds the link information of the parent shard block to the block of the current Ontology main chain, and the message queue information between the pieces. The block relationship between shard and parent shard is shown in Figure-2.

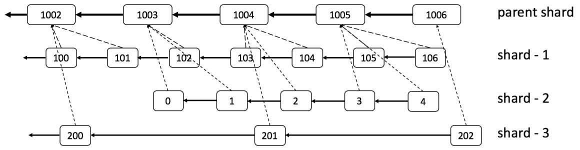

Figure 2. Blockchains of Ontology shards

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 the shard will only get consensus within the shard, that is, transaction processing performance of a shard depends only on the node performance in the shard and the network performance among shard nodes. This ensures linearly scalability of the 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 of the 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 transcation processing of Onotology Sharding and give scalablity analysis of the cross-sharding transcation 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 $(pk_i, sk_i)$, 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 $msg_i^p = (B_i, \sigma_i^p)$. 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 requirement, the node 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 the 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 the transaction.

There are two ways to determine the order between conflicted transactions[17]: 1. Sort transactions by transaction's Transaction-Begin order. 2. 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 the resources needed in the step of Transaction-Begin. This method is simple to implement, but resource locking time is longer. Since resources are dependent by transaction they 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 the 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 the data from the snapshot, and the Transaction-Write will store the update 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 the final commit phase, if the underlying kvstore supports fine-grained locking, only the 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 the 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 the 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,

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

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,

$$B_i^{\mathcal{S}} := (\mathtt{Hdr}_i^{\mathcal{S}}, \mathtt{blkShardMsg}, \mathtt{txs}),$$

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

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

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

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

For the sake of security, it requires that

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

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

$$\mathcal{C}_c^{\mathcal{S}} := (\mathcal{S}, c, \mathit{ops}),$$

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 $\mathcal{C}_c^{\mathcal{S}}$.

$\mathit{ops} = [\{\mathit{get}, \mathit{put}, \mathit{localInvoke}, \mathit{remoteInvoke}\}^*]$ 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 the Ontology shard consists of eight components:

$$\begin{aligned} T \quad &:= \quad (\mathcal{S}, c, hash, sender, arguments, \\ & \qquad feeAmount, \{E_T^{\mathcal{S}}\}), \end{aligned}$$

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,

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

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 number of all shards which is involved in processing of $T$.

$O_T^{\mathcal{S}}$ is the transaction $T$'s output on shard $\mathcal{S}$, where *fee* keeps paid fees to shard $\mathcal{S}$ for $T$ and *result* is the execution result.

**Definition 3.7** (Channel)**.** Channel is a bidirectional communication system between two shards in the 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 $\mathcal{S}'$,

$$\mathcal{CH}_{\mathcal{S}\leftrightarrow\mathcal{S}'} :=< \mathcal{Q}_{\mathcal{S}\to\mathcal{S}'}, \mathcal{Q}_{\mathcal{S}'\to\mathcal{S}} >,$$

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 $\ell$ is a one-way message queue based on Merkle tree,

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

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

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

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

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.

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

$$\begin{aligned} \mathtt{MsgBody}_{\mathtt{req}} &:= \{(tx, args, feeAmount)\}, \\ \mathtt{MsgBody}_{\mathtt{rsp}} &:= \{(tx, O_{tx}^{\mathcal{S}})\}, \end{aligned}$$

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 is implemented based on Proof-of-Stake (PoS), and each node participates through staking. Meanwhile, the consensus of Ontology main chain uses a two-tier network model. Assuming that, in Ontology Network, there are $\mathcal{N}$ nodes that are responsible for processing transactions in the network and collaborate to maintain the state of Ontology main chain. Nodes in network can be candidate nodes or consensus nodes. The role of node in the network is adjusted in each consensus epoch based on stake. In each epoch, a certain number of nodes are randomly selected by VRF (Verifiable Random Function)[20, 23, 26], and consensus is decided by means of BFT (Byzantine Fault Tolerance)[8] among the selected consensus nodes. The Ontology main chain's governance contract is responsible for managing nodes in network.

To achieve continuous scalability, shards in the network should also support scale-out. There are two ways to scale-out a shard: one is to split one shard into two shards, and the other is to build child shards based on current shards. Both methods can achieve performance scalability of sharding, but also bring certain limitations. The first way makes all the shards at the same level, all based on the root shard (main chain), which can guarantee the security of the shards. However, in splitting smart contracts in original shards, they have to be migrated to new shards. The migrated smart contract will only be able to access resources and data 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. This way, new shards can be built based on the original shard, and smart contracts in new shard will be able to directly access resources in the original shard. In addition, although the main chain is constructed in PoS mode, shards can be configured with other consensus algorithms, and hierarchical sharding can be this kind of scenario.

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 the parent shard, where the parent shard can be the root shard, or another shard. Each Node $p_i$ maintains a set of shards this node joined. In Ontology shards, if node $p_i$ 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 uses VBFT's consensus algorithm, and pledged stakes of shard are managed by the Shard Management Contract (SMC) in the parent shard.

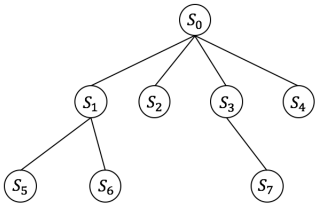

FIGURE 3. Hierarchical Sharding

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

In Ontology sharding's network model, each shard establishes its own sub-network, and uses the network model of the Ontology main chain inside the sub-network. Each shard can have its own maximum delay. The sub-networks of shards are independent of each other, that is, messages of shards are only transmitted between nodes of the shard, and node $p_k$ will not receive shard messages in $\mathcal{S}_i$ if $p_k$ is not in $\mathcal{S}_i$'s peers list.

3.4. **Security Model.** Ontology's consensus algorithm is based on PoS. Defining $W_t$ as total node staking of the entire network, $\beta$ as consensus security factor, $W_e$ as total honest node staking in the network: consensus safety requirements $W_e > W_t \cdot (1 - \beta)$. The consensus security of the Ontology main chain depends mainly on two parameters: $W_t$ and $\beta$, consensus parameter $\beta$ is set when the blockchain is created. After $\beta$ 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.

The Ontology sharding network uses the same consensus algorithm of the Ontology main chain. Shard nodes participate in the shard by staking. The security of a shard depends on the amount of stakes pledged for the shard. Ontology sharding's consensus model adds a new parameter $\gamma$, which defines the proportion of the number of stakes that nodes can pledge on children shards, $\gamma < 50\%$. The security model of the Ontology shard network is defined as:

$$\begin{aligned} \beta_{parent} &\geq \beta_{child} \\ \gamma &< 50\% \\ W_{s'} &> W_s \quad if \quad s' = s.parent \end{aligned}$$

In the Ontology network all the stake contracts are deployed in the main chain (root shard), if node $p$ wants to increase the pledge of funds in the shard $\mathcal{S}$, the stake must be pledged from the root chain, and the root shard is completed by layered asset transfer and the pledge to the current shard. Therefore, if $\mathcal{S}_{ancient}$ is the ancient shard of $\mathcal{S}$, the staked stake in $\mathcal{S}_{ancient}$ is necessarily greater than the staked 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 is based on a dual token model in which ONT is used for governance, ONG is used for economic incentives, and fees for all transactions in the network are paid through ONG. In the Ontology sharding network, this economic model design will continue, and ONG will be used as the transaction fee payment method for transactions in the shard, so that it encourages the nodes to participate in the transaction processing and state maintenance of shards. The uniform of transaction fees also contributes to the security of Ontology shards.

3.6. **Transaction Model.** In the Ontology sharding network, transactions processed within a shard are synchronous. Being the same as the current transaction execution process, it satisfies SSER.

Cross-shard transactions are executed asynchronously. Because synchronous cross-shard transaction processing will make shard pause to wait response for another shard, this can cause the entire sharding network to suspend, and bring great security risks to all shards. Therefore, the synchronization model is not applicable for transactions across shards.

In the Ontology sharding network, cross-shard transactions are processed with an asynchronous model, but transactions inside one shard are still processed synchronously. 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, the Ontology shard network provides transactional guarantees for cross-sliced transactions while ensuring:

- Atomicity of transactions;
- Shard internal transactions are executed in the order of transactions in the block. For cross-shard transactions, the commit request order is taken as the commit order;
- Shard internal transactions are executed synchronously. For cross-sharding transactions, the execution is asynchronous, and the commit of the transaction is synchronous;
- State data of all sync-nodes on the shard are consistent;
- Block synchronization does not need to synchronize any data from other shards;
- The processing results of cross-shard transactions can be verified and consistent across shards.

In addition, the same consistency model across fragmented transactions will greatly simplify the development of smart contracts and is 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 network is managed by the parent shard $\mathcal{S}_p$ Shard Management Contract (SMC). The creation request of shard $\mathcal{S}$ is defined as:

$$\begin{aligned} T_{create} \quad := \quad & (param_{net}, param_{stake}, \\ & param_{consensus}, param_{gov}, param_{fee}) \end{aligned}$$

which respectively corresponds to the network rules of the new shard, the consensus algorithm and parameters, the governance rules, the transaction fee configuration, and the like. 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 the parameters. If all parameters are verified, the SMC contract will assign $s$ to $\mathcal{S}$, set state of $\mathcal{S}$ as $init$, and save the configuration information of $\mathcal{S}$ in the SMC contract. The newly created shard is

$$\begin{aligned} \mathcal{S} \quad := \quad & (s, S_p.s, state = init, \\ & peerList = \emptyset, stakes = \emptyset, channel = \emptyset, \\ & contracts = \emptyset, kvstore = \emptyset, transactions = \emptyset) \end{aligned}$$

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 $\mathcal{S}_p$.

$$T_{join} := (\mathcal{S}.s, pk_p, stake_p)$$

The SMC contract in $\mathcal{S}_p$ will first verify that $p$ meets the requirements specified in $T_{create}$ and then add the $T_{join}.pk_p$ and $T_{join}.stake_p$ to the $\mathcal{S}.peerList$ and $\mathcal{S}.stakes$. If $\mathcal{S}$ satisfies the startup requirement, the SMC contract sets $\mathcal{S}.state$ to $active$ and creates a genesis block of $\mathcal{S}$ 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 the genesis block, nodes in $\mathcal{S}$ start processing transactions in $\mathcal{S}$. At this point, shard $\mathcal{S}$ is instantiated.

4.3. **How Peers Exit Shard.** Ontology node $p$ in the $\mathcal{S}$ can apply for exiting from shard at any time. If $\mathcal{S}.state$ is in *init* or *archived* state, the exit request should be sent to the 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 *active*, 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 the SMC contract of $\mathcal{S}_p$ is notified. Finally, the SMC contract of $\mathcal{S}_p$ set node $p$ exited from shard $\mathcal{S}$, and return its stakes.

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

The state transition diagram of shard $\mathcal{S}$ is shown in Figure-4.

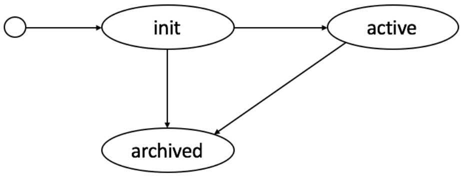

Figure 4. States of Ontology shard

## 5. Communication among Shards

In Ontology shard, smart contracts are deployed inside of the shard, and state data of smart contracts is stored in kvstore of the corresponding shard. If smart contract $C$ 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 sharding 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 the 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 the provision of cross-shard communication.

Shards in Ontology network communicate through channels, a mechanism of serverless asynchronous communication. The channel between Ontology shards consists of $Ch$ in the shards at both ends of the queue. As shown in Figure-5, each $Ch$ is composed of two unidirectionally Merkle Queue: $Q_{in}$ and $Q_{out}$.

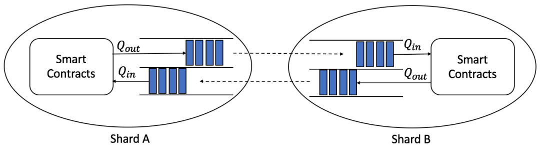

Figure 5. Communication Channels of Ontology shards

In the Ontology shard definition, shard message channel is part of the shard state and is maintained by all nodes in the shard. Smart contracts in the shard can call the interface of $Ch$ to send or reply to message requests to other shards. Similarly, all messages received in $Q_{in}$ will be sent to their target smart contracts in shards for processing.

When smart contract $\mathcal{C}$ in $\mathcal{S}$ processes transaction $T$ in block $B$, $\mathcal{C}$ needs to send a cross-shard message to shard $\mathcal{S}'$. The sending process is:

(1) The cross-shard transaction $T_{req}$ is built by system service of $\mathcal{S}$ based on parameters provided by $\mathcal{C}$.
(2) $T_{req}$ is stored in one internal temporary queue of $\mathcal{S}$.
(3) After all transactions in $B$ are processed, system service of $\mathcal{S}$ builds a ShardMsg $m_{req}$ including all cross-shard transactions $T_{req}$ in the internal temporary queue, and adds $m_{req}$ to $\mathcal{S}.Q_{out}$, which will update the Merkle root of $\mathcal{S}.Q_{out}$.
(4) The new Merkle root is saved in the block header of next block and gets consensused in $\mathcal{S}$.

(5) After consensus is complete, nodes in $\mathcal{S}$ confirm the validity of the $m_{req}$ in $\mathcal{S}.Q_{out}$, and constructs a $T_{fwd}$ transaction which contains $m_{req}$ to $\mathcal{S}'$. To ensure the validity of $m_{req}$, $T_{fwd}$ includes the Merkle proof of $\mathcal{S}.Q_{out}$.

(6) Like one normal transaction in $\mathcal{S}'$, $T_{fwd}$ is included into the block $B'$ of $\mathcal{S}'$ and completes the $B'$ consensus.

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

(8) System contract of $\mathcal{S}'$ verifies the validity of $T_{fwd}$, and then puts $m_{req}$ in $T_{fwd}$ into the $\mathcal{S}'.Q_{in}$.

(9) While $\mathcal{S}'.Q_{in}$ is not empty, system contract of $\mathcal{S}'$ keeps getting cross-shard msg from queue, and process all $T_{req}$ by the same order in message.

5.1. **Message Ordering in Queue.** As defined by $Msg$, in $\mathcal{S}$, $T_{req}$ generated in one block will be included into one $Msg$, that is, only one ShardMsg can be constructed per block. If there is no new cross-shard request from the block, there will no $Msg$ for the block. Similar as blocks in the blockchain, $Msg$ contains the block number of its previous $Msg$,

$$Msg_i.Hdr.blockNum > Msg_{i-1}.Hdr.blockNum,$$

so the cross-shard message Merkle queue can be considered as a chain. When ShardMsg is received at $\mathcal{S}'$, the blockNum and lastMsgBlockNum in ShardMsg is firstly verifed with Merkle proof, then $T_{req}$ in ShardMsg is processed in the order of ShardMsg in the message queue.

5.2. **Safe Delivery of Message in Queue.** As defined by ShardMsg queue, message queues are constructed based on Merkle Tree. After ShardMsg is added to the queue, the new queue's MerkleRoot is confirmed by the consensus of subsequent blocks. Therefore, the message queue is safe in source shard $\mathcal{S}$. In target shard $\mathcal{S}'$, transaction $T_{fwd}$ contains the Merkle proof of ShardMsg, so nodes in $\mathcal{S}'$ can verify $T_{fwd}$ with the block header of the corresponding block in $\mathcal{S}$ and previous messages in $\mathcal{S}'.Q_{in}$. Therefore, the message queue is safe in target shard $\mathcal{S}'$.

## 6. Cross-Shard Transcation Processing

In the 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 between shards in the Ontology network. Cross-shard Smart Contract Management Contracts (CCMC) of parent shards provide smart contract addressing services for the Ontology network. CCMC runs in parent shards, and all smart contracts in the shard network that support cross-chain invocation need to be registered in its parent shard's 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 all nodes in the shard network, and shards in the network can always get updated addresses of where the target smart contract is located.

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 $\mathcal{S}$ will serve transactions from other shards, it first 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 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 whether $\mathcal{C}$ and $\{\mathcal{C}.dependence_i\}$ form a circular dependency. If there is, registration of $\mathcal{C}$ will failed.

6.2. **Notification Call in Shards.** Invoking smart contract with NotificationCall is similar with in-shard smart contracts invocation. With NotificationCall, smart contract $\mathcal{C}_1$ in shard $\mathcal{S}_1$ is able to invoke other smart contract $\mathcal{C}_2$ in shard $\mathcal{S}_2$ asynchronously if $\mathcal{C}_2$ 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 $\mathcal{C}_1$ and forwarding $T$ to $\mathcal{S}_2$ with a reliable message queue between $\mathcal{S}_1$ and $\mathcal{S}_2$. Cross-shard smart contract invocation with NotificationCall does not provide transactionality and response from $\mathcal{C}_2$, and just forwarding transactions from $\mathcal{C}_1$ in $\mathcal{S}_1$ to $\mathcal{C}_2$ in $\mathcal{S}_2$.

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

$$\begin{aligned} Fee_{NotificationCall} &= Fee_{source} + Fee_{target_{basic}} \\ &\quad + Fee_{target_{incr}} \end{aligned}$$

The basic fee, including $Fee_{source}$ and $Fee_{target_{Basic}}$, is paid in the originating shard $\mathcal{S}_1$. If more fee is need to paid to the target shard $\mathcal{S}_2$, the incremental portion $Fee_{target_{incr}}$ will be charged in the target shard.

If the sender of $T$ does not have an account or insufficient balance in the target shard $\mathcal{S}_2$, NotificationCall will fail.

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

Ontology sharding implements cross-shard transaction 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 completed internally in shard, and remoteInvoke will trigger cross-shard TransactionalCall. The implementation of MVCO on the Ontology sharding is: when processing transaction $T$ in shard $\mathcal{S}$, one snapshot of shard state data 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 operations 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 the reliable message queue between $\mathcal{S}$ and $\mathcal{S}'$. 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, transaction result of $T$, including transaction result of $T_{req}$ in $\mathcal{S}'$, will be committed in shard $\mathcal{S}$ and $\mathcal{S}'$ through a 2PC protocol. If a transaction result commitment on any shard fails, the transaction $T$ will be aborted on all the shards to ensure the atomicity of cross-shard transactions.

The state diagram of transaction execution in the Ontology shard network is shown in Figure-6.

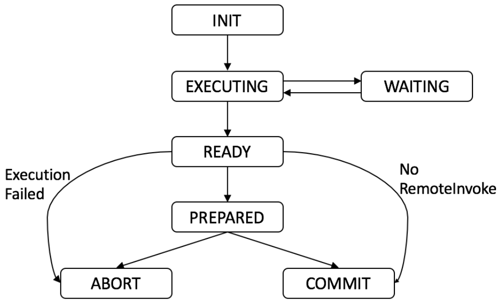

Figure 6. States of Cross-Shard Transaction

The algorithm of transaction processing is as follows:

**Algorithm 1** read and write

```
function READ($T, k$)
    $v \leftarrow \emptyset$
    if $\langle k, v\rangle \in T.writeset$ then
        return $v$
    end if
    if $\langle k, v\rangle \in T.readset$ then
        return $v$
    end if
    $\langle v, ver\rangle \leftarrow getKV(T.snapshot, k)$
    $T.readset \leftarrow T.readset \cup \{\langle k, v, ver\rangle\}$
    return $v$
end function

function WRITE($T, k, v$)
    $T.writeset \leftarrow T.writeset \setminus \{\langle k, -\rangle\} \cup \{\langle k, v\rangle\}$
end function
```

After the execution of the transaction, the following 2PC protocol will be executed to complete the commitment of a transaction. All shards $\{\mathcal{S}\}$ participating in transaction $T$ are recorded in $T.shards$. If $T.result = true$, $T$ will first send $Prepare$ message to all shards

**Algorithm 2** LocalInvoke

```
function LocalInvoke($T, \mathcal{S}, \mathcal{C}, args$)
    if $\mathcal{C} \notin \mathcal{S}.contracts$ then
        return $\emptyset$
    end if
    if $T.snapshot$ is $nil$ then
        $T.snapshot \leftarrow \mathcal{S}.kvstore.snapshot()$
        $T.shards \leftarrow \{\mathcal{S}\}$
    end if
    $ops \leftarrow \mathcal{C}.ops$
    $ops.args \leftarrow args$
    for all $op \in ops$ do
        if $op = READ$ then
            $op.value \leftarrow Read(T, op.key)$
        else if $op = WRITE$ then
            $Write(T, op.key, op.value)$
        else
            if $op.shard = \mathcal{S}.s$ then
                $op.value \leftarrow LocalInvoke(T, \mathcal{S}, op.contract, op.args)$
            else
                $op.value \leftarrow RemoteInvoke(T, op.shard, op.contract, op.args)$
            end if
        end if
    end for
    return $op.value$
end function
```

**Algorithm 3** RemoteInvoke

```
procedure upon receive RemoteInvoke($T, \mathcal{S}', \mathcal{C}', args$)    ▷ RemoteInvoke from $\mathcal{S}$
    $result \leftarrow \emptyset$
    $T' \leftarrow t : \{t\} \in \mathcal{S}.transactions \wedge t.hash = T.hash$
    if $T'$ is $nil$ then
        $T' \leftarrow \mathcal{S}'.newtransaction(T)$
        $\mathcal{S}.transactions \leftarrow \mathcal{S}.transactions \cup \{T'\}$
    end if
    $results \leftarrow LocalInvoke(T', \mathcal{S}', \mathcal{C}', args)$
    $\mathcal{S}'$ send response $\{result, T'.shards\}$ to $\mathcal{S}$
end procedure

procedure upon receive RemoteResponse($T, \mathcal{S}, rsp$)    ▷ response from $\mathcal{S}'$
    $T.shards \leftarrow T.shards \cup rsp.shards$
    $LocalInvoke(T, \mathcal{S}, T.contract, rsp.result)$
end procedure
```

in $T.shards$. Upon receiving the $Prepare$ message, shard $\mathcal{S}'$ first verifies the validity of the state of $T'$ in the shard, then tries to obtain exclusive lock of all resources in $T'.writeset$, and shares lock of all resources in $T'.readset$, and then validate $T'.reaset$ by checking if there are other transactions to update resources in $T'.readset$. If verifications pass, shard $\mathcal{S}'$ will return one $Prepared_{commit}$ message to shard $\mathcal{S}$. If the verification fails, shard $\mathcal{S}'$ will release all resources acquired by $T'$ on shard $\mathcal{S}'$, then return one $Prepared_{abort}$ message to shard $\mathcal{S}$. After shard $\mathcal{S}$ received $Prepared$ messages of all the shards in $\mathcal{S}$, it determines whether transaction $T$ can be committed. If any shard in $\mathcal{S}$ returns $Prepared_{abort}$, shard $\mathcal{S}$ will abort transaction $T$, and send $Abort$ message to all shards in $T.shards$. If all shards in $\mathcal{S}$ return $Prepared_{commit}$, the decision will be to commit, shard $\mathcal{S}$ will send a $Commit$ message to all shards with reliable message queues. On receiving the $Commit$ message, shard $\mathcal{S}'$ commits $T'.writeset$ to $\mathcal{S}'.kvstore$. If the $Abort$ message is received, shard $\mathcal{S}'$ will discard $T'.writeset$, and record the state of $T'$ as ”Abort”, then release all resources that $T'$ held in shard $\mathcal{S}'$. After all shard in $T.shards$ have completed their transaction commit or abort, transaction $T$ is completed. The specific algorithm is as follows:

### 6.4. Scalablity Analysis of Cross-Shard Transaction Processing.

6.4.1. *WFQ.* In the above Ontology sharding algorithm, execution of transaction $T$ is based on its isolated snapshot, and read operations in $T$ read from its writeset and snapshot, so read operations in Ontology sharding algorithm are wait frees. In the Ontology sharding algorithm, two transactions conflict each other if there are write conflicts. If cross-shard transaction $T$ is read-only, that is, its localInvoke and remoteInvoke are also read-only, $T.writeset$ is empty in all shards, $T$ does not need to commit. Therefore, the Ontology sharding algorithm satisfies WFQ.

**Algorithm 4** commit

```
function COMMIT($T, \mathcal{S}$)
    $outcome \leftarrow \top$
    if $T.writeset = \emptyset \wedge T.shards = \{\mathcal{S}\}$ then
        return $\emptyset$
    end if
    Send $Prepare(T, \mathcal{S})$ to $\mathcal{S}'$   ($\forall\ \mathcal{S}' \in T.shards$)
    for all $\mathcal{S}' \in T.shards$ do
        Wait to receive $Prepared(T, \mathcal{S}', result)$ from $\mathcal{S}'$ or timeout
        if $result = \bot$ then
            $outcome \leftarrow \bot$
            break
        end if
    end for
    Send $Decide(T, \mathcal{S}', outcome)$ to $\mathcal{S}'$ ($\forall \mathcal{S}' \in T.shards$)
end function
```

**Algorithm 5** prepare

```
procedure UPON RECEIVE PREPARE($T, \mathcal{S}'$)                ▷ prepare from $\mathcal{S}$
    $T' \leftarrow t : \{t\} \in \mathcal{S}'.transactions \wedge t.hash = T.hash$
    if $T'$ is $nil$ then
        return Send $Prepared(T, \mathcal{S}', \bot)$
    end if
    if $T'.tryExclLock(T'.writeset)$ is $failed$ then
        return Send $Prepared(T, \mathcal{S}', \bot)$
    end if
    if $T'.trySharedLock(T'.readset)$ is $failed$ then
        $T'.exclUnlock(T'.writeset)$
        return Send $Prepared(T, \mathcal{S}', \bot)$
    end if
    $outcome \leftarrow validate(T', \mathcal{S}', T'.readset)$
    if $outcome = \bot$ then
        $T'.sharedUnlock(T'.readset)$
        $T'.exclUnlock(T'.writeset)$
    end if
    return Send $Prepared(T, \mathcal{S}', outcome)$
end procedure
```

**Algorithm 6** decide

```
procedure UPON RECEIVE DECIDE($T, \mathcal{S}', outcome$)                ▷ decide from $\mathcal{S}$
    $T' \leftarrow t : \{t\} \in \mathcal{S}'.transactions \wedge t.hash = T.hash$
    if $T'$ is $nil$ then
        return
    end if
    if $outcome = \top$ then
        for all $\langle k, v \rangle \in T'.writeset$ do
            $v.ver \leftarrow max(v.ver, \mathcal{S}'.getVer(k) + 1)$
            $\mathcal{S}'.kvstore.put(\langle k, v \rangle)$
        end for
    end if
    if $T' holdslocks$ then
        $T'.sharedUnlock(T'.readset)$
        $T'.exclUnlock(T'.writeset)$
    end if
end procedure
```

**Algorithm 7** validate

```
function VALIDATE($T, \mathcal{S}, readset$)
    for all $\langle k, v \rangle \in readset$ do
        if $\mathcal{S}.getLatestVersion(k) > v.ver$ then
            return $\bot$
        end if
    end for
    return $\top$
end function
```

6.4.2. *GPR.* In the Ontology sharding network, state data is bound with a smart contract. If smart contract $\mathcal{C}$ is in shard $\mathcal{S}$, state data of $\mathcal{C}$ is only accessible in 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 store data accessed in $T$ participate in the processing, and 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 serialibility 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 the Ontology sharding algorithm, the 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, the Ontology sharding algorithm satisfies MCS.

6.4.4. *ForwFresh.* In the current Ontology fragmentation algorithm, a 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 $\mathcal{S}$ will be based on this snapshot. So ForwFresh is not satisfied currently.

As can be seen from the above, the 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 the Ontology main chain and ONG settlements can only be finalized on the main chain. Fees of all transactions in the Ontology sharding network are settled through 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 $F_{T,\mathcal{S}'}$,

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

$T.E_T^{\mathcal{S}}.fee$ 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 section.

7.1. **Fee management in shard.** If a user wants to invoke a smart contract in a shard, he need to prepay ONG to the shard. The prepaid ONG will be mortgaged at the SMC contract of the parent shard, and SMC will notify the shard to update the 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. **User extracts pledged ONG.** If the user wants to extract prepaid ONG on shard, user sends 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 conseumed 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.

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

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 the shard is determined only by performance of nodes in the shard.

Key features in 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 are smart contracts. When one smart contract is relied on by many other smart contracts, it will cause excessive cross-sharding transactions, which may greatly affect the performance of the sharding network. Next 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, and security of shards are guaranteed by proof of stake. 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 the 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, faster, and easier to manage parts called data fragments. Although sharding has been an important part of traditional database technology for many years, its implementation in blockchain is still in research[1, 15]. This is because traditional blockchains require all nodes to carry all the data on the blockchain to ensure processing of transactions are 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 on an ultra-large-scale network. For cross-shard transactions, besides atomicity, Ontology sharding also provides fee settlement and contract governance.