Lecture 10: Sharding: Scaling Storage, Computation and Communication

Principles of Blockchains, University of Illinois,
Professor: Pramod Viswanath
Scribe: Ranvir Rana

March 2, 2021

Abstract

Bitcoin security relies on full replication: all nodes store the full ledger, validate the full ledger and communicate the full ledger to each other. The security of Bitcoin is tied crucially to the replication and we see that there is an inherent trade-off between security and efficiency. Sharding aims to ameliorate, or entirely eliminate, this tradeoff: Can a blockchain be both secure and efficient in storage, communication, and computation requirements? Such are the goals of this lecture.

1 Introduction

Bitcoin uses *full replication*: everyone stores every block, everyone validates/computes on every block and everyone communicates every block. Thus the number of participants does not change the load per node (in terms of storage, compute, communication); i.e., the total load of the network grows linearly with the number of participants in the blockchain.

More nodes joining the network implies greater participation which comes with the need for greater throughput; however, if every node replicates the complete ledger, the waste in resources (and hence the cost of a transaction) increases with the increasing number of nodes. This additional resource burden is a negative network effect; not desirable in distributed systems relying on a big network for security.

Many distributed systems scale their "load" with an increasing number of nodes. Such scaling with the increase in the number of nodes is termed as *horizontal scaling*. Consider Bit-torrent protocol for file-sharing: For example, if more nodes join the network, the number of nodes storing a file of interest increases. Thus a single file request can be routed to a single node with lower probability, hence decreasing the load per file-request. The property of the distributed system where the performance (in Bit-torrent's case, the number of files that can be distributed) scales *linearly with the number of participating nodes* is called *horizontal scaling*. This is in contrast to *vertical scaling*, where the performance is improved to the best possible for a fixed number of nodes (and associated peer to peer network); this was the focus of the previous two lectures, improving throughput and latency for a fixed underlying physical network.

Blockchains have a different requirement with simple file-sharing systems in addition to being Byzantine fault tolerant: it needs to maintain consensus on ledger order, i.e., every node should agree on the location of any particular transaction in the ledger. Intuitively, if you need to decrease the "load" per node per transaction, you need nodes to not see all transactions. But how can a node agree on the order of all transactions if it hasn't seen all transactions? The fact that some protocols can achieve horizontal scaling seems surprising! Horizontal scaling is achieved using an idea called *sharding*; the idea, with origins in distributed database theory, is to split the database into subsets

with each subset stored by a different subset of nodes. Thus full replication is averted. To that end, consider the following first order approach to sharding in blockchains.

First order approach to Sharding. Suppose the ledger can be split into K exclusive and exhaustive non-intersecting subsets; this is natural when the ledger is composed of "independent" applications which do not interact with each other and no transaction crosses the application boundaries. Further in the lecture, we will remove this assumption by introducing *cross-shard* transactions. The simplest idea is to let each of K applications be managed as independent blockchains. The ledger is split into K non-interacting sub-ledgers called shards. However the total number of participating nodes, N is shared across the blockchains; in sharding, each node participates in maintaining only one of the shards. In this first order approach, each shard runs its own consensus protocol independent of other shards, e.g., we maintain K blockchains (managed by the longest chain protocol) in parallel, with each blockchain maintained by N/K nodes. Since a node maintains only one shard, it maintains only a $1/K$ fraction of the ledger (also denoted as the "state" of the blockchain); thus the maintenance (storage, computation/validation, and communication) costs are a fraction $1/K$ of the overall ledger. In principle, K can increase linearly with N, while ensuring each shard is maintained by a constant number of nodes; thus optimal horizontal scaling is achieved. This increase in efficiency comes with *reduced security*: the overall security is limited by the security of any one of the shards and an adversary can tamper with the state by attacking just one shard. Attacking a single shard is easier than if a single blockchain was maintaining all the applications; this is because the attacker needs to only compete with N/K nodes instead of N nodes (in the PoW longest chain protocol, the adversary congregates all its mining effort on one of the shards while honest mining is split among the shards). In summary, the first order approach provides an order K horizontal scaling, but by decreasing security by a factor of K. How to provision horizontal scaling with no security repercussions is the topic of this lecture.

2 Multiconsensus architecture

Consider the following extension of the first order approach: this time, the allocation of which node maintains which shard is uniformly random. In the first version, the random allocation is managed by a *cryptographically secure oracle* which all nodes have access to; further any node can verify if a fellow participant is managing the shard that it rightfully should (according to the oracle). The key impact of this modification is that adversaries can no longer congregate in a single shard (with the aim of attacking it); the intuition is that if majority exists in the overall set of nodes, then the random node to shard allocation engine transfers this property to each shard thus maintaining the original security level.

The random reallocation is enabled by a node to shard allocation engine (N2S). The cryptographically secure random allocations of N2S can be maintained by a separate service or as part of the blockchain. RandHound is a state of the art distributed randomness generator, external to the blockchain. An intra-blockchain N2S assigns nodes to shards using a (shared) distributed randomness, generated using a consensus engine *shared across all shards* and commonly referred to as a "beacon" consensus engine. The beacon only contains state commitments and N2S allocation metadata. Since N2S allocation metadata mostly involves validator ids and state-commitments are mostly cryptographic hashes posted at large intervals of time (a hash per 10,000 shard transactions for example), the load on a beacon chain is minimal compared to transactions payload on a shard. This makes the "beacon" consensus engine lightweight and thus we can ask all nodes to maintain it.

While this solution seems a straightforward way to achieve horizontal scaling, the corresponding security and level of scaling are quite limited:

- First, the N2S allocation inherently requires each node to have an *identity*. So this sharding procedure cannot be permissionless, a departure from Bitcoin's design.

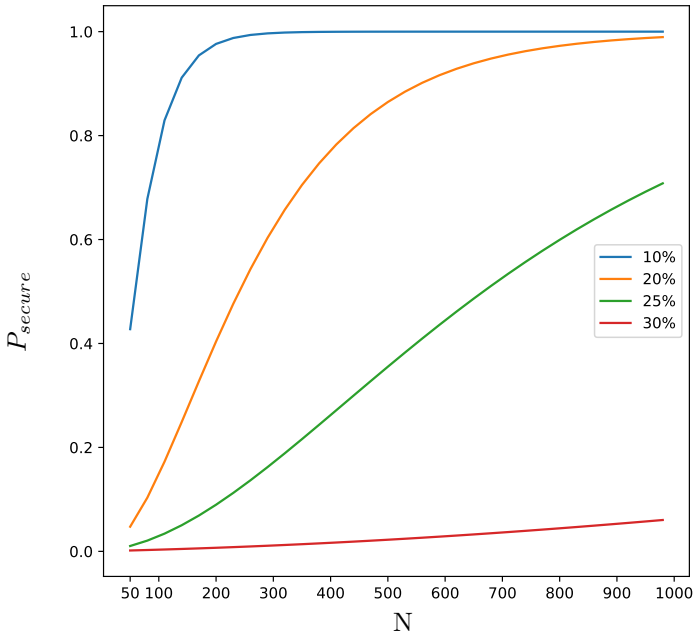


Figure 1: Probability that every shard is secure (i.e., honest supermajority in each of the $K = 10$ shards) as a function of the number of users N under different adversarial models (10%, 20%, 25% and 30% net adversarial fraction)

- Second, for the random allocation to preserve the fraction of honest hashing power in *each* shard (so global majority of honest hash power translates to majority honest hash power in each shard and hence provide security of consensus in each shard), the number of nodes per shard has to be substantial. This limits the number of shards K as a function of the number of nodes N, see Figure 1.

- Third, the architecture is not secure against an *adaptive* adversary, which can corrupt miners *after* they have been allocated to a shard; an adversary can readily implement this attack in practice by allowing miners to advertise their shard allocations on the public Internet, allowing for collusion (i.e., corruptions) inside one of the shards. This is especially problematic as only a small fraction of nodes need to collude to attack a shard (a majority of the $\frac{N}{K}$ nodes in each shard). A standard method to avoid such an attack is to regularly *reallocate* the N2S allocations; the idea is that by the time allocated nodes can coordinate and collude, a reallocation is conducted. Security is provided by rotation among the shard allocations only if such rotations are fast enough; the faster the rotation, the more the overhead of the N2S procedure (the overhead scales linearly with the number of the nodes N), diminishing the horizontal scaling effect.

Each of these three limitations is resolved in a full sharding solution, described next.

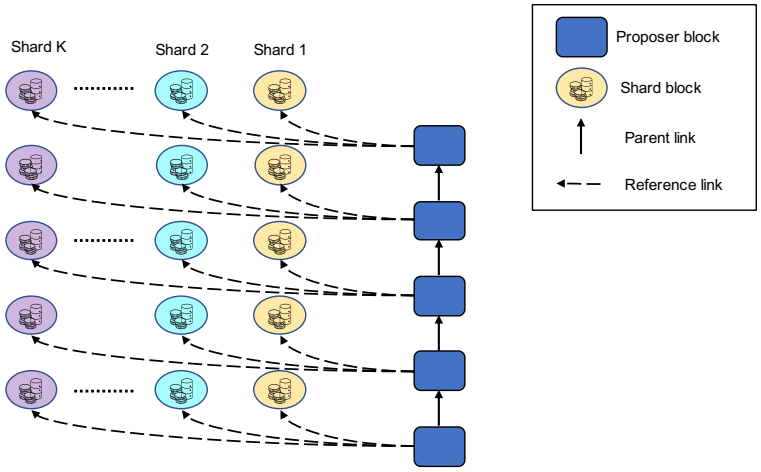


Figure 2: Data structures in the Uniconsensus architecture.

3 Uniconsensus architecture

Rather than relying on a small subset of nodes to maintain a shard, the uniconsensus architecture relies on the consensus amongst all nodes to maintain each shard; this significantly improves multi-consensus architecture's security properties. This is done by having every node participate in a single main consensus engine, as the word "uniconsensus" connotes. While every node participates in the main consensus engine, each node picks a shard of its choice to mine shard blocks (self-allocation). The single consensus engine only maintains a *log of the hash of shard blocks* and hence is scalable. Its light size allows every node to maintain the consensus engine. But how to couple the shard blocks with the main consensus engine, and how to ensure this coupling is adversary-resistant? This is achieved using the same idea we have seen in the past three lectures, scaling fairness (Lecture 7), scaling throughput (Lecture 8) and scaling latency (Lecture 9): many-for-one mining of a superblock and cryptographic sortition into constituent sub-blocks.

The data structure of a uniconsensus architecture consists of two types of block structures: a *proposer* blocktree consisting of *proposer* blocks responsible for the single consensus engine and K shard ledgers consisting of shard blocks. The proposer blocks are organized into a blocktree by a PoW based Nakamoto longest chain consensus protocol. Proposer blocks also contain hash-pointers to shard blocks which are used to order shard blocks in each shard using the longest chain of the Proposer blocktree. Shard blocks are identified with their respective shards via their $ShardID$. The shard blocks contain shard transactions and as with transaction blocks in Prism (Lectures 8 and 9), do not contain any blocktree pointers. This lack of pointers is because the ordering of shard blocks is inferred solely from their order in the main consensus engine (the longest chain of the proposer blocktree) – exactly as in Prism. The sole job of shard block mining is adversary resistance and plays no role in shard block ordering. This data structure is depicted in Figure 2, where the shard ledger is constructed by ordering shard blocks according to the order of their corresponding hash pointers in the proposer chain.

Security relies on the fact that every miner mines proposer blocks and not solely shard blocks; We ensure this via *many-for-one mining* and *sortition*. Proposer and shard blocks are mined *simultaneously* using a 2-for-1 PoW sortition algorithm. A mining node creates a superblock comprising a potential shard block (for a shard of the node's choice) and a potential proposer block and mines a nonce on a header comprising of hash commitments of both the blocks. If the hash of this superblock falls between 0 and τ_1 (proposer block mining difficulty), then it is treated as a proposer block, else if it falls between τ_1 and τ_2 (a shard specific variable) it is treated as a shard block for

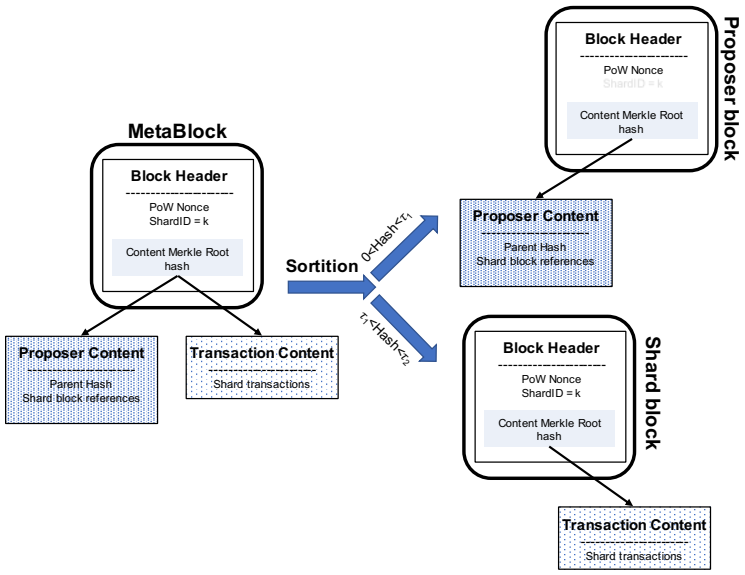


Figure 3: Uniconsensus architecture: 2-for-one PoW sortition.

the identified shard. This ensures that adversaries cannot focus their mining resources selectively on mining either proposer or shard blocks. Moreover, the potential shard block's shard-id is fixed during the mining process, ensuring that a single mining attempt only creates a shard block for one shard (of the miner's choice). The shard block mining difficulty is periodically adjusted for each shard to ensure a constant shard block generation rate for each shard. This 2-for-one PoW sortition is shown in Figure 3.

It is straightforward to see that the uniconsensus architecture scales the resource usage. Any node only maintains the proposer blocktree and a shard ledger of its choice; the node will not maintain the shard ledgers of the rest $K-1$ shards. Maintaining only a fraction $\frac{1}{K}$ of the overall ledger improves efficiency and enables scaling with the number of nodes since most of the storage, computation, and communication resources of the blockchain are used up in maintaining transactions.

An honest consensus majority is no longer needed in each shard to preserve safety; hence uniconsensus architecture is safe against a fully adaptive adversary. Consider a shard with less than a majority of honest nodes; the shard's adversarial nodes cannot change the log of shard blocks without violating the consensus engine's safety. This is prohibitively difficult since everyone maintains the consensus engine.

An important distinction with multiconsensus is that each node is free to join a shard of its choice; an N2S allocation is not required since an honest majority within a shard is no longer required. This architecture guarantees the order of transactions in the shards. Shard nodes perform shard block execution and sanitization of invalid transactions once the order is finalized. We summarize how the uniconsensus architecture overcomes the limitations of multiconsensus architecture:

1. **Identity-free sharding**. No N2S allocation is needed, hence no need to have an on-chain validator/miner identity.

2. **Small number of nodes per shard**. There is no need for honest majority within a shard, hence the constraints established in Figure 1 do not apply.

3. **Adaptive adversary resistance**. Destroying honest majority in a shard is no longer an attack; this achieves resistance to safety violations by corrupting a shard. Moreover the mapping

of miner-identity to shard-id does not exist on-chain, weakening the targeting capabilities of adaptive adversaries.

Uniconsensus architecture allows nodes to self-allocate. While this adds to the freedom of nodes to choose a shard of their choice and removes the node's shard allocation from public knowledge, it allows a liveness attack. An adversary can concentrate its mining power on one shard and drown out honest shard blocks. The fraction of honest blocks to the adversarial blocks is significantly reduced due to a large fraction of adversarial nodes in the shard. This adversarial concentration can throttle the throughput of honest transactions in the shard under attack, leading to a loss in liveness. A dynamic self-allocation (DSA) algorithm like Free2Shard can be used to prevent any such liveness attacks. DSA lets honest users allocate themselves to shards in response to adversarial action without the use of any central authority. An intuitive way as to how DSA works is by allowing honest nodes to choose to relocate themselves to shards that have low throughput (due to liveness attack); such relocation is further incentivized by high transaction fees due to the low throughput.

4 Bootstrap and State-commitment

In both architectures, honest nodes need to relocate themselves to new shards; an important question is how to *efficiently* relocate to the new shard. This problem is similar to bootstrapping in a blockchain. One option is to process the whole ledger; however, doing so would nullify any scaling benefits from sharding since a node will now have to process the ledger of all shards eventually. A more efficient way to deal with this issue is to download the latest state of that shard. However, the bootstrapping node cannot trust a shard-node to provide the correct state. This highlights the need for *trusted* state-commitments. State-commitments are *accumulators* of a ledger's state, which are agreed upon to be correct by everyone in the network. The bootstrapping node can download the state and verify its correctness using the state-commitments. We discuss the details on the accumulator used for state commitments and how they are generated next.

4.1 Accumulator for state-commitment

In the simplest sense, a ledger's state consists of $(account, value)$ tuples. We need to design an accumulator for these tuples. Since the state-commitments need agreement over the network, there needs to be a one-to-one mapping (within cryptographic bounds) from state to state-commitment.

The most common accumulator we have used throughout the course is the root of a Merkle tree. Unfortunately a Merkle tree's root cannot be used for state-commitments: while each node has the same state, the nodes can order the tuples differently while creating a Merkle tree, leading to different roots breaking the one-to-one mapping. The key is to use an *ordered* Merkle tree with order defined by account number; this way, the one-to-one mapping is preserved. However there are performance limitations to using an ordered Merkle tree for state: it is very costly to add or remove an account from an ordered Merkle tree. Adding an account will lead to insertion in the middle, a very expensive operation if the Merkle tree is large. Hence, ordered-Merkle trees are not used for state-commitments.

The dynamic nature of the ledger state with insertions and deletions of accounts warrants the use of a *sparse* Merkle tree. A sparse Merkle tree is a Merkle tree with a fixed but very large number of leaves; there exists a distinct leaf in the tree for every possible output from a cryptographic hash function (every possible account number). It can be simulated efficiently because the tree is sparse (i.e., most leaves are empty). Moreover, insertion and deletion of accounts is computationally simple since a distinct leaf already exists. The sparse Merkle tree is constructed with the value of a leaf being the balance in that account (leaf id = account id) if it exists or null if it does not exist. This sparse Merkle tree for only 2 accounts in the state is illustrated in Figure 4. In practice, a compressed

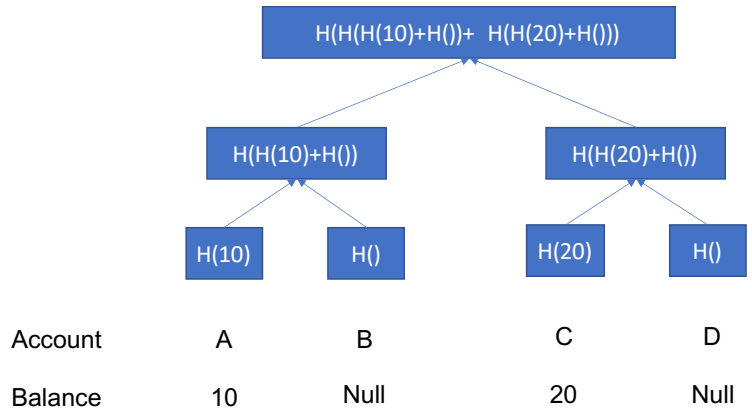


Figure 4: State-commitment using a sparse Merkle tree.

form of sparse Merkle trees called Merkle Patricia Trie (MPT) is used; MPT has similar insertion and deletion complexity (O(1)) as a sparse Merkle tree.

A state commitment consists of the root of a Merkle Patricia Trie of a shard's execution state. It is generated at regular intervals (termed epochs) on the ledger. The root is posted on the beacon chain (multiconsensus architecture) or the main consensus engine (uniconsensus architecture).

4.2 Generation and agreement on state commitments

The entire process of state commitments discussed so far assumes that state commitments are honest. In practice, a byzantine-fault-tolerant way of posting state-commitments needs to be established for both architectures. This is the focus of the following discussion.

4.2.1 Multiconsensus

State-commitment generation is particularly straightforward for the multiconsensus architecture due to the strong underlying assumptions that the nodes have identity and that each shard has an honest majority. Under this assumption, a state-root is generated by any node and signed by a majority of the shard's nodes making it a state-commitment. The state-commitment can be assumed to be correct since each shard is assumed to have an honest super-majority. Better yet, a state-commitment can just be treated as a regular transaction on the shard with deterministic validation rules within the ledger (execution state at that point in the ledger must match the state-commitment). State-commitments are periodically posted on the beacon chain to make it easy for incoming nodes from different shards to obtain the latest state commitment. Posting state-commitments on beacon chain is fine since state-commitments are very small in size (size of a cryptographic hash output).

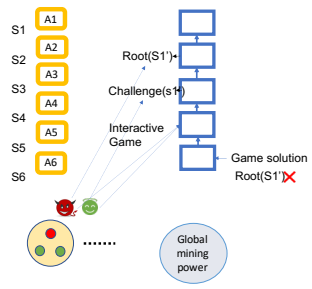
4.2.2 Uniconsensus

State-commitment generation for the uniconsensus architecture is a bit more nuanced, stemming from the fact that we no longer suppose identity of the nodes or that each shard has an honest majority. Moreover, we also assume that not all transactions included in the ledger are valid; this is because validation is decoupled from ordering, i.e., transaction validation is conducted by the *sanitization* process after the order of transactions has been agreed upon. One cannot query the whole network to verify the state-commitment since it will defeat the purpose of sharding (all nodes will have to process all shards). Thus protocols that don't rely on a majority of nodes being honest to run a verifiable computation are needed. The solution below incorporates two new ideas:

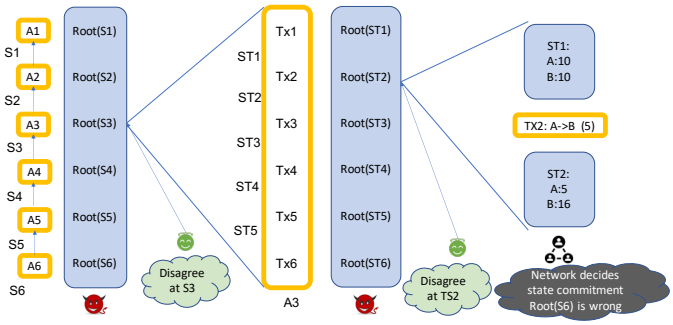
(1)*interactivity* among the nodes generating the state-commitment and (2) relying on honest nodes to detect *fraud* in the broadcasts, instead of the standard approach of proving the veracity of a broadcast.

Fraud-proof based state-commitments follow a "assume correct if not proven wrong within a time-frame" approach. The state-commitments are generated by specific parties in a shard called *commitment leaders* and their broadcasts scrutinized for fraud by honest shard nodes. If a fraud is found, then the proof of fraud is posted by one of the honest nodes (called challenger) on the main consensus engine: proposer chain. If no fraud proof is posted on the proposer chain within a time-frame, the state-commitment is assumed to be honest. This process is depicted in Figure 5a.

Since we aim to keep the proposer chain light (recall that every node maintains the proposer chain), the fraud proof has to be light. Consecutive state-commitments summarize thousands of transactions and we do not want the consensus engine to process all those transactions to verify a fraud-proof. Ideally we want the incorrect computation of state to be proven using just one transaction (or better yet, just one *bytecode*). The pinning down of the fraud proof of incorrect computation to one transaction is achieved by a logarithmic search across all transactions, described next.



(a) Commitment-leader and challenger



(b) Interactive fraud proof

Figure 5: State-commitment in uniconsensus architecture

The protocol for pinning down fraud-proof to one transaction is described briefly below: A *logarithmic fraud search* mechanism consists of two parties: A commitment-leader and a challenger, each belonging to the same shard. Suppose the commitment-leader is adversarial, and posts a state-commitment of the shard on the consensus engine. On seeing an invalid state-commitment, a challenger posts a challenge stating that the computation is incorrect. The following sequence of interactions summarizes the protocol once a fraud is detected and challenged.

- The commitment-leader posts intermediate states for the challenged state.
- The challenger responds with a number indicating the first intermediate state when the challenger's view differs from the leader.
- The game continues to the next round with the leader posting intermediate states for the smaller challenged state and the challenger responding according to the previous step.

The interactions end when the conflict is resolved down to one transaction, i.e., the search for the single transaction is complete. The identified transaction is posted on the consensus engine to decide if a fraud has occurred or not. An example of this transaction search is illustrated in Figure 5b. Once the fraud has been detected and the incriminating transaction found, this specific transaction is excised from the ledger and consensus is regained. In certain designs, the commitment-leader is penalized for this fraud: this penalty can be outside the blockchain in a PoW setting. We will

shortly see the permissioning mechanisms of Proof of Stake (PoS) where the penalization can be handled *within* the blockchain (e.g., via "slashing a collateral transaction required to be posted by the commitment-leaders"). The fraud-proof mechanism together with the ability to penalize adversarial behavior incentivizes commitment-leaders to behave according to the protocol.

5 Cross-shard transactions

We have assumed so far that the state can be split into K subsets that do not interact with one another. While this setup appropriately models a large class of applications, it is interesting to resolve sharding to the fullest extent, by supporting cross-shard interactions. This is the focus of this section. For ease of understanding we will use the following example throughout this section: We want a cross-shard transaction to send funds (x_A, x_B) from input shards A, B to (y_C, y_D) in output shards C, D. A fundamental question in enabling cross-shard transactions is the following: How do validators in shards C and D ensure that funds (x_A, x_B) exist in shards A and B?

Scalability requires that nodes maintaining a shard do not need to know the state of other shards. Thus nodes in output shards cannot directly verify whether the funds are available in input shards. However, an important observation is that each shard as a whole is safe and live in multiconsensus, and there are protocols that ensure safety and liveness of state commitments in uniconsensus (as discussed above). Thus, each shard (or their state commitments) can be thought of as *crash tolerant* and the following commit/abort protocols can be used for successful communication between shards.

A required property for cross-shard transactions is *atomicity*: if a transaction is committed (aborted) in one shard, then it should be committed (aborted) in all participating shards. In the context of our example, if funds x_B do not exist in shard B, the transaction should not be executed in shard A, C and D respectively. Atomicity is simple to state, yet subtle to ensure. Consider the following one-stage cross-shard transaction protocol that appears tailor-made for ensuring atomicity:

- Input shards (A and B in our example) update their state by locking funds x_A and x_B respectively for this cross-shard transaction tx. Their latest state commitments $S1_A$ and $S1_B$ can be used to prove that funds x_A and x_B are locked and cannot be used in shards A and B anymore.

- Proof of lock of funds x_A and x_B for transaction tx is posted on shards C and D respectively. This proof is a simple merkle-proof from state commitments $S1_A$ and $S1_B$ available to both the output shards. On validating a correct merkle proof, shards C and D release funds y_C and y_D respectively.

Unfortunately, this single stage protocol violates atomicity in the following scenario: suppose funds x_B are not available in shard B. When this transaction is initiated, shard A will lock x_A and this will be reflected in state commitment $S1_A$. However, shard B will not lock x_B since those funds do not exist. Since a proof of lock from both input shards is not available, output shards C and D will not release funds y_C and y_D. The fund x_A is locked forever, thus tx was committed in shard A and aborted in the rest of the shards; breaking atomicity.

To ensure atomicity, there must be a way to unlock the funds if one of the input shards fails to lock. To this end, the protocol above is generalized to multiphase commits: In particular, two-phase commitment protocols work as follows:

- Phase 1:
  - A transaction manager TM is assigned to read state commitments across shards; usually a TM is the spending party/ies.
  - If the input funds x_A, x_B are available in shard A and B, respectively, both shards update their state to lock the funds if available.

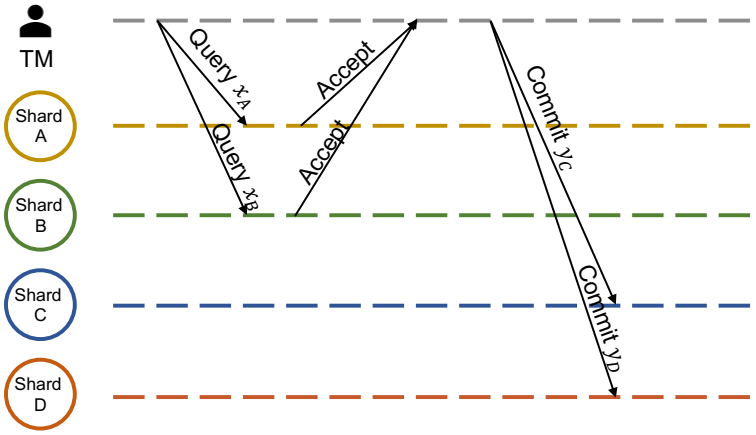


Figure 6: Cross-shard two-phase commitment.

  - If an input fund in shard A is not available, the shard updates its state to mark fund x_A unavailable by adding a *receipt of unavailability* to its state.

- Phase 2:
  - If the TM sees that funds are locked in all input shards as per state commitments $S1_A$ and $S1_B$, it sends a proof of lock to output shards to generate (y_C, y_D).
  - If the TM sees that funds are unavailable in one of the input shards as per state commitments $S1_A$ and $S1_B$, it sends a rollback transaction to all input shards which consists of proof of *receipt of unavailability* in one of the input shards. The rollback transaction reclaims locked funds in the input shards.

This two-phase commitment protocol ensures that all the input funds of the transaction are either spent or unspent; this in turn guarantees atomicity. There will never arise a case where some input funds are spent, and some are not spent. Consider the case discussed above; funds x_B are not available in shard B. When this transaction is initiated shard A will lock x_A and it will be shown in state commitment $S1_A$, however, shard B will not lock x_B but post *receipt of unavailability* in the state commitment $S1_B$. The TM can now send a rollback transaction to shard A to unlock x_A. Hence the transaction would have been aborted in all the participating shards.

References

Scaling blockchains horizontally has been an active area of research for quite some time. Most of the existing sharding solutions have utilized multiconsensus architectures: A good starting point to understanding a concrete sharding protocol that utilizes multiconsensus architectures is Elastico, rapidchain and Omniledger. Practical manifestations of multiconsensus include Ethereum 2.0 and Polkadot. Please refer to Randhound for detailed information on randomness generation used for N2S allocation in many of these protocols.

Various new horizontal scaling proposals utilize uniconsensus including Free2Shard, Aspen, Nightshade, and Lazyledger. The concrete uniconsensus protocol described in these lecture notes is derived from Free2Shard; it also contains DSA algorithms that can be used to solve the liveness issue.

Many partial scaling protocols have been developed that scale one or more of the three resources: Zilliqa scales computation by partitioning validation and Polyshard scales storage by using coded storage and computation.

Detailed information on state roots and Merle Patricia Tries are in the Ethereum Yellow paper. Free2Shard explains the interactive fraud-proof game in brief; a comprehensive description is in Truebit and Arbitrum. The Atomix and Omniledger protocols provide concrete examples of cross-shard transactions that use two-phase commits.