# Two-Phase Commit Using Blockchain

Benjamin Jacob Marks
*Stanford University*
benmarks@stanford.edu

Heron Yang
*Stanford University*
heronyang@stanford.edu

Yuewei Na
*Stanford University*
yueweina@stanford.edu

***Abstract*—We created a decentralized system that uses a public blockchain as a two-phase commit (2PC) coordinator, so users can securely commit an atomic transaction across any databases that support our interface. Users treat our system as one large database with ACID, despite being composed of many individual databases that are unaware of each other's existence. By using a blockchain as a coordinator, it is resilient to failures such as network outages, power outages, and disk failures that cause traditional 2PC implementations to block. Our initial integration is with the Ethereum blockchain and LMDB databases; however, our system provides a generic interface to integrate with other blockchains and databases.**

***Index Terms*—atomic commit, blockchain, smart contract, distributed system, database transaction**

## I. Introduction

Two-Phase Commit (2PC) is a widely-adopted protocol that allows distributed systems to reach the consensus of distributed transactions. It ensures that the transaction can be either committed or aborted at all distributed machines. However, blocking is proven [3] to be to be inevitable even in synchronous systems where bounds on delays can be reliably estimated.

Since the invention of Bitcoin, the underlying technology, the blockchain, not only gained attraction from cryptocurrency systems like Ethereum but also academic research about computer systems. At the core of blockchain, it is a protocol of reaching consensus by using Proof-of-Work or Proof-of-Stake. Previous work [1] has shown that synchronous blockchain can only increase the reliability of the original 2PC protocol but also make it non-blocking when various kinds of failures happen.

In this paper, we further extended the previous work with an optimized system architecture and properly designed interfaces. These contributions can make 2PC with blockchain work in production and at scale.

### A. Key Client Features

*1) Namespaced Distributed Transaction:* We shard the data space with namespaces. Each cohort is only responsible for one namespace. A client transaction is composed of a series of ¡namespace, operation¿ pairs. An operation is either Put(key, value) or Get(key). The Coordinator will forward the operation to the cohort(s) that's responsible for that namespace. Our system guarantees that the transaction can be committed or aborted with consensus from multiple distributed cohorts.

*2) Simple Async Interface:* To a client, they only need to first send CommitAtomicTransaction-Request to the coordinator and then send request GetTransactionResult to fetch the result of their transaction.

### B. System Properties

Compared to the traditional version of 2PC protocol, our approach increases extensibility, scalability and reliability.

*1) Extensibility:* We can plug any blockchain, regardless of being public permissionless or private permissioned into our blockchain interface - 'StartVoting', 'Vote', 'GetVotingDecision'. We can also plug any database system that has ACID support into our transaction delegation interface. The transaction delegation interface implements most common transaction operations - 'Begin', 'Commit', 'Abort', 'Get', 'Put'.

*2) Scalability:* Our Coordinator is designed to be light-weight to handle high throughput. This is achieved by delegating 2 parts of work in traditional 2PC [2]. First, the coordination and voting work is done in the blockchain. Second, it divides the

client transaction into shards and forwards to the corresponding cohort that has the data.

Adding a cohort to the system is as simple as making the coordinator aware of the address of the newly added cohort. The system is horizontally scalable.

*3) Reliability:* With the introduction of namespace, it can improve cohort reliability by adding a cohort whose namespace is the same with one or more of the existing cohorts. This essentially adds another replication in the state replication machine.

On the other hand, leveraging permissionless blockchain increases coordinator reliability. When a coordinator crashes, the commitment states are all kept in the blockchain.

### C. Design Overview

3.1 Architecture and Components We define a transaction as a list of operations starting with BEGIN() followed by multiple GET() or PUT() and ending with COMMIT(). Client creates a transaction and requests Coordinator to commit. Coordinator distributes the transaction to corresponding Cohorts to prepare then offload the remaining 2PC responsibility to the blockchain. Each cohort prepares the transaction locally, votes to Blockchain if ready, checks on Blockchain to see if a transaction should be committed, then finally commits the transaction to its database.

The architecture is similar to the traditional 2PC architecture where we have one coordinator and multiple cohorts - except each cohort talks to the blockchain directly to vote and to look up the voting decision as the coordinator offload the 2PC voting responsibility to the blockchain.

We describe details of each component and each in the remaining section 4.1, the complete flow of handling a successful transaction in Section 4.2, and how we handle failures in Section 4.3.

Fig. Client starts a transaction request with Coordinator. Coordinator distributes the transaction to Cohorts and offload the 2PC responsibility to blockchain. Cohort interacts with the blockchain directly to vote and to look up a transaction status.

*1) Blockchain:* The blockchain component performs a 2PC protocol so it's responsible for tracking voting states of each transaction and making decisions of committing. The implementation uses Ethereum smart contract to store the followings states of each transaction (indexed by the transaction ID) internally: $transaction_state$ Voting states of each cohort of a transaction. $transaction_config$ Transaction configure of a transaction including the number of cohorts and the timeout time.

And, the smart contract exposes the following APIs externally: $StartVoting(transaction_id, cohorts, vote_timeout_time)$ Called by the coordinator to start voting on a transaction. 'cohorts' is the number of cohorts that will vote on this transaction later. $vote_timeout_time$ is the absolute time when the 2PC stops taking. $Vote(transaction_id, cohort_id, ballot)$ Called by the cohort to vote COMMIT or ABORT on a transaction before timeout. $GetVotingDecision(transaction_id)$ Called by any component to check the voting state on a transaction.

StartVoting() initializes the states for a transaction and Vote() updates the voting status of a transaction. GetVotingDecision() makes the decision by following the below logic: If any ABORT vote is received, return ABORT decision. If all COMMIT votes are received, return COMMIT decision. If not (1) and not (2), returns ABORT decision if timed out; otherwise, return PENDING decision.

*2) Database:* // heron: should introduce database key, namespace, lock.

*3) Coordinator:* // heron: should describe sub-transaction.

*4) Cohort:*

### D. System Interactions (a.k.a Life of a Transaction)

This section describes the end-to-end life of a transaction without handling failures.

1) Client composes a transaction request and passes it to Coordinator through 'CommitAomticTransaction()'. Coordinator returns a transaction ID for the client to look up the transaction result later.
2) Coordinator does the followings in parallel: (a) Coordinator decomposes the transaction into sub-transaction and distributes them to the corresponding cohort through 'PrepareTransaction()'. (b) Coordinator asks Blockchain to trace votes for the new transaction through 'StartVoting()'.

3) Cohort executes a sub-transaction by forwarding the operations to the database up to the point before Commit(). Cohort writes an entry to both the undo and redo log.
4) Cohort votes to Blockchain through 'Vote()' if it wants to commit or abort the transaction.
5) Cohort sleeps until the expected transaction timeout time and gets the voting decision from Blockchain through 'GetVotingDecision()'.
6) If the voting decision is to commit the transaction, cohort commits the transaction onto the database and releases acquired locks; otherwise, undoes the transaction using the undo log and releases acquired locks. If a transaction is committed, the cohort writes the GET operation results to disk.
7) (Independent to the previous steps) Client has waited until the timeout time and requests the transaction result from Coordinator through 'GetTransactionResult()', using the transaction ID given in step 1.
8) Coordinator gets (1) the voting decision of the inquiry transaction from Blockchain through 'GetTransactionResult()' and (2) the results from the disk written by the cohort.

### *E. Handling Failures*

// How does the whole system deal with crash failures?

*1) Crash of coordinator:*

*2) Crash of cohort:*

*3) Data loss of database:*

*4) Slow blockchain:* We use Ethereum for our blockchain implementation. According to [this article], it takes around 15 seconds to 5 minutes to process a blockchain transaction on Ethereum, which indicates that the blockchain is relatively slow compared to other components within our system.

A slow blockchain affects the number of COMMIT votes to collect before the timeout time, which leads to more transactions to abort eventually; however, the blockchain slowness doesn't affect the promised atomicity. The client or the coordinator can specify a longer timeout time if needed.

## II. Implementation

### *A. Environment and Tools*

// Which tool, machine, etc.

*1) Blockchain:* We wrote our two-phase-commit smart contract with Solidity and developed it with Truffle [https://trufflesuite.com/]. With Truffle, we compile our smart contract and deploy it onto a local blockchain simulator.

To interact with the deploy smart contract, we need a smart contract client and be used by the coordinator and cohorts. Since we implement most of the code in C++ and failed to find a C++ smart contract client library, we use a JavaScript library (web3.js [https://web3js.readthedocs.io/en/v1.7.3/]) instead and add an additional RPC layer for our C++ code to use.

### *B. Testing and Evaluation*

// How do we know if the system is performing correctly?

*1) Blockchain Cost:* "Gas Used" is the fee the blockchain caller pays to the miner for processing the transaction. In the Truffle development environment, we simulate the scenario where we have two cohorts voting to commit, the total gas we pay is around 0.33 USD [reference: https://automatedwebtools.com/gwei-to-usd-calculator/].

## III. Discussion

// What have we done? What went well and what didn't? // Future directions?