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. Compared to the traditional version of 2PC protocol [2], our proposed 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 and horizontally scalable. This is achieved by delegating coordination and voting work to the blockchain. The delegation makes the coordinator stateless and allows many coordinator jobs to co-exist to balance the load and tolerate job failures. To cohort, adding a cohort to the system is as simple as making the coordinator be aware of the address of the newly added cohort. So our system is horizontally scalable.

3) Reliability: Our system has significantly better reliability than the traditional 2PC protocol for three reasons. First, the system leverages blockchain to maintain the coordination state. The stateless coordinator can easily recover as long as it knows the participating cohorts. Second, The coordinator is stateless and horizontally scalable. We can keep multiple jobs running concurrently and place a load-balancing layer on top of it. Third, with the introduction of $namespace$ (see II-A3), 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.

II. Design Overview

A. 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 II-A, the complete flow of handling a successful transaction in II-B, and how we handle failures in II-C.

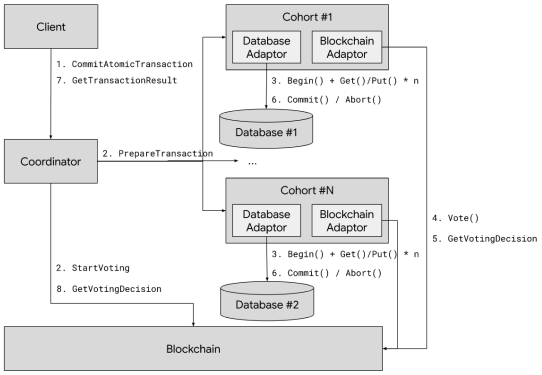

Fig. 1. Client starts a transaction request with Coordinator. Coordinator distributes the transaction to Cohorts and offload the 2PC responsibility to blockchain. Cohort interacts with the blockchain directly to vote and to look up a transaction status.

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.

We designed a `BlockchainAdapter` interface. It 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: The database component is responsible for committing sub-transactions in each cohort. It abstracts typical ACID databases with 5 APIs in a DatabaseAdapter

- `Begin()`: starts a transaction
- `Commit()`: commits a started transaction
- `Abort()`: aborts a started transaction
- `Put(key, value)`: puts value in a row whose primary key is key
- `Get(key)`: gets the value from the row whose primary key is key

We require the underlying database to be ACID because the database can handle locks and transactions natively without making cohorts implementing complicated locking logic for Put and Get operations. But for databases that only support weak snapshot isolation (e.g. LMDB [4]), the cohort only needs a single write lock to prevent other write transactions to be started to make sure that transactions are serialized.

3) Coordinator: The Coordinator exposes two APIs to the clients

- `CommitAtomicTransaction(CommitAtomicTransactionRequest, CommitAtomicTransactionResponse)`
- `GetTransactionResult(GetTransactionResultRequest, GetTransactionResultResponse)`

When a coordinator receives a `CommitAtomicTransaction` call from the client, it will first decompose the transaction into sub-transactions, which will be executed in the cohorts, based on the $namespace$ information. We shard the data key space into $namespaces$. Each cohort is responsible for a range

of keys under the same $namespace$. After transaction decomposition, it will send `PrepareTransaction` requests with the sub-transaction info to the cohorts. At the same time, it will send `StartVoting` to the blockchain to trace the voting status from the cohorts. Different from the traditional 2PC protocol, our coordinator is designed to delegate voting and coordination work to the blockchain. From here on, it no longer handles any work unless `GetTransactionResult` is requested from the client.

When the client sends `GetTransactionResult` to the coordinator, it reads the transaction status from the blockchain. If the status is $PENDING$ or $ABORTED$, it will return the status directly. If the transaction is $COMMITTED$, it will further send requests to the cohorts to fetch the Get result as well and return the status and the get results altogether. If not all cohorts return the Get result (e.g. due to crashes or network latency), the coordinator will send a partial response to the client with the ones that responded along with and indicator that the response is not yet complete.

4) Cohort: The Cohort exposes two APIs to the Coordinator

- `PrepareTransaction( PrepareTransactionRequest, PrepareTransactionResponse)`
- `GetTransactionResult( GetTransactionResultRequest, GetTransactionResultResponse)`

The $PrepareTransactionRequest$ contains a $transaction\_id$, a list of operations for the sub-transaction and an abort deadline. When a cohort receives a PrepareTransaction request from the coordinator, it takes these steps

1) Start a new thread in the thread pool.
2) Creates a `DatabaseAdapter` object.
3) Starts a transaction with `DatabaseAdapter ::Begin()`.
4) Do a series of requested Put or Get operations in the sub-transaction. If any operation is reported invalid by the database, the cohort will send an $ABORT$ ballot to the blockchain via `BlockchainAdapter::Vote()`.
5) If all operations succeed, it will store the got values in both an in-memory hashmap as well as persistently on disk keyed by the $transaction\_id$ to prepare for future `GetTransactionResult` calls.
6) Set a timer based on the input abort deadline.
7) If the timer reaches the deadline, it will send a request `BlockchainAdapter:: GetVotingDecision(transaction_id)` to the blockchain.
   a) If the blockchain states that the transaction should be committed, it will call `DatabaseAdapter::Commit()`.
   b) If the blockchain states that the transaction should be aborted, it will call `DatabaseAdapter::Abort()`.

When the cohort receives a `GetTransaction-Result` request, if it has committed the transaction, it simply reads the in-memory hashmap based on the $transaction\_id$ in the request and returns the got values as response. If it has not yet received a commit or abort response from the blockchain, it returns a pending response instead.

B. 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 time-out 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 memory or the disk written by the cohort.

C. Handling Failures

As described above, one of the main benefits of our system compared to traditional 2PC implementations is improved resiliency. By delegating the voting and coordination to the blockchain, our system will not block on any one machine that crashes or has a slow or partitioned network.

1) Crash of coordinator: The state of the coordinator includes only the list of which cohorts are involved in each active transaction. Thus if the coordinator crashes, when it recovers, it will respond to client requests for the Get responses of the transaction by indicating that it is unable to find the transaction. As a result, the client can directly ask the cohorts what the Get responses of the transaction are. Unlike in traditional 2PC, where the coordinator stores how each cohort has voted, in our design, this information is stored in the blockchain. Our system is resilient to such crashes, as the transaction does not need to block on the coordinator's recovery.

2) Crash of cohort: Our system is also resilient to crashes of cohorts, as these crashes do not cause any other cohorts to block for longer than the $vote\_timeout\_time$. If a cohort crashes before sending a vote to commit, then the blockchain will wait until the $vote\_timeout\_time$ and all cohorts will abort the transaction. If a cohort crashes after voting to commit and the blockchain decides that the transaction should commit, then the cohort will replay all transactions that committed while it crashed and recovered. To replay the transactions, the cohort will acquire the same locks it had for the transactions and send the relevant put and get requests to the database and commit the transactions in the database. Since it has already computed the response, if the coordinator asks for the get responses for any of these transactions, the cohort can safely return them, even if they have not yet committed in the database. This can be implemented in the future but we decided not to include it in our first version.

3) 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.

III. Implementation

In this section we first describe the implementation of our overall system. Then we'll illustrate the implementation of individual components.

A. Environment

We implement the coordinator and the cohorts with about 2800 lines of C++ code. For communications between components, we use gPRC and ProtoBuf. For providing the extensibility of swapping different database and blockchain implementations, we provide generic interfaces. We tested end-to-end behavior on a single machine with different processes simulating each component.

1) Blockchain: We wrote our two-phase-commit smart contract with Solidity and developed it with Truffle [https://trufflesuite.com/]. Truffle provides the development environment for us to compile smart contracts, test it, and deploy it locally for debugging. Once we've deployed our smart contract, the coordinator and the cohorts can interact with the contract through a smart contract client library. However, we didn't find a proper C++ smart contract client library to use, so we use a JavaScript library (web.js[https://web3js.readthedocs.io/en/v1.7.3/]) instead and add an additional gRPC layer to translate to C++.

While testing in the Truffle development, we learned the gas used for successfully committing a transaction with two cohorts is around 0.33 USD [reference] (see VI for breakdowns).

2) Database: We designed the DatabaseAdapter interface and implemented the methods with LMDB primitives. LMDB provides process-oriented functions like `open()`, `begin()`, `commit()`, `put()`, `get()`. Our object-oriented C++ implementation has almost 500 lines of code. We chose LMDB because it is fast, lightweight and easy to test in memory. But the interface is extensible to other databases like SQLite [6] as well.

3) Coordinator: The coordinator deterministically computes a global identifier for the transaction based on the client's provided identifier and the client's hostname and port. This allows the coordinator to avoid sending duplicate (potentially non-idempotent) requests to cohorts if the client sends the same request multiple times. This can happen if the client does not hear the response for its original request from the coordinator in time and tries to resend it. By using SHA256 to generate the transaction identifiers, it is extremely unlikely for transactions from different clients or with different client identifiers to have the same global identifier.

4) Cohort: One optimization we used for the cohort is that the `PrepareTransactionRequest` includes an indicator if this is the only cohort for the transaction. In such cases, we can avoid the expensive requests to the blockchain and the cohort can immediately commit a valid transaction as soon as it finishes preparing it in the database. This is safe because it knows that no other cohort can abort the transaction and that it is guaranteed to be atomic within its own database.

IV. Discussion

1) Latency tradeoffs: Our approach enables the client to tradeoff between latency and the likelihood of committing a transaction. When the client sets a longer time for timeout, it's more likely to collect sufficient votes from all cohorts to make a COMMIT decision but may have higher latency if a cohort crashes. There is a limit to the minimum timeout, which can be no shorter than the time it takes to finalize a new blockchain block.

As part of a future effort, we can extend the system to tradeoff between further latency reductions and safety by having cohorts send their votes to the coordinator as well as to the blockchain. If the coordinator receives commit votes from all cohorts, it can update the decision in the blockchain and tell all cohorts to commit. In this procedure, we would not need to wait for a new block to be mined in the blockchain to commit a transaction. The loss of safety comes from edge cases where the blockchain is slow to process the transaction and it ends up not receiving the votes in time and thus deciding to abort. If some cohorts received commit messages from the coordinator and others did not and then asked the blockchain for the decision, the cohorts who received commit messages would commit, while the others would abort the transaction. As we expect this to only affect a very small number of transactions, clients would be able to indicate if their transaction can tolerate a small chance of non-atomicity for much faster latency.

2) Distributed coordinator: Currently the coordinator is stateless, so it is easy to add new coordinator servers, as long as each one handles a disjoint set of transactions. However, if a coordinator crashes, then clients cannot ask other coordinators for the Get responses of a transaction, since those coordinators will not know how to find the cohorts for the transaction. As part of a future effort, we could have the coordinators send each other the cohorts for each transaction so that they can respond to client requests even if the original coordinator for that request crashes.

V. Reference

VI. Appendix

TABLE I
GAS USED, COST IN ETH/USD PER BLOCKCHAIN TRANSACTION FOR A SUCCESSFUL 2PC COMMIT WITH 2 COHORTS. DATA AS OF 2022/05/30.

	Gas Used (Gwei)	Cost in ETH	Cost in USD
StartVoting	73441	0.000073441	0.143
Vote 1	56451	0.000056451	0.110
Vote 2	39513	0.000039513	0.077