Two-Phase Commit Using Blockchain

Benjamin Marks
Stanford University
Stanford, California, USA
benmarks@stanford.edu

Heron Yang
Stanford University
Stanford, California, USA
heronyang@stanford.edu

Yuewei Na
Stanford University
Stanford, California, USA
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.

1 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 [1] to be to be inevitable even in synchronous systems where bounds on delays can be reliably estimated.

Since the invention of Bitcoin, the underlying blockchain technology has not only gained attraction from cryptocurrency systems like Ethereum [2] but also academic research about computer systems. At its core, blockchain is a protocol of reaching consensus by using Proof-of-Work. Previous work [3] has shown that a synchronous blockchain can increase the reliability of the original 2PC protocol and 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 make 2PC with blockchain work in production and at scale. Compared to the traditional version of 2PC protocol [4], our proposed approach increases extensibility, scalability and reliability.

1.0.1 Extensibility. We can plug any blockchain, regardless of whether it is public/permissionless or private/permissioned, into our blockchain interface which exposes `StartVoting()`, `Vote()`, `GetVotingDecision()`. We can also plug any database system that has ACID support into our transaction delegation interface which implements common transaction operations - `Begin()`, `Commit()`, `Abort()`, `Get()`, `Put()`.

1.0.2 Scalability. We design our coordinator to be light-weight to handle high throughput and to be 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. Adding a cohort to the system is as simple as making potential clients aware of the address of the newly added cohort, so our system is horizontally scalable.

1.0.3 Reliability. Our system has better reliability than the traditional 2PC protocol for two reasons. First, the system leverages the 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 to be tolerant of individual coordinator node failures.

2 Design Overview

2.1 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()`. The client creates a transaction and requests the coordinator to commit. The coordinator distributes the transaction to the corresponding cohorts to prepare and

then offloads the remaining 2PC responsibility to the blockchain. Each cohort prepares the transaction locally, votes to the blockchain if ready, checks on the blockchain to see if a transaction should be committed, and 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 offloads the 2PC voting responsibility to the blockchain.

We describe the details of each component in the remaining 2.1, the complete flow of handling a successful transaction in 2.2, and how we handle failures in 2.3.

2.1.1 Blockchain. The blockchain component performs a 2PC protocol, so it's responsible for tracking voting states of each transaction and making decisions of whether to commit or abort. The implementation uses Ethereum smart contracts to store the followings states of each transaction (indexed by the transaction ID) internally:

- `transaction_state` stores voting states of each cohort of a transaction.
- `transaction_config` stores the transaction configure of each transaction including the number of cohorts and the vote timeout time.

We designed a `BlockchainAdapter` interface, which exposes the following APIs externally to the coordinator and the cohort:

- `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 on the blockchain when the 2PC stops taking votes.
- `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 of 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:

1. If any `ABORT` vote is received, return an `ABORT` decision.
2. If all `COMMIT` votes are received, return a `COMMIT` decision.
3. If not (1) and not (2), return an `ABORT` decision if the blockchain time has passed the vote timeout time; otherwise, return a `PENDING` decision.

2.1.2 Database. The database component is responsible for committing sub-transactions in each cohort and 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 requiring cohorts to implement complicated locking logic for `Put` and `Get` operations. However, for databases that only support weak snapshot isolation (e.g. LMDB [5]), the cohort needs a single write lock to prevent other write transactions to be started to make sure that transactions are serialized.

2.1.3 Coordinator. The coordinator exposes two APIs to the clients:

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

The data key space is split into `namespaces`, each of which corresponds to the single cohort responsible for all keys in that range. When the coordinator receives a `CommitAtomicTransaction()` call from the client, it first sends `StartVoting()` to the blockchain to track the voting status from the cohorts. Then it decomposes the transaction into sub-transactions, which will be executed in the cohorts, based on the namespace of each operation. After transaction decomposition, the coordinator sends `PrepareTransaction()` requests with the

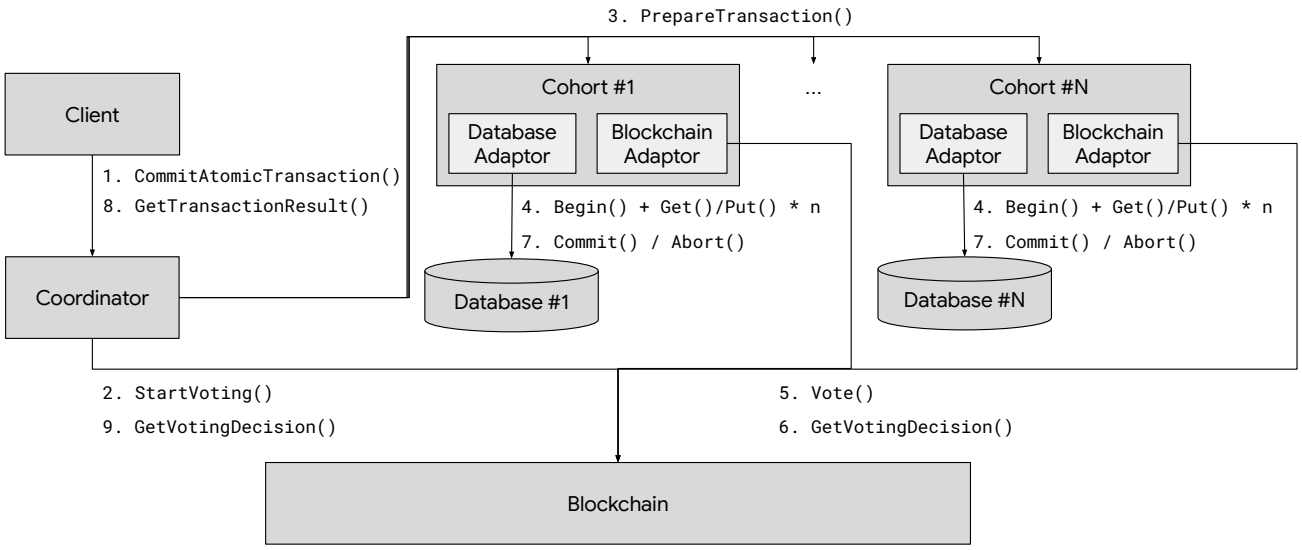

Figure 1. This diagram displays the main components within our architecture and example steps of successfully committing a transaction.

sub-transactions to the cohorts. Different from the traditional 2PC protocol, we design our coordinator to delegate voting and coordination work to the blockchain. From here on, the coordinator no longer handles any work unless `GetTransactionResult()` is requested from the client.

When the client sends `GetTransactionResult()` to the coordinator, the coordinator reads the transaction status from the blockchain. If the blockchain returns a `PENDING` or `ABORTED` status, the coordinator will return the status directly. If the transaction is `COMMITTED`, the coordinator will send requests to the cohorts to fetch the `Get` results and then return the status with the `Get` results. If not all cohorts return the `Get` results (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 an indicator that the response is not yet complete.

As described in 2.1.1, the coordinator does not need to persist any information to disk as all voting information is stored in the blockchain and all other information (e.g. list of cohorts involved in the transaction) is also known by the client.

2.1.4 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 of the sub-transaction and the vote timeout time, after which the blockchain will not accept votes. When a cohort receives a `PrepareTransaction()` request from the coordinator, it takes these steps:

1. Start a new thread in the thread pool.
2. Create a `DatabaseAdapter` object.
3. Start a transaction with `DatabaseAdapter::Begin()`.
4. Acquire the relevant locks for the sub-transaction.
5. Execute a series of requested `Put` or `Get` operations in the sub-transaction.
6. If any operation is reported invalid by the database or locks cannot be acquired in time, the cohort will send an `ABORT` ballot to the blockchain via `BlockchainAdapter::Vote()`.
7. If all operations succeed, it will store the retrieved `Get` values in an in-memory hashmap as well as persistently on disk keyed by the `transaction_id` to prepare for future `GetTransactionResult()` calls.

8. Set a timer based on the input vote timeout time.
9. If the timer reaches the vote timeout time, 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()` and release any acquired locks.
   b. If the blockchain states that the transaction should be aborted, it will call `DatabaseAdapter::Abort()` and release any acquired locks.

When the cohort receives a `GetTransactionResult()` request and 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.

2.2 System Interactions (a.k.a Life of a Transaction)

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

1. The client composes a transaction request and passes it to the coordinator through `CommitAomticTransaction()`. The coordinator returns a transaction ID for the client to look up the transaction result later.
2. The coordinator asks the blockchain to trace votes for the new transaction through `StartVoting()`.
3. The coordinator decomposes the transaction into sub-transactions and distributes them to the corresponding cohort through `PrepareTransaction()`.
4. Each cohort executes a sub-transaction by acquiring locks and forwarding the operations to the database up to the point before Commit(). The cohort writes an entry to the redo log to recover in case it crashes. Additionally, the cohort writes the `Get` operation results to disk for faster responses after a crash and recovery.
5. The cohort votes to the blockchain through `Vote()` indicating that it wants to commit or abort the transaction.
6. The cohort waits until the expected transaction vote timeout time and gets the voting decision from the blockchain through `GetVotingDecision()`.
7. If the voting decision is to commit the transaction, the cohort commits the sub-transaction onto the database and releases any acquired locks; otherwise, it aborts the sub-transaction in the database and releases any acquired locks.
8. (Independent of the previous steps) The client waits until the vote timeout time and requests the transaction result from the coordinator through `GetTransactionResult()`, using the transaction ID given in step 1.
9. The coordinator gets (1) the voting decision of the inquiry transaction from the blockchain through `GetTransactionResult()` and (2) the `Get` operation results from each cohort.

2.3 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 does not block on any one machine that crashes or has a partitioned network.

2.3.1 Crash of the 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, and the cohorts can commit or abort even while the coordinator is unresponsive.

2.3.2 Crash of a 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.

2.3.3 Slow blockchain. We use Ethereum for our blockchain implementation. According to [6], 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 increases the likelihood that not all commit votes can be collected before the vote 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 vote timeout time if needed.

3 Implementation

In this section we first describe the implementation of our overall system. Then, we illustrate the implementation of individual components. The source code of our implementation is available at licensed under GNU Lesser General Public License. Our implementation depends on the following libraries: truffle [7], gganacheanache [8], sqlite3 [9], grpc [10], web3 [11], abseil-cpp [12], rule_nodejs [13], gflags [14], glog [15], openssl [16], threadpool [17], lmdb [18], and lmdbxx [19].

3.1 Environment

We implemented the coordinator and the cohorts with about 2800 lines of C++ code. For communications between components, we used gRPC [10] and ProtoBuf [20]. For providing the extensibility of swapping different database and blockchain implementations, we provided generic interfaces. We tested end-to-end behaviors on a single machine with different processes simulating each component.

3.1.1 Blockchain. We wrote our two-phase-commit smart contract with Solidity and developed it with Truffle [7]. Truffle provided the development environment for us to compile smart contracts, test it, and deploy it locally for debugging. Once we deployed our smart contract, the coordinator and the cohorts interacted 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 used a JavaScript library (web.js[ref]) instead and added 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 [21] (see 4.0.2 for breakdowns).

3.1.2 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. The interface is extensible to other databases like SQLite [22].

3.1.3 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.

3.1.4 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 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.

4 Discussion

4.0.1 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 the vote timeout time, 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 and 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.

4.0.2 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 they can respond to client requests even if the original coordinator for that request crashes.

References

[1] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142, 1981.

[2] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[3] Paul Ezhilchelvan, Amjad Aldweesh, and Aad van Moorsel. Non-blocking two-phase commit using blockchain. *Concurrency and Computation: Practice and Experience*, 32(12):e5276, 2020.

[4] Butler Lampson and David Lomet. A new presumed commit optimization for two phase commit. In *19th International Conference on Very Large Data Bases (VLDB'93)*, pages 630–640, 1993.

[5] Gavin Henry. Howard chu on lightning memory-mapped database. *Ieee Software*, 36(06):83–87, 2019.

[6] EG Station. How long does an ethereum transaction really take.

[7] trufflesuite/truffle: A tool for developing smart contracts. crafted with the finest cacaos. https://github.com/trufflesuite/truffle. Accessed: 2022-06-03.

[8] trufflesuite/ganache: A tool for creating a local blockchain for fast ethereum development. https://github.com/trufflesuite/ganache. Accessed: 2022-06-03.

[9] Tryghost/node-sqlite3: Asynchronous, non-blocking sqlite3 bindings for node.js. https://github.com/TryGhost/node-sqlite3. Accessed: 2022-06-03.

[10] grpc/grpc: The c based grpc (c++, python, ruby, objective-c, php, c#). https://github.com/grpc/grpc. Accessed: 2022-06-03.

[11] Chainsafe/web3.js: Ethereum javascript api. https://github.com/ChainSafe/web3.js. Accessed: 2022-06-03.

[12] abseil/abseil-cpp: Abseil common libraries (c++). https://github.com/abseil/abseil-cpp. Accessed: 2022-06-03.

[13] bazelbuild/rules_nodejs: Javascript and nodejs rules for bazel. https://github.com/bazelbuild/rules_nodejs. Accessed: 2022-06-03.

[14] gflags/gflags: The gflags package contains a c++ library that implements commandline flags processing. https://github.com/gflags/gflags. Accessed: 2022-06-03.

[15] google/glog: C++ implementation of the google logging module. https://github.com/google/glog. Accessed: 2022-06-03.

[16] Not possible to fast-forward, aborting. https://github.com/openssl/openssl/. Accessed: 2022-06-03.

[17] Barak Shoshany. A C++17 Thread Pool for High-Performance Scientific Computing. *arXiv e-prints*, May 2021.

[18] Lmdb: Lightning memory-mapped database manager (lmdb). http://www.lmdb.tech/doc/. Accessed: 2022-06-03.

[19] drycpp/lmdbxx: C++11 wrapper for the lmdb embedded b+ tree database library. https://github.com/drycpp/lmdbxx/. Accessed: 2022-06-03.

[20] protocolbuffers/protobuf: Protocol buffers - google's data interchange format. https://github.com/protocolbuffers/protobuf. Accessed:

2022-06-03.
[21] Gwei to usd calculator. https://automatedwebtools.com/gwei-to-usd-calculator/. Accessed: 2022-06-03.
[22] ST Bhosale, Miss Tejaswini Patil, and Miss Pooja Patil. Sqlite: Light database system. *Int. J. Comput. Sci. Mob. Comput*, 44(4):882–885, 2015.

Table 1. 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
Total	169405	0.000169405	0.330