2-Phase Commit With a Blockchain Coordinator

Elliot Dauber and Isaac Cheriyuot
Stanford University

Abstract

The goal of this project is to remove the blocking aspect of the classic 2-phase commit protocol over a sharded database system. This most often becomes a problem when the coordinator of a transaction (an instance of 2-phase commit) shuts down. To solve this, we created a system where most of the coordinator functionality is pushed to an Ethereum smart contract, which stores the state and controls the data flow of all the 2-phase commit transaction, ensuring the client and nodes will always be able to determine the outcome of the transactions they care about. On top of this, we built a system for managing nodes in the system, adding new nodes, and recovering from node failure.

1. INTRODUCTION

Almost every application we use nowadays lives on a distributed systems. While this has allowed for an incredible increase in the scale and power achievable by these applications it has also introduced a wealth of new and challenging problems. One of the largest of these problems is achieving consensus. In any distributed system there is information spread across disparate nodes, where nodes can be anything from individual machines to server warehouses. When attempting to store information in such a system we need to guarantee that for any request all nodes involved in said request are capable of executing it and do so together, thus having consensus on the overall state of the system. The Two Phase Commit Protocol[2] is one solution to this problem. At a high level, this protocol consists of on coordinator who receives a request and a cohort of nodes that the request is for as well as two distinct phases of processing. In the first phase, the coordinator delegates commands to relevant nodes in the cohort to execute the request, of which they they vote on whether or not theses commands are executable. All nodes involved must vote to commit in order for the request to commit, at which time the coordinator will send a commit message to the involved nodes to which they must ack after executing the command before the request is considered fulfilled. A major problem with this system arises when the coordinator fails mid protocol. If the coordinator fails permanently, some nodes will never complete their transactions. After a node has sent an agreement message to the coordinator, it will block until a commit or rollback is received. In order to resolve this we have chosen to replace the coordinator with a blockchain contract[1] as the chain is guaranteed to persist where as any coordinator can fail. Here is the Github repo with the source code: https://github.com/elliotdauber/2pc-blockchain

2. RELATED WORK

We used two main pieces of literature to build our solution. The first is ”A New Presumed Commit Optimization for Two Phase Commit” one of the most foundational papers in terms of modern two phase commit protocols[2]. This paper informed our understanding of the two phase commit protocol and how to construct one from the ground up. The second piece ”Non-Blocking Two Phase Commit Using Blockchain” outlines a blockchain two phase commit protocol approaching the same problem from the same angle as us and thus we used this as a basis for our own implementation[1].

[1] We still have a coordinator but its only role is distributing transactions and their corresponding contract address

3. DESIGN

3.1. CLIENT DESIGN (API)

Here is the (very simple) client API:

```
makeRequest(queries=Query[],
            access=w,
            block=True)
```

Each Query object has two required fields. Here is the structure of the Query object:

```
struct Query {
  pk: String,
  sql: String
}
```

When the `makeRequest` API is invoked by the client, the pk of each query is hashed, and the request is sent over the internet to the system to be processed. Along with the request is the address of a smart contract on the Ethereum blockchain that is deployed by the client library upon invocation of the API (section 3.3). For simplicity, the design is currently such that each SQL query that is created must only touch one PK. This can be easily changed to include , but for the sake of time, we chose this design decision for the time being. However, if a client wants to make a single transaction that touches multiple PKs, it can simply submit a transaction with multiple query objects.

Aside from the list of Query objects that must be passed into each `makeRequest` API call, there are two optional parameters that have default values:

`access`: This flag sets the access type for the query – "r" (read) or "w" (write). By default, this is set to "w". It is most likely possible to infer the access type from the sql query itself, but for simplicity in our original design, we added this flag.

`block`: This flag tells the client library whether or not to wait for the transaction to be completed before moving on. This is most useful for write transactions that must be completed before subsequent write transactions can occur, and for read transactions where the outcome is to be used in a subsequent transaction.

On blocking transactions, the client receives a response from the system. For read transactions that commit, the data is sent back in the form `SQLResponse[]`, where each `SQLResponse` object is a list of the items that were read through the query. For read transactions that don't commit, the status of the 2pc protocol (abort or timeout) is returned as a string. For write transactions, the status is returned no matter what.

3.2. NODE DESIGN

Every node in the system can act as both a coordinator and cohort member in 2PC transactions, and can handle as many transactions as the process can physically handle. Below we break down the node design into its components

3.2.1 COORDINATOR DESIGN

When a node is sent a transaction from a client using the `makeRequest` API, it becomes the coordinator for that transaction. Then, using the directory structure it stores, it determines which nodes to send the work to. Note that the coordinator may be a part of the cohort, in which case it sends the data to itself. Each node stores a database shard for a specific set of PK's, and the coordinator only sends the work specific to those PKs to each node. This reduces the amount of data sent over the internet.

Once the coordinator determines which nodes (the cohort) to send the work to, it calls the `request` method (section 4.2) on the smart contract deployed by the client.

Once the contract is initialized, the coordinator sends the work, along with the contract address and some other data to each of the nodes in the transaction's cohort. The coordinator then replies to the client with the contract address and some other data.

3.2.2 COHORT DESIGN

Upon receiving work from a coordinator, each node in the cohort of a transaction does the following:

First, the node logs the address of the smart contract associated with the transaction, as well as the work to be done, to a log file on disk.

Then, the node determines whether or not it can do the work, based on whether or not the node has committed to transacting on any of the PKs in the transaction as well as if the table and entry being manipulated exist. Based on this decision, the node votes on the smart contract at the address provided by the coordinator. If the node votes to commit, it adds the PKs in the work to a set that is used to determine what work can be committed to in future transactions. These PKs are removed from the set when the node is sure that the transaction has committed or aborted.

Once the node has determined work can be done, it asynchronously waits on the Ethereum contract's state to be updated or for a set timeout to be reached. If all involved nodes vote to COMMIT, the contract's state will be changed to COMMIT and all nodes will know to commit the associate changes to their respective databases. Otherwise, the contract state will either be set to TIMEOUT or ABORT, causing involved nodes to abandon the associated work.

3.2.3 DIRECTORY DESIGN

Each node stores a data structure called a Directory that stores data concerning which nodes contain the database shards for any given PKs. This structure uses consistent hashing to store and update the information, using virtual nodes to reduce disruption when adding or removing a node[2] from the system.

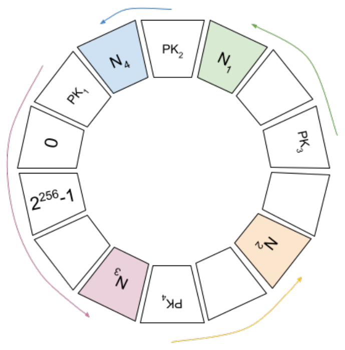
We implement this by mapping randomly selected unique SHA256 hash values to nodes where the number of hashes associated with a given node defines our number of "virtual nodes". From here, to find the node associated with a pk we take the SHA256 of our given pk x and find the closest hash value less than x that is mapped to a node. This node is responsible for the data associated with our given pk. Thus we can add and remove hash to node mappings to quickly add and remove recognized nodes in our system. This can all be seen in figure 1.

The Directory plays a central roll in allowing both the coordinator node to send work to the cohort nodes and allowing the cohort nodes to redistribute data when necessary.

3.3. BLOCKCHAIN DESIGN

A new smart contract is created for every transaction that a client initiates (one call from a client of the `makeRequest` API). In other words, there is a separate contract on the blockchain that handles each individual transaction.

[2]We actually only store the URL of the node as the Directory is primarily used for informing inter-node communication



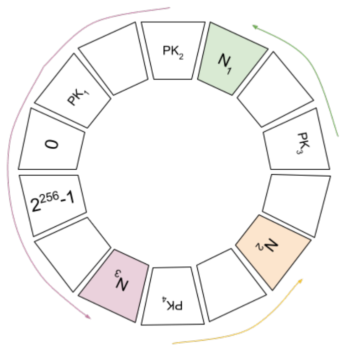


Figure 1. (Bottom) We glue 0 and $2^{256} - 1$ together, so that objects are instead assigned to the node that is closest in the counter-clockwise direction. This solves the problem of the last object being to the left of the last Node. (Top) Adding a new node N_3. PK_2 is reassigned from N_3 to N_4.

Each smart contract stores the state of the transaction, and thus can be viewed as a state machine. The possible states the contract can be in are INIT, VOTING, COMMIT, ABORT, and TIMEOUT. The contract also stores a string called data, which contains the results of the SQL transactions for read requests. Individual query results are separated by double semicolons (";;").

Each contract stores a timeout, which is determined by the coordinator of the corresponding transaction. This timeout is used to activate the TIMEOUT state if a transaction has expired.

3.4. SYSTEM FLOW

Figure 2 is a diagram of the entire control and data flow of the system. The number labels correspond to the order of communication for a single write transaction. Below is the action at each label. Note that for read transactions, there is an extra step 8.5 where the nodes send their read results to the contract using the its `set_data` function.

1. The client deploys a smart contract for the transaction on the Ethereum blockchain.
2. The Ethereum blockchain responds with the address of the contract.
3. The client sends a transaction to the coordinator using the MakeRequest API.
4. The coordinator calls the `request` method on the smart contract deployed by the client.
5. The coordinator responds to the client, signifying that it has launched the smart contract.

6 6. The coordinator sends the work to the nodes. Its work as a coordinator is now complete. Note that the coordinator is not part of the coordinator in this example, but that is a possibility.

7. The nodes log the smart contract address, as well as the actual work to be done.
8. The nodes vote on the transaction using the contract's `voter` method, then periodically call its `getState` method until there is a verdict.
9. The smart contract responds with the verdict.
10. If the verdict is COMMIT, the nodes transact on their database shards.
11. The nodes respond to the client that they have completed the transaction.
12. If the transaction times out on the client side (it doesn't receive responses from all cohort nodes within the timeout set by the coordinator), it queries the smart contract for the verdict.
13. The smart contract responds with the verdict.

3.5. LIVENESS AND CORRECTNESS PROOF

We provide here a proof of the liveness of this system – that is, that the system cannot block indefinitely regardless of node failure. Note that this protocol does not protect against client crashes or failures.

We begin with the client sending a transaction to the coordinator using the `makeRequest` API. If the coordinator crashes before sending a response to the client, the client will timeout and check the smart contract for the verdict. Under normal operation, the coordinator first sets a timeout, which we will call Δ. Next, at time t_0, the coordinator initializes the smart contract using its `request` method and passes in Δ. Let's denote the time that the smart contract starts it timer as t_b Then, the contract will consider the transaction timed out at time $t_0 + \Delta$.

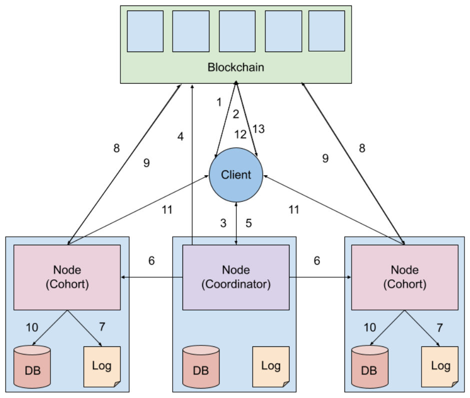


Figure 2. The flow of communication throughout the entire 2-phase commit system for a write transaction. Note that each of the nodes are running the same code, but that the middle node is the coordinator for this particular transaction, while the other two nodes are the cohort for this particular transaction.

After this happens, the coordinator sends the timeout, along with other data, to the other nodes and the client. Let's denote the time the client receives the timeout t_c and the time node i receives the timeout as t_{n_i}. We notice that $t_0 < t_b < t_c$ and t_{n_i}.

For blocking transactions (set by the client through the `makeRequest` API), the client waits for responses from each of the nodes in the cohort for the given transaction for Δ time. At time $t_c + \Delta$, if the client has not received all of these responses, it checks the smart contract for a verdict. The contract must have either reached a verdict or timed out by time $t_c + \Delta$ because $t_c > t_b$. Therefore, even if any nodes crash, this protocol will reach an outcome.

We will now show that when a client reaches an outcome, it is guaranteed that the cohort nodes for a transaction will have already seen that same outcome.

When the cohort nodes receive their work, they vote on whether or not they can do the work. For blocking transactions, if node i votes to COMMIT, it will then repeatedly query the smart contract for a verdict. If it sees a COMMIT or ABORT verdict before the contract times out, it will send this verdict to the client. If this message is the last one required for the client to verify a verdict, it is guaranteed that the client will see the same outcome as all of the nodes.

However, if node i sees a TIMEOUT, it is guaranteed that the client will also see a TIMEOUT because both t_c and t_{n_i} are greater that t_b. Once a contract is in the TIMEOUT state, it cannot ever change state again, so even if a node votes on a contract after timeout has occurred, this will not affect the state of the contract, and both the client and node will abort the transaction.

We have shown that due to the communication flow and use of timeouts in our system, the algorithm is both live and correct.

3.6. DYNAMIC NODE ALLOCATION

Understanding that in practice, any user may want to scale or move a given database we added support for dynamic node count. This allows for the addition or removal of nodes to the system on the fly automatically transferring data when necessary. In order for a node to be added there must be no ID or URL conflicts and a given node must first be able to ingest all the data already assigned to pk's in it's range.

4. IMPLEMENTATION

The vast majority of our system is implemented in Python, though the smart contract is implemented in Solidity, for use on the Ethereum blockchain. We realize that Python is slow compared to a language like C++ or Go, but since this is our first time building a distributed system, and there is a really nice library for interaction with the Ethereum blockchain in Python, we decided to write it in Python.

4.1. COMMUNICATION

In order to ease communication between the nodes in our system (including all of the 2pc nodes and the client nodes), we used the Python library GRPC, which is a library for defining and executing remote procedure calls (RPCs). Below we describe the various RPCs that we implemented to facilitate communication in our system:

`SendWork`: This is the RPC that is invoked when the `makeRequest` API is invoked at the client. This request is sent to a random node (which acts as the coordinator for the corresponding transaction), and contains the following data: An array of Query objects, with the PK hashed using SHA256; The url of the client, so that the request can be responded to; The access type of the transaction ("r" or "w"); The address of the smart contract associated with the transaction; The coordinator node responds to this RPC with the timeout that it set for the transaction, and the number of nodes that are required for the transaction to be considered complete (the cohort size).

`ReceiveWork`: This is the RPC that is invoked by the coordinator for a transaction once it has received a `sendWork` RPC from a client. After determining which nodes to send the work to (described in section 3.2.3), the coordinator node sends this RPC to each of the nodes that must carry out the work. It contains the following data: An array of Query objects. This is a subset of the query objects that the coordinator received in the `sendWork` RPC, specifically only the Query objects that are to be transacted on the given node; The address of the smart contract that the coordinator deployed, corresponding to the transaction that is being carried out; The timeout that the coordinator set for the transaction; The url of the client that started the transaction; The access type of the transaction ("r" or "w");

`AddNode`: This is the RPC that is invoked by any node looking to join a running system. It makes the call to only one node in the existing system, lets call this node the orchestrating node, with only information describing itself (i.e. ID, URL, etc...). The orchestrating node first checks for any ID or URL conflicts in the Directory. If none exist, the orchestrating node proceeds to set aside three unused virtual nodes from our Directory. With these virtual nodes the orchestrating node proceeds to send a more detailed AddNode RPC to each node whose virtual nodes overlap with the newly proposed ones. This more detailed RPC includes the new nodes metadata, the proposed virtual nodes, and the old virtual node that overlaps one of the proposed ones. Nodes that receive the detailed AddNode request, skip the aforementioned processes and immediately retrieve any transactions from their log that include pks that fall under the proposed virtual nodes. If there are any such transactions the node proceeds to make a `MoveData` request back to the new node. If that completes successfully the new node is added to the nodes directory and it returns a success. If all nodes return success the orchestrating node also adds the node to its directory and hands off the directory information to the new node.[3].

[3]We want to note that while we thought adding dynamic node allocation would a be cool feature our current implementation is

`MoveData`: This RPC can be invoked by any node when going through the processes of adding a new node to the system, upon receiving the request the new node attempting to join the system simply performs the transactions in the request, ignoring the overhead of the ReceiveWork RPC.

4.2. BLOCKCHAIN IMPLEMENTATION

We based much of our smart contract implementation on the paper[1] we read, while also adding a few new features. The smart contract is implemented as such:

`request(num_nodes, timeout)`: Initiates the transaction, setting the state to VOTING, and setting the number of nodes participating in the transaction and the timeout for the transaction.

`voter(vote, nodeid)`: Called from the cohort nodes, this function casts a vote on the transaction (1 for commit, 2 for abort). Once the vote has been counted, if the contract determines that the number of votes $\geq$ the number of nodes in the transaction, the state is switched to COMMIT, unless any node has voted to abort, in which case the state is switched to ABORT.

`set_data(data)`: Appends the data provided by a node to the data string on the contract.

`verdict()`: Forces a verdict on the contract. Specifically, if the contract is in the VOTING state and the timeout has expired, the contract is set to the TIMEOUT state. This is only called by a node or the client once it is sure the timeout has expired.

`getState()`: Returns the current state of the transaction as a string.

`getData()`: Returns the data associated with the transaction as a string.

5. EXPERIENCES AND LESSONS LEARNED

We learned a lot about building a system from the ground up that has a lot of moving parts that all need to be able to handle failures. That being said, there are still many ways that our design and implementation can be improved.

vulnerable to mid stage failures crippling the replication process for the Directory or databases

One area for improvement is how we think about abstractions. For example, we currently have the client deploying the smart contracts and checking the them for transaction status and data outcomes on timeout. However, this is slightly strange because blockchain credentials would need to be stored on the client, which could be unsafe. It would be cleaner to implement a solution where the client doesn't need to touch the blockchain, but there is a tradeoff with the amount of the work the coordinator should be doing.

6. CONCLUSIONS

We have improved the classic 2-phase commit protocol by implementing a non-blocking version that relies on the Ethereum blockchain to serve as a coordinator in transactions. Now, hen nodes go down or are unresponsive due to network delays, clients can still determine the outcomes of their transactions.

This project is a proof-of-concept and is not aiming to beat any speed records in relation to the classic 2-phase commit protocol. This is supported by the design choice to implement the project in Python – we wanted to be able to prototype and test these features quickly, but were not focused on performance in the context of speed. Thus, we have not provided any quantitative evaluation of this project.

There is still work to be done to make this a system that would work in a production environment, but this project serves as a proof of concept for the new 2-phase commit protocol. The costs to deploy and transact on these Ethereum smart contracts is impractical for a high volume of requests. However, as blockchains are becoming an increasingly popular tool, it is reasonable to assume that these costs will significantly decline in the future, making this system more practical.

Additionally, this project shows the potential for the blockchain to be utilized for even more aspects of the system. Looking forward, we could see the blockchain being used for node management, client-node communication, logging, and more. Blockchains are an extremely powerful tool for distributed systems engineering because of their guarantees, and this project is an example of how to get started about thinking about blockchain integration into classic distributed systems problem spaces.

References

[1] P. Ezhilchelvan, A. Aldweesh, and A. V. Moorsel. Non-blocking two phase commit using blockchain, 2018. https://dl.acm.org/doi/10.1145/3211933.3211940. 1, 6

[2] B. Lampson and D. Lomet. A new presumed commit optimization for two phase commit, 1993. https://www.vldb.org/conf/1993/P630.PDF. 1