Adaptive Cross-Network Transactions With Node.js and Ethereum

ISHAAN GULATI, JACK SLOANE, and NEHA SURANA, Virginia Tech, USA

Blockchains have desirable properties such as transparency, traceability, and trustability. These properties make blockchains a popular choice for many businesses and applications. There are, however, several limitations when it comes to applying this technology more widely. Scalability is debatably the most significant limitation as blockchain networks like Ethereum continue to grow. Transaction throughput is dependent on the network congestion; if more nodes are added to expand the network, the transaction speed is impacted immensely. A highly effective way to address such scalability problems is to use sharding, wherein the network is partitioned into smaller, independent shards which are each responsible for an independent subset of the transactions and state. In this paper, we present a Node.js-ethereum framework which operates atop two private Ethereum blockchain networks to facilitate and mediate adaptive cross-network transactions - based on the principles of sharding and cross-shard transactions. Our framework for adaptive synchronous and asynchronous cross-network transactions is generic and extendable to be used with real-world Ethereum-based digital coins on active private networks.

ACM Reference Format:
Ishaan Gulati, Jack Sloane, and Neha Surana. 2022. Adaptive Cross-Network Transactions With Node.js and Ethereum. *ACM Trans. Graph.* 37, 4, Article 111 (March 2022), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Blockchain technology is developing quickly and being adopted for use in many different fields and applications. Although the interest in and development of blockchain technology can be primarily attributed to the cryptocurrency Bitcoin, Ethereum is the most pervasive general-purpose blockchain. As the usage and scale of blockchains like Ethereum continue to grow, some limitations are preventing blockchain technology from being applied more widely. These scalability limitations stem from the three fundamental tasks of nodes in a blockchain network: The nodes must process and validate transactions with an acceptable degree of time-overhead; the nodes must relay validated transactions and finished blocks to all the other nodes in the network quickly enough to enable an ongoing global consensus, and the nodes must each store the current and historical state of the whole ledger [Skidanov and Polosukhin 2019]. These requirements require computing power, network bandwidth, and storage capacity, respectively. Of these requirements, compute power for processing and validating transactions is the biggest bottleneck in real world-applications of large blockchain networks. For instance, Ethereum can only process about 20 transactions per second, whereas Visa, a centralized network, can handle 65,000 transactions per second.

Authors' address: Ishaan Gulati, ishaangulati97@vt.edu; Jack Sloane, sloanej@vt.edu; Neha Surana, nehasurana@vt.edu, Virginia Tech, Blacksburg, Virginia, USA, 24061-0002.

0730-0301/2022/3-ART111 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

One approach to addressing this scaling problem is called sharding. In a sharded blockchain network, the nodes are divided into smaller groups or shards, and each group is in charge of processing an independent subset of the transactions and storing an independent subset of the global state. In short, sharding is when a blockchain is divided into multiple parallel blockchains [Liu et al. 2021]. In any practical implementation of sharding, it is crucial that the system has a mechanism for cross-shard transactions. That is transactions that involve multiple shards. Without an infrastructure for cross-shard transactions, a sharded blockchain is as good as two independent blockchains. Blockchain scalability improvements via sharding is an ongoing area of research, and multiple cross-shard transaction solutions exist. Some sharding schemes adopt the two-phase commit (2PC) protocol to implement cross-shard transactions. This protocol is atomic and blocking, requiring nodes across both shards to stop what they are doing and collaborate on validating the cross-shard transaction and produce the resulting block in each respective chain. As such, while 2PC's atomic nature makes it desirable, it has inherent limitations of higher latency and lower throughput for cross-shard transactions. Other sharding schemes exist where-in a cross-shard transaction is executed in both shards asynchronously, the "Credit" shard executing its half once it has sufficient evidence that the "Debit" shard has executed its portion [Skidanov and Polosukhin 2019]. An asynchronous protocol allows for lower latency and higher throughput than a synchronous protocol, but has an atomicity problem; there is a chance that a block resulting from the cross-shard transaction could be orphaned on one shard but not on the other, resulting in a transaction that is only partially applied.

2 OVERVIEW

Our project is essentially a Node.js-Ethereum framework in which a Node.js application operates atop two Ethereum private blockchain networks to facilitate and mediate adaptive cross-network transactions. This Node.js application interacts with the blockchains by using RPC to invoke smart contracts and access their state. Our application adopts the concepts of cross-shard transactions but works at a higher level - where separate blockchains embody shards, and a Node.js Mediator embodies the concept of the "coordinator/beacon chain" (used in many low-level sharding implementations). In this sense, our system is an adaptable, and scalable solution for connecting two private Ethereum networks, meant for users who are creating a multi-network distributed application, and need an effective way to bridge the two chains together to perform cross-chain transactions involving the transfer of custom smart contract resident tokens.

We aim to design and implement a prototype, high-level interpretation of a sharding scheme capable of adaptive cross-network transactions in a simplified, controlled environment. Our "sharding" scheme and adaptive cross-shard transaction mechanism are generic in terms of design principles, but the implementation is specific to Ethereum and Ethereum-compatible tools. As we said, our prototype system is a simple approach to sharding where-in two private Ethereum blockchain networks act as the shards, and a Node.js Mediator takes on the role of "beacon" or "coordinator" chain. In real-world applications, a beacon/coordinator chain may have many responsibilities including validator assignment to shards, rebalancing shards, and in general performing operations necessary for the maintenance of the entire network (NearNigthShade). In our simplified prototype, the Node.js Mediator is responsible for enabling adaptive cross-network transactions by evaluating the overall network traffic and coordinating asynchronous and synchronous cross-shard transactions accordingly. To facilitate cross-network transactions, the Mediator interacts with the two Ethereum networks by using an Ethereum RPC API for Node.js. In addition to the Mediator, another Node.js program called the Simulator handles the preparation and housekeeping necessary for testing the cross-chain transactions. The Node.js Simulator simulates an active blockchain environment by automatically performing randomized, ongoing transactions within each shard-chain at varying rates.

Our synchronous cross-chain transaction protocol is based on the two-phase commit protocol, often abbreviated as "2PC". In the "prepare phase" of our 2PC cross-shard transaction protocol, the creator of the transaction, which we call the client, broadcasts the transaction information to the input chain from the Node.js Mediator. In reality, the client would be a real person, but for development purposes, the Mediator itself assumes the role of the client - automatically and repeatedly initializing cross-network transactions. Upon receiving the transaction from the client, the input chain first validates it by checking whether the input account has enough balance to credit for the transaction. If so, the input shard locks the input account, so that no future transaction can involve the input account until it is unlocked, and return-broadcasts a certificate of readiness to the Node.js Mediator. If the input of the transaction is found to be invalid, the input shard return-broadcasts a certificate of rejection. Next, in the "commit phase" of the cross-shard transaction, the client waits for the "ready" or "reject" certificate from the input chain. If it receives "ready", the client broadcasts the transaction to the output shard, which then adds the desired balance to the appropriate account and sends a confirmation certificate to the client. The client then relays the confirmation certificate to the input chain, which finally spends the transaction amount and unlocks the input account for future transactions.

For asynchronous transactions across shards, we simply want the "credit" shard to execute its part of the transaction once it has sufficient evidence that the "debit" shard has executed its portion. In our implementation, the Node.js Mediator orchestrates asynchronous cross-chain transactions much like how it does for synchronous cross-chain transactions; however, now the Node.js Mediator does not need to block both chain and wait for a unanimous confirmation from the both networks. In short, the synchronous implementation uses a js-mediated atomic-swap-like behavior which can be described as: "Alice receives X amount of coins in network A and Bob gets deducted X amount of coins in network B, and both of these changes happen at the same time or not at all". Alternatively, the asynchronous implementation is non-atomic, but if the transaction validation fails or is rejected on one network, an "abort" message is relayed back to the other network and the transaction is reversed.

3 DESIGN

The architecture of our overall system is shown in Figure 1. We use two private Ethereum networks which contain their own respective set of accounts, each with coin balances managed by a simple smart contract-based dApp - this is a custom smart contract-resident digital coin, separate from Ether. Additionally, two smart contracts are deployed on both blockchains. One of the contracts, called `Coin`, is responsible for minting an initial coin balance to all accounts, transacting coins between accounts, providing balance information, and otherwise managing the account coin balances. The other smart contract, called `CrossNetworkTransaction`, is responsible for executing the intra-network steps of our synchronous and asynchronous cross-network transaction protocols, and storing the transaction information of intermediate steps in the contract's state. The specifics of these smart contracts are explained in detail in the implementation section.

The main job of the mediator is to automatically facilitate cross network transactions synchronously and asynchronously. A major responsibility of the mediator is to evaluate the "busyness" of the private Ethereum networks in order to adaptively change the cross-network transaction protocol between synchronous and asynchronous. If the mediator receives a request for a cross-network transaction while the networks are busy, it will perform the transaction asynchronously; and oppositely, if the networks are relatively un-busy, it will perform the transaction synchronously. In our project, the mediator uses system level stats of CPU and memory usage as an approximation to decide whether the transactions should be processed asynchronously or synchronously. To get the system level stats we use the `os` Node.js module. `os.loadavg()` returns the usage of each CPU core, `os.totalmem()` returns the total system memory, and `os.freemem()` returns the free system memory.

The Simulator automatically performs random transactions to simulate a real network activity. It's a recurring loop of execution turned on to start the simulation of both private networks. There are multiple simulation loops that run concurrently by this Simulator. Simulator is written using Node.js, and uses web3.js to interact with blockchains and Atomics to facilitate atomic read and write operations between concurrent accesses to the accounts. Process of

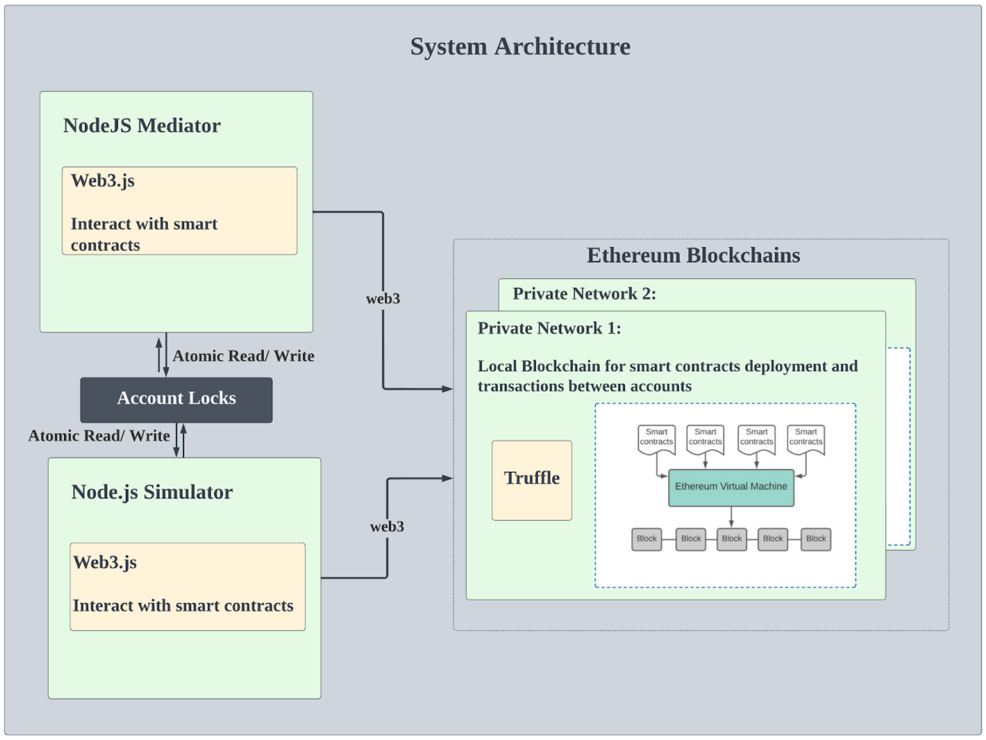


Fig. 1. System Architecture

simulation starts with fetching and unlocking all the accounts on both of the blockchains, followed by a simulation loop. Each iteration of simulation uses a Mediator to decide whether the transaction is to be performed synchronously or asynchronously.

During our development so far, there have been a few key tools which we have used to begin the implementation of our system. Go Ethereum (Geth) is the implementation of the Ethereum protocol written in the Go language. It offers a command line interface for configuring and creating a private network, and for configuring and running an Ethereum node. Next, the Truffle Framework provides a set of in-built tools which we use for smart contract compilation, linking and deployment. Truffle supports the rapid development of decentralized applications by providing automated contract testing capabilities. It also enables users to directly communicate with the smart contracts using its powerful interactive console. Lastly, we use the Web3.js API to communicate with the Ethereum network and invoke smart contracts on the blockchain from Node.js. Web3.js is a collection of libraries that enables us to interact with local Ethereum nodes using an HTTP, IPC or WebSocket connection.

3.1 Synchronous Cross-Network Transaction Protocol

In this section we describe our synchronous cross-network transaction protocol. The steps of our protocol are heavily based on the two-phase commit protocol for distributed atomic transactions, but there are some differences made to adapt it to the context of cross-blockchain transactions. Additionally, the low-level implementation details are specific to Ethereum smart contracts and the web3.js API for interacting with an Ethereum network using Node.js. This project focuses on cross-network transactions, based on the principles of cross-shard transactions. As such, the custom digital coin which we implemented in smart contracts on our private networks is a simplification of a real-world digital coin - it is a means to implement cross-network transactions. Regardless, our protocols for synchronous and asynchronous cross-network transactions are generic and extendable to be used with real-world Ethereum-based digital coins and dApps on active, real-world private networks.

3.1.1 Commit Request Phase

(1) The Mediator sends a “prepare message” to both networks, and waits until it has received a reply from them both (after the next two steps are completed).

(2) Locks are placed on the accounts involved in the transaction. The sender network verifies sufficient balance for the transaction. Both networks write an entry to their undo log (in case the one or both networks reject). Both networks execute the transaction up to the point where they will be asked to commit.

(3) Both networks reply with an agreement message if the preparation actions succeeded, or an abort message if the transaction verification failed, or any other problem occurred that will make it impossible to commit.

3.1.2 Commit Phase

Success: If the Mediator received an agreement message from all participants during the commit-request phase

(1) The Mediator broadcasts a commit message to both networks.

(2) Both networks commit the transaction.

(3) Both networks send an acknowledgement to the Mediator.

(4) The Mediator releases all the locks and resources held when both acknowledgements have been received, finishing the transaction.

Failure: If either network replies "abort" during the commit-request phase

(1) The Mediator broadcasts a rollback message to both networks.

(2) Both networks undo the transaction using the undo log.

(3) Both networks send an acknowledgement to the Mediator.

(4) The Mediator releases all the locks and resources held when both acknowledgements have been received, finishing the transaction.

3.2 Asynchronous Cross-Network Transaction Protocol

In this section we describe our asynchronous cross-network transaction protocol. For an asynchronous cross-network transaction, the "Credit" network executes its half once it has evidence that the "Debit" network executed its half. Since both of these halves happen asynchronously, there is a non-zero chance that one half of the transaction will be orphaned on one chain, while the other half becomes canonical on the other chain. This risk is inherent to the asynchronous protocol, but measures can be taken to resolve an instance of this atomicity failure. So, in conjunction with our asynchronous protocol, we provide a mechanism for reversing a cross-network transaction - in the case of atomicity failure. The high level algorithm for asynchronous cross-network transactions is as follows:

(1) The Mediator sends a message to the "debit" network to commit its portion of the transaction (deduct coins), and waits for a reply. The "debit" network replies "success/failure" depending on whether the account involved had sufficient balance and committed the transaction, or had insufficient balance. If the debit amount is committed, the transaction is logged to an "undo log" which can be used to reverse the transaction later, if needed.

(2) If the Mediator receives the "success" response from the debit network, it sends a message to the credit network to commit its portion of the transaction (add coins). Again, once the debit amount is committed, the transaction is logged to an "undo log" which can be used to reverse the transaction later - if needed.

4 IMPLEMENTATION

4.1 Simulator and Mediator Implementation

The Simulator and Mediator are implemented using Node.js and provide an Express.js app server for users to trigger the simulation operation. Simulation process starts with fetching and unlocking all the accounts in both the blockchains, followed by a loop(transaction loop) for creating multiple transactions. During each iteration of the transaction loop the Mediator decides whether the transaction should be executed asynchronously or synchronously based on system level stats of CPU and memory usage.

```
const transactions = [];
for (let i = 0; i < 500; i++) {
    // force is used to run the transactions in user specified way.
    if (force !== 'true' || force === false) {
        type = mediate();
    }
    if (type == 'SYNC') {
        transactions.push(makeSyncTransactions({
            chain1Web3, chain2Web3,
            chain1Contract, chain2Contract,
            chain1Accounts, chain2Accounts
        }));
    } else if (type === 'COMBINATION') {
        if (i % 2 == 0) {
            transactions.push(makeSyncTransactions({
                chain1Web3, chain2Web3,
                chain1Contract, chain2Contract,
                chain1Accounts, chain2Accounts
            }));
        } else {
            transactions.push(makeAsyncTransaction({
                chain1Web3, chain2Web3,
                chain1Contract, chain2Contract,
                chain1Accounts, chain2Accounts
            }));
        }
    } else {
        transactions.push(makeAsyncTransaction({
            chain1Web3, chain2Web3,
            chain1Contract, chain2Contract,
            chain1Accounts, chain2Accounts
        }));
    }
}
await Promise.all(transactions); // concurrently run the promises.
```

The above code snippet shows our simulation logic. The simulator calls the `mediate` function to determined whether to perform a group of cross-network transactions according to the synchronous, or asynchronous protocol. The `mediate` function follows the strategy behavioral pattern and decides which protocol to use at runtime. We then use the `Promise.all` function to perform multiple transactions concurrently.

```
function mediate() {
    const memoryUsagePercentage = ((os.totalmem() - os.freemem()) / os.
          totalmem()) * 100;
    const [cpuLoad] = os.loadavg();
    if (memoryUsagePercentage <= 45 && cpuLoad <= 3.5) {
        return 'SYNC';
    } else if ((memoryUsagePercentage >= 45 && memoryUsagePercentage <=
          65) && (cpuLoad >= 3.5 && cpuLoad <= 55)) {
        return 'COMBINATION';
    }
    return 'ASYNC';
}
```

The `mediate` function is responsible for deciding whether to execute a group of cross-network transaction asynchronously, synchronously, or execute a combination of asynchronous and synchronous transactions. It uses `os.loadavg()` to get the average CPU load over the past one minute, `os.totalmem()` to get the total system memory, and `os.freemem()` to free system memory.

4.2 Synchronous Cross-Network Transactions

In this section we describe, in detail, the implementation of our synchronous cross-network transaction protocol. Recall that our synchronous protocol consists of two phases: The commit request phase, and the commit phase. We organize the implementation-description of these phases according to the numerical steps described in the design section 3.1. The commit request phase of the synchronous protocol has three numerical steps, the commit phase of the synchronous protocol is divided into "success" and "failure" parts, each with four numerical steps.

4.2.1 Commit Request Phase:

(1)
  (a) The "prepare message" sent to one network includes the address of the account involved in the transaction that is resident to that network, and the transaction amount.
  (b) The "prepare message" is sent via a function call to both networks' `CrossNetworkTransaction` smart contract. This smart contract has been previously deployed on both networks. This function is defined as follows:
  `prepare(address account, int amount)`

  The function takes the parts of the message listed above as arguments. The js Mediator discerns which account is resident to which network, and includes only that one proper address in the function call to each respective network's `CrossNetworkTransaction` smart contract. Additionally, the js Mediator passes a negative `amount` to the network with the sending account, and a positive `amount` to the network with the sending account.

(2)
  (a) Locks are placed on the accounts involved in the transaction → Before invoking the `CrossNetwork-Transaction` contract's `prepare()` function, on both respective networks, the js Mediator generates a unique transaction id, and places a lock on the both accounts involved in the cross-network transaction. The transaction id is used like a password to take and free a given account lock. This prevents any other cross-network transaction request from involving that locked account until the cross-network transaction request which placed the lock finishes, and removes the lock using the transaction id. The js Simulation program uses the same accounts locks, and does not prompt a random transaction until attaining the locks for both accounts involved.
  (b) The sender network verifies sufficient balance for the transaction → If the `amount` passed to `prepare()` is positive, then the account is receiving coins, so there is no further verification required. If the `amount` is negative, then the account is sending coins, so we must verify sufficient funds for the transaction. In the body of the `prepare()` function, we call a function of the `Coin` contract (the state of which the account balances reside) called `getBalance(address account)`. If the account balance returned is less than the amount required, the `prepare()` function must return an abort message (see part 3 for details). If the account balance is sufficient, the function proceeds to execute the transaction up to the point where it will be asked to commit (see part 2.c for details).
  (c) . Both networks execute the transaction up to the point where they will be asked to commit → in the `prepare()` function of the `CrossNetworkTransaction` contract, the amount argument is appended to a global state variable mapping called `pending_transactions`. This is a mapping from an account address to an integer representing the transaction amount. So, while committing the transaction means changing the balance mapping in the `Coin` contract (which does not happen until the commit phase), in this step of the commit request phase, we change the state of the `CrossNetworkTransaction` contract to reflect an imminent cross-network transaction by assigning `pending_transactions[account]` to `amount`.
  (d) Both networks write an entry to their undo log (in case the other network rejects) → undoing the preparation phase simply means clearing the global mapping entry of a `pending_transactions[account]`.In other words, `pending_transactions` doubles as the undo log - the actual use of which is described in commit phase.

(3) In the case that the account was found to have insufficient funds for the transaction, the `prepare()` function in the `CrossNetworkTransaction` contract returns an integer value of -1 to the js Mediator indicating that the transaction should be aborted. Otherwise, an integer value of 1 is returned to the js Mediator. The consequence of this return value is outlined in the next phase, the commit phase.

4.2.2 Commit Phase

Success:

1. Upon receiving the return value of 1 (ready) from the `prepare()` function of the `CrossNetworkTransaction` contract for both networks, the js Mediator confirms that the state change resulting from the function has been mined, and invokes the `commit(address account)` function of the `CrossNetworkTransaction` contract, passing in the proper account address as the argument.

2. In the body of the `commit()` function, the global mapping state variable `pending_transactions[account]` (defined in Commit Request Phase 2.c) is read and the integer value is used to call one of the publicly accessible functions, `deductBalance(address account, uint amount)`, or `addBalance(address account, uint amount)`, in the `Coin` smart contract (the state of which the account balances reside). This effectively commits the transaction.

3. The `commit()` function then calls the `clearTransactionLog(address account)` helper function in the same contract which clears `pending_transactions[account]` to zero. The function then returns "1" to the js Mediator indicating completion.

4. Upon receiving a return value from the `CrossNetworkTransaction` for both respective networks, it confirms that the state change resulting from the transaction has been mined. Finally, the Mediator releases the account lock for both accounts using the transaction id it generated before setting the locks in the commit request phase. This concludes the cross-network transaction.

Failure:

1. If the return value from the `prepare()` function of the `CrossNetworkTransaction` contract for one of the networks is -1 (abort) and the other is 1 (ready), then the network that responded ready needs to abort the cross-network transaction preparation; the network that responded "abort" did not change the state of its `CrossNetworkTransaction` contract in preparation for the cross-network transaction, so it does not require further actions. If both networks responded with "abort", then neither network requires any more actions, and the js Mediator can proceed to release the account lock for both accounts, canceling the cross-network transaction.

2. If necessary, the js Mediator invokes the `abort(address account)` function of the CrossNetworkTransaction contract on the network that previously responded with an "abort" flag. In this function, the `pending_transactions[account]` entry is cleared by setting it to 0.

3. The function then returns "1" to the js Mediator indicating completion.

4. Upon receiving a return value from the `CrossNetworkTransaction` contract's `abort()` function, the js Mediator confirms that the state change resulting from the function call has been mined, and it releases the account lock for both accounts using the transaction id it generated before setting the locks in the commit request phase. This concludes the cancellation of the cross-network transaction.

4.3 Asynchronous Cross-Network Transaction Implementation

In this section we describe, in detail, the implementation of our asynchronous cross-network transaction protocol. We organize the implementation-description of the asynchronous protocol according to the two numerical steps described in the design section 3.2.

(1)
   (a) The Mediator, given the cross-network transaction information including the sender account, the receiver account, and the transaction amount, invokes the `commitAsynDebit(address account, uint amount)` function on the sending-network's `CrossNetworkTransaction` contract using the sender account for the account argument.

   (b) In the body of the `commitAsyncDebit()` function we call the `getBalance(address account)` function of the Coin contract (the state of which the account balances reside). If the account balance returned is less than the amount required, the `commitAsyncDebit()` function returns -1 to the Mediator, indicating failure. If the account balance is sufficient, the function immediately proceeds to commit the balance reduction by calling the publicly accessible function `deductBalance(address account, uint amount)` in the Coin smart contract.

   (c) Then, the function logs the transaction in the undo log. The undo log is implemented as a global state variable mapping called `undo_log`. This is a mapping from an account address to an integer representing the transaction amount to be reversed.

   (d) Lastly, the function returns the 1 to the Mediator, indicating success.

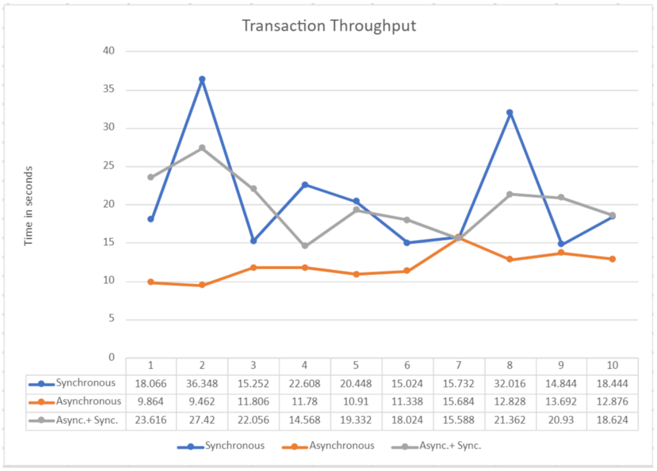
(2) If the Mediator receives a non-negative return value from the `commitAsyncDebit()` function, it proceeds to invoke the `commitAsyncCredit(address account, int amount)` function on the receiving-Network's `CrossNetworkTransaction` contract using the receiver account for the account argument. This function operates the same as `commitAsyncDebit()`, except that no verification of sufficient funds is required, and it commits a balance addition instead of a reduction.

As mentioned earlier, in real-world networks, there is a nonzero chance that one of the two blocks containing its half of the transaction will be orphaned while the other blockchain's block becomes canonical, thus making the transaction only partially applied. In our simplified environment, this will never actually happen. However, we can handle a hypothetical instance of atomicity failure by implementing a `reverseTransaction(address account)` function in the `CrossNetworkTransaction contract`. We can then invoke this function sporadically from the Mediator to simulate the occurrence of an atomicity failure. In the body of this function, the transaction amount to reverse is retrieved from `undo_log[account]`, described in 1.c. The sign of the amount is used to decide whether to call `addBalance()` or `deductBalance()` to reverse the transaction.

5 EVALUATION

We ran our experiments using two Ethereum private networks on a MacBook Pro 16 2019, with 2.6GHz 6-core Intel Core i7, Turbo Boost up to 4.5GHz, and a 12MB shared L3 cache, 512GB SSD, 16GB of 2666MHz DDR4 onboard memory, and AMD Radeon Pro 5300M with 4GB of GDDR6 memory and automatic graphics switching.

To evaluate our protocols, we compared the throughputs of transactions by executing multiple transactions several times using our synchronous, asynchronous, and a combination of asynchronous and synchronous(Mediator and Simulator) algorithms. We executed these transactions by using two Ethereum private networks for making cross-network transactions.



	1	2	3	4	5	6	7	8	9	10
Synchronous	18.066	36.348	15.252	22.608	20.448	15.024	15.732	32.016	14.844	18.444
Asynchronous	9.864	9.462	11.806	11.78	10.91	11.338	15.684	12.828	13.692	12.876
Async.+ Sync.	23.616	27.42	22.056	14.568	19.332	18.024	15.588	21.362	20.93	18.624

Fig. 2. Throughput Evaluation and Comparison

During our evaluation, we observed that asynchronous transactions exhibited the best performance throughput and took an average of 12 seconds to execute a single transaction. Whereas synchronously running transactions took an average of 21 seconds compared to the combination of asynchronous and synchronous algorithms, which took 20 seconds.

6 RELATED WORK

6.1 On Sharding Open Blockchains with Smart Contracts

Current blockchain systems contain several drawbacks in areas such as scalability, latency, and processing throughput. Sharding enables parallel confirmations of transactions and helps in mitigating these drawbacks. This mechanism requires miners to communicate frequently by using separate consensus protocols. In [Tao et al. 2020] researchers propose a new distributed and dynamic sharding system based on their key observation that users participating in a single smart contract can be validated independently and double spending can be prevented. Through their intra-shard transaction selection mechanism, they encourage miners to select different subsets of transactions for validation and use a parameter unification method. This aims to further improve the efficiency of these two algorithms, reduce communication costs and also improve system reliability. Their results showed an improvement in the throughput and a significant reduction of empty blocks.

Both our work and the work described in this paper incorporate smart contracts to facilitate the functionality of a sharded blockchain system. While our work focuses specifically on cross-shard transactions and abstracts away much of the detail of the surrounding sharding system, this work proposes a novel sharding system made to improve the throughput of blockchain systems based on smart contracts. A core motivation for their system is the recognition that cross-shard communication is expensive, and their solution to this problem is to design a system that minimizes the need for cross-shard communication altogether. Our solution can be perceived as better because it is generalizable to any typical sharded blockchain, and does not require the adoption of an entirely new, and highly specialized sharding paradigm.

6.2 Efficient Cross-Shard Transaction Execution in Sharded Blockchains

Sharding is a promising blockchain scaling solution yet it suffers from issues such as high latency and low throughput when it comes to cross-shard transactions. The root cause of these issues arises from the locking assets for extended time periods seen commonly in the two-phase commit protocol. In [Das et al. 2020], researchers introduce a new paradigm called Rivet, which maintains disjoint states of multiple worker shards, has a single reference shard running consensus and processes a subset of transactions in the system. The need for consensus within each worker shard is removed by Rivet and the shards can tolerate more failures which result in the communication overhead being reduced. Worker shards require fewer replicas and communication as they only vouch for the validity of the blocks. The reference chain, if substituted with a hierarchy of reference chains each coordinating with commitments and cross-shard transactions between a subset of worker shards would allow the system to process more transactions concurrently.

Both our work and the work presented in this paper focus on the problem of cross-shard transactions, and specifically address the limitations of the two-phase-commit protocol (and other blocking protocols). Their solution uses a "reference" shard, similar to our coordinator chain, which helps facilitate cross-shard transactions. The difference is that their reference shard takes over all consensus responsibility from the worker shards in order to allow for higher cross-shard transaction throughput. Our adaptive cross-shard transaction solution is made to work with a standard sharded system in which each shard runs consensus only on its own blocks - a quality that may be desirable depending on the application. Additionally, their work achieves higher cross-shard transaction throughput at the cost of increased intra-shard latency, while our solution does not have this drawback.

6.3 Cross-shard Transaction Processing in Sharding Blockchains

Past works like RSTBP and RSTSBP [Liu et al. 2020] have also proposed enhancements to the 2 Phase Commit protocol for speeding up cross-shard transactions. RSTBP modifies the promise phase of the 2PC protocol by processing a large number of transactions simultaneously by running a Byzantine Fault Tolerant algorithm. During the preparation phase, all shards generate an availability certificate to prove input availability. Input availability means that it has not been spent by any transaction and is not locked. To produce a certificate BFT algorithm must be invoked by a shard to reach an agreement and generate majority certificates. Locking an input saves from double-spending. During the commit phase, shards that receive all related certificates verify if the transactions are valid or not by checking the availability of all the inputs. RSTBT modifies the prepare phase in the 2PC and processes a batch of inputs from different transactions by input shards simultaneously using a BFT algorithm. During the commit phase, all the transactions are processed at once. This helps in reducing the number of BFT calls by at least a factor of the number of transactions in a transaction set(batch). RSTSBP scheme adds the Merkel tree construction step to the preparation phase. Merkel tree root contains every shard as a leaf node. The root of the Merkel tree is committed by the BFT algorithm during the availability certificate generation step and is generated for each related shard, containing a committed root, a leaf node, and a hash path. Merkle tree reduces the number of hash values by logm2 compared to logq2 in RSTBP where m denotes the number of shards and q represents the number of total inputs.

RSTBP and RSTSBP [Liu et al. 2020] aim at improving the efficiency of 2PC by reducing the number of cross-shard byzantine calls. The work in the paper focuses on bringing improvements at the core protocol level. Our work aims at using smart contracts for processing cross-shard transactions using remote procedure calls, and tends to abstract away the complexity of core protocol sharding and provides a customizable interface through smart contracts.

6.4 SharPer: Sharding Permissioned Blockchains Over Network Clusters

Works like SharPer [Amiri et al. 2021] have also introduced sharding in permissioned blockchains. A permissioned blockchain consists of a set of known nodes that might be untrusted, placed over data centers, public clouds, or local infrastructures. In SharPer, nodes are clustered and data shards are replicated across each node in the network. SharPer is Byzantine fault-tolerant and all nodes are either byzantine nodes or crash-only nodes. The blockchain ledger is formed in the form of a directed acyclic graph and each cluster maintains a view of the ledger and also incorporates decentralized flattened protocols to establish cross-shard consensus. It uses PBFT for intra-shard consensus. The decentralized nature of consensus enables the parallel processing of transactions with no overlapping clusters. It also provides a deterministic safety guarantee in networks where more than half of the nodes crash only or two third of the nodes are Byzantine.

SharPer aims at building a scalable blockchain system for permissioned nodes through the use of sharding, and proposes shifts to core protocol and consensus mechanisms for processing inter and intra shard transactions in presence of byzantine entities. Our approach intends to use smart contracts for processing intra shard transactions using remote procedure calls. Sharper is specialised for permissioned blockchains, making its use less feasible for permissionless blockchains like Ethereum or Bitcoin. We do not intend to tamper with the core protocol mechanisms and use abstractions provided by smart contracts making our approach more accessible and customizable.

7 CONCLUSION

We have developed a Node.js based Ethereum framework which operates atop two Ethereum private blockchain networks to facilitate and mediate adaptive cross-network transactions. Our framework is based on the principles of sharding and cross-shard transactions. In our evaluations using two Ethereum private networks, we have found that transactions executed using our asynchronous algorithm exhibited the best performance throughput. Evaluation of our framework is highly constrained due to limited hardware capabilities of our experimental set up. We had to rely on OS level stats such as CPU and Memory usage to decide to process transactions synchronously or asynchronously, since real-time blockchain level metrics were not accessible to us.

In future our framework can be extended to support better system interaction using a GUI. The coin implementation in our smart contracts provides means to implement cross network transactions. The proposed synchronous and asynchronous protocols for cross shard transactions are generic and can be easily extended to any Ethereum based real-word digital app(dApp) and digital coins on an active private network. The CrossNetworkTransaction smart contract provides an abstraction for Coin smart contract which is responsible for low level operations such as adding and updating balances and this will provide an easy way for application level customizations by future developers and escrow accounts could be used for transaction reversals.

References

Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. *SharPer: Sharding Permissioned Blockchains Over Network Clusters.* Association for Computing Machinery, New York, NY, USA, 76–88. https://doi.org/10.1145/3448016.3452807

Sourav Das, Vinith Krishnan, and Ling Ren. 2020. Efficient Cross-Shard Transaction Execution in Sharded Blockchains. *CoRR* abs/2007.14521 (2020). arXiv:2007.14521 https://arxiv.org/abs/2007.14521

Yizhong Liu, Jianwei Liu, Jiayuan Yin, Geng Li, Hui Yu, and Qianhong Wu. 2020. Cross-shard Transaction Processing in Sharding Blockchains. In *Algorithms and Architectures for Parallel Processing*, Meikang Qiu (Ed.). Springer International Publishing, Cham, 324–339.

Yuechen Tao, Bo Li, Jingjie Jiang, Hok Chu Ng, Cong Wang, and Baochun Li. 2020. On Sharding Open Blockchains with Smart Contracts. In *2020 IEEE 36th International Conference on Data Engineering (ICDE).* 1357–1368. https://doi.org/10.1109/ICDE48307.2020.00121

8 MEMBER CONTRIBUTIONS

Every team member worked equally towards this project, contributing 33.33% of the total work load each:

Ishaan's Contribution:

- Ethereum private network environment setup
- Main developer of synchronous protocol
- Aided in development of asynchronous protocol
- Main developer of Mediator and Simulator
- Smart contract / Solidity developer

Jack's Contribution:

- Ethereum private network environment setup
- System architecture design
- Main developer of asynchronous protocol
- Aided in development of Synchronous protocol
- Smart contract / Solidity developer

Neha's Contribution:

- Ethereum private network environment setup
- Evaluation and Testing
- Main author of report and presentation
- Aided in development of synchronous and asynchronous protocols
- Smart contract / Solidity developer