CroCRPC: Cross-Chain Remote Procedure Calls Framework for dApps

Avishek De
UC Santa Barbara
Santa Barbara, California, USA
avishekde.iitr@gmail.com

Divyakant Agrawal
UC Santa Barbara
Santa Barbara, California, USA
divyagrawal@ucsb.edu

Amr El Abbadi
UC Santa Barbara
Santa Barbara, California, USA
amr@cs.ucsb.edu

ABSTRACT

In this paper, we introduce **CroCRPC**, a **Cro**ss **C**hain **R**emote **P**rocedure **C**all (RPC) framework which can be used to interact between applications distributed across multiple blockchains. It provides a mechanism of communication between different blockchain networks, allowing the exchange of data, assets and transactions in a secure and efficient manner. It requires the deployment of CroCRPC smart contracts in each of the chains that the user wants to interact with. There are two interfaces offered by CroCRPC as bundles that are used to implement the participating contracts - the **Server Bundle** and the **Client Bundle**. The server contract, also referred to as callee should necessarily have an exhaustive list of the library functions that need remote calls support. Finally, the server also requires the implementation of a Node.js polling process (integrated into the CroCRPC server bundle) running for each individual application contract that handles the delivery of responses back to the source contract (the caller). The guarantee and integrity of message delivery across chains is handled by the LayerZero framework.

VLDB Workshop Reference Format:
Avishek De, Divyakant Agrawal, and Amr El Abbadi. CroCRPC: Cross-Chain Remote Procedure Calls Framework for dApps. VLDB 2024 Workshop: Sixth International Workshop on Foundations and Applications of Blockchain.

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/AvishekDe/CroCRPC.

1 INTRODUCTION

With the rise of decentralized applications and the increasing adoption of blockchain technology, the need for interoperability between different blockchain networks has become paramount. Frameworks like **LayerZero [16]** and **Axelar [17]** have already made huge strides in the area of cross-chain message transport frameworks, while **Polkadot [2]** has been instrumental in building an interoperability hub. In this paper we propose **CroCRPC: Cross-Chain Remote Procedure Calls Framework for dApps**, built one level on top of LayerZero, that will be particularly important for the development of advanced decentralized applications that require interactions between multiple blockchain networks. It allows a contract to act as a remote procedure host, whose APIs and endpoints can be invoked from Client contracts hosted on the same or different chains. Sample use cases can be decentralized exchanges (DeXs) allowing users to swap their tokens across chains without requiring an intermediary token. Similarly, lending and borrowing protocols can also use CroCRPC to transfer assets across chains.

In this paper we first review current research in this domain, and existing solutions that partially solves the problem of cross-chain communication. We also briefly discuss low-level message transport layer protocols that perform the heavy-lifting for sending payloads across chains, while ensuring features such as guaranteed delivery and trustlessness. Then we describe the CroCRPC framework. We introduce the notions of the *Client Bundle*, which users can inherit to invoke remote procedures located elsewhere, and *Server Bundle*, which needs to be inherited on the host applications allowing RPC functionality. This includes the *polling process* that is vital to sending back responses to the client. This is followed by a couple of real world example applications that use our framework to cater to users across chains. We end with a brief discussion about our evaluation metrics in terms of latency, gas cost and scalability, which are important factors that need to be understood before developing a cross chain application. The current implementation of CroCRPC is provided in **Solidity [4]** and it supports chains belonging to the Ethereum [7] universe. The server process is offered in **Node.js [1]** but can be easily extended to other languages as long as they support the *Ethereum JSON RPC API*.

2 BACKGROUND

The birth of Bitcoin led to an explosion of multiple Layer 1 blockchains with their own set of applications. While most of these distributed ledgers had their individual use cases, they operated in their own silos [20] with their exclusive set of applications, tokens, mining incentives and consensus mechanisms for adding new blocks into the network. Without meaningful interaction between applications across chains, such applications will have reduced usage since users want to stick to their blockchain of choice, additional overheads of converting one kind of token to another and implementation difficulties for cross chain message communication in terms of **reliability**, **trustlessness** and **guaranteed delivery**. A medium of communication between these chains was hence deemed extremely essential for the proliferation of blockchains. A lot of research has incrementally added trustless messaging layers between these chains, but their implementation is still rudimentary and requires massive programming efforts to shape it into applications that can coexist on multiple chains. A **remote procedure call framework for blockchain applications**, hence, in our opinion,

Proceedings of the VLDB Endowment. ISSN 2150-8097.

will bring significant benefits to end-users wanting to make their distributed applications available across multiple chains.

2.1 Remote Procedure Calls

Communication in distributed systems is characterized by transfer of information between programs residing across different hosts [6] (or smart contracts, in the case of blockchains). Traditionally remote procedure calls (RPCs) are expected to behave similar to local method calls via a suitable layer of abstraction [6]. The key difference is that the calls and information flow are made over the network and the failure in one autonomous process generally does not affect the other. An error in the callee procedure can be detected by the absence of a response to the caller, or by receiving an error code.

These isolation and failure detection capabilities are generally the prime benefits of having a system broken down into services and one of the reasons RPCs are preferred over a single monolith. In terms of blockchain, it is the only way to call procedures available in a host smart contract from an application residing on a different chain. Another important notion of remote calls is the unreliability of the message transfer medium. Since messages are transmitted over the internet, packet losses, message tampering and ordering inadequacies are quite common. For blockchains, there is the added complexity of ensuring trustlessness and guaranteed delivery (discussed in Section 3.1.2).

2.2 Related Work

This section builds an understanding on the current research being done in this domain and explains how CroCRPC tries to bridge the gap in cross chain RPC space.

Ethereum JSON RPC API [14] is developed by the Ethereum foundation. JSON is a lightweight data interchange format. It can represent all common data types in computer science. The Ethereum JSON RPC is a set of APIs designed to allow software developers interact with the Ethereum blockchain by reading blockchain data or sending transactions to the network. Via these libraries, developers can deploy new contracts, change the state of a contract, invoke procedures on a smart contract to execute some functionality or develop UIs to abstract the details of a contract. The Ethereum JSON RPC is the basic building block for developing and interacting with smart contracts in the Ethereum domain but it falls short of providing a way for smart contracts to be able to invoke procedures from a contract residing **on a different chain**.

Cross Framework [11] allows for the development of cross chain smart contracts that can access and invoke remote procedures from other chains. It is designed to support enterprise distributed ledger systems such as **Hyperledger [19]** and **Tendermint [12]** along with traditional Ethereum blockchains. It is built on top of the **IBC [8]** message transport layer for cross-chain communication. Cross provides framework support for multiple atomic commit protocols and a state store that supports committing, rollback, and concurrency control for contract states. CroCRPC provides similar functionality to the Cross framework but has a thinner client than Cross and is built on top of the LayerZero communication protocol, which in itself is thinner than IBC and relaxes the full on-chain header synchronization assumption required by IBC. This leads to lesser setup time and smaller generated contracts. IBC's hard requirement for deterministic finality prevents Cross from working on heterogeneous ledgers with **varied network topology** which can be achieved with CroCRPC based on LayerZero. We elaborate on this in Section 3. Currently, CroCRPC only provides clients in Solidity, whereas Cross also provides options in GoLang [15].

Polkadot [2] is a Layer 0 blockchain and an **interoperability hub** that provides the infrastructure on top of which users can build their own Layer 1 application-specific blockchains called **parachains**. It allows unrelated parachains built in the Polkadot framework to send not just tokens and assets but any kind of data and information between them. To achieve this, Polkadot uses a relay chain which is responsible for shared network security, consensus, and interoperability. The relay chain validates data and ensures that it is understandable across the board. It also has the concept of bridges to interact with external chains such as Bitcoin, but does not provide native RPC functionality for traditional Ethereum-like Layer 1 networks. CroCRPC is an attempt at implementing a plug-and-play interface, which does not need the user to perform any heavy lifting in terms of integration work or being able to speak a "custom language" to support cross chain applications.

3 CROSS CHAIN COMMUNICATION PRIMITIVES

The key motivation for sustained research on cross-chain communication protocols is interoperability, which refers to the ability of users and decentralized applications (dApps) to interact seamlessly across blockchains. Capabilities include the ability to read the state of contracts on external blockchains, send messages across blockchain boundaries or move assets across chains. Blockchain bridges connect two chains and allow users to send assets across chains but building bridges one by one for all combinations of Layer 1 chains is difficult. Also, redundant effort to build ad-hoc bridges might come at the cost of security risks [5]. That is where communication protocols such as LayerZero come into the picture and resolve the problem by providing frameworks that can be used to send cross chain messages without additional set up.

3.1 LayerZero

LayerZero [16] is a message transfer protocol that provides trustless valid delivery of messages across blockchains. In this paper, we have used LayerZero as the transport layer underneath CroCRPC. We describe in brief the LayerZero components, protocol and endpoints and how they form the building blocks behind our CroCRPC abstraction.

3.1.1 Components and Protocol. LayerZero **endpoint** is the main user facing interface. Every chain supported by LayerZero has the LayerZero components implemented as a sequence of on-chain smart contracts. The endpoint consists of four modules: the Communicator, Validator, Network, and Libraries. The first three comprises the core functionalities of LayerZero while every chain supported is implemented as an additional library. LayerZero also uses an **Oracle**, which is a third party service that reads a block header and transfers it to a different chain. In the default implementation of LayerZero, **Chainlink [13]** is used as the oracle component, which, however can be reconfigured. LayerZero also uses a **Relayer**, which

computes a Merkle proof that a transaction was indeed included in the block and forwards the proof of the transaction.

3.1.2 Message Transfer Features. For bridging, LayerZero needs to securely determine when a particular transaction is finalized on the initial chain for which it utilizes the Oracle and Relayer. It guarantees the following two features for every cross chain transaction.

Trustlessness: Users do not need to trust the components of LayerZero. Instead, it requires a weaker condition of independence between the Oracle and Relayer.

Valid Delivery: LayerZero guarantees valid delivery as long as there is no collusion between the Oracle and the Relayer, since it is statistically impossible to send a transaction proof that can be validated against a block header without knowledge of that specific block header, and vice versa.

3.2 Inter-Blockchain Communication Protocol

The **Inter-Blockchain Communication (IBC) Protocol [8]** is used to handle authentication and transport of data between two chains. Unlike most trusted bridging technologies, IBC provides a permissionless way for relaying data packets between blockchains. IBC implementations exist for blockchain networks such as **Hyperledger Fabric** and **Corda [10]** and is used as the transport layer for the Cross framework referenced in Section 2.2. Unlike LayerZero, IBC uses no third-parties and hence does not have any additional trust considerations. IBC consists of the **transport layer** which is responsible for transporting, authenticating and ordering data packets and the **application layer** which builds on top of the transport layer and specifies how data packets need to be interpreted. Using LayerZero instead of traditional IBC for CroCRPC has certain advantages. IBC's Transport Layer governs how light clients store and verify data, perform connection handshakes, and establish message channels. It is a full light client implementation that requires explicit on-chain full header synchronization for the handshake to succeed. In contrast, LayerZero relaxes this assumption by streaming block headers from the Oracle on-demand which is an efficient way of achieving full header synchronization state through an off-chain entity. Another disadvantage of IBC's transport layer is that it can only communicate between blockchains having deterministic finality. For IBC to work with proof of work systems, a finality threshold is required. LayerZero natively resolves the above problem as it allows the Oracle to act as the agent that enforces the necessary finality threshold.

3.3 Axelar Network

The **Axelar [17]** stack provides a decentralized network called the **Axelar chain**, protocols, tools and APIs that allow for cross chain communication. It provides a **Cross Chain Gateway protocol** that allows the connection of autonomous blockchain ecosystems to Axelar and a **Cross Chain Transfer Protocol** which application developers can use to connect their contracts to the network and perform cross-chain method invocations. **Validators** monitor Axelar's smart contracts deployed on connected chains, approve requests coming through those contracts on chain A and pass them to chain B to be executed. In contrast, LayerZero relies on two independent entities called the Relayer and Oracle to bypass the requirement of the **middle chain**. The addition of this middle chain means that while Axelar has a tighter control over its ecosystem, it **requires a lot of resources** as compared to LayerZero, including validators requiring to run nodes that connect to both Axelar and the external chains. LayerZero on the other hand is extremely **lightweight** and **cost-effective** and incorporating new chains into the network is faster. The major advantage of LayerZero over Axelar is the initial deployment speed. Removing the intermediate consensus layer to allow for direct communication between chains comes at a minor security risk of introducing third parties in the verification framework, which is minimized by the weak requirement of ensuring zero collusion between the Oracle and the Relayer. This means that an assumption is made about the Oracle and the Relayer being independent, honest actors. LayerZero, by design, does not guarantee that these components are independent but it is left to the user to ensure that they are.

4 CROCRPC

CroCRPC is a high level interface built on top of the LayerZero protocol to allow invoking remote procedure calls across chains via the network. The response is received asynchronously similar to a callback. This allows the caller to invoke multiple RPCs without the need of waiting for the results to come back. Given the considerable time taken to mint transactions and add them to the blockchain, a roundtrip journey of parameters and responses sometimes take a few minutes to complete. Hence, it would not be viable to implement a synchronous system as the program would waste a considerable part of its runtime waiting for results. CroCRPC uses message passing via LayerZero endpoints to simulate the RPC invocation as compared to HTTP requests and responses.

4.1 Overview

CroCRPC implements a **Server bundle** and a **Client Bundle** mimicking the interaction between clients and servers in an actual RPC domain [6]. The client bundle emits a message which encapsulates information about the remote method to be invoked along with its required arguments. This is similar to the *Request* phase of an HTTP call. This message then reaches the LayerZero endpoint on the source chain before getting to the corresponding endpoint on the destination chain. On the destination chain, the endpoint forwards the message to the smart contract of interest which handles it. When the response needs to be sent back to the client, a special polling process provided as part of the server bundle crafts another message encapsulating the response to be sent back to the client. It then uses the LayerZero sender method implementation on the server contract to direct the message back to the client. The route is exact opposite of the initial route used for the request. Fig. 1 shows the overall path of messages during the RPC invocation. The *purple arrows* represent messages that simulate the work of a **client request**, while the *green arrows* represent the **server responses**.

As an example, lets consider a user wants to invoke `Client Utility 1` which is used to call the `Server RPC 1`. The utility function calls the `Seriality Encoder` inside the Client Bundle to encapsulate the messages and then uses the `LayerZero Sender` to transmit a message. This goes to the on-chain endpoint which in turn sends a cross-chain-message to the endpoint on chain B. The message is then transferred to the `LayerZero Receiver` inside the

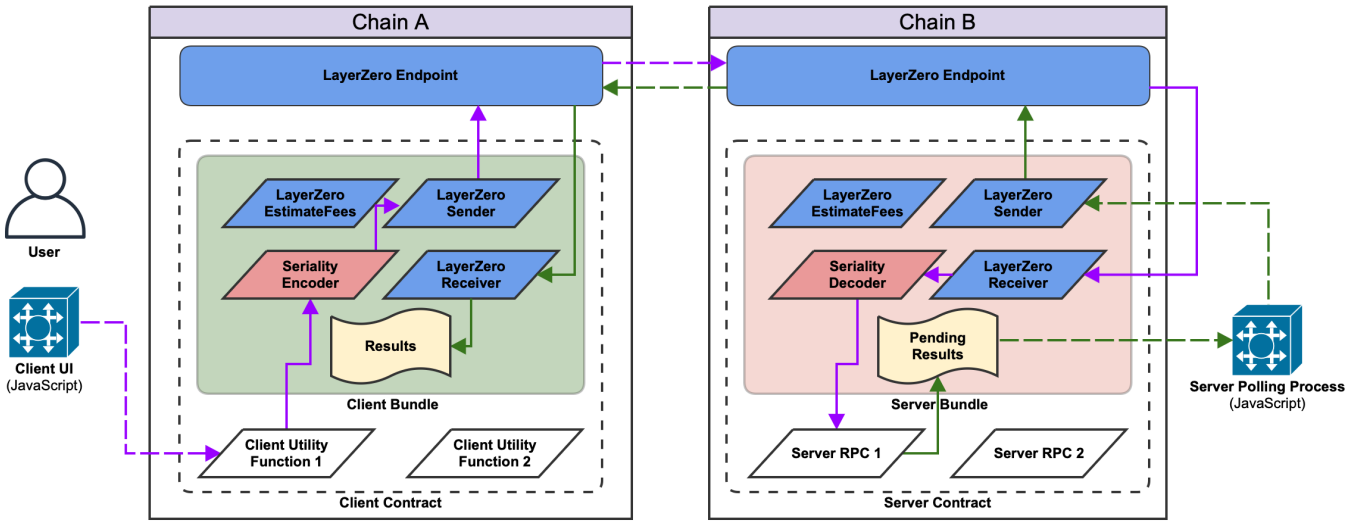

Figure 1: Message Paths during a typical RPC invocation using CroCRPC

`server bundle` which decodes the message using the `Seriality Decoder`. Then it calls the `Server RPC 1` and saves the results to the `Pending Results` pool. A server polling process linked to the server contract picks up this result via polling and uses a similar pathway to respond back to the client contract on Chain A. The Client UI will have access to this response via the `Results` data member.

Owing to the limitations of Solidity, CroCRPC does need to know the exhaustive list of remote functions available for users to invoke, and have their signatures preloaded into the server contract using the CroCRPC server bundle. This leads to certain application level limitations which are explained in Section 5.3. Every on-chain application that wants to host procedures that are accessible to other applications across chains need to implement the receiver bundle as part of their contract. Similarly, any application that wants to invoke remote procedures in a different chain needs to have the client bundle deployed. We explain these two bundles in more detail below.

4.2 Client Bundle

The client bundle is shown in green in Fig 1, and is inherited inside the client Smart Contract that has the necessary dependencies and additional functionality to keep track of the RPC results received from different chains. It abstracts the LayerZero functionality including the `send` and `receive` operations and also includes state variables to store and process results. The Client Bundle also includes the **Seriality Encoder** libraries method to serialize the function name and arguments that can then be sent as a binary message across the network. An *offset* variable inside the encoder is preset to 200 and reflects the length of the longest message that can be sent to the destination. Any message longer than offset will be truncated and the client might receive incorrect results or errors. Another auxiliary method called `estimateFees` is provided by LayerZero and can be used by the client UI to estimate the amount of gas that needs to be sent along with the message in order to be accepted by the blockchain network. Utility functions written in the client contract can ideally treat the client bundle as a black box and write application specific code that will run seamlessly given proper method arguments.

4.3 Server Bundle

The server bundle is shown in light red in Fig 1 and is inherited inside the server Smart Contract that has the necessary dependencies to host procedures that can be invoked from other chains. Similar to the client bundle it abstracts the LayerZero `send` and `receive` functionality. It has the **Seriality Decoder** to make sense of the bytes received from the client. After parsing the payload, the corresponding RPC is called which may be present inside the Server contract or sometimes in a different library. The result is stored in the `pending results` pool to be picked up for re-transmission across the network. Applications hosting functions that can be used by cross chain contracts need to inherit the `server bundle` and treat it like a black box from where function call requests are received and in turn, send back the results for such calls. Once the result is written into `pending results`, the server bundle waits on the server polling process to send the responses back to the client.

4.4 Server Polling Process

This is an additional process that runs on the server and interacts with the **server contract** mainly for sending back responses to the client. Additionally, it can call the APIs in the server contract, make state changes, perform actions which require cryptographic primitives and any other functions that require algorithmic privileges. In our present implementation, the server process waits for 7 seconds before polling again. The reason why we designed a separate polling process to send results back to the client is twofold.

First, it is easier to send the appropriate gas amount required to send a message from one chain to another from outside the contract. We can use custom wallets on each of these chains to indicate the source from which we want to spend the gas for sending

back the response. Calculating and implementing it from within the contract is cumbersome and includes complex logic such as **airdropping coins from the client contract**. Pushing this outside to the application code also allows a certain level of flexibility in customizing the amount of gas that is sent with the message. If there is a high priority message that needs to be transmitted quickly, we can increase the amount of gas to make it mint faster. Second, it allows us to parallelize consumption of messages from the `pending results` pool. Compare this with multiple consumers trying to read from a single queue to dispatch results more quickly. It also prevents the LayerZero methods inside the server bundle from getting blocked while the results are being relayed back to the client, while increasing the robustness of the overall architecture.

5 APPLICATIONS

In this section, we describe two blockchain prototype applications that demonstrate the utility of the CroCRPC framework. These apps serve to mimic their non-blockchain centralized counterparts with the added benefit of being implemented cross-chain in a distributed fashion. Since all contracts are public, this reduces the requirements of trust on and control of individual entities in the entire application. With LayerZero's architecture involving an Oracle and a Relayer, it is also non-trivial to send tampered evidence of transactions via cross-chain messages, thereby guaranteeing trustlessness.

5.1 Distributed Banking Application

This sample application simulates a cross-chain banking system where the bank is located on a chain that might be different from its customers. Instead of requiring all customers to be on the same blockchain, the bank can implement the CroCRPC framework which allows the clients to interact with the bank via remote procedure calls. In this example, the bank implements the server bundle to expose the APIs that the customers need for transactions. Customers implement the client bundle and send every transaction as an encoded cross-chain LayerZero message that is received and parsed at the destination blockchain Bank smart contract. The banking server also runs a polling process that checks for potential API responses and sends it back to the customer contract. This can be used to provided metadata back to the customer such as their current account balance, or simply an acknowledgement on whether the transaction succeeded or failed. Fig 2 shows the interface between the customer and the bank and the different components involved. Every additional customer will have implemented the client bundle which has not been shown in the figure for brevity. The *remote procedures* available to the customer UI are as follows:

```
int balance = makeDeposit(int amount)
int balance = sendMoney(address receiver, int amount)
```

The *int balance* return value is saved in the customer smart contract which can be observed asynchronously through the UI.

Communication Flow. We simulate here a transaction where **Customer A on the Goerli testnet** wants to send $100 to **Customer B on BSC testnet**. Both A and B are customers to Bank which is hosted on Fantom testnet. With the assumption that Customer A has sufficient balance to fulfil the transaction, we will have the following steps in the application.

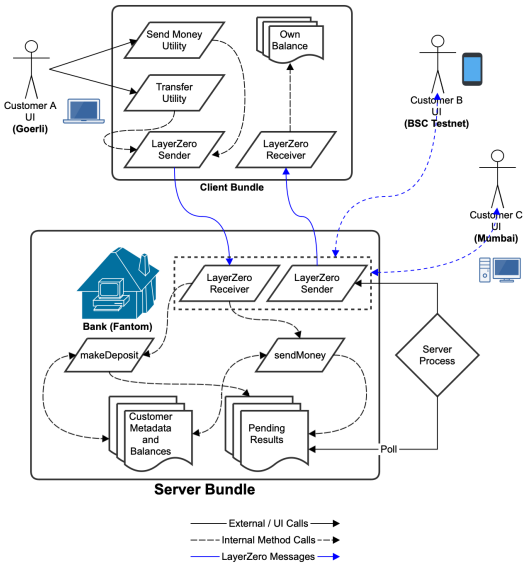

Figure 2: Distributed Banking Application Diagram

Step 1: A uses the `transfer utility` on his client UI.
Step 2: The `transfer utility` calls the corresponding API inside the customer smart contract that crafts and sends a LayerZero message `msg` to the bank smart contract on the Fantom testnet. This message contains information regarding the amount and the details of the receiver (smart contract address of B).
Step 3: The `LayerZero receiver` on the bank smart contract gets invoked with the receipt of message `msg`, parses it and calls the appropriate banking API to handle the request. In this case, it will call the `sendMoney` api with the amount and receiver information.
Step 4: `sendMoney` performs the transaction and updates the customer balances and other metadata. It also adds the balance information into the `pending results` pool.
Step 5: The `server process` polls repeatedly for new entries in `pending results`. Once it finds new entries, it creates a LayerZero message encoding the customer balances and metadata regarding the location of the affected customer smart contracts. It then uses the LayerZero sender API to transmit the messages back to the customers A and B.
Step 6: The `LayerZero receiver` on A and B's smart contracts reads the incoming message and updates their local balances.

5.2 Decentralized Voting Application

Voting is a fundamental democratic activity. While people think that paper ballots are generally the way to ensure everyone gets to vote, it is cumbersome and inefficient. Online election methods are extremely risky since any minor flaw in the application can lead to massive vote rigging. That is where blockchain technology comes into the picture to address these problems [18]. Since smart contracts are always public and verifiable, the contents of a contract and hence the results cannot be manipulated. Since it is distributed on multiple hosts, it is also less prone to faults such as a single server going down. It also ensures the anonymity of the voters, while ensuring that each voter gets a proof that their vote has been

recorded. In this second example application, we have tried to address these problems with a decentralized voting application based on the CroCRPC framework. Fig 3 shows the interface between the voter and the `VoteTopic` and the different components involved. Every additional voter will have implemented the client bundle which has not been shown in the figure for brevity.

Similar to the previous application, the `VoteTopic` is implemented as a smart contract which implements the CroCRPC framework's **server bundle**. It also houses the server polling process to check for responses that need to be sent back to the voters and sends them using the LayerZero framework. Every voter, irrespective of the blockchain deploys a smart contract that implements the client bundle. A Javascript UI can be used to interface with the utility functions available to the voter. The *remote procedures* available to the client (i.e, voter) for this application are the following:

```
uint transactionID = castVote(address topic, uint16
                     chainID, uint16 index)
```

Here, the voters provide the address of the contract (VoteTopic) that they want to vote for, the `chainID` where it is deployed and the index for the option they want to vote for. In turn, the remote procedure returns a `transactionID` that can be used to uniquely identify the vote and can be used later for tracking purposes.

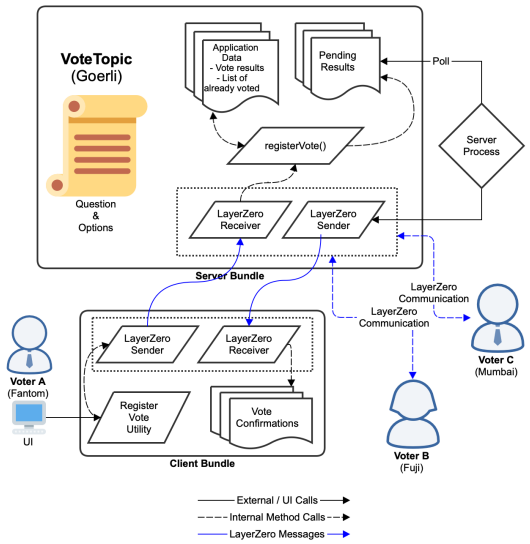

Figure 3: Decentralized Voting Application Diagram

Communication Flow. We simulate here a transaction where **Voter A on the Fantom testnet** wants to vote for **option C** in a `VoteTopic` having a question and four options (A to D) deployed on Goerli testnet. We will have the following steps in the application.

Step 1: Voter A uses the `castVote utility` on his client UI with `index=2`

Step 2: The `castVote utility` calls the corresponding API inside the customer smart contract that crafts and sends a LayerZero message `msg` to the VoteTopic smart contract on the Goerli testnet. This message contains information regarding the index/option that the client wants to vote for.

Step 3: The `LayerZero receiver` on the VoteTopic smart contract gets invoked with the receipt of message `msg`, parses it and calls the appropriate API to register the vote. In this case, it will call the `registerVote` API with the vote option and voter information.

Step 4: The API first checks if the voter is casting a vote for the first time, else it will ignore the vote as a duplicate. Once verified that this is a new voter, it will record the vote and generate a transaction ID which is a *keccak256* [9] hash of the voter address.

Step 5: `registerVote` also updates the `pending results` pool with information about the voter and the transactionID that needs to be sent back.

Step 6: The `server process` polls repeatedly for new entries in `pending results`. Once it finds new entries, it creates a LayerZero message encoding the transactionID for the recorded vote. It then uses the LayerZero sender API to transmit the messages back to the voter A.

Step 7: The `LayerZero receiver` on A's smart contract reads the incoming message and updates their local mapping saving the transactionID and the address of the vote topic.

5.3 Complexities and Potential Improvements

The sample applications described above demonstrates a communication flow that is very simple and works to illustrate the CroCRPC interface. This, therefore overlooks certain complexities which would need to be implemented for a fully functional application. For example, we are assuming that the features provided by the bank to its customers are known from the beginning and **does not change**. If the bank decides to implement new features in the future, we would require to create a new library smart contract. A good design would be to link this smart contract via its address, which is stored in the main banking contract as a *modifiable data member*. This design prevents recreating a contract every time we make a change to the underlying API libraries. If this implementation is not feasible, we would require every customer contract to update the address of the bank contract whenever the bank deploys a newer contract. We could have an `updateBankAddress` utility in the customer UI or implement a versioning mechanism on the client side to handle use cases where a certain API is not compatible with newer deployments. Also, this design would work well for a small customer pool. However, as the number of customers increase, we might need to also increase the number of `server process` clones that re-transmits results back to the customers. That brings with it its own complications related to the access of **critical code blocks** and **synchronized sections** for code paths racing to update the same data member. Owing to Solidity limitations, this in itself requires additional third-party libraries to properly implement and manage. Similar arguments can be made for the Voting application.

5.4 Security Assumptions

The applications mentioned here in the paper are fairly simple and are presented just to demonstrate various use cases of cross chain remote procedure calls. Security implications have been ignored. For example, in the banking application, we have not demonstrated how to verify that it is the actual owner of the account who is making a `sendTransaction` and not someone else. Similarly, in the Decentralized Voting application, our proposed implementation

would not be able to identify if the actual user is casting the vote or someone else. One way to overcome these security implications is to have some sort of ***public key-private key cryptography*** [3] built into these smart contracts. This way the messages that are received from the client UI implementations can be verified to identify the actual owner. However, such cryptography primitives are not predominantly widespread in blockchain applications and is outside the current scope of the paper.

6 EVALUATION

In this section, we evaluate the performance of the CroCRPC framework in terms of gas cost, latency and scalability. While these metrics can vary widely and are not the most important design decisions while choosing a network to act as the server and client, the data below gives us a general idea of the ballpark ranges and estimates. For the contracts to send cross chain messages, we need to fund our source wallets with gas tokens, to be given as incentive to the LayerZero Oracle and Relayer. The latency shows the response time frame and scalability discusses the steps that can be taken to make the servers able to handle increasing load.

6.1 Gas Fees

Remote procedure calls require two different gas fees to complete a round-trip transaction. The first is needed when the remote method name and arguments are encoded and sent to the server contract and is paid by the client wallet. The second is required when the server polling process encodes and sends back the response to the client contract. This is paid by the server wallet. LayerZero provides an implementation of the `estimateFees` function which returns a dynamic fee based on the Oracle and Relayer prices for the **destination Chain ID**. These values are dependent on the payload, the originating smart contract and is provided in the ***native gas token units*** for the corresponding chain. Fig 4 shows the variation in gas fees in the native token for the same payload across different destination chains, for some of the source testnet chains. Note that these chains are chosen according to the availability of LayerZero endpoints for those testnets. The graph shows gas cost for every destination chain shown in the legend when a message is sent from the source chains shown along the x-axis.

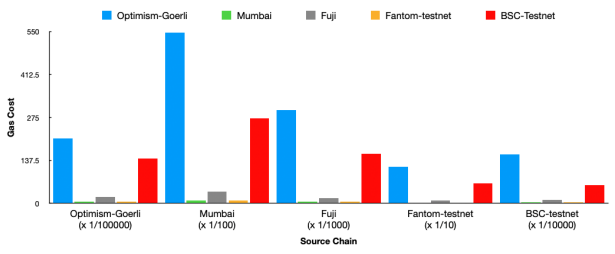

Figure 4: Typical Gas Fees for sending a cross chain message

From the figure, we can observe that it takes the **highest** amount of gas fees to send a message to the **Optimism-Goerli** chain from any of the other chains. **BSC-Testnet** comes second and is still significantly higher than the other testnests under consideration. **Mumbai** and **Fantom** are the **cheapest** when it comes to charging gas fees with Fuji lying dead centre of our sample set. These considerations might prove essential in deciding potential RPC hosts when the same features are available on multiple chains as users would typically like to reduce the amount of gas fees they pay for invoking remote procedures. The gas fees from different source chains have been scaled for the graph to be more meaningful.

Latency Metrics (seconds)		Server Contract		
		Mumbai	Fuji	Fantom-testnet
Client Contract	Mumbai	-	94	100
	Fuji	129	-	114
	Fantom-testnet	117	77	-

Table 1: Latency metrics for a complete remote procedure call

6.2 Response Latency

Remote procedure calls are naturally subject to network latency because messages are transmitted over the internet. The responses received from the server contract can be treated like a callback function and the main client process should not wait for the result. In Table 1, we have noted the latency metrics in terms of seconds that pass between the client contract invoking a remote procedure until it receives a response back from the server. We show the results for three of the chains which have the lowest gas cost among the ones tested. We observe that the delay is somewhere between a little over a minute to over two minutes. Latency can be considerably higher for mainnet chains and during periods of high transaction volume when nodes are busy processing many transactions. We can reduce the latency by providing more gas tokens as fees but there are other factors which determine the total delay. Some of these include network congestion, network propagation delay, smart contract execution time and ordering of message delivery.

6.3 Scalability

For server applications handling a large number of remote procedure calls per second, the number of clients invoking the RPCs may exceed the rate at which the server process is capable of sending back responses. In such circumstances, it is fairly easy to modify the server bundle to increase the number of polling processes handling the response transmission. More polling processes naturally have a higher transmission rate. If the order in which responses are sent back to the respective clients does not matter, the **non-blocking LayerZero interface** can be implemented to not wait for a successful message transmission confirmation before sending out the next. In such a system, however, **re-entrancy guards** should be added in the server contract for methods that should only be called serially. This prevents destructive state changes and ensures the integrity of contract state variables.

7 CONCLUSION

This paper introduces the design and implementation of CroCRPC, a cross chain RPC framework that allows developers to transfer

information and assets and invoke procedures across chains without an intermediary. The framework provides server and client bundles which make it easier for users to plug and play without requiring extensive integration to interoperability hubs or converting code to suit a specific protocol. The underlying LayerZero transport layer allows native transactions between supported chains, while newer chains can be added by just implementing additional libraries. There is no initial or ongoing setup cost required to convert a server application to act as an RPC host.

We believe CroCRPC will provide the underlying fiber for a variety of cross-chain decentralized applications in the near future allowing unhindered movement of information and assets in a safe and secure manner across chains. Indie developers will easily be able to port their applications from a centralized server based model to blockchains and reap the benefits of decentralization, distributed fault-tolerant robust systems, trustlessness and transparency.

REFERENCES

[1] [n.d.]. Node.js Docs. https://nodejs.org/en/about. [Online; accessed 10-May-2023].

[2] [n.d.]. Polkadot. https://polkadot.network. [Online; accessed 03-May-2023].

[3] [n.d.]. Public-key Cryptography. https://en.wikipedia.org/wiki/Public-key_cryptography. [Online; accessed 5-Aug-2024].

[4] [n.d.]. Solidity Docs. https://docs.soliditylang.org/en/v0.8.17/index.html. [Online; accessed 10-May-2023].

[5] 2022. Axelar Network, LayerZero and the future of interoperability. https://orangutans.substack.com/p/axelar-network-layerzero-and-the. [Online; accessed 03-May-2023].

[6] Andrew D. Birrell and Bruce Jay Nelson. 1981. "Remote Procedure Call". *Report No. CSL 81-9, Xerox Palo Alto Research Center* (1981).

[7] Vitalik Buterin. 2014. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.* Technical Report. Ethereum Foundation.

[8] Christopher Goes. [n.d.]. *"The Interblockchain Communication Protocol: An Overview".* Technical Report. Interchain GmbH.

[9] Seth Hoffert Michaël Peeters Gilles Van Assche Guido Bertoni, Joan Daemen and Ronny Van Keer. [n.d.]. Keccak specifications summary. https://keccak.team/keccak_specs_summary.html. [Online; accessed 5-Aug-2024].

[10] Mike Hearn. 2019. *Corda: A Distributed Ledger.* Technical Report.

[11] Ryo Sato Jun Kimura. [n.d.]. Cross Framework. https://github.com/datachainlab/cross. [Online; accessed 03-May-2023].

[12] Jae Kwon. 2014. *Tendermint: Consensus without Mining.* Technical Report.

[13] Benedict Chan Alex Coventry Steve Ellis Ari Juels Farinaz Koushanfar Andrew Miller Brendan Magauran Daniel Moroz Sergey Nazarov Alexandru Topliceanu Florian Tram‘er Fan Zhang Lorenz Breidenbach, Christian Cachin. 2021. *"Chainlink 2.0: Next steps in the evolution of decentralized oracle networks".* Technical Report.

[14] Hiroyuki Naito Timur Badretdinov Andrei Kostakov RimTaeX Léo Vincent Pablo Pettinari Nico Corwin Smith F. Eugene Aumson Ardis Lu Vid Kersic 0xtekgrinder David Murdoch Tom Kaan Uzdoğan Kirill Makarov Joseph Cook Sahil Aujla Alex John C. Vernaleo Neeraj Gahlot Joshua Sam Richards Paul Wackerow, Amit Kumar Mishra. [n.d.]. Ethereum JSON RPC. https://ethereum.org/en/developers/docs/apis/json-rpc/. [Online; accessed 03-May-2023].

[15] Rob Pike. [n.d.]. *Go at Google: Language Design in the Service of Software Engineering.* Technical Report. Google Inc.

[16] Caleb Banister Ryan Zarick, Bryan Pellegrino. 2021. *LayerZero: Trustless Omnichain Interoperability Protocol.* Technical Report. LayerZero Labs.

[17] Georgios Vlachos Sergey Gorbunov. 2021. *Axelar Network: Connecting Applications with Blockchain Ecosystems.* Technical Report. Axelar Foundation.

[18] Linta Islam Syada Tasmia Alvi, Mohammed Nasir Uddin and Sajib Ahamed. 2022. DVTChain: A blockchain-based decentralized mechanism to ensure the security of digital voting system voting system. *Journal of King Saud University - Computer and Information Sciences* 34, 9 (2022), 6855–6871. https://doi.org/10.1016/j.jksuci.2022.06.014

[19] Mic Bowman Christian Cachin Nick Gaski Nathan George Gordon Graham Daniel Hardman Ram Jagadeesan Travin Keith Renat Khasanshyn Murali Krishna Tracy Kuhrt Arnaud Le Hors Jonathan Levi Stanislav Liberman Esther Mendez Dan Middleton Hart Montgomery Dan O'Prey Drummond Reed Stefan Teis Dave Voell Greg Wallace Baohua Yang Tamas Blummer, Sean Bohan. 2021. *An Introduction to Hyperledger.* Technical Report. Hyperledger Foundation.

[20] Bryan Phern Chern Teoh and Bak Aun Teoh. 2022. Blockchain Interoperability: Connecting Supply Chains Towards Mass Adoption. In *Design in Maritime Engineering*, Azman Ismail, Wardiah Mohd Dahalan, and Andreas Öchsner (Eds.). Springer International Publishing, Cham, 299–309.