.

Router Chain

Draft v1.0

February 2023

Abstract

Over the past few years, we have witnessed Web 3.0 evolve from a novel innovation to an ethos defining the new era of computation and digital governance. The maturity of Ethereum's infrastructure and ecosystem fostered true innovation, which has paved the way for a variety of decentralized applications. However, increased adoption by users led to scalability issues, resulting in the emergence of multiple layer 1 and layer 2 networks. Blockchain trilemma design trade-offs in most of these solutions have been focused on reduced gas costs and improved throughput. While these new networks have played an essential role in onboarding new users to the DeFi ecosystem, their proliferation has resulted in the fragmentation of liquidity, user base, and activity across multiple networks. Due to these factors, interoperability is no longer a luxury; it is a requirement for any dApp seeking to capitalize on the subsequent adoption wave. Attempts have been made in the past to solve this problem by enabling communication across chains, but their usability is limited by a number of issues. Towards this end, this paper introduces the Router chain, an interoperability hub built at a crucial intersection of decentralization, scalability, and security. With an architecture enabling communication between various blockchain ecosystems - EVM & non-EVM, support for middleware contracts and application-specific bridging logic, and a developer tooling suite facilitating continuous integration and development of cross-chain dApps, the Router chain will catalyze the growth of the cross-chain ecosystem.

Contents

Glossary

atomicity If an operation is atomic, it will either be seen as yet to be started or as completed and not in any partially completed state. In other words, either all the sub-operations in that operation will be successful, or none of them will be successful.

block headers The part of the block which includes all the metadata, including block height, block hash, the Merkle root of all the transactions included in the block, and the timestamp at which the block was mined, amongst other things. They act as a summary of the block.

blockchain trilemma The idea that finding the right balance between security, decentralization, and scalability is very hard in a blockchain.

composability The ability to use existing components and resources as building blocks for new applications.

interoperability The ability of a system to connect and work with other systems in a coordinated fashion while imposing no limits on the end user. In the field of blockchain, interoperability refers to the ability of disparate blockchain systems to interact with each other.

layer 1 Base layer blockchain architecture, e.g., Bitcoin, Ethereum. In recent years, multiple layer 1 alternatives to Ethereum have come to the fore. Most of these blockchains have made certain changes to Ethereum's existing infrastructure, like introducing sharding or implementing a different consensus mechanism.

layer 2 Blockchain networks implemented as smart contracts on top of another blockchain (layer 1). Layer 2 networks do not make any changes to the underlying blockchain architecture. For example, layer 2 networks on Ethereum take advantage of Ethereum's decentralized security model while negating its scalability constraints by adding another layer of transactions on top of it. There are mainly three ways in which layer 2 scaling solutions are exercised - State Channels, Plasma, and Rollups. Popular examples of layer 2 blockchains include Polygon, Optimism, and Arbitrum.

Merkle root The hash of all the hashes of all the transactions included in a block. It is part of the block header.

multisig A special type of digital signature that requires two or more users to sign.

nonce A pseudo-random number that can only be used once. It is used by blockchains primarily as a counter.

oracles Data feeds that bring information from external sources and provide it to smart contracts on chain. Decentralized oracles play a significant role in any blockchain system since blockchains cannot access off-chain information on their own.

pseudo RNG Defined by the use of deterministic algorithms to generate random numbers.

siloed Refers to something that is isolated from other.

state proof A cryptographic proof delineating all the state changes that happened in a specific set of blocks.

stateful In the context of decentralized systems, a stateful entity is one that can maintain past data. Smart contracts are considered stateful because they can store data in the form of variables.

stateless In the context of decentralized systems, a stateless entity is one that does not maintain past data.

tendermint An open-source blockchain protocol that allows developers to build secure decentralized applications.

true RNG Defined by the use of external physical attributes such as radioactive decay, airwave static, and atmospheric noise, among others, to generate random numbers.

Web 3.0 Third generation of internet services built on top of decentralized data networks with the aim to make the internet more open and trustless.

Acronyms

BFT Byzantine Fault Tolerant.

DeFi Decentralized Finance.

ECDSA Elliptic Curve Digital Signature Algorithm.

EVM Ethereum Virtual Machine.

HTLCs Hash Time-Locked Contracts.

IBC Inter-Blockchain Communication Protocol.

MPC Multi Party Computation.

PoA Proof of Authority.

PoS Proof of Stake.

RNG Random Number Generator.

SBT Soulbound Token.

1 Introduction

1.1 Background

It has been well established that the cornerstone of innovation is rooted in iteration as much as it is in invention. In 2008, Bitcoin's whitepaper [1], which was later abstracted to build the first blockchain network, was an instance of an inimitable invention. It challenged existing archaic institutions from their very foundations and provided a viable alternative to the traditional norms of transaction, computation, and communication. The evolution and subsequent maturity of the whole Web 3.0 ecosystem has been a process of iterative innovations. It has involved isolating the caveats of existing technical architecture and building viable products to address them. Bitcoin's technical restraints to build and run dApps gave birth to the Ethereum network. In turn, Ethereum's scalability constraints gave way to multiple layer 1 and layer 2 blockchains, each trying to carve out a niche for themselves by capitalizing on a core competency to onboard new applications and attract developers, users, and liquidity providers. A few of these blockchains have been able to achieve this goal by committing strategic trade-offs to balance the blockchain trilemma. While their developments have played a vital role in onboarding new users onto Web 3.0, it has also amplified a foundational quandary of a blockchain network. The absence of interoperability.

Up until now, every blockchain ecosystem has operated and matured in silos. A trait that gave them gravitas and catalyzed their initial usage is now becoming the primary factor for limiting their developer ecosystem and, subsequently, their user base. The time and effort required to explore various blockchain ecosystems impede most users from tapping into the benefits of different blockchains. Blockchain interoperability will, therefore, play a pivotal role in the holistic evolution of the Web 3.0 ecosystem and will subsequently facilitate the next wave of Web 3.0 adoption. It is no longer a cosmetic add-on feature that serves as a means to impress the end users. It has become a must-have - without a robust mechanism that extends the composability of DeFi and promotes communication among various blockchains, the success of current and upcoming L1s/L2s is a non-starter.

The self-evident and almost natural solution to the problem of blockchain interoperability is cross-chain bridges. Over the past few months, there has been a sudden influx of cross-chain bridges with varying capabilities - some only allow for the transfer of tokens, whereas some also provide support for message passing between independent platforms. And even though these bridges have made significant strides towards achieving an interoperable ecosystem, they suffer from varying issues that hamper their usability. From doubts about their security and decentralization to their inability to simplify cross-chain integrations, existing cross-chain platforms have often come short of the expectations placed upon them.

1.2 A Brief History of Router V1

The Router team was among the first in a long line of cross-chain projects to comprehend the problem of blockchain interoperability and propose a solution for the same. In January of 2022, we launched V1 of Router Protocol - an extensible multi-directional bridge connecting existing and emerging layer 1 and layer 2 blockchains to allow contract-level data flow across them. This could be a token that is locked on the source chain and redeemed on the target chain or an operation that is initiated on the source chain and executed on the target chain. Two key applications built on top of Router V1 are Voyager, a permissionless cross-chain asset swap protocol, and CrossTalk, a modular cross-chain framework that leverages Router's infrastructure to enable seamless state transitions across multiple chains akin to IBC for Cosmos.

Even in its current form, Router Protocol, through its modular architecture, is superior to most arbitrary messaging bridges. Most notably in the aspects of security, speed, and convenience. However, in a true testimony to the spirit of innovation and continuous improvement, a core value of Team Router, we're upgrading the architecture to come up with an even more elegant solution to the problem of interoperability. Before we delve in deeper, it's worthwhile to first explore Router V1's validation setup and its challenges.

1.2.1 Router V1 Validation Mechanism

Router V1, a Proof of Authority (PoA) bridge, maintains a set of nodes/validators that listen for events on the source chain, generate proposals for those events, and submit signed proposals on the destination chain as a vote. For a transfer to be accepted and sent across the bridge, it must receive enough votes to exceed a predetermined threshold. All the bridge contracts maintain a safe list of addresses so that votes received only from approved router nodes are taken into consideration.

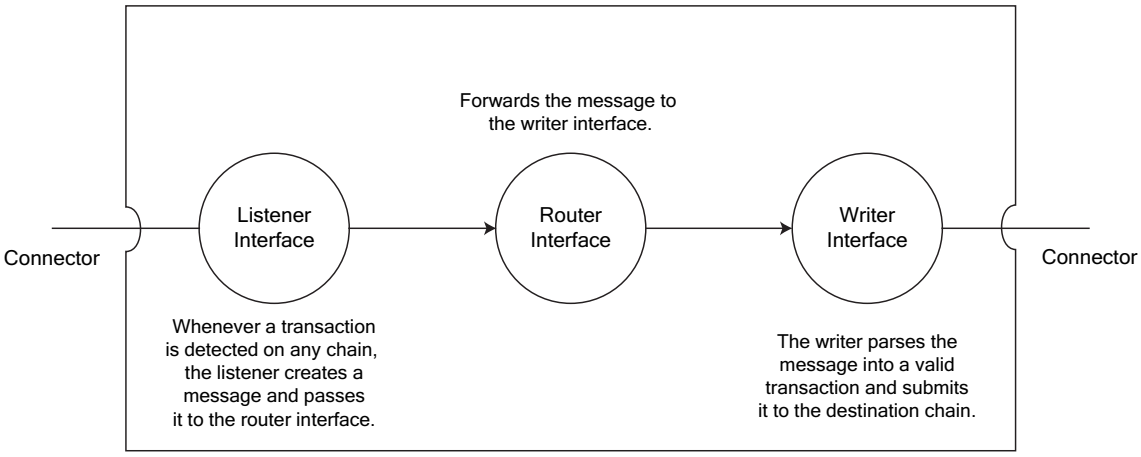

Figure 1: Architecture of a Router Node in Router V1

Each router node runs a Golang chain module that has four components:

- **Connector:** Responsible for connecting a router node to all chains. A connector is shared by a node's listener and writer.
- **Listener:** Actively observes chain state transitions to listen for initiated transfers. Whenever a transfer is detected, it constructs a message and passes it to the router interface. The message has a total of 6 parameters: source chain ID, destination chain ID, resource ID, transfer type (asset transfer or generic transfer), deposit nonce, and payload (data for the specific transfer).
- **Router:** Receives constructed messages from listeners and forwards them to the writer.
- **Writer:** Responsible for parsing the bridge message into a valid transaction and submitting it to the destination chain.

1.2.2 Router V1 Challenges

1. **Stateless nature of the validators:**
   (a) No way to maintain states during the communication between two chains, which shuts the door on a number of potential applications. E.g., Cross-chain governance, cross-chain oracles, etc.
   (b) Code redundancy - Same business logic has to be implemented on bridge contracts deployed on all the chains.
   (c) No support for application-specific bridging logic.
2. **Limited validator set:** Due to the cost involved in submitting votes on the destination chain, the current validation scheme can get prohibitively expensive as we increase the number of validators.

1.3 Intro to Router Chain

To address the shortcomings of Router V1 and strike the perfect balance between security, decentralization, and scalability/throughput - we introduce a new version of Router Protocol powered by

a tendermint-based Router chain. A blockchain focused primarily on enabling state transitions across chains, the Router chain will sit as a hub between various EVM and non-EVM ecosystems. Features that set Router chain apart from other interoperability solutions include, but are not limited to:

1. **Support for middleware contracts:** Maintain states and implement custom business logic directly in the bridging layer.
2. **Plug-and-play for developers:** Router will provide a transcendent open-source developer tooling suite to assist with the continuous integration and development of cross-chain dApps.
3. **CrossTalk:** Developers looking to build cross-chain applications without any custom bridging logic can leverage Router's easy-to-integrate smart contract library, CrossTalk.
4. **Support for various kinds of use cases:** Batching, sequencing, and atomicity can be enforced directly from the Router chain.
5. **Data aggregation:** Contracts on the Router chain can serve as data aggregation modules for various cross-chain and multi-chain applications.
6. **Cross-chain meta transactions:** By leveraging Router as their cross-chain infra provider, applications can enable gasless cross-chain transactions by delegating the execution of a request to a third-party service.
7. **Composability:** The Router chain will have inbuilt support for global applications such as oracles and liquidity pools/bridges, to name a few, which will help in easier integration of other applications.

1.4 Organization

In the sections that follow, we will first comprehend the shortcomings of the existing bridging technologies, followed by a deep dive into the Router chain's architecture, working, transfer flows, and security considerations. Following that, we'll explore Router's CrossTalk framework that abstracts Router chain's architecture into an easy-to-use smart contract library. Next, we will expand upon Router's fee model for various flows, followed by a look into the features afforded by Router. In drawing the paper to a close, we will briefly touch upon other facets that can be added to the Router chain in the future.

2 Existing Solutions

The problem of blockchain interoperability has become impossible to ignore as more siloed ecosystems emerge across the DeFi space. As mentioned above, to solve this issue, multiple bridging technologies have come to the fore in recent times. Based on the level of trust required, these solutions can be classified into three broad categories:

- **Trustless:** These systems do not require their users to place any trust in third-party actors.
- **Centralized Trust-based:** The system's control rests with a few external actors, and users must assume that they are not malicious.
- **Decentralized Trust-based:** The system is governed by an extensive network of third-party actors; users have to trust that the majority of them are not malicious.

Examples of trustless bridging technologies include Optimistic bridges, and light-client-based bridges, whereas PoA bridges come under the category of centralized trust-based bridges. Chain-based approaches fall into the third category, i.e., they follow a decentralized trust-based model. In this section, we will examine some of the widely deployed cross-chain technologies and their advantages/shortcomings.

2.1 Hash Time Locked Contracts (HTLCs)

One of the earliest models developed to enable cross-chain operations was Hash Time-Locked Contracts (HTLCs), which employs hash locks [2] and timelocks [3] to ensure that the operations remain atomic. Even though HTLC implementations differ across projects, the overall concept remains the same [4]. Let us understand the working of HTLCs using the following example:

Step 1: Alice hashes a secret code to obtain hash lock h_1. Alice also generates a timelock t_1 corresponding to an upper bound in which the hash lock can be unlocked.

Step 2: Alice uses these locks to create a smart contract c_a on chain A and locks her funds in that contract.

Step 3: Bob acknowledges that Alice has locked her funds.

Step 4: Bob uses the same hash lock h_1 and a different timelock t_2 to create a contract c_b on chain B. To ensure that Bob gets adequate time to claim funds from contract c_a, t_2 will be less than t_1.

Step 5: Alice unlocks Bob's funds from contract c_b thereby revealing the secret code.

Step 6: Bob uses the revealed secret to unlock Alice's funds from contract c_a on chain A.

Step 7: If the swap does not go through, Alice and Bob can claim their funds back once the timelock on the individual contracts expires.

One upside of using HTLC techniques is that they do not introduce any trust assumptions. However, they suffer from a range of issues that limits their efficacy:

- They require all the concerned parties to always be online. Both the sender and the receiver need to monitor the involved blockchains during execution actively.
- They are very slow and inefficient since every cross-chain swap requires a total of four transactions (two on each blockchain) [5].
- Given the high fees and waiting periods involved with HTLC-based swaps, the scalability of this approach is also a concern.

2.2 Proof of Authority (PoA) Bridges

Proof of Authority (PoA) bridges rely on a small set of outside actors that listen to events on the source chain, validate them, and relay them to the destination chain. Incentivization and slashing mechanisms are often kept in place to ensure the integrity of these actors. For the most part, PoA solutions work well -

- Since few validators are involved, the consensus can be achieved in very little time, ensuring a low-latency transfer of funds/messages.
- Adding support for new chains is straightforward - the validators can simply update their configuration to subscribe to events coming from the newly supported chain.

The only, albeit quite a significant, drawback of PoA bridges is that they are trust-based, not trustless. PoA bridges necessitate that their users place trust in a federation of third-party validators. Since the number of entities at play in a PoA system is relatively low compared to a Proof of Stake (PoS) system, there is a possibility of collusion; a dishonest majority of the authorities can manipulate the system to their advantage and to the detriment of the end user.

2.3 Light Client Node Approach

A light client can be defined as a smart contract that parses source chain block headers on the destination chain. Since a light client node maintains a record of historical block headers, it can verify that a particular event has indeed taken place on the source blockchain. In a light-client-based bridge, external actors named relayers forward events from the source chain to the destination chain, including block headers, state proofs, and other relevant data. Following this, the source chain's light client node on the destination chain cross-references its records to verify that a particular event was recorded on the source chain before executing a corresponding action on the destination chain. When compared to PoA bridges (Section 2.2), light-client-based bridges have two main advantages:

- There is no need to maintain a new validation layer [6].
- There are no trust assumptions - even though a third-party actor forwards an event from the source chain to the destination chain, light clients can independently validate the event's existence using the block headers it holds.

However, light-client-based approaches also suffer from a few issues:

1. **Light client nodes can be incredibly expensive to operate:** Due to the costs associated with executing gas-intensive validation logic, updating block headers is a costly affair, especially for light clients running on Ethereum [7]. To combat this, some bridges batch these block headers before relaying them to Ethereum, which can be problematic for time-sensitive applications. For example, Rainbow bridge, one of the most popular implementations of a light client node, batches Near block headers and sends them to Ethereum after a period of 12-16 hours. This can lead to long waiting times when bridging assets from Near to Ethereum.
2. **Adding a new chain to the mix is resource-intensive:** For every new chain, (a) a new light client has to be deployed on all the existing chains, and (b) light clients of all the existing chains need to be deployed on the new chain [8].

2.4 Ultra Light Client Node with Oracle-based Bridge Adaptors

To alleviate the concerns surrounding the costs involved in running a light client node, a few projects opt to replace a light client node with an ultra-light client node. An ultra-light client interface is similar to a light client node in that it also validates whether a transaction has been committed on the source chain. However, unlike a light client node, an ultra-light client node does not keep track of block headers; it cannot compute the transaction proof on its own - an external actor is required to forward the transaction proof. To prevent any single party from tampering with the block headers and transaction proofs to include a malicious transaction, the entity relaying the block headers and the entity relaying the transaction proof must be different. A prominent implementation of this approach uses an oracle to relay the block headers and a relayer to forward the transaction proofs.

Although such a model addresses the cost constraints of a light-client-based model, it introduces a new trust assumption - the oracle and the relayer will not collude. Although this problem is unlikely to arise while using reliable decentralized oracles, finding such oracles is not an easy task. In addition to this, as with light-client-based bridges, including a new chain into the mix can be quite challenging with this approach.

2.5 Relays/Sidechains

Another prominent strategy to achieve interoperability is based on relays/sidechains. Relays are abstractions (often a smart contract or a script) deployed on some chain A with light-client-like verification capabilities over chain B [9]. Sidechains, on the other hand, can be defined as independent blockchain networks that are connected to another blockchain, typically called a mainchain or a parent chain, via a two-way bridge. Their functioning is similar to that of relays in the sense that every sidechain can read and verify the information on the main chain. The block data is passed onto the sidechain for each new block appended to the main chain. The sidechain itself implements the standard verification mechanism of the mainchain's consensus algorithm and can therefore verify the block's validity.

Cosmos and Polkadot are two of the most active projects using this technology to achieve cross-chain interoperability.

One issue with sidechain-based projects is that the inconsistency of consensus rates between different blockchains can impact the validity of cross-chain transactions. Another major drawback with sidechain implementations is that they typically have the ability to read and interpret data only from their parent chain or other sidechains connected to the parent chain, i.e., there is no support for communication with other blockchains.

2.6 Optimistic Bridges

Optimistic verification of cross-chain requests is another technique that has gained much traction in recent months. It is one of the more secure approaches to interoperability. Here is how optimistic bridges generally work:

Step 1: A user or an application posts data to a contract on the source chain.

Step 2: A third-party entity validates this data by signing a Merkle root containing the aforementioned data and committing it to the source chain. Some implementations of optimistic bridges require these entities to bond funds while signing the Merkle root. In the case of a fraudulent Merkle root, these funds are slashed.

Step 3: The root committed in the previous step is read by relayers and submitted to the destination chain.

Step 4: Following data submission to the destination chain, a challenge period starts wherein anyone watching the system can provide a fraud-proof and stop the transaction from going through.

Step 5: If no one flags the transaction as a fraud during the challenge period, the data is considered valid, and the transaction can be executed on the destination chain.

Optimistic bridges are considered trustless because they require only one honest node to watch the network to ensure no malicious activity occurs. However, the foundation of its core competency also gives rise to its most significant drawback - high latency. Any cross-chain request sent via an optimistic bridge cannot be executed with instant finality, i.e., applications/users will have to wait for the challenge period to end before their request is marked as completed. Such a solution is suboptimal for any user/application needing a low-latency solution, whether for asset transfer or a generic message transfer.

2.7 Existing Chain-based Approaches

To address the diverse variety of issues plaguing existing interoperability solutions, a few chain-based technologies have come forward in the past few months. Such solutions deploy a dedicated PoS blockchain that acts as a hub connecting various blockchains. During a cross-chain transaction, transactions mined on the source chain are validated on this hub chain by a dedicated network of validators. Following this, the transactions are relayed to the specified destination chain for the corresponding action.

Even though certain trust assumptions are involved in a chain-based interoperability solution, specifically on the chain's PoS validators, they have proved to be one of the most elegant interoperability solutions to date by finding a good balance between decentralization, security, and throughput. That being said, chain-based solutions have not been able to realize their full potential - the lack of support for a stateful middleware limits the features and use-case they can enable.

- **Application-level security and infrastructure-level security are tightly coupled:** Since dApps cannot deploy any middleware logic, they cannot enforce their own security mechanisms. All dApps have to rely on the underlying security mechanism of the blockchain.
- **No support for application-specific bridging logic:** Applications cannot persist any states in the middleware, limiting them from referring to historical information and implementing the "If this, then that" kind of logic in their applications.
- **Code redundancy:** The same business logic has to be implemented on contracts deployed across the chains.

3 Router Chain

3.1 What is Router Chain?

The Router chain is a layer 1 blockchain that leverages tendermint's Byzantine Fault Tolerant (BFT) consensus engine. As a Proof of Stake (PoS) blockchain, the Router chain is primarily run by a network of validators with economic incentives to act honestly. The Router chain is built using the Cosmos SDK and encapsulates all the features of Cosmos, including fast block times, robust security mechanisms, and, most importantly, CosmWasm - a security-first smart contract platform. By leveraging the CosmWasm toolkit, developers can start building secure blockchain applications on the Router chain from scratch or port their existing applications to the Router chain with minimal overhead.

In addition to its functionalities as a blockchain network, the Router chain provides an innovative solution to the problem of blockchain interoperability. Apart from validating state changes on the Router chain, validators running on the Router chain also monitor state changes on other chains. Applications on the Router chain can write custom logic to trigger events in response to these external state changes. Additionally, applications on the Router chain can leverage a trustless network of relayers to update states on external chains directly from the Router chain. Simply put, the Router architecture allows contracts on one chain to interact with contracts on other chains in a secure and decentralized manner. More details regarding the Router chain and how it enables cross-chain communication are given in the following sections.

3.2 Characteristics

3.2.1 Decentralized Trust-based

With Router V2, we chose a decentralized trust-based approach over a trustless approach due to the inability of the latter to support middleware contracts, which would have restricted the types of applications that could be built using Router. In the new approach, any cross-chain request initiated from a third-party chain has to go through the Router chain's tendermint-based PoS consensus mechanism. With multiple independent validators securing the network and a minimum validation requirement of two-thirds plus one vote (on the basis of voting power), the Router chain minimizes the amount of trust required by the user on the system. Furthermore, any validator having excessive downtime or engaging in any kind of malicious activity will be penalized by having a portion of their staked ROUTE slashed. This mechanism will ensure that validators have no economic incentive to carry out a malicious transaction or disregard a valid transaction.

3.2.2 Support for Middleware Contracts

One of the main characteristics of Router V2 is its ability to support middleware contracts. For the uninitiated, in the current interoperability setups, applications cannot enforce an "If this, then that" logic, as all transactions initiated on the source chain are routed to the destination chain as is. To add any application-specific bridging logic, applications must refactor their code and deploy it on multiple chains, which is both inconvenient and inefficient.

With Router, applications can leverage the middleware contract capability to implement custom business logic directly in the bridging layer:

- Features such as batching, sequenced transactions, and atomicity can be enforced directly from the Router chain.
- With stateful middleware in place, case-based routing is possible.
- Limited code redundancy - no need to duplicate computation logic; only the final execution function needs to be deployed on all external chains.

3.2.3 Router Chain as a Data Aggregation Layer

The Router chain can serve as the accounting and data aggregation layer for various cross-chain and multi-chain applications. For example, cross-chain governance can be carried out directly via a governance contract on the Router chain, which can allow users to create and vote on proposals. The

inbound and outbound modules will serve as a cross-chain synchronizer to communicate data and voting results between other chains (Ethereum, Polygon, BSC, etc.) and the Router chain.

3.2.4 Separate Inbound and Outbound Flows

Router splits all cross-chain requests between two third-party chains into an inbound flow and an outbound flow. The inbound flow involves the passing of requests from other chains to the Router chain, whereas, in the outbound flow, requests from the Router chain are forwarded to other chains. By having two separate flows for incoming and outgoing requests, Router ensures that:

1. Applications have greater control over their business logic (via middleware contracts between the inbound and outbound flow).
2. If required, transactions can be batched/sequenced on the Router chain before they are sent to the destination chain.
3. Cross-chain requests are validated by the Router chain's validator set before they are picked up by relayers.

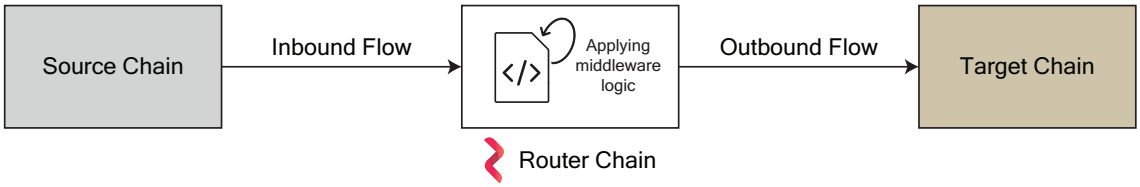

Figure 2: Inbound and Outbound Flow

Important attributes to keep in mind about the inbound and outbound requests:

- A set of orchestrators validates all inbound requests before they are forwarded to the Router chain. More information regarding orchestrators is provided in Section 3.4.3.
- All inbound requests do not necessarily need to have a corresponding outbound request. For example, any asset transfer from an external chain to the Router chain will only require an inbound request telling a contract on the Router chain to unlock funds.
- Not every outbound request is necessarily generated by an inbound request. For example, oracles on the Router chain may aggregate data from various sources, perform computation on a contract on the Router chain, and then broadcast the latest feed to all the other chains. This event has no inbound flow because the instruction to broadcast the feed can be given directly on the Router chain.
- All outbound requests are forwarded to the destination chain via a relayer. More details about relayers are given in Section 3.4.11.

Note: Applications that do not require middleware capabilities do not need to deploy any contract on the Router chain. They can use CrossTalk (see Section 4), a cross-chain framework by Router that abstracts the entire inbound and outbound flow in a single cross-chain request.

3.2.5 Multilayer Security

Applications building on the Router chain do not need to rely solely on the Router chain's security measures to secure their applications - applications can deploy and leverage a custom security layer on top of the infrastructure level security provided by the Router chain. For instance, before a cross-chain request is picked up by the relayer, an application can enforce an MPC-based or multisig verification of the inbound request on the middleware contract. In fact, applications can also implement custom security checks once the request reaches the destination chain. More about application-level security is given in Section 3.5.3.

3.2.6 Composability

Any infrastructure that encourages developers to build applications on top of it should be highly composable. Keeping that principle in mind, we have ensured that various out-of-the-box applications on the Router chain provide components and functionalities that developers can freely integrate into their applications.

3.2.6.1 Global Liquidity

Several use cases of a bridging solution, including but not limited to cross-chain staking, cross-chain prediction markets, and cross-chain lending/borrowing, depend directly on its ability to transfer funds across chains securely and efficiently. To that end, the Router chain ships with an inbuilt asset-swapping engine that acts as the gateway to the liquidity managed by Router Protocol. Any application requiring access to Router's fund transfer capabilities can tap into these liquidity pools to move funds.

Consider a project that wants to move funds from one chain to another along with an instruction to mint an NFT using the transferred funds. To do this, the application can create a sequenced request with two contract calls - the first call will unlock funds on the destination chain using Router's asset transfer bridge, and the second call will take the unlocked funds and execute the function to mint the user-specified NFT on the destination chain. Additional details about Router's asset-swapping capabilities will be unveiled in a separate paper, due to be published soon.

3.2.6.2 Oracles

One of the most critical requirements while building a dApp is that of a decentralized oracle - not just for price feeds of different assets, but for gas price estimation and other data feeds (based on application-specific use-case). To spare the developers from the painstaking process of sourcing and integrating reliable oracles, the Router chain will have a smart contract module that maintains multiple price feeds. This contract will fetch the price feed from reliable oracle providers, such as the Band Protocol, at regular time intervals.

3.2.6.3 Inter-Blockchain Communication Protocol (IBC)

IBC is a communication standard that allows applications built on any Cosmos-based chain to interact with each other. Since the Router chain is built using the Cosmos SDK, any application built on it can use IBC to interact directly with applications on other Cosmos blockchains like Injective, Osmosis, and others. This interaction can be a token transfer or an instruction transfer.

3.3 Workflows

Now that we understand the primary characteristics of the Router chain as an interoperability solution, let's examine the overall lifecycle of passing a cross-chain request via the Router chain. As stated in Section 3.2.4, all cross-chain requests between two external chains consist of two legs: inbound and outbound. In the inbound workflow, the transaction is sent from the source chain to the Router chain, whereas in the outbound workflow, the transaction is sent from the Router chain to the destination chain.

3.3.1 Inbound Worklow

Step 1: a) A user initiates a cross-chain action on an application's smart contract on the source chain.

b) The application contract calls the `requestToRouter()` function on the Router Gateway contract with the following parameters:

- `routerBridgeContract` - address of the smart contract to be invoked on the Router chain.
- `payload` - data to be passed to the bridge contract on the Router chain, which should include the destination chain details such as the `destChainType` and the `destChainId`, as well as an array of `contractCalls` that have to be made on the destination chain. In case the request does not have a corresponding outbound flow,

i.e., the destination chain is the Router chain itself, it need not include `destChainType` and `destChainId` in the payload.

- `gasLimit` - the gas limit required to execute the request on the Router chain.
- `feePayer` - address on the Router chain from which the cross-chain fee will be deducted. It can be either one of the three: (a) the user address, (b) the application contract address, or (c) `NONE`. If the `feePayer` address is set to `NONE`, then any entity on the Router chain can act as the `feePayer`.

Step 2: The Gateway contract on the source chain emits an event that is listened to by the orchestrators on the Router chain.

Step 3: After validating the event with the help of the attestation module, the inbound module marks the incoming request as validated.

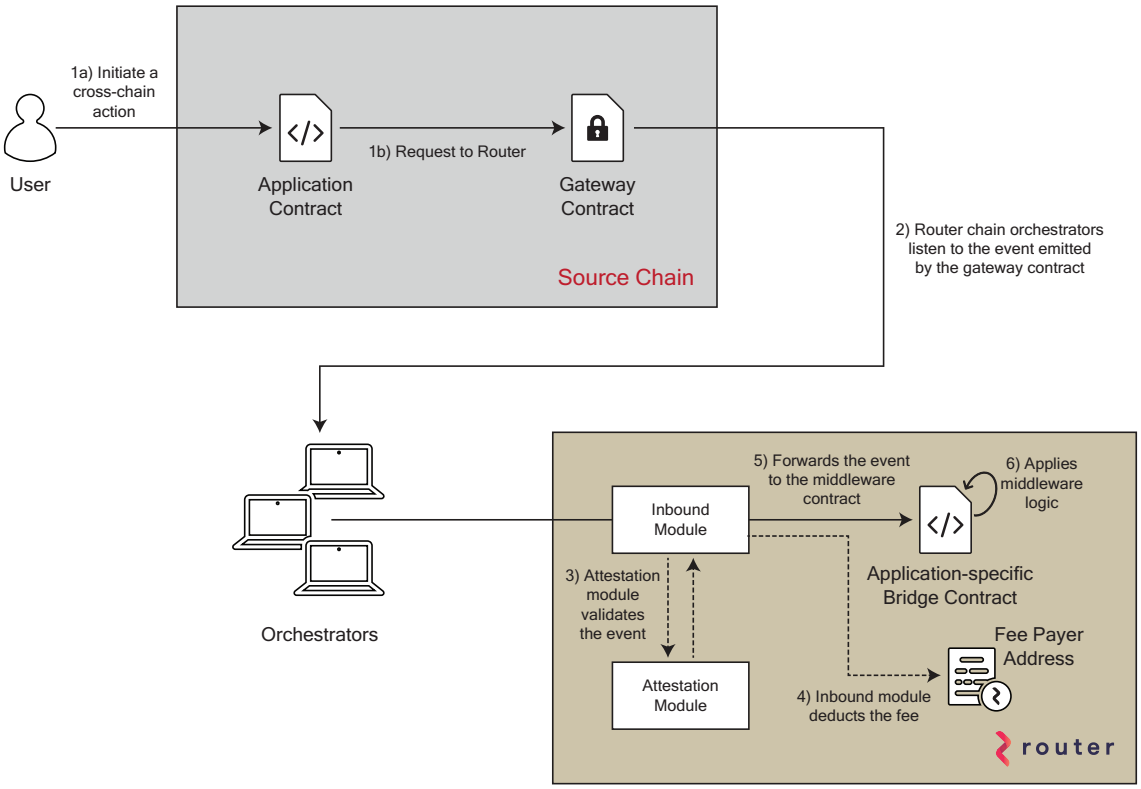

Figure 3: Inbound Workflow

Step 4: Once the incoming request is marked as validated, the inbound module will try to deduct the fee from the designated `feePayer` address as per the `gasLimit` set by the user and the `gasPrice` configured on the Router chain.

Step 5: Once the inbound module deducts the fee successfully, it will try to execute the transaction on the specified bridge contract.

Step 6: After the request reaches the bridge contract on the Router chain, the bridge contract will validate if the source contract address from which the request has been generated is correct or not, following which it will apply its custom logic based on inbound request.

- If the Router chain is the destination chain, the bridge contract will execute the relevant functions and terminate the request on the Router chain itself.
- If an external chain is the destination chain, the bridge contract performs the following tasks:

(i) applies the custom bridging logic

(ii) generates an outbound request

(iii) parses the payload to pass `destChainId` and `destChainType` as separate parameters in the outbound request

(iv) uses the gas price oracle to estimate and pass the `gasPrice` (see Section 3.4.9.1 for more details)

3.3.2 Outbound Workflow

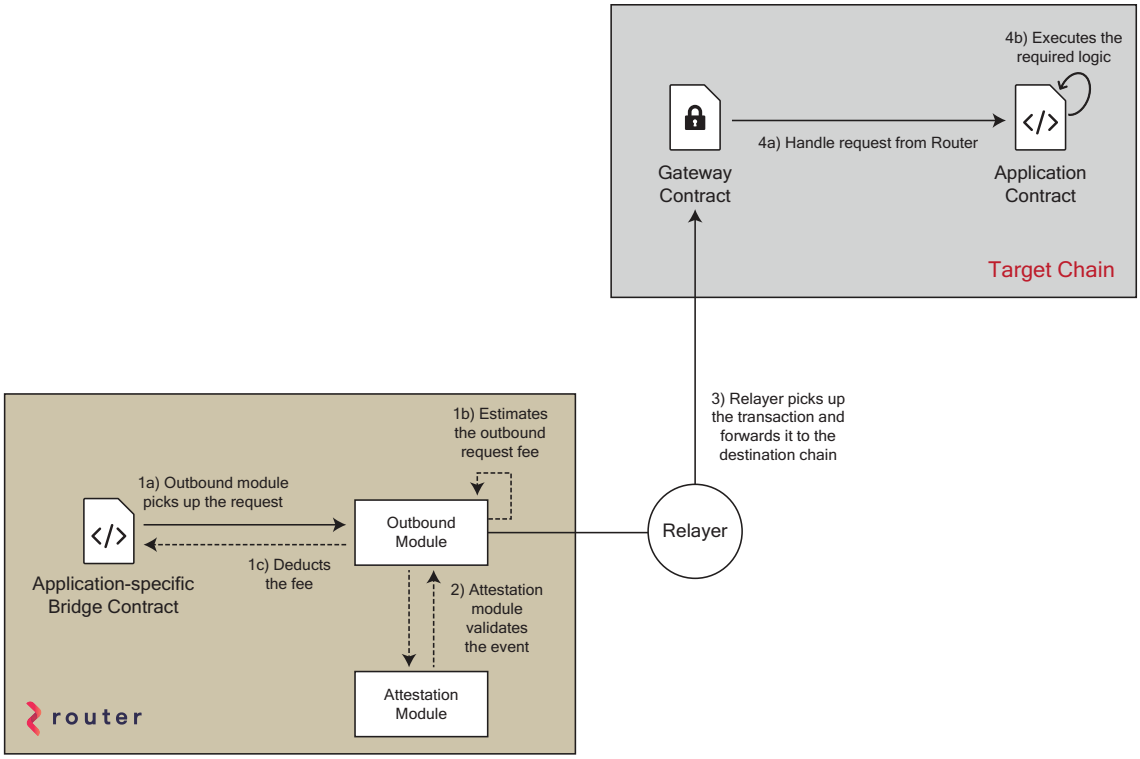

Figure 4: Outbound Workflow

Step 1: **a)** After the transaction initiated by the bridge contract is mined on the Router chain, the outbound module picks up the corresponding outbound request.

b) The outbound module estimates the `OutgoingTxFee` using the `gasLimit` and `gasPrice` specified by the bridge contract in the outbound request. To convert the fee from the destination chain's native token to ROUTE tokens, the outbound module uses the token price oracle (see Section 5.2.2 for more details).

c) The `OutgoingTxFee` is deducted from the bridge contract that created the outbound request.

Step 2: The outbound module collects all the signatures given by orchestrators and uses the attestation module to validate the outbound request.

Step 3: Once the majority voting power is achieved, the relayer polling the outbound requests of that particular bridge contract picks up the transaction and forwards the event to the Router Gateway contract on the destination chain.

Step 4: **a)** The Gateway contract on the destination chain calls the `handleRequestFromRouter()` function on the application contract on the destination chain.

b) The application contract on the destination chain will take appropriate actions based on the data transferred.

Note: In case there is no need for application-specific bridging logic, applications do not need to include a bridge contract on the Router chain. They can use Router's CrossTalk framework (refer to Section 4) to plug cross-chain functionalities into their existing codebase with minimal overhead.

3.3.3 Acknowledgment Workflow

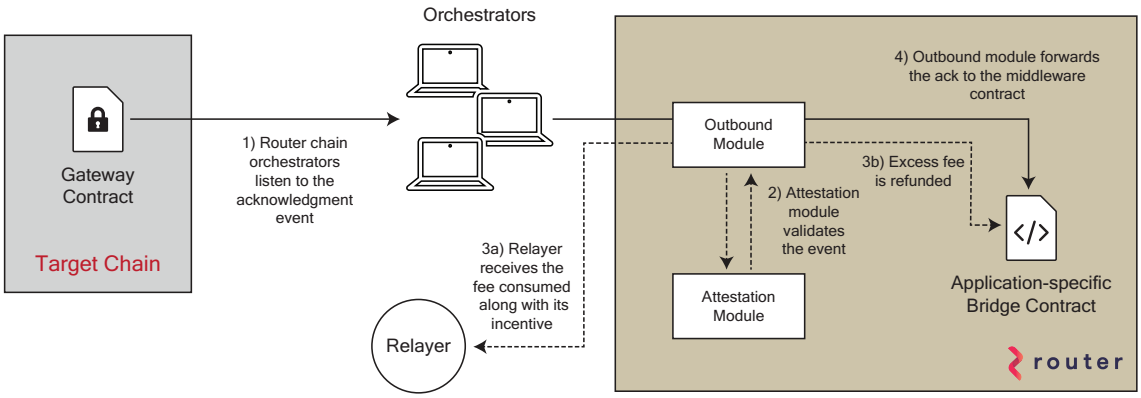

Figure 5: Acknowledgment Workflow

Step 1: After the `handleRequestFromRouter()` function execution is complete on the destination chain, the destination chain's Gateway contract emits an acknowledgment event that is listened to by the orchestrators on the Router chain.

Step 2: Upon receiving the intimation of majority confirmations from the attestation module, the outbound module marks the ack request as `VALIDATED`.

Step 3: Once validated, the acknewledgment request is processed on the Router chain:

- **a)** `(RelayerIncentive + FeeConsumed)` is transferred to the relayer address,
- **b)** `(OutgoingTxFee - FeeConsumed)` is refunded to the bridge contract on the Router chain.

Step 4: Once the acknowledgment is processed on the Router chain, it is sent to the application's bridge contract.

3.4 Architectural Components

3.4.1 Application Contracts

These are contracts deployed by applications on third-party chains and serve as the intermediary between end users of the application and the Router cross-chain infra. In the lifecycle of a cross-chain transaction, these contracts are responsible for making the `requestToRouter()` function call to the Gateway contracts on the source chain by passing the address of the bridge contract on the Router chain as well as the relevant payload. On the destination chain, application contracts will execute the instructions forwarded by the Gateway contract using the `handleRequestFromRouter()` function.

3.4.2 Gateway Contracts

As their name implies, Gateway contracts serve as the interface between application contracts on third-party chains and application-specific bridge contracts on the Router chain. Gateway contract functions include:

- `requestToRouter()` - The application contract on the source chain can call its corresponding bridge contract on the Router chain by invoking `requestToRouter()` on the Gateway contract with the relevant parameters. Upon receiving this function call, the Gateway contract emits a `SendToRouterEvent` that is listened to by the Router chain orchestrators.

- `requestFromRouter()` - The bridge contract on the Router chain can call its application contract on the destination chain by submitting an outbound request with the relevant parameters. Relayers will eventually submit the outbound request to the destination chain by invoking the `requestFromRouter()` function on the Gateway contract, which will subsequently call the `handleRequestFromRouter()` function on the application's destination contract to handle the transaction request coming from the Router chain.

3.4.3 Orchestrators

Router orchestrators are actors who work in conjunction with the Router chain's validators to listen to various inbound events from other chains, attest their validity, and forward them to the Router chain. They are also tasked with the responsibility of attesting to the validity of outbound requests before they can be picked up by the relayers. All validators must run an orchestrator to be a part of the Router chain ecosystem.

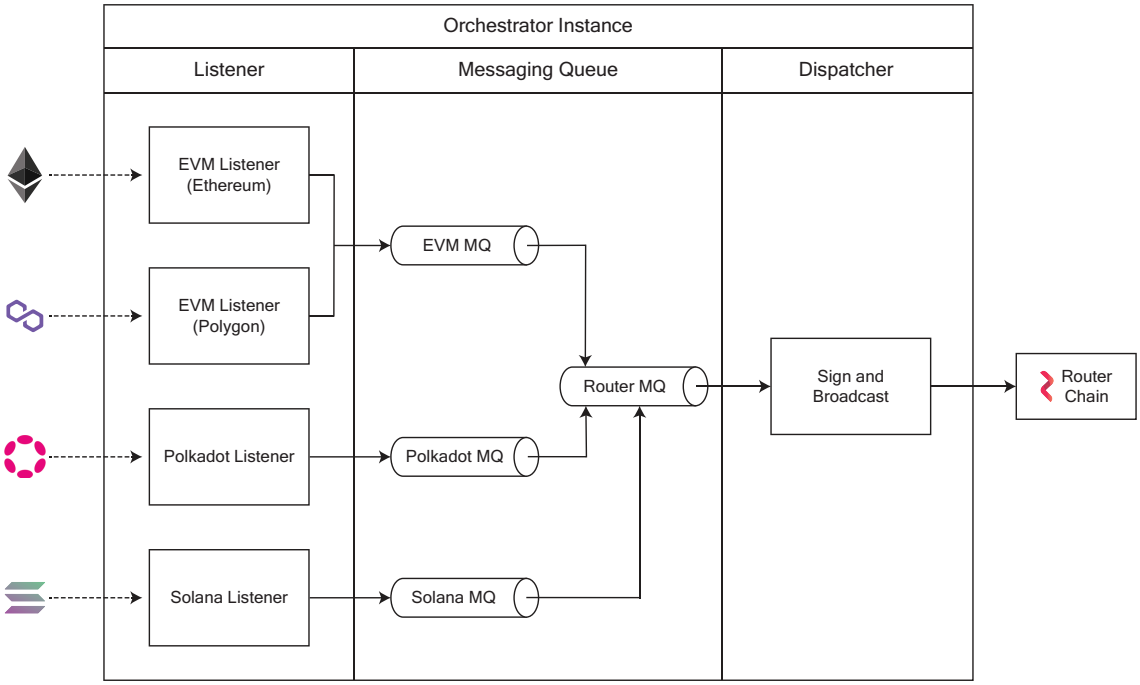

Figure 6: Orchestrator Architecture

Working

At a high level, a Router orchestrator works like a funnel that gathers events from various chains and posts them to the Router chain. To do so, an orchestrator uses a listener and dispatcher model wherein the listener module aggregates events while the dispatcher module forwards these events to the Router chain [10].

- **Listener:** The listener module of an orchestrator listens to events emitted from specific chains based on the `chainType` parameter in the configuration provided to it. Listeners operate as threads (goroutines) under an orchestrator. All listeners subscribe to multiple types of events, including a regular inbound request, an acknowledgment request, and a CrossTalk request, among

others. Once the listener module receives an event, irrespective of its type, it parses the event into the same format. Once the message is prepared, the listener adds it to the messaging queue.

- **Message Queue:** The message queues are used to enqueue and deliver transformed messages to consumers (dispatchers) in a first-in-first-out manner while ensuring that duplicate messages are automatically discarded.
- **Dispatcher:** The dispatcher module is essentially responsible for streamlining the incoming requests by (a) listening to the message queue, (b) signing the messages, and (c) broadcasting them to the Router chain.

Besides their inbound duties, orchestrators also verify the pending requests on the outbound module. To do so, an orchestrator has to listen to the transactions occurring on the Router chain.

Addressing Orchestrator Scalability

Listening to multiple blockchains at the same time is a resource-intensive task, and therefore, proper measures need to be taken to guarantee the scalability of the orchestrator module:

- **Multiple Threads:** Each orchestrator can run multiple listeners as goroutines/threads, each responsible for listening to one specific chain for Router-specific events. On top of the scalability it provides, this approach allows us to remain modular in our design. For chains that do not have support for Golang, instead of developing a new orchestrator, we can just build listeners and attach them to existing orchestrators to continue the operation.
- **Message Queuing using RabbitMQ:** Orchestrators on the Router chain use RabbitMQ [11], a dedicated message broker, to handle the message passing between the listener and the dispatcher module. RabbitMQ's ability to maintain states (messages) until they are received allows for rollbacks and failover handling without any overhead.

3.4.4 Inbound Module

As mentioned in Section 3.2.4, the inbound module acts as the interface to the Router chain for all the inbound requests from third-party chains. Upon receiving an inbound request from an orchestrator, the inbound module:

Step 1: Checks that the validator running that orchestrator instance is in the validator set.

Step 2: Checks if this request has already been received from other orchestrators. If not, the inbound module creates a new inbound event and persists it in the store.

Step 3: Waits for $\frac{2}{3} + 1$ validation from the attestation module - as the inbound module receives votes from all the orchestrators for an inbound request, it will submit votes to the attestation module.

Once the attestation module processes all the votes and attains a $\frac{2}{3} + 1$ majority, it triggers the inbound module, upon which the inbound module will:

a) Mark the status of the inbound request as `VALIDATED`.

b) Deduct the fee from the `feePayer` as per the `gasLimit` set by the user and the `gasPrice` configured on the Router chain. Once the fee is deducted, the inbound module will:

   (i) Change the inbound request's status to `READY_TO_EXECUTE`.

   (ii) Call the CosmWasm bridge contract with the relevant payload.

If the bridge contract call is successful, the inbound module will set the inbound request's state to `EXECUTED`. However, if the bridge contract call fails, the status of the request is set to `EXECUTION_FAILED`. This request can be retried later by sending a `retryInbound` tx to the Router chain. Refer to Appendix A to know more about the redelegation mechanism for failed requests.

3.4.5 Attestation Module

The Router chain is built on tendermint consensus, in which every block is processed only if at least $\frac{2}{3}+1$ validators (by voting power) approve the block. For cross-chain bridging using PoS, it is imperative to make sure that at least $\frac{2}{3}+1$ validators approve the inbound requests to and outbound requests from the Router chain. The attestation module solves this purpose.

Attestation Module Functionalities

1) Valset (Validator Set) management
More about valset and valset management is given in Section 3.4.6.

2) Aggregating all the votes received for an inbound/outbound request claim
The attestation module persists all the votes and emits a new event for each vote received by it from the inbound/outbound module.

Step 1: At the end of each block, the attestation module checks for the $\frac{2}{3}+1$ majority.

Step 2: Once a $\frac{2}{3}+1$ majority is attained for an inbound/outbound request, the attestation module increases the last observed event nonce of `chainId` on the Multichain module.

Step 3: Post that, it invokes the inbound/outbound module, stating that the validation is successful.

3.4.6 Validator Set (Valset)

Each validator set consists of a nonce, a list of validators, and the height on the Router chain at which the valset is created. The validator set on the Router chain should be consistent across all the Gateway contracts.

Updating the Valset

Here is how the valset is updated on all the third-party chains:

Step 1: At the end of each block, the attestation module checks if the valset power has changed by more than 5%. If it has, the attestation module creates a new valset request.

Step 2: Orchestrators will query the attestation module for the latest valset request and confirm the new valset request by sending a `MsgValsetConfirm` tx.

Step 3: Once the $\frac{2}{3}+1$ majority has approved the new valset request, the relayer will pick the valset request and send the `submitValsetUpdate()` contract call on all the Gateway contracts of all the configured chains in the Multichain module.

Step 4: The Gateway contracts will verify the signatures, replace the old valset with the new valset and emit a `ValsetUpdate` event.

Step 5: The orchestrators will listen to the `ValsetUpdate` event and submit a tx to the Router chain, confirming that the valset has been updated on all the chains.

Step 6: On receiving the confirmation from the $\frac{2}{3}+1$ majority, the Router chain will update the valset nonce in the Multichain module.

3.4.7 Multichain Module

The Multichain module persists the configuration of all the external chains supported by the Router chain and provides APIs to query the **ChainConfig**, which consists of the following:

- **ChainID:** Network ID of the supported blockchain, for example, 1 for Ethereum mainnet, 137 for Polygon, etc.
- **ChainName:** Name of the chain (Polygon, Ethereum, etc.)

- **Symbol:** Native gas token symbol for the supported chain (ETH for Ethereum, MATIC for Polygon, etc.)
- **ChainType:** EVM, Cosmos, Solana, Polkadot, etc.
- **ConfirmationsRequired:** To make sure that the tx is finalized, the orchestrator has to wait for a specified number of network confirmations on each blockchain, which is determined by this parameter
- **GatewayContractAddress:** Router Gateway contract address
- **GatewayContractHeight:** Router Gateway contract deployment height
- **RouteContractAddress:** ROUTE ERC20 contract address on the chain
- **LastObservedEventNonce:** Event nonce of the latest event that was observed on the chain (initially, it will be set to zero)
- **LastObservedValsetNonce:** Valset nonce of the latest `ValsetUpdate` event (initially, it will be set to zero)

Adding a New Chain

To support any new chain, the following steps need to be followed:

Step 1: Deploy a Router Gateway contract on the new chain with the current validator set.

Step 2: Create a chain integration governance proposal and send it to the Multichain module.

Step 3: If not already present, deploy a ROUTE token contract on the new chain.

Step 4: Once the governance proposal is passed, the requested chain config is added to the Multichain module.

3.4.8 Application-specific Bridge Contracts

The application-specific bridge contracts are middleware contracts deployed on the Router chain that are built using CosmWasm and include the logic required to process the inbound request from a third-party chain and forward the processed request to another third-party blockchain via the Router chain's outbound module.

To ensure that a faux contract doesn't execute any of the functions in these contracts, a bridge contract should always maintain a mapping of the `chainId` and addresses of all the application contracts (deployed on the third-party chains) that can execute its functions. Along with the payload, the Gateway contract will always pass the `msg.sender` parameter, which can be cross-referenced by the bridge contract to determine whether the source chain application contract is genuine or not.

3.4.9 Token and Gas Price Oracles

For a bridge contract to create and submit an outbound request from the Router chain to any external destination chain, it should be aware of the current gas price on the destination chain. Additionally, a bridge contract may require the price of any external chain's native gas token for internal calculation. To address this, we need oracles on the Router chain, which provision the token and gas prices of various chains.

3.4.9.1 Gas Price Oracle

The steps involved in querying gas prices and providing a generalized gas price oracle to the contracts on the Router chain are as follows:

Step 1: A simple microservice will be used to query the gas price on different chains and submit the same in the form of a transaction on the Router chain.

Step 2: The Router chain, upon receiving the gas prices from multiple providers, will take a median and update the oracle module state with the gas price.

Step 3: At any given time, application-specific bridge contracts can query the oracle module for the latest gas prices of external chains and pass the `gasLimit` parameter for the outbound request accordingly.

3.4.9.2 Token Price Oracle

Token prices of all the native tokens of all the chains in the Multichain module will be fetched from the Band Protocol via Cosmos' IBC. The system has been designed in a way that different types of providers can be supported in the long term. The steps involved in querying price feeds and providing a generalized token price oracle to the applications on the Router chain are as follows:

Step 1: At regular intervals, the Router chain will generate a Band IBC oracle request to the oracle module on the Router chain. The length of these intervals is decided using governance and added to the chain configuration.

Step 2: Upon receiving the request, the module will query the Band Protocol for the latest price feed of all the assets specified in the multichain module.

Step 3: Upon receiving the Band IBC price feed, the oracle module will update the latest price of the assets in its contract state.

Step 4: At any given time, any application can query the oracle module for the latest price feed of any chain's native asset. Upon receiving the request, the oracle module will return the most recent price of the specified asset from its contract state.

Note: In addition to the bridge contracts, the token price oracle is also used by the outbound module to estimate the outbound transaction fee in ROUTE tokens.

3.4.10 Outbound Module

The application-specific smart contract on the Router chain will send its outbound request to the outbound module with the call data, which eventually needs to be forwarded to the application contract on the destination chain. The steps involved in this process are as follows:

Step 1: An application-specific bridge contract submits an outbound request to the outbound module.

Step 2: The outbound module estimates the fee for the request in ROUTE tokens and deducts it from the bridge contract.

Step 3: Orchestrators submit confirmations for the outbound request.

Step 4: Upon receiving $\frac{2}{3} + 1$ majority confirmations, the relayer will broadcast the outbound transaction to the destination chain.

Step 5: Orchestrator receives an acknowledgment event from the Gateway contract, following which it creates and submits an acknowledgment to the outbound module. Upon $\frac{2}{3} + 1$ majority confirmations, the outbound ack is processed and

- `(RelayerFee + FeeConsumed)` is transferred to the relayer address,
- `(OutgoingTxFee - FeeConsumed)` is refunded to the bridge contract on the Router chain.

‘

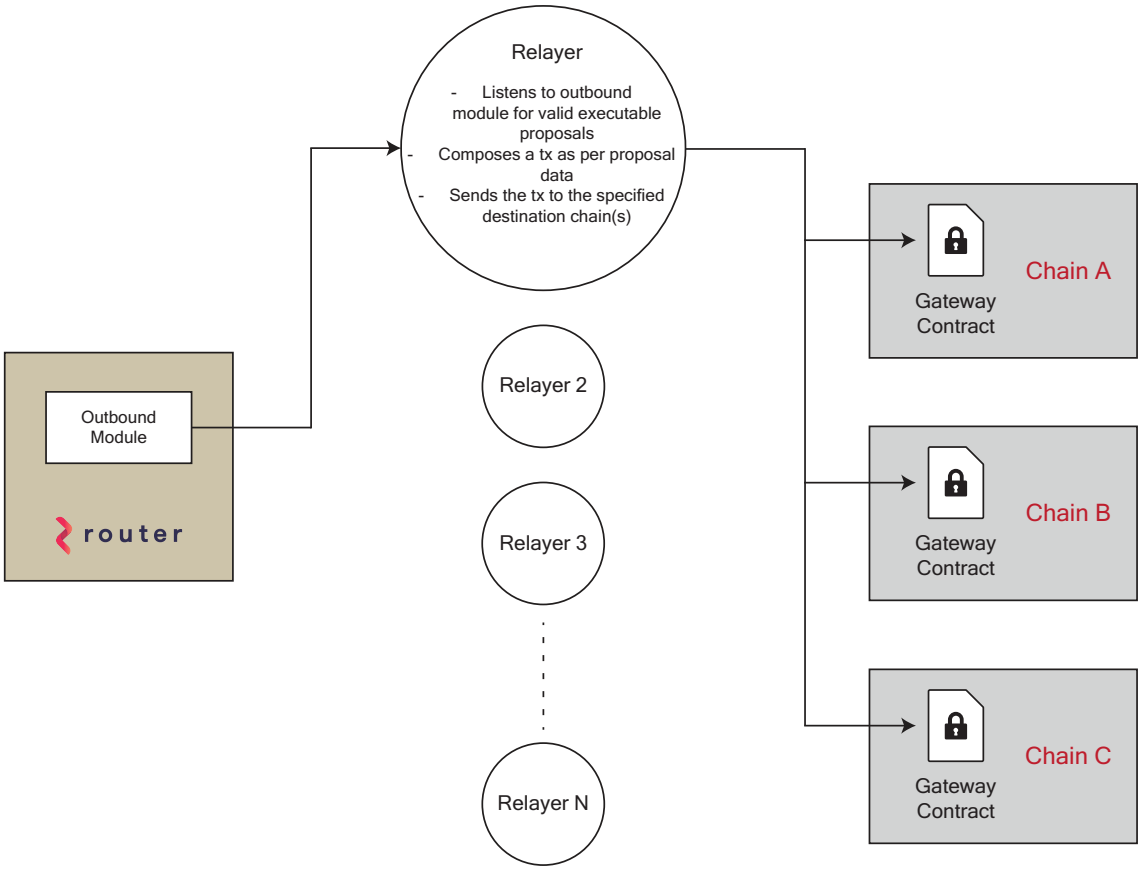

Figure 7: Relayer Functioning

3.4.11 Relayers

Relayers are permissionless entities that relay executable proposals from the Router chain to a specific destination chain. The Router chain has a set of relayers operated by various third parties, which distributes the responsibility. In the set, each relayer listens to the Router chain and relays data to the destination chains as and when required. These relayers also carry out subsequent actions based on the events that have been transmitted.

Functionalities

1. The relayer will be able to submit outbound requests to the destination chain.
2. The relayer can choose to whitelist bridge contract addresses and process outbound requests originating from only those addresses.
3. The relayer will be able to submit a `ValsetUpdate` request to all the destination chains configured on the multichain module.
4. The relayer will securely hold the private keys to wallets on different chains. These chains can be of distinct types, such as EVM, Cosmos, and Substrate, among others.

3.4.11.1 Working

Step 1: The listener interface of the relayer will listen to the Router chain for unprocessed outbound requests and add them to a tx queue.

Step 2: The processor interface of the relayer will fetch all the unprocessed requests from the tx queue.

Step 3: The relayer will transform the request into a defined message format where it queries the Router chain and fetches:

a) the current valset

b) payload from the outbound request

c) signatures of validators who signed the outbound request

Step 4: The relayer validates if the request has received $\frac{2}{3} + 1$ votes. If it has, it will estimate the gas price required to submit the outbound request to the destination chain and check if sufficient gas price is provided by the bridge contract to execute the request.

Step 5: Finally, the request is relayed to the destination chain defined in the outbound request.

Step 6: Once the request is successfully executed on the destination chain and an acknowledgment is received for the same, the relayer receives the fee it incurred in posting the tx on the destination chain and an additional fee on top of it.

3.4.11.2 Addressing Relayer Collisions

In some cases, multiple relayers may pick up the same outbound request. To avoid collisions while submitting the transaction on the destination chain, relayers may choose to implement collision prevention strategies at their end. For example, relayers can include a specified time offset within their logic to ensure they wait for a certain amount of time before delivering the transaction to the destination chain. If another relayer submits the transaction within this time frame, they can simply discard it. Even if relayers do not implement any collision prevention strategy, no transaction that has already been executed will ever get replayed thanks to the event nonce-level validation done by the Router Gateway contracts on the destination chain. The Gateway contracts always maintain a mapping of the most recent event nonce that has been executed. Since event nonces are incremental, if any request with an event nonce equal to or less than the mapped event nonce is received, it is ignored by the Gateway contract.

3.4.11.3 Addressing Scalability Constraints via the Use of Application-specific Relayers

Scaling issues might arise if we process the outbound requests sequentially, i.e., in the order of event nonce. To address this, the Router chain's relay architecture allows for the parallel execution of outbound requests. Since the relayer network is permissionless, each application can run its custom relayer to process its outbound requests. This way, an outbound request from one application bridge contract does not affect the outbound requests from another.

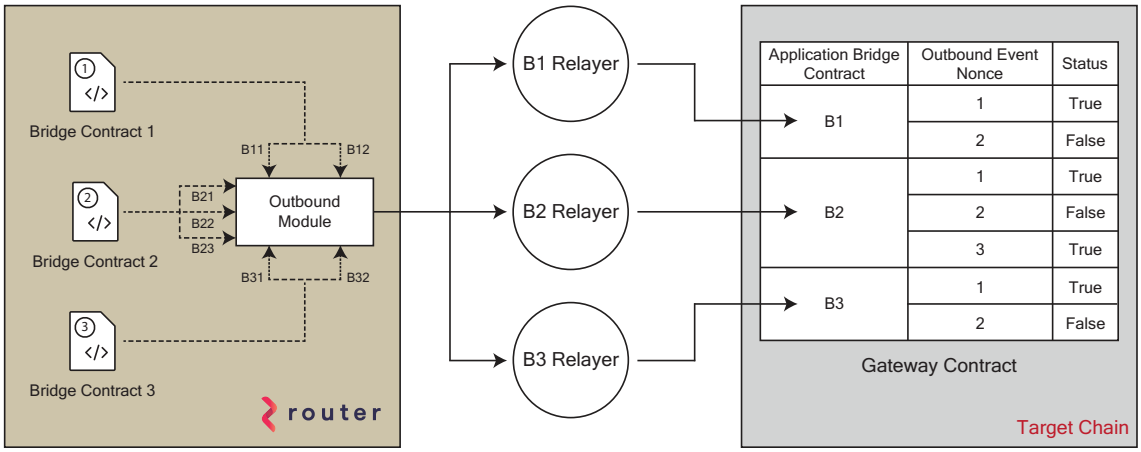

Figure 8: Improved Scalability using Application-specific Relayers

3.4.11.4 Gas Estimation

All relayers will need to run a `GasEstimator` module to estimate the gas price required to submit the outbound request to the destination chain. If a third-party estimator service like Owlracle, Eth Gas Station, or others is available, the relayer will estimate the gas using that; if not, the relayer will estimate using an RPC endpoint.

3.4.11.5 Trustlessness

To ensure that the data forwarded by the relayer is not tampered with, the Gateway contract on the destination chain calls the `validateVotes()` function, which decodes the signed call data to verify the validator signatures. Upon ensuring the authenticity of the request, the call data is executed.

3.4.11.6 Manual Relaying using Router's Web Relayer

Web relayer is a user interface being provided by the Router team using which a tx payload can be manually relayed on the destination chain.

- If anyone wants faster execution for their request on the destination side, they can manually increase the gas price and perform the tx from their wallets using the web relayer.
- Transactions stuck on the Router chain for any reason can be replayed using the web relayer.

3.5 Network Security

3.5.1 Infrastructure-level Security

Security is an essential aspect of any blockchain network. The Router chain derives its security from its underlying tendermint consensus engine.

Decentralized Network of Validators

For any block to be mined, $\frac{2}{3} + 1$ validators by voting power will be required to achieve consensus, meaning the network can only be compromised if validators holding a combined voting power of more than 67% decide to collude and engage in malicious activities.

PoS Economics

Any node having excessive downtime or engaging in any kind of malicious activity will be penalized by having a portion of their staked ROUTE slashed. This mechanism will ensure that the validator nodes have no economic incentive to carry out a malicious transaction. Moreover, all validators who are properly aligned with the network will be eligible for a portion of the block rewards. More about the validator economics will be disclosed in the subsequent versions of this paper.

Byzantine Fault Tolerance

The Router chain can tolerate up to $\frac{1}{3}$ of its validator nodes being faulty. This includes both inactive nodes and malicious nodes.

ED25519 Signature Scheme

All communication between the Router chain validators is secured by ED25519 [12], an elliptic curve-based encryption scheme [13], considered one of the fastest and most secure authentication mechanisms.

Decentralized Governance

Any updates on the Router blockchain will be made after a governance decision undertaken by the ROUTE token holders and stakers.

3.5.2 Bridge-level Security

Any inbound request from an external chain to the Router chain undergoes a validation process wherein the orchestrators attest to the presence of the inbound event's corresponding source chain transaction. Similarly, before an outbound request is picked up by a relayer, the orchestrators verify if the event's corresponding transaction on the Router chain has been mined. For signing the attestations during both the inbound and outbound validation, the orchestrators use the Elliptic Curve Digital Signature Algorithm (ECDSA) [14].

3.5.3 Application-level Security

3.5.3.1 On Router Chain

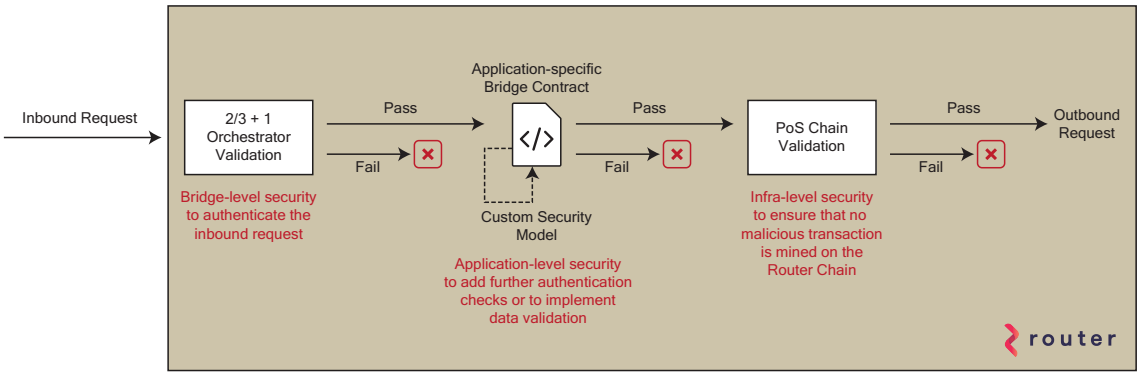

Figure 9: Multilayer Security on the Router Chain

The Router chain provides generic message-passing bridge security - it ensures that the data reaching the destination chain is authentic, i.e., the data/instruction generated by the application on the source chain reaches unaltered to the destination chain. However, to add another layer of security - whether to check the validity of the data or to have an additional authenticity check, applications can put in smart contract-level checks like having their own validators sign the contract on the Router chain before the data is forwarded to the outbound module and eventually to the destination chain.

Additionally, applications can design and implement their custom governance strategies to handle updates to the application. In fact, due to the ability of Router chain contracts to change the state of contracts deployed on external chains, the governance decisions taken on the Router chain can be broadcasted and executed on other chains.

3.5.3.2 On Destination Chain

For developers who do not wish to implement application-level checks on the Router chain itself or who are looking to leverage Router's CrossTalk framework instead of the middleware flow, we will be enabling a customizable security layer on all the external chains. Akin to Hyperlane's Sovereign Consensus [15], this security layer will allow applications to specify their own security standards, such as an m-out-of-n multisig or a fraud-proof-based authentication model (optimistic approach), among others.

4 CrossTalk

4.1 What is CrossTalk?

Router's CrossTalk library is an extensible cross-chain framework that enables seamless state transitions across multiple chains. In simple terms, this library leverages Router's infrastructure to allow contracts on one chain to pass instructions to contracts deployed on another chain without needing to deploy any contracts on the Router chain. The library is structured in a way that it can be integrated seamlessly into your development environment to allow for cross-chain message passing without disturbing other parts of your product.

4.2 Why is CrossTalk Required?

Based on our initial market research, we have discovered that a fair number of cross-chain applications do not require a customized bridging logic for their functioning. Instead, they merely need an infra layer that allows them to pass instructions between different blockchain networks. And while the Router chain middleware flow enables the applications to route instructions/messages from one chain to another, to do so, they need to deploy a dedicated contract with the routing logic on the Router chain. Even though we provide a template for a standard bridging contract, we do not want the developers, especially the ones new to the Cosmos ecosystem, to go through the task of deploying and managing a middleware contract.

With CrossTalk, applications can transform their existing single-chain and multi-chain applications into cross-chain applications by including nothing but a few lines of code in their existing contracts. Therefore, for cross-chain dApps that do not require custom bridging logic or any data aggregation layer in the middle, Router's CrossTalk framework is the best option.

4.3 CrossTalk Workflow

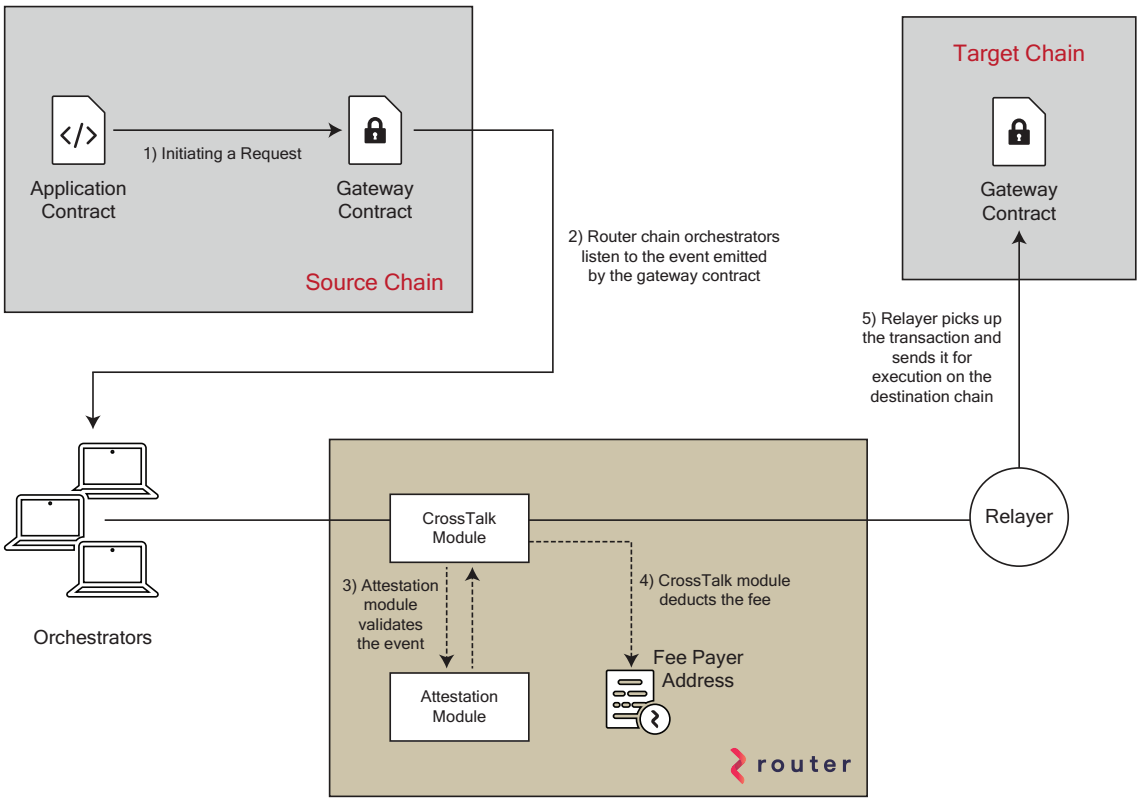

Figure 10: CrossTalk Workflow

Step 1: Initiating a Request

An application initiates a CrossTalk Request on the source chain by calling the `requestToDest()` function on Router's Gateway contract with the following parameters:

- `contractCalls` - Includes an array of payloads and an array of destination chain contract addresses where the respective payload should be sent. The payload will include the ABI-encoded function calls that are to be executed on the destination chain.
- `requestArgs` - A struct comprising of the following subparameters:

- ⋆ `isAtomicCalls` - Since users can use `requestToDest()` to send multiple cross-chain calls in one go, `isAtomicCalls` boolean value ensures whether the calls are atomic.
- ⋆ `expiryTimestamp` - The timestamp by which your cross-chain call will expire.
- ⋆ `feePayer` - This specifies the address on the Router chain from which the cross-chain fee will be deducted.

- `destinationChainParams` - Via this parameter, you can specify the gas you are willing to pay to execute your request along with the details of the destination chain where the request needs to be executed. This parameter needs to be passed as a struct containing the following subparameters:
  - ⋆ `gasLimit` - The gas limit required to execute the cross-chain request on the destination chain.
  - ⋆ `gasPrice` - The gas price the user is willing to pay for the execution of the request. If passed as 0, this will be estimated by the chain's gas price oracle.
  - ⋆ `destChainId` - Chain ID of the destination chain in string format.
  - ⋆ `destChainType` - This represents the type of chain. The values for chain types are given in table 1.

Table 1: Types of External Chains

Chain Type	Value
EVM	0
Cosmos	1
Polkadot	2
Solana	3
Near	4

- `ackType` - When the contract calls are executed on the destination chain, the Router chain receives an acknowledgment from the destination chain, which specifies whether the execution was successful or did it result in some error. Users can choose to get this acknowledgment from the Router chain to the source chain. Valid `ackType` values are:
  1. `NO_ACK` - when no acknowledgment is required
  2. `ACK_ON_SUCCESS` - when an acknowledgment is required only if the request is successfully executed
  3. `ACK_ON_FAILURE` - when an acknowledgment is only required in case of failures
  4. `ACK_ON_BOTH` - when an acknowledgment is required whether or not the request succeeds
- `ackGasParams` - If you opted to receive the acknowledgment on the source chain, you would need to write a callback function to handle the acknowledgment. The ackGasParams parameter includes the `gasLimit` and `gasPrice` required to execute the callback function on the source chain when the acknowledgment is received.

Once the source chain tx is successful, the Gateway contract will emit an event for the same.

Step 2: Listening to the Event

After listening to the event generated by the source chain Gateway contract, the orchestrator parses it into a valid tx and broadcasts it to the CrossTalk module on the Router chain along with its signature. The CrossTalk module is similar to the inbound module in its functioning. More about the CrossTalk module's working is given in the steps that follow.

Step 3: Validating the CrossTalk Request

Upon receiving the tx from the orchestrator, the CrossTalk module:

- Checks if this request has already been received from other orchestrators. If not, it creates a new CrossTalk request and persists it in its store.
- Performs the required $\frac{2}{3} + 1$ validation with the help of the attestation module. Once completed, the CrossTalk will change the status of the CrossTalk request to `VALIDATED`.

Step 4: Deducting the Fee

Once the request is validated, the CrossTalk module will deduct the fee from the `feePayer` as per the set `gasLimit` and `gasPrice`. Once the fee is deducted successfully, the status of the request is changed to `READY_TO_EXECUTE`.

- If `feePayer` is set to `NONE`, or the designated `feePayer` address does not have sufficient balance to pay for the fees, the status of the request will remain in the `VALIDATED` state. In such a scenario, anyone can become the `feePayer` for the request by sending a `PayFee` tx. By leveraging this feature, applications can also enable cross-chain meta transactions for their end users. Refer to Section 6.1 for more details regarding Router's cross-chain meta transactions.
- To ensure that the `feePayer` cannot sabotage the transaction by lowering the gas parameters, we have a check to ensure that it can only increase the `gasPrice` and `gasLimit` given as arguments in the `requestToDest()` function.

Step 5: Execution

Once the request is ready for execution, a relayer will try to pick it up and execute it on the destination chain by sending a transaction to the destination chain Gateway contract.

4.4 CrossTalk Acknowledgment Workflow

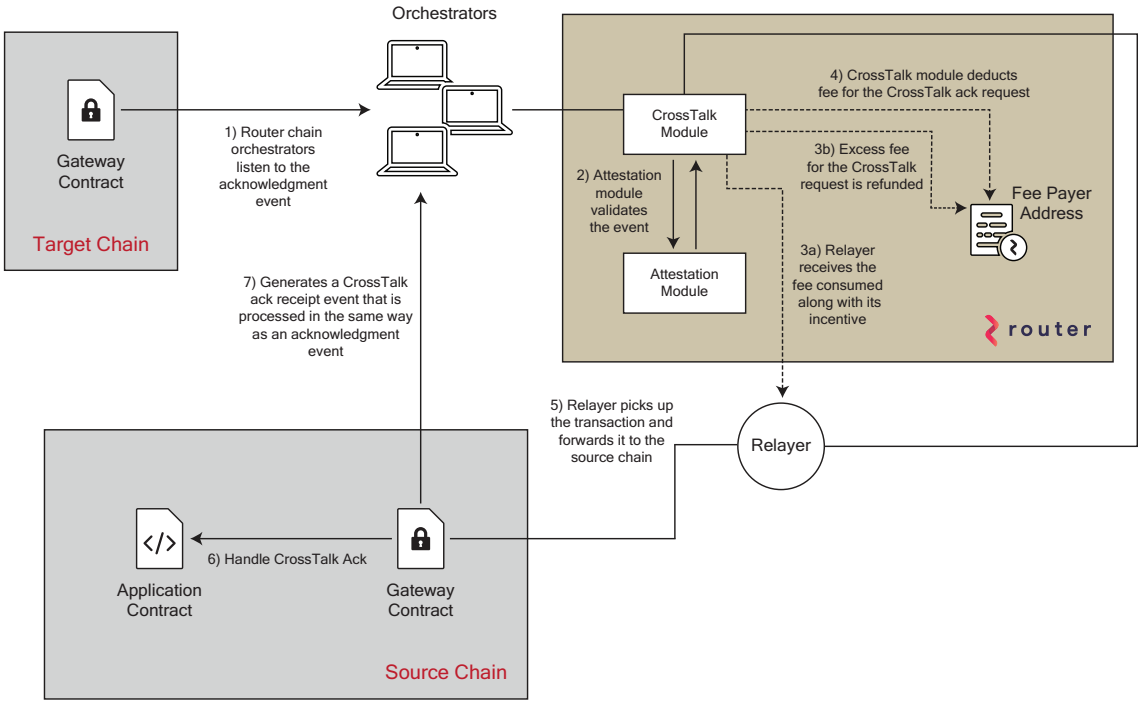

Figure 11: CrossTalk Acknowledgment Workflow

Step 1: Listening to the Acknowledgment Event

Once the CrossTalk Request is successfully executed, the destination chain Gateway contract will emit a CrossTalk acknowledgment event. The orchestrator listens to this event and composes a Router chain tx. The orchestrator broadcasts this tx with its signature to the CrossTalk module on the Router chain.

Step 2: Validating the CrossTalk Acknowledgment Request

Upon receiving the tx from the orchestrator, the CrossTalk module:

- Checks if this request has already been received from other orchestrators. If not, it creates a new CrossTalk ack request and persists it in its store.
- Conducts the required $\frac{2}{3} + 1$ validation with the help of the attestation module. Upon successful validation, the request is marked as `VALIDATED`.

Step 3: Processing the Relayer Incentive

Once the acknowledgment is validated, the CrossTalk module (a) processes the incentive for relayer, and (b) refunds the extra fee, if any, to the `feePayer`.

Step 4: Deducting the Fee

After the relayer incentive is processed, the CrossTalk module will check whether the user's acknowledgment request specification matches the type of acknowledgment received from the destination. For example, this check will return false if the user requested acknowledgments only in case of failures, but the request was successful.

- If true, the CrossTalk module will deduct the fee from the `feePayer` as per the configured `gasLimit` and `gasPrice` in the `ackGasParams`. Once the fee is deducted, the status is changed to `READY_TO_EXECUTE`.
- If not, the acknowledgment will remain in the `VALIDATED` state.

Step 5: Execution

Once the acknowledgment request is ready for execution, a relayer will try to pick it up and execute it on the source chain by sending the a tx to the source chain Gateway contract.

Step 6: Handling the Acknowledgment on the Source Chain

Applications need to include a `handleCrossTalkAck()` function in their smart contracts to handle the acknowledgment from the destination chain. This function will receive the following parameters:

- `eventIdentifier` - This is the same nonce you receive while calling the `requestToDest()` function on the source chain Gateway contract. You can verify whether your request was executed on the destination chain using this nonce.
- `execFlags` - Since multiple payloads can be sent to multiple contract addresses on the destination chain, the `execFlags` is an array of boolean values that tells you the status of the individual calls.
- `execData` - This parameter includes an array of bytes that provides the return values from every call included in the CrossTalk request. Based on the application's requirement, this data can be decoded and processed on the source chain

Step 7: Generating and Processing the CrossTalk Acknowledgment Receipt

Once the acknowledgment is successfully processed on the source chain, the Gateway contract will emit an event for the CrossTalk acknowledgment receipt. This event will be processed in the same way as a CrossTalk acknowledgment event:

- Orchestrators will listen to this event and forward it to the CrossTalk module.
- The CrossTalk module will validate it with the help of the attestation module.
- Finally, the relayer incentive will be processed, and the excess gas paid for the acknowledgment will be refunded to the `feePayer`.

4.5 Different Types of CrossTalk Requests

Now that we have an understanding of the CrossTalk workflow, let us take a look at the different types of requests that can be sent using CrossTalk. We will be categorizing the requests based on three different properties:

1) Number of Contract Calls

Depending on the number of contract calls present in a cross-chain request, a CrossTalk request can be categorized into two types:

- **Single-call Request:** A request that includes only one contract call for execution on the destination chain.
- **Multi-call Request:** A request that includes multiple contract calls for execution on the destination chain.

Consider an application that allows users to transfer their ERC20 tokens from one chain to another. If only one ERC20 token is being transferred, then the request will fall under the former category. However, if multiple tokens are transferred in a single request, it will be categorized as a multi-call request.

2) Atomicity

For single-call requests, the value of the `isAtomicCalls` parameter is ignored since there is only one contract call to execute. However, in the case of multi-call requests, the value of the `isAtomicCalls` flag can separate a request into two categories:

- **Non-atomic Request:** If `isAtomicCalls` is set to false.
- **Atomic Request:** If `isAtomicCalls` is set to true.

3) Acknowledgment Requirement

Depending on the need for an application to receive an acknowledgment for its request on the source chain, a CrossTalk request can be split into two types:

- **Requests Without Acknowledgment:** If an acknowledgment is not required on the source chain after the request is executed on the destination chain, the `ackType` can be set to `NO_ACK`, and the gas limit and gas price for `ackGasParams` can be sent as $(0, 0)$.
- **Request With Acknowledgment:** If an acknowledgment is required, the `ackType` param can be set to `ACK_ON_SUCCESS`, `ACK_ON_FAILURE`, or `ACK_ON_BOTH`, depending on the requirement. A valid gas price and limit must also be set under the `ackGasParams` for executing the acknowledgment handler function on the source chain. The gas price can be passed as 0 if you want to delegate the acknowledgment request's gas price estimation to the gas price oracle on the Router chain.

If an acknowledgment is anticipated on the source chain, an acknowledgment handler function with the relevant logic to handle the acknowledgment on the source chain has to be implemented by the application in their contract. If an acknowledgment is not anticipated, the acknowledgment handler function can be left empty, as it will never get invoked.

5 Fee Management

We have briefly discussed the fee mechanisms for a cross-chain request via the Router middleware, Voyager, and CrossTalk while talking about their respective workflows. In this section, we'll take a deeper look at all aspects of our fee model.

5.1 Gas and Fee Payer Considerations

- The gas price for the execution of inbound transactions on the Router chain is decided via governance and included directly in the chain configuration.
- In the case of the middleware flow, the `gasLimit` and `feePayer` addresses specified by the user on the source chain are used for executing the inbound request on Router's bridge contract. And since the gas price for the execution of a transaction on the Router chain is pre-configured on the chain itself, users are not required to pass a separate `gasPrice` parameter while sending a cross-chain request via the middleware flow.
- However, in the case of CrossTalk, users are required to specify both the `gasPrice` and `gasLimit` in their request. This is because the gas parameters and the fee payer specified in a CrossTalk request are used for the execution of the request on the destination chain. In case the `gasPrice` is set to 0, it will be estimated by the gas price oracle module on the Router chain.
- Since we are using the ECDSA, an EVM-based address can easily be converted to a Router chain address and vice versa, which means that the sender address will have a corresponding address on the Router chain. Therefore, the `senderAddress` can also be set as the `feePayer` address.
- While exercising the option to run their own relayer, applications might want to leave the task of gas limit estimation to the relayers. In such a scenario, they can pass the `gasLimit` as 0.

5.2 Middleware Flow Fee Model

As discussed in the previous sections, the middleware flow on the Router chain is divided into two parts - inbound and outbound, each of which has a fee associated with it.

5.2.1 Inbound Request Fee Structure

To execute an inbound request on the Router chain, users are required to configure a `gasLimit` parameter in their cross-chain request. This `gasLimit` is multiplied by the `gasPrice` present in the Router chain configuration to calculate the amount of ROUTE tokens to be deducted from the user-specified `feePayer` address. This fee is used to cover the cost of transaction execution (bridge contract call) on the Router chain. Note that if your `feePayer` does not have sufficient ROUTE balance, the transaction will not be executed. Since the `feePayer` address cannot be changed once set, you will have to top up the `feePayer` address to ensure the execution of your request.

5.2.2 Outbound Request Fee Structure

The bridge contract must pass reasonable `gasPrice` `gasLimit` parameters to cover the cost of executing the outbound leg of the cross-chain request on the destination chain. Once the outbound module receives the outbound request, it queries the oracle module for the latest price of the ROUTE token and the native gas token of the specified destination chain. It uses the gas price fetched using the gas price oracle, and the token prices fetched using the token price oracle to convert the gas cost involved in the execution of the outbound request from the destination chain native token to the ROUTE token.

5.3 CrossTalk Fee Model

5.3.1 Deducting Fee

CrossTalk works on a prepaid fee model. Upon receiving the CrossTalk Request, the Router chain will calculate the estimated fee for executing the transaction on the destination chain in terms of ROUTE tokens and deduct the fee plus incentive from the `feePayer` address upfront.

```
EstimatedFeeInRoute = EstimatedFeeInDestNativeToken * PriceRatio
```

where:

```
EstimatedFeeInDestNativeToken = DestGasLimit * GasPriceInDestNativeToken
PriceRatio = DestNativeTokenPrice / RouteTokenPrice
```

Important Notes:

- Fee and relayer incentive for any cross-chain request on Router have to be paid in ROUTE tokens only.
- To prevent Sybil attacks on the Router chain, Router's Gateway contract on the source chain charges a minimal fee from the application contract to cover the cost of orchestrator validation. This fee is paid in the source chain's native token.

5.3.2 Handling Refunds

Once the Router chain receives the CrossTalkAck generated by the destination chain's Gateway contract, it (a) pays the relayer address the incentive + fee used from the already deducted fee, and (b) refunds the `feePayer` address the extra fee deducted. This mechanism ensures the following:

- The relayer receives its incentive automatically without any delay.
- The applications can send extra gas limit as a buffer since they will get automatic refunds in case of a surplus fee.

6 Features

6.1 Cross-chain Meta Transactions

With Router V2, we are coming up with a first-of-its-kind cross-chain meta-transaction capability. While sending a CrossTalk or an Inbound Request, users/applications can configure a `feePayer` parameter, which specifies the address from which the fee is to be deducted on the Router chain. This parameter can be set to either one of the following: (a) the user address, (b) the application contract address, or (c) `NONE`. After a request is marked as validated by the Router chain, if the `feePayer` address is set to `NONE`, then any entity on the Router chain can function as the `feePayer` by sending a `PayFee` tx. By delegating the execution of a request to a third-party service, applications can enable feeless cross-chain transactions for their end users.

6.2 Decentralized Cross-chain Read Requests

One of the most underrated, albeit important, aspects of blockchain interoperability is being able to read the state of contracts present on one chain (say chain A) from a different chain (say chain B). A good example of this could be a Soulbound Token (SBT). Let us assume that every user gets a SBT on chain A, which contains the user's Date of Birth (DoB) information. This information can come in handy for multiple dApps that want to restrict users below a particular age. Creating this SBT on multiple chains will not make sense, but having the information of the SBT across multiple chains is essential for dApps to be able to access this information and use it. To achieve this, applications can use Router to generate a decentralized read request between two chains. This request will include (a) the

contract state to read on the destination chain (in this case, the user's age) and (b) the operation to be performed when the data is received back on the source chain (in this case, it could be to accept/deny user's request to access a gambling application). More details in regards to generating, sending, and handling a cross-chain read request are given in Appendix B.

6.3 Transaction Batching

Transaction batching can help a lot of applications cut costs for their cross-chain operations. Depending on the requirement of the application, Router can be asked to execute batches of transactions either directly on the Router chain or on the destination chains.

6.3.1 Batching at the Router Chain

Router V2 enables applications to batch and execute transactions directly on the Router chain with the help of schedulers. Consider a scenario wherein users want to mint an NFT on Polygon from various chains. We can aggregate users' requests from different chains on the bridge contract, i.e., collate all the requests in a single payload and then execute that call when a certain number of requests are received. This kind of system will save costs involved in (a) executing individual transactions on the Router chain and (b) relaying separate transactions to the destination chain, in this case, Polygon. The entire step-by-step flow of batched cross-chain NFT minting can be found in Appendix C.

6.3.2 Batching at the Destination Chain

For executing a transaction batch on the destination chain(s), the bridge contract will create and send an `OutboundBatchRequests` object, which is an array of `OutboundBatchRequest`, to the Router chain. Note that destination chain based batching can be of two types:

a) single-chain batching, wherein all transactions in a batch need to be executed on the same destination chain, and

b) multi-chain batching, wherein batches within the `OutboundBatchRequests` array can span multiple destination chains.

Single-chain Batching

For every `OutboundBatchRequest`, i.e., for any batch going to the same destination chain, transactions are, by default, executed in the order in which they are specified. The Router chain will create an `OutgoingBatchTx` for each `OutboundBatchRequest`, which eventually gets signed by orchestrators and broadcasted to the Gateway contract by a relayer. Each `OutboundBatchRequest` will consist of an array of `ContractCalls` - an object consisting of `DestinationContractAddress` and `Payload`.

Multi-chain Batching

In multi-chain batching, the `OutboundBatchRequests` array consists of batches that have to be executed on separate destination chains. Multi-chain batching can further be divided into two categories:

1) Ordered multi-chain batching: All batches need to be executed in sequential order.

2) Unordered multi-chain batching: Batches can be executed in any order.

For the latter, the relayer will broadcast all the batches in an `OutboundBatchRequests` array simultaneously to the respective destination chains. All the transactions within each batch will be executed sequentially as per the single chain batching mechanism. For the former, however, applications will have to enforce it using the bridge contract on the Router chain -

Step 1: The bridge contract submits the first batch request to the Router chain.

Step 2: The outbound module sends the batch request to the designated destination chain.

Step 3: The bridge contract receives an Ack for its request.

Step 4: If the Ack is okay, the bridge contract submits the next batch request to the Router chain. Steps 2, 3 and 4 can be repeated until all the batches are processed.

To make it easier for the bridge contract to handle acknowledgments, the Router chain will assign an `OutboundBatchNonce` whenever the bridge contract submits an `OutboundBatchRequests`. Eventually, when the bridge contract receives an `OutboundAck`, it can correlate the `OutboundAck` with the previously submitted request using the `OutboundBatchNonce`.

6.4 Batch Atomicity

Router V2 provides support for atomic batch delivery, i.e., for ensuring atomicity across multiple transactions in a batch to save users from the overhead of deploying wrapper contracts to handle multiple function requests. It is important to note that atomicity can only be ensured for batches submitted to a single destination. For atomic execution of transactions, the application bridge contract should create an `OutboundBatchRequest` with flag `IsAtomic` set `true`. Once the batch request is submitted to the Gateway contract, it will verify the signatures and execute all the contract calls in the batch via a proxy contract. If any of the contract calls fail, the proxy contract will revert, thereby ensuring atomicity. In the case of non-atomic requests, errors arising in any of the requests will be handled on the proxy contract to ensure that the transaction doesn't get reverted and all the remaining contract calls are executed.

6.5 Expiry Timestamp

Every outbound request contains an `expiryTimestamp`, which, as the name suggests, determines the latest timestamp by which an outgoing transaction should be executed. If the relayer fails to pick up the transaction by this time, the batch is discarded from the queue. The Gateway contracts will also have a check to ensure that a request submitted by a relayer has an `expiryTimestamp` greater than the `currentTimestamp`. If not, it will revert the transaction and send an acknowledgment back to the bridge contract. Since the application-specific bridge contracts will be able to specify the `expiryTimestamp` in their outbound batch requests, they can keep the `expiryTimestamp` close to the `currentTimestamp` for time-sensitive requests.

6.6 MetaMask Compatibility

Unlike most chains built using the Cosmos SDK, the Router chain will have full-fledged support for MetaMask, one of the most popular non-custodial crypto wallets. This means that users will be able to:

a) Add the Router chain to the list of networks on their MetaMask wallet.

b) Import their Router chain assets and balances to their MetaMask wallet.

c) Review transaction details, including tx data, gas price, and gas limit, and sign transactions directly from their MetaMask wallet while connected to the Router chain.

7 Future Work

7.1 Interoperability with Private Blockchains

Recent years have witnessed increased adoption of private blockchains across various sectors - e-commerce, healthcare, logistics, insurance, and financial services. The proliferation of private blockchains has been accompanied by growing concerns regarding their security. Due to the presence of fewer nodes, private blockchains can be more easily compromised compared to public blockchains. To mitigate this risk, we are planning to build a decentralized communication channel between public and private blockchains on top of the Router chain. Such a channel will allow private blockchains to commit their state proofs onto a public blockchain and thereby leverage the security guarantees of large public blockchains.

7.2 Providing Instant Finality for Transactions on Optimistic Blockchains

To solve for Ethereum's scalability constraints, various optimistic rollups have come to the fore. They seek to minimize transaction costs by batching various blocks of transactions before committing them on Ethereum. Once a batch gets committed on Ethereum, it must go through a challenge period before it can be considered valid. While this challenge period goes on, activities on the optimistic rollup can continue. However, if anyone generates and submits a valid fraud-proof during the challenge period, then the state of the entire optimistic rollup gets rolled back to the state that existed before the fraudulent batch was committed on Ethereum.

Since optimistic rollups, by design, do not have a way to achieve instant finality, there is always a theoretical possibility of transactions getting rolled back. This can be particularly problematic in cases where the optimistic rollup is part of a cross-chain transaction. Imagine a scenario wherein a user sends some funds from an optimistic rollup, Optimism, to another blockchain, Avalanche. Once the transaction is mined on Optimism, the bridge deducts funds from the user's wallet on Optimism and credits them to the user's wallet on Avalanche. Let us assume that after a few hours, someone found and proved that one of the transactions preceding this transaction was fraudulent. Now, Optimism will be rolled back to a state wherein the user's funds are not deducted on Optimism, but since Avalanche has not been rolled back - the user also has funds on Avalanche.

To solve the problem of instant finality for cross-chain contract calls, we will use the Router chain to provide the proof of validity of the rollup chain state committed to Ethereum. The validators of the Router chain will run a verifier of the Optimistic rollup chain. The verifier will construct the state root of the Optimistic rollup batch offline and compare it with the state root of the batch submitted to Ethereum. If the state root matches, the validators will submit an attestation to the Router chain with the batch state root. Once $\frac{2}{3} + 1$ majority of validators attest the state root, it's added to the Router chain state. Following this, an outbound transaction consisting of the attested state root gets relayed to the Gateway contract on Ethereum, thereby providing instant proof of validity. As the validators have verified the state root already, it is guaranteed that a fraud-proof against the batch will not be created. The dApps on the Ethereum chain can query the Gateway contract for the attested state root and instantly proceed with the contract call rather than waiting for the challenge period.

7.3 Orchestrator Optimization using MPC

Even with the various measures to address orchestrator scalability (refer to Section 3.4.3), the voting process can be resource intensive. To reduce the time and cost overheads associated with the current orchestrator voting process while maintaining the same security benchmarks, we will switch to an MPC-based voting mechanism in the coming months. In the new design, a new P2P network will be established amongst the orchestrators to validate all the inbound and outbound requests.

As mentioned in Section 3.4.3, every validator on the Router chain runs an orchestrator instance. In the new model, every validator will also have a corresponding sequence number that is derived from its Router chain address. At any given time, the orchestrator instance associated with the validator will listen to inbound events from all the supported chains. If the event nonce matches the sequence number of the validator, then the orchestrator will initiate an Multi Party Computation (MPC) request on the P2P network along with its signature. Upon receiving the MPC request, other orchestrators on the P2P channel will fetch and validate that specific event. Upon successful validation, the orchestrators will add their own signatures against the request. Upon receiving the required threshold, the request is transmitted to the Router chain as an inbound message. It is important to note that this model is currently in the ideation phase, and most of its mechanics, including its underlying digital signature algorithm, are yet to be finalized.

7.4 Automated Triggers/Scheduler-as-a-Service

One of the most critical features of the Router chain, once implemented, will be the inclusion of a scheduling service that will allow for the automatic execution of specified contract functions without the need for an external entity to trigger that event. To the best of our knowledge, no other blockchain provides an inbuilt service for the time-based or condition-based triggering of contract functions without needing a custom Web 2.0 service. Various use cases can be realized by leveraging this scheduler. For

example, a bridge contract on the Router chain listening to external price feeds can trigger a buy/sell function if the price of the monitored token moves above or falls below a certain threshold.

7.5 Enabling Custom Security Models on the Destination Chain

Security is all-important in a cross-chain infrastructure. As a PoS network built on Tendermint, the Router chain's baseline security model is one of the most robust. However, at Router, we believe that applications leveraging our infrastructure should be able to implement their own safeguards on top of our baseline model if they so require. In fact, we have already alluded to the fact that a modular security mechanism is one of the core facets of Router V2. Applications utilizing Router's middleware flow can already take advantage of this option by coding custom security logic on their Router chain contracts.

As noted in Section 3.5.3.2, we are also working on offering a customizable security layer on external chains for applications looking to implement their security measures on the destination chain instead of on the Router chain (including applications built using CrossTalk). Via this security layer, applications will have the option to configure their security standards based on various parameters, such as the following:

1. **Type of call:** Applications might not want an additional security layer for a read call. However, for a cross-chain write, an application might require a few further checks.
2. **Source chain:** An application might want higher security for transactions originating from specific chains.
3. **Transfer value:** In case of asset transfers, or sequenced asset and instruction transfers, an application might want to place additional safeguards if the ticket size of the transfer is above a certain threshold.
4. **Latency sensitivity:** For transactions that need to be executed in a low-latency environment, applications might not want to include an auxiliary security model.

To implement destination-chain-level security checks, applications will be required to deploy their security requirements as a contract on the external chains. Whenever Router's Gateway contracts receive that application's outbound request from the relayers, they'll query this contract (the address of this contract can be specified as a parameter in the payload). The contract should return a boolean value, allowing the Gateway contract to discern whether or not the request passed the application's security check. If the request passes the security check, the Gateway contract will store it until a permissionless entity executes it by providing the request nonce. If not, the request will be discarded.

7.6 Random Number Generator

In the past few years, randomness has proved to be a key ingredient in multiple Web 3.0 disciplines, ranging from Play-to-Earn games to forms of gambling like prediction markets. Given the intricate mathematics involved, a majority of these applications currently have to build their own randomness source from the ground up, leading to additional overhead. To solve this problem, Router V2 will ship with support for a Random Number Generator (RNG), which is cryptographically secure and can be easily integrated by any dApp on the Router chain. Since a True RNG cannot be established on a blockchain, we will devise a Pseudo RNG that will accept one or more initial values as input, perform mathematical operations on it, and produce pseudo-random determinism sequences as output.

8 Concluding Remarks

As it is with other domains, users are all-important in Web 3.0. As more and more blockchains come up, the user base gets fragmented across them. Even after much innovation in the blockchain interoperability space, cross-chain applications have been unable to make the most of their potential due to the issues plaguing the current crop of cross-chain solutions. Even though low latency, high security, and complete decentralization are necessary conditions in a cross-chain infrastructure, they are

insufficient to combat the current problem. To get the most out of the current applications, what is needed, it seems, is an infrastructure that allows applications to apply custom bridging logic. Towards this end, in Router V2, we are leveraging the Router chain - a Cosmos-based blockchain with an environment supporting the development and execution of CosmWasm smart contracts, as a hub-chain to provide a highly customizable cross-chain infrastructure. This new infrastructure will enable a new wave of dApps to take advantage of cross-chain composability.

References

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, Dec 2008. Accessed: 2022-08-01.

[2] Hashlock. https://en.bitcoin.it/wiki/Hashlock, 2015.

[3] Timelock. https://en.bitcoin.it/wiki/Timelock, 2016.

[4] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. A survey on blockchain interoperability: Past, present, and future trends. *arXiv preprint arXiv:2005.14282*, 2020.

[5] Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, and William Knottenbelt. Xclaim: Trustless, interoperable, cryptocurrency-backed assets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 193–210, 2019.

[6] HashHub Research. Explaining the structure and types of blockchain bridges. https://mirror.xyz/0x8Df126302a7EA75E7013Ec8Ad6bFaC14DD84a5fF/Cxa5L18eGjr8y08VkELx9m61-yl1-h-oTT2z7SjNhuo, Feb 2022.

[7] Rick Delaney. Blockchain bridges explained — how crosschain messaging protocols work. https://www.okx.com/academy/en/blockchain-bridges-explained-how-crosschain-messaging-protocols-work#Decentralized-or-trust-minimized-bridges, Mar 2022.

[8] Dmitriy Berenzon. Blockchain bridges: Building networks of cryptonetworks. https://medium.com/1kxnetwork/blockchain-bridges-5db6afac44f8, Sep 2021.

[9] Léonard LYS, Arthur Micoulet, and Maria Potop-Butucaru. R-SWAP: Relay based atomic cross-chain swap protocol. Research report, Sorbonne Université, April 2021.

[10] Billy Rennekamp. Gravity Bridge. https://github.com/cosmos/gravity-bridge. Accessed: 2022-01-13.

[11] VMware. Rabbitmq: Messaging that just works. https://www.rabbitmq.com/. Accessed: 2022-11-16.

[12] Interchain Foundation. Secure P2P. https://docs.tendermint.com/v0.33/tendermint-core/secure-p2p.html. Accessed: 2022-11-16.

[13] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 124–142. Springer, 2011.

[14] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[15] Hyperlane Team. Sovereign Consensus. https://docs.hyperlane.xyz/docs/protocol/sovereign-consensus. Accessed: 2022-02-08.

[16] cgewecke. hardhat-gas-reporter. https://github.com/cgewecke/hardhat-gas-reporter#readme. Accessed: 2022-02-08.

Appendices

A Redelegation Mechanism for Failed Requests

When the inbound module tries to execute an inbound or outbound acknowledgment request on the Router chain, the request may fail to execute on the middleware contract for various reasons. As the corresponding source chain action has already taken place, there should be a way to allow the applications to try and execute the failed request again. The Router chain has a redelegation mechanism to enable this functionality for failed inbound and outbound acknowledgment requests. Anyone can trigger this mechanism by sending a `redelegate` transaction to the inbound module. However, unlike the standard execution flow, the middleware contract won't pay the fee for executing the transaction. Instead, the sender of the `redelegate` transaction will have to pay the fee for executing the transaction on the middleware contract.

B Cross-chain Read Request

B.1 Creating and Sending a Cross-chain Read Request

To create a cross-chain read request, one needs to call the `readQueryToDest()` function on Router's Gateway contract with the following parameters:

1) contractCalls

The `contractCalls` parameter includes an array of payloads (read function calls) and an array of contract addresses on the destination chain where the respective payload should be sent. The payload will include the ABI-encoded function calls that are to be executed on the destination chain.

2) requestArgs

This parameter includes a struct comprising of the following subparameters:

- `isAtomicCalls` - Since users can use the `readQueryToDest()` function to send multiple cross-chain calls in one go, the `isAtomicCalls` boolean value ensures whether the calls are atomic.
  - If this variable is set to **true**, either all the contract calls will be executed on the destination chain, or none of them will be executed.
  - If this variable is set to **false**, then even if some of the contract calls fail on the destination chain, other calls won't be affected.
- `expiryTimestamp` - The timestamp by which your cross-chain call will expire. If your call is not executed on the destination chain by this time, it will be reverted. If you don't want any expiry timestamp, pass `type(uint64).max` as the `expiryTimestamp`.
- `feePayer` - The `feePayer` parameter specifies the address on the Router chain from which the cross-chain fee will be deducted. It can be either one of the three: (a) the user address, (b) the application contract address, or (c) `NONE`. If the `feePayer` is set to `NONE`, then any entity on the Router chain can act as the `feePayer`.

3) destinationChainParams

In this parameter, users can specify the gas they are willing to pay to execute their read request, along with the details of the destination chain where the request needs to be executed. This parameter needs to be passed as a struct containing the following subparameters:

- **gasLimit:** The gas limit required to execute the cross-chain read request on the destination chain.
- **gasPrice:** The gas price that the user is willing to pay for the execution of the request.

- **destChainId:** Chain ID of the destination chain in string format.
- **destChainType:** This represents the type of chain. The values for chain types are given in table 1.

4) ackGasParams

Once the cross-chain read request is executed on the destination chain, the requested data is sent back to the source chain as an acknowledgment. To handle this acknowledgment, users need to include a callback function (discussed in Appendix B.2). The ackGasParams parameter includes the `gasLimit` and `gasPrice` required to execute the callback function on the source chain when the acknowledgment is received. The gas limit depends on the complexity of the callback function, and the gas price depends on the source chain congestion. The gas limit can easily be calculated using the hardhat-gas-reporter plugin [16]. For the gas price, you can use web3/ethers library's `provider.getGasPrice()` function.

B.2 Handling the Acknowledgment on the Source Chain

Once the read request is executed on the destination chain, the requested data is sent along with an acknowledgment to the source chain. To handle the acknowledgment, developers need to include a `handleCrossTalkAck()` function in their contract. This function will have three parameters, namely:

1) eventIdentifier

This is the same nonce you receive while calling the `readQueryToDest()` function on the source chain Gateway contract. Using this nonce, you can verify whether a particular request was executed on the destination chain.

2) execFlags

Since multiple payloads can be sent to multiple contract addresses on the destination chain, the `execFlags` is an array of boolean values that tells you the status of the individual calls.

3) execData

The `execData` parameter is an array of bytes that provides the return values from every read call included in the read request. Based on the application's requirement, this data can be decoded and processed on the source chain.

C Batched Cross-chain NFT Minting via the Router Chain

Step 1: A user calls the NFT contract for a cross-chain mint and buy. The NFT contract takes the funds from the user's account and deposits them in the contract. The amount of funds to be taken is already set on the source contract based on the NFT minting price. The application contract then calls the `requestToRouter()` function on the Router Gateway contract with the relevant parameters.

Step 2: The Router Gateway contract emits an event that is listened to by the orchestrators on the Router chain.

Step 3: After validating the event and deducting the fee for the inbound request, the inbound module passes the event to the bridge contract on the Router chain.

Step 4: Upon receiving the cross-chain mint and buy request, the bridge contract will store the request and increment the value of a counter by 1. Every time, after incrementing the counter, the bridge contract will check if it has received the required number of requests (say N). If not, it will do nothing further. If it has, the bridge contract will parse the payload of all the stored requests into a single payload with all the user addresses and generate an outbound request for minting N NFTs on the destination chain.

Step 5: After the transaction initiated by the bridge contract is mined on the Router chain, the outbound module picks it up.

Step 6: Once the orchestrators sign the outbound request, the relayer picks the event and forwards it to the Gateway contract on the destination chain.

Step 7: The Gateway contract on the destination chain calls the `handleRequestFromRouter()` function on the NFT contract on the destination chain.

Step 8: The NFT contract on the destination chain will then mint the NFTs on the destination chain to all the user addresses specified in the payload.