GAS DISTRIBUTION PROTOCOL

FOR PERMISSIONED PUBLIC ETHEREUM-BASED NETWORKS

GAS DISTRIBUTION PROTOCOL

FOR PERMISSIONED PUBLIC ETHEREUM-BASED NETWORKS

Authors:

Marcos Allende, Technical Leader of LACChain and IT Specialist in Blockchain, Digital Assets, Quantum Technologies, and SSI at IDB, USA

Adrian Pareja, Blockchain Lead Architect, LACChain, Perú

Supervisors:

Alejandro Pardo, Leader of LACChain and Principal Specialist at IDB Lab, USA

Mariana Gutierrez, Lead Specialist at IDB, USA

Design:

.Puntoaparte Editores

The open-source implementation of the GAS Distribution Protocol presented in this paper, as well as tutorials and examples for its use, can be found at https://github.com/LACNetNetworks/gas-management. LACNet is the Underlying Orchestration Vehicle that orchestrates the LACChain Blockchain Networks developed by the LACChain Alliance.

TABLE OF CONTENTS

1

CONTEXT

The throughput of blockchain networks is inherently limited[1]. This is due to several reasons. First, blockchain networks are decentralized, which leads to time constraints for replicating transactions across the network and executing these transactions by the nodes. Second, nodes maintain a copy of the entire transactional history, which leads to space constraints because the size of the history needs to be controlled and limited in order to stay manageable. For this reason, networks limit the size and execution capacity that blocks and nodes can assume.

Throughput limitations lead to a "supply and demand" challenge, which leads to the following questions: How can we manage situations in which the amount of individuals or entities that want to use a blockchain network exceed the network's ability to support them? Permissionless networks have addressed this with dynamic transaction-fee mechanisms. The more network space and computational capacity a person or entity requires for their registries/transactions, the more they have to pay. Consequently, the more users are using the network, the more expensive it becomes for each of them to use it. It is a classic supply and demand approach: transaction fees go up until supply and demand curves meet and equilibrium is reached.

Because of that, there is a big problem with transaction-fee-based networks which is that they quickly become very expensive. The more successful they are in attracting users, the more unaffordable they become for these users. For example, Bitcoin transaction fee average oscillated between $2 and $63 over the year 2021, and Ethereum averaged transaction fees between $17 and $63 in the fourth quarter of 2021[2], respectively. This is why transaction-fee-based blockchain networks can hardly be an option for government and enterprise use cases that generate large amounts of transactions. This becomes even more unfeasible when taking into account fee volatility, which makes forecasts for budget allocation very difficult.

1 The number of transactions thar can be processed per second.

2 https://bitinfocharts.com/

As opposed to permissionless networks, there are also permissioned blockchain networks which among other differences have usually erased transaction fees. These networks have been seen as the way to go for high transactional use cases. In the LACChain Framework for Permissioned Public Networks we discussed how a permissioned public network can meet the benefits of permissionless networks and be at the same better for high transactional applications.[3]

Erasing transaction fees is definitely good for users, as they can send transactions for free or with a fixed membership cost. However, two main trade-offs must be addressed. One is what are the incentives for the entities participating and looking after the network's welness if nobody gets rewarded with transaction fees (assuming that there is neither a native token or cryptocurrency serving this purpose). This is addressed in the LACChain Framework mentioned above. The second and probably most critical issue is how is the access and use of the network (i.e., the distribution of resources) managed, in a way that the network does not collapse due to the fact that everyone could in principle send as many transactions as they want for free. This paper describes the open-source solution developed by LACNet to address this issue in Ethereum-based networks specifically, the solution has been implemented, tested, and evaluated in networks based on the Hyperlergder Besu protocol.

3 M. Allende. (2021). LACChain Framework for Permissioned Public Blockchain Networks. Inter-American Development Bank. Retrieved from https://publications.iadb.org/en/lacchain-framework-permissioned-public-blockchain-networks-blockchain-technology-blockchain

2

SOLUTION

In order to manage the use of GAS in Ethereum-based permissioned public blockchain networks that have erased transaction fees, LACNet has developed the first GAS distribution protocol of its kind. LACNet's GAS distribution protocol presented in this document is aligned with the LACChain Framework for Permissioned Public Networks.

Often, GAS distribution protocols for blockchain networks consist of a faucet from which account owners can request GAS. This has notable limitations. One limitation is that it might be difficult or impossible to prevent permissioned entities from transferring GAS to non-permissioned entities which, in a permissioned network, might enable non-permissioned users to broadcast transactions. Another limitation is that it might be difficult or impossible to prevent a permissioned or non-permissioned user to accumulate GAS and use it to provoke a DoS attack and collapse the network by using all that GAS in a short period of time.

LACNet's GAS distribution protocol for Ethereum-based Networks consists of a set of smart contracts that, among other things:

- Assign GAS per block to the accounts associated with permissioned writer nodes in a dynamic way based on how stressed the network is at each time point (the more stressed it is, the less GAS is distributed). GAS is not distributed or made available directly to end-user accounts.
- Serve as a proxy to evaluate every transaction sent to the network and check that (i) each transaction has been signed by a permissioned writer node and (ii) the writer node has enough GAS left for registering that transaction in the current block.

This protocol gives the power and responsibility to writer nodes rather than end users, as writer nodes will have access to the GAS that allows to broadcast transactions. Therefore, writer nodes decide which end-users they allow to use their node and also their GAS to send transactions to the blockchain. Writer nodes are required to intercept each transaction they receive from end-users or client applications and wrap them into meta-transactions they then broadcast to the network. These meta-transactions contain the original transaction as well as the signature of the permissioned writer node that is broadcasting the transaction to the network.[4]

4 This solution intentionally requires that writer nodes sign the transactions they send to the network in order to establish an accountability framework that makes them legally accountable for those transactions. Currently, Ethereum-based network transactions are signed only by anonymous end-users and it is not possible to track which nodes introduced or replicated a specific transaction. Thus, it is uncertain who is responsible for the information that is registered in a blockchain network. Therefore, it is possible that all nodes in the network are blamed for some data they did not introduce nor validate. However, it is immutably recorded in the copy that they maintain locally. For a more detailed discussion on this topic, read the LACChain Framework for Permissioned Public Blockchain Networks.

Instead of sending the transactions to the recipient contract indicated by the original end-user sender, the writer node is required to send the transactions to a set of proxy smart contracts. In general terms, these smart contracts verify that the writer node has enough GAS and the signature is valid.

2.1. Architecture

A high-level architecture diagram for the implementation using Hyperledger Besu as the underlying protocol is presented in Figure 1. The solution has two main components:

1. The smart contracts that contain the logic described in the previous section.
2. The writer node back-end components necessary to generate, sign, and send the meta-transactions to the proxy smart contracts.

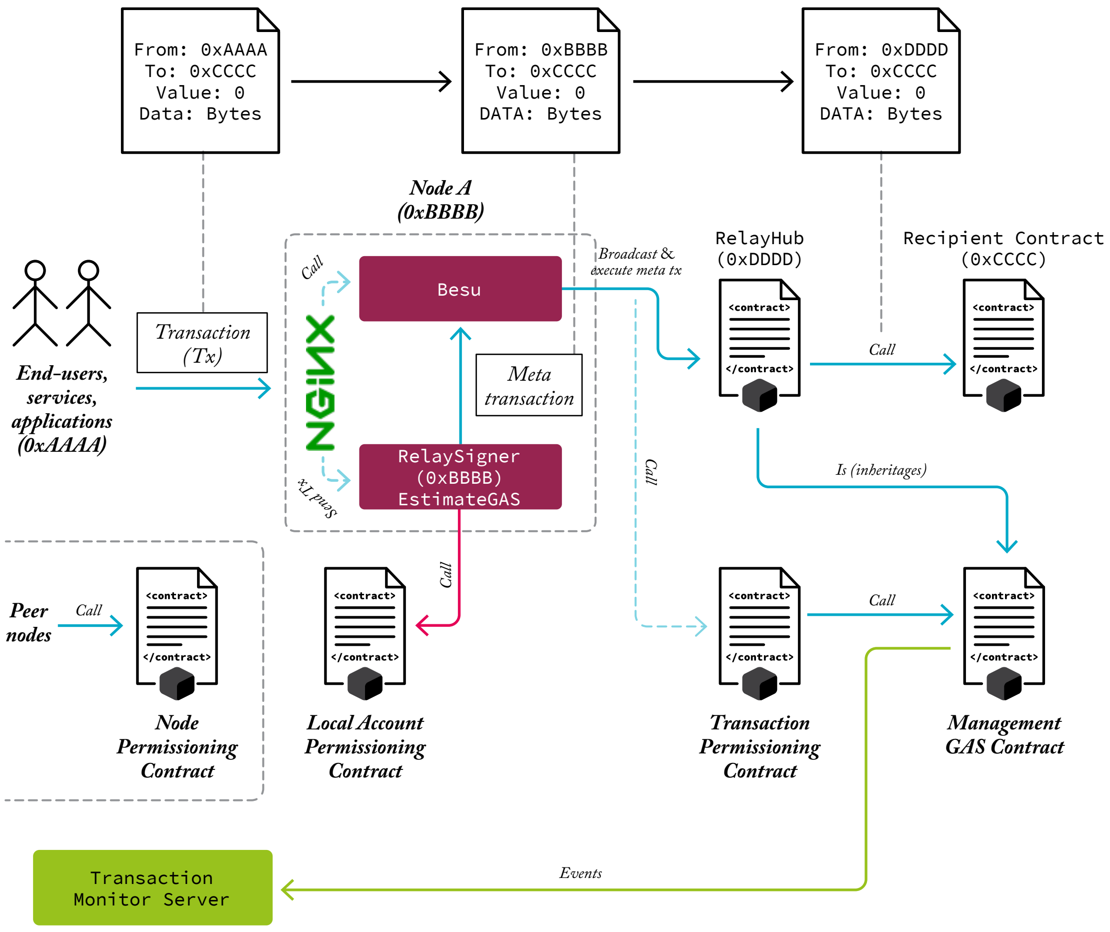

Figure 1. High-level architecture diagram.

2.1.1. Smart Contracts

The proposed solution involves six smart contracts: the Transaction Permissioning Smart Contract, the Node Permission Smart Contract, the RelayHub Smart Contract, the GAS Management Smart Contract, the Recipient Smart Contract, and the Local Account Permissioning Smart Contract. These smart contracts play two roles: 1) realize the GAS distribution protocol perse by carrying out GAS accounting and distribution, and 2) verify the transaction permissioning (both at the network and writer node levels).

- **Transaction Permissioning Smart Contract:** The role of this contract is to only allow transactions that meet the network requirements (see Section 2.2 for more information):

    - Transactions must be directed to the RelayHub smart contract.
    - Transaction GAS price must be 0.
    - The GAS limit of the transaction must be sufficient enough to execute the RelayHub smart contract.
    - The writer node that broadcasts the meta-transaction must be the one indicated by the end-user or application in the original transaction (as explained in Section 2.3) and the expiration time for the transaction set by the end-user must be respected.

Transactions that do not satisfy these rules will be rejected by the network.

- **Node Permissioning Smart Contract:** This contract contains the list of permissioned nodes and defines the rules for TCP connections between nodes according to the topology and the routing rules explained in the LACChain Blockchain Framework for Permissioned Public Networks. Nodes that are not registered in this contract cannot connect to the network. Before two nodes establish a connection, they must verify against this contract whether the connection is allowed.
- **Relay Hub Smart Contract:** This contract is based on the EIP 1077.[5] The contract receives a meta-transaction (the transaction generated and signed by the writer node containing the original transaction sent by the sender) whose data field is decoded by RLP to obtain the parameters of the original transaction, which are the address of the original sender, GAS limit, nonce, original data sent, and destination address. The RelayHub Smart Contract validates the signer's signature against the original message to guarantee that the message came from the signer. Then, it verifies that the node has not reached the GAS limit for that block. If everything is correct, the transaction is forwarded to the recipient contract indicated by the original sender (if there is one) or creates a new smart contract (if that was the intention).

5 REF https://github.com/status-im/EIPs/blob/update-1077/EIPS/eip-1077.md

- **GAS Management Smart Contract:** Keeps track of how much GAS each permissioned node (the address or addresses associated to it) has per block, and registers how much GAS is being consumed in the last N blocks. It also sets the GAS limit that each node can consume depending on the network's degree of stress in the last N blocks.
- **Recipient Smart Contract:** This contract is the final destination of the transaction sent by a user or client application (unless the transaction was not intended to call an existing smart contract, for instance, because it was intended to deploy a new contract). This contract executes the function chosen with the parameters sent in the original transaction.
- **Local Account Permissioning Contract:** Writer nodes can deploy, maintain, and customize a Local Account Permissioning Contract, which is a permissioning layer that writer node operators can use to filter (by whitelisting) the reliable addresses (senders) they allow to send transactions to the network through their writer nodes under rules customized by the writer node operator such that transactions that do not meet the requirements (presented in Section 2.2) are broadcasted to the network. This allows for each writer node operator to define its own rules for transactions to be broadcasted to the network. For writer nodes exposed to external users, services, and applications, this is extremely important because writer node operators are accountable for transactions that their node broadcasts, despite who the original sender is.

When a new node is added to the network, the Permissioning Committee registers it into the Node Permissioning Smart Contract. Next, the organization senders' addresses, whose node was previously added to the Node Permissioning Smart Contract, are added to the Transaction Permissioning Smart Contract. Automatically, the Transaction Permissioning Smart Contract invokes the GAS Management Contract which sets a GAS limit for the new node addresses and recalculates the GAS limit for the existing nodes.

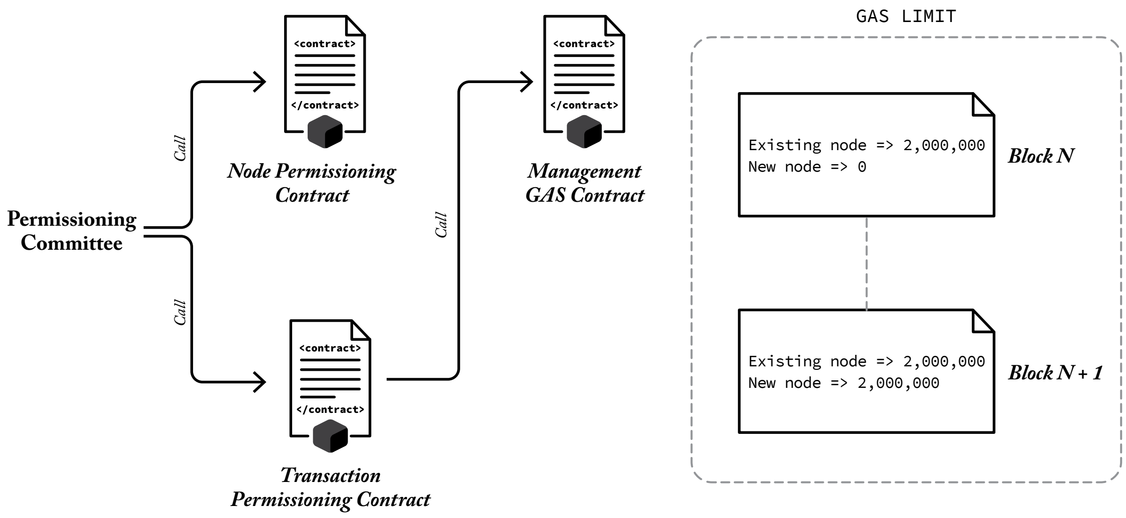

Figure 2. Representation of the flow for adding a new node and sender address.

Each time a transaction is sent to the network, nodes check it against the Transaction Permissioning Smart Contract. First, writers check the transaction coming from the off-chain service or application, then boots check the transaction coming from the writer, and finally validators check the transaction coming from the boots. If the transaction does not satisfy the requirements (introduced above and covered in detail in Section 2.2), the transaction will be rejected (i.e., not executed and erased from the transaction pool).

The validator nodes of the network, which are responsible for generating blocks, have the Transaction Permissioning configuration set, which makes them check every transaction they receive against the Transaction Permissioning Contract before they are executed in the network. It is worth mentioning that if a validator node becomes malicious by modifying the configuration of its node to accept invalid transactions into the network, this validator node will be banned by the other validators. For an invalid transaction to be introduced into a block, ⅔ +1 validator nodes need to be corrupted. If validator node operators tried such an attack, they would incur a legal penalty because validator nodes are committed contractually to not attack the network.

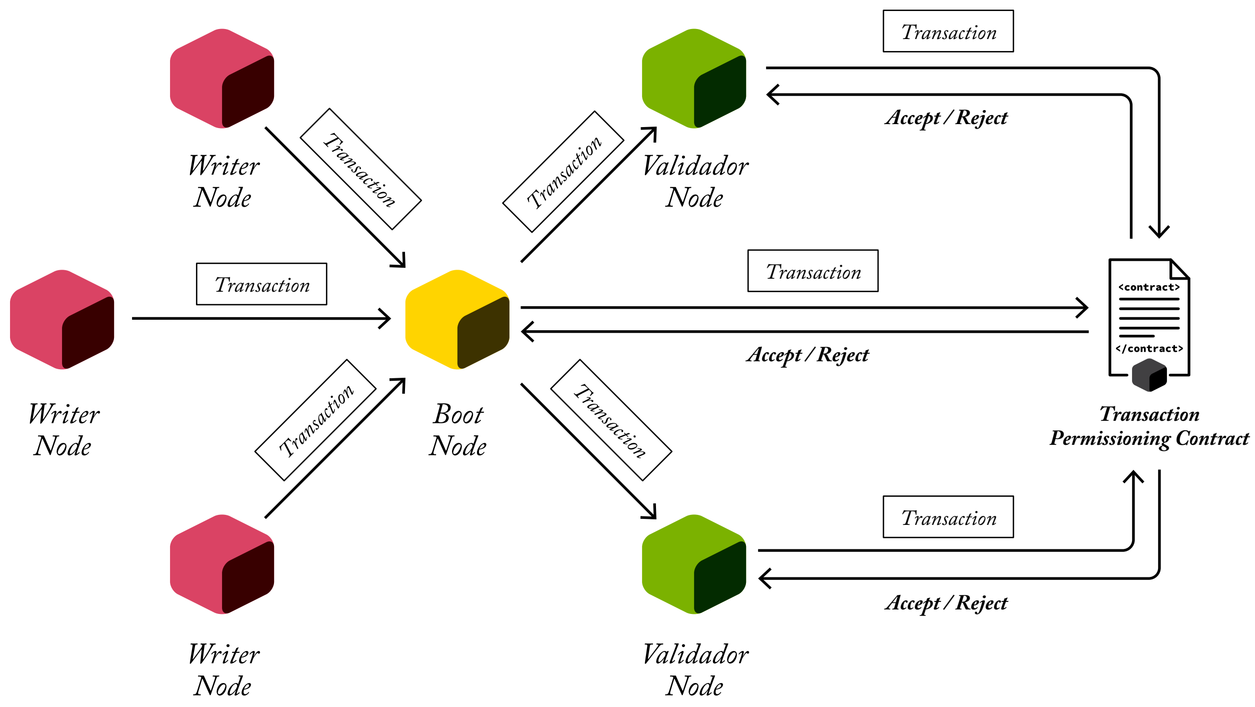

Figure 3. High level flow of the verification of transactions against the Transaction Permissioning Smart Contract by Validator Nodes

When the transaction meets the validity conditions, validator nodes execute the transaction against the RelayHub contract, where the nonce is verified to be greater than the previous one in order to guarantee that the transaction is unique and not duplicated. Before the transaction is forwarded to the Recipient Smart Contract (or creates a new smart contract itself), it is verified that the node has not consumed all the GAS assigned to it for that block GAS Management Contract. If the transaction GAS limit is lower than the amount of GAS remaining for the writer node in the current block, the transaction is executed and the amount of GAS used is discounted in the GAS Management Contract. If the transaction execution cannot be completed, the amount of GAS consumed is discounted and a "not enough GAS" error event is emitted.

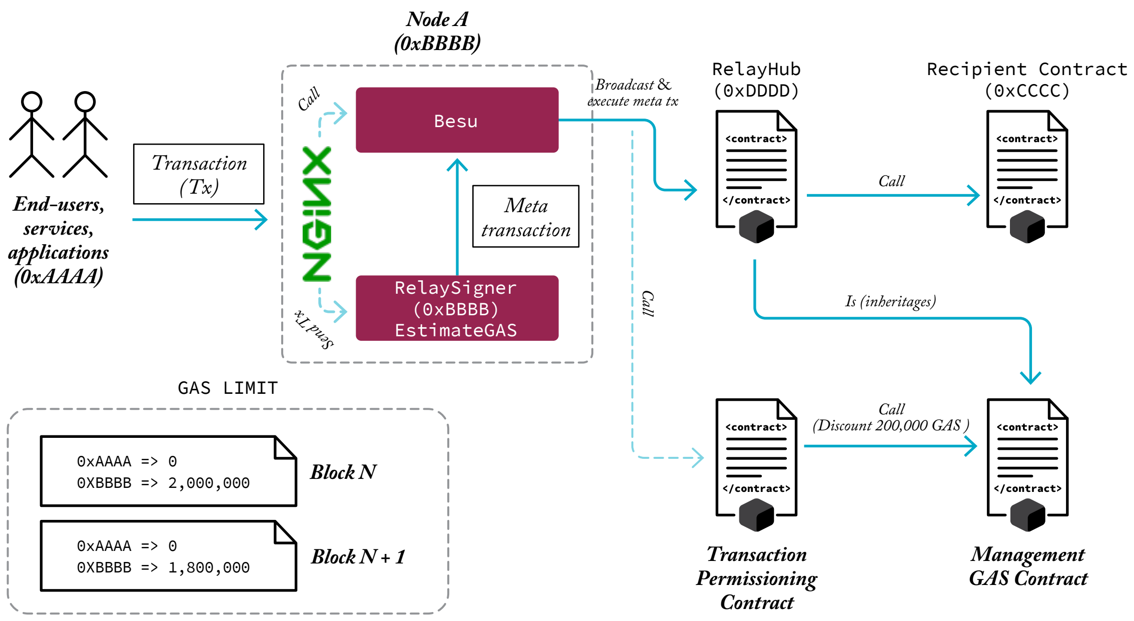

Figure 4. Representation of the flow for sending a transaction that consumes 200,000 GAS.

The RelayHub emits an event when a bad or doubtful transaction is sent, including:

- The nonce sent is repeated or lower than the one registered in the contract for that user's address.
- The GAS limit sent in the transaction exceeds the block logical GAS limit set in the contract (it is not the same as the genesis GAS limit).
- The transaction is sent to a contract that has an empty code.
- The transaction tries to deploy an empty code contract.
- The user transaction has an invalid signature.
- The GAS limit of the transaction exceeds the amount of GAS left available for the writer node that is broadcasting the transaction.

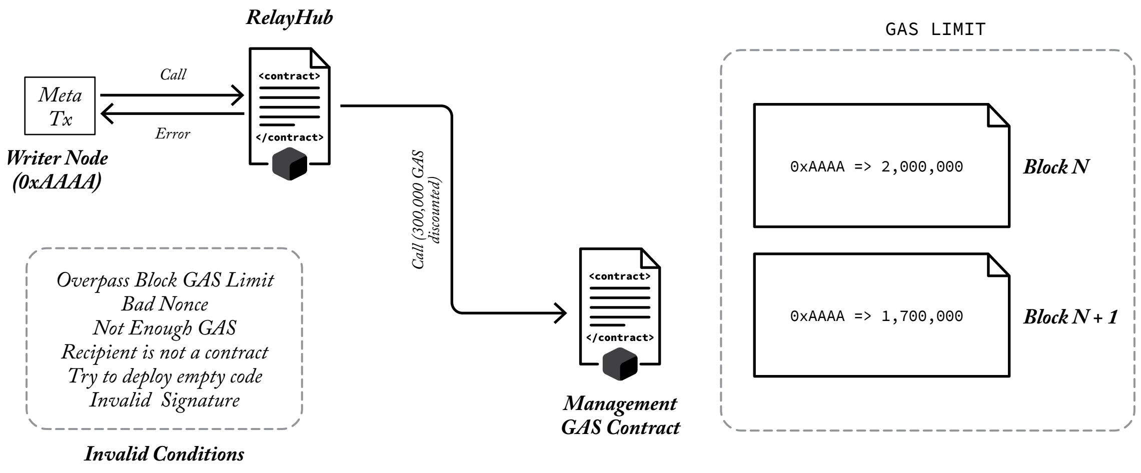

Figure 5. Representation of the flow for sending an invalid transaction.

When an organization that owns a writer node exceeds the GAS limit assigned to them in a block, it will be marked as an "exceeded GAS limit" error and the organization responsible for the writer node is automatically removed from the Transaction Permissioning Contract, which prevents it from sending transactions. When a node is banned, the Permissioning Committee reaches out to the node operator to clarify the situation and add the node back into the Transaction Permissioning Contract.

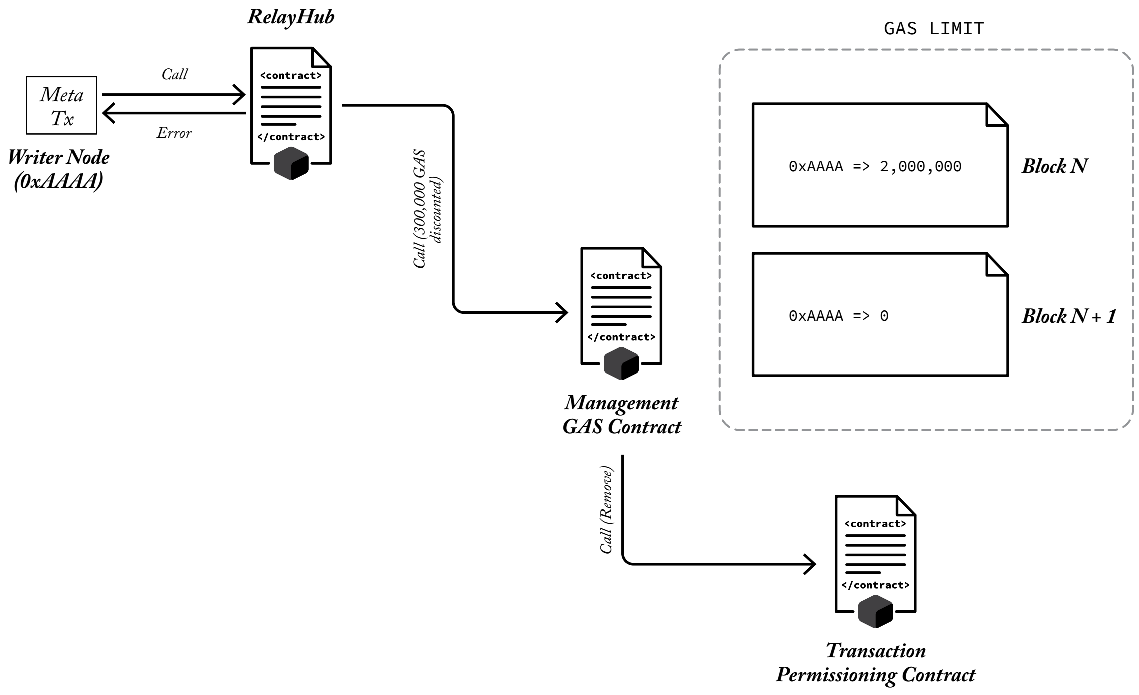

Figure 6. Representation of the flow for exceeding node GAS limit.

When a private transaction is sent by the end-user or application to the writer node, it goes through the RelaySigner, which calls the GAS Management Smart Contract to discount a fixed amount of 25000. Then, it redirects the transaction to the Besu process, and this is communicated with the Private Transaction Manager (PTM) service, which shares the transaction with the PTM nodes indicated trough the private channel.

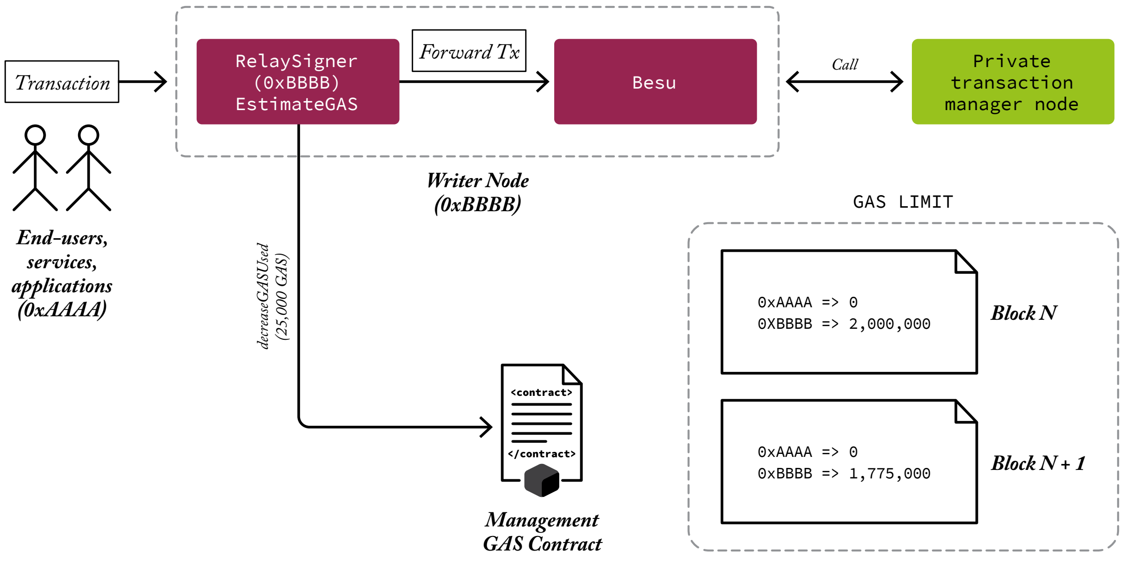

Figure 7. Representation of the flow for sending a private transaction.

2.1.2. Writer Node

The LACNet writer nodes consist of a Besu node configured as a writer (according to the LACChain topology) and is provided with a nginx and a RelaySigner. We have designed the GAS management solution to be as transparent as possible to end-users, meaning that services and applications on top can send transactions almost as if the proxy smart contracts did not exist. It is the writer node that turns the transaction in a meta-transaction that meets the GAS and node-signature requirements. Writer nodes can also deploy a Local Account Permissioning Contract as a filter to avoid unknown accounts from using their node, explained in Section 2.1.1.

- **Nginx:** Acts as a reverse proxy and SSL termination (terminates HTTPS traffic from clients). It accepts all transactions to the node and forwards them to the RelaySigner. Direct queries to the node do not go through the RelaySigner; nginx redirects them to the Besu RPC port.

- **RelaySigner:** Evaluates the type of message from an application (service or end-user) that arrives to the node. If it is a raw transaction (RLP unsigned from final user or application), then the RelaySigner decomposes it, builds a meta transaction, signs it, and sends it to the network. If it is another type of message, such as contract calls, the RelaySigner gets the recipient, number of connected nodes, block information or transaction, and forwards the message directly to the Besu component.
- **Besu:** A client of the LACNet network, with which the p2p connection is maintained. It broadcasts transactions to the other nodes of the network. This component receives meta-transactions from the RelaySigner to broadcast to network.

2.2. Requirements for Transactions to Be Valid

The node back-end components described in Section 2.1 allow writer nodes to send only valid transactions to the network, and they will be banned if they decide to manipulate these components such that invalid transactions are sent. Transactions that do not meet the requirements detailed below are rejected by all the non-corrupted nodes after they check them against the Transaction Permissioning Contract. The Transaction Permissioning Contract is a security layer of rules to determine whether a transaction should be executed and included into a block (see Section 2.1.1). As explained in Section 2.2.1, every node that listens to a transaction check that it is a valid one before replicating it. Therefore, it is first the writer's check followed by the boot's check, and afterwards by the validator's check.

- The destination of the transaction must be the RelayHub (relay contract), through which all transactions must go through.
- The GAS limit set for the transaction must cover the RelayHub execution (fixed and variable parts). The variable part of the GAS limit depends on the amount of bytes sent in the transaction, which guarantees the execution of the RelayHub contract and avoids exceptions for lack of GAS (an out of GAS exception).
- The GAS price sent in the transaction must be 0, because in networks orchestrated by LACNet the GAS price is always 0 (see Section 2).
- In order to guarantee that the transactions sent by end-users or services and applications on top are not altered by writer nodes or malicious parties, the original sender is required to add two additional parameters to the function parameters of the destination contract: the "nodeAddress" and the "expiration". Doing this guarantees that a transaction sent by an end-user cannot be re-sent by any node other than the one selected by the end-user and the transaction is not executed after the expiration time indicated by the end user.

- The transaction sender must be an address registered by the organization running a writer node in the Transaction Permissioning Contract, which prevents anyone from sending transactions to the network. Additionally, transactions must be signed with the private key of the organization node's registered address, which must also match the nodeAdress selected by the end user to broadcast its transaction. Only permissioned writers can send transactions to the network.

2.3. GAS Distribution Algorithm

The frequency with which the new GAS limit is set for each participating writer node of the network is every N blocks. This parameter is configurable by the Permissioning Committee. In order to determine the GAS limit for each node per block, the Permissioning Committee establishes the maximum amount of GAS that the network can process per block without suffering delays in the block generation, G_T (see Section 3).

The type of membership the node has purchased determines the amount of GAS per block the node has access to. There could be defined any number of memberships such as the following:

- **Basic membership:** The node GAS limit is set to X per block.
- **Standard membership:** The node GAS limit is set to Y per block.
- **Premium membership:** There is not a pre-defined GAS limit. The GAS available per block minus the GAS reserved for basic and standard nodes is distributed equally among all the premium nodes.

Table 1 gives the ideal amount of GAS consumed on average by different typical transactions in the networks orchestrated by LACNet.

Type of transaction	Estimated average amount GAS consumed
Change of simple attribute type Uint256	120K
Notarization (register a hash)	140K
Token transfer	145K
Register a verifiable credential	275K
Deploy ERC20 smart contract	1M
Deploy complex smart contract	3M

Table 1. Estimated average amount of GAS consumed by frequent transactions.

The idea of "reserving GAS" can lead to scenarios where some nodes might not be using all their GAS per block whereas others might need more. For this reason, this GAS distribution protocol is very flexible and attempts to maximize the amount of GAS per block made available to each premium writer node. This is achieved by updating the GAS limit every Y blocks for premium nodes according to how much the network is being used by all the permissioned nodes.

The amount of GAS available for each premium writer node per block changes dynamically depending on how much GAS all the writer nodes have used in the last N blocks (this parameter is adjustable). The higher the amount of GAS used by the writer nodes, the lower the GAS limit will be. If all the nodes (basic, standard, and premium) have reached the GAS limit available to them per block, then each premium node will have a GAS limit per block equal to the total amount of GAS minus the GAS reserved for basic and standard, divided equally by the number of premium nodes. If writer nodes have not used all the GAS assign to them, premium nodes are allowed to spend up to 5 times the previous amount up to 50% of the total GAS available for all nodes-.

The formula to calculate the GAS limit per block is as follows:

- N_B **:** Number of writer nodes with Basic membership package.
- N_S: Number of writer nodes with Standard membership package.
- N_P: Number of writer nodes with Premium membership package.
- G_T: Amount of GAS per block that the network can process without operating under stress.
- G_B: GAS limit per block for writer nodes with Basic membership package.
- G_S: GAS limit per block for writer nodes with Standard membership package.
- G_P: GAS limit per block for writer nodes with Premium membership package.
- ϑ : Fraction of consumed on average in the last 10 minutes by all the writer nodes.

$$G_P = \left(-5 \frac{\theta}{G_T - (G_B N_B + G_S N_S)} + 6 \right) \frac{G_T - (G_B N_B + G_S N_S)}{N_P} \quad \text{up to a maximum of } \mathbf{0.5} G_T$$

This formula leads to the following scenarios:

$\vartheta = 0.2$	$G_P = 5 \dfrac{G_T - (G_B N_B + G_S N_S)}{N_P}$
$\vartheta = 0.4$	$G_P = 4 \dfrac{G_T - (G_B N_B + G_S N_S)}{N_P}$
$\vartheta = 0.6$	$G_P = 3 \dfrac{G_T - (G_B N_B + G_S N_S)}{N_P}$
$\vartheta = 0.8$	$G_P = 2 \dfrac{G_T - (G_B N_B + G_S N_S)}{N_P}$
$\vartheta = 1$	$G_P = \dfrac{G_T - (G_B N_B + G_S N_S)}{N_P}$

Table 2. Relationship between usage of the network and GAS limit for premium writer nodes.

Let's explore the formula with an example. Let us suppose that the amount of GAS per block that the network can process without operating under stress (G_T) is 150M. Let's also set the GAS limit for basic nodes in 0.5M and the GAS limit for standard nodes in 1.5M. In a network with 20 basic, 20 standard, and 20 premium nodes, basic nodes would have access to a maximum of (0.5M GAS limit x 20 nodes) 10M of GAS, and standard nodes would have access to a maximum of (1.5M GAS limit x 20 nodes) 30M of GAS. The 20 premium nodes would then have access to a maximum of (150M GAS total – 10M GAS for basic nodes – 30M GAS for standard nodes) 110M of GAS to be equally used among them, which is equal to a GAS limit per premium node of 5.5M GAS per block. But this is the case *only* if these 60 nodes are using all the GAS they have available (thus the parameter ϑ is equal to 1). If, conversely, the under 20% of the network network is used (the 150M of GAS per block), then basic and standard nodes would keep their 0.5M and 1.5M GAS limit per block, respectively, but premium nodes have up to 5 times higher GAS limit, which in this example, would mean (5.5M GAS limit x 5) 22.5M GAS limit per block. Appendix A presents results of stress tests performed in the LACNet Networks using the GAS distribution mechanism. Appendix B analyzes the monthly cost in Ether and USD of this amount of GAS in the Ethereum Mainnet.

3

POTENTIAL ATTACKS TO THE NETWORK AND MITIGATIONS

As explained in Section 1, the GAS distribution protocol is a layer designed and developed by LACNet to prevent DoS attacks and require writer nodes to sign the transactions they broadcast to the network so they can be made accountable for them. As in every GAS distribution protocol, there are always potential attacks to be carried out by malicious node operators or end-users. It is essential to be able to identify these attacks in order to mitigate them. With the help of Coinspect, the LACNet team has developed protocols to mitigate all potential following attacks from an end-user/application/end-service with access to a writer node or from a malicious writer node to the network.

By attacks to the network we are referring to any behavior than can impact the network negatively, including sending invalid transactions, attempting to modify Admin Smart Contracts (such as the Node Permissioning Contract, the RelyHub, or the Transaction Permissioning Contract) without permission, allowing non-permissioned nodes to broadcast transactions, attempting to use more GAS than the GAS limit available, attempting to attack other writer nodes or end-users by forwarding or overwriting pending transactions, or trying to overload the transaction pool of other nodes with transactions that will not be executed for different reasons. LACNet, as an orchestration vehicle, is responsible for ensuring that all these potential attacks are prevented or mitigated before they can impact the network.

3.1. DoS Attack by Exceeding GAS Limit

The first and most basic attack that any malicious writer node operator or end-user/application/service with access to a writer node can attempt is sending transactions that exceed the GAS limit assigned for that writer node in that block, which could lead to a DoS attack. Writer nodes would need to modify their signer in order to perform this attack because the signer is set by default to refuse sending transactions that overpass the GAS remaining in each block (see Section 2.1.2).

The GAS distribution protocol prevents the attack by banning the writer node as soon as the first transaction that exceeds the GAS limit is broadcasted. This ban consists of removing the writer node from the Transaction Permissioning Contract so new transactions will automatically be rejected by validator nodes. To be added back into the Transaction Permissioning Contract, the writer node operator needs to clarify the potential DoS attempt to the Permissioning Committee.

3.2. Transactions with Lower GAS Limit than Required

As the base technology of the LACNet Besu network is Ethereum, every transaction executed by the nodes involves a computational cost, which is measured in units of GAS. When a transaction is sent with insufficient GAS, the transaction will not be fully executed by the validators. This releases an out of GAS (OOG) exception indicating that there was not enough GAS to complete the full execution of the transaction.

A malicious end-user could send multiple transactions with a GAS limit lower than what the transaction actually requires for execution, thereby attempting to generate the OOG exception. When the exception is launched, it does not execute the step of GAS reduction in the GAS Management Smart Contract. This makes it such that a malicious end-user or node operator could evade the accounting of how much GAS the node has consumed in that block. This situation could incur a ban for the writer node operator if it leads to the writer node attempting to exceed its GAS limit.

This potential attack is prevented with a prior check by validator nodes against the Transaction Permissioning Contract that discards these types of transactions before they are executed. Prior to the execution of the transaction by validator nodes, validator nodes check against the Transaction Permissioning Contract to confirm that the GAS limit of the transaction, based on the amount of bytes sent, is higher than what will be required to execute the RelayHub contract. By doing this, we guarantee that the amount of GAS is sufficient enough to complete the execution of the RelayHub contract and ensure we account for the GAS used by the node that sent the transaction, as well as the OOG exception. In case the GAS limit sent in the transaction does not cover the RelayHub execution, the transaction is rejected prior to the execution, and thus, no computation is involved.

3.3. Setting the RelayHub as the Final Destination for a Transaction

All transactions must go through the RelayHub contract before they reach their destination contract. A malicious user may want to re-enter the RelayHub contract with the intention of consuming GAS in a block without going through the accounting of the GAS protocol. The malicious user sends a transaction in which the target contract, after going through the RelayHub, becomes the RelayHub itself. This potential attack is prevented by the RelayHub by not allowing re-entry. The RelayHub would return an error and the GAS consumed by the transaction is discounted to the writer node.

3.4. Trying to Modify Admin Contracts

A malicious user might want to modify admin contracts such as the Node Permissioning Contract, the Transaction Permissioning Contract, or the GAS Management Contract. For example, the malicious user would send a transaction in which the final destination contract after going through the RelayHub would be the method of removing nodes from the GAS Management Contract.

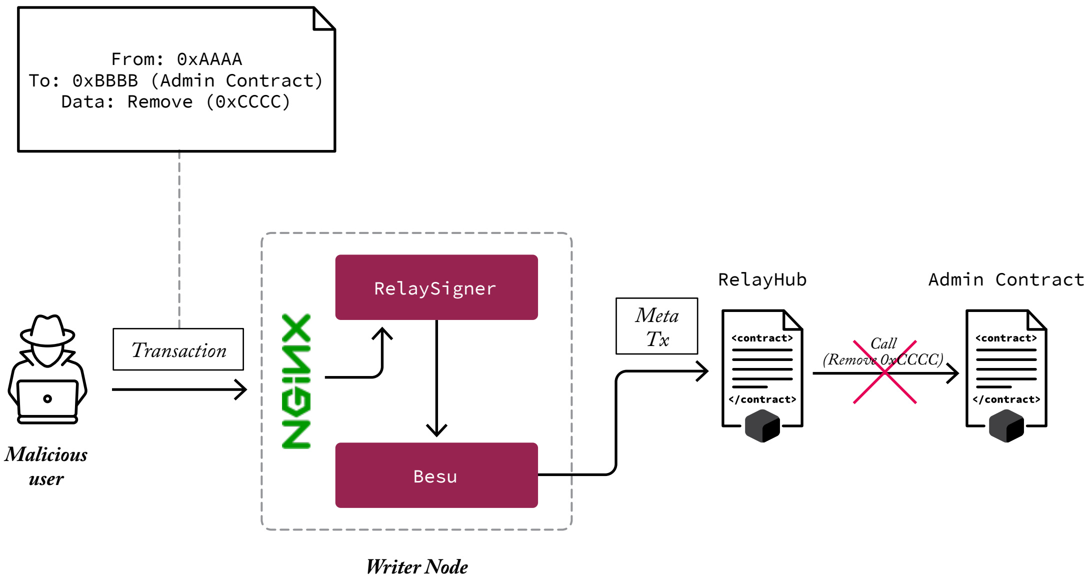

Figure 8. Atempt of trying to modify an Admin Contract.

This potential attack is prevented by the RelayHub. When the validator nodes execute transactions against the RelayHub, the RelayHub confirms that the target contract is not any of the management contracts. This prevents end-users from reaching the contracts through the RelayHub, which has permission, through these contracts, to enable authorized users to interact with these contracts. Only authorized users specified in the RelayHub are allowed to interact with the Admin Contracts. Through this process, the malicious transaction would not be executed against the Admin Contracts and the GAS would be discounted to the writer node that broadcasted it.

3.5. Transactions Broadcasted by Non-Permissioned Nodes

LACNet networks are permissioned networks, which means that permission is required for the node to join the network and connect to other nodes in the network. This permission is based on a permissioning smart contract. The Permissioning Committee updates the smart contract every time a node is added o removed from the permissioning, and all the permissioned nodes accept or reject connections based on the list of nodes reflected in the permissioning smart contracts.

A malicious writer node operator could remove its default node's permissioning settings (so it stops following the connections indicated in the Node Permissioning Smart Contracts), thereby accepting connections from new nodes. The malicious writer node operator would deploy a new malicious node or allow a third-party unauthorized node to connect to its node, allowing the unauthorized node or party to broadcast transactions that are replicated to the network by the permissioned node with the corrupted permissioning configuration.

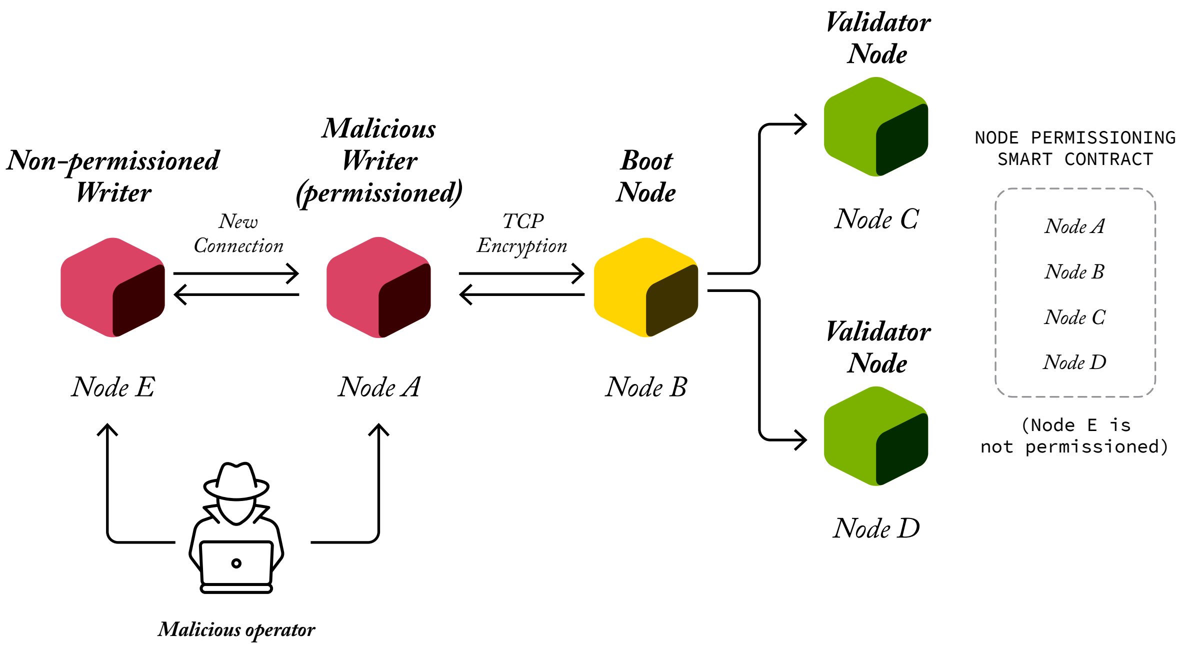

Figure 9. Malicious node attempting to serve as a proxy for non-permissioned nodes.

This potential attack is prevented by the Transaction Permissioning Smart Contract which rejects the transaction before it is execute because it is an invalid transaction as it is not signed by a permissioned node. The malicious node serving as a proxy will also be banned.

GAS
DISTRIBUTION PROTOCOL

4

GUIDELINES AND RECOMMENDATIONS FOR WRITER NODE OPERATORS

In this section we analyze the GAS distribution mechanism from the perspective of writer node operators. First, we review the only two modifications they need to introduce into solutions developed for Ethereum and Hyperledger Besu Networks in order to be adapted for LACNet GAS distribution mechanism and therefore to work in LACNet Networks. Second, we provide recommendations for writer node operators to protect their nodes from potential attacks.

4.1. Modifications to Solutions Developed for Ethereum Networks

As mentioned before, this GAS distribution mechanism has been designed to be as transparent as possible for end-users. Thus, the writer node back-end wraps the original transaction into a meta-transaction and signs it as required by the proxy smart contracts.

This GAS distribution mechanism brings only two small changes for end-users. The first is a difference in how to retrieve the original sender of a transaction and the second is indicating the writer node and the expiration time selected to broadcast the transaction.

4.1.1. Retrieving the Original Sender

The original transactions are not sent directly to the Recipient Smart Contract but by the RelyHub Contract (once it is checked that they meet the requirements detailed in Section 2.2). The transactions going to the RelyHub are, as explained before, meta-transactions wrapping the original transactions which are generated by the writer nodes.This makes it such that the sender of the transactions (the meta-transactions) reaching the Recipient Smart Contract is the RelayHub contract.

Because of this, it is necessary to have a mechanism to obtain the address of the original sender of the final meta-transaction (i.e., the client or user who sent the transaction to the writer node). To achieve this, we take advantage of the atomicity of the transaction execution in the EVM. That is, every time a transaction is sent to the RelayHub, the address of the original sender is stored, which is then retrieved by making a call to the RelayHub from the recipient contract.

This function to obtain the original sender is located in an abstract contract, which has to be inherited by all the contracts that will be deployed in the network.

```
praqma solidity >=0.4.22 <0.7.0;
/**
 *@title Storage
 *@dev Store & retrieve value in a
 variable
 */
contract Storage {
   mapping (address => uint256) bank;
   /**
    *@dev Store value in variable
    *@paran num value to store
    */
   function store(utnt256 nun) public {
   bank[msg.sender] = nun;
   }
   /**
    *@dev Return value
    *@return value of “number”
    */
   function retrieve(address sender)
   public view returns (uint256) {
      return bank[sender];
   }
```

```
praqma solidity >=0.4.22 <0.7.0;
import “./BaseRelayRecipient.sol”;
contract Storage is
BaseRelayRecipient{
   uint256 number;
   address owner;
   constructor () public {
      owner = _msgSender();
   }
   function getTrustedForwarder()
   public view returns(address) {
      return trustedForwarder;
   }
   function store(uint256 num)
   public {
      number = num;
      emit ValueSeted(_msgSender());
   }
   function getOwner() public view
   returns (address){
      return owner;
   }
   event ValueSeted(address sender);
}
```

Figure 10. Example of how to get the sender in this protocol.

4.1.2. Specifying Writer Node Address and Expiration Time

To avoid various attacks, such as the one described in Section 3, end-users, applications and services on top that send transactions must add two new parameters to their transaction to indicate the address of the node they want to broadcast their transaction to as well as the expiration time for the transaction to be executed. These two parameters are checked by validator nodes against the Transaction Permissioning Smart Contract (see Section 2.1.1). Parameters are as follows:

- nodeAddress(type:address): This parameter is the address associated to the private key that signs the transactions in the RelaySigner. Otherwise, by default, it is the address of the writer node through which the transactions will be sent.
- expiration(type:uint256): This parameter is the timestamp (Unix timestamp) that determines the expiration time for transaction to be executed. After this time, the transaction cannot be executed (added in a block).

4.2. Security Recommendations for Writer Node Operators

Writer node operators are contractually committed to LACNet to not send transactions with a GAS limit higher than what they have left per block, nor violating any other rule in the network-under penalty of being banned. Therefore, the entity operating the writer node must ensure that neither its use of the node nor the use by authorized third parties (e.g., applications and end users) violates these clauses.

It is not the responsibility of LACNet to mitigate attacks from writer nodes by corrupt end-users or node administrators, because the operation of these nodes is the responsibility of the writer node operators. Further, even if security layers were developed, the operators of these nodes could always modify them, generating vulnerabilities that would fall under their responsibility. This is due to the decentralized architecture of the network, which allows each node to be self-managed.

However, LACNet is not intended to let these writer node operators down with regard to potential attacks by end-users, services and applications on top, or others. On the contrary, LACNet has identified a set of threats posed to writer node operators and has provided tools, protocols, and suggestions for the writer node operators to mitigate any attack either to them or to the network through them.

It is important to note that even in the event that the writer node operator does not want or cannot protect the writer node from the attack, the attack will not affect the network and other node operators because the malicious node will be immediately banned as soon as any attempt of misbehavior in the network is identified (such as exceeding GAS limit).

4.2.1. Restricting and Protecting Access to the Writer Node

With prior knowledge that a writer node is exposed, attacks could be carried out by malicious users from both outside and inside the network. It is essential that writer nodes' RPC and WebSocket ports are never open publicly without some layer of security. In the case of boots and validators, both types of nodes are committed via SLA to not have RPC nor WebSocket open, and to not have APIs enabled for sending transactions.

4.2.2. Checking and Filtering Transactions

Each network writer node has the ability to deploy and customize a Local Account Permissioning Contract, which is a permissioning layer that writer node operators can use to filter (by whitelisting) the reliable addresses (senders) they allow to send transactions to the network through their writer node under rules customizable by the writer node operator.

This allows for each writer node operator to define its own rules for broadcasting transactions to the network. For writer nodes exposed to external users, services, and applications, this is extremely important, because writer node operators will be accountable for the transactions their node broadcasts, no matter who the original sender is.

Examples of rules include verifying the trail, the GAS limit, the destination of the transaction or the GAS price. They can also be more granular rules, such as verifying the data being sent in the transaction.

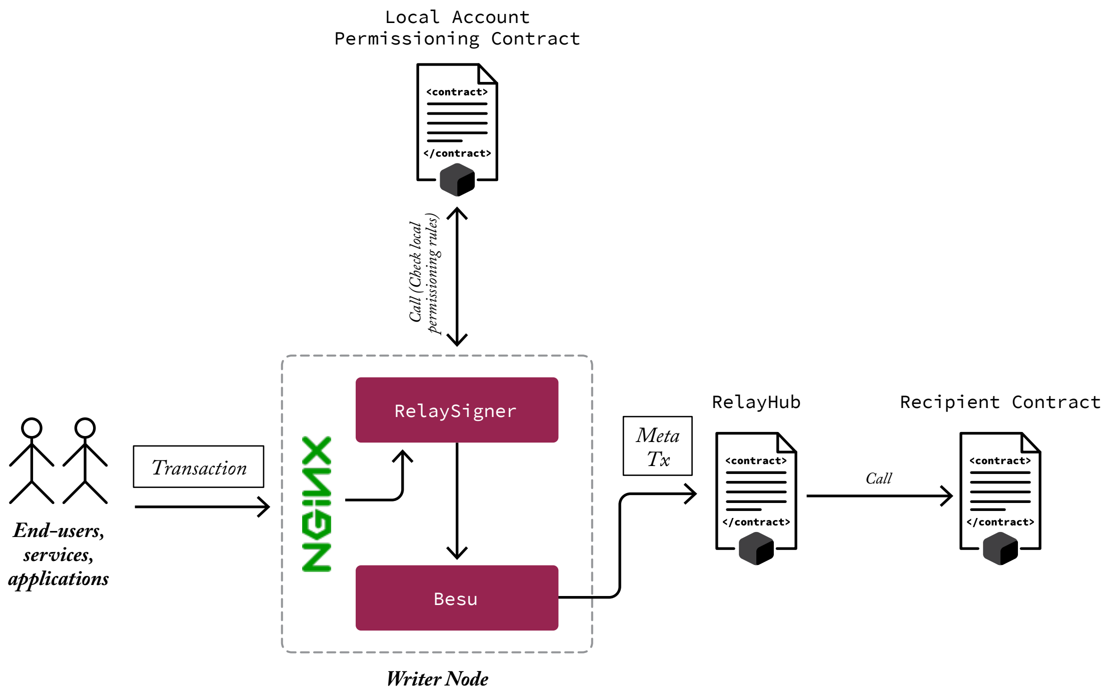
Local Account Permissioning Contract
<contract>
</contract>
Call (Check local permissioning rules)
End-users, services, applications
Transaction
NGINX
RelaySigner
Besu
Writer Node
Meta Tx
RelayHub
Call
Recipient Contract

Figure 11. Local permissioning layer at the writer node level.

4.2.3. Resending and Overwriting Transactions

Transactions sent by an innocent node to the network are replicated to all bootnodes. The bootnodes then replicate the transaction to the other writers and validators, which then execute the transactions. A malicious node could see a transaction in its pool and attempt to extract and forward it through its malicious node. It would do so with the intention that the second transaction is executed by the validators and the original transaction is executed with errors with its GAS consumed anyway. Even if the original transaction is executed at the attacker's expense, the end-user might be cheated because the end-user's attempt would be unsuccessful.

This potential attack is mitigated by the requirement that transactions must specify the writer node that is selected to broadcast them (otherwise they are rejected by the Transaction Permissioning Contract). Therefore, transactions will only be accepted if they have the signature of the node address that was indicated by the end user. Additionally, end-users must indicate an expiration time. For more information, see Section 4.1.2.

An alternative version of this attack could be to attempt to overwrite a transaction by increasing the GAS price of the duplicated transaction so validators are more likely to accept it first. This is impossible because the network has a GAS price of zero, so any transaction with a GAS price higher than zero is also rejected by the Transaction Permissioning Smart Contract.

APPENDIX A: STRESS-TESTING

LACNet ran different tests to understand the performance of the network with the GAS distribution mechanism when the network is being heavily utilized. The objective of the stress tests was to identify a utilization threshold where network performance begins to deteriorate. For the stress tests, LACNet set up smart contracts of various complexities and executed the stress test scenarios that were monitored and analyzed by considering the impact on network performance.

Network deterioration is quantified by examining block interval time. The genesis file for the network specifies the block interval time as 2 seconds. Blocks that take longer than 2 seconds to be produced constitute a situation in which network performance has deteriorated and has an impact on the transaction finality and throughput of the network. Thresholds for network deterioration are defined as situations where the likelihood of experiencing a block greater than 2 seconds is higher than normal. Deterioration is broken down into two types: 1) Moderate: block interval time between 2 and 4 seconds, and 2) Severe: block interval time greater than 4 seconds.

The hardware of the nodes is critical for the performance of the network. Also, all boots and validators must have similar hardware characteristics. Otherwise, the lower performing get delayed in transaction processing leading to delays in transaction propagation and block generation. The tests were run with the minimum hardware and software requirements indicated for LACNet Mainnet, which are:

- **CPU:** 4 virtual CPUs
- **RAM Memory:** 16 GB
- **Hard Disk:** 100 GB SSD (70,000 IOPS READ, 50,000 IOPS WRITE)
- **Operating System:** Ubuntu 16.04, Ubuntu 18.04, Ubuntu 20.04, Centos7, always 64 bits
- **Ethereum version:** Berlin
- **Besu versión:** 21.7.3

The results are presented in Table 3:

Contract	Tx/s Sent	Max_Tx/Block	Gas_Used/ block	Gas/ Transaction	Degradation_ block_time	Finality
Simple Contract	200	270 tx	33,301,601	123,339	----	170 Tx/s
ERC20	70	134 tx	18,822,763	140,468	----	67 Tx/s
ERC20	100	206 tx	28,930,167	140,468	----	90 Tx/s
ERC20	150	364 tx	51,110,429	140,468	3-4 seg	100 Tx/s
Identity	80	345 tx	93,139,220	269,968	3-4 seg	77 Tx/s
Register Covid	50	104 tx	35,862,281	344,830	----	50 Tx/s
Register Covid	80	306 tx	105,495,141	344,830	3-4 seg	60 Tx/s
ARM Aduanas	7	21 tx	64,562,539	3'074,149	2-3 seg	7 Tx/s
ARM Aduanas	10	41 tx	126,040,127	3'074,149	4-6 seg	7 Tx/s
DIDRegistry	50	122 tx	12,046,599	98,669	---	50 Tx/s
DIDRegistry	100	241 tx	23,826,973	98,669	3-4 seg	70 Tx/s
ERC-721	50	102 Tx	26,491,918	310,468	----	47 Tx/s
ERC-721	100	236 Tx	54,322,443	310,468	----	57 Tx/s
ERC-721	150	280 Tx	61,007,722	310,468	3-seg	80 Tx/s

Table 3. Throughput results for different types of transactions.

The results from the tests allow for several conclusions that need to be understood as conditioned to the hardware characteristics utilized:

- The two main parameters that produce block degradation are number of transactions and GAS used. The increase of these two parameters lead independently to block degradation.
- The network was able to process 170 tx/s simple transactions of 123,339 GAS/tx without degradation.
- The network was able to process 105M GAS per block with a moderate degradation that delayed the generation of the block to 3-4 seconds instead of the expected 2 seconds.

APPENDIX B: PRICE COMPARISON WITH ETHEREUM-BASED MAINNETS WITH TRANSACTION FEES

As we introduced in Section 1, the transaction-fee based protocols adopted by the most popular permissionless blockchain networks, such as Bitcoin and Ethereum, make them very unaffordable for use cases involving large amounts of transactions, such as typical government and enterprise solutions. In this appendix, we compare the difference in price between the transaction-fee based permissionless Ethereum Mainnet and the membership-fee based permissioned LACNet Network[6].

Type of LACNet membership	Maximum GAS per month available in LACNet*	Monthly cost of LACNet	Equivalent cost of same GAS in Ethereum**
Basic	648,000,000,000	$170	$77,760,000
Standard	1,940,000,000,000	$380	$233,280,000
Premium	6,480,000,000,000	$1250	$777,600,000

*This calculation is the result of multiplying the maximum GAS per block for each membership in LACNet by the average block number per month, considering block generation every 2 seconds.

**This calculation is the result of multiplying the maximum amount of GAS per month for each membership in LACNet by the price of the unit of GAS in Ethereum, using that 1 unit of GAS are 60gWei=6E-08Ether and 1Ether=$2,000.

The maximum amount of GAS that the basic membership of LACNet (by the beginning of year 2022) offers per month on its Network for 170 dollars would cost an equivalent of more than 77 million dollars on Ethereum Mainnet. The intermediate, offers a maximum amount of GAS for 380 dollars that in Ethereum would cost more than 233 million dollars, and with the premium goes from 1250 dollars to more than 777 million dollars. Unlike Ethereum Mainnet, LACChain is a network designed for highly transactional projects.

6 The memberhip fees for the LACChain Mainnet Network orchestrated by LACNet are available at https://lacnet.lacchain.net/get-your-membership/. For these calculations, we have taken the membership fees as of May 2022

GAS
DISTRIBUTION PROTOCOL