Recent discussions around communication abstractions have highlighted the importance of a clear spec around handling L1 → L2 messages within the rollup. In this article, we aim to outline an MVP process for consuming L1 messages and raise several open questions that warrant further investigation.

In a previous article, we created the following image to illustrate communication abstractions, I will provide an image here to serve as a refresher:

[

image

1531×631 113 KB

](https://europe1.discourse-cdn.com/business20/uploads/aztec/original/1X/25d944675c2f3acfd477efb47f609e5784f6da3f.jpeg)

The Lifecycle of L1->L2 Messages: A Brief Overview

The process of integrating L1 messages into L2 can be broken down into distinct stages:

1. A portal contract (or any l1 contract) writes to the rollup's message queue.
2. When the sequencer processes the next rollup, it selects X messages from the message queue and appends them to the L1->L2 Ready messages tree.
3. L2 contracts consume these messages by nullifying them.

To visualise this process, consider the following illustration of multiple portal contracts writing to the rollup's message queue:

[

image

826×363 25 KB

](https://europe1.discourse-cdn.com/business20/uploads/aztec/original/1X/d7d1b3456c8a3e3eff058f8535c5ec7d11ff5e4e.png)

As the sequencer processes the next rollup, it selects messages from the pending queue and inserts them into the L1->L2 messages tree:

[

image

1232×393 43.4 KB

](https://europe1.discourse-cdn.com/business20/uploads/aztec/original/1X/1c23a9ecfb5f5c70c44ec8e8c77cdbfd4dea1487.png)

Once messages have been inserted into this tree, L2 contracts can consume them by creating nullifiers, which are then inserted into the Nullifier Tree.

Key Concerns

Several caveats and open questions arise from this proposed approach:

1. How should the sequencer select which messages to include in the next block, in a way that prevents censorship?
2. What is the upper bound of messages to be included?
3. How can we verify which messages have been consumed within the contract?
4. How much will a user pay for the messages to prevent DDOS style attacks.

**For the immediate future / MVP I am proposing the following **

1. FIFO, it is simple to implement and is likely the fairest until some form of fee structure is established.
2. Fixed number of 32 messages per rollup. (This is an arbitrary value, I imagine it will be increased massively appraoching production)
3. Create append only tree, perform batch insertion into the tree. (also perform sha256 hash of inserted leaves to

compare with inside the contract).

4. Fees have not yet been modelled, and will not be implemented for the MVP.

Validating Consumed Messages within the Contract

One challenge is to determine how the rollup can validate that messages have been correctly appended to the message tree. In this article, we assume that L1-message appending will be executed within the root rollup, though it could also be implemented within a separate circuit validated (recursively) within the root rollup. The final decision will be based on performance considerations (yet to be benchmarked).

To verify that the hash of the expected messages matches the hash of the messages produced by the sequencer as a public input, we propose the following Solidity code:

function _getL1ToL2MessagesHash(uint256 _numberOfMessages) internal returns (bytes32) { // Create a memory structure to store messages bytes32[] memory messages = new uint;

// read messages from storage -> messages

return sha256(messages); }

This can be compared with a public input extracted from the proof calldata:

function processRollup(bytes calldata proof) external { // ... bytes messagesHash = _getL1ToL2MessagesHashFromCalldata(proof); require(messagesHash == _getL1ToL2MessagesHash(x), "Pending Messages hash does Not match"); }

Within the circuit, the pending messages hash will be reconstructed by hashing all provided values into the pending_l1_to_l2_messages_input. The circuit will then reconstruct the messages into a subtree and perform a batch insertion onto the L1_to_L2_messages tree. SHA256 will be used for this as it is available via a precompile and is cheaper than keccak to evaluate within a circuit.

Fields that will be added to the Root Rollup Circuit

The root rollup circuit will need to take in a series of messages as its private input and output the hash of them as a public input:

Private input addition:

std::array l1_l2_messages;

Public input addition:

std::array l1_l2_messages_hash;

Alternatively this public input could be collapsed into the single calldata_hash

which will reduce the number of public inputs by one, however it can equally be included in the solidity hash generation.

The root rollup and calldata will also need to include some extra tree information.

AppendOnlyTreeSnapshot startL1ToL2MessagesTreeSnapshot AppendOnlyTreeSnapshot endL1ToL2MessagesTreeSnapshot

In addition to a snapshot of the current state of the l1 messages tree, there will also need to be a historicL1ToL2MessagesTree

as users will need to be able to create membership proofs for messages against a given root if they wish to consume them.

AppendOnlyTreeSnapshot startHistoricL1ToL2MessagesTreeSnapshot AppendOnlyTreeSnapshot endHistoricL1ToL2MessagesTreeSnapshot.

Let me know if you think there is anything missing from this MVP spec, I will be adding to this article in the coming days.