L1 <--> L2 communication

Disclaimer Disclaimer We are building Aztec as transparently as we can. The documents published here are living documents. The protocol, sandbox, language, and tools are all subject to change over time.

Please seehere for details of known Aztec protocol and Aztec Sandbox limitations.

If you would like to help us build Aztec:

- Contribute code onGitHub
- ; or
- Join inforum
- discussions. In Aztec, what we callportals are the key element in facilitating communication between L1 and L2. While typical L2 solutions rely on synchronous communication with L1, Aztec's privacy-first nature means this is not possible. You can learn more about why in the previous section.

Traditional L1 <-> L2 communication might involve direct calls between L2 nd L1 contracts. However, in Aztec, due to the privacy components and the way transactions are processed (kernel proofs built on historical data), direct calls between L1 and L2 would not be possible if we want to maintain privacy.

Portals are the solution to this problem, acting as bridges for communication between the two layers. These portals can transmit messages from public functions in L1 to private functions in L2 and vice versa, thus enabling messaging while maintaining privacy.

This page covers:

- How portals enable privacy communication between L1 and L2
- How messages are sent, received, and processed
- Message Boxes and how they work
- How and why linking of contracts between L1 and L2 occurs

Objective

The goal is to set up a minimal-complexity mechanism, that will allow a base-layer (L1) and the Aztec Network (L2) to communicate arbitrary messages such that:

- L2 functions cancall
- L1 functions.
- L1 functions cancall
- L2 functions.
- The rollup-block size have a limited impact by the messages and their size.

High Level Overview

This document will contain communication abstractions that we use to support interaction betweenprivate functions,public functions and Layer 1 portal contracts.

Fundamental restrictions for Aztec:

- L1 and L2 have very different execution environments, stuff that is cheap on L1 is most often expensive on L2 and vice versa. As an example,keccak256
- is cheap on L1, but very expensive on L2.
- L1 and L2 have causal ordering, simply meaning that we cannot execute something on L1 that depends on something happening on L2 and vice versa.
- Private
- function calls are fully "prepared" and proven by the user, which provides the kernel proof along with commitments and nullifiers to the sequencer.
- Public
- functions altering public state (updatable storage) must be executed at the current "head" of the chain, which only the sequencer can ensure, so these must be executed separately to theprivate
- functions.
- Private
- andpublic
- functions within Aztec are therefore ordered such that firstprivate
- functions are executed, and thenpublic
- . For a more detailed description of why, see above.

- There is anexplicit 1:1 link
- from a L2 contract to an L1 contract, and only the messages between a pair is allowed. SeePortal
- for more information.
- Messages are consumables, and can only be consumed by the recipient. SeeMessage Boxes
- for more information.

With the aforementioned restrictions taken into account, cross-chain messages can be operated in a similar manner to whenpublic functions must transmit information toprivate functions. In such a scenario, a "message" is created and conveyed to the recipient for future use. It is worth noting that any call made between different domains (private, public, cross-chain) is unilateral in nature. In other words, the caller is unaware of the outcome of the initiated call until told when some later rollup is executed (if at all). This can be regarded as message passing, providing us with a consistent mental model across all domains, which is convenient.

As an illustration, suppose a private function adds a cross-chain call. In such a case, the private function would not have knowledge of the result of the cross-chain call within the same rollup (since it has yet to be executed).

Similarly to the ordering of private and public functions, we can also reap the benefits of intentionally ordering messages between L1 and L2. When a message is sent from L1 to L2, it has been "emitted" by an action in the past (an L1 interaction), allowing us to add it to the list of consumables at the "beginning" of the block execution. This practical approach means that a message could be consumed in the same block it is included. In a sophisticated setup, rollupn n n could send an L2 to L1 message that is then consumed on L1, and the response is added already inn + 1 n+1 n

+ 1 . However, messages going from L2 to L1 will be added as they are emitted.

info Because everything is unilateral and async, the application developer have to explicitly handle failure cases such that user can gracefully recover. Example where recovering is of utmost importance is token bridges, where it is very inconvenient if the locking of funds on one domain occur, but never the minting or unlocking on the other.

Components

Portal

A "portal" refers to the part of an application residing on L1, which is associated with a particular L2 address (the confidential part of the application). The link between them is established explicitly to reduce access control complexity. On public chains, access control information such as a whitelist in a mapping or similar data structure can simply be placed in public storage. However, this is not feasible for contracts in Aztec. Recall that public storage can only be accessed (up to date) by public functions which are called AFTER the private functions. This implies that access control values in public storage only work for public functions. One possible workaround is to store them in private data, but this is not always practical for generic token bridges and other similar use cases where the values must be publicly known to ensure that the system remains operational. Instead, we chose to use a hard link between the portal and the L2 address.

info Note, that we at no point require the "portal" to be a contract, it could be an EOA on L1.

Message Boxes

In a logical sense, a Message Box functions as a one-way message passing mechanism with two ends, one residing on each side of the divide, i.e., one component on L1 and another on L2. Essentially, these boxes are utilized to transmit messages between L1 and L2 via the rollup contract. The boxes can be envisaged as multi-sets that enable the same message to be inserted numerous times, a feature that is necessary to accommodate scenarios where, for instance, "deposit 10 eth to A" is required multiple times. The diagram below provides a detailed illustration of how one can perceive a message box in a logical context.

- Here, asender
- will insert a message into thepending
- set, the specific constraints of the actions depend on the implementation domain, but for now, say that anyone can insert into the pending set.
- At some point, a rollup will be executed, in this step messages are "moved" from pending on Domain A, to ready on Domain B. Note that consuming the message is "pulling & deleting" (or nullifying). The action is atomic, so a message that is consumed from the pending set MUST be added to the ready set, or the state transition should fail. A further constraint on moving messages along the way, is that only messages where thesender
- andrecipient
- pair exists in a leaf in the contracts tree are allowed!
- When the message have been added to the ready set, therecipient
- can consume the message as part of a function call.

Something that might seem weird when compared to other cross-chain setups, is that we are "pulling" messages, and that the message don't need to be calldata for a function call. ForArbitrum and the like, execution is happening FROM the "message bridge", which then calls the L1 contract. For us, you call the L1 contract, and it should then consume messages from the message box. Why?Privacy ! When pushing, we would be needing fullcalldata . Which for functions with private inputs is not really something we want as that calldata for L1 -> L2 transactions are committed to on L1, e.g., publicly sharing the inputs to a private function.

By instead pulling, we can have the "message" be something that is derived from the arguments instead. This way, a private function to perform second half of a deposit, could leak the "value" deposited and "who" made the deposit (as this is done on L1), but the new owner can be hidden on L2.

To support messages in both directions we logically require two of these message boxes (one in each direction), and then message passing between L1 and L2 is supported! However, due to the limitations of each domain, the message box for sending messages into the rollup and sending messages out are not fully symmetrical. In reality, the setup looks closer to the following:

info The L2 -> L1 pending messages set only exist logically, as it is practically unnecessary. For anything to happen to the L2 state (e.g., update the pending messages), the state will be updated on L1, meaning that we could just as well insert the messages directly into the ready set.

Rollup Contract

The rollup contract has a few very important responsibilities. The contract must keep track of theL2 rollup state root , performstate transitions and ensure that the data is available for anyone else to synchronize to the current state.

To ensure thatstate transitions are performed correctly, the contract will derive public inputs for therollup circuit based on the input data, and then use averifier contract to validate that inputs correctly transition the current state to the next. All data needed for the public inputs to the circuit must be from the rollup block, ensuring that the block is available. For a valid proof, therollup state root is updated and it will emit anevent to make it easy for anyone to find the data by event spotting.

As part ofstate transitions where cross-chain messages are included, the contract must "move" messages along the way, e.g., from "pending" to "ready".

Kernel Circuit

For L2 to L1 messages, the public inputs of a user-proof will contain a dynamic array of messages to be added, of size at mostMAX_MESSAGESTACK_DEPTH , limited to ensure it is not impossible to include the transaction. The circuit must ensure, that all messages have asender/recipient pair, and that those pairs exist in the contracts tree and that thesender is the L2 contract that actually emitted the message. For consuming L1 to L2 messages the circuit must create proper nullifiers.

Rollup Circuit

The rollup circuit must ensure that, provided two statesS S S andS ′ S' S ′ and the rollup blockB B B , applyingB B B toS S S using the transition function must give usS ′ S' S ′ , e.g.,T (S , B) ↦ S ′ T(S, B) \mapsto S' T (S ,

B)

↦ S ′ . If this is not the case, the constraints are not satisfied.

For the sake of cross-chain messages, this means inserting and nullifying L1→ \rightarrow → L2 in the trees, and publish L2→ \rightarrow → L1 messages on chain. These messages should only be inserted if thesender andrecipient match an entry in the contracts leaf (as checked by the kernel).

Messages

While a message could theoretically be arbitrarily long, we want to limit the cost of the insertion on L1 as much as possible. Therefore, we allow the users to send 32 bytes of "content" between L1 and L2. If 32 suffices, no packing required. If the 32 is too "small" for the message directly, the sender should simply pass along asha256(content) instead of the content directly (note that this hash should fit in a field element which is ~254 bits. More info on this below). The content can then either be emitted as an event on L2 or kept by the sender, who should then be the only entity that can "unpack" the message. In this manner, there is some way to "unpack" the content on the receiving domain.

The message that is passed along, require thesender/recipient pair to be communicated as well (we need to know who should receive the message and be able to check). By having the pending messages be a contract on L1, we can ensure that thesender = msg.sender and let onlycontent andrecipient be provided by the caller. Summing up, we can use the struct's seen below, and only store the commitment (sha256(LxToLyMsg)) on chain or in the trees, this way, we need only update a single storage slot per message.

struct

L1Actor

{ address : actor , uint256 : chainId , } struct

L2Actor

{ bytes32 : actor , uint256 : version , }

struct

L1ToL2Msg

{ L1Actor : sender , L2Actor : recipient , bytes32 : content , bytes32 : secretHash , uint32 deadline , uint64 fee , }

struct

L2ToL1Msg

{ L2Actor : sender , L1Actor : recipient , bytes32 : content , } info Thebytes32 elements forcontent andsecretHash hold values that must fit in a field element (~ 254 bits). info The nullifier computation should include the index of the message in the message tree to ensure that it is possible to send duplicate messages (e.g., 2 x deposit of 500 dai to the same account).

To make it possible to hide when a specific message is consumed, theL1ToL2Msg is extended with asecretHash field, where thesecretPreimage is used as part of the nullifier computation. This way, it is not possible for someone just seeing theL1ToL2Msg on L1 to know when it is consumed on L2. Also, we include thedeadline andfee values to have a fee-market for message inclusion and to ensure that messages are not stuck in the pending set forever.

Combined Architecture

The following diagram shows the overall architecture, combining the earlier sections.

Linking L1 and L2 contracts

As mentioned earlier, there will be a link between L1 and L2 contracts (with the L1 part of the link being the portal contract), this link is created at "birth" when the contract leaf is inserted. However, the specific requirements of the link is not yet fully decided. And we will outline a few options below.

The reasoning behind having a link, comes from the difficulty of L2 access control (see "A note on L2 access control"). By having a link that only allows 1 contract (specified at deployment) to send messages to the L2 contract makes this issue "go away" from the application developers point of view as the message could only come from the specified contract. The complexity is moved to the protocol layer, which must now ensure that messages to the L2 contract are only sent from the specified L1 contract.

info The design space for linking L1 and L2 contracts is still open, and we are looking into making access control more efficient to use in the models.

One L2 contract linking to one L1

One option is to have a 1:1 link between L1 and L2 contracts. This would mean that the L2 contract would only be able to receive messages from the specified L1 contract but also that the L1 should only be able to send messages to the specified L2 contract. This model is very restrictive, but makes access control easy to handle (but with no freedom).

It is possible to model many-to-many relationships through implementing "relays" and listing those. However, L2 contracts that want to use the relay would have to either use dynamic access control to ensure that messages are coming from the relayer and that they where indeed relayed from the correct L1 contract. Essentially back in a similar case to no links.

To enforce the restriction, the circuit must ensure that neither of the contracts have been used in any other links. Something that in itself gives us a few problems on frontrunning, but could be mitigated with a handshake between the L1 and L2 contract.

Many L2 contracts linking to one L1

From the L2 contract receiving messages, this model is very similar to the 1:1, only one L1 contract could be the sender of messages so no extra work needed there. On the L1 side of things, as many L2 could be sending messages to the L1 contract, we need to be able to verify that the message is coming from the correct L2 contract. However, this can be done using easy access control in the form of storage variables on the L1 contract, moving the design-space back to something that closely resembles multi-contract systems on L1.

When the L1 contract can itself handle where messages are coming from (it could before as well but useless as only 1 address could send), we don't need to worry about it being in only a single pair. The circuits can therefore simply insert the contract leafs without requiring it to ensure that neither have been used before.

With many L2's reading from the same L1, we can also more easily setup generic bridges (with many assets) living in a single L1 contract but minting multiple L2 assets, as the L1 contract can handle the access control and the L2's simply point to it as the portal. This reduces the complexity of the L2 contracts as all access control is handled by the L1 contract.

Open Questions

- Can we handle L2 access control without public function calls?* Essentially, can we have "private shared state" that is updated very sparingly but where we accept the race-conditions as they are desired in specific instances.
- What is the best way to handle "linking", with efficient access control, could use this directly.
- What is the best way to handle messages in a multi-rollup system? E.g., rollup upgrade is rejected by some part of users that use the old rollup.* What happens to pending messages (sent on old system then upgrade)?
-
    - Should both versions push messages into same message boxes?
-
    - How should users or developers signal what versions their contracts respects as the "current" version?Edit this page