A general-purpose bridge for Ethereum Layer 2s

Nick Johnson

Follow

--

Listen

Share

With the proliferation of layer 2 solutions for Ethereum that are starting to reach maturity, it's important that ENS is able to provide resolution services across the entire ecosystem, as well as making it possible for ENS users to take advantage of the efficiencies made possible by Layer 2 solutions. Subsequent to a post by Vitalik that suggested a possible means for this, the ENS team and the wider ENS and L2 community have been working on a general-purpose "Layer 2 bridge" that makes cross-platform interoperability possible for both ENS and other applications that need to be able to retrieve data from a variety of offchain sources in a trustless fashion.

At the last working group meeting on the 27th of October, I presented a demo of a preliminary implementation of this idea. In this post I will describe the solution in detail.

Goals

Broadly speaking, layer 2 solutions and other related systems function by reducing the need to interact directly with Ethereum by taking state that would previously have been stored and accessed onchain, and moving it elsewhere, while still retaining enough information on Ethereum to verify the correctness of that data. For example, in one common approach, a Rollup, the state is stored in a separate system, and only witness data such as a Merkle Root is retained on the Ethereum chain. Given this witness data and access to the layer 2 solution, a participant can construct a proof of the validity of any data contained on the layer 2 system, which can be validated on Ethereum.

This definition is broader than what most people would think of as "Layer 2" — it also encompasses other means of reducing onchain stored data such as airdrops that use merkle trees of account balances, and tokens that emit events but do not store account balances onchain.

The key problem for ENS and many other applications, in a world with a profusion of different, incompatible layer 2 solutions, is how to retrieve data from one of these systems in a way that is trust-minimised — that is, it does not introduce any new trust assumptions — without the application having to act as a client of each layer 2 solution that could potentially store data that is of interest to it.

A naive solution would be to require that all systems use the same format for witness data, but this is lacking for two reasons: First, the format and type of witness data is highly dependent on the implementation details of the system in question — a ZK Rollup will use different primitives to an Optimistic Rollup — and second, it still leaves the client with no way to actually fetch the data.

Any practical solution will need to satisfy several criteria:

- Clients must not need to build explicit support for each system they may interface with.
- Clients must be able to verify the returned data is valid, preferably without any expansion of the trust model beyond that implied by the L2 solution being used.
- The solution must not require structural changes from an L2 platform in order to function.
- Third-parties must be able to build an interface to an L2 platform without the involvement or support of those maintaining the platform.

Solution Overview

Our proposed solution relies on standardising a means by which a client can retrieve data from an external system — a gateway service — and on standardising a way to verify that the returned data is correct.

Accordingly, there are two major components: First, a smart contract on Ethereum L1, which provides the client with a means of discovering a gateway, and of verifying the correctness of the gateway's response. Second, a gateway service, which understands how to integrate with a given L2 and how to format the data for the contract's consumption.

Fetching data under this model is a three-step process:

1. Query the contract for the data being requested. Rather than returning the result directly, the contract returns two values: a gateway URL

and a calldata prefix

.

1. Send an HTTP POST request to the gateway URL

with the same query data as in step 1. The gateway returns an opaque value, resolver calldata

. Verify that the resolver calldata

starts with the calldata prefix

provided by the contract in step 1.

1. Query the contract or transact with it, supplying the resolver calldata

from step 2. The contract verifies the validity of the data and, if valid, returns the result

or executes the transaction.

Because the gateway service is responsible for understanding how to interface with the L2 being used, this simple protocol enables clients to fetch data from offchain by utilising a gateway service that understands how to interface with the specific L2 — but removes any need for the client to understand anything about the L2 in question. To use this system, each application needs to implement and deploy a gateway service for the L2 they wish to interact with, and a verifier contract, with a shared protocol understood by both gateway and contract. In many cases, these gateways can themselves by very general-purpose, relieving the need for duplication of effort across different applications.

Significantly, the three step workflow above can be abstracted away for the caller entirely; a library that understands this protocol can make the whole process look the same as a regular web3 contract call, meaning that not only does the application not need to know which L2 they're interacting with — they don't have to know they're interacting with an L2 at all!

The gateway is severely constrained in its ability to return false or misleading results by the protocol itself. The verification logic implemented in the contract ensures that any invalid result will be detected in step 3, while the required prefix returned by the contract in step 1, and verified in step 2, prevents the gateway from answering a query with a valid answer for a different query.

Worked Example

For a simple example of how this works in practice, we will demonstrate with an ERC20 token that is preloaded with a set of balances, and a "Layer 2" that is a simple static merkle-tree. Our initial onchain implementation looks something like this:

This simple solution has an obvious problem: the deployer has to populate the preload

mapping with all of the balances at deployment time, an expensive operation. They would much rather store this data offchain, and allow callers to claim their balance if they can prove they have one. We can do this fairly easily using a merkle tree:

(For simplicity, we are omitting the implementation of verifyProof.

)

This works very well as far as the contract goes; the author no longer needs to spend a great deal of ether preloading all the balances; a single merkle root suffices, and the callers can pay the cost of proving ownership of tokens when they wish to claim them.

However, now the callers have to understand the specific process for generating proofs, as well as knowing where to fetch the list of balances from in order to generate a proof for their account. It would be far preferable if we could have the interface from the first solution, with the efficiency of the second. This is where our solution comes in.

First, we add methods matching the call signatures of the original claim

and claimableBalance

methods:

A caller to these functions gets back two values: First, a required prefix for a later callback, and second, the URL of a gateway service to consult. The prefix enforces two things: that the callback will be to the corresponding ...WithProof function, and that its first argument will be the address supplied. This prevents the gateway from responding with data for another address.

Next, we need to implement a gateway server that can satisfy the client's queries for these methods. Using claim

as an example, this is fairly straightforward:

(Again, we are assuming suitable implementations for functions such as getProof

for brevity.)

The gateway service here simply has to decode the function call data for the call to claim

that the client sent, assemble a proof — or in the case of an actual L2, consult the L2 to assemble a proof — and encode the result as a call to claimWithProof

, returning it to the client.

Finally, the client verifies that the returned calldata starts with the prefix asserted by the contract, and if it does, submits the calldata as a transaction to the contract.

The implementation of claimableBalance

is substantially the same, with the exception that the client uses the calldata to make a call to the contract, treating the return value as the final result of the call.

Security Considerations & Trust Model

If we assume that the client trusts the original contract — by which we mean, expects it to behave in a certain way, which is verifiable by examining the published source-code — then this system introduces no new trust assumptions. Though the gateway is an external process, its scope for misbehaviour is limited to denial of service.

First, if we trust the contract, we should also trust it to specify a gateway URL that can answer our queries. Second, we can also trust it to implement sufficient validation to ensure that the gateway's responses are accurate, both by specifying a calldata prefix in the first step, and by verifying the gateway's response in the last step.

Thus, a gateway that attempts to respond incorrectly — either with incorrect data or an incorrect proof — will be detected when the contract performs its verification in step 3. A gateway that attempts to respond correctly, but with a response to a query other than the one the user made, will be detected by the calldata prefix check in step 2. The client can ensure this by examining the behaviour of the contract — or relying on someone else's examination of the contract — before interacting with it.

A gateway can refuse to respond at all, effectively denying service, and this may happen maliciously or due to accident or neglect. For this reason, we propose that any final specification should make it easy for users to fork the service and provide their own gateway to be used in place of the contract's with similar ease to how users currently can fork a DApp frontend.

Application to ENS

Application of this system to ENS is relatively straightforward. Resolvers can implement the protocol described in this post for resolving any of their data fields, and so new resolver implementations and corresponding gateways can be deployed for each L2 wishing to support storage and retrieval of ENS data. A user wishing to take advantage need only store their records on the appropriate L2, and make a one-time transaction on Ethereum specifying the relevant resolver address as the one to use for their name.

To make this more generally useful, ENS should also be modified to support some form of wildcard resolution, where a failure to look up a name results in consulting the resolver for the parent of that name — so if 'foo.example.eth' does not exist, the client will perform the lookup on the resolver of 'example.eth'. This permits storing entire subtrees of ENS on other systems, rather than just the records for a single name.

Unresolved Issues

- While some applications such as ENS benefit from the extra layer of indirection that comes from allowing the contract to specify its gateway URL, others such as the token example above would be better served by encoding this as a part of the contract's ABI, making it easier for users to fork. An eventual solution should ideally allow both of these options, without imposing undue burden either.
- Presently, a client cannot distinguish between a gateway that returns invalid calldata (for example, by providing an

invalid proof) and a call that will revert regardless. Some provision needs to be made for distinguishing these two cases — for example, by requiring that the contract use a specific revert reason if validation of the proof data fails.

- This solution needs a name catchier than “A general-purpose bridge for Ethereum L2s”.

Try It Yourself

All of the source code for my demo can be foundhere.

Follow us

- Subscribe to our email list
- Follow us on Twitter
- Join the discussion on our forum