Lambda Protocol: Programmable logic powered by Bitcoin inscriptions

Lambda

1 Introduction

Current protocols on Bitcoin are focusing heavily on token transactions, with statically defined operations and their inferred balance changes. The operations are defined at the protocol creation and are really limited, because every byte needed in the inscription JSON is costing users money. **But why** does it have to be so limited? Why can we not create a protocol which is dynamic, fluid, interactable and open? Opening the door to many more applications, DeFi and general computation. **So again we ask, "why?"**

1.1 Protocols

But first of, what are protocols on Bitcoin? Because it might be hard to grasp the further details if you are unfamiliar with them. Protocols in the simplest terms are

- an agreed upon definition of operations
- each operation mutates something
- each mutation is pure -> same state + mutation leads to the same state afterwards

However, the important thing is. Where the mutation happens, and where the state is kept is up to each user to decide. If you feel adventurous you can do it with pen and paper. And send anyone asking for the current state a copy using a pigeon.

Small example to demonstrate a simple token-focused protocol:

1. Alex creates 100 PROTO token
2. Alex sends Beth 5 PROTO token
3. Beth sends Luke 3 PROTO token
4. Alex sends Luke 10 PROTO token

If you stop at this point you should be able to know the balances of each person. Alex: 85; Beth: 2; Luke: 13. That is all the protocol defines. And the defined operations are *create* and *send*.

2 Problem Statement

So what are problems with the current protocols. They don't have any programmable logic in them. BRC-20 for example, only has deploy, mint and transfer operations. Transfer looks like this

```
{
  "p": "brc-20",
  "op": "transfer",
  "tick": "ordi",
  "amt": "100"
}
```

But you can put whatever you want into this JSON. Like our protocol does, as long as it is a valid protocol inscription. Nowadays there are more powerful protocols, however they are still only focused on tokens and not the bigger ecosystem. The only limitations is the amount of data for a valid protocol inscription. Theoretically, we could write code inside an inscription, but this would be really costly.

3 Solution

The question is, how can you define a protocol which can execute any function, only with inscriptions. The answer is simple. You have real code and functions, with the inscription being the selector of it.

Simple example, a file called math.ts with the function increment. Now the inscription could look like math.function. This leads to the same behavior as above, provided the math.ts file is open-source. The value in the beginning of increment is 0.

1. math.increment
2. math.increment
3. math.increment

at this point the value is 3. Somebody could write another contract with more functionality, interactions and much more.

4 The Protocol

With all this is mind, let me introduce Lambda Protocol. Is it built around inscriptions which in turn trigger some contract functions. To make the contracts as powerful as possible they are written in TypeScript and get some additional APIs out of the box.

- Arguments
- Metadata
- Events
- Ecosystem API

Each function behaves like a database transaction either all is executed or none. It must not be that a contract executes halfway, writes the updated state and then errors. Either all or nothing. Queries are read only, allowing anybody to read state at any time. An inscription for this protocol looks like

```
{
    "p": "lam",
    "op": "call",
    "contract": "token",
    "function": "transfer",
    "args": [
        "alex",
        10
    ]
}
```

Contract and function are the selectors to choose which function is executed, the arguments are defined by the contract creator and are passed through to the contract itself. So this inscription indicates a transfer of 10 token to Alex. However, it is impossible to be certain until we read the token contract itself. So, due diligence on what contract you are interacting with is of extremely importance (just as it is in any other Smart Contract protocol), as anyone able could put anything behind the transfer function, even malicious code.

4.1 Rules of the Protocol

1. only inscriptions can mutate state
2. each inscription specifies the contract, function and arguments
3. each contract has access to the event logger API, metadata and ecosystem API
4. each contract function is either executed fully or not at all, it must not be executed halfway and mutate persistent state
5. contracts must not access the underlying hardware or damage the hardware or software, if they do they are invalid and will be rejected / removed.
6. contracts must have a reasonable execution time, otherwise they are invalid and will be rejected / removed

7. contracts must be pure. Running the inscription with the same beginning state must always result in the same end state
8. contracts are written in TypeScript or JavaScript
9. contracts don't have access to any other APIs, then the ones provided by the protocol
10. contracts are immutable

5 Benefits

What is possible with this? Everything! You can write any logic, any software, anything which is then triggered by inscription written to the blockchain. Let me repeat, any customizable logic can be executed with inscriptions. The examples of DeFi applications are unlimited. You are able to use your Bitcoin as collateral to borrow a stablecoin, to farm a shitcoin with and provide liquidity for a DEX with the same shitcoin/stablecoin pair. All natively on Bitcoin.

Keep in mind though, this is on Bitcoin, not a sidechain or centralized entity. It is all on Bitcoin (minus the source files for the contracts). The only thing centralized is the state management triggered by the inscriptions. Therefore, if you query a state for your token balance, make sure the Lambda indexer serving your request is honest. You can always build and run your own indexer. An honest indexer, updates the state via the inscriptions and contracts. It is easy to see honest indexers as the contracts are pure per default, same state + contract = same resulting state.

6 Example Contract

How does a contract look like and how can you use all the APIs.

Short explanation. Contract is the base of everything. Implementing a TypeScript class is all that needs to be done. Each function can be triggered via inscriptions or queries. However, only inscriptions save the state.

So how do we go about making a small contract writing something to storage for the sender. We need the sender of course and also want to log events. That is all we need. We use some utilities to parse the arguments with zod, a validation library, and use a Map to store the message of each sender.

```
import { Contract, ContractParams } from "./types/contract.ts";
import { z } from "zod";
import { argsParsing } from "./utils/args-parsing.ts";

export default class ReadAndStore implements Contract {
  activeOn = 800000;
  message = new Map<string, string>();

  store = ({ metadata, eventLogger, args }: ContractParams) => {
    const schema = z.tuple([z.string()]);
    const [valueToStore] = argsParsing(schema, args, "save");

    this.message.set(metadata.sender, valueToStore);

    eventLogger.log({
      type: "SAVE",
      message: `${valueToStore} stored for ${metadata.sender}`,
    });
  };

  read = (args: unknown[]) => {
    const schema = z.tuple([z.string()]);
    const [from] = argsParsing(schema, args, "read");
    return this.message.get(from) ?? "";
  };

  blockNumber = ({ metadata }: ContractParams) => {
    return metadata.blockNumber;
  };
}
```

We can see this is really simple and the APIs provide us with everything we need. Metadata includes more useful stuff you might need in your own contracts. The eventLogger API is used to include events into the transaction. They are helpful for many things, including token transfers, swaps and so on. To get more ideas about contracts you can visit the GitHub repository.

7 Implementation of an Indexer

We provide an indexer, query server and execution module for anyone to run. This will provide honest protocol state to anyone via REST API, additionally we will run this exact software and provide public endpoints to it. In the future the code will be open-source and we are planning to make it decentralized.

8 LMDA

Lambda Protocol is launching its own native token called LMDA. This token is used across the ecosystem to provide utility and governance to Lambda Protocol. There are going to be 1 billion LMDA tokens. only around 15% are available at the LMDA launch date, the rest is either locked and vested or up to the community to decide. We have any utilities planned for LMDA, like staking, and even more as the protocol matures and moves into a decentralized version of itself.

8.1 Tokenomics

- 3% advisors
- 5% Initial Treasury (Locked Long Term)
- 5% Private Sales / OTC
- 10% BasedPad IDO
- 16% Team (vested over 2 years)
- 30% Locked Treasury (gradual unlocking over 2 years)
- 31% Ecosystem / Community

9 Conclusion

In conclusion, this protocol allows arbitrary contracts to be executed via inscriptions. This opens up the possibility for DeFi, and much more. Our implementation of the protocol includes an indexer, query server and Lambda Engine to provide honest state for the users of this protocol. However, if you want to, you are free to create your own protocol implementation and it is going to have the same state at any given time. It is exciting to see what the developers come up with and see the ecosystem grow bigger than anybody can imagine.