Graph Protocol Specification

Version: 0.0.1

Stage: stage wip

Authors:

- Brandon Ramirez

Abstract

This document presents *Graph Protocol* ("the protocol"), a protocol for indexing public blockchain data and querying this data via a decentralized network. The canonical network implementing the protocol is referred to as *The Graph* ("the network").

Graph Protocol falls into a category we refer to as a *layer 2 read-scalability* solution. Its purpose is to enable decentralized applications (dApps) to query public blockchain data efficiently and trustlessly via a service that, like blockchains and the Internet itself, operates as a public utility. This is in the interest of minimizing the role of brittle, centralized infrastructure seen in many "decentralized" application architectures today.

This specification covers the network architecture, protocol interfaces, algorithms, and economic incentives required to build a network that is robust, performant, cost-efficient, and enables a high margin of economic security for queries processed via the network.

Philosophy

This spec defines a hybrid network design in which the core mechanisms are decentralized and run on the blockchain, but some building blocks are still centralized. A future version of this specification will target full decentralization. This is in keeping with our team's philosophy of shipping early and delivering immediate value, while incrementally decentralizing, as research and the state of external ecosystem dependencies progress.

See this slide from this recent research talk for more info on this approach.

Disclaimer

This spec defines a protocol that is still being implemented. Until a fully stable reference implementation exists, the specification is likely to change in breaking ways.

Table of Contents

Architecture Overview

High-Level Architecture

```
    +-------------------------------------------------------------------+
    |                                                                   |
    | Decentralized Application (dApp)                                  |
    |                                                                   |
    +-+---------------------------------^--------------------------+---+
      |                                 |                          |
      |                              Queries                       |
      |                                 |                          |
      |   +-----------------------------+--------------------+     |
      |   |                                                  |     |
      |   |  Query Nodes and Clients                         | Micropayments
      |   |                                                  |     |
      |   +---------+-------------------^--------------------+     |
      |             |                   |                          |
Transactions   Attestations     (Reads, Attestations)              |
      |             |                   |                          |
      |   +---------v-----------+  +----+--------------------------v---+
      |   |                     |  |                                    |
      |   |  Fisherman Service  |  | Indexing Nodes                     |
      |   |                     |  |                                    |
      |   +---------+-----------+  +----^-------------------^-----------+
      |             |                   |                   |
      |         Disputes          (Events, Data)          Data
      |             |                   |                   |
    +-v-------------v-------------------+-----+ +-----------+-----------+
    |                                         | |                       |
    |                Ethereum                 | |   IPFS               |
    |                                         | |                      |
    +-----------------------------------------+ +----------------------+
```

Overview

The Graph supports a command query responsibility segregation (CQRS) pattern where dApps send commands (transactions) directly to the underlying Ethereum blockchain, but issue queries (reads) against the layer 2 Indexing Nodes, in exchange for micropayments, via a Query Node or Query Client. In addition to being able to scale reads independently from transactions, there is the added benefit of being able to specify read semantics which differ from the more limited write semantics supported by the Ethereum blockchain (see Data Modeling). Indeed it allows the dApp to query a completely different view on the underlying blockchain data, which may be augmented by data that is stored off-chain on IPFS. Read responses are accompanied by attestations, messages which certify the correctness of a response, which a Query Node may optionally provide to a Fisherman Service to improve the economic security guarantees as to the correctness of responses in the network (see Mechanism Design).

Note: While the above diagram conveys the full architecture of a dApp interacting with The Graph, the protocol is primarily concerned with the interface to the Indexing Nodes, as well as as the mechanisms implemented by a series of smart contracts which shall be deployed to the Ethereum mainnet. Query Nodes, Query Clients, and Fisherman Services are "extra-protocol", which is to say that while they may be described in this document to add color, the protocol is agnostic to their specific interfaces and logic, and indeed we expect there to arise multiple implementations with distinct interfaces and logic. There may also arise multiple implementations for the Indexing Nodes, in a variety of languages, however, the service interfaces of each implementation must adhere strictly to the protocol defined in this specification.

Components

Decentralized Application (dApp)

This is an application run by an end user in their browser or on their device. Its data and business logic primarily live on the Ethereum blockchain and IPFS, meaning that it is safe from censorship and the economic risks of the developers shutting down or going out of business. In exchange for this robustness, a user assumes the cost of operating the infrastructure required to power the dApp by paying gas costs to transact against the Ethereum blockchain and making micropayments for metered usage of The Graph to query the data required to power the dApp. The dApp may interact with The Graph via an embedded Query Client or external Query Node.

Query [Nodes | Clients]

Query Nodes provide an abstraction on top of the low-level read API provided by the Indexing Nodes. The Query Nodes may optionally choose to provide a GraphQL interface, SQL interface, or traditional REST interface, whatever is best-suited toward the respective domain in which it will be used. We include a reference JavaScript Query Node that provides a GraphQL interface and may be embedded and extended in a server or browser application as a Query Client.

In addition to providing an interface to dApps, the Query Node is responsible for discovering Indexing Nodes in the network that are indexing a specific dataset, and selecting an Indexing Node to read from based on factors such as price and performance (see Query Processing). It may also optionally forward attestations along to a Fisherman Service.

Indexing Nodes

Indexing Nodes index one or more user-defined datasets, called *subgraphs*. These nodes perform a deterministic streaming extract, transform and load (ETL) of events emitted by the Ethereum blockchain. These events are processed by user-defined logic called *mappings* which run deterministically inside a WASM runtime, and are also able to load additional data from the Ethereum blockchain or IPFS, in order to compute the current state of a subgraph. See Datasets) for more information.

Note: Here, and throughout this document, "event" is used in its standard usage, meaning data which is emitted asynchronously and may act as a trigger for computation. This is to disambiguate from "Solidity events," which build atop Ethereum's low-level logging facilities, and will be referred to throughout this specification as "Solidity events" or "Ethereum logs". Indexing Nodes will process events which include Ethereum logs, new blocks, as well as internal and external Ethereum transactions.

Indexing Nodes implement a standard interface for reading from indexes and to advertise compute and bandwidth prices for read operations. See JSON-RPC API) for full interface.

Fisherman Service

Fisherman Services accept read responses and attestations which they may verify, and in the event of an invalid response, may file a protocol-level dispute (see

Mechanism Design). Note that whether or not the Fisherman Service actually verifies the response is completely opaque to the end-user and the protocol.

In the v1 of the protocol, the Graph Protocol team will operate a Fisherman service.

IPFS

"IPFS" refers to the Interplanetary File Service, a decentralized content-addressed storage network. Data stored on IPFS is identified by a content ID (CID) which is computed by encoding and hashing the content being stored.1 It has become a common pattern to store these CIDs in Ethereum contracts, providing a form of cheap, decentralized, off-chain storage. The Graph will index data linked in IPFS, referenced in Ethereum smart contracts, to support this use case.

The Graph also uses IPFS in this way - subgraph manifests, are stored on IPFS and referenced on-chain in the protocol's smart contracts (see Subgraph Manifest). In the future, we may store indexed data and even query results on IPFS, as having data stored in a global decentralized file system increases chances for reuse, adds redundancy for widely used data and effectively gives you caching for free.

Ethereum

Ethereum is a blockchain that can run small Turing-complete executable programs called smart contracts. Consensus is built around the results of these computations, using a Byzantine fault tolerant (BFT) consensus algorithm, meaning that a centralized actor cannot easily tamper with or rewrite the results of past computations.2

The Ethereum blockchain plays two principle roles in the protocol. First, dApps include business logic implemented as smart contracts deployed to the Ethereum blockchain, which in turn emit events and store data that is indexed by The Graph. Second, the mechanisms that define the incentives and economic security of The Graph are themselves implemented as smart contracts deployed to the Ethereum blockchain.

Footnotes

- [1] https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf
- [2] https://github.com/ethereum/wiki/wiki/White-Paper

Mechanism Design

Overview

The protocol implements a *work token* token model1 in which Indexing Node operators stake deposits of Graph Tokens for particular datasets, called subgraphs, to gain the right to participate in the data retrieval marketplaces for that dataset--indexing data and responding to read requests in exchange for micropayments. This deposit is forfeit in the event that the work is not performed correctly, or is performed maliciously, as defined in the slashing conditions.

There are secondary mechanisms in the protocol that also require a staking of tokens, such as curation, stake delegation, and name registration, all of which will be expanded upon in their respective sections.

Graph Token

We introduce a native token for the protocol, Graph Tokens, which are the only token that may be used for staking in the network. However, ETH or DAI is used for paying for read operations, thus reducing friction and balance sheet risk for end-users of dApps that query The Graph. Graph Tokens will have variable inflation to reward specific activities in the network, as described in Inflation Rewards.

Governance

There are several parameters throughout this mechanism design that are set via a governance process. In the v1 specification, governance will consist of a multi-sig wallet contract controlled by the Graph Protocol team.

In future versions of the protocol, more decentralized forms of governance will explored.

Staking

Indexing Nodes deposit a `stakingAmount` of Graph Tokens to process read requests for a specific dataset, which is identified by its `subgraphID`.

For a `stakingAmount` to be considered valid, it must meet the following requirements:

- `stakingAmount >= minStakingAmount` where `minStakingAmount` is set via governance.

- The `stakingAmount` must be in the set of the top N staking amounts, where N is determined by the `maxIndexers` parameter that is set via governance.

Indexing Nodes that have staked for a dataset are not limited by the protocol in how many read requests they may process for that dataset. However, it may be assumed that Indexing Nodes with higher deposits will receive more read requests and, thus, collect more fees, if all else is equal, as this represents a greater economic security margin to the end user.

Data Retrieval Market

Indexing Nodes which have staked to index a particular dataset, will be discoverable in the data retrieval market for that dataset.

Indexing Nodes compete to have the most compelling combination of economic security margin (the amount of tokens staked), performance and price to attract read requests from users of the network. See Query Processing for an example market interaction.

Indexing Nodes receive requests which include a Read Operation and a Locked Transfer.

The Read Operation fully defines the data that is being requested, while the Locked Transfer is a micropayment that is paid, conditional, on the Indexing Node producing a Read Response along with a signed Attestation message which certifies the response data is correct.

Data Retrieval Pricing

Pricing in the data retrieval market is set according to the bandwidth and compute required to process a request.

Compute is priced as a `gasPrice`, denominated in ETH or DAI, where the `gas` required for a request is determined by the specific read operation and parameters. See Read Interface for operation specific gas prices.

Bandwidth is priced in `bytesPrice`, denominated in ETH or DAI, where `bytes` refers to the size of the `data` portion of the response, measured in bytes.

Indexing Nodes respond with their compute and bandwidth costs in response to the `getPrices` method in the JSON-RPC API.

Verification

Fisherman Service

A Fisherman Service is an economic agent who verifies read responses in exchange for a reward in cases where they detect that an Indexing Node has attested to an incorrect response, and the Fisherman successfully disputes the response on-chain.

In the v1 of the protocol, the Graph Protocol team will operate a Fisherman service. This is to accommodate the fact, that in the absence of forced errors in the v1 protocol, Fisherman rewards should go to zero overtime, and thus must have altruistic motives in order to perform their service.

Dispute Resolution

Dispute resolution is handled through an on-chain dispute resolution process. In future versions of the protocol, this may involve programmatically verifying proofs or using a Truebit-style verification game, but in the v1 specification, the outcome of a dispute will be decided by a centralized arbitrator interacting with the on-chain dispute resolution process.

To dispute a response, a Fisherman must submit the attestation of the response they are disputing as well as a deposit.

TODO Define deposit amount for Fisherman disputes

In the event of a successful dispute the Indexing Node forfeits the entire deposit of tokens they staked on the dataset for which they produced an incorrect response. The Fisherman, in turn, receives a reward equal to a percentage of the slashed deposit.

TODO Define slashing reward for successful disputes

In the event of an unsuccessful dispute, the Fisherman forfeits the entire deposit they submitted with their dispute.

Market Discovery

Market discovery is the process by which Indexing Nodes choose which datasets to index and serve data on.

When the data retrieval market for a particular dataset is active, an Indexing Node may observe payment activity on-chain to decide if it would be profitable to participate in that market.

With little to no activity for a newly created dataset, however, payment activity provides a poor signal. Instead, this signal to the network is provided by a *Curation Market*.2

Curation Market

Curators are economic agents who earn rewards by betting on the future economic value of datasets, perhaps with the benefit of private information.

A Curator stakes a deposit of Graph Tokens for a particular dataset in exchange for dataset-specific *subgraph tokens*. These tokens entitle the holder to a portion of a curation reward, which is paid in Graph Tokens through inflation. See Inflation Rewards for how curation reward is calculated for each dataset.

Subgraph tokens are issued according to a bonding curve, making it more expensive to mint subgraph tokens by locking up Graph Tokens as the amount of bonded tokens increases, thus making it more expensive to purchase a share of future curator inflation rewards.

TODO Define bonding curve for curation market or bonding curve parameters.

Inflation Rewards

The total monetary inflation rate of Graph Tokens over a given inflation period is the sum of its two constituent components:

`inflationRate = curatorRewardRate + participationRewardRate`

As indicated in the formula above, inflation is used to reward curation of datasets and participation in the network.

Participation Adjusted Inflation

To encourage Graph Token holders to participate in the network, the protocol implements a participation-adjusted inflation3 reward.

The participation reward to the entire network is calculated as a function of a `targetParticipationRate` that is set via governance. If `actualParticipationRate == targetParticipationRate`, then `participationRewardRate = 0`. Conversely, the lower the actual participation rate is relative to the target participation rate, the higher the participation reward.

TODO Decide on actual function for relating `participationRewardRate` to `targetParticipationRate`.

To incentivize actual work being provided to the network, not just staking, the participation reward will be distributed to Indexing Nodes who are staking for datasets with the most economic activity.

Let `participationReward[s[i]]` be the participation reward allotted to Indexing Node `i` staked on dataset `s`. Let `transactionVolume[s[i]]` be the economic value of all data retrieval fees paid to Indexing Node `i` for data from dataset `s` over a given inflation period, and `totalTransactionVolume` be the total transaction volume for the entire network over the same period, both measured in ETH. Then, we can define the participation reward as:

Let `participationReward[i]` be the participation reward allotted to Indexing Node `i`. Let `aggregateTransactionValue[i]` be the total value of all data retrieval fees paid to Indexing Node `i` over a given inflation period, and `totalTransactionValue` be the total value of all transactions in the network over the same period, both measured in ETH. Then we can define the participation reward as:

```
participationReward[i] = (aggregateTransactionValue[i] / totalTransactionValue)
* participationRewardRate * totalTokenSupply.
```

Curator Inflation Reward

The `curationRewardRate` is defined as a percentage of the total Graph Token supply and is set via governance. As with the participation reward, it is paid via inflation.

Let `aggregateTransactionValue[s]` be the total value of all transactions in the data retrieval market for a dataset `s` over a given inflation period. We can define `curationReward[s]`, the total curation reward shared by all Curators of a dataset `s` over a given inflation period:

```
curatorReward[s] = (aggregateTransactionValue[s] / totalTransactionValue)
* curationRewardRate * totalTokenSupply
```

The share of this dataset-specific curation reward that an individual Curator receives is determined according to the share of the curator reward they acquired in the curation market.

Inflation Periods

Inflation rewards are calculated over a period of time, measured in blocks, according to a `roundDuration` parameter that is set via governance.

TODO How should round duration be set to balance gas costs and facilitating a dynamic market?

For a given round `R`, the inflation rewards for that round are made available at the end of round `R+1`.

This provides adequate time for off-chain micropayments to be settled on-chain. This settlement on-chain also provides a market signal. So, `roundDuration` should be set sufficiently small to provide a good market signal, but sufficiently large to reduce the amount of on-chain transactions required to redeem inflation rewards on an on-going basis.

Stake Delegation

Participation in the protocol is a specialized activity. In the case of Curators, it entails accurately predicting the future value of datasets to the network, while in the case of Indexing Nodes, it requires operating infrastructure to index and serve data.

Token holders who do not feel equipped to perform one of these functions may *delegate* their tokens to an Indexing Node that is staked for a particular dataset. In this case, the delegator is the residual claimant for their stake, earning participation rewards according to the activities of the delegatee Indexing Node but also forfeiting their stake in the event that the delagatee Indexing Node is slashed.

Footnotes

- [1] https://multicoin.capital/2018/02/13/new-models-utility-tokens/
- [2] https://medium.com/@simondlr/introducing-curation-markets-trade-popularity-of-memes-information-with-code-70bf6fed9881
- [3] https://medium.com/@petkanics/inflation-and-participation-in-stake-based-token-protocols-1593688612bf

Query Processing

Background

In some respects, The Graph resembles a traditional distributed query engine, where users may retrieve data that is distributed across a variety of stores via a single query interface, typically SQL.

In this analogy, Query Nodes play the role of query engine, and Indexing Nodes play the role of data stores.

Notably, the protocol only defines the *read interface* to the Indexing Nodes, which is consumed by the Query Nodes, while remaining agnostic to the actual implementation of the Query Nodes.

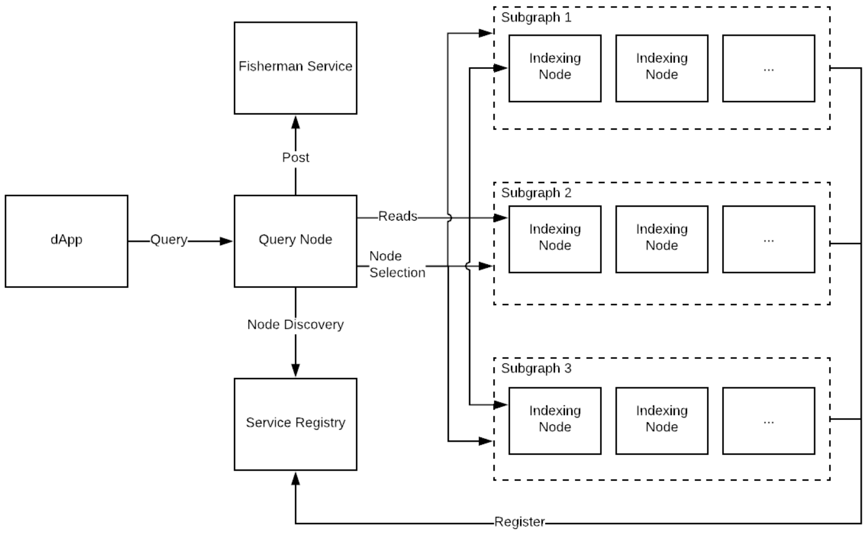
Some Query Node implementations may provide an SQL interface, while others may provide a GraphQL interface. Query Nodes may implement different heuristics for balancing the requirements of price, performance, and economic security or make these algorithms pluggable. Indeed, some users may choose to forgo the Query Node altogether and directly consume the lower-level read interface exposed by the Indexing Nodes.

While this last case is certainly possible, we present the architecture and high-level algorithms for consuming the data indexed by The Graph via a distributed query engine because that is the intended usage pattern of the protocol.

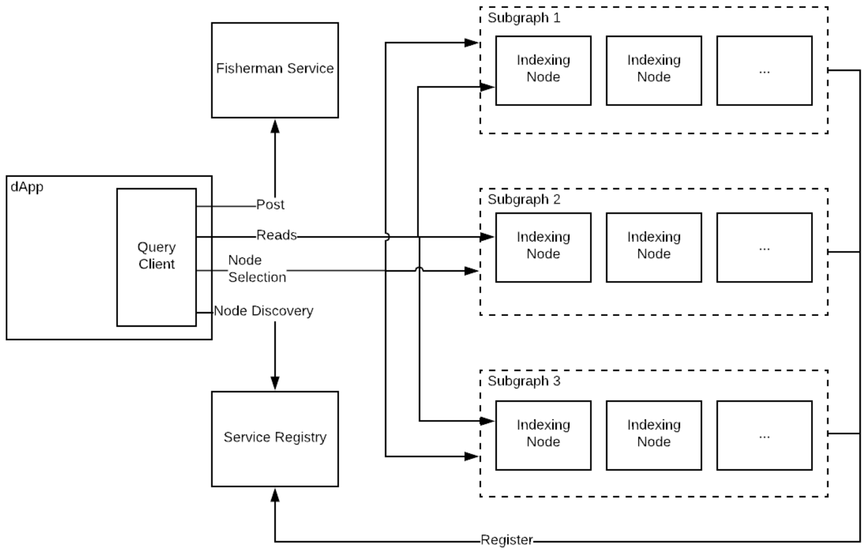
The specification will omit detailed steps in the algorithms that are left to implementers. However, the Graph Protocol team will implement a reference Query Node/Client that provides a concrete example of how these algorithms might be implemented in order to provide a GraphQL interface to The Graph.

Query Processing Architecture

With Query Node



With Query Client



Overview

As shown in the diagrams above, the query processing may take place via a Query Client, which is embedded in the end-user application, or it may take place via a Query Node that is external to the application. In the latter case, the Query Node may be running locally on the user's machine or as an external service that is accessed via the Internet.

In either construction, query processing consists of the following steps:

1. Query Planning (Optional)
2. Service Discovery
3. Service Selection
4. Processing and Payment
5. Response Collation

Design

Query Planning

In this stage, the Query Node transforms a query into a plan, consisting of an ordered set of lower-level read operations that may be used to retrieve the data specified by the query. These steps are encapsulated in some intermediate representation (IR).

Implementor's Note
Query planning is optional. For example, producing a query plan is common for most SQL databases, but for

Query Optimization

Query plans may optionally be optimized based on a variety of heuristics and algorithms, which are out of the scope of this specification.

Service Discovery

Processing a query plan, or processing a query directly, results in low-level read operations being made to Indexing Nodes. Each read operation corresponds to a specific dataset and, thus, needs to be made against an Indexing Node for that dataset. In the Service Discovery step, the Query Node locates Indexing Nodes for a specific dataset as well as important metadata that is useful in deciding which Indexing Node to issue read operations to, such as price, performance, and economic security margin.

Locating Indexing Nodes

To locate Indexing Nodes with data for a specific dataset, the Query Node makes several calls to the service discovery layer, which is implemented as several smart contracts on the Ethereum mainnet:

1. Resolve subgraph names via the Graph Name Service (GNS).
2. Identify per-subgraph Indexing Nodes via the Staking Contract.
3. Identify Indexing Node URLs via the Service Registry.

Collecting Indexing Node Metadata

After identifying the URLs of all Indexing Nodes for a given dataset, the next step is to collect the metadata for price, performance, and economic security margin. This information should be cached for future Service Discovery steps for subsequent queries.

Fetching price and latency for a node is done via a single call to the Indexing Node RPC API and returns the following data: the latency required to fulfill the request; a `bandwidthPrice` measured in price per byte transmitted over the network; and a `gasPrice`, which captures the cost of compute and IO for a given read operation.

Economic security margin is the amount that an Indexing Node has staked and is willing to forfeit in the event that they provide an incorrect response to a read operation. The Query Node receives this in the previously made call to the Staking Contract.

Implementor's Note
The Query Node does not need to make calls to every Indexing Node for a given dataset. It could choose to c

Service Selection

In the Service Selection stage, Query Nodes choose which Indexing Nodes to transact with for each read operation. An algorithm for this stage could incorporate `latency` (measured in ms), `economicSecurityMargin` (measured in Graph Tokens), `gasPrice`, and `bytesPrice` (the cost of sending a byte over the network).

A naive algorithm for service selection could look like the following:

1. Filter Indexing Nodes where `economicSecurityMargin < minEconomicSecurityMargin`.
    - If no Indexing Nodes remain, return an error to the sender of the query for this piece of data, and specify the reason.
2. Filter Indexing Nodes where `latency < minLatency`.
    - If no Indexing Nodes remain, increase `minLatency` by 33%, and repeat the step.
3. Estimate the cost of the read operation for each remaining Index.
    - Assume 80% of the maximum possible entities returnable by the query will be returned.
    - Assume 25% of the max field size (in bytes) for each entity field with variable size.
    - Calculate the bandwidth and gas costs based on the above assumptions and the gas costs specified in the Read Interface.
4. Choose the Indexing Node with the lowest estimated cost for the read operation.

In this example algorithm, `minLatency` and `minEconomicSecurityMargin` could be set per dataset or for all datasets. Additionally, it could be set by the Query Node or sent as metadata with an individual query.

Processing and Payment

Available read operations are defined in the Read Interface, and are sent to the Indexing Nodes via the JSON-RPC API. They are accompanied by Locked Transfers, conditional micropayments that may be unlocked by the Indexing Node producing a Read Response and a signed Attestation message certifying the response data is correct.

v1-spec: Update links and
minor edits

Response Collation

Once all read operations have been processed, the resulting data must be collated into a response that fulfills the read schema of the query interface provided by the Query Node. This response is then returned to the sender of the query.

Payment Channels

Overview

The protocol adopts payment channels, as a means of facilitating micropayments that are paid to Indexing Nodes in exchange for reading from indexes.

The protocol's payment channel architecture follows the Raiden Network Specification1, with a few notable differences:

1. The Graph v1 implements a hub-and-spoke topology that dramatically simplifies payment routing, compared to a fully distributed network topology.
2. The Graph v1 introduces a new concept, *minting channels*, to get around prohibitively large balance requirements for the payment channel hub.
3. The Graph uses an alternate locking mechanism for mediated payments that is tailored to the domain of reading data from indexes.
4. Payment channels are one-way and may be withdrawn from by payment receiver without closing channel. See Payment Channel(#payment-channel).
5. Balance Proofs may be exchanged off-chain directly between the sender and final recipient of a mediate transfer. See Micropayment Routing.

In this construction, the Payment Channel Hub acts as a *trusted intermediary*, although users of the protocol have no counter-party risk with respect to one another. A future version of this protocol will move away from the hub and spoke topology in favor of a decentralized payment channel network topology. At that point, minting channels will also likely be removed from the specification.

Hub-and-Spoke Topology

Architecture

```
+-------------------+-------------------+------------------+
|                   |                   |                  |
|                   |                   |                  |
  +----+-----+        +----+-----+       +-----+----+             |
  |          |        |          |       |          |             |
  | Indexing |        | Indexing |       | Indexing |             |
  | Node     |        | Node     |       | Node     |             |
  |          |        |          |       |          |             |
  +-----^----+        +----^-----+       +-----^----+         Sell GRT/
        |                  |                   |              Buy ETH
        |                  |                   |                  |
        |                  |                   |                  |
        +---GRT-----+     GRT     +-----GRT----+                  |
                    |      |      |                               |
 minting channels   |      |      |                               |
                    |      |      |                               |
                +---+------+------+---+                  +--------v------+
                |                     |      Sell ETH/   |               |
                | Payment Channel Hub +------------------> Token Auction |
                |                     |      Buy GRT     |               |
                +---^------^------^---+                  +---------------+
                    |      |      |
 payment channels   |      |      |
                    |      |      |
      +-----ETH-----+     ETH     +----ETH---+
      |                    |                 |
      |                    |                 |
+-----+------+      +------+-----+     +-----+------+
|            |      |            |     |            |
|  End user  |      |  End user  |     |  End user  |
|            |      |            |     |            |
+------------+      +------------+     +------------+
```

High-Level Design

End users pay Indexing Nodes via the Payment Channel Hub. Micropayments from end users to the Payment Channel Hub are denominated in ETH or DAI, while micropayments from the Payment Channel Hub to Indexing Nodes are denominated in Graph Tokens, which the Payment Channel Hub mints.

To determine the exchange rate between ETH or DAI and Graph Tokens, the Payment Channel Hub reads from an on-chain Token Auction contract that acts as a price feed. The Token Auction contract also acts as a sink for the Graph Tokens that are minted by the Payment Channel Hub and provides a mechanism for selling the ETH or DAI that the Payment Channel Hub collects.

Payment Channel Hub

The Payment Channel Hub is a service, an externally owned Ethereum account operated by the Graph Protocol Team. It acts as a counter party for payment channels with end users and for minting channels with Indexing Nodes.

To act as a counterparty for the minting channel contract, the Ethereum account corresponding to the Payment Channel Hub must be designated as a *treasurer* of the Graph Token (GRT) ERC-20 contract, which grants the hub the right to mint Graph Tokens.

Payment Channel

The payment channel presented here is modeled off the payment channel design in the Raiden Specification2 with the key difference that payments are only made in one direction from the end-user to the Payment Channel Hub. This difference enables other simplifications in the design:

1. Balance proofs may be settled on-chain continuously, without closing the channel.
2. Tokens may be withdrawn from the channel by the Payment Channel Hub continuously, without closing the channel.

As is the case with normal payment channel contracts, deposits may be made by participants, specifically the end-user, on an ongoing basis. For the end-user to withdraw tokens that they deposited, the channel must be closed and settled.

Minting Channel

Traditional payment channels involve exchanging off-chain messages that are "backed" by a deposit in a channel on-chain, which may be used to settle the final balance when the payment channel is closed. We present a variation on this construction, where instead of being backed by a deposit, payments in the channel are backed by the ability of one participant, the sender, to mint the token that the micropayments are denominated in.

Specifically, the Payment Channel Hub has the ability to mint Graph Tokens to pay Indexing Nodes an amount equivalent to the amount of ETH or DAI paid toward that Indexing Node by an end user. The minting channel acts as the second leg of a mediated transfer.

The minting channel should be settled once per *round*. See Mechanism Design for more information.

Micropayment Routing

Because of the hub-and-spoke payment channel, micropayment routing is trivial. Payment must always go through the Payment Channel Hub, and only deposits in the first leg of the mediate payment need be checked to confirm that there are sufficient funds to cover the transfer. As such, balance proofs may be exchanged directly between sender and final recipient of a mediated micropayment, and the receiver may send messages to the payment channel hub on the sender's behalf. This facilitates sending valid Locked Transfer messages in-band with requests to the JSON RPC API.

Token Auction

With the minting channel construction, new Graph Tokens are minted in direct proportion to the amount of value exchanged between end users and Indexing Nodes in a given round. The Token Auction acts as a sink for these new Graph Tokens, whereby the Payment Channel "buys back" the Graph Tokens previously minted in exchange for the ETH or DAI collected in the payment channels through an on-chain auction mechanism.

TODO Select on-chain auction/ price feed mechanism.

The Token Auctions implemented as smart contracts on the Ethereum blockchain also act as on-chain price feeds, indicating an exchange rate between the supported tokens and GRT. This may be used by the Payment Channel Hub to determine the amount of Graph Tokens to send on the second leg of the mediated transfer via the minting channel.

Data Retrieval Timelock

Traditional mediated transfers via payment channels use a hash timelock, in which payments are unlocked by providing the pre-image to a hash. In The Graph, micropayments are unlocked by the Indexing Node performing useful work, that of reading from an index, and providing an attestation that the work was performed correctly. Rather than having a fixed amount of tokens locked, the amount of tokens unlocked are dynamic, based on the amount of bandwidth and computation required to fulfill the request, and maximum computation and bandwidth limits which is defined in the lock.

See Data Retrieval Timelock in the Messages section of the specification.

Footnotes

- [1] https://github.com/raiden-network/spec

- [2] https://raiden-network-specification.readthedocs.io/en/latest/smart_contracts.html#tokennetwork-channel-protocol-overview

Read Interface

Overview

To participate in the data retrieval market, Indexing Nodes implement a low-level read interface to the indexed data in their store. The read interface not only provides the means of retrieving data from an Indexing Node, but it also defines a contract that an Indexing Node is agreeing to uphold or else be slashed. This is enabled by attestations, which assert that a response was produced correctly and may be verified on-chain.

Calling Read Operations

Available read operations are defined by the respective interface of the index being read from. See Index Abstract Data Structures and Index Types for more information.

While the read interfaces are described using a TypeScript notation, all the interfaces are language agnostic and defined in terms of JSON types.

Calling these read operations is done via JSON RPC 2.01. See the full JSON RPC API.

The method of interest here is `callReadOp` which accepts the following parameters:

1. `Object`

- `blockHash`: `String` - The hash of the Ethereum block from which to read the data.
- `subgraphID`: `String` - The ID of the subgraph to read from.
- `index`: `Object` - The IndexRecord of the index being read from.
- `op`: `String` - The name of the read operation.
- `params`: `[any]` - The parameters passed into the called read operation.

2. `Object` - A Locked Transfer message which serves as a conditional micropayment for the read operation.

The `readIndex` method returns the following:

1. `Object`

- `data`: `any` - The data retrieved by the read operation.

- `attestation`: Object - An attestation that `data` is a correct response for the given read operation (see Attestation).

```
// request
{
  "method": "readIndex",
  "params": [
    {
      "blockHash": "xbf133b670857b983fc1b8f08759bc860378179042a0dba30b30e26d6f7f919d1",
      "subgraphID": "QmTeW79w7QQ6Npa3b1d5tANreCDxF2iDaAPsDvW6KtLmfB",
      "index": {
        "indexType": "kv"
      },
      "op": "get"
      "params": ["User:1"]
    }
  ],
  "jsonrpc": "2.0"
}
// response
{
  "data": {
    "firstName": "Vitalik",
    "lastName": "Buterin",
  },
  // TODO: Provide more realistic attestations
  "attestation": 0x0122340
}
```

Example - Entity doesn't exist

```
// request
{
  "method": "readIndex",
  "params": [
    {
      "blockHash": "xbf133b670857b983fc1b8f08759bc860378179042a0dba30b30e26d6f7f919d1",
      "index": {
        "indexType": "kv"
      },
      "op": "get"
      "params": ["User:1"]
    }
  ],
  "jsonrpc": "2.0"
}
```

```
// response
{
  "data": null,
  // TODO: Provide more realistic attestations
  "attestation": 0x0122340
}
```

Indexes

All read operations require that the caller specify an index. Index data structures efficiently organize the data to support different read access patterns.

Indexes may include the entire dataset or cover only a subset. This is useful for enabling sharding, where different Indexing Nodes may store different subsets of the dataset to reduce the storage requirements for a single Indexing Node or enable better read performance.

Indexes are defined by an `IndexRecord` which has the following shape:

Field Name	Field Type	Description
db	String	The identifier of the database model being used.
indexType	String	An identifier of the index type used for the respective database model.
partition	String	The name of the entity or interface which should be covered by the index.
options	Object	Options specific to the type of index.

Example Index Records

Given a dataset with the following data model:

```
interface EthereumAccount {
  id: ID!
  address: String!
}

type Contract implements EthereumAccount {
  id: ID!
  address: String!
}

type User implements EthereumAccount {
  id: ID!
  address: String!
  name: FullName
}
```

```
type FullName {
  first: String!
  last: String!
}
```

Then, the following would be valid index names for that dataset:

Index Name
`{ db: "entitydb", indexType: "dictionary" }`
`{ db: "entitydb", "indexType: "dictionary", partition: "User" }`
`{ db: "entitydb", indexType: "searchTree", options: { sortBy: ["id"] } }`
`{ db: "entitydb", indexType: "searchTree", partition: "EthereumAccount", options: { sortBy:`
`{ db: "entitydb", indexType: "searchTree", partition: "User",  options: { sortBy: ["name.fi`

Index Abstract Data Structures

All concrete index types implement an indexing abstract data structure, which specify the interface, semantics, and gas costs for read operations against that index.

The concrete types (i.e., `K` and `V` shown below), as well as the implicit comparator function to determine sort order, are specified by each concrete index type.

Dictionary

Type

Dictionary<K,V>

Operations

Op	Signature	Description	Gas Cost
get	`(key: K) => V`	Retrieves a value by its key.	`opCostDictionaryGet` (set via governance)

Search Tree

Type

`SearchTree<K,V>`

Operations

Op	Signature
find	`(predicate: FilterPredicate, options?: { gte?: K, lt?: K } ) => V`
findLast	`(predicate: FilterPredicate, options?: { gt?: K, lte?: K } ) => V`
get	`(key: K) => V`
take	`(count: Number, options?: { skip?: Number, gte?: K, lt?: K}) => [V]`
takeUntil	`(predicate: FilterPredicate, options?:{ skip?: Number, gte?: K, lt?: K}) => [`
takeWhile	`(predicate: FilterPredicate, options?:{ skip?: Number, gte?: K, lt?: K}) => [`
takeLast	`(count: Number, options?: { skip?: Number, gt?: K, lte?: K })`
takeLastUntil	`(predicate: FilterPredicate, options?:{ skip?: Number, gt?: K, lte?: K}) => [`
takeLastWhile	`(predicate: FilterPredicate, options?:{ skip?: Number, gt?: K, lte?: K}) => [`

Filter Predicates

Filter predicates allow for declaratively asserting whether a value meets certain criteria. Filter predicates are expressed as objects that can be passed into several low-level index read operations, such as `takeWhile` and `find`.

Structure

Filter predicates are expressed through a simple DSL:

```
type FilterPredicate = FilterPredicateAnd | FilterPredicateOr | FilterPredicateLeaf

interface FilterPredicateAnd {
  and: [FilterPredicate];
}

interface FilterPredicateOr {
  or: [FilterPredicate];
}

type FilterPredicateLeaf = StringFilter | NumberFilter | BooleanFilter

interface BaseFilter {
  // The field the predicate will be applied to. Nested fields may be
  // specified by concatenating field names with a "."
  // If no field is specified, the predicate will be applied to the value. This
  // is only supported if the value is a primitive type.
  field?: String;
}

// If multiple filter clauses are supplied, they will be treated as a logical AND.
interface StringFilter extends BaseFilter {
  equals?: String;
  notEquals?: String;
  // Contains string
  contains?: String;
  // Does not contain string
  notContains?: String;
  startsWith?: String;
  notStartsWith?: String;
  endsWith?: String;
  notEndsWith?: String;
  // Less than
  lt?: String;
  // Less than or equal to
  lte?: String;
  // Greater than
  gt?: String;
  // Greater than or equal to
  gte?: String;
  // Contained in list
  in?: [String];
  // Not contained in list
  notIn?: [String];
}


// If multiple filter clauses are supplied, they will be treated as a logical AND.
interface NumberFilter extends BaseFilter {
  equals?: Number;
  notEquals?: Number;
  // Less than
  lt?: Number;
  // Less than or equal to
  lte?: Number;
  // Greater than
  gt?: Number;
  // Greater than or equal to
  gte?: Number;
  // Contained in list
  in?: [Number];
  // Not contained in list
  notIn?: [Number];
}

// If multiple filter clauses are supplied, they will be treated as a logical AND.
interface BooleanFilter extends BaseFilter {
  equals?: Boolean;
  notEquals?: Boolean;
```

```
}
```

Example - Simple Value Filter Predicate

```
{
  equals: 12
}
```

Example - Object Filter Predicate

```
{
  field: "fullName",
  contains: "Vitalik"
}
```

Example - Filter Predicate with Boolean Operators and Nested Fields

```
{
 and: [
   {
     field: "name.first",
     equals: "Vitalik"
   },
   {
     field: "name.last",
     equals: "Buterin"
   }
 ]
}
```

Gas Cost

The clauses in the filter predicate DSL can be grouped into several buckets of operation types, which share equivalent gas cost calculations:

Operation Type	Description
Number Comparison	Includes `lt`, `lte`, `gt`, `gte`, `equals` and `notEquals` clauses on Number types.
String Comparison	Includes `lt`, `lte`, `gt`, `gte`, `startsWith`, `notStartsWith`, `endsWith`, `notEndsWith`, `equ`
Bit Comparison	Includes `equals` and `notEquals` clauses on Boolean types. Also used for combining tw
String Match	Used for `contains` and `notContains` clauses on String types

Database Models

The semantics of reading from an Indexing Node are determined by the database model that the index being read from implements, such as key-value (KV), entity-attribute-value (EAV) and the relational model. Index types are prefixed with a short label indicating the database model the index implements:

- `entitydb` - An entity database model.
- `rdb` - Relational database model. Not supported in this version of the protocol.
- `eav` - Entity-attribute-value database model. Not supported in this version of the protocol.

The database model also defines the available partitions and index types for use in read operations.

In the v1 protocol, we only support the entity database model.

Entity Database Model

In the protocol's entity database model, entities are stored as key-value pairs, where the key is a concatenation of the entity type and the entity ID, and the value is an entity object.

This database model is referenced as `entitydb` in Index Records.

Example - Entities Stored as Key-Value Pairs

Key	Value
`user:1`	`{ id: "user:1", user: "Alice", age: 17 }`
`user:2`	`{ id: "user:2", user: "Bob", age: 47 }`

Partitions

Partitions define the subset of the data that is covered by the index.

Possible values of `<partition>` in an Index Record:

- none - Includes all entities in the dataset. Default partition, `partition` key should be ommitted in IndexRecord.
- `"<entityType>"` - Includes entities of the type specified by `entityType`. The entity type name is case-sensitive.
- `"<interface>"` - Includes entities that implement the provided interface. The interface name is case-sensitive.

Index Types

Entity DB Indexes

Dictionary

The entity dictionary supports simple key-value lookups of entities by their entity type and ID, in constant time.

Name

`dictionary`

Database Model

`entitydb`

Type

`Dictionary<K, V>`

- `K`: `String` - The id of the entity.
- `V`: `Object` - An entity that conforms to its type as defined in the schema of the dataset.

Options

None

Search Tree

This index supports iterating through entities, ordered by possibly nested attribute values. Supports compound indexes, where an entity is sorted first by one attribute, then by another.

Name

`searchTree`

Database Model

`entitydb`

Type

`SearchTree<K, V>`

- `K`: `String` | `Number` | `Object` - The value of the sortKey, which is either a primitive value in the case of single-attribute indexes, or an object containing two attribute-value pairs in the case of compound indexes.
- `V`: An entity that conforms to its type as defined in the schema of the dataset.

Options

- `sortBy`: `Array`
    1. `String` - The first attribute to sort by, using `.` to indicate nested attributes (i.e., `"name.first"`)
    2. `String` - The second attribute to sort by, using `.` to indicate nested attributes (i.e., `"name.last"`)

Footnotes

- [1] https://www.jsonrpc.org/specification
- [2] https://github.com/multiformats/multicodec

Messages

Off-Chain Messages

Encoding

Off-chain messages are encoded using JSON1, a light-weight data interchange format, and the mostly commonly used format for exchanging data on the web.

Off-chain messages may be referenced in an on-chain message via a Content ID (CID). These are produced according to the IPLD CID V1 specification2.

CIDs must use the canonical CBOR encoding3, and SHA-256 multi-hash.

In producing CIDs for JSON RPC messages, the optional `id` field from the JSON-RPC 2.0 specification should be omitted, as well as the optional conditional micropayment in the `readIndex` params list.

Message Types

Read Operation

Fields

Field Name	Field Type	Description
blockHash	String	The hash of the Ethereum block as of which to read the data.
subgraphID	String	The ID of the subgraph to read from.
index	Object	The Index Record corresponding to the index being read from.
op	String	The name of the read operation.
params	Array	The parameters passed into the called read operation.

Index Record

Fields

Field Name	Field Type	Description
db	String	The identifier of the database model being used.
indexType	String	An identifier of the index type used for the respective database model.
partition	String	The name of the entity or interface which should be covered by the index.
options	Array	Parameters specific to the type of index.

Locked Transfer

A message intended to be exchanged off-chain as a conditional micropayment in the data retrieval market for a subgraph. Accompanied by a Payment Channel Balance Proof which may be redeemed on-chain.

Fields

Field Name	Field Type	Description
chainID	Number	EIP155 chain ID.
tokenDenomination	String	Token denomination. Must be "ETH" or "DAI".
transferredAmount	Number	A monotonically increasing amount of tokens which have been sent in the
receiver	String	The Ethereum address of the final destination of the micropayment. Must
subgraphID	String	The ID of the subgraph for which the receiver must be staked.
maxLockedAmount	Number	The maximum amount of tokens locked in pending transfers.
locksRoot	String	The root of a Merkle tree containing all locked data retrieval timelocks.
lock	Object	The Off-chain Data Retrieval Timelock corresponding to the most recent
nonce	Number	A monotonically increasing nonce value starting at `1`. Used for strictly or
v	Number	The ECDSA recovery ID of the corresponding Payment Channel Balance
r	String	The ECDSA signature r of the corresponding Payment Channel Balance
s	String	The ECDSA signature v of the corresponding Payment Channel Balance

Off-chain Data Retrieval Timelock

An off-chain representation of the Data Retrieval Timelock

Fields

Field Name	Field Type	Description
expiration	Number	The block until which the locked transfer may be settled on-chain.
gasPrice	Number	Amount of tokens locked.

Field Name	Field Type	Description
maxGas	Number	The maximum amount of gas to be consumed in the read operation.
bytesPrice	Number	The price to pay per byte served.
maxBytes	Number	The maximum amount of bytes to be sent over the wire
maxTokens	Number	The maximum amount of tokens to be paid.
requestCID	String	The content ID of the read operation to which the Indexing Node must respond w

Read Response

Fields

There are several possible types for a read response:

Success

Sent if the read operation was successful, within the gas and response size limits specified. Includes the return data an attestation that the response is correct.

Field Name	Field Type	Description
status	String	The constant "SUCCESS"
data	any	The result of calling the read operation.
attestation	Object	An Attestation, where the `requestCID` is the CID of the object containing the ab

Max Gas Exceeded

Sent if the maximum amount of gas specified was consumed before the read operation could complete. The caller of the read operation is responsible for paying for the computation, but not for any bandwidth.

Field Name	Field Type	Description
status	String	The constant "MAX_GAS_EXCEEDED"
attestation	Object	An Attestation, where the `requestCID` is the CID of the object containing the ab

Max Bytes Exceeded

Sent if the result of calling the read operation is larger than the `maxBytes` parameter in the data retrieval timelock. The caller of the read operation is responsible for paying for the computation, but not for any bandwidth.

Field Name	Field Type	Description
status	String	The constant "MAX_BYTES_EXCEEDED"

Insufficient Funds

Sent if the maximum amount of tokens which may be consumed by the read operation would exceed the balance in the payment channel.

Field Name	Field Type	Description
status	String	The constant "INSUFFICIENT_FUNDS".

Price Too Low

Sent if the Indexing Node is unwilling to provide the service at the prices offered by the caller.

Field Name	Field Type	Description
status	String	The constant "PRICE_TOO_LOW".
askingPrice	Object	A price listing object.

Price Listing

A price listing advertising an Indexing Nodes asking price for computation and bandwidth, denominated in a specific token.

Field Name	Field Type	Description
token	String	
gasPrice	Number	The price of a unit of gas, denominated in the token included in the listing. Must
bytesPrice	Number	The price per byte in the read operation result data, denominated in the token in

On-Chain Messages

Encoding

Unsigned messages are encoded according to the ABIv2 specification4, while signed messages are encoded according to [EIP 712 specification5.

Signed message formats are accompanied by a typed structured data definition, which can be used to compute the type, the type hash, and the data of a message according to the EIP 712 specification. Types are written as Solidity code, but are intended to be compatible with any language that compiles to EVM bytecode.

EIP 712 Domain Separator

The EIP 712 specification requires defining a domain separator to disambiguate signed messages intended for different chains, different protocols, or different versions of the same protocol.

The domain separator for the protocol has the following chain-agnostic parameters:

- **name** - 'graphprotocol'
- **version** - '0'

Additionally there are chain-specific parameters:

- mainnet
    - **chainid** - 1
    - **verifyingContract** - TBD
- ropsten
    - **chainid** - 3
    - **verifyingContract** - TBD
- kovan
    - **chainid** - 42
    - **verifyingContract** - TBD
- rinkeby
    - **chainid** - 4
    - **verifyingContract** - TBD

Message Types

Attestation

Fields

Field Name	Field Type	Description
requestCID	bytes	The content ID of the message.
responseCID	bytes	The content ID of the response.
gasUsed	uint256	The gas used to process the read operation.
responseBytes	uint256	The size of the response data in bytes.
v	uint256	The ECDSA recovery ID .
r	bytes32	The ECDSA signature r.
s	bytes32	The ECDSA signature v.

EIP712 Struct Type

```
struct Attestation {
  bytes requestCID;
  bytes responseCID;
  uint256 gasUsed;
  uint256 responseBytes;
}
```

Payment Channel Balance Proof

The Payment Channel Balance Proof is a signed off-chain message which represents a micropayment between an end user of The Graph and the Payment Channel Hub via a payment channel. Because all payment channels have the Payment Channel Hub as the receiver, it is sufficient to be able to identify the token denomination and the sender's Ethereum address (this may be derived from the signature), as well as the subgraph on which they are staked, to uniquely identify the channel to which the balance proof applies.

Fields

Field Name	Field Type	Description
chainID	uint256	EIP155 chain ID.
tokenDenomination	string	Token denomination. Must be "ETH" or "DAI".
transferredAmount	uint256	A monotonically increasing amount of tokens which have been sent in the
receiver	address	The Ethereum address of the final destination of the micropayment. Must
subgraphID	bytes	The ID of the subgraph for which the receiver must be staked.
maxLockedAmount	uint256	The maximum amount of tokens locked in pending transfers.
locksRoot	bytes32	The root of a Merkle tree containing all locked data retrieval timelocks.
nonce	uint256	A monotonically increasing nonce value starting at `1`. Used for strictly or
v	uint256	The ECDSA recovery ID .
r	bytes32	The ECDSA signature r.
s	bytes32	The ECDSA signature v.

Minting Channel Balance Proof

The Minting Channel Balance Proof is a signed off-chain message which represents a micropayment between the Payment Channel Hub and an Indexing Node in The Graph. Because all payment channels have the Payment Channel Hub as the sender, it is sufficient to be able to identify the token denomination and the receiver's Ethereum address to uniquely identify the channel to which the balance proof applies.

Fields

Field Name	Field Type	Description
chainID	uint256	EIP155 chain ID.
tokenDenomination	string	Token denomination. Must be "ETH" or "DAI".
transferredAmount	uint256	A monotonically increasing amount of tokens which have been sent in the

Field Name	Field Type	Description
maxLockedAmount	uint256	The maximum amount of tokens locked in pending transfers.
locksRoot	bytes32	The root of a Merkle tree containing all locked data retrieval timelocks.
nonce	uint256	A monotonically increasing nonce value starting at `1`. Used for strictly or
v	uint256	The ECDSA recovery ID .
r	bytes32	The ECDSA signature r.
s	bytes32	The ECDSA signature v.

EIP712 Struct Type

```
struct MintingChannelBalanceProof {
  uint256 chainID;
  address tokenNetworkAddress;
  uint256 channelID;
  uint256 transferredAmount;
  uint256 maxLockedAmount;
  bytes32 locksRoot;
  uint256 nonce;
}
```

Data Retrieval Timelock

Fields

Field Name	Field Type	Description
expiration	uint256	The block until which the locked transfer may be settled on-chain.
gasPrice	uint256	Amount of tokens locked.
maxGas	uint256	The maximum amount of gas to be consumed in the read operation.
bytesPrice	uint256	The price to pay per byte served.
maxBytes	uint256	The maximum amount of bytes to be sent over the wire
maxTokens	uint256	The maximum amount of tokens to be paid.
requestCID	bytes	The content ID of the read operation to which the Indexing Node must respond w

Footnotes

- [1] http://json.org
- [2] https://github.com/ipld/cid#cidv1
- [3] https://tools.ietf.org/html/rfc7049#section-3.9
- [4] https://solidity.readthedocs.io/en/develop/abi-spec.html
- [5] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-712.md

JSON RPC API

This API uses JSON RPC 2.01, a light-weight, transport agnostic RPC protocol.

Methods

- getPrices
- ping
- callReadOp

Reference

getPrices

Retrieves the gas and bandwidth pricing of an Indexing Node in a specific token denomination. Prices returned are informational only, they do not represent a commitment by the Indexing Node.

Parameters

1. `String` (optional) - The symbol of the token which prices are being requested in. Valid values are 'ETH' or 'DAI'. If not specified, prices will be returned for all tokens the node denominates prices in.

Returns

`Array<Object>`

- `token`: `String` - The symbol of the token which the prices are denominated in.
- `gasPrice`: `Number` | `null` - The price of a unit of gas. If no price denominated in the specified token, `null`.
- `bandwidthPrice`: `Number` | `null` - The price of sending one byte over the network. If no price denominated in the specified token, `null`.

Example

```
{
  "method": "getPrices",
  "params": ["DAI"],
  "jsonrpc": "2.0"
}
```

```
// response
{
  "result": [
    {
      "token": "DAI",
      "bandwidthPrice": 0.01,
      "gasPrice": 0.025

    }
  ],
  "jsonrpc": "2.0"
}
```

ping

Pings a node to check that is it is available and gauge the latency of the ’pong’ response.

Parameters

None

Returns

`String` - The string "pong".

Example

```
// request
{
  "method": "ping",
  "jsonrpc": "2.0"
}

// response
{
  "result": "pong"
  "jsonrpc": "2.0"
}
```

callReadOp

Calls a low-level read operation on a database index.

Parameters

1. `Object`

- `blockHash`: `String` - The hash of the Ethereum block as of which to read the data.
- `subgraphID`: `String` - The ID of the subgraph to read from.
- `index`: `Object` - The Index Record of the index being read from.
- `op`: `String` - The name of the read operation.
- `params`: `[any]` - The parameters passed into the called read operation.

2. `Object` - A Locked Transfer message which serves as a conditional micropayment for the read operation.

Returns

Returns one of the following message types:

Read Result

`Object`

- `type`: `String` - The constant "READ_RESULT"
- `data`: `any` - The data retrieved by the read operation, if any.
- `attestation`: Object - An attestation that `data` and `type` is a correct response for the given read operation (see Attestation).

Not Enough Gas

Indicates that that the gas limit was consumed without completing the computation. Payment will still be made to the Indexing Node for computation performed. No data is returned. `Object`

- `type`: `String` - The constant "NOT_ENOUGH_GAS"
- `attestation`: Object - An attestation that `type` is a correct response for the given read operation (see Attestation).

Not Enough Bandwidth

Indicates that that the bandwidth limit is insufficient to cover the response size. Payment will still be made to the Indexing Node for computation performed (but not for bandwidth). No data is returned. `Object`

- `type`: `String` - The constant "NOT_ENOUGH_BANDWIDTH"
- `attestation`: Object - An attestation that `type` is a correct response for the given read operation (see Attestation).

Insufficient Funds

Indicates that that there are insufficient funds in the payment channel to cover the maximum amount of tokens that may be spent completing the read operation. `Object`

- `type`: `String` - The constant "INSUFFICIENT_FUNDS"

Price Too Low

Indicates that the price offered for gas or bandwidth is lower than what the Indexing Node will accept. Response includes up-to-date prices. `Object`

- `type`: `String` - The constant "PRICE_TOO_LOW"
- `prices`: `Object` - The currently advertised prices for the Indexing Node
  - `token`: `String` - The symbol of the token which the prices are denominated in.
  - `gasPrice`: `Number` | `null` - The price of a unit of gas. If no price denominated in the specified token, `null`.
  - `bandwidthPrice`: `Number` | `null` - The price of sending one byte over the network. If no price denominated in the specified token, `null`.

Must Include Payment

Indicates that the Indexing Nodes expects a conditional micropayment to be included with the request.

`Object`

- `type`: `String` - The constant "MUST_INCLUDE_PAYMENT"

Example

Example - Entity exists

```
// request
{
  "method": "callReadOp",
  "params": [
    {
      "blockHash": "xbf133b670857b983fc1b8f08759bc860378179042a0dba30b30e26d6f7f919d1",
      "subgraphID": "QmTeW79w7QQ6Npa3b1d5tANreCDxF2iDaAPsDvW6KtLmfB",
      "index": {
        "indexType": "kv"
      },
      "op": "get"
      "params": ["User:1"]
    }
  ],
  "jsonrpc": "2.0"
}
// response
{
  "data": {
    "firstName": "Vitalik",
    "lastName": "Buterin",
  },
  // TODO: Provide more realistic attestations
  "attestation": 0x0122340
}
```

Example - Entity doesn't exist

```
// request
{
  "method": "callReadOp",
  "params": [
    {
      "blockHash": "xbf133b670857b983fc1b8f08759bc860378179042a0dba30b30e26d6f7f919d1",
      "index": {
        "indexType": "kv"
      },
      "op": "get"
      "params": ["User:1"]
    }
  ],
  "jsonrpc": "2.0"
}
// response
{
  "data": null,
  // TODO: Provide more realistic attestations
  "attestation": 0x0122340
}
```

Footnotes

- [1] https://www.jsonrpc.org/specification

Datasets

Object Diagram

```
           +--------------+     +-------------------+
           |              |     |                   |
Ethereum   | GNS Registry |     | Staking Contract  |
           |              |     | (Indexing Records)|
           |              |     |                   |
           +--------------+     +-------------------+
                  |1                     |1
                  |                      |
+----------------------------------------------------------------------------------+
                  |                      |
                  |     +-------------+  |  +------------+
                  |     |             |  |  |            |
                  |    *|  Subgraph   | *|  |  Index     |
IPFS              +----->  Manifest   +--v--+  Records   |
                        |             |1   *|            |
                        +-----+-------+     +------------+
                              |
                 +------------+----------+
                 |                       |
        +--------v-------+       +-------v-------+
        |                |       |               |
        | Mapping        |       | Data Model    |
        | (WASM Module)  |       | (GraphQL IDL) |
        |                |       |               |
        +----------------+       +---------------+
```

Overview

Datasets that may be queried through The Graph are referred to as *subgraphs* because they represent a subset of the data that is available to query in the network. Subgraphs are defined in a *subgraph manifest*, which is a top-level IPLD document that defines how Ethereum and IPFS data is ingested and loaded into The Graph. Importantly, while the subgraph manifest includes a logical data model for the dataset, it does not specify a specific storage format, database model, or index method. These are defined as *Index Records* and are associated with a subgraph manifest by indexing nodes in the Staking Contract on-chain.

Subgraph manifests are immutable and referenced according to the IPLD CID v1 specification. This CID is referred to in this specification as a *Subgraph ID*. Mutable names may be assigned to subgraph IDs via the Graph Name Service

(GNS). These are not consumed on-chain anywhere in the protocol and are mainly a convenience for users interacting with The Graph. These names may also be used in composing a unified global schema in the query interface of Query Nodes. See Query Processing for more information. In future versions of the protocol, names will play a more useful role in various forms of the subgraph composition.

Subgraph Creation

Creating a subgraph involves the following steps, in no specific order:

- Create subgraph manifest (Subgraph Manifest)
- Define data model (Data Modeling)
- Define mappings (Mappings API)

Subgraph Deployment

Deploying a subgraph involves the following steps:

1. Deploy subgraph manifest to IPLD and get subgraph ID.
2. Curate or index the subgraph, referenced by subgraph ID, in the Staking Contract (Mechanism Design).
3. (Optional) Associate a human-friendly name with the SubgraphID in the Graph Name Service.

Data Modeling

The schema of your dataset--that is, the entity types, values and relationships that are available to query--are defined through the GraphQL Interface Definition Language (IDL)1.

Entities

Entities

Entities are defined as GraphQL types decorated with an `@entity`. All entities must have an `id: ID!` field defined.

Example

Define a `Token` entity:

```
@entity
type Token {
  # The unique ID of this entity
  id: ID!
  name: String!
  symbol: String!
  decimals: Int!
}
```

An attribute on an entity type may be specified as unique, in which case the value for that attribute must be unique among all instances of that entity type.

Example

Define a `File` entity with a unique content hash:

```
@entity
type File {
  id: ID!
  name: String!
  bytes: Bytes!
  length: Int!
  # Only one File entity may be created with a given
  # content hash.
  hash: String! @unique
}
```

Built-In Types

GraphQL Built-In Scalars

All the scalars defined in the GraphQL spec are supported: `Int`, `Float`, `String`, `Boolean`, and `ID`.

Bytes

There is a `Bytes` scalar for variable-length byte arrays.

Additionally, fixed-length byte scalar types between 1 and 32 bytes are supported: `Byte`, `Bytes1` (an alias for `Byte`), `Bytes2`, `Bytes3` ... `Bytes31`, and `Bytes32`.

Numbers

The GraphQL spec defines `Int` and `Float` to have sizes of 32 bytes.

This API additionally includes `BigInt` and `BigFloat` number types to represent arbitrarily large integer or floating point numbers, respectively.

There are also fixed-size number types to represent numbers between 1 and 32 bytes long. The suffix is specified by the number of bits.

Signed integers all share the `Int` prefix: `Int8`, `Int16`, `Int24`, `Int32` (an alias of `Int`) ... `Int240`, `Int248`, and `Int256`.

There are corresponding unsigned integer types prefixed with `UInt`: `UInt8`, `UInt16`, `UInt24`, `UInt32` ... `UInt240`, `UInt248`, and `UInt256`.

All number types other than `Int` and `Float`, which are serialized as JSON number types, are serialized as strings.

Even though the serialization format is the same, having the sizes captured in the type system provides better self-documentation and enables tooling that generates convenient deserializers in statically typed languages.

Value Objects

All types not decorated with the `@entity` decorator are value objects. Value object types may be used as the type of entity attributes, and do not have unique `id` attributes themselves.

Entity Relationships

An entity may have a relationship to one or more other entities in your data model. Relations are unidirectional.

Despite being unidirectional, attributes may be defined on entities which facilitate navigating relationships in the reverse direction. See Reverse Lookups.

Basics

Relationships are defined on entities just like any other scalar type, except that the type specified is that of another entity.

Example

Define a `Transaction` entity type with an optional one-to-one relationship with a `TransactionReceipt` entity type:

```
@entity
type Transaction {
  id: ID!
  transactionReceipt: TransactionReceipt
}

@entity
type TransactionReceipt {
  id: ID!
  transaction: Transaction
}
```

Example

Define a `Token` entity type with a required one-to-many relationship with a `TokenBalance` entity type:

```
@entity
type Token {
  id: ID!
  tokenBalances: [TokenBalance!]!
}

@entity
type TokenBalance {
  id: ID!
  amount: Int!
}
```

Reverse Lookups

Reverse lookups can be defined on an entity through the `@derivedFrom` field. This creates a "virtual" field on the entity that may be queried but cannot be set manually through the mappings API. Rather, it is derived from the relationship defined on the other entity.

The type of a `@derivedFrom` field must be a collection since multiple entities may specify relationships to a single entity.

Example

Define a reverse lookup from a `User` entity type to an `Organization` entity type:

```
@entity
type Organization {
  id: ID!
  name: String!
  members: [User]!
}
```

```
@entity
type User {
  id: ID!
  name: String!
  organizations: [Organization!] @derivedFrom(field: "members")
}
```

Footnotes

- [1] http://facebook.github.io/graphql/draft/#sec-Type-System

Subgraph Manifest

TODO This file has moved to https://github.com/graphprotocol/graph-node. Please confirm. Datasets references this file with ../subgraph-manifest, so it needs to remain in this repo.

Overview

The Subgraph manifest specifies all the information required to index and query a specific subgraph. It is the entry point to your subgraph, so to speak.

The subgraph manifest, and all the files linked from it, are what is deployed to IPFS, and hashed to produce a subgraph ID that can be referenced on Ethereum and used to retrieve your subgraph in The Graph.

Format

The subgraph manifest follows the IPLD specification, which defines a data model for linking decentralized and universally addressable data structures.1 Supported formats include YAML and JSON. All examples in this section are written as YAML.

Top-Level API

Field	Type	Description
specVersion	*String*	A semver version indicating which version of this API is being used.
schema	*Schema*	The GraphQL schema of this subgraph
dataSources	*[Data Source Spec]*	Each Data Source spec defines data which will be ingested, and transfor

Schema

Field	Type	Description
file	*Path*	The path of the GraphQL IDL file, either locally or on IPFS

Data Source

Field	Type	Description
kind	*String	The type of data source. Possible values: *ethereum/contract*
name	*String*	The name of the source data. Will be used to generate APIs in mapp
source	*EthereumContractSource*	The source data on a blockchain such as Ethereum
mapping	*Mapping*	The transformation logic applied to the data prior to being indexed

EthereumContractSource

Field	Type	Description
address	*String*	The address of the source data in its respective blockchain
abi	*String*	The name of the ABI for this Ethereum contract (see `abis` in `mapping` manifest)

Mapping

The `mapping` field may be one of the following supported mapping manifests:

- Ethereum Events Mapping

Ethereum Events Mapping

Field	Type	Description
kind	*String*	Must be "ethereum/events" for Ethereum Events Mapping
apiVersion	*String*	Semver string of the version of the Mappings API which will be used by t
language	*String*	The language of the runtime for the Mapping API. Possible values: *wasm*
entities	*[String]*	A list of entities which will be ingested as part of this mapping. Must cor
abis	*ABI*	ABIs for the contract classes which should be generated in the Mapping A
eventHandlers	*EventHandler*	Handlers for specific events, which will be defined in the mapping script
file	*Path*	The path of the mapping script

EventHandler

Field	Type	Description
event	*String*	An identifier for an event which will be handled in the mapping script. For Ethereum cor
handler	*String*	The name of an exported function in the mapping script which should handle the specifie

Path

A path has one field `path` which either refers to a path of a file on the local dev machine, or an IPLD link.

When using the Graph-CLI, local paths may be used during development, and then the tool will take care of deploying linked files to IPFS and replacing the local paths with IPLD links at deploy time.

Field	Type	Description
path	*String or IPLD Link*	A path to a local file or an IPLD link

Footnotes

- [1] https://github.com/ipld/specs
- [2] https://github.com/ipld/specs/blob/master/Codecs/DAG-JSON.md

Mappings API

Overview

Mappings define how data is extracted or ingested from one or more data sources, transformed, and then loaded in a format that follows a specific data model. At their core, mappings are simply WASM modules, and the mappings API is a set of host external functions that are injected into the WASM runtime and implement a specific interface. These function interfaces are in-protocol.

Additionally, extra-protocol APIs may be defined in userspace, which implement an API in a higher-level language that compiles to WASM. The Graph Protocol team created one such API, which is included here as a reference example.

WASM API

The Mappings API can be split into two portions, the *ingest* or *extract* API and the *store* API (how data is loaded). We present an ingest API tailored to event-sourcing Ethereum smart contract data, but future versions of the protocol will enable ingest APIs specific to other decentralized data sources. The *store API* will not need to change to support new data sources.

Note: There are also a number of utility functions that are currently injected into the WASM runtime for convenience. These will either be implemented natively in WASM or fully specified in a future version of this spec.

Ingest

Ethereum

Data is ingested from Ethereum by event-sourcing Solidity events, as well as other triggers, defined in the subgraph manifest. The WASM module referenced in the subgraph manifest is expected to have handlers that correspond to the handlers defined in the subgraph manifest.

See this reference implementation for how these handlers should be called.

Additionally, we inject functions for calling Ethereum smart contracts for additional data that is not included in the Ethereum event:

- ethereum_call

See this reference for these additional functions.

Store

The store API includes the following methods:

- set
- get
- remove

See this reference implementation for these external functions.

Utilities

We currently inject the following utility functions into the WASM runtime, which may be changed or removed in a future version of the protocol:

- bytes_to_string
- bytes_to_hex
- big_int_to_hex
- big_int_to_i32
- json_to_i64
- json_to_u64
- json_to_f64
- json_to_big_int

- crypto_keccak_256
- big_int_plus
- big_int_minus
- big_int_times
- big_int_divided_by
- big_int_mod
- string_to_h160

See this reference implementation for these external functions.

Higher-Level APIs

Higher-level APIs provide context for the low-level APIs, described above, in a higher-level programming language that compiles to WASM.

AssemblyScript

AssemblyScript is a subset of TypeScript that compiles to WASM. It only natively supports a handful of types, 32 and 64-bit floating point and integer numeric types, but we extend the runtime with additional higher-level types, such as `TypedMap` and `BigInt`, to facilitate a more developer-friendly API.

See this reference for all types, external functions, and utilities.

Types

Basic Types

- `TypedMap<K, V>`
- `TypedMapEntry<K, V>`
- `BytesArray`
- `Bytes`
- `BigInt`
- `Value`
- `ValueKind`
- `ValuePayload`

Serialization Formats

- `JSONValue`
- `JSONValueKind`
- `JSONValuePayload`

Ethereum Types

- `EthereumValue`
- `EthereumValueKind`
- `EthereumBlock`
- `EthereumTransaction`
- `EthereumEvent`
- `EthereumEventParams`
- `SmartContractCall`
- `SmartContract`

Store Types

- `Entity`

Ingest

Ethereum

- `ethereum.call`

IPFS

- `ipfs.cat`

Store

- `store.set`
- `store.get`
- `store.remove`

Utilities

- `crypto.keccak256`
- `json.fromBytes`
- `json.toI64`
- `json.toU64`
- `json.toF64`
- `json.toBigInt`
- `typeConversion.bytesToString`
- `typeConversion.bytesToHex`
- `typeConversion.bigIntToString`
- `typeConversion.stringToH160`
- `typeConversion.i32ToBigInt`
- `typeConversion.bigIntToI32`

- `bigInt.plus`
- `bigInt.minus`
- `bigInt.times`
- `bigInt.dividedBy`
- `bigInt.mod`