Polkadot Networking

Protocol Specification

Contents

Document Status: This document in its current form is incomplete and considered work in progress. Any reports regarding falseness or clarifications are appreciated.

0.1 Introduction

The Polkadot network is decentralized and does not rely on any central authority or entity in order to achieve a its fullest potential of provided functionality. Each node with the network can authenticate itself and its peers by using cryptographic keys, including establishing fully encrypted connections. The networking protocol is based on the open and standardized `libp2p` protocol, including the usage of the distributed Kademlia hash table for peer discovery.

0.1.1 External Documentation

The completeness of implementing the Polkadot networking protocol requires the usage of external documentation.

- lipp2p
- Kademlia
- Noise
- Protocol Buffers

0.1.2 Discovery mechanism

The Polkadot Host uses varies mechanism to find peers within the network, to establish and maintain a list of peers and to share that list with other peers from the network.

The Polkadot Host uses various mechanism for peer dicovery.

- Bootstrap nodes - hard-coded node identities and addresses provided by network configuration itself. Those addresses are selected an updated by the developers of the Polkadot Host. Node addresses should be selected based on a reputation metric, such as reliability and uptime.
- mDNS - performs a broadcast to the local network. Nodes that might be listing can respond the the broadcast.
- Kademlia requests - Kademlia supports `FIND_NODE` requests, where nodes respond with their list of available peers.

0.1.3 Connection establishment

The Polkadot Host can establish a connection with any peer it knows the address. `libp2p` uses the `multistream-select` protocol in order to establish an encryption and multiplexing layer. The Polkadot Host supports multiple base-layer protocols:

- TCP/IP - addresses in the form of `/ip4/1.2.3.4/tcp/` establish a TCP connection and negotiate a encryption and multiplexing layer.

- Websockets - addresses in the form of `/ip4/1.2.3.4/ws/` establish a TCP connection and negotiate the Websocket protocol within the connection. Additionally, a encryption and multiplexing layer is negotiated within the Websocket connection.

- DNS - addresses in form of `/dns/website.domain/tcp/` and `/dns/website.domain/ws/`.

After a base-layer protocol is established, the Polkadot Host will apply the Noise protocol.

0.1.4 Substreams

After the node establishes a connection with a peer, the use of multiplexing allows the Polkadot Host to open substreams. Substreams allow the negotiation of *application-specific protocols*, where each protocol servers a specific utility.

The Polkadot Host adoptes the following, standardized `libp2p` application-specific protocols:

- `/ipfs/ping/1.0.0` - Open a substream to a peer and initialize a ping to verify if a connection is till alive. If the peer does not respond, the connection is dropped.

- `/ipfs/id/1.0.0` - Open a substream to a peer to ask information about that peer.

- `/dot/kad/` - Open a substream for Kademlia `FIND_NODE` requests.

Additional, non-standardized protocols:

- `/dot/sync/2` - a request and response protocol that allows the Polkadot Host to perform information about blocks.

- `/dot/light/2` - a request and response protocol that allows a light client to perform information about the state.

- `/dot/transactions/1` - a notification protocol which sends transactions to connected peers.

- `/dot/block-announces/1` - a notification protocol which sends blocks to connected peers.

0.2 Network Messages

0.2.1 API Package

ProtoBuf details:

- syntax: proto3

- package: api.v1

BlockRequest

Request block data from a peer.

```
message BlockRequest {
    // Bits of block data to request.
    uint32 fields = 1;
    // Start from this block.
    oneof from_block {
        // Start with given hash.
        bytes hash = 2;
        // Start with given block number.
        bytes number = 3;
    }
    // End at this block. An implementation defined
    // maximum is used when unspecified.
    bytes to_block = 4; // optional
    // Sequence direction.
    Direction direction = 5;
    // Maximum number of blocks to return. An implementation
    // defined maximum is used when unspecified.
    uint32 max_blocks = 6; // optional
}

// Block enumeration direction
enum Direction {
    // Enumerate in ascending order
    // (from child to parent).
    Ascending = 0;
    // Enumerate in descending order
    // (from parent to canonical child).
    Descending = 1;
}
```

BlockResponse

Response to Block Request.

```
message BlockResponse {
    // Block data for the requested sequence.
    repeated BlockData blocks = 1;
}

// Block data sent in the response.
message BlockData {
    // Block header hash.
    bytes hash = 1;
```

```
    // Block header if requested.
    bytes header = 2; // optional
    // Block body if requested.
    repeated bytes body = 3; // optional
    // Block receipt if requested.
    bytes receipt = 4; // optional
    // Block message queue if requested.
    bytes message_queue = 5; // optional
    // Justification if requested.
    bytes justification = 6; // optional
    // True if justification should be treated as present but
    // empty. This hack is unfortunately necessary because
    // shortcomings in the protobuf format otherwise doesn't
    // make it possible to differentiate between a lack of
    // justification and an empty justification.
    bool is_empty_justification = 7; // optional, false if absent
}
```

0.2.2 Light Package

ProtoBuf details:

- syntax: proto3
- package: api.v1.light

Request

Enumerate all possible light client request messages.

```
message Request {
    oneof request {
        RemoteCallRequest remote_call_request = 1;
        RemoteReadRequest remote_read_request = 2;
        RemoteHeaderRequest remote_header_request = 3;
        RemoteReadChildRequest remote_read_child_request = 4;
        RemoteChangesRequest remote_changes_request = 5;
    }
}
```

Response

Enumerate all possible light client response messages.

```
message Response {
    oneof response {
        RemoteCallResponse remote_call_response = 1;
```

```
        RemoteReadResponse remote_read_response = 2;
        RemoteHeaderResponse remote_header_response = 3;
        RemoteChangesResponse remote_changes_response = 4;
    }
}
```

RemoteCallRequest

Remote call request.

```
message RemoteCallRequest {
    // Block at which to perform call.
    bytes block = 2;
    // Method name.
    string method = 3;
    // Call data.
    bytes data = 4;
}
```

RemoteCallResponse

Remote call response.

```
message RemoteCallResponse {
    // Execution proof.
    bytes proof = 2;
}
```

RemoteReadRequest

Remote storage read request.

```
message RemoteReadRequest {
    // Block at which to perform call.
    bytes block = 2;
    // Storage keys.
    repeated bytes keys = 3;
}
```

RemoteReadResponse

Remote read response.

```
message RemoteReadResponse {
    // Read proof.
    bytes proof = 2;
}
```

RemoteReadChildRequest

Remote storage read child request.

```
message RemoteReadChildRequest {
    // Block at which to perform call.
    bytes block = 2;
    // Child Storage key, this is relative
    // to the child type storage location.
    bytes storage_key = 3;
    // Storage keys.
    repeated bytes keys = 6;
}
```

RemoteHeaderRequest

Remote header request.

```
message RemoteHeaderRequest {
    // Block number to request header for.
    bytes block = 2;
}
```

RemoteHeaderResponse

Remote header response.

```
message RemoteHeaderResponse {
    // Header. None if proof generation has failed
    // (e.g. header is unknown).
    bytes header = 2; // optional
    // Header proof.
    bytes proof = 3;
}
```

RemoteChangesRequest

Remote changes request.

```
message RemoteChangesRequest {
    // Hash of the first block of the range (including first)
    // where changes are requested.
    bytes first = 2;
    // Hash of the last block of the range (including last)
    // where changes are requested.
    bytes last = 3;
    // Hash of the first block for which the requester has
```

```
    // the changes trie root. All other
    // affected roots must be proved.
    bytes min = 4;
    // Hash of the last block that we can use when
    // querying changes.
    bytes max = 5;
    // Storage child node key which changes are requested.
    bytes storage_key = 6; // optional
    // Storage key which changes are requested.
    bytes key = 7;
}
```

RemoteChangesResponse

Remote changes response.

```
message RemoteChangesResponse {
    // Proof has been generated using block with this number
    // as a max block. Should be less than or equal to the
    // RemoteChangesRequest::max block number.
    bytes max = 2;
    // Changes proof.
    repeated bytes proof = 3;
    // Changes tries roots missing on the requester' node.
    repeated Pair roots = 4;
    // Missing changes tries roots proof.
    bytes roots_proof = 5;
}

// A pair of arbitrary bytes.
message Pair {
    // The first element of the pair.
    bytes fst = 1;
    // The second element of the pair.
    bytes snd = 2;
}
```

0.2.3 Finality Package

ProtoBuf details:

- syntax: proto3
- package: api.v1.finality

FinalityProofRequest

Request a finality proof from a peer.

```
message FinalityProofRequest {
    // SCALE-encoded hash of the block to request.
    bytes block_hash = 1;
    // Opaque chain-specific additional request data.
    bytes request = 2;
}
```

FinalityProofResponse

Response to a finality proof request.

```
message FinalityProofResponse {
    // Opaque chain-specific finality proof.
    // Empty if no such proof exists.
    bytes proof = 1; // optional
}
```