Communication in Geth (2)

QnA session

Node Discovery via DNS

> Ref: EIP-1459

Many Ethereum clients contain hard-coded bootstrap node lists. Updating those lists requires a software update and, effort is required from the software maintainers to ensure the list is up to date. **As a result, the provided lists are usually small, giving the software little choice of initial entry point into the network.**

To enhance current state, EIP-1459, which suggests "Node Discovery via DNS", is proposed. This specification describes a scheme for authenticated, updateable **node lists retrievable via DNS**. In order to use such a list, the client only requires information about the DNS name and the public key that signs list.

Difference between discv4 & discv5

discv5 enhances privacy, efficiency, and flexibility than discv4.

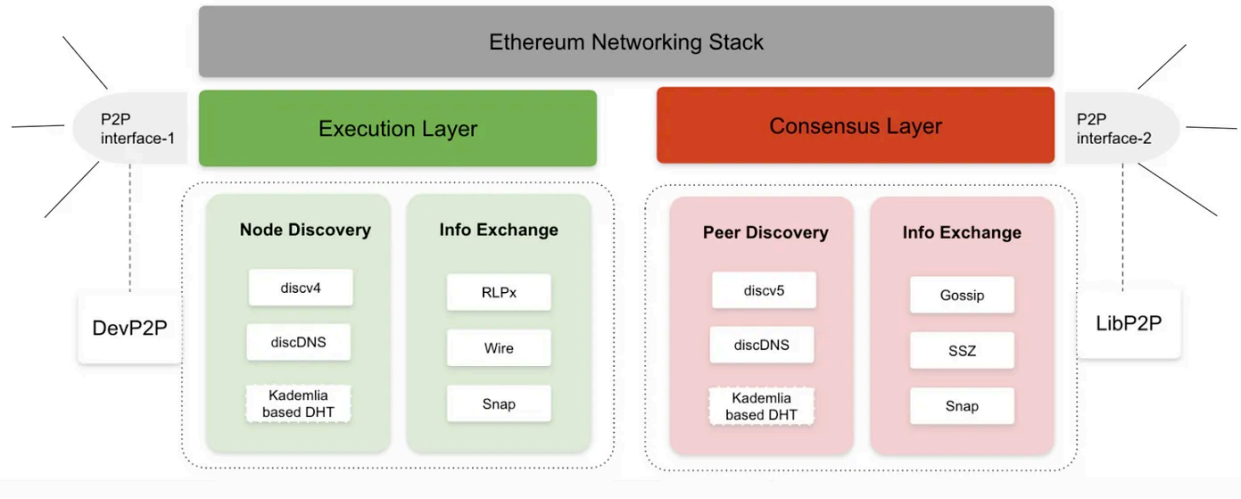
1. Topic-based Node Discovery
    - Discovery v5 introduces a topic-based node discovery mechanism. This allows nodes to advertise and search for other nodes based on specific topics (like shard IDs in Ethereum 2.0), facilitating more efficient and relevant peer discovery.
    - This is particularly important for Ethereum 2.0's sharding mechanism, where nodes might want to connect to peers in the same shard or with specific roles.
2. Enhanced Privacy Features

- Discovery v5 includes improved privacy features. It uses a ticketing system to reduce the likelihood of nodes being tracked or targeted based on their discovery requests.
- This system can mitigate certain types of Eclipse attacks (where an attacker monopolizes all of the victim's peer connections).

3. Integration with libp2p
    - Discovery v5 includes improved privacy features. It uses a ticketing system to reduce the likelihood of nodes being tracked or targeted based on their discovery requests.
    - This system can mitigate certain types of Eclipse attacks (where an attacker monopolizes all of the victim's peer connections).
4. ENR Extensions
    - The protocol uses Ethereum Node Records for storing and sharing information about nodes. Discovery v5 extends the ENR format to support more flexible and diverse node information.
    - This extension allows for the inclusion of additional metadata about nodes, which can be used for more effective network management and peer selection.
5. Improved NAT(Network Address Translation) Traversal
    - The protocol uses Ethereum Node Records for storing and sharing information about nodes. Discovery v5 extends the ENR format to support more flexible and diverse node information.
    - This extension allows for the inclusion of additional metadata about nodes, which can be used for more effective network management and peer selection.
6. Stateless Protocol Design
    - The protocol is designed to be more stateless than v4, reducing the amount of state that nodes need to maintain about their peers.
    - This can lead to lower resource consumption and improved scalability.
7. Security Enhancements

- Discovery v5 incorporates additional security enhancements, including better resistance to spamming and denial of service attacks.
- These improvements are critical for maintaining the integrity and reliability of the network.

DevP2P



After new nodes enter the network, their interactions are governed by protocols in the DevP2P stack.

They sit top of TCP and include RLPx protocol, wire protocol, and several sub-protocols.

RLPx Protocol

RLPx is the main transport protocol used in DevP2P. It provides **a framework for encrypted, authenticated communication between nodes**. RLPx uses the RLP(Recursive Length Prefix) encoding for data serialization and is secured by the

Elliptic Curve Integrated Encryption Scheme(ECIES) for privacy and integrity of communications.

- transport layer
- secure communication
- multiplexing
- framework for sub-protocols

> RLP is a very space-efficient method of encoding data into a minimal structure for sending between nodes.

Initial Handshake

An RLPx connection is established by creating a TCP connection and agreeing on ephemeral key material for further encrypted and authenticated communication. **The process of creating those session keys is the 'handshake' and is carried out between the 'initiator'(the node which opened the TCP connection) and the 'recipient'(the node which accepted it).**

1. initiator connects to recipient and send its `auth` message
2. recipient accepts, decrypts and verifies `auth`
3. recipient generates `auth-ack` message
4. recipient derives secrets and sends the first encrypted frame containing the Hello message
5. initiator receives `auth-ack` and derives secrets
6. initiator send its first encrypted frame containing initiator Hello message
7. recipient receives and authenticates first encrypted frame
8. initiator receives and authenticates first encrypted frame
9. cryptographic handshake is complete if MAC(message authentication code) of first encrypted frame is valid on both sides

**** Hello message**

After the initial handshake, both sides of the connection must send either Hello message or a Disconnect message.

Hello message is first packet sent over the connection, and sent once by both sides. No other messages may be sent until a Hello is received.

`[protocolVersion: P, clientId: B, capabilities, listenPort: P, nodeKey: B_64, ...]`

```
// go-ethereum/p2p/server.go

// conn wraps a network connection with information gathered
// during the two handshakes.
type conn struct {
    fd net.Conn
    transport
    node  *enode.Node
    flags connFlag
    cont  chan error // The run loop uses cont to signal errors to SetupConn.
    caps  []Cap      // valid after the protocol handshake
    name  string     // valid after the protocol handshake
}

type transport interface {
    // The two handshakes.
    doEncHandshake(prv *ecdsa.PrivateKey) (*ecdsa.PublicKey, error)
    doProtoHandshake(our *protoHandshake) (*protoHandshake, error)
    // The MsgReadWriter can only be used after the encryption
    // handshake has completed. The code uses conn.id to track this
    // by setting it to a non-nil value after the encryption handshake.
    MsgReadWriter
    // transports must provide Close because we use MsgPipe in some of
    // the tests. Closing the actual network connection doesn't do
    // anything in those tests because MsgPipe doesn't use it.
    close(err error)
```

```
}

func (srv *Server) setupConn(c *conn, flags connFlag, dialDest *enode.Node) er
    // Prevent leftover pending conns from entering the handshake.
    srv.lock.Lock()
    running := srv.running
    srv.lock.Unlock()
    if !running {
        return errServerStopped
    }

    // If dialing, figure out the remote public key.
    if dialDest != nil {
        dialPubkey := new(ecdsa.PublicKey)
        if err := dialDest.Load((*enode.Secp256k1)(dialPubkey)); err != nil {
            err = fmt.Errorf("%w: dial destination doesn't have a secp256k1 public ke
            srv.log.Trace("Setting up connection failed", "addr", c.fd.RemoteAddr(), "c
            return err
        }
    }

    // Run the RLPx handshake.
    remotePubkey, err := c.doEncHandshake(srv.PrivateKey)
    if err != nil {
        srv.log.Trace("Failed RLPx handshake", "addr", c.fd.RemoteAddr(), "conn", c
        return fmt.Errorf("%w: %v", errEncHandshakeError, err)
    }
    if dialDest != nil {
        c.node = dialDest
    } else {
        c.node = nodeFromConn(remotePubkey, c.fd)
    }
    clog := srv.log.New("id", c.node.ID(), "addr", c.fd.RemoteAddr(), "conn", c.flag
    err = srv.checkpoint(c, srv.checkpointPostHandshake)
    if err != nil {
        clog.Trace("Rejected peer", "err", err)
```

```
        return err
    }

    // Run the capability negotiation handshake.
    phs, err := c.doProtoHandshake(srv.ourHandshake)
    if err != nil {
        clog.Trace("Failed p2p handshake", "err", err)
        return fmt.Errorf("%w: %v", errProtoHandshakeError, err)
    }
    if id := c.node.ID(); !bytes.Equal(crypto.Keccak256(phs.ID), id[:]) {
        clog.Trace("Wrong devp2p handshake identity", "phsid", hex.EncodeToStrin
        return DiscUnexpectedIdentity
    }
    c.caps, c.name = phs.Caps, phs.Name
    err = srv.checkpoint(c, srv.checkpointAddPeer)
    if err != nil {
        clog.Trace("Rejected peer", "err", err)
        return err
    }

    return nil
}
```

```
// go-ethereum/p2p/transport.go

func (t *rlpxTransport) doEncHandshake(prv *ecdsa.PrivateKey) (*ecdsa.PublicK
    t.conn.SetDeadline(time.Now().Add(handshakeTimeout))
    return t.conn.Handshake(prv)
}
```

```
// go-ethereum/p2p/rlpx/rlpx.go

// Handshake performs the handshake. This must be called before any data is wr
// or read from the connection.
```

```
func (c *Conn) Handshake(prv *ecdsa.PrivateKey) (*ecdsa.PublicKey, error) {
    var (
        sec Secrets
        err error
        h   handshakeState
    )
    if c.dialDest != nil {
        sec, err = h.runInitiator(c.conn, prv, c.dialDest)
    } else {
        sec, err = h.runRecipient(c.conn, prv)
    }
    if err != nil {
        return nil, err
    }
    c.InitWithSecrets(sec)
    c.session.rbuf = h.rbuf
    c.session.wbuf = h.wbuf
    return sec.remote, err
}

// runInitiator negotiates a session token on conn.
// it should be called on the dialing side of the connection.
//
// prv is the local client's private key.
func (h *handshakeState) runInitiator(conn io.ReadWriter, prv *ecdsa.PrivateKey,
    h.initiator = true
    h.remote = ecies.ImportECDSAPublic(remote)

    authMsg, err := h.makeAuthMsg(prv)
    if err != nil {
        return s, err
    }
    authPacket, err := h.sealEIP8(authMsg)
    if err != nil {
        return s, err
    }
```

```
    if _, err = conn.Write(authPacket); err != nil {
        return s, err
    }

    authRespMsg := new(authRespV4)
    authRespPacket, err := h.readMsg(authRespMsg, prv, conn)
    if err != nil {
        return s, err
    }
    if err := h.handleAuthResp(authRespMsg); err != nil {
        return s, err
    }

    return h.secrets(authPacket, authRespPacket)
}

// runRecipient negotiates a session token on conn.
// it should be called on the listening side of the connection.
//
// prv is the local client's private key.
func (h *handshakeState) runRecipient(conn io.ReadWriter, prv *ecdsa.PrivateKe
    authMsg := new(authMsgV4)
    authPacket, err := h.readMsg(authMsg, prv, conn)
    if err != nil {
        return s, err
    }
    if err := h.handleAuthMsg(authMsg, prv); err != nil {
        return s, err
    }

    authRespMsg, err := h.makeAuthResp()
    if err != nil {
        return s, err
    }
    authRespPacket, err := h.sealEIP8(authRespMsg)
```

```
    if err != nil {
        return s, err
    }
    if _, err = conn.Write(authRespPacket); err != nil {
        return s, err
    }

    return h.secrets(authPacket, authRespPacket)
}
```

1. p2p server starts (encrypted) handshake w/ its own private key
2. If destination of outgoing connection is nil, it means current node is receiving the connection. It starts handshake as a recipient.
3. If destination of outgoing connection is not a nil, it means current node is initiating the connection. It starts handshake as a initiator.
4. **1)handshake as an initiator:** make authenticated msg → encrypt authMsg → send it to the receiver → extract connection secrets from the handshake value
5. **2)handshake as a receiver:** decode and read encrypted msg → make response to authMsg and encode it → send it to the initiator → extract connection secrets from the handshake value
6. Initialize Secrets during handshake process. Secrets structure contains keys and other cryptographic material necessary for securing the connection.

Wire Protocol

Initially, the wire protocol defined three main tasks:

- chain synchronization
- block propagation
- transactions exchange

However, once Ethereum switched to pos, block propagation and chain synchronization became part of the consensus layer. Transaction exchange is still

the remit of the execution clients.

Transaction exchange refers to exchanging pending transactions between nodes so that miners can select some of them for inclusion in the next block.

Sub-protocols

DevP2P allows multiple sub-protocols to run on the same network connection. This means that nodes can simultaneously participate in different Ethereum functionalities(such as transferring blocks, transactions, and node discovery) over a single connection.

LES(Light Ethereum Subprotocol)

It is protocol used by light clients, which only download block headers as they appear and fetch other parts of the blockchain on-demand.

Snap(Ethereum Snapshot Protocol)

Facilitating the exchange of Ethereum state snapshots between peers.

Wit(witness protocol)

Facilitating the exchange of Ethereum state witnesses between peers.

---

reference

- https://github.com/ethereum/devp2p
- https://eips.ethereum.org/EIPS/eip-1459