Communication in Geth (1)

The execution layer's networking protocol is divided into two stacks:

- **Discovery stack**: built on top of UDP and allows a new node to find peers to connect
- **DevP2P stack**: sits on top of TCP and enables nodes to exchange information

---

Discovery Stack

process of finding other nodes in network

- bootstrap using a small set of bootnodes

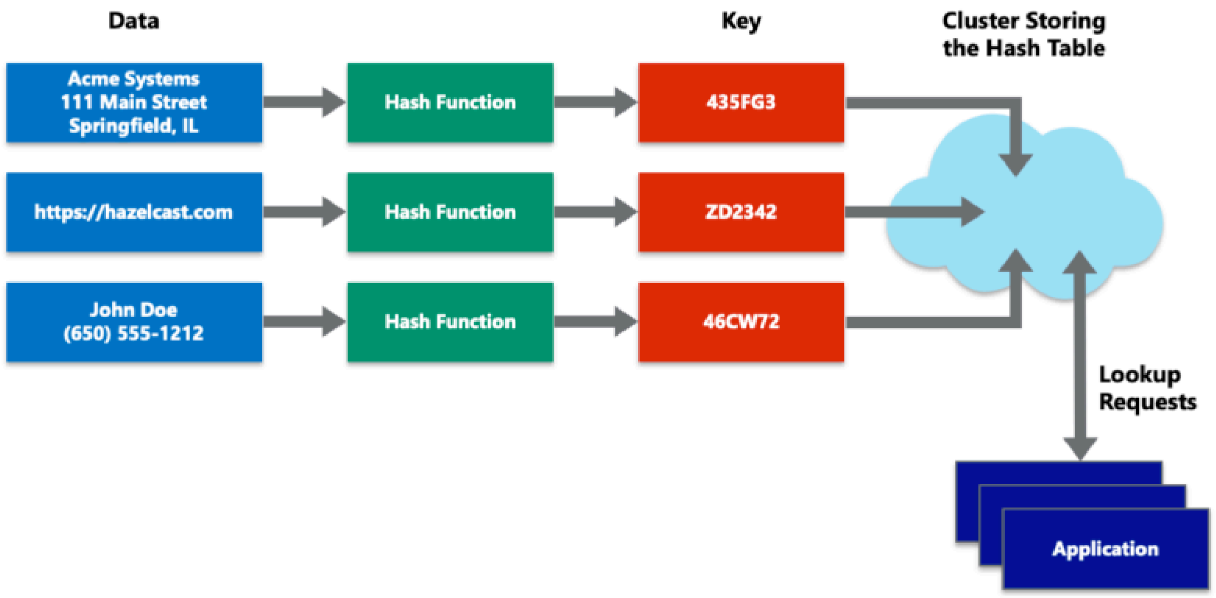
```
// go-ethereum/params/bootnodes.go

// MainnetBootnodes are the enode URLs of the P2P bootstrap nodes running on
// the main Ethereum network.
```

```
var MainnetBootnodes = []string{
   // Ethereum Foundation Go Bootnodes
   "enode://d860a01f9722d78051619d1e2351aba3f43f943f6f00718d1b9baa41
01932a1f5011f16bb2b1bb35db20d6fe28fa0bf09636d26a87d31de9ec6203eee
db1f666@18.138.108.67:30303", // bootnode-aws-ap-southeast-1-001
   "enode://22a8232c3abc76a16ae9d6c3b164f98775fe226f0917b0ca871128a
74a8e9630b458460865bab457221f1d448dd9791d24c4e5d88786180ac185df
813a68d4de@3.209.45.79:30303", // bootnode-aws-us-east-1-001
   "enode://2b252ab6a1d0f971d9722cb839a42cb81db019ba44c08754628ab
4a823487071b5695317c8ccd085219c3a03af063495b2f1da8d18218da2d6a8
2981b45e6ffc@65.108.70.101:30303", // bootnode-hetzner-hel
   "enode://4aeb4ab6c14b23e2c4cfdce879c04b0748a20d8e9b59e25ded2a
08143e265c6c25936e74cbc8e641e3312ca288673d91f2f93f8e277de3cfa444
ecdaaf982052@157.90.35.166:30303", // bootnode-hetzner-fsn
}
```

Distributed hash table

A distributed hash table (DHT) is a decentralized data storage system that stores and retrieves data based on key-value pairs. Each node in a DHT is responsible for a set of keys and their associated values. Any node can retrieve the value associated with a given key.



Geth uses DHT to share lists of nodes. Each node has a version of this table containing the information required to connect to its closest peers.

The "closeness" is not geographical - distance is defined by the similarity of the node's ID. Each node's table is regularly refreshed as a security feature.

`distance(n₁, n₂) = keccak256(n₁) XOR keccak256(n₂)`

Process of Discovery

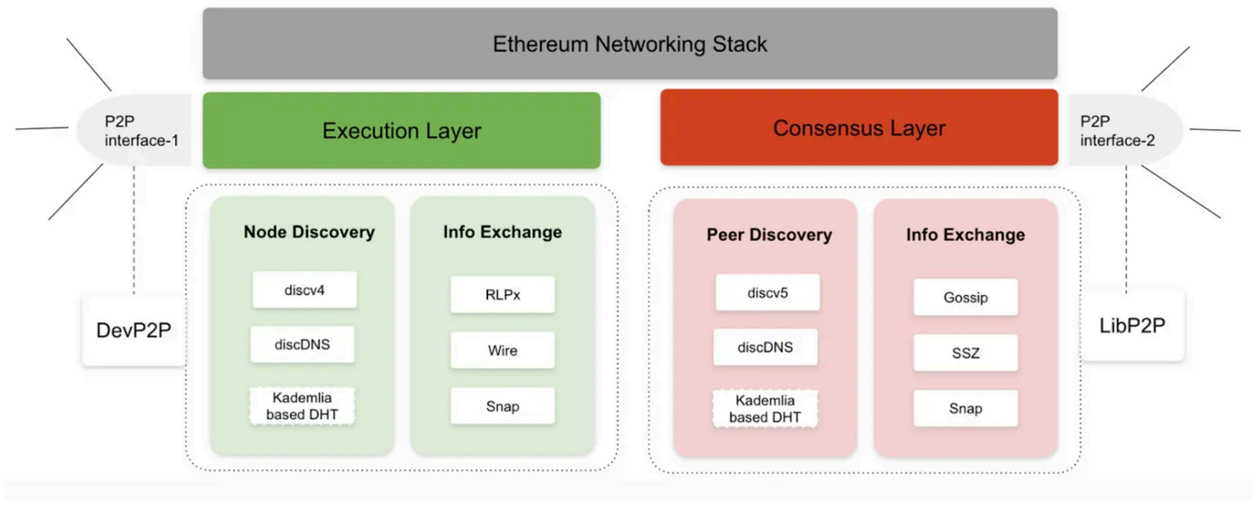
Starts with a game of PING-PONG.

1. The initial messages that alerts a bootnode to the existence of a new node entering the network is a PING.
   a. includes hashed information about the new node, the bootnode and an expiry time-stamp
2. The bootnode receives the PING and returns a PONG containing the PING hash. If the PING and PONG hashes match then the connection between the new node and bootnode is verified and they are said to have "bonded".
3. Once bonded, the new node can send a FIND-NEIGHBOURS request to the bootnode.
   a. The data returned by bootnode includes list of peers that new node can connect to.
4. Begins PING-PONG exchange with each of them.

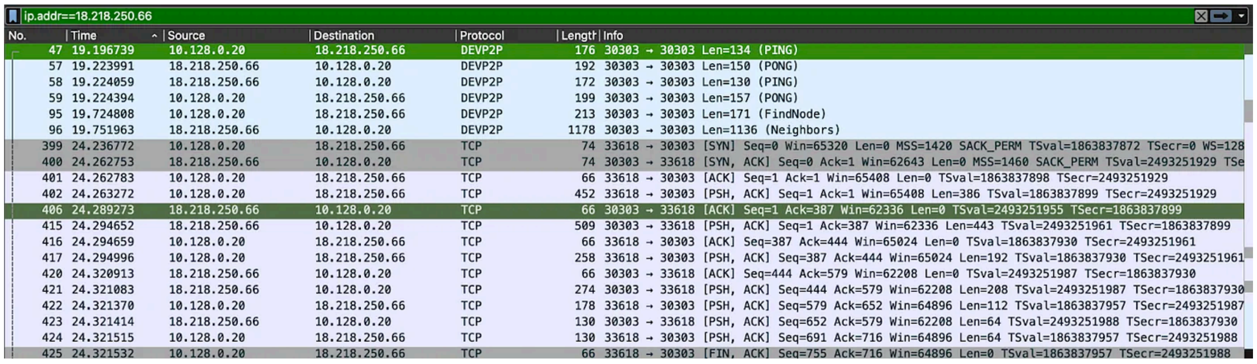
```
1. Start client
2. Send message to bootnode
3. Bond to bootnode
4. Find neighbours
5. Bond to neighbours
```

Discovery V4 protocol

Execution clients are currently using the Discv4 discovery protocol and there is an active effort to migrate to the Discv5 protocol.



Use wireshark & wireshark dissector to decode the DevP2P traffic.



```
ip.addr==18.218.250.66
No.   Time        Source          Destination     Protocol  Length Info
   47 19.196739   10.128.0.20     18.218.250.66   DEVP2P     176 30303 → 30303 Len=134 (PING)
   57 19.223991   18.218.250.66   10.128.0.20     DEVP2P     192 30303 → 30303 Len=150 (PONG)
   58 19.224059   18.218.250.66   10.128.0.20     DEVP2P     172 30303 → 30303 Len=130 (PING)
   59 19.224394   10.128.0.20     18.218.250.66   DEVP2P     199 30303 → 30303 Len=157 (PONG)
   95 19.724808   10.128.0.20     18.218.250.66   DEVP2P     213 30303 → 30303 Len=171 (FindNode)
   96 19.751963   18.218.250.66   10.128.0.20     DEVP2P    1178 30303 → 30303 Len=1136 (Neighbors)
  399 24.236772   10.128.0.20     18.218.250.66   TCP         74 33618 → 30303 [SYN] Seq=0 Win=65320 Len=0 MSS=1420 SACK_PERM TSval=1863837872 TSecr=0 WS=128
  400 24.262753   18.218.250.66   10.128.0.20     TCP         74 30303 → 33618 [SYN, ACK] Seq=0 Ack=1 Win=62643 Len=0 MSS=1460 SACK_PERM TSval=2493251929 TSe
  401 24.262783   10.128.0.20     18.218.250.66   TCP         66 33618 → 30303 [ACK] Seq=1 Ack=1 Win=65408 Len=0 TSval=1863837898 TSecr=2493251929
  402 24.263272   10.128.0.20     18.218.250.66   TCP        452 33618 → 30303 [PSH, ACK] Seq=1 Ack=1 Win=65408 Len=386 TSval=1863837899 TSecr=2493251929
  406 24.289273   18.218.250.66   10.128.0.20     TCP         66 30303 → 33618 [ACK] Seq=1 Ack=387 Win=62336 Len=0 TSval=2493251955 TSecr=1863837899
  415 24.294652   18.218.250.66   10.128.0.20     TCP        509 30303 → 33618 [PSH, ACK] Seq=1 Ack=387 Win=62336 Len=443 TSval=2493251961 TSecr=1863837899
  416 24.294659   10.128.0.20     18.218.250.66   TCP         66 33618 → 30303 [ACK] Seq=387 Ack=444 Win=65024 Len=0 TSval=1863837930 TSecr=2493251961
  417 24.294996   10.128.0.20     18.218.250.66   TCP        258 33618 → 30303 [PSH, ACK] Seq=387 Ack=444 Win=65024 Len=192 TSval=1863837930 TSecr=2493251961
  420 24.320913   18.218.250.66   10.128.0.20     TCP         66 30303 → 33618 [ACK] Seq=444 Ack=579 Win=62208 Len=0 TSval=2493251987 TSecr=1863837930
  421 24.321083   18.218.250.66   10.128.0.20     TCP        274 30303 → 33618 [PSH, ACK] Seq=444 Ack=579 Win=62208 Len=208 TSval=2493251987 TSecr=1863837930
  422 24.321370   10.128.0.20     18.218.250.66   TCP        178 33618 → 30303 [PSH, ACK] Seq=579 Ack=652 Win=64896 Len=112 TSval=1863837957 TSecr=2493251987
  423 24.321414   18.218.250.66   10.128.0.20     TCP        130 30303 → 33618 [PSH, ACK] Seq=652 Ack=579 Win=62208 Len=64 TSval=2493251988 TSecr=1863837930
  424 24.321515   10.128.0.20     18.218.250.66   TCP        130 33618 → 30303 [PSH, ACK] Seq=691 Ack=716 Win=64896 Len=64 TSval=1863837957 TSecr=2493251988
  425 24.321532   10.128.0.20     18.218.250.66   TCP         66 33618 → 30303 [FIN, ACK] Seq=755 Ack=716 Win=64896 Len=0 TSval=1863837957 TSecr=2493251988
```

In the image above, we can see there are `PING/PONG/FindNode/Neighbors` messages.

Then let's follow each step of Discovery in go-ethereum client.

0. handlePacket

- handle packet which received from other nodes

```
// go-ethereum/p2p/discover/v4_udp.go

func (t *UDPv4) handlePacket(from *net.UDPAddr, buf []byte) error {
    rawpacket, fromKey, hash, err := v4wire.Decode(buf)
    if err != nil {
        t.log.Debug("Bad discv4 packet", "addr", from, "err", err)
        return err
    }
    packet := t.wrapPacket(rawpacket)
    fromID := fromKey.ID()
    if err == nil && packet.preverify != nil {
        err = packet.preverify(packet, from, fromID, fromKey)
    }
    t.log.Trace("<< "+packet.Name(), "id", fromID, "addr", from, "err", err)
    if err == nil && packet.handle != nil {
        packet.handle(packet, from, fromID, hash)
    }
    return err
}
```

1. decode data from other nodes
2. verify packet(kind of data structure)
3. handle packet based on what kind of packet it is

```
// Decode reads a discovery v4 packet.
func Decode(input []byte) (Packet, Pubkey, []byte, error) {
    if len(input) < headSize+1 {
        return nil, Pubkey{}, nil, ErrPacketTooSmall
    }
    hash, sig, sigdata := input[:macSize], input[macSize:headSize], input[headSize
    shouldhash := crypto.Keccak256(input[macSize:])
    if !bytes.Equal(hash, shouldhash) {
        return nil, Pubkey{}, nil, ErrBadHash
    }
    fromKey, err := recoverNodeKey(crypto.Keccak256(input[headSize:]), sig)
    if err != nil {
        return nil, fromKey, hash, err
    }

    var req Packet
    switch ptype := sigdata[0]; ptype {
    case PingPacket:
        req = new(Ping)
    case PongPacket:
        req = new(Pong)
    case FindnodePacket:
        req = new(Findnode)
    case NeighborsPacket:
        req = new(Neighbors)
    case ENRRequestPacket:
        req = new(ENRRequest)
    case ENRResponsePacket:
        req = new(ENRResponse)
    default:
        return nil, fromKey, hash, fmt.Errorf("unknown type: %d", ptype)
    }
    // Here we use NewStream to allow for additional data after the first
    // RLP object (forward-compatibility).
    s := rlp.NewStream(bytes.NewReader(sigdata[1:]), 0)
    err = s.Decode(req)
    return req, fromKey, hash, err
}
```

▼ **Ping packet(0x01)**

```
packet-data = [version, from, to, expiration, enr-seq ...]
version = 4
from = [sender-ip, sender-udp-port, sender-tcp-port]
to = [recipient-ip, recipient-udp-port, 0]
```

▼ **Pong packet(0x02)**

```
packet-data = [to, ping-hash, expiration, enr-seq, ...]
```

▼ **FindNode packet(0x03)**

```
packet-data = [target, expiration, ...]
```

▼ **Neighbors packet(0x04)**

```
packet-data = [nodes, expiration, ...]
nodes = [[ip, udp-port, tcp-port, node-id], ...]
```

▼ **ENRRequest packet(0x05)**

```
packet-data = [expiration]
```

▼ **ENRResponse packet(0x06)**

```
packet-data = [request-hash, ENR]
```

1. PING

Bootnode receives PING messages and then send PONG messages to new node.

```
// go-ethereum/p2p/discover/v4_udp.go

// PING/v4

func (t *UDPv4) verifyPing(h *packetHandlerV4, from *net.UDPAddr, fromID eno
    req := h.Packet.(*v4wire.Ping)

    if v4wire.Expired(req.Expiration) {
        return errExpired
    }
    senderKey, err := v4wire.DecodePubkey(crypto.S256(), fromKey)
    if err != nil {
        return err
    }
    h.senderKey = senderKey
    return nil
}

func (t *UDPv4) handlePing(h *packetHandlerV4, from *net.UDPAddr, fromID enc
    req := h.Packet.(*v4wire.Ping)

    // Reply.
    t.send(from, fromID, &v4wire.Pong{
        To:         v4wire.NewEndpoint(from, req.From.TCP),
        ReplyTok:   mac,
        Expiration: uint64(time.Now().Add(expiration).Unix()),
        ENRSeq:     t.localNode.Node().Seq(),
    })

    // Ping back if our last pong on file is too far in the past.
    n := wrapNode(enode.NewV4(h.senderKey, from.IP, int(req.From.TCP), from.P
    if time.Since(t.db.LastPongReceived(n.ID(), from.IP)) > bondExpiration {
        t.sendPing(fromID, from, func() {
            t.tab.addVerifiedNode(n)
        })
    } else {
        t.tab.addVerifiedNode(n)
    }

    // Update node database and endpoint predictor.
    t.db.UpdateLastPingReceived(n.ID(), from.IP, time.Now())
    t.localNode.UDPEndpointStatement(from, &net.UDPAddr{IP: req.To.IP, Port: int
}
```

1. Check whether PING message is expired or not
2. Decode public key
3. Send Pong message to other node
4. If there is no error occurred, add remote node `n` to verified node list

5. Also update node database and endpoint predictor

▼ pingLoop

```
// go-ethereum/p2p/peer.go

const pingInterval = 15 * time.Second

// Peer represents a connected remote node.
type Peer struct {
    rw *conn
    running map[string]*protoRW
    log log.Logger
    created mclock.AbsTime

    wg sync.WaitGroup
    protoErr chan error
    closed chan struct{}
    pingRecv chan struct{}
    disc chan DiscReason

    // events receives message send / receive events if set
    events *event.Feed
    testPipe *MsgPipeRW // for testing
}


func (p *Peer) pingLoop() {
    defer p.wg.Done()

    ping := time.NewTimer(pingInterval)
    defer ping.Stop()

    for {
        select {
```

```
        case <-ping.C:
            if err := SendItems(p.rw, pingMsg); err != nil {
                p.protoErr <- err
                return
            }
            ping.Reset(pingInterval)
        case <-p.pingRecv:
            SendItems(p.rw, pongMsg)
        case <-p.closed:
            return
        }
    }
}
```

SendItems()

```
// go-ethereum/p2p/messages.go

type MsgWriter interface {
    // WriteMsg sends a message. It will block until the message's
    // Payload has been consumed by the other end.
    //
    // Note that messages can be sent only once because their
    // payload reader is drained.
    WriteMsg(Msg) error
}

// Send writes an RLP-encoded message with the given code.
// data should encode as an RLP list.
func Send(w MsgWriter, msgcode uint64, data interface{}) error {
    size, r, err := rlp.EncodeToReader(data)
    if err != nil {
        return err
    }
    return w.WriteMsg(Msg{Code: msgcode, Size: uint32(size), Payload: r})
}

// SendItems writes an RLP with the given code and data elements.
// For a call such as:
//
// SendItems(w, code, e1, e2, e3)
//
// the message payload will be an RLP list containing the items:
//
// [e1, e2, e3]
func SendItems(w MsgWriter, msgcode uint64, elems ...interface{}) error {
    return Send(w, msgcode, elems)
}
```

2. PONG

The new node receives Pong message.

```
// go-ethereum/p2p/discover/v4_udp.go

// PONG/v4

func (t *UDPv4) verifyPong(h *packetHandlerV4, from *net.UDPAddr, fromID enc
    req := h.Packet.(*v4wire.Pong)

    if v4wire.Expired(req.Expiration) {
        return errExpired
    }
    if !t.handleReply(fromID, from.IP, req) {
        return errUnsolicitedReply
    }
    t.localNode.UDPEndpointStatement(from, &net.UDPAddr{IP: req.To.IP, Port: int
    t.db.UpdateLastPongReceived(fromID, from.IP, time.Now())
    return nil
}
```

1. Check whether Pong message is expired or not.
2. Check Pong message matches its Ping message.
3. If every verification step is passed, local node database will be updated.

3. FIND NODE

```
// go-ethereum/p2p/discover/v4_udp.go

// FINDNODE/v4

func (t *UDPv4) verifyFindnode(h *packetHandlerV4, from *net.UDPAddr, fromID
    req := h.Packet.(*v4wire.Findnode)

    if v4wire.Expired(req.Expiration) {
        return errExpired
    }
    if !t.checkBond(fromID, from.IP) {
        // No endpoint proof pong exists, we don't process the packet. This prevent
        // attack vector where the discovery protocol could be used to amplify traffi
        // DDOS attack. A malicious actor would send a findnode request with the IP
        // and UDP port of the target as the source address. The recipient of the fin
        // packet would then send a neighbors packet (which is a much bigger pack
        // findnode) to the victim.
        return errUnknownNode
    }
    return nil
}

func (t *UDPv4) handleFindnode(h *packetHandlerV4, from *net.UDPAddr, fromI
    req := h.Packet.(*v4wire.Findnode)
    // Determine closest nodes.
    target := enode.ID(crypto.Keccak256Hash(req.Target[:]))
    closest := t.tab.findnodeByID(target, bucketSize, true).entries

    // Send neighbors in chunks with at most maxNeighbors per packet
    // to stay below the packet size limit.
    p := v4wire.Neighbors{Expiration: uint64(time.Now().Add(expiration).Unix())}
    var sent bool
    for _, n := range closest {
        if netutil.CheckRelayIP(from.IP, n.IP()) == nil {
            p.Nodes = append(p.Nodes, nodeToRPC(n))
        }
        if len(p.Nodes) == v4wire.MaxNeighbors {
            t.send(from, fromID, &p)
            p.Nodes = p.Nodes[:0]
            sent = true
        }
    }
    if len(p.Nodes) > 0 || !sent {
        t.send(from, fromID, &p)
    }
}
```

1. Check whether FindNode message is expired or not.
2. To prevent DDOS attack, the node check for its bond.
3. After every verification step is passed,
4. Find the closest node in table.
5. Send node list to the node which request FindNode message.

***maximum number of node list is 12.*

4. NEIGHBORS

It verifies given FindNode response is valid.

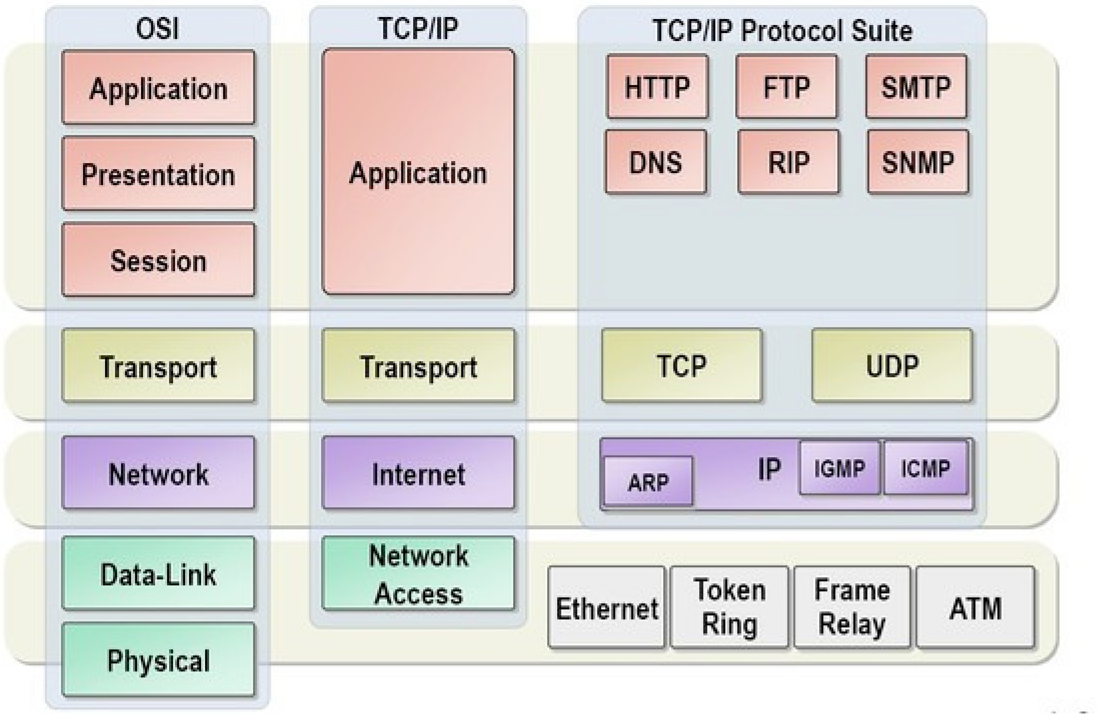
```
// go-ethereum/p2p/discover/v4_udp.go

// NEIGHBORS/v4

func (t *UDPv4) verifyNeighbors(h *packetHandlerV4, from *net.UDPAddr, fromI
    req := h.Packet.(*v4wire.Neighbors)

    if v4wire.Expired(req.Expiration) {
        return errExpired
    }
    if !t.handleReply(fromID, from.IP, h.Packet) {
        return errUnsolicitedReply
    }
    return nil
}
```

Why built on UDP?



TCP(Transmission Control Protocol) and UDP(User Datagram Protocol) are two of the main protocols used in network communications, particularly in the context of the Internet Protocol(IP). They are part of Transport Layer in the OSI(Open Systems Interconnection) model and have different characteristics and uses:

> TCP is used when reliability is essential, and data needs to arrive intact and in order. UDP is preferred when speed is crucial, and some loss of data can be tolerated, such as in live audio or video streams.

For discovery, where a node simply wants to make its presence known in order to then establish a formal connection with a peer, UDP is sufficient. However, for the rest of the networking stack, UDP is not fit for purpose.

DevP2P Stack

reference

- https://ethereum.org/en/developers/docs/networking-layer
- https://hazelcast.com/glossary/distributed-hash-table/
- https://medium.com/coinmonks/dissecting-the-ethereum-networking-stack-node-discovery-4b3f7895f83f