F:\git\java\mar3\filemonitor\target\go-ethereum\go-ethereum-4.doc

0:F:\git\coin\ethereum\go-ethereum\les\peer.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package les implements the Light Ethereum Subprotocol.
package les

import (
"errors"
"fmt"
"math/big"
"sync"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/eth"
"github.com/ethereum/go-ethereum/les/flowcontrol"
"github.com/ethereum/go-ethereum/p2p"
"github.com/ethereum/go-ethereum/rlp"
)

var (
errClosed   = errors.New("peer set is closed")
errAlreadyRegistered = errors.New("peer is already registered")
errNotRegistered   = errors.New("peer is not registered")
)

const (
maxHeadInfoLen   = 20
maxResponseErrors = 50 // number of invalid responses tolerated (makes the protocol less brittle but still avoids spam)
)

type peer struct {
*p2p.Peer

rw p2p.MsgReadWriter

version int   // Protocol version negotiated
network uint64 // Network ID being on

```
id string

headInfo *announceData
lock     sync.RWMutex

announceChn chan announceData
sendQueue   *execQueue

poolEntry      *poolEntry
hasBlock       func(common.Hash, uint64) bool
responseErrors int

fcClient       *flowcontrol.ClientNode // nil if the peer is server only
fcServer       *flowcontrol.ServerNode // nil if the peer is client only
fcServerParams *flowcontrol.ServerParams
fcCosts        requestCostTable
}

func newPeer(version int, network uint64, p *p2p.Peer, rw p2p.MsgReadWriter) *peer {
id := p.ID()

return &peer{
Peer:        p,
rw:          rw,
version:     version,
network:     network,
id:          fmt.Sprintf("%x", id[:8]),
announceChn: make(chan announceData, 20),
}
}

func (p *peer) canQueue() bool {
return p.sendQueue.canQueue()
}

func (p *peer) queueSend(f func()) {
p.sendQueue.queue(f)
}

// Info gathers and returns a collection of metadata known about a peer.
func (p *peer) Info() *eth.PeerInfo {
```

```
return &eth.PeerInfo{
Version:    p.version,
Difficulty: p.Td(),
Head:       fmt.Sprintf("%x", p.Head()),
}
}

// Head retrieves a copy of the current head (most recent) hash of the peer.
func (p *peer) Head() (hash common.Hash) {
p.lock.RLock()
defer p.lock.RUnlock()

copy(hash[:], p.headInfo.Hash[:])
return hash
}

func (p *peer) HeadAndTd() (hash common.Hash, td *big.Int) {
p.lock.RLock()
defer p.lock.RUnlock()

copy(hash[:], p.headInfo.Hash[:])
return hash, p.headInfo.Td
}

func (p *peer) headBlockInfo() blockInfo {
p.lock.RLock()
defer p.lock.RUnlock()

return blockInfo{Hash: p.headInfo.Hash, Number: p.headInfo.Number, Td: p.headInfo.Td}
}

// Td retrieves the current total difficulty of a peer.
func (p *peer) Td() *big.Int {
p.lock.RLock()
defer p.lock.RUnlock()

return new(big.Int).Set(p.headInfo.Td)
}

// waitBefore implements distPeer interface
func (p *peer) waitBefore(maxCost uint64) (time.Duration, float64) {
return p.fcServer.CanSend(maxCost)
```

```
}

func sendRequest(w p2p.MsgWriter, msgcode, reqID, cost uint64, data interface{}) error {
type req struct {
ReqID uint64
Data  interface{}
}
return p2p.Send(w, msgcode, req{reqID, data})
}

func sendResponse(w p2p.MsgWriter, msgcode, reqID, bv uint64, data interface{}) error {
type resp struct {
ReqID, BV uint64
Data      interface{}
}
return p2p.Send(w, msgcode, resp{reqID, bv, data})
}

func (p *peer) GetRequestCost(msgcode uint64, amount int) uint64 {
p.lock.RLock()
defer p.lock.RUnlock()

cost := p.fcCosts[msgcode].baseCost + p.fcCosts[msgcode].reqCost*uint64(amount)
if cost > p.fcServerParams.BufLimit {
cost = p.fcServerParams.BufLimit
}
return cost
}

// HasBlock checks if the peer has a given block
func (p *peer) HasBlock(hash common.Hash, number uint64) bool {
p.lock.RLock()
hasBlock := p.hasBlock
p.lock.RUnlock()
return hasBlock != nil && hasBlock(hash, number)
}

// SendAnnounce announces the availability of a number of blocks through
// a hash notification.
func (p *peer) SendAnnounce(request announceData) error {
return p2p.Send(p.rw, AnnounceMsg, request)
}
```

```
// SendBlockHeaders sends a batch of block headers to the remote peer.
func (p *peer) SendBlockHeaders(reqID, bv uint64, headers []*types.Header) error {
return sendResponse(p.rw, BlockHeadersMsg, reqID, bv, headers)
}

// SendBlockBodiesRLP sends a batch of block contents to the remote peer from
// an already RLP encoded format.
func (p *peer) SendBlockBodiesRLP(reqID, bv uint64, bodies []rlp.RawValue) error {
return sendResponse(p.rw, BlockBodiesMsg, reqID, bv, bodies)
}

// SendCodeRLP sends a batch of arbitrary internal data, corresponding to the
// hashes requested.
func (p *peer) SendCode(reqID, bv uint64, data [][]byte) error {
return sendResponse(p.rw, CodeMsg, reqID, bv, data)
}

// SendReceiptsRLP sends a batch of transaction receipts, corresponding to the
// ones requested from an already RLP encoded format.
func (p *peer) SendReceiptsRLP(reqID, bv uint64, receipts []rlp.RawValue) error {
return sendResponse(p.rw, ReceiptsMsg, reqID, bv, receipts)
}

// SendProofs sends a batch of merkle proofs, corresponding to the ones requested.
func (p *peer) SendProofs(reqID, bv uint64, proofs proofsData) error {
return sendResponse(p.rw, ProofsMsg, reqID, bv, proofs)
}

// SendHeaderProofs sends a batch of header proofs, corresponding to the ones requested.
func (p *peer) SendHeaderProofs(reqID, bv uint64, proofs []ChtResp) error {
return sendResponse(p.rw, HeaderProofsMsg, reqID, bv, proofs)
}

// RequestHeadersByHash fetches a batch of blocks' headers corresponding to the
// specified header query, based on the hash of an origin block.
func (p *peer) RequestHeadersByHash(reqID, cost uint64, origin common.Hash, amount int, skip
int, reverse bool) error {
p.Log().Debug("Fetching batch of headers", "count", amount, "fromhash", origin, "skip", skip,
"reverse", reverse)
return sendRequest(p.rw, GetBlockHeadersMsg, reqID, cost, &getBlockHeadersData{Origin:
hashOrNumber{Hash: origin}, Amount: uint64(amount), Skip: uint64(skip), Reverse: reverse})
```

```
}

// RequestHeadersByNumber fetches a batch of blocks' headers corresponding to the
// specified header query, based on the number of an origin block.
func (p *peer) RequestHeadersByNumber(reqID, cost, origin uint64, amount int, skip int, reverse
bool) error {
p.Log().Debug("Fetching batch of headers", "count", amount, "fromnum", origin, "skip", skip,
"reverse", reverse)
return sendRequest(p.rw, GetBlockHeadersMsg, reqID, cost, &getBlockHeadersData{Origin:
hashOrNumber{Number: origin}, Amount: uint64(amount), Skip: uint64(skip), Reverse: reverse})
}

// RequestBodies fetches a batch of blocks' bodies corresponding to the hashes
// specified.
func (p *peer) RequestBodies(reqID, cost uint64, hashes []common.Hash) error {
p.Log().Debug("Fetching batch of block bodies", "count", len(hashes))
return sendRequest(p.rw, GetBlockBodiesMsg, reqID, cost, hashes)
}

// RequestCode fetches a batch of arbitrary data from a node's known state
// data, corresponding to the specified hashes.
func (p *peer) RequestCode(reqID, cost uint64, reqs []*CodeReq) error {
p.Log().Debug("Fetching batch of codes", "count", len(reqs))
return sendRequest(p.rw, GetCodeMsg, reqID, cost, reqs)
}

// RequestReceipts fetches a batch of transaction receipts from a remote node.
func (p *peer) RequestReceipts(reqID, cost uint64, hashes []common.Hash) error {
p.Log().Debug("Fetching batch of receipts", "count", len(hashes))
return sendRequest(p.rw, GetReceiptsMsg, reqID, cost, hashes)
}

// RequestProofs fetches a batch of merkle proofs from a remote node.
func (p *peer) RequestProofs(reqID, cost uint64, reqs []*ProofReq) error {
p.Log().Debug("Fetching batch of proofs", "count", len(reqs))
return sendRequest(p.rw, GetProofsMsg, reqID, cost, reqs)
}

// RequestHeaderProofs fetches a batch of header merkle proofs from a remote node.
func (p *peer) RequestHeaderProofs(reqID, cost uint64, reqs []*ChtReq) error {
p.Log().Debug("Fetching batch of header proofs", "count", len(reqs))
return sendRequest(p.rw, GetHeaderProofsMsg, reqID, cost, reqs)
```

```
}

func (p *peer) SendTxs(reqID, cost uint64, txs types.Transactions) error {
p.Log().Debug("Fetching batch of transactions", "count", len(txs))
return p2p.Send(p.rw, SendTxMsg, txs)
}

type keyValueEntry struct {
Key   string
Value rlp.RawValue
}
type keyValueList []keyValueEntry
type keyValueMap map[string]rlp.RawValue

func (l keyValueList) add(key string, val interface{}) keyValueList {
var entry keyValueEntry
entry.Key = key
if val == nil {
val = uint64(0)
}
enc, err := rlp.EncodeToBytes(val)
if err == nil {
entry.Value = enc
}
return append(l, entry)
}

func (l keyValueList) decode() keyValueMap {
m := make(keyValueMap)
for _, entry := range l {
m[entry.Key] = entry.Value
}
return m
}

func (m keyValueMap) get(key string, val interface{}) error {
enc, ok := m[key]
if !ok {
return errResp(ErrHandshakeMissingKey, "%s", key)
}
if val == nil {
return nil
```

```
}
return rlp.DecodeBytes(enc, val)
}

func (p *peer) sendReceiveHandshake(sendList keyValueList) (keyValueList, error) {
// Send out own handshake in a new thread
errc := make(chan error, 1)
go func() {
errc <- p2p.Send(p.rw, StatusMsg, sendList)
}()
// In the mean time retrieve the remote status message
msg, err := p.rw.ReadMsg()
if err != nil {
return nil, err
}
if msg.Code != StatusMsg {
return nil, errResp(ErrNoStatusMsg, "first msg has code %x (!= %x)", msg.Code, StatusMsg)
}
if msg.Size > ProtocolMaxMsgSize {
return nil, errResp(ErrMsgTooLarge, "%v > %v", msg.Size, ProtocolMaxMsgSize)
}
// Decode the handshake
var recvList keyValueList
if err := msg.Decode(&recvList); err != nil {
return nil, errResp(ErrDecode, "msg %v: %v", msg, err)
}
if err := <-errc; err != nil {
return nil, err
}
return recvList, nil
}

// Handshake executes the les protocol handshake, negotiating version number,
// network IDs, difficulties, head and genesis blocks.
func (p *peer) Handshake(td *big.Int, head common.Hash, headNum uint64, genesis
common.Hash, server *LesServer) error {
p.lock.Lock()
defer p.lock.Unlock()

var send keyValueList
send = send.add("protocolVersion", uint64(p.version))
send = send.add("networkId", uint64(p.network))
```

```
send = send.add("headTd", td)
send = send.add("headHash", head)
send = send.add("headNum", headNum)
send = send.add("genesisHash", genesis)
if server != nil {
send = send.add("serveHeaders", nil)
send = send.add("serveChainSince", uint64(0))
send = send.add("serveStateSince", uint64(0))
send = send.add("txRelay", nil)
send = send.add("flowControl/BL", server.defParams.BufLimit)
send = send.add("flowControl/MRR", server.defParams.MinRecharge)
list := server.fcCostStats.getCurrentList()
send = send.add("flowControl/MRC", list)
p.fcCosts = list.decode()
}
recvList, err := p.sendReceiveHandshake(send)
if err != nil {
return err
}
recv := recvList.decode()

var rGenesis, rHash common.Hash
var rVersion, rNetwork, rNum uint64
var rTd *big.Int

if err := recv.get("protocolVersion", &rVersion); err != nil {
return err
}
if err := recv.get("networkId", &rNetwork); err != nil {
return err
}
if err := recv.get("headTd", &rTd); err != nil {
return err
}
if err := recv.get("headHash", &rHash); err != nil {
return err
}
if err := recv.get("headNum", &rNum); err != nil {
return err
}
if err := recv.get("genesisHash", &rGenesis); err != nil {
return err
```

```
}

if rGenesis != genesis {
return errResp(ErrGenesisBlockMismatch, "%x (!= %x)", rGenesis[:8], genesis[:8])
}
if rNetwork != p.network {
return errResp(ErrNetworkIdMismatch, "%d (!= %d)", rNetwork, p.network)
}
if int(rVersion) != p.version {
return errResp(ErrProtocolVersionMismatch, "%d (!= %d)", rVersion, p.version)
}
if server != nil {
// until we have a proper peer connectivity API, allow LES connection to other servers
/*if recv.get("serveStateSince", nil) == nil {
return errResp(ErrUselessPeer, "wanted client, got server")
}*/
p.fcClient = flowcontrol.NewClientNode(server.fcManager, server.defParams)
} else {
if recv.get("serveChainSince", nil) != nil {
return errResp(ErrUselessPeer, "peer cannot serve chain")
}
if recv.get("serveStateSince", nil) != nil {
return errResp(ErrUselessPeer, "peer cannot serve state")
}
if recv.get("txRelay", nil) != nil {
return errResp(ErrUselessPeer, "peer cannot relay transactions")
}
params := &flowcontrol.ServerParams{}
if err := recv.get("flowControl/BL", &params.BufLimit); err != nil {
return err
}
if err := recv.get("flowControl/MRR", &params.MinRecharge); err != nil {
return err
}
var MRC RequestCostList
if err := recv.get("flowControl/MRC", &MRC); err != nil {
return err
}
p.fcServerParams = params
p.fcServer = flowcontrol.NewServerNode(params)
p.fcCosts = MRC.decode()
}
```

```
p.headInfo = &announceData{Td: rTd, Hash: rHash, Number: rNum}
return nil
}

// String implements fmt.Stringer.
func (p *peer) String() string {
return fmt.Sprintf("Peer %s [%s]", p.id,
fmt.Sprintf("les/%d", p.version),
)
}

// peerSetNotify is a callback interface to notify services about added or
// removed peers
type peerSetNotify interface {
registerPeer(*peer)
unregisterPeer(*peer)
}

// peerSet represents the collection of active peers currently participating in
// the Light Ethereum sub-protocol.
type peerSet struct {
peers      map[string]*peer
lock       sync.RWMutex
notifyList []peerSetNotify
closed     bool
}

// newPeerSet creates a new peer set to track the active participants.
func newPeerSet() *peerSet {
return &peerSet{
peers: make(map[string]*peer),
}
}

// notify adds a service to be notified about added or removed peers
func (ps *peerSet) notify(n peerSetNotify) {
ps.lock.Lock()
defer ps.lock.Unlock()

ps.notifyList = append(ps.notifyList, n)
for _, p := range ps.peers {
```

```
go n.registerPeer(p)
}
}

// Register injects a new peer into the working set, or returns an error if the
// peer is already known.
func (ps *peerSet) Register(p *peer) error {
ps.lock.Lock()
defer ps.lock.Unlock()

if ps.closed {
return errClosed
}
if _, ok := ps.peers[p.id]; ok {
return errAlreadyRegistered
}
ps.peers[p.id] = p
p.sendQueue = newExecQueue(100)
for _, n := range ps.notifyList {
go n.registerPeer(p)
}
return nil
}

// Unregister removes a remote peer from the active set, disabling any further
// actions to/from that particular entity. It also initiates disconnection at the networking layer.
func (ps *peerSet) Unregister(id string) error {
ps.lock.Lock()
defer ps.lock.Unlock()

if p, ok := ps.peers[id]; !ok {
return errNotRegistered
} else {
for _, n := range ps.notifyList {
go n.unregisterPeer(p)
}
p.sendQueue.quit()
p.Peer.Disconnect(p2p.DiscUselessPeer)
}
delete(ps.peers, id)
return nil
}
```

```
// AllPeerIDs returns a list of all registered peer IDs
func (ps *peerSet) AllPeerIDs() []string {
ps.lock.RLock()
defer ps.lock.RUnlock()

res := make([]string, len(ps.peers))
idx := 0
for id := range ps.peers {
res[idx] = id
idx++
}
return res
}

// Peer retrieves the registered peer with the given id.
func (ps *peerSet) Peer(id string) *peer {
ps.lock.RLock()
defer ps.lock.RUnlock()

return ps.peers[id]
}

// Len returns if the current number of peers in the set.
func (ps *peerSet) Len() int {
ps.lock.RLock()
defer ps.lock.RUnlock()

return len(ps.peers)
}

// BestPeer retrieves the known peer with the currently highest total difficulty.
func (ps *peerSet) BestPeer() *peer {
ps.lock.RLock()
defer ps.lock.RUnlock()

var (
bestPeer *peer
bestTd   *big.Int
)
for _, p := range ps.peers {
if td := p.Td(); bestPeer == nil || td.Cmp(bestTd) > 0 {
```

```
bestPeer, bestTd = p, td
}
}
return bestPeer
}

// AllPeers returns all peers in a list
func (ps *peerSet) AllPeers() []*peer {
ps.lock.RLock()
defer ps.lock.RUnlock()

list := make([]*peer, len(ps.peers))
i := 0
for _, peer := range ps.peers {
list[i] = peer
i++
}
return list
}

// Close disconnects all peers.
// No new peers can be registered after Close has returned.
func (ps *peerSet) Close() {
ps.lock.Lock()
defer ps.lock.Unlock()

for _, p := range ps.peers {
p.Disconnect(p2p.DiscQuitting)
}
ps.closed = true
}

1:F:\git\coin\ethereum\go-ethereum\les\protocol.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package les implements the Light Ethereum Subprotocol.
package les

import (
"fmt"
"io"
"math/big"
```

```
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/rlp"
)

// Constants to match up protocol versions and messages
const (
lpv1 = 1
)

// Supported versions of the les protocol (first is primary).
var ProtocolVersions = []uint{lpv1}

// Number of implemented message corresponding to different protocol versions.
var ProtocolLengths = []uint64{15}

const (
NetworkId          = 1
ProtocolMaxMsgSize = 10 * 1024 * 1024 // Maximum cap on the size of a protocol message
)

// les protocol message codes
const (
// Protocol messages belonging to LPV1
StatusMsg          = 0x00
AnnounceMsg        = 0x01
GetBlockHeadersMsg = 0x02
BlockHeadersMsg    = 0x03
GetBlockBodiesMsg  = 0x04
BlockBodiesMsg     = 0x05
GetReceiptsMsg     = 0x06
ReceiptsMsg        = 0x07
GetProofsMsg       = 0x08
ProofsMsg          = 0x09
GetCodeMsg         = 0x0a
CodeMsg            = 0x0b
SendTxMsg          = 0x0c
GetHeaderProofsMsg = 0x0d
HeaderProofsMsg    = 0x0e
)
```

```
type errCode int

const (
ErrMsgTooLarge = iota
ErrDecode
ErrInvalidMsgCode
ErrProtocolVersionMismatch
ErrNetworkIdMismatch
ErrGenesisBlockMismatch
ErrNoStatusMsg
ErrExtraStatusMsg
ErrSuspendedPeer
ErrUselessPeer
ErrRequestRejected
ErrUnexpectedResponse
ErrInvalidResponse
ErrTooManyTimeouts
ErrHandshakeMissingKey
)

func (e errCode) String() string {
return errorToString[int(e)]
}

// XXX change once legacy code is out
var errorToString = map[int]string{
ErrMsgTooLarge:             "Message too long",
ErrDecode:                  "Invalid message",
ErrInvalidMsgCode:          "Invalid message code",
ErrProtocolVersionMismatch: "Protocol version mismatch",
ErrNetworkIdMismatch:       "NetworkId mismatch",
ErrGenesisBlockMismatch:    "Genesis block mismatch",
ErrNoStatusMsg:             "No status message",
ErrExtraStatusMsg:          "Extra status message",
ErrSuspendedPeer:           "Suspended peer",
ErrRequestRejected:         "Request rejected",
ErrUnexpectedResponse:      "Unexpected response",
ErrInvalidResponse:         "Invalid response",
ErrTooManyTimeouts:         "Too many request timeouts",
ErrHandshakeMissingKey:     "Key missing from handshake message",
}
```

```
type chainManager interface {
GetBlockHashesFromHash(hash common.Hash, amount uint64) (hashes []common.Hash)
GetBlock(hash common.Hash) (block *types.Block)
Status() (td *big.Int, currentBlock common.Hash, genesisBlock common.Hash)
}

// announceData is the network packet for the block announcements.
type announceData struct {
Hash       common.Hash // Hash of one particular block being announced
Number     uint64      // Number of one particular block being announced
Td         *big.Int    // Total difficulty of one particular block being announced
ReorgDepth uint64
Update     keyValueList

haveHeaders uint64 // we have the headers of the remote peer's chain up to this number
headKnown   bool
requested   bool
next        *announceData
}

type blockInfo struct {
Hash   common.Hash // Hash of one particular block being announced
Number uint64      // Number of one particular block being announced
Td     *big.Int    // Total difficulty of one particular block being announced
}

// getBlockHashesData is the network packet for the hash based hash retrieval.
type getBlockHashesData struct {
Hash   common.Hash
Amount uint64
}

// getBlockHeadersData represents a block header query.
type getBlockHeadersData struct {
Origin  hashOrNumber // Block from which to retrieve headers
Amount  uint64       // Maximum number of headers to retrieve
Skip    uint64       // Blocks to skip between consecutive headers
Reverse bool         // Query direction (false = rising towards latest, true = falling towards genesis)
}

// hashOrNumber is a combined field for specifying an origin block.
type hashOrNumber struct {
```

```
Hash   common.Hash // Block hash from which to retrieve headers (excludes Number)
Number uint64      // Block hash from which to retrieve headers (excludes Hash)
}

// EncodeRLP is a specialized encoder for hashOrNumber to encode only one of the
// two contained union fields.
func (hn *hashOrNumber) EncodeRLP(w io.Writer) error {
if hn.Hash == (common.Hash{}) {
return rlp.Encode(w, hn.Number)
}
if hn.Number != 0 {
return fmt.Errorf("both origin hash (%x) and number (%d) provided", hn.Hash, hn.Number)
}
return rlp.Encode(w, hn.Hash)
}

// DecodeRLP is a specialized decoder for hashOrNumber to decode the contents
// into either a block hash or a block number.
func (hn *hashOrNumber) DecodeRLP(s *rlp.Stream) error {
_, size, _ := s.Kind()
origin, err := s.Raw()
if err == nil {
switch {
case size == 32:
err = rlp.DecodeBytes(origin, &hn.Hash)
case size <= 8:
err = rlp.DecodeBytes(origin, &hn.Number)
default:
err = fmt.Errorf("invalid input size %d for origin", size)
}
}
return err
}

// newBlockData is the network packet for the block propagation message.
type newBlockData struct {
Block *types.Block
TD    *big.Int
}

// blockBodiesData is the network packet for block content distribution.
type blockBodiesData []*types.Body
```

```
// CodeData is the network response packet for a node data retrieval.
type CodeData []struct {
Value []byte
}

type proofsData [][]rlp.RawValue

2:F:\git\coin\ethereum\go-ethereum\les\randselect.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package les implements the Light Ethereum Subprotocol.
package les

import (
"math/rand"
)

// wrsItem interface should be implemented by any entries that are to be selected from
// a weightedRandomSelect set. Note that recalculating monotonously decreasing item
// weights on-demand (without constantly calling update) is allowed
type wrsItem interface {
Weight() int64
}

// weightedRandomSelect is capable of weighted random selection from a set of items
type weightedRandomSelect struct {
root *wrsNode
idx  map[wrsItem]int
}

// newWeightedRandomSelect returns a new weightedRandomSelect structure
func newWeightedRandomSelect() *weightedRandomSelect {
return &weightedRandomSelect{root: &wrsNode{maxItems: wrsBranches}, idx:
make(map[wrsItem]int)}
}

// update updates an item's weight, adds it if it was non-existent or removes it if
// the new weight is zero. Note that explicitly updating decreasing weights is not necessary.
func (w *weightedRandomSelect) update(item wrsItem) {
w.setWeight(item, item.Weight())
}
```

```
// remove removes an item from the set
func (w *weightedRandomSelect) remove(item wrsItem) {
w.setWeight(item, 0)
}

// setWeight sets an item's weight to a specific value (removes it if zero)
func (w *weightedRandomSelect) setWeight(item wrsItem, weight int64) {
idx, ok := w.idx[item]
if ok {
w.root.setWeight(idx, weight)
if weight == 0 {
delete(w.idx, item)
}
} else {
if weight != 0 {
if w.root.itemCnt == w.root.maxItems {
// add a new level
newRoot := &wrsNode{sumWeight: w.root.sumWeight, itemCnt: w.root.itemCnt, level: w.root.level
+ 1, maxItems: w.root.maxItems * wrsBranches}
newRoot.items[0] = w.root
newRoot.weights[0] = w.root.sumWeight
w.root = newRoot
}
w.idx[item] = w.root.insert(item, weight)
}
}
}

// choose randomly selects an item from the set, with a chance proportional to its
// current weight. If the weight of the chosen element has been decreased since the
// last stored value, returns it with a newWeight/oldWeight chance, otherwise just
// updates its weight and selects another one
func (w *weightedRandomSelect) choose() wrsItem {
for {
if w.root.sumWeight == 0 {
return nil
}
val := rand.Int63n(w.root.sumWeight)
choice, lastWeight := w.root.choose(val)
weight := choice.Weight()
if weight != lastWeight {
```

```
w.setWeight(choice, weight)
}
if weight >= lastWeight || rand.Int63n(lastWeight) < weight {
return choice
}
}
}

const wrsBranches = 8 // max number of branches in the wrsNode tree

// wrsNode is a node of a tree structure that can store wrsItems or further wrsNodes.
type wrsNode struct {
items                [wrsBranches]interface{}
weights               [wrsBranches]int64
sumWeight               int64
level, itemCnt, maxItems int
}

// insert recursively inserts a new item to the tree and returns the item index
func (n *wrsNode) insert(item wrsItem, weight int64) int {
branch := 0
for n.items[branch] != nil && (n.level == 0 || n.items[branch].(*wrsNode).itemCnt ==
n.items[branch].(*wrsNode).maxItems) {
branch++
if branch == wrsBranches {
panic(nil)
}
}
n.itemCnt++
n.sumWeight += weight
n.weights[branch] += weight
if n.level == 0 {
n.items[branch] = item
return branch
} else {
var subNode *wrsNode
if n.items[branch] == nil {
subNode = &wrsNode{maxItems: n.maxItems / wrsBranches, level: n.level - 1}
n.items[branch] = subNode
} else {
subNode = n.items[branch].(*wrsNode)
}
```

```
subIdx := subNode.insert(item, weight)
return subNode.maxItems*branch + subIdx
}
}

// setWeight updates the weight of a certain item (which should exist) and returns
// the change of the last weight value stored in the tree
func (n *wrsNode) setWeight(idx int, weight int64) int64 {
if n.level == 0 {
oldWeight := n.weights[idx]
n.weights[idx] = weight
diff := weight - oldWeight
n.sumWeight += diff
if weight == 0 {
n.items[idx] = nil
n.itemCnt--
}
return diff
}
branchItems := n.maxItems / wrsBranches
branch := idx / branchItems
diff := n.items[branch].(*wrsNode).setWeight(idx-branch*branchItems, weight)
n.weights[branch] += diff
n.sumWeight += diff
if weight == 0 {
n.itemCnt--
}
return diff
}

// choose recursively selects an item from the tree and returns it along with its weight
func (n *wrsNode) choose(val int64) (wrsItem, int64) {
for i, w := range n.weights {
if val < w {
if n.level == 0 {
return n.items[i].(wrsItem), n.weights[i]
} else {
return n.items[i].(*wrsNode).choose(val)
}
} else {
val -= w
}
```

```
}
panic(nil)
}

3:F:\git\coin\ethereum\go-ethereum\les\randselect_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package les

import (
"math/rand"
"testing"
)

type testWrsItem struct {
idx  int
widx *int
}

func (t *testWrsItem) Weight() int64 {
w := *t.widx
if w == -1 || w == t.idx {
return int64(t.idx + 1)
}
return 0
}

func TestWeightedRandomSelect(t *testing.T) {
testFn := func(cnt int) {
s := newWeightedRandomSelect()
w := -1
list := make([]testWrsItem, cnt)
for i := range list {
list[i] = testWrsItem{idx: i, widx: &w}
s.update(&list[i])
}
w = rand.Intn(cnt)
c := s.choose()
if c == nil {
t.Errorf("expected item, got nil")
} else {
if c.(*testWrsItem).idx != w {
```

```
t.Errorf("expected another item")
}
}
w = -2
if s.choose() != nil {
t.Errorf("expected nil, got item")
}
}
testFn(1)
testFn(10)
testFn(100)
testFn(1000)
testFn(10000)
testFn(100000)
testFn(1000000)
}

4:F:\git\coin\ethereum\go-ethereum\les\request_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package les

import (
"context"
"testing"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/light"
)

var testBankSecureTrieKey = secAddr(testBankAddress)

func secAddr(addr common.Address) []byte {
return crypto.Keccak256(addr[:])
}

type accessTestFn func(db ethdb.Database, bhash common.Hash, number uint64)
light.OdrRequest
```

```
func TestBlockAccessLes1(t *testing.T) { testAccess(t, 1, tfBlockAccess) }

func tfBlockAccess(db ethdb.Database, bhash common.Hash, number uint64) light.OdrRequest {
return &light.BlockRequest{Hash: bhash, Number: number}
}

func TestReceiptsAccessLes1(t *testing.T) { testAccess(t, 1, tfReceiptsAccess) }

func tfReceiptsAccess(db ethdb.Database, bhash common.Hash, number uint64)
light.OdrRequest {
return &light.ReceiptsRequest{Hash: bhash, Number: number}
}

func TestTrieEntryAccessLes1(t *testing.T) { testAccess(t, 1, tfTrieEntryAccess) }

func tfTrieEntryAccess(db ethdb.Database, bhash common.Hash, number uint64)
light.OdrRequest {
return &light.TrieRequest{Id: light.StateTrieID(core.GetHeader(db, bhash,
core.GetBlockNumber(db, bhash))), Key: testBankSecureTrieKey}
}

func TestCodeAccessLes1(t *testing.T) { testAccess(t, 1, tfCodeAccess) }

func tfCodeAccess(db ethdb.Database, bhash common.Hash, number uint64) light.OdrRequest {
header := core.GetHeader(db, bhash, core.GetBlockNumber(db, bhash))
if header.Number.Uint64() < testContractDeployed {
return nil
}
sti := light.StateTrieID(header)
ci := light.StorageTrieID(sti, crypto.Keccak256Hash(testContractAddr[:]), common.Hash{})
return &light.CodeRequest{Id: ci, Hash: crypto.Keccak256Hash(testContractCodeDeployed)}
}

func testAccess(t *testing.T, protocol int, fn accessTestFn) {
// Assemble the test environment
peers := newPeerSet()
dist := newRequestDistributor(peers, make(chan struct{}))
rm := newRetrieveManager(peers, dist, nil)
db, _ := ethdb.NewMemDatabase()
ldb, _ := ethdb.NewMemDatabase()
odr := NewLesOdr(ldb, rm)
```

```
pm := newTestProtocolManagerMust(t, false, 4, testChainGen, nil, nil, db)
lpm := newTestProtocolManagerMust(t, true, 0, nil, peers, odr, ldb)
_, err1, lpeer, err2 := newTestPeerPair("peer", protocol, pm, lpm)
select {
case <-time.After(time.Millisecond * 100):
case err := <-err1:
t.Fatalf("peer 1 handshake error: %v", err)
case err := <-err2:
t.Fatalf("peer 1 handshake error: %v", err)
}

lpm.synchronise(lpeer)

test := func(expFail uint64) {
for i := uint64(0); i <= pm.blockchain.CurrentHeader().Number.Uint64(); i++ {
bhash := core.GetCanonicalHash(db, i)
if req := fn(ldb, bhash, i); req != nil {
ctx, cancel := context.WithTimeout(context.Background(), 200*time.Millisecond)
defer cancel()

err := odr.Retrieve(ctx, req)
got := err == nil
exp := i < expFail
if exp && !got {
t.Errorf("object retrieval failed")
}
if !exp && got {
t.Errorf("unexpected object retrieval success")
}
}
}
}

// temporarily remove peer to test odr fails
peers.Unregister(lpeer.id)
time.Sleep(time.Millisecond * 10) // ensure that all peerSetNotify callbacks are executed
// expect retrievals to fail (except genesis block) without a les peer
test(0)

peers.Register(lpeer)
time.Sleep(time.Millisecond * 10) // ensure that all peerSetNotify callbacks are executed
```

```
lpeer.lock.Lock()
lpeer.hasBlock = func(common.Hash, uint64) bool { return true }
lpeer.lock.Unlock()
// expect all retrievals to pass
test(5)
}

5:F:\git\coin\ethereum\go-ethereum\les\retrieve.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package light implements on-demand retrieval capable state and chain objects
// for the Ethereum Light Client.
package les

import (
"context"
"crypto/rand"
"encoding/binary"
"sync"
"time"

"github.com/ethereum/go-ethereum/common/mclock"
)

var (
retryQueue        = time.Millisecond * 100
softRequestTimeout = time.Millisecond * 500
hardRequestTimeout = time.Second * 10
)

// retrieveManager is a layer on top of requestDistributor which takes care of
// matching replies by request ID and handles timeouts and resends if necessary.
type retrieveManager struct {
dist       *requestDistributor
peers      *peerSet
serverPool peerSelector

lock     sync.RWMutex
sentReqs map[uint64]*sentReq
}

// validatorFunc is a function that processes a reply message
```

```
type validatorFunc func(distPeer, *Msg) error

// peerSelector receives feedback info about response times and timeouts
type peerSelector interface {
adjustResponseTime(*poolEntry, time.Duration, bool)
}

// sentReq represents a request sent and tracked by retrieveManager
type sentReq struct {
rm       *retrieveManager
req      *distReq
id       uint64
validate validatorFunc

eventsCh chan reqPeerEvent
stopCh   chan struct{}
stopped  bool
err      error

lock   sync.RWMutex // protect access to sentTo map
sentTo map[distPeer]sentReqToPeer

reqQueued    bool // a request has been queued but not sent
reqSent      bool // a request has been sent but not timed out
reqSrtoCount int  // number of requests that reached soft (but not hard) timeout
}

// sentReqToPeer notifies the request-from-peer goroutine (tryRequest) about a response
// delivered by the given peer. Only one delivery is allowed per request per peer,
// after which delivered is set to true, the validity of the response is sent on the
// valid channel and no more responses are accepted.
type sentReqToPeer struct {
delivered bool
valid     chan bool
}

// reqPeerEvent is sent by the request-from-peer goroutine (tryRequest) to the
// request state machine (retrieveLoop) through the eventsCh channel.
type reqPeerEvent struct {
event int
peer  distPeer
}
```

```
const (
rpSent = iota // if peer == nil, not sent (no suitable peers)
rpSoftTimeout
rpHardTimeout
rpDeliveredValid
rpDeliveredInvalid
)

// newRetrieveManager creates the retrieve manager
func newRetrieveManager(peers *peerSet, dist *requestDistributor, serverPool peerSelector)
*retrieveManager {
return &retrieveManager{
peers:      peers,
dist:       dist,
serverPool: serverPool,
sentReqs:   make(map[uint64]*sentReq),
}
}

// retrieve sends a request (to multiple peers if necessary) and waits for an answer
// that is delivered through the deliver function and successfully validated by the
// validator callback. It returns when a valid answer is delivered or the context is
// cancelled.
func (rm *retrieveManager) retrieve(ctx context.Context, reqID uint64, req *distReq, val
validatorFunc) error {
sentReq := rm.sendReq(reqID, req, val)
select {
case <-sentReq.stopCh:
case <-ctx.Done():
sentReq.stop(ctx.Err())
}
return sentReq.getError()
}

// sendReq starts a process that keeps trying to retrieve a valid answer for a
// request from any suitable peers until stopped or succeeded.
func (rm *retrieveManager) sendReq(reqID uint64, req *distReq, val validatorFunc) *sentReq {
r := &sentReq{
rm:       rm,
req:      req,
id:       reqID,
```

```
sentTo:   make(map[distPeer]sentReqToPeer),
stopCh:   make(chan struct{}),
eventsCh: make(chan reqPeerEvent, 10),
validate: val,
}

canSend := req.canSend
req.canSend = func(p distPeer) bool {
// add an extra check to canSend: the request has not been sent to the same peer before
r.lock.RLock()
_, sent := r.sentTo[p]
r.lock.RUnlock()
return !sent && canSend(p)
}

request := req.request
req.request = func(p distPeer) func() {
// before actually sending the request, put an entry into the sentTo map
r.lock.Lock()
r.sentTo[p] = sentReqToPeer{false, make(chan bool, 1)}
r.lock.Unlock()
return request(p)
}
rm.lock.Lock()
rm.sentReqs[reqID] = r
rm.lock.Unlock()

go r.retrieveLoop()
return r
}

// deliver is called by the LES protocol manager to deliver reply messages to waiting requests
func (rm *retrieveManager) deliver(peer distPeer, msg *Msg) error {
rm.lock.RLock()
req, ok := rm.sentReqs[msg.ReqID]
rm.lock.RUnlock()

if ok {
return req.deliver(peer, msg)
}
return errResp(ErrUnexpectedResponse, "reqID = %v", msg.ReqID)
}
```

```
// reqStateFn represents a state of the retrieve loop state machine
type reqStateFn func() reqStateFn

// retrieveLoop is the retrieval state machine event loop
func (r *sentReq) retrieveLoop() {
go r.tryRequest()
r.reqQueued = true
state := r.stateRequesting

for state != nil {
state = state()
}

r.rm.lock.Lock()
delete(r.rm.sentReqs, r.id)
r.rm.lock.Unlock()
}

// stateRequesting: a request has been queued or sent recently; when it reaches soft timeout,
// a new request is sent to a new peer
func (r *sentReq) stateRequesting() reqStateFn {
select {
case ev := <-r.eventsCh:
r.update(ev)
switch ev.event {
case rpSent:
if ev.peer == nil {
// request send failed, no more suitable peers
if r.waiting() {
// we are already waiting for sent requests which may succeed so keep waiting
return r.stateNoMorePeers
}
// nothing to wait for, no more peers to ask, return with error
r.stop(ErrNoPeers)
// no need to go to stopped state because waiting() already returned false
return nil
}
case rpSoftTimeout:
// last request timed out, try asking a new peer
go r.tryRequest()
r.reqQueued = true
```

```
return r.stateRequesting
case rpDeliveredValid:
r.stop(nil)
return r.stateStopped
}
return r.stateRequesting
case <-r.stopCh:
return r.stateStopped
}
}

// stateNoMorePeers: could not send more requests because no suitable peers are available.
// Peers may become suitable for a certain request later or new peers may appear so we
// keep trying.
func (r *sentReq) stateNoMorePeers() reqStateFn {
select {
case <-time.After(retryQueue):
go r.tryRequest()
r.reqQueued = true
return r.stateRequesting
case ev := <-r.eventsCh:
r.update(ev)
if ev.event == rpDeliveredValid {
r.stop(nil)
return r.stateStopped
}
return r.stateNoMorePeers
case <-r.stopCh:
return r.stateStopped
}
}

// stateStopped: request succeeded or cancelled, just waiting for some peers
// to either answer or time out hard
func (r *sentReq) stateStopped() reqStateFn {
for r.waiting() {
r.update(<-r.eventsCh)
}
return nil
}

// update updates the queued/sent flags and timed out peers counter according to the event
```

```
func (r *sentReq) update(ev reqPeerEvent) {
switch ev.event {
case rpSent:
r.reqQueued = false
if ev.peer != nil {
r.reqSent = true
}
case rpSoftTimeout:
r.reqSent = false
r.reqSrtoCount++
case rpHardTimeout, rpDeliveredValid, rpDeliveredInvalid:
r.reqSrtoCount--
}
}

// waiting returns true if the retrieval mechanism is waiting for an answer from
// any peer
func (r *sentReq) waiting() bool {
return r.reqQueued || r.reqSent || r.reqSrtoCount > 0
}

// tryRequest tries to send the request to a new peer and waits for it to either
// succeed or time out if it has been sent. It also sends the appropriate reqPeerEvent
// messages to the request's event channel.
func (r *sentReq) tryRequest() {
sent := r.rm.dist.queue(r.req)
var p distPeer
select {
case p = <-sent:
case <-r.stopCh:
if r.rm.dist.cancel(r.req) {
p = nil
} else {
p = <-sent
}
}

r.eventsCh <- reqPeerEvent{rpSent, p}
if p == nil {
return
}
```

```
reqSent := mclock.Now()
srto, hrto := false, false

r.lock.RLock()
s, ok := r.sentTo[p]
r.lock.RUnlock()
if !ok {
panic(nil)
}

defer func() {
// send feedback to server pool and remove peer if hard timeout happened
pp, ok := p.(*peer)
if ok && r.rm.serverPool != nil {
respTime := time.Duration(mclock.Now() - reqSent)
r.rm.serverPool.adjustResponseTime(pp.poolEntry, respTime, srto)
}
if hrto {
pp.Log().Debug("Request timed out hard")
if r.rm.peers != nil {
r.rm.peers.Unregister(pp.id)
}
}

r.lock.Lock()
delete(r.sentTo, p)
r.lock.Unlock()
}()

select {
case ok := <-s.valid:
if ok {
r.eventsCh <- reqPeerEvent{rpDeliveredValid, p}
} else {
r.eventsCh <- reqPeerEvent{rpDeliveredInvalid, p}
}
return
case <-time.After(softRequestTimeout):
srto = true
r.eventsCh <- reqPeerEvent{rpSoftTimeout, p}
}
```

```
select {
case ok := <-s.valid:
if ok {
r.eventsCh <- reqPeerEvent{rpDeliveredValid, p}
} else {
r.eventsCh <- reqPeerEvent{rpDeliveredInvalid, p}
}
case <-time.After(hardRequestTimeout):
hrto = true
r.eventsCh <- reqPeerEvent{rpHardTimeout, p}
}
}

// deliver a reply belonging to this request
func (r *sentReq) deliver(peer distPeer, msg *Msg) error {
r.lock.Lock()
defer r.lock.Unlock()

s, ok := r.sentTo[peer]
if !ok || s.delivered {
return errResp(ErrUnexpectedResponse, "reqID = %v", msg.ReqID)
}
valid := r.validate(peer, msg) == nil
r.sentTo[peer] = sentReqToPeer{true, s.valid}
s.valid <- valid
if !valid {
return errResp(ErrInvalidResponse, "reqID = %v", msg.ReqID)
}
return nil
}

// stop stops the retrieval process and sets an error code that will be returned
// by getError
func (r *sentReq) stop(err error) {
r.lock.Lock()
if !r.stopped {
r.stopped = true
r.err = err
close(r.stopCh)
}
r.lock.Unlock()
}
```

```
// getError returns any retrieval error (either internally generated or set by the
// stop function) after stopCh has been closed
func (r *sentReq) getError() error {
return r.err
}

// genReqID generates a new random request ID
func genReqID() uint64 {
var rnd [8]byte
rand.Read(rnd[:])
return binary.BigEndian.Uint64(rnd[:])
}

6:F:\git\coin\ethereum\go-ethereum\les\server.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package les implements the Light Ethereum Subprotocol.
package les

import (
"encoding/binary"
"math"
"sync"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/eth"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/les/flowcontrol"
"github.com/ethereum/go-ethereum/light"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/p2p"
"github.com/ethereum/go-ethereum/p2p/discv5"
"github.com/ethereum/go-ethereum/rlp"
"github.com/ethereum/go-ethereum/trie"
)

type LesServer struct {
protocolManager *ProtocolManager
```

```
fcManager       *flowcontrol.ClientManager // nil if our node is client only
fcCostStats     *requestCostStats
defParams       *flowcontrol.ServerParams
lesTopic        discv5.Topic
quitSync        chan struct{}
stopped         bool
}

func NewLesServer(eth *eth.Ethereum, config *eth.Config) (*LesServer, error) {
quitSync := make(chan struct{})
pm, err := NewProtocolManager(eth.BlockChain().Config(), false, config.NetworkId,
eth.EventMux(), eth.Engine(), newPeerSet(), eth.BlockChain(), eth.TxPool(), eth.ChainDb(), nil, nil,
quitSync, new(sync.WaitGroup))
if err != nil {
return nil, err
}
pm.blockLoop()

srv := &LesServer{
protocolManager: pm,
quitSync:        quitSync,
lesTopic:        lesTopic(eth.BlockChain().Genesis().Hash()),
}
pm.server = srv

srv.defParams = &flowcontrol.ServerParams{
BufLimit:    300000000,
MinRecharge: 50000,
}
srv.fcManager = flowcontrol.NewClientManager(uint64(config.LightServ), 10, 1000000000)
srv.fcCostStats = newCostStats(eth.ChainDb())
return srv, nil
}

func (s *LesServer) Protocols() []p2p.Protocol {
return s.protocolManager.SubProtocols
}

// Start starts the LES server
func (s *LesServer) Start(srvr *p2p.Server) {
s.protocolManager.Start()
go func() {
```

```
logger := log.New("topic", s.lesTopic)
logger.Info("Starting topic registration")
defer logger.Info("Terminated topic registration")

srvr.DiscV5.RegisterTopic(s.lesTopic, s.quitSync)
}()
}

// Stop stops the LES service
func (s *LesServer) Stop() {
s.fcCostStats.store()
s.fcManager.Stop()
go func() {
<-s.protocolManager.noMorePeers
}()
s.protocolManager.Stop()
}

type requestCosts struct {
baseCost, reqCost uint64
}

type requestCostTable map[uint64]*requestCosts

type RequestCostList []struct {
MsgCode, BaseCost, ReqCost uint64
}

func (list RequestCostList) decode() requestCostTable {
table := make(requestCostTable)
for _, e := range list {
table[e.MsgCode] = &requestCosts{
baseCost: e.BaseCost,
reqCost:  e.ReqCost,
}
}
return table
}

func (table requestCostTable) encode() RequestCostList {
list := make(RequestCostList, len(table))
for idx, code := range reqList {
```

```
list[idx].MsgCode = code
list[idx].BaseCost = table[code].baseCost
list[idx].ReqCost = table[code].reqCost
}
return list
}

type linReg struct {
sumX, sumY, sumXX, sumXY float64
cnt                      uint64
}

const linRegMaxCnt = 100000

func (l *linReg) add(x, y float64) {
if l.cnt >= linRegMaxCnt {
sub := float64(l.cnt+1-linRegMaxCnt) / linRegMaxCnt
l.sumX -= l.sumX * sub
l.sumY -= l.sumY * sub
l.sumXX -= l.sumXX * sub
l.sumXY -= l.sumXY * sub
l.cnt = linRegMaxCnt - 1
}
l.cnt++
l.sumX += x
l.sumY += y
l.sumXX += x * x
l.sumXY += x * y
}

func (l *linReg) calc() (b, m float64) {
if l.cnt == 0 {
return 0, 0
}
cnt := float64(l.cnt)
d := cnt*l.sumXX - l.sumX*l.sumX
if d < 0.001 {
return l.sumY / cnt, 0
}
m = (cnt*l.sumXY - l.sumX*l.sumY) / d
b = (l.sumY / cnt) - (m * l.sumX / cnt)
return b, m
```

```
}

func (l *linReg) toBytes() []byte {
var arr [40]byte
binary.BigEndian.PutUint64(arr[0:8], math.Float64bits(l.sumX))
binary.BigEndian.PutUint64(arr[8:16], math.Float64bits(l.sumY))
binary.BigEndian.PutUint64(arr[16:24], math.Float64bits(l.sumXX))
binary.BigEndian.PutUint64(arr[24:32], math.Float64bits(l.sumXY))
binary.BigEndian.PutUint64(arr[32:40], l.cnt)
return arr[:]
}

func linRegFromBytes(data []byte) *linReg {
if len(data) != 40 {
return nil
}
l := &linReg{}
l.sumX = math.Float64frombits(binary.BigEndian.Uint64(data[0:8]))
l.sumY = math.Float64frombits(binary.BigEndian.Uint64(data[8:16]))
l.sumXX = math.Float64frombits(binary.BigEndian.Uint64(data[16:24]))
l.sumXY = math.Float64frombits(binary.BigEndian.Uint64(data[24:32]))
l.cnt = binary.BigEndian.Uint64(data[32:40])
return l
}

type requestCostStats struct {
lock  sync.RWMutex
db    ethdb.Database
stats map[uint64]*linReg
}

type requestCostStatsRlp []struct {
MsgCode uint64
Data    []byte
}

var rcStatsKey = []byte("_requestCostStats")

func newCostStats(db ethdb.Database) *requestCostStats {
stats := make(map[uint64]*linReg)
for _, code := range reqList {
stats[code] = &linReg{cnt: 100}
```

```
}

if db != nil {
data, err := db.Get(rcStatsKey)
var statsRlp requestCostStatsRlp
if err == nil {
err = rlp.DecodeBytes(data, &statsRlp)
}
if err == nil {
for _, r := range statsRlp {
if stats[r.MsgCode] != nil {
if l := linRegFromBytes(r.Data); l != nil {
stats[r.MsgCode] = l
}
}
}
}
}

return &requestCostStats{
db:    db,
stats: stats,
}
}

func (s *requestCostStats) store() {
s.lock.Lock()
defer s.lock.Unlock()

statsRlp := make(requestCostStatsRlp, len(reqList))
for i, code := range reqList {
statsRlp[i].MsgCode = code
statsRlp[i].Data = s.stats[code].toBytes()
}

if data, err := rlp.EncodeToBytes(statsRlp); err == nil {
s.db.Put(rcStatsKey, data)
}
}

func (s *requestCostStats) getCurrentList() RequestCostList {
s.lock.Lock()
```

```
defer s.lock.Unlock()

list := make(RequestCostList, len(reqList))
//fmt.Println("RequestCostList")
for idx, code := range reqList {
b, m := s.stats[code].calc()
//fmt.Println(code, s.stats[code].cnt, b/1000000, m/1000000)
if m < 0 {
b += m
m = 0
}
if b < 0 {
b = 0
}

list[idx].MsgCode = code
list[idx].BaseCost = uint64(b * 2)
list[idx].ReqCost = uint64(m * 2)
}
return list
}

func (s *requestCostStats) update(msgCode, reqCnt, cost uint64) {
s.lock.Lock()
defer s.lock.Unlock()

c, ok := s.stats[msgCode]
if !ok || reqCnt == 0 {
return
}
c.add(float64(reqCnt), float64(cost))
}

func (pm *ProtocolManager) blockLoop() {
pm.wg.Add(1)
sub := pm.eventMux.Subscribe(core.ChainHeadEvent{})
newCht := make(chan struct{}, 10)
newCht <- struct{}{}
go func() {
var mu sync.Mutex
var lastHead *types.Header
lastBroadcastTd := common.Big0
```

```
for {
select {
case ev := <-sub.Chan():
peers := pm.peers.AllPeers()
if len(peers) > 0 {
header := ev.Data.(core.ChainHeadEvent).Block.Header()
hash := header.Hash()
number := header.Number.Uint64()
td := core.GetTd(pm.chainDb, hash, number)
if td != nil && td.Cmp(lastBroadcastTd) > 0 {
var reorg uint64
if lastHead != nil {
reorg = lastHead.Number.Uint64() - core.FindCommonAncestor(pm.chainDb, header,
lastHead).Number.Uint64()
}
lastHead = header
lastBroadcastTd = td

log.Debug("Announcing block to peers", "number", number, "hash", hash, "td", td, "reorg", reorg)

announce := announceData{Hash: hash, Number: number, Td: td, ReorgDepth: reorg}
for _, p := range peers {
select {
case p.announceChn <- announce:
default:
pm.removePeer(p.id)
}
}
}
}
newCht <- struct{}{}
case <-newCht:
go func() {
mu.Lock()
more := makeCht(pm.chainDb)
mu.Unlock()
if more {
time.Sleep(time.Millisecond * 10)
newCht <- struct{}{}
}
}()
case <-pm.quitSync:
```

```
sub.Unsubscribe()
pm.wg.Done()
return
}
}
}()
}

var (
lastChtKey = []byte("LastChtNumber") // chtNum (uint64 big endian)
chtPrefix  = []byte("cht")           // chtPrefix + chtNum (uint64 big endian) -> trie root hash
)

func getChtRoot(db ethdb.Database, num uint64) common.Hash {
var encNumber [8]byte
binary.BigEndian.PutUint64(encNumber[:], num)
data, _ := db.Get(append(chtPrefix, encNumber[:]...))
return common.BytesToHash(data)
}

func storeChtRoot(db ethdb.Database, num uint64, root common.Hash) {
var encNumber [8]byte
binary.BigEndian.PutUint64(encNumber[:], num)
db.Put(append(chtPrefix, encNumber[:]...), root[:])
}

func makeCht(db ethdb.Database) bool {
headHash := core.GetHeadBlockHash(db)
headNum := core.GetBlockNumber(db, headHash)

var newChtNum uint64
if headNum > light.ChtConfirmations {
newChtNum = (headNum - light.ChtConfirmations) / light.ChtFrequency
}

var lastChtNum uint64
data, _ := db.Get(lastChtKey)
if len(data) == 8 {
lastChtNum = binary.BigEndian.Uint64(data[:])
}
if newChtNum <= lastChtNum {
return false
```

```
}

var t *trie.Trie
if lastChtNum > 0 {
var err error
t, err = trie.New(getChtRoot(db, lastChtNum), db)
if err != nil {
lastChtNum = 0
}
}
if lastChtNum == 0 {
t, _ = trie.New(common.Hash{}, db)
}

for num := lastChtNum * light.ChtFrequency; num < (lastChtNum+1)*light.ChtFrequency; num++ {
hash := core.GetCanonicalHash(db, num)
if hash == (common.Hash{}) {
panic("Canonical hash not found")
}
td := core.GetTd(db, hash, num)
if td == nil {
panic("TD not found")
}
var encNumber [8]byte
binary.BigEndian.PutUint64(encNumber[:], num)
var node light.ChtNode
node.Hash = hash
node.Td = td
data, _ := rlp.EncodeToBytes(node)
t.Update(encNumber[:], data)
}

root, err := t.Commit()
if err != nil {
lastChtNum = 0
} else {
lastChtNum++

log.Trace("Generated CHT", "number", lastChtNum, "root", root.Hex())

storeChtRoot(db, lastChtNum, root)
var data [8]byte
```

```
binary.BigEndian.PutUint64(data[:], lastChtNum)
db.Put(lastChtKey, data[:])
}

return newChtNum > lastChtNum
}

7:F:\git\coin\ethereum\go-ethereum\les\serverpool.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package les implements the Light Ethereum Subprotocol.
package les

import (
"fmt"
"io"
"math"
"math/rand"
"net"
"strconv"
"sync"
"time"

"github.com/ethereum/go-ethereum/common/mclock"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/p2p"
"github.com/ethereum/go-ethereum/p2p/discover"
"github.com/ethereum/go-ethereum/p2p/discv5"
"github.com/ethereum/go-ethereum/rlp"
)

const (
// After a connection has been ended or timed out, there is a waiting period
// before it can be selected for connection again.
// waiting period = base delay * (1 + random(1))
// base delay = shortRetryDelay for the first shortRetryCnt times after a
// successful connection, after that longRetryDelay is applied
shortRetryCnt   = 5
shortRetryDelay = time.Second * 5
longRetryDelay  = time.Minute * 10
// maxNewEntries is the maximum number of newly discovered (never connected) nodes.
```

```
// If the limit is reached, the least recently discovered one is thrown out.
maxNewEntries = 1000
// maxKnownEntries is the maximum number of known (already connected) nodes.
// If the limit is reached, the least recently connected one is thrown out.
// (not that unlike new entries, known entries are persistent)
maxKnownEntries = 1000
// target for simultaneously connected servers
targetServerCount = 5
// target for servers selected from the known table
// (we leave room for trying new ones if there is any)
targetKnownSelect = 3
// after dialTimeout, consider the server unavailable and adjust statistics
dialTimeout = time.Second * 30
// targetConnTime is the minimum expected connection duration before a server
// drops a client without any specific reason
targetConnTime = time.Minute * 10
// new entry selection weight calculation based on most recent discovery time:
// unity until discoverExpireStart, then exponential decay with discoverExpireConst
discoverExpireStart = time.Minute * 20
discoverExpireConst = time.Minute * 20
// known entry selection weight is dropped by a factor of exp(-failDropLn) after
// each unsuccessful connection (restored after a successful one)
failDropLn = 0.1
// known node connection success and quality statistics have a long term average
// and a short term value which is adjusted exponentially with a factor of
// pstatRecentAdjust with each dial/connection and also returned exponentially
// to the average with the time constant pstatReturnToMeanTC
pstatRecentAdjust   = 0.1
pstatReturnToMeanTC = time.Hour
// node address selection weight is dropped by a factor of exp(-addrFailDropLn) after
// each unsuccessful connection (restored after a successful one)
addrFailDropLn = math.Ln2
// responseScoreTC and delayScoreTC are exponential decay time constants for
// calculating selection chances from response times and block delay times
responseScoreTC = time.Millisecond * 100
delayScoreTC    = time.Second * 5
timeoutPow      = 10
// peerSelectMinWeight is added to calculated weights at request peer selection
// to give poorly performing peers a little chance of coming back
peerSelectMinWeight = 0.005
// initStatsWeight is used to initialize previously unknown peers with good
// statistics to give a chance to prove themselves
```

```
initStatsWeight = 1
)

// serverPool implements a pool for storing and selecting newly discovered and already
// known light server nodes. It received discovered nodes, stores statistics about
// known nodes and takes care of always having enough good quality servers connected.
type serverPool struct {
db     ethdb.Database
dbKey  []byte
server *p2p.Server
quit   chan struct{}
wg     *sync.WaitGroup
connWg sync.WaitGroup

topic discv5.Topic

discSetPeriod chan time.Duration
discNodes     chan *discv5.Node
discLookups   chan bool

entries              map[discover.NodeID]*poolEntry
lock                 sync.Mutex
timeout, enableRetry chan *poolEntry
adjustStats          chan poolStatAdjust

knownQueue, newQueue       poolEntryQueue
knownSelect, newSelect     *weightedRandomSelect
knownSelected, newSelected int
fastDiscover               bool
}

// newServerPool creates a new serverPool instance
func newServerPool(db ethdb.Database, quit chan struct{}, wg *sync.WaitGroup) *serverPool {
pool := &serverPool{
db:          db,
quit:        quit,
wg:          wg,
entries:     make(map[discover.NodeID]*poolEntry),
timeout:     make(chan *poolEntry, 1),
adjustStats:  make(chan poolStatAdjust, 100),
enableRetry:  make(chan *poolEntry, 1),
knownSelect:  newWeightedRandomSelect(),
```

```
newSelect:    newWeightedRandomSelect(),
fastDiscover: true,
}
pool.knownQueue = newPoolEntryQueue(maxKnownEntries, pool.removeEntry)
pool.newQueue = newPoolEntryQueue(maxNewEntries, pool.removeEntry)
return pool
}

func (pool *serverPool) start(server *p2p.Server, topic discv5.Topic) {
pool.server = server
pool.topic = topic
pool.dbKey = append([]byte("serverPool/"), []byte(topic)...)
pool.wg.Add(1)
pool.loadNodes()

go pool.eventLoop()

pool.checkDial()
if pool.server.DiscV5 != nil {
pool.discSetPeriod = make(chan time.Duration, 1)
pool.discNodes = make(chan *discv5.Node, 100)
pool.discLookups = make(chan bool, 100)
go pool.server.DiscV5.SearchTopic(pool.topic, pool.discSetPeriod, pool.discNodes,
pool.discLookups)
}
}

// connect should be called upon any incoming connection. If the connection has been
// dialed by the server pool recently, the appropriate pool entry is returned.
// Otherwise, the connection should be rejected.
// Note that whenever a connection has been accepted and a pool entry has been returned,
// disconnect should also always be called.
func (pool *serverPool) connect(p *peer, ip net.IP, port uint16) *poolEntry {
pool.lock.Lock()
defer pool.lock.Unlock()
entry := pool.entries[p.ID()]
if entry == nil {
entry = pool.findOrNewNode(p.ID(), ip, port)
}
p.Log().Debug("Connecting to new peer", "state", entry.state)
if entry.state == psConnected || entry.state == psRegistered {
return nil
```

```
}
pool.connWg.Add(1)
entry.peer = p
entry.state = psConnected
addr := &poolEntryAddress{
ip:       ip,
port:     port,
lastSeen: mclock.Now(),
}
entry.lastConnected = addr
entry.addr = make(map[string]*poolEntryAddress)
entry.addr[addr.strKey()] = addr
entry.addrSelect = *newWeightedRandomSelect()
entry.addrSelect.update(addr)
return entry
}

// registered should be called after a successful handshake
func (pool *serverPool) registered(entry *poolEntry) {
log.Debug("Registered new entry", "enode", entry.id)
pool.lock.Lock()
defer pool.lock.Unlock()

entry.state = psRegistered
entry.regTime = mclock.Now()
if !entry.known {
pool.newQueue.remove(entry)
entry.known = true
}
pool.knownQueue.setLatest(entry)
entry.shortRetry = shortRetryCnt
}

// disconnect should be called when ending a connection. Service quality statistics
// can be updated optionally (not updated if no registration happened, in this case
// only connection statistics are updated, just like in case of timeout)
func (pool *serverPool) disconnect(entry *poolEntry) {
log.Debug("Disconnected old entry", "enode", entry.id)
pool.lock.Lock()
defer pool.lock.Unlock()

if entry.state == psRegistered {
```

```
connTime := mclock.Now() - entry.regTime
connAdjust := float64(connTime) / float64(targetConnTime)
if connAdjust > 1 {
connAdjust = 1
}
stopped := false
select {
case <-pool.quit:
stopped = true
default:
}
if stopped {
entry.connectStats.add(1, connAdjust)
} else {
entry.connectStats.add(connAdjust, 1)
}
}

entry.state = psNotConnected
if entry.knownSelected {
pool.knownSelected--
} else {
pool.newSelected--
}
pool.setRetryDial(entry)
pool.connWg.Done()
}

const (
pseBlockDelay = iota
pseResponseTime
pseResponseTimeout
)

// poolStatAdjust records are sent to adjust peer block delay/response time statistics
type poolStatAdjust struct {
adjustType int
entry      *poolEntry
time       time.Duration
}

// adjustBlockDelay adjusts the block announce delay statistics of a node
```

```
func (pool *serverPool) adjustBlockDelay(entry *poolEntry, time time.Duration) {
if entry == nil {
return
}
pool.adjustStats <- poolStatAdjust{pseBlockDelay, entry, time}
}

// adjustResponseTime adjusts the request response time statistics of a node
func (pool *serverPool) adjustResponseTime(entry *poolEntry, time time.Duration, timeout bool) {
if entry == nil {
return
}
if timeout {
pool.adjustStats <- poolStatAdjust{pseResponseTimeout, entry, time}
} else {
pool.adjustStats <- poolStatAdjust{pseResponseTime, entry, time}
}
}

// eventLoop handles pool events and mutex locking for all internal functions
func (pool *serverPool) eventLoop() {
lookupCnt := 0
var convTime mclock.AbsTime
if pool.discSetPeriod != nil {
pool.discSetPeriod <- time.Millisecond * 100
}
for {
select {
case entry := <-pool.timeout:
pool.lock.Lock()
if !entry.removed {
pool.checkDialTimeout(entry)
}
pool.lock.Unlock()

case entry := <-pool.enableRetry:
pool.lock.Lock()
if !entry.removed {
entry.delayedRetry = false
pool.updateCheckDial(entry)
}
pool.lock.Unlock()
```

```
case adj := <-pool.adjustStats:
pool.lock.Lock()
switch adj.adjustType {
case pseBlockDelay:
adj.entry.delayStats.add(float64(adj.time), 1)
case pseResponseTime:
adj.entry.responseStats.add(float64(adj.time), 1)
adj.entry.timeoutStats.add(0, 1)
case pseResponseTimeout:
adj.entry.timeoutStats.add(1, 1)
}
pool.lock.Unlock()

case node := <-pool.discNodes:
pool.lock.Lock()
entry := pool.findOrNewNode(discover.NodeID(node.ID), node.IP, node.TCP)
pool.updateCheckDial(entry)
pool.lock.Unlock()

case conv := <-pool.discLookups:
if conv {
if lookupCnt == 0 {
convTime = mclock.Now()
}
lookupCnt++
if pool.fastDiscover && (lookupCnt == 50 || time.Duration(mclock.Now()-convTime) > time.Minute)
{
pool.fastDiscover = false
if pool.discSetPeriod != nil {
pool.discSetPeriod <- time.Minute
}
}
}

case <-pool.quit:
if pool.discSetPeriod != nil {
close(pool.discSetPeriod)
}
pool.connWg.Wait()
pool.saveNodes()
pool.wg.Done()
```

```
return

}
}
}

func (pool *serverPool) findOrNewNode(id discover.NodeID, ip net.IP, port uint16) *poolEntry {
now := mclock.Now()
entry := pool.entries[id]
if entry == nil {
log.Debug("Discovered new entry", "id", id)
entry = &poolEntry{
id:         id,
addr:       make(map[string]*poolEntryAddress),
addrSelect: *newWeightedRandomSelect(),
shortRetry: shortRetryCnt,
}
pool.entries[id] = entry
// initialize previously unknown peers with good statistics to give a chance to prove themselves
entry.connectStats.add(1, initStatsWeight)
entry.delayStats.add(0, initStatsWeight)
entry.responseStats.add(0, initStatsWeight)
entry.timeoutStats.add(0, initStatsWeight)
}
entry.lastDiscovered = now
addr := &poolEntryAddress{
ip:   ip,
port: port,
}
if a, ok := entry.addr[addr.strKey()]; ok {
addr = a
} else {
entry.addr[addr.strKey()] = addr
}
addr.lastSeen = now
entry.addrSelect.update(addr)
if !entry.known {
pool.newQueue.setLatest(entry)
}
return entry
}
```

```
// loadNodes loads known nodes and their statistics from the database
func (pool *serverPool) loadNodes() {
enc, err := pool.db.Get(pool.dbKey)
if err != nil {
return
}
var list []*poolEntry
err = rlp.DecodeBytes(enc, &list)
if err != nil {
log.Debug("Failed to decode node list", "err", err)
return
}
for _, e := range list {
log.Debug("Loaded server stats", "id", e.id, "fails", e.lastConnected.fails,
"conn", fmt.Sprintf("%v/%v", e.connectStats.avg, e.connectStats.weight),
"delay", fmt.Sprintf("%v/%v", time.Duration(e.delayStats.avg), e.delayStats.weight),
"response", fmt.Sprintf("%v/%v", time.Duration(e.responseStats.avg), e.responseStats.weight),
"timeout", fmt.Sprintf("%v/%v", e.timeoutStats.avg, e.timeoutStats.weight))
pool.entries[e.id] = e
pool.knownQueue.setLatest(e)
pool.knownSelect.update((*knownEntry)(e))
}
}

// saveNodes saves known nodes and their statistics into the database. Nodes are
// ordered from least to most recently connected.
func (pool *serverPool) saveNodes() {
list := make([]*poolEntry, len(pool.knownQueue.queue))
for i := range list {
list[i] = pool.knownQueue.fetchOldest()
}
enc, err := rlp.EncodeToBytes(list)
if err == nil {
pool.db.Put(pool.dbKey, enc)
}
}

// removeEntry removes a pool entry when the entry count limit is reached.
// Note that it is called by the new/known queues from which the entry has already
// been removed so removing it from the queues is not necessary.
func (pool *serverPool) removeEntry(entry *poolEntry) {
pool.newSelect.remove((*discoveredEntry)(entry))
```

```
pool.knownSelect.remove((*knownEntry)(entry))
entry.removed = true
delete(pool.entries, entry.id)
}

// setRetryDial starts the timer which will enable dialing a certain node again
func (pool *serverPool) setRetryDial(entry *poolEntry) {
delay := longRetryDelay
if entry.shortRetry > 0 {
entry.shortRetry--
delay = shortRetryDelay
}
delay += time.Duration(rand.Int63n(int64(delay) + 1))
entry.delayedRetry = true
go func() {
select {
case <-pool.quit:
case <-time.After(delay):
select {
case <-pool.quit:
case pool.enableRetry <- entry:
}
}
}()
}

// updateCheckDial is called when an entry can potentially be dialed again. It updates
// its selection weights and checks if new dials can/should be made.
func (pool *serverPool) updateCheckDial(entry *poolEntry) {
pool.newSelect.update((*discoveredEntry)(entry))
pool.knownSelect.update((*knownEntry)(entry))
pool.checkDial()
}

// checkDial checks if new dials can/should be made. It tries to select servers both
// based on good statistics and recent discovery.
func (pool *serverPool) checkDial() {
fillWithKnownSelects := !pool.fastDiscover
for pool.knownSelected < targetKnownSelect {
entry := pool.knownSelect.choose()
if entry == nil {
fillWithKnownSelects = false
```

```
break
}
pool.dial((*poolEntry)(entry.(*knownEntry)), true)
}
for pool.knownSelected+pool.newSelected < targetServerCount {
entry := pool.newSelect.choose()
if entry == nil {
break
}
pool.dial((*poolEntry)(entry.(*discoveredEntry)), false)
}
if fillWithKnownSelects {
// no more newly discovered nodes to select and since fast discover period
// is over, we probably won't find more in the near future so select more
// known entries if possible
for pool.knownSelected < targetServerCount {
entry := pool.knownSelect.choose()
if entry == nil {
break
}
pool.dial((*poolEntry)(entry.(*knownEntry)), true)
}
}
}

// dial initiates a new connection
func (pool *serverPool) dial(entry *poolEntry, knownSelected bool) {
if pool.server == nil || entry.state != psNotConnected {
return
}
entry.state = psDialed
entry.knownSelected = knownSelected
if knownSelected {
pool.knownSelected++
} else {
pool.newSelected++
}
addr := entry.addrSelect.choose().(*poolEntryAddress)
log.Debug("Dialing new peer", "lesaddr", entry.id.String()+"@"+addr.strKey(), "set",
len(entry.addr), "known", knownSelected)
entry.dialed = addr
go func() {
```

```
pool.server.AddPeer(discover.NewNode(entry.id, addr.ip, addr.port, addr.port))
select {
case <-pool.quit:
case <-time.After(dialTimeout):
select {
case <-pool.quit:
case pool.timeout <- entry:
}
}
}()
}

// checkDialTimeout checks if the node is still in dialed state and if so, resets it
// and adjusts connection statistics accordingly.
func (pool *serverPool) checkDialTimeout(entry *poolEntry) {
if entry.state != psDialed {
return
}
log.Debug("Dial timeout", "lesaddr", entry.id.String()+"@"+entry.dialed.strKey())
entry.state = psNotConnected
if entry.knownSelected {
pool.knownSelected--
} else {
pool.newSelected--
}
entry.connectStats.add(0, 1)
entry.dialed.fails++
pool.setRetryDial(entry)
}

const (
psNotConnected = iota
psDialed
psConnected
psRegistered
)

// poolEntry represents a server node and stores its current state and statistics.
type poolEntry struct {
peer                *peer
id                  discover.NodeID
addr                map[string]*poolEntryAddress
```

```
lastConnected, dialed *poolEntryAddress
addrSelect           weightedRandomSelect

lastDiscovered             mclock.AbsTime
known, knownSelected       bool
connectStats, delayStats   poolStats
responseStats, timeoutStats poolStats
state                      int
regTime                    mclock.AbsTime
queueIdx                   int
removed                    bool

delayedRetry bool
shortRetry   int
}

func (e *poolEntry) EncodeRLP(w io.Writer) error {
return rlp.Encode(w, []interface{}{e.id, e.lastConnected.ip, e.lastConnected.port,
e.lastConnected.fails, &e.connectStats, &e.delayStats, &e.responseStats, &e.timeoutStats})
}

func (e *poolEntry) DecodeRLP(s *rlp.Stream) error {
var entry struct {
ID                      discover.NodeID
IP                      net.IP
Port                    uint16
Fails                   uint
CStat, DStat, RStat, TStat poolStats
}
if err := s.Decode(&entry); err != nil {
return err
}
addr := &poolEntryAddress{ip: entry.IP, port: entry.Port, fails: entry.Fails, lastSeen: mclock.Now()}
e.id = entry.ID
e.addr = make(map[string]*poolEntryAddress)
e.addr[addr.strKey()] = addr
e.addrSelect = *newWeightedRandomSelect()
e.addrSelect.update(addr)
e.lastConnected = addr
e.connectStats = entry.CStat
e.delayStats = entry.DStat
e.responseStats = entry.RStat
```

```
e.timeoutStats = entry.TStat
e.shortRetry = shortRetryCnt
e.known = true
return nil
}

// discoveredEntry implements wrsItem
type discoveredEntry poolEntry

// Weight calculates random selection weight for newly discovered entries
func (e *discoveredEntry) Weight() int64 {
if e.state != psNotConnected || e.delayedRetry {
return 0
}
t := time.Duration(mclock.Now() - e.lastDiscovered)
if t <= discoverExpireStart {
return 1000000000
} else {
return int64(1000000000 * math.Exp(-float64(t-discoverExpireStart)/float64(discoverExpireConst)))
}
}

// knownEntry implements wrsItem
type knownEntry poolEntry

// Weight calculates random selection weight for known entries
func (e *knownEntry) Weight() int64 {
if e.state != psNotConnected || !e.known || e.delayedRetry {
return 0
}
return int64(1000000000 * e.connectStats.recentAvg() * math.Exp(-
float64(e.lastConnected.fails)*failDropLn-e.responseStats.recentAvg()/float64(responseScoreTC)-
e.delayStats.recentAvg()/float64(delayScoreTC)) * math.Pow((1-e.timeoutStats.recentAvg()),
timeoutPow))
}

// poolEntryAddress is a separate object because currently it is necessary to remember
// multiple potential network addresses for a pool entry. This will be removed after
// the final implementation of v5 discovery which will retrieve signed and serial
// numbered advertisements, making it clear which IP/port is the latest one.
type poolEntryAddress struct {
ip       net.IP
```

```
port     uint16
lastSeen mclock.AbsTime // last time it was discovered, connected or loaded from db
fails    uint           // connection failures since last successful connection (persistent)
}

func (a *poolEntryAddress) Weight() int64 {
t := time.Duration(mclock.Now() - a.lastSeen)
return int64(1000000*math.Exp(-float64(t)/float64(discoverExpireConst)-
float64(a.fails)*addrFailDropLn)) + 1
}

func (a *poolEntryAddress) strKey() string {
return a.ip.String() + ":" + strconv.Itoa(int(a.port))
}

// poolStats implement statistics for a certain quantity with a long term average
// and a short term value which is adjusted exponentially with a factor of
// pstatRecentAdjust with each update and also returned exponentially to the
// average with the time constant pstatReturnToMeanTC
type poolStats struct {
sum, weight, avg, recent float64
lastRecalc               mclock.AbsTime
}

// init initializes stats with a long term sum/update count pair retrieved from the database
func (s *poolStats) init(sum, weight float64) {
s.sum = sum
s.weight = weight
var avg float64
if weight > 0 {
avg = s.sum / weight
}
s.avg = avg
s.recent = avg
s.lastRecalc = mclock.Now()
}

// recalc recalculates recent value return-to-mean and long term average
func (s *poolStats) recalc() {
now := mclock.Now()
s.recent = s.avg + (s.recent-s.avg)*math.Exp(-float64(now-
s.lastRecalc)/float64(pstatReturnToMeanTC))
```

```
if s.sum == 0 {
s.avg = 0
} else {
if s.sum > s.weight*1e30 {
s.avg = 1e30
} else {
s.avg = s.sum / s.weight
}
}
s.lastRecalc = now
}

// add updates the stats with a new value
func (s *poolStats) add(value, weight float64) {
s.weight += weight
s.sum += value * weight
s.recalc()
}

// recentAvg returns the short-term adjusted average
func (s *poolStats) recentAvg() float64 {
s.recalc()
return s.recent
}

func (s *poolStats) EncodeRLP(w io.Writer) error {
return rlp.Encode(w, []interface{}{math.Float64bits(s.sum), math.Float64bits(s.weight)})
}

func (s *poolStats) DecodeRLP(st *rlp.Stream) error {
var stats struct {
SumUint, WeightUint uint64
}
if err := st.Decode(&stats); err != nil {
return err
}
s.init(math.Float64frombits(stats.SumUint), math.Float64frombits(stats.WeightUint))
return nil
}

// poolEntryQueue keeps track of its least recently accessed entries and removes
// them when the number of entries reaches the limit
```

```
type poolEntryQueue struct {
queue                map[int]*poolEntry // known nodes indexed by their latest lastConnCnt value
newPtr, oldPtr, maxCnt int
removeFromPool       func(*poolEntry)
}

// newPoolEntryQueue returns a new poolEntryQueue
func newPoolEntryQueue(maxCnt int, removeFromPool func(*poolEntry)) poolEntryQueue {
return poolEntryQueue{queue: make(map[int]*poolEntry), maxCnt: maxCnt, removeFromPool:
removeFromPool}
}

// fetchOldest returns and removes the least recently accessed entry
func (q *poolEntryQueue) fetchOldest() *poolEntry {
if len(q.queue) == 0 {
return nil
}
for {
if e := q.queue[q.oldPtr]; e != nil {
delete(q.queue, q.oldPtr)
q.oldPtr++
return e
}
q.oldPtr++
}
}

// remove removes an entry from the queue
func (q *poolEntryQueue) remove(entry *poolEntry) {
if q.queue[entry.queueIdx] == entry {
delete(q.queue, entry.queueIdx)
}
}

// setLatest adds or updates a recently accessed entry. It also checks if an old entry
// needs to be removed and removes it from the parent pool too with a callback function.
func (q *poolEntryQueue) setLatest(entry *poolEntry) {
if q.queue[entry.queueIdx] == entry {
delete(q.queue, entry.queueIdx)
} else {
if len(q.queue) == q.maxCnt {
e := q.fetchOldest()
```

```
q.remove(e)
q.removeFromPool(e)
}
}
entry.queueIdx = q.newPtr
q.queue[entry.queueIdx] = entry
q.newPtr++
}

8:F:\git\coin\ethereum\go-ethereum\les\sync.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package les

import (
"context"
"time"

"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/eth/downloader"
"github.com/ethereum/go-ethereum/light"
)

const (
//forceSyncCycle      = 10 * time.Second // Time interval to force syncs, even if few peers are available
minDesiredPeerCount = 5 // Amount of peers desired to start syncing
)

// syncer is responsible for periodically synchronising with the network, both
// downloading hashes and blocks as well as handling the announcement handler.
func (pm *ProtocolManager) syncer() {
// Start and ensure cleanup of sync mechanisms
//pm.fetcher.Start()
//defer pm.fetcher.Stop()
defer pm.downloader.Terminate()

// Wait for different events to fire synchronisation operations
//forceSync := time.Tick(forceSyncCycle)
for {
select {
case <-pm.newPeerCh:
```

```
/*// Make sure we have peers to select from, then sync
if pm.peers.Len() < minDesiredPeerCount {
break
}
go pm.synchronise(pm.peers.BestPeer())
*/
/*case <-forceSync:
// Force a sync even if not enough peers are present
go pm.synchronise(pm.peers.BestPeer())
*/
case <-pm.noMorePeers:
return
}
}
}

func (pm *ProtocolManager) needToSync(peerHead blockInfo) bool {
head := pm.blockchain.CurrentHeader()
currentTd := core.GetTd(pm.chainDb, head.Hash(), head.Number.Uint64())
return currentTd != nil && peerHead.Td.Cmp(currentTd) > 0
}

// synchronise tries to sync up our local block chain with a remote peer.
func (pm *ProtocolManager) synchronise(peer *peer) {
// Short circuit if no peers are available
if peer == nil {
return
}

// Make sure the peer's TD is higher than our own.
if !pm.needToSync(peer.headBlockInfo()) {
return
}

ctx, cancel := context.WithTimeout(context.Background(), time.Second*5)
defer cancel()
pm.blockchain.(*light.LightChain).SyncCht(ctx)
pm.downloader.Synchronise(peer.id, peer.Head(), peer.Td(), downloader.LightSync)
}

9:F:\git\coin\ethereum\go-ethereum\les\txrelay.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.
```

```
package les

import (
"sync"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/types"
)

type ltrInfo struct {
tx     *types.Transaction
sentTo map[*peer]struct{}
}

type LesTxRelay struct {
txSent       map[common.Hash]*ltrInfo
txPending    map[common.Hash]struct{}
ps           *peerSet
peerList     []*peer
peerStartPos int
lock         sync.RWMutex

reqDist *requestDistributor
}

func NewLesTxRelay(ps *peerSet, reqDist *requestDistributor) *LesTxRelay {
r := &LesTxRelay{
txSent:    make(map[common.Hash]*ltrInfo),
txPending: make(map[common.Hash]struct{}),
ps:        ps,
reqDist:   reqDist,
}
ps.notify(r)
return r
}

func (self *LesTxRelay) registerPeer(p *peer) {
self.lock.Lock()
defer self.lock.Unlock()

self.peerList = self.ps.AllPeers()
```

```
}

func (self *LesTxRelay) unregisterPeer(p *peer) {
self.lock.Lock()
defer self.lock.Unlock()

self.peerList = self.ps.AllPeers()
}

// send sends a list of transactions to at most a given number of peers at
// once, never resending any particular transaction to the same peer twice
func (self *LesTxRelay) send(txs types.Transactions, count int) {
sendTo := make(map[*peer]types.Transactions)

self.peerStartPos++ // rotate the starting position of the peer list
if self.peerStartPos >= len(self.peerList) {
self.peerStartPos = 0
}

for _, tx := range txs {
hash := tx.Hash()
ltr, ok := self.txSent[hash]
if !ok {
ltr = &ltrInfo{
tx:     tx,
sentTo: make(map[*peer]struct{}),
}
self.txSent[hash] = ltr
self.txPending[hash] = struct{}{}
}

if len(self.peerList) > 0 {
cnt := count
pos := self.peerStartPos
for {
peer := self.peerList[pos]
if _, ok := ltr.sentTo[peer]; !ok {
sendTo[peer] = append(sendTo[peer], tx)
ltr.sentTo[peer] = struct{}{}
cnt--
}
if cnt == 0 {
```

```
break // sent it to the desired number of peers
}
pos++
if pos == len(self.peerList) {
pos = 0
}
if pos == self.peerStartPos {
break // tried all available peers
}
}
}
}

for p, list := range sendTo {
pp := p
ll := list

reqID := genReqID()
rq := &distReq{
getCost: func(dp distPeer) uint64 {
peer := dp.(*peer)
return peer.GetRequestCost(SendTxMsg, len(ll))
},
canSend: func(dp distPeer) bool {
return dp.(*peer) == pp
},
request: func(dp distPeer) func() {
peer := dp.(*peer)
cost := peer.GetRequestCost(SendTxMsg, len(ll))
peer.fcServer.QueueRequest(reqID, cost)
return func() { peer.SendTxs(reqID, cost, ll) }
},
}
self.reqDist.queue(rq)
}
}

func (self *LesTxRelay) Send(txs types.Transactions) {
self.lock.Lock()
defer self.lock.Unlock()

self.send(txs, 3)
```

```
}

func (self *LesTxRelay) NewHead(head common.Hash, mined []common.Hash, rollback
[]common.Hash) {
self.lock.Lock()
defer self.lock.Unlock()

for _, hash := range mined {
delete(self.txPending, hash)
}

for _, hash := range rollback {
self.txPending[hash] = struct{}{}
}

if len(self.txPending) > 0 {
txs := make(types.Transactions, len(self.txPending))
i := 0
for hash := range self.txPending {
txs[i] = self.txSent[hash].tx
i++
}
self.send(txs, 1)
}
}

func (self *LesTxRelay) Discard(hashes []common.Hash) {
self.lock.Lock()
defer self.lock.Unlock()

for _, hash := range hashes {
delete(self.txSent, hash)
delete(self.txPending, hash)
}
}

10:F:\git\coin\ethereum\go-ethereum\light\lightchain.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package light

import (
```

```
"context"
"math/big"
"sync"
"sync/atomic"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/consensus"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/params"
"github.com/ethereum/go-ethereum/rlp"
"github.com/hashicorp/golang-lru"
)

var (
bodyCacheLimit  = 256
blockCacheLimit = 256
)

// LightChain represents a canonical chain that by default only handles block
// headers, downloading block bodies and receipts on demand through an ODR
// interface. It only does header validation during chain insertion.
type LightChain struct {
hc           *core.HeaderChain
chainDb      ethdb.Database
odr          OdrBackend
eventMux     *event.TypeMux
genesisBlock *types.Block

mu      sync.RWMutex
chainmu sync.RWMutex
procmu  sync.RWMutex

bodyCache    *lru.Cache // Cache for the most recent block bodies
bodyRLPCache *lru.Cache // Cache for the most recent block bodies in RLP encoded format
blockCache   *lru.Cache // Cache for the most recent entire blocks

quit    chan struct{}
```

```
running int32 // running must be called automically
// procInterrupt must be atomically called
procInterrupt int32 // interrupt signaler for block processing
wg            sync.WaitGroup

engine consensus.Engine
}

// NewLightChain returns a fully initialised light chain using information
// available in the database. It initialises the default Ethereum header
// validator.
func NewLightChain(odr OdrBackend, config *params.ChainConfig, engine consensus.Engine,
mux *event.TypeMux) (*LightChain, error) {
bodyCache, _ := lru.New(bodyCacheLimit)
bodyRLPCache, _ := lru.New(bodyCacheLimit)
blockCache, _ := lru.New(blockCacheLimit)

bc := &LightChain{
chainDb:      odr.Database(),
odr:          odr,
eventMux:     mux,
quit:         make(chan struct{}),
bodyCache:    bodyCache,
bodyRLPCache: bodyRLPCache,
blockCache:   blockCache,
engine:       engine,
}
var err error
bc.hc, err = core.NewHeaderChain(odr.Database(), config, bc.engine, bc.getProcInterrupt)
if err != nil {
return nil, err
}
bc.genesisBlock, _ = bc.GetBlockByNumber(NoOdr, 0)
if bc.genesisBlock == nil {
return nil, core.ErrNoGenesis
}
if bc.genesisBlock.Hash() == params.MainnetGenesisHash {
// add trusted CHT
WriteTrustedCht(bc.chainDb, TrustedCht{Number: 805, Root:
common.HexToHash("85e4286fe0a730390245c49de8476977afdae0eb5530b277f62a52b12313d
50f")})
log.Info("Added trusted CHT for mainnet")
```

```
}

if err := bc.loadLastState(); err != nil {
return nil, err
}
// Check the current state of the block hashes and make sure that we do not have any of the bad
blocks in our chain
for hash := range core.BadHashes {
if header := bc.GetHeaderByHash(hash); header != nil {
log.Error("Found bad hash, rewinding chain", "number", header.Number, "hash",
header.ParentHash)
bc.SetHead(header.Number.Uint64() - 1)
log.Error("Chain rewind was successful, resuming normal operation")
}
}
return bc, nil
}

func (self *LightChain) getProcInterrupt() bool {
return atomic.LoadInt32(&self.procInterrupt) == 1
}

// Odr returns the ODR backend of the chain
func (self *LightChain) Odr() OdrBackend {
return self.odr
}

// loadLastState loads the last known chain state from the database. This method
// assumes that the chain manager mutex is held.
func (self *LightChain) loadLastState() error {
if head := core.GetHeadHeaderHash(self.chainDb); head == (common.Hash{}) {
// Corrupt or empty database, init from scratch
self.Reset()
} else {
if header := self.GetHeaderByHash(head); header != nil {
self.hc.SetCurrentHeader(header)
}
}

// Issue a status log and return
header := self.hc.CurrentHeader()
headerTd := self.GetTd(header.Hash(), header.Number.Uint64())
```

```
log.Info("Loaded most recent local header", "number", header.Number, "hash", header.Hash(),
"td", headerTd)

return nil
}

// SetHead rewinds the local chain to a new head. Everything above the new
// head will be deleted and the new one set.
func (bc *LightChain) SetHead(head uint64) {
bc.mu.Lock()
defer bc.mu.Unlock()

bc.hc.SetHead(head, nil)
bc.loadLastState()
}

// GasLimit returns the gas limit of the current HEAD block.
func (self *LightChain) GasLimit() *big.Int {
self.mu.RLock()
defer self.mu.RUnlock()

return self.hc.CurrentHeader().GasLimit
}

// LastBlockHash return the hash of the HEAD block.
func (self *LightChain) LastBlockHash() common.Hash {
self.mu.RLock()
defer self.mu.RUnlock()

return self.hc.CurrentHeader().Hash()
}

// Status returns status information about the current chain such as the HEAD Td,
// the HEAD hash and the hash of the genesis block.
func (self *LightChain) Status() (td *big.Int, currentBlock common.Hash, genesisBlock
common.Hash) {
self.mu.RLock()
defer self.mu.RUnlock()

header := self.hc.CurrentHeader()
hash := header.Hash()
return self.GetTd(hash, header.Number.Uint64()), hash, self.genesisBlock.Hash()
```

```
}

// Reset purges the entire blockchain, restoring it to its genesis state.
func (bc *LightChain) Reset() {
bc.ResetWithGenesisBlock(bc.genesisBlock)
}

// ResetWithGenesisBlock purges the entire blockchain, restoring it to the
// specified genesis state.
func (bc *LightChain) ResetWithGenesisBlock(genesis *types.Block) {
// Dump the entire block chain and purge the caches
bc.SetHead(0)

bc.mu.Lock()
defer bc.mu.Unlock()

// Prepare the genesis block and reinitialise the chain
if err := core.WriteTd(bc.chainDb, genesis.Hash(), genesis.NumberU64(), genesis.Difficulty()); err
!= nil {
log.Crit("Failed to write genesis block TD", "err", err)
}
if err := core.WriteBlock(bc.chainDb, genesis); err != nil {
log.Crit("Failed to write genesis block", "err", err)
}
bc.genesisBlock = genesis
bc.hc.SetGenesis(bc.genesisBlock.Header())
bc.hc.SetCurrentHeader(bc.genesisBlock.Header())
}

// Accessors

// Engine retrieves the light chain's consensus engine.
func (bc *LightChain) Engine() consensus.Engine { return bc.engine }

// Genesis returns the genesis block
func (bc *LightChain) Genesis() *types.Block {
return bc.genesisBlock
}

// GetBody retrieves a block body (transactions and uncles) from the database
// or ODR service by hash, caching it if found.
func (self *LightChain) GetBody(ctx context.Context, hash common.Hash) (*types.Body, error) {
```

```
// Short circuit if the body's already in the cache, retrieve otherwise
if cached, ok := self.bodyCache.Get(hash); ok {
body := cached.(*types.Body)
return body, nil
}
body, err := GetBody(ctx, self.odr, hash, self.hc.GetBlockNumber(hash))
if err != nil {
return nil, err
}
// Cache the found body for next time and return
self.bodyCache.Add(hash, body)
return body, nil
}

// GetBodyRLP retrieves a block body in RLP encoding from the database or
// ODR service by hash, caching it if found.
func (self *LightChain) GetBodyRLP(ctx context.Context, hash common.Hash) (rlp.RawValue,
error) {
// Short circuit if the body's already in the cache, retrieve otherwise
if cached, ok := self.bodyRLPCache.Get(hash); ok {
return cached.(rlp.RawValue), nil
}
body, err := GetBodyRLP(ctx, self.odr, hash, self.hc.GetBlockNumber(hash))
if err != nil {
return nil, err
}
// Cache the found body for next time and return
self.bodyRLPCache.Add(hash, body)
return body, nil
}

// HasBlock checks if a block is fully present in the database or not, caching
// it if present.
func (bc *LightChain) HasBlock(hash common.Hash) bool {
blk, _ := bc.GetBlockByHash(NoOdr, hash)
return blk != nil
}

// GetBlock retrieves a block from the database or ODR service by hash and number,
// caching it if found.
func (self *LightChain) GetBlock(ctx context.Context, hash common.Hash, number uint64)
(*types.Block, error) {
```

```
// Short circuit if the block's already in the cache, retrieve otherwise
if block, ok := self.blockCache.Get(hash); ok {
return block.(*types.Block), nil
}
block, err := GetBlock(ctx, self.odr, hash, number)
if err != nil {
return nil, err
}
// Cache the found block for next time and return
self.blockCache.Add(block.Hash(), block)
return block, nil
}

// GetBlockByHash retrieves a block from the database or ODR service by hash,
// caching it if found.
func (self *LightChain) GetBlockByHash(ctx context.Context, hash common.Hash) (*types.Block,
error) {
return self.GetBlock(ctx, hash, self.hc.GetBlockNumber(hash))
}

// GetBlockByNumber retrieves a block from the database or ODR service by
// number, caching it (associated with its hash) if found.
func (self *LightChain) GetBlockByNumber(ctx context.Context, number uint64) (*types.Block,
error) {
hash, err := GetCanonicalHash(ctx, self.odr, number)
if hash == (common.Hash{}) || err != nil {
return nil, err
}
return self.GetBlock(ctx, hash, number)
}

// Stop stops the blockchain service. If any imports are currently in progress
// it will abort them using the procInterrupt.
func (bc *LightChain) Stop() {
if !atomic.CompareAndSwapInt32(&bc.running, 0, 1) {
return
}
close(bc.quit)
atomic.StoreInt32(&bc.procInterrupt, 1)

bc.wg.Wait()
log.Info("Blockchain manager stopped")
```

```
}

// Rollback is designed to remove a chain of links from the database that aren't
// certain enough to be valid.
func (self *LightChain) Rollback(chain []common.Hash) {
self.mu.Lock()
defer self.mu.Unlock()

for i := len(chain) - 1; i >= 0; i-- {
hash := chain[i]

if head := self.hc.CurrentHeader(); head.Hash() == hash {
self.hc.SetCurrentHeader(self.GetHeader(head.ParentHash, head.Number.Uint64()-1))
}
}
}

// postChainEvents iterates over the events generated by a chain insertion and
// posts them into the event mux.
func (self *LightChain) postChainEvents(events []interface{}) {
for _, event := range events {
if event, ok := event.(core.ChainEvent); ok {
if self.LastBlockHash() == event.Hash {
self.eventMux.Post(core.ChainHeadEvent{Block: event.Block})
}
}
// Fire the insertion events individually too
self.eventMux.Post(event)
}
}

// InsertHeaderChain attempts to insert the given header chain in to the local
// chain, possibly creating a reorg. If an error is returned, it will return the
// index number of the failing header as well an error describing what went wrong.
//
// The verify parameter can be used to fine tune whether nonce verification
// should be done or not. The reason behind the optional check is because some
// of the header retrieval mechanisms already need to verfy nonces, as well as
// because nonces can be verified sparsely, not needing to check each.
//
// In the case of a light chain, InsertHeaderChain also creates and posts light
// chain events when necessary.
```

```
func (self *LightChain) InsertHeaderChain(chain []*types.Header, checkFreq int) (int, error) {
start := time.Now()
if i, err := self.hc.ValidateHeaderChain(chain, checkFreq); err != nil {
return i, err
}

// Make sure only one thread manipulates the chain at once
self.chainmu.Lock()
defer func() {
self.chainmu.Unlock()
time.Sleep(time.Millisecond * 10) // ugly hack; do not hog chain lock in case syncing is CPU-limited by validation
}()

self.wg.Add(1)
defer self.wg.Done()

var events []interface{}
whFunc := func(header *types.Header) error {
self.mu.Lock()
defer self.mu.Unlock()

status, err := self.hc.WriteHeader(header)

switch status {
case core.CanonStatTy:
log.Debug("Inserted new header", "number", header.Number, "hash", header.Hash())
events = append(events, core.ChainEvent{Block: types.NewBlockWithHeader(header), Hash: header.Hash()})

case core.SideStatTy:
log.Debug("Inserted forked header", "number", header.Number, "hash", header.Hash())
events = append(events, core.ChainSideEvent{Block: types.NewBlockWithHeader(header)})
}
return err
}
i, err := self.hc.InsertHeaderChain(chain, whFunc, start)
go self.postChainEvents(events)
return i, err
}

// CurrentHeader retrieves the current head header of the canonical chain. The
```

```
// header is retrieved from the HeaderChain's internal cache.
func (self *LightChain) CurrentHeader() *types.Header {
self.mu.RLock()
defer self.mu.RUnlock()

return self.hc.CurrentHeader()
}

// GetTd retrieves a block's total difficulty in the canonical chain from the
// database by hash and number, caching it if found.
func (self *LightChain) GetTd(hash common.Hash, number uint64) *big.Int {
return self.hc.GetTd(hash, number)
}

// GetTdByHash retrieves a block's total difficulty in the canonical chain from the
// database by hash, caching it if found.
func (self *LightChain) GetTdByHash(hash common.Hash) *big.Int {
return self.hc.GetTdByHash(hash)
}

// GetHeader retrieves a block header from the database by hash and number,
// caching it if found.
func (self *LightChain) GetHeader(hash common.Hash, number uint64) *types.Header {
return self.hc.GetHeader(hash, number)
}

// GetHeaderByHash retrieves a block header from the database by hash, caching it if
// found.
func (self *LightChain) GetHeaderByHash(hash common.Hash) *types.Header {
return self.hc.GetHeaderByHash(hash)
}

// HasHeader checks if a block header is present in the database or not, caching
// it if present.
func (bc *LightChain) HasHeader(hash common.Hash) bool {
return bc.hc.HasHeader(hash)
}

// GetBlockHashesFromHash retrieves a number of block hashes starting at a given
// hash, fetching towards the genesis block.
func (self *LightChain) GetBlockHashesFromHash(hash common.Hash, max uint64)
[]common.Hash {
```

```
return self.hc.GetBlockHashesFromHash(hash, max)
}

// GetHeaderByNumber retrieves a block header from the database by number,
// caching it (associated with its hash) if found.
func (self *LightChain) GetHeaderByNumber(number uint64) *types.Header {
return self.hc.GetHeaderByNumber(number)
}

// GetHeaderByNumberOdr retrieves a block header from the database or network
// by number, caching it (associated with its hash) if found.
func (self *LightChain) GetHeaderByNumberOdr(ctx context.Context, number uint64)
(*types.Header, error) {
if header := self.hc.GetHeaderByNumber(number); header != nil {
return header, nil
}
return GetHeaderByNumber(ctx, self.odr, number)
}

func (self *LightChain) SyncCht(ctx context.Context) bool {
headNum := self.CurrentHeader().Number.Uint64()
cht := GetTrustedCht(self.chainDb)
if headNum+1 < cht.Number*ChtFrequency {
num := cht.Number*ChtFrequency - 1
header, err := GetHeaderByNumber(ctx, self.odr, num)
if header != nil && err == nil {
self.mu.Lock()
if self.hc.CurrentHeader().Number.Uint64() < header.Number.Uint64() {
self.hc.SetCurrentHeader(header)
}
self.mu.Unlock()
return true
}
}
return false
}

// LockChain locks the chain mutex for reading so that multiple canonical hashes can be
// retrieved while it is guaranteed that they belong to the same version of the chain
func (self *LightChain) LockChain() {
self.chainmu.RLock()
}
```

```
// UnlockChain unlocks the chain mutex
func (self *LightChain) UnlockChain() {
self.chainmu.RUnlock()
}

11:F:\git\coin\ethereum\go-ethereum\light\lightchain_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package light

import (
"context"
"fmt"
"math/big"
"testing"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/consensus/ethash"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/params"
)

// So we can deterministically seed different blockchains
var (
canonicalSeed = 1
forkSeed      = 2
)

// makeHeaderChain creates a deterministic chain of headers rooted at parent.
func makeHeaderChain(parent *types.Header, n int, db ethdb.Database, seed int) []*types.Header
{
blocks, _ := core.GenerateChain(params.TestChainConfig, types.NewBlockWithHeader(parent),
db, n, func(i int, b *core.BlockGen) {
b.SetCoinbase(common.Address{0: byte(seed), 19: byte(i)})
})
headers := make([]*types.Header, len(blocks))
for i, block := range blocks {
headers[i] = block.Header()
```

```
}
return headers
}

// newCanonical creates a chain database, and injects a deterministic canonical
// chain. Depending on the full flag, if creates either a full block chain or a
// header only chain.
func newCanonical(n int) (ethdb.Database, *LightChain, error) {
db, _ := ethdb.NewMemDatabase()
gspec := core.Genesis{Config: params.TestChainConfig}
genesis := gspec.MustCommit(db)
blockchain, _ := NewLightChain(&dummyOdr{db: db}, gspec.Config, ethash.NewFaker(),
new(event.TypeMux))

// Create and inject the requested chain
if n == 0 {
return db, blockchain, nil
}
// Header-only chain requested
headers := makeHeaderChain(genesis.Header(), n, db, canonicalSeed)
_, err := blockchain.InsertHeaderChain(headers, 1)
return db, blockchain, err
}

// newTestLightChain creates a LightChain that doesn't validate anything.
func newTestLightChain() *LightChain {
db, _ := ethdb.NewMemDatabase()
gspec := &core.Genesis{
Difficulty: big.NewInt(1),
Config:     params.TestChainConfig,
}
gspec.MustCommit(db)
lc, err := NewLightChain(&dummyOdr{db: db}, gspec.Config, ethash.NewFullFaker(),
new(event.TypeMux))
if err != nil {
panic(err)
}
return lc
}

// Test fork of length N starting from block i
func testFork(t *testing.T, LightChain *LightChain, i, n int, comparator func(td1, td2 *big.Int)) {
```

```
// Copy old chain up to #i into a new db
db, LightChain2, err := newCanonical(i)
if err != nil {
t.Fatal("could not make new canonical in testFork", err)
}
// Assert the chains have the same header/block at #i
var hash1, hash2 common.Hash
hash1 = LightChain.GetHeaderByNumber(uint64(i)).Hash()
hash2 = LightChain2.GetHeaderByNumber(uint64(i)).Hash()
if hash1 != hash2 {
t.Errorf("chain content mismatch at %d: have hash %v, want hash %v", i, hash2, hash1)
}
// Extend the newly created chain
var (
headerChainB []*types.Header
)
headerChainB = makeHeaderChain(LightChain2.CurrentHeader(), n, db, forkSeed)
if _, err := LightChain2.InsertHeaderChain(headerChainB, 1); err != nil {
t.Fatalf("failed to insert forking chain: %v", err)
}
// Sanity check that the forked chain can be imported into the original
var tdPre, tdPost *big.Int

tdPre = LightChain.GetTdByHash(LightChain.CurrentHeader().Hash())
if err := testHeaderChainImport(headerChainB, LightChain); err != nil {
t.Fatalf("failed to import forked header chain: %v", err)
}
tdPost = LightChain.GetTdByHash(headerChainB[len(headerChainB)-1].Hash())
// Compare the total difficulties of the chains
comparator(tdPre, tdPost)
}

func printChain(bc *LightChain) {
for i := bc.CurrentHeader().Number.Uint64(); i > 0; i-- {
b := bc.GetHeaderByNumber(uint64(i))
fmt.Printf("\t%x %v\n", b.Hash(), b.Difficulty)
}
}

// testHeaderChainImport tries to process a chain of header, writing them into
// the database if successful.
func testHeaderChainImport(chain []*types.Header, lightchain *LightChain) error {
```

```
for _, header := range chain {
// Try and validate the header
if err := lightchain.engine.VerifyHeader(lightchain.hc, header, true); err != nil {
return err
}
// Manually insert the header into the database, but don't reorganize (allows subsequent testing)
lightchain.mu.Lock()
core.WriteTd(lightchain.chainDb, header.Hash(), header.Number.Uint64(),
new(big.Int).Add(header.Difficulty, lightchain.GetTdByHash(header.ParentHash)))
core.WriteHeader(lightchain.chainDb, header)
lightchain.mu.Unlock()
}
return nil
}

// Tests that given a starting canonical chain of a given size, it can be extended
// with various length chains.
func TestExtendCanonicalHeaders(t *testing.T) {
length := 5

// Make first chain starting from genesis
_, processor, err := newCanonical(length)
if err != nil {
t.Fatalf("failed to make new canonical chain: %v", err)
}
// Define the difficulty comparator
better := func(td1, td2 *big.Int) {
if td2.Cmp(td1) <= 0 {
t.Errorf("total difficulty mismatch: have %v, expected more than %v", td2, td1)
}
}
// Start fork from current height
testFork(t, processor, length, 1, better)
testFork(t, processor, length, 2, better)
testFork(t, processor, length, 5, better)
testFork(t, processor, length, 10, better)
}

// Tests that given a starting canonical chain of a given size, creating shorter
// forks do not take canonical ownership.
func TestShorterForkHeaders(t *testing.T) {
length := 10
```

```
// Make first chain starting from genesis
_, processor, err := newCanonical(length)
if err != nil {
t.Fatalf("failed to make new canonical chain: %v", err)
}
// Define the difficulty comparator
worse := func(td1, td2 *big.Int) {
if td2.Cmp(td1) >= 0 {
t.Errorf("total difficulty mismatch: have %v, expected less than %v", td2, td1)
}
}
// Sum of numbers must be less than `length` for this to be a shorter fork
testFork(t, processor, 0, 3, worse)
testFork(t, processor, 0, 7, worse)
testFork(t, processor, 1, 1, worse)
testFork(t, processor, 1, 7, worse)
testFork(t, processor, 5, 3, worse)
testFork(t, processor, 5, 4, worse)
}

// Tests that given a starting canonical chain of a given size, creating longer
// forks do take canonical ownership.
func TestLongerForkHeaders(t *testing.T) {
length := 10

// Make first chain starting from genesis
_, processor, err := newCanonical(length)
if err != nil {
t.Fatalf("failed to make new canonical chain: %v", err)
}
// Define the difficulty comparator
better := func(td1, td2 *big.Int) {
if td2.Cmp(td1) <= 0 {
t.Errorf("total difficulty mismatch: have %v, expected more than %v", td2, td1)
}
}
// Sum of numbers must be greater than `length` for this to be a longer fork
testFork(t, processor, 0, 11, better)
testFork(t, processor, 0, 15, better)
testFork(t, processor, 1, 10, better)
testFork(t, processor, 1, 12, better)
```

```
testFork(t, processor, 5, 6, better)
testFork(t, processor, 5, 8, better)
}

// Tests that given a starting canonical chain of a given size, creating equal
// forks do take canonical ownership.
func TestEqualForkHeaders(t *testing.T) {
length := 10

// Make first chain starting from genesis
_, processor, err := newCanonical(length)
if err != nil {
t.Fatalf("failed to make new canonical chain: %v", err)
}
// Define the difficulty comparator
equal := func(td1, td2 *big.Int) {
if td2.Cmp(td1) != 0 {
t.Errorf("total difficulty mismatch: have %v, want %v", td2, td1)
}
}
// Sum of numbers must be equal to `length` for this to be an equal fork
testFork(t, processor, 0, 10, equal)
testFork(t, processor, 1, 9, equal)
testFork(t, processor, 2, 8, equal)
testFork(t, processor, 5, 5, equal)
testFork(t, processor, 6, 4, equal)
testFork(t, processor, 9, 1, equal)
}

// Tests that chains missing links do not get accepted by the processor.
func TestBrokenHeaderChain(t *testing.T) {
// Make chain starting from genesis
db, LightChain, err := newCanonical(10)
if err != nil {
t.Fatalf("failed to make new canonical chain: %v", err)
}
// Create a forked chain, and try to insert with a missing link
chain := makeHeaderChain(LightChain.CurrentHeader(), 5, db, forkSeed)[1:]
if err := testHeaderChainImport(chain, LightChain); err == nil {
t.Errorf("broken header chain not reported")
}
}
```

```
func makeHeaderChainWithDiff(genesis *types.Block, d []int, seed byte) []*types.Header {
var chain []*types.Header
for i, difficulty := range d {
header := &types.Header{
Coinbase:    common.Address{seed},
Number:      big.NewInt(int64(i + 1)),
Difficulty:  big.NewInt(int64(difficulty)),
UncleHash:   types.EmptyUncleHash,
TxHash:      types.EmptyRootHash,
ReceiptHash: types.EmptyRootHash,
}
if i == 0 {
header.ParentHash = genesis.Hash()
} else {
header.ParentHash = chain[i-1].Hash()
}
chain = append(chain, types.CopyHeader(header))
}
return chain
}

type dummyOdr struct {
OdrBackend
db ethdb.Database
}

func (odr *dummyOdr) Database() ethdb.Database {
return odr.db
}

func (odr *dummyOdr) Retrieve(ctx context.Context, req OdrRequest) error {
return nil
}

// Tests that reorganizing a long difficult chain after a short easy one
// overwrites the canonical numbers and links in the database.
func TestReorgLongHeaders(t *testing.T) {
testReorg(t, []int{1, 2, 4}, []int{1, 2, 3, 4}, 10)
}

// Tests that reorganizing a short difficult chain after a long easy one
```

```
// overwrites the canonical numbers and links in the database.
func TestReorgShortHeaders(t *testing.T) {
testReorg(t, []int{1, 2, 3, 4}, []int{1, 10}, 11)
}

func testReorg(t *testing.T, first, second []int, td int64) {
bc := newTestLightChain()

// Insert an easy and a difficult chain afterwards
bc.InsertHeaderChain(makeHeaderChainWithDiff(bc.genesisBlock, first, 11), 1)
bc.InsertHeaderChain(makeHeaderChainWithDiff(bc.genesisBlock, second, 22), 1)
// Check that the chain is valid number and link wise
prev := bc.CurrentHeader()
for header := bc.GetHeaderByNumber(bc.CurrentHeader().Number.Uint64() - 1);
header.Number.Uint64() != 0; prev, header = header,
bc.GetHeaderByNumber(header.Number.Uint64()-1) {
if prev.ParentHash != header.Hash() {
t.Errorf("parent header hash mismatch: have %x, want %x", prev.ParentHash, header.Hash())
}
}
// Make sure the chain total difficulty is the correct one
want := new(big.Int).Add(bc.genesisBlock.Difficulty(), big.NewInt(td))
if have := bc.GetTdByHash(bc.CurrentHeader().Hash()); have.Cmp(want) != 0 {
t.Errorf("total difficulty mismatch: have %v, want %v", have, want)
}
}

// Tests that the insertion functions detect banned hashes.
func TestBadHeaderHashes(t *testing.T) {
bc := newTestLightChain()

// Create a chain, ban a hash and try to import
var err error
headers := makeHeaderChainWithDiff(bc.genesisBlock, []int{1, 2, 4}, 10)
core.BadHashes[headers[2].Hash()] = true
if _, err = bc.InsertHeaderChain(headers, 1); err != core.ErrBlacklistedHash {
t.Errorf("error mismatch: have: %v, want %v", err, core.ErrBlacklistedHash)
}
}

// Tests that bad hashes are detected on boot, and the chan rolled back to a
// good state prior to the bad hash.
```

```
func TestReorgBadHeaderHashes(t *testing.T) {
bc := newTestLightChain()

// Create a chain, import and ban aferwards
headers := makeHeaderChainWithDiff(bc.genesisBlock, []int{1, 2, 3, 4}, 10)

if _, err := bc.InsertHeaderChain(headers, 1); err != nil {
t.Fatalf("failed to import headers: %v", err)
}
if bc.CurrentHeader().Hash() != headers[3].Hash() {
t.Errorf("last header hash mismatch: have: %x, want %x", bc.CurrentHeader().Hash(),
headers[3].Hash())
}
core.BadHashes[headers[3].Hash()] = true
defer func() { delete(core.BadHashes, headers[3].Hash()) }()

// Create a new LightChain and check that it rolled back the state.
ncm, err := NewLightChain(&dummyOdr{db: bc.chainDb}, params.TestChainConfig,
ethash.NewFaker(), new(event.TypeMux))
if err != nil {
t.Fatalf("failed to create new chain manager: %v", err)
}
if ncm.CurrentHeader().Hash() != headers[2].Hash() {
t.Errorf("last header hash mismatch: have: %x, want %x", ncm.CurrentHeader().Hash(),
headers[2].Hash())
}
}
```

12:F:\git\coin\ethereum\go-ethereum\light\odr.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package light implements on-demand retrieval capable state and chain objects
// for the Ethereum Light Client.
package light

import (
"context"
"math/big"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/types"
```

```
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/rlp"
)

// NoOdr is the default context passed to an ODR capable function when the ODR
// service is not required.
var NoOdr = context.Background()

// OdrBackend is an interface to a backend service that handles ODR retrievals type
type OdrBackend interface {
Database() ethdb.Database
Retrieve(ctx context.Context, req OdrRequest) error
}

// OdrRequest is an interface for retrieval requests
type OdrRequest interface {
StoreResult(db ethdb.Database)
}

// TrieID identifies a state or account storage trie
type TrieID struct {
BlockHash, Root common.Hash
BlockNumber     uint64
AccKey          []byte
}

// StateTrieID returns a TrieID for a state trie belonging to a certain block
// header.
func StateTrieID(header *types.Header) *TrieID {
return &TrieID{
BlockHash:   header.Hash(),
BlockNumber: header.Number.Uint64(),
AccKey:      nil,
Root:        header.Root,
}
}

// StorageTrieID returns a TrieID for a contract storage trie at a given account
// of a given state trie. It also requires the root hash of the trie for
// checking Merkle proofs.
func StorageTrieID(state *TrieID, addrHash, root common.Hash) *TrieID {
```

```
return &TrieID{
BlockHash:   state.BlockHash,
BlockNumber: state.BlockNumber,
AccKey:      addrHash[:],
Root:        root,
}
}

// TrieRequest is the ODR request type for state/storage trie entries
type TrieRequest struct {
OdrRequest
Id    *TrieID
Key   []byte
Proof []rlp.RawValue
}

// StoreResult stores the retrieved data in local database
func (req *TrieRequest) StoreResult(db ethdb.Database) {
storeProof(db, req.Proof)
}

// storeProof stores the new trie nodes obtained from a merkle proof in the database
func storeProof(db ethdb.Database, proof []rlp.RawValue) {
for _, buf := range proof {
hash := crypto.Keccak256(buf)
val, _ := db.Get(hash)
if val == nil {
db.Put(hash, buf)
}
}
}

// CodeRequest is the ODR request type for retrieving contract code
type CodeRequest struct {
OdrRequest
Id   *TrieID // references storage trie of the account
Hash common.Hash
Data []byte
}

// StoreResult stores the retrieved data in local database
func (req *CodeRequest) StoreResult(db ethdb.Database) {
```

```
db.Put(req.Hash[:], req.Data)
}

// BlockRequest is the ODR request type for retrieving block bodies
type BlockRequest struct {
OdrRequest
Hash   common.Hash
Number uint64
Rlp    []byte
}

// StoreResult stores the retrieved data in local database
func (req *BlockRequest) StoreResult(db ethdb.Database) {
core.WriteBodyRLP(db, req.Hash, req.Number, req.Rlp)
}

// ReceiptsRequest is the ODR request type for retrieving block bodies
type ReceiptsRequest struct {
OdrRequest
Hash     common.Hash
Number   uint64
Receipts types.Receipts
}

// StoreResult stores the retrieved data in local database
func (req *ReceiptsRequest) StoreResult(db ethdb.Database) {
core.WriteBlockReceipts(db, req.Hash, req.Number, req.Receipts)
}

// TrieRequest is the ODR request type for state/storage trie entries
type ChtRequest struct {
OdrRequest
ChtNum, BlockNum uint64
ChtRoot          common.Hash
Header           *types.Header
Td               *big.Int
Proof            []rlp.RawValue
}

// StoreResult stores the retrieved data in local database
func (req *ChtRequest) StoreResult(db ethdb.Database) {
// if there is a canonical hash, there is a header too
```

```
core.WriteHeader(db, req.Header)
hash, num := req.Header.Hash(), req.Header.Number.Uint64()
core.WriteTd(db, hash, num, req.Td)
core.WriteCanonicalHash(db, hash, num)
//storeProof(db, req.Proof)
}

13:F:\git\coin\ethereum\go-ethereum\light\odr_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package light

import (
"bytes"
"context"
"errors"
"math/big"
"testing"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/math"
"github.com/ethereum/go-ethereum/consensus/ethash"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/core/vm"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/params"
"github.com/ethereum/go-ethereum/rlp"
"github.com/ethereum/go-ethereum/trie"
)

var (
testBankKey, _  =
crypto.HexToECDSA("b71c71a67e1177ad4e901695e1b4b9ee17ae16c6668d313eac2f96dbcda3f
291")
testBankAddress = crypto.PubkeyToAddress(testBankKey.PublicKey)
testBankFunds   = big.NewInt(100000000)
```

```
acc1Key, _ =
crypto.HexToECDSA("8a1f9a8f95be41cd7ccb6168179afb4504aefe388d1e14474d32c45c72ce7b
7a")
acc2Key, _ =
crypto.HexToECDSA("49a7b37aa6f6645917e7b807e9d1c00d4fa71f18343b0d4122a4d2df64dd6f
ee")
acc1Addr   = crypto.PubkeyToAddress(acc1Key.PublicKey)
acc2Addr   = crypto.PubkeyToAddress(acc2Key.PublicKey)

testContractCode =
common.Hex2Bytes("606060405260cc806010600396000f360606040526000357c01000000000
000000000000000000000000000000000000000000000000000009004806360cd2685146041578063c16
431b914606b57603f565b005b6055600480803590602001909190505060a9565b6040518082815
260200191505060405180910390f35b6088600480803590602001909190803590602001909190505
050608a565b005b806000600050836064811015600257909001600005b50819055505b5050565b6
000600060005082606481101560025790900160005b5054905060c7565b91905056")
testContractAddr common.Address

bigTxGas = new(big.Int).SetUint64(params.TxGas)
)

type testOdr struct {
OdrBackend
sdb, ldb ethdb.Database
disable  bool
}

func (odr *testOdr) Database() ethdb.Database {
return odr.ldb
}

var ErrOdrDisabled = errors.New("ODR disabled")

func (odr *testOdr) Retrieve(ctx context.Context, req OdrRequest) error {
if odr.disable {
return ErrOdrDisabled
}
switch req := req.(type) {
case *BlockRequest:
req.Rlp = core.GetBodyRLP(odr.sdb, req.Hash, core.GetBlockNumber(odr.sdb, req.Hash))
case *ReceiptsRequest:
req.Receipts = core.GetBlockReceipts(odr.sdb, req.Hash, core.GetBlockNumber(odr.sdb,
```

```
req.Hash))
case *TrieRequest:
t, _ := trie.New(req.Id.Root, odr.sdb)
req.Proof = t.Prove(req.Key)
case *CodeRequest:
req.Data, _ = odr.sdb.Get(req.Hash[:])
}
req.StoreResult(odr.ldb)
return nil
}

type odrTestFn func(ctx context.Context, db ethdb.Database, bc *core.BlockChain, lc *LightChain,
bhash common.Hash) ([]byte, error)

func TestOdrGetBlockLes1(t *testing.T) { testChainOdr(t, 1, odrGetBlock) }

func odrGetBlock(ctx context.Context, db ethdb.Database, bc *core.BlockChain, lc *LightChain,
bhash common.Hash) ([]byte, error) {
var block *types.Block
if bc != nil {
block = bc.GetBlockByHash(bhash)
} else {
block, _ = lc.GetBlockByHash(ctx, bhash)
}
if block == nil {
return nil, nil
}
rlp, _ := rlp.EncodeToBytes(block)
return rlp, nil
}

func TestOdrGetReceiptsLes1(t *testing.T) { testChainOdr(t, 1, odrGetReceipts) }

func odrGetReceipts(ctx context.Context, db ethdb.Database, bc *core.BlockChain, lc *LightChain,
bhash common.Hash) ([]byte, error) {
var receipts types.Receipts
if bc != nil {
receipts = core.GetBlockReceipts(db, bhash, core.GetBlockNumber(db, bhash))
} else {
receipts, _ = GetBlockReceipts(ctx, lc.Odr(), bhash, core.GetBlockNumber(db, bhash))
}
if receipts == nil {
```

```
return nil, nil
}
rlp, _ := rlp.EncodeToBytes(receipts)
return rlp, nil
}

func TestOdrAccountsLes1(t *testing.T) { testChainOdr(t, 1, odrAccounts) }

func odrAccounts(ctx context.Context, db ethdb.Database, bc *core.BlockChain, lc *LightChain,
bhash common.Hash) ([]byte, error) {
dummyAddr := common.HexToAddress("1234567812345678123456781234567812345678")
acc := []common.Address{testBankAddress, acc1Addr, acc2Addr, dummyAddr}

var st *state.StateDB
if bc == nil {
header := lc.GetHeaderByHash(bhash)
st = NewState(ctx, header, lc.Odr())
} else {
header := bc.GetHeaderByHash(bhash)
st, _ = state.New(header.Root, state.NewDatabase(db))
}

var res []byte
for _, addr := range acc {
bal := st.GetBalance(addr)
rlp, _ := rlp.EncodeToBytes(bal)
res = append(res, rlp...)
}
return res, st.Error()
}

func TestOdrContractCallLes1(t *testing.T) { testChainOdr(t, 1, odrContractCall) }

type callmsg struct {
types.Message
}

func (callmsg) CheckNonce() bool { return false }

func odrContractCall(ctx context.Context, db ethdb.Database, bc *core.BlockChain, lc *LightChain,
bhash common.Hash) ([]byte, error) {
data :=
```

```
common.Hex2Bytes("60CD26850000000000000000000000000000000000000000000000000000000000000000")
config := params.TestChainConfig

var res []byte
for i := 0; i < 3; i++ {
data[35] = byte(i)

var (
st     *state.StateDB
header *types.Header
chain  core.ChainContext
)
if bc == nil {
chain = lc
header = lc.GetHeaderByHash(bhash)
st = NewState(ctx, header, lc.Odr())
} else {
chain = bc
header = bc.GetHeaderByHash(bhash)
st, _ = state.New(header.Root, state.NewDatabase(db))
}

// Perform read-only call.
st.SetBalance(testBankAddress, math.MaxBig256)
msg := callmsg{types.NewMessage(testBankAddress, &testContractAddr, 0, new(big.Int),
big.NewInt(1000000), new(big.Int), data, false)}
context := core.NewEVMContext(msg, header, chain, nil)
vmenv := vm.NewEVM(context, st, config, vm.Config{})
gp := new(core.GasPool).AddGas(math.MaxBig256)
ret, _, _ := core.ApplyMessage(vmenv, msg, gp)
res = append(res, ret...)
if st.Error() != nil {
return res, st.Error()
}
}
return res, nil
}

func testChainGen(i int, block *core.BlockGen) {
signer := types.HomesteadSigner{}
switch i {
```

```
case 0:
// In block 1, the test bank sends account #1 some ether.
tx, _ := types.SignTx(types.NewTransaction(block.TxNonce(testBankAddress), acc1Addr,
big.NewInt(10000), bigTxGas, nil, nil), signer, testBankKey)
block.AddTx(tx)
case 1:
// In block 2, the test bank sends some more ether to account #1.
// acc1Addr passes it on to account #2.
// acc1Addr creates a test contract.
tx1, _ := types.SignTx(types.NewTransaction(block.TxNonce(testBankAddress), acc1Addr,
big.NewInt(1000), bigTxGas, nil, nil), signer, testBankKey)
nonce := block.TxNonce(acc1Addr)
tx2, _ := types.SignTx(types.NewTransaction(nonce, acc2Addr, big.NewInt(1000), bigTxGas, nil,
nil), signer, acc1Key)
nonce++
tx3, _ := types.SignTx(types.NewContractCreation(nonce, big.NewInt(0), big.NewInt(1000000),
big.NewInt(0), testContractCode), signer, acc1Key)
testContractAddr = crypto.CreateAddress(acc1Addr, nonce)
block.AddTx(tx1)
block.AddTx(tx2)
block.AddTx(tx3)
case 2:
// Block 3 is empty but was mined by account #2.
block.SetCoinbase(acc2Addr)
block.SetExtra([]byte("yeehaw"))
data :=
common.Hex2Bytes("C16431B900000000000000000000000000000000000000000000000000000000000000010000000000000000000000000000000000000000000000000000000000000001")
tx, _ := types.SignTx(types.NewTransaction(block.TxNonce(testBankAddress), testContractAddr,
big.NewInt(0), big.NewInt(100000), nil, data), signer, testBankKey)
block.AddTx(tx)
case 3:
// Block 4 includes blocks 2 and 3 as uncle headers (with modified extra data).
b2 := block.PrevBlock(1).Header()
b2.Extra = []byte("foo")
block.AddUncle(b2)
b3 := block.PrevBlock(2).Header()
b3.Extra = []byte("foo")
block.AddUncle(b3)
data :=
common.Hex2Bytes("C16431B900000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000000000000000000000002")
```

```
tx, _ := types.SignTx(types.NewTransaction(block.TxNonce(testBankAddress), testContractAddr,
big.NewInt(0), big.NewInt(100000), nil, data), signer, testBankKey)
block.AddTx(tx)
}
}

func testChainOdr(t *testing.T, protocol int, fn odrTestFn) {
var (
evmux   = new(event.TypeMux)
sdb, _  = ethdb.NewMemDatabase()
ldb, _  = ethdb.NewMemDatabase()
gspec   = core.Genesis{Alloc: core.GenesisAlloc{testBankAddress: {Balance: testBankFunds}}}
genesis = gspec.MustCommit(sdb)
)
gspec.MustCommit(ldb)
// Assemble the test environment
blockchain, _ := core.NewBlockChain(sdb, params.TestChainConfig, ethash.NewFullFaker(),
evmux, vm.Config{})
gchain, _ := core.GenerateChain(params.TestChainConfig, genesis, sdb, 4, testChainGen)
if _, err := blockchain.InsertChain(gchain); err != nil {
t.Fatal(err)
}

odr := &testOdr{sdb: sdb, ldb: ldb}
lightchain, err := NewLightChain(odr, params.TestChainConfig, ethash.NewFullFaker(), evmux)
if err != nil {
t.Fatal(err)
}
headers := make([]*types.Header, len(gchain))
for i, block := range gchain {
headers[i] = block.Header()
}
if _, err := lightchain.InsertHeaderChain(headers, 1); err != nil {
t.Fatal(err)
}

test := func(expFail int) {
for i := uint64(0); i <= blockchain.CurrentHeader().Number.Uint64(); i++ {
bhash := core.GetCanonicalHash(sdb, i)
b1, err := fn(NoOdr, sdb, blockchain, nil, bhash)
if err != nil {
t.Fatalf("error in full-node test for block %d: %v", i, err)
```

```
}

ctx, cancel := context.WithTimeout(context.Background(), 200*time.Millisecond)
defer cancel()

exp := i < uint64(expFail)
b2, err := fn(ctx, ldb, nil, lightchain, bhash)
if err != nil && exp {
t.Errorf("error in ODR test for block %d: %v", i, err)
}

eq := bytes.Equal(b1, b2)
if exp && !eq {
t.Errorf("ODR test output for block %d doesn't match full node", i)
}
}
}

// expect retrievals to fail (except genesis block) without a les peer
t.Log("checking without ODR")
odr.disable = true
test(1)

// expect all retrievals to pass with ODR enabled
t.Log("checking with ODR")
odr.disable = false
test(len(gchain))

// still expect all retrievals to pass, now data should be cached locally
t.Log("checking without ODR, should be cached")
odr.disable = true
test(len(gchain))
}

14:F:\git\coin\ethereum\go-ethereum\light\odr_util.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package light

import (
"bytes"
"context"
```

```
"errors"
"math/big"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/rlp"
)

var sha3_nil = crypto.Keccak256Hash(nil)

var (
ErrNoTrustedCht = errors.New("No trusted canonical hash trie")
ErrNoHeader     = errors.New("Header not found")

ChtFrequency     = uint64(4096)
ChtConfirmations = uint64(2048)
trustedChtKey    = []byte("TrustedCHT")
)

type ChtNode struct {
Hash common.Hash
Td   *big.Int
}

type TrustedCht struct {
Number uint64
Root   common.Hash
}

func GetTrustedCht(db ethdb.Database) TrustedCht {
data, _ := db.Get(trustedChtKey)
var res TrustedCht
if err := rlp.DecodeBytes(data, &res); err != nil {
return TrustedCht{0, common.Hash{}}
}
return res
}

func WriteTrustedCht(db ethdb.Database, cht TrustedCht) {
```

```
data, _ := rlp.EncodeToBytes(cht)
db.Put(trustedChtKey, data)
}

func DeleteTrustedCht(db ethdb.Database) {
db.Delete(trustedChtKey)
}

func GetHeaderByNumber(ctx context.Context, odr OdrBackend, number uint64) (*types.Header,
error) {
db := odr.Database()
hash := core.GetCanonicalHash(db, number)
if (hash != common.Hash{}) {
// if there is a canonical hash, there is a header too
header := core.GetHeader(db, hash, number)
if header == nil {
panic("Canonical hash present but header not found")
}
return header, nil
}

cht := GetTrustedCht(db)
if number >= cht.Number*ChtFrequency {
return nil, ErrNoTrustedCht
}

r := &ChtRequest{ChtRoot: cht.Root, ChtNum: cht.Number, BlockNum: number}
if err := odr.Retrieve(ctx, r); err != nil {
return nil, err
} else {
return r.Header, nil
}
}

func GetCanonicalHash(ctx context.Context, odr OdrBackend, number uint64) (common.Hash,
error) {
hash := core.GetCanonicalHash(odr.Database(), number)
if (hash != common.Hash{}) {
return hash, nil
}
header, err := GetHeaderByNumber(ctx, odr, number)
if header != nil {
```

```
return header.Hash(), nil
}
return common.Hash{}, err
}

// GetBodyRLP retrieves the block body (transactions and uncles) in RLP encoding.
func GetBodyRLP(ctx context.Context, odr OdrBackend, hash common.Hash, number uint64)
(rlp.RawValue, error) {
if data := core.GetBodyRLP(odr.Database(), hash, number); data != nil {
return data, nil
}
r := &BlockRequest{Hash: hash, Number: number}
if err := odr.Retrieve(ctx, r); err != nil {
return nil, err
} else {
return r.Rlp, nil
}
}

// GetBody retrieves the block body (transactons, uncles) corresponding to the
// hash.
func GetBody(ctx context.Context, odr OdrBackend, hash common.Hash, number uint64)
(*types.Body, error) {
data, err := GetBodyRLP(ctx, odr, hash, number)
if err != nil {
return nil, err
}
body := new(types.Body)
if err := rlp.Decode(bytes.NewReader(data), body); err != nil {
return nil, err
}
return body, nil
}

// GetBlock retrieves an entire block corresponding to the hash, assembling it
// back from the stored header and body.
func GetBlock(ctx context.Context, odr OdrBackend, hash common.Hash, number uint64)
(*types.Block, error) {
// Retrieve the block header and body contents
header := core.GetHeader(odr.Database(), hash, number)
if header == nil {
return nil, ErrNoHeader
```

```
}
body, err := GetBody(ctx, odr, hash, number)
if err != nil {
return nil, err
}
// Reassemble the block and return
return types.NewBlockWithHeader(header).WithBody(body.Transactions, body.Uncles), nil
}

// GetBlockReceipts retrieves the receipts generated by the transactions included
// in a block given by its hash.
func GetBlockReceipts(ctx context.Context, odr OdrBackend, hash common.Hash, number uint64)
(types.Receipts, error) {
receipts := core.GetBlockReceipts(odr.Database(), hash, number)
if receipts != nil {
return receipts, nil
}
r := &ReceiptsRequest{Hash: hash, Number: number}
if err := odr.Retrieve(ctx, r); err != nil {
return nil, err
}
return r.Receipts, nil
}

15:F:\git\coin\ethereum\go-ethereum\light\trie.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package light

import (
"context"
"fmt"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/trie"
)

func NewState(ctx context.Context, head *types.Header, odr OdrBackend) *state.StateDB {
state, _ := state.New(head.Root, NewStateDatabase(ctx, head, odr))
```

```
return state
}

func NewStateDatabase(ctx context.Context, head *types.Header, odr OdrBackend)
state.Database {
return &odrDatabase{ctx, StateTrieID(head), odr}
}

type odrDatabase struct {
ctx     context.Context
id      *TrieID
backend OdrBackend
}

func (db *odrDatabase) OpenTrie(root common.Hash) (state.Trie, error) {
return &odrTrie{db: db, id: db.id}, nil
}

func (db *odrDatabase) OpenStorageTrie(addrHash, root common.Hash) (state.Trie, error) {
return &odrTrie{db: db, id: StorageTrieID(db.id, addrHash, root)}, nil
}

func (db *odrDatabase) CopyTrie(t state.Trie) state.Trie {
switch t := t.(type) {
case *odrTrie:
cpy := &odrTrie{db: t.db, id: t.id}
if t.trie != nil {
cpytrie := *t.trie
cpy.trie = &cpytrie
}
return cpy
default:
panic(fmt.Errorf("unknown trie type %T", t))
}
}

func (db *odrDatabase) ContractCode(addrHash, codeHash common.Hash) ([]byte, error) {
if codeHash == sha3_nil {
return nil, nil
}
if code, err := db.backend.Database().Get(codeHash[:]); err == nil {
return code, nil
```

```
}
id := *db.id
id.AccKey = addrHash[:]
req := &CodeRequest{Id: &id, Hash: codeHash}
err := db.backend.Retrieve(db.ctx, req)
return req.Data, err
}

func (db *odrDatabase) ContractCodeSize(addrHash, codeHash common.Hash) (int, error) {
code, err := db.ContractCode(addrHash, codeHash)
return len(code), err
}

type odrTrie struct {
db   *odrDatabase
id   *TrieID
trie *trie.Trie
}

func (t *odrTrie) TryGet(key []byte) ([]byte, error) {
key = crypto.Keccak256(key)
var res []byte
err := t.do(key, func() (err error) {
res, err = t.trie.TryGet(key)
return err
})
return res, err
}

func (t *odrTrie) TryUpdate(key, value []byte) error {
key = crypto.Keccak256(key)
return t.do(key, func() error {
return t.trie.TryDelete(key)
})
}

func (t *odrTrie) TryDelete(key []byte) error {
key = crypto.Keccak256(key)
return t.do(key, func() error {
return t.trie.TryDelete(key)
})
}
```

```
func (t *odrTrie) CommitTo(db trie.DatabaseWriter) (common.Hash, error) {
if t.trie == nil {
return t.id.Root, nil
}
return t.trie.CommitTo(db)
}

func (t *odrTrie) Hash() common.Hash {
if t.trie == nil {
return t.id.Root
}
return t.trie.Hash()
}

func (t *odrTrie) NodeIterator(startkey []byte) trie.NodeIterator {
return newNodeIterator(t, startkey)
}

func (t *odrTrie) GetKey(sha []byte) []byte {
return nil
}

// do tries and retries to execute a function until it returns with no error or
// an error type other than MissingNodeError
func (t *odrTrie) do(key []byte, fn func() error) error {
for {
var err error
if t.trie == nil {
t.trie, err = trie.New(t.id.Root, t.db.backend.Database())
}
if err == nil {
err = fn()
}
if _, ok := err.(*trie.MissingNodeError); !ok {
return err
}
r := &TrieRequest{Id: t.id, Key: key}
if err := t.db.backend.Retrieve(t.db.ctx, r); err != nil {
return fmt.Errorf("can't fetch trie key %x: %v", key, err)
}
}
```

```
}

type nodeIterator struct {
trie.NodeIterator
t   *odrTrie
err error
}

func newNodeIterator(t *odrTrie, startkey []byte) trie.NodeIterator {
it := &nodeIterator{t: t}
// Open the actual non-ODR trie if that hasn't happened yet.
if t.trie == nil {
it.do(func() error {
t, err := trie.New(t.id.Root, t.db.backend.Database())
if err == nil {
it.t.trie = t
}
return err
})
}
it.do(func() error {
it.NodeIterator = it.t.trie.NodeIterator(startkey)
return it.NodeIterator.Error()
})
return it
}

func (it *nodeIterator) Next(descend bool) bool {
var ok bool
it.do(func() error {
ok = it.NodeIterator.Next(descend)
return it.NodeIterator.Error()
})
return ok
}

// do runs fn and attempts to fill in missing nodes by retrieving.
func (it *nodeIterator) do(fn func() error) {
var lasthash common.Hash
for {
it.err = fn()
missing, ok := it.err.(*trie.MissingNodeError)
```

```
if !ok {
return
}
if missing.NodeHash == lasthash {
it.err = fmt.Errorf("retrieve loop for trie node %x", missing.NodeHash)
return
}
lasthash = missing.NodeHash
r := &TrieRequest{Id: it.t.id, Key: nibblesToKey(missing.Path)}
if it.err = it.t.db.backend.Retrieve(it.t.db.ctx, r); it.err != nil {
return
}
}
}

func (it *nodeIterator) Error() error {
if it.err != nil {
return it.err
}
return it.NodeIterator.Error()
}

func nibblesToKey(nib []byte) []byte {
if len(nib) > 0 && nib[len(nib)-1] == 0x10 {
nib = nib[:len(nib)-1] // drop terminator
}
if len(nib)&1 == 1 {
nib = append(nib, 0) // make even
}
key := make([]byte, len(nib)/2)
for bi, ni := 0, 0; ni < len(nib); bi, ni = bi+1, ni+2 {
key[bi] = nib[ni]<<4 | nib[ni+1]
}
return key
}

16:F:\git\coin\ethereum\go-ethereum\light\trie_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package light

import (
```

```
"bytes"
"context"
"fmt"
"testing"

"github.com/davecgh/go-spew/spew"
"github.com/ethereum/go-ethereum/consensus/ethash"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/core/vm"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/params"
"github.com/ethereum/go-ethereum/trie"
)

func TestNodeIterator(t *testing.T) {
var (
fulldb, _  = ethdb.NewMemDatabase()
lightdb, _ = ethdb.NewMemDatabase()
gspec      = core.Genesis{Alloc: core.GenesisAlloc{testBankAddress: {Balance: testBankFunds}}}
genesis    = gspec.MustCommit(fulldb)
)
gspec.MustCommit(lightdb)
blockchain, _ := core.NewBlockChain(fulldb, params.TestChainConfig, ethash.NewFullFaker(),
new(event.TypeMux), vm.Config{})
gchain, _ := core.GenerateChain(params.TestChainConfig, genesis, fulldb, 4, testChainGen)
if _, err := blockchain.InsertChain(gchain); err != nil {
panic(err)
}

ctx := context.Background()
odr := &testOdr{sdb: fulldb, ldb: lightdb}
head := blockchain.CurrentHeader()
lightTrie, _ := NewStateDatabase(ctx, head, odr).OpenTrie(head.Root)
fullTrie, _ := state.NewDatabase(fulldb).OpenTrie(head.Root)
if err := diffTries(fullTrie, lightTrie); err != nil {
t.Fatal(err)
}
}

func diffTries(t1, t2 state.Trie) error {
```

```
i1 := trie.NewIterator(t1.NodeIterator(nil))
i2 := trie.NewIterator(t2.NodeIterator(nil))
for i1.Next() && i2.Next() {
if !bytes.Equal(i1.Key, i2.Key) {
spew.Dump(i2)
return fmt.Errorf("tries have different keys %x, %x", i1.Key, i2.Key)
}
if !bytes.Equal(i2.Value, i2.Value) {
return fmt.Errorf("tries differ at key %x", i1.Key)
}
}
switch {
case i1.Err != nil:
return fmt.Errorf("full trie iterator error: %v", i1.Err)
case i2.Err != nil:
return fmt.Errorf("light trie iterator error: %v", i1.Err)
case i1.Next():
return fmt.Errorf("full trie iterator has more k/v pairs")
case i2.Next():
return fmt.Errorf("light trie iterator has more k/v pairs")
}
return nil
}

17:F:\git\coin\ethereum\go-ethereum\light\txpool.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package light

import (
"context"
"fmt"
"sync"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/log"
```

```
"github.com/ethereum/go-ethereum/params"
"github.com/ethereum/go-ethereum/rlp"
)

// txPermanent is the number of mined blocks after a mined transaction is
// considered permanent and no rollback is expected
var txPermanent = uint64(500)

// TxPool implements the transaction pool for light clients, which keeps track
// of the status of locally created transactions, detecting if they are included
// in a block (mined) or rolled back. There are no queued transactions since we
// always receive all locally signed transactions in the same order as they are
// created.
type TxPool struct {
config   *params.ChainConfig
signer   types.Signer
quit     chan bool
eventMux *event.TypeMux
events   *event.TypeMuxSubscription
mu       sync.RWMutex
chain    *LightChain
odr      OdrBackend
chainDb  ethdb.Database
relay    TxRelayBackend
head     common.Hash
nonce    map[common.Address]uint64            // "pending" nonce
pending  map[common.Hash]*types.Transaction   // pending transactions by tx hash
mined    map[common.Hash][]*types.Transaction // mined transactions by block hash
clearIdx uint64                               // earliest block nr that can contain mined tx info

homestead bool
}

// TxRelayBackend provides an interface to the mechanism that forwards transacions
// to the ETH network. The implementations of the functions should be non-blocking.
//
// Send instructs backend to forward new transactions
// NewHead notifies backend about a new head after processed by the tx pool,
//  including  mined and rolled back transactions since the last event
// Discard notifies backend about transactions that should be discarded either
//  because they have been replaced by a re-send or because they have been mined
//  long ago and no rollback is expected
```

```
type TxRelayBackend interface {
Send(txs types.Transactions)
NewHead(head common.Hash, mined []common.Hash, rollback []common.Hash)
Discard(hashes []common.Hash)
}

// NewTxPool creates a new light transaction pool
func NewTxPool(config *params.ChainConfig, eventMux *event.TypeMux, chain *LightChain, relay
TxRelayBackend) *TxPool {
pool := &TxPool{
config:   config,
signer:   types.HomesteadSigner{},
nonce:    make(map[common.Address]uint64),
pending:  make(map[common.Hash]*types.Transaction),
mined:    make(map[common.Hash][]*types.Transaction),
quit:     make(chan bool),
eventMux: eventMux,
events:   eventMux.Subscribe(core.ChainHeadEvent{}),
chain:    chain,
relay:    relay,
odr:      chain.Odr(),
chainDb:  chain.Odr().Database(),
head:     chain.CurrentHeader().Hash(),
clearIdx: chain.CurrentHeader().Number.Uint64(),
}
go pool.eventLoop()

return pool
}

// currentState returns the light state of the current head header
func (pool *TxPool) currentState(ctx context.Context) *state.StateDB {
return NewState(ctx, pool.chain.CurrentHeader(), pool.odr)
}

// GetNonce returns the "pending" nonce of a given address. It always queries
// the nonce belonging to the latest header too in order to detect if another
// client using the same key sent a transaction.
func (pool *TxPool) GetNonce(ctx context.Context, addr common.Address) (uint64, error) {
state := pool.currentState(ctx)
nonce := state.GetNonce(addr)
if state.Error() != nil {
```

```
return 0, state.Error()
}
sn, ok := pool.nonce[addr]
if ok && sn > nonce {
nonce = sn
}
if !ok || sn < nonce {
pool.nonce[addr] = nonce
}
return nonce, nil
}

type txBlockData struct {
BlockHash  common.Hash
BlockIndex uint64
Index      uint64
}

// storeTxBlockData stores the block position of a mined tx in the local db
func (pool *TxPool) storeTxBlockData(txh common.Hash, tbd txBlockData) {
//fmt.Println("storeTxBlockData", txh, tbd)
data, _ := rlp.EncodeToBytes(tbd)
pool.chainDb.Put(append(txh[:], byte(1)), data)
}

// removeTxBlockData removes the stored block position of a rolled back tx
func (pool *TxPool) removeTxBlockData(txh common.Hash) {
//fmt.Println("removeTxBlockData", txh)
pool.chainDb.Delete(append(txh[:], byte(1)))
}

// txStateChanges stores the recent changes between pending/mined states of
// transactions. True means mined, false means rolled back, no entry means no change
type txStateChanges map[common.Hash]bool

// setState sets the status of a tx to either recently mined or recently rolled back
func (txc txStateChanges) setState(txHash common.Hash, mined bool) {
val, ent := txc[txHash]
if ent && (val != mined) {
delete(txc, txHash)
} else {
txc[txHash] = mined
```

```
}
}

// getLists creates lists of mined and rolled back tx hashes
func (txc txStateChanges) getLists() (mined []common.Hash, rollback []common.Hash) {
for hash, val := range txc {
if val {
mined = append(mined, hash)
} else {
rollback = append(rollback, hash)
}
}
return
}

// checkMinedTxs checks newly added blocks for the currently pending transactions
// and marks them as mined if necessary. It also stores block position in the db
// and adds them to the received txStateChanges map.
func (pool *TxPool) checkMinedTxs(ctx context.Context, hash common.Hash, idx uint64, txc
txStateChanges) error {
//fmt.Println("checkMinedTxs")
if len(pool.pending) == 0 {
return nil
}
//fmt.Println("len(pool) =", len(pool.pending))

block, err := GetBlock(ctx, pool.odr, hash, idx)
var receipts types.Receipts
if err != nil {
//fmt.Println(err)
return err
}
//fmt.Println("len(block.Transactions()) =", len(block.Transactions()))

list := pool.mined[hash]
for i, tx := range block.Transactions() {
txHash := tx.Hash()
//fmt.Println(" txHash:", txHash)
if tx, ok := pool.pending[txHash]; ok {
//fmt.Println("TX FOUND")
if receipts == nil {
receipts, err = GetBlockReceipts(ctx, pool.odr, hash, idx)
```

```
if err != nil {
return err
}
if len(receipts) != len(block.Transactions()) {
panic(nil) // should never happen if hashes did match
}
core.SetReceiptsData(pool.config, block, receipts)
}
//fmt.Println("WriteReceipt", receipts[i].TxHash)
core.WriteReceipt(pool.chainDb, receipts[i])
pool.storeTxBlockData(txHash, txBlockData{hash, idx, uint64(i)})
delete(pool.pending, txHash)
list = append(list, tx)
txc.setState(txHash, true)
}
}
if list != nil {
pool.mined[hash] = list
}
return nil
}

// rollbackTxs marks the transactions contained in recently rolled back blocks
// as rolled back. It also removes block position info from the db and adds them
// to the received txStateChanges map.
func (pool *TxPool) rollbackTxs(hash common.Hash, txc txStateChanges) {
if list, ok := pool.mined[hash]; ok {
for _, tx := range list {
txHash := tx.Hash()
pool.removeTxBlockData(txHash)
pool.pending[txHash] = tx
txc.setState(txHash, false)
}
delete(pool.mined, hash)
}
}

// reorgOnNewHead sets a new head header, processing (and rolling back if necessary)
// the blocks since the last known head and returns a txStateChanges map containing
// the recently mined and rolled back transaction hashes. If an error (context
// timeout) occurs during checking new blocks, it leaves the locally known head
// at the latest checked block and still returns a valid txStateChanges, making it
```

```
// possible to continue checking the missing blocks at the next chain head event
func (pool *TxPool) reorgOnNewHead(ctx context.Context, newHeader *types.Header)
(txStateChanges, error) {
txc := make(txStateChanges)
oldh := pool.chain.GetHeaderByHash(pool.head)
newh := newHeader
// find common ancestor, create list of rolled back and new block hashes
var oldHashes, newHashes []common.Hash
for oldh.Hash() != newh.Hash() {
if oldh.Number.Uint64() >= newh.Number.Uint64() {
oldHashes = append(oldHashes, oldh.Hash())
oldh = pool.chain.GetHeader(oldh.ParentHash, oldh.Number.Uint64()-1)
}
if oldh.Number.Uint64() < newh.Number.Uint64() {
newHashes = append(newHashes, newh.Hash())
newh = pool.chain.GetHeader(newh.ParentHash, newh.Number.Uint64()-1)
if newh == nil {
// happens when CHT syncing, nothing to do
newh = oldh
}
}
}
if oldh.Number.Uint64() < pool.clearIdx {
pool.clearIdx = oldh.Number.Uint64()
}
// roll back old blocks
for _, hash := range oldHashes {
pool.rollbackTxs(hash, txc)
}
pool.head = oldh.Hash()
// check mined txs of new blocks (array is in reversed order)
for i := len(newHashes) - 1; i >= 0; i-- {
hash := newHashes[i]
if err := pool.checkMinedTxs(ctx, hash, newHeader.Number.Uint64()-uint64(i), txc); err != nil {
return txc, err
}
pool.head = hash
}

// clear old mined tx entries of old blocks
if idx := newHeader.Number.Uint64(); idx > pool.clearIdx+txPermanent {
idx2 := idx - txPermanent
```

```
if len(pool.mined) > 0 {
for i := pool.clearIdx; i < idx2; i++ {
hash := core.GetCanonicalHash(pool.chainDb, i)
if list, ok := pool.mined[hash]; ok {
hashes := make([]common.Hash, len(list))
for i, tx := range list {
hashes[i] = tx.Hash()
}
pool.relay.Discard(hashes)
delete(pool.mined, hash)
}
}
}
pool.clearIdx = idx2
}

return txc, nil
}

// blockCheckTimeout is the time limit for checking new blocks for mined
// transactions. Checking resumes at the next chain head event if timed out.
const blockCheckTimeout = time.Second * 3

// eventLoop processes chain head events and also notifies the tx relay backend
// about the new head hash and tx state changes
func (pool *TxPool) eventLoop() {
for ev := range pool.events.Chan() {
switch ev.Data.(type) {
case core.ChainHeadEvent:
pool.setNewHead(ev.Data.(core.ChainHeadEvent).Block.Header())
// hack in order to avoid hogging the lock; this part will
// be replaced by a subsequent PR.
time.Sleep(time.Millisecond)
}
}
}

func (pool *TxPool) setNewHead(head *types.Header) {
pool.mu.Lock()
defer pool.mu.Unlock()

ctx, cancel := context.WithTimeout(context.Background(), blockCheckTimeout)
```

```
defer cancel()

txc, _ := pool.reorgOnNewHead(ctx, head)
m, r := txc.getLists()
pool.relay.NewHead(pool.head, m, r)
pool.homestead = pool.config.IsHomestead(head.Number)
pool.signer = types.MakeSigner(pool.config, head.Number)
}

// Stop stops the light transaction pool
func (pool *TxPool) Stop() {
close(pool.quit)
pool.events.Unsubscribe()
log.Info("Transaction pool stopped")
}

// Stats returns the number of currently pending (locally created) transactions
func (pool *TxPool) Stats() (pending int) {
pool.mu.RLock()
defer pool.mu.RUnlock()

pending = len(pool.pending)
return
}

// validateTx checks whether a transaction is valid according to the consensus rules.
func (pool *TxPool) validateTx(ctx context.Context, tx *types.Transaction) error {
// Validate sender
var (
from common.Address
err  error
)

// Validate the transaction sender and it's sig. Throw
// if the from fields is invalid.
if from, err = types.Sender(pool.signer, tx); err != nil {
return core.ErrInvalidSender
}
// Last but not least check for nonce errors
currentState := pool.currentState(ctx)
if n := currentState.GetNonce(from); n > tx.Nonce() {
return core.ErrNonceTooLow
```

```
}

// Check the transaction doesn't exceed the current
// block limit gas.
header := pool.chain.GetHeaderByHash(pool.head)
if header.GasLimit.Cmp(tx.Gas()) < 0 {
return core.ErrGasLimit
}

// Transactions can't be negative. This may never happen
// using RLP decoded transactions but may occur if you create
// a transaction using the RPC for example.
if tx.Value().Sign() < 0 {
return core.ErrNegativeValue
}

// Transactor should have enough funds to cover the costs
// cost == V + GP * GL
if b := currentState.GetBalance(from); b.Cmp(tx.Cost()) < 0 {
return core.ErrInsufficientFunds
}

// Should supply enough intrinsic gas
if tx.Gas().Cmp(core.IntrinsicGas(tx.Data(), tx.To() == nil, pool.homestead)) < 0 {
return core.ErrIntrinsicGas
}

return currentState.Error()
}

// add validates a new transaction and sets its state pending if processable.
// It also updates the locally stored nonce if necessary.
func (self *TxPool) add(ctx context.Context, tx *types.Transaction) error {
hash := tx.Hash()

if self.pending[hash] != nil {
return fmt.Errorf("Known transaction (%x)", hash[:4])
}
err := self.validateTx(ctx, tx)
if err != nil {
return err
}
```

```
if _, ok := self.pending[hash]; !ok {
self.pending[hash] = tx

nonce := tx.Nonce() + 1

addr, _ := types.Sender(self.signer, tx)
if nonce > self.nonce[addr] {
self.nonce[addr] = nonce
}

// Notify the subscribers. This event is posted in a goroutine
// because it's possible that somewhere during the post "Remove transaction"
// gets called which will then wait for the global tx pool lock and deadlock.
go self.eventMux.Post(core.TxPreEvent{Tx: tx})
}

// Print a log message if low enough level is set
log.Debug("Pooled new transaction", "hash", hash, "from", log.Lazy{Fn: func() common.Address {
from, _ := types.Sender(self.signer, tx); return from }}, "to", tx.To())
return nil
}

// Add adds a transaction to the pool if valid and passes it to the tx relay
// backend
func (self *TxPool) Add(ctx context.Context, tx *types.Transaction) error {
self.mu.Lock()
defer self.mu.Unlock()

data, err := rlp.EncodeToBytes(tx)
if err != nil {
return err
}

if err := self.add(ctx, tx); err != nil {
return err
}
//fmt.Println("Send", tx.Hash())
self.relay.Send(types.Transactions{tx})

self.chainDb.Put(tx.Hash().Bytes(), data)
return nil
```

```
}

// AddTransactions adds all valid transactions to the pool and passes them to
// the tx relay backend
func (self *TxPool) AddBatch(ctx context.Context, txs []*types.Transaction) {
self.mu.Lock()
defer self.mu.Unlock()
var sendTx types.Transactions

for _, tx := range txs {
if err := self.add(ctx, tx); err == nil {
sendTx = append(sendTx, tx)
}
}
if len(sendTx) > 0 {
self.relay.Send(sendTx)
}
}

// GetTransaction returns a transaction if it is contained in the pool
// and nil otherwise.
func (tp *TxPool) GetTransaction(hash common.Hash) *types.Transaction {
// check the txs first
if tx, ok := tp.pending[hash]; ok {
return tx
}
return nil
}

// GetTransactions returns all currently processable transactions.
// The returned slice may be modified by the caller.
func (self *TxPool) GetTransactions() (txs types.Transactions, err error) {
self.mu.RLock()
defer self.mu.RUnlock()

txs = make(types.Transactions, len(self.pending))
i := 0
for _, tx := range self.pending {
txs[i] = tx
i++
}
return txs, nil
```

```
}

// Content retrieves the data content of the transaction pool, returning all the
// pending as well as queued transactions, grouped by account and nonce.
func (self *TxPool) Content() (map[common.Address]types.Transactions,
map[common.Address]types.Transactions) {
self.mu.RLock()
defer self.mu.RUnlock()

// Retrieve all the pending transactions and sort by account and by nonce
pending := make(map[common.Address]types.Transactions)
for _, tx := range self.pending {
account, _ := types.Sender(self.signer, tx)
pending[account] = append(pending[account], tx)
}
// There are no queued transactions in a light pool, just return an empty map
queued := make(map[common.Address]types.Transactions)
return pending, queued
}

// RemoveTransactions removes all given transactions from the pool.
func (self *TxPool) RemoveTransactions(txs types.Transactions) {
self.mu.Lock()
defer self.mu.Unlock()
var hashes []common.Hash
for _, tx := range txs {
//self.RemoveTx(tx.Hash())
hash := tx.Hash()
delete(self.pending, hash)
self.chainDb.Delete(hash[:])
hashes = append(hashes, hash)
}
self.relay.Discard(hashes)
}

// RemoveTx removes the transaction with the given hash from the pool.
func (pool *TxPool) RemoveTx(hash common.Hash) {
pool.mu.Lock()
defer pool.mu.Unlock()
// delete from pending pool
delete(pool.pending, hash)
pool.chainDb.Delete(hash[:])
```

```
pool.relay.Discard([]common.Hash{hash})
}

18:F:\git\coin\ethereum\go-ethereum\light\txpool_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package light

import (
"context"
"math"
"math/big"
"testing"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/consensus/ethash"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/core/vm"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/params"
)

type testTxRelay struct {
send, discard, mined chan int
}

func (self *testTxRelay) Send(txs types.Transactions) {
self.send <- len(txs)
}

func (self *testTxRelay) NewHead(head common.Hash, mined []common.Hash, rollback
[]common.Hash) {
m := len(mined)
if m != 0 {
self.mined <- m
}
}

func (self *testTxRelay) Discard(hashes []common.Hash) {
```

```
self.discard <- len(hashes)
}

const poolTestTxs = 1000
const poolTestBlocks = 100

// test tx 0..n-1
var testTx [poolTestTxs]*types.Transaction

// txs sent before block i
func sentTx(i int) int {
return int(math.Pow(float64(i)/float64(poolTestBlocks), 0.9) * poolTestTxs)
}

// txs included in block i or before that (minedTx(i) <= sentTx(i))
func minedTx(i int) int {
return int(math.Pow(float64(i)/float64(poolTestBlocks), 1.1) * poolTestTxs)
}

func txPoolTestChainGen(i int, block *core.BlockGen) {
s := minedTx(i)
e := minedTx(i + 1)
for i := s; i < e; i++ {
block.AddTx(testTx[i])
}
}

func TestTxPool(t *testing.T) {
for i := range testTx {
testTx[i], _ = types.SignTx(types.NewTransaction(uint64(i), acc1Addr, big.NewInt(10000),
bigTxGas, nil, nil), types.HomesteadSigner{}, testBankKey)
}

var (
evmux   = new(event.TypeMux)
sdb, _  = ethdb.NewMemDatabase()
ldb, _  = ethdb.NewMemDatabase()
gspec   = core.Genesis{Alloc: core.GenesisAlloc{testBankAddress: {Balance: testBankFunds}}}
genesis = gspec.MustCommit(sdb)
)
gspec.MustCommit(ldb)
// Assemble the test environment
```

```
blockchain, _ := core.NewBlockChain(sdb, params.TestChainConfig, ethash.NewFullFaker(),
evmux, vm.Config{})
gchain, _ := core.GenerateChain(params.TestChainConfig, genesis, sdb, poolTestBlocks,
txPoolTestChainGen)
if _, err := blockchain.InsertChain(gchain); err != nil {
panic(err)
}

odr := &testOdr{sdb: sdb, ldb: ldb}
relay := &testTxRelay{
send:    make(chan int, 1),
discard: make(chan int, 1),
mined:   make(chan int, 1),
}
lightchain, _ := NewLightChain(odr, params.TestChainConfig, ethash.NewFullFaker(), evmux)
txPermanent = 50
pool := NewTxPool(params.TestChainConfig, evmux, lightchain, relay)
ctx, cancel := context.WithTimeout(context.Background(), 1*time.Second)
defer cancel()

for ii, block := range gchain {
i := ii + 1
s := sentTx(i - 1)
e := sentTx(i)
for i := s; i < e; i++ {
pool.Add(ctx, testTx[i])
got := <-relay.send
exp := 1
if got != exp {
t.Errorf("relay.Send expected len = %d, got %d", exp, got)
}
}

if _, err := lightchain.InsertHeaderChain([]*types.Header{block.Header()}, 1); err != nil {
panic(err)
}

got := <-relay.mined
exp := minedTx(i) - minedTx(i-1)
if got != exp {
t.Errorf("relay.NewHead expected len(mined) = %d, got %d", exp, got)
}
```

```
exp = 0
if i > int(txPermanent)+1 {
exp = minedTx(i-int(txPermanent)-1) - minedTx(i-int(txPermanent)-2)
}
if exp != 0 {
got = <-relay.discard
if got != exp {
t.Errorf("relay.Discard expected len = %d, got %d", exp, got)
}
}
}
}
```

19:F:\git\coin\ethereum\go-ethereum\log\doc.go

Getting Started

To get started, you'll want to import the library:

```
import log "github.com/inconshreveable/log15"
```

Now you're ready to start logging:

```
func main() {
    log.Info("Program starting", "args", os.Args())
}
```

Convention

Because recording a human-meaningful message is common and good practice, the first argument to every
logging method is the value to the *implicit* key 'msg'.

Additionally, the level you choose for a message will be automatically added with the key 'lvl', and so
will the current timestamp with key 't'.

You may supply any additional context as a set of key/value pairs to the logging function. log15 allows

you to favor terseness, ordering, and speed over safety. This is a reasonable tradeoff for logging functions. You don't need to explicitly state keys/values, log15 understands that they alternate
in the variadic argument list:

```
log.Warn("size out of bounds", "low", lowBound, "high", highBound, "val", val)
```

If you really do favor your type-safety, you may choose to pass a log.Ctx instead:

```
log.Warn("size out of bounds", log.Ctx{"low": lowBound, "high": highBound, "val": val})
```

Context loggers

Frequently, you want to add context to a logger so that you can track actions associated with it. An http
request is a good example. You can easily create new loggers that have context that is automatically included
with each log line:

```
requestlogger := log.New("path", r.URL.Path)

// later
requestlogger.Debug("db txn commit", "duration", txnTimer.Finish())
```

This will output a log line that includes the path context that is attached to the logger:

```
lvl=dbug t=2014-05-02T16:07:23-0700 path=/repo/12/add_hook msg="db txn commit" duration=0.12
```

Handlers

The Handler interface defines where log lines are printed to and how they are formated. Handler is a
single interface that is inspired by net/http's handler interface:

```
type Handler interface {
    Log(r *Record) error
}
```

Handlers can filter records, format them, or dispatch to multiple other Handlers.
This package implements a number of Handlers for common logging patterns that are
easily composed to create flexible, custom logging structures.

Here's an example handler that prints logfmt output to Stdout:

```
handler := log.StreamHandler(os.Stdout, log.LogfmtFormat())
```

Here's an example handler that defers to two other handlers. One handler only prints records
from the rpc package in logfmt to standard out. The other prints records at Error level
or above in JSON formatted output to the file /var/log/service.json

```
handler := log.MultiHandler(
    log.LvlFilterHandler(log.LvlError, log.Must.FileHandler("/var/log/service.json",
log.JsonFormat())),
    log.MatchFilterHandler("pkg", "app/rpc" log.StdoutHandler())
)
```

Logging File Names and Line Numbers

This package implements three Handlers that add debugging information to the
context, CallerFileHandler, CallerFuncHandler and CallerStackHandler. Here's
an example that adds the source file and line number of each logging call to
the context.

```
h := log.CallerFileHandler(log.StdoutHandler)
log.Root().SetHandler(h)
...
log.Error("open file", "err", err)
```

This will output a line that looks like:

```
lvl=eror t=2014-05-02T16:07:23-0700 msg="open file" err="file not found" caller=data.go:42
```

Here's an example that logs the call stack rather than just the call site.

```
h := log.CallerStackHandler("%+v", log.StdoutHandler)
log.Root().SetHandler(h)
...
log.Error("open file", "err", err)
```

This will output a line that looks like:

```
    lvl=eror t=2014-05-02T16:07:23-0700 msg="open file" err="file not found"
stack="[pkg/data.go:42 pkg/cmd/main.go]"
```

The "%+v" format instructs the handler to include the path of the source file relative to the compile time GOPATH. The github.com/go-stack/stack package documents the full list of formatting verbs and modifiers available.

Custom Handlers

The Handler interface is so simple that it's also trivial to write your own. Let's create an example handler which tries to write to one handler, but if that fails it falls back to writing to another handler and includes the error that it encountered when trying to write to the primary. This might be useful when trying to log over a network socket, but if that fails you want to log those records to a file on disk.

```
    type BackupHandler struct {
        Primary Handler
        Secondary Handler
    }

    func (h *BackupHandler) Log (r *Record) error {
        err := h.Primary.Log(r)
        if err != nil {
            r.Ctx = append(ctx, "primary_err", err)
            return h.Secondary.Log(r)
        }
        return nil
    }
```

This pattern is so useful that a generic version that handles an arbitrary number of Handlers is included as part of this library called FailoverHandler.

Logging Expensive Operations

Sometimes, you want to log values that are extremely expensive to compute, but you don't want to pay
the price of computing them if you haven't turned up your logging level to a high level of detail.

This package provides a simple type to annotate a logging operation that you want to be evaluated lazily, just when it is about to be logged, so that it would not be evaluated if an upstream Handler filters it out. Just wrap any function which takes no arguments with the log.Lazy type. For example:

```
func factorRSAKey() (factors []int) {
    // return the factors of a very large number
}

log.Debug("factors", log.Lazy{factorRSAKey})
```

If this message is not logged for any reason (like logging at the Error level), then factorRSAKey is never evaluated.

Dynamic context values

The same log.Lazy mechanism can be used to attach context to a logger which you want to be evaluated when the message is logged, but not when the logger is created. For example, let's imagine
a game where you have Player objects:

```
type Player struct {
    name string
    alive bool
    log.Logger
}
```

You always want to log a player's name and whether they're alive or dead, so when you create the player
object, you might do:

```
p := &Player{name: name, alive: true}
p.Logger = log.New("name", p.name, "alive", p.alive)
```

Only now, even after a player has died, the logger will still report they are alive because the logging
context is evaluated when the logger was created. By using the Lazy wrapper, we can defer the evaluation
of whether the player is alive or not to each log message, so that the log records will reflect the player's
current state no matter when the log message is written:

```
p := &Player{name: name, alive: true}
isAlive := func() bool { return p.alive }
player.Logger = log.New("name", p.name, "alive", log.Lazy{isAlive})
```

Terminal Format

If log15 detects that stdout is a terminal, it will configure the default
handler for it (which is log.StdoutHandler) to use TerminalFormat. This format
logs records nicely for your terminal, including color-coded output based
on log level.

Error Handling

Becasuse log15 allows you to step around the type system, there are a few ways you can specify
invalid arguments to the logging functions. You could, for example, wrap something that is not
a zero-argument function with log.Lazy or pass a context key that is not a string. Since logging libraries
are typically the mechanism by which errors are reported, it would be onerous for the logging functions
to return errors. Instead, log15 handles errors by making these guarantees to you:

- Any log record containing an error will still be printed with the error explained to you as part of the log record.

- Any log record containing an error will include the context key LOG15_ERROR, enabling you to easily
(and if you like, automatically) detect if any of your logging calls are passing bad values.

Understanding this, you might wonder why the Handler interface can return an error value in its Log method. Handlers
are encouraged to return errors only if they fail to write their log records out to an external source like if the
syslog daemon is not responding. This allows the construction of useful handlers which cope with those failures
like the FailoverHandler.

Library Use

log15 is intended to be useful for library authors as a way to provide configurable logging to
users of their library. Best practice for use in a library is to always disable all output for your logger
by default and to provide a public Logger instance that consumers of your library can configure.
Like so:

```
package yourlib

import "github.com/inconshreveable/log15"
```

```
var Log = log.New()

func init() {
    Log.SetHandler(log.DiscardHandler())
}
```

Users of your library may then enable it if they like:

```
import "github.com/inconshreveable/log15"
import "example.com/yourlib"

func main() {
    handler := // custom handler setup
    yourlib.Log.SetHandler(handler)
}
```

Best practices attaching logger context

The ability to attach context to a logger is a powerful one. Where should you do it and why? I favor embedding a Logger directly into any persistent object in my application and adding unique, tracing context keys to it. For instance, imagine I am writing a web browser:

```
type Tab struct {
    url string
    render *RenderingContext
    // ...

    Logger
}

func NewTab(url string) *Tab {
    return &Tab {
        // ...
        url: url,

        Logger: log.New("url", url),
    }
}
```

When a new tab is created, I assign a logger to it with the url of the tab as context so it can easily be traced through the logs.

Now, whenever we perform any operation with the tab, we'll log with its embedded logger and it will include the tab title automatically:

```
tab.Debug("moved position", "idx", tab.idx)
```

There's only one problem. What if the tab url changes? We could use log.Lazy to make sure the current url is always written, but that would mean that we couldn't trace a tab's full lifetime through our logs after the user navigate to a new URL.

Instead, think about what values to attach to your loggers the same way you think about what to use as a key in a SQL database schema. If it's possible to use a natural key that is unique for the lifetime of the object, do so. But otherwise, log15's ext package has a handy RandId function to let you generate what you might call "surrogate keys" They're just random hex identifiers to use for tracing. Back to our Tab example, we would prefer to set up our Logger like so:

```
import logext "github.com/inconshreveable/log15/ext"

t := &Tab {
    // ...
    url: url,
}

t.Logger = log.New("id", logext.RandId(8), "url", log.Lazy{t.getUrl})
return t
```

Now we'll have a unique traceable identifier even across loading new urls, but we'll still be able to see the tab's current url in the log messages.

Must

For all Handler functions which can return an error, there is a version of that function which will return no error but panics on failure. They are all available on the Must object. For example:

```
log.Must.FileHandler("/path", log.JsonFormat)
log.Must.NetHandler("tcp", ":1234", log.JsonFormat)
```

Inspiration and Credit

All of the following excellent projects inspired the design of this library:

code.google.com/p/log4go

github.com/op/go-logging

github.com/technoweenie/grohl

github.com/Sirupsen/logrus

github.com/kr/logfmt

github.com/spacemonkeygo/spacelog

golang's stdlib, notably io and net/http

The Name

https://xkcd.com/927/

*/
package log

20:F:\git\coin\ethereum\go-ethereum\log\format.go

```
const (
timeFormat     = "2006-01-02T15:04:05-0700"
termTimeFormat = "01-02|15:04:05"
floatFormat    = 'f'
termMsgJust    = 40
)

// locationTrims are trimmed for display to avoid unwieldy log lines.
var locationTrims = []string{
"github.com/ethereum/go-ethereum/",
}

// PrintOrigins sets or unsets log location (file:line) printing for terminal
// format output.
func PrintOrigins(print bool) {
if print {
atomic.StoreUint32(&locationEnabled, 1)
```

```
} else {
atomic.StoreUint32(&locationEnabled, 0)
}
}

// locationEnabled is an atomic flag controlling whether the terminal formatter
// should append the log locations too when printing entries.
var locationEnabled uint32

// locationLength is the maxmimum path length encountered, which all logs are
// padded to to aid in alignment.
var locationLength uint32

// fieldPadding is a global map with maximum field value lengths seen until now
// to allow padding log contexts in a bit smarter way.
var fieldPadding = make(map[string]int)

// fieldPaddingLock is a global mutex protecting the field padding map.
var fieldPaddingLock sync.RWMutex

type Format interface {
Format(r *Record) []byte
}

// FormatFunc returns a new Format object which uses
// the given function to perform record formatting.
func FormatFunc(f func(*Record) []byte) Format {
return formatFunc(f)
}

type formatFunc func(*Record) []byte

func (f formatFunc) Format(r *Record) []byte {
return f(r)
}

// TerminalStringer is an analogous interface to the stdlib stringer, allowing
// own types to have custom shortened serialization formats when printed to the
// screen.
type TerminalStringer interface {
TerminalString() string
}
```

```
// TerminalFormat formats log records optimized for human readability on
// a terminal with color-coded level output and terser human friendly timestamp.
// This format should only be used for interactive programs or while developing.
//
//     [TIME] [LEVEL] MESAGE key=value key=value ...
//
// Example:
//
//     [May 16 20:58:45] [DBUG] remove route ns=haproxy addr=127.0.0.1:50002
//
func TerminalFormat(usecolor bool) Format {
return FormatFunc(func(r *Record) []byte {
var color = 0
if usecolor {
switch r.Lvl {
case LvlCrit:
color = 35
case LvlError:
color = 31
case LvlWarn:
color = 33
case LvlInfo:
color = 32
case LvlDebug:
color = 36
case LvlTrace:
color = 34
}
}

b := &bytes.Buffer{}
lvl := r.Lvl.AlignedString()
if atomic.LoadUint32(&locationEnabled) != 0 {
// Log origin printing was requested, format the location path and line number
location := fmt.Sprintf("%+v", r.Call)
for _, prefix := range locationTrims {
location = strings.TrimPrefix(location, prefix)
}
// Maintain the maximum location length for fancyer alignment
align := int(atomic.LoadUint32(&locationLength))
if align < len(location) {
```

```
align = len(location)
atomic.StoreUint32(&locationLength, uint32(align))
}
padding := strings.Repeat(" ", align-len(location))

// Assemble and print the log heading
if color > 0 {
fmt.Fprintf(b, "\x1b[%dm%s\x1b[0m[%s|%s]%s %s ", color, lvl, r.Time.Format(termTimeFormat),
location, padding, r.Msg)
} else {
fmt.Fprintf(b, "%s[%s|%s]%s %s ", lvl, r.Time.Format(termTimeFormat), location, padding, r.Msg)
}
} else {
if color > 0 {
fmt.Fprintf(b, "\x1b[%dm%s\x1b[0m[%s] %s ", color, lvl, r.Time.Format(termTimeFormat), r.Msg)
} else {
fmt.Fprintf(b, "%s[%s] %s ", lvl, r.Time.Format(termTimeFormat), r.Msg)
}
}
// try to justify the log output for short messages
length := utf8.RuneCountInString(r.Msg)
if len(r.Ctx) > 0 && length < termMsgJust {
b.Write(bytes.Repeat([]byte{' '}, termMsgJust-length))
}
// print the keys logfmt style
logfmt(b, r.Ctx, color, true)
return b.Bytes()
})
}

// LogfmtFormat prints records in logfmt format, an easy machine-parseable but human-readable
// format for key/value pairs.
//
// For more details see: http://godoc.org/github.com/kr/logfmt
//
func LogfmtFormat() Format {
return FormatFunc(func(r *Record) []byte {
common := []interface{}{r.KeyNames.Time, r.Time, r.KeyNames.Lvl, r.Lvl, r.KeyNames.Msg,
r.Msg}
buf := &bytes.Buffer{}
logfmt(buf, append(common, r.Ctx...), 0, false)
return buf.Bytes()
```

```
})
}

func logfmt(buf *bytes.Buffer, ctx []interface{}, color int, term bool) {
for i := 0; i < len(ctx); i += 2 {
if i != 0 {
buf.WriteByte(' ')
}

k, ok := ctx[i].(string)
v := formatLogfmtValue(ctx[i+1], term)
if !ok {
k, v = errorKey, formatLogfmtValue(k, term)
}

// XXX: we should probably check that all of your key bytes aren't invalid
fieldPaddingLock.RLock()
padding := fieldPadding[k]
fieldPaddingLock.RUnlock()

length := utf8.RuneCountInString(v)
if padding < length {
padding = length

fieldPaddingLock.Lock()
fieldPadding[k] = padding
fieldPaddingLock.Unlock()
}
if color > 0 {
fmt.Fprintf(buf, "\x1b[%dm%s\x1b[0m=", color, k)
} else {
buf.WriteString(k)
buf.WriteByte('=')
}
buf.WriteString(v)
if i < len(ctx)-2 {
buf.Write(bytes.Repeat([]byte{' '}, padding-length))
}
}
buf.WriteByte('\n')
}
```

```
// JsonFormat formats log records as JSON objects separated by newlines.
// It is the equivalent of JsonFormatEx(false, true).
func JsonFormat() Format {
return JsonFormatEx(false, true)
}

// JsonFormatEx formats log records as JSON objects. If pretty is true,
// records will be pretty-printed. If lineSeparated is true, records
// will be logged with a new line between each record.
func JsonFormatEx(pretty, lineSeparated bool) Format {
jsonMarshal := json.Marshal
if pretty {
jsonMarshal = func(v interface{}) ([]byte, error) {
return json.MarshalIndent(v, "", "    ")
}
}

return FormatFunc(func(r *Record) []byte {
props := make(map[string]interface{})

props[r.KeyNames.Time] = r.Time
props[r.KeyNames.Lvl] = r.Lvl.String()
props[r.KeyNames.Msg] = r.Msg

for i := 0; i < len(r.Ctx); i += 2 {
k, ok := r.Ctx[i].(string)
if !ok {
props[errorKey] = fmt.Sprintf("%+v is not a string key", r.Ctx[i])
}
props[k] = formatJsonValue(r.Ctx[i+1])
}

b, err := jsonMarshal(props)
if err != nil {
b, _ = jsonMarshal(map[string]string{
errorKey: err.Error(),
})
return b
}

if lineSeparated {
b = append(b, '\n')
```

```
}

return b
})
}

func formatShared(value interface{}) (result interface{}) {
defer func() {
if err := recover(); err != nil {
if v := reflect.ValueOf(value); v.Kind() == reflect.Ptr && v.IsNil() {
result = "nil"
} else {
panic(err)
}
}
}()

switch v := value.(type) {
case time.Time:
return v.Format(timeFormat)

case error:
return v.Error()

case fmt.Stringer:
return v.String()

default:
return v
}
}

func formatJsonValue(value interface{}) interface{} {
value = formatShared(value)
switch value.(type) {
case int, int8, int16, int32, int64, float32, float64, uint, uint8, uint16, uint32, uint64, string:
return value
default:
return fmt.Sprintf("%+v", value)
}
}
```

```
// formatValue formats a value for serialization
func formatLogfmtValue(value interface{}, term bool) string {
if value == nil {
return "nil"
}

if t, ok := value.(time.Time); ok {
// Performance optimization: No need for escaping since the provided
// timeFormat doesn't have any escape characters, and escaping is
// expensive.
return t.Format(timeFormat)
}
if term {
if s, ok := value.(TerminalStringer); ok {
// Custom terminal stringer provided, use that
return escapeString(s.TerminalString())
}
}
value = formatShared(value)
switch v := value.(type) {
case bool:
return strconv.FormatBool(v)
case float32:
return strconv.FormatFloat(float64(v), floatFormat, 3, 64)
case float64:
return strconv.FormatFloat(v, floatFormat, 3, 64)
case int, int8, int16, int32, int64, uint, uint8, uint16, uint32, uint64:
return fmt.Sprintf("%d", value)
case string:
return escapeString(v)
default:
return escapeString(fmt.Sprintf("%+v", value))
}
}

var stringBufPool = sync.Pool{
New: func() interface{} { return new(bytes.Buffer) },
}

func escapeString(s string) string {
needsQuotes := false
needsEscape := false
```

```
for _, r := range s {
if r <= ' ' || r == '=' || r == '"' {
needsQuotes = true
}
if r == '\\' || r == '"' || r == '\n' || r == '\r' || r == '\t' {
needsEscape = true
}
}
if needsEscape == false && needsQuotes == false {
return s
}
e := stringBufPool.Get().(*bytes.Buffer)
e.WriteByte('"')
for _, r := range s {
switch r {
case '\\', '"':
e.WriteByte('\\')
e.WriteByte(byte(r))
case '\n':
e.WriteString("\\n")
case '\r':
e.WriteString("\\r")
case '\t':
e.WriteString("\\t")
default:
e.WriteRune(r)
}
}
e.WriteByte('"')
var ret string
if needsQuotes {
ret = e.String()
} else {
ret = string(e.Bytes()[1 : e.Len()-1])
}
e.Reset()
stringBufPool.Put(e)
return ret
}
```

21:F:\git\coin\ethereum\go-ethereum\log\handler.go

```
// The Handler interface defines where and how log records are written.
```

```
// Handlers are composable, providing you great flexibility in combining
// them to achieve the logging structure that suits your applications.
type Handler interface {
Log(r *Record) error
}

// FuncHandler returns a Handler that logs records with the given
// function.
func FuncHandler(fn func(r *Record) error) Handler {
return funcHandler(fn)
}

type funcHandler func(r *Record) error

func (h funcHandler) Log(r *Record) error {
return h(r)
}

// StreamHandler writes log records to an io.Writer
// with the given format. StreamHandler can be used
// to easily begin writing log records to other
// outputs.
//
// StreamHandler wraps itself with LazyHandler and SyncHandler
// to evaluate Lazy objects and perform safe concurrent writes.
func StreamHandler(wr io.Writer, fmtr Format) Handler {
h := FuncHandler(func(r *Record) error {
_, err := wr.Write(fmtr.Format(r))
return err
})
return LazyHandler(SyncHandler(h))
}

// SyncHandler can be wrapped around a handler to guarantee that
// only a single Log operation can proceed at a time. It's necessary
// for thread-safe concurrent writes.
func SyncHandler(h Handler) Handler {
var mu sync.Mutex
return FuncHandler(func(r *Record) error {
defer mu.Unlock()
mu.Lock()
return h.Log(r)
```

```
})
}

// FileHandler returns a handler which writes log records to the give file
// using the given format. If the path
// already exists, FileHandler will append to the given file. If it does not,
// FileHandler will create the file with mode 0644.
func FileHandler(path string, fmtr Format) (Handler, error) {
f, err := os.OpenFile(path, os.O_CREATE|os.O_APPEND|os.O_WRONLY, 0644)
if err != nil {
return nil, err
}
return closingHandler{f, StreamHandler(f, fmtr)}, nil
}

// NetHandler opens a socket to the given address and writes records
// over the connection.
func NetHandler(network, addr string, fmtr Format) (Handler, error) {
conn, err := net.Dial(network, addr)
if err != nil {
return nil, err
}

return closingHandler{conn, StreamHandler(conn, fmtr)}, nil
}

// XXX: closingHandler is essentially unused at the moment
// it's meant for a future time when the Handler interface supports
// a possible Close() operation
type closingHandler struct {
io.WriteCloser
Handler
}

func (h *closingHandler) Close() error {
return h.WriteCloser.Close()
}

// CallerFileHandler returns a Handler that adds the line number and file of
// the calling function to the context with key "caller".
func CallerFileHandler(h Handler) Handler {
return FuncHandler(func(r *Record) error {
```

```
r.Ctx = append(r.Ctx, "caller", fmt.Sprint(r.Call))
return h.Log(r)
})
}

// CallerFuncHandler returns a Handler that adds the calling function name to
// the context with key "fn".
func CallerFuncHandler(h Handler) Handler {
return FuncHandler(func(r *Record) error {
r.Ctx = append(r.Ctx, "fn", formatCall("%+n", r.Call))
return h.Log(r)
})
}

// This function is here to please go vet on Go < 1.8.
func formatCall(format string, c stack.Call) string {
return fmt.Sprintf(format, c)
}

// CallerStackHandler returns a Handler that adds a stack trace to the context
// with key "stack". The stack trace is formated as a space separated list of
// call sites inside matching []'s. The most recent call site is listed first.
// Each call site is formatted according to format. See the documentation of
// package github.com/go-stack/stack for the list of supported formats.
func CallerStackHandler(format string, h Handler) Handler {
return FuncHandler(func(r *Record) error {
s := stack.Trace().TrimBelow(r.Call).TrimRuntime()
if len(s) > 0 {
r.Ctx = append(r.Ctx, "stack", fmt.Sprintf(format, s))
}
return h.Log(r)
})
}

// FilterHandler returns a Handler that only writes records to the
// wrapped Handler if the given function evaluates true. For example,
// to only log records where the 'err' key is not nil:
//
//    logger.SetHandler(FilterHandler(func(r *Record) bool {
//        for i := 0; i < len(r.Ctx); i += 2 {
//            if r.Ctx[i] == "err" {
//                return r.Ctx[i+1] != nil
```

```
//            }
//        }
//        return false
//    }, h))
//
func FilterHandler(fn func(r *Record) bool, h Handler) Handler {
return FuncHandler(func(r *Record) error {
if fn(r) {
return h.Log(r)
}
return nil
})
}

// MatchFilterHandler returns a Handler that only writes records
// to the wrapped Handler if the given key in the logged
// context matches the value. For example, to only log records
// from your ui package:
//
//    log.MatchFilterHandler("pkg", "app/ui", log.StdoutHandler)
//
func MatchFilterHandler(key string, value interface{}, h Handler) Handler {
return FilterHandler(func(r *Record) (pass bool) {
switch key {
case r.KeyNames.Lvl:
return r.Lvl == value
case r.KeyNames.Time:
return r.Time == value
case r.KeyNames.Msg:
return r.Msg == value
}

for i := 0; i < len(r.Ctx); i += 2 {
if r.Ctx[i] == key {
return r.Ctx[i+1] == value
}
}
return false
}, h)
}

// LvlFilterHandler returns a Handler that only writes
```

```
// records which are less than the given verbosity
// level to the wrapped Handler. For example, to only
// log Error/Crit records:
//
//     log.LvlFilterHandler(log.LvlError, log.StdoutHandler)
//
func LvlFilterHandler(maxLvl Lvl, h Handler) Handler {
return FilterHandler(func(r *Record) (pass bool) {
return r.Lvl <= maxLvl
}, h)
}

// A MultiHandler dispatches any write to each of its handlers.
// This is useful for writing different types of log information
// to different locations. For example, to log to a file and
// standard error:
//
//     log.MultiHandler(
//         log.Must.FileHandler("/var/log/app.log", log.LogfmtFormat()),
//         log.StderrHandler)
//
func MultiHandler(hs ...Handler) Handler {
return FuncHandler(func(r *Record) error {
for _, h := range hs {
// what to do about failures?
h.Log(r)
}
return nil
})
}

// A FailoverHandler writes all log records to the first handler
// specified, but will failover and write to the second handler if
// the first handler has failed, and so on for all handlers specified.
// For example you might want to log to a network socket, but failover
// to writing to a file if the network fails, and then to
// standard out if the file write fails:
//
//     log.FailoverHandler(
//         log.Must.NetHandler("tcp", ":9090", log.JsonFormat()),
//         log.Must.FileHandler("/var/log/app.log", log.LogfmtFormat()),
//         log.StdoutHandler)
```

```
//
// All writes that do not go to the first handler will add context with keys of
// the form "failover_err_{idx}" which explain the error encountered while
// trying to write to the handlers before them in the list.
func FailoverHandler(hs ...Handler) Handler {
return FuncHandler(func(r *Record) error {
var err error
for i, h := range hs {
err = h.Log(r)
if err == nil {
return nil
} else {
r.Ctx = append(r.Ctx, fmt.Sprintf("failover_err_%d", i), err)
}
}

return err
})
}

// ChannelHandler writes all records to the given channel.
// It blocks if the channel is full. Useful for async processing
// of log messages, it's used by BufferedHandler.
func ChannelHandler(recs chan<- *Record) Handler {
return FuncHandler(func(r *Record) error {
recs <- r
return nil
})
}

// BufferedHandler writes all records to a buffered
// channel of the given size which flushes into the wrapped
// handler whenever it is available for writing. Since these
// writes happen asynchronously, all writes to a BufferedHandler
// never return an error and any errors from the wrapped handler are ignored.
func BufferedHandler(bufSize int, h Handler) Handler {
recs := make(chan *Record, bufSize)
go func() {
for m := range recs {
_ = h.Log(m)
}
}()
```

```
return ChannelHandler(recs)
}

// LazyHandler writes all values to the wrapped handler after evaluating
// any lazy functions in the record's context. It is already wrapped
// around StreamHandler and SyslogHandler in this library, you'll only need
// it if you write your own Handler.
func LazyHandler(h Handler) Handler {
return FuncHandler(func(r *Record) error {
// go through the values (odd indices) and reassign
// the values of any lazy fn to the result of its execution
hadErr := false
for i := 1; i < len(r.Ctx); i += 2 {
lz, ok := r.Ctx[i].(Lazy)
if ok {
v, err := evaluateLazy(lz)
if err != nil {
hadErr = true
r.Ctx[i] = err
} else {
if cs, ok := v.(stack.CallStack); ok {
v = cs.TrimBelow(r.Call).TrimRuntime()
}
r.Ctx[i] = v
}
}
}

if hadErr {
r.Ctx = append(r.Ctx, errorKey, "bad lazy")
}

return h.Log(r)
})
}

func evaluateLazy(lz Lazy) (interface{}, error) {
t := reflect.TypeOf(lz.Fn)

if t.Kind() != reflect.Func {
return nil, fmt.Errorf("INVALID_LAZY, not func: %+v", lz.Fn)
}
```

```
if t.NumIn() > 0 {
return nil, fmt.Errorf("INVALID_LAZY, func takes args: %+v", lz.Fn)
}

if t.NumOut() == 0 {
return nil, fmt.Errorf("INVALID_LAZY, no func return val: %+v", lz.Fn)
}

value := reflect.ValueOf(lz.Fn)
results := value.Call([]reflect.Value{})
if len(results) == 1 {
return results[0].Interface(), nil
} else {
values := make([]interface{}, len(results))
for i, v := range results {
values[i] = v.Interface()
}
return values, nil
}
}

// DiscardHandler reports success for all writes but does nothing.
// It is useful for dynamically disabling logging at runtime via
// a Logger's SetHandler method.
func DiscardHandler() Handler {
return FuncHandler(func(r *Record) error {
return nil
})
}

// The Must object provides the following Handler creation functions
// which instead of returning an error parameter only return a Handler
// and panic on failure: FileHandler, NetHandler, SyslogHandler, SyslogNetHandler
var Must muster

func must(h Handler, err error) Handler {
if err != nil {
panic(err)
}
return h
}
```

```
type muster struct{}

func (m muster) FileHandler(path string, fmtr Format) Handler {
return must(FileHandler(path, fmtr))
}

func (m muster) NetHandler(network, addr string, fmtr Format) Handler {
return must(NetHandler(network, addr, fmtr))
}

22:F:\git\coin\ethereum\go-ethereum\log\handler_glog.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package log

import (
"errors"
"fmt"
"regexp"
"runtime"
"strconv"
"strings"
"sync"
"sync/atomic"
)

// errVmoduleSyntax is returned when a user vmodule pattern is invalid.
var errVmoduleSyntax = errors.New("expect comma-separated list of filename=N")

// errTraceSyntax is returned when a user backtrace pattern is invalid.
var errTraceSyntax = errors.New("expect file.go:234")

// GlogHandler is a log handler that mimics the filtering features of Google's
// glog logger: setting global log levels; overriding with callsite pattern
// matches; and requesting backtraces at certain positions.
type GlogHandler struct {
origin Handler // The origin handler this wraps

level     uint32 // Current log level, atomically accessible
override  uint32 // Flag whether overrides are used, atomically accessible
backtrace uint32 // Flag whether backtrace location is set
```

```
patterns  []pattern       // Current list of patterns to override with
siteCache map[uintptr]Lvl // Cache of callsite pattern evaluations
location  string          // file:line location where to do a stackdump at
lock      sync.RWMutex    // Lock protecting the override pattern list
}

// NewGlogHandler creates a new log handler with filtering functionality similar
// to Google's glog logger. The returned handler implements Handler.
func NewGlogHandler(h Handler) *GlogHandler {
return &GlogHandler{
origin: h,
}
}

// pattern contains a filter for the Vmodule option, holding a verbosity level
// and a file pattern to match.
type pattern struct {
pattern *regexp.Regexp
level   Lvl
}

// Verbosity sets the glog verbosity ceiling. The verbosity of individual packages
// and source files can be raised using Vmodule.
func (h *GlogHandler) Verbosity(level Lvl) {
atomic.StoreUint32(&h.level, uint32(level))
}

// Vmodule sets the glog verbosity pattern.
//
// The syntax of the argument is a comma-separated list of pattern=N, where the
// pattern is a literal file name or "glob" pattern matching and N is a V level.
//
// For instance:
//
//  pattern="gopher.go=3"
//   sets the V level to 3 in all Go files named "gopher.go"
//
//  pattern="foo=3"
//   sets V to 3 in all files of any packages whose import path ends in "foo"
//
//  pattern="foo/*=3"
```

```
//   sets V to 3 in all files of any packages whose import path contains "foo"
func (h *GlogHandler) Vmodule(ruleset string) error {
var filter []pattern
for _, rule := range strings.Split(ruleset, ",") {
// Empty strings such as from a trailing comma can be ignored
if len(rule) == 0 {
continue
}
// Ensure we have a pattern = level filter rule
parts := strings.Split(rule, "=")
if len(parts) != 2 {
return errVmoduleSyntax
}
parts[0] = strings.TrimSpace(parts[0])
parts[1] = strings.TrimSpace(parts[1])
if len(parts[0]) == 0 || len(parts[1]) == 0 {
return errVmoduleSyntax
}
// Parse the level and if correct, assemble the filter rule
level, err := strconv.Atoi(parts[1])
if err != nil {
return errVmoduleSyntax
}
if level <= 0 {
continue // Ignore. It's harmless but no point in paying the overhead.
}
// Compile the rule pattern into a regular expression
matcher := ".*"
for _, comp := range strings.Split(parts[0], "/") {
if comp == "*" {
matcher += "(/.*)?"
} else if comp != "" {
matcher += "/" + regexp.QuoteMeta(comp)
}
}
if !strings.HasSuffix(parts[0], ".go") {
matcher += "/[^/]+\\.go"
}
matcher = matcher + "$"

re, _ := regexp.Compile(matcher)
filter = append(filter, pattern{re, Lvl(level)})
```

```
}
// Swap out the vmodule pattern for the new filter system
h.lock.Lock()
defer h.lock.Unlock()

h.patterns = filter
h.siteCache = make(map[uintptr]Lvl)
atomic.StoreUint32(&h.override, uint32(len(filter)))

return nil
}

// BacktraceAt sets the glog backtrace location. When set to a file and line
// number holding a logging statement, a stack trace will be written to the Info
// log whenever execution hits that statement.
//
// Unlike with Vmodule, the ".go" must be present.
func (h *GlogHandler) BacktraceAt(location string) error {
// Ensure the backtrace location contains two non-empty elements
parts := strings.Split(location, ":")
if len(parts) != 2 {
return errTraceSyntax
}
parts[0] = strings.TrimSpace(parts[0])
parts[1] = strings.TrimSpace(parts[1])
if len(parts[0]) == 0 || len(parts[1]) == 0 {
return errTraceSyntax
}
// Ensure the .go prefix is present and the line is valid
if !strings.HasSuffix(parts[0], ".go") {
return errTraceSyntax
}
if _, err := strconv.Atoi(parts[1]); err != nil {
return errTraceSyntax
}
// All seems valid
h.lock.Lock()
defer h.lock.Unlock()

h.location = location
atomic.StoreUint32(&h.backtrace, uint32(len(location)))
```

```
return nil
}

// Log implements Handler.Log, filtering a log record through the global, local
// and backtrace filters, finally emitting it if either allow it through.
func (h *GlogHandler) Log(r *Record) error {
// If backtracing is requested, check whether this is the callsite
if atomic.LoadUint32(&h.backtrace) > 0 {
// Everything below here is slow. Although we could cache the call sites the
// same way as for vmodule, backtracing is so rare it's not worth the extra
// complexity.
h.lock.RLock()
match := h.location == r.Call.String()
h.lock.RUnlock()

if match {
// Callsite matched, raise the log level to info and gather the stacks
r.Lvl = LvlInfo

buf := make([]byte, 1024*1024)
buf = buf[:runtime.Stack(buf, true)]
r.Msg += "\n\n" + string(buf)
}
}
// If the global log level allows, fast track logging
if atomic.LoadUint32(&h.level) >= uint32(r.Lvl) {
return h.origin.Log(r)
}
// If no local overrides are present, fast track skipping
if atomic.LoadUint32(&h.override) == 0 {
return nil
}
// Check callsite cache for previously calculated log levels
h.lock.RLock()
lvl, ok := h.siteCache[r.Call.PC()]
h.lock.RUnlock()

// If we didn't cache the callsite yet, calculate it
if !ok {
h.lock.Lock()
for _, rule := range h.patterns {
if rule.pattern.MatchString(fmt.Sprintf("%+s", r.Call)) {
```

```
h.siteCache[r.Call.PC()], lvl, ok = rule.level, rule.level, true
break
}
}
// If no rule matched, remember to drop log the next time
if !ok {
h.siteCache[r.Call.PC()] = 0
}
h.lock.Unlock()
}
if lvl >= r.Lvl {
return h.origin.Log(r)
}
return nil
}
```

23:F:\git\coin\ethereum\go-ethereum\log\handler_go13.go

```
func (h *swapHandler) Log(r *Record) error {
return h.Get().Log(r)
}

func (h *swapHandler) Get() Handler {
return *(*Handler)(atomic.LoadPointer(&h.handler))
}

func (h *swapHandler) Swap(newHandler Handler) {
atomic.StorePointer(&h.handler, unsafe.Pointer(&newHandler))
}
```

24:F:\git\coin\ethereum\go-ethereum\log\handler_go14.go

```
}

func (h *swapHandler) Swap(newHandler Handler) {
h.handler.Store(&newHandler)
}

func (h *swapHandler) Get() Handler {
return *h.handler.Load().(*Handler)
}
```

25:F:\git\coin\ethereum\go-ethereum\log\logger.go

```
type Lvl int

const (
LvlCrit Lvl = iota
LvlError
LvlWarn
LvlInfo
LvlDebug
LvlTrace
)

// Aligned returns a 5-character string containing the name of a Lvl.
func (l Lvl) AlignedString() string {
switch l {
case LvlTrace:
return "TRACE"
case LvlDebug:
return "DEBUG"
case LvlInfo:
return "INFO "
case LvlWarn:
return "WARN "
case LvlError:
return "ERROR"
case LvlCrit:
return "CRIT "
default:
panic("bad level")
}
}

// Strings returns the name of a Lvl.
func (l Lvl) String() string {
switch l {
case LvlTrace:
return "trce"
case LvlDebug:
return "dbug"
case LvlInfo:
return "info"
case LvlWarn:
```

```
return "warn"
case LvlError:
return "eror"
case LvlCrit:
return "crit"
default:
panic("bad level")
}
}

// Returns the appropriate Lvl from a string name.
// Useful for parsing command line args and configuration files.
func LvlFromString(lvlString string) (Lvl, error) {
switch lvlString {
case "trace", "trce":
return LvlTrace, nil
case "debug", "dbug":
return LvlDebug, nil
case "info":
return LvlInfo, nil
case "warn":
return LvlWarn, nil
case "error", "eror":
return LvlError, nil
case "crit":
return LvlCrit, nil
default:
return LvlDebug, fmt.Errorf("Unknown level: %v", lvlString)
}
}

// A Record is what a Logger asks its handler to write
type Record struct {
Time     time.Time
Lvl      Lvl
Msg      string
Ctx      []interface{}
Call     stack.Call
KeyNames RecordKeyNames
}

type RecordKeyNames struct {
```

```
Time string
Msg  string
Lvl  string
}

// A Logger writes key/value pairs to a Handler
type Logger interface {
// New returns a new Logger that has this logger's context plus the given context
New(ctx ...interface{}) Logger

// GetHandler gets the handler associated with the logger.
GetHandler() Handler

// SetHandler updates the logger to write records to the specified handler.
SetHandler(h Handler)

// Log a message at the given level with context key/value pairs
Trace(msg string, ctx ...interface{})
Debug(msg string, ctx ...interface{})
Info(msg string, ctx ...interface{})
Warn(msg string, ctx ...interface{})
Error(msg string, ctx ...interface{})
Crit(msg string, ctx ...interface{})
}

type logger struct {
ctx []interface{}
h   *swapHandler
}

func (l *logger) write(msg string, lvl Lvl, ctx []interface{}) {
l.h.Log(&Record{
Time: time.Now(),
Lvl:  lvl,
Msg:  msg,
Ctx:  newContext(l.ctx, ctx),
Call: stack.Caller(2),
KeyNames: RecordKeyNames{
Time: timeKey,
Msg:  msgKey,
Lvl:  lvlKey,
},
```

```
})
}

func (l *logger) New(ctx ...interface{}) Logger {
child := &logger{newContext(l.ctx, ctx), new(swapHandler)}
child.SetHandler(l.h)
return child
}

func newContext(prefix []interface{}, suffix []interface{}) []interface{} {
normalizedSuffix := normalize(suffix)
newCtx := make([]interface{}, len(prefix)+len(normalizedSuffix))
n := copy(newCtx, prefix)
copy(newCtx[n:], normalizedSuffix)
return newCtx
}

func (l *logger) Trace(msg string, ctx ...interface{}) {
l.write(msg, LvlTrace, ctx)
}

func (l *logger) Debug(msg string, ctx ...interface{}) {
l.write(msg, LvlDebug, ctx)
}

func (l *logger) Info(msg string, ctx ...interface{}) {
l.write(msg, LvlInfo, ctx)
}

func (l *logger) Warn(msg string, ctx ...interface{}) {
l.write(msg, LvlWarn, ctx)
}

func (l *logger) Error(msg string, ctx ...interface{}) {
l.write(msg, LvlError, ctx)
}

func (l *logger) Crit(msg string, ctx ...interface{}) {
l.write(msg, LvlCrit, ctx)
os.Exit(1)
}
```

```
func (l *logger) GetHandler() Handler {
return l.h.Get()
}

func (l *logger) SetHandler(h Handler) {
l.h.Swap(h)
}

func normalize(ctx []interface{}) []interface{} {
// if the caller passed a Ctx object, then expand it
if len(ctx) == 1 {
if ctxMap, ok := ctx[0].(Ctx); ok {
ctx = ctxMap.toArray()
}
}

// ctx needs to be even because it's a series of key/value pairs
// no one wants to check for errors on logging functions,
// so instead of erroring on bad input, we'll just make sure
// that things are the right length and users can fix bugs
// when they see the output looks wrong
if len(ctx)%2 != 0 {
ctx = append(ctx, nil, errorKey, "Normalized odd number of arguments by adding nil")
}

return ctx
}

// Lazy allows you to defer calculation of a logged value that is expensive
// to compute until it is certain that it must be evaluated with the given filters.
//
// Lazy may also be used in conjunction with a Logger's New() function
// to generate a child logger which always reports the current value of changing
// state.
//
// You may wrap any function which takes no arguments to Lazy. It may return any
// number of values of any type.
type Lazy struct {
Fn interface{}
}

// Ctx is a map of key/value pairs to pass as context to a log function
```

```
// Use this only if you really need greater safety around the arguments you pass
// to the logging functions.
type Ctx map[string]interface{}

func (c Ctx) toArray() []interface{} {
arr := make([]interface{}, len(c)*2)

i := 0
for k, v := range c {
arr[i] = k
arr[i+1] = v
i += 2
}

return arr
}

26:F:\git\coin\ethereum\go-ethereum\log\root.go
}

// New returns a new logger with the given context.
// New is a convenient alias for Root().New
func New(ctx ...interface{}) Logger {
return root.New(ctx...)
}

// Root returns the root logger
func Root() Logger {
return root
}

// The following functions bypass the exported logger methods (logger.Debug,
// etc.) to keep the call depth the same for all paths to logger.write so
// runtime.Caller(2) always refers to the call site in client code.

// Trace is a convenient alias for Root().Trace
func Trace(msg string, ctx ...interface{}) {
root.write(msg, LvlTrace, ctx)
}

// Debug is a convenient alias for Root().Debug
func Debug(msg string, ctx ...interface{}) {
```

```
root.write(msg, LvlDebug, ctx)
}

// Info is a convenient alias for Root().Info
func Info(msg string, ctx ...interface{}) {
root.write(msg, LvlInfo, ctx)
}

// Warn is a convenient alias for Root().Warn
func Warn(msg string, ctx ...interface{}) {
root.write(msg, LvlWarn, ctx)
}

// Error is a convenient alias for Root().Error
func Error(msg string, ctx ...interface{}) {
root.write(msg, LvlError, ctx)
}

// Crit is a convenient alias for Root().Crit
func Crit(msg string, ctx ...interface{}) {
root.write(msg, LvlCrit, ctx)
os.Exit(1)
}

27:F:\git\coin\ethereum\go-ethereum\log\syslog.go
}

// SyslogNetHandler opens a connection to a log daemon over the network and writes
// all log records to it.
func SyslogNetHandler(net, addr string, priority syslog.Priority, tag string, fmtr Format) (Handler,
error) {
wr, err := syslog.Dial(net, addr, priority, tag)
return sharedSyslog(fmtr, wr, err)
}

func sharedSyslog(fmtr Format, sysWr *syslog.Writer, err error) (Handler, error) {
if err != nil {
return nil, err
}
h := FuncHandler(func(r *Record) error {
var syslogFn = sysWr.Info
switch r.Lvl {
```

```
case LvlCrit:
syslogFn = sysWr.Crit
case LvlError:
syslogFn = sysWr.Err
case LvlWarn:
syslogFn = sysWr.Warning
case LvlInfo:
syslogFn = sysWr.Info
case LvlDebug:
syslogFn = sysWr.Debug
case LvlTrace:
syslogFn = func(m string) error { return nil } // There's no syslog level for trace
}

s := strings.TrimSpace(string(fmtr.Format(r)))
return syslogFn(s)
})
return LazyHandler(&closingHandler{sysWr, h}), nil
}

func (m muster) SyslogHandler(priority syslog.Priority, tag string, fmtr Format) Handler {
return must(SyslogHandler(priority, tag, fmtr))
}

func (m muster) SyslogNetHandler(net, addr string, priority syslog.Priority, tag string, fmtr Format)
Handler {
return must(SyslogNetHandler(net, addr, priority, tag, fmtr))
}

28:F:\git\coin\ethereum\go-ethereum\log\term\terminal_appengine.go

29:F:\git\coin\ethereum\go-ethereum\log\term\terminal_darwin.go

30:F:\git\coin\ethereum\go-ethereum\log\term\terminal_freebsd.go
Cc     [20]uint8
Ispeed uint32
Ospeed uint32
}

31:F:\git\coin\ethereum\go-ethereum\log\term\terminal_linux.go

32:F:\git\coin\ethereum\go-ethereum\log\term\terminal_netbsd.go
```

33:F:\git\coin\ethereum\go-ethereum\log\term\terminal_notwindows.go
// IsTty returns true if the given file descriptor is a terminal.
func IsTty(fd uintptr) bool {
var termios Termios
_, _, err := syscall.Syscall6(syscall.SYS_IOCTL, fd, ioctlReadTermios,
uintptr(unsafe.Pointer(&termios)), 0, 0, 0)
return err == 0
}

34:F:\git\coin\ethereum\go-ethereum\log\term\terminal_openbsd.go

35:F:\git\coin\ethereum\go-ethereum\log\term\terminal_solaris.go

36:F:\git\coin\ethereum\go-ethereum\log\term\terminal_windows.go
var kernel32 = syscall.NewLazyDLL("kernel32.dll")

var (
procGetConsoleMode = kernel32.NewProc("GetConsoleMode")
)

// IsTty returns true if the given file descriptor is a terminal.
func IsTty(fd uintptr) bool {
var st uint32
r, _, e := syscall.Syscall(procGetConsoleMode.Addr(), 2, fd, uintptr(unsafe.Pointer(&st)), 0)
return r != 0 && e == 0
}

37:F:\git\coin\ethereum\go-ethereum\metrics\disk.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package metrics

// DiskStats is the per process disk io stats.
type DiskStats struct {
ReadCount int64 // Number of read operations executed
ReadBytes int64 // Total number of bytes read
WriteCount int64 // Number of write operations executed
WriteBytes int64 // Total number of byte written
}

38:F:\git\coin\ethereum\go-ethereum\metrics\disk_linux.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains the Linux implementation of process disk IO counter retrieval.

package metrics

import (
"bufio"
"fmt"
"io"
"os"
"strconv"
"strings"
)

// ReadDiskStats retrieves the disk IO stats belonging to the current process.
func ReadDiskStats(stats *DiskStats) error {
// Open the process disk IO counter file
inf, err := os.Open(fmt.Sprintf("/proc/%d/io", os.Getpid()))
if err != nil {
return err
}
defer inf.Close()
in := bufio.NewReader(inf)

// Iterate over the IO counter, and extract what we need
for {
// Read the next line and split to key and value
line, err := in.ReadString('\n')
if err != nil {
if err == io.EOF {
return nil
}
return err
}
parts := strings.Split(line, ":")
if len(parts) != 2 {
continue
}
key := strings.TrimSpace(parts[0])
value, err := strconv.ParseInt(strings.TrimSpace(parts[1]), 10, 64)
if err != nil {
```

```
return err
}

// Update the counter based on the key
switch key {
case "syscr":
stats.ReadCount = value
case "syscw":
stats.WriteCount = value
case "rchar":
stats.ReadBytes = value
case "wchar":
stats.WriteBytes = value
}
}
}
```

39:F:\git\coin\ethereum\go-ethereum\metrics\disk_nop.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// +build !linux

package metrics

import "errors"

// ReadDiskStats retrieves the disk IO stats belonging to the current process.
func ReadDiskStats(stats *DiskStats) error {
return errors.New("Not implemented")
}
```

40:F:\git\coin\ethereum\go-ethereum\metrics\metrics.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package metrics provides general system and process level metrics collection.
package metrics

import (
"os"
"runtime"
"strings"
"time"
```

```
"github.com/ethereum/go-ethereum/log"
"github.com/rcrowley/go-metrics"
"github.com/rcrowley/go-metrics/exp"
)

// MetricsEnabledFlag is the CLI flag name to use to enable metrics collections.
var MetricsEnabledFlag = "metrics"

// Enabled is the flag specifying if metrics are enable or not.
var Enabled = false

// Init enables or disables the metrics system. Since we need this to run before
// any other code gets to create meters and timers, we'll actually do an ugly hack
// and peek into the command line args for the metrics flag.
func init() {
for _, arg := range os.Args {
if strings.TrimLeft(arg, "-") == MetricsEnabledFlag {
log.Info("Enabling metrics collection")
Enabled = true
}
}
exp.Exp(metrics.DefaultRegistry)
}

// NewCounter create a new metrics Counter, either a real one of a NOP stub depending
// on the metrics flag.
func NewCounter(name string) metrics.Counter {
if !Enabled {
return new(metrics.NilCounter)
}
return metrics.GetOrRegisterCounter(name, metrics.DefaultRegistry)
}

// NewMeter create a new metrics Meter, either a real one of a NOP stub depending
// on the metrics flag.
func NewMeter(name string) metrics.Meter {
if !Enabled {
return new(metrics.NilMeter)
}
return metrics.GetOrRegisterMeter(name, metrics.DefaultRegistry)
}
```

```
// NewTimer create a new metrics Timer, either a real one of a NOP stub depending
// on the metrics flag.
func NewTimer(name string) metrics.Timer {
if !Enabled {
return new(metrics.NilTimer)
}
return metrics.GetOrRegisterTimer(name, metrics.DefaultRegistry)
}

// CollectProcessMetrics periodically collects various metrics about the running
// process.
func CollectProcessMetrics(refresh time.Duration) {
// Short circuit if the metrics system is disabled
if !Enabled {
return
}
// Create the various data collectors
memstats := make([]*runtime.MemStats, 2)
diskstats := make([]*DiskStats, 2)
for i := 0; i < len(memstats); i++ {
memstats[i] = new(runtime.MemStats)
diskstats[i] = new(DiskStats)
}
// Define the various metrics to collect
memAllocs := metrics.GetOrRegisterMeter("system/memory/allocs", metrics.DefaultRegistry)
memFrees := metrics.GetOrRegisterMeter("system/memory/frees", metrics.DefaultRegistry)
memInuse := metrics.GetOrRegisterMeter("system/memory/inuse", metrics.DefaultRegistry)
memPauses := metrics.GetOrRegisterMeter("system/memory/pauses", metrics.DefaultRegistry)

var diskReads, diskReadBytes, diskWrites, diskWriteBytes metrics.Meter
if err := ReadDiskStats(diskstats[0]); err == nil {
diskReads = metrics.GetOrRegisterMeter("system/disk/readcount", metrics.DefaultRegistry)
diskReadBytes = metrics.GetOrRegisterMeter("system/disk/readdata", metrics.DefaultRegistry)
diskWrites = metrics.GetOrRegisterMeter("system/disk/writecount", metrics.DefaultRegistry)
diskWriteBytes = metrics.GetOrRegisterMeter("system/disk/writedata", metrics.DefaultRegistry)
} else {
log.Debug("Failed to read disk metrics", "err", err)
}
// Iterate loading the different stats and updating the meters
for i := 1; ; i++ {
runtime.ReadMemStats(memstats[i%2])
```

```
memAllocs.Mark(int64(memstats[i%2].Mallocs - memstats[(i-1)%2].Mallocs))
memFrees.Mark(int64(memstats[i%2].Frees - memstats[(i-1)%2].Frees))
memInuse.Mark(int64(memstats[i%2].Alloc - memstats[(i-1)%2].Alloc))
memPauses.Mark(int64(memstats[i%2].PauseTotalNs - memstats[(i-1)%2].PauseTotalNs))

if ReadDiskStats(diskstats[i%2]) == nil {
diskReads.Mark(int64(diskstats[i%2].ReadCount - diskstats[(i-1)%2].ReadCount))
diskReadBytes.Mark(int64(diskstats[i%2].ReadBytes - diskstats[(i-1)%2].ReadBytes))
diskWrites.Mark(int64(diskstats[i%2].WriteCount - diskstats[(i-1)%2].WriteCount))
diskWriteBytes.Mark(int64(diskstats[i%2].WriteBytes - diskstats[(i-1)%2].WriteBytes))
}
time.Sleep(refresh)
}
}
```

41:F:\git\coin\ethereum\go-ethereum\miner\agent.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package miner

import (
"sync"

"sync/atomic"

"github.com/ethereum/go-ethereum/consensus"
"github.com/ethereum/go-ethereum/log"
)

type CpuAgent struct {
mu sync.Mutex

workCh        chan *Work
stop          chan struct{}
quitCurrentOp chan struct{}
returnCh      chan<- *Result

chain  consensus.ChainReader
engine consensus.Engine

isMining int32 // isMining indicates whether the agent is currently mining
}
```

```
func NewCpuAgent(chain consensus.ChainReader, engine consensus.Engine) *CpuAgent {
miner := &CpuAgent{
chain:  chain,
engine: engine,
stop:   make(chan struct{}, 1),
workCh: make(chan *Work, 1),
}
return miner
}

func (self *CpuAgent) Work() chan<- *Work            { return self.workCh }
func (self *CpuAgent) SetReturnCh(ch chan<- *Result) { self.returnCh = ch }

func (self *CpuAgent) Stop() {
self.stop <- struct{}{}
}

func (self *CpuAgent) Start() {
if !atomic.CompareAndSwapInt32(&self.isMining, 0, 1) {
return // agent already started
}
go self.update()
}

func (self *CpuAgent) update() {
out:
for {
select {
case work := <-self.workCh:
self.mu.Lock()
if self.quitCurrentOp != nil {
close(self.quitCurrentOp)
}
self.quitCurrentOp = make(chan struct{})
go self.mine(work, self.quitCurrentOp)
self.mu.Unlock()
case <-self.stop:
self.mu.Lock()
if self.quitCurrentOp != nil {
close(self.quitCurrentOp)
self.quitCurrentOp = nil
```

```
}
self.mu.Unlock()
break out
}
}

done:
// Empty work channel
for {
select {
case <-self.workCh:
default:
break done
}
}
atomic.StoreInt32(&self.isMining, 0)
}

func (self *CpuAgent) mine(work *Work, stop <-chan struct{}) {
if result, err := self.engine.Seal(self.chain, work.Block, stop); result != nil {
log.Info("Successfully sealed new block", "number", result.Number(), "hash", result.Hash())
self.returnCh <- &Result{work, result}
} else {
if err != nil {
log.Warn("Block sealing failed", "err", err)
}
self.returnCh <- nil
}
}

func (self *CpuAgent) GetHashRate() int64 {
if pow, ok := self.engine.(consensus.PoW); ok {
return int64(pow.Hashrate())
}
return 0
}

42:F:\git\coin\ethereum\go-ethereum\miner\miner.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package miner implements Ethereum block creation and mining.
package miner
```

```
import (
"fmt"
"sync/atomic"

"github.com/ethereum/go-ethereum/accounts"
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/consensus"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/eth/downloader"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/params"
)

// Backend wraps all methods required for mining.
type Backend interface {
AccountManager() *accounts.Manager
BlockChain() *core.BlockChain
TxPool() *core.TxPool
ChainDb() ethdb.Database
}

// Miner creates blocks and searches for proof-of-work values.
type Miner struct {
mux *event.TypeMux

worker *worker

coinbase common.Address
mining   int32
eth      Backend
engine   consensus.Engine

canStart    int32 // can start indicates whether we can start the mining operation
shouldStart int32 // should start indicates whether we should start after sync
}

func New(eth Backend, config *params.ChainConfig, mux *event.TypeMux, engine
```

```
consensus.Engine) *Miner {
miner := &Miner{
eth:      eth,
mux:      mux,
engine:   engine,
worker:   newWorker(config, engine, common.Address{}, eth, mux),
canStart: 1,
}
miner.Register(NewCpuAgent(eth.BlockChain(), engine))
go miner.update()

return miner
}

// update keeps track of the downloader events. Please be aware that this is a one shot type of
update loop.
// It's entered once and as soon as `Done` or `Failed` has been broadcasted the events are
unregistered and
// the loop is exited. This to prevent a major security vuln where external parties can DOS you with
blocks
// and halt your mining operation for as long as the DOS continues.
func (self *Miner) update() {
events := self.mux.Subscribe(downloader.StartEvent{}, downloader.DoneEvent{},
downloader.FailedEvent{})
out:
for ev := range events.Chan() {
switch ev.Data.(type) {
case downloader.StartEvent:
atomic.StoreInt32(&self.canStart, 0)
if self.Mining() {
self.Stop()
atomic.StoreInt32(&self.shouldStart, 1)
log.Info("Mining aborted due to sync")
}
case downloader.DoneEvent, downloader.FailedEvent:
shouldStart := atomic.LoadInt32(&self.shouldStart) == 1

atomic.StoreInt32(&self.canStart, 1)
atomic.StoreInt32(&self.shouldStart, 0)
if shouldStart {
self.Start(self.coinbase)
}
```

```
// unsubscribe. we're only interested in this event once
events.Unsubscribe()
// stop immediately and ignore all further pending events
break out
}
}
}

func (self *Miner) Start(coinbase common.Address) {
atomic.StoreInt32(&self.shouldStart, 1)
self.worker.setEtherbase(coinbase)
self.coinbase = coinbase

if atomic.LoadInt32(&self.canStart) == 0 {
log.Info("Network syncing, will start miner afterwards")
return
}
atomic.StoreInt32(&self.mining, 1)

log.Info("Starting mining operation")
self.worker.start()
self.worker.commitNewWork()
}

func (self *Miner) Stop() {
self.worker.stop()
atomic.StoreInt32(&self.mining, 0)
atomic.StoreInt32(&self.shouldStart, 0)
}

func (self *Miner) Register(agent Agent) {
if self.Mining() {
agent.Start()
}
self.worker.register(agent)
}

func (self *Miner) Unregister(agent Agent) {
self.worker.unregister(agent)
}

func (self *Miner) Mining() bool {
```

```
return atomic.LoadInt32(&self.mining) > 0
}

func (self *Miner) HashRate() (tot int64) {
if pow, ok := self.engine.(consensus.PoW); ok {
tot += int64(pow.Hashrate())
}
// do we care this might race? is it worth we're rewriting some
// aspects of the worker/locking up agents so we can get an accurate
// hashrate?
for agent := range self.worker.agents {
if _, ok := agent.(*CpuAgent); !ok {
tot += agent.GetHashRate()
}
}
return
}

func (self *Miner) SetExtra(extra []byte) error {
if uint64(len(extra)) > params.MaximumExtraDataSize {
return fmt.Errorf("Extra exceeds max length. %d > %v", len(extra),
params.MaximumExtraDataSize)
}
self.worker.setExtra(extra)
return nil
}

// Pending returns the currently pending block and associated state.
func (self *Miner) Pending() (*types.Block, *state.StateDB) {
return self.worker.pending()
}

// PendingBlock returns the currently pending block.
//
// Note, to access both the pending block and the pending state
// simultaneously, please use Pending(), as the pending state can
// change between multiple method calls
func (self *Miner) PendingBlock() *types.Block {
return self.worker.pendingBlock()
}

func (self *Miner) SetEtherbase(addr common.Address) {
```

```
self.coinbase = addr
self.worker.setEtherbase(addr)
}

43:F:\git\coin\ethereum\go-ethereum\miner\remote_agent.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package miner

import (
"errors"
"math/big"
"sync"
"sync/atomic"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/consensus"
"github.com/ethereum/go-ethereum/consensus/ethash"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/log"
)

type hashrate struct {
ping time.Time
rate uint64
}

type RemoteAgent struct {
mu sync.Mutex

quitCh   chan struct{}
workCh   chan *Work
returnCh chan<- *Result

chain       consensus.ChainReader
engine      consensus.Engine
currentWork *Work
work        map[common.Hash]*Work

hashrateMu sync.RWMutex
hashrate   map[common.Hash]hashrate
```

```
running int32 // running indicates whether the agent is active. Call atomically
}

func NewRemoteAgent(chain consensus.ChainReader, engine consensus.Engine) *RemoteAgent
{
return &RemoteAgent{
chain:    chain,
engine:   engine,
work:     make(map[common.Hash]*Work),
hashrate: make(map[common.Hash]hashrate),
}
}

func (a *RemoteAgent) SubmitHashrate(id common.Hash, rate uint64) {
a.hashrateMu.Lock()
defer a.hashrateMu.Unlock()

a.hashrate[id] = hashrate{time.Now(), rate}
}

func (a *RemoteAgent) Work() chan<- *Work {
return a.workCh
}

func (a *RemoteAgent) SetReturnCh(returnCh chan<- *Result) {
a.returnCh = returnCh
}

func (a *RemoteAgent) Start() {
if !atomic.CompareAndSwapInt32(&a.running, 0, 1) {
return
}
a.quitCh = make(chan struct{})
a.workCh = make(chan *Work, 1)
go a.loop(a.workCh, a.quitCh)
}

func (a *RemoteAgent) Stop() {
if !atomic.CompareAndSwapInt32(&a.running, 1, 0) {
return
}
```

```
close(a.quitCh)
close(a.workCh)
}

// GetHashRate returns the accumulated hashrate of all identifier combined
func (a *RemoteAgent) GetHashRate() (tot int64) {
a.hashrateMu.RLock()
defer a.hashrateMu.RUnlock()

// this could overflow
for _, hashrate := range a.hashrate {
tot += int64(hashrate.rate)
}
return
}

func (a *RemoteAgent) GetWork() ([3]string, error) {
a.mu.Lock()
defer a.mu.Unlock()

var res [3]string

if a.currentWork != nil {
block := a.currentWork.Block

res[0] = block.HashNoNonce().Hex()
seedHash := ethash.SeedHash(block.NumberU64())
res[1] = common.BytesToHash(seedHash).Hex()
// Calculate the "target" to be returned to the external miner
n := big.NewInt(1)
n.Lsh(n, 255)
n.Div(n, block.Difficulty())
n.Lsh(n, 1)
res[2] = common.BytesToHash(n.Bytes()).Hex()

a.work[block.HashNoNonce()] = a.currentWork
return res, nil
}
return res, errors.New("No work available yet, don't panic.")
}

// SubmitWork tries to inject a pow solution into the remote agent, returning
```

```
// whether the solution was accepted or not (not can be both a bad pow as well as
// any other error, like no work pending).
func (a *RemoteAgent) SubmitWork(nonce types.BlockNonce, mixDigest, hash common.Hash)
bool {
a.mu.Lock()
defer a.mu.Unlock()

// Make sure the work submitted is present
work := a.work[hash]
if work == nil {
log.Info("Work submitted but none pending", "hash", hash)
return false
}
// Make sure the Engine solutions is indeed valid
result := work.Block.Header()
result.Nonce = nonce
result.MixDigest = mixDigest

if err := a.engine.VerifySeal(a.chain, result); err != nil {
log.Warn("Invalid proof-of-work submitted", "hash", hash, "err", err)
return false
}
block := work.Block.WithSeal(result)

// Solutions seems to be valid, return to the miner and notify acceptance
a.returnCh <- &Result{work, block}
delete(a.work, hash)

return true
}

// loop monitors mining events on the work and quit channels, updating the internal
// state of the rmeote miner until a termination is requested.
//
// Note, the reason the work and quit channels are passed as parameters is because
// RemoteAgent.Start() constantly recreates these channels, so the loop code cannot
// assume data stability in these member fields.
func (a *RemoteAgent) loop(workCh chan *Work, quitCh chan struct{}) {
ticker := time.Tick(5 * time.Second)

for {
select {
```

```
case <-quitCh:
return
case work := <-workCh:
a.mu.Lock()
a.currentWork = work
a.mu.Unlock()
case <-ticker:
// cleanup
a.mu.Lock()
for hash, work := range a.work {
if time.Since(work.createdAt) > 7*(12*time.Second) {
delete(a.work, hash)
}
}
a.mu.Unlock()

a.hashrateMu.Lock()
for id, hashrate := range a.hashrate {
if time.Since(hashrate.ping) > 10*time.Second {
delete(a.hashrate, id)
}
}
a.hashrateMu.Unlock()
}
}
}
```

44:F:\git\coin\ethereum\go-ethereum\miner\unconfirmed.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package miner

import (
"container/ring"
"sync"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/log"
)

// headerRetriever is used by the unconfirmed block set to verify whether a previously
```

```
// mined block is part of the canonical chain or not.
type headerRetriever interface {
// GetHeaderByNumber retrieves the canonical header associated with a block number.
GetHeaderByNumber(number uint64) *types.Header
}

// unconfirmedBlock is a small collection of metadata about a locally mined block
// that is placed into a unconfirmed set for canonical chain inclusion tracking.
type unconfirmedBlock struct {
index uint64
hash  common.Hash
}

// unconfirmedBlocks implements a data structure to maintain locally mined blocks
// have have not yet reached enough maturity to guarantee chain inclusion. It is
// used by the miner to provide logs to the user when a previously mined block
// has a high enough guarantee to not be reorged out of te canonical chain.
type unconfirmedBlocks struct {
chain  headerRetriever // Blockchain to verify canonical status through
depth  uint            // Depth after which to discard previous blocks
blocks *ring.Ring      // Block infos to allow canonical chain cross checks
lock   sync.RWMutex    // Protects the fields from concurrent access
}

// newUnconfirmedBlocks returns new data structure to track currently unconfirmed blocks.
func newUnconfirmedBlocks(chain headerRetriever, depth uint) *unconfirmedBlocks {
return &unconfirmedBlocks{
chain: chain,
depth: depth,
}
}

// Insert adds a new block to the set of unconfirmed ones.
func (set *unconfirmedBlocks) Insert(index uint64, hash common.Hash) {
// If a new block was mined locally, shift out any old enough blocks
set.Shift(index)

// Create the new item as its own ring
item := ring.New(1)
item.Value = &unconfirmedBlock{
index: index,
hash:  hash,
```

```
}
// Set as the initial ring or append to the end
set.lock.Lock()
defer set.lock.Unlock()

if set.blocks == nil {
set.blocks = item
} else {
set.blocks.Move(-1).Link(item)
}
// Display a log for the user to notify of a new mined block unconfirmed
log.Info(" mined potential block", "number", index, "hash", hash)
}

// Shift drops all unconfirmed blocks from the set which exceed the unconfirmed sets depth
// allowance, checking them against the canonical chain for inclusion or staleness
// report.
func (set *unconfirmedBlocks) Shift(height uint64) {
set.lock.Lock()
defer set.lock.Unlock()

for set.blocks != nil {
// Retrieve the next unconfirmed block and abort if too fresh
next := set.blocks.Value.(*unconfirmedBlock)
if next.index+uint64(set.depth) > height {
break
}
// Block seems to exceed depth allowance, check for canonical status
header := set.chain.GetHeaderByNumber(next.index)
switch {
case header == nil:
log.Warn("Failed to retrieve header of mined block", "number", next.index, "hash", next.hash)
case header.Hash() == next.hash:
log.Info(" block reached canonical chain", "number", next.index, "hash", next.hash)
default:
log.Info(" block  became a side fork", "number", next.index, "hash", next.hash)
}
// Drop the block out of the ring
if set.blocks.Value == set.blocks.Next().Value {
set.blocks = nil
} else {
set.blocks = set.blocks.Move(-1)
```

```
set.blocks.Unlink(1)
set.blocks = set.blocks.Move(1)
}
}
}
```

45:F:\git\coin\ethereum\go-ethereum\miner\unconfirmed_test.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package miner

import (
"testing"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/types"
)

// noopHeaderRetriever is an implementation of headerRetriever that always
// returns nil for any requested headers.
type noopHeaderRetriever struct{}

func (r *noopHeaderRetriever) GetHeaderByNumber(number uint64) *types.Header {
return nil
}

// Tests that inserting blocks into the unconfirmed set accumulates them until
// the desired depth is reached, after which they begin to be dropped.
func TestUnconfirmedInsertBounds(t *testing.T) {
limit := uint(10)

pool := newUnconfirmedBlocks(new(noopHeaderRetriever), limit)
for depth := uint64(0); depth < 2*uint64(limit); depth++ {
// Insert multiple blocks for the same level just to stress it
for i := 0; i < int(depth); i++ {
pool.Insert(depth, common.Hash([32]byte{byte(depth), byte(i)}))
}
// Validate that no blocks below the depth allowance are left in
pool.blocks.Do(func(block interface{}) {
if block := block.(*unconfirmedBlock); block.index+uint64(limit) <= depth {
t.Errorf("depth %d: block %x not dropped", depth, block.hash)
}
```

```
})
}
}

// Tests that shifting blocks out of the unconfirmed set works both for normal
// cases as well as for corner cases such as empty sets, empty shifts or full
// shifts.
func TestUnconfirmedShifts(t *testing.T) {
// Create a pool with a few blocks on various depths
limit, start := uint(10), uint64(25)

pool := newUnconfirmedBlocks(new(noopHeaderRetriever), limit)
for depth := start; depth < start+uint64(limit); depth++ {
pool.Insert(depth, common.Hash([32]byte{byte(depth)}))
}
// Try to shift below the limit and ensure no blocks are dropped
pool.Shift(start + uint64(limit) - 1)
if n := pool.blocks.Len(); n != int(limit) {
t.Errorf("unconfirmed count mismatch: have %d, want %d", n, limit)
}
// Try to shift half the blocks out and verify remainder
pool.Shift(start + uint64(limit) - 1 + uint64(limit/2))
if n := pool.blocks.Len(); n != int(limit)/2 {
t.Errorf("unconfirmed count mismatch: have %d, want %d", n, limit/2)
}
// Try to shift all the remaining blocks out and verify emptyness
pool.Shift(start + 2*uint64(limit))
if n := pool.blocks.Len(); n != 0 {
t.Errorf("unconfirmed count mismatch: have %d, want %d", n, 0)
}
// Try to shift out from the empty set and make sure it doesn't break
pool.Shift(start + 3*uint64(limit))
if n := pool.blocks.Len(); n != 0 {
t.Errorf("unconfirmed count mismatch: have %d, want %d", n, 0)
}
}

46:F:\git\coin\ethereum\go-ethereum\miner\worker.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package miner
```

```
import (
"bytes"
"fmt"
"math/big"
"sync"
"sync/atomic"
"time"

"github.com/ethereum/go-ethereum/accounts"
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/consensus"
"github.com/ethereum/go-ethereum/consensus/misc"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/core/vm"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/params"
"gopkg.in/fatih/set.v0"
)

const (
resultQueueSize  = 10
miningLogAtDepth = 5
)

// Agent can register themself with the worker
type Agent interface {
Work() chan<- *Work
SetReturnCh(chan<- *Result)
Stop()
Start()
GetHashRate() int64
}

// Work is the workers current environment and holds
// all of the current state information
type Work struct {
config *params.ChainConfig
signer types.Signer
```

```
state     *state.StateDB // apply state changes here
ancestors *set.Set       // ancestor set (used for checking uncle parent validity)
family    *set.Set       // family set (used for checking uncle invalidity)
uncles    *set.Set       // uncle set
tcount    int            // tx count in cycle
failedTxs types.Transactions

Block *types.Block // the new block

header   *types.Header
txs      []*types.Transaction
receipts []*types.Receipt

createdAt time.Time
}

type Result struct {
Work  *Work
Block *types.Block
}

// worker is the main object which takes care of applying messages to the new state
type worker struct {
config *params.ChainConfig
engine consensus.Engine

mu sync.Mutex

// update loop
mux    *event.TypeMux
events *event.TypeMuxSubscription
wg     sync.WaitGroup

agents map[Agent]struct{}
recv   chan *Result

eth     Backend
chain   *core.BlockChain
proc    core.Validator
chainDb ethdb.Database
```

```
coinbase common.Address
extra    []byte

currentMu sync.Mutex
current   *Work

uncleMu        sync.Mutex
possibleUncles map[common.Hash]*types.Block

txQueueMu sync.Mutex
txQueue   map[common.Hash]*types.Transaction

unconfirmed *unconfirmedBlocks // set of locally mined blocks pending canonicalness
confirmations

// atomic status counters
mining int32
atWork int32

fullValidation bool
}

func newWorker(config *params.ChainConfig, engine consensus.Engine, coinbase
common.Address, eth Backend, mux *event.TypeMux) *worker {
worker := &worker{
config:         config,
engine:         engine,
eth:            eth,
mux:            mux,
chainDb:        eth.ChainDb(),
recv:           make(chan *Result, resultQueueSize),
chain:          eth.BlockChain(),
proc:           eth.BlockChain().Validator(),
possibleUncles: make(map[common.Hash]*types.Block),
coinbase:       coinbase,
txQueue:        make(map[common.Hash]*types.Transaction),
agents:         make(map[Agent]struct{}),
unconfirmed:    newUnconfirmedBlocks(eth.BlockChain(), 5),
fullValidation: false,
}
worker.events = worker.mux.Subscribe(core.ChainHeadEvent{}, core.ChainSideEvent{},
core.TxPreEvent{})
```

```
go worker.update()

go worker.wait()
worker.commitNewWork()

return worker
}

func (self *worker) setEtherbase(addr common.Address) {
self.mu.Lock()
defer self.mu.Unlock()
self.coinbase = addr
}

func (self *worker) setExtra(extra []byte) {
self.mu.Lock()
defer self.mu.Unlock()
self.extra = extra
}

func (self *worker) pending() (*types.Block, *state.StateDB) {
self.currentMu.Lock()
defer self.currentMu.Unlock()

if atomic.LoadInt32(&self.mining) == 0 {
return types.NewBlock(
self.current.header,
self.current.txs,
nil,
self.current.receipts,
), self.current.state.Copy()
}
return self.current.Block, self.current.state.Copy()
}

func (self *worker) pendingBlock() *types.Block {
self.currentMu.Lock()
defer self.currentMu.Unlock()

if atomic.LoadInt32(&self.mining) == 0 {
return types.NewBlock(
self.current.header,
```

```
self.current.txs,
nil,
self.current.receipts,
)
}
return self.current.Block
}

func (self *worker) start() {
self.mu.Lock()
defer self.mu.Unlock()

atomic.StoreInt32(&self.mining, 1)

// spin up agents
for agent := range self.agents {
agent.Start()
}
}

func (self *worker) stop() {
self.wg.Wait()

self.mu.Lock()
defer self.mu.Unlock()
if atomic.LoadInt32(&self.mining) == 1 {
for agent := range self.agents {
agent.Stop()
}
}
atomic.StoreInt32(&self.mining, 0)
atomic.StoreInt32(&self.atWork, 0)
}

func (self *worker) register(agent Agent) {
self.mu.Lock()
defer self.mu.Unlock()
self.agents[agent] = struct{}{}
agent.SetReturnCh(self.recv)
}

func (self *worker) unregister(agent Agent) {
```

```
self.mu.Lock()
defer self.mu.Unlock()
delete(self.agents, agent)
agent.Stop()
}

func (self *worker) update() {
for event := range self.events.Chan() {
// A real event arrived, process interesting content
switch ev := event.Data.(type) {
case core.ChainHeadEvent:
self.commitNewWork()
case core.ChainSideEvent:
self.uncleMu.Lock()
self.possibleUncles[ev.Block.Hash()] = ev.Block
self.uncleMu.Unlock()
case core.TxPreEvent:
// Apply transaction to the pending state if we're not mining
if atomic.LoadInt32(&self.mining) == 0 {
self.currentMu.Lock()

acc, _ := types.Sender(self.current.signer, ev.Tx)
txs := map[common.Address]types.Transactions{acc: {ev.Tx}}
txset := types.NewTransactionsByPriceAndNonce(txs)

self.current.commitTransactions(self.mux, txset, self.chain, self.coinbase)
self.currentMu.Unlock()
}
}
}
}

func (self *worker) wait() {
for {
mustCommitNewWork := true
for result := range self.recv {
atomic.AddInt32(&self.atWork, -1)

if result == nil {
continue
}
block := result.Block
```

```
work := result.Work

if self.fullValidation {
if _, err := self.chain.InsertChain(types.Blocks{block}); err != nil {
log.Error("Mined invalid block", "err", err)
continue
}
go self.mux.Post(core.NewMinedBlockEvent{Block: block})
} else {
work.state.CommitTo(self.chainDb, self.config.IsEIP158(block.Number()))
stat, err := self.chain.WriteBlock(block)
if err != nil {
log.Error("Failed writing block to chain", "err", err)
continue
}
// update block hash since it is now available and not when the receipt/log of individual
transactions were created
for _, r := range work.receipts {
for _, l := range r.Logs {
l.BlockHash = block.Hash()
}
}
for _, log := range work.state.Logs() {
log.BlockHash = block.Hash()
}

// check if canon block and write transactions
if stat == core.CanonStatTy {
// This puts transactions in a extra db for rpc
core.WriteTransactions(self.chainDb, block)
// store the receipts
core.WriteReceipts(self.chainDb, work.receipts)
// Write map map bloom filters
core.WriteMipmapBloom(self.chainDb, block.NumberU64(), work.receipts)
// implicit by posting ChainHeadEvent
mustCommitNewWork = false
}

// broadcast before waiting for validation
go func(block *types.Block, logs []*types.Log, receipts []*types.Receipt) {
self.mux.Post(core.NewMinedBlockEvent{Block: block})
self.mux.Post(core.ChainEvent{Block: block, Hash: block.Hash(), Logs: logs})
```

```
if stat == core.CanonStatTy {
self.mux.Post(core.ChainHeadEvent{Block: block})
self.mux.Post(logs)
}
if err := core.WriteBlockReceipts(self.chainDb, block.Hash(), block.NumberU64(), receipts); err !=
nil {
log.Warn("Failed writing block receipts", "err", err)
}
}(block, work.state.Logs(), work.receipts)
}
// Insert the block into the set of pending ones to wait for confirmations
self.unconfirmed.Insert(block.NumberU64(), block.Hash())

if mustCommitNewWork {
self.commitNewWork()
}
}
}
}

// push sends a new work task to currently live miner agents.
func (self *worker) push(work *Work) {
if atomic.LoadInt32(&self.mining) != 1 {
return
}
for agent := range self.agents {
atomic.AddInt32(&self.atWork, 1)
if ch := agent.Work(); ch != nil {
ch <- work
}
}
}

// makeCurrent creates a new environment for the current cycle.
func (self *worker) makeCurrent(parent *types.Block, header *types.Header) error {
state, err := self.chain.StateAt(parent.Root())
if err != nil {
return err
}
work := &Work{
config:    self.config,
```

```
signer:    types.NewEIP155Signer(self.config.ChainId),
state:     state,
ancestors: set.New(),
family:    set.New(),
uncles:    set.New(),
header:    header,
createdAt: time.Now(),
}

// when 08 is processed ancestors contain 07 (quick block)
for _, ancestor := range self.chain.GetBlocksFromHash(parent.Hash(), 7) {
for _, uncle := range ancestor.Uncles() {
work.family.Add(uncle.Hash())
}
work.family.Add(ancestor.Hash())
work.ancestors.Add(ancestor.Hash())
}
wallets := self.eth.AccountManager().Wallets()
accounts := make([]accounts.Account, 0, len(wallets))
for _, wallet := range wallets {
accounts = append(accounts, wallet.Accounts()...)
}
// Keep track of transactions which return errors so they can be removed
work.tcount = 0
self.current = work
return nil
}

func (self *worker) commitNewWork() {
self.mu.Lock()
defer self.mu.Unlock()
self.uncleMu.Lock()
defer self.uncleMu.Unlock()
self.currentMu.Lock()
defer self.currentMu.Unlock()

tstart := time.Now()
parent := self.chain.CurrentBlock()

tstamp := tstart.Unix()
if parent.Time().Cmp(new(big.Int).SetInt64(tstamp)) >= 0 {
tstamp = parent.Time().Int64() + 1
```

```
}
// this will ensure we're not going off too far in the future
if now := time.Now().Unix(); tstamp > now+1 {
wait := time.Duration(tstamp-now) * time.Second
log.Info("Mining too far in the future", "wait", common.PrettyDuration(wait))
time.Sleep(wait)
}

num := parent.Number()
header := &types.Header{
ParentHash: parent.Hash(),
Number:     num.Add(num, common.Big1),
GasLimit:   core.CalcGasLimit(parent),
GasUsed:    new(big.Int),
Extra:      self.extra,
Time:       big.NewInt(tstamp),
}
// Only set the coinbase if we are mining (avoid spurious block rewards)
if atomic.LoadInt32(&self.mining) == 1 {
header.Coinbase = self.coinbase
}
if err := self.engine.Prepare(self.chain, header); err != nil {
log.Error("Failed to prepare header for mining", "err", err)
return
}
// If we are care about TheDAO hard-fork check whether to override the extra-data or not
if daoBlock := self.config.DAOForkBlock; daoBlock != nil {
// Check whether the block is among the fork extra-override range
limit := new(big.Int).Add(daoBlock, params.DAOForkExtraRange)
if header.Number.Cmp(daoBlock) >= 0 && header.Number.Cmp(limit) < 0 {
// Depending whether we support or oppose the fork, override differently
if self.config.DAOForkSupport {
header.Extra = common.CopyBytes(params.DAOForkBlockExtra)
} else if bytes.Equal(header.Extra, params.DAOForkBlockExtra) {
header.Extra = []byte{} // If miner opposes, don't let it use the reserved extra-data
}
}
}
// Could potentially happen if starting to mine in an odd state.
err := self.makeCurrent(parent, header)
if err != nil {
log.Error("Failed to create mining context", "err", err)
```

```
return
}
// Create the current work task and check any fork transitions needed
work := self.current
if self.config.DAOForkSupport && self.config.DAOForkBlock != nil &&
self.config.DAOForkBlock.Cmp(header.Number) == 0 {
misc.ApplyDAOHardFork(work.state)
}
pending, err := self.eth.TxPool().Pending()
if err != nil {
log.Error("Failed to fetch pending transactions", "err", err)
return
}
txs := types.NewTransactionsByPriceAndNonce(pending)
work.commitTransactions(self.mux, txs, self.chain, self.coinbase)

self.eth.TxPool().RemoveBatch(work.failedTxs)

// compute uncles for the new block.
var (
uncles    []*types.Header
badUncles []common.Hash
)
for hash, uncle := range self.possibleUncles {
if len(uncles) == 2 {
break
}
if err := self.commitUncle(work, uncle.Header()); err != nil {
log.Trace("Bad uncle found and will be removed", "hash", hash)
log.Trace(fmt.Sprint(uncle))

badUncles = append(badUncles, hash)
} else {
log.Debug("Committing new uncle to block", "hash", hash)
uncles = append(uncles, uncle.Header())
}
}
for _, hash := range badUncles {
delete(self.possibleUncles, hash)
}
// Create the new block to seal with the consensus engine
if work.Block, err = self.engine.Finalize(self.chain, header, work.state, work.txs, uncles,
```

```
work.receipts); err != nil {
log.Error("Failed to finalize block for sealing", "err", err)
return
}
// We only care about logging if we're actually mining.
if atomic.LoadInt32(&self.mining) == 1 {
log.Info("Commit new mining work", "number", work.Block.Number(), "txs", work.tcount, "uncles",
len(uncles), "elapsed", common.PrettyDuration(time.Since(tstart)))
self.unconfirmed.Shift(work.Block.NumberU64() - 1)
}
self.push(work)
}

func (self *worker) commitUncle(work *Work, uncle *types.Header) error {
hash := uncle.Hash()
if work.uncles.Has(hash) {
return fmt.Errorf("uncle not unique")
}
if !work.ancestors.Has(uncle.ParentHash) {
return fmt.Errorf("uncle's parent unknown (%x)", uncle.ParentHash[0:4])
}
if work.family.Has(hash) {
return fmt.Errorf("uncle already in family (%x)", hash)
}
work.uncles.Add(uncle.Hash())
return nil
}

func (env *Work) commitTransactions(mux *event.TypeMux, txs
*types.TransactionsByPriceAndNonce, bc *core.BlockChain, coinbase common.Address) {
gp := new(core.GasPool).AddGas(env.header.GasLimit)

var coalescedLogs []*types.Log

for {
// Retrieve the next transaction and abort if all done
tx := txs.Peek()
if tx == nil {
break
}
// Error may be ignored here. The error has already been checked
// during transaction acceptance is the transaction pool.
```

```
//
// We use the eip155 signer regardless of the current hf.
from, _ := types.Sender(env.signer, tx)
// Check whether the tx is replay protected. If we're not in the EIP155 hf
// phase, start ignoring the sender until we do.
if tx.Protected() && !env.config.IsEIP155(env.header.Number) {
log.Trace("Ignoring reply protected transaction", "hash", tx.Hash(), "eip155",
env.config.EIP155Block)

txs.Pop()
continue
}
// Start executing the transaction
env.state.Prepare(tx.Hash(), common.Hash{}, env.tcount)

err, logs := env.commitTransaction(tx, bc, coinbase, gp)
switch err {
case core.ErrGasLimitReached:
// Pop the current out-of-gas transaction without shifting in the next from the account
log.Trace("Gas limit exceeded for current block", "sender", from)
txs.Pop()

case nil:
// Everything ok, collect the logs and shift in the next transaction from the same account
coalescedLogs = append(coalescedLogs, logs...)
env.tcount++
txs.Shift()

default:
// Pop the current failed transaction without shifting in the next from the account
log.Trace("Transaction failed, will be removed", "hash", tx.Hash(), "err", err)
env.failedTxs = append(env.failedTxs, tx)
txs.Pop()
}
}

if len(coalescedLogs) > 0 || env.tcount > 0 {
// make a copy, the state caches the logs and these logs get "upgraded" from pending to mined
// logs by filling in the block hash when the block was mined by the local miner. This can
// cause a race condition if a log was "upgraded" before the PendingLogsEvent is processed.
cpy := make([]*types.Log, len(coalescedLogs))
for i, l := range coalescedLogs {
```

```
cpy[i] = new(types.Log)
*cpy[i] = *l
}
go func(logs []*types.Log, tcount int) {
if len(logs) > 0 {
mux.Post(core.PendingLogsEvent{Logs: logs})
}
if tcount > 0 {
mux.Post(core.PendingStateEvent{})
}
}(cpy, env.tcount)
}
}

func (env *Work) commitTransaction(tx *types.Transaction, bc *core.BlockChain, coinbase
common.Address, gp *core.GasPool) (error, []*types.Log) {
snap := env.state.Snapshot()

receipt, _, err := core.ApplyTransaction(env.config, bc, &coinbase, gp, env.state, env.header, tx,
env.header.GasUsed, vm.Config{})
if err != nil {
env.state.RevertToSnapshot(snap)
return err, nil
}
env.txs = append(env.txs, tx)
env.receipts = append(env.receipts, receipt)

return nil, receipt.Logs
}

47:F:\git\coin\ethereum\go-ethereum\mobile\accounts.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains all the wrappers from the accounts package to support client side key
// management on mobile platforms.

package geth

import (
"errors"
"time"
```

```
"github.com/ethereum/go-ethereum/accounts"
"github.com/ethereum/go-ethereum/accounts/keystore"
"github.com/ethereum/go-ethereum/crypto"
)

const (
// StandardScryptN is the N parameter of Scrypt encryption algorithm, using 256MB
// memory and taking approximately 1s CPU time on a modern processor.
StandardScryptN = int(keystore.StandardScryptN)

// StandardScryptP is the P parameter of Scrypt encryption algorithm, using 256MB
// memory and taking approximately 1s CPU time on a modern processor.
StandardScryptP = int(keystore.StandardScryptP)

// LightScryptN is the N parameter of Scrypt encryption algorithm, using 4MB
// memory and taking approximately 100ms CPU time on a modern processor.
LightScryptN = int(keystore.LightScryptN)

// LightScryptP is the P parameter of Scrypt encryption algorithm, using 4MB
// memory and taking approximately 100ms CPU time on a modern processor.
LightScryptP = int(keystore.LightScryptP)
)

// Account represents a stored key.
type Account struct{ account accounts.Account }

// Accounts represents a slice of accounts.
type Accounts struct{ accounts []accounts.Account }

// Size returns the number of accounts in the slice.
func (a *Accounts) Size() int {
return len(a.accounts)
}

// Get returns the account at the given index from the slice.
func (a *Accounts) Get(index int) (account *Account, _ error) {
if index < 0 || index >= len(a.accounts) {
return nil, errors.New("index out of bounds")
}
return &Account{a.accounts[index]}, nil
}
```

```
// Set sets the account at the given index in the slice.
func (a *Accounts) Set(index int, account *Account) error {
if index < 0 || index >= len(a.accounts) {
return errors.New("index out of bounds")
}
a.accounts[index] = account.account
return nil
}

// GetAddress retrieves the address associated with the account.
func (a *Account) GetAddress() *Address {
return &Address{a.account.Address}
}

// GetURL retrieves the canonical URL of the account.
func (a *Account) GetURL() string {
return a.account.URL.String()
}

// KeyStore manages a key storage directory on disk.
type KeyStore struct{ keystore *keystore.KeyStore }

// NewKeyStore creates a keystore for the given directory.
func NewKeyStore(keydir string, scryptN, scryptP int) *KeyStore {
return &KeyStore{keystore: keystore.NewKeyStore(keydir, scryptN, scryptP)}
}

// HasAddress reports whether a key with the given address is present.
func (ks *KeyStore) HasAddress(address *Address) bool {
return ks.keystore.HasAddress(address.address)
}

// GetAccounts returns all key files present in the directory.
func (ks *KeyStore) GetAccounts() *Accounts {
return &Accounts{ks.keystore.Accounts()}
}

// DeleteAccount deletes the key matched by account if the passphrase is correct.
// If a contains no filename, the address must match a unique key.
func (ks *KeyStore) DeleteAccount(account *Account, passphrase string) error {
return ks.keystore.Delete(account.account, passphrase)
}
```

```
// SignHash calculates a ECDSA signature for the given hash. The produced signature
// is in the [R || S || V] format where V is 0 or 1.
func (ks *KeyStore) SignHash(address *Address, hash []byte) (signature []byte, _ error) {
return ks.keystore.SignHash(accounts.Account{Address: address.address}, hash)
}

// SignTx signs the given transaction with the requested account.
func (ks *KeyStore) SignTx(account *Account, tx *Transaction, chainID *BigInt) (*Transaction,
error) {
if chainID == nil { // Null passed from mobile app
chainID = new(BigInt)
}
signed, err := ks.keystore.SignTx(account.account, tx.tx, chainID.bigint)
if err != nil {
return nil, err
}
return &Transaction{signed}, nil
}

// SignHashPassphrase signs hash if the private key matching the given address can
// be decrypted with the given passphrase. The produced signature is in the
// [R || S || V] format where V is 0 or 1.
func (ks *KeyStore) SignHashPassphrase(account *Account, passphrase string, hash []byte)
(signature []byte, _ error) {
return ks.keystore.SignHashWithPassphrase(account.account, passphrase, hash)
}

// SignTxPassphrase signs the transaction if the private key matching the
// given address can be decrypted with the given passphrase.
func (ks *KeyStore) SignTxPassphrase(account *Account, passphrase string, tx *Transaction,
chainID *BigInt) (*Transaction, error) {
if chainID == nil { // Null passed from mobile app
chainID = new(BigInt)
}
signed, err := ks.keystore.SignTxWithPassphrase(account.account, passphrase, tx.tx,
chainID.bigint)
if err != nil {
return nil, err
}
return &Transaction{signed}, nil
}
```

```
// Unlock unlocks the given account indefinitely.
func (ks *KeyStore) Unlock(account *Account, passphrase string) error {
return ks.keystore.TimedUnlock(account.account, passphrase, 0)
}

// Lock removes the private key with the given address from memory.
func (ks *KeyStore) Lock(address *Address) error {
return ks.keystore.Lock(address.address)
}

// TimedUnlock unlocks the given account with the passphrase. The account stays
// unlocked for the duration of timeout (nanoseconds). A timeout of 0 unlocks the
// account until the program exits. The account must match a unique key file.
//
// If the account address is already unlocked for a duration, TimedUnlock extends or
// shortens the active unlock timeout. If the address was previously unlocked
// indefinitely the timeout is not altered.
func (ks *KeyStore) TimedUnlock(account *Account, passphrase string, timeout int64) error {
return ks.keystore.TimedUnlock(account.account, passphrase, time.Duration(timeout))
}

// NewAccount generates a new key and stores it into the key directory,
// encrypting it with the passphrase.
func (ks *KeyStore) NewAccount(passphrase string) (*Account, error) {
account, err := ks.keystore.NewAccount(passphrase)
if err != nil {
return nil, err
}
return &Account{account}, nil
}

// UpdateAccount changes the passphrase of an existing account.
func (ks *KeyStore) UpdateAccount(account *Account, passphrase, newPassphrase string) error {
return ks.keystore.Update(account.account, passphrase, newPassphrase)
}

// ExportKey exports as a JSON key, encrypted with newPassphrase.
func (ks *KeyStore) ExportKey(account *Account, passphrase, newPassphrase string) (key []byte,
_ error) {
return ks.keystore.Export(account.account, passphrase, newPassphrase)
}
```

```
// ImportKey stores the given encrypted JSON key into the key directory.
func (ks *KeyStore) ImportKey(keyJSON []byte, passphrase, newPassphrase string) (account
*Account, _ error) {
acc, err := ks.keystore.Import(keyJSON, passphrase, newPassphrase)
if err != nil {
return nil, err
}
return &Account{acc}, nil
}

// ImportECDSAKey stores the given encrypted JSON key into the key directory.
func (ks *KeyStore) ImportECDSAKey(key []byte, passphrase string) (account *Account, _ error) {
privkey, err := crypto.ToECDSA(key)
if err != nil {
return nil, err
}
acc, err := ks.keystore.ImportECDSA(privkey, passphrase)
if err != nil {
return nil, err
}
return &Account{acc}, nil
}

// ImportPreSaleKey decrypts the given Ethereum presale wallet and stores
// a key file in the key directory. The key file is encrypted with the same passphrase.
func (ks *KeyStore) ImportPreSaleKey(keyJSON []byte, passphrase string) (ccount *Account, _
error) {
account, err := ks.keystore.ImportPreSaleKey(keyJSON, passphrase)
if err != nil {
return nil, err
}
return &Account{account}, nil
}

48:F:\git\coin\ethereum\go-ethereum\mobile\android_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package geth

import (
"io/ioutil"
```

```
"os"
"os/exec"
"path/filepath"
"runtime"
"testing"
"time"

"github.com/ethereum/go-ethereum/internal/build"
)

// androidTestClass is a Java class to do some lightweight tests against the Android
// bindings. The goal is not to test each individual functionality, rather just to
// catch breaking API and/or implementation changes.
const androidTestClass = `
package go;

import android.test.InstrumentationTestCase;
import android.test.MoreAsserts;

import java.math.BigInteger;
import java.util.Arrays;

import org.ethereum.geth.*;

public class AndroidTest extends InstrumentationTestCase {
public AndroidTest() {}

public void testAccountManagement() {
// Create an encrypted keystore with light crypto parameters.
KeyStore ks = new KeyStore(getInstrumentation().getContext().getFilesDir() + "/keystore",
Geth.LightScryptN, Geth.LightScryptP);

try {
// Create a new account with the specified encryption passphrase.
Account newAcc = ks.newAccount("Creation password");

// Export the newly created account with a different passphrase. The returned
// data from this method invocation is a JSON encoded, encrypted key-file.
byte[] jsonAcc = ks.exportKey(newAcc, "Creation password", "Export password");

// Update the passphrase on the account created above inside the local keystore.
ks.updateAccount(newAcc, "Creation password", "Update password");
```

```
// Delete the account updated above from the local keystore.
ks.deleteAccount(newAcc, "Update password");

// Import back the account we've exported (and then deleted) above with yet
// again a fresh passphrase.
Account impAcc = ks.importKey(jsonAcc, "Export password", "Import password");

// Create a new account to sign transactions with
Account signer = ks.newAccount("Signer password");

Transaction tx = new Transaction(
1, new Address("0x0000000000000000000000000000000000000000"),
new BigInt(0), new BigInt(0), new BigInt(1), null); // Random empty transaction
BigInt chain = new BigInt(1); // Chain identifier of the main net

// Sign a transaction with a single authorization
Transaction signed = ks.signTxPassphrase(signer, "Signer password", tx, chain);

// Sign a transaction with multiple manually cancelled authorizations
ks.unlock(signer, "Signer password");
signed = ks.signTx(signer, tx, chain);
ks.lock(signer.getAddress());

// Sign a transaction with multiple automatically cancelled authorizations
ks.timedUnlock(signer, "Signer password", 1000000000);
signed = ks.signTx(signer, tx, chain);
} catch (Exception e) {
fail(e.toString());
}
}

public void testInprocNode() {
Context ctx = new Context();

try {
// Start up a new inprocess node
Node node = new Node(getInstrumentation().getContext().getFilesDir() + "/.ethereum", new
NodeConfig());
node.start();

// Retrieve some data via function calls (we don't really care about the results)
```

```
NodeInfo info = node.getNodeInfo();
info.getName();
info.getListenerAddress();
info.getProtocols();

// Retrieve some data via the APIs (we don't really care about the results)
EthereumClient ec = node.getEthereumClient();
ec.getBlockByNumber(ctx, -1).getNumber();

NewHeadHandler handler = new NewHeadHandler() {
@Override public void onError(String error)          {}
@Override public void onNewHead(final Header header) {}
};
ec.subscribeNewHead(ctx, handler,  16);
} catch (Exception e) {
fail(e.toString());
}
}

// Tests that recovering transaction signers works for both Homestead and EIP155
// signatures too. Regression test for go-ethereum issue #14599.
public void testIssue14599() {
try {
byte[] preEIP155RLP = new
BigInteger("f901fc8032830138808080b901ae60056013565b6101918061001d6000396000f35b33
6000819055505600600160006e060020a6000350480630a874df61461003a57806341c0e1b514
610058578063a02b161e14610066578063dbbdf0831461007757005b610045600435610149565b
80600160a060020a0316600052602060000f35b610060610161565b60006000f35b6100716004356
100d4565b60006000f35b6100856004356024356100856008b565b60006000f35b600054600160a06002
0a031632600160a060020a031614156100ac576100b1565b6100d0565b8060018360005260205
60406000208190555081600060005260206000a15b5050565b600054600160a060020a03163360
0160a060020a0316141580156101185750336001600160a060020a0316600182600052602052604060
002054600160a060020a031614155b61012157610126565b610146565b60006001826000526020
526040600020819055508060006000526020600a15b50565b60006001826000526020526040600
0205490509190505565b600054600160a060020a031633600160a060020a031614610181576101
8f565b600054600160a060020a0316ff5b561ca0c5689ed1ad124753d54576dfb4b571465a41900a
1dff4058d8adf16f752013d0a01221cbd70ec28c94a3b55ec771bcbc70778d6ee0b51ca7ea9514594
c861b1884", 16).toByteArray();
preEIP155RLP = Arrays.copyOfRange(preEIP155RLP, 1, preEIP155RLP.length);

byte[] postEIP155RLP = new
BigInteger("f86b8084773594008252089 4ef5bbb9bba2e1ca69ef81b23a8727d889f3ef0a1880de0b
```

```
6b3a7640000802ba06fef16c44726a102e6d55a651740636ef8aec6df3ebf009e7b0c1f29e4ac114a
a057e7fbc69760b522a78bb568cfc37a58bfdcf6ea86cb8f9b550263f58074b9cc", 16).toByteArray();
postEIP155RLP = Arrays.copyOfRange(postEIP155RLP, 1, postEIP155RLP.length);

Transaction preEIP155 =  new Transaction(preEIP155RLP);
Transaction postEIP155 = new Transaction(postEIP155RLP);

preEIP155.getFrom(null);          // Homestead should accept homestead
preEIP155.getFrom(new BigInt(4));  // EIP155 should accept homestead (missing chain ID)
postEIP155.getFrom(new BigInt(4)); // EIP155 should accept EIP 155

try {
postEIP155.getFrom(null);
fail("EIP155 transaction accepted by Homestead");
} catch (Exception e) {}
} catch (Exception e) {
fail(e.toString());
}
}
}
`


// TestAndroid runs the Android java test class specified above.
//
// This requires the gradle command in PATH and the Android SDK whose path is available
// through ANDROID_HOME environment variable. To successfully run the tests, an Android
// device must also be available with debugging enabled.
//
// This method has been adapted from golang.org/x/mobile/bind/java/seq_test.go/runTest
func TestAndroid(t *testing.T) {
// Skip tests on Windows altogether
if runtime.GOOS == "windows" {
t.Skip("cannot test Android bindings on Windows, skipping")
}
// Make sure all the Android tools are installed
if _, err := exec.Command("which", "gradle").CombinedOutput(); err != nil {
t.Skip("command gradle not found, skipping")
}
if sdk := os.Getenv("ANDROID_HOME"); sdk == "" {
t.Skip("ANDROID_HOME environment var not set, skipping")
}
if _, err := exec.Command("which", "gomobile").CombinedOutput(); err != nil {
```

```
t.Log("gomobile missing, installing it...")
if _, err := exec.Command("go", "install", "golang.org/x/mobile/cmd/gomobile").CombinedOutput();
err != nil {
t.Fatalf("install failed: %v", err)
}
t.Log("initializing gomobile...")
start := time.Now()
if _, err := exec.Command("gomobile", "init").CombinedOutput(); err != nil {
t.Fatalf("initialization failed: %v", err)
}
t.Logf("initialization took %v", time.Since(start))
}
// Create and switch to a temporary workspace
workspace, err := ioutil.TempDir("", "geth-android-")
if err != nil {
t.Fatalf("failed to create temporary workspace: %v", err)
}
defer os.RemoveAll(workspace)

pwd, err := os.Getwd()
if err != nil {
t.Fatalf("failed to get current working directory: %v", err)
}
if err := os.Chdir(workspace); err != nil {
t.Fatalf("failed to switch to temporary workspace: %v", err)
}
defer os.Chdir(pwd)

// Create the skeleton of the Android project
for _, dir := range []string{"src/main", "src/androidTest/java/org/ethereum/gethtest", "libs"} {
err = os.MkdirAll(dir, os.ModePerm)
if err != nil {
t.Fatal(err)
}
}
// Generate the mobile bindings for Geth and add the tester class
gobind := exec.Command("gomobile", "bind", "-javapkg", "org.ethereum",
"github.com/ethereum/go-ethereum/mobile")
if output, err := gobind.CombinedOutput(); err != nil {
t.Logf("%s", output)
t.Fatalf("failed to run gomobile bind: %v", err)
}
```

```
build.CopyFile(filepath.Join("libs", "geth.aar"), "geth.aar", os.ModePerm)

if err = ioutil.WriteFile(filepath.Join("src", "androidTest", "java", "org", "ethereum", "gethtest",
"AndroidTest.java"), []byte(androidTestClass), os.ModePerm); err != nil {
t.Fatalf("failed to write Android test class: %v", err)
}
// Finish creating the project and run the tests via gradle
if err = ioutil.WriteFile(filepath.Join("src", "main", "AndroidManifest.xml"), []byte(androidManifest),
os.ModePerm); err != nil {
t.Fatalf("failed to write Android manifest: %v", err)
}
if err = ioutil.WriteFile("build.gradle", []byte(gradleConfig), os.ModePerm); err != nil {
t.Fatalf("failed to write gradle build file: %v", err)
}
if output, err := exec.Command("gradle", "connectedAndroidTest").CombinedOutput(); err != nil {
t.Logf("%s", output)
t.Errorf("failed to run gradle test: %v", err)
}
}

const androidManifest = `<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="org.ethereum.gethtest"
  android:versionCode="1"
  android:versionName="1.0">

<uses-permission android:name="android.permission.INTERNET" />
</manifest>`

const gradleConfig = `buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.5.0'
    }
}
allprojects {
    repositories { jcenter() }
}
apply plugin: 'com.android.library'
android {
```

```
    compileSdkVersion 'android-19'
    buildToolsVersion '21.1.2'
    defaultConfig { minSdkVersion 15 }
}
repositories {
    flatDir { dirs 'libs' }
}
dependencies {
    compile 'com.android.support:appcompat-v7:19.0.0'
    compile(name: "geth", ext: "aar")
}
`
```

49:F:\git\coin\ethereum\go-ethereum\mobile\big.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains all the wrappers from the math/big package.

package geth

import (
"errors"
"math/big"
)

// A BigInt represents a signed multi-precision integer.
type BigInt struct {
bigint *big.Int
}

// NewBigInt allocates and returns a new BigInt set to x.
func NewBigInt(x int64) *BigInt {
return &BigInt{big.NewInt(x)}
}

// GetBytes returns the absolute value of x as a big-endian byte slice.
func (bi *BigInt) GetBytes() []byte {
return bi.bigint.Bytes()
}

// String returns the value of x as a formatted decimal string.
func (bi *BigInt) String() string {
```

```
return bi.bigint.String()
}

// GetInt64 returns the int64 representation of x. If x cannot be represented in
// an int64, the result is undefined.
func (bi *BigInt) GetInt64() int64 {
return bi.bigint.Int64()
}

// SetBytes interprets buf as the bytes of a big-endian unsigned integer and sets
// the big int to that value.
func (bi *BigInt) SetBytes(buf []byte) {
bi.bigint.SetBytes(buf)
}

// SetInt64 sets the big int to x.
func (bi *BigInt) SetInt64(x int64) {
bi.bigint.SetInt64(x)
}

// SetString sets the big int to x.
//
// The string prefix determines the actual conversion base. A prefix of "0x" or
// "0X" selects base 16; the "0" prefix selects base 8, and a "0b" or "0B" prefix
// selects base 2. Otherwise the selected base is 10.
func (bi *BigInt) SetString(x string, base int) {
bi.bigint.SetString(x, base)
}

// BigInts represents a slice of big ints.
type BigInts struct{ bigints []*big.Int }

// Size returns the number of big ints in the slice.
func (bi *BigInts) Size() int {
return len(bi.bigints)
}

// Get returns the bigint at the given index from the slice.
func (bi *BigInts) Get(index int) (bigint *BigInt, _ error) {
if index < 0 || index >= len(bi.bigints) {
return nil, errors.New("index out of bounds")
}
```

```
return &BigInt{bi.bigints[index]}, nil
}

// Set sets the big int at the given index in the slice.
func (bi *BigInts) Set(index int, bigint *BigInt) error {
if index < 0 || index >= len(bi.bigints) {
return errors.New("index out of bounds")
}
bi.bigints[index] = bigint.bigint
return nil
}

// GetString returns the value of x as a formatted string in some number base.
func (bi *BigInt) GetString(base int) string {
return bi.bigint.Text(base)
}

50:F:\git\coin\ethereum\go-ethereum\mobile\bind.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains all the wrappers from the bind package.

package geth

import (
"math/big"
"strings"

"github.com/ethereum/go-ethereum/accounts/abi"
"github.com/ethereum/go-ethereum/accounts/abi/bind"
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/types"
)

// Signer is an interaface defining the callback when a contract requires a
// method to sign the transaction before submission.
type Signer interface {
Sign(*Address, *Transaction) (tx *Transaction, _ error)
}

type signer struct {
sign bind.SignerFn
```

```
}

func (s *signer) Sign(addr *Address, unsignedTx *Transaction) (signedTx *Transaction, _ error) {
sig, err := s.sign(types.HomesteadSigner{}, addr.address, unsignedTx.tx)
if err != nil {
return nil, err
}
return &Transaction{sig}, nil
}

// CallOpts is the collection of options to fine tune a contract call request.
type CallOpts struct {
opts bind.CallOpts
}

// NewCallOpts creates a new option set for contract calls.
func NewCallOpts() *CallOpts {
return new(CallOpts)
}

func (opts *CallOpts) IsPending() bool    { return opts.opts.Pending }
func (opts *CallOpts) GetGasLimit() int64 { return 0 /* TODO(karalabe) */ }

// GetContext cannot be reliably implemented without identity preservation
(https://github.com/golang/go/issues/16876)
// Even then it's awkward to unpack the subtleties of a Go context out to Java.
// func (opts *CallOpts) GetContext() *Context { return &Context{opts.opts.Context} }

func (opts *CallOpts) SetPending(pending bool)     { opts.opts.Pending = pending }
func (opts *CallOpts) SetGasLimit(limit int64)     { /* TODO(karalabe) */ }
func (opts *CallOpts) SetContext(context *Context) { opts.opts.Context = context.context }

// TransactOpts is the collection of authorization data required to create a
// valid Ethereum transaction.
type TransactOpts struct {
opts bind.TransactOpts
}

func (opts *TransactOpts) GetFrom() *Address    { return &Address{opts.opts.From} }
func (opts *TransactOpts) GetNonce() int64      { return opts.opts.Nonce.Int64() }
func (opts *TransactOpts) GetValue() *BigInt    { return &BigInt{opts.opts.Value} }
func (opts *TransactOpts) GetGasPrice() *BigInt { return &BigInt{opts.opts.GasPrice} }
```

```
func (opts *TransactOpts) GetGasLimit() int64   { return opts.opts.GasLimit.Int64() }

// GetSigner cannot be reliably implemented without identity preservation
(https://github.com/golang/go/issues/16876)
// func (opts *TransactOpts) GetSigner() Signer { return &signer{opts.opts.Signer} }

// GetContext cannot be reliably implemented without identity preservation
(https://github.com/golang/go/issues/16876)
// Even then it's awkward to unpack the subtleties of a Go context out to Java.
//func (opts *TransactOpts) GetContext() *Context { return &Context{opts.opts.Context} }

func (opts *TransactOpts) SetFrom(from *Address) { opts.opts.From = from.address }
func (opts *TransactOpts) SetNonce(nonce int64)  { opts.opts.Nonce = big.NewInt(nonce) }
func (opts *TransactOpts) SetSigner(s Signer) {
opts.opts.Signer = func(signer types.Signer, addr common.Address, tx *types.Transaction)
(*types.Transaction, error) {
sig, err := s.Sign(&Address{addr}, &Transaction{tx})
if err != nil {
return nil, err
}
return sig.tx, nil
}
}
func (opts *TransactOpts) SetValue(value *BigInt)      { opts.opts.Value = value.bigint }
func (opts *TransactOpts) SetGasPrice(price *BigInt)   { opts.opts.GasPrice = price.bigint }
func (opts *TransactOpts) SetGasLimit(limit int64)     { opts.opts.GasLimit = big.NewInt(limit) }
func (opts *TransactOpts) SetContext(context *Context) { opts.opts.Context = context.context }

// BoundContract is the base wrapper object that reflects a contract on the
// Ethereum network. It contains a collection of methods that are used by the
// higher level contract bindings to operate.
type BoundContract struct {
contract *bind.BoundContract
address  common.Address
deployer *types.Transaction
}

// DeployContract deploys a contract onto the Ethereum blockchain and binds the
// deployment address with a wrapper.
func DeployContract(opts *TransactOpts, abiJSON string, bytecode []byte, client *EthereumClient,
args *Interfaces) (contract *BoundContract, _ error) {
// Deploy the contract to the network
```

```
parsed, err := abi.JSON(strings.NewReader(abiJSON))
if err != nil {
return nil, err
}
addr, tx, bound, err := bind.DeployContract(&opts.opts, parsed, bytecode, client.client,
args.objects...)
if err != nil {
return nil, err
}
return &BoundContract{
contract: bound,
address:  addr,
deployer: tx,
}, nil
}

// BindContract creates a low level contract interface through which calls and
// transactions may be made through.
func BindContract(address *Address, abiJSON string, client *EthereumClient) (contract
*BoundContract, _ error) {
parsed, err := abi.JSON(strings.NewReader(abiJSON))
if err != nil {
return nil, err
}
return &BoundContract{
contract: bind.NewBoundContract(address.address, parsed, client.client, client.client),
address:  address.address,
}, nil
}

func (c *BoundContract) GetAddress() *Address { return &Address{c.address} }
func (c *BoundContract) GetDeployer() *Transaction {
if c.deployer == nil {
return nil
}
return &Transaction{c.deployer}
}

// Call invokes the (constant) contract method with params as input values and
// sets the output to result.
func (c *BoundContract) Call(opts *CallOpts, out *Interfaces, method string, args *Interfaces) error
{
```

```
results := make([]interface{}, len(out.objects))
copy(results, out.objects)
if err := c.contract.Call(&opts.opts, &results, method, args.objects...); err != nil {
return err
}
copy(out.objects, results)
return nil
}

// Transact invokes the (paid) contract method with params as input values.
func (c *BoundContract) Transact(opts *TransactOpts, method string, args *Interfaces) (tx
*Transaction, _ error) {
rawTx, err := c.contract.Transact(&opts.opts, method, args.objects)
if err != nil {
return nil, err
}
return &Transaction{rawTx}, nil
}

// Transfer initiates a plain transaction to move funds to the contract, calling
// its default method if one is available.
func (c *BoundContract) Transfer(opts *TransactOpts) (tx *Transaction, _ error) {
rawTx, err := c.contract.Transfer(&opts.opts)
if err != nil {
return nil, err
}
return &Transaction{rawTx}, nil
}

51:F:\git\coin\ethereum\go-ethereum\mobile\common.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains all the wrappers from the common package.

package geth

import (
"encoding/hex"
"errors"
"fmt"
"strings"
```

```
"github.com/ethereum/go-ethereum/common"
)

// Hash represents the 32 byte Keccak256 hash of arbitrary data.
type Hash struct {
hash common.Hash
}

// NewHashFromBytes converts a slice of bytes to a hash value.
func NewHashFromBytes(binary []byte) (hash *Hash, _ error) {
h := new(Hash)
if err := h.SetBytes(binary); err != nil {
return nil, err
}
return h, nil
}

// NewHashFromHex converts a hex string to a hash value.
func NewHashFromHex(hex string) (hash *Hash, _ error) {
h := new(Hash)
if err := h.SetHex(hex); err != nil {
return nil, err
}
return h, nil
}

// SetBytes sets the specified slice of bytes as the hash value.
func (h *Hash) SetBytes(hash []byte) error {
if length := len(hash); length != common.HashLength {
return fmt.Errorf("invalid hash length: %v != %v", length, common.HashLength)
}
copy(h.hash[:], hash)
return nil
}

// GetBytes retrieves the byte representation of the hash.
func (h *Hash) GetBytes() []byte {
return h.hash[:]
}

// SetHex sets the specified hex string as the hash value.
func (h *Hash) SetHex(hash string) error {
```

```
hash = strings.ToLower(hash)
if len(hash) >= 2 && hash[:2] == "0x" {
hash = hash[2:]
}
if length := len(hash); length != 2*common.HashLength {
return fmt.Errorf("invalid hash hex length: %v != %v", length, 2*common.HashLength)
}
bin, err := hex.DecodeString(hash)
if err != nil {
return err
}
copy(h.hash[:], bin)
return nil
}

// GetHex retrieves the hex string representation of the hash.
func (h *Hash) GetHex() string {
return h.hash.Hex()
}

// Hashes represents a slice of hashes.
type Hashes struct{ hashes []common.Hash }

// NewHashes creates a slice of uninitialized Hashes.
func NewHashes(size int) *Hashes {
return &Hashes{
hashes: make([]common.Hash, size),
}
}

// NewHashesEmpty creates an empty slice of Hashes values.
func NewHashesEmpty() *Hashes {
return NewHashes(0)
}

// Size returns the number of hashes in the slice.
func (h *Hashes) Size() int {
return len(h.hashes)
}

// Get returns the hash at the given index from the slice.
func (h *Hashes) Get(index int) (hash *Hash, _ error) {
```

```
if index < 0 || index >= len(h.hashes) {
return nil, errors.New("index out of bounds")
}
return &Hash{h.hashes[index]}, nil
}

// Set sets the Hash at the given index in the slice.
func (h *Hashes) Set(index int, hash *Hash) error {
if index < 0 || index >= len(h.hashes) {
return errors.New("index out of bounds")
}
h.hashes[index] = hash.hash
return nil
}

// Append adds a new Hash element to the end of the slice.
func (h *Hashes) Append(hash *Hash) {
h.hashes = append(h.hashes, hash.hash)
}

// Address represents the 20 byte address of an Ethereum account.
type Address struct {
address common.Address
}

// NewAddressFromBytes converts a slice of bytes to a hash value.
func NewAddressFromBytes(binary []byte) (address *Address, _ error) {
a := new(Address)
if err := a.SetBytes(binary); err != nil {
return nil, err
}
return a, nil
}

// NewAddressFromHex converts a hex string to a address value.
func NewAddressFromHex(hex string) (address *Address, _ error) {
a := new(Address)
if err := a.SetHex(hex); err != nil {
return nil, err
}
return a, nil
}
```

```
// SetBytes sets the specified slice of bytes as the address value.
func (a *Address) SetBytes(address []byte) error {
if length := len(address); length != common.AddressLength {
return fmt.Errorf("invalid address length: %v != %v", length, common.AddressLength)
}
copy(a.address[:], address)
return nil
}

// GetBytes retrieves the byte representation of the address.
func (a *Address) GetBytes() []byte {
return a.address[:]
}

// SetHex sets the specified hex string as the address value.
func (a *Address) SetHex(address string) error {
address = strings.ToLower(address)
if len(address) >= 2 && address[:2] == "0x" {
address = address[2:]
}
if length := len(address); length != 2*common.AddressLength {
return fmt.Errorf("invalid address hex length: %v != %v", length, 2*common.AddressLength)
}
bin, err := hex.DecodeString(address)
if err != nil {
return err
}
copy(a.address[:], bin)
return nil
}

// GetHex retrieves the hex string representation of the address.
func (a *Address) GetHex() string {
return a.address.Hex()
}

// Addresses represents a slice of addresses.
type Addresses struct{ addresses []common.Address }

// NewAddresses creates a slice of uninitialized addresses.
func NewAddresses(size int) *Addresses {
```

```
return &Addresses{
addresses: make([]common.Address, size),
}
}

// NewAddressesEmpty creates an empty slice of Addresses values.
func NewAddressesEmpty() *Addresses {
return NewAddresses(0)
}

// Size returns the number of addresses in the slice.
func (a *Addresses) Size() int {
return len(a.addresses)
}

// Get returns the address at the given index from the slice.
func (a *Addresses) Get(index int) (address *Address, _ error) {
if index < 0 || index >= len(a.addresses) {
return nil, errors.New("index out of bounds")
}
return &Address{a.addresses[index]}, nil
}

// Set sets the address at the given index in the slice.
func (a *Addresses) Set(index int, address *Address) error {
if index < 0 || index >= len(a.addresses) {
return errors.New("index out of bounds")
}
a.addresses[index] = address.address
return nil
}

// Append adds a new address element to the end of the slice.
func (a *Addresses) Append(address *Address) {
a.addresses = append(a.addresses, address.address)
}

52:F:\git\coin\ethereum\go-ethereum\mobile\context.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains all the wrappers from the golang.org/x/net/context package to support
// client side context management on mobile platforms.
```

```
package geth

import (
"context"
"time"
)

// Context carries a deadline, a cancelation signal, and other values across API
// boundaries.
type Context struct {
context context.Context
cancel  context.CancelFunc
}

// NewContext returns a non-nil, empty Context. It is never canceled, has no
// values, and has no deadline. It is typically used by the main function,
// initialization, and tests, and as the top-level Context for incoming requests.
func NewContext() *Context {
return &Context{
context: context.Background(),
}
}

// WithCancel returns a copy of the original context with cancellation mechanism
// included.
//
// Canceling this context releases resources associated with it, so code should
// call cancel as soon as the operations running in this Context complete.
func (c *Context) WithCancel() *Context {
child, cancel := context.WithCancel(c.context)
return &Context{
context: child,
cancel:  cancel,
}
}

// WithDeadline returns a copy of the original context with the deadline adjusted
// to be no later than the specified time.
//
// Canceling this context releases resources associated with it, so code should
// call cancel as soon as the operations running in this Context complete.
```

```
func (c *Context) WithDeadline(sec int64, nsec int64) *Context {
child, cancel := context.WithDeadline(c.context, time.Unix(sec, nsec))
return &Context{
context: child,
cancel:  cancel,
}
}

// WithTimeout returns a copy of the original context with the deadline adjusted
// to be no later than now + the duration specified.
//
// Canceling this context releases resources associated with it, so code should
// call cancel as soon as the operations running in this Context complete.
func (c *Context) WithTimeout(nsec int64) *Context {
child, cancel := context.WithTimeout(c.context, time.Duration(nsec))
return &Context{
context: child,
cancel:  cancel,
}
}

53:F:\git\coin\ethereum\go-ethereum\mobile\discover.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains all the wrappers from the accounts package to support client side enode
// management on mobile platforms.

package geth

import (
"errors"

"github.com/ethereum/go-ethereum/p2p/discv5"
)

// Enode represents a host on the network.
type Enode struct {
node *discv5.Node
}

// NewEnode parses a node designator.
//
```

```
// There are two basic forms of node designators
//   - incomplete nodes, which only have the public key (node ID)
//   - complete nodes, which contain the public key and IP/Port information
//
// For incomplete nodes, the designator must look like one of these
//
//    enode://<hex node id>
//    <hex node id>
//
// For complete nodes, the node ID is encoded in the username portion
// of the URL, separated from the host by an @ sign. The hostname can
// only be given as an IP address, DNS domain names are not allowed.
// The port in the host name section is the TCP listening port. If the
// TCP and UDP (discovery) ports differ, the UDP port is specified as
// query parameter "discport".
//
// In the following example, the node URL describes
// a node with IP address 10.3.58.6, TCP listening port 30303
// and UDP discovery port 30301.
//
//    enode://<hex node id>@10.3.58.6:30303?discport=30301
func NewEnode(rawurl string) (enode *Enode, _ error) {
node, err := discv5.ParseNode(rawurl)
if err != nil {
return nil, err
}
return &Enode{node}, nil
}

// Enodes represents a slice of accounts.
type Enodes struct{ nodes []*discv5.Node }

// NewEnodes creates a slice of uninitialized enodes.
func NewEnodes(size int) *Enodes {
return &Enodes{
nodes: make([]*discv5.Node, size),
}
}

// NewEnodesEmpty creates an empty slice of Enode values.
func NewEnodesEmpty() *Enodes {
return NewEnodes(0)
```

```
}

// Size returns the number of enodes in the slice.
func (e *Enodes) Size() int {
return len(e.nodes)
}

// Get returns the enode at the given index from the slice.
func (e *Enodes) Get(index int) (enode *Enode, _ error) {
if index < 0 || index >= len(e.nodes) {
return nil, errors.New("index out of bounds")
}
return &Enode{e.nodes[index]}, nil
}

// Set sets the enode at the given index in the slice.
func (e *Enodes) Set(index int, enode *Enode) error {
if index < 0 || index >= len(e.nodes) {
return errors.New("index out of bounds")
}
e.nodes[index] = enode.node
return nil
}

// Append adds a new enode element to the end of the slice.
func (e *Enodes) Append(enode *Enode) {
e.nodes = append(e.nodes, enode.node)
}

54:F:\git\coin\ethereum\go-ethereum\mobile\doc.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package geth contains the simplified mobile APIs to go-ethereum.
//
// The scope of this package is *not* to allow writing a custom Ethereum client
// with pieces plucked from go-ethereum, rather to allow writing native dapps on
// mobile platforms. Keep this in mind when using or extending this package!
//
// API limitations
//
// Since gomobile cannot bridge arbitrary types between Go and Android/iOS, the
// exposed APIs need to be manually wrapped into simplified types, with custom
```

```
// constructors and getters/setters to ensure that they can be meaninfully used
// from Java/ObjC too.
//
// With this in mind, please try to limit the scope of this package and only add
// essentials without which mobile support cannot work, especially since manually
// syncing the code will be unwieldy otherwise. In the long term we might consider
// writing custom library generators, but those are out of scope now.
//
// Content wise each file in this package corresponds to an entire Go package
// from the go-ethereum repository. Please adhere to this scoping to prevent this
// package getting unmaintainable.
//
// Wrapping guidelines:
//
// Every type that is to be exposed should be wrapped into its own plain struct,
// which internally contains a single field: the original go-ethereum version.
// This is needed because gomobile cannot expose named types for now.
//
// Whenever a method argument or a return type is a custom struct, the pointer
// variant should always be used as value types crossing over between language
// boundaries might have strange behaviors.
//
// Slices of types should be converted into a single multiplicative type wrapping
// a go slice with the methods `Size`, `Get` and `Set`. Further slice operations
// should not be provided to limit the remote code complexity. Arrays should be
// avoided as much as possible since they complicate bounds checking.
//
// If a method has multiple return values (e.g. some return + an error), those
// are generated as output arguments in ObjC. To avoid weird generated names like
// ret_0 for them, please always assign names to output variables if tuples.
//
// Note, a panic *cannot* cross over language boundaries, instead will result in
// an undebuggable SEGFAULT in the process. For error handling only ever use error
// returns, which may be the only or the second return.
package geth

55:F:\git\coin\ethereum\go-ethereum\mobile\ethclient.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains a wrapper for the Ethereum client.

package geth
```

```
import (
"math/big"

"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/ethclient"
)

// EthereumClient provides access to the Ethereum APIs.
type EthereumClient struct {
client *ethclient.Client
}

// NewEthereumClient connects a client to the given URL.
func NewEthereumClient(rawurl string) (client *EthereumClient, _ error) {
rawClient, err := ethclient.Dial(rawurl)
return &EthereumClient{rawClient}, err
}

// GetBlockByHash returns the given full block.
func (ec *EthereumClient) GetBlockByHash(ctx *Context, hash *Hash) (block *Block, _ error) {
rawBlock, err := ec.client.BlockByHash(ctx.context, hash.hash)
return &Block{rawBlock}, err
}

// GetBlockByNumber returns a block from the current canonical chain. If number is <0, the
// latest known block is returned.
func (ec *EthereumClient) GetBlockByNumber(ctx *Context, number int64) (block *Block, _ error) {
if number < 0 {
rawBlock, err := ec.client.BlockByNumber(ctx.context, nil)
return &Block{rawBlock}, err
}
rawBlock, err := ec.client.BlockByNumber(ctx.context, big.NewInt(number))
return &Block{rawBlock}, err
}

// GetHeaderByHash returns the block header with the given hash.
func (ec *EthereumClient) GetHeaderByHash(ctx *Context, hash *Hash) (header *Header, _ error)
{
rawHeader, err := ec.client.HeaderByHash(ctx.context, hash.hash)
return &Header{rawHeader}, err
}
```

```
// GetHeaderByNumber returns a block header from the current canonical chain. If number is <0,
// the latest known header is returned.
func (ec *EthereumClient) GetHeaderByNumber(ctx *Context, number int64) (header *Header, _
error) {
if number < 0 {
rawHeader, err := ec.client.HeaderByNumber(ctx.context, nil)
return &Header{rawHeader}, err
}
rawHeader, err := ec.client.HeaderByNumber(ctx.context, big.NewInt(number))
return &Header{rawHeader}, err
}

// GetTransactionByHash returns the transaction with the given hash.
func (ec *EthereumClient) GetTransactionByHash(ctx *Context, hash *Hash) (tx *Transaction, _
error) {
// TODO(karalabe): handle isPending
rawTx, _, err := ec.client.TransactionByHash(ctx.context, hash.hash)
return &Transaction{rawTx}, err
}

// GetTransactionCount returns the total number of transactions in the given block.
func (ec *EthereumClient) GetTransactionCount(ctx *Context, hash *Hash) (count int, _ error) {
rawCount, err := ec.client.TransactionCount(ctx.context, hash.hash)
return int(rawCount), err
}

// GetTransactionInBlock returns a single transaction at index in the given block.
func (ec *EthereumClient) GetTransactionInBlock(ctx *Context, hash *Hash, index int) (tx
*Transaction, _ error) {
rawTx, err := ec.client.TransactionInBlock(ctx.context, hash.hash, uint(index))
return &Transaction{rawTx}, err

}

// GetTransactionReceipt returns the receipt of a transaction by transaction hash.
// Note that the receipt is not available for pending transactions.
func (ec *EthereumClient) GetTransactionReceipt(ctx *Context, hash *Hash) (receipt *Receipt, _
error) {
rawReceipt, err := ec.client.TransactionReceipt(ctx.context, hash.hash)
return &Receipt{rawReceipt}, err
}
```

```
// SyncProgress retrieves the current progress of the sync algorithm. If there's
// no sync currently running, it returns nil.
func (ec *EthereumClient) SyncProgress(ctx *Context) (progress *SyncProgress, _ error) {
rawProgress, err := ec.client.SyncProgress(ctx.context)
if rawProgress == nil {
return nil, err
}
return &SyncProgress{*rawProgress}, err
}

// NewHeadHandler is a client-side subscription callback to invoke on events and
// subscription failure.
type NewHeadHandler interface {
OnNewHead(header *Header)
OnError(failure string)
}

// SubscribeNewHead subscribes to notifications about the current blockchain head
// on the given channel.
func (ec *EthereumClient) SubscribeNewHead(ctx *Context, handler NewHeadHandler, buffer int)
(sub *Subscription, _ error) {
// Subscribe to the event internally
ch := make(chan *types.Header, buffer)
rawSub, err := ec.client.SubscribeNewHead(ctx.context, ch)
if err != nil {
return nil, err
}
// Start up a dispatcher to feed into the callback
go func() {
for {
select {
case header := <-ch:
handler.OnNewHead(&Header{header})

case err := <-rawSub.Err():
handler.OnError(err.Error())
return
}
}
}()
return &Subscription{rawSub}, nil
```

```
}

// State Access

// GetBalanceAt returns the wei balance of the given account.
// The block number can be <0, in which case the balance is taken from the latest known block.
func (ec *EthereumClient) GetBalanceAt(ctx *Context, account *Address, number int64) (balance
*BigInt, _ error) {
if number < 0 {
rawBalance, err := ec.client.BalanceAt(ctx.context, account.address, nil)
return &BigInt{rawBalance}, err
}
rawBalance, err := ec.client.BalanceAt(ctx.context, account.address, big.NewInt(number))
return &BigInt{rawBalance}, err
}

// GetStorageAt returns the value of key in the contract storage of the given account.
// The block number can be <0, in which case the value is taken from the latest known block.
func (ec *EthereumClient) GetStorageAt(ctx *Context, account *Address, key *Hash, number
int64) (storage []byte, _ error) {
if number < 0 {
return ec.client.StorageAt(ctx.context, account.address, key.hash, nil)
}
return ec.client.StorageAt(ctx.context, account.address, key.hash, big.NewInt(number))
}

// GetCodeAt returns the contract code of the given account.
// The block number can be <0, in which case the code is taken from the latest known block.
func (ec *EthereumClient) GetCodeAt(ctx *Context, account *Address, number int64) (code []byte,
_ error) {
if number < 0 {
return ec.client.CodeAt(ctx.context, account.address, nil)
}
return ec.client.CodeAt(ctx.context, account.address, big.NewInt(number))
}

// GetNonceAt returns the account nonce of the given account.
// The block number can be <0, in which case the nonce is taken from the latest known block.
func (ec *EthereumClient) GetNonceAt(ctx *Context, account *Address, number int64) (nonce
int64, _ error) {
if number < 0 {
rawNonce, err := ec.client.NonceAt(ctx.context, account.address, nil)
```

```
return int64(rawNonce), err
}
rawNonce, err := ec.client.NonceAt(ctx.context, account.address, big.NewInt(number))
return int64(rawNonce), err
}

// Filters

// FilterLogs executes a filter query.
func (ec *EthereumClient) FilterLogs(ctx *Context, query *FilterQuery) (logs *Logs, _ error) {
rawLogs, err := ec.client.FilterLogs(ctx.context, query.query)
if err != nil {
return nil, err
}
// Temp hack due to vm.Logs being []*vm.Log
res := make([]*types.Log, len(rawLogs))
for i, log := range rawLogs {
res[i] = &log
}
return &Logs{res}, nil
}

// FilterLogsHandler is a client-side subscription callback to invoke on events and
// subscription failure.
type FilterLogsHandler interface {
OnFilterLogs(log *Log)
OnError(failure string)
}

// SubscribeFilterLogs subscribes to the results of a streaming filter query.
func (ec *EthereumClient) SubscribeFilterLogs(ctx *Context, query *FilterQuery, handler
FilterLogsHandler, buffer int) (sub *Subscription, _ error) {
// Subscribe to the event internally
ch := make(chan types.Log, buffer)
rawSub, err := ec.client.SubscribeFilterLogs(ctx.context, query.query, ch)
if err != nil {
return nil, err
}
// Start up a dispatcher to feed into the callback
go func() {
for {
select {
```

```
case log := <-ch:
handler.OnFilterLogs(&Log{&log})

case err := <-rawSub.Err():
handler.OnError(err.Error())
return
}
}
}()
return &Subscription{rawSub}, nil
}

// Pending State

// GetPendingBalanceAt returns the wei balance of the given account in the pending state.
func (ec *EthereumClient) GetPendingBalanceAt(ctx *Context, account *Address) (balance
*BigInt, _ error) {
rawBalance, err := ec.client.PendingBalanceAt(ctx.context, account.address)
return &BigInt{rawBalance}, err
}

// GetPendingStorageAt returns the value of key in the contract storage of the given account in the
pending state.
func (ec *EthereumClient) GetPendingStorageAt(ctx *Context, account *Address, key *Hash)
(storage []byte, _ error) {
return ec.client.PendingStorageAt(ctx.context, account.address, key.hash)
}

// GetPendingCodeAt returns the contract code of the given account in the pending state.
func (ec *EthereumClient) GetPendingCodeAt(ctx *Context, account *Address) (code []byte, _
error) {
return ec.client.PendingCodeAt(ctx.context, account.address)
}

// GetPendingNonceAt returns the account nonce of the given account in the pending state.
// This is the nonce that should be used for the next transaction.
func (ec *EthereumClient) GetPendingNonceAt(ctx *Context, account *Address) (nonce int64, _
error) {
rawNonce, err := ec.client.PendingNonceAt(ctx.context, account.address)
return int64(rawNonce), err
}
```

```
// GetPendingTransactionCount returns the total number of transactions in the pending state.
func (ec *EthereumClient) GetPendingTransactionCount(ctx *Context) (count int, _ error) {
rawCount, err := ec.client.PendingTransactionCount(ctx.context)
return int(rawCount), err
}

// Contract Calling

// CallContract executes a message call transaction, which is directly executed in the VM
// of the node, but never mined into the blockchain.
//
// blockNumber selects the block height at which the call runs. It can be <0, in which
// case the code is taken from the latest known block. Note that state from very old
// blocks might not be available.
func (ec *EthereumClient) CallContract(ctx *Context, msg *CallMsg, number int64) (output []byte,
_ error) {
if number < 0 {
return ec.client.CallContract(ctx.context, msg.msg, nil)
}
return ec.client.CallContract(ctx.context, msg.msg, big.NewInt(number))
}

// PendingCallContract executes a message call transaction using the EVM.
// The state seen by the contract call is the pending state.
func (ec *EthereumClient) PendingCallContract(ctx *Context, msg *CallMsg) (output []byte, _
error) {
return ec.client.PendingCallContract(ctx.context, msg.msg)
}

// SuggestGasPrice retrieves the currently suggested gas price to allow a timely
// execution of a transaction.
func (ec *EthereumClient) SuggestGasPrice(ctx *Context) (price *BigInt, _ error) {
rawPrice, err := ec.client.SuggestGasPrice(ctx.context)
return &BigInt{rawPrice}, err
}

// EstimateGas tries to estimate the gas needed to execute a specific transaction based on
// the current pending state of the backend blockchain. There is no guarantee that this is
// the true gas limit requirement as other transactions may be added or removed by miners,
// but it should provide a basis for setting a reasonable default.
func (ec *EthereumClient) EstimateGas(ctx *Context, msg *CallMsg) (gas *BigInt, _ error) {
rawGas, err := ec.client.EstimateGas(ctx.context, msg.msg)
```

```
return &BigInt{rawGas}, err
}

// SendTransaction injects a signed transaction into the pending pool for execution.
//
// If the transaction was a contract creation use the TransactionReceipt method to get the
// contract address after the transaction has been mined.
func (ec *EthereumClient) SendTransaction(ctx *Context, tx *Transaction) error {
return ec.client.SendTransaction(ctx.context, tx.tx)
}

56:F:\git\coin\ethereum\go-ethereum\mobile\ethereum.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains all the wrappers from the go-ethereum root package.

package geth

import (
"errors"
"math/big"

ethereum "github.com/ethereum/go-ethereum"
"github.com/ethereum/go-ethereum/common"
)

// Subscription represents an event subscription where events are
// delivered on a data channel.
type Subscription struct {
sub ethereum.Subscription
}

// Unsubscribe cancels the sending of events to the data channel
// and closes the error channel.
func (s *Subscription) Unsubscribe() {
s.sub.Unsubscribe()
}

// CallMsg contains parameters for contract calls.
type CallMsg struct {
msg ethereum.CallMsg
}
```

```
// NewCallMsg creates an empty contract call parameter list.
func NewCallMsg() *CallMsg {
return new(CallMsg)
}

func (msg *CallMsg) GetFrom() *Address    { return &Address{msg.msg.From} }
func (msg *CallMsg) GetGas() int64        { return msg.msg.Gas.Int64() }
func (msg *CallMsg) GetGasPrice() *BigInt { return &BigInt{msg.msg.GasPrice} }
func (msg *CallMsg) GetValue() *BigInt    { return &BigInt{msg.msg.Value} }
func (msg *CallMsg) GetData() []byte      { return msg.msg.Data }
func (msg *CallMsg) GetTo() *Address {
if to := msg.msg.To; to != nil {
return &Address{*msg.msg.To}
}
return nil
}

func (msg *CallMsg) SetFrom(address *Address)  { msg.msg.From = address.address }
func (msg *CallMsg) SetGas(gas int64)          { msg.msg.Gas = big.NewInt(gas) }
func (msg *CallMsg) SetGasPrice(price *BigInt) { msg.msg.GasPrice = price.bigint }
func (msg *CallMsg) SetValue(value *BigInt)    { msg.msg.Value = value.bigint }
func (msg *CallMsg) SetData(data []byte)       { msg.msg.Data = data }
func (msg *CallMsg) SetTo(address *Address) {
if address == nil {
msg.msg.To = nil
}
msg.msg.To = &address.address
}

// SyncProgress gives progress indications when the node is synchronising with
// the Ethereum network.
type SyncProgress struct {
progress ethereum.SyncProgress
}

func (p *SyncProgress) GetStartingBlock() int64 { return int64(p.progress.StartingBlock) }
func (p *SyncProgress) GetCurrentBlock() int64  { return int64(p.progress.CurrentBlock) }
func (p *SyncProgress) GetHighestBlock() int64  { return int64(p.progress.HighestBlock) }
func (p *SyncProgress) GetPulledStates() int64  { return int64(p.progress.PulledStates) }
func (p *SyncProgress) GetKnownStates() int64   { return int64(p.progress.KnownStates) }
```

```
// Topics is a set of topic lists to filter events with.
type Topics struct{ topics [][]common.Hash }

// NewTopics creates a slice of uninitialized Topics.
func NewTopics(size int) *Topics {
return &Topics{
topics: make([][]common.Hash, size),
}
}

// NewTopicsEmpty creates an empty slice of Topics values.
func NewTopicsEmpty() *Topics {
return NewTopics(0)
}

// Size returns the number of topic lists inside the set
func (t *Topics) Size() int {
return len(t.topics)
}

// Get returns the topic list at the given index from the slice.
func (t *Topics) Get(index int) (hashes *Hashes, _ error) {
if index < 0 || index >= len(t.topics) {
return nil, errors.New("index out of bounds")
}
return &Hashes{t.topics[index]}, nil
}

// Set sets the topic list at the given index in the slice.
func (t *Topics) Set(index int, topics *Hashes) error {
if index < 0 || index >= len(t.topics) {
return errors.New("index out of bounds")
}
t.topics[index] = topics.hashes
return nil
}

// Append adds a new topic list to the end of the slice.
func (t *Topics) Append(topics *Hashes) {
t.topics = append(t.topics, topics.hashes)
}
```

```
// FilterQuery contains options for contact log filtering.
type FilterQuery struct {
query ethereum.FilterQuery
}

// NewFilterQuery creates an empty filter query for contact log filtering.
func NewFilterQuery() *FilterQuery {
return new(FilterQuery)
}

func (fq *FilterQuery) GetFromBlock() *BigInt    { return &BigInt{fq.query.FromBlock} }
func (fq *FilterQuery) GetToBlock() *BigInt      { return &BigInt{fq.query.ToBlock} }
func (fq *FilterQuery) GetAddresses() *Addresses { return &Addresses{fq.query.Addresses} }
func (fq *FilterQuery) GetTopics() *Topics       { return &Topics{fq.query.Topics} }

func (fq *FilterQuery) SetFromBlock(fromBlock *BigInt)    { fq.query.FromBlock = fromBlock.bigint }
func (fq *FilterQuery) SetToBlock(toBlock *BigInt)        { fq.query.ToBlock = toBlock.bigint }
func (fq *FilterQuery) SetAddresses(addresses *Addresses) { fq.query.Addresses =
addresses.addresses }
func (fq *FilterQuery) SetTopics(topics *Topics)          { fq.query.Topics = topics.topics }

57:F:\git\coin\ethereum\go-ethereum\mobile\geth.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains all the wrappers from the node package to support client side node
// management on mobile platforms.

package geth

import (
"encoding/json"
"fmt"
"path/filepath"

"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/eth"
"github.com/ethereum/go-ethereum/eth/downloader"
"github.com/ethereum/go-ethereum/ethclient"
"github.com/ethereum/go-ethereum/ethstats"
"github.com/ethereum/go-ethereum/les"
"github.com/ethereum/go-ethereum/node"
"github.com/ethereum/go-ethereum/p2p"
```

```
"github.com/ethereum/go-ethereum/p2p/nat"
"github.com/ethereum/go-ethereum/params"
whisper "github.com/ethereum/go-ethereum/whisper/whisperv5"
)

// NodeConfig represents the collection of configuration values to fine tune the Geth
// node embedded into a mobile process. The available values are a subset of the
// entire API provided by go-ethereum to reduce the maintenance surface and dev
// complexity.
type NodeConfig struct {
// Bootstrap nodes used to establish connectivity with the rest of the network.
BootstrapNodes *Enodes

// MaxPeers is the maximum number of peers that can be connected. If this is
// set to zero, then only the configured static and trusted peers can connect.
MaxPeers int

// EthereumEnabled specifies whether the node should run the Ethereum protocol.
EthereumEnabled bool

// EthereumNetworkID is the network identifier used by the Ethereum protocol to
// decide if remote peers should be accepted or not.
EthereumNetworkID int64 // uint64 in truth, but Java can't handle that...

// EthereumGenesis is the genesis JSON to use to seed the blockchain with. An
// empty genesis state is equivalent to using the mainnet's state.
EthereumGenesis string

// EthereumDatabaseCache is the system memory in MB to allocate for database caching.
// A minimum of 16MB is always reserved.
EthereumDatabaseCache int

// EthereumNetStats is a netstats connection string to use to report various
// chain, transaction and node stats to a monitoring server.
//
// It has the form "nodename:secret@host:port"
EthereumNetStats string

// WhisperEnabled specifies whether the node should run the Whisper protocol.
WhisperEnabled bool
}
```

```
// defaultNodeConfig contains the default node configuration values to use if all
// or some fields are missing from the user's specified list.
var defaultNodeConfig = &NodeConfig{
BootstrapNodes:        FoundationBootnodes(),
MaxPeers:              25,
EthereumEnabled:       true,
EthereumNetworkID:     1,
EthereumDatabaseCache: 16,
}

// NewNodeConfig creates a new node option set, initialized to the default values.
func NewNodeConfig() *NodeConfig {
config := *defaultNodeConfig
return &config
}

// Node represents a Geth Ethereum node instance.
type Node struct {
node *node.Node
}

// NewNode creates and configures a new Geth node.
func NewNode(datadir string, config *NodeConfig) (stack *Node, _ error) {
// If no or partial configurations were specified, use defaults
if config == nil {
config = NewNodeConfig()
}
if config.MaxPeers == 0 {
config.MaxPeers = defaultNodeConfig.MaxPeers
}
if config.BootstrapNodes == nil || config.BootstrapNodes.Size() == 0 {
config.BootstrapNodes = defaultNodeConfig.BootstrapNodes
}
// Create the empty networking stack
nodeConf := &node.Config{
Name:        clientIdentifier,
Version:     params.Version,
DataDir:     datadir,
KeyStoreDir: filepath.Join(datadir, "keystore"), // Mobile should never use internal keystores!
P2P: p2p.Config{
NoDiscovery:      true,
DiscoveryV5:      true,
```

```
DiscoveryV5Addr:  ":0",
BootstrapNodesV5: config.BootstrapNodes.nodes,
ListenAddr:       ":0",
NAT:              nat.Any(),
MaxPeers:         config.MaxPeers,
},
}
rawStack, err := node.New(nodeConf)
if err != nil {
return nil, err
}

var genesis *core.Genesis
if config.EthereumGenesis != "" {
// Parse the user supplied genesis spec if not mainnet
genesis = new(core.Genesis)
if err := json.Unmarshal([]byte(config.EthereumGenesis), genesis); err != nil {
return nil, fmt.Errorf("invalid genesis spec: %v", err)
}
// If we have the testnet, hard code the chain configs too
if config.EthereumGenesis == TestnetGenesis() {
genesis.Config = params.TestnetChainConfig
if config.EthereumNetworkID == 1 {
config.EthereumNetworkID = 3
}
}
}
// Register the Ethereum protocol if requested
if config.EthereumEnabled {
ethConf := eth.DefaultConfig
ethConf.Genesis = genesis
ethConf.SyncMode = downloader.LightSync
ethConf.NetworkId = uint64(config.EthereumNetworkID)
ethConf.DatabaseCache = config.EthereumDatabaseCache
if err := rawStack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
return les.New(ctx, &ethConf)
}); err != nil {
return nil, fmt.Errorf("ethereum init: %v", err)
}
// If netstats reporting is requested, do it
if config.EthereumNetStats != "" {
if err := rawStack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
```

```
var lesServ *les.LightEthereum
ctx.Service(&lesServ)

return ethstats.New(config.EthereumNetStats, nil, lesServ)
}); err != nil {
return nil, fmt.Errorf("netstats init: %v", err)
}
}
}
// Register the Whisper protocol if requested
if config.WhisperEnabled {
if err := rawStack.Register(func(*node.ServiceContext) (node.Service, error) {
return whisper.New(&whisper.DefaultConfig), nil
}); err != nil {
return nil, fmt.Errorf("whisper init: %v", err)
}
}
return &Node{rawStack}, nil
}

// Start creates a live P2P node and starts running it.
func (n *Node) Start() error {
return n.node.Start()
}

// Stop terminates a running node along with all it's services. In the node was
// not started, an error is returned.
func (n *Node) Stop() error {
return n.node.Stop()
}

// GetEthereumClient retrieves a client to access the Ethereum subsystem.
func (n *Node) GetEthereumClient() (client *EthereumClient, _ error) {
rpc, err := n.node.Attach()
if err != nil {
return nil, err
}
return &EthereumClient{ethclient.NewClient(rpc)}, nil
}

// GetNodeInfo gathers and returns a collection of metadata known about the host.
func (n *Node) GetNodeInfo() *NodeInfo {
```

```
return &NodeInfo{n.node.Server().NodeInfo()}
}

// GetPeersInfo returns an array of metadata objects describing connected peers.
func (n *Node) GetPeersInfo() *PeerInfos {
return &PeerInfos{n.node.Server().PeersInfo()}
}

58:F:\git\coin\ethereum\go-ethereum\mobile\geth_android.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// +build android

package geth

// clientIdentifier is a hard coded identifier to report into the network.
var clientIdentifier = "GethDroid"

59:F:\git\coin\ethereum\go-ethereum\mobile\geth_ios.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// +build ios

package geth

// clientIdentifier is a hard coded identifier to report into the network.
var clientIdentifier = "iGeth"

60:F:\git\coin\ethereum\go-ethereum\mobile\geth_other.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// +build !android,!ios

package geth

// clientIdentifier is a hard coded identifier to report into the network.
var clientIdentifier = "GethMobile"

61:F:\git\coin\ethereum\go-ethereum\mobile\init.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains initialization code for the mbile library.
```

```
package geth

import (
"os"
"runtime"

"github.com/ethereum/go-ethereum/log"
)

func init() {
// Initialize the logger
log.Root().SetHandler(log.LvlFilterHandler(log.LvlInfo, log.StreamHandler(os.Stderr,
log.TerminalFormat(false))))

// Initialize the goroutine count
runtime.GOMAXPROCS(runtime.NumCPU())
}

62:F:\git\coin\ethereum\go-ethereum\mobile\interface.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains perverted wrappers to allow crossing over empty interfaces.

package geth

import (
"errors"
"math/big"

"github.com/ethereum/go-ethereum/common"
)

// Interface represents a wrapped version of Go's interface{}, with the capacity
// to store arbitrary data types.
//
// Since it's impossible to get the arbitrary-ness converted between Go and mobile
// platforms, we're using explicit getters and setters for the conversions. There
// is of course no point in enumerating everything, just enough to support the
// contract bindins requiring client side generated code.
type Interface struct {
object interface{}
```

```
}

// NewInterface creates a new empty interface that can be used to pass around
// generic types.
func NewInterface() *Interface {
return new(Interface)
}

func (i *Interface) SetBool(b bool)               { i.object = &b }
func (i *Interface) SetBools(bs []bool)           { i.object = &bs }
func (i *Interface) SetString(str string)         { i.object = &str }
func (i *Interface) SetStrings(strs *Strings)     { i.object = &strs.strs }
func (i *Interface) SetBinary(binary []byte)      { i.object = &binary }
func (i *Interface) SetBinaries(binaries [][]byte) { i.object = &binaries }
func (i *Interface) SetAddress(address *Address)  { i.object = &address.address }
func (i *Interface) SetAddresses(addrs *Addresses) { i.object = &addrs.addresses }
func (i *Interface) SetHash(hash *Hash)           { i.object = &hash.hash }
func (i *Interface) SetHashes(hashes *Hashes)     { i.object = &hashes.hashes }
func (i *Interface) SetInt8(n int8)               { i.object = &n }
func (i *Interface) SetInt16(n int16)             { i.object = &n }
func (i *Interface) SetInt32(n int32)             { i.object = &n }
func (i *Interface) SetInt64(n int64)             { i.object = &n }
func (i *Interface) SetUint8(bigint *BigInt)      { n := uint8(bigint.bigint.Uint64()); i.object = &n }
func (i *Interface) SetUint16(bigint *BigInt)     { n := uint16(bigint.bigint.Uint64()); i.object = &n }
func (i *Interface) SetUint32(bigint *BigInt)     { n := uint32(bigint.bigint.Uint64()); i.object = &n }
func (i *Interface) SetUint64(bigint *BigInt)     { n := uint64(bigint.bigint.Uint64()); i.object = &n }
func (i *Interface) SetBigInt(bigint *BigInt)     { i.object = &bigint.bigint }
func (i *Interface) SetBigInts(bigints *BigInts)  { i.object = &bigints.bigints }

func (i *Interface) SetDefaultBool()      { i.object = new(bool) }
func (i *Interface) SetDefaultBools()     { i.object = new([]bool) }
func (i *Interface) SetDefaultString()    { i.object = new(string) }
func (i *Interface) SetDefaultStrings()   { i.object = new([]string) }
func (i *Interface) SetDefaultBinary()    { i.object = new([]byte) }
func (i *Interface) SetDefaultBinaries()  { i.object = new([][]byte) }
func (i *Interface) SetDefaultAddress()   { i.object = new(common.Address) }
func (i *Interface) SetDefaultAddresses() { i.object = new([]common.Address) }
func (i *Interface) SetDefaultHash()      { i.object = new(common.Hash) }
func (i *Interface) SetDefaultHashes()    { i.object = new([]common.Hash) }
func (i *Interface) SetDefaultInt8()      { i.object = new(int8) }
func (i *Interface) SetDefaultInt16()     { i.object = new(int16) }
func (i *Interface) SetDefaultInt32()     { i.object = new(int32) }
```

```
func (i *Interface) SetDefaultInt64()     { i.object = new(int64) }
func (i *Interface) SetDefaultUint8()     { i.object = new(uint8) }
func (i *Interface) SetDefaultUint16()    { i.object = new(uint16) }
func (i *Interface) SetDefaultUint32()    { i.object = new(uint32) }
func (i *Interface) SetDefaultUint64()    { i.object = new(uint64) }
func (i *Interface) SetDefaultBigInt()    { i.object = new(*big.Int) }
func (i *Interface) SetDefaultBigInts()   { i.object = new([]*big.Int) }

func (i *Interface) GetBool() bool            { return *i.object.(*bool) }
func (i *Interface) GetBools() []bool         { return *i.object.(*[]bool) }
func (i *Interface) GetString() string        { return *i.object.(*string) }
func (i *Interface) GetStrings() *Strings     { return &Strings{*i.object.(*[]string)} }
func (i *Interface) GetBinary() []byte        { return *i.object.(*[]byte) }
func (i *Interface) GetBinaries() [][]byte    { return *i.object.(*[][]byte) }
func (i *Interface) GetAddress() *Address     { return &Address{*i.object.(*common.Address)} }
func (i *Interface) GetAddresses() *Addresses { return &Addresses{*i.object.(*[]common.Address)}
}
func (i *Interface) GetHash() *Hash           { return &Hash{*i.object.(*common.Hash)} }
func (i *Interface) GetHashes() *Hashes       { return &Hashes{*i.object.(*[]common.Hash)} }
func (i *Interface) GetInt8() int8            { return *i.object.(*int8) }
func (i *Interface) GetInt16() int16          { return *i.object.(*int16) }
func (i *Interface) GetInt32() int32          { return *i.object.(*int32) }
func (i *Interface) GetInt64() int64          { return *i.object.(*int64) }
func (i *Interface) GetUint8() *BigInt {
return &BigInt{new(big.Int).SetUint64(uint64(*i.object.(*uint8)))}
}
func (i *Interface) GetUint16() *BigInt {
return &BigInt{new(big.Int).SetUint64(uint64(*i.object.(*uint16)))}
}
func (i *Interface) GetUint32() *BigInt {
return &BigInt{new(big.Int).SetUint64(uint64(*i.object.(*uint32)))}
}
func (i *Interface) GetUint64() *BigInt {
return &BigInt{new(big.Int).SetUint64(*i.object.(*uint64))}
}
func (i *Interface) GetBigInt() *BigInt   { return &BigInt{*i.object.(**big.Int)} }
func (i *Interface) GetBigInts() *BigInts { return &BigInts{*i.object.(*[]*big.Int)} }

// Interfaces is a slices of wrapped generic objects.
type Interfaces struct {
objects []interface{}
}
```

```
// NewInterfaces creates a slice of uninitialized interfaces.
func NewInterfaces(size int) *Interfaces {
return &Interfaces{
objects: make([]interface{}, size),
}
}

// Size returns the number of interfaces in the slice.
func (i *Interfaces) Size() int {
return len(i.objects)
}

// Get returns the bigint at the given index from the slice.
func (i *Interfaces) Get(index int) (iface *Interface, _ error) {
if index < 0 || index >= len(i.objects) {
return nil, errors.New("index out of bounds")
}
return &Interface{i.objects[index]}, nil
}

// Set sets the big int at the given index in the slice.
func (i *Interfaces) Set(index int, object *Interface) error {
if index < 0 || index >= len(i.objects) {
return errors.New("index out of bounds")
}
i.objects[index] = object.object
return nil
}

63:F:\git\coin\ethereum\go-ethereum\mobile\logger.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package geth

import (
"os"

"github.com/ethereum/go-ethereum/log"
)

// SetVerbosity sets the global verbosity level (between 0 and 6 - see logger/verbosity.go).
```

```
func SetVerbosity(level int) {
log.Root().SetHandler(log.LvlFilterHandler(log.Lvl(level), log.StreamHandler(os.Stderr,
log.TerminalFormat(false))))
}

64:F:\git\coin\ethereum\go-ethereum\mobile\p2p.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains wrappers for the p2p package.

package geth

import (
"errors"

"github.com/ethereum/go-ethereum/p2p"
)

// NodeInfo represents pi short summary of the information known about the host.
type NodeInfo struct {
info *p2p.NodeInfo
}

func (ni *NodeInfo) GetID() string             { return ni.info.ID }
func (ni *NodeInfo) GetName() string           { return ni.info.Name }
func (ni *NodeInfo) GetEnode() string          { return ni.info.Enode }
func (ni *NodeInfo) GetIP() string             { return ni.info.IP }
func (ni *NodeInfo) GetDiscoveryPort() int     { return ni.info.Ports.Discovery }
func (ni *NodeInfo) GetListenerPort() int      { return ni.info.Ports.Listener }
func (ni *NodeInfo) GetListenerAddress() string { return ni.info.ListenAddr }
func (ni *NodeInfo) GetProtocols() *Strings {
protos := []string{}
for proto := range ni.info.Protocols {
protos = append(protos, proto)
}
return &Strings{protos}
}

// PeerInfo represents pi short summary of the information known about pi connected peer.
type PeerInfo struct {
info *p2p.PeerInfo
}
```

```
func (pi *PeerInfo) GetID() string            { return pi.info.ID }
func (pi *PeerInfo) GetName() string          { return pi.info.Name }
func (pi *PeerInfo) GetCaps() *Strings        { return &Strings{pi.info.Caps} }
func (pi *PeerInfo) GetLocalAddress() string  { return pi.info.Network.LocalAddress }
func (pi *PeerInfo) GetRemoteAddress() string { return pi.info.Network.RemoteAddress }

// PeerInfos represents a slice of infos about remote peers.
type PeerInfos struct {
infos []*p2p.PeerInfo
}

// Size returns the number of peer info entries in the slice.
func (pi *PeerInfos) Size() int {
return len(pi.infos)
}

// Get returns the peer info at the given index from the slice.
func (pi *PeerInfos) Get(index int) (info *PeerInfo, _ error) {
if index < 0 || index >= len(pi.infos) {
return nil, errors.New("index out of bounds")
}
return &PeerInfo{pi.infos[index]}, nil
}

65:F:\git\coin\ethereum\go-ethereum\mobile\params.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains all the wrappers from the params package.

package geth

import (
"encoding/json"

"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/p2p/discv5"
"github.com/ethereum/go-ethereum/params"
)

// MainnetGenesis returns the JSON spec to use for the main Ethereum network. It
// is actually empty since that defaults to the hard coded binary genesis block.
```

```
func MainnetGenesis() string {
return ""
}

// TestnetGenesis returns the JSON spec to use for the Ethereum test network.
func TestnetGenesis() string {
enc, err := json.Marshal(core.DefaultTestnetGenesisBlock())
if err != nil {
panic(err)
}
return string(enc)
}

// FoundationBootnodes returns the enode URLs of the P2P bootstrap nodes operated
// by the foundation running the V5 discovery protocol.
func FoundationBootnodes() *Enodes {
nodes := &Enodes{nodes: make([]*discv5.Node, len(params.DiscoveryV5Bootnodes))}
for i, url := range params.DiscoveryV5Bootnodes {
nodes.nodes[i] = discv5.MustParseNode(url)
}
return nodes
}

66:F:\git\coin\ethereum\go-ethereum\mobile\primitives.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains various wrappers for primitive types.

package geth

import (
"errors"
"fmt"
)

// Strings represents s slice of strs.
type Strings struct{ strs []string }

// Size returns the number of strs in the slice.
func (s *Strings) Size() int {
return len(s.strs)
}
```

```
// Get returns the string at the given index from the slice.
func (s *Strings) Get(index int) (str string, _ error) {
if index < 0 || index >= len(s.strs) {
return "", errors.New("index out of bounds")
}
return s.strs[index], nil
}

// Set sets the string at the given index in the slice.
func (s *Strings) Set(index int, str string) error {
if index < 0 || index >= len(s.strs) {
return errors.New("index out of bounds")
}
s.strs[index] = str
return nil
}

// String implements the Stringer interface.
func (s *Strings) String() string {
return fmt.Sprintf("%v", s.strs)
}

67:F:\git\coin\ethereum\go-ethereum\mobile\types.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains all the wrappers from the core/types package.

package geth

import (
"encoding/json"
"errors"
"fmt"

"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/rlp"
)

// A Nonce is a 64-bit hash which proves (combined with the mix-hash) that
// a sufficient amount of computation has been carried out on a block.
type Nonce struct {
```

```
nonce types.BlockNonce
}

// GetBytes retrieves the byte representation of the block nonce.
func (n *Nonce) GetBytes() []byte {
return n.nonce[:]
}

// GetHex retrieves the hex string representation of the block nonce.
func (n *Nonce) GetHex() string {
return fmt.Sprintf("0x%x", n.nonce[:])
}

// Bloom represents a 256 bit bloom filter.
type Bloom struct {
bloom types.Bloom
}

// GetBytes retrieves the byte representation of the bloom filter.
func (b *Bloom) GetBytes() []byte {
return b.bloom[:]
}

// GetHex retrieves the hex string representation of the bloom filter.
func (b *Bloom) GetHex() string {
return fmt.Sprintf("0x%x", b.bloom[:])
}

// Header represents a block header in the Ethereum blockchain.
type Header struct {
header *types.Header
}

// NewHeaderFromRLP parses a header from an RLP data dump.
func NewHeaderFromRLP(data []byte) (*Header, error) {
h := &Header{
header: new(types.Header),
}
if err := rlp.DecodeBytes(data, h.header); err != nil {
return nil, err
}
return h, nil
```

```
}

// EncodeRLP encodes a header into an RLP data dump.
func (h *Header) EncodeRLP() ([]byte, error) {
return rlp.EncodeToBytes(h.header)
}

// NewHeaderFromJSON parses a header from an JSON data dump.
func NewHeaderFromJSON(data string) (*Header, error) {
h := &Header{
header: new(types.Header),
}
if err := json.Unmarshal([]byte(data), h.header); err != nil {
return nil, err
}
return h, nil
}

// EncodeJSON encodes a header into an JSON data dump.
func (h *Header) EncodeJSON() (string, error) {
data, err := json.Marshal(h.header)
return string(data), err
}

// String implements the fmt.Stringer interface to print some semi-meaningful
// data dump of the header for debugging purposes.
func (h *Header) String() string {
return h.header.String()
}

func (h *Header) GetParentHash() *Hash   { return &Hash{h.header.ParentHash} }
func (h *Header) GetUncleHash() *Hash    { return &Hash{h.header.UncleHash} }
func (h *Header) GetCoinbase() *Address  { return &Address{h.header.Coinbase} }
func (h *Header) GetRoot() *Hash         { return &Hash{h.header.Root} }
func (h *Header) GetTxHash() *Hash       { return &Hash{h.header.TxHash} }
func (h *Header) GetReceiptHash() *Hash  { return &Hash{h.header.ReceiptHash} }
func (h *Header) GetBloom() *Bloom       { return &Bloom{h.header.Bloom} }
func (h *Header) GetDifficulty() *BigInt { return &BigInt{h.header.Difficulty} }
func (h *Header) GetNumber() int64       { return h.header.Number.Int64() }
func (h *Header) GetGasLimit() int64     { return h.header.GasLimit.Int64() }
func (h *Header) GetGasUsed() int64      { return h.header.GasUsed.Int64() }
func (h *Header) GetTime() int64         { return h.header.Time.Int64() }
```

```
func (h *Header) GetExtra() []byte       { return h.header.Extra }
func (h *Header) GetMixDigest() *Hash    { return &Hash{h.header.MixDigest} }
func (h *Header) GetNonce() *Nonce       { return &Nonce{h.header.Nonce} }
func (h *Header) GetHash() *Hash         { return &Hash{h.header.Hash()} }

// Headers represents a slice of headers.
type Headers struct{ headers []*types.Header }

// Size returns the number of headers in the slice.
func (h *Headers) Size() int {
return len(h.headers)
}

// Get returns the header at the given index from the slice.
func (h *Headers) Get(index int) (header *Header, _ error) {
if index < 0 || index >= len(h.headers) {
return nil, errors.New("index out of bounds")
}
return &Header{h.headers[index]}, nil
}

// Block represents an entire block in the Ethereum blockchain.
type Block struct {
block *types.Block
}

// NewBlockFromRLP parses a block from an RLP data dump.
func NewBlockFromRLP(data []byte) (*Block, error) {
b := &Block{
block: new(types.Block),
}
if err := rlp.DecodeBytes(data, b.block); err != nil {
return nil, err
}
return b, nil
}

// EncodeRLP encodes a block into an RLP data dump.
func (b *Block) EncodeRLP() ([]byte, error) {
return rlp.EncodeToBytes(b.block)
}
```

```
// NewBlockFromJSON parses a block from an JSON data dump.
func NewBlockFromJSON(data string) (*Block, error) {
b := &Block{
block: new(types.Block),
}
if err := json.Unmarshal([]byte(data), b.block); err != nil {
return nil, err
}
return b, nil
}

// EncodeJSON encodes a block into an JSON data dump.
func (b *Block) EncodeJSON() (string, error) {
data, err := json.Marshal(b.block)
return string(data), err
}

// String implements the fmt.Stringer interface to print some semi-meaningful
// data dump of the block for debugging purposes.
func (b *Block) String() string {
return b.block.String()
}

func (b *Block) GetParentHash() *Hash   { return &Hash{b.block.ParentHash()} }
func (b *Block) GetUncleHash() *Hash    { return &Hash{b.block.UncleHash()} }
func (b *Block) GetCoinbase() *Address  { return &Address{b.block.Coinbase()} }
func (b *Block) GetRoot() *Hash         { return &Hash{b.block.Root()} }
func (b *Block) GetTxHash() *Hash       { return &Hash{b.block.TxHash()} }
func (b *Block) GetReceiptHash() *Hash  { return &Hash{b.block.ReceiptHash()} }
func (b *Block) GetBloom() *Bloom       { return &Bloom{b.block.Bloom()} }
func (b *Block) GetDifficulty() *BigInt { return &BigInt{b.block.Difficulty()} }
func (b *Block) GetNumber() int64       { return b.block.Number().Int64() }
func (b *Block) GetGasLimit() int64     { return b.block.GasLimit().Int64() }
func (b *Block) GetGasUsed() int64      { return b.block.GasUsed().Int64() }
func (b *Block) GetTime() int64         { return b.block.Time().Int64() }
func (b *Block) GetExtra() []byte       { return b.block.Extra() }
func (b *Block) GetMixDigest() *Hash    { return &Hash{b.block.MixDigest()} }
func (b *Block) GetNonce() int64        { return int64(b.block.Nonce()) }

func (b *Block) GetHash() *Hash        { return &Hash{b.block.Hash()} }
func (b *Block) GetHashNoNonce() *Hash { return &Hash{b.block.HashNoNonce()} }
```

```
func (b *Block) GetHeader() *Header           { return &Header{b.block.Header()} }
func (b *Block) GetUncles() *Headers          { return &Headers{b.block.Uncles()} }
func (b *Block) GetTransactions() *Transactions { return &Transactions{b.block.Transactions()} }
func (b *Block) GetTransaction(hash *Hash) *Transaction {
return &Transaction{b.block.Transaction(hash.hash)}
}

// Transaction represents a single Ethereum transaction.
type Transaction struct {
tx *types.Transaction
}

// NewTransaction creates a new transaction with the given properties.
func NewTransaction(nonce int64, to *Address, amount, gasLimit, gasPrice *BigInt, data []byte)
*Transaction {
return &Transaction{types.NewTransaction(uint64(nonce), to.address, amount.bigint,
gasLimit.bigint, gasPrice.bigint, data)}
}

// NewTransactionFromRLP parses a transaction from an RLP data dump.
func NewTransactionFromRLP(data []byte) (*Transaction, error) {
tx := &Transaction{
tx: new(types.Transaction),
}
if err := rlp.DecodeBytes(data, tx.tx); err != nil {
return nil, err
}
return tx, nil
}

// EncodeRLP encodes a transaction into an RLP data dump.
func (tx *Transaction) EncodeRLP() ([]byte, error) {
return rlp.EncodeToBytes(tx.tx)
}

// NewTransactionFromJSON parses a transaction from an JSON data dump.
func NewTransactionFromJSON(data string) (*Transaction, error) {
tx := &Transaction{
tx: new(types.Transaction),
}
if err := json.Unmarshal([]byte(data), tx.tx); err != nil {
return nil, err
```

```
}
return tx, nil
}

// EncodeJSON encodes a transaction into an JSON data dump.
func (tx *Transaction) EncodeJSON() (string, error) {
data, err := json.Marshal(tx.tx)
return string(data), err
}

// String implements the fmt.Stringer interface to print some semi-meaningful
// data dump of the transaction for debugging purposes.
func (tx *Transaction) String() string {
return tx.tx.String()
}

func (tx *Transaction) GetData() []byte      { return tx.tx.Data() }
func (tx *Transaction) GetGas() int64        { return tx.tx.Gas().Int64() }
func (tx *Transaction) GetGasPrice() *BigInt { return &BigInt{tx.tx.GasPrice()} }
func (tx *Transaction) GetValue() *BigInt    { return &BigInt{tx.tx.Value()} }
func (tx *Transaction) GetNonce() int64      { return int64(tx.tx.Nonce()) }

func (tx *Transaction) GetHash() *Hash    { return &Hash{tx.tx.Hash()} }
func (tx *Transaction) GetSigHash() *Hash { return
&Hash{tx.tx.SigHash(types.HomesteadSigner{})} }
func (tx *Transaction) GetCost() *BigInt  { return &BigInt{tx.tx.Cost()} }

func (tx *Transaction) GetFrom(chainID *BigInt) (address *Address, _ error) {
if chainID == nil { // Null passed from mobile app
chainID = new(BigInt)
}
from, err := types.Sender(types.NewEIP155Signer(chainID.bigint), tx.tx)
return &Address{from}, err
}

func (tx *Transaction) GetTo() *Address {
if to := tx.tx.To(); to != nil {
return &Address{*to}
}
return nil
}
```

```
func (tx *Transaction) WithSignature(sig []byte) (signedTx *Transaction, _ error) {
rawTx, err := tx.tx.WithSignature(types.HomesteadSigner{}, sig)
return &Transaction{rawTx}, err
}

// Transactions represents a slice of transactions.
type Transactions struct{ txs types.Transactions }

// Size returns the number of transactions in the slice.
func (txs *Transactions) Size() int {
return len(txs.txs)
}

// Get returns the transaction at the given index from the slice.
func (txs *Transactions) Get(index int) (tx *Transaction, _ error) {
if index < 0 || index >= len(txs.txs) {
return nil, errors.New("index out of bounds")
}
return &Transaction{txs.txs[index]}, nil
}

// Receipt represents the results of a transaction.
type Receipt struct {
receipt *types.Receipt
}

// NewReceiptFromRLP parses a transaction receipt from an RLP data dump.
func NewReceiptFromRLP(data []byte) (*Receipt, error) {
r := &Receipt{
receipt: new(types.Receipt),
}
if err := rlp.DecodeBytes(data, r.receipt); err != nil {
return nil, err
}
return r, nil
}

// EncodeRLP encodes a transaction receipt into an RLP data dump.
func (r *Receipt) EncodeRLP() ([]byte, error) {
return rlp.EncodeToBytes(r.receipt)
}
```

```
// NewReceiptFromJSON parses a transaction receipt from an JSON data dump.
func NewReceiptFromJSON(data string) (*Receipt, error) {
r := &Receipt{
receipt: new(types.Receipt),
}
if err := json.Unmarshal([]byte(data), r.receipt); err != nil {
return nil, err
}
return r, nil
}

// EncodeJSON encodes a transaction receipt into an JSON data dump.
func (r *Receipt) EncodeJSON() (string, error) {
data, err := rlp.EncodeToBytes(r.receipt)
return string(data), err
}

// String implements the fmt.Stringer interface to print some semi-meaningful
// data dump of the transaction receipt for debugging purposes.
func (r *Receipt) String() string {
return r.receipt.String()
}

func (r *Receipt) GetPostState() []byte          { return r.receipt.PostState }
func (r *Receipt) GetCumulativeGasUsed() *BigInt { return &BigInt{r.receipt.CumulativeGasUsed} }
func (r *Receipt) GetBloom() *Bloom              { return &Bloom{r.receipt.Bloom} }
func (r *Receipt) GetLogs() *Logs                { return &Logs{r.receipt.Logs} }
func (r *Receipt) GetTxHash() *Hash              { return &Hash{r.receipt.TxHash} }
func (r *Receipt) GetContractAddress() *Address  { return &Address{r.receipt.ContractAddress} }
func (r *Receipt) GetGasUsed() *BigInt           { return &BigInt{r.receipt.GasUsed} }

68:F:\git\coin\ethereum\go-ethereum\mobile\vm.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains all the wrappers from the core/types package.

package geth

import (
"errors"

"github.com/ethereum/go-ethereum/core/types"
```

```
)

// Log represents a contract log event. These events are generated by the LOG
// opcode and stored/indexed by the node.
type Log struct {
log *types.Log
}

func (l *Log) GetAddress() *Address  { return &Address{l.log.Address} }
func (l *Log) GetTopics() *Hashes    { return &Hashes{l.log.Topics} }
func (l *Log) GetData() []byte       { return l.log.Data }
func (l *Log) GetBlockNumber() int64 { return int64(l.log.BlockNumber) }
func (l *Log) GetTxHash() *Hash      { return &Hash{l.log.TxHash} }
func (l *Log) GetTxIndex() int       { return int(l.log.TxIndex) }
func (l *Log) GetBlockHash() *Hash   { return &Hash{l.log.BlockHash} }
func (l *Log) GetIndex() int         { return int(l.log.Index) }

// Logs represents a slice of VM logs.
type Logs struct{ logs []*types.Log }

// Size returns the number of logs in the slice.
func (l *Logs) Size() int {
return len(l.logs)
}

// Get returns the log at the given index from the slice.
func (l *Logs) Get(index int) (log *Log, _ error) {
if index < 0 || index >= len(l.logs) {
return nil, errors.New("index out of bounds")
}
return &Log{l.logs[index]}, nil
}

69:F:\git\coin\ethereum\go-ethereum\node\api.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package node

import (
"fmt"
"strings"
"time"
```

```
"github.com/ethereum/go-ethereum/common/hexutil"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/p2p"
"github.com/ethereum/go-ethereum/p2p/discover"
"github.com/rcrowley/go-metrics"
)

// PrivateAdminAPI is the collection of administrative API methods exposed only
// over a secure RPC channel.
type PrivateAdminAPI struct {
node *Node // Node interfaced by this API
}

// NewPrivateAdminAPI creates a new API definition for the private admin methods
// of the node itself.
func NewPrivateAdminAPI(node *Node) *PrivateAdminAPI {
return &PrivateAdminAPI{node: node}
}

// AddPeer requests connecting to a remote node, and also maintaining the new
// connection at all times, even reconnecting if it is lost.
func (api *PrivateAdminAPI) AddPeer(url string) (bool, error) {
// Make sure the server is running, fail otherwise
server := api.node.Server()
if server == nil {
return false, ErrNodeStopped
}
// Try to add the url as a static peer and return
node, err := discover.ParseNode(url)
if err != nil {
return false, fmt.Errorf("invalid enode: %v", err)
}
server.AddPeer(node)
return true, nil
}

// RemovePeer disconnects from a a remote node if the connection exists
func (api *PrivateAdminAPI) RemovePeer(url string) (bool, error) {
// Make sure the server is running, fail otherwise
server := api.node.Server()
if server == nil {
```

```
return false, ErrNodeStopped
}
// Try to remove the url as a static peer and return
node, err := discover.ParseNode(url)
if err != nil {
return false, fmt.Errorf("invalid enode: %v", err)
}
server.RemovePeer(node)
return true, nil
}

// StartRPC starts the HTTP RPC API server.
func (api *PrivateAdminAPI) StartRPC(host *string, port *int, cors *string, apis *string) (bool, error)
{
api.node.lock.Lock()
defer api.node.lock.Unlock()

if api.node.httpHandler != nil {
return false, fmt.Errorf("HTTP RPC already running on %s", api.node.httpEndpoint)
}

if host == nil {
h := DefaultHTTPHost
if api.node.config.HTTPHost != "" {
h = api.node.config.HTTPHost
}
host = &h
}
if port == nil {
port = &api.node.config.HTTPPort
}

allowedOrigins := api.node.config.HTTPCors
if cors != nil {
allowedOrigins = nil
for _, origin := range strings.Split(*cors, ",") {
allowedOrigins = append(allowedOrigins, strings.TrimSpace(origin))
}
}

modules := api.node.httpWhitelist
if apis != nil {
```

```
modules = nil
for _, m := range strings.Split(*apis, ",") {
modules = append(modules, strings.TrimSpace(m))
}
}

if err := api.node.startHTTP(fmt.Sprintf("%s:%d", *host, *port), api.node.rpcAPIs, modules,
allowedOrigins); err != nil {
return false, err
}
return true, nil
}

// StopRPC terminates an already running HTTP RPC API endpoint.
func (api *PrivateAdminAPI) StopRPC() (bool, error) {
api.node.lock.Lock()
defer api.node.lock.Unlock()

if api.node.httpHandler == nil {
return false, fmt.Errorf("HTTP RPC not running")
}
api.node.stopHTTP()
return true, nil
}

// StartWS starts the websocket RPC API server.
func (api *PrivateAdminAPI) StartWS(host *string, port *int, allowedOrigins *string, apis *string)
(bool, error) {
api.node.lock.Lock()
defer api.node.lock.Unlock()

if api.node.wsHandler != nil {
return false, fmt.Errorf("WebSocket RPC already running on %s", api.node.wsEndpoint)
}

if host == nil {
h := DefaultWSHost
if api.node.config.WSHost != "" {
h = api.node.config.WSHost
}
host = &h
}
```

```
if port == nil {
port = &api.node.config.WSPort
}

origins := api.node.config.WSOrigins
if allowedOrigins != nil {
origins = nil
for _, origin := range strings.Split(*allowedOrigins, ",") {
origins = append(origins, strings.TrimSpace(origin))
}
}

modules := api.node.config.WSModules
if apis != nil {
modules = nil
for _, m := range strings.Split(*apis, ",") {
modules = append(modules, strings.TrimSpace(m))
}
}

if err := api.node.startWS(fmt.Sprintf("%s:%d", *host, *port), api.node.rpcAPIs, modules, origins);
err != nil {
return false, err
}
return true, nil
}

// StopRPC terminates an already running websocket RPC API endpoint.
func (api *PrivateAdminAPI) StopWS() (bool, error) {
api.node.lock.Lock()
defer api.node.lock.Unlock()

if api.node.wsHandler == nil {
return false, fmt.Errorf("WebSocket RPC not running")
}
api.node.stopWS()
return true, nil
}

// PublicAdminAPI is the collection of administrative API methods exposed over
// both secure and unsecure RPC channels.
type PublicAdminAPI struct {
```

```
node *Node // Node interfaced by this API
}

// NewPublicAdminAPI creates a new API definition for the public admin methods
// of the node itself.
func NewPublicAdminAPI(node *Node) *PublicAdminAPI {
return &PublicAdminAPI{node: node}
}

// Peers retrieves all the information we know about each individual peer at the
// protocol granularity.
func (api *PublicAdminAPI) Peers() ([]*p2p.PeerInfo, error) {
server := api.node.Server()
if server == nil {
return nil, ErrNodeStopped
}
return server.PeersInfo(), nil
}

// NodeInfo retrieves all the information we know about the host node at the
// protocol granularity.
func (api *PublicAdminAPI) NodeInfo() (*p2p.NodeInfo, error) {
server := api.node.Server()
if server == nil {
return nil, ErrNodeStopped
}
return server.NodeInfo(), nil
}

// Datadir retrieves the current data directory the node is using.
func (api *PublicAdminAPI) Datadir() string {
return api.node.DataDir()
}

// PublicDebugAPI is the collection of debugging related API methods exposed over
// both secure and unsecure RPC channels.
type PublicDebugAPI struct {
node *Node // Node interfaced by this API
}

// NewPublicDebugAPI creates a new API definition for the public debug methods
// of the node itself.
```

```
func NewPublicDebugAPI(node *Node) *PublicDebugAPI {
return &PublicDebugAPI{node: node}
}

// Metrics retrieves all the known system metric collected by the node.
func (api *PublicDebugAPI) Metrics(raw bool) (map[string]interface{}, error) {
// Create a rate formatter
units := []string{"", "K", "M", "G", "T", "E", "P"}
round := func(value float64, prec int) string {
unit := 0
for value >= 1000 {
unit, value, prec = unit+1, value/1000, 2
}
return fmt.Sprintf(fmt.Sprintf("%%.%df%s", prec, units[unit]), value)
}
format := func(total float64, rate float64) string {
return fmt.Sprintf("%s (%s/s)", round(total, 0), round(rate, 2))
}
// Iterate over all the metrics, and just dump for now
counters := make(map[string]interface{})
metrics.DefaultRegistry.Each(func(name string, metric interface{}) {
// Create or retrieve the counter hierarchy for this metric
root, parts := counters, strings.Split(name, "/")
for _, part := range parts[:len(parts)-1] {
if _, ok := root[part]; !ok {
root[part] = make(map[string]interface{})
}
root = root[part].(map[string]interface{})
}
name = parts[len(parts)-1]

// Fill the counter with the metric details, formatting if requested
if raw {
switch metric := metric.(type) {
case metrics.Meter:
root[name] = map[string]interface{}{
"AvgRate01Min": metric.Rate1(),
"AvgRate05Min": metric.Rate5(),
"AvgRate15Min": metric.Rate15(),
"MeanRate":     metric.RateMean(),
"Overall":      float64(metric.Count()),
}
```

```
case metrics.Timer:
root[name] = map[string]interface{}{
"AvgRate01Min": metric.Rate1(),
"AvgRate05Min": metric.Rate5(),
"AvgRate15Min": metric.Rate15(),
"MeanRate":     metric.RateMean(),
"Overall":      float64(metric.Count()),
"Percentiles": map[string]interface{}{
"5":  metric.Percentile(0.05),
"20": metric.Percentile(0.2),
"50": metric.Percentile(0.5),
"80": metric.Percentile(0.8),
"95": metric.Percentile(0.95),
},
}

default:
root[name] = "Unknown metric type"
}
} else {
switch metric := metric.(type) {
case metrics.Meter:
root[name] = map[string]interface{}{
"Avg01Min": format(metric.Rate1()*60, metric.Rate1()),
"Avg05Min": format(metric.Rate5()*300, metric.Rate5()),
"Avg15Min": format(metric.Rate15()*900, metric.Rate15()),
"Overall":  format(float64(metric.Count()), metric.RateMean()),
}

case metrics.Timer:
root[name] = map[string]interface{}{
"Avg01Min": format(metric.Rate1()*60, metric.Rate1()),
"Avg05Min": format(metric.Rate5()*300, metric.Rate5()),
"Avg15Min": format(metric.Rate15()*900, metric.Rate15()),
"Overall":  format(float64(metric.Count()), metric.RateMean()),
"Maximum":  time.Duration(metric.Max()).String(),
"Minimum":  time.Duration(metric.Min()).String(),
"Percentiles": map[string]interface{}{
"5":  time.Duration(metric.Percentile(0.05)).String(),
"20": time.Duration(metric.Percentile(0.2)).String(),
"50": time.Duration(metric.Percentile(0.5)).String(),
```

```
"80": time.Duration(metric.Percentile(0.8)).String(),
"95": time.Duration(metric.Percentile(0.95)).String(),
},
}

default:
root[name] = "Unknown metric type"
}
}
})
return counters, nil
}

// PublicWeb3API offers helper utils
type PublicWeb3API struct {
stack *Node
}

// NewPublicWeb3API creates a new Web3Service instance
func NewPublicWeb3API(stack *Node) *PublicWeb3API {
return &PublicWeb3API{stack}
}

// ClientVersion returns the node name
func (s *PublicWeb3API) ClientVersion() string {
return s.stack.Server().Name
}

// Sha3 applies the ethereum sha3 implementation on the input.
// It assumes the input is hex encoded.
func (s *PublicWeb3API) Sha3(input hexutil.Bytes) hexutil.Bytes {
return crypto.Keccak256(input)
}

70:F:\git\coin\ethereum\go-ethereum\node\config.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package node

import (
"crypto/ecdsa"
"fmt"
```

```
"io/ioutil"
"os"
"path/filepath"
"runtime"
"strings"

"github.com/ethereum/go-ethereum/accounts"
"github.com/ethereum/go-ethereum/accounts/keystore"
"github.com/ethereum/go-ethereum/accounts/usbwallet"
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/p2p"
"github.com/ethereum/go-ethereum/p2p/discover"
)

var (
datadirPrivateKey      = "nodekey"            // Path within the datadir to the node's private key
datadirDefaultKeyStore = "keystore"           // Path within the datadir to the keystore
datadirStaticNodes     = "static-nodes.json"  // Path within the datadir to the static node list
datadirTrustedNodes    = "trusted-nodes.json" // Path within the datadir to the trusted node list
datadirNodeDatabase    = "nodes"              // Path within the datadir to store the node infos
)

// Config represents a small collection of configuration values to fine tune the
// P2P network layer of a protocol stack. These values can be further extended by
// all registered services.
type Config struct {
// Name sets the instance name of the node. It must not contain the / character and is
// used in the devp2p node identifier. The instance name of geth is "geth". If no
// value is specified, the basename of the current executable is used.
Name string `toml:"-"`

// UserIdent, if set, is used as an additional component in the devp2p node identifier.
UserIdent string `toml:",omitempty"`

// Version should be set to the version number of the program. It is used
// in the devp2p node identifier.
Version string `toml:"-"`

// DataDir is the file system folder the node should use for any data storage
// requirements. The configured data directory will not be directly shared with
```

```
// registered services, instead those can use utility methods to create/access
// databases or flat files. This enables ephemeral nodes which can fully reside
// in memory.
DataDir string

// Configuration of peer-to-peer networking.
P2P p2p.Config

// KeyStoreDir is the file system folder that contains private keys. The directory can
// be specified as a relative path, in which case it is resolved relative to the
// current directory.
//
// If KeyStoreDir is empty, the default location is the "keystore" subdirectory of
// DataDir. If DataDir is unspecified and KeyStoreDir is empty, an ephemeral directory
// is created by New and destroyed when the node is stopped.
KeyStoreDir string `toml:",omitempty"`

// UseLightweightKDF lowers the memory and CPU requirements of the key store
// scrypt KDF at the expense of security.
UseLightweightKDF bool `toml:",omitempty"`

// NoUSB disables hardware wallet monitoring and connectivity.
NoUSB bool `toml:",omitempty"`

// IPCPath is the requested location to place the IPC endpoint. If the path is
// a simple file name, it is placed inside the data directory (or on the root
// pipe path on Windows), whereas if it's a resolvable path name (absolute or
// relative), then that specific path is enforced. An empty path disables IPC.
IPCPath string `toml:",omitempty"`

// HTTPHost is the host interface on which to start the HTTP RPC server. If this
// field is empty, no HTTP API endpoint will be started.
HTTPHost string `toml:",omitempty"`

// HTTPPort is the TCP port number on which to start the HTTP RPC server. The
// default zero value is/ valid and will pick a port number randomly (useful
// for ephemeral nodes).
HTTPPort int `toml:",omitempty"`

// HTTPCors is the Cross-Origin Resource Sharing header to send to requesting
// clients. Please be aware that CORS is a browser enforced security, it's fully
// useless for custom HTTP clients.
```

```
HTTPCors []string `toml:",omitempty"`

// HTTPModules is a list of API modules to expose via the HTTP RPC interface.
// If the module list is empty, all RPC API endpoints designated public will be
// exposed.
HTTPModules []string `toml:",omitempty"`

// WSHost is the host interface on which to start the websocket RPC server. If
// this field is empty, no websocket API endpoint will be started.
WSHost string `toml:",omitempty"`

// WSPort is the TCP port number on which to start the websocket RPC server. The
// default zero value is/ valid and will pick a port number randomly (useful for
// ephemeral nodes).
WSPort int `toml:",omitempty"`

// WSOrigins is the list of domain to accept websocket requests from. Please be
// aware that the server can only act upon the HTTP request the client sends and
// cannot verify the validity of the request header.
WSOrigins []string `toml:",omitempty"`

// WSModules is a list of API modules to expose via the websocket RPC interface.
// If the module list is empty, all RPC API endpoints designated public will be
// exposed.
WSModules []string `toml:",omitempty"`
}

// IPCEndpoint resolves an IPC endpoint based on a configured value, taking into
// account the set data folders as well as the designated platform we're currently
// running on.
func (c *Config) IPCEndpoint() string {
// Short circuit if IPC has not been enabled
if c.IPCPath == "" {
return ""
}
// On windows we can only use plain top-level pipes
if runtime.GOOS == "windows" {
if strings.HasPrefix(c.IPCPath, `\\.\pipe\`) {
return c.IPCPath
}
return `\\.\pipe\` + c.IPCPath
}
```

```
// Resolve names into the data directory full paths otherwise
if filepath.Base(c.IPCPath) == c.IPCPath {
if c.DataDir == "" {
return filepath.Join(os.TempDir(), c.IPCPath)
}
return filepath.Join(c.DataDir, c.IPCPath)
}
return c.IPCPath
}

// NodeDB returns the path to the discovery node database.
func (c *Config) NodeDB() string {
if c.DataDir == "" {
return "" // ephemeral
}
return c.resolvePath("nodes")
}

// DefaultIPCEndpoint returns the IPC path used by default.
func DefaultIPCEndpoint(clientIdentifier string) string {
if clientIdentifier == "" {
clientIdentifier = strings.TrimSuffix(filepath.Base(os.Args[0]), ".exe")
if clientIdentifier == "" {
panic("empty executable name")
}
}
config := &Config{DataDir: DefaultDataDir(), IPCPath: clientIdentifier + ".ipc"}
return config.IPCEndpoint()
}

// HTTPEndpoint resolves an HTTP endpoint based on the configured host interface
// and port parameters.
func (c *Config) HTTPEndpoint() string {
if c.HTTPHost == "" {
return ""
}
return fmt.Sprintf("%s:%d", c.HTTPHost, c.HTTPPort)
}

// DefaultHTTPEndpoint returns the HTTP endpoint used by default.
func DefaultHTTPEndpoint() string {
config := &Config{HTTPHost: DefaultHTTPHost, HTTPPort: DefaultHTTPPort}
```

```
return config.HTTPEndpoint()
}

// WSEndpoint resolves an websocket endpoint based on the configured host interface
// and port parameters.
func (c *Config) WSEndpoint() string {
if c.WSHost == "" {
return ""
}
return fmt.Sprintf("%s:%d", c.WSHost, c.WSPort)
}

// DefaultWSEndpoint returns the websocket endpoint used by default.
func DefaultWSEndpoint() string {
config := &Config{WSHost: DefaultWSHost, WSPort: DefaultWSPort}
return config.WSEndpoint()
}

// NodeName returns the devp2p node identifier.
func (c *Config) NodeName() string {
name := c.name()
// Backwards compatibility: previous versions used title-cased "Geth", keep that.
if name == "geth" || name == "geth-testnet" {
name = "Geth"
}
if c.UserIdent != "" {
name += "/" + c.UserIdent
}
if c.Version != "" {
name += "/v" + c.Version
}
name += "/" + runtime.GOOS + "-" + runtime.GOARCH
name += "/" + runtime.Version()
return name
}

func (c *Config) name() string {
if c.Name == "" {
progname := strings.TrimSuffix(filepath.Base(os.Args[0]), ".exe")
if progname == "" {
panic("empty executable name, set Config.Name")
}
```

```
return progname
}
return c.Name
}

// These resources are resolved differently for "geth" instances.
var isOldGethResource = map[string]bool{
"chaindata":          true,
"nodes":              true,
"nodekey":            true,
"static-nodes.json":  true,
"trusted-nodes.json": true,
}

// resolvePath resolves path in the instance directory.
func (c *Config) resolvePath(path string) string {
if filepath.IsAbs(path) {
return path
}
if c.DataDir == "" {
return ""
}
// Backwards-compatibility: ensure that data directory files created
// by geth 1.4 are used if they exist.
if c.name() == "geth" && isOldGethResource[path] {
oldpath := ""
if c.Name == "geth" {
oldpath = filepath.Join(c.DataDir, path)
}
if oldpath != "" && common.FileExist(oldpath) {
// TODO: print warning
return oldpath
}
}
return filepath.Join(c.instanceDir(), path)
}

func (c *Config) instanceDir() string {
if c.DataDir == "" {
return ""
}
return filepath.Join(c.DataDir, c.name())
```

```
}

// NodeKey retrieves the currently configured private key of the node, checking
// first any manually set key, falling back to the one found in the configured
// data folder. If no key can be found, a new one is generated.
func (c *Config) NodeKey() *ecdsa.PrivateKey {
// Use any specifically configured key.
if c.P2P.PrivateKey != nil {
return c.P2P.PrivateKey
}
// Generate ephemeral key if no datadir is being used.
if c.DataDir == "" {
key, err := crypto.GenerateKey()
if err != nil {
log.Crit(fmt.Sprintf("Failed to generate ephemeral node key: %v", err))
}
return key
}

keyfile := c.resolvePath(datadirPrivateKey)
if key, err := crypto.LoadECDSA(keyfile); err == nil {
return key
}
// No persistent key found, generate and store a new one.
key, err := crypto.GenerateKey()
if err != nil {
log.Crit(fmt.Sprintf("Failed to generate node key: %v", err))
}
instanceDir := filepath.Join(c.DataDir, c.name())
if err := os.MkdirAll(instanceDir, 0700); err != nil {
log.Error(fmt.Sprintf("Failed to persist node key: %v", err))
return key
}
keyfile = filepath.Join(instanceDir, datadirPrivateKey)
if err := crypto.SaveECDSA(keyfile, key); err != nil {
log.Error(fmt.Sprintf("Failed to persist node key: %v", err))
}
return key
}

// StaticNodes returns a list of node enode URLs configured as static nodes.
func (c *Config) StaticNodes() []*discover.Node {
```

```
return c.parsePersistentNodes(c.resolvePath(datadirStaticNodes))
}

// TrusterNodes returns a list of node enode URLs configured as trusted nodes.
func (c *Config) TrusterNodes() []*discover.Node {
return c.parsePersistentNodes(c.resolvePath(datadirTrustedNodes))
}

// parsePersistentNodes parses a list of discovery node URLs loaded from a .json
// file from within the data directory.
func (c *Config) parsePersistentNodes(path string) []*discover.Node {
// Short circuit if no node config is present
if c.DataDir == "" {
return nil
}
if _, err := os.Stat(path); err != nil {
return nil
}
// Load the nodes from the config file.
var nodelist []string
if err := common.LoadJSON(path, &nodelist); err != nil {
log.Error(fmt.Sprintf("Can't load node file %s: %v", path, err))
return nil
}
// Interpret the list as a discovery node array
var nodes []*discover.Node
for _, url := range nodelist {
if url == "" {
continue
}
node, err := discover.ParseNode(url)
if err != nil {
log.Error(fmt.Sprintf("Node URL %s: %v\n", url, err))
continue
}
nodes = append(nodes, node)
}
return nodes
}

func makeAccountManager(conf *Config) (*accounts.Manager, string, error) {
scryptN := keystore.StandardScryptN
```

```
scryptP := keystore.StandardScryptP
if conf.UseLightweightKDF {
scryptN = keystore.LightScryptN
scryptP = keystore.LightScryptP
}

var (
keydir    string
ephemeral string
err       error
)
switch {
case filepath.IsAbs(conf.KeyStoreDir):
keydir = conf.KeyStoreDir
case conf.DataDir != "":
if conf.KeyStoreDir == "" {
keydir = filepath.Join(conf.DataDir, datadirDefaultKeyStore)
} else {
keydir, err = filepath.Abs(conf.KeyStoreDir)
}
case conf.KeyStoreDir != "":
keydir, err = filepath.Abs(conf.KeyStoreDir)
default:
// There is no datadir.
keydir, err = ioutil.TempDir("", "go-ethereum-keystore")
ephemeral = keydir
}
if err != nil {
return nil, "", err
}
if err := os.MkdirAll(keydir, 0700); err != nil {
return nil, "", err
}
// Assemble the account manager and supported backends
backends := []accounts.Backend{
keystore.NewKeyStore(keydir, scryptN, scryptP),
}
if !conf.NoUSB {
if ledgerhub, err := usbwallet.NewLedgerHub(); err != nil {
log.Warn(fmt.Sprintf("Failed to start Ledger hub, disabling: %v", err))
} else {
backends = append(backends, ledgerhub)
```

```
}
}
return accounts.NewManager(backends...), ephemeral, nil
}

71:F:\git\coin\ethereum\go-ethereum\node\config_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package node

import (
"bytes"
"io/ioutil"
"os"
"path/filepath"
"runtime"
"testing"

"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/p2p"
)

// Tests that datadirs can be successfully created, be them manually configured
// ones or automatically generated temporary ones.
func TestDatadirCreation(t *testing.T) {
// Create a temporary data dir and check that it can be used by a node
dir, err := ioutil.TempDir("", "")
if err != nil {
t.Fatalf("failed to create manual data dir: %v", err)
}
defer os.RemoveAll(dir)

if _, err := New(&Config{DataDir: dir}); err != nil {
t.Fatalf("failed to create stack with existing datadir: %v", err)
}
// Generate a long non-existing datadir path and check that it gets created by a node
dir = filepath.Join(dir, "a", "b", "c", "d", "e", "f")
if _, err := New(&Config{DataDir: dir}); err != nil {
t.Fatalf("failed to create stack with creatable datadir: %v", err)
}
if _, err := os.Stat(dir); err != nil {
t.Fatalf("freshly created datadir not accessible: %v", err)
```

```
}
// Verify that an impossible datadir fails creation
file, err := ioutil.TempFile("", "")
if err != nil {
t.Fatalf("failed to create temporary file: %v", err)
}
defer os.Remove(file.Name())

dir = filepath.Join(file.Name(), "invalid/path")
if _, err := New(&Config{DataDir: dir}); err == nil {
t.Fatalf("protocol stack created with an invalid datadir")
}
}

// Tests that IPC paths are correctly resolved to valid endpoints of different
// platforms.
func TestIPCPathResolution(t *testing.T) {
var tests = []struct {
DataDir  string
IPCPath  string
Windows  bool
Endpoint string
}{
{"", "", false, ""},
{"data", "", false, ""},
{"", "geth.ipc", false, filepath.Join(os.TempDir(), "geth.ipc")},
{"data", "geth.ipc", false, "data/geth.ipc"},
{"data", "./geth.ipc", false, "./geth.ipc"},
{"data", "/geth.ipc", false, "/geth.ipc"},
{"", "", true, ``},
{"data", "", true, ``},
{"", "geth.ipc", true, `\\.\pipe\geth.ipc`},
{"data", "geth.ipc", true, `\\.\pipe\geth.ipc`},
{"data", `\\.\pipe\geth.ipc`, true, `\\.\pipe\geth.ipc`},
}
for i, test := range tests {
// Only run when platform/test match
if (runtime.GOOS == "windows") == test.Windows {
if endpoint := (&Config{DataDir: test.DataDir, IPCPath: test.IPCPath}).IPCEndpoint(); endpoint !=
test.Endpoint {
t.Errorf("test %d: IPC endpoint mismatch: have %s, want %s", i, endpoint, test.Endpoint)
}
```

```
}
}
}

// Tests that node keys can be correctly created, persisted, loaded and/or made
// ephemeral.
func TestNodeKeyPersistency(t *testing.T) {
// Create a temporary folder and make sure no key is present
dir, err := ioutil.TempDir("", "node-test")
if err != nil {
t.Fatalf("failed to create temporary data directory: %v", err)
}
defer os.RemoveAll(dir)

keyfile := filepath.Join(dir, "unit-test", datadirPrivateKey)

// Configure a node with a preset key and ensure it's not persisted
key, err := crypto.GenerateKey()
if err != nil {
t.Fatalf("failed to generate one-shot node key: %v", err)
}
config := &Config{Name: "unit-test", DataDir: dir, P2P: p2p.Config{PrivateKey: key}}
config.NodeKey()
if _, err := os.Stat(filepath.Join(keyfile)); err == nil {
t.Fatalf("one-shot node key persisted to data directory")
}

// Configure a node with no preset key and ensure it is persisted this time
config = &Config{Name: "unit-test", DataDir: dir}
config.NodeKey()
if _, err := os.Stat(keyfile); err != nil {
t.Fatalf("node key not persisted to data directory: %v", err)
}
if _, err = crypto.LoadECDSA(keyfile); err != nil {
t.Fatalf("failed to load freshly persisted node key: %v", err)
}
blob1, err := ioutil.ReadFile(keyfile)
if err != nil {
t.Fatalf("failed to read freshly persisted node key: %v", err)
}

// Configure a new node and ensure the previously persisted key is loaded
```

```
config = &Config{Name: "unit-test", DataDir: dir}
config.NodeKey()
blob2, err := ioutil.ReadFile(filepath.Join(keyfile))
if err != nil {
t.Fatalf("failed to read previously persisted node key: %v", err)
}
if !bytes.Equal(blob1, blob2) {
t.Fatalf("persisted node key mismatch: have %x, want %x", blob2, blob1)
}

// Configure ephemeral node and ensure no key is dumped locally
config = &Config{Name: "unit-test", DataDir: ""}
config.NodeKey()
if _, err := os.Stat(filepath.Join(".", "unit-test", datadirPrivateKey)); err == nil {
t.Fatalf("ephemeral node key persisted to disk")
}
}
```

72:F:\git\coin\ethereum\go-ethereum\node\defaults.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package node

import (
"os"
"os/user"
"path/filepath"
"runtime"

"github.com/ethereum/go-ethereum/p2p"
"github.com/ethereum/go-ethereum/p2p/nat"
)

const (
DefaultHTTPHost = "localhost" // Default host interface for the HTTP RPC server
DefaultHTTPPort = 8545        // Default TCP port for the HTTP RPC server
DefaultWSHost   = "localhost" // Default host interface for the websocket RPC server
DefaultWSPort   = 8546        // Default TCP port for the websocket RPC server
)

// DefaultConfig contains reasonable default settings.
var DefaultConfig = Config{
```

```
DataDir:     DefaultDataDir(),
HTTPPort:    DefaultHTTPPort,
HTTPModules: []string{"net", "web3"},
WSPort:      DefaultWSPort,
WSModules:   []string{"net", "web3"},
P2P: p2p.Config{
ListenAddr:      ":30303",
DiscoveryV5Addr: ":30304",
MaxPeers:        25,
NAT:             nat.Any(),
},
}

// DefaultDataDir is the default data directory to use for the databases and other
// persistence requirements.
func DefaultDataDir() string {
// Try to place the data folder in the user's home dir
home := homeDir()
if home != "" {
if runtime.GOOS == "darwin" {
return filepath.Join(home, "Library", "Ethereum")
} else if runtime.GOOS == "windows" {
return filepath.Join(home, "AppData", "Roaming", "Ethereum")
} else {
return filepath.Join(home, ".ethereum")
}
}
// As we cannot guess a stable location, return empty and handle later
return ""
}

func homeDir() string {
if home := os.Getenv("HOME"); home != "" {
return home
}
if usr, err := user.Current(); err == nil {
return usr.HomeDir
}
return ""
}
```

73:F:\git\coin\ethereum\go-ethereum\node\doc.go

// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

/*
Package node sets up multi-protocol Ethereum nodes.

In the model exposed by this package, a node is a collection of services which use shared resources to provide RPC APIs. Services can also offer devp2p protocols, which are wired up to the devp2p network when the node instance is started.

Resources Managed By Node

All file-system resources used by a node instance are located in a directory called the data directory. The location of each resource can be overridden through additional node configuration. The data directory is optional. If it is not set and the location of a resource is otherwise unspecified, package node will create the resource in memory.

To access to the devp2p network, Node configures and starts p2p.Server. Each host on the devp2p network has a unique identifier, the node key. The Node instance persists this key across restarts. Node also loads static and trusted node lists and ensures that knowledge about other hosts is persisted.

JSON-RPC servers which run HTTP, WebSocket or IPC can be started on a Node. RPC modules offered by registered services will be offered on those endpoints. Users can restrict any endpoint to a subset of RPC modules. Node itself offers the "debug", "admin" and "web3" modules.

Service implementations can open LevelDB databases through the service context. Package node chooses the file system location of each database. If the node is configured to run without a data directory, databases are opened in memory instead.

Node also creates the shared store of encrypted Ethereum account keys. Services can access the account manager through the service context.

Sharing Data Directory Among Instances

Multiple node instances can share a single data directory if they have distinct instance names (set through the Name config option). Sharing behaviour depends on the type of resource.

devp2p-related resources (node key, static/trusted node lists, known hosts database) are

stored in a directory with the same name as the instance. Thus, multiple node instances using the same data directory will store this information in different subdirectories of the data directory.

LevelDB databases are also stored within the instance subdirectory. If multiple node instances use the same data directory, openening the databases with identical names will create one database for each instance.

The account key store is shared among all node instances using the same data directory unless its location is changed through the KeyStoreDir configuration option.

Data Directory Sharing Example

In this exanple, two node instances named A and B are started with the same data directory. Mode instance A opens the database "db", node instance B opens the databases "db" and "db-2". The following files will be created in the data directory:

```
  data-directory/
      A/
         nodekey           -- devp2p node key of instance A
         nodes/            -- devp2p discovery knowledge database of instance A
         db/               -- LevelDB content for "db"
      A.ipc                -- JSON-RPC UNIX domain socket endpoint of instance A
      B/
         nodekey           -- devp2p node key of node B
         nodes/            -- devp2p discovery knowledge database of instance B
         static-nodes.json  -- devp2p static node list of instance B
         db/               -- LevelDB content for "db"
         db-2/             -- LevelDB content for "db-2"
      B.ipc                -- JSON-RPC UNIX domain socket endpoint of instance A
      keystore/            -- account key store, used by both instances
*/
package node
```

74:F:\git\coin\ethereum\go-ethereum\node\errors.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package node

import (
"fmt"
```

```
"reflect"
)

// DuplicateServiceError is returned during Node startup if a registered service
// constructor returns a service of the same type that was already started.
type DuplicateServiceError struct {
Kind reflect.Type
}

// Error generates a textual representation of the duplicate service error.
func (e *DuplicateServiceError) Error() string {
return fmt.Sprintf("duplicate service: %v", e.Kind)
}

// StopError is returned if a Node fails to stop either any of its registered
// services or itself.
type StopError struct {
Server   error
Services map[reflect.Type]error
}

// Error generates a textual representation of the stop error.
func (e *StopError) Error() string {
return fmt.Sprintf("server: %v, services: %v", e.Server, e.Services)
}

75:F:\git\coin\ethereum\go-ethereum\node\node.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package node

import (
"errors"
"fmt"
"net"
"os"
"path/filepath"
"reflect"
"strings"
"sync"
"syscall"
```

```
"github.com/ethereum/go-ethereum/accounts"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/internal/debug"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/p2p"
"github.com/ethereum/go-ethereum/rpc"
"github.com/syndtr/goleveldb/leveldb/storage"
)

var (
ErrDatadirUsed    = errors.New("datadir already used")
ErrNodeStopped    = errors.New("node not started")
ErrNodeRunning    = errors.New("node already running")
ErrServiceUnknown = errors.New("unknown service")

datadirInUseErrnos = map[uint]bool{11: true, 32: true, 35: true}
)

// Node is a container on which services can be registered.
type Node struct {
eventmux *event.TypeMux // Event multiplexer used between the services of a stack
config   *Config
accman   *accounts.Manager

ephemeralKeystore string          // if non-empty, the key directory that will be removed by Stop
instanceDirLock   storage.Storage // prevents concurrent use of instance directory

serverConfig p2p.Config
server       *p2p.Server // Currently running P2P networking layer

serviceFuncs []ServiceConstructor     // Service constructors (in dependency order)
services     map[reflect.Type]Service // Currently running services

rpcAPIs       []rpc.API   // List of APIs currently provided by the node
inprocHandler *rpc.Server // In-process RPC request handler to process the API requests

ipcEndpoint string       // IPC endpoint to listen at (empty = IPC disabled)
ipcListener net.Listener // IPC RPC listener socket to serve API requests
ipcHandler  *rpc.Server  // IPC RPC request handler to process the API requests

httpEndpoint  string       // HTTP endpoint (interface + port) to listen at (empty = HTTP disabled)
```

```
httpWhitelist []string     // HTTP RPC modules to allow through this endpoint
httpListener  net.Listener // HTTP RPC listener socket to server API requests
httpHandler   *rpc.Server  // HTTP RPC request handler to process the API requests

wsEndpoint string       // Websocket endpoint (interface + port) to listen at (empty = websocket disabled)
wsListener net.Listener // Websocket RPC listener socket to server API requests
wsHandler  *rpc.Server  // Websocket RPC request handler to process the API requests

stop chan struct{} // Channel to wait for termination notifications
lock sync.RWMutex
}

// New creates a new P2P node, ready for protocol registration.
func New(conf *Config) (*Node, error) {
// Copy config and resolve the datadir so future changes to the current
// working directory don't affect the node.
confCopy := *conf
conf = &confCopy
if conf.DataDir != "" {
absdatadir, err := filepath.Abs(conf.DataDir)
if err != nil {
return nil, err
}
conf.DataDir = absdatadir
}
// Ensure that the instance name doesn't cause weird conflicts with
// other files in the data directory.
if strings.ContainsAny(conf.Name, `/\`) {
return nil, errors.New(`Config.Name must not contain '/' or '\'`)
}
if conf.Name == datadirDefaultKeyStore {
return nil, errors.New(`Config.Name cannot be "` + datadirDefaultKeyStore + `"`)
}
if strings.HasSuffix(conf.Name, ".ipc") {
return nil, errors.New(`Config.Name cannot end in ".ipc"`)
}
// Ensure that the AccountManager method works before the node has started.
// We rely on this in cmd/geth.
am, ephemeralKeystore, err := makeAccountManager(conf)
if err != nil {
return nil, err
```

```
}
// Note: any interaction with Config that would create/touch files
// in the data directory or instance directory is delayed until Start.
return &Node{
accman:            am,
ephemeralKeystore: ephemeralKeystore,
config:            conf,
serviceFuncs:      []ServiceConstructor{},
ipcEndpoint:       conf.IPCEndpoint(),
httpEndpoint:      conf.HTTPEndpoint(),
wsEndpoint:        conf.WSEndpoint(),
eventmux:          new(event.TypeMux),
}, nil
}

// Register injects a new service into the node's stack. The service created by
// the passed constructor must be unique in its type with regard to sibling ones.
func (n *Node) Register(constructor ServiceConstructor) error {
n.lock.Lock()
defer n.lock.Unlock()

if n.server != nil {
return ErrNodeRunning
}
n.serviceFuncs = append(n.serviceFuncs, constructor)
return nil
}

// Start create a live P2P node and starts running it.
func (n *Node) Start() error {
n.lock.Lock()
defer n.lock.Unlock()

// Short circuit if the node's already running
if n.server != nil {
return ErrNodeRunning
}
if err := n.openDataDir(); err != nil {
return err
}

// Initialize the p2p server. This creates the node key and
```

```
// discovery databases.
n.serverConfig = n.config.P2P
n.serverConfig.PrivateKey = n.config.NodeKey()
n.serverConfig.Name = n.config.NodeName()
if n.serverConfig.StaticNodes == nil {
n.serverConfig.StaticNodes = n.config.StaticNodes()
}
if n.serverConfig.TrustedNodes == nil {
n.serverConfig.TrustedNodes = n.config.TrusterNodes()
}
if n.serverConfig.NodeDatabase == "" {
n.serverConfig.NodeDatabase = n.config.NodeDB()
}
running := &p2p.Server{Config: n.serverConfig}
log.Info("Starting peer-to-peer node", "instance", n.serverConfig.Name)

// Otherwise copy and specialize the P2P configuration
services := make(map[reflect.Type]Service)
for _, constructor := range n.serviceFuncs {
// Create a new context for the particular service
ctx := &ServiceContext{
config:         n.config,
services:       make(map[reflect.Type]Service),
EventMux:       n.eventmux,
AccountManager: n.accman,
}
for kind, s := range services { // copy needed for threaded access
ctx.services[kind] = s
}
// Construct and save the service
service, err := constructor(ctx)
if err != nil {
return err
}
kind := reflect.TypeOf(service)
if _, exists := services[kind]; exists {
return &DuplicateServiceError{Kind: kind}
}
services[kind] = service
}
// Gather the protocols and start the freshly assembled P2P server
for _, service := range services {
```

```
running.Protocols = append(running.Protocols, service.Protocols()...)
}
if err := running.Start(); err != nil {
if errno, ok := err.(syscall.Errno); ok && datadirInUseErrnos[uint(errno)] {
return ErrDatadirUsed
}
return err
}
// Start each of the services
started := []reflect.Type{}
for kind, service := range services {
// Start the next service, stopping all previous upon failure
if err := service.Start(running); err != nil {
for _, kind := range started {
services[kind].Stop()
}
running.Stop()

return err
}
// Mark the service started for potential cleanup
started = append(started, kind)
}
// Lastly start the configured RPC interfaces
if err := n.startRPC(services); err != nil {
for _, service := range services {
service.Stop()
}
running.Stop()
return err
}
// Finish initializing the startup
n.services = services
n.server = running
n.stop = make(chan struct{})

return nil
}

func (n *Node) openDataDir() error {
if n.config.DataDir == "" {
return nil // ephemeral
```

```
}

instdir := filepath.Join(n.config.DataDir, n.config.name())
if err := os.MkdirAll(instdir, 0700); err != nil {
return err
}
// Try to open the instance directory as LevelDB storage. This creates a lock file
// which prevents concurrent use by another instance as well as accidental use of the
// instance directory as a database.
storage, err := storage.OpenFile(instdir, true)
if err != nil {
return err
}
n.instanceDirLock = storage
return nil
}

// startRPC is a helper method to start all the various RPC endpoint during node
// startup. It's not meant to be called at any time afterwards as it makes certain
// assumptions about the state of the node.
func (n *Node) startRPC(services map[reflect.Type]Service) error {
// Gather all the possible APIs to surface
apis := n.apis()
for _, service := range services {
apis = append(apis, service.APIs()...)
}
// Start the various API endpoints, terminating all in case of errors
if err := n.startInProc(apis); err != nil {
return err
}
if err := n.startIPC(apis); err != nil {
n.stopInProc()
return err
}
if err := n.startHTTP(n.httpEndpoint, apis, n.config.HTTPModules, n.config.HTTPCors); err != nil {
n.stopIPC()
n.stopInProc()
return err
}
if err := n.startWS(n.wsEndpoint, apis, n.config.WSModules, n.config.WSOrigins); err != nil {
n.stopHTTP()
n.stopIPC()
```

```
n.stopInProc()
return err
}
// All API endpoints started successfully
n.rpcAPIs = apis
return nil
}

// startInProc initializes an in-process RPC endpoint.
func (n *Node) startInProc(apis []rpc.API) error {
// Register all the APIs exposed by the services
handler := rpc.NewServer()
for _, api := range apis {
if err := handler.RegisterName(api.Namespace, api.Service); err != nil {
return err
}
log.Debug(fmt.Sprintf("InProc registered %T under '%s'", api.Service, api.Namespace))
}
n.inprocHandler = handler
return nil
}

// stopInProc terminates the in-process RPC endpoint.
func (n *Node) stopInProc() {
if n.inprocHandler != nil {
n.inprocHandler.Stop()
n.inprocHandler = nil
}
}

// startIPC initializes and starts the IPC RPC endpoint.
func (n *Node) startIPC(apis []rpc.API) error {
// Short circuit if the IPC endpoint isn't being exposed
if n.ipcEndpoint == "" {
return nil
}
// Register all the APIs exposed by the services
handler := rpc.NewServer()
for _, api := range apis {
if err := handler.RegisterName(api.Namespace, api.Service); err != nil {
return err
}
```

```
log.Debug(fmt.Sprintf("IPC registered %T under '%s'", api.Service, api.Namespace))
}
// All APIs registered, start the IPC listener
var (
listener net.Listener
err      error
)
if listener, err = rpc.CreateIPCListener(n.ipcEndpoint); err != nil {
return err
}
go func() {
log.Info(fmt.Sprintf("IPC endpoint opened: %s", n.ipcEndpoint))

for {
conn, err := listener.Accept()
if err != nil {
// Terminate if the listener was closed
n.lock.RLock()
closed := n.ipcListener == nil
n.lock.RUnlock()
if closed {
return
}
// Not closed, just some error; report and continue
log.Error(fmt.Sprintf("IPC accept failed: %v", err))
continue
}
go handler.ServeCodec(rpc.NewJSONCodec(conn),
rpc.OptionMethodInvocation|rpc.OptionSubscriptions)
}
}()
// All listeners booted successfully
n.ipcListener = listener
n.ipcHandler = handler

return nil
}

// stopIPC terminates the IPC RPC endpoint.
func (n *Node) stopIPC() {
if n.ipcListener != nil {
n.ipcListener.Close()
```

```
n.ipcListener = nil

log.Info(fmt.Sprintf("IPC endpoint closed: %s", n.ipcEndpoint))
}
if n.ipcHandler != nil {
n.ipcHandler.Stop()
n.ipcHandler = nil
}
}

// startHTTP initializes and starts the HTTP RPC endpoint.
func (n *Node) startHTTP(endpoint string, apis []rpc.API, modules []string, cors []string) error {
// Short circuit if the HTTP endpoint isn't being exposed
if endpoint == "" {
return nil
}
// Generate the whitelist based on the allowed modules
whitelist := make(map[string]bool)
for _, module := range modules {
whitelist[module] = true
}
// Register all the APIs exposed by the services
handler := rpc.NewServer()
for _, api := range apis {
if whitelist[api.Namespace] || (len(whitelist) == 0 && api.Public) {
if err := handler.RegisterName(api.Namespace, api.Service); err != nil {
return err
}
log.Debug(fmt.Sprintf("HTTP registered %T under '%s'", api.Service, api.Namespace))
}
}
// All APIs registered, start the HTTP listener
var (
listener net.Listener
err      error
)
if listener, err = net.Listen("tcp", endpoint); err != nil {
return err
}
go rpc.NewHTTPServer(cors, handler).Serve(listener)
log.Info(fmt.Sprintf("HTTP endpoint opened: http://%s", endpoint))
```

```
// All listeners booted successfully
n.httpEndpoint = endpoint
n.httpListener = listener
n.httpHandler = handler

return nil
}

// stopHTTP terminates the HTTP RPC endpoint.
func (n *Node) stopHTTP() {
if n.httpListener != nil {
n.httpListener.Close()
n.httpListener = nil

log.Info(fmt.Sprintf("HTTP endpoint closed: http://%s", n.httpEndpoint))
}
if n.httpHandler != nil {
n.httpHandler.Stop()
n.httpHandler = nil
}
}

// startWS initializes and starts the websocket RPC endpoint.
func (n *Node) startWS(endpoint string, apis []rpc.API, modules []string, wsOrigins []string) error {
// Short circuit if the WS endpoint isn't being exposed
if endpoint == "" {
return nil
}
// Generate the whitelist based on the allowed modules
whitelist := make(map[string]bool)
for _, module := range modules {
whitelist[module] = true
}
// Register all the APIs exposed by the services
handler := rpc.NewServer()
for _, api := range apis {
if whitelist[api.Namespace] || (len(whitelist) == 0 && api.Public) {
if err := handler.RegisterName(api.Namespace, api.Service); err != nil {
return err
}
log.Debug(fmt.Sprintf("WebSocket registered %T under '%s'", api.Service, api.Namespace))
}
```

```
}
// All APIs registered, start the HTTP listener
var (
listener net.Listener
err      error
)
if listener, err = net.Listen("tcp", endpoint); err != nil {
return err
}
go rpc.NewWSServer(wsOrigins, handler).Serve(listener)
log.Info(fmt.Sprintf("WebSocket endpoint opened: ws://%s", endpoint))

// All listeners booted successfully
n.wsEndpoint = endpoint
n.wsListener = listener
n.wsHandler = handler

return nil
}

// stopWS terminates the websocket RPC endpoint.
func (n *Node) stopWS() {
if n.wsListener != nil {
n.wsListener.Close()
n.wsListener = nil

log.Info(fmt.Sprintf("WebSocket endpoint closed: ws://%s", n.wsEndpoint))
}
if n.wsHandler != nil {
n.wsHandler.Stop()
n.wsHandler = nil
}
}

// Stop terminates a running node along with all it's services. In the node was
// not started, an error is returned.
func (n *Node) Stop() error {
n.lock.Lock()
defer n.lock.Unlock()

// Short circuit if the node's not running
if n.server == nil {
```

```
return ErrNodeStopped
}

// Terminate the API, services and the p2p server.
n.stopWS()
n.stopHTTP()
n.stopIPC()
n.rpcAPIs = nil
failure := &StopError{
Services: make(map[reflect.Type]error),
}
for kind, service := range n.services {
if err := service.Stop(); err != nil {
failure.Services[kind] = err
}
}
n.server.Stop()
n.services = nil
n.server = nil

// Release instance directory lock.
if n.instanceDirLock != nil {
n.instanceDirLock.Close()
n.instanceDirLock = nil
}

// unblock n.Wait
close(n.stop)

// Remove the keystore if it was created ephemerally.
var keystoreErr error
if n.ephemeralKeystore != "" {
keystoreErr = os.RemoveAll(n.ephemeralKeystore)
}

if len(failure.Services) > 0 {
return failure
}
if keystoreErr != nil {
return keystoreErr
}
return nil
```

```
}

// Wait blocks the thread until the node is stopped. If the node is not running
// at the time of invocation, the method immediately returns.
func (n *Node) Wait() {
n.lock.RLock()
if n.server == nil {
n.lock.RUnlock()
return
}
stop := n.stop
n.lock.RUnlock()

<-stop
}

// Restart terminates a running node and boots up a new one in its place. If the
// node isn't running, an error is returned.
func (n *Node) Restart() error {
if err := n.Stop(); err != nil {
return err
}
if err := n.Start(); err != nil {
return err
}
return nil
}

// Attach creates an RPC client attached to an in-process API handler.
func (n *Node) Attach() (*rpc.Client, error) {
n.lock.RLock()
defer n.lock.RUnlock()

if n.server == nil {
return nil, ErrNodeStopped
}
return rpc.DialInProc(n.inprocHandler), nil
}

// Server retrieves the currently running P2P network layer. This method is meant
// only to inspect fields of the currently running server, life cycle management
// should be left to this Node entity.
```

```
func (n *Node) Server() *p2p.Server {
n.lock.RLock()
defer n.lock.RUnlock()

return n.server
}

// Service retrieves a currently running service registered of a specific type.
func (n *Node) Service(service interface{}) error {
n.lock.RLock()
defer n.lock.RUnlock()

// Short circuit if the node's not running
if n.server == nil {
return ErrNodeStopped
}
// Otherwise try to find the service to return
element := reflect.ValueOf(service).Elem()
if running, ok := n.services[element.Type()]; ok {
element.Set(reflect.ValueOf(running))
return nil
}
return ErrServiceUnknown
}

// DataDir retrieves the current datadir used by the protocol stack.
// Deprecated: No files should be stored in this directory, use InstanceDir instead.
func (n *Node) DataDir() string {
return n.config.DataDir
}

// InstanceDir retrieves the instance directory used by the protocol stack.
func (n *Node) InstanceDir() string {
return n.config.instanceDir()
}

// AccountManager retrieves the account manager used by the protocol stack.
func (n *Node) AccountManager() *accounts.Manager {
return n.accman
}

// IPCEndpoint retrieves the current IPC endpoint used by the protocol stack.
```

```
func (n *Node) IPCEndpoint() string {
return n.ipcEndpoint
}

// HTTPEndpoint retrieves the current HTTP endpoint used by the protocol stack.
func (n *Node) HTTPEndpoint() string {
return n.httpEndpoint
}

// WSEndpoint retrieves the current WS endpoint used by the protocol stack.
func (n *Node) WSEndpoint() string {
return n.wsEndpoint
}

// EventMux retrieves the event multiplexer used by all the network services in
// the current protocol stack.
func (n *Node) EventMux() *event.TypeMux {
return n.eventmux
}

// OpenDatabase opens an existing database with the given name (or creates one if no
// previous can be found) from within the node's instance directory. If the node is
// ephemeral, a memory database is returned.
func (n *Node) OpenDatabase(name string, cache, handles int) (ethdb.Database, error) {
if n.config.DataDir == "" {
return ethdb.NewMemDatabase()
}
return ethdb.NewLDBDatabase(n.config.resolvePath(name), cache, handles)
}

// ResolvePath returns the absolute path of a resource in the instance directory.
func (n *Node) ResolvePath(x string) string {
return n.config.resolvePath(x)
}

// apis returns the collection of RPC descriptors this node offers.
func (n *Node) apis() []rpc.API {
return []rpc.API{
{
Namespace: "admin",
Version:   "1.0",
Service:   NewPrivateAdminAPI(n),
```

```
}, {
Namespace: "admin",
Version:   "1.0",
Service:   NewPublicAdminAPI(n),
Public:    true,
}, {
Namespace: "debug",
Version:   "1.0",
Service:   debug.Handler,
}, {
Namespace: "debug",
Version:   "1.0",
Service:   NewPublicDebugAPI(n),
Public:    true,
}, {
Namespace: "web3",
Version:   "1.0",
Service:   NewPublicWeb3API(n),
Public:    true,
},
}
}
```

76:F:\git\coin\ethereum\go-ethereum\node\node_example_test.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package node_test

import (
"fmt"
"log"

"github.com/ethereum/go-ethereum/node"
"github.com/ethereum/go-ethereum/p2p"
"github.com/ethereum/go-ethereum/rpc"
)

// SampleService is a trivial network service that can be attached to a node for
// life cycle management.
//
// The following methods are needed to implement a node.Service:
//  - Protocols() []p2p.Protocol - devp2p protocols the service can communicate on
```

```
//  - APIs() []rpc.API           - api methods the service wants to expose on rpc channels
//  - Start() error              - method invoked when the node is ready to start the service
//  - Stop() error               - method invoked when the node terminates the service
type SampleService struct{}

func (s *SampleService) Protocols() []p2p.Protocol { return nil }
func (s *SampleService) APIs() []rpc.API           { return nil }
func (s *SampleService) Start(*p2p.Server) error   { fmt.Println("Service starting..."); return nil }
func (s *SampleService) Stop() error               { fmt.Println("Service stopping..."); return nil }

func ExampleService() {
// Create a network node to run protocols with the default values.
stack, err := node.New(&node.Config{})
if err != nil {
log.Fatalf("Failed to create network node: %v", err)
}
// Create and register a simple network service. This is done through the definition
// of a node.ServiceConstructor that will instantiate a node.Service. The reason for
// the factory method approach is to support service restarts without relying on the
// individual implementations' support for such operations.
constructor := func(context *node.ServiceContext) (node.Service, error) {
return new(SampleService), nil
}
if err := stack.Register(constructor); err != nil {
log.Fatalf("Failed to register service: %v", err)
}
// Boot up the entire protocol stack, do a restart and terminate
if err := stack.Start(); err != nil {
log.Fatalf("Failed to start the protocol stack: %v", err)
}
if err := stack.Restart(); err != nil {
log.Fatalf("Failed to restart the protocol stack: %v", err)
}
if err := stack.Stop(); err != nil {
log.Fatalf("Failed to stop the protocol stack: %v", err)
}
// Output:
// Service starting...
// Service stopping...
// Service starting...
// Service stopping...
}
```

77:F:\git\coin\ethereum\go-ethereum\node\node_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package node

import (
"errors"
"io/ioutil"
"os"
"reflect"
"testing"
"time"

"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/p2p"
"github.com/ethereum/go-ethereum/rpc"
)

var (
testNodeKey, _ = crypto.GenerateKey()
)

func testNodeConfig() *Config {
return &Config{
Name: "test node",
P2P: p2p.Config{PrivateKey: testNodeKey},
}
}

// Tests that an empty protocol stack can be started, restarted and stopped.
func TestNodeLifeCycle(t *testing.T) {
stack, err := New(testNodeConfig())
if err != nil {
t.Fatalf("failed to create protocol stack: %v", err)
}
// Ensure that a stopped node can be stopped again
for i := 0; i < 3; i++ {
if err := stack.Stop(); err != ErrNodeStopped {
t.Fatalf("iter %d: stop failure mismatch: have %v, want %v", i, err, ErrNodeStopped)
}
}

```
// Ensure that a node can be successfully started, but only once
if err := stack.Start(); err != nil {
t.Fatalf("failed to start node: %v", err)
}
if err := stack.Start(); err != ErrNodeRunning {
t.Fatalf("start failure mismatch: have %v, want %v ", err, ErrNodeRunning)
}
// Ensure that a node can be restarted arbitrarily many times
for i := 0; i < 3; i++ {
if err := stack.Restart(); err != nil {
t.Fatalf("iter %d: failed to restart node: %v", i, err)
}
}
// Ensure that a node can be stopped, but only once
if err := stack.Stop(); err != nil {
t.Fatalf("failed to stop node: %v", err)
}
if err := stack.Stop(); err != ErrNodeStopped {
t.Fatalf("stop failure mismatch: have %v, want %v ", err, ErrNodeStopped)
}
}

// Tests that if the data dir is already in use, an appropriate error is returned.
func TestNodeUsedDataDir(t *testing.T) {
// Create a temporary folder to use as the data directory
dir, err := ioutil.TempDir("", "")
if err != nil {
t.Fatalf("failed to create temporary data directory: %v", err)
}
defer os.RemoveAll(dir)

// Create a new node based on the data directory
original, err := New(&Config{DataDir: dir})
if err != nil {
t.Fatalf("failed to create original protocol stack: %v", err)
}
if err := original.Start(); err != nil {
t.Fatalf("failed to start original protocol stack: %v", err)
}
defer original.Stop()

// Create a second node based on the same data directory and ensure failure
```

```
duplicate, err := New(&Config{DataDir: dir})
if err != nil {
t.Fatalf("failed to create duplicate protocol stack: %v", err)
}
if err := duplicate.Start(); err != ErrDatadirUsed {
t.Fatalf("duplicate datadir failure mismatch: have %v, want %v", err, ErrDatadirUsed)
}
}

// Tests whether services can be registered and duplicates caught.
func TestServiceRegistry(t *testing.T) {
stack, err := New(testNodeConfig())
if err != nil {
t.Fatalf("failed to create protocol stack: %v", err)
}
// Register a batch of unique services and ensure they start successfully
services := []ServiceConstructor{NewNoopServiceA, NewNoopServiceB, NewNoopServiceC}
for i, constructor := range services {
if err := stack.Register(constructor); err != nil {
t.Fatalf("service #%d: registration failed: %v", i, err)
}
}
if err := stack.Start(); err != nil {
t.Fatalf("failed to start original service stack: %v", err)
}
if err := stack.Stop(); err != nil {
t.Fatalf("failed to stop original service stack: %v", err)
}
// Duplicate one of the services and retry starting the node
if err := stack.Register(NewNoopServiceB); err != nil {
t.Fatalf("duplicate registration failed: %v", err)
}
if err := stack.Start(); err == nil {
t.Fatalf("duplicate service started")
} else {
if _, ok := err.(*DuplicateServiceError); !ok {
t.Fatalf("duplicate error mismatch: have %v, want %v", err, DuplicateServiceError{})
}
}
}

// Tests that registered services get started and stopped correctly.
```

```
func TestServiceLifeCycle(t *testing.T) {
stack, err := New(testNodeConfig())
if err != nil {
t.Fatalf("failed to create protocol stack: %v", err)
}
// Register a batch of life-cycle instrumented services
services := map[string]InstrumentingWrapper{
"A": InstrumentedServiceMakerA,
"B": InstrumentedServiceMakerB,
"C": InstrumentedServiceMakerC,
}
started := make(map[string]bool)
stopped := make(map[string]bool)

for id, maker := range services {
id := id // Closure for the constructor
constructor := func(*ServiceContext) (Service, error) {
return &InstrumentedService{
startHook: func(*p2p.Server) { started[id] = true },
stopHook:  func() { stopped[id] = true },
}, nil
}
if err := stack.Register(maker(constructor)); err != nil {
t.Fatalf("service %s: registration failed: %v", id, err)
}
}
// Start the node and check that all services are running
if err := stack.Start(); err != nil {
t.Fatalf("failed to start protocol stack: %v", err)
}
for id := range services {
if !started[id] {
t.Fatalf("service %s: freshly started service not running", id)
}
if stopped[id] {
t.Fatalf("service %s: freshly started service already stopped", id)
}
}
// Stop the node and check that all services have been stopped
if err := stack.Stop(); err != nil {
t.Fatalf("failed to stop protocol stack: %v", err)
}
```

```
for id := range services {
if !stopped[id] {
t.Fatalf("service %s: freshly terminated service still running", id)
}
}
}

// Tests that services are restarted cleanly as new instances.
func TestServiceRestarts(t *testing.T) {
stack, err := New(testNodeConfig())
if err != nil {
t.Fatalf("failed to create protocol stack: %v", err)
}
// Define a service that does not support restarts
var (
running bool
started int
)
constructor := func(*ServiceContext) (Service, error) {
running = false

return &InstrumentedService{
startHook: func(*p2p.Server) {
if running {
panic("already running")
}
running = true
started++
},
}, nil
}
// Register the service and start the protocol stack
if err := stack.Register(constructor); err != nil {
t.Fatalf("failed to register the service: %v", err)
}
if err := stack.Start(); err != nil {
t.Fatalf("failed to start protocol stack: %v", err)
}
defer stack.Stop()

if !running || started != 1 {
t.Fatalf("running/started mismatch: have %v/%d, want true/1", running, started)
```

```
}
// Restart the stack a few times and check successful service restarts
for i := 0; i < 3; i++ {
if err := stack.Restart(); err != nil {
t.Fatalf("iter %d: failed to restart stack: %v", i, err)
}
}
if !running || started != 4 {
t.Fatalf("running/started mismatch: have %v/%d, want true/4", running, started)
}
}

// Tests that if a service fails to initialize itself, none of the other services
// will be allowed to even start.
func TestServiceConstructionAbortion(t *testing.T) {
stack, err := New(testNodeConfig())
if err != nil {
t.Fatalf("failed to create protocol stack: %v", err)
}
// Define a batch of good services
services := map[string]InstrumentingWrapper{
"A": InstrumentedServiceMakerA,
"B": InstrumentedServiceMakerB,
"C": InstrumentedServiceMakerC,
}
started := make(map[string]bool)
for id, maker := range services {
id := id // Closure for the constructor
constructor := func(*ServiceContext) (Service, error) {
return &InstrumentedService{
startHook: func(*p2p.Server) { started[id] = true },
}, nil
}
if err := stack.Register(maker(constructor)); err != nil {
t.Fatalf("service %s: registration failed: %v", id, err)
}
}
// Register a service that fails to construct itself
failure := errors.New("fail")
failer := func(*ServiceContext) (Service, error) {
return nil, failure
}
```

```
if err := stack.Register(failer); err != nil {
t.Fatalf("failer registration failed: %v", err)
}
// Start the protocol stack and ensure none of the services get started
for i := 0; i < 100; i++ {
if err := stack.Start(); err != failure {
t.Fatalf("iter %d: stack startup failure mismatch: have %v, want %v", i, err, failure)
}
for id := range services {
if started[id] {
t.Fatalf("service %s: started should not have", id)
}
delete(started, id)
}
}
}

// Tests that if a service fails to start, all others started before it will be
// shut down.
func TestServiceStartupAbortion(t *testing.T) {
stack, err := New(testNodeConfig())
if err != nil {
t.Fatalf("failed to create protocol stack: %v", err)
}
// Register a batch of good services
services := map[string]InstrumentingWrapper{
"A": InstrumentedServiceMakerA,
"B": InstrumentedServiceMakerB,
"C": InstrumentedServiceMakerC,
}
started := make(map[string]bool)
stopped := make(map[string]bool)

for id, maker := range services {
id := id // Closure for the constructor
constructor := func(*ServiceContext) (Service, error) {
return &InstrumentedService{
startHook: func(*p2p.Server) { started[id] = true },
stopHook:  func() { stopped[id] = true },
}, nil
}
if err := stack.Register(maker(constructor)); err != nil {
```

```
t.Fatalf("service %s: registration failed: %v", id, err)
}
}
// Register a service that fails to start
failure := errors.New("fail")
failer := func(*ServiceContext) (Service, error) {
return &InstrumentedService{
start: failure,
}, nil
}
if err := stack.Register(failer); err != nil {
t.Fatalf("failer registration failed: %v", err)
}
// Start the protocol stack and ensure all started services stop
for i := 0; i < 100; i++ {
if err := stack.Start(); err != failure {
t.Fatalf("iter %d: stack startup failure mismatch: have %v, want %v", i, err, failure)
}
for id := range services {
if started[id] && !stopped[id] {
t.Fatalf("service %s: started but not stopped", id)
}
delete(started, id)
delete(stopped, id)
}
}
}

// Tests that even if a registered service fails to shut down cleanly, it does
// not influece the rest of the shutdown invocations.
func TestServiceTerminationGuarantee(t *testing.T) {
stack, err := New(testNodeConfig())
if err != nil {
t.Fatalf("failed to create protocol stack: %v", err)
}
// Register a batch of good services
services := map[string]InstrumentingWrapper{
"A": InstrumentedServiceMakerA,
"B": InstrumentedServiceMakerB,
"C": InstrumentedServiceMakerC,
}
started := make(map[string]bool)
```

```
stopped := make(map[string]bool)

for id, maker := range services {
id := id // Closure for the constructor
constructor := func(*ServiceContext) (Service, error) {
return &InstrumentedService{
startHook: func(*p2p.Server) { started[id] = true },
stopHook:  func() { stopped[id] = true },
}, nil
}
if err := stack.Register(maker(constructor)); err != nil {
t.Fatalf("service %s: registration failed: %v", id, err)
}
}
// Register a service that fails to shot down cleanly
failure := errors.New("fail")
failer := func(*ServiceContext) (Service, error) {
return &InstrumentedService{
stop: failure,
}, nil
}
if err := stack.Register(failer); err != nil {
t.Fatalf("failer registration failed: %v", err)
}
// Start the protocol stack, and ensure that a failing shut down terminates all
for i := 0; i < 100; i++ {
// Start the stack and make sure all is online
if err := stack.Start(); err != nil {
t.Fatalf("iter %d: failed to start protocol stack: %v", i, err)
}
for id := range services {
if !started[id] {
t.Fatalf("iter %d, service %s: service not running", i, id)
}
if stopped[id] {
t.Fatalf("iter %d, service %s: service already stopped", i, id)
}
}
// Stop the stack, verify failure and check all terminations
err := stack.Stop()
if err, ok := err.(*StopError); !ok {
t.Fatalf("iter %d: termination failure mismatch: have %v, want StopError", i, err)
```

```
} else {
failer := reflect.TypeOf(&InstrumentedService{})
if err.Services[failer] != failure {
t.Fatalf("iter %d: failer termination failure mismatch: have %v, want %v", i, err.Services[failer],
failure)
}
if len(err.Services) != 1 {
t.Fatalf("iter %d: failure count mismatch: have %d, want %d", i, len(err.Services), 1)
}
}
for id := range services {
if !stopped[id] {
t.Fatalf("iter %d, service %s: service not terminated", i, id)
}
delete(started, id)
delete(stopped, id)
}
}
}

// TestServiceRetrieval tests that individual services can be retrieved.
func TestServiceRetrieval(t *testing.T) {
// Create a simple stack and register two service types
stack, err := New(testNodeConfig())
if err != nil {
t.Fatalf("failed to create protocol stack: %v", err)
}
if err := stack.Register(NewNoopService); err != nil {
t.Fatalf("noop service registration failed: %v", err)
}
if err := stack.Register(NewInstrumentedService); err != nil {
t.Fatalf("instrumented service registration failed: %v", err)
}
// Make sure none of the services can be retrieved until started
var noopServ *NoopService
if err := stack.Service(&noopServ); err != ErrNodeStopped {
t.Fatalf("noop service retrieval mismatch: have %v, want %v", err, ErrNodeStopped)
}
var instServ *InstrumentedService
if err := stack.Service(&instServ); err != ErrNodeStopped {
t.Fatalf("instrumented service retrieval mismatch: have %v, want %v", err, ErrNodeStopped)
}
```

```
// Start the stack and ensure everything is retrievable now
if err := stack.Start(); err != nil {
t.Fatalf("failed to start stack: %v", err)
}
defer stack.Stop()

if err := stack.Service(&noopServ); err != nil {
t.Fatalf("noop service retrieval mismatch: have %v, want %v", err, nil)
}
if err := stack.Service(&instServ); err != nil {
t.Fatalf("instrumented service retrieval mismatch: have %v, want %v", err, nil)
}
}

// Tests that all protocols defined by individual services get launched.
func TestProtocolGather(t *testing.T) {
stack, err := New(testNodeConfig())
if err != nil {
t.Fatalf("failed to create protocol stack: %v", err)
}
// Register a batch of services with some configured number of protocols
services := map[string]struct {
Count int
Maker InstrumentingWrapper
}{
"Zero Protocols":  {0, InstrumentedServiceMakerA},
"Single Protocol": {1, InstrumentedServiceMakerB},
"Many Protocols":  {25, InstrumentedServiceMakerC},
}
for id, config := range services {
protocols := make([]p2p.Protocol, config.Count)
for i := 0; i < len(protocols); i++ {
protocols[i].Name = id
protocols[i].Version = uint(i)
}
constructor := func(*ServiceContext) (Service, error) {
return &InstrumentedService{
protocols: protocols,
}, nil
}
if err := stack.Register(config.Maker(constructor)); err != nil {
t.Fatalf("service %s: registration failed: %v", id, err)
```

```
}
}
// Start the services and ensure all protocols start successfully
if err := stack.Start(); err != nil {
t.Fatalf("failed to start protocol stack: %v", err)
}
defer stack.Stop()

protocols := stack.Server().Protocols
if len(protocols) != 26 {
t.Fatalf("mismatching number of protocols launched: have %d, want %d", len(protocols), 26)
}
for id, config := range services {
for ver := 0; ver < config.Count; ver++ {
launched := false
for i := 0; i < len(protocols); i++ {
if protocols[i].Name == id && protocols[i].Version == uint(ver) {
launched = true
break
}
}
if !launched {
t.Errorf("configured protocol not launched: %s v%d", id, ver)
}
}
}
}

// Tests that all APIs defined by individual services get exposed.
func TestAPIGather(t *testing.T) {
stack, err := New(testNodeConfig())
if err != nil {
t.Fatalf("failed to create protocol stack: %v", err)
}
// Register a batch of services with some configured APIs
calls := make(chan string, 1)
makeAPI := func(result string) *OneMethodApi {
return &OneMethodApi{fun: func() { calls <- result }}
}
services := map[string]struct {
APIs  []rpc.API
Maker InstrumentingWrapper
```

```
}{
"Zero APIs": {
[]rpc.API{}, InstrumentedServiceMakerA},
"Single API": {
[]rpc.API{
{Namespace: "single", Version: "1", Service: makeAPI("single.v1"), Public: true},
}, InstrumentedServiceMakerB},
"Many APIs": {
[]rpc.API{
{Namespace: "multi", Version: "1", Service: makeAPI("multi.v1"), Public: true},
{Namespace: "multi.v2", Version: "2", Service: makeAPI("multi.v2"), Public: true},
{Namespace: "multi.v2.nested", Version: "2", Service: makeAPI("multi.v2.nested"), Public: true},
}, InstrumentedServiceMakerC},
}

for id, config := range services {
config := config
constructor := func(*ServiceContext) (Service, error) {
return &InstrumentedService{apis: config.APIs}, nil
}
if err := stack.Register(config.Maker(constructor)); err != nil {
t.Fatalf("service %s: registration failed: %v", id, err)
}
}
// Start the services and ensure all API start successfully
if err := stack.Start(); err != nil {
t.Fatalf("failed to start protocol stack: %v", err)
}
defer stack.Stop()

// Connect to the RPC server and verify the various registered endpoints
client, err := stack.Attach()
if err != nil {
t.Fatalf("failed to connect to the inproc API server: %v", err)
}
defer client.Close()

tests := []struct {
Method string
Result string
}{
{"single_theOneMethod", "single.v1"},
```

```
{"multi_theOneMethod", "multi.v1"},
{"multi.v2_theOneMethod", "multi.v2"},
{"multi.v2.nested_theOneMethod", "multi.v2.nested"},
}
for i, test := range tests {
if err := client.Call(nil, test.Method); err != nil {
t.Errorf("test %d: API request failed: %v", i, err)
}
select {
case result := <-calls:
if result != test.Result {
t.Errorf("test %d: result mismatch: have %s, want %s", i, result, test.Result)
}
case <-time.After(time.Second):
t.Fatalf("test %d: rpc execution timeout", i)
}
}
}
```

78:F:\git\coin\ethereum\go-ethereum\node\service.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package node

import (
"reflect"

"github.com/ethereum/go-ethereum/accounts"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/p2p"
"github.com/ethereum/go-ethereum/rpc"
)

// ServiceContext is a collection of service independent options inherited from
// the protocol stack, that is passed to all constructors to be optionally used;
// as well as utility methods to operate on the service environment.
type ServiceContext struct {
config         *Config
services       map[reflect.Type]Service // Index of the already constructed services
EventMux       *event.TypeMux           // Event multiplexer used for decoupled notifications
AccountManager *accounts.Manager        // Account manager created by the node.
```

```
}

// OpenDatabase opens an existing database with the given name (or creates one
// if no previous can be found) from within the node's data directory. If the
// node is an ephemeral one, a memory database is returned.
func (ctx *ServiceContext) OpenDatabase(name string, cache int, handles int) (ethdb.Database,
error) {
if ctx.config.DataDir == "" {
return ethdb.NewMemDatabase()
}
db, err := ethdb.NewLDBDatabase(ctx.config.resolvePath(name), cache, handles)
if err != nil {
return nil, err
}
return db, nil
}

// ResolvePath resolves a user path into the data directory if that was relative
// and if the user actually uses persistent storage. It will return an empty string
// for emphemeral storage and the user's own input for absolute paths.
func (ctx *ServiceContext) ResolvePath(path string) string {
return ctx.config.resolvePath(path)
}

// Service retrieves a currently running service registered of a specific type.
func (ctx *ServiceContext) Service(service interface{}) error {
element := reflect.ValueOf(service).Elem()
if running, ok := ctx.services[element.Type()]; ok {
element.Set(reflect.ValueOf(running))
return nil
}
return ErrServiceUnknown
}

// ServiceConstructor is the function signature of the constructors needed to be
// registered for service instantiation.
type ServiceConstructor func(ctx *ServiceContext) (Service, error)

// Service is an individual protocol that can be registered into a node.
//
// Notes:
//
```

```
// • Service life-cycle management is delegated to the node. The service is allowed to
// initialize itself upon creation, but no goroutines should be spun up outside of the
// Start method.
//
// • Restart logic is not required as the node will create a fresh instance
// every time a service is started.
type Service interface {
// Protocols retrieves the P2P protocols the service wishes to start.
Protocols() []p2p.Protocol

// APIs retrieves the list of RPC descriptors the service provides
APIs() []rpc.API

// Start is called after all services have been constructed and the networking
// layer was also initialized to spawn any goroutines required by the service.
Start(server *p2p.Server) error

// Stop terminates all goroutines belonging to the service, blocking until they
// are all terminated.
Stop() error
}

79:F:\git\coin\ethereum\go-ethereum\node\service_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package node

import (
"fmt"
"io/ioutil"
"os"
"path/filepath"
"testing"
)

// Tests that databases are correctly created persistent or ephemeral based on
// the configured service context.
func TestContextDatabases(t *testing.T) {
// Create a temporary folder and ensure no database is contained within
dir, err := ioutil.TempDir("", "")
if err != nil {
t.Fatalf("failed to create temporary data directory: %v", err)
```

```
}
defer os.RemoveAll(dir)

if _, err := os.Stat(filepath.Join(dir, "database")); err == nil {
t.Fatalf("non-created database already exists")
}
// Request the opening/creation of a database and ensure it persists to disk
ctx := &ServiceContext{config: &Config{Name: "unit-test", DataDir: dir}}
db, err := ctx.OpenDatabase("persistent", 0, 0)
if err != nil {
t.Fatalf("failed to open persistent database: %v", err)
}
db.Close()

if _, err := os.Stat(filepath.Join(dir, "unit-test", "persistent")); err != nil {
t.Fatalf("persistent database doesn't exists: %v", err)
}
// Request th opening/creation of an ephemeral database and ensure it's not persisted
ctx = &ServiceContext{config: &Config{DataDir: ""}}
db, err = ctx.OpenDatabase("ephemeral", 0, 0)
if err != nil {
t.Fatalf("failed to open ephemeral database: %v", err)
}
db.Close()

if _, err := os.Stat(filepath.Join(dir, "ephemeral")); err == nil {
t.Fatalf("ephemeral database exists")
}
}

// Tests that already constructed services can be retrieves by later ones.
func TestContextServices(t *testing.T) {
stack, err := New(testNodeConfig())
if err != nil {
t.Fatalf("failed to create protocol stack: %v", err)
}
// Define a verifier that ensures a NoopA is before it and NoopB after
verifier := func(ctx *ServiceContext) (Service, error) {
var objA *NoopServiceA
if ctx.Service(&objA) != nil {
return nil, fmt.Errorf("former service not found")
}
```

```
var objB *NoopServiceB
if err := ctx.Service(&objB); err != ErrServiceUnknown {
return nil, fmt.Errorf("latters lookup error mismatch: have %v, want %v", err, ErrServiceUnknown)
}
return new(NoopService), nil
}
// Register the collection of services
if err := stack.Register(NewNoopServiceA); err != nil {
t.Fatalf("former failed to register service: %v", err)
}
if err := stack.Register(verifier); err != nil {
t.Fatalf("failed to register service verifier: %v", err)
}
if err := stack.Register(NewNoopServiceB); err != nil {
t.Fatalf("latter failed to register service: %v", err)
}
// Start the protocol stack and ensure services are constructed in order
if err := stack.Start(); err != nil {
t.Fatalf("failed to start stack: %v", err)
}
defer stack.Stop()
}
```

80:F:\git\coin\ethereum\go-ethereum\node\utils_test.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains a batch of utility type declarations used by the tests. As the node
// operates on unique types, a lot of them are needed to check various features.

package node

import (
"reflect"

"github.com/ethereum/go-ethereum/p2p"
"github.com/ethereum/go-ethereum/rpc"
)

// NoopService is a trivial implementation of the Service interface.
type NoopService struct{}

func (s *NoopService) Protocols() []p2p.Protocol { return nil }
```

```
func (s *NoopService) APIs() []rpc.API           { return nil }
func (s *NoopService) Start(*p2p.Server) error   { return nil }
func (s *NoopService) Stop() error               { return nil }

func NewNoopService(*ServiceContext) (Service, error) { return new(NoopService), nil }

// Set of services all wrapping the base NoopService resulting in the same method
// signatures but different outer types.
type NoopServiceA struct{ NoopService }
type NoopServiceB struct{ NoopService }
type NoopServiceC struct{ NoopService }
type NoopServiceD struct{ NoopService }

func NewNoopServiceA(*ServiceContext) (Service, error) { return new(NoopServiceA), nil }
func NewNoopServiceB(*ServiceContext) (Service, error) { return new(NoopServiceB), nil }
func NewNoopServiceC(*ServiceContext) (Service, error) { return new(NoopServiceC), nil }
func NewNoopServiceD(*ServiceContext) (Service, error) { return new(NoopServiceD), nil }

// InstrumentedService is an implementation of Service for which all interface
// methods can be instrumented both return value as well as event hook wise.
type InstrumentedService struct {
protocols []p2p.Protocol
apis      []rpc.API
start     error
stop      error

protocolsHook func()
startHook     func(*p2p.Server)
stopHook      func()
}

func NewInstrumentedService(*ServiceContext) (Service, error) { return
new(InstrumentedService), nil }

func (s *InstrumentedService) Protocols() []p2p.Protocol {
if s.protocolsHook != nil {
s.protocolsHook()
}
return s.protocols
}

func (s *InstrumentedService) APIs() []rpc.API {
```

```
return s.apis
}

func (s *InstrumentedService) Start(server *p2p.Server) error {
if s.startHook != nil {
s.startHook(server)
}
return s.start
}

func (s *InstrumentedService) Stop() error {
if s.stopHook != nil {
s.stopHook()
}
return s.stop
}

// InstrumentingWrapper is a method to specialize a service constructor returning
// a generic InstrumentedService into one returning a wrapping specific one.
type InstrumentingWrapper func(base ServiceConstructor) ServiceConstructor

func InstrumentingWrapperMaker(base ServiceConstructor, kind reflect.Type) ServiceConstructor
{
return func(ctx *ServiceContext) (Service, error) {
obj, err := base(ctx)
if err != nil {
return nil, err
}
wrapper := reflect.New(kind)
wrapper.Elem().Field(0).Set(reflect.ValueOf(obj).Elem())

return wrapper.Interface().(Service), nil
}
}

// Set of services all wrapping the base InstrumentedService resulting in the
// same method signatures but different outer types.
type InstrumentedServiceA struct{ InstrumentedService }
type InstrumentedServiceB struct{ InstrumentedService }
type InstrumentedServiceC struct{ InstrumentedService }

func InstrumentedServiceMakerA(base ServiceConstructor) ServiceConstructor {
```

```
return InstrumentingWrapperMaker(base, reflect.TypeOf(InstrumentedServiceA{}))
}

func InstrumentedServiceMakerB(base ServiceConstructor) ServiceConstructor {
return InstrumentingWrapperMaker(base, reflect.TypeOf(InstrumentedServiceB{}))
}

func InstrumentedServiceMakerC(base ServiceConstructor) ServiceConstructor {
return InstrumentingWrapperMaker(base, reflect.TypeOf(InstrumentedServiceC{}))
}

// OneMethodApi is a single-method API handler to be returned by test services.
type OneMethodApi struct {
fun func()
}

func (api *OneMethodApi) TheOneMethod() {
if api.fun != nil {
api.fun()
}
}
```

81:F:\git\coin\ethereum\go-ethereum\p2p\dial.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package p2p

import (
"container/heap"
"crypto/rand"
"errors"
"fmt"
"net"
"time"

"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/p2p/discover"
"github.com/ethereum/go-ethereum/p2p/netutil"
)

const (
// This is the amount of time spent waiting in between
```

```
// redialing a certain node.
dialHistoryExpiration = 30 * time.Second

// Discovery lookups are throttled and can only run
// once every few seconds.
lookupInterval = 4 * time.Second

// If no peers are found for this amount of time, the initial bootnodes are
// attempted to be connected.
fallbackInterval = 20 * time.Second

// Endpoint resolution is throttled with bounded backoff.
initialResolveDelay = 60 * time.Second
maxResolveDelay     = time.Hour
)

// dialstate schedules dials and discovery lookups.
// it get's a chance to compute new tasks on every iteration
// of the main loop in Server.run.
type dialstate struct {
maxDynDials int
ntab        discoverTable
netrestrict *netutil.Netlist

lookupRunning bool
dialing       map[discover.NodeID]connFlag
lookupBuf     []*discover.Node // current discovery lookup results
randomNodes   []*discover.Node // filled from Table
static        map[discover.NodeID]*dialTask
hist          *dialHistory

start     time.Time        // time when the dialer was first used
bootnodes []*discover.Node // default dials when there are no peers
}

type discoverTable interface {
Self() *discover.Node
Close()
Resolve(target discover.NodeID) *discover.Node
Lookup(target discover.NodeID) []*discover.Node
ReadRandomNodes([]*discover.Node) int
}
```

```
// the dial history remembers recent dials.
type dialHistory []pastDial

// pastDial is an entry in the dial history.
type pastDial struct {
id  discover.NodeID
exp time.Time
}

type task interface {
Do(*Server)
}

// A dialTask is generated for each node that is dialed. Its
// fields cannot be accessed while the task is running.
type dialTask struct {
flags       connFlag
dest        *discover.Node
lastResolved time.Time
resolveDelay time.Duration
}

// discoverTask runs discovery table operations.
// Only one discoverTask is active at any time.
// discoverTask.Do performs a random lookup.
type discoverTask struct {
results []*discover.Node
}

// A waitExpireTask is generated if there are no other tasks
// to keep the loop in Server.run ticking.
type waitExpireTask struct {
time.Duration
}

func newDialState(static []*discover.Node, bootnodes []*discover.Node, ntab discoverTable,
maxdyn int, netrestrict *netutil.Netlist) *dialstate {
s := &dialstate{
maxDynDials: maxdyn,
ntab:        ntab,
netrestrict: netrestrict,
```

```
static:      make(map[discover.NodeID]*dialTask),
dialing:     make(map[discover.NodeID]connFlag),
bootnodes:   make([]*discover.Node, len(bootnodes)),
randomNodes: make([]*discover.Node, maxdyn/2),
hist:        new(dialHistory),
}
copy(s.bootnodes, bootnodes)
for _, n := range static {
s.addStatic(n)
}
return s
}

func (s *dialstate) addStatic(n *discover.Node) {
// This overwites the task instead of updating an existing
// entry, giving users the opportunity to force a resolve operation.
s.static[n.ID] = &dialTask{flags: staticDialedConn, dest: n}
}

func (s *dialstate) removeStatic(n *discover.Node) {
// This removes a task so future attempts to connect will not be made.
delete(s.static, n.ID)
}

func (s *dialstate) newTasks(nRunning int, peers map[discover.NodeID]*Peer, now time.Time)
[]task {
if s.start == (time.Time{}) {
s.start = now
}

var newtasks []task
addDial := func(flag connFlag, n *discover.Node) bool {
if err := s.checkDial(n, peers); err != nil {
log.Trace("Skipping dial candidate", "id", n.ID, "addr", &net.TCPAddr{IP: n.IP, Port: int(n.TCP)},
"err", err)
return false
}
s.dialing[n.ID] = flag
newtasks = append(newtasks, &dialTask{flags: flag, dest: n})
return true
}
```

```
// Compute number of dynamic dials necessary at this point.
needDynDials := s.maxDynDials
for _, p := range peers {
if p.rw.is(dynDialedConn) {
needDynDials--
}
}
for _, flag := range s.dialing {
if flag&dynDialedConn != 0 {
needDynDials--
}
}

// Expire the dial history on every invocation.
s.hist.expire(now)

// Create dials for static nodes if they are not connected.
for id, t := range s.static {
err := s.checkDial(t.dest, peers)
switch err {
case errNotWhitelisted, errSelf:
log.Warn("Removing static dial candidate", "id", t.dest.ID, "addr", &net.TCPAddr{IP: t.dest.IP, Port:
int(t.dest.TCP)}, "err", err)
delete(s.static, t.dest.ID)
case nil:
s.dialing[id] = t.flags
newtasks = append(newtasks, t)
}
}
// If we don't have any peers whatsoever, try to dial a random bootnode. This
// scenario is useful for the testnet (and private networks) where the discovery
// table might be full of mostly bad peers, making it hard to find good ones.
if len(peers) == 0 && len(s.bootnodes) > 0 && needDynDials > 0 && now.Sub(s.start) >
fallbackInterval {
bootnode := s.bootnodes[0]
s.bootnodes = append(s.bootnodes[:0], s.bootnodes[1:]...)
s.bootnodes = append(s.bootnodes, bootnode)

if addDial(dynDialedConn, bootnode) {
needDynDials--
}
}
```

```
// Use random nodes from the table for half of the necessary
// dynamic dials.
randomCandidates := needDynDials / 2
if randomCandidates > 0 {
n := s.ntab.ReadRandomNodes(s.randomNodes)
for i := 0; i < randomCandidates && i < n; i++ {
if addDial(dynDialedConn, s.randomNodes[i]) {
needDynDials--
}
}
}
// Create dynamic dials from random lookup results, removing tried
// items from the result buffer.
i := 0
for ; i < len(s.lookupBuf) && needDynDials > 0; i++ {
if addDial(dynDialedConn, s.lookupBuf[i]) {
needDynDials--
}
}
s.lookupBuf = s.lookupBuf[:copy(s.lookupBuf, s.lookupBuf[i:])]
// Launch a discovery lookup if more candidates are needed.
if len(s.lookupBuf) < needDynDials && !s.lookupRunning {
s.lookupRunning = true
newtasks = append(newtasks, &discoverTask{})
}

// Launch a timer to wait for the next node to expire if all
// candidates have been tried and no task is currently active.
// This should prevent cases where the dialer logic is not ticked
// because there are no pending events.
if nRunning == 0 && len(newtasks) == 0 && s.hist.Len() > 0 {
t := &waitExpireTask{s.hist.min().exp.Sub(now)}
newtasks = append(newtasks, t)
}
return newtasks
}

var (
errSelf             = errors.New("is self")
errAlreadyDialing   = errors.New("already dialing")
errAlreadyConnected = errors.New("already connected")
errRecentlyDialed   = errors.New("recently dialed")
```

```
errNotWhitelisted   = errors.New("not contained in netrestrict whitelist")
)

func (s *dialstate) checkDial(n *discover.Node, peers map[discover.NodeID]*Peer) error {
_, dialing := s.dialing[n.ID]
switch {
case dialing:
return errAlreadyDialing
case peers[n.ID] != nil:
return errAlreadyConnected
case s.ntab != nil && n.ID == s.ntab.Self().ID:
return errSelf
case s.netrestrict != nil && !s.netrestrict.Contains(n.IP):
return errNotWhitelisted
case s.hist.contains(n.ID):
return errRecentlyDialed
}
return nil
}

func (s *dialstate) taskDone(t task, now time.Time) {
switch t := t.(type) {
case *dialTask:
s.hist.add(t.dest.ID, now.Add(dialHistoryExpiration))
delete(s.dialing, t.dest.ID)
case *discoverTask:
s.lookupRunning = false
s.lookupBuf = append(s.lookupBuf, t.results...)
}
}

func (t *dialTask) Do(srv *Server) {
if t.dest.Incomplete() {
if !t.resolve(srv) {
return
}
}
success := t.dial(srv, t.dest)
// Try resolving the ID of static nodes if dialing failed.
if !success && t.flags&staticDialedConn != 0 {
if t.resolve(srv) {
t.dial(srv, t.dest)
```

```
}
}
}

// resolve attempts to find the current endpoint for the destination
// using discovery.
//
// Resolve operations are throttled with backoff to avoid flooding the
// discovery network with useless queries for nodes that don't exist.
// The backoff delay resets when the node is found.
func (t *dialTask) resolve(srv *Server) bool {
if srv.ntab == nil {
log.Debug("Can't resolve node", "id", t.dest.ID, "err", "discovery is disabled")
return false
}
if t.resolveDelay == 0 {
t.resolveDelay = initialResolveDelay
}
if time.Since(t.lastResolved) < t.resolveDelay {
return false
}
resolved := srv.ntab.Resolve(t.dest.ID)
t.lastResolved = time.Now()
if resolved == nil {
t.resolveDelay *= 2
if t.resolveDelay > maxResolveDelay {
t.resolveDelay = maxResolveDelay
}
log.Debug("Resolving node failed", "id", t.dest.ID, "newdelay", t.resolveDelay)
return false
}
// The node was found.
t.resolveDelay = initialResolveDelay
t.dest = resolved
log.Debug("Resolved node", "id", t.dest.ID, "addr", &net.TCPAddr{IP: t.dest.IP, Port:
int(t.dest.TCP)})
return true
}

// dial performs the actual connection attempt.
func (t *dialTask) dial(srv *Server, dest *discover.Node) bool {
addr := &net.TCPAddr{IP: dest.IP, Port: int(dest.TCP)}
```

```
fd, err := srv.Dialer.Dial("tcp", addr.String())
if err != nil {
log.Trace("Dial error", "task", t, "err", err)
return false
}
mfd := newMeteredConn(fd, false)
srv.setupConn(mfd, t.flags, dest)
return true
}

func (t *dialTask) String() string {
return fmt.Sprintf("%v %x %v:%d", t.flags, t.dest.ID[:8], t.dest.IP, t.dest.TCP)
}

func (t *discoverTask) Do(srv *Server) {
// newTasks generates a lookup task whenever dynamic dials are
// necessary. Lookups need to take some time, otherwise the
// event loop spins too fast.
next := srv.lastLookup.Add(lookupInterval)
if now := time.Now(); now.Before(next) {
time.Sleep(next.Sub(now))
}
srv.lastLookup = time.Now()
var target discover.NodeID
rand.Read(target[:])
t.results = srv.ntab.Lookup(target)
}

func (t *discoverTask) String() string {
s := "discovery lookup"
if len(t.results) > 0 {
s += fmt.Sprintf(" (%d results)", len(t.results))
}
return s
}

func (t waitExpireTask) Do(*Server) {
time.Sleep(t.Duration)
}
func (t waitExpireTask) String() string {
return fmt.Sprintf("wait for dial hist expire (%v)", t.Duration)
}
```

```
// Use only these methods to access or modify dialHistory.
func (h dialHistory) min() pastDial {
return h[0]
}
func (h *dialHistory) add(id discover.NodeID, exp time.Time) {
heap.Push(h, pastDial{id, exp})
}
func (h dialHistory) contains(id discover.NodeID) bool {
for _, v := range h {
if v.id == id {
return true
}
}
return false
}
func (h *dialHistory) expire(now time.Time) {
for h.Len() > 0 && h.min().exp.Before(now) {
heap.Pop(h)
}
}

// heap.Interface boilerplate
func (h dialHistory) Len() int           { return len(h) }
func (h dialHistory) Less(i, j int) bool { return h[i].exp.Before(h[j].exp) }
func (h dialHistory) Swap(i, j int)      { h[i], h[j] = h[j], h[i] }
func (h *dialHistory) Push(x interface{}) {
*h = append(*h, x.(pastDial))
}
func (h *dialHistory) Pop() interface{} {
old := *h
n := len(old)
x := old[n-1]
*h = old[0 : n-1]
return x
}

82:F:\git\coin\ethereum\go-ethereum\p2p\dial_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package p2p
```

```
import (
"encoding/binary"
"net"
"reflect"
"testing"
"time"

"github.com/davecgh/go-spew/spew"
"github.com/ethereum/go-ethereum/p2p/discover"
"github.com/ethereum/go-ethereum/p2p/netutil"
)

func init() {
spew.Config.Indent = "\t"
}

type dialtest struct {
init   *dialstate // state before and after the test.
rounds []round
}

type round struct {
peers []*Peer // current peer set
done  []task  // tasks that got done this round
new   []task  // the result must match this one
}

func runDialTest(t *testing.T, test dialtest) {
var (
vtime   time.Time
running int
)
pm := func(ps []*Peer) map[discover.NodeID]*Peer {
m := make(map[discover.NodeID]*Peer)
for _, p := range ps {
m[p.rw.id] = p
}
return m
}
for i, round := range test.rounds {
for _, task := range round.done {
running--
```

```
if running < 0 {
panic("running task counter underflow")
}
test.init.taskDone(task, vtime)
}

new := test.init.newTasks(running, pm(round.peers), vtime)
if !sametasks(new, round.new) {
t.Errorf("round %d: new tasks mismatch:\ngot %v\nwant %v\nstate: %v\nrunning: %v\n",
i, spew.Sdump(new), spew.Sdump(round.new), spew.Sdump(test.init), spew.Sdump(running))
}

// Time advances by 16 seconds on every round.
vtime = vtime.Add(16 * time.Second)
running += len(new)
}
}

type fakeTable []*discover.Node

func (t fakeTable) Self() *discover.Node                    { return new(discover.Node) }
func (t fakeTable) Close()                                  {}
func (t fakeTable) Lookup(discover.NodeID) []*discover.Node  { return nil }
func (t fakeTable) Resolve(discover.NodeID) *discover.Node   { return nil }
func (t fakeTable) ReadRandomNodes(buf []*discover.Node) int { return copy(buf, t) }

// This test checks that dynamic dials are launched from discovery results.
func TestDialStateDynDial(t *testing.T) {
runDialTest(t, dialtest{
init: newDialState(nil, nil, fakeTable{}, 5, nil),
rounds: []round{
// A discovery query is launched.
{
peers: []*Peer{
{rw: &conn{flags: staticDialedConn, id: uintID(0)}},
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
},
new: []task{&discoverTask{}},
},
// Dynamic dials are launched when it completes.
{
```

```
peers: []*Peer{
{rw: &conn{flags: staticDialedConn, id: uintID(0)}},
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
},
done: []task{
&discoverTask{results: []*discover.Node{
{ID: uintID(2)}, // this one is already connected and not dialed.
{ID: uintID(3)},
{ID: uintID(4)},
{ID: uintID(5)},
{ID: uintID(6)}, // these are not tried because max dyn dials is 5
{ID: uintID(7)}, // ...
}},
},
new: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(3)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(4)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(5)}},
},
},
// Some of the dials complete but no new ones are launched yet because
// the sum of active dial count and dynamic peer count is == maxDynDials.
{
peers: []*Peer{
{rw: &conn{flags: staticDialedConn, id: uintID(0)}},
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
{rw: &conn{flags: dynDialedConn, id: uintID(3)}},
{rw: &conn{flags: dynDialedConn, id: uintID(4)}},
},
done: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(3)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(4)}},
},
},
// No new dial tasks are launched in the this round because
// maxDynDials has been reached.
{
peers: []*Peer{
{rw: &conn{flags: staticDialedConn, id: uintID(0)}},
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
```

```
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
{rw: &conn{flags: dynDialedConn, id: uintID(3)}},
{rw: &conn{flags: dynDialedConn, id: uintID(4)}},
{rw: &conn{flags: dynDialedConn, id: uintID(5)}},
},
done: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(5)}},
},
new: []task{
&waitExpireTask{Duration: 14 * time.Second},
},
},
// In this round, the peer with id 2 drops off. The query
// results from last discovery lookup are reused.
{
peers: []*Peer{
{rw: &conn{flags: staticDialedConn, id: uintID(0)}},
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(3)}},
{rw: &conn{flags: dynDialedConn, id: uintID(4)}},
{rw: &conn{flags: dynDialedConn, id: uintID(5)}},
},
new: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(6)}},
},
},
// More peers (3,4) drop off and dial for ID 6 completes.
// The last query result from the discovery lookup is reused
// and a new one is spawned because more candidates are needed.
{
peers: []*Peer{
{rw: &conn{flags: staticDialedConn, id: uintID(0)}},
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(5)}},
},
done: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(6)}},
},
new: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(7)}},
&discoverTask{},
},
```

```
},
// Peer 7 is connected, but there still aren't enough dynamic peers
// (4 out of 5). However, a discovery is already running, so ensure
// no new is started.
{
peers: []*Peer{
{rw: &conn{flags: staticDialedConn, id: uintID(0)}},
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(5)}},
{rw: &conn{flags: dynDialedConn, id: uintID(7)}},
},
done: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(7)}},
},
},
// Finish the running node discovery with an empty set. A new lookup
// should be immediately requested.
{
peers: []*Peer{
{rw: &conn{flags: staticDialedConn, id: uintID(0)}},
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(5)}},
{rw: &conn{flags: dynDialedConn, id: uintID(7)}},
},
done: []task{
&discoverTask{},
},
new: []task{
&discoverTask{},
},
},
},
})
}

// Tests that bootnodes are dialed if no peers are connectd, but not otherwise.
func TestDialStateDynDialBootnode(t *testing.T) {
bootnodes := []*discover.Node{
{ID: uintID(1)},
{ID: uintID(2)},
{ID: uintID(3)},
}
```

```
table := fakeTable{
{ID: uintID(4)},
{ID: uintID(5)},
{ID: uintID(6)},
{ID: uintID(7)},
{ID: uintID(8)},
}
runDialTest(t, dialtest{
init: newDialState(nil, bootnodes, table, 5, nil),
rounds: []round{
// 2 dynamic dials attempted, bootnodes pending fallback interval
{
new: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(4)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(5)}},
&discoverTask{},
},
},
// No dials succeed, bootnodes still pending fallback interval
{
done: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(4)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(5)}},
},
},
// No dials succeed, bootnodes still pending fallback interval
{},
// No dials succeed, 2 dynamic dials attempted and 1 bootnode too as fallback interval was
reached
{
new: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(1)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(4)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(5)}},
},
},
// No dials succeed, 2nd bootnode is attempted
{
done: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(1)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(4)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(5)}},
```

```
},
new: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(2)}},
},
},
// No dials succeed, 3rd bootnode is attempted
{
done: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(2)}},
},
new: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(3)}},
},
},
// No dials succeed, 1st bootnode is attempted again, expired random nodes retried
{
done: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(3)}},
},
new: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(1)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(4)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(5)}},
},
},
// Random dial succeeds, no more bootnodes are attempted
{
peers: []*Peer{
{rw: &conn{flags: dynDialedConn, id: uintID(4)}},
},
done: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(1)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(4)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(5)}},
},
},
},
})
}

func TestDialStateDynDialFromTable(t *testing.T) {
// This table always returns the same random nodes
```

```
// in the order given below.
table := fakeTable{
{ID: uintID(1)},
{ID: uintID(2)},
{ID: uintID(3)},
{ID: uintID(4)},
{ID: uintID(5)},
{ID: uintID(6)},
{ID: uintID(7)},
{ID: uintID(8)},
}

runDialTest(t, dialtest{
init: newDialState(nil, nil, table, 10, nil),
rounds: []round{
// 5 out of 8 of the nodes returned by ReadRandomNodes are dialed.
{
new: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(1)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(2)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(3)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(4)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(5)}},
&discoverTask{},
},
},
// Dialing nodes 1,2 succeeds. Dials from the lookup are launched.
{
peers: []*Peer{
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
},
done: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(1)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(2)}},
&discoverTask{results: []*discover.Node{
{ID: uintID(10)},
{ID: uintID(11)},
{ID: uintID(12)},
}},
},
new: []task{
```

```
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(10)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(11)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(12)}},
&discoverTask{},
},
},
// Dialing nodes 3,4,5 fails. The dials from the lookup succeed.
{
peers: []*Peer{
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
{rw: &conn{flags: dynDialedConn, id: uintID(10)}},
{rw: &conn{flags: dynDialedConn, id: uintID(11)}},
{rw: &conn{flags: dynDialedConn, id: uintID(12)}},
},
done: []task{
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(3)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(4)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(5)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(10)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(11)}},
&dialTask{flags: dynDialedConn, dest: &discover.Node{ID: uintID(12)}},
},
},
// Waiting for expiry. No waitExpireTask is launched because the
// discovery query is still running.
{
peers: []*Peer{
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
{rw: &conn{flags: dynDialedConn, id: uintID(10)}},
{rw: &conn{flags: dynDialedConn, id: uintID(11)}},
{rw: &conn{flags: dynDialedConn, id: uintID(12)}},
},
},
// Nodes 3,4 are not tried again because only the first two
// returned random nodes (nodes 1,2) are tried and they're
// already connected.
{
peers: []*Peer{
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
```

```
{rw: &conn{flags: dynDialedConn, id: uintID(10)}},
{rw: &conn{flags: dynDialedConn, id: uintID(11)}},
{rw: &conn{flags: dynDialedConn, id: uintID(12)}},
},
},
},
})
}

// This test checks that candidates that do not match the netrestrict list are not dialed.
func TestDialStateNetRestrict(t *testing.T) {
// This table always returns the same random nodes
// in the order given below.
table := fakeTable{
{ID: uintID(1), IP: net.ParseIP("127.0.0.1")},
{ID: uintID(2), IP: net.ParseIP("127.0.0.2")},
{ID: uintID(3), IP: net.ParseIP("127.0.0.3")},
{ID: uintID(4), IP: net.ParseIP("127.0.0.4")},
{ID: uintID(5), IP: net.ParseIP("127.0.2.5")},
{ID: uintID(6), IP: net.ParseIP("127.0.2.6")},
{ID: uintID(7), IP: net.ParseIP("127.0.2.7")},
{ID: uintID(8), IP: net.ParseIP("127.0.2.8")},
}
restrict := new(netutil.Netlist)
restrict.Add("127.0.2.0/24")

runDialTest(t, dialtest{
init: newDialState(nil, nil, table, 10, restrict),
rounds: []round{
{
new: []task{
&dialTask{flags: dynDialedConn, dest: table[4]},
&discoverTask{},
},
},
},
})
}

// This test checks that static dials are launched.
func TestDialStateStaticDial(t *testing.T) {
wantStatic := []*discover.Node{
```

```
{ID: uintID(1)},
{ID: uintID(2)},
{ID: uintID(3)},
{ID: uintID(4)},
{ID: uintID(5)},
}

runDialTest(t, dialtest{
init: newDialState(wantStatic, nil, fakeTable{}, 0, nil),
rounds: []round{
// Static dials are launched for the nodes that
// aren't yet connected.
{
peers: []*Peer{
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
},
new: []task{
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(3)}},
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(4)}},
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(5)}},
},
},
// No new tasks are launched in this round because all static
// nodes are either connected or still being dialed.
{
peers: []*Peer{
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
{rw: &conn{flags: staticDialedConn, id: uintID(3)}},
},
done: []task{
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(3)}},
},
},
// No new dial tasks are launched because all static
// nodes are now connected.
{
peers: []*Peer{
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
{rw: &conn{flags: staticDialedConn, id: uintID(3)}},
```

```
{rw: &conn{flags: staticDialedConn, id: uintID(4)}},
{rw: &conn{flags: staticDialedConn, id: uintID(5)}},
},
done: []task{
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(4)}},
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(5)}},
},
new: []task{
&waitExpireTask{Duration: 14 * time.Second},
},
},
// Wait a round for dial history to expire, no new tasks should spawn.
{
peers: []*Peer{
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
{rw: &conn{flags: staticDialedConn, id: uintID(3)}},
{rw: &conn{flags: staticDialedConn, id: uintID(4)}},
{rw: &conn{flags: staticDialedConn, id: uintID(5)}},
},
},
// If a static node is dropped, it should be immediately redialed,
// irrespective whether it was originally static or dynamic.
{
peers: []*Peer{
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: staticDialedConn, id: uintID(3)}},
{rw: &conn{flags: staticDialedConn, id: uintID(5)}},
},
new: []task{
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(2)}},
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(4)}},
},
},
},
})
}

// This test checks that past dials are not retried for some time.
func TestDialStateCache(t *testing.T) {
wantStatic := []*discover.Node{
{ID: uintID(1)},
```

```
{ID: uintID(2)},
{ID: uintID(3)},
}

runDialTest(t, dialtest{
init: newDialState(wantStatic, nil, fakeTable{}, 0, nil),
rounds: []round{
// Static dials are launched for the nodes that
// aren't yet connected.
{
peers: nil,
new: []task{
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(1)}},
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(2)}},
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(3)}},
},
},
// No new tasks are launched in this round because all static
// nodes are either connected or still being dialed.
{
peers: []*Peer{
{rw: &conn{flags: staticDialedConn, id: uintID(1)}},
{rw: &conn{flags: staticDialedConn, id: uintID(2)}},
},
done: []task{
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(1)}},
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(2)}},
},
},
// A salvage task is launched to wait for node 3's history
// entry to expire.
{
peers: []*Peer{
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
},
done: []task{
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(3)}},
},
new: []task{
&waitExpireTask{Duration: 14 * time.Second},
},
```

```
},
// Still waiting for node 3's entry to expire in the cache.
{
peers: []*Peer{
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
},
},
// The cache entry for node 3 has expired and is retried.
{
peers: []*Peer{
{rw: &conn{flags: dynDialedConn, id: uintID(1)}},
{rw: &conn{flags: dynDialedConn, id: uintID(2)}},
},
new: []task{
&dialTask{flags: staticDialedConn, dest: &discover.Node{ID: uintID(3)}},
},
},
},
})
}

func TestDialResolve(t *testing.T) {
resolved := discover.NewNode(uintID(1), net.IP{127, 0, 55, 234}, 3333, 4444)
table := &resolveMock{answer: resolved}
state := newDialState(nil, nil, table, 0, nil)

// Check that the task is generated with an incomplete ID.
dest := discover.NewNode(uintID(1), nil, 0, 0)
state.addStatic(dest)
tasks := state.newTasks(0, nil, time.Time{})
if !reflect.DeepEqual(tasks, []task{&dialTask{flags: staticDialedConn, dest: dest}}) {
t.Fatalf("expected dial task, got %#v", tasks)
}

// Now run the task, it should resolve the ID once.
config := Config{Dialer: &net.Dialer{Deadline: time.Now().Add(-5 * time.Minute)}}
srv := &Server{ntab: table, Config: config}
tasks[0].Do(srv)
if !reflect.DeepEqual(table.resolveCalls, []discover.NodeID{dest.ID}) {
t.Fatalf("wrong resolve calls, got %v", table.resolveCalls)
}
```

```
// Report it as done to the dialer, which should update the static node record.
state.taskDone(tasks[0], time.Now())
if state.static[uintID(1)].dest != resolved {
t.Fatalf("state.dest not updated")
}
}

// compares task lists but doesn't care about the order.
func sametasks(a, b []task) bool {
if len(a) != len(b) {
return false
}
next:
for _, ta := range a {
for _, tb := range b {
if reflect.DeepEqual(ta, tb) {
continue next
}
}
return false
}
return true
}

func uintID(i uint32) discover.NodeID {
var id discover.NodeID
binary.BigEndian.PutUint32(id[:], i)
return id
}

// implements discoverTable for TestDialResolve
type resolveMock struct {
resolveCalls []discover.NodeID
answer       *discover.Node
}

func (t *resolveMock) Resolve(id discover.NodeID) *discover.Node {
t.resolveCalls = append(t.resolveCalls, id)
return t.answer
}
```

```
func (t *resolveMock) Self() *discover.Node                    { return new(discover.Node) }
func (t *resolveMock) Close()                                  {}
func (t *resolveMock) Bootstrap([]*discover.Node)              {}
func (t *resolveMock) Lookup(discover.NodeID) []*discover.Node  { return nil }
func (t *resolveMock) ReadRandomNodes(buf []*discover.Node) int { return 0 }

83:F:\git\coin\ethereum\go-ethereum\p2p\discover\database.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains the node database, storing previously seen nodes and any collected
// metadata about them for QoS purposes.

package discover

import (
"bytes"
"crypto/rand"
"encoding/binary"
"os"
"sync"
"time"

"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/rlp"
"github.com/syndtr/goleveldb/leveldb"
"github.com/syndtr/goleveldb/leveldb/errors"
"github.com/syndtr/goleveldb/leveldb/iterator"
"github.com/syndtr/goleveldb/leveldb/opt"
"github.com/syndtr/goleveldb/leveldb/storage"
"github.com/syndtr/goleveldb/leveldb/util"
)

var (
nodeDBNilNodeID      = NodeID{}       // Special node ID to use as a nil element.
nodeDBNodeExpiration = 24 * time.Hour // Time after which an unseen node should be dropped.
nodeDBCleanupCycle   = time.Hour      // Time period for running the expiration task.
)

// nodeDB stores all nodes we know about.
type nodeDB struct {
lvl    *leveldb.DB   // Interface to the database itself
```

```
self   NodeID        // Own node id to prevent adding it into the database
runner sync.Once     // Ensures we can start at most one expirer
quit   chan struct{} // Channel to signal the expiring thread to stop
}

// Schema layout for the node database
var (
nodeDBVersionKey = []byte("version") // Version of the database to flush if changes
nodeDBItemPrefix = []byte("n:")      // Identifier to prefix node entries with

nodeDBDiscoverRoot      = ":discover"
nodeDBDiscoverPing      = nodeDBDiscoverRoot + ":lastping"
nodeDBDiscoverPong      = nodeDBDiscoverRoot + ":lastpong"
nodeDBDiscoverFindFails = nodeDBDiscoverRoot + ":findfail"
)

// newNodeDB creates a new node database for storing and retrieving infos about
// known peers in the network. If no path is given, an in-memory, temporary
// database is constructed.
func newNodeDB(path string, version int, self NodeID) (*nodeDB, error) {
if path == "" {
return newMemoryNodeDB(self)
}
return newPersistentNodeDB(path, version, self)
}

// newMemoryNodeDB creates a new in-memory node database without a persistent
// backend.
func newMemoryNodeDB(self NodeID) (*nodeDB, error) {
db, err := leveldb.Open(storage.NewMemStorage(), nil)
if err != nil {
return nil, err
}
return &nodeDB{
lvl:  db,
self: self,
quit: make(chan struct{}),
}, nil
}

// newPersistentNodeDB creates/opens a leveldb backed persistent node database,
// also flushing its contents in case of a version mismatch.
```

```
func newPersistentNodeDB(path string, version int, self NodeID) (*nodeDB, error) {
opts := &opt.Options{OpenFilesCacheCapacity: 5}
db, err := leveldb.OpenFile(path, opts)
if _, iscorrupted := err.(*errors.ErrCorrupted); iscorrupted {
db, err = leveldb.RecoverFile(path, nil)
}
if err != nil {
return nil, err
}
// The nodes contained in the cache correspond to a certain protocol version.
// Flush all nodes if the version doesn't match.
currentVer := make([]byte, binary.MaxVarintLen64)
currentVer = currentVer[:binary.PutVarint(currentVer, int64(version))]

blob, err := db.Get(nodeDBVersionKey, nil)
switch err {
case leveldb.ErrNotFound:
// Version not found (i.e. empty cache), insert it
if err := db.Put(nodeDBVersionKey, currentVer, nil); err != nil {
db.Close()
return nil, err
}

case nil:
// Version present, flush if different
if !bytes.Equal(blob, currentVer) {
db.Close()
if err = os.RemoveAll(path); err != nil {
return nil, err
}
return newPersistentNodeDB(path, version, self)
}
}
return &nodeDB{
lvl:  db,
self: self,
quit: make(chan struct{}),
}, nil
}

// makeKey generates the leveldb key-blob from a node id and its particular
// field of interest.
```

```
func makeKey(id NodeID, field string) []byte {
if bytes.Equal(id[:], nodeDBNilNodeID[:]) {
return []byte(field)
}
return append(nodeDBItemPrefix, append(id[:], field...)...)
}

// splitKey tries to split a database key into a node id and a field part.
func splitKey(key []byte) (id NodeID, field string) {
// If the key is not of a node, return it plainly
if !bytes.HasPrefix(key, nodeDBItemPrefix) {
return NodeID{}, string(key)
}
// Otherwise split the id and field
item := key[len(nodeDBItemPrefix):]
copy(id[:], item[:len(id)])
field = string(item[len(id):])

return id, field
}

// fetchInt64 retrieves an integer instance associated with a particular
// database key.
func (db *nodeDB) fetchInt64(key []byte) int64 {
blob, err := db.lvl.Get(key, nil)
if err != nil {
return 0
}
val, read := binary.Varint(blob)
if read <= 0 {
return 0
}
return val
}

// storeInt64 update a specific database entry to the current time instance as a
// unix timestamp.
func (db *nodeDB) storeInt64(key []byte, n int64) error {
blob := make([]byte, binary.MaxVarintLen64)
blob = blob[:binary.PutVarint(blob, n)]

return db.lvl.Put(key, blob, nil)
```

```
}

// node retrieves a node with a given id from the database.
func (db *nodeDB) node(id NodeID) *Node {
blob, err := db.lvl.Get(makeKey(id, nodeDBDiscoverRoot), nil)
if err != nil {
return nil
}
node := new(Node)
if err := rlp.DecodeBytes(blob, node); err != nil {
log.Error("Failed to decode node RLP", "err", err)
return nil
}
node.sha = crypto.Keccak256Hash(node.ID[:])
return node
}

// updateNode inserts - potentially overwriting - a node into the peer database.
func (db *nodeDB) updateNode(node *Node) error {
blob, err := rlp.EncodeToBytes(node)
if err != nil {
return err
}
return db.lvl.Put(makeKey(node.ID, nodeDBDiscoverRoot), blob, nil)
}

// deleteNode deletes all information/keys associated with a node.
func (db *nodeDB) deleteNode(id NodeID) error {
deleter := db.lvl.NewIterator(util.BytesPrefix(makeKey(id, "")), nil)
for deleter.Next() {
if err := db.lvl.Delete(deleter.Key(), nil); err != nil {
return err
}
}
return nil
}

// ensureExpirer is a small helper method ensuring that the data expiration
// mechanism is running. If the expiration goroutine is already running, this
// method simply returns.
//
// The goal is to start the data evacuation only after the network successfully
```

```
// bootstrapped itself (to prevent dumping potentially useful seed nodes). Since
// it would require significant overhead to exactly trace the first successful
// convergence, it's simpler to "ensure" the correct state when an appropriate
// condition occurs (i.e. a successful bonding), and discard further events.
func (db *nodeDB) ensureExpirer() {
db.runner.Do(func() { go db.expirer() })
}

// expirer should be started in a go routine, and is responsible for looping ad
// infinitum and dropping stale data from the database.
func (db *nodeDB) expirer() {
tick := time.Tick(nodeDBCleanupCycle)
for {
select {
case <-tick:
if err := db.expireNodes(); err != nil {
log.Error("Failed to expire nodedb items", "err", err)
}

case <-db.quit:
return
}
}
}

// expireNodes iterates over the database and deletes all nodes that have not
// been seen (i.e. received a pong from) for some allotted time.
func (db *nodeDB) expireNodes() error {
threshold := time.Now().Add(-nodeDBNodeExpiration)

// Find discovered nodes that are older than the allowance
it := db.lvl.NewIterator(nil, nil)
defer it.Release()

for it.Next() {
// Skip the item if not a discovery node
id, field := splitKey(it.Key())
if field != nodeDBDiscoverRoot {
continue
}
// Skip the node if not expired yet (and not self)
if !bytes.Equal(id[:], db.self[:]) {
```

```
if seen := db.lastPong(id); seen.After(threshold) {
continue
}
}
// Otherwise delete all associated information
db.deleteNode(id)
}
return nil
}

// lastPing retrieves the time of the last ping packet send to a remote node,
// requesting binding.
func (db *nodeDB) lastPing(id NodeID) time.Time {
return time.Unix(db.fetchInt64(makeKey(id, nodeDBDiscoverPing)), 0)
}

// updateLastPing updates the last time we tried contacting a remote node.
func (db *nodeDB) updateLastPing(id NodeID, instance time.Time) error {
return db.storeInt64(makeKey(id, nodeDBDiscoverPing), instance.Unix())
}

// lastPong retrieves the time of the last successful contact from remote node.
func (db *nodeDB) lastPong(id NodeID) time.Time {
return time.Unix(db.fetchInt64(makeKey(id, nodeDBDiscoverPong)), 0)
}

// updateLastPong updates the last time a remote node successfully contacted.
func (db *nodeDB) updateLastPong(id NodeID, instance time.Time) error {
return db.storeInt64(makeKey(id, nodeDBDiscoverPong), instance.Unix())
}

// findFails retrieves the number of findnode failures since bonding.
func (db *nodeDB) findFails(id NodeID) int {
return int(db.fetchInt64(makeKey(id, nodeDBDiscoverFindFails)))
}

// updateFindFails updates the number of findnode failures since bonding.
func (db *nodeDB) updateFindFails(id NodeID, fails int) error {
return db.storeInt64(makeKey(id, nodeDBDiscoverFindFails), int64(fails))
}

// querySeeds retrieves random nodes to be used as potential seed nodes
```

```
// for bootstrapping.
func (db *nodeDB) querySeeds(n int, maxAge time.Duration) []*Node {
var (
now   = time.Now()
nodes = make([]*Node, 0, n)
it    = db.lvl.NewIterator(nil, nil)
id    NodeID
)
defer it.Release()

seek:
for seeks := 0; len(nodes) < n && seeks < n*5; seeks++ {
// Seek to a random entry. The first byte is incremented by a
// random amount each time in order to increase the likelihood
// of hitting all existing nodes in very small databases.
ctr := id[0]
rand.Read(id[:])
id[0] = ctr + id[0]%16
it.Seek(makeKey(id, nodeDBDiscoverRoot))

n := nextNode(it)
if n == nil {
id[0] = 0
continue seek // iterator exhausted
}
if n.ID == db.self {
continue seek
}
if now.Sub(db.lastPong(n.ID)) > maxAge {
continue seek
}
for i := range nodes {
if nodes[i].ID == n.ID {
continue seek // duplicate
}
}
nodes = append(nodes, n)
}
return nodes
}

// reads the next node record from the iterator, skipping over other
```

```
// database entries.
func nextNode(it iterator.Iterator) *Node {
for end := false; !end; end = !it.Next() {
id, field := splitKey(it.Key())
if field != nodeDBDiscoverRoot {
continue
}
var n Node
if err := rlp.DecodeBytes(it.Value(), &n); err != nil {
log.Warn("Failed to decode node RLP", "id", id, "err", err)
continue
}
return &n
}
return nil
}

// close flushes and closes the database files.
func (db *nodeDB) close() {
close(db.quit)
db.lvl.Close()
}

84:F:\git\coin\ethereum\go-ethereum\p2p\discover\database_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discover

import (
"bytes"
"io/ioutil"
"net"
"os"
"path/filepath"
"reflect"
"testing"
"time"
)

var nodeDBKeyTests = []struct {
id    NodeID
field string
```

```
key   []byte
}{
{
id:    NodeID{},
field: "version",
key:   []byte{0x76, 0x65, 0x72, 0x73, 0x69, 0x6f, 0x6e}, // field
},
{
id:
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
field: ":discover",
key: []byte{0x6e, 0x3a, // prefix
0x1d, 0xd9, 0xd6, 0x5c, 0x45, 0x52, 0xb5, 0xeb, // node id
0x43, 0xd5, 0xad, 0x55, 0xa2, 0xee, 0x3f, 0x56, //
0xc6, 0xcb, 0xc1, 0xc6, 0x4a, 0x5c, 0x8d, 0x65, //
0x9f, 0x51, 0xfc, 0xd5, 0x1b, 0xac, 0xe2, 0x43, //
0x51, 0x23, 0x2b, 0x8d, 0x78, 0x21, 0x61, 0x7d, //
0x2b, 0x29, 0xb5, 0x4b, 0x81, 0xcd, 0xef, 0xb9, //
0xb3, 0xe9, 0xc3, 0x7d, 0x7f, 0xd5, 0xf6, 0x32, //
0x70, 0xbc, 0xc9, 0xe1, 0xa6, 0xf6, 0xa4, 0x39, //
0x3a, 0x64, 0x69, 0x73, 0x63, 0x6f, 0x76, 0x65, 0x72, // field
},
},
}

func TestNodeDBKeys(t *testing.T) {
for i, tt := range nodeDBKeyTests {
if key := makeKey(tt.id, tt.field); !bytes.Equal(key, tt.key) {
t.Errorf("make test %d: key mismatch: have 0x%x, want 0x%x", i, key, tt.key)
}
id, field := splitKey(tt.key)
if !bytes.Equal(id[:], tt.id[:]) {
t.Errorf("split test %d: id mismatch: have 0x%x, want 0x%x", i, id, tt.id)
}
if field != tt.field {
t.Errorf("split test %d: field mismatch: have 0x%x, want 0x%x", i, field, tt.field)
}
}
}

var nodeDBInt64Tests = []struct {
```

```
key   []byte
value int64
}{
{key: []byte{0x01}, value: 1},
{key: []byte{0x02}, value: 2},
{key: []byte{0x03}, value: 3},
}

func TestNodeDBInt64(t *testing.T) {
db, _ := newNodeDB("", Version, NodeID{})
defer db.close()

tests := nodeDBInt64Tests
for i := 0; i < len(tests); i++ {
// Insert the next value
if err := db.storeInt64(tests[i].key, tests[i].value); err != nil {
t.Errorf("test %d: failed to store value: %v", i, err)
}
// Check all existing and non existing values
for j := 0; j < len(tests); j++ {
num := db.fetchInt64(tests[j].key)
switch {
case j <= i && num != tests[j].value:
t.Errorf("test %d, item %d: value mismatch: have %v, want %v", i, j, num, tests[j].value)
case j > i && num != 0:
t.Errorf("test %d, item %d: value mismatch: have %v, want %v", i, j, num, 0)
}
}
}
}

func TestNodeDBFetchStore(t *testing.T) {
node := NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{192, 168, 0, 1},
30303,
30303,
)
inst := time.Now()
num := 314
```

```
db, _ := newNodeDB("", Version, NodeID{})
defer db.close()

// Check fetch/store operations on a node ping object
if stored := db.lastPing(node.ID); stored.Unix() != 0 {
t.Errorf("ping: non-existing object: %v", stored)
}
if err := db.updateLastPing(node.ID, inst); err != nil {
t.Errorf("ping: failed to update: %v", err)
}
if stored := db.lastPing(node.ID); stored.Unix() != inst.Unix() {
t.Errorf("ping: value mismatch: have %v, want %v", stored, inst)
}
// Check fetch/store operations on a node pong object
if stored := db.lastPong(node.ID); stored.Unix() != 0 {
t.Errorf("pong: non-existing object: %v", stored)
}
if err := db.updateLastPong(node.ID, inst); err != nil {
t.Errorf("pong: failed to update: %v", err)
}
if stored := db.lastPong(node.ID); stored.Unix() != inst.Unix() {
t.Errorf("pong: value mismatch: have %v, want %v", stored, inst)
}
// Check fetch/store operations on a node findnode-failure object
if stored := db.findFails(node.ID); stored != 0 {
t.Errorf("find-node fails: non-existing object: %v", stored)
}
if err := db.updateFindFails(node.ID, num); err != nil {
t.Errorf("find-node fails: failed to update: %v", err)
}
if stored := db.findFails(node.ID); stored != num {
t.Errorf("find-node fails: value mismatch: have %v, want %v", stored, num)
}
// Check fetch/store operations on an actual node object
if stored := db.node(node.ID); stored != nil {
t.Errorf("node: non-existing object: %v", stored)
}
if err := db.updateNode(node); err != nil {
t.Errorf("node: failed to update: %v", err)
}
if stored := db.node(node.ID); stored == nil {
t.Errorf("node: not found")
```

```
} else if !reflect.DeepEqual(stored, node) {
t.Errorf("node: data mismatch: have %v, want %v", stored, node)
}
}

var nodeDBSeedQueryNodes = []struct {
node *Node
pong time.Time
}{
// This one should not be in the result set because its last
// pong time is too far in the past.
{
node: NewNode(
MustHexID("0x84d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{127, 0, 0, 3},
30303,
30303,
),
pong: time.Now().Add(-3 * time.Hour),
},
// This one shouldn't be in in the result set because its
// nodeID is the local node's ID.
{
node: NewNode(
MustHexID("0x57d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{127, 0, 0, 3},
30303,
30303,
),
pong: time.Now().Add(-4 * time.Second),
},

// These should be in the result set.
{
node: NewNode(
MustHexID("0x22d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{127, 0, 0, 1},
30303,
30303,
```

```
),
pong: time.Now().Add(-2 * time.Second),
},
{
node: NewNode(
MustHexID("0x44d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{127, 0, 0, 2},
30303,
30303,
),
pong: time.Now().Add(-3 * time.Second),
},
{
node: NewNode(
MustHexID("0xe2d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{127, 0, 0, 3},
30303,
30303,
),
pong: time.Now().Add(-1 * time.Second),
},
}

func TestNodeDBSeedQuery(t *testing.T) {
db, _ := newNodeDB("", Version, nodeDBSeedQueryNodes[1].node.ID)
defer db.close()

// Insert a batch of nodes for querying
for i, seed := range nodeDBSeedQueryNodes {
if err := db.updateNode(seed.node); err != nil {
t.Fatalf("node %d: failed to insert: %v", i, err)
}
if err := db.updateLastPong(seed.node.ID, seed.pong); err != nil {
t.Fatalf("node %d: failed to insert lastPong: %v", i, err)
}
}

// Retrieve the entire batch and check for duplicates
seeds := db.querySeeds(len(nodeDBSeedQueryNodes)*2, time.Hour)
have := make(map[NodeID]struct{})
```

```
for _, seed := range seeds {
have[seed.ID] = struct{}{}
}
want := make(map[NodeID]struct{})
for _, seed := range nodeDBSeedQueryNodes[2:] {
want[seed.node.ID] = struct{}{}
}
if len(seeds) != len(want) {
t.Errorf("seed count mismatch: have %v, want %v", len(seeds), len(want))
}
for id := range have {
if _, ok := want[id]; !ok {
t.Errorf("extra seed: %v", id)
}
}
for id := range want {
if _, ok := have[id]; !ok {
t.Errorf("missing seed: %v", id)
}
}
}

func TestNodeDBPersistency(t *testing.T) {
root, err := ioutil.TempDir("", "nodedb-")
if err != nil {
t.Fatalf("failed to create temporary data folder: %v", err)
}
defer os.RemoveAll(root)

var (
testKey = []byte("somekey")
testInt = int64(314)
)

// Create a persistent database and store some values
db, err := newNodeDB(filepath.Join(root, "database"), Version, NodeID{})
if err != nil {
t.Fatalf("failed to create persistent database: %v", err)
}
if err := db.storeInt64(testKey, testInt); err != nil {
t.Fatalf("failed to store value: %v.", err)
}
```

```
db.close()

// Reopen the database and check the value
db, err = newNodeDB(filepath.Join(root, "database"), Version, NodeID{})
if err != nil {
t.Fatalf("failed to open persistent database: %v", err)
}
if val := db.fetchInt64(testKey); val != testInt {
t.Fatalf("value mismatch: have %v, want %v", val, testInt)
}
db.close()

// Change the database version and check flush
db, err = newNodeDB(filepath.Join(root, "database"), Version+1, NodeID{})
if err != nil {
t.Fatalf("failed to open persistent database: %v", err)
}
if val := db.fetchInt64(testKey); val != 0 {
t.Fatalf("value mismatch: have %v, want %v", val, 0)
}
db.close()
}

var nodeDBExpirationNodes = []struct {
node *Node
pong time.Time
exp  bool
}{
{
node: NewNode(
MustHexID("0x01d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{127, 0, 0, 1},
30303,
30303,
),
pong: time.Now().Add(-nodeDBNodeExpiration + time.Minute),
exp:  false,
}, {
node: NewNode(
MustHexID("0x02d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
```

```
net.IP{127, 0, 0, 2},
30303,
30303,
),
pong: time.Now().Add(-nodeDBNodeExpiration - time.Minute),
exp:  true,
},
}

func TestNodeDBExpiration(t *testing.T) {
db, _ := newNodeDB("", Version, NodeID{})
defer db.close()

// Add all the test nodes and set their last pong time
for i, seed := range nodeDBExpirationNodes {
if err := db.updateNode(seed.node); err != nil {
t.Fatalf("node %d: failed to insert: %v", i, err)
}
if err := db.updateLastPong(seed.node.ID, seed.pong); err != nil {
t.Fatalf("node %d: failed to update pong: %v", i, err)
}
}
// Expire some of them, and check the rest
if err := db.expireNodes(); err != nil {
t.Fatalf("failed to expire nodes: %v", err)
}
for i, seed := range nodeDBExpirationNodes {
node := db.node(seed.node.ID)
if (node == nil && !seed.exp) || (node != nil && seed.exp) {
t.Errorf("node %d: expiration mismatch: have %v, want %v", i, node, seed.exp)
}
}
}

func TestNodeDBSelfExpiration(t *testing.T) {
// Find a node in the tests that shouldn't expire, and assign it as self
var self NodeID
for _, node := range nodeDBExpirationNodes {
if !node.exp {
self = node.node.ID
break
}
```

```
}
db, _ := newNodeDB("", Version, self)
defer db.close()

// Add all the test nodes and set their last pong time
for i, seed := range nodeDBExpirationNodes {
if err := db.updateNode(seed.node); err != nil {
t.Fatalf("node %d: failed to insert: %v", i, err)
}
if err := db.updateLastPong(seed.node.ID, seed.pong); err != nil {
t.Fatalf("node %d: failed to update pong: %v", i, err)
}
}
// Expire the nodes and make sure self has been evacuated too
if err := db.expireNodes(); err != nil {
t.Fatalf("failed to expire nodes: %v", err)
}
node := db.node(self)
if node != nil {
t.Errorf("self not evacuated")
}
}
```

85:F:\git\coin\ethereum\go-ethereum\p2p\discover\node.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discover

import (
"crypto/ecdsa"
"crypto/elliptic"
"encoding/hex"
"errors"
"fmt"
"math/big"
"math/rand"
"net"
"net/url"
"regexp"
"strconv"
"strings"
```

```
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/crypto/secp256k1"
)

const NodeIDBits = 512

// Node represents a host on the network.
// The fields of Node may not be modified.
type Node struct {
IP       net.IP // len 4 for IPv4 or 16 for IPv6
UDP, TCP uint16 // port numbers
ID       NodeID // the node's public key

// This is a cached copy of sha3(ID) which is used for node
// distance calculations. This is part of Node in order to make it
// possible to write tests that need a node at a certain distance.
// In those tests, the content of sha will not actually correspond
// with ID.
sha common.Hash

// whether this node is currently being pinged in order to replace
// it in a bucket
contested bool
}

// NewNode creates a new node. It is mostly meant to be used for
// testing purposes.
func NewNode(id NodeID, ip net.IP, udpPort, tcpPort uint16) *Node {
if ipv4 := ip.To4(); ipv4 != nil {
ip = ipv4
}
return &Node{
IP:  ip,
UDP: udpPort,
TCP: tcpPort,
ID:  id,
sha: crypto.Keccak256Hash(id[:]),
}
}

func (n *Node) addr() *net.UDPAddr {
```

```
return &net.UDPAddr{IP: n.IP, Port: int(n.UDP)}
}

// Incomplete returns true for nodes with no IP address.
func (n *Node) Incomplete() bool {
return n.IP == nil
}

// checks whether n is a valid complete node.
func (n *Node) validateComplete() error {
if n.Incomplete() {
return errors.New("incomplete node")
}
if n.UDP == 0 {
return errors.New("missing UDP port")
}
if n.TCP == 0 {
return errors.New("missing TCP port")
}
if n.IP.IsMulticast() || n.IP.IsUnspecified() {
return errors.New("invalid IP (multicast/unspecified)")
}
_, err := n.ID.Pubkey() // validate the key (on curve, etc.)
return err
}

// The string representation of a Node is a URL.
// Please see ParseNode for a description of the format.
func (n *Node) String() string {
u := url.URL{Scheme: "enode"}
if n.Incomplete() {
u.Host = fmt.Sprintf("%x", n.ID[:])
} else {
addr := net.TCPAddr{IP: n.IP, Port: int(n.TCP)}
u.User = url.User(fmt.Sprintf("%x", n.ID[:]))
u.Host = addr.String()
if n.UDP != n.TCP {
u.RawQuery = "discport=" + strconv.Itoa(int(n.UDP))
}
}
return u.String()
}
```

```
var incompleteNodeURL = regexp.MustCompile("(?i)^(?:enode://)?([0-9a-f]+)$")

// ParseNode parses a node designator.
//
// There are two basic forms of node designators
//   - incomplete nodes, which only have the public key (node ID)
//   - complete nodes, which contain the public key and IP/Port information
//
// For incomplete nodes, the designator must look like one of these
//
//    enode://<hex node id>
//    <hex node id>
//
// For complete nodes, the node ID is encoded in the username portion
// of the URL, separated from the host by an @ sign. The hostname can
// only be given as an IP address, DNS domain names are not allowed.
// The port in the host name section is the TCP listening port. If the
// TCP and UDP (discovery) ports differ, the UDP port is specified as
// query parameter "discport".
//
// In the following example, the node URL describes
// a node with IP address 10.3.58.6, TCP listening port 30303
// and UDP discovery port 30301.
//
//    enode://<hex node id>@10.3.58.6:30303?discport=30301
func ParseNode(rawurl string) (*Node, error) {
if m := incompleteNodeURL.FindStringSubmatch(rawurl); m != nil {
id, err := HexID(m[1])
if err != nil {
return nil, fmt.Errorf("invalid node ID (%v)", err)
}
return NewNode(id, nil, 0, 0), nil
}
return parseComplete(rawurl)
}

func parseComplete(rawurl string) (*Node, error) {
var (
id              NodeID
ip              net.IP
tcpPort, udpPort uint64
```

```
)
u, err := url.Parse(rawurl)
if err != nil {
return nil, err
}
if u.Scheme != "enode" {
return nil, errors.New("invalid URL scheme, want \"enode\"")
}
// Parse the Node ID from the user portion.
if u.User == nil {
return nil, errors.New("does not contain node ID")
}
if id, err = HexID(u.User.String()); err != nil {
return nil, fmt.Errorf("invalid node ID (%v)", err)
}
// Parse the IP address.
host, port, err := net.SplitHostPort(u.Host)
if err != nil {
return nil, fmt.Errorf("invalid host: %v", err)
}
if ip = net.ParseIP(host); ip == nil {
return nil, errors.New("invalid IP address")
}
// Ensure the IP is 4 bytes long for IPv4 addresses.
if ipv4 := ip.To4(); ipv4 != nil {
ip = ipv4
}
// Parse the port numbers.
if tcpPort, err = strconv.ParseUint(port, 10, 16); err != nil {
return nil, errors.New("invalid port")
}
udpPort = tcpPort
qv := u.Query()
if qv.Get("discport") != "" {
udpPort, err = strconv.ParseUint(qv.Get("discport"), 10, 16)
if err != nil {
return nil, errors.New("invalid discport in query")
}
}
return NewNode(id, ip, uint16(udpPort), uint16(tcpPort)), nil
}
```

```
// MustParseNode parses a node URL. It panics if the URL is not valid.
func MustParseNode(rawurl string) *Node {
n, err := ParseNode(rawurl)
if err != nil {
panic("invalid node URL: " + err.Error())
}
return n
}

// MarshalText implements encoding.TextMarshaler.
func (n *Node) MarshalText() ([]byte, error) {
return []byte(n.String()), nil
}

// UnmarshalText implements encoding.TextUnmarshaler.
func (n *Node) UnmarshalText(text []byte) error {
dec, err := ParseNode(string(text))
if err == nil {
*n = *dec
}
return err
}

// NodeID is a unique identifier for each node.
// The node identifier is a marshaled elliptic curve public key.
type NodeID [NodeIDBits / 8]byte

// NodeID prints as a long hexadecimal number.
func (n NodeID) String() string {
return fmt.Sprintf("%x", n[:])
}

// The Go syntax representation of a NodeID is a call to HexID.
func (n NodeID) GoString() string {
return fmt.Sprintf("discover.HexID(\"%x\")", n[:])
}

// TerminalString returns a shortened hex string for terminal logging.
func (n NodeID) TerminalString() string {
return hex.EncodeToString(n[:8])
}
```

```
// HexID converts a hex string to a NodeID.
// The string may be prefixed with 0x.
func HexID(in string) (NodeID, error) {
var id NodeID
b, err := hex.DecodeString(strings.TrimPrefix(in, "0x"))
if err != nil {
return id, err
} else if len(b) != len(id) {
return id, fmt.Errorf("wrong length, want %d hex chars", len(id)*2)
}
copy(id[:], b)
return id, nil
}

// MustHexID converts a hex string to a NodeID.
// It panics if the string is not a valid NodeID.
func MustHexID(in string) NodeID {
id, err := HexID(in)
if err != nil {
panic(err)
}
return id
}

// PubkeyID returns a marshaled representation of the given public key.
func PubkeyID(pub *ecdsa.PublicKey) NodeID {
var id NodeID
pbytes := elliptic.Marshal(pub.Curve, pub.X, pub.Y)
if len(pbytes)-1 != len(id) {
panic(fmt.Errorf("need %d bit pubkey, got %d bits", (len(id)+1)*8, len(pbytes)))
}
copy(id[:], pbytes[1:])
return id
}

// Pubkey returns the public key represented by the node ID.
// It returns an error if the ID is not a point on the curve.
func (id NodeID) Pubkey() (*ecdsa.PublicKey, error) {
p := &ecdsa.PublicKey{Curve: crypto.S256(), X: new(big.Int), Y: new(big.Int)}
half := len(id) / 2
p.X.SetBytes(id[:half])
p.Y.SetBytes(id[half:])
```

```
if !p.Curve.IsOnCurve(p.X, p.Y) {
return nil, errors.New("id is invalid secp256k1 curve point")
}
return p, nil
}

// recoverNodeID computes the public key used to sign the
// given hash from the signature.
func recoverNodeID(hash, sig []byte) (id NodeID, err error) {
pubkey, err := secp256k1.RecoverPubkey(hash, sig)
if err != nil {
return id, err
}
if len(pubkey)-1 != len(id) {
return id, fmt.Errorf("recovered pubkey has %d bits, want %d bits", len(pubkey)*8, (len(id)+1)*8)
}
for i := range id {
id[i] = pubkey[i+1]
}
return id, nil
}

// distcmp compares the distances a->target and b->target.
// Returns -1 if a is closer to target, 1 if b is closer to target
// and 0 if they are equal.
func distcmp(target, a, b common.Hash) int {
for i := range target {
da := a[i] ^ target[i]
db := b[i] ^ target[i]
if da > db {
return 1
} else if da < db {
return -1
}
}
return 0
}

// table of leading zero counts for bytes [0..255]
var lzcount = [256]int{
8, 7, 6, 6, 5, 5, 5, 5,
4, 4, 4, 4, 4, 4, 4, 4,
```

```
3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3,
2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
}

// logdist returns the logarithmic distance between a and b, log2(a ^ b).
func logdist(a, b common.Hash) int {
lz := 0
for i := range a {
x := a[i] ^ b[i]
if x == 0 {
lz += 8
} else {
lz += lzcount[x]
break
```

```
}
}
return len(a)*8 - lz
}

// hashAtDistance returns a random hash such that logdist(a, b) == n
func hashAtDistance(a common.Hash, n int) (b common.Hash) {
if n == 0 {
return a
}
// flip bit at position n, fill the rest with random bits
b = a
pos := len(a) - n/8 - 1
bit := byte(0x01) << (byte(n%8) - 1)
if bit == 0 {
pos++
bit = 0x80
}
b[pos] = a[pos]&^bit | ^a[pos]&bit // TODO: randomize end bits
for i := pos + 1; i < len(a); i++ {
b[i] = byte(rand.Intn(255))
}
return b
}

86:F:\git\coin\ethereum\go-ethereum\p2p\discover\node_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discover

import (
"fmt"
"math/big"
"math/rand"
"net"
"reflect"
"strings"
"testing"
"testing/quick"
"time"

"github.com/ethereum/go-ethereum/common"
```

```
"github.com/ethereum/go-ethereum/crypto"
)

func ExampleNewNode() {
id :=
MustHexID("1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b
8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439")

// Complete nodes contain UDP and TCP endpoints:
n1 := NewNode(id, net.ParseIP("2001:db8:3c4d:15::abcd:ef12"), 52150, 30303)
fmt.Println("n1:", n1)
fmt.Println("n1.Incomplete() ->", n1.Incomplete())

// An incomplete node can be created by passing zero values
// for all parameters except id.
n2 := NewNode(id, nil, 0, 0)
fmt.Println("n2:", n2)
fmt.Println("n2.Incomplete() ->", n2.Incomplete())

// Output:
// n1:
enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d78
21617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@[2001:db8:3c4d:15::abcd:ef12]:
30303?discport=52150
// n1.Incomplete() -> false
// n2:
enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d78
21617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439
// n2.Incomplete() -> true
}

var parseNodeTests = []struct {
rawurl     string
wantError  string
wantResult *Node
}{
{
rawurl:    "http://foobar",
wantError: `invalid URL scheme, want "enode"`,
},
{
rawurl:    "enode://01010101@123.124.125.126:3",
```

```
wantError: `invalid node ID (wrong length, want 128 hex chars)`,
},
// Complete nodes with IP address.
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@hostname:3",
wantError: `invalid IP address`,
},
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@127.0.0.1:foo",
wantError: `invalid port`,
},
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@127.0.0.1:3?discport=foo",
wantError: `invalid discport in query`,
},
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@127.0.0.1:52150",
wantResult: NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{0x7f, 0x0, 0x0, 0x1},
52150,
52150,
),
},
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@[::]:52150",
wantResult: NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.ParseIP("::"),
52150,
```

```
52150,
),
},
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@[2001:db8:3c4d:15::abcd:ef12
]:52150",
wantResult: NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.ParseIP("2001:db8:3c4d:15::abcd:ef12"),
52150,
52150,
),
},
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@127.0.0.1:52150?discport=22
334",
wantResult: NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{0x7f, 0x0, 0x0, 0x1},
22334,
52150,
),
},
// Incomplete nodes with no address.
{
rawurl:
"1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d
2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439",
wantResult: NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
nil, 0, 0,
),
},
{
rawurl:
```

```
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439",
wantResult: NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
nil, 0, 0,
),
},
// Invalid URLs
{
rawurl:    "01010101",
wantError: `invalid node ID (wrong length, want 128 hex chars)`,
},
{
rawurl:    "enode://01010101",
wantError: `invalid node ID (wrong length, want 128 hex chars)`,
},
{
// This test checks that errors from url.Parse are handled.
rawurl:    "://foo",
wantError: `parse ://foo: missing protocol scheme`,
},
}

func TestParseNode(t *testing.T) {
for _, test := range parseNodeTests {
n, err := ParseNode(test.rawurl)
if test.wantError != "" {
if err == nil {
t.Errorf("test %q:\n  got nil error, expected %#q", test.rawurl, test.wantError)
continue
} else if err.Error() != test.wantError {
t.Errorf("test %q:\n  got error %#q, expected %#q", test.rawurl, err.Error(), test.wantError)
continue
}
} else {
if err != nil {
t.Errorf("test %q:\n  unexpected error: %v", test.rawurl, err)
continue
}
if !reflect.DeepEqual(n, test.wantResult) {
t.Errorf("test %q:\n  result mismatch:\ngot:  %#v, want: %#v", test.rawurl, n, test.wantResult)
```

```
}
}
}
}

func TestNodeString(t *testing.T) {
for i, test := range parseNodeTests {
if test.wantError == "" && strings.HasPrefix(test.rawurl, "enode://") {
str := test.wantResult.String()
if str != test.rawurl {
t.Errorf("test %d: Node.String() mismatch:\ngot:  %s\nwant: %s", i, str, test.rawurl)
}
}
}
}

func TestHexID(t *testing.T) {
ref := NodeID{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 128, 106, 217, 182, 31, 165, 174, 1, 67, 7, 235, 220, 150, 66, 83, 173, 205, 159,
44, 10, 57, 42, 161, 26, 188}
id1 :=
MustHexID("0x0000000000000000000000000000000000000000000000000000000000000000806ad9b61fa5ae014307ebdc964253adcd9f2c0a392aa11abc")
id2 :=
MustHexID("0000000000000000000000000000000000000000000000000000000000000000806ad9b61fa5ae014307ebdc964253adcd9f2c0a392aa11abc")

if id1 != ref {
t.Errorf("wrong id1\ngot  %v\nwant %v", id1[:], ref[:])
}
if id2 != ref {
t.Errorf("wrong id2\ngot  %v\nwant %v", id2[:], ref[:])
}
}

func TestNodeID_recover(t *testing.T) {
prv := newkey()
hash := make([]byte, 32)
sig, err := crypto.Sign(hash, prv)
if err != nil {
t.Fatalf("signing error: %v", err)
}
```

```
pub := PubkeyID(&prv.PublicKey)
recpub, err := recoverNodeID(hash, sig)
if err != nil {
t.Fatalf("recovery error: %v", err)
}
if pub != recpub {
t.Errorf("recovered wrong pubkey:\ngot:  %v\nwant: %v", recpub, pub)
}

ecdsa, err := pub.Pubkey()
if err != nil {
t.Errorf("Pubkey error: %v", err)
}
if !reflect.DeepEqual(ecdsa, &prv.PublicKey) {
t.Errorf("Pubkey mismatch:\n  got:  %#v\n  want: %#v", ecdsa, &prv.PublicKey)
}
}

func TestNodeID_pubkeyBad(t *testing.T) {
ecdsa, err := NodeID{}.Pubkey()
if err == nil {
t.Error("expected error for zero ID")
}
if ecdsa != nil {
t.Error("expected nil result")
}
}

func TestNodeID_distcmp(t *testing.T) {
distcmpBig := func(target, a, b common.Hash) int {
tbig := new(big.Int).SetBytes(target[:])
abig := new(big.Int).SetBytes(a[:])
bbig := new(big.Int).SetBytes(b[:])
return new(big.Int).Xor(tbig, abig).Cmp(new(big.Int).Xor(tbig, bbig))
}
if err := quick.CheckEqual(distcmp, distcmpBig, quickcfg()); err != nil {
t.Error(err)
}
}

// the random tests is likely to miss the case where they're equal.
```

```
func TestNodeID_distcmpEqual(t *testing.T) {
base := common.Hash{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
x := common.Hash{15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}
if distcmp(base, x, x) != 0 {
t.Errorf("distcmp(base, x, x) != 0")
}
}

func TestNodeID_logdist(t *testing.T) {
logdistBig := func(a, b common.Hash) int {
abig, bbig := new(big.Int).SetBytes(a[:]), new(big.Int).SetBytes(b[:])
return new(big.Int).Xor(abig, bbig).BitLen()
}
if err := quick.CheckEqual(logdist, logdistBig, quickcfg()); err != nil {
t.Error(err)
}
}

// the random tests is likely to miss the case where they're equal.
func TestNodeID_logdistEqual(t *testing.T) {
x := common.Hash{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
if logdist(x, x) != 0 {
t.Errorf("logdist(x, x) != 0")
}
}

func TestNodeID_hashAtDistance(t *testing.T) {
// we don't use quick.Check here because its output isn't
// very helpful when the test fails.
cfg := quickcfg()
for i := 0; i < cfg.MaxCount; i++ {
a := gen(common.Hash{}, cfg.Rand).(common.Hash)
dist := cfg.Rand.Intn(len(common.Hash{}) * 8)
result := hashAtDistance(a, dist)
actualdist := logdist(result, a)

if dist != actualdist {
t.Log("a:     ", a)
t.Log("result:", result)
t.Fatalf("#%d: distance of result is %d, want %d", i, actualdist, dist)
}
}
```

```
}

func quickcfg() *quick.Config {
return &quick.Config{
MaxCount: 5000,
Rand:     rand.New(rand.NewSource(time.Now().Unix())),
}
}

// TODO: The Generate method can be dropped when we require Go >= 1.5
// because testing/quick learned to generate arrays in 1.5.

func (NodeID) Generate(rand *rand.Rand, size int) reflect.Value {
var id NodeID
m := rand.Intn(len(id))
for i := len(id) - 1; i > m; i-- {
id[i] = byte(rand.Uint32())
}
return reflect.ValueOf(id)
}

87:F:\git\coin\ethereum\go-ethereum\p2p\discover\ntp.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains the NTP time drift detection via the SNTP protocol:
//   https://tools.ietf.org/html/rfc4330

package discover

import (
"fmt"
"net"
"sort"
"time"

"github.com/ethereum/go-ethereum/log"
)

const (
ntpPool   = "pool.ntp.org" // ntpPool is the NTP server to query for the current time
ntpChecks = 3              // Number of measurements to do against the NTP server
)
```

```
// durationSlice attaches the methods of sort.Interface to []time.Duration,
// sorting in increasing order.
type durationSlice []time.Duration

func (s durationSlice) Len() int           { return len(s) }
func (s durationSlice) Less(i, j int) bool { return s[i] < s[j] }
func (s durationSlice) Swap(i, j int)      { s[i], s[j] = s[j], s[i] }

// checkClockDrift queries an NTP server for clock drifts and warns the user if
// one large enough is detected.
func checkClockDrift() {
drift, err := sntpDrift(ntpChecks)
if err != nil {
return
}
if drift < -driftThreshold || drift > driftThreshold {
log.Warn(fmt.Sprintf("System clock seems off by %v, which can prevent network connectivity",
drift))
log.Warn("Please enable network time synchronisation in system settings.")
} else {
log.Debug("NTP sanity check done", "drift", drift)
}
}

// sntpDrift does a naive time resolution against an NTP server and returns the
// measured drift. This method uses the simple version of NTP. It's not precise
// but should be fine for these purposes.
//
// Note, it executes two extra measurements compared to the number of requested
// ones to be able to discard the two extremes as outliers.
func sntpDrift(measurements int) (time.Duration, error) {
// Resolve the address of the NTP server
addr, err := net.ResolveUDPAddr("udp", ntpPool+":123")
if err != nil {
return 0, err
}
// Construct the time request (empty package with only 2 fields set):
//   Bits 3-5: Protocol version, 3
//   Bits 6-8: Mode of operation, client, 3
request := make([]byte, 48)
request[0] = 3<<3 | 3
```

```
// Execute each of the measurements
drifts := []time.Duration{}
for i := 0; i < measurements+2; i++ {
// Dial the NTP server and send the time retrieval request
conn, err := net.DialUDP("udp", nil, addr)
if err != nil {
return 0, err
}
defer conn.Close()

sent := time.Now()
if _, err = conn.Write(request); err != nil {
return 0, err
}
// Retrieve the reply and calculate the elapsed time
conn.SetDeadline(time.Now().Add(5 * time.Second))

reply := make([]byte, 48)
if _, err = conn.Read(reply); err != nil {
return 0, err
}
elapsed := time.Since(sent)

// Reconstruct the time from the reply data
sec := uint64(reply[43]) | uint64(reply[42])<<8 | uint64(reply[41])<<16 | uint64(reply[40])<<24
frac := uint64(reply[47]) | uint64(reply[46])<<8 | uint64(reply[45])<<16 | uint64(reply[44])<<24

nanosec := sec*1e9 + (frac*1e9)>>32

t := time.Date(1900, 1, 1, 0, 0, 0, 0, time.UTC).Add(time.Duration(nanosec)).Local()

// Calculate the drift based on an assumed answer time of RRT/2
drifts = append(drifts, sent.Sub(t)+elapsed/2)
}
// Calculate average drif (drop two extremities to avoid outliers)
sort.Sort(durationSlice(drifts))

drift := time.Duration(0)
for i := 1; i < len(drifts)-1; i++ {
drift += drifts[i]
}
```

```
return drift / time.Duration(measurements), nil
}

88:F:\git\coin\ethereum\go-ethereum\p2p\discover\table.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package discover implements the Node Discovery Protocol.
//
// The Node Discovery protocol provides a way to find RLPx nodes that
// can be connected to. It uses a Kademlia-like protocol to maintain a
// distributed database of the IDs and endpoints of all listening
// nodes.
package discover

import (
"crypto/rand"
"encoding/binary"
"errors"
"fmt"
"net"
"sort"
"sync"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/log"
)

const (
alpha      = 3  // Kademlia concurrency factor
bucketSize = 16 // Kademlia bucket size
hashBits   = len(common.Hash{}) * 8
nBuckets   = hashBits + 1 // Number of buckets

maxBondingPingPongs = 16
maxFindnodeFailures = 5

autoRefreshInterval = 1 * time.Hour
seedCount           = 30
seedMaxAge          = 5 * 24 * time.Hour
)
```

```
type Table struct {
mutex   sync.Mutex        // protects buckets, their content, and nursery
buckets [nBuckets]*bucket // index of known nodes by distance
nursery []*Node           // bootstrap nodes
db      *nodeDB           // database of known nodes

refreshReq chan chan struct{}
closeReq   chan struct{}
closed     chan struct{}

bondmu    sync.Mutex
bonding   map[NodeID]*bondproc
bondslots chan struct{} // limits total number of active bonding processes

nodeAddedHook func(*Node) // for testing

net  transport
self *Node // metadata of the local node
}

type bondproc struct {
err  error
n    *Node
done chan struct{}
}

// transport is implemented by the UDP transport.
// it is an interface so we can test without opening lots of UDP
// sockets and without generating a private key.
type transport interface {
ping(NodeID, *net.UDPAddr) error
waitping(NodeID) error
findnode(toid NodeID, addr *net.UDPAddr, target NodeID) ([]*Node, error)
close()
}

// bucket contains nodes, ordered by their last activity. the entry
// that was most recently active is the first element in entries.
type bucket struct{ entries []*Node }

func newTable(t transport, ourID NodeID, ourAddr *net.UDPAddr, nodeDBPath string) (*Table,
```

```
error) {
// If no node database was given, use an in-memory one
db, err := newNodeDB(nodeDBPath, Version, ourID)
if err != nil {
return nil, err
}
tab := &Table{
net:        t,
db:         db,
self:       NewNode(ourID, ourAddr.IP, uint16(ourAddr.Port), uint16(ourAddr.Port)),
bonding:    make(map[NodeID]*bondproc),
bondslots:  make(chan struct{}, maxBondingPingPongs),
refreshReq: make(chan chan struct{}),
closeReq:   make(chan struct{}),
closed:     make(chan struct{}),
}
for i := 0; i < cap(tab.bondslots); i++ {
tab.bondslots <- struct{}{}
}
for i := range tab.buckets {
tab.buckets[i] = new(bucket)
}
go tab.refreshLoop()
return tab, nil
}

// Self returns the local node.
// The returned node should not be modified by the caller.
func (tab *Table) Self() *Node {
return tab.self
}

// ReadRandomNodes fills the given slice with random nodes from the
// table. It will not write the same node more than once. The nodes in
// the slice are copies and can be modified by the caller.
func (tab *Table) ReadRandomNodes(buf []*Node) (n int) {
tab.mutex.Lock()
defer tab.mutex.Unlock()
// TODO: tree-based buckets would help here
// Find all non-empty buckets and get a fresh slice of their entries.
var buckets [][]*Node
for _, b := range tab.buckets {
```

```
if len(b.entries) > 0 {
buckets = append(buckets, b.entries[:])
}
}
if len(buckets) == 0 {
return 0
}
// Shuffle the buckets.
for i := uint32(len(buckets)) - 1; i > 0; i-- {
j := randUint(i)
buckets[i], buckets[j] = buckets[j], buckets[i]
}
// Move head of each bucket into buf, removing buckets that become empty.
var i, j int
for ; i < len(buf); i, j = i+1, (j+1)%len(buckets) {
b := buckets[j]
buf[i] = &(*b[0])
buckets[j] = b[1:]
if len(b) == 1 {
buckets = append(buckets[:j], buckets[j+1:]...)
}
if len(buckets) == 0 {
break
}
}
return i + 1
}

func randUint(max uint32) uint32 {
if max == 0 {
return 0
}
var b [4]byte
rand.Read(b[:])
return binary.BigEndian.Uint32(b[:]) % max
}

// Close terminates the network listener and flushes the node database.
func (tab *Table) Close() {
select {
case <-tab.closed:
// already closed.
```

```
case tab.closeReq <- struct{}{}:
<-tab.closed // wait for refreshLoop to end.
}
}

// SetFallbackNodes sets the initial points of contact. These nodes
// are used to connect to the network if the table is empty and there
// are no known nodes in the database.
func (tab *Table) SetFallbackNodes(nodes []*Node) error {
for _, n := range nodes {
if err := n.validateComplete(); err != nil {
return fmt.Errorf("bad bootstrap/fallback node %q (%v)", n, err)
}
}
tab.mutex.Lock()
tab.nursery = make([]*Node, 0, len(nodes))
for _, n := range nodes {
cpy := *n
// Recompute cpy.sha because the node might not have been
// created by NewNode or ParseNode.
cpy.sha = crypto.Keccak256Hash(n.ID[:])
tab.nursery = append(tab.nursery, &cpy)
}
tab.mutex.Unlock()
tab.refresh()
return nil
}

// Resolve searches for a specific node with the given ID.
// It returns nil if the node could not be found.
func (tab *Table) Resolve(targetID NodeID) *Node {
// If the node is present in the local table, no
// network interaction is required.
hash := crypto.Keccak256Hash(targetID[:])
tab.mutex.Lock()
cl := tab.closest(hash, 1)
tab.mutex.Unlock()
if len(cl.entries) > 0 && cl.entries[0].ID == targetID {
return cl.entries[0]
}
// Otherwise, do a network lookup.
result := tab.Lookup(targetID)
```

```
for _, n := range result {
if n.ID == targetID {
return n
}
}
return nil
}

// Lookup performs a network search for nodes close
// to the given target. It approaches the target by querying
// nodes that are closer to it on each iteration.
// The given target does not need to be an actual node
// identifier.
func (tab *Table) Lookup(targetID NodeID) []*Node {
return tab.lookup(targetID, true)
}

func (tab *Table) lookup(targetID NodeID, refreshIfEmpty bool) []*Node {
var (
target         = crypto.Keccak256Hash(targetID[:])
asked          = make(map[NodeID]bool)
seen           = make(map[NodeID]bool)
reply          = make(chan []*Node, alpha)
pendingQueries = 0
result         *nodesByDistance
)
// don't query further if we hit ourself.
// unlikely to happen often in practice.
asked[tab.self.ID] = true

for {
tab.mutex.Lock()
// generate initial result set
result = tab.closest(target, bucketSize)
tab.mutex.Unlock()
if len(result.entries) > 0 || !refreshIfEmpty {
break
}
// The result set is empty, all nodes were dropped, refresh.
// We actually wait for the refresh to complete here. The very
// first query will hit this case and run the bootstrapping
// logic.
```

```
<-tab.refresh()
refreshIfEmpty = false
}

for {
// ask the alpha closest nodes that we haven't asked yet
for i := 0; i < len(result.entries) && pendingQueries < alpha; i++ {
n := result.entries[i]
if !asked[n.ID] {
asked[n.ID] = true
pendingQueries++
go func() {
// Find potential neighbors to bond with
r, err := tab.net.findnode(n.ID, n.addr(), targetID)
if err != nil {
// Bump the failure counter to detect and evacuate non-bonded entries
fails := tab.db.findFails(n.ID) + 1
tab.db.updateFindFails(n.ID, fails)
log.Trace("Bumping findnode failure counter", "id", n.ID, "failcount", fails)

if fails >= maxFindnodeFailures {
log.Trace("Too many findnode failures, dropping", "id", n.ID, "failcount", fails)
tab.delete(n)
}
}
reply <- tab.bondall(r)
}()
}
}
if pendingQueries == 0 {
// we have asked all closest nodes, stop the search
break
}
// wait for the next reply
for _, n := range <-reply {
if n != nil && !seen[n.ID] {
seen[n.ID] = true
result.push(n, bucketSize)
}
}
pendingQueries--
}
```

```
return result.entries
}

func (tab *Table) refresh() <-chan struct{} {
done := make(chan struct{})
select {
case tab.refreshReq <- done:
case <-tab.closed:
close(done)
}
return done
}

// refreshLoop schedules doRefresh runs and coordinates shutdown.
func (tab *Table) refreshLoop() {
var (
timer   = time.NewTicker(autoRefreshInterval)
waiting []chan struct{} // accumulates waiting callers while doRefresh runs
done    chan struct{}   // where doRefresh reports completion
)
loop:
for {
select {
case <-timer.C:
if done == nil {
done = make(chan struct{})
go tab.doRefresh(done)
}
case req := <-tab.refreshReq:
waiting = append(waiting, req)
if done == nil {
done = make(chan struct{})
go tab.doRefresh(done)
}
case <-done:
for _, ch := range waiting {
close(ch)
}
waiting = nil
done = nil
case <-tab.closeReq:
break loop
```

```
}
}

if tab.net != nil {
tab.net.close()
}
if done != nil {
<-done
}
for _, ch := range waiting {
close(ch)
}
tab.db.close()
close(tab.closed)
}

// doRefresh performs a lookup for a random target to keep buckets
// full. seed nodes are inserted if the table is empty (initial
// bootstrap or discarded faulty peers).
func (tab *Table) doRefresh(done chan struct{}) {
defer close(done)

// The Kademlia paper specifies that the bucket refresh should
// perform a lookup in the least recently used bucket. We cannot
// adhere to this because the findnode target is a 512bit value
// (not hash-sized) and it is not easily possible to generate a
// sha3 preimage that falls into a chosen bucket.
// We perform a lookup with a random target instead.
var target NodeID
rand.Read(target[:])
result := tab.lookup(target, false)
if len(result) > 0 {
return
}

// The table is empty. Load nodes from the database and insert
// them. This should yield a few previously seen nodes that are
// (hopefully) still alive.
seeds := tab.db.querySeeds(seedCount, seedMaxAge)
seeds = tab.bondall(append(seeds, tab.nursery...))

if len(seeds) == 0 {
```

```
log.Debug("No discv4 seed nodes found")
}
for _, n := range seeds {
age := log.Lazy{Fn: func() time.Duration { return time.Since(tab.db.lastPong(n.ID)) }}
log.Trace("Found seed node in database", "id", n.ID, "addr", n.addr(), "age", age)
}
tab.mutex.Lock()
tab.stuff(seeds)
tab.mutex.Unlock()

// Finally, do a self lookup to fill up the buckets.
tab.lookup(tab.self.ID, false)
}

// closest returns the n nodes in the table that are closest to the
// given id. The caller must hold tab.mutex.
func (tab *Table) closest(target common.Hash, nresults int) *nodesByDistance {
// This is a very wasteful way to find the closest nodes but
// obviously correct. I believe that tree-based buckets would make
// this easier to implement efficiently.
close := &nodesByDistance{target: target}
for _, b := range tab.buckets {
for _, n := range b.entries {
close.push(n, nresults)
}
}
return close
}

func (tab *Table) len() (n int) {
for _, b := range tab.buckets {
n += len(b.entries)
}
return n
}

// bondall bonds with all given nodes concurrently and returns
// those nodes for which bonding has probably succeeded.
func (tab *Table) bondall(nodes []*Node) (result []*Node) {
rc := make(chan *Node, len(nodes))
for i := range nodes {
go func(n *Node) {
```

```
nn, _ := tab.bond(false, n.ID, n.addr(), uint16(n.TCP))
rc <- nn
}(nodes[i])
}
for range nodes {
if n := <-rc; n != nil {
result = append(result, n)
}
}
return result
}

// bond ensures the local node has a bond with the given remote node.
// It also attempts to insert the node into the table if bonding succeeds.
// The caller must not hold tab.mutex.
//
// A bond is must be established before sending findnode requests.
// Both sides must have completed a ping/pong exchange for a bond to
// exist. The total number of active bonding processes is limited in
// order to restrain network use.
//
// bond is meant to operate idempotently in that bonding with a remote
// node which still remembers a previously established bond will work.
// The remote node will simply not send a ping back, causing waitping
// to time out.
//
// If pinged is true, the remote node has just pinged us and one half
// of the process can be skipped.
func (tab *Table) bond(pinged bool, id NodeID, addr *net.UDPAddr, tcpPort uint16) (*Node, error) {
if id == tab.self.ID {
return nil, errors.New("is self")
}
// Retrieve a previously known node and any recent findnode failures
node, fails := tab.db.node(id), 0
if node != nil {
fails = tab.db.findFails(id)
}
// If the node is unknown (non-bonded) or failed (remotely unknown), bond from scratch
var result error
age := time.Since(tab.db.lastPong(id))
if node == nil || fails > 0 || age > nodeDBNodeExpiration {
log.Trace("Starting bonding ping/pong", "id", id, "known", node != nil, "failcount", fails, "age", age)
```

```
tab.bondmu.Lock()
w := tab.bonding[id]
if w != nil {
// Wait for an existing bonding process to complete.
tab.bondmu.Unlock()
<-w.done
} else {
// Register a new bonding process.
w = &bondproc{done: make(chan struct{})}
tab.bonding[id] = w
tab.bondmu.Unlock()
// Do the ping/pong. The result goes into w.
tab.pingpong(w, pinged, id, addr, tcpPort)
// Unregister the process after it's done.
tab.bondmu.Lock()
delete(tab.bonding, id)
tab.bondmu.Unlock()
}
// Retrieve the bonding results
result = w.err
if result == nil {
node = w.n
}
}
if node != nil {
// Add the node to the table even if the bonding ping/pong
// fails. It will be relaced quickly if it continues to be
// unresponsive.
tab.add(node)
tab.db.updateFindFails(id, 0)
}
return node, result
}

func (tab *Table) pingpong(w *bondproc, pinged bool, id NodeID, addr *net.UDPAddr, tcpPort
uint16) {
// Request a bonding slot to limit network usage
<-tab.bondslots
defer func() { tab.bondslots <- struct{}{} }()

// Ping the remote side and wait for a pong.
```

```
if w.err = tab.ping(id, addr); w.err != nil {
close(w.done)
return
}
if !pinged {
// Give the remote node a chance to ping us before we start
// sending findnode requests. If they still remember us,
// waitping will simply time out.
tab.net.waitping(id)
}
// Bonding succeeded, update the node database.
w.n = NewNode(id, addr.IP, uint16(addr.Port), tcpPort)
tab.db.updateNode(w.n)
close(w.done)
}

// ping a remote endpoint and wait for a reply, also updating the node
// database accordingly.
func (tab *Table) ping(id NodeID, addr *net.UDPAddr) error {
tab.db.updateLastPing(id, time.Now())
if err := tab.net.ping(id, addr); err != nil {
return err
}
tab.db.updateLastPong(id, time.Now())

// Start the background expiration goroutine after the first
// successful communication. Subsequent calls have no effect if it
// is already running. We do this here instead of somewhere else
// so that the search for seed nodes also considers older nodes
// that would otherwise be removed by the expiration.
tab.db.ensureExpirer()
return nil
}

// add attempts to add the given node its corresponding bucket. If the
// bucket has space available, adding the node succeeds immediately.
// Otherwise, the node is added if the least recently active node in
// the bucket does not respond to a ping packet.
//
// The caller must not hold tab.mutex.
func (tab *Table) add(new *Node) {
b := tab.buckets[logdist(tab.self.sha, new.sha)]
```

```
tab.mutex.Lock()
defer tab.mutex.Unlock()
if b.bump(new) {
return
}
var oldest *Node
if len(b.entries) == bucketSize {
oldest = b.entries[bucketSize-1]
if oldest.contested {
// The node is already being replaced, don't attempt
// to replace it.
return
}
oldest.contested = true
// Let go of the mutex so other goroutines can access
// the table while we ping the least recently active node.
tab.mutex.Unlock()
err := tab.ping(oldest.ID, oldest.addr())
tab.mutex.Lock()
oldest.contested = false
if err == nil {
// The node responded, don't replace it.
return
}
}
added := b.replace(new, oldest)
if added && tab.nodeAddedHook != nil {
tab.nodeAddedHook(new)
}
}

// stuff adds nodes the table to the end of their corresponding bucket
// if the bucket is not full. The caller must hold tab.mutex.
func (tab *Table) stuff(nodes []*Node) {
outer:
for _, n := range nodes {
if n.ID == tab.self.ID {
continue // don't add self
}
bucket := tab.buckets[logdist(tab.self.sha, n.sha)]
for i := range bucket.entries {
if bucket.entries[i].ID == n.ID {
```

```
continue outer // already in bucket
}
}
if len(bucket.entries) < bucketSize {
bucket.entries = append(bucket.entries, n)
if tab.nodeAddedHook != nil {
tab.nodeAddedHook(n)
}
}
}
}

// delete removes an entry from the node table (used to evacuate
// failed/non-bonded discovery peers).
func (tab *Table) delete(node *Node) {
tab.mutex.Lock()
defer tab.mutex.Unlock()
bucket := tab.buckets[logdist(tab.self.sha, node.sha)]
for i := range bucket.entries {
if bucket.entries[i].ID == node.ID {
bucket.entries = append(bucket.entries[:i], bucket.entries[i+1:]...)
return
}
}
}

func (b *bucket) replace(n *Node, last *Node) bool {
// Don't add if b already contains n.
for i := range b.entries {
if b.entries[i].ID == n.ID {
return false
}
}
// Replace last if it is still the last entry or just add n if b
// isn't full. If is no longer the last entry, it has either been
// replaced with someone else or became active.
if len(b.entries) == bucketSize && (last == nil || b.entries[bucketSize-1].ID != last.ID) {
return false
}
if len(b.entries) < bucketSize {
b.entries = append(b.entries, nil)
}
```

```
copy(b.entries[1:], b.entries)
b.entries[0] = n
return true
}

func (b *bucket) bump(n *Node) bool {
for i := range b.entries {
if b.entries[i].ID == n.ID {
// move it to the front
copy(b.entries[1:], b.entries[:i])
b.entries[0] = n
return true
}
}
return false
}

// nodesByDistance is a list of nodes, ordered by
// distance to target.
type nodesByDistance struct {
entries []*Node
target  common.Hash
}

// push adds the given node to the list, keeping the total size below maxElems.
func (h *nodesByDistance) push(n *Node, maxElems int) {
ix := sort.Search(len(h.entries), func(i int) bool {
return distcmp(h.target, h.entries[i].sha, n.sha) > 0
})
if len(h.entries) < maxElems {
h.entries = append(h.entries, n)
}
if ix == len(h.entries) {
// farther away than all nodes we already have.
// if there was room for it, the node is now the last element.
} else {
// slide existing entries down to make room
// this will overwrite the entry we just appended.
copy(h.entries[ix+1:], h.entries[ix:])
h.entries[ix] = n
}
}
```

89:F:\git\coin\ethereum\go-ethereum\p2p\discover\table_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discover

import (
"crypto/ecdsa"
"fmt"
"math/rand"

"net"
"reflect"
"testing"
"testing/quick"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
)

func TestTable_pingReplace(t *testing.T) {
doit := func(newNodeIsResponding, lastInBucketIsResponding bool) {
transport := newPingRecorder()
tab, _ := newTable(transport, NodeID{}, &net.UDPAddr{}, "")
defer tab.Close()
pingSender :=
NewNode(MustHexID("a502af0f59b2aab7746995408c79e9ca312d2793cc997e44fc55eda62f0150
bbb8c59a6f9269ba3a081518b62699ee807c7c19c20125ddfccca872608af9e370"), net.IP{}, 99,
99)

// fill up the sender's bucket.
last := fillBucket(tab, 253)

// this call to bond should replace the last node
// in its bucket if the node is not responding.
transport.responding[last.ID] = lastInBucketIsResponding
transport.responding[pingSender.ID] = newNodeIsResponding
tab.bond(true, pingSender.ID, &net.UDPAddr{}, 0)

// first ping goes to sender (bonding pingback)
if !transport.pinged[pingSender.ID] {

```
t.Error("table did not ping back sender")
}
if newNodeIsResponding {
// second ping goes to oldest node in bucket
// to see whether it is still alive.
if !transport.pinged[last.ID] {
t.Error("table did not ping last node in bucket")
}
}

tab.mutex.Lock()
defer tab.mutex.Unlock()
if l := len(tab.buckets[253].entries); l != bucketSize {
t.Errorf("wrong bucket size after bond: got %d, want %d", l, bucketSize)
}

if lastInBucketIsResponding || !newNodeIsResponding {
if !contains(tab.buckets[253].entries, last.ID) {
t.Error("last entry was removed")
}
if contains(tab.buckets[253].entries, pingSender.ID) {
t.Error("new entry was added")
}
} else {
if contains(tab.buckets[253].entries, last.ID) {
t.Error("last entry was not removed")
}
if !contains(tab.buckets[253].entries, pingSender.ID) {
t.Error("new entry was not added")
}
}
}

doit(true, true)
doit(false, true)
doit(true, false)
doit(false, false)
}

func TestBucket_bumpNoDuplicates(t *testing.T) {
t.Parallel()
cfg := &quick.Config{
```

```
MaxCount: 1000,
Rand:     rand.New(rand.NewSource(time.Now().Unix())),
Values: func(args []reflect.Value, rand *rand.Rand) {
// generate a random list of nodes. this will be the content of the bucket.
n := rand.Intn(bucketSize-1) + 1
nodes := make([]*Node, n)
for i := range nodes {
nodes[i] = nodeAtDistance(common.Hash{}, 200)
}
args[0] = reflect.ValueOf(nodes)
// generate random bump positions.
bumps := make([]int, rand.Intn(100))
for i := range bumps {
bumps[i] = rand.Intn(len(nodes))
}
args[1] = reflect.ValueOf(bumps)
},
}

prop := func(nodes []*Node, bumps []int) (ok bool) {
b := &bucket{entries: make([]*Node, len(nodes))}
copy(b.entries, nodes)
for i, pos := range bumps {
b.bump(b.entries[pos])
if hasDuplicates(b.entries) {
t.Logf("bucket has duplicates after %d/%d bumps:", i+1, len(bumps))
for _, n := range b.entries {
t.Logf("  %p", n)
}
return false
}
}
return true
}
if err := quick.Check(prop, cfg); err != nil {
t.Error(err)
}
}

// fillBucket inserts nodes into the given bucket until
// it is full. The node's IDs dont correspond to their
// hashes.
```

```
func fillBucket(tab *Table, ld int) (last *Node) {
b := tab.buckets[ld]
for len(b.entries) < bucketSize {
b.entries = append(b.entries, nodeAtDistance(tab.self.sha, ld))
}
return b.entries[bucketSize-1]
}

// nodeAtDistance creates a node for which logdist(base, n.sha) == ld.
// The node's ID does not correspond to n.sha.
func nodeAtDistance(base common.Hash, ld int) (n *Node) {
n = new(Node)
n.sha = hashAtDistance(base, ld)
n.IP = net.IP{10, 0, 2, byte(ld)}
copy(n.ID[:], n.sha[:]) // ensure the node still has a unique ID
return n
}

type pingRecorder struct{ responding, pinged map[NodeID]bool }

func newPingRecorder() *pingRecorder {
return &pingRecorder{make(map[NodeID]bool), make(map[NodeID]bool)}
}

func (t *pingRecorder) findnode(toid NodeID, toaddr *net.UDPAddr, target NodeID) ([]*Node, error)
{
panic("findnode called on pingRecorder")
}
func (t *pingRecorder) close() {}
func (t *pingRecorder) waitping(from NodeID) error {
return nil // remote always pings
}
func (t *pingRecorder) ping(toid NodeID, toaddr *net.UDPAddr) error {
t.pinged[toid] = true
if t.responding[toid] {
return nil
} else {
return errTimeout
}
}

func TestTable_closest(t *testing.T) {
```

```
t.Parallel()

test := func(test *closeTest) bool {
// for any node table, Target and N
tab, _ := newTable(nil, test.Self, &net.UDPAddr{}, "")
defer tab.Close()
tab.stuff(test.All)

// check that doClosest(Target, N) returns nodes
result := tab.closest(test.Target, test.N).entries
if hasDuplicates(result) {
t.Errorf("result contains duplicates")
return false
}
if !sortedByDistanceTo(test.Target, result) {
t.Errorf("result is not sorted by distance to target")
return false
}

// check that the number of results is min(N, tablen)
wantN := test.N
if tlen := tab.len(); tlen < test.N {
wantN = tlen
}
if len(result) != wantN {
t.Errorf("wrong number of nodes: got %d, want %d", len(result), wantN)
return false
} else if len(result) == 0 {
return true // no need to check distance
}

// check that the result nodes have minimum distance to target.
for _, b := range tab.buckets {
for _, n := range b.entries {
if contains(result, n.ID) {
continue // don't run the check below for nodes in result
}
farthestResult := result[len(result)-1].sha
if distcmp(test.Target, n.sha, farthestResult) < 0 {
t.Errorf("table contains node that is closer to target but it's not in result")
t.Logf("  Target:          %v", test.Target)
t.Logf("  Farthest Result: %v", farthestResult)
```

```
t.Logf("  ID:               %v", n.ID)
return false
}
}
}
return true
}
if err := quick.Check(test, quickcfg()); err != nil {
t.Error(err)
}
}

func TestTable_ReadRandomNodesGetAll(t *testing.T) {
cfg := &quick.Config{
MaxCount: 200,
Rand:     rand.New(rand.NewSource(time.Now().Unix())),
Values: func(args []reflect.Value, rand *rand.Rand) {
args[0] = reflect.ValueOf(make([]*Node, rand.Intn(1000)))
},
}
test := func(buf []*Node) bool {
tab, _ := newTable(nil, NodeID{}, &net.UDPAddr{}, "")
defer tab.Close()
for i := 0; i < len(buf); i++ {
ld := cfg.Rand.Intn(len(tab.buckets))
tab.stuff([]*Node{nodeAtDistance(tab.self.sha, ld)})
}
gotN := tab.ReadRandomNodes(buf)
if gotN != tab.len() {
t.Errorf("wrong number of nodes, got %d, want %d", gotN, tab.len())
return false
}
if hasDuplicates(buf[:gotN]) {
t.Errorf("result contains duplicates")
return false
}
return true
}
if err := quick.Check(test, cfg); err != nil {
t.Error(err)
}
}
```

```
type closeTest struct {
Self   NodeID
Target common.Hash
All    []*Node
N      int
}

func (*closeTest) Generate(rand *rand.Rand, size int) reflect.Value {
t := &closeTest{
Self:   gen(NodeID{}, rand).(NodeID),
Target: gen(common.Hash{}, rand).(common.Hash),
N:      rand.Intn(bucketSize),
}
for _, id := range gen([]NodeID{}, rand).([]NodeID) {
t.All = append(t.All, &Node{ID: id})
}
return reflect.ValueOf(t)
}

func TestTable_Lookup(t *testing.T) {
self := nodeAtDistance(common.Hash{}, 0)
tab, _ := newTable(lookupTestnet, self.ID, &net.UDPAddr{}, "")
defer tab.Close()

// lookup on empty table returns no nodes
if results := tab.Lookup(lookupTestnet.target); len(results) > 0 {
t.Fatalf("lookup on empty table returned %d results: %#v", len(results), results)
}
// seed table with initial node (otherwise lookup will terminate immediately)
seed := NewNode(lookupTestnet.dists[256][0], net.IP{}, 256, 0)
tab.stuff([]*Node{seed})

results := tab.Lookup(lookupTestnet.target)
t.Logf("results:")
for _, e := range results {
t.Logf("  ld=%d, %x", logdist(lookupTestnet.targetSha, e.sha), e.sha[:])
}
if len(results) != bucketSize {
t.Errorf("wrong number of results: got %d, want %d", len(results), bucketSize)
}
if hasDuplicates(results) {
```

```
t.Errorf("result set contains duplicate entries")
}
if !sortedByDistanceTo(lookupTestnet.targetSha, results) {
t.Errorf("result set not sorted by distance to target")
}
// TODO: check result nodes are actually closest
}

// This is the test network for the Lookup test.
// The nodes were obtained by running testnet.mine with a random NodeID as target.
var lookupTestnet = &preminedTestnet{
target:
MustHexID("166aea4f556532c6d34e8b740e5d314af7e9ac0ca79833bd751d6b665f12dfd38ec563
c363b32f02aef4a80b44fd3def94612d497b99cb5f17fd24de454927ec"),
targetSha: common.Hash{0x5c, 0x94, 0x4e, 0xe5, 0x1c, 0x5a, 0xe9, 0xf7, 0x2a, 0x95, 0xec, 0xcb,
0x8a, 0xed, 0x3, 0x74, 0xee, 0xcb, 0x51, 0x19, 0xd7, 0x20, 0xcb, 0xea, 0x68, 0x13, 0xe8, 0xe0,
0xd6, 0xad, 0x92, 0x61},
dists: [257][]NodeID{
240: {
MustHexID("2001ad5e3e80c71b952161bc0186731cf5ffe942d24a79230a0555802296238e57ea7a
32f5b6f18564eadc1c65389448481f8c9338df0a3dbd18f708cbc2cbcb"),
MustHexID("6ba3f4f57d084b6bf94cc4555b8c657e4a8ac7b7baf23c6874efc21dd1e4f56b7eb2721
e07f5242d2f1d8381fc8cae535e860197c69236798ba1ad231b105794"),
},
244: {
MustHexID("696ba1f0a9d55c59246f776600542a9e6432490f0cd78f8bb55a196918df2081a9b521c
3c3ba48e465a75c10768807717f8f689b0b4adce00e1c75737552a178"),
},
246: {
MustHexID("d6d32178bdc38416f46ffb8b3ec9e4cb2cfff8d04dd7e4311a70e403cb62b10be1b4473
11b60b4f9ee221a8131fc2cbd45b96dd80deba68a949d467241facfa8"),
MustHexID("3ea3d04a43a3dfb5ac11cffc2319248cf41b6279659393c2f55b8a0a5fc9d12581a9d97
ef5d8ff9b5abf3321a290e8f63a4f785f450dc8a672aba3ba2ff4fdab"),
MustHexID("2fc897f05ae585553e5c014effd3078f84f37f9333afacffb109f00ca8e7a3373de810a394
6be971cbccdfd40249f9fe7f322118ea459ac71acca85a1ef8b7f4"),
},
247: {
MustHexID("3155e1427f85f10a5c9a7755877748041af1bcd8d474ec065eb33df57a97babf54bfd21
03575fa829115d224c523596b401065a97f74010610fce76382c0bf32"),
MustHexID("312c55512422cf9b8a4097e9a6ad79402e87a15ae909a4bfefa22398f03d20951933be
ea1e4dfa6f968212385e829f04c2d314fc2d4e255e0d3bc08792b069db"),
MustHexID("38643200b172dcfef857492156971f0e6aa2c538d8b74010f8e140811d53b98c765dd2
```

d96126051913f44582e8c199ad7c6d6819e9a56483f637feaac9448aac"),
MustHexID("8dcab8618c3253b558d459da53bd8fa68935a719aff8b811197101a4b2b47dd2d4729
5286fc00cc081bb542d760717d1bdd6bec2c37cd72eca367d6dd3b9df73"),
MustHexID("8b58c6073dd98bbad4e310b97186c8f822d3a5c7d57af40e2136e88e315afd115edb27
d2d0685a908cfe5aa49d0debdda6e6e63972691d6bd8c5af2d771dd2a9"),
MustHexID("2cbb718b7dc682da19652e7d9eb4fefaf7b7147d82c1c2b6805edf77b85e29fde9f6da1
95741467ff2638dc62c8d3e014ea5686693c15ed0080b6de90354c137"),
MustHexID("e84027696d3f12f2de30a9311afea8fbd313c2360daff52bb5fc8c7094d5295758bec313
4e4eef24e4cdf377b40da344993284628a7a346eba94f74160998feb"),
MustHexID("f1357a4f04f9d33753a57c0b65ba20a5d8777abbffd04e906014491c9103fb08590e455
48d37aa4bd70965e2e81ddba94f31860348df01469eec8c1829200a68"),
MustHexID("4ab0a75941b12892369b4490a1928c8ca52a9ad6d3dffbd1d8c0b907bc200fe74c022d
011ec39b64808a39c0ca41f1d3254386c3e7733e7044c44259486461b6"),
MustHexID("d45150a72dc74388773e68e03133a3b5f51447fe91837d566706b3c035ee4b56f160c8
78c6273394daee7f56cc398985269052f22f75a8057df2fe6172765354"),
},
248: {
MustHexID("6aadfce366a189bab08ac84721567483202c86590642ea6d6a14f37ca78d82bdb6509
eb7b8b2f6f63c78ae3ae1d8837c89509e41497d719b23ad53dd81574afa"),
MustHexID("a605ecfd6069a4cf4cf7f5840e5bc0ce10d23a3ac59e2aaa70c6afd5637359d2519b452
4f56fc2ca180cdbebe54262f720ccaae8c1b28fd553c485675831624d"),
MustHexID("29701451cb9448ca33fc33680b44b840d815be90146eb521641efbffed0859c154e889
2d3906eae9934bfacee72cd1d2fa9dd050fd18888eea49da155ab0efd2"),
MustHexID("3ed426322dee7572b08592e1e079f8b6c6b30e10e6243edd144a6a48fdbdb83df73a6
e41b1143722cb82604f2203a32758610b5d9544f44a1a7921ba001528c1"),
MustHexID("b2e2a2b7fdd363572a3256e75435fab1da3b16f7891a8bd2015f30995dae665d7eabfd
194d87d99d5df628b4bbc7b04e5b492c596422dd8272746c7a1b0b8e4f"),
MustHexID("0c69c9756162c593e85615b814ce57a2a8ca2df6c690b9c4e4602731b61e1531a3bbe
3f7114271554427ffabea80ad8f36fa95a49fa77b675ae182c6ccac1728"),
MustHexID("8d28be21d5a97b0876442fa4f5e5387f5bf3faad0b6f13b8607b64d6e448c0991ca28dd
7fe2f64eb8eadd7150bff5d5666aa6ed868b84c71311f4ba9a38569dd"),
MustHexID("2c677e1c64b9c9df6359348a7f5f33dc79e22f0177042486d125f8b6ca7f0dc756b1f672
aceee5f1746bcff80aaf6f92a8dc0c9fbeb259b3fa0da060de5ab7e8"),
MustHexID("3994880f94a8678f0cd247a43f474a8af375d2a072128da1ad6cae84a244105ff85e94fc
7d8496f639468de7ee998908a91c7e33ef7585fff92e984b210941a1"),
MustHexID("b45a9153c08d002a48090d15d61a7c7dad8c2af85d4ff5bd36ce23a9a11e0709bf8d56
614c7b193bc028c16cbf7f20dfbcc751328b64a924995d47b41e452422"),
MustHexID("057ab3a9e53c7a84b0f3fc586117a525cdd18e313f52a67bf31798d48078e325abe5cfe
e3f6c2533230cb37d0549289d692a29dd400e899b8552d4b928f6f907"),
MustHexID("0ddf663d308791eb92e6bd88a2f8cb45e4f4f35bb16708a0e6ff7f1362aa6a73fedd0a1b
1557fb3365e38e1b79d6918e2fae2788728b70c9ab6b51a3b94a4338"),
MustHexID("f637e07ff50cc1e3731735841c4798411059f2023abcf3885674f3e8032531b0edca50fd

715df6feb489b6177c345374d64f4b07d257a7745de393a107b013a5"),
MustHexID("e24ec7c6eec094f63c7b3239f56d311ec5a3e45bc4e622a1095a65b95eea6fe13e29f3
b6b7a2cbfe40906e3989f17ac834c3102dd0cadaaa26e16ee06d782b72"),
MustHexID("b76ea1a6fd6506ef6e3506a4f1f60ed6287fff8114af6141b2ff13e61242331b54082b023
cfea5b3083354a4fb3f9eb8be01fb4a518f579e731a5d0707291a6b"),
MustHexID("9b53a37950ca8890ee349b325032d7b672cab7eced178d3060137b24ef6b92a439779
22d5bdfb4a3409a2d80128e02f795f9dae6d7d99973ad0e23a2afb8442f"),
},
249: {
MustHexID("675ae65567c3c72c50c73bc0fd4f61f202ea5f93346ca57b551de3411ccc614fad61cb9
035493af47615311b9d44ee7a161972ee4d77c28fe1ec029d01434e6a"),
MustHexID("8eb81408389da88536ae5800392b16ef5109d7ea132c18e9a82928047ecdb502693f6
e4a4cdd18b54296caf561db937185731456c456c98bfe7de0baf0eaa495"),
MustHexID("2adba8b1612a541771cb93a726a38a4b88e97b18eced2593eb7daf82f05a5321ca94a
72cc780c306ff21e551a932fc2c6d791e4681907b5ceab7f084c3fa2944"),
MustHexID("b1b4bfbda514d9b8f35b1c28961da5d5216fe50548f4066f69af3b7666a3b2e06eac646
735e963e5c8f8138a2fb95af15b13b23ff00c6986eccc0efaa8ee6fb4"),
MustHexID("d2139281b289ad0e4d7b4243c4364f5c51aac8b60f4806135de06b12b5b369c9e43a6
eb494eab860d115c15c6fbb8c5a1b0e382972e0e460af395b8385363de7"),
MustHexID("4a693df4b8fc5bdc7cec342c3ed2e228d7c5b4ab7321ddaa6cccbeb45b05a9f1d95766
b4002e6d4791c2deacb8a667aadea6a700da28a3eea810a30395701bbc"),
MustHexID("ab41611195ec3c62bb8cd762ee19fb182d194fd141f4a66780efbef4b07ce916246c022
b841237a3a6b512a93431157edd221e854ed2a259b72e9c5351f44d0c"),
MustHexID("68e8e26099030d10c3c703ae7045c0a48061fb88058d853b3e67880014c449d431101
4da99d617d3150a20f1a3da5e34bf0f14f1c51fe4dd9d58afd222823176"),
MustHexID("3fbcacf546fb129cd70fc48de3b593ba99d3c473798bc309292aca280320e0eacc04442
c914cad5c4cf6950345ba79b0d51302df88285d4e83ee3fe41339eee7"),
MustHexID("1d4a623659f7c8f80b6c3939596afdf42e78f892f682c768ad36eb7bfba402dbf97aea3a
268f3badd8fe7636be216edf3d67ee1e08789ebbc7be625056bd7109"),
MustHexID("a283c474ab09da02bbc96b16317241d0627646fcc427d1fe790b76a7bf1989ced90f92
101a973047ae9940c92720dffbac8eff21df8cae468a50f72f9e159417"),
MustHexID("dbf7e5ad7f87c3dfecae65d87c3039e14ed0bdc56caf00ce81931073e2e16719d74629
5512ff7937a15c3b03603e7c41a4f9df94fcd37bb200dd8f332767e9cb"),
MustHexID("caaa070a26692f64fc77f30d7b5ae980d419b4393a0f442b1c821ef58c0862898b0d22f
74a4f8c5d83069493e3ec0b92f17dc1fe6e4cd437c1ec25039e7ce839"),
MustHexID("874cc8d1213beb65c4e0e1de38ef5d8165235893ac74ab5ea937c885eaab25c8d79da
d0456e9fd3e9450626cac7e107b004478fb59842f067857f39a47cee695"),
MustHexID("d94193f236105010972f5df1b7818b55846592a0445b9cdc4eaed811b8c4c0f7c27dc8
cc9837a4774656d6b34682d6d329d42b6ebb55da1d475c2474dc3dfdf4"),
MustHexID("edd9af6aded4094e9785637c28fccbd3980cbe28e2eb9a411048a23c2ace4bd6b0b70
88a7817997b49a3dd05fc6929ca6c7abbb69438dbdabe65e971d2a794b2"),
},

250: {
MustHexID("53a5bd1215d4ab709ae8fdc2ced50bba320bced78bd9c5dc92947fb402250c914891786db0978c898c058493f86fc68b1c5de8a5cb36336150ac7a88655b6c39"),
MustHexID("b7f79e3ab59f79262623c9ccefc8f01d682323aee56ffbe295437487e9d5acaf556a9c92e1f1c6a9601f2b9eb6b027ae1aeaebac71d61b9b78e88676efd3e1a3"),
MustHexID("d374bf7e8d7ffff69cc00bebff38ef5bc1dcb0a8d51c1a3d70e61ac6b2e2d6617109254b0ac224354dfbf79009fe4239e09020c483cc60c071e00b9238684f30"),
MustHexID("1e1eac1c9add703eb252eb991594f8f5a173255d526a855fab24ae57dc277e055bc3c7a7ae0b45d437c4f47a72d97eb7b126f2ba344ba6c0e14b2c6f27d4b1e6"),
MustHexID("ae28953f63d4bc4e706712a59319c111f5ff8f312584f65d7436b4cd3d14b217b958f8486bad666b4481fe879019fb1f767cf15b3e3e2711efc33b56d460448a"),
MustHexID("934bb1edf9c7a318b82306aca67feb3d6b434421fa275d694f0b4927afd8b1d3935b727fd4ff6e3d012e0c82f1824385174e8c6450ade59c2a43281a4b3446b6"),
MustHexID("9eef3f28f70ce19637519a0916555bf76d26de31312ac656cf9d3e379899ea44e4dd7ffcce923b4f3563f8a00489a34bd6936db0cbb4c959d32c49f017e07d05"),
MustHexID("82200872e8f871c48f1fad13daec6478298099b591bb3dbc4ef6890aa28ebee5860d07d70be62f4c0af85085a90ae8179ee8f937cf37915c67ea73e704b03ee7"),
MustHexID("6c75a5834a08476b7fc37ff3dc2011dc3ea3b36524bad7a6d319b18878fad813c0ba76d1f4555cacd3890c865438c21f0e0aed1f80e0a157e642124c69f43a11"),
MustHexID("995b873742206cb02b736e73a88580c2aacb0bd4a3c97a647b647bcab3f5e03c0e0736520a8b3600da09edf4248991fb01091ec7ff3ec7cdc8a1beae011e7aae"),
MustHexID("c773a056594b5cdef2e850d30891ff0e927c3b1b9c35cd8e8d53a1017001e237468e1ece3ae33d612ca3e6abb0a9169aa352e9dcda358e5af2ad982b577447db"),
MustHexID("2b46a5f6923f475c6be99ec6d134437a6d11f6bb4b4ac6bcd94572fa1092639d1c08aefcb51f0912f0a060f71d4f38ee4da70ecc16010b05dd4a674aab14c3a"),
MustHexID("af6ab501366debbaa0d22e20e9688f32ef6b3b644440580fd78de4fe0e99e2a16eb5636bbae0d1c259df8ddda77b35b9a35cbc36137473e9c68fbc9d203ba842"),
MustHexID("c9f6f2dd1a941926f03f770695bda289859e85fabaf94baaae20b93e5015dc014ba41150176a36a1884adb52f405194693e63b0c464a6891cc9cc1c80d450326"),
MustHexID("5b116f0751526868a909b61a30b0c5282c37df6925cc03ddea556ef0d0602a9595fd6c14d371f8ed7d45d89918a032dcd22be4342a8793d88fdbeb3ca3d75bd7"),
MustHexID("50f3222fb6b82481c7c813b2172e1daea43e2710a443b9c2a57a12bd160dd37e20f87aa968c82ad639af6972185609d47036c0d93b4b7269b74ebd7073221c10"),
},
251: {
MustHexID("9b8f702a62d1bee67bedfeb102eca7f37fa1713e310f0d6651cc0c33ea7c5477575289ccd463e5a2574a00a676a1fdce05658ba447bb9d2827f0ba47b947e894"),
MustHexID("b97532eb83054ed054b4abdf413bb30c00e4205545c93521554dbe77faa3cfaa5bd31ef466a107b0b34a71ec97214c0c83919720142cddac93aa7a3e928d4708"),
MustHexID("2f7a5e952bfb67f2f90b8441b5fadc9ee13b1dcde3afeeb3dd64bf937f86663cc5c55d1fa83952b5422763c7df1b7f2794b751c6be316ebc0beb4942e65ab8c1"),
MustHexID("42c7483781727051a0b3660f14faf39e0d33de5e643702ae933837d036508ab856ce7

eec8ec89c4929a4901256e5233a3d847d5d4893f91bcf21835a9a880fee"),
MustHexID("873bae27bf1dc854408fba94046a53ab0c965cebe1e4e12290806fc62b88deb1f4a47f9e18f78fc0e7913a0c6e42ac4d0fc3a20cea6bc65f0c8a0ca90b67521e"),
MustHexID("a7e3a370bbd761d413f8d209e85886f68bf73d5c3089b2dc6fa42aab1ecb5162635497eed95dee2417f3c9c74a3e76319625c48ead2e963c7de877cd4551f347"),
MustHexID("528597534776a40df2addaaea15b6ff832ce36b9748a265768368f657e76d58569d9f30dbb91e91cf0ae7efe8f402f17aa0ae15f5c55051ba03ba830287f4c42"),
MustHexID("461d8bd4f13c3c09031fdb84f104ed737a52f630261463ce0bdb5704259bab4b737dda688285b8444dbecaecad7f50f835190b38684ced5e90c54219e5adf1bc"),
MustHexID("6ec50c0be3fd232737090fc0111caaf0bb6b18f72be453428087a11a97fd6b52db0344acbf789a689bd4f5f50f79017ea784f8fd6fe723ad6ae675b9e3b13e21"),
MustHexID("12fc5e2f77a83fdcc727b79d8ae7fe6a516881138d3011847ee136b400fed7cfba1f53fd7a9730253c7aa4f39abeacd04f138417ba7fcb0f36cccc3514e0dab6"),
MustHexID("4fdbe75914ccd0bce02101606a1ccf3657ec963e3b3c20239d5fec87673fe446d649b4f15f1fe1a40e6cfbd446dda2d31d40bb602b1093b8fcd5f139ba0eb46a"),
MustHexID("3753668a0f6281e425ea69b52cb2d17ab97afbe6eb84cf5d25425bc5e53009388857640668fadd7c110721e6047c9697803bd8a6487b43bb343bfa32ebf24039"),
MustHexID("2e81b16346637dec4410fd88e527346145b9c0a849dbf2628049ac7dae016c8f4305649d5659ec77f1e8a0fac0db457b6080547226f06283598e3740ad94849a"),
MustHexID("802c3cc27f91c89213223d758f8d2ecd41135b357b6d698f24d811cdf113033a81c38e0bdff574a5c005b00a8c193dc2531f8c1fa05fa60acf0ab6f2858af09f"),
MustHexID("fcc9a2e1ac3667026ff16192876d1813bb75abdbf39b929a92863012fe8b1d890badea7a0de36274d5c1eb1e8f975785532c50d80fd44b1a4b692f437303393f"),
MustHexID("6d8b3efb461151dd4f6de809b62726f5b89e9b38e9ba1391967f61cde844f7528fecf821b74049207cee5a527096b31f3ad623928cd3ce51d926fa345a6b2951"),
},
252: {
MustHexID("f1ae93157cc48c2075dd5868fbf523e79e06caf4b8198f352f6e526680b78ff4227263de92612f7d63472bd09367bb92a636fff16fe46ccf41614f7a72495c2a"),
MustHexID("587f482d111b239c27c0cb89b51dd5d574db8efd8de14a2e6a1400c54d4567e77c65f89c1da52841212080b91604104768350276b6682f2f961cdaf4039581c7"),
MustHexID("e3f88274d35cefdaabdf205afe0e80e936cc982b8e3e47a84ce664c413b29016a4fb4f3a3ebae0a2f79671f8323661ed462bf4390af94c424dc8ace0c301b90f"),
MustHexID("0ddc736077da9a12ba410dc5ea63cbcbe7659dd08596485b2bff3435221f82c10d263efd9af938e128464be64a178b7cd22e19f400d5802f4c9df54bf89f2619"),
MustHexID("784aa34d833c6ce63fcc1279630113c3272e82c4ae8c126c5a52a88ac461b6baeed4244e607b05dc14e5b2f41c70a273c3804dea237f14f7a1e546f6d1309d14"),
MustHexID("f253a2c354ee0e27cfcae786d726753d4ad24be6516b279a936195a487de4a59dbc296accf20463749ff55293263ed8c1b6365eecb248d44e75e9741c0d18205"),
MustHexID("a1910b80357b3ad9b4593e0628922939614dc9056a5fbf477279c8b2c1d0b4b31d89a0c09d0d41f795271d14d3360ef08a3f821e65e7e1f56c07a36afe49c7c5"),
MustHexID("f1168552c2efe541160f0909b0b4a9d6aeedcf595cdf0e9b165c97e3e197471a1ee6320

e93389edfba28af6eaf10de98597ad56e7ab1b504ed762451996c3b98"),
MustHexID("b0c8e5d2c8634a7930e1a6fd082e448c6cf9d2d8b7293558b59238815a4df926c286bf
297d2049f14e8296a6eb3256af614ec1812c4f2bbe807673b58bf14c8c"),
MustHexID("0fb346076396a38badc342df3679b55bd7f40a609ab103411fe45082c01f12ea016729
e95914b2b5540e987ff5c9b133e85862648e7f36abdfd23100d248d234"),
MustHexID("f736e0cc83417feaa280d9483f5d4d72d1b036cd0c6d9cbdeb8ac35ceb2604780de46d
ddaa32a378474e1d5ccdf79b373331c30c7911ade2ae32f98832e5de1f"),
MustHexID("8b02991457602f42b38b342d3f2259ae4100c354b3843885f7e4e07bd644f64dab94bb
7f38a3915f8b7f11d8e3f81c28e07a0078cf79d7397e38a7b7e0c857e2"),
MustHexID("9221d9f04a8a184993d12baa91116692bb685f887671302999d69300ad103eb2d2c75
a09d8979404c6dd28f12362f58a1a43619c493d9108fd47588a23ce5824"),
MustHexID("652797801744dada833fff207d67484742eea6835d695925f3e618d71b68ec3c65bdd8
5b4302b2cdcb835ad3f94fd00d8da07e570b41bc0d2bcf69a8de1b3284"),
MustHexID("d84f06fe64debc4cd0625e36d19b99014b6218375262cc2209202bdbafd7dffcc4e34ce
6398e182e02fd8faeed622c3e175545864902dfd3d1ac57647cddf4c6"),
MustHexID("d0ed87b294f38f1d741eb601020eeec30ac16331d05880fe27868f1e454446de367d74
57b41c79e202eaf9525b029e4f1d7e17d85a55f83a557c005c68d7328a"),
},
253: {
MustHexID("ad4485e386e3cc7c7310366a7c38fb810b8896c0d52e55944bfd320ca294e7912d6c5
3c0a0cf85e7ce226e92491d60430e86f8f15cda0161ed71893fb4a9e3a1"),
MustHexID("36d0e7e5b7734f98c6183eeeb8ac5130a85e910a925311a19c4941b1290f945d4fc399
6b12ef4966960b6fa0fb29b1604f83a0f81bd5fd6398d2e1a22e46af0c"),
MustHexID("7d307d8acb4a561afa23bdf0bd945d35c90245e26345ec3a1f9f7df354222a7cdcb8133
9c9ed6744526c27a1a0c8d10857e98df942fa433602facac71ac68a31"),
MustHexID("d97bf55f88c83fae36232661af115d66ca600fc4bd6d1fb35ff9bb4dad674c02cf8c8d05f
317525b5522250db58bb1ecafb7157392bf5aa61b178c61f098d995"),
MustHexID("7045d678f1f9eb7a4613764d17bd5698796494d0bf977b16f2dbc272b8a0f7858a6080
5c022fc3d1fe4f31c37e63cdaca0416c0d053ef48a815f8b19121605e0"),
MustHexID("14e1f21418d445748de2a95cd9a8c3b15b506f86a0acabd8af44bb968ce39885b19c88
22af61b3dd58a34d1f265baec30e3ae56149dc7d2aa4a538f7319f69c8"),
MustHexID("b9453d78281b66a4eac95a1546017111eaaa5f92a65d0de10b1122940e92b319728a
24edf4dec6acc412321b1c95266d39c7b3a5d265c629c3e49a65fb022c09"),
MustHexID("e8a49248419e3824a00d86af422f22f7366e2d4922b304b7169937616a01d9d6fa5abf
5cc01061a352dc866f48e1fa2240dbb453d872b1d7be62bdfc1d5e248c"),
MustHexID("bebcff24b52362f30e0589ee573ce2d86f073d58d18e6852a592fa86ceb1a6c9b96d7fb
9ec7ed1ed98a51b6743039e780279f6bb49d0a04327ac7a182d9a56f6"),
MustHexID("d0835e5a4291db249b8d2fca9f503049988180c7d247bedaa2cf3a1bad0a76709360a8
5d4f9a1423b2cbc82bb4d94b47c0cde20afc430224834c49fe312a9ae3"),
MustHexID("6b087fe2a2da5e4f0b0f4777598a4a7fb66bf77dbd5bfc44e8a7eaa432ab585a6e22689
1f56a7d4f5ed11a7c57b90f1661bba1059590ca4267a35801c2802913"),
MustHexID("d901e5bde52d1a0f4ddf010a686a53974cdae4ebe5c6551b3c37d6b6d635d38d5b0e5f

80bc0186a2c7809dbf3a42870dd09643e68d32db896c6da8ba734579e7"),
MustHexID("96419fb80efae4b674402bb969ebaab86c1274f29a83a311e24516d36cdf148fe21754
d46c97688cdd7468f24c08b13e4727c29263393638a3b37b99ff60ebca"),
MustHexID("7b9c1889ae916a5d5abcdfb0aaedcc9c6f9eb1c1a4f68d0c2d034fe79ac610ce917c3ab
c670744150fa891bfcd8ab14fed6983fca964de920aa393fa7b326748"),
MustHexID("7a369b2b8962cc4c65900be046482fbf7c14f98a135bbbae25152c82ad168fb2097b3d
1429197cf46d3ce9fdeb64808f908a489cc6019725db040060fdfe5405"),
MustHexID("47bcae48288da5ecc7f5058dfa07cf14d89d06d6e449cb946e237aa6652ea050d9f5a2
4a65efdc0013ccf232bf88670979eddef249b054f63f38da9d7796dbd8"),
},
254: {
MustHexID("099739d7abc8abd38ecc7a816c521a1168a4dbd359fa7212a5123ab583ffa1cf485a5fe
d219575d6475dbcdd541638b2d3631a6c7fce7474e7fe3cba1d4d5853"),
MustHexID("c2b01603b088a7182d0cf7ef29fb2b04c70acb320fccf78526bf9472e10c74ee70b3fcfa6
f4b11d167bd7d3bc4d936b660f2c9bff934793d97cb21750e7c3d31"),
MustHexID("20e4d8f45f2f863e94b45548c1ef22a11f7d36f263e4f8623761e05a64c4572379b000a
52211751e2561b0f14f4fc92dd4130410c8ccc71eb4f0e95a700d4ca9"),
MustHexID("27f4a16cc085e72d86e25c98bd2eca173eaaee7565c78ec5a52e9e12b2211f35de81b
5b45e9195de2ebfe29106742c59112b951a04eb7ae48822911fc1f9389e"),
MustHexID("55db5ee7d98e7f0b1c3b9d5be6f2bc619a1b86c3cdd513160ad4dcf267037a5fffad527
ac15d50aeb32c59c13d1d4c1e567ebbf4de0d25236130c8361f9aac63"),
MustHexID("883df308b0130fc928a8559fe50667a0fff80493bc09685d18213b2db241a3ad11310ed
86b0ef662b3ce21fc3d9aa7f3fc24b8d9afe17c7407e9afd3345ae548"),
MustHexID("c7af968cc9bc8200c3ee1a387405f7563be1dce6710a3439f42ea40657d0eae9d2b3c1
6c42d779605351fcdece4da637b9804e60ca08cfb89aec32c197beffa6"),
MustHexID("3e66f2b788e3ff1d04106b80597915cd7afa06c405a7ae026556b6e583dca8e05cfbab5
039bb9a1b5d06083ffe8de5780b1775550e7218f5e98624bf7af9a0a8"),
MustHexID("4fc7f53764de3337fdaec0a711d35d3a923e72fa65025444d12230b3552ed43d9b2d1a
d08ccb11f2d50c58809e6dd74dde910e195294fca3b47ae5a3967cc479"),
MustHexID("bafdfdcf6ccaa989436752fa97c77477b6baa7deb374b16c095492c529eb133e8e2f99e
1977012b64767b9d34b2cf6d2048ed489bd822b5139b523f6a423167b"),
MustHexID("7f5d78008a4312fe059104ce80202c82b8915c2eb4411c6b812b16f7642e57c00f2c94
25121f5cbac4257fe0b3e81ef5dea97ea2dbaa98f6a8b6fd4d1e5980bb"),
MustHexID("598c37fe78f922751a052f463aeb0cb0bc7f52b7c2a4cf2da72ec0931c7c32175d4165d
0f8998f7320e87324ac3311c03f9382a5385c55f0407b7a66b2acd864"),
MustHexID("f758c4136e1c148777a7f3275a76e2db0b2b04066fd738554ec398c1c6cc9fb47e14a3
b4c87bd47deaeab3ffd2110514c3855685a374794daff87b605b27ee2e"),
MustHexID("0307bb9e4fd865a49dcf1fe4333d1b944547db650ab580af0b33e53c4fef6c789531110f
ac801bbcbce21fc4d6f61b6d5b24abdf5b22e3030646d579f6dca9c2"),
MustHexID("82504b6eb49bb2c0f91a7006ce9cefdbaf6df38706198502c2e06601091fc9dc91e4f15
db3410d45c6af355bc270b0f268d3dff560f956985c7332d4b10bd1ed"),
MustHexID("b39b5b677b45944ceebe76e76d1f051de2f2a0ec7b0d650da52135743e66a9a5dba45

f638258f9a7545d9a790c7fe6d3fdf82c25425c7887323e45d27d06c057"),
},
255: {
MustHexID("5c4d58d46e055dd1f093f81ee60a675e1f02f54da6206720adee4dccef9b67a31efc5c2a2949c31a04ee31beadc79aba10da31440a1f9ff2a24093c63c36d784"),
MustHexID("ea72161ffdd4b1e124c7b93b0684805f4c4b58d617ed498b37a145c670dbc2e04976f8785583d9c805ffbf343c31d492d79f841652bbbd01b61ed85640b23495"),
MustHexID("51caa1d93352d47a8e531692a3612adac1e8ac68d0a200d086c1c57ae1e1a91aa285ab242e8c52ef9d7afe374c9485b122ae815f1707b875569d0433c1c3ce85"),
MustHexID("c08397d5751b47bd3da044b908be0fb0e510d3149574dff7aeab33749b023bb171b5769990fe17469dbebc100bc150e798aeda426a2dcc766699a225fddd75c6"),
MustHexID("0222c1c194b749736e593f937fad67ee348ac57287a15c7e42877aa38a9b87732a408bca370f812efd0eedbff13e6d5b854bf3ba1dec431a796ed47f32552b09"),
MustHexID("03d859cd46ef02d9bfad5268461a6955426845eef4126de6be0fa4e8d7e0727ba2385b78f1a883a8239e95ebb814f2af8379632c7d5b100688eebc5841209582"),
MustHexID("64d5004b7e043c39ff0bd10cb20094c287721d5251715884c280a612b494b3e9e1c64ba6f67614994c7d969a0d0c0295d107d53fc225d47c44c4b82852d6f960"),
MustHexID("b0a5eefb2dab6f786670f35bf9641eefe6dd87fd3f1362bcab4aaa792903500ab23d88fae68411372e0813b057535a601d46e454323745a948017f6063a47b1f"),
MustHexID("0cc6df0a3433d448b5684d2a3ffa9d1a825388177a18f44ad0008c7bd7702f1ec0fc38b83506f7de689c3b6ecb552599927e29699eed6bb867ff08f80068b287"),
MustHexID("50772f7b8c03a4e153355fbbf79c8a80cf32af656ff0c7873c99911099d04a0dae0674706c357e0145ad017a0ade65e6052cb1b0d574fcd6f67da3eee0ace66b"),
MustHexID("1ae37829c9ef41f8b508b82259ebac76b1ed900d7a45c08b7970f25d2d48ddd1829e2f11423a18749940b6dab8598c6e416cef0efd47e46e51f29a0bc65b37cd"),
MustHexID("ba973cab31c2af091fc1644a93527d62b2394999e2b6ccbf158dd5ab9796a43d408786f1803ef4e29debfeb62fce2b6caa5ab2b24d1549c822a11c40c2856665"),
MustHexID("bc413ad270dd6ea25bddba78f3298b03b8ba6f8608ac03d06007d4116fa78ef5a0cfe8c80155089382fc7a193243ee5500082660cb5d7793f60f2d7d18650964"),
MustHexID("5a6a9ef07634d9eec3baa87c997b529b92652afa11473dfee41ef7037d5c06e0ddb9fe842364462d79dd31cff8a59a1b8d5bc2b810dea1d4cbbd3beb80ecec83"),
MustHexID("f492c6ee2696d5f682f7f537757e52744c2ae560f1090a07024609e903d334e9e174fc01609c5a229ddbcac36c9d21adaf6457dab38a25bfd44f2f0ee4277998"),
MustHexID("459e4db99298cb0467a90acee6888b08bb857450deac11015cced5104853be5adce5b69c740968bc7f931495d671a70cad9f48546d7cd203357fe9af0e8d2164"),
},
256: {
MustHexID("a8593af8a4aef7b806b5197612017951bac8845a1917ca9a6a15dd6086d608505144990b245785c4cd2d67a295701c7aac2aa18823fb0033987284b019656268"),
MustHexID("d2eebef914928c3aad77fc1b2a495f52d2294acf5edaa7d8a530b540f094b861a68fe8348a46a7c302f08ab609d85912a4968eacfea0740847b29421b4795d9e"),
MustHexID("b14bfcb31495f32b650b63cf7d08492e3e29071fdc73cf2da0da48d4b191a70ba1a65f4

```
2ad8c343206101f00f8a48e8db4b08bf3f622c0853e7323b250835b91"),
MustHexID("7feaee0d818c03eb30e4e0bf03ade0f3c21ca38e938a761aa1781cf70bda8cc5cd631a
6cc53dd44f1d4a6d3e2dae6513c6c66ee50cb2f0e9ad6f7e319b309fd9"),
MustHexID("4ca3b657b139311db8d583c25dd5963005e46689e1317620496cc64129c7f3e528708
20e0ec7941d28809311df6db8a2867bbd4f235b4248af24d7a9c22d1232"),
MustHexID("1181defb1d16851d42dd951d84424d6bd1479137f587fa184d5a8152be6b6b16ed08b
cdb2c2ed8539bcde98c80c432875f9f724737c316a2bd385a39d3cab1d8"),
MustHexID("d9dd818769fa0c3ec9f553c759b92476f082817252a04a47dc1777740b1731d280058c
66f982812f173a294acf4944a85ba08346e2de153ba3ba41ce8a62cb64"),
MustHexID("bd7c4f8a9e770aa915c771b15e107ca123d838762da0d3ffc53aa6b53e9cd076cffc534
ec4d2e4c334c683f1f5ea72e0e123f6c261915ed5b58ac1b59f003d88"),
MustHexID("3dd5739c73649d510456a70e9d6b46a855864a4a3f744e088fd8c8da11b18e4c9b5f2d
7da50b1c147b2bae5ca9609ae01f7a3cdea9dce34f80a91d29cd82f918"),
MustHexID("f0d7df1efc439b4bcc0b762118c1cfa99b2a6143a9f4b10e3c9465125f4c9fca4ab88a25
04169bbcad65492cf2f50da9dd5d077c39574a944f94d8246529066b"),
MustHexID("dd598b9ba441448e5fb1a6ec6c5f5aa9605bad6e223297c729b1705d11d05f6bfd3d41
988b694681ae69bb03b9a08bff4beab5596503d12a39bffb5cd6e94c7c"),
MustHexID("3fce284ac97e567aebae681b15b7a2b6df9d873945536335883e4bbc26460c0643705
37f323fd1ada828ea43154992d14ac0cec0940a2bd2a3f42ec156d60c83"),
MustHexID("7c8dfa8c1311cb14fb29a8ac11bca23ecc115e56d9fcf7b7ac1db9066aa4eb39f8b1dabf
46e192a65be95ebfb4e839b5ab4533fef414921825e996b210dd53bd"),
MustHexID("cafa6934f82120456620573d7f801390ed5e16ed619613a37e409e44ab355ef755e835
65a913b48a9466db786f8d4fbd590bfec474c2524d4a2608d4eafd6abd"),
MustHexID("9d16600d0dd310d77045769fed2cb427f32db88cd57d86e49390c2ba8a9698cfa856f7
75be2013237226e7bf47b248871cf865d23015937d1edeb20db5e3e760"),
MustHexID("17be6b6ba54199b1d80eff866d348ea11d8a4b341d63ad9a6681d3ef8a43853ac564d
153eb2a8737f0afc9ab320f6f95c55aa11aaa13bbb1ff422fd16bdf8188"),
},
},
}

type preminedTestnet struct {
target    NodeID
targetSha common.Hash // sha3(target)
dists     [hashBits + 1][]NodeID
}

func (tn *preminedTestnet) findnode(toid NodeID, toaddr *net.UDPAddr, target NodeID) ([]*Node,
error) {
// current log distance is encoded in port number
// fmt.Println("findnode query at dist", toaddr.Port)
if toaddr.Port == 0 {
```

```
panic("query to node at distance 0")
}
next := uint16(toaddr.Port) - 1
var result []*Node
for i, id := range tn.dists[toaddr.Port] {
result = append(result, NewNode(id, net.ParseIP("127.0.0.1"), next, uint16(i)))
}
return result, nil
}

func (*preminedTestnet) close()                                 {}
func (*preminedTestnet) waitping(from NodeID) error             { return nil }
func (*preminedTestnet) ping(toid NodeID, toaddr *net.UDPAddr) error { return nil }

// mine generates a testnet struct literal with nodes at
// various distances to the given target.
func (n *preminedTestnet) mine(target NodeID) {
n.target = target
n.targetSha = crypto.Keccak256Hash(n.target[:])
found := 0
for found < bucketSize*10 {
k := newkey()
id := PubkeyID(&k.PublicKey)
sha := crypto.Keccak256Hash(id[:])
ld := logdist(n.targetSha, sha)
if len(n.dists[ld]) < bucketSize {
n.dists[ld] = append(n.dists[ld], id)
fmt.Println("found ID with ld", ld)
found++
}
}
fmt.Println("&preminedTestnet{")
fmt.Printf("target: %#v,\n", n.target)
fmt.Printf("targetSha: %#v,\n", n.targetSha)
fmt.Printf("dists: [%d][]NodeID{\n", len(n.dists))
for ld, ns := range n.dists {
if len(ns) == 0 {
continue
}
fmt.Printf("%d: []NodeID{\n", ld)
for _, n := range ns {
fmt.Printf("MustHexID(\"%x\"),\n", n[:])
```

```
}
fmt.Println("},")
}
fmt.Println("},")
fmt.Println("}")
}

func hasDuplicates(slice []*Node) bool {
seen := make(map[NodeID]bool)
for i, e := range slice {
if e == nil {
panic(fmt.Sprintf("nil *Node at %d", i))
}
if seen[e.ID] {
return true
}
seen[e.ID] = true
}
return false
}

func sortedByDistanceTo(distbase common.Hash, slice []*Node) bool {
var last common.Hash
for i, e := range slice {
if i > 0 && distcmp(distbase, e.sha, last) < 0 {
return false
}
last = e.sha
}
return true
}

func contains(ns []*Node, id NodeID) bool {
for _, n := range ns {
if n.ID == id {
return true
}
}
return false
}

// gen wraps quick.Value so it's easier to use.
```

```
// it generates a random value of the given value's type.
func gen(typ interface{}, rand *rand.Rand) interface{} {
v, ok := quick.Value(reflect.TypeOf(typ), rand)
if !ok {
panic(fmt.Sprintf("couldn't generate random value of type %T", typ))
}
return v.Interface()
}

func newkey() *ecdsa.PrivateKey {
key, err := crypto.GenerateKey()
if err != nil {
panic("couldn't generate key: " + err.Error())
}
return key
}

90:F:\git\coin\ethereum\go-ethereum\p2p\discover\udp.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discover

import (
"bytes"
"container/list"
"crypto/ecdsa"
"errors"
"fmt"
"net"
"time"

"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/p2p/nat"
"github.com/ethereum/go-ethereum/p2p/netutil"
"github.com/ethereum/go-ethereum/rlp"
)

const Version = 4

// Errors
var (
```

```
errPacketTooSmall   = errors.New("too small")
errBadHash          = errors.New("bad hash")
errExpired          = errors.New("expired")
errUnsolicitedReply = errors.New("unsolicited reply")
errUnknownNode      = errors.New("unknown node")
errTimeout          = errors.New("RPC timeout")
errClockWarp        = errors.New("reply deadline too far in the future")
errClosed           = errors.New("socket closed")
)

// Timeouts
const (
respTimeout = 500 * time.Millisecond
sendTimeout = 500 * time.Millisecond
expiration  = 20 * time.Second

ntpFailureThreshold = 32               // Continuous timeouts after which to check NTP
ntpWarningCooldown  = 10 * time.Minute // Minimum amount of time to pass before repeating
NTP warning
driftThreshold      = 10 * time.Second // Allowed clock drift before warning user
)

// RPC packet types
const (
pingPacket = iota + 1 // zero is 'reserved'
pongPacket
findnodePacket
neighborsPacket
)

// RPC request structures
type (
ping struct {
Version    uint
From, To   rpcEndpoint
Expiration uint64
// Ignore additional fields (for forward compatibility).
Rest []rlp.RawValue `rlp:"tail"`
}

// pong is the reply to ping.
pong struct {
```

```
// This field should mirror the UDP envelope address
// of the ping packet, which provides a way to discover the
// the external address (after NAT).
To rpcEndpoint

ReplyTok   []byte // This contains the hash of the ping packet.
Expiration uint64 // Absolute timestamp at which the packet becomes invalid.
// Ignore additional fields (for forward compatibility).
Rest []rlp.RawValue `rlp:"tail"`
}

// findnode is a query for nodes close to the given target.
findnode struct {
Target     NodeID // doesn't need to be an actual public key
Expiration uint64
// Ignore additional fields (for forward compatibility).
Rest []rlp.RawValue `rlp:"tail"`
}

// reply to findnode
neighbors struct {
Nodes      []rpcNode
Expiration uint64
// Ignore additional fields (for forward compatibility).
Rest []rlp.RawValue `rlp:"tail"`
}

rpcNode struct {
IP  net.IP // len 4 for IPv4 or 16 for IPv6
UDP uint16 // for discovery protocol
TCP uint16 // for RLPx protocol
ID  NodeID
}

rpcEndpoint struct {
IP  net.IP // len 4 for IPv4 or 16 for IPv6
UDP uint16 // for discovery protocol
TCP uint16 // for RLPx protocol
}
)

func makeEndpoint(addr *net.UDPAddr, tcpPort uint16) rpcEndpoint {
```

```
ip := addr.IP.To4()
if ip == nil {
ip = addr.IP.To16()
}
return rpcEndpoint{IP: ip, UDP: uint16(addr.Port), TCP: tcpPort}
}

func (t *udp) nodeFromRPC(sender *net.UDPAddr, rn rpcNode) (*Node, error) {
if rn.UDP <= 1024 {
return nil, errors.New("low port")
}
if err := netutil.CheckRelayIP(sender.IP, rn.IP); err != nil {
return nil, err
}
if t.netrestrict != nil && !t.netrestrict.Contains(rn.IP) {
return nil, errors.New("not contained in netrestrict whitelist")
}
n := NewNode(rn.ID, rn.IP, rn.UDP, rn.TCP)
err := n.validateComplete()
return n, err
}

func nodeToRPC(n *Node) rpcNode {
return rpcNode{ID: n.ID, IP: n.IP, UDP: n.UDP, TCP: n.TCP}
}

type packet interface {
handle(t *udp, from *net.UDPAddr, fromID NodeID, mac []byte) error
name() string
}

type conn interface {
ReadFromUDP(b []byte) (n int, addr *net.UDPAddr, err error)
WriteToUDP(b []byte, addr *net.UDPAddr) (n int, err error)
Close() error
LocalAddr() net.Addr
}

// udp implements the RPC protocol.
type udp struct {
conn        conn
netrestrict *netutil.Netlist
```

```
priv        *ecdsa.PrivateKey
ourEndpoint rpcEndpoint

addpending chan *pending
gotreply   chan reply

closing chan struct{}
nat     nat.Interface

*Table
}

// pending represents a pending reply.
//
// some implementations of the protocol wish to send more than one
// reply packet to findnode. in general, any neighbors packet cannot
// be matched up with a specific findnode packet.
//
// our implementation handles this by storing a callback function for
// each pending reply. incoming packets from a node are dispatched
// to all the callback functions for that node.
type pending struct {
// these fields must match in the reply.
from  NodeID
ptype byte

// time when the request must complete
deadline time.Time

// callback is called when a matching reply arrives. if it returns
// true, the callback is removed from the pending reply queue.
// if it returns false, the reply is considered incomplete and
// the callback will be invoked again for the next matching reply.
callback func(resp interface{}) (done bool)

// errc receives nil when the callback indicates completion or an
// error if no further reply is received within the timeout.
errc chan<- error
}

type reply struct {
from  NodeID
```

```
ptype byte
data  interface{}
// loop indicates whether there was
// a matching request by sending on this channel.
matched chan<- bool
}

// ListenUDP returns a new table that listens for UDP packets on laddr.
func ListenUDP(priv *ecdsa.PrivateKey, laddr string, natm nat.Interface, nodeDBPath string,
netrestrict *netutil.Netlist) (*Table, error) {
addr, err := net.ResolveUDPAddr("udp", laddr)
if err != nil {
return nil, err
}
conn, err := net.ListenUDP("udp", addr)
if err != nil {
return nil, err
}
tab, _, err := newUDP(priv, conn, natm, nodeDBPath, netrestrict)
if err != nil {
return nil, err
}
log.Info("UDP listener up", "self", tab.self)
return tab, nil
}

func newUDP(priv *ecdsa.PrivateKey, c conn, natm nat.Interface, nodeDBPath string, netrestrict
*netutil.Netlist) (*Table, *udp, error) {
udp := &udp{
conn:        c,
priv:        priv,
netrestrict: netrestrict,
closing:     make(chan struct{}),
gotreply:    make(chan reply),
addpending:  make(chan *pending),
}
realaddr := c.LocalAddr().(*net.UDPAddr)
if natm != nil {
if !realaddr.IP.IsLoopback() {
go nat.Map(natm, udp.closing, "udp", realaddr.Port, realaddr.Port, "ethereum discovery")
}
// TODO: react to external IP changes over time.
```

```
if ext, err := natm.ExternalIP(); err == nil {
realaddr = &net.UDPAddr{IP: ext, Port: realaddr.Port}
}
}
// TODO: separate TCP port
udp.ourEndpoint = makeEndpoint(realaddr, uint16(realaddr.Port))
tab, err := newTable(udp, PubkeyID(&priv.PublicKey), realaddr, nodeDBPath)
if err != nil {
return nil, nil, err
}
udp.Table = tab

go udp.loop()
go udp.readLoop()
return udp.Table, udp, nil
}

func (t *udp) close() {
close(t.closing)
t.conn.Close()
// TODO: wait for the loops to end.
}

// ping sends a ping message to the given node and waits for a reply.
func (t *udp) ping(toid NodeID, toaddr *net.UDPAddr) error {
// TODO: maybe check for ReplyTo field in callback to measure RTT
errc := t.pending(toid, pongPacket, func(interface{}) bool { return true })
t.send(toaddr, pingPacket, &ping{
Version:    Version,
From:       t.ourEndpoint,
To:         makeEndpoint(toaddr, 0), // TODO: maybe use known TCP port from DB
Expiration: uint64(time.Now().Add(expiration).Unix()),
})
return <-errc
}

func (t *udp) waitping(from NodeID) error {
return <-t.pending(from, pingPacket, func(interface{}) bool { return true })
}

// findnode sends a findnode request to the given node and waits until
// the node has sent up to k neighbors.
```

```
func (t *udp) findnode(toid NodeID, toaddr *net.UDPAddr, target NodeID) ([]*Node, error) {
nodes := make([]*Node, 0, bucketSize)
nreceived := 0
errc := t.pending(toid, neighborsPacket, func(r interface{}) bool {
reply := r.(*neighbors)
for _, rn := range reply.Nodes {
nreceived++
n, err := t.nodeFromRPC(toaddr, rn)
if err != nil {
log.Trace("Invalid neighbor node received", "ip", rn.IP, "addr", toaddr, "err", err)
continue
}
nodes = append(nodes, n)
}
return nreceived >= bucketSize
})
t.send(toaddr, findnodePacket, &findnode{
Target:     target,
Expiration: uint64(time.Now().Add(expiration).Unix()),
})
err := <-errc
return nodes, err
}

// pending adds a reply callback to the pending reply queue.
// see the documentation of type pending for a detailed explanation.
func (t *udp) pending(id NodeID, ptype byte, callback func(interface{}) bool) <-chan error {
ch := make(chan error, 1)
p := &pending{from: id, ptype: ptype, callback: callback, errc: ch}
select {
case t.addpending <- p:
// loop will handle it
case <-t.closing:
ch <- errClosed
}
return ch
}

func (t *udp) handleReply(from NodeID, ptype byte, req packet) bool {
matched := make(chan bool, 1)
select {
case t.gotreply <- reply{from, ptype, req, matched}:
```

```
// loop will handle it
return <-matched
case <-t.closing:
return false
}
}

// loop runs in its own goroutine. it keeps track of
// the refresh timer and the pending reply queue.
func (t *udp) loop() {
var (
plist        = list.New()
timeout      = time.NewTimer(0)
nextTimeout  *pending // head of plist when timeout was last reset
contTimeouts = 0      // number of continuous timeouts to do NTP checks
ntpWarnTime  = time.Unix(0, 0)
)
<-timeout.C // ignore first timeout
defer timeout.Stop()

resetTimeout := func() {
if plist.Front() == nil || nextTimeout == plist.Front().Value {
return
}
// Start the timer so it fires when the next pending reply has expired.
now := time.Now()
for el := plist.Front(); el != nil; el = el.Next() {
nextTimeout = el.Value.(*pending)
if dist := nextTimeout.deadline.Sub(now); dist < 2*respTimeout {
timeout.Reset(dist)
return
}
// Remove pending replies whose deadline is too far in the
// future. These can occur if the system clock jumped
// backwards after the deadline was assigned.
nextTimeout.errc <- errClockWarp
plist.Remove(el)
}
nextTimeout = nil
timeout.Stop()
}
```

```
for {
resetTimeout()

select {
case <-t.closing:
for el := plist.Front(); el != nil; el = el.Next() {
el.Value.(*pending).errc <- errClosed
}
return

case p := <-t.addpending:
p.deadline = time.Now().Add(respTimeout)
plist.PushBack(p)

case r := <-t.gotreply:
var matched bool
for el := plist.Front(); el != nil; el = el.Next() {
p := el.Value.(*pending)
if p.from == r.from && p.ptype == r.ptype {
matched = true
// Remove the matcher if its callback indicates
// that all replies have been received. This is
// required for packet types that expect multiple
// reply packets.
if p.callback(r.data) {
p.errc <- nil
plist.Remove(el)
}
// Reset the continuous timeout counter (time drift detection)
contTimeouts = 0
}
}
r.matched <- matched

case now := <-timeout.C:
nextTimeout = nil

// Notify and remove callbacks whose deadline is in the past.
for el := plist.Front(); el != nil; el = el.Next() {
p := el.Value.(*pending)
if now.After(p.deadline) || now.Equal(p.deadline) {
p.errc <- errTimeout
```

```
plist.Remove(el)
contTimeouts++
}
}
// If we've accumulated too many timeouts, do an NTP time sync check
if contTimeouts > ntpFailureThreshold {
if time.Since(ntpWarnTime) >= ntpWarningCooldown {
ntpWarnTime = time.Now()
go checkClockDrift()
}
contTimeouts = 0
}
}
}
}

const (
macSize  = 256 / 8
sigSize  = 520 / 8
headSize = macSize + sigSize // space of packet frame data
)

var (
headSpace = make([]byte, headSize)

// Neighbors replies are sent across multiple packets to
// stay below the 1280 byte limit. We compute the maximum number
// of entries by stuffing a packet until it grows too large.
maxNeighbors int
)

func init() {
p := neighbors{Expiration: ^uint64(0)}
maxSizeNode := rpcNode{IP: make(net.IP, 16), UDP: ^uint16(0), TCP: ^uint16(0)}
for n := 0; ; n++ {
p.Nodes = append(p.Nodes, maxSizeNode)
size, _, err := rlp.EncodeToReader(p)
if err != nil {
// If this ever happens, it will be caught by the unit tests.
panic("cannot encode: " + err.Error())
}
if headSize+size+1 >= 1280 {
```

```
maxNeighbors = n
break
}
}
}

func (t *udp) send(toaddr *net.UDPAddr, ptype byte, req packet) error {
packet, err := encodePacket(t.priv, ptype, req)
if err != nil {
return err
}
_, err = t.conn.WriteToUDP(packet, toaddr)
log.Trace(">> "+req.name(), "addr", toaddr, "err", err)
return err
}

func encodePacket(priv *ecdsa.PrivateKey, ptype byte, req interface{}) ([]byte, error) {
b := new(bytes.Buffer)
b.Write(headSpace)
b.WriteByte(ptype)
if err := rlp.Encode(b, req); err != nil {
log.Error("Can't encode discv4 packet", "err", err)
return nil, err
}
packet := b.Bytes()
sig, err := crypto.Sign(crypto.Keccak256(packet[headSize:]), priv)
if err != nil {
log.Error("Can't sign discv4 packet", "err", err)
return nil, err
}
copy(packet[macSize:], sig)
// add the hash to the front. Note: this doesn't protect the
// packet in any way. Our public key will be part of this hash in
// The future.
copy(packet, crypto.Keccak256(packet[macSize:]))
return packet, nil
}

// readLoop runs in its own goroutine. it handles incoming UDP packets.
func (t *udp) readLoop() {
defer t.conn.Close()
// Discovery packets are defined to be no larger than 1280 bytes.
```

```
// Packets larger than this size will be cut at the end and treated
// as invalid because their hash won't match.
buf := make([]byte, 1280)
for {
nbytes, from, err := t.conn.ReadFromUDP(buf)
if netutil.IsTemporaryError(err) {
// Ignore temporary read errors.
log.Debug("Temporary UDP read error", "err", err)
continue
} else if err != nil {
// Shut down the loop for permament errors.
log.Debug("UDP read error", "err", err)
return
}
t.handlePacket(from, buf[:nbytes])
}
}

func (t *udp) handlePacket(from *net.UDPAddr, buf []byte) error {
packet, fromID, hash, err := decodePacket(buf)
if err != nil {
log.Debug("Bad discv4 packet", "addr", from, "err", err)
return err
}
err = packet.handle(t, from, fromID, hash)
log.Trace("<< "+packet.name(), "addr", from, "err", err)
return err
}

func decodePacket(buf []byte) (packet, NodeID, []byte, error) {
if len(buf) < headSize+1 {
return nil, NodeID{}, nil, errPacketTooSmall
}
hash, sig, sigdata := buf[:macSize], buf[macSize:headSize], buf[headSize:]
shouldhash := crypto.Keccak256(buf[macSize:])
if !bytes.Equal(hash, shouldhash) {
return nil, NodeID{}, nil, errBadHash
}
fromID, err := recoverNodeID(crypto.Keccak256(buf[headSize:]), sig)
if err != nil {
return nil, NodeID{}, 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)
default:
return nil, fromID, hash, fmt.Errorf("unknown type: %d", ptype)
}
s := rlp.NewStream(bytes.NewReader(sigdata[1:]), 0)
err = s.Decode(req)
return req, fromID, hash, err
}

func (req *ping) handle(t *udp, from *net.UDPAddr, fromID NodeID, mac []byte) error {
if expired(req.Expiration) {
return errExpired
}
t.send(from, pongPacket, &pong{
To:         makeEndpoint(from, req.From.TCP),
ReplyTok:   mac,
Expiration: uint64(time.Now().Add(expiration).Unix()),
})
if !t.handleReply(fromID, pingPacket, req) {
// Note: we're ignoring the provided IP address right now
go t.bond(true, fromID, from, req.From.TCP)
}
return nil
}

func (req *ping) name() string { return "PING/v4" }

func (req *pong) handle(t *udp, from *net.UDPAddr, fromID NodeID, mac []byte) error {
if expired(req.Expiration) {
return errExpired
}
if !t.handleReply(fromID, pongPacket, req) {
return errUnsolicitedReply
```

```
}
return nil
}

func (req *pong) name() string { return "PONG/v4" }

func (req *findnode) handle(t *udp, from *net.UDPAddr, fromID NodeID, mac []byte) error {
if expired(req.Expiration) {
return errExpired
}
if t.db.node(fromID) == nil {
// No bond exists, we don't process the packet. This prevents
// an attack vector where the discovery protocol could be used
// to amplify traffic in a DDOS attack. A malicious actor
// would send a findnode request with the IP address and UDP
// port of the target as the source address. The recipient of
// the findnode packet would then send a neighbors packet
// (which is a much bigger packet than findnode) to the victim.
return errUnknownNode
}
target := crypto.Keccak256Hash(req.Target[:])
t.mutex.Lock()
closest := t.closest(target, bucketSize).entries
t.mutex.Unlock()

p := neighbors{Expiration: uint64(time.Now().Add(expiration).Unix())}
// Send neighbors in chunks with at most maxNeighbors per packet
// to stay below the 1280 byte limit.
for i, n := range closest {
if netutil.CheckRelayIP(from.IP, n.IP) != nil {
continue
}
p.Nodes = append(p.Nodes, nodeToRPC(n))
if len(p.Nodes) == maxNeighbors || i == len(closest)-1 {
t.send(from, neighborsPacket, &p)
p.Nodes = p.Nodes[:0]
}
}
return nil
}

func (req *findnode) name() string { return "FINDNODE/v4" }
```

```
func (req *neighbors) handle(t *udp, from *net.UDPAddr, fromID NodeID, mac []byte) error {
if expired(req.Expiration) {
return errExpired
}
if !t.handleReply(fromID, neighborsPacket, req) {
return errUnsolicitedReply
}
return nil
}

func (req *neighbors) name() string { return "NEIGHBORS/v4" }

func expired(ts uint64) bool {
return time.Unix(int64(ts), 0).Before(time.Now())
}

91:F:\git\coin\ethereum\go-ethereum\p2p\discover\udp_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discover

import (
"bytes"
"crypto/ecdsa"
"encoding/binary"
"encoding/hex"
"errors"
"fmt"
"io"
"math/rand"
"net"
"path/filepath"
"reflect"
"runtime"
"sync"
"testing"
"time"

"github.com/davecgh/go-spew/spew"
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
```

```
"github.com/ethereum/go-ethereum/rlp"
)

func init() {
spew.Config.DisableMethods = true
}

// shared test variables
var (
futureExp          = uint64(time.Now().Add(10 * time.Hour).Unix())
testTarget         = NodeID{0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1}
testRemote         = rpcEndpoint{IP: net.ParseIP("1.1.1.1").To4(), UDP: 1, TCP: 2}
testLocalAnnounced = rpcEndpoint{IP: net.ParseIP("2.2.2.2").To4(), UDP: 3, TCP: 4}
testLocal          = rpcEndpoint{IP: net.ParseIP("3.3.3.3").To4(), UDP: 5, TCP: 6}
)

type udpTest struct {
t                   *testing.T
pipe                *dgramPipe
table               *Table
udp                 *udp
sent                [][]byte
localkey, remotekey *ecdsa.PrivateKey
remoteaddr          *net.UDPAddr
}

func newUDPTest(t *testing.T) *udpTest {
test := &udpTest{
t:          t,
pipe:       newpipe(),
localkey:   newkey(),
remotekey:  newkey(),
remoteaddr: &net.UDPAddr{IP: net.IP{10, 0, 1, 99}, Port: 30303},
}
test.table, test.udp, _ = newUDP(test.localkey, test.pipe, nil, "", nil)
return test
}

// handles a packet as if it had been sent to the transport.
func (test *udpTest) packetIn(wantError error, ptype byte, data packet) error {
enc, err := encodePacket(test.remotekey, ptype, data)
if err != nil {
```

```
return test.errorf("packet (%d) encode error: %v", ptype, err)
}
test.sent = append(test.sent, enc)
if err = test.udp.handlePacket(test.remoteaddr, enc); err != wantError {
return test.errorf("error mismatch: got %q, want %q", err, wantError)
}
return nil
}

// waits for a packet to be sent by the transport.
// validate should have type func(*udpTest, X) error, where X is a packet type.
func (test *udpTest) waitPacketOut(validate interface{}) error {
dgram := test.pipe.waitPacketOut()
p, _, _, err := decodePacket(dgram)
if err != nil {
return test.errorf("sent packet decode error: %v", err)
}
fn := reflect.ValueOf(validate)
exptype := fn.Type().In(0)
if reflect.TypeOf(p) != exptype {
return test.errorf("sent packet type mismatch, got: %v, want: %v", reflect.TypeOf(p), exptype)
}
fn.Call([]reflect.Value{reflect.ValueOf(p)})
return nil
}

func (test *udpTest) errorf(format string, args ...interface{}) error {
_, file, line, ok := runtime.Caller(2) // errorf + waitPacketOut
if ok {
file = filepath.Base(file)
} else {
file = "???"
line = 1
}
err := fmt.Errorf(format, args...)
fmt.Printf("\t%s:%d: %v\n", file, line, err)
test.t.Fail()
return err
}

func TestUDP_packetErrors(t *testing.T) {
test := newUDPTest(t)
```

```
defer test.table.Close()

test.packetIn(errExpired, pingPacket, &ping{From: testRemote, To: testLocalAnnounced, Version:
Version})
test.packetIn(errUnsolicitedReply, pongPacket, &pong{ReplyTok: []byte{}, Expiration: futureExp})
test.packetIn(errUnknownNode, findnodePacket, &findnode{Expiration: futureExp})
test.packetIn(errUnsolicitedReply, neighborsPacket, &neighbors{Expiration: futureExp})
}

func TestUDP_pingTimeout(t *testing.T) {
t.Parallel()
test := newUDPTest(t)
defer test.table.Close()

toaddr := &net.UDPAddr{IP: net.ParseIP("1.2.3.4"), Port: 2222}
toid := NodeID{1, 2, 3, 4}
if err := test.udp.ping(toid, toaddr); err != errTimeout {
t.Error("expected timeout error, got", err)
}
}

func TestUDP_responseTimeouts(t *testing.T) {
t.Parallel()
test := newUDPTest(t)
defer test.table.Close()

rand.Seed(time.Now().UnixNano())
randomDuration := func(max time.Duration) time.Duration {
return time.Duration(rand.Int63n(int64(max)))
}

var (
nReqs      = 200
nTimeouts  = 0                     // number of requests with ptype > 128
nilErr     = make(chan error, nReqs) // for requests that get a reply
timeoutErr = make(chan error, nReqs) // for requests that time out
)
for i := 0; i < nReqs; i++ {
// Create a matcher for a random request in udp.loop. Requests
// with ptype <= 128 will not get a reply and should time out.
// For all other requests, a reply is scheduled to arrive
// within the timeout window.
```

```
p := &pending{
ptype:    byte(rand.Intn(255)),
callback: func(interface{}) bool { return true },
}
binary.BigEndian.PutUint64(p.from[:], uint64(i))
if p.ptype <= 128 {
p.errc = timeoutErr
test.udp.addpending <- p
nTimeouts++
} else {
p.errc = nilErr
test.udp.addpending <- p
time.AfterFunc(randomDuration(60*time.Millisecond), func() {
if !test.udp.handleReply(p.from, p.ptype, nil) {
t.Logf("not matched: %v", p)
}
})
}
time.Sleep(randomDuration(30 * time.Millisecond))
}

// Check that all timeouts were delivered and that the rest got nil errors.
// The replies must be delivered.
var (
recvDeadline        = time.After(20 * time.Second)
nTimeoutsRecv, nNil = 0, 0
)
for i := 0; i < nReqs; i++ {
select {
case err := <-timeoutErr:
if err != errTimeout {
t.Fatalf("got non-timeout error on timeoutErr %d: %v", i, err)
}
nTimeoutsRecv++
case err := <-nilErr:
if err != nil {
t.Fatalf("got non-nil error on nilErr %d: %v", i, err)
}
nNil++
case <-recvDeadline:
t.Fatalf("exceeded recv deadline")
}
```

```
}
if nTimeoutsRecv != nTimeouts {
t.Errorf("wrong number of timeout errors received: got %d, want %d", nTimeoutsRecv, nTimeouts)
}
if nNil != nReqs-nTimeouts {
t.Errorf("wrong number of successful replies: got %d, want %d", nNil, nReqs-nTimeouts)
}
}

func TestUDP_findnodeTimeout(t *testing.T) {
t.Parallel()
test := newUDPTest(t)
defer test.table.Close()

toaddr := &net.UDPAddr{IP: net.ParseIP("1.2.3.4"), Port: 2222}
toid := NodeID{1, 2, 3, 4}
target := NodeID{4, 5, 6, 7}
result, err := test.udp.findnode(toid, toaddr, target)
if err != errTimeout {
t.Error("expected timeout error, got", err)
}
if len(result) > 0 {
t.Error("expected empty result, got", result)
}
}

func TestUDP_findnode(t *testing.T) {
test := newUDPTest(t)
defer test.table.Close()

// put a few nodes into the table. their exact
// distribution shouldn't matter much, although we need to
// take care not to overflow any bucket.
targetHash := crypto.Keccak256Hash(testTarget[:])
nodes := &nodesByDistance{target: targetHash}
for i := 0; i < bucketSize; i++ {
nodes.push(nodeAtDistance(test.table.self.sha, i+2), bucketSize)
}
test.table.stuff(nodes.entries)

// ensure there's a bond with the test node,
// findnode won't be accepted otherwise.
```

```
test.table.db.updateNode(NewNode(
PubkeyID(&test.remotekey.PublicKey),
test.remoteaddr.IP,
uint16(test.remoteaddr.Port),
99,
))
// check that closest neighbors are returned.
test.packetIn(nil, findnodePacket, &findnode{Target: testTarget, Expiration: futureExp})
expected := test.table.closest(targetHash, bucketSize)

waitNeighbors := func(want []*Node) {
test.waitPacketOut(func(p *neighbors) {
if len(p.Nodes) != len(want) {
t.Errorf("wrong number of results: got %d, want %d", len(p.Nodes), bucketSize)
}
for i := range p.Nodes {
if p.Nodes[i].ID != want[i].ID {
t.Errorf("result mismatch at %d:\n  got:  %v\n  want: %v", i, p.Nodes[i], expected.entries[i])
}
}
})
}
waitNeighbors(expected.entries[:maxNeighbors])
waitNeighbors(expected.entries[maxNeighbors:])
}

func TestUDP_findnodeMultiReply(t *testing.T) {
test := newUDPTest(t)
defer test.table.Close()

// queue a pending findnode request
resultc, errc := make(chan []*Node), make(chan error)
go func() {
rid := PubkeyID(&test.remotekey.PublicKey)
ns, err := test.udp.findnode(rid, test.remoteaddr, testTarget)
if err != nil && len(ns) == 0 {
errc <- err
} else {
resultc <- ns
}
}()
```

```
// wait for the findnode to be sent.
// after it is sent, the transport is waiting for a reply
test.waitPacketOut(func(p *findnode) {
if p.Target != testTarget {
t.Errorf("wrong target: got %v, want %v", p.Target, testTarget)
}
})

// send the reply as two packets.
list := []*Node{
MustParseNode("enode://ba85011c70bcc5c04d8607d3a0ed29aa6179c092cbdda10d5d32684fb3
3ed01bd94f588ca8f91ac48318087dcb02eaf36773a7a453f0eedd6742af668097b29c@10.0.1.16:3
0303?discport=30304"),
MustParseNode("enode://81fa361d25f157cd421c60dcc28d8dac5ef6a89476633339c5df30287474
520caca09627da18543d9079b5b288698b542d56167aa5c09111e55acdbbdf2ef799@10.0.1.16:3
0303"),
MustParseNode("enode://9bffefd833d53fac8e652415f4973bee289e8b1a5c6c4cbe70abf817ce8a6
4cee11b823b66a987f51aaa9fba0d6a91b3e6bf0d5a5d1042de8e9eeea057b217f8@10.0.1.36:303
01?discport=17"),
MustParseNode("enode://1b5b4aa662d7cb44a7221bfba67302590b643028197a7d5214790f3bac
7aaa4a3241be9e83c09cf1f6c69d007c634faae3dc1b1221793e8446c0b3a09de65960@10.0.1.16:
30303"),
}
rpclist := make([]rpcNode, len(list))
for i := range list {
rpclist[i] = nodeToRPC(list[i])
}
test.packetIn(nil, neighborsPacket, &neighbors{Expiration: futureExp, Nodes: rpclist[:2]})
test.packetIn(nil, neighborsPacket, &neighbors{Expiration: futureExp, Nodes: rpclist[2:]})

// check that the sent neighbors are all returned by findnode
select {
case result := <-resultc:
want := append(list[:2], list[3:]...)
if !reflect.DeepEqual(result, want) {
t.Errorf("neighbors mismatch:\n  got:  %v\n  want: %v", result, want)
}
case err := <-errc:
t.Errorf("findnode error: %v", err)
case <-time.After(5 * time.Second):
t.Error("findnode did not return within 5 seconds")
}
```

```
}

func TestUDP_successfulPing(t *testing.T) {
test := newUDPTest(t)
added := make(chan *Node, 1)
test.table.nodeAddedHook = func(n *Node) { added <- n }
defer test.table.Close()

// The remote side sends a ping packet to initiate the exchange.
go test.packetIn(nil, pingPacket, &ping{From: testRemote, To: testLocalAnnounced, Version:
Version, Expiration: futureExp})

// the ping is replied to.
test.waitPacketOut(func(p *pong) {
pinghash := test.sent[0][:macSize]
if !bytes.Equal(p.ReplyTok, pinghash) {
t.Errorf("got pong.ReplyTok %x, want %x", p.ReplyTok, pinghash)
}
wantTo := rpcEndpoint{
// The mirrored UDP address is the UDP packet sender
IP: test.remoteaddr.IP, UDP: uint16(test.remoteaddr.Port),
// The mirrored TCP port is the one from the ping packet
TCP: testRemote.TCP,
}
if !reflect.DeepEqual(p.To, wantTo) {
t.Errorf("got pong.To %v, want %v", p.To, wantTo)
}
})

// remote is unknown, the table pings back.
test.waitPacketOut(func(p *ping) error {
if !reflect.DeepEqual(p.From, test.udp.ourEndpoint) {
t.Errorf("got ping.From %v, want %v", p.From, test.udp.ourEndpoint)
}
wantTo := rpcEndpoint{
// The mirrored UDP address is the UDP packet sender.
IP: test.remoteaddr.IP, UDP: uint16(test.remoteaddr.Port),
TCP: 0,
}
if !reflect.DeepEqual(p.To, wantTo) {
t.Errorf("got ping.To %v, want %v", p.To, wantTo)
}
```

```
return nil
})
test.packetIn(nil, pongPacket, &pong{Expiration: futureExp})

// the node should be added to the table shortly after getting the
// pong packet.
select {
case n := <-added:
rid := PubkeyID(&test.remotekey.PublicKey)
if n.ID != rid {
t.Errorf("node has wrong ID: got %v, want %v", n.ID, rid)
}
if !n.IP.Equal(test.remoteaddr.IP) {
t.Errorf("node has wrong IP: got %v, want: %v", n.IP, test.remoteaddr.IP)
}
if int(n.UDP) != test.remoteaddr.Port {
t.Errorf("node has wrong UDP port: got %v, want: %v", n.UDP, test.remoteaddr.Port)
}
if n.TCP != testRemote.TCP {
t.Errorf("node has wrong TCP port: got %v, want: %v", n.TCP, testRemote.TCP)
}
case <-time.After(2 * time.Second):
t.Errorf("node was not added within 2 seconds")
}
}

var testPackets = []struct {
input      string
wantPacket interface{}
}{
{
input:
"71dbda3a79554728d4f94411e42ee1f8b0d561c10e1e5f5893367948c6a7d70bb87b235fa28a770
70271b6c164a2dce8c7e13a5739b53b5e96f2e5acb0e458a02902f5965d55ecbeb2ebb6cabb8b2b
232896a36b737666c55265ad0a68412f250001ea04cb847f000001820cfa8215a8d7900000000000
0000000000000000000018208ae820d058443b9a355",
wantPacket: &ping{
Version:    4,
From:       rpcEndpoint{net.ParseIP("127.0.0.1").To4(), 3322, 5544},
To:         rpcEndpoint{net.ParseIP("::1"), 2222, 3333},
Expiration: 1136239445,
Rest:       []rlp.RawValue{},
```

},
},
{
input:
"e9614ccfd9fc3e74360018522d30e1419a143407ffcce748de3e22116b7e8dc92ff74788c0b6663aa
a3d67d641936511c8f8d6ad8698b820a7cf9e1be7155e9a241f556658c55428ec0563514365799a4
be2be5a685a80971ddcfa80cb422cdd0101ec04cb847f000001820cfa8215a8d7900000000000000
0000000000000000018208ae820d058443b9a3550102",
wantPacket: &ping{
Version: 4,
From: rpcEndpoint{net.ParseIP("127.0.0.1").To4(), 3322, 5544},
To: rpcEndpoint{net.ParseIP("::1"), 2222, 3333},
Expiration: 1136239445,
Rest: []rlp.RawValue{{0x01}, {0x02}},
},
},
{
input:
"577be4349c4dd26768081f58de4c6f375a7a22f3f7adda654d1428637412c3d7fe917cadc56d4e5e
7ffae1dbe3efffb9849feb71b262de37977e7c7a44e677295680e9e38ab26bee2fcbae207fba3ff3d74
069a50b902a82c9903ed37cc993c50001f83e82022bd79020010db83c4d001500000000abcdef12
820cfa8215a8d79020010db885a308d313198a2e037073488208ae82823a8443b9a355c50102030
40531b9019afde696e582a78fa8d95ea13ce3297d4afb8ba6433e4154caa5ac6431af1b80ba76023f
a4090c408f6b4bc3701562c031041d4702971d102c9ab7fa5eed4cd6bab8f7af956f7d565ee191708
4a95398b6a21eac920fe3dd1345ec0a7ef39367ee69ddf092cbfe5b93e5e568ebc491983c09c76d9
22dc3",
wantPacket: &ping{
Version: 555,
From: rpcEndpoint{net.ParseIP("2001:db8:3c4d:15::abcd:ef12"), 3322, 5544},
To: rpcEndpoint{net.ParseIP("2001:db8:85a3:8d3:1319:8a2e:370:7348"), 2222, 33338},
Expiration: 1136239445,
Rest: []rlp.RawValue{{0xC5, 0x01, 0x02, 0x03, 0x04, 0x05}},
},
},
{
input:
"09b2428d83348d27cdf7064ad9024f526cebc19e4958f0fdad87c15eb598dd61d08423e0bf66b206
9869e1724125f820d851c136684082774f870e614d95a2855d000f05d1648b2d5945470bc187c2d2
216fbe870f43ed0909009882e176a46b0102f846d79020010db885a308d313198a2e03707348820
8ae82823aa0fbc914b16819237dcd8801d7e53f69e9719adecb3cc0e790c57e91ca4461c9548443b
9a355c6010203c2040506a0c969a58f6f9095004c0177a6b47f451530cab38966a25cca5cb58f0555
42124e",

wantPacket: &pong{
To: rpcEndpoint{net.ParseIP("2001:db8:85a3:8d3:1319:8a2e:370:7348"), 2222, 33338},
ReplyTok:
common.Hex2Bytes("fbc914b16819237dcd8801d7e53f69e9719adecb3cc0e790c57e91ca4461c954"),
Expiration: 1136239445,
Rest: []rlp.RawValue{{0xC6, 0x01, 0x02, 0x03, 0xC2, 0x04, 0x05}, {0x06}},
},
},
{
input:
"c7c44041b9f7c7e41934417ebac9a8e1a4c6298f74553f2fcfdcae6ed6fe53163eb3d2b52e39fe91831b8a927bf4fc222c3902202027e5e9eb812195f95d20061ef5cd31d502e47ecb61183f74a504fe04c51e73df81f25c4d506b26db4517490103f84eb840ca634cae0d49acb401d8a4c6b6fe8c55b70d115bf400769cc1400f3258cd31387574077f301b421bc84df7266c44e9e6d569fc56be00812904767bf5ccd1fc7f8443b9a35582999983999999280dc62cc8255c73471e0a61da0c89acdc0e035e260add7fc0c04ad9ebf3919644c91cb247affc82b69bd2ca235c71eab8e49737c937a2c396",
wantPacket: &findnode{
Target:
MustHexID("ca634cae0d49acb401d8a4c6b6fe8c55b70d115bf400769cc1400f3258cd31387574077f301b421bc84df7266c44e9e6d569fc56be00812904767bf5ccd1fc7f"),
Expiration: 1136239445,
Rest: []rlp.RawValue{{0x82, 0x99, 0x99}, {0x83, 0x99, 0x99, 0x99}},
},
},
{
input:
"c679fc8fe0b8b12f06577f2e802d34f6fa257e6137a995f6f4cbfc9ee50ed3710faf6e66f932c4c8d81d64343f429651328758b47d3dbc02c4042f0fff6946a50f4a49037a72bb550f3a7872363a83e1b9ee6469856c24eb4ef80b7535bcf99c0004f9015bf90150f84d846321163782115c82115db8403155e1427f85f10a5c9a7755877748041af1bcd8d474ec065eb33df57a97babf54bfd2103575fa829115d224c523596b401065a97f74010610fce76382c0bf32f84984010203040101b840312c55512422cf9b8a4097e9a6ad79402e87a15ae909a4bfefa22398f03d20951933beea1e4dfa6f968212385e829f04c2d314fc2d4e255e0d3bc08792b069dbf8599020010db83c4d001500000000abcdef12820d05820d05b84038643200b172dcfef857492156971f0e6aa2c538d8b74010f8e140811d53b98c765dd2d96126051913f44582e8c199ad7c6d6819e9a56483f637feaac9448aacf8599020010db885a308d313198a2e037073488203e78203e8b8408dcab8618c3253b558d459da53bd8fa68935a719aff8b811197101a4b2b47dd2d47295286fc00cc081bb542d760717d1bdd6bec2c37cd72eca367d6dd3b9df738443b9a355010203b525a138aa34383fec3d2719a0",
wantPacket: &neighbors{
Nodes: []rpcNode{
{

```
ID:
MustHexID("3155e1427f85f10a5c9a7755877748041af1bcd8d474ec065eb33df57a97babf54bfd21
03575fa829115d224c523596b401065a97f74010610fce76382c0bf32"),
IP:  net.ParseIP("99.33.22.55").To4(),
UDP: 4444,
TCP: 4445,
},
{
ID:
MustHexID("312c55512422cf9b8a4097e9a6ad79402e87a15ae909a4bfefa22398f03d20951933be
ea1e4dfa6f968212385e829f04c2d314fc2d4e255e0d3bc08792b069db"),
IP:  net.ParseIP("1.2.3.4").To4(),
UDP: 1,
TCP: 1,
},
{
ID:
MustHexID("38643200b172dcfef857492156971f0e6aa2c538d8b74010f8e140811d53b98c765dd2
d96126051913f44582e8c199ad7c6d6819e9a56483f637feaac9448aac"),
IP:  net.ParseIP("2001:db8:3c4d:15::abcd:ef12"),
UDP: 3333,
TCP: 3333,
},
{
ID:
MustHexID("8dcab8618c3253b558d459da53bd8fa68935a719aff8b811197101a4b2b47dd2d4729
5286fc00cc081bb542d760717d1bdd6bec2c37cd72eca367d6dd3b9df73"),
IP:  net.ParseIP("2001:db8:85a3:8d3:1319:8a2e:370:7348"),
UDP: 999,
TCP: 1000,
},
},
Expiration: 1136239445,
Rest:       []rlp.RawValue{{0x01}, {0x02}, {0x03}},
},
},
}

func TestForwardCompatibility(t *testing.T) {
testkey, _ :=
crypto.HexToECDSA("b71c71a67e1177ad4e901695e1b4b9ee17ae16c6668d313eac2f96dbcda3f
291")
```

```
wantNodeID := PubkeyID(&testkey.PublicKey)

for _, test := range testPackets {
input, err := hex.DecodeString(test.input)
if err != nil {
t.Fatalf("invalid hex: %s", test.input)
}
packet, nodeid, _, err := decodePacket(input)
if err != nil {
t.Errorf("did not accept packet %s\n%v", test.input, err)
continue
}
if !reflect.DeepEqual(packet, test.wantPacket) {
t.Errorf("got %s\nwant %s", spew.Sdump(packet), spew.Sdump(test.wantPacket))
}
if nodeid != wantNodeID {
t.Errorf("got id %v\nwant id %v", nodeid, wantNodeID)
}
}
}

// dgramPipe is a fake UDP socket. It queues all sent datagrams.
type dgramPipe struct {
mu      *sync.Mutex
cond    *sync.Cond
closing chan struct{}
closed  bool
queue   [][]byte
}

func newpipe() *dgramPipe {
mu := new(sync.Mutex)
return &dgramPipe{
closing: make(chan struct{}),
cond:    &sync.Cond{L: mu},
mu:      mu,
}
}

// WriteToUDP queues a datagram.
func (c *dgramPipe) WriteToUDP(b []byte, to *net.UDPAddr) (n int, err error) {
msg := make([]byte, len(b))
```

```
copy(msg, b)
c.mu.Lock()
defer c.mu.Unlock()
if c.closed {
return 0, errors.New("closed")
}
c.queue = append(c.queue, msg)
c.cond.Signal()
return len(b), nil
}

// ReadFromUDP just hangs until the pipe is closed.
func (c *dgramPipe) ReadFromUDP(b []byte) (n int, addr *net.UDPAddr, err error) {
<-c.closing
return 0, nil, io.EOF
}

func (c *dgramPipe) Close() error {
c.mu.Lock()
defer c.mu.Unlock()
if !c.closed {
close(c.closing)
c.closed = true
}
return nil
}

func (c *dgramPipe) LocalAddr() net.Addr {
return &net.UDPAddr{IP: testLocal.IP, Port: int(testLocal.UDP)}
}

func (c *dgramPipe) waitPacketOut() []byte {
c.mu.Lock()
defer c.mu.Unlock()
for len(c.queue) == 0 {
c.cond.Wait()
}
p := c.queue[0]
copy(c.queue, c.queue[1:])
c.queue = c.queue[:len(c.queue)-1]
return p
}
```

92:F:\git\coin\ethereum\go-ethereum\p2p\discv5\database.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains the node database, storing previously seen nodes and any collected
// metadata about them for QoS purposes.

package discv5

import (
"bytes"
"crypto/rand"
"encoding/binary"
"fmt"
"os"
"sync"
"time"

"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/rlp"
"github.com/syndtr/goleveldb/leveldb"
"github.com/syndtr/goleveldb/leveldb/errors"
"github.com/syndtr/goleveldb/leveldb/iterator"
"github.com/syndtr/goleveldb/leveldb/opt"
"github.com/syndtr/goleveldb/leveldb/storage"
"github.com/syndtr/goleveldb/leveldb/util"
)

var (
nodeDBNilNodeID   = NodeID{}   // Special node ID to use as a nil element.
nodeDBNodeExpiration = 24 * time.Hour // Time after which an unseen node should be dropped.
nodeDBCleanupCycle   = time.Hour   // Time period for running the expiration task.
)

// nodeDB stores all nodes we know about.
type nodeDB struct {
lvl   *leveldb.DB   // Interface to the database itself
self   NodeID   // Own node id to prevent adding it into the database
runner sync.Once   // Ensures we can start at most one expirer
quit   chan struct{} // Channel to signal the expiring thread to stop
}

```
// Schema layout for the node database
var (
nodeDBVersionKey = []byte("version") // Version of the database to flush if changes
nodeDBItemPrefix = []byte("n:")      // Identifier to prefix node entries with

nodeDBDiscoverRoot          = ":discover"
nodeDBDiscoverPing          = nodeDBDiscoverRoot + ":lastping"
nodeDBDiscoverPong          = nodeDBDiscoverRoot + ":lastpong"
nodeDBDiscoverFindFails     = nodeDBDiscoverRoot + ":findfail"
nodeDBDiscoverLocalEndpoint = nodeDBDiscoverRoot + ":localendpoint"
nodeDBTopicRegTickets       = ":tickets"
)

// newNodeDB creates a new node database for storing and retrieving infos about
// known peers in the network. If no path is given, an in-memory, temporary
// database is constructed.
func newNodeDB(path string, version int, self NodeID) (*nodeDB, error) {
if path == "" {
return newMemoryNodeDB(self)
}
return newPersistentNodeDB(path, version, self)
}

// newMemoryNodeDB creates a new in-memory node database without a persistent
// backend.
func newMemoryNodeDB(self NodeID) (*nodeDB, error) {
db, err := leveldb.Open(storage.NewMemStorage(), nil)
if err != nil {
return nil, err
}
return &nodeDB{
lvl:  db,
self: self,
quit: make(chan struct{}),
}, nil
}

// newPersistentNodeDB creates/opens a leveldb backed persistent node database,
// also flushing its contents in case of a version mismatch.
func newPersistentNodeDB(path string, version int, self NodeID) (*nodeDB, error) {
opts := &opt.Options{OpenFilesCacheCapacity: 5}
```

```
db, err := leveldb.OpenFile(path, opts)
if _, iscorrupted := err.(*errors.ErrCorrupted); iscorrupted {
db, err = leveldb.RecoverFile(path, nil)
}
if err != nil {
return nil, err
}
// The nodes contained in the cache correspond to a certain protocol version.
// Flush all nodes if the version doesn't match.
currentVer := make([]byte, binary.MaxVarintLen64)
currentVer = currentVer[:binary.PutVarint(currentVer, int64(version))]

blob, err := db.Get(nodeDBVersionKey, nil)
switch err {
case leveldb.ErrNotFound:
// Version not found (i.e. empty cache), insert it
if err := db.Put(nodeDBVersionKey, currentVer, nil); err != nil {
db.Close()
return nil, err
}

case nil:
// Version present, flush if different
if !bytes.Equal(blob, currentVer) {
db.Close()
if err = os.RemoveAll(path); err != nil {
return nil, err
}
return newPersistentNodeDB(path, version, self)
}
}
return &nodeDB{
lvl:  db,
self: self,
quit: make(chan struct{}),
}, nil
}

// makeKey generates the leveldb key-blob from a node id and its particular
// field of interest.
func makeKey(id NodeID, field string) []byte {
if bytes.Equal(id[:], nodeDBNilNodeID[:]) {
```

```
return []byte(field)
}
return append(nodeDBItemPrefix, append(id[:], field...)...)
}

// splitKey tries to split a database key into a node id and a field part.
func splitKey(key []byte) (id NodeID, field string) {
// If the key is not of a node, return it plainly
if !bytes.HasPrefix(key, nodeDBItemPrefix) {
return NodeID{}, string(key)
}
// Otherwise split the id and field
item := key[len(nodeDBItemPrefix):]
copy(id[:], item[:len(id)])
field = string(item[len(id):])

return id, field
}

// fetchInt64 retrieves an integer instance associated with a particular
// database key.
func (db *nodeDB) fetchInt64(key []byte) int64 {
blob, err := db.lvl.Get(key, nil)
if err != nil {
return 0
}
val, read := binary.Varint(blob)
if read <= 0 {
return 0
}
return val
}

// storeInt64 update a specific database entry to the current time instance as a
// unix timestamp.
func (db *nodeDB) storeInt64(key []byte, n int64) error {
blob := make([]byte, binary.MaxVarintLen64)
blob = blob[:binary.PutVarint(blob, n)]
return db.lvl.Put(key, blob, nil)
}

func (db *nodeDB) storeRLP(key []byte, val interface{}) error {
```

```
blob, err := rlp.EncodeToBytes(val)
if err != nil {
return err
}
return db.lvl.Put(key, blob, nil)
}

func (db *nodeDB) fetchRLP(key []byte, val interface{}) error {
blob, err := db.lvl.Get(key, nil)
if err != nil {
return err
}
err = rlp.DecodeBytes(blob, val)
if err != nil {
log.Warn(fmt.Sprintf("key %x (%T) %v", key, val, err))
}
return err
}

// node retrieves a node with a given id from the database.
func (db *nodeDB) node(id NodeID) *Node {
var node Node
if err := db.fetchRLP(makeKey(id, nodeDBDiscoverRoot), &node); err != nil {
return nil
}
node.sha = crypto.Keccak256Hash(node.ID[:])
return &node
}

// updateNode inserts - potentially overwriting - a node into the peer database.
func (db *nodeDB) updateNode(node *Node) error {
return db.storeRLP(makeKey(node.ID, nodeDBDiscoverRoot), node)
}

// deleteNode deletes all information/keys associated with a node.
func (db *nodeDB) deleteNode(id NodeID) error {
deleter := db.lvl.NewIterator(util.BytesPrefix(makeKey(id, "")), nil)
for deleter.Next() {
if err := db.lvl.Delete(deleter.Key(), nil); err != nil {
return err
}
}
```

```
return nil
}

// ensureExpirer is a small helper method ensuring that the data expiration
// mechanism is running. If the expiration goroutine is already running, this
// method simply returns.
//
// The goal is to start the data evacuation only after the network successfully
// bootstrapped itself (to prevent dumping potentially useful seed nodes). Since
// it would require significant overhead to exactly trace the first successful
// convergence, it's simpler to "ensure" the correct state when an appropriate
// condition occurs (i.e. a successful bonding), and discard further events.
func (db *nodeDB) ensureExpirer() {
db.runner.Do(func() { go db.expirer() })
}

// expirer should be started in a go routine, and is responsible for looping ad
// infinitum and dropping stale data from the database.
func (db *nodeDB) expirer() {
tick := time.Tick(nodeDBCleanupCycle)
for {
select {
case <-tick:
if err := db.expireNodes(); err != nil {
log.Error(fmt.Sprintf("Failed to expire nodedb items: %v", err))
}

case <-db.quit:
return
}
}
}

// expireNodes iterates over the database and deletes all nodes that have not
// been seen (i.e. received a pong from) for some allotted time.
func (db *nodeDB) expireNodes() error {
threshold := time.Now().Add(-nodeDBNodeExpiration)

// Find discovered nodes that are older than the allowance
it := db.lvl.NewIterator(nil, nil)
defer it.Release()
```

```
for it.Next() {
// Skip the item if not a discovery node
id, field := splitKey(it.Key())
if field != nodeDBDiscoverRoot {
continue
}
// Skip the node if not expired yet (and not self)
if !bytes.Equal(id[:], db.self[:]) {
if seen := db.lastPong(id); seen.After(threshold) {
continue
}
}
// Otherwise delete all associated information
db.deleteNode(id)
}
return nil
}

// lastPing retrieves the time of the last ping packet send to a remote node,
// requesting binding.
func (db *nodeDB) lastPing(id NodeID) time.Time {
return time.Unix(db.fetchInt64(makeKey(id, nodeDBDiscoverPing)), 0)
}

// updateLastPing updates the last time we tried contacting a remote node.
func (db *nodeDB) updateLastPing(id NodeID, instance time.Time) error {
return db.storeInt64(makeKey(id, nodeDBDiscoverPing), instance.Unix())
}

// lastPong retrieves the time of the last successful contact from remote node.
func (db *nodeDB) lastPong(id NodeID) time.Time {
return time.Unix(db.fetchInt64(makeKey(id, nodeDBDiscoverPong)), 0)
}

// updateLastPong updates the last time a remote node successfully contacted.
func (db *nodeDB) updateLastPong(id NodeID, instance time.Time) error {
return db.storeInt64(makeKey(id, nodeDBDiscoverPong), instance.Unix())
}

// findFails retrieves the number of findnode failures since bonding.
func (db *nodeDB) findFails(id NodeID) int {
return int(db.fetchInt64(makeKey(id, nodeDBDiscoverFindFails)))
```

```
}

// updateFindFails updates the number of findnode failures since bonding.
func (db *nodeDB) updateFindFails(id NodeID, fails int) error {
return db.storeInt64(makeKey(id, nodeDBDiscoverFindFails), int64(fails))
}

// localEndpoint returns the last local endpoint communicated to the
// given remote node.
func (db *nodeDB) localEndpoint(id NodeID) *rpcEndpoint {
var ep rpcEndpoint
if err := db.fetchRLP(makeKey(id, nodeDBDiscoverLocalEndpoint), &ep); err != nil {
return nil
}
return &ep
}

func (db *nodeDB) updateLocalEndpoint(id NodeID, ep rpcEndpoint) error {
return db.storeRLP(makeKey(id, nodeDBDiscoverLocalEndpoint), &ep)
}

// querySeeds retrieves random nodes to be used as potential seed nodes
// for bootstrapping.
func (db *nodeDB) querySeeds(n int, maxAge time.Duration) []*Node {
var (
now   = time.Now()
nodes = make([]*Node, 0, n)
it    = db.lvl.NewIterator(nil, nil)
id    NodeID
)
defer it.Release()

seek:
for seeks := 0; len(nodes) < n && seeks < n*5; seeks++ {
// Seek to a random entry. The first byte is incremented by a
// random amount each time in order to increase the likelihood
// of hitting all existing nodes in very small databases.
ctr := id[0]
rand.Read(id[:])
id[0] = ctr + id[0]%16
it.Seek(makeKey(id, nodeDBDiscoverRoot))
```

```
n := nextNode(it)
if n == nil {
id[0] = 0
continue seek // iterator exhausted
}
if n.ID == db.self {
continue seek
}
if now.Sub(db.lastPong(n.ID)) > maxAge {
continue seek
}
for i := range nodes {
if nodes[i].ID == n.ID {
continue seek // duplicate
}
}
nodes = append(nodes, n)
}
return nodes
}

func (db *nodeDB) fetchTopicRegTickets(id NodeID) (issued, used uint32) {
key := makeKey(id, nodeDBTopicRegTickets)
blob, _ := db.lvl.Get(key, nil)
if len(blob) != 8 {
return 0, 0
}
issued = binary.BigEndian.Uint32(blob[0:4])
used = binary.BigEndian.Uint32(blob[4:8])
return
}

func (db *nodeDB) updateTopicRegTickets(id NodeID, issued, used uint32) error {
key := makeKey(id, nodeDBTopicRegTickets)
blob := make([]byte, 8)
binary.BigEndian.PutUint32(blob[0:4], issued)
binary.BigEndian.PutUint32(blob[4:8], used)
return db.lvl.Put(key, blob, nil)
}

// reads the next node record from the iterator, skipping over other
// database entries.
```

```
func nextNode(it iterator.Iterator) *Node {
for end := false; !end; end = !it.Next() {
id, field := splitKey(it.Key())
if field != nodeDBDiscoverRoot {
continue
}
var n Node
if err := rlp.DecodeBytes(it.Value(), &n); err != nil {
log.Warn(fmt.Sprintf("invalid node %x: %v", id, err))
continue
}
return &n
}
return nil
}

// close flushes and closes the database files.
func (db *nodeDB) close() {
close(db.quit)
db.lvl.Close()
}

93:F:\git\coin\ethereum\go-ethereum\p2p\discv5\database_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discv5

import (
"bytes"
"io/ioutil"
"net"
"os"
"path/filepath"
"reflect"
"testing"
"time"
)

var nodeDBKeyTests = []struct {
id    NodeID
field string
key   []byte
```

```
}{
{
id:    NodeID{},
field: "version",
key:   []byte{0x76, 0x65, 0x72, 0x73, 0x69, 0x6f, 0x6e}, // field
},
{
id:
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
field: ":discover",
key: []byte{0x6e, 0x3a, // prefix
0x1d, 0xd9, 0xd6, 0x5c, 0x45, 0x52, 0xb5, 0xeb, // node id
0x43, 0xd5, 0xad, 0x55, 0xa2, 0xee, 0x3f, 0x56, //
0xc6, 0xcb, 0xc1, 0xc6, 0x4a, 0x5c, 0x8d, 0x65, //
0x9f, 0x51, 0xfc, 0xd5, 0x1b, 0xac, 0xe2, 0x43, //
0x51, 0x23, 0x2b, 0x8d, 0x78, 0x21, 0x61, 0x7d, //
0x2b, 0x29, 0xb5, 0x4b, 0x81, 0xcd, 0xef, 0xb9, //
0xb3, 0xe9, 0xc3, 0x7d, 0x7f, 0xd5, 0xf6, 0x32, //
0x70, 0xbc, 0xc9, 0xe1, 0xa6, 0xf6, 0xa4, 0x39, //
0x3a, 0x64, 0x69, 0x73, 0x63, 0x6f, 0x76, 0x65, 0x72, // field
},
},
}

func TestNodeDBKeys(t *testing.T) {
for i, tt := range nodeDBKeyTests {
if key := makeKey(tt.id, tt.field); !bytes.Equal(key, tt.key) {
t.Errorf("make test %d: key mismatch: have 0x%x, want 0x%x", i, key, tt.key)
}
id, field := splitKey(tt.key)
if !bytes.Equal(id[:], tt.id[:]) {
t.Errorf("split test %d: id mismatch: have 0x%x, want 0x%x", i, id, tt.id)
}
if field != tt.field {
t.Errorf("split test %d: field mismatch: have 0x%x, want 0x%x", i, field, tt.field)
}
}
}

var nodeDBInt64Tests = []struct {
key   []byte
```

```
value int64
}{
{key: []byte{0x01}, value: 1},
{key: []byte{0x02}, value: 2},
{key: []byte{0x03}, value: 3},
}

func TestNodeDBInt64(t *testing.T) {
db, _ := newNodeDB("", Version, NodeID{})
defer db.close()

tests := nodeDBInt64Tests
for i := 0; i < len(tests); i++ {
// Insert the next value
if err := db.storeInt64(tests[i].key, tests[i].value); err != nil {
t.Errorf("test %d: failed to store value: %v", i, err)
}
// Check all existing and non existing values
for j := 0; j < len(tests); j++ {
num := db.fetchInt64(tests[j].key)
switch {
case j <= i && num != tests[j].value:
t.Errorf("test %d, item %d: value mismatch: have %v, want %v", i, j, num, tests[j].value)
case j > i && num != 0:
t.Errorf("test %d, item %d: value mismatch: have %v, want %v", i, j, num, 0)
}
}
}
}

func TestNodeDBFetchStore(t *testing.T) {
node := NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{192, 168, 0, 1},
30303,
30303,
)
inst := time.Now()
num := 314

db, _ := newNodeDB("", Version, NodeID{})
```

```
defer db.close()

// Check fetch/store operations on a node ping object
if stored := db.lastPing(node.ID); stored.Unix() != 0 {
t.Errorf("ping: non-existing object: %v", stored)
}
if err := db.updateLastPing(node.ID, inst); err != nil {
t.Errorf("ping: failed to update: %v", err)
}
if stored := db.lastPing(node.ID); stored.Unix() != inst.Unix() {
t.Errorf("ping: value mismatch: have %v, want %v", stored, inst)
}
// Check fetch/store operations on a node pong object
if stored := db.lastPong(node.ID); stored.Unix() != 0 {
t.Errorf("pong: non-existing object: %v", stored)
}
if err := db.updateLastPong(node.ID, inst); err != nil {
t.Errorf("pong: failed to update: %v", err)
}
if stored := db.lastPong(node.ID); stored.Unix() != inst.Unix() {
t.Errorf("pong: value mismatch: have %v, want %v", stored, inst)
}
// Check fetch/store operations on a node findnode-failure object
if stored := db.findFails(node.ID); stored != 0 {
t.Errorf("find-node fails: non-existing object: %v", stored)
}
if err := db.updateFindFails(node.ID, num); err != nil {
t.Errorf("find-node fails: failed to update: %v", err)
}
if stored := db.findFails(node.ID); stored != num {
t.Errorf("find-node fails: value mismatch: have %v, want %v", stored, num)
}
// Check fetch/store operations on an actual node object
if stored := db.node(node.ID); stored != nil {
t.Errorf("node: non-existing object: %v", stored)
}
if err := db.updateNode(node); err != nil {
t.Errorf("node: failed to update: %v", err)
}
if stored := db.node(node.ID); stored == nil {
t.Errorf("node: not found")
} else if !reflect.DeepEqual(stored, node) {
```

```
t.Errorf("node: data mismatch: have %v, want %v", stored, node)
}
}

var nodeDBSeedQueryNodes = []struct {
node *Node
pong time.Time
}{
// This one should not be in the result set because its last
// pong time is too far in the past.
{
node: NewNode(
MustHexID("0x84d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{127, 0, 0, 3},
30303,
30303,
),
pong: time.Now().Add(-3 * time.Hour),
},
// This one shouldn't be in in the result set because its
// nodeID is the local node's ID.
{
node: NewNode(
MustHexID("0x57d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{127, 0, 0, 3},
30303,
30303,
),
pong: time.Now().Add(-4 * time.Second),
},

// These should be in the result set.
{
node: NewNode(
MustHexID("0x22d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{127, 0, 0, 1},
30303,
30303,
),
```

```
pong: time.Now().Add(-2 * time.Second),
},
{
node: NewNode(
MustHexID("0x44d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{127, 0, 0, 2},
30303,
30303,
),
pong: time.Now().Add(-3 * time.Second),
},
{
node: NewNode(
MustHexID("0xe2d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{127, 0, 0, 3},
30303,
30303,
),
pong: time.Now().Add(-1 * time.Second),
},
}

func TestNodeDBSeedQuery(t *testing.T) {
db, _ := newNodeDB("", Version, nodeDBSeedQueryNodes[1].node.ID)
defer db.close()

// Insert a batch of nodes for querying
for i, seed := range nodeDBSeedQueryNodes {
if err := db.updateNode(seed.node); err != nil {
t.Fatalf("node %d: failed to insert: %v", i, err)
}
if err := db.updateLastPong(seed.node.ID, seed.pong); err != nil {
t.Fatalf("node %d: failed to insert lastPong: %v", i, err)
}
}

// Retrieve the entire batch and check for duplicates
seeds := db.querySeeds(len(nodeDBSeedQueryNodes)*2, time.Hour)
have := make(map[NodeID]struct{})
for _, seed := range seeds {
```

```
have[seed.ID] = struct{}{}
}
want := make(map[NodeID]struct{})
for _, seed := range nodeDBSeedQueryNodes[2:] {
want[seed.node.ID] = struct{}{}
}
if len(seeds) != len(want) {
t.Errorf("seed count mismatch: have %v, want %v", len(seeds), len(want))
}
for id := range have {
if _, ok := want[id]; !ok {
t.Errorf("extra seed: %v", id)
}
}
for id := range want {
if _, ok := have[id]; !ok {
t.Errorf("missing seed: %v", id)
}
}
}

func TestNodeDBPersistency(t *testing.T) {
root, err := ioutil.TempDir("", "nodedb-")
if err != nil {
t.Fatalf("failed to create temporary data folder: %v", err)
}
defer os.RemoveAll(root)

var (
testKey = []byte("somekey")
testInt = int64(314)
)

// Create a persistent database and store some values
db, err := newNodeDB(filepath.Join(root, "database"), Version, NodeID{})
if err != nil {
t.Fatalf("failed to create persistent database: %v", err)
}
if err := db.storeInt64(testKey, testInt); err != nil {
t.Fatalf("failed to store value: %v.", err)
}
db.close()
```

```
// Reopen the database and check the value
db, err = newNodeDB(filepath.Join(root, "database"), Version, NodeID{})
if err != nil {
t.Fatalf("failed to open persistent database: %v", err)
}
if val := db.fetchInt64(testKey); val != testInt {
t.Fatalf("value mismatch: have %v, want %v", val, testInt)
}
db.close()

// Change the database version and check flush
db, err = newNodeDB(filepath.Join(root, "database"), Version+1, NodeID{})
if err != nil {
t.Fatalf("failed to open persistent database: %v", err)
}
if val := db.fetchInt64(testKey); val != 0 {
t.Fatalf("value mismatch: have %v, want %v", val, 0)
}
db.close()
}

var nodeDBExpirationNodes = []struct {
node *Node
pong time.Time
exp  bool
}{
{
node: NewNode(
MustHexID("0x01d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{127, 0, 0, 1},
30303,
30303,
),
pong: time.Now().Add(-nodeDBNodeExpiration + time.Minute),
exp:  false,
}, {
node: NewNode(
MustHexID("0x02d9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{127, 0, 0, 2},
```

```
30303,
30303,
),
pong: time.Now().Add(-nodeDBNodeExpiration - time.Minute),
exp:  true,
},
}

func TestNodeDBExpiration(t *testing.T) {
db, _ := newNodeDB("", Version, NodeID{})
defer db.close()

// Add all the test nodes and set their last pong time
for i, seed := range nodeDBExpirationNodes {
if err := db.updateNode(seed.node); err != nil {
t.Fatalf("node %d: failed to insert: %v", i, err)
}
if err := db.updateLastPong(seed.node.ID, seed.pong); err != nil {
t.Fatalf("node %d: failed to update pong: %v", i, err)
}
}
// Expire some of them, and check the rest
if err := db.expireNodes(); err != nil {
t.Fatalf("failed to expire nodes: %v", err)
}
for i, seed := range nodeDBExpirationNodes {
node := db.node(seed.node.ID)
if (node == nil && !seed.exp) || (node != nil && seed.exp) {
t.Errorf("node %d: expiration mismatch: have %v, want %v", i, node, seed.exp)
}
}
}

func TestNodeDBSelfExpiration(t *testing.T) {
// Find a node in the tests that shouldn't expire, and assign it as self
var self NodeID
for _, node := range nodeDBExpirationNodes {
if !node.exp {
self = node.node.ID
break
}
}
```

```
db, _ := newNodeDB("", Version, self)
defer db.close()

// Add all the test nodes and set their last pong time
for i, seed := range nodeDBExpirationNodes {
if err := db.updateNode(seed.node); err != nil {
t.Fatalf("node %d: failed to insert: %v", i, err)
}
if err := db.updateLastPong(seed.node.ID, seed.pong); err != nil {
t.Fatalf("node %d: failed to update pong: %v", i, err)
}
}
// Expire the nodes and make sure self has been evacuated too
if err := db.expireNodes(); err != nil {
t.Fatalf("failed to expire nodes: %v", err)
}
node := db.node(self)
if node != nil {
t.Errorf("self not evacuated")
}
}

94:F:\git\coin\ethereum\go-ethereum\p2p\discv5\net.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discv5

import (
"bytes"
"crypto/ecdsa"
"errors"
"fmt"
"net"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/mclock"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/crypto/sha3"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/p2p/nat"
"github.com/ethereum/go-ethereum/p2p/netutil"
```

```
"github.com/ethereum/go-ethereum/rlp"
)

var (
errInvalidEvent = errors.New("invalid in current state")
errNoQuery      = errors.New("no pending query")
errWrongAddress = errors.New("unknown sender address")
)

const (
autoRefreshInterval   = 1 * time.Hour
bucketRefreshInterval = 1 * time.Minute
seedCount             = 30
seedMaxAge            = 5 * 24 * time.Hour
lowPort               = 1024
)

const testTopic = "foo"

const (
printDebugLogs   = false
printTestImgLogs = false
)

func debugLog(s string) {
if printDebugLogs {
fmt.Println(s)
}
}

// Network manages the table and all protocol interaction.
type Network struct {
db          *nodeDB // database of known nodes
conn        transport
netrestrict *netutil.Netlist

closed           chan struct{}          // closed when loop is done
closeReq         chan struct{}          // 'request to close'
refreshReq       chan []*Node           // lookups ask for refresh on this channel
refreshResp      chan (<-chan struct{}) // ...and get the channel to block on from this one
read             chan ingressPacket     // ingress packets arrive here
timeout          chan timeoutEvent
```

```
queryReq         chan *findnodeQuery // lookups submit findnode queries on this channel
tableOpReq       chan func()
tableOpResp      chan struct{}
topicRegisterReq chan topicRegisterReq
topicSearchReq   chan topicSearchReq

// State of the main loop.
tab           *Table
topictab      *topicTable
ticketStore   *ticketStore
nursery       []*Node
nodes         map[NodeID]*Node // tracks active nodes with state != known
timeoutTimers map[timeoutEvent]*time.Timer

// Revalidation queues.
// Nodes put on these queues will be pinged eventually.
slowRevalidateQueue []*Node
fastRevalidateQueue []*Node

// Buffers for state transition.
sendBuf []*ingressPacket
}

// transport is implemented by the UDP transport.
// it is an interface so we can test without opening lots of UDP
// sockets and without generating a private key.
type transport interface {
sendPing(remote *Node, remoteAddr *net.UDPAddr, topics []Topic) (hash []byte)
sendNeighbours(remote *Node, nodes []*Node)
sendFindnodeHash(remote *Node, target common.Hash)
sendTopicRegister(remote *Node, topics []Topic, topicIdx int, pong []byte)
sendTopicNodes(remote *Node, queryHash common.Hash, nodes []*Node)

send(remote *Node, ptype nodeEvent, p interface{}) (hash []byte)

localAddr() *net.UDPAddr
Close()
}

type findnodeQuery struct {
remote   *Node
target   common.Hash
```

```
reply    chan<- []*Node
nresults int // counter for received nodes
}

type topicRegisterReq struct {
add   bool
topic Topic
}

type topicSearchReq struct {
topic  Topic
found  chan<- *Node
lookup chan<- bool
delay  time.Duration
}

type topicSearchResult struct {
target lookupInfo
nodes  []*Node
}

type timeoutEvent struct {
ev   nodeEvent
node *Node
}

func newNetwork(conn transport, ourPubkey ecdsa.PublicKey, natm nat.Interface, dbPath string,
netrestrict *netutil.Netlist) (*Network, error) {
ourID := PubkeyID(&ourPubkey)

var db *nodeDB
if dbPath != "<no database>" {
var err error
if db, err = newNodeDB(dbPath, Version, ourID); err != nil {
return nil, err
}
}

tab := newTable(ourID, conn.localAddr())
net := &Network{
db:               db,
conn:             conn,
```

```
netrestrict:      netrestrict,
tab:              tab,
topictab:         newTopicTable(db, tab.self),
ticketStore:      newTicketStore(),
refreshReq:       make(chan []*Node),
refreshResp:      make(chan (<-chan struct{})),
closed:           make(chan struct{}),
closeReq:         make(chan struct{}),
read:             make(chan ingressPacket, 100),
timeout:          make(chan timeoutEvent),
timeoutTimers:    make(map[timeoutEvent]*time.Timer),
tableOpReq:       make(chan func()),
tableOpResp:      make(chan struct{}),
queryReq:         make(chan *findnodeQuery),
topicRegisterReq: make(chan topicRegisterReq),
topicSearchReq:   make(chan topicSearchReq),
nodes:            make(map[NodeID]*Node),
}
go net.loop()
return net, nil
}

// Close terminates the network listener and flushes the node database.
func (net *Network) Close() {
net.conn.Close()
select {
case <-net.closed:
case net.closeReq <- struct{}{}:
<-net.closed
}
}

// Self returns the local node.
// The returned node should not be modified by the caller.
func (net *Network) Self() *Node {
return net.tab.self
}

// ReadRandomNodes fills the given slice with random nodes from the
// table. It will not write the same node more than once. The nodes in
// the slice are copies and can be modified by the caller.
func (net *Network) ReadRandomNodes(buf []*Node) (n int) {
```

```
net.reqTableOp(func() { n = net.tab.readRandomNodes(buf) })
return n
}

// SetFallbackNodes sets the initial points of contact. These nodes
// are used to connect to the network if the table is empty and there
// are no known nodes in the database.
func (net *Network) SetFallbackNodes(nodes []*Node) error {
nursery := make([]*Node, 0, len(nodes))
for _, n := range nodes {
if err := n.validateComplete(); err != nil {
return fmt.Errorf("bad bootstrap/fallback node %q (%v)", n, err)
}
// Recompute cpy.sha because the node might not have been
// created by NewNode or ParseNode.
cpy := *n
cpy.sha = crypto.Keccak256Hash(n.ID[:])
nursery = append(nursery, &cpy)
}
net.reqRefresh(nursery)
return nil
}

// Resolve searches for a specific node with the given ID.
// It returns nil if the node could not be found.
func (net *Network) Resolve(targetID NodeID) *Node {
result := net.lookup(crypto.Keccak256Hash(targetID[:]), true)
for _, n := range result {
if n.ID == targetID {
return n
}
}
return nil
}

// Lookup performs a network search for nodes close
// to the given target. It approaches the target by querying
// nodes that are closer to it on each iteration.
// The given target does not need to be an actual node
// identifier.
//
// The local node may be included in the result.
```

```
func (net *Network) Lookup(targetID NodeID) []*Node {
return net.lookup(crypto.Keccak256Hash(targetID[:]), false)
}

func (net *Network) lookup(target common.Hash, stopOnMatch bool) []*Node {
var (
asked          = make(map[NodeID]bool)
seen           = make(map[NodeID]bool)
reply          = make(chan []*Node, alpha)
result         = nodesByDistance{target: target}
pendingQueries = 0
)
// Get initial answers from the local node.
result.push(net.tab.self, bucketSize)
for {
// Ask the  closest nodes that we haven't asked yet.
for i := 0; i < len(result.entries) && pendingQueries < alpha; i++ {
n := result.entries[i]
if !asked[n.ID] {
asked[n.ID] = true
pendingQueries++
net.reqQueryFindnode(n, target, reply)
}
}
if pendingQueries == 0 {
// We have asked all closest nodes, stop the search.
break
}
// Wait for the next reply.
select {
case nodes := <-reply:
for _, n := range nodes {
if n != nil && !seen[n.ID] {
seen[n.ID] = true
result.push(n, bucketSize)
if stopOnMatch && n.sha == target {
return result.entries
}
}
}
pendingQueries--
case <-time.After(respTimeout):
```

```
// forget all pending requests, start new ones
pendingQueries = 0
reply = make(chan []*Node, alpha)
}
}
return result.entries
}

func (net *Network) RegisterTopic(topic Topic, stop <-chan struct{}) {
select {
case net.topicRegisterReq <- topicRegisterReq{true, topic}:
case <-net.closed:
return
}
select {
case <-net.closed:
case <-stop:
select {
case net.topicRegisterReq <- topicRegisterReq{false, topic}:
case <-net.closed:
}
}
}

func (net *Network) SearchTopic(topic Topic, setPeriod <-chan time.Duration, found chan<-
*Node, lookup chan<- bool) {
for {
select {
case <-net.closed:
return
case delay, ok := <-setPeriod:
select {
case net.topicSearchReq <- topicSearchReq{topic: topic, found: found, lookup: lookup, delay:
delay}:
case <-net.closed:
return
}
if !ok {
return
}
}
}
```

```
}

func (net *Network) reqRefresh(nursery []*Node) <-chan struct{} {
select {
case net.refreshReq <- nursery:
return <-net.refreshResp
case <-net.closed:
return net.closed
}
}

func (net *Network) reqQueryFindnode(n *Node, target common.Hash, reply chan []*Node) bool {
q := &findnodeQuery{remote: n, target: target, reply: reply}
select {
case net.queryReq <- q:
return true
case <-net.closed:
return false
}
}

func (net *Network) reqReadPacket(pkt ingressPacket) {
select {
case net.read <- pkt:
case <-net.closed:
}
}

func (net *Network) reqTableOp(f func()) (called bool) {
select {
case net.tableOpReq <- f:
<-net.tableOpResp
return true
case <-net.closed:
return false
}
}

// TODO: external address handling.

type topicSearchInfo struct {
lookupChn chan<- bool
```

```
period    time.Duration
}

const maxSearchCount = 5

func (net *Network) loop() {
var (
refreshTimer       = time.NewTicker(autoRefreshInterval)
bucketRefreshTimer = time.NewTimer(bucketRefreshInterval)
refreshDone        chan struct{} // closed when the 'refresh' lookup has ended
)

// Tracking the next ticket to register.
var (
nextTicket        *ticketRef
nextRegisterTimer *time.Timer
nextRegisterTime  <-chan time.Time
)
defer func() {
if nextRegisterTimer != nil {
nextRegisterTimer.Stop()
}
}()
resetNextTicket := func() {
t, timeout := net.ticketStore.nextFilteredTicket()
if t != nextTicket {
nextTicket = t
if nextRegisterTimer != nil {
nextRegisterTimer.Stop()
nextRegisterTime = nil
}
if t != nil {
nextRegisterTimer = time.NewTimer(timeout)
nextRegisterTime = nextRegisterTimer.C
}
}
}

// Tracking registration and search lookups.
var (
topicRegisterLookupTarget lookupInfo
topicRegisterLookupDone   chan []*Node
```

```
topicRegisterLookupTick   = time.NewTimer(0)
searchReqWhenRefreshDone  []topicSearchReq
searchInfo                = make(map[Topic]topicSearchInfo)
activeSearchCount         int
)
topicSearchLookupDone := make(chan topicSearchResult, 100)
topicSearch := make(chan Topic, 100)
<-topicRegisterLookupTick.C

statsDump := time.NewTicker(10 * time.Second)

loop:
for {
resetNextTicket()

select {
case <-net.closeReq:
debugLog("<-net.closeReq")
break loop

// Ingress packet handling.
case pkt := <-net.read:
//fmt.Println("read", pkt.ev)
debugLog("<-net.read")
n := net.internNode(&pkt)
prestate := n.state
status := "ok"
if err := net.handle(n, pkt.ev, &pkt); err != nil {
status = err.Error()
}
log.Trace("", "msg", log.Lazy{Fn: func() string {
return fmt.Sprintf("<<< (%d) %v from %x@%v: %v -> %v (%v)",
net.tab.count, pkt.ev, pkt.remoteID[:8], pkt.remoteAddr, prestate, n.state, status)
}})
// TODO: persist state if n.state goes >= known, delete if it goes <= known

// State transition timeouts.
case timeout := <-net.timeout:
debugLog("<-net.timeout")
if net.timeoutTimers[timeout] == nil {
// Stale timer (was aborted).
continue
```

```
}
delete(net.timeoutTimers, timeout)
prestate := timeout.node.state
status := "ok"
if err := net.handle(timeout.node, timeout.ev, nil); err != nil {
status = err.Error()
}
log.Trace("", "msg", log.Lazy{Fn: func() string {
return fmt.Sprintf("--- (%d) %v for %x@%v: %v -> %v (%v)",
net.tab.count, timeout.ev, timeout.node.ID[:8], timeout.node.addr(), prestate, timeout.node.state,
status)
}})

// Querying.
case q := <-net.queryReq:
debugLog("<-net.queryReq")
if !q.start(net) {
q.remote.deferQuery(q)
}

// Interacting with the table.
case f := <-net.tableOpReq:
debugLog("<-net.tableOpReq")
f()
net.tableOpResp <- struct{}{}

// Topic registration stuff.
case req := <-net.topicRegisterReq:
debugLog("<-net.topicRegisterReq")
if !req.add {
net.ticketStore.removeRegisterTopic(req.topic)
continue
}
net.ticketStore.addTopic(req.topic, true)
// If we're currently waiting idle (nothing to look up), give the ticket store a
// chance to start it sooner. This should speed up convergence of the radius
// determination for new topics.
// if topicRegisterLookupDone == nil {
if topicRegisterLookupTarget.target == (common.Hash{}) {
debugLog("topicRegisterLookupTarget == null")
if topicRegisterLookupTick.Stop() {
<-topicRegisterLookupTick.C
```

```
}
target, delay := net.ticketStore.nextRegisterLookup()
topicRegisterLookupTarget = target
topicRegisterLookupTick.Reset(delay)
}

case nodes := <-topicRegisterLookupDone:
debugLog("<-topicRegisterLookupDone")
net.ticketStore.registerLookupDone(topicRegisterLookupTarget, nodes, func(n *Node) []byte {
net.ping(n, n.addr())
return n.pingEcho
})
target, delay := net.ticketStore.nextRegisterLookup()
topicRegisterLookupTarget = target
topicRegisterLookupTick.Reset(delay)
topicRegisterLookupDone = nil

case <-topicRegisterLookupTick.C:
debugLog("<-topicRegisterLookupTick")
if (topicRegisterLookupTarget.target == common.Hash{}) {
target, delay := net.ticketStore.nextRegisterLookup()
topicRegisterLookupTarget = target
topicRegisterLookupTick.Reset(delay)
topicRegisterLookupDone = nil
} else {
topicRegisterLookupDone = make(chan []*Node)
target := topicRegisterLookupTarget.target
go func() { topicRegisterLookupDone <- net.lookup(target, false) }()
}

case <-nextRegisterTime:
debugLog("<-nextRegisterTime")
net.ticketStore.ticketRegistered(*nextTicket)
//fmt.Println("sendTopicRegister", nextTicket.t.node.addr().String(), nextTicket.t.topics,
nextTicket.idx, nextTicket.t.pong)
net.conn.sendTopicRegister(nextTicket.t.node, nextTicket.t.topics, nextTicket.idx,
nextTicket.t.pong)

case req := <-net.topicSearchReq:
if refreshDone == nil {
debugLog("<-net.topicSearchReq")
info, ok := searchInfo[req.topic]
```

```
if ok {
if req.delay == time.Duration(0) {
delete(searchInfo, req.topic)
net.ticketStore.removeSearchTopic(req.topic)
} else {
info.period = req.delay
searchInfo[req.topic] = info
}
continue
}
if req.delay != time.Duration(0) {
var info topicSearchInfo
info.period = req.delay
info.lookupChn = req.lookup
searchInfo[req.topic] = info
net.ticketStore.addSearchTopic(req.topic, req.found)
topicSearch <- req.topic
}
} else {
searchReqWhenRefreshDone = append(searchReqWhenRefreshDone, req)
}

case topic := <-topicSearch:
if activeSearchCount < maxSearchCount {
activeSearchCount++
target := net.ticketStore.nextSearchLookup(topic)
go func() {
nodes := net.lookup(target.target, false)
topicSearchLookupDone <- topicSearchResult{target: target, nodes: nodes}
}()
}
period := searchInfo[topic].period
if period != time.Duration(0) {
go func() {
time.Sleep(period)
topicSearch <- topic
}()
}

case res := <-topicSearchLookupDone:
activeSearchCount--
if lookupChn := searchInfo[res.target.topic].lookupChn; lookupChn != nil {
```

```
lookupChn <- net.ticketStore.radius[res.target.topic].converged
}
net.ticketStore.searchLookupDone(res.target, res.nodes, func(n *Node) []byte {
net.ping(n, n.addr())
return n.pingEcho
}, func(n *Node, topic Topic) []byte {
if n.state == known {
return net.conn.send(n, topicQueryPacket, topicQuery{Topic: topic}) // TODO: set expiration
} else {
if n.state == unknown {
net.ping(n, n.addr())
}
return nil
}
})

case <-statsDump.C:
debugLog("<-statsDump.C")
/*r, ok := net.ticketStore.radius[testTopic]
if !ok {
fmt.Printf("(%x) no radius @ %v\n", net.tab.self.ID[:8], time.Now())
} else {
topics := len(net.ticketStore.tickets)
tickets := len(net.ticketStore.nodes)
rad := r.radius / (maxRadius/10000+1)
fmt.Printf("(%x) topics:%d radius:%d tickets:%d @ %v\n", net.tab.self.ID[:8], topics, rad, tickets,
time.Now())
}*/

tm := mclock.Now()
for topic, r := range net.ticketStore.radius {
if printTestImgLogs {
rad := r.radius / (maxRadius/1000000 + 1)
minrad := r.minRadius / (maxRadius/1000000 + 1)
fmt.Printf("*R %d %v %016x %v\n", tm/1000000, topic, net.tab.self.sha[:8], rad)
fmt.Printf("*MR %d %v %016x %v\n", tm/1000000, topic, net.tab.self.sha[:8], minrad)
}
}
for topic, t := range net.topictab.topics {
wp := t.wcl.nextWaitPeriod(tm)
if printTestImgLogs {
fmt.Printf("*W %d %v %016x %d\n", tm/1000000, topic, net.tab.self.sha[:8], wp/1000000)
```

```
}
}

// Periodic / lookup-initiated bucket refresh.
case <-refreshTimer.C:
debugLog("<-refreshTimer.C")
// TODO: ideally we would start the refresh timer after
// fallback nodes have been set for the first time.
if refreshDone == nil {
refreshDone = make(chan struct{})
net.refresh(refreshDone)
}
case <-bucketRefreshTimer.C:
target := net.tab.chooseBucketRefreshTarget()
go func() {
net.lookup(target, false)
bucketRefreshTimer.Reset(bucketRefreshInterval)
}()
case newNursery := <-net.refreshReq:
debugLog("<-net.refreshReq")
if newNursery != nil {
net.nursery = newNursery
}
if refreshDone == nil {
refreshDone = make(chan struct{})
net.refresh(refreshDone)
}
net.refreshResp <- refreshDone
case <-refreshDone:
debugLog("<-net.refreshDone")
refreshDone = nil
list := searchReqWhenRefreshDone
searchReqWhenRefreshDone = nil
go func() {
for _, req := range list {
net.topicSearchReq <- req
}
}()
}
}
debugLog("loop stopped")
```

```
log.Debug(fmt.Sprintf("shutting down"))
if net.conn != nil {
net.conn.Close()
}
if refreshDone != nil {
// TODO: wait for pending refresh.
//<-refreshResults
}
// Cancel all pending timeouts.
for _, timer := range net.timeoutTimers {
timer.Stop()
}
if net.db != nil {
net.db.close()
}
close(net.closed)
}

// Everything below runs on the Network.loop goroutine
// and can modify Node, Table and Network at any time without locking.

func (net *Network) refresh(done chan<- struct{}) {
var seeds []*Node
if net.db != nil {
seeds = net.db.querySeeds(seedCount, seedMaxAge)
}
if len(seeds) == 0 {
seeds = net.nursery
}
if len(seeds) == 0 {
log.Trace(fmt.Sprint("no seed nodes found"))
close(done)
return
}
for _, n := range seeds {
log.Debug("", "msg", log.Lazy{Fn: func() string {
var age string
if net.db != nil {
age = time.Since(net.db.lastPong(n.ID)).String()
} else {
age = "unknown"
}
```

```
return fmt.Sprintf("seed node (age %s): %v", age, n)
}})
n = net.internNodeFromDB(n)
if n.state == unknown {
net.transition(n, verifyinit)
}
// Force-add the seed node so Lookup does something.
// It will be deleted again if verification fails.
net.tab.add(n)
}
// Start self lookup to fill up the buckets.
go func() {
net.Lookup(net.tab.self.ID)
close(done)
}()
}

// Node Interning.

func (net *Network) internNode(pkt *ingressPacket) *Node {
if n := net.nodes[pkt.remoteID]; n != nil {
n.IP = pkt.remoteAddr.IP
n.UDP = uint16(pkt.remoteAddr.Port)
n.TCP = uint16(pkt.remoteAddr.Port)
return n
}
n := NewNode(pkt.remoteID, pkt.remoteAddr.IP, uint16(pkt.remoteAddr.Port),
uint16(pkt.remoteAddr.Port))
n.state = unknown
net.nodes[pkt.remoteID] = n
return n
}

func (net *Network) internNodeFromDB(dbn *Node) *Node {
if n := net.nodes[dbn.ID]; n != nil {
return n
}
n := NewNode(dbn.ID, dbn.IP, dbn.UDP, dbn.TCP)
n.state = unknown
net.nodes[n.ID] = n
return n
}
```

```
func (net *Network) internNodeFromNeighbours(sender *net.UDPAddr, rn rpcNode) (n *Node, err
error) {
if rn.ID == net.tab.self.ID {
return nil, errors.New("is self")
}
if rn.UDP <= lowPort {
return nil, errors.New("low port")
}
n = net.nodes[rn.ID]
if n == nil {
// We haven't seen this node before.
n, err = nodeFromRPC(sender, rn)
if net.netrestrict != nil && !net.netrestrict.Contains(n.IP) {
return n, errors.New("not contained in netrestrict whitelist")
}
if err == nil {
n.state = unknown
net.nodes[n.ID] = n
}
return n, err
}
if !n.IP.Equal(rn.IP) || n.UDP != rn.UDP || n.TCP != rn.TCP {
err = fmt.Errorf("metadata mismatch: got %v, want %v", rn, n)
}
return n, err
}

// nodeNetGuts is embedded in Node and contains fields.
type nodeNetGuts struct {
// This is a cached copy of sha3(ID) which is used for node
// distance calculations. This is part of Node in order to make it
// possible to write tests that need a node at a certain distance.
// In those tests, the content of sha will not actually correspond
// with ID.
sha common.Hash

// State machine fields. Access to these fields
// is restricted to the Network.loop goroutine.
state             *nodeState
pingEcho          []byte           // hash of last ping sent by us
pingTopics        []Topic          // topic set sent by us in last ping
```

```
deferredQueries   []*findnodeQuery // queries that can't be sent yet
pendingNeighbours *findnodeQuery   // current query, waiting for reply
queryTimeouts     int
}

func (n *nodeNetGuts) deferQuery(q *findnodeQuery) {
n.deferredQueries = append(n.deferredQueries, q)
}

func (n *nodeNetGuts) startNextQuery(net *Network) {
if len(n.deferredQueries) == 0 {
return
}
nextq := n.deferredQueries[0]
if nextq.start(net) {
n.deferredQueries = append(n.deferredQueries[:0], n.deferredQueries[1:]...)
}
}

func (q *findnodeQuery) start(net *Network) bool {
// Satisfy queries against the local node directly.
if q.remote == net.tab.self {
closest := net.tab.closest(crypto.Keccak256Hash(q.target[:]), bucketSize)
q.reply <- closest.entries
return true
}
if q.remote.state.canQuery && q.remote.pendingNeighbours == nil {
net.conn.sendFindnodeHash(q.remote, q.target)
net.timedEvent(respTimeout, q.remote, neighboursTimeout)
q.remote.pendingNeighbours = q
return true
}
// If the node is not known yet, it won't accept queries.
// Initiate the transition to known.
// The request will be sent later when the node reaches known state.
if q.remote.state == unknown {
net.transition(q.remote, verifyinit)
}
return false
}

// Node Events (the input to the state machine).
```

```
type nodeEvent uint

//go:generate stringer -type=nodeEvent

const (
invalidEvent nodeEvent = iota // zero is reserved

// Packet type events.
// These correspond to packet types in the UDP protocol.
pingPacket
pongPacket
findnodePacket
neighborsPacket
findnodeHashPacket
topicRegisterPacket
topicQueryPacket
topicNodesPacket

// Non-packet events.
// Event values in this category are allocated outside
// the packet type range (packet types are encoded as a single byte).
pongTimeout nodeEvent = iota + 256
pingTimeout
neighboursTimeout
)

// Node State Machine.

type nodeState struct {
name     string
handle   func(*Network, *Node, nodeEvent, *ingressPacket) (next *nodeState, err error)
enter    func(*Network, *Node)
canQuery bool
}

func (s *nodeState) String() string {
return s.name
}

var (
unknown          *nodeState
```

```
verifyinit       *nodeState
verifywait       *nodeState
remoteverifywait *nodeState
known            *nodeState
contested        *nodeState
unresponsive     *nodeState
)

func init() {
unknown = &nodeState{
name: "unknown",
enter: func(net *Network, n *Node) {
net.tab.delete(n)
n.pingEcho = nil
// Abort active queries.
for _, q := range n.deferredQueries {
q.reply <- nil
}
n.deferredQueries = nil
if n.pendingNeighbours != nil {
n.pendingNeighbours.reply <- nil
n.pendingNeighbours = nil
}
n.queryTimeouts = 0
},
handle: func(net *Network, n *Node, ev nodeEvent, pkt *ingressPacket) (*nodeState, error) {
switch ev {
case pingPacket:
net.handlePing(n, pkt)
net.ping(n, pkt.remoteAddr)
return verifywait, nil
default:
return unknown, errInvalidEvent
}
},
}

verifyinit = &nodeState{
name: "verifyinit",
enter: func(net *Network, n *Node) {
net.ping(n, n.addr())
},
```

```
handle: func(net *Network, n *Node, ev nodeEvent, pkt *ingressPacket) (*nodeState, error) {
switch ev {
case pingPacket:
net.handlePing(n, pkt)
return verifywait, nil
case pongPacket:
err := net.handleKnownPong(n, pkt)
return remoteverifywait, err
case pongTimeout:
return unknown, nil
default:
return verifyinit, errInvalidEvent
}
},
}

verifywait = &nodeState{
name: "verifywait",
handle: func(net *Network, n *Node, ev nodeEvent, pkt *ingressPacket) (*nodeState, error) {
switch ev {
case pingPacket:
net.handlePing(n, pkt)
return verifywait, nil
case pongPacket:
err := net.handleKnownPong(n, pkt)
return known, err
case pongTimeout:
return unknown, nil
default:
return verifywait, errInvalidEvent
}
},
}

remoteverifywait = &nodeState{
name: "remoteverifywait",
enter: func(net *Network, n *Node) {
net.timedEvent(respTimeout, n, pingTimeout)
},
handle: func(net *Network, n *Node, ev nodeEvent, pkt *ingressPacket) (*nodeState, error) {
switch ev {
case pingPacket:
```

```
net.handlePing(n, pkt)
return remoteverifywait, nil
case pingTimeout:
return known, nil
default:
return remoteverifywait, errInvalidEvent
}
},
}

known = &nodeState{
name:     "known",
canQuery: true,
enter: func(net *Network, n *Node) {
n.queryTimeouts = 0
n.startNextQuery(net)
// Insert into the table and start revalidation of the last node
// in the bucket if it is full.
last := net.tab.add(n)
if last != nil && last.state == known {
// TODO: do this asynchronously
net.transition(last, contested)
}
},
handle: func(net *Network, n *Node, ev nodeEvent, pkt *ingressPacket) (*nodeState, error) {
switch ev {
case pingPacket:
net.handlePing(n, pkt)
return known, nil
case pongPacket:
err := net.handleKnownPong(n, pkt)
return known, err
default:
return net.handleQueryEvent(n, ev, pkt)
}
},
}

contested = &nodeState{
name:     "contested",
canQuery: true,
enter: func(net *Network, n *Node) {
```

```
net.ping(n, n.addr())
},
handle: func(net *Network, n *Node, ev nodeEvent, pkt *ingressPacket) (*nodeState, error) {
switch ev {
case pongPacket:
// Node is still alive.
err := net.handleKnownPong(n, pkt)
return known, err
case pongTimeout:
net.tab.deleteReplace(n)
return unresponsive, nil
case pingPacket:
net.handlePing(n, pkt)
return contested, nil
default:
return net.handleQueryEvent(n, ev, pkt)
}
},
}

unresponsive = &nodeState{
name:     "unresponsive",
canQuery: true,
handle: func(net *Network, n *Node, ev nodeEvent, pkt *ingressPacket) (*nodeState, error) {
switch ev {
case pingPacket:
net.handlePing(n, pkt)
return known, nil
case pongPacket:
err := net.handleKnownPong(n, pkt)
return known, err
default:
return net.handleQueryEvent(n, ev, pkt)
}
},
}
}

// handle processes packets sent by n and events related to n.
func (net *Network) handle(n *Node, ev nodeEvent, pkt *ingressPacket) error {
//fmt.Println("handle", n.addr().String(), n.state, ev)
if pkt != nil {
```

```
if err := net.checkPacket(n, ev, pkt); err != nil {
//fmt.Println("check err:", err)
return err
}
// Start the background expiration goroutine after the first
// successful communication. Subsequent calls have no effect if it
// is already running. We do this here instead of somewhere else
// so that the search for seed nodes also considers older nodes
// that would otherwise be removed by the expirer.
if net.db != nil {
net.db.ensureExpirer()
}
}
if n.state == nil {
n.state = unknown //???
}
next, err := n.state.handle(net, n, ev, pkt)
net.transition(n, next)
//fmt.Println("new state:", n.state)
return err
}

func (net *Network) checkPacket(n *Node, ev nodeEvent, pkt *ingressPacket) error {
// Replay prevention checks.
switch ev {
case pingPacket, findnodeHashPacket, neighborsPacket:
// TODO: check date is > last date seen
// TODO: check ping version
case pongPacket:
if !bytes.Equal(pkt.data.(*pong).ReplyTok, n.pingEcho) {
// fmt.Println("pong reply token mismatch")
return fmt.Errorf("pong reply token mismatch")
}
n.pingEcho = nil
}
// Address validation.
// TODO: Ideally we would do the following:
//  - reject all packets with wrong address except ping.
//  - for ping with new address, transition to verifywait but keep the
//    previous node (with old address) around. if the new one reaches known,
//    swap it out.
return nil
```

```
}

func (net *Network) transition(n *Node, next *nodeState) {
if n.state != next {
n.state = next
if next.enter != nil {
next.enter(net, n)
}
}

// TODO: persist/unpersist node
}

func (net *Network) timedEvent(d time.Duration, n *Node, ev nodeEvent) {
timeout := timeoutEvent{ev, n}
net.timeoutTimers[timeout] = time.AfterFunc(d, func() {
select {
case net.timeout <- timeout:
case <-net.closed:
}
})
}

func (net *Network) abortTimedEvent(n *Node, ev nodeEvent) {
timer := net.timeoutTimers[timeoutEvent{ev, n}]
if timer != nil {
timer.Stop()
delete(net.timeoutTimers, timeoutEvent{ev, n})
}
}

func (net *Network) ping(n *Node, addr *net.UDPAddr) {
//fmt.Println("ping", n.addr().String(), n.ID.String(), n.sha.Hex())
if n.pingEcho != nil || n.ID == net.tab.self.ID {
//fmt.Println(" not sent")
return
}
debugLog(fmt.Sprintf("ping(node = %x)", n.ID[:8]))
n.pingTopics = net.ticketStore.regTopicSet()
n.pingEcho = net.conn.sendPing(n, addr, n.pingTopics)
net.timedEvent(respTimeout, n, pongTimeout)
}
```

```
func (net *Network) handlePing(n *Node, pkt *ingressPacket) {
debugLog(fmt.Sprintf("handlePing(node = %x)", n.ID[:8]))
ping := pkt.data.(*ping)
n.TCP = ping.From.TCP
t := net.topictab.getTicket(n, ping.Topics)

pong := &pong{
To:         makeEndpoint(n.addr(), n.TCP), // TODO: maybe use known TCP port from DB
ReplyTok:   pkt.hash,
Expiration: uint64(time.Now().Add(expiration).Unix()),
}
ticketToPong(t, pong)
net.conn.send(n, pongPacket, pong)
}

func (net *Network) handleKnownPong(n *Node, pkt *ingressPacket) error {
debugLog(fmt.Sprintf("handleKnownPong(node = %x)", n.ID[:8]))
net.abortTimedEvent(n, pongTimeout)
now := mclock.Now()
ticket, err := pongToTicket(now, n.pingTopics, n, pkt)
if err == nil {
// fmt.Printf("(%x) ticket: %+v\n", net.tab.self.ID[:8], pkt.data)
net.ticketStore.addTicket(now, pkt.data.(*pong).ReplyTok, ticket)
} else {
debugLog(fmt.Sprintf(" error: %v", err))
}

n.pingEcho = nil
n.pingTopics = nil
return err
}

func (net *Network) handleQueryEvent(n *Node, ev nodeEvent, pkt *ingressPacket) (*nodeState,
error) {
switch ev {
case findnodePacket:
target := crypto.Keccak256Hash(pkt.data.(*findnode).Target[:])
results := net.tab.closest(target, bucketSize).entries
net.conn.sendNeighbours(n, results)
return n.state, nil
case neighborsPacket:
```

```
err := net.handleNeighboursPacket(n, pkt)
return n.state, err
case neighboursTimeout:
if n.pendingNeighbours != nil {
n.pendingNeighbours.reply <- nil
n.pendingNeighbours = nil
}
n.queryTimeouts++
if n.queryTimeouts > maxFindnodeFailures && n.state == known {
return contested, errors.New("too many timeouts")
}
return n.state, nil

// v5

case findnodeHashPacket:
results := net.tab.closest(pkt.data.(*findnodeHash).Target, bucketSize).entries
net.conn.sendNeighbours(n, results)
return n.state, nil
case topicRegisterPacket:
//fmt.Println("got topicRegisterPacket")
regdata := pkt.data.(*topicRegister)
pong, err := net.checkTopicRegister(regdata)
if err != nil {
//fmt.Println(err)
return n.state, fmt.Errorf("bad waiting ticket: %v", err)
}
net.topictab.useTicket(n, pong.TicketSerial, regdata.Topics, int(regdata.Idx), pong.Expiration,
pong.WaitPeriods)
return n.state, nil
case topicQueryPacket:
// TODO: handle expiration
topic := pkt.data.(*topicQuery).Topic
results := net.topictab.getEntries(topic)
if _, ok := net.ticketStore.tickets[topic]; ok {
results = append(results, net.tab.self) // we're not registering in our own table but if we're
advertising, return ourselves too
}
if len(results) > 10 {
results = results[:10]
}
var hash common.Hash
```

```
copy(hash[:], pkt.hash)
net.conn.sendTopicNodes(n, hash, results)
return n.state, nil
case topicNodesPacket:
p := pkt.data.(*topicNodes)
if net.ticketStore.gotTopicNodes(n, p.Echo, p.Nodes) {
n.queryTimeouts++
if n.queryTimeouts > maxFindnodeFailures && n.state == known {
return contested, errors.New("too many timeouts")
}
}
return n.state, nil

default:
return n.state, errInvalidEvent
}
}

func (net *Network) checkTopicRegister(data *topicRegister) (*pong, error) {
var pongpkt ingressPacket
if err := decodePacket(data.Pong, &pongpkt); err != nil {
return nil, err
}
if pongpkt.ev != pongPacket {
return nil, errors.New("is not pong packet")
}
if pongpkt.remoteID != net.tab.self.ID {
return nil, errors.New("not signed by us")
}
// check that we previously authorised all topics
// that the other side is trying to register.
if rlpHash(data.Topics) != pongpkt.data.(*pong).TopicHash {
return nil, errors.New("topic hash mismatch")
}
if data.Idx < 0 || int(data.Idx) >= len(data.Topics) {
return nil, errors.New("topic index out of range")
}
return pongpkt.data.(*pong), nil
}

func rlpHash(x interface{}) (h common.Hash) {
hw := sha3.NewKeccak256()
```

```
rlp.Encode(hw, x)
hw.Sum(h[:0])
return h
}

func (net *Network) handleNeighboursPacket(n *Node, pkt *ingressPacket) error {
if n.pendingNeighbours == nil {
return errNoQuery
}
net.abortTimedEvent(n, neighboursTimeout)

req := pkt.data.(*neighbors)
nodes := make([]*Node, len(req.Nodes))
for i, rn := range req.Nodes {
nn, err := net.internNodeFromNeighbours(pkt.remoteAddr, rn)
if err != nil {
log.Debug(fmt.Sprintf("invalid neighbour (%v) from %x@%v: %v", rn.IP, n.ID[:8], pkt.remoteAddr,
err))
continue
}
nodes[i] = nn
// Start validation of query results immediately.
// This fills the table quickly.
// TODO: generates way too many packets, maybe do it via queue.
if nn.state == unknown {
net.transition(nn, verifyinit)
}
}
// TODO: don't ignore second packet
n.pendingNeighbours.reply <- nodes
n.pendingNeighbours = nil
// Now that this query is done, start the next one.
n.startNextQuery(net)
return nil
}

95:F:\git\coin\ethereum\go-ethereum\p2p\discv5\net_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discv5

import (
```

```
"fmt"
"net"
"testing"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
)

func TestNetwork_Lookup(t *testing.T) {
key, _ := crypto.GenerateKey()
network, err := newNetwork(lookupTestnet, key.PublicKey, nil, "", nil)
if err != nil {
t.Fatal(err)
}
lookupTestnet.net = network
defer network.Close()

// lookup on empty table returns no nodes
// if results := network.Lookup(lookupTestnet.target, false); len(results) > 0 {
// t.Fatalf("lookup on empty table returned %d results: %#v", len(results), results)
// }
// seed table with initial node (otherwise lookup will terminate immediately)
seeds := []*Node{NewNode(lookupTestnet.dists[256][0], net.IP{10, 0, 2, 99}, lowPort+256, 999)}
if err := network.SetFallbackNodes(seeds); err != nil {
t.Fatal(err)
}
time.Sleep(3 * time.Second)

results := network.Lookup(lookupTestnet.target)
t.Logf("results:")
for _, e := range results {
t.Logf("  ld=%d, %x", logdist(lookupTestnet.targetSha, e.sha), e.sha[:])
}
if len(results) != bucketSize {
t.Errorf("wrong number of results: got %d, want %d", len(results), bucketSize)
}
if hasDuplicates(results) {
t.Errorf("result set contains duplicate entries")
}
if !sortedByDistanceTo(lookupTestnet.targetSha, results) {
t.Errorf("result set not sorted by distance to target")
```

```
}
// TODO: check result nodes are actually closest
}

// This is the test network for the Lookup test.
// The nodes were obtained by running testnet.mine with a random NodeID as target.
var lookupTestnet = &preminedTestnet{
target:
MustHexID("166aea4f556532c6d34e8b740e5d314af7e9ac0ca79833bd751d6b665f12dfd38ec563
c363b32f02aef4a80b44fd3def94612d497b99cb5f17fd24de454927ec"),
targetSha: common.Hash{0x5c, 0x94, 0x4e, 0xe5, 0x1c, 0x5a, 0xe9, 0xf7, 0x2a, 0x95, 0xec, 0xcb,
0x8a, 0xed, 0x3, 0x74, 0xee, 0xcb, 0x51, 0x19, 0xd7, 0x20, 0xcb, 0xea, 0x68, 0x13, 0xe8, 0xe0,
0xd6, 0xad, 0x92, 0x61},
dists: [257][]NodeID{
240: {
MustHexID("2001ad5e3e80c71b952161bc0186731cf5ffe942d24a79230a0555802296238e57ea7a
32f5b6f18564eadc1c65389448481f8c9338df0a3dbd18f708cbc2cbcb"),
MustHexID("6ba3f4f57d084b6bf94cc4555b8c657e4a8ac7b7baf23c6874efc21dd1e4f56b7eb2721
e07f5242d2f1d8381fc8cae535e860197c69236798ba1ad231b105794"),
},
244: {
MustHexID("696ba1f0a9d55c59246f776600542a9e6432490f0cd78f8bb55a196918df2081a9b521c
3c3ba48e465a75c10768807717f8f689b0b4adce00e1c75737552a178"),
},
246: {
MustHexID("d6d32178bdc38416f46ffb8b3ec9e4cb2cfff8d04dd7e4311a70e403cb62b10be1b4473
11b60b4f9ee221a8131fc2cbd45b96dd80deba68a949d467241facfa8"),
MustHexID("3ea3d04a43a3dfb5ac11cffc2319248cf41b6279659393c2f55b8a0a5fc9d12581a9d97
ef5d8ff9b5abf3321a290e8f63a4f785f450dc8a672aba3ba2ff4fdab"),
MustHexID("2fc897f05ae585553e5c014effd3078f84f37f9333afacffb109f00ca8e7a3373de810a394
6be971cbccdfd40249f9fe7f322118ea459ac71acca85a1ef8b7f4"),
},
247: {
MustHexID("3155e1427f85f10a5c9a7755877748041af1bcd8d474ec065eb33df57a97babf54bfd21
03575fa829115d224c523596b401065a97f74010610fce76382c0bf32"),
MustHexID("312c55512422cf9b8a4097e9a6ad79402e87a15ae909a4bfefa22398f03d20951933be
ea1e4dfa6f968212385e829f04c2d314fc2d4e255e0d3bc08792b069db"),
MustHexID("38643200b172dcfef857492156971f0e6aa2c538d8b74010f8e140811d53b98c765dd2
d96126051913f44582e8c199ad7c6d6819e9a56483f637feaac9448aac"),
MustHexID("8dcab8618c3253b558d459da53bd8fa68935a719aff8b811197101a4b2b47dd2d4729
5286fc00cc081bb542d760717d1bdd6bec2c37cd72eca367d6dd3b9df73"),
MustHexID("8b58c6073dd98bbad4e310b97186c8f822d3a5c7d57af40e2136e88e315afd115edb27
```

d2d0685a908cfe5aa49d0debdda6e6e63972691d6bd8c5af2d771dd2a9"),
MustHexID("2cbb718b7dc682da19652e7d9eb4fefaf7b7147d82c1c2b6805edf77b85e29fde9f6da1
95741467ff2638dc62c8d3e014ea5686693c15ed0080b6de90354c137"),
MustHexID("e84027696d3f12f2de30a9311afea8fbd313c2360daff52bb5fc8c7094d5295758bec313
4e4eef24e4cdf377b40da344993284628a7a346eba94f74160998feb"),
MustHexID("f1357a4f04f9d33753a57c0b65ba20a5d8777abbffd04e906014491c9103fb08590e455
48d37aa4bd70965e2e81ddba94f31860348df01469eec8c1829200a68"),
MustHexID("4ab0a75941b12892369b4490a1928c8ca52a9ad6d3dffbd1d8c0b907bc200fe74c022d
011ec39b64808a39c0ca41f1d3254386c3e7733e7044c44259486461b6"),
MustHexID("d45150a72dc74388773e68e03133a3b5f51447fe91837d566706b3c035ee4b56f160c8
78c6273394daee7f56cc398985269052f22f75a8057df2fe6172765354"),
},
248: {
MustHexID("6aadfce366a189bab08ac84721567483202c86590642ea6d6a14f37ca78d82bdb6509
eb7b8b2f6f63c78ae3ae1d8837c89509e41497d719b23ad53dd81574afa"),
MustHexID("a605ecfd6069a4cf4cf7f5840e5bc0ce10d23a3ac59e2aaa70c6afd5637359d2519b452
4f56fc2ca180cdbebe54262f720ccaae8c1b28fd553c485675831624d"),
MustHexID("29701451cb9448ca33fc33680b44b840d815be90146eb521641efbffed0859c154e889
2d3906eae9934bfacee72cd1d2fa9dd050fd18888eea49da155ab0efd2"),
MustHexID("3ed426322dee7572b08592e1e079f8b6c6b30e10e6243edd144a6a48fdbdb83df73a6
e41b1143722cb82604f2203a32758610b5d9544f44a1a7921ba001528c1"),
MustHexID("b2e2a2b7fdd363572a3256e75435fab1da3b16f7891a8bd2015f30995dae665d7eabfd
194d87d99d5df628b4bbc7b04e5b492c596422dd8272746c7a1b0b8e4f"),
MustHexID("0c69c9756162c593e85615b814ce57a2a8ca2df6c690b9c4e4602731b61e1531a3bbe
3f7114271554427ffabea80ad8f36fa95a49fa77b675ae182c6ccac1728"),
MustHexID("8d28be21d5a97b0876442fa4f5e5387f5bf3faad0b6f13b8607b64d6e448c0991ca28dd
7fe2f64eb8eadd7150bff5d5666aa6ed868b84c71311f4ba9a38569dd"),
MustHexID("2c677e1c64b9c9df6359348a7f5f33dc79e22f0177042486d125f8b6ca7f0dc756b1f672
aceee5f1746bcff80aaf6f92a8dc0c9fbeb259b3fa0da060de5ab7e8"),
MustHexID("3994880f94a8678f0cd247a43f474a8af375d2a072128da1ad6cae84a244105ff85e94fc
7d8496f639468de7ee998908a91c7e33ef7585fff92e984b210941a1"),
MustHexID("b45a9153c08d002a48090d15d61a7c7dad8c2af85d4ff5bd36ce23a9a11e0709bf8d56
614c7b193bc028c16cbf7f20dfbcc751328b64a924995d47b41e452422"),
MustHexID("057ab3a9e53c7a84b0f3fc586117a525cdd18e313f52a67bf31798d48078e325abe5cfe
e3f6c2533230cb37d0549289d692a29dd400e899b8552d4b928f6f907"),
MustHexID("0ddf663d308791eb92e6bd88a2f8cb45e4f4f35bb16708a0e6ff7f1362aa6a73fedd0a1b
1557fb3365e38e1b79d6918e2fae2788728b70c9ab6b51a3b94a4338"),
MustHexID("f637e07ff50cc1e3731735841c4798411059f2023abcf3885674f3e8032531b0edca50fd
715df6feb489b6177c345374d64f4b07d257a7745de393a107b013a5"),
MustHexID("e24ec7c6eec094f63c7b3239f56d311ec5a3e45bc4e622a1095a65b95eea6fe13e29f3
b6b7a2cbfe40906e3989f17ac834c3102dd0cadaaa26e16ee06d782b72"),
MustHexID("b76ea1a6fd6506ef6e3506a4f1f60ed6287fff8114af6141b2ff13e61242331b54082b023

cfea5b3083354a4fb3f9eb8be01fb4a518f579e731a5d0707291a6b"),
MustHexID("9b53a37950ca8890ee349b325032d7b672cab7eced178d3060137b24ef6b92a439779
22d5bdfb4a3409a2d80128e02f795f9dae6d7d99973ad0e23a2afb8442f"),
},
249: {
MustHexID("675ae65567c3c72c50c73bc0fd4f61f202ea5f93346ca57b551de3411ccc614fad61cb9
035493af47615311b9d44ee7a161972ee4d77c28fe1ec029d01434e6a"),
MustHexID("8eb81408389da88536ae5800392b16ef5109d7ea132c18e9a82928047ecdb502693f6
e4a4cdd18b54296caf561db937185731456c456c98bfe7de0baf0eaa495"),
MustHexID("2adba8b1612a541771cb93a726a38a4b88e97b18eced2593eb7daf82f05a5321ca94a
72cc780c306ff21e551a932fc2c6d791e4681907b5ceab7f084c3fa2944"),
MustHexID("b1b4bfbda514d9b8f35b1c28961da5d5216fe50548f4066f69af3b7666a3b2e06eac646
735e963e5c8f8138a2fb95af15b13b23ff00c6986eccc0efaa8ee6fb4"),
MustHexID("d2139281b289ad0e4d7b4243c4364f5c51aac8b60f4806135de06b12b5b369c9e43a6
eb494eab860d115c15c6fbb8c5a1b0e382972e0e460af395b8385363de7"),
MustHexID("4a693df4b8fc5bdc7cec342c3ed2e228d7c5b4ab7321ddaa6cccbeb45b05a9f1d95766
b4002e6d4791c2deacb8a667aadea6a700da28a3eea810a30395701bbc"),
MustHexID("ab41611195ec3c62bb8cd762ee19fb182d194fd141f4a66780efbef4b07ce916246c022
b841237a3a6b512a93431157edd221e854ed2a259b72e9c5351f44d0c"),
MustHexID("68e8e26099030d10c3c703ae7045c0a48061fb88058d853b3e67880014c449d431101
4da99d617d3150a20f1a3da5e34bf0f14f1c51fe4dd9d58afd222823176"),
MustHexID("3fbcacf546fb129cd70fc48de3b593ba99d3c473798bc309292aca280320e0eacc04442
c914cad5c4cf6950345ba79b0d51302df88285d4e83ee3fe41339eee7"),
MustHexID("1d4a623659f7c8f80b6c3939596afdf42e78f892f682c768ad36eb7bfba402dbf97aea3a
268f3badd8fe7636be216edf3d67ee1e08789ebbc7be625056bd7109"),
MustHexID("a283c474ab09da02bbc96b16317241d0627646fcc427d1fe790b76a7bf1989ced90f92
101a973047ae9940c92720dffbac8eff21df8cae468a50f72f9e159417"),
MustHexID("dbf7e5ad7f87c3dfecae65d87c3039e14ed0bdc56caf00ce81931073e2e16719d74629
5512ff7937a15c3b03603e7c41a4f9df94fcd37bb200dd8f332767e9cb"),
MustHexID("caaa070a26692f64fc77f30d7b5ae980d419b4393a0f442b1c821ef58c0862898b0d22f
74a4f8c5d83069493e3ec0b92f17dc1fe6e4cd437c1ec25039e7ce839"),
MustHexID("874cc8d1213beb65c4e0e1de38ef5d8165235893ac74ab5ea937c885eaab25c8d79da
d0456e9fd3e9450626cac7e107b004478fb59842f067857f39a47cee695"),
MustHexID("d94193f236105010972f5df1b7818b55846592a0445b9cdc4eaed811b8c4c0f7c27dc8
cc9837a4774656d6b34682d6d329d42b6ebb55da1d475c2474dc3dfdf4"),
MustHexID("edd9af6aded4094e9785637c28fccbd3980cbe28e2eb9a411048a23c2ace4bd6b0b70
88a7817997b49a3dd05fc6929ca6c7abbb69438dbdabe65e971d2a794b2"),
},
250: {
MustHexID("53a5bd1215d4ab709ae8fdc2ced50bba320bced78bd9c5dc92947fb402250c9148917
86db0978c898c058493f86fc68b1c5de8a5cb36336150ac7a88655b6c39"),
MustHexID("b7f79e3ab59f79262623c9ccefc8f01d682323aee56ffbe295437487e9d5acaf556a9c92

e1f1c6a9601f2b9eb6b027ae1aeaebac71d61b9b78e88676efd3e1a3"),
MustHexID("d374bf7e8d7ffff69cc00bebff38ef5bc1dcb0a8d51c1a3d70e61ac6b2e2d6617109254b
0ac224354dfbf79009fe4239e09020c483cc60c071e00b9238684f30"),
MustHexID("1e1eac1c9add703eb252eb991594f8f5a173255d526a855fab24ae57dc277e055bc3c7
a7ae0b45d437c4f47a72d97eb7b126f2ba344ba6c0e14b2c6f27d4b1e6"),
MustHexID("ae28953f63d4bc4e706712a59319c111f5ff8f312584f65d7436b4cd3d14b217b958f848
6bad666b4481fe879019fb1f767cf15b3e3e2711efc33b56d460448a"),
MustHexID("934bb1edf9c7a318b82306aca67feb3d6b434421fa275d694f0b4927afd8b1d3935b72
7fd4ff6e3d012e0c82f1824385174e8c6450ade59c2a43281a4b3446b6"),
MustHexID("9eef3f28f70ce19637519a0916555bf76d26de31312ac656cf9d3e379899ea44e4dd7ffc
ce923b4f3563f8a00489a34bd6936db0cbb4c959d32c49f017e07d05"),
MustHexID("82200872e8f871c48f1fad13daec6478298099b591bb3dbc4ef6890aa28ebee5860d07
d70be62f4c0af85085a90ae8179ee8f937cf37915c67ea73e704b03ee7"),
MustHexID("6c75a5834a08476b7fc37ff3dc2011dc3ea3b36524bad7a6d319b18878fad813c0ba76
d1f4555cacd3890c865438c21f0e0aed1f80e0a157e642124c69f43a11"),
MustHexID("995b873742206cb02b736e73a88580c2aacb0bd4a3c97a647b647bcab3f5e03c0e073
6520a8b3600da09edf4248991fb01091ec7ff3ec7cdc8a1beae011e7aae"),
MustHexID("c773a056594b5cdef2e850d30891ff0e927c3b1b9c35cd8e8d53a1017001e237468e1e
ce3ae33d612ca3e6abb0a9169aa352e9dcda358e5af2ad982b577447db"),
MustHexID("2b46a5f6923f475c6be99ec6d134437a6d11f6bb4b4ac6bcd94572fa1092639d1c08ae
efcb51f0912f0a060f71d4f38ee4da70ecc16010b05dd4a674aab14c3a"),
MustHexID("af6ab501366debbaa0d22e20e9688f32ef6b3b644440580fd78de4fe0e99e2a16eb563
6bbae0d1c259df8ddda77b35b9a35cbc36137473e9c68fbc9d203ba842"),
MustHexID("c9f6f2dd1a941926f03f770695bda289859e85fabaf94baaae20b93e5015dc014ba4115
0176a36a1884adb52f405194693e63b0c464a6891cc9cc1c80d450326"),
MustHexID("5b116f0751526868a909b61a30b0c5282c37df6925cc03ddea556ef0d0602a9595fd6c
14d371f8ed7d45d89918a032dcd22be4342a8793d88fdbeb3ca3d75bd7"),
MustHexID("50f3222fb6b82481c7c813b2172e1daea43e2710a443b9c2a57a12bd160dd37e20f87a
a968c82ad639af6972185609d47036c0d93b4b7269b74ebd7073221c10"),
},
251: {
MustHexID("9b8f702a62d1bee67bedfeb102eca7f37fa1713e310f0d6651cc0c33ea7c5477575289c
cd463e5a2574a00a676a1fdce05658ba447bb9d2827f0ba47b947e894"),
MustHexID("b97532eb83054ed054b4abdf413bb30c00e4205545c93521554dbe77faa3cfaa5bd31e
f466a107b0b34a71ec97214c0c83919720142cddac93aa7a3e928d4708"),
MustHexID("2f7a5e952bfb67f2f90b8441b5fadc9ee13b1dcde3afeeb3dd64bf937f86663cc5c55d1fa
83952b5422763c7df1b7f2794b751c6be316ebc0beb4942e65ab8c1"),
MustHexID("42c7483781727051a0b3660f14faf39e0d33de5e643702ae933837d036508ab856ce7
eec8ec89c4929a4901256e5233a3d847d5d4893f91bcf21835a9a880fee"),
MustHexID("873bae27bf1dc854408fba94046a53ab0c965cebe1e4e12290806fc62b88deb1f4a47f9
e18f78fc0e7913a0c6e42ac4d0fc3a20cea6bc65f0c8a0ca90b67521e"),
MustHexID("a7e3a370bbd761d413f8d209e85886f68bf73d5c3089b2dc6fa42aab1ecb5162635497

eed95dee2417f3c9c74a3e76319625c48ead2e963c7de877cd4551f347"),
MustHexID("528597534776a40df2addaaea15b6ff832ce36b9748a265768368f657e76d58569d9f3
0dbb91e91cf0ae7efe8f402f17aa0ae15f5c55051ba03ba830287f4c42"),
MustHexID("461d8bd4f13c3c09031fdb84f104ed737a52f630261463ce0bdb5704259bab4b737dda
688285b8444dbecaecad7f50f835190b38684ced5e90c54219e5adf1bc"),
MustHexID("6ec50c0be3fd232737090fc0111caaf0bb6b18f72be453428087a11a97fd6b52db0344a
cbf789a689bd4f5f50f79017ea784f8fd6fe723ad6ae675b9e3b13e21"),
MustHexID("12fc5e2f77a83fdcc727b79d8ae7fe6a516881138d3011847ee136b400fed7cfba1f53fd
7a9730253c7aa4f39abeacd04f138417ba7fcb0f36cccc3514e0dab6"),
MustHexID("4fdbe75914ccd0bce02101606a1ccf3657ec963e3b3c20239d5fec87673fe446d649b4f
15f1fe1a40e6cfbd446dda2d31d40bb602b1093b8fcd5f139ba0eb46a"),
MustHexID("3753668a0f6281e425ea69b52cb2d17ab97afbe6eb84cf5d25425bc5e5300938885764
0668fadd7c110721e6047c9697803bd8a6487b43bb343bfa32ebf24039"),
MustHexID("2e81b16346637dec4410fd88e527346145b9c0a849dbf2628049ac7dae016c8f430564
9d5659ec77f1e8a0fac0db457b6080547226f06283598e3740ad94849a"),
MustHexID("802c3cc27f91c89213223d758f8d2ecd41135b357b6d698f24d811cdf113033a81c38e
0bdff574a5c005b00a8c193dc2531f8c1fa05fa60acf0ab6f2858af09f"),
MustHexID("fcc9a2e1ac3667026ff16192876d1813bb75abdbf39b929a92863012fe8b1d890badea7
a0de36274d5c1eb1e8f975785532c50d80fd44b1a4b692f437303393f"),
MustHexID("6d8b3efb461151dd4f6de809b62726f5b89e9b38e9ba1391967f61cde844f7528fecf82
1b74049207cee5a527096b31f3ad623928cd3ce51d926fa345a6b2951"),
},
252: {
MustHexID("f1ae93157cc48c2075dd5868fbf523e79e06caf4b8198f352f6e526680b78ff4227263de
92612f7d63472bd09367bb92a636fff16fe46ccf41614f7a72495c2a"),
MustHexID("587f482d111b239c27c0cb89b51dd5d574db8efd8de14a2e6a1400c54d4567e77c65f8
9c1da52841212080b91604104768350276b6682f2f961cdaf4039581c7"),
MustHexID("e3f88274d35cefdaabdf205afe0e80e936cc982b8e3e47a84ce664c413b29016a4fb4f3
a3ebae0a2f79671f8323661ed462bf4390af94c424dc8ace0c301b90f"),
MustHexID("0ddc736077da9a12ba410dc5ea63cbcbe7659dd08596485b2bff3435221f82c10d263e
fd9af938e128464be64a178b7cd22e19f400d5802f4c9df54bf89f2619"),
MustHexID("784aa34d833c6ce63fcc1279630113c3272e82c4ae8c126c5a52a88ac461b6baeed42
44e607b05dc14e5b2f41c70a273c3804dea237f14f7a1e546f6d1309d14"),
MustHexID("f253a2c354ee0e27cfcae786d726753d4ad24be6516b279a936195a487de4a59dbc29
6accf20463749ff55293263ed8c1b6365eecb248d44e75e9741c0d18205"),
MustHexID("a1910b80357b3ad9b4593e0628922939614dc9056a5fbf477279c8b2c1d0b4b31d89a
0c09d0d41f795271d14d3360ef08a3f821e65e7e1f56c07a36afe49c7c5"),
MustHexID("f1168552c2efe541160f0909b0b4a9d6aeedcf595cdf0e9b165c97e3e197471a1ee6320
e93389edfba28af6eaf10de98597ad56e7ab1b504ed762451996c3b98"),
MustHexID("b0c8e5d2c8634a7930e1a6fd082e448c6cf9d2d8b7293558b59238815a4df926c286bf
297d2049f14e8296a6eb3256af614ec1812c4f2bbe807673b58bf14c8c"),
MustHexID("0fb346076396a38badc342df3679b55bd7f40a609ab103411fe45082c01f12ea016729

e95914b2b5540e987ff5c9b133e85862648e7f36abdfd23100d248d234"),
MustHexID("f736e0cc83417feaa280d9483f5d4d72d1b036cd0c6d9cbdeb8ac35ceb2604780de46d
ddaa32a378474e1d5ccdf79b373331c30c7911ade2ae32f98832e5de1f"),
MustHexID("8b02991457602f42b38b342d3f2259ae4100c354b3843885f7e4e07bd644f64dab94bb
7f38a3915f8b7f11d8e3f81c28e07a0078cf79d7397e38a7b7e0c857e2"),
MustHexID("9221d9f04a8a184993d12baa91116692bb685f887671302999d69300ad103eb2d2c75
a09d8979404c6dd28f12362f58a1a43619c493d9108fd47588a23ce5824"),
MustHexID("652797801744dada833fff207d67484742eea6835d695925f3e618d71b68ec3c65bdd8
5b4302b2cdcb835ad3f94fd00d8da07e570b41bc0d2bcf69a8de1b3284"),
MustHexID("d84f06fe64debc4cd0625e36d19b99014b6218375262cc2209202bdbafd7dffcc4e34ce
6398e182e02fd8faeed622c3e175545864902dfd3d1ac57647cddf4c6"),
MustHexID("d0ed87b294f38f1d741eb601020eeec30ac16331d05880fe27868f1e454446de367d74
57b41c79e202eaf9525b029e4f1d7e17d85a55f83a557c005c68d7328a"),
},
253: {
MustHexID("ad4485e386e3cc7c7310366a7c38fb810b8896c0d52e55944bfd320ca294e7912d6c5
3c0a0cf85e7ce226e92491d60430e86f8f15cda0161ed71893fb4a9e3a1"),
MustHexID("36d0e7e5b7734f98c6183eeeb8ac5130a85e910a925311a19c4941b1290f945d4fc399
6b12ef4966960b6fa0fb29b1604f83a0f81bd5fd6398d2e1a22e46af0c"),
MustHexID("7d307d8acb4a561afa23bdf0bd945d35c90245e26345ec3a1f9f7df354222a7cdcb8133
9c9ed6744526c27a1a0c8d10857e98df942fa433602facac71ac68a31"),
MustHexID("d97bf55f88c83fae36232661af115d66ca600fc4bd6d1fb35ff9bb4dad674c02cf8c8d05f
317525b5522250db58bb1ecafb7157392bf5aa61b178c61f098d995"),
MustHexID("7045d678f1f9eb7a4613764d17bd5698796494d0bf977b16f2dbc272b8a0f7858a6080
5c022fc3d1fe4f31c37e63cdaca0416c0d053ef48a815f8b19121605e0"),
MustHexID("14e1f21418d445748de2a95cd9a8c3b15b506f86a0acabd8af44bb968ce39885b19c88
22af61b3dd58a34d1f265baec30e3ae56149dc7d2aa4a538f7319f69c8"),
MustHexID("b9453d78281b66a4eac95a1546017111eaaa5f92a65d0de10b1122940e92b319728a
24edf4dec6acc412321b1c95266d39c7b3a5d265c629c3e49a65fb022c09"),
MustHexID("e8a49248419e3824a00d86af422f22f7366e2d4922b304b7169937616a01d9d6fa5abf
5cc01061a352dc866f48e1fa2240dbb453d872b1d7be62bdfc1d5e248c"),
MustHexID("bebcff24b52362f30e0589ee573ce2d86f073d58d18e6852a592fa86ceb1a6c9b96d7fb
9ec7ed1ed98a51b6743039e780279f6bb49d0a04327ac7a182d9a56f6"),
MustHexID("d0835e5a4291db249b8d2fca9f503049988180c7d247bedaa2cf3a1bad0a76709360a8
5d4f9a1423b2cbc82bb4d94b47c0cde20afc430224834c49fe312a9ae3"),
MustHexID("6b087fe2a2da5e4f0b0f4777598a4a7fb66bf77dbd5bfc44e8a7eaa432ab585a6e22689
1f56a7d4f5ed11a7c57b90f1661bba1059590ca4267a35801c2802913"),
MustHexID("d901e5bde52d1a0f4ddf010a686a53974cdae4ebe5c6551b3c37d6b6d635d38d5b0e5f
80bc0186a2c7809dbf3a42870dd09643e68d32db896c6da8ba734579e7"),
MustHexID("96419fb80efae4b674402bb969ebaab86c1274f29a83a311e24516d36cdf148fe21754
d46c97688cdd7468f24c08b13e4727c29263393638a3b37b99ff60ebca"),
MustHexID("7b9c1889ae916a5d5abcdfb0aaedcc9c6f9eb1c1a4f68d0c2d034fe79ac610ce917c3ab

c670744150fa891bfcd8ab14fed6983fca964de920aa393fa7b326748"),
MustHexID("7a369b2b8962cc4c65900be046482fbf7c14f98a135bbbae25152c82ad168fb2097b3d1429197cf46d3ce9fdeb64808f908a489cc6019725db040060fdfe5405"),
MustHexID("47bcae48288da5ecc7f5058dfa07cf14d89d06d6e449cb946e237aa6652ea050d9f5a24a65efdc0013ccf232bf88670979eddef249b054f63f38da9d7796dbd8"),
},
254: {
MustHexID("099739d7abc8abd38ecc7a816c521a1168a4dbd359fa7212a5123ab583ffa1cf485a5fed219575d6475dbcdd541638b2d3631a6c7fce7474e7fe3cba1d4d5853"),
MustHexID("c2b01603b088a7182d0cf7ef29fb2b04c70acb320fccf78526bf9472e10c74ee70b3fcfa6f4b11d167bd7d3bc4d936b660f2c9bff934793d97cb21750e7c3d31"),
MustHexID("20e4d8f45f2f863e94b45548c1ef22a11f7d36f263e4f8623761e05a64c4572379b000a52211751e2561b0f14f4fc92dd4130410c8ccc71eb4f0e95a700d4ca9"),
MustHexID("27f4a16cc085e72d86e25c98bd2eca173eaaee7565c78ec5a52e9e12b2211f35de81b5b45e9195de2ebfe29106742c59112b951a04eb7ae48822911fc1f9389e"),
MustHexID("55db5ee7d98e7f0b1c3b9d5be6f2bc619a1b86c3cdd513160ad4dcf267037a5fffad527ac15d50aeb32c59c13d1d4c1e567ebbf4de0d25236130c8361f9aac63"),
MustHexID("883df308b0130fc928a8559fe50667a0fff80493bc09685d18213b2db241a3ad11310ed86b0ef662b3ce21fc3d9aa7f3fc24b8d9afe17c7407e9afd3345ae548"),
MustHexID("c7af968cc9bc8200c3ee1a387405f7563be1dce6710a3439f42ea40657d0eae9d2b3c16c42d779605351fcdece4da637b9804e60ca08cfb89aec32c197beffa6"),
MustHexID("3e66f2b788e3ff1d04106b80597915cd7afa06c405a7ae026556b6e583dca8e05cfbab5039bb9a1b5d06083ffe8de5780b1775550e7218f5e98624bf7af9a0a8"),
MustHexID("4fc7f53764de3337fdaec0a711d35d3a923e72fa65025444d12230b3552ed43d9b2d1ad08ccb11f2d50c58809e6dd74dde910e195294fca3b47ae5a3967cc479"),
MustHexID("bafdfdcf6ccaa989436752fa97c77477b6baa7deb374b16c095492c529eb133e8e2f99e1977012b64767b9d34b2cf6d2048ed489bd822b5139b523f6a423167b"),
MustHexID("7f5d78008a4312fe059104ce80202c82b8915c2eb4411c6b812b16f7642e57c00f2c9425121f5cbac4257fe0b3e81ef5dea97ea2dbaa98f6a8b6fd4d1e5980bb"),
MustHexID("598c37fe78f922751a052f463aeb0cb0bc7f52b7c2a4cf2da72ec0931c7c32175d4165d0f8998f7320e87324ac3311c03f9382a5385c55f0407b7a66b2acd864"),
MustHexID("f758c4136e1c148777a7f3275a76e2db0b2b04066fd738554ec398c1c6cc9fb47e14a3b4c87bd47deaeab3ffd2110514c3855685a374794daff87b605b27ee2e"),
MustHexID("0307bb9e4fd865a49dcf1fe4333d1b944547db650ab580af0b33e53c4fef6c789531110fac801bbcbce21fc4d6f61b6d5b24abdf5b22e3030646d579f6dca9c2"),
MustHexID("82504b6eb49bb2c0f91a7006ce9cefdbaf6df38706198502c2e06601091fc9dc91e4f15db3410d45c6af355bc270b0f268d3dff560f956985c7332d4b10bd1ed"),
MustHexID("b39b5b677b45944ceebe76e76d1f051de2f2a0ec7b0d650da52135743e66a9a5dba45f638258f9a7545d9a790c7fe6d3fdf82c25425c7887323e45d27d06c057"),
},
255: {
MustHexID("5c4d58d46e055dd1f093f81ee60a675e1f02f54da6206720adee4dccef9b67a31efc5c2

a2949c31a04ee31beadc79aba10da31440a1f9ff2a24093c63c36d784"),
MustHexID("ea72161ffdd4b1e124c7b93b0684805f4c4b58d617ed498b37a145c670dbc2e04976f8
785583d9c805ffbf343c31d492d79f841652bbbd01b61ed85640b23495"),
MustHexID("51caa1d93352d47a8e531692a3612adac1e8ac68d0a200d086c1c57ae1e1a91aa285
ab242e8c52ef9d7afe374c9485b122ae815f1707b875569d0433c1c3ce85"),
MustHexID("c08397d5751b47bd3da044b908be0fb0e510d3149574dff7aeab33749b023bb171b57
69990fe17469dbebc100bc150e798aeda426a2dcc766699a225fddd75c6"),
MustHexID("0222c1c194b749736e593f937fad67ee348ac57287a15c7e42877aa38a9b87732a408
bca370f812efd0eedbff13e6d5b854bf3ba1dec431a796ed47f32552b09"),
MustHexID("03d859cd46ef02d9bfad5268461a6955426845eef4126de6be0fa4e8d7e0727ba2385b
78f1a883a8239e95ebb814f2af8379632c7d5b100688eebc5841209582"),
MustHexID("64d5004b7e043c39ff0bd10cb20094c287721d5251715884c280a612b494b3e9e1c64
ba6f67614994c7d969a0d0c0295d107d53fc225d47c44c4b82852d6f960"),
MustHexID("b0a5eefb2dab6f786670f35bf9641eefe6dd87fd3f1362bcab4aaa792903500ab23d88fa
e68411372e0813b057535a601d46e454323745a948017f6063a47b1f"),
MustHexID("0cc6df0a3433d448b5684d2a3ffa9d1a825388177a18f44ad0008c7bd7702f1ec0fc38b
83506f7de689c3b6ecb552599927e29699eed6bb867ff08f80068b287"),
MustHexID("50772f7b8c03a4e153355fbbf79c8a80cf32af656ff0c7873c99911099d04a0dae067470
6c357e0145ad017a0ade65e6052cb1b0d574fcd6f67da3eee0ace66b"),
MustHexID("1ae37829c9ef41f8b508b82259ebac76b1ed900d7a45c08b7970f25d2d48ddd1829e2f
11423a18749940b6dab8598c6e416cef0efd47e46e51f29a0bc65b37cd"),
MustHexID("ba973cab31c2af091fc1644a93527d62b2394999e2b6ccbf158dd5ab9796a43d408786
f1803ef4e29debfeb62fce2b6caa5ab2b24d1549c822a11c40c2856665"),
MustHexID("bc413ad270dd6ea25bddba78f3298b03b8ba6f8608ac03d06007d4116fa78ef5a0cfe8c
80155089382fc7a193243ee5500082660cb5d7793f60f2d7d18650964"),
MustHexID("5a6a9ef07634d9eec3baa87c997b529b92652afa11473dfee41ef7037d5c06e0ddb9fe8
42364462d79dd31cff8a59a1b8d5bc2b810dea1d4cbbd3beb80ecec83"),
MustHexID("f492c6ee2696d5f682f7f537757e52744c2ae560f1090a07024609e903d334e9e174fc0
1609c5a229ddbcac36c9d21adaf6457dab38a25bfd44f2f0ee4277998"),
MustHexID("459e4db99298cb0467a90acee6888b08bb857450deac11015cced5104853be5adce5
b69c740968bc7f931495d671a70cad9f48546d7cd203357fe9af0e8d2164"),
},
256: {
MustHexID("a8593af8a4aef7b806b5197612017951bac8845a1917ca9a6a15dd6086d6085051449
90b245785c4cd2d67a295701c7aac2aa18823fb0033987284b019656268"),
MustHexID("d2eebef914928c3aad77fc1b2a495f52d2294acf5edaa7d8a530b540f094b861a68fe83
48a46a7c302f08ab609d85912a4968eacfea0740847b29421b4795d9e"),
MustHexID("b14bfcb31495f32b650b63cf7d08492e3e29071fdc73cf2da0da48d4b191a70ba1a65f4
2ad8c343206101f00f8a48e8db4b08bf3f622c0853e7323b250835b91"),
MustHexID("7feaee0d818c03eb30e4e0bf03ade0f3c21ca38e938a761aa1781cf70bda8cc5cd631a
6cc53dd44f1d4a6d3e2dae6513c6c66ee50cb2f0e9ad6f7e319b309fd9"),
MustHexID("4ca3b657b139311db8d583c25dd5963005e46689e1317620496cc64129c7f3e528708

```
20e0ec7941d28809311df6db8a2867bbd4f235b4248af24d7a9c22d1232"),
MustHexID("1181defb1d16851d42dd951d84424d6bd1479137f587fa184d5a8152be6b6b16ed08b
cdb2c2ed8539bcde98c80c432875f9f724737c316a2bd385a39d3cab1d8"),
MustHexID("d9dd818769fa0c3ec9f553c759b92476f082817252a04a47dc1777740b1731d280058c
66f982812f173a294acf4944a85ba08346e2de153ba3ba41ce8a62cb64"),
MustHexID("bd7c4f8a9e770aa915c771b15e107ca123d838762da0d3ffc53aa6b53e9cd076cffc534
ec4d2e4c334c683f1f5ea72e0e123f6c261915ed5b58ac1b59f003d88"),
MustHexID("3dd5739c73649d510456a70e9d6b46a855864a4a3f744e088fd8c8da11b18e4c9b5f2d
7da50b1c147b2bae5ca9609ae01f7a3cdea9dce34f80a91d29cd82f918"),
MustHexID("f0d7df1efc439b4bcc0b762118c1cfa99b2a6143a9f4b10e3c9465125f4c9fca4ab88a25
04169bbcad65492cf2f50da9dd5d077c39574a944f94d8246529066b"),
MustHexID("dd598b9ba441448e5fb1a6ec6c5f5aa9605bad6e223297c729b1705d11d05f6bfd3d41
988b694681ae69bb03b9a08bff4beab5596503d12a39bffb5cd6e94c7c"),
MustHexID("3fce284ac97e567aebae681b15b7a2b6df9d873945536335883e4bbc26460c0643705
37f323fd1ada828ea43154992d14ac0cec0940a2bd2a3f42ec156d60c83"),
MustHexID("7c8dfa8c1311cb14fb29a8ac11bca23ecc115e56d9fcf7b7ac1db9066aa4eb39f8b1dabf
46e192a65be95ebfb4e839b5ab4533fef414921825e996b210dd53bd"),
MustHexID("cafa6934f82120456620573d7f801390ed5e16ed619613a37e409e44ab355ef755e835
65a913b48a9466db786f8d4fbd590bfec474c2524d4a2608d4eafd6abd"),
MustHexID("9d16600d0dd310d77045769fed2cb427f32db88cd57d86e49390c2ba8a9698cfa856f7
75be2013237226e7bf47b248871cf865d23015937d1edeb20db5e3e760"),
MustHexID("17be6b6ba54199b1d80eff866d348ea11d8a4b341d63ad9a6681d3ef8a43853ac564d
153eb2a8737f0afc9ab320f6f95c55aa11aaa13bbb1ff422fd16bdf8188"),
},
},
}

type preminedTestnet struct {
target    NodeID
targetSha common.Hash // sha3(target)
dists     [hashBits + 1][]NodeID
net       *Network
}

func (tn *preminedTestnet) sendFindnode(to *Node, target NodeID) {
panic("sendFindnode called")
}

func (tn *preminedTestnet) sendFindnodeHash(to *Node, target common.Hash) {
// current log distance is encoded in port number
// fmt.Println("findnode query at dist", toaddr.Port)
if to.UDP <= lowPort {
```

```
panic("query to node at or below distance 0")
}
next := to.UDP - 1
var result []rpcNode
for i, id := range tn.dists[to.UDP-lowPort] {
result = append(result, nodeToRPC(NewNode(id, net.ParseIP("10.0.2.99"), next,
uint16(i)+1+lowPort)))
}
injectResponse(tn.net, to, neighborsPacket, &neighbors{Nodes: result})
}

func (tn *preminedTestnet) sendPing(to *Node, addr *net.UDPAddr, topics []Topic) []byte {
injectResponse(tn.net, to, pongPacket, &pong{ReplyTok: []byte{1}})
return []byte{1}
}

func (tn *preminedTestnet) send(to *Node, ptype nodeEvent, data interface{}) (hash []byte) {
switch ptype {
case pingPacket:
injectResponse(tn.net, to, pongPacket, &pong{ReplyTok: []byte{1}})
case pongPacket:
// ignored
case findnodeHashPacket:
// current log distance is encoded in port number
// fmt.Println("findnode query at dist", toaddr.Port-lowPort)
if to.UDP <= lowPort {
panic("query to node at or below  distance 0")
}
next := to.UDP - 1
var result []rpcNode
for i, id := range tn.dists[to.UDP-lowPort] {
result = append(result, nodeToRPC(NewNode(id, net.ParseIP("10.0.2.99"), next,
uint16(i)+1+lowPort)))
}
injectResponse(tn.net, to, neighborsPacket, &neighbors{Nodes: result})
default:
panic("send(" + ptype.String() + ")")
}
return []byte{2}
}

func (tn *preminedTestnet) sendNeighbours(to *Node, nodes []*Node) {
```

```
panic("sendNeighbours called")
}

func (tn *preminedTestnet) sendTopicQuery(to *Node, topic Topic) {
panic("sendTopicQuery called")
}

func (tn *preminedTestnet) sendTopicNodes(to *Node, queryHash common.Hash, nodes []*Node)
{
panic("sendTopicNodes called")
}

func (tn *preminedTestnet) sendTopicRegister(to *Node, topics []Topic, idx int, pong []byte) {
panic("sendTopicRegister called")
}

func (*preminedTestnet) Close() {}

func (*preminedTestnet) localAddr() *net.UDPAddr {
return &net.UDPAddr{IP: net.ParseIP("10.0.1.1"), Port: 40000}
}

// mine generates a testnet struct literal with nodes at
// various distances to the given target.
func (n *preminedTestnet) mine(target NodeID) {
n.target = target
n.targetSha = crypto.Keccak256Hash(n.target[:])
found := 0
for found < bucketSize*10 {
k := newkey()
id := PubkeyID(&k.PublicKey)
sha := crypto.Keccak256Hash(id[:])
ld := logdist(n.targetSha, sha)
if len(n.dists[ld]) < bucketSize {
n.dists[ld] = append(n.dists[ld], id)
fmt.Println("found ID with ld", ld)
found++
}
}
fmt.Println("&preminedTestnet{")
fmt.Printf("target: %#v,\n", n.target)
fmt.Printf("targetSha: %#v,\n", n.targetSha)
```

```
fmt.Printf("dists: [%d][]NodeID{\n", len(n.dists))
for ld, ns := range n.dists {
if len(ns) == 0 {
continue
}
fmt.Printf("%d: []NodeID{\n", ld)
for _, n := range ns {
fmt.Printf("MustHexID(\"%x\"),\n", n[:])
}
fmt.Println("},")
}
fmt.Println("},")
fmt.Println("}")
}

func injectResponse(net *Network, from *Node, ev nodeEvent, packet interface{}) {
go net.reqReadPacket(ingressPacket{remoteID: from.ID, remoteAddr: from.addr(), ev: ev, data:
packet})
}

96:F:\git\coin\ethereum\go-ethereum\p2p\discv5\node.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discv5

import (
"crypto/ecdsa"
"crypto/elliptic"
"encoding/hex"
"errors"
"fmt"
"math/big"
"math/rand"
"net"
"net/url"
"regexp"
"strconv"
"strings"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
)
```

```
// Node represents a host on the network.
// The public fields of Node may not be modified.
type Node struct {
IP       net.IP // len 4 for IPv4 or 16 for IPv6
UDP, TCP uint16 // port numbers
ID       NodeID // the node's public key

// Network-related fields are contained in nodeNetGuts.
// These fields are not supposed to be used off the
// Network.loop goroutine.
nodeNetGuts
}

// NewNode creates a new node. It is mostly meant to be used for
// testing purposes.
func NewNode(id NodeID, ip net.IP, udpPort, tcpPort uint16) *Node {
if ipv4 := ip.To4(); ipv4 != nil {
ip = ipv4
}
return &Node{
IP:         ip,
UDP:        udpPort,
TCP:        tcpPort,
ID:         id,
nodeNetGuts: nodeNetGuts{sha: crypto.Keccak256Hash(id[:])},
}
}

func (n *Node) addr() *net.UDPAddr {
return &net.UDPAddr{IP: n.IP, Port: int(n.UDP)}
}

func (n *Node) setAddr(a *net.UDPAddr) {
n.IP = a.IP
if ipv4 := a.IP.To4(); ipv4 != nil {
n.IP = ipv4
}
n.UDP = uint16(a.Port)
}

// compares the given address against the stored values.
```

```
func (n *Node) addrEqual(a *net.UDPAddr) bool {
ip := a.IP
if ipv4 := a.IP.To4(); ipv4 != nil {
ip = ipv4
}
return n.UDP == uint16(a.Port) && n.IP.Equal(ip)
}

// Incomplete returns true for nodes with no IP address.
func (n *Node) Incomplete() bool {
return n.IP == nil
}

// checks whether n is a valid complete node.
func (n *Node) validateComplete() error {
if n.Incomplete() {
return errors.New("incomplete node")
}
if n.UDP == 0 {
return errors.New("missing UDP port")
}
if n.TCP == 0 {
return errors.New("missing TCP port")
}
if n.IP.IsMulticast() || n.IP.IsUnspecified() {
return errors.New("invalid IP (multicast/unspecified)")
}
_, err := n.ID.Pubkey() // validate the key (on curve, etc.)
return err
}

// The string representation of a Node is a URL.
// Please see ParseNode for a description of the format.
func (n *Node) String() string {
u := url.URL{Scheme: "enode"}
if n.Incomplete() {
u.Host = fmt.Sprintf("%x", n.ID[:])
} else {
addr := net.TCPAddr{IP: n.IP, Port: int(n.TCP)}
u.User = url.User(fmt.Sprintf("%x", n.ID[:]))
u.Host = addr.String()
if n.UDP != n.TCP {
```

```
u.RawQuery = "discport=" + strconv.Itoa(int(n.UDP))
}
}
return u.String()
}

var incompleteNodeURL = regexp.MustCompile("(?i)^(?:enode://)?([0-9a-f]+)$")

// ParseNode parses a node designator.
//
// There are two basic forms of node designators
//   - incomplete nodes, which only have the public key (node ID)
//   - complete nodes, which contain the public key and IP/Port information
//
// For incomplete nodes, the designator must look like one of these
//
//    enode://<hex node id>
//    <hex node id>
//
// For complete nodes, the node ID is encoded in the username portion
// of the URL, separated from the host by an @ sign. The hostname can
// only be given as an IP address, DNS domain names are not allowed.
// The port in the host name section is the TCP listening port. If the
// TCP and UDP (discovery) ports differ, the UDP port is specified as
// query parameter "discport".
//
// In the following example, the node URL describes
// a node with IP address 10.3.58.6, TCP listening port 30303
// and UDP discovery port 30301.
//
//    enode://<hex node id>@10.3.58.6:30303?discport=30301
func ParseNode(rawurl string) (*Node, error) {
if m := incompleteNodeURL.FindStringSubmatch(rawurl); m != nil {
id, err := HexID(m[1])
if err != nil {
return nil, fmt.Errorf("invalid node ID (%v)", err)
}
return NewNode(id, nil, 0, 0), nil
}
return parseComplete(rawurl)
}
```

```
func parseComplete(rawurl string) (*Node, error) {
var (
id              NodeID
ip              net.IP
tcpPort, udpPort uint64
)
u, err := url.Parse(rawurl)
if err != nil {
return nil, err
}
if u.Scheme != "enode" {
return nil, errors.New("invalid URL scheme, want \"enode\"")
}
// Parse the Node ID from the user portion.
if u.User == nil {
return nil, errors.New("does not contain node ID")
}
if id, err = HexID(u.User.String()); err != nil {
return nil, fmt.Errorf("invalid node ID (%v)", err)
}
// Parse the IP address.
host, port, err := net.SplitHostPort(u.Host)
if err != nil {
return nil, fmt.Errorf("invalid host: %v", err)
}
if ip = net.ParseIP(host); ip == nil {
return nil, errors.New("invalid IP address")
}
// Ensure the IP is 4 bytes long for IPv4 addresses.
if ipv4 := ip.To4(); ipv4 != nil {
ip = ipv4
}
// Parse the port numbers.
if tcpPort, err = strconv.ParseUint(port, 10, 16); err != nil {
return nil, errors.New("invalid port")
}
udpPort = tcpPort
qv := u.Query()
if qv.Get("discport") != "" {
udpPort, err = strconv.ParseUint(qv.Get("discport"), 10, 16)
if err != nil {
return nil, errors.New("invalid discport in query")
```

```
}
}
return NewNode(id, ip, uint16(udpPort), uint16(tcpPort)), nil
}

// MustParseNode parses a node URL. It panics if the URL is not valid.
func MustParseNode(rawurl string) *Node {
n, err := ParseNode(rawurl)
if err != nil {
panic("invalid node URL: " + err.Error())
}
return n
}

// MarshalText implements encoding.TextMarshaler.
func (n *Node) MarshalText() ([]byte, error) {
return []byte(n.String()), nil
}

// UnmarshalText implements encoding.TextUnmarshaler.
func (n *Node) UnmarshalText(text []byte) error {
dec, err := ParseNode(string(text))
if err == nil {
*n = *dec
}
return err
}

// type nodeQueue []*Node
//
// // pushNew adds n to the end if it is not present.
// func (nl *nodeList) appendNew(n *Node) {
// for _, entry := range n {
// if entry == n {
// return
// }
// }
// *nq = append(*nq, n)
// }
//
// // popRandom removes a random node. Nodes closer to
// // to the head of the beginning of the have a slightly higher probability.
```

```
// func (nl *nodeList) popRandom() *Node {
// ix := rand.Intn(len(*nq))
// //TODO: probability as mentioned above.
// nl.removeIndex(ix)
// }
//
// func (nl *nodeList) removeIndex(i int) *Node {
// slice = *nl
// if len(*slice) <= i {
// return nil
// }
// *nl = append(slice[:i], slice[i+1:]...)
// }

const nodeIDBits = 512

// NodeID is a unique identifier for each node.
// The node identifier is a marshaled elliptic curve public key.
type NodeID [nodeIDBits / 8]byte

// NodeID prints as a long hexadecimal number.
func (n NodeID) String() string {
return fmt.Sprintf("%x", n[:])
}

// The Go syntax representation of a NodeID is a call to HexID.
func (n NodeID) GoString() string {
return fmt.Sprintf("discover.HexID(\"%x\")", n[:])
}

// HexID converts a hex string to a NodeID.
// The string may be prefixed with 0x.
func HexID(in string) (NodeID, error) {
var id NodeID
b, err := hex.DecodeString(strings.TrimPrefix(in, "0x"))
if err != nil {
return id, err
} else if len(b) != len(id) {
return id, fmt.Errorf("wrong length, want %d hex chars", len(id)*2)
}
copy(id[:], b)
return id, nil
```

```
}

// MustHexID converts a hex string to a NodeID.
// It panics if the string is not a valid NodeID.
func MustHexID(in string) NodeID {
id, err := HexID(in)
if err != nil {
panic(err)
}
return id
}

// PubkeyID returns a marshaled representation of the given public key.
func PubkeyID(pub *ecdsa.PublicKey) NodeID {
var id NodeID
pbytes := elliptic.Marshal(pub.Curve, pub.X, pub.Y)
if len(pbytes)-1 != len(id) {
panic(fmt.Errorf("need %d bit pubkey, got %d bits", (len(id)+1)*8, len(pbytes)))
}
copy(id[:], pbytes[1:])
return id
}

// Pubkey returns the public key represented by the node ID.
// It returns an error if the ID is not a point on the curve.
func (id NodeID) Pubkey() (*ecdsa.PublicKey, error) {
p := &ecdsa.PublicKey{Curve: crypto.S256(), X: new(big.Int), Y: new(big.Int)}
half := len(id) / 2
p.X.SetBytes(id[:half])
p.Y.SetBytes(id[half:])
if !p.Curve.IsOnCurve(p.X, p.Y) {
return nil, errors.New("id is invalid secp256k1 curve point")
}
return p, nil
}

func (id NodeID) mustPubkey() ecdsa.PublicKey {
pk, err := id.Pubkey()
if err != nil {
panic(err)
}
return *pk
```

```
}

// recoverNodeID computes the public key used to sign the
// given hash from the signature.
func recoverNodeID(hash, sig []byte) (id NodeID, err error) {
pubkey, err := crypto.Ecrecover(hash, sig)
if err != nil {
return id, err
}
if len(pubkey)-1 != len(id) {
return id, fmt.Errorf("recovered pubkey has %d bits, want %d bits", len(pubkey)*8, (len(id)+1)*8)
}
for i := range id {
id[i] = pubkey[i+1]
}
return id, nil
}

// distcmp compares the distances a->target and b->target.
// Returns -1 if a is closer to target, 1 if b is closer to target
// and 0 if they are equal.
func distcmp(target, a, b common.Hash) int {
for i := range target {
da := a[i] ^ target[i]
db := b[i] ^ target[i]
if da > db {
return 1
} else if da < db {
return -1
}
}
return 0
}

// table of leading zero counts for bytes [0..255]
var lzcount = [256]int{
8, 7, 6, 6, 5, 5, 5, 5,
4, 4, 4, 4, 4, 4, 4, 4,
3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3,
2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2,
```

```
2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
}

// logdist returns the logarithmic distance between a and b, log2(a ^ b).
func logdist(a, b common.Hash) int {
lz := 0
for i := range a {
x := a[i] ^ b[i]
if x == 0 {
lz += 8
} else {
lz += lzcount[x]
break
}
}
return len(a)*8 - lz
}
```

```
// hashAtDistance returns a random hash such that logdist(a, b) == n
func hashAtDistance(a common.Hash, n int) (b common.Hash) {
if n == 0 {
return a
}
// flip bit at position n, fill the rest with random bits
b = a
pos := len(a) - n/8 - 1
bit := byte(0x01) << (byte(n%8) - 1)
if bit == 0 {
pos++
bit = 0x80
}
b[pos] = a[pos]&^bit | ^a[pos]&bit // TODO: randomize end bits
for i := pos + 1; i < len(a); i++ {
b[i] = byte(rand.Intn(255))
}
return b
}
```

97:F:\git\coin\ethereum\go-ethereum\p2p\discv5\nodeevent_string.go

```
)

func (i nodeEvent) String() string {
switch {
case 0 <= i && i <= 8:
return _nodeEvent_name_0[_nodeEvent_index_0[i]:_nodeEvent_index_0[i+1]]
case 265 <= i && i <= 267:
i -= 265
return _nodeEvent_name_1[_nodeEvent_index_1[i]:_nodeEvent_index_1[i+1]]
default:
return fmt.Sprintf("nodeEvent(%d)", i)
}
}
```

98:F:\git\coin\ethereum\go-ethereum\p2p\discv5\node_test.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discv5

import (
```

```
"fmt"
"math/big"
"math/rand"
"net"
"reflect"
"strings"
"testing"
"testing/quick"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
)

func ExampleNewNode() {
id :=
MustHexID("1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b
8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439")

// Complete nodes contain UDP and TCP endpoints:
n1 := NewNode(id, net.ParseIP("2001:db8:3c4d:15::abcd:ef12"), 52150, 30303)
fmt.Println("n1:", n1)
fmt.Println("n1.Incomplete() ->", n1.Incomplete())

// An incomplete node can be created by passing zero values
// for all parameters except id.
n2 := NewNode(id, nil, 0, 0)
fmt.Println("n2:", n2)
fmt.Println("n2.Incomplete() ->", n2.Incomplete())

// Output:
// n1:
enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d78
21617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@[2001:db8:3c4d:15::abcd:ef12]:
30303?discport=52150
// n1.Incomplete() -> false
// n2:
enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d78
21617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439
// n2.Incomplete() -> true
}
```

```
var parseNodeTests = []struct {
rawurl     string
wantError  string
wantResult *Node
}{
{
rawurl:    "http://foobar",
wantError: `invalid URL scheme, want "enode"`,
},
{
rawurl:    "enode://01010101@123.124.125.126:3",
wantError: `invalid node ID (wrong length, want 128 hex chars)`,
},
// Complete nodes with IP address.
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@hostname:3",
wantError: `invalid IP address`,
},
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@127.0.0.1:foo",
wantError: `invalid port`,
},
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@127.0.0.1:3?discport=foo",
wantError: `invalid discport in query`,
},
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@127.0.0.1:52150",
wantResult: NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{0x7f, 0x0, 0x0, 0x1},
52150,
52150,
```

```
),
},
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@[::]:52150",
wantResult: NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.ParseIP("::"),
52150,
52150,
),
},
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@[2001:db8:3c4d:15::abcd:ef12
]:52150",
wantResult: NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.ParseIP("2001:db8:3c4d:15::abcd:ef12"),
52150,
52150,
),
},
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439@127.0.0.1:52150?discport=22
334",
wantResult: NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
net.IP{0x7f, 0x0, 0x0, 0x1},
22334,
52150,
),
},
// Incomplete nodes with no address.
{
```

```
rawurl:
"1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7821617d
2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439",
wantResult: NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
nil, 0, 0,
),
},
{
rawurl:
"enode://1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace24351232b8d7
821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439",
wantResult: NewNode(
MustHexID("0x1dd9d65c4552b5eb43d5ad55a2ee3f56c6cbc1c64a5c8d659f51fcd51bace2435123
2b8d7821617d2b29b54b81cdefb9b3e9c37d7fd5f63270bcc9e1a6f6a439"),
nil, 0, 0,
),
},
// Invalid URLs
{
rawurl:    "01010101",
wantError: `invalid node ID (wrong length, want 128 hex chars)`,
},
{
rawurl:    "enode://01010101",
wantError: `invalid node ID (wrong length, want 128 hex chars)`,
},
{
// This test checks that errors from url.Parse are handled.
rawurl:    "://foo",
wantError: `parse ://foo: missing protocol scheme`,
},
}

func TestParseNode(t *testing.T) {
for _, test := range parseNodeTests {
n, err := ParseNode(test.rawurl)
if test.wantError != "" {
if err == nil {
t.Errorf("test %q:\n  got nil error, expected %#q", test.rawurl, test.wantError)
continue
```

```
} else if err.Error() != test.wantError {
t.Errorf("test %q:\n  got error %#q, expected %#q", test.rawurl, err.Error(), test.wantError)
continue
}
} else {
if err != nil {
t.Errorf("test %q:\n  unexpected error: %v", test.rawurl, err)
continue
}
if !reflect.DeepEqual(n, test.wantResult) {
t.Errorf("test %q:\n  result mismatch:\ngot:  %#v, want: %#v", test.rawurl, n, test.wantResult)
}
}
}
}

func TestNodeString(t *testing.T) {
for i, test := range parseNodeTests {
if test.wantError == "" && strings.HasPrefix(test.rawurl, "enode://") {
str := test.wantResult.String()
if str != test.rawurl {
t.Errorf("test %d: Node.String() mismatch:\ngot:  %s\nwant: %s", i, str, test.rawurl)
}
}
}
}

func TestHexID(t *testing.T) {
ref := NodeID{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 128, 106, 217, 182, 31, 165, 174, 1, 67, 7, 235, 220, 150, 66, 83, 173, 205, 159,
44, 10, 57, 42, 161, 26, 188}
id1 :=
MustHexID("0x00000000000000000000000000000000000000000000000000000000000000000000000000806ad9b61fa5ae014307ebdc964253adcd9f2c0a392aa11abc")
id2 :=
MustHexID("0000000000000000000000000000000000000000000000000000000000000000000000000806ad9b61fa5ae014307ebdc964253adcd9f2c0a392aa11abc")

if id1 != ref {
t.Errorf("wrong id1\ngot  %v\nwant %v", id1[:], ref[:])
}
if id2 != ref {
```

```
t.Errorf("wrong id2\ngot  %v\nwant %v", id2[:], ref[:])
}
}

func TestNodeID_recover(t *testing.T) {
prv := newkey()
hash := make([]byte, 32)
sig, err := crypto.Sign(hash, prv)
if err != nil {
t.Fatalf("signing error: %v", err)
}

pub := PubkeyID(&prv.PublicKey)
recpub, err := recoverNodeID(hash, sig)
if err != nil {
t.Fatalf("recovery error: %v", err)
}
if pub != recpub {
t.Errorf("recovered wrong pubkey:\ngot:  %v\nwant: %v", recpub, pub)
}

ecdsa, err := pub.Pubkey()
if err != nil {
t.Errorf("Pubkey error: %v", err)
}
if !reflect.DeepEqual(ecdsa, &prv.PublicKey) {
t.Errorf("Pubkey mismatch:\n  got:  %#v\n  want: %#v", ecdsa, &prv.PublicKey)
}
}

func TestNodeID_pubkeyBad(t *testing.T) {
ecdsa, err := NodeID{}.Pubkey()
if err == nil {
t.Error("expected error for zero ID")
}
if ecdsa != nil {
t.Error("expected nil result")
}
}

func TestNodeID_distcmp(t *testing.T) {
distcmpBig := func(target, a, b common.Hash) int {
```

```
tbig := new(big.Int).SetBytes(target[:])
abig := new(big.Int).SetBytes(a[:])
bbig := new(big.Int).SetBytes(b[:])
return new(big.Int).Xor(tbig, abig).Cmp(new(big.Int).Xor(tbig, bbig))
}
if err := quick.CheckEqual(distcmp, distcmpBig, quickcfg()); err != nil {
t.Error(err)
}
}

// the random tests is likely to miss the case where they're equal.
func TestNodeID_distcmpEqual(t *testing.T) {
base := common.Hash{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
x := common.Hash{15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}
if distcmp(base, x, x) != 0 {
t.Errorf("distcmp(base, x, x) != 0")
}
}

func TestNodeID_logdist(t *testing.T) {
logdistBig := func(a, b common.Hash) int {
abig, bbig := new(big.Int).SetBytes(a[:]), new(big.Int).SetBytes(b[:])
return new(big.Int).Xor(abig, bbig).BitLen()
}
if err := quick.CheckEqual(logdist, logdistBig, quickcfg()); err != nil {
t.Error(err)
}
}

// the random tests is likely to miss the case where they're equal.
func TestNodeID_logdistEqual(t *testing.T) {
x := common.Hash{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
if logdist(x, x) != 0 {
t.Errorf("logdist(x, x) != 0")
}
}

func TestNodeID_hashAtDistance(t *testing.T) {
// we don't use quick.Check here because its output isn't
// very helpful when the test fails.
cfg := quickcfg()
for i := 0; i < cfg.MaxCount; i++ {
```

```
a := gen(common.Hash{}, cfg.Rand).(common.Hash)
dist := cfg.Rand.Intn(len(common.Hash{}) * 8)
result := hashAtDistance(a, dist)
actualdist := logdist(result, a)

if dist != actualdist {
t.Log("a:     ", a)
t.Log("result:", result)
t.Fatalf("#%d: distance of result is %d, want %d", i, actualdist, dist)
}
}
}

func quickcfg() *quick.Config {
return &quick.Config{
MaxCount: 5000,
Rand:     rand.New(rand.NewSource(time.Now().Unix())),
}
}

// TODO: The Generate method can be dropped when we require Go >= 1.5
// because testing/quick learned to generate arrays in 1.5.

func (NodeID) Generate(rand *rand.Rand, size int) reflect.Value {
var id NodeID
m := rand.Intn(len(id))
for i := len(id) - 1; i > m; i-- {
id[i] = byte(rand.Uint32())
}
return reflect.ValueOf(id)
}

99:F:\git\coin\ethereum\go-ethereum\p2p\discv5\ntp.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains the NTP time drift detection via the SNTP protocol:
//   https://tools.ietf.org/html/rfc4330

package discv5

import (
"fmt"
```

```
"net"
"sort"
"strings"
"time"

"github.com/ethereum/go-ethereum/log"
)

const (
ntpPool   = "pool.ntp.org" // ntpPool is the NTP server to query for the current time
ntpChecks = 3              // Number of measurements to do against the NTP server
)

// durationSlice attaches the methods of sort.Interface to []time.Duration,
// sorting in increasing order.
type durationSlice []time.Duration

func (s durationSlice) Len() int           { return len(s) }
func (s durationSlice) Less(i, j int) bool { return s[i] < s[j] }
func (s durationSlice) Swap(i, j int)      { s[i], s[j] = s[j], s[i] }

// checkClockDrift queries an NTP server for clock drifts and warns the user if
// one large enough is detected.
func checkClockDrift() {
drift, err := sntpDrift(ntpChecks)
if err != nil {
return
}
if drift < -driftThreshold || drift > driftThreshold {
warning := fmt.Sprintf("System clock seems off by %v, which can prevent network connectivity",
drift)
howtofix := fmt.Sprintf("Please enable network time synchronisation in system settings")
separator := strings.Repeat("-", len(warning))

log.Warn(fmt.Sprint(separator))
log.Warn(fmt.Sprint(warning))
log.Warn(fmt.Sprint(howtofix))
log.Warn(fmt.Sprint(separator))
} else {
log.Debug(fmt.Sprintf("Sanity NTP check reported %v drift, all ok", drift))
}
}
```

```
// sntpDrift does a naive time resolution against an NTP server and returns the
// measured drift. This method uses the simple version of NTP. It's not precise
// but should be fine for these purposes.
//
// Note, it executes two extra measurements compared to the number of requested
// ones to be able to discard the two extremes as outliers.
func sntpDrift(measurements int) (time.Duration, error) {
// Resolve the address of the NTP server
addr, err := net.ResolveUDPAddr("udp", ntpPool+":123")
if err != nil {
return 0, err
}
// Construct the time request (empty package with only 2 fields set):
//   Bits 3-5: Protocol version, 3
//   Bits 6-8: Mode of operation, client, 3
request := make([]byte, 48)
request[0] = 3<<3 | 3

// Execute each of the measurements
drifts := []time.Duration{}
for i := 0; i < measurements+2; i++ {
// Dial the NTP server and send the time retrieval request
conn, err := net.DialUDP("udp", nil, addr)
if err != nil {
return 0, err
}
defer conn.Close()

sent := time.Now()
if _, err = conn.Write(request); err != nil {
return 0, err
}
// Retrieve the reply and calculate the elapsed time
conn.SetDeadline(time.Now().Add(5 * time.Second))

reply := make([]byte, 48)
if _, err = conn.Read(reply); err != nil {
return 0, err
}
elapsed := time.Since(sent)
```

```
// Reconstruct the time from the reply data
sec := uint64(reply[43]) | uint64(reply[42])<<8 | uint64(reply[41])<<16 | uint64(reply[40])<<24
frac := uint64(reply[47]) | uint64(reply[46])<<8 | uint64(reply[45])<<16 | uint64(reply[44])<<24

nanosec := sec*1e9 + (frac*1e9)>>32

t := time.Date(1900, 1, 1, 0, 0, 0, 0, time.UTC).Add(time.Duration(nanosec)).Local()

// Calculate the drift based on an assumed answer time of RRT/2
drifts = append(drifts, sent.Sub(t)+elapsed/2)
}
// Calculate average drif (drop two extremities to avoid outliers)
sort.Sort(durationSlice(drifts))

drift := time.Duration(0)
for i := 1; i < len(drifts)-1; i++ {
drift += drifts[i]
}
return drift / time.Duration(measurements), nil
}

100:F:\git\coin\ethereum\go-ethereum\p2p\discv5\sim_run_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discv5

import (
"bufio"
"bytes"
"encoding/binary"
"errors"
"fmt"
"io"
"os"
"os/exec"
"runtime"
"strings"
"testing"
)

func getnacl() (string, error) {
switch runtime.GOARCH {
```

```
case "amd64":
_, err := exec.LookPath("sel_ldr_x86_64")
return "amd64p32", err
case "i386":
_, err := exec.LookPath("sel_ldr_i386")
return "i386", err
default:
return "", errors.New("nacl is not supported on " + runtime.GOARCH)
}
}

// runWithPlaygroundTime executes the caller
// in the NaCl sandbox with faketime enabled.
//
// This function must be called from a Test* function
// and the caller must skip the actual test when isHost is true.
func runWithPlaygroundTime(t *testing.T) (isHost bool) {
if runtime.GOOS == "nacl" {
return false
}

// Get the caller.
callerPC, _, _, ok := runtime.Caller(1)
if !ok {
panic("can't get caller")
}
callerFunc := runtime.FuncForPC(callerPC)
if callerFunc == nil {
panic("can't get caller")
}
callerName := callerFunc.Name()[strings.LastIndexByte(callerFunc.Name(), '.')+1:]
if !strings.HasPrefix(callerName, "Test") {
panic("must be called from witin a Test* function")
}
testPattern := "^" + callerName + "$"

// Unfortunately runtime.faketime (playground time mode) only works on NaCl. The NaCl
// SDK must be installed and linked into PATH for this to work.
arch, err := getnacl()
if err != nil {
t.Skip(err)
}
```

```
// Compile and run the calling test using NaCl.
// The extra tag ensures that the TestMain function in sim_main_test.go is used.
cmd := exec.Command("go", "test", "-v", "-tags", "faketime_simulation", "-timeout", "100h", "-run",
testPattern, ".")
cmd.Env = append([]string{"GOOS=nacl", "GOARCH=" + arch}, os.Environ()...)
stdout, _ := cmd.StdoutPipe()
stderr, _ := cmd.StderrPipe()
go skipPlaygroundOutputHeaders(os.Stdout, stdout)
go skipPlaygroundOutputHeaders(os.Stderr, stderr)
if err := cmd.Run(); err != nil {
t.Error(err)
}

// Ensure that the test function doesn't run in the (non-NaCl) host process.
return true
}

func skipPlaygroundOutputHeaders(out io.Writer, in io.Reader) {
// Additional output can be printed without the headers
// before the NaCl binary starts running (e.g. compiler error messages).
bufin := bufio.NewReader(in)
output, err := bufin.ReadBytes(0)
output = bytes.TrimSuffix(output, []byte{0})
if len(output) > 0 {
out.Write(output)
}
if err != nil {
return
}
bufin.UnreadByte()

// Playback header: 0 0 P B <8-byte time> <4-byte data length>
head := make([]byte, 4+8+4)
for {
if _, err := io.ReadFull(bufin, head); err != nil {
if err != io.EOF {
fmt.Fprintln(out, "read error:", err)
}
return
}
if !bytes.HasPrefix(head, []byte{0x00, 0x00, 'P', 'B'}) {
```

```
fmt.Fprintf(out, "expected playback header, got %q\n", head)
io.Copy(out, bufin)
return
}
// Copy data until next header.
size := binary.BigEndian.Uint32(head[12:])
io.CopyN(out, bufin, int64(size))
}
}
```

101:F:\git\coin\ethereum\go-ethereum\p2p\discv5\sim_test.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discv5

import (
"crypto/ecdsa"
"encoding/binary"
"fmt"
"math/rand"
"net"
"strconv"
"sync"
"sync/atomic"
"testing"
"time"

"github.com/ethereum/go-ethereum/common"
)

// In this test, nodes try to randomly resolve each other.
func TestSimRandomResolve(t *testing.T) {
t.Skip("boring")
if runWithPlaygroundTime(t) {
return
}

sim := newSimulation()
bootnode := sim.launchNode(false)

// A new node joins every 10s.
launcher := time.NewTicker(10 * time.Second)
```

```
go func() {
for range launcher.C {
net := sim.launchNode(false)
go randomResolves(t, sim, net)
if err := net.SetFallbackNodes([]*Node{bootnode.Self()}); err != nil {
panic(err)
}
fmt.Printf("launched @ %v: %x\n", time.Now(), net.Self().ID[:16])
}
}()

time.Sleep(3 * time.Hour)
launcher.Stop()
sim.shutdown()
sim.printStats()
}

func TestSimTopics(t *testing.T) {
t.Skip("NaCl test")
if runWithPlaygroundTime(t) {
return
}
sim := newSimulation()
bootnode := sim.launchNode(false)

go func() {
nets := make([]*Network, 1024)
for i := range nets {
net := sim.launchNode(false)
nets[i] = net
if err := net.SetFallbackNodes([]*Node{bootnode.Self()}); err != nil {
panic(err)
}
time.Sleep(time.Second * 5)
}

for i, net := range nets {
if i < 256 {
stop := make(chan struct{})
go net.RegisterTopic(testTopic, stop)
go func() {
//time.Sleep(time.Second * 36000)
```

```
time.Sleep(time.Second * 40000)
close(stop)
}()
time.Sleep(time.Millisecond * 100)
}
//time.Sleep(time.Second * 10)
//time.Sleep(time.Second)
/*if i%500 == 499 {
time.Sleep(time.Second * 9501)
} else {
time.Sleep(time.Second)
}*/
}
}()

// A new node joins every 10s.
/*launcher := time.NewTicker(5 * time.Second)
cnt := 0
var printNet *Network
go func() {
for range launcher.C {
cnt++
if cnt <= 1000 {
log := false //(cnt == 500)
net := sim.launchNode(log)
if log {
printNet = net
}
if cnt > 500 {
go net.RegisterTopic(testTopic, nil)
}
if err := net.SetFallbackNodes([]*Node{bootnode.Self()}); err != nil {
panic(err)
}
}
//fmt.Printf("launched @ %v: %x\n", time.Now(), net.Self().ID[:16])
}
}()
*/
time.Sleep(55000 * time.Second)
//launcher.Stop()
sim.shutdown()
```

```
//sim.printStats()
//printNet.log.printLogs()
}

/*func testHierarchicalTopics(i int) []Topic {
digits := strconv.FormatInt(int64(256+i/4), 4)
res := make([]Topic, 5)
for i, _ := range res {
res[i] = Topic("foo" + digits[1:i+1])
}
return res
}*/

func testHierarchicalTopics(i int) []Topic {
digits := strconv.FormatInt(int64(128+i/8), 2)
res := make([]Topic, 8)
for i := range res {
res[i] = Topic("foo" + digits[1:i+1])
}
return res
}

func TestSimTopicHierarchy(t *testing.T) {
t.Skip("NaCl test")
if runWithPlaygroundTime(t) {
return
}
sim := newSimulation()
bootnode := sim.launchNode(false)

go func() {
nets := make([]*Network, 1024)
for i := range nets {
net := sim.launchNode(false)
nets[i] = net
if err := net.SetFallbackNodes([]*Node{bootnode.Self()}); err != nil {
panic(err)
}
time.Sleep(time.Second * 5)
}

stop := make(chan struct{})
```

```
for i, net := range nets {
//if i < 256 {
for _, topic := range testHierarchicalTopics(i)[:5] {
//fmt.Println("reg", topic)
go net.RegisterTopic(topic, stop)
}
time.Sleep(time.Millisecond * 100)
//}
}
time.Sleep(time.Second * 90000)
close(stop)
}()

time.Sleep(100000 * time.Second)
sim.shutdown()
}

func randomResolves(t *testing.T, s *simulation, net *Network) {
randtime := func() time.Duration {
return time.Duration(rand.Intn(50)+20) * time.Second
}
lookup := func(target NodeID) bool {
result := net.Resolve(target)
return result != nil && result.ID == target
}

timer := time.NewTimer(randtime())
for {
select {
case <-timer.C:
target := s.randomNode().Self().ID
if !lookup(target) {
t.Errorf("node %x: target %x not found", net.Self().ID[:8], target[:8])
}
timer.Reset(randtime())
case <-net.closed:
return
}
}
}

type simulation struct {
```

```
mu      sync.RWMutex
nodes   map[NodeID]*Network
nodectr uint32
}

func newSimulation() *simulation {
return &simulation{nodes: make(map[NodeID]*Network)}
}

func (s *simulation) shutdown() {
s.mu.RLock()
alive := make([]*Network, 0, len(s.nodes))
for _, n := range s.nodes {
alive = append(alive, n)
}
defer s.mu.RUnlock()

for _, n := range alive {
n.Close()
}
}

func (s *simulation) printStats() {
s.mu.Lock()
defer s.mu.Unlock()
fmt.Println("node counter:", s.nodectr)
fmt.Println("alive nodes:", len(s.nodes))

// for _, n := range s.nodes {
// fmt.Printf("%x\n", n.tab.self.ID[:8])
// transport := n.conn.(*simTransport)
// fmt.Println("   joined:", transport.joinTime)
// fmt.Println("   sends:", transport.hashctr)
// fmt.Println("   table size:", n.tab.count)
// }

/*for _, n := range s.nodes {
fmt.Println()
fmt.Printf("*** Node %x\n", n.tab.self.ID[:8])
n.log.printLogs()
}*/
```

```
}

func (s *simulation) randomNode() *Network {
s.mu.Lock()
defer s.mu.Unlock()

n := rand.Intn(len(s.nodes))
for _, net := range s.nodes {
if n == 0 {
return net
}
n--
}
return nil
}

func (s *simulation) launchNode(log bool) *Network {
var (
num = s.nodectr
key = newkey()
id  = PubkeyID(&key.PublicKey)
ip  = make(net.IP, 4)
)
s.nodectr++
binary.BigEndian.PutUint32(ip, num)
ip[0] = 10
addr := &net.UDPAddr{IP: ip, Port: 30303}

transport := &simTransport{joinTime: time.Now(), sender: id, senderAddr: addr, sim: s, priv: key}
net, err := newNetwork(transport, key.PublicKey, nil, "<no database>", nil)
if err != nil {
panic("cannot launch new node: " + err.Error())
}

s.mu.Lock()
s.nodes[id] = net
s.mu.Unlock()

return net
}

func (s *simulation) dropNode(id NodeID) {
```

```
s.mu.Lock()
n := s.nodes[id]
delete(s.nodes, id)
s.mu.Unlock()

n.Close()
}

type simTransport struct {
joinTime   time.Time
sender     NodeID
senderAddr *net.UDPAddr
sim        *simulation
hashctr    uint64
priv       *ecdsa.PrivateKey
}

func (st *simTransport) localAddr() *net.UDPAddr {
return st.senderAddr
}

func (st *simTransport) Close() {}

func (st *simTransport) send(remote *Node, ptype nodeEvent, data interface{}) (hash []byte) {
hash = st.nextHash()
var raw []byte
if ptype == pongPacket {
var err error
raw, _, err = encodePacket(st.priv, byte(ptype), data)
if err != nil {
panic(err)
}
}

st.sendPacket(remote.ID, ingressPacket{
remoteID:   st.sender,
remoteAddr: st.senderAddr,
hash:       hash,
ev:         ptype,
data:       data,
rawData:    raw,
})
```

```
return hash
}

func (st *simTransport) sendPing(remote *Node, remoteAddr *net.UDPAddr, topics []Topic) []byte {
hash := st.nextHash()
st.sendPacket(remote.ID, ingressPacket{
remoteID:   st.sender,
remoteAddr: st.senderAddr,
hash:       hash,
ev:         pingPacket,
data: &ping{
Version:    4,
From:       rpcEndpoint{IP: st.senderAddr.IP, UDP: uint16(st.senderAddr.Port), TCP: 30303},
To:         rpcEndpoint{IP: remoteAddr.IP, UDP: uint16(remoteAddr.Port), TCP: 30303},
Expiration: uint64(time.Now().Unix() + int64(expiration)),
Topics:     topics,
},
})
return hash
}

func (st *simTransport) sendPong(remote *Node, pingHash []byte) {
raddr := remote.addr()

st.sendPacket(remote.ID, ingressPacket{
remoteID:   st.sender,
remoteAddr: st.senderAddr,
hash:       st.nextHash(),
ev:         pongPacket,
data: &pong{
To:         rpcEndpoint{IP: raddr.IP, UDP: uint16(raddr.Port), TCP: 30303},
ReplyTok:   pingHash,
Expiration: uint64(time.Now().Unix() + int64(expiration)),
},
})
}

func (st *simTransport) sendFindnodeHash(remote *Node, target common.Hash) {
st.sendPacket(remote.ID, ingressPacket{
remoteID:   st.sender,
remoteAddr: st.senderAddr,
hash:       st.nextHash(),
```

```
ev:         findnodeHashPacket,
data: &findnodeHash{
Target:     target,
Expiration: uint64(time.Now().Unix() + int64(expiration)),
},
})
}

func (st *simTransport) sendTopicRegister(remote *Node, topics []Topic, idx int, pong []byte) {
//fmt.Println("send", topics, pong)
st.sendPacket(remote.ID, ingressPacket{
remoteID:   st.sender,
remoteAddr: st.senderAddr,
hash:       st.nextHash(),
ev:         topicRegisterPacket,
data: &topicRegister{
Topics: topics,
Idx:    uint(idx),
Pong:   pong,
},
})
}

func (st *simTransport) sendTopicNodes(remote *Node, queryHash common.Hash, nodes
[]*Node) {
rnodes := make([]rpcNode, len(nodes))
for i := range nodes {
rnodes[i] = nodeToRPC(nodes[i])
}
st.sendPacket(remote.ID, ingressPacket{
remoteID:   st.sender,
remoteAddr: st.senderAddr,
hash:       st.nextHash(),
ev:         topicNodesPacket,
data:       &topicNodes{Echo: queryHash, Nodes: rnodes},
})
}

func (st *simTransport) sendNeighbours(remote *Node, nodes []*Node) {
// TODO: send multiple packets
rnodes := make([]rpcNode, len(nodes))
for i := range nodes {
```

```
rnodes[i] = nodeToRPC(nodes[i])
}
st.sendPacket(remote.ID, ingressPacket{
remoteID:   st.sender,
remoteAddr: st.senderAddr,
hash:       st.nextHash(),
ev:         neighborsPacket,
data: &neighbors{
Nodes:      rnodes,
Expiration: uint64(time.Now().Unix() + int64(expiration)),
},
})
}

func (st *simTransport) nextHash() []byte {
v := atomic.AddUint64(&st.hashctr, 1)
var hash common.Hash
binary.BigEndian.PutUint64(hash[:], v)
return hash[:]
}

const packetLoss = 0 // 1/1000

func (st *simTransport) sendPacket(remote NodeID, p ingressPacket) {
if rand.Int31n(1000) >= packetLoss {
st.sim.mu.RLock()
recipient := st.sim.nodes[remote]
st.sim.mu.RUnlock()

time.AfterFunc(200*time.Millisecond, func() {
recipient.reqReadPacket(p)
})
}
}

102:F:\git\coin\ethereum\go-ethereum\p2p\discv5\sim_testmain_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// +build go1.4,nacl,faketime_simulation

package discv5
```

```
import (
"os"
"runtime"
"testing"
"unsafe"
)

// Enable fake time mode in the runtime, like on the go playground.
// There is a slight chance that this won't work because some go code
// might have executed before the variable is set.

//go:linkname faketime runtime.faketime
var faketime = 1

func TestMain(m *testing.M) {
// We need to use unsafe somehow in order to get access to go:linkname.
_ = unsafe.Sizeof(0)

// Run the actual test. runWithPlaygroundTime ensures that the only test
// that runs is the one calling it.
runtime.GOMAXPROCS(8)
os.Exit(m.Run())
}

103:F:\git\coin\ethereum\go-ethereum\p2p\discv5\table.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package discv5 implements the RLPx v5 Topic Discovery Protocol.
//
// The Topic Discovery protocol provides a way to find RLPx nodes that
// can be connected to. It uses a Kademlia-like protocol to maintain a
// distributed database of the IDs and endpoints of all listening
// nodes.
package discv5

import (
"crypto/rand"
"encoding/binary"
"fmt"
"net"
"sort"
```

```
"github.com/ethereum/go-ethereum/common"
)

const (
alpha      = 3  // Kademlia concurrency factor
bucketSize = 16 // Kademlia bucket size
hashBits   = len(common.Hash{}) * 8
nBuckets   = hashBits + 1 // Number of buckets

maxBondingPingPongs = 16
maxFindnodeFailures = 5
)

type Table struct {
count         int               // number of nodes
buckets       [nBuckets]*bucket // index of known nodes by distance
nodeAddedHook func(*Node)       // for testing
self          *Node             // metadata of the local node
}

// bucket contains nodes, ordered by their last activity. the entry
// that was most recently active is the first element in entries.
type bucket struct {
entries      []*Node
replacements []*Node
}

func newTable(ourID NodeID, ourAddr *net.UDPAddr) *Table {
self := NewNode(ourID, ourAddr.IP, uint16(ourAddr.Port), uint16(ourAddr.Port))
tab := &Table{self: self}
for i := range tab.buckets {
tab.buckets[i] = new(bucket)
}
return tab
}

const printTable = false

// chooseBucketRefreshTarget selects random refresh targets to keep all Kademlia
// buckets filled with live connections and keep the network topology healthy.
// This requires selecting addresses closer to our own with a higher probability
// in order to refresh closer buckets too.
```

```
//
// This algorithm approximates the distance distribution of existing nodes in the
// table by selecting a random node from the table and selecting a target address
// with a distance less than twice of that of the selected node.
// This algorithm will be improved later to specifically target the least recently
// used buckets.
func (tab *Table) chooseBucketRefreshTarget() common.Hash {
entries := 0
if printTable {
fmt.Println()
}
for i, b := range tab.buckets {
entries += len(b.entries)
if printTable {
for _, e := range b.entries {
fmt.Println(i, e.state, e.addr().String(), e.ID.String(), e.sha.Hex())
}
}
}

prefix := binary.BigEndian.Uint64(tab.self.sha[0:8])
dist := ^uint64(0)
entry := int(randUint(uint32(entries + 1)))
for _, b := range tab.buckets {
if entry < len(b.entries) {
n := b.entries[entry]
dist = binary.BigEndian.Uint64(n.sha[0:8]) ^ prefix
break
}
entry -= len(b.entries)
}

ddist := ^uint64(0)
if dist+dist > dist {
ddist = dist
}
targetPrefix := prefix ^ randUint64n(ddist)

var target common.Hash
binary.BigEndian.PutUint64(target[0:8], targetPrefix)
rand.Read(target[8:])
return target
```

```
}

// readRandomNodes fills the given slice with random nodes from the
// table. It will not write the same node more than once. The nodes in
// the slice are copies and can be modified by the caller.
func (tab *Table) readRandomNodes(buf []*Node) (n int) {
// TODO: tree-based buckets would help here
// Find all non-empty buckets and get a fresh slice of their entries.
var buckets [][]*Node
for _, b := range tab.buckets {
if len(b.entries) > 0 {
buckets = append(buckets, b.entries[:])
}
}
if len(buckets) == 0 {
return 0
}
// Shuffle the buckets.
for i := uint32(len(buckets)) - 1; i > 0; i-- {
j := randUint(i)
buckets[i], buckets[j] = buckets[j], buckets[i]
}
// Move head of each bucket into buf, removing buckets that become empty.
var i, j int
for ; i < len(buf); i, j = i+1, (j+1)%len(buckets) {
b := buckets[j]
buf[i] = &(*b[0])
buckets[j] = b[1:]
if len(b) == 1 {
buckets = append(buckets[:j], buckets[j+1:]...)
}
if len(buckets) == 0 {
break
}
}
return i + 1
}

func randUint(max uint32) uint32 {
if max < 2 {
return 0
}
```

```
var b [4]byte
rand.Read(b[:])
return binary.BigEndian.Uint32(b[:]) % max
}

func randUint64n(max uint64) uint64 {
if max < 2 {
return 0
}
var b [8]byte
rand.Read(b[:])
return binary.BigEndian.Uint64(b[:]) % max
}

// closest returns the n nodes in the table that are closest to the
// given id. The caller must hold tab.mutex.
func (tab *Table) closest(target common.Hash, nresults int) *nodesByDistance {
// This is a very wasteful way to find the closest nodes but
// obviously correct. I believe that tree-based buckets would make
// this easier to implement efficiently.
close := &nodesByDistance{target: target}
for _, b := range tab.buckets {
for _, n := range b.entries {
close.push(n, nresults)
}
}
return close
}

// add attempts to add the given node its corresponding bucket. If the
// bucket has space available, adding the node succeeds immediately.
// Otherwise, the node is added to the replacement cache for the bucket.
func (tab *Table) add(n *Node) (contested *Node) {
//fmt.Println("add", n.addr().String(), n.ID.String(), n.sha.Hex())
if n.ID == tab.self.ID {
return
}
b := tab.buckets[logdist(tab.self.sha, n.sha)]
switch {
case b.bump(n):
// n exists in b.
return nil
```

```
case len(b.entries) < bucketSize:
// b has space available.
b.addFront(n)
tab.count++
if tab.nodeAddedHook != nil {
tab.nodeAddedHook(n)
}
return nil
default:
// b has no space left, add to replacement cache
// and revalidate the last entry.
// TODO: drop previous node
b.replacements = append(b.replacements, n)
if len(b.replacements) > bucketSize {
copy(b.replacements, b.replacements[1:])
b.replacements = b.replacements[:len(b.replacements)-1]
}
return b.entries[len(b.entries)-1]
}
}

// stuff adds nodes the table to the end of their corresponding bucket
// if the bucket is not full.
func (tab *Table) stuff(nodes []*Node) {
outer:
for _, n := range nodes {
if n.ID == tab.self.ID {
continue // don't add self
}
bucket := tab.buckets[logdist(tab.self.sha, n.sha)]
for i := range bucket.entries {
if bucket.entries[i].ID == n.ID {
continue outer // already in bucket
}
}
if len(bucket.entries) < bucketSize {
bucket.entries = append(bucket.entries, n)
tab.count++
if tab.nodeAddedHook != nil {
tab.nodeAddedHook(n)
}
}
```

```
}
}

// delete removes an entry from the node table (used to evacuate
// failed/non-bonded discovery peers).
func (tab *Table) delete(node *Node) {
//fmt.Println("delete", node.addr().String(), node.ID.String(), node.sha.Hex())
bucket := tab.buckets[logdist(tab.self.sha, node.sha)]
for i := range bucket.entries {
if bucket.entries[i].ID == node.ID {
bucket.entries = append(bucket.entries[:i], bucket.entries[i+1:]...)
tab.count--
return
}
}
}

func (tab *Table) deleteReplace(node *Node) {
b := tab.buckets[logdist(tab.self.sha, node.sha)]
i := 0
for i < len(b.entries) {
if b.entries[i].ID == node.ID {
b.entries = append(b.entries[:i], b.entries[i+1:]...)
tab.count--
} else {
i++
}
}
// refill from replacement cache
// TODO: maybe use random index
if len(b.entries) < bucketSize && len(b.replacements) > 0 {
ri := len(b.replacements) - 1
b.addFront(b.replacements[ri])
tab.count++
b.replacements[ri] = nil
b.replacements = b.replacements[:ri]
}
}

func (b *bucket) addFront(n *Node) {
b.entries = append(b.entries, nil)
copy(b.entries[1:], b.entries)
```

```
b.entries[0] = n
}

func (b *bucket) bump(n *Node) bool {
for i := range b.entries {
if b.entries[i].ID == n.ID {
// move it to the front
copy(b.entries[1:], b.entries[:i])
b.entries[0] = n
return true
}
}
return false
}

// nodesByDistance is a list of nodes, ordered by
// distance to target.
type nodesByDistance struct {
entries []*Node
target  common.Hash
}

// push adds the given node to the list, keeping the total size below maxElems.
func (h *nodesByDistance) push(n *Node, maxElems int) {
ix := sort.Search(len(h.entries), func(i int) bool {
return distcmp(h.target, h.entries[i].sha, n.sha) > 0
})
if len(h.entries) < maxElems {
h.entries = append(h.entries, n)
}
if ix == len(h.entries) {
// farther away than all nodes we already have.
// if there was room for it, the node is now the last element.
} else {
// slide existing entries down to make room
// this will overwrite the entry we just appended.
copy(h.entries[ix+1:], h.entries[ix:])
h.entries[ix] = n
}
}
```