F:\git\java\mar3\filemonitor\target\go-ethereum\go-ethereum-5.doc

0:F:\git\coin\ethereum\go-ethereum\p2p\discv5\ticket.go

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

package discv5

import (
"bytes"
"encoding/binary"
"fmt"
"math"
"math/rand"
"sort"
"time"

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

const (
ticketTimeBucketLen = time.Minute
timeWindow          = 10 // * ticketTimeBucketLen
wantTicketsInWindow = 10
collectFrequency    = time.Second * 30
registerFrequency   = time.Second * 60
maxCollectDebt      = 10
maxRegisterDebt     = 5
keepTicketConst     = time.Minute * 10
keepTicketExp       = time.Minute * 5
targetWaitTime      = time.Minute * 10
topicQueryTimeout   = time.Second * 5
topicQueryResend    = time.Minute
// topic radius detection
maxRadius           = 0xffffffffffffffff
radiusTC            = time.Minute * 20
radiusBucketsPerBit = 8
minSlope            = 1
minPeakSize         = 40
maxNoAdjust         = 20
lookupWidth         = 8
```

```
minRightSum        = 20
searchForceQuery    = 4
)

// timeBucket represents absolute monotonic time in minutes.
// It is used as the index into the per-topic ticket buckets.
type timeBucket int

type ticket struct {
topics  []Topic
regTime []mclock.AbsTime // Per-topic local absolute time when the ticket can be used.

// The serial number that was issued by the server.
serial uint32
// Used by registrar, tracks absolute time when the ticket was created.
issueTime mclock.AbsTime

// Fields used only by registrants
node   *Node  // the registrar node that signed this ticket
refCnt int    // tracks number of topics that will be registered using this ticket
pong   []byte // encoded pong packet signed by the registrar
}

// ticketRef refers to a single topic in a ticket.
type ticketRef struct {
t   *ticket
idx int // index of the topic in t.topics and t.regTime
}

func (ref ticketRef) topic() Topic {
return ref.t.topics[ref.idx]
}

func (ref ticketRef) topicRegTime() mclock.AbsTime {
return ref.t.regTime[ref.idx]
}

func pongToTicket(localTime mclock.AbsTime, topics []Topic, node *Node, p *ingressPacket)
(*ticket, error) {
wps := p.data.(*pong).WaitPeriods
if len(topics) != len(wps) {
return nil, fmt.Errorf("bad wait period list: got %d values, want %d", len(topics), len(wps))
}
if rlpHash(topics) != p.data.(*pong).TopicHash {
return nil, fmt.Errorf("bad topic hash")
}
t := &ticket{
issueTime: localTime,
node:      node,
topics:    topics,
pong:      p.rawData,
regTime:   make([]mclock.AbsTime, len(wps)),
}
// Convert wait periods to local absolute time.
for i, wp := range wps {
t.regTime[i] = localTime + mclock.AbsTime(time.Second*time.Duration(wp))
}
return t, nil
}

func ticketToPong(t *ticket, pong *pong) {
pong.Expiration = uint64(t.issueTime / mclock.AbsTime(time.Second))
pong.TopicHash = rlpHash(t.topics)
pong.TicketSerial = t.serial
pong.WaitPeriods = make([]uint32, len(t.regTime))
for i, regTime := range t.regTime {
pong.WaitPeriods[i] = uint32(time.Duration(regTime-t.issueTime) / time.Second)
}
}

type ticketStore struct {
// radius detector and target address generator
// exists for both searched and registered topics
radius map[Topic]*topicRadius

// Contains buckets (for each absolute minute) of tickets
// that can be used in that minute.
// This is only set if the topic is being registered.
tickets     map[Topic]topicTickets
regtopics   []Topic
nodes       map[*Node]*ticket
nodeLastReq map[*Node]reqInfo

lastBucketFetched timeBucket
```

```
nextTicketCached  *ticketRef
nextTicketReg     mclock.AbsTime

searchTopicMap        map[Topic]searchTopic
nextTopicQueryCleanup mclock.AbsTime
queriesSent           map[*Node]map[common.Hash]sentQuery
}

type searchTopic struct {
foundChn chan<- *Node
}

type sentQuery struct {
sent   mclock.AbsTime
lookup lookupInfo
}

type topicTickets struct {
buckets             map[timeBucket][]ticketRef
nextLookup, nextReg mclock.AbsTime
}

func newTicketStore() *ticketStore {
return &ticketStore{
radius:         make(map[Topic]*topicRadius),
tickets:        make(map[Topic]topicTickets),
nodes:          make(map[*Node]*ticket),
nodeLastReq:    make(map[*Node]reqInfo),
searchTopicMap: make(map[Topic]searchTopic),
queriesSent:    make(map[*Node]map[common.Hash]sentQuery),
}
}

// addTopic starts tracking a topic. If register is true,
// the local node will register the topic and tickets will be collected.
func (s *ticketStore) addTopic(t Topic, register bool) {
debugLog(fmt.Sprintf(" addTopic(%v, %v)", t, register))
if s.radius[t] == nil {
s.radius[t] = newTopicRadius(t)
}
if register && s.tickets[t].buckets == nil {
s.tickets[t] = topicTickets{buckets: make(map[timeBucket][]ticketRef)}
}
}

func (s *ticketStore) addSearchTopic(t Topic, foundChn chan<- *Node) {
s.addTopic(t, false)
if s.searchTopicMap[t].foundChn == nil {
s.searchTopicMap[t] = searchTopic{foundChn: foundChn}
}
}

func (s *ticketStore) removeSearchTopic(t Topic) {
if st := s.searchTopicMap[t]; st.foundChn != nil {
delete(s.searchTopicMap, t)
}
}

// removeRegisterTopic deletes all tickets for the given topic.
func (s *ticketStore) removeRegisterTopic(topic Topic) {
debugLog(fmt.Sprintf(" removeRegisterTopic(%v)", topic))
for _, list := range s.tickets[topic].buckets {
for _, ref := range list {
ref.t.refCnt--
if ref.t.refCnt == 0 {
delete(s.nodes, ref.t.node)
delete(s.nodeLastReq, ref.t.node)
}
}
}
delete(s.tickets, topic)
}

func (s *ticketStore) regTopicSet() []Topic {
topics := make([]Topic, 0, len(s.tickets))
for topic := range s.tickets {
topics = append(topics, topic)
}
return topics
}

// nextRegisterLookup returns the target of the next lookup for ticket collection.
func (s *ticketStore) nextRegisterLookup() (lookup lookupInfo, delay time.Duration) {
debugLog("nextRegisterLookup()")
```

```
firstTopic, ok := s.iterRegTopics()
for topic := firstTopic; ok; {
debugLog(fmt.Sprintf(" checking topic %v, len(s.tickets[topic]) = %d", topic,
len(s.tickets[topic].buckets)))
if s.tickets[topic].buckets != nil && s.needMoreTickets(topic) {
next := s.radius[topic].nextTarget(false)
debugLog(fmt.Sprintf(" %x 1s", next.target[:8]))
return next, 100 * time.Millisecond
}
topic, ok = s.iterRegTopics()
if topic == firstTopic {
break // We have checked all topics.
}
}
debugLog(" null, 40s")
return lookupInfo{}, 40 * time.Second
}

func (s *ticketStore) nextSearchLookup(topic Topic) lookupInfo {
tr := s.radius[topic]
target := tr.nextTarget(tr.radiusLookupCnt >= searchForceQuery)
if target.radiusLookup {
tr.radiusLookupCnt++
} else {
tr.radiusLookupCnt = 0
}
return target
}

// iterRegTopics returns topics to register in arbitrary order.
// The second return value is false if there are no topics.
func (s *ticketStore) iterRegTopics() (Topic, bool) {
debugLog("iterRegTopics()")
if len(s.regtopics) == 0 {
if len(s.tickets) == 0 {
debugLog(" false")
return "", false
}
// Refill register list.
for t := range s.tickets {
s.regtopics = append(s.regtopics, t)
}
}
topic := s.regtopics[len(s.regtopics)-1]
s.regtopics = s.regtopics[:len(s.regtopics)-1]
debugLog(" " + string(topic) + " true")
return topic, true
}

func (s *ticketStore) needMoreTickets(t Topic) bool {
return s.tickets[t].nextLookup < mclock.Now()
}

// ticketsInWindow returns the tickets of a given topic in the registration window.
func (s *ticketStore) ticketsInWindow(t Topic) []ticketRef {
ltBucket := s.lastBucketFetched
var res []ticketRef
tickets := s.tickets[t].buckets
for g := ltBucket; g < ltBucket+timeWindow; g++ {
res = append(res, tickets[g]...)
}
debugLog(fmt.Sprintf("ticketsInWindow(%v) = %v", t, len(res)))
return res
}

func (s *ticketStore) removeExcessTickets(t Topic) {
tickets := s.ticketsInWindow(t)
if len(tickets) <= wantTicketsInWindow {
return
}
sort.Sort(ticketRefByWaitTime(tickets))
for _, r := range tickets[wantTicketsInWindow:] {
s.removeTicketRef(r)
}
}

type ticketRefByWaitTime []ticketRef

// Len is the number of elements in the collection.
func (s ticketRefByWaitTime) Len() int {
return len(s)
}

func (r ticketRef) waitTime() mclock.AbsTime {
```

```
return r.t.regTime[r.idx] - r.t.issueTime
}

// Less reports whether the element with
// index i should sort before the element with index j.
func (s ticketRefByWaitTime) Less(i, j int) bool {
return s[i].waitTime() < s[j].waitTime()
}

// Swap swaps the elements with indexes i and j.
func (s ticketRefByWaitTime) Swap(i, j int) {
s[i], s[j] = s[j], s[i]
}

func (s *ticketStore) addTicketRef(r ticketRef) {
topic := r.t.topics[r.idx]
t := s.tickets[topic]
if t.buckets == nil {
return
}
bucket := timeBucket(r.t.regTime[r.idx] / mclock.AbsTime(ticketTimeBucketLen))
t.buckets[bucket] = append(t.buckets[bucket], r)
r.t.refCnt++

min := mclock.Now() - mclock.AbsTime(collectFrequency)*maxCollectDebt
if t.nextLookup < min {
t.nextLookup = min
}
t.nextLookup += mclock.AbsTime(collectFrequency)
s.tickets[topic] = t

//s.removeExcessTickets(topic)
}

func (s *ticketStore) nextFilteredTicket() (t *ticketRef, wait time.Duration) {
now := mclock.Now()
for {
t, wait = s.nextRegisterableTicket()
if t == nil {
return
}
regTime := now + mclock.AbsTime(wait)
topic := t.t.topics[t.idx]
if regTime >= s.tickets[topic].nextReg {
return
}
s.removeTicketRef(*t)
}
}

func (s *ticketStore) ticketRegistered(t ticketRef) {
now := mclock.Now()

topic := t.t.topics[t.idx]
tt := s.tickets[topic]
min := now - mclock.AbsTime(registerFrequency)*maxRegisterDebt
if min > tt.nextReg {
tt.nextReg = min
}
tt.nextReg += mclock.AbsTime(registerFrequency)
s.tickets[topic] = tt

s.removeTicketRef(t)
}

// nextRegisterableTicket returns the next ticket that can be used
// to register.
//
// If the returned wait time <= zero the ticket can be used. For a positive
// wait time, the caller should requery the next ticket later.
//
// A ticket can be returned more than once with <= zero wait time in case
// the ticket contains multiple topics.
func (s *ticketStore) nextRegisterableTicket() (t *ticketRef, wait time.Duration) {
defer func() {
if t == nil {
debugLog(" nil")
} else {
debugLog(fmt.Sprintf(" node = %x sn = %v wait = %v", t.t.node.ID[:8], t.t.serial, wait))
}
}()

debugLog("nextRegisterableTicket()")
now := mclock.Now()
```

```
if s.nextTicketCached != nil {
return s.nextTicketCached, time.Duration(s.nextTicketCached.topicRegTime() - now)
}

for bucket := s.lastBucketFetched; ; bucket++ {
var (
empty      = true    // true if there are no tickets
nextTicket ticketRef // uninitialized if this bucket is empty
)
for _, tickets := range s.tickets {
//s.removeExcessTickets(topic)
if len(tickets.buckets) != 0 {
empty = false
if list := tickets.buckets[bucket]; list != nil {
for _, ref := range list {
//debugLog(fmt.Sprintf(" nrt bucket = %d node = %x sn = %v wait = %v", bucket, ref.t.node.ID[:8],
ref.t.serial, time.Duration(ref.topicRegTime()-now)))
if nextTicket.t == nil || ref.topicRegTime() < nextTicket.topicRegTime() {
nextTicket = ref
}
}
}
}
}
if empty {
return nil, 0
}
if nextTicket.t != nil {
wait = time.Duration(nextTicket.topicRegTime() - now)
s.nextTicketCached = &nextTicket
return &nextTicket, wait
}
s.lastBucketFetched = bucket
}
}

// removeTicket removes a ticket from the ticket store
func (s *ticketStore) removeTicketRef(ref ticketRef) {
debugLog(fmt.Sprintf("removeTicketRef(node = %x sn = %v)", ref.t.node.ID[:8], ref.t.serial))
topic := ref.topic()
tickets := s.tickets[topic].buckets
if tickets == nil {
```

```
return
}
bucket := timeBucket(ref.t.regTime[ref.idx] / mclock.AbsTime(ticketTimeBucketLen))
list := tickets[bucket]
idx := -1
for i, bt := range list {
if bt.t == ref.t {
idx = i
break
}
}
if idx == -1 {
panic(nil)
}
list = append(list[:idx], list[idx+1:]...)
if len(list) != 0 {
tickets[bucket] = list
} else {
delete(tickets, bucket)
}
ref.t.refCnt--
if ref.t.refCnt == 0 {
delete(s.nodes, ref.t.node)
delete(s.nodeLastReq, ref.t.node)
}

// Make nextRegisterableTicket return the next available ticket.
s.nextTicketCached = nil
}

type lookupInfo struct {
target       common.Hash
topic        Topic
radiusLookup bool
}

type reqInfo struct {
pingHash []byte
lookup   lookupInfo
time     mclock.AbsTime
}
```

```
// returns -1 if not found
func (t *ticket) findIdx(topic Topic) int {
for i, tt := range t.topics {
if tt == topic {
return i
}
}
return -1
}

func (s *ticketStore) registerLookupDone(lookup lookupInfo, nodes []*Node, ping func(n *Node)
[]byte) {
now := mclock.Now()
for i, n := range nodes {
if i == 0 || (binary.BigEndian.Uint64(n.sha[:8])^binary.BigEndian.Uint64(lookup.target[:8])) <
s.radius[lookup.topic].minRadius {
if lookup.radiusLookup {
if lastReq, ok := s.nodeLastReq[n]; !ok || time.Duration(now-lastReq.time) > radiusTC {
s.nodeLastReq[n] = reqInfo{pingHash: ping(n), lookup: lookup, time: now}
}
} else {
if s.nodes[n] == nil {
s.nodeLastReq[n] = reqInfo{pingHash: ping(n), lookup: lookup, time: now}
}
}
}
}
}

func (s *ticketStore) searchLookupDone(lookup lookupInfo, nodes []*Node, ping func(n *Node)
[]byte, query func(n *Node, topic Topic) []byte) {
now := mclock.Now()
for i, n := range nodes {
if i == 0 || (binary.BigEndian.Uint64(n.sha[:8])^binary.BigEndian.Uint64(lookup.target[:8])) <
s.radius[lookup.topic].minRadius {
if lookup.radiusLookup {
if lastReq, ok := s.nodeLastReq[n]; !ok || time.Duration(now-lastReq.time) > radiusTC {
s.nodeLastReq[n] = reqInfo{pingHash: ping(n), lookup: lookup, time: now}
}
} // else {
if s.canQueryTopic(n, lookup.topic) {
hash := query(n, lookup.topic)
if hash != nil {
s.addTopicQuery(common.BytesToHash(hash), n, lookup)
}
}
//}
}
}
}

func (s *ticketStore) adjustWithTicket(now mclock.AbsTime, targetHash common.Hash, t *ticket) {
for i, topic := range t.topics {
if tt, ok := s.radius[topic]; ok {
tt.adjustWithTicket(now, targetHash, ticketRef{t, i})
}
}
}

func (s *ticketStore) addTicket(localTime mclock.AbsTime, pingHash []byte, t *ticket) {
debugLog(fmt.Sprintf("add(node = %x sn = %v)", t.node.ID[:8], t.serial))

lastReq, ok := s.nodeLastReq[t.node]
if !(ok && bytes.Equal(pingHash, lastReq.pingHash)) {
return
}
s.adjustWithTicket(localTime, lastReq.lookup.target, t)

if lastReq.lookup.radiusLookup || s.nodes[t.node] != nil {
return
}

topic := lastReq.lookup.topic
topicIdx := t.findIdx(topic)
if topicIdx == -1 {
return
}

bucket := timeBucket(localTime / mclock.AbsTime(ticketTimeBucketLen))
if s.lastBucketFetched == 0 || bucket < s.lastBucketFetched {
s.lastBucketFetched = bucket
}

if _, ok := s.tickets[topic]; ok {
```

```
wait := t.regTime[topicIdx] - localTime
rnd := rand.ExpFloat64()
if rnd > 10 {
rnd = 10
}
if float64(wait) < float64(keepTicketConst)+float64(keepTicketExp)*rnd {
// use the ticket to register this topic
//fmt.Println("addTicket", t.node.ID[:8], t.node.addr().String(), t.serial, t.pong)
s.addTicketRef(ticketRef{t, topicIdx})
}
}

if t.refCnt > 0 {
s.nextTicketCached = nil
s.nodes[t.node] = t
}
}

func (s *ticketStore) getNodeTicket(node *Node) *ticket {
if s.nodes[node] == nil {
debugLog(fmt.Sprintf("getNodeTicket(%x) sn = nil", node.ID[:8]))
} else {
debugLog(fmt.Sprintf("getNodeTicket(%x) sn = %v", node.ID[:8], s.nodes[node].serial))
}
return s.nodes[node]
}

func (s *ticketStore) canQueryTopic(node *Node, topic Topic) bool {
qq := s.queriesSent[node]
if qq != nil {
now := mclock.Now()
for _, sq := range qq {
if sq.lookup.topic == topic && sq.sent > now-mclock.AbsTime(topicQueryResend) {
return false
}
}
}
return true
}

func (s *ticketStore) addTopicQuery(hash common.Hash, node *Node, lookup lookupInfo) {
now := mclock.Now()
```

```
qq := s.queriesSent[node]
if qq == nil {
qq = make(map[common.Hash]sentQuery)
s.queriesSent[node] = qq
}
qq[hash] = sentQuery{sent: now, lookup: lookup}
s.cleanupTopicQueries(now)
}

func (s *ticketStore) cleanupTopicQueries(now mclock.AbsTime) {
if s.nextTopicQueryCleanup > now {
return
}
exp := now - mclock.AbsTime(topicQueryResend)
for n, qq := range s.queriesSent {
for h, q := range qq {
if q.sent < exp {
delete(qq, h)
}
}
if len(qq) == 0 {
delete(s.queriesSent, n)
}
}
s.nextTopicQueryCleanup = now + mclock.AbsTime(topicQueryTimeout)
}

func (s *ticketStore) gotTopicNodes(from *Node, hash common.Hash, nodes []rpcNode) (timeout
bool) {
now := mclock.Now()
//fmt.Println("got", from.addr().String(), hash, len(nodes))
qq := s.queriesSent[from]
if qq == nil {
return true
}
q, ok := qq[hash]
if !ok || now > q.sent+mclock.AbsTime(topicQueryTimeout) {
return true
}
inside := float64(0)
if len(nodes) > 0 {
inside = 1
}
s.radius[q.lookup.topic].adjust(now, q.lookup.target, from.sha, inside)
chn := s.searchTopicMap[q.lookup.topic].foundChn
if chn == nil {
//fmt.Println("no channel")
return false
}
for _, node := range nodes {
ip := node.IP
if ip.IsUnspecified() || ip.IsLoopback() {
ip = from.IP
}
n := NewNode(node.ID, ip, node.UDP-1, node.TCP-1) // subtract one from port while discv5 is
running in test mode on UDPport+1
select {
case chn <- n:
default:
return false
}
}
return false
}

type topicRadius struct {
topic           Topic
topicHashPrefix   uint64
radius, minRadius uint64
buckets          []topicRadiusBucket
converged        bool
radiusLookupCnt   int
}

type topicRadiusEvent int

const (
trOutside topicRadiusEvent = iota
trInside
trNoAdjust
trCount
)

type topicRadiusBucket struct {
```

```
weights    [trCount]float64
lastTime   mclock.AbsTime
value      float64
lookupSent map[common.Hash]mclock.AbsTime
}

func (b *topicRadiusBucket) update(now mclock.AbsTime) {
if now == b.lastTime {
return
}
exp := math.Exp(-float64(now-b.lastTime) / float64(radiusTC))
for i, w := range b.weights {
b.weights[i] = w * exp
}
b.lastTime = now

for target, tm := range b.lookupSent {
if now-tm > mclock.AbsTime(respTimeout) {
b.weights[trNoAdjust] += 1
delete(b.lookupSent, target)
}
}
}

func (b *topicRadiusBucket) adjust(now mclock.AbsTime, inside float64) {
b.update(now)
if inside <= 0 {
b.weights[trOutside] += 1
} else {
if inside >= 1 {
b.weights[trInside] += 1
} else {
b.weights[trInside] += inside
b.weights[trOutside] += 1 - inside
}
}
}

func newTopicRadius(t Topic) *topicRadius {
topicHash := crypto.Keccak256Hash([]byte(t))
topicHashPrefix := binary.BigEndian.Uint64(topicHash[0:8])
```

```
return &topicRadius{
topic:          t,
topicHashPrefix: topicHashPrefix,
radius:         maxRadius,
minRadius:      maxRadius,
}
}

func (r *topicRadius) getBucketIdx(addrHash common.Hash) int {
prefix := binary.BigEndian.Uint64(addrHash[0:8])
var log2 float64
if prefix != r.topicHashPrefix {
log2 = math.Log2(float64(prefix ^ r.topicHashPrefix))
}
bucket := int((64 - log2) * radiusBucketsPerBit)
max := 64*radiusBucketsPerBit - 1
if bucket > max {
return max
}
if bucket < 0 {
return 0
}
return bucket
}

func (r *topicRadius) targetForBucket(bucket int) common.Hash {
min := math.Pow(2, 64-float64(bucket+1)/radiusBucketsPerBit)
max := math.Pow(2, 64-float64(bucket)/radiusBucketsPerBit)
a := uint64(min)
b := randUint64n(uint64(max - min))
xor := a + b
if xor < a {
xor = ^uint64(0)
}
prefix := r.topicHashPrefix ^ xor
var target common.Hash
binary.BigEndian.PutUint64(target[0:8], prefix)
globalRandRead(target[8:])
return target
}

// package rand provides a Read function in Go 1.6 and later, but
// we can't use it yet because we still support Go 1.5.
func globalRandRead(b []byte) {
pos := 0
val := 0
for n := 0; n < len(b); n++ {
if pos == 0 {
val = rand.Int()
pos = 7
}
b[n] = byte(val)
val >>= 8
pos--
}
}

func (r *topicRadius) isInRadius(addrHash common.Hash) bool {
nodePrefix := binary.BigEndian.Uint64(addrHash[0:8])
dist := nodePrefix ^ r.topicHashPrefix
return dist < r.radius
}

func (r *topicRadius) chooseLookupBucket(a, b int) int {
if a < 0 {
a = 0
}
if a > b {
return -1
}
c := 0
for i := a; i <= b; i++ {
if i >= len(r.buckets) || r.buckets[i].weights[trNoAdjust] < maxNoAdjust {
c++
}
}
if c == 0 {
return -1
}
rnd := randUint(uint32(c))
for i := a; i <= b; i++ {
if i >= len(r.buckets) || r.buckets[i].weights[trNoAdjust] < maxNoAdjust {
if rnd == 0 {
return i
}
rnd--
}
}
panic(nil) // should never happen
}

func (r *topicRadius) needMoreLookups(a, b int, maxValue float64) bool {
var max float64
if a < 0 {
a = 0
}
if b >= len(r.buckets) {
b = len(r.buckets) - 1
if r.buckets[b].value > max {
max = r.buckets[b].value
}
}
if b >= a {
for i := a; i <= b; i++ {
if r.buckets[i].value > max {
max = r.buckets[i].value
}
}
}
return maxValue-max < minPeakSize
}

func (r *topicRadius) recalcRadius() (radius uint64, radiusLookup int) {
maxBucket := 0
maxValue := float64(0)
now := mclock.Now()
v := float64(0)
for i := range r.buckets {
r.buckets[i].update(now)
v += r.buckets[i].weights[trOutside] - r.buckets[i].weights[trInside]
r.buckets[i].value = v
//fmt.Printf("%v %v | ", v, r.buckets[i].weights[trNoAdjust])
}
//fmt.Println()
slopeCross := -1
for i, b := range r.buckets {
```

```
v := b.value
if v < float64(i)*minSlope {
slopeCross = i
break
}
if v > maxValue {
maxValue = v
maxBucket = i + 1
}
}

minRadBucket := len(r.buckets)
sum := float64(0)
for minRadBucket > 0 && sum < minRightSum {
minRadBucket--
b := r.buckets[minRadBucket]
sum += b.weights[trInside] + b.weights[trOutside]
}
r.minRadius = uint64(math.Pow(2, 64-float64(minRadBucket)/radiusBucketsPerBit))

lookupLeft := -1
if r.needMoreLookups(0, maxBucket-lookupWidth-1, maxValue) {
lookupLeft = r.chooseLookupBucket(maxBucket-lookupWidth, maxBucket-1)
}
lookupRight := -1
if slopeCross != maxBucket && (minRadBucket <= maxBucket ||
r.needMoreLookups(maxBucket+lookupWidth, len(r.buckets)-1, maxValue)) {
for len(r.buckets) <= maxBucket+lookupWidth {
r.buckets = append(r.buckets, topicRadiusBucket{lookupSent:
make(map[common.Hash]mclock.AbsTime)})
}
lookupRight = r.chooseLookupBucket(maxBucket, maxBucket+lookupWidth-1)
}
if lookupLeft == -1 {
radiusLookup = lookupRight
} else {
if lookupRight == -1 {
radiusLookup = lookupLeft
} else {
if randUint(2) == 0 {
radiusLookup = lookupLeft
} else {
```

```
radiusLookup = lookupRight
}
}
}

//fmt.Println("mb", maxBucket, "sc", slopeCross, "mrb", minRadBucket, "ll", lookupLeft, "lr",
lookupRight, "mv", maxValue)

if radiusLookup == -1 {
// no more radius lookups needed at the moment, return a radius
r.converged = true
rad := maxBucket
if minRadBucket < rad {
rad = minRadBucket
}
radius = ^uint64(0)
if rad > 0 {
radius = uint64(math.Pow(2, 64-float64(rad)/radiusBucketsPerBit))
}
r.radius = radius
}

return
}

func (r *topicRadius) nextTarget(forceRegular bool) lookupInfo {
if !forceRegular {
_, radiusLookup := r.recalcRadius()
if radiusLookup != -1 {
target := r.targetForBucket(radiusLookup)
r.buckets[radiusLookup].lookupSent[target] = mclock.Now()
return lookupInfo{target: target, topic: r.topic, radiusLookup: true}
}
}

radExt := r.radius / 2
if radExt > maxRadius-r.radius {
radExt = maxRadius - r.radius
}
rnd := randUint64n(r.radius) + randUint64n(2*radExt)
if rnd > radExt {
rnd -= radExt
```

```
} else {
rnd = radExt - rnd
}

prefix := r.topicHashPrefix ^ rnd
var target common.Hash
binary.BigEndian.PutUint64(target[0:8], prefix)
globalRandRead(target[8:])
return lookupInfo{target: target, topic: r.topic, radiusLookup: false}
}

func (r *topicRadius) adjustWithTicket(now mclock.AbsTime, targetHash common.Hash, t
ticketRef) {
wait := t.t.regTime[t.idx] - t.t.issueTime
inside := float64(wait)/float64(targetWaitTime) - 0.5
if inside > 1 {
inside = 1
}
if inside < 0 {
inside = 0
}
r.adjust(now, targetHash, t.t.node.sha, inside)
}

func (r *topicRadius) adjust(now mclock.AbsTime, targetHash, addrHash common.Hash, inside
float64) {
bucket := r.getBucketIdx(addrHash)
//fmt.Println("adjust", bucket, len(r.buckets), inside)
if bucket >= len(r.buckets) {
return
}
r.buckets[bucket].adjust(now, inside)
delete(r.buckets[bucket].lookupSent, targetHash)
}

1:F:\git\coin\ethereum\go-ethereum\p2p\discv5\topic.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discv5

import (
"container/heap"
```

```
"fmt"
"math"
"math/rand"
"time"

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

const (
maxEntries         = 10000
maxEntriesPerTopic = 50

fallbackRegistrationExpiry = 1 * time.Hour
)

type Topic string

type topicEntry struct {
topic   Topic
fifoIdx uint64
node    *Node
expire  mclock.AbsTime
}

type topicInfo struct {
entries            map[uint64]*topicEntry
fifoHead, fifoTail uint64
rqItem             *topicRequestQueueItem
wcl                waitControlLoop
}

// removes tail element from the fifo
func (t *topicInfo) getFifoTail() *topicEntry {
for t.entries[t.fifoTail] == nil {
t.fifoTail++
}
tail := t.entries[t.fifoTail]
t.fifoTail++
return tail
}

type nodeInfo struct {
```

```
entries                          map[Topic]*topicEntry
lastIssuedTicket, lastUsedTicket uint32
// you can't register a ticket newer than lastUsedTicket before noRegUntil (absolute time)
noRegUntil mclock.AbsTime
}

type topicTable struct {
db                    *nodeDB
self                  *Node
nodes                  map[*Node]*nodeInfo
topics                 map[Topic]*topicInfo
globalEntries         uint64
requested             topicRequestQueue
requestCnt            uint64
lastGarbageCollection mclock.AbsTime
}

func newTopicTable(db *nodeDB, self *Node) *topicTable {
if printTestImgLogs {
fmt.Printf("*N %016x\n", self.sha[:8])
}
return &topicTable{
db:     db,
nodes:  make(map[*Node]*nodeInfo),
topics: make(map[Topic]*topicInfo),
self:   self,
}
}

func (t *topicTable) getOrNewTopic(topic Topic) *topicInfo {
ti := t.topics[topic]
if ti == nil {
rqItem := &topicRequestQueueItem{
topic:    topic,
priority: t.requestCnt,
}
ti = &topicInfo{
entries: make(map[uint64]*topicEntry),
rqItem:  rqItem,
}
t.topics[topic] = ti
heap.Push(&t.requested, rqItem)
}
return ti
}

func (t *topicTable) checkDeleteTopic(topic Topic) {
ti := t.topics[topic]
if ti == nil {
return
}
if len(ti.entries) == 0 && ti.wcl.hasMinimumWaitPeriod() {
delete(t.topics, topic)
heap.Remove(&t.requested, ti.rqItem.index)
}
}

func (t *topicTable) getOrNewNode(node *Node) *nodeInfo {
n := t.nodes[node]
if n == nil {
//fmt.Printf("newNode %016x %016x\n", t.self.sha[:8], node.sha[:8])
var issued, used uint32
if t.db != nil {
issued, used = t.db.fetchTopicRegTickets(node.ID)
}
n = &nodeInfo{
entries:          make(map[Topic]*topicEntry),
lastIssuedTicket: issued,
lastUsedTicket:   used,
}
t.nodes[node] = n
}
return n
}

func (t *topicTable) checkDeleteNode(node *Node) {
if n, ok := t.nodes[node]; ok && len(n.entries) == 0 && n.noRegUntil < mclock.Now() {
//fmt.Printf("deleteNode %016x %016x\n", t.self.sha[:8], node.sha[:8])
delete(t.nodes, node)
}
}

func (t *topicTable) storeTicketCounters(node *Node) {
n := t.getOrNewNode(node)
```

```
if t.db != nil {
t.db.updateTopicRegTickets(node.ID, n.lastIssuedTicket, n.lastUsedTicket)
}
}

func (t *topicTable) getEntries(topic Topic) []*Node {
t.collectGarbage()

te := t.topics[topic]
if te == nil {
return nil
}
nodes := make([]*Node, len(te.entries))
i := 0
for _, e := range te.entries {
nodes[i] = e.node
i++
}
t.requestCnt++
t.requested.update(te.rqItem, t.requestCnt)
return nodes
}

func (t *topicTable) addEntry(node *Node, topic Topic) {
n := t.getOrNewNode(node)
// clear previous entries by the same node
for _, e := range n.entries {
t.deleteEntry(e)
}
// ***
n = t.getOrNewNode(node)

tm := mclock.Now()
te := t.getOrNewTopic(topic)

if len(te.entries) == maxEntriesPerTopic {
t.deleteEntry(te.getFifoTail())
}

if t.globalEntries == maxEntries {
t.deleteEntry(t.leastRequested()) // not empty, no need to check for nil
}
```

```
fifoIdx := te.fifoHead
te.fifoHead++
entry := &topicEntry{
topic:   topic,
fifoIdx: fifoIdx,
node:    node,
expire:  tm + mclock.AbsTime(fallbackRegistrationExpiry),
}
if printTestImgLogs {
fmt.Printf("*+ %d %v %016x %016x\n", tm/1000000, topic, t.self.sha[:8], node.sha[:8])
}
te.entries[fifoIdx] = entry
n.entries[topic] = entry
t.globalEntries++
te.wcl.registered(tm)
}

// removes least requested element from the fifo
func (t *topicTable) leastRequested() *topicEntry {
for t.requested.Len() > 0 && t.topics[t.requested[0].topic] == nil {
heap.Pop(&t.requested)
}
if t.requested.Len() == 0 {
return nil
}
return t.topics[t.requested[0].topic].getFifoTail()
}

// entry should exist
func (t *topicTable) deleteEntry(e *topicEntry) {
if printTestImgLogs {
fmt.Printf("*- %d %v %016x %016x\n", mclock.Now()/1000000, e.topic, t.self.sha[:8],
e.node.sha[:8])
}
ne := t.nodes[e.node].entries
delete(ne, e.topic)
if len(ne) == 0 {
t.checkDeleteNode(e.node)
}
te := t.topics[e.topic]
delete(te.entries, e.fifoIdx)
```

```
if len(te.entries) == 0 {
t.checkDeleteTopic(e.topic)
}
t.globalEntries--
}

// It is assumed that topics and waitPeriods have the same length.
func (t *topicTable) useTicket(node *Node, serialNo uint32, topics []Topic, idx int, issueTime
uint64, waitPeriods []uint32) (registered bool) {
debugLog(fmt.Sprintf("useTicket %v %v %v", serialNo, topics, waitPeriods))
//fmt.Println("useTicket", serialNo, topics, waitPeriods)
t.collectGarbage()

n := t.getOrNewNode(node)
if serialNo < n.lastUsedTicket {
return false
}

tm := mclock.Now()
if serialNo > n.lastUsedTicket && tm < n.noRegUntil {
return false
}
if serialNo != n.lastUsedTicket {
n.lastUsedTicket = serialNo
n.noRegUntil = tm + mclock.AbsTime(noRegTimeout())
t.storeTicketCounters(node)
}

currTime := uint64(tm / mclock.AbsTime(time.Second))
regTime := issueTime + uint64(waitPeriods[idx])
relTime := int64(currTime - regTime)
if relTime >= -1 && relTime <= regTimeWindow+1 { // give clients a little security margin on both
ends
if e := n.entries[topics[idx]]; e == nil {
t.addEntry(node, topics[idx])
} else {
// if there is an active entry, don't move to the front of the FIFO but prolong expire time
e.expire = tm + mclock.AbsTime(fallbackRegistrationExpiry)
}
return true
}
```

```
return false
}

func (topictab *topicTable) getTicket(node *Node, topics []Topic) *ticket {
topictab.collectGarbage()

now := mclock.Now()
n := topictab.getOrNewNode(node)
n.lastIssuedTicket++
topictab.storeTicketCounters(node)

t := &ticket{
issueTime: now,
topics:    topics,
serial:    n.lastIssuedTicket,
regTime:   make([]mclock.AbsTime, len(topics)),
}
for i, topic := range topics {
var waitPeriod time.Duration
if topic := topictab.topics[topic]; topic != nil {
waitPeriod = topic.wcl.waitPeriod
} else {
waitPeriod = minWaitPeriod
}

t.regTime[i] = now + mclock.AbsTime(waitPeriod)
}
return t
}

const gcInterval = time.Minute

func (t *topicTable) collectGarbage() {
tm := mclock.Now()
if time.Duration(tm-t.lastGarbageCollection) < gcInterval {
return
}
t.lastGarbageCollection = tm

for node, n := range t.nodes {
for _, e := range n.entries {
if e.expire <= tm {
```

```
t.deleteEntry(e)
}
}

t.checkDeleteNode(node)
}

for topic := range t.topics {
t.checkDeleteTopic(topic)
}
}

const (
minWaitPeriod   = time.Minute
regTimeWindow   = 10 // seconds
avgnoRegTimeout = time.Minute * 10
// target average interval between two incoming ad requests
wcTargetRegInterval = time.Minute * 10 / maxEntriesPerTopic
//
wcTimeConst = time.Minute * 10
)

// initialization is not required, will set to minWaitPeriod at first registration
type waitControlLoop struct {
lastIncoming mclock.AbsTime
waitPeriod   time.Duration
}

func (w *waitControlLoop) registered(tm mclock.AbsTime) {
w.waitPeriod = w.nextWaitPeriod(tm)
w.lastIncoming = tm
}

func (w *waitControlLoop) nextWaitPeriod(tm mclock.AbsTime) time.Duration {
period := tm - w.lastIncoming
wp := time.Duration(float64(w.waitPeriod) * math.Exp((float64(wcTargetRegInterval)-
float64(period))/float64(wcTimeConst)))
if wp < minWaitPeriod {
wp = minWaitPeriod
}
return wp
}
```

```
func (w *waitControlLoop) hasMinimumWaitPeriod() bool {
return w.nextWaitPeriod(mclock.Now()) == minWaitPeriod
}

func noRegTimeout() time.Duration {
e := rand.ExpFloat64()
if e > 100 {
e = 100
}
return time.Duration(float64(avgnoRegTimeout) * e)
}

type topicRequestQueueItem struct {
topic    Topic
priority uint64
index    int
}

// A topicRequestQueue implements heap.Interface and holds topicRequestQueueItems.
type topicRequestQueue []*topicRequestQueueItem

func (tq topicRequestQueue) Len() int { return len(tq) }

func (tq topicRequestQueue) Less(i, j int) bool {
return tq[i].priority < tq[j].priority
}

func (tq topicRequestQueue) Swap(i, j int) {
tq[i], tq[j] = tq[j], tq[i]
tq[i].index = i
tq[j].index = j
}

func (tq *topicRequestQueue) Push(x interface{}) {
n := len(*tq)
item := x.(*topicRequestQueueItem)
item.index = n
*tq = append(*tq, item)
}

func (tq *topicRequestQueue) Pop() interface{} {
```

```
old := *tq
n := len(old)
item := old[n-1]
item.index = -1
*tq = old[0 : n-1]
return item
}

func (tq *topicRequestQueue) update(item *topicRequestQueueItem, priority uint64) {
item.priority = priority
heap.Fix(tq, item.index)
}

2:F:\git\coin\ethereum\go-ethereum\p2p\discv5\topic_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discv5

import (
"encoding/binary"
"testing"
"time"

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

func TestTopicRadius(t *testing.T) {
now := mclock.Now()
topic := Topic("qwerty")
rad := newTopicRadius(topic)
targetRad := (^uint64(0)) / 100

waitFn := func(addr common.Hash) time.Duration {
prefix := binary.BigEndian.Uint64(addr[0:8])
dist := prefix ^ rad.topicHashPrefix
relDist := float64(dist) / float64(targetRad)
relTime := (1 - relDist/2) * 2
if relTime < 0 {
relTime = 0
}
return time.Duration(float64(targetWaitTime) * relTime)
}

bcnt := 0
cnt := 0
var sum float64
for cnt < 100 {
addr := rad.nextTarget(false).target
wait := waitFn(addr)
ticket := &ticket{
topics:  []Topic{topic},
regTime: []mclock.AbsTime{mclock.AbsTime(wait)},
node:    &Node{nodeNetGuts: nodeNetGuts{sha: addr}},
}
rad.adjustWithTicket(now, addr, ticketRef{ticket, 0})
if rad.radius != maxRadius {
cnt++
sum += float64(rad.radius)
} else {
bcnt++
if bcnt > 500 {
t.Errorf("Radius did not converge in 500 iterations")
}
}
}
avgRel := sum / float64(cnt) / float64(targetRad)
if avgRel > 1.05 || avgRel < 0.95 {
t.Errorf("Average/target ratio is too far from 1 (%v)", avgRel)
}
}

3:F:\git\coin\ethereum\go-ethereum\p2p\discv5\udp.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/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 request structures
type (
ping struct {
Version    uint
From, To   rpcEndpoint
Expiration uint64
// v5
Topics []Topic

// 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.

// v5
TopicHash    common.Hash
TicketSerial uint32
WaitPeriods  []uint32

// 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"`
}

// findnode is a query for nodes close to the given target.
findnodeHash struct {
Target     common.Hash
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"`
}

topicRegister struct {
Topics []Topic
Idx    uint
Pong   []byte
}

topicQuery struct {
Topic      Topic
Expiration uint64
}

// reply to topicQuery
topicNodes struct {
Echo  common.Hash
Nodes []rpcNode
}

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
}
)

const (
macSize  = 256 / 8
sigSize  = 520 / 8
```

```
headSize = macSize + sigSize // space of packet frame data
)

// 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.
var maxNeighbors = func() int {
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 {
return n
}
}
}()

var maxTopicNodes = func() int {
p := topicNodes{}
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 {
return n
}
}
}()

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 (e1 rpcEndpoint) equal(e2 rpcEndpoint) bool {
return e1.UDP == e2.UDP && e1.TCP == e2.TCP && e1.IP.Equal(e2.IP)
}

func nodeFromRPC(sender *net.UDPAddr, rn rpcNode) (*Node, error) {
if err := netutil.CheckRelayIP(sender.IP, rn.IP); err != nil {
return nil, err
}
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 ingressPacket struct {
remoteID   NodeID
remoteAddr *net.UDPAddr
ev         nodeEvent
hash       []byte
data       interface{} // one of the RPC structs
rawData    []byte
}

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
priv        *ecdsa.PrivateKey
ourEndpoint rpcEndpoint
```

```
nat        nat.Interface
net        *Network
}

// 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) (*Network, error) {
transport, err := listenUDP(priv, laddr)
if err != nil {
return nil, err
}
net, err := newNetwork(transport, priv.PublicKey, natm, nodeDBPath, netrestrict)
if err != nil {
return nil, err
}
transport.net = net
go transport.readLoop()
return net, nil
}

func listenUDP(priv *ecdsa.PrivateKey, laddr string) (*udp, 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
}
return &udp{conn: conn, priv: priv, ourEndpoint: makeEndpoint(addr, uint16(addr.Port))}, nil
}

func (t *udp) localAddr() *net.UDPAddr {
return t.conn.LocalAddr().(*net.UDPAddr)
}

func (t *udp) Close() {
t.conn.Close()
}

func (t *udp) send(remote *Node, ptype nodeEvent, data interface{}) (hash []byte) {
hash, _ = t.sendPacket(remote.ID, remote.addr(), byte(ptype), data)
```

```
return hash
}

func (t *udp) sendPing(remote *Node, toaddr *net.UDPAddr, topics []Topic) (hash []byte) {
hash, _ = t.sendPacket(remote.ID, toaddr, byte(pingPacket), ping{
Version:    Version,
From:       t.ourEndpoint,
To:         makeEndpoint(toaddr, uint16(toaddr.Port)), // TODO: maybe use known TCP port from
DB
Expiration: uint64(time.Now().Add(expiration).Unix()),
Topics:     topics,
})
return hash
}

func (t *udp) sendFindnode(remote *Node, target NodeID) {
t.sendPacket(remote.ID, remote.addr(), byte(findnodePacket), findnode{
Target:     target,
Expiration: uint64(time.Now().Add(expiration).Unix()),
})
}

func (t *udp) sendNeighbours(remote *Node, results []*Node) {
// Send neighbors in chunks with at most maxNeighbors per packet
// to stay below the 1280 byte limit.
p := neighbors{Expiration: uint64(time.Now().Add(expiration).Unix())}
for i, result := range results {
p.Nodes = append(p.Nodes, nodeToRPC(result))
if len(p.Nodes) == maxNeighbors || i == len(results)-1 {
t.sendPacket(remote.ID, remote.addr(), byte(neighborsPacket), p)
p.Nodes = p.Nodes[:0]
}
}
}

func (t *udp) sendFindnodeHash(remote *Node, target common.Hash) {
t.sendPacket(remote.ID, remote.addr(), byte(findnodeHashPacket), findnodeHash{
Target:     target,
Expiration: uint64(time.Now().Add(expiration).Unix()),
})
}
```

```
func (t *udp) sendTopicRegister(remote *Node, topics []Topic, idx int, pong []byte) {
t.sendPacket(remote.ID, remote.addr(), byte(topicRegisterPacket), topicRegister{
Topics: topics,
Idx:    uint(idx),
Pong:   pong,
})
}

func (t *udp) sendTopicNodes(remote *Node, queryHash common.Hash, nodes []*Node) {
p := topicNodes{Echo: queryHash}
if len(nodes) == 0 {
t.sendPacket(remote.ID, remote.addr(), byte(topicNodesPacket), p)
return
}
for i, result := range nodes {
if netutil.CheckRelayIP(remote.IP, result.IP) != nil {
continue
}
p.Nodes = append(p.Nodes, nodeToRPC(result))
if len(p.Nodes) == maxTopicNodes || i == len(nodes)-1 {
t.sendPacket(remote.ID, remote.addr(), byte(topicNodesPacket), p)
p.Nodes = p.Nodes[:0]
}
}
}

func (t *udp) sendPacket(toid NodeID, toaddr *net.UDPAddr, ptype byte, req interface{}) (hash
[]byte, err error) {
//fmt.Println("sendPacket", nodeEvent(ptype), toaddr.String(), toid.String())
packet, hash, err := encodePacket(t.priv, ptype, req)
if err != nil {
//fmt.Println(err)
return hash, err
}
log.Trace(fmt.Sprintf(">>> %v to %x@%v", nodeEvent(ptype), toid[:8], toaddr))
if _, err = t.conn.WriteToUDP(packet, toaddr); err != nil {
log.Trace(fmt.Sprint("UDP send failed:", err))
}
//fmt.Println(err)
return hash, err
}
```

```
// zeroed padding space for encodePacket.
var headSpace = make([]byte, headSize)

func encodePacket(priv *ecdsa.PrivateKey, ptype byte, req interface{}) (p, hash []byte, err error) {
b := new(bytes.Buffer)
b.Write(headSpace)
b.WriteByte(ptype)
if err := rlp.Encode(b, req); err != nil {
log.Error(fmt.Sprint("error encoding packet:", err))
return nil, nil, err
}
packet := b.Bytes()
sig, err := crypto.Sign(crypto.Keccak256(packet[headSize:]), priv)
if err != nil {
log.Error(fmt.Sprint("could not sign packet:", err))
return nil, nil, err
}
copy(packet[macSize:], sig)
// add the hash to the front. Note: this doesn't protect the
// packet in any way.
hash = crypto.Keccak256(packet[macSize:])
copy(packet, hash)
return packet, hash, nil
}

// readLoop runs in its own goroutine. it injects ingress UDP packets
// into the network loop.
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(fmt.Sprintf("Temporary read error: %v", err))
continue
} else if err != nil {
// Shut down the loop for permament errors.
log.Debug(fmt.Sprintf("Read error: %v", err))
return
}
t.handlePacket(from, buf[:nbytes])
}
}

func (t *udp) handlePacket(from *net.UDPAddr, buf []byte) error {
pkt := ingressPacket{remoteAddr: from}
if err := decodePacket(buf, &pkt); err != nil {
log.Debug(fmt.Sprintf("Bad packet from %v: %v", from, err))
//fmt.Println("bad packet", err)
return err
}
t.net.reqReadPacket(pkt)
return nil
}

func decodePacket(buffer []byte, pkt *ingressPacket) error {
if len(buffer) < headSize+1 {
return errPacketTooSmall
}
buf := make([]byte, len(buffer))
copy(buf, buffer)
hash, sig, sigdata := buf[:macSize], buf[macSize:headSize], buf[headSize:]
shouldhash := crypto.Keccak256(buf[macSize:])
if !bytes.Equal(hash, shouldhash) {
return errBadHash
}
fromID, err := recoverNodeID(crypto.Keccak256(buf[headSize:]), sig)
if err != nil {
return err
}
pkt.rawData = buf
pkt.hash = hash
pkt.remoteID = fromID
switch pkt.ev = nodeEvent(sigdata[0]); pkt.ev {
case pingPacket:
pkt.data = new(ping)
case pongPacket:
pkt.data = new(pong)
case findnodePacket:
pkt.data = new(findnode)
```

```
case neighborsPacket:
pkt.data = new(neighbors)
case findnodeHashPacket:
pkt.data = new(findnodeHash)
case topicRegisterPacket:
pkt.data = new(topicRegister)
case topicQueryPacket:
pkt.data = new(topicQuery)
case topicNodesPacket:
pkt.data = new(topicNodes)
default:
return fmt.Errorf("unknown packet type: %d", sigdata[0])
}
s := rlp.NewStream(bytes.NewReader(sigdata[1:]), 0)
err = s.Decode(pkt.data)
return err
}

4:F:\git\coin\ethereum\go-ethereum\p2p\discv5\udp_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package discv5

import (
"encoding/hex"
"errors"
"io"
"net"
"reflect"
"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{1, 2, 3, 4}, Port: 30303},
// }
// test.table, test.udp, _ = newUDP(test.localkey, test.pipe, 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_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:
// if !reflect.DeepEqual(result, list) {
// t.Errorf("neighbors mismatch:\n  got:  %v\n  want: %v", result, list)
// }
// 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 !bytes.Equal(n.IP, 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
232896a36b737666c55265ad0a68412f250001ea04cb847f000001820cfa8215a8d790000000000000
00000000000000000000018208ae820d058443b9a355",
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
be2be5a685a80971ddcfa80cb422cdd0101ec04cb847f000001820cfa8215a8d790000000000000
0000000000000000000018208ae820d058443b9a3550102",
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("fbc914b16819237dcd8801d7e53f69e9719adecb3cc0e790c57e91ca4461c9
54"),
Expiration: 1136239445,
Rest:       []rlp.RawValue{{0xC6, 0x01, 0x02, 0x03, 0xC2, 0x04, 0x05}, {0x06}},
},
},
{
input:
"c7c44041b9f7c7e41934417ebac9a8e1a4c6298f74553f2fcfdcae6ed6fe53163eb3d2b52e39fe918
31b8a927bf4fc222c3902202027e5e9eb812195f95d20061ef5cd31d502e47ecb61183f74a504fe04
c51e73df81f25c4d506b26db4517490103f84eb840ca634cae0d49acb401d8a4c6b6fe8c55b70d115
bf400769cc1400f3258cd31387574077f301b421bc84df7266c44e9e6d569fc56be00812904767bf5c
cd1fc7f8443b9a35582999983999999280dc62cc8255c73471e0a61da0c89acdc0e035e260add7fc
0c04ad9ebf3919644c91cb247affc82b69bd2ca235c71eab8e49737c937a2c396",
wantPacket: &findnode{
Target:
MustHexID("ca634cae0d49acb401d8a4c6b6fe8c55b70d115bf400769cc1400f3258cd3138757407
7f301b421bc84df7266c44e9e6d569fc56be00812904767bf5ccd1fc7f"),
Expiration: 1136239445,
Rest:       []rlp.RawValue{{0x82, 0x99, 0x99}, {0x83, 0x99, 0x99, 0x99}},
},
},
{
input:
"c679fc8fe0b8b12f06577f2e802d34f6fa257e6137a995f6f4cbfc9ee50ed3710faf6e66f932c4c8d81d
64343f429651328758b47d3dbc02c4042f0fff6946a50f4a49037a72bb550f3a7872363a83e1b9ee64
69856c24eb4ef80b7535bcf99c0004f9015bf90150f84d846321163782115c82115db8403155e1427
f85f10a5c9a7755877748041af1bcd8d474ec065eb33df57a97babf54bfd2103575fa829115d224c52
3596b401065a97f74010610fce76382c0bf32f84984010203040101b840312c55512422cf9b8a4097
e9a6ad79402e87a15ae909a4bfefa22398f03d20951933beea1e4dfa6f968212385e829f04c2d314fc
2d4e255e0d3bc08792b069dbf8599020010db83c4d001500000000abcdef12820d05820d05b8403
8643200b172dcfef857492156971f0e6aa2c538d8b74010f8e140811d53b98c765dd2d9612605191
3f44582e8c199ad7c6d6819e9a56483f637feaac9448aacf8599020010db885a308d313198a2e037
073488203e78203e8b8408dcab8618c3253b558d459da53bd8fa68935a719aff8b811197101a4b2b
47dd2d47295286fc00cc081bb542d760717d1bdd6bec2c37cd72eca367d6dd3b9df738443b9a3550
10203b525a138aa34383fec3d2719a0",
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) {
t.Skip("skipped while working on discovery v5")

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)
}
var pkt ingressPacket
if err := decodePacket(input, &pkt); err != nil {
t.Errorf("did not accept packet %s\n%v", test.input, err)
continue
}
if !reflect.DeepEqual(pkt.data, test.wantPacket) {
t.Errorf("got %s\nwant %s", spew.Sdump(pkt.data), spew.Sdump(test.wantPacket))
}
if pkt.remoteID != wantNodeID {
t.Errorf("got id %v\nwant id %v", pkt.remoteID, 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
}
```

5:F:\git\coin\ethereum\go-ethereum\p2p\message.go

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

package p2p

import (
"bytes"
"errors"
"fmt"
"io"
"io/ioutil"
"net"
"sync"
"sync/atomic"
"time"

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

// Msg defines the structure of a p2p message.
//
// Note that a Msg can only be sent once since the Payload reader is
// consumed during sending. It is not possible to create a Msg and
// send it any number of times. If you want to reuse an encoded
// structure, encode the payload into a byte array and create a
// separate Msg with a bytes.Reader as Payload for each send.
type Msg struct {
Code       uint64
Size       uint32 // size of the paylod
Payload    io.Reader
ReceivedAt time.Time
}

// Decode parses the RLP content of a message into
// the given value, which must be a pointer.
//
// For the decoding rules, please see package rlp.
func (msg Msg) Decode(val interface{}) error {
s := rlp.NewStream(msg.Payload, uint64(msg.Size))
if err := s.Decode(val); err != nil {
return newPeerError(errInvalidMsg, "(code %x) (size %d) %v", msg.Code, msg.Size, err)
}
return nil
}

func (msg Msg) String() string {
return fmt.Sprintf("msg #%v (%v bytes)", msg.Code, msg.Size)
}

// Discard reads any remaining payload data into a black hole.
func (msg Msg) Discard() error {
_, err := io.Copy(ioutil.Discard, msg.Payload)
return err
}

type MsgReader interface {
ReadMsg() (Msg, error)
}
```

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

// MsgReadWriter provides reading and writing of encoded messages.
// Implementations should ensure that ReadMsg and WriteMsg can be
// called simultaneously from multiple goroutines.
type MsgReadWriter interface {
MsgReader
MsgWriter
}

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

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

// netWrapper wraps a MsgReadWriter with locks around
// ReadMsg/WriteMsg and applies read/write deadlines.
```

```
type netWrapper struct {
rmu, wmu sync.Mutex

rtimeout, wtimeout time.Duration
conn              net.Conn
wrapped           MsgReadWriter
}

func (rw *netWrapper) ReadMsg() (Msg, error) {
rw.rmu.Lock()
defer rw.rmu.Unlock()
rw.conn.SetReadDeadline(time.Now().Add(rw.rtimeout))
return rw.wrapped.ReadMsg()
}

func (rw *netWrapper) WriteMsg(msg Msg) error {
rw.wmu.Lock()
defer rw.wmu.Unlock()
rw.conn.SetWriteDeadline(time.Now().Add(rw.wtimeout))
return rw.wrapped.WriteMsg(msg)
}

// eofSignal wraps a reader with eof signaling. the eof channel is
// closed when the wrapped reader returns an error or when count bytes
// have been read.
type eofSignal struct {
wrapped io.Reader
count   uint32 // number of bytes left
eof     chan<- struct{}
}

// note: when using eofSignal to detect whether a message payload
// has been read, Read might not be called for zero sized messages.
func (r *eofSignal) Read(buf []byte) (int, error) {
if r.count == 0 {
if r.eof != nil {
r.eof <- struct{}{}
r.eof = nil
}
return 0, io.EOF
}
```

```
max := len(buf)
if int(r.count) < len(buf) {
max = int(r.count)
}
n, err := r.wrapped.Read(buf[:max])
r.count -= uint32(n)
if (err != nil || r.count == 0) && r.eof != nil {
r.eof <- struct{}{} // tell Peer that msg has been consumed
r.eof = nil
}
return n, err
}

// MsgPipe creates a message pipe. Reads on one end are matched
// with writes on the other. The pipe is full-duplex, both ends
// implement MsgReadWriter.
func MsgPipe() (*MsgPipeRW, *MsgPipeRW) {
var (
c1, c2  = make(chan Msg), make(chan Msg)
closing = make(chan struct{})
closed  = new(int32)
rw1     = &MsgPipeRW{c1, c2, closing, closed}
rw2     = &MsgPipeRW{c2, c1, closing, closed}
)
return rw1, rw2
}

// ErrPipeClosed is returned from pipe operations after the
// pipe has been closed.
var ErrPipeClosed = errors.New("p2p: read or write on closed message pipe")

// MsgPipeRW is an endpoint of a MsgReadWriter pipe.
type MsgPipeRW struct {
w       chan<- Msg
r       <-chan Msg
closing chan struct{}
closed  *int32
}

// WriteMsg sends a messsage on the pipe.
// It blocks until the receiver has consumed the message payload.
func (p *MsgPipeRW) WriteMsg(msg Msg) error {
```

```
if atomic.LoadInt32(p.closed) == 0 {
consumed := make(chan struct{}, 1)
msg.Payload = &eofSignal{msg.Payload, msg.Size, consumed}
select {
case p.w <- msg:
if msg.Size > 0 {
// wait for payload read or discard
select {
case <-consumed:
case <-p.closing:
}
}
return nil
case <-p.closing:
}
}
return ErrPipeClosed
}

// ReadMsg returns a message sent on the other end of the pipe.
func (p *MsgPipeRW) ReadMsg() (Msg, error) {
if atomic.LoadInt32(p.closed) == 0 {
select {
case msg := <-p.r:
return msg, nil
case <-p.closing:
}
}
return Msg{}, ErrPipeClosed
}

// Close unblocks any pending ReadMsg and WriteMsg calls on both ends
// of the pipe. They will return ErrPipeClosed. Close also
// interrupts any reads from a message payload.
func (p *MsgPipeRW) Close() error {
if atomic.AddInt32(p.closed, 1) != 1 {
// someone else is already closing
atomic.StoreInt32(p.closed, 1) // avoid overflow
return nil
}
close(p.closing)
return nil
```

```
}

// ExpectMsg reads a message from r and verifies that its
// code and encoded RLP content match the provided values.
// If content is nil, the payload is discarded and not verified.
func ExpectMsg(r MsgReader, code uint64, content interface{}) error {
msg, err := r.ReadMsg()
if err != nil {
return err
}
if msg.Code != code {
return fmt.Errorf("message code mismatch: got %d, expected %d", msg.Code, code)
}
if content == nil {
return msg.Discard()
} else {
contentEnc, err := rlp.EncodeToBytes(content)
if err != nil {
panic("content encode error: " + err.Error())
}
if int(msg.Size) != len(contentEnc) {
return fmt.Errorf("message size mismatch: got %d, want %d", msg.Size, len(contentEnc))
}
actualContent, err := ioutil.ReadAll(msg.Payload)
if err != nil {
return err
}
if !bytes.Equal(actualContent, contentEnc) {
return fmt.Errorf("message payload mismatch:\ngot:  %x\nwant: %x", actualContent, contentEnc)
}
}
return nil
}
```

6:F:\git\coin\ethereum\go-ethereum\p2p\message_test.go

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

package p2p

import (
"bytes"
"encoding/hex"
```

```
"fmt"
"io"
"runtime"
"strings"
"testing"
"time"
)

func ExampleMsgPipe() {
rw1, rw2 := MsgPipe()
go func() {
Send(rw1, 8, [][]byte{{0, 0}})
Send(rw1, 5, [][]byte{{1, 1}})
rw1.Close()
}()

for {
msg, err := rw2.ReadMsg()
if err != nil {
break
}
var data [][]byte
msg.Decode(&data)
fmt.Printf("msg: %d, %x\n", msg.Code, data[0])
}
// Output:
// msg: 8, 0000
// msg: 5, 0101
}

func TestMsgPipeUnblockWrite(t *testing.T) {
loop:
for i := 0; i < 100; i++ {
rw1, rw2 := MsgPipe()
done := make(chan struct{})
go func() {
if err := SendItems(rw1, 1); err == nil {
t.Error("EncodeMsg returned nil error")
} else if err != ErrPipeClosed {
t.Errorf("EncodeMsg returned wrong error: got %v, want %v", err, ErrPipeClosed)
}
close(done)
}()

// this call should ensure that EncodeMsg is waiting to
// deliver sometimes. if this isn't done, Close is likely to
// be executed before EncodeMsg starts and then we won't test
// all the cases.
runtime.Gosched()

rw2.Close()
select {
case <-done:
case <-time.After(200 * time.Millisecond):
t.Errorf("write didn't unblock")
break loop
}
}
}

// This test should panic if concurrent close isn't implemented correctly.
func TestMsgPipeConcurrentClose(t *testing.T) {
rw1, _ := MsgPipe()
for i := 0; i < 10; i++ {
go rw1.Close()
}
}

func TestEOFSignal(t *testing.T) {
rb := make([]byte, 10)

// empty reader
eof := make(chan struct{}, 1)
sig := &eofSignal{new(bytes.Buffer), 0, eof}
if n, err := sig.Read(rb); n != 0 || err != io.EOF {
t.Errorf("Read returned unexpected values: (%v, %v)", n, err)
}
select {
case <-eof:
default:
t.Error("EOF chan not signaled")
}

// count before error
```

```
eof = make(chan struct{}, 1)
sig = &eofSignal{bytes.NewBufferString("aaaaaaaa"), 4, eof}
if n, err := sig.Read(rb); n != 4 || err != nil {
t.Errorf("Read returned unexpected values: (%v, %v)", n, err)
}
select {
case <-eof:
default:
t.Error("EOF chan not signaled")
}

// error before count
eof = make(chan struct{}, 1)
sig = &eofSignal{bytes.NewBufferString("aaaa"), 999, eof}
if n, err := sig.Read(rb); n != 4 || err != nil {
t.Errorf("Read returned unexpected values: (%v, %v)", n, err)
}
if n, err := sig.Read(rb); n != 0 || err != io.EOF {
t.Errorf("Read returned unexpected values: (%v, %v)", n, err)
}
select {
case <-eof:
default:
t.Error("EOF chan not signaled")
}

// no signal if neither occurs
eof = make(chan struct{}, 1)
sig = &eofSignal{bytes.NewBufferString("aaaaaaaaaaaaaaaaaaaaa"), 999, eof}
if n, err := sig.Read(rb); n != 10 || err != nil {
t.Errorf("Read returned unexpected values: (%v, %v)", n, err)
}
select {
case <-eof:
t.Error("unexpected EOF signal")
default:
}
}

func unhex(str string) []byte {
r := strings.NewReplacer("\t", "", " ", "", "\n", "")
b, err := hex.DecodeString(r.Replace(str))
```

```
if err != nil {
panic(fmt.Sprintf("invalid hex string: %q", str))
}
return b
}

7:F:\git\coin\ethereum\go-ethereum\p2p\metrics.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Contains the meters and timers used by the networking layer.

package p2p

import (
"net"

"github.com/ethereum/go-ethereum/metrics"
)

var (
ingressConnectMeter = metrics.NewMeter("p2p/InboundConnects")
ingressTrafficMeter = metrics.NewMeter("p2p/InboundTraffic")
egressConnectMeter  = metrics.NewMeter("p2p/OutboundConnects")
egressTrafficMeter  = metrics.NewMeter("p2p/OutboundTraffic")
)

// meteredConn is a wrapper around a network TCP connection that meters both the
// inbound and outbound network traffic.
type meteredConn struct {
*net.TCPConn // Network connection to wrap with metering
}

// newMeteredConn creates a new metered connection, also bumping the ingress or
// egress connection meter. If the metrics system is disabled, this function
// returns the original object.
func newMeteredConn(conn net.Conn, ingress bool) net.Conn {
// Short circuit if metrics are disabled
if !metrics.Enabled {
return conn
}
// Otherwise bump the connection counters and wrap the connection
if ingress {
```

```
ingressConnectMeter.Mark(1)
} else {
egressConnectMeter.Mark(1)
}
return &meteredConn{conn.(*net.TCPConn)}
}

// Read delegates a network read to the underlying connection, bumping the ingress
// traffic meter along the way.
func (c *meteredConn) Read(b []byte) (n int, err error) {
n, err = c.TCPConn.Read(b)
ingressTrafficMeter.Mark(int64(n))
return
}

// Write delegates a network write to the underlying connection, bumping the
// egress traffic meter along the way.
func (c *meteredConn) Write(b []byte) (n int, err error) {
n, err = c.TCPConn.Write(b)
egressTrafficMeter.Mark(int64(n))
return
}
```

8:F:\git\coin\ethereum\go-ethereum\p2p\nat\nat.go

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

// Package nat provides access to common network port mapping protocols.
package nat

import (
"errors"
"fmt"
"net"
"strings"
"sync"
"time"

"github.com/ethereum/go-ethereum/log"
"github.com/jackpal/go-nat-pmp"
)

// An implementation of nat.Interface can map local ports to ports
// accessible from the Internet.
type Interface interface {
// These methods manage a mapping between a port on the local
// machine to a port that can be connected to from the internet.
//
// protocol is "UDP" or "TCP". Some implementations allow setting
// a display name for the mapping. The mapping may be removed by
// the gateway when its lifetime ends.
AddMapping(protocol string, extport, intport int, name string, lifetime time.Duration) error
DeleteMapping(protocol string, extport, intport int) error

// This method should return the external (Internet-facing)
// address of the gateway device.
ExternalIP() (net.IP, error)

// Should return name of the method. This is used for logging.
String() string
}

// Parse parses a NAT interface description.
// The following formats are currently accepted.
// Note that mechanism names are not case-sensitive.
//
//     "" or "none"         return nil
//     "extip:77.12.33.4"   will assume the local machine is reachable on the given IP
//     "any"                uses the first auto-detected mechanism
//     "upnp"               uses the Universal Plug and Play protocol
//     "pmp"                uses NAT-PMP with an auto-detected gateway address
//     "pmp:192.168.0.1"    uses NAT-PMP with the given gateway address
func Parse(spec string) (Interface, error) {
var (
parts = strings.SplitN(spec, ":", 2)
mech  = strings.ToLower(parts[0])
ip    net.IP
)
if len(parts) > 1 {
ip = net.ParseIP(parts[1])
if ip == nil {
return nil, errors.New("invalid IP address")
}
}
switch mech {
```

```
case "", "none", "off":
return nil, nil
case "any", "auto", "on":
return Any(), nil
case "extip", "ip":
if ip == nil {
return nil, errors.New("missing IP address")
}
return ExtIP(ip), nil
case "upnp":
return UPnP(), nil
case "pmp", "natpmp", "nat-pmp":
return PMP(ip), nil
default:
return nil, fmt.Errorf("unknown mechanism %q", parts[0])
}
}

const (
mapTimeout        = 20 * time.Minute
mapUpdateInterval = 15 * time.Minute
)

// Map adds a port mapping on m and keeps it alive until c is closed.
// This function is typically invoked in its own goroutine.
func Map(m Interface, c chan struct{}, protocol string, extport, intport int, name string) {
log := log.New("proto", protocol, "extport", extport, "intport", intport, "interface", m)
refresh := time.NewTimer(mapUpdateInterval)
defer func() {
refresh.Stop()
log.Debug("Deleting port mapping")
m.DeleteMapping(protocol, extport, intport)
}()
if err := m.AddMapping(protocol, extport, intport, name, mapTimeout); err != nil {
log.Debug("Couldn't add port mapping", "err", err)
} else {
log.Info("Mapped network port")
}
for {
select {
case _, ok := <-c:
if !ok {
```

```
return
}
case <-refresh.C:
log.Trace("Refreshing port mapping")
if err := m.AddMapping(protocol, extport, intport, name, mapTimeout); err != nil {
log.Debug("Couldn't add port mapping", "err", err)
}
refresh.Reset(mapUpdateInterval)
}
}
}

// ExtIP assumes that the local machine is reachable on the given
// external IP address, and that any required ports were mapped manually.
// Mapping operations will not return an error but won't actually do anything.
func ExtIP(ip net.IP) Interface {
if ip == nil {
panic("IP must not be nil")
}
return extIP(ip)
}

type extIP net.IP

func (n extIP) ExternalIP() (net.IP, error) { return net.IP(n), nil }
func (n extIP) String() string              { return fmt.Sprintf("ExtIP(%v)", net.IP(n)) }

// These do nothing.
func (extIP) AddMapping(string, int, int, string, time.Duration) error { return nil }
func (extIP) DeleteMapping(string, int, int) error                     { return nil }

// Any returns a port mapper that tries to discover any supported
// mechanism on the local network.
func Any() Interface {
// TODO: attempt to discover whether the local machine has an
// Internet-class address. Return ExtIP in this case.
return startautodisc("UPnP or NAT-PMP", func() Interface {
found := make(chan Interface, 2)
go func() { found <- discoverUPnP() }()
go func() { found <- discoverPMP() }()
for i := 0; i < cap(found); i++ {
if c := <-found; c != nil {
```

```
return c
}
}
return nil
})
}

// UPnP returns a port mapper that uses UPnP. It will attempt to
// discover the address of your router using UDP broadcasts.
func UPnP() Interface {
return startautodisc("UPnP", discoverUPnP)
}

// PMP returns a port mapper that uses NAT-PMP. The provided gateway
// address should be the IP of your router. If the given gateway
// address is nil, PMP will attempt to auto-discover the router.
func PMP(gateway net.IP) Interface {
if gateway != nil {
return &pmp{gw: gateway, c: natpmp.NewClient(gateway)}
}
return startautodisc("NAT-PMP", discoverPMP)
}

// autodisc represents a port mapping mechanism that is still being
// auto-discovered. Calls to the Interface methods on this type will
// wait until the discovery is done and then call the method on the
// discovered mechanism.
//
// This type is useful because discovery can take a while but we
// want return an Interface value from UPnP, PMP and Auto immediately.
type autodisc struct {
what string // type of interface being autodiscovered
once sync.Once
doit func() Interface

mu    sync.Mutex
found Interface
}

func startautodisc(what string, doit func() Interface) Interface {
// TODO: monitor network configuration and rerun doit when it changes.
return &autodisc{what: what, doit: doit}
```

```
}

func (n *autodisc) AddMapping(protocol string, extport, intport int, name string, lifetime
time.Duration) error {
if err := n.wait(); err != nil {
return err
}
return n.found.AddMapping(protocol, extport, intport, name, lifetime)
}

func (n *autodisc) DeleteMapping(protocol string, extport, intport int) error {
if err := n.wait(); err != nil {
return err
}
return n.found.DeleteMapping(protocol, extport, intport)
}

func (n *autodisc) ExternalIP() (net.IP, error) {
if err := n.wait(); err != nil {
return nil, err
}
return n.found.ExternalIP()
}

func (n *autodisc) String() string {
n.mu.Lock()
defer n.mu.Unlock()
if n.found == nil {
return n.what
} else {
return n.found.String()
}
}

// wait blocks until auto-discovery has been performed.
func (n *autodisc) wait() error {
n.once.Do(func() {
n.mu.Lock()
n.found = n.doit()
n.mu.Unlock()
})
if n.found == nil {
```

```
return fmt.Errorf("no %s router discovered", n.what)
}
return nil
}
```

9:F:\git\coin\ethereum\go-ethereum\p2p\nat\natpmp.go

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

package nat

import (
"fmt"
"net"
"strings"
"time"

"github.com/jackpal/go-nat-pmp"
)

// natPMPClient adapts the NAT-PMP protocol implementation so it conforms to
// the common interface.
type pmp struct {
gw net.IP
c  *natpmp.Client
}

func (n *pmp) String() string {
return fmt.Sprintf("NAT-PMP(%v)", n.gw)
}

func (n *pmp) ExternalIP() (net.IP, error) {
response, err := n.c.GetExternalAddress()
if err != nil {
return nil, err
}
return response.ExternalIPAddress[:], nil
}

func (n *pmp) AddMapping(protocol string, extport, intport int, name string, lifetime time.Duration)
error {
if lifetime <= 0 {
return fmt.Errorf("lifetime must not be <= 0")
```

```
}
// Note order of port arguments is switched between our
// AddMapping and the client's AddPortMapping.
_, err := n.c.AddPortMapping(strings.ToLower(protocol), intport, extport, int(lifetime/time.Second))
return err
}

func (n *pmp) DeleteMapping(protocol string, extport, intport int) (err error) {
// To destroy a mapping, send an add-port with an internalPort of
// the internal port to destroy, an external port of zero and a
// time of zero.
_, err = n.c.AddPortMapping(strings.ToLower(protocol), intport, 0, 0)
return err
}

func discoverPMP() Interface {
// run external address lookups on all potential gateways
gws := potentialGateways()
found := make(chan *pmp, len(gws))
for i := range gws {
gw := gws[i]
go func() {
c := natpmp.NewClient(gw)
if _, err := c.GetExternalAddress(); err != nil {
found <- nil
} else {
found <- &pmp{gw, c}
}
}()
}
// return the one that responds first.
// discovery needs to be quick, so we stop caring about
// any responses after a very short timeout.
timeout := time.NewTimer(1 * time.Second)
defer timeout.Stop()
for range gws {
select {
case c := <-found:
if c != nil {
return c
}
case <-timeout.C:
```

```
return nil
}
}
return nil
}

var (
// LAN IP ranges
_, lan10, _  = net.ParseCIDR("10.0.0.0/8")
_, lan176, _ = net.ParseCIDR("172.16.0.0/12")
_, lan192, _ = net.ParseCIDR("192.168.0.0/16")
)

// TODO: improve this. We currently assume that (on most networks)
// the router is X.X.X.1 in a local LAN range.
func potentialGateways() (gws []net.IP) {
ifaces, err := net.Interfaces()
if err != nil {
return nil
}
for _, iface := range ifaces {
ifaddrs, err := iface.Addrs()
if err != nil {
return gws
}
for _, addr := range ifaddrs {
switch x := addr.(type) {
case *net.IPNet:
if lan10.Contains(x.IP) || lan176.Contains(x.IP) || lan192.Contains(x.IP) {
ip := x.IP.Mask(x.Mask).To4()
if ip != nil {
ip[3] = ip[3] | 0x01
gws = append(gws, ip)
}
}
}
}
}
return gws
}
```

10:F:\git\coin\ethereum\go-ethereum\p2p\nat\natupnp.go

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

package nat

import (
"errors"
"fmt"
"net"
"strings"
"time"

"github.com/huin/goupnp"
"github.com/huin/goupnp/dcps/internetgateway1"
"github.com/huin/goupnp/dcps/internetgateway2"
)

const soapRequestTimeout = 3 * time.Second

type upnp struct {
dev     *goupnp.RootDevice
service string
client  upnpClient
}

type upnpClient interface {
GetExternalIPAddress() (string, error)
AddPortMapping(string, uint16, string, uint16, string, bool, string, uint32) error
DeletePortMapping(string, uint16, string) error
GetNATRSIPStatus() (sip bool, nat bool, err error)
}

func (n *upnp) ExternalIP() (addr net.IP, err error) {
ipString, err := n.client.GetExternalIPAddress()
if err != nil {
return nil, err
}
ip := net.ParseIP(ipString)
if ip == nil {
return nil, errors.New("bad IP in response")
}
return ip, nil
}
```

```
func (n *upnp) AddMapping(protocol string, extport, intport int, desc string, lifetime time.Duration)
error {
ip, err := n.internalAddress()
if err != nil {
return nil
}
protocol = strings.ToUpper(protocol)
lifetimeS := uint32(lifetime / time.Second)
return n.client.AddPortMapping("", uint16(extport), protocol, uint16(intport), ip.String(), true, desc,
lifetimeS)
}

func (n *upnp) internalAddress() (net.IP, error) {
devaddr, err := net.ResolveUDPAddr("udp4", n.dev.URLBase.Host)
if err != nil {
return nil, err
}
ifaces, err := net.Interfaces()
if err != nil {
return nil, err
}
for _, iface := range ifaces {
addrs, err := iface.Addrs()
if err != nil {
return nil, err
}
for _, addr := range addrs {
switch x := addr.(type) {
case *net.IPNet:
if x.Contains(devaddr.IP) {
return x.IP, nil
}
}
}
}
return nil, fmt.Errorf("could not find local address in same net as %v", devaddr)
}

func (n *upnp) DeleteMapping(protocol string, extport, intport int) error {
return n.client.DeletePortMapping("", uint16(extport), strings.ToUpper(protocol))
}
```

```
func (n *upnp) String() string {
return "UPNP " + n.service
}

// discoverUPnP searches for Internet Gateway Devices
// and returns the first one it can find on the local network.
func discoverUPnP() Interface {
found := make(chan *upnp, 2)
// IGDv1
go discover(found, internetgateway1.URN_WANConnectionDevice_1, func(dev
*goupnp.RootDevice, sc goupnp.ServiceClient) *upnp {
switch sc.Service.ServiceType {
case internetgateway1.URN_WANIPConnection_1:
return &upnp{dev, "IGDv1-IP1", &internetgateway1.WANIPConnection1{ServiceClient: sc}}
case internetgateway1.URN_WANPPPConnection_1:
return &upnp{dev, "IGDv1-PPP1", &internetgateway1.WANPPPConnection1{ServiceClient: sc}}
}
return nil
})
// IGDv2
go discover(found, internetgateway2.URN_WANConnectionDevice_2, func(dev
*goupnp.RootDevice, sc goupnp.ServiceClient) *upnp {
switch sc.Service.ServiceType {
case internetgateway2.URN_WANIPConnection_1:
return &upnp{dev, "IGDv2-IP1", &internetgateway2.WANIPConnection1{ServiceClient: sc}}
case internetgateway2.URN_WANIPConnection_2:
return &upnp{dev, "IGDv2-IP2", &internetgateway2.WANIPConnection2{ServiceClient: sc}}
case internetgateway2.URN_WANPPPConnection_1:
return &upnp{dev, "IGDv2-PPP1", &internetgateway2.WANPPPConnection1{ServiceClient: sc}}
}
return nil
})
for i := 0; i < cap(found); i++ {
if c := <-found; c != nil {
return c
}
}
return nil
}

// finds devices matching the given target and calls matcher for all
// advertised services of each device. The first non-nil service found
// is sent into out. If no service matched, nil is sent.
func discover(out chan<- *upnp, target string, matcher func(*goupnp.RootDevice,
goupnp.ServiceClient) *upnp) {
devs, err := goupnp.DiscoverDevices(target)
if err != nil {
out <- nil
return
}
found := false
for i := 0; i < len(devs) && !found; i++ {
if devs[i].Root == nil {
continue
}
devs[i].Root.Device.VisitServices(func(service *goupnp.Service) {
if found {
return
}
// check for a matching IGD service
sc := goupnp.ServiceClient{
SOAPClient: service.NewSOAPClient(),
RootDevice: devs[i].Root,
Location:   devs[i].Location,
Service:    service,
}
sc.SOAPClient.HTTPClient.Timeout = soapRequestTimeout
upnp := matcher(devs[i].Root, sc)
if upnp == nil {
return
}
// check whether port mapping is enabled
if _, nat, err := upnp.client.GetNATRSIPStatus(); err != nil || !nat {
return
}
out <- upnp
found = true
})
}
if !found {
out <- nil
}
}
```

11:F:\git\coin\ethereum\go-ethereum\p2p\nat\natupnp_test.go

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

package nat

import (
"fmt"
"io"
"net"
"net/http"
"runtime"
"strings"
"testing"

"github.com/huin/goupnp/httpu"
)

func TestUPNP_DDWRT(t *testing.T) {
if runtime.GOOS == "windows" {
t.Skipf("disabled to avoid firewall prompt")
}

dev := &fakeIGD{
t: t,
ssdpResp: "HTTP/1.1 200 OK\r\n" +
"Cache-Control: max-age=300\r\n" +
"Date: Sun, 10 May 2015 10:05:33 GMT\r\n" +
"Ext: \r\n" +
"Location: http://{{listenAddr}}/InternetGatewayDevice.xml\r\n" +
"Server: POSIX UPnP/1.0 DD-WRT Linux/V24\r\n" +
"ST: urn:schemas-upnp-org:device:WANConnectionDevice:1\r\n" +
"USN: uuid:CB2471CC-CF2E-9795-8D9C-E87B34C16800::urn:schemas-upnp-
org:device:WANConnectionDevice:1\r\n" +
"\r\n",
httpResps: map[string]string{
"GET /InternetGatewayDevice.xml": `
 <?xml version="1.0"?>
 <root xmlns="urn:schemas-upnp-org:device-1-0">
 <specVersion>
 <major>1</major>
 <minor>0</minor>
```

```
</specVersion>
<device>
<deviceType>urn:schemas-upnp-org:device:InternetGatewayDevice:1</deviceType>
<manufacturer>DD-WRT</manufacturer>
<manufacturerURL>http://www.dd-wrt.com</manufacturerURL>
<modelDescription>Gateway</modelDescription>
<friendlyName>Asus RT-N16:DD-WRT</friendlyName>
<modelName>Asus RT-N16</modelName>
<modelNumber>V24</modelNumber>
<serialNumber>0000001</serialNumber>
<modelURL>http://www.dd-wrt.com</modelURL>
<UDN>uuid:A13AB4C3-3A14-E386-DE6A-EFEA923A06FE</UDN>
<serviceList>
<service>
<serviceType>urn:schemas-upnp-org:service:Layer3Forwarding:1</serviceType>
<serviceId>urn:upnp-org:serviceId:L3Forwarding1</serviceId>
<SCPDURL>/x_layer3forwarding.xml</SCPDURL>
<controlURL>/control?Layer3Forwarding</controlURL>
<eventSubURL>/event?Layer3Forwarding</eventSubURL>
</service>
</serviceList>
<deviceList>
<device>
<deviceType>urn:schemas-upnp-org:device:WANDevice:1</deviceType>
<friendlyName>WANDevice</friendlyName>
<manufacturer>DD-WRT</manufacturer>
<manufacturerURL>http://www.dd-wrt.com</manufacturerURL>
<modelDescription>Gateway</modelDescription>
<modelName>router</modelName>
<modelURL>http://www.dd-wrt.com</modelURL>
<UDN>uuid:48FD569B-F9A9-96AE-4EE6-EB403D3DB91A</UDN>
<serviceList>
<service>
<serviceType>urn:schemas-upnp-org:service:WANCommonInterfaceConfig:1</serviceType>
<serviceId>urn:upnp-org:serviceId:WANCommonIFC1</serviceId>
<SCPDURL>/x_wancommoninterfaceconfig.xml</SCPDURL>
<controlURL>/control?WANCommonInterfaceConfig</controlURL>
<eventSubURL>/event?WANCommonInterfaceConfig</eventSubURL>
</service>
</serviceList>
<deviceList>
<device>
```

```xml
<deviceType>urn:schemas-upnp-org:device:WANConnectionDevice:1</deviceType>
<friendlyName>WAN Connection Device</friendlyName>
<manufacturer>DD-WRT</manufacturer>
<manufacturerURL>http://www.dd-wrt.com</manufacturerURL>
<modelDescription>Gateway</modelDescription>
<modelName>router</modelName>
<modelURL>http://www.dd-wrt.com</modelURL>
<UDN>uuid:CB2471CC-CF2E-9795-8D9C-E87B34C16800</UDN>
<serviceList>
<service>
<serviceType>urn:schemas-upnp-org:service:WANIPConnection:1</serviceType>
<serviceId>urn:upnp-org:serviceId:WANIPConn1</serviceId>
<SCPDURL>/x_wanipconnection.xml</SCPDURL>
<controlURL>/control?WANIPConnection</controlURL>
<eventSubURL>/event?WANIPConnection</eventSubURL>
</service>
</serviceList>
</device>
</deviceList>
</device>
<device>
<deviceType>urn:schemas-upnp-org:device:LANDevice:1</deviceType>
<friendlyName>LANDevice</friendlyName>
<manufacturer>DD-WRT</manufacturer>
<manufacturerURL>http://www.dd-wrt.com</manufacturerURL>
<modelDescription>Gateway</modelDescription>
<modelName>router</modelName>
<modelURL>http://www.dd-wrt.com</modelURL>
<UDN>uuid:04021998-3B35-2BDB-7B3C-99DA4435DA09</UDN>
<serviceList>
<service>
<serviceType>urn:schemas-upnp-org:service:LANHostConfigManagement:1</serviceType>
<serviceId>urn:upnp-org:serviceId:LANHostCfg1</serviceId>
<SCPDURL>/x_lanhostconfigmanagement.xml</SCPDURL>
<controlURL>/control?LANHostConfigManagement</controlURL>
<eventSubURL>/event?LANHostConfigManagement</eventSubURL>
</service>
</serviceList>
</device>
</deviceList>
<presentationURL>http://{{listenAddr}}</presentationURL>
</device>
```

```
 </root>
`,
// The response to our GetNATRSIPStatus call. This
// particular implementation has a bug where the elements
// inside u:GetNATRSIPStatusResponse are not properly
// namespaced.
"POST /control?WANIPConnection": `
 <s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <s:Body>
 <u:GetNATRSIPStatusResponse xmlns:u="urn:schemas-upnp-org:service:WANIPConnection:1">
 <NewRSIPAvailable>0</NewRSIPAvailable>
 <NewNATEnabled>1</NewNATEnabled>
 </u:GetNATRSIPStatusResponse>
 </s:Body>
 </s:Envelope>
`,
},
}
if err := dev.listen(); err != nil {
t.Skipf("cannot listen: %v", err)
}
dev.serve()
defer dev.close()

// Attempt to discover the fake device.
discovered := discoverUPnP()
if discovered == nil {
t.Fatalf("not discovered")
}
upnp, _ := discovered.(*upnp)
if upnp.service != "IGDv1-IP1" {
t.Errorf("upnp.service mismatch: got %q, want %q", upnp.service, "IGDv1-IP1")
}
wantURL := "http://" + dev.listener.Addr().String() + "/InternetGatewayDevice.xml"
if upnp.dev.URLBaseStr != wantURL {
t.Errorf("upnp.dev.URLBaseStr mismatch: got %q, want %q", upnp.dev.URLBaseStr, wantURL)
}
}

// fakeIGD presents itself as a discoverable UPnP device which sends
// canned responses to HTTPU and HTTP requests.
```

```
type fakeIGD struct {
t *testing.T // for logging

listener      net.Listener
mcastListener *net.UDPConn

// This should be a complete HTTP response (including headers).
// It is sent as the response to any sspd packet. Any occurrence
// of "{{listenAddr}}" is replaced with the actual TCP listen
// address of the HTTP server.
ssdpResp string
// This one should contain XML payloads for all requests
// performed. The keys contain method and path, e.g. "GET /foo/bar".
// As with ssdpResp, "{{listenAddr}}" is replaced with the TCP
// listen address.
httpResps map[string]string
}

// httpu.Handler
func (dev *fakeIGD) ServeMessage(r *http.Request) {
dev.t.Logf(`HTTPU request %s %s`, r.Method, r.RequestURI)
conn, err := net.Dial("udp4", r.RemoteAddr)
if err != nil {
fmt.Printf("reply Dial error: %v", err)
return
}
defer conn.Close()
io.WriteString(conn, dev.replaceListenAddr(dev.ssdpResp))
}

// http.Handler
func (dev *fakeIGD) ServeHTTP(w http.ResponseWriter, r *http.Request) {
if resp, ok := dev.httpResps[r.Method+" "+r.RequestURI]; ok {
dev.t.Logf(`HTTP request "%s %s" --> %d`, r.Method, r.RequestURI, 200)
io.WriteString(w, dev.replaceListenAddr(resp))
} else {
dev.t.Logf(`HTTP request "%s %s" --> %d`, r.Method, r.RequestURI, 404)
w.WriteHeader(http.StatusNotFound)
}
}

func (dev *fakeIGD) replaceListenAddr(resp string) string {
```

```
return strings.Replace(resp, "{{listenAddr}}", dev.listener.Addr().String(), -1)
}

func (dev *fakeIGD) listen() (err error) {
if dev.listener, err = net.Listen("tcp", "127.0.0.1:0"); err != nil {
return err
}
laddr := &net.UDPAddr{IP: net.ParseIP("239.255.255.250"), Port: 1900}
if dev.mcastListener, err = net.ListenMulticastUDP("udp", nil, laddr); err != nil {
dev.listener.Close()
return err
}
return nil
}

func (dev *fakeIGD) serve() {
go httpu.Serve(dev.mcastListener, dev)
go http.Serve(dev.listener, dev)
}

func (dev *fakeIGD) close() {
dev.mcastListener.Close()
dev.listener.Close()
}

12:F:\git\coin\ethereum\go-ethereum\p2p\nat\nat_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package nat

import (
"net"
"testing"
"time"
)

// This test checks that autodisc doesn't hang and returns
// consistent results when multiple goroutines call its methods
// concurrently.
func TestAutoDiscRace(t *testing.T) {
ad := startautodisc("thing", func() Interface {
time.Sleep(500 * time.Millisecond)
return extIP{33, 44, 55, 66}
})

// Spawn a few concurrent calls to ad.ExternalIP.
type rval struct {
ip  net.IP
err error
}
results := make(chan rval, 50)
for i := 0; i < cap(results); i++ {
go func() {
ip, err := ad.ExternalIP()
results <- rval{ip, err}
}()
}

// Check that they all return the correct result within the deadline.
deadline := time.After(2 * time.Second)
for i := 0; i < cap(results); i++ {
select {
case <-deadline:
t.Fatal("deadline exceeded")
case rval := <-results:
if rval.err != nil {
t.Errorf("result %d: unexpected error: %v", i, rval.err)
}
wantIP := net.IP{33, 44, 55, 66}
if !rval.ip.Equal(wantIP) {
t.Errorf("result %d: got IP %v, want %v", i, rval.ip, wantIP)
}
}
}
}
```

13:F:\git\coin\ethereum\go-ethereum\p2p\netutil\error.go

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

package netutil

// IsTemporaryError checks whether the given error should be considered temporary.
func IsTemporaryError(err error) bool {
tempErr, ok := err.(interface {
```

```
Temporary() bool
})
return ok && tempErr.Temporary() || isPacketTooBig(err)
}

14:F:\git\coin\ethereum\go-ethereum\p2p\netutil\error_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package netutil

import (
"net"
"testing"
"time"
)

// This test checks that isPacketTooBig correctly identifies
// errors that result from receiving a UDP packet larger
// than the supplied receive buffer.
func TestIsPacketTooBig(t *testing.T) {
listener, err := net.ListenPacket("udp", "127.0.0.1:0")
if err != nil {
t.Fatal(err)
}
defer listener.Close()
sender, err := net.Dial("udp", listener.LocalAddr().String())
if err != nil {
t.Fatal(err)
}
defer sender.Close()

sendN := 1800
recvN := 300
for i := 0; i < 20; i++ {
go func() {
buf := make([]byte, sendN)
for i := range buf {
buf[i] = byte(i)
}
sender.Write(buf)
}()
```

```
buf := make([]byte, recvN)
listener.SetDeadline(time.Now().Add(1 * time.Second))
n, _, err := listener.ReadFrom(buf)
if err != nil {
if nerr, ok := err.(net.Error); ok && nerr.Timeout() {
continue
}
if !isPacketTooBig(err) {
t.Fatalf("unexpected read error: %v", err)
}
continue
}
if n != recvN {
t.Fatalf("short read: %d, want %d", n, recvN)
}
for i := range buf {
if buf[i] != byte(i) {
t.Fatalf("error in pattern")
break
}
}
}
}
```

15:F:\git\coin\ethereum\go-ethereum\p2p\netutil\net.go

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

// Package netutil contains extensions to the net package.
package netutil

import (
"errors"
"net"
"strings"
)

var lan4, lan6, special4, special6 Netlist

func init() {
// Lists from RFC 5735, RFC 5156,
// https://www.iana.org/assignments/iana-ipv4-special-registry/
lan4.Add("0.0.0.0/8")              // "This" network
```

```
lan4.Add("10.0.0.0/8")             // Private Use
lan4.Add("172.16.0.0/12")          // Private Use
lan4.Add("192.168.0.0/16")         // Private Use
lan6.Add("fe80::/10")              // Link-Local
lan6.Add("fc00::/7")               // Unique-Local
special4.Add("192.0.0.0/29")       // IPv4 Service Continuity
special4.Add("192.0.0.9/32")       // PCP Anycast
special4.Add("192.0.0.170/32")     // NAT64/DNS64 Discovery
special4.Add("192.0.0.171/32")     // NAT64/DNS64 Discovery
special4.Add("192.0.2.0/24")       // TEST-NET-1
special4.Add("192.31.196.0/24")    // AS112
special4.Add("192.52.193.0/24")    // AMT
special4.Add("192.88.99.0/24")     // 6to4 Relay Anycast
special4.Add("192.175.48.0/24")    // AS112
special4.Add("198.18.0.0/15")      // Device Benchmark Testing
special4.Add("198.51.100.0/24")    // TEST-NET-2
special4.Add("203.0.113.0/24")     // TEST-NET-3
special4.Add("255.255.255.255/32") // Limited Broadcast

// http://www.iana.org/assignments/iana-ipv6-special-registry/
special6.Add("100::/64")
special6.Add("2001::/32")
special6.Add("2001:1::1/128")
special6.Add("2001:2::/48")
special6.Add("2001:3::/32")
special6.Add("2001:4:112::/48")
special6.Add("2001:5::/32")
special6.Add("2001:10::/28")
special6.Add("2001:20::/28")
special6.Add("2001:db8::/32")
special6.Add("2002::/16")
}

// Netlist is a list of IP networks.
type Netlist []net.IPNet

// ParseNetlist parses a comma-separated list of CIDR masks.
// Whitespace and extra commas are ignored.
func ParseNetlist(s string) (*Netlist, error) {
ws := strings.NewReplacer(" ", "", "\n", "", "\t", "")
masks := strings.Split(ws.Replace(s), ",")
l := make(Netlist, 0)
```

```
for _, mask := range masks {
if mask == "" {
continue
}
_, n, err := net.ParseCIDR(mask)
if err != nil {
return nil, err
}
l = append(l, *n)
}
return &l, nil
}

// MarshalTOML implements toml.MarshalerRec.
func (l Netlist) MarshalTOML() interface{} {
list := make([]string, 0, len(l))
for _, net := range l {
list = append(list, net.String())
}
return list
}

// UnmarshalTOML implements toml.UnmarshalerRec.
func (l *Netlist) UnmarshalTOML(fn func(interface{}) error) error {
var masks []string
if err := fn(&masks); err != nil {
return err
}
for _, mask := range masks {
_, n, err := net.ParseCIDR(mask)
if err != nil {
return err
}
*l = append(*l, *n)
}
return nil
}

// Add parses a CIDR mask and appends it to the list. It panics for invalid masks and is
// intended to be used for setting up static lists.
func (l *Netlist) Add(cidr string) {
_, n, err := net.ParseCIDR(cidr)
```

```
if err != nil {
panic(err)
}
*l = append(*l, *n)
}

// Contains reports whether the given IP is contained in the list.
func (l *Netlist) Contains(ip net.IP) bool {
if l == nil {
return false
}
for _, net := range *l {
if net.Contains(ip) {
return true
}
}
return false
}

// IsLAN reports whether an IP is a local network address.
func IsLAN(ip net.IP) bool {
if ip.IsLoopback() {
return true
}
if v4 := ip.To4(); v4 != nil {
return lan4.Contains(v4)
}
return lan6.Contains(ip)
}

// IsSpecialNetwork reports whether an IP is located in a special-use network range
// This includes broadcast, multicast and documentation addresses.
func IsSpecialNetwork(ip net.IP) bool {
if ip.IsMulticast() {
return true
}
if v4 := ip.To4(); v4 != nil {
return special4.Contains(v4)
}
return special6.Contains(ip)
}
```

```
var (
errInvalid     = errors.New("invalid IP")
errUnspecified = errors.New("zero address")
errSpecial     = errors.New("special network")
errLoopback    = errors.New("loopback address from non-loopback host")
errLAN         = errors.New("LAN address from WAN host")
)

// CheckRelayIP reports whether an IP relayed from the given sender IP
// is a valid connection target.
//
// There are four rules:
//   - Special network addresses are never valid.
//   - Loopback addresses are OK if relayed by a loopback host.
//   - LAN addresses are OK if relayed by a LAN host.
//   - All other addresses are always acceptable.
func CheckRelayIP(sender, addr net.IP) error {
if len(addr) != net.IPv4len && len(addr) != net.IPv6len {
return errInvalid
}
if addr.IsUnspecified() {
return errUnspecified
}
if IsSpecialNetwork(addr) {
return errSpecial
}
if addr.IsLoopback() && !sender.IsLoopback() {
return errLoopback
}
if IsLAN(addr) && !IsLAN(sender) {
return errLAN
}
return nil
}

16:F:\git\coin\ethereum\go-ethereum\p2p\netutil\net_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package netutil

import (
"net"
```

```
"reflect"
"testing"

"github.com/davecgh/go-spew/spew"
)

func TestParseNetlist(t *testing.T) {
var tests = []struct {
input    string
wantErr  error
wantList *Netlist
}{
{
input:    "",
wantList: &Netlist{},
},
{
input:    "127.0.0.0/8",
wantErr:  nil,
wantList: &Netlist{{IP: net.IP{127, 0, 0, 0}, Mask: net.CIDRMask(8, 32)}},
},
{
input:   "127.0.0.0/44",
wantErr: &net.ParseError{Type: "CIDR address", Text: "127.0.0.0/44"},
},
{
input: "127.0.0.0/16, 23.23.23.23/24,",
wantList: &Netlist{
{IP: net.IP{127, 0, 0, 0}, Mask: net.CIDRMask(16, 32)},
{IP: net.IP{23, 23, 23, 0}, Mask: net.CIDRMask(24, 32)},
},
},
}

for _, test := range tests {
l, err := ParseNetlist(test.input)
if !reflect.DeepEqual(err, test.wantErr) {
t.Errorf("%q: got error %q, want %q", test.input, err, test.wantErr)
continue
}
if !reflect.DeepEqual(l, test.wantList) {
spew.Dump(l)
spew.Dump(test.wantList)
t.Errorf("%q: got %v, want %v", test.input, l, test.wantList)
}
}
}

func TestNilNetListContains(t *testing.T) {
var list *Netlist
checkContains(t, list.Contains, nil, []string{"1.2.3.4"})
}

func TestIsLAN(t *testing.T) {
checkContains(t, IsLAN,
[]string{ // included
"0.0.0.0",
"0.2.0.8",
"127.0.0.1",
"10.0.1.1",
"10.22.0.3",
"172.31.252.251",
"192.168.1.4",
"fe80::f4a1:8eff:fec5:9d9d",
"febf::ab32:2233",
"fc00::4",
},
[]string{ // excluded
"192.0.2.1",
"1.0.0.0",
"172.32.0.1",
"fec0::2233",
},
)
}

func TestIsSpecialNetwork(t *testing.T) {
checkContains(t, IsSpecialNetwork,
[]string{ // included
"192.0.2.1",
"192.0.2.44",
"2001:db8:85a3:8d3:1319:8a2e:370:7348",
"255.255.255.255",
"224.0.0.22", // IPv4 multicast
```

```
"ff05::1:3",  // IPv6 multicast
},
[]string{ // excluded
"192.0.3.1",
"1.0.0.0",
"172.32.0.1",
"fec0::2233",
},
)
}

func checkContains(t *testing.T, fn func(net.IP) bool, inc, exc []string) {
for _, s := range inc {
if !fn(parseIP(s)) {
t.Error("returned false for included address", s)
}
}
for _, s := range exc {
if fn(parseIP(s)) {
t.Error("returned true for excluded address", s)
}
}
}

func parseIP(s string) net.IP {
ip := net.ParseIP(s)
if ip == nil {
panic("invalid " + s)
}
return ip
}

func TestCheckRelayIP(t *testing.T) {
tests := []struct {
sender, addr string
want         error
}{
{"127.0.0.1", "0.0.0.0", errUnspecified},
{"192.168.0.1", "0.0.0.0", errUnspecified},
{"23.55.1.242", "0.0.0.0", errUnspecified},
{"127.0.0.1", "255.255.255.255", errSpecial},
{"192.168.0.1", "255.255.255.255", errSpecial},
{"23.55.1.242", "255.255.255.255", errSpecial},
{"192.168.0.1", "127.0.2.19", errLoopback},
{"23.55.1.242", "192.168.0.1", errLAN},

{"127.0.0.1", "127.0.2.19", nil},
{"127.0.0.1", "192.168.0.1", nil},
{"127.0.0.1", "23.55.1.242", nil},
{"192.168.0.1", "192.168.0.1", nil},
{"192.168.0.1", "23.55.1.242", nil},
{"23.55.1.242", "23.55.1.242", nil},
}

for _, test := range tests {
err := CheckRelayIP(parseIP(test.sender), parseIP(test.addr))
if err != test.want {
t.Errorf("%s from %s: got %q, want %q", test.addr, test.sender, err, test.want)
}
}
}

func BenchmarkCheckRelayIP(b *testing.B) {
sender := parseIP("23.55.1.242")
addr := parseIP("23.55.1.2")
for i := 0; i < b.N; i++ {
CheckRelayIP(sender, addr)
}
}
```

17:F:\git\coin\ethereum\go-ethereum\p2p\netutil\toobig_notwindows.go

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

//+build !windows

package netutil

// isPacketTooBig reports whether err indicates that a UDP packet didn't
// fit the receive buffer. There is no such error on
// non-Windows platforms.
func isPacketTooBig(err error) bool {
return false
}
```

18:F:\git\coin\ethereum\go-ethereum\p2p\netutil\toobig_windows.go

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

//+build windows

package netutil

import (
"net"
"os"
"syscall"
)

const _WSAEMSGSIZE = syscall.Errno(10040)

// isPacketTooBig reports whether err indicates that a UDP packet didn't
// fit the receive buffer. On Windows, WSARecvFrom returns
// code WSAEMSGSIZE and no data if this happens.
func isPacketTooBig(err error) bool {
if opErr, ok := err.(*net.OpError); ok {
if scErr, ok := opErr.Err.(*os.SyscallError); ok {
return scErr.Err == _WSAEMSGSIZE
}
return opErr.Err == _WSAEMSGSIZE
}
return false
}
```

19:F:\git\coin\ethereum\go-ethereum\p2p\peer.go

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

package p2p

import (
"fmt"
"io"
"net"
"sort"
"sync"
"time"

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

```
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/p2p/discover"
"github.com/ethereum/go-ethereum/rlp"
)

const (
baseProtocolVersion    = 4
baseProtocolLength     = uint64(16)
baseProtocolMaxMsgSize = 2 * 1024

pingInterval = 15 * time.Second
)

const (
// devp2p message codes
handshakeMsg = 0x00
discMsg      = 0x01
pingMsg      = 0x02
pongMsg      = 0x03
getPeersMsg  = 0x04
peersMsg     = 0x05
)

// protoHandshake is the RLP structure of the protocol handshake.
type protoHandshake struct {
Version    uint64
Name       string
Caps       []Cap
ListenPort uint64
ID         discover.NodeID

// Ignore additional fields (for forward compatibility).
Rest []rlp.RawValue `rlp:"tail"`
}

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

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

// NewPeer returns a peer for testing purposes.
func NewPeer(id discover.NodeID, name string, caps []Cap) *Peer {
pipe, _ := net.Pipe()
conn := &conn{fd: pipe, transport: nil, id: id, caps: caps, name: name}
peer := newPeer(conn, nil)
close(peer.closed) // ensures Disconnect doesn't block
return peer
}

// ID returns the node's public key.
func (p *Peer) ID() discover.NodeID {
return p.rw.id
}

// Name returns the node name that the remote node advertised.
func (p *Peer) Name() string {
return p.rw.name
}

// Caps returns the capabilities (supported subprotocols) of the remote peer.
func (p *Peer) Caps() []Cap {
// TODO: maybe return copy
return p.rw.caps
}

// RemoteAddr returns the remote address of the network connection.
func (p *Peer) RemoteAddr() net.Addr {
return p.rw.fd.RemoteAddr()
}

// LocalAddr returns the local address of the network connection.
func (p *Peer) LocalAddr() net.Addr {
return p.rw.fd.LocalAddr()
}

// Disconnect terminates the peer connection with the given reason.
// It returns immediately and does not wait until the connection is closed.
func (p *Peer) Disconnect(reason DiscReason) {
select {
case p.disc <- reason:
case <-p.closed:
}
}

// String implements fmt.Stringer.
func (p *Peer) String() string {
return fmt.Sprintf("Peer %x %v", p.rw.id[:8], p.RemoteAddr())
}

func newPeer(conn *conn, protocols []Protocol) *Peer {
protomap := matchProtocols(protocols, conn.caps, conn)
p := &Peer{
rw:       conn,
running:  protomap,
created:  mclock.Now(),
disc:     make(chan DiscReason),
protoErr: make(chan error, len(protomap)+1), // protocols + pingLoop
closed:   make(chan struct{}),
log:      log.New("id", conn.id, "conn", conn.flags),
}
return p
}

func (p *Peer) Log() log.Logger {
return p.log
}

func (p *Peer) run() (remoteRequested bool, err error) {
var (
writeStart = make(chan struct{}, 1)
writeErr   = make(chan error, 1)
readErr    = make(chan error, 1)
reason     DiscReason // sent to the peer
)
p.wg.Add(2)
go p.readLoop(readErr)
go p.pingLoop()
```

```
// Start all protocol handlers.
writeStart <- struct{}{}
p.startProtocols(writeStart, writeErr)

// Wait for an error or disconnect.
loop:
for {
select {
case err = <-writeErr:
// A write finished. Allow the next write to start if
// there was no error.
if err != nil {
reason = DiscNetworkError
break loop
}
writeStart <- struct{}{}
case err = <-readErr:
if r, ok := err.(DiscReason); ok {
remoteRequested = true
reason = r
} else {
reason = DiscNetworkError
}
break loop
case err = <-p.protoErr:
reason = discReasonForError(err)
break loop
case err = <-p.disc:
break loop
}
}

close(p.closed)
p.rw.close(reason)
p.wg.Wait()
return remoteRequested, err
}

func (p *Peer) pingLoop() {
ping := time.NewTicker(pingInterval)
defer p.wg.Done()
defer ping.Stop()
```

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

func (p *Peer) readLoop(errc chan<- error) {
defer p.wg.Done()
for {
msg, err := p.rw.ReadMsg()
if err != nil {
errc <- err
return
}
msg.ReceivedAt = time.Now()
if err = p.handle(msg); err != nil {
errc <- err
return
}
}
}

func (p *Peer) handle(msg Msg) error {
switch {
case msg.Code == pingMsg:
msg.Discard()
go SendItems(p.rw, pongMsg)
case msg.Code == discMsg:
var reason [1]DiscReason
// This is the last message. We don't need to discard or
// check errors because, the connection will be closed after it.
rlp.Decode(msg.Payload, &reason)
return reason[0]
case msg.Code < baseProtocolLength:
// ignore other base protocol messages
```

```
return msg.Discard()
default:
// it's a subprotocol message
proto, err := p.getProto(msg.Code)
if err != nil {
return fmt.Errorf("msg code out of range: %v", msg.Code)
}
select {
case proto.in <- msg:
return nil
case <-p.closed:
return io.EOF
}
}
return nil
}

func countMatchingProtocols(protocols []Protocol, caps []Cap) int {
n := 0
for _, cap := range caps {
for _, proto := range protocols {
if proto.Name == cap.Name && proto.Version == cap.Version {
n++
}
}
}
return n
}

// matchProtocols creates structures for matching named subprotocols.
func matchProtocols(protocols []Protocol, caps []Cap, rw MsgReadWriter) map[string]*protoRW {
sort.Sort(capsByNameAndVersion(caps))
offset := baseProtocolLength
result := make(map[string]*protoRW)

outer:
for _, cap := range caps {
for _, proto := range protocols {
if proto.Name == cap.Name && proto.Version == cap.Version {
// If an old protocol version matched, revert it
if old := result[cap.Name]; old != nil {
offset -= old.Length
}
// Assign the new match
result[cap.Name] = &protoRW{Protocol: proto, offset: offset, in: make(chan Msg), w: rw}
offset += proto.Length

continue outer
}
}
}
return result
}

func (p *Peer) startProtocols(writeStart <-chan struct{}, writeErr chan<- error) {
p.wg.Add(len(p.running))
for _, proto := range p.running {
proto := proto
proto.closed = p.closed
proto.wstart = writeStart
proto.werr = writeErr
p.log.Trace(fmt.Sprintf("Starting protocol %s/%d", proto.Name, proto.Version))
go func() {
err := proto.Run(p, proto)
if err == nil {
p.log.Trace(fmt.Sprintf("Protocol %s/%d returned", proto.Name, proto.Version))
err = errProtocolReturned
} else if err != io.EOF {
p.log.Trace(fmt.Sprintf("Protocol %s/%d failed", proto.Name, proto.Version), "err", err)
}
p.protoErr <- err
p.wg.Done()
}()
}
}

// getProto finds the protocol responsible for handling
// the given message code.
func (p *Peer) getProto(code uint64) (*protoRW, error) {
for _, proto := range p.running {
if code >= proto.offset && code < proto.offset+proto.Length {
return proto, nil
}
}
```

```
return nil, newPeerError(errInvalidMsgCode, "%d", code)
}

type protoRW struct {
Protocol
in     chan Msg        // receices read messages
closed <-chan struct{} // receives when peer is shutting down
wstart <-chan struct{} // receives when write may start
werr   chan<- error    // for write results
offset uint64
w      MsgWriter
}

func (rw *protoRW) WriteMsg(msg Msg) (err error) {
if msg.Code >= rw.Length {
return newPeerError(errInvalidMsgCode, "not handled")
}
msg.Code += rw.offset
select {
case <-rw.wstart:
err = rw.w.WriteMsg(msg)
// Report write status back to Peer.run. It will initiate
// shutdown if the error is non-nil and unblock the next write
// otherwise. The calling protocol code should exit for errors
// as well but we don't want to rely on that.
rw.werr <- err
case <-rw.closed:
err = fmt.Errorf("shutting down")
}
return err
}

func (rw *protoRW) ReadMsg() (Msg, error) {
select {
case msg := <-rw.in:
msg.Code -= rw.offset
return msg, nil
case <-rw.closed:
return Msg{}, io.EOF
}
}
```

```
// PeerInfo represents a short summary of the information known about a connected
// peer. Sub-protocol independent fields are contained and initialized here, with
// protocol specifics delegated to all connected sub-protocols.
type PeerInfo struct {
ID      string   `json:"id"`   // Unique node identifier (also the encryption key)
Name    string   `json:"name"` // Name of the node, including client type, version, OS, custom data
Caps    []string `json:"caps"` // Sum-protocols advertised by this particular peer
Network struct {
LocalAddress  string `json:"localAddress"`  // Local endpoint of the TCP data connection
RemoteAddress string `json:"remoteAddress"` // Remote endpoint of the TCP data connection
} `json:"network"`
Protocols map[string]interface{} `json:"protocols"` // Sub-protocol specific metadata fields
}

// Info gathers and returns a collection of metadata known about a peer.
func (p *Peer) Info() *PeerInfo {
// Gather the protocol capabilities
var caps []string
for _, cap := range p.Caps() {
caps = append(caps, cap.String())
}
// Assemble the generic peer metadata
info := &PeerInfo{
ID:        p.ID().String(),
Name:      p.Name(),
Caps:      caps,
Protocols: make(map[string]interface{}),
}
info.Network.LocalAddress = p.LocalAddr().String()
info.Network.RemoteAddress = p.RemoteAddr().String()

// Gather all the running protocol infos
for _, proto := range p.running {
protoInfo := interface{}("unknown")
if query := proto.Protocol.PeerInfo; query != nil {
if metadata := query(p.ID()); metadata != nil {
protoInfo = metadata
} else {
protoInfo = "handshake"
}
}
info.Protocols[proto.Name] = protoInfo
}
return info
}

20:F:\git\coin\ethereum\go-ethereum\p2p\peer_error.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package p2p

import (
"errors"
"fmt"
)

const (
errInvalidMsgCode = iota
errInvalidMsg
)

var errorToString = map[int]string{
errInvalidMsgCode: "invalid message code",
errInvalidMsg:     "invalid message",
}

type peerError struct {
code    int
message string
}

func newPeerError(code int, format string, v ...interface{}) *peerError {
desc, ok := errorToString[code]
if !ok {
panic("invalid error code")
}
err := &peerError{code, desc}
if format != "" {
err.message += ": " + fmt.Sprintf(format, v...)
}
return err
}

func (self *peerError) Error() string {
```

```
return self.message
}

var errProtocolReturned = errors.New("protocol returned")

type DiscReason uint

const (
DiscRequested DiscReason = iota
DiscNetworkError
DiscProtocolError
DiscUselessPeer
DiscTooManyPeers
DiscAlreadyConnected
DiscIncompatibleVersion
DiscInvalidIdentity
DiscQuitting
DiscUnexpectedIdentity
DiscSelf
DiscReadTimeout
DiscSubprotocolError = 0x10
)

var discReasonToString = [...]string{
DiscRequested:           "disconnect requested",
DiscNetworkError:        "network error",
DiscProtocolError:       "breach of protocol",
DiscUselessPeer:         "useless peer",
DiscTooManyPeers:        "too many peers",
DiscAlreadyConnected:    "already connected",
DiscIncompatibleVersion: "incompatible p2p protocol version",
DiscInvalidIdentity:     "invalid node identity",
DiscQuitting:            "client quitting",
DiscUnexpectedIdentity:  "unexpected identity",
DiscSelf:                "connected to self",
DiscReadTimeout:         "read timeout",
DiscSubprotocolError:    "subprotocol error",
}

func (d DiscReason) String() string {
if len(discReasonToString) < int(d) {
return fmt.Sprintf("unknown disconnect reason %d", d)
}
return discReasonToString[d]
}

func (d DiscReason) Error() string {
return d.String()
}

func discReasonForError(err error) DiscReason {
if reason, ok := err.(DiscReason); ok {
return reason
}
if err == errProtocolReturned {
return DiscQuitting
}
peerError, ok := err.(*peerError)
if ok {
switch peerError.code {
case errInvalidMsgCode, errInvalidMsg:
return DiscProtocolError
default:
return DiscSubprotocolError
}
}
return DiscSubprotocolError
}
```

21:F:\git\coin\ethereum\go-ethereum\p2p\peer_test.go

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

package p2p

import (
"errors"
"fmt"
"math/rand"
"net"
"reflect"
"testing"
"time"
)
```

```
var discard = Protocol{
Name:   "discard",
Length: 1,
Run: func(p *Peer, rw MsgReadWriter) error {
for {
msg, err := rw.ReadMsg()
if err != nil {
return err
}
fmt.Printf("discarding %d\n", msg.Code)
if err = msg.Discard(); err != nil {
return err
}
}
},
}

func testPeer(protos []Protocol) (func(), *conn, *Peer, <-chan error) {
fd1, fd2 := net.Pipe()
c1 := &conn{fd: fd1, transport: newTestTransport(randomID(), fd1)}
c2 := &conn{fd: fd2, transport: newTestTransport(randomID(), fd2)}
for _, p := range protos {
c1.caps = append(c1.caps, p.cap())
c2.caps = append(c2.caps, p.cap())
}

peer := newPeer(c1, protos)
errc := make(chan error, 1)
go func() {
_, err := peer.run()
errc <- err
}()

closer := func() { c2.close(errors.New("close func called")) }
return closer, c2, peer, errc
}

func TestPeerProtoReadMsg(t *testing.T) {
proto := Protocol{
Name:   "a",
Length: 5,
Run: func(peer *Peer, rw MsgReadWriter) error {
```

```
if err := ExpectMsg(rw, 2, []uint{1}); err != nil {
t.Error(err)
}
if err := ExpectMsg(rw, 3, []uint{2}); err != nil {
t.Error(err)
}
if err := ExpectMsg(rw, 4, []uint{3}); err != nil {
t.Error(err)
}
return nil
},
}

closer, rw, _, errc := testPeer([]Protocol{proto})
defer closer()

Send(rw, baseProtocolLength+2, []uint{1})
Send(rw, baseProtocolLength+3, []uint{2})
Send(rw, baseProtocolLength+4, []uint{3})

select {
case err := <-errc:
if err != errProtocolReturned {
t.Errorf("peer returned error: %v", err)
}
case <-time.After(2 * time.Second):
t.Errorf("receive timeout")
}
}

func TestPeerProtoEncodeMsg(t *testing.T) {
proto := Protocol{
Name:   "a",
Length: 2,
Run: func(peer *Peer, rw MsgReadWriter) error {
if err := SendItems(rw, 2); err == nil {
t.Error("expected error for out-of-range msg code, got nil")
}
if err := SendItems(rw, 1, "foo", "bar"); err != nil {
t.Errorf("write error: %v", err)
}
return nil
```

```
},
}
closer, rw, _, _ := testPeer([]Protocol{proto})
defer closer()

if err := ExpectMsg(rw, 17, []string{"foo", "bar"}); err != nil {
t.Error(err)
}
}

func TestPeerPing(t *testing.T) {
closer, rw, _, _ := testPeer(nil)
defer closer()
if err := SendItems(rw, pingMsg); err != nil {
t.Fatal(err)
}
if err := ExpectMsg(rw, pongMsg, nil); err != nil {
t.Error(err)
}
}

func TestPeerDisconnect(t *testing.T) {
closer, rw, _, disc := testPeer(nil)
defer closer()
if err := SendItems(rw, discMsg, DiscQuitting); err != nil {
t.Fatal(err)
}
select {
case reason := <-disc:
if reason != DiscQuitting {
t.Errorf("run returned wrong reason: got %v, want %v", reason, DiscQuitting)
}
case <-time.After(500 * time.Millisecond):
t.Error("peer did not return")
}
}

// This test is supposed to verify that Peer can reliably handle
// multiple causes of disconnection occurring at the same time.
func TestPeerDisconnectRace(t *testing.T) {
maybe := func() bool { return rand.Intn(1) == 1 }
```

```
for i := 0; i < 1000; i++ {
protoclose := make(chan error)
protodisc := make(chan DiscReason)
closer, rw, p, disc := testPeer([]Protocol{
{
Name:   "closereq",
Run:    func(p *Peer, rw MsgReadWriter) error { return <-protoclose },
Length: 1,
},
{
Name:   "disconnect",
Run:    func(p *Peer, rw MsgReadWriter) error { p.Disconnect(<-protodisc); return nil },
Length: 1,
},
})

// Simulate incoming messages.
go SendItems(rw, baseProtocolLength+1)
go SendItems(rw, baseProtocolLength+2)
// Close the network connection.
go closer()
// Make protocol "closereq" return.
protoclose <- errors.New("protocol closed")
// Make protocol "disconnect" call peer.Disconnect
protodisc <- DiscAlreadyConnected
// In some cases, simulate something else calling peer.Disconnect.
if maybe() {
go p.Disconnect(DiscInvalidIdentity)
}
// In some cases, simulate remote requesting a disconnect.
if maybe() {
go SendItems(rw, discMsg, DiscQuitting)
}

select {
case <-disc:
case <-time.After(2 * time.Second):
// Peer.run should return quickly. If it doesn't the Peer
// goroutines are probably deadlocked. Call panic in order to
// show the stacks.
panic("Peer.run took to long to return.")
}
```

```
}
}

func TestNewPeer(t *testing.T) {
name := "nodename"
caps := []Cap{{"foo", 2}, {"bar", 3}}
id := randomID()
p := NewPeer(id, name, caps)
if p.ID() != id {
t.Errorf("ID mismatch: got %v, expected %v", p.ID(), id)
}
if p.Name() != name {
t.Errorf("Name mismatch: got %v, expected %v", p.Name(), name)
}
if !reflect.DeepEqual(p.Caps(), caps) {
t.Errorf("Caps mismatch: got %v, expected %v", p.Caps(), caps)
}

p.Disconnect(DiscAlreadyConnected) // Should not hang
}

func TestMatchProtocols(t *testing.T) {
tests := []struct {
Remote []Cap
Local  []Protocol
Match  map[string]protoRW
}{
{
// No remote capabilities
Local: []Protocol{{Name: "a"}},
},
{
// No local protocols
Remote: []Cap{{Name: "a"}},
},
{
// No mutual protocols
Remote: []Cap{{Name: "a"}},
Local:  []Protocol{{Name: "b"}},
},
{
// Some matches, some differences
```

```
Remote: []Cap{{Name: "local"}, {Name: "match1"}, {Name: "match2"}},
Local:  []Protocol{{Name: "match1"}, {Name: "match2"}, {Name: "remote"}},
Match:  map[string]protoRW{"match1": {Protocol: Protocol{Name: "match1"}}, "match2": {Protocol:
Protocol{Name: "match2"}}},
},
{
// Various alphabetical ordering
Remote: []Cap{{Name: "aa"}, {Name: "ab"}, {Name: "bb"}, {Name: "ba"}},
Local:  []Protocol{{Name: "ba"}, {Name: "bb"}, {Name: "ab"}, {Name: "aa"}},
Match:  map[string]protoRW{"aa": {Protocol: Protocol{Name: "aa"}}, "ab": {Protocol:
Protocol{Name: "ab"}}, "ba": {Protocol: Protocol{Name: "ba"}}, "bb": {Protocol: Protocol{Name:
"bb"}}},
},
{
// No mutual versions
Remote: []Cap{{Version: 1}},
Local:  []Protocol{{Version: 2}},
},
{
// Multiple versions, single common
Remote: []Cap{{Version: 1}, {Version: 2}},
Local:  []Protocol{{Version: 2}, {Version: 3}},
Match:  map[string]protoRW{"": {Protocol: Protocol{Version: 2}}},
},
{
// Multiple versions, multiple common
Remote: []Cap{{Version: 1}, {Version: 2}, {Version: 3}, {Version: 4}},
Local:  []Protocol{{Version: 2}, {Version: 3}},
Match:  map[string]protoRW{"": {Protocol: Protocol{Version: 3}}},
},
{
// Various version orderings
Remote: []Cap{{Version: 4}, {Version: 1}, {Version: 3}, {Version: 2}},
Local:  []Protocol{{Version: 2}, {Version: 3}, {Version: 1}},
Match:  map[string]protoRW{"": {Protocol: Protocol{Version: 3}}},
},
{
// Versions overriding sub-protocol lengths
Remote: []Cap{{Version: 1}, {Version: 2}, {Version: 3}, {Name: "a"}},
Local:  []Protocol{{Version: 1, Length: 1}, {Version: 2, Length: 2}, {Version: 3, Length: 3}, {Name:
"a"}},
Match:  map[string]protoRW{"": {Protocol: Protocol{Version: 3}}, "a": {Protocol: Protocol{Name:
```

```
"a"}, offset: 3}},
},
}

for i, tt := range tests {
result := matchProtocols(tt.Local, tt.Remote, nil)
if len(result) != len(tt.Match) {
t.Errorf("test %d: negotiation mismatch: have %v, want %v", i, len(result), len(tt.Match))
continue
}
// Make sure all negotiated protocols are needed and correct
for name, proto := range result {
match, ok := tt.Match[name]
if !ok {
t.Errorf("test %d, proto '%s': negotiated but shouldn't have", i, name)
continue
}
if proto.Name != match.Name {
t.Errorf("test %d, proto '%s': name mismatch: have %v, want %v", i, name, proto.Name,
match.Name)
}
if proto.Version != match.Version {
t.Errorf("test %d, proto '%s': version mismatch: have %v, want %v", i, name, proto.Version,
match.Version)
}
if proto.offset-baseProtocolLength != match.offset {
t.Errorf("test %d, proto '%s': offset mismatch: have %v, want %v", i, name, proto.offset-
baseProtocolLength, match.offset)
}
}
// Make sure no protocols missed negotiation
for name := range tt.Match {
if _, ok := result[name]; !ok {
t.Errorf("test %d, proto '%s': not negotiated, should have", i, name)
continue
}
}
}
}
```

22:F:\git\coin\ethereum\go-ethereum\p2p\protocol.go

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

```
package p2p

import (
"fmt"

"github.com/ethereum/go-ethereum/p2p/discover"
)

// Protocol represents a P2P subprotocol implementation.
type Protocol struct {
// Name should contain the official protocol name,
// often a three-letter word.
Name string

// Version should contain the version number of the protocol.
Version uint

// Length should contain the number of message codes used
// by the protocol.
Length uint64

// Run is called in a new groutine when the protocol has been
// negotiated with a peer. It should read and write messages from
// rw. The Payload for each message must be fully consumed.
//
// The peer connection is closed when Start returns. It should return
// any protocol-level error (such as an I/O error) that is
// encountered.
Run func(peer *Peer, rw MsgReadWriter) error

// NodeInfo is an optional helper method to retrieve protocol specific metadata
// about the host node.
NodeInfo func() interface{}

// PeerInfo is an optional helper method to retrieve protocol specific metadata
// about a certain peer in the network. If an info retrieval function is set,
// but returns nil, it is assumed that the protocol handshake is still running.
PeerInfo func(id discover.NodeID) interface{}
}

func (p Protocol) cap() Cap {
```

```
return Cap{p.Name, p.Version}
}

// Cap is the structure of a peer capability.
type Cap struct {
Name    string
Version uint
}

func (cap Cap) RlpData() interface{} {
return []interface{}{cap.Name, cap.Version}
}

func (cap Cap) String() string {
return fmt.Sprintf("%s/%d", cap.Name, cap.Version)
}

type capsByNameAndVersion []Cap

func (cs capsByNameAndVersion) Len() int      { return len(cs) }
func (cs capsByNameAndVersion) Swap(i, j int) { cs[i], cs[j] = cs[j], cs[i] }
func (cs capsByNameAndVersion) Less(i, j int) bool {
return cs[i].Name < cs[j].Name || (cs[i].Name == cs[j].Name && cs[i].Version < cs[j].Version)
}

23:F:\git\coin\ethereum\go-ethereum\p2p\rlpx.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package p2p

import (
"bytes"
"crypto/aes"
"crypto/cipher"
"crypto/ecdsa"
"crypto/elliptic"
"crypto/hmac"
"crypto/rand"
"encoding/binary"
"errors"
"fmt"
"hash"
```

```
"io"
mrand "math/rand"
"net"
"sync"
"time"

"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/crypto/ecies"
"github.com/ethereum/go-ethereum/crypto/secp256k1"
"github.com/ethereum/go-ethereum/crypto/sha3"
"github.com/ethereum/go-ethereum/p2p/discover"
"github.com/ethereum/go-ethereum/rlp"
)

const (
maxUint24 = ^uint32(0) >> 8

sskLen = 16 // ecies.MaxSharedKeyLength(pubKey) / 2
sigLen = 65 // elliptic S256
pubLen = 64 // 512 bit pubkey in uncompressed representation without format byte
shaLen = 32 // hash length (for nonce etc)

authMsgLen  = sigLen + shaLen + pubLen + shaLen + 1
authRespLen = pubLen + shaLen + 1

eciesOverhead = 65 /* pubkey */ + 16 /* IV */ + 32 /* MAC */

encAuthMsgLen  = authMsgLen + eciesOverhead  // size of encrypted pre-EIP-8 initiator
handshake
encAuthRespLen = authRespLen + eciesOverhead // size of encrypted pre-EIP-8 handshake reply

// total timeout for encryption handshake and protocol
// handshake in both directions.
handshakeTimeout = 5 * time.Second

// This is the timeout for sending the disconnect reason.
// This is shorter than the usual timeout because we don't want
// to wait if the connection is known to be bad anyway.
discWriteTimeout = 1 * time.Second
)

// rlpx is the transport protocol used by actual (non-test) connections.
// It wraps the frame encoder with locks and read/write deadlines.
type rlpx struct {
fd net.Conn

rmu, wmu sync.Mutex
rw       *rlpxFrameRW
}

func newRLPX(fd net.Conn) transport {
fd.SetDeadline(time.Now().Add(handshakeTimeout))
return &rlpx{fd: fd}
}

func (t *rlpx) ReadMsg() (Msg, error) {
t.rmu.Lock()
defer t.rmu.Unlock()
t.fd.SetReadDeadline(time.Now().Add(frameReadTimeout))
return t.rw.ReadMsg()
}

func (t *rlpx) WriteMsg(msg Msg) error {
t.wmu.Lock()
defer t.wmu.Unlock()
t.fd.SetWriteDeadline(time.Now().Add(frameWriteTimeout))
return t.rw.WriteMsg(msg)
}

func (t *rlpx) close(err error) {
t.wmu.Lock()
defer t.wmu.Unlock()
// Tell the remote end why we're disconnecting if possible.
if t.rw != nil {
if r, ok := err.(DiscReason); ok && r != DiscNetworkError {
t.fd.SetWriteDeadline(time.Now().Add(discWriteTimeout))
SendItems(t.rw, discMsg, r)
}
}
t.fd.Close()
}

// doEncHandshake runs the protocol handshake using authenticated
// messages. the protocol handshake is the first authenticated message
// and also verifies whether the encryption handshake 'worked' and the
// remote side actually provided the right public key.
func (t *rlpx) doProtoHandshake(our *protoHandshake) (their *protoHandshake, err error) {
// Writing our handshake happens concurrently, we prefer
// returning the handshake read error. If the remote side
// disconnects us early with a valid reason, we should return it
// as the error so it can be tracked elsewhere.
werr := make(chan error, 1)
go func() { werr <- Send(t.rw, handshakeMsg, our) }()
if their, err = readProtocolHandshake(t.rw, our); err != nil {
<-werr // make sure the write terminates too
return nil, err
}
if err := <-werr; err != nil {
return nil, fmt.Errorf("write error: %v", err)
}
return their, nil
}

func readProtocolHandshake(rw MsgReader, our *protoHandshake) (*protoHandshake, error) {
msg, err := rw.ReadMsg()
if err != nil {
return nil, err
}
if msg.Size > baseProtocolMaxMsgSize {
return nil, fmt.Errorf("message too big")
}
if msg.Code == discMsg {
// Disconnect before protocol handshake is valid according to the
// spec and we send it ourself if the posthanshake checks fail.
// We can't return the reason directly, though, because it is echoed
// back otherwise. Wrap it in a string instead.
var reason [1]DiscReason
rlp.Decode(msg.Payload, &reason)
return nil, reason[0]
}
if msg.Code != handshakeMsg {
return nil, fmt.Errorf("expected handshake, got %x", msg.Code)
}
var hs protoHandshake
if err := msg.Decode(&hs); err != nil {
return nil, err
}
if (hs.ID == discover.NodeID{}) {
return nil, DiscInvalidIdentity
}
return &hs, nil
}

func (t *rlpx) doEncHandshake(prv *ecdsa.PrivateKey, dial *discover.Node) (discover.NodeID,
error) {
var (
sec secrets
err error
)
if dial == nil {
sec, err = receiverEncHandshake(t.fd, prv, nil)
} else {
sec, err = initiatorEncHandshake(t.fd, prv, dial.ID, nil)
}
if err != nil {
return discover.NodeID{}, err
}
t.wmu.Lock()
t.rw = newRLPXFrameRW(t.fd, sec)
t.wmu.Unlock()
return sec.RemoteID, nil
}

// encHandshake contains the state of the encryption handshake.
type encHandshake struct {
initiator bool
remoteID  discover.NodeID

remotePub            *ecies.PublicKey  // remote-pubk
initNonce, respNonce []byte            // nonce
randomPrivKey        *ecies.PrivateKey // ecdhe-random
remoteRandomPub      *ecies.PublicKey  // ecdhe-random-pubk
}

// secrets represents the connection secrets
// which are negotiated during the encryption handshake.
type secrets struct {
RemoteID              discover.NodeID
```

```
AES, MAC              []byte
EgressMAC, IngressMAC hash.Hash
Token                 []byte
}

// RLPx v4 handshake auth (defined in EIP-8).
type authMsgV4 struct {
gotPlain bool // whether read packet had plain format.

Signature       [sigLen]byte
InitiatorPubkey [pubLen]byte
Nonce           [shaLen]byte
Version         uint

// Ignore additional fields (forward-compatibility)
Rest []rlp.RawValue `rlp:"tail"`
}

// RLPx v4 handshake response (defined in EIP-8).
type authRespV4 struct {
RandomPubkey [pubLen]byte
Nonce        [shaLen]byte
Version      uint

// Ignore additional fields (forward-compatibility)
Rest []rlp.RawValue `rlp:"tail"`
}

// secrets is called after the handshake is completed.
// It extracts the connection secrets from the handshake values.
func (h *encHandshake) secrets(auth, authResp []byte) (secrets, error) {
ecdheSecret, err := h.randomPrivKey.GenerateShared(h.remoteRandomPub, sskLen, sskLen)
if err != nil {
return secrets{}, err
}

// derive base secrets from ephemeral key agreement
sharedSecret := crypto.Keccak256(ecdheSecret, crypto.Keccak256(h.respNonce, h.initNonce))
aesSecret := crypto.Keccak256(ecdheSecret, sharedSecret)
s := secrets{
RemoteID: h.remoteID,
AES:      aesSecret,
MAC:      crypto.Keccak256(ecdheSecret, aesSecret),
}

// setup sha3 instances for the MACs
mac1 := sha3.NewKeccak256()
mac1.Write(xor(s.MAC, h.respNonce))
mac1.Write(auth)
mac2 := sha3.NewKeccak256()
mac2.Write(xor(s.MAC, h.initNonce))
mac2.Write(authResp)
if h.initiator {
s.EgressMAC, s.IngressMAC = mac1, mac2
} else {
s.EgressMAC, s.IngressMAC = mac2, mac1
}

return s, nil
}

// staticSharedSecret returns the static shared secret, the result
// of key agreement between the local and remote static node key.
func (h *encHandshake) staticSharedSecret(prv *ecdsa.PrivateKey) ([]byte, error) {
return ecies.ImportECDSA(prv).GenerateShared(h.remotePub, sskLen, sskLen)
}

// initiatorEncHandshake negotiates a session token on conn.
// it should be called on the dialing side of the connection.
//
// prv is the local client's private key.
func initiatorEncHandshake(conn io.ReadWriter, prv *ecdsa.PrivateKey, remoteID
discover.NodeID, token []byte) (s secrets, err error) {
h := &encHandshake{initiator: true, remoteID: remoteID}
authMsg, err := h.makeAuthMsg(prv, token)
if err != nil {
return s, err
}
authPacket, err := sealEIP8(authMsg, h)
if err != nil {
return s, err
}
if _, err = conn.Write(authPacket); err != nil {
return s, err
```

```
}

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

// makeAuthMsg creates the initiator handshake message.
func (h *encHandshake) makeAuthMsg(prv *ecdsa.PrivateKey, token []byte) (*authMsgV4, error) {
rpub, err := h.remoteID.Pubkey()
if err != nil {
return nil, fmt.Errorf("bad remoteID: %v", err)
}
h.remotePub = ecies.ImportECDSAPublic(rpub)
// Generate random initiator nonce.
h.initNonce = make([]byte, shaLen)
if _, err := rand.Read(h.initNonce); err != nil {
return nil, err
}
// Generate random keypair to for ECDH.
h.randomPrivKey, err = ecies.GenerateKey(rand.Reader, crypto.S256(), nil)
if err != nil {
return nil, err
}

// Sign known message: static-shared-secret ^ nonce
token, err = h.staticSharedSecret(prv)
if err != nil {
return nil, err
}
signed := xor(token, h.initNonce)
signature, err := crypto.Sign(signed, h.randomPrivKey.ExportECDSA())
if err != nil {
return nil, err
}
```

```
msg := new(authMsgV4)
copy(msg.Signature[:], signature)
copy(msg.InitiatorPubkey[:], crypto.FromECDSAPub(&prv.PublicKey)[1:])
copy(msg.Nonce[:], h.initNonce)
msg.Version = 4
return msg, nil
}

func (h *encHandshake) handleAuthResp(msg *authRespV4) (err error) {
h.respNonce = msg.Nonce[:]
h.remoteRandomPub, err = importPublicKey(msg.RandomPubkey[:])
return err
}

// receiverEncHandshake negotiates a session token on conn.
// it should be called on the listening side of the connection.
//
// prv is the local client's private key.
// token is the token from a previous session with this node.
func receiverEncHandshake(conn io.ReadWriter, prv *ecdsa.PrivateKey, token []byte) (s secrets,
err error) {
authMsg := new(authMsgV4)
authPacket, err := readHandshakeMsg(authMsg, encAuthMsgLen, prv, conn)
if err != nil {
return s, err
}
h := new(encHandshake)
if err := h.handleAuthMsg(authMsg, prv); err != nil {
return s, err
}

authRespMsg, err := h.makeAuthResp()
if err != nil {
return s, err
}
var authRespPacket []byte
if authMsg.gotPlain {
authRespPacket, err = authRespMsg.sealPlain(h)
} else {
authRespPacket, err = sealEIP8(authRespMsg, h)
}
if err != nil {
```

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

func (h *encHandshake) handleAuthMsg(msg *authMsgV4, prv *ecdsa.PrivateKey) error {
// Import the remote identity.
h.initNonce = msg.Nonce[:]
h.remoteID = msg.InitiatorPubkey
rpub, err := h.remoteID.Pubkey()
if err != nil {
return fmt.Errorf("bad remoteID: %#v", err)
}
h.remotePub = ecies.ImportECDSAPublic(rpub)

// Generate random keypair for ECDH.
// If a private key is already set, use it instead of generating one (for testing).
if h.randomPrivKey == nil {
h.randomPrivKey, err = ecies.GenerateKey(rand.Reader, crypto.S256(), nil)
if err != nil {
return err
}
}

// Check the signature.
token, err := h.staticSharedSecret(prv)
if err != nil {
return err
}
signedMsg := xor(token, h.initNonce)
remoteRandomPub, err := secp256k1.RecoverPubkey(signedMsg, msg.Signature[:])
if err != nil {
return err
}
h.remoteRandomPub, _ = importPublicKey(remoteRandomPub)
return nil
}

func (h *encHandshake) makeAuthResp() (msg *authRespV4, err error) {
```

```
// Generate random nonce.
h.respNonce = make([]byte, shaLen)
if _, err = rand.Read(h.respNonce); err != nil {
return nil, err
}

msg = new(authRespV4)
copy(msg.Nonce[:], h.respNonce)
copy(msg.RandomPubkey[:], exportPubkey(&h.randomPrivKey.PublicKey))
msg.Version = 4
return msg, nil
}

func (msg *authMsgV4) sealPlain(h *encHandshake) ([]byte, error) {
buf := make([]byte, authMsgLen)
n := copy(buf, msg.Signature[:])
n += copy(buf[n:], crypto.Keccak256(exportPubkey(&h.randomPrivKey.PublicKey)))
n += copy(buf[n:], msg.InitiatorPubkey[:])
n += copy(buf[n:], msg.Nonce[:])
buf[n] = 0 // token-flag
return ecies.Encrypt(rand.Reader, h.remotePub, buf, nil, nil)
}

func (msg *authMsgV4) decodePlain(input []byte) {
n := copy(msg.Signature[:], input)
n += shaLen // skip sha3(initiator-ephemeral-pubk)
n += copy(msg.InitiatorPubkey[:], input[n:])
copy(msg.Nonce[:], input[n:])
msg.Version = 4
msg.gotPlain = true
}

func (msg *authRespV4) sealPlain(hs *encHandshake) ([]byte, error) {
buf := make([]byte, authRespLen)
n := copy(buf, msg.RandomPubkey[:])
copy(buf[n:], msg.Nonce[:])
return ecies.Encrypt(rand.Reader, hs.remotePub, buf, nil, nil)
}

func (msg *authRespV4) decodePlain(input []byte) {
n := copy(msg.RandomPubkey[:], input)
copy(msg.Nonce[:], input[n:])
```

```
msg.Version = 4
}

var padSpace = make([]byte, 300)

func sealEIP8(msg interface{}, h *encHandshake) ([]byte, error) {
buf := new(bytes.Buffer)
if err := rlp.Encode(buf, msg); err != nil {
return nil, err
}
// pad with random amount of data. the amount needs to be at least 100 bytes to make
// the message distinguishable from pre-EIP-8 handshakes.
pad := padSpace[:mrand.Intn(len(padSpace)-100)+100]
buf.Write(pad)
prefix := make([]byte, 2)
binary.BigEndian.PutUint16(prefix, uint16(buf.Len()+eciesOverhead))

enc, err := ecies.Encrypt(rand.Reader, h.remotePub, buf.Bytes(), nil, prefix)
return append(prefix, enc...), err
}

type plainDecoder interface {
decodePlain([]byte)
}

func readHandshakeMsg(msg plainDecoder, plainSize int, prv *ecdsa.PrivateKey, r io.Reader)
([]byte, error) {
buf := make([]byte, plainSize)
if _, err := io.ReadFull(r, buf); err != nil {
return buf, err
}
// Attempt decoding pre-EIP-8 "plain" format.
key := ecies.ImportECDSA(prv)
if dec, err := key.Decrypt(rand.Reader, buf, nil, nil); err == nil {
msg.decodePlain(dec)
return buf, nil
}
// Could be EIP-8 format, try that.
prefix := buf[:2]
size := binary.BigEndian.Uint16(prefix)
if size < uint16(plainSize) {
return buf, fmt.Errorf("size underflow, need at least %d bytes", plainSize)
}
buf = append(buf, make([]byte, size-uint16(plainSize)+2)...)
if _, err := io.ReadFull(r, buf[plainSize:]); err != nil {
return buf, err
}
dec, err := key.Decrypt(rand.Reader, buf[2:], nil, prefix)
if err != nil {
return buf, err
}
// Can't use rlp.DecodeBytes here because it rejects
// trailing data (forward-compatibility).
s := rlp.NewStream(bytes.NewReader(dec), 0)
return buf, s.Decode(msg)
}

// importPublicKey unmarshals 512 bit public keys.
func importPublicKey(pubKey []byte) (*ecies.PublicKey, error) {
var pubKey65 []byte
switch len(pubKey) {
case 64:
// add 'uncompressed key' flag
pubKey65 = append([]byte{0x04}, pubKey...)
case 65:
pubKey65 = pubKey
default:
return nil, fmt.Errorf("invalid public key length %v (expect 64/65)", len(pubKey))
}
// TODO: fewer pointless conversions
pub := crypto.ToECDSAPub(pubKey65)
if pub.X == nil {
return nil, fmt.Errorf("invalid public key")
}
return ecies.ImportECDSAPublic(pub), nil
}

func exportPubkey(pub *ecies.PublicKey) []byte {
if pub == nil {
panic("nil pubkey")
}
return elliptic.Marshal(pub.Curve, pub.X, pub.Y)[1:]
}
```

```
func xor(one, other []byte) (xor []byte) {
xor = make([]byte, len(one))
for i := 0; i < len(one); i++ {
xor[i] = one[i] ^ other[i]
}
return xor
}

var (
// this is used in place of actual frame header data.
// TODO: replace this when Msg contains the protocol type code.
zeroHeader = []byte{0xC2, 0x80, 0x80}
// sixteen zero bytes
zero16 = make([]byte, 16)
)

// rlpxFrameRW implements a simplified version of RLPx framing.
// chunked messages are not supported and all headers are equal to
// zeroHeader.
//
// rlpxFrameRW is not safe for concurrent use from multiple goroutines.
type rlpxFrameRW struct {
conn io.ReadWriter
enc  cipher.Stream
dec  cipher.Stream

macCipher  cipher.Block
egressMAC  hash.Hash
ingressMAC hash.Hash
}

func newRLPXFrameRW(conn io.ReadWriter, s secrets) *rlpxFrameRW {
macc, err := aes.NewCipher(s.MAC)
if err != nil {
panic("invalid MAC secret: " + err.Error())
}
encc, err := aes.NewCipher(s.AES)
if err != nil {
panic("invalid AES secret: " + err.Error())
}
// we use an all-zeroes IV for AES because the key used
// for encryption is ephemeral.
```

```
iv := make([]byte, encc.BlockSize())
return &rlpxFrameRW{
conn:       conn,
enc:        cipher.NewCTR(encc, iv),
dec:        cipher.NewCTR(encc, iv),
macCipher:  macc,
egressMAC:  s.EgressMAC,
ingressMAC: s.IngressMAC,
}
}

func (rw *rlpxFrameRW) WriteMsg(msg Msg) error {
ptype, _ := rlp.EncodeToBytes(msg.Code)

// write header
headbuf := make([]byte, 32)
fsize := uint32(len(ptype)) + msg.Size
if fsize > maxUint24 {
return errors.New("message size overflows uint24")
}
putInt24(fsize, headbuf) // TODO: check overflow
copy(headbuf[3:], zeroHeader)
rw.enc.XORKeyStream(headbuf[:16], headbuf[:16]) // first half is now encrypted

// write header MAC
copy(headbuf[16:], updateMAC(rw.egressMAC, rw.macCipher, headbuf[:16]))
if _, err := rw.conn.Write(headbuf); err != nil {
return err
}

// write encrypted frame, updating the egress MAC hash with
// the data written to conn.
tee := cipher.StreamWriter{S: rw.enc, W: io.MultiWriter(rw.conn, rw.egressMAC)}
if _, err := tee.Write(ptype); err != nil {
return err
}
if _, err := io.Copy(tee, msg.Payload); err != nil {
return err
}
if padding := fsize % 16; padding > 0 {
if _, err := tee.Write(zero16[:16-padding]); err != nil {
return err
}
}

// write frame MAC. egress MAC hash is up to date because
// frame content was written to it as well.
fmacseed := rw.egressMAC.Sum(nil)
mac := updateMAC(rw.egressMAC, rw.macCipher, fmacseed)
_, err := rw.conn.Write(mac)
return err
}

func (rw *rlpxFrameRW) ReadMsg() (msg Msg, err error) {
// read the header
headbuf := make([]byte, 32)
if _, err := io.ReadFull(rw.conn, headbuf); err != nil {
return msg, err
}
// verify header mac
shouldMAC := updateMAC(rw.ingressMAC, rw.macCipher, headbuf[:16])
if !hmac.Equal(shouldMAC, headbuf[16:]) {
return msg, errors.New("bad header MAC")
}
rw.dec.XORKeyStream(headbuf[:16], headbuf[:16]) // first half is now decrypted
fsize := readInt24(headbuf)
// ignore protocol type for now

// read the frame content
var rsize = fsize // frame size rounded up to 16 byte boundary
if padding := fsize % 16; padding > 0 {
rsize += 16 - padding
}
framebuf := make([]byte, rsize)
if _, err := io.ReadFull(rw.conn, framebuf); err != nil {
return msg, err
}

// read and validate frame MAC. we can re-use headbuf for that.
rw.ingressMAC.Write(framebuf)
fmacseed := rw.ingressMAC.Sum(nil)
if _, err := io.ReadFull(rw.conn, headbuf[:16]); err != nil {
return msg, err
}
```

```
shouldMAC = updateMAC(rw.ingressMAC, rw.macCipher, fmacseed)
if !hmac.Equal(shouldMAC, headbuf[:16]) {
return msg, errors.New("bad frame MAC")
}

// decrypt frame content
rw.dec.XORKeyStream(framebuf, framebuf)

// decode message code
content := bytes.NewReader(framebuf[:fsize])
if err := rlp.Decode(content, &msg.Code); err != nil {
return msg, err
}
msg.Size = uint32(content.Len())
msg.Payload = content
return msg, nil
}

// updateMAC reseeds the given hash with encrypted seed.
// it returns the first 16 bytes of the hash sum after seeding.
func updateMAC(mac hash.Hash, block cipher.Block, seed []byte) []byte {
aesbuf := make([]byte, aes.BlockSize)
block.Encrypt(aesbuf, mac.Sum(nil))
for i := range aesbuf {
aesbuf[i] ^= seed[i]
}
mac.Write(aesbuf)
return mac.Sum(nil)[:16]
}

func readInt24(b []byte) uint32 {
return uint32(b[2]) | uint32(b[1])<<8 | uint32(b[0])<<16
}

func putInt24(v uint32, b []byte) {
b[0] = byte(v >> 16)
b[1] = byte(v >> 8)
b[2] = byte(v)
}
```

24:F:\git\coin\ethereum\go-ethereum\p2p\rlpx_test.go

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

```
package p2p

import (
"bytes"
"crypto/rand"
"errors"
"fmt"
"io"
"io/ioutil"
"net"
"reflect"
"strings"
"sync"
"testing"
"time"

"github.com/davecgh/go-spew/spew"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/crypto/ecies"
"github.com/ethereum/go-ethereum/crypto/sha3"
"github.com/ethereum/go-ethereum/p2p/discover"
"github.com/ethereum/go-ethereum/rlp"
)

func TestSharedSecret(t *testing.T) {
prv0, _ := crypto.GenerateKey() // = ecdsa.GenerateKey(crypto.S256(), rand.Reader)
pub0 := &prv0.PublicKey
prv1, _ := crypto.GenerateKey()
pub1 := &prv1.PublicKey

ss0, err := ecies.ImportECDSA(prv0).GenerateShared(ecies.ImportECDSAPublic(pub1), sskLen,
sskLen)
if err != nil {
return
}
ss1, err := ecies.ImportECDSA(prv1).GenerateShared(ecies.ImportECDSAPublic(pub0), sskLen,
sskLen)
if err != nil {
return
}
t.Logf("Secret:\n%v %x\n%v %x", len(ss0), ss0, len(ss0), ss1)
```

```
if !bytes.Equal(ss0, ss1) {
t.Errorf("dont match :(")
}
}

func TestEncHandshake(t *testing.T) {
for i := 0; i < 10; i++ {
start := time.Now()
if err := testEncHandshake(nil); err != nil {
t.Fatalf("i=%d %v", i, err)
}
t.Logf("(without token) %d %v\n", i+1, time.Since(start))
}
for i := 0; i < 10; i++ {
tok := make([]byte, shaLen)
rand.Reader.Read(tok)
start := time.Now()
if err := testEncHandshake(tok); err != nil {
t.Fatalf("i=%d %v", i, err)
}
t.Logf("(with token) %d %v\n", i+1, time.Since(start))
}
}

func testEncHandshake(token []byte) error {
type result struct {
side string
id   discover.NodeID
err  error
}
var (
prv0, _  = crypto.GenerateKey()
prv1, _  = crypto.GenerateKey()
fd0, fd1 = net.Pipe()
c0, c1   = newRLPX(fd0).(*rlpx), newRLPX(fd1).(*rlpx)
output   = make(chan result)
)

go func() {
r := result{side: "initiator"}
defer func() { output <- r }()
defer fd0.Close()
```

```
dest := &discover.Node{ID: discover.PubkeyID(&prv1.PublicKey)}
r.id, r.err = c0.doEncHandshake(prv0, dest)
if r.err != nil {
return
}
id1 := discover.PubkeyID(&prv1.PublicKey)
if r.id != id1 {
r.err = fmt.Errorf("remote ID mismatch: got %v, want: %v", r.id, id1)
}
}()
go func() {
r := result{side: "receiver"}
defer func() { output <- r }()
defer fd1.Close()

r.id, r.err = c1.doEncHandshake(prv1, nil)
if r.err != nil {
return
}
id0 := discover.PubkeyID(&prv0.PublicKey)
if r.id != id0 {
r.err = fmt.Errorf("remote ID mismatch: got %v, want: %v", r.id, id0)
}
}()

// wait for results from both sides
r1, r2 := <-output, <-output
if r1.err != nil {
return fmt.Errorf("%s side error: %v", r1.side, r1.err)
}
if r2.err != nil {
return fmt.Errorf("%s side error: %v", r2.side, r2.err)
}

// compare derived secrets
if !reflect.DeepEqual(c0.rw.egressMAC, c1.rw.ingressMAC) {
return fmt.Errorf("egress mac mismatch:\n c0.rw: %#v\n c1.rw: %#v", c0.rw.egressMAC,
c1.rw.ingressMAC)
}
if !reflect.DeepEqual(c0.rw.ingressMAC, c1.rw.egressMAC) {
return fmt.Errorf("ingress mac mismatch:\n c0.rw: %#v\n c1.rw: %#v", c0.rw.ingressMAC,
c1.rw.egressMAC)
}
if !reflect.DeepEqual(c0.rw.enc, c1.rw.enc) {
return fmt.Errorf("enc cipher mismatch:\n c0.rw: %#v\n c1.rw: %#v", c0.rw.enc, c1.rw.enc)
}
if !reflect.DeepEqual(c0.rw.dec, c1.rw.dec) {
return fmt.Errorf("dec cipher mismatch:\n c0.rw: %#v\n c1.rw: %#v", c0.rw.dec, c1.rw.dec)
}
return nil
}

func TestProtocolHandshake(t *testing.T) {
var (
prv0, _ = crypto.GenerateKey()
node0   = &discover.Node{ID: discover.PubkeyID(&prv0.PublicKey), IP: net.IP{1, 2, 3, 4}, TCP: 33}
hs0     = &protoHandshake{Version: 3, ID: node0.ID, Caps: []Cap{{"a", 0}, {"b", 2}}}

prv1, _ = crypto.GenerateKey()
node1   = &discover.Node{ID: discover.PubkeyID(&prv1.PublicKey), IP: net.IP{5, 6, 7, 8}, TCP: 44}
hs1     = &protoHandshake{Version: 3, ID: node1.ID, Caps: []Cap{{"c", 1}, {"d", 3}}}

fd0, fd1 = net.Pipe()
wg       sync.WaitGroup
)

wg.Add(2)
go func() {
defer wg.Done()
defer fd1.Close()
rlpx := newRLPX(fd0)
remid, err := rlpx.doEncHandshake(prv0, node1)
if err != nil {
t.Errorf("dial side enc handshake failed: %v", err)
return
}
if remid != node1.ID {
t.Errorf("dial side remote id mismatch: got %v, want %v", remid, node1.ID)
return
}

phs, err := rlpx.doProtoHandshake(hs0)
if err != nil {
```

```
t.Errorf("dial side proto handshake error: %v", err)
return
}
phs.Rest = nil
if !reflect.DeepEqual(phs, hs1) {
t.Errorf("dial side proto handshake mismatch:\ngot: %s\nwant: %s\n", spew.Sdump(phs),
spew.Sdump(hs1))
return
}
rlpx.close(DiscQuitting)
}()
go func() {
defer wg.Done()
defer fd1.Close()
rlpx := newRLPX(fd1)
remid, err := rlpx.doEncHandshake(prv1, nil)
if err != nil {
t.Errorf("listen side enc handshake failed: %v", err)
return
}
if remid != node0.ID {
t.Errorf("listen side remote id mismatch: got %v, want %v", remid, node0.ID)
return
}

phs, err := rlpx.doProtoHandshake(hs1)
if err != nil {
t.Errorf("listen side proto handshake error: %v", err)
return
}
phs.Rest = nil
if !reflect.DeepEqual(phs, hs0) {
t.Errorf("listen side proto handshake mismatch:\ngot: %s\nwant: %s\n", spew.Sdump(phs),
spew.Sdump(hs0))
return
}

if err := ExpectMsg(rlpx, discMsg, []DiscReason{DiscQuitting}); err != nil {
t.Errorf("error receiving disconnect: %v", err)
}
}()
wg.Wait()
```

```
}

func TestProtocolHandshakeErrors(t *testing.T) {
our := &protoHandshake{Version: 3, Caps: []Cap{{"foo", 2}, {"bar", 3}}, Name: "quux"}
tests := []struct {
code uint64
msg  interface{}
err  error
}{
{
code: discMsg,
msg:  []DiscReason{DiscQuitting},
err:  DiscQuitting,
},
{
code: 0x989898,
msg:  []byte{1},
err:  errors.New("expected handshake, got 989898"),
},
{
code: handshakeMsg,
msg:  make([]byte, baseProtocolMaxMsgSize+2),
err:  errors.New("message too big"),
},
{
code: handshakeMsg,
msg:  []byte{1, 2, 3},
err:  newPeerError(errInvalidMsg, "(code 0) (size 4) rlp: expected input list for
p2p.protoHandshake"),
},
{
code: handshakeMsg,
msg:  &protoHandshake{Version: 3},
err:  DiscInvalidIdentity,
},
}

for i, test := range tests {
p1, p2 := MsgPipe()
go Send(p1, test.code, test.msg)
_, err := readProtocolHandshake(p2, our)
if !reflect.DeepEqual(err, test.err) {
```

```
t.Errorf("test %d: error mismatch: got %q, want %q", i, err, test.err)
}
}
}

func TestRLPXFrameFake(t *testing.T) {
buf := new(bytes.Buffer)
hash := fakeHash([]byte{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})
rw := newRLPXFrameRW(buf, secrets{
AES:        crypto.Keccak256(),
MAC:        crypto.Keccak256(),
IngressMAC: hash,
EgressMAC:  hash,
})

golden := unhex(`
00828ddae471818bb0bfa6b551d1cb42
01010101010101010101010101010101
ba628a4ba590cb43f7848f41c4382885
01010101010101010101010101010101
`)

// Check WriteMsg. This puts a message into the buffer.
if err := Send(rw, 8, []uint{1, 2, 3, 4}); err != nil {
t.Fatalf("WriteMsg error: %v", err)
}
written := buf.Bytes()
if !bytes.Equal(written, golden) {
t.Fatalf("output mismatch:\n  got:  %x\n  want: %x", written, golden)
}

// Check ReadMsg. It reads the message encoded by WriteMsg, which
// is equivalent to the golden message above.
msg, err := rw.ReadMsg()
if err != nil {
t.Fatalf("ReadMsg error: %v", err)
}
if msg.Size != 5 {
t.Errorf("msg size mismatch: got %d, want %d", msg.Size, 5)
}
if msg.Code != 8 {
```

```
t.Errorf("msg code mismatch: got %d, want %d", msg.Code, 8)
}
payload, _ := ioutil.ReadAll(msg.Payload)
wantPayload := unhex("C401020304")
if !bytes.Equal(payload, wantPayload) {
t.Errorf("msg payload mismatch:\ngot  %x\nwant %x", payload, wantPayload)
}
}

type fakeHash []byte

func (fakeHash) Write(p []byte) (int, error) { return len(p), nil }
func (fakeHash) Reset()                      {}
func (fakeHash) BlockSize() int              { return 0 }

func (h fakeHash) Size() int           { return len(h) }
func (h fakeHash) Sum(b []byte) []byte { return append(b, h...) }

func TestRLPXFrameRW(t *testing.T) {
var (
aesSecret      = make([]byte, 16)
macSecret      = make([]byte, 16)
egressMACinit  = make([]byte, 32)
ingressMACinit = make([]byte, 32)
)
for _, s := range [][]byte{aesSecret, macSecret, egressMACinit, ingressMACinit} {
rand.Read(s)
}
conn := new(bytes.Buffer)

s1 := secrets{
AES:        aesSecret,
MAC:        macSecret,
EgressMAC:  sha3.NewKeccak256(),
IngressMAC: sha3.NewKeccak256(),
}
s1.EgressMAC.Write(egressMACinit)
s1.IngressMAC.Write(ingressMACinit)
rw1 := newRLPXFrameRW(conn, s1)

s2 := secrets{
AES:        aesSecret,
MAC:        macSecret,
EgressMAC:  sha3.NewKeccak256(),
IngressMAC: sha3.NewKeccak256(),
}
s2.EgressMAC.Write(ingressMACinit)
s2.IngressMAC.Write(egressMACinit)
rw2 := newRLPXFrameRW(conn, s2)

// send some messages
for i := 0; i < 10; i++ {
// write message into conn buffer
wmsg := []interface{}{"foo", "bar", strings.Repeat("test", i)}
err := Send(rw1, uint64(i), wmsg)
if err != nil {
t.Fatalf("WriteMsg error (i=%d): %v", i, err)
}

// read message that rw1 just wrote
msg, err := rw2.ReadMsg()
if err != nil {
t.Fatalf("ReadMsg error (i=%d): %v", i, err)
}
if msg.Code != uint64(i) {
t.Fatalf("msg code mismatch: got %d, want %d", msg.Code, i)
}
payload, _ := ioutil.ReadAll(msg.Payload)
wantPayload, _ := rlp.EncodeToBytes(wmsg)
if !bytes.Equal(payload, wantPayload) {
t.Fatalf("msg payload mismatch:\ngot  %x\nwant %x", payload, wantPayload)
}
}
}

type handshakeAuthTest struct {
input       string
isPlain     bool
wantVersion uint
wantRest    []rlp.RawValue
}

var eip8HandshakeAuthTests = []handshakeAuthTest{
// (Auth) RLPx v4 plain encoding
```

```
{
input: `
048ca79ad18e4b0659fab4853fe5bc58eb83992980f4c9cc147d2aa31532efd29a3d3dc6a3d89eaf
913150cfc777ce0ce4af2758bf4810235f6e6ceccfee1acc6b22c005e9e3a49d6448610a58e98744
ba3ac0399e82692d67c1f58849050b3024e21a52c9d3b01d871ff5f210817912773e610443a9ef14
2e91cdba0bd77b5fdf0769b05671fc35f83d83e4d3b0b000c6b2a1b1bba89e0fc51bf4e460df3105
c444f14be226458940d6061c296350937ffd5e3acaceeaaefd3c6f74be8e23e0f45163cc7ebd7622
0f0128410fd05250273156d548a414444ae2f7dea4dfca2d43c057adb701a715bf59f6fb66b2d1d2
0f2c703f851cbf5ac47396d9ca65b6260bd141ac4d53e2de585a73d1750780db4c9ee4cd4d225173
a4592ee77e2bd94d0be3691f3b406f9bba9b591fc63facc016bfa8
`,
isPlain:     true,
wantVersion: 4,
},
// (Auth) EIP-8 encoding
{
input: `
01b304ab7578555167be8154d5cc456f567d5ba302662433674222360f08d5f1534499d3678b513b
0fca474f3a514b18e75683032eb63fccb16c156dc6eb2c0b1593f0d84ac74f6e475f1b8d56116b84
9634a8c458705bf83a626ea0384d4d7341aae591fae42ce6bd5c850bfe0b999a694a49bbbaf3ef6c
da61110601d3b4c02ab6c30437257a6e0117792631a4b47c1d52fc0f8f89caadeb7d02770bf999cc
147d2df3b62e1ffb2c9d8c125a3984865356266bca11ce7d3a688663a51d82defaa8aad69da39ab6
d5470e81ec5f2a7a47fb865ff7cca21516f9299a07b1bc63ba56c7a1a892112841ca44b6e0034dee
70c9adabc15d76a54f443593fafdc3b27af8059703f88928e199cb122362a4b35f62386da7caad09
c001edaeb5f8a06d2b26fb6cb93c52a9fca51853b68193916982358fe1e5369e249875bb8d0d0ec3
6f917bc5e1eafd5896d46bd61ff23f1a863a8a8dcd54c7b109b771c8e61ec9c8908c733c0263440e
2aa067241aaa433f0bb053c7b31a838504b148f570c0ad62837129e547678c5190341e4f1693956c
3bf7678318e2d5b5340c9e488eefea198576344afbdf66db5f51204a6961a63ce072c8926c
`,
wantVersion: 4,
wantRest:    []rlp.RawValue{},
},
// (Auth) RLPx v4 EIP-8 encoding with version 56, additional list elements
{
input: `
01b8044c6c312173685d1edd268aa95e1d495474c6959bcdd10067ba4c9013df9e40ff45f5bfd6f7
2471f93a91b493f8e00abc4b80f682973de715d77ba3a005a242eb859f9a211d93a347fa64b597bf
280a6b88e26299cf263b01b8dfdb712278464fd1c25840b995e84d367d743f66c0e54a586725b7bb
f12acca27170ae3283c1073adda4b6d79f27656993aefccf16e0d0409fe07db2dc398a1b7e8ee93b
cd181485fd332f381d6a050fba4c7641a5112ac1b0b61168d20f01b479e19adf7fdbfa0905f63352
bfc7e23cf3357657455119d879c78d3cf8c8c06375f3f7d4861aa02a122467e069acaf513025ff19
6641f6d2810ce493f51bee9c966b15c5043505350392b57645385a18c78f14669cc4d960446c1757
```

```
1b7c5d725021babbcd786957f3d17089c084907bda22c2b2675b4378b114c601d858802a55345a1
5
116bc61da4193996187ed70d16730e9ae6b3bb8787ebcaea1871d850997ddc08b4f4ea668fbf3740
7ac044b55be0908ecb94d4ed172ece66fd31bfdadf2b97a8bc690163ee11f5b575a4b44e36e2bfb2
f0fce91676fd64c7773bac6a003f481fddd0bae0a1f31aa27504e2a533af4cef3b623f4791b2cca6
d490
`,
wantVersion: 56,
wantRest:    []rlp.RawValue{{0x01}, {0x02}, {0xC2, 0x04, 0x05}},
},
}

type handshakeAckTest struct {
input       string
wantVersion uint
wantRest    []rlp.RawValue
}

var eip8HandshakeRespTests = []handshakeAckTest{
// (Ack) RLPx v4 plain encoding
{
input: `
049f8abcfa9c0dc65b982e98af921bc0ba6e4243169348a236abe9df5f93aa69d99cadddaa387662
b0ff2c08e9006d5a11a278b1b3331e5aaabf0a32f01281b6f4ede0e09a2d5f585b26513cb794d963
5a57563921c04a9090b4f14ee42be1a5461049af4ea7a7f49bf4c97a352d39c8d02ee4acc416388c
1c66cec761d2bc1c72da6ba143477f049c9d2dde846c252c111b904f630ac98e51609b3b1f58168d
dca6505b7196532e5f85b259a20c45e1979491683fee108e9660edbf38f3add489ae73e3dda2c71b
d1497113d5c755e942d1
`,
wantVersion: 4,
},
// (Ack) EIP-8 encoding
{
input: `
01ea0451958701280a56482929d3b0757da8f7fbe5286784beead59d95089c217c9b917788989470
0
b0e330cc6e4fb383c0340ed85fab836ec9fb8a49672712aeabbdfd1e837c1ff4cace34311cd7f4de
05d59279e3524ab26ef753a0095637ac88f2b499b9914b5f64e143eae548a1066e14cd2f4bd7f814
c4652f11b254f8a2d0191e2f5546fae6055694aed14d906df79ad3b407d94692694e259191cde171
ad542fc588fa2b7333313d82a9f887332f1dfc36cea03f831cb9a23fea05b33deb999e85489e645f
6aab1872475d488d7bd6c7c120caf28dbfc5d6833888155ed69d34dbdc39c1f299be1057810f34fb
e754d021bfca14dc989753d61c413d261934e1a9c67ee060a25eefb54e81a4d14baff922180c395d
```

```
3f998d70f46f6b58306f969627ae364497e73fc27f6d17ae45a413d322cb8814276be6ddd13b885b
201b943213656cde498fa0e9ddc8e0b8f8a53824fbd82254f3e2c17e8eaea009c38b4aa0a3f306e8
797db43c25d68e86f262e564086f59a2fc60511c42abfb3057c247a8a8fe4fb3ccbadde17514b7ac
8000cdb6a912778426260c47f38919a91f25f4b5ffb455d6aaaf150f7e5529c100ce62d6d92826a7
1778d809bdf60232ae21ce8a437eca8223f45ac37f6487452ce626f549b3b5fdee26afd2072e4bc7
5833c2464c805246155289f4
`,
wantVersion: 4,
wantRest:    []rlp.RawValue{},
},
// (Ack) EIP-8 encoding with version 57, additional list elements
{
input: `
01f004076e58aae772bb101ab1a8e64e01ee96e64857ce82b1113817c6cdd52c09d26f7b90981cd
7
ae835aeac72e1573b8a0225dd56d157a010846d888dac7464baf53f2ad4e3d584531fa203658fab0
3a06c9fd5e35737e417bc28c1cbf5e5dfc666de7090f69c3b29754725f84f75382891c561040ea1d
dc0d8f381ed1b9d0d4ad2a0ec021421d847820d6fa0ba66eaf58175f1b235e851c7e2124069fbc20
2888ddb3ac4d56bcbd1b9b7eab59e78f2e2d400905050f4a92dec1c4bdf797b3fc9b2f8e84a482f3
d800386186712dae00d5c386ec9387a5e9c9a1aca5a573ca91082c7d68421f388e79127a5177d4f
8
590237364fd348c9611fa39f78dcdceee3f390f07991b7b47e1daa3ebcb6ccc9607811cb17ce51f1
c8c2c5098dbdd28fca547b3f58c01a424ac05f869f49c6a34672ea2cbbc558428aa1fe48bbfd6115
8b1b735a65d99f21e70dbc020bfdface9f724a0d1fb5895db971cc81aa7608baa0920abb0a565c9c
436e2fd13323428296c86385f2384e408a31e104670df0791d93e743a3a5194ee6b076fb6323ca59
3011b7348c16cf58f66b9633906ba54a2ee803187344b394f75dd2e663a57b956cb830dd7a908d4f
39a2336a61ef9fda549180d4ccde21514d117b6c6fd07a9102b5efe710a32af4eeacae2cb3b1dec0
35b9593b48b9d3ca4c13d245d5f04169b0b1
`,
wantVersion: 57,
wantRest:    []rlp.RawValue{{0x06}, {0xC2, 0x07, 0x08}, {0x81, 0xFA}},
},
}

func TestHandshakeForwardCompatibility(t *testing.T) {
var (
keyA, _       =
crypto.HexToECDSA("49a7b37aa6f6645917e7b807e9d1c00d4fa71f18343b0d4122a4d2df64dd6f
ee")
keyB, _       =
crypto.HexToECDSA("b71c71a67e1177ad4e901695e1b4b9ee17ae16c6668d313eac2f96dbcda3f
291")
```

```
pubA          = crypto.FromECDSAPub(&keyA.PublicKey)[1:]
pubB          = crypto.FromECDSAPub(&keyB.PublicKey)[1:]
ephA, _       =
crypto.HexToECDSA("869d6ecf5211f1cc60418a13b9d870b22959d0c16f02bec714c960dd2298a3
2d")
ephB, _       =
crypto.HexToECDSA("e238eb8e04fee6511ab04c6dd3c89ce097b11f25d584863ac2b6d5b35b184
7e4")
ephPubA       = crypto.FromECDSAPub(&ephA.PublicKey)[1:]
ephPubB       = crypto.FromECDSAPub(&ephB.PublicKey)[1:]
nonceA        =
unhex("7e968bba13b6c50e2c4cd7f241cc0d64d1ac25c7f5952df231ac6a2bda8ee5d6")
nonceB        =
unhex("559aead08264d5795d3909718cdd05abd49572e84fe55590eef31a88a08fdffd")
_, _, _, _    = pubA, pubB, ephPubA, ephPubB
authSignature =
unhex("299ca6acfd35e3d72d8ba3d1e2b60b5561d5af5218eb5bc182045769eb4226910a301acae
3b369fffc4a4899d6b02531e89fd4fe36a2cf0d93607ba470b50f7800")
_             = authSignature
)
makeAuth := func(test handshakeAuthTest) *authMsgV4 {
msg := &authMsgV4{Version: test.wantVersion, Rest: test.wantRest, gotPlain: test.isPlain}
copy(msg.Signature[:], authSignature)
copy(msg.InitiatorPubkey[:], pubA)
copy(msg.Nonce[:], nonceA)
return msg
}
makeAck := func(test handshakeAckTest) *authRespV4 {
msg := &authRespV4{Version: test.wantVersion, Rest: test.wantRest}
copy(msg.RandomPubkey[:], ephPubB)
copy(msg.Nonce[:], nonceB)
return msg
}

// check auth msg parsing
for _, test := range eip8HandshakeAuthTests {
r := bytes.NewReader(unhex(test.input))
msg := new(authMsgV4)
ciphertext, err := readHandshakeMsg(msg, encAuthMsgLen, keyB, r)
if err != nil {
t.Errorf("error for input %x:\n  %v", unhex(test.input), err)
continue
}
if !bytes.Equal(ciphertext, unhex(test.input)) {
t.Errorf("wrong ciphertext for input %x:\n  %x", unhex(test.input), ciphertext)
}
want := makeAuth(test)
if !reflect.DeepEqual(msg, want) {
t.Errorf("wrong msg for input %x:\ngot %s\nwant %s", unhex(test.input), spew.Sdump(msg),
spew.Sdump(want))
}
}

// check auth resp parsing
for _, test := range eip8HandshakeRespTests {
input := unhex(test.input)
r := bytes.NewReader(input)
msg := new(authRespV4)
ciphertext, err := readHandshakeMsg(msg, encAuthRespLen, keyA, r)
if err != nil {
t.Errorf("error for input %x:\n  %v", input, err)
continue
}
if !bytes.Equal(ciphertext, input) {
t.Errorf("wrong ciphertext for input %x:\n  %x", input, err)
}
want := makeAck(test)
if !reflect.DeepEqual(msg, want) {
t.Errorf("wrong msg for input %x:\ngot %s\nwant %s", input, spew.Sdump(msg),
spew.Sdump(want))
}
}

// check derivation for (Auth, Ack) on recipient side
var (
hs = &encHandshake{
initiator:     false,
respNonce:     nonceB,
randomPrivKey: ecies.ImportECDSA(ephB),
}
authCiphertext     = unhex(eip8HandshakeAuthTests[1].input)
authRespCiphertext = unhex(eip8HandshakeRespTests[1].input)
authMsg            = makeAuth(eip8HandshakeAuthTests[1])
wantAES            =
unhex("80e8632c05fed6fc2a13b0f8d31a3cf645366239170ea067065aba8e28bac487")
wantMAC            =
unhex("2ea74ec5dae199227dff1af715362700e989d889d7a493cb0639691efb8e5f98")
wantFooIngressHash =
unhex("0c7ec6340062cc46f5e9f1e3cf86f8c8c403c5a0964f5df0ebd34a75ddc86db5")
)
if err := hs.handleAuthMsg(authMsg, keyB); err != nil {
t.Fatalf("handleAuthMsg: %v", err)
}
derived, err := hs.secrets(authCiphertext, authRespCiphertext)
if err != nil {
t.Fatalf("secrets: %v", err)
}
if !bytes.Equal(derived.AES, wantAES) {
t.Errorf("aes-secret mismatch:\ngot %x\nwant %x", derived.AES, wantAES)
}
if !bytes.Equal(derived.MAC, wantMAC) {
t.Errorf("mac-secret mismatch:\ngot %x\nwant %x", derived.MAC, wantMAC)
}
io.WriteString(derived.IngressMAC, "foo")
fooIngressHash := derived.IngressMAC.Sum(nil)
if !bytes.Equal(fooIngressHash, wantFooIngressHash) {
t.Errorf("ingress-mac('foo') mismatch:\ngot %x\nwant %x", fooIngressHash, wantFooIngressHash)
}
}
```

25:F:\git\coin\ethereum\go-ethereum\p2p\server.go

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

// Package p2p implements the Ethereum p2p network protocols.
package p2p

import (
"crypto/ecdsa"
"errors"
"fmt"
"net"
"sync"
"time"

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

```
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/p2p/discover"
"github.com/ethereum/go-ethereum/p2p/discv5"
"github.com/ethereum/go-ethereum/p2p/nat"
"github.com/ethereum/go-ethereum/p2p/netutil"
)

const (
defaultDialTimeout      = 15 * time.Second
refreshPeersInterval    = 30 * time.Second
staticPeerCheckInterval = 15 * time.Second

// Maximum number of concurrently handshaking inbound connections.
maxAcceptConns = 50

// Maximum number of concurrently dialing outbound connections.
maxActiveDialTasks = 16

// Maximum time allowed for reading a complete message.
// This is effectively the amount of time a connection can be idle.
frameReadTimeout = 30 * time.Second

// Maximum amount of time allowed for writing a complete message.
frameWriteTimeout = 20 * time.Second
)

var errServerStopped = errors.New("server stopped")

// Config holds Server options.
type Config struct {
// This field must be set to a valid secp256k1 private key.
PrivateKey *ecdsa.PrivateKey `toml:"-"`

// MaxPeers is the maximum number of peers that can be
// connected. It must be greater than zero.
MaxPeers int

// MaxPendingPeers is the maximum number of peers that can be pending in the
// handshake phase, counted separately for inbound and outbound connections.
// Zero defaults to preset values.
MaxPendingPeers int `toml:",omitempty"`
```

```
// NoDiscovery can be used to disable the peer discovery mechanism.
// Disabling is useful for protocol debugging (manual topology).
NoDiscovery bool

// DiscoveryV5 specifies whether the the new topic-discovery based V5 discovery
// protocol should be started or not.
DiscoveryV5 bool `toml:",omitempty"`

// Listener address for the V5 discovery protocol UDP traffic.
DiscoveryV5Addr string `toml:",omitempty"`

// Name sets the node name of this server.
// Use common.MakeName to create a name that follows existing conventions.
Name string `toml:"-"`

// BootstrapNodes are used to establish connectivity
// with the rest of the network.
BootstrapNodes []*discover.Node

// BootstrapNodesV5 are used to establish connectivity
// with the rest of the network using the V5 discovery
// protocol.
BootstrapNodesV5 []*discv5.Node `toml:",omitempty"`

// Static nodes are used as pre-configured connections which are always
// maintained and re-connected on disconnects.
StaticNodes []*discover.Node

// Trusted nodes are used as pre-configured connections which are always
// allowed to connect, even above the peer limit.
TrustedNodes []*discover.Node

// Connectivity can be restricted to certain IP networks.
// If this option is set to a non-nil value, only hosts which match one of the
// IP networks contained in the list are considered.
NetRestrict *netutil.Netlist `toml:",omitempty"`

// NodeDatabase is the path to the database containing the previously seen
// live nodes in the network.
NodeDatabase string `toml:",omitempty"`

// Protocols should contain the protocols supported
// by the server. Matching protocols are launched for
// each peer.
Protocols []Protocol `toml:"-"`

// If ListenAddr is set to a non-nil address, the server
// will listen for incoming connections.
//
// If the port is zero, the operating system will pick a port. The
// ListenAddr field will be updated with the actual address when
// the server is started.
ListenAddr string

// If set to a non-nil value, the given NAT port mapper
// is used to make the listening port available to the
// Internet.
NAT nat.Interface `toml:",omitempty"`

// If Dialer is set to a non-nil value, the given Dialer
// is used to dial outbound peer connections.
Dialer *net.Dialer `toml:"-"`

// If NoDial is true, the server will not dial any peers.
NoDial bool `toml:",omitempty"`
}

// Server manages all peer connections.
type Server struct {
// Config fields may not be modified while the server is running.
Config

// Hooks for testing. These are useful because we can inhibit
// the whole protocol stack.
newTransport func(net.Conn) transport
newPeerHook  func(*Peer)

lock    sync.Mutex // protects running
running bool

ntab         discoverTable
listener     net.Listener
ourHandshake *protoHandshake
lastLookup   time.Time
```

```
DiscV5       *discv5.Network

// These are for Peers, PeerCount (and nothing else).
peerOp     chan peerOpFunc
peerOpDone chan struct{}

quit          chan struct{}
addstatic     chan *discover.Node
removestatic  chan *discover.Node
posthandshake chan *conn
addpeer       chan *conn
delpeer       chan peerDrop
loopWG        sync.WaitGroup // loop, listenLoop
}

type peerOpFunc func(map[discover.NodeID]*Peer)

type peerDrop struct {
*Peer
err       error
requested bool // true if signaled by the peer
}

type connFlag int

const (
dynDialedConn connFlag = 1 << iota
staticDialedConn
inboundConn
trustedConn
)

// conn wraps a network connection with information gathered
// during the two handshakes.
type conn struct {
fd net.Conn
transport
flags connFlag
cont  chan error      // The run loop uses cont to signal errors to setupConn.
id    discover.NodeID // valid after the encryption handshake
caps  []Cap           // valid after the protocol handshake
name  string          // valid after the protocol handshake
```

```
}

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

func (c *conn) String() string {
s := c.flags.String()
if (c.id != discover.NodeID{}) {
s += " " + c.id.String()
}
s += " " + c.fd.RemoteAddr().String()
return s
}

func (f connFlag) String() string {
s := ""
if f&trustedConn != 0 {
s += "-trusted"
}
if f&dynDialedConn != 0 {
s += "-dyndial"
}
if f&staticDialedConn != 0 {
s += "-staticdial"
}
if f&inboundConn != 0 {
s += "-inbound"
}
if s != "" {
s = s[1:]
}
```

```
return s
}

func (c *conn) is(f connFlag) bool {
return c.flags&f != 0
}

// Peers returns all connected peers.
func (srv *Server) Peers() []*Peer {
var ps []*Peer
select {
// Note: We'd love to put this function into a variable but
// that seems to cause a weird compiler error in some
// environments.
case srv.peerOp <- func(peers map[discover.NodeID]*Peer) {
for _, p := range peers {
ps = append(ps, p)
}
}:
<-srv.peerOpDone
case <-srv.quit:
}
return ps
}

// PeerCount returns the number of connected peers.
func (srv *Server) PeerCount() int {
var count int
select {
case srv.peerOp <- func(ps map[discover.NodeID]*Peer) { count = len(ps) }:
<-srv.peerOpDone
case <-srv.quit:
}
return count
}

// AddPeer connects to the given node and maintains the connection until the
// server is shut down. If the connection fails for any reason, the server will
// attempt to reconnect the peer.
func (srv *Server) AddPeer(node *discover.Node) {
select {
case srv.addstatic <- node:
```

```
case <-srv.quit:
}
}

// RemovePeer disconnects from the given node
func (srv *Server) RemovePeer(node *discover.Node) {
select {
case srv.removestatic <- node:
case <-srv.quit:
}
}

// Self returns the local node's endpoint information.
func (srv *Server) Self() *discover.Node {
srv.lock.Lock()
defer srv.lock.Unlock()

if !srv.running {
return &discover.Node{IP: net.ParseIP("0.0.0.0")}
}
return srv.makeSelf(srv.listener, srv.ntab)
}

func (srv *Server) makeSelf(listener net.Listener, ntab discoverTable) *discover.Node {
// If the server's not running, return an empty node.
// If the node is running but discovery is off, manually assemble the node infos.
if ntab == nil {
// Inbound connections disabled, use zero address.
if listener == nil {
return &discover.Node{IP: net.ParseIP("0.0.0.0"), ID:
discover.PubkeyID(&srv.PrivateKey.PublicKey)}
}
// Otherwise inject the listener address too
addr := listener.Addr().(*net.TCPAddr)
return &discover.Node{
ID:  discover.PubkeyID(&srv.PrivateKey.PublicKey),
IP:  addr.IP,
TCP: uint16(addr.Port),
}
}
// Otherwise return the discovery node.
return ntab.Self()
```

```
}

// Stop terminates the server and all active peer connections.
// It blocks until all active connections have been closed.
func (srv *Server) Stop() {
srv.lock.Lock()
defer srv.lock.Unlock()
if !srv.running {
return
}
srv.running = false
if srv.listener != nil {
// this unblocks listener Accept
srv.listener.Close()
}
close(srv.quit)
srv.loopWG.Wait()
}

// Start starts running the server.
// Servers can not be re-used after stopping.
func (srv *Server) Start() (err error) {
srv.lock.Lock()
defer srv.lock.Unlock()
if srv.running {
return errors.New("server already running")
}
srv.running = true
log.Info("Starting P2P networking")

// static fields
if srv.PrivateKey == nil {
return fmt.Errorf("Server.PrivateKey must be set to a non-nil key")
}
if srv.newTransport == nil {
srv.newTransport = newRLPX
}
if srv.Dialer == nil {
srv.Dialer = &net.Dialer{Timeout: defaultDialTimeout}
}
srv.quit = make(chan struct{})
srv.addpeer = make(chan *conn)
srv.delpeer = make(chan peerDrop)
srv.posthandshake = make(chan *conn)
srv.addstatic = make(chan *discover.Node)
srv.removestatic = make(chan *discover.Node)
srv.peerOp = make(chan peerOpFunc)
srv.peerOpDone = make(chan struct{})

// node table
if !srv.NoDiscovery {
ntab, err := discover.ListenUDP(srv.PrivateKey, srv.ListenAddr, srv.NAT, srv.NodeDatabase,
srv.NetRestrict)
if err != nil {
return err
}
if err := ntab.SetFallbackNodes(srv.BootstrapNodes); err != nil {
return err
}
srv.ntab = ntab
}

if srv.DiscoveryV5 {
ntab, err := discv5.ListenUDP(srv.PrivateKey, srv.DiscoveryV5Addr, srv.NAT, "", srv.NetRestrict)
//srv.NodeDatabase)
if err != nil {
return err
}
if err := ntab.SetFallbackNodes(srv.BootstrapNodesV5); err != nil {
return err
}
srv.DiscV5 = ntab
}

dynPeers := (srv.MaxPeers + 1) / 2
if srv.NoDiscovery {
dynPeers = 0
}
dialer := newDialState(srv.StaticNodes, srv.BootstrapNodes, srv.ntab, dynPeers, srv.NetRestrict)

// handshake
srv.ourHandshake = &protoHandshake{Version: baseProtocolVersion, Name: srv.Name, ID:
discover.PubkeyID(&srv.PrivateKey.PublicKey)}
for _, p := range srv.Protocols {
```

```
srv.ourHandshake.Caps = append(srv.ourHandshake.Caps, p.cap())
}
// listen/dial
if srv.ListenAddr != "" {
if err := srv.startListening(); err != nil {
return err
}
}
if srv.NoDial && srv.ListenAddr == "" {
log.Warn("P2P server will be useless, neither dialing nor listening")
}

srv.loopWG.Add(1)
go srv.run(dialer)
srv.running = true
return nil
}

func (srv *Server) startListening() error {
// Launch the TCP listener.
listener, err := net.Listen("tcp", srv.ListenAddr)
if err != nil {
return err
}
laddr := listener.Addr().(*net.TCPAddr)
srv.ListenAddr = laddr.String()
srv.listener = listener
srv.loopWG.Add(1)
go srv.listenLoop()
// Map the TCP listening port if NAT is configured.
if !laddr.IP.IsLoopback() && srv.NAT != nil {
srv.loopWG.Add(1)
go func() {
nat.Map(srv.NAT, srv.quit, "tcp", laddr.Port, laddr.Port, "ethereum p2p")
srv.loopWG.Done()
}()
}
return nil
}

type dialer interface {
newTasks(running int, peers map[discover.NodeID]*Peer, now time.Time) []task
```

```
taskDone(task, time.Time)
addStatic(*discover.Node)
removeStatic(*discover.Node)
}

func (srv *Server) run(dialstate dialer) {
defer srv.loopWG.Done()
var (
peers        = make(map[discover.NodeID]*Peer)
trusted      = make(map[discover.NodeID]bool, len(srv.TrustedNodes))
taskdone     = make(chan task, maxActiveDialTasks)
runningTasks []task
queuedTasks  []task // tasks that can't run yet
)
// Put trusted nodes into a map to speed up checks.
// Trusted peers are loaded on startup and cannot be
// modified while the server is running.
for _, n := range srv.TrustedNodes {
trusted[n.ID] = true
}

// removes t from runningTasks
delTask := func(t task) {
for i := range runningTasks {
if runningTasks[i] == t {
runningTasks = append(runningTasks[:i], runningTasks[i+1:]...)
break
}
}
}
// starts until max number of active tasks is satisfied
startTasks := func(ts []task) (rest []task) {
i := 0
for ; len(runningTasks) < maxActiveDialTasks && i < len(ts); i++ {
t := ts[i]
log.Trace("New dial task", "task", t)
go func() { t.Do(srv); taskdone <- t }()
runningTasks = append(runningTasks, t)
}
return ts[i:]
}
scheduleTasks := func() {
```

```
// Start from queue first.
queuedTasks = append(queuedTasks[:0], startTasks(queuedTasks)...)
// Query dialer for new tasks and start as many as possible now.
if len(runningTasks) < maxActiveDialTasks {
nt := dialstate.newTasks(len(runningTasks)+len(queuedTasks), peers, time.Now())
queuedTasks = append(queuedTasks, startTasks(nt)...)
}
}

running:
for {
scheduleTasks()

select {
case <-srv.quit:
// The server was stopped. Run the cleanup logic.
break running
case n := <-srv.addstatic:
// This channel is used by AddPeer to add to the
// ephemeral static peer list. Add it to the dialer,
// it will keep the node connected.
log.Debug("Adding static node", "node", n)
dialstate.addStatic(n)
case n := <-srv.removestatic:
// This channel is used by RemovePeer to send a
// disconnect request to a peer and begin the
// stop keeping the node connected
log.Debug("Removing static node", "node", n)
dialstate.removeStatic(n)
if p, ok := peers[n.ID]; ok {
p.Disconnect(DiscRequested)
}
case op := <-srv.peerOp:
// This channel is used by Peers and PeerCount.
op(peers)
srv.peerOpDone <- struct{}{}
case t := <-taskdone:
// A task got done. Tell dialstate about it so it
// can update its state and remove it from the active
// tasks list.
log.Trace("Dial task done", "task", t)
dialstate.taskDone(t, time.Now())
delTask(t)
case c := <-srv.posthandshake:
// A connection has passed the encryption handshake so
// the remote identity is known (but hasn't been verified yet).
if trusted[c.id] {
// Ensure that the trusted flag is set before checking against MaxPeers.
c.flags |= trustedConn
}
// TODO: track in-progress inbound node IDs (pre-Peer) to avoid dialing them.
c.cont <- srv.encHandshakeChecks(peers, c)
case c := <-srv.addpeer:
// At this point the connection is past the protocol handshake.
// Its capabilities are known and the remote identity is verified.
err := srv.protoHandshakeChecks(peers, c)
if err == nil {
// The handshakes are done and it passed all checks.
p := newPeer(c, srv.Protocols)
name := truncateName(c.name)
log.Debug("Adding p2p peer", "id", c.id, "name", name, "addr", c.fd.RemoteAddr(), "peers",
len(peers)+1)
peers[c.id] = p
go srv.runPeer(p)
}
// The dialer logic relies on the assumption that
// dial tasks complete after the peer has been added or
// discarded. Unblock the task last.
c.cont <- err
case pd := <-srv.delpeer:
// A peer disconnected.
d := common.PrettyDuration(mclock.Now() - pd.created)
pd.log.Debug("Removing p2p peer", "duration", d, "peers", len(peers)-1, "req", pd.requested, "err",
pd.err)
delete(peers, pd.ID())
}
}

log.Trace("P2P networking is spinning down")

// Terminate discovery. If there is a running lookup it will terminate soon.
if srv.ntab != nil {
srv.ntab.Close()
}
```

```
if srv.DiscV5 != nil {
srv.DiscV5.Close()
}
// Disconnect all peers.
for _, p := range peers {
p.Disconnect(DiscQuitting)
}
// Wait for peers to shut down. Pending connections and tasks are
// not handled here and will terminate soon-ish because srv.quit
// is closed.
for len(peers) > 0 {
p := <-srv.delpeer
p.log.Trace("<-delpeer (spindown)", "remainingTasks", len(runningTasks))
delete(peers, p.ID())
}
}

func (srv *Server) protoHandshakeChecks(peers map[discover.NodeID]*Peer, c *conn) error {
// Drop connections with no matching protocols.
if len(srv.Protocols) > 0 && countMatchingProtocols(srv.Protocols, c.caps) == 0 {
return DiscUselessPeer
}
// Repeat the encryption handshake checks because the
// peer set might have changed between the handshakes.
return srv.encHandshakeChecks(peers, c)
}

func (srv *Server) encHandshakeChecks(peers map[discover.NodeID]*Peer, c *conn) error {
switch {
case !c.is(trustedConn|staticDialedConn) && len(peers) >= srv.MaxPeers:
return DiscTooManyPeers
case peers[c.id] != nil:
return DiscAlreadyConnected
case c.id == srv.Self().ID:
return DiscSelf
default:
return nil
}
}

type tempError interface {
Temporary() bool
```

```
}

// listenLoop runs in its own goroutine and accepts
// inbound connections.
func (srv *Server) listenLoop() {
defer srv.loopWG.Done()
log.Info("RLPx listener up", "self", srv.makeSelf(srv.listener, srv.ntab))

// This channel acts as a semaphore limiting
// active inbound connections that are lingering pre-handshake.
// If all slots are taken, no further connections are accepted.
tokens := maxAcceptConns
if srv.MaxPendingPeers > 0 {
tokens = srv.MaxPendingPeers
}
slots := make(chan struct{}, tokens)
for i := 0; i < tokens; i++ {
slots <- struct{}{}
}

for {
// Wait for a handshake slot before accepting.
<-slots

var (
fd  net.Conn
err error
)
for {
fd, err = srv.listener.Accept()
if tempErr, ok := err.(tempError); ok && tempErr.Temporary() {
log.Debug("Temporary read error", "err", err)
continue
} else if err != nil {
log.Debug("Read error", "err", err)
return
}
break
}

// Reject connections that do not match NetRestrict.
if srv.NetRestrict != nil {
```

```
if tcp, ok := fd.RemoteAddr().(*net.TCPAddr); ok && !srv.NetRestrict.Contains(tcp.IP) {
log.Debug("Rejected conn (not whitelisted in NetRestrict)", "addr", fd.RemoteAddr())
fd.Close()
slots <- struct{}{}
continue
}
}

fd = newMeteredConn(fd, true)
log.Trace("Accepted connection", "addr", fd.RemoteAddr())

// Spawn the handler. It will give the slot back when the connection
// has been established.
go func() {
srv.setupConn(fd, inboundConn, nil)
slots <- struct{}{}
}()
}
}

// setupConn runs the handshakes and attempts to add the connection
// as a peer. It returns when the connection has been added as a peer
// or the handshakes have failed.
func (srv *Server) setupConn(fd net.Conn, flags connFlag, dialDest *discover.Node) {
// Prevent leftover pending conns from entering the handshake.
srv.lock.Lock()
running := srv.running
srv.lock.Unlock()
c := &conn{fd: fd, transport: srv.newTransport(fd), flags: flags, cont: make(chan error)}
if !running {
c.close(errServerStopped)
return
}
// Run the encryption handshake.
var err error
if c.id, err = c.doEncHandshake(srv.PrivateKey, dialDest); err != nil {
log.Trace("Failed RLPx handshake", "addr", c.fd.RemoteAddr(), "conn", c.flags, "err", err)
c.close(err)
return
}
clog := log.New("id", c.id, "addr", c.fd.RemoteAddr(), "conn", c.flags)
// For dialed connections, check that the remote public key matches.
```

```
if dialDest != nil && c.id != dialDest.ID {
c.close(DiscUnexpectedIdentity)
clog.Trace("Dialed identity mismatch", "want", c, dialDest.ID)
return
}
if err := srv.checkpoint(c, srv.posthandshake); err != nil {
clog.Trace("Rejected peer before protocol handshake", "err", err)
c.close(err)
return
}
// Run the protocol handshake
phs, err := c.doProtoHandshake(srv.ourHandshake)
if err != nil {
clog.Trace("Failed proto handshake", "err", err)
c.close(err)
return
}
if phs.ID != c.id {
clog.Trace("Wrong devp2p handshake identity", "err", phs.ID)
c.close(DiscUnexpectedIdentity)
return
}
c.caps, c.name = phs.Caps, phs.Name
if err := srv.checkpoint(c, srv.addpeer); err != nil {
clog.Trace("Rejected peer", "err", err)
c.close(err)
return
}
// If the checks completed successfully, runPeer has now been
// launched by run.
}

func truncateName(s string) string {
if len(s) > 20 {
return s[:20] + "..."
}
return s
}

// checkpoint sends the conn to run, which performs the
// post-handshake checks for the stage (posthandshake, addpeer).
func (srv *Server) checkpoint(c *conn, stage chan<- *conn) error {
```

```
select {
case stage <- c:
case <-srv.quit:
return errServerStopped
}
select {
case err := <-c.cont:
return err
case <-srv.quit:
return errServerStopped
}
}

// runPeer runs in its own goroutine for each peer.
// it waits until the Peer logic returns and removes
// the peer.
func (srv *Server) runPeer(p *Peer) {
if srv.newPeerHook != nil {
srv.newPeerHook(p)
}
remoteRequested, err := p.run()
// Note: run waits for existing peers to be sent on srv.delpeer
// before returning, so this send should not select on srv.quit.
srv.delpeer <- peerDrop{p, err, remoteRequested}
}

// NodeInfo represents a short summary of the information known about the host.
type NodeInfo struct {
ID    string `json:"id"`    // Unique node identifier (also the encryption key)
Name  string `json:"name"`  // Name of the node, including client type, version, OS, custom data
Enode string `json:"enode"` // Enode URL for adding this peer from remote peers
IP    string `json:"ip"`    // IP address of the node
Ports struct {
Discovery int `json:"discovery"` // UDP listening port for discovery protocol
Listener  int `json:"listener"`  // TCP listening port for RLPx
} `json:"ports"`
ListenAddr string                 `json:"listenAddr"`
Protocols  map[string]interface{} `json:"protocols"`
}

// NodeInfo gathers and returns a collection of metadata known about the host.
func (srv *Server) NodeInfo() *NodeInfo {
```

```
node := srv.Self()

// Gather and assemble the generic node infos
info := &NodeInfo{
Name:       srv.Name,
Enode:      node.String(),
ID:         node.ID.String(),
IP:         node.IP.String(),
ListenAddr: srv.ListenAddr,
Protocols:  make(map[string]interface{}),
}
info.Ports.Discovery = int(node.UDP)
info.Ports.Listener = int(node.TCP)

// Gather all the running protocol infos (only once per protocol type)
for _, proto := range srv.Protocols {
if _, ok := info.Protocols[proto.Name]; !ok {
nodeInfo := interface{}("unknown")
if query := proto.NodeInfo; query != nil {
nodeInfo = proto.NodeInfo()
}
info.Protocols[proto.Name] = nodeInfo
}
}
return info
}

// PeersInfo returns an array of metadata objects describing connected peers.
func (srv *Server) PeersInfo() []*PeerInfo {
// Gather all the generic and sub-protocol specific infos
infos := make([]*PeerInfo, 0, srv.PeerCount())
for _, peer := range srv.Peers() {
if peer != nil {
infos = append(infos, peer.Info())
}
}
// Sort the result array alphabetically by node identifier
for i := 0; i < len(infos); i++ {
for j := i + 1; j < len(infos); j++ {
if infos[i].ID > infos[j].ID {
infos[i], infos[j] = infos[j], infos[i]
}
```

```
}
}
return infos
}

26:F:\git\coin\ethereum\go-ethereum\p2p\server_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package p2p

import (
"crypto/ecdsa"
"errors"
"math/rand"
"net"
"reflect"
"testing"
"time"

"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/crypto/sha3"
"github.com/ethereum/go-ethereum/p2p/discover"
)

func init() {
// log.Root().SetHandler(log.LvlFilterHandler(log.LvlError, log.StreamHandler(os.Stderr,
log.TerminalFormat(false))))
}

type testTransport struct {
id discover.NodeID
*rlpx

closeErr error
}

func newTestTransport(id discover.NodeID, fd net.Conn) transport {
wrapped := newRLPX(fd).(*rlpx)
wrapped.rw = newRLPXFrameRW(fd, secrets{
MAC:        zero16,
AES:        zero16,
IngressMAC: sha3.NewKeccak256(),
EgressMAC:  sha3.NewKeccak256(),
})
return &testTransport{id: id, rlpx: wrapped}
}

func (c *testTransport) doEncHandshake(prv *ecdsa.PrivateKey, dialDest *discover.Node)
(discover.NodeID, error) {
return c.id, nil
}

func (c *testTransport) doProtoHandshake(our *protoHandshake) (*protoHandshake, error) {
return &protoHandshake{ID: c.id, Name: "test"}, nil
}

func (c *testTransport) close(err error) {
c.rlpx.fd.Close()
c.closeErr = err
}

func startTestServer(t *testing.T, id discover.NodeID, pf func(*Peer)) *Server {
config := Config{
Name:       "test",
MaxPeers:   10,
ListenAddr: "127.0.0.1:0",
PrivateKey: newkey(),
}
server := &Server{
Config:       config,
newPeerHook:  pf,
newTransport: func(fd net.Conn) transport { return newTestTransport(id, fd) },
}
if err := server.Start(); err != nil {
t.Fatalf("Could not start server: %v", err)
}
return server
}

func TestServerListen(t *testing.T) {
// start the test server
connected := make(chan *Peer)
remid := randomID()
srv := startTestServer(t, remid, func(p *Peer) {
```

```
if p.ID() != remid {
t.Error("peer func called with wrong node id")
}
if p == nil {
t.Error("peer func called with nil conn")
}
connected <- p
})
defer close(connected)
defer srv.Stop()

// dial the test server
conn, err := net.DialTimeout("tcp", srv.ListenAddr, 5*time.Second)
if err != nil {
t.Fatalf("could not dial: %v", err)
}
defer conn.Close()

select {
case peer := <-connected:
if peer.LocalAddr().String() != conn.RemoteAddr().String() {
t.Errorf("peer started with wrong conn: got %v, want %v",
peer.LocalAddr(), conn.RemoteAddr())
}
peers := srv.Peers()
if !reflect.DeepEqual(peers, []*Peer{peer}) {
t.Errorf("Peers mismatch: got %v, want %v", peers, []*Peer{peer})
}
case <-time.After(1 * time.Second):
t.Error("server did not accept within one second")
}
}

func TestServerDial(t *testing.T) {
// run a one-shot TCP server to handle the connection.
listener, err := net.Listen("tcp", "127.0.0.1:0")
if err != nil {
t.Fatalf("could not setup listener: %v", err)
}
defer listener.Close()
accepted := make(chan net.Conn)
go func() {
```

```
conn, err := listener.Accept()
if err != nil {
t.Error("accept error:", err)
return
}
accepted <- conn
}()

// start the server
connected := make(chan *Peer)
remid := randomID()
srv := startTestServer(t, remid, func(p *Peer) { connected <- p })
defer close(connected)
defer srv.Stop()

// tell the server to connect
tcpAddr := listener.Addr().(*net.TCPAddr)
srv.AddPeer(&discover.Node{ID: remid, IP: tcpAddr.IP, TCP: uint16(tcpAddr.Port)})

select {
case conn := <-accepted:
defer conn.Close()

select {
case peer := <-connected:
if peer.ID() != remid {
t.Errorf("peer has wrong id")
}
if peer.Name() != "test" {
t.Errorf("peer has wrong name")
}
if peer.RemoteAddr().String() != conn.LocalAddr().String() {
t.Errorf("peer started with wrong conn: got %v, want %v",
peer.RemoteAddr(), conn.LocalAddr())
}
peers := srv.Peers()
if !reflect.DeepEqual(peers, []*Peer{peer}) {
t.Errorf("Peers mismatch: got %v, want %v", peers, []*Peer{peer})
}
case <-time.After(1 * time.Second):
t.Error("server did not launch peer within one second")
}
```

```
case <-time.After(1 * time.Second):
t.Error("server did not connect within one second")
}
}

// This test checks that tasks generated by dialstate are
// actually executed and taskdone is called for them.
func TestServerTaskScheduling(t *testing.T) {
var (
done           = make(chan *testTask)
quit, returned = make(chan struct{}), make(chan struct{})
tc             = 0
tg             = taskgen{
newFunc: func(running int, peers map[discover.NodeID]*Peer) []task {
tc++
return []task{&testTask{index: tc - 1}}
},
doneFunc: func(t task) {
select {
case done <- t.(*testTask):
case <-quit:
}
},
}
)

// The Server in this test isn't actually running
// because we're only interested in what run does.
srv := &Server{
Config:  Config{MaxPeers: 10},
quit:    make(chan struct{}),
ntab:    fakeTable{},
running: true,
}
srv.loopWG.Add(1)
go func() {
srv.run(tg)
close(returned)
}()

var gotdone []*testTask
```

```
for i := 0; i < 100; i++ {
gotdone = append(gotdone, <-done)
}
for i, task := range gotdone {
if task.index != i {
t.Errorf("task %d has wrong index, got %d", i, task.index)
break
}
if !task.called {
t.Errorf("task %d was not called", i)
break
}
}

close(quit)
srv.Stop()
select {
case <-returned:
case <-time.After(500 * time.Millisecond):
t.Error("Server.run did not return within 500ms")
}
}

// This test checks that Server doesn't drop tasks,
// even if newTasks returns more than the maximum number of tasks.
func TestServerManyTasks(t *testing.T) {
alltasks := make([]task, 300)
for i := range alltasks {
alltasks[i] = &testTask{index: i}
}

var (
srv        = &Server{quit: make(chan struct{}), ntab: fakeTable{}, running: true}
done       = make(chan *testTask)
start, end = 0, 0
)
defer srv.Stop()
srv.loopWG.Add(1)
go srv.run(taskgen{
newFunc: func(running int, peers map[discover.NodeID]*Peer) []task {
start, end = end, end+maxActiveDialTasks+10
if end > len(alltasks) {
```

```
end = len(alltasks)
}
return alltasks[start:end]
},
doneFunc: func(tt task) {
done <- tt.(*testTask)
},
})

doneset := make(map[int]bool)
timeout := time.After(2 * time.Second)
for len(doneset) < len(alltasks) {
select {
case tt := <-done:
if doneset[tt.index] {
t.Errorf("task %d got done more than once", tt.index)
} else {
doneset[tt.index] = true
}
case <-timeout:
t.Errorf("%d of %d tasks got done within 2s", len(doneset), len(alltasks))
for i := 0; i < len(alltasks); i++ {
if !doneset[i] {
t.Logf("task %d not done", i)
}
}
return
}
}
}

type taskgen struct {
newFunc  func(running int, peers map[discover.NodeID]*Peer) []task
doneFunc func(task)
}

func (tg taskgen) newTasks(running int, peers map[discover.NodeID]*Peer, now time.Time) []task
{
return tg.newFunc(running, peers)
}
func (tg taskgen) taskDone(t task, now time.Time) {
tg.doneFunc(t)
```

```
}
func (tg taskgen) addStatic(*discover.Node) {
}
func (tg taskgen) removeStatic(*discover.Node) {
}

type testTask struct {
index  int
called bool
}

func (t *testTask) Do(srv *Server) {
t.called = true
}

// This test checks that connections are disconnected
// just after the encryption handshake when the server is
// at capacity. Trusted connections should still be accepted.
func TestServerAtCap(t *testing.T) {
trustedID := randomID()
srv := &Server{
Config: Config{
PrivateKey:   newkey(),
MaxPeers:     10,
NoDial:       true,
TrustedNodes: []*discover.Node{{ID: trustedID}},
},
}
if err := srv.Start(); err != nil {
t.Fatalf("could not start: %v", err)
}
defer srv.Stop()

newconn := func(id discover.NodeID) *conn {
fd, _ := net.Pipe()
tx := newTestTransport(id, fd)
return &conn{fd: fd, transport: tx, flags: inboundConn, id: id, cont: make(chan error)}
}

// Inject a few connections to fill up the peer set.
for i := 0; i < 10; i++ {
c := newconn(randomID())
```

```
if err := srv.checkpoint(c, srv.addpeer); err != nil {
t.Fatalf("could not add conn %d: %v", i, err)
}
}
// Try inserting a non-trusted connection.
c := newconn(randomID())
if err := srv.checkpoint(c, srv.posthandshake); err != DiscTooManyPeers {
t.Error("wrong error for insert:", err)
}
// Try inserting a trusted connection.
c = newconn(trustedID)
if err := srv.checkpoint(c, srv.posthandshake); err != nil {
t.Error("unexpected error for trusted conn @posthandshake:", err)
}
if !c.is(trustedConn) {
t.Error("Server did not set trusted flag")
}

}

func TestServerSetupConn(t *testing.T) {
id := randomID()
srvkey := newkey()
srvid := discover.PubkeyID(&srvkey.PublicKey)
tests := []struct {
dontstart bool
tt        *setupTransport
flags     connFlag
dialDest  *discover.Node

wantCloseErr error
wantCalls    string
}{
{
dontstart:    true,
tt:           &setupTransport{id: id},
wantCalls:    "close,",
wantCloseErr: errServerStopped,
},
{
tt:           &setupTransport{id: id, encHandshakeErr: errors.New("read error")},
flags:        inboundConn,
wantCalls:    "doEncHandshake,close,",
wantCloseErr: errors.New("read error"),
},
{
tt:           &setupTransport{id: id},
dialDest:     &discover.Node{ID: randomID()},
flags:        dynDialedConn,
wantCalls:    "doEncHandshake,close,",
wantCloseErr: DiscUnexpectedIdentity,
},
{
tt:           &setupTransport{id: id, phs: &protoHandshake{ID: randomID()}},
dialDest:     &discover.Node{ID: id},
flags:        dynDialedConn,
wantCalls:    "doEncHandshake,doProtoHandshake,close,",
wantCloseErr: DiscUnexpectedIdentity,
},
{
tt:           &setupTransport{id: id, protoHandshakeErr: errors.New("foo")},
dialDest:     &discover.Node{ID: id},
flags:        dynDialedConn,
wantCalls:    "doEncHandshake,doProtoHandshake,close,",
wantCloseErr: errors.New("foo"),
},
{
tt:           &setupTransport{id: srvid, phs: &protoHandshake{ID: srvid}},
flags:        inboundConn,
wantCalls:    "doEncHandshake,close,",
wantCloseErr: DiscSelf,
},
{
tt:           &setupTransport{id: id, phs: &protoHandshake{ID: id}},
flags:        inboundConn,
wantCalls:    "doEncHandshake,doProtoHandshake,close,",
wantCloseErr: DiscUselessPeer,
},
}

for i, test := range tests {
srv := &Server{
Config: Config{
PrivateKey: srvkey,
MaxPeers:   10,
NoDial:     true,
Protocols:  []Protocol{discard},
},
newTransport: func(fd net.Conn) transport { return test.tt },
}
if !test.dontstart {
if err := srv.Start(); err != nil {
t.Fatalf("couldn't start server: %v", err)
}
}
p1, _ := net.Pipe()
srv.setupConn(p1, test.flags, test.dialDest)
if !reflect.DeepEqual(test.tt.closeErr, test.wantCloseErr) {
t.Errorf("test %d: close error mismatch: got %q, want %q", i, test.tt.closeErr, test.wantCloseErr)
}
if test.tt.calls != test.wantCalls {
t.Errorf("test %d: calls mismatch: got %q, want %q", i, test.tt.calls, test.wantCalls)
}
}
}

type setupTransport struct {
id              discover.NodeID
encHandshakeErr error

phs               *protoHandshake
protoHandshakeErr error

calls    string
closeErr error
}

func (c *setupTransport) doEncHandshake(prv *ecdsa.PrivateKey, dialDest *discover.Node)
(discover.NodeID, error) {
c.calls += "doEncHandshake,"
return c.id, c.encHandshakeErr
}
func (c *setupTransport) doProtoHandshake(our *protoHandshake) (*protoHandshake, error) {
c.calls += "doProtoHandshake,"
if c.protoHandshakeErr != nil {
return nil, c.protoHandshakeErr
```

```
}
return c.phs, nil
}
func (c *setupTransport) close(err error) {
c.calls += "close,"
c.closeErr = err
}

// setupConn shouldn't write to/read from the connection.
func (c *setupTransport) WriteMsg(Msg) error {
panic("WriteMsg called on setupTransport")
}
func (c *setupTransport) ReadMsg() (Msg, error) {
panic("ReadMsg called on setupTransport")
}

func newkey() *ecdsa.PrivateKey {
key, err := crypto.GenerateKey()
if err != nil {
panic("couldn't generate key: " + err.Error())
}
return key
}

func randomID() (id discover.NodeID) {
for i := range id {
id[i] = byte(rand.Intn(255))
}
return id
}
```

27:F:\git\coin\ethereum\go-ethereum\params\bootnodes.go

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

package params

// MainnetBootnodes are the enode URLs of the P2P bootstrap nodes running on
// the main Ethereum network.
var MainnetBootnodes = []string{

// Ethereum Foundation Go Bootnodes
"enode://a979fb575495b8d6db44f750317d0f4622bf4c2aa3365d6af7c284339968eef29b69ad0dce
72a4d8db5ebb4968de0e3bec910127f134779fbcb0cb6d3331163c@52.16.188.185:30303", // IE
"enode://3f1d12044546b76342d59d4a05532c14b85aa669704bfe1f864fe079415aa2c02d743e032
18e57a33fb94523adb54032871a6c51b2cc5514cb7c7e35b3ed0a99@13.93.211.84:30303",  //
US-WEST
"enode://78de8a0916848093c73790ead81d1928bec737d565119932b98c6b100d944b7a95e94f84
7f689fc723399d2e31129d182f7ef3863f2b4c820abbf3ab2722344d@191.235.84.50:30303", // BR
"enode://158f8aab45f6d19c6cbf4a089c2670541a8da11978a2f90dbf6a502a4a3bab80d288afdbeb
7ec0ef6d92de563767f3b1ea9e8e334ca711e9f8e2df5a0385e8e6@13.75.154.138:30303", // AU
"enode://1118980bf48b0a3640bdba04e0fe78b1add18e1cd99bf22d53daac1fd9972ad650df52176
e7c7d89d1114cfef2bc23a2959aa54998a46afcf7d91809f0855082@52.74.57.123:30303",  // SG

// Ethereum Foundation Cpp Bootnodes
"enode://979b7fa28feeb35a4741660a16076f1943202cb72b6af70d327f053e248bab9ba81760f39d
0701ef1d8f89cc1fbd2cacba0710a12cd5314d5e0c9021aa3637f9@5.1.83.226:30303", // DE

}

// TestnetBootnodes are the enode URLs of the P2P bootstrap nodes running on the
// Ropsten test network.
var TestnetBootnodes = []string{
"enode://6ce05930c72abc632c58e2e4324f7c7ea478cec0ed4fa2528982cf34483094e9cbc9216e7
aa349691242576d552a2a56aaeae426c5303ded677ce455ba1acd9d@13.84.180.240:30303", //
US-TX
"enode://20c9ad97c081d63397d7b685a412227a40e23c8bdc6688c6f37e97cfbc22d2b4d1db1510
d8f61e6a8866ad7f0e17c02b14182d37ea7c3c8b9c2683aeb6b733a1@52.169.14.227:30303", // IE
}

// RinkebyBootnodes are the enode URLs of the P2P bootstrap nodes running on the
// Rinkeby test network.
var RinkebyBootnodes = []string{
"enode://a24ac7c5484ef4ed0c5eb2d36620ba4e4aa13b8c84684e1b4aab0cebea2ae45cb4d375b7
7eab56516d34bfbd3c1a833fc51296ff084b770b94fb9028c4d25ccf@52.169.42.101:30303", // IE
}

// RinkebyV5Bootnodes are the enode URLs of the P2P bootstrap nodes running on the
// Rinkeby test network for the experimental RLPx v5 topic-discovery network.
var RinkebyV5Bootnodes = []string{
"enode://a24ac7c5484ef4ed0c5eb2d36620ba4e4aa13b8c84684e1b4aab0cebea2ae45cb4d375b7
7eab56516d34bfbd3c1a833fc51296ff084b770b94fb9028c4d25ccf@52.169.42.101:30303?discpor
t=30304", // IE
}
```

```
// DiscoveryV5Bootnodes are the enode URLs of the P2P bootstrap nodes for the
// experimental RLPx v5 topic-discovery network.
var DiscoveryV5Bootnodes = []string{
"enode://0cc5f5ffb5d9098c8b8c62325f3797f56509bff942704687b6530992ac706e2cb946b90a34f
1f19548cd3c7baccbcaea354531e5983c7d1bc0dee16ce4b6440b@40.118.3.223:30305",
"enode://1c7a64d76c0334b0418c004af2f67c50e36a3be60b5e4790bdac0439d21603469a85fad36
f2473c9a80eb043ae60936df905fa28f1ff614c3e5dc34f15dcd2dc@40.118.3.223:30308",
"enode://85c85d7143ae8bb96924f2b54f1b3e70d8c4d367af305325d30a61385a432f247d2c75c45
c6b4a60335060d072d7f5b35dd1d4c45f76941f62a4f83b6e75daaf@40.118.3.223:30309",
}

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

package params

import (
"fmt"
"math"
"math/big"

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

var (
MainnetGenesisHash =
common.HexToHash("0xd4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb
8fa3") // Mainnet genesis hash to enforce below configs on
TestnetGenesisHash =
common.HexToHash("0x41941023680923e0fe4d74a34bdac8141f2540e3ae90623718e47d66d1c
a4a2d") // Testnet genesis hash to enforce below configs on
)

var (
// MainnetChainConfig is the chain parameters to run a node on the main network.
MainnetChainConfig = &ChainConfig{
ChainId:         big.NewInt(1),
HomesteadBlock:  big.NewInt(1150000),
DAOForkBlock:    big.NewInt(1920000),
DAOForkSupport:  true,
EIP150Block:     big.NewInt(2463000),
EIP150Hash:
```

```
common.HexToHash("0x2086799aeebeae135c246c65021c82b4e15a2c451340993aacfd2751886
514f0"),
EIP155Block:     big.NewInt(2675000),
EIP158Block:     big.NewInt(2675000),
MetropolisBlock: big.NewInt(math.MaxInt64), // Don't enable yet

Ethash: new(EthashConfig),
}

// TestnetChainConfig contains the chain parameters to run a node on the Ropsten test network.
TestnetChainConfig = &ChainConfig{
ChainId:         big.NewInt(3),
HomesteadBlock:  big.NewInt(0),
DAOForkBlock:    nil,
DAOForkSupport:  true,
EIP150Block:     big.NewInt(0),
EIP150Hash:
common.HexToHash("0x41941023680923e0fe4d74a34bdac8141f2540e3ae90623718e47d66d1c
a4a2d"),
EIP155Block:     big.NewInt(10),
EIP158Block:     big.NewInt(10),
MetropolisBlock: big.NewInt(math.MaxInt64), // Don't enable yet

Ethash: new(EthashConfig),
}

// RinkebyChainConfig contains the chain parameters to run a node on the Rinkeby test network.
RinkebyChainConfig = &ChainConfig{
ChainId:         big.NewInt(4),
HomesteadBlock:  big.NewInt(1),
DAOForkBlock:    nil,
DAOForkSupport:  true,
EIP150Block:     big.NewInt(2),
EIP150Hash:
common.HexToHash("0x9b095b36c15eaf13044373aef8ee0bd3a382a5abb92e402afa44b8249c3a
90e9"),
EIP155Block:     big.NewInt(3),
EIP158Block:     big.NewInt(3),
MetropolisBlock: big.NewInt(math.MaxInt64), // Don't enable yet

Clique: &CliqueConfig{
Period: 15,
Epoch:  30000,
},
}

// AllProtocolChanges contains every protocol change (EIPs)
// introduced and accepted by the Ethereum core developers.
//
// This configuration is intentionally not using keyed fields.
// This configuration must *always* have all forks enabled, which
// means that all fields must be set at all times. This forces
// anyone adding flags to the config to also have to set these
// fields.
AllProtocolChanges = &ChainConfig{big.NewInt(1337), big.NewInt(0), nil, false, big.NewInt(0),
common.Hash{}, big.NewInt(0), big.NewInt(0), big.NewInt(math.MaxInt64) /*disabled*/,
new(EthashConfig), nil}
TestChainConfig    = &ChainConfig{big.NewInt(1), big.NewInt(0), nil, false, big.NewInt(0),
common.Hash{}, big.NewInt(0), big.NewInt(0), nil, new(EthashConfig), nil}
TestRules          = TestChainConfig.Rules(new(big.Int))
)

// ChainConfig is the core config which determines the blockchain settings.
//
// ChainConfig is stored in the database on a per block basis. This means
// that any network, identified by its genesis block, can have its own
// set of configuration options.
type ChainConfig struct {
ChainId *big.Int `json:"chainId"` // Chain id identifies the current chain and is used for replay
protection

HomesteadBlock *big.Int `json:"homesteadBlock,omitempty"` // Homestead switch block (nil = no
fork, 0 = already homestead)
DAOForkBlock   *big.Int `json:"daoForkBlock,omitempty"`   // TheDAO hard-fork switch block (nil =
no fork)
DAOForkSupport bool     `json:"daoForkSupport,omitempty"` // Whether the nodes supports or
opposes the DAO hard-fork

// EIP150 implements the Gas price changes (https://github.com/ethereum/EIPs/issues/150)
EIP150Block *big.Int    `json:"eip150Block,omitempty"` // EIP150 HF block (nil = no fork)
EIP150Hash  common.Hash `json:"eip150Hash,omitempty"`  // EIP150 HF hash (fast sync aid)

EIP155Block *big.Int `json:"eip155Block,omitempty"` // EIP155 HF block
EIP158Block *big.Int `json:"eip158Block,omitempty"` // EIP158 HF block
```

```
MetropolisBlock *big.Int `json:"metropolisBlock,omitempty"` // Metropolis switch block (nil = no
fork, 0 = alraedy on homestead)

// Various consensus engines
Ethash *EthashConfig `json:"ethash,omitempty"`
Clique *CliqueConfig `json:"clique,omitempty"`
}

// EthashConfig is the consensus engine configs for proof-of-work based sealing.
type EthashConfig struct{}

// String implements the stringer interface, returning the consensus engine details.
func (c *EthashConfig) String() string {
return "ethash"
}

// CliqueConfig is the consensus engine configs for proof-of-authority based sealing.
type CliqueConfig struct {
Period uint64 `json:"period"` // Number of seconds between blocks to enforce
Epoch  uint64 `json:"epoch"`  // Epoch length to reset votes and checkpoint
}

// String implements the stringer interface, returning the consensus engine details.
func (c *CliqueConfig) String() string {
return "clique"
}

// String implements the fmt.Stringer interface.
func (c *ChainConfig) String() string {
var engine interface{}
switch {
case c.Ethash != nil:
engine = c.Ethash
case c.Clique != nil:
engine = c.Clique
default:
engine = "unknown"
}
return fmt.Sprintf("{ChainID: %v Homestead: %v DAO: %v DAOSupport: %v EIP150: %v EIP155:
%v EIP158: %v Metropolis: %v Engine: %v}",
c.ChainId,
c.HomesteadBlock,
c.DAOForkBlock,
c.DAOForkSupport,
c.EIP150Block,
c.EIP155Block,
c.EIP158Block,
c.MetropolisBlock,
engine,
)
}

// IsHomestead returns whether num is either equal to the homestead block or greater.
func (c *ChainConfig) IsHomestead(num *big.Int) bool {
return isForked(c.HomesteadBlock, num)
}

// IsDAO returns whether num is either equal to the DAO fork block or greater.
func (c *ChainConfig) IsDAOFork(num *big.Int) bool {
return isForked(c.DAOForkBlock, num)
}

func (c *ChainConfig) IsEIP150(num *big.Int) bool {
return isForked(c.EIP150Block, num)
}

func (c *ChainConfig) IsEIP155(num *big.Int) bool {
return isForked(c.EIP155Block, num)
}

func (c *ChainConfig) IsEIP158(num *big.Int) bool {
return isForked(c.EIP158Block, num)
}

func (c *ChainConfig) IsMetropolis(num *big.Int) bool {
return isForked(c.MetropolisBlock, num)
}

// GasTable returns the gas table corresponding to the current phase (homestead or homestead
reprice).
//
// The returned GasTable's fields shouldn't, under any circumstances, be changed.
func (c *ChainConfig) GasTable(num *big.Int) GasTable {
```

```
if num == nil {
return GasTableHomestead
}
switch {
case c.IsEIP158(num):
return GasTableEIP158
case c.IsEIP150(num):
return GasTableHomesteadGasRepriceFork
default:
return GasTableHomestead
}
}

// CheckCompatible checks whether scheduled fork transitions have been imported
// with a mismatching chain configuration.
func (c *ChainConfig) CheckCompatible(newcfg *ChainConfig, height uint64) *ConfigCompatError
{
bhead := new(big.Int).SetUint64(height)

// Iterate checkCompatible to find the lowest conflict.
var lasterr *ConfigCompatError
for {
err := c.checkCompatible(newcfg, bhead)
if err == nil || (lasterr != nil && err.RewindTo == lasterr.RewindTo) {
break
}
lasterr = err
bhead.SetUint64(err.RewindTo)
}
return lasterr
}

func (c *ChainConfig) checkCompatible(newcfg *ChainConfig, head *big.Int) *ConfigCompatError {
if isForkIncompatible(c.HomesteadBlock, newcfg.HomesteadBlock, head) {
return newCompatError("Homestead fork block", c.HomesteadBlock, newcfg.HomesteadBlock)
}
if isForkIncompatible(c.DAOForkBlock, newcfg.DAOForkBlock, head) {
return newCompatError("DAO fork block", c.DAOForkBlock, newcfg.DAOForkBlock)
}
if c.IsDAOFork(head) && c.DAOForkSupport != newcfg.DAOForkSupport {
return newCompatError("DAO fork support flag", c.DAOForkBlock, newcfg.DAOForkBlock)
}
```

```
if isForkIncompatible(c.EIP150Block, newcfg.EIP150Block, head) {
return newCompatError("EIP150 fork block", c.EIP150Block, newcfg.EIP150Block)
}
if isForkIncompatible(c.EIP155Block, newcfg.EIP155Block, head) {
return newCompatError("EIP155 fork block", c.EIP155Block, newcfg.EIP155Block)
}
if isForkIncompatible(c.EIP158Block, newcfg.EIP158Block, head) {
return newCompatError("EIP158 fork block", c.EIP158Block, newcfg.EIP158Block)
}
if c.IsEIP158(head) && !configNumEqual(c.ChainId, newcfg.ChainId) {
return newCompatError("EIP158 chain ID", c.EIP158Block, newcfg.EIP158Block)
}
if isForkIncompatible(c.MetropolisBlock, newcfg.MetropolisBlock, head) {
return newCompatError("Metropolis fork block", c.MetropolisBlock, newcfg.MetropolisBlock)
}
return nil
}

// isForkIncompatible returns true if a fork scheduled at s1 cannot be rescheduled to
// block s2 because head is already past the fork.
func isForkIncompatible(s1, s2, head *big.Int) bool {
return (isForked(s1, head) || isForked(s2, head)) && !configNumEqual(s1, s2)
}

// isForked returns whether a fork scheduled at block s is active at the given head block.
func isForked(s, head *big.Int) bool {
if s == nil || head == nil {
return false
}
return s.Cmp(head) <= 0
}

func configNumEqual(x, y *big.Int) bool {
if x == nil {
return y == nil
}
if y == nil {
return x == nil
}
return x.Cmp(y) == 0
}
```

```
// ConfigCompatError is raised if the locally-stored blockchain is initialised with a
// ChainConfig that would alter the past.
type ConfigCompatError struct {
What string
// block numbers of the stored and new configurations
StoredConfig, NewConfig *big.Int
// the block number to which the local chain must be rewound to correct the error
RewindTo uint64
}

func newCompatError(what string, storedblock, newblock *big.Int) *ConfigCompatError {
var rew *big.Int
switch {
case storedblock == nil:
rew = newblock
case newblock == nil || storedblock.Cmp(newblock) < 0:
rew = storedblock
default:
rew = newblock
}
err := &ConfigCompatError{what, storedblock, newblock, 0}
if rew != nil && rew.Sign() > 0 {
err.RewindTo = rew.Uint64() - 1
}
return err
}

func (err *ConfigCompatError) Error() string {
return fmt.Sprintf("mismatching %s in database (have %d, want %d, rewindto %d)", err.What,
err.StoredConfig, err.NewConfig, err.RewindTo)
}

// Rules wraps ChainConfig and is merely syntatic sugar or can be used for functions
// that do not have or require information about the block.
//
// Rules is a one time interface meaning that it shouldn't be used in between transition
// phases.
type Rules struct {
ChainId                                    *big.Int
IsHomestead, IsEIP150, IsEIP155, IsEIP158 bool
IsMetropolis                               bool
}
```

```
func (c *ChainConfig) Rules(num *big.Int) Rules {
chainId := c.ChainId
if chainId == nil {
chainId = new(big.Int)
}
return Rules{ChainId: new(big.Int).Set(chainId), IsHomestead: c.IsHomestead(num), IsEIP150:
c.IsEIP150(num), IsEIP155: c.IsEIP155(num), IsEIP158: c.IsEIP158(num), IsMetropolis:
c.IsMetropolis(num)}
}
```

29:F:\git\coin\ethereum\go-ethereum\params\config_test.go

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

package params

import (
"math/big"
"reflect"
"testing"
)

func TestCheckCompatible(t *testing.T) {
type test struct {
stored, new *ChainConfig
head        uint64
wantErr     *ConfigCompatError
}
tests := []test{
{stored: AllProtocolChanges, new: AllProtocolChanges, head: 0, wantErr: nil},
{stored: AllProtocolChanges, new: AllProtocolChanges, head: 100, wantErr: nil},
{
stored:  &ChainConfig{EIP150Block: big.NewInt(10)},
new:     &ChainConfig{EIP150Block: big.NewInt(20)},
head:    9,
wantErr: nil,
},
{
stored: AllProtocolChanges,
new:    &ChainConfig{HomesteadBlock: nil},
head:   3,
wantErr: &ConfigCompatError{
```

```
What:         "Homestead fork block",
StoredConfig: big.NewInt(0),
NewConfig:    nil,
RewindTo:     0,
},
},
{
stored: AllProtocolChanges,
new:    &ChainConfig{HomesteadBlock: big.NewInt(1)},
head:   3,
wantErr: &ConfigCompatError{
What:         "Homestead fork block",
StoredConfig: big.NewInt(0),
NewConfig:    big.NewInt(1),
RewindTo:     0,
},
},
{
stored: &ChainConfig{HomesteadBlock: big.NewInt(30), EIP150Block: big.NewInt(10)},
new:    &ChainConfig{HomesteadBlock: big.NewInt(25), EIP150Block: big.NewInt(20)},
head:   25,
wantErr: &ConfigCompatError{
What:         "EIP150 fork block",
StoredConfig: big.NewInt(10),
NewConfig:    big.NewInt(20),
RewindTo:     9,
},
},
}

for _, test := range tests {
err := test.stored.CheckCompatible(test.new, test.head)
if !reflect.DeepEqual(err, test.wantErr) {
t.Errorf("error mismatch:\nstored: %v\nnew: %v\nhead: %v\nerr: %v\nwant: %v", test.stored,
test.new, test.head, err, test.wantErr)
}
}
}
```

30:F:\git\coin\ethereum\go-ethereum\params\dao.go

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

```
package params

import (
"math/big"

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

// DAOForkBlockExtra is the block header extra-data field to set for the DAO fork
// point and a number of consecutive blocks to allow fast/light syncers to correctly
// pick the side they want  ("dao-hard-fork").
var DAOForkBlockExtra = common.FromHex("0x64616f2d686172642d666f726b")

// DAOForkExtraRange is the number of consecutive blocks from the DAO fork point
// to override the extra-data in to prevent no-fork attacks.
var DAOForkExtraRange = big.NewInt(10)

// DAORefundContract is the address of the refund contract to send DAO balances to.
var DAORefundContract =
common.HexToAddress("0xbf4ed7b27f1d666546e30d74d50d173d20bca754")

// DAODrainList is the list of accounts whose full balances will be moved into a
// refund contract at the beginning of the dao-fork block.
func DAODrainList() []common.Address {
return []common.Address{
common.HexToAddress("0xd4fe7bc31cedb7bfb8a345f31e668033056b2728"),
common.HexToAddress("0xb3fb0e5aba0e20e5c49d252dfd30e102b171a425"),
common.HexToAddress("0x2c19c7f9ae8b751e37aeb2d93a699722395ae18f"),
common.HexToAddress("0xecd135fa4f61a655311e86238c92adcd779555d2"),
common.HexToAddress("0x1975bd06d486162d5dc297798dfc41edd5d160a7"),
common.HexToAddress("0xa3acf3a1e16b1d7c315e23510fdd7847b48234f6"),
common.HexToAddress("0x319f70bab6845585f412ec7724b744fec6095c85"),
common.HexToAddress("0x06706dd3f2c9abf0a21ddcc6941d9b86f0596936"),
common.HexToAddress("0x5c8536898fbb74fc7445814902fd08422eac56d0"),
common.HexToAddress("0x6966ab0d485353095148a2155858910e0965b6f9"),
common.HexToAddress("0x779543a0491a837ca36ce8c635d6154e3c4911a6"),
common.HexToAddress("0x2a5ed960395e2a49b1c758cef4aa15213cfd874c"),
common.HexToAddress("0x5c6e67ccd5849c0d29219c4f95f1a7a93b3f5dc5"),
common.HexToAddress("0x9c50426be05db97f5d64fc54bf89eff947f0a321"),
common.HexToAddress("0x200450f06520bdd6c527622a273333384d870efb"),
common.HexToAddress("0xbe8539bfe837b67d1282b2b1d61c3f723966f049"),
common.HexToAddress("0x6b0c4d41ba9ab8d8cfb5d379c69a612f2ced8ecb"),
common.HexToAddress("0xf1385fb24aad0cd7432824085e42aff90886fef5"),
common.HexToAddress("0xd1ac8b1ef1b69ff51d1d401a476e7e612414f091"),
common.HexToAddress("0x8163e7fb499e90f8544ea62bbf80d21cd26d9efd"),
common.HexToAddress("0x51e0ddd9998364a2eb38588679f0d2c42653e4a6"),
common.HexToAddress("0x627a0a960c079c21c34f7612d5d230e01b4ad4c7"),
common.HexToAddress("0xf0b1aa0eb660754448a7937c022e30aa692fe0c5"),
common.HexToAddress("0x24c4d950dfd4dd1902bbed3508144a54542bba94"),
common.HexToAddress("0x9f27daea7aca0aa0446220b98d028715e3bc803d"),
common.HexToAddress("0xa5dc5acd6a7968a4554d89d65e59b7fd3bff0f90"),
common.HexToAddress("0xd9aef3a1e38a39c16b31d1ace71bca8ef58d315b"),
common.HexToAddress("0x63ed5a272de2f6d968408b4acb9024f4cc208ebf"),
common.HexToAddress("0x6f6704e5a10332af6672e50b3d9754dc460dfa4d"),
common.HexToAddress("0x77ca7b50b6cd7e2f3fa008e24ab793fd56cb15f6"),
common.HexToAddress("0x492ea3bb0f3315521c31f273e565b868fc090f17"),
common.HexToAddress("0x0ff30d6de14a8224aa97b78aea5388d1c51c1f00"),
common.HexToAddress("0x9ea779f907f0b315b364b0cfc39a0fde5b02a416"),
common.HexToAddress("0xceaeb481747ca6c540a000c1f3641f8cef161fa7"),
common.HexToAddress("0xcc34673c6c40e791051898567a1222daf90be287"),
common.HexToAddress("0x579a80d909f346fbfb1189493f521d7f48d52238"),
common.HexToAddress("0xe308bd1ac5fda103967359b2712dd89deffb7973"),
common.HexToAddress("0x4cb31628079fb14e4bc3cd5e30c2f7489b00960c"),
common.HexToAddress("0xac1ecab32727358dba8962a0f3b261731aad9723"),
common.HexToAddress("0x4fd6ace747f06ece9c49699c7cabc62d02211f75"),
common.HexToAddress("0x440c59b325d2997a134c2c7c60a8c61611212bad"),
common.HexToAddress("0x4486a3d68fac6967006d7a517b889fd3f98c102b"),
common.HexToAddress("0x9c15b54878ba618f494b38f0ae7443db6af648ba"),
common.HexToAddress("0x27b137a85656544b1ccb5a0f2e561a5703c6a68f"),
common.HexToAddress("0x21c7fdb9ed8d291d79ffd82eb2c4356ec0d81241"),
common.HexToAddress("0x23b75c2f6791eef49c69684db4c6c1f93bf49a50"),
common.HexToAddress("0x1ca6abd14d30affe533b24d7a21bff4c2d5e1f3b"),
common.HexToAddress("0xb9637156d330c0d605a791f1c31ba5890582fe1c"),
common.HexToAddress("0x6131c42fa982e56929107413a9d526fd99405560"),
common.HexToAddress("0x1591fc0f688c81fbeb17f5426a162a7024d430c2"),
common.HexToAddress("0x542a9515200d14b68e934e9830d91645a980dd7a"),
common.HexToAddress("0xc4bbd073882dd2add2424cf47d35213405b01324"),
common.HexToAddress("0x782495b7b3355efb2833d56ecb34dc22ad7dfcc4"),
common.HexToAddress("0x58b95c9a9d5d26825e70a82b6adb139d3fd829eb"),
common.HexToAddress("0x3ba4d81db016dc2890c81f3acec2454bff5aada5"),
common.HexToAddress("0xb52042c8ca3f8aa246fa79c3feaa3d959347c0ab"),
common.HexToAddress("0xe4ae1efdfc53b73893af49113d8694a057b9c0d1"),
common.HexToAddress("0x3c02a7bc0391e86d91b7d144e61c2c01a25a79c5"),
common.HexToAddress("0x0737a6b837f97f46ebade41b9bc3e1c509c85c53"),
common.HexToAddress("0x97f43a37f595ab5dd318fb46e7a155eae057317a"),
common.HexToAddress("0x52c5317c848ba20c7504cb2c8052abd1fde29d03"),
common.HexToAddress("0x4863226780fe7c0356454236d3b1c8792785748d"),
common.HexToAddress("0x5d2b2e6fcbe3b11d26b525e085ff818dae332479"),
common.HexToAddress("0x5f9f3392e9f62f63b8eac0beb55541fc8627f42c"),
common.HexToAddress("0x057b56736d32b86616a10f619859c6cd6f59092a"),
common.HexToAddress("0x9aa008f65de0b923a2a4f02012ad034a5e2e2192"),
common.HexToAddress("0x304a554a310c7e546dfe434669c62820b7d83490"),
common.HexToAddress("0x914d1b8b43e92723e64fd0a06f5bdb8dd9b10c79"),
common.HexToAddress("0x4deb0033bb26bc534b197e61d19e0733e5679784"),
common.HexToAddress("0x07f5c1e1bc2c93e0402f23341973a0e043f7bf8a"),
common.HexToAddress("0x35a051a0010aba705c9008d7a7eff6fb88f6ea7b"),
common.HexToAddress("0x4fa802324e929786dbda3b8820dc7834e9134a2a"),
common.HexToAddress("0x9da397b9e80755301a3b32173283a91c0ef6c87e"),
common.HexToAddress("0x8d9edb3054ce5c5774a420ac37ebae0ac02343c6"),
common.HexToAddress("0x0101f3be8ebb4bbd39a2e3b9a3639d4259832fd9"),
common.HexToAddress("0x5dc28b15dffed94048d73806ce4b7a4612a1d48f"),
common.HexToAddress("0xbcf899e6c7d9d5a215ab1e3444c86806fa854c76"),
common.HexToAddress("0x12e626b0eebfe86a56d633b9864e389b45dcb260"),
common.HexToAddress("0xa2f1ccba9395d7fcb155bba8bc92db9bafaeade7"),
common.HexToAddress("0xec8e57756626fdc07c63ad2eafbd28d08e7b0ca5"),
common.HexToAddress("0xd164b088bd9108b60d0ca3751da4bceb207b0782"),
common.HexToAddress("0x6231b6d0d5e77fe001c2a460bd9584fee60d409b"),
common.HexToAddress("0x1cba23d343a983e9b5cfd19496b9a9701ada385f"),
common.HexToAddress("0xa82f360a8d3455c5c41366975bde739c37bfeb8a"),
common.HexToAddress("0x9fcd2deaff372a39cc679d5c5e4de7bafb0b1339"),
common.HexToAddress("0x005f5cee7a43331d5a3d3eec71305925a62f34b6"),
common.HexToAddress("0x0e0da70933f4c7849fc0d203f5d1d43b9ae4532d"),
common.HexToAddress("0xd131637d5275fd1a68a3200f4ad25c71a2a9522e"),
common.HexToAddress("0xbc07118b9ac290e4622f5e77a0853539789effbe"),
common.HexToAddress("0x47e7aa56d6bdf3f36be34619660de61275420af8"),
common.HexToAddress("0xacd87e28b0c9d1254e868b81cba4cc20d9a32225"),
common.HexToAddress("0xadf80daec7ba8dcf15392f1ac611fff65d94f880"),
common.HexToAddress("0x5524c55fb03cf21f549444ccbecb664d0acad706"),
common.HexToAddress("0x40b803a9abce16f50f36a77ba41180eb90023925"),
common.HexToAddress("0xfe24cdd8648121a43a7c86d289be4dd2951ed49f"),
common.HexToAddress("0x17802f43a0137c506ba92291391a8a8f207f487d"),
common.HexToAddress("0x253488078a4edf4d6f42f113d1e62836a942cf1a"),
common.HexToAddress("0x86af3e9626fce1957c82e88cbf04ddf3a2ed7915"),
common.HexToAddress("0xb136707642a4ea12fb4bae820f03d2562ebff487"),
common.HexToAddress("0xdbe9b615a3ae8709af8b93336ce9b477e4ac0940"),
common.HexToAddress("0xf14c14075d6c4ed84b86798af0956deef67365b5"),
common.HexToAddress("0xca544e5c4687d109611d0f8f928b53a25af72448"),
common.HexToAddress("0xaeeb8ff27288bdabc0fa5ebb731b6f409507516c"),
common.HexToAddress("0xcbb9d3703e651b0d496cdefb8b92c25aeb2171f7"),
common.HexToAddress("0x6d87578288b6cb5549d5076a207456a1f6a63dc0"),
common.HexToAddress("0xb2c6f0dfbb716ac562e2d85d6cb2f8d5ee87603e"),
common.HexToAddress("0xaccc230e8a6e5be9160b8cdf2864dd2a001c28b6"),
common.HexToAddress("0x2b3455ec7fedf16e646268bf88846bd7a2319bb2"),
common.HexToAddress("0x4613f3bca5c44ea06337a9e439fbc6d42e501d0a"),
common.HexToAddress("0xd343b217de44030afaa275f54d31a9317c7f441e"),
common.HexToAddress("0x84ef4b2357079cd7a7c69fd7a37cd0609a679106"),
common.HexToAddress("0xda2fef9e4a3230988ff17df2165440f37e8b1708"),
common.HexToAddress("0xf4c64518ea10f995918a454158c6b61407ea345c"),
common.HexToAddress("0x7602b46df5390e432ef1c307d4f2c9ff6d65cc97"),
common.HexToAddress("0xbb9bc244d798123fde783fcc1c72d3bb8c189413"),
common.HexToAddress("0x807640a13483f8ac783c557fcdf27be11ea4ac7a"),
}
}
```

31:F:\git\coin\ethereum\go-ethereum\params\denomination.go

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

package params

const (
// These are the multipliers for ether denominations.
// Example: To get the wei value of an amount in 'douglas', use
//
//    new(big.Int).Mul(value, big.NewInt(params.Douglas))
//
Wei      = 1
Ada      = 1e3
Babbage  = 1e6
Shannon  = 1e9
Szabo    = 1e12
Finney   = 1e15
Ether    = 1e18
Einstein = 1e21
Douglas  = 1e42
)
```

32:F:\git\coin\ethereum\go-ethereum\params\gas_table.go

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

```
package params

type GasTable struct {
ExtcodeSize uint64
ExtcodeCopy uint64
Balance     uint64
SLoad       uint64
Calls       uint64
Suicide     uint64

ExpByte uint64

// CreateBySuicide occurs when the
// refunded account is one that does
// not exist. This logic is similar
// to call. May be left nil. Nil means
// not charged.
CreateBySuicide uint64
}

var (
// GasTableHomestead contain the gas prices for
// the homestead phase.
GasTableHomestead = GasTable{
ExtcodeSize: 20,
ExtcodeCopy: 20,
Balance:     20,
SLoad:       50,
Calls:       40,
Suicide:     0,
ExpByte:     10,
}

// GasTableHomestead contain the gas re-prices for
// the homestead phase.
//
// TODO rename to GasTableEIP150
GasTableHomesteadGasRepriceFork = GasTable{
ExtcodeSize: 700,
ExtcodeCopy: 700,
Balance:     400,
SLoad:       200,
Calls:       700,
Suicide:     5000,
ExpByte:     10,

CreateBySuicide: 25000,
}

GasTableEIP158 = GasTable{
ExtcodeSize: 700,
ExtcodeCopy: 700,
Balance:     400,
SLoad:       200,
Calls:       700,
Suicide:     5000,
ExpByte:     50,

CreateBySuicide: 25000,
}
)

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

package params

import "math/big"

const (
MaximumExtraDataSize  uint64 = 32    // Maximum size extra data may be after Genesis.
ExpByteGas            uint64 = 10    // Times ceil(log256(exponent)) for the EXP instruction.
SloadGas              uint64 = 50    // Multiplied by the number of 32-byte words that are copied
(round up) for any *COPY operation and added.
CallValueTransferGas  uint64 = 9000  // Paid for CALL when the value transfer is non-zero.
CallNewAccountGas     uint64 = 25000 // Paid for CALL when the destination address didn't exist
prior.
TxGas                 uint64 = 21000 // Per transaction not creating a contract. NOTE: Not payable on
data of calls between transactions.
TxGasContractCreation uint64 = 53000 // Per transaction that creates a contract. NOTE: Not
payable on data of calls between transactions.
TxDataZeroGas         uint64 = 4     // Per byte of data attached to a transaction that equals zero.
NOTE: Not payable on data of calls between transactions.
```

```
QuadCoeffDiv         uint64 = 512   // Divisor for the quadratic particle of the memory cost equation.
SstoreSetGas         uint64 = 20000 // Once per SLOAD operation.
LogDataGas           uint64 = 8     // Per byte in a LOG* operation's data.
CallStipend          uint64 = 2300  // Free gas given at beginning of call.
EcrecoverGas         uint64 = 3000  //
Sha256WordGas        uint64 = 12    //

Sha3Gas          uint64 = 30    // Once per SHA3 operation.
Sha256Gas        uint64 = 60    //
IdentityWordGas  uint64 = 3     //
Sha3WordGas      uint64 = 6     // Once per word of the SHA3 operation's data.
SstoreResetGas   uint64 = 5000  // Once per SSTORE operation if the zeroness changes from
zero.
SstoreClearGas   uint64 = 5000  // Once per SSTORE operation if the zeroness doesn't change.
SstoreRefundGas  uint64 = 15000 // Once per SSTORE operation if the zeroness changes to
zero.
JumpdestGas      uint64 = 1     // Refunded gas, once per SSTORE operation if the zeroness
changes to zero.
IdentityGas      uint64 = 15    //
EpochDuration    uint64 = 30000 // Duration between proof-of-work epochs.
CallGas          uint64 = 40    // Once per CALL operation & message call transaction.
CreateDataGas    uint64 = 200   //
Ripemd160Gas     uint64 = 600   //
Ripemd160WordGas uint64 = 120   //
CallCreateDepth  uint64 = 1024  // Maximum depth of call/create stack.
ExpGas           uint64 = 10    // Once per EXP instruction
LogGas           uint64 = 375   // Per LOG* operation.
CopyGas          uint64 = 3     //
StackLimit       uint64 = 1024  // Maximum size of VM stack allowed.
TierStepGas      uint64 = 0     // Once per operation, for a selection of them.
LogTopicGas      uint64 = 375   // Multiplied by the * of the LOG*, per LOG transaction. e.g. LOG0
incurs 0 * c_txLogTopicGas, LOG4 incurs 4 * c_txLogTopicGas.
CreateGas        uint64 = 32000 // Once per CREATE operation & contract-creation transaction.
SuicideRefundGas uint64 = 24000 // Refunded following a suicide operation.
MemoryGas        uint64 = 3     // Times the address of the (highest referenced byte in memory +
1). NOTE: referencing happens on read, write and in instructions such as RETURN and CALL.
TxDataNonZeroGas uint64 = 68    // Per byte of data attached to a transaction that is not equal to
zero. NOTE: Not payable on data of calls between transactions.

MaxCodeSize = 24576
)
```

```
var (
GasLimitBoundDivisor   = big.NewInt(1024)                  // The bound divisor of the gas limit, used in
update calculations.
MinGasLimit            = big.NewInt(5000)                  // Minimum the gas limit may ever be.
GenesisGasLimit        = big.NewInt(4712388)               // Gas limit of the Genesis block.
TargetGasLimit         = new(big.Int).Set(GenesisGasLimit) // The artificial target
DifficultyBoundDivisor = big.NewInt(2048)                  // The bound divisor of the difficulty, used in
the update calculations.
GenesisDifficulty      = big.NewInt(131072)                // Difficulty of the Genesis block.
MinimumDifficulty      = big.NewInt(131072)                // The minimum that the difficulty may ever
be.
DurationLimit          = big.NewInt(13)                    // The decision boundary on the blocktime
duration used to determine whether difficulty should go up or not.
)
```

34:F:\git\coin\ethereum\go-ethereum\params\version.go

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

package params

import (
"fmt"
)

const (
VersionMajor = 1          // Major version component of the current release
VersionMinor = 6          // Minor version component of the current release
VersionPatch = 7          // Patch version component of the current release
VersionMeta  = "unstable" // Version metadata to append to the version string
)

// Version holds the textual version string.
var Version = func() string {
v := fmt.Sprintf("%d.%d.%d", VersionMajor, VersionMinor, VersionPatch)
if VersionMeta != "" {
v += "-" + VersionMeta
}
return v
}()

func VersionWithCommit(gitCommit string) string {
vsn := Version
if len(gitCommit) >= 8 {
vsn += "-" + gitCommit[:8]
}
return vsn
}

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

package rlp

import (
"bufio"
"bytes"
"encoding/binary"
"errors"
"fmt"
"io"
"math/big"
"reflect"
"strings"
)

var (
errNoPointer     = errors.New("rlp: interface given to Decode must be a pointer")
errDecodeIntoNil = errors.New("rlp: pointer given to Decode must not be nil")
)

// Decoder is implemented by types that require custom RLP
// decoding rules or need to decode into private fields.
//
// The DecodeRLP method should read one value from the given
// Stream. It is not forbidden to read less or more, but it might
// be confusing.
type Decoder interface {
DecodeRLP(*Stream) error
}

// Decode parses RLP-encoded data from r and stores the result in the
// value pointed to by val. Val must be a non-nil pointer. If r does
// not implement ByteReader, Decode will do its own buffering.
//
```

```
// Decode uses the following type-dependent decoding rules:
//
// If the type implements the Decoder interface, decode calls
// DecodeRLP.
//
// To decode into a pointer, Decode will decode into the value pointed
// to. If the pointer is nil, a new value of the pointer's element
// type is allocated. If the pointer is non-nil, the existing value
// will be reused.
//
// To decode into a struct, Decode expects the input to be an RLP
// list. The decoded elements of the list are assigned to each public
// field in the order given by the struct's definition. The input list
// must contain an element for each decoded field. Decode returns an
// error if there are too few or too many elements.
//
// The decoding of struct fields honours certain struct tags, "tail",
// "nil" and "-".
//
// The "-" tag ignores fields.
//
// For an explanation of "tail", see the example.
//
// The "nil" tag applies to pointer-typed fields and changes the decoding
// rules for the field such that input values of size zero decode as a nil
// pointer. This tag can be useful when decoding recursive types.
//
//     type StructWithEmptyOK struct {
//         Foo *[20]byte `rlp:"nil"`
//     }
//
// To decode into a slice, the input must be a list and the resulting
// slice will contain the input elements in order. For byte slices,
// the input must be an RLP string. Array types decode similarly, with
// the additional restriction that the number of input elements (or
// bytes) must match the array's length.
//
// To decode into a Go string, the input must be an RLP string. The
// input bytes are taken as-is and will not necessarily be valid UTF-8.
//
// To decode into an unsigned integer type, the input must also be an RLP
// string. The bytes are interpreted as a big endian representation of
// the integer. If the RLP string is larger than the bit size of the
// type, Decode will return an error. Decode also supports *big.Int.
// There is no size limit for big integers.
//
// To decode into an interface value, Decode stores one of these
// in the value:
//
//  []interface{}, for RLP lists
//  []byte, for RLP strings
//
// Non-empty interface types are not supported, nor are booleans,
// signed integers, floating point numbers, maps, channels and
// functions.
//
// Note that Decode does not set an input limit for all readers
// and may be vulnerable to panics cause by huge value sizes. If
// you need an input limit, use
//
//     NewStream(r, limit).Decode(val)
func Decode(r io.Reader, val interface{}) error {
// TODO: this could use a Stream from a pool.
return NewStream(r, 0).Decode(val)
}

// DecodeBytes parses RLP data from b into val.
// Please see the documentation of Decode for the decoding rules.
// The input must contain exactly one value and no trailing data.
func DecodeBytes(b []byte, val interface{}) error {
// TODO: this could use a Stream from a pool.
r := bytes.NewReader(b)
if err := NewStream(r, uint64(len(b))).Decode(val); err != nil {
return err
}
if r.Len() > 0 {
return ErrMoreThanOneValue
}
return nil
}

type decodeError struct {
msg string
typ reflect.Type
```

```
ctx []string
}

func (err *decodeError) Error() string {
ctx := ""
if len(err.ctx) > 0 {
ctx = ", decoding into "
for i := len(err.ctx) - 1; i >= 0; i-- {
ctx += err.ctx[i]
}
}
return fmt.Sprintf("rlp: %s for %v%s", err.msg, err.typ, ctx)
}

func wrapStreamError(err error, typ reflect.Type) error {
switch err {
case ErrCanonInt:
return &decodeError{msg: "non-canonical integer (leading zero bytes)", typ: typ}
case ErrCanonSize:
return &decodeError{msg: "non-canonical size information", typ: typ}
case ErrExpectedList:
return &decodeError{msg: "expected input list", typ: typ}
case ErrExpectedString:
return &decodeError{msg: "expected input string or byte", typ: typ}
case errUintOverflow:
return &decodeError{msg: "input string too long", typ: typ}
case errNotAtEOL:
return &decodeError{msg: "input list has too many elements", typ: typ}
}
return err
}

func addErrorContext(err error, ctx string) error {
if decErr, ok := err.(*decodeError); ok {
decErr.ctx = append(decErr.ctx, ctx)
}
return err
}

var (
decoderInterface = reflect.TypeOf(new(Decoder)).Elem()
bigInt           = reflect.TypeOf(big.Int{})
```

```
)

func makeDecoder(typ reflect.Type, tags tags) (dec decoder, err error) {
kind := typ.Kind()
switch {
case typ == rawValueType:
return decodeRawValue, nil
case typ.Implements(decoderInterface):
return decodeDecoder, nil
case kind != reflect.Ptr && reflect.PtrTo(typ).Implements(decoderInterface):
return decodeDecoderNoPtr, nil
case typ.AssignableTo(reflect.PtrTo(bigInt)):
return decodeBigInt, nil
case typ.AssignableTo(bigInt):
return decodeBigIntNoPtr, nil
case isUint(kind):
return decodeUint, nil
case kind == reflect.Bool:
return decodeBool, nil
case kind == reflect.String:
return decodeString, nil
case kind == reflect.Slice || kind == reflect.Array:
return makeListDecoder(typ, tags)
case kind == reflect.Struct:
return makeStructDecoder(typ)
case kind == reflect.Ptr:
if tags.nilOK {
return makeOptionalPtrDecoder(typ)
}
return makePtrDecoder(typ)
case kind == reflect.Interface:
return decodeInterface, nil
default:
return nil, fmt.Errorf("rlp: type %v is not RLP-serializable", typ)
}
}

func decodeRawValue(s *Stream, val reflect.Value) error {
r, err := s.Raw()
if err != nil {
return err
}
```

```
val.SetBytes(r)
return nil
}

func decodeUint(s *Stream, val reflect.Value) error {
typ := val.Type()
num, err := s.uint(typ.Bits())
if err != nil {
return wrapStreamError(err, val.Type())
}
val.SetUint(num)
return nil
}

func decodeBool(s *Stream, val reflect.Value) error {
b, err := s.Bool()
if err != nil {
return wrapStreamError(err, val.Type())
}
val.SetBool(b)
return nil
}

func decodeString(s *Stream, val reflect.Value) error {
b, err := s.Bytes()
if err != nil {
return wrapStreamError(err, val.Type())
}
val.SetString(string(b))
return nil
}

func decodeBigIntNoPtr(s *Stream, val reflect.Value) error {
return decodeBigInt(s, val.Addr())
}

func decodeBigInt(s *Stream, val reflect.Value) error {
b, err := s.Bytes()
if err != nil {
return wrapStreamError(err, val.Type())
}
i := val.Interface().(*big.Int)
```

```
if i == nil {
i = new(big.Int)
val.Set(reflect.ValueOf(i))
}
// Reject leading zero bytes
if len(b) > 0 && b[0] == 0 {
return wrapStreamError(ErrCanonInt, val.Type())
}
i.SetBytes(b)
return nil
}

func makeListDecoder(typ reflect.Type, tag tags) (decoder, error) {
etype := typ.Elem()
if etype.Kind() == reflect.Uint8 && !reflect.PtrTo(etype).Implements(decoderInterface) {
if typ.Kind() == reflect.Array {
return decodeByteArray, nil
} else {
return decodeByteSlice, nil
}
}
etypeinfo, err := cachedTypeInfo1(etype, tags{})
if err != nil {
return nil, err
}
var dec decoder
switch {
case typ.Kind() == reflect.Array:
dec = func(s *Stream, val reflect.Value) error {
return decodeListArray(s, val, etypeinfo.decoder)
}
case tag.tail:
// A slice with "tail" tag can occur as the last field
// of a struct and is supposed to swallow all remaining
// list elements. The struct decoder already called s.List,
// proceed directly to decoding the elements.
dec = func(s *Stream, val reflect.Value) error {
return decodeSliceElems(s, val, etypeinfo.decoder)
}
default:
dec = func(s *Stream, val reflect.Value) error {
return decodeListSlice(s, val, etypeinfo.decoder)
}
}
return dec, nil
}

func decodeListSlice(s *Stream, val reflect.Value, elemdec decoder) error {
size, err := s.List()
if err != nil {
return wrapStreamError(err, val.Type())
}
if size == 0 {
val.Set(reflect.MakeSlice(val.Type(), 0, 0))
return s.ListEnd()
}
if err := decodeSliceElems(s, val, elemdec); err != nil {
return err
}
return s.ListEnd()
}

func decodeSliceElems(s *Stream, val reflect.Value, elemdec decoder) error {
i := 0
for ; ; i++ {
// grow slice if necessary
if i >= val.Cap() {
newcap := val.Cap() + val.Cap()/2
if newcap < 4 {
newcap = 4
}
newv := reflect.MakeSlice(val.Type(), val.Len(), newcap)
reflect.Copy(newv, val)
val.Set(newv)
}
if i >= val.Len() {
val.SetLen(i + 1)
}
// decode into element
if err := elemdec(s, val.Index(i)); err == EOL {
break
} else if err != nil {
return addErrorContext(err, fmt.Sprint("[", i, "]"))
}
```

```
}
if i < val.Len() {
val.SetLen(i)
}
return nil
}

func decodeListArray(s *Stream, val reflect.Value, elemdec decoder) error {
if _, err := s.List(); err != nil {
return wrapStreamError(err, val.Type())
}
vlen := val.Len()
i := 0
for ; i < vlen; i++ {
if err := elemdec(s, val.Index(i)); err == EOL {
break
} else if err != nil {
return addErrorContext(err, fmt.Sprint("[", i, "]"))
}
}
if i < vlen {
return &decodeError{msg: "input list has too few elements", typ: val.Type()}
}
return wrapStreamError(s.ListEnd(), val.Type())
}

func decodeByteSlice(s *Stream, val reflect.Value) error {
b, err := s.Bytes()
if err != nil {
return wrapStreamError(err, val.Type())
}
val.SetBytes(b)
return nil
}

func decodeByteArray(s *Stream, val reflect.Value) error {
kind, size, err := s.Kind()
if err != nil {
return err
}
vlen := val.Len()
switch kind {
```

```
case Byte:
if vlen == 0 {
return &decodeError{msg: "input string too long", typ: val.Type()}
}
if vlen > 1 {
return &decodeError{msg: "input string too short", typ: val.Type()}
}
bv, _ := s.Uint()
val.Index(0).SetUint(bv)
case String:
if uint64(vlen) < size {
return &decodeError{msg: "input string too long", typ: val.Type()}
}
if uint64(vlen) > size {
return &decodeError{msg: "input string too short", typ: val.Type()}
}
slice := val.Slice(0, vlen).Interface().([]byte)
if err := s.readFull(slice); err != nil {
return err
}
// Reject cases where single byte encoding should have been used.
if size == 1 && slice[0] < 128 {
return wrapStreamError(ErrCanonSize, val.Type())
}
case List:
return wrapStreamError(ErrExpectedString, val.Type())
}
return nil
}

func makeStructDecoder(typ reflect.Type) (decoder, error) {
fields, err := structFields(typ)
if err != nil {
return nil, err
}
dec := func(s *Stream, val reflect.Value) (err error) {
if _, err := s.List(); err != nil {
return wrapStreamError(err, typ)
}
for _, f := range fields {
err := f.info.decoder(s, val.Field(f.index))
if err == EOL {
```

```
return &decodeError{msg: "too few elements", typ: typ}
} else if err != nil {
return addErrorContext(err, "."+typ.Field(f.index).Name)
}
}
return wrapStreamError(s.ListEnd(), typ)
}
return dec, nil
}

// makePtrDecoder creates a decoder that decodes into
// the pointer's element type.
func makePtrDecoder(typ reflect.Type) (decoder, error) {
etype := typ.Elem()
etypeinfo, err := cachedTypeInfo1(etype, tags{})
if err != nil {
return nil, err
}
dec := func(s *Stream, val reflect.Value) (err error) {
newval := val
if val.IsNil() {
newval = reflect.New(etype)
}
if err = etypeinfo.decoder(s, newval.Elem()); err == nil {
val.Set(newval)
}
return err
}
return dec, nil
}

// makeOptionalPtrDecoder creates a decoder that decodes empty values
// as nil. Non-empty values are decoded into a value of the element type,
// just like makePtrDecoder does.
//
// This decoder is used for pointer-typed struct fields with struct tag "nil".
func makeOptionalPtrDecoder(typ reflect.Type) (decoder, error) {
etype := typ.Elem()
etypeinfo, err := cachedTypeInfo1(etype, tags{})
if err != nil {
return nil, err
}
```

```
dec := func(s *Stream, val reflect.Value) (err error) {
kind, size, err := s.Kind()
if err != nil || size == 0 && kind != Byte {
// rearm s.Kind. This is important because the input
// position must advance to the next value even though
// we don't read anything.
s.kind = -1
// set the pointer to nil.
val.Set(reflect.Zero(typ))
return err
}
newval := val
if val.IsNil() {
newval = reflect.New(etype)
}
if err = etypeinfo.decoder(s, newval.Elem()); err == nil {
val.Set(newval)
}
return err
}
return dec, nil
}

var ifsliceType = reflect.TypeOf([]interface{}{})

func decodeInterface(s *Stream, val reflect.Value) error {
if val.Type().NumMethod() != 0 {
return fmt.Errorf("rlp: type %v is not RLP-serializable", val.Type())
}
kind, _, err := s.Kind()
if err != nil {
return err
}
if kind == List {
slice := reflect.New(ifsliceType).Elem()
if err := decodeListSlice(s, slice, decodeInterface); err != nil {
return err
}
val.Set(slice)
} else {
b, err := s.Bytes()
if err != nil {
```

```
return err
}
val.Set(reflect.ValueOf(b))
}
return nil
}

// This decoder is used for non-pointer values of types
// that implement the Decoder interface using a pointer receiver.
func decodeDecoderNoPtr(s *Stream, val reflect.Value) error {
return val.Addr().Interface().(Decoder).DecodeRLP(s)
}

func decodeDecoder(s *Stream, val reflect.Value) error {
// Decoder instances are not handled using the pointer rule if the type
// implements Decoder with pointer receiver (i.e. always)
// because it might handle empty values specially.
// We need to allocate one here in this case, like makePtrDecoder does.
if val.Kind() == reflect.Ptr && val.IsNil() {
val.Set(reflect.New(val.Type().Elem()))
}
return val.Interface().(Decoder).DecodeRLP(s)
}

// Kind represents the kind of value contained in an RLP stream.
type Kind int

const (
Byte Kind = iota
String
List
)

func (k Kind) String() string {
switch k {
case Byte:
return "Byte"
case String:
return "String"
case List:
return "List"
default:
```

```
return fmt.Sprintf("Unknown(%d)", k)
}
}

var (
// EOL is returned when the end of the current list
// has been reached during streaming.
EOL = errors.New("rlp: end of list")

// Actual Errors
ErrExpectedString = errors.New("rlp: expected String or Byte")
ErrExpectedList   = errors.New("rlp: expected List")
ErrCanonInt       = errors.New("rlp: non-canonical integer format")
ErrCanonSize      = errors.New("rlp: non-canonical size information")
ErrElemTooLarge   = errors.New("rlp: element is larger than containing list")
ErrValueTooLarge  = errors.New("rlp: value size exceeds available input length")

// This error is reported by DecodeBytes if the slice contains
// additional data after the first RLP value.
ErrMoreThanOneValue = errors.New("rlp: input contains more than one value")

// internal errors
errNotInList    = errors.New("rlp: call of ListEnd outside of any list")
errNotAtEOL     = errors.New("rlp: call of ListEnd not positioned at EOL")
errUintOverflow = errors.New("rlp: uint overflow")
)

// ByteReader must be implemented by any input reader for a Stream. It
// is implemented by e.g. bufio.Reader and bytes.Reader.
type ByteReader interface {
io.Reader
io.ByteReader
}

// Stream can be used for piecemeal decoding of an input stream. This
// is useful if the input is very large or if the decoding rules for a
// type depend on the input structure. Stream does not keep an
// internal buffer. After decoding a value, the input reader will be
// positioned just before the type information for the next value.
//
// When decoding a list and the input position reaches the declared
// length of the list, all operations will return error EOL.
// The end of the list must be acknowledged using ListEnd to continue
// reading the enclosing list.
//
// Stream is not safe for concurrent use.
type Stream struct {
r ByteReader

// number of bytes remaining to be read from r.
remaining uint64
limited   bool

// auxiliary buffer for integer decoding
uintbuf []byte

kind    Kind   // kind of value ahead
size    uint64 // size of value ahead
byteval byte   // value of single byte in type tag
kinderr error  // error from last readKind
stack   []listpos
}

type listpos struct{ pos, size uint64 }

// NewStream creates a new decoding stream reading from r.
//
// If r implements the ByteReader interface, Stream will
// not introduce any buffering.
//
// For non-toplevel values, Stream returns ErrElemTooLarge
// for values that do not fit into the enclosing list.
//
// Stream supports an optional input limit. If a limit is set, the
// size of any toplevel value will be checked against the remaining
// input length. Stream operations that encounter a value exceeding
// the remaining input length will return ErrValueTooLarge. The limit
// can be set by passing a non-zero value for inputLimit.
//
// If r is a bytes.Reader or strings.Reader, the input limit is set to
// the length of r's underlying data unless an explicit limit is
// provided.
func NewStream(r io.Reader, inputLimit uint64) *Stream {
s := new(Stream)
```

```
s.Reset(r, inputLimit)
return s
}

// NewListStream creates a new stream that pretends to be positioned
// at an encoded list of the given length.
func NewListStream(r io.Reader, len uint64) *Stream {
s := new(Stream)
s.Reset(r, len)
s.kind = List
s.size = len
return s
}

// Bytes reads an RLP string and returns its contents as a byte slice.
// If the input does not contain an RLP string, the returned
// error will be ErrExpectedString.
func (s *Stream) Bytes() ([]byte, error) {
kind, size, err := s.Kind()
if err != nil {
return nil, err
}
switch kind {
case Byte:
s.kind = -1 // rearm Kind
return []byte{s.byteval}, nil
case String:
b := make([]byte, size)
if err = s.readFull(b); err != nil {
return nil, err
}
if size == 1 && b[0] < 128 {
return nil, ErrCanonSize
}
return b, nil
default:
return nil, ErrExpectedString
}
}

// Raw reads a raw encoded value including RLP type information.
func (s *Stream) Raw() ([]byte, error) {
```

```
kind, size, err := s.Kind()
if err != nil {
return nil, err
}
if kind == Byte {
s.kind = -1 // rearm Kind
return []byte{s.byteval}, nil
}
// the original header has already been read and is no longer
// available. read content and put a new header in front of it.
start := headsize(size)
buf := make([]byte, uint64(start)+size)
if err := s.readFull(buf[start:]); err != nil {
return nil, err
}
if kind == String {
puthead(buf, 0x80, 0xB8, size)
} else {
puthead(buf, 0xC0, 0xF7, size)
}
return buf, nil
}

// Uint reads an RLP string of up to 8 bytes and returns its contents
// as an unsigned integer. If the input does not contain an RLP string, the
// returned error will be ErrExpectedString.
func (s *Stream) Uint() (uint64, error) {
return s.uint(64)
}

func (s *Stream) uint(maxbits int) (uint64, error) {
kind, size, err := s.Kind()
if err != nil {
return 0, err
}
switch kind {
case Byte:
if s.byteval == 0 {
return 0, ErrCanonInt
}
s.kind = -1 // rearm Kind
return uint64(s.byteval), nil
case String:
if size > uint64(maxbits/8) {
return 0, errUintOverflow
}
v, err := s.readUint(byte(size))
switch {
case err == ErrCanonSize:
// Adjust error because we're not reading a size right now.
return 0, ErrCanonInt
case err != nil:
return 0, err
case size > 0 && v < 128:
return 0, ErrCanonSize
default:
return v, nil
}
default:
return 0, ErrExpectedString
}
}

// Bool reads an RLP string of up to 1 byte and returns its contents
// as a boolean. If the input does not contain an RLP string, the
// returned error will be ErrExpectedString.
func (s *Stream) Bool() (bool, error) {
num, err := s.uint(8)
if err != nil {
return false, err
}
switch num {
case 0:
return false, nil
case 1:
return true, nil
default:
return false, fmt.Errorf("rlp: invalid boolean value: %d", num)
}
}

// List starts decoding an RLP list. If the input does not contain a
// list, the returned error will be ErrExpectedList. When the list's
// end has been reached, any Stream operation will return EOL.
```

```
func (s *Stream) List() (size uint64, err error) {
kind, size, err := s.Kind()
if err != nil {
return 0, err
}
if kind != List {
return 0, ErrExpectedList
}
s.stack = append(s.stack, listpos{0, size})
s.kind = -1
s.size = 0
return size, nil
}

// ListEnd returns to the enclosing list.
// The input reader must be positioned at the end of a list.
func (s *Stream) ListEnd() error {
if len(s.stack) == 0 {
return errNotInList
}
tos := s.stack[len(s.stack)-1]
if tos.pos != tos.size {
return errNotAtEOL
}
s.stack = s.stack[:len(s.stack)-1] // pop
if len(s.stack) > 0 {
s.stack[len(s.stack)-1].pos += tos.size
}
s.kind = -1
s.size = 0
return nil
}

// Decode decodes a value and stores the result in the value pointed
// to by val. Please see the documentation for the Decode function
// to learn about the decoding rules.
func (s *Stream) Decode(val interface{}) error {
if val == nil {
return errDecodeIntoNil
}
rval := reflect.ValueOf(val)
rtyp := rval.Type()
```

```
if rtyp.Kind() != reflect.Ptr {
return errNoPointer
}
if rval.IsNil() {
return errDecodeIntoNil
}
info, err := cachedTypeInfo(rtyp.Elem(), tags{})
if err != nil {
return err
}

err = info.decoder(s, rval.Elem())
if decErr, ok := err.(*decodeError); ok && len(decErr.ctx) > 0 {
// add decode target type to error so context has more meaning
decErr.ctx = append(decErr.ctx, fmt.Sprint("(", rtyp.Elem(), ")"))
}
return err
}

// Reset discards any information about the current decoding context
// and starts reading from r. This method is meant to facilitate reuse
// of a preallocated Stream across many decoding operations.
//
// If r does not also implement ByteReader, Stream will do its own
// buffering.
func (s *Stream) Reset(r io.Reader, inputLimit uint64) {
if inputLimit > 0 {
s.remaining = inputLimit
s.limited = true
} else {
// Attempt to automatically discover
// the limit when reading from a byte slice.
switch br := r.(type) {
case *bytes.Reader:
s.remaining = uint64(br.Len())
s.limited = true
case *strings.Reader:
s.remaining = uint64(br.Len())
s.limited = true
default:
s.limited = false
}
```

```
}
// Wrap r with a buffer if it doesn't have one.
bufr, ok := r.(ByteReader)
if !ok {
bufr = bufio.NewReader(r)
}
s.r = bufr
// Reset the decoding context.
s.stack = s.stack[:0]
s.size = 0
s.kind = -1
s.kinderr = nil
if s.uintbuf == nil {
s.uintbuf = make([]byte, 8)
}
}

// Kind returns the kind and size of the next value in the
// input stream.
//
// The returned size is the number of bytes that make up the value.
// For kind == Byte, the size is zero because the value is
// contained in the type tag.
//
// The first call to Kind will read size information from the input
// reader and leave it positioned at the start of the actual bytes of
// the value. Subsequent calls to Kind (until the value is decoded)
// will not advance the input reader and return cached information.
func (s *Stream) Kind() (kind Kind, size uint64, err error) {
var tos *listpos
if len(s.stack) > 0 {
tos = &s.stack[len(s.stack)-1]
}
if s.kind < 0 {
s.kinderr = nil
// Don't read further if we're at the end of the
// innermost list.
if tos != nil && tos.pos == tos.size {
return 0, 0, EOL
}
s.kind, s.size, s.kinderr = s.readKind()
if s.kinderr == nil {
```

```
if tos == nil {
// At toplevel, check that the value is smaller
// than the remaining input length.
if s.limited && s.size > s.remaining {
s.kinderr = ErrValueTooLarge
}
} else {
// Inside a list, check that the value doesn't overflow the list.
if s.size > tos.size-tos.pos {
s.kinderr = ErrElemTooLarge
}
}
}
}
// Note: this might return a sticky error generated
// by an earlier call to readKind.
return s.kind, s.size, s.kinderr
}

func (s *Stream) readKind() (kind Kind, size uint64, err error) {
b, err := s.readByte()
if err != nil {
if len(s.stack) == 0 {
// At toplevel, Adjust the error to actual EOF. io.EOF is
// used by callers to determine when to stop decoding.
switch err {
case io.ErrUnexpectedEOF:
err = io.EOF
case ErrValueTooLarge:
err = io.EOF
}
}
return 0, 0, err
}
s.byteval = 0
switch {
case b < 0x80:
// For a single byte whose value is in the [0x00, 0x7F] range, that byte
// is its own RLP encoding.
s.byteval = b
return Byte, 0, nil
case b < 0xB8:
```

```
// Otherwise, if a string is 0-55 bytes long,
// the RLP encoding consists of a single byte with value 0x80 plus the
// length of the string followed by the string. The range of the first
// byte is thus [0x80, 0xB7].
return String, uint64(b - 0x80), nil
case b < 0xC0:
// If a string is more than 55 bytes long, the
// RLP encoding consists of a single byte with value 0xB7 plus the length
// of the length of the string in binary form, followed by the length of
// the string, followed by the string. For example, a length-1024 string
// would be encoded as 0xB90400 followed by the string. The range of
// the first byte is thus [0xB8, 0xBF].
size, err = s.readUint(b - 0xB7)
if err == nil && size < 56 {
err = ErrCanonSize
}
return String, size, err
case b < 0xF8:
// If the total payload of a list
// (i.e. the combined length of all its items) is 0-55 bytes long, the
// RLP encoding consists of a single byte with value 0xC0 plus the length
// of the list followed by the concatenation of the RLP encodings of the
// items. The range of the first byte is thus [0xC0, 0xF7].
return List, uint64(b - 0xC0), nil
default:
// If the total payload of a list is more than 55 bytes long,
// the RLP encoding consists of a single byte with value 0xF7
// plus the length of the length of the payload in binary
// form, followed by the length of the payload, followed by
// the concatenation of the RLP encodings of the items. The
// range of the first byte is thus [0xF8, 0xFF].
size, err = s.readUint(b - 0xF7)
if err == nil && size < 56 {
err = ErrCanonSize
}
return List, size, err
}
}

func (s *Stream) readUint(size byte) (uint64, error) {
switch size {
case 0:
```

```
s.kind = -1 // rearm Kind
return 0, nil
case 1:
b, err := s.readByte()
return uint64(b), err
default:
start := int(8 - size)
for i := 0; i < start; i++ {
s.uintbuf[i] = 0
}
if err := s.readFull(s.uintbuf[start:]); err != nil {
return 0, err
}
if s.uintbuf[start] == 0 {
// Note: readUint is also used to decode integer
// values. The error needs to be adjusted to become
// ErrCanonInt in this case.
return 0, ErrCanonSize
}
return binary.BigEndian.Uint64(s.uintbuf), nil
}
}

func (s *Stream) readFull(buf []byte) (err error) {
if err := s.willRead(uint64(len(buf))); err != nil {
return err
}
var nn, n int
for n < len(buf) && err == nil {
nn, err = s.r.Read(buf[n:])
n += nn
}
if err == io.EOF {
err = io.ErrUnexpectedEOF
}
return err
}

func (s *Stream) readByte() (byte, error) {
if err := s.willRead(1); err != nil {
return 0, err
}
```

```
b, err := s.r.ReadByte()
if err == io.EOF {
err = io.ErrUnexpectedEOF
}
return b, err
}

func (s *Stream) willRead(n uint64) error {
s.kind = -1 // rearm Kind

if len(s.stack) > 0 {
// check list overflow
tos := s.stack[len(s.stack)-1]
if n > tos.size-tos.pos {
return ErrElemTooLarge
}
s.stack[len(s.stack)-1].pos += n
}
if s.limited {
if n > s.remaining {
return ErrValueTooLarge
}
s.remaining -= n
}
return nil
}

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

package rlp

import (
"bytes"
"fmt"
)

type structWithTail struct {
A, B uint
C    []uint `rlp:"tail"`
}
```

```
func ExampleDecode_structTagTail() {
// In this example, the "tail" struct tag is used to decode lists of
// differing length into a struct.
var val structWithTail

err := Decode(bytes.NewReader([]byte{0xC4, 0x01, 0x02, 0x03, 0x04}), &val)
fmt.Printf("with 4 elements: err=%v val=%v\n", err, val)

err = Decode(bytes.NewReader([]byte{0xC6, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06}), &val)
fmt.Printf("with 6 elements: err=%v val=%v\n", err, val)

// Note that at least two list elements must be present to
// fill fields A and B:
err = Decode(bytes.NewReader([]byte{0xC1, 0x01}), &val)
fmt.Printf("with 1 element: err=%q\n", err)

// Output:
// with 4 elements: err=<nil> val={1 2 [3 4]}
// with 6 elements: err=<nil> val={1 2 [3 4 5 6]}
// with 1 element: err="rlp: too few elements for rlp.structWithTail"
}

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

package rlp

import (
"bytes"
"encoding/hex"
"errors"
"fmt"
"io"
"math/big"
"reflect"
"strings"
"testing"
)

func TestStreamKind(t *testing.T) {
tests := []struct {
input    string
wantKind Kind
wantLen  uint64
}{
{"00", Byte, 0},
{"01", Byte, 0},
{"7F", Byte, 0},
{"80", String, 0},
{"B7", String, 55},
{"B90400", String, 1024},
{"BFFFFFFFFFFFFFFFFF", String, ^uint64(0)},
{"C0", List, 0},
{"C8", List, 8},
{"F7", List, 55},
{"F90400", List, 1024},
{"FFFFFFFFFFFFFFFFFF", List, ^uint64(0)},
}

for i, test := range tests {
// using plainReader to inhibit input limit errors.
s := NewStream(newPlainReader(unhex(test.input)), 0)
kind, len, err := s.Kind()
if err != nil {
t.Errorf("test %d: Kind returned error: %v", i, err)
continue
}
if kind != test.wantKind {
t.Errorf("test %d: kind mismatch: got %d, want %d", i, kind, test.wantKind)
}
if len != test.wantLen {
t.Errorf("test %d: len mismatch: got %d, want %d", i, len, test.wantLen)
}
}
}

func TestNewListStream(t *testing.T) {
ls := NewListStream(bytes.NewReader(unhex("0101010101")), 3)
if k, size, err := ls.Kind(); k != List || size != 3 || err != nil {
t.Errorf("Kind() returned (%v, %d, %v), expected (List, 3, nil)", k, size, err)
}
if size, err := ls.List(); size != 3 || err != nil {
t.Errorf("List() returned (%d, %v), expected (3, nil)", size, err)
}
```

```
for i := 0; i < 3; i++ {
if val, err := ls.Uint(); val != 1 || err != nil {
t.Errorf("Uint() returned (%d, %v), expected (1, nil)", val, err)
}
}
if err := ls.ListEnd(); err != nil {
t.Errorf("ListEnd() returned %v, expected (3, nil)", err)
}
}

func TestStreamErrors(t *testing.T) {
withoutInputLimit := func(b []byte) *Stream {
return NewStream(newPlainReader(b), 0)
}
withCustomInputLimit := func(limit uint64) func([]byte) *Stream {
return func(b []byte) *Stream {
return NewStream(bytes.NewReader(b), limit)
}
}

type calls []string
tests := []struct {
string
calls
newStream func([]byte) *Stream // uses bytes.Reader if nil
error     error
}{
{"C0", calls{"Bytes"}, nil, ErrExpectedString},
{"C0", calls{"Uint"}, nil, ErrExpectedString},
{"89000000000000000001", calls{"Uint"}, nil, errUintOverflow},
{"00", calls{"List"}, nil, ErrExpectedList},
{"80", calls{"List"}, nil, ErrExpectedList},
{"C0", calls{"List", "Uint"}, nil, EOL},
{"C8C9010101010101010101", calls{"List", "Kind"}, nil, ErrElemTooLarge},
{"C3C2010201", calls{"List", "List", "Uint", "Uint", "ListEnd", "Uint"}, nil, EOL},
{"00", calls{"ListEnd"}, nil, errNotInList},
{"C401020304", calls{"List", "Uint", "ListEnd"}, nil, errNotAtEOL},

// Non-canonical integers (e.g. leading zero bytes).
{"00", calls{"Uint"}, nil, ErrCanonInt},
{"820002", calls{"Uint"}, nil, ErrCanonInt},
{"8133", calls{"Uint"}, nil, ErrCanonSize},
{"817F", calls{"Uint"}, nil, ErrCanonSize},
{"8180", calls{"Uint"}, nil, nil},

// Non-valid boolean
{"02", calls{"Bool"}, nil, errors.New("rlp: invalid boolean value: 2")},

// Size tags must use the smallest possible encoding.
// Leading zero bytes in the size tag are also rejected.
{"8100", calls{"Uint"}, nil, ErrCanonSize},
{"8100", calls{"Bytes"}, nil, ErrCanonSize},
{"8101", calls{"Bytes"}, nil, ErrCanonSize},
{"817F", calls{"Bytes"}, nil, ErrCanonSize},
{"8180", calls{"Bytes"}, nil, nil},
{"B800", calls{"Kind"}, withoutInputLimit, ErrCanonSize},
{"B90000", calls{"Kind"}, withoutInputLimit, ErrCanonSize},
{"B90055", calls{"Kind"}, withoutInputLimit, ErrCanonSize},
{"BA0002FFFF", calls{"Bytes"}, withoutInputLimit, ErrCanonSize},
{"F800", calls{"Kind"}, withoutInputLimit, ErrCanonSize},
{"F90000", calls{"Kind"}, withoutInputLimit, ErrCanonSize},
{"F90055", calls{"Kind"}, withoutInputLimit, ErrCanonSize},
{"FA0002FFFF", calls{"List"}, withoutInputLimit, ErrCanonSize},

// Expected EOF
{"", calls{"Kind"}, nil, io.EOF},
{"", calls{"Uint"}, nil, io.EOF},
{"", calls{"List"}, nil, io.EOF},
{"8180", calls{"Uint", "Uint"}, nil, io.EOF},
{"C0", calls{"List", "ListEnd", "List"}, nil, io.EOF},

{"", calls{"List"}, withoutInputLimit, io.EOF},
{"8180", calls{"Uint", "Uint"}, withoutInputLimit, io.EOF},
{"C0", calls{"List", "ListEnd", "List"}, withoutInputLimit, io.EOF},

// Input limit errors.
{"81", calls{"Bytes"}, nil, ErrValueTooLarge},
{"81", calls{"Uint"}, nil, ErrValueTooLarge},
{"81", calls{"Raw"}, nil, ErrValueTooLarge},
{"BFFFFFFFFFFFFFFFFFFF", calls{"Bytes"}, nil, ErrValueTooLarge},
{"C801", calls{"List"}, nil, ErrValueTooLarge},

// Test for list element size check overflow.
{"CD04040404FFFFFFFFFFFFFFFFFF0303", calls{"List", "Uint", "Uint", "Uint", "Uint", "List"}, nil,
ErrElemTooLarge},

// Test for input limit overflow. Since we are counting the limit
// down toward zero in Stream.remaining, reading too far can overflow
// remaining to a large value, effectively disabling the limit.
{"C40102030401", calls{"Raw", "Uint"}, withCustomInputLimit(5), io.EOF},
{"C4010203048180", calls{"Raw", "Uint"}, withCustomInputLimit(6), ErrValueTooLarge},

// Check that the same calls are fine without a limit.
{"C40102030401", calls{"Raw", "Uint"}, withoutInputLimit, nil},
{"C4010203048180", calls{"Raw", "Uint"}, withoutInputLimit, nil},

// Unexpected EOF. This only happens when there is
// no input limit, so the reader needs to be 'dumbed down'.
{"81", calls{"Bytes"}, withoutInputLimit, io.ErrUnexpectedEOF},
{"81", calls{"Uint"}, withoutInputLimit, io.ErrUnexpectedEOF},
{"BFFFFFFFFFFFFFFF", calls{"Bytes"}, withoutInputLimit, io.ErrUnexpectedEOF},
{"C801", calls{"List", "Uint", "Uint"}, withoutInputLimit, io.ErrUnexpectedEOF},

// This test verifies that the input position is advanced
// correctly when calling Bytes for empty strings. Kind can be called
// any number of times in between and doesn't advance.
{"C3808080", calls{
"List",  // enter the list
"Bytes", // past first element

"Kind", "Kind", "Kind", // this shouldn't advance

"Bytes", // past second element

"Kind", "Kind", // can't hurt to try

"Bytes", // past final element
"Bytes", // this one should fail
}, nil, EOL},
}

testfor:
for i, test := range tests {
if test.newStream == nil {
test.newStream = func(b []byte) *Stream { return NewStream(bytes.NewReader(b), 0) }
}
```

```
s := test.newStream(unhex(test.string))
rs := reflect.ValueOf(s)
for j, call := range test.calls {
fval := rs.MethodByName(call)
ret := fval.Call(nil)
err := "<nil>"
if lastret := ret[len(ret)-1].Interface(); lastret != nil {
err = lastret.(error).Error()
}
if j == len(test.calls)-1 {
want := "<nil>"
if test.error != nil {
want = test.error.Error()
}
if err != want {
t.Log(test)
t.Errorf("test %d: last call (%s) error mismatch\ngot:  %s\nwant: %s",
i, call, err, test.error)
}
} else if err != "<nil>" {
t.Log(test)
t.Errorf("test %d: call %d (%s) unexpected error: %q", i, j, call, err)
continue testfor
}
}
}
}

func TestStreamList(t *testing.T) {
s := NewStream(bytes.NewReader(unhex("C80102030405060708")), 0)

len, err := s.List()
if err != nil {
t.Fatalf("List error: %v", err)
}
if len != 8 {
t.Fatalf("List returned invalid length, got %d, want 8", len)
}

for i := uint64(1); i <= 8; i++ {
v, err := s.Uint()
if err != nil {
```

```
t.Fatalf("Uint error: %v", err)
}
if i != v {
t.Errorf("Uint returned wrong value, got %d, want %d", v, i)
}
}

if _, err := s.Uint(); err != EOL {
t.Errorf("Uint error mismatch, got %v, want %v", err, EOL)
}
if err = s.ListEnd(); err != nil {
t.Fatalf("ListEnd error: %v", err)
}
}

func TestStreamRaw(t *testing.T) {
s := NewStream(bytes.NewReader(unhex("C58401010101")), 0)
s.List()

want := unhex("8401010101")
raw, err := s.Raw()
if err != nil {
t.Fatal(err)
}
if !bytes.Equal(want, raw) {
t.Errorf("raw mismatch: got %x, want %x", raw, want)
}
}

func TestDecodeErrors(t *testing.T) {
r := bytes.NewReader(nil)

if err := Decode(r, nil); err != errDecodeIntoNil {
t.Errorf("Decode(r, nil) error mismatch, got %q, want %q", err, errDecodeIntoNil)
}

var nilptr *struct{}
if err := Decode(r, nilptr); err != errDecodeIntoNil {
t.Errorf("Decode(r, nilptr) error mismatch, got %q, want %q", err, errDecodeIntoNil)
}

if err := Decode(r, struct{}{}); err != errNoPointer {
```

```
t.Errorf("Decode(r, struct{}{}) error mismatch, got %q, want %q", err, errNoPointer)
}

expectErr := "rlp: type chan bool is not RLP-serializable"
if err := Decode(r, new(chan bool)); err == nil || err.Error() != expectErr {
t.Errorf("Decode(r, new(chan bool)) error mismatch, got %q, want %q", err, expectErr)
}

if err := Decode(r, new(uint)); err != io.EOF {
t.Errorf("Decode(r, new(int)) error mismatch, got %q, want %q", err, io.EOF)
}
}

type decodeTest struct {
input string
ptr   interface{}
value interface{}
error string
}

type simplestruct struct {
A uint
B string
}

type recstruct struct {
I     uint
Child *recstruct `rlp:"nil"`
}

type invalidTail1 struct {
A uint `rlp:"tail"`
B string
}

type invalidTail2 struct {
A uint
B string `rlp:"tail"`
}

type tailRaw struct {
A    uint
```

```
Tail []RawValue `rlp:"tail"`
}

type tailUint struct {
A    uint
Tail []uint `rlp:"tail"`
}

var (
veryBigInt = big.NewInt(0).Add(
big.NewInt(0).Lsh(big.NewInt(0xFFFFFFFFFFFFFF), 16),
big.NewInt(0xFFFF),
)
)

type hasIgnoredField struct {
A uint
B uint `rlp:"-"`
C uint
}

var decodeTests = []decodeTest{
// booleans
{input: "01", ptr: new(bool), value: true},
{input: "80", ptr: new(bool), value: false},
{input: "02", ptr: new(bool), error: "rlp: invalid boolean value: 2"},

// integers
{input: "05", ptr: new(uint32), value: uint32(5)},
{input: "80", ptr: new(uint32), value: uint32(0)},
{input: "820505", ptr: new(uint32), value: uint32(0x0505)},
{input: "83050505", ptr: new(uint32), value: uint32(0x050505)},
{input: "8405050505", ptr: new(uint32), value: uint32(0x05050505)},
{input: "850505050505", ptr: new(uint32), error: "rlp: input string too long for uint32"},
{input: "C0", ptr: new(uint32), error: "rlp: expected input string or byte for uint32"},
{input: "00", ptr: new(uint32), error: "rlp: non-canonical integer (leading zero bytes) for uint32"},
{input: "8105", ptr: new(uint32), error: "rlp: non-canonical size information for uint32"},
{input: "820004", ptr: new(uint32), error: "rlp: non-canonical integer (leading zero bytes) for
uint32"},
{input: "B8020004", ptr: new(uint32), error: "rlp: non-canonical size information for uint32"},

// slices
```

```
{input: "C0", ptr: new([]uint), value: []uint{}},
{input: "C80102030405060708", ptr: new([]uint), value: []uint{1, 2, 3, 4, 5, 6, 7, 8}},
{input: "F8020004", ptr: new([]uint), error: "rlp: non-canonical size information for []uint"},

// arrays
{input: "C50102030405", ptr: new([5]uint), value: [5]uint{1, 2, 3, 4, 5}},
{input: "C0", ptr: new([5]uint), error: "rlp: input list has too few elements for [5]uint"},
{input: "C102", ptr: new([5]uint), error: "rlp: input list has too few elements for [5]uint"},
{input: "C6010203040506", ptr: new([5]uint), error: "rlp: input list has too many elements for
[5]uint"},
{input: "F8020004", ptr: new([5]uint), error: "rlp: non-canonical size information for [5]uint"},

// zero sized arrays
{input: "C0", ptr: new([0]uint), value: [0]uint{}},
{input: "C101", ptr: new([0]uint), error: "rlp: input list has too many elements for [0]uint"},

// byte slices
{input: "01", ptr: new([]byte), value: []byte{1}},
{input: "80", ptr: new([]byte), value: []byte{}},
{input: "8D6162636465666768696A6B6C6D", ptr: new([]byte), value: []byte("abcdefghijklm")},
{input: "C0", ptr: new([]byte), error: "rlp: expected input string or byte for []uint8"},
{input: "8105", ptr: new([]byte), error: "rlp: non-canonical size information for []uint8"},

// byte arrays
{input: "02", ptr: new([1]byte), value: [1]byte{2}},
{input: "8180", ptr: new([1]byte), value: [1]byte{128}},
{input: "850102030405", ptr: new([5]byte), value: [5]byte{1, 2, 3, 4, 5}},

// byte array errors
{input: "02", ptr: new([5]byte), error: "rlp: input string too short for [5]uint8"},
{input: "80", ptr: new([5]byte), error: "rlp: input string too short for [5]uint8"},
{input: "820000", ptr: new([5]byte), error: "rlp: input string too short for [5]uint8"},
{input: "C0", ptr: new([5]byte), error: "rlp: expected input string or byte for [5]uint8"},
{input: "C3010203", ptr: new([5]byte), error: "rlp: expected input string or byte for [5]uint8"},
{input: "86010203040506", ptr: new([5]byte), error: "rlp: input string too long for [5]uint8"},
{input: "8105", ptr: new([1]byte), error: "rlp: non-canonical size information for [1]uint8"},
{input: "817F", ptr: new([1]byte), error: "rlp: non-canonical size information for [1]uint8"},

// zero sized byte arrays
{input: "80", ptr: new([0]byte), value: [0]byte{}},
{input: "01", ptr: new([0]byte), error: "rlp: input string too long for [0]uint8"},
{input: "8101", ptr: new([0]byte), error: "rlp: input string too long for [0]uint8"},
// strings
{input: "00", ptr: new(string), value: "\000"},
{input: "8D6162636465666768696A6B6C6D", ptr: new(string), value: "abcdefghijklm"},
{input: "C0", ptr: new(string), error: "rlp: expected input string or byte for string"},

// big ints
{input: "01", ptr: new(*big.Int), value: big.NewInt(1)},
{input: "89FFFFFFFFFFFFFFFFFF", ptr: new(*big.Int), value: veryBigInt},
{input: "10", ptr: new(big.Int), value: *big.NewInt(16)}, // non-pointer also works
{input: "C0", ptr: new(*big.Int), error: "rlp: expected input string or byte for *big.Int"},
{input: "820001", ptr: new(big.Int), error: "rlp: non-canonical integer (leading zero bytes) for
*big.Int"},
{input: "8105", ptr: new(big.Int), error: "rlp: non-canonical size information for *big.Int"},

// structs
{
input: "C50583343434",
ptr:   new(simplestruct),
value: simplestruct{5, "444"},
},
{
input: "C601C402C203C0",
ptr:   new(recstruct),
value: recstruct{1, &recstruct{2, &recstruct{3, nil}}},
},

// struct errors
{
input: "C0",
ptr:   new(simplestruct),
error: "rlp: too few elements for rlp.simplestruct",
},
{
input: "C105",
ptr:   new(simplestruct),
error: "rlp: too few elements for rlp.simplestruct",
},
{
input: "C7C50583343434C0",
ptr:   new([]*simplestruct),
error: "rlp: too few elements for rlp.simplestruct, decoding into ([]*rlp.simplestruct)[1]",
},
{
input: "83222222",
ptr:   new(simplestruct),
error: "rlp: expected input list for rlp.simplestruct",
},
{
input: "C3010101",
ptr:   new(simplestruct),
error: "rlp: input list has too many elements for rlp.simplestruct",
},
{
input: "C501C3C00000",
ptr:   new(recstruct),
error: "rlp: expected input string or byte for uint, decoding into (rlp.recstruct).Child.I",
},
{
input: "C0",
ptr:   new(invalidTail1),
error: "rlp: invalid struct tag \"tail\" for rlp.invalidTail1.A (must be on last field)",
},
{
input: "C0",
ptr:   new(invalidTail2),
error: "rlp: invalid struct tag \"tail\" for rlp.invalidTail2.B (field type is not slice)",
},
{
input: "C50102C20102",
ptr:   new(tailUint),
error: "rlp: expected input string or byte for uint, decoding into (rlp.tailUint).Tail[1]",
},

// struct tag "tail"
{
input: "C3010203",
ptr:   new(tailRaw),
value: tailRaw{A: 1, Tail: []RawValue{unhex("02"), unhex("03")}},
},
{
input: "C20102",
ptr:   new(tailRaw),
value: tailRaw{A: 1, Tail: []RawValue{unhex("02")}},
},
{
input: "C101",
ptr:   new(tailRaw),
value: tailRaw{A: 1, Tail: []RawValue{}},
},

// struct tag "-"
{
input: "C20102",
ptr:   new(hasIgnoredField),
value: hasIgnoredField{A: 1, C: 2},
},

// RawValue
{input: "01", ptr: new(RawValue), value: RawValue(unhex("01"))},
{input: "82FFFF", ptr: new(RawValue), value: RawValue(unhex("82FFFF"))},
{input: "C20102", ptr: new([]RawValue), value: []RawValue{unhex("01"), unhex("02")}},

// pointers
{input: "00", ptr: new(*[]byte), value: &[]byte{0}},
{input: "80", ptr: new(*uint), value: uintp(0)},
{input: "C0", ptr: new(*uint), error: "rlp: expected input string or byte for uint"},
{input: "07", ptr: new(*uint), value: uintp(7)},
{input: "817F", ptr: new(*uint), error: "rlp: non-canonical size information for uint"},
{input: "8180", ptr: new(*uint), value: uintp(0x80)},
{input: "C109", ptr: new(*[]uint), value: &[]uint{9}},
{input: "C58403030303", ptr: new(*[][]byte), value: &[][]byte{{3, 3, 3, 3}}},

// check that input position is advanced also for empty values.
{input: "C3808005", ptr: new([]*uint), value: []*uint{uintp(0), uintp(0), uintp(5)}},

// interface{}
{input: "00", ptr: new(interface{}), value: []byte{0}},
{input: "01", ptr: new(interface{}), value: []byte{1}},
{input: "80", ptr: new(interface{}), value: []byte{}},
{input: "850505050505", ptr: new(interface{}), value: []byte{5, 5, 5, 5, 5}},
{input: "C0", ptr: new(interface{}), value: []interface{}{}},
{input: "C50183040404", ptr: new(interface{}), value: []interface{}{[]byte{1}, []byte{4, 4, 4}}},
{
input: "C3010203",
ptr:   new([]io.Reader),
error: "rlp: type io.Reader is not RLP-serializable",
},

// fuzzer crashes
{
input: "c330f9c030f93030ce3030303030303030bd303030303030",
ptr:   new(interface{}),
error: "rlp: element is larger than containing list",
},
}

func uintp(i uint) *uint { return &i }

func runTests(t *testing.T, decode func([]byte, interface{}) error) {
for i, test := range decodeTests {
input, err := hex.DecodeString(test.input)
if err != nil {
t.Errorf("test %d: invalid hex input %q", i, test.input)
continue
}
err = decode(input, test.ptr)
if err != nil && test.error == "" {
t.Errorf("test %d: unexpected Decode error: %v\ndecoding into %T\ninput %q",
i, err, test.ptr, test.input)
continue
}
if test.error != "" && fmt.Sprint(err) != test.error {
t.Errorf("test %d: Decode error mismatch\ngot  %v\nwant %v\ndecoding into %T\ninput %q",
i, err, test.error, test.ptr, test.input)
continue
}
deref := reflect.ValueOf(test.ptr).Elem().Interface()
if err == nil && !reflect.DeepEqual(deref, test.value) {
t.Errorf("test %d: value mismatch\ngot  %#v\nwant %#v\ndecoding into %T\ninput %q",
i, deref, test.value, test.ptr, test.input)
}
}
}

func TestDecodeWithByteReader(t *testing.T) {
runTests(t, func(input []byte, into interface{}) error {
return Decode(bytes.NewReader(input), into)
})
}

// plainReader reads from a byte slice but does not
// implement ReadByte. It is also not recognized by the
// size validation. This is useful to test how the decoder
// behaves on a non-buffered input stream.
type plainReader []byte

func newPlainReader(b []byte) io.Reader {
return (*plainReader)(&b)
}

func (r *plainReader) Read(buf []byte) (n int, err error) {
if len(*r) == 0 {
return 0, io.EOF
}
n = copy(buf, *r)
*r = (*r)[n:]
return n, nil
}

func TestDecodeWithNonByteReader(t *testing.T) {
runTests(t, func(input []byte, into interface{}) error {
return Decode(newPlainReader(input), into)
})
}

func TestDecodeStreamReset(t *testing.T) {
s := NewStream(nil, 0)
runTests(t, func(input []byte, into interface{}) error {
s.Reset(bytes.NewReader(input), 0)
return s.Decode(into)
})
}

type testDecoder struct{ called bool }

func (t *testDecoder) DecodeRLP(s *Stream) error {
if _, err := s.Uint(); err != nil {
return err
}
```

```
t.called = true
return nil
}

func TestDecodeDecoder(t *testing.T) {
var s struct {
T1 testDecoder
T2 *testDecoder
T3 **testDecoder
}
if err := Decode(bytes.NewReader(unhex("C3010203")), &s); err != nil {
t.Fatalf("Decode error: %v", err)
}

if !s.T1.called {
t.Errorf("DecodeRLP was not called for (non-pointer) testDecoder")
}

if s.T2 == nil {
t.Errorf("*testDecoder has not been allocated")
} else if !s.T2.called {
t.Errorf("DecodeRLP was not called for *testDecoder")
}

if s.T3 == nil || *s.T3 == nil {
t.Errorf("**testDecoder has not been allocated")
} else if !(*s.T3).called {
t.Errorf("DecodeRLP was not called for **testDecoder")
}
}

type byteDecoder byte

func (bd *byteDecoder) DecodeRLP(s *Stream) error {
_, err := s.Uint()
*bd = 255
return err
}

func (bd byteDecoder) called() bool {
return bd == 255
}
```

```
// This test verifies that the byte slice/byte array logic
// does not kick in for element types implementing Decoder.
func TestDecoderInByteSlice(t *testing.T) {
var slice []byteDecoder
if err := Decode(bytes.NewReader(unhex("C101")), &slice); err != nil {
t.Errorf("unexpected Decode error %v", err)
} else if !slice[0].called() {
t.Errorf("DecodeRLP not called for slice element")
}

var array [1]byteDecoder
if err := Decode(bytes.NewReader(unhex("C101")), &array); err != nil {
t.Errorf("unexpected Decode error %v", err)
} else if !array[0].called() {
t.Errorf("DecodeRLP not called for array element")
}
}

func ExampleDecode() {
input, _ := hex.DecodeString("C90A1486666F6F626172")

type example struct {
A, B    uint
private uint // private fields are ignored
String  string
}

var s example
err := Decode(bytes.NewReader(input), &s)
if err != nil {
fmt.Printf("Error: %v\n", err)
} else {
fmt.Printf("Decoded value: %#v\n", s)
}
// Output:
// Decoded value: rlp.example{A:0xa, B:0x14, private:0x0, String:"foobar"}
}

func ExampleDecode_structTagNil() {
// In this example, we'll use the "nil" struct tag to change
// how a pointer-typed field is decoded. The input contains an RLP
// list of one element, an empty string.
input := []byte{0xC1, 0x80}

// This type uses the normal rules.
// The empty input string is decoded as a pointer to an empty Go string.
var normalRules struct {
String *string
}
Decode(bytes.NewReader(input), &normalRules)
fmt.Printf("normal: String = %q\n", *normalRules.String)

// This type uses the struct tag.
// The empty input string is decoded as a nil pointer.
var withEmptyOK struct {
String *string `rlp:"nil"`
}
Decode(bytes.NewReader(input), &withEmptyOK)
fmt.Printf("with nil tag: String = %v\n", withEmptyOK.String)

// Output:
// normal: String = ""
// with nil tag: String = <nil>
}

func ExampleStream() {
input, _ := hex.DecodeString("C90A1486666F6F626172")
s := NewStream(bytes.NewReader(input), 0)

// Check what kind of value lies ahead
kind, size, _ := s.Kind()
fmt.Printf("Kind: %v size:%d\n", kind, size)

// Enter the list
if _, err := s.List(); err != nil {
fmt.Printf("List error: %v\n", err)
return
}

// Decode elements
fmt.Println(s.Uint())
fmt.Println(s.Uint())
fmt.Println(s.Bytes())
```

```
// Acknowledge end of list
if err := s.ListEnd(); err != nil {
fmt.Printf("ListEnd error: %v\n", err)
}
// Output:
// Kind: List size:9
// 10 <nil>
// 20 <nil>
// [102 111 111 98 97 114] <nil>
}

func BenchmarkDecode(b *testing.B) {
enc := encodeTestSlice(90000)
b.SetBytes(int64(len(enc)))
b.ReportAllocs()
b.ResetTimer()

for i := 0; i < b.N; i++ {
var s []uint
r := bytes.NewReader(enc)
if err := Decode(r, &s); err != nil {
b.Fatalf("Decode error: %v", err)
}
}
}

func BenchmarkDecodeIntSliceReuse(b *testing.B) {
enc := encodeTestSlice(100000)
b.SetBytes(int64(len(enc)))
b.ReportAllocs()
b.ResetTimer()

var s []uint
for i := 0; i < b.N; i++ {
r := bytes.NewReader(enc)
if err := Decode(r, &s); err != nil {
b.Fatalf("Decode error: %v", err)
}
}
}
```

```
func encodeTestSlice(n uint) []byte {
s := make([]uint, n)
for i := uint(0); i < n; i++ {
s[i] = i
}
b, err := EncodeToBytes(s)
if err != nil {
panic(fmt.Sprintf("encode error: %v", err))
}
return b
}

func unhex(str string) []byte {
b, err := hex.DecodeString(strings.Replace(str, " ", "", -1))
if err != nil {
panic(fmt.Sprintf("invalid hex string: %q", str))
}
return b
}
```

38:F:\git\coin\ethereum\go-ethereum\rlp\doc.go

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

/*
Package rlp implements the RLP serialization format.

The purpose of RLP (Recursive Linear Prefix) is to encode arbitrarily
nested arrays of binary data, and RLP is the main encoding method used
to serialize objects in Ethereum. The only purpose of RLP is to encode
structure; encoding specific atomic data types (eg. strings, ints,
floats) is left up to higher-order protocols; in Ethereum integers
must be represented in big endian binary form with no leading zeroes
(thus making the integer value zero equivalent to the empty byte
array).

RLP values are distinguished by a type tag. The type tag precedes the
value in the input stream and defines the size and kind of the bytes
that follow.
*/
package rlp
```

39:F:\git\coin\ethereum\go-ethereum\rlp\encode.go

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

package rlp

import (
"fmt"
"io"
"math/big"
"reflect"
"sync"
)

var (
// Common encoded values.
// These are useful when implementing EncodeRLP.
EmptyString = []byte{0x80}
EmptyList   = []byte{0xC0}
)

// Encoder is implemented by types that require custom
// encoding rules or want to encode private fields.
type Encoder interface {
// EncodeRLP should write the RLP encoding of its receiver to w.
// If the implementation is a pointer method, it may also be
// called for nil pointers.
//
// Implementations should generate valid RLP. The data written is
// not verified at the moment, but a future version might. It is
// recommended to write only a single value but writing multiple
// values or no value at all is also permitted.
EncodeRLP(io.Writer) error
}

// Encode writes the RLP encoding of val to w. Note that Encode may
// perform many small writes in some cases. Consider making w
// buffered.
//
// Encode uses the following type-dependent encoding rules:
//
// If the type implements the Encoder interface, Encode calls
// EncodeRLP. This is true even for nil pointers, please see the
// documentation for Encoder.
```

```
//
// To encode a pointer, the value being pointed to is encoded. For nil
// pointers, Encode will encode the zero value of the type. A nil
// pointer to a struct type always encodes as an empty RLP list.
// A nil pointer to an array encodes as an empty list (or empty string
// if the array has element type byte).
//
// Struct values are encoded as an RLP list of all their encoded
// public fields. Recursive struct types are supported.
//
// To encode slices and arrays, the elements are encoded as an RLP
// list of the value's elements. Note that arrays and slices with
// element type uint8 or byte are always encoded as an RLP string.
//
// A Go string is encoded as an RLP string.
//
// An unsigned integer value is encoded as an RLP string. Zero always
// encodes as an empty RLP string. Encode also supports *big.Int.
//
// An interface value encodes as the value contained in the interface.
//
// Boolean values are not supported, nor are signed integers, floating
// point numbers, maps, channels and functions.
func Encode(w io.Writer, val interface{}) error {
if outer, ok := w.(*encbuf); ok {
// Encode was called by some type's EncodeRLP.
// Avoid copying by writing to the outer encbuf directly.
return outer.encode(val)
}
eb := encbufPool.Get().(*encbuf)
defer encbufPool.Put(eb)
eb.reset()
if err := eb.encode(val); err != nil {
return err
}
return eb.toWriter(w)
}

// EncodeBytes returns the RLP encoding of val.
// Please see the documentation of Encode for the encoding rules.
func EncodeToBytes(val interface{}) ([]byte, error) {
eb := encbufPool.Get().(*encbuf)
defer encbufPool.Put(eb)
eb.reset()
if err := eb.encode(val); err != nil {
return nil, err
}
return eb.toBytes(), nil
}

// EncodeReader returns a reader from which the RLP encoding of val
// can be read. The returned size is the total size of the encoded
// data.
//
// Please see the documentation of Encode for the encoding rules.
func EncodeToReader(val interface{}) (size int, r io.Reader, err error) {
eb := encbufPool.Get().(*encbuf)
eb.reset()
if err := eb.encode(val); err != nil {
return 0, nil, err
}
return eb.size(), &encReader{buf: eb}, nil
}

type encbuf struct {
str     []byte      // string data, contains everything except list headers
lheads  []*listhead // all list headers
lhsize  int         // sum of sizes of all encoded list headers
sizebuf []byte      // 9-byte auxiliary buffer for uint encoding
}

type listhead struct {
offset int // index of this header in string data
size   int // total size of encoded data (including list headers)
}

// encode writes head to the given buffer, which must be at least
// 9 bytes long. It returns the encoded bytes.
func (head *listhead) encode(buf []byte) []byte {
return buf[:puthead(buf, 0xC0, 0xF7, uint64(head.size))]
}

// headsize returns the size of a list or string header
// for a value of the given size.
```

```
func headsize(size uint64) int {
if size < 56 {
return 1
}
return 1 + intsize(size)
}

// puthead writes a list or string header to buf.
// buf must be at least 9 bytes long.
func puthead(buf []byte, smalltag, largetag byte, size uint64) int {
if size < 56 {
buf[0] = smalltag + byte(size)
return 1
} else {
sizesize := putint(buf[1:], size)
buf[0] = largetag + byte(sizesize)
return sizesize + 1
}
}

// encbufs are pooled.
var encbufPool = sync.Pool{
New: func() interface{} { return &encbuf{sizebuf: make([]byte, 9)} },
}

func (w *encbuf) reset() {
w.lhsize = 0
if w.str != nil {
w.str = w.str[:0]
}
if w.lheads != nil {
w.lheads = w.lheads[:0]
}
}

// encbuf implements io.Writer so it can be passed it into EncodeRLP.
func (w *encbuf) Write(b []byte) (int, error) {
w.str = append(w.str, b...)
return len(b), nil
}

func (w *encbuf) encode(val interface{}) error {
```

```
rval := reflect.ValueOf(val)
ti, err := cachedTypeInfo(rval.Type(), tags{})
if err != nil {
return err
}
return ti.writer(rval, w)
}

func (w *encbuf) encodeStringHeader(size int) {
if size < 56 {
w.str = append(w.str, 0x80+byte(size))
} else {
// TODO: encode to w.str directly
sizesize := putint(w.sizebuf[1:], uint64(size))
w.sizebuf[0] = 0xB7 + byte(sizesize)
w.str = append(w.str, w.sizebuf[:sizesize+1]...)
}
}

func (w *encbuf) encodeString(b []byte) {
if len(b) == 1 && b[0] <= 0x7F {
// fits single byte, no string header
w.str = append(w.str, b[0])
} else {
w.encodeStringHeader(len(b))
w.str = append(w.str, b...)
}
}

func (w *encbuf) list() *listhead {
lh := &listhead{offset: len(w.str), size: w.lhsize}
w.lheads = append(w.lheads, lh)
return lh
}

func (w *encbuf) listEnd(lh *listhead) {
lh.size = w.size() - lh.offset - lh.size
if lh.size < 56 {
w.lhsize += 1 // length encoded into kind tag
} else {
w.lhsize += 1 + intsize(uint64(lh.size))
}
```

```
}

func (w *encbuf) size() int {
return len(w.str) + w.lhsize
}

func (w *encbuf) toBytes() []byte {
out := make([]byte, w.size())
strpos := 0
pos := 0
for _, head := range w.lheads {
// write string data before header
n := copy(out[pos:], w.str[strpos:head.offset])
pos += n
strpos += n
// write the header
enc := head.encode(out[pos:])
pos += len(enc)
}
// copy string data after the last list header
copy(out[pos:], w.str[strpos:])
return out
}

func (w *encbuf) toWriter(out io.Writer) (err error) {
strpos := 0
for _, head := range w.lheads {
// write string data before header
if head.offset-strpos > 0 {
n, err := out.Write(w.str[strpos:head.offset])
strpos += n
if err != nil {
return err
}
}
// write the header
enc := head.encode(w.sizebuf)
if _, err = out.Write(enc); err != nil {
return err
}
}
if strpos < len(w.str) {
```

```
// write string data after the last list header
_, err = out.Write(w.str[strpos:])
}
return err
}

// encReader is the io.Reader returned by EncodeToReader.
// It releases its encbuf at EOF.
type encReader struct {
buf    *encbuf // the buffer we're reading from. this is nil when we're at EOF.
lhpos  int     // index of list header that we're reading
strpos int     // current position in string buffer
piece  []byte  // next piece to be read
}

func (r *encReader) Read(b []byte) (n int, err error) {
for {
if r.piece = r.next(); r.piece == nil {
// Put the encode buffer back into the pool at EOF when it
// is first encountered. Subsequent calls still return EOF
// as the error but the buffer is no longer valid.
if r.buf != nil {
encbufPool.Put(r.buf)
r.buf = nil
}
return n, io.EOF
}
nn := copy(b[n:], r.piece)
n += nn
if nn < len(r.piece) {
// piece didn't fit, see you next time.
r.piece = r.piece[nn:]
return n, nil
}
r.piece = nil
}
}

// next returns the next piece of data to be read.
// it returns nil at EOF.
func (r *encReader) next() []byte {
switch {
```

```
case r.buf == nil:
return nil

case r.piece != nil:
// There is still data available for reading.
return r.piece

case r.lhpos < len(r.buf.lheads):
// We're before the last list header.
head := r.buf.lheads[r.lhpos]
sizebefore := head.offset - r.strpos
if sizebefore > 0 {
// String data before header.
p := r.buf.str[r.strpos:head.offset]
r.strpos += sizebefore
return p
} else {
r.lhpos++
return head.encode(r.buf.sizebuf)
}

case r.strpos < len(r.buf.str):
// String data at the end, after all list headers.
p := r.buf.str[r.strpos:]
r.strpos = len(r.buf.str)
return p

default:
return nil
}
}

var (
encoderInterface = reflect.TypeOf(new(Encoder)).Elem()
big0             = big.NewInt(0)
)

// makeWriter creates a writer function for the given type.
func makeWriter(typ reflect.Type, ts tags) (writer, error) {
kind := typ.Kind()
switch {
case typ == rawValueType:
```

```
return writeRawValue, nil
case typ.Implements(encoderInterface):
return writeEncoder, nil
case kind != reflect.Ptr && reflect.PtrTo(typ).Implements(encoderInterface):
return writeEncoderNoPtr, nil
case kind == reflect.Interface:
return writeInterface, nil
case typ.AssignableTo(reflect.PtrTo(bigInt)):
return writeBigIntPtr, nil
case typ.AssignableTo(bigInt):
return writeBigIntNoPtr, nil
case isUint(kind):
return writeUint, nil
case kind == reflect.Bool:
return writeBool, nil
case kind == reflect.String:
return writeString, nil
case kind == reflect.Slice && isByte(typ.Elem()):
return writeBytes, nil
case kind == reflect.Array && isByte(typ.Elem()):
return writeByteArray, nil
case kind == reflect.Slice || kind == reflect.Array:
return makeSliceWriter(typ, ts)
case kind == reflect.Struct:
return makeStructWriter(typ)
case kind == reflect.Ptr:
return makePtrWriter(typ)
default:
return nil, fmt.Errorf("rlp: type %v is not RLP-serializable", typ)
}
}

func isByte(typ reflect.Type) bool {
return typ.Kind() == reflect.Uint8 && !typ.Implements(encoderInterface)
}

func writeRawValue(val reflect.Value, w *encbuf) error {
w.str = append(w.str, val.Bytes()...)
return nil
}

func writeUint(val reflect.Value, w *encbuf) error {
```

```
i := val.Uint()
if i == 0 {
w.str = append(w.str, 0x80)
} else if i < 128 {
// fits single byte
w.str = append(w.str, byte(i))
} else {
// TODO: encode int to w.str directly
s := putint(w.sizebuf[1:], i)
w.sizebuf[0] = 0x80 + byte(s)
w.str = append(w.str, w.sizebuf[:s+1]...)
}
return nil
}

func writeBool(val reflect.Value, w *encbuf) error {
if val.Bool() {
w.str = append(w.str, 0x01)
} else {
w.str = append(w.str, 0x80)
}
return nil
}

func writeBigIntPtr(val reflect.Value, w *encbuf) error {
ptr := val.Interface().(*big.Int)
if ptr == nil {
w.str = append(w.str, 0x80)
return nil
}
return writeBigInt(ptr, w)
}

func writeBigIntNoPtr(val reflect.Value, w *encbuf) error {
i := val.Interface().(big.Int)
return writeBigInt(&i, w)
}

func writeBigInt(i *big.Int, w *encbuf) error {
if cmp := i.Cmp(big0); cmp == -1 {
return fmt.Errorf("rlp: cannot encode negative *big.Int")
} else if cmp == 0 {
```

```
w.str = append(w.str, 0x80)
} else {
w.encodeString(i.Bytes())
}
return nil
}

func writeBytes(val reflect.Value, w *encbuf) error {
w.encodeString(val.Bytes())
return nil
}

func writeByteArray(val reflect.Value, w *encbuf) error {
if !val.CanAddr() {
// Slice requires the value to be addressable.
// Make it addressable by copying.
copy := reflect.New(val.Type()).Elem()
copy.Set(val)
val = copy
}
size := val.Len()
slice := val.Slice(0, size).Bytes()
w.encodeString(slice)
return nil
}

func writeString(val reflect.Value, w *encbuf) error {
s := val.String()
if len(s) == 1 && s[0] <= 0x7f {
// fits single byte, no string header
w.str = append(w.str, s[0])
} else {
w.encodeStringHeader(len(s))
w.str = append(w.str, s...)
}
return nil
}

func writeEncoder(val reflect.Value, w *encbuf) error {
return val.Interface().(Encoder).EncodeRLP(w)
}
```

```
// writeEncoderNoPtr handles non-pointer values that implement Encoder
// with a pointer receiver.
func writeEncoderNoPtr(val reflect.Value, w *encbuf) error {
if !val.CanAddr() {
// We can't get the address. It would be possible to make the
// value addressable by creating a shallow copy, but this
// creates other problems so we're not doing it (yet).
//
// package json simply doesn't call MarshalJSON for cases like
// this, but encodes the value as if it didn't implement the
// interface. We don't want to handle it that way.
return fmt.Errorf("rlp: game over: unadressable value of type %v, EncodeRLP is pointer method",
val.Type())
}
return val.Addr().Interface().(Encoder).EncodeRLP(w)
}

func writeInterface(val reflect.Value, w *encbuf) error {
if val.IsNil() {
// Write empty list. This is consistent with the previous RLP
// encoder that we had and should therefore avoid any
// problems.
w.str = append(w.str, 0xC0)
return nil
}
eval := val.Elem()
ti, err := cachedTypeInfo(eval.Type(), tags{})
if err != nil {
return err
}
return ti.writer(eval, w)
}

func makeSliceWriter(typ reflect.Type, ts tags) (writer, error) {
etypeinfo, err := cachedTypeInfo1(typ.Elem(), tags{})
if err != nil {
return nil, err
}
writer := func(val reflect.Value, w *encbuf) error {
if !ts.tail {
defer w.listEnd(w.list())
}
```

```
vlen := val.Len()
for i := 0; i < vlen; i++ {
if err := etypeinfo.writer(val.Index(i), w); err != nil {
return err
}
}
return nil
}
return writer, nil
}

func makeStructWriter(typ reflect.Type) (writer, error) {
fields, err := structFields(typ)
if err != nil {
return nil, err
}
writer := func(val reflect.Value, w *encbuf) error {
lh := w.list()
for _, f := range fields {
if err := f.info.writer(val.Field(f.index), w); err != nil {
return err
}
}
w.listEnd(lh)
return nil
}
return writer, nil
}

func makePtrWriter(typ reflect.Type) (writer, error) {
etypeinfo, err := cachedTypeInfo1(typ.Elem(), tags{})
if err != nil {
return nil, err
}

// determine nil pointer handler
var nilfunc func(*encbuf) error
kind := typ.Elem().Kind()
switch {
case kind == reflect.Array && isByte(typ.Elem().Elem()):
nilfunc = func(w *encbuf) error {
w.str = append(w.str, 0x80)
return nil
}
case kind == reflect.Struct || kind == reflect.Array:
nilfunc = func(w *encbuf) error {
// encoding the zero value of a struct/array could trigger
// infinite recursion, avoid that.
w.listEnd(w.list())
return nil
}
default:
zero := reflect.Zero(typ.Elem())
nilfunc = func(w *encbuf) error {
return etypeinfo.writer(zero, w)
}
}

writer := func(val reflect.Value, w *encbuf) error {
if val.IsNil() {
return nilfunc(w)
} else {
return etypeinfo.writer(val.Elem(), w)
}
}
return writer, err
}

// putint writes i to the beginning of b in big endian byte
// order, using the least number of bytes needed to represent i.
func putint(b []byte, i uint64) (size int) {
switch {
case i < (1 << 8):
b[0] = byte(i)
return 1
case i < (1 << 16):
b[0] = byte(i >> 8)
b[1] = byte(i)
return 2
case i < (1 << 24):
b[0] = byte(i >> 16)
b[1] = byte(i >> 8)
b[2] = byte(i)
return 3
```

```
case i < (1 << 32):
b[0] = byte(i >> 24)
b[1] = byte(i >> 16)
b[2] = byte(i >> 8)
b[3] = byte(i)
return 4
case i < (1 << 40):
b[0] = byte(i >> 32)
b[1] = byte(i >> 24)
b[2] = byte(i >> 16)
b[3] = byte(i >> 8)
b[4] = byte(i)
return 5
case i < (1 << 48):
b[0] = byte(i >> 40)
b[1] = byte(i >> 32)
b[2] = byte(i >> 24)
b[3] = byte(i >> 16)
b[4] = byte(i >> 8)
b[5] = byte(i)
return 6
case i < (1 << 56):
b[0] = byte(i >> 48)
b[1] = byte(i >> 40)
b[2] = byte(i >> 32)
b[3] = byte(i >> 24)
b[4] = byte(i >> 16)
b[5] = byte(i >> 8)
b[6] = byte(i)
return 7
default:
b[0] = byte(i >> 56)
b[1] = byte(i >> 48)
b[2] = byte(i >> 40)
b[3] = byte(i >> 32)
b[4] = byte(i >> 24)
b[5] = byte(i >> 16)
b[6] = byte(i >> 8)
b[7] = byte(i)
return 8
}
}
```

```
// intsize computes the minimum number of bytes required to store i.
func intsize(i uint64) (size int) {
for size = 1; ; size++ {
if i >>= 8; i == 0 {
return size
}
}
}

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

package rlp

import (
"fmt"
"io"
)

type MyCoolType struct {
Name string
a, b uint
}

// EncodeRLP writes x as RLP list [a, b] that omits the Name field.
func (x *MyCoolType) EncodeRLP(w io.Writer) (err error) {
// Note: the receiver can be a nil pointer. This allows you to
// control the encoding of nil, but it also means that you have to
// check for a nil receiver.
if x == nil {
err = Encode(w, []uint{0, 0})
} else {
err = Encode(w, []uint{x.a, x.b})
}
return err
}

func ExampleEncoder() {
var t *MyCoolType // t is nil pointer to MyCoolType
bytes, _ := EncodeToBytes(t)
fmt.Printf("%v  %X\n", t, bytes)
t = &MyCoolType{Name: "foobar", a: 5, b: 6}
bytes, _ = EncodeToBytes(t)
fmt.Printf("%v  %X\n", t, bytes)

// Output:
// <nil>  C28080
// &{foobar 5 6}  C20506
}

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

package rlp

import (
"bytes"
"errors"
"fmt"
"io"
"io/ioutil"
"math/big"
"sync"
"testing"
)

type testEncoder struct {
err error
}

func (e *testEncoder) EncodeRLP(w io.Writer) error {
if e == nil {
w.Write([]byte{0, 0, 0, 0})
} else if e.err != nil {
return e.err
} else {
w.Write([]byte{0, 1, 0, 1, 0, 1, 0, 1, 0, 1})
}
return nil
}

type byteEncoder byte
```

```
func (e byteEncoder) EncodeRLP(w io.Writer) error {
w.Write(EmptyList)
return nil
}

type encodableReader struct {
A, B uint
}

func (e *encodableReader) Read(b []byte) (int, error) {
panic("called")
}

type namedByteType byte

var (
_ = Encoder(&testEncoder{})
_ = Encoder(byteEncoder(0))

reader io.Reader = &encodableReader{1, 2}
)

type encTest struct {
val           interface{}
output, error string
}

var encTests = []encTest{
// booleans
{val: true, output: "01"},
{val: false, output: "80"},

// integers
{val: uint32(0), output: "80"},
{val: uint32(127), output: "7F"},
{val: uint32(128), output: "8180"},
{val: uint32(256), output: "820100"},
{val: uint32(1024), output: "820400"},
{val: uint32(0xFFFFFF), output: "83FFFFFF"},
{val: uint32(0xFFFFFFFF), output: "84FFFFFFFF"},
{val: uint64(0xFFFFFFFF), output: "84FFFFFFFF"},
{val: uint64(0xFFFFFFFFFF), output: "85FFFFFFFFFF"},
{val: uint64(0xFFFFFFFFFFFF), output: "86FFFFFFFFFFFF"},
{val: uint64(0xFFFFFFFFFFFFFF), output: "87FFFFFFFFFFFFFF"},
{val: uint64(0xFFFFFFFFFFFFFFFF), output: "88FFFFFFFFFFFFFFFF"},

// big integers (should match uint for small values)
{val: big.NewInt(0), output: "80"},
{val: big.NewInt(1), output: "01"},
{val: big.NewInt(127), output: "7F"},
{val: big.NewInt(128), output: "8180"},
{val: big.NewInt(256), output: "820100"},
{val: big.NewInt(1024), output: "820400"},
{val: big.NewInt(0xFFFFFF), output: "83FFFFFF"},
{val: big.NewInt(0xFFFFFFFF), output: "84FFFFFFFF"},
{val: big.NewInt(0xFFFFFFFFFF), output: "85FFFFFFFFFF"},
{val: big.NewInt(0xFFFFFFFFFFFF), output: "86FFFFFFFFFFFF"},
{val: big.NewInt(0xFFFFFFFFFFFFFF), output: "87FFFFFFFFFFFFFF"},
{
val:    big.NewInt(0).SetBytes(unhex("102030405060708090A0B0C0D0E0F2")),
output: "8F102030405060708090A0B0C0D0E0F2",
},
{
val:
big.NewInt(0).SetBytes(unhex("0100020003000400050006000700080009000A000B000C000D00
0E01")),
output: "9C0100020003000400050006000700080009000A000B000C000D000E01",
},
{
val:
big.NewInt(0).SetBytes(unhex("0100000000000000000000000000000000000000000000000000
00000000000000")),
output: "A1010000000000000000000000000000000000000000000000000000000000000000",
},

// non-pointer big.Int
{val: *big.NewInt(0), output: "80"},
{val: *big.NewInt(0xFFFFFF), output: "83FFFFFF"},

// negative ints are not supported
{val: big.NewInt(-1), error: "rlp: cannot encode negative *big.Int"},

// byte slices, strings
```

```
{val: []byte{}, output: "80"},
{val: []byte{0x7E}, output: "7E"},
{val: []byte{0x7F}, output: "7F"},
{val: []byte{0x80}, output: "8180"},
{val: []byte{1, 2, 3}, output: "83010203"},

{val: []namedByteType{1, 2, 3}, output: "83010203"},
{val: [...]namedByteType{1, 2, 3}, output: "83010203"},

{val: "", output: "80"},
{val: "\x7E", output: "7E"},
{val: "\x7F", output: "7F"},
{val: "\x80", output: "8180"},
{val: "dog", output: "83646F67"},
{
val:    "Lorem ipsum dolor sit amet, consectetur adipisicing eli",
output:
"B74C6F72656D20697073756D20646F6C6F722073697420616D65742C20636F6E7365637465747
47572206164697069736963696E6720656C69",
},
{
val:    "Lorem ipsum dolor sit amet, consectetur adipisicing elit",
output:
"B8384C6F72656D20697073756D20646F6C6F722073697420616D65742C20636F6E736563746
5747572206164697069736963696E6720656C6974",
},
{
val:    "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur mauris magna, suscipit
sed vehicula non, iaculis faucibus tortor. Proin suscipit ultricies malesuada. Duis tortor elit, dictum
quis tristique eu, ultrices at risus. Morbi a est imperdiet mi ullamcorper aliquet suscipit nec lorem.
Aenean quis leo mollis, vulputate elit varius, consequat enim. Nulla ultrices turpis justo, et posuere
urna consectetur nec. Proin non convallis metus. Donec tempor ipsum in mauris congue
sollicitudin. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae;
Suspendisse convallis sem vel massa faucibus, eget lacinia lacus tempor. Nulla quis ultricies
purus. Proin auctor rhoncus nibh condimentum mollis. Aliquam consequat enim at metus luctus, a
eleifend purus egestas. Curabitur at nibh metus. Nam bibendum, neque at auctor tristique, lorem
libero aliquet arcu, non interdum tellus lectus sit amet eros. Cras rhoncus, metus ac ornare cursus,
dolor justo ultrices metus, at ullamcorper volutpat",
output:
"B904004C6F72656D20697073756D20646F6C6F722073697420616D65742C20636F6E7365637
4657475722061646970697363696E6720656C69742E20437572616269747572206D6175726973
206D61676E612C20737573636970697420736564207665686963756C61206E6F6E2C20696163
756C697320666175636962757320746F72746F722E2050726F696E20737573636970697420756
C74726963696573206D616C6573756164612E204475697320746F72746F7220656C69742C206
4696374756D2071756973207472697374697175652065752C20756C7472696365732061742072
697375732E204D6F72626920612065737420696D70657264696574206D6920756C6C616D636F
7270657220616C6971756574207375736369706974206E6563206C6F72656D2E2041656E6561
6E2071756973206C656F206D6F6C6C69732C2076756C70757461746520656C6974207661726
975732C20636F6E73657175617420656E696D2E204E756C6C6120756C747269636573207475
72706973206A7573746F2C20657420706F73756572652075726E6120636F6E736563746574757
2206E65632E2050726F696E206E6F6E20636F6E76616C6C6973206D657475732E20446F6E65
632074656D706F7220697073756D20696E206D617572697320636F6E67756520736F6C6C6963
69747564696E2E20566573746962756C756D20616E746520697073756D207072696D69732069
6E206661756369627573206F726369206C756374757320657420756C74726963657320706F737
565726520637562696C69612043757261653B2053757370656E646973736520636F6E76616C6
C69732073656D2076656C206D617373612066617563696275732C2065676574206C6163696E6
961206C616375732074656D706F722E204E756C6C61207175697320756C74726963696573207
0757275732E2050726F696E20617563746F722072686F6E637573206E69626820636F6E64696
D656E74756D206D6F6C6C69732E20416C697175616D20636F6E73657175617420656E696D20
6174206D65747573206C75637475732C206120656C656966656E642070757275732065676573
7461732E20437572616269747572206174206E696268206D657475732E204E616D2062696265
6E64756D2C206E6571756520617420617563746F72207472697374697175652C206C6F72656D
206C696265726F20616C697175657420617263752C206E6F6E20696E74657264756D2074656C
6C7573206C65637475732073697420616D65742065726F732E20437261732072686F6E637573
2C206D65747573206163206F726E617265206375727375732C20646F6C6F72206A7573746F20
756C747269636573206D657475732C20617420756C6C616D636F7270657220766F6C75747061
74",
},

// slices
{val: []uint{}, output: "C0"},
{val: []uint{1, 2, 3}, output: "C3010203"},
{
// [ [], [[]], [ [], [[]] ] ]
val:    []interface{}{[]interface{}{}, [][]interface{}{{}}, []interface{}{[]interface{}{}, [][]interface{}{{}}}},
output: "C7C0C1C0C3C0C1C0",
},
{
val:    []string{"aaa", "bbb", "ccc", "ddd", "eee", "fff", "ggg", "hhh", "iii", "jjj", "kkk", "lll", "mmm", "nnn",
"ooo"},
output:
"F83C8361616183626262836363638364646483656565836666668367676783686868836969698
36A6A6A836B6B6B836C6C6C836D6D6D836E6E6E836F6F6F",
},
{
val:    []interface{}{uint(1), uint(0xFFFFFF), []interface{}{[]uint{4, 5, 5}}, "abc"},
output: "CE0183FFFFFFC4C304050583616263",
},
{
val: [][]string{
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
{"asdf", "qwer", "zxcv"},
},
output:
"F90200CF84617364668471776572847A786376CF84617364668471776572847A786376CF8461
7364668471776572847A786376CF84617364668471776572847A786376CF84617364668471776
572847A786376CF84617364668471776572847A786376CF84617364668471776572847A78637
6CF84617364668471776572847A786376CF84617364668471776572847A786376CF846173646
68471776572847A786376CF84617364668471776572847A786376CF8461736466847177657284
7A786376CF84617364668471776572847A786376CF84617364668471776572847A786376CF84
617364668471776572847A786376CF84617364668471776572847A786376CF846173646684717
76572847A786376CF84617364668471776572847A786376CF84617364668471776572847A786
376CF84617364668471776572847A786376CF84617364668471776572847A786376CF8461736
4668471776572847A786376CF84617364668471776572847A786376CF84617364668471776572
847A786376CF84617364668471776572847A786376CF84617364668471776572847A786376CF
84617364668471776572847A786376CF84617364668471776572847A786376CF8461736466847
1776572847A786376CF84617364668471776572847A786376CF84617364668471776572847A7
86376CF84617364668471776572847A786376",
},

// RawValue
{val: RawValue(unhex("01")), output: "01"},
{val: RawValue(unhex("82FFFF")), output: "82FFFF"},
{val: []RawValue{unhex("01"), unhex("02")}, output: "C20102"},

// structs
{val: simplestruct{}, output: "C28080"},
{val: simplestruct{A: 3, B: "foo"}, output: "C50383666F6F"},
{val: &recstruct{5, nil}, output: "C205C0"},
{val: &recstruct{5, &recstruct{4, &recstruct{3, nil}}}, output: "C605C404C203C0"},
{val: &tailRaw{A: 1, Tail: []RawValue{unhex("02"), unhex("03")}}, output: "C3010203"},
{val: &tailRaw{A: 1, Tail: []RawValue{unhex("02")}}, output: "C20102"},
{val: &tailRaw{A: 1, Tail: []RawValue{}}, output: "C101"},
{val: &tailRaw{A: 1, Tail: nil}, output: "C101"},
{val: &hasIgnoredField{A: 1, B: 2, C: 3}, output: "C20103"},

// nil
{val: (*uint)(nil), output: "80"},
{val: (*string)(nil), output: "80"},
{val: (*[]byte)(nil), output: "80"},
{val: (*[10]byte)(nil), output: "80"},
{val: (*big.Int)(nil), output: "80"},
{val: (*[]string)(nil), output: "C0"},
{val: (*[10]string)(nil), output: "C0"},
{val: (*[]interface{})(nil), output: "C0"},
{val: (*[]struct{ uint })(nil), output: "C0"},
{val: (*interface{})(nil), output: "C0"},
// interfaces
{val: []io.Reader{reader}, output: "C3C20102"}, // the contained value is a struct

// Encoder
{val: (*testEncoder)(nil), output: "00000000"},
{val: &testEncoder{}, output: "00010001000100010001"},
{val: &testEncoder{errors.New("test error")}, error: "test error"},
// verify that pointer method testEncoder.EncodeRLP is called for
// addressable non-pointer values.
{val: &struct{ TE testEncoder }{testEncoder{}}, output: "CA00010001000100010001"},
{val: &struct{ TE testEncoder }{testEncoder{errors.New("test error")}}, error: "test error"},
// verify the error for non-addressable non-pointer Encoder
{val: testEncoder{}, error: "rlp: game over: unadressable value of type rlp.testEncoder, EncodeRLP
is pointer method"},
// verify the special case for []byte
{val: []byteEncoder{0, 1, 2, 3, 4}, output: "C5C0C0C0C0C0"},
}

func runEncTests(t *testing.T, f func(val interface{}) ([]byte, error)) {
for i, test := range encTests {
output, err := f(test.val)
if err != nil && test.error == "" {
t.Errorf("test %d: unexpected error: %v\nvalue %#v\ntype %T",
i, err, test.val, test.val)
continue
}
if test.error != "" && fmt.Sprint(err) != test.error {
t.Errorf("test %d: error mismatch\ngot   %v\nwant  %v\nvalue %#v\ntype  %T",
i, err, test.error, test.val, test.val)
continue
}
if err == nil && !bytes.Equal(output, unhex(test.output)) {
t.Errorf("test %d: output mismatch:\ngot   %X\nwant  %s\nvalue %#v\ntype  %T",
i, output, test.output, test.val, test.val)
}
}
}

func TestEncode(t *testing.T) {
runEncTests(t, func(val interface{}) ([]byte, error) {
b := new(bytes.Buffer)
err := Encode(b, val)
```

```
return b.Bytes(), err
})
}

func TestEncodeToBytes(t *testing.T) {
runEncTests(t, EncodeToBytes)
}

func TestEncodeToReader(t *testing.T) {
runEncTests(t, func(val interface{}) ([]byte, error) {
_, r, err := EncodeToReader(val)
if err != nil {
return nil, err
}
return ioutil.ReadAll(r)
})
}

func TestEncodeToReaderPiecewise(t *testing.T) {
runEncTests(t, func(val interface{}) ([]byte, error) {
size, r, err := EncodeToReader(val)
if err != nil {
return nil, err
}

// read output piecewise
output := make([]byte, size)
for start, end := 0, 0; start < size; start = end {
if remaining := size - start; remaining < 3 {
end += remaining
} else {
end = start + 3
}
n, err := r.Read(output[start:end])
end = start + n
if err == io.EOF {
break
} else if err != nil {
return nil, err
}
}
return output, nil
```

```
})
}

// This is a regression test verifying that encReader
// returns its encbuf to the pool only once.
func TestEncodeToReaderReturnToPool(t *testing.T) {
buf := make([]byte, 50)
wg := new(sync.WaitGroup)
for i := 0; i < 5; i++ {
wg.Add(1)
go func() {
for i := 0; i < 1000; i++ {
_, r, _ := EncodeToReader("foo")
ioutil.ReadAll(r)
r.Read(buf)
r.Read(buf)
r.Read(buf)
r.Read(buf)
}
wg.Done()
}()
}
wg.Wait()
}

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

package rlp

import (
"io"
"reflect"
)

// RawValue represents an encoded RLP value and can be used to delay
// RLP decoding or to precompute an encoding. Note that the decoder does
// not verify whether the content of RawValues is valid RLP.
type RawValue []byte

var rawValueType = reflect.TypeOf(RawValue{})
```

```
// ListSize returns the encoded size of an RLP list with the given
// content size.
func ListSize(contentSize uint64) uint64 {
return uint64(headsize(contentSize)) + contentSize
}

// Split returns the content of first RLP value and any
// bytes after the value as subslices of b.
func Split(b []byte) (k Kind, content, rest []byte, err error) {
k, ts, cs, err := readKind(b)
if err != nil {
return 0, nil, b, err
}
return k, b[ts : ts+cs], b[ts+cs:], nil
}

// SplitString splits b into the content of an RLP string
// and any remaining bytes after the string.
func SplitString(b []byte) (content, rest []byte, err error) {
k, content, rest, err := Split(b)
if err != nil {
return nil, b, err
}
if k == List {
return nil, b, ErrExpectedString
}
return content, rest, nil
}

// SplitList splits b into the content of a list and any remaining
// bytes after the list.
func SplitList(b []byte) (content, rest []byte, err error) {
k, content, rest, err := Split(b)
if err != nil {
return nil, b, err
}
if k != List {
return nil, b, ErrExpectedList
}
return content, rest, nil
}
```

```
// CountValues counts the number of encoded values in b.
func CountValues(b []byte) (int, error) {
i := 0
for ; len(b) > 0; i++ {
_, tagsize, size, err := readKind(b)
if err != nil {
return 0, err
}
b = b[tagsize+size:]
}
return i, nil
}

func readKind(buf []byte) (k Kind, tagsize, contentsize uint64, err error) {
if len(buf) == 0 {
return 0, 0, 0, io.ErrUnexpectedEOF
}
b := buf[0]
switch {
case b < 0x80:
k = Byte
tagsize = 0
contentsize = 1
case b < 0xB8:
k = String
tagsize = 1
contentsize = uint64(b - 0x80)
// Reject strings that should've been single bytes.
if contentsize == 1 && buf[1] < 128 {
return 0, 0, 0, ErrCanonSize
}
case b < 0xC0:
k = String
tagsize = uint64(b-0xB7) + 1
contentsize, err = readSize(buf[1:], b-0xB7)
case b < 0xF8:
k = List
tagsize = 1
contentsize = uint64(b - 0xC0)
default:
k = List
tagsize = uint64(b-0xF7) + 1
```

```
contentsize, err = readSize(buf[1:], b-0xF7)
}
if err != nil {
return 0, 0, 0, err
}
// Reject values larger than the input slice.
if contentsize > uint64(len(buf))-tagsize {
return 0, 0, 0, ErrValueTooLarge
}
return k, tagsize, contentsize, err
}

func readSize(b []byte, slen byte) (uint64, error) {
if int(slen) > len(b) {
return 0, io.ErrUnexpectedEOF
}
var s uint64
switch slen {
case 1:
s = uint64(b[0])
case 2:
s = uint64(b[0])<<8 | uint64(b[1])
case 3:
s = uint64(b[0])<<16 | uint64(b[1])<<8 | uint64(b[2])
case 4:
s = uint64(b[0])<<24 | uint64(b[1])<<16 | uint64(b[2])<<8 | uint64(b[3])
case 5:
s = uint64(b[0])<<32 | uint64(b[1])<<24 | uint64(b[2])<<16 | uint64(b[3])<<8 | uint64(b[4])
case 6:
s = uint64(b[0])<<40 | uint64(b[1])<<32 | uint64(b[2])<<24 | uint64(b[3])<<16 | uint64(b[4])<<8 |
uint64(b[5])
case 7:
s = uint64(b[0])<<48 | uint64(b[1])<<40 | uint64(b[2])<<32 | uint64(b[3])<<24 | uint64(b[4])<<16 |
uint64(b[5])<<8 | uint64(b[6])
case 8:
s = uint64(b[0])<<56 | uint64(b[1])<<48 | uint64(b[2])<<40 | uint64(b[3])<<32 | uint64(b[4])<<24 |
uint64(b[5])<<16 | uint64(b[6])<<8 | uint64(b[7])
}
// Reject sizes < 56 (shouldn't have separate size) and sizes with
// leading zero bytes.
if s < 56 || b[0] == 0 {
return 0, ErrCanonSize
```

```
}
return s, nil
}

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

package rlp

import (
"bytes"
"io"
"reflect"
"testing"
)

func TestCountValues(t *testing.T) {
tests := []struct {
input string // note: spaces in input are stripped by unhex
count int
err   error
}{
// simple cases
{"", 0, nil},
{"00", 1, nil},
{"80", 1, nil},
{"C0", 1, nil},
{"01 02 03", 3, nil},
{"01 C406070809 02", 3, nil},
{"820101 820202 8403030303 04", 4, nil},

// size errors
{"8142", 0, ErrCanonSize},
{"01 01 8142", 0, ErrCanonSize},
{"02 84020202", 0, ErrValueTooLarge},

{
input:
"A12000BF49F440A1CD0527E4D06E2765654C0F56452257516D793A9B8D604DCFDF2AB853
F851808D10000000000000000000000000A056E81F171BCC55A6FF8345E692C0F86E5B48E01
B996CADC001622FB5E363B421A0C5D2460186F7233C927E7DB2DCC703C0E500B653CA822
73B7BFAD8045D85A470",
count: 2,
},
}
for i, test := range tests {
count, err := CountValues(unhex(test.input))
if count != test.count {
t.Errorf("test %d: count mismatch, got %d want %d\ninput: %s", i, count, test.count, test.input)
}
if !reflect.DeepEqual(err, test.err) {
t.Errorf("test %d: err mismatch, got %q want %q\ninput: %s", i, err, test.err, test.input)
}
}
}

func TestSplitTypes(t *testing.T) {
if _, _, err := SplitString(unhex("C100")); err != ErrExpectedString {
t.Errorf("SplitString returned %q, want %q", err, ErrExpectedString)
}
if _, _, err := SplitList(unhex("01")); err != ErrExpectedList {
t.Errorf("SplitString returned %q, want %q", err, ErrExpectedList)
}
if _, _, err := SplitList(unhex("81FF")); err != ErrExpectedList {
t.Errorf("SplitString returned %q, want %q", err, ErrExpectedList)
}
}

func TestSplit(t *testing.T) {
tests := []struct {
input     string
kind      Kind
val, rest string
err       error
}{
{input: "01FFFF", kind: Byte, val: "01", rest: "FFFF"},
{input: "80FFFF", kind: String, val: "", rest: "FFFF"},
{input: "C3010203", kind: List, val: "010203"},

// errors
{input: "", err: io.ErrUnexpectedEOF},

{input: "8141", err: ErrCanonSize, rest: "8141"},
{input: "B800", err: ErrCanonSize, rest: "B800"},
{input: "B802FFFF", err: ErrCanonSize, rest: "B802FFFF"},
{input: "B90000", err: ErrCanonSize, rest: "B90000"},
{input: "B90055", err: ErrCanonSize, rest: "B90055"},
{input: "BA0002FFFF", err: ErrCanonSize, rest: "BA0002FFFF"},
{input: "F800", err: ErrCanonSize, rest: "F800"},
{input: "F90000", err: ErrCanonSize, rest: "F90000"},
{input: "F90055", err: ErrCanonSize, rest: "F90055"},
{input: "FA0002FFFF", err: ErrCanonSize, rest: "FA0002FFFF"},

{input: "8501010101", err: ErrValueTooLarge, rest: "8501010101"},
{input: "C60607080902", err: ErrValueTooLarge, rest: "C60607080902"},

// size check overflow
{input: "BFFFFFFFFFFFFFFFFF", err: ErrValueTooLarge, rest: "BFFFFFFFFFFFFFFFFF"},
{input: "FFFFFFFFFFFFFFFFFF", err: ErrValueTooLarge, rest: "FFFFFFFFFFFFFFFFFF"},

{
input:
"B838FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF",
err:   ErrValueTooLarge,
rest:
"B838FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF",
},
{
input:
"F838FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF",
err:   ErrValueTooLarge,
rest:
"F838FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF",
},

// a few bigger values, just for kicks
{
input:
"F839FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF",
kind:  List,
val:
```

```
"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF",
rest: "",
},
{
input:
"F90211A060EF29F20CC1007AE6E9530AEE16F4B31F8F1769A2D1264EC995C6D1241868D6
A07C62AB8AC9838F5F5877B20BB37B387BC2106E97A3D52172CBEDB5EE17C36008A00EAB
6B7324AADC0F6047C6AFC8229F09F7CF451B51D67C8DFB08D49BA8C3C626A04453343B2F
3A6E42FCF87948F88AF7C8FC16D0C2735CBA7F026836239AB2C15FA024635C7291C882CE
4C0763760C1A362DFC3FFCD802A55722236DE058D74202ACA0A220C808DE10F55E40AB25
255201CFF009EA181D3906638E944EE2BF34049984A08D325AB26796F1CCB470F69C0F8425
01DC35D368A0C2575B2D243CFD1E8AB0FDA0B5298FF60DA5069463D610513C9F04F24051
348391A143AFFAB7197DFACDEA72A02D2A7058A4463F8FB69378369E11EF33AE3252E2DB
86CB545B36D3C26DDECE5AA0888F97BCA8E0BD83DC5B3B91CFF5FAF2F66F9501010682D
67EF4A3B4E66115FBA0E8175A60C93BE9ED02921958F0EA55DA0FB5E4802AF5846147BAD
92BC2D8AF26A08B3376FF433F3A4250FA64B7F804004CAC5807877D91C4427BD1CD05CF9
12ED8A09B32EF0F03BD13C37FF950C0CCCEFCCDD6669F2E7F2AA5CB859928E84E29763E
A09BBA5E46610C8C8B1F8E921E5691BF8C7E40D75825D5EA3217AA9C3A8A355F39A0EEB9
5BC78251CCCEC54A97F19755C4A59A293544EEE6119AFA50531211E53C4FA00B6E86FE15
0BF4A9E0FEEE9C90F5465E617A861BB5E357F942881EE762212E2580",
kind: List,
val:
"A060EF29F20CC1007AE6E9530AEE16F4B31F8F1769A2D1264EC995C6D1241868D6A07C62
AB8AC9838F5F5877B20BB37B387BC2106E97A3D52172CBEDB5EE17C36008A00EAB6B7324
AADC0F6047C6AFC8229F09F7CF451B51D67C8DFB08D49BA8C3C626A04453343B2F3A6E42
FCF87948F88AF7C8FC16D0C2735CBA7F026836239AB2C15FA024635C7291C882CE4C07637
60C1A362DFC3FFCD802A55722236DE058D74202ACA0A220C808DE10F55E40AB25255201C
FF009EA181D3906638E944EE2BF34049984A08D325AB26796F1CCB470F69C0F842501DC35
D368A0C2575B2D243CFD1E8AB0FDA0B5298FF60DA5069463D610513C9F04F24051348391A
143AFFAB7197DFACDEA72A02D2A7058A4463F8FB69378369E11EF33AE3252E2DB86CB545
B36D3C26DDECE5AA0888F97BCA8E0BD83DC5B3B91CFF5FAF2F66F9501010682D67EF4A3
B4E66115FBA0E8175A60C93BE9ED02921958F0EA55DA0FB5E4802AF5846147BAD92BC2D8
AF26A08B3376FF433F3A4250FA64B7F804004CAC5807877D91C4427BD1CD05CF912ED8A09
B32EF0F03BD13C37FF950C0CCCEFCCDD6669F2E7F2AA5CB859928E84E29763EA09BBA5E
46610C8C8B1F8E921E5691BF8C7E40D75825D5EA3217AA9C3A8A355F39A0EEB95BC78251
CCCEC54A97F19755C4A59A293544EEE6119AFA50531211E53C4FA00B6E86FE150BF4A9E0
FEEE9C90F5465E617A861BB5E357F942881EE762212E2580",
rest: "",
},
{
input:
```

```
"F877A12000BF49F440A1CD0527E4D06E2765654C0F56452257516D793A9B8D604DCFDF2A
B853F851808D10000000000000000000000000A056E81F171BCC55A6FF8345E692C0F86E5B4
8E01B996CADC001622FB5E363B421A0C5D2460186F7233C927E7DB2DCC703C0E500B653C
A82273B7BFAD8045D85A470",
kind:  List,
val:
"A12000BF49F440A1CD0527E4D06E2765654C0F56452257516D793A9B8D604DCFDF2AB853
F851808D10000000000000000000000000A056E81F171BCC55A6FF8345E692C0F86E5B48E01
B996CADC001622FB5E363B421A0C5D2460186F7233C927E7DB2DCC703C0E500B653CA822
73B7BFAD8045D85A470",
rest:  "",
},
}

for i, test := range tests {
kind, val, rest, err := Split(unhex(test.input))
if kind != test.kind {
t.Errorf("test %d: kind mismatch: got %v, want %v", i, kind, test.kind)
}
if !bytes.Equal(val, unhex(test.val)) {
t.Errorf("test %d: val mismatch: got %x, want %s", i, val, test.val)
}
if !bytes.Equal(rest, unhex(test.rest)) {
t.Errorf("test %d: rest mismatch: got %x, want %s", i, rest, test.rest)
}
if err != test.err {
t.Errorf("test %d: error mismatch: got %q, want %q", i, err, test.err)
}
}
}

func TestReadSize(t *testing.T) {
tests := []struct {
input string
slen  byte
size  uint64
err   error
}{
{input: "", slen: 1, err: io.ErrUnexpectedEOF},
{input: "FF", slen: 2, err: io.ErrUnexpectedEOF},
{input: "00", slen: 1, err: ErrCanonSize},
{input: "36", slen: 1, err: ErrCanonSize},
{input: "37", slen: 1, err: ErrCanonSize},
{input: "38", slen: 1, size: 0x38},
{input: "FF", slen: 1, size: 0xFF},
{input: "FFFF", slen: 2, size: 0xFFFF},
{input: "FFFFFF", slen: 3, size: 0xFFFFFF},
{input: "FFFFFFFF", slen: 4, size: 0xFFFFFFFF},
{input: "FFFFFFFFFF", slen: 5, size: 0xFFFFFFFFFF},
{input: "FFFFFFFFFFFF", slen: 6, size: 0xFFFFFFFFFFFF},
{input: "FFFFFFFFFFFFFF", slen: 7, size: 0xFFFFFFFFFFFFFF},
{input: "FFFFFFFFFFFFFFFF", slen: 8, size: 0xFFFFFFFFFFFFFFFF},
{input: "0102", slen: 2, size: 0x0102},
{input: "010203", slen: 3, size: 0x010203},
{input: "01020304", slen: 4, size: 0x01020304},
{input: "0102030405", slen: 5, size: 0x0102030405},
{input: "010203040506", slen: 6, size: 0x010203040506},
{input: "01020304050607", slen: 7, size: 0x01020304050607},
{input: "0102030405060708", slen: 8, size: 0x0102030405060708},
}

for _, test := range tests {
size, err := readSize(unhex(test.input), test.slen)
if err != test.err {
t.Errorf("readSize(%s, %d): error mismatch: got %q, want %q", test.input, test.slen, err, test.err)
continue
}
if size != test.size {
t.Errorf("readSize(%s, %d): size mismatch: got %#x, want %#x", test.input, test.slen, size,
test.size)
}
}
}

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

package rlp

import (
"fmt"
"reflect"
"strings"
"sync"
```

```
)

var (
typeCacheMutex sync.RWMutex
typeCache      = make(map[typekey]*typeinfo)
)

type typeinfo struct {
decoder
writer
}

// represents struct tags
type tags struct {
// rlp:"nil" controls whether empty input results in a nil pointer.
nilOK bool
// rlp:"tail" controls whether this field swallows additional list
// elements. It can only be set for the last field, which must be
// of slice type.
tail bool
// rlp:"-" ignores fields.
ignored bool
}

type typekey struct {
reflect.Type
// the key must include the struct tags because they
// might generate a different decoder.
tags
}

type decoder func(*Stream, reflect.Value) error

type writer func(reflect.Value, *encbuf) error

func cachedTypeInfo(typ reflect.Type, tags tags) (*typeinfo, error) {
typeCacheMutex.RLock()
info := typeCache[typekey{typ, tags}]
typeCacheMutex.RUnlock()
if info != nil {
return info, nil
}
```

```
// not in the cache, need to generate info for this type.
typeCacheMutex.Lock()
defer typeCacheMutex.Unlock()
return cachedTypeInfo1(typ, tags)
}

func cachedTypeInfo1(typ reflect.Type, tags tags) (*typeinfo, error) {
key := typekey{typ, tags}
info := typeCache[key]
if info != nil {
// another goroutine got the write lock first
return info, nil
}
// put a dummmy value into the cache before generating.
// if the generator tries to lookup itself, it will get
// the dummy value and won't call itself recursively.
typeCache[key] = new(typeinfo)
info, err := genTypeInfo(typ, tags)
if err != nil {
// remove the dummy value if the generator fails
delete(typeCache, key)
return nil, err
}
*typeCache[key] = *info
return typeCache[key], err
}

type field struct {
index int
info  *typeinfo
}

func structFields(typ reflect.Type) (fields []field, err error) {
for i := 0; i < typ.NumField(); i++ {
if f := typ.Field(i); f.PkgPath == "" { // exported
tags, err := parseStructTag(typ, i)
if err != nil {
return nil, err
}
if tags.ignored {
continue
}
info, err := cachedTypeInfo1(f.Type, tags)
if err != nil {
return nil, err
}
fields = append(fields, field{i, info})
}
}
return fields, nil
}

func parseStructTag(typ reflect.Type, fi int) (tags, error) {
f := typ.Field(fi)
var ts tags
for _, t := range strings.Split(f.Tag.Get("rlp"), ",") {
switch t = strings.TrimSpace(t); t {
case "":
case "-":
ts.ignored = true
case "nil":
ts.nilOK = true
case "tail":
ts.tail = true
if fi != typ.NumField()-1 {
return ts, fmt.Errorf(`rlp: invalid struct tag "tail" for %v.%s (must be on last field)`, typ, f.Name)
}
if f.Type.Kind() != reflect.Slice {
return ts, fmt.Errorf(`rlp: invalid struct tag "tail" for %v.%s (field type is not slice)`, typ, f.Name)
}
default:
return ts, fmt.Errorf("rlp: unknown struct tag %q on %v.%s", t, typ, f.Name)
}
}
return ts, nil
}

func genTypeInfo(typ reflect.Type, tags tags) (info *typeinfo, err error) {
info = new(typeinfo)
if info.decoder, err = makeDecoder(typ, tags); err != nil {
return nil, err
}
if info.writer, err = makeWriter(typ, tags); err != nil {
return nil, err
```

```
}
return info, nil
}

func isUint(k reflect.Kind) bool {
return k >= reflect.Uint && k <= reflect.Uintptr
}

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

package rpc

import (
"bytes"
"container/list"
"context"
"encoding/json"
"errors"
"fmt"
"net"
"net/url"
"reflect"
"strconv"
"strings"
"sync"
"sync/atomic"
"time"

"github.com/ethereum/go-ethereum/log"
)

var (
ErrClientQuit                = errors.New("client is closed")
ErrNoResult                  = errors.New("no result in JSON-RPC response")
ErrSubscriptionQueueOverflow = errors.New("subscription queue overflow")
)

const (
// Timeouts
tcpKeepAliveInterval = 30 * time.Second
defaultDialTimeout   = 10 * time.Second // used when dialing if the context has no deadline
```

```
defaultWriteTimeout  = 10 * time.Second // used for calls if the context has no deadline
subscribeTimeout     = 5 * time.Second  // overall timeout eth_subscribe, rpc_modules calls
)

const (
// Subscriptions are removed when the subscriber cannot keep up.
//
// This can be worked around by supplying a channel with sufficiently sized buffer,
// but this can be inconvenient and hard to explain in the docs. Another issue with
// buffered channels is that the buffer is static even though it might not be needed
// most of the time.
//
// The approach taken here is to maintain a per-subscription linked list buffer
// shrinks on demand. If the buffer reaches the size below, the subscription is
// dropped.
maxClientSubscriptionBuffer = 8000
)

// BatchElem is an element in a batch request.
type BatchElem struct {
Method string
Args   []interface{}
// The result is unmarshaled into this field. Result must be set to a
// non-nil pointer value of the desired type, otherwise the response will be
// discarded.
Result interface{}
// Error is set if the server returns an error for this request, or if
// unmarshaling into Result fails. It is not set for I/O errors.
Error error
}

// A value of this type can a JSON-RPC request, notification, successful response or
// error response. Which one it is depends on the fields.
type jsonrpcMessage struct {
Version string          `json:"jsonrpc"`
ID      json.RawMessage `json:"id,omitempty"`
Method  string          `json:"method,omitempty"`
Params  json.RawMessage `json:"params,omitempty"`
Error   *jsonError      `json:"error,omitempty"`
Result  json.RawMessage `json:"result,omitempty"`
}
```

```go
func (msg *jsonrpcMessage) isNotification() bool {
return msg.ID == nil && msg.Method != ""
}

func (msg *jsonrpcMessage) isResponse() bool {
return msg.hasValidID() && msg.Method == "" && len(msg.Params) == 0
}

func (msg *jsonrpcMessage) hasValidID() bool {
return len(msg.ID) > 0 && msg.ID[0] != '{' && msg.ID[0] != '['
}

func (msg *jsonrpcMessage) String() string {
b, _ := json.Marshal(msg)
return string(b)
}

// Client represents a connection to an RPC server.
type Client struct {
idCounter   uint32
connectFunc func(ctx context.Context) (net.Conn, error)
isHTTP      bool

// writeConn is only safe to access outside dispatch, with the
// write lock held. The write lock is taken by sending on
// requestOp and released by sending on sendDone.
writeConn net.Conn

// for dispatch
close       chan struct{}
didQuit     chan struct{}                // closed when client quits
reconnected chan net.Conn                // where write/reconnect sends the new connection
readErr     chan error                   // errors from read
readResp    chan []*jsonrpcMessage       // valid messages from read
requestOp   chan *requestOp              // for registering response IDs
sendDone    chan error                   // signals write completion, releases write lock
respWait    map[string]*requestOp        // active requests
subs        map[string]*ClientSubscription // active subscriptions
}

type requestOp struct {
ids  []json.RawMessage
```

```
err  error
resp chan *jsonrpcMessage // receives up to len(ids) responses
sub  *ClientSubscription  // only set for EthSubscribe requests
}

func (op *requestOp) wait(ctx context.Context) (*jsonrpcMessage, error) {
select {
case <-ctx.Done():
return nil, ctx.Err()
case resp := <-op.resp:
return resp, op.err
}
}

// Dial creates a new client for the given URL.
//
// The currently supported URL schemes are "http", "https", "ws" and "wss". If rawurl is a
// file name with no URL scheme, a local socket connection is established using UNIX
// domain sockets on supported platforms and named pipes on Windows. If you want to
// configure transport options, use DialHTTP, DialWebsocket or DialIPC instead.
//
// For websocket connections, the origin is set to the local host name.
//
// The client reconnects automatically if the connection is lost.
func Dial(rawurl string) (*Client, error) {
return DialContext(context.Background(), rawurl)
}

// DialContext creates a new RPC client, just like Dial.
//
// The context is used to cancel or time out the initial connection establishment. It does
// not affect subsequent interactions with the client.
func DialContext(ctx context.Context, rawurl string) (*Client, error) {
u, err := url.Parse(rawurl)
if err != nil {
return nil, err
}
switch u.Scheme {
case "http", "https":
return DialHTTP(rawurl)
case "ws", "wss":
return DialWebsocket(ctx, rawurl, "")
case "":
return DialIPC(ctx, rawurl)
default:
return nil, fmt.Errorf("no known transport for URL scheme %q", u.Scheme)
}
}

func newClient(initctx context.Context, connectFunc func(context.Context) (net.Conn, error))
(*Client, error) {
conn, err := connectFunc(initctx)
if err != nil {
return nil, err
}
_, isHTTP := conn.(*httpConn)

c := &Client{
writeConn:   conn,
isHTTP:      isHTTP,
connectFunc: connectFunc,
close:       make(chan struct{}),
didQuit:     make(chan struct{}),
reconnected: make(chan net.Conn),
readErr:     make(chan error),
readResp:    make(chan []*jsonrpcMessage),
requestOp:   make(chan *requestOp),
sendDone:    make(chan error, 1),
respWait:    make(map[string]*requestOp),
subs:        make(map[string]*ClientSubscription),
}
if !isHTTP {
go c.dispatch(conn)
}
return c, nil
}

func (c *Client) nextID() json.RawMessage {
id := atomic.AddUint32(&c.idCounter, 1)
return []byte(strconv.FormatUint(uint64(id), 10))
}

// SupportedModules calls the rpc_modules method, retrieving the list of
// APIs that are available on the server.
```

```
func (c *Client) SupportedModules() (map[string]string, error) {
var result map[string]string
ctx, cancel := context.WithTimeout(context.Background(), subscribeTimeout)
defer cancel()
err := c.CallContext(ctx, &result, "rpc_modules")
return result, err
}

// Close closes the client, aborting any in-flight requests.
func (c *Client) Close() {
if c.isHTTP {
return
}
select {
case c.close <- struct{}{}:
<-c.didQuit
case <-c.didQuit:
}
}

// Call performs a JSON-RPC call with the given arguments and unmarshals into
// result if no error occurred.
//
// The result must be a pointer so that package json can unmarshal into it. You
// can also pass nil, in which case the result is ignored.
func (c *Client) Call(result interface{}, method string, args ...interface{}) error {
ctx := context.Background()
return c.CallContext(ctx, result, method, args...)
}

// CallContext performs a JSON-RPC call with the given arguments. If the context is
// canceled before the call has successfully returned, CallContext returns immediately.
//
// The result must be a pointer so that package json can unmarshal into it. You
// can also pass nil, in which case the result is ignored.
func (c *Client) CallContext(ctx context.Context, result interface{}, method string, args
...interface{}) error {
msg, err := c.newMessage(method, args...)
if err != nil {
return err
}
op := &requestOp{ids: []json.RawMessage{msg.ID}, resp: make(chan *jsonrpcMessage, 1)}
if c.isHTTP {
err = c.sendHTTP(ctx, op, msg)
} else {
err = c.send(ctx, op, msg)
}
if err != nil {
return err
}

// dispatch has accepted the request and will close the channel it when it quits.
switch resp, err := op.wait(ctx); {
case err != nil:
return err
case resp.Error != nil:
return resp.Error
case len(resp.Result) == 0:
return ErrNoResult
default:
return json.Unmarshal(resp.Result, &result)
}
}

// BatchCall sends all given requests as a single batch and waits for the server
// to return a response for all of them.
//
// In contrast to Call, BatchCall only returns I/O errors. Any error specific to
// a request is reported through the Error field of the corresponding BatchElem.
//
// Note that batch calls may not be executed atomically on the server side.
func (c *Client) BatchCall(b []BatchElem) error {
ctx := context.Background()
return c.BatchCallContext(ctx, b)
}

// BatchCall sends all given requests as a single batch and waits for the server
// to return a response for all of them. The wait duration is bounded by the
// context's deadline.
//
// In contrast to CallContext, BatchCallContext only returns errors that have occurred
// while sending the request. Any error specific to a request is reported through the
// Error field of the corresponding BatchElem.
```

```
//
// Note that batch calls may not be executed atomically on the server side.
func (c *Client) BatchCallContext(ctx context.Context, b []BatchElem) error {
msgs := make([]*jsonrpcMessage, len(b))
op := &requestOp{
ids:  make([]json.RawMessage, len(b)),
resp: make(chan *jsonrpcMessage, len(b)),
}
for i, elem := range b {
msg, err := c.newMessage(elem.Method, elem.Args...)
if err != nil {
return err
}
msgs[i] = msg
op.ids[i] = msg.ID
}

var err error
if c.isHTTP {
err = c.sendBatchHTTP(ctx, op, msgs)
} else {
err = c.send(ctx, op, msgs)
}

// Wait for all responses to come back.
for n := 0; n < len(b) && err == nil; n++ {
var resp *jsonrpcMessage
resp, err = op.wait(ctx)
if err != nil {
break
}
// Find the element corresponding to this response.
// The element is guaranteed to be present because dispatch
// only sends valid IDs to our channel.
var elem *BatchElem
for i := range msgs {
if bytes.Equal(msgs[i].ID, resp.ID) {
elem = &b[i]
break
}
}
if resp.Error != nil {
```

```
elem.Error = resp.Error
continue
}
if len(resp.Result) == 0 {
elem.Error = ErrNoResult
continue
}
elem.Error = json.Unmarshal(resp.Result, elem.Result)
}
return err
}

// ShhSubscribe calls the "shh_subscribe" method with the given arguments,
// registering a subscription. Server notifications for the subscription are
// sent to the given channel. The element type of the channel must match the
// expected type of content returned by the subscription.
//
// The context argument cancels the RPC request that sets up the subscription but has no
// effect on the subscription after ShhSubscribe has returned.
//
// Slow subscribers will be dropped eventually. Client buffers up to 8000 notifications
// before considering the subscriber dead. The subscription Err channel will receive
// ErrSubscriptionQueueOverflow. Use a sufficiently large buffer on the channel or ensure
// that the channel usually has at least one reader to prevent this issue.
func (c *Client) ShhSubscribe(ctx context.Context, channel interface{}, args ...interface{})
(*ClientSubscription, error) {
// Check type of channel first.
chanVal := reflect.ValueOf(channel)
if chanVal.Kind() != reflect.Chan || chanVal.Type().ChanDir()&reflect.SendDir == 0 {
panic("first argument to ShhSubscribe must be a writable channel")
}
if chanVal.IsNil() {
panic("channel given to ShhSubscribe must not be nil")
}
if c.isHTTP {
return nil, ErrNotificationsUnsupported
}

msg, err := c.newMessage("shh"+subscribeMethodSuffix, args...)
if err != nil {
return nil, err
}
```

```
op := &requestOp{
ids:  []json.RawMessage{msg.ID},
resp: make(chan *jsonrpcMessage),
sub:  newClientSubscription(c, "shh", chanVal),
}

// Send the subscription request.
// The arrival and validity of the response is signaled on sub.quit.
if err := c.send(ctx, op, msg); err != nil {
return nil, err
}
if _, err := op.wait(ctx); err != nil {
return nil, err
}
return op.sub, nil
}

// EthSubscribe calls the "eth_subscribe" method with the given arguments,
// registering a subscription. Server notifications for the subscription are
// sent to the given channel. The element type of the channel must match the
// expected type of content returned by the subscription.
//
// The context argument cancels the RPC request that sets up the subscription but has no
// effect on the subscription after EthSubscribe has returned.
//
// Slow subscribers will be dropped eventually. Client buffers up to 8000 notifications
// before considering the subscriber dead. The subscription Err channel will receive
// ErrSubscriptionQueueOverflow. Use a sufficiently large buffer on the channel or ensure
// that the channel usually has at least one reader to prevent this issue.
func (c *Client) EthSubscribe(ctx context.Context, channel interface{}, args ...interface{})
(*ClientSubscription, error) {
// Check type of channel first.
chanVal := reflect.ValueOf(channel)
if chanVal.Kind() != reflect.Chan || chanVal.Type().ChanDir()&reflect.SendDir == 0 {
panic("first argument to EthSubscribe must be a writable channel")
}
if chanVal.IsNil() {
panic("channel given to EthSubscribe must not be nil")
}
if c.isHTTP {
return nil, ErrNotificationsUnsupported
}
```

```
msg, err := c.newMessage("eth"+subscribeMethodSuffix, args...)
if err != nil {
return nil, err
}
op := &requestOp{
ids:  []json.RawMessage{msg.ID},
resp: make(chan *jsonrpcMessage),
sub:  newClientSubscription(c, "eth", chanVal),
}

// Send the subscription request.
// The arrival and validity of the response is signaled on sub.quit.
if err := c.send(ctx, op, msg); err != nil {
return nil, err
}
if _, err := op.wait(ctx); err != nil {
return nil, err
}
return op.sub, nil
}

func (c *Client) newMessage(method string, paramsIn ...interface{}) (*jsonrpcMessage, error) {
params, err := json.Marshal(paramsIn)
if err != nil {
return nil, err
}
return &jsonrpcMessage{Version: "2.0", ID: c.nextID(), Method: method, Params: params}, nil
}

// send registers op with the dispatch loop, then sends msg on the connection.
// if sending fails, op is deregistered.
func (c *Client) send(ctx context.Context, op *requestOp, msg interface{}) error {
select {
case c.requestOp <- op:
log.Trace("", "msg", log.Lazy{Fn: func() string {
return fmt.Sprint("sending ", msg)
}})
err := c.write(ctx, msg)
c.sendDone <- err
return err
case <-ctx.Done():
```

```
// This can happen if the client is overloaded or unable to keep up with
// subscription notifications.
return ctx.Err()
case <-c.didQuit:
return ErrClientQuit
}
}

func (c *Client) write(ctx context.Context, msg interface{}) error {
deadline, ok := ctx.Deadline()
if !ok {
deadline = time.Now().Add(defaultWriteTimeout)
}
// The previous write failed. Try to establish a new connection.
if c.writeConn == nil {
if err := c.reconnect(ctx); err != nil {
return err
}
}
c.writeConn.SetWriteDeadline(deadline)
err := json.NewEncoder(c.writeConn).Encode(msg)
if err != nil {
c.writeConn = nil
}
return err
}

func (c *Client) reconnect(ctx context.Context) error {
newconn, err := c.connectFunc(ctx)
if err != nil {
log.Trace(fmt.Sprintf("reconnect failed: %v", err))
return err
}
select {
case c.reconnected <- newconn:
c.writeConn = newconn
return nil
case <-c.didQuit:
newconn.Close()
return ErrClientQuit
}
}
```

```
// dispatch is the main loop of the client.
// It sends read messages to waiting calls to Call and BatchCall
// and subscription notifications to registered subscriptions.
func (c *Client) dispatch(conn net.Conn) {
// Spawn the initial read loop.
go c.read(conn)

var (
lastOp        *requestOp    // tracks last send operation
requestOpLock = c.requestOp // nil while the send lock is held
reading       = true        // if true, a read loop is running
)
defer close(c.didQuit)
defer func() {
c.closeRequestOps(ErrClientQuit)
conn.Close()
if reading {
// Empty read channels until read is dead.
for {
select {
case <-c.readResp:
case <-c.readErr:
return
}
}
}
}()

for {
select {
case <-c.close:
return

// Read path.
case batch := <-c.readResp:
for _, msg := range batch {
switch {
case msg.isNotification():
log.Trace("", "msg", log.Lazy{Fn: func() string {
return fmt.Sprint("<-readResp: notification ", msg)
}})
c.handleNotification(msg)
case msg.isResponse():
log.Trace("", "msg", log.Lazy{Fn: func() string {
return fmt.Sprint("<-readResp: response ", msg)
}})
c.handleResponse(msg)
default:
log.Debug("", "msg", log.Lazy{Fn: func() string {
return fmt.Sprint("<-readResp: dropping weird message", msg)
}})
// TODO: maybe close
}
}

case err := <-c.readErr:
log.Debug(fmt.Sprintf("<-readErr: %v", err))
c.closeRequestOps(err)
conn.Close()
reading = false

case newconn := <-c.reconnected:
log.Debug(fmt.Sprintf("<-reconnected: (reading=%t) %v", reading, conn.RemoteAddr()))
if reading {
// Wait for the previous read loop to exit. This is a rare case.
conn.Close()
<-c.readErr
}
go c.read(newconn)
reading = true
conn = newconn

// Send path.
case op := <-requestOpLock:
// Stop listening for further send ops until the current one is done.
requestOpLock = nil
lastOp = op
for _, id := range op.ids {
c.respWait[string(id)] = op
}

case err := <-c.sendDone:
if err != nil {
```

```
// Remove response handlers for the last send. We remove those here
// because the error is already handled in Call or BatchCall. When the
// read loop goes down, it will signal all other current operations.
for _, id := range lastOp.ids {
delete(c.respWait, string(id))
}
}
// Listen for send ops again.
requestOpLock = c.requestOp
lastOp = nil
}
}
}

// closeRequestOps unblocks pending send ops and active subscriptions.
func (c *Client) closeRequestOps(err error) {
didClose := make(map[*requestOp]bool)

for id, op := range c.respWait {
// Remove the op so that later calls will not close op.resp again.
delete(c.respWait, id)

if !didClose[op] {
op.err = err
close(op.resp)
didClose[op] = true
}
}
for id, sub := range c.subs {
delete(c.subs, id)
sub.quitWithError(err, false)
}
}

func (c *Client) handleNotification(msg *jsonrpcMessage) {
if !strings.HasSuffix(msg.Method, notificationMethodSuffix) {
log.Debug(fmt.Sprint("dropping non-subscription message: ", msg))
return
}
var subResult struct {
ID     string          `json:"subscription"`
Result json.RawMessage `json:"result"`
```

```
}
if err := json.Unmarshal(msg.Params, &subResult); err != nil {
log.Debug(fmt.Sprint("dropping invalid subscription message: ", msg))
return
}
if c.subs[subResult.ID] != nil {
c.subs[subResult.ID].deliver(subResult.Result)
}
}

func (c *Client) handleResponse(msg *jsonrpcMessage) {
op := c.respWait[string(msg.ID)]
if op == nil {
log.Debug(fmt.Sprintf("unsolicited response %v", msg))
return
}
delete(c.respWait, string(msg.ID))
// For normal responses, just forward the reply to Call/BatchCall.
if op.sub == nil {
op.resp <- msg
return
}
// For subscription responses, start the subscription if the server
// indicates success. EthSubscribe gets unblocked in either case through
// the op.resp channel.
defer close(op.resp)
if msg.Error != nil {
op.err = msg.Error
return
}
if op.err = json.Unmarshal(msg.Result, &op.sub.subid); op.err == nil {
go op.sub.start()
c.subs[op.sub.subid] = op.sub
}
}

// Reading happens on a dedicated goroutine.

func (c *Client) read(conn net.Conn) error {
var (
buf json.RawMessage
dec = json.NewDecoder(conn)
```

```
)
readMessage := func() (rs []*jsonrpcMessage, err error) {
buf = buf[:0]
if err = dec.Decode(&buf); err != nil {
return nil, err
}
if isBatch(buf) {
err = json.Unmarshal(buf, &rs)
} else {
rs = make([]*jsonrpcMessage, 1)
err = json.Unmarshal(buf, &rs[0])
}
return rs, err
}

for {
resp, err := readMessage()
if err != nil {
c.readErr <- err
return err
}
c.readResp <- resp
}
}

// Subscriptions.

// A ClientSubscription represents a subscription established through EthSubscribe.
type ClientSubscription struct {
client    *Client
etype     reflect.Type
channel   reflect.Value
namespace string
subid     string
in        chan json.RawMessage

quitOnce sync.Once     // ensures quit is closed once
quit     chan struct{} // quit is closed when the subscription exits
errOnce  sync.Once     // ensures err is closed once
err      chan error
}
```

```
func newClientSubscription(c *Client, namespace string, channel reflect.Value) *ClientSubscription
{
sub := &ClientSubscription{
client:    c,
namespace: namespace,
etype:     channel.Type().Elem(),
channel:   channel,
quit:      make(chan struct{}),
err:       make(chan error, 1),
in:        make(chan json.RawMessage),
}
return sub
}

// Err returns the subscription error channel. The intended use of Err is to schedule
// resubscription when the client connection is closed unexpectedly.
//
// The error channel receives a value when the subscription has ended due
// to an error. The received error is nil if Close has been called
// on the underlying client and no other error has occurred.
//
// The error channel is closed when Unsubscribe is called on the subscription.
func (sub *ClientSubscription) Err() <-chan error {
return sub.err
}

// Unsubscribe unsubscribes the notification and closes the error channel.
// It can safely be called more than once.
func (sub *ClientSubscription) Unsubscribe() {
sub.quitWithError(nil, true)
sub.errOnce.Do(func() { close(sub.err) })
}

func (sub *ClientSubscription) quitWithError(err error, unsubscribeServer bool) {
sub.quitOnce.Do(func() {
// The dispatch loop won't be able to execute the unsubscribe call
// if it is blocked on deliver. Close sub.quit first because it
// unblocks deliver.
close(sub.quit)
if unsubscribeServer {
sub.requestUnsubscribe()
}
if err != nil {
if err == ErrClientQuit {
err = nil // Adhere to subscription semantics.
}
sub.err <- err
}
})
}

func (sub *ClientSubscription) deliver(result json.RawMessage) (ok bool) {
select {
case sub.in <- result:
return true
case <-sub.quit:
return false
}
}

func (sub *ClientSubscription) start() {
sub.quitWithError(sub.forward())
}

func (sub *ClientSubscription) forward() (err error, unsubscribeServer bool) {
cases := []reflect.SelectCase{
{Dir: reflect.SelectRecv, Chan: reflect.ValueOf(sub.quit)},
{Dir: reflect.SelectRecv, Chan: reflect.ValueOf(sub.in)},
{Dir: reflect.SelectSend, Chan: sub.channel},
}
buffer := list.New()
defer buffer.Init()
for {
var chosen int
var recv reflect.Value
if buffer.Len() == 0 {
// Idle, omit send case.
chosen, recv, _ = reflect.Select(cases[:2])
} else {
// Non-empty buffer, send the first queued item.
cases[2].Send = reflect.ValueOf(buffer.Front().Value)
chosen, recv, _ = reflect.Select(cases)
}
```

```
switch chosen {
case 0: // <-sub.quit
return nil, false
case 1: // <-sub.in
val, err := sub.unmarshal(recv.Interface().(json.RawMessage))
if err != nil {
return err, true
}
if buffer.Len() == maxClientSubscriptionBuffer {
return ErrSubscriptionQueueOverflow, true
}
buffer.PushBack(val)
case 2: // sub.channel<-
cases[2].Send = reflect.Value{} // Don't hold onto the value.
buffer.Remove(buffer.Front())
}
}
}

func (sub *ClientSubscription) unmarshal(result json.RawMessage) (interface{}, error) {
val := reflect.New(sub.etype)
err := json.Unmarshal(result, val.Interface())
return val.Elem().Interface(), err
}

func (sub *ClientSubscription) requestUnsubscribe() error {
var result interface{}
return sub.client.Call(&result, sub.namespace+unsubscribeMethodSuffix, sub.subid)
}
```

46:F:\git\coin\ethereum\go-ethereum\rpc\client_example_test.go

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

package rpc_test

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

"github.com/ethereum/go-ethereum/rpc"
```

```
)

// In this example, our client whishes to track the latest 'block number'
// known to the server. The server supports two methods:
//
// eth_getBlockByNumber("latest", {})
//    returns the latest block object.
//
// eth_subscribe("newBlocks")
//    creates a subscription which fires block objects when new blocks arrive.

type Block struct {
Number *big.Int
}

func ExampleClientSubscription() {
// Connect the client.
client, _ := rpc.Dial("ws://127.0.0.1:8485")
subch := make(chan Block)

// Ensure that subch receives the latest block.
go func() {
for i := 0; ; i++ {
if i > 0 {
time.Sleep(2 * time.Second)
}
subscribeBlocks(client, subch)
}
}()

// Print events from the subscription as they arrive.
for block := range subch {
fmt.Println("latest block:", block.Number)
}
}

// subscribeBlocks runs in its own goroutine and maintains
// a subscription for new blocks.
func subscribeBlocks(client *rpc.Client, subch chan Block) {
ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
defer cancel()
```

```
// Subscribe to new blocks.
sub, err := client.EthSubscribe(ctx, subch, "newBlocks")
if err != nil {
fmt.Println("subscribe error:", err)
return
}

// The connection is established now.
// Update the channel with the current block.
var lastBlock Block
if err := client.CallContext(ctx, &lastBlock, "eth_getBlockByNumber", "latest"); err != nil {
fmt.Println("can't get latest block:", err)
return
}
subch <- lastBlock

// The subscription will deliver events to the channel. Wait for the
// subscription to end for any reason, then loop around to re-establish
// the connection.
fmt.Println("connection lost: ", <-sub.Err())
}
```

47:F:\git\coin\ethereum\go-ethereum\rpc\client_test.go

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

package rpc

import (
"context"
"fmt"
"math/rand"
"net"
"net/http"
"net/http/httptest"
"os"
"reflect"
"runtime"
"sync"
"testing"
"time"

"github.com/davecgh/go-spew/spew"
```

```
"github.com/ethereum/go-ethereum/log"
)

func TestClientRequest(t *testing.T) {
server := newTestServer("service", new(Service))
defer server.Stop()
client := DialInProc(server)
defer client.Close()

var resp Result
if err := client.Call(&resp, "service_echo", "hello", 10, &Args{"world"}); err != nil {
t.Fatal(err)
}
if !reflect.DeepEqual(resp, Result{"hello", 10, &Args{"world"}}) {
t.Errorf("incorrect result %#v", resp)
}
}

func TestClientBatchRequest(t *testing.T) {
server := newTestServer("service", new(Service))
defer server.Stop()
client := DialInProc(server)
defer client.Close()

batch := []BatchElem{
{
Method: "service_echo",
Args:   []interface{}{"hello", 10, &Args{"world"}},
Result: new(Result),
},
{
Method: "service_echo",
Args:   []interface{}{"hello2", 11, &Args{"world"}},
Result: new(Result),
},
{
Method: "no_such_method",
Args:   []interface{}{1, 2, 3},
Result: new(int),
},
}
if err := client.BatchCall(batch); err != nil {
```

```
t.Fatal(err)
}
wantResult := []BatchElem{
{
Method: "service_echo",
Args:   []interface{}{"hello", 10, &Args{"world"}},
Result: &Result{"hello", 10, &Args{"world"}},
},
{
Method: "service_echo",
Args:   []interface{}{"hello2", 11, &Args{"world"}},
Result: &Result{"hello2", 11, &Args{"world"}},
},
{
Method: "no_such_method",
Args:   []interface{}{1, 2, 3},
Result: new(int),
Error:  &jsonError{Code: -32601, Message: "The method no_such_method_ does not exist/is not
available"},
},
}
if !reflect.DeepEqual(batch, wantResult) {
t.Errorf("batch results mismatch:\ngot %swant %s", spew.Sdump(batch),
spew.Sdump(wantResult))
}
}

// func TestClientCancelInproc(t *testing.T) { testClientCancel("inproc", t) }
func TestClientCancelWebsocket(t *testing.T) { testClientCancel("ws", t) }
func TestClientCancelHTTP(t *testing.T)      { testClientCancel("http", t) }
func TestClientCancelIPC(t *testing.T)       { testClientCancel("ipc", t) }

// This test checks that requests made through CallContext can be canceled by canceling
// the context.
func testClientCancel(transport string, t *testing.T) {
server := newTestServer("service", new(Service))
defer server.Stop()

// What we want to achieve is that the context gets canceled
// at various stages of request processing. The interesting cases
// are:
//  - cancel during dial
//  - cancel while performing a HTTP request
//  - cancel while waiting for a response
//
// To trigger those, the times are chosen such that connections
// are killed within the deadline for every other call (maxKillTimeout
// is 2x maxCancelTimeout).
//
// Once a connection is dead, there is a fair chance it won't connect
// successfully because the accept is delayed by 1s.
maxContextCancelTimeout := 300 * time.Millisecond
fl := &flakeyListener{
maxAcceptDelay: 1 * time.Second,
maxKillTimeout: 600 * time.Millisecond,
}

var client *Client
switch transport {
case "ws", "http":
c, hs := httpTestClient(server, transport, fl)
defer hs.Close()
client = c
case "ipc":
c, l := ipcTestClient(server, fl)
defer l.Close()
client = c
default:
panic("unknown transport: " + transport)
}

// These tests take a lot of time, run them all at once.
// You probably want to run with -parallel 1 or comment out
// the call to t.Parallel if you enable the logging.
t.Parallel()

// The actual test starts here.
var (
wg       sync.WaitGroup
nreqs    = 10
ncallers = 6
)
caller := func(index int) {
defer wg.Done()
```

```
for i := 0; i < nreqs; i++ {
var (
ctx     context.Context
cancel  func()
timeout = time.Duration(rand.Int63n(int64(maxContextCancelTimeout)))
)
if index < ncallers/2 {
// For half of the callers, create a context without deadline
// and cancel it later.
ctx, cancel = context.WithCancel(context.Background())
time.AfterFunc(timeout, cancel)
} else {
// For the other half, create a context with a deadline instead. This is
// different because the context deadline is used to set the socket write
// deadline.
ctx, cancel = context.WithTimeout(context.Background(), timeout)
}
// Now perform a call with the context.
// The key thing here is that no call will ever complete successfully.
err := client.CallContext(ctx, nil, "service_sleep", 2*maxContextCancelTimeout)
if err != nil {
log.Debug(fmt.Sprint("got expected error:", err))
} else {
t.Errorf("no error for call with %v wait time", timeout)
}
cancel()
}
}
wg.Add(ncallers)
for i := 0; i < ncallers; i++ {
go caller(i)
}
wg.Wait()
}

func TestClientSubscribeInvalidArg(t *testing.T) {
server := newTestServer("service", new(Service))
defer server.Stop()
client := DialInProc(server)
defer client.Close()

check := func(shouldPanic bool, arg interface{}) {
```

```
defer func() {
err := recover()
if shouldPanic && err == nil {
t.Errorf("EthSubscribe should've panicked for %#v", arg)
}
if !shouldPanic && err != nil {
t.Errorf("EthSubscribe shouldn't have panicked for %#v", arg)
buf := make([]byte, 1024*1024)
buf = buf[:runtime.Stack(buf, false)]
t.Error(err)
t.Error(string(buf))
}
}()
client.EthSubscribe(context.Background(), arg, "foo_bar")
}
check(true, nil)
check(true, 1)
check(true, (chan int)(nil))
check(true, make(<-chan int))
check(false, make(chan int))
check(false, make(chan<- int))
}

func TestClientSubscribe(t *testing.T) {
server := newTestServer("eth", new(NotificationTestService))
defer server.Stop()
client := DialInProc(server)
defer client.Close()

nc := make(chan int)
count := 10
sub, err := client.EthSubscribe(context.Background(), nc, "someSubscription", count, 0)
if err != nil {
t.Fatal("can't subscribe:", err)
}
for i := 0; i < count; i++ {
if val := <-nc; val != i {
t.Fatalf("value mismatch: got %d, want %d", val, i)
}
}

sub.Unsubscribe()
```

```
select {
case v := <-nc:
t.Fatal("received value after unsubscribe:", v)
case err := <-sub.Err():
if err != nil {
t.Fatalf("Err returned a non-nil error after explicit unsubscribe: %q", err)
}
case <-time.After(1 * time.Second):
t.Fatalf("subscription not closed within 1s after unsubscribe")
}
}

// In this test, the connection drops while EthSubscribe is
// waiting for a response.
func TestClientSubscribeClose(t *testing.T) {
service := &NotificationTestService{
gotHangSubscriptionReq:  make(chan struct{}),
unblockHangSubscription: make(chan struct{}),
}
server := newTestServer("eth", service)
defer server.Stop()
client := DialInProc(server)
defer client.Close()

var (
nc   = make(chan int)
errc = make(chan error)
sub  *ClientSubscription
err  error
)
go func() {
sub, err = client.EthSubscribe(context.Background(), nc, "hangSubscription", 999)
errc <- err
}()

<-service.gotHangSubscriptionReq
client.Close()
service.unblockHangSubscription <- struct{}{}

select {
case err := <-errc:
if err == nil {
```

```
t.Errorf("EthSubscribe returned nil error after Close")
}
if sub != nil {
t.Error("EthSubscribe returned non-nil subscription after Close")
}
case <-time.After(1 * time.Second):
t.Fatalf("EthSubscribe did not return within 1s after Close")
}
}

// This test checks that Client doesn't lock up when a single subscriber
// doesn't read subscription events.
func TestClientNotificationStorm(t *testing.T) {
server := newTestServer("eth", new(NotificationTestService))
defer server.Stop()

doTest := func(count int, wantError bool) {
client := DialInProc(server)
defer client.Close()
ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
defer cancel()

// Subscribe on the server. It will start sending many notifications
// very quickly.
nc := make(chan int)
sub, err := client.EthSubscribe(ctx, nc, "someSubscription", count, 0)
if err != nil {
t.Fatal("can't subscribe:", err)
}
defer sub.Unsubscribe()

// Process each notification, try to run a call in between each of them.
for i := 0; i < count; i++ {
select {
case val := <-nc:
if val != i {
t.Fatalf("(%d/%d) unexpected value %d", i, count, val)
}
case err := <-sub.Err():
if wantError && err != ErrSubscriptionQueueOverflow {
t.Fatalf("(%d/%d) got error %q, want %q", i, count, err, ErrSubscriptionQueueOverflow)
} else if !wantError {
```

```
t.Fatalf("(%d/%d) got unexpected error %q", i, count, err)
}
return
}
var r int
err := client.CallContext(ctx, &r, "eth_echo", i)
if err != nil {
if !wantError {
t.Fatalf("(%d/%d) call error: %v", i, count, err)
}
return
}
}
}

doTest(8000, false)
doTest(10000, true)
}

func TestClientHTTP(t *testing.T) {
server := newTestServer("service", new(Service))
defer server.Stop()

client, hs := httpTestClient(server, "http", nil)
defer hs.Close()
defer client.Close()

// Launch concurrent requests.
var (
results    = make([]Result, 100)
errc       = make(chan error)
wantResult = Result{"a", 1, new(Args)}
)
defer client.Close()
for i := range results {
i := i
go func() {
errc <- client.Call(&results[i], "service_echo",
wantResult.String, wantResult.Int, wantResult.Args)
}()
}
```

```
// Wait for all of them to complete.
timeout := time.NewTimer(5 * time.Second)
defer timeout.Stop()
for i := range results {
select {
case err := <-errc:
if err != nil {
t.Fatal(err)
}
case <-timeout.C:
t.Fatalf("timeout (got %d/%d) results)", i+1, len(results))
}
}

// Check results.
for i := range results {
if !reflect.DeepEqual(results[i], wantResult) {
t.Errorf("result %d mismatch: got %#v, want %#v", i, results[i], wantResult)
}
}
}

func TestClientReconnect(t *testing.T) {
startServer := func(addr string) (*Server, net.Listener) {
srv := newTestServer("service", new(Service))
l, err := net.Listen("tcp", addr)
if err != nil {
t.Fatal(err)
}
go http.Serve(l, srv.WebsocketHandler([]string{"*"}))
return srv, l
}

ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
defer cancel()

// Start a server and corresponding client.
s1, l1 := startServer("127.0.0.1:0")
client, err := DialContext(ctx, "ws://"+l1.Addr().String())
if err != nil {
t.Fatal("can't dial", err)
}
```

```
// Perform a call. This should work because the server is up.
var resp Result
if err := client.CallContext(ctx, &resp, "service_echo", "", 1, nil); err != nil {
t.Fatal(err)
}

// Shut down the server and try calling again. It shouldn't work.
l1.Close()
s1.Stop()
if err := client.CallContext(ctx, &resp, "service_echo", "", 2, nil); err == nil {
t.Error("successful call while the server is down")
t.Logf("resp: %#v", resp)
}

// Allow for some cool down time so we can listen on the same address again.
time.Sleep(2 * time.Second)

// Start it up again and call again. The connection should be reestablished.
// We spawn multiple calls here to check whether this hangs somehow.
s2, l2 := startServer(l1.Addr().String())
defer l2.Close()
defer s2.Stop()

start := make(chan struct{})
errors := make(chan error, 20)
for i := 0; i < cap(errors); i++ {
go func() {
<-start
var resp Result
errors <- client.CallContext(ctx, &resp, "service_echo", "", 3, nil)
}()
}
close(start)
errcount := 0
for i := 0; i < cap(errors); i++ {
if err = <-errors; err != nil {
errcount++
}
}
t.Log("err:", err)
if errcount > 1 {
```

```
t.Errorf("expected one error after disconnect, got %d", errcount)
}
}

func newTestServer(serviceName string, service interface{}) *Server {
server := NewServer()
if err := server.RegisterName(serviceName, service); err != nil {
panic(err)
}
return server
}

func httpTestClient(srv *Server, transport string, fl *flakeyListener) (*Client, *httptest.Server) {
// Create the HTTP server.
var hs *httptest.Server
switch transport {
case "ws":
hs = httptest.NewUnstartedServer(srv.WebsocketHandler([]string{"*"}))
case "http":
hs = httptest.NewUnstartedServer(srv)
default:
panic("unknown HTTP transport: " + transport)
}
// Wrap the listener if required.
if fl != nil {
fl.Listener = hs.Listener
hs.Listener = fl
}
// Connect the client.
hs.Start()
client, err := Dial(transport + "://" + hs.Listener.Addr().String())
if err != nil {
panic(err)
}
return client, hs
}

func ipcTestClient(srv *Server, fl *flakeyListener) (*Client, net.Listener) {
// Listen on a random endpoint.
endpoint := fmt.Sprintf("go-ethereum-test-ipc-%d-%d", os.Getpid(), rand.Int63())
if runtime.GOOS == "windows" {
endpoint = `\\.\pipe\` + endpoint
```

```
} else {
endpoint = os.TempDir() + "/" + endpoint
}
l, err := ipcListen(endpoint)
if err != nil {
panic(err)
}
// Connect the listener to the server.
if fl != nil {
fl.Listener = l
l = fl
}
go srv.ServeListener(l)
// Connect the client.
client, err := Dial(endpoint)
if err != nil {
panic(err)
}
return client, l
}

// flakeyListener kills accepted connections after a random timeout.
type flakeyListener struct {
net.Listener
maxKillTimeout time.Duration
maxAcceptDelay time.Duration
}

func (l *flakeyListener) Accept() (net.Conn, error) {
delay := time.Duration(rand.Int63n(int64(l.maxAcceptDelay)))
time.Sleep(delay)

c, err := l.Listener.Accept()
if err == nil {
timeout := time.Duration(rand.Int63n(int64(l.maxKillTimeout)))
time.AfterFunc(timeout, func() {
log.Debug(fmt.Sprintf("killing conn %v after %v", c.LocalAddr(), timeout))
c.Close()
})
}
return c, err
}
```

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

/*
Package rpc provides access to the exported methods of an object across a network
or other I/O connection. After creating a server instance objects can be registered,
making it visible from the outside. Exported methods that follow specific
conventions can be called remotely. It also has support for the publish/subscribe
pattern.

Methods that satisfy the following criteria are made available for remote access:
 - object must be exported
 - method must be exported
 - method returns 0, 1 (response or error) or 2 (response and error) values
 - method argument(s) must be exported or builtin types
 - method returned value(s) must be exported or builtin types

An example method:
 func (s *CalcService) Add(a, b int) (int, error)

When the returned error isn't nil the returned integer is ignored and the error is
send back to the client. Otherwise the returned integer is send back to the client.

Optional arguments are supported by accepting pointer values as arguments. E.g.
if we want to do the addition in an optional finite field we can accept a mod
argument as pointer value.

 func (s *CalService) Add(a, b int, mod *int) (int, error)

This RPC method can be called with 2 integers and a null value as third argument.
In that case the mod argument will be nil. Or it can be called with 3 integers,
in that case mod will be pointing to the given third argument. Since the optional
argument is the last argument the RPC package will also accept 2 integers as
arguments. It will pass the mod argument as nil to the RPC method.

The server offers the ServeCodec method which accepts a ServerCodec instance. It will
read requests from the codec, process the request and sends the response back to the
client using the codec. The server can execute requests concurrently. Responses
can be sent back to the client out of order.

An example server which uses the JSON codec:
 type CalculatorService struct {}

 func (s *CalculatorService) Add(a, b int) int {
return a + b
 }

 func (s *CalculatorService Div(a, b int) (int, error) {
if b == 0 {
return 0, errors.New("divide by zero")
}
return a/b, nil
 }

 calculator := new(CalculatorService)
 server := NewServer()
 server.RegisterName("calculator", calculator")

 l, _ := net.ListenUnix("unix", &net.UnixAddr{Net: "unix", Name: "/tmp/calculator.sock"})
 for {
c, _ := l.AcceptUnix()
codec := v2.NewJSONCodec(c)
go server.ServeCodec(codec)
 }
```

The package also supports the publish subscribe pattern through the use of subscriptions.
A method that is considered eligible for notifications must satisfy the following criteria:

- object must be exported
- method must be exported
- first method argument type must be context.Context
- method argument(s) must be exported or builtin types
- method must return the tuple Subscription, error

An example method:

```
 func (s *BlockChainService) NewBlocks(ctx context.Context) (Subscription, error) {
 ...
 }
```

Subscriptions are deleted when:

- the user sends an unsubscribe request
- the connection which was used to create the subscription is closed. This can be initiated by the client and server. The server will close the connection on an write error or when the queue of buffered notifications gets too big.

```
*/
package rpc

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

package rpc

import "fmt"

// request is for an unknown service
type methodNotFoundError struct {
service string
method  string
}

func (e *methodNotFoundError) ErrorCode() int { return -32601 }

func (e *methodNotFoundError) Error() string {
return fmt.Sprintf("The method %s%s%s does not exist/is not available", e.service,
serviceMethodSeparator, e.method)
}

// received message isn't a valid request
type invalidRequestError struct{ message string }

func (e *invalidRequestError) ErrorCode() int { return -32600 }

func (e *invalidRequestError) Error() string { return e.message }

// received message is invalid
type invalidMessageError struct{ message string }

func (e *invalidMessageError) ErrorCode() int { return -32700 }

func (e *invalidMessageError) Error() string { return e.message }

// unable to decode supplied params, or an invalid number of parameters
type invalidParamsError struct{ message string }

func (e *invalidParamsError) ErrorCode() int { return -32602 }
func (e *invalidParamsError) Error() string { return e.message }

// logic error, callback returned an error
type callbackError struct{ message string }

func (e *callbackError) ErrorCode() int { return -32000 }

func (e *callbackError) Error() string { return e.message }

// issued when a request is received after the server is issued to stop.
type shutdownError struct{}

func (e *shutdownError) ErrorCode() int { return -32000 }

func (e *shutdownError) Error() string { return "server is shutting down" }

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

package rpc

import (
"bytes"
"context"
"encoding/json"
"fmt"
"io"
"io/ioutil"
"net"
"net/http"
"sync"
"time"

"github.com/rs/cors"
)

const (
maxHTTPRequestContentLength = 1024 * 128
)

var nullAddr, _ = net.ResolveTCPAddr("tcp", "127.0.0.1:0")
```

```
type httpConn struct {
client    *http.Client
req       *http.Request
closeOnce sync.Once
closed    chan struct{}
}

// httpConn is treated specially by Client.
func (hc *httpConn) LocalAddr() net.Addr              { return nullAddr }
func (hc *httpConn) RemoteAddr() net.Addr             { return nullAddr }
func (hc *httpConn) SetReadDeadline(time.Time) error  { return nil }
func (hc *httpConn) SetWriteDeadline(time.Time) error { return nil }
func (hc *httpConn) SetDeadline(time.Time) error      { return nil }
func (hc *httpConn) Write([]byte) (int, error)        { panic("Write called") }

func (hc *httpConn) Read(b []byte) (int, error) {
<-hc.closed
return 0, io.EOF
}

func (hc *httpConn) Close() error {
hc.closeOnce.Do(func() { close(hc.closed) })
return nil
}

// DialHTTP creates a new RPC clients that connection to an RPC server over HTTP.
func DialHTTP(endpoint string) (*Client, error) {
req, err := http.NewRequest("POST", endpoint, nil)
if err != nil {
return nil, err
}
req.Header.Set("Content-Type", "application/json")
req.Header.Set("Accept", "application/json")

initctx := context.Background()
return newClient(initctx, func(context.Context) (net.Conn, error) {
return &httpConn{client: new(http.Client), req: req, closed: make(chan struct{})}, nil
})
}

func (c *Client) sendHTTP(ctx context.Context, op *requestOp, msg interface{}) error {
hc := c.writeConn.(*httpConn)
```

```
respBody, err := hc.doRequest(ctx, msg)
if err != nil {
return err
}
defer respBody.Close()
var respmsg jsonrpcMessage
if err := json.NewDecoder(respBody).Decode(&respmsg); err != nil {
return err
}
op.resp <- &respmsg
return nil
}

func (c *Client) sendBatchHTTP(ctx context.Context, op *requestOp, msgs []*jsonrpcMessage)
error {
hc := c.writeConn.(*httpConn)
respBody, err := hc.doRequest(ctx, msgs)
if err != nil {
return err
}
defer respBody.Close()
var respmsgs []jsonrpcMessage
if err := json.NewDecoder(respBody).Decode(&respmsgs); err != nil {
return err
}
for i := 0; i < len(respmsgs); i++ {
op.resp <- &respmsgs[i]
}
return nil
}

func (hc *httpConn) doRequest(ctx context.Context, msg interface{}) (io.ReadCloser, error) {
body, err := json.Marshal(msg)
if err != nil {
return nil, err
}
req := hc.req.WithContext(ctx)
req.Body = ioutil.NopCloser(bytes.NewReader(body))
req.ContentLength = int64(len(body))

resp, err := hc.client.Do(req)
if err != nil {
```

```
return nil, err
}
return resp.Body, nil
}

// httpReadWriteNopCloser wraps a io.Reader and io.Writer with a NOP Close method.
type httpReadWriteNopCloser struct {
io.Reader
io.Writer
}

// Close does nothing and returns always nil
func (t *httpReadWriteNopCloser) Close() error {
return nil
}

// NewHTTPServer creates a new HTTP RPC server around an API provider.
//
// Deprecated: Server implements http.Handler
func NewHTTPServer(cors []string, srv *Server) *http.Server {
return &http.Server{Handler: newCorsHandler(srv, cors)}
}

// ServeHTTP serves JSON-RPC requests over HTTP.
func (srv *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
if r.ContentLength > maxHTTPRequestContentLength {
http.Error(w,
fmt.Sprintf("content length too large (%d>%d)", r.ContentLength,
maxHTTPRequestContentLength),
http.StatusRequestEntityTooLarge)
return
}
w.Header().Set("content-type", "application/json")

// create a codec that reads direct from the request body until
// EOF and writes the response to w and order the server to process
// a single request.
codec := NewJSONCodec(&httpReadWriteNopCloser{r.Body, w})
defer codec.Close()
srv.ServeSingleRequest(codec, OptionMethodInvocation)
}
```

```
func newCorsHandler(srv *Server, allowedOrigins []string) http.Handler {
// disable CORS support if user has not specified a custom CORS configuration
if len(allowedOrigins) == 0 {
return srv
}

c := cors.New(cors.Options{
AllowedOrigins: allowedOrigins,
AllowedMethods: []string{"POST", "GET"},
MaxAge:         600,
AllowedHeaders: []string{"*"},
})
return c.Handler(srv)
}
```

51:F:\git\coin\ethereum\go-ethereum\rpc\inproc.go

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

package rpc

import (
"context"
"net"
)

// NewInProcClient attaches an in-process connection to the given RPC server.
func DialInProc(handler *Server) *Client {
initctx := context.Background()
c, _ := newClient(initctx, func(context.Context) (net.Conn, error) {
p1, p2 := net.Pipe()
go handler.ServeCodec(NewJSONCodec(p1), OptionMethodInvocation|OptionSubscriptions)
return p2, nil
})
return c
}
```

52:F:\git\coin\ethereum\go-ethereum\rpc\ipc.go

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

package rpc

import (
"context"
"fmt"
"net"

"github.com/ethereum/go-ethereum/log"
)

// CreateIPCListener creates an listener, on Unix platforms this is a unix socket, on
// Windows this is a named pipe
func CreateIPCListener(endpoint string) (net.Listener, error) {
return ipcListen(endpoint)
}

// ServeListener accepts connections on l, serving JSON-RPC on them.
func (srv *Server) ServeListener(l net.Listener) error {
for {
conn, err := l.Accept()
if err != nil {
return err
}
log.Trace(fmt.Sprint("accepted conn", conn.RemoteAddr()))
go srv.ServeCodec(NewJSONCodec(conn), OptionMethodInvocation|OptionSubscriptions)
}
}

// DialIPC create a new IPC client that connects to the given endpoint. On Unix it assumes
// the endpoint is the full path to a unix socket, and Windows the endpoint is an
// identifier for a named pipe.
//
// The context is used for the initial connection establishment. It does not
// affect subsequent interactions with the client.
func DialIPC(ctx context.Context, endpoint string) (*Client, error) {
return newClient(ctx, func(ctx context.Context) (net.Conn, error) {
return newIPCConnection(ctx, endpoint)
})
}

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

// +build darwin dragonfly freebsd linux nacl netbsd openbsd solaris
```

```
package rpc

import (
"context"
"net"
"os"
"path/filepath"
)

// ipcListen will create a Unix socket on the given endpoint.
func ipcListen(endpoint string) (net.Listener, error) {
// Ensure the IPC path exists and remove any previous leftover
if err := os.MkdirAll(filepath.Dir(endpoint), 0751); err != nil {
return nil, err
}
os.Remove(endpoint)
l, err := net.Listen("unix", endpoint)
if err != nil {
return nil, err
}
os.Chmod(endpoint, 0600)
return l, nil
}

// newIPCConnection will connect to a Unix socket on the given endpoint.
func newIPCConnection(ctx context.Context, endpoint string) (net.Conn, error) {
return dialContext(ctx, "unix", endpoint)
}

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

// +build windows

package rpc

import (
"context"
"net"
"time"

"gopkg.in/natefinch/npipe.v2"
```

```
)

// This is used if the dialing context has no deadline. It is much smaller than the
// defaultDialTimeout because named pipes are local and there is no need to wait so long.
const defaultPipeDialTimeout = 2 * time.Second

// ipcListen will create a named pipe on the given endpoint.
func ipcListen(endpoint string) (net.Listener, error) {
return npipe.Listen(endpoint)
}

// newIPCConnection will connect to a named pipe with the given endpoint as name.
func newIPCConnection(ctx context.Context, endpoint string) (net.Conn, error) {
timeout := defaultPipeDialTimeout
if deadline, ok := ctx.Deadline(); ok {
timeout = deadline.Sub(time.Now())
if timeout < 0 {
timeout = 0
}
}
return npipe.DialTimeout(endpoint, timeout)
}

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

package rpc

import (
"bytes"
"encoding/json"
"fmt"
"io"
"reflect"
"strconv"
"strings"
"sync"

"github.com/ethereum/go-ethereum/log"
)

const (
jsonrpcVersion           = "2.0"
serviceMethodSeparator   = "_"
subscribeMethodSuffix    = "_subscribe"
unsubscribeMethodSuffix  = "_unsubscribe"
notificationMethodSuffix = "_subscription"
)

type jsonRequest struct {
Method  string          `json:"method"`
Version string          `json:"jsonrpc"`
Id      json.RawMessage `json:"id,omitempty"`
Payload json.RawMessage `json:"params,omitempty"`
}

type jsonSuccessResponse struct {
Version string      `json:"jsonrpc"`
Id      interface{} `json:"id,omitempty"`
Result  interface{} `json:"result"`
}

type jsonError struct {
Code    int         `json:"code"`
Message string      `json:"message"`
Data    interface{} `json:"data,omitempty"`
}

type jsonErrResponse struct {
Version string      `json:"jsonrpc"`
Id      interface{} `json:"id,omitempty"`
Error   jsonError   `json:"error"`
}

type jsonSubscription struct {
Subscription string      `json:"subscription"`
Result       interface{} `json:"result,omitempty"`
}

type jsonNotification struct {
Version string           `json:"jsonrpc"`
Method  string           `json:"method"`
Params  jsonSubscription `json:"params"`
}
```

```
// jsonCodec reads and writes JSON-RPC messages to the underlying connection. It
// also has support for parsing arguments and serializing (result) objects.
type jsonCodec struct {
closer sync.Once          // close closed channel once
closed chan interface{}   // closed on Close
decMu  sync.Mutex         // guards d
d      *json.Decoder      // decodes incoming requests
encMu  sync.Mutex         // guards e
e      *json.Encoder      // encodes responses
rw     io.ReadWriteCloser // connection
}

func (err *jsonError) Error() string {
if err.Message == "" {
return fmt.Sprintf("json-rpc error %d", err.Code)
}
return err.Message
}

func (err *jsonError) ErrorCode() int {
return err.Code
}

// NewJSONCodec creates a new RPC server codec with support for JSON-RPC 2.0
func NewJSONCodec(rwc io.ReadWriteCloser) ServerCodec {
d := json.NewDecoder(rwc)
d.UseNumber()
return &jsonCodec{closed: make(chan interface{}), d: d, e: json.NewEncoder(rwc), rw: rwc}
}

// isBatch returns true when the first non-whitespace characters is '['
func isBatch(msg json.RawMessage) bool {
for _, c := range msg {
// skip insignificant whitespace (http://www.ietf.org/rfc/rfc4627.txt)
if c == 0x20 || c == 0x09 || c == 0x0a || c == 0x0d {
continue
}
return c == '['
}
return false
}
```

```
// ReadRequestHeaders will read new requests without parsing the arguments. It will
// return a collection of requests, an indication if these requests are in batch
// form or an error when the incoming message could not be read/parsed.
func (c *jsonCodec) ReadRequestHeaders() ([]rpcRequest, bool, Error) {
c.decMu.Lock()
defer c.decMu.Unlock()

var incomingMsg json.RawMessage
if err := c.d.Decode(&incomingMsg); err != nil {
return nil, false, &invalidRequestError{err.Error()}
}

if isBatch(incomingMsg) {
return parseBatchRequest(incomingMsg)
}

return parseRequest(incomingMsg)
}

// checkReqId returns an error when the given reqId isn't valid for RPC method calls.
// valid id's are strings, numbers or null
func checkReqId(reqId json.RawMessage) error {
if len(reqId) == 0 {
return fmt.Errorf("missing request id")
}
if _, err := strconv.ParseFloat(string(reqId), 64); err == nil {
return nil
}
var str string
if err := json.Unmarshal(reqId, &str); err == nil {
return nil
}
return fmt.Errorf("invalid request id")
}

// parseRequest will parse a single request from the given RawMessage. It will return
// the parsed request, an indication if the request was a batch or an error when
// the request could not be parsed.
func parseRequest(incomingMsg json.RawMessage) ([]rpcRequest, bool, Error) {
var in jsonRequest
if err := json.Unmarshal(incomingMsg, &in); err != nil {
```

```
return nil, false, &invalidMessageError{err.Error()}
}

if err := checkReqId(in.Id); err != nil {
return nil, false, &invalidMessageError{err.Error()}
}

// subscribe are special, they will always use `subscribeMethod` as first param in the payload
if strings.HasSuffix(in.Method, subscribeMethodSuffix) {
reqs := []rpcRequest{{id: &in.Id, isPubSub: true}}
if len(in.Payload) > 0 {
// first param must be subscription name
var subscribeMethod [1]string
if err := json.Unmarshal(in.Payload, &subscribeMethod); err != nil {
log.Debug(fmt.Sprintf("Unable to parse subscription method: %v\n", err))
return nil, false, &invalidRequestError{"Unable to parse subscription request"}
}

reqs[0].service, reqs[0].method = strings.TrimSuffix(in.Method, subscribeMethodSuffix),
subscribeMethod[0]
reqs[0].params = in.Payload
return reqs, false, nil
}
return nil, false, &invalidRequestError{"Unable to parse subscription request"}
}

if strings.HasSuffix(in.Method, unsubscribeMethodSuffix) {
return []rpcRequest{{id: &in.Id, isPubSub: true,
method: in.Method, params: in.Payload}}, false, nil
}

elems := strings.Split(in.Method, serviceMethodSeparator)
if len(elems) != 2 {
return nil, false, &methodNotFoundError{in.Method, ""}
}

// regular RPC call
if len(in.Payload) == 0 {
return []rpcRequest{{service: elems[0], method: elems[1], id: &in.Id}}, false, nil
}

return []rpcRequest{{service: elems[0], method: elems[1], id: &in.Id, params: in.Payload}}, false, nil
```

```
}

// parseBatchRequest will parse a batch request into a collection of requests from the given
RawMessage, an indication
// if the request was a batch or an error when the request could not be read.
func parseBatchRequest(incomingMsg json.RawMessage) ([]rpcRequest, bool, Error) {
var in []jsonRequest
if err := json.Unmarshal(incomingMsg, &in); err != nil {
return nil, false, &invalidMessageError{err.Error()}
}

requests := make([]rpcRequest, len(in))
for i, r := range in {
if err := checkReqId(r.Id); err != nil {
return nil, false, &invalidMessageError{err.Error()}
}

id := &in[i].Id

// subscribe are special, they will always use `subscriptionMethod` as first param in the payload
if strings.HasSuffix(r.Method, subscribeMethodSuffix) {
requests[i] = rpcRequest{id: id, isPubSub: true}
if len(r.Payload) > 0 {
// first param must be subscription name
var subscribeMethod [1]string
if err := json.Unmarshal(r.Payload, &subscribeMethod); err != nil {
log.Debug(fmt.Sprintf("Unable to parse subscription method: %v\n", err))
return nil, false, &invalidRequestError{"Unable to parse subscription request"}
}

requests[i].service, requests[i].method = strings.TrimSuffix(r.Method, subscribeMethodSuffix),
subscribeMethod[0]
requests[i].params = r.Payload
continue
}

return nil, true, &invalidRequestError{"Unable to parse (un)subscribe request arguments"}
}

if strings.HasSuffix(r.Method, unsubscribeMethodSuffix) {
requests[i] = rpcRequest{id: id, isPubSub: true, method: r.Method, params: r.Payload}
continue
```

```
}

if len(r.Payload) == 0 {
requests[i] = rpcRequest{id: id, params: nil}
} else {
requests[i] = rpcRequest{id: id, params: r.Payload}
}
if elem := strings.Split(r.Method, serviceMethodSeparator); len(elem) == 2 {
requests[i].service, requests[i].method = elem[0], elem[1]
} else {
requests[i].err = &methodNotFoundError{r.Method, ""}
}
}

return requests, true, nil
}

// ParseRequestArguments tries to parse the given params (json.RawMessage) with the given
// types. It returns the parsed values or an error when the parsing failed.
func (c *jsonCodec) ParseRequestArguments(argTypes []reflect.Type, params interface{})
([]reflect.Value, Error) {
if args, ok := params.(json.RawMessage); !ok {
return nil, &invalidParamsError{"Invalid params supplied"}
} else {
return parsePositionalArguments(args, argTypes)
}
}

// parsePositionalArguments tries to parse the given args to an array of values with the
// given types. It returns the parsed values or an error when the args could not be
// parsed. Missing optional arguments are returned as reflect.Zero values.
func parsePositionalArguments(rawArgs json.RawMessage, types []reflect.Type) ([]reflect.Value,
Error) {
// Read beginning of the args array.
dec := json.NewDecoder(bytes.NewReader(rawArgs))
if tok, _ := dec.Token(); tok != json.Delim('[') {
return nil, &invalidParamsError{"non-array args"}
}
// Read args.
args := make([]reflect.Value, 0, len(types))
for i := 0; dec.More(); i++ {
if i >= len(types) {
```

```
return nil, &invalidParamsError{fmt.Sprintf("too many arguments, want at most %d", len(types))}
}
argval := reflect.New(types[i])
if err := dec.Decode(argval.Interface()); err != nil {
return nil, &invalidParamsError{fmt.Sprintf("invalid argument %d: %v", i, err)}
}
if argval.IsNil() && types[i].Kind() != reflect.Ptr {
return nil, &invalidParamsError{fmt.Sprintf("missing value for required argument %d", i)}
}
args = append(args, argval.Elem())
}
// Read end of args array.
if _, err := dec.Token(); err != nil {
return nil, &invalidParamsError{err.Error()}
}
// Set any missing args to nil.
for i := len(args); i < len(types); i++ {
if types[i].Kind() != reflect.Ptr {
return nil, &invalidParamsError{fmt.Sprintf("missing value for required argument %d", i)}
}
args = append(args, reflect.Zero(types[i]))
}
return args, nil
}

// CreateResponse will create a JSON-RPC success response with the given id and reply as
result.
func (c *jsonCodec) CreateResponse(id interface{}, reply interface{}) interface{} {
if isHexNum(reflect.TypeOf(reply)) {
return &jsonSuccessResponse{Version: jsonrpcVersion, Id: id, Result: fmt.Sprintf(`%#x`, reply)}
}
return &jsonSuccessResponse{Version: jsonrpcVersion, Id: id, Result: reply}
}

// CreateErrorResponse will create a JSON-RPC error response with the given id and error.
func (c *jsonCodec) CreateErrorResponse(id interface{}, err Error) interface{} {
return &jsonErrResponse{Version: jsonrpcVersion, Id: id, Error: jsonError{Code: err.ErrorCode(),
Message: err.Error()}}
}

// CreateErrorResponseWithInfo will create a JSON-RPC error response with the given id and
error.
```

```
// info is optional and contains additional information about the error. When an empty string is
passed it is ignored.
func (c *jsonCodec) CreateErrorResponseWithInfo(id interface{}, err Error, info interface{})
interface{} {
return &jsonErrResponse{Version: jsonrpcVersion, Id: id,
Error: jsonError{Code: err.ErrorCode(), Message: err.Error(), Data: info}}
}

// CreateNotification will create a JSON-RPC notification with the given subscription id and event
as params.
func (c *jsonCodec) CreateNotification(subid, namespace string, event interface{}) interface{} {
if isHexNum(reflect.TypeOf(event)) {
return &jsonNotification{Version: jsonrpcVersion, Method: namespace + notificationMethodSuffix,
Params: jsonSubscription{Subscription: subid, Result: fmt.Sprintf(`%#x`, event)}}
}

return &jsonNotification{Version: jsonrpcVersion, Method: namespace + notificationMethodSuffix,
Params: jsonSubscription{Subscription: subid, Result: event}}
}

// Write message to client
func (c *jsonCodec) Write(res interface{}) error {
c.encMu.Lock()
defer c.encMu.Unlock()

return c.e.Encode(res)
}

// Close the underlying connection
func (c *jsonCodec) Close() {
c.closer.Do(func() {
close(c.closed)
c.rw.Close()
})
}

// Closed returns a channel which will be closed when Close is called
func (c *jsonCodec) Closed() <-chan interface{} {
return c.closed
}
```

56:F:\git\coin\ethereum\go-ethereum\rpc\json_test.go

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

package rpc

import (
"bufio"
"bytes"
"encoding/json"
"reflect"
"strconv"
"testing"
)

type RWC struct {
*bufio.ReadWriter
}

func (rwc *RWC) Close() error {
return nil
}

func TestJSONRequestParsing(t *testing.T) {
server := NewServer()
service := new(Service)

if err := server.RegisterName("calc", service); err != nil {
t.Fatalf("%v", err)
}

req := bytes.NewBufferString(`{"id": 1234, "jsonrpc": "2.0", "method": "calc_add", "params": [11,
22]}`)
var str string
reply := bytes.NewBufferString(str)
rw := &RWC{bufio.NewReadWriter(bufio.NewReader(req), bufio.NewWriter(reply))}

codec := NewJSONCodec(rw)

requests, batch, err := codec.ReadRequestHeaders()
if err != nil {
t.Fatalf("%v", err)
}
```

```
if batch {
t.Fatalf("Request isn't a batch")
}

if len(requests) != 1 {
t.Fatalf("Expected 1 request but got %d requests - %v", len(requests), requests)
}

if requests[0].service != "calc" {
t.Fatalf("Expected service 'calc' but got '%s'", requests[0].service)
}

if requests[0].method != "add" {
t.Fatalf("Expected method 'Add' but got '%s'", requests[0].method)
}

if rawId, ok := requests[0].id.(*json.RawMessage); ok {
id, e := strconv.ParseInt(string(*rawId), 0, 64)
if e != nil {
t.Fatalf("%v", e)
}
if id != 1234 {
t.Fatalf("Expected id 1234 but got %d", id)
}
} else {
t.Fatalf("invalid request, expected *json.RawMesage got %T", requests[0].id)
}

var arg int
args := []reflect.Type{reflect.TypeOf(arg), reflect.TypeOf(arg)}

v, err := codec.ParseRequestArguments(args, requests[0].params)
if err != nil {
t.Fatalf("%v", err)
}

if len(v) != 2 {
t.Fatalf("Expected 2 argument values, got %d", len(v))
}

if v[0].Int() != 11 || v[1].Int() != 22 {
t.Fatalf("expected %d == 11 && %d == 22", v[0].Int(), v[1].Int())
```

```
}
}

func TestJSONRequestParamsParsing(t *testing.T) {

var (
stringT = reflect.TypeOf("")
intT    = reflect.TypeOf(0)
intPtrT = reflect.TypeOf(new(int))

stringV = reflect.ValueOf("abc")
i       = 1
intV    = reflect.ValueOf(i)
intPtrV = reflect.ValueOf(&i)
)

var validTests = []struct {
input    string
argTypes []reflect.Type
expected []reflect.Value
}{
{`[]`, []reflect.Type{}, []reflect.Value{}},
{`[]`, []reflect.Type{intPtrT}, []reflect.Value{intPtrV}},
{`[1]`, []reflect.Type{intT}, []reflect.Value{intV}},
{`[1,"abc"]`, []reflect.Type{intT, stringT}, []reflect.Value{intV, stringV}},
{`[null]`, []reflect.Type{intPtrT}, []reflect.Value{intPtrV}},
{`[null,"abc"]`, []reflect.Type{intPtrT, stringT, intPtrT}, []reflect.Value{intPtrV, stringV, intPtrV}},
{`[null,"abc",null]`, []reflect.Type{intPtrT, stringT, intPtrT}, []reflect.Value{intPtrV, stringV, intPtrV}},
}

codec := jsonCodec{}

for _, test := range validTests {
params := (json.RawMessage)([]byte(test.input))
args, err := codec.ParseRequestArguments(test.argTypes, params)

if err != nil {
t.Fatal(err)
}

var match []interface{}
json.Unmarshal([]byte(test.input), &match)
```

```
if len(args) != len(test.argTypes) {
t.Fatalf("expected %d parsed args, got %d", len(test.argTypes), len(args))
}

for i, arg := range args {
expected := test.expected[i]

if arg.Kind() != expected.Kind() {
t.Errorf("expected type for param %d in %s", i, test.input)
}

if arg.Kind() == reflect.Int && arg.Int() != expected.Int() {
t.Errorf("expected int(%d), got int(%d) in %s", expected.Int(), arg.Int(), test.input)
}

if arg.Kind() == reflect.String && arg.String() != expected.String() {
t.Errorf("expected string(%s), got string(%s) in %s", expected.String(), arg.String(), test.input)
}
}
}

var invalidTests = []struct {
input    string
argTypes []reflect.Type
}{
{`[]`, []reflect.Type{intT}},
{`[null]`, []reflect.Type{intT}},
{`[1]`, []reflect.Type{stringT}},
{`[1,2]`, []reflect.Type{stringT}},
{`["abc", null]`, []reflect.Type{stringT, intT}},
}

for i, test := range invalidTests {
if _, err := codec.ParseRequestArguments(test.argTypes, test.input); err == nil {
t.Errorf("expected test %d - %s to fail", i, test.input)
}
}
}
```

57:F:\git\coin\ethereum\go-ethereum\rpc\server.go

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

```
package rpc

import (
"context"
"fmt"
"reflect"
"runtime"
"strings"
"sync"
"sync/atomic"

"github.com/ethereum/go-ethereum/log"
"gopkg.in/fatih/set.v0"
)

const (
notificationBufferSize = 10000 // max buffered notifications before codec is closed

MetadataApi = "rpc"
)

// CodecOption specifies which type of messages this codec supports
type CodecOption int

const (
// OptionMethodInvocation is an indication that the codec supports RPC method calls
OptionMethodInvocation CodecOption = 1 << iota

// OptionSubscriptions is an indication that the codec suports RPC notifications
OptionSubscriptions = 1 << iota // support pub sub
)

// NewServer will create a new server instance with no registered handlers.
func NewServer() *Server {
server := &Server{
services:      make(serviceRegistry),
subscriptions: make(subscriptionRegistry),
codecs:        set.New(),
run:           1,
}
```

```
// register a default service which will provide meta information about the RPC service such as the
services and
// methods it offers.
rpcService := &RPCService{server}
server.RegisterName(MetadataApi, rpcService)

return server
}

// RPCService gives meta information about the server.
// e.g. gives information about the loaded modules.
type RPCService struct {
server *Server
}

// Modules returns the list of RPC services with their version number
func (s *RPCService) Modules() map[string]string {
modules := make(map[string]string)
for name := range s.server.services {
modules[name] = "1.0"
}
return modules
}

// RegisterName will create a service for the given rcvr type under the given name. When no
methods on the given rcvr
// match the criteria to be either a RPC method or a subscription an error is returned. Otherwise a
new service is
// created and added to the service collection this server instance serves.
func (s *Server) RegisterName(name string, rcvr interface{}) error {
if s.services == nil {
s.services = make(serviceRegistry)
}

svc := new(service)
svc.typ = reflect.TypeOf(rcvr)
rcvrVal := reflect.ValueOf(rcvr)

if name == "" {
return fmt.Errorf("no service name for type %s", svc.typ.String())
}
if !isExported(reflect.Indirect(rcvrVal).Type().Name()) {
```

```
return fmt.Errorf("%s is not exported", reflect.Indirect(rcvrVal).Type().Name())
}

methods, subscriptions := suitableCallbacks(rcvrVal, svc.typ)

// already a previous service register under given sname, merge methods/subscriptions
if regsvc, present := s.services[name]; present {
if len(methods) == 0 && len(subscriptions) == 0 {
return fmt.Errorf("Service %T doesn't have any suitable methods/subscriptions to expose", rcvr)
}
for _, m := range methods {
regsvc.callbacks[formatName(m.method.Name)] = m
}
for _, s := range subscriptions {
regsvc.subscriptions[formatName(s.method.Name)] = s
}
return nil
}

svc.name = name
svc.callbacks, svc.subscriptions = methods, subscriptions

if len(svc.callbacks) == 0 && len(svc.subscriptions) == 0 {
return fmt.Errorf("Service %T doesn't have any suitable methods/subscriptions to expose", rcvr)
}

s.services[svc.name] = svc
return nil
}

// hasOption returns true if option is included in options, otherwise false
func hasOption(option CodecOption, options []CodecOption) bool {
for _, o := range options {
if option == o {
return true
}
}
return false
}

// serveRequest will reads requests from the codec, calls the RPC callback and
// writes the response to the given codec.
```

```
//
// If singleShot is true it will process a single request, otherwise it will handle
// requests until the codec returns an error when reading a request (in most cases
// an EOF). It executes requests in parallel when singleShot is false.
func (s *Server) serveRequest(codec ServerCodec, singleShot bool, options CodecOption) error {
var pend sync.WaitGroup

defer func() {
if err := recover(); err != nil {
const size = 64 << 10
buf := make([]byte, size)
buf = buf[:runtime.Stack(buf, false)]
log.Error(fmt.Sprint(string(buf)))
}
s.codecsMu.Lock()
s.codecs.Remove(codec)
s.codecsMu.Unlock()

return
}()

ctx, cancel := context.WithCancel(context.Background())
defer cancel()

// if the codec supports notification include a notifier that callbacks can use
// to send notification to clients. It is thight to the codec/connection. If the
// connection is closed the notifier will stop and cancels all active subscriptions.
if options&OptionSubscriptions == OptionSubscriptions {
ctx = context.WithValue(ctx, notifierKey{}, newNotifier(codec))
}
s.codecsMu.Lock()
if atomic.LoadInt32(&s.run) != 1 { // server stopped
s.codecsMu.Unlock()
return &shutdownError{}
}
s.codecs.Add(codec)
s.codecsMu.Unlock()

// test if the server is ordered to stop
for atomic.LoadInt32(&s.run) == 1 {
reqs, batch, err := s.readRequest(codec)
if err != nil {
```

```
// If a parsing error occurred, send an error
if err.Error() != "EOF" {
log.Debug(fmt.Sprintf("read error %v\n", err))
codec.Write(codec.CreateErrorResponse(nil, err))
}
// Error or end of stream, wait for requests and tear down
pend.Wait()
return nil
}

// check if server is ordered to shutdown and return an error
// telling the client that his request failed.
if atomic.LoadInt32(&s.run) != 1 {
err = &shutdownError{}
if batch {
resps := make([]interface{}, len(reqs))
for i, r := range reqs {
resps[i] = codec.CreateErrorResponse(&r.id, err)
}
codec.Write(resps)
} else {
codec.Write(codec.CreateErrorResponse(&reqs[0].id, err))
}
return nil
}
// If a single shot request is executing, run and return immediately
if singleShot {
if batch {
s.execBatch(ctx, codec, reqs)
} else {
s.exec(ctx, codec, reqs[0])
}
return nil
}
// For multi-shot connections, start a goroutine to serve and loop back
pend.Add(1)

go func(reqs []*serverRequest, batch bool) {
defer pend.Done()
if batch {
s.execBatch(ctx, codec, reqs)
} else {
```

```
s.exec(ctx, codec, reqs[0])
}
}(reqs, batch)
}
return nil
}

// ServeCodec reads incoming requests from codec, calls the appropriate callback and writes the
// response back using the given codec. It will block until the codec is closed or the server is
// stopped. In either case the codec is closed.
func (s *Server) ServeCodec(codec ServerCodec, options CodecOption) {
defer codec.Close()
s.serveRequest(codec, false, options)
}

// ServeSingleRequest reads and processes a single RPC request from the given codec. It will not
// close the codec unless a non-recoverable error has occurred. Note, this method will return after
// a single request has been processed!
func (s *Server) ServeSingleRequest(codec ServerCodec, options CodecOption) {
s.serveRequest(codec, true, options)
}

// Stop will stop reading new requests, wait for stopPendingRequestTimeout to allow pending
requests to finish,
// close all codecs which will cancel pending requests/subscriptions.
func (s *Server) Stop() {
if atomic.CompareAndSwapInt32(&s.run, 1, 0) {
log.Debug(fmt.Sprint("RPC Server shutdown initiatied"))
s.codecsMu.Lock()
defer s.codecsMu.Unlock()
s.codecs.Each(func(c interface{}) bool {
c.(ServerCodec).Close()
return true
})
}
}

// createSubscription will call the subscription callback and returns the subscription id or error.
func (s *Server) createSubscription(ctx context.Context, c ServerCodec, req *serverRequest) (ID,
error) {
// subscription have as first argument the context following optional arguments
args := []reflect.Value{req.callb.rcvr, reflect.ValueOf(ctx)}
args = append(args, req.args...)
reply := req.callb.method.Func.Call(args)

if !reply[1].IsNil() { // subscription creation failed
return "", reply[1].Interface().(error)
}

return reply[0].Interface().(*Subscription).ID, nil
}

// handle executes a request and returns the response from the callback.
func (s *Server) handle(ctx context.Context, codec ServerCodec, req *serverRequest) (interface{},
func()) {
if req.err != nil {
return codec.CreateErrorResponse(&req.id, req.err), nil
}

if req.isUnsubscribe { // cancel subscription, first param must be the subscription id
if len(req.args) >= 1 && req.args[0].Kind() == reflect.String {
notifier, supported := NotifierFromContext(ctx)
if !supported { // interface doesn't support subscriptions (e.g. http)
return codec.CreateErrorResponse(&req.id, &callbackError{ErrNotificationsUnsupported.Error()}),
nil
}

subid := ID(req.args[0].String())
if err := notifier.unsubscribe(subid); err != nil {
return codec.CreateErrorResponse(&req.id, &callbackError{err.Error()}), nil
}

return codec.CreateResponse(req.id, true), nil
}
return codec.CreateErrorResponse(&req.id, &invalidParamsError{"Expected subscription id as first
argument"}), nil
}

if req.callb.isSubscribe {
subid, err := s.createSubscription(ctx, codec, req)
if err != nil {
return codec.CreateErrorResponse(&req.id, &callbackError{err.Error()}), nil
}
// active the subscription after the sub id was successfully sent to the client
activateSub := func() {
notifier, _ := NotifierFromContext(ctx)
notifier.activate(subid, req.svcname)
}

return codec.CreateResponse(req.id, subid), activateSub
}

// regular RPC call, prepare arguments
if len(req.args) != len(req.callb.argTypes) {
rpcErr := &invalidParamsError{fmt.Sprintf("%s%s%s expects %d parameters, got %d",
req.svcname, serviceMethodSeparator, req.callb.method.Name,
len(req.callb.argTypes), len(req.args))}
return codec.CreateErrorResponse(&req.id, rpcErr), nil
}

arguments := []reflect.Value{req.callb.rcvr}
if req.callb.hasCtx {
arguments = append(arguments, reflect.ValueOf(ctx))
}
if len(req.args) > 0 {
arguments = append(arguments, req.args...)
}

// execute RPC method and return result
reply := req.callb.method.Func.Call(arguments)
if len(reply) == 0 {
return codec.CreateResponse(req.id, nil), nil
}

if req.callb.errPos >= 0 { // test if method returned an error
if !reply[req.callb.errPos].IsNil() {
e := reply[req.callb.errPos].Interface().(error)
res := codec.CreateErrorResponse(&req.id, &callbackError{e.Error()})
return res, nil
}
}
return codec.CreateResponse(req.id, reply[0].Interface()), nil
}

// exec executes the given request and writes the result back using the codec.
```

```
func (s *Server) exec(ctx context.Context, codec ServerCodec, req *serverRequest) {
var response interface{}
var callback func()
if req.err != nil {
response = codec.CreateErrorResponse(&req.id, req.err)
} else {
response, callback = s.handle(ctx, codec, req)
}

if err := codec.Write(response); err != nil {
log.Error(fmt.Sprintf("%v\n", err))
codec.Close()
}

// when request was a subscribe request this allows these subscriptions to be actived
if callback != nil {
callback()
}
}

// execBatch executes the given requests and writes the result back using the codec.
// It will only write the response back when the last request is processed.
func (s *Server) execBatch(ctx context.Context, codec ServerCodec, requests []*serverRequest) {
responses := make([]interface{}, len(requests))
var callbacks []func()
for i, req := range requests {
if req.err != nil {
responses[i] = codec.CreateErrorResponse(&req.id, req.err)
} else {
var callback func()
if responses[i], callback = s.handle(ctx, codec, req); callback != nil {
callbacks = append(callbacks, callback)
}
}
}

if err := codec.Write(responses); err != nil {
log.Error(fmt.Sprintf("%v\n", err))
codec.Close()
}

// when request holds one of more subscribe requests this allows these subscriptions to be
```

```
activated
for _, c := range callbacks {
c()
}
}

// readRequest requests the next (batch) request from the codec. It will return the collection
// of requests, an indication if the request was a batch, the invalid request identifier and an
// error when the request could not be read/parsed.
func (s *Server) readRequest(codec ServerCodec) ([]*serverRequest, bool, Error) {
reqs, batch, err := codec.ReadRequestHeaders()
if err != nil {
return nil, batch, err
}

requests := make([]*serverRequest, len(reqs))

// verify requests
for i, r := range reqs {
var ok bool
var svc *service

if r.err != nil {
requests[i] = &serverRequest{id: r.id, err: r.err}
continue
}

if r.isPubSub && strings.HasSuffix(r.method, unsubscribeMethodSuffix) {
requests[i] = &serverRequest{id: r.id, isUnsubscribe: true}
argTypes := []reflect.Type{reflect.TypeOf("")} // expect subscription id as first arg
if args, err := codec.ParseRequestArguments(argTypes, r.params); err == nil {
requests[i].args = args
} else {
requests[i].err = &invalidParamsError{err.Error()}
}
continue
}

if svc, ok = s.services[r.service]; !ok { // rpc method isn't available
requests[i] = &serverRequest{id: r.id, err: &methodNotFoundError{r.service, r.method}}
continue
}
if r.isPubSub { // eth_subscribe, r.method contains the subscription method name
if callb, ok := svc.subscriptions[r.method]; ok {
requests[i] = &serverRequest{id: r.id, svcname: svc.name, callb: callb}
if r.params != nil && len(callb.argTypes) > 0 {
argTypes := []reflect.Type{reflect.TypeOf("")}
argTypes = append(argTypes, callb.argTypes...)
if args, err := codec.ParseRequestArguments(argTypes, r.params); err == nil {
requests[i].args = args[1:] // first one is service.method name which isn't an actual argument
} else {
requests[i].err = &invalidParamsError{err.Error()}
}
}
} else {
requests[i] = &serverRequest{id: r.id, err: &methodNotFoundError{r.method, r.method}}
}
continue
}

if callb, ok := svc.callbacks[r.method]; ok { // lookup RPC method
requests[i] = &serverRequest{id: r.id, svcname: svc.name, callb: callb}
if r.params != nil && len(callb.argTypes) > 0 {
if args, err := codec.ParseRequestArguments(callb.argTypes, r.params); err == nil {
requests[i].args = args
} else {
requests[i].err = &invalidParamsError{err.Error()}
}
}
continue
}

requests[i] = &serverRequest{id: r.id, err: &methodNotFoundError{r.service, r.method}}
}

return requests, batch, nil
}
```

58:F:\git\coin\ethereum\go-ethereum\rpc\server_test.go

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

package rpc
```

```
import (
"context"
"encoding/json"
"net"
"reflect"
"testing"
"time"
)

type Service struct{}

type Args struct {
S string
}

func (s *Service) NoArgsRets() {
}

type Result struct {
String string
Int    int
Args   *Args
}

func (s *Service) Echo(str string, i int, args *Args) Result {
return Result{str, i, args}
}

func (s *Service) EchoWithCtx(ctx context.Context, str string, i int, args *Args) Result {
return Result{str, i, args}
}

func (s *Service) Sleep(ctx context.Context, duration time.Duration) {
select {
case <-time.After(duration):
case <-ctx.Done():
}
}

func (s *Service) Rets() (string, error) {
return "", nil
}
```

```
func (s *Service) InvalidRets1() (error, string) {
return nil, ""
}

func (s *Service) InvalidRets2() (string, string) {
return "", ""
}

func (s *Service) InvalidRets3() (string, string, error) {
return "", "", nil
}

func (s *Service) Subscription(ctx context.Context) (*Subscription, error) {
return nil, nil
}

func TestServerRegisterName(t *testing.T) {
server := NewServer()
service := new(Service)

if err := server.RegisterName("calc", service); err != nil {
t.Fatalf("%v", err)
}

if len(server.services) != 2 {
t.Fatalf("Expected 2 service entries, got %d", len(server.services))
}

svc, ok := server.services["calc"]
if !ok {
t.Fatalf("Expected service calc to be registered")
}

if len(svc.callbacks) != 5 {
t.Errorf("Expected 5 callbacks for service 'calc', got %d", len(svc.callbacks))
}

if len(svc.subscriptions) != 1 {
t.Errorf("Expected 1 subscription for service 'calc', got %d", len(svc.subscriptions))
}
}
```

```
func testServerMethodExecution(t *testing.T, method string) {
server := NewServer()
service := new(Service)

if err := server.RegisterName("test", service); err != nil {
t.Fatalf("%v", err)
}

stringArg := "string arg"
intArg := 1122
argsArg := &Args{"abcde"}
params := []interface{}{stringArg, intArg, argsArg}

request := map[string]interface{}{
"id":      12345,
"method":  "test_" + method,
"version": "2.0",
"params":  params,
}

clientConn, serverConn := net.Pipe()
defer clientConn.Close()

go server.ServeCodec(NewJSONCodec(serverConn), OptionMethodInvocation)

out := json.NewEncoder(clientConn)
in := json.NewDecoder(clientConn)

if err := out.Encode(request); err != nil {
t.Fatal(err)
}

response := jsonSuccessResponse{Result: &Result{}}
if err := in.Decode(&response); err != nil {
t.Fatal(err)
}

if result, ok := response.Result.(*Result); ok {
if result.String != stringArg {
t.Errorf("expected %s, got : %s\n", stringArg, result.String)
}
if result.Int != intArg {
t.Errorf("expected %d, got %d\n", intArg, result.Int)
}
if !reflect.DeepEqual(result.Args, argsArg) {
t.Errorf("expected %v, got %v\n", argsArg, result)
}
} else {
t.Fatalf("invalid response: expected *Result - got: %T", response.Result)
}
}

func TestServerMethodExecution(t *testing.T) {
testServerMethodExecution(t, "echo")
}

func TestServerMethodWithCtx(t *testing.T) {
testServerMethodExecution(t, "echoWithCtx")
}
```

59:F:\git\coin\ethereum\go-ethereum\rpc\subscription.go

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

package rpc

import (
"context"
"errors"
"sync"
)

var (
// ErrNotificationsUnsupported is returned when the connection doesn't support notifications
ErrNotificationsUnsupported = errors.New("notifications not supported")
// ErrNotificationNotFound is returned when the notification for the given id is not found
ErrSubscriptionNotFound = errors.New("subscription not found")
)

// ID defines a pseudo random number that is used to identify RPC subscriptions.
type ID string

// a Subscription is created by a notifier and tight to that notifier. The client can use
// this subscription to wait for an unsubscribe request for the client, see Err().
```

```
type Subscription struct {
ID        ID
namespace string
err       chan error // closed on unsubscribe
}

// Err returns a channel that is closed when the client send an unsubscribe request.
func (s *Subscription) Err() <-chan error {
return s.err
}

// notifierKey is used to store a notifier within the connection context.
type notifierKey struct{}

// Notifier is tight to a RPC connection that supports subscriptions.
// Server callbacks use the notifier to send notifications.
type Notifier struct {
codec    ServerCodec
subMu    sync.RWMutex // guards active and inactive maps
stopped  bool
active   map[ID]*Subscription
inactive map[ID]*Subscription
}

// newNotifier creates a new notifier that can be used to send subscription
// notifications to the client.
func newNotifier(codec ServerCodec) *Notifier {
return &Notifier{
codec:    codec,
active:   make(map[ID]*Subscription),
inactive: make(map[ID]*Subscription),
}
}

// NotifierFromContext returns the Notifier value stored in ctx, if any.
func NotifierFromContext(ctx context.Context) (*Notifier, bool) {
n, ok := ctx.Value(notifierKey{}).(*Notifier)
return n, ok
}

// CreateSubscription returns a new subscription that is coupled to the
// RPC connection. By default subscriptions are inactive and notifications
// are dropped until the subscription is marked as active. This is done
// by the RPC server after the subscription ID is send to the client.
func (n *Notifier) CreateSubscription() *Subscription {
s := &Subscription{ID: NewID(), err: make(chan error)}
n.subMu.Lock()
n.inactive[s.ID] = s
n.subMu.Unlock()
return s
}

// Notify sends a notification to the client with the given data as payload.
// If an error occurs the RPC connection is closed and the error is returned.
func (n *Notifier) Notify(id ID, data interface{}) error {
n.subMu.RLock()
defer n.subMu.RUnlock()

sub, active := n.active[id]
if active {
notification := n.codec.CreateNotification(string(id), sub.namespace, data)
if err := n.codec.Write(notification); err != nil {
n.codec.Close()
return err
}
}
return nil
}

// Closed returns a channel that is closed when the RPC connection is closed.
func (n *Notifier) Closed() <-chan interface{} {
return n.codec.Closed()
}

// unsubscribe a subscription.
// If the subscription could not be found ErrSubscriptionNotFound is returned.
func (n *Notifier) unsubscribe(id ID) error {
n.subMu.Lock()
defer n.subMu.Unlock()
if s, found := n.active[id]; found {
close(s.err)
delete(n.active, id)
return nil
}
```

```
return ErrSubscriptionNotFound
}

// activate enables a subscription. Until a subscription is enabled all
// notifications are dropped. This method is called by the RPC server after
// the subscription ID was sent to client. This prevents notifications being
// send to the client before the subscription ID is send to the client.
func (n *Notifier) activate(id ID, namespace string) {
n.subMu.Lock()
defer n.subMu.Unlock()
if sub, found := n.inactive[id]; found {
sub.namespace = namespace
n.active[id] = sub
delete(n.inactive, id)
}
}

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

package rpc

import (
"context"
"encoding/json"
"fmt"
"net"
"sync"
"testing"
"time"
)

type NotificationTestService struct {
mu          sync.Mutex
unsubscribed bool

gotHangSubscriptionReq  chan struct{}
unblockHangSubscription chan struct{}
}

func (s *NotificationTestService) Echo(i int) int {
return i
```

```
}

func (s *NotificationTestService) wasUnsubCallbackCalled() bool {
s.mu.Lock()
defer s.mu.Unlock()
return s.unsubscribed
}

func (s *NotificationTestService) Unsubscribe(subid string) {
s.mu.Lock()
s.unsubscribed = true
s.mu.Unlock()
}

func (s *NotificationTestService) SomeSubscription(ctx context.Context, n, val int) (*Subscription,
error) {
notifier, supported := NotifierFromContext(ctx)
if !supported {
return nil, ErrNotificationsUnsupported
}

// by explicitly creating an subscription we make sure that the subscription id is send back to the
client
// before the first subscription.Notify is called. Otherwise the events might be send before the
response
// for the eth_subscribe method.
subscription := notifier.CreateSubscription()

go func() {
// test expects n events, if we begin sending event immediately some events
// will probably be dropped since the subscription ID might not be send to
// the client.
time.Sleep(5 * time.Second)
for i := 0; i < n; i++ {
if err := notifier.Notify(subscription.ID, val+i); err != nil {
return
}
}

select {
case <-notifier.Closed():
s.mu.Lock()
s.unsubscribed = true
s.mu.Unlock()
case <-subscription.Err():
s.mu.Lock()
s.unsubscribed = true
s.mu.Unlock()
}
}()

return subscription, nil
}

// HangSubscription blocks on s.unblockHangSubscription before
// sending anything.
func (s *NotificationTestService) HangSubscription(ctx context.Context, val int) (*Subscription,
error) {
notifier, supported := NotifierFromContext(ctx)
if !supported {
return nil, ErrNotificationsUnsupported
}

s.gotHangSubscriptionReq <- struct{}{}
<-s.unblockHangSubscription
subscription := notifier.CreateSubscription()

go func() {
notifier.Notify(subscription.ID, val)
}()
return subscription, nil
}

func TestNotifications(t *testing.T) {
server := NewServer()
service := &NotificationTestService{}

if err := server.RegisterName("eth", service); err != nil {
t.Fatalf("unable to register test service %v", err)
}

clientConn, serverConn := net.Pipe()

go server.ServeCodec(NewJSONCodec(serverConn),
OptionMethodInvocation|OptionSubscriptions)

out := json.NewEncoder(clientConn)
in := json.NewDecoder(clientConn)

n := 5
val := 12345
request := map[string]interface{}{
"id":      1,
"method":  "eth_subscribe",
"version": "2.0",
"params":  []interface{}{"someSubscription", n, val},
}

// create subscription
if err := out.Encode(request); err != nil {
t.Fatal(err)
}

var subid string
response := jsonSuccessResponse{Result: subid}
if err := in.Decode(&response); err != nil {
t.Fatal(err)
}

var ok bool
if _, ok = response.Result.(string); !ok {
t.Fatalf("expected subscription id, got %T", response.Result)
}

for i := 0; i < n; i++ {
var notification jsonNotification
if err := in.Decode(&notification); err != nil {
t.Fatalf("%v", err)
}

if int(notification.Params.Result.(float64)) != val+i {
t.Fatalf("expected %d, got %d", val+i, notification.Params.Result)
}
}

clientConn.Close() // causes notification unsubscribe callback to be called
```

```
time.Sleep(1 * time.Second)

if !service.wasUnsubCallbackCalled() {
t.Error("unsubscribe callback not called after closing connection")
}
}

func waitForMessages(t *testing.T, in *json.Decoder, successes chan<- jsonSuccessResponse,
failures chan<- jsonErrResponse, notifications chan<- jsonNotification) {

// read and parse server messages
for {
var rmsg json.RawMessage
if err := in.Decode(&rmsg); err != nil {
return
}

var responses []map[string]interface{}
if rmsg[0] == '[' {
if err := json.Unmarshal(rmsg, &responses); err != nil {
t.Fatalf("Received invalid message: %s", rmsg)
}
} else {
var msg map[string]interface{}
if err := json.Unmarshal(rmsg, &msg); err != nil {
t.Fatalf("Received invalid message: %s", rmsg)
}
responses = append(responses, msg)
}

for _, msg := range responses {
// determine what kind of msg was received and broadcast
// it to over the corresponding channel
if _, found := msg["result"]; found {
successes <- jsonSuccessResponse{
Version: msg["jsonrpc"].(string),
Id:      msg["id"],
Result:  msg["result"],
}
continue
}
if _, found := msg["error"]; found {
```

```
params := msg["params"].(map[string]interface{})
failures <- jsonErrResponse{
Version: msg["jsonrpc"].(string),
Id:      msg["id"],
Error:   jsonError{int(params["subscription"].(float64)), params["message"].(string),
params["data"]},
}
continue
}
if _, found := msg["params"]; found {
params := msg["params"].(map[string]interface{})
notifications <- jsonNotification{
Version: msg["jsonrpc"].(string),
Method:  msg["method"].(string),
Params:  jsonSubscription{params["subscription"].(string), params["result"]},
}
continue
}
t.Fatalf("Received invalid message: %s", msg)
}
}
}

// TestSubscriptionMultipleNamespaces ensures that subscriptions can exists
// for multiple different namespaces.
func TestSubscriptionMultipleNamespaces(t *testing.T) {
var (
namespaces             = []string{"eth", "shh", "bzz"}
server                 = NewServer()
service                = NotificationTestService{}
clientConn, serverConn = net.Pipe()

out           = json.NewEncoder(clientConn)
in            = json.NewDecoder(clientConn)
successes     = make(chan jsonSuccessResponse)
failures      = make(chan jsonErrResponse)
notifications = make(chan jsonNotification)
)

// setup and start server
for _, namespace := range namespaces {
if err := server.RegisterName(namespace, &service); err != nil {
```

```
t.Fatalf("unable to register test service %v", err)
}
}

go server.ServeCodec(NewJSONCodec(serverConn),
OptionMethodInvocation|OptionSubscriptions)
defer server.Stop()

// wait for message and write them to the given channels
go waitForMessages(t, in, successes, failures, notifications)

// create subscriptions one by one
n := 3
for i, namespace := range namespaces {
request := map[string]interface{}{
"id":      i,
"method":  fmt.Sprintf("%s_subscribe", namespace),
"version": "2.0",
"params":  []interface{}{"someSubscription", n, i},
}

if err := out.Encode(&request); err != nil {
t.Fatalf("Could not create subscription: %v", err)
}
}

// create all subscriptions in 1 batch
var requests []interface{}
for i, namespace := range namespaces {
requests = append(requests, map[string]interface{}{
"id":      i,
"method":  fmt.Sprintf("%s_subscribe", namespace),
"version": "2.0",
"params":  []interface{}{"someSubscription", n, i},
})
}

if err := out.Encode(&requests); err != nil {
t.Fatalf("Could not create subscription in batch form: %v", err)
}

timeout := time.After(30 * time.Second)
```

```
subids := make(map[string]string, 2*len(namespaces))
count := make(map[string]int, 2*len(namespaces))

for {
done := true
for id, _ := range count {
if count, found := count[id]; !found || count < (2*n) {
done = false
}
}

if done && len(count) == len(namespaces) {
break
}

select {
case suc := <-successes: // subscription created
subids[namespaces[int(suc.Id.(float64))]] = suc.Result.(string)
case failure := <-failures:
t.Errorf("received error: %v", failure.Error)
case notification := <-notifications:
if cnt, found := count[notification.Params.Subscription]; found {
count[notification.Params.Subscription] = cnt + 1
} else {
count[notification.Params.Subscription] = 1
}
case <-timeout:
for _, namespace := range namespaces {
subid, found := subids[namespace]
if !found {
t.Errorf("Subscription for '%s' not created", namespace)
continue
}
if count, found := count[subid]; !found || count < n {
t.Errorf("Didn't receive all notifications (%d<%d) in time for namespace '%s'", count, n,
namespace)
}
}
return
}
}
}
```

61:F:\git\coin\ethereum\go-ethereum\rpc\types.go

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

package rpc

import (
"fmt"
"math"
"reflect"
"strings"
"sync"

"github.com/ethereum/go-ethereum/common/hexutil"
"gopkg.in/fatih/set.v0"
)

// API describes the set of methods offered over the RPC interface
type API struct {
Namespace string      // namespace under which the rpc methods of Service are exposed
Version   string      // api version for DApp's
Service   interface{} // receiver instance which holds the methods
Public    bool        // indication if the methods must be considered safe for public use
}

// callback is a method callback which was registered in the server
type callback struct {
rcvr        reflect.Value  // receiver of method
method      reflect.Method // callback
argTypes    []reflect.Type // input argument types
hasCtx      bool           // method's first argument is a context (not included in argTypes)
errPos      int            // err return idx, of -1 when method cannot return error
isSubscribe bool           // indication if the callback is a subscription
}

// service represents a registered object
type service struct {
name          string        // name for service
rcvr          reflect.Value // receiver of methods for the service
typ           reflect.Type  // receiver type
callbacks     callbacks     // registered handlers
subscriptions subscriptions // available subscriptions/notifications
```

```
}

// serverRequest is an incoming request
type serverRequest struct {
id            interface{}
svcname       string
rcvr          reflect.Value
callb         *callback
args          []reflect.Value
isUnsubscribe bool
err           Error
}

type serviceRegistry map[string]*service       // collection of services
type callbacks map[string]*callback            // collection of RPC callbacks
type subscriptions map[string]*callback        // collection of subscription callbacks
type subscriptionRegistry map[string]*callback // collection of subscription callbacks

// Server represents a RPC server
type Server struct {
services       serviceRegistry
muSubcriptions sync.Mutex // protects subscriptions
subscriptions  subscriptionRegistry

run      int32
codecsMu sync.Mutex
codecs   *set.Set
}

// rpcRequest represents a raw incoming RPC request
type rpcRequest struct {
service  string
method   string
id       interface{}
isPubSub bool
params   interface{}
err      Error // invalid batch element
}

// Error wraps RPC errors, which contain an error code in addition to the message.
type Error interface {
Error() string  // returns the message
```

```
ErrorCode() int // returns the code
}

// ServerCodec implements reading, parsing and writing RPC messages for the server side of
// a RPC session. Implementations must be go-routine safe since the codec can be called in
// multiple go-routines concurrently.
type ServerCodec interface {
// Read next request
ReadRequestHeaders() ([]rpcRequest, bool, Error)
// Parse request argument to the given types
ParseRequestArguments(argTypes []reflect.Type, params interface{}) ([]reflect.Value, Error)
// Assemble success response, expects response id and payload
CreateResponse(id interface{}, reply interface{}) interface{}
// Assemble error response, expects response id and error
CreateErrorResponse(id interface{}, err Error) interface{}
// Assemble error response with extra information about the error through info
CreateErrorResponseWithInfo(id interface{}, err Error, info interface{}) interface{}
// Create notification response
CreateNotification(id, namespace string, event interface{}) interface{}
// Write msg to client.
Write(msg interface{}) error
// Close underlying data stream
Close()
// Closed when underlying connection is closed
Closed() <-chan interface{}
}

type BlockNumber int64

const (
PendingBlockNumber  = BlockNumber(-2)
LatestBlockNumber   = BlockNumber(-1)
EarliestBlockNumber = BlockNumber(0)
)

// UnmarshalJSON parses the given JSON fragment into a BlockNumber. It supports:
// - "latest", "earliest" or "pending" as string arguments
// - the block number
// Returned errors:
// - an invalid block number error when the given argument isn't a known strings
// - an out of range error when the given block number is either too little or too large
func (bn *BlockNumber) UnmarshalJSON(data []byte) error {
```

```
input := strings.TrimSpace(string(data))
if len(input) >= 2 && input[0] == "" && input[len(input)-1] == "" {
input = input[1 : len(input)-1]
}

switch input {
case "earliest":
*bn = EarliestBlockNumber
return nil
case "latest":
*bn = LatestBlockNumber
return nil
case "pending":
*bn = PendingBlockNumber
return nil
}

blckNum, err := hexutil.DecodeUint64(input)
if err != nil {
return err
}
if blckNum > math.MaxInt64 {
return fmt.Errorf("Blocknumber too high")
}

*bn = BlockNumber(blckNum)
return nil
}

func (bn BlockNumber) Int64() int64 {
return (int64)(bn)
}
```

62:F:\git\coin\ethereum\go-ethereum\rpc\types_test.go

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

package rpc

import (
"encoding/json"
"testing"
```

```
"github.com/ethereum/go-ethereum/common/math"
)

func TestBlockNumberJSONUnmarshal(t *testing.T) {
tests := []struct {
input    string
mustFail bool
expected BlockNumber
}{
0:  {`"0x"`, true, BlockNumber(0)},
1:  {`"0x0"`, false, BlockNumber(0)},
2:  {`"0X1"`, false, BlockNumber(1)},
3:  {`"0x00"`, true, BlockNumber(0)},
4:  {`"0x01"`, true, BlockNumber(0)},
5:  {`"0x1"`, false, BlockNumber(1)},
6:  {`"0x12"`, false, BlockNumber(18)},
7:  {`"0x7fffffffffffffff"`, false, BlockNumber(math.MaxInt64)},
8:  {`"0x8000000000000000"`, true, BlockNumber(0)},
9:  {"0", true, BlockNumber(0)},
10: {`"ff"`, true, BlockNumber(0)},
11: {`"pending"`, false, PendingBlockNumber},
12: {`"latest"`, false, LatestBlockNumber},
13: {`"earliest"`, false, EarliestBlockNumber},
14: {`someString`, true, BlockNumber(0)},
15: {`""`, true, BlockNumber(0)},
16: {``, true, BlockNumber(0)},
}

for i, test := range tests {
var num BlockNumber
err := json.Unmarshal([]byte(test.input), &num)
if test.mustFail && err == nil {
t.Errorf("Test %d should fail", i)
continue
}
if !test.mustFail && err != nil {
t.Errorf("Test %d should pass but got err: %v", i, err)
continue
}
if num != test.expected {
t.Errorf("Test %d got unexpected value, want %d, got %d", i, test.expected, num)
}
}
}

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

package rpc

import (
"bufio"
"context"
crand "crypto/rand"
"encoding/binary"
"encoding/hex"
"math/big"
"math/rand"
"reflect"
"strings"
"sync"
"time"
"unicode"
"unicode/utf8"
)

var (
subscriptionIDGenMu sync.Mutex
subscriptionIDGen   = idGenerator()
)

// Is this an exported - upper case - name?
func isExported(name string) bool {
rune, _ := utf8.DecodeRuneInString(name)
return unicode.IsUpper(rune)
}

// Is this type exported or a builtin?
func isExportedOrBuiltinType(t reflect.Type) bool {
for t.Kind() == reflect.Ptr {
t = t.Elem()
}
// PkgPath will be non-empty even for an exported type,
// so we need to check the type name as well.
```

```
return isExported(t.Name()) || t.PkgPath() == ""
}

var contextType = reflect.TypeOf((*context.Context)(nil)).Elem()

// isContextType returns an indication if the given t is of context.Context or *context.Context type
func isContextType(t reflect.Type) bool {
for t.Kind() == reflect.Ptr {
t = t.Elem()
}
return t == contextType
}

var errorType = reflect.TypeOf((*error)(nil)).Elem()

// Implements this type the error interface
func isErrorType(t reflect.Type) bool {
for t.Kind() == reflect.Ptr {
t = t.Elem()
}
return t.Implements(errorType)
}

var subscriptionType = reflect.TypeOf((*Subscription)(nil)).Elem()

// isSubscriptionType returns an indication if the given t is of Subscription or *Subscription type
func isSubscriptionType(t reflect.Type) bool {
for t.Kind() == reflect.Ptr {
t = t.Elem()
}
return t == subscriptionType
}

// isPubSub tests whether the given method has as as first argument a context.Context
// and returns the pair (Subscription, error)
func isPubSub(methodType reflect.Type) bool {
// numIn(0) is the receiver type
if methodType.NumIn() < 2 || methodType.NumOut() != 2 {
return false
}

return isContextType(methodType.In(1)) &&
isSubscriptionType(methodType.Out(0)) &&
isErrorType(methodType.Out(1))
}

// formatName will convert to first character to lower case
func formatName(name string) string {
ret := []rune(name)
if len(ret) > 0 {
ret[0] = unicode.ToLower(ret[0])
}
return string(ret)
}

var bigIntType = reflect.TypeOf((*big.Int)(nil)).Elem()

// Indication if this type should be serialized in hex
func isHexNum(t reflect.Type) bool {
if t == nil {
return false
}
for t.Kind() == reflect.Ptr {
t = t.Elem()
}

return t == bigIntType
}

var blockNumberType = reflect.TypeOf((*BlockNumber)(nil)).Elem()

// Indication if the given block is a BlockNumber
func isBlockNumber(t reflect.Type) bool {
if t == nil {
return false
}

for t.Kind() == reflect.Ptr {
t = t.Elem()
}

return t == blockNumberType
}
```

```
// suitableCallbacks iterates over the methods of the given type. It will determine if a method
satisfies the criteria
// for a RPC callback or a subscription callback and adds it to the collection of callbacks or
subscriptions. See server
// documentation for a summary of these criteria.
func suitableCallbacks(rcvr reflect.Value, typ reflect.Type) (callbacks, subscriptions) {
callbacks := make(callbacks)
subscriptions := make(subscriptions)

METHODS:
for m := 0; m < typ.NumMethod(); m++ {
method := typ.Method(m)
mtype := method.Type
mname := formatName(method.Name)
if method.PkgPath != "" { // method must be exported
continue
}

var h callback
h.isSubscribe = isPubSub(mtype)
h.rcvr = rcvr
h.method = method
h.errPos = -1

firstArg := 1
numIn := mtype.NumIn()
if numIn >= 2 && mtype.In(1) == contextType {
h.hasCtx = true
firstArg = 2
}

if h.isSubscribe {
h.argTypes = make([]reflect.Type, numIn-firstArg) // skip rcvr type
for i := firstArg; i < numIn; i++ {
argType := mtype.In(i)
if isExportedOrBuiltinType(argType) {
h.argTypes[i-firstArg] = argType
} else {
continue METHODS
}
}
```

```
subscriptions[mname] = &h
continue METHODS
}

// determine method arguments, ignore first arg since it's the receiver type
// Arguments must be exported or builtin types
h.argTypes = make([]reflect.Type, numIn-firstArg)
for i := firstArg; i < numIn; i++ {
argType := mtype.In(i)
if !isExportedOrBuiltinType(argType) {
continue METHODS
}
h.argTypes[i-firstArg] = argType
}

// check that all returned values are exported or builtin types
for i := 0; i < mtype.NumOut(); i++ {
if !isExportedOrBuiltinType(mtype.Out(i)) {
continue METHODS
}
}

// when a method returns an error it must be the last returned value
h.errPos = -1
for i := 0; i < mtype.NumOut(); i++ {
if isErrorType(mtype.Out(i)) {
h.errPos = i
break
}
}

if h.errPos >= 0 && h.errPos != mtype.NumOut()-1 {
continue METHODS
}

switch mtype.NumOut() {
case 0, 1:
break
case 2:
if h.errPos == -1 { // method must one return value and 1 error
continue METHODS
}
```

```
break
default:
continue METHODS
}

callbacks[mname] = &h
}

return callbacks, subscriptions
}

// idGenerator helper utility that generates a (pseudo) random sequence of
// bytes that are used to generate identifiers.
func idGenerator() *rand.Rand {
if seed, err := binary.ReadVarint(bufio.NewReader(crand.Reader)); err == nil {
return rand.New(rand.NewSource(seed))
}
return rand.New(rand.NewSource(int64(time.Now().Nanosecond())))
}

// NewID generates a identifier that can be used as an identifier in the RPC interface.
// e.g. filter and subscription identifier.
func NewID() ID {
subscriptionIDGenMu.Lock()
defer subscriptionIDGenMu.Unlock()

id := make([]byte, 16)
for i := 0; i < len(id); i += 7 {
val := subscriptionIDGen.Int63()
for j := 0; i+j < len(id) && j < 7; j++ {
id[i+j] = byte(val)
val >>= 8
}
}

rpcId := hex.EncodeToString(id)
// rpc ID's are RPC quantities, no leading zero's and 0 is 0x0
rpcId = strings.TrimLeft(rpcId, "0")
if rpcId == "" {
rpcId = "0"
}
```

```
return ID("0x" + rpcId)
}

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

package rpc

import (
"strings"
"testing"
)

func TestNewID(t *testing.T) {
hexchars := "0123456789ABCDEFabcdef"
for i := 0; i < 100; i++ {
id := string(NewID())
if !strings.HasPrefix(id, "0x") {
t.Fatalf("invalid ID prefix, want '0x...', got %s", id)
}

id = id[2:]
if len(id) == 0 || len(id) > 32 {
t.Fatalf("invalid ID length, want len(id) > 0 && len(id) <= 32), got %d", len(id))
}

for i := 0; i < len(id); i++ {
if strings.IndexByte(hexchars, id[i]) == -1 {
t.Fatalf("unexpected byte, want any valid hex char, got %c", id[i])
}
}
}
}

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

package rpc

import (
"context"
"crypto/tls"
```

```
"fmt"
"net"
"net/http"
"net/url"
"os"
"strings"
"time"

"github.com/ethereum/go-ethereum/log"
"golang.org/x/net/websocket"
"gopkg.in/fatih/set.v0"
)

// WebsocketHandler returns a handler that serves JSON-RPC to WebSocket connections.
//
// allowedOrigins should be a comma-separated list of allowed origin URLs.
// To allow connections with any origin, pass "*".
func (srv *Server) WebsocketHandler(allowedOrigins []string) http.Handler {
return websocket.Server{
Handshake: wsHandshakeValidator(allowedOrigins),
Handler: func(conn *websocket.Conn) {
srv.ServeCodec(NewJSONCodec(conn), OptionMethodInvocation|OptionSubscriptions)
},
}
}

// NewWSServer creates a new websocket RPC server around an API provider.
//
// Deprecated: use Server.WebsocketHandler
func NewWSServer(allowedOrigins []string, srv *Server) *http.Server {
return &http.Server{Handler: srv.WebsocketHandler(allowedOrigins)}
}

// wsHandshakeValidator returns a handler that verifies the origin during the
// websocket upgrade process. When a '*' is specified as an allowed origins all
// connections are accepted.
func wsHandshakeValidator(allowedOrigins []string) func(*websocket.Config, *http.Request) error
{
origins := set.New()
allowAllOrigins := false

for _, origin := range allowedOrigins {
```

```
if origin == "*" {
allowAllOrigins = true
}
if origin != "" {
origins.Add(strings.ToLower(origin))
}
}

// allow localhost if no allowedOrigins are specified.
if len(origins.List()) == 0 {
origins.Add("http://localhost")
if hostname, err := os.Hostname(); err == nil {
origins.Add("http://" + strings.ToLower(hostname))
}
}

log.Debug(fmt.Sprintf("Allowed origin(s) for WS RPC interface %v\n", origins.List()))

f := func(cfg *websocket.Config, req *http.Request) error {
origin := strings.ToLower(req.Header.Get("Origin"))
if allowAllOrigins || origins.Has(origin) {
return nil
}
log.Debug(fmt.Sprintf("origin '%s' not allowed on WS-RPC interface\n", origin))
return fmt.Errorf("origin %s not allowed", origin)
}

return f
}

// DialWebsocket creates a new RPC client that communicates with a JSON-RPC server
// that is listening on the given endpoint.
//
// The context is used for the initial connection establishment. It does not
// affect subsequent interactions with the client.
func DialWebsocket(ctx context.Context, endpoint, origin string) (*Client, error) {
if origin == "" {
var err error
if origin, err = os.Hostname(); err != nil {
return nil, err
}
if strings.HasPrefix(endpoint, "wss") {
```

```
origin = "https://" + strings.ToLower(origin)
} else {
origin = "http://" + strings.ToLower(origin)
}
}
config, err := websocket.NewConfig(endpoint, origin)
if err != nil {
return nil, err
}

return newClient(ctx, func(ctx context.Context) (net.Conn, error) {
return wsDialContext(ctx, config)
})
}

func wsDialContext(ctx context.Context, config *websocket.Config) (*websocket.Conn, error) {
var conn net.Conn
var err error
switch config.Location.Scheme {
case "ws":
conn, err = dialContext(ctx, "tcp", wsDialAddress(config.Location))
case "wss":
dialer := contextDialer(ctx)
conn, err = tls.DialWithDialer(dialer, "tcp", wsDialAddress(config.Location), config.TlsConfig)
default:
err = websocket.ErrBadScheme
}
if err != nil {
return nil, err
}
ws, err := websocket.NewClient(config, conn)
if err != nil {
conn.Close()
return nil, err
}
return ws, err
}

var wsPortMap = map[string]string{"ws": "80", "wss": "443"}

func wsDialAddress(location *url.URL) string {
if _, ok := wsPortMap[location.Scheme]; ok {
```

```
if _, _, err := net.SplitHostPort(location.Host); err != nil {
return net.JoinHostPort(location.Host, wsPortMap[location.Scheme])
}
}
return location.Host
}

func dialContext(ctx context.Context, network, addr string) (net.Conn, error) {
d := &net.Dialer{KeepAlive: tcpKeepAliveInterval}
return d.DialContext(ctx, network, addr)
}

func contextDialer(ctx context.Context) *net.Dialer {
dialer := &net.Dialer{Cancel: ctx.Done(), KeepAlive: tcpKeepAliveInterval}
if deadline, ok := ctx.Deadline(); ok {
dialer.Deadline = deadline
} else {
dialer.Deadline = time.Now().Add(defaultDialTimeout)
}
return dialer
}

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

package tests

import (
"math/big"
"path/filepath"
"testing"
)

func TestBcValidBlockTests(t *testing.T) {
err := RunBlockTest(big.NewInt(1000000), nil, nil, filepath.Join(blockTestDir,
"bcValidBlockTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestBcUncleHeaderValidityTests(t *testing.T) {
```

```
err := RunBlockTest(big.NewInt(1000000), nil, nil, filepath.Join(blockTestDir,
"bcUncleHeaderValiditiy.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestBcUncleTests(t *testing.T) {
err := RunBlockTest(big.NewInt(1000000), nil, nil, filepath.Join(blockTestDir, "bcUncleTest.json"),
BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestBcForkUncleTests(t *testing.T) {
err := RunBlockTest(big.NewInt(1000000), nil, nil, filepath.Join(blockTestDir, "bcForkUncle.json"),
BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestBcInvalidHeaderTests(t *testing.T) {
err := RunBlockTest(big.NewInt(1000000), nil, nil, filepath.Join(blockTestDir,
"bcInvalidHeaderTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestBcInvalidRLPTests(t *testing.T) {
err := RunBlockTest(big.NewInt(1000000), nil, nil, filepath.Join(blockTestDir,
"bcInvalidRLPTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestBcRPCAPITests(t *testing.T) {
err := RunBlockTest(big.NewInt(1000000), nil, nil, filepath.Join(blockTestDir,
"bcRPC_API_Test.json"), BlockSkipTests)
```

```
if err != nil {
t.Fatal(err)
}
}

func TestBcForkBlockTests(t *testing.T) {
err := RunBlockTest(big.NewInt(1000000), nil, nil, filepath.Join(blockTestDir,
"bcForkBlockTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestBcForkStress(t *testing.T) {
err := RunBlockTest(big.NewInt(1000000), nil, nil, filepath.Join(blockTestDir,
"bcForkStressTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestBcTotalDifficulty(t *testing.T) {
// skip because these will fail due to selfish mining fix
t.Skip()

err := RunBlockTest(big.NewInt(1000000), nil, nil, filepath.Join(blockTestDir,
"bcTotalDifficultyTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestBcWallet(t *testing.T) {
err := RunBlockTest(big.NewInt(1000000), nil, nil, filepath.Join(blockTestDir, "bcWalletTest.json"),
BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestBcGasPricer(t *testing.T) {
err := RunBlockTest(big.NewInt(1000000), nil, nil, filepath.Join(blockTestDir,
"bcGasPricerTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

// TODO: iterate over files once we got more than a few
func TestBcRandom(t *testing.T) {
err := RunBlockTest(big.NewInt(1000000), nil, big.NewInt(10), filepath.Join(blockTestDir,
"RandomTests/bl201507071825GO.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestBcMultiChain(t *testing.T) {
// skip due to selfish mining
t.Skip()

err := RunBlockTest(big.NewInt(1000000), nil, big.NewInt(10), filepath.Join(blockTestDir,
"bcMultiChainTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestBcState(t *testing.T) {
err := RunBlockTest(big.NewInt(1000000), nil, big.NewInt(10), filepath.Join(blockTestDir,
"bcStateTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

// Homestead tests
func TestHomesteadBcValidBlockTests(t *testing.T) {
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcValidBlockTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}
```

```
func TestHomesteadBcUncleHeaderValidityTests(t *testing.T) {
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcUncleHeaderValiditiy.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestHomesteadBcUncleTests(t *testing.T) {
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcUncleTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestHomesteadBcInvalidHeaderTests(t *testing.T) {
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcInvalidHeaderTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestHomesteadBcRPCAPITests(t *testing.T) {
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcRPC_API_Test.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestHomesteadBcForkStress(t *testing.T) {
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcForkStressTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestHomesteadBcTotalDifficulty(t *testing.T) {
```

```
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcTotalDifficultyTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestHomesteadBcWallet(t *testing.T) {
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcWalletTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestHomesteadBcGasPricer(t *testing.T) {
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcGasPricerTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestHomesteadBcMultiChain(t *testing.T) {
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcMultiChainTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestHomesteadBcState(t *testing.T) {
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcStateTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

// DAO hard-fork tests
func TestDAOBcTheDao(t *testing.T) {
err := RunBlockTest(big.NewInt(5), big.NewInt(8), nil, filepath.Join(blockTestDir, "TestNetwork",
"bcTheDaoTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestEIP150Bc(t *testing.T) {
err := RunBlockTest(big.NewInt(0), big.NewInt(8), big.NewInt(10), filepath.Join(blockTestDir,
"TestNetwork", "bcEIP150Test.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestHomesteadBcExploit(t *testing.T) {
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcExploitTest.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}
func TestHomesteadBcShanghaiLove(t *testing.T) {
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcShanghaiLove.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}
func TestHomesteadBcSuicideIssue(t *testing.T) {
err := RunBlockTest(big.NewInt(0), nil, nil, filepath.Join(blockTestDir, "Homestead",
"bcSuicideIssue.json"), BlockSkipTests)
if err != nil {
t.Fatal(err)
}
}
```

67:F:\git\coin\ethereum\go-ethereum\tests\block_test_util.go

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

package tests

import (
"bytes"
"encoding/hex"
"fmt"
"io"
"math/big"
"runtime"
"strconv"
"strings"

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

// Block Test JSON Format
type BlockTest struct {
Genesis *types.Block

Json          *btJSON
preAccounts   map[string]btAccount
postAccounts  map[string]btAccount
lastblockhash string
}

type btJSON struct {
Blocks             []btBlock
GenesisBlockHeader btHeader
Pre                map[string]btAccount
PostState          map[string]btAccount
Lastblockhash      string
}

type btBlock struct {
BlockHeader  *btHeader
```

```
Rlp          string
Transactions []btTransaction
UncleHeaders []*btHeader
}

type btAccount struct {
Balance    string
Code       string
Nonce      string
Storage    map[string]string
PrivateKey string
}

type btHeader struct {
Bloom            string
Coinbase         string
MixHash          string
Nonce            string
Number           string
Hash             string
ParentHash       string
ReceiptTrie      string
SeedHash         string
StateRoot        string
TransactionsTrie string
UncleHash        string

ExtraData  string
Difficulty string
GasLimit   string
GasUsed    string
Timestamp  string
}

type btTransaction struct {
Data     string
GasLimit string
GasPrice string
Nonce    string
R        string
S        string
To       string
```

```
V       string
Value   string
}

func RunBlockTestWithReader(homesteadBlock, daoForkBlock, gasPriceFork *big.Int, r
io.Reader, skipTests []string) error {
btjs := make(map[string]*btJSON)
if err := readJson(r, &btjs); err != nil {
return err
}

bt, err := convertBlockTests(btjs)
if err != nil {
return err
}

if err := runBlockTests(homesteadBlock, daoForkBlock, gasPriceFork, bt, skipTests); err != nil {
return err
}
return nil
}

func RunBlockTest(homesteadBlock, daoForkBlock, gasPriceFork *big.Int, file string, skipTests
[]string) error {
btjs := make(map[string]*btJSON)
if err := readJsonFile(file, &btjs); err != nil {
return err
}

bt, err := convertBlockTests(btjs)
if err != nil {
return err
}
if err := runBlockTests(homesteadBlock, daoForkBlock, gasPriceFork, bt, skipTests); err != nil {
return err
}
return nil
}

func runBlockTests(homesteadBlock, daoForkBlock, gasPriceFork *big.Int, bt
map[string]*BlockTest, skipTests []string) error {
skipTest := make(map[string]bool, len(skipTests))
```

```
for _, name := range skipTests {
skipTest[name] = true
}

for name, test := range bt {
if skipTest[name] /*|| name != "CallingCanonicalContractFromFork_CALLCODE"*/ {
log.Info(fmt.Sprint("Skipping block test", name))
continue
}
// test the block
if err := runBlockTest(homesteadBlock, daoForkBlock, gasPriceFork, test); err != nil {
return fmt.Errorf("%s: %v", name, err)
}
log.Info(fmt.Sprint("Block test passed: ", name))

}
return nil
}

func runBlockTest(homesteadBlock, daoForkBlock, gasPriceFork *big.Int, test *BlockTest) error {
// import pre accounts & construct test genesis block & state root
db, _ := ethdb.NewMemDatabase()
if _, err := test.InsertPreState(db); err != nil {
return fmt.Errorf("InsertPreState: %v", err)
}

core.WriteTd(db, test.Genesis.Hash(), 0, test.Genesis.Difficulty())
core.WriteBlock(db, test.Genesis)
core.WriteCanonicalHash(db, test.Genesis.Hash(), test.Genesis.NumberU64())
core.WriteHeadBlockHash(db, test.Genesis.Hash())
evmux := new(event.TypeMux)
config := &params.ChainConfig{HomesteadBlock: homesteadBlock, DAOForkBlock:
daoForkBlock, DAOForkSupport: true, EIP150Block: gasPriceFork}
chain, err := core.NewBlockChain(db, config, ethash.NewShared(), evmux, vm.Config{})
if err != nil {
return err
}
defer chain.Stop()

//vm.Debug = true
validBlocks, err := test.TryBlocksInsert(chain)
if err != nil {
```

```
return err
}

lastblockhash := common.HexToHash(test.lastblockhash)
cmlast := chain.LastBlockHash()
if lastblockhash != cmlast {
return fmt.Errorf("lastblockhash validation mismatch: want: %x, have: %x", lastblockhash, cmlast)
}

newDB, err := chain.State()
if err != nil {
return err
}
if err = test.ValidatePostState(newDB); err != nil {
return fmt.Errorf("post state validation failed: %v", err)
}

return test.ValidateImportedHeaders(chain, validBlocks)
}

// InsertPreState populates the given database with the genesis
// accounts defined by the test.
func (t *BlockTest) InsertPreState(db ethdb.Database) (*state.StateDB, error) {
statedb, err := state.New(common.Hash{}, state.NewDatabase(db))
if err != nil {
return nil, err
}
for addrString, acct := range t.preAccounts {
code, err := hex.DecodeString(strings.TrimPrefix(acct.Code, "0x"))
if err != nil {
return nil, err
}
balance, ok := new(big.Int).SetString(acct.Balance, 0)
if !ok {
return nil, err
}
nonce, err := strconv.ParseUint(prepInt(16, acct.Nonce), 16, 64)
if err != nil {
return nil, err
}

addr := common.HexToAddress(addrString)
```

```
statedb.CreateAccount(addr)
statedb.SetCode(addr, code)
statedb.SetBalance(addr, balance)
statedb.SetNonce(addr, nonce)
for k, v := range acct.Storage {
statedb.SetState(common.HexToAddress(addrString), common.HexToHash(k),
common.HexToHash(v))
}
}

root, err := statedb.CommitTo(db, false)
if err != nil {
return nil, fmt.Errorf("error writing state: %v", err)
}
if t.Genesis.Root() != root {
return nil, fmt.Errorf("computed state root does not match genesis block: genesis=%x
computed=%x", t.Genesis.Root().Bytes()[:4], root.Bytes()[:4])
}
return statedb, nil
}

/* See https://github.com/ethereum/tests/wiki/Blockchain-Tests-II

   Whether a block is valid or not is a bit subtle, it's defined by presence of
   blockHeader, transactions and uncleHeaders fields. If they are missing, the block is
   invalid and we must verify that we do not accept it.

   Since some tests mix valid and invalid blocks we need to check this for every block.

   If a block is invalid it does not necessarily fail the test, if it's invalidness is
   expected we are expected to ignore it and continue processing and then validate the
   post state.
*/
func (t *BlockTest) TryBlocksInsert(blockchain *core.BlockChain) ([]btBlock, error) {
validBlocks := make([]btBlock, 0)
// insert the test blocks, which will execute all transactions
for _, b := range t.Json.Blocks {
cb, err := mustConvertBlock(b)
if err != nil {
if b.BlockHeader == nil {
continue // OK - block is supposed to be invalid, continue with next block
} else {
```

```
return nil, fmt.Errorf("Block RLP decoding failed when expected to succeed: %v", err)
}
}
// RLP decoding worked, try to insert into chain:
blocks := types.Blocks{cb}
i, err := blockchain.InsertChain(blocks)
if err != nil {
if b.BlockHeader == nil {
continue // OK - block is supposed to be invalid, continue with next block
} else {
return nil, fmt.Errorf("Block #%v insertion into chain failed: %v", blocks[i].Number(), err)
}
}
if b.BlockHeader == nil {
return nil, fmt.Errorf("Block insertion should have failed")
}

// validate RLP decoding by checking all values against test file JSON
if err = validateHeader(b.BlockHeader, cb.Header()); err != nil {
return nil, fmt.Errorf("Deserialised block header validation failed: %v", err)
}
validBlocks = append(validBlocks, b)
}
return validBlocks, nil
}

func validateHeader(h *btHeader, h2 *types.Header) error {
expectedBloom := mustConvertBytes(h.Bloom)
if !bytes.Equal(expectedBloom, h2.Bloom.Bytes()) {
return fmt.Errorf("Bloom: want: %x have: %x", expectedBloom, h2.Bloom.Bytes())
}

expectedCoinbase := mustConvertBytes(h.Coinbase)
if !bytes.Equal(expectedCoinbase, h2.Coinbase.Bytes()) {
return fmt.Errorf("Coinbase: want: %x have: %x", expectedCoinbase, h2.Coinbase.Bytes())
}

expectedMixHashBytes := mustConvertBytes(h.MixHash)
if !bytes.Equal(expectedMixHashBytes, h2.MixDigest.Bytes()) {
return fmt.Errorf("MixHash: want: %x have: %x", expectedMixHashBytes, h2.MixDigest.Bytes())
}
```

```
expectedNonce := mustConvertBytes(h.Nonce)
if !bytes.Equal(expectedNonce, h2.Nonce[:]) {
return fmt.Errorf("Nonce: want: %x have: %x", expectedNonce, h2.Nonce)
}

expectedNumber := mustConvertBigInt(h.Number, 16)
if expectedNumber.Cmp(h2.Number) != 0 {
return fmt.Errorf("Number: want: %v have: %v", expectedNumber, h2.Number)
}

expectedParentHash := mustConvertBytes(h.ParentHash)
if !bytes.Equal(expectedParentHash, h2.ParentHash.Bytes()) {
return fmt.Errorf("Parent hash: want: %x have: %x", expectedParentHash, h2.ParentHash.Bytes())
}

expectedReceiptHash := mustConvertBytes(h.ReceiptTrie)
if !bytes.Equal(expectedReceiptHash, h2.ReceiptHash.Bytes()) {
return fmt.Errorf("Receipt hash: want: %x have: %x", expectedReceiptHash,
h2.ReceiptHash.Bytes())
}

expectedTxHash := mustConvertBytes(h.TransactionsTrie)
if !bytes.Equal(expectedTxHash, h2.TxHash.Bytes()) {
return fmt.Errorf("Tx hash: want: %x have: %x", expectedTxHash, h2.TxHash.Bytes())
}

expectedStateHash := mustConvertBytes(h.StateRoot)
if !bytes.Equal(expectedStateHash, h2.Root.Bytes()) {
return fmt.Errorf("State hash: want: %x have: %x", expectedStateHash, h2.Root.Bytes())
}

expectedUncleHash := mustConvertBytes(h.UncleHash)
if !bytes.Equal(expectedUncleHash, h2.UncleHash.Bytes()) {
return fmt.Errorf("Uncle hash: want: %x have: %x", expectedUncleHash, h2.UncleHash.Bytes())
}

expectedExtraData := mustConvertBytes(h.ExtraData)
if !bytes.Equal(expectedExtraData, h2.Extra) {
return fmt.Errorf("Extra data: want: %x have: %x", expectedExtraData, h2.Extra)
}

expectedDifficulty := mustConvertBigInt(h.Difficulty, 16)
```

```
if expectedDifficulty.Cmp(h2.Difficulty) != 0 {
return fmt.Errorf("Difficulty: want: %v have: %v", expectedDifficulty, h2.Difficulty)
}

expectedGasLimit := mustConvertBigInt(h.GasLimit, 16)
if expectedGasLimit.Cmp(h2.GasLimit) != 0 {
return fmt.Errorf("GasLimit: want: %v have: %v", expectedGasLimit, h2.GasLimit)
}
expectedGasUsed := mustConvertBigInt(h.GasUsed, 16)
if expectedGasUsed.Cmp(h2.GasUsed) != 0 {
return fmt.Errorf("GasUsed: want: %v have: %v", expectedGasUsed, h2.GasUsed)
}

expectedTimestamp := mustConvertBigInt(h.Timestamp, 16)
if expectedTimestamp.Cmp(h2.Time) != 0 {
return fmt.Errorf("Timestamp: want: %v have: %v", expectedTimestamp, h2.Time)
}

return nil
}

func (t *BlockTest) ValidatePostState(statedb *state.StateDB) error {
// validate post state accounts in test file against what we have in state db
for addrString, acct := range t.postAccounts {
// XXX: is is worth it checking for errors here?
addr, err := hex.DecodeString(addrString)
if err != nil {
return err
}
code, err := hex.DecodeString(strings.TrimPrefix(acct.Code, "0x"))
if err != nil {
return err
}
balance, ok := new(big.Int).SetString(acct.Balance, 0)
if !ok {
return err
}
nonce, err := strconv.ParseUint(prepInt(16, acct.Nonce), 16, 64)
if err != nil {
return err
}
// address is indirectly verified by the other fields, as it's the db key
code2 := statedb.GetCode(common.BytesToAddress(addr))
balance2 := statedb.GetBalance(common.BytesToAddress(addr))
nonce2 := statedb.GetNonce(common.BytesToAddress(addr))
if !bytes.Equal(code2, code) {
return fmt.Errorf("account code mismatch for addr: %s want: %s have: %s", addrString,
hex.EncodeToString(code), hex.EncodeToString(code2))
}
if balance2.Cmp(balance) != 0 {
return fmt.Errorf("account balance mismatch for addr: %s, want: %d, have: %d", addrString,
balance, balance2)
}
if nonce2 != nonce {
return fmt.Errorf("account nonce mismatch for addr: %s want: %d have: %d", addrString, nonce,
nonce2)
}
}
return nil
}

func (test *BlockTest) ValidateImportedHeaders(cm *core.BlockChain, validBlocks []btBlock) error
{
// to get constant lookup when verifying block headers by hash (some tests have many blocks)
bmap := make(map[string]btBlock, len(test.Json.Blocks))
for _, b := range validBlocks {
bmap[b.BlockHeader.Hash] = b
}

// iterate over blocks backwards from HEAD and validate imported
// headers vs test file. some tests have reorgs, and we import
// block-by-block, so we can only validate imported headers after
// all blocks have been processed by ChainManager, as they may not
// be part of the longest chain until last block is imported.
for b := cm.CurrentBlock(); b != nil && b.NumberU64() != 0; b =
cm.GetBlockByHash(b.Header().ParentHash) {
bHash := common.Bytes2Hex(b.Hash().Bytes()) // hex without 0x prefix
if err := validateHeader(bmap[bHash].BlockHeader, b.Header()); err != nil {
return fmt.Errorf("Imported block header validation failed: %v", err)
}
}
return nil
}
```

```
func convertBlockTests(in map[string]*btJSON) (map[string]*BlockTest, error) {
out := make(map[string]*BlockTest)
for name, test := range in {
var err error
if out[name], err = convertBlockTest(test); err != nil {
return out, fmt.Errorf("bad test %q: %v", name, err)
}
}
return out, nil
}

func convertBlockTest(in *btJSON) (out *BlockTest, err error) {
// the conversion handles errors by catching panics.
// you might consider this ugly, but the alternative (passing errors)
// would be much harder to read.
defer func() {
if recovered := recover(); recovered != nil {
buf := make([]byte, 64<<10)
buf = buf[:runtime.Stack(buf, false)]
err = fmt.Errorf("%v\n%s", recovered, buf)
}
}()
out = &BlockTest{preAccounts: in.Pre, postAccounts: in.PostState, Json: in, lastblockhash:
in.Lastblockhash}
out.Genesis = mustConvertGenesis(in.GenesisBlockHeader)
return out, err
}

func mustConvertGenesis(testGenesis btHeader) *types.Block {
hdr := mustConvertHeader(testGenesis)
hdr.Number = big.NewInt(0)

return types.NewBlockWithHeader(hdr)
}

func mustConvertHeader(in btHeader) *types.Header {
// hex decode these fields
header := &types.Header{
//SeedHash:    mustConvertBytes(in.SeedHash),
MixDigest:   mustConvertHash(in.MixHash),
Bloom:       mustConvertBloom(in.Bloom),
ReceiptHash: mustConvertHash(in.ReceiptTrie),
TxHash:      mustConvertHash(in.TransactionsTrie),
Root:        mustConvertHash(in.StateRoot),
Coinbase:    mustConvertAddress(in.Coinbase),
UncleHash:   mustConvertHash(in.UncleHash),
ParentHash:  mustConvertHash(in.ParentHash),
Extra:       mustConvertBytes(in.ExtraData),
GasUsed:     mustConvertBigInt(in.GasUsed, 16),
GasLimit:    mustConvertBigInt(in.GasLimit, 16),
Difficulty:  mustConvertBigInt(in.Difficulty, 16),
Time:        mustConvertBigInt(in.Timestamp, 16),
Nonce:       types.EncodeNonce(mustConvertUint(in.Nonce, 16)),
}
return header
}

func mustConvertBlock(testBlock btBlock) (*types.Block, error) {
var b types.Block
r := bytes.NewReader(mustConvertBytes(testBlock.Rlp))
err := rlp.Decode(r, &b)
return &b, err
}

func mustConvertBytes(in string) []byte {
if in == "0x" {
return []byte{}
}
h := unfuckFuckedHex(strings.TrimPrefix(in, "0x"))
out, err := hex.DecodeString(h)
if err != nil {
panic(fmt.Errorf("invalid hex: %q", h))
}
return out
}

func mustConvertHash(in string) common.Hash {
out, err := hex.DecodeString(strings.TrimPrefix(in, "0x"))
if err != nil {
panic(fmt.Errorf("invalid hex: %q", in))
}
return common.BytesToHash(out)
}
```

```
func mustConvertAddress(in string) common.Address {
out, err := hex.DecodeString(strings.TrimPrefix(in, "0x"))
if err != nil {
panic(fmt.Errorf("invalid hex: %q", in))
}
return common.BytesToAddress(out)
}

func mustConvertBloom(in string) types.Bloom {
out, err := hex.DecodeString(strings.TrimPrefix(in, "0x"))
if err != nil {
panic(fmt.Errorf("invalid hex: %q", in))
}
return types.BytesToBloom(out)
}

func mustConvertBigInt(in string, base int) *big.Int {
in = prepInt(base, in)
out, ok := new(big.Int).SetString(in, base)
if !ok {
panic(fmt.Errorf("invalid integer: %q", in))
}
return out
}

func mustConvertUint(in string, base int) uint64 {
in = prepInt(base, in)
out, err := strconv.ParseUint(in, base, 64)
if err != nil {
panic(fmt.Errorf("invalid integer: %q", in))
}
return out
}

func LoadBlockTests(file string) (map[string]*BlockTest, error) {
btjs := make(map[string]*btJSON)
if err := readJsonFile(file, &btjs); err != nil {
return nil, err
}

return convertBlockTests(btjs)
}

// Nothing to see here, please move along...
func prepInt(base int, s string) string {
if base == 16 {
s = strings.TrimPrefix(s, "0x")
if len(s) == 0 {
s = "00"
}
s = nibbleFix(s)
}
return s
}

// don't ask
func unfuckFuckedHex(almostHex string) string {
return nibbleFix(strings.Replace(almostHex, "v", "", -1))
}

func nibbleFix(s string) string {
if len(s)%2 != 0 {
s = "0" + s
}
return s
}

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

// Package tests implements execution of Ethereum JSON tests.
package tests

import (
"encoding/json"
"fmt"
"io"
"io/ioutil"
"net/http"
"os"
"path/filepath"
)
```

```
var (
baseDir            = filepath.Join(".", "files")
blockTestDir       = filepath.Join(baseDir, "BlockchainTests")
stateTestDir       = filepath.Join(baseDir, "StateTests")
transactionTestDir = filepath.Join(baseDir, "TransactionTests")
vmTestDir          = filepath.Join(baseDir, "VMTests")
rlpTestDir         = filepath.Join(baseDir, "RLPTests")

BlockSkipTests = []string{
// These tests are not valid, as they are out of scope for RLP and
// the consensus protocol.
"BLOCK__RandomByteAtTheEnd",
"TRANSCT__RandomByteAtTheEnd",
"BLOCK__ZeroByteAtTheEnd",
"TRANSCT__ZeroByteAtTheEnd",

"ChainAtoChainB_blockorder2",
"ChainAtoChainB_blockorder1",

"GasLimitHigherThan2p63m1", // not yet ;)
"SuicideIssue",             // fails genesis check
}

/* Go client does not support transaction (account) nonces above 2^64. This
technically breaks consensus but is regarded as "reasonable
engineering constraint" as accounts cannot easily reach such high
nonce values in practice
*/
TransSkipTests = []string{
"TransactionWithHihghNonce256",
"Vitalik_15",
"Vitalik_16",
"Vitalik_17",
}
StateSkipTests = []string{}
VmSkipTests    = []string{}
)

func readJson(reader io.Reader, value interface{}) error {
data, err := ioutil.ReadAll(reader)
if err != nil {
return fmt.Errorf("error reading JSON file: %v", err)
```

```
}
if err = json.Unmarshal(data, &value); err != nil {
if syntaxerr, ok := err.(*json.SyntaxError); ok {
line := findLine(data, syntaxerr.Offset)
return fmt.Errorf("JSON syntax error at line %v: %v", line, err)
}
return fmt.Errorf("JSON unmarshal error: %v", err)
}
return nil
}

func readJsonHttp(uri string, value interface{}) error {
resp, err := http.Get(uri)
if err != nil {
return err
}
defer resp.Body.Close()

return readJson(resp.Body, value)
}

func readJsonFile(fn string, value interface{}) error {
file, err := os.Open(fn)
if err != nil {
return err
}
defer file.Close()

err = readJson(file, value)
if err != nil {
return fmt.Errorf("%s in file %s", err.Error(), fn)
}
return nil
}

// findLine returns the line number for the given offset into data.
func findLine(data []byte, offset int64) (line int) {
line = 1
for i, r := range string(data) {
if int64(i) >= offset {
return
}
```

```
if r == '\n' {
line++
}
}
return
}
```

69:F:\git\coin\ethereum\go-ethereum\tests\rlp_test.go

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

package tests

import (
"path/filepath"
"testing"
)

func TestRLP(t *testing.T) {
err := RunRLPTest(filepath.Join(rlpTestDir, "rlptest.json"), nil)
if err != nil {
t.Fatal(err)
}
}

func TestRLP_invalid(t *testing.T) {
err := RunRLPTest(filepath.Join(rlpTestDir, "invalidRLPTest.json"), nil)
if err != nil {
t.Fatal(err)
}
}
```

70:F:\git\coin\ethereum\go-ethereum\tests\rlp_test_util.go

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

package tests

import (
"bytes"
"encoding/hex"
"errors"
"fmt"
"io"
```

```
"math/big"
"os"
"strings"

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

// RLPTest is the JSON structure of a single RLP test.
type RLPTest struct {
// If the value of In is "INVALID" or "VALID", the test
// checks whether Out can be decoded into a value of
// type interface{}.
//
// For other JSON values, In is treated as a driver for
// calls to rlp.Stream. The test also verifies that encoding
// In produces the bytes in Out.
In interface{}

// Out is a hex-encoded RLP value.
Out string
}

// RunRLPTest runs the tests in the given file, skipping tests by name.
func RunRLPTest(file string, skip []string) error {
f, err := os.Open(file)
if err != nil {
return err
}
defer f.Close()
return RunRLPTestWithReader(f, skip)
}

// RunRLPTest runs the tests encoded in r, skipping tests by name.
func RunRLPTestWithReader(r io.Reader, skip []string) error {
var tests map[string]*RLPTest
if err := readJson(r, &tests); err != nil {
return err
}
for _, s := range skip {
delete(tests, s)
}
for name, test := range tests {
```

```
if err := test.Run(); err != nil {
return fmt.Errorf("test %q failed: %v", name, err)
}
}
return nil
}

// Run executes the test.
func (t *RLPTest) Run() error {
outb, err := hex.DecodeString(t.Out)
if err != nil {
return fmt.Errorf("invalid hex in Out")
}

// Handle simple decoding tests with no actual In value.
if t.In == "VALID" || t.In == "INVALID" {
return checkDecodeInterface(outb, t.In == "VALID")
}

// Check whether encoding the value produces the same bytes.
in := translateJSON(t.In)
b, err := rlp.EncodeToBytes(in)
if err != nil {
return fmt.Errorf("encode failed: %v", err)
}
if !bytes.Equal(b, outb) {
return fmt.Errorf("encode produced %x, want %x", b, outb)
}
// Test stream decoding.
s := rlp.NewStream(bytes.NewReader(outb), 0)
return checkDecodeFromJSON(s, in)
}

func checkDecodeInterface(b []byte, isValid bool) error {
err := rlp.DecodeBytes(b, new(interface{}))
switch {
case isValid && err != nil:
return fmt.Errorf("decoding failed: %v", err)
case !isValid && err == nil:
return fmt.Errorf("decoding of invalid value succeeded")
}
return nil
```

```
}

// translateJSON makes test json values encodable with RLP.
func translateJSON(v interface{}) interface{} {
switch v := v.(type) {
case float64:
return uint64(v)
case string:
if len(v) > 0 && v[0] == '#' { // # starts a faux big int.
big, ok := new(big.Int).SetString(v[1:], 10)
if !ok {
panic(fmt.Errorf("bad test: bad big int: %q", v))
}
return big
}
return []byte(v)
case []interface{}:
new := make([]interface{}, len(v))
for i := range v {
new[i] = translateJSON(v[i])
}
return new
default:
panic(fmt.Errorf("can't handle %T", v))
}
}

// checkDecodeFromJSON decodes from s guided by exp. exp drives the
// Stream by invoking decoding operations (Uint, Big, List, ...) based
// on the type of each value. The value decoded from the RLP stream
// must match the JSON value.
func checkDecodeFromJSON(s *rlp.Stream, exp interface{}) error {
switch exp := exp.(type) {
case uint64:
i, err := s.Uint()
if err != nil {
return addStack("Uint", exp, err)
}
if i != exp {
return addStack("Uint", exp, fmt.Errorf("result mismatch: got %d", i))
}
case *big.Int:
```

```
big := new(big.Int)
if err := s.Decode(&big); err != nil {
return addStack("Big", exp, err)
}
if big.Cmp(exp) != 0 {
return addStack("Big", exp, fmt.Errorf("result mismatch: got %d", big))
}
case []byte:
b, err := s.Bytes()
if err != nil {
return addStack("Bytes", exp, err)
}
if !bytes.Equal(b, exp) {
return addStack("Bytes", exp, fmt.Errorf("result mismatch: got %x", b))
}
case []interface{}:
if _, err := s.List(); err != nil {
return addStack("List", exp, err)
}
for i, v := range exp {
if err := checkDecodeFromJSON(s, v); err != nil {
return addStack(fmt.Sprintf("[%d]", i), exp, err)
}
}
if err := s.ListEnd(); err != nil {
return addStack("ListEnd", exp, err)
}
default:
panic(fmt.Errorf("unhandled type: %T", exp))
}
return nil
}

func addStack(op string, val interface{}, err error) error {
lines := strings.Split(err.Error(), "\n")
lines = append(lines, fmt.Sprintf("\t%s: %v", op, val))
return errors.New(strings.Join(lines, "\n"))
}
```

71:F:\git\coin\ethereum\go-ethereum\tests\state_test.go

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

```
package tests

import (
"math/big"
"os"
"path/filepath"
"testing"

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

func BenchmarkStateCall1024(b *testing.B) {
fn := filepath.Join(stateTestDir, "stCallCreateCallCodeTest.json")
if err := BenchVmTest(fn, bconf{"Call1024BalanceTooLow", true, os.Getenv("JITVM") == "true"},
b); err != nil {
b.Error(err)
}
}

func TestStateSystemOperations(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stSystemOperationsTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestStateExample(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stExample.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestStatePreCompiledContracts(t *testing.T) {
```

```
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stPreCompiledContracts.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestStateRecursiveCreate(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stRecursiveCreate.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestStateSpecial(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stSpecialTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestStateRefund(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stRefundTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}
```

```
func TestStateBlockHash(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stBlockHashTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestStateInitCode(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stInitCodeTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestStateLog(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stLogTests.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestStateTransaction(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stTransactionTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestStateTransition(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stTransitionTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestCallCreateCallCode(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stCallCreateCallCodeTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestCallCodes(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stCallCodes.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestMemory(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stMemoryTest.json")
```

```
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestMemoryStress(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

if os.Getenv("TEST_VM_COMPLEX") == "" {
t.Skip()
}
fn := filepath.Join(stateTestDir, "stMemoryStressTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestQuadraticComplexity(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

if os.Getenv("TEST_VM_COMPLEX") == "" {
t.Skip()
}
fn := filepath.Join(stateTestDir, "stQuadraticComplexityTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestSolidity(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stSolidityTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
```

```
}

func TestWallet(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "stWalletTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestStateTestsRandom(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fns, _ := filepath.Glob("./files/StateTests/RandomTests/*")
for _, fn := range fns {
t.Log("running:", fn)
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(fn, err)
}
}
}

// homestead tests
func TestHomesteadDelegateCall(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: big.NewInt(1150000),
}

fn := filepath.Join(stateTestDir, "Homestead", "stDelegatecallTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadStateSystemOperations(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stSystemOperationsTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadStatePreCompiledContracts(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stPreCompiledContracts.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadStateRecursiveCreate(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stSpecialTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadStateRefund(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stRefundTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadStateInitCode(t *testing.T) {
```

```
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stInitCodeTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadStateLog(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stLogTests.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadStateTransaction(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stTransactionTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadCallCreateCallCode(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stCallCreateCallCodeTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}
```

```
func TestHomesteadCallCodes(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stCallCodes.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadMemory(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stMemoryTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadMemoryStress(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

if os.Getenv("TEST_VM_COMPLEX") == "" {
t.Skip()
}
fn := filepath.Join(stateTestDir, "Homestead", "stMemoryStressTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadQuadraticComplexity(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}
if os.Getenv("TEST_VM_COMPLEX") == "" {
t.Skip()
}
fn := filepath.Join(stateTestDir, "Homestead", "stQuadraticComplexityTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadWallet(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stWalletTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadDelegateCodes(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stCallDelegateCodes.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestHomesteadDelegateCodesCallCode(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stCallDelegateCodesCallCode.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}
```

```
func TestHomesteadBounds(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
}

fn := filepath.Join(stateTestDir, "Homestead", "stBoundsTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

// EIP150 tests
func TestEIP150Specific(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "stEIPSpecificTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150SingleCodeGasPrice(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "stEIPSingleCodeGasPrices.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150MemExpandingCalls(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}
fn := filepath.Join(stateTestDir, "EIP150", "stMemExpandingEIPCalls.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150HomesteadStateSystemOperations(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stSystemOperationsTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150HomesteadStatePreCompiledContracts(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stPreCompiledContracts.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150HomesteadStateRecursiveCreate(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stSpecialTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}
```

```
func TestEIP150HomesteadStateRefund(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stRefundTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150HomesteadStateInitCode(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stInitCodeTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150HomesteadStateLog(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stLogTests.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150HomesteadStateTransaction(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}
fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stTransactionTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150HomesteadCallCreateCallCode(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stCallCreateCallCodeTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150HomesteadCallCodes(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stCallCodes.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150HomesteadMemory(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stMemoryTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}
```

```
func TestEIP150HomesteadMemoryStress(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

if os.Getenv("TEST_VM_COMPLEX") == "" {
t.Skip()
}
fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stMemoryStressTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150HomesteadQuadraticComplexity(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

if os.Getenv("TEST_VM_COMPLEX") == "" {
t.Skip()
}
fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stQuadraticComplexityTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150HomesteadWallet(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stWalletTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}
```

```
func TestEIP150HomesteadDelegateCodes(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stCallDelegateCodes.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150HomesteadDelegateCodesCallCode(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stCallDelegateCodesCallCode.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP150HomesteadBounds(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
}

fn := filepath.Join(stateTestDir, "EIP150", "Homestead", "stBoundsTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

// EIP158 tests
func TestEIP158Create(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "stCreateTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158Specific(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "stEIP158SpecificTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158NonZeroCalls(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "stNonZeroCallsTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158ZeroCalls(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "stZeroCallsTest.json")
```

```
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158_150Specific(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "EIP150", "stEIPSpecificTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158_150SingleCodeGasPrice(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "EIP150", "stEIPsingleCodeGasPrices.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158_150MemExpandingCalls(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "EIP150", "stMemExpandingEIPCalls.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
```

```
}

func TestEIP158HomesteadStateSystemOperations(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stSystemOperationsTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadStatePreCompiledContracts(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stPreCompiledContracts.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadStateRecursiveCreate(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stSpecialTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadStateRefund(t *testing.T) {
```

```
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stRefundTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadStateInitCode(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stInitCodeTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadStateLog(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stLogTests.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadStateTransaction(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stTransactionTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadCallCreateCallCode(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stCallCreateCallCodeTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadCallCodes(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stCallCodes.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadMemory(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}
```

```
fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stMemoryTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadMemoryStress(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

if os.Getenv("TEST_VM_COMPLEX") == "" {
t.Skip()
}
fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stMemoryStressTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadQuadraticComplexity(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

if os.Getenv("TEST_VM_COMPLEX") == "" {
t.Skip()
}
fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stQuadraticComplexityTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadWallet(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stWalletTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadDelegateCodes(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stCallDelegateCodes.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadDelegateCodesCallCode(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}

fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stCallDelegateCodesCallCode.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

func TestEIP158HomesteadBounds(t *testing.T) {
chainConfig := &params.ChainConfig{
HomesteadBlock: new(big.Int),
EIP150Block:    big.NewInt(2457000),
EIP158Block:    params.MainnetChainConfig.EIP158Block,
}
```

```
fn := filepath.Join(stateTestDir, "EIP158", "Homestead", "stBoundsTest.json")
if err := RunStateTest(chainConfig, fn, StateSkipTests); err != nil {
t.Error(err)
}
}

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

package tests

import (
"bytes"
"fmt"
"io"
"strconv"
"strings"
"testing"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/math"
"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/log"
"github.com/ethereum/go-ethereum/params"
)

func RunStateTestWithReader(chainConfig *params.ChainConfig, r io.Reader, skipTests []string)
error {
tests := make(map[string]VmTest)
if err := readJson(r, &tests); err != nil {
return err
}

if err := runStateTests(chainConfig, tests, skipTests); err != nil {
return err
}

return nil
}
```

```
func RunStateTest(chainConfig *params.ChainConfig, p string, skipTests []string) error {
tests := make(map[string]VmTest)
if err := readJsonFile(p, &tests); err != nil {
return err
}

if err := runStateTests(chainConfig, tests, skipTests); err != nil {
return err
}

return nil

}

func BenchStateTest(chainConfig *params.ChainConfig, p string, conf bconf, b *testing.B) error {
tests := make(map[string]VmTest)
if err := readJsonFile(p, &tests); err != nil {
return err
}
test, ok := tests[conf.name]
if !ok {
return fmt.Errorf("test not found: %s", conf.name)
}

// XXX Yeah, yeah...
env := make(map[string]string)
env["currentCoinbase"] = test.Env.CurrentCoinbase
env["currentDifficulty"] = test.Env.CurrentDifficulty
env["currentGasLimit"] = test.Env.CurrentGasLimit
env["currentNumber"] = test.Env.CurrentNumber
env["previousHash"] = test.Env.PreviousHash
if n, ok := test.Env.CurrentTimestamp.(float64); ok {
env["currentTimestamp"] = strconv.Itoa(int(n))
} else {
env["currentTimestamp"] = test.Env.CurrentTimestamp.(string)
}

b.ResetTimer()
for i := 0; i < b.N; i++ {
benchStateTest(chainConfig, test, env, b)
}
```

```
return nil
}

func benchStateTest(chainConfig *params.ChainConfig, test VmTest, env map[string]string, b
*testing.B) {
b.StopTimer()
db, _ := ethdb.NewMemDatabase()
statedb := makePreState(db, test.Pre)
b.StartTimer()

RunState(chainConfig, statedb, db, env, test.Exec)
}

func runStateTests(chainConfig *params.ChainConfig, tests map[string]VmTest, skipTests []string)
error {
skipTest := make(map[string]bool, len(skipTests))
for _, name := range skipTests {
skipTest[name] = true
}

for name, test := range tests {
if skipTest[name] /*|| name != "JUMPDEST_Attack"*/ {
log.Info(fmt.Sprint("Skipping state test", name))
continue
}

//fmt.Println("StateTest:", name)
if err := runStateTest(chainConfig, test); err != nil {
return fmt.Errorf("%s: %s\n", name, err.Error())
}

//log.Info(fmt.Sprint("State test passed: ", name))
//fmt.Println(string(statedb.Dump()))
}
return nil

}

func runStateTest(chainConfig *params.ChainConfig, test VmTest) error {
db, _ := ethdb.NewMemDatabase()
statedb := makePreState(db, test.Pre)
```

```
// XXX Yeah, yeah...
env := make(map[string]string)
env["currentCoinbase"] = test.Env.CurrentCoinbase
env["currentDifficulty"] = test.Env.CurrentDifficulty
env["currentGasLimit"] = test.Env.CurrentGasLimit
env["currentNumber"] = test.Env.CurrentNumber
env["previousHash"] = test.Env.PreviousHash
if n, ok := test.Env.CurrentTimestamp.(float64); ok {
env["currentTimestamp"] = strconv.Itoa(int(n))
} else {
env["currentTimestamp"] = test.Env.CurrentTimestamp.(string)
}

ret, logs, root, _ := RunState(chainConfig, statedb, db, env, test.Transaction)

// Return value:
var rexp []byte
if strings.HasPrefix(test.Out, "#") {
n, _ := strconv.Atoi(test.Out[1:])
rexp = make([]byte, n)
} else {
rexp = common.FromHex(test.Out)
}
if !bytes.Equal(rexp, ret) {
return fmt.Errorf("return failed. Expected %x, got %x\n", rexp, ret)
}
// Post state content:
for addr, account := range test.Post {
address := common.HexToAddress(addr)
if !statedb.Exist(address) {
return fmt.Errorf("did not find expected post-state account: %s", addr)
}
if balance := statedb.GetBalance(address);
balance.Cmp(math.MustParseBig256(account.Balance)) != 0 {
return fmt.Errorf("(%x) balance failed. Expected: %v have: %v\n", address[:4],
math.MustParseBig256(account.Balance), balance)
}
if nonce := statedb.GetNonce(address); nonce != math.MustParseUint64(account.Nonce) {
return fmt.Errorf("(%x) nonce failed. Expected: %v have: %v\n", address[:4], account.Nonce,
nonce)
}
for addr, value := range account.Storage {
v := statedb.GetState(address, common.HexToHash(addr))
vexp := common.HexToHash(value)
if v != vexp {
return fmt.Errorf("storage failed:\n%x: %s:\nexpected: %x\nhave:     %x\n(%v %v)\n", address[:4],
addr, vexp, v, vexp.Big(), v.Big())
}
}
}
// Root:
if common.HexToHash(test.PostStateRoot) != root {
return fmt.Errorf("Post state root error. Expected: %s have: %x", test.PostStateRoot, root)
}
// Logs:
return checkLogs(test.Logs, logs)
}

func RunState(chainConfig *params.ChainConfig, statedb *state.StateDB, db ethdb.Database,
env, tx map[string]string) ([]byte, []*types.Log, common.Hash, error) {
environment, msg := NewEVMEnvironment(false, chainConfig, statedb, env, tx)
gaspool := new(core.GasPool).AddGas(math.MustParseBig256(env["currentGasLimit"]))

snapshot := statedb.Snapshot()
ret, _, err := core.ApplyMessage(environment, msg, gaspool)
if err != nil {
statedb.RevertToSnapshot(snapshot)
}
root, _ := statedb.CommitTo(db, chainConfig.IsEIP158(environment.Context.BlockNumber))
return ret, statedb.Logs(), root, err
}
```

73:F:\git\coin\ethereum\go-ethereum\tests\transaction_test.go

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

package tests

import (
"math/big"
"path/filepath"
"testing"

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

```
)

func TestTransactions(t *testing.T) {
config := &params.ChainConfig{}
err := RunTransactionTests(config, filepath.Join(transactionTestDir, "ttTransactionTest.json"),
TransSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestWrongRLPTransactions(t *testing.T) {
config := &params.ChainConfig{}
err := RunTransactionTests(config, filepath.Join(transactionTestDir,
"ttWrongRLPTransaction.json"), TransSkipTests)
if err != nil {
t.Fatal(err)
}
}

func Test10MBTransactions(t *testing.T) {
config := &params.ChainConfig{}
err := RunTransactionTests(config, filepath.Join(transactionTestDir, "tt10mbDataField.json"),
TransSkipTests)
if err != nil {
t.Fatal(err)
}
}

// homestead tests
func TestHomesteadTransactions(t *testing.T) {
config := &params.ChainConfig{
HomesteadBlock: big.NewInt(0),
}
err := RunTransactionTests(config, filepath.Join(transactionTestDir, "Homestead",
"ttTransactionTest.json"), TransSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestHomesteadWrongRLPTransactions(t *testing.T) {
```

```
config := &params.ChainConfig{
HomesteadBlock: big.NewInt(0),
}
err := RunTransactionTests(config, filepath.Join(transactionTestDir, "Homestead",
"ttWrongRLPTransaction.json"), TransSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestHomestead10MBTransactions(t *testing.T) {
config := &params.ChainConfig{
HomesteadBlock: big.NewInt(0),
}
err := RunTransactionTests(config, filepath.Join(transactionTestDir, "Homestead",
"tt10mbDataField.json"), TransSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestHomesteadVitalik(t *testing.T) {
config := &params.ChainConfig{
HomesteadBlock: big.NewInt(0),
}
err := RunTransactionTests(config, filepath.Join(transactionTestDir, "Homestead",
"ttTransactionTestEip155VitaliksTests.json"), TransSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestTxEIP155Transaction(t *testing.T) {
config := &params.ChainConfig{
ChainId:        big.NewInt(1),
HomesteadBlock: big.NewInt(0),
EIP155Block:    big.NewInt(0),
}
err := RunTransactionTests(config, filepath.Join(transactionTestDir, "EIP155",
"ttTransactionTest.json"), TransSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestTxEIP155VitaliksTests(t *testing.T) {
config := &params.ChainConfig{
ChainId:        big.NewInt(1),
HomesteadBlock: big.NewInt(0),
EIP155Block:    big.NewInt(0),
}
err := RunTransactionTests(config, filepath.Join(transactionTestDir, "EIP155",
"ttTransactionTestEip155VitaliksTests.json"), TransSkipTests)
if err != nil {
t.Fatal(err)
}
}

func TestTxEIP155VRule(t *testing.T) {
config := &params.ChainConfig{
ChainId:        big.NewInt(1),
HomesteadBlock: big.NewInt(0),
EIP155Block:    big.NewInt(0),
}
err := RunTransactionTests(config, filepath.Join(transactionTestDir, "EIP155",
"ttTransactionTestVRule.json"), TransSkipTests)
if err != nil {
t.Fatal(err)
}
}
```

74:F:\git\coin\ethereum\go-ethereum\tests\transaction_test_util.go

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

package tests

import (
"bytes"
"errors"
"fmt"
"io"
"runtime"

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

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

// Transaction Test JSON Format
type TtTransaction struct {
Data     string
GasLimit string
GasPrice string
Nonce    string
R        string
S        string
To       string
V        string
Value    string
}

type TransactionTest struct {
Blocknumber string
Rlp         string
Sender      string
Transaction TtTransaction
}

func RunTransactionTestsWithReader(config *params.ChainConfig, r io.Reader, skipTests
[]string) error {
skipTest := make(map[string]bool, len(skipTests))
for _, name := range skipTests {
skipTest[name] = true
}

bt := make(map[string]TransactionTest)
if err := readJson(r, &bt); err != nil {
return err
}

for name, test := range bt {
// if the test should be skipped, return
if skipTest[name] {
```

```
log.Info(fmt.Sprint("Skipping transaction test", name))
return nil
}
// test the block
if err := runTransactionTest(config, test); err != nil {
return err
}
log.Info(fmt.Sprint("Transaction test passed: ", name))

}
return nil
}

func RunTransactionTests(config *params.ChainConfig, file string, skipTests []string) error {
tests := make(map[string]TransactionTest)
if err := readJsonFile(file, &tests); err != nil {
return err
}

if err := runTransactionTests(config, tests, skipTests); err != nil {
return err
}
return nil
}

func runTransactionTests(config *params.ChainConfig, tests map[string]TransactionTest,
skipTests []string) error {
skipTest := make(map[string]bool, len(skipTests))
for _, name := range skipTests {
skipTest[name] = true
}

for name, test := range tests {
// if the test should be skipped, return
if skipTest[name] {
log.Info(fmt.Sprint("Skipping transaction test", name))
return nil
}

// test the block
if err := runTransactionTest(config, test); err != nil {
return fmt.Errorf("%s: %v", name, err)
```

```
}
log.Info(fmt.Sprint("Transaction test passed: ", name))

}
return nil
}

func runTransactionTest(config *params.ChainConfig, txTest TransactionTest) (err error) {
tx := new(types.Transaction)
err = rlp.DecodeBytes(mustConvertBytes(txTest.Rlp), tx)

if err != nil {
if txTest.Sender == "" {
// RLP decoding failed and this is expected (test OK)
return nil
} else {
// RLP decoding failed but is expected to succeed (test FAIL)
return fmt.Errorf("RLP decoding failed when expected to succeed: %s", err)
}
}

validationError := verifyTxFields(config, txTest, tx)
if txTest.Sender == "" {
if validationError != nil {
// RLP decoding works but validation should fail (test OK)
return nil
} else {
// RLP decoding works but validation should fail (test FAIL)
// (this should not be possible but added here for completeness)
return errors.New("Field validations succeeded but should fail")
}
}

if txTest.Sender != "" {
if validationError == nil {
// RLP decoding works and validations pass (test OK)
return nil
} else {
// RLP decoding works and validations pass (test FAIL)
return fmt.Errorf("Field validations failed after RLP decoding: %s", validationError)
}
}
```

```
return errors.New("Should not happen: verify RLP decoding and field validation")
}

func verifyTxFields(chainConfig *params.ChainConfig, txTest TransactionTest, decodedTx
*types.Transaction) (err error) {
defer func() {
if recovered := recover(); recovered != nil {
buf := make([]byte, 64<<10)
buf = buf[:runtime.Stack(buf, false)]
err = fmt.Errorf("%v\n%s", recovered, buf)
}
}()

var decodedSender common.Address

signer := types.MakeSigner(chainConfig, math.MustParseBig256(txTest.Blocknumber))
decodedSender, err = types.Sender(signer, decodedTx)
if err != nil {
return err
}

expectedSender := mustConvertAddress(txTest.Sender)
if expectedSender != decodedSender {
return fmt.Errorf("Sender mismatch: %x %x", expectedSender, decodedSender)
}

expectedData := mustConvertBytes(txTest.Transaction.Data)
if !bytes.Equal(expectedData, decodedTx.Data()) {
return fmt.Errorf("Tx input data mismatch: %#v %#v", expectedData, decodedTx.Data())
}

expectedGasLimit := mustConvertBigInt(txTest.Transaction.GasLimit, 16)
if expectedGasLimit.Cmp(decodedTx.Gas()) != 0 {
return fmt.Errorf("GasLimit mismatch: %v %v", expectedGasLimit, decodedTx.Gas())
}

expectedGasPrice := mustConvertBigInt(txTest.Transaction.GasPrice, 16)
if expectedGasPrice.Cmp(decodedTx.GasPrice()) != 0 {
return fmt.Errorf("GasPrice mismatch: %v %v", expectedGasPrice, decodedTx.GasPrice())
}

expectedNonce := mustConvertUint(txTest.Transaction.Nonce, 16)
```

```
if expectedNonce != decodedTx.Nonce() {
return fmt.Errorf("Nonce mismatch: %v %v", expectedNonce, decodedTx.Nonce())
}

v, r, s := decodedTx.RawSignatureValues()
expectedR := mustConvertBigInt(txTest.Transaction.R, 16)
if r.Cmp(expectedR) != 0 {
return fmt.Errorf("R mismatch: %v %v", expectedR, r)
}
expectedS := mustConvertBigInt(txTest.Transaction.S, 16)
if s.Cmp(expectedS) != 0 {
return fmt.Errorf("S mismatch: %v %v", expectedS, s)
}
expectedV := mustConvertBigInt(txTest.Transaction.V, 16)
if v.Cmp(expectedV) != 0 {
return fmt.Errorf("V mismatch: %v %v", expectedV, v)
}

expectedTo := mustConvertAddress(txTest.Transaction.To)
if decodedTx.To() == nil {
if expectedTo != common.BytesToAddress([]byte{}) { // "empty" or "zero" address
return fmt.Errorf("To mismatch when recipient is nil (contract creation): %v", expectedTo)
}
} else {
if expectedTo != *decodedTx.To() {
return fmt.Errorf("To mismatch: %v %v", expectedTo, *decodedTx.To())
}
}

expectedValue := mustConvertBigInt(txTest.Transaction.Value, 16)
if expectedValue.Cmp(decodedTx.Value()) != 0 {
return fmt.Errorf("Value mismatch: %v %v", expectedValue, decodedTx.Value())
}

return nil
}
```

75:F:\git\coin\ethereum\go-ethereum\tests\util.go

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

package tests
```

```
import (
"bytes"
"fmt"
"math/big"
"os"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/math"
"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/log"
"github.com/ethereum/go-ethereum/params"
)

var (
ForceJit  bool
EnableJit bool
)

func init() {
log.Root().SetHandler(log.LvlFilterHandler(log.LvlCrit, log.StreamHandler(os.Stderr,
log.TerminalFormat(false))))
if os.Getenv("JITVM") == "true" {
ForceJit = true
EnableJit = true
}
}

func checkLogs(tlog []Log, logs []*types.Log) error {
if len(tlog) != len(logs) {
return fmt.Errorf("log length mismatch. Expected %d, got %d", len(tlog), len(logs))
} else {
for i, log := range tlog {
if common.HexToAddress(log.AddressF) != logs[i].Address {
return fmt.Errorf("log address expected %v got %x", log.AddressF, logs[i].Address)
}

if !bytes.Equal(logs[i].Data, common.FromHex(log.DataF)) {
```

```
return fmt.Errorf("log data expected %v got %x", log.DataF, logs[i].Data)
}

if len(log.TopicsF) != len(logs[i].Topics) {
return fmt.Errorf("log topics length expected %d got %d", len(log.TopicsF), logs[i].Topics)
} else {
for j, topic := range log.TopicsF {
if common.HexToHash(topic) != logs[i].Topics[j] {
return fmt.Errorf("log topic[%d] expected %v got %x", j, topic, logs[i].Topics[j])
}
}
}
genBloom := math.PaddedBigBytes(types.LogsBloom([]*types.Log{logs[i]}), 256)

if !bytes.Equal(genBloom, common.Hex2Bytes(log.BloomF)) {
return fmt.Errorf("bloom mismatch")
}
}
}
return nil
}

type Account struct {
Balance string
Code    string
Nonce   string
Storage map[string]string
}

type Log struct {
AddressF string   `json:"address"`
DataF    string   `json:"data"`
TopicsF  []string `json:"topics"`
BloomF   string   `json:"bloom"`
}

func (self Log) Address() []byte      { return common.Hex2Bytes(self.AddressF) }
func (self Log) Data() []byte         { return common.Hex2Bytes(self.DataF) }
func (self Log) RlpData() interface{} { return nil }
func (self Log) Topics() [][]byte {
t := make([][]byte, len(self.TopicsF))
for i, topic := range self.TopicsF {
```

```
t[i] = common.Hex2Bytes(topic)
}
return t
}

func makePreState(db ethdb.Database, accounts map[string]Account) *state.StateDB {
sdb := state.NewDatabase(db)
statedb, _ := state.New(common.Hash{}, sdb)
for addr, account := range accounts {
insertAccount(statedb, addr, account)
}
// Commit and re-open to start with a clean state.
root, _ := statedb.CommitTo(db, false)
statedb, _ = state.New(root, sdb)
return statedb
}

func insertAccount(state *state.StateDB, saddr string, account Account) {
if common.IsHex(account.Code) {
account.Code = account.Code[2:]
}
addr := common.HexToAddress(saddr)
state.SetCode(addr, common.Hex2Bytes(account.Code))
state.SetNonce(addr, math.MustParseUint64(account.Nonce))
state.SetBalance(addr, math.MustParseBig256(account.Balance))
for a, v := range account.Storage {
state.SetState(addr, common.HexToHash(a), common.HexToHash(v))
}
}

type VmEnv struct {
CurrentCoinbase   string
CurrentDifficulty string
CurrentGasLimit   string
CurrentNumber     string
CurrentTimestamp  interface{}
PreviousHash      string
}

type VmTest struct {
Callcreates interface{}
//Env         map[string]string
```

```
Env           VmEnv
Exec          map[string]string
Transaction   map[string]string
Logs          []Log
Gas           string
Out           string
Post          map[string]Account
Pre           map[string]Account
PostStateRoot string
}

func NewEVMEnvironment(vmTest bool, chainConfig *params.ChainConfig, statedb
*state.StateDB, envValues map[string]string, tx map[string]string) (*vm.EVM, core.Message) {
var (
data  = common.FromHex(tx["data"])
gas   = math.MustParseBig256(tx["gasLimit"])
price = math.MustParseBig256(tx["gasPrice"])
value = math.MustParseBig256(tx["value"])
nonce = math.MustParseUint64(tx["nonce"])
)

origin := common.HexToAddress(tx["caller"])
if len(tx["secretKey"]) > 0 {
key, _ := crypto.HexToECDSA(tx["secretKey"])
origin = crypto.PubkeyToAddress(key.PublicKey)
}

var to *common.Address
if len(tx["to"]) > 2 {
t := common.HexToAddress(tx["to"])
to = &t
}

msg := types.NewMessage(origin, to, nonce, value, gas, price, data, true)

initialCall := true
canTransfer := func(db vm.StateDB, address common.Address, amount *big.Int) bool {
if vmTest {
if initialCall {
initialCall = false
return true
}
}
return core.CanTransfer(db, address, amount)
}
transfer := func(db vm.StateDB, sender, recipient common.Address, amount *big.Int) {
if vmTest {
return
}
core.Transfer(db, sender, recipient, amount)
}

context := vm.Context{
CanTransfer: canTransfer,
Transfer:    transfer,
GetHash: func(n uint64) common.Hash {
return common.BytesToHash(crypto.Keccak256([]byte(big.NewInt(int64(n)).String())))
},

Origin:      origin,
Coinbase:    common.HexToAddress(envValues["currentCoinbase"]),
BlockNumber: math.MustParseBig256(envValues["currentNumber"]),
Time:        math.MustParseBig256(envValues["currentTimestamp"]),
GasLimit:    math.MustParseBig256(envValues["currentGasLimit"]),
Difficulty:  math.MustParseBig256(envValues["currentDifficulty"]),
GasPrice:    price,
}
if context.GasPrice == nil {
context.GasPrice = new(big.Int)
}
return vm.NewEVM(context, statedb, chainConfig, vm.Config{NoRecursion: vmTest}), msg
}
```

76:F:\git\coin\ethereum\go-ethereum\tests\vm_test.go

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

package tests

import (
"os"
"path/filepath"
"testing"
)
```

```
func BenchmarkVmAckermann32Tests(b *testing.B) {
fn := filepath.Join(vmTestDir, "vmPerformanceTest.json")
if err := BenchVmTest(fn, bconf{"ackermann32", os.Getenv("JITFORCE") == "true",
os.Getenv("JITVM") == "true"}, b); err != nil {
b.Error(err)
}
}

func BenchmarkVmFibonacci16Tests(b *testing.B) {
fn := filepath.Join(vmTestDir, "vmPerformanceTest.json")
if err := BenchVmTest(fn, bconf{"fibonacci16", os.Getenv("JITFORCE") == "true",
os.Getenv("JITVM") == "true"}, b); err != nil {
b.Error(err)
}
}

// I've created a new function for each tests so it's easier to identify where the problem lies if any of
them fail.
func TestVmVMArithmetic(t *testing.T) {
fn := filepath.Join(vmTestDir, "vmArithmeticTest.json")
if err := RunVmTest(fn, VmSkipTests); err != nil {
t.Error(err)
}
}

func TestVmBitwiseLogicOperation(t *testing.T) {
fn := filepath.Join(vmTestDir, "vmBitwiseLogicOperationTest.json")
if err := RunVmTest(fn, VmSkipTests); err != nil {
t.Error(err)
}
}

func TestVmBlockInfo(t *testing.T) {
fn := filepath.Join(vmTestDir, "vmBlockInfoTest.json")
if err := RunVmTest(fn, VmSkipTests); err != nil {
t.Error(err)
}
}

func TestVmEnvironmentalInfo(t *testing.T) {
fn := filepath.Join(vmTestDir, "vmEnvironmentalInfoTest.json")
if err := RunVmTest(fn, VmSkipTests); err != nil {
```

```
t.Error(err)
}
}

func TestVmFlowOperation(t *testing.T) {
fn := filepath.Join(vmTestDir, "vmIOandFlowOperationsTest.json")
if err := RunVmTest(fn, VmSkipTests); err != nil {
t.Error(err)
}
}

func TestVmLogTest(t *testing.T) {
fn := filepath.Join(vmTestDir, "vmLogTest.json")
if err := RunVmTest(fn, VmSkipTests); err != nil {
t.Error(err)
}
}

func TestVmPerformance(t *testing.T) {
fn := filepath.Join(vmTestDir, "vmPerformanceTest.json")
if err := RunVmTest(fn, VmSkipTests); err != nil {
t.Error(err)
}
}

func TestVmPushDupSwap(t *testing.T) {
fn := filepath.Join(vmTestDir, "vmPushDupSwapTest.json")
if err := RunVmTest(fn, VmSkipTests); err != nil {
t.Error(err)
}
}

func TestVmVMSha3(t *testing.T) {
fn := filepath.Join(vmTestDir, "vmSha3Test.json")
if err := RunVmTest(fn, VmSkipTests); err != nil {
t.Error(err)
}
}

func TestVm(t *testing.T) {
fn := filepath.Join(vmTestDir, "vmtests.json")
if err := RunVmTest(fn, VmSkipTests); err != nil {
```

```
t.Error(err)
}
}

func TestVmLog(t *testing.T) {
fn := filepath.Join(vmTestDir, "vmLogTest.json")
if err := RunVmTest(fn, VmSkipTests); err != nil {
t.Error(err)
}
}

func TestVmInputLimits(t *testing.T) {
fn := filepath.Join(vmTestDir, "vmInputLimits.json")
if err := RunVmTest(fn, VmSkipTests); err != nil {
t.Error(err)
}
}

func TestVmInputLimitsLight(t *testing.T) {
fn := filepath.Join(vmTestDir, "vmInputLimitsLight.json")
if err := RunVmTest(fn, VmSkipTests); err != nil {
t.Error(err)
}
}

func TestVmVMRandom(t *testing.T) {
fns, _ := filepath.Glob(filepath.Join(baseDir, "RandomTests", "*"))
for _, fn := range fns {
if err := RunVmTest(fn, VmSkipTests); err != nil {
t.Error(err)
}
}
}
```

77:F:\git\coin\ethereum\go-ethereum\tests\vm_test_util.go

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

package tests

import (
"bytes"
"fmt"
```

```
"io"
"math/big"
"strconv"
"testing"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/math"
"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/log"
"github.com/ethereum/go-ethereum/params"
)

func RunVmTestWithReader(r io.Reader, skipTests []string) error {
tests := make(map[string]VmTest)
err := readJson(r, &tests)
if err != nil {
return err
}

if err != nil {
return err
}

if err := runVmTests(tests, skipTests); err != nil {
return err
}

return nil
}

type bconf struct {
name    string
precomp bool
jit     bool
}

func BenchVmTest(p string, conf bconf, b *testing.B) error {
tests := make(map[string]VmTest)
err := readJsonFile(p, &tests)
```

```
if err != nil {
return err
}

test, ok := tests[conf.name]
if !ok {
return fmt.Errorf("test not found: %s", conf.name)
}

env := make(map[string]string)
env["currentCoinbase"] = test.Env.CurrentCoinbase
env["currentDifficulty"] = test.Env.CurrentDifficulty
env["currentGasLimit"] = test.Env.CurrentGasLimit
env["currentNumber"] = test.Env.CurrentNumber
env["previousHash"] = test.Env.PreviousHash
if n, ok := test.Env.CurrentTimestamp.(float64); ok {
env["currentTimestamp"] = strconv.Itoa(int(n))
} else {
env["currentTimestamp"] = test.Env.CurrentTimestamp.(string)
}

/*
if conf.precomp {
program := vm.NewProgram(test.code)
err := vm.AttachProgram(program)
if err != nil {
return err
}
}
*/

b.ResetTimer()
for i := 0; i < b.N; i++ {
benchVmTest(test, env, b)
}

return nil
}

func benchVmTest(test VmTest, env map[string]string, b *testing.B) {
b.StopTimer()
db, _ := ethdb.NewMemDatabase()
```

```
statedb := makePreState(db, test.Pre)
b.StartTimer()

RunVm(statedb, env, test.Exec)
}

func RunVmTest(p string, skipTests []string) error {
tests := make(map[string]VmTest)
err := readJsonFile(p, &tests)
if err != nil {
return err
}

if err := runVmTests(tests, skipTests); err != nil {
return err
}

return nil
}

func runVmTests(tests map[string]VmTest, skipTests []string) error {
skipTest := make(map[string]bool, len(skipTests))
for _, name := range skipTests {
skipTest[name] = true
}

for name, test := range tests {
if skipTest[name] /*|| name != "exp0"*/ {
log.Info(fmt.Sprint("Skipping VM test", name))
continue
}

if err := runVmTest(test); err != nil {
return fmt.Errorf("%s %s", name, err.Error())
}

log.Info(fmt.Sprint("VM test passed: ", name))
//fmt.Println(string(statedb.Dump()))
}
return nil
}
```

```
func runVmTest(test VmTest) error {
db, _ := ethdb.NewMemDatabase()
statedb := makePreState(db, test.Pre)

// XXX Yeah, yeah...
env := make(map[string]string)
env["currentCoinbase"] = test.Env.CurrentCoinbase
env["currentDifficulty"] = test.Env.CurrentDifficulty
env["currentGasLimit"] = test.Env.CurrentGasLimit
env["currentNumber"] = test.Env.CurrentNumber
env["previousHash"] = test.Env.PreviousHash
if n, ok := test.Env.CurrentTimestamp.(float64); ok {
env["currentTimestamp"] = strconv.Itoa(int(n))
} else {
env["currentTimestamp"] = test.Env.CurrentTimestamp.(string)
}

var (
ret  []byte
gas  *big.Int
err  error
logs []*types.Log
)

ret, logs, gas, err = RunVm(statedb, env, test.Exec)

// Compare expected and actual return
rexp := common.FromHex(test.Out)
if !bytes.Equal(rexp, ret) {
return fmt.Errorf("return failed. Expected %x, got %x\n", rexp, ret)
}

// Check gas usage
if len(test.Gas) == 0 && err == nil {
return fmt.Errorf("gas unspecified, indicating an error. VM returned (incorrectly) successful")
} else {
gexp := math.MustParseBig256(test.Gas)
if gexp.Cmp(gas) != 0 {
return fmt.Errorf("gas failed. Expected %v, got %v\n", gexp, gas)
}
}
```

```
// check post state
for address, account := range test.Post {
accountAddr := common.HexToAddress(address)
if !statedb.Exist(accountAddr) {
continue
}
for addr, value := range account.Storage {
v := statedb.GetState(accountAddr, common.HexToHash(addr))
vexp := common.HexToHash(value)
if v != vexp {
return fmt.Errorf("(%x: %s) storage failed. Expected %x, got %x (%v %v)\n", addr[:4], addr, vexp,
v, vexp.Big(), v.Big())
}
}
}

// check logs
if len(test.Logs) > 0 {
lerr := checkLogs(test.Logs, logs)
if lerr != nil {
return lerr
}
}

return nil
}

func RunVm(statedb *state.StateDB, env, exec map[string]string) ([]byte, []*types.Log, *big.Int,
error) {
chainConfig := &params.ChainConfig{
HomesteadBlock: params.MainnetChainConfig.HomesteadBlock,
DAOForkBlock:   params.MainnetChainConfig.DAOForkBlock,
DAOForkSupport: true,
}
var (
to    = common.HexToAddress(exec["address"])
from  = common.HexToAddress(exec["caller"])
data  = common.FromHex(exec["data"])
gas   = math.MustParseBig256(exec["gas"])
value = math.MustParseBig256(exec["value"])
)
caller := statedb.GetOrNewStateObject(from)
```

```
vm.PrecompiledContracts = make(map[common.Address]vm.PrecompiledContract)

environment, _ := NewEVMEnvironment(true, chainConfig, statedb, env, exec)
ret, g, err := environment.Call(caller, to, data, gas.Uint64(), value)
return ret, statedb.Logs(), new(big.Int).SetUint64(g), err
}

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

package trie

// Trie keys are dealt with in three distinct encodings:
//
// KEYBYTES encoding contains the actual key and nothing else. This encoding is the
// input to most API functions.
//
// HEX encoding contains one byte for each nibble of the key and an optional trailing
// 'terminator' byte of value 0x10 which indicates whether or not the node at the key
// contains a value. Hex key encoding is used for nodes loaded in memory because it's
// convenient to access.
//
// COMPACT encoding is defined by the Ethereum Yellow Paper (it's called "hex prefix
// encoding" there) and contains the bytes of the key and a flag. The high nibble of the
// first byte contains the flag; the lowest bit encoding the oddness of the length and
// the second-lowest encoding whether the node at the key is a value node. The low nibble
// of the first byte is zero in the case of an even number of nibbles and the first nibble
// in the case of an odd number. All remaining nibbles (now an even number) fit properly
// into the remaining bytes. Compact encoding is used for nodes stored on disk.

func hexToCompact(hex []byte) []byte {
terminator := byte(0)
if hasTerm(hex) {
terminator = 1
hex = hex[:len(hex)-1]
}
buf := make([]byte, len(hex)/2+1)
buf[0] = terminator << 5 // the flag byte
if len(hex)&1 == 1 {
buf[0] |= 1 << 4 // odd flag
buf[0] |= hex[0] // first nibble is contained in the first byte
hex = hex[1:]
}
decodeNibbles(hex, buf[1:])
return buf
}

func compactToHex(compact []byte) []byte {
base := keybytesToHex(compact)
base = base[:len(base)-1]
// apply terminator flag
if base[0] >= 2 {
base = append(base, 16)
}
// apply odd flag
chop := 2 - base[0]&1
return base[chop:]
}

func keybytesToHex(str []byte) []byte {
l := len(str)*2 + 1
var nibbles = make([]byte, l)
for i, b := range str {
nibbles[i*2] = b / 16
nibbles[i*2+1] = b % 16
}
nibbles[l-1] = 16
return nibbles
}

// hexToKeybytes turns hex nibbles into key bytes.
// This can only be used for keys of even length.
func hexToKeybytes(hex []byte) []byte {
if hasTerm(hex) {
hex = hex[:len(hex)-1]
}
if len(hex)&1 != 0 {
panic("can't convert hex key of odd length")
}
key := make([]byte, (len(hex)+1)/2)
decodeNibbles(hex, key)
return key
}
```

```
func decodeNibbles(nibbles []byte, bytes []byte) {
for bi, ni := 0, 0; ni < len(nibbles); bi, ni = bi+1, ni+2 {
bytes[bi] = nibbles[ni]<<4 | nibbles[ni+1]
}
}

// prefixLen returns the length of the common prefix of a and b.
func prefixLen(a, b []byte) int {
var i, length = 0, len(a)
if len(b) < length {
length = len(b)
}
for ; i < length; i++ {
if a[i] != b[i] {
break
}
}
return i
}

// hasTerm returns whether a hex key has the terminator flag.
func hasTerm(s []byte) bool {
return len(s) > 0 && s[len(s)-1] == 16
}
```

79:F:\git\coin\ethereum\go-ethereum\trie\encoding_test.go

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

package trie

import (
"bytes"
"testing"
)

func TestHexCompact(t *testing.T) {
tests := []struct{ hex, compact []byte }{
// empty keys, with and without terminator.
{hex: []byte{}, compact: []byte{0x00}},
{hex: []byte{16}, compact: []byte{0x20}},
// odd length, no terminator
{hex: []byte{1, 2, 3, 4, 5}, compact: []byte{0x11, 0x23, 0x45}},
// even length, no terminator
{hex: []byte{0, 1, 2, 3, 4, 5}, compact: []byte{0x00, 0x01, 0x23, 0x45}},
// odd length, terminator
{hex: []byte{15, 1, 12, 11, 8, 16 /*term*/}, compact: []byte{0x3f, 0x1c, 0xb8}},
// even length, terminator
{hex: []byte{0, 15, 1, 12, 11, 8, 16 /*term*/}, compact: []byte{0x20, 0x0f, 0x1c, 0xb8}},
}
for _, test := range tests {
if c := hexToCompact(test.hex); !bytes.Equal(c, test.compact) {
t.Errorf("hexToCompact(%x) -> %x, want %x", test.hex, c, test.compact)
}
if h := compactToHex(test.compact); !bytes.Equal(h, test.hex) {
t.Errorf("compactToHex(%x) -> %x, want %x", test.compact, h, test.hex)
}
}
}

func TestHexKeybytes(t *testing.T) {
tests := []struct{ key, hexIn, hexOut []byte }{
{key: []byte{}, hexIn: []byte{16}, hexOut: []byte{16}},
{key: []byte{}, hexIn: []byte{}, hexOut: []byte{16}},
{
key:    []byte{0x12, 0x34, 0x56},
hexIn:  []byte{1, 2, 3, 4, 5, 6, 16},
hexOut: []byte{1, 2, 3, 4, 5, 6, 16},
},
{
key:    []byte{0x12, 0x34, 0x5},
hexIn:  []byte{1, 2, 3, 4, 0, 5, 16},
hexOut: []byte{1, 2, 3, 4, 0, 5, 16},
},
{
key:    []byte{0x12, 0x34, 0x56},
hexIn:  []byte{1, 2, 3, 4, 5, 6},
hexOut: []byte{1, 2, 3, 4, 5, 6, 16},
},
}
for _, test := range tests {
if h := keybytesToHex(test.key); !bytes.Equal(h, test.hexOut) {
t.Errorf("keybytesToHex(%x) -> %x, want %x", test.key, h, test.hexOut)
}
if k := hexToKeybytes(test.hexIn); !bytes.Equal(k, test.key) {
```

```
t.Errorf("hexToKeybytes(%x) -> %x, want %x", test.hexIn, k, test.key)
}
}
}

func BenchmarkHexToCompact(b *testing.B) {
testBytes := []byte{0, 15, 1, 12, 11, 8, 16 /*term*/}
for i := 0; i < b.N; i++ {
hexToCompact(testBytes)
}
}

func BenchmarkCompactToHex(b *testing.B) {
testBytes := []byte{0, 15, 1, 12, 11, 8, 16 /*term*/}
for i := 0; i < b.N; i++ {
compactToHex(testBytes)
}
}

func BenchmarkKeybytesToHex(b *testing.B) {
testBytes := []byte{7, 6, 6, 5, 7, 2, 6, 2, 16}
for i := 0; i < b.N; i++ {
keybytesToHex(testBytes)
}
}

func BenchmarkHexToKeybytes(b *testing.B) {
testBytes := []byte{7, 6, 6, 5, 7, 2, 6, 2, 16}
for i := 0; i < b.N; i++ {
hexToKeybytes(testBytes)
}
}

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

package trie

import (
"fmt"

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

```
)

// MissingNodeError is returned by the trie functions (TryGet, TryUpdate, TryDelete)
// in the case where a trie node is not present in the local database. It contains
// information necessary for retrieving the missing node.
type MissingNodeError struct {
NodeHash common.Hash // hash of the missing node
Path     []byte      // hex-encoded path to the missing node
}

func (err *MissingNodeError) Error() string {
return fmt.Sprintf("missing trie node %x (path %x)", err.NodeHash, err.Path)
}

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

package trie

import (
"bytes"
"hash"
"sync"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto/sha3"
"github.com/ethereum/go-ethereum/rlp"
)

type hasher struct {
tmp                *bytes.Buffer
sha                hash.Hash
cachegen, cachelimit uint16
}

// hashers live in a global pool.
var hasherPool = sync.Pool{
New: func() interface{} {
return &hasher{tmp: new(bytes.Buffer), sha: sha3.NewKeccak256()}
},
}
```

```
func newHasher(cachegen, cachelimit uint16) *hasher {
h := hasherPool.Get().(*hasher)
h.cachegen, h.cachelimit = cachegen, cachelimit
return h
}

func returnHasherToPool(h *hasher) {
hasherPool.Put(h)
}

// hash collapses a node down into a hash node, also returning a copy of the
// original node initialized with the computed hash to replace the original one.
func (h *hasher) hash(n node, db DatabaseWriter, force bool) (node, node, error) {
// If we're not storing the node, just hashing, use available cached data
if hash, dirty := n.cache(); hash != nil {
if db == nil {
return hash, n, nil
}
if n.canUnload(h.cachegen, h.cachelimit) {
// Unload the node from cache. All of its subnodes will have a lower or equal
// cache generation number.
cacheUnloadCounter.Inc(1)
return hash, hash, nil
}
if !dirty {
return hash, n, nil
}
}
// Trie not processed yet or needs storage, walk the children
collapsed, cached, err := h.hashChildren(n, db)
if err != nil {
return hashNode{}, n, err
}
hashed, err := h.store(collapsed, db, force)
if err != nil {
return hashNode{}, n, err
}
// Cache the hash of the ndoe for later reuse and remove
// the dirty flag in commit mode. It's fine to assign these values directly
// without copying the node first because hashChildren copies it.
cachedHash, _ := hashed.(hashNode)
switch cn := cached.(type) {
```

```
case *shortNode:
cn.flags.hash = cachedHash
if db != nil {
cn.flags.dirty = false
}
case *fullNode:
cn.flags.hash = cachedHash
if db != nil {
cn.flags.dirty = false
}
}
return hashed, cached, nil
}

// hashChildren replaces the children of a node with their hashes if the encoded
// size of the child is larger than a hash, returning the collapsed node as well
// as a replacement for the original node with the child hashes cached in.
func (h *hasher) hashChildren(original node, db DatabaseWriter) (node, node, error) {
var err error

switch n := original.(type) {
case *shortNode:
// Hash the short node's child, caching the newly hashed subtree
collapsed, cached := n.copy(), n.copy()
collapsed.Key = hexToCompact(n.Key)
cached.Key = common.CopyBytes(n.Key)

if _, ok := n.Val.(valueNode); !ok {
collapsed.Val, cached.Val, err = h.hash(n.Val, db, false)
if err != nil {
return original, original, err
}
}
if collapsed.Val == nil {
collapsed.Val = valueNode(nil) // Ensure that nil children are encoded as empty strings.
}
return collapsed, cached, nil

case *fullNode:
// Hash the full node's children, caching the newly hashed subtrees
collapsed, cached := n.copy(), n.copy()
```

```
for i := 0; i < 16; i++ {
if n.Children[i] != nil {
collapsed.Children[i], cached.Children[i], err = h.hash(n.Children[i], db, false)
if err != nil {
return original, original, err
}
} else {
collapsed.Children[i] = valueNode(nil) // Ensure that nil children are encoded as empty strings.
}
}
cached.Children[16] = n.Children[16]
if collapsed.Children[16] == nil {
collapsed.Children[16] = valueNode(nil)
}
return collapsed, cached, nil

default:
// Value and hash nodes don't have children so they're left as were
return n, original, nil
}
}

func (h *hasher) store(n node, db DatabaseWriter, force bool) (node, error) {
// Don't store hashes or empty nodes.
if _, isHash := n.(hashNode); n == nil || isHash {
return n, nil
}
// Generate the RLP encoding of the node
h.tmp.Reset()
if err := rlp.Encode(h.tmp, n); err != nil {
panic("encode error: " + err.Error())
}

if h.tmp.Len() < 32 && !force {
return n, nil // Nodes smaller than 32 bytes are stored inside their parent
}
// Larger nodes are replaced by their hash and stored in the database.
hash, _ := n.cache()
if hash == nil {
h.sha.Reset()
h.sha.Write(h.tmp.Bytes())
hash = hashNode(h.sha.Sum(nil))
```

```
}
if db != nil {
return hash, db.Put(hash, h.tmp.Bytes())
}
return hash, nil
}

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

package trie

import (
"bytes"
"container/heap"
"errors"

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

// Iterator is a key-value trie iterator that traverses a Trie.
type Iterator struct {
nodeIt NodeIterator

Key   []byte // Current data key on which the iterator is positioned on
Value []byte // Current data value on which the iterator is positioned on
Err   error
}

// NewIterator creates a new key-value iterator from a node iterator
func NewIterator(it NodeIterator) *Iterator {
return &Iterator{
nodeIt: it,
}
}

// Next moves the iterator forward one key-value entry.
func (it *Iterator) Next() bool {
for it.nodeIt.Next(true) {
if it.nodeIt.Leaf() {
it.Key = it.nodeIt.LeafKey()
it.Value = it.nodeIt.LeafBlob()
return true
}
}
it.Key = nil
it.Value = nil
it.Err = it.nodeIt.Error()
return false
}

// NodeIterator is an iterator to traverse the trie pre-order.
type NodeIterator interface {
// Next moves the iterator to the next node. If the parameter is false, any child
// nodes will be skipped.
Next(bool) bool
// Error returns the error status of the iterator.
Error() error

// Hash returns the hash of the current node.
Hash() common.Hash
// Parent returns the hash of the parent of the current node. The hash may be the one
// grandparent if the immediate parent is an internal node with no hash.
Parent() common.Hash
// Path returns the hex-encoded path to the current node.
// Callers must not retain references to the return value after calling Next.
// For leaf nodes, the last element of the path is the 'terminator symbol' 0x10.
Path() []byte

// Leaf returns true iff the current node is a leaf node.
// LeafBlob, LeafKey return the contents and key of the leaf node. These
// method panic if the iterator is not positioned at a leaf.
// Callers must not retain references to their return value after calling Next
Leaf() bool
LeafBlob() []byte
LeafKey() []byte
}

// nodeIteratorState represents the iteration state at one particular node of the
// trie, which can be resumed at a later invocation.
type nodeIteratorState struct {
hash    common.Hash // Hash of the node being iterated (nil if not standalone)
node    node        // Trie node being iterated
parent  common.Hash // Hash of the first full ancestor node (nil if current is the root)
```

```
index   int         // Child to be processed next
pathlen int         // Length of the path to this node
}

type nodeIterator struct {
trie  *Trie                // Trie being iterated
stack []*nodeIteratorState // Hierarchy of trie nodes persisting the iteration state
path  []byte               // Path to the current node
err   error                // Failure set in case of an internal error in the iterator
}

// iteratorEnd is stored in nodeIterator.err when iteration is done.
var iteratorEnd = errors.New("end of iteration")

// seekError is stored in nodeIterator.err if the initial seek has failed.
type seekError struct {
key []byte
err error
}

func (e seekError) Error() string {
return "seek error: " + e.err.Error()
}

func newNodeIterator(trie *Trie, start []byte) NodeIterator {
if trie.Hash() == emptyState {
return new(nodeIterator)
}
it := &nodeIterator{trie: trie}
it.err = it.seek(start)
return it
}

func (it *nodeIterator) Hash() common.Hash {
if len(it.stack) == 0 {
return common.Hash{}
}
return it.stack[len(it.stack)-1].hash
}

func (it *nodeIterator) Parent() common.Hash {
if len(it.stack) == 0 {
```

```
return common.Hash{}
}
return it.stack[len(it.stack)-1].parent
}

func (it *nodeIterator) Leaf() bool {
return hasTerm(it.path)
}

func (it *nodeIterator) LeafBlob() []byte {
if len(it.stack) > 0 {
if node, ok := it.stack[len(it.stack)-1].node.(valueNode); ok {
return []byte(node)
}
}
panic("not at leaf")
}

func (it *nodeIterator) LeafKey() []byte {
if len(it.stack) > 0 {
if _, ok := it.stack[len(it.stack)-1].node.(valueNode); ok {
return hexToKeybytes(it.path)
}
}
panic("not at leaf")
}

func (it *nodeIterator) Path() []byte {
return it.path
}

func (it *nodeIterator) Error() error {
if it.err == iteratorEnd {
return nil
}
if seek, ok := it.err.(seekError); ok {
return seek.err
}
return it.err
}

// Next moves the iterator to the next node, returning whether there are any
```

```go
// further nodes. In case of an internal error this method returns false and
// sets the Error field to the encountered failure. If `descend` is false,
// skips iterating over any subnodes of the current node.
func (it *nodeIterator) Next(descend bool) bool {
if it.err == iteratorEnd {
return false
}
if seek, ok := it.err.(seekError); ok {
if it.err = it.seek(seek.key); it.err != nil {
return false
}
}
// Otherwise step forward with the iterator and report any errors.
state, parentIndex, path, err := it.peek(descend)
it.err = err
if it.err != nil {
return false
}
it.push(state, parentIndex, path)
return true
}

func (it *nodeIterator) seek(prefix []byte) error {
// The path we're looking for is the hex encoded key without terminator.
key := keybytesToHex(prefix)
key = key[:len(key)-1]
// Move forward until we're just before the closest match to key.
for {
state, parentIndex, path, err := it.peek(bytes.HasPrefix(key, it.path))
if err == iteratorEnd {
return iteratorEnd
} else if err != nil {
return seekError{prefix, err}
} else if bytes.Compare(path, key) >= 0 {
return nil
}
it.push(state, parentIndex, path)
}
}

// peek creates the next state of the iterator.
func (it *nodeIterator) peek(descend bool) (*nodeIteratorState, *int, []byte, error) {
```

```
if len(it.stack) == 0 {
// Initialize the iterator if we've just started.
root := it.trie.Hash()
state := &nodeIteratorState{node: it.trie.root, index: -1}
if root != emptyRoot {
state.hash = root
}
err := state.resolve(it.trie, nil)
return state, nil, nil, err
}
if !descend {
// If we're skipping children, pop the current node first
it.pop()
}

// Continue iteration to the next child
for len(it.stack) > 0 {
parent := it.stack[len(it.stack)-1]
ancestor := parent.hash
if (ancestor == common.Hash{}) {
ancestor = parent.parent
}
state, path, ok := it.nextChild(parent, ancestor)
if ok {
if err := state.resolve(it.trie, path); err != nil {
return parent, &parent.index, path, err
}
return state, &parent.index, path, nil
}
// No more child nodes, move back up.
it.pop()
}
return nil, nil, nil, iteratorEnd
}

func (st *nodeIteratorState) resolve(tr *Trie, path []byte) error {
if hash, ok := st.node.(hashNode); ok {
resolved, err := tr.resolveHash(hash, path)
if err != nil {
return err
}
st.node = resolved
st.hash = common.BytesToHash(hash)
}
return nil
}

func (it *nodeIterator) nextChild(parent *nodeIteratorState, ancestor common.Hash)
(*nodeIteratorState, []byte, bool) {
switch node := parent.node.(type) {
case *fullNode:
// Full node, move to the first non-nil child.
for i := parent.index + 1; i < len(node.Children); i++ {
child := node.Children[i]
if child != nil {
hash, _ := child.cache()
state := &nodeIteratorState{
hash:    common.BytesToHash(hash),
node:    child,
parent:  ancestor,
index:   -1,
pathlen: len(it.path),
}
path := append(it.path, byte(i))
parent.index = i - 1
return state, path, true
}
}
case *shortNode:
// Short node, return the pointer singleton child
if parent.index < 0 {
hash, _ := node.Val.cache()
state := &nodeIteratorState{
hash:    common.BytesToHash(hash),
node:    node.Val,
parent:  ancestor,
index:   -1,
pathlen: len(it.path),
}
path := append(it.path, node.Key...)
return state, path, true
}
}
return parent, it.path, false
```

```
}

func (it *nodeIterator) push(state *nodeIteratorState, parentIndex *int, path []byte) {
it.path = path
it.stack = append(it.stack, state)
if parentIndex != nil {
*parentIndex += 1
}
}

func (it *nodeIterator) pop() {
parent := it.stack[len(it.stack)-1]
it.path = it.path[:parent.pathlen]
it.stack = it.stack[:len(it.stack)-1]
}

func compareNodes(a, b NodeIterator) int {
if cmp := bytes.Compare(a.Path(), b.Path()); cmp != 0 {
return cmp
}
if a.Leaf() && !b.Leaf() {
return -1
} else if b.Leaf() && !a.Leaf() {
return 1
}
if cmp := bytes.Compare(a.Hash().Bytes(), b.Hash().Bytes()); cmp != 0 {
return cmp
}
if a.Leaf() && b.Leaf() {
return bytes.Compare(a.LeafBlob(), b.LeafBlob())
}
return 0
}

type differenceIterator struct {
a, b  NodeIterator // Nodes returned are those in b - a.
eof   bool         // Indicates a has run out of elements
count int          // Number of nodes scanned on either trie
}

// NewDifferenceIterator constructs a NodeIterator that iterates over elements in b that
// are not in a. Returns the iterator, and a pointer to an integer recording the number
// of nodes seen.
func NewDifferenceIterator(a, b NodeIterator) (NodeIterator, *int) {
a.Next(true)
it := &differenceIterator{
a: a,
b: b,
}
return it, &it.count
}

func (it *differenceIterator) Hash() common.Hash {
return it.b.Hash()
}

func (it *differenceIterator) Parent() common.Hash {
return it.b.Parent()
}

func (it *differenceIterator) Leaf() bool {
return it.b.Leaf()
}

func (it *differenceIterator) LeafBlob() []byte {
return it.b.LeafBlob()
}

func (it *differenceIterator) LeafKey() []byte {
return it.b.LeafKey()
}

func (it *differenceIterator) Path() []byte {
return it.b.Path()
}

func (it *differenceIterator) Next(bool) bool {
// Invariants:
// - We always advance at least one element in b.
// - At the start of this function, a's path is lexically greater than b's.
if !it.b.Next(true) {
return false
}
it.count += 1
if it.eof {
// a has reached eof, so we just return all elements from b
return true
}

for {
switch compareNodes(it.a, it.b) {
case -1:
// b jumped past a; advance a
if !it.a.Next(true) {
it.eof = true
return true
}
it.count += 1
case 1:
// b is before a
return true
case 0:
// a and b are identical; skip this whole subtree if the nodes have hashes
hasHash := it.a.Hash() == common.Hash{}
if !it.b.Next(hasHash) {
return false
}
it.count += 1
if !it.a.Next(hasHash) {
it.eof = true
return true
}
it.count += 1
}
}
}

func (it *differenceIterator) Error() error {
if err := it.a.Error(); err != nil {
return err
}
return it.b.Error()
}

type nodeIteratorHeap []NodeIterator
```

```
func (h nodeIteratorHeap) Len() int            { return len(h) }
func (h nodeIteratorHeap) Less(i, j int) bool  { return compareNodes(h[i], h[j]) < 0 }
func (h nodeIteratorHeap) Swap(i, j int)       { h[i], h[j] = h[j], h[i] }
func (h *nodeIteratorHeap) Push(x interface{}) { *h = append(*h, x.(NodeIterator)) }
func (h *nodeIteratorHeap) Pop() interface{} {
n := len(*h)
x := (*h)[n-1]
*h = (*h)[0 : n-1]
return x
}

type unionIterator struct {
items *nodeIteratorHeap // Nodes returned are the union of the ones in these iterators
count int               // Number of nodes scanned across all tries
}

// NewUnionIterator constructs a NodeIterator that iterates over elements in the union
// of the provided NodeIterators. Returns the iterator, and a pointer to an integer
// recording the number of nodes visited.
func NewUnionIterator(iters []NodeIterator) (NodeIterator, *int) {
h := make(nodeIteratorHeap, len(iters))
copy(h, iters)
heap.Init(&h)

ui := &unionIterator{items: &h}
return ui, &ui.count
}

func (it *unionIterator) Hash() common.Hash {
return (*it.items)[0].Hash()
}

func (it *unionIterator) Parent() common.Hash {
return (*it.items)[0].Parent()
}

func (it *unionIterator) Leaf() bool {
return (*it.items)[0].Leaf()
}

func (it *unionIterator) LeafBlob() []byte {
```

```
return (*it.items)[0].LeafBlob()
}

func (it *unionIterator) LeafKey() []byte {
return (*it.items)[0].LeafKey()
}

func (it *unionIterator) Path() []byte {
return (*it.items)[0].Path()
}

// Next returns the next node in the union of tries being iterated over.
//
// It does this by maintaining a heap of iterators, sorted by the iteration
// order of their next elements, with one entry for each source trie. Each
// time Next() is called, it takes the least element from the heap to return,
// advancing any other iterators that also point to that same element. These
// iterators are called with descend=false, since we know that any nodes under
// these nodes will also be duplicates, found in the currently selected iterator.
// Whenever an iterator is advanced, it is pushed back into the heap if it still
// has elements remaining.
//
// In the case that descend=false - eg, we're asked to ignore all subnodes of the
// current node - we also advance any iterators in the heap that have the current
// path as a prefix.
func (it *unionIterator) Next(descend bool) bool {
if len(*it.items) == 0 {
return false
}

// Get the next key from the union
least := heap.Pop(it.items).(NodeIterator)

// Skip over other nodes as long as they're identical, or, if we're not descending, as
// long as they have the same prefix as the current node.
for len(*it.items) > 0 && ((!descend && bytes.HasPrefix((*it.items)[0].Path(), least.Path())) ||
compareNodes(least, (*it.items)[0]) == 0) {
skipped := heap.Pop(it.items).(NodeIterator)
// Skip the whole subtree if the nodes have hashes; otherwise just skip this node
if skipped.Next(skipped.Hash() == common.Hash{}) {
it.count += 1
// If there are more elements, push the iterator back on the heap
heap.Push(it.items, skipped)
}
}

if least.Next(descend) {
it.count += 1
heap.Push(it.items, least)
}

return len(*it.items) > 0
}

func (it *unionIterator) Error() error {
for i := 0; i < len(*it.items); i++ {
if err := (*it.items)[i].Error(); err != nil {
return err
}
}
return nil
}

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

package trie

import (
"bytes"
"fmt"
"math/rand"
"testing"

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

func TestIterator(t *testing.T) {
trie := newEmpty()
vals := []struct{ k, v string }{
{"do", "verb"},
{"ether", "wookiedoo"},
{"horse", "stallion"},
{"shaman", "horse"},
{"doge", "coin"},
{"dog", "puppy"},
{"somethingveryoddindeedthis is", "myothernodedata"},
}
all := make(map[string]string)
for _, val := range vals {
all[val.k] = val.v
trie.Update([]byte(val.k), []byte(val.v))
}
trie.Commit()

found := make(map[string]string)
it := NewIterator(trie.NodeIterator(nil))
for it.Next() {
found[string(it.Key)] = string(it.Value)
}

for k, v := range all {
if found[k] != v {
t.Errorf("iterator value mismatch for %s: got %q want %q", k, found[k], v)
}
}
}

type kv struct {
k, v []byte
t    bool
}

func TestIteratorLargeData(t *testing.T) {
trie := newEmpty()
vals := make(map[string]*kv)

for i := byte(0); i < 255; i++ {
value := &kv{common.LeftPadBytes([]byte{i}, 32), []byte{i}, false}
value2 := &kv{common.LeftPadBytes([]byte{10, i}, 32), []byte{i}, false}
trie.Update(value.k, value.v)
trie.Update(value2.k, value2.v)
vals[string(value.k)] = value
vals[string(value2.k)] = value2
}
```

```
it := NewIterator(trie.NodeIterator(nil))
for it.Next() {
vals[string(it.Key)].t = true
}

var untouched []*kv
for _, value := range vals {
if !value.t {
untouched = append(untouched, value)
}
}

if len(untouched) > 0 {
t.Errorf("Missed %d nodes", len(untouched))
for _, value := range untouched {
t.Error(value)
}
}
}

// Tests that the node iterator indeed walks over the entire database contents.
func TestNodeIteratorCoverage(t *testing.T) {
// Create some arbitrary test trie to iterate
db, trie, _ := makeTestTrie()

// Gather all the node hashes found by the iterator
hashes := make(map[common.Hash]struct{})
for it := trie.NodeIterator(nil); it.Next(true); {
if it.Hash() != (common.Hash{}) {
hashes[it.Hash()] = struct{}{}
}
}
// Cross check the hashes and the database itself
for hash := range hashes {
if _, err := db.Get(hash.Bytes()); err != nil {
t.Errorf("failed to retrieve reported node %x: %v", hash, err)
}
}
for _, key := range db.(*ethdb.MemDatabase).Keys() {
if _, ok := hashes[common.BytesToHash(key)]; !ok {
t.Errorf("state entry not reported %x", key)
}
}
}

type kvs struct{ k, v string }

var testdata1 = []kvs{
{"barb", "ba"},
{"bard", "bc"},
{"bars", "bb"},
{"bar", "b"},
{"fab", "z"},
{"food", "ab"},
{"foos", "aa"},
{"foo", "a"},
}

var testdata2 = []kvs{
{"aardvark", "c"},
{"bar", "b"},
{"barb", "bd"},
{"bars", "be"},
{"fab", "z"},
{"foo", "a"},
{"foos", "aa"},
{"food", "ab"},
{"jars", "d"},
}

func TestIteratorSeek(t *testing.T) {
trie := newEmpty()
for _, val := range testdata1 {
trie.Update([]byte(val.k), []byte(val.v))
}

// Seek to the middle.
it := NewIterator(trie.NodeIterator([]byte("fab")))
if err := checkIteratorOrder(testdata1[4:], it); err != nil {
t.Fatal(err)
}

// Seek to a non-existent key.
```

```
it = NewIterator(trie.NodeIterator([]byte("barc")))
if err := checkIteratorOrder(testdata1[1:], it); err != nil {
t.Fatal(err)
}

// Seek beyond the end.
it = NewIterator(trie.NodeIterator([]byte("z")))
if err := checkIteratorOrder(nil, it); err != nil {
t.Fatal(err)
}
}

func checkIteratorOrder(want []kvs, it *Iterator) error {
for it.Next() {
if len(want) == 0 {
return fmt.Errorf("didn't expect any more values, got key %q", it.Key)
}
if !bytes.Equal(it.Key, []byte(want[0].k)) {
return fmt.Errorf("wrong key: got %q, want %q", it.Key, want[0].k)
}
want = want[1:]
}
if len(want) > 0 {
return fmt.Errorf("iterator ended early, want key %q", want[0])
}
return nil
}

func TestDifferenceIterator(t *testing.T) {
triea := newEmpty()
for _, val := range testdata1 {
triea.Update([]byte(val.k), []byte(val.v))
}
triea.Commit()

trieb := newEmpty()
for _, val := range testdata2 {
trieb.Update([]byte(val.k), []byte(val.v))
}
trieb.Commit()

found := make(map[string]string)
```

```
di, _ := NewDifferenceIterator(triea.NodeIterator(nil), trieb.NodeIterator(nil))
it := NewIterator(di)
for it.Next() {
found[string(it.Key)] = string(it.Value)
}

all := []struct{ k, v string }{
{"aardvark", "c"},
{"barb", "bd"},
{"bars", "be"},
{"jars", "d"},
}
for _, item := range all {
if found[item.k] != item.v {
t.Errorf("iterator value mismatch for %s: got %v want %v", item.k, found[item.k], item.v)
}
}
if len(found) != len(all) {
t.Errorf("iterator count mismatch: got %d values, want %d", len(found), len(all))
}
}

func TestUnionIterator(t *testing.T) {
triea := newEmpty()
for _, val := range testdata1 {
triea.Update([]byte(val.k), []byte(val.v))
}
triea.Commit()

trieb := newEmpty()
for _, val := range testdata2 {
trieb.Update([]byte(val.k), []byte(val.v))
}
trieb.Commit()

di, _ := NewUnionIterator([]NodeIterator{triea.NodeIterator(nil), trieb.NodeIterator(nil)})
it := NewIterator(di)

all := []struct{ k, v string }{
{"aardvark", "c"},
{"barb", "ba"},
{"barb", "bd"},
{"bard", "bc"},
{"bars", "bb"},
{"bars", "be"},
{"bar", "b"},
{"fab", "z"},
{"food", "ab"},
{"foos", "aa"},
{"foo", "a"},
{"jars", "d"},
}

for i, kv := range all {
if !it.Next() {
t.Errorf("Iterator ends prematurely at element %d", i)
}
if kv.k != string(it.Key) {
t.Errorf("iterator value mismatch for element %d: got key %s want %s", i, it.Key, kv.k)
}
if kv.v != string(it.Value) {
t.Errorf("iterator value mismatch for element %d: got value %s want %s", i, it.Value, kv.v)
}
}
if it.Next() {
t.Errorf("Iterator returned extra values.")
}
}

func TestIteratorNoDups(t *testing.T) {
var tr Trie
for _, val := range testdata1 {
tr.Update([]byte(val.k), []byte(val.v))
}
checkIteratorNoDups(t, tr.NodeIterator(nil), nil)
}

// This test checks that nodeIterator.Next can be retried after inserting missing trie nodes.
func TestIteratorContinueAfterError(t *testing.T) {
db, _ := ethdb.NewMemDatabase()
tr, _ := New(common.Hash{}, db)
for _, val := range testdata1 {
tr.Update([]byte(val.k), []byte(val.v))
}
```

```
tr.Commit()
wantNodeCount := checkIteratorNoDups(t, tr.NodeIterator(nil), nil)
keys := db.Keys()
t.Log("node count", wantNodeCount)

for i := 0; i < 20; i++ {
// Create trie that will load all nodes from DB.
tr, _ := New(tr.Hash(), db)

// Remove a random node from the database. It can't be the root node
// because that one is already loaded.
var rkey []byte
for {
if rkey = keys[rand.Intn(len(keys))]; !bytes.Equal(rkey, tr.Hash().Bytes()) {
break
}
}
rval, _ := db.Get(rkey)
db.Delete(rkey)

// Iterate until the error is hit.
seen := make(map[string]bool)
it := tr.NodeIterator(nil)
checkIteratorNoDups(t, it, seen)
missing, ok := it.Error().(*MissingNodeError)
if !ok || !bytes.Equal(missing.NodeHash[:], rkey) {
t.Fatal("didn't hit missing node, got", it.Error())
}

// Add the node back and continue iteration.
db.Put(rkey, rval)
checkIteratorNoDups(t, it, seen)
if it.Error() != nil {
t.Fatal("unexpected error", it.Error())
}
if len(seen) != wantNodeCount {
t.Fatal("wrong node iteration count, got", len(seen), "want", wantNodeCount)
}
}
}

// Similar to the test above, this one checks that failure to create nodeIterator at a
// certain key prefix behaves correctly when Next is called. The expectation is that Next
// should retry seeking before returning true for the first time.
func TestIteratorContinueAfterSeekError(t *testing.T) {
// Commit test trie to db, then remove the node containing "bars".
db, _ := ethdb.NewMemDatabase()
ctr, _ := New(common.Hash{}, db)
for _, val := range testdata1 {
ctr.Update([]byte(val.k), []byte(val.v))
}
root, _ := ctr.Commit()
barNodeHash :=
common.HexToHash("05041990364eb72fcb1127652ce40d8bab765f2bfe53225b1170d276cc101c
2e")
barNode, _ := db.Get(barNodeHash[:])
db.Delete(barNodeHash[:])

// Create a new iterator that seeks to "bars". Seeking can't proceed because
// the node is missing.
tr, _ := New(root, db)
it := tr.NodeIterator([]byte("bars"))
missing, ok := it.Error().(*MissingNodeError)
if !ok {
t.Fatal("want MissingNodeError, got", it.Error())
} else if missing.NodeHash != barNodeHash {
t.Fatal("wrong node missing")
}

// Reinsert the missing node.
db.Put(barNodeHash[:], barNode[:])

// Check that iteration produces the right set of values.
if err := checkIteratorOrder(testdata1[2:], NewIterator(it)); err != nil {
t.Fatal(err)
}
}

func checkIteratorNoDups(t *testing.T, it NodeIterator, seen map[string]bool) int {
if seen == nil {
seen = make(map[string]bool)
}
for it.Next(true) {
if seen[string(it.Path())] {
```

```
t.Fatalf("iterator visited node path %x twice", it.Path())
}
seen[string(it.Path())] = true
}
return len(seen)
}

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

package trie

import (
"fmt"
"io"
"strings"

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

var indices = []string{"0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "a", "b", "c", "d", "e", "f", "[17]"}

type node interface {
fstring(string) string
cache() (hashNode, bool)
canUnload(cachegen, cachelimit uint16) bool
}

type (
fullNode struct {
Children [17]node // Actual trie node data to encode/decode (needs custom encoder)
flags    nodeFlag
}
shortNode struct {
Key   []byte
Val   node
flags nodeFlag
}
hashNode  []byte
valueNode []byte
)
```

```
// EncodeRLP encodes a full node into the consensus RLP format.
func (n *fullNode) EncodeRLP(w io.Writer) error {
return rlp.Encode(w, n.Children)
}

func (n *fullNode) copy() *fullNode   { copy := *n; return &copy }
func (n *shortNode) copy() *shortNode { copy := *n; return &copy }

// nodeFlag contains caching-related metadata about a node.
type nodeFlag struct {
hash  hashNode // cached hash of the node (may be nil)
gen   uint16   // cache generation counter
dirty bool     // whether the node has changes that must be written to the database
}

// canUnload tells whether a node can be unloaded.
func (n *nodeFlag) canUnload(cachegen, cachelimit uint16) bool {
return !n.dirty && cachegen-n.gen >= cachelimit
}

func (n *fullNode) canUnload(gen, limit uint16) bool  { return n.flags.canUnload(gen, limit) }
func (n *shortNode) canUnload(gen, limit uint16) bool { return n.flags.canUnload(gen, limit) }
func (n hashNode) canUnload(uint16, uint16) bool      { return false }
func (n valueNode) canUnload(uint16, uint16) bool     { return false }

func (n *fullNode) cache() (hashNode, bool)  { return n.flags.hash, n.flags.dirty }
func (n *shortNode) cache() (hashNode, bool) { return n.flags.hash, n.flags.dirty }
func (n hashNode) cache() (hashNode, bool)   { return nil, true }
func (n valueNode) cache() (hashNode, bool)  { return nil, true }

// Pretty printing.
func (n *fullNode) String() string  { return n.fstring("") }
func (n *shortNode) String() string { return n.fstring("") }
func (n hashNode) String() string   { return n.fstring("") }
func (n valueNode) String() string  { return n.fstring("") }

func (n *fullNode) fstring(ind string) string {
resp := fmt.Sprintf("[\n%s  ", ind)
for i, node := range n.Children {
if node == nil {
resp += fmt.Sprintf("%s: <nil> ", indices[i])
} else {
resp += fmt.Sprintf("%s: %v", indices[i], node.fstring(ind+"  "))
}
}
return resp + fmt.Sprintf("\n%s] ", ind)
}
func (n *shortNode) fstring(ind string) string {
return fmt.Sprintf("{%x: %v} ", n.Key, n.Val.fstring(ind+"  "))
}
func (n hashNode) fstring(ind string) string {
return fmt.Sprintf("<%x> ", []byte(n))
}
func (n valueNode) fstring(ind string) string {
return fmt.Sprintf("%x ", []byte(n))
}

func mustDecodeNode(hash, buf []byte, cachegen uint16) node {
n, err := decodeNode(hash, buf, cachegen)
if err != nil {
panic(fmt.Sprintf("node %x: %v", hash, err))
}
return n
}

// decodeNode parses the RLP encoding of a trie node.
func decodeNode(hash, buf []byte, cachegen uint16) (node, error) {
if len(buf) == 0 {
return nil, io.ErrUnexpectedEOF
}
elems, _, err := rlp.SplitList(buf)
if err != nil {
return nil, fmt.Errorf("decode error: %v", err)
}
switch c, _ := rlp.CountValues(elems); c {
case 2:
n, err := decodeShort(hash, buf, elems, cachegen)
return n, wrapError(err, "short")
case 17:
n, err := decodeFull(hash, buf, elems, cachegen)
return n, wrapError(err, "full")
default:
return nil, fmt.Errorf("invalid number of list elements: %v", c)
```

```
}
}

func decodeShort(hash, buf, elems []byte, cachegen uint16) (node, error) {
kbuf, rest, err := rlp.SplitString(elems)
if err != nil {
return nil, err
}
flag := nodeFlag{hash: hash, gen: cachegen}
key := compactToHex(kbuf)
if hasTerm(key) {
// value node
val, _, err := rlp.SplitString(rest)
if err != nil {
return nil, fmt.Errorf("invalid value node: %v", err)
}
return &shortNode{key, append(valueNode{}, val...), flag}, nil
}
r, _, err := decodeRef(rest, cachegen)
if err != nil {
return nil, wrapError(err, "val")
}
return &shortNode{key, r, flag}, nil
}

func decodeFull(hash, buf, elems []byte, cachegen uint16) (*fullNode, error) {
n := &fullNode{flags: nodeFlag{hash: hash, gen: cachegen}}
for i := 0; i < 16; i++ {
cld, rest, err := decodeRef(elems, cachegen)
if err != nil {
return n, wrapError(err, fmt.Sprintf("[%d]", i))
}
n.Children[i], elems = cld, rest
}
val, _, err := rlp.SplitString(elems)
if err != nil {
return n, err
}
if len(val) > 0 {
n.Children[16] = append(valueNode{}, val...)
}
return n, nil
```

```
}

const hashLen = len(common.Hash{})

func decodeRef(buf []byte, cachegen uint16) (node, []byte, error) {
kind, val, rest, err := rlp.Split(buf)
if err != nil {
return nil, buf, err
}
switch {
case kind == rlp.List:
// 'embedded' node reference. The encoding must be smaller
// than a hash in order to be valid.
if size := len(buf) - len(rest); size > hashLen {
err := fmt.Errorf("oversized embedded node (size is %d bytes, want size < %d)", size, hashLen)
return nil, buf, err
}
n, err := decodeNode(nil, buf, cachegen)
return n, rest, err
case kind == rlp.String && len(val) == 0:
// empty node
return nil, rest, nil
case kind == rlp.String && len(val) == 32:
return append(hashNode{}, val...), rest, nil
default:
return nil, nil, fmt.Errorf("invalid RLP string size %d (want 0 or 32)", len(val))
}
}

// wraps a decoding error with information about the path to the
// invalid child node (for debugging encoding issues).
type decodeError struct {
what  error
stack []string
}

func wrapError(err error, ctx string) error {
if err == nil {
return nil
}
if decErr, ok := err.(*decodeError); ok {
decErr.stack = append(decErr.stack, ctx)
```

```
return decErr
}
return &decodeError{err, []string{ctx}}
}

func (err *decodeError) Error() string {
return fmt.Sprintf("%v (decode path: %s)", err.what, strings.Join(err.stack, "<-"))
}

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

package trie

import "testing"

func TestCanUnload(t *testing.T) {
tests := []struct {
flag                 nodeFlag
cachegen, cachelimit uint16
want                 bool
}{
{
flag: nodeFlag{dirty: true, gen: 0},
want: false,
},
{
flag:     nodeFlag{dirty: false, gen: 0},
cachegen: 0, cachelimit: 0,
want: true,
},
{
flag:     nodeFlag{dirty: false, gen: 65534},
cachegen: 65535, cachelimit: 1,
want: true,
},
{
flag:     nodeFlag{dirty: false, gen: 65534},
cachegen: 0, cachelimit: 1,
want: true,
},
{
```

```
flag:     nodeFlag{dirty: false, gen: 1},
cachegen: 65535, cachelimit: 1,
want: true,
},
}

for _, test := range tests {
if got := test.flag.canUnload(test.cachegen, test.cachelimit); got != test.want {
t.Errorf("%+v\n   got %t, want %t", test, got, test.want)
}
}
}

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

package trie

import (
"bytes"
"errors"
"fmt"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto/sha3"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/rlp"
)

// Prove constructs a merkle proof for key. The result contains all
// encoded nodes on the path to the value at key. The value itself is
// also included in the last node and can be retrieved by verifying
// the proof.
//
// If the trie does not contain a value for key, the returned proof
// contains all nodes of the longest existing prefix of the key
// (at least the root node), ending with the node that proves the
// absence of the key.
func (t *Trie) Prove(key []byte) []rlp.RawValue {
// Collect all nodes on the path to key.
key = keybytesToHex(key)
nodes := []node{}
```

```
tn := t.root
for len(key) > 0 && tn != nil {
switch n := tn.(type) {
case *shortNode:
if len(key) < len(n.Key) || !bytes.Equal(n.Key, key[:len(n.Key)]) {
// The trie doesn't contain the key.
tn = nil
} else {
tn = n.Val
key = key[len(n.Key):]
}
nodes = append(nodes, n)
case *fullNode:
tn = n.Children[key[0]]
key = key[1:]
nodes = append(nodes, n)
case hashNode:
var err error
tn, err = t.resolveHash(n, nil)
if err != nil {
log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
return nil
}
default:
panic(fmt.Sprintf("%T: invalid node: %v", tn, tn))
}
}
hasher := newHasher(0, 0)
proof := make([]rlp.RawValue, 0, len(nodes))
for i, n := range nodes {
// Don't bother checking for errors here since hasher panics
// if encoding doesn't work and we're not writing to any database.
n, _, _ = hasher.hashChildren(n, nil)
hn, _ := hasher.store(n, nil, false)
if _, ok := hn.(hashNode); ok || i == 0 {
// If the node's database encoding is a hash (or is the
// root node), it becomes a proof element.
enc, _ := rlp.EncodeToBytes(n)
proof = append(proof, enc)
}
}
return proof
```

```
}

// VerifyProof checks merkle proofs. The given proof must contain the
// value for key in a trie with the given root hash. VerifyProof
// returns an error if the proof contains invalid trie nodes or the
// wrong value.
func VerifyProof(rootHash common.Hash, key []byte, proof []rlp.RawValue) (value []byte, err error)
{
key = keybytesToHex(key)
sha := sha3.NewKeccak256()
wantHash := rootHash.Bytes()
for i, buf := range proof {
sha.Reset()
sha.Write(buf)
if !bytes.Equal(sha.Sum(nil), wantHash) {
return nil, fmt.Errorf("bad proof node %d: hash mismatch", i)
}
n, err := decodeNode(wantHash, buf, 0)
if err != nil {
return nil, fmt.Errorf("bad proof node %d: %v", i, err)
}
keyrest, cld := get(n, key)
switch cld := cld.(type) {
case nil:
if i != len(proof)-1 {
return nil, fmt.Errorf("key mismatch at proof node %d", i)
} else {
// The trie doesn't contain the key.
return nil, nil
}
case hashNode:
key = keyrest
wantHash = cld
case valueNode:
if i != len(proof)-1 {
return nil, errors.New("additional nodes at end of proof")
}
return cld, nil
}
}
return nil, errors.New("unexpected end of proof")
}
```

```
func get(tn node, key []byte) ([]byte, node) {
for {
switch n := tn.(type) {
case *shortNode:
if len(key) < len(n.Key) || !bytes.Equal(n.Key, key[:len(n.Key)]) {
return nil, nil
}
tn = n.Val
key = key[len(n.Key):]
case *fullNode:
tn = n.Children[key[0]]
key = key[1:]
case hashNode:
return key, n
case nil:
return key, nil
case valueNode:
return nil, n
default:
panic(fmt.Sprintf("%T: invalid node: %v", tn, tn))
}
}
}
```

87:F:\git\coin\ethereum\go-ethereum\trie\proof_test.go

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

package trie

import (
"bytes"
crand "crypto/rand"
mrand "math/rand"
"testing"
"time"

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

func init() {
```

```
mrand.Seed(time.Now().Unix())
}

func TestProof(t *testing.T) {
trie, vals := randomTrie(500)
root := trie.Hash()
for _, kv := range vals {
proof := trie.Prove(kv.k)
if proof == nil {
t.Fatalf("missing key %x while constructing proof", kv.k)
}
val, err := VerifyProof(root, kv.k, proof)
if err != nil {
t.Fatalf("VerifyProof error for key %x: %v\nraw proof: %x", kv.k, err, proof)
}
if !bytes.Equal(val, kv.v) {
t.Fatalf("VerifyProof returned wrong value for key %x: got %x, want %x", kv.k, val, kv.v)
}
}
}

func TestOneElementProof(t *testing.T) {
trie := new(Trie)
updateString(trie, "k", "v")
proof := trie.Prove([]byte("k"))
if proof == nil {
t.Fatal("nil proof")
}
if len(proof) != 1 {
t.Error("proof should have one element")
}
val, err := VerifyProof(trie.Hash(), []byte("k"), proof)
if err != nil {
t.Fatalf("VerifyProof error: %v\nraw proof: %x", err, proof)
}
if !bytes.Equal(val, []byte("v")) {
t.Fatalf("VerifyProof returned wrong value: got %x, want 'k'", val)
}
}

func TestVerifyBadProof(t *testing.T) {
trie, vals := randomTrie(800)
```

```
root := trie.Hash()
for _, kv := range vals {
proof := trie.Prove(kv.k)
if proof == nil {
t.Fatal("nil proof")
}
mutateByte(proof[mrand.Intn(len(proof))])
if _, err := VerifyProof(root, kv.k, proof); err == nil {
t.Fatalf("expected proof to fail for key %x", kv.k)
}
}
}

// mutateByte changes one byte in b.
func mutateByte(b []byte) {
for r := mrand.Intn(len(b)); ; {
new := byte(mrand.Intn(255))
if new != b[r] {
b[r] = new
break
}
}
}

func BenchmarkProve(b *testing.B) {
trie, vals := randomTrie(100)
var keys []string
for k := range vals {
keys = append(keys, k)
}

b.ResetTimer()
for i := 0; i < b.N; i++ {
kv := vals[keys[i%len(keys)]]
if trie.Prove(kv.k) == nil {
b.Fatalf("nil proof for %x", kv.k)
}
}
}

func BenchmarkVerifyProof(b *testing.B) {
trie, vals := randomTrie(100)
```

```
root := trie.Hash()
var keys []string
var proofs [][]rlp.RawValue
for k := range vals {
keys = append(keys, k)
proofs = append(proofs, trie.Prove([]byte(k)))
}

b.ResetTimer()
for i := 0; i < b.N; i++ {
im := i % len(keys)
if _, err := VerifyProof(root, []byte(keys[im]), proofs[im]); err != nil {
b.Fatalf("key %x: %v", keys[im], err)
}
}
}

func randomTrie(n int) (*Trie, map[string]*kv) {
trie := new(Trie)
vals := make(map[string]*kv)
for i := byte(0); i < 100; i++ {
value := &kv{common.LeftPadBytes([]byte{i}, 32), []byte{i}, false}
value2 := &kv{common.LeftPadBytes([]byte{i + 10}, 32), []byte{i}, false}
trie.Update(value.k, value.v)
trie.Update(value2.k, value2.v)
vals[string(value.k)] = value
vals[string(value2.k)] = value2
}
for i := 0; i < n; i++ {
value := &kv{randBytes(32), randBytes(20), false}
trie.Update(value.k, value.v)
vals[string(value.k)] = value
}
return trie, vals
}

func randBytes(n int) []byte {
r := make([]byte, n)
crand.Read(r)
return r
}
```

88:F:\git\coin\ethereum\go-ethereum\trie\secure_trie.go

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

package trie

import (
"fmt"

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

var secureKeyPrefix = []byte("secure-key-")

const secureKeyLength = 11 + 32 // Length of the above prefix + 32byte hash

// SecureTrie wraps a trie with key hashing. In a secure trie, all
// access operations hash the key using keccak256. This prevents
// calling code from creating long chains of nodes that
// increase the access time.
//
// Contrary to a regular trie, a SecureTrie can only be created with
// New and must have an attached database. The database also stores
// the preimage of each key.
//
// SecureTrie is not safe for concurrent use.
type SecureTrie struct {
trie             Trie
hashKeyBuf       [secureKeyLength]byte
secKeyBuf        [200]byte
secKeyCache      map[string][]byte
secKeyCacheOwner *SecureTrie // Pointer to self, replace the key cache on mismatch
}

// NewSecure creates a trie with an existing root node from db.
//
// If root is the zero hash or the sha3 hash of an empty string, the
// trie is initially empty. Otherwise, New will panic if db is nil
// and returns MissingNodeError if the root node cannot be found.
//
// Accessing the trie loads nodes from db on demand.
// Loaded nodes are kept around until their 'cache generation' expires.
// A new cache generation is created by each call to Commit.
// cachelimit sets the number of past cache generations to keep.
func NewSecure(root common.Hash, db Database, cachelimit uint16) (*SecureTrie, error) {
if db == nil {
panic("NewSecure called with nil database")
}
trie, err := New(root, db)
if err != nil {
return nil, err
}
trie.SetCacheLimit(cachelimit)
return &SecureTrie{trie: *trie}, nil
}

// Get returns the value for key stored in the trie.
// The value bytes must not be modified by the caller.
func (t *SecureTrie) Get(key []byte) []byte {
res, err := t.TryGet(key)
if err != nil {
log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
}
return res
}

// TryGet returns the value for key stored in the trie.
// The value bytes must not be modified by the caller.
// If a node was not found in the database, a MissingNodeError is returned.
func (t *SecureTrie) TryGet(key []byte) ([]byte, error) {
return t.trie.TryGet(t.hashKey(key))
}

// Update associates key with value in the trie. Subsequent calls to
// Get will return value. If value has length zero, any existing value
// is deleted from the trie and calls to Get will return nil.
//
// The value bytes must not be modified by the caller while they are
// stored in the trie.
func (t *SecureTrie) Update(key, value []byte) {
if err := t.TryUpdate(key, value); err != nil {
log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
}
}
```

```
// TryUpdate associates key with value in the trie. Subsequent calls to
// Get will return value. If value has length zero, any existing value
// is deleted from the trie and calls to Get will return nil.
//
// The value bytes must not be modified by the caller while they are
// stored in the trie.
//
// If a node was not found in the database, a MissingNodeError is returned.
func (t *SecureTrie) TryUpdate(key, value []byte) error {
hk := t.hashKey(key)
err := t.trie.TryUpdate(hk, value)
if err != nil {
return err
}
t.getSecKeyCache()[string(hk)] = common.CopyBytes(key)
return nil
}

// Delete removes any existing value for key from the trie.
func (t *SecureTrie) Delete(key []byte) {
if err := t.TryDelete(key); err != nil {
log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
}
}

// TryDelete removes any existing value for key from the trie.
// If a node was not found in the database, a MissingNodeError is returned.
func (t *SecureTrie) TryDelete(key []byte) error {
hk := t.hashKey(key)
delete(t.getSecKeyCache(), string(hk))
return t.trie.TryDelete(hk)
}

// GetKey returns the sha3 preimage of a hashed key that was
// previously used to store a value.
func (t *SecureTrie) GetKey(shaKey []byte) []byte {
if key, ok := t.getSecKeyCache()[string(shaKey)]; ok {
return key
}
key, _ := t.trie.db.Get(t.secKey(shaKey))
return key
```

```
}

// Commit writes all nodes and the secure hash pre-images to the trie's database.
// Nodes are stored with their sha3 hash as the key.
//
// Committing flushes nodes from memory. Subsequent Get calls will load nodes
// from the database.
func (t *SecureTrie) Commit() (root common.Hash, err error) {
return t.CommitTo(t.trie.db)
}

func (t *SecureTrie) Hash() common.Hash {
return t.trie.Hash()
}

func (t *SecureTrie) Root() []byte {
return t.trie.Root()
}

func (t *SecureTrie) Copy() *SecureTrie {
cpy := *t
return &cpy
}

// NodeIterator returns an iterator that returns nodes of the underlying trie. Iteration
// starts at the key after the given start key.
func (t *SecureTrie) NodeIterator(start []byte) NodeIterator {
return t.trie.NodeIterator(start)
}

// CommitTo writes all nodes and the secure hash pre-images to the given database.
// Nodes are stored with their sha3 hash as the key.
//
// Committing flushes nodes from memory. Subsequent Get calls will load nodes from
// the trie's database. Calling code must ensure that the changes made to db are
// written back to the trie's attached database before using the trie.
func (t *SecureTrie) CommitTo(db DatabaseWriter) (root common.Hash, err error) {
if len(t.getSecKeyCache()) > 0 {
for hk, key := range t.secKeyCache {
if err := db.Put(t.secKey([]byte(hk)), key); err != nil {
return common.Hash{}, err
}
}
t.secKeyCache = make(map[string][]byte)
}
return t.trie.CommitTo(db)
}

// secKey returns the database key for the preimage of key, as an ephemeral buffer.
// The caller must not hold onto the return value because it will become
// invalid on the next call to hashKey or secKey.
func (t *SecureTrie) secKey(key []byte) []byte {
buf := append(t.secKeyBuf[:0], secureKeyPrefix...)
buf = append(buf, key...)
return buf
}

// hashKey returns the hash of key as an ephemeral buffer.
// The caller must not hold onto the return value because it will become
// invalid on the next call to hashKey or secKey.
func (t *SecureTrie) hashKey(key []byte) []byte {
h := newHasher(0, 0)
h.sha.Reset()
h.sha.Write(key)
buf := h.sha.Sum(t.hashKeyBuf[:0])
returnHasherToPool(h)
return buf
}

// getSecKeyCache returns the current secure key cache, creating a new one if
// ownership changed (i.e. the current secure trie is a copy of another owning
// the actual cache).
func (t *SecureTrie) getSecKeyCache() map[string][]byte {
if t != t.secKeyCacheOwner {
t.secKeyCacheOwner = t
t.secKeyCache = make(map[string][]byte)
}
return t.secKeyCache
}
```

89:F:\git\coin\ethereum\go-ethereum\trie\secure_trie_test.go

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

package trie
```

```
import (
"bytes"
"runtime"
"sync"
"testing"

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

func newEmptySecure() *SecureTrie {
db, _ := ethdb.NewMemDatabase()
trie, _ := NewSecure(common.Hash{}, db, 0)
return trie
}

// makeTestSecureTrie creates a large enough secure trie for testing.
func makeTestSecureTrie() (ethdb.Database, *SecureTrie, map[string][]byte) {
// Create an empty trie
db, _ := ethdb.NewMemDatabase()
trie, _ := NewSecure(common.Hash{}, db, 0)

// Fill it with some arbitrary data
content := make(map[string][]byte)
for i := byte(0); i < 255; i++ {
// Map the same data under multiple keys
key, val := common.LeftPadBytes([]byte{1, i}, 32), []byte{i}
content[string(key)] = val
trie.Update(key, val)

key, val = common.LeftPadBytes([]byte{2, i}, 32), []byte{i}
content[string(key)] = val
trie.Update(key, val)

// Add some other data to inflate the trie
for j := byte(3); j < 13; j++ {
key, val = common.LeftPadBytes([]byte{j, i}, 32), []byte{j, i}
content[string(key)] = val
trie.Update(key, val)
}
}
trie.Commit()

// Return the generated trie
return db, trie, content
}

func TestSecureDelete(t *testing.T) {
trie := newEmptySecure()
vals := []struct{ k, v string }{
{"do", "verb"},
{"ether", "wookiedoo"},
{"horse", "stallion"},
{"shaman", "horse"},
{"doge", "coin"},
{"ether", ""},
{"dog", "puppy"},
{"shaman", ""},
}
for _, val := range vals {
if val.v != "" {
trie.Update([]byte(val.k), []byte(val.v))
} else {
trie.Delete([]byte(val.k))
}
}
hash := trie.Hash()
exp :=
common.HexToHash("29b235a58c3c25ab83010c327d5932bcf05324b7d6b1185e650798034783c
a9d")
if hash != exp {
t.Errorf("expected %x got %x", exp, hash)
}
}

func TestSecureGetKey(t *testing.T) {
trie := newEmptySecure()
trie.Update([]byte("foo"), []byte("bar"))

key := []byte("foo")
value := []byte("bar")
seckey := crypto.Keccak256(key)
```

```
if !bytes.Equal(trie.Get(key), value) {
t.Errorf("Get did not return bar")
}
if k := trie.GetKey(seckey); !bytes.Equal(k, key) {
t.Errorf("GetKey returned %q, want %q", k, key)
}
}

func TestSecureTrieConcurrency(t *testing.T) {
// Create an initial trie and copy if for concurrent access
_, trie, _ := makeTestSecureTrie()

threads := runtime.NumCPU()
tries := make([]*SecureTrie, threads)
for i := 0; i < threads; i++ {
cpy := *trie
tries[i] = &cpy
}
// Start a batch of goroutines interactng with the trie
pend := new(sync.WaitGroup)
pend.Add(threads)
for i := 0; i < threads; i++ {
go func(index int) {
defer pend.Done()

for j := byte(0); j < 255; j++ {
// Map the same data under multiple keys
key, val := common.LeftPadBytes([]byte{byte(index), 1, j}, 32), []byte{j}
tries[index].Update(key, val)

key, val = common.LeftPadBytes([]byte{byte(index), 2, j}, 32), []byte{j}
tries[index].Update(key, val)

// Add some other data to inflate the trie
for k := byte(3); k < 13; k++ {
key, val = common.LeftPadBytes([]byte{byte(index), k, j}, 32), []byte{k, j}
tries[index].Update(key, val)
}
}
tries[index].Commit()
}(i)
}
// Wait for all threads to finish
pend.Wait()
}

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

package trie

import (
"errors"
"fmt"

"github.com/ethereum/go-ethereum/common"
"gopkg.in/karalabe/cookiejar.v2/collections/prque"
)

// ErrNotRequested is returned by the trie sync when it's requested to process a
// node it did not request.
var ErrNotRequested = errors.New("not requested")

// ErrAlreadyProcessed is returned by the trie sync when it's requested to process a
// node it already processed previously.
var ErrAlreadyProcessed = errors.New("already processed")

// request represents a scheduled or already in-flight state retrieval request.
type request struct {
hash common.Hash // Hash of the node data content to retrieve
data []byte      // Data content of the node, cached until all subtrees complete
raw  bool        // Whether this is a raw entry (code) or a trie node

parents []*request // Parent state nodes referencing this entry (notify all upon completion)
depth   int        // Depth level within the trie the node is located to prioritise DFS
deps    int        // Number of dependencies before allowed to commit this node

callback TrieSyncLeafCallback // Callback to invoke if a leaf node it reached on this branch
}

// SyncResult is a simple list to return missing nodes along with their request
// hashes.
type SyncResult struct {
```

```
Hash common.Hash // Hash of the originally unknown trie node
Data []byte      // Data content of the retrieved node
}

// syncMemBatch is an in-memory buffer of successfully downloaded but not yet
// persisted data items.
type syncMemBatch struct {
batch map[common.Hash][]byte // In-memory membatch of recently ocmpleted items
order []common.Hash          // Order of completion to prevent out-of-order data loss
}

// newSyncMemBatch allocates a new memory-buffer for not-yet persisted trie nodes.
func newSyncMemBatch() *syncMemBatch {
return &syncMemBatch{
batch: make(map[common.Hash][]byte),
order: make([]common.Hash, 0, 256),
}
}

// TrieSyncLeafCallback is a callback type invoked when a trie sync reaches a
// leaf node. It's used by state syncing to check if the leaf node requires some
// further data syncing.
type TrieSyncLeafCallback func(leaf []byte, parent common.Hash) error

// TrieSync is the main state trie synchronisation scheduler, which provides yet
// unknown trie hashes to retrieve, accepts node data associated with said hashes
// and reconstructs the trie step by step until all is done.
type TrieSync struct {
database DatabaseReader          // Persistent database to check for existing entries
membatch *syncMemBatch           // Memory buffer to avoid frequest database writes
requests map[common.Hash]*request // Pending requests pertaining to a key hash
queue    *prque.Prque            // Priority queue with the pending requests
}

// NewTrieSync creates a new trie data download scheduler.
func NewTrieSync(root common.Hash, database DatabaseReader, callback
TrieSyncLeafCallback) *TrieSync {
ts := &TrieSync{
database: database,
membatch: newSyncMemBatch(),
requests: make(map[common.Hash]*request),
queue:    prque.New(),
}
ts.AddSubTrie(root, 0, common.Hash{}, callback)
return ts
}

// AddSubTrie registers a new trie to the sync code, rooted at the designated parent.
func (s *TrieSync) AddSubTrie(root common.Hash, depth int, parent common.Hash, callback
TrieSyncLeafCallback) {
// Short circuit if the trie is empty or already known
if root == emptyRoot {
return
}
if _, ok := s.membatch.batch[root]; ok {
return
}
key := root.Bytes()
blob, _ := s.database.Get(key)
if local, err := decodeNode(key, blob, 0); local != nil && err == nil {
return
}
// Assemble the new sub-trie sync request
req := &request{
hash:     root,
depth:    depth,
callback: callback,
}
// If this sub-trie has a designated parent, link them together
if parent != (common.Hash{}) {
ancestor := s.requests[parent]
if ancestor == nil {
panic(fmt.Sprintf("sub-trie ancestor not found: %x", parent))
}
ancestor.deps++
req.parents = append(req.parents, ancestor)
}
s.schedule(req)
}

// AddRawEntry schedules the direct retrieval of a state entry that should not be
// interpreted as a trie node, but rather accepted and stored into the database
// as is. This method's goal is to support misc state metadata retrievals (e.g.
// contract code).
```

```
func (s *TrieSync) AddRawEntry(hash common.Hash, depth int, parent common.Hash) {
// Short circuit if the entry is empty or already known
if hash == emptyState {
return
}
if _, ok := s.membatch.batch[hash]; ok {
return
}
if blob, _ := s.database.Get(hash.Bytes()); blob != nil {
return
}
// Assemble the new sub-trie sync request
req := &request{
hash:  hash,
raw:   true,
depth: depth,
}
// If this sub-trie has a designated parent, link them together
if parent != (common.Hash{}) {
ancestor := s.requests[parent]
if ancestor == nil {
panic(fmt.Sprintf("raw-entry ancestor not found: %x", parent))
}
ancestor.deps++
req.parents = append(req.parents, ancestor)
}
s.schedule(req)
}

// Missing retrieves the known missing nodes from the trie for retrieval.
func (s *TrieSync) Missing(max int) []common.Hash {
requests := []common.Hash{}
for !s.queue.Empty() && (max == 0 || len(requests) < max) {
requests = append(requests, s.queue.PopItem().(common.Hash))
}
return requests
}

// Process injects a batch of retrieved trie nodes data, returning if something
// was committed to the database and also the index of an entry if processing of
// it failed.
func (s *TrieSync) Process(results []SyncResult) (bool, int, error) {
```

```
committed := false

for i, item := range results {
// If the item was not requested, bail out
request := s.requests[item.Hash]
if request == nil {
return committed, i, ErrNotRequested
}
if request.data != nil {
return committed, i, ErrAlreadyProcessed
}
// If the item is a raw entry request, commit directly
if request.raw {
request.data = item.Data
s.commit(request)
committed = true
continue
}
// Decode the node data content and update the request
node, err := decodeNode(item.Hash[:], item.Data, 0)
if err != nil {
return committed, i, err
}
request.data = item.Data

// Create and schedule a request for all the children nodes
requests, err := s.children(request, node)
if err != nil {
return committed, i, err
}
if len(requests) == 0 && request.deps == 0 {
s.commit(request)
committed = true
continue
}
request.deps += len(requests)
for _, child := range requests {
s.schedule(child)
}
}
return committed, 0, nil
}
```

```
// Commit flushes the data stored in the internal membatch out to persistent
// storage, returning th enumber of items written and any occurred error.
func (s *TrieSync) Commit(dbw DatabaseWriter) (int, error) {
// Dump the membatch into a database dbw
for i, key := range s.membatch.order {
if err := dbw.Put(key[:], s.membatch.batch[key]); err != nil {
return i, err
}
}
written := len(s.membatch.order)

// Drop the membatch data and return
s.membatch = newSyncMemBatch()
return written, nil
}

// Pending returns the number of state entries currently pending for download.
func (s *TrieSync) Pending() int {
return len(s.requests)
}

// schedule inserts a new state retrieval request into the fetch queue. If there
// is already a pending request for this node, the new request will be discarded
// and only a parent reference added to the old one.
func (s *TrieSync) schedule(req *request) {
// If we're already requesting this node, add a new reference and stop
if old, ok := s.requests[req.hash]; ok {
old.parents = append(old.parents, req.parents...)
return
}
// Schedule the request for future retrieval
s.queue.Push(req.hash, float32(req.depth))
s.requests[req.hash] = req
}

// children retrieves all the missing children of a state trie entry for future
// retrieval scheduling.
func (s *TrieSync) children(req *request, object node) ([]*request, error) {
// Gather all the children of the node, irrelevant whether known or not
type child struct {
node  node
```

```
depth int
}
children := []child{}

switch node := (object).(type) {
case *shortNode:
children = []child{{
node:  node.Val,
depth: req.depth + len(node.Key),
}}
case *fullNode:
for i := 0; i < 17; i++ {
if node.Children[i] != nil {
children = append(children, child{
node:  node.Children[i],
depth: req.depth + 1,
})
}
}
default:
panic(fmt.Sprintf("unknown node: %+v", node))
}
// Iterate over the children, and request all unknown ones
requests := make([]*request, 0, len(children))
for _, child := range children {
// Notify any external watcher of a new key/value node
if req.callback != nil {
if node, ok := (child.node).(valueNode); ok {
if err := req.callback(node, req.hash); err != nil {
return nil, err
}
}
}
// If the child references another node, resolve or schedule
if node, ok := (child.node).(hashNode); ok {
// Try to resolve the node from the local database
hash := common.BytesToHash(node)
if _, ok := s.membatch.batch[hash]; ok {
continue
}
blob, _ := s.database.Get(node)
if local, err := decodeNode(node[:], blob, 0); local != nil && err == nil {
```

```
continue
}
// Locally unknown node, schedule for retrieval
requests = append(requests, &request{
hash:     hash,
parents:  []*request{req},
depth:    child.depth,
callback: req.callback,
})
}
}
return requests, nil
}

// commit finalizes a retrieval request and stores it into the membatch. If any
// of the referencing parent requests complete due to this commit, they are also
// committed themselves.
func (s *TrieSync) commit(req *request) (err error) {
// Write the node content to the membatch
s.membatch.batch[req.hash] = req.data
s.membatch.order = append(s.membatch.order, req.hash)

delete(s.requests, req.hash)

// Check all parents for completion
for _, parent := range req.parents {
parent.deps--
if parent.deps == 0 {
if err := s.commit(parent); err != nil {
return err
}
}
}
return nil
}
```

91:F:\git\coin\ethereum\go-ethereum\trie\sync_test.go

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

package trie

import (
"bytes"
"testing"

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

// makeTestTrie create a sample test trie to test node-wise reconstruction.
func makeTestTrie() (ethdb.Database, *Trie, map[string][]byte) {
// Create an empty trie
db, _ := ethdb.NewMemDatabase()
trie, _ := New(common.Hash{}, db)

// Fill it with some arbitrary data
content := make(map[string][]byte)
for i := byte(0); i < 255; i++ {
// Map the same data under multiple keys
key, val := common.LeftPadBytes([]byte{1, i}, 32), []byte{i}
content[string(key)] = val
trie.Update(key, val)

key, val = common.LeftPadBytes([]byte{2, i}, 32), []byte{i}
content[string(key)] = val
trie.Update(key, val)

// Add some other data to inflate the trie
for j := byte(3); j < 13; j++ {
key, val = common.LeftPadBytes([]byte{j, i}, 32), []byte{j, i}
content[string(key)] = val
trie.Update(key, val)
}
}
trie.Commit()

// Return the generated trie
return db, trie, content
}

// checkTrieContents cross references a reconstructed trie with an expected data
// content map.
func checkTrieContents(t *testing.T, db Database, root []byte, content map[string][]byte) {
// Check root availability and trie contents
```

```
trie, err := New(common.BytesToHash(root), db)
if err != nil {
t.Fatalf("failed to create trie at %x: %v", root, err)
}
if err := checkTrieConsistency(db, common.BytesToHash(root)); err != nil {
t.Fatalf("inconsistent trie at %x: %v", root, err)
}
for key, val := range content {
if have := trie.Get([]byte(key)); !bytes.Equal(have, val) {
t.Errorf("entry %x: content mismatch: have %x, want %x", key, have, val)
}
}
}

// checkTrieConsistency checks that all nodes in a trie are indeed present.
func checkTrieConsistency(db Database, root common.Hash) error {
// Create and iterate a trie rooted in a subnode
trie, err := New(root, db)
if err != nil {
return nil // Consider a non existent state consistent
}
it := trie.NodeIterator(nil)
for it.Next(true) {
}
return it.Error()
}

// Tests that an empty trie is not scheduled for syncing.
func TestEmptyTrieSync(t *testing.T) {
emptyA, _ := New(common.Hash{}, nil)
emptyB, _ := New(emptyRoot, nil)

for i, trie := range []*Trie{emptyA, emptyB} {
db, _ := ethdb.NewMemDatabase()
if req := NewTrieSync(common.BytesToHash(trie.Root()), db, nil).Missing(1); len(req) != 0 {
t.Errorf("test %d: content requested for empty trie: %v", i, req)
}
}
}

// Tests that given a root hash, a trie can sync iteratively on a single thread,
// requesting retrieval tasks and returning all of them in one go.
```

```
func TestIterativeTrieSyncIndividual(t *testing.T) { testIterativeTrieSync(t, 1) }
func TestIterativeTrieSyncBatched(t *testing.T)    { testIterativeTrieSync(t, 100) }

func testIterativeTrieSync(t *testing.T, batch int) {
// Create a random trie to copy
srcDb, srcTrie, srcData := makeTestTrie()

// Create a destination trie and sync with the scheduler
dstDb, _ := ethdb.NewMemDatabase()
sched := NewTrieSync(common.BytesToHash(srcTrie.Root()), dstDb, nil)

queue := append([]common.Hash{}, sched.Missing(batch)...)
for len(queue) > 0 {
results := make([]SyncResult, len(queue))
for i, hash := range queue {
data, err := srcDb.Get(hash.Bytes())
if err != nil {
t.Fatalf("failed to retrieve node data for %x: %v", hash, err)
}
results[i] = SyncResult{hash, data}
}
if _, index, err := sched.Process(results); err != nil {
t.Fatalf("failed to process result #%d: %v", index, err)
}
if index, err := sched.Commit(dstDb); err != nil {
t.Fatalf("failed to commit data #%d: %v", index, err)
}
queue = append(queue[:0], sched.Missing(batch)...)
}
// Cross check that the two tries are in sync
checkTrieContents(t, dstDb, srcTrie.Root(), srcData)
}

// Tests that the trie scheduler can correctly reconstruct the state even if only
// partial results are returned, and the others sent only later.
func TestIterativeDelayedTrieSync(t *testing.T) {
// Create a random trie to copy
srcDb, srcTrie, srcData := makeTestTrie()

// Create a destination trie and sync with the scheduler
dstDb, _ := ethdb.NewMemDatabase()
sched := NewTrieSync(common.BytesToHash(srcTrie.Root()), dstDb, nil)
```

```
queue := append([]common.Hash{}, sched.Missing(10000)...)
for len(queue) > 0 {
// Sync only half of the scheduled nodes
results := make([]SyncResult, len(queue)/2+1)
for i, hash := range queue[:len(results)] {
data, err := srcDb.Get(hash.Bytes())
if err != nil {
t.Fatalf("failed to retrieve node data for %x: %v", hash, err)
}
results[i] = SyncResult{hash, data}
}
if _, index, err := sched.Process(results); err != nil {
t.Fatalf("failed to process result #%d: %v", index, err)
}
if index, err := sched.Commit(dstDb); err != nil {
t.Fatalf("failed to commit data #%d: %v", index, err)
}
queue = append(queue[len(results):], sched.Missing(10000)...)
}
// Cross check that the two tries are in sync
checkTrieContents(t, dstDb, srcTrie.Root(), srcData)
}

// Tests that given a root hash, a trie can sync iteratively on a single thread,
// requesting retrieval tasks and returning all of them in one go, however in a
// random order.
func TestIterativeRandomTrieSyncIndividual(t *testing.T) { testIterativeRandomTrieSync(t, 1) }
func TestIterativeRandomTrieSyncBatched(t *testing.T)    { testIterativeRandomTrieSync(t, 100) }

func testIterativeRandomTrieSync(t *testing.T, batch int) {
// Create a random trie to copy
srcDb, srcTrie, srcData := makeTestTrie()

// Create a destination trie and sync with the scheduler
dstDb, _ := ethdb.NewMemDatabase()
sched := NewTrieSync(common.BytesToHash(srcTrie.Root()), dstDb, nil)

queue := make(map[common.Hash]struct{})
for _, hash := range sched.Missing(batch) {
queue[hash] = struct{}{}
}
```

```
for len(queue) > 0 {
// Fetch all the queued nodes in a random order
results := make([]SyncResult, 0, len(queue))
for hash := range queue {
data, err := srcDb.Get(hash.Bytes())
if err != nil {
t.Fatalf("failed to retrieve node data for %x: %v", hash, err)
}
results = append(results, SyncResult{hash, data})
}
// Feed the retrieved results back and queue new tasks
if _, index, err := sched.Process(results); err != nil {
t.Fatalf("failed to process result #%d: %v", index, err)
}
if index, err := sched.Commit(dstDb); err != nil {
t.Fatalf("failed to commit data #%d: %v", index, err)
}
queue = make(map[common.Hash]struct{})
for _, hash := range sched.Missing(batch) {
queue[hash] = struct{}{}
}
}
// Cross check that the two tries are in sync
checkTrieContents(t, dstDb, srcTrie.Root(), srcData)
}

// Tests that the trie scheduler can correctly reconstruct the state even if only
// partial results are returned (Even those randomly), others sent only later.
func TestIterativeRandomDelayedTrieSync(t *testing.T) {
// Create a random trie to copy
srcDb, srcTrie, srcData := makeTestTrie()

// Create a destination trie and sync with the scheduler
dstDb, _ := ethdb.NewMemDatabase()
sched := NewTrieSync(common.BytesToHash(srcTrie.Root()), dstDb, nil)

queue := make(map[common.Hash]struct{})
for _, hash := range sched.Missing(10000) {
queue[hash] = struct{}{}
}
for len(queue) > 0 {
// Sync only half of the scheduled nodes, even those in random order
results := make([]SyncResult, 0, len(queue)/2+1)
for hash := range queue {
data, err := srcDb.Get(hash.Bytes())
if err != nil {
t.Fatalf("failed to retrieve node data for %x: %v", hash, err)
}
results = append(results, SyncResult{hash, data})

if len(results) >= cap(results) {
break
}
}
// Feed the retrieved results back and queue new tasks
if _, index, err := sched.Process(results); err != nil {
t.Fatalf("failed to process result #%d: %v", index, err)
}
if index, err := sched.Commit(dstDb); err != nil {
t.Fatalf("failed to commit data #%d: %v", index, err)
}
for _, result := range results {
delete(queue, result.Hash)
}
for _, hash := range sched.Missing(10000) {
queue[hash] = struct{}{}
}
}
// Cross check that the two tries are in sync
checkTrieContents(t, dstDb, srcTrie.Root(), srcData)
}

// Tests that a trie sync will not request nodes multiple times, even if they
// have such references.
func TestDuplicateAvoidanceTrieSync(t *testing.T) {
// Create a random trie to copy
srcDb, srcTrie, srcData := makeTestTrie()

// Create a destination trie and sync with the scheduler
dstDb, _ := ethdb.NewMemDatabase()
sched := NewTrieSync(common.BytesToHash(srcTrie.Root()), dstDb, nil)

queue := append([]common.Hash{}, sched.Missing(0)...)
requested := make(map[common.Hash]struct{})
```

```
for len(queue) > 0 {
results := make([]SyncResult, len(queue))
for i, hash := range queue {
data, err := srcDb.Get(hash.Bytes())
if err != nil {
t.Fatalf("failed to retrieve node data for %x: %v", hash, err)
}
if _, ok := requested[hash]; ok {
t.Errorf("hash %x already requested once", hash)
}
requested[hash] = struct{}{}

results[i] = SyncResult{hash, data}
}
if _, index, err := sched.Process(results); err != nil {
t.Fatalf("failed to process result #%d: %v", index, err)
}
if index, err := sched.Commit(dstDb); err != nil {
t.Fatalf("failed to commit data #%d: %v", index, err)
}
queue = append(queue[:0], sched.Missing(0)...)
}
// Cross check that the two tries are in sync
checkTrieContents(t, dstDb, srcTrie.Root(), srcData)
}

// Tests that at any point in time during a sync, only complete sub-tries are in
// the database.
func TestIncompleteTrieSync(t *testing.T) {
// Create a random trie to copy
srcDb, srcTrie, _ := makeTestTrie()

// Create a destination trie and sync with the scheduler
dstDb, _ := ethdb.NewMemDatabase()
sched := NewTrieSync(common.BytesToHash(srcTrie.Root()), dstDb, nil)

added := []common.Hash{}
queue := append([]common.Hash{}, sched.Missing(1)...)
for len(queue) > 0 {
// Fetch a batch of trie nodes
results := make([]SyncResult, len(queue))
for i, hash := range queue {
data, err := srcDb.Get(hash.Bytes())
if err != nil {
t.Fatalf("failed to retrieve node data for %x: %v", hash, err)
}
results[i] = SyncResult{hash, data}
}
// Process each of the trie nodes
if _, index, err := sched.Process(results); err != nil {
t.Fatalf("failed to process result #%d: %v", index, err)
}
if index, err := sched.Commit(dstDb); err != nil {
t.Fatalf("failed to commit data #%d: %v", index, err)
}
for _, result := range results {
added = append(added, result.Hash)
}
// Check that all known sub-tries in the synced trie are complete
for _, root := range added {
if err := checkTrieConsistency(dstDb, root); err != nil {
t.Fatalf("trie inconsistent: %v", err)
}
}
// Fetch the next batch to retrieve
queue = append(queue[:0], sched.Missing(1)...)
}
// Sanity check that removing any node from the database is detected
for _, node := range added[1:] {
key := node.Bytes()
value, _ := dstDb.Get(key)

dstDb.Delete(key)
if err := checkTrieConsistency(dstDb, added[0]); err == nil {
t.Fatalf("trie inconsistency not caught, missing: %x", key)
}
dstDb.Put(key, value)
}
}
```

92:F:\git\coin\ethereum\go-ethereum\trie\trie.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.
// Package trie implements Merkle Patricia Tries.
package trie

import (
"bytes"
"fmt"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto/sha3"
"github.com/ethereum/go-ethereum/log"
"github.com/rcrowley/go-metrics"
)

var (
// This is the known root hash of an empty trie.
emptyRoot =
common.HexToHash("56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b42
1")
// This is the known hash of an empty state trie entry.
emptyState common.Hash
)

var (
cacheMissCounter   = metrics.NewRegisteredCounter("trie/cachemiss", nil)
cacheUnloadCounter = metrics.NewRegisteredCounter("trie/cacheunload", nil)
)

// CacheMisses retrieves a global counter measuring the number of cache misses
// the trie had since process startup. This isn't useful for anything apart from
// trie debugging purposes.
func CacheMisses() int64 {
return cacheMissCounter.Count()
}

// CacheUnloads retrieves a global counter measuring the number of cache unloads
// the trie did since process startup. This isn't useful for anything apart from
// trie debugging purposes.
func CacheUnloads() int64 {
return cacheUnloadCounter.Count()
}

func init() {
```

```
sha3.NewKeccak256().Sum(emptyState[:0])
}

// Database must be implemented by backing stores for the trie.
type Database interface {
DatabaseReader
DatabaseWriter
}

// DatabaseReader wraps the Get method of a backing store for the trie.
type DatabaseReader interface {
Get(key []byte) (value []byte, err error)
}

// DatabaseWriter wraps the Put method of a backing store for the trie.
type DatabaseWriter interface {
// Put stores the mapping key->value in the database.
// Implementations must not hold onto the value bytes, the trie
// will reuse the slice across calls to Put.
Put(key, value []byte) error
}

// Trie is a Merkle Patricia Trie.
// The zero value is an empty trie with no database.
// Use New to create a trie that sits on top of a database.
//
// Trie is not safe for concurrent use.
type Trie struct {
root         node
db           Database
originalRoot common.Hash

// Cache generation values.
// cachegen increases by one with each commit operation.
// new nodes are tagged with the current generation and unloaded
// when their generation is older than than cachegen-cachelimit.
cachegen, cachelimit uint16
}

// SetCacheLimit sets the number of 'cache generations' to keep.
// A cache generation is created by a call to Commit.
func (t *Trie) SetCacheLimit(l uint16) {
```

```
t.cachelimit = l
}

// newFlag returns the cache flag value for a newly created node.
func (t *Trie) newFlag() nodeFlag {
return nodeFlag{dirty: true, gen: t.cachegen}
}

// New creates a trie with an existing root node from db.
//
// If root is the zero hash or the sha3 hash of an empty string, the
// trie is initially empty and does not require a database. Otherwise,
// New will panic if db is nil and returns a MissingNodeError if root does
// not exist in the database. Accessing the trie loads nodes from db on demand.
func New(root common.Hash, db Database) (*Trie, error) {
trie := &Trie{db: db, originalRoot: root}
if (root != common.Hash{}) && root != emptyRoot {
if db == nil {
panic("trie.New: cannot use existing root without a database")
}
rootnode, err := trie.resolveHash(root[:], nil)
if err != nil {
return nil, err
}
trie.root = rootnode
}
return trie, nil
}

// NodeIterator returns an iterator that returns nodes of the trie. Iteration starts at
// the key after the given start key.
func (t *Trie) NodeIterator(start []byte) NodeIterator {
return newNodeIterator(t, start)
}

// Get returns the value for key stored in the trie.
// The value bytes must not be modified by the caller.
func (t *Trie) Get(key []byte) []byte {
res, err := t.TryGet(key)
if err != nil {
log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
}
```

```
return res
}

// TryGet returns the value for key stored in the trie.
// The value bytes must not be modified by the caller.
// If a node was not found in the database, a MissingNodeError is returned.
func (t *Trie) TryGet(key []byte) ([]byte, error) {
key = keybytesToHex(key)
value, newroot, didResolve, err := t.tryGet(t.root, key, 0)
if err == nil && didResolve {
t.root = newroot
}
return value, err
}

func (t *Trie) tryGet(origNode node, key []byte, pos int) (value []byte, newnode node, didResolve
bool, err error) {
switch n := (origNode).(type) {
case nil:
return nil, nil, false, nil
case valueNode:
return n, n, false, nil
case *shortNode:
if len(key)-pos < len(n.Key) || !bytes.Equal(n.Key, key[pos:pos+len(n.Key)]) {
// key not found in trie
return nil, n, false, nil
}
value, newnode, didResolve, err = t.tryGet(n.Val, key, pos+len(n.Key))
if err == nil && didResolve {
n = n.copy()
n.Val = newnode
n.flags.gen = t.cachegen
}
return value, n, didResolve, err
case *fullNode:
value, newnode, didResolve, err = t.tryGet(n.Children[key[pos]], key, pos+1)
if err == nil && didResolve {
n = n.copy()
n.flags.gen = t.cachegen
n.Children[key[pos]] = newnode
}
return value, n, didResolve, err
```

```
case hashNode:
child, err := t.resolveHash(n, key[:pos])
if err != nil {
return nil, n, true, err
}
value, newnode, _, err := t.tryGet(child, key, pos)
return value, newnode, true, err
default:
panic(fmt.Sprintf("%T: invalid node: %v", origNode, origNode))
}
}

// Update associates key with value in the trie. Subsequent calls to
// Get will return value. If value has length zero, any existing value
// is deleted from the trie and calls to Get will return nil.
//
// The value bytes must not be modified by the caller while they are
// stored in the trie.
func (t *Trie) Update(key, value []byte) {
if err := t.TryUpdate(key, value); err != nil {
log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
}
}

// TryUpdate associates key with value in the trie. Subsequent calls to
// Get will return value. If value has length zero, any existing value
// is deleted from the trie and calls to Get will return nil.
//
// The value bytes must not be modified by the caller while they are
// stored in the trie.
//
// If a node was not found in the database, a MissingNodeError is returned.
func (t *Trie) TryUpdate(key, value []byte) error {
k := keybytesToHex(key)
if len(value) != 0 {
_, n, err := t.insert(t.root, nil, k, valueNode(value))
if err != nil {
return err
}
t.root = n
} else {
_, n, err := t.delete(t.root, nil, k)
```

```
if err != nil {
return err
}
t.root = n
}
return nil
}

func (t *Trie) insert(n node, prefix, key []byte, value node) (bool, node, error) {
if len(key) == 0 {
if v, ok := n.(valueNode); ok {
return !bytes.Equal(v, value.(valueNode)), value, nil
}
return true, value, nil
}
switch n := n.(type) {
case *shortNode:
matchlen := prefixLen(key, n.Key)
// If the whole key matches, keep this short node as is
// and only update the value.
if matchlen == len(n.Key) {
dirty, nn, err := t.insert(n.Val, append(prefix, key[:matchlen]...), key[matchlen:], value)
if !dirty || err != nil {
return false, n, err
}
return true, &shortNode{n.Key, nn, t.newFlag()}, nil
}
// Otherwise branch out at the index where they differ.
branch := &fullNode{flags: t.newFlag()}
var err error
_, branch.Children[n.Key[matchlen]], err = t.insert(nil, append(prefix, n.Key[:matchlen+1]...),
n.Key[matchlen+1:], n.Val)
if err != nil {
return false, nil, err
}
_, branch.Children[key[matchlen]], err = t.insert(nil, append(prefix, key[:matchlen+1]...),
key[matchlen+1:], value)
if err != nil {
return false, nil, err
}
// Replace this shortNode with the branch if it occurs at index 0.
if matchlen == 0 {
```

```
return true, branch, nil
}
// Otherwise, replace it with a short node leading up to the branch.
return true, &shortNode{key[:matchlen], branch, t.newFlag()}, nil

case *fullNode:
dirty, nn, err := t.insert(n.Children[key[0]], append(prefix, key[0]), key[1:], value)
if !dirty || err != nil {
return false, n, err
}
n = n.copy()
n.flags = t.newFlag()
n.Children[key[0]] = nn
return true, n, nil

case nil:
return true, &shortNode{key, value, t.newFlag()}, nil

case hashNode:
// We've hit a part of the trie that isn't loaded yet. Load
// the node and insert into it. This leaves all child nodes on
// the path to the value in the trie.
rn, err := t.resolveHash(n, prefix)
if err != nil {
return false, nil, err
}
dirty, nn, err := t.insert(rn, prefix, key, value)
if !dirty || err != nil {
return false, rn, err
}
return true, nn, nil

default:
panic(fmt.Sprintf("%T: invalid node: %v", n, n))
}
}

// Delete removes any existing value for key from the trie.
func (t *Trie) Delete(key []byte) {
if err := t.TryDelete(key); err != nil {
log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
}
```

```
}

// TryDelete removes any existing value for key from the trie.
// If a node was not found in the database, a MissingNodeError is returned.
func (t *Trie) TryDelete(key []byte) error {
k := keybytesToHex(key)
_, n, err := t.delete(t.root, nil, k)
if err != nil {
return err
}
t.root = n
return nil
}

// delete returns the new root of the trie with key deleted.
// It reduces the trie to minimal form by simplifying
// nodes on the way up after deleting recursively.
func (t *Trie) delete(n node, prefix, key []byte) (bool, node, error) {
switch n := n.(type) {
case *shortNode:
matchlen := prefixLen(key, n.Key)
if matchlen < len(n.Key) {
return false, n, nil // don't replace n on mismatch
}
if matchlen == len(key) {
return true, nil, nil // remove n entirely for whole matches
}
// The key is longer than n.Key. Remove the remaining suffix
// from the subtrie. Child can never be nil here since the
// subtrie must contain at least two other values with keys
// longer than n.Key.
dirty, child, err := t.delete(n.Val, append(prefix, key[:len(n.Key)]...), key[len(n.Key):])
if !dirty || err != nil {
return false, n, err
}
switch child := child.(type) {
case *shortNode:
// Deleting from the subtrie reduced it to another
// short node. Merge the nodes to avoid creating a
// shortNode{..., shortNode{...}}. Use concat (which
// always creates a new slice) instead of append to
// avoid modifying n.Key since it might be shared with
// other nodes.
return true, &shortNode{concat(n.Key, child.Key...), child.Val, t.newFlag()}, nil
default:
return true, &shortNode{n.Key, child, t.newFlag()}, nil
}

case *fullNode:
dirty, nn, err := t.delete(n.Children[key[0]], append(prefix, key[0]), key[1:])
if !dirty || err != nil {
return false, n, err
}
n = n.copy()
n.flags = t.newFlag()
n.Children[key[0]] = nn

// Check how many non-nil entries are left after deleting and
// reduce the full node to a short node if only one entry is
// left. Since n must've contained at least two children
// before deletion (otherwise it would not be a full node) n
// can never be reduced to nil.
//
// When the loop is done, pos contains the index of the single
// value that is left in n or -2 if n contains at least two
// values.
pos := -1
for i, cld := range n.Children {
if cld != nil {
if pos == -1 {
pos = i
} else {
pos = -2
break
}
}
}
if pos >= 0 {
if pos != 16 {
// If the remaining entry is a short node, it replaces
// n and its key gets the missing nibble tacked to the
// front. This avoids creating an invalid
// shortNode{..., shortNode{...}}.  Since the entry
// might not be loaded yet, resolve it just for this
// check.
cnode, err := t.resolve(n.Children[pos], prefix)
if err != nil {
return false, nil, err
}
if cnode, ok := cnode.(*shortNode); ok {
k := append([]byte{byte(pos)}, cnode.Key...)
return true, &shortNode{k, cnode.Val, t.newFlag()}, nil
}
}
// Otherwise, n is replaced by a one-nibble short node
// containing the child.
return true, &shortNode{[]byte{byte(pos)}, n.Children[pos], t.newFlag()}, nil
}
// n still contains at least two values and cannot be reduced.
return true, n, nil

case valueNode:
return true, nil, nil

case nil:
return false, nil, nil

case hashNode:
// We've hit a part of the trie that isn't loaded yet. Load
// the node and delete from it. This leaves all child nodes on
// the path to the value in the trie.
rn, err := t.resolveHash(n, prefix)
if err != nil {
return false, nil, err
}
dirty, nn, err := t.delete(rn, prefix, key)
if !dirty || err != nil {
return false, rn, err
}
return true, nn, nil

default:
panic(fmt.Sprintf("%T: invalid node: %v (%v)", n, n, key))
}
}
```

```
func concat(s1 []byte, s2 ...byte) []byte {
r := make([]byte, len(s1)+len(s2))
copy(r, s1)
copy(r[len(s1):], s2)
return r
}

func (t *Trie) resolve(n node, prefix []byte) (node, error) {
if n, ok := n.(hashNode); ok {
return t.resolveHash(n, prefix)
}
return n, nil
}

func (t *Trie) resolveHash(n hashNode, prefix []byte) (node, error) {
cacheMissCounter.Inc(1)

enc, err := t.db.Get(n)
if err != nil || enc == nil {
return nil, &MissingNodeError{NodeHash: common.BytesToHash(n), Path: prefix}
}
dec := mustDecodeNode(n, enc, t.cachegen)
return dec, nil
}

// Root returns the root hash of the trie.
// Deprecated: use Hash instead.
func (t *Trie) Root() []byte { return t.Hash().Bytes() }

// Hash returns the root hash of the trie. It does not write to the
// database and can be used even if the trie doesn't have one.
func (t *Trie) Hash() common.Hash {
hash, cached, _ := t.hashRoot(nil)
t.root = cached
return common.BytesToHash(hash.(hashNode))
}

// Commit writes all nodes to the trie's database.
// Nodes are stored with their sha3 hash as the key.
//
// Committing flushes nodes from memory.
// Subsequent Get calls will load nodes from the database.
```

```
func (t *Trie) Commit() (root common.Hash, err error) {
if t.db == nil {
panic("Commit called on trie with nil database")
}
return t.CommitTo(t.db)
}

// CommitTo writes all nodes to the given database.
// Nodes are stored with their sha3 hash as the key.
//
// Committing flushes nodes from memory. Subsequent Get calls will
// load nodes from the trie's database. Calling code must ensure that
// the changes made to db are written back to the trie's attached
// database before using the trie.
func (t *Trie) CommitTo(db DatabaseWriter) (root common.Hash, err error) {
hash, cached, err := t.hashRoot(db)
if err != nil {
return (common.Hash{}), err
}
t.root = cached
t.cachegen++
return common.BytesToHash(hash.(hashNode)), nil
}

func (t *Trie) hashRoot(db DatabaseWriter) (node, node, error) {
if t.root == nil {
return hashNode(emptyRoot.Bytes()), nil, nil
}
h := newHasher(t.cachegen, t.cachelimit)
defer returnHasherToPool(h)
return h.hash(t.root, db, true)
}
```