A closer look at Geth

tuna@tomochain.com

About me

- Blockchain Lead Engineer @ Tomochain
- I’ve been virtualized and containerized for almost 10 years.
  - Kubernetes, Docker, CloudStack contributor
  - Kubeless Lead Engineer @ Bitnami
  - Kubernetes Lead Engineer @ Skippbox
  - Software Engineer @ FPT, Viettel, Bkav

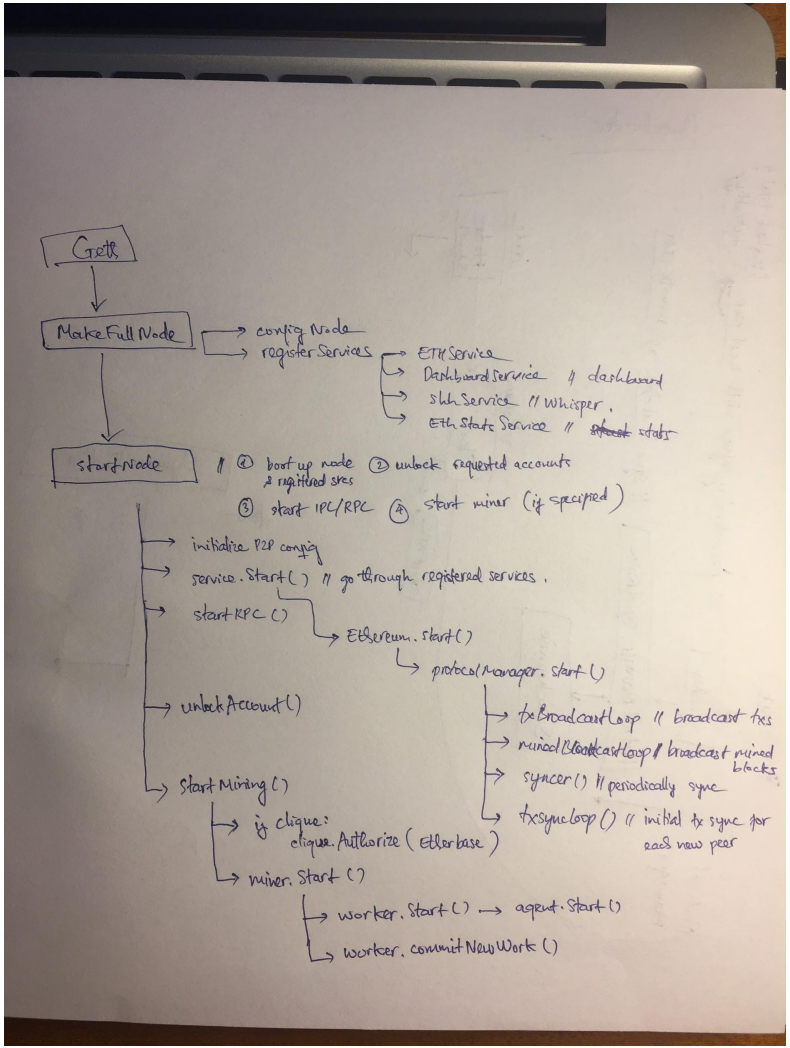
The 10000 foot view

Some little pieces of Geth...

- Block synchronization
- Mining
- Consensus
- Transaction Pool
- Account
- P2P
- RLP
- RPC
- Crypto
- Contracts

1. Block synchronization

What happens once a node starts...



Protocol Manager

```
func (pm *ProtocolManager) Start(maxPeers int) {
    pm.maxPeers = maxPeers

    // broadcast transactions
    pm.txCh = make(chan core.TxPreEvent, txChanSize)
    pm.txSub = pm.txpool.SubscribeTxPreEvent(pm.txCh)
    go pm.txBroadcastLoop()

    // broadcast mined blocks
    pm.minedBlockSub = pm.eventMux.Subscribe(core.NewMinedBlockEvent{})
    go pm.minedBroadcastLoop()

    // start sync handlers
    go pm.syncer()
    go pm.txsyncLoop()
}
```

Broadcast txs

```
func (self *ProtocolManager) txBroadcastLoop() {
    for {
        select {
        case event := <-self.txCh:
            self.BroadcastTx(event.Tx.Hash(), event.Tx)

        // Err() channel will be closed when unsubscribing.
        case <-self.txSub.Err():
            return
        }
    }
}
```

Broadcast tx

```
// BroadcastTx will propagate a transaction to all peers which are not known to
// already have the given transaction.
func (pm *ProtocolManager) BroadcastTx(hash common.Hash, tx *types.Transaction) {
    // Broadcast transaction to a batch of peers not knowing about it
    peers := pm.peers.PeersWithoutTx(hash)
    //FIXME include this again: peers = peers[:int(math.Sqrt(float64(len(peers))))]
    for _, peer := range peers {
        peer.SendTransactions(types.Transactions{tx})
    }
    log.Trace( msg: "Broadcast transaction",  ctx: "hash", hash, "recipients", len(peers))
}
```

Broadcast mined blocks

```
// Mined broadcast loop
func (self *ProtocolManager) minedBroadcastLoop() {
    // automatically stops if unsubscribe
    for obj := range self.minedBlockSub.Chan() {
        switch ev := obj.Data.(type) {
        case core.NewMinedBlockEvent:
            self.BroadcastBlock(ev.Block, propagate: true)  // First propagate block to peers
            self.BroadcastBlock(ev.Block, propagate: false) // Only then announce to the rest
        }
    }
}
```

Broadcast mined block

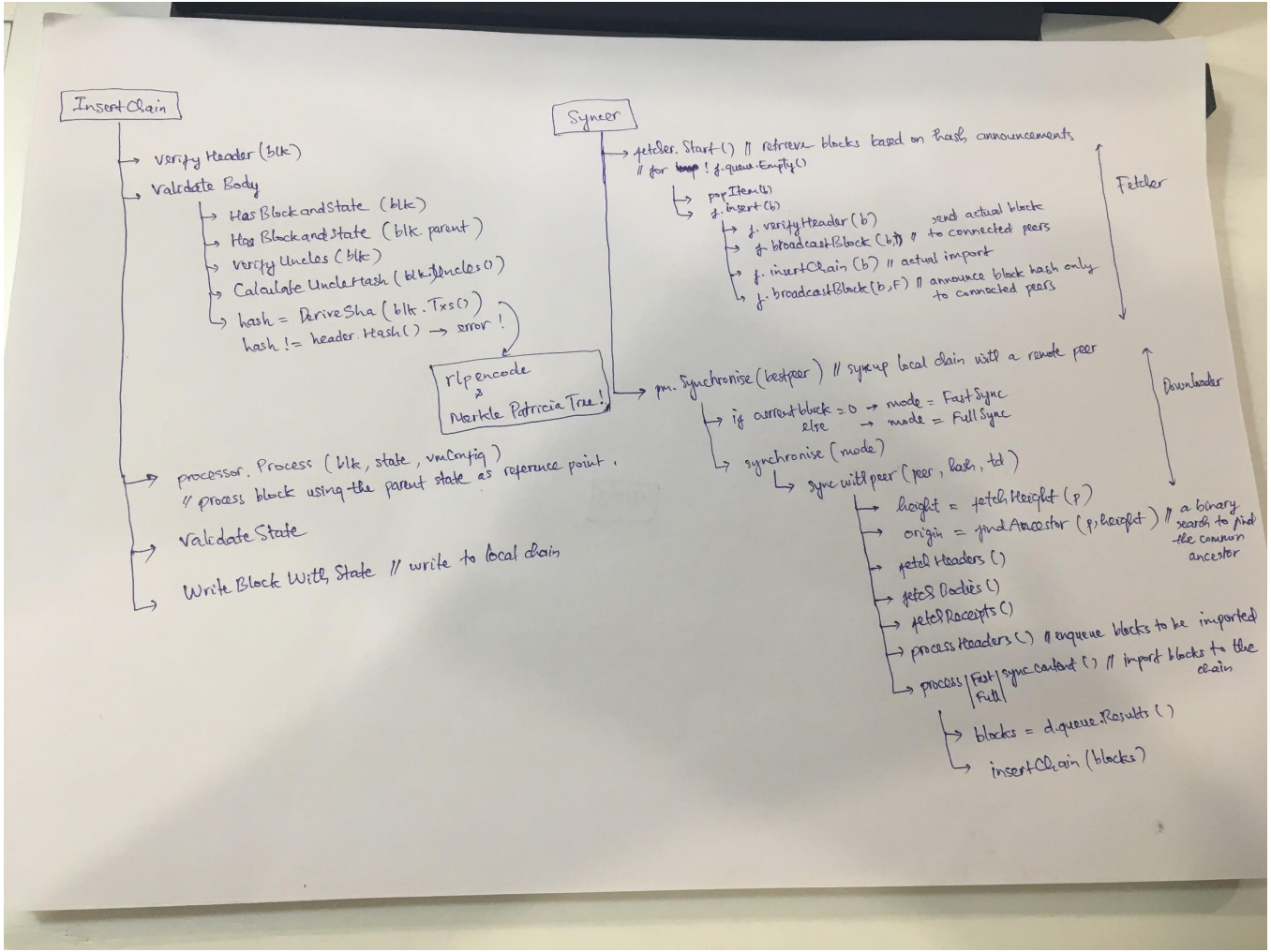
```
// BroadcastBlock will either propagate a block to a subset of it's peers, or
// will only announce it's availability (depending what's requested).
func (pm *ProtocolManager) BroadcastBlock(block *types.Block, propagate bool) {
    hash := block.Hash()
    peers := pm.peers.PeersWithoutBlock(hash)

    // If propagation is requested, send to a subset of the peer
    if propagate {
        // Calculate the TD of the block (it's not imported yet, so block.Td is not valid)
        var td *big.Int
        if parent := pm.blockchain.GetBlock(block.ParentHash(), block.NumberU64()-1); parent != nil {
            td = new(big.Int).Add(block.Difficulty(), pm.blockchain.GetTd(block.ParentHash(), block.NumberU64()-1))
        } else {
            log.Error( msg: "Propagating dangling block",  ctx: "number", block.Number(), "hash", hash)
            return
        }
        // Send the block to a subset of our peers
        transfer := peers[:int(math.Sqrt(float64(len(peers))))]
        for _, peer := range transfer {
            peer.SendNewBlock(block, td)
        }
        log.Trace( msg: "Propagated block",  ctx: "hash", hash, "recipients", len(transfer), "duration", common.PrettyDura
        return
    }
    // Otherwise if the block is indeed in out own chain, announce it
    if pm.blockchain.HasBlock(hash, block.NumberU64()) {
        for _, peer := range peers {
            peer.SendNewBlockHashes([]common.Hash{hash}, []uint64{block.NumberU64()})
        }
        log.Trace( msg: "Announced block",  ctx: "hash", hash, "recipients", len(peers), "duration", common.PrettyDuratio
    }
}
```

Block syncer

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

    // Wait for different events to fire synchronisation operations
    forceSync := time.NewTicker(forceSyncCycle)
    defer forceSync.Stop()

    for {
        select {
        case <-pm.newPeerCh:
            // Make sure we have peers to select from, then sync
            if pm.peers.Len() < minDesiredPeerCount {
                break
            }
            go pm.synchronise(pm.peers.BestPeer())

        case <-forceSync.C:
            // Force a sync even if not enough peers are present
            go pm.synchronise(pm.peers.BestPeer())

        case <-pm.noMorePeers:
            return
        }
    }
}
```

Pick up the best peer

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

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


InsertChain
verifyHeader(blk)
validateBody
HasBlockAndState(blk)
HasBlockAndState(blk.parent)
verifyUncles(blk)
CalculateUncleHash(blk.Uncles())
hash = DeriveSha(blk.Txs())
hash != header.Hash() → error!
rlpencode
↘
Merkle Patricia Tree!
processor.Process(blk, state, vmConfig)
// process block using the parent state as reference point.
validateState
WriteBlockWithState // write to local chain
Syncer
fetcher.Start() // retrieve blocks based on hash announcements
// for ~~loop~~ ! f.queue.Empty()
popItem()
f.insert(b)
f.verifyHeader(b)
f.broadcastBlock(b,T) // send actual block to connected peers
f.insertChain(b) // actual import
f.broadcastBlock(b,F) // announce block hash only to connected peers
Fetcher
pm.Synchronise(bestpeer) // sync up local chain with a remote peer
if currentBlock = 0 → mode = FastSync
else → mode = FullSync
synchronise(mode)
syncWithPeer(peer, hash, td)
height = fetchHeight(p)
origin = findAncestor(p,height) // a binary search to find the common ancestor
fetchHeaders()
fetchBodies()
fetchReceipts()
processHeaders() // enqueue blocks to be imported
process[Fast|Full]SyncContent() // import blocks to the chain
blocks = d.queue.Results()
insertChain(blocks)
Downloader

Full sync content

```
// processFullSyncContent takes fetch results from the queue and imports them into the chain.
func (d *Downloader) processFullSyncContent() error {
    for {
        results := d.queue.Results( block: true)
        if len(results) == 0 {
            return nil
        }
        if d.chainInsertHook != nil {
            d.chainInsertHook(results)
        }
        if err := d.importBlockResults(results); err != nil {
            return err
        }
    }
}
```

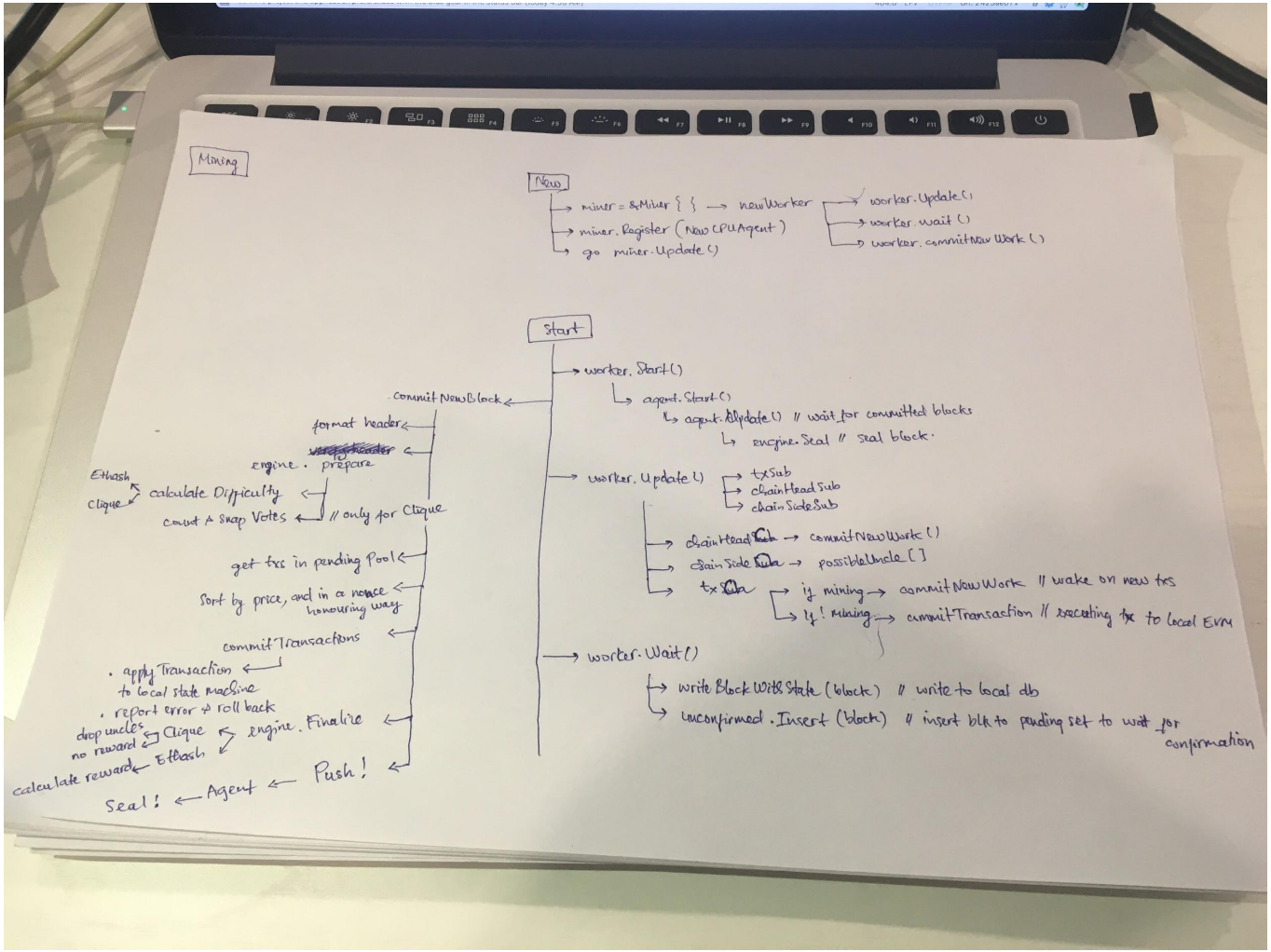
Process a block

```go
// Process processes the state changes according to the Ethereum rules by running
// the transaction messages using the statedb and applying any rewards to both
// the processor (coinbase) and any included uncles.
//
// Process returns the receipts and logs accumulated during the process and
// returns the amount of gas that was used in the process. If any of the
// transactions failed to execute due to insufficient gas it will return an error.
func (p *StateProcessor) Process(block *types.Block, statedb *state.StateDB, cfg vm.Config) (types.Receipts, []*types.Log, uint64,
    var (
        receipts types.Receipts
        usedGas  = new(uint64)
        header   = block.Header()
        allLogs  []*types.Log
        gp       = new(GasPool).AddGas(block.GasLimit())
    )
    // Mutate the the block and state according to any hard-fork specs
    if p.config.DAOForkSupport && p.config.DAOForkBlock != nil && p.config.DAOForkBlock.Cmp(block.Number()) == 0 {
        misc.ApplyDAOHardFork(statedb)
    }
    // Iterate over and process the individual transactions
    for i, tx := range block.Transactions() {
        statedb.Prepare(tx.Hash(), block.Hash(), i)
        receipt, _, err := ApplyTransaction(p.config, p.bc,  author: nil, gp, statedb, header, tx, usedGas, cfg)
        if err != nil {
            return nil, nil, 0, err
        }
        receipts = append(receipts, receipt)
        allLogs = append(allLogs, receipt.Logs...)
    }
    // Finalize the block, applying any consensus engine specific extras (e.g. block rewards)
    p.engine.Finalize(p.bc, header, statedb, block.Transactions(), block.Uncles(), receipts)

    return receipts, allLogs, *usedGas, nil
}
```

Apply transactions

```go
// ApplyTransaction attempts to apply a transaction to the given state database
// and uses the input parameters for its environment. It returns the receipt
// for the transaction, gas used and an error if the transaction failed,
// indicating the block was invalid.
func ApplyTransaction(config *params.ChainConfig, bc *BlockChain, author *common.Address, gp *GasPool, statedb *state.StateDB, header
    msg, err := tx.AsMessage(types.MakeSigner(config, header.Number))
    if err != nil {
        return nil, 0, err
    }
    // Create a new context to be used in the EVM environment
    context := NewEVMContext(msg, header, bc, author)
    // Create a new environment which holds all relevant information
    // about the transaction and calling mechanisms.
    vmenv := vm.NewEVM(context, statedb, config, cfg)
    // Apply the transaction to the current state (included in the env)
    _, gas, failed, err := ApplyMessage(vmenv, msg, gp)
    if err != nil {
        return nil, 0, err
    }
    // Update the state with pending changes
    var root []byte
    if config.IsByzantium(header.Number) {
        statedb.Finalise( deleteEmptyObjects: true)
    } else {
        root = statedb.IntermediateRoot(config.IsEIP158(header.Number)).Bytes()
    }
    *usedGas += gas

    // Create a new receipt for the transaction, storing the intermediate root and gas used by the tx
    // based on the eip phase, we're passing wether the root touch-delete accounts.
    receipt := types.NewReceipt(root, failed, *usedGas)
    receipt.TxHash = tx.Hash()
    receipt.GasUsed = gas
    // if the transaction created a contract, store the creation address in the receipt.
    if msg.To() == nil {
        receipt.ContractAddress = crypto.CreateAddress(vmenv.Context.Origin, tx.Nonce())
    }
    // Set the receipt logs and create a bloom for filtering
    receipt.Logs = statedb.GetLogs(tx.Hash())
    receipt.Bloom = types.CreateBloom(types.Receipts{receipt})

    return receipt, gas, err
}
```

2. Mining


Mining
New
miner = &Miner{} → newWorker
miner.Register(NewCPUAgent)
go miner.Update()
worker.Update()
worker.wait()
worker.commitNewWork()
Start
worker.Start()
agent.Start()
agent.dlpdate() // wait for committed blocks
engine.Seal // seal block.
commitNewBlock
format header
engine.prepare
Ethash
Clique
calculate Difficulty
count & snap Votes // only for Clique
get txs in pending Pool
Sort by price, and in a nonce honouring way
commitTransactions
. apply Transaction to local state machine
. report error & roll back
drop uncles
no reward
Clique
engine.Finalize
calculate reward
Ethash
Seal!
Agent
Push!
worker.Update()
txSub
chainHeadSub
chainSideSub
chainHead → commitNewWork()
chainSide → possibleUncle[]
tx → if mining → commitNewWork // wake on new txs
if ! mining → commitTransaction // executing tx to local EVM
worker.Wait()
writeBlockWithState(block) // write to local db
unconfirmed.Insert(block) // insert blk to pending set to wait for confirmation

Mining

```
// Miner creates blocks and searches for proof-of-work values.
type Miner struct {
    mux *event.TypeMux

    worker *worker

    coinbase common.Address
    mining   int32
    eth      Backend
    engine   consensus.Engine

    canStart    int32 // can start indicates whether we can start the mining operation
    shouldStart int32 // should start indicates whether we should start after sync
}

func New(eth Backend, config *params.ChainConfig, mux *event.TypeMux, engine consensus.Engine) *Miner {
    miner := &Miner{
        eth:      eth,
        mux:      mux,
        engine:   engine,
        worker:   newWorker(config, engine, common.Address{}, eth, mux),
        canStart: 1,
    }
    miner.Register(NewCpuAgent(eth.BlockChain(), engine))
    go miner.update()

    return miner
}
```

Mining

```
func (self *Miner) Start(coinbase common.Address) {
    atomic.StoreInt32(&self.shouldStart, val: 1)
    self.SetEtherbase(coinbase)

    if atomic.LoadInt32(&self.canStart) == 0 {
        log.Info( msg: "Network syncing, will start miner afterwards")
        return
    }
    atomic.StoreInt32(&self.mining, val: 1)

    log.Info( msg: "Starting mining operation")
    self.worker.start()
    self.worker.commitNewWork()
}

func (self *Miner) Stop() {
    self.worker.stop()
    atomic.StoreInt32(&self.mining, val: 0)
    atomic.StoreInt32(&self.shouldStart, val: 0)
}
```

```
func (self *Miner) Register(agent Agent) {
    if self.Mining() {
        agent.Start()
    }
    self.worker.register(agent)
}

func (self *Miner) Unregister(agent Agent) {
    self.worker.unregister(agent)
}
```

Agent loop...

```
func (self *CpuAgent) update() {
out:
    for {
        select {
        case work := <-self.workCh:
            self.mu.Lock()
            if self.quitCurrentOp != nil {
                close(self.quitCurrentOp)
            }
            self.quitCurrentOp = make(chan struct{})
            go self.mine(work, self.quitCurrentOp)
            self.mu.Unlock()
        case <-self.stop:
            self.mu.Lock()
            if self.quitCurrentOp != nil {
                close(self.quitCurrentOp)
                self.quitCurrentOp = nil
            }
            self.mu.Unlock()
            break out
        }
    }
```

```
func (self *CpuAgent) mine(work *Work, stop <-chan struct{}) {
    if result, err := self.engine.Seal(self.chain, work.Block, stop); result != nil {
        log.Info( msg: "Successfully sealed new block",  ctx: "number", result.Number(), "hash", result.Hash())
        self.returnCh <- &Result{ Work: work,  Block: result}
    } else {
        if err != nil {
            log.Warn( msg: "Block sealing failed",  ctx: "err", err)
        }
        self.returnCh <- nil
    }
}
```

Worker loop...

```
for {
    // A real event arrived, process interesting content
    select {
    // Handle ChainHeadEvent
    case <-self.chainHeadCh:
        self.commitNewWork()

    // Handle ChainSideEvent
    case ev := <-self.chainSideCh:
        self.uncleMu.Lock()
        self.possibleUncles[ev.Block.Hash()] = ev.Block
        self.uncleMu.Unlock()

    // Handle TxPreEvent
    case ev := <-self.txCh:
        // Apply transaction to the pending state if we're not mining
        if atomic.LoadInt32(&self.mining) == 0 {
            self.currentMu.Lock()
            acc, _ := types.Sender(self.current.signer, ev.Tx)
            txs := map[common.Address]types.Transactions{acc: {ev.Tx}}
            txset := types.NewTransactionsByPriceAndNonce(self.current.signer, txs)

            self.current.commitTransactions(self.mux, txset, self.chain, self.coinbase)
            self.updateSnapshot()
            self.currentMu.Unlock()
        } else {
            // If we're mining, but nothing is being processed, wake on new transactions
            if self.config.Clique != nil && self.config.Clique.Period == 0 {
                self.commitNewWork()
            }
        }

    // System stopped
    case <-self.txSub.Err():
        return
    case <-self.chainHeadSub.Err():
        return
    case <-self.chainSideSub.Err():
        return
    }
}
```

3. Consensus

Interfaces

```
// ChainReader defines a small collection of methods needed to access the local
// blockchain during header and/or uncle verification.
type ChainReader interface {
    // Config retrieves the blockchain's chain configuration.
    Config() *params.ChainConfig

    // CurrentHeader retrieves the current header from the local chain.
    CurrentHeader() *types.Header

    // GetHeader retrieves a block header from the database by hash and number.
    GetHeader(hash common.Hash, number uint64) *types.Header

    // GetHeaderByNumber retrieves a block header from the database by number.
    GetHeaderByNumber(number uint64) *types.Header

    // GetHeaderByHash retrieves a block header from the database by its hash.
    GetHeaderByHash(hash common.Hash) *types.Header

s of method specification 'GetBlock'

    // GetBlock retrieves a block from the database by hash and number.
    GetBlock(hash common.Hash, number uint64) *types.Block
}
```

Interfaces

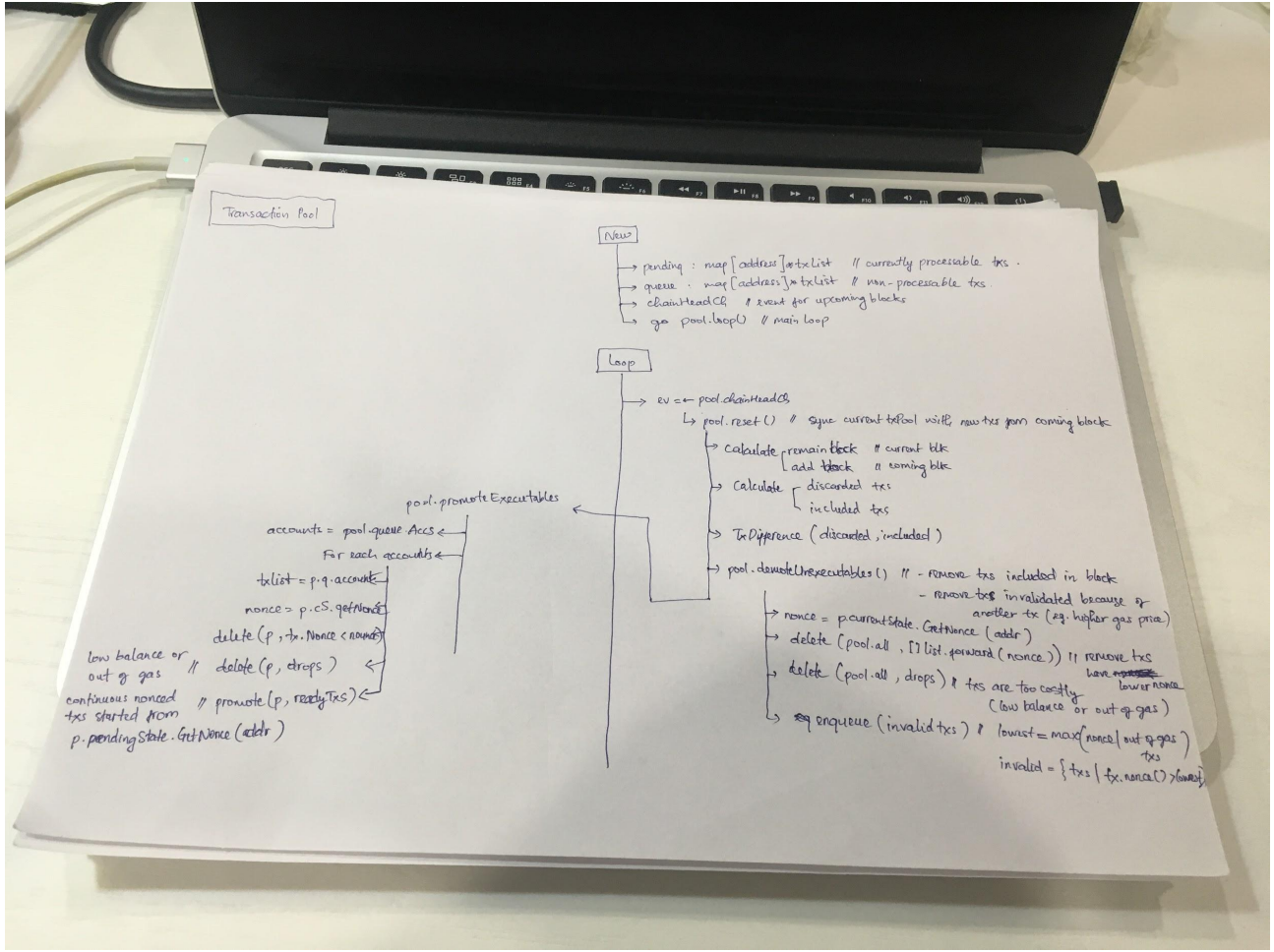
```go
// Engine is an algorithm agnostic consensus engine.
type Engine interface {
    // Author retrieves the Ethereum address of the account that minted the given
    // block, which may be different from the header's coinbase if a consensus
    // engine is based on signatures.
    Author(header *types.Header) (common.Address, error)

    // VerifyHeader checks whether a header conforms to the consensus rules of a
    // given engine. Verifying the seal may be done optionally here, or explicitly
    // via the VerifySeal method.
    VerifyHeader(chain ChainReader, header *types.Header, seal bool) error

    // VerifyHeaders is similar to VerifyHeader, but verifies a batch of headers
    // concurrently. The method returns a quit channel to abort the operations and
    // a results channel to retrieve the async verifications (the order is that of
    // the input slice).
    VerifyHeaders(chain ChainReader, headers []*types.Header, seals []bool) (chan<- struct{}, <-chan error)

    // VerifyUncles verifies that the given block's uncles conform to the consensus
    // rules of a given engine.
    VerifyUncles(chain ChainReader, block *types.Block) error

    // VerifySeal checks whether the crypto seal on a header is valid according to
    // the consensus rules of the given engine.
    VerifySeal(chain ChainReader, header *types.Header) error

    // Prepare initializes the consensus fields of a block header according to the
    // rules of a particular engine. The changes are executed inline.
    Prepare(chain ChainReader, header *types.Header) error

    // Finalize runs any post-transaction state modifications (e.g. block rewards)
    // and assembles the final block.
    // Note: The block header and state database might be updated to reflect any
    // consensus rules that happen at finalization (e.g. block rewards).
    Finalize(chain ChainReader, header *types.Header, state *state.StateDB, txs []*types.Transaction,
        uncles []*types.Header, receipts []*types.Receipt) (*types.Block, error)

    // Seal generates a new block for the given input block with the local miner's
    // seal place on top.
    Seal(chain ChainReader, block *types.Block, stop <-chan struct{}) (*types.Block, error)

    // CalcDifficulty is the difficulty adjustment algorithm. It returns the difficulty
    // that a new block should have.
    CalcDifficulty(chain ChainReader, time uint64, parent *types.Header) *big.Int

    // APIs returns the RPC APIs this consensus engine provides.
    APIs(chain ChainReader) []rpc.API
```

Clique

```go
// New creates a Clique proof-of-authority consensus engine with the initial
// signers set to the ones provided by the user.
func New(config *params.CliqueConfig, db ethdb.Database) *Clique {
    // Set any missing consensus parameters to their defaults
    conf := *config
    if conf.Epoch == 0 {
        conf.Epoch = epochLength
    }
    // Allocate the snapshot caches and create the engine
    recents, _ := lru.NewARC(inmemorySnapshots)
    signatures, _ := lru.NewARC(inmemorySignatures)

    return &Clique{
        config:     &conf,
        db:         db,
        recents:    recents,
        signatures: signatures,
        proposals:  make(map[common.Address]bool),
    }
}
```

It’s better to look at code...

4. Transaction Pool


Transaction Pool
New
pending : map[address]*txList // currently processable txs.
queue : map[address]*txList // non-processable txs.
chainHeadCh // event for upcoming blocks
go pool.loop() // main loop
Loop
ev := <- pool.chainHeadCh
pool.reset() // sync current txPool with new txs from coming block
Calculate remain block // current blk
add block // coming blk
Calculate discarded txs
included txs
TxDifference (discarded, included)
pool.demoteUnexecutables() // - remove txs included in block
- remove txs invalidated because of another tx (eg. higher gas price)
nonce = p.currentState.GetNonce(addr)
delete (pool.all, []list.forward(nonce)) // remove txs have lower nonce
delete (pool.all, drops) // txs are too costly (low balance or out of gas)
enqueue (invalid txs) // lowest = max(nonce | out of gas txs)
invalid = {txs | tx.nonce() > lowest}
pool.promoteExecutables
accounts = pool.queue Accs
For each accounts
txlist = p.q.accounts
nonce = p.cS.getNonce
delete (p, tx.Nonce < nonce)
low balance or out of gas // delete (p, drops)
continuous nonced txs started from p.pendingState.GetNonce(addr)
// promote (p, readyTxs)

Transaction Pool

```go
// NewTxPool creates a new transaction pool to gather, sort and filter inbound
// transactions from the network.
func NewTxPool(config TxPoolConfig, chainconfig *params.ChainConfig, chain blockChain) *TxPool {
    // Sanitize the input to ensure no vulnerable gas prices are set
    config = (&config).sanitize()

    // Create the transaction pool with its initial settings
    pool := &TxPool{
        config:      config,
        chainconfig: chainconfig,
        chain:       chain,
        signer:      types.NewEIP155Signer(chainconfig.ChainId),
        pending:     make(map[common.Address]*txList),
        queue:       make(map[common.Address]*txList),
        beats:       make(map[common.Address]time.Time),
        all:         make(map[common.Hash]*types.Transaction),
        chainHeadCh: make(chan ChainHeadEvent, chainHeadChanSize),
        gasPrice:    new(big.Int).SetUint64(config.PriceLimit),
    }
    pool.locals = newAccountSet(pool.signer)
    pool.priced = newTxPricedList(&pool.all)
    pool.reset( oldHead: nil, chain.CurrentBlock().Header())

    // If local transactions and journaling is enabled, load from disk
    if !config.NoLocals && config.Journal != "" {
        pool.journal = newTxJournal(config.Journal)

        if err := pool.journal.load(pool.AddLocal); err != nil {
            log.Warn( msg: "Failed to load transaction journal",  ctx: "err", err)
        }
        if err := pool.journal.rotate(pool.local()); err != nil {
            log.Warn( msg: "Failed to rotate transaction journal",  ctx: "err", err)
        }
    }
    // Subscribe events from blockchain
    pool.chainHeadSub = pool.chain.SubscribeChainHeadEvent(pool.chainHeadCh)

    // Start the event loop and return
    pool.wg.Add( delta: 1)
    go pool.loop()

    return pool
}
```

New block comes...

```
// Handle ChainHeadEvent
case ev := <-pool.chainHeadCh:
    if ev.Block != nil {
        pool.mu.Lock()
        if pool.chainconfig.IsHomestead(ev.Block.Number()) {
            pool.homestead = true
        }
        pool.reset(head.Header(), ev.Block.Header())
        head = ev.Block

        pool.mu.Unlock()
    }
// Be unsubscribed due to system stopped
case <-pool.chainHeadSub.Err():
    return
```

Demote Unexecutables

```go
// demoteUnexecutables removes invalid and processed transactions from the pools
// executable/pending queue and any subsequent transactions that become unexecutable
// are moved back into the future queue.
func (pool *TxPool) demoteUnexecutables() {
    // Iterate over all accounts and demote any non-executable transactions
    for addr, list := range pool.pending {
        nonce := pool.currentState.GetNonce(addr)

        // Drop all transactions that are deemed too old (low nonce)
        for _, tx := range list.Forward(nonce) {
            hash := tx.Hash()
            log.Trace( msg: "Removed old pending transaction",  ctx: "hash", hash)
            delete(pool.all, hash)
            pool.priced.Removed()
        }
        // Drop all transactions that are too costly (low balance or out of gas), and queue any invalids
        drops, invalids := list.Filter(pool.currentState.GetBalance(addr), pool.currentMaxGas)
        for _, tx := range drops {
            hash := tx.Hash()
            log.Trace( msg: "Removed unpayable pending transaction",  ctx: "hash", hash)
            delete(pool.all, hash)
            pool.priced.Removed()
            pendingNofundsCounter.Inc(1)
        }
        for _, tx := range invalids {
            hash := tx.Hash()
            log.Trace( msg: "Demoting pending transaction",  ctx: "hash", hash)
            pool.enqueueTx(hash, tx)
        }
        // If there's a gap in front, warn (should never happen) and postpone all transactions
        if list.Len() > 0 && list.txs.Get(nonce) == nil {
            for _, tx := range list.Cap( threshold: 0) {
                hash := tx.Hash()
                log.Error( msg: "Demoting invalidated transaction",  ctx: "hash", hash)
                pool.enqueueTx(hash, tx)
            }
        }
        // Delete the entire queue entry if it became empty.
        if list.Empty() {
            delete(pool.pending, addr)
            delete(pool.beats, addr)
        }
    }
}
```

Promote Executables

```
// Iterate over all accounts and promote any executable transactions
for _, addr := range accounts {
    list := pool.queue[addr]
    if list == nil {
        continue // Just in case someone calls with a non existing account
    }
    // Drop all transactions that are deemed too old (low nonce)
    for _, tx := range list.Forward(pool.currentState.GetNonce(addr)) {
        hash := tx.Hash()
        log.Trace( msg: "Removed old queued transaction", ctx: "hash", hash)
        delete(pool.all, hash)
        pool.priced.Removed()
    }
    // Drop all transactions that are too costly (low balance or out of gas)
    drops, _ := list.Filter(pool.currentState.GetBalance(addr), pool.currentMaxGas)
    for _, tx := range drops {
        hash := tx.Hash()
        log.Trace( msg: "Removed unpayable queued transaction", ctx: "hash", hash)
        delete(pool.all, hash)
        pool.priced.Removed()
        queuedNofundsCounter.Inc(1)
    }
    // Gather all executable transactions and promote them
    for _, tx := range list.Ready(pool.pendingState.GetNonce(addr)) {
        hash := tx.Hash()
        log.Trace( msg: "Promoting queued transaction", ctx: "hash", hash)
        pool.promoteTx(addr, hash, tx)
    }
    // Drop all transactions over the allowed limit
    if !pool.locals.contains(addr) {
        for _, tx := range list.Cap(int(pool.config.AccountQueue)) {
            hash := tx.Hash()
            delete(pool.all, hash)
            pool.priced.Removed()
            queuedRateLimitCounter.Inc(1)
            log.Trace( msg: "Removed cap-exceeding queued transaction", ctx: "hash", hash)
        }
    }
    // Delete the entire queue entry if it became empty.
    if list.Empty() {
        delete(pool.queue, addr)
    }
}
```

Promote Tx

```go
// promoteTx adds a transaction to the pending (processable) list of transactions.
//
// Note, this method assumes the pool lock is held!
func (pool *TxPool) promoteTx(addr common.Address, hash common.Hash, tx *types.Transaction) {
    // Try to insert the transaction into the pending queue
    if pool.pending[addr] == nil {
        pool.pending[addr] = newTxList( strict: true)
    }
    list := pool.pending[addr]

    inserted, old := list.Add(tx, pool.config.PriceBump)
    if !inserted {
        // An older transaction was better, discard this
        delete(pool.all, hash)
        pool.priced.Removed()

        pendingDiscardCounter.Inc(1)
        return
    }
    // Otherwise discard any previous transaction and mark this
    if old != nil {
        delete(pool.all, old.Hash())
        pool.priced.Removed()

        pendingReplaceCounter.Inc(1)
    }
    // Failsafe to work around direct pending inserts (tests)
    if pool.all[hash] == nil {
        pool.all[hash] = tx
        pool.priced.Put(tx)
    }
    // Set the potentially new pending nonce and notify any subsystems of the new tx
    pool.beats[addr] = time.Now()
    pool.pendingState.SetNonce(addr, tx.Nonce()+1)

    go pool.txFeed.Send(TxPreEvent{ Tx: tx})
}
```

Discussion