Transaction Mempool

Objectives

1. To provide necessary methods to **create and close mempool**
2. To provide necessary methods to **add or drop transactions** to/from the mempool
3. To provide necessary methods to **validate transactions** and decide whether to kick them or not.
4. To provide necessary methods to help block proposer (will be searchers in MEV-Boost case) to **compose a transaction batch from the mempool**

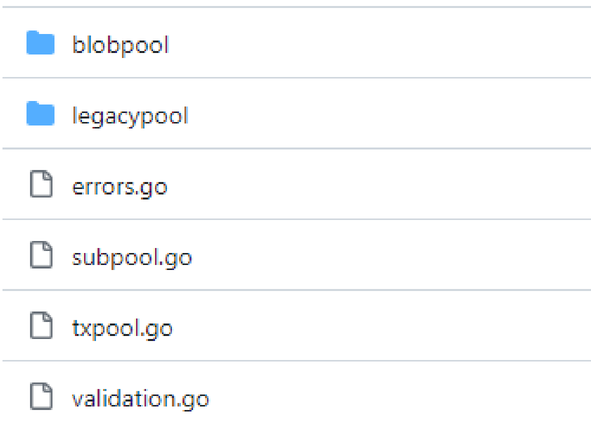
High Level Overview

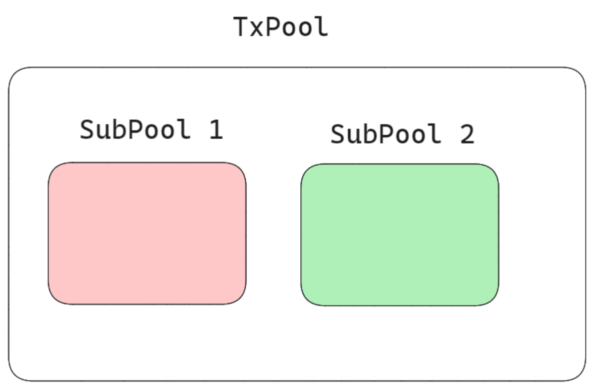
- Mempool is a place where pending transactions stay before they are executed.
- Creating a mempool means **creating a pool structure**, **maintaining a list of transactions**, and **running a goroutine** that subscribes the addition of new block and updates the state of mempool
- After EIP-4844, there comes “blob transaction”, whose syntax and validation logic is quite different from legacy transactions.
- So the mempool should also be separated from the legacy mempool.
- For this purpose, geth enables the client to create various transaction-specific pools, called `subpool` .
- Subpools share the same interface and managed by primary transaction pool (`txpool.go`), since they have to be updated and assembled into one conherent view for block production.
- Every pool **MUST** implement all the methods in `subpool.go` ⇒ Every pool is an instance of subpool interface.

Currently two pools are implemented like this way: `blobpool` and `legacypool`. Any additional tx type can generate extra txpool.

For instance, there will be `aapool` if native account abstraction is implemented on Ethereum.

(Look at core/txpool)





Code Review

- Let’s take a look at subpool interface first.

core/txpool/subpool.go

```
type SubPool interface {
    // Filter is a selector used to decide whether a transaction whould be added
    // to this particular subpool.
    Filter(tx *types.Transaction) bool

    // Init sets the base parameters of the subpool, allowing it to load any saved
    // transactions from disk and also permitting internal maintenance routines to
    // start up.
    //
```

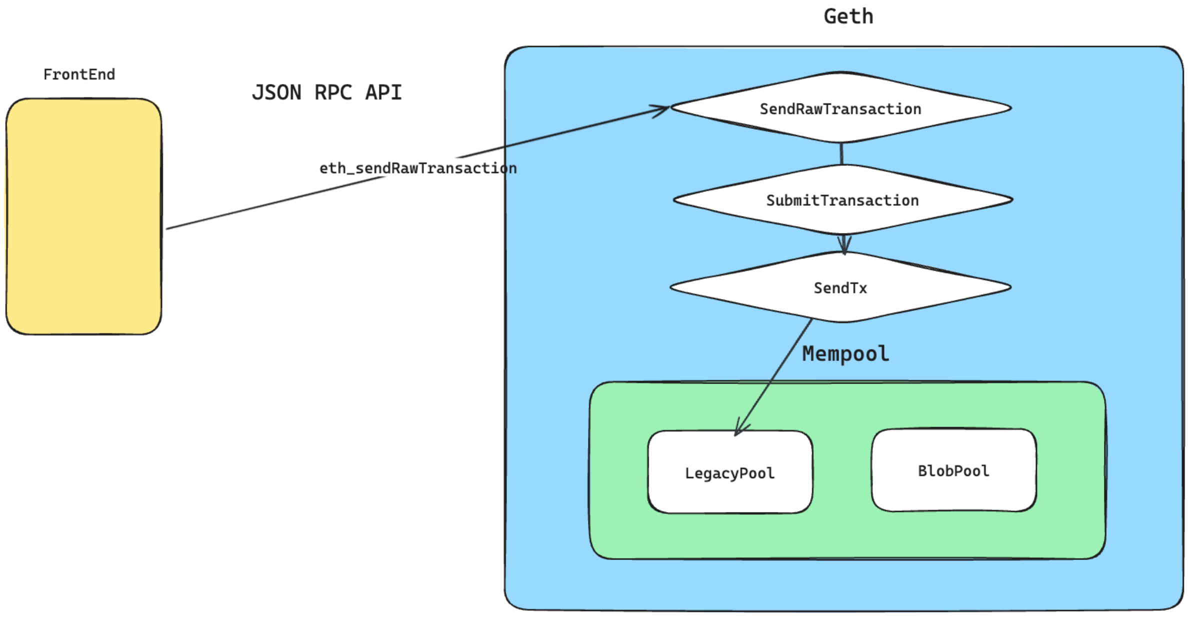
```
// These should not be passed as a constructor argument - nor should the poo
// start by themselves - in order to keep multiple subpools in lockstep with
// one another.
Init(gasTip *big.Int, head *types.Header, reserve AddressReserver) error

// Close terminates any background processing threads and releases any held
// resources.
Close() error

// Reset retrieves the current state of the blockchain and ensures the content
// of the transaction pool is valid with regard to the chain state.
Reset(oldHead, newHead *types.Header)

// SetGasTip updates the minimum price required by the subpool for a new
// transaction, and drops all transactions below this threshold.
SetGasTip(tip *big.Int)

// Has returns an indicator whether subpool has a transaction cached with the
// given hash.
Has(hash common.Hash) bool

// Get returns a transaction if it is contained in the pool, or nil otherwise.
Get(hash common.Hash) *types.Transaction

// Add enqueues a batch of transactions into the pool if they are valid. Due
// to the large transaction churn, add may postpone fully integrating the tx
// to a later point to batch multiple ones together.
Add(txs []*types.Transaction, local bool, sync bool) []error

// Pending retrieves all currently processable transactions, grouped by origin
// account and sorted by nonce.
Pending(enforceTips bool) map[common.Address][]*LazyTransaction

// SubscribeTransactions subscribes to new transaction events. The subscribe
// can decide whether to receive notifications only for newly seen transactions
// or also for reorged out ones.
```

```
    SubscribeTransactions(ch chan<- core.NewTxsEvent, reorgs bool) event.Subs

    // Nonce returns the next nonce of an account, with all transactions executable
    // by the pool already applied on top.
    Nonce(addr common.Address) uint64

    // Stats retrieves the current pool stats, namely the number of pending and the
    // number of queued (non-executable) transactions.
    Stats() (int, int)

    // Content retrieves the data content of the transaction pool, returning all the
    // pending as well as queued transactions, grouped by account and sorted by
    Content() (map[common.Address][]*types.Transaction, map[common.Address

    // ContentFrom retrieves the data content of the transaction pool, returning the
    // pending as well as queued transactions of this address, grouped by nonce.
    ContentFrom(addr common.Address) ([]*types.Transaction, []*types.Transacti

    // Locals retrieves the accounts currently considered local by the pool.
    Locals() []common.Address

    // Status returns the known status (unknown/pending/queued) of a transaction
    // identified by their hashes.
    Status(hash common.Hash) TxStatus
}
```

- There are methods that creates / closes subpool, add / filter transactions inside the subpool, and retrieve status of transactions or the subpool.
- I'll introduce two of widely-used functions, `Add` & `Reset`.

`Add` & `Reset`

1. **`Add(txs []*types.Transaction, local bool, sync bool) []error`**



- This is a function that adds a set of transactions to the subpool.

▼ code

- When you send tx through JSON RPC API of Infura, Alchemy, or whatever, you are calling `eth_sendTransaction` or `eth_sendRawTransaction` method.
- Both methods are at [internal/ethapi/api.go], and they are implemented like the following code.

```
func (s *TransactionAPI) SendRawTransaction(ctx context.Context, in
    tx := new(types.Transaction)
    if err := tx.UnmarshalBinary(input); err != nil {
        return common.Hash{}, err
    }
    return SubmitTransaction(ctx, s.b, tx)
}
```

and they calls `SubmitTransaction` method, which triggers `SendTx` method at [eth/api_backend.go].

```
func (b *EthAPIBackend) SendTx(ctx context.Context, signedTx *type
    return b.eth.txPool.Add([]*types.Transaction{signedTx}, true, false
}
```

- Add function is implemented like the following code in legacypool.

```
func (pool *LegacyPool) Add(txs []*types.Transaction, local, sync boo
    // Filter out known ones without obtaining the pool lock or recoveri
    var (
        errs = make([]error, len(txs))
        news = make([]*types.Transaction, 0, len(txs))
    )
    for i, tx := range txs {
        // If the transaction is known, pre-set the error slot
        if pool.all.Get(tx.Hash()) != nil {
            errs[i] = ErrAlreadyKnown
            knownTxMeter.Mark(1)
            continue
        }
        // Exclude transactions with basic errors, e.g invalid signatures a
        // insufficient intrinsic gas as soon as possible and cache sender
        // in transactions before obtaining lock
        if err := pool.validateTxBasics(tx, local); err != nil {
            errs[i] = err
            invalidTxMeter.Mark(1)
            continue
        }
        // Accumulate all unknown transactions for deeper processing
        news = append(news, tx)
    }
    if len(news) == 0 {
        return errs
    }
```

```
    // Process all the new transaction and merge any errors into the ori
    pool.mu.Lock()
    newErrs, dirtyAddrs := pool.addTxsLocked(news, local)
    pool.mu.Unlock()

    var nilSlot = 0
    for _, err := range newErrs {
        for errs[nilSlot] != nil {
            nilSlot++
        }
        errs[nilSlot] = err
        nilSlot++
    }
    // Reorg the pool internals if needed and return
    done := pool.requestPromoteExecutables(dirtyAddrs)
    if sync {
        <-done
    }
    return errs
}
```

- It executes a for loop for every transactions.
- First, it validates if the transaction meets all requirements (valid signature, sufficient gas, ...)
- And it uses mutex to avoid errors from concurrency:
- Inside the mutex lock and unlock, `addTxsLocked` function adds the transaction into the pool. Since using mutex means that there is somewhere using goroutine inside the `addTxsLocked` function.

  ```
  func (pool *LegacyPool) addTxsLocked(txs []*types.Transaction,
      dirty := newAccountSet(pool.signer)
      errs := make([]error, len(txs))
      for i, tx := range txs {
  ```

```
        replaced, err := pool.add(tx, local)
        errs[i] = err
        if err == nil && !replaced {
            dirty.addTx(tx)
        }
    }
    validTxMeter.Mark(int64(len(dirty.accounts)))
    return errs, dirty
}
```

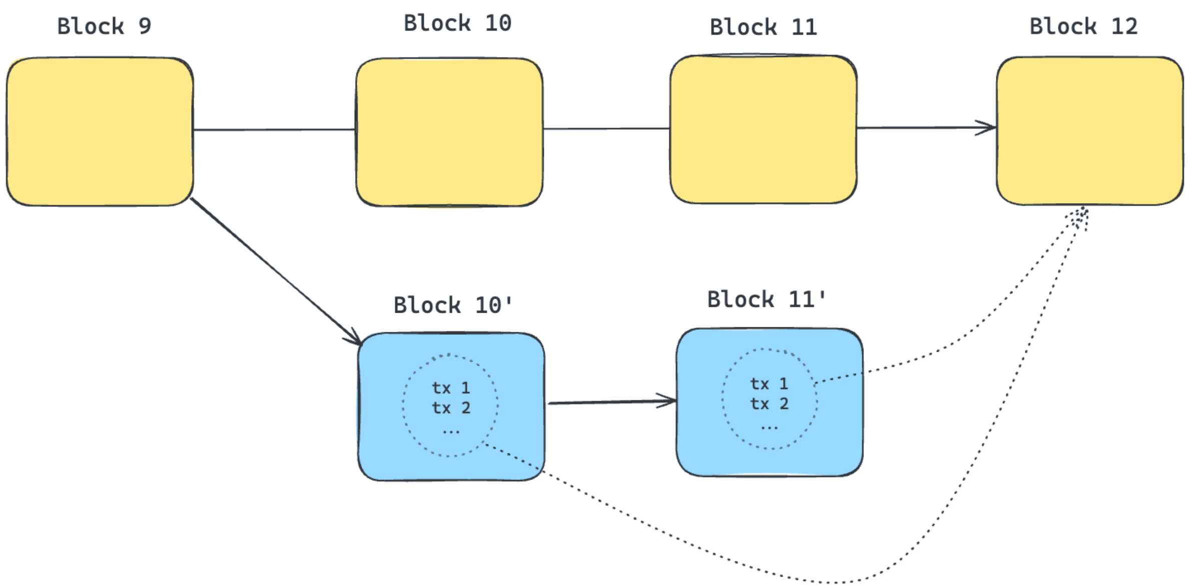
Unfortunately, I failed to find the exact place where the concurrency matters :(

IMO, maybe this line matters:

`if uint64(pool.all.Slots()+numSlots(tx))...`

This line checks the current slots of the subpool. If the pool is full, it drops the transaction. And, the slot data is a global state of each subpool. If each transaction is executed in a concurrent manner, checking in this line may produce unexpected error.

2. `Reset(oldHead, newHead *types.Header)`
    - Reset retrieves the current state of the blockchain and ensures the content of the transaction pool is valid with regard to the chain state.



- Think of a situation of reorg: Transactions in the ‘orphan blocks’ need to be back to the mempool.
- `Reset` updates the content of the pool with the given `newHead` which means a header of the new block, bringing all txs in the re-orged blocks.

▼ code

```
func (pool *LegacyPool) reset(oldHead, newHead *types.Header) {
    // If we're reorging an old state, reinject all dropped transactions
    var reinject types.Transactions

    if oldHead != nil && oldHead.Hash() != newHead.ParentHash {
        // If the reorg is too deep, avoid doing it (will happen during fast sync
        oldNum := oldHead.Number.Uint64()
        newNum := newHead.Number.Uint64()

        if depth := uint64(math.Abs(float64(oldNum) - float64(newNum))); d
            log.Debug("Skipping deep transaction reorg", "depth", depth)
        } else {
            // Reorg seems shallow enough to pull in all transactions into mem
            var (
                rem = pool.chain.GetBlock(oldHead.Hash(), oldHead.Number.Ui
```

```
        add = pool.chain.GetBlock(newHead.Hash(), newHead.Number.
    )
    if rem == nil {
        // This can happen if a setHead is performed, where we simply
        // head from the chain.
        // If that is the case, we don't have the lost transactions anymor
        // there's nothing to add
        if newNum >= oldNum {
            // If we reorged to a same or higher number, then it's not a ca
            log.Warn("Transaction pool reset with missing old head",
                "old", oldHead.Hash(), "oldnum", oldNum, "new", newHead
            return
        }
        // If the reorg ended up on a lower number, it's indicative of setl
        log.Debug("Skipping transaction reset caused by setHead",
            "old", oldHead.Hash(), "oldnum", oldNum, "new", newHead.H
        // We still need to update the current state s.th. the lost transact
    } else {
        if add == nil {
            // if the new head is nil, it means that something happened be
            // the firing of newhead-event and _now_: most likely a
            // reorg caused by sync-reversion or explicit sethead back to
            // earlier block.
            log.Warn("Transaction pool reset with missing new head", "nu
            return
        }
        var discarded, included types.Transactions
        for rem.NumberU64() > add.NumberU64() {
            discarded = append(discarded, rem.Transactions()...)
            if rem = pool.chain.GetBlock(rem.ParentHash(), rem.NumberU
                log.Error("Unrooted old chain seen by tx pool", "block", old
                return
            }
        }
        for add.NumberU64() > rem.NumberU64() {
            included = append(included, add.Transactions()...)
```

```
            if add = pool.chain.GetBlock(add.ParentHash(), add.NumberU
                log.Error("Unrooted new chain seen by tx pool", "block", ne
                return
            }
        }
        for rem.Hash() != add.Hash() {
            discarded = append(discarded, rem.Transactions()...)
            if rem = pool.chain.GetBlock(rem.ParentHash(), rem.NumberU
                log.Error("Unrooted old chain seen by tx pool", "block", old
                return
            }
            included = append(included, add.Transactions()...)
            if add = pool.chain.GetBlock(add.ParentHash(), add.NumberU
                log.Error("Unrooted new chain seen by tx pool", "block", ne
                return
            }
        }
        lost := make([]*types.Transaction, 0, len(discarded))
        for _, tx := range types.TxDifference(discarded, included) {
            if pool.Filter(tx) {
                lost = append(lost, tx)
            }
        }
        reinject = lost
    }
  }
}
// Initialize the internal state to the current head
if newHead == nil {
    newHead = pool.chain.CurrentBlock() // Special case during testing
}
statedb, err := pool.chain.StateAt(newHead.Root)
if err != nil {
    log.Error("Failed to reset txpool state", "err", err)
    return
}
```

```
    pool.currentHead.Store(newHead)
    pool.currentState = statedb
    pool.pendingNonces = newNoncer(statedb)

    costFn := types.NewL1CostFunc(pool.chainconfig, statedb)
    pool.l1CostFn = func(dataGas types.RollupGasData) *big.Int {
        return costFn(newHead.Number.Uint64(), dataGas, false)
    }

    // Inject any transactions discarded due to reorgs
    log.Debug("Reinjecting stale transactions", "count", len(reinject))
    core.SenderCacher.Recover(pool.signer, reinject)
    pool.addTxsLocked(reinject, false)
}
```

- It includes all transactions in the orphan blocks in `reinject` array, and calls `addTxsLocked` to add them inside the mempool.

`core/txpool/txpool.go`

- So how is a mempool created? `txpool.go` provides a method called `New` that allows a client to start the mempool.

▼ code

- `New`

```
func New(gasTip *big.Int, chain BlockChain, subpools []SubPool) (*Tx
    // Retrieve the current head so that all subpools and this main coorc
    // pool will have the same starting state, even if the chain moves for
    // during initialization.
    head := chain.CurrentBlock()

    pool := &TxPool{
        subpools:     subpools,
        reservations: make(map[common.Address]SubPool),
```

```
        quit:        make(chan chan error),
    }
    for i, subpool := range subpools {
        if err := subpool.Init(gasTip, head, pool.reserver(i, subpool)); err !
            for j := i - 1; j >= 0; j-- {
                subpools[j].Close()
            }
            return nil, err
        }
    }
    go pool.loop(head, chain)
    return pool, nil
}
```

`loop` **function**

```
func (p *TxPool) loop(head *types.Header, chain BlockChain) {
    // Subscribe to chain head events to trigger subpool resets
    var (
        newHeadCh  = make(chan core.ChainHeadEvent)
        newHeadSub = chain.SubscribeChainHeadEvent(newHeadCh)
    )
    defer newHeadSub.Unsubscribe()

    // Track the previous and current head to feed to an idle reset
    var (
        oldHead = head
        newHead = oldHead
    )
    // Consume chain head events and start resets when none is runnin
    var (
        resetBusy = make(chan struct{}, 1) // Allow 1 reset to run concurr
        resetDone = make(chan *types.Header)
    )
    var errc chan error
    for errc == nil {
```

```
        // Something interesting might have happened, run a reset if there
        // one needed but none is running. The resetter will run on its own
        // goroutine to allow chain head events to be consumed contiguo
        if newHead != oldHead {
            // Try to inject a busy marker and start a reset if successful
            select {
            case resetBusy <- struct{}{}:
                // Busy marker injected, start a new subpool reset
                go func(oldHead, newHead *types.Header) {
                    for _, subpool := range p.subpools {
                        subpool.Reset(oldHead, newHead)
                    }
                    resetDone <- newHead
                }(oldHead, newHead)

            default:
                // Reset already running, wait until it finishes
            }
        }
        // Wait for the next chain head event or a previous reset finish
        select {
        case event := <-newHeadCh:
            // Chain moved forward, store the head for later consumption
            newHead = event.Block.Header()

        case head := <-resetDone:
            // Previous reset finished, update the old head and allow a new
            oldHead = head
            <-resetBusy

        case errc = <-p.quit:
            // Termination requested, break out on the next loop round
        }
    }
    // Notify the closer of termination (no error possible for now)
```

```
    errc <- nil
}
```

- There is a goroutine inside the `loop` function, which subscribes to the block header change with `Reset` function.
- So if chain reorg happens, the client will include all txs in the orphan blocks.