BlockChain Protocols & Concurrency Bugs

Huan Ke, Haryadi Gunawi
Outline

BlockChain Protocols:
- Transact
- WriteBlock
- Mining
- Sync
- Consensus

Concurrency Bugs:
- Local concurrency bugs (1)
- Distributed concurrency bugs (5)
Send Transaction

The basic way of sending a simple transaction is as follows:

```javascript
eth.sendTransaction({
  from: sender,
  to: receiver,
  value: amount
})
```
Send Transaction

SendTransaction

SubmitTransaction

Txpool

pending Transactions

pending

all

queue

1

1

2: promoteExecutables
Pending Transaction

```javascript
> eth.pendingTransactions
{
  blockHash: null,
  blockNumber: null,
  from: "0x16c0a3f6832f4b3bb0a6b4e0ca15635da91c588e",
  gas: 90000,
  gasPrice: 18000000000,
  hash: "0x8bab394b6dc22a322b35586ffca184662a35fd150bbc389c4561169b8baf77db",
  input: "0x",
  nonce: 0,
  r: "0x4c6ecdebdeb07372fe0ba8dc70db871c923a2f97b96fc7105dc49de46f5a5d",
  s: "0x14fcb125067382b47a531ad7af73cb60102f1469773745c1330ba6b45b9c55e3",
  to: null,
  transactionIndex: 0,
  v: "0x7da",
  value: 1
}
Transaction Receipt

root: the hash of the root of the state trie
receptRoot: the hash of the array of receipts for a given block.
Send Transaction

- SendTransaction
- SubmitTransaction

1: all
2: promoteExecutables
3: Post TxPreEvent

Txpool

pending
Transact

`txBroadcastLoop`: send tx message to peers.
Transact

Receiver:

- Mark a transaction as known to the peer, ensuring that it will never be propagated to this particular peer.
- TxMsg: add the transactions to the local txpool.
- post TxPreEvent, which trigger the tx to be broadcasted to unknown peer.
Transact

```
peer.MarkTransaction(tx.hash())
```

```
txpool.AddRemote()
```

```
TxMsg
```

```
Txpool
```
Transact

TxMsg

peer.MarkTransaction(tx.hash())
txpool.AddRemote()

3: Post TxPreEvent

Txpool

all

queue

2: promoteExecutables

pending

0
Block

Block Header

Block Body

Block

Block Header (Block Hash)

Prev Hash

Nonce

Root Hash

Hash01

Hash23

Hash0

Hash1

Hash2

Hash3

Tx0

Tx1

Tx2

Tx3
WriteBlock

BlockChain represent the canonical chain given a database with a genesis block.
WriteBlock

BlockChain represent the canonical chain given a database with a genesis block.
**WriteBlock**

BlockChain represent the canonical chain given a database with a genesis block.

Write block body and header into database

injects a new block into the current blockchain
Start Mining

The basic way of start mining is as follows:

- miner.start()

Miner creates blocks and searches for proof of work values
Woker

Woker is the main object which takes care of applying messages to the new state
CpuAgent

The basic way of start mining is as follows:

- miner.start(4)
Mining

Miner.start()        Worker              Agent

post NewMinedBlock

minedBroadcastLoop
Mining

\texttt{minedBroadcastLoop}: send new mined block message to peers.
Mining

Receiver:

- **NewBlockMsg** → mark the block as present at the remote peer; update the peers total difficulty if better than the previous.

- **NewBlockHashesMsg** → mark the hashes as present at the remote peer; schedule all the hashes for retrieval; announce the fetcher of the potential availability of a new block in the network.
Mining: NewBlockMsg

NewBlockMsg

peer.MarkBlock(block.Hash))

catcher.Enqueue.(peer, block)

update peer td based on block
Fetcher

Fetcher is responsible for accumulating block announcement from various peers and schedule them for retrieval.
Mining: NewBlockMsg

NewBlockMsg

fetcher.Enqueue.(peer, block)

fetcher.inject channel

fetcher.queue
Mining: NewBlockMsg

NewBlockMsg

fetcher.Enqueue.(peer, block)

fetcher.inject channel

fetcher.loop()
Mining: NewBlockMsg

NewBlockMsg

fetcher.Enqueue.(peer, block)

fetcher.inject channel

fetcher.queue

fetcher.loop()

validate the header and propagate the block if it it passes

broadcastBlock
Mining: NewBlockHashesMsg

NewBlockHashesMsg

p.MarkBlock(block.Hash)

if unknown:
    fetcher.Notify(peer, blockHash)
Mining: NewBlockHashesMsg

```
fetcher.loop()
```
Mining: NewBlockHashesMsg

if the block still didn’t arrive, queue for fetching, send out block header/body request
Synchronisation

Synchronisation modes:

- Full Sync
- Fast Sync

Two sync handler:

- syncer
- ttxsyncLoop
Synchronization: Syncer

Periodically synchronizing with the network, both downloading the hashes and blocks as well as the announcement handler.

- when there is a new peer, it will synchronise with the best peer.
- every 10s, it will synchronise with the best peer.
Synchronization: txsyncLoop

It takes care of the initial tx sync for each new connection. When a new peer appears, we relay all currently pending transactions. In order to minimise egress bandwidth usage, we send the transactions in small packs to one peer at a time.
Synchronization

- PeerSet
- Fetcher
- Downloader
Downloader

It fetches hashes and blocks from remote peers.

- queue: scheduling for selecting the hashes to download
- peers: set of active peers from which download can proceed.
- syncStatsChainOrigin/syncStatsChainHeight
- syncStatsState
- lightChain/blockChain
- dropPeer: drop a peer for misbehaving
- headerCh/bodyCh/receiptCh
- stateSyncStart/trackStateReq/stateCh
Synchronization

Downloader.synchronise(
  ● peerId
  ● peerHead
  ● peerTd)
Synchronization

Downloader.synchronise(
  ● peerId
  ● peerHead
  ● peerTd)

GetBlockHeaderMsg
Synchronization

Downloader.synchronise(
  ● peerId
  ● peerHead
  ● peerTd)

BlockHeaderMsg

GetBlockHeaderMsg
Synchronisation: Full/Fast

- **download block header and body.** It’s almost the same as block inserting, including header validation, tx validation and tx processing, account state transition. It will consume the CPU and disk.

- **download block header, body and receipt.** It will not process transaction during insertion. sync the state with a block height, afterwards, it will exploit full mode to construct. It will speed up and won’t generate a lot of history info to reduce disk consumption but takes more network resources because it downloading receipts and states.
Synchronization

fetchHeaders
fetchBodies

Fast

processFastSyncContent

Full

processFullSyncContent
fetcheReceipts
ProcessSyncContent

FastSyncPivot: block number where the fast sync pivots into archive synchronisation mode.

- $(\text{max\_block} - (64 + \text{rand}(0, 256)))$ or something like that

$$P, \text{before}P, \text{after}P = \text{splitAroundPivot}(\text{pivot, results})$$

- commitFastSyncData(beforeP, stateSync) (Fast)
- commitPivotBlock(P) (Fast)
- importBlockResults(afterP) (Fast/Full)
State

Ethereum is an account-based blockchain. The world state is a mapping between address and account states. Though not stored on the blockchain, it is assumed that the implementation will maintain this mapping in a modified Merkle Patricia tree.
State Trie
Fast Sync

processFastSync → Downloader

Downloader → StateFetcher
Fast Sync

1. processFastSync
2. stateSyncStart

Downloader

StateFetcher
Fast Sync

1. Downloader

2. stateSyncStart

3. RunStateSync

processFastSync

StateFetcher

handler

run()
RunStateSync

Run → it’s responsible for the assignment of new tasks to peers as well as for the processing of inbound data.
**RunStateSync**

Run → it’s responsible for the assignment of new tasks to peers as well as for the processing of inbound data.
RunStateSync

Run → it’s responsible for the assignment of new tasks to peers as well as for the processing of inbound data.
RunStateSync

Run → it’s responsible for the assignment of new tasks to peers as well as for the processing of inbound data.

Process state request.
Consensus (PoW)

Consensus achieved using Proof-of-Work.

- New transactions are broadcasted to all nodes.
- Each node collects new transactions into a block.
- Each agent works on finding a difficult proof of work for its block.
- When a node finds a PoW, it broadcast the block to all nodes.
Consensus (PoW)

Consensus achieved using Proof-of-Work.

- Nodes accept the block only if all transactions in it are valid and not already spent.
- Nodes always consider the longest chain to be the correct one and will keep working on extending it.
Consensus (PoW)

Miner.start()

step1: start()

step2: commitNewWork

post NewMinedBlock

PoW: cryptographic puzzle

Proof of Work

Cryptographic Hash

irreversible: f(?) = 356
f(5) = 43 not 356, try again
f(300) = 685 try again
f(34) = 59 try again
f( ) < threshold?
Before the fork
Two Blocks Simultaneously
Two Blocks Propagate
A New Block Extends one Fork
Re-converges on Longest Chain
Re-converges on Longest Chain
Re-converges on Longest Chain

The community adopts 6 blocks as standard confirmation period.
Concurrency Bugs

**data race bugs** → two conflicting accesses to one shared variable are executed without proper synchronization, e.g., not protected by a common lock.

**deadlock bugs** → two or more operations circularly wait for each other to release the acquired resource (e.g., locks)

**Atomicity-violation bugs** → concurrent execution unexpectedly violating the atomicity of a certain code region.
Local Currency Bug

```
parent := self.chain.GetBlock(current.block.ParentHash())

if parent == nil {
    // in case geth quits the blockchain db can be closed and this returns nil
    return
}

current.coinbase.SetGasPool(core.CalcGasLimit(parent))
```
Distributed Concurrency Bug 1

Local Mining

Remote Importing

WriteBlock
Distributed Concurrency Bug 1

Local Mining

Remote Importing
Distributed Concurrency Bug 2

pendingState=50
currentState=49

reset

Txpool

pendingState=51
currentState=50

pendingState=currentState
pendingState.Nonce=PendingTx.Nonce+1
Distributed Concurrency Bug 2

pendingState=50
currentState=49

reset

Txpool

Tx: 50
pendingState=50
currentState=49

Tx: 51
pendingState=51
currentState=51

pendingState=currentState
pendingState.Nonce=PendingTx.Nonce+1
Distributed Concurrency Bug 3

block timestamp > receiver node’s timestamp

futureBlocks
Distributed Concurrency Bug 3

block timestamp > receiver node’s timestamp

```go
def futureBlocks:
    blocks := make([]types.Block, self.futureBlocks.Len())
    for i, hash := range self.futureBlocks.Keys() {
        block, _ := self.futureBlocks.Get(hash)
        blocks[i] = block.(*types.Block)
```
Distributed Concurrency Bug 3

**block timestamp > receiver node’s timestamp**

**futureBlocks**

```go
blocks := make([]types.Block, self.futureBlocks.Len())
for i, hash := range self.futureBlocks.Keys() {
    block, _ := self.futureBlocks.Get(hash)
    blocks[i] = block.(types.Block)
}
```

**block timestamp < receiver node’s timestamp**
Distributed Concurrency Bug 3

block timestamp > receiver node’s timestamp

futureBlocks

block timestamp < receiver node’s timestamp

blocks := make([]types.Block, 0, self.futureBlocks.Len())
for _, hash := range self.futureBlocks.Keys() {
    if block, exist := self.futureBlocks.Get(hash); exist {
        blocks = append(blocks, block.(*types.Block))
    }
}
Distributed Concurrency Bug 4

block timestamp > receiver node’s timestamp

futureBlocks

```
func (self *ChainManager) procFutureBlocks() {
    blocks := make([]types.Block, len(self.futureBlocks.blocks))
    self.futureBlocks.Each(func(i int, block *types.Block) {
        blocks[i] = block
    })
}```
Distributed Concurrency Bug 4

block timestamp > receiver node’s timestamp

futureBlocks

```
func (self *ChainManager) procFutureBlocks() {
    blocks := make([]*types.Block, len(self.futureBlocks.blocks))
    self.futureBlocks.Each(func(i int, block *types.Block) {
        blocks[i] = block
    })
}
```
Distributed Concurrency Bug 4

block timestamp > receiver node’s timestamp

futureBlocks

n+1

```go
func (self *ChainManager) procFutureBlocks() {
    blocks := make([]types.Block, len(self.futureBlocks.blocks))
    self.futureBlocks.Each(func(i int, block *types.Block) {
        blocks[i] = block
    })
}
```
Distributed Concurrency Bug 5
Distributed Concurrency Bug 5

PeerDrop

Request

cancel stateSync Task
Distributed Concurrency Bug 5

PeerDrop

Request

cancel stateSync Task
Distributed Concurrency Bug 5

PeerDrop

Request

new StateSync Task

cancel stateSync Task
Distributed Concurrency Bug 5

```go
case req := <-timeout:
    if active[req.peer.id] != req {
        continue
    }

    finished = append(finished, req)
    delete(active, req.peer.id)
```

Request

PeerDrop

new StateSync Task

cancel stateSync Task
Thank you!
Questions?

http://ucare.cs.uchicago.edu