A CONCURRENT PERSPECTIVE ON BLOCKCHAIN

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
HUAN KE

CHICAGO, ILLINOIS
JUNE 2018
# LIST OF FIGURES

1.1 **How does blockchain change the way we transact.** ........................................... v

2.1 **Architecture of Ethereum.** ................................................................. ix

2.2 **Transaction Authentication.** .............................................................. x

2.3 **Block Header.** ................................................................................... xii

2.4 **State Trie.** ........................................................................................ iii

2.5 **Blockchain Data Structure.** ............................................................... xiv

3.1 **How blockchain works.** ................................................................. xxvi

3.2 **Transaction Protocols.** ................................................................. xxvii

3.3 **Mining Protocol.** ............................................................................. xxxii

3.4 **Sync Protocol.** ............................................................................... xxxiii

3.5 **Blockchain Fork.** ........................................................................... xxxv

4.1 **Statistical Overview.** ................................................................. xxxvii

4.2 **Blockchain bug distribution.** ........................................................... xxxviii

4.3 **Transfer Money.** ........................................................................... lxiv
ABSTRACT

With the widespread of blockchain technology, its vulnerable to bugs that often occur in traditional concurrent programs. We study 63 concurrency bugs from three blockchain based systems: Go-Ethereum, Cpp-Ethereum, and Rust-Ethereum regarding to bugs triggering, symptoms, fix and violated specifications. It systematically checks the concurrency vulnerabilities of Blockchain systems and categorize the concurrency specifications, which acts (1) as a reference for developers to know and avoid pitfalls during implementing blockchain protocol and (2) as a guide for researchers to foster the development of analysis and verification for blockchain systems.
CHAPTER 1
INTRODUCTION

With the widespread of blockchain, it shows extraordinary talents in economic. By design, blockchain is a decentralized technology. It uses a distributed ledger that records the transactions, such as the exchange of assets or data, among the participants in the network. Participants govern and agree by consensus on the updates to the records in the ledger. No central authority or third-party mediator, e.g., a financial institution, is involved. Regarding to this, blockchain is essentially a mechanism to bring everyone to the highest degree of accountability. Looking back to the blockchain history, Satoshi Nakamoto (a pseudonym) introduced Bitcoin\[7\], a peer-to-peer electronic cash system in 2008. It allows for online payments from one user to another without going through the trusted central bank system. In 2015, Vitalik Buterin established Ethereum Foundation to create Ethereum\[2\], which is a general blockchain platform for decentralized applications. Figure 1.1 explicitly describe how the blockchain platform Ethereum change the way we transact.

![Figure 1.1: How does blockchain change the way we transact.](image_url)

As a revolutionary technology, blockchain is attracting people’s attention. More and more
blockchain based systems come out. However, they exists some inequalities and vulnerabilities. Enormous reports about money stolen from the Bitcoin/Flexcoin fear users. The double spend attack [6] is inevitable in blockchain. Regarding to these securities, [4] empirically studied the attacks in the Ethereum system. Likewise, [10] investigated all kinds of bugs in blockchain. As a concurrent system, blockchain is encountering enormous concurrency issues as the traditional systems. From the perspective of implementation, Geth are suffering from almost 200 bugs that miss locks and forget to unlock. Most of them can lead to concurrency bugs, resulting in transaction missing, node crash, peer disconnection and so on. In addition to it, there are 604 concurrency reports based on the key words such as ”race”, ”deadlock”, ”concurrent”, ”atomic”, and ”lock”. From the perspective of blockchain protocol, it contains many internal concurrency characteristics, which bring a lot of new concurrency issues. For example, Geth and Parity reported about 2000 issues when blockchain is handling synchronization. However, no one uncovers synchronization’s internal concurrency and the protocol design weakness. All in all, this young blockchain technology is warning us to pay attention to the concurrency in the blockchain based systems.

In order to study the concurrency, we start with the specifications in blockchain. For any concurrency bug, it must violate the specifications to some extend. Surprisingly, we found the Bitcoin has no specification reference. If anyone tries to make a Bitcoin implementation using modern practices or languages that didn’t exist when Satoshi created bitcoin, he is referring to existing codebase to implement. The lacking of specifications make our project more difficult but more rewarding. On the other hand, Ethereum system provides the specification in its yellow paper [11], which provides a formal specification of the system. Indeed, this was the first formal specification of any blockchain protocol. However, in #2586 developers argue that current specifications should change; #596 updated specification for eth call RPC method; #2397 mentioned the specification is wrong. As we see, the current Ethereum specifications are not 100% correct. We can’t depend on wrong specification to detect concurrency bugs.

Furthermore, we notice that there aren’t many concurrency related specification in Ethereum. For example, with respect to the transaction nonce value, the yellow paper specify that the transac-
tion nonce should equal to the sender account’s current nonce. Prior to the execution, the transaction nonce should be incremented by one. This specification works in face of serialized transactions. However, when the user submits two concurrent transactions as in #2793, both of them will exploit this specification and increment its own nonce value. As a result, these two transactions have the same nonce and one of them will be missed in the blockchain. For this transaction loss case, blockchain’s current specifications still cannot cover concurrency specifications. As a result, we cannot detect the concurrency bug with given specifications.

In face of above challenges, we focus on studying the blockchain’s distributed concurrency bugs and corresponding specifications. As far as we know, this is the first paper that uncovers the concurrency specification in blockchain systems. We figure out the concurrency in blockchain systems come from protocol concurrency and implementation concurrency. The former mainly contains some data race like concurrent transactions and blocks, which are carefully specified but lack correct concurrency specifications. Meanwhile, protocol concurrency involves some parallel operations, e.g., synchronization protocol, which don’t have any specification until now. The latter mainly comes from the locking problems. Current blockchain specifications are hard to detect the. Therefore, we made two major contributions:

- **A first quantitative study on concurrency specifications in Blockchain.** This paper systematically checks the concurrency vulnerabilities of Blockchain systems and categorize the concurrency specifications, which acts (1) as a reference for developers to know and avoid pitfalls during implementing blockchain protocol and (2) as a guide for researchers to foster the developement of analysis and verification for blockchain systems.

- **A new concurrency detection approach** It exploits a model checker to explore the local and distributed concurrency bugs. Until now, it has hit 6 distributed concurrency bugs by our model checker by symmetry and parallel state space reduction polices. In future, we will extend the model checker for local concurrency bugs.
CHAPTER 2

BLOCKCHAIN ARCHITECTURE

As we mentioned above, blockchain technology supports a lot of decentralized applications. One of the examples is Bitcoin, which is a centralized application for payment. Here, we will introduce another example, Ethereum, which adds another layer by allowing users to put code on its blockchain that executes automatically. We call the code as smart contract, which allows users to agree some basic code, a contract, to alert the system to execute transactions. For example, if the temperature goes higher than 70 degrees, the code pays Alice, otherwise, it pays Bob. All participants hold a copy of this agreement, which supports users to build small decentralized applications like the payment example. When we want to build larger and more complex decentralized applications, smart contracts can use more complex code to support them. Therefore, Ethereum is a decentralized platform leveraging smart contracts and underlying blockchain technology. Figure 2.1 shows the architecture of the Ethereum system, where the smart contracts run on top of Ethereum Virtual machine (EVM) and RPC calls to communicate with the underlying blockchain protocol. This paper will focus on studying the blockchain and smart contracts are out of our scope.

Here we will figure out how blockchain provides a better way from four aspects: (1) blockchain’s unique data structure is tamper-evident. Once transactions are recorded, they cannot be altered; (2) blockchain’s consensus protocol outperforms previous p2p decentralized system which failed in face of byzantine generals problems. (3) blockchain supports mining mechanism, which revolutionarily achieve consensus through computations. (4) blockchain shares a public ledger to every participant in the network. Throughout the paper, we will use Ethereum terminology to describe blockchain protocol.

2.1 Blockchain

Blockchain is secure, digital ledger of data, organized in blocks that contains transactions inside and are linked sequentially. It’s impossible to tamper any transaction or block without tampering
the subsequent linked blocks as well. This unique data structure makes blockchain powerful in face of attackers. In the section, we will separately introduce blockchain inner data structures: transactions, blocks, and state and how they comprise of the whole blockchain.

2.1.1 Transaction

A transaction refers to sending a given amount of digital money from one account address to another. It contains several fields:

- **from**: the sender of the transaction.
- **to**: the recipient of the transaction.
- **value**: the amount of money of the transaction.
- **nonce**: a scalar value equal to the number of transactions sent by the sender.
- **gas**: a unit of measuring the computational work of running transactions. The creation of gas is to separate the computation work from the market price, which fluctuates rapidly but computations don’t change in the same way.

- **gasLimit**: the maximum amount of gas used for executing the transaction.

- **gasPrice**: a delicate property

In blockchain, cryptography is utilized to secure the identities of users. Each transaction is encrypted on the sender side and decrypted on the receiver side. Here, we illustrate an Ethereum transaction example in Figure 2.2, where Alice wants to send Bob some money, she creates a transaction and signs it using her private key. Meanwhile, Bob will get a copy of Alice’s public key and verify the transaction with the public key. After the transaction is authenticated by Bob, it will be included into a block and later imported to the blockchain.

**Figure 2.2: Transaction Authentication.**

x
Besides the cryptograph verification about identify, each transaction should be validated its correctness before including into a block. Firstly, the transaction should be well-formed RLP; Secondly, the transaction’s nonce is valid and the gas is no larger than the gasLimit; Thirdly, the transaction value is at most equal to the account balance. After the initial verification phase, the transaction is in a pending status, waiting for a new mined block to have it included. Then the execution of a valid transaction contains some calculations like the account nonce is incremented by one, the account balance is reduced by the transaction value, the sum of the transaction’s gas limit and the gas utilised in the block prior must be no greater than the current block’s gasLimit and the computation results are in an eventual state.

2.1.2 Block

A block is a container data structure, which is used to hold submitted transactions. It consists of two parts: block header and block body. The transactions are stored in the block body and the header contains the metadata, which is used to represent the block identity in the blockchain and manage the transactions in its body. To be specific, the block header contains several pieces of information:

- **Number**: the number of ancestor blocks.

- **ParentHash**: the hash value of the previous block.

- **Nonce**: a value combined with mix-hash to prove a sufficient amount of computation has been carried out on the block.

- **Timestamp**: the unix time when the block is created.

- **ommerHash**: the hash of sibling of the block’s parent.

- **Difficulty**: a value corresponding to the difficulty level of the block. It’s decided by the previous block’s difficulty and timestamp.
• **ExtraData**: an arbitrary byte array containing data relevant to this block.

• **GasLimit**: the limit of gas expenditure per block.

• **GasUsed**: the total gas used in transactions in the block.

• **MixHash**: a value combined with nonce to prove that a sufficient amount of computation has been carried out on this block.

• **stateRoot**: the root node of the state trie.

• **transactionRoot**: the root node of the transaction trie, which is a trie structure populated with each transaction in the block’s included transactions.

• **receiptRoot**: the root node of the transaction receipt trie. The transaction receipt is the result of executing a transaction.
As we see in Figure 2.3, if any transaction in Block $N$ is tampered, then transaction root value will be changed, which leads Block $N$’s hash changed. Therefore, the Block $N$ hash value doesn’t match with the previous hash value in Block $N + 1$. All in all, the linked mechanism between blocks guarantees the history transaction data are hard for a hacker to corrupt.

2.1.3 State

Bitcoin is an address-based blockchain system, which needs to recalculate all the transactions when inquiring about some address’s balance. Ethereum improves this feature by incorporating account management. The account state essentially consists of a key-value map, where the keys are addresses and the values are account declarations, listing the balance, nonce, code and storage for each account.

- **nonce**: the number of transaction sent from this address.
- **balance**: the money owned by this address.
- **storageRoot**: the hash of root node of the state trie.
- **codeHash**: the hash of the EVM code of the account.

Figure 2.4: **State Trie.**
Technically, the states are assembled into a state tree (patricia tree) linked to accounts as in Figure 2.4. We call the root node of the trie as state root, which is placed in the block header. Therefore, we can consider the current state as the state when current block is added to the blockchain. Then the next state is next block’s state. The intermediate transactions between current block and next block are the state transition function. Once those transactions are executed (included in next block and added to the blockchain), the blockchain will update the account states correspondingly. With introducing the state concept, users can inquiry the state of the blockchain at any time in the past without having to recalculate everything from the beginning. To be simple, each block keeps the current state when it’s created.

2.1.4 Database

![Blockchain Data Structure](image)

Figure 2.5: Blockchain Data Structure.

With above three types of data, the blockchain are constructed as in Figure 2.5. The genesis block is the first block in the blockchain. Differing with other blocks, the genesis block doesn’t reference a previous block and it’s hardcoded into the blockchain as the start point. When the fist
block is created, it will be appended after the genesis block, which stores the genesis block hash inside the first block header. Similarly, the second block will be linked after first block, and the third block will be linked after the second block and so on. Through this way, all blocks are linked together and comprise of the blockchain, which is the so-called chain data.

The bitcoin system’s database just contains the chain data, while Ethereum’s database contains chain data and state data due to the account incorporation. The state data are not stored in the blockchain, and it maintains the mapping between addresses and accounts. Ethereum uses an underlying state database to store state data for different block timings.

2.2 Consensus

We are going to go through a list of consensus mechanisms which are used in the blockchain systems.

2.2.1 Proof of Work

The Proof of Work (PoW) [7] protocol is used in decentralized ledger to deter denial of service attacks and other service abuses on a network by demanding a considerable amount of work from the service requester. A key feature of the proof of work protocol is the asymmetry property: the work is hard (but feasible) on the requesters side while easy to check on the providers side. In bitcoin, the proof of work is implemented by asking the requester (miner) to produce the hash that satisfy certain conditions, such as the number of leading zeros. The hash function takes the block header and the nonce as the inputs and outputs the corresponding hash value. It is easy to evaluate the hash value from the inputs but it is very hard to guess the inputs from the hash value (non-invertible). So, in order to add a block into the blockchain, the miner must keep feeding the hash function (SHA-256) with the block header and different nonces until it produces the desired hash value. The average amount of work required to produce the desired hash value is exponential in the number of leading zeros. The attackers cannot change a single record in the bitcoin blockchain...
without changing all the subsequent records and redoing the associated proof of work. The proof of work is literally per CPU per vote. The most popular decision is represented by the longest chain, which has the largest amount of work invested in. If the major CPU power is at the hand of honest nodes, the honest chain will grow the fastest and outpace any cheating chains. To modify a past block, the attackers have to modify all the subsequent blocks and redo the proof of work associated, which makes it very difficult for the attackers to catch up with the work of the honest nodes. The probability of the attackers to catch up with the honest nodes diminishes exponentially as the number of subsequent blocks increases.

### 2.2.2 Proof of State

Proof of stake (PoS)\[9\] is designed to overcome the high electricity consumption of the mining process of the proof of work algorithm. In proof of stake, the generator of the next block is chosen by various combinations of random selection and an accounts stake (i.e., the account balance or age). Proof of stake needs a way to determine the next valid block. Just selecting by account balance would lead to undesirable centralization since the most wealthiest account would have permanent advantage over other users. Methods to avoid such situation includes:

- Randomized block selection. For example, in BlackCoin the next block is decided by a formula that looks for the lowest hash value in combination with the size of the stake. With the stake information being public, which account is the next to forge a block can be predicted.

- Coin age-based selection. For example, in Peercoin the stake of coins which have the largest product of the number of coins and the number of days the coins have been held has the greatest probability of signing the next block. In such a way larger and older sets of coins have larger probabilities of being assigned to the next block.
2.2.3 Practical Byzantine Fault Tolerance

Byzantine fault is derived from the Byzantine's general problem, where actors have to reach a consensus on a concerted strategy to avoid catastrophic failure, however, some of the actors are unreliable. Byzantine fault tolerance decides the reliability of a fault-tolerant system, especially of distributed computing systems, where components may fail and it is hard to know which specific ones are failed. In Byzantine failure, a component (e.g. a server) can inconsistently appear failed and functioning to failure detectors, exhibiting different symptoms to different detectors. In a network, it is hard for the other components to declare one component failed and shut that component down since they have to firstly reach a consensus on which component is failed. Practical Byzantine fault tolerance [5] is defined to achieve consensus in the cases of Byzantine failures. It applies the concept of replicated state machine and voting by replicas for state changes. The algorithm requires 3f+1 replicas to vote to tolerate f failing nodes.

2.2.4 Proof of Elapsed Time

Proof of elapsed time (PoET) is a blockchain consensus algorithm that avoids high resource utilization and high electricity consumption of proof of work algorithm. PoET improves the efficiency of the process by using a random leader election model in a Trusted Execution Environment (Intels software guard extensions (SGX)). For the consensus to work correctly, it randomly distributes the leader elections among all participating nodes in the blockchain network. Each participating node is asked to generate a random wait time and goes to sleep in that period. The first node to wake up (with the shortest wait time) will have to right to forge a new block into the blockchain and broadcast the information to the whole blockchain network. The same process is conducted repeatedly in forging new blocks into the blockchain. The randomly generated waiting times ensure that the leader role is randomly distributed among all participating nodes in the network. The drawback of PoET is the dependency on specialized hardware (Intel SGX).
2.2.5 Proof of Authority

Proof of authority (PoA) algorithm is used to deliver comparatively fast transactions in blockchain (compared with proof of work) through a consensus mechanism based on identity as a stake. PoA does not depend on nodes to solve computationally intensive mathematical problems, but instead assigns the authority nodes that are explicitly allowed to put transactions into new blocks and forge the blockchain. The newly forged blockchain has to be signed off by the majority of authorities and becomes a permanent record after that. Such mechanisms of PoA have the block issuers traceable and make it easier to maintain a private chain. Each PoA individual has the right to become the authority and there is an incentive for the individuals to retain the position that have been gained. Reputations are attached to every identity and the authorities are incentivized to uphold the transaction process as no one wishes to have his/her identity attached with negative reputation.

2.2.6 Ripple

Ripple consensus algorithm demands each node to define a Unique Node List (UNL), which comprises of other Ripple nodes that are trusted not to collude against the given node. Each UNL needs to have at least 40% overlap with other nodes UNLs in the network. Each node collect transactions in the network and put the transactions in its candidate set. To reach a consensus on which transactions will forge the block, each node in the network will consult other nodes in its UNL, get the votes and broadcast the votes. Based on the accumulated votes, each node refines its candidate set and transactions receiving the largest number of votes are passed to the next round. When a candidate set receives more than 80% of votes from all nodes in the UNL, the candidate set becomes a valid block or a ledger in Ripple. Such consensus is usually reached in just a few rounds. The finalized ledger is considered the Last Closest Ledger (LCL) and is added to the blockchain by each node. Next round of consensus will begin with new transactions and old transactions that are not added to the LCL. Consensus in the entire blockchain network will be reached when the individual sub-networks have their consensuses.
2.2.7 Stellar

Stellar consensus algorithm applies the concept of quorums and quorum slices. Quorum is a collection of nodes that are sufficient enough to reach agreements. A quorum slice is a subset of a quorum that is sufficient to convince one particular node of an agreement. An individual node can appear on multiple quorum slices. Open participation is introduced by Stellar by allowing each individual node to choose subsets of nodes from its slices. The quorum slices and quorums could be selected according to real life business relationships between various parties thereby leveraging trusts that already exist in business relationships. Quorums have to intersect each other so as to reach global consensus in the entire blockchain network. Global consensus is reached from decisions made in individual nodes.

<table>
<thead>
<tr>
<th>Consensus Protocols</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proof-of-Work (POW)</td>
<td>Bitcoin</td>
</tr>
<tr>
<td>Proof-of-State (POS)</td>
<td>Ethereum 2.0</td>
</tr>
<tr>
<td>Proof-of-Elapsed-Time (PoET)</td>
<td>Hyperledger Sawtooth</td>
</tr>
<tr>
<td>Proof-of-Authority (PoA)</td>
<td>Parity</td>
</tr>
<tr>
<td>Ripple</td>
<td>Ripple</td>
</tr>
<tr>
<td>Stellar</td>
<td>Stellar</td>
</tr>
<tr>
<td>PBFT</td>
<td>Hyperledger Fabric</td>
</tr>
</tbody>
</table>

Table 2.1: Consensus Protocol.

2.3 Miner

In blockchain p2p network, everyone can act on the transactions and access to the blockchain, how can they agree on a universal transactions order without a central authority? There are some special nodes, called miners, which validate new transactions and record them on the blockchain. In the Bitcoin system, miners provide computation power to solve a difficult mathematical problem based on cryptographic hash (SHA256) algorithm, which always produce the same arbitrary length data given the same inputs. It is impossible to compute the same hash with two different inputs (collision). It is also impossible to predict the output of any given data in advance. Due to
this characteristic, mining is a process of consuming computational resource to prove enough work is carried out to confirm transactions. In the case of Bitcoin, it takes approximately 10 minutes to mine a new block to confirm the transactions. Although mining is slow in finding a nonce value to solve the Proof of Work, it’s easy to prove the nonce is valid once anyone solved it. Everyone validates independently new blocks following the same hash function, it assures that miners cannot cheat, which is the key component of the blockchain decentralized consensus.

As we see, miners provide a mechanism to allow the blockchain to be a decentralized security and ensure fairness while keeping blockchain network stable and safe. There are two types of rewards: new bitcoins or transaction fees to incentive miners to join solving the complex mathematical problem. As for the detailed hash algorithm, we will keep the Ethereum blockchain as an example through this paper.

2.3.1 Ethash

Ethereum uses Ethhash[1] as its core mining algorithm, the general steps are as follows:

- There exists a seed which can be computed for each block by scanning through the block headers up until that point.

- From the seed, one can compute a 16 MB pseudorandom cache. Light clients store the cache.

- From the cache, we can generate a 1 GB dataset, with the property that each item in the dataset depends on only a small number of items from the cache. Full clients and miners store the dataset. The dataset grows linearly with time.

- Mining involves grabbing random slices of the dataset and hashing them together. Verification can be done with low memory by using the cache to regenerate the specific pieces of the dataset that you need, so you only need to store the cache.
2.3.2 CPU mining

The CPU of a computer performs the will of the software loaded on the computer and execute computing for the machine. In the early days, Bitcoin solved the mathematical problems with CPU processor. As the popularity of Bitcoin increase, more miners join the network, making it more difficult for individuals to solve the math problems.

2.3.3 GPU mining

The GPU, a more powerful processor, is a part of the video rendering system of a computer. GPU Mining is drastically faster and more efficient than CPU mining, therefore, more and more people tend to exploit GPU mining instead of CPU mining.

2.4 Network

A network of computing "nodes" make up the blockchain, where nodes are connected, they can be used to validate and relay transactions. Each node gets a copy of the blockchain, which is updated among network nodeis by broadcast and synchronisation mechanisms. Every node in the network can be considered as an administrator of the blockchain. Once a node join the network, it has an incentive of winning digital assets for participating and maintaining the network in a right way. Therefore, every node is responsible for broadcasting latest transactions and blocks to network and eager to synchronize the latest blockchain.

2.4.1 Broadcast

There are mainly two types of data needed to be broadcasted to the network. The first one is transaction, which is submitted by the users. The second one is block, which is generated by the miners. Whenever there are a new transaction or block, it will be broadcasted and known to the network. Here, we illustrate an example that nodes should undergo in face of submitting a transaction to the network.
• The new transaction is broadcasted to nodes in the network.

• When a node receives the new transaction, it will check whether the transaction is correctly formed and the account balance is enough.

• After initially verifying the transaction, it will be stored in the local transaction pool.

• Miners in the network work on constructing new blocks. If they find a nonce that solves the PoW, they will include the pool’s transactions in the new created block.

• The new block will be broadcasted to the network.

• When a node receives a new block, it will verify the block is valid and transactions within the block are valid.

• After the successfully verification, the node will append the block to previous accepted blocks, thus growing the blockchain.

As we see, broadcasting in Blockchain can guarantee all the latest transaction and block notified to the network. Even without a centralized node, nodes in the network can communicate and verify transactions and blocks for each other. Through this way, every node is able to have the same copy of the blockchain.

2.4.2 Synchronization

Instead of notifying a block or a transaction like above, synchronization provides a way to download multiple blocks at a time. This mechanism is important in blockchain due to two reasons: (1) the public blockchain network spreads over the world, which lead to some nodes not received broadcasting message promptly. This case is very common in face of the poor network, therefore, the nodes that lag behind need a mechanism to catch up with the network. (2) when a new node joins the network, its blockchain is empty, which requires the node to download the latest blockchain from the network. In sum, synchronization guarantees the network nodes on the
same page in a quick and efficient way. There are two main synchronization mechanisms: full synchronization and fast synchronization.

**Full Synchronization** means download the whole blockchain data including block headers and bodies. Meanwhile, it needs to process and verify all transactions included in blocks, which is a big bottleneck in full synchronization as it requires a large amount of time to complete it. As pointed in [], it takes a user 3 days to download all of the Bitcoin blocks from year 2009 to 2017. Currently there are three ways to do the full synchronization:

- Depending on the internal sync protocol, it will sync with the network periodically.
- Depending on the external user operation, it will import the remote blockchain to local disk.
- Users can choose to copy the remote node’s database to local machine, which completely replace the local blockchain.

**Fast Synchronization** downloads the block headers and bodies like the full synchronization to make up the blockchain. Meanwhile, fast sync will download the transaction receipts and pull an entire recent stat database. This allows a node to retain the status of the latest node without replay all transactions that ever happened in history. Through fast synchronization, it greatly improved the sync performance.

### 2.5 Blockchain Platforms

In this section, we will list several blockchain platforms.
<table>
<thead>
<tr>
<th>Implementation Language</th>
<th>Consensus</th>
<th>Data Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go-Ethereum</td>
<td>PoW</td>
<td>Account-based</td>
</tr>
<tr>
<td>Parity</td>
<td>PoA</td>
<td>Account-based</td>
</tr>
<tr>
<td>Cpp-Ethereum</td>
<td>PoW</td>
<td>Account-based</td>
</tr>
<tr>
<td>Pyethapp</td>
<td>PoW</td>
<td>Account-based</td>
</tr>
<tr>
<td>EthereumJ</td>
<td>PoW</td>
<td>Account-based</td>
</tr>
<tr>
<td>Hyperledger Fabric</td>
<td>PBFT</td>
<td>Key-value</td>
</tr>
<tr>
<td>Hyperledger Sawtooth</td>
<td>Proof of elapsed time</td>
<td>Key-value</td>
</tr>
<tr>
<td>Bitcoin</td>
<td>C++</td>
<td>PoW</td>
</tr>
<tr>
<td>Corda</td>
<td>Raft</td>
<td>Transaction-based</td>
</tr>
<tr>
<td>Ripple</td>
<td>Ripple consensus</td>
<td>Account-based</td>
</tr>
<tr>
<td>Stellar</td>
<td>Stellar consensus</td>
<td>Account-based</td>
</tr>
<tr>
<td>BigchainDB</td>
<td>Python</td>
<td>Trusted validators</td>
</tr>
<tr>
<td>LiteCoin</td>
<td>PoW (memory)</td>
<td>Transaction-based</td>
</tr>
</tbody>
</table>

Table 2.2: Blockchain Platforms.
CHAPTER 3

CONCURRENCY IN BLOCKCHAIN

Blockchain technology uses a digital ledger recording transactions, which is distributed across the network of private computers (nodes). Each node is able to start mining to create blocks for holding transactions and keep a copy of the blockchain ledger. Figure 3.1 depicts how a transaction is confirmed in the blockchain and how a new peer joins the blockchain network.

When a user submits a new transaction, it will be added to the local transaction pool. This pool will go through Tx verify phase and broadcast phase. Once a transaction is verified, it will be considered as an executable transaction, waiting for a new block to have it included. Meanwhile, the transaction will be broadcasted to the network, which guarantees other nodes to be notified. During the mining phase, there are a lot of miners competing to solve a difficult mathematical problem based on a cryptographic hash algorithm. We call it the Proof of Work (PoW), which is used to prove a miner spends enough computation resource to solve the problem. Among all miners, the first one solving PoW can create a valid block and broadcast it to the network. The block will store valid transactions in its body, then the blockchain will validate this block and write it to the blockchain. To be specific, the block will be Tx verification will validate a transaction and insert it into the non-executable queue for later pending promotion and execution. To be specific, it will drop transactions under the minimal accepted gas price, discard transactions that are already known based on the hash value, make sure the accounts exist, check nonce, check transaction doesn’t exceed the current block’s gas limit, ensure transactor has enough funds to cover the cost. Then it is waiting for moving transactions that have become processable from the future queue to the set of pending (processable) transactions. To be specific, it will drop any transactions that are deemed too old (low nonce) and too costly (low balance). If the pending queued more transactions than the hard limit, it will drop oldest ones.

Additionally, when a new node joins the blockchain network, it needs to download the latest blockchain in the network. We call it sync phase, which will download the blockchain from the best peer. More specifically, blockchain downloading contains block hashes, headers, bodies, transac-
tion receipts and state downloading. Once fetching all of them, it will construct the blockchain by ordering and combining them on its local machine. Through the synchronization, the new node can join the network with latest blockchain data.

3.1 Protocol Overview

In this section, we will introduce all the subprotocols inside blockchain with respect to transaction, block, chain, peer, broadcast, mining and sync.

3.1.1 Transaction

For transaction related protocol, it contains tx submission, tx verification and tx confirmation, which is shown in Figure 3.2.
**tx submission**  Tx submission will construct the transaction data, which contains account nonce, receiveipient, transaction amount, gas limit, gas price, and V.R.S. To be specific, When a user sends a transaction, it will sign it with the key associated with provided arguments and extract the nonce from the txpool’s pending state. For each node, it can receive a transaction from the network or submits it locally, all of them will be added to TxPool, which is memory pool to store all currently known transactions, if they are valid.

**tx verification**  The TxPool separates transactions into processable ones and future ones. The former is transactions currently pending for inclusion in the next block. The latter is transactions that are being scheduled for future execution only. Therefore, tx submission will add a single transaction to the future queue if it is valid according to the consensus rule.

Tx verification will validate a transaction and insert it into the non-executable queue for later pending promotion and execution. To be specific, it will drop transactions under the minimal accepted gas price, discard transactions that are already known based on the hash value, make sure the accounts exist, check nonce, check transaction doesn’t exceed the current block’s gas limit,
ensure transactor has enough funds to cover the cost. Then it is waiting for moving transactions that have become processable from the future queue to the set of pending (processable) transactions. To be specific, it will drop any transactions that are deemed too old (low nonce) and too costly (low balance). If the pending queued more transactions than the hard limit, it will drop oldest ones.

**tx confirmation**  
Tx confirmation means the associated block is added to the blockchain. Then the txpool will remove any transactions that have been included in the block or have been invalidated because of another transaction (e.g., higher gas price). The txpool removes invalid and processed transactions, where executable/pending queue and any subsequent transactions that become unexecutable are moved back into the future queue. To be specific, it will drop all transactions that are too costly (low balance) and too old (low nonce) and queue invalid transactions back to the future queue.

When a new block is added to the blockchain, the txpool will remove any transactions that have been included in the block or have been invalidated because of another transaction (e.g., higher gas price). The txpool removes invalid and processed transactions, where executable/pending queue and any subsequent transactions that become unexecutable are moved back into the future queue. To be specific, it will drop all transactions that are too costly (low balance) and too old (low nonce) and queue invalid transactions back to the future queue.

### 3.1.2 Block

For block related protocol, it contains block creation, block validation and block insertion.

**block creation**  
Geth prepares a fresh new block, which is built on the blockchain’s current block. It retrieves all currently processable transactions from the txpool, then commit transactions, which is to be included in the new block. Meanwhile, it commit uncle blocks to be included in the new block. The preparing phase is to commit new work on block, then multiple agents start the mining computation on this new block. Until now, the fresh new block is already created and waiting for
the nonce to be mined by PoW algorithm, which mainly does some mathematical computation to find a nonce that matches a target value.

The miner is responsible for creating a new block. The worker is mainly used to take care of applying messages to the system new state. It will register some CPU/GPU agent to do the PoW, whose result is a nonce value, which will mix with other variables to construct a full block. It will be returned back to the waiting worker. Creating a new block whose nonce will be mined.

**block validation** Once creating a new block, it will commit the the state changes to the database. The commit operation is used to validate state changes against the root hash stored in a block. Then it will validate the block header and try to write it to the chain, where if the block’s parent hash matches with the chain, it will insert the block as the new head of the chain, otherwise it’s in side status without inserting the block. For the canonical status, it will write transactions, receipts, and map bloom filters.

- **WriteTransactions**, store the transactions associated with a specific block into the given database. Besides writing the transactions, it also stores a metadata entry along with the transaction, detailing the position of this within the blockchain.

- **WriteReceipts**, stores a batch of transaction receipts into the database.

- **WriteMipmapBloom**, writes each address included in the receipts’ logs to the MIP bloom bin.

Then it will post NewMinedBlockEvent, ChainEvent, and ChainHeadEvent, storing all the transaction receipts belonging to a block as a single receipt slice. This is used during chain reorganisations for rescheduling dropped transactions. Finally, it will extract the canonical block based on the block number, if the hashes are matched, the local mined block will be stored temporarily and wait for 5 blocks for confirmation.

**block insertion** Writing a block to the chain, it will calculate the total difficulty of the block, comparing it with local chain’s total difficulty. Irrelevant of the canonical status, write the block
itself to the database, where serializes a block into the database. To be specific, it writes the block body first to retain database consistency and then store the block header as well to signal full block ownership. After writing the block into the database, it will check the block total difficulty and parent block hash. It will reorganize the blockchain or insert the block as the new head of the chain.

3.1.3 Chain

Blockchain represents the canonical chain given a database with a genesis block. The Blockchain manages chain imports, reverts, chain reorganisations. Importing blocks into the blockchain happens according to the set of rules defined by the two state validator. Processing of blocks is done using the Processor which processes the included transactions. Validation of the state is done in the second part of the Validator. Failing results in aborting of the import. The blockchain also helps in returning blocks from any chain included in the database as well as blocks that represents the canonical chain.

chain import Chain importing will attempt to insert the given chain into the canonical chain or otherwise create a fork. If an error is returned, it will return the index number of the failing block as well as an error describing what went wrong. To be specific, iterating over the block of the given chain (1) set the block total difficulty, which add the block difficulty with parent block’s total difficulty; (2) call into the block processor, which will attempt to process the given block’s transactions and applies them on top of the block’s parent state; (3) write the given block to chain database; (4) compare the td of the last known block in the canonical chain to make sure it’s greater, at this point, it’s possible that a different chain (fork), which will trigger to merge two different chains into the new canonical chain. (5) set the total difficulty of the blockchain and insert the block into the blockchain; (6) if the block’s total difficulty is less than the canonical chain’s td, it will create a chain side event.
chain export

chain reorganization The blockchain.WriteBlock is a method used by the miner whenever it mines a new block and have it imported into the local chain. If the block’s external total difficulty is higher than current block’s local total difficulty, it will add the new block to the canonical chain. However, if the block’s parent hash is not equal to current block’s hash, then it will reorganize the current block and the new block. Chain reorganization takes two blocks, an old chain and new chain. It will reconstruct the blocks and insert them to be part of the new canonical chain and accumulates potential missing transactions and post an event about them. To be specific, it looks for the common block and create the new chain and old chain based on the new block and current block. For the new chain, it will insert the block in the canonical way, write canonical receipts and transactions, and map bloom filter into the chain database. Then calculate the difference between old chain and new chain’s transactions (deleted and added transactions), delete the transaction difference. When transactions get deleted from the database, it means the receipts were created in the fork must also be deleted. Meanwhile, there are three different goroutines to post Removed-TransactionEvent, RemovedLogsEvent, ChainSideEvent. As for RemoveTransactionEvent, it will notify the txPool to delete the transaction difference. As for the ChainSideEvent, it will notify the worker to add all old chain’s blocks into the possible uncles.

3.1.4 Peer

When the node synchronizes with the peer, which starts a block synchronization based on the hash chain from the specific peer and head hash, it will fetch hashes and blocks specifically. Fetching blocks iteratively downloads the entire schedules blockchain, taking any available peers, reserving a chunk of blocks will fetch hashes and blocks specifically. When the node synchronizes with the peer, which starts a block synchronization based on the hash chain from the specific peer and head hash, it will
fetch hashes and blocks specifically. Fetching blocks iteratively downloads the entire schedules blockchain, taking any available peers, reserving a chunk of blocks for each, wait for delivery and periodically checking for timeout. If no blocks were delivered, demote the peer and set it idle. If all were successful, promote the peer and set it idle. If peer probably timed out with its delivery but came through in the end, demote the peer and set it idle. If peer did something semi-useful, demote the peer and set it idle. Here, SetIdle sets the peer to idle, allowing it to execute new retrieval requests. Its block retrieval allowance will also be updated either upwards or downwards, depending on whether the previous fetch completed in time or not.

### 3.1.5 Mining

![Mining Protocol](image)

Figure 3.3: **Mining Protocol.**

#1988 when the transactions are processed in the miner for pending block, the block hash isn’t yet known. As a result the generated receipts and logs don’t yet contains the blockhash. The fix is to update the block hash when the block has been mined. Figure 3.3 is the main workflow for miners, which use some cpu or gpu agents to do the Proof of Work.
3.1.6 Broadcast

Broadcast protocol mainly sends out the new transaction and block message to the whole blockchain network.

3.1.7 Fast Sync

Instead of downloading the entire blockchain and process all transactions that ever happened in history, fast synchronization downloads the transactions receipts along the blocks, pulls an entire recent state database, and accepts block hashes and associated blocks, then it reconstructs the blockchain locally. At this point, fast sync involves obvious concurrency characteristics: parallel hashes, blocks, receipts, state downloading and separate block processing to reconstruct the blockchain. Figure 3.4 depicts the 5 parallel downloading and constructing in fast sync protocol.

Figure 3.4: Sync Protocol.
3.2 Protocol Concurrency

Although blockchain protocol revolutionizes the transactions, it suffers from some non-deterministic characteristics. In Tx verify phase, when a user submits two transactions simultaneously, how does the transaction pool order the transaction? In mining phase, when two miners create two valid blocks at the same time, which block will be added to the blockchain? In the block write phase, when it receives two blocks through the network, which one should be written to the blockchain? Based on above analysis, we found blockchain is mainly handling transaction and block data without a central scheduler, it exposes more potential data race issues.

3.2.1 Tx Submission Race

As pointed out in #2950, when a user submits 2 concurrent transactions, it might have the same nonce value for both. As a consequence, when the block includes transactions inside, it will order based on the nonce value, which will lead to one of the transaction missing in the block.

In the Tx verify phase, when a user submits two transactions simultaneously, in theory, a transaction list is never manipulated concurrently. However [8] points out the smart contracts lead to transaction execution overlapping. This concurrent perspective discloses the concurrent transaction dangers and warn us to pay attention to the gap between theory and implementation. As defined in the transaction pool, it must ensure serialized access. It’s still possible the real implementation violates this assumption. All in All, we cannot stop users submit concurrent transactions, does the blockchain handle them correctly?

3.2.2 Block Creation Race

A subset of nodes, called miners, organize valid transactions into blocks. In blockchain system like Bitcoin and Ethereum, miners race to create new blocks, a process that requires solving a computationally mathematical puzzle, called the Proof of Work. The first miner to solve the puzzle which involves randomly guessing at a number called a nonce, which is combined with other data
in the block to create an encrypted digital fingerprint, called a hash.

In the mining phase, the non-determinism is obvious. Miners compete to solve PoW and the first one who mined a block will broadcast the new block to network, notifying other nodes to verify and accept the new block to achieve consensus. Here, miners are possible to create two new blocks at the same time. Miners themselves have no mechanism to decide which block should be valid. On the contrary, miners notify blockchain to add two concurrent blocks, which will force the blockchain to split into two branches and form a blockchain fork like Figure 3.5. Although eventually there are some mechanisms to converge these two branches to a unique branch, we call it canonical blockchain, the mining protocol essentially brings the concurrency weakness.

![Blockchain Fork](image)

**Figure 3.5:** Blockchain Fork.

### 3.2.3 Block Insertion Race

In the block write phase, all incoming blocks will be written to a same local blockchain. As we mentioned in the broadcast phase, once there is a new block created, it will be broadcasted to the network. Therefore, it's highly likely to have concurrent blocks coming, then the blockchain should decide the block orders. Due to this, blockchain is exposing itself to concurrent ‘attacks’. Similar to the transaction pool, blockchain assumes itself to ensure serialized access.

### 3.2.4 Sync and other Race

In the sync phase, a new peer will download the whole blockchain from the best peer. As we see, the blockchain is a kind of append data structure. Downloading a complete blockchain means a large amount of data. As pointed in [], synchronization is still ongoing even three days. For this
purpose, parallel downloading is efficient and necessary. In Ethereum, the fast synchronization requires to download block hashes, headers, boides, transaction receipts, and state concurrently. Then the node waits for all the downloading results in a group to reconstruct the local blockchain. With the concurrency characteristics in the synchronization protocol, blockchain downloading may fail or stall no matter which part gets wrong. As reported in Github, there are about 1000 synchronization related reports. Even the developers don’t have a better solution to solve synchronization failure because any part of these concurrent downloadings could be the culprit. Normally, the developers recommend users to clean the blockchain database and resync from the beginning.
CHAPTER 4
CONCURREN CY BUGS IN BLOCKCHAIN

4.1 Methology

We categorized our 64 concurrency bugs into local concurrency (l), distributed concurrency (d) and combination of local and distributed concurrency (ld) as shown in Figure 4.1.a. Among all these concurrency bugs, there are 67% are protocol related bugs and 33% implementation related ones. The protocol related concurrency bugs are mainly caused by protocol design, but the implementation bugs are caused by the developers’ coding error. For the implementation bugs, they are mainly from three cases: deadlock (40%), double close/lock (30%), and access before creating (40%).

Figure 4.1: Statistical Overview.

As for the bug characteristics, we discuss their involved protocols, fix methods and bug symptoms in Figure 4.1.d, 4.1.b, and 4.1.e. Sync, peer and mining protocols are the top three protocols that cause the concurrency bugs, and most of the symptoms are hang, crash, performance degra-
dation, data loss, peer disconnect. Finally, 33% of bugs are fixed by adding locks and 23% bugs are used to check nil pointer or corresponding variables. The remaining bugs are fixed by change the local order process (18%) and the processing mechanism (26%).

In the end, there are 39% bugs are triggered by order violation, 30% by atomicity violation, 14% by crash/reboot events and 17% by group wait for multiple goroutines.

4.2 Statistics Overview

As we see, a lot of concurrency bugs involve 2 protocols, we list all the protocols race among all bugs in Figure 4.2

Figure 4.2: Blockchain bug distribution.

ProtocolManager manages the peers capable with the ethereum network, when a new peer is connected, ProtocolManager will handshake with the peer through StatusMsg, register the peer in the downloader, propagate existing txns to the peer, and wait for incoming messages from the peer to handle.
After creating the ProtocolManager, it will be started, which mainly do four things: subscribe TxPreEvent to broadcast transactions, subscribe NewMinedBlockEvent to broadcast mined blocks, periodically synchronise with the network, both downloading hashes and blocks as well as retrieved cached ones, and collecting hash notifications, periodically checking all unknown ones and individually fetching them.

When the protocol manager received the NewMinedBlockEvent, it will propagate the block to a subset of its connected peers, only notifying the rest of the block’s appearance. To be specific, if we have 5 peers, then it will send NewBlockHashesMsg to peer4 and peer5, but send NewBlockMsg to peer1, peer2 and peer3. When a node receives NewBlockHashesMsg from the peer, it will mark the hashes as present at the remote peer, set the peer’s recent hash, and then collect all the hash announcements into the newHashCh. When a node receives NewBlockMsg from the peer, it will validate block fields, and then import the new block into the blockchain. Here, importing block will mark the block as present at the remote peer and set the peer’s recent hash as above. Then if there exists a td, it will reset the peer’s td. If the block is already known or its difficulty is lower than our, just update the peer’s td and return. Otherwise, it will insert the newly received block and propagate to other peers.

Inside the fetcher goroutine, the protocol manager reads a batch of hashes that notified from the newHashCh, then the fetcher will every notifyCheckCycle=100ms to clean up any expired (5s) block fetches, check if any notified blocks failed (500ms) to arrive, and finally it sends out all block requests that failed hash announcement, which sends GetBlockMsg and waits for new blocks through the newBlockCh. It firstly filter explicitly requested blocks from the hash announcements, then start a new goroutine to import these explicit blocks into the chain manager.

#14379 fixes deadlock on Wait() Node restart operation terminates a running node and then boots up a new one in its place. If a wait call happens between the termination and booting, then wait call will consider the node nil, then just stop without unlocking. Finally the restart call will continue booting up the node, which requires acquiring the node’s lock, therefore, the node will
stall in starting running it. The fix is to unlock before returning.

**#15526 accounts: fix two races in the account manager**  There is an initialization and run data race in the account manager. The manager first subscribe to wallet events, then did an account listing, and only afterwards fired up a goroutine to consume the subscriptions. If the keystore contains more than 8 accounts, the wallet listing actually fired 8+ events, overflowing the manager’s subscription and blocking. Since the manager itself is firing the events internally from the listing, it cannot consume them, locking itself up. The fix is to do the initial scan of the wallet first and only afterwards subscribe to events.

(2) Two concurrent scan results in the same files being gathered as new, the first obtaining the lock reports them as new, the second reports them as modified. The result is that all accounts get added, deleted and readed to keystore account cache for every concurrent scan. The fix is to make the folder metadata scan lock the entire file cache. This lock is held for 50ms anyway, so it’s not a heavy congestion.

There are two types of bugs: protocol related and implemented related.

### 4.3 Protocol Bugs

#### 4.3.1 Transaction

**#1223 fixed race condition for subscriptions**  The transaction pool will subscribe ChainEvent inside the eventLoop goroutine, and unsubscribe it when stop the pool. Therefore, in face of transaction pool starting and stopping in a quick succession, the pool’s eventLoop goroutine hasn’t subscribe, then the stop operation will unsubscribe nil pointer. The fix is to move the subscribe into the transaction pool initialization instead of being inside the eventLoop’s goroutine.

- **Protocols:** tx
- **Symptom:** crash
• **Fix:** move the subscribe into the transaction pool initialization instead of being outside the eventloop’s goroutine

• **Spec:** unsubscribe should happen after subscribe.

### #1865 Geth Deadlock in chainmanager after posting RemovedTransactionEvent

Once a transaction is included in blockchain, it will be removed from the transaction pool. Specifically, the chainmanager posts a RemovedTransactionEvent, notifying the transaction pool to remove it. Here the chainmanager has previously been locked prior to posting RemovedTransactionEvent. Therefore, the transaction pool cannot get the chainmanager’s lock.

• **Protocols:** tx

• **Symptom:** deadlock

• **Fix:** post RemovedTxEvent in a separate goroutine, then the txpool will not indefinitely wait for the chainmanger’s lock.

• **Spec:** RemovedTxEvent must guarantee the txpool to remove the corresponding tx

### #2950 Geth Transactions being given the same nonce

When the user submits a new transaction, it will get latest nonce from the sender’s account address and increment by one as its own nonce. To be specific, the transaction reads the nonce from the stateDB, which just holds a read lock instead of rw lock, then in face of two concurrent transactions for the given account, they get the same nonce and both increment one. As a result, these two transactions have the same nonce value, which leads to one of them missing while being included into a block.

• **Protocols:** tx submit

• **Symptom:** tx miss

• **Fix:** use a rw lock for the stateDB, which prevents any concurrent txns to update their nonce values simultaneously.
- **Spec**: there are no two transactions with the same nonce per address.

**#3412 core: bugfix state change race condition in txpool**  
As pointed out in the Ethereum yellow paper, when finalizing one block, the protocol is required to finalize the transactions and state. Therefore when a new block comes, the txpool needs to update the state based on the executed and pending txns. For example, the current state nonce should be equal to the nonce of the last sent transactions and the pending state’s nonce should be the sum of current state’s nonce and the number of pending txns. When resetting these states, the txpool ignores the coming of the second block, which may include all the pending transactions into its body. Then the txpool will not consider them as pending, which directly affects the result of pending state nonce, as a result they are gone due to the next block’s coming.

- **Protocols**: tx confirm

- **Symptom**: pending state out of sync

- **Fix**: pass the current state to each reset operation, then event the second block comes, it will not affect the current state view of the first block

- **Spec**: pending state’s nonce can never be less than current state’s nonce

**#15085 core: make txpool operate on immutable state**  
Apparently the transaction pool (and handling in general) had 3 different update pathways: 1. On chain head events (and most notably reorgs), the pool’s internal state was reset to the current chain state (and transactions filtered accordingly). 2. On transaction removals events (caused by reorgs), the old transactions were rescheduled. 3. On miner invalidations, transactions we’re yet again concurrently dropped from the pool. During reorgs, the pool absolutely surely goes out of sync with itself, because its tracked nonces are updated based on head events, but its tracked transactions are updated based on removal events. These by their very nature happen concurrently, so any interaction with the transaction pool in between is undefined.
• **Protocols:** tx confirm

• **Symptom:** txpool out of sync

• **Fix:** reduce the tx pool’s API surface to one single entry point: chain head events.

When a chain head event is received, we reset the pool’s internal nonces to the current state as currently, but afterwards we gather the reorged transactions within the pool itself and apply any diffs before releasing the lock. This ensures that nonces, states and transactions are updated atomically and cannot go out of sync between each other.

### 4.3.2 Block

**#1765** when inserting blocks into database, we first mark the new block as a head and only then write the actual block itself. This seems a data race as an query in between the two operations will read invalid state (i.e., the head block will not exist)

• **Protocols:** block insert

• **Symptom:** crash

• **Fix:** change the order

• **Spec:** mark the block as head happen after inserting the block to database

**#1785 parity invalid state root while importing new block**

• **Protocols:** block insert

• **Symptom:** mismatch between blockchain and db transactions

• **Fix:** change the order

• **Spec:** blockchain should commit best block after the db transactions have been committed.
#1917 split out block validator and state processor

#3557 Cpp-Ethereum  BlockQueue exceed the limit, the sync paused forever.

#15204 Bad block Error: unknown ancestor

- **Protocols:** block insert
- **Symptom:** bad block, unknown ancestor
- **Fix:** avoid writing existing block again.
- **Spec:** each block (except genesis block) should have a parent block

### 4.3.3  Chain

#1220-1 fix a lock annoyance and potential deadlock  There are two places importing chain, then the td read and write are locked by the same lock. Then one goroutine is writing, then the other is not allowed to read td.

- **Protocols:** Chain Import
- **Symptom:** deadlock
- **Fix:** separate the write lock and read lock

#1746 eth.getTransactionReceipt returns unmined transactions  the transactions were indeed successfully mined at a certain point, but then a higher TD block (not containing the txn) appeared, the chain was reorganised and the transaction history overwritten. The issue was that all nodes marked the tx already done, so nonce bothered to resent them.

- **Protocols:** Chain Reorg
- **Symptom:** tx miss
• **Fix:** move missing txns back to the txpool

• **Gap:** design weakness.

**#1988-2** there may be a database race in this code. The core.InsertChain ensures that only a single thread is allowed to muck around in the database, but the miner essentially goes around this lock, which imho could cause weird reorg issues where a mined block gets organized out before all the individual writes (txns, receipts, mipmaps) get flushed into the database.

**#2298 block is lost and no one wrapped it as uncle**

• **Protocols:** Chain Reorg

• **Symptom:** block miss

• **Fix:** during reorganisation of chains, the old chain is regarded as uncle

• **Gap:** design weakness.

A block must specify a parent, 0 or more uncles. An uncle included in block B must have the following properties: It must be a direct child of the k-th generation ancestor of B, where \( c \leq k \leq 7 \) I cannot be an ancestor of B, An uncle must be a valid block header, but does not need to be a previously verified or even a valid block, An uncle must be different from all uncles included in previous blocks and all other uncles included in the same block; For every uncle U in block B, the miner of B gets an additional 3.12% to its coinbase reward and the miner of U gets 93% of a standard coinbase reward

**#2793 Geth Solve a remote import and local mine data race** When successfully mining a new block, it will be added into the local blockchain. If there is a remote blockchain imported before adding phase, which updated the local blockchain, then there already existed one block in the blockchain based on the new block’s number. At this point, although the mined block is valid, the blockchain will consider it stale and refuse to add to the blockchain. Even the miner continues...
mining more blocks, they will encounter the same problem. The fix for this bug is to lock the blockchain, which prevents the local adding from being interrupted by remote importing.

- **Protocols:** Chain Import
- **Symptom:** mined block failed writing
- **Fix:** add a lock
- **Gap:** after successfully mining 6 succession blocks, the blockchain should confirm the first block, otherwise it must have some issues during writing blocks instead of mining.

### 4.3.4 Peer

#### #593 p2p deadlock on disconnecting a peer twice in quick succession

When a server adds a peer, it will handshake with the peer first and start a new goroutine to run a peer, which waits until there are some protocol error or read error, thus disconnecting the peer. To be specific, there is a readLoop goroutine to read a communication message, when it returns a read error, peer.run will disconnect the peer. When peer.run receives some protocol error, it will disconnect the peer and waiting for readloop to exit. However, the read loop is waiting for the protocol to read a message, which will return some read error to disconnect the peer. The peer.run is still waiting for accept the disconnect, then the second disconnect will never be triggered because peer.run is past the point where it waits for read error to disconnect peer.

- **Protocols:** peer management
- **Symptom:** deadlock, node stalling
- **Fix:** Disconnect a peer by returning from the protocol loop instead of calling Disconnect.
- **Reason:** It’s not safe to wait for the protocol without a timeout.
- **Gap:** The code assumes that the protocol will always accept messages rather quickly.
Assume node1 has best peer node2, then when adding two new peers node3 and node4, it will compare their td with the best peer node2’s td. If node3 and node4 have larger td than node2’s td, then it will switch the best peer with node3 and node4. However, the code doesn’t lock the best peer. It’s possible that adding the new peers node3 and node4 both consider node2 as the current best peer, then close the corresponding channels. Finally, it will close the closed channels in face of the 2 concurrent new peers.

- **Protocols**: peer management
- **Symptom**: crash
- **Fix**: add a lock to serialize adding peers
- **Gap**: The code assumes to add peers serially

Remove peer can be called even after listen loop and dial loop have returned.

- **Protocols**: peer management
- **Symptom**: unknown
- **Fix**: add a lock to peer set
- **Gap**: Assume no new peers can be added at the point when dial and listen loop are down

When a node wants to add a new peer, it will create, initialize and start the new. Since the node should track all peers, which will add the new peer after finishing the handshaking and starting. However, if the nodes wants to remove the peer, it might have no peer to remove because adding peer operation happens in the end.

- **Protocols**: p2p
• **Symptom:** crash

• **Fix:** change the local ordering in the node

• **Spec:** add the new peer to the peer set before handshake and starting.

**#1027 Data race in the eth synchroniser** sync protocol calls `getBestPeer(pm.peers)` at multiple occasions, but access to this map is not protected and can be modified during peer additions and removals.

• **Protocols:** peer, broadcast

• **Symptom:** some peers with missing blocks/txns, best peer issue, unhealthy peer messages

• **Fix:** make the peer set thread safe

• **Gap:** assume no peers can be added at the point.

**#1216 fix close data race** In P2P, `doProtoHandShake` will send `handshakeMsg` to `werr channel` in a new goroutine, then it will read `Protocol handshake`, where it will stop if there is any error. If `readProtocolHandshake` error happens first before successfully sending the `handshakeMsg`, then it will leave it in the flight. We should make sure the sending finished before stop the whole process. The fix is to read the `werr channel` to make sure the write terminates too.

• **Protocols:** peer

• **Symptom:** unknown

• **Fix:** read the `werr channel` to make sure the write terminates

• **Gap:** assume sending will complete rather quickly.
#1261 track write errors and prevent writes during shutdown

- **Protocols:** p2p, crash

- **Symptom:** unknown

- **Fix:** prevent new writes from starting after Server.stop has been called.

- **Gap:** When a write fails, shutdown is initiated immediately and no new writes can start.

#2321 eth/downloader: fix a throughput estimation data race  When the downloader was estimating the throughput of a peer and updating its value, we accidentally only locked the peer’s mutex for reading, not writing. Interestingly this data race doesn’t get hit by the current code, only surfaced by the concurrent headers.

- **Protocols:** p2p

- **Symptom:** unknown

- **Fix:** use a write lock.

- **Gap:** concurrent headers.

#2523 improve shutdown synchronization  The stop method of eth.ProtocolManager, miner.Miner, and core.TxPool is asynchronous. Left over peer sessions generate events which are processed after Stop even through the database has already been closed. There is a small chance of deadlock when you drop all currently existing peers and then wait for shutdown. A concurrently running Run() function could have been already taken the path to handle the peer but not yet added it to the peer set at the moment when you drop everyone in the set. In this case, the peer still gets added and shut down wait will wait forever.

- **Protocols:** sync, stop, peer

- **Symptom:** deadlock, hang
• **Fix:** empty the peer set periodically until the wait returned, make stop synchronous using `sync.waitGroup`.

• **Spec:** peer

#3326 cannot connect to static/trusted peer with –maxpeer 0  when set maxpeer 0, event trusted/static peers can not connect. Trusted peers are supposed to be able to bypass the max-Peer limit. This logic was appropriate executed in server.go, but the eth handler and les handler also check maxPeers which are previously ignoring the trusted flag for peers.

• **Protocols:** peer

• **Symptom:** set –maxpeer 0, event trusted/static peers can not connect

• **Fix:** allow exceeding maxPeers for trusted peers

• **Spec:** no matter how to set maxpeer, the trusted/static peers can connect

#16146 fix race condition in whisper/peer.go  it’s caused by the indeterministic access to both peer.fullNode and peer.bloomFliter. The fix is to add required mutex to protect access to both fields.

4.3.5  **Sync**

#974 Geth Handle a potential unknown parent attack handle a potential unknown parent attack. Currently the downloader accepts a hash chain blindly, and then it starts pulling associated blocks blindly again. After reconstructing an order locally, it starts delivering it upstream, which will detect any potential forgeries or invalid data. TakeBlocks currently checks if the head block’s parent has been integrated, and if not yet, then it waits for a previous take to complete. However, if the head has a forged parent, then the waiting is actually stalling. Since the forged parent will never be integrated. This will essentially block the downloader indefinitely. This bug happens due to the implementation doesn’t handle the forged parent and assume the network is reliable.
• **Protocols:** sync

• **Symptom:** sync stall

• **Fix:** serialize the access to take blocks such that unknown parents can be avoided.

• **Spec:** Each block’s parent hash should be equal to the previous block’s hash, if they are not equal, we can detect the unknown parent error.

**#996 Geth circumvent download race between crosscheck and hashes**  circumvent download race between crosscheck and hashes. This is a download race that can rarely occur, but if you have a shitty network connection, then it becomes much more probable. Essentially if you try to cross check with the last hash from a retrieved chunk, then its parent is not yet available, and if you manage to get the cross checked block faster than its parent hash. Inside the code, downloader’s fetchHashes will try to fetch a random block to verify the hash batch.

• **Protocols:** sync

• **Symptom:** crash

• **Fix:** skip the last hash as the cross check races with the next hash fetch

• **Spec:** design weakness, each block’s parent hash should be equal to the previous block’s hash.

**#1206 rare downloader lockup during heavy churn**  During heavy peer churn, we try to send a request to a peer, it’s dropped before we get to push out the request, and a channel to send the message on become nil, blocking forever.

• **Protocols:** sync, peer crash

• **Symptom:** lockup

• **Fix:** add a timeout mechanism
**#1216 fix data race in the hash announcement processing**  The fetcher goroutine extracts the explicit fetches when blocks arrived. Another importer goroutine will import explicit blocks one by one into the chain manager. There is a data race when a new explicit block during importing previous explicit blocks. The fix is to create a closure with the retrieved blocks and origin peers inside the fetcher goroutine, then give the closure to importer goroutine.

**#1224 collect and report import progress**

- **Protocols**: sync

- **Symptom**: performance issue, query delayed

**#1287 fix processing interrupt caused by temp cancel**  when a processing is aborted due to a cancellation, and no new one is ever spawned, leading to a lockup: We start importing a batch of very heavy blocks, the first 256 out of an entire 1024 cache The downloader starts fetching to refill the cache, but fails and aborts the entire sync However, the processor is inserting very heavy blocks, so it doesn’t notice the cancel A new sync is started, and the cache is filled with fresh blocks Even though blocks arrive, no new processor is started since the previous is still running The previous finishes its heavy import, notices the cancelled op and terminates (no restart check) Since the cache is already full, no new processor is started in place of the old Deadlock

- **Protocols**: sync, cancel

- **Symptom**: deadlock

- **Fix**: add atomic boolean to signal termination

- **Spec**: assume the block inserting is fast.

**#1371 maybe fetcher deadlock knownBlocks/knownTxns**  The knownBlocks/knownTxns are locked, when comparing their sizes with maxiKnownBlocks/maxKnownTxns, it will remove the
redundant blocks or txns, which holds the lock inside remove operation. Therefore there is double
lock in this case.

**#1843 Geth Always send termination wakes, clean leftover**  In fast synchronization, downloaders start fetching hashes and fetching blocks concurrently. When the hash fetching terminates successfully, it will send a true signal through the wake channel to the blocks fetching. However, the block fetching fails before reading from the wake channel, which keeps the true signal stored inside. In the next sync cycle, the block fetcher will read the true signal from the wake channel in the beginning and terminate prematurely. It cause the hash fetcher to wait indefinitely because it will send the terminate signal to a non-exisiting block fetcher.

- **Protocols**: sync, goroutine failure
- **Symptom**: deadlock, hang
- **Fix**: empty the wake channel prior to sync
- **Spec**: fast sync requires downloading equal number of hashes and blocks

**#1939 eth: don’t block sync goroutines that short circuit**  The block insertion into the local chain always force-set the head header and head fast-block to the block being inserted. This is required so that a previous fast/light sync doesn’t leave head headers pointin to old/stale side-chains. However, we always reset the heads, even if they were on the correct chain. This opened up a race condition where the downloader pulled some headers, inserted them and then they went missing.

- **Protocols**: sync
- **Symptom**: block loss
- **Fix**: only reset head/fastblock if stale
- **Spec**: sync will download canonical chain’s blocks.
#2211 core: writeHeader very rare crash  When the insertion of a batch of headers partially succeeds, but fails before writing all, the rollback is called before pushing the succeeded headers into the rollback list. This causes them not to be rolled back. When doing fast sync and importing headers, if part of the batch import succeeded but part of it not, then the rollback didn’t remove the part that actually succeeded (it only appended them to the list after importing the entire batch successfully), so even though the receipts were rolled back, the headers not. This should not really cause any issues because when we sync the next time, we do the ancestor lookup from our head receipts and not head headers. Nonetheless it’s bad behavior so the PR fixes it by rolling back the successful part of a header insert too, even if some failed.

- **Protocols**: sync

- **Symptom**: 

- **Fix**: rolling back the successful part of a header insert too even if some failed.

- **Spec**: assume the rollback will not fail partially.

#2844 eth/downloader: fix #2841 spikes by block fetches  A data race between import and sync cycle. The issue was in the search for the latest common ancestor in a presumed split. The current geth relies on the peer to send a correct block height together with its total difficulty. The search for the latest common ancestor starts at the height that the peer reported as highest, looking backwards.

To that end, often 192 block headers are being requested from the peer backwards from its reported height. Presumably when the height and/or total difficulty were reported wrongly, the block at the reported height is then accepted by the local client as latest common ancestor with the peer. Because it’s the latest matching of the blocks that the peer was asked about, in the case that all blocks match the local chain, which should never occur, but frequently does.

Blocks after the reported height are not checked. This often goes wrong and results into waste of resources. If the real latest common ancestor came after the height reported by the peer, this is liv
not detected. Likewise if the chains were in fact identical, in both cases the block at the height originally reported by the peer as best remains the official latest common ancestor in the eyes of the local client.

The results in all headers and bodies of all blocks being fetched beginning from that reported height forward to the actual height the peer could deliver headers for. Whatever wrong height the peer would report, block headers and bodies would be refetched from that the height on, resulting in some cases into thousands of blocks being frequently refetched. Due to the internal architecture of geth, the blocks were immediately pow-checked, creating cpu load, and likely the I/O pressure before being found to already be in the local chain, and discarded.

- **Protocols:** sync

- **Symptom:** refetch block, may lead to memory leak, traffic and cpu spikes by useless block fetches, disk I/O hammering

- **Fix:** the block at the reported height is compared the local chain first. If they match, the conclusion about differing total difficulty must be assumed wrong and based on wrong reporting from the peer.

- **Spec:** internal architecture of geth

#2868 eth/downloader: abort sync if master drops (timeout prev) currently the downloader tracks peers drops for "data downloads", so that if a remote node is disconnected, all its assigned fetches are returned into the queue to be pulledd by someone else.

However fetching the remote height, fetching the common ancestor and fetching the skeleton chain are done outside the "fetch queue", so none of these code segment are aware that a peer was dropped. This results in them having to wait for a timeout to hit before aborting and starting over.

- **Protocols:** sync, crash

- **Symptom:** wait for a timeout to hit before aborting and starting over.
• **Fix:** track peer drops

• **Spec:**

**#15138 Geth Track peer drops and deassign state sync tasks**  
As mentioned in above chapter, fast sync requires the blocks and state downloading in parallel. During the sync, the state download will not be cancelled on peer drop but relying on a timeout to detect such scenario. If the peer reconnects before the timeout, it will be assigned a new state download task. After the timeout, the old one should be scheduled to discard, which easily trigger to discard the new one as well. Then the downloaded states miss some state entries and will not match with the latest block’s state root, which mistakely assumes the peer downloading is stalling, not fitting with the expected ones. As a result, it will drop the peer, which will automatically reconnect again, eventually being dropped due to the same reason.

• **Protocols:** sync, crash

• **Symptom:** peer churn, reconnect and disconnect again and again.

• **Fix:** track peer departure events instead of depending on a timeout mechanism

• **Spec:** fast sync must download the latest blockchain from the best peer

**#15364 data race in les.stop**

• **Protocols:** sync, crash

• **Symptom:** data race caused unsaved operation with wait group

• **Fix:** wg.Add should not happen inside the goroutine but before starting it

• **Spec:** wg.Add should be after wg.Wait
#16103 les: fix light fetcher database race

Fix #16101, which invalid memory address or nil pointer dereference. It avoids a potential chain db data race that can cause checkAnnouncedHeaders being called with a nil header. There is also another check for missing parent of validated header, which in theory should not happen but still it would be nice to know about if it happens.

(hk) les/fetcher is lightFetcher, which implements retrieval of newly announced headers. It also provide a peerHasBlock function for the ODR system to ensure that we only request data related to a certain block from peers who have already processed and announced that block. Fix: it add a lock to protect access to the fetcher’s internal state variables except sent requests.

#16509 wait all fetcher goroutines to exit before terminating

It spawns 5 fetcher goroutines to fetch the blockchain data separately. When terminating the downloader (e.g., Ctrl+C), the downloader may terminates itself without waiting all in-flight fetcher goroutines to exit. The problem is that wgCancel.Done() is only called when each of the fetchers return, but some of the fetchers (for example, the header fetcher as well as the state fetcher) can call dropPeer(), inside which if it is the master peer, it calls d.Cancel() which waits on wgCancel before it can return. The symptom is to cause geth to deadlock.

- **Protocols:** sync, cancel
- **Symptom:** deadlock
- **Fix:** move sync.waitGroup from the spawn to downloader internal to make sure all fetchers have exited when we call cancel

#16539 eth/downloader: fix for issue #16539

- **Protocols:** sync, cancel
- **Symptom:** deadlock
#2647 make fast sync resilient to critical section fails  It fixes a possible deadlock if header downloads is finished simulatenously with a sync cancellation. Therefore the header processor quits due to the abortion and never reads the success signal pending by the header downloader.

4.3.6  Potpourri

#627-2 proper locking to prevent parent unknown invalid blocks due to race in peer head info update  When called on an already connected peer, it means a newBlockMsg is received, and peer head info is update. To be specific, comparing peer’s current block hash with local block hash, if they are not matched, update peer with local td and current block hash. peer.run has a main loop for head section process, when receiving signal from AddBlock that head block of current best peer has been received, it will update the current block and its parent block. On the other hand, add peer will set chain info, which will reset current and parent blocks as nil. Therefore, the previous update parent block might race with the later setting current block nil, which has no parent block erro.

- **Protocols:** peer management
- **Symptom:** crash
- **Fix:** add a lock to protect peer head info update
- **Specification:** broadcast messages should happen after peer connect phase.

4.3.7  Mining

#681 Eventer and RPC in deadlock but only when console is started  The miner is not receiv- ing core.ChainHeadEvent because it blocks on a nil channel.

- **Protocols:** Mining
- **Symptom:** miner threads are mining on block #53xx while current head was actually #73xx.
Fix: miner starts a newly registered agent if the miner is running.

Gap: unknown, check code later

#1031 miner is not stopping at stop command  When it starts miner’s CpuAgent, it will create quit and quitCurrentOp channel and starts a update goroutine, which will close the quitCurrentOp channel and start mining a new block by using the PoW algorithm. While mining, it will reset the quitCurrentOp channel again. However when stopping the miner, it will close the quit and quitCurrentOp channel, which is already closed in the update goroutine, then miner is not stopping at stop command. The fix is to add a check in case of closing a closed channel.

Protocols: Mining

Symptom: crash

Fix: add a nil check when receiving a result from agent.

Spec: miner should be stopped at stop command.

#1103 geth crash after import finished  #1105 explained the race condition. The blockchain commands don’t need the full stack. With this change, p2p, miner, downloader, etc are no longer started for blockchain operations.

Protocols: Mining

Symptom: crash

Fix: the miner checks if it has received a block before using it.

Gap: It assumes that the blockchain db will not close during mining.

parent issue for ethash/PoW security issues
#1500 miner: fix current work data race  when arriving transactions would race with current work retrievals due to wrong lock usage.

#1587 miner: fixed worker race condition  The worker has background goroutine wait(), whenever the a block is mined, it will commitNewWork, which mainly update the current environment, and then commit the transactions, uncles and update the state, finally to push the work to agent to mine in another goroutine. Here, when we push the new work to agent, it commits a new work, which will update the self.current work to the agent. The fix is to give the current work to agent instead of depending on a global self.current variable.

- **Protocols:** Mining
- **Symptom:** corrupt work
- **Fix:** give the current work to agent instead of depending on a global self.current variable.

#1988-1 miner where blockhash in receipts and logs is left empty  When the transactions are processed in the miner for the pending block the block hash isn’t yet known. As a result the generated receipts and logs don’t yet contain the block hash.

- **Protocols:** mining
- **Symptom:** receipt and logs don’t contain block hashes.
- **Fix:** update the blockhash when the block has been mined.

#2949 Geth Miner: prevent attempts to close nil quit channel in agent  When the user starts mining, he can use the ”minerthreads” parameter to specify the number of parallel mining threads (default to the total number of processor cores), then Ethereum system will start corresponding number of agents. Therefore in face of miner.Start and miner.Stop in quick succession, it involves starting multiple agents and closing them with interleavings to some extend. In the go-ethereum implementation, it creates a quit channel when starting a agent and close it when stopping the lx
agent. Then the miner.Start doesn’t yet start some agent but the miner.Stop already aborts it, which implies a race between creating and closing a quit channel. It can result in a panic due to closing nil quit channel. The fix is to create the quit channel in the agent initialization phase instead of starting phase. The bug violates a general specification: closing a channel should happen after creating it.

- **Protocols:** mining, start/stop
- **Symptom:** crash
- **Fix:** move the quit channel from the start to initialization

**#3390 ethstats: check if received event is valid** There is a race condition in the ethstats.Service#loop function. When the node is stopped the event mux is stopped which causes channels associated with subscriptions to be closed. This returns a nil value which isn’t checked.

- **Protocols:** mining, stop
- **Symptom:** crash
- **Fix:** add a nil check

**#3430 miner: clean up unconfirmed mined block tracking** When we mine a new block, the block is currently placed into some ring buffer to track whether a few blocks later it becomes a canonical block or gets discarded as a side chain. This logic is to handle this was scattered over the miner code, which made it really hard to reason about, furthermore it was quite a racy code producing all kinds of warning from the race detector when mining on an easy chain.

**#3431 Miner race fixes** fix two data race in the miner (1) fix a race between remote agent start / loop, where start() method try to create the quit channel, but another loop() method keeps accessing them in a select. Now, it already remove the variable accesses welcome from the loop() and passed

1x1
the channel as parameters. Possibly a whole rewrite would be welcome on this part as recreating
channel concurrently is a very bad juju.

(2) fix data race on setting etherbase/extra data. miner set the etherbase and extra data fields of
the worker, even though the worker might have seen still alive with a previous run. Based on the
fix code, it move the SetExtra from Miner to Worker. It makes sure the fields are set using accessor
methods with proper locking.

#14407 consensus/ethash: fix a timestamp update race When generating ethash verification
caches and mining datasets, we’ve retrieved the current epoch’s struct and bumped its timestamp,
generated it and bumped its timestamp again. This first bump was racy as it wasn’t protected by a
lock. In reality however we don’t really want to bump the timestamp before generating the data,
only for entities newly created.

4.3.8 Console

#1220-2 fix a race condition accessing the gas limit when the client queries the gas limit, it
will return the current block’s gas limit through the chain manager. The insert function will inject a
block into the current blockchain, wihch will update the chain manager’s current block. Therefore,
it might hang there until the block insertion finished.

- **Protocols:** Chain Import

- **Symptom:** peformance issue, query delayed

- **Fix:** add a read lock

- **Gap:** unknown

#1224 collect and report import progress

- **Protocols:** sync
• **Symptom:** performance issue, query delayed

• **Fix:**

• **Spec:**

#1323 race condition in http interface

#2354 miner: fetch pending block/state in on go  The miner has two utility methods to retrieve the currently pending stats: PendingState and PendingBlock. Since certain methods require both of these (e.g., gas estimation), calling them individually introduces a data race whereby the block/state may go out of sync in between calls. The fix is to merge two calls into one, ensuring that retrieving them both are protected by the same mutex.#pending block and state are only known by the miner.

#1785 Parity invalid state root error while importing new block  When importing a new block, the blockchain will commit the block and the transaction database will commit corresponding transactions as well, which means the transaction db will close transactions. This bug is due to transaction db commits happens after blockchain committing the block. Then if there is any update of transactions, it will lead to a mismatch between blockchain’s and db’s transactions. The specification is that blockchain should commit best block after db transactions have been committed.

### 4.4 Implementation Bugs

#### 4.4.1 Double Close

#### 4.4.2 Access before Creating

• #1496 data race in filter creation and event firing

• #1665 geth crashed on filter creation

• #2711 filter race: concurrent map read and write
• #15440 data race caused channel assignment after starting reading goroutine

• #15343 core: fire tx event on

• #16682 eth/filter check nil when unsubscribe

4.4.3 Deadlock

1865, #2305, 1154, 1865, 2949, 1031, 15138?, 3557, 15364

Figure 4.3: Transfer Money.

Transfer Money [3] In Figure 4.3, it pointed out a race condition vulnerability that allows the attackers to steal money from users’ account. An attack calls transfer function when there is an external call like withdrawBalance. It’s possible that the attack transfers some money during receiving withdraw request but before the balance is set to 0. Above attack occurs because of the cross function race condition, the same bug can also happen across multiple contracts if they share state. We specify that transactions should be atomic executions given the same account address, which can protect the account balance from being modified in concurrent transactions. This bug is caused assuming that a single transaction execution is atomic.

4.5 Comparisons

Protocol concurrency bugs mainly come from some data race, like concurrent transactions and blocks, and concurrent blockchain update. Synchronization related bugs mainly lead to partial
deadlock. Since there are 5 independent and parallel downloadings, any part of them failed, it will lead to other parts wait indefinitely.
CHAPTER 5
CONCURRENCY SPECIFICATION

5.1 Violated Specifications

We can see from above section, current specifications are not enough to cover concurrency bugs. Therefore we need more concurrency specifications. In this section we will list existing concurrency specifications that are able to uncover bugs. Meanwhile, we will infer some other concurrency specifications for two purposes. (1) detect bugs that existing specification cannot cover. (2) look for new concurrency bugs according to the inferred specifications.

Most implementation related bugs can be detected with existing specification. However, the protocol related bugs are hard to find. Therefore, we need adds more concurrency specifications to solve the protocol concurrency bugs. And we should figure out what current specifications are violated (and what assumptions are violated) for the implementation concurrency bugs.

As the popularity of Bitcoin, more and more people try to make the Bitcoin implementation better. However, they are essentially making bugs for fixing bugs while being compatible with Bitcoin Core. The basic reason is lacking of a standard specification for the Bitcoin implementation. For this purpose, Ethereum provides a reference specification [11] that all developers agree on. Therefore, developers can implement an Ethereum client in whatever language they want.

5.1.1 Known

In this section, we will introduce some known specifications in Ethereum yellow paper [11]. It mainly introduced the state, transaction and block specifications, which can provide a good guidance for our data concurrency analysis. However, with regard to protocols like mining and synchronization, it doesn’t mention any of them. As for transaction specification, [11] focus on transaction format regulation, intrinsic validity and execution.

Transaction's Intrinsic Validity
The transaction is well-formed RLP, with no additional trailing bytes;

The transaction signature is valid;

The transaction’s nonce is valid (equivalent to the sender account’s current nonce);

The gas limit is no smaller than the intrinsic gas used by the transaction;

the sender’s account balance contains at least the cost required by up-front payment.

Transaction Execution The execution of a valid transaction has follows specifications:

• The nonce of the account of the sender is increment by one.

• The balance is reduced by part of the up-front cost.

• The sum of the transaction’s gas limit and the gas utilised in this block prior must be no greater than the block’s gasLimit.

GasLimit is the current block’s gas limit, it discards any transaction above that as unexecutable, since it will never fit into the block. [11] has a rigorous specification for the construction of a formal block structure.

Block Header Validity

• The block number is the parent’s block number incremented by one.

• The canonical difficulty of a block header is defined as D(H), which can we inferred the next block’ difficulty value is larger than the current block’s.

• The canonical gas limit of a block of header must fulfil the relation ..

• The timestamp of a block must fulfile the relation ..

• The nonce of a block must satisfy ....
**Holistic Validity** We can assert a block’s validity if and only if it satisfies several conditions: it must be internally consistent with the ommer and transactions block hashes and the given transactions.

**Block Finalization** The process of finalising a block involves four stages:

- Validate (or if mining, determine) ommer;
- Validate (or if mining, determine) transactions;
- Apply rewards;
- Verify (or, if mining, compute a valid) state and nonce.

The state trie requires a simple database backend that maintains a mapping between addresses (160-bit identifiers) and account states. We name this underlying database the state database.

- The root node of this structure is cryptographically dependent on all internal data and as such its hash can can be used as secure identity for entire system state.
- Being an immutable data structure, it allows any previous state to be recalled by simply altering the root hash accordingly

### 5.1.2 Inferred

Based on known specifications, we notice that there aren’t any concurrency related specification in Ethereum. For example, with respect to the transaction nonce value, the yellow paper specify that the transaction nonce should equal to the sender account’s current nonce. Prior to the execution, the transaction nonce should be incremented by one. This specification works in face of serialized transactions. However, when the user submits two concurrent transactions as in #2793, both of them will exploit this specification and increment its own nonce value. As a result, these two transactions have the same nonce and one of them will be missed in the blockchain. For this transaction loss case, blockchain is in urgent of concurrency specifications. Here, we rule that there is no two transactions with the same nonce.
Likewise, as for the account balance, the transaction execution defined that the balance is reduced by transaction value. This specification works only if transactions related to the same account are executed sequentially. If it’s the case of [3], transfer transaction finishes before withdraw transaction set the balance to be zero, then an attack can take advantage of this race condition to steal money. Therefore, blockchain should guarantee that transactions from the same sender are an atomic execution.

The yellow paper specifies the transaction format and execution steps, which leaves transactions adding to and removing from the network empty. In the Ethereum system, the newly submitted transaction will be stored in a transaction pool and the bad and used transactions will be removed from the pool. #2793 is an example of two transactions submitted to the transaction pool and get the same nonce value. #1865 is a counterexample of removing transactions from the pool. More specifically, once a transaction is included into the blockchain, the chainmanager will post a RemovedTransactionEvent, notifying the transaction pool to remove the corresponding transaction. However, the transaction pool cannot get the chainmanager’s lock because it’s locked prior to posting RemovedTransactionEvent, which causes a deadlock bug. Based on this, we specify that RemovedTransactionEvent should successfully remove the transaction from the pool. Likewise, any transaction adding events should guarantee the transactions are added to the pool.

Among our 13 concurrency bugs, they indeed satisfy the specification in the yellow paper. However, two valid transactions, blocks or states still trigger the bugs. Therefore, we needs more concurrency specifications. Meanwhile, 5 out 13 bugs are synchronization related bugs. Yellow paper doesn’t regulate developers how to specify the synchronization. That’s why there are about 1000 reports about synchronization failures and developers always recommend to clean the database and resync from the beginning. In Table 5.1, we list all concurrency specifications for our 13 bugs.
<table>
<thead>
<tr>
<th>Bugs</th>
<th>Protocol Concurrency Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>#2950 Geth</td>
<td>There is no two transactions with the same nonce for the same account address</td>
</tr>
<tr>
<td>#transfer money [3]</td>
<td>transaction executions should be atomic for the same account address</td>
</tr>
<tr>
<td>#2793 Geth</td>
<td>after successfully mining 6 succession blocks, the blockchain should confirm the first block.</td>
</tr>
<tr>
<td>#3412 Geth</td>
<td>pendingState’s nonce can never be less than currentState’s nonce</td>
</tr>
<tr>
<td>#15138 Geth</td>
<td>Fast sync will download latest block from the peer.</td>
</tr>
<tr>
<td>#1843 Geth</td>
<td>Sync downloads the equal number of hashes and blocks.</td>
</tr>
<tr>
<td>#3557 cpp-ethereum</td>
<td>blocks in BlockQueue cannot exceed the maximum limit.</td>
</tr>
<tr>
<td>#1785 Parity</td>
<td>commit applies best block only after db transaction have been commited</td>
</tr>
<tr>
<td>#996 Geth</td>
<td>each block (except genesis block) should have a parent block.</td>
</tr>
<tr>
<td>#974 Geth</td>
<td>each block’s parent hash should equal to the previous block’s hash.</td>
</tr>
<tr>
<td>#2523 Geth</td>
<td>during dropping peers, it’s not allowed to add same peers.</td>
</tr>
<tr>
<td>#1569 Parity</td>
<td>sealing_.working should happen before transaction queue operations.</td>
</tr>
<tr>
<td>#15364 Geth</td>
<td>wg.add should happen after wg.wait.</td>
</tr>
<tr>
<td>#2949 Geth</td>
<td>Close a channel should happen after making a channel.</td>
</tr>
<tr>
<td>#2305 Geth</td>
<td>If a block is added to the futureBlocks, it will be inserted into blockchain</td>
</tr>
<tr>
<td>#1154 Geth</td>
<td>If a block is added to the futureBlocks, it will be inserted into blockchain.</td>
</tr>
<tr>
<td>#1865 Geth</td>
<td>RemovedTransactionEvent should remove corresponding transaction from the pool.</td>
</tr>
<tr>
<td>#1031 Geth</td>
<td>It’s not allowed to close a channel twice</td>
</tr>
<tr>
<td>#704-2</td>
<td>assume no new peers can be added while stopping the server</td>
</tr>
<tr>
<td>#627-1</td>
<td>assume to add new peers serially</td>
</tr>
<tr>
<td>#593</td>
<td>assume the p2p protocol will always accept messages rather quickly</td>
</tr>
<tr>
<td>#1287</td>
<td>assume the sync batch blocks downloading will not mid fail</td>
</tr>
<tr>
<td>#2211</td>
<td>assume the insertion of a batch of headers will not fail before writing all</td>
</tr>
</tbody>
</table>

Table 5.1: **Concurrency Specifications.** *DaLo (Data Loss), HgDl (Hang Deadlock), HgIl(Hang Infinite loop), StIn(State Inconsistency), CrNd (Crash Node), CrPr(Crash Process), CrTh (Crash Thread)*
REFERENCES


