Ethereum Reliability Engineering

Analysis and summary on how Ethereum consensus bugs are debugged.

Wei Tang

August 21, 2019

Contents

In this report, we analyze past Ethereum consensus incidents, and attempt to form a general framework for Ethereum Reliability Engineering. While we try our best to improve the software engineering process to **prevent** bugs from happening, it is also equally important that we have methodologies in place to **react**, in case a bug does happen in consensus. This reduces the amount of work developers need to do for *firefighting*, so that more time can be focused on writing and delivering new features.

Definition

Developing a blockchain *software*, such as an Ethereum node implementation, is not only about shipping the software. The software being shipped collectively form a *network*, and it interacts with the network. The network has its *reliability* definition that is related but independent of the software. It can be argued that maintaining the network reliability is equally important, or in many cases, more important, compared with maintaining the reliability of the software.

This is the case because of two things. First, what's being developed is a decentralized network. A node being down, either due to a hardware failure or a software bug, does not necessily have impact on the network, or even the user. For many times, the software can just be restarted, without even any impact on the user. Second, the network maintains some sorts of global state. This means while the network is decentralized, a global failure is still possible, making the network temporarily unusable, even when all the nodes are still up. This can be the result of a consensus split of two major softwares being used in the network, or other kind of attacks.

Due to the above, besides developing the software, we also need to have methodologies in place for maintaining the reliability of the network.

Availability

Note that the concept of availability we use here is related to availability as in reliability, but not the availability concept in Byzantium Fault Tolerence systems.

One important metric to define a blockchain's availability is through its non-attack transaction throughput. We first define an ideal metric, and then specialize that metric to give a practical definition.

We define an ideal observer who is able to inspect all nodes running in this network, and can travel forward or backward in time. While in many consensus systems there is no concept of finalization, we can define a block being finalized, if at any time when the observer travels forward in time, the block is canonical in chain across all concerned node implementations. With this, a transaction being finalized can be defined as when the block the transaction is included becomes finalized. The network transaction throughput is thus the amount of transactions being finalized over `S` seconds. Practically, we can define a block is "final" if it has been confirmed by `B` ancestry blocks.

With this, we call a blockchain network being *up*, if at a measured period, its non-attack transaction throughput is over `N`. Otherwise we call it being down. The availability then can simply be defined by the blockchain's uptime divided by total time.

This definition is for measurement purpose, so the reader may find some of the above concept being subjective. For example, whether a transaction is "normal" or "attack" transaction is up to the measurer's decision. Another note is that for the above transaction throughput definition, we only count "final" transactions when there is an agreed canonical chain on all (or most) node implementations. This is because in case of a consensus split, the network is effectively not functional.

Principles

Below we list several principles in maintaining a blockchain system. Those are the things that we think would benefit us, or at least are worth debating on.

Acceptable risks

As many reliability engineering practices have pointed out, designing a 100% system is not only inpractical, but also it may be worse for the service. Extreme reliability costs development resources, and limits how fast new features can be deployed. The opportunity cost can be critical in evolving fields such as blockchains.

This does not mean we shouldn't carry out extensive software reviews, security audit and other safety measures before deploying anything on the blockchain. Those **prevention measures** are absolutely necessary. However, in addition to those prevention measures, we should adopt the mentality that *blockchain developers are humans*, and we can make mistakes no matter how extensive those prevention measures are. As a result, we should also have **reaction measures** in case mistakes happen. This also includes how we should design the system so that in case of failure, the damage can be minimized.

Living System

Blockchain is a living system. This means we don't deploy it once and freeze the rules. Requirements change, and the conditions in which the blockchain is running also change. Features deployed on the blockchain should be extensible. In addition, because of this, we should also properly prioritize **automation** for repeatitive work (such as setting the hard fork block number) so that burden on developers can be reduced.

History is Important

Past experience has shown us that what has been written into the state plays an important role when designing new features for the system. It is important that we make sure new features don't accidentially change behavior of any existing state. Otherwise it can even result in users' fund being lost.

Types of Failures

Failures can happen in different layers.

- **Local layer**: happens in local backend, filesystems or operating systems.

- **Networking layer**: happens in P2P network communication.
- **Consensus layer**: happens in fork choice or state transition.
- **Smart contract layer**: happens in smart contract applications.

In the current version of this report, we focus mostly on **consensus layer** failures and partially touch **smart contract layer** failures.

Consensus Layer Failures

Node Crash

Given incorrect input, the node may crash, and thus be out of the network. The user may lose mining rewards, or not get transaction confirmed because of this. If this is an attack, the attacker may be able to crash a group of nodes at the same time. However, it is unlikely that the attacker will be able to crash multiple implementations at the same time.

This category of failure will only need one implementation to release a fix and without coordination of the whole network.

Denial of Service or Specification Attack

The attacker can find DoS vector in specification and exploit that to the whole network. This will cause the network to run slowly. It has impact on availability, but normal transactions may still be able to be processed.

If this happens, the network will need to conduct an emergency hard fork. Availability may be damaged but the network can still be up.

Consensus Split

Two different implementations contain bugs, which make them to have different ideas about what should be the next valid state, or what is a valid block. To make discussions below easier, we consider two implementations `A` and `B`, define the set of states or blocks that implementations `A` and `B` think is valid as `SA` and `SB`, respectively. If `SA` is a subset of `SB`, then the network still have a chance to self-heal if the collective hash power of `A` is greater than `B`. However, in all other cases, each implementation will only try to build on top of its own best blocks, and the chain is split forever if there is no human intervention.

During the split, the chain becomes mostly unusable because it's effectively two different chains, if no majority is selected. This can completely destroy its availability during this period.

Structural Failure

This category refers to the case if the fork choice rule being used is exploited (for example, in case of 51% attacks), or if any of the crypto algorithm we use is exploited (for example, keccak256 becomes vulnerable).

Smart Contract Layer Failures

Two cases can result in smart contract layer failure. The first case is where the smart contract contains a bug. The second case is where a new feature added to the blockchain caused the smart contract to exhibit a new bug, which previously does not exist. In this report, we focus only on the second case.

Severities of Failures

Failures in consensus layer or those in base layer but impact smart contracts, usually by default gain its severity as *critical*. Below, however, we want to define several sub-categories for consensus layer failures. Because not all consensus failures should be treated equally, this can help us to understand what different strategies we need to apply to individual ones, and also how we can reduce a failure's severity through engineering.

Consensus Layer Failures

Type A

A consensus layer failure that minimally affects transaction throughput. This category can apply to *State Transition Crash*, if miners or validators have opted in to run multiple implementations at the same time. When one implementation crashes, another one can immediately take over and continue to build blocks and accept transactions.

Type B

Failures that considerably impact transaction throughput. We define this case for when transaction is still being confirmed, but considerably slower. This is the case for any denial of service attack on specifications, if the network is being exploited for its resources.

Type C

When the network is effectively halted and does not confirm transactions at all. For Proof of Work system, this is the case if the network is under 51% attacks. This is currently also the case for most of the *Consensus Split* scenarios. Later in this report, we will try to provide a method on how do we turn *Consensus Split* scenarios from *Type C* to *Type B*.

Note that in *Consensus Split* scenario, even when a bug results in users' fund being lost in one chain, as long as another chain is healthy, we can still classify it as *Type C* but not *Type D*, because later we can always reorg back to the healthy chain.

Type D

Or *Doomsday*, is when the network is completely broken and users fund can be completely lost. This category is reserved for scenarios such as if we find an exploit in Proof of Work algorithm that allows an attack to craft new valid blocks easily, or if quantum computing becomes practical and completely breaks some of the cryptography algorithms we use.

Smart Contract Layer Failures

We also define two smart contract layer severity so that discussions below can be easier. But note that this section can be incomplete, compared with *Consensus Layer Failures*.

Type S-1

In this case, we have a targeted group of users' fund being lost. The bug, however, is not exploitable further and other users are not affected.

Type S-2

In this case, we have a smart contract bug that potentially affects the whole network.

Prevention

In this section, we documents the current measurements of preventing any bugs from being written in the first place.

- **Developer Review**: when a new feature is being proposed, it will go through a lengthy review process by developers to see if any issues have been found.
- **Security Audit**: the proposal will also possibly be reviewed by independent auditers to further ensure it's safe.
- **Testing**: the feature is implemented in clients, which will then be tested. The hivetests continuously run for all client releases. For each major hard fork, fuzzy testing is also carried out.

There are also several other lessons learned that can help to prevent mistakes, which we will discuss in post-mortem section.

Reaction

Below we briefly describe the current process how consensus bugs are detected and being fixed.

Monitor

Tools like `ethstats` and `forkmonitor` are used to monitor the health of the network.

Analyze

The goal of this process is to provide a reproducable test case that can be then debugged by developers from different node implementations.

A failure might be detected during the normal development process, where the bug has not been exploited, or it might be detected through monitoring. For the former case, the test case is usually prepared by the discoverer, or the bug might be really simple that a fix is trivial. For the later case, the reason of the failure is not usually already known. The steps to identify the cause and produce a test case usually work like this:

- We first find out which block caused the failure. This is usually self-evident in case of a consensus split – find the last common block.
- It then involves finding out which transaction caused the failure. There is no current easy way to do it. One way to do this is to debug print the intermediate state root, and check from which transaction the root becomes different.

- Once the transaction is identified. The next step would be to create a reproducable test case for the failure. evmlab's reproducer is capable of doing this.

Debug

Once the cause is identified, developers then proceed on to debug the actual issue. This is usually done by deriving the debug output of opcode traces via specific EVM utilities that are capable of executing the test cases, then find out the first difference in the trace. This trace is usually sufficient to fully understand the bug.

Release

Once the bug is fixed, an emergency release is usually scheduled.

Improvements

In this section, we list several potential improvements that may be applied in our reliability engineering.

Make Consensus Split Less Dangerous

We explained before why *Consensus Split* is a *Type C* failure. In many cases, once a split happens, it is not possible for the network to self heal unless new releases of the softwares are made. During this period, the split is permanent and unless the majority chain has already been determined, it's not possible to claim new transactions have been confirmed. Here, we explain how we may be able to turn it into a *Type B* failure with the network's coordination.

The reason why the split is usually permanent is that both node implementation `A` and node implementation `B` produce blocks `BA` and `BB` that wouldn't be accepted by each other. From this point on, no matter how the hashrate distributes in the network, both side refuses to reorg to the other chain.

To prevent this, the network must have collectively done things to make sure the majority does not produce `BA` and `BB` initially. This can be done through N-version programming. When miners produce a block, for transactions to be included, we run it through multiple node implementations, and only include those transactions where all node implementations agree.

To make this process run smoothly, it is necessary to make our node implementations modular, and most likely run on different OS processes. The **importer** deals with importing new blocks, either received locally or on the network, and broadcast blocks it mined. The **producer** deals with transaction pool, and authoring new blocks. When a new block is to be authored, the producer picks up transactions from its transaction pool, send it to multiple EVM backends and then verify the results. Only after then the producer sends the block to the importer, where it forms a chain, or waiting to be mined.

In normal operation, this architecture would mostly perform identical to our current node implementations. When consensus failure happens, the producer would reject transactions that would have caused consensus split, thus preventing a *Type C* failure. However, in this case, the attacker can craft large amount of new transactions that he or she knows would be rejected, broadcast to the network, and attempt to carry out denial of service attack. As long as the producer is rate limited, we may suffer from major impact in transaction throughput, but normal transactions still have a chance to be processed. Thus it becomes a *Type B* failure.

Simulation

Many of us would agree that our current process for fixing a consensus failure remains much to be improved. When a failure does happen, it is usually the case that we don't feel fully prepared because it is rare, yet does occur. Therefore, it is important that we carry out more simulations for hard forks and for handling consensus splits.

Usually we want to launch short-lived testnets to carry out those simulations or experiments. Launching a network should be a really cheap operation and may be even integrate in CI for automation.

Automation

Carrying out hard fork and adding new features to the blockchain is a repeatitive process. It is thus natural that we should apply more automation to it. This can possibly also include migration framework for the blockchain that allows it to transition to a new format.

Lessons Learned

In this section we examine some past incidents to gain some insights and lessons learned. The partial list of all incidents we examine include:

- **The Shanghai Attacks**: (2016.11, *Specification Attach*, *Type B*) the attack that ultimately led to hard fork of EIP-150.
- **EIP-161 deletion and revert**: (2016.11, *Consensus Split*, *Type C*) a bug that happened due to Geth and Parity having different bahaviour in account deletion when the state reverts.
- **EIP-1283 refund overflow**: (2018.11, *Consensus Split*, *Type C*) a bug in implementation of gas refund counter for EIP-1283 that caused an consensus split. This is exploited in Ropsten testnet.
- **EXTCODEHASH empty account**: (2018.11, *Consensus Split*, not exploited) a bug due to Geth and Parity returning different results for `EXTCODEHASH` opcode for empty accounts.
- **EIP-1283 reentry**: (2019.01, *Smart Contract Attack*, *Type S-2*) design in EIP-1283 accidentially opens previous contract to reentry-attack. This ultimately resulted in Constantinople being delayed.
- **EIP-98 validation**: (2018.06, *Consensus Split*, *Type C*) issue that caused client to still accept the transaction even when it is unsigned, and EIP-98 is disabled. This is exploited on Kovan testnet.
- **Byzantium DoS**: (2017.10, *Node Crash*, not exploited) potential issue that potentially allow an implementation to be DoSed.
- **BLOCKHASH result**: (2015.10, *Consensus Split*, not exploited) bug that can causes consensus split due to incorrect result returned by `BLOCKHASH`.
- **TheDAO**: (2016.06, *Smart Contract Attack* and *DoS*, *Type B* and *Type S-1*) a bug in smart contract caused the fund being exploited by the attacker. The later soft fork is pointed out to potentially be a DoS vulnerability. It eventually leads to a hard fork.
- **EIP-999**: (2017.11, *Smart Contract Attack*, *Type S-1*) a bug in multisig contract that caused a library smart contract being killed, thus the funds frozen.

Observations

- **Consensus split is one of the most common type of consensus failure.** And unfortunately, it is also the kind that is not easy to debug. It is thus important that we spend devops time to make debugging consensus split much easier.
- **Nearly all bugs are about interactions with state or gas costs.** This means that while EVM is probably the source of many smart contract layer bugs, it does not mainly contribute to consensus layer bugs. Just replacing EVM with WebAssembly wouldn't have prevented those bugs. It must be combined with better gas cost models and state operation models.