Istanbul Byzantine Fault Tolerance · Issue #650 · ethereum/EIPs

Libreta:	Primera Libreta
Creado:	30/07/2018 20:55
URLOrigen:	https://github.com/ethereum/EIPs/issues/650

search-shortcut-hint.svg
30/07/2018 20:58, 413 B

Pull requests
Issues
Marketplace
Explore

ethereum / **EIPs**

Watch 763

Code | Issues 259 | Pull requests 76 | Projects 0 | Insights

Istanbul Byzantine Fault Tolerance #650

Open **yutelin** opened this issue on 22 Jun 2017 · 27 comments

vutelin commented on 22 Jun 2017 • edited

Change log:

- Aug 8, 2017:
  - Add gossip network
- Jul 24, 2017:
  - Add block locking mechanism.
  - Performance/bug fixes.
- Jun 26, 2017:
  - Add `extraData` tools.
  - Update notes and discussions on zero gas price transaction
- Jun 22, 2017:
  - Initial proposal of Istanbul BFT consensus protocol.

Pull request

ethereum/go-ethereum#14674

Istanbul byzantine fault tolerant consensus protocol

Note, this work is deeply inspired by Clique POA. We've tried to design as similar a mechanism as possible in the protocol layer

Terminology

- **Validator**: Block validation participant.
- **Proposer**: A block validation participant that is chosen to propose block in a consensus round.
- **Round**: Consensus round. A round starts with the proposer creating a block proposal and ends with a block commitment or r
- **Proposal**: New block generation proposal which is undergoing consensus processing.
- **Sequence**: Sequence number of a proposal. A sequence number should be greater than all previous sequence numbers. Curr
- **Backlog**: The storage to keep future consensus messages.
- **Round state**: Consensus messages of a specific sequence and round, including pre-prepare message, prepare message, and
- **Consensus proof**: The commitment signatures of a block that can prove the block has gone through the consensus process
- **Snapshot**: The validator voting state from last epoch.

Assignees
No one assigned

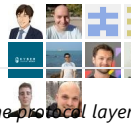
Labels
None yet

Projects
None yet

Milestone
No milestone

Notifications
You're not receivi
from this thread.

16 participants



Consensus

Istanbul BFT is inspired by Castro-Liskov 99 paper. However, the original PBFT needed quite a bit of tweaking to make it work with expect to generate a verifiable new block rather than a bunch of read/write operations to the file system.

Istanbul BFT inherits from the original PBFT by using 3-phase consensus, `PRE-PREPARE`, `PREPARE`, and `COMMIT`. The system can tc `PRE-PREPARE` message from the proposer, validators enter the state of `PRE-PREPARED` and then broadcast `PREPARE` message. Thi chain. Lastly, validators wait for `2F + 1` of `COMMIT` messages to enter `COMMITTED` state and then insert the block to the chain.

Blocks in Istanbul BFT protocol are final, which means that there are no forks and any valid block must be somewhere in the main c would cause an issue on block hash calculation. Since the same block from different validators can have different set of `COMMIT` sig

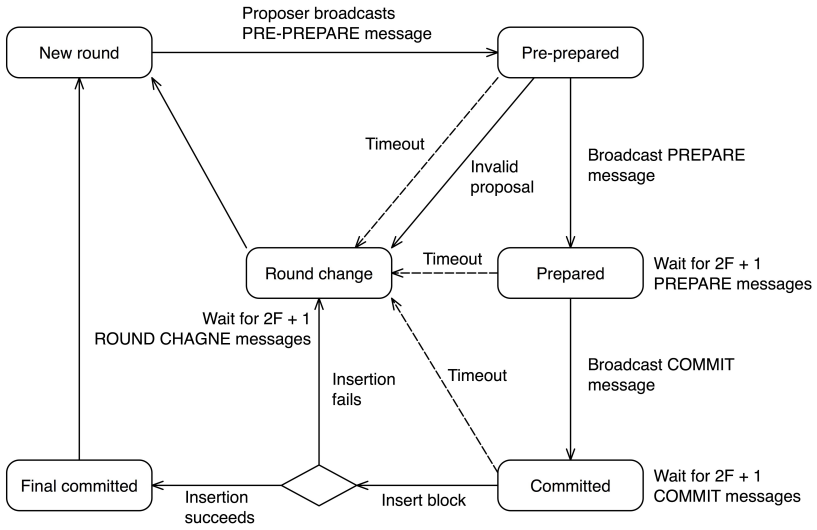
Consensus states

Istanbul BFT is a state machine replication algorithm. Each validator maintains a state machine replica in order reach block consensu

States:

- `NEW ROUND`: Proposer to send new block proposal. Validators wait for `PRE-PREPARE` message.
- `PRE-PREPARED`: A validator has received `PRE-PREPARE` message and broadcasts `PREPARE` message. Then it waits for `2F + 1`
- `PREPARED`: A validator has received `2F + 1` of `PREPARE` messages and broadcasts `COMMIT` messages. Then it waits for `2F +`
- `COMMITTED`: A validator has received `2F + 1` of `COMMIT` messages and is able to insert the proposed block into the blockchai
- `FINAL COMMITTED`: A new block is successfully inserted into the blockchain and the validator is ready for the next round.
- `ROUND CHANGE`: A validator is waiting for `2F + 1` of `ROUND CHANGE` messages on the same proposed round number.

State transitions:



- `NEW ROUND -> PRE-PREPARED`:
  - **Proposer** collects transactions from txpool.
  - **Proposer** generates a block proposal and broadcasts it to validators. It then enters the `PRE-PREPARED` state.
  - Each **validator** enters `PRE-PREPARED` upon receiving the `PRE-PREPARE` message with the following conditions:
    - Block proposal is from the valid proposer.
    - Block header is valid.
    - Block proposal's sequence and round match the **validator**'s state.
  - **Validator** broadcasts `PREPARE` message to other validators.
- `PRE-PREPARED -> PREPARED`:
  - **Validator** receives `2F + 1` of valid `PREPARE` messages to enter `PREPARED` state. Valid messages conform to the followir
    - Matched sequence and round.
    - Matched block hash.
    - Messages are from known validators.
  - **Validator** broadcasts `COMMIT` message upon entering `PREPARED` state.
- `PREPARED -> COMMITTED`:
  - **Validator** receives `2F + 1` of valid `COMMIT` messages to enter `COMMITTED` state. Valid messages conform to the followir

    - Matched sequence and round.
    - Matched block hash.
    - Messages are from known validators.
- `COMMITTED -> FINAL COMMITTED`:
    - **Validator** appends `2F + 1` commitment signatures to `extraData` and tries to insert the block into the blockchain.
    - **Validator** enters `FINAL COMMITTED` state when insertion succeeds.
- `FINAL COMMITTED -> NEW ROUND`:
    - **Validators** pick a new **proposer** and starts a new round timer.

Round change flow

- There are three conditions that would trigger `ROUND CHANGE`:
    - Round change timer expires.
    - Invalid `PREPREPARE` message.
    - Block insertion fails.
- When a validator notices that one of the above conditions applies, it broadcasts a `ROUND CHANGE` message along with the pro
    - If the validator has received `ROUND CHANGE` messages from its peers, it picks the largest round number which has `F + 1`
    - Otherwise, it picks `1 + current round number` as the proposed round number.
- Whenever a validator receives `F + 1` of `ROUND CHANGE` messages on the same proposed round number, it compares the rece
- Upon receiving `2F + 1` of `ROUND CHANGE` messages on the same proposed round number, the **validator** exits the round cha
- Another condition that a validator jumps out of round change loop is when it receives verified block(s) through peer synchron

Proposer selection

Currently we support two policies: **round robin** and **sticky proposer**.

- Round robin: in a round robin setting, proposer will change in every block and round change.
- Sticky proposer: in a sticky proposer setting, propose will change only when a round change happens.

Validator list voting

We use a similar validator voting mechanism as Clique and copy most of the content from Clique EIP. Every epoch transaction rese

For all transactions blocks:

- Proposer can cast one vote to propose a change to the validators list.
- Only the latest proposal per target beneficiary is kept from a single validator.
- Votes are tallied live as the chain progresses (concurrent proposals allowed).
- Proposals reaching majority consensus `VALIDATOR_LIMIT` come into effect immediately.
- Invalid proposals are not to be penalized for client implementation simplicity.
- A proposal coming into effect entails discarding all pending votes for that proposal (both for and against) and starts with a cle

Future message and backlog

In an asynchronous network environment, one may receive future messages which cannot be processed in the current state. For e

Optimization

To speed up the consensus process, a validator that received `2F + 1` of `COMMIT` messages prior to receiving `2F + 1` of `PREPARE`

Constants

We define the following constants:

- `EPOCH_LENGTH`: Number of blocks after which to checkpoint and reset the pending votes.
    - Suggested `30000` for the testnet to remain analogous to the main net `ethash` epoch.
- `REQUEST_TIMEOUT`: Timeout for each consensus round before firing a round change in millisecond.
- `BLOCK_PERIOD`: Minimum timestamp difference in seconds between two consecutive blocks.
- `PROPOSER_POLICY`: Proposer selection policy, defaults to round robin.
- `ISTANBUL_DIGEST`: Fixed magic number `0x63746963616c2062797a616e74696e65206661756c7420746f6c6572616e6365` of `mixD`
- `DEFAULT_DIFFICULTY`: Default block difficulty, which is set to `0x0000000000000001`.
- `EXTRA_VANITY`: Fixed number of extra-data prefix bytes reserved for proposer vanity.
    - Suggested `32 bytes` to retain the current extra-data allowance and/or use.
- `NONCE_AUTH`: Magic nonce number `0xffffffffffffffff` to vote on adding a validator.

- `NONCE_DROP` : Magic nonce number `0x0000000000000000` to vote on removing a validator.
- `UNCLE_HASH` : Always `Keccak256(RLP([]))` as uncles are meaningless outside of PoW.
- `PREPREPARE_MSG_CODE` : Fixed number `0` . Message code for `PREPREPARE` message.
- `COMMIT_MSG_CODE` : Fixed number `1` . Message code for `COMMIT` message.
- `ROUND_CHANGE_MSG_CODE` : Fixed number `2` . Message code for `ROUND CHANGE` message.

We also define the following per-block constants:

- `BLOCK_NUMBER` : Block height in the chain, where the height of the genesis block is 0.
- `N` : Number of authorized validators.
- `F` : Number of allowed faulty validators.
- `VALIDATOR_INDEX` : Index of the block validator in the sorted list of current authorized validators.
- `VALIDATOR_LIMIT` : Number of validators to pass an authorization or de-authorization proposal.
  - Must be `floor(N / 2) + 1` to enforce majority consensus on a chain.

Block header

We didn't invent a new block header for Istanbul BFT. Instead, we follow Clique in repurposing the `ethash` header fields as follows

- `beneficiary` : Address to propose modifying the list of validator with.
  - Should be filled with zeroes normally, modified only while voting.
  - Arbitrary values are permitted nonetheless (even meaningless ones such as voting out non validators) to avoid extra com
- `nonce` : Proposer proposal regarding the account defined by the beneficiary field.
  - Should be `NONCE_DROP` to propose deauthorizing beneficiary as a existing validator.
  - Should be `NONCE_AUTH` to propose authorizing beneficiary as a new validator.
  - **Must** be filled with zeroes, `NONCE_DROP` or `NONCE_AUTH`
- `mixHash` : Fixed magic number `0x63746963616c2062797a616e74696e65206661756c7420746f6c6572616e6365` for Istanbul block
- `ommersHash` : **Must** be `UNCLE_HASH` as uncles are meaningless outside of PoW.
- `timestamp` : **Must** be at least the parent timestamp + `BLOCK_PERIOD`
- `difficulty` : **Must** be filled with `0x0000000000000001` .
- `extraData` : Combined field for signer vanity and RLP encoded Istanbul extra data, where Istanbul extra data contains validatc

```
type IstanbulExtra struct {
        Validators    []common.Address  //Validator addresses
        Seal          []byte                    //Proposer seal 65 bytes
        CommittedSeal [][]byte                  //Committed seal, 65 * len(Validators) bytes
}
```

Thus the `extraData` would be in the form of `EXTRA_VANITY` | `ISTANBUL_EXTRA` where | represents a fixed index to separate

  - First `EXTRA_VANITY` bytes (fixed) may contain arbitrary proposer vanity data.
  - `ISTANBUL_EXTRA` bytes are the RLP encoded Istanbul extra data calculated from `RLP(IstanbulExtra)` , where `RLP()` is R
    - `Validators` : The list of validators, which **must** be sorted in ascending order.
    - `Seal` : The proposer's signature sealing of the header.
    - `CommittedSeal` : The list of commitment signature seals as consensus proof.

Block hash, proposer seal, and committed seals

The Istanbul block hash calculation is different from the `ethash` block hash calculation due to the following reasons:

1. The proposer needs to put proposer seal in `extraData` to prove the block is signed by the chosen proposer.
2. The validators need to put `2F + 1` of committed seals as consensus proof in `extraData` to prove the block has gone throug

The calculation is still similar to the `ethash` block hash calculation, with the exception that we need to deal with `extraData` . We ca

Proposer seal calculation

By the time of proposer seal calculation, the committed seals are still unknown, so we calculate the seal with those unknowns empt

- `Proposer seal` : `SignECDSA(Keccak256(RLP(Header)), PrivateKey)`
- `PrivateKey` : Proposer's private key.

- `Header` : Same as `ethash` header only with a different `extraData` .
- `extraData` : `vanity | RLP(IstanbulExtra)` , where in the `IstanbulExtra` , `CommittedSeal` and `Seal` are empty arrays.

Block hash calculation

While calculating block hash, we need to exclude committed seals since that data is dynamic between different validators. Therefore

- `Header` : Same as ethash header only with a different `extraData` .
- `extraData` : `vanity | RLP(IstanbulExtra)` , where in the `IstanbulExtra` , `CommittedSeal` is an empty array.

Consensus proof

Before inserting a block into the blockchain, each validator needs to collect `2F + 1` of committed seals from other validators to co
section.

Committed seal calculation:

Committed seal is calculated by each of the validator signing the hash along with `COMMIT_MSG_CODE` message code of its private ke

- `Committed seal` : `SignECDSA(Keccak256(CONCAT(Hash, COMMIT_MSG_CODE)), PrivateKey)` .
- `CONCAT(Hash, COMMIT_MSG_CODE)` : Concatenate block hash and `COMMIT_MSG_CODE` bytes.
- `PrivateKey` : Signing validator's private key.

Block locking mechanism

Locking mechanism is introduced to resolve safety issues. In general, when a proposer is locked at certain height `H` with a block `B`

Lock

A lock `Lock(B, H)` contains a block and its height, which means its belonging validator is currently locked at certain block `B` and

Lock and unlock

- Lock: A validator is locked when it receives `2F + 1` `PREPARE` messages on a block `B` at height `H` .
- Unlock: A validator is unlocked at height `H` and block `B` when it fails to insert block `B` to blockchain.

Protocol (`+2/3` validators are locked with `Lock(B,H)`)

- `PRE-PREPARE` :
  - **Proposer**:
    - Case 1, proposer is locked: Broadcasts `PRE-PREPARE` on `B` , and enters `PREPARED` state.
    - Case 2, proposer is not locked: Broadcasts `PRE-PREPARE` on block `B'` .
  - **Validator**:
    - Case 1, received `PRE-PREPARE` on existing block: Ignore.
      - Note: It will eventually lead to a round change, and the proposer will get the old block through synchronization.
    - Case 2, validator is locked:
      - Case 2.1, received `PRE-PREPARE` on `B` : Broadcasts `PREPARE` on `B` .
      - Case 2.2, received `PRE-PREPARE` on `B'` : Broadcasts `ROUND CHANGE` .
    - Case 3, validator is not locked:
      - Case 3.1, received `PRE-PREPARE` on `B` : Broadcasts `PREPARE` on `B` .
      - Case 3.2, received `PRE-PREPARE` on `B'` : Broadcasts `PREPARE` on `B'` .
        - Note: This consensus round will eventually get into round change since `+2/3` are locked at `B` and which w
- `PREPARE` :
  - Case 1, validator is locked:
    - Case 1.1, received `PREPARE` on `B` : Broadcasts `COMMIT` on `B` , and enters `PREPARED` state.
      - Note: This shouldn't happen though, it should have skipped this step and entered `PREPARED` in `PRE-PREPARE` st
    - Case 1.2, received `PREPARE` on `B'` : Ignore.
      - Note: There shouldn't be `+1/3` `PREPARE` on `B'` since `+2/3` are locked at `B` . Thus the consensus round on `B'`
  - Case 2, validator is not locked:
    - Case 2.1, received `PREPARE` on `B` : Waits for `2F + 1` `PREPARE` messages on `B` .
      - Note: Most likely it will receive `2F + 1` `COMMIT` messages prior to receiving `2F + 1` `PREPARE` messages since t
    - Case 2.2, received `PREPARE` on `B'` : Waits for `2F + 1` `PREPARE` message on `B'` .
      - Note: This consensus will eventually get into round change since `+2/3` validators are locked on `B` and which wo
- `COMMIT` :

    - Validator must be locked:
        - Case 1, received `COMMIT` on `B`: Waits for `2F + 1 COMMIT` messages.
        - Case 2, received `COMMIT` on `B'`: Shouldn't happen.

Locking cases

- Round change:
    - Case 1, `+2/3` are locked:
        - If proposer is locked, it'd propose `B`.
        - Else it'd propose `B'`, but which will lead to another round change.
        - Conclusion: eventually `B` will be committed by honest validators.
    - Case 2, `+1/3 ~ 2/3` are locked:
        - If proposer is locked, it'd propose `B`.
        - Else it'd propose `B'`. However, since `+1/3` are locked at `B`, no validators can ever receive `2F + 1 PREPARE` on `B'`
        - Conclusion: eventually `B` will be committed by honest validators.
    - Case 3, `-1/3` are locked:
        - If propose is locked, it'd propose `B`.
        - Else it'd propose `B'`. If `+2/3` reach consensus on `B'`, those locked `-1/3` will get `B'` through synchronization and
        - Conclusion: it can be `B` or other block `B'` be finally committed.
- Round change caused by insertion failure:
    - It will fall in one of the above round change cases.
        - If the block is actually bad (cannot be inserted to blockchain), eventually `+2/3` validators will unlock block `B` at `H` ar
        - If the block is good (can be inserted to blockchain), then it would still be one of the above round change cases.
- `-1/3` validators insert the block successfully, but others successfully trigger round change, meaning `+1/3` are still locked at `L`
    - Case 1, proposer has inserted `B`: Proposer will propose `B'` at `H'`, but `+1/3` are locked at `B`, so `B'` won't pass the con
    - Case 2, proposer hasn't inserted `B`:
        - Case 2.1, proposer is locked: Proposer proposes `B`.
        - Case 2.2, proposer is not locked: Proposer will propose `B'` at `H`. The rest is the same as above case 1.
- `+1/3` validators insert the block successfully, `-2/3` are trying to trigger round change at `H`.
    - Case 1, proposer has inserted `B`: Proposer will propose `B'` at `H'`, but won't pass the consensus until `+1/3` get `B` thro
    - Case 2, proposer has not inserted `B`:
        - Case 2.1, proposer is locked: Proposer proposes `B`.
        - Case 2.2, proposer is not locked: Proposer proposes `B'` at `H`. The rest is the same as above case 1.
- `+2/3` validators insert the block successfully, `-1/3` are trying to trigger round change at `H`.
    - Case 1, proposer has inserted `B`: proposer will propose `B'` at `H'`, which may lead to a successful consensus. Then those
    - Case 2, proposer has not inserted `B`:
        - Case 2.1, proposer is locked: Proposer proposes `B`.
        - Case 2.2, proposer is not locked: Proposer proposes `B'` at `H`. Since `+2/3` have `B` at `H` already, this round would c

Gossip network

Traditionally, validators need to be strongly connected in order to reach stable consensus results, which means all validators need t
any two validators are seen connected when either they are directly connected or they are connected with one or more validators i

How to run

Running Istanbul BFT validators and nodes is similar to running the official node in a private chain. First of all, you need to initialize

```
geth  --datadir "/eth" init "/eth/genesis.json"
```

Then,
for validators:

```
geth --datadir "/eth" --mine --minerthreads 1 --syncmode "full"
```

for regular nodes:

```
geth --datadir "/eth"
```

```
geth --datadir ./eth
```

Note on `syncmode`:
`--syncmode "full"` is required for the first set of validators to initialize a new network. Since we are using fetcher to insert blocks,

```
inserter := func(blocks types.Blocks) (int, error) {
                // If fast sync is running, deny importing weird blocks
                if atomic.LoadUint32(&manager.fastSync) == 1 {
                        log.Warn("Discarded bad propagated block", "number", blocks[0].Number(), "hash", blocks[
                        return 0, nil
                }
                atomic.StoreUint32(&manager.acceptTxs, 1) // Mark initial sync done on any fetcher import
                return manager.blockchain.InsertChain(blocks)
}
```

The sync mode affects only if there are some existing blocks, so there is no impact for initializing a new network.

For the later joined validators, we don't need to use full mode as they can get blocks by downloader. After the first sync from peers

Command line options

```
$geth help

ISTANBUL OPTIONS:
  --istanbul.requesttimeout value  Timeout for each Istanbul round in milliseconds (default: 10000)
  --istanbul.blockperiod value     Default minimum difference between two consecutive block's timestamps in secc
```

Nodekey and validator

To be a validator, a node needs to meet the following conditions:

- Its account (the address derived from its nodekey) needs to be listed in `extraData`'s validators section.
- Use its nodekey as its private key to sign consensus messages.

genesis.json

To run the Istanbul BFT chain, the `config` field is required, and the `pbft` subfield must present. Example as the following:

```
{
  "config": {
    "chainId": 2016,
    "istanbul": {
                "epoch": 30000,
                "policy" 0,
        }
  },
  "timestamp": "0x0",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "extraData": "0x0000000000000000000000000000000000000000000000000000000000000000f89af85494475cc98b5521ab2a1335
  "gasLimit": "0x47e7c4",
  "mixhash": "0x63746963616c2062797a616e74696e65206661756c7420746f6c6572616e6365",
  "coinbase": "0x3333333333333333333333333333333333333333",
  "nonce": "0x0",
  "difficulity": "0x0",
  "alloc": {}
}
```

`extraData` tools

We've create a set of `extraData` coding tools in istanbul-tools repository to help developers to manually generate `genesis.json`

Encoding:
Before encoding you need to define a toml file with `vanity` and `validators` fields to define proposer vanity and validator set. Ple

Command:

```
istanbul encode --config ./config.toml
```

Decoding:
Use `--extradata` option to give the `extraData` hex string. The output would show the following if presents: vanity, validator set

Use `--extradata` option to give the `extradata` hex string. The output would show the following if presents: vanity, validator set,

Command:

```
istanbul decode --extradata <EXTRA_DATA_HEX_STRING>
```

Ottoman testnet

We have setup a testnet for public testing. There are initially 4 validators and no designated faulty nodes. In the future, we want to

Run testnet node

```
geth --ottoman
```

Faulty node

We have implemented a simple faulty node that can make a validator run faulty behaviors during consensus. There are five behavi

- `NotBroadcast` : The validator doesn't broadcast any message.
- `SendWrongMsg` : The validator sends out messages with wrong message codes.
- `ModifySig` : The validator modifies the message signatures.
- `AlwaysPropose` : The validator always sends out proposals.
- `AlwaysRoundChange` : The validator always sends `ROUND CHANGE` while receiving messages.
- `BadBlock` : The validator proposes a block with bad body

Run following command to enable faulty node:

```
geth --istanbul.faultymode <MODE>
```

Where the `<MODE>` can be the following number:

- `0` : Disable faulty behaviors.
- `1` : Randomly run any faulty behaviors.
- `2` : `NotBroadcast` .
- `3` : `SendWrongMsg` .
- `4` : `ModifySig` .
- `5` : `AlwaysPropose` .
- `6` : `AlwaysRoundChange` .
- `7` : `BadBlock` .

Background

The idea of implementing a byzantine fault tolerance (BFT) consensus came from the challenges we faced while building blockchain

Banking systems tend to form a private chain or consortium chain to run their applications. PBFT is ideal for these settings. These e
private/consortium chains.

Remaining Tasks

- **Testnet**: Currently the *Ottoman* testnet only has 4 validators. We'd like to extend it to 22 validator nodes and setup some fa
- **Weighted round robin**: This will require a redesign of the `extraData` field, but should be fairly straightforward.
- **Remove or make block period configurable**: In certain setups, it may make sense to generate as many blocks as possible
- **Benchmarking and stress testing**:
  - Validator scalability.
  - Node scalability.
  - Transaction per second.
- **Smarter way to detect faulty proposer**: A proposer can always generate empty blocks or small blocks without being actir
- **Formal proof of safety and liveness**.

Notes and discussions

Does it still make sense to use gas?

Yes. We still need gas to prevent infinite loops and any kind of EVM exhaustion.

Does it make sense to charge gas in a consortium chain?

The network would be vulnerable if every account has unlimited gas or unlimited transaction sending power. However, to enable sc

Put consensus proof in the next block?

Currently our block header can be varied in `extraData` depending on its source validator because of the need to put consensus p
confirmation to reach finality (not instant finality).

Proof of lock

Inspired by *Tendermint*. We are still considering whether to add it to this EIP. Further efficiency benefits can be realized by reusing

Contribution

The work was initiated and open sourced by the Amis team. We're looking for developers around the world to contribute. Please fe

- @alanchchen
- @bailantaotao
- @markya0616
- @tailingchen
- @yutelin

Forked repository (and original implementation branch)

https://github.com/getamis/go-ethereum/tree/feature/pbft

Clarifications and feedback

TBD

bobsummerwill commented on 25 Jun 2017

Fantastic work, guys!

vbuterin commented on 26 Jun 2017 Collaborator

Great work!

> Block insertion fails

Can you explain when block insertion might fail? I'm struggling to see why block insertion would ever fail for a valid proposal.

> Return transaction fee to sender

Why not just accept zero-gasprice transactions?

> We have implemented a simple faulty node that can make a validator run faulty behaviors during consensus.

Have you tried running the network with >=1/3 faulty nodes? If so, what does the result look like; what kinds of failures do you see in practice?

vutelin commented on 26 Jun 2017 • edited

Thanks **@vbuterin**

> Block insertion fails

Before actually inserting the block into the chain, the consensus only validates the block header. Inserting will do more checks so it can fail with other reasons.

> Return transaction fee to sender

You're right. We've updated the EIP according.

> testing >=1/3 faulty nodes?

Yes.

- If there are more than 2/3 of faulty nodes, those faulty nodes can control the consensus. They can generate faulty blocks or keep running round change.
- If there are more than 1/3 and less than 2/3 of faulty nodes, it will keep running round change and no consensus can be reached.

vbuterin commented on 26 Jun 2017 Collaborator

> If there are more than 1/3 and less than 2/3 of faulty nodes, it will keep running round change and no consensus can be reached.

Theoretically it's also possible to finalize two conflicting blocks, if the proposer is one of the Byzantine nodes and makes two proposals and each get 2/3 prepares+commits. Though I guess that's fairly unlikely to happen in practice and so won't appear in that many random tests.

deanstef commented on 26 Jun 2017

> Each validator enters PRE-PREPARED upon receiving the PRE-PREPARE message with the following conditions:
> Block proposal is from the valid proposer.
> ***Block header is valid.***
> Block proposal's sequence and round match the validator's state.

I know the meaning of block validity, but outside the PoW this is a little bit ambiguous.
When a block is defined Valid or not without the proof-of-work?

ice09 commented on 26 Jun 2017

> sequence number should be greater than all pervious sequence numbers.

pervious -> previous

I like the structure, but for someone not accustomed to the terminology, `2F + 1` not defining up to section `constants` makes it more difficult to understand.

vutelin commented on 27 Jun 2017

@vbuterin

> Theoretically it's also possible to finalize two conflicting blocks, if the proposer is one of the Byzantine nodes and makes two proposals and each get 2/3 prepares+commits. Though I guess that's fairly unlikely to happen in practice and so won't appear in that many random tests.

Yes, I think you are right. Suppose there are f+1 faulty nodes, f+f good nodes, and the propose is among the faulty nodes. The proposer can send first f good nodes A block and second f good nodes B block. Then both

groups can receive 2f+1 of prepares+commits for block A and B respectively. Thus two conflicting blocks can be finalized.

@deanstef

> I know the meaning of block validity, but outside the PoW this is a little bit ambiguous. When a block is defined Valid or not without the proof-of-work?

Each validator puts `2F+1` committed seals into the `extraData` field in block header before inserting the block into the chain, which is seen as the consensus proof of the associated block. `extraData` also contains proposer seal for validators to verify the block source during consensus (same mechanism as in Clique).

@ice09
Thanks, we've updated this EIP accordingly.

deanstef commented on 27 Jun 2017

> Each validator puts 2F+1 committed seals into the extraData field in block header before inserting the block into the chain, which is seen as the consensus proof of the associated block. extraData also contains proposer seal for validators to verify the block source during consensus (same mechanism as in Clique).

Great! I was a little confuse through Valid block and Consensus Proof, your response is helpful also for the meaning of validation in Clique. Thank you.
Nice work guys !

ebuchman commented on 29 Jun 2017

> Round change timer expires.

Can you clarify when this timer starts? Is there one timer for the whole round, like in PBFT (well, in PBFT the timer starts once the client request is received), or is there a new timer at each phase (pre-prepared, prepared, etc.) as the figure seems to suggest?

Unless there is additional mechanism not described above (or perhaps I am just missing something), I think this protocol may have safety issues across round changes, as there does not seem to be anything stopping validators from committing a new block in a new round after others have committed in the previous round. This is what the "locking" mechanism in Tendermint addresses. In PBFT it's handled by broadcasting much more information during the round change. When you "blockchainify" PBFT, you can do away with this extra information if you're careful to introduce something like Tendermint's locking mechanism. I suspect that if you address these issues, you will end up with a protocol that is roughly identical (if not exactly identical) to Tendermint. Happy to discuss further and collaborate on this - great initiative!

vutelin commented on 29 Jun 2017

@ebuchman

> Can you clarify when this timer starts?

Yes, there is only one timer which is reset/triggered in every beginning of a new round.

> safety issues across round changes

Yes, in some extreme cases there might be safety issues. For example, say there is only one validator which receives `2F+1` commits but all the others do not. Then that validator would insert a valid block in to its chain while others would start a new round on the same block height. Eventually that might lead to conflict blocks.. We've put locking mechanism in the remaining tasks section. And yeah, we're looking forward to collaboration with Tendermint!

kumavis commented on 30 Jun 2017 Member

Sticky proposer seems like it would be able to submit empty blocks or censorship transactions if it never passed through the RoundChange state. As long as they submit valid blocks, they can hold their Proposer

role indefinitely.

kumavis commented on 30 Jun 2017 Member

> Blocks in Istanbul BFT protocol are final, which means that there are no forks and any valid block must be somewhere in the main chain.

Seems like a strong claim considering there is no penalty to being a faulty node (e.g. voting on multiple forks)

vutelin commented on 1 Jul 2017

@kumavis

> Faulty sticky proposer can keep generating empty valid blocks.

Yes, sticky proposer policy can lead to this issue. We've listed "faulty propose detection" in the remaining tasks section aiming to resolve it. One possible way is to switch to round robin policy whenever a validator sees an empty block. However, sticky proposer can still hack it by generating very small block every round.

> Block finality and penalty on faulty node.

Detecting faulty node deterministically is hard which makes penalize faulty nodes even harder. For simplicity, this PR doesn't dive into this topic. It might be worth looking in the follow up EIP and research.
Block finality is indeed a strong claim. In some rare case as **@ebuchman** pointed out, there might be safety issues. We listed it in remaining tasks section as well, and are looking to resolve it by introducing some kind of locking mechanism.

epoquehu commented on 4 Jul 2017 • edited

Awesome work! Can you give us a sense of performance benchmark in terms of throughput and latency? Thanks!

vutelin commented on 4 Jul 2017

@epoquehu

> Throughput and latency

In our preliminary testing result with 4 validators setup, the consensus time took around 10ms ~ 100ms, depending on how many transactions per block. In our testing, we allow each block to contain up to 2000 transactions.
Regarding throughput, the transaction per second (TPS) ranges from 400 ~ 1200; however, there are still too many Geth factors that significantly affect the result. We are trying to fix some of them and workaround some of them as well.
More comprehensive benchmarking and stress testing is still in progress. Stay tuned!

vutelin commented on 24 Jul 2017

Update: 68cbcf

- Add block locking mechanism.
- Performance/bug fixes.

vutelin referenced this issue in **ethereum/go-ethereum** on 28 Jul 2017

How many Consensus algorithm does ethereum involve? #14871 Closed

mawenpeng commented on 2 Aug 2017

Is there any way to keep the nodekey (account private key) secured? Seems like it's left there unencrypted.

vutelin commented on 8 Aug 2017

Update: 0f066fb

- Add gossip network

mikesmo commented on 10 Aug 2017

Great work on developing Istanbul!

One comment on "Does it still make sense to use gas?"

I've developed a testnet (using Ethermint) and modified the client to not charge gas. I wanted to bounce this idea of others to see whether this it is valid...

To avoid the infinite loop problem, the validators ensure the that smart contracts being published to the blockchain are sent from a small set of white-listed accounts.

These accounts are trusted by the consortium to only publish smart contracts that have gone through a strict review process.

I suppose in the extreme edge case that a computationally expensive slipped through and was published by mistake, then the validators stop and rollback to before the event.

Does this sound reasonable?

Appreciate any feedback on the faults with such an implementation.

Thanks.

 **yutelin** referenced this issue in **jpmorganchase/quorum** on 31 Aug 2017

consensus: add Istanbul BFT #166 Closed

 **tailingchen** referenced this issue in **getamis/eth-client** on 26 Oct 2017

add readme #5 Merged

 **marcossanlab** referenced this issue in **alastria/alastria-node** on 15 Dec 2017

Estudio de la utilización del protocolo de consenso Istanbul BFT #45 Closed

stevenroose commented on 24 Jan

The current implementation (as found in Quorum) breaks the concept of the "pending" block, used in several calls, but most notably in `eth_getTransactionCount` (`PendingNonceAt` in `ethclient`):

In Ethereum, the pending block means the latest confirmed block + all pending transactions the node is aware of. This means that directly after a transaction is sent to the node (through RPC), the transaction count (aka nonce) in the "pending" block is increased. A lot of tools, like abigen in this repo or any other tool where tx signing occurs at the application level instead of in geth, rely on this for making multiple transactions at once. After the first one, the result of `eth_getTransactionCount` will increase so that a valid second tx can be crafted.

With the current implementation of Istanbul, the definition of the "pending block" seem to be different. When submitting a transaction, the result for `eth_getTransactionCount` for the sender in the "pending" block does not change. When a new block is confirmed (not containing this tx), it does change however (while the value for "latest" doesn't). Then, on the next block confirmation, the "latest" also changes because the tx is in the confirmed block.

So this seems to mean that the "pending block" definition changed from "latest block + pending txs" to "the block that is currently being voted on". I consider this a bug: if this is done on purpose, it breaks with a lot of

block that is currently being voted on". I consider this a bug, if this is done on purpose, it breaks with a lot of existing applications (all users of abigen, f.e.) and should be reconsidered.

I originally reported about this issue in the Quorum repo, but there doesn't seem to be a good place to report bugs in Istanbul other than here.

This was referenced on 24 Jan

[WIP] consensus, core, eth, miner: use the same insert behavior between proposer and validators getamis/go-ethereum#123 Open

api: Fetch pending state from txpool ethereum/go-ethereum#15963 Open

patrickmn referenced this issue in **jpmorganchase/quorum** on 25 Jan

Why do all nodes run blockvoting strategy #31 Closed

Matthalp commented on 2 Feb

I'm sorry to disrupt the technical discussion here with a non-technical question: What is the intention for including this in the EIP repository? In particular I was wondering:

(1) Is this proposal seeking public protocol adoption (it seems private chain focused, really at extending `quorum` with the aims of also moving upstream to `geth`)?
(2) Does the scope of EIPs in this repository extend beyond public chain protocol improvements?

yutelin referenced this issue in **jpmorganchase/quorum** on 7 Feb

coinbase & accounts 0 are diiferent when trying to setup istanbul network #259 Closed

sifmelcara referenced this issue in **jpmorganchase/quorum** on 5 Mar

Istanbul BFT's design cannot successfully tolerate fail-stop failures #305 Open

kimmylin referenced this issue in **ethereum/go-ethereum** on 26 Mar

consensus, eth: Istanbul interface proposal #16385 Open

renuseabhava commented on 20 Apr

I have used set of extraData coding tools in istanbul-tools repository to manually generate genesis.json & defined toml file too, but when i starts nodes, it throws error as "Failed to decode message from payload", err="unauthorized address"

cvkonduru commented on 1 May

Fantastic work

This was referenced on 1 Jun

--mine required? jpmorganchase/quorum-examples#101 Closed

[Question] How to setup limited nodes as Raft consensus nodes? jpmorganchase/quorum#394 Closed

d4nd commented on 22 Jun

Thank you guys very much for this great contribuition.
I would like to know about the progress on it.

ineffectualproperty referenced this issue in **hyperledger/sawtooth-rfcs** on 28 Jun

PBFT consensus #19 Open

michaelkunzma... commented 8 days ago

@renuseabhaya I had the same issue. My problem was that with Istanbul, you do not use a "regular" account (meaning, an account that you generate using `geth account new`) to makes nodes validators. You need to use the node key and create an account from the node key.

@yutelin Can you explain what the rationale was behind using an account address, derived from the node key, to identify validators instead of using the regular enode ID that is already being used for identifying nodes?

vutelin commented 8 days ago

@michaelkunzmann-sap enode id is from node key.

michaelkunzma... commented 7 days ago • edited

@yutelin Yes, correct. So currently we are using

```
istanbul.propose("0x23971dab0b29c27fa0de9226c45bef04d9f39156", true)
```

Where `0x23971dab0b29c27fa0de9226c45bef04d9f39156` is the "address" of the node to be permitted. As far as I understand, this a we create with `geth account new`, since it is derived from the node key:

```
node_address = address(pub(node_key))
```

Since the enode id is also derived from the private node key (in its original purpose), is it possible to use the enode id instead of th generating an address from node key.

```
istanbul.propose("6f8a80d14311c39f35f516fa664deaaaa13e85b2f7493f37f6144d86991ec012937307647bd3b9a82abe2974e1407
```

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

Styling with Markdown is supported