mitum

Release proto2

Nov 08, 2019

Introduction

CHAPTER 1

Introduction

Mitum is a general privacy blockchain with flexible and resilient way. Mitum can be used for various kind of purposes, public and private blockchain like cryptocurrency network, data-centric blockchain for arbitrary data, or secure anonymity voting system, etc.

Basically mitum can provide these main features.

- **SECURITY**: All the in-coming and out-coming messages is signed by signature of sender, so there will be no chance some damaged or malicious messages to be infiltrated.
- **DESIGNING NETWORK**: mitum network is designed at bootstrap with various policies, network own data types and its native features. These designed factors can be updated without downtime or termination of node and the entire network.
- **APPLIANCE**: Data in mitum can be defined and designed. Any arbitrary type of data can be supported in mitum. Inside mitum there is a plugin system, so new type of data can be added thru plugin. If you want to launch cryptocurrency network, you can design currency model, define your own currency unit and even inflation rate, etc.
- **CONSENSUS**: mitum guarantees *finality*. Once the block and it's data are established, it will not be changed or revoked.
- **CONSENSUS**: mitum verifies and establishes data by the consensus protocol. We created the consensus protocol, called ISAAC+, which is newly devised and based on the manner of PBFT. ISAAC+ focuses on *finality* of block. It guarantees liveness, security and limited fault tolerance. ISAAC+ can be extended for the open and public environment, so new nodes can join the network without the external allowance.
- **CONSENSUS**: ISAAC+ works like well-hardened axe, it is hard to break and resilient from external impact. When some nodes are not intact, it tries to continue agreement. When some blocks are lost in nodes, these data are restored without breaking consensus, the missed consensus messages also be delivered to the edge of network.
- **CONSENSUS**: Due to ISAAC+, mitum can process huge amounts of messages, like currency transactions, consensus ballots, or any kind of messages for agreement.
- **DATA PRIVACY**: For privacy of user, mitum supports untraceable account. Basically thanks to the consensus process, accounts can be easily traced by anyone, who did fund it, who sent to it, whom it sent, and it's related data too. But unlike zcash, or hyperledge, mitum tries to support the privacy by transparent account. Transparent

account can be created by the legitimate account, but hides which source account makes it. It breaks the link from the originated source account.

- **DATA**: All the data is stored by hierarchical tree structure(AVL tree). It makes to store and search data efficiently.
- **DATA**: mitum does only rely on the data in the established data in block. The volatile data in node will not be used for consensus process and most of important data will be saved in block, so it can be updated by agreement by consensus protocol.
- **NETWORK**: the basic networking protocol is UDP for consensus process. By the nature of UDP, there is no need to keep or check the connection between consensus nodes.
- **DATA**: mitum supports various kind of storage database: LevelDB, MongoDB, MySQL, PostgreSQL, etc. By the purpose and scale of mitum network, you can choose the best storage database.
- **MANAGEMENT**: For handling the expected or unexpected situation, mitum will provide the management console to the node operator. By this management console, node operator will control his/her node manually.
- **NETWORK VOTING**: All consensus node has the special right to vote for the important decision of network, e.g. allowance or exile of newbie consensus node, updating network policies, etc.

This document will introduce mitum and will describe the working mechanism of mitum.

CHAPTER 2

How mitum Works

2.1 TL;DR

As described in the section, “*Introduction*”, the mitum network consists of the multiple consensus nodes.

See also:

Standalone mode For development or research purpose, you can compose the network with only one node. In standalone mode, every operation will be same, even with the consensus process.

As mitum is blockchain, basically mitum network tries to store the incoming data by trusted way. This is simple process for new data.

1. New message is received by one of nodes in the network.
2. New message contains the user data.
3. Message and it’s data are validated by the nodes.
4. Each node tries to get the agreement for the new message and it’s data.
5. If nodes get agreement to store new data, the new data will be established in the next block.

The important things in the process are,

- The agreement will be done by the consensus protocol, ISAAC+, and the agreement is made by voting with consensus nodes
- All the incoming message is validated by consensus nodes.
- Only agreed data is established(stored) in the block.

This is the normal process of PBFT based blockchain. Mitum follows the classic scenario of PBFT.

2.2 Uncompressed Version

For detailed explanation, we can assume the simple situation,

- 10 nodes: `m0`, `m1`, `m2`, `m3`, `m4`, `m5`, `m6`, `m7`, `m8`, `m9`
- suffrage group members: all node
- number of acting suffrage group members: 4 nodes
- Each node can reach others.
- Each node does not share the storage and network with others.
- Nodes, `m0`, . . . , `m8` are already working and `m9` is just booted.

This example situation will be applied throughout this document.

See also:

4 nodes is the minimum number of consensus nodes. The detailed mitum network will be described in the section, "*Designing Network*".

See also:

The detailed information about the bootstrapping mitum, will be described in the section, ""

See also:

About the consensus process, the section, "*Consensus Protocol, ISAAC+*".

2.2.1 `m9` is booted

- After `m9` are booted, each node will check it's current block state and environment to join the network. At this time, node also tries to check the global network consensus state, which block height and round are proceeded currently.
- When everything is OK for joining consensus, `m9` joins consensus.
- The next consensus voting is for the next block, which has the height, `H33` and it's round is `R0`.

2.2.2 New data message received

- `m9` got the new data message, `B1`. It has the data, `D1`.
- `m9` tries to broadcast `B1` to the other consensus nodes.
- The network selects `m1` as the new proposer for the next block(`H33` and `R0`)
- `m1` will propose the new proposal, `P1` with `B1`.
- All the consensus nodes tries to establish `P1` for the next block.
- To establish `m1`'s `P1`, the majority should be reached for 2 steps.
- When each node receives the new proposal, `P1` from the legitimated proposer, it estimates `P1` and vote on `P1`.
- After `P1` is passed thru *SIGN* and *ACCEPT* voting stage, `P1` will be established in the next block(`H33`, `R0`)

CHAPTER 3

Consensus Protocol, ISAAC+

ISAAC+ is the consensus protocol, based on classic [PBFT] . ISAAC+ shares also the limitations of PBFT, but ISAAC+ will fill the hole by the engineering and achieves more finality with the goods of PBFT in decentralized way.

CHAPTER 4

ISAAC+ Mechanism

As described at the previous section, the consensus voting is processed thru voting stages, *INIT*, *SIGN* and *ACCEPT*. The node, which can participate in consensus process, set its state to *Consensus* state. If node is under other state, it means node is not ready for participating consensus process.

4.1 Voting and Round

In ISAAC+, node votes to make agreement with the others. How node can vote? The voting process is simple, just broadcasts Ballot to the entire network and gathers ballots from others.

Every node knows which nodes are in suffrage group and which nodes are in acting suffrage group at this stage and round. The member information does not need to be shared or synced, because it is managed and established at block like other data of block.

Voting occurs by round. For example, the latest block is `H33`,

1. *INIT* : `H34`, `R0`
2. *SIGN* : `H34`, `R0`
    - Failed to get agreement: *INIT* : `H34`, `R1`
3. *SIGN* : `H34`, `R1`
4. *ACCEPT* : `H34`, `R1`
5. New *INIT* : `H35`, `R0`

Like the example, round is unique within new block voting.

CHAPTER 5

ISAAC+: Compared with classic PBFT

Basically the well-known principles of PBFT are,

- Consensus tries to reach the majority by voting.
- Each view has a leader and it proposes the next block.
- Each node validates and votes the next block of leader.

ISAAC+ is also based these rules and has some additional rules,

- Each round has a proposer(leader) and it proposes proposal, which is the contents of next block.
- Each acting suffrage group member validates the proposal and votes the established result of proposal.
  - If acting round is failed to get agreement, all the suffrage group members move to next round.
- To start new round, all the nodes in acting suffrage group must agree at the next *INIT* stage.
  - If network is failed to agree at *INIT* stage, suffrage group members try to get agreement of *INIT* stage.

Like PBFT, the recommended agreement threshold within acting suffrage group is at least 67% and to keep consensus going in the network level, the recommended number of nodes should be greater than 4. 4 nodes means the consensus will be going if 1 node fails at most. To make network to be more sustainable against the failed nodes, the number of nodes should be increased by the rule, `3F+1` (`F` is the number of failed node)

5.1 Phases and Stages

Traditionally PBFT has 4 phases, each phase represents how the incoming request from client reaches to the consensus.

1. *Request*
2. *Pre-Prepare*
3. *Prepare*
4. *Commit*

ISAAC+ has also similar 3 phases(in ISAAC+, it is called voting stage),

1. *INIT*
2. *SIGN*
3. *ACCPET*

Each stage does have similar meaning.

INIT

- All the node of suffrage group members participate.
- At this stage, suffrage group members will get agreement on the new block and it's round, which is voted at the previous *ACCEPT* stage.
- When the agreement is reached, the next block will be established and start new round for next block

SIGN

- Only members of acting suffrage group participate.
- Similar to the *Pre-Prepare*, the selected proposer(leader in PBFT) broadcasts its proposal to the network
- Similar to the *Prepare*, each node will validate the proposal

ACCEPT

- Only members of acting suffrage group participate.
- Similar to the *Prepare*, when *ACCEPT* stage gets agreement, the new block is ready to established and ready to move next block.

Unlike the classic PBFT, ISAAC+ will establish the new block at the *INIT* stage by all the member of suffrage group, this means all the members of suffrage group should agree the new block for establishing it.

CHAPTER 6

Node State

Node in mitum network has 5 different state.

- *Booting*
- *Syncing*
- *Joining*
- *Consensus*
- *Stopped*

The basic life cycle of node state is:

```
Booting -> Syncing -> Joining -> Consensus -> Stopped
```

Node decides it's condition and transit it's state. For example:

- Node finds it's block state is different from majority:

```
Consensus -> Syncing ... -> Joining -> Consensus
```

- Node is not in suffrage group:

```
Booting -> Syncing ...
```

- During *Joining* state, node finds it's block state is different from majority:

```
Joining -> Syncing ... -> Joining -> Consensus
```

6.1 States

6.1.1 Booting

- Node is just deployed and prepare its resources to join the network.

6.1.2 Syncing

- The newly started node tries to sync its block state to the latest of network.
- If node can not participate consensus, that means, it is not in suffrage group, it will stay the *Syncing* state for syncing the latest block state.

6.1.3 Joining

- After node finishes *Syncing*, node tries to join consensus process.
- Node checks the current acting block height and round.
- If acting block height and round is acceptable to node, move to *Consensus* state.

6.1.4 Consensus

- At this state, node can participate consensus process.

6.1.5 Stopped

- By any reason when node is stopping or stopped.

CHAPTER 7

Voting Stage

The voting stages was already explained briefly. At this section, more details will be described.

In mitum, there are 3 voting stages, *INIT*, *SIGN* and *ACCEPT*. These have very similar role with classic PBFT. The voting flows thru stages:

```
INIT -> SIGN -> ACCEPT
```

Strictly to say, there is one more important step between *INIT* and *SIGN*, that is *Proposal*.

```
INIT -> Proposal -> SIGN -> ACCEPT
```

Except proposal, voting occurs and then consensus process moves to next stage when the voting result reaches to majority(over voting *threshold*).

7.1 INIT: Stage For Suffrage Group

The first step of consensus starts at *INIT* voting. Unlike the other stages all the suffrage group members(`m0`, . . . , `m9`) votes at this stage. *INIT* stage has important features:

- Finally establishing the next block of the previously proposed proposal.
- Making agreement for next block
- Making agreement for new round

The ballot for *INIT* stage has information:

- New block
- Round of new block(previous voting)
- Next round

INIT voting is finished, this means, network gets agreement on the next block and new round. Node can select acting suffrage group members for next block and new round.

When *INIT* voting is finished, the new proposer of newly selected acting suffrage group should propose new proposal for next block to the entire network.

Note:

Q. Node is booted and previous block is already established. Does *INIT* ballot should contain new block and previous round information?

A. TL;DR Yes. Basically *INIT* stage works for verifying the previous voting result and finally establishing it. If previous voting is already established, the new block and it's round information will be used the new starting round is based on the valid block or not.

7.2 Proposal

Proposal is proposed by proposer, which is selected node from acting suffrage group. Proposal has this information:

- Next block
- Next round
- Messages for next block

Block is the result of validating proposal, so suffrage group members will validate proposal.

If proposal is not received:

- Node in the suffrage group waits the proposal from the proposer for the given time.
- Node does not receive the expected proposal.
- Node will start new round.
- In the new round, the another acting suffrage group and another proposer will be selected.

If proposal is received, but invalid:

- Node in the suffrage group receives the expected proposal.
- But it has some problems:
  - Proposal message is not well-formatted, or
  - The information of proposal is not correct, or
  - Messages in the proposal is not valid, etc.
- Node will start new round.

Non-intact proposer will be evaluated also as faulty node by the suffrage group.

7.3 Stages For Acting Suffrage Group

SIGN and *ACCEPT* stage is done by acting suffrage group, not by suffrage group.

Even if acting suffrage group fails to get agreement at *SIGN* or *ACCEPT*, the consensus process keeps going. The agreement of suffrage group decides to the next block, not by one of acting suffrage group.

For example,

- Acting suffrage group members: `m0`, `m1`, `m2`, `m3`

- Proposer: `m0`
- Threshold for majority: 3 of 4
- Proposal: `P0`

Voting for the round, `R0`

node	stage	block
`m0`	*SIGN*	`H0`
`m1`	*SIGN*	`H0`
`m2`	*SIGN*	`H1`
`m3`	*SIGN*	`H1`

This *SIGN* voting fails to get agreement, * `H0`: `m0`, `m1`; under threshold 3 * `H1`: `m2`, `m3`; also under threshold 3

No hash failed to be over threshold(3). With this voting result, they move to the next stage, *ACCEPT*:

node	stage	block
`m0`	*ACCEPT*	`H0`
`m1`	*ACCEPT*	`H0`
`m2`	*ACCEPT*	`H1`
`m3`	*ACCEPT*	`H1`

Even if agreement failed, why acting suffrage members moves to *ACCEPT*? In ISAAC+, the agreement failure inside acting suffrage group does not mean the agreement failure of entire network. As described earlier, acting suffrage group exists for proposing proposal and verifying the ability and health of node continuously. The voting of acting suffrage group will be evaluated by all the suffrage group members and if some node be thought as faulty node, it will be handled by network policy and rule.

After *ACCEPT* stage is finished, the next *INIT* stage will be like this;

node	stage	block
`m0`	*INIT*	`H0`
`m1`	*INIT*	`H0`
`m2`	*INIT*	`H1` *
`m3`	*INIT*	`H1` *
`m4`	*INIT*	`H0`
`m5`	*INIT*	`H0`
`m6`	*INIT*	`H0`
`m7`	*INIT*	`H0`
`m8`	*INIT*	`H0`
`m9`	*INIT*	`H1` *

The result of voting:

- `H0`: `m0`, `m1`, `m4`, `m5`, `m6`, `m7`, `m8`
- `H1`: `m2`, `m3`, `m9`

`H0` gets votes over threshold, 7 in the suffrage group

Note:

Q. The threshold, 7 is different from 3, threshold of acting suffrage group, why?

A. The default threshold percent is 67%, this means at least 2/3 nodes should agree on the same result. The 7 is 67% of the number of all the suffrage group members.

The suffrage group agreed on `H0` and `H0` will be established as the new block, and then newly selected acting suffrage group will start new round for next block.

7.3.1 SIGN

After agreement of *INIT* stage, consensus process moves to *SIGN* stage. The voting at this stage is on the proposal for the next block. Basically proposal has the contents of the next block, so node checks and validates the content of proposal. Each node can produce the next block from proposal and vote by the produced next block.

The ballot for *SIGN* stage has this information:

- The latest block
- Round
- Proposal
- Next block

When same next blocks from *SIGN* ballots reaches majority(over threshold), the consensus process moves to *ACCEPT* stage.

7.3.2 ACCEPT

ACCEPT stage is the final stage of acting suffrage group. The consensus process will work if *INIT* stage be started after *SIGN* without *ACCEPT* stage. This stage maybe looks redundant, but there are some reasons:

- During 2 stage, *SIGN* and *ACCEPT* by the acting suffrage group, the suffrage group will have enough time to share result rather than with only *SIGN* stage.
- The minority node at *SIGN* stage can have chance to correct its decision. With node maybe estimated as none-intact node by only *SIGN* voting.

The ballot for *ACCEPT* stage has this information:

- The latest block
- Round
- Proposal
- Next block

CHAPTER 8

ISAAC+: Weakness and Limitations

Too obviously ISAAC+ is not perfect as consensus protocol. At this time, ISAAC+ is the practical solution over PBFT. At this section the weaknesses and limitations of ISAAC+ will be introduced.

8.1 One Node Performance Entire Network Performance

The performance at this point is the ability how much contents can be handled at a time. With ISAAC+, the performance of entire network is almost same with the performance of one node. That is to say, one node can decide the maximum amount of contents for one block. At the worst case, the performance of entire network set to the one of the poorest node.

This problem is not native to ISAAC+, most of the implementation of PBFT suffers same problem.

To solve this problem, ISAAC+ has the suffrage group. Network designer designs its network at the first time, the designer will set the several policies for new member for suffrage group like:

- Low network latency
- Enough computing power
- etc.

The new node, who want to join the suffrage group, should satisfy these condition.

We can assume that node was decent, but after joining group, it became weird node, too long to propose new proposal and too long to vote. At this case, the designer also set the rule for exiling faulty node.

8.2 Network Latency is Important

For ISAAC+ working correctly, it requires fast network speed for node. It needs to broadcast voting ballot message to the others fast, because to complete one voting round within the given time.

Network designer can decide how much time node will wait for the expected ballot from others. The more node has enough duration, the network coverage will be wider. With short duration, network will require very low network latency.

8.3 Too many ballot messages between suffrage group

ISAAC+ works by voting and voting is done by ballot messages. All the members of suffrage group should broadcast its own ballot to the others. The more node participate network, the number of messages will be increased exponentially.

Usually the size of ballot message is very tiny, but huge messages always will be a big burden in most systems. To reduce the problem, mitum basically rely on the UDP network layer and binary messaging serialization.

CHAPTER 9

Contest: ISAAC+ Consensus Simulator

There are huge amounts of documentations for various kind of consensus protocol and also there are lots of implementations for helping to understand its mechanism. Contest is the implementation for simulating the detailed mechanism of ISAAC+.

Contest provides these features:

- All the settings for network can be set by simple configuration file, YAML.
- The size of network can be set.
- New features can be easily added.
- SQL-like expression can be used for checking node state and network.

9.1 Introduction

With configuration file, you can build your own test mitum network with the number of consensus nodes and can run it. The network of contest will produce the log messages and with it, you can watch how ISAAC+ and mitum works.

For example, this is simple contest configuration file:

Listing 1: sample.yml

```yaml
condition:
    all:
        node_state:
            - current_state="booting" AND new_state="joining"

        new_block:
            - m="new block created" AND block.height>=12
```

This config will check the 2 conditions on *all* nodes from json **log messages** and if matched log found, the output will be printed. The first condition is,

```
current_state="booting" AND new_state="joining"
```

This expression is from SQL-like, especially it is similar with a part of *WHERE* expression of SQL. This expression will check if the state of node changes from `booting` to `joining`.

```
m="new block created" AND block.height>=12
```

This expression will check if the message of log(`m`) is `new block created` and it's block height(`block.height`) is over `12`.

With this config file, contest can be run like this:

```
$ ./contest run sample.yml \
    --log ./contest-sample \
    --number-of-nodes 4 \
    --exit-after 10s
```

The output will be:

```
...

query: (and:(node = [n2]), (m = [new block created]), (block.height >= [12]), (block.
↪round = [0]))
matched log:
{
  "level": "info",
  "node": "n2",
  "m": "new block created"
}
================================================================================
query: (and:(node = [n3]), (m = [new block created]), (block.height >= [12]), (block.
↪round = [0]))
matched log:
{
  "level": "info",
  "node": "n3",
  "m": "new block created"
}
================================================================================
query: (and:(node = [n1]), (m = [new block created]), (block.height >= [12]), (block.
↪round = [0]))
matched log:
{
  "level": "info",
  "node": "n1",
  "m": "new block created"
}
================================================================================
query: (and:(node = [n4]), (m = [new block created]), (block.height >= [12]), (block.
↪round = [0]))
matched log:
{
  "level": "info",
  "node": "n4",
  "m": "new block created"
}
================================================================================
```

(continues on next page)

(continued from previous page)

```
query: (and:(node = [n2]), (current_state = [booting]), (new_state = [joining]))
matched log:
{
  "level": "info",
  "node": "n2",
  "current_state": "booting",
  "new_state": "joining",
  "m": "state changed"
}
================================================================================
query: (and:(node = [n3]), (current_state = [booting]), (new_state = [joining]))
matched log:
{
  "level": "info",
  "node": "n3",
  "current_state": "booting",
  "new_state": "joining",
  "m": "state changed"
}
================================================================================
query: (and:(node = [n4]), (current_state = [booting]), (new_state = [joining]))
matched log:
{
  "level": "info",
  "node": "n4",
  "current_state": "booting",
  "new_state": "joining",
  "m": "state changed"
}
================================================================================
query: (and:(node = [n1]), (current_state = [booting]), (new_state = [joining]))
matched log:
{
  "level": "info",
  "node": "n1",
  "current_state": "booting",
  "new_state": "joining",
  "m": "state changed"
}
================================================================================

...

exit 0
```

The output of command will produce the result of checking conditions with the matched log messages.

9.1.1 Installation

The detailed instruction about installation is at Contest project page.

9.2 Condition

condition field can be defined in the top-level of The config file. The sub fields can be defined with condition expressions.

Listing 2: sample.yml

```yaml
global:
  policy:
      threshold: 67
      interval_broadcast_init_ballot_in_join: 5s
      timeout_wait_vote_result_in_join: 6s
      timeout_wait_ballot: 6s

condition:
    all:
        node_state:
            - current_state = "booting" AND new_state = "joining"

        new_block:
            - m LIKE "new block created" AND block.height >= 12

    proposer:
        n1:
            - m LIKE "propose new proposal" AND vr.height = 11 AND vr.round = 0 AND␣
↪vr.stage = "INIT" AND vr.agreement = "MAJORITY"
```

Basically the section of condition field has these structure:

```yaml
condition:
    section_name:
        name #0:
            - experssion #0
            - experssion #1

        name #1:
            - experssion #2
            - experssion #3
```

all section is predefined section, the conditions in all section, will be applied to all the nodes. For example, the conditions under node_state should be matched to all nodes. new_block also too.

Note: For debugging or testing condition expressions, contest command has query sub-command.

```
$ contest query -h
query logs

Usage:
  contest query <log> [flags]

Flags:
  -h, --help                help for query
      --pretty              pretty json output
      --query stringArray   query

$ contest query /tmp/contest.log \
```

(continues on next page)

(continued from previous page)

```
    --query 'current_state = "booting" AND new_state = "joining"'
...
{
  "level": "info",
  "node": "n1",
  "module": "state-controller",
  "current_state": "booting",
  "new_state": "join",
  "t": "2019-09-30T23:00:09.107501+09:00",
  "caller": "/Users/spikeekips/workspace/mitum/src/isaac/state_controller.go:93",
  "m": "state changed"
}

$ echo $?
0
```

Note: If `contest query` fails to find the matched condition, exit code will be `1`.

9.2.1 Condition Matching

Condition expression works like SQL *WHERE* clause, almost same. Like SQL, expression can be defined by the SQL rule.

```
<column name> <comparison or operators> <value>
```

In contest, `<column name>` of the condition expression is the nested field name of one json log message. For example, to check the highlighted parts,

```
{
  "level": "debug",
  "node": "n4",
  "module": "state-controller",
  "seal": {
    "type": "ballot",
    "hash": "sl:2qQNQGcsquu731Z13NtSA1Qtcovgso7atzXYRi6vuxVB",
    "header": {
      "signer":
↪"GCX3QWQFFSOQFBX3TWYHVB62VX7GKRGEN6GTLI3SNVU7OMRSOKCEE3LW:public:stellar",
      "signature":
↪"29cZuExcnZMCWL2xdUhLUbLMAneXcQ5jcCQ6J5YuAuYjqKaQZmEbC5daRPSxLsYsrzdiY2nYadcz2D1LRqk4xKJ4
↪",
      "bodyHash": "ballot:DgMWNQew8tr4XbQw2n4dYy1j9jMGZphvWijhbVEe2yfC",
      "signedAt": "2019-09-30T17:05:58.423751+09:00"
    },
    "body": {
      "hash": "ballot:DgMWNQew8tr4XbQw2n4dYy1j9jMGZphvWijhbVEe2yfC",
      "node": "na:EYdsb4wfdNnup25RL97LBC5HMf8d56C79fTj3R8iKU4C",
      "stage": "INIT",
      "height": "11",
      "round": 0,
      "proposal": "pp:C6Z3RcavkBCWLa5yw6vsMYDugyYqRiL9FJ6JSpkDwLf6",
      "block": "bk:8w2xSGEKqvKL51ne6Wk4Wum8b6UzQustLgzkcAhXHxxE",
```

(continues on next page)

(continued from previous page)

```
      "last_block": "bk:52FK4q8CmpYutvbWmQr4Q7HuY7yrSJZerPG5neE6fDqi",
      "last_round": 11
    }
  },
  "t": "2019-09-30T17:05:58.428867+09:00",
  "caller": "/Users/spikeekips/workspace/mitum/src/isaac/state_controller.go:150",
  "m": "seal received; ballot"
}
```

The condition will be,

```
level = "debug" AND "body.height = "11"
```

The interesting expression is `body.height`. The sub field can be defined as `.` connected fields.

In contest, these operators is supported:

- `=`
- `<`
- `>`
- `<=`
- `>=`
- `!=`
- `in`
- `not in`
- `like`
- `not like`
- `regexp`
- `not regexp`

See also:

The detailed usage of each operator can be found at Where (SQL) .

9.3 Cases

Contest is written for simulating and testing ISAAC+ consensus protocol, to find the missing points and wrong concepts. In this section, the various kind of situations and cases will be tested by contest.

9.3.1 Custom Policy

Listing 3: custom-policy.yml

```
global:
  policy:
    threshold: 67
    interval_broadcast_init_ballot_in_join: 5s
    timeout_wait_vote_result_in_join: 6s
    timeout_wait_ballot: 6s

conditions:
  all:
     # base state
```

(continues on next page)

(continued from previous page)

```
      # {
      #   "level": "info",
      #   "node": "n0",
      #   "current_state": "booting",
      #   "new_state": "joining",
      #   "m": "state changed"
      # }
      - current_state="booting" AND new_state="joining"
      # {
      #   "level": "info",
      #   "node": "n0",
      #   "current_state": "joining",
      #   "new_state": "consensus",
      #   "m": "state changed"
      # }
      - current_state="joining" AND new_state="consensus"

      # new block created
      - m="new block created" and block.height="12" and block.round=0
```

In mitum, there are several factors for policy, these factors can control how mitum and consensus works.:

Note: The default value of each factor will be found at defaultPolicyConfig.

threshold By default, `threshold` is `67` percent. This means how many nodes should agree on voting stage. `67` percent needs 2/3 of all nodes. If it is `100`, nodes agree unanimously.

interval_broadcast_init_ballot_in_join This factor can control how often node will send *INIT* ballot in *join* state. If `3s`, node will send *INIT* ballot every 3 seconds.

timeout_wait_vote_result_in_join Node is in *join* state and waits *INIT* ballots from others, but fails to get enough ballots within `timeout_wait_vote_result_in_join` time, node will analyze the exact situation of network.

timeout_wait_ballot In consensus state, node will wait ballots for `timeout_wait_ballot` and if fails to get enough ballots within `timeout_wait_ballot`, node will move to next round.

timeout_wait_init_ballot In consensus state, node will wait *INIT* ballots for `timeout_wait_init_ballot` and if fails to get enough ballots within `timeout_wait_init_ballot`, node will change it's state to *Joining*, it means consensus process will be stopped and tries to check the health of network.

Note: You can find all the policy factors at PolicyConfig in source.

Note: By default, contest will set the latest *block height* to `11` with *round* `0`.

Run contest:

```
$ ./contest run custom-policy.yml \
    --log ./contest-sample \
    --number-of-nodes 4 \
    --exit-after 10s
```

If all the condition are matched, contest will exit with exit code, `0` with the matched logs.

9.3.2 Voting Failure

Failure Nodes Over Blocking Number At INIT Stage

Under situation

- Suffrage group members should vote for INIT stage.
- But some nodes does not offer the INIT ballot,
- The number of these nodes is over *blocking number*.
- Timed out in a given time, each node fails to get enough ballots for INIT stage.

Expected actions

- Each node stops the consensus process and changes its state to *Joining*.
- **:strike:‘Each node will request *VoteProof* to the others.‘**

Listing 4: failure-voting-init-over-blocking-number.yml

```yaml
nodes:
  n2:
    modules:
      ballot_maker:
        name: ConditionBallotMaker
        conditions:
          # {
          #   "ballot": {
          #     "current_proposal": "sl:7e8sQiEQReVtoe9HcQBfAgNr4V1ZDgx18jpEstuAiA6f",
          #     "current_round": 0,
          #     "last_block": "bk:2Xii7H6ykkD58euEHe8DEhAZJPxT3owUEj7EHY9e5HGH",
          #     "last_round": 0,
          #     "next_block": "bk:8pyPKQdX78sAe83zq8EoR6NgvfCqRqKNHduBG3xZLHNJ",
          #     "next_height": 13,
          #     "stage": "INIT"
          #   },
          #   "block": {
          #     "height": 11,
          #     "proposal": "sl:8QiEL44ptpYWVgRUxRr9D3KiBYknetonqeJCjHAPD8js",
          #     "round": 0
          #   },
          #   "node": "n3",
          #   "previousBlock": {
          #     "height": 10,
          #     "proposal": "pp:JC5VCGQWagkpSCAXxMa8737LCjkh1gQNkL91jGPmwCjo",
          #     "round": 11
          #   },
          #   "state": "consensus"
          # }
          - condition: ballot.next_height="13" AND ballot.stage in ("INIT")
            actions:
              - action: empty-ballot
  n3:
    modules:
      ballot_maker:
```

(continues on next page)

(continued from previous page)

```
        name: ConditionBallotMaker
        conditions:
          - condition: ballot.next_height="13" AND ballot.stage in ("INIT")
            actions:
              - action: empty-ballot

conditions:
  all:
    # base state
    # {
    #   "level": "info",
    #   "current_state": "booting",
    #   "new_state": "joining",
    #   "m": "state changed"
    # }
    - current_state="booting" AND new_state="joining"

    # {
    #   "level": "info",
    #   "current_state": "joining",
    #   "new_state": "consensus",
    #   "m": "state changed"
    # }
    - current_state="joining" AND new_state="consensus"

    # got ACCEPT majority of block, 12 and roundm, 0
    # {
    #   "level": "debug",
    #   "height": "12",
    #   "round": 0,
    #   "total": 4,
    #   "threshold": 3,
    #   "stage": "ACCEPT",
    #   "set": [
    #     3
    #   ],
    #   "is_finished": true,
    #   "m": "check majority"
    # }
    - m="check majority" AND height="12" AND round=0 AND stage="ACCEPT" AND is_
↪finished=true

    # after timeed out, all nodes moves to joining state
    # {
    #   "level": "info",
    #   "current_state": "consensus",
    #   "new_state": "joining",
    #   "m": "state changed"
    # }
    - current_state="consensus" AND new_state="joining"
```

```
$ ./contest run failure-voting-init-over-blocking-number.yml \
    --log ./contest-failure-voting-init-over-blocking-number
$ echo $?
0
```

This is the filtered majority checking messages:

```
$ cat ./contest-failure-voting-init-over-blocking-number/n0.log | \
     grep -i 'check majority' | \
     jq -c '[.height, .round, .stage, .total, .threshold, .is_finished, .m]' | \
     column -s ',' -t
["11"  0  "INIT"    4  3  false  1     "check majority"]
["11"  0  "INIT"    4  3  false  2     "check majority"]
["11"  0  "INIT"    4  3  true   3     "check majority"]
["11"  0  "SIGN"    4  3  false  1     "check majority"]
["11"  0  "SIGN"    4  3  false  2     "check majority"]
["11"  0  "SIGN"    4  3  true   3     "check majority"]
["11"  0  "SIGN"    4  3  true   null  "check majority    but closed"]
["11"  0  "ACCEPT"  4  3  false  1     "check majority"]
["11"  0  "ACCEPT"  4  3  false  2     "check majority"]
["11"  0  "ACCEPT"  4  3  true   3     "check majority"]
["11"  0  "ACCEPT"  4  3  true   null  "check majority    but closed"]
["12"  0  "INIT"    4  3  false  1     "check majority"]
["12"  0  "INIT"    4  3  false  2     "check majority"]
["12"  0  "INIT"    4  3  true   3     "check majority"]
["12"  0  "SIGN"    4  3  false  1     "check majority"]
["12"  0  "SIGN"    4  3  false  2     "check majority"]
["12"  0  "SIGN"    4  3  true   3     "check majority"]
["12"  0  "SIGN"    4  3  true   null  "check majority    but closed"]
["12"  0  "ACCEPT"  4  3  false  1     "check majority"]
["12"  0  "ACCEPT"  4  3  false  2     "check majority"]
["12"  0  "ACCEPT"  4  3  true   3     "check majority"]
["12"  0  "ACCEPT"  4  3  true   null  "check majority    but closed"]
["13"  0  "INIT"    4  3  false  1     "check majority"]
["13"  0  "INIT"    4  3  false  2     "check majority"]
```

This shows the node, `n0` checks majority on the incoming ballots. As we expected, the voting was done from the block, `11` to `13`.

Failure Nodes Under Blocking Number At INIT Stage

Under situation

- Suffrage group members should vote for INIT stage.
- But some nodes does not offer the INIT ballot,
- The number of these nodes is **under** *blocking number*.

Expected actions

- Consensus does not stop.

Listing 5: failure-voting-init-under-blocking-number.yml

```yaml
nodes:
  n3:
    modules:
      ballot_maker:
        name: ConditionBallotMaker
        conditions:
          # will not make INIT ballot when height, 13
          - condition: ballot.next_height="13" AND ballot.stage in ("INIT")
            actions:
              - action: empty-ballot
```

(continues on next page)

(continued from previous page)

```

conditions:
  all:
    # base state
    - current_state="booting" AND new_state="joining"
    - current_state="joining" AND new_state="consensus"

    # got ACCEPT majority for height, 12 and round, 0
    - m="check majority" AND height="12" AND round=0 AND stage="ACCEPT" AND is_
↪finished=true

    # got INIT majority for height, 13 and round, 0
    - m="check majority" AND height="13" AND round=0 AND stage="INIT" AND is_
↪finished=true

    # new block created
    - m="new block created" AND block.height="13" AND block.round=0
```

```
 $ ./contest run failure-voting-init-under-blocking-number.yml \
     --log ./contest-failure-voting-init-under-blocking-number
 $ echo $?
 0
```

This is the filtered majority checking messages:

```
$ cat ./contest-failure-voting-init-under-blocking-number/n0.log | \
     grep -i 'check majority' | \
     jq -c '[.height, .round, .stage, .total, .threshold, .is_finished, .m]' | \
     column -s ',' -t
["11"  0  "INIT"    4  3  false  "check majority"]
["11"  0  "INIT"    4  3  false  "check majority"]
["11"  0  "INIT"    4  3  true   "check majority"]
["11"  0  "SIGN"    4  3  false  "check majority"]
["11"  0  "SIGN"    4  3  false  "check majority"]
["11"  0  "SIGN"    4  3  true   "check majority"]
["11"  0  "SIGN"    4  3  true   "check majority     but closed"]
["11"  0  "ACCEPT"  4  3  false  "check majority"]
["11"  0  "ACCEPT"  4  3  false  "check majority"]
["11"  0  "ACCEPT"  4  3  true   "check majority"]
["11"  0  "ACCEPT"  4  3  true   "check majority     but closed"]
["12"  0  "INIT"    4  3  false  "check majority"]
["12"  0  "INIT"    4  3  false  "check majority"]
["12"  0  "INIT"    4  3  true   "check majority"]
["12"  0  "SIGN"    4  3  false  "check majority"]
["12"  0  "SIGN"    4  3  false  "check majority"]
["12"  0  "SIGN"    4  3  true   "check majority"]
["12"  0  "SIGN"    4  3  true   "check majority     but closed"]
["12"  0  "ACCEPT"  4  3  false  "check majority"]
["12"  0  "ACCEPT"  4  3  false  "check majority"]
["12"  0  "ACCEPT"  4  3  true   "check majority"]
["12"  0  "ACCEPT"  4  3  true   "check majority     but closed"]
["13"  0  "INIT"    4  3  false  "check majority"]
["13"  0  "INIT"    4  3  false  "check majority"]
["13"  0  "INIT"    4  3  true   "check majority"]
["13"  0  "SIGN"    4  3  false  "check majority"]
["13"  0  "SIGN"    4  3  false  "check majority"]
```

(continues on next page)

(continued from previous page)

```
["13"  0  "SIGN"    4  3  true   "check majority"]
["13"  0  "SIGN"    4  3  true   "check majority     but closed"]
["13"  0  "ACCEPT"  4  3  false  "check majority"]
["13"  0  "ACCEPT"  4  3  false  "check majority"]
["13"  0  "ACCEPT"  4  3  true   "check majority"]
["13"  0  "ACCEPT"  4  3  true   "check majority     but closed"]
["14"  0  "INIT"    4  3  false  "check majority"]
["14"  0  "INIT"    4  3  false  "check majority"]
["14"  0  "INIT"    4  3  true   "check majority"]
```

As the result, the consensus process did not stop, `n0` stores the next block, `13`.

Failure Of Proposing

Under situation

- Proposer is selected after INIT stage.
- Proposer node does not propose the proposal within a given time.
- Timed out in a given time, each node fails to get the proposal from the proposer.

Expected actions

- Each node tries to move the next round.
- Each node broadcasts next INIT ballots for next round.

Listing 6: failure-voting-init-over-blocking-number.yml

```yaml
global:
  modules:
    suffrage:
      name: ConditionSuffrage
      conditions:
        # set proposer to n3 when height, 12
        - condition: suffrage.height="13"
          actions:
            - action: fixed-proposer
              value: n3
    proposal_maker:
      name: ConditionProposalMaker
      delay: 1s
      conditions:
nodes:
  n3:
    modules:
      proposal_maker:
        name: ConditionProposalMaker
        delay: 1s
        conditions:
          # will not make proposal when height, 12 and round, 0
          - condition: proposal.height="13" AND proposal.round in (0)
            actions:
              - action: empty-proposal

conditions:
  all:
```

(continues on next page)

(continued from previous page)

```
    - current_state="booting" AND new_state="joining"
    - current_state="joining" AND new_state="consensus"

    # move to next round
    # {
    #   "level": "debug",
    #   "module": "proposal-timeout",
    #   "count": 0,
    #   "limit": 0,
    #   "callbacks": 1,
    #   "elapsed": 15.210288,
    #   "m": "callback executed"
    # }
    - module="proposal-timeout" AND m LIKE "callback executed" AND elapsed < 50
    - m="check majority" AND height="13" AND round=1 AND stage="INIT" AND is_
↪finished=true

    # new block created
    - m="new block created" AND block.height="13" AND block.round=1
```

```
$ ./contest run failure-proposing.yml --log ./contest-failure-proposing
$ echo $?
0
```

To verify how each node did the consensus process,

```
$ cat ./contest-failure-voting-init-under-blocking-number/n0.log | \
    grep -i 'check majority' | \
    jq -c '[.height, .round, .stage, .total, .threshold, .is_finished, .m]' | \
    column -s ',' -t
 ["n1"  "12"  0    "ACCEPT"  4    3    false  "check majority"]
 ["n2"  "12"  0    "ACCEPT"  4    3    false  "check majority"]
 ["n3"  "12"  0    "ACCEPT"  4    3    false  "check majority"]
 ["n0"  "12"  0    "ACCEPT"  4    3    false  "check majority"]
 ...
 ["n2"  "12"  0    "ACCEPT"  4    3    true   "check majority but closed"]
 ["n2"  "13"  0    "INIT"    4    3    false  "check majority"]
 ["n0"  "13"  0    "INIT"    4    3    false  "check majority"]
 ["n1"  "13"  0    "INIT"    4    3    false  "check majority"]
 ["n3"  "13"  0    "INIT"    4    3    false  "check majority"]
 ["n2"  "13"  0    "INIT"    4    3    false  "check majority"]
 ["n0"  "13"  0    "INIT"    4    3    false  "check majority"]
 ["n0"  "13"  0    "INIT"    4    3    true   "check majority"]
 ["n1"  "13"  0    "INIT"    4    3    false  "check majority"]
 ["n3"  "13"  0    "INIT"    4    3    false  "check majority"]
 ["n3"  "13"  0    "INIT"    4    3    true   "check majority"]
 ["n3"  "13"  0    "INIT"    4    3    true   "check majority but closed"]
 ["n1"  "13"  0    "INIT"    4    3    true   "check majority"]
 ["n2"  "13"  0    "INIT"    4    3    true   "check majority"]
 ["n2"  "13"  0    "INIT"    4    3    true   "check majority but closed"]
 ["n0"  "13"  1    "INIT"    4    3    false  "check majority"]
 ["n3"  "13"  1    "INIT"    4    3    false  "check majority"]
 ["n2"  "13"  1    "INIT"    4    3    false  "check majority"]
 ["n1"  "13"  1    "INIT"    4    3    false  "check majority"]
 ["n0"  "13"  1    "INIT"    4    3    false  "check majority"]
 ["n3"  "13"  1    "INIT"    4    3    false  "check majority"]
```

(continues on next page)

(continued from previous page)

```
["n1"  "13"  1     "INIT"    4     3     false  "check majority"]
["n1"  "13"  1     "INIT"    4     3     true   "check majority"]
["n3"  "13"  1     "INIT"    4     3     true   "check majority"]
["n2"  "13"  1     "INIT"    4     3     false  "check majority"]
["n2"  "13"  1     "INIT"    4     3     true   "check majority"]
["n0"  "13"  1     "INIT"    4     3     true   "check majority"]
["n3"  "13"  1     "SIGN"    4     3     false  "check majority"]
["n1"  "13"  1     "SIGN"    4     3     false  "check majority"]
["n3"  "13"  1     "SIGN"    4     3     false  "check majority"]
["n0"  "13"  1     "SIGN"    4     3     false  "check majority"]
...
["n3"  "13"  1     "SIGN"    4     3     true   "check majority"]
```

The fixed proposer, `n3` did not propose the proposal for height, `13`, round, `0`, the other nodes moved to the next round, `11` for the height, `13`, and then eventually all the nodes did keep the consensus process.

Failure Of SIGN, ACCEPT Stages

Under situation

- Suffrage group members should vote for SIGN stage.
- But some nodes in **acting suffrage group** does not offer the SIGN ballot.
- Timed out in a given time, each node fails to get enough ballots for SIGN stage.

Expected actions

- Each node stops the current vote,
- Each node broadcasts next INIT ballots for next round.

Note:

blocking number In voting, to reach a majority for YES, the YES ballots must be over threshold. *blocking number* is the minimum number to prevent to reach the majority. For example,

- there are 4 total voters,
- threshold for majority is 3,

At this condition, *blocking number* is 2. Simply to say,

```
blocking number = <voters> - <threshold> + 1
```

9.3.3 DRAW: Voting Result Ends in Tie

Under situation

- Suffrage group members should vote for INIT stage.
- But some nodes does offer the different INIT ballot,
- The number of these nodes is over *blocking number*.
- Timed out in a given time, each node fails to get enough ballots for INIT stage.

Expected actions

- Each node stops the current round and tries to start new round.

Listing 7: failure-voting-init-draw.yml

```yaml
# n2 and n3 will make ballot, which has wrong block hash when height, 13 and round, 0; previous round is 0.
nodes:
  n2:
    modules:
      ballot_maker:
        name: ConditionBallotMaker
        conditions:
          - condition: ballot.next_height="13" AND ballot.current_round=0 AND ballot.last_round=0 AND ballot.stage in ("INIT")
            actions:
              - action: random-next_block
  n3:
    modules:
      ballot_maker:
        name: ConditionBallotMaker
        conditions:
          - condition: ballot.next_height="13" AND ballot.current_round=0 AND ballot.last_round=0  AND ballot.stage in ("INIT")
            actions:
              - action: random-next_block

conditions:
  all:
    # base state
    - current_state="booting" AND new_state="joining"
    - current_state="joining" AND new_state="consensus"

    # got ACCEPT majority of block, 12
    - m="check majority" AND height="12" AND round=0 AND stage="ACCEPT" AND is_finished=true

    # got INIT majority of block, 13 and round, 0, but drew
    - m="check majority" AND height="13" AND round=0 AND stage="INIT" AND agreement="DRAW" AND is_finished=true

    # after failed, start next round of block, 12 with round 1
    - m="check majority" AND height="12" AND round=1 AND stage="INIT" AND agreement="MAJORITY" AND is_finished=true

    # got INIT majority of block, 13 and round, 0
    - m="check majority" AND height="13" AND round=0 AND stage="INIT" AND agreement="MAJORITY" AND is_finished=true

    # new block created for height, 13 and next round, 0
    - m="new block created" AND block.height="13" AND block.round=0
```

```
$ ./contest run failure-voting-init-draw.yml \
    --log ./contest-failure-voting-init-draw
$ echo $?
0
```

This is the filtered majority checking messages:

```
$ cat ./contest-failure-voting-init-draw/n0.log | \
    grep -i 'check majority' | \
    jq -c '[.height, .round, .stage, .total, .threshold, .is_finished, .m]' | \
    column -s ',' -t
 ["12"  0  "ACCEPT"  4  3  false  "NOTYET"    "check majority"]
 ["12"  0  "ACCEPT"  4  3  false  "NOTYET"    "check majority"]
 ["12"  0  "ACCEPT"  4  3  true   "MAJORITY"  "check majority"]
 ["12"  0  "ACCEPT"  4  3  true   null        "check majority    but closed"]
 ["13"  0  "INIT"    4  3  false  "NOTYET"    "check majority"]
 ["13"  0  "INIT"    4  3  false  "NOTYET"    "check majority"]
 ["13"  0  "INIT"    4  3  true   "DRAW"      "check majority"]
 ["13"  0  "INIT"    4  3  true   null        "check majority    but closed"]
 ["12"  1  "INIT"    4  3  false  "NOTYET"    "check majority"]
 ["12"  1  "INIT"    4  3  false  "NOTYET"    "check majority"]
 ["12"  1  "INIT"    4  3  true   "MAJORITY"  "check majority"]
 ["12"  1  "INIT"    4  3  true   null        "check majority    but closed"]
 ["12"  1  "SIGN"    4  3  false  "NOTYET"    "check majority"]
 ["12"  1  "SIGN"    4  3  false  "NOTYET"    "check majority"]
 ["12"  1  "SIGN"    4  3  true   "MAJORITY"  "check majority"]
 ["12"  1  "SIGN"    4  3  true   null        "check majority    but closed"]
 ["12"  1  "ACCEPT"  4  3  false  "NOTYET"    "check majority"]
 ["12"  1  "ACCEPT"  4  3  false  "NOTYET"    "check majority"]
 ["12"  1  "ACCEPT"  4  3  true   "MAJORITY"  "check majority"]
 ["12"  1  "ACCEPT"  4  3  true   null        "check majority    but closed"]
 ["13"  0  "INIT"    4  3  false  "NOTYET"    "check majority"]
 ["13"  0  "INIT"    4  3  false  "NOTYET"    "check majority"]
 ["13"  0  "INIT"    4  3  true   "MAJORITY"  "check majority"]
```

9.3.4 Sync

Under situation

- But some nodes make different block,
- The number of these nodes is under *blocking number*.

Expected actions

- These nodes will move to sync state for syncing their block state with network.

Listing 8: sync-bad-block.yml

```yaml
global:
  modules:
    proposal_maker:
      name: ConditionProposalMaker
      delay: 1s
      conditions:
nodes:
  n3:
    modules:
      proposal_validator:
        name: ConditionProposalValidator
        conditions:
          # will make bad block when height, 13 and round, 0
          - condition: block.height="13" AND block.round in (0)
            actions:
```

(continues on next page)

(continued from previous page)

```
                  - action: block-hash
                    value: bk:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

conditions:
  all:
     - current_state="booting" AND new_state="joining"
     - current_state="joining" AND new_state="consensus"

     - m="check majority" AND height="14" AND round=0 AND stage="INIT" AND is_
↪finished=true

  network_creates_new_block:
     # new block created
     - node IN ("n0", "n1", "n2") AND m="new block created" AND block.height="13" AND␣
↪block.round=0
     - node IN ("n0", "n1", "n2") AND m="new block created" AND block.height="15" AND␣
↪block.round=0

  but_n3_move_to_sync:
     - node="n3" AND current_state="consensus" AND new_state="syncing"
```

```
$ ./contest run sync-bad-block.yml --log ./contest-sync-bad-block
$ echo $?
0
```

This is the filtered new block messages:

```
$ cat ./contest-sync-bad-block/all.log | \
     grep -i 'new block created' | \
     jq -c '[.node, .block.height, .block.round, .block.hash.hash]' | \
     column -s ',' -t
 ["n0",10,0,"bk:8rZSbCCNeEh1e6cecsfWa6Zx6ZuxbhhriEp8gGtsU5qQ"]
 ["n1",10,0,"bk:8rZSbCCNeEh1e6cecsfWa6Zx6ZuxbhhriEp8gGtsU5qQ"]
 ["n2",10,0,"bk:8rZSbCCNeEh1e6cecsfWa6Zx6ZuxbhhriEp8gGtsU5qQ"]
 ["n3",10,0,"bk:8rZSbCCNeEh1e6cecsfWa6Zx6ZuxbhhriEp8gGtsU5qQ"]
 ["n0",11,0,"bk:DobUDexxfWjLznXGLEajtDQzGfgQ9yQYg3JxHJkmm1To"]
 ["n1",11,0,"bk:DobUDexxfWjLznXGLEajtDQzGfgQ9yQYg3JxHJkmm1To"]
 ["n2",11,0,"bk:DobUDexxfWjLznXGLEajtDQzGfgQ9yQYg3JxHJkmm1To"]
 ["n3",11,0,"bk:DobUDexxfWjLznXGLEajtDQzGfgQ9yQYg3JxHJkmm1To"]
 ["n0",12,0,"bk:C6JyiHt9q5GPDenoNxPzohCB52XXBZ7BcuwyzPbgpi5P"]
 ["n1",12,0,"bk:C6JyiHt9q5GPDenoNxPzohCB52XXBZ7BcuwyzPbgpi5P"]
 ["n2",12,0,"bk:C6JyiHt9q5GPDenoNxPzohCB52XXBZ7BcuwyzPbgpi5P"]
 ["n3",12,0,"bk:C6JyiHt9q5GPDenoNxPzohCB52XXBZ7BcuwyzPbgpi5P"]
 ["n0",13,0,"bk:4CdvUY1w3jsn1cT1KW1AoGFicBHe3SeQDgzHy4YjjHSr"]
 ["n1",13,0,"bk:4CdvUY1w3jsn1cT1KW1AoGFicBHe3SeQDgzHy4YjjHSr"]
 ["n2",13,0,"bk:4CdvUY1w3jsn1cT1KW1AoGFicBHe3SeQDgzHy4YjjHSr"]
 ["n0",14,0,"bk:AXLi5g43jiyAXiK6G6dpE5uX6wjExWZEMF3ki1CnT7us"]
 ["n1",14,0,"bk:AXLi5g43jiyAXiK6G6dpE5uX6wjExWZEMF3ki1CnT7us"]
 ["n2",14,0,"bk:AXLi5g43jiyAXiK6G6dpE5uX6wjExWZEMF3ki1CnT7us"]
 ["n0",15,0,"bk:ANVfJE2gufoX2yc6ByyaHgMyb5Dbd7tZCszjrbviuyzc"]
 ["n1",15,0,"bk:ANVfJE2gufoX2yc6ByyaHgMyb5Dbd7tZCszjrbviuyzc"]
 ["n2",15,0,"bk:ANVfJE2gufoX2yc6ByyaHgMyb5Dbd7tZCszjrbviuyzc"]
```

Note: The current contest and mitum is under heavily development and is based on proto2.

CHAPTER 10

Network

Mitum network consists of the distributed separated nodes. These nodes comprise the mitum network. Basically mitum network has one decision at a time and the decision will be established as the Block.

CHAPTER 11

Node and Group

In mitum network, there are several types of nodes by its role and situations.

11.1 Home Node

Node owner calls his/her node as home node.

11.2 Suffrage Group

- Basically this member will keep the network going correctly.
- This member has the right to participate the consensus process.
    - Node, not in suffrage group can not participate the consensus process.
- The new block is established by the agreement of suffrage group.
- This member also have the privilege to vote on the decision for network policies.
- The member of this group is not hard-coded or managed by another system. The member of this group can be decided by the agreement of network.

11.3 Acting Suffrage Group

- ISAAC+ will open the new round for establishing the next new block. In new round, not all the suffrage group members participate, only the limited members can join.
- In new round, the limited members is selected randomly, these members will be "acting suffrage group members".
- The member of acting suffrage group will be tested and challenged, whether it works correctly without problem by the other suffrage group members.

Note: If some node in acting suffrage group members works weirdly, it may be exiled by network policy.

11.4 Proposer

- In new round, the new proposer also be selected within acting suffrage members.
- Basically proposer will propose new proposal for next block.

11.5 Non-Consensus Node

- This node is all the nodes not to belong to the suffrage group.
- This node will keep track of the established blocks from suffrage group.
- If this node proves itself to have enough qualifications for consensus process, this node will be one of the suffrage group.

11.6 Faulty Node

Faulty node(or failed node) is the failed or malfunctioning node. Node can be faulty node in these cases,

- Maybe crashed, or
- has the different states to other nodes, or
- responses the different acts to other nodes, etc.

Any node can be faulty node, whether it is acting suffrage group, suffrage group or even not in both.

Commonly faulty node is:

- When it does not respond the request from other nodes
- When it seems to have the different block state to the nodes

Faulty node in suffrage group is:

- When it does not serve *BlockProof*
- When it serves the invalid *BlockProof*
- When it does not serve *VotingProof*
- When it serves the invalid *VotingProof*

Faulty node in acting suffrage group is:

- When it is proposer and does not broadcast proposal
- When it does not broadcast ballots
- When it broadcast the invalid ballot

CHAPTER 12

Designing Network

Mitum is designed for general purpose blockchain. To fill this requirement, policy and data of mitum can be configurable and manageable by practical way. The network designer will design his/her network in 2 parts:

- Data
- Policy

12.1 Data

Simply to say, data is the contents of block. In block any kind of arbitrary contents can be stored. There are several built-in data types in mitum, and new types of data can be defined by the network designer.

Roughly data can be categorized by 2 kinds:

- Defined data
- Anonymous data

All data belongs to the predefined type and has the unique id within globally.

Defined data

is the data, which is statically defined outside block. It is managed in each node and shared thru network. It is under control of its type.

Anonymous data

represents any kind of undefined data.

The size of data is limited up to certain amount by network policy. Basically data can be created, updated and removed.

12.2 Policy

Most of distributed system should share the basic principles to the siblings. These principles can be shared and should be synced. For example, how many nodes should be selected as acting suffrage group and the way to select proposer from acting suffrage group.

In mitum these kinds of principles, most of the policies are managed in block like data. This means that:

- Policy can be shared to the entire network without additional mechanism.
- Policy can be updated by consensus like data.

Note: The initial policy is set by the network designer.

12.3 Model

By designing data and policy, the designer can build and launch his/her own model of network.

For example, the designer want to build currency model in mitum. He/Her can define several currencies and it's related data and add additional policy.

Data types:

- Account
- Balance

Policy:

- Total amount
- Minimum amount of new balance
- Multisig
- Inflation
- etc

CHAPTER 13

Contribution

Mitum started as open source project and it will be. Any kind of contribution will be welcome. At this time, mitum needs your help at these parts:

- Development
- Documentation
- Applications for mitum

When you have idea or proposal for mitum, please submit new issue at github or email to spikeekips@gmail.com.

This document is managed at github. Please feel free to submit your PR.

Bibliography

[PBFT] Miguel Castro and Babara Liskov at 1999, [Practical Byzantine Fault Tolerance](http://pmg.csail.mit.edu/papers/osdi99.pdf)

Index