이스탐불 내부구조

합의 알고리즘 차이점

**BFT(Byzantine Fault Tolerance)**

|  |  |  |  |
| --- | --- | --- | --- |
| 구분 | 이스탐불(Istanbul BFT) | P(Practical)BFT | PODC |
|  | 1. Pre-Prepare 2. Prepare 3. Commit | Client  Primary  Backup #1  Backup #2  Backup #3( 배신자) | Front Node  일반노드  운영위노드  코디네이터  상임위노드 |
|  |  | Request  Pre-prepare  Prepare  Commit  reply | Request  Pre-prepare  d-select  d-commit |
|  | proposer를 뽑는다 |  | 코디네이터 뽑는다. |
|  |  |  |  |
|  |  |  |  |

<https://steemit.com/kr/@kanghamin/istanbul-byzantine-fault-tolerance>

Istanbul Byzantine Fault Tolerance(=IBFT)에서는 3단계로 합의과정이 이루어져있다.

1.Pre-Prepare

2.Prepare

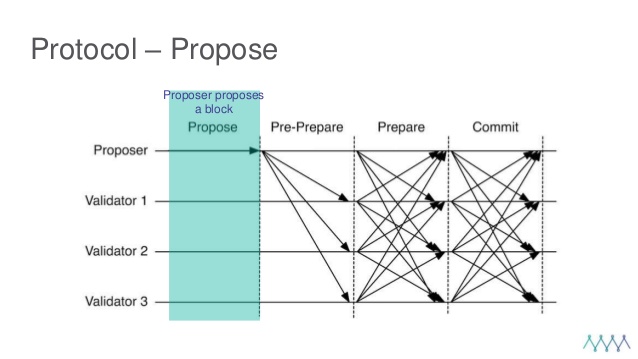
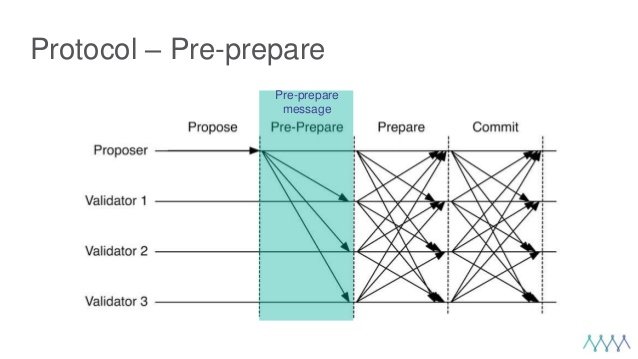
3. Commit

시스템은 잘못된 노드(나쁜 노드)가 F라고 가정했을때 총 노드수가 3F+1이상이면 돌아갈 수 있다.

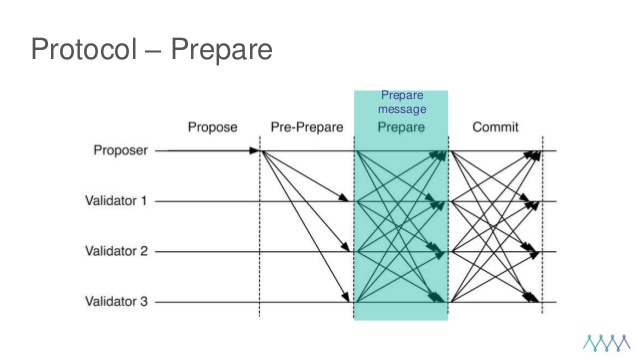
* N = 3F +1 ( N= node, F = faulty node)

합의를 이끌어내는 이들을 검증자 (Validators)라고 하며 검증자들은 블록을 생성하기전(매 라운드 전)에 proposer를 뽑는다. 그럼 이제부터 합의과정이 시작되는데,

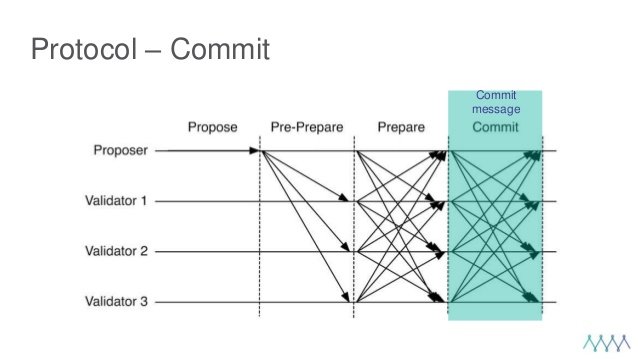
첫째, proposer 는 새로운 블록을 선택하고 제안하며 이것을 pre-prepare 메세지와 함께 공표한다.

둘째, 검증자들은 proposer로부터 pre-prepare 메세지를 받으면 pre-prepare 상태로 돌입하며 prepare 메세지를 보낸다. 이과정은 모든 검증자들이 같은 라운드, 같은 순서에 있는지 확인하기 위함이다.



셋째, 2F+1 의 prepare 메세지를 받으면 검증자들은 prepare 상태로 돌입하며 commit 메세지를 보낸다.



이 과정은 제안된 블록이 수락되었다는것은 피어들에게 알리는 역할을 하며 수락된 블록을 체인에 연결시킨다. 2F+1 의 commit 메세지를 받으면 commit 상태로 돌입하며 그 블록은 체인에 연결된다.

전제 알고리즘은 그림과 같고 합의가 실패한경우 Round change로 돌아가 다시 라운드를 시작한다.

합의과정은 3단계로 이루어져있으며 IBFT에서 블록은 Finality를 의미한다. 즉, 포크가 불가능하며 유효한 블록은 무조건 메인체인에 존재한다. 잘못된 (나쁜) 노드에 의해 잘못된 체인이 생성되는것을 방지하기위해 블록을 체인에 연결하기전 검증자들은 commit 서명(2F+1)을 헤더안의 extra data에 첨부한다. 그러므로, 블록들은 스스로 검증가능하며 라이트 클라이언트로 지원가능하다.

하지만 Dynamic extra data는 블록 해쉬계산에 문제를 일으킬 수 있다. 같은 블록이여도 검증자마다 다른 commit 서명을 가질 수 있으므로 블록해쉬도 다르게 가질 수 있기때문이다. 이를 해결하기 위해서 IBFT에선 commit 서명을 제외한채 블록해쉬를 계산한다. 그러면 블록헤더에 Consensus proof 도 넣을 수 있을뿐만 아니라 블록/블록해쉬의 일관성도 유지할 수 있다.

참조 : <https://github.com/ethereum/EIPs/issues/650>  
<https://www.slideshare.net/YuTeLin1/istanbul-bft>

**state machine 은 별도 파일 참조.**

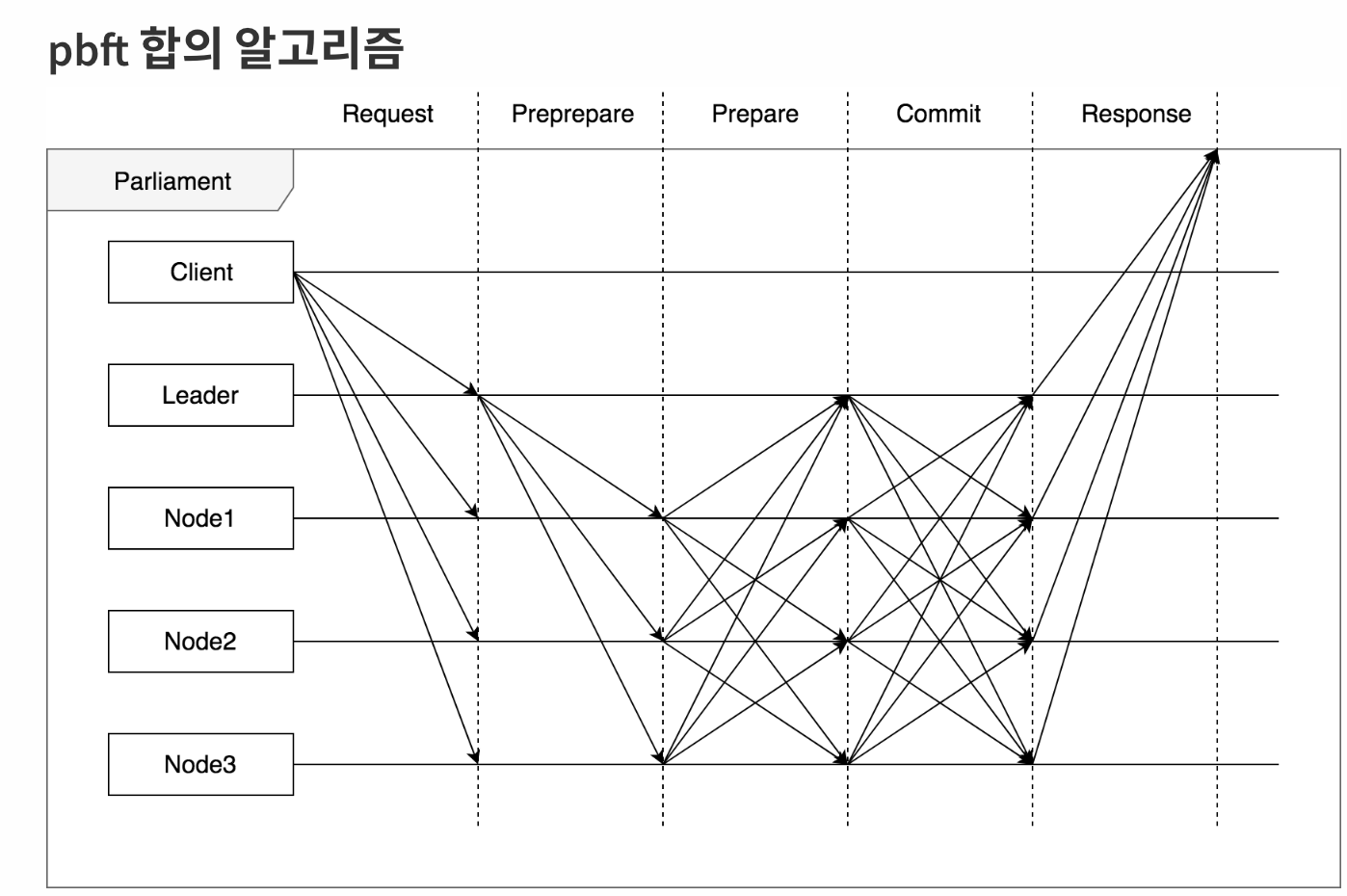
Istanbulbft-170704103901.pdf

**================================**

**PBFT(Practical Byzantine Fault Tolerance)**

PBFT는 PoW나 PoS와 마찬가지로 Byzantine Fault 모델이지만 **PoW와 PoS의 단점인 파이널리티의 불확실성과 성능 문제를 해결한 것**입니다. Hyperledger Fabric과 Eris 등 컨소시엄형에서 이용하고 있는 블록체인 기반 기술에 많이 채택되고 있습니다.

PBFT는 네트워크의 모든 참여자를 미리 알고있어야 합니다. 참가자 1명이 프라이머리(Primary, 리더)가 되고 자신을 포함한 모든 참가자에게 요청을 보냅니다. 그 요청에 대한 결과를 집계한 뒤 다수의 값을 사용해 블록을 확정합니다. 부정한 노드 수를 n개라고 하면 노드 수는 3n+1개여야 하며, 확정에는 n+1개 이상의 노드가 필요합니다.



기본적인 PBFT 알고리즘은 다음과 같은 순서로 이루어지게 됩니다.

1. 클라이언트가 네트워크 구성원에게 어떤 합의문에 대해 합의할 것을 요청합니다.
2. 합의 요청을 받은 노드들 중 리더 노드는 네트워크 내의 모든 구성원에게 특정 합의에 대한 합의를 시작할 것을 알리는 **preprepare message** 를 전달합니다.
3. **preprepare message** 를 전달받은 모든 노드는 다시 모든 노드에게 **prepare message** 를 전달합니다.
4. 전체 네트워크 구성원들 중 정족수(2/3) 이상의 노드에게 받은 **preparemessage** 인 **commit message** 를 모든 노드들에게 전달합니다.
5. 위 과정이 끝나면 모든 노드들은 정족수이상이 합의한 결과를 가지게 됩니다.

pbft의 목적은 어디까지나 합의하고자 하는 블록에 대한 합의이기에 다음과 같은 과정에 따라 pbft 기반의 합의 알고리즘이 동작됩니다.

1. 리더 노드의 consensus 컴포넌트가 블록 생성을 담당하는 blockchain 컴포넌트로 부터 합의하고자 하는 블록을 제안받습니다.
2. 리더 노드는 해당 블록에 대한 합의를 진행하기 위해 현재 네트워크를 구성하고 있는 정보를 담고 있는 **PeerTable** 에서 전부 혹은 부분적인 노드를 선출하여 **Parliament** 를 구성합니다.
3. 리더 노드가 **Parliament** 에 속한 모든 **Representative** 들에게 **preprepare message** 를 전달합니다.
4. **preprepare message** 를 받은 모든 **representative** 들은 받은 정보를 통해 각자의 **Parliament** 를 구축하고 다른 **Representative** 들에게 **prepare message** 를 전달합니다.
5. 각 **Representative** 들은 정족수 이상의 prepare 메세지를 받기 까지 받은 모든 **prepare message** 를 **prepare message pool** 에 저장하고 정족수 이상이 넘으면 **commit message** 를 전파합니다.
6. 전체 의회 구성원의 1/3 이상에게 commit message를 받은 각 **Representative** 들은 **Commit message** 내의 **Proposed Block** 에 대한 승인을 하는 이벤트를 발생시키고 블록체인 컴포넌트는 해당 블록에 대한 검증을 시작합니다.  
   **Proposed Block** 이 confirm 되기 까지 **Commit message** 들은 **Commit message pool** 에 저장됩니다.

*// Istanbul is a consensus engine to avoid byzantine failure*type Istanbul interface {  
 Engine  
  
 *// Handle a message from peer* HandleMsg(pubKey \*ecdsa.PublicKey, data []byte) error  
  
 *// Receive new chain head block* NewChainHead(block \*types.Block)  
  
 *// Start the engine* Start(chain ChainReader, inserter func(block \*types.Block) error) error  
  
 *// Stop the engine* Stop() error  
}

*// Engine is an algorithm agnostic consensus engine.*type Engine interface {  
 *// Author retrieves the Ethereum address of the account that minted the given  
 // block, which may be different from the header's coinbase if a consensus  
 // engine is based on signatures.* Author(header \*types.Header) (common.Address, error)  
  
 *// VerifyHeader checks whether a header conforms to the consensus rules of a  
 // given engine. Verifying the seal may be done optionally here, or explicitly  
 // via the VerifySeal method.* VerifyHeader(chain ChainReader, header \*types.Header, seal bool) error  
  
 *// VerifyHeaders is similar to VerifyHeader, but verifies a batch of headers  
 // concurrently. The method returns a quit channel to abort the operations and  
 // a results channel to retrieve the async verifications (the order is that of  
 // the input slice).* VerifyHeaders(chain ChainReader, headers []\*types.Header, seals []bool) (chan<- struct{}, <-chan error)  
  
 *// VerifyUncles verifies that the given block's uncles conform to the consensus  
 // rules of a given engine.* VerifyUncles(chain ChainReader, block \*types.Block) error  
  
 *// VerifySeal checks whether the crypto seal on a header is valid according to  
 // the consensus rules of the given engine.* VerifySeal(chain ChainReader, header \*types.Header) error  
  
 *// Prepare initializes the consensus fields of a block header according to the  
 // rules of a particular engine. The changes are executed inline.* Prepare(chain ChainReader, header \*types.Header) error  
  
 *// Finalize runs any post-transaction state modifications (e.g. block rewards)  
 // and assembles the final block.  
 // Note: The block header and state database might be updated to reflect any  
 // consensus rules that happen at finalization (e.g. block rewards).* Finalize(chain ChainReader, header \*types.Header, state \*state.StateDB, txs []\*types.Transaction,  
 uncles []\*types.Header, receipts []\*types.Receipt) (\*types.Block, error)  
  
 *// Seal generates a new block for the given input block with the local miner's  
 // seal place on top.* Seal(chain ChainReader, block \*types.Block, stop <-chan struct{}) (\*types.Block, error)  
  
 *// APIs returns the RPC APIs this consensus engine provides.* APIs(chain ChainReader) []rpc.API  
}