Near protocol

Outline

- Protocol Overview
- Threshold Proof of Stake (TPoS)
- **Nightshade**
  - Block structure
  - Doomslug
  - Epoch Switch
  - Data availability
  - Cross-shard tx
  - Validation
- Slashing

Protocol Overview

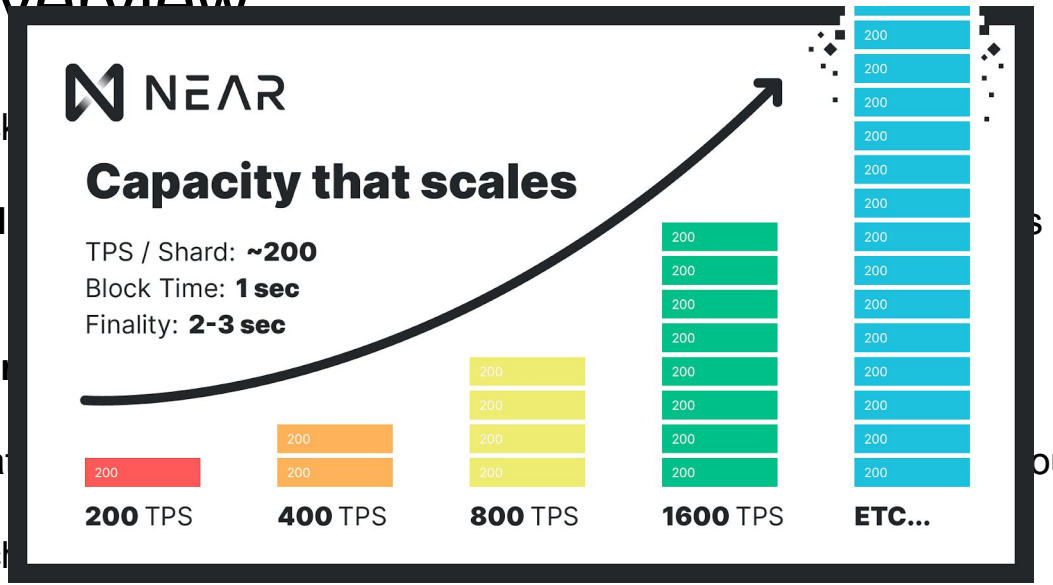
Epoch: ½ day block

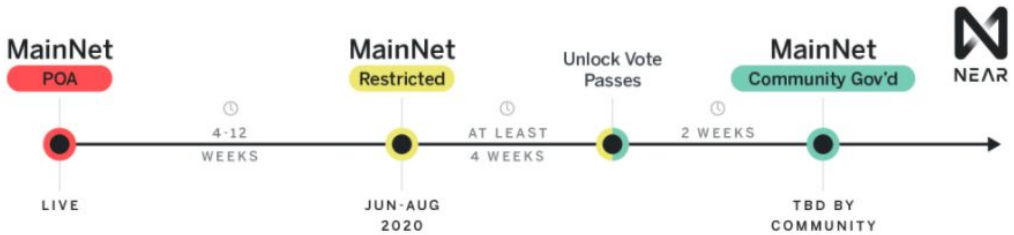
Validator: **Block Prod** s is proportional to uptime and stake

Non-Validator: (**Fisher**

If the validator generat out and lose its seat.

Re-stake after 2 epoch





TPoS (determining the **minimum threshold** number of tokens for a single seat)

Validator: stake >= seat price

Auction: how many "**seats**" will be allocated to each prospective validator

$$seatPrice = argmax_{x \in N}(\sum_{v \in V} \lfloor \frac{stake_v}{x} \rfloor \geq numSeats)$$

With a fixed stake, the more the seat, the lower the price

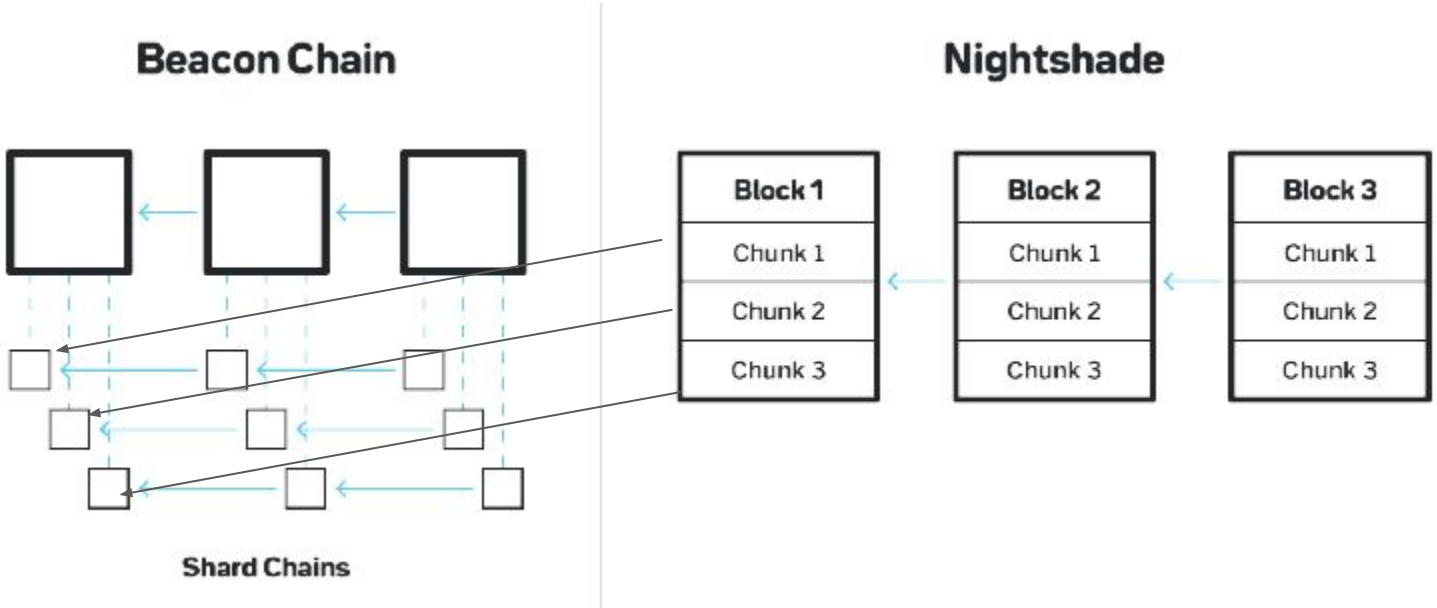
With a fixed seats, the more the stake, the higher the price

At the end of the ***epoch T−1***, everyone on the network run **auction** to determine validators for the ***epoch T+1***

Nightshade (mainchain)

From shard chains to shard chunks, block contains chunks, chunk contains tx

Validator only maintains && validates the state of their corresponding shards



```rust
pub struct BlockV1 {
    pub header: BlockHeader,
    pub chunks: Vec<ShardChunkHeader>,
    pub challenges: Challenges,

    // Data to confirm the correctness of randomness beacon output
    pub vrf_value: near_crypto::vrf::Value,
    pub vrf_proof: near_crypto::vrf::Proof,
}
```

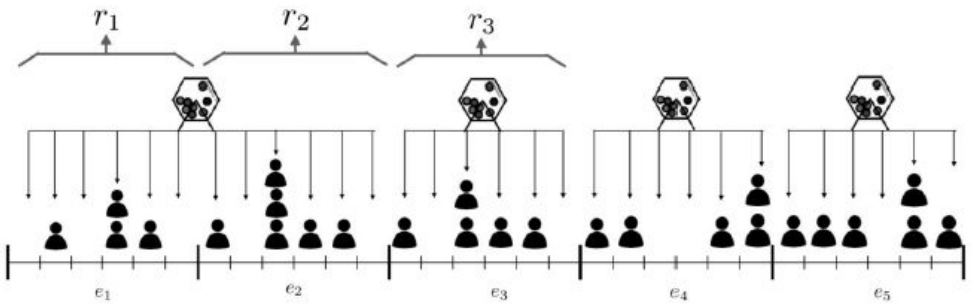
```rust
struct BlockHeader {
    ...
    prev_hash: BlockHash,
    height: BlockHeight,
    epoch_id: EpochId,
    last_final_block_hash: BlockHash,
    approvals: Vec<Option<Signature>>
    ...
}
```

```rust
pub struct ShardChunkHeader {
    pub inner: ShardChunkHeaderInner,

    pub height_included: BlockHeight,

    /// Signature of the chunk producer.
    pub signature: Signature,

    #[borsh_skip]
    pub hash: ChunkHash,
}
```

BABE: VRF (Verifiable Random Function)

(v, π) ← ***VRF(r, i, sk)***, where

- *r* = Epoch randomness
- *i* = Slot number
- *sk* = Secret key



Near VRF:

(v, π) ← ***VRF(r, sk)***, where

- r = Epoch last block randomness
- sk =secret key

Use ***V*** to shuffle validator → determine production schedule,

Therefore it doesn't have ***multiple-candidates*** or ***no-candidates*** problem

Total **9** seats

V1	V1	V1	V2	V2	V3	V3	V4	V5

Block Producer 6 seats
Hidden Validator 3 seats

V2	V1	V3	V4	V1	V2	V3	V1	V5

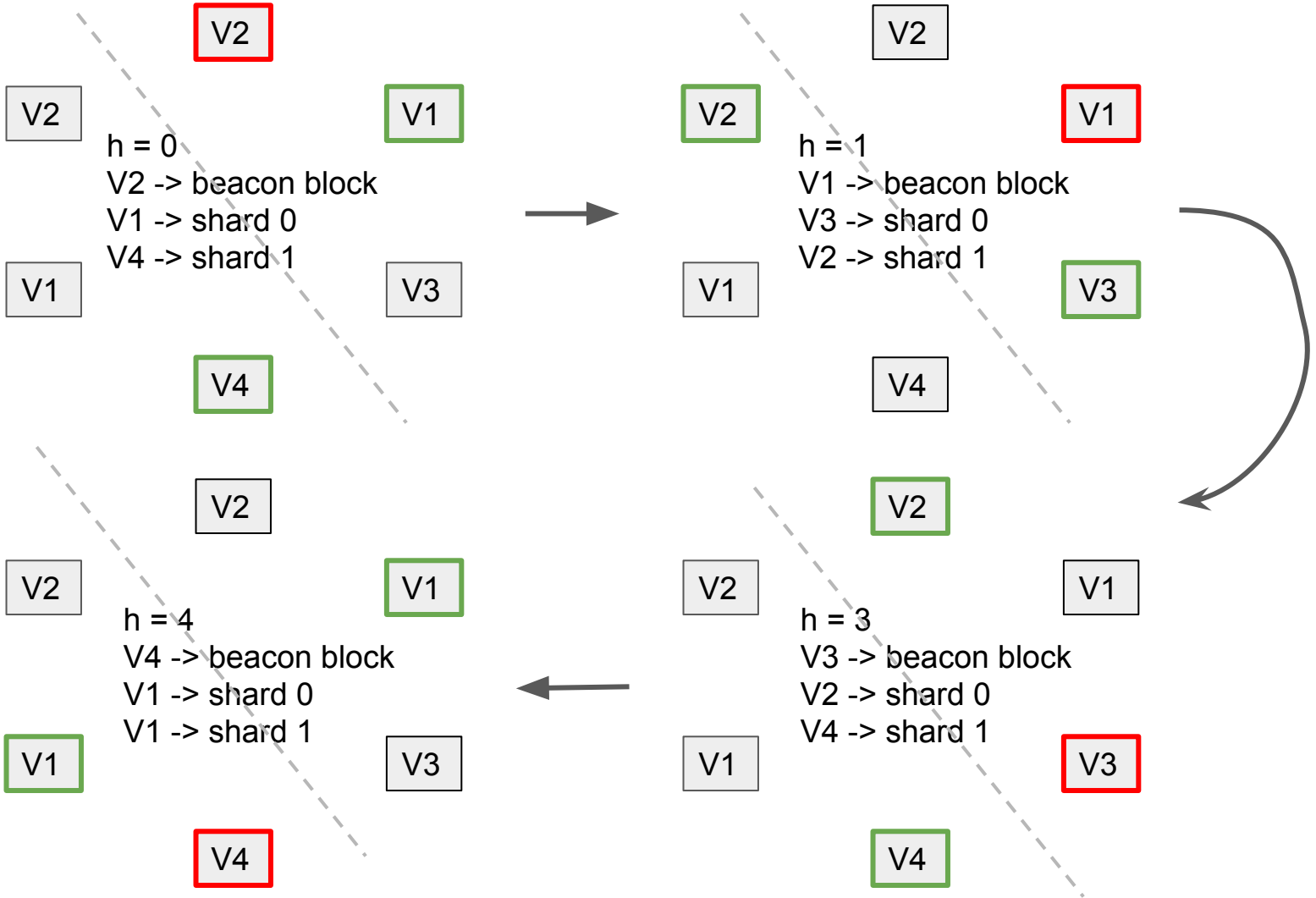
Each shard 3 seats

V2	V1	V3	V4	V1	V2	V3	V1	V5

Shard 0 Shard 1

Note: BP & CP cannot be the same guy at the same time, BP has the higher priority

One validator can work for multiple shards


V2
V2
V1
h = 0
V2 -> beacon block
V1 -> shard 0
V4 -> shard 1
V1
V3
V4
V2
V2
V1
h = 1
V1 -> beacon block
V3 -> shard 0
V2 -> shard 1
V1
V3
V4
V2
V2
V1
h = 3
V3 -> beacon block
V2 -> shard 0
V4 -> shard 1
V1
V3
V4
V2
V2
V1
h = 4
V4 -> beacon block
V1 -> shard 0
V1 -> shard 1
V1
V3
V4

Doomslug

Block can be produced in **1 round** as long as >⅔ **honest** BP online

V(h) sends **only one** of approvals for Block(h):

partially synchronous

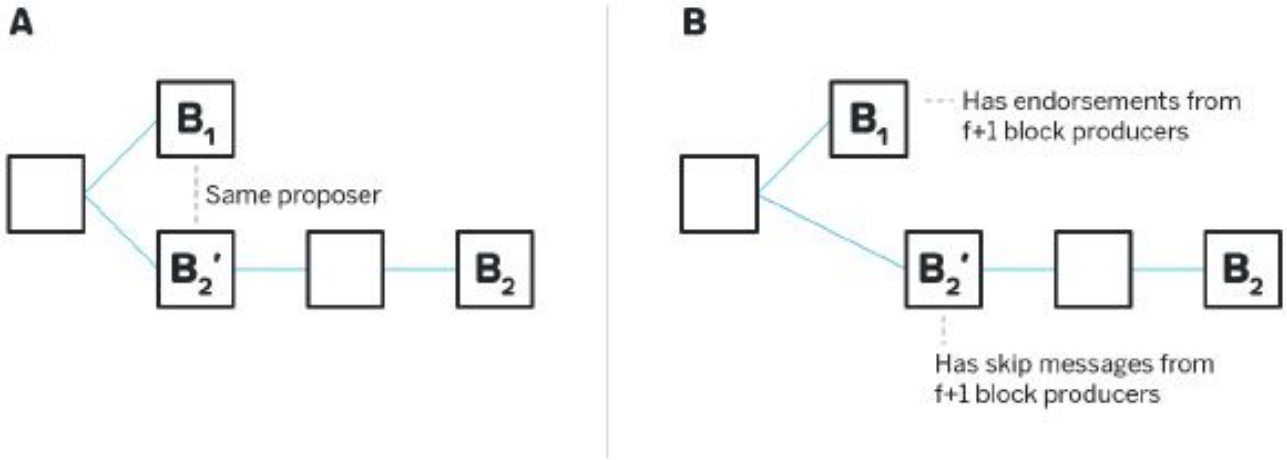
- *E(B,v)*: Endorsement (if received block)
- *S(h, n, v)*: Skip *n* blocks starting at height *h*

BP(h+1) collects of *E* or *S* from >⅔ V(h) and produce Block(h+1)

Finality condition: Block is finalized if at least 2 blocks build on top of it and these 3 blocks have consecutive height

1. conflicting endorsement:
   honest BP can never produce 2 endorsements with the same prev_height

2. conflicting skip and endorsement:
   honest BP can never produce both skip or endorsement for same target

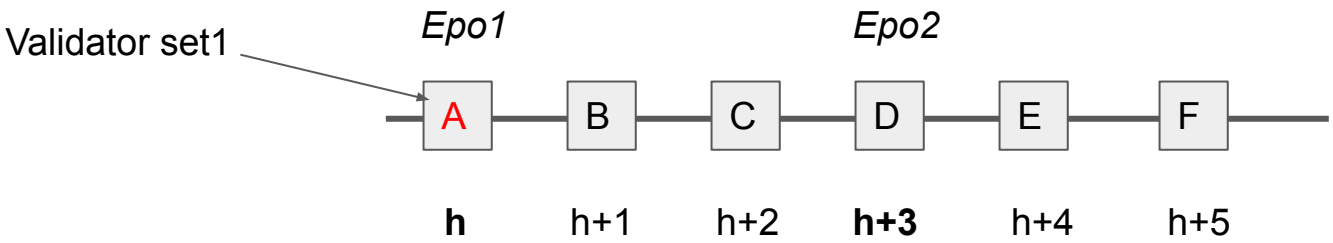


Epoch switches

Epoch starts at height ***h***, epoch length >= **3**

1. *h(prev(B))* < ***h*** + *epoch_length* - 3 (if the block is **not** in the **last two** blocks of *e_cur*)

   → *Epoch* 1, collect ⅔ approval from current epoch



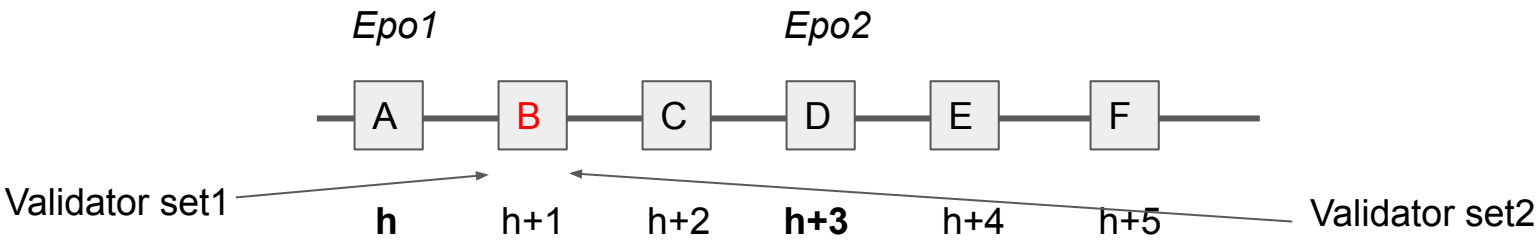
E.g. epoch_length = 3

Block A: *Epoch* 1, ⅔ approvals of *BP(e_cur)*

*2. h(prev(B)) >= **h** + epoch_length - 3 && h(last_final(prev(B))) < **h** + epoch_length - 3*

(if the block is the **second last** block of e_cur)

→ *Epoch* 1, collect ⅔ approval from current epoch && ⅔ approval from next epoch

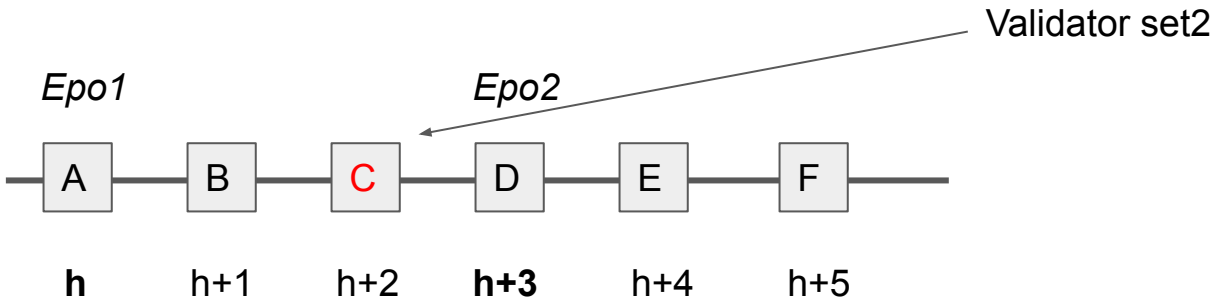


E.g. epoch_length = 3

Block B: *Epoch* 1, ⅔ approvals of *BP(e_cur && e_next)*

*3. h(last_final(prev(B))) >= **h** + epoch_length - 3* (if the block is the **last** block of e_cur)

→ *Epoch* 2, collect ⅔ approval from next epoch

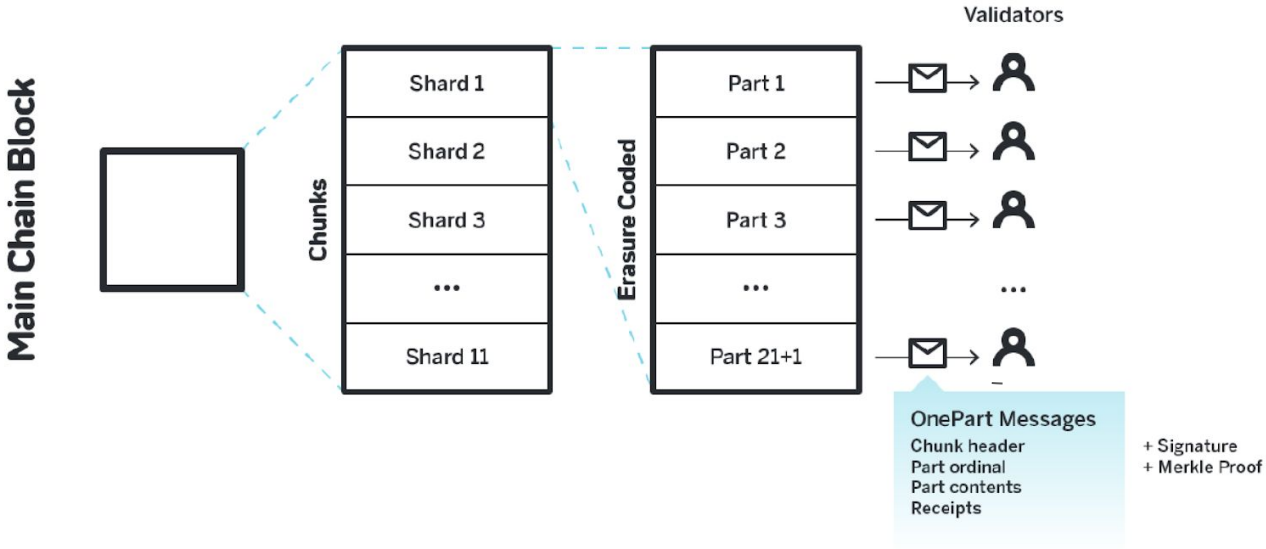


E.g. epoch_length = 3

Block C: *Epoch* 2, ⅔ approvals of *BP(e_next)*

Erasure Code

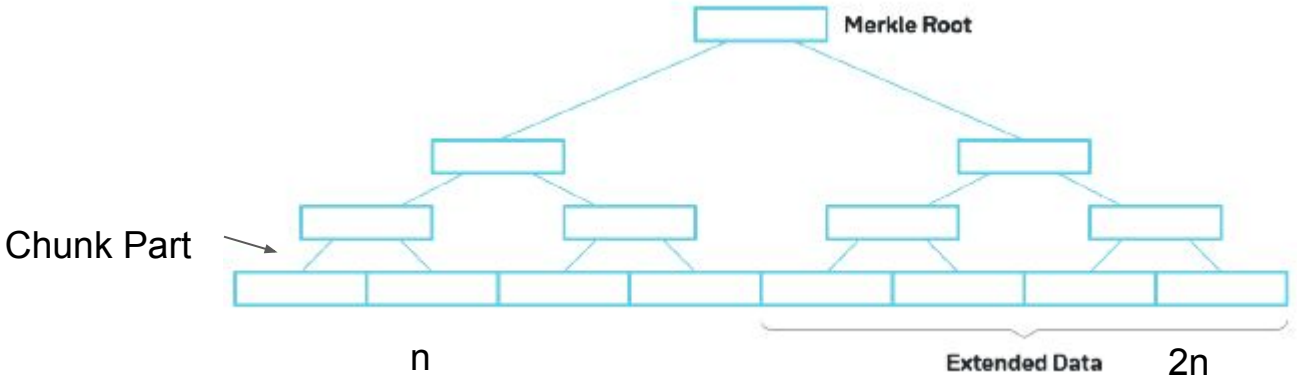
Since no participant downloads the full state, to ensure the data availability, CP create an erasure coded version of a chunk, ⅛ chunk parts can reconstruct the whole chunk.



Shard0 (part2)
Shard1 (part1)
Shard2 (part4)
...

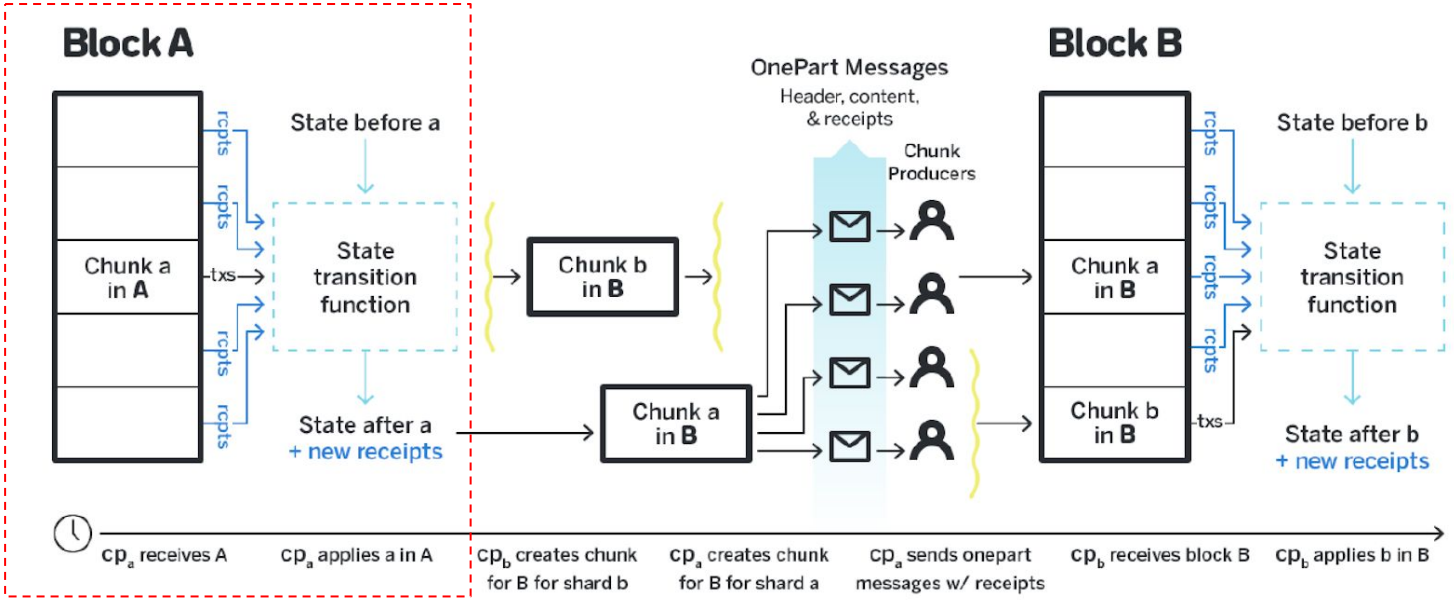
Validator process a main chain block only if they have **all** *onepart* messages for each chunk.

Chunk Producers fetche other parts from the peers and reconstructs the chunks they care about.



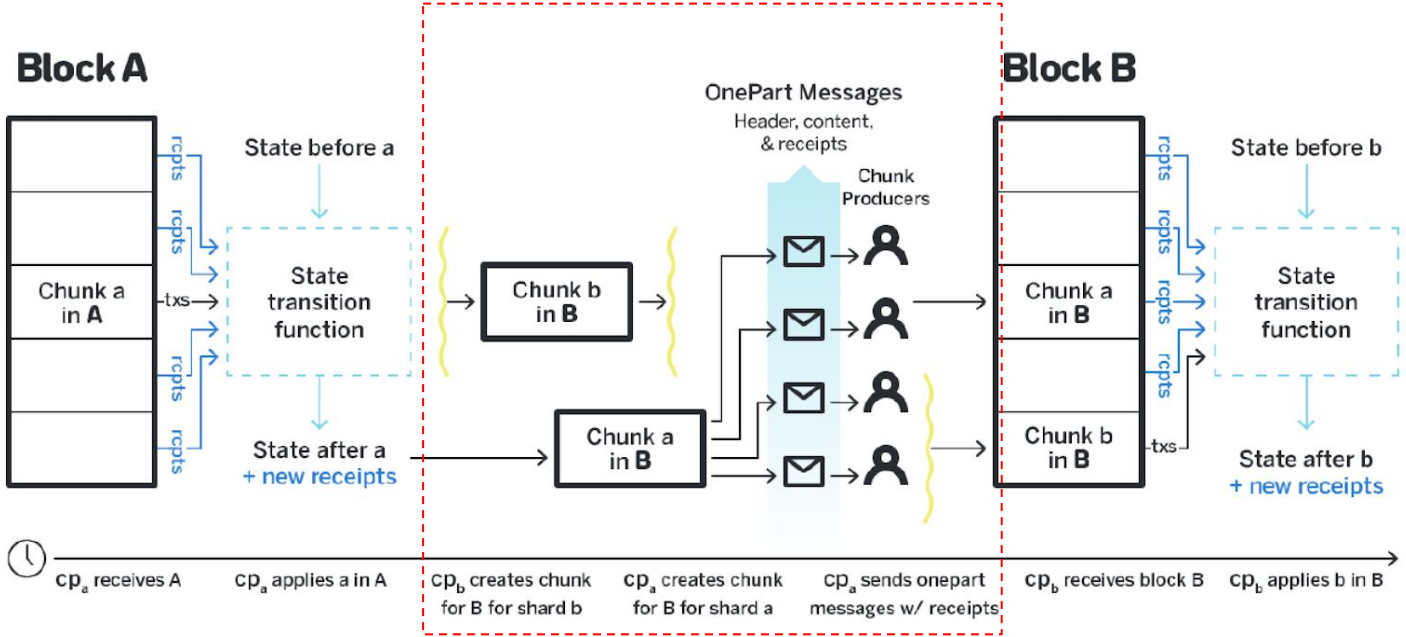
Cross-shard transactions

Producing and storing the receipts. cp_a for shard ***a*** receives block A, applies tx and generates receipt ***r***. cp_a stores all receipts by the source shard id.

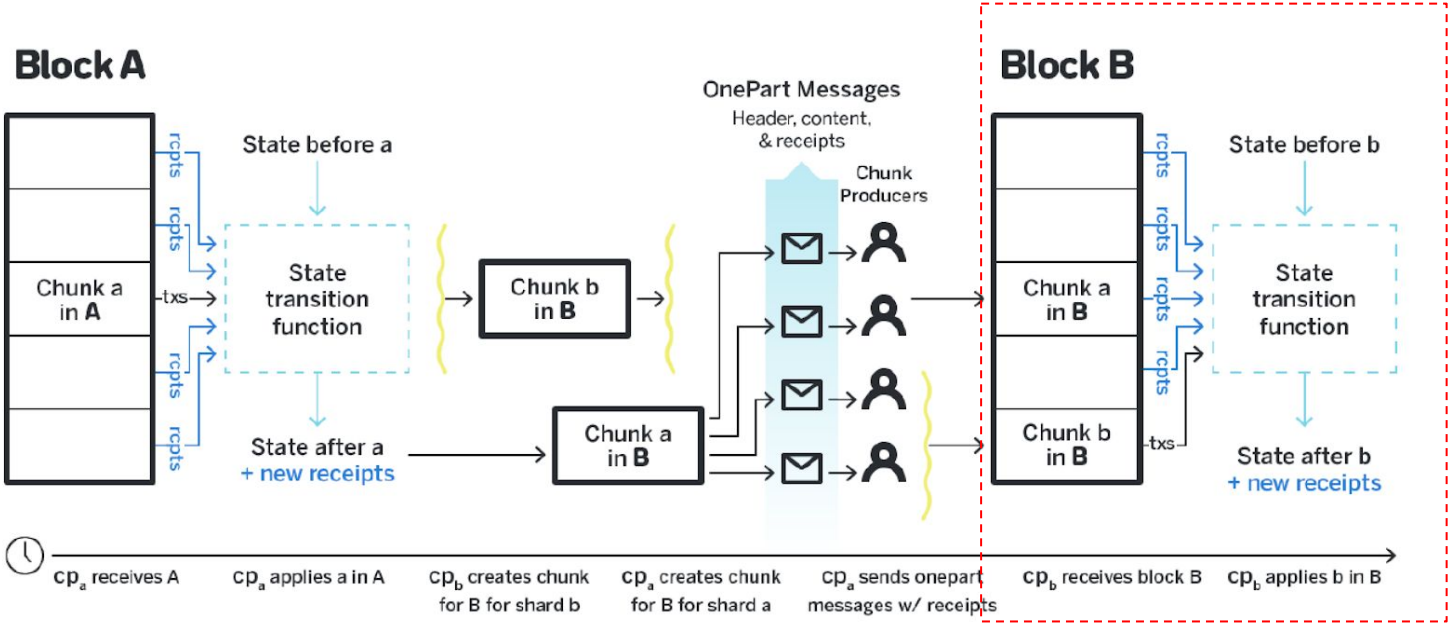


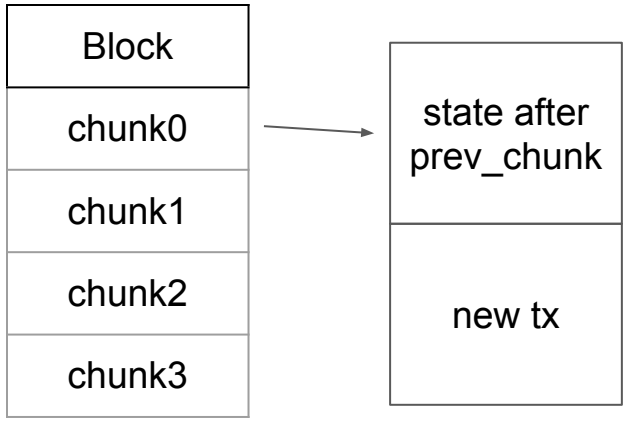
https://www.youtube.com/watch?v=mhJXsOKoSdg&list=PL9tzQn_TEuFW_t9QDzlQJZpEQnhcZte2y&index=1

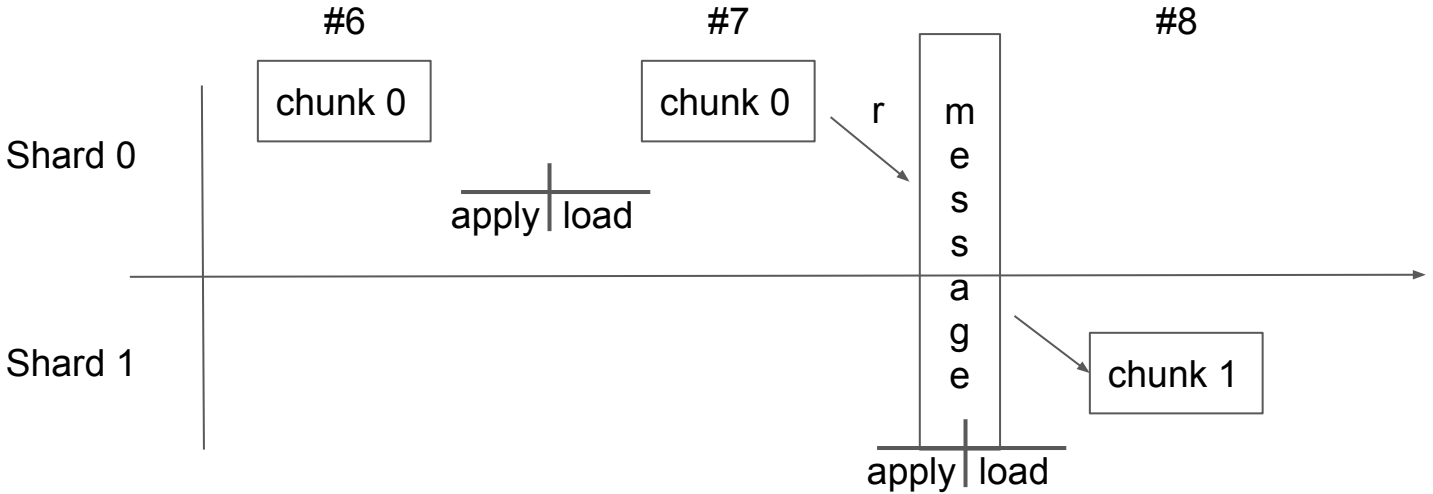
Distributing the receipts. cp_a includes receipts into chunk ***a*** for block B. Distributes the receipt to the particular BP, who cares about as the destination, in the *onepart* message.



Receiving the receipts. If block B have all the *onepart* messages, they have all incoming receipts. Participant apply both the receipts, as well as the transactions.




Block
chunk0
chunk1
chunk2
chunk3
state after prev_chunk
new tx


#6
#7
#8
chunk 0
chunk 0
r
message
Shard 0
Shard 1
apply
load
chunk 1
apply
load

Chunk validation

Chunk can only be validated by the participants that maintain the state.

Limit L_s bytes of state that a single tx can cumulatively read / write. Any tx touches more than L_s state is invalid. If there is invalid chunk, provide challenge.

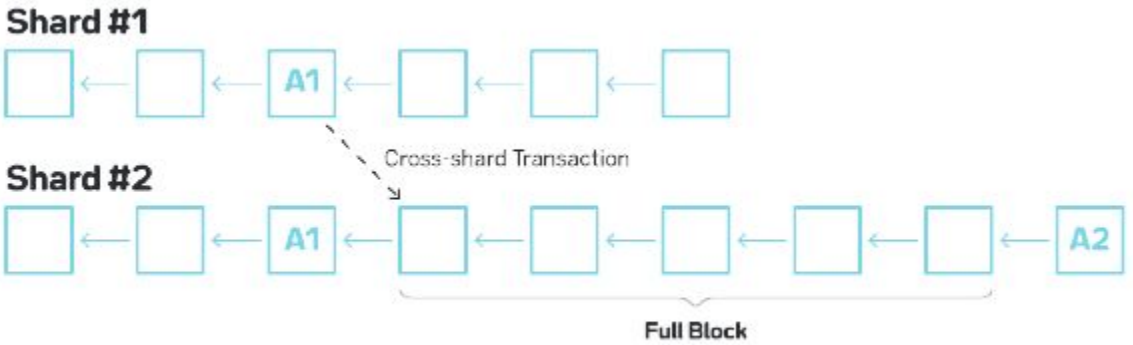
```rust
pub struct Challenge {
    pub body: ChallengeBody,
    pub account_id: AccountId,
    pub signature: Signature,

    #[borsh_skip]
    pub hash: CryptoHash,
}
```

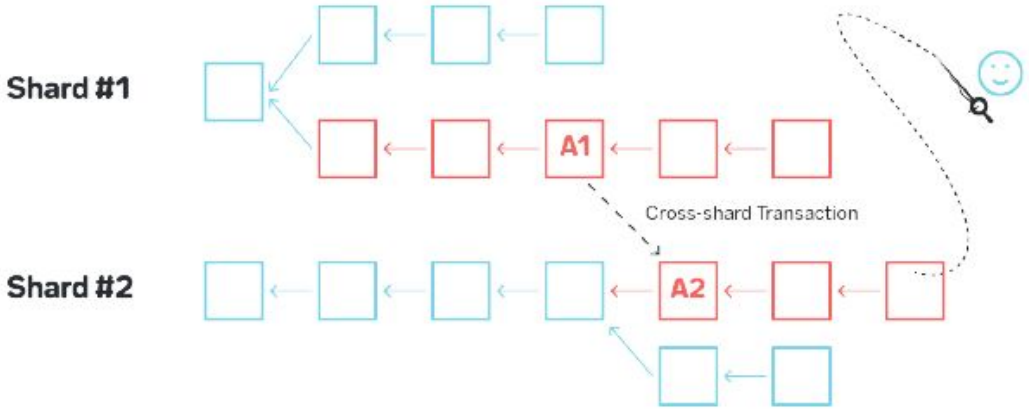
```rust
pub struct ChunkProofs {
    /// Encoded block header that contains invalid chunk.
    pub block_header: Vec<u8>,
    /// Merkle proof of inclusion of this chunk.
    pub merkle_proof: MerklePath,
    /// Invalid chunk in an encoded form or in a decoded form.
    pub chunk: MaybeEncodedShardChunk,
}
```

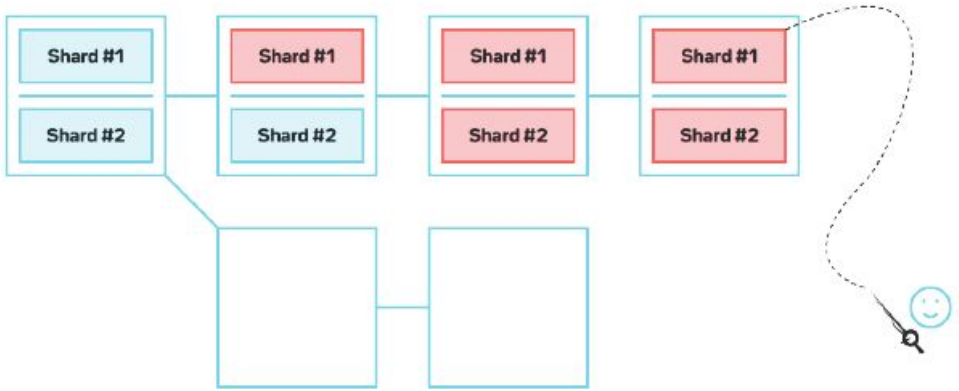
Challenge Period

Cross-shard tx, which is for the destination shard, don’t wait for the challenge period, they apply the receipt immediately, but then roll back if found invalid chunk

About 20 blocks period, fisherman can submit a challenge




Shard #1
A1
Cross-shard Transaction
Shard #2
A2


Shard #1
Shard #2
Shard #1
Shard #2
Shard #1
Shard #2
Shard #1
Shard #2

Slash

1. **Double Signing**: Signing two or more different blocks at the same height
2. **Invalid Chunks**: Signing a chunk with an invalid data or computational result

Since double signing may happens in malicious validators or non-malicious validators, to balance the risk of accidental slashing, NEAR uses "***Progressive slashing***"

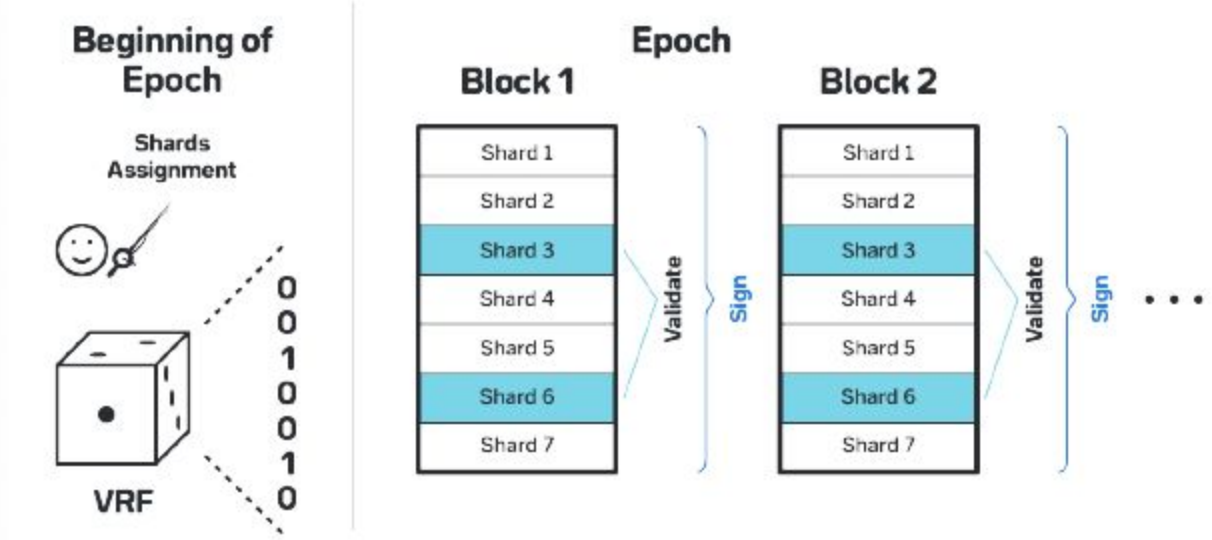
```
3 * malicious_stake / total_stake
```

E.g. A has 1 % of the total stake (total 50,000,000, A 500,000).
If A double signs and malicious actors < ⅓, A will lose 3% of his stake in that epoch --> 485,000 returned, 15,000 burned

For invalid chunk, the full stake of the validator gets slashed

Hidden Validator (WIP)

Since the shards assignment is concealed, HV signs on the full block



Doomslug + NFG Economics

Outline

- Doomslug (paper)
- Nightshade Finality Gadget (paper)
- Doomslug (in practice)
- Economics

Doomslug

Block can be produced in **1 round** as long as > 50% **honest** BP online

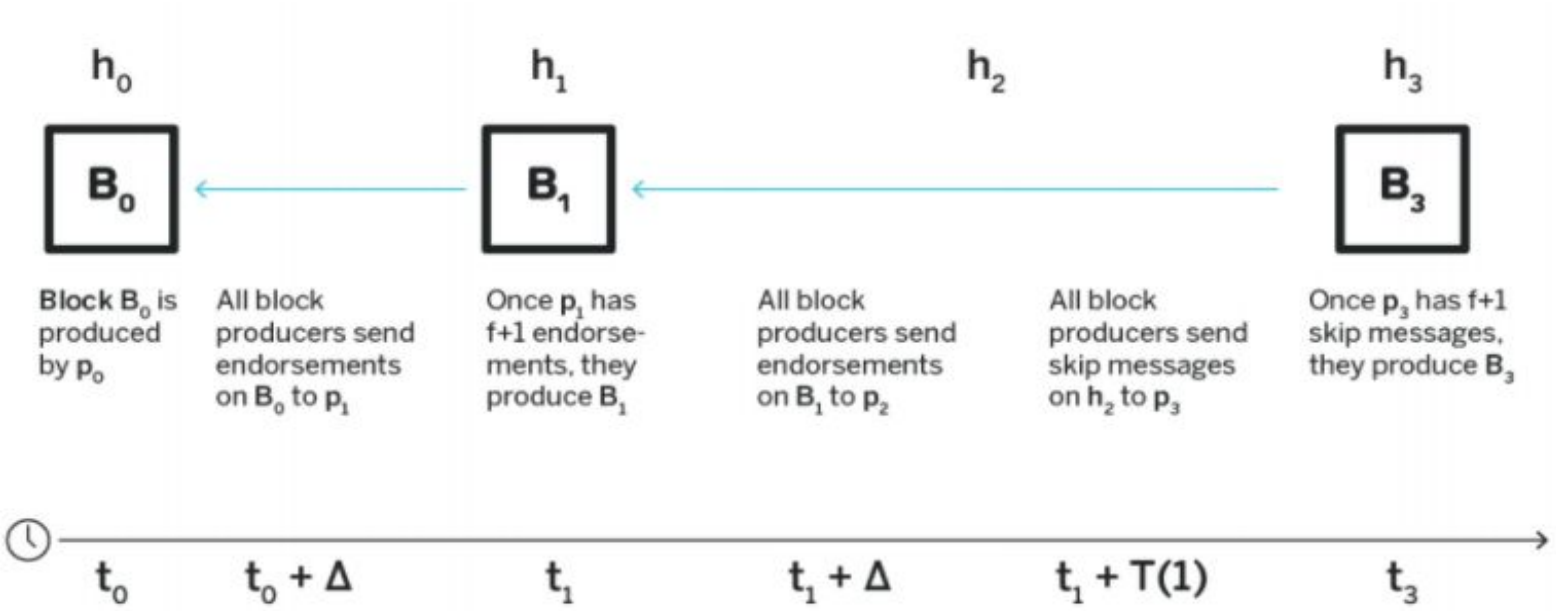
V_i sends **only one** of approvals for Block(h):

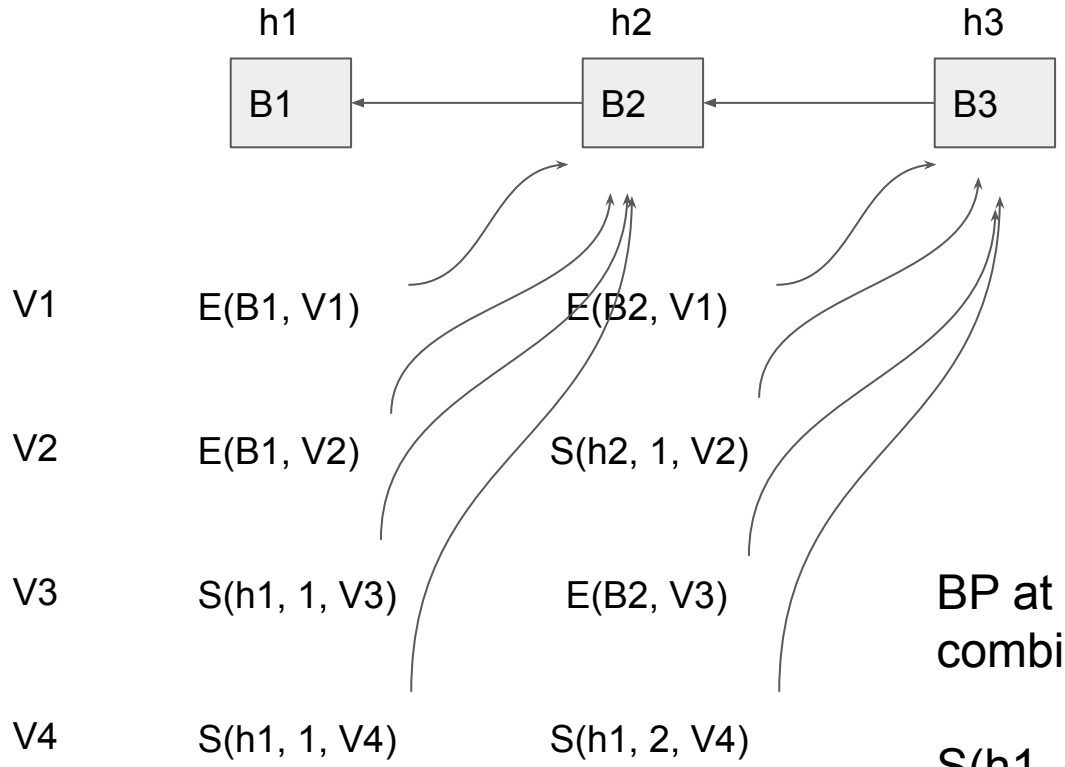
- $E(B,v_i)$: Endorsement (if received block)
- $S(h, n, v_i)$: Skip *n* blocks starting at height *h*

partially synchronous

BP(h+1) collects of *E* or *S* from > 50% V_i and produce Block(h+1)

If Block(h+1) includes > 50% $E(B_h,V_i)$, Block(h) has doomslug finality


h0
h1
h2
h3
B0
B1
B3
Block B0 is produced by p0
All block producers send endorsements on B0 to p1
Once p1 has f+1 endorse-ments, they produce B1
All block producers send endorsements on B1 to p2
All block producers send skip messages on h2 to p3
Once p3 has f+1 skip messages, they produce B3
t0
t0 + Δ
t1
t1 + Δ
t1 + T(1)
t3



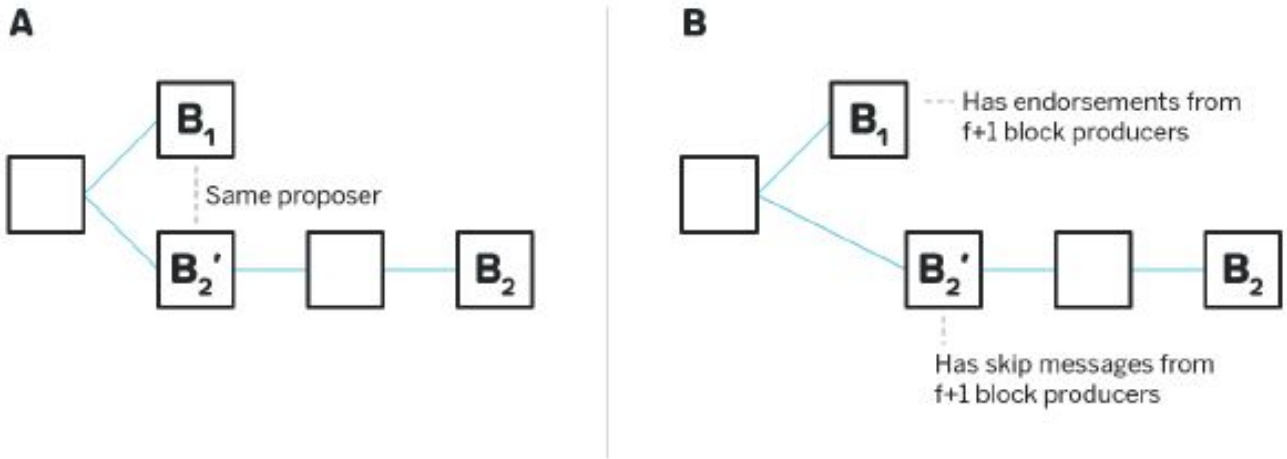
BP at height h3 must collect any combination of

$S(h1, 2, BP_i)$, $S(h2, 1, BP_i)$, and $E(B2, BP_i)$ from > 50% BPs

Safety A produced block can't be reverted unless at least one BP is slashed

1. conflicting endorsement:
   Same BP doesn't propose two blocks for the same height

2. conflicting skip and endorsement:
   A BP can never produce both skip or endorsement for same height



Liveness A block will be produced in finite time as long as > 50% honest BP online

When BP(h) produces the block without > 50% endorsements, BP(h) would wait for $T(h - h_{final}) / 2$ (4Δ) between they first received an E or S for height *h* from an honest V_i until BP(h) actually produced the block.

1Δ: all honest V skip or endorse h - 2 to BP(h-1)
2Δ: BP(h-1) collected > 50% messages, produced block at h-1
3Δ: all honest V saw block h-1, and immediately endorse it
4Δ: BP(h) collected endorsement for block at h-1

Therefore, any message sent to BP(h) by any honest V_i was an endorsement, and the **Block** at height h - 1 **has doomslug finality in finite time**.

partial sync network:

$T(h-h_{final}) >= 8\Delta$

Nightshade Finality Gadget (NFG)

Block can be finalized in **2 rounds** communication as long as >⅔ **honest** BP online

Straight line voting:

partially synchronous

1st round: B' collects ***Approval*** from ⅔ honest BP, ***quorum pre-vote*** B

2nd round: B'' collects ***Approval*** from ⅔ honest BP, ***quorum pre-commit*** B

https://www.youtube.com/watch?v=fU7hLt8nJUk&list=PL9tzQn_TEuFW_t9QDzlQJZpEQnhcZte2y&index=7

Approval: $<v_i\ p_i\ r>$

v: Block Producer

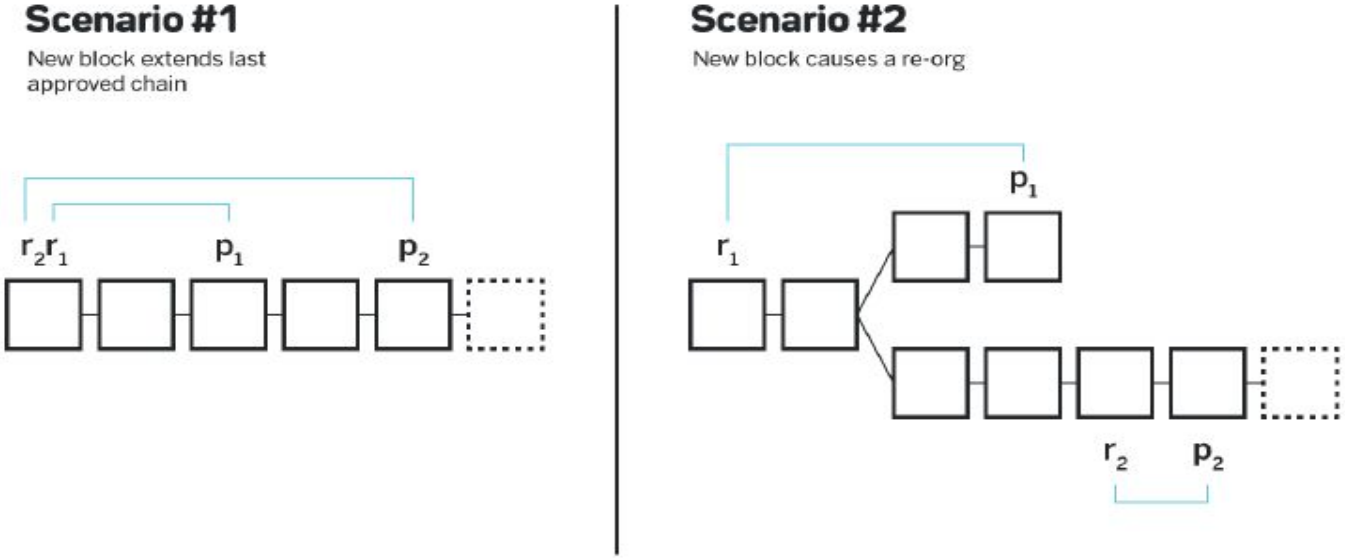
p: block is being approved

r: reference block

Same reference block → Same chain

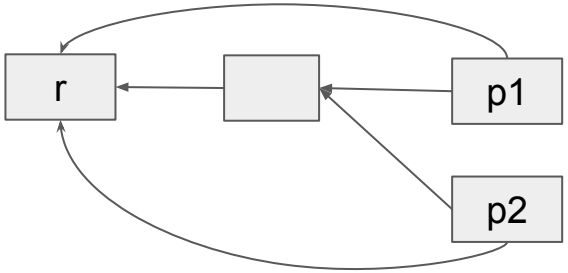
Different reference block → approval not intersect

Approval is produced on **doomslug** block and **canonical** chain

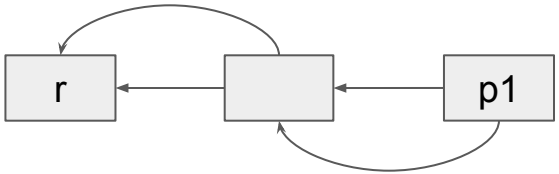


Slashable Approval

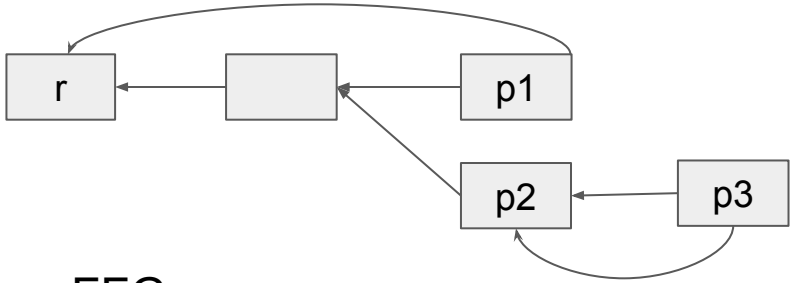
1. different chain, same r



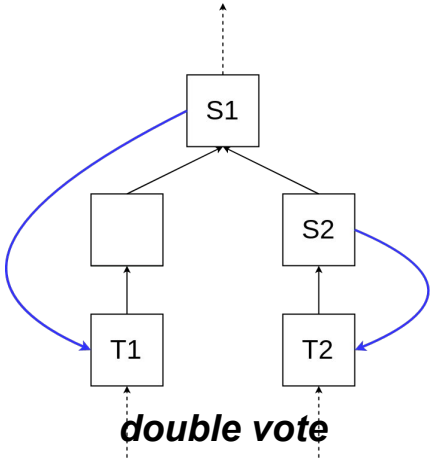
2. same chain, different r



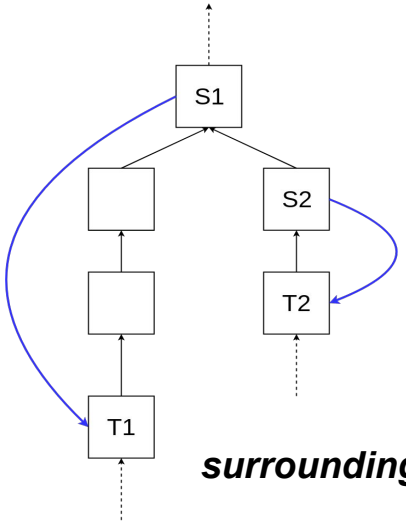
3. different chain, intersect approval



Casper FFG



double vote



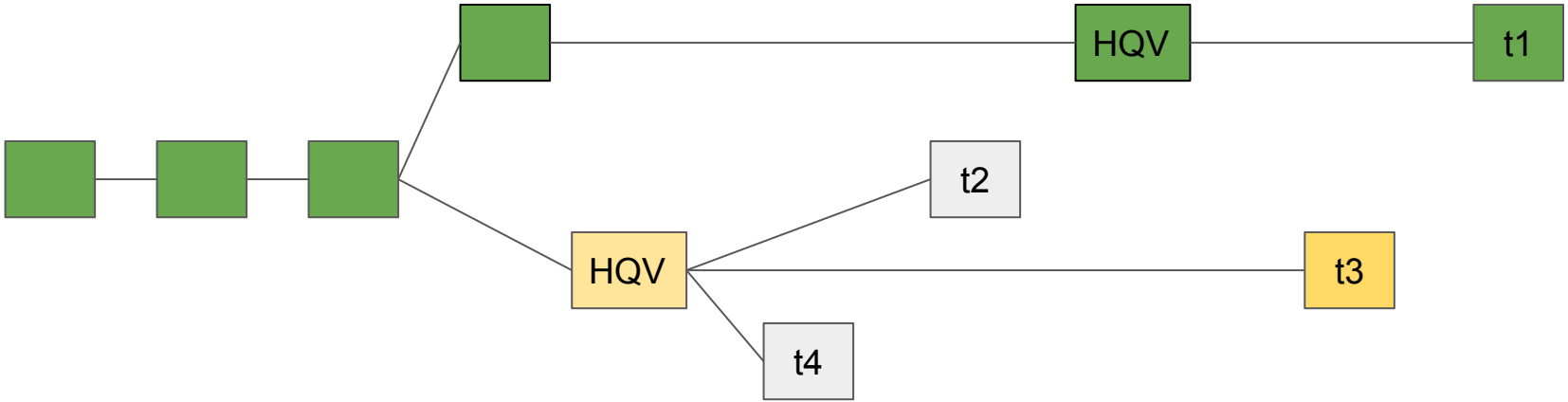
surrounding vote

NFG (*Fork Choice Rule*)

Block weight w: *w(B)* = height of B

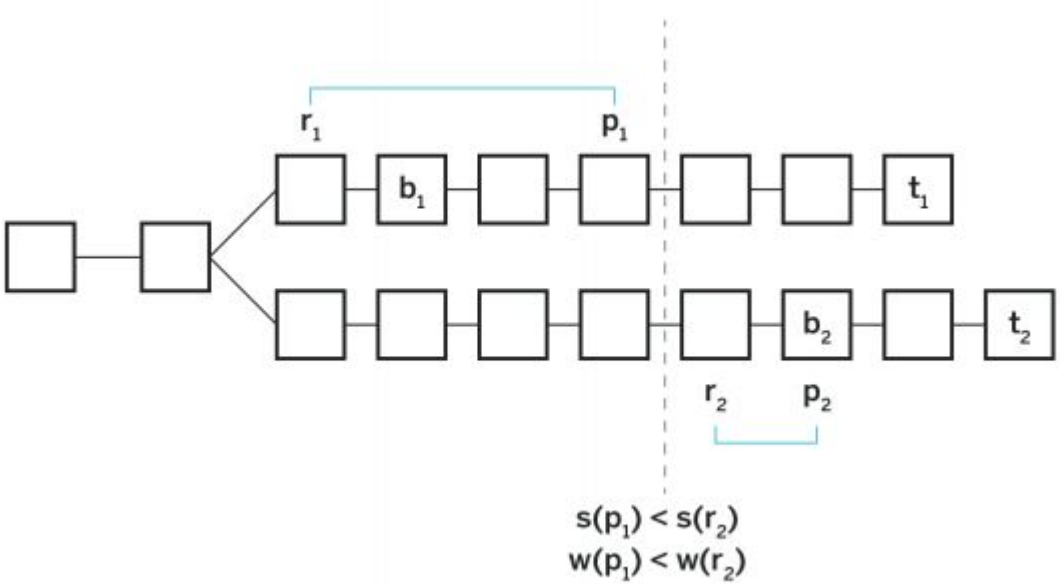
Block score s: *w(HQV(B))*, *HQV(B)* is the heaviest block that has pre-vote before B

Canonical chain: chain with the higher *s(tip)* or higher *w(tip)*



Safety

If a block b1 is finalized in a chain with a tip t1, no block b2 such that w(b2) > w(b1) can have a quorum pre-vote by the same BP set in a chain with some tip t2



Proof: b2 can have quorum pre-vote

B1 finalized, s(p1) >= w(b1)

∵ w(p1) < w(r2), s(p1) < s(r2)
∴ s(r2) >= w(b1)

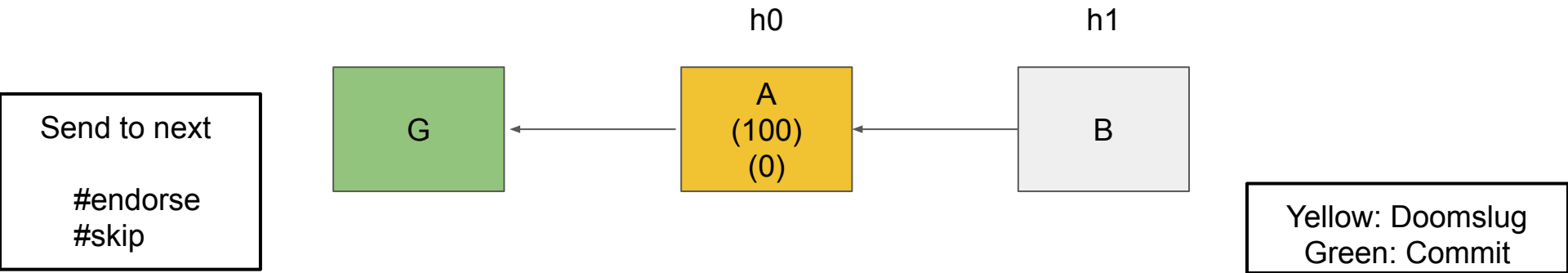
This means there is b2' with weight >= w(b1) ❌

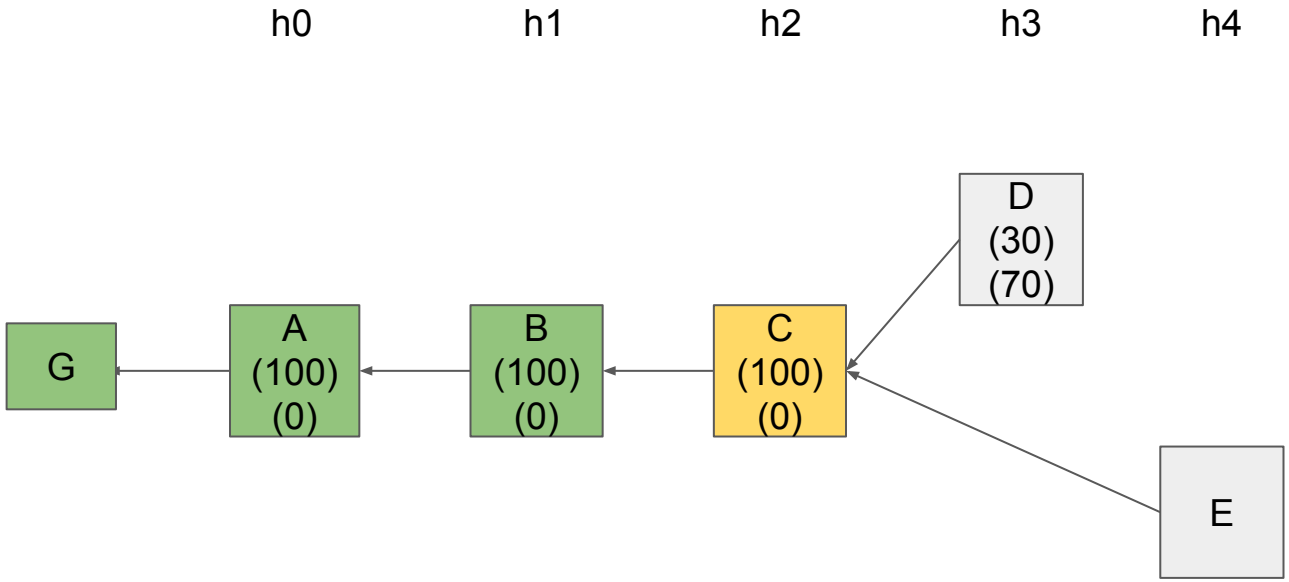
Doomslug in practice

Block includes **>** ⅔ approvals' signatures (endorsement, or skip) from other V_i

Protocol finality

A block is final if there are two blocks built on top of it and these three blocks have consecutive heights.

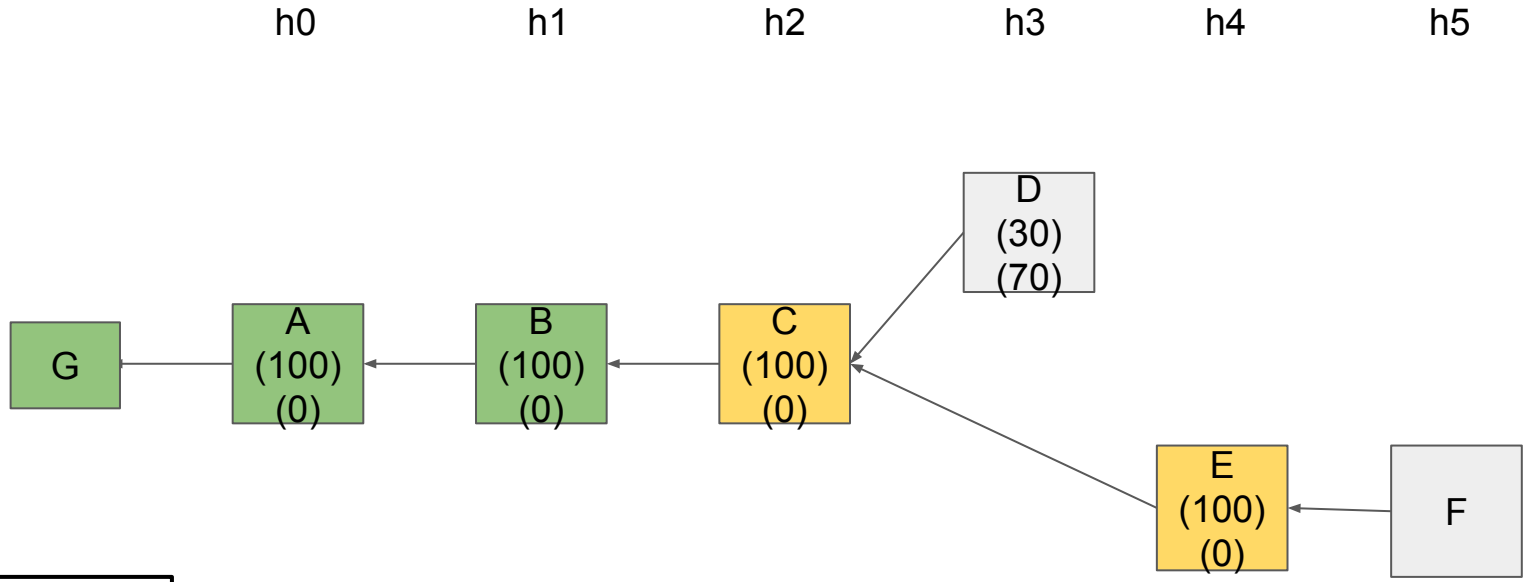




Send to next

#endorse
#skip

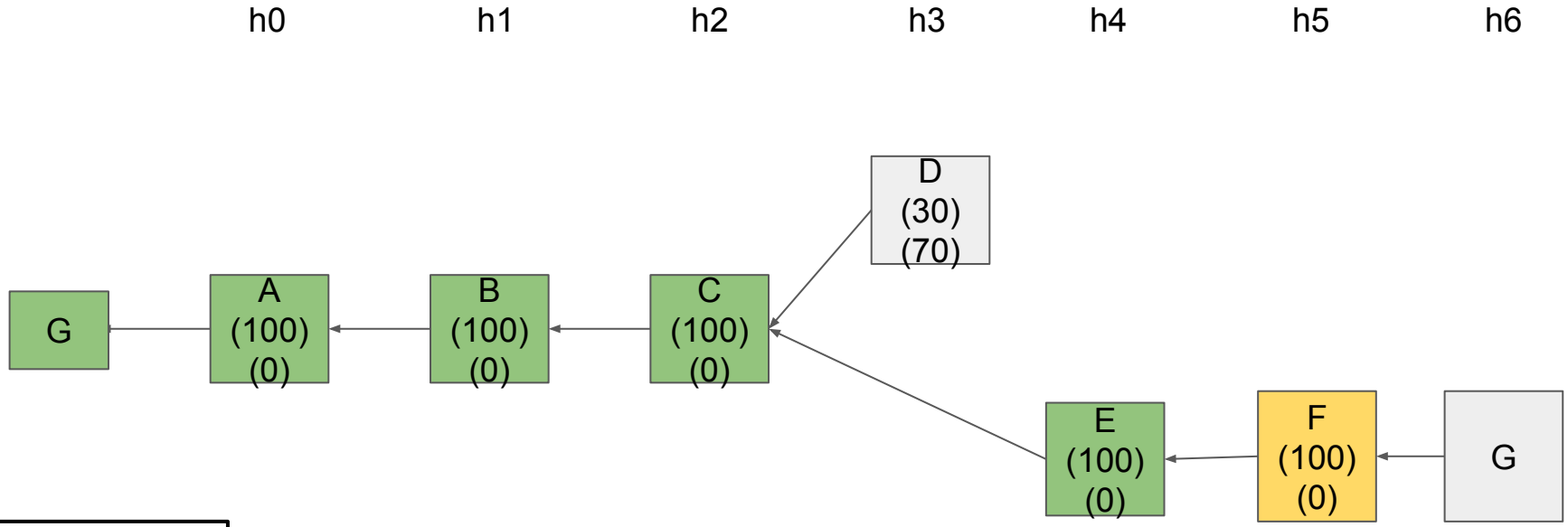
Yellow: Doomslug
Green: Commit



Send to next

#endorse
#skip

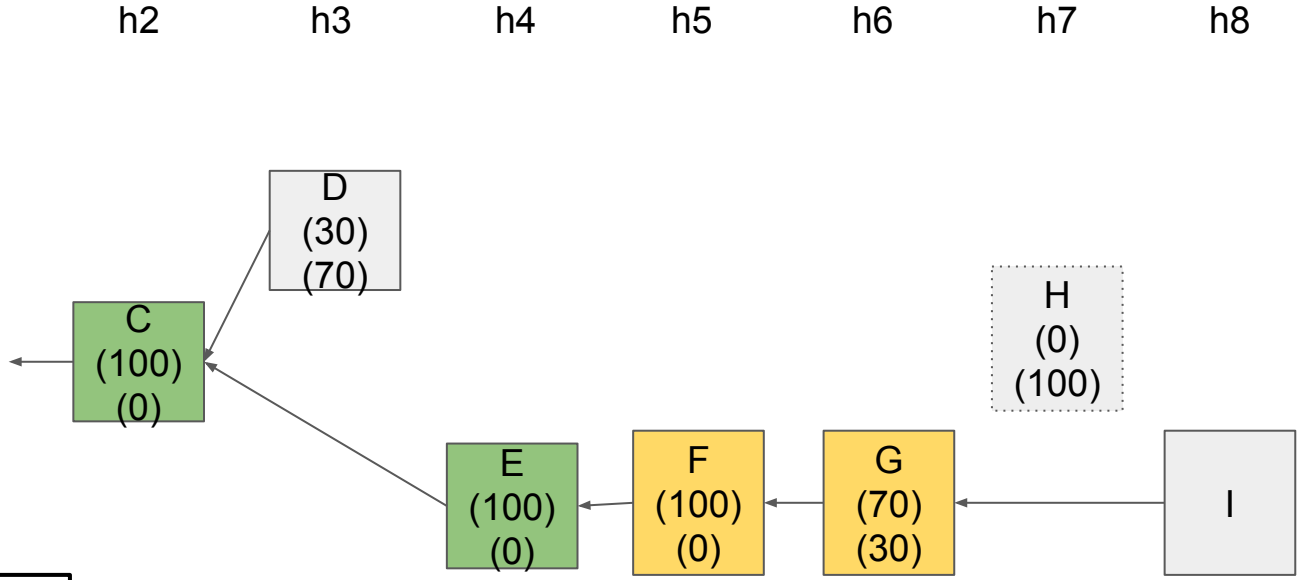
Yellow: Doomslug
Green: Commit



Send to next

#endorse
#skip

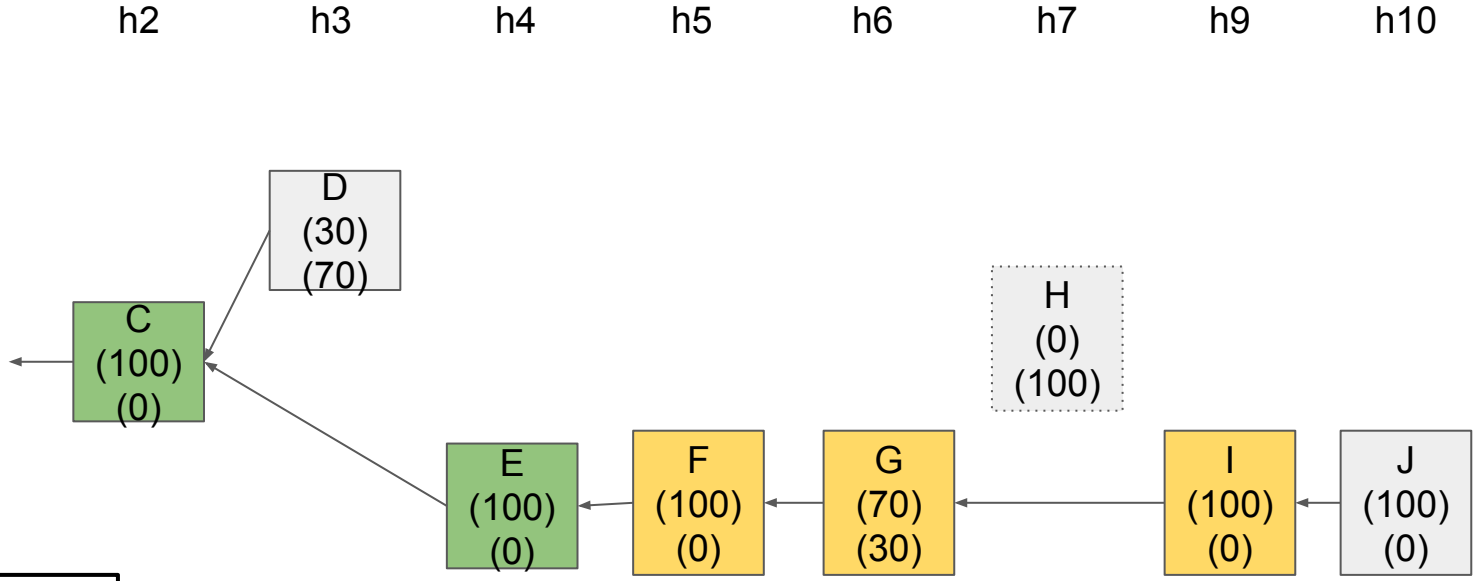
Yellow: Doomslug
Green: Commit



Send to next

#endorse
#skip

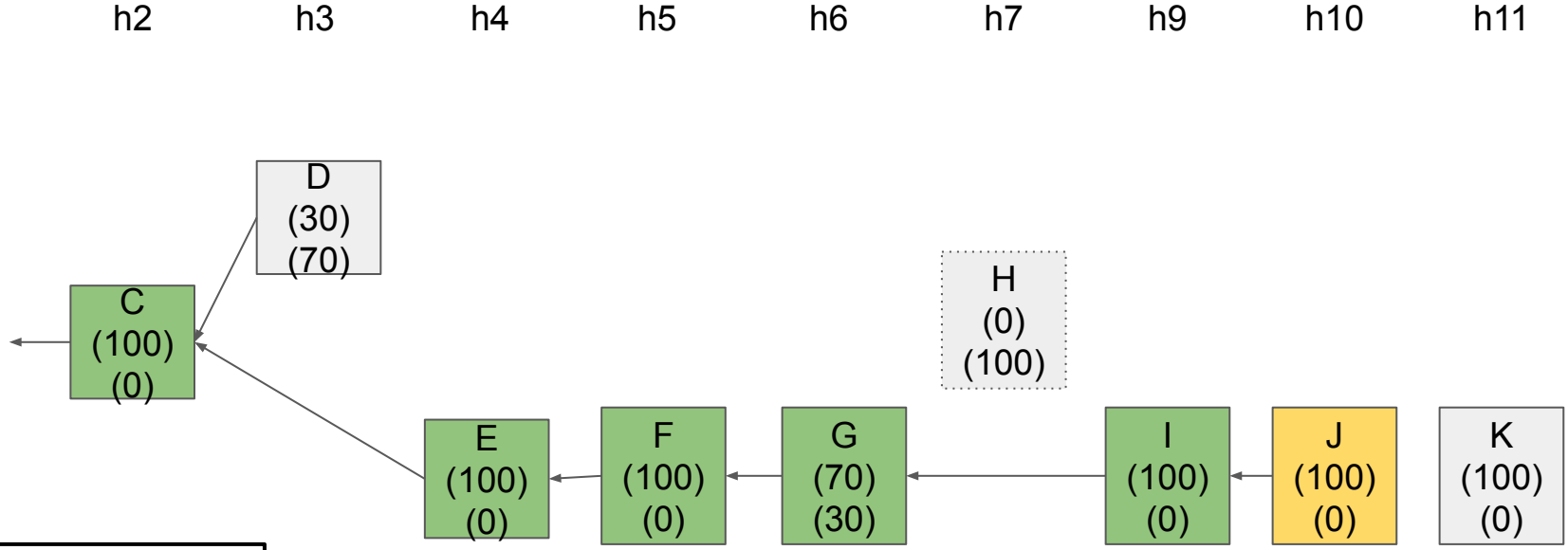
Yellow: Doomslug
Green: Commit



Send to next

#endorse
#skip

Yellow: Doomslug
Green: Commit



Send to next

#endorse
#skip

Yellow: Doomslug
Green: Commit

Economics

Dynamic sharded blockchain allows to maintain a network which almost never has a capacity issues by re-balancing the load based on changing circumstances.

Because of re-sharding, an application (or account) is not defined by the shard it is located in.

Allows all charges to be priced the same, removing the price distinction between crossshard and intra-shard transactions.

Rewards

Rewards is on epoch level, at the end of every epoch, the rewards are then distributed between Validators, Developers and the Protocol Treasury.

$$epochReward_t = coinbaseReward_t + epochFees_t$$

epochFees: stateFee + txFee

90% *epochReward* is used as *totalValidatorReward*

10% *ecpochReward* give to Protocol Treasury

To minimize coinbase, sets a ceiling for the maximum coinbase and dynamically decreases the coinbase depending on the amount of total fees in the system. This ensures a minimum epoch reward, and with growth in usage, reduces inflation.

$$coinbaseReward_t = \begin{cases} 0 & epochFees_t \geq maxConibase \\ maxCoinbase - epochFees_t & otherwise \end{cases}$$

$$epochRewards = \begin{cases} epochFees_t & epochFees_t >= maxCoinbase \\ maxCoinbase & \text{otherwise} \end{cases}$$

Burns txFee

avg # of tx/day	Min Ⓝ in fees/day	Ⓝ mint/day	Annual inflation
1,000	0.1	136,986	5.000%
10,000	1	136,985	5.000%
100,000	10	136,976	5.000%
1,000,000	100	136,886	4.996%
10,000,000	1,000	135,986	4.964%
100,000,000	10,000	126,986	4.635%
1,000,000,000	100,000	36,986	1.350%
1,500,000,000	150,000	-13,014	-0.475%
2,000,000,000	200,000	-63,014	-2.300%

Transaction Fee

$$gasFee = gasFee \times (1 + (\frac{gasUsed}{gasLimit} - \frac{1}{2}) \times adjFee)$$

Predict gasFee:

– *gasLimitindex*, the maximum amount of gas that is allowed in each shard at that index

– *gasUsedindex*, shard, the amount of gas actually used in each shard at that index

- adding a strict ***expiration*** (TTL) on transactions to avoid accumulated txs
- define ***minGasPrice*** to avoid low fee

gasPrice is universal across all shards, if a tx that touches multiple shards, the price is known ahead of time and can easily be calculated. (still needs ***dynamic resharding*** to solve ***imbalance problem*** of different shard)

Operation	TGas	fee (mN)	fee (Ⓝ)
Create Account	0.42	0.042	4.2×10^{-5}
Send Funds	0.45	0.045	4.5×10^{-5}
Stake	0.50	0.050	5.0×10^{-5}
Add Full Access Key	0.42	0.042	4.2×10^{-5}
Delete Key	0.41	0.041	4.1×10^{-5}

```
near call myContract.testnet myFunction "{ \"arg1\": \"val1\" }" --gas=300000000000000
```

Storage Fee

Account Data

- Balance
- Locked balance (for staking)
- Code of the contract
- Key-value storage of the contract. Stored in a ordered trie
- Access Keys
- Postponed ActionReceipts
- Received DataReceipts

each block time each account is charged *StoragePrice* × *SizeOf(account)* tokens.

minBalance(non-staking): *pokeThreshold* × *storagePrice* × *SizeOf(account)*

minBalance(staking): 4 × *epochLength* × *storagePrice* × *SizeOf(account)*

If account's balance goes below ***minBalance*** anyone can send a special transaction that clears state from this account. As a reward the steward gets some pokeReward of the remaining balance on the account.

update the accounts only when they are already changed by some transaction:

- Each account has a *StoragePaidAt* field.

- Current balance is then calculated
  a. *balance − StoragePrice × SizeOf(account) × (curBlock−StoragePaidAt)*.

- When account is modified, we recompute the size of the state and update *balance* given formula above, setting *StoragePaidAt* at the current block.