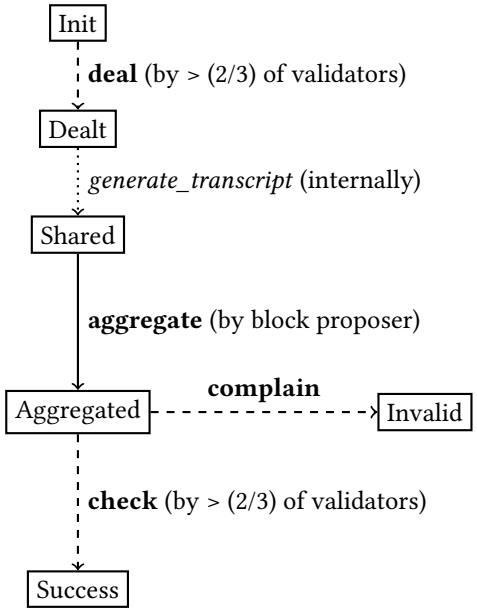
Spec Diagrams

HELIAX TEAM

Contents

Author's address: Heliax Team.

1 DKG



1.1 Start

The protocol for distributing signing keys (DKG) runs once per epoch, which means roughly once per day in our case.

The starting phase is triggered by the block proposer. The *start* message will contain the new epoch and a list of validator public keys together with their respective stake weights.

```
fn start(
    pk: PublicKey,
    epoch: Timestamp,
    stakes: Vec<(ValidatorPublicKey, ValidatorStakePercentage)>)
```

This phase is computationally expensive, since the block proposer needs to gather the current staking weights. Furthermore, we want to delay the creation of the new instance of the DKG as long as possible to minimise the divergence of the snapshot taken by the block proposer from the actual state by the time the new instance is used. This constraint implies that (at least) two instances of the DKG will run in parallel and that the new block proposer will be different from the existing one, as it needs to do some extra work during the existing DKG instance.

Before the end of the current epoch, we need to run the DKG for the next epoch. We'll run each DKG in a separate process.

Corner cases:

(1) *Fallback*: What if the DKG is not completed before the expiration of the current epoch?
(2) *Stale keys*: What if the public keys used to encrypt transactions existing in the mempool become no longer valid once a new instance of the DKG starts?

Note: The block proposer is chosen by Tendermint (?).

1.2 Announce

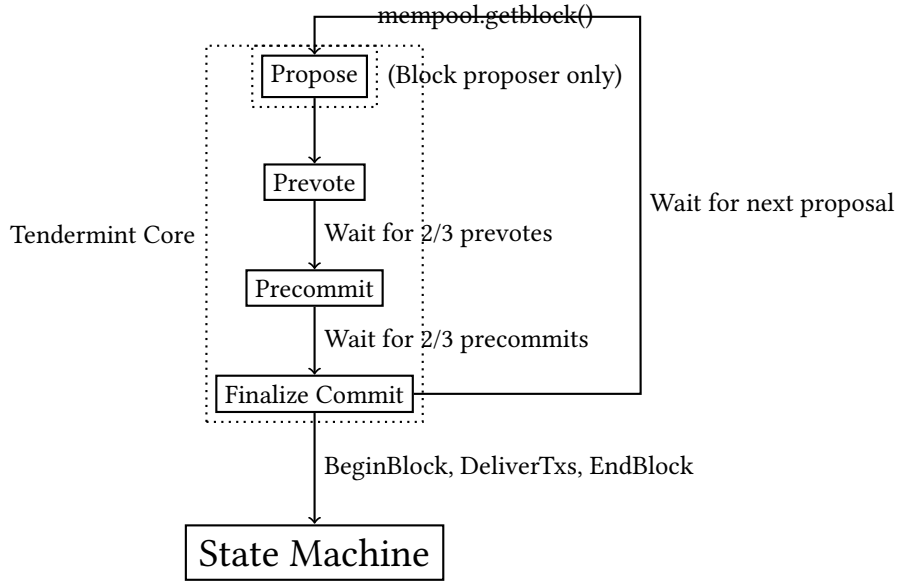
Once the DKG protocol is *Started*, every validator will **announce** their encryption key. This is the only stage in which validators intervene.

```
fn announce(pk: PublicKey)
```

1.3 Aggregate

Only one node, the block proposer, is necessary in this phase. Validators will check the aggregation was done correctly

2 ABCI



3 ABCI++