Tendermint

在具有拜占庭错误的分布式网络中保证交易的顺序一致

主要内容

- Tendermint 组成
- 术语简介
- 共识
- 锁机制
- ABCI Application

Tendermint 组成

- Consensus Engine(Tendermint Core)
  保证交易在善意节点上以同样的顺序被打包

- Application Blockchain Interface(ABCI)
  可以用多种语言编写应用，处理交易的逻辑均交给上层应用

安全性：可以抵抗小于1/3恶意节点的任何行为（如 宕机、攻击等）
一致性: 所有非恶意节点提交交易顺序一致

术语

Validator

- 公钥标识，轮流提出新区快,并对提案进行投票
- 所有validator起始于共同的初始状态（包含一个有序列表），propose和vote均需要利用其私钥签名，可在出问题之后查找保存在日志中的签名情况寻找恶意节点
- 超时未收到提案或无效提案可以发起投票跳过本轮的提案者
- 1/3或更多validator未响应，网络停止

Polka

收到超过2/3的validator 都 “**pre-vote** for a single block at a given round”

Mem-pool

交易先被提交到Mem Cache中，利用application logic验证交易，如果有效，提交到MemPool中，并在网络中gossip（ordered multicast algorithm），交易会按照在此节点的处理顺序发送到其他对等节点。

Proposer从Mempool中提取有序的交易列表，打包到区块中，区块提交后，包含在本区块的交易从MemPool中移除，池中其他交易会被application logic 重新验证。

Consensus Engine

- **proposal**

新区块由本轮正确的提案者提出，并传播到其他validator，在给定时钟内没有收到有效提案，则提案发起者可以被投票跳过

- **Votes**（两阶段投票）

1.*pre-vote：*

Pre-vote for block：收到完整有效区块，对这个区块提案签名pre-vote ,并广播自己的投票

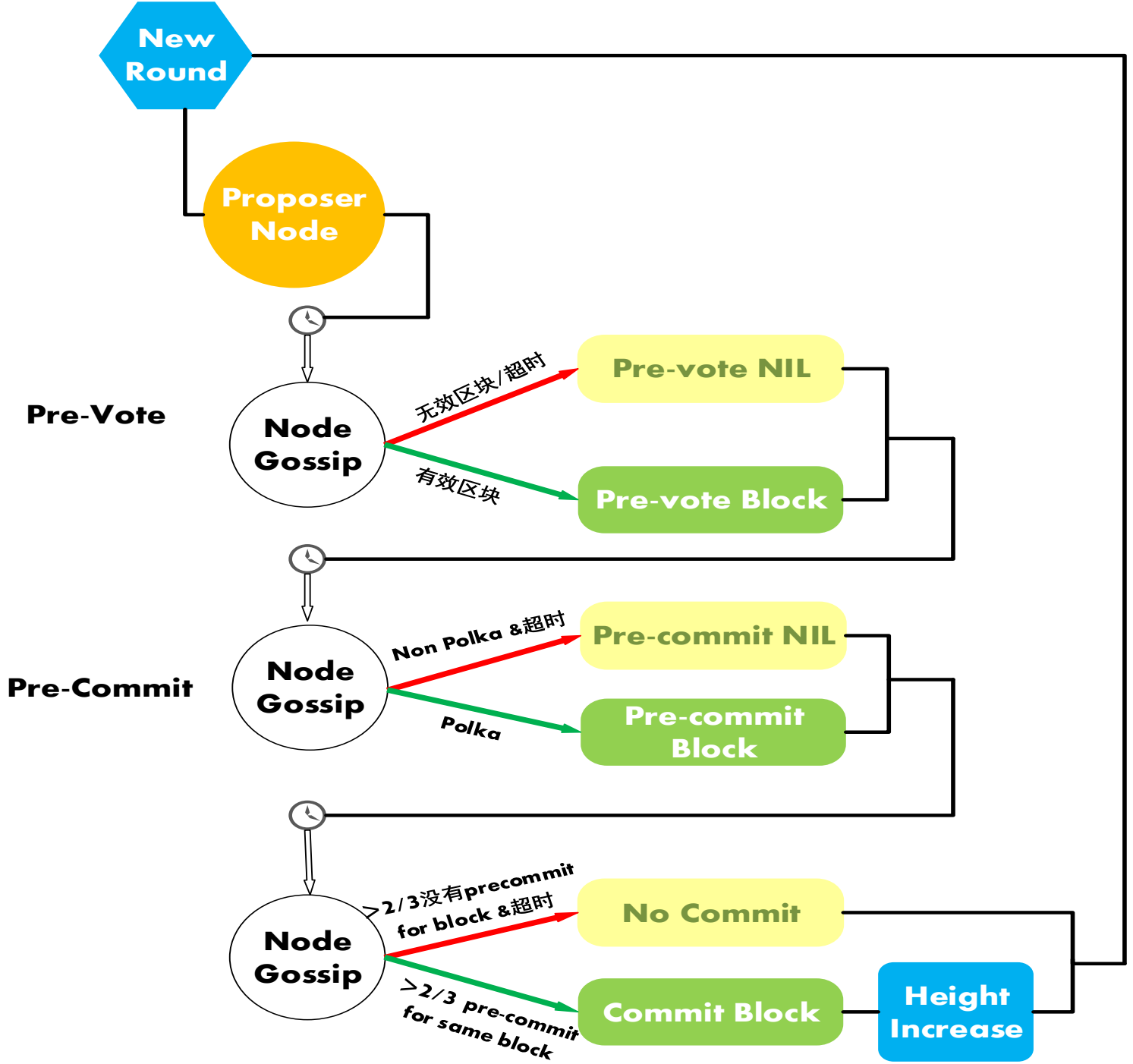
pre-vote for nil 时钟超时或者区块无效（如收到的是之前轮的区块、非本轮提议者的提案）

2.*pre-commit：*

pre-commit for block：收到超过2/3的pre-vote for same block，验证pre-vote消息的签名，都没有问题后，签名pre-commit for a block消息，并广播

pre-commit for block：未收到polka，且时钟超时，则略过验证签名的过程，直接签名pre-commit for nil

- **Lock(详见后面)**


New Round
Proposer Node
Pre-Vote
Node Gossip
无效区块/超时
有效区块
Pre-vote NIL
Pre-vote Block
Pre-Commit
Node Gossip
Non Polka &超时
Polka
Pre-commit NIL
Pre-commit Block
Node Gossip
>2/3没有precommit for block &超时
>2/3 pre-commit for same block
No Commit
Commit Block
Height Increase

锁机制

- Prevote-the-Lock

一个validator收到对同一个区块超过2/3的**pre-vote** 时，这个validator的预投票和预提交被**锁定**为这个区块。之后的轮里不可以再推翻这个预提交,后续轮里，必须为这个锁定的区块进行pre-vote，若这个validator是本轮的提案发起者，则要propose锁定的这个区块

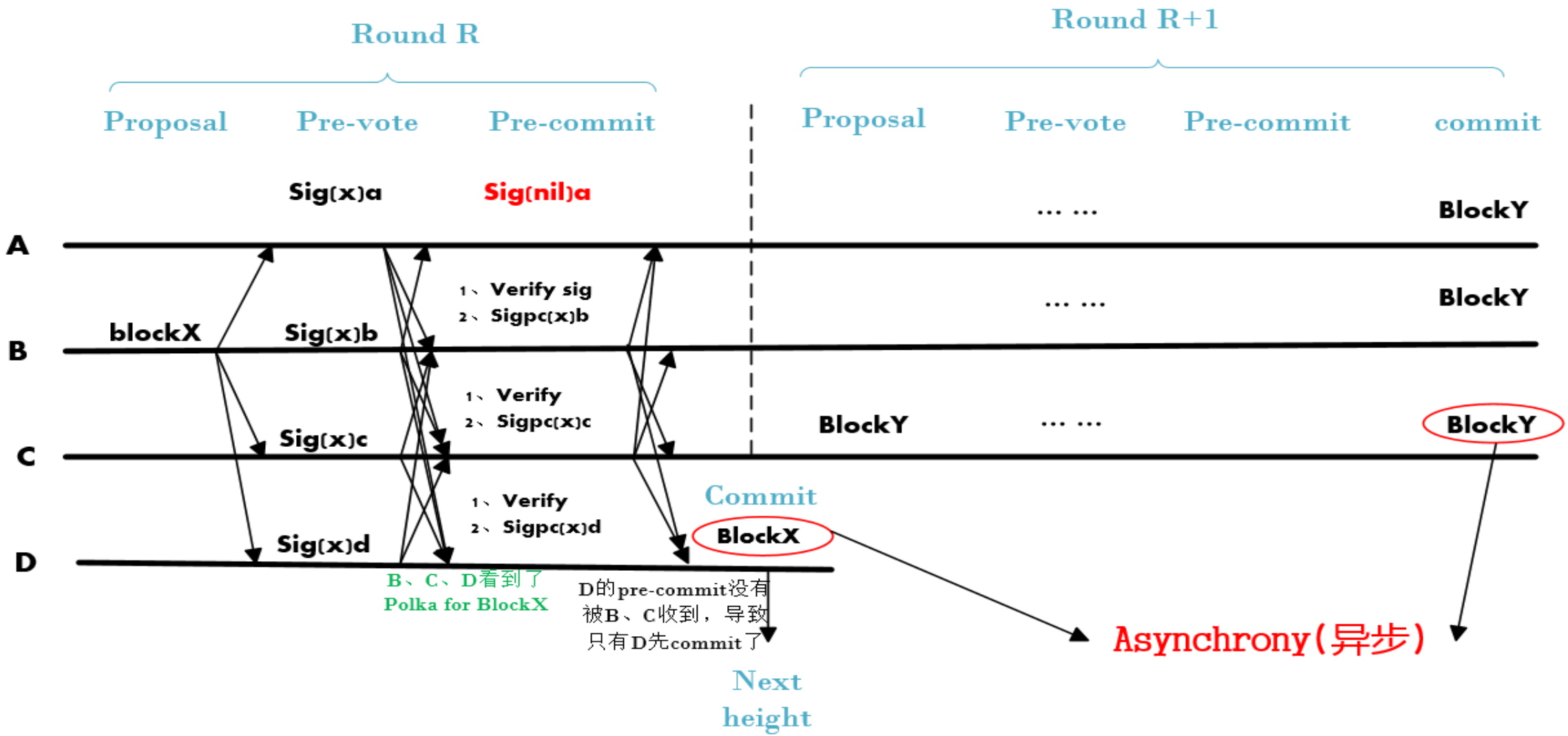
- Unlock-on-polka

当在后续的轮里收到对新区块（与锁定的区块不同）的polka时，允许validators解开之前的锁。

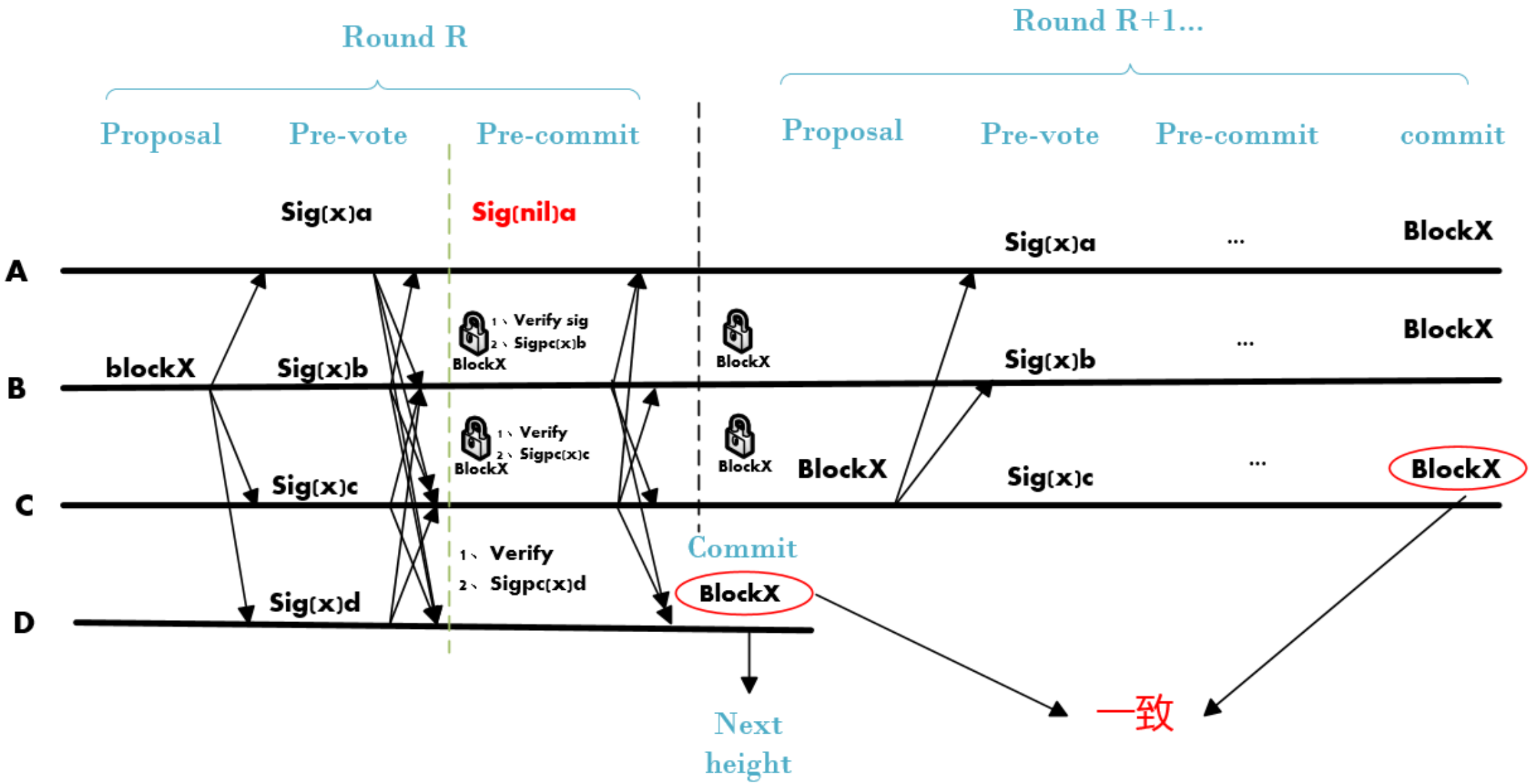
仅有预提交锁是不够的，必须有解锁能力，否则会损失系统的liveness

只有收到polka后，才能pre-commit for new height

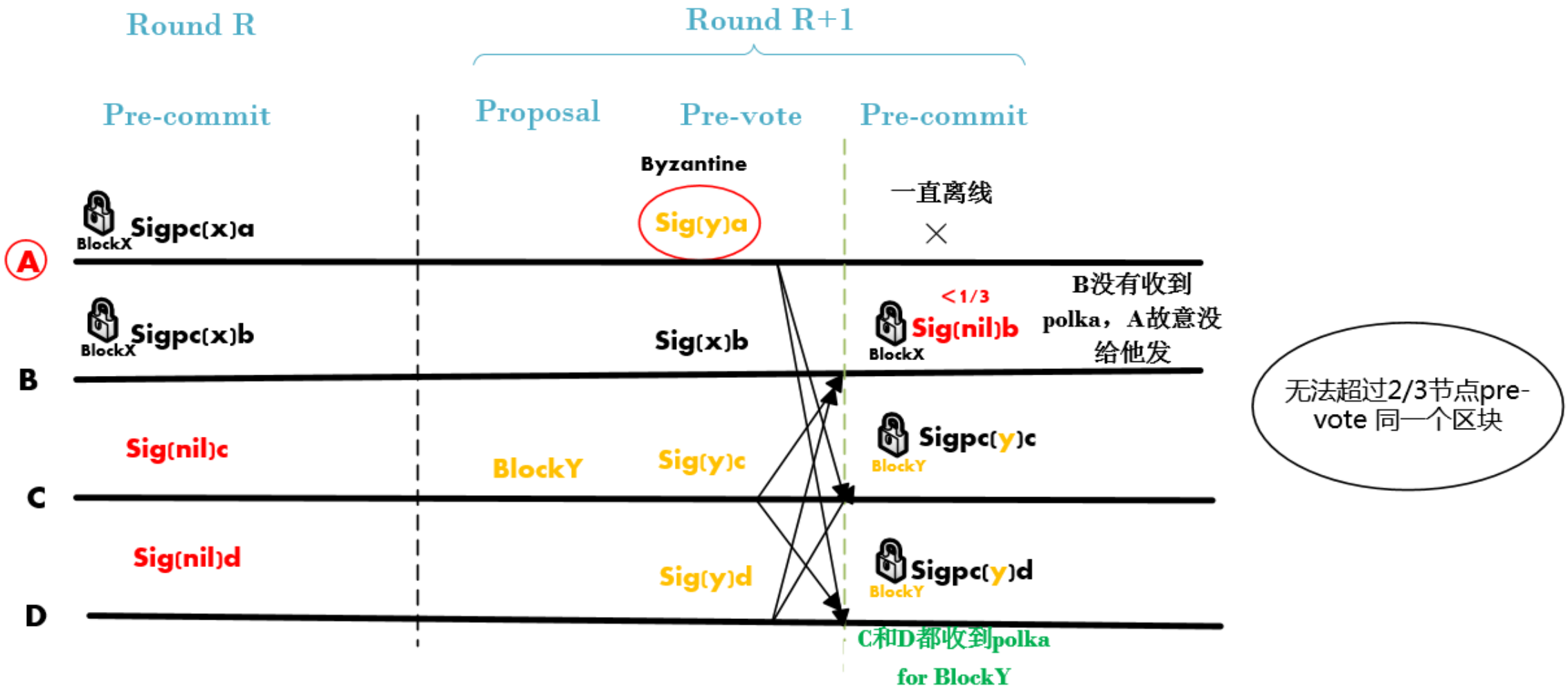
无锁机制时


Round R
Round R+1
Proposal
Pre-vote
Pre-commit
Proposal
Pre-vote
Pre-commit
commit
Sig(x)a
Sig(nil)a
… …
BlockY
A
B
C
D
blockX
Sig(x)b
1、Verify sig
2、Sigpc(x)b
… …
BlockY
1、Verify
2、Sigpc(x)c
Sig(x)c
BlockY
… …
BlockY
Commit
1、Verify
2、Sigpc(x)d
Sig(x)d
BlockX
B、C、D看到了
Polka for BlockX
D的pre-commit没有
被B、C收到，导致
只有D先commit了
Asynchrony(异步)
Next
height

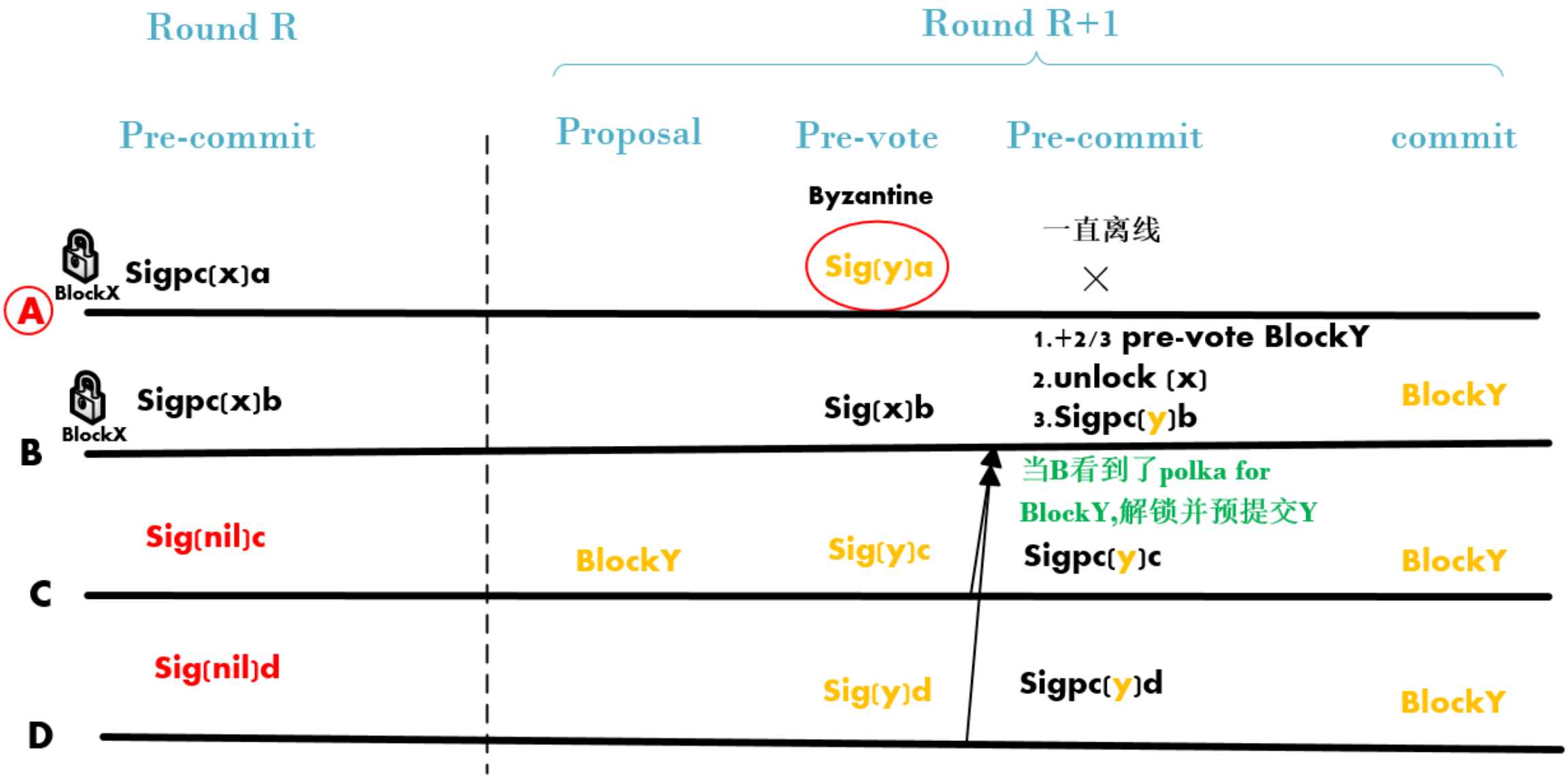
Prevote the lock



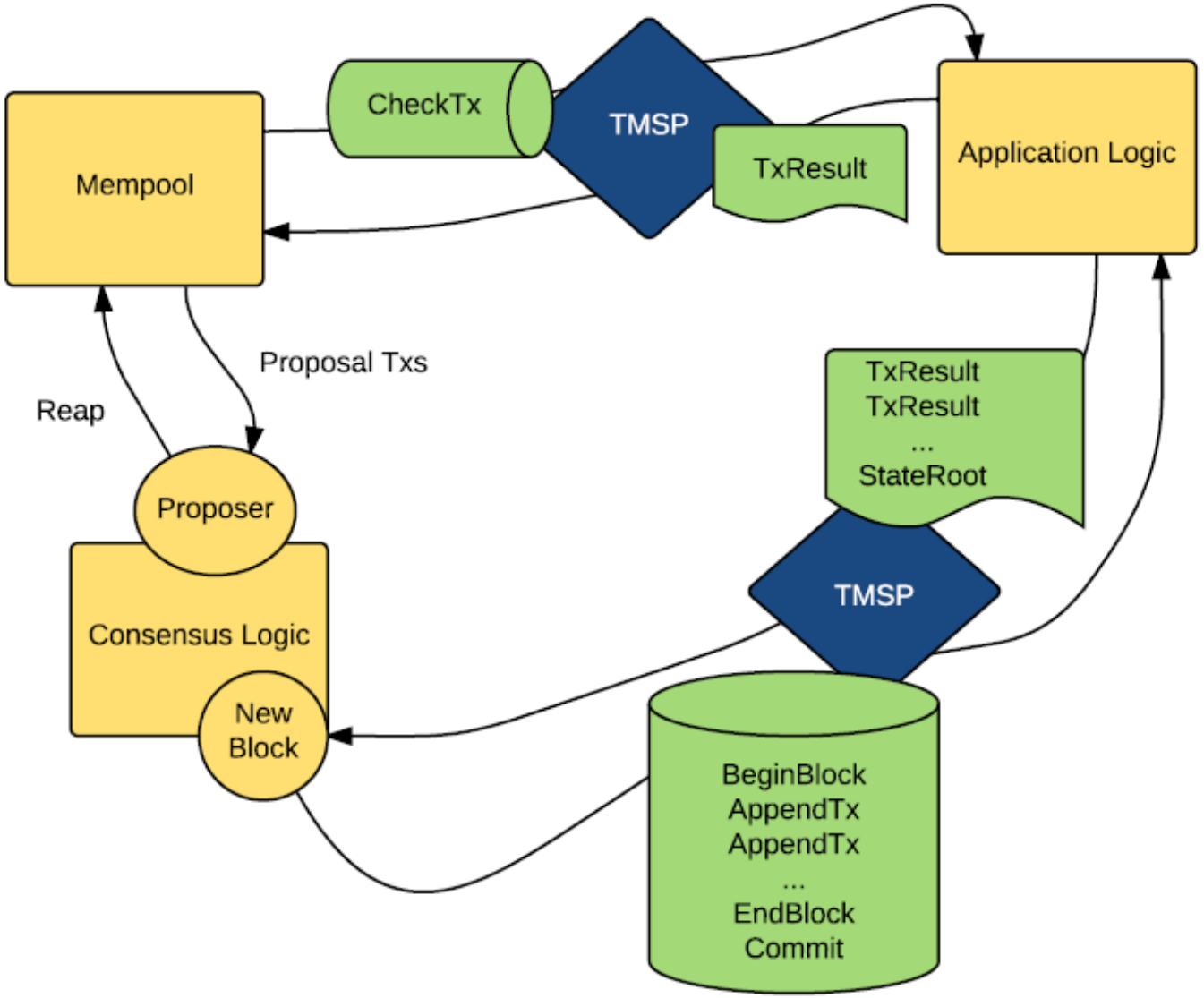
Without unlock-on-polka

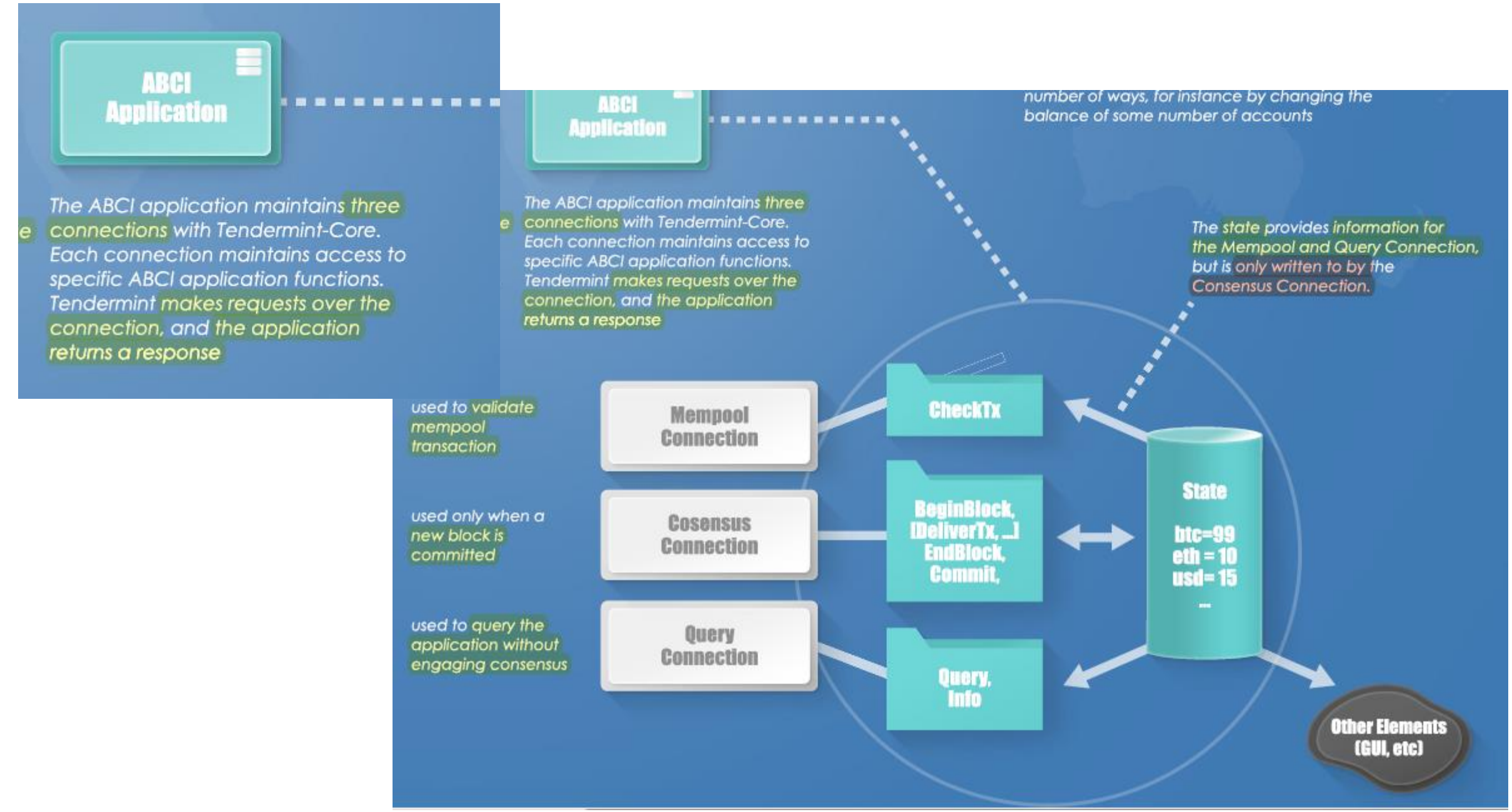


Unlock-on-Polka



Application Blockchain Interface(ABCI)




ABCI Application
The ABCI application maintains three connections with Tendermint-Core. Each connection maintains access to specific ABCI application functions. Tendermint makes requests over the connection, and the application returns a response
ABCI Application
The ABCI application maintains three connections with Tendermint-Core. Each connection maintains access to specific ABCI application functions. Tendermint makes requests over the connection, and the application returns a response
number of ways, for instance by changing the balance of some number of accounts
The state provides information for the Mempool and Query Connection, but is only written to by the Consensus Connection.
used to validate mempool transaction
Mempool Connection
CheckTx
used only when a new block is committed
Cosensus Connection
BeginBlock, [DeliverTx, ...] EndBlock, Commit,
State
btc=99
eth = 10
usd= 15
...
used to query the application without engaging consensus
Query Connection
Query, Info
Other Elements (GUI, etc)

区块结构

Block is composed of three parts:

- Block header
- List of transactions
- LastCommit

```
block := &Block{
    Header: &Header{
        ChainID:         chainID,
        Height:          height,
        Time:            time.Now(),
        NumTxs:          len(txs),   //transaction quantity
        LastBlockID:     prevBlockID,
        ValidatorsHash: valHash, //hash of the current Validators
        AppHash:         appHash, // state merkle root of txs from the previous block
                                   // represents the state of the actual application (not blockchain state)

    },
    LastCommit: commit,
    Data: &Data{
        Txs: txs,
    },
```

LastCommits

- 区块高度为H，第R轮Proposal的validator为A，区块中lastCommits字段存放的是高度为H-1时A节点收到超过2/3节点的Pre-commit 消息

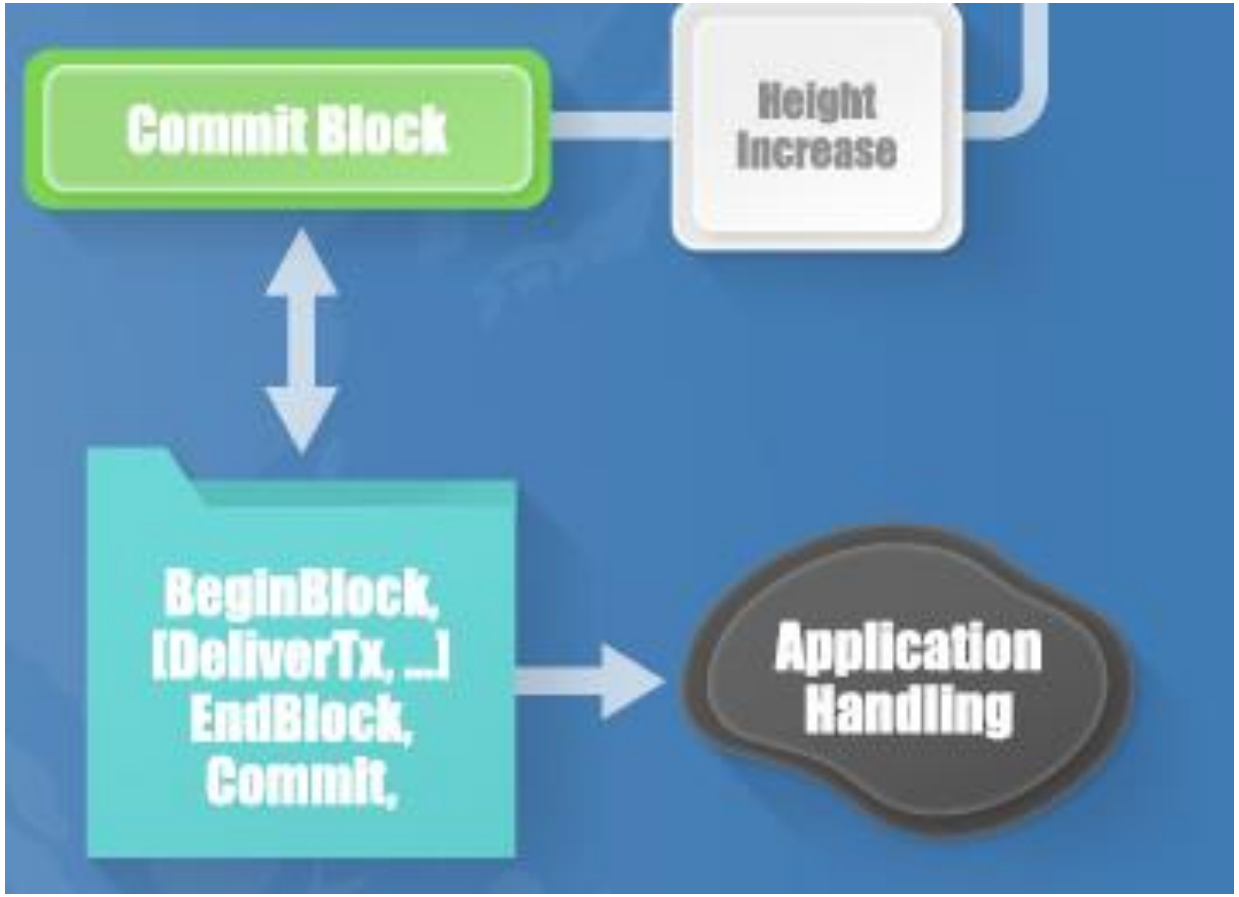
优点

- ✓ Easy to use
- ✓ Simple to understand
- ✓ Highly performance
- ✓ Useful for wide variety of distributed application
- ✓ Support validator set dynamic changes

验证者集合动态改变

Through the TMSP interface using the EndBlock message

- Request :若想要动态增删验证者节点，写在交易中，AppendTx时传给应用逻辑
- Response :应用程序将更新过后的验证节点列表以及权重放入对EndBlock 请求的响应消息中，返回给共识引擎，以对验证节点列表进行更新



优点：

不需要为增删节点设置独特消息类型和连接；不需要停止服务去增删节点

Problems----validator轮换机制

- Validator的发出提案的次数与其所占股权份额是否相关？大股东可以更多次的发出提案？设计可能的轮询机制

- 高度H，在第0轮即达成共识提交了新的区块，进入新的高度H+1，又是从第0轮开始Proposal，那么这个Proposer应该是谁？新的验证者节点 or 高度H时第0轮的验证者节点。

Tendermint形式化描述

$$Consensus := \prod_{i=1}^{N} PR_i^{0,\emptyset,\emptyset,}$$

每个节点共识的起始状态：r=0,proposal for round 0为空集
为此height区块所有round投票的投票集合为空集

$$PR_i^{r,p,v} := \text{if } i = proposer(r) \text{ then}$$
$$propose_i!(prop) \mid PV_i^{r,prop,v}, \text{ where } prop = chooseProposal(p)$$
$$\text{else if } p \neq \emptyset \text{ then}$$
$$PV_i^{r,p,v}$$
$$\text{else}$$
$$propose_{proposer(r)}?(prop).PV_i^{r,prop,v} + susp_{proposer(r)}.PV_i^{r,\emptyset,v}$$

Propose阶段：

1. If 节点i是leader：
   通过propose信道发送(!代表发送消息)prop，
   发送完毕后，进入Pre-vote阶段,其中将p=prop
2. Else if p不为空集，则进入PV阶段
3. Else i节点的propose信道准备接收第r轮的提案，
   若收到prop，则进入PV阶段。（.代表前一个动作做完接着做下一个动作）
   若超时，一直没接收到proposal，则pre-vote for nil

$$PV_i^{r,p,v} := prevote_i!(p) \mid (\nu\, c)(\prod_{j=1}^{n} prevote_j?(w).c!(prevote_j, w) \mid PV1_i^{r,p,v}(c))$$

PV阶段：i节点的prevote信道发送proposal，同时有一个并发进程，创建了信道c，每个节点都记录其他节点（包括自己）发送的prevote消息，然后通过c信道发送一个消息对（**prevotej,w**），将r、p、v和c信道作为参数传递给PV1进程

$$PV1_i^{r,p,v}(c) := \text{if } max_b(|\{w \in v_r^1 : w.block = b\}|) > \frac{2}{3}N \text{ then}$$

$$PC_i^{r,b,v}$$

$$\text{else if } |v_r^1| > \frac{2}{3}N \text{ then}$$

$$PC_i^{r,\emptyset,v}$$

$$\text{else}$$

$$c?(pv, vote).\ \text{if } vote.round < r \text{ then}$$

$$pv?(w).c!(pv, w) \mid PV1_i^{r,p,v}(c)$$

$$\text{else if } vote.round = r \text{ then}$$

$$PV1_i^{r,p,vote::v}(c)$$

$$\text{else}$$

$$PR_i^{vote.round,p,vote::v}$$

PV1阶段：

1. 如果对区块b的prevote票数超过了2/3，则进入PC阶段
2. Else if 如果不是对区块b的prevote达到了2/3（如对nil投票超过2/3） 则进入PC阶段，但是proposal为空

3 . Else （没有收到超过2/3的对同一个东西的pre-vote）
c信道等待接收(pv,vote)消息对.
if (vote.round < r)，则pv信道等待接收第r轮的prevote消息，收到后通过c信道发送(pv,w),同时有一个进程进入PV1阶段(递归)
else if (vote.round=r),则将这个prevote的投票记录并进v集合里，并递归进入PV1process
else **round>r** (什么时候发生这种情况？)进入PR阶段，设置r=vote.round，p还是本节点接收到的proposal p将(pv,vote),这个vote值并进v里。

$$PC_i^{r,p,v} := precommit_i!(p) \mid (\nu\, c)(\prod_{j=1}^{n} precommit_j?(w).c!(preccomit_j, w) \mid PC1_i^{r,p,v}(c))$$

$$\begin{aligned}
PC1_i^{r,p,v}(c) := \text{if } & max_b(|\left\{w \in v_r^2 : w.block = b\right\}|) > \frac{2}{3}N \text{ then} \\
& d_i!(b) \\
& \text{else if } |v_r^2| > \frac{2}{3}N \text{ then} \\
& \quad PR_i^{r+1,\emptyset,v} \\
& \text{else} \\
& \quad c?(pc, vote).\ \text{if } vote.round < r \text{ then} \\
& \quad\quad pc?(w).c!(pc, w) \mid PC1_i^{r,p,v}(c) \\
& \quad \text{else if } vote.round = r \text{ then} \\
& \quad\quad PC1_i^{r,p,vote::v}(c) \\
& \quad \text{else} \\
& \quad\quad PR_i^{vote.round,p,vote::v}
\end{aligned}$$

附录

The grammar of a simple π-calculus, in Backus-Naur form, is as follows

$$
\begin{array}{rrr}
P := & 0 & void \\
| & P \mid P & par \\
| & \alpha.P & guard \\
| & \alpha.P + \alpha.P & guarded\text{-}choice \\
| & (\nu x)P & fresh \\
 & & \\
| & F^s(y) & func \\
 & & \\
\alpha := & \tau & null \\
| & x!(y) & send \\
| & x?(y) & receive \\
| & susp_i & suspect
\end{array}
$$

P：process

=

0 ：空进程

P|P :并发的两个进程

α.p ： 在alpha这个动作发生后，才可以执行进程p

α.P+α.P ： 可以为alpha和Beta，alpha和beta会不确定的执行某一个，然后就会导致执行的是alpha后的进程还是beta后的进程

(vx)P:创建新的通道x，只有在P进程才可以访问这个通道

Alpha是一个action：可能为空；x!(y)：通过x信道发送y；X?（y）:通过y信道接收y；Susp为怀疑（？）

- $F^{s}(y)$ ： 允许传递变量s和y到process F ， 可以递归执行
- s是state-like(?)变量， y是channels

- Susp 动作是在超时之后触发的