IBC
(跨链通信Draft)

苏玉萌

2017.11.24

目录

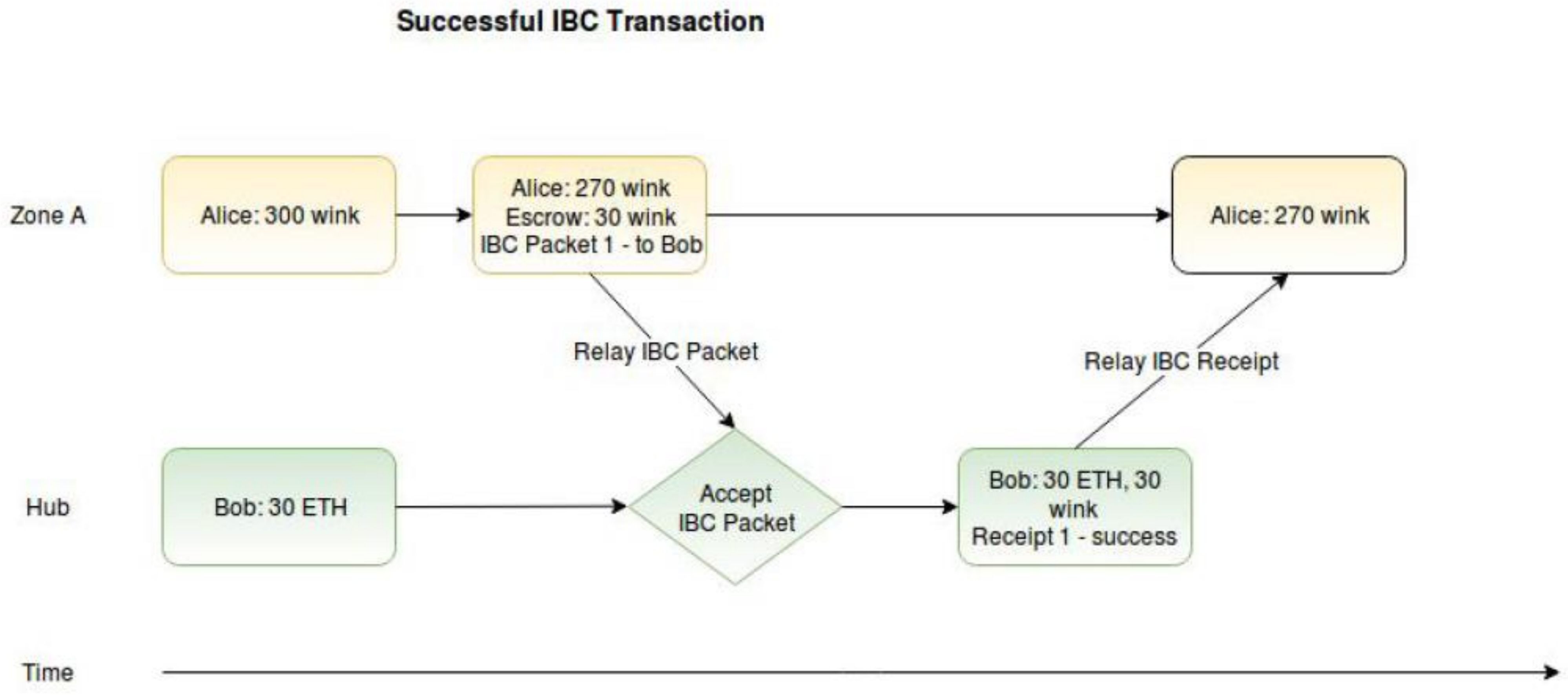
- 实际案例
- 消息通信原语
  - 注册 (permissioned action)
  - 消息队列设计
  - 发送IBC Packet
  - 转发IBC Packet
  - 收据
  - 验证者集合改变
- 轻客户端Proofs （验证区块头、动态验证者集合改变、Merkle proofs）
- 更高级的消息（超时、清理queue、链请求断开连接、处理硬分叉、各种路由和转发的）

privileged transaction

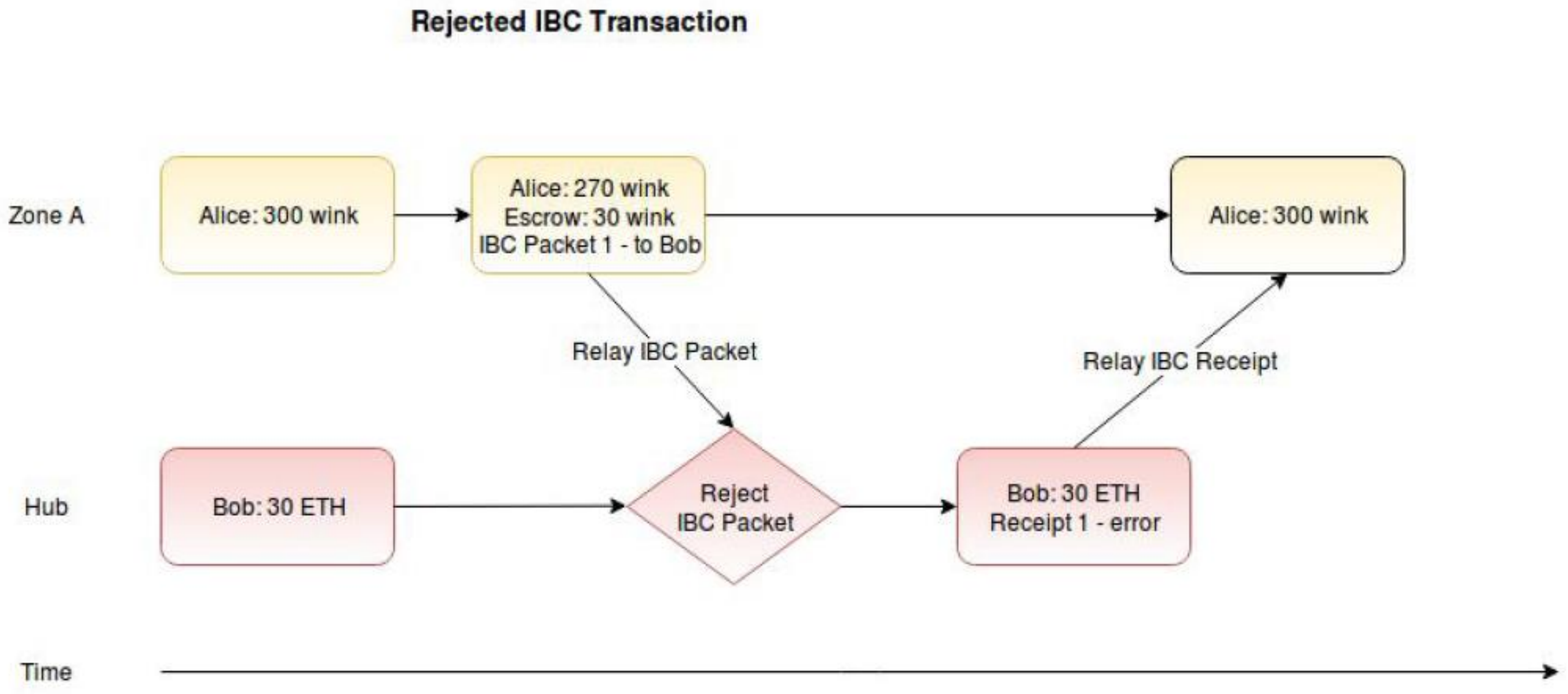
Inter Blockchain Communication 协议

- 在两个“侧链”上，可以进行跨链token交换，得益于Tendermint的Instant Finality，可以快速传递token
- 采用“消息传递”机制
- 由Hub和Zone组成，Hub和Zone都是区块链系统，Zone之间的跨链通信由Hub去中继，而Zone的正常运行是完全自治的。
- 协议可扩展，未来可以扩展为其他交易逻辑的跨链通信实现
  - 优点： 吞吐率与交易延迟可媲美中心化转账
不用交易双方实时在线
速度快
不会分叉

实际案例


Successful IBC Transaction
Zone A
Alice: 300 wink
Alice: 270 wink
Escrow: 30 wink
IBC Packet 1 - to Bob
Alice: 270 wink
Relay IBC Packet
Relay IBC Receipt
Hub
Bob: 30 ETH
Accept
IBC Packet
Bob: 30 ETH, 30
wink
Receipt 1 - success
Time

实际案例



Rejected IBC Transaction
Zone A
Alice: 300 wink
Alice: 270 wink
Escrow: 30 wink
IBC Packet 1 - to Bob
Alice: 300 wink
Relay IBC Packet
Relay IBC Receipt
Hub
Bob: 30 ETH
Reject
IBC Packet
Bob: 30 ETH
Receipt 1 - error
Time

消息队列设计

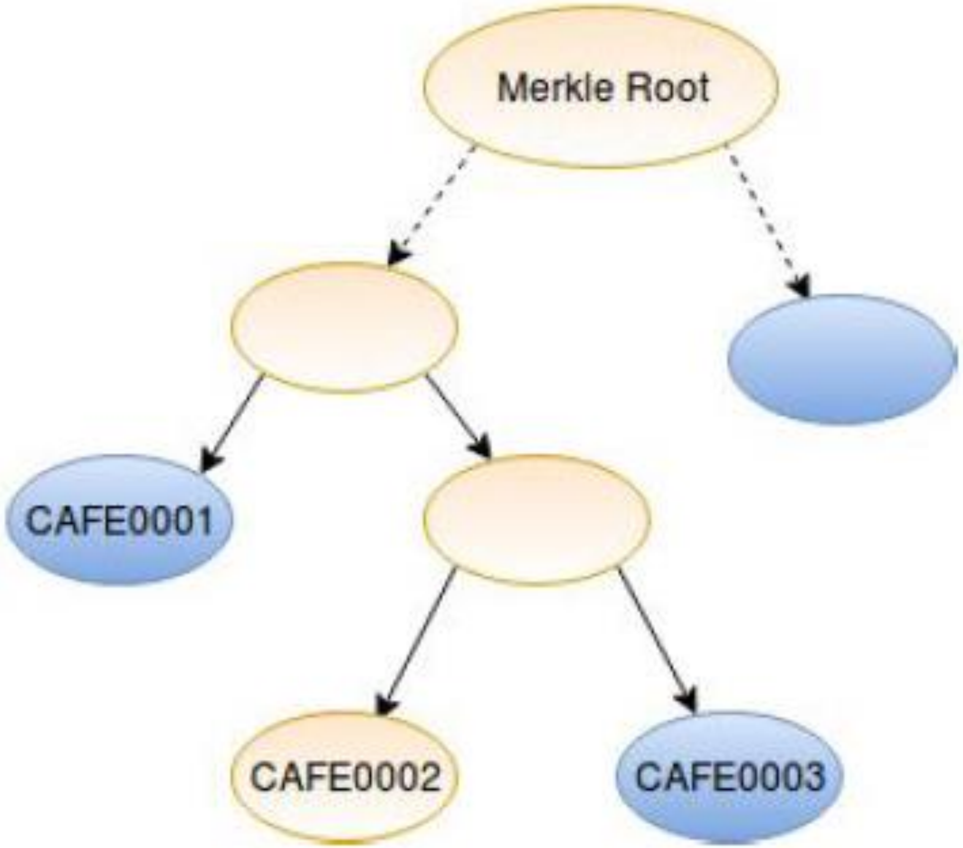
- 其他chain在本zone注册后，分别获得一个OutQueue和InQueue
- 每个Queue有唯一的前缀 Prefix
- key： 8字节大端整数，代表该Queue里包的序列号
- Prefix + key（一个前缀代表某个Queue，下属多个key-value对，每个key-value是一个具体的Packet） -----> IAVL tree上的节点
- multiple key-value pairs in the merkle data store of each blockchain.

  Key :Packet Sequence number

  Value：Packet (同样包含sequence number，为了merkle proof更简便)

消息队列设计

Queue structure and proof



消息队列设计

- 特点

  只可以添加序列号比当前sequence number大1的包

  可快速找的最新的包

  可证明某个指定序列号的包是否存在，若存在，可以从K-V存储中获得value（即Packet ）

  Packet被写入K-V存储后，是不可修改的

逻辑整理

- 先在本zone对交易进行共识，需要产生跨链交易的场景：如向其他链注册、对注册链更新自己的validator set 和头部等、跨链转账、收据、断开连接等等
- 交易共识通过后，需要产生Packet ，Packet的形态也需要全网达成共识。Packet的payload区域存的是什么呢？（tx？）（当区块链逻辑认为包是有效的，包才会真正产生）
- 发送Packet也需要全网共识，大家一致同意，才去发送包

IBC交易类型

IBCPacketCreateTx

- 已经注册完成
- **IBCPacketCreateTx ： 创造一个packet**
- IBC模块计算下一个sequence number，并将“Queue前缀+计算出的序列号”放入Outqueue中
- 包会被写入本地merkle化的存储位置
- 当区块链逻辑认为包是有效的，且确实可以发送，包才会真正产生，区块链逻辑会为这个包保留一个全局变量，

```go
type IBCPacketCreateTx struct {
  Packet
}

type Packet struct {
  SrcChainID string
  DstChainID string
  Sequence   uint64
  Type string
  Payload    []byte
}
```

注册（Permissioned action）

- 特殊注册交易**IBCRegisterChainTx**

type IBCRegisterChainTx struct { BlockchainGenesis }

type BlockchainGenesis struct { ChainID string Genesis string }

A向B发送IBCRegisterChainTx，会发送ChainID(A),Header 、Commit和Validator set

```go
// Header defines the structure of a Tendermint block header
type Header struct {
    ChainID        string     `json:"chain_id"`
    Height         int        `json:"height"`
    Time           time.Time  `json:"time"`
    NumTxs         int        `json:"num_txs"` // XXX: Can we get rid of this?
    LastBlockID    BlockID    `json:"last_block_id"`
    LastCommitHash data.Bytes `json:"last_commit_hash"` // commit from validators from the last block
    DataHash       data.Bytes `json:"data_hash"`        // transactions
    ValidatorsHash data.Bytes `json:"validators_hash"`  // validators for the current block
    AppHash        data.Bytes `json:"app_hash"`         // state after txs from the previous block
}
```

```go
// Commit contains the evidence that a block was committed by a set of validators.
// NOTE: Commit is empty for height 1, but never nil.
type Commit struct {
    // NOTE: The Precommits are in order of address to preserve the bonded ValidatorSet order.
    // Any peer with a block can gossip precommits by index with a peer without recalculating the
    // active ValidatorSet.
    BlockID    BlockID `json:"blockID"`
    Precommits []*Vote `json:"precommits"`

    // Volatile
    firstPrecommit *Vote
    hash           data.Bytes
    bitArray       *cmn.BitArray  //BitArray returns a BitArray of which validators voted in this commit
}
```

```go
// Volatile state for each Validator
// NOTE: The Accum is not included in Validator.Hash();
// make sure to update that method if changes are made here
type Validator struct {
	Address     data.Bytes    `json:"address"`
	PubKey      crypto.PubKey `json:"pub_key"`
	VotingPower int64         `json:"voting_power"`

	Accum int64 `json:"accum"`
}
```

```go
// ValidatorSet represent a set of *Validator at a given height.
// The validators can be fetched by address or index.
// The index is in order of .Address, so the indices are fixed
// for all rounds of a given blockchain height.
// On the other hand, the .AccumPower of each validator and
// the designated .GetProposer() of a set changes every round,
// upon calling .IncrementAccum().
// NOTE: Not goroutine-safe.
// NOTE: All get/set to validators should copy the value for safety.
// TODO: consider validator Accum overflow
type ValidatorSet struct {
	// NOTE: persisted via reflect, must be exported.
	Validators []*Validator `json:"validators"`
	Proposer   *Validator   `json:"proposer"`

	// cached (unexported)
	totalVotingPower int64
}
```

更新验证者集合

- **IBCUpdateChainTx：** 更新注册链的状态

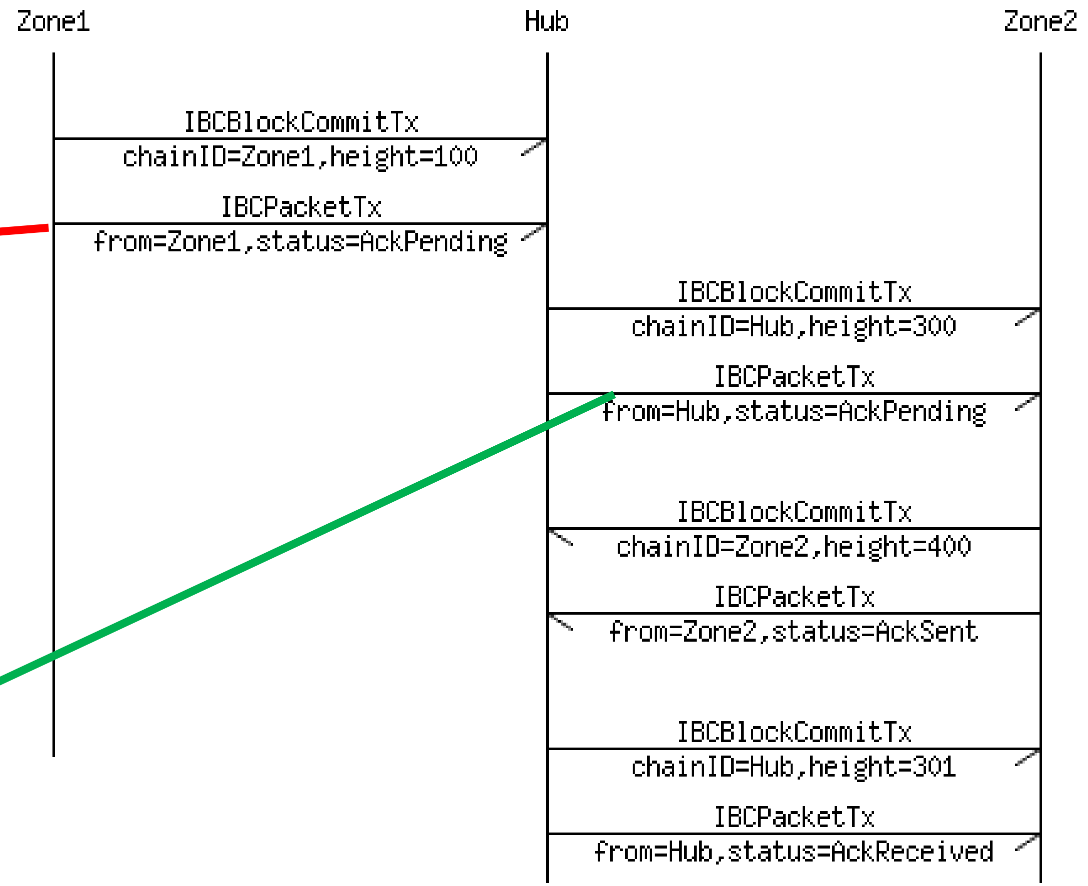
```
type UpdateChainTx struct {
    Header *types.Header `json:"header"`
    Commit *types.Commit `json:"commit"`
    Validators    *types.ValidatorSet `json:"validator_set"`
}
```

IBCPacketPostTx

```go
type PostPacketTx struct {
    // The immediate source of the packet, not always Packet.SrcChainID
    FromChainID string `json:"src_chain"`
    // The block height in which Packet was committed, to check Proof
    FromChainHeight uint64 `json:"src_height"`
    // this proof must match the header and the packet.Bytes()
    Proof  *iavl.KeyExistsProof `json:"proof"`
    Key    data.Bytes           `json:"key"`
    Packet Packet               `json:"packet"`
}
```

例子

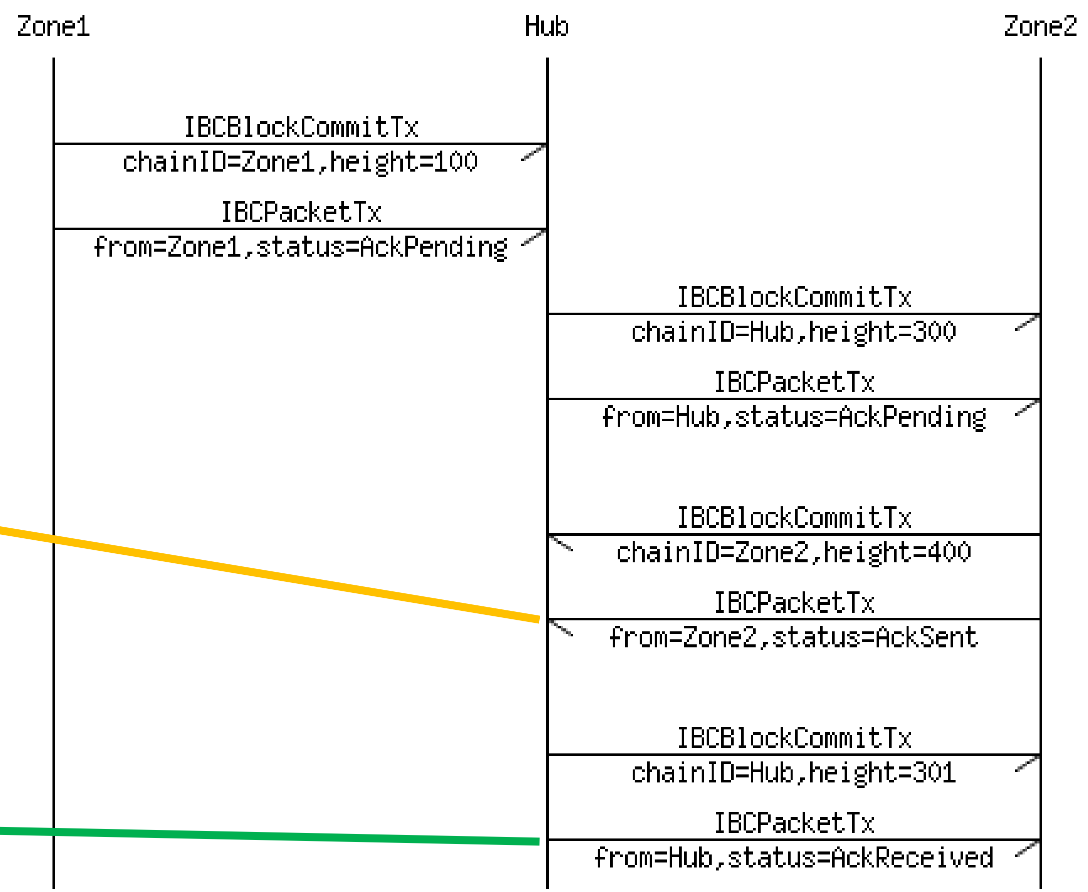
- `FromChainID` : "Shard1"
- `FromBlockHeight` : 100 (say)
- `Packet` : an `IBCPacket` :
  - `Header` : an `IBCPacketHeader` :
    - `SrcChainID` : "Shard1"
    - `DstChainID` : "Shard2"
    - `Number` : 200 (say)
    - `Status` : `AckPending`
    - `Type` : "coin"
    - `MaxHeight` : 350 (say "Hub" is currently at height 300)
  - `Payload` : <The bytes of a "coin" payload>



- `FromChainID` : "Hub"
- `FromBlockHeight` : 300
- `Packet` : an `IBCPacket` :
  - `Header` : an `IBCPacketHeader` :
    - `SrcChainID` : "Shard1"
    - `DstChainID` : "Shard2"
    - `Number` : 200
    - `Status` : `AckPending`
    - `Type` : "coin"
    - `MaxHeight` : 350
  - `Payload` : <The same bytes of a "coin" payload>

例子

- `FromChainID` : "Shard2"
- `FromBlockHeight` : 400 (say)
- `Packet` : an `IBCPacket` :
  - `Header` : an `IBCPacketHeader` :
    - `SrcChainID` : "Shard1"
    - `DstChainID` : "Shard2"
    - `Number` : 200
    - `Status` : `AckSent`
    - `Type` : "coin"
    - `MaxHeight` : 350
  - `PayloadHash` : <The hash bytes of the same "coi



- `FromChainID` : "Hub"
- `FromBlockHeight` : 301
- `Packet` : an `IBCPacket` :
  - `Header` : an `IBCPacketHeader` :
    - `SrcChainID` : "Shard1"
    - `DstChainID` : "Shard2"
    - `Number` : 200
    - `Status` : `AckReceived`
    - `Type` : "coin"
    - `MaxHeight` : 350
  - `PayloadHash` : <The hash bytes of the same "coin" payload>

消息原语

- 消息队列设计
- 注册 (permissioned action)
- 发送IBC Packet
- 验证者集合改变

- 转发IBC Packet
- 收据

Relay Packet

- 转发节点只需要 light client proof ,转发节点不需要是验证者或是全节点
- 如何给转发包的节点发送交易费？

In order to bootstrap a system, the chain developers can provide a special account with enough money to relay, and pay the fees for all packets, until users have tokens on both chains.

- 一次跨链可以发送多个包，以节省转发过程中校验、交易费，可以缓存一些包，去打包发送
- 允许多路并行转发，但是不允许单节点试图重复转发多次一个包

Receipt

- 当IBC包被发送到其他链

-> valid：proof与已知头部一致，将Packet存在Inqueue里

将交易发送到合适的智能合约去校验有效性，并进行执行

成功执行：proof证明成功执行，返回一个ABCI结果 success

执行失败：proof证明失败，返回结果error

接收到的Packet + Proof+执行的结果 放入接收方的Inqueue中

- 中继节点查看proof,确定是发送方发送的包在接收方处得到了执行，将收据发回给链A
- 收据转发给发送方，收据将触发一系列发送方处的智能合约

Light Client Proofs

- **验证区块头部**： 转发节点信任验证者集合，有注册过的头部和验证者集合-> 检查收到的Packet中的头部是不是超过2/3的voting power的验证投票即可

- 上述安全的前提： 转发节点存储的头部是被可信的验证者集合签名的头部，且验证者没有处于申请解绑资产的过程中。

- 若收到分叉/假头部可以作为罚没的证明，罚没发出、签名这个包的validator1/3的资产

Light Client Proofs—验证者集合改变

- 验证改变是有效的
- 用改变后的验证者集合去验证头部
- （需要提供验证者集合的完整数据，头部包含的仅是哈希，不够）

- First, that the new header, validators, and signatures are internally consistent
  - We have a new set of validators that matches the hash on the new header
  - At least 2/3 of the voting power of the new set validates the new header
- Second, that the new header is also valid in the eyes of our trust set
  - Verify at least 2/3 of the voting power of our trusted set, which are also in the new set, properly signed a commit to the new header

Merkle Proof

- 验证包真实存在于发送方处

- 头部里有AppHash字段：给定Packet的校验路径，可以由一系列哈希最终得到AppHash，若与头部中的一致，即可验证packet

```go
// Header defines the structure of a Tendermint block header
type Header struct {
	ChainID        string      `json:"chain_id"`
	Height         int         `json:"height"`
	Time           time.Time   `json:"time"`
	NumTxs         int         `json:"num_txs"` // XXX: Can we get rid of this?
	LastBlockID    BlockID     `json:"last_block_id"`
	LastCommitHash data.Bytes `json:"last_commit_hash"` // commit from validators from the last block
	DataHash       data.Bytes `json:"data_hash"`        // transactions
	ValidatorsHash data.Bytes `json:"validators_hash"`  // validators for the current block
	AppHash        data.Bytes `json:"app_hash"`         // state after txs from the previous block
}
```

Timeout

- 利用IBC Packet 的MaxHeight字段，设置在接收方的某个高度前收到包才有效，否则超时不会执行。

1、发送方在包里添加MaxHeight：H，期望在接收方H高度前包被收到

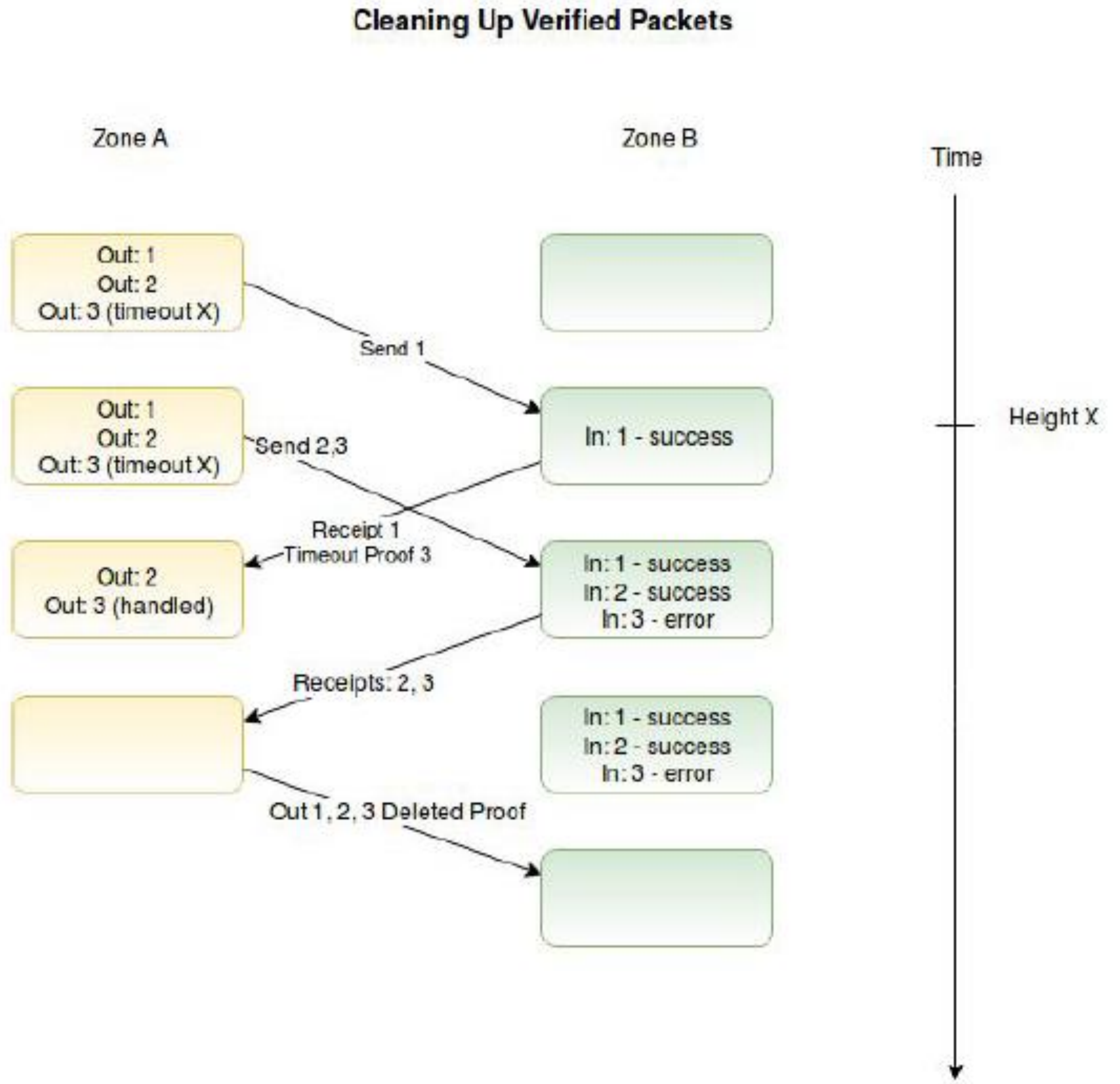
2、发送方确认超时的方式：过了一段时间，接收方高度H’（>H），发送方请求接收方，查询在H’高度是否存在发出的包，产生一个non-existence proof ，证明在H’高度前接收方没有收到发出的包，将proof发回发送方。让发送方确认在在这之后，如果接收方收到了包，不会执行。发送方可以安全的解冻资金。

删除包

- 当收到收据，可以将已经完成跨链通信的包从发送方队列（out queue）中删除。收据必须按序列号处理，因此第一个收据一定是队列头部的包的收据。

- 需要删除接收方in queue中的包：设计新的交易类型，当收据被发送方收到后，删除out queue中的队首的包，产生一个 non-existence proof ，发回接收方，让其删除in queue中的相应内容。

Clean up 例子



Cleaning Up Verified Packets
Zone A
Zone B
Time
Out: 1
Out: 2
Out: 3 (timeout X)
Send 1
In: 1 - success
Height X
Out: 1
Out: 2
Out: 3 (timeout X)
Send 2,3
Receipt 1
Timeout Proof 3
Out: 2
Out: 3 (handled)
In: 1 - success
In: 2 - success
In: 3 - error
Receipts: 2, 3
In: 1 - success
In: 2 - success
In: 3 - error
Out 1, 2, 3 Deleted Proof

断开zone之间的连接（特权交易）

- 处理还在进行的不会引起不一致的跨链通信
- 解决断开连接后的遗留问题 （是否还有credit、已经转移出去的token但是never returned）
- 若认为某个连接是恶意的，可以添加“hard shutdown”flag，将所有未处理的交易都处理为error，然后快速断开连接。

断开—处理还在进行的跨链通信

- 在out queue队尾添加“disconnect”packet，避免发送包
- 对新收到的包，return error
- 这两个操作会被转发的节点获取，并在转发节点处触发相同的操作，即队尾“disconnect 包”，对新收到的包return error
- 之前已经进行发送的包，要等到收到收据，当out queue中所有包都收到收据，才真正断开连接

硬分叉（特权交易）

- 当产生硬分叉，可能会导致分成的两个部分都没有办法达到超过2/3投票力，会导致共识失败，分成的两个部分要强势改变原始的验证者集合，而客户端需要手动选择follow那一条链。
- IBC协议不会同时支持两条分叉链
- 添加一个特权交易，通过on-chain governance ，设置IBC连接follow某条链

多链路由

- 星型结构
- 一些问题
  - 信任可传递吗？（A信任B，B信任C → A信任C？）
  - Path finding （两个zone非直接相连，应该向哪个queue写消息呢？）

路由—网关（Path finding ）

- 基于IP协议，将IP地址替换成chain id
- 丢弃形成环路的packet
- 确保Packet的请求转发路径和响应转发路径是一致的
- 发送包时path finding的顺序：
    - 1、两个zone直接连接
    - 2、“ethermint,dex”
    - 3、列出前缀“cosmos-*”
    - 4、default gateway

路由—Blind Relay

- 不会考虑包的内容，单纯根据source 和 destination 的chain id去路由，逐跳去转发
- 单跳，盲转发是可行的
- 扩展到多跳，需要所有转发路径上相关节点，都要互相信任，即都要互相注册，转发节点间要互相有账户，去决策是否接受这个包的这笔交易（是否在我的链上具有足够的credit：是否是token的发行方？ 我是否在交易发送方上有一笔交易滞留金？）

路由—Vouching Relay

- 担保路由： 转发节点要知道转发的Packet的内容，并且会在本地执行，只有当转发节点认为这个包有效，才会转发出去这个包。each hub must fully verify the message before forwarding it to the next hop.
- 可以设计特殊的应用逻辑去response 这个包（比如说？）
- Hub会为每个相连的zone维持一个total-in 和total-out 余额。

Zone Discovery

- Zone A和Zone B都和hub建立了连接，彼此信任。那么A和B可以通过Hub去彼此建立IBC通信链路，不需要人工验证

- 信任Zone A的轻客户端，可以信任Hub，通过Hub去信任B，与zone B建立互信。

End

An `IBCBlockCommitTx` transaction is composed of:

- `ChainID (string)`: The ID of the blockchain
- `BlockHash ([]byte)`: The block-hash bytes, the Merkle root which includes the app-hash
- `BlockPartsHeader (PartSetHeader)`: The block part-set header bytes, only needed to verify vote signatures
- `BlockHeight (int)`: The height of the commit
- `BlockRound (int)`: The round of the commit
- `Commit ([]Vote)`: The +⅔ Tendermint `Precommit` votes that comprise a block commit
- `ValidatorsHash ([]byte)`: A Merkle-tree root hash of the new validator set
- `ValidatorsHashProof (SimpleProof)`: A SimpleTree Merkle-proof for proving the `ValidatorsHash` against the `BlockHash`
- `AppHash ([]byte)`: A IAVLTree Merkle-tree root hash of the application state
- `AppHashProof (SimpleProof)`: A SimpleTree Merkle-proof for proving the `AppHash` against the `BlockHash`

IBCPacketTx

An `IBCPacket` is composed of:

- `Header (IBCPacketHeader)` : The packet header
- `Payload ([]byte)` : The bytes of the packet payload. *Optional*
- `PayloadHash ([]byte)` : The hash for the bytes of the packet. *Optional*

Either one of `Payload` or `PayloadHash` must be present. The hash of an `IBCPacket` is a simple Merkle root of the two items, `Header` and `Payload` . An `IBCPacket` without the full payload is called an *abbreviated packet*.

An `IBCPacketHeader` is composed of:

- `SrcChainID (string)` : The source blockchain ID
- `DstChainID (string)` : The destination blockchain ID
- `Number (int)` : A unique number for all packets
- `Status (enum)` : Can be one of `AckPending` , `AckSent` , `AckReceived` , `NoAck` , or `Timeout`
- `Type (string)` : The types are application-dependent. Gnuclear reserves the "coin" packet type
- `MaxHeight (int)` : If status is not `NoAckWanted` or `AckReceived` by this height, status becomes `Timeout` . *Optional*

An `IBCPacketTx` transaction is composed of:

- `FromChainID (string)`: The ID of the blockchain which is providing this packet; not necessarily the source
- `FromBlockHeight (int)`: The blockchain height in which the following packet is included (Merkle-ized) in the block-hash of the source chain
- `Packet (IBCPacket)`: A packet of data, whose status may be one of `AckPending`, `AckSent`, `AckReceived`, `NoAck`, or `Timeout`
- `PacketProof (IAVLProof)`: A IAVLTree Merkle-proof for proving the packet's hash against the `AppHash` of the source chain at given height

The sequence for sending a packet from "Shard1" to "Shard2" through the "Hub" is depicted in {Figure X}. First, an `IBCPacketTx` proves to "Hub" that the packet is included in the app-state of "Shard1". Then, another `IBCPacketTx` proves to "Shard2" that the packet is included in the app-state of "Hub". During this procedure, the `IBCPacket` fields are identical: the `SrcChainID` is always "Shard1", and the `DstChainID` is always "Shard2".

The `PacketProof` must have the correct Merkle-proof path, as follows:

```
IBC/<SrcChainID>/<DstChainID>/<Number>
```

IBC交易类型

- **IBCRegisterChainTx：** 在其他链上注册本链
- **IBCUpdateChainTx：** 在链上更新另一个链的状态
  IBCPacketCreateTx： 创造一个**packet**
- **IBCPacketPostTx：**