IBC & public-private blockchain interoperation

Christopher Goes, Cosmos

About this talk

- Explain IBC **design goals**
- Demystify implementation
    - IBC is a **protocol**!
    - How to do it yourself (rapid version)
- Walk through example
    - Using IBC to connect Ethereum & Zcash
    - **Shielded ERC20 transactions**: privately send your existing ERC20 tokens

What - “Inter-blockchain communication”

- Mechanism for **relaying data packets** between chain A & chain B
- Off-chain relayers responsible for watching chain A & committing transactions to chain B
  - Relayers are not permissioned, **anyone can relay**
- **Opaque payload**: IBC protocol defines causal ordering semantics, any application logic can be implemented on top

What - Authenticated message passing

- Rules on chain B which define a subset of accepted packets
  - If packet has been received on B, know something about state of A
- Chain B verifies **proof** that packet has been sent on chain A
- Various methods of doing so, with different security properties
  - Chain B acts as a Bitcoin-style light client of chain A (Merkle proofs)
  - Intermediary “peg zone” with finality and accountable validator set
- **Must trust** - consensus algorithms and light client verification

What - Use cases

- Basics
  - Transfer tokens (fungible/non-fungible) across chains
  - Conditionalize contract execution or output locking on state of another chain
- Sharding-esque
  - Split & parallelize contract logic across multiple chains with compatible VMs
    - IBC exposes a channel-like primitive
  - Compose hybrid EVM-Cosmos-Zcash state machines
  - Delegated security - validators for one chain slashable over IBC on another
- Key point - **heterogenous chains**
  - Various combinations of IBC primitives can be implemented on each chain
  - Sovereign chains can have custom features (zkSNARKs!)

How - Security model

- Goal - allow preservation of “contract invariants” across chains
    - Burn asset T on chain A, redeem “T vouchers” on chain B
    - Burn T vouchers on chain B, redeem T on chain A
    - (which is really the voucher? No canonical chain!)
- State machine of B must verify that state machine of A burned T
    - Can encapsulate some of this verification in the known components of Alpha/Beta state machines prior to creating IBC connection
        - Or prove it directly if code is stored in state and VMs are compatible
    - After A burns T, writes some data to a provable store
        - Then B can just verify light client proof of written data
        - Data couldn’t have been written if T hadn’t been burned

How - Connection lifecycle

- Connection - data necessary to verify packets between two chains
- Opening a connection
  - **Root-of-trust** - genesis block, initial validator set added “at connection creation”
    - When smart contract is deployed
    - When governance implements upgrade
  - Any connection user can check the root-of-trust
- Updating trusted headers
  - **Verify update** from H_1 to H_2
    - Tendermint - Track validator set, check signed by previous validator set
    - Nakamoto consensus - continues previous chain, most work seen
  - Headers allow packet proof verification
- Closing a connection
  - In exceptional cases - fork, safety violation
  - Can be done by permissioned entity (governance) or some on-chain proof-of-fraud

How - Strictly ordered message passing

- IBC implements a **vector clock** for two processes (blockchains)
    - *A*, *B* chains of interest
    - *i* packet counter
    - *x*, *y* events (other transactions)
    - -> before, => implies
    - $A_{send:i} \rightarrow B_{receive:i}$
    - $B_{receive:i} \rightarrow A_{receipt:i}$
    - $A_{send:i} \rightarrow A_{send:i+1}$
    - $x \rightarrow A_{send:i} \Rightarrow x \rightarrow B_{receive:i}$
    - $y \rightarrow B_{receive:i} \Rightarrow y \rightarrow A_{receipt:i}$
- Consensus provides single canonical ordering on a chain
- IBC provides single canonical ordering across chains
- Easily generalized to an *n*-process vector clock for multi-chain ordering
    - *a* on chain A before *b* on chain B before *c* on chain C

How - Ordering guarantees

- Ordering guarantee can be used to reason about the combined state of both chains as a whole
    - Example: fungible token total supply
        - If packet *i* is sent on A, burn tokens
        - If packet *i* is received on B, mint vouchers
            - Tokens must have previously been burned on A
            - Total supply conserved
        - Counters prevent replay on the same chain
        - Metadata prevents replay on other chains

How - Channels

- Channel: abstraction providing ordering guarantee
  - Set of four queues, two per chain
    - On chain A
      - Outgoing A-B
      - Incoming B-A receipts
    - On chain B
      - Outgoing B-A
      - Incoming A-B receipts
  - Each queue keeps a counter
  - Packets can only be sent & received in order
    - If counters mismatch, reject

How - Packets

- Packet - Individual datagram with opaque payload & metadata
    - Five-tuple - (type, sequence, source, destination, data)
        - **Type** - multiplexing (one connection, many applications)
        - **Sequence** - ordering guarantee (prevent replay)
        - **Source** - source chain (prevent replay masquerading from another chain)
        - **Destination** - destination chain (prevent replay on another chain)
        - **Data** - opaque application-specific payload
    - Packets are committed **once, in order, from only one chain to only one chain**

How - Receipt, acknowledgement, timeout

- IBC receipt - IBC packet back to the source chain
  - Proves original packet was received & acted upon
    - Relevant application-specific action was taken
  - Data for proof can then be deleted from Merkle tree
- IBC timeout on source chain
  - Each packet additionally contains timeout *t* relative to destination chain state
  - Destination chain rejects packets past *t*
  - *t* can be height or timestamp, must be proved back to A for asset release
    - Via same proof as IBC packets (Merkle path)
    - **Provides safety** if packet isn't committed on destination chain
    - Assets can be un-escrowed, released to original sender

Why - One asset, many chains

- Assets can be freely transferred
    - Send your BTC from Bitcoin, to Ethereum, to Cosmos, to Zcash, then back
    - **Always the “same” BTC** - can be redeemed back through that path to vanilla BTC
- Multi-hop routing reduces implementation cost
    - Assets can be sent along any path of individually-connected chains
- Permissionless
    - Set up another chain, implement IBC with an existing asset, add new features

Why - New features, old security

- When you need privacy, send your BTC to Zcash
  - Usually keep it on the Bitcoin chain for security
  - (but be careful about linkability!)
- No trusted setup risk
  - IBC contract on Bitcoin (or peg) can track total supply in/out
  - If trusted setup was compromised:
    - BTC on Zcash chain is at risk
    - BTC on Bitcoin chain is at no risk - IBC contract will cap inflation
  - **Risk is always opt-in!**
- Generalizable: supply for fungible tokens, uniqueness for NFTs, etc.

Why - Opt-in upgrades

- Add new features to a chain, **no governance required**
  - Copy the chain, add the features, create an IBC connection back to the old chain
  - Anyone can elect to move their existing assets to the new chain
    - Could allow moving back to the old chain - or not
  - Security?
    - Naive (simple) PoS - less secure initially, more secure as staking token moves
    - Delegated security - New chain validators slashable on old chain over IBC
- No need for a new asset
- Both chains can peacefully coexist

Aside - Consensus requirements

- State finality
  - **IBC safety requires finality**, otherwise coins could be double-spent
  - Consensus landscape
    - Tendermint/PBFT - Instant finality
    - Casper FFG - Fast finality
    - Nakamoto consensus (PoW, Tezos/Ouroboros Praos PoS)
      - Probabilistic finality, must pick a threshold
      - Could vary threshold based on transfer amount (~risk)
- Proof verification
  - With smart contracts or custom state machine - natively
  - Without smart contracts - separate “peg chain” to bridge

Aside - Bridged “Peg-zone”

- Accountable federated peg
  - Cosmos Bonded PoS (and similar) have configurable slashing conditions
  - Assets controlled by weighted (or k-of-m) **multi-signature on pegged chain**
    - e.g. Bitcoin, Zcash, Monero
    - Future alternative - multiparty ECDSA
  - **Second Tendermint chain** with validator set of multi-signature
    - Transactions committed to the bridge chain must be signed by multi-sig, then can be relayed to main chain
    - Any transactions signed but not committed to bridge chain are slashable!
      - Also slashable - failure to update multi-sig if validator set is changed
    - Received transactions on main chain can be relayed back to bridge chain for ease-of-verification
    - Configurable finality threshold (*n* confirmations)
  - Main (pegged) chain **needs no additional features**

Example - Ethereum ERC20 ⇔ Zcash UIT

- Zcash UITs (User-issued tokens)
  - https://github.com/zcash/zcash/issues/830
  - Bitcoin-style “colored coins” on Zcash
  - Not in Sapling but maybe soon? (see discussion on the issue)
  - Alternatives: Zcash-fork per ERC20 token, or ZEC-as-ERC20
- Associate each ERC20 token contract with a unique UIT identifier
  - “ibc/ethereum/{contract address}”
  - Identifiers can be created lazily on demand (and also for new tokens)
- ERC20 token vouchers can be shielded on the Zcash chain
- Can remain so indefinitely and be shielded-transferred as usual
- Once unshielded, can be transferred back to the original ERC20

ERC20 ⇔ UIT - Components

- Dedicated smart contract on Ethereum
  - Holds ERC20 tokens in escrow
  - Provide proof that x of token y has been escrowed
  - Verify proof that UIT has been burned and release escrowed ERC20 tokens
  - (channel ordering semantics are optional since transfers are commutative)
- Zcash multisignature (or multiparty ECDSA) peg chain
  - Multisignature (unshielded) account on Zcash chain
  - Signer set of multisignature account runs Tendermint consensus on peg chain
  - Peg chain
    - Verifies proofs of ERC20 token escrow on Ethereum
    - Provides proofs of UITs burned to Ethereum smart contract
    - Slashes multisignature members if they commit a fault

ERC20 ⇔ UIT - Step I (Ethereum)

- User has some ERC20 token *T*, on the Ethereum chain
- User calls ERC20 *transfer* to send token to IBC smart contract
  - IBC contract **escrows ERC20 token**
  - IBC contract **logs the escrow event** in the Patricia trie
    - Includes: sender, token, amount, desired destination address

ERC20 ⇔ UIT - Step II (Zcash peg chain)

- IBC packet sent from Ethereum to Zcash peg chain
    - IBC relayer (user or third-party) **constructs IBC packet**
        - Packet references escrow event on Ethereum chain
    - IBC relayer **commits packet** in a transaction to Zcash peg chain
        - Zcash peg chain state machine verifies proof of event

ERC20 ⇔ UIT - Step III (Zcash)

- Zcash peg chain validators sign **UIT-mint transaction**
  - UIT-mint transaction mints the associated UIT denomination in the amount proved by the Ethereum event on the peg chain, to the user-specified destination address
  - Peg chain validators or user commit mint transaction to Zcash chain
- User can then **transact as normal** with a Zcash UIT
  - Shield to a z-address, spend privately, unshield/shield again as desired
  - Privacy equivalent: ERC20-side is “unshielded pool”
    - Be careful about pattern-matching linkability!
- UIT can be **sent back to Ethereum in reverse**
  - Provably burn the UIT on Zcash, submit proof to Zcash peg chain
  - IBC relayer submits packet to Ethereum smart contract which unescrows the tokens

ERC20 ⇔ UIT - Properties

- **Permissionless** & opt-in
  - Anyone can implement for any token
  - Which chain to keep on, when to move up to the asset holder
  - No risk for existing ERC20 token holders of inflation
    - IBC contract can track total supply in/out
- Full Zcash-level **privacy** when shielded
  - Same level of linkability as transparent addresses / unshielded pool
- **Scalable** implementation
  - Same logic (and one peg chain) can handle all ERC20 contracts & associated UITs

Future Research - higher-assurance proofs

- IBC security dependent on correct light client proofs
- Various degrees of light-client assurance
    - Bitcoin - relatively **low**
        - No state transition verification
        - Long forks do seem to be unlikely in practice...
    - Cosmos/Tendermint - **medium**
        - No state transition verification
        - But - lying to light client always a slashable offense
            - IBC packet data is public, so it would definitely be caught!
        - Security dependent on value-at-stake (hence the Cosmos hub-spoke model)
    - Coda Protocol / recursive SNARKs - **high**
        - Can verify full state transition history!
        - Still need to disincentivize forks

Future Research - Byzantine recovery

- What if consensus breaks?
  - No valid chain, *n* valid chains (valid for IBC = can provide light client proofs)
  - Wide space of possibilities
    - *n* valid chains: can spot duplicate headers when published
    - 0 valid chains: no progress without intervention (but timeouts protect assets)
- What if one chain's state machine has a bug?
  - No protection right now - **if you hold assets on the chain, you accept the risk**
  - Could implement fraud proofs
    - Complex requirements on state machines (on both chains)
  - Where applicable, governance on one chain could "fix" - credit the original tokens at parity with balances on the other chain before the bug was exploited
    - Not an ideal solution, irregular state change
- Ideal case - asset security (invariant set) as long as 1-of-n chains are correct & live

Questions?

- Cosmos will implement IBC for our stack, but the protocol is open!
  - Alternative implementations, contributions, ideas welcome
- Spec - github.com/cosmos/cosmos-sdk/tree/develop/docs/spec/ibc
- Email - cwgoes@tendermint.com
- Check-out the break-out!
  - Rob Habermeier leading on zero-knowledge state transitions in blockchains, **15:40 in Breakout 2**