Cosmos IBC Specification

(First Draft)

Ethan Frey

frey@tendermint.com

Sep. 29, 2017

Abstract

This paper specifies the Cosmos IBC (inter blockchain communication) protocol, which was first described in the Cosmos white paper[1] in June 2016. There are other techniques to include two chains in an atomic operation, such as “hashed time locked contracts”[2], but many just guarantee both transactions succeed or both fail. IBC creates complete 2-way “sidechains”, actually enabling transfer of value across chains, and takes full advantage of Tendermint’s instant finality to enable quick transmission of tokens.

IBC uses the message-passing paradigm and allows the participating chains to be independent. Each chain maintains a local partial order, while messages track any cross-chain causality relations. Once two chains have registered a trust relationship, packets can be securely sent from one chain to the other representing transfers of tokens from an account on one chain to an account on the other chain. The protocol is also extensible beyond token transfers, although designing secure communication logic for other types of applications is still an area of research. The protocol makes no assumptions of block times or network delays in the transmission of the packets between chains, and thus is highly robust in a heterogeneous environment.

This paper explains the requirements and structure of the Cosmos IBC protocol. It aims to provide enough detail to fully understand and analyze the security of the protocol.

---

1 https://github.com/cosmos/cosmos/blob/2d400fa5129596f800c824a1b0cbce92c052955e/WHITEPAPER.md#inter-blockchain-communication-ibc

2 https://z.cash/blog/htlc-bip.html

Contents

1. Introduction

2. Example Scenario

3. **Basic Messaging Semantics**
   3.1. Reliable Messaging Queue
   3.2. Registering Chains
   3.3. Validator Changes
   3.4. Sending a Packet
   3.5. Relaying a Packet
   3.6. Receipts

4. Light Client Proofs
   4.1. Verifying Block Headers
   4.2. Dynamic Validator Sets
   4.3. Merkle Proofs

5. Advanced Messages
   5.1. Timeouts
   5.2. Clean up
   5.3. Disconnecting Chains
   5.4. Handling Hard Forks
   5.5. Multi-chain routing
   5.5.1. Gateway
   5.5.2. Blind relay
   5.5.3. Vouching relay
   5.5.4. Matchmaker

6. **Conclusion**

1 Introduction

Cosmos IBC is designed around the Cosmos network and the Tendermint consensus engine, fundamentally relying on Tendermint's property of instant finality (meaning forks are never created). The Cosmos network will consist of the Cosmos Hub and multiple, independent zones, which all communicate with the hub via IBC. The Cosmos Hub can also bridge different zones by relaying IBC messages between them.

We begin with a concrete example of a simple case, which can be used to give form to any of the following abstract explanations. After that, the semantics of the message passing used in this example is explained. This is followed by the technical underpinnings of the secure proofs and packet transmission.

After that, there is discussion of more advanced message types, such as timeouts and routing to make larger networks efficient over time, and consideration given to the ability of the protocol to handle all expected conditions.

In addition to providing a theoretical understanding and reference document for the IBC protocol, this document aims to make convincing arguments for the secure foundations of all protocols. This document informs the ongoing Cosmos project and we will continue to adapt it as we inform the theory with data from praxis.

2 Example Scenario

To make this less abstract, here is an example of how this would work in a concrete case. Alice wants to send 30 winks (example token name) from her account on Cosmos zone X to Bob on the Hub. Assuming zone X had already established an IBC connection with the hub, Alice would create a transaction on zone X to initiate the transfer. Zone X would then freeze those tokens in some escrow and create an IBC packet requesting the Hub to create the corresponding tokens in Bob's account. In essence, the validator set on zone X is guaranteeing the destruction of these tokens in exchange for minting them on the Hub.

A separate relay process (which anyone can run as client software) can take a proof of that IBC packet from zone X and post it to the hub. The hub would then verify the block header, merkle proof, and sequence number to ensure this packet is a valid IBC packet from zone X. At this point there are two options, either zone X has enough credit on the hub to mint 30 winks and it is accepted, or it does not and the packet is rejected. If the packet is

accepted, the Hub mints 30 fresh winks in Bob's account, and stores the packet along with a record of the success in its incoming queue. If the packet is rejected, no tokens are minted, and a record of failure is stored in the queue.

What is this credit I refer to? We don't want every chain that attaches to the hub to be able to mint arbitrary numbers of arbitrary tokens on the hub, or all economic guarantees are quickly rendered meaningless. The Hub must use its own logic to validate whether it trusts zone X enough to accept those 30 winks. If zone X is the source of winks (native token to X), the Hub will give it a very large credit to send these winks onto the Hub. If Hub has previously sent 500 winks to zone X, then it must store this information and allow zone X to send those 500 winks (and no more) back onto the Hub, to allow the free flow of tokens. This logic can provide security in token transfers, other application will need their own logic in order to maintain global invariants, without fully trusting all zones in the network.

After the transaction was successfully or unsuccessfully executed on the Hub, we want to relay back a receipt of the transaction (along with proof) back to zone X to complete the cycle. This receipt is just another type of IBC packet and performed in the same manner as the send. The only difference is that executing the receipt on the originating zone must never cause an error. If the transaction was successful, then zone X will destroy the 30 winks held in escrow and the tokens have successfully moved zones. If the transaction was rejected for whatever reason, the escrow is released back to Alice's account as if nothing had happened.

This means that nothing can touch the escrow on zone X until a response (success or failure) is received. And sent tokens cannot be released unless we have proof the packet was rejected by the receiving chain. No tokens are created or destroyed, and no double-spend can be performed, nor are tokens ever lost in some inter-chain ether. The sender will just have to wait for two packets to be relayed and executed (which could be a dozen seconds in many cases). Consideration of handling hard-foking zones (such as ETH/ETC or BTC/BCC) is covered in the advanced section below.

3 Message Passing Semantics

When seeking to scale a blockchain, one must find a way to increase the number of parallel writes that are possible without violating any security guarantees. Preventing double-spends requires strictly serializable access (read and write) to any given account, but we need some secure way to allow

multiple transactions to run at once without any potential for maliciously exploiting race conditions.

With the IBC protocol, we seek to avoid the problem of allowing multiple independent nodes to apply transactions to the same state space. This problem is difficult even without byzantine actors if you don't want to lose any data (eg. strong eventual consistency), and in the face of malicious actors seeking to exploit any inconsistencies for their own gain, it becomes extremely difficult. The most mathematically sound approach for securely reconciling various partial orderings into a consistent global ordering is CRDTs, which ensure all alternative partial orderings of a fixed set of transactions will converge to the same result. CRDTs are indeed very interesting, but by their nature do not allow for enforcing invariants that are essential in blockchain use cases (such as that an account balance may never be negative).

Another solution to this problem is to use sharding, where each shard has access to only part of the state space. This can increase throughput, but also makes any transaction that touches multiple shards extremely difficult to perform correctly. Queries that touch multiple shards and need a consistent view can be performed by using snapshots, but this can be difficult in a highly distributed environment. If you want to securely, atomically query and modify data on two shards, you need something like locking and three-phase commit, which is used in many databases. However, this is a blocking action if you want to guarantee ordering and also introduces synchronicity and timing assumptions in the communication layer, which make it unsuitable to a distributed (multi-)blockchain context.

We take a different approach by defining zones. Each zone is an independent blockchain, with its own application logic, and with its own transactions and data store. Zones should be separated along usage lines, so the vast majority of transactions only affect one zone, but we can guarantee secure, non-blocking semantics for any cross-chain transactions. Each zone is a complete, independent system, and they communicate using messaging. They can guarantee the correct ordering and execution of transactions in their local zone, while allowing parallel execution of transactions in other zones. The only items that require global ordering are messages sent between systems.

Messaging in distributed systems is a deeply researched field and a primitive upon which many other systems are built upon. We can model asynchronous messaging, and make no timing assumptions on the communication channels. By doing this, we allow each zone to move at its own speed, unblocked by any other zone, but able to communicate as fast as the network allows at that moment.

Another benefit of using message passing as our primitive, is that the receiver is able to apply its own security checks on the incoming message. Just because one zone sends a message to add 50 ETH to a given account, and we know this zone, doesn't mean we have to add the balance. We can add our own business logic upon receiving the message to decide whether we want to reject the message, and how we want to act upon it if we accept it. This is very difficult to impossible to achieve in a shared-state scenario. Message passing allows each zone to ensure its security and autonomy, while simultaneously allowing the different systems to work as one whole. This can be seen as an analogue to a micro-services architecture, but across organizational boundaries.

To build a useful algorithms upon a provable asynchronous messaging primitive, we need to define a few higher order constructs to be shared among all systems, that allow us to work with some easier guarantees.

3.1 Reliable Messaging Queue

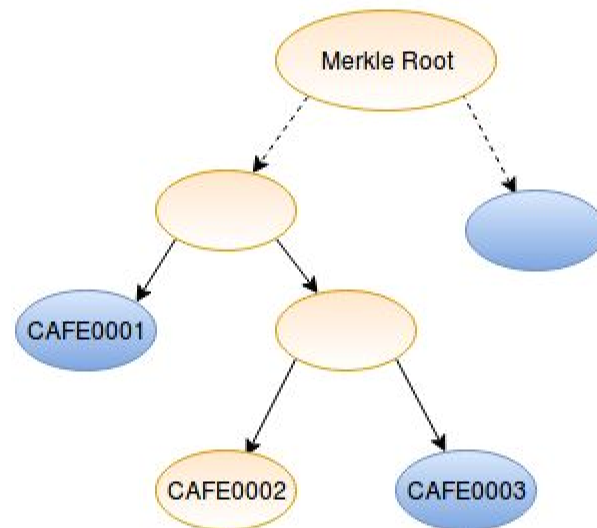
The first primitive we introduce here is a reliable messaging queue (hereafter just referred to as a queue), typical in asynchronous message passing, to allow us to guarantee a causal ordering, and avoid blocking.

This queue is persisted as multiple key-value pairs in the merkle data store of each blockchain. Each queue has a unique prefix and keys are created by appending an 8-byte big endian integer representing the sequence number to that prefix. Note that this encoding provides us a lexicographical order of the keys consistent with their sequence number and allows us to quickly find the latest packet, as well as prove the content of a packet (or lack of packet) at a given sequence. (Assuming access to range proofs as in our IAVL+ merkle tree[3])

You can only add a packet that is exactly one sequence higher than the current highest packet. To make proofs easier, we will also store the sequence number inside the packet itself (the value of the key-value pair). Once a packet is written it must be immutable (except for deleting during cleanup, see below). This allows the receiving zone to accept a proof for packet Z at zone past height H, and rely on it still existing in the same state in the source zone when the receiving zone processes the transaction (asynchronicity requirement). The blockchain application logic must guarantee these constraints for all queues, to ensure proper functioning to the messaging layer.

[3] https://github.com/tendermint/iavl/

If our IBC queue has the prefix 0xCAFE, we can see how multiple packets are stored in the merkle tree with their sequence number appended, so that we can easily provide a merkle proof for the existence (or non-existence) of a given packet. Below we illustrate the path for constructing the proof for the packet sequence number 2, highlighted in orange.



The remainder of this section defines basic message types that can be sent over this queue.

3.2 Registering Chains (permissioned action)

Before we can make a connection between two chains, we need to register them with one another. This is especially important, as all light client proofs require a seed of trust that is vital to verifying all headers, and allows it to determine the true chain in the presence of malicious forks not approved by the consensus algorithm.

To form a connection between A and B, we must register A on B, as well as B on A. These are symmetric processes, so we just describe registering B on A in this case. Upon registration, A adds a trusted header and validator set for B into its data store in a secure location. This is used for the verification of all future headers from B, as well as the validator set changes. A also creates two queues with different names (prefixes).

- *ibc:<chain id of B>:out* - all outgoing packets destined to chain B

- *ibc:<chain id of B>:in* - all incoming packets from chain B along with their result of execution

Note that registering a chain should be a permissioned action and performed with human verification. This assigns trust that this header and validator set represent the proper state of the chain id, and they don't belong to some shadow chain trying to mirror the real chain. Unlike the PoW longest-chain-wins algorithm, trust must be explicitly assigned at one point in proof-of-stake.

Furthermore, the registration packet must also define the algorithm we use to verify the merkle proofs used in the IBC packets on these queues. By default this is the merkle proof format from our IAVL tree, but other merkle-ized data stores, such as the Patricia trie, could be supported. This must be set once on registration, and the algorithm must be supported on the participating chain. This algorithm is consistently used for the lifetime of the queue to verify each packet or receipt against the trusted headers.

3.3 Validator Changes

The set of validators in any production chain should be expected to evolve over time. When we register a new chain, we register a statement of trust to a specific set of validators (more specifically we bind our trust to the condition of a header possessing a certain threshold of signatures matching a given set of public keys). When this set of keys we should trust changes, we need a message to notify the other chain of this change. Since validator changes are critical to the consensus algorithm, this information is stored in a standard format in the Tendermint block headers.

We define a special update packet that contains a Tendermint header and the new validator set, which we can send to the receiving chain. The receiving chain can verify if the purported change to the validator set is valid and use that to verify all future packets.

3.4 Sending a Packet

Sending an IBC packet involves the blockchain application logic calling the IBC module with a packet it wants to send, along with the destination chain id. If the chain was properly registered already, the IBC module simply calculates the next sequence number and adds it to the *out queue*. The sequence numbers are specific to the connection and are generated by the sending chain. They must be monotonically increasing and continuous.

This packet is written to our merkle-ized data store, so it can be embedded in proofs that are tied to block headers. These proofs of a key-value pair representing this packet then are passed around to other chains.

The blockchain application must limit who can write to the key-space of a queue, so arbitrary smart contracts cannot just write a transaction there, but packets are only generated after the blockchain logic determines they are a valid packet to go out, and maintain the global invariants relative to that packet type (eg. the escrow freezing in the example above).

3.5 Relaying a Packet

In order to get the packet to the other chain, we rely on one or more relays to post the *out proof* on chain A to the *in queue* on chain B. Since these only need light client proofs, the relay doesn't need to be a validator or even a full-node. Indeed, it is ideal if each user initiates the creation of an IBC packet also relays it to the recipient chain. The only constraint is that the relay must be able to pay the appropriate fees on the destination chain.

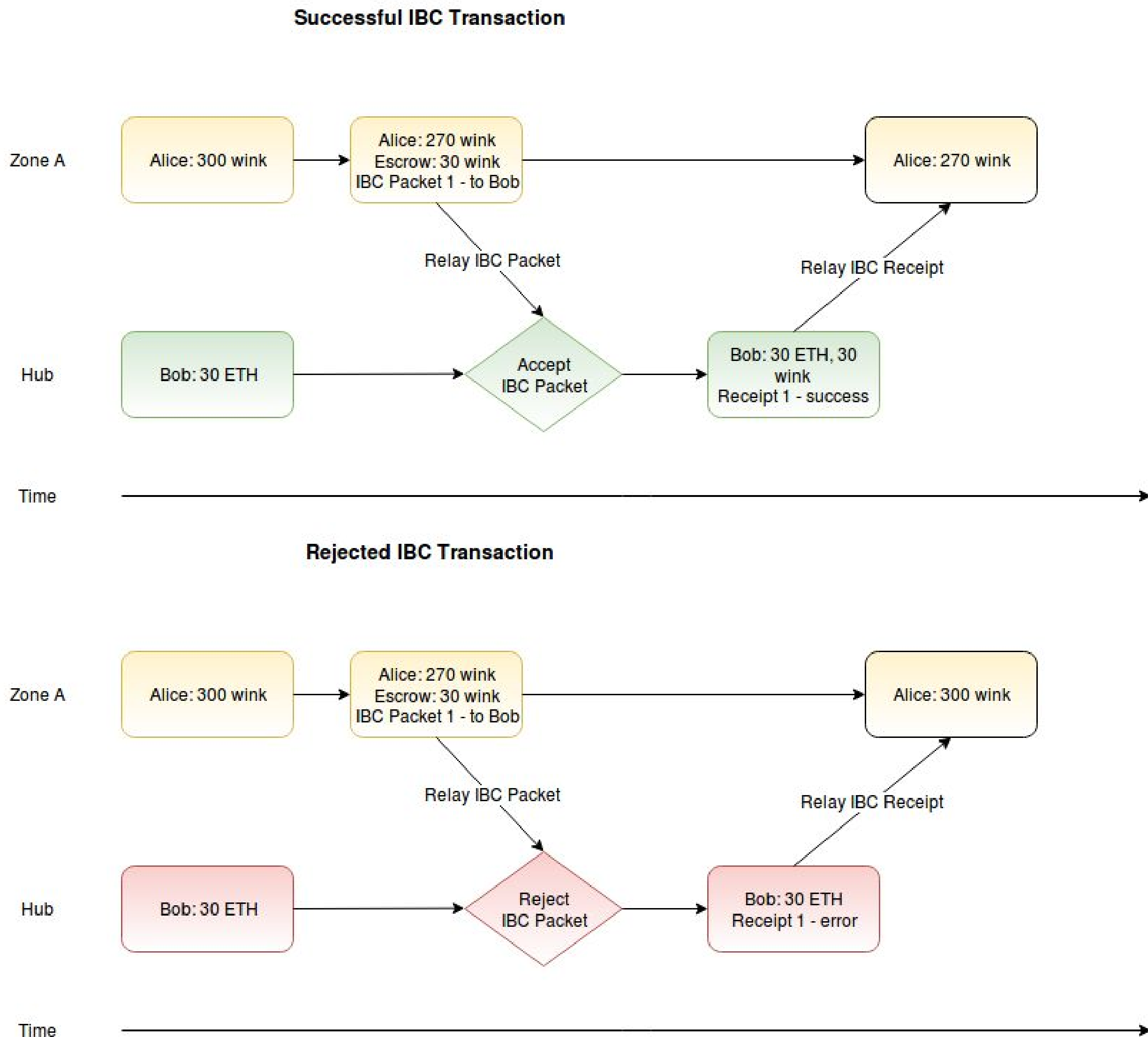
In order to bootstrap a system, the chain developers can provide a special account with enough money to relay, and pay the fees for all packets, until users have tokens on both chains. However, people using IBC should be responsible for paying these fees themselves. Note that updating a header is a costly transaction (ca. 100 signature verifications), while posting a proof for a known header is much cheaper (ca. 20 hash calculations). Thus, to optimize, many packets can be bundled together and sent at the same height, even if they were created at different heights, which can save computational cost by adding a bit of latency.

The system must safely allow multiple relays to run in parallel, rejecting any duplicate posts. But also ideally preventing that they attempt many duplicate posts in the first place, as they waste bandwidth and compute time on the chains.

3.6 Receipts

When an IBC packet is posted to another chain and it is considered valid (eg. the proof matches a known header for the source chain), we must store it in the *in queue* whether or not the execution was successful, so there is proof it was processed. The transaction should be sent to the appropriate smart contract to validate and execute it, then return a standard ABCI result (success or error). The received packet along with the result is written to the *in queue*.

Another relay process may then take a proof that this packet was processed on chain B and post that to chain A as a receipt. This receipt will be handled by chain A to trigger further application logic, as well as clean up the *in queue* (see advanced section). The same process that relays the request from A to B can relay the response from B to A, but it is possible to use another process as well.



4 Light Client Proofs

The basis of IBC is the ability to perform efficient light client proofs on-chain with minimal data transfer required. Furthermore, the proof of a

valid header must be completely defined by a deterministic set of data (eg. a header and the signatures), which can be written to the blockchain, and not depend on any externalities (like “is this currently part of the longest fork”). This makes it possible for a new node to replay the entire chain and also validate the validity of all IBC packets and the current state of the connection without any external dependencies.

It would be possible to post all headers of the remote chain to the receiving chain, however, the computational and storage requirements would be considerable for the blockchain, and maintaining connections with 20 zones in parallel (as is the design of the Cosmos Hub) would be prohibitively expensive. Anyone looking to support IBC with a different BFT engine is welcome to get in touch with the Tendermint core team to work out the details.

While not specific to IBC, light-client proofs provide cryptographically secure verification of IBC packets that allow us to build trusted message queues. A less technical presentation of light client proofs can be found on the Cosmos blog.

4.1 Verifying Block Headers

If we trust the validator set, we can validate the header simply by verifying the that over 2/3 of the voting power in that set properly signed a commit for that header. Since the validator set hash is in the header, we can quickly verify if this header defines the same validators set as we trust, and in that case, can ignore any intervening headers.

This is only safe if we have a valid signed header by the trusted validator set that is within the unbonding period for staking. In that case, if we were given a false (forked) header, we could use this as proof to slash the stake of all the double-signing validators. This is the same security guarantee of any non-validating full node as well, so even in the presence of some ultra-powerful malicious actors, this makes the cost of faking a proof equal to at least one third of all staked tokens, which should be significantly higher than any short-term gain.

4.2 Dynamic Validator Sets

If the validator hash is different than the trusted one, we must both verify is the change is valid, as well as use the new set to validate the header. Since the entire validator set is not provided by default when we give a header and commit votes, this must be provided as extra data to the certifier.

A validator change in Tendermint can be securely verified with the following checks:

- 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

In that case, we can update to this header, and update the trusted validator set, with the same guarantees as above (the ability to slash at least one third of all staked tokens on any false proof).

4.3 Merkle Proofs

If we trust a header, then we have a known AppHash at a given height. With that known hash, we can verify any merkle proof from the data store. That is, if we have a series of hashes leading from a given key-value pair up to the trusted AppHash, then we can trust that this data actually belonged to this chain at the given height.

There are different possible merkle tree implementations and each one will have a different proof format with a different verification logic. We should set the algorithm we use to verify the key-value pairs when we initiate our trust to a chain. As long as we can provide a smallish (ie. less than 2KB or so) proof tying any key-value pair to a trusted header, and we provide a portable and lightweight library for verifying the proofs, we can use the data store with IBC.

This sequence of proofs (validators, headers, data) that allow us to validate the presence of a key-value pair with minimal data transfer forms the basis for the entire IBC protocol. The end result of all these proofs is that we are able to cryptographically verify a key-value pair stored in one chain at a given height. We can also prove that there is no value at a given key, which is important for some structures we will see later. If the data store allows us to do range queries (all k-v pairs between A and B) or first/last in range, we can make many of the below operations more efficient.

5 Advanced Messages

The above sections describe a secure protocol that can handle all normal situations between the chains, both success and error cases. However, for

this to work on a long-running blockchain with many possible failure modes on both sides of the connection, we have to extend the protocol. We can add features such as secure timeout, securely cleaning up old messages in the queue, handling hard forks on the connected zone, and relaying messages over multiple zones for scalability.

5.1 Timeouts

Sometimes it is desirable to have some timeout, an upper limit to how long you will wait for a transaction to be processed before considering it an error. At the same time, this is an obvious attack vector for a double spend, just delaying the relay of the response or waiting to post the packet in the first place and then relaying it right after the cutoff to take advantage of any clock skew.

One solution to this is to include a timeout in the IBC packet itself. When sending it, one can specify a block height (or header timestamp?) on the receiving chain after which it is no longer valid. If the packet is posted before that height, it will be processed normally. If it is posted after that height, it will be a guaranteed error. Note that to make this secure, the timeout must be relative to a condition on the receiving queue.

Assume that both chains are running well, but no one wants to pay the fees to relay the headers and packets for IBC. Carl sent a packet on zone A some time ago, set to expire at height H. Now he queries the hub for his packet at height H' (> H). He produces a proof of non-existence that his packet was never received by the hub at height H' and posts this to the sending zone A. Zone A assert that that if this packet was posted at any H'' > H', it would be rejected by the hub and result in an error. Thus, zone A can safely handle this as an error case and unfreeze Carl's tokens held in escrow.

Adding reasonable time outs to all packets allows us to gracefully handle any errors with the IBC relay processes, or a flood of unrelayed "spam" IBC packets. Please note that this requires that the destination chain is running and reachable. One can prove nothing in a complete network partition, and must wait to connect. The timeout must be proven on the recipient chain, not simply the absence of a response on the sending chain.

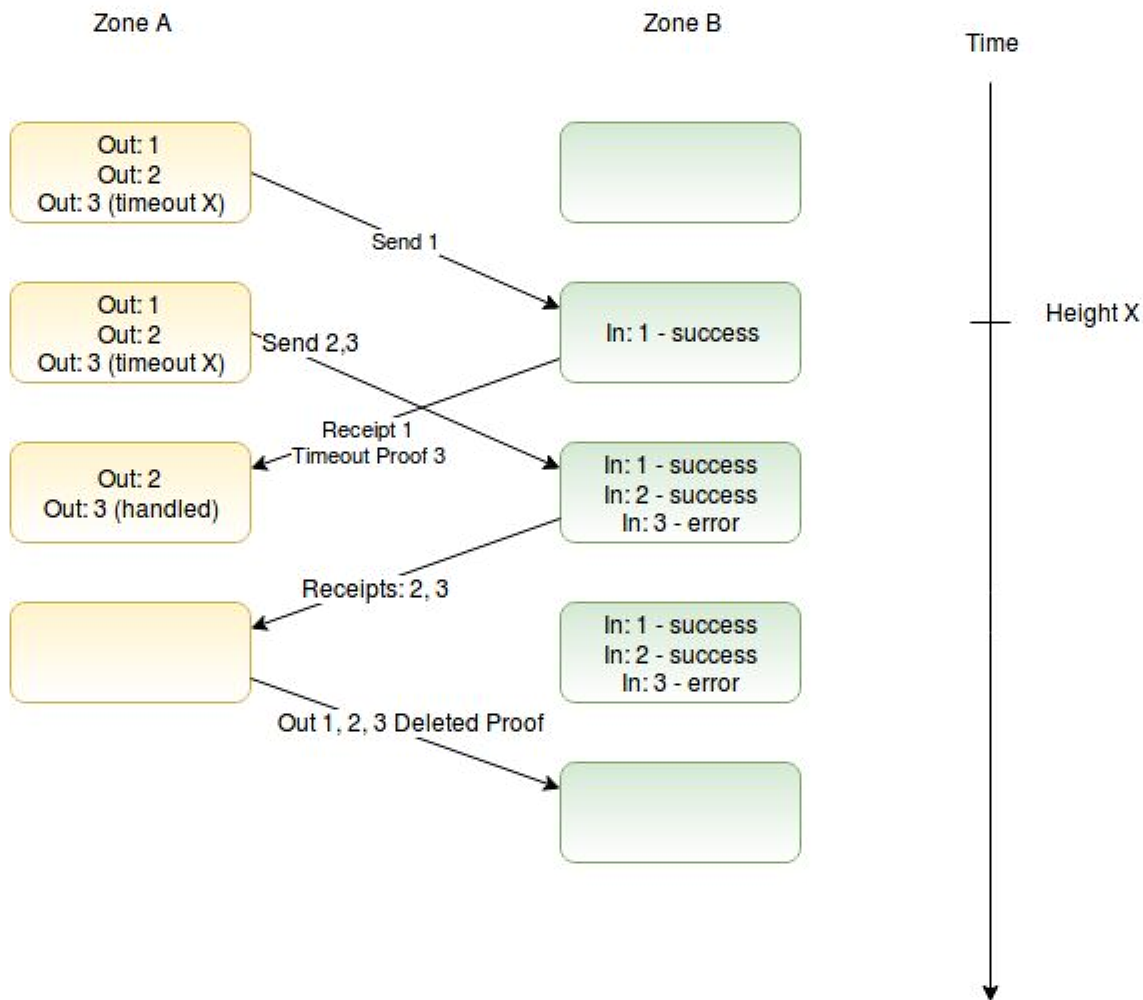
5.2 Clean up

If left to run indefinitely, these queues could grow without limit and create a major storage requirement on the chains. However, if we delete packets too early, we can lose potentially important information that has not yet been relayed and sacrifice reliability. Thus, we need an algorithm that will

delete information from the queue only once we can prove it was properly relayed to the other chain.

Once a receipt comes back, we can delete the packet from the *out queue* as soon as the receipt was processed. It must be the first packet at this point, as the other receipts are only processed if they arrived in order. Timeouts may return errors out of order, but they should just mark the packet as "handled" and wait for a proper receipt of a higher sequence number in order to clean them up.

On the receiving side, we need to clean up the *in queue* as well. We can add one more transaction type to relay information that the receipt was received. Since posting a receipt to the source chain triggers the deletion of the corresponding packet in the *out queue*, we can use a proof of non-existence of a packet in the *out queue* of the source chain as grounds to delete the matching packet from the *in queue* of the destination chain (it clearly must have existed previously for it to have been in the *in queue* at all).



5.3 Disconnecting (privileged transaction)

At some point, we may wish to remove a connection between two chains. This can be due to a lack of interest between the two chains (no use of IBC), a change of network topology (routing over the hub or directly), or a lost of trust between the chains (suspicion that one chain is not destroying tokens as promised).

Disconnecting from another chain has two components. The first is properly handling all in-process messages without leading to an inconsistency, and the second is resolving any left over state after the disconnect (does the DEX still have a credit on the hub, what do we do with the money that got sent there but never returned?). While the first issue can be solved through technical means, once the disconnection is complete, the

resolution of any outstanding promises must be dealt with by on-chain governance.

The first step to disconnect is to prevent any more packets from being sent. We can add a special “disconnect” packet to the end of the *out queue*, and reject any more attempts to send. We also return errors on any new incoming packets. This disconnect packet should be relayed to the other chain, so it can trigger the same process and disconnect both sides gracefully.

There may well be several packets that were properly sent, and still waiting on a receipt. The normal case would be to await these receipts and process them as normal according to our business logic, only removing the connection completely once these receipts have been received. This will leave us with a consistent state between the two chains after shutdown that is easier to resolve via governance.

If there is proof of malicious activity or belief there is some exploit in the application logic handling IBC, we can add a “hard shutdown” flag to immediately handle all outstanding packets as if there was an error (like in the timeout case), and immediately shut down the connection

5.4 Handling Forks (privileged transaction)

The IBC protocol follows the remote zone passed on the consensus rules. Whatever the super-majority (> ⅔) of the remote validators say is the state is the state. If there is a hard fork (non backwards compatible change) that all validators agree upon, then the light client protocol used in IBC will follow this chain without problem. However, there are rare cases when this is impossible or undesirable.

If there is a big divide in the remote chain and they split eg. 60-40 as to the direction of the chain, then the light-client protocol will refuses to follow either fork. If both sides declare a hard fork and continue with new validator sets that are not compatible with the consensus engine (they don’t have ⅔ support from the previous block), then users will have to manually tell their local client which chain to follow (or fork and follow both with different IDs).

The IBC protocol doesn’t have the option to follow both chains as the accounting metrics and value of tokens would be thrown off. We add a privileged transaction, so that through on-chain governance the IBC connection can be set to follow one of the two chains. The chosen remote chain would retain its connection to us without issue. If we wished to

support both, we should manually add a separate connection for the split chain (eg. ETC) and provide a different source token and trust levels for it.

5.5 Multi-chain routing

When we have many chains, it becomes inefficient to connect them all directly to one another. With 20 chains, each chain must maintain 40 IBC queues, and we will need 190 or O(N^2) relay processes to move all the messages. To make this scale beyond a handful of chains, we need to allow some chains to relay others. The simplest topology is a star configuration (Cosmos hub), where the 20 chains would all connect to the hub. The hub must maintain the 40 queues, the others only 2. And we would only need 20 or O(N) relay processes.

However, once we allow relaying, or indirect connections, there becomes the issue of extending trust. Is trust fully transitive (A trusts B and B trusts C, so A trusts C)? Or can we add some limits? There also arises the issue of path finding, how one chain knows the queue in which it should write a message if it is not directly to the destination chain.

5.5.1 Gateway

The issue of pathfinding has been solved by many successful algorithms already, and it makes sense to build upon their concepts. The basis of IP is that a packet is relayed multiple times to its end destination. This is solved by having a routing table of all direct connections, as well as a gateway, which may route to a given subnet, or serve as a default gateway to any address that was not otherwise registered. We will replace IP addresses with chain ids, but retain the general architecture.

By default any IBC connection between two chains is a direct connection. Any packet destined to that given chain id will be sent to that IBC queue. We can also register that IBC connection as a gateway to a list of chain ids (eg. “ethermint,dex”), a subset of the chain ids using a prefix (eg. “cosmos-*”, or as a default for any unknown chain id.

When we send a packet, we first check for direct connections, then for a gateway explicitly listing this chain, then one listing the prefix, then the default gateway, rejecting the packet if there is no match and no default gateway. When receiving a packet on an IBC queue, we verify that it is directly sent from the chain we are connected to (which is used for all proofs), as well as that the listed original source would be routed to this same queue upon send (only one correct entry for each packet).

As a poor network setup could lead to infinite loops, we need to automatically drop any packet that has been routed in a circle (if it ends up on the same chain again). We also need to makes sure the results get routed all the way back to the original source with the same algorithm. The second part depends on the type of relay we perform.

Any chain can perform any type of relay, and in fact, may perform different functions depending on the content of the packet being relayed.

5.5.2 Blind Relay

A blind relay doesn't do any introspection of the contents of the packets, just looks up the source and destination chains and routes it one step closer to it. In this case the hub just testifies to chain B that it received the packets from chain A. This trust relationship is fine for one hop, but if extended to many hops, means the trust that this packet was correctly relayed is the minimal trust in all the hubs involved in relaying it, as any one of them could lie. (That said, the trust level for a blockchain, and even more a hub chain, should be extremely high).

In this case, the trust is extended in the same manner as the routing gateways. If I trust the Cosmos hub to relay any "cosmos-*" chains, I will trust that any packet it sends me from "cosmos-dex" is actually from the Cosmos dex. But I will not trust that the message it sends me from "polka-core" is really from polkadot. If I was interested in communicating with polka-core, I would have a separate IBC queue set up for that purpose.

In this case, any result of a transaction is relayed back to the source by reversing the algorithm. We note that the trust relations mirror the routing setup, so this should not introduce any problems.

Since a blind relay requires no knowledge of the packet contents, it is the default response to a packet if no other relay type is set. One example of this is if we have an escrow in "cosmos-dex" that can only be released by a message from "cosmos-oracle" chain, the hub can relay the message from "cosmos-oracle" to "cosmos-dex" without being aware of the application logic they are using, which allows the zones to innovate communication mechanisms quickly without requiring explicit support on the hub (which should be conservative in order to maximize security).

5.5.3 Vouching Relay

A vouching relay must understand the packet being relayed, and will execute it locally, forwarding it on only once it has determined it is valid. In this way, we can increase the trust level in multi-hop messages, as each hub

must fully verify the message before forwarding it to the next hop. We also allow special application logic to respond to the packet.

One particular use of this type of relay is in the SendTx message, which is the original transaction type we will support via IBC. If we have a blind relay, we can forward messages from any chain to any other to send money, but each chain will have to keep an account with each other chain to determine if they will accept this payment from that chain (is it the issuer of this token? do I have a prior balance with them? etc.).

In order to simplify this, the hub will serve the role of a clearing house, maintaining balances among all chains. All zones should accept any payment from the hub, that the hub has deemed proper. The hub will keep a balance of total-in and total-out for all payments into and out of any zone attached to it. It can also fixed amounts of tokens to be minted by certain zones (eg. CETH from the ethermint zone).

To make this clearer, consider an example. One user sends 100 CETH from ethermint to the dex. The hub verifies that ethermint can make this payment, and sends the tokens along, noting a balance of 100 CETH on the dex. Later on another user on the dex sends 50 CETH to zone X. Zone X and the dex have never had any transactions and if sent via a blind relay, zone X would reject this payment. Making the dex significantly less useful. However, as the hub is a vouching relay for the SendTx transactions, it can verify that, yes, the dex is sending out fewer tokens than were sent in, and it vouches for this transaction to zone X.

We note that in many specific cases, a vouching relay can add significant utility as well as trust to the routing.

It also requires that every transaction is executed in each intermediate zone, and care must be taken to properly relay the result of those transactions all the way to the source zone, regardless of where it broke. Since a relay from A -> hub 1 -> hub 2 -> B would be comprised of three separate transactions, if no further thought was made, the act of B rejecting the message from hub 2 would never be relayed back to hub 1, much less zone A. Thus, while relaying vouched messages, we must maintain the entire return path as part of the message, which each vouching relay appends to when creating a new message to pass on.

5.5.4 Matchmaker

In some cases we don't even want the relays for every message, but just to leverage the trust relations established over these paths in order to more easily connect to new zones. This is particularly true for light clients, but

also for zone discovery when two zones will have much communication between each other and want to avoid the overhead of routing. (Since the relay handles all the network connections, relaying IBC messages has to do with trust relations, not network connectivity).

Let us assume that zone A and zone B both have trust relations with the hub. Since the list of known zones, as well as current headers and validator sets are all stored in the merkle data store f the hub, zone A could query the hub for a valid seed of trust for zone B, and use that to register an IBC connection. This can be used to largely automate the initial seeding of trust, which is the weakest link of IBC communication, and generally must be approved by on-chain governance, to require human verification of the validity of the header and the validator set.

Likewise, a light-client that trusts zone A, could use the matchmaker functionality to get trust of the hub, and then zone B, allowing it to later interact directly and securely with zone B, without requiring any other manual trust verification step. Since the assignment of trust on one header requires manual validation, allowing secure automatic discovery of the validator sets of all attached zones is a big benefit of any client to the Cosmos Hub, allowing the light client to seamlessly expand to query an chain in the entire network.

6 Conclusion

We have demonstrated a secure, performant, and flexible protocol for connecting multiple blockchains in the Cosmos ecosystem with various network architectures. This builds upon previous design and implementation of the basic components, and seeks to define an industrial strength protocol as secure as the blockchain itself.

We have so far concentrated on one transaction type, which is the transfer of tokens across zones. This is used as the mapping of local invariants to global invariants can be solved rather simply with the concept of credit. We will extend this paper in a future version to provide a framework for designing new application-specific transaction types for IBC and verifying their correctness.

Much of the code in the first four sections is already implemented in go as part of the Cosmos SDK, and we are currently documenting the packet types and reference implementation to enable developers to build Cosmos IBC compatible smart contracts in languages other than go.