Cosmos Inter-Blockchain Communication (IBC) Protocol

Abstract

This paper specifies the Cosmos Inter-Blockchain Communication (IBC) protocol. The IBC protocol defines a set of semantics for authenticated, strictly-ordered message passing between two blockchains with independent consensus algorithms.

The core IBC protocol is payload-agnostic. On top of IBC, developers can implement the semantics of a particular application, enabling users to transfer valuable assets between different blockchains while preserving the contractual guarantees of the asset in question - such as scarcity and fungibility for a currency or global uniqueness for a digital kitty-cat.

IBC requires two blockchains with cheaply verifiable rapid finality and Merkle tree substate proofs. The protocol makes no assumptions of block confirmation times or maximum network latency of packet transmissions, and the two consensus algorithms remain completely independent. Each chain maintains a local partial order and inter-chain message sequencing ensures cross-chain linearity. Once the two chains have registered a trust relationship, cryptographically verifiable packets can be sent between them.

IBC was first outlined in the Cosmos Whitepaper, and later described in more detail by the IBC specification paper. This document supersedes both. It explains the requirements and structure of the protocol and provides sufficient detail for both analysis and implementation.

Contents

1 Overview

(Back to table of contents)

1.1 Summary

The IBC protocol creates a mechanism by which two replicated fault-tolerant state machines may pass messages to each other. These messages provide a base layer for the creation of communicating blockchain architecture that overcomes challenges in the scalability and extensibility of computing blockchain environments.

The IBC protocol assumes that multiple applications are running on their own blockchain with their own state and own logic. Communication is achieved over an ordered message queue primitive, allowing the creation of complex inter-chain processes without trusted third parties.

The message packets are not signed by one psuedonymous account, or even multiple, as in multi-signature sidechain implementations. Rather, IBC assigns authorization of the packets to the source blockchain's consensus algorithm, performing light-client style verification on the destination chain. The Byzantine-fault-tolerant properties of the underlying blockchains are preserved: a user transferring assets between two chains using IBC must trust only the consensus algorithms of both chains.

In this paper, we define a process of posting block headers and Merkle tree proofs to enable secure verification of individual packets. We then describe how to combine

these packets into a messaging queue to guarantee ordered delivery. We then explain how to handle packet receipts (response/error) on the source chain, which enables the creation of asynchronous RPC-like protocols on top of IBC. Finally, we detail some optimizations and how to handle Byzantine blockchains.

1.2 Definitions

Blockchain - A replicated fault-tolerant state machine with a distributed consensus algorithm. The smallest unit produced through consensus is a block, which may contain many transactions, each applying some arbitrary mutation to the state.

Module - We assume that the state machine of each blockchain is comprised of multiple components that have limited rights to execute some particular set of state transfers (these are modules in the Cosmos SDK or smart contracts in Ethereum).

Finality - The guarantee that a given block will not be reverted within some predefined conditions of a consensus algorithm. All proof-of-work systems offer probabilistic finality, which means that the difficulty of reverting a block increases as the block is embedded more deeply in the chain. Many proof-of-stake systems offer much weaker guarantees, based only on the honesty of the block producers. BFT algorithms such as Tendermint guarantee complete finality upon production of a block (unless over two thirds of the validators collude to break consensus, in which case the offenders can be identified and punished - further discussion of that scenario is outside the scope of this document).

Attributable - Knowledge of the pseudonymous identity which made a statement, whom we can punish with some deduction of value (slashing) if the statement is false. Synonymous with accountability.

Unbonding period - Proof-of-stake algorithms need to lock the stake (prevent transfers) for some time to provide a lower bound for the length of a long-range attack [3]. Complete finality is associated with a subset of the proof-of-stake class of consensus algorithms. We assume the proof-of-stake algorithms utilized by the two blockchains have some unbonding period P.

1.3 Threat Models

False statements - Any information we receive may be false.

Network partitions and delays - We assume an asynchronous, adversarial network with unbounded latency. Network messages may be modified, reordered, duplicated, or selectively dropped. Actors may be arbitrarily partitioned by a powerful adversary. The IBC protocol favors correctness over liveness where applicable.

Byzantine actors - An entire blockchain may not act according to protocol. This must be detectable and provable, allowing the communicating blockchain to revoke trust and take necessary action. Application-level protocols designed on top of IBC should consider and mitigate this risk in a manner suitable to their application.

2 Connections

(Back to table of contents)

The basis of IBC is the ability to verify in the on-chain consensus ruleset of chain `B` that a data packet received on chain `B` was correctly generated on chain `A`. This establishes a cross-chain linearity guarantee: upon validation of that packet on chain `B` we know that the packet has been executed on chain `A` and any associated logic resolved (such as assets being escrowed), and we can safely perform application logic on chain `B` (such as generating vouchers on chain `B` for the chain `A` assets which can later be redeemed with a packet in the opposite direction).

This section outlines the abstraction of an IBC *connection*: the state and consensus ruleset necessary to perform IBC packet verification.

2.1 Definitions

- Chain `A` is the source blockchain from which the IBC packet is sent
- Chain `B` is the destination blockchain on which the IBC packet is received
- `H_h` is the signed header of chain `A` at height `h`
- `C_h` is a subset of the consensus ruleset of chain `A` at height `h`
- `V_kh` is the value stored on chain `A` under key `k` at height `h`
- `P` is the unbonding period of chain `P`, in units of time
- `dt(a, b)` is the time difference between events `a` and `b`

Note that of all these, only `H_h` defines a signature and is thus attributable.

2.2 Requirements

To facilitate an IBC connection, the two blockchains must provide the following proofs:

1. Given a trusted `H_h` and `C_h` and an attributable update message `U_h`, it is possible to prove `H_h'` where `C_h' == C_h` and `dt(now, H_h) < P`
2. Given a trusted `H_h` and `C_h` and an attributable change message `X_h`, it is possible to prove `H_h'` where `C_h' /= C_h` and `dt(now, H_h) < P`
3. Given a trusted `H_h` and a Merkle proof `M_kvh` it is possible to prove `V_kh`

It is possible to make use of the structure of BFT consensus to construct extremely lightweight and provable messages `U_h'` and `X_h'`. The implementation of these requirements with Tendermint consensus is defined in Appendix E. Another algorithm able to provide equally strong guarantees (such as Casper) is also compatible with IBC but must define its own set of update and change messages.

The Merkle proof `M_kvh` is a well-defined concept in the blockchain space, and provides a compact proof that the key value pair `(k, v)` is consistent with a Merkle root stored in `H_h`. Handling the case where `k` is not in the store requires a separate proof of non-existence, which is not supported by all Merkle stores. Thus, we define

the proof only as a proof of existence. There is no valid proof for missing keys, and we design the algorithm to work without it.

Blockchains supporting IBC must implement Merkle proof verification:

```
valid(H_h, M_kvh) ⇒ true | false
```

2.3 Connection Lifecycle

2.3.1 Opening a connection

All proofs require an initial `H_h` and `C_h` for some `h`, where `dt(now, H_h) < P`.

Establishing a bidirectional initial root-of-trust between the two blockchains (`A` to `B` and `B` to `A`) — `H_ah` and `C_ah` stored on chain `B`, and `H_bh` and `C_bh` stored on chain `A` — is necessary before any IBC packets can be sent.

Any header may be from a malicious chain (e.g. shadowing a real chain state with a fake validator set), so a subjective decision is required before establishing a connection. This can be performed permissionlessly, in which case users later utilizing the IBC channel must check the root-of-trust themselves, or authorized by on-chain governance for additional assurance.

2.3.2 Following block headers

We define two messages `U_h` and `X_h`, which together allow us to securely advance our trust from some known `H_n` to some future `H_h` where `h > n`. Some implementations may require that `h == n + 1` (all headers must be processed in order). IBC implemented on top of Tendermint or similar BFT algorithms requires only that `delta-vals(C_n, C_h) < ⅓` (each step must have a change of less than one-third of the validator set)[4].

Either requirement is compatible with IBC. However, by supporting proofs where `h - n > 1`, we can follow the block headers much more efficiently in situations where the majority of blocks do not include an IBC packet between chains `A` and `B`, and enable low-bandwidth connections to be implemented at very low cost. If there are packets to relay every block, these two requirements collapse to the same case (every header must be relayed).

Since these messages `U_h` and `X_h` provide all knowledge of the remote blockchain, we require that they not just be provable, but also attributable. As such, any attempt to violate the finality guarantees in headers posted to chain `B` can be submitted back to chain `A` for punishment, in the same manner that chain `A` would independently punish (slash) identified Byzantine actors.

More formally, given existing set of trust `T = {(H_i, C_i), (H_j, C_j), …}`, we must provide:

```
valid(T, X_h | U_h) ⇒ true | false | unknown
```

`valid` must fulfill the following properties:

```
if H_h-1 ∈ T then
  valid(T, X_h | U_h) ⇒ true | false
  ∃ (U_h | X_h) ⇒ valid(T, X_h | U_h)

if C_h ∉ T then
  valid(T, U_h) ⇒ false
```

We can then process update transactions as follows:

```
update(T, X_h | U_h) ⇒ success | failure

update(T, X_h | U_h) = match valid(T, X_h | U_h) with
  false ⇒ fail with "invalid proof"
  unknown ⇒ fail with "need a proof between current and h"
  true ⇒
    set T = T ∪ (H_h, C_h)
```

Define `max(T)` as `max(h, where H_h ∈ T)`. For any `T` with `max(T) == h-1`, there must exist some `X_h | U_h` so that `max(update(T, X_h | U_h)) == h`. By induction, there must exist a set of proofs, such that `max(update…(T,...)) == h + n` for any `n`.

Bisection can be used to discover this set of proofs. That is, given `max(T) == n` and `valid(T, X_h | U_h) == unknown`, we then try `update(T, X_b | U_b)`, where `_b == (h + n) / 2`. The base case is where `valid(T, X_h | U_h) == true` and is guaranteed to exist if `h == max(T) + 1`.

2.3.3 Closing a connection

IBC implementations may optionally include the ability to close an IBC connection and prevent further header updates, simply causing `update(T, X_h | U_h)` as defined above to always return `false`.

Closing a connection may break application invariants (such as fungiblity - token vouchers on chain `B` will no longer be redeemable for tokens on chain `A`) and should only be undertaken in extreme circumstances such as Byzantine behavior of the connected chain.

Closure may be permissioned to an on-chain governance system, an identifiable party on the other chain (such as a signer quorum, although this will not work in some Byzantine cases), or any user who submits an application-specific fraud proof. When a connection is closed, application-specific measures may be undertaken to recover assets held on a Byzantine chain. We defer further discussion to Appendix D.

3 Channels & Packets

(Back to table of contents)

3.1 Background

IBC uses a cross-chain message passing model that makes no assumptions about network synchrony. IBC *data packets* (hereafter just *packets*) are relayed from one blockchain to the other by external infrastructure. Chain A and chain B confirm new blocks independently, and packets from one chain to the other may be delayed or censored arbitrarily. The speed of packet transmission and confirmation is limited only by the speed of the underlying chains.

The IBC protocol as defined here is payload-agnostic. The packet receiver on chain B decides how to act upon the incoming message, and may add its own application logic to determine which state transactions to apply according to what data the packet contains. Both chains must only agree that the packet has been received.

To facilitate useful application logic, we introduce an IBC *channel*: a set of reliable messaging queues that allows us to guarantee a cross-chain causal ordering[5] of IBC packets. Causal ordering means that if packet *x* is processed before packet *y* on chain A, packet *x* must also be processed before packet *y* on chain B.

IBC channels implement a vector clock[2] for the restricted case of two processes (in our case, blockchains). Given $x \rightarrow y$ means *x* is causally before *y*, chains A and B, and $a \Rightarrow b$ means *a* implies *b*:

$A{:}send(msgi) \rightarrow B{:}receive(msgi)$

$B{:}receive(msgi) \rightarrow A{:}receipt(msgi)$

$A{:}send(msgi) \rightarrow A{:}send(msgi{+}1)$

$x \rightarrow A{:}send(msgi) \Rightarrow x \rightarrow B{:}receive(msgi)$

$y \rightarrow B{:}receive(msgi) \Rightarrow y \rightarrow A{:}receipt(msgi)$

Every transaction on the same chain already has a well-defined causality relation (order in history). IBC provides an ordering guarantee across two chains which can be used to reason about the combined state of both chains as a whole.

For example, an application may wish to allow a single tokenized asset to be transferred between and held on multiple blockchains while preserving fungibility and conservation of supply. The application can mint asset vouchers on chain B when a particular IBC packet is committed to chain B, and require outgoing sends of that packet on chain A to escrow an equal amount of the asset on chain A until the vouchers are later redeemed back to chain A with an IBC packet in the reverse direction. This ordering guarantee along with correct application logic can ensure that total supply is preserved across both chains and that any vouchers minted on chain B can later be redeemed back to chain A.

This section provides definitions for packets and channels, a high-level specification of the queue interface, and a list of the necessary proofs. To implement wire-

compatible IBC, chain `A` and chain `B` must also use a common encoding format. An example binary encoding format can be found in Appendix C.

3.2 Definitions

3.2.1 Packet

We define an IBC *packet* `P` as the five-tuple `(type, sequence, source, destination, data)`, where:

`type` is an opaque routing field

`sequence` is an unsigned, arbitrary-precision integer

`source` is a string uniquely identifying the chain, connection, and channel from which this packet was sent

`destination` is a string uniquely identifying the chain, connection, and channel which should receive this packet

`data` is an opaque application payload

3.2.2 Receipt

We define an IBC *receipt* `R` as the four-tuple `(sequence, source, destination, result)`, where

`sequence` is an unsigned, arbitrary-precision integer

`source` is a string uniquely identifying the chain, connection, and channel from which this packet was sent

`destination` is a string uniquely identifying the chain, connection, and channel which should receive this packet

`result` is an opaque application result payload

3.2.3 Queue

To implement strict message ordering, we introduce an ordered *queue*. A queue can be conceptualized as a slice of an infinite array. Two numerical indices - `q_head` and `q_tail` - bound the slice, such that for every `index` where `q_head <= index < q_tail`, there is a queue element `q[index]`. Elements can be appended to the tail (end) and removed from the head (beginning). We introduce one further method, `advance`, to facilitate efficient queue cleanup.

Each IBC-supporting blockchain must provide a queue abstraction with the following functionality:

`init`

```
set q_head = 0
set q_tail = 0

peek ⇒ e

match q_head == q_tail with
  true ⇒ return nil
  false ⇒
    return q[q_head]

pop ⇒ e

match q_head == q_tail with
  true ⇒ return nil
  false ⇒
    set q_head = q_head + 1
    return q_head - 1

retrieve(i) ⇒ e

match q_head <= i < q_tail with
  true ⇒ return q[i]
  false ⇒ return nil

push(e)

set q[q_tail] = e
set q_tail = q_tail + 1

advance(i)

set q_head = i
set q_tail = max(q_tail, i)

head ⇒ i

return q_head

tail ⇒ i

return q_tail
```

3.2.4 Channel

We introduce the abstraction of an IBC *channel*: a set of the required packet queues to facilitate ordered bidirectional communication between two blockchains `A` and `B`. An IBC connection, as defined earlier, can have any number of associated channels. IBC connections handle header initialization & updates. All IBC channels use the same connection, but implement independent queues and thus independent ordering guarantees.

An IBC channel consists of four distinct queues, two on each chain:

`outgoing_A`: Outgoing IBC packets from chain `A` to chain `B`, stored on chain `A`

`incoming_A`: IBC receipts for incoming IBC packets from chain `B`, stored on chain `A`

`outgoing_B`: Outgoing IBC packets from chain `B` to chain `A`, stored on chain `B`

`incoming_B`: IBC receipts for incoming IBC packets from chain `A`, stored on chain `B`

3.3 Requirements

In order to provide the ordering guarantees specified above, each blockchain utilizing the IBC protocol must provide proofs that particular IBC packets have been stored at particular indices in the outgoing packet queue, and particular IBC packet execution results have been stored at particular indices in the incoming packet queue.

We use the previously-defined Merkle proof `M_kvh` to provide the requisite proofs. In order to do so, we must define a unique, deterministic key in the Merkle store for each message in the queue:

```
key = (queue name, head | tail | index)
```

The index is stored as a fixed-length unsigned integer in big endian format, so that the lexicographical order of the byte representation of the key is consistent with their sequence number. This allows us to quickly iterate over the queue, as well as prove the content of a packet (or lack of packet) at a given sequence. `head` and `tail` are two special constants that store an integer index, and are chosen such that their serializated representation cannot collide with that of any possible index.

Once written to the queue, a packet must be immutable (except for deletion when popped from the queue). That is, if a value `v` is written to a queue, then every valid proof `M_kvh` must refer to the same `v`. In practice, this means that an IBC implementation must ensure that only the IBC module can write to the IBC subspace of the blockchain's Merkle store.

Each incoming & outgoing queue for each connection must be provably associated with another uniquely identified chain, connection, and channel so that an observer can prove that a message was intended for that chain and only that chain. This can easily be done by prefixing the queue keys in the Merkle store with strings unique to the chain (such as chain identifier), connection, and channel.

3.4 Sending a packet

To send an IBC packet, an application module on the source chain must call the send method of the IBC module, providing a packet as defined above. The IBC module must ensure that the destination chain was already properly registered and that the calling module has permission to write this packet. If all is in order, the IBC module simply pushes the packet to the tail of `outgoing_a`, which enables all the proofs described above.

The packet must provide routing information in the `type` field, so that different modules can write different kinds of packets and maintain any application-level invariants related to this area. For example, a "coin" module can ensure a fixed supply, or a "NFT" module can ensure token uniqueness. The IBC module on the destination chain must associate every supported packet type with a particular handler (`f_type`).

To send an IBC packet from blockchain `A` to blockchain `B`:

```
send(P{type, sequence, source, destination, data}) ⇒ success | failure

case
  source /= (A, connection, channel) ⇒ fail with "wrong sender"
  sequence /= tail(outgoing_A) ⇒ fail with "wrong sequence"
  otherwise ⇒
    push(outgoing_A, P)
    success
```

Note that the `sequence`, `source`, and `destination` can all be encoded in the Merkle tree key for the channel and do not need to be stored individually in each packet.

3.5 Receiving a packet

Upon packet receipt, chain `B` must check that the packet is valid, that it was intended for the destination, and that all previous packets have been processed. `receive` must write the receipt queue upon accepting a valid packet regardless of the result of handler execution so that future packets can be processed.

To receive an IBC packet on blockchain `B` from a source chain `A`, with a Merkle proof `M_kvh` and the current set of trusted headers for that chain `T_A`:

```
receive(P{type, sequence, source, destination, data}, M_kvh) ⇒ success
| failure

case
  incoming_B == nil ⇒ fail with "unregistered sender"
  destination /= (B, connection, channel) ⇒ fail with "wrong destinati
on"
  sequence /= head(Incoming_B) ⇒ fail with "out of order"
  H_h not in T_A ⇒ fail with "must submit header for height h"
  valid(H_h, M_kvh) == false ⇒ fail with "invalid Merkle proof"
  otherwise ⇒
    set result = f_type(data)
    push(incoming_B, R{tail(incoming_B), (B, connection, channel), (A,
connection, channel), result})
    success
```

3.6 Handling a receipt

When we wish to create a transaction that atomically commits or rolls back across two chains, we must look at the execution result returned in the IBC receipt. For example, if I want to send tokens from Alice on chain `A` to Bob on chain `B`, chain `A` must decrement Alice's account *if and only if* Bob's account was incremented on chain `B`. We can achieve that by storing a protected intermediate state on chain `A` (escrowing the assets in question), which is then committed or rolled back based on the result of executing the transaction on chain `B`.

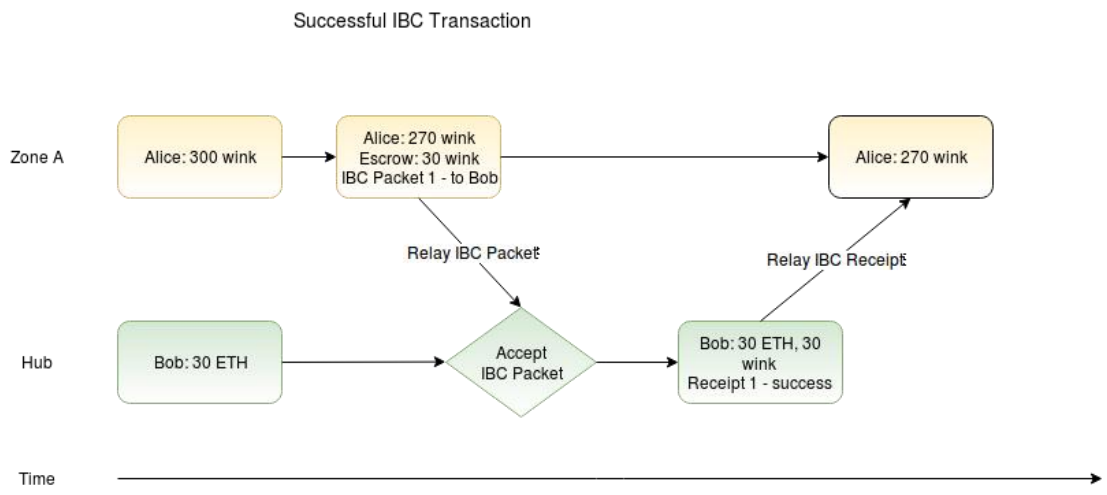
To do this requires that we not only provably send a packet from chain `A` to chain `B`, but provably return the result of executing that packet (the receipt `data`) from chain `B` to chain `A`. If a valid IBC packet was sent from `A` to `B`, then the result of executing it is stored in `incoming_B`. Since the receipts are stored in a queue with the same key construction as the sending queue, we can generate the same set of proofs for them, and perform a similar sequence of steps to handle a receipt coming back to `A` for a message previously sent to `B`. Receipts, like packets, are processed in order.

To handle an IBC receipt on blockchain `A` received from blockchain `B`, with a Merkle proof `M_kvh` and the current set of trusted headers for that chain `T_B`:

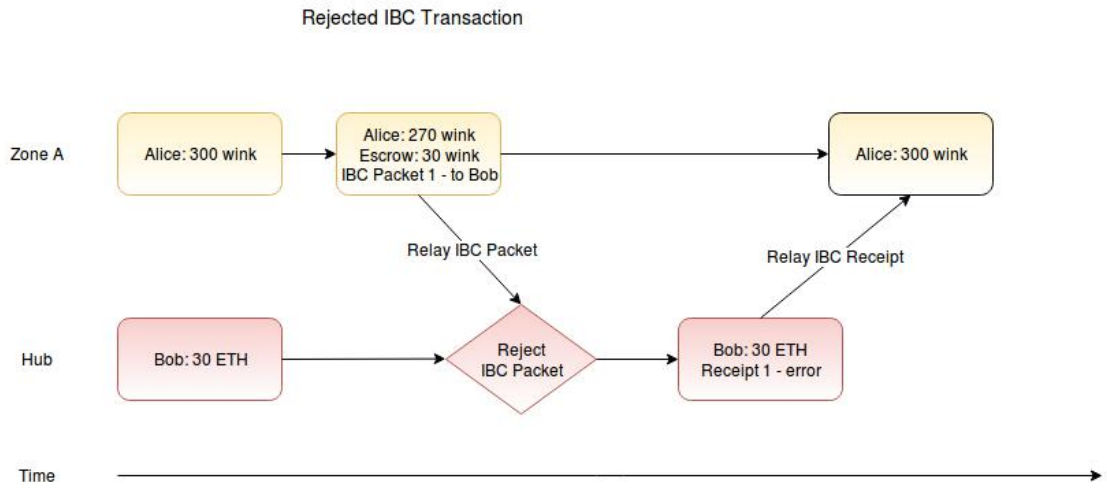
```
handle_receipt(R{sequence, source, destination, data}, M_kvh)

case
  outgoing_A == nil ⇒ fail with "unregistered sender"
  destination /= (A, connection, channel) ⇒ fail with "wrong destinati
on"
  sequence /= head(incoming_A) ⇒ fail with "out of order"
  H_h not in T_B ⇒ fail with "must submit header for height h"
  valid(H_h, M_kvh) == false ⇒ fail with "invalid Merkle proof"
  otherwise ⇒
    set P{type, _, _, _, _} = pop(outgoing_A)
    f_type(result)
    success
```

This allows applications to reason about ordering and enforce application-level guarantees by committing or reverting state changes on chain `A` based on the result of packet execution on chain `B`:



Successful Transaction



Rejected Transaction

3.7 Packet relayer

The blockchain itself only records the *intention* to send the given message to the recipient chain. Physical network packet relay must be performed by off-chain infrastructure. We define the concept of a *relay* process that connects two chains by querying one for all outgoing packets & proofs, then committing those packets & proofs to the recipient chain.

The relay process must have access to accounts on both chains with sufficient balance to pay for transaction fees but needs no other permissions. Relayers may employ application-level methods to recoup these fees. Any number of *relay* processes may be safely run in parallel. However, they will consume unnecessary fees if they submit the same proof multiple times, so some minimal coordination is ideal.

As an example, here is a naive algorithm for relaying outgoing packets from `A` to `B` and incoming receipts from `B` back to `A`. All reads of variables belonging to a chain imply queries and all function calls imply submitting a transaction to the blockchain.

```
while true
   set pending = tail(outgoing_A)
   set received = tail(incoming_B)
   if pending > received
       set U_h = A.latestHeader
       if U_h /= B.knownHeaderA
          B.updateHeader(U_h)
       for i from received to pending
           set P = outgoing_A[i]
           set M_kvh = A.prove(U_h, P)
           B.receive(P, M_kvh)
```

Note that updating a header is a costly transaction compared to posting a Merkle proof for a known header. Thus, a process could wait until many messages are pending, then submit one header along with multiple Merkle proofs, rather than a separate header for each message. This decreases total computation cost (and fees) at the price of additional latency and is a trade-off each relay can dynamically adjust.

4 Optimizations

(Back to table of contents)

The above sections describe a secure messaging protocol that can handle all normal situations between two blockchains. All messages are processed exactly once and in order, and applications can guarantee invariants over their combined state on both chains. IBC can be further extended and optimized to provide additional guarantees and minimize costs on the underlying blockchains. We detail two extensions: packet timeouts and packet cleanup.

4.1 Timeouts

Application semantics may require some timeout: an upper limit to how long the chain will wait for a transaction to be processed before considering it an error. Since the two chains have different local clocks, this is an obvious attack vector for a double spend - an attacker may delay the relay of the receipt or wait to send the packet until right after the timeout - so applications cannot safely implement naive timeout logic themselves.

One solution is to include a timeout in the IBC packet itself. When sending a packet, one can specify a block height or timestamp on chain `B` after which the packet is no longer valid. If the packet is posted before the cutoff, it will be processed normally. If it is posted after the cutoff, it will be a guaranteed error. In order to provide the necessary guarantees, the timeout must be specified relative to a condition on the receiving chain, and the sending chain must have proof of this condition after the cutoff.

For a sending chain `A` and a receiving chain `B`, with an IBC packet `P={_, i, _, _, _}` and some height `h` on chain `B`, the base IBC protocol provides the following guarantees:

`A:M_kvh == ∅` if message `i` was not sent before height `h`

`A:M_kvh == ∅` if message `i` was sent and the corresponding receipt received before height `h` (and the receipts for all messages j < i were also handled)

`A:M_kvh /= ∅` otherwise, if message `i` was sent but the receipt has not yet been processed

`B:M_kvh == ∅` if message `i` was not received before height `h`

`B:M_kvh /= ∅` if message `i` was received before height `h`

We can make a few modifications of the above protocol to allow us to prove timeouts, by adding some fields to the messages in the send queue and defining an expired function that returns true iff `h > maxHeight` or `timestamp(H_h) > maxTime`.

```
P = (type, sequence, source, destination, data, maxHeight, maxTime)
```

```
expired(H_h, P) ⇒ true | false
```

We then update message handling in `receive`, so that chain `B` doesn't even call the handler function if the timeout was reached but instead directly writes an error in the receipt queue:

```
receive
```

```
case
  ...
  expired(latestHeader, v) ⇒ push(incoming_b, R{..., TimeoutError})
  otherwise ⇒
    set result = f_type(data)
    push(incoming_B, R{tail(incoming_B), (B, connection, channel), (A,
connection, channel), result})
```

The `receipt_handler` function on chain `A` can now verify timeouts and pass valid timeout receipts to the application handler (which can revert state changes such as escrowing assets):

```
receipt_handler
```

```
case
  ...
  result == TimeoutError ⇒ case
    not expired(H_h, P) ⇒ fail with "message timeout not yet reached"
    otherwise ⇒ f_type(R, TimeoutError)
  ...
```

This adds one more guarantee:

`A:M_kvh == ∅` if message i was sent and timeout proven before height h (and the receipts for all messages j < i were also handled).

Now chain `A` can rollback all transactions that were blocked by this flood of unrelayed packets - since they can never confirm - without waiting for chain `B` to process them and return a receipt. Adding reasonable timeouts to all packets allows us to gracefully handle any errors with the IBC relay processes or a flood of unrelayed "spam" IBC packets. If a blockchain requires a timeout on all messages and imposes some reasonable upper limit, we can guarantee that if a packet is not processed by the upper limit of the timeout period, then all previous packets must also have either been processed or reached the timeout period.

Note that in order to avoid any possible "double-spend" attacks, the timeout algorithm 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.

Additionally, if timestamp-based timeouts are used instead of height-based timeouts, the destination chain's consensus ruleset must enforce always-increasing timestamps (or the sending chain must use a more complex `expired` function).

4.2 Cleanup

While we clean up the *send queue* upon getting a receipt, if left to run indefinitely, the *receipt queues* could grow without limit and create a major storage cost for the chains. However, we must not delete receipts until they have been proven to be processed by the sending chain, or we lose important information and sacrifice reliability.

Additionally, with the above timeout implementation, when we perform the timeout on the sending chain, we do not update the *receipt queue* on the receiving chain, and now it is blocked waiting for a packet `i`, which no longer exists on the sending chain. We can update the guarantees of the receipt queue as follows to allow us to handle both:

`B:M_kvh == ∅` if packet `i` was not received before height `h`

`B:M_kvh == ∅` if packet i was provably resolved on the sending chain before height `h`

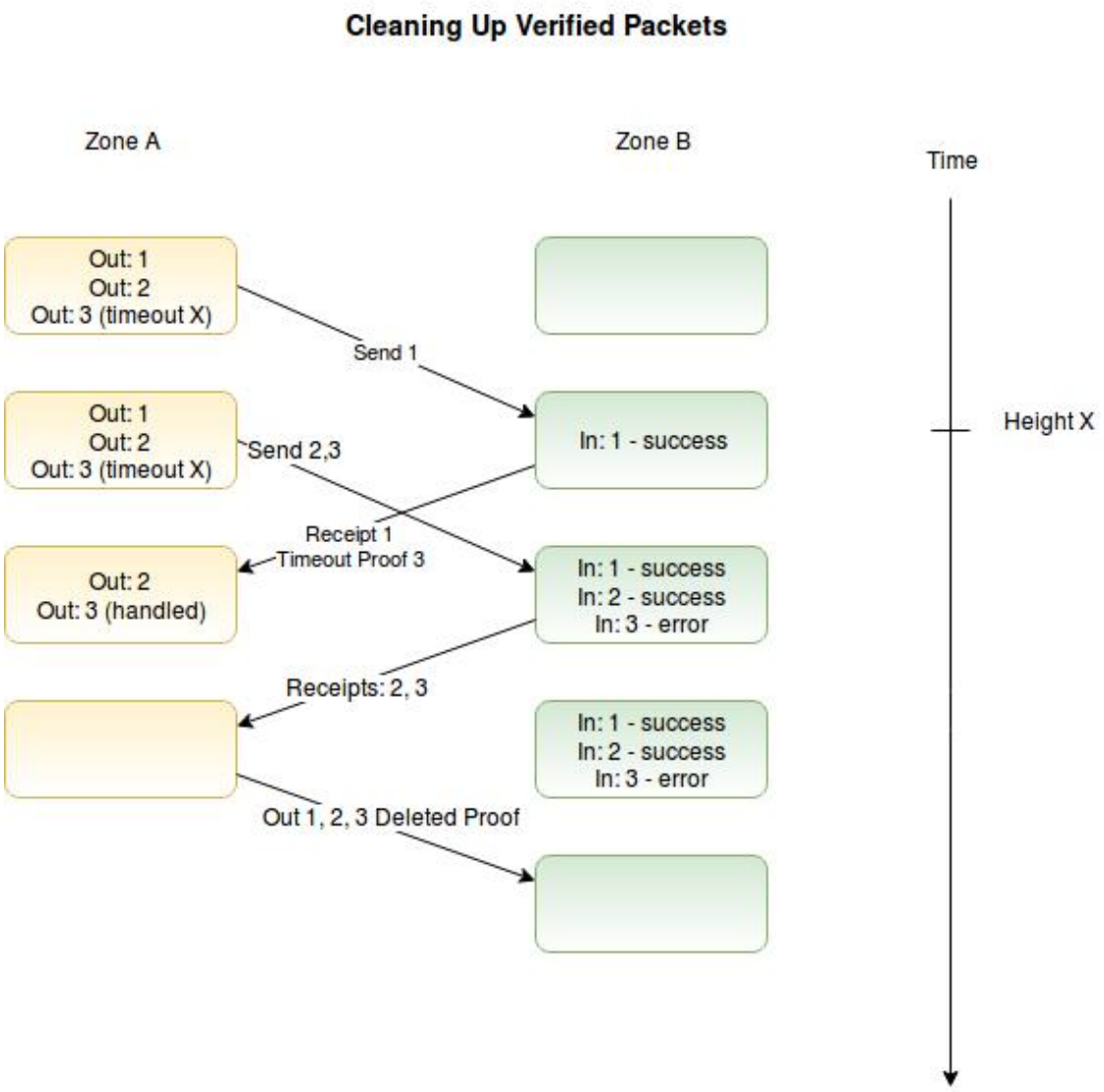
`B:M_kvh /= ∅` otherwise (if packet `i` was processed before height `h` but chain `A` has not handled the receipt)

Consider a connection where many messages have been sent, and their receipts processed on the sending chain, either explicitly or through a timeout. We wish to quickly advance over all the processed messages, either for a normal cleanup, or to prepare the queue for normal use again after timeouts.

Through the definition of the send queue, we know that all packets `i < head` have been fully processed and all packets `head <= i < tail` are awaiting processing. By proving a much advanced `head` of `outgoing_B`, we can demonstrate that the sending chain already handled all messages. Thus, we can safely advance `incoming_A` to the new head of `outgoing_B`.

```
cleanup(A, M_kvh, head) = case
  incoming_A == ∅  => fail with "unknown sender"
  H_h ∉  T_B => fail with "must submit header for height h"
  not valid(H_h, M_kvh, head) => fail with "invalid Merkle proof of out
going_B queue height"
  head >= head(incoming_A) => fail with "cleanup must go forward"
  otherwise =>
    advance(incoming_A, head)
```

This allows us to invoke the `cleanup` function to resolve all outstanding messages up to and including `index` with one Merkle proof. Note that if this handles both recovering from a blocked queue after timeouts, as well as a routine cleanup method to recover space. In the cleanup scenario, we assume that there may also be a number of packets that have been processed by the receiving chain, but not yet posted to the sending chain, `tail(incoming_B) > head(outgoing_A)`. As such, `advance` must not modify any packets between the head and the tail.



Cleaning up Packets

5 Conclusion

(Back to table of contents)

We have demonstrated a secure, performant, and flexible protocol for cross-blockchain messaging, and provided sufficient detail to reason about the correctness and efficiency of the protocol.

This document defines solely a message queue protocol - not the application-level semantics which must sit on top of it to enable asset transfer between two chains. We will shortly release a separate paper on Cosmos IBC that defines the application logic used for direct value transfer as well as routing over the Cosmos hub. That paper builds upon the IBC protocol defined here and provides a first example of how to reason about application logic and global invariants in the context of IBC.

There is a reference implementation of the Cosmos IBC protocol as part of the Cosmos SDK, written in Golang and released under the Apache license. To facilitate implementations in other langauages which are wire-compatible with the Cosmos implementation, the following appendices define exact message and binary encoding formats.

References

(Back to table of contents)

1:

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

2:

https://en.wikipedia.org/wiki/Vector_clock

3:

https://blog.cosmos.network/consensus-compare-casper-vs-tendermint-6df154ad56ae#215d

4:

https://blog.cosmos.network/light-clients-in-tendermint-consensus-1237cfbda104

5:

http://scattered-thoughts.net/blog/2012/08/16/causal-ordering/

6:

https://github.com/ethereum/wiki/wiki/RLP

7:

https://developers.google.com/protocol-buffers/

8:

https://github.com/tendermint/go-wire

9:

https://developers.google.com/protocol-buffers/docs/proto3

10:

https://en.wikipedia.org/wiki/Merkle_tree

11:

https://chainpoint.org/

12:

https://github.com/tendermint/iavl

Appendices

(Back to table of contents)

Appendix A: Encoding Libraries

The specification has focused on semantics and functionality of the IBC protocol. However in order to facilitate the communication between multiple implementations of the protocol, we seek to define a standard syntax, or binary encoding, of the data structures defined above. Many structures are universal and for these, we provide one standard syntax. Other structures, such as _Hh , Uh , _and *Xh* are tied to the consensus engine and we can define the standard encoding for tendermint, but support for additional consensus engines must be added separately. Finally, there are some aspects of the messaging, such as the envelope to post this data (fees, nonce, signatures, etc.), which is different for every chain, and must be known to the relay, but are not important to the IBC algorithm itself and left undefined.

In defining a standard binary encoding for all the "universal" components, we wish to make use of a standardized library, with efficient serialization and support in multiple languages. We considered two main formats: Ethereum's RLP[6] and Google's Protobuf[7]. We decided for protobuf, as it is more widely supported, is more expressive for different data types, and supports code generation for very efficient (de)serialization codecs. It does have a learning curve and more setup to generate the code from the type specifications, but the ibc data types should not change often and this code generation setup only needs to happen once per language (and can be exposed in a common repo), so this is not a strong counter-argument. Efficiency, expressiveness, and wider support rule in its favor. It is also widely used in gRPC and in many microservice architectures.

The tendermint-specific data structures are encoded with go-wire[8], the native binary encoding used inside of tendermint. Most blockchains define their own formats, and until some universal format for headers and signatures among blockchains emerge, it seems very premature to enforce any encoding here. These are defined as arbitrary byte slices in the protocol, to be parsed in an consensus engine-dependent manner.

For the following appendixes, the data structure specifications will be in proto3[9] format.

Appendix B: IBC Queue Format

The foundational data structure of the IBC protocol are the packet queues stored inside each chain. We start with a well-defined binary representation of the keys and values used in these queues. The encodings mirror the semantics defined above:

key = (*remote id, [send|receipt], [head|tail|index])*

Vsend = (maxHeight, maxTime, type, data)

Vreceipt = (result, [success|error code])

Keys and values are binary encoded and stored as bytes in the Merkle tree in order to generate the root hash stored in the block header, which validates all proofs. They are treated as arrays of bytes by the Merkle proofs for deterministically generating the sequence of hashes and passed as such in all interchain messages. Once the validity of a key value pair has been determined from the Merkle proof and header, the payload bytes can be deserialized and interpreted by the protocol.

See binary format as protobuf specification

Appendix C: Merkle Proof Formats

A Merkle tree (or a trie) generates a single hash that can be used to prove any element of the tree. In order to generate this hash, we first hash the leaf nodes, then hash multiple leaf nodes together to get the hash of an inner node (two or more, based on degree k of the k-ary tree), and repeat for each level of the tree until we end up with one root hash. With a known root hash (which is included in the block headers), the existence of a particular key/value in the tree can be proven by tracing the path to the value and revealing the (k-1) hashes for the paths not taken on each level ([10]).

There are a number of different implementations of this basic idea, using different hash functions, as well as prefixes to prevent second preimage attacks (differentiating leaf nodes from inner nodes). Rather than force all chains that wish to participate in IBC to use the same data store, we provide a data structure that can represent Merkle proofs from a variety of data stores, and provide for chaining proofs to allow for subtrees. While searching for a solution, we did find the chainpoint proof format[11], which inspired this design significantly, but didn't (yet) offer the flexibility we needed.

We generalize the left/right idiom to the concatenation a (possibly empty) fixed prefix, the (just calculated) last hash, and a (possibly empty) fixed suffix. We must only define two fields on each level and can represent any type, even a 16-ary Patricia tree, with this structure. One must only translate from the store's native proof to this format, and it can be verified by any chain, providing compatibility with arbitrary data stores.

The proof format also allows for chaining of trees, combining multiple Merkle stores into a "multi-store". Many applications (such as the EVM) define a data store with a large proof size for internal use. Rather than force them to change the store (impossible), or live with huge proofs (inefficient), we provide the possibility to express Merkle proofs connecting multiple subtrees. Thus, one could have one subtree for data, and a second for IBC. Each tree produces its own Merkle root, and these are then hashed together to produce the root hash that is stored in the block header.

A valid Merkle proof for IBC must either consist of a proof of one tree, and prepend `ibc` to all key names as defined above, or use a subtree named `ibc` in the first section, and store the key names as above in the second tree.

In order to minimize the size of their Merkle proofs, we recommend using Tendermint's IAVL+ tree implementation[12], which is designed for optimal proof size and released under a permissive license. It uses an AVL tree (a type of binary tree) with ripemd160 as the hashing algorithm at each stage. This produces optimally compact proofs, ideal for posting in blockchain transactions. For a data store of *n* values, there will be *log2(n)* levels, each requiring one 20-byte hash for proving the branch not taken (plus possible metadata for the level). We can express a proof in a tree of 1 million elements in something around 400 bytes. If we further store all IBC messages in a separate subtree, we should expect the count of nodes in this tree to be a few thousand, and require less than 400 bytes, even for blockchains with a large state.

See binary format as protobuf specification

Appendix D: Byzantine Recovery Strategies

IBC guarantees reliable, ordered packet delivery in the face of malicious nodes or relays, on top of which application invariants can be ensured. However, all guarantees break down when the blockchain on the other end of the connection exhibits Byzantine behavior. This can take two forms: a failure of the consensus mechanism (reverting previously finalized blocks), or a failure at the application level (not correctly performing the application-level functions on the packet).

The IBC protocol can detect a limited class of Byzantine faults at the consensus level by identifying duplicate headers -- if an IBC module ever sees two different headers for the same height (or any evidence that headers belong to different forks), then it can freeze the connection immediately. State reconciliation (e.g. restoring token balances to owners of vouchers on the other chain) must be handled by blockchain governance.

If there is a big divide in the remote chain and the validation set splits (e.g. 60-40 weighted) as to the direction of the chain, then the light-client header update 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 the connection(s) will

need to be reopened manually (by governance on the local chain) and set to the new header set(s). The IBC protocol doesn't have the option to follow both chains as the queue and associated state must map to exactly one remote chain. In a fork, the chain can continue the connection with one fork, and optionally make a fresh connection with the other fork.

Another kind of Byzantine action is at the application level. Let us assume packets represent transfer of value. If chain `A` sends a message with `x` tokens to chain `B`, then it promises to remove `x` tokens from the local supply. And if chain `B` handles this message successfully, it promises to credit `x` token vouchers to the account indicated in the packet. If chain `A` does not remove tokens from supply, or chain `B` does not generate vouchers, the application invariants (conservation of supply & fungibility) break down.

The IBC protocol does not handle these kinds of errors. They must be handled individually by each application. Applications could use Plasma-like fraud proofs to allow state recovery on one chain if fraud can be proved on the other chain. Although complex to implement, a correct implementation would allow applications to guarantee their invariants as long as *either* blockchain's consensus algorithm behaves correctly (and this could be extended to `n` chains). Economic incentives can additionally be used to disincentivize any kind of provable fraud.

Appendix E: Tendermint Header Proofs

{ Ensure this is correct. }

TODO: clean this all up

This is a mess now, we need to figure out what formats we use, define go-wire, etc. or just point to the source???? Will do more later, need help here from the tendermint core team.

In order to prove a merkle root, we must fully define the headers, signatures, and validator information returned from the Tendermint consensus engine, as well as the rules by which to verify a header. We also define here the messages used for creating and removing connections to other blockchains as well as how to handle forks.

Building Blocks: Header, PubKey, Signature, Commit, ValidatorSet

→ needs input/support from Tendermint Core team (and go-crypto)

Registering Chain

Updating Header

Validator Changes

ROOT of trust

As mentioned in the definitions, all proofs are based on an original assumption. The root of trust here is either the genesis block (if it is newer than the unbonding period) or any signed header of the other chain.

When governance on a pair of chain, the respective chains must agree to a root of trust on the counterparty chain. This can be the genesis block on a chain that launches with an IBC channel or a later block header.

From this signed header, one can check the validator set against the validator hash stored in the header, and then verify the signatures match. This provides internal consistency and accountability, but if 5 nodes provide you different headers (eg. of forks), you must make a subjective decision which one to trust. This should be performed by on-chain governance to avoid an exploitable position of trust.

VERIFYING HEADERS

Once we have a trusted header with a known validator set, we can quickly validate any new header with the same validator set. To validate a new header, simply verifying that the validator hash has not changed, and that over 2/3 of the voting power in that set has properly signed a commit for that header. We can skip all intervening headers, as we have complete finality (no forks) and accountability (to punish a double-sign).

This is safe as long as 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 demonstrates the importance of attribution and is the same security guarantee of any non-validating full node. Even in the presence of some ultra-powerful malicious actors, this makes the cost of creating a fake proof for a header equal to at least one third of all staked tokens, which should be significantly higher than any gain of a false message.

UPDATING VALIDATORS SET

If the validator hash is different than the trusted one, we must simultaneously both verify that if the change is valid while, as well as use using 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).