IBC Protocol Specification

v0.3.1

Ethan Frey
frey@tendermint.com

Nov. 30, 2017

Abstract

This paper specifies the IBC (inter blockchain communication) protocol, which was first described in the Cosmos white paper[1] in June 2016. The IBC protocol uses authenticated message passing to simultaneously solve two problems: transferring value (and state) between two distinct chains, as well as sharding one chain securely. IBC follows the message-passing paradigm and assumes the participating chains are independent.

Each chain maintains a local partial order, while inter-chain messages track any cross-chain causality relations. Once two chains have registered a trust relationship, cryptographically provable packets can be securely sent between the chains, using Tendermint's instant finality for quick and efficient transmission.

We currently use this protocol for secure value transfer in the Cosmos Hub, but the protocol can support arbitrary application logic. Details of how Cosmos Hub uses IBC to securely route and transfer value are provided in a separate paper, along with a framework for expressing global invariants. 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 requires cryptographic proofs for every message, and thus is highly robust in a heterogeneous environment with Byzantine actors. 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/master/WHITEPAPER.md#inter-blockchain-communication-ibc

Contents

1 Overview

The IBC protocol creates a mechanism by which multiple sovereign replicated fault tolerant state machines my 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 extremely secure message queue protocol, allowing the creation of complex inter-chain processes without trusted parties. This architecture can be seen as a parallel to microservices in the blockchain space, and the IBC protocol can be seen as an analog to the AMQP messaging protocol[2], used by StormMQ, RabbitMQ, etc.

The message packets are not signed by one psuedonymous account, or even multiple. Rather, IBC effectively assigns authorization of the packets to the blockchain's consensus algorithm itself. Not only are blockchains highly secure, they are auditable and have an extremely high creation cost in comparison to cryptographic key pairs. This prevents Sybil attacks and allows out-of-protocol accountability, since any byzantine behavior is provable and can be published to damage the reputation/value of the other blockchain. By using registered blockchains as "actors" in the system, we can achieve extremely high security through a combination of cryptography and incentives.

In this paper, we define a process of posting block headers and merkle proofs to enable secure verification of individual packets. We then describe how to combine these packets into a messaging queue to guarantee reliable, in-order delivery of message. We then explain how to securely handle receipts (response/error), which enables the creation of asynchronous RPC-like protocols. Finally, we detail some optimizations and how to handle byzantine blockchains.

1.1 Definitions

Blockchain - an immutable ledger created through distributed consensus, coupled with a deterministic state machine to process the transactions on the ledger. The smallest unit produced through consensus is a block, which may contain many transactions.

2 http://www.amqp.org/sites/amqp.org/files/amqp.pdf

Module - we assume that the state machine of the blockchain is comprised of multiple components (modules or smart contracts) that have limited rights, and they can only interact over pre-defined interfaces rather than directly mutating internal state.

Finality - a guarantee that a given block will not be reverted within some predefined conditions. All proof of work systems offer probabilistic finality, which means the probability of that a block will be reverted approaches 0. A "better", alternative chain could exist, but the cost of creation increases rapidly over time. Many "proof of stake" systems offer much weaker guarantees, based only on the honesty of the miners. However, BFT algorithms such as Tendermint guarantee complete finality upon production of a block, unless over two thirds of the validators collude to break consensus. This collusion is provable and can be punished.

Knowledge - what is certain to be true.

Provable - the existence of irrefutable mathematical (often cryptographic) proof of the truth of a given statement. These can be expressed as: given knowledge **A** and a statement **s**, then **B** must be true. This is a form of deductive proof and they can be chained together without losing validity.

Attributable - provable knowledge of who made a statement. If a statement is provably false, then it is known which actor lied. Attributable statements allow us to build incentives against lying, which help enforce finality. This is also referred to as accountability.

Root of Trust - any proof depends on some prior assumptions, however simple they are. We refer to the first assumption we make as the root of trust, and all our knowledge of the system is derived from this root through a provable chain of information. We seek to make this root of trust as simple and a verifiable as possible, since if the original assignment of trust is false, all conclusions drawn will also be false.

Unbonding Period - Proof of Stake algorithms need to freeze the stake for some time to provide a lower bound for the length of a long-range attack[3]. Since complete finality is associated with a subset of the Proof of Stake class of consensus algorithms, I will assume all implementations that support IBC have some unbonding period P, such that if my last knowledge of the blockchain is older than P, I can no longer trust any message without a new

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

root of trust.

The IBC protocol requires each actor to be a blockchain with complete finality. All transitions must be provable and attributable to (at least) one actor. That implies the smallest unit of trust is the consensus algorithm of a blockchain.

1.2 Threat Models

False statements - any information we receive may be false, all actors must have enough knowledge be able to prove its correctness without external dependencies. All statements should be attributable.

Network partitions and delays - we assume an asynchronous, adversarial network. Any message may or may not reach the destination. They may be modified or selectively dropped. Messages may reach the destination out of order and may arrive multiple times. There is no upper limit to the time it takes for a message to be received. Actors may be arbitrarily partitioned by a powerful adversary. The protocol favors correctness over liveness. That is, it only acts upon information that is provably correct.

Byzantine actors - it is possible that an entire blockchain is not acting according to protocol. This must be detectable and provable, allowing the communicating blockchain to revoke trust and take necessary action. Furthermore, we should design application-level protocols on top of IBC to minimize risk exposure in the face of Byzantine actors.

2 Proofs

The basis of IBC is the ability to perform efficient proofs of a message packet on-chain and deterministically. All transactions must be attributable and provable without depending on any information outside of the blockchain. We define the following variables: H_h is the signed header at height h, C_h are the consensus rules at height h, and P is the unbonding period of this blockchain. $V_{k,h}$ is the value stored under key k at height h. Note that of all these, only H_h defines a signature and is thus attributable.

To support an IBC connection, two actors must be able to make the following proofs to each other:

- 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 $\Delta(now, H_h) < P$

- given a trusted H_h and C_h and an attributable change message $X_{h'}$ it is possible to prove $H_{h'}$ where $C_{h'} \neq C_h$ and $\Delta(now, H_h) < P$
- given a trusted H_h and a merkle proof $M_{k,v,h}$ it is possible to prove $V_{k,h}$

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 is defined in Appendix E. Another engine able to provide equally strong guarantees (such as Casper) should be theoretically compatible with IBC, and must define its own set of update/change messages.

The merkle proof $M_{k,v,h}$ 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.

$valid(H_h, M_{k,v,h}) \Rightarrow [true \mid false]$

2.1 Establishing a Root of Trust

As mentioned in the definitions, all proofs are based on an original assumption. In this case it is H_h and C_h for some h, where $\Delta(now, H_h) < P$.

Any header may be from a malicious chain (eg. shadowing a real chain id with a fake validator set), so a subjective decision is required before establishing a connection. This should be performed by on-chain governance to avoid an exploitable position of trust. Establishing a bidirectional root of trust between two blockchains (A trusts B and B trusts A) is a necessary and sufficient prerequisite for all other IBC activity.

Development of a fully open and decentralized PKI for tracking blockchains is an open research question for future iterations of the IBC protocol.

2.2 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 a future H_h where $h > n$. Some implementations may provide the additional limitation that $h = n + 1$, which requires us to process every header. Tendermint allows us to exploit

knowledge of the BFT algorithm to only require the additional limitation $\Delta_{vals}(C_n, C_h) < \frac{1}{3}$, that each step must have a change of less than one-third of the validator set[4].

Any of these requirements allows us to support IBC for the given block chain. However, by supporting proofs where $h\text{-}n > 1$, we can follow the block headers much more efficiently in situations where the majority of blocks do not include an IBC message between chains A and B, and enable low-bandwidth connections to be implemented at very low cost. If there are messages to relay every block, then these collapse to the same case, relaying every header.

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 or provide fake proof can be submitted to the remote blockchain for punishment, in the same manner that any violation of the internal consensus algorithm is punished. This incentive enhances the security guarantees and avoids the nothing-at-stake issue in IBC as well.

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

$valid(T, X_h \mid U_h) \Rightarrow [true \mid false \mid unknown]$
if $H_{h-1} \in T$ *then*
    $valid(T, X_h \mid U_h) \Rightarrow [true \mid false]$
    there must exist some U_h *or* X_h *that evaluates to true*
if $C_h \notin T$ *then*
    $valid(T, U_h) \Rightarrow false$

and can process update transactions as follows:

$update(T, X_h \mid U_h) \Rightarrow$ *match* $valid(T, X_h \mid U_h)$
    $false \Rightarrow$ *return Error("invalid proof")*
    unknown $\Rightarrow$ *return Error("need a proof between current and h")*
    $true \Rightarrow T \cup (H_h, C_h)$

We define $max(T)$ as $max(h, where\ H_h \in T)$ for any T with $max(T) = h\text{-}1$. And from above, there must exist some $X_h \mid U_h$ so that $max(update(T, X_h \mid U_h)) = h$. By induction, we can see there must exist a set of proofs, such that $max(update...(T,...)) = h+n$ for any n.

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

We also can see the validity of using bisection as an optimization to discover this set of proofs. That is, given $max(T) = n$ and $valid(T, X_h \mid U_h) = unknown$, we then try $update(T, X_b \mid U_b)$, where $b = (h+n)/2$. The base case is where $valid(T, X_h \mid U_h) = true$ and is guaranteed to exist if $h=max(T)+1$.

3 Messaging Queue

Messaging in distributed systems is a deeply researched field and a primitive upon which many other systems are built. We can model asynchronous message passing, 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 decides how to act upon the incoming message. Just because one zone sends a message and we have an IBC connection with this zone, doesn't mean we have to execute the requested action. Each zone can add its own business logic upon receiving the message to decide whether to accept or reject the message. To maintain consistency, both sides must only agree on the proper state transitions associated with accepting or rejecting.

This encapsulation 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 microservices architecture, but across organizational boundaries.

To build useful algorithms upon a provable asynchronous messaging primitive, we introduce a reliable messaging queue (hereafter just referred to as a queue), typical in asynchronous message passing, to allow us to guarantee a causal ordering[5], and avoid blocking.

Causal ordering means that if x is causally before y on chain A, it must also be on chain B. Many events may happen concurrently (unrelated tx on two different blockchains) with no causality relation, but every transaction on the same chain has a clear causality relation (same as the order in the blockchain).

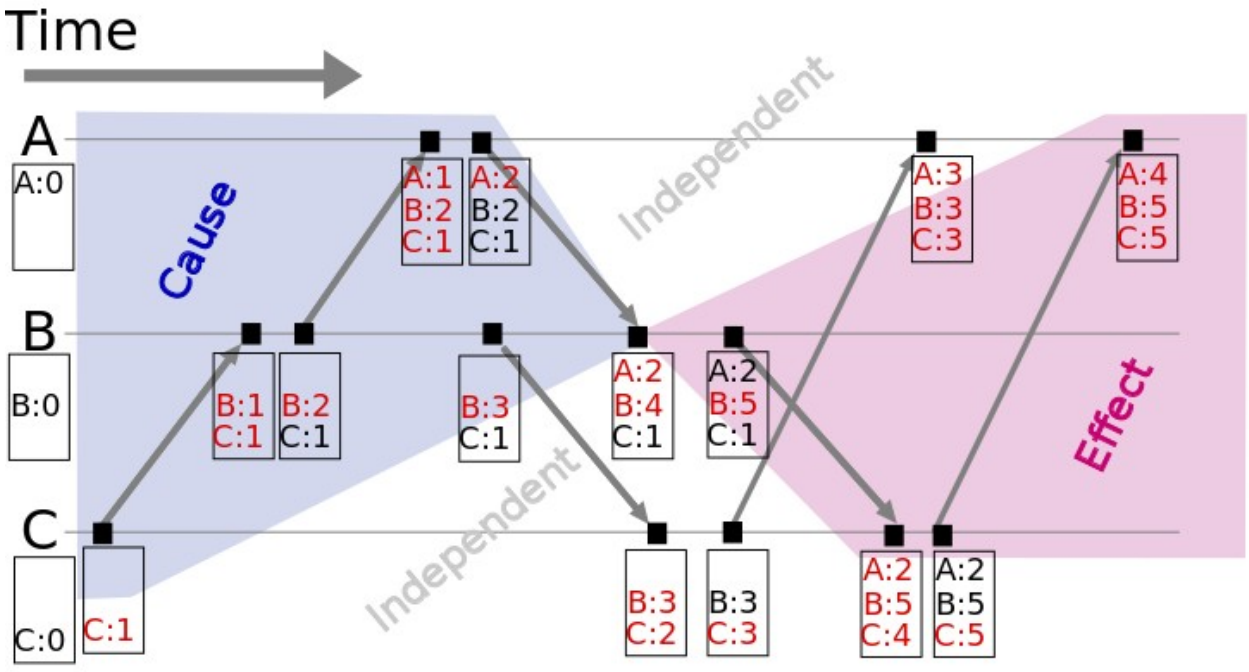
Message passing implies a causal ordering over multiple chains and these can be important for reasoning on the system. Given $x \rightarrow y$ means x is

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

causally before y, and chains A and B, and $a \Rightarrow b$ means a implies b:

$A{:}send(msg_i) \rightarrow B{:}receive(msg_i)$
$B{:}receive(msg_i) \rightarrow A{:}receipt(msg_i)$
$A{:}send(msg_i) \rightarrow A{:}send(msg_{i+1})$

$x \rightarrow A{:}send(msg_i) \Rightarrow x \rightarrow B{:}receive(msg_i)$
$y \rightarrow B{:}receive(msg_i) \Rightarrow y \rightarrow A{:}receipt(msg_i)$



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

In this section, we define an efficient implementation of a secure, reliable messaging queue.

3.1 Merkle Proofs for Queues

Given the three proofs we have available, we make use of the most flexible one, $M_{k,v,h}$, to provide proofs for a message queue. To do so, we must define a unique, deterministic, and predictable key in the merkle store for each message in the queue. We also define a clearly defined format for the content of each message in the queue, which can be parsed by all chains participating in IBC. The key format and queue ordering are conceptually explained here. The binary encoding format can be found in Appendix C.

We can visualize a queue as a slice pointing into an infinite sized array. It maintains a head and a tail pointing to two indexes, such that there is data for every index where *head <= index < tail*. Data is pushed to the tail and popped from the head. Another method, *advance*, is introduced to pop all messages until *i*, and is useful for cleanup:

init: $q_{head} = q_{tail} = 0$
peek ⇒m: *if* $q_{head} = q_{tail}$ *{ return None } else { return* $q[q_{head}]$ *}*
pop ⇒m: *if* $q_{head} = q_{tail}$ *{ return None } else {* q_{head}*++; return* $q[q_{head}-1]$ *}*
push(m): $q[q_{tail}] = m$; q_{tail}*++*
advance(i): $q_{head} = i$; $q_{tail} = max(q_{tail}, i)$
head ⇒i: q_{head}
tail⇒i: q_{tail}

Based upon this needed functionality, we define a set of keys to be stored in the merkle tree, which allows us to efficiently implement and prove any of the above queries.

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 serialization cannot collide with any possible index.

A message queue is simply a set of serialized packets stored at predefined keys in a merkle store, which can produce proofs for any key. Once a packet is written it must be immutable (except for deleting when popped from the queue). That is, if a value v is written to a queue, then every valid proof $M_{k,v,h}$ must refer to the same v. This property is essential to safely process asynchronous messages.

Every IBC implementation must provide a protected subspace of the merkle store for use by each queue that cannot be affected by other modules.

3.2 Naming Queues

As mentioned above, in order for the receiver to unambiguously interpret the merkle proofs, we need a unique, deterministic, and predictable key in the merkle store for each message in the queue. We explained how the indexes are generated to provide each message in a queue a unique key, and mentioned the need for a unique name for each queue.

The queue name must be unambiguously associated with a given connection to another chain, so an observer can prove if a message was intended for chain A or chain B. In order to do so, upon registration of a connection with a remote chain, we create two queues with different names (prefixes).

- *ibc:<chain id of A>:send* - all outgoing packets destined to chain A
- *ibc:<chain id of A>:receipt* - the results of executing the packets received from chain A

These two queues have different purposes and store messages of different types. By parsing the key of a merkle proof, a recipient can uniquely identify which queue, if any, this message belongs to. We now define k = (*remote id, [send|receipt], index)*. This tuple is used to route and verify every message, before the contents of the packet are processed by the appropriate application logic.

3.3 Message Contents

Up to this point, we have focused on the semantics of the message key, and how we can produce a unique identifier for every possible message in every possible connection. The actual data written at the location has been left as an opaque blob, put by providing some structure to the messages, we can enable more functionality.

We define every message in a *send queue* to consist of a well-known type and opaque data. The IBC protocol relies on the type for routing, and lets the appropriate module process the data as it sees fit. The *receipt queue* stores if it was an error, an optional error code, and an optional return value. We use the same index as the received message, so that the results of $A{:}q_{B.send}[i]$ are stored at $B{:}q_{A.receipt}[i]$. (read: the message at index i in the *send* queue for chain B as stored on chain A)

V_{send} = *(type, data)*
$V_{receipt}$ = *(result, [success|error code])*

3.4 Sending a Message

A proper implementation of IBC requires all relevant state to be encapsulated, so that other modules can only interact with it via a fixed API (to be defined in the next sections) rather than directly mutating internal state. This allows the IBC module to provide security guarantees.

Sending an IBC packet involves an application module calling the send method of the IBC module with a packet and a destination chain id. 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 so, the IBC module simply pushes the packet to the tail of the *send queue*, which enables all the proofs described above.

The permissioning of which module can write which packet can be defined per type, so this module can maintain any application-level invariants related to this area. Thus, the "coin" module can maintain the constant supply of tokens, while another module can maintain its own invariants, without IBC messages providing a means to escape their encapsulations. The IBC module must associate every supported message type with a particular handler (f_{type}) and return an error for unsupported types.

$$(IBCsend(D, type, data) \Rightarrow Success) \Rightarrow push(q_{D.send}, V_{send}\{type, data\})$$

We also consider how a given blockchain A is expected to receive the packet from a source chain S with a merkle proof, given the current set of trusted headers for that chain, T_S:

$$A{:}IBCreceive(S, M_{k,v,h}) \Rightarrow match$$
$$q_{S.receipt} = \emptyset \Rightarrow Error(\text{"unregistered sender"}),$$
$$k = (\_, reciept, \_) \Rightarrow Error(\text{"must be a send"}),$$
$$k = (d, \_, \_) \text{ and } d \neq A \Rightarrow Error(\text{"sent to a different chain"}),$$
$$k = (\_, send, i) \text{ and } head(q_{S.receipt}) \neq i \Rightarrow Error(\text{"out of order"}),$$
$$H_h \notin T_S \Rightarrow Error(\text{"must submit header for height h"}),$$
$$valid(H_h, M_{k,v,h}) = false \Rightarrow Error(\text{"invalid merkle proof"}),$$
$$v = (type, data) \Rightarrow (result, err) := f_{type}(data); push(q_{S.receipt}, (result, err)); Success$$

Note that this requires not only an valid proof, but also that the proper header as well as all prior messages were previously submitted. This returns success upon accepting a proper message, even if the message execution returned an error (which must then be relayed to the sender).

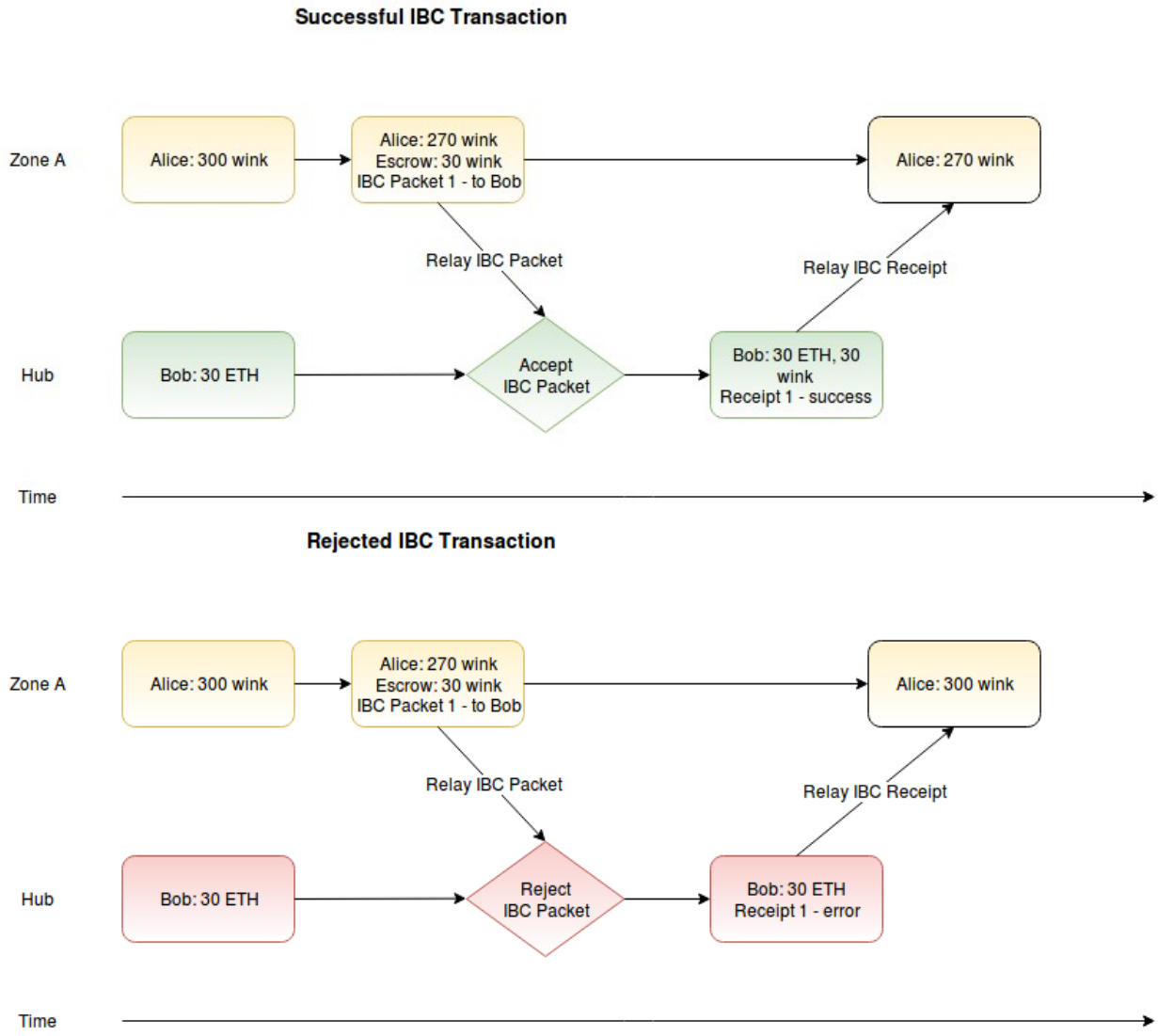
3.5 Receipts

When we wish to create a transaction that atomically commits or rolls back across two chains, we must look at the receipts from sending the original message. 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, 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 provable send a message from chain A to chain B, but probably~~provably~~ return the result of that message (the receipt) from chain B to chain A. As one noticed above in the implementation of *IBCreceive*, if the valid IBC message was sent from A to B, then the result of executing it, even if it was an error, is stored in $B{:}q_{A.receipt}$. 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 *S* for a message previously sent to *A*:

S:IBCreceipt(A, $M_{k,v,h}$) ⇒ match
$q_{A.send}$ =∅⇒Error("unregistered sender"),
k = (_, send, _) ⇒Error("must be a recipient"),
k = (d, _, _) and d ≠S⇒Error("sent to a different chain"),
$H_h \notin T_A$ ⇒Error("must submit header for height h"),
not valid(H_h ,$M_{k,v,h}$) ⇒Error("invalid merkle proof"),
k = (_, receipt, head|tail)⇒Error("only accepts message proofs"),
k = (_, receipt, i) and head($q_{S.send}$) ≠i⇒Error("out of order"),
v = (_, error) ⇒(type, data) := pop($q_{S.send}$); $rollback_{type}$(data); Success
v = (res, success) ⇒(type, data) := pop($q_{S.send}$); $commit_{type}$(data, res); Success

This enforces that the receipts are processed in order, to allow some the application to make use of some basic assumptions about ordering. It also removes the message from the send queue, as there is now proof it was processed on the receiving chain and there is no more need to store this information.



3.6 Relay Process

The blockchain itself only records the *intention* to send the given message to the recipient chain, it doesn't make any network connections as that would add unbounded delays and non-determinism into the state machine. We define the concept of a *relay* process that connects two chain by querying one for all proofs needed to prove outgoing messages and submit these 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. Many *relay* processes may run in parallel without violating any safety consideration. 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 send messages from A to B, without error handling. We must also concurrently run the relay of receipts from B back to A, in order to complete the cycle. Note that all reads of variables belonging to a chain imply queries and all function calls imply submitting a transaction to the blockchain.

```
while true
    pending := tail(A:q_B.send)
    received := tail(B:q_A.receive)
    if pending > received
        U_h := A:latestHeader
        B:updateHeader(U_h)
        for i :=received...pending
            k := (B, send, i)
            packet := A:M_k,v,h
            B:IBCreceive(A, packet)
    sleep(desiredLatency)
```

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.

In the presence of multiple concurrent relays, any given relay can perform local optimizations to minimize the number of headers it submits, but remember the frequency of header submissions defines the latency of the packet transfer.

Indeed, it is ideal if each user that 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. However, in order to avoid bottlenecks, a group may sponsor an account to pay fees for a public relayer that moves all unrelayed packets (perhaps with a high latency).

4 Optimizations

The above sections describe a secure messaging protocol that can handle all normal situations between two blockchains. It guarantees that all messages are processed exactly once and in order, and provides a mechanism for non-blocking atomic transactions spanning two blockchains. However, to increase efficiency over millions of messages with many possible failure modes on both sides of the connection, we can extend the protocol. These extensions allow us to clean up the receipt queue to avoid state bloat, as well as more gracefully recover from cases where large numbers of messages are not being relayed, or other failure modes in the remote chain.

4.1 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 receipt or waiting to send the message in the first place and then relaying it right after the cutoff to take advantage of different local clocks on the two chains.

One solution to this is to include a timeout in the IBC message itself. When sending it, one can specify a block height or timestamp on the **receiving** chain after which it is no longer valid. If the message is posted before the cutoff, it will be processed normally. If it is posted after that cutoff, it will be a guaranteed error on the receiving chain. Note that to make this secure, the timeout must be relative to a condition on the **receiving** chain, and the sending chain must have proof of the state of the receiving chain after the cutoff in order to release assets on the sending chain.

For a sending chain A and a receiving chain B, with $k=(\_, \_, i)$ for $A{:}q_{B.send}$ or $B{:}q_{A.receipt}$ we currently have the following guarantees:

$A{:}M_{k,v,h} = \varnothing$ *if message i was not sent before height h*
$A{:}M_{k,v,h} = \varnothing$ *if message i was sent and receipt received before height h*
(and the receipts for all messages j < i were also handled)
$A{:}M_{k,v,h} \neq \varnothing$ *otherwise (message result is not yet processed)*

$B{:}M_{k,v,h} = \varnothing$ *if message i was not received before height h*
$B{:}M_{k,v,h} \neq \varnothing$ *if message i was received before height h*
(and all messages j < i were received)

Based on these guarantees, 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 $timestamp(H_h) > maxTime$.

$V_{send} = (maxHeight, maxTime, type, data)$
$expired(H_h, V_{send}) \Rightarrow [true/false]$

We then update message handling in *IBCreceive*, so it doesn't even call the handler function if the timeout was reached, but rather directly writes and error in the receipt queue:

IBCreceive:

> ....
> $expired(latestHeader, v) \Rightarrow push(q_{S.receipt}, (None, TimeoutError))$;
> $v = (\_, \_, type, data) \Rightarrow (result, err) := f_{type}(data); push(q_{S.receipt}, (result, err))$;

and add a new *IBCtimeout* function to accept tail proofs to demonstrate that the message was not processed at some given header on the recipient chain. This allows the sender chain to assert timeouts locally.

$S:IBCtimeout(A, M_{k,v,h}) \Rightarrow match$
> $q_{A.send} = \varnothing \Rightarrow Error(\text{“unregistered sender”})$,
> $k = (\_, send, \_) \Rightarrow Error(\text{“must be a receipt”})$,
> $k = (d, \_, \_)$ and $d \neq S \Rightarrow Error(\text{“sent to a different chain”})$,
> $H_h \notin T_A \Rightarrow Error(\text{“must submit header for height h”})$,
> $not\ valid(H_h, M_{k,v,h}) \Rightarrow Error(\text{“invalid merkle proof”})$,
> $k = (S, receipt, tail) \Rightarrow match$
>> $tail \geq head(q_{S.send}) \Rightarrow Error(\text{“receipt exists, no timeout proof”})$
>> $not\ expired(peek(q_{S.send})) \Rightarrow Error(\text{“message timeout not yet reached”})$
>> $default \Rightarrow (\_, \_, type, data) := pop(q_{S.send}); rollback_{type}(data); Success$
> $default \Rightarrow Error(\text{“must be a tail proof”})$

which processes timeouts in order, and adds one more condition to the queues:

$A:M_{k,v,h} = \varnothing$ *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 messages, without waiting for chain B to process them and return a receipt. 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. If a blockchain requires a timeout on all messages, and imposes some reasonable upper limit (or just assigns it automatically), we can guarantee that if message *i* is not processed by the upper limit of the timeout period, then all previous messages 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.

4.2 Clean up

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 requirement 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.

The observant reader may also notice, that 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 message *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:

$B{:}M_{k,v,h} = \varnothing$ *if message i was not received before height h*
$B{:}M_{k,v,h} = \varnothing$ *if message i was provably resolved on the sending chain before height h*
$B{:}M_{k,v,h} \neq \varnothing$ *otherwise (if message i was processed before height h,*
and no ack of receipt from the sending chain)

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 above, we see that all messages $i <$ *head* have been fully processed, and all messages *head* $<= i <$ *tail* are awaiting processing. By proving a much advanced *head* of the *send queue*, we can demonstrate that the sending chain already handled all messages. Thus, we can safely advance our local *receipt queue* to the new head of the remote *send queue*.

S:IBCcleanup(A, $M_{k,v,h}$) ⇒ match

$q_{A.receipt}$ =⊘⇒Error("unknown sender"),

k = (_, send, _) ⇒Error("must be for the send queue"),

k = (d, _, _) and d ≠S⇒Error("sent to a different chain"),

k≠ (_, _, head) ⇒Error("Need a proof of the head of the queue"),

H_h ∉T_A ⇒Error("must submit header for height h"),

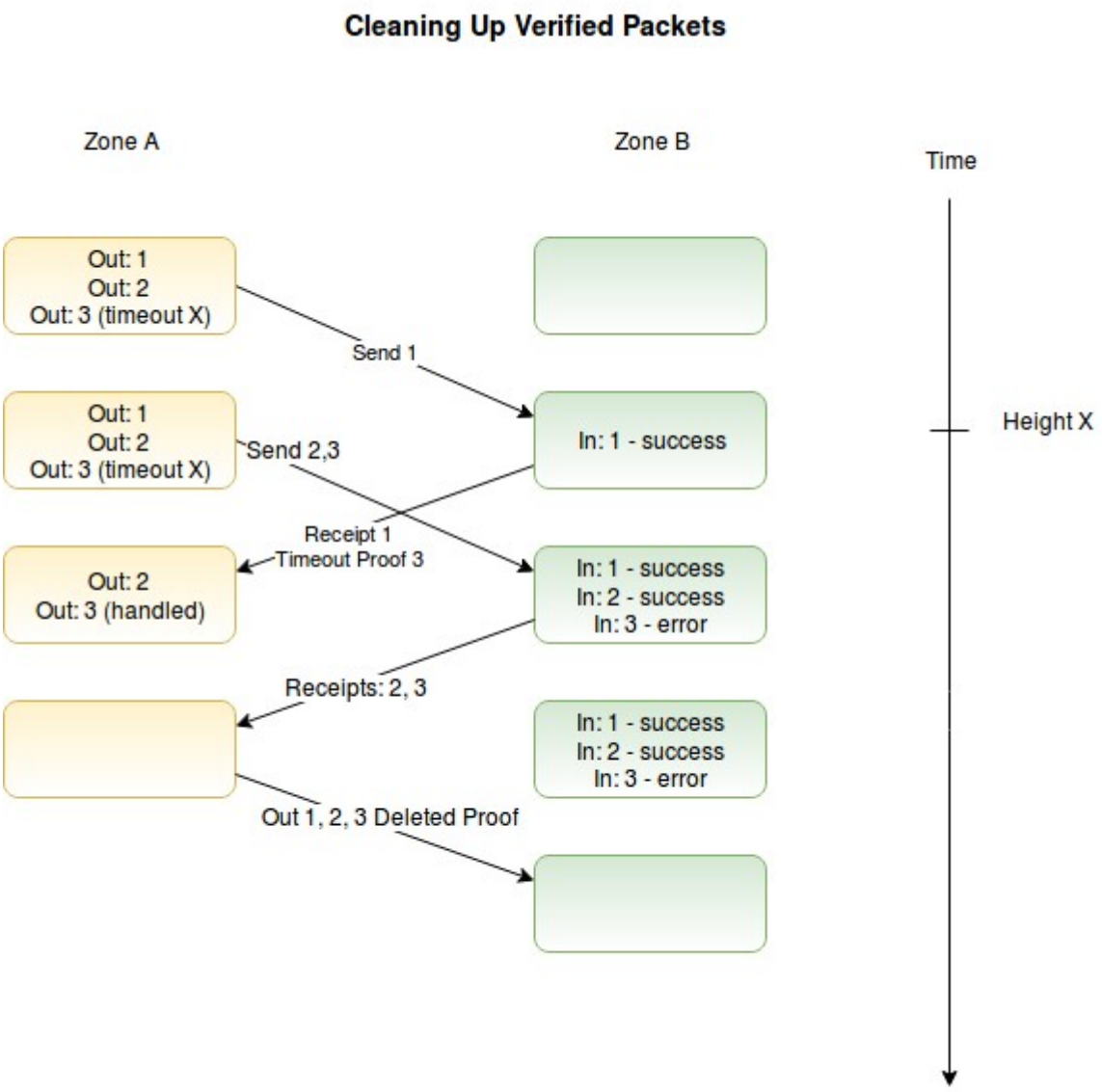
not valid(H_h ,$M_{k,v,h}$) ⇒Error("invalid merkle proof"),

head := v ⇒match

head <= head($q_{A.receipt}$) ⇒Error("cleanup must go forward"),

default ⇒advance($q_{A.receipt}$, head); Success

This allows us to invoke the *IBCcleanup* function to resolve all outstanding messages up to and including *head* 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 messages that have been processed by the receiving chain, but not yet posted to the sending chain, *tail(B:$q_{A.reciept}$) > head(A:$q_{B.send}$)*. As such, the *advance* function must not modify any messages between the head and the tail.



4.3 Handling Byzantine Failures

While every message is guaranteed reliable in the face of malicious nodes or relays, all guarantees break down when the entire blockchain on the other end of the connection exhibits byzantine faults. These can be in two forms: failures of the consensus mechanism (reversing “final” blocks), or failure at the application level (not performing the action defined by the message).

The IBC protocol can only detect byzantine faults at the consensus level, and is designed to halt with an error upon detecting any such fault. That is, if it ever sees two different headers for the same height (or any evidence that headers belong to different forks), then it must freeze the connection immediately. The resolution of the fault must be handled by the blockchain governance, as this is a serious incident and cannot be predefined.

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 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 (which will also have to adjust internally to wipe its view of the connection clean).

The other major byzantine action is at the application level. Let us assume messages 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 with a success code, it promises to credit X tokens to the account mentioned in the message. What if A isn't actually removing tokens from the supply, or if B is not actually crediting accounts?

Such application level issues cannot be proven in a generic sense, but must be handled individually by each application. The activity should be provable in some manner (as it is all in an auditable blockchain), but there are too many failure modes to attempt to enumerate, so we rely on the vigilance of the participants in the extremely rare case of a rogue blockchain. Of course, this misbehavior is provable and can negatively impact the value of the offending chain, providing economic incentives for any normal chain not to run malicious applications over IBC.

5 Conclusion

We have demonstrated a secure, performant, and flexible protocol for connecting two blockchains with complete finality using a secure, reliable messaging queue. The algorithm and semantics of all data types have been defined above, which provides a solid basis for reasoning about correctness and efficiency of the algorithm.

The observant reader may note that while we have defined a message queue protocol, we have not yet defined how to use that to transfer value within

the Cosmos ecosystem. 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 go and freely usable under the Apache license. For those wish to write an implementation of IBC in another language, or who want to analyze the specification further, the following appendixes define the exact message formats and binary encoding.

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 H_h, U_h, and X_h 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.

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

Appendix B: IBC Queue Format

The foundational data structure of the IBC protocol are the message 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])
V_{send} *= (maxHeight, maxTime, type, data)*
$V_{receipt}$ *= (result, [success|error code])*

```
message QueueName {
     // chain_id is which chain this queue is
     // associated with
     string chain_id = 1;
     enum Purpose {
          SEND = 0;
          RECEIPT = 1;
     }
     Purpose purpose = 2;
}


// StateKey is a key for the head/tail of a given queue
message StateKey {
     QueueName queue = 1;
     // both encode into one byte with varint encoding
     // never clash with 8 byte message indexes
     enum State {
          HEAD = 0;
          TAIL = 0x7f;
     }
     State state = 2;

}

// StateValue is the type stored under a StateKey
message StateValue {
   fixed64 index = 1;
}
```

```
// MessageKey is the key for message *index* in a given queue
message MessageKey {
     QueueName queue = 1;
     fixed64 index = 2;
}

// SendValue is stored under a MessageKey in the SEND queue
message SendValue {
     uint64 maxHeight = 1;
     google.protobuf.Timestamp maxTime = 2;
     // use kind instead of type to avoid keyword conflict
     bytes kind = 3;
     bytes data = 4;
}

// ReceiptValue is stored under a MessageKey in the RECEIPT queue
message ReceiptValue {
     // 0 is success, others are application-defined errors
     int32 errorCode = 1;
     // contains result on success, optional info on error
     bytes data = 2;
}
```

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 bytes can be deserialized and the contents interpreted by the protocol.

Appendix C: Merkle Proof Formats

A merkle tree (or a trie) generates one hash that can prove every element of the tree. Generating this hash starts with hashing the leaf nodes. Then hashing 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 continue hashing together the inner nodes at each level of the tree, until it reaches a root hash. Once you have a known root hash, you can prove key/value belongs to this tree by tracing the path to the value and revealing the (k-1) hashes for all the paths we did not take on each level. If this is new to you, you can read a basic introduction[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 sub-trees. 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 concatenating 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 for 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 their 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

10 https://en.wikipedia.org/wiki/Merkle_tree
11 https://chainpoint.org/

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.

For those who wish 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 freely available for use. 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 $log_2(n)$ 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 quite large state.

```
// HashOp is the hashing algorithm we use at each level
enum HashOp {
    RIPEMD160 = 0;
    SHA224 = 1;
    SHA256 = 2;
    SHA384 = 3;
    SHA512 = 4;
    SHA3_224 = 5;
    SHA3_256 = 6;
    SHA3_384 = 7;
    SHA3_512 = 8;
    SHA256_X2 = 9;
};
```

12 https://github.com/tendermint/iavl

```
// Op represents one hash in a chain of hashes.
// An operation takes the output of the last level and returns
// a hash for the next level:
// Op(last) => Operation(prefix + last + sufix)
//
// A simple left/right hash would simply set prefix=left or
// suffix=right and leave the other blank. However, one could
// also represent the a Patricia trie proof by setting
// prefix to the rlp encoding of all nodes before the branch
// we select, and suffix to all those after the one we select.
message Op {
    bytes prefix = 1;
    bytes suffix = 2;
    HashOp op = 3;
}

// Data is the end value stored, used to generate the initial
hash
message Data {
    bytes prefix = 1;
    bytes key = 2;
    bytes value = 3;
    HashOp op = 4;
    // If it is KeyValue, this is the data we want
    // If it is SubTree, key is name of the tree, value is root
hash
    // Expect another branch to follow
    enum DataType {
        KeyValue = 0;
        SubTree = 1;
    }
    DataType dataType = 5;
}

// Branch will hash data and then pass it through operations from
// last to first in order to calculate the root node.
//
// Visualize Branch as representing the data closest to root as
the
// first item, and the leaf as the last item.
message Branch {
    repeated Op operations = 1;
    Data data = 2;
}
```

```
// MerkleProof shows a veriable path from the data to
// a root hash (potentially spanning multiple sub-trees).
message MerkleProof {
     // identify the header this is rooted in
     string chainId = 1;
     uint64 height = 2;

     // this hash must match the header as well as the
     // calculation from below
     bytes rootHash = 3;

     // branches start from the value, and then may
     // include multiple subtree branches to embed it
     //
     // The first branch must have dataType KeyValue
     // Following branches must have dataType SubTree
     repeated Branch branches = 1;
}
```

Appendix D: Universal IBC Packets

The structures above can be used to define standard encodings for the basic IBC transactions that must be exposed by a blockchain: *IBCreceive*, *IBCreceipt*, *IBCtimeout*, and *IBCcleanup*. As mentioned above, these are not complete transactions to be posted as is to a blockchain, but rather the "data" content of a transaction, which must also contain fees, nonce, and signatures. The other IBC transaction types *IBCregisterChain*, *IBCupdateHeader*, and *IBCchangeValidators* are specific to the consensus engine and use unique encodings. We define the tendermint-specific format in the next section.

```
// IBCPacket sends a proven key/value pair from an IBCQueue.
// Depending on the type of message, we require a certain type
// of key (MessageKey at a given height, or StateKey).
//
// Includes src_chain and src_height to look up the proper
// header to verify the merkle proof.
message IBCPacket {
     // chain id it is coming from
     string src_chain = 1;
     // height for the header the proof belongs to
     uint64 src_height = 2;
     // the message type, which determines what key/value mean
     enum MsgType {
          RECEIVE = 0;
          RECEIPT = 1;
          TIMEOUT = 2;
          CLEANUP = 3;
     }
     MsgType msgType = 3;
     bytes key = 4;
     bytes value = 5;
     // the proof of the message
     MerkleProof proof = 6;
}
```

Appendix E: Tendermint Header Proofs

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).