The Interblockchain Communication Protocol

IBC Specification Team

4th January 2020 — **1.0.0-rc5**

Contents

[keywords, comments, strings]

1 Architectural Overview

1.1 Abstraction definitions

1.1.1 Actor

An *actor*, or a *user* (used interchangeably), is an entity interacting with the IBC protocol. An actor can be a human end-user, a module or smart contract running on a blockchain, or an off-chain relayer process capable of signing transactions.

1.1.2 Machine / Chain / Ledger

A *machine*, *chain*, *blockchain*, or *ledger* (used interchangeably), is a state machine (which may be a distributed ledger, or "blockchain", although a strict chain of blocks may not be required) implementing part or all of the IBC specification.

1.1.3 Relayer process

A *relayer process* is an off-chain process responsible for relaying IBC packet data & metadata between two or more machines by scanning their states & submitting transactions.

1.1.4 State Machine

The *state machine* of a particular chain defines the structure of the state as well as the set of rules which determines valid transactions that trigger state-transitions based on the current state agreed upon by the consensus algorithm of the chain.

1.1.5 Consensus

A *consensus* algorithm is the protocol used by the set of processes operating a distributed ledger to come to agreement on the same state, generally under the presence of a bounded number of Byzantine faults.

1.1.6 Consensus State

The *consensus state* is the set of information about the state of a consensus algorithm required to verify proofs about the output of that consensus algorithm (e.g. commitment roots in signed headers).

1.1.7 Commitment

A cryptographic *commitment* is a way to cheaply verify membership or non-membership of a key/value pair in a mapping, where the mapping can be committed to with a short witness string.

1.1.8 Header

A *header* is an update to the consensus state of a particular blockchain, including a commitment to the current state, that can be verified in a well-defined fashion by a "light client" algorithm.

1.1.9 CommitmentProof

A *commitment proof* is the proof structure which proves whether a particular key maps to a particular value in a committed-to set or not.

1.1.10 Handler Module

The IBC *handler module* is the module within the state machine which implements ICS 25, managing clients, connections, & channels, verifying proofs, and storing appropriate commitments for packets.

1.1.11 Routing Module

The IBC *routing module* is the module within the state machine which implements ICS 26, routing packets between the handler module and other modules on the host state machine which utilise the routing module's external interface.

1.1.12 Datagram

A *datagram* is an opaque bytestring transmitted over some physical network, and handled by the IBC routing module implemented in the ledger's state machine. In some implementations, the datagram may be a field in a ledger-specific transaction or message data structure which also contains other information (e.g. a fee for spam prevention, nonce for replay prevention, type identifier to route to the IBC handler, etc.). All IBC sub-protocols (such as opening a connection, creating a channel, sending a packet) are defined in terms of sets of datagrams and protocols for handling them through the routing module.

1.1.13 Connection

A *connection* is a set of persistent data structures on two chains that contain information about the consensus state of the other ledger in the connection. Updates to the consensus state of one chain changes the state of the connection object on the other chain.

1.1.14 Channel

A *channel* is a set of persistent data structures on two chains that contain metadata to facilitate packet ordering, exactly-once delivery, and replay prevention. Packets sent through a channel change its internal state. Channels are associated with connections in a many-to-one relationship — a single connection can have any number of associated channels, and all channels must have a single associated connection, which must have been created prior to the creation of the channel.

1.1.15 Packet

A *packet* is a particular data structure with sequence-related metadata (defined by the IBC specification) and an opaque value field referred to as the packet *data* (with semantics defined by the application layer, e.g. token amount and denomination). Packets are sent through a particular channel (and by extension, through a particular connection).

1.1.16 Module

A *module* is a sub-component of the state machine of a particular blockchain which may interact with the IBC handler and alter state according to the *data* field of particular IBC packets sent or received (minting or burning tokens, for example).

1.1.17 Handshake

A *handshake* is a particular class of sub-protocol involving multiple datagrams, generally used to initialise some common state on the two involved chains such as trusted states for each others' consensus algorithms.

1.1.18 Sub-protocol

Sub-protocols are defined as a set of datagram kinds and functions which must be implemented by the IBC handler module of the implementing blockchain.

Datagrams must be relayed between chains by an external relayer process. This relayer process is assumed to behave in an arbitrary manner — no safety properties are dependent on its behaviour, although progress is generally dependent on the existence of at least one correct relayer process.

IBC sub-protocols are reasoned about as interactions between two chains `A` and `B` — there is no prior distinction between these two chains and they are assumed to be executing the same, correct IBC protocol. `A` is simply by convention the chain which goes first in the sub-protocol and `B` the chain which goes second. Protocol definitions should generally avoid including `A` and `B` in variable names to avoid confusion (as the chains themselves do not know whether they are `A` or `B` in the protocol).

1.1.19 Authentication

Authentication is the property of ensuring that datagrams were in fact sent by a particular chain in the manner defined by the IBC handler.

1.2 Property definitions

1.2.1 Finality

Finality is the quantifiable assurance provided by a consensus algorithm that a particular block will not be reverted, subject to certain assumptions about the behaviour of the validator set. The IBC protocol requires finality, although it need not be absolute (for example, a threshold finality gadget for a Nakamoto consensus algorithm will provide finality subject to economic assumptions about how miners behave).

1.2.2 Misbehaviour

Misbehaviour is a class of consensus fault defined by a consensus algorithm & detectable (possibly also attributable) by the light client of that consensus algorithm.

1.2.3 Equivocation

Equivocation is a particular class of consensus fault committed by a validator or validators which sign votes on multiple different successors to a single block in an invalid manner. All equivocations are misbehaviours.

1.2.4 Data availability

Data availability is the ability of off-chain relayer processes to retrieve data in the state of a machine within some time bound.

1.2.5 Data confidentiality

Data confidentiality is the ability of the host state machine to refuse to make particular data available to particular parties without impairing the functionality of the IBC protocol.

1.2.6 Non-repudiability

Non-repudiability is the inability of a machine to successfully dispute having sent a particular packet or committed a particular state. IBC is a non-repudiable protocol, modulo data confidentiality choices made by state machines.

1.2.7 Consensus liveness

Consensus liveness is the continuance of block production by the consensus algorithm of a particular machine.

1.2.8 Transactional liveness

Transactional liveness is the continued confirmation of incoming transactions (which transactions should be clear by context) by the consensus algorithm of a particular machine. Transactional liveness requires consensus liveness, but consensus liveness does not necessarily provide transactional liveness. Transactional liveness implies censorship resistance.

1.2.9 Bounded consensus liveness

Bounded consensus liveness is consensus liveness within a particular bound.

1.2.10 Bounded transactional liveness

Bounded transactional liveness is transactional liveness within a particular bound.

1.2.11 Exactly-once safety

Exactly-once safety is the property that a packet is confirmed no more than once (and generally exactly-once assuming eventual transactional liveness).

1.2.12 Deliver-or-timeout safety

Deliver-or-timeout safety is the property that a packet will either be delivered & executed or will timeout in a way that can be proved back to the sender.

1.2.13 Constant (w.r.t. complexity)

Constant, when referring to space or time complexity, means `O(1)`.

1.2.14 Succinct

Succinct, when referring to space or time complexity, means `O(poly(log n))` or better.

1.3 What is IBC?

The *inter-blockchain communication protocol* is a reliable & secure inter-module communication protocol, where modules are deterministic processes that run on independent machines, including replicated state machines (like “blockchains” or “distributed ledgers”).

IBC can be used by any application which builds on top of reliable & secure inter-module communication. Example applications include cross-chain asset transfer, atomic swaps, multi-chain smart contracts (with or without mutually comprehensible VMs), and data & code sharding of various kinds.

1.4 What is IBC not?

IBC is not an application-layer protocol: it handles data transport, authentication, and reliability only.

IBC is not an atomic-swap protocol: arbitrary cross-chain data transfer and computation is supported.

IBC is not a token transfer protocol: token transfer is a possible application-layer use of the IBC protocol.

IBC is not a sharding protocol: there is no single state machine being split across chains, but rather a diverse set of different state machines on different chains which share some common interfaces.

IBC is not a layer-two scaling protocol: all chains implementing IBC exist on the same "layer", although they may occupy different points in the network topology, and there is not necessarily a single root chain or single validator set.

1.5 Motivation

The two predominant blockchains at the time of writing, Bitcoin and Ethereum, currently support about seven and about twenty transactions per second respectively. Both have been operating at capacity in recent past despite still being utilised primarily by a user-base of early-adopter enthusiasts. Throughput is a limitation for most blockchain use cases, and throughput limitations are a fundamental limitation of distributed state machines, since every (validating) node in the network must process every transaction (modulo future zero-knowledge constructions, which are out-of-scope of this specification at present), store all state, and communicate with other validating nodes. Faster consensus algorithms, such as Tendermint, may increase throughput by a large constant factor but will be unable to scale indefinitely for this reason. In order to support the transaction throughput, application diversity, and cost efficiency required to facilitate wide deployment of distributed ledger applications, execution and storage must be split across many independent consensus instances which can run concurrently.

One design direction is to shard a single programmable state machine across separate chains, referred to as "shards", which execute concurrently and store disjoint partitions of the state. In order to reason about safety and liveness, and in order to correctly route data and code between shards, these designs must take a "top-down approach" — constructing a particular network topology, featuring a single root ledger and a star or tree of shards, and engineering protocol rules & incentives to enforce that topology. This approach possesses advantages in simplicity and predictability, but faces hard technical problems, requires the adherence of all shards to a single validator set (or randomly elected subset thereof) and a single state machine or mutually comprehensible VM, and may face future problems in social scalability due to the necessity of reaching global consensus on alterations to the network topology.

Furthermore, any single consensus algorithm, state machine, and unit of Sybil resistance may fail to provide the requisite levels of security and versatility. Consensus instances are limited in the number of independent operators they can support, meaning that the amortised benefits from corrupting any particular operator increase as the value secured by the consensus instance increases — while the cost to corrupt the operator, which will always reflect the cheapest path (e.g. physical key exfiltration or social engineering), likely cannot scale indefinitely. A single global state machine must cater to the common denominator of a diverse application set, making it less well-suited for any particular application than a specialised state machine would be. Operators of a single consensus instance may abuse their privileged position to extract rent from applications which cannot easily elect to exit. It would be preferable to construct a mechanism by which separate, sovereign consensus instances & state machines can safely, voluntarily interact while sharing only a minimum requisite common interface.

The *interblockchain communication protocol* takes a different approach to a differently formulated version of the scaling & interoperability problems: enabling safe, reliable interoperation of a network of heterogeneous distributed ledgers, arranged in an unknown topology, preserving secrecy where possible, where the ledgers can diversify, develop, and rearrange independently of each other or of a particular imposed topology or state machine design. In a wide, dynamic network of interoperating chains, sporadic Byzantine faults are expected, so the protocol must also detect, mitigate, and contain the potential damage of Byzantine faults in accordance with the requirements of the applications & ledgers involved. For a longer list of design principles, see here.

To facilitate this heterogeneous interoperation, the interblockchain communication protocol takes a "bottom-up" approach, specifying the set of requirements, functions, and properties necessary to implement interoperation between two ledgers, and then specifying different ways in which multiple interoperating ledgers might be composed which preserve the requirements of higher-level protocols and occupy different points in the safety/speed tradeoff space. IBC thus presumes nothing about and requires nothing of the overall network topology, and of the implementing ledgers requires only that a known, minimal set of functions are available and properties fulfilled. Indeed, ledgers within IBC are defined as their light client consensus validation functions, thus expanding the range of what a "ledger" can be to include single machines and complex consensus algorithms alike.

IBC is an end-to-end, connection-oriented, stateful protocol for reliable, optionally ordered, authenticated communication between modules on separate machines. IBC implementations are expected to be co-resident with higher-level modules and protocols on the host state machine. State machines hosting IBC must provide a certain set of functions for consensus transcript verification and cryptographic commitment proof generation, and IBC packet relayers (off-chain processes) are expected to have access to network protocols and physical data-links as required to read the state of one machine and submit data to another.

1.6 Scope

IBC handles authentication, transport, and ordering of structured data packets relayed between modules on separate machines. The protocol is defined between modules on two machines, but designed for safe simultaneous use between any number of modules on any number of machines connected in arbitrarily topologies.

1.7 Interfaces

IBC sits between modules — smart contracts, other state machine components, or otherwise independent pieces of application logic on state machines — on one side, and underlying consensus protocols, machines, and network infrastructure (e.g. TCP/IP), on the other side.

IBC provides to modules a set of functions much like the functions which might be provided to a module for interacting with another module on the same state machine: sending data packets and receiving data packets on an established connection & channel (primitives for authentication & ordering, see definitions) — in addition to calls to manage the protocol state: opening and closing connections and channels, choosing connection, channel, and packet delivery options, and inspecting connection & channel status.

IBC assumes functionalities and properties of the underlying consensus protocols and machines as defined in ICS 2, primarily finality (or thresholding finality gadgets), cheaply-verifiable consensus transcripts, and simple key/value store functionality. On the network side, IBC requires only eventual data delivery — no authentication, synchrony, or ordering properties are assumed (these properties are defined precisely later on).

1.7.1 Protocol relations

```
+------------------------------+                           +------------------------------+
| Distributed Ledger A         |                           | Distributed Ledger B         |
|                              |                           |                              |
| +--------------------------+ |                           | +--------------------------+ |
| | State Machine            | |                           | | State Machine            | |
| |                          | |                           | |                          | |
| | +----------+     +-----+ | |        +---------+        | | +-----+     +----------+ | |
| | | Module A | <-> | IBC | | | <----> | Relayer | <----> | | | IBC | <-> | Module B | | |
| | +----------+     +-----+ | |        +---------+        | | +-----+     +----------+ | |
| +--------------------------+ |                           | +--------------------------+ |
+------------------------------+                           +------------------------------+
```

1.8 Operation

The primary purpose of IBC is to provide reliable, authenticated, ordered communication between modules running on independent host machines. This requires protocol logic in the following areas:

- Data relay
- Data confidentiality & legibility
- Reliability
- Flow control
- Authentication
- Statefulness

- Multiplexing
- Serialisation

The following paragraphs outline the protocol logic within IBC for each area.

1.8.1 Data relay

In the IBC architecture, modules are not directly sending messages to each other over networking infrastructure, but rather creating messages to be sent which are then physically relayed by monitoring "relayer processes". IBC assumes the existence of a set of relayer processes with access to an underlying network protocol stack (likely TCP/IP, UDP/IP, or QUIC/IP) and physical interconnect infrastructure. These relayer processes monitor a set of machines implementing the IBC protocol, continuously scanning the state of each machine and executing transactions on another machine when outgoing packets have been committed. For correct operation and progress in a connection between two machines, IBC requires only that at least one correct and live relayer process exists which can relay between the machines.

1.8.2 Data confidentiality & legibility

The IBC protocol requires only that the minimum data necessary for correct operation of the IBC protocol be made available & legible (serialised in a standardised format), and the state machine may elect to make that data available only to specific relayers (though the details thereof are out-of-scope of this specification). This data consists of consensus state, client, connection, channel, and packet information, and any auxiliary state structure necessary to construct proofs of inclusion or exclusion of particular key/value pairs in state. All data which must be proved to another machine must also be legible; i.e., it must be serialised in a format defined by this specification.

1.8.3 Reliability

The network layer and relayer processes may behave in arbitrary ways, dropping, reordering, or duplicating packets, purposely attempting to send invalid transactions, or otherwise acting in a Byzantine fashion. This must not compromise the safety or liveness of IBC. This is achieved by assigning a sequence number to each packet sent over an IBC connection (at the time of send), which is checked by the IBC handler (the part of the state machine implementing the IBC protocol) on the receiving machine, and providing a method for the sending machine to check that the receiving machine has in fact received and handled a packet before sending more packets or taking further action. Cryptographic commitments are used to prevent datagram forgery: the sending machine commits to outgoing packets, and the receiving machine checks these commitments, so datagrams altered in transit by a relayer will be rejected. IBC also supports unordered channels, which do not enforce ordering of packet receives relative to sends but still enforce exactly-once delivery.

1.8.4 Flow control

IBC does not provide specific provisions for compute-level or economic-level flow control. The underlying machines will have compute throughput limitations and flow control mechanisms of their own (such as "gas" markets). Application-level economic flow control — limiting the rate of particular packets according to their content — may be useful to ensure security properties (limiting the value on a single machine) and contain damage from Byzantine faults (allowing a challenge period to prove an equivocation, then closing a connection). For example, an application transferring value over an IBC channel might want to limit the rate of value transfer per block to limit damage from potential Byzantine behaviour. IBC provides facilities for modules to reject packets and leaves particulars up to the higher-level application protocols.

1.8.5 Authentication

All datagrams in IBC are authenticated: a block finalised by the consensus algorithm of the sending machine must commit to the outgoing packet via a cryptographic commitment, and the receiving chain's IBC handler must verify both the consensus transcript and the cryptographic commitment proof that the datagram was sent before acting upon it.

1.8.6 Statefulness

Reliability, flow control, and authentication as described above require that IBC initialises and maintains certain status information for each datastream. This information is split between two abstractions: connections & channels. Each connection object contains information about the consensus state of the connected machine. Each channel, specific to a pair of modules, contains information concerning negotiated encoding & multiplexing options and state & sequence numbers. When two modules wish to communicate, they must locate an existing connection & channel between their two machines, or initialise a new connection & channels if none yet exists. Initialising connections & channels requires a multi-step handshake which, once complete, ensures that only the two intended machines are connected, in the case of connections, and ensures that two modules are connected and that future datagrams relayed will be authenticated, encoded, and sequenced as desired, in the case of channels.

1.8.7 Multiplexing

To allow for many modules within a single host machine to use an IBC connection simultaneously, IBC provides a set of channels within each connection, which each uniquely identify a datastream over which packets can be sent in order (in the case of an ordered module), and always exactly once, to a destination module on the receiving machine. Channels are usually expected to be associated with a single module on each machine, but one-to-many and many-to-one channels are also possible. The number of channels is unbounded, facilitating concurrent throughput limited only by the throughput of the underlying machines with only a single connection necessary to track consensus information (and consensus transcript verification cost thus amortised across all channels using the connection).

1.8.8 Serialisation

IBC serves as the interface boundary between otherwise mutually incomprehensible machines, and must provide the requisite mutual comprehensibility of the minimal set of data structure encodings & datagram formats in order to allow two machines which both correctly implement the protocol to understand each other. For this purpose, the IBC specification defines canonical encodings of data structures which must be serialised and relayed or checked in proofs between two machines talking over IBC, provided in proto3 format in this repository.

> Note that a subset of proto3 which provides canonical encodings (the same structure always serialises to the same bytes) must be used. Maps and unknown fields are thus prohibited.

1.9 Dataflow

IBC can be conceptualised as a layered protocol stack, through which data flows top-to-bottom (when sending IBC packets) and bottom-to-top (when receiving IBC packets).

The “handler” is the part of the state machine implementing the IBC protocol, which is responsible for translating calls from modules to and from packets and routing them appropriately to and from channels & connections.

Consider the path of an IBC packet between two chains — call them A and B:

1.9.1 Diagram

```
+---------------------------------------------------------------------------------------------+
| Distributed Ledger A                                                                        |
|                                                                                             |
| +----------+     +----------------------------------------------------------+               |
| |          |     | IBC Module                                               |               |
| | Module A | --> |                                                          | --> Consensus |
| |          |     | Handler --> Packet --> Channel --> Connection --> Client |               |
| +----------+     +----------------------------------------------------------+               |
+---------------------------------------------------------------------------------------------+

    +---------+
==> | Relayer | ==>
    +---------+

+--------------------------------------------------------------------------------------------+
| Distributed Ledger B                                                                       |
|                                                                                            |
|               +---------------------------------------------------------+     +----------+ |
|               | IBC Module                                              |     |          | |
| Consensus --> |                                                         | --> | Module B | |
|               | Client -> Connection --> Channel --> Packet --> Handler |     |          | |
|               +---------------------------------------------------------+     +----------+ |
+--------------------------------------------------------------------------------------------+
```

1.9.2 Steps

1. On chain *A*
    1. Module (application-specific)
    2. Handler (parts defined in different ICSs)
    3. Packet (defined in ICS 4)
    4. Channel (defined in ICS 4)
    5. Connection (defined in ICS 3)
    6. Client (defined in ICS 2)
    7. Consensus (confirms the transaction with the outgoing packet)
2. Off-chain
    1. Relayer (defined in ICS 18)
3. On chain *B*
    1. Consensus (confirms the transaction with the incoming packet)
    2. Client (defined in ICS 2)
    3. Connection (defined in ICS 3)
    4. Channel (defined in ICS 4)
    5. Packet (defined in ICS 4)
    6. Handler (parts defined in different ICSs)
    7. Module (application-specific)

1.10 Versatility

IBC is designed to be a *versatile* protocol. The protocol supports *heterogeneous* blockchains whose state machines implement different semantics in different languages. Applications written on top of IBC can be *composed* together, and IBC protocol steps themselves can be *automated*.

1.10.1 Heterogeneity

IBC can be implemented by any consensus algorithm and state machine with a basic set of requirements (fast finality, constant-size state commitments, and succinct commitment proofs). The protocol handles data authentication, transport, and ordering — common requirements of any multi-chain application — but is agnostic to the semantics of the application itself. Heterogeneous chains connected over IBC must understand a compatible application-layer "interface" (such as for transferring tokens), but once across the IBC interface handler, the state machines can support arbitrary bespoke functionality (such as shielded transactions).

1.10.2 Composability

Applications written on top of IBC can be composed together by both protocol developers and users. IBC defines a set of primitives for authentication, transport, and ordering, and a set of application-layer standards for asset & data semantics. Chains which support compatible standards can be connected together and transacted between by any user who elects to open a connection (or reuse a connection), and assets & data can be relayed across multiple chains both automatically ("multi-hop") and manually (by sending several IBC relay transactions in sequence).

1.10.3 Automatability

The “users”, or “actors”, in IBC — who initiate connections, create channels, send packets, report Byzantine fraud, etc. — may be, but need not be, human. Modules, smart contracts, and automated off-chain processes can make use of the protocol (subject to e.g. gas costs to charge for computation) and take actions on their own or in concert. Complex interactions across multiple chains (such as the three-step connection opening handshake or multi-hop token transfers) are designed such that all but the single initiating action can be abstracted away from the user. Eventually, it may be possible to automatically spin up a new blockchain (modulo physical infrastructure provisioning), start IBC connections, and make use of the new chain’s state machine & throughput entirely automatically.

1.11 Modularity

IBC is designed to be a *modular* protocol. The protocol is constructed as a series of layered components with explicit security properties & requirements. Implementations of a component at a particular layer can vary (such as a different consensus algorithm or connection opening procedure) as long as they provide the requisite properties to the higher layers (such as finality, < 1/3 Byzantine safety, or embedded trusted states on two chains). State machines need only understand compatible subsets of the IBC protocol (e.g. lite client verification algorithms for each other’s consensus) in order to safely interact.

1.12 Locality

IBC is designed to be a *local* protocol, meaning that only information about the two connected chains is necessary to reason about the security and correctness of a bidirectional IBC connection. Security requirements of the authentication primitives refer only to consensus algorithms and validator sets of the blockchains involved in the connection, and a blockchains maintaining a set of IBC connections need only understand the state of the chains to which it is connected (no matter which other chains those chains are connected to).

1.12.1 Locality of communication & information

IBC makes no assumptions, and relies upon no characteristics, of the topological structure of the network of blockchains in which it is operating. No view of the global network-of-blockchains topology is required: security & correctness can be reasoned about at the level of a single connection between two chains, and by composition reasoned about for sub-graphs in the network topology. Users and chains can reason about their assumptions and risks given information about only part of the network graph of blockchains they know and assume to be correct (to variable degrees).

There is no necessary “root chain” in IBC — some sub-graphs of the global network may evolve into a hub-spoke structure, others may remain tightly connected, others still may take on more exotic topologies. Channels are end-to-end; in the first version IBC will only support one-hop paths, but multi-hop paths will be supported in the future (though automatic routing is not necessarily likely or safe due to the consensus algorithm correctness assumptions involved).

Application data, however, may have salient non-local properties which users of the protocol will need to pay attention to, such as the original source zone of a token which might have been sent on a complex multi-hop path, the original stake & identity of a validator offering their services through cross-chain validation, or the original smart contract with which a particular object-capability key managing a non-fungible token is associated. These non-local properties do not need to be understood by the IBC protocol itself, but they will need to be reasoned about by users and higher-level applications.

1.12.2 Locality of correctness assumptions & security

Users of IBC — at the blockchain level and at the human or smart contract level — choose which consensus algorithms, state machines, and validator sets they “assume to be correct” (to behave in a particular way, e.g. < 1/3 Byzantine) and in which ways they assume correctness. Assuming the IBC protocol is implemented correctly, users are never exposed to risks of application-level invariant violations (such as asset inflation) due to Byzantine behaviour or faulty state machines transitions committed by validator sets or blockchains they did not explicitly decide to assume to be correct. This is particularly important in the expected large network topology of interconnected blockchains, where some number of blockchains and validator sets can be expected to be Byzantine occasionally — IBC, implemented conservatively, bounds the risk and limits the possible damage incurred.

1.12.3 Locality of permissioning

Actions in IBC — such as opening a connection, creating a channel, or sending a packet — are permissioned locally by the state machines and actors involved in a particular connection between two chains. Individual chains could choose to require approval from a permissioning mechanism (such as governance) for specific application-layer actions (such as delegated-security slashing), but for the base protocol, actions are permissionless (modulo gas & storage costs) — by default, connections can be opened, channels created, and packets sent without any approval process. Of course, users themselves must inspect the state & consensus of each IBC connection and decide whether it is safe to used (based e.g. on the trusted states stored).

1.13 Efficiency

IBC is designed to be an *efficient* protocol: the amortised cost of interchain data & asset relay should be mostly comprised of the cost of the underlying state transitions or operations associated with packets (such as transferring tokens), plus some small constant overhead.

2 ICS 001 - ICS Standard

2.1 What is an ICS?

An inter-chain standard (ICS) is a design document describing a particular protocol, standard, or feature expected to be of use to the Cosmos ecosystem. An ICS should list the desired properties of the standard, explain the design rationale, and provide a concise but comprehensive technical specification. The primary ICS author is responsible for pushing the proposal through the standardisation process, soliciting input and support from the community, and communicating with relevant stakeholders to ensure (social) consensus.

The inter-chain standardisation process should be the primary vehicle for proposing ecosystem-wide protocols, changes, and features, and ICS documents should persist after consensus as a record of design decisions and an information repository for future implementers.

Inter-chain standards should *not* be used for proposing changes to a particular blockchain (such as the Cosmos Hub), specifying implementation particulars (such as language-specific data structures), or debating governance proposals on existing Cosmos blockchains (although it is possible that individual blockchains in the Cosmos ecosystem may utilise their governance processes to approve or reject inter-chain standards).

2.2 Components

An ICS consists of a header, synopsis, specification, history log, and copyright notice. All top-level sections are required. References should be included inline as links, or tabulated at the bottom of the section if necessary.

2.2.1 Header

An ICS header contains metadata relevant to the ICS.

Required fields `ics: #` - ICS number (assigned sequentially)

`title` - ICS title (keep it short & sweet)

`stage` - Current ICS stage, see PROCESS.md for the list of possible stages.

See README.md for a description of the ICS acceptance stages.

`category` - ICS category, one of the following: - `meta` - A standard about the ICS process - `IBC/TAO` - A standard about an inter-blockchain communication system core transport, authentication, and ordering layer protocol. - `IBC/APP` - A standard about an inter-blockchain communication system application layer protocol.

`author` - ICS author(s) & contact information (in order of preference: email, GitHub handle, Twitter handle, other contact methods likely to elicit response). The first author is the primary "owner" of the ICS and is responsible for advancing it through the standardisation process. Subsequent author ordering should be in order of contribution amount.

`created` - Date ICS was first created (`YYYY-MM-DD`)

`modified` - Date ICS was last modified (`YYYY-MM-DD`)

Optional fields `requires` - Other ICS standards, referenced by number, which are required or depended upon by this standard.

`required-by` - Other ICS standards, referenced by number, which require or depend upon this standard.

`replaces` - Another ICS standard replaced or supplanted by this standard, if applicable.

`replaced-by` - Another ICS standard which replaces or supplants this standard, if applicable.

2.2.2 Synopsis

Following the header, an ICS should include a brief (~200 word) synopsis providing a high-level description of and rationale for the specification.

2.2.3 Specification

The specification section is the main component of an ICS, and should contain protocol documentation, design rationale, required references, and technical details where appropriate.

Sub-components The specification may have any or all of the following sub-components, as appropriate to the particular ICS. Included sub-components should be listed in the order specified here.

- *Motivation* - A rationale for the existence of the proposed feature, or the proposed changes to an existing feature.
- *Definitions* - A list of new terms or concepts utilised in this ICS or required to understand this ICS. Any terms not defined in the top-level "docs" folder must be defined here.
- *Desired Properties* - A list of the desired properties or characteristics of the protocol or feature specified, and expected effects or failures when the properties are violated.
- *Technical Specification* - All technical details of the proposed protocol including syntax, semantics, sub-protocols, data structures, algorithms, and pseudocode as appropriate. The technical specification should be detailed enough such that separate correct implementations of the specification without knowledge of each other are compatible.
- *Backwards Compatibility* - A discussion of compatibility (or lack thereof) with previous feature or protocol versions.
- *Forwards Compatibility* - A discussion of compatibility (or lack thereof) with future possible or expected features or protocol versions.
- *Example Implementation* - A concrete example implementation or description of an expected implementation to serve as the primary reference for implementers.
- *Other Implementations* - A list of candidate or finalised implementations (external references, not inline).

2.2.4 History

An ICS should include a history section, listing any inspiring documents and a plaintext log of significant changes.

See an example history section below.

2.2.5 Copyright

An ICS should include a copyright section waiving rights via Apache 2.0.

2.3 Formatting

2.3.1 General

ICS specifications must be written in GitHub-flavoured Markdown.

For a GitHub-flavoured Markdown cheat sheet, see here. For a local Markdown renderer, see here.

2.3.2 Language

ICS specifications should be written in Simple English, avoiding obscure terminology and unnecessary jargon. For excellent examples of Simple English, please see the Simple English Wikipedia.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in specifications are to be interpreted as described in RFC 2119.

2.3.3 Pseudocode

Pseudocode in specifications should be language-agnostic and formatted in a simple imperative standard, with line numbers, variables, simple conditional blocks, for loops, and English fragments where necessary to explain further functionality such as scheduling timeouts. LaTeX images should be avoided because they are difficult to review in diff form.

Pseudocode for structs should be written in simple Typescript, as interfaces.

Example pseudocode struct:

```
interface Connection {
  state: ConnectionState
  version: Version
  counterpartyIdentifier: Identifier
  consensusState: ConsensusState
}
```

Pseudocode for algorithms should be written in simple Typescript, as functions.

Example pseudocode algorithm:

```
function startRound(round) {
  round_p = round
  step_p = PROPOSE
  if (proposer(h_p, round_p) === p) {
    if (validValue_p !== nil)
      proposal = validValue_p
    else
      proposal = getValue()
    broadcast( {PROPOSAL, h_p, round_p, proposal, validRound} )
  } else
    schedule(onTimeoutPropose(h_p, round_p), timeoutPropose(round_p))
}
```

2.4 History

This specification was significantly inspired by and derived from Ethereum's EIP 1, which was in turn derived from Bitcoin's BIP process and Python's PEP process. Antecedent authors are not responsible for any shortcomings of this ICS spec or the ICS process. Please direct all comments to the ICS repository maintainers.

Mar 4, 2019 - Initial draft finished and submitted as a PR

Mar 7, 2019 - Draft merged

Apr 11, 2019 - Updates to pseudocode formatting, add definitions subsection

Aug 17, 2019 - Clarifications to categories

2.5 Copyright

All content herein is licensed under Apache 2.0.

3 ICS 023 - Vector Commitments

3.1 Synopsis

A *vector commitment* is a construction that produces a constant-size, binding commitment to an indexed vector of elements and short membership and/or non-membership proofs for any indices & elements in the vector. This specification enumerates the functions and properties required of commitment constructions used in the IBC protocol. In particular, commitments utilised in IBC are required to be *positionally binding*: they must be able to prove existence or nonexistence of values at specific positions (indices).

3.1.1 Motivation

In order to provide a guarantee of a particular state transition having occurred on one chain which can be verified on another chain, IBC requires an efficient cryptographic construction to prove inclusion or non-inclusion of particular values at particular paths in state.

3.1.2 Definitions

The *manager* of a vector commitment is the actor with the ability and responsibility to add or remove items from the commitment. Generally this will be the state machine of a blockchain.

The *prover* is the actor responsible for generating proofs of inclusion or non-inclusion of particular elements. Generally this will be a relayer (see ICS 18).

The *verifier* is the actor who checks proofs in order to verify that the manager of the commitment did or did not add a particular element. Generally this will be an IBC handler (module implementing IBC) running on another chain.

Commitments are instantiated with particular *path* and *value* types, which are assumed to be arbitrary serialisable data.

A *negligible function* is a function that grows more slowly than the reciprocal of every positive polynomial, as defined here.

3.1.3 Desired Properties

This document only defines desired properties, not a concrete implementation — see "Properties" below.

3.2 Technical Specification

3.2.1 Datatypes

A commitment construction MUST specify the following datatypes, which are otherwise opaque (need not be introspected) but MUST be serialisable:

Commitment State A `CommitmentState` is the full state of the commitment, which will be stored by the manager.

```
type CommitmentState = object
```

Commitment Root A `CommitmentRoot` commits to a particular commitment state and should be constant-size.

In certain commitment constructions with constant-size states, `CommitmentState` and `CommitmentRoot` may be the same type.

```
type CommitmentRoot = object
```

Commitment Path A `CommitmentPath` is the path used to verify commitment proofs, which can be an arbitrary structured object (defined by a commitment type). It must be computed by `applyPrefix` (defined below).

```
type CommitmentPath = object
```

Prefix A `CommitmentPrefix` defines a store prefix of the commitment proof. It is applied to the path before the path is passed to the proof verification functions.

```
type CommitmentPrefix = object
```

The function `applyPrefix` constructs a new commitment path from the arguments. It interprets the path argument in the context of the prefix argument.

For two `(prefix, path)` tuples, `applyPrefix(prefix, path)` MUST return the same key only if the tuple elements are equal.

`applyPrefix` MUST be implemented per `Path`, as `Path` can have different concrete structures. `applyPrefix` MAY accept multiple `CommitmentPrefix` types.

The `CommitmentPath` returned by `applyPrefix` does not need to be serialisable (e.g. it might be a list of tree node identifiers), but it does need an equality comparison.

```
type applyPrefix = (prefix: CommitmentPrefix, path: Path) => CommitmentPath
```

Proof A `CommitmentProof` demonstrates membership or non-membership for an element or set of elements, verifiable in conjunction with a known commitment root. Proofs should be succinct.

```
type CommitmentProof = object
```

3.2.2 Required functions

A commitment construction MUST provide the following functions, defined over paths as serialisable objects and values as byte arrays:

```
type Path = string

type Value = []byte
```

Initialisation The `generate` function initialises the state of the commitment from an initial (possibly empty) map of paths to values.

```
type generate = (initial: Map<Path, Value>) => CommitmentState
```

Root calculation The `calculateRoot` function calculates a constant-size commitment to the commitment state which can be used to verify proofs.

```
type calculateRoot = (state: CommitmentState) => CommitmentRoot
```

Adding & removing elements The `set` function sets a path to a value in the commitment.

```
type set = (state: CommitmentState, path: Path, value: Value) => CommitmentState
```

The `remove` function removes a path and associated value from a commitment.

```
type remove = (state: CommitmentState, path: Path) => CommitmentState
```

Proof generation The `createMembershipProof` function generates a proof that a particular commitment path has been set to a particular value in a commitment.

```
type createMembershipProof = (state: CommitmentState, path: CommitmentPath, value: Value) =>
    CommitmentProof
```

The `createNonMembershipProof` function generates a proof that a commitment path has not been set to any value in a commitment.

```
type createNonMembershipProof = (state: CommitmentState, path: CommitmentPath) => CommitmentProof
```

Proof verification The `verifyMembership` function verifies a proof that a path has been set to a particular value in a commitment.

```
type verifyMembership = (root: CommitmentRoot, proof: CommitmentProof, path: CommitmentPath, value:
    Value) => boolean
```

The `verifyNonMembership` function verifies a proof that a path has not been set to any value in a commitment.

```
type verifyNonMembership = (root: CommitmentRoot, proof: CommitmentProof, path: CommitmentPath) =>
    boolean
```

3.2.3 Optional functions

A commitment construction MAY provide the following functions:

The `batchVerifyMembership` function verifies a proof that many paths have been set to specific values in a commitment.

```
type batchVerifyMembership = (root: CommitmentRoot, proof: CommitmentProof, items: Map<CommitmentPath,
    Value>) => boolean
```

The `batchVerifyNonMembership` function verifies a proof that many paths have not been set to any value in a commitment.

```
type batchVerifyNonMembership = (root: CommitmentRoot, proof: CommitmentProof, paths: Set<
    CommitmentPath>) => boolean
```

If defined, these functions MUST produce the same result as the conjunctive union of `verifyMembership` and `verifyNonMembership` respectively (efficiency may vary):

```
batchVerifyMembership(root, proof, items) ===
  all(items.map((item) => verifyMembership(root, proof, item.path, item.value)))
```

```
batchVerifyNonMembership(root, proof, items) ===
  all(items.map((item) => verifyNonMembership(root, proof, item.path)))
```

If batch verification is possible and more efficient than individual verification of one proof per element, a commitment construction SHOULD define batch verification functions.

3.2.4 Properties & Invariants

Commitments MUST be *complete*, *sound*, and *position binding*. These properties are defined with respect to a security parameter `k`, which MUST be agreed upon by the manager, prover, and verifier (and often will be constant for the commitment algorithm).

Completeness Commitment proofs MUST be *complete*: path => value mappings which have been added to the commitment can always be proved to have been included, and paths which have not been included can always be proved to have been excluded, except with probability negligible in `k`.

For any prefix `prefix` and any path `path` last set to a value `value` in the commitment `acc`,

```
root = getRoot(acc)
proof = createMembershipProof(acc, applyPrefix(prefix, path), value)
```

```
Probability(verifyMembership(root, proof, applyPrefix(prefix, path), value) === false) negligible in k
```

For any prefix `prefix` and any path `path` not set in the commitment `acc`, for all values of `proof` and all values of `value`,

```
root = getRoot(acc)
proof = createNonMembershipProof(acc, applyPrefix(prefix, path))
```

```
Probability(verifyNonMembership(root, proof, applyPrefix(prefix, path)) === false) negligible in k
```

Soundness Commitment proofs MUST be *sound*: path => value mappings which have not been added to the commitment cannot be proved to have been included, or paths which have been added to the commitment excluded, except with probability negligible in a configurable security parameter `k`.

For any prefix `prefix` and any path `path` last set to a value `value` in the commitment `acc`, for all values of `proof`,

```
Probability(verifyNonMembership(root, proof, applyPrefix(prefix, path)) === true) negligible in k
```

For any prefix `prefix` and any path `path` not set in the commitment `acc`, for all values of `proof` and all values of `value`,

```
Probability(verifyMembership(root, proof, applyPrefix(prefix, path), value) === true) negligible in k
```

Position binding Commitment proofs MUST be *position binding*: a given commitment path can only map to one value, and a commitment proof cannot prove that the same path opens to a different value except with probability negligible in k.

For any prefix `prefix` and any path `path` set in the commitment `acc`, there is one `value` for which:

```
root = getRoot(acc)
proof = createMembershipProof(acc, applyPrefix(prefix, path), value)
```

```
Probability(verifyMembership(root, proof, applyPrefix(prefix, path), value) === false) negligible in k
```

For all other values `otherValue` where `value !== otherValue`, for all values of `proof`,

```
Probability(verifyMembership(root, proof, applyPrefix(prefix, path), otherValue) === true) negligible
    in k
```

4 ICS 024 - Host Requirements

4.1 Synopsis

This specification defines the minimal set of interfaces which must be provided and properties which must be fulfilled by a state machine hosting an implementation of the interblockchain communication protocol.

4.1.1 Motivation

IBC is designed to be a common standard which will be hosted by a variety of blockchains & state machines and must clearly define the requirements of the host.

4.1.2 Definitions

4.1.3 Desired Properties

IBC should require as simple an interface from the underlying state machine as possible to maximise the ease of correct implementation.

4.2 Technical Specification

4.2.1 Module system

The host state machine must support a module system, whereby self-contained, potentially mutually distrusted packages of code can safely execute on the same ledger, control how and when they allow other modules to communicate with them, and be identified and manipulated by a "master module" or execution environment.

The IBC/TAO specifications define the implementations of two modules: the core "IBC handler" module and the "IBC relayer" module. IBC/APP specifications further define other modules for particular packet handling application logic. IBC requires that the "master module" or execution environment can be used to grant other modules on the host state machine access to the IBC handler module and/or the IBC routing module, but otherwise does not impose requirements on the functionality or communication abilities of any other modules which may be co-located on the state machine.

4.2.2 Paths, identifiers, separators

An `Identifier` is a bytestring used as a key for an object stored in state, such as a connection, channel, or light client. Identifiers MUST consist of alphanumeric characters only. Identifiers MUST be non-empty (of positive integer length).

A `Path` is a bytestring used as the key for an object stored in state. Paths MUST contain only identifiers, constant alphanumeric strings, and the separator `"/"`.

Identifiers are not intended to be valuable resources — to prevent name squatting, minimum length requirements or pseudorandom generation MAY be implemented, but particular restrictions are not imposed by this specification.

The separator `"/"` is used to separate and concatenate two identifiers or an identifier and a constant bytestring. Identifiers MUST NOT contain the `"/"` character, which prevents ambiguity.

Variable interpolation, denoted by curly braces, is used throughout this specification as shorthand to define path formats, e.g. `client/{clientIdentifier}/consensusState`.

4.2.3 Key/value Store

The host state machine MUST provide a key/value store interface with three functions that behave in the standard way:

```
type get = (path: Path) => Value | void
```

```
type set = (path: Path, value: Value) => void
```

```
type delete = (path: Path) => void
```

`Path` is as defined above. `Value` is an arbitrary bytestring encoding of a particular data structure. Encoding details are left to separate ICSs.

These functions MUST be permissioned to the IBC handler module (the implementation of which is described in separate standards) only, so only the IBC handler module can `set` or `delete` the paths that can be read by `get`. This can possibly be implemented as a sub-store (prefixed key-space) of a larger key/value store used by the entire state machine.

Host state machines MUST provide two instances of this interface - a `provableStore` for storage read by (i.e. proven to) other chains, and a `privateStore` for storage local to the host, upon which `get` , `set`, and `delete` can be called, e.g. `provableStore.set('some/path', 'value')`.

The `provableStore`:

- MUST write to a key/value store whose data can be externally proved with a vector commitment as defined in ICS 23.
- MUST use canonical data structure encodings provided in these specifications as proto3 files

The `privateStore`:

- MAY support external proofs, but is not required to - the IBC handler will never write data to it which needs to be proved.
- MAY use canonical proto3 data structures, but is not required to - it can use whatever format is preferred by the application environment.

> Note: any key/value store interface which provides these methods & properties is sufficient for IBC. Host state machines may implement "proxy stores" with path & value mappings which do not directly match the path & value pairs set and retrieved through the store interface — paths could be grouped into buckets & values stored in pages which could be proved in a single commitment, path-spaces could be remapped non-contiguously in some bijective manner, etc — as long as `get`, `set`, and `delete` behave as expected and other machines can verify commitment proofs of path & value pairs (or their absence) in the provable store. If applicable, the store must expose this mapping externally so that clients (including relayers) can determine the store layout & how to construct proofs. Clients of a machine using such a proxy store must also understand the mapping, so it will require either a new client type or a parameterised client.

> Note: this interface does not necessitate any particular storage backend or backend data layout. State machines may elect to use a storage backend configured in accordance with their needs, as long as the store on top fulfils the specified interface and provides commitment proofs.

4.2.4 Path-space

At present, IBC/TAO recommends the following path prefixes for the `provableStore` and `privateStore`.

Future paths may be used in future versions of the protocol, so the entire key-space in the provable store MUST be reserved for the IBC handler.

Keys used in the provable store MAY safely vary on a per-client-type basis as long as there exists a bipartite mapping between the key formats defined herein and the ones actually used in the machine's implementation.

Parts of the private store MAY safely be used for other purposes as long as the IBC handler has exclusive access to the specific keys required. Keys used in the private store MAY safely vary as long as there exists a bipartite mapping between the key formats defined herein and the ones actually used in the private store implementation.

Store	Path format	Value type	Defined in
privateStore	"clients/{identifier}"	ClientState	ICS 2
provableStore	"clients/{identifier}/consensusState"	ConsensusState	ICS 2
provableStore	"clients/{identifier}/type"	ClientType	ICS 2
provableStore	"connections/{identifier}"	ConnectionEnd	ICS 3
privateStore	"ports/{identifier}"	CapabilityKey	ICS 5
provableStore	"ports/{identifier}/channels/{identifier}"	ChannelEnd	ICS 4
provableStore	"ports/{identifier}/channels/{identifier}/key"	CapabilityKey	ICS 4
provableStore	"ports/{identifier}/channels/{identifier}/nextSequenceRecv"	uint64	ICS 4
provableStore	"ports/{identifier}/channels/{identifier}/packets/{sequence}"	bytes	ICS 4
provableStore	"ports/{identifier}/channels/{identifier}/acknowledgements/{sequence}"	bytes	ICS 4
privateStore	"callbacks/{identifier}"	ModuleCallbacks	ICS 26

4.2.5 Module layout

Represented spatially, the layout of modules & their included specifications on a host state machine looks like so (Aardvark, Betazoid, and Cephalopod are arbitrary modules):

```
+-----------------------------------------------------------------------------------+
|                                                                                   |
| Host State Machine                                                                |
|                                                                                   |
| +-------------------+       +--------------------+      +-----------------------+ |
| | Module Aardvark   | <-->  | IBC Routing Module |      | IBC Handler Module    | |
| +-------------------+       |                    |      |                       | |
|                             | Implements ICS 26. |      | Implements ICS 2, 3,  | |
|                             |                    |      | 4, 5 internally.      | |
| +-------------------+       |                    |      |                       | |
| | Module Betazoid   | <-->  |                    | -->  | Exposes interface     | |
| +-------------------+       |                    |      | defined in ICS 25.    | |
|                             |                    |      |                       | |
| +-------------------+       |                    |      |                       | |
| | Module Cephalopod | <-->  |                    |      |                       | |
| +-------------------+       +--------------------+      +-----------------------+ |
|                                                                                   |
+-----------------------------------------------------------------------------------+
```

4.2.6 Consensus state introspection

Host state machines MUST provide the ability to introspect their current height, with `getCurrentHeight`:

```
type getCurrentHeight = () => uint64
```

Host state machines MUST define a unique `ConsensusState` type fulfilling the requirements of ICS 2, with a canonical binary serialisation.

Host state machines MUST provide the ability to introspect their own consensus state, with `getConsensusState`:

```
type getConsensusState = (height: uint64) => ConsensusState
```

`getConsensusState` MUST return the consensus state for at least some number `n` of contiguous recent heights, where `n` is constant for the host state machine. Heights older than `n` MAY be safely pruned (causing future calls to fail for those heights).

Host state machines MUST provide the ability to introspect this stored recent consensus state count `n`, with `getStoredRecentConsensusStateCount`:

```
type getStoredRecentConsensusStateCount = () => uint64
```

4.2.7 Commitment path introspection

Host chains MUST provide the ability to inspect their commitment path, with `getCommitmentPrefix`:

```
type getCommitmentPrefix = () => CommitmentPrefix
```

The result `CommitmentPrefix` is the prefix used by the host state machine's key-value store. With the `CommitmentRoot root` and `CommitmentState state` of the host state machine, the following property MUST be preserved:

```
if provableStore.get(path) === value {
  prefixedPath = applyPrefix(getCommitmentPrefix(), path)
  if value !== nil {
    proof = createMembershipProof(state, prefixedPath, value)
    assert(verifyMembership(root, proof, prefixedPath, value))
  } else {
    proof = createNonMembershipProof(state, prefixedPath)
    assert(verifyNonMembership(root, proof, prefixedPath))
  }
}
```

For a host state machine, the return value of `getCommitmentPrefix` MUST be constant.

4.2.8 Port system

Host state machines MUST implement a port system, where the IBC handler can allow different modules in the host state machine to bind to uniquely named ports. Ports are identified by an `Identifier`.

Host state machines MUST implement permission interaction with the IBC handler such that:

- Once a module has bound to a port, no other modules can use that port until the module releases it
- A single module can bind to multiple ports
- Ports are allocated first-come first-serve and "reserved" ports for known modules can be bound when the state machine is first started

This permissioning can be implemented with unique references (object capabilities) for each port (a la the Cosmos SDK), with source authentication (a la Ethereum), or with some other method of access control, in any case enforced by the host state machine. See ICS 5 for details.

Modules that wish to make use of particular IBC features MAY implement certain handler functions, e.g. to add additional logic to a channel handshake with an associated module on another state machine.

4.2.9 Datagram submission

Host state machines which implement the routing module MAY define a `submitDatagram` function to submit datagrams, which will be included in transactions, directly to the routing module (defined in ICS 26):

```
type submitDatagram = (datagram: Datagram) => void
```

`submitDatagram` allows relayer processes to submit IBC datagrams directly to the routing module on the host state machine. Host state machines MAY require that the relayer process submitting the datagram has an account to pay transaction fees, signs over the datagram in a larger transaction structure, etc — `submitDatagram` MUST define & construct any such packaging required.

4.2.10 Exception system

Host state machines MUST support an exception system, whereby a transaction can abort execution and revert any previously made state changes (including state changes in other modules happening within the same transaction), excluding gas consumed & fee payments as appropriate, and a system invariant violation can halt the state machine.

This exception system MUST be exposed through two functions: `abortTransactionUnless` and `abortSystemUnless`, where the former reverts the transaction and the latter halts the state machine.

```
type abortTransactionUnless = (bool) => void
```

If the boolean passed to `abortTransactionUnless` is `true`, the host state machine need not do anything. If the boolean passed to `abortTransactionUnless` is `false`, the host state machine MUST abort the transaction and revert any previously made state changes, excluding gas consumed & fee payments as appropriate.

```
type abortSystemUnless = (bool) => void
```

If the boolean passed to `abortSystemUnless` is `true`, the host state machine need not do anything. If the boolean passed to `abortSystemUnless` is `false`, the host state machine MUST halt.

4.2.11 Data availability

For deliver-or-timeout safety, host state machines MUST have eventual data availability, such that any key/value pairs in state can be eventually retrieved by relayers. For exactly-once safety, data availability is not required.

For liveness of packet relay, host state machines MUST have bounded transactional liveness (and thus necessarily consensus liveness), such that incoming transactions are confirmed within a block height bound (in particular, less than the timeouts assign to the packets).

IBC packet data, and other data which is not directly stored in the state vector but is relied upon by relayers, MUST be available to & efficiently computable by relayer processes.

Light clients of particular consensus algorithms may have different and/or more strict data availability requirements.

4.2.12 Event logging system

The host state machine MUST provide an event logging system whereby arbitrary data can be logged in the course of transaction execution which can be stored, indexed, and later queried by processes executing the state machine. These event logs are utilised by relayers to read IBC packet data & timeouts, which are not stored directly in the chain state (as this storage is presumed to be expensive) but are instead committed to with a succinct cryptographic commitment (only the commitment is stored).

This system is expected to have at minimum one function for emitting log entries and one function for querying past logs, approximately as follows.

The function `emitLogEntry` can be called by the state machine during transaction execution to write a log entry:

```
type emitLogEntry = (topic: string, data: []byte) => void
```

The function `queryByTopic` can be called by an external process (such as a relayer) to retrieve all log entries associated with a given topic written by transactions which were executed at a given height.

```
type queryByTopic = (height: uint64, topic: string) => Array< []byte >
```

More complex query functionality MAY also be supported, and may allow for more efficient relayer process queries, but is not required.

5 ICS 002 - Client Semantics

5.1 Synopsis

This standard specifies the properties that consensus algorithms of machines implementing the interblockchain communication protocol are required to satisfy. These properties are necessary for efficient and safe verification in the higher-level protocol abstractions. The algorithm utilised in IBC to verify the consensus transcript & state sub-components of another machine is referred to as a “validity predicate”, and pairing it with a state that the verifier assumes to be correct forms a “light client” (often shortened to “client”).

This standard also specifies how light clients will be stored, registered, and updated in the canonical IBC handler. The stored client instances will be introspectable by a third party actor, such as a user inspecting the state of the chain and deciding whether or not to send an IBC packet.

5.1.1 Motivation

In the IBC protocol, an actor, which may be an end user, an off-chain process, or a machine, needs to be able to verify updates to the state of another machine which the other machine’s consensus algorithm has agreed upon, and reject any possible updates which the other machine’s consensus algorithm has not agreed upon. A light client is the algorithm with which a machine can do so. This standard formalises the light client model and requirements, so that the IBC protocol can easily integrate with new machines which are running new consensus algorithms as long as associated light client algorithms fulfilling the listed requirements are provided.

Beyond the properties described in this specification, IBC does not impose any requirements on the internal operation of machines and their consensus algorithms. A machine may consist of a single process signing operations with a private key, a quorum of processes signing in unison, many processes operating a Byzantine fault-tolerant consensus algorithm, or other configurations yet to be invented — from the perspective of IBC, a machine is defined entirely by its light client validation & equivocation detection logic. Clients will generally not include validation of the state transition logic in general (as that would be equivalent to simply executing the other state machine), but may elect to validate parts of state transitions in particular cases.

Clients could also act as thresholding views of other clients. In the case where modules utilising the IBC protocol to interact with probabilistic-finality consensus algorithms which might require different finality thresholds for different applications, one write-only client could be created to track headers and many read-only clients with different finality thresholds (confirmation depths after which state roots are considered final) could use that same state.

The client protocol should also support third-party introduction. Alice, a module on a machine, wants to introduce Bob, a second module on a second machine who Alice knows (and who knows Alice), to Carol, a third module on a third machine, who Alice knows but Bob does not. Alice must utilise an existing channel to Bob to communicate the canonically-serialisable validity predicate for Carol, with which Bob can then open a connection and channel so that Bob and Carol can talk directly. If necessary, Alice may also communicate to Carol the validity predicate for Bob, prior to Bob's connection attempt, so that Carol knows to accept the incoming request.

Client interfaces should also be constructed so that custom validation logic can be provided safely to define a custom client at runtime, as long as the underlying state machine can provide an appropriate gas metering mechanism to charge for compute and storage. On a host state machine which supports WASM execution, for example, the validity predicate and equivocation predicate could be provided as executable WASM functions when the client instance is created.

5.1.2 Definitions

- `get`, `set`, `Path`, and `Identifier` are as defined in ICS 24.
- `CommitmentRoot` is as defined in ICS 23. It must provide an inexpensive way for downstream logic to verify whether key/value pairs are present in state at a particular height.
- `ConsensusState` is an opaque type representing the state of a validity predicate. `ConsensusState` must be able to verify state updates agreed upon by the associated consensus algorithm. It must also be serialisable in a canonical fashion so that third parties, such as counterparty machines, can check that a particular machine has stored a particular `ConsensusState`. It must finally be introspectable by the state machine which it is for, such that the state machine can look up its own `ConsensusState` at a past height.
- `ClientState` is an opaque type representing the state of a client. A `ClientState` must expose query functions to verify membership or non-membership of key/value pairs in state at particular heights and to retrieve the current `ConsensusState`.

5.1.3 Desired Properties

Light clients must provide a secure algorithm to verify other chains' canonical headers, using the existing `ConsensusState`. The higher level abstractions will then be able to verify sub-components of the state with the `CommitmentRoot`s stored in the `ConsensusState`, which are guaranteed to have been committed by the other chain's consensus algorithm.

Validity predicates are expected to reflect the behaviour of the full nodes which are running the corresponding consensus algorithm. Given a `ConsensusState` and a list of messages, if a full node accepts the new `Header` generated with `Commit`, then the light client MUST also accept it, and if a full node rejects it, then the light client MUST also reject it.

Light clients are not replaying the whole message transcript, so it is possible under cases of consensus misbehaviour that the light clients' behaviour differs from the full nodes'. In this case, a misbehaviour proof which proves the divergence between the validity predicate and the full node can be generated and submitted to the chain so that the chain can safely deactivate the light client, invalidate past state roots, and await higher-level intervention.

5.2 Technical Specification

This specification outlines what each *client type* must define. A client type is a set of definitions of the data structures, initialisation logic, validity predicate, and misbehaviour predicate required to operate a light client. State machines implementing the IBC protocol can support any number of client types, and each client type can be instantiated with different initial consensus states in order to track different consensus instances. In order to establish a connection between two machines (see ICS 3), the machines must each support the client type corresponding to the other machine's consensus algorithm.

Specific client types shall be defined in later versions of this specification and a canonical list shall exist in this repository. Machines implementing the IBC protocol are expected to respect these client types, although they may elect to support only a subset.

5.2.1 Data Structures

ConsensusState `ConsensusState` is an opaque data structure defined by a client type, used by the validity predicate to verify new commits & state roots. Likely the structure will contain the last commit produced by the consensus process, including signatures and validator set metadata.

`ConsensusState` MUST be generated from an instance of `Consensus`, which assigns unique heights for each `ConsensusState` (such that each height has exactly one associated consensus state). Two `ConsensusState`s on the same chain SHOULD NOT have the same height if they do not have equal commitment roots. Such an event is called an “equivocation” and MUST be classified as misbehaviour. Should one occur, a proof should be generated and submitted so that the client can be frozen and previous state roots invalidated as necessary.

The `ConsensusState` of a chain MUST have a canonical serialisation, so that other chains can check that a stored consensus state is equal to another (see ICS 24 for the keyspace table).

```
type ConsensusState = bytes
```

The `ConsensusState` MUST be stored under a particular key, defined below, so that other chains can verify that a particular consensus state has been stored.

Header A `Header` is an opaque data structure defined by a client type which provides information to update a `ConsensusState`. Headers can be submitted to an associated client to update the stored `ConsensusState`. They likely contain a height, a proof, a commitment root, and possibly updates to the validity predicate.

```
type Header = bytes
```

Consensus `Consensus` is a `Header` generating function which takes the previous `ConsensusState` with the messages and returns the result.

```
type Consensus = (ConsensusState, [Message]) => Header
```

5.2.2 Blockchain

A blockchain is a consensus algorithm which generates valid `Header`s. It generates a unique list of headers starting from a genesis `ConsensusState` with arbitrary messages.

`Blockchain` is defined as

```
interface Blockchain {
  genesis: ConsensusState
  consensus: Consensus
}
```

where * `Genesis` is the genesis `ConsensusState` * `Consensus` is the header generating function

The headers generated from a `Blockchain` are expected to satisfy the following:

1. Each `Header` MUST NOT have more than one direct child

- Satisfied if: finality & safety
- Possible violation scenario: validator double signing, chain reorganisation (Nakamoto consensus)

2. Each `Header` MUST eventually have at least one direct child

- Satisfied if: liveness, light-client verifier continuity
- Possible violation scenario: synchronised halt, incompatible hard fork

3. Each `Header`s MUST be generated by `Consensus`, which ensures valid state transitions
   - Satisfied if: correct block generation & state machine
   - Possible violation scenario: invariant break, super-majority validator cartel

Unless the blockchain satisfies all of the above the IBC protocol may not work as intended: the chain can receive multiple conflicting packets, the chain cannot recover from the timeout event, the chain can steal the user's asset, etc.

The validity of the validity predicate is dependent on the security model of the `Consensus`. For example, the `Consensus` can be a proof of authority with a trusted operator, or a proof of stake but with insufficient value of stake. In such cases, it is possible that the security assumptions break, the correspondence between `Consensus` and the validity predicate no longer exists, and the behaviour of the validity predicate becomes undefined. Also, the `Blockchain` may not longer satisfy the requirements above, which will cause the chain to be incompatible with the IBC protocol. In cases of attributable faults, a misbehaviour proof can be generated and submitted to the chain storing the client to safely freeze the light client and prevent further IBC packet relay.

Validity predicate A validity predicate is an opaque function defined by a client type to verify `Header`s depending on the current `ConsensusState`. Using the validity predicate SHOULD be far more computationally efficient than replaying the full consensus algorithm for the given parent `Header` and the list of network messages.

The validity predicate & client state update logic are combined into a single `checkValidityAndUpdateState` type, which is defined as

```
type checkValidityAndUpdateState = (Header) => Void
```

`checkValidityAndUpdateState` MUST throw an exception if the provided header was not valid.

If the provided header was valid, the client MUST also mutate internal state to store now-finalised consensus roots and update any necessary signature authority tracking (e.g. changes to the validator set) for future calls to the validity predicate.

Misbehaviour predicate A misbehaviour predicate is an opaque function defined by a client type, used to check if data constitutes a violation of the consensus protocol. This might be two signed headers with different state roots but the same height, a signed header containing invalid state transitions, or other evidence of malfeasance as defined by the consensus algorithm.

The misbehaviour predicate & client state update logic are combined into a single `checkMisbehaviourAndUpdateState` type, which is defined as

```
type checkMisbehaviourAndUpdateState = (bytes) => Void
```

`checkMisbehaviourAndUpdateState` MUST throw an exception if the provided evidence was not valid.

If misbehaviour was valid, the client MUST also mutate internal state to mark appropriate heights which were previously considered valid as invalid, according to the nature of the misbehaviour.

ClientState `ClientState` is an opaque data structure defined by a client type. It may keep arbitrary internal state to track verified roots and past misbehaviours.

Light clients are representation-opaque — different consensus algorithms can define different light client update algorithms — but they must expose this common set of query functions to the IBC handler.

```
type ClientState = bytes
```

Client types must also define a method to initialize a client state with a provided consensus state:

```
type initialize = (state: ConsensusState) => ClientState
```

CommitmentProof `CommitmentProof` is an opaque data structure defined by a client type in accordance with ICS 23. It is utilised to verify presence or absence of a particular key/value pair in state at a particular finalised height (necessarily associated with a particular commitment root).

State verification Client types must define functions to authenticate internal state of the state machine which the client tracks. Internal implementation details may differ (for example, a loopback client could simply read directly from the state and require no proofs).

Required functions `verifyClientConsensusState` verifies a proof of the consensus state of the specified client stored on the target machine.

```
type verifyClientConsensusState = (
  clientState: ClientState,
  height: uint64,
  proof: CommitmentProof,
  clientIdentifier: Identifier,
  consensusState: ConsensusState)
  => boolean
```

`verifyConnectionState` verifies a proof of the connection state of the specified connection end stored on the target machine.

```
type verifyConnectionState = (
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  connectionIdentifier: Identifier,
  connectionEnd: ConnectionEnd)
  => boolean
```

`verifyChannelState` verifies a proof of the channel state of the specified channel end, under the specified port, stored on the target machine.

```
type verifyChannelState = (
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  channelEnd: ChannelEnd)
  => boolean
```

`verifyPacketCommitment` verifies a proof of an outgoing packet commitment at the specified port, specified channel, and specified sequence.

```
type verifyPacketCommitment = (
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64,
  commitment: bytes)
  => boolean
```

`verifyPacketAcknowledgement` verifies a proof of an incoming packet acknowledgement at the specified port, specified channel, and specified sequence.

```
type verifyPacketAcknowledgement = (
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64,
  acknowledgement: bytes)
  => boolean
```

`verifyPacketAcknowledgementAbsence` verifies a proof of the absence of an incoming packet acknowledgement at the specified port, specified channel, and specified sequence.

```
type verifyPacketAcknowledgementAbsence = (
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64)
  => boolean
```

`verifyNextSequenceRecv` verifies a proof of the next sequence number to be received of the specified channel at the specified port.

```
type verifyNextSequenceRecv = (
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  nextSequenceRecv: uint64)
  => boolean
```

Implementation strategies Loopback

A loopback client of a local machine merely reads from the local state, to which it must have access.

Simple signatures

A client of a solo machine with a known public key checks signatures on messages sent by that local machine, which are provided as the `Proof` parameter. The `height` parameter can be used as a replay protection nonce.

Multi-signature or threshold signature schemes can also be used in such a fashion.

Proxy clients

Proxy clients verify another (proxy) machine's verification of the target machine, by including in the proof first a proof of the client state on the proxy machine, and then a secondary proof of the sub-state of the target machine with respect to the client state on the proxy machine. This allows the proxy client to avoid storing and tracking the consensus state of the target machine itself, at the cost of adding security assumptions of proxy machine correctness.

Merklized state trees

For clients of state machines with Merklized state trees, these functions can be implemented by calling `verifyMembership` or `verifyNonMembership`, using a verified Merkle root stored in the `ClientState`, to verify presence or absence of particular key/value pairs in state at particular heights in accordance with ICS 23.

```
type verifyMembership = (ClientState, uint64, CommitmentProof, Path, Value) => boolean
```

```
type verifyNonMembership = (ClientState, uint64, CommitmentProof, Path) => boolean
```

5.2.3 Sub-protocols

IBC handlers MUST implement the functions defined below.

Identifier validation Clients are stored under a unique `Identifier` prefix. This ICS does not require that client identifiers be generated in a particular manner, only that they be unique. However, it is possible to restrict the space of `Identifier`s if required. The validation function `validateClientIdentifier` MAY be provided.

```
type validateClientIdentifier = (id: Identifier) => boolean
```

If not provided, the default `validateClientIdentifier` will always return `true`.

Path-space `clientStatePath` takes an `Identifier` and returns a `Path` under which to store a particular client state.

```
function clientStatePath(id: Identifier): Path {
    return "clients/{id}/state"
}
```

`clientTypePath` takes an `Identifier` and returns `Path` under which to store the type of a particular client.

```
function clientTypePath(id: Identifier): Path {
    return "clients/{id}/type"
}
```

Consensus states MUST be stored separately so that they can be independently verified.

`consensusStatePath` takes an `Identifier` and returns a `Path` under which to store the consensus state of a client.

```
function consensusStatePath(id: Identifier): Path {
    return "clients/{id}/consensusState"
}
```

Utilising past roots To avoid race conditions between client updates (which change the state root) and proof-carrying transactions in handshakes or packet receipt, many IBC handler functions allow the caller to specify a particular past root to reference, which is looked up by height. IBC handler functions which do this must ensure that they also perform any requisite checks on the height passed in by the caller to ensure logical correctness.

Create Calling `createClient` with the specified identifier & initial consensus state creates a new client.

```
function createClient(
  id: Identifier,
  clientType: ClientType,
  consensusState: ConsensusState) {
    abortTransactionUnless(validateClientIdentifier(id))
    abortTransactionUnless(privateStore.get(clientStatePath(id)) === null)
    abortSystemUnless(provableStore.get(clientTypePath(id)) === null)
    clientState = clientType.initialize(consensusState)
    privateStore.set(clientStatePath(id), clientState)
    provableStore.set(clientTypePath(id), clientType)
}
```

Query Client consensus state and client internal state can be queried by identifier. The returned client state must fulfil an interface allowing membership / non-membership verification.

```
function queryClientConsensusState(id: Identifier): ConsensusState {
    return provableStore.get(consensusStatePath(id))
}
```

```
function queryClient(id: Identifier): ClientState {
    return privateStore.get(clientStatePath(id))
}
```

Update Updating a client is done by submitting a new `Header`. The `Identifier` is used to point to the stored `ClientState` that the logic will update. When a new `Header` is verified with the stored `ClientState`'s validity predicate and `ConsensusState`, the client MUST update its internal state accordingly, possibly finalising commitment roots and updating the signature authority logic in the stored consensus state.

```
function updateClient(
  id: Identifier,
  header: Header) {
    clientType = provableStore.get(clientTypePath(id))
    abortTransactionUnless(clientType !== null)
    clientState = privateStore.get(clientStatePath(id))
    abortTransactionUnless(clientState !== null)
    clientType.checkValidityAndUpdateState(clientState, header)
}
```

Misbehaviour If the client detects evidence of misbehaviour, the client can be alerted, possibly invalidating previously valid state roots & preventing future updates.

```
function submitMisbehaviourToClient(
  id: Identifier,
  evidence: bytes) {
    clientType = provableStore.get(clientTypePath(id))
    abortTransactionUnless(clientType !== null)
    clientState = privateStore.get(clientStatePath(id))
    abortTransactionUnless(clientState !== null)
    clientType.checkMisbehaviourAndUpdateState(clientState, evidence)
}
```

5.2.4 Example Implementation

An example validity predicate is constructed for a chain running a single-operator consensus algorithm, where the valid blocks are signed by the operator. The operator signing Key can be changed while the chain is running.

The client-specific types are then defined as follows:

- `ConsensusState` stores the latest height and latest public key
- `Header`s contain a height, a new commitment root, a signature by the operator, and possibly a new public key
- `checkValidityAndUpdateState` checks that the submitted height is monotonically increasing and that the signature is correct, then mutates the internal state
- `checkMisbehaviourAndUpdateState` checks for two headers with the same height & different commitment roots, then mutates the internal state

```
interface ClientState {
  frozen: boolean
  pastPublicKeys: Set<PublicKey>
  verifiedRoots: Map<uint64, CommitmentRoot>
}

interface ConsensusState {
  sequence: uint64
  publicKey: PublicKey
}

interface Header {
  sequence: uint64
  commitmentRoot: CommitmentRoot
  signature: Signature
  newPublicKey: Maybe<PublicKey>
}

interface Evidence {
  h1: Header
  h2: Header
}

// algorithm run by operator to commit a new block
function commit(
  commitmentRoot: CommitmentRoot,
  sequence: uint64,
  newPublicKey: Maybe<PublicKey>): Header {
    signature = privateKey.sign(commitmentRoot, sequence, newPublicKey)
    header = {sequence, commitmentRoot, signature, newPublicKey}
    return header
}

// initialisation function defined by the client type
function initialize(consensusState: ConsensusState): ClientState {
  return {
    frozen: false,
    pastPublicKeys: Set.singleton(consensusState.publicKey),
    verifiedRoots: Map.empty()
  }
}

// validity predicate function defined by the client type
function checkValidityAndUpdateState(
  clientState: ClientState,
  header: Header) {
    abortTransactionUnless(consensusState.sequence + 1 === header.sequence)
    abortTransactionUnless(consensusState.publicKey.verify(header.signature))
    if (header.newPublicKey !== null) {
      consensusState.publicKey = header.newPublicKey
      clientState.pastPublicKeys.add(header.newPublicKey)
    }
    consensusState.sequence = header.sequence
    clientState.verifiedRoots[sequence] = header.commitmentRoot
}

function verifyClientConsensusState(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  clientIdentifier: Identifier,
  consensusState: ConsensusState) {
    path = applyPrefix(prefix, "clients/{clientIdentifier}/consensusState")
    abortTransactionUnless(!clientState.frozen)
    return clientState.verifiedRoots[sequence].verifyMembership(path, consensusState, proof)
}

function verifyConnectionState(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  connectionIdentifier: Identifier,
  connectionEnd: ConnectionEnd) {
    path = applyPrefix(prefix, "connection/{connectionIdentifier}")
    abortTransactionUnless(!clientState.frozen)
    return clientState.verifiedRoots[sequence].verifyMembership(path, connectionEnd, proof)
}

function verifyChannelState(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  channelEnd: ChannelEnd) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}")
    abortTransactionUnless(!clientState.frozen)
    return clientState.verifiedRoots[sequence].verifyMembership(path, channelEnd, proof)
}

function verifyPacketCommitment(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64,
  commitment: bytes) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}/packets/{sequence}"
         )
    abortTransactionUnless(!clientState.frozen)
    return clientState.verifiedRoots[sequence].verifyMembership(path, commitment, proof)
}

function verifyPacketAcknowledgement(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64,
  acknowledgement: bytes) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}/acknowledgements/{
         sequence}")
    abortTransactionUnless(!clientState.frozen)
    return clientState.verifiedRoots[sequence].verifyMembership(path, acknowledgement, proof)
}

function verifyPacketAcknowledgementAbsence(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}/acknowledgements/{
        sequence}")
    abortTransactionUnless(!clientState.frozen)
    return clientState.verifiedRoots[sequence].verifyNonMembership(path, proof)
}

function verifyNextSequenceRecv(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  nextSequenceRecv: uint64) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}/nextSequenceRecv")
    abortTransactionUnless(!clientState.frozen)
    return clientState.verifiedRoots[sequence].verifyMembership(path, nextSequenceRecv, proof)
}

// misbehaviour verification function defined by the client type
// any duplicate signature by a past or current key freezes the client
function checkMisbehaviourAndUpdateState(
  clientState: ClientState,
  evidence: Evidence) {
    h1 = evidence.h1
    h2 = evidence.h2
    abortTransactionUnless(clientState.pastPublicKeys.contains(h1.publicKey))
    abortTransactionUnless(h1.sequence === h2.sequence)
    abortTransactionUnless(h1.commitmentRoot !== h2.commitmentRoot || h1.publicKey !== h2.publicKey)
    abortTransactionUnless(h1.publicKey.verify(h1.signature))
    abortTransactionUnless(h2.publicKey.verify(h2.signature))
    clientState.frozen = true
}
```

5.2.5 Properties & Invariants

- Client identifiers are immutable & first-come-first-serve. Clients cannot be deleted (allowing deletion would potentially allow future replay of past packets if identifiers were re-used).

6 ICS 003 - Connection Semantics

6.1 Synopsis

This standards document describes the abstraction of an IBC *connection*: two stateful objects (*connection ends*) on two separate chains, each associated with a light client of the other chain, which together facilitate cross-chain sub-state verification and packet association (through channels). A protocol for safely establishing a connection between two chains is described.

6.1.1 Motivation

The core IBC protocol provides *authorisation* and *ordering* semantics for packets: guarantees, respectively, that packets have been committed on the sending blockchain (and according state transitions executed, such as escrowing tokens), and that they have been committed exactly once in a particular order and can be delivered exactly once in that same order. The *connection* abstraction specified in this standard, in conjunction with the *client* abstraction specified in ICS 2, defines the *authorisation* semantics of IBC. Ordering semantics are described in ICS 4).

6.1.2 Definitions

Client-related types & functions are as defined in ICS 2.

Commitment proof related types & functions are defined in ICS 23

`Identifier` and other host state machine requirements are as defined in ICS 24. The identifier is not necessarily intended to be a human-readable name (and likely should not be, to discourage squatting or racing for identifiers).

The opening handshake protocol allows each chain to verify the identifier used to reference the connection on the other chain, enabling modules on each chain to reason about the reference on the other chain.

An *actor*, as referred to in this specification, is an entity capable of executing datagrams who is paying for computation / storage (via gas or a similar mechanism) but is otherwise untrusted. Possible actors include:

- End users signing with an account key
- On-chain smart contracts acting autonomously or in response to another transaction
- On-chain modules acting in response to another transaction or in a scheduled manner

6.1.3 Desired Properties

- Implementing blockchains should be able to safely allow untrusted actors to open and update connections.

Pre-Establishment Prior to connection establishment:

- No further IBC sub-protocols should operate, since cross-chain sub-states cannot be verified.
- The initiating actor (who creates the connection) must be able to specify an initial consensus state for the chain to connect to and an initial consensus state for the connecting chain (implicitly, e.g. by sending the transaction).

During Handshake Once a negotiation handshake has begun:

- Only the appropriate handshake datagrams can be executed in order.
- No third chain can masquerade as one of the two handshaking chains

Post-Establishment Once a negotiation handshake has completed:

- The created connection objects on both chains contain the consensus states specified by the initiating actor.
- No other connection objects can be maliciously created on other chains by replaying datagrams.

6.2 Technical Specification

6.2.1 Data Structures

This ICS defines the `ConnectionState` and `ConnectionEnd` types:

```
enum ConnectionState {
  INIT,
  TRYOPEN,
  OPEN,
}
```

```
interface ConnectionEnd {
  state: ConnectionState
  counterpartyConnectionIdentifier: Identifier
  counterpartyPrefix: CommitmentPrefix
  clientIdentifier: Identifier
  counterpartyClientIdentifier: Identifier
  version: string | []string
}
```

- The `state` field describes the current state of the connection end.
- The `counterpartyConnectionIdentifier` field identifies the connection end on the counterparty chain associated with this connection.

- The `clientIdentifier` field identifies the client associated with this connection.
- The `counterpartyClientIdentifier` field identifies the client on the counterparty chain associated with this connection.
- The `version` field is an opaque string which can be utilised to determine encodings or protocols for channels or packets utilising this connection.

6.2.2 Store paths

Connection paths are stored under a unique identifier.

```
function connectionPath(id: Identifier): Path {
    return "connections/{id}"
}
```

A reverse mapping from clients to a set of connections (utilised to look up all connections using a client) is stored under a unique prefix per-client:

```
function clientConnectionsPath(clientIdentifier: Identifier): Path {
    return "clients/{clientIdentifier}/connections"
}
```

6.2.3 Helper functions

`addConnectionToClient` is used to add a connection identifier to the set of connections associated with a client.

```
function addConnectionToClient(
  clientIdentifier: Identifier,
  connectionIdentifier: Identifier) {
    conns = privateStore.get(clientConnectionsPath(clientIdentifier))
    conns.add(connectionIdentifier)
    privateStore.set(clientConnectionsPath(clientIdentifier), conns)
}
```

`removeConnectionFromClient` is used to remove a connection identifier from the set of connections associated with a client.

```
function removeConnectionFromClient(
  clientIdentifier: Identifier,
  connectionIdentifier: Identifier) {
    conns = privateStore.get(clientConnectionsPath(clientIdentifier))
    conns.remove(connectionIdentifier)
    privateStore.set(clientConnectionsPath(clientIdentifier), conns)
}
```

Helper functions are defined by the connection to pass the `CommitmentPrefix` associated with the connection to the verification function provided by the client. In the other parts of the specifications, these functions MUST be used for introspecting other chains' state, instead of directly calling the verification functions on the client.

```
function verifyClientConsensusState(
  connection: ConnectionEnd,
  height: uint64,
  proof: CommitmentProof,
  clientIdentifier: Identifier,
  consensusState: ConsensusState) {
    client = queryClient(connection.clientIdentifier)
    return client.verifyClientConsensusState(connection, height, connection.counterpartyPrefix, proof,
         clientIdentifier, consensusState)
}

function verifyConnectionState(
  connection: ConnectionEnd,
  height: uint64,
  proof: CommitmentProof,
  connectionIdentifier: Identifier,
  connectionEnd: ConnectionEnd) {
    client = queryClient(connection.clientIdentifier)
    return client.verifyConnectionState(connection, height, connection.counterpartyPrefix, proof,
         connectionIdentifier, connectionEnd)
}

function verifyChannelState(
  connection: ConnectionEnd,
  height: uint64,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  channelEnd: ChannelEnd) {
    client = queryClient(connection.clientIdentifier)
    return client.verifyChannelState(connection, height, connection.counterpartyPrefix, proof,
         portIdentifier, channelIdentifier, channelEnd)
}

function verifyPacketCommitment(
  connection: ConnectionEnd,
  height: uint64,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64,
  commitment: bytes) {
    client = queryClient(connection.clientIdentifier)
    return client.verifyPacketCommitment(connection, height, connection.counterpartyPrefix, proof,
         portIdentifier, channelIdentifier, commitment)
}

function verifyPacketAcknowledgement(
  connection: ConnectionEnd,
  height: uint64,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64,
  acknowledgement: bytes) {
    client = queryClient(connection.clientIdentifier)
    return client.verifyPacketAcknowledgement(connection, height, connection.counterpartyPrefix, proof,
          portIdentifier, channelIdentifier, acknowledgement)
}

function verifyPacketAcknowledgementAbsence(
  connection: ConnectionEnd,
  height: uint64,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64) {
    client = queryClient(connection.clientIdentifier)
    return client.verifyPacketAcknowledgementAbsence(connection, height, connection.counterpartyPrefix,
          proof, portIdentifier, channelIdentifier)
}

function verifyNextSequenceRecv(
  connection: ConnectionEnd,
  height: uint64,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  nextSequenceRecv: uint64) {
    client = queryClient(connection.clientIdentifier)
    return client.verifyNextSequenceRecv(connection, height, connection.counterpartyPrefix, proof,
         portIdentifier, channelIdentifier, nextSequenceRecv)
}
```

6.2.4 Sub-protocols

This ICS defines the opening handshake subprotocol. Once opened, connections cannot be closed and identifiers cannot be reallocated (this prevents packet replay or authorisation confusion).

Header tracking and misbehaviour detection are defined in ICS 2.

Figure 1: State Machine Diagram

Identifier validation Connections are stored under a unique `Identifier` prefix. The validation function `validateConnectionIdentifier` MAY be provided.

```
type validateConnectionIdentifier = (id: Identifier) => boolean
```

If not provided, the default `validateConnectionIdentifier` function will always return `true`.

Versioning During the handshake process, two ends of a connection come to agreement on a version bytestring associated with that connection. At the moment, the contents of this version bytestring are opaque to the IBC core protocol. In the future, it might be used to indicate what kinds of channels can utilise the connection in question, or what encoding formats channel-related datagrams will use. At present, host state machine MAY utilise the version data to negotiate encodings, priorities, or connection-specific metadata related to custom logic on top of IBC.

Host state machines MAY also safely ignore the version data or specify an empty string.

An implementation MUST define a function `getCompatibleVersions` which returns the list of versions it supports, ranked by descending preference order.

```
type getCompatibleVersions = () => []string
```

An implementation MUST define a function `pickVersion` to choose a version from a list of versions proposed by a counterparty.

```
type pickVersion = ([]string) => string
```

Opening Handshake The opening handshake sub-protocol serves to initialise consensus states for two chains on each other.

The opening handshake defines four datagrams: *ConnOpenInit*, *ConnOpenTry*, *ConnOpenAck*, and *ConnOpenConfirm*.

A correct protocol execution flows as follows (note that all calls are made through modules per ICS 25):

Initiator	Datagram	Chain acted upon	Prior state (A, B)	Posterior state (A, B)
Actor	`ConnOpenInit`	A	(none, none)	(INIT, none)
Relayer	`ConnOpenTry`	B	(INIT, none)	(INIT, TRYOPEN)
Relayer	`ConnOpenAck`	A	(INIT, TRYOPEN)	(OPEN, TRYOPEN)
Relayer	`ConnOpenConfirm`	B	(OPEN, TRYOPEN)	(OPEN, OPEN)

At the end of an opening handshake between two chains implementing the sub-protocol, the following properties hold:

- Each chain has each other's correct consensus state as originally specified by the initiating actor.
- Each chain has knowledge of and has agreed to its identifier on the other chain.

This sub-protocol need not be permissioned, modulo anti-spam measures.

ConnOpenInit initialises a connection attempt on chain A.

```
function connOpenInit(
  identifier: Identifier,
  desiredCounterpartyConnectionIdentifier: Identifier,
  counterpartyPrefix: CommitmentPrefix,
  clientIdentifier: Identifier,
  counterpartyClientIdentifier: Identifier) {
    abortTransactionUnless(validateConnectionIdentifier(identifier))
    abortTransactionUnless(provableStore.get(connectionPath(identifier)) == null)
    state = INIT
    connection = ConnectionEnd{state, desiredCounterpartyConnectionIdentifier, counterpartyPrefix,
      clientIdentifier, counterpartyClientIdentifier, getCompatibleVersions()}
    provableStore.set(connectionPath(identifier), connection)
    addConnectionToClient(clientIdentifier, identifier)
}
```

ConnOpenTry relays notice of a connection attempt on chain A to chain B (this code is executed on chain B).

```
function connOpenTry(
  desiredIdentifier: Identifier,
  counterpartyConnectionIdentifier: Identifier,
  counterpartyPrefix: CommitmentPrefix,
  counterpartyClientIdentifier: Identifier,
  clientIdentifier: Identifier,
  counterpartyVersions: string[],
  proofInit: CommitmentProof,
  proofConsensus: CommitmentProof,
  proofHeight: uint64,
  consensusHeight: uint64) {
    abortTransactionUnless(validateConnectionIdentifier(desiredIdentifier))
    abortTransactionUnless(consensusHeight <= getCurrentHeight())
    expectedConsensusState = getConsensusState(consensusHeight)
    expected = ConnectionEnd{INIT, desiredIdentifier, getCommitmentPrefix(),
         counterpartyClientIdentifier,
                             clientIdentifier, counterpartyVersions}
    version = pickVersion(counterpartyVersions)
    connection = ConnectionEnd{state, counterpartyConnectionIdentifier, counterpartyPrefix,
                               clientIdentifier, counterpartyClientIdentifier, version}
    abortTransactionUnless(connection.verifyConnectionState(proofHeight, proofInit,
         counterpartyConnectionIdentifier, expected))
    abortTransactionUnless(connection.verifyClientConsensusState(proofHeight, proofConsensus,
         counterpartyClientIdentifier, expectedConsensusState))
    previous = provableStore.get(connectionPath(desiredIdentifier))
    abortTransactionUnless(
      (previous === null) ||
      (previous.state === INIT &&
        previous.counterpartyConnectionIdentifier === counterpartyConnectionIdentifier &&
        previous.counterpartyPrefix === counterpartyPrefix &&
        previous.clientIdentifier === clientIdentifier &&
        previous.counterpartyClientIdentifier === counterpartyClientIdentifier &&
        previous.version === version))
    identifier = desiredIdentifier
    state = TRYOPEN
    provableStore.set(connectionPath(identifier), connection)
    addConnectionToClient(clientIdentifier, identifier)
}
```

ConnOpenAck relays acceptance of a connection open attempt from chain B back to chain A (this code is executed on chain A).

```
function connOpenAck(
  identifier: Identifier,
  version: string,
  proofTry: CommitmentProof,
  proofConsensus: CommitmentProof,
  proofHeight: uint64,
  consensusHeight: uint64) {
    abortTransactionUnless(consensusHeight <= getCurrentHeight())
    connection = provableStore.get(connectionPath(identifier))
    abortTransactionUnless(connection.state === INIT || connection.state === TRYOPEN)
    expectedConsensusState = getConsensusState(consensusHeight)
    expected = ConnectionEnd{TRYOPEN, identifier, getCommitmentPrefix(),
                             connection.counterpartyClientIdentifier, connection.clientIdentifier,
                             version}
    abortTransactionUnless(connection.verifyConnectionState(proofHeight, proofTry, connection.
         counterpartyConnectionIdentifier, expected))
    abortTransactionUnless(connection.verifyClientConsensusState(proofHeight, proofConsensus,
         connection.counterpartyClientIdentifier, expectedConsensusState))
    connection.state = OPEN
    abortTransactionUnless(getCompatibleVersions().indexOf(version) !== -1)
    connection.version = version
    provableStore.set(connectionPath(identifier), connection)
}
```

ConnOpenConfirm confirms opening of a connection on chain A to chain B, after which the connection is open on both chains (this code is executed on chain B).

```
function connOpenConfirm(
  identifier: Identifier,
  proofAck: CommitmentProof,
  proofHeight: uint64) {
    connection = provableStore.get(connectionPath(identifier))
    abortTransactionUnless(connection.state === TRYOPEN)
    expected = ConnectionEnd{OPEN, identifier, getCommitmentPrefix(), connection.
         counterpartyClientIdentifier,
                             connection.clientIdentifier, connection.version}
    abortTransactionUnless(connection.verifyConnectionState(proofHeight, proofAck, connection.
         counterpartyConnectionIdentifier, expected))
    connection.state = OPEN
    provableStore.set(connectionPath(identifier), connection)
}
```

Querying Connections can be queried by identifier with `queryConnection`.

```
function queryConnection(id: Identifier): ConnectionEnd | void {
    return provableStore.get(connectionPath(id))
}
```

Connections associated with a particular client can be queried by client identifier with `queryClientConnections`.

```
function queryClientConnections(id: Identifier): Set<Identifier> {
    return privateStore.get(clientConnectionsPath(id))
}
```

6.2.5 Properties & Invariants

- Connection identifiers are first-come-first-serve: once a connection has been negotiated, a unique identifier pair exists between two chains.
- The connection handshake cannot be man-in-the-middled by another blockchain's IBC handler.

7 ICS 005 - Port Allocation

7.1 Synopsis

This standard specifies the port allocation system by which modules can bind to uniquely named ports allocated by the IBC handler. Ports can then be used to open channels and can be transferred or later released by the module which originally bound to them.

7.1.1 Motivation

The interblockchain communication protocol is designed to facilitate module-to-module traffic, where modules are independent, possibly mutually distrusted, self-contained elements of code executing on sovereign ledgers. In order to provide the desired end-to-end semantics, the IBC handler must permission channels to particular modules. This specification defines the *port allocation and ownership* system which realises that model.

Conventions may emerge as to what kind of module logic is bound to a particular port name, such as “bank” for fungible token handling or “staking” for interchain collateralisation. This is analogous to port 80’s common use for HTTP servers — the protocol cannot enforce that particular module logic is actually bound to conventional ports, so users must check that themselves. Ephemeral ports with pseudorandom identifiers may be created for temporary protocol handling.

Modules may bind to multiple ports and connect to multiple ports bound to by another module on a separate machine. Any number of (uniquely identified) channels can utilise a single port simultaneously. Channels are end-to-end between two ports, each of which must have been previously bound to by a module, which will then control that end of the channel.

Optionally, the host state machine can elect to expose port binding only to a specially-permissioned module manager, by generating a capability key specifically for the ability to bind ports. The module manager can then control which ports modules can bind to with a custom rule-set, and transfer ports to modules only when it has validated the port name & module. This role can be played by the routing module (see ICS 26).

7.1.2 Definitions

`Identifier`, `get`, `set`, and `delete` are defined as in ICS 24.

A *port* is a particular kind of identifier which is used to permission channel opening and usage to modules.

A *module* is a sub-component of the host state machine independent of the IBC handler. Examples include Ethereum smart contracts and Cosmos SDK & Substrate modules. The IBC specification makes no assumptions of module functionality other than the ability of the host state machine to use object-capability or source authentication to permission ports to modules.

7.1.3 Desired Properties

- Once a module has bound to a port, no other modules can use that port until the module releases it
- A module can, on its option, release a port or transfer it to another module
- A single module can bind to multiple ports at once
- Ports are allocated first-come first-serve and “reserved” ports for known modules can be bound when the chain is first started

As a helpful comparison, the following analogies to TCP are roughly accurate:

IBC Concept	TCP/IP Concept	Differences
IBC	TCP	Many, see the architecture documents describing IBC
Port (e.g. “bank”)	Port (e.g. 80)	No low-number reserved ports, ports are strings
Module (e.g. “bank”)	Application (e.g. Nginx)	Application-specific
Client	-	No direct analogy, a bit like L2 routing and a bit like TLS

IBC Concept	TCP/IP Concept	Differences
Connection	-	No direct analogy, folded into connections in TCP
Channel	Connection	Any number of channels can be opened to or from a port simultaneously

7.2 Technical Specification

7.2.1 Data Structures

The host state machine MUST support either object-capability reference or source authentication for modules.

In the former object-capability case, the IBC handler must have the ability to generate *object-capabilities*, unique, opaque references which can be passed to a module and will not be duplicable by other modules. Two examples are store keys as used in the Cosmos SDK (reference) and object references as used in Agoric's Javascript runtime (reference).

```
type CapabilityKey object
```

```
function newCapabilityPath(): CapabilityKey {
  // provided by host state machine, e.g. pointer address in Cosmos SDK
}
```

In the latter source authentication case, the IBC handler must have the ability to securely read the *source identifier* of the calling module, a unique string for each module in the host state machine, which cannot be altered by the module or faked by another module. An example is smart contract addresses as used by Ethereum (reference).

```
type SourceIdentifier string
```

```
function callingModuleIdentifier(): SourceIdentifier {
  // provided by host state machine, e.g. contract address in Ethereum
}
```

`generate` and `authenticate` functions are then defined as follows.

In the former case, `generate` returns a new object-capability key, which must be returned by the outer-layer function, and `authenticate` requires that the outer-layer function take an extra argument `capability`, which is an object-capability key with uniqueness enforced by the host state machine. Outer-layer functions are any functions exposed by the IBC handler (ICS 25) or routing module (ICS 26) to modules.

```
function generate(): CapabilityKey {
    return newCapabilityPath()
}
```

```
function authenticate(key: CapabilityKey): boolean {
    return capability === key
}
```

In the latter case, `generate` returns the calling module's identifier and `authenticate` merely checks it.

```
function generate(): SourceIdentifier {
    return callingModuleIdentifier()
}
```

```
function authenticate(id: SourceIdentifier): boolean {
    return callingModuleIdentifier() === id
}
```

Store paths `portPath` takes an `Identifier` and returns the store path under which the object-capability reference or owner module identifier associated with a port should be stored.

```
function portPath(id: Identifier): Path {
    return "ports/{id}"
}
```

7.2.2 Sub-protocols

Identifier validation Owner module identifier for ports are stored under a unique `Identifier` prefix. The validation function `validatePortIdentifier` MAY be provided.

```
type validatePortIdentifier = (id: Identifier) => boolean
```

If not provided, the default `validatePortIdentifier` function will always return `true`.

Binding to a port The IBC handler MUST implement `bindPort`. `bindPort` binds to an unallocated port, failing if the port has already been allocated.

If the host state machine does not implement a special module manager to control port allocation, `bindPort` SHOULD be available to all modules. If it does, `bindPort` SHOULD only be callable by the module manager.

```
function bindPort(id: Identifier) {
    abortTransactionUnless(validatePortIdentifier(id))
    abortTransactionUnless(privateStore.get(portPath(id)) === null)
    key = generate()
    privateStore.set(portPath(id), key)
    return key
}
```

Transferring ownership of a port If the host state machine supports object-capabilities, no additional protocol is necessary, since the port reference is a bearer capability. If it does not, the IBC handler MAY implement the following `transferPort` function.

`transferPort` SHOULD be available to all modules.

```
function transferPort(id: Identifier) {
    abortTransactionUnless(authenticate(privateStore.get(portPath(id))))
    key = generate()
    privateStore.set(portPath(id), key)
}
```

Releasing a port The IBC handler MUST implement the `releasePort` function, which allows a module to release a port such that other modules may then bind to it.

`releasePort` SHOULD be available to all modules.

> Warning: releasing a port will allow other modules to bind to that port and possibly intercept incoming channel opening handshakes. Modules should release ports only when doing so is safe.

```
function releasePort(id: Identifier) {
    abortTransactionUnless(authenticate(privateStore.get(portPath(id))))
    privateStore.delete(portPath(id))
}
```

7.2.3 Properties & Invariants

- By default, port identifiers are first-come-first-serve: once a module has bound to a port, only that module can utilise the port until the module transfers or releases it. A module manager can implement custom logic which overrides this.

8 ICS 004 - Channel & Packet Semantics

8.1 Synopsis

The "channel" abstraction provides message delivery semantics to the interblockchain communication protocol, in three categories: ordering, exactly-once delivery, and module permissioning. A channel serves as a conduit for packets passing between a module on one chain and a module on another, ensuring that packets are executed only once, delivered in the order in which they were sent (if necessary), and delivered only to the corresponding module owning the other end of the channel on the destination chain. Each channel is associated with a particular connection, and a connection may have any number of associated channels, allowing the use of common identifiers and amortising the cost of header verification across all the channels utilising a connection & light client.

Channels are payload-agnostic. The modules which send and receive IBC packets decide how to construct packet data and how to act upon the incoming packet data, and must utilise their own application logic to determine which state transactions to apply according to what data the packet contains.

8.1.1 Motivation

The interblockchain communication protocol uses a cross-chain message passing model. IBC *packets* are relayed from one blockchain to the other by external relayer processes. Chain `A` and chain `B` confirm new blocks independently, and packets from one chain to the other may be delayed, censored, or re-ordered arbitrarily. Packets are visible to relayers and can be read from a blockchain by any relayer process and submitted to any other blockchain.

> The IBC protocol must provide ordering (for ordered channels) and exactly-once delivery guarantees to allow applications to reason about the combined state of connected modules on two chains. 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`.

In order to provide the desired ordering, exactly-once delivery, and module permissioning semantics to the application layer, the interblockchain communication protocol must implement an abstraction to enforce these semantics — channels are this abstraction.

8.1.2 Definitions

`ConsensusState` is as defined in ICS 2.

`Connection` is as defined in ICS 3.

`Port` and `authenticate` are as defined in ICS 5.

`hash` is a generic collision-resistant hash function, the specifics of which must be agreed on by the modules utilising the channel.

`Identifier`, `get`, `set`, `delete`, `getCurrentHeight`, and module-system related primitives are as defined in ICS 24.

A *channel* is a pipeline for exactly-once packet delivery between specific modules on separate blockchains, which has at least one end capable of sending packets and one end capable of receiving packets.

A *bidirectional* channel is a channel where packets can flow in both directions: from `A` to `B` and from `B` to `A`.

A *unidirectional* channel is a channel where packets can only flow in one direction: from `A` to `B` (or from `B` to `A`, the order of naming is arbitrary).

An *ordered* channel is a channel where packets are delivered exactly in the order which they were sent.

An *unordered* channel is a channel where packets can be delivered in any order, which may differ from the order in which they were sent.

```
enum ChannelOrder {
  ORDERED,
  UNORDERED,
}
```

Directionality and ordering are independent, so one can speak of a bidirectional unordered channel, a unidirectional ordered channel, etc.

All channels provide exactly-once packet delivery, meaning that a packet sent on one end of a channel is delivered no more and no less than once, eventually, to the other end.

This specification only concerns itself with *bidirectional* channels. *Unidirectional* channels can use almost exactly the same protocol and will be outlined in a future ICS.

An end of a channel is a data structure on one chain storing channel metadata:

```
interface ChannelEnd {
  state: ChannelState
  ordering: ChannelOrder
  counterpartyPortIdentifier: Identifier
  counterpartyChannelIdentifier: Identifier
  connectionHops: [Identifier]
  version: string
}
```

- The `state` is the current state of the channel end.
- The `ordering` field indicates whether the channel is ordered or unordered.
- The `counterpartyPortIdentifier` identifies the port on the counterparty chain which owns the other end of the channel.
- The `counterpartyChannelIdentifier` identifies the channel end on the counterparty chain.
- The `nextSequenceSend`, stored separately, tracks the sequence number for the next packet to be sent.
- The `nextSequenceRecv`, stored separately, tracks the sequence number for the next packet to be received.
- The `connectionHops` stores the list of connection identifiers, in order, along which packets sent on this channel will travel. At the moment this list must be of length 1. In the future multi-hop channels may be supported.
- The `version` string stores an opaque channel version, which is agreed upon during the handshake. This can determine module-level configuration such as which packet encoding is used for the channel. This version is not used by the core IBC protocol.

Channel ends have a *state*:

```
enum ChannelState {
  INIT,
  TRYOPEN,
  OPEN,
  CLOSED,
}
```

- A channel end in `INIT` state has just started the opening handshake.
- A channel end in `TRYOPEN` state has acknowledged the handshake step on the counterparty chain.
- A channel end in `OPEN` state has completed the handshake and is ready to send and receive packets.
- A channel end in `CLOSED` state has been closed and can no longer be used to send or receive packets.

A `Packet`, in the interblockchain communication protocol, is a particular interface defined as follows:

```
interface Packet {
  sequence: uint64
  timeoutHeight: uint64
  sourcePort: Identifier
  sourceChannel: Identifier
  destPort: Identifier
  destChannel: Identifier
  data: bytes
}
```

- The `sequence` number corresponds to the order of sends and receives, where a packet with an earlier sequence number must be sent and received before a packet with a later sequence number.
- The `timeoutHeight` indicates a consensus height on the destination chain after which the packet will no longer be processed, and will instead count as having timed-out.
- The `sourcePort` identifies the port on the sending chain.
- The `sourceChannel` identifies the channel end on the sending chain.
- The `destPort` identifies the port on the receiving chain.
- The `destChannel` identifies the channel end on the receiving chain.

- The `data` is an opaque value which can be defined by the application logic of the associated modules.

Note that a `Packet` is never directly serialised. Rather it is an intermediary structure used in certain function calls that may need to be created or processed by modules calling the IBC handler.

An `OpaquePacket` is a packet, but cloaked in an obscuring data type by the host state machine, such that a module cannot act upon it other than to pass it to the IBC handler. The IBC handler can cast a `Packet` to an `OpaquePacket` and vice versa.

```
type OpaquePacket = object
```

8.1.3 Desired Properties

Efficiency

- The speed of packet transmission and confirmation should be limited only by the speed of the underlying chains. Proofs should be batchable where possible.

Exactly-once delivery

- IBC packets sent on one end of a channel should be delivered exactly once to the other end.
- No network synchrony assumptions should be required for exactly-once safety. If one or both of the chains halt, packets may be delivered no more than once, and once the chains resume packets should be able to flow again.

Ordering

- On ordered channels, packets should be sent and received in the same order: if packet x is sent before packet y by a channel end on chain `A`, packet x must be received before packet y by the corresponding channel end on chain `B`.
- On unordered channels, packets may be sent and received in any order. Unordered packets, like ordered packets, have individual timeouts specified in terms of the destination chain's height.

Permissioning

- Channels should be permissioned to one module on each end, determined during the handshake and immutable afterwards (higher-level logic could tokenize channel ownership by tokenising ownership of the port). Only the module associated with a channel end should be able to send or receive on it.

8.2 Technical Specification

8.2.1 Dataflow visualisation

The architecture of clients, connections, channels and packets:

Figure 2: Dataflow Visualisation

8.2.2 Preliminaries

Store paths Channel structures are stored under a store path prefix unique to a combination of a port identifier and channel identifier:

```
function channelPath(portIdentifier: Identifier, channelIdentifier: Identifier): Path {
    return "ports/{portIdentifier}/channels/{channelIdentifier}"
}
```

The capability key associated with a channel is stored under the `channelCapabilityPath`:

```
function channelCapabilityPath(portIdentifier: Identifier, channelIdentifier: Identifier): Path {
  return "{channelPath(portIdentifier, channelIdentifier)}/key"
}
```

The `nextSequenceSend` and `nextSequenceRecv` unsigned integer counters are stored separately so they can be proved individually:

```
function nextSequenceSendPath(portIdentifier: Identifier, channelIdentifier: Identifier): Path {
    return "{channelPath(portIdentifier, channelIdentifier)}/nextSequenceSend"
}

function nextSequenceRecvPath(portIdentifier: Identifier, channelIdentifier: Identifier): Path {
    return "{channelPath(portIdentifier, channelIdentifier)}/nextSequenceRecv"
}
```

Constant-size commitments to packet data fields are stored under the packet sequence number:

```
function packetCommitmentPath(portIdentifier: Identifier, channelIdentifier: Identifier, sequence:
     uint64): Path {
    return "{channelPath(portIdentifier, channelIdentifier)}/packets/" + sequence
}
```

Absence of the path in the store is equivalent to a zero-bit.

Packet acknowledgement data are stored under the `packetAcknowledgementPath`:

```
function packetAcknowledgementPath(portIdentifier: Identifier, channelIdentifier: Identifier, sequence:
      uint64): Path {
    return "{channelPath(portIdentifier, channelIdentifier)}/acknowledgements/" + sequence
}
```

Unordered channels MUST always write a acknowledgement (even an empty one) to this path so that the absence of such can be used as proof-of-timeout. Ordered channels MAY write an acknowledgement, but are not required to.

8.2.3 Versioning

During the handshake process, two ends of a channel come to agreement on a version bytestring associated with that channel. The contents of this version bytestring are and will remain opaque to the IBC core protocol. Host state machines MAY utilise the version data to indicate supported IBC/APP protocols, agree on packet encoding formats, or negotiate other channel-related metadata related to custom logic on top of IBC.

Host state machines MAY also safely ignore the version data or specify an empty string.

8.2.4 Sub-protocols

> Note: If the host state machine is utilising object capability authentication (see ICS 005), all functions utilising ports take an additional capability parameter.

Identifier validation Channels are stored under a unique `(portIdentifier, channelIdentifier)` prefix. The validation function `validatePortIdentifier` MAY be provided.

```
type validateChannelIdentifier = (portIdentifier: Identifier, channelIdentifier: Identifier) => boolean
```

If not provided, the default `validateChannelIdentifier` function will always return `true`.

Figure 3: Channel State Machine

Channel lifecycle management

Initiator	Datagram	Chain acted upon	Prior state (A, B)	Posterior state (A, B)
Actor	ChanOpenInit	A	(none, none)	(INIT, none)
Relayer	ChanOpenTry	B	(INIT, none)	(INIT, TRYOPEN)
Relayer	ChanOpenAck	A	(INIT, TRYOPEN)	(OPEN, TRYOPEN)
Relayer	ChanOpenConfirm	B	(OPEN, TRYOPEN)	(OPEN, OPEN)

Initiator	Datagram	Chain acted upon	Prior state (A, B)	Posterior state (A, B)
Actor	ChanCloseInit	A	(OPEN, OPEN)	(CLOSED, OPEN)
Relayer	ChanCloseConfirm	B	(CLOSED, OPEN)	(CLOSED, CLOSED)

Opening handshake The `chanOpenInit` function is called by a module to initiate a channel opening handshake with a module on another chain.

The opening channel must provide the identifiers of the local channel identifier, local port, remote port, and remote channel identifier.

When the opening handshake is complete, the module which initiates the handshake will own the end of the created channel on the host ledger, and the counterparty module which it specifies will own the other end of the created channel on the counterparty chain. Once a channel is created, ownership cannot be changed (although higher-level abstractions could be implemented to provide this).

```
function chanOpenInit(
  order: ChannelOrder,
  connectionHops: [Identifier],
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  counterpartyPortIdentifier: Identifier,
  counterpartyChannelIdentifier: Identifier,
  version: string): CapabilityKey {
    abortTransactionUnless(validateChannelIdentifier(portIdentifier, channelIdentifier))

    abortTransactionUnless(connectionHops.length === 1) // for v1 of the IBC protocol

    abortTransactionUnless(provableStore.get(channelPath(portIdentifier, channelIdentifier)) === null)
    connection = provableStore.get(connectionPath(connectionHops[0]))

    // optimistic channel handshakes are allowed
    abortTransactionUnless(connection !== null)
    abortTransactionUnless(connection.state !== CLOSED)
    abortTransactionUnless(authenticate(privateStore.get(portPath(portIdentifier))))
    channel = ChannelEnd{INIT, order, counterpartyPortIdentifier,
                         counterpartyChannelIdentifier, connectionHops, version}
    provableStore.set(channelPath(portIdentifier, channelIdentifier), channel)
    key = generate()
    provableStore.set(channelCapabilityPath(portIdentifier, channelIdentifier), key)
    provableStore.set(nextSequenceSendPath(portIdentifier, channelIdentifier), 1)
    provableStore.set(nextSequenceRecvPath(portIdentifier, channelIdentifier), 1)
    return key
}
```

The `chanOpenTry` function is called by a module to accept the first step of a channel opening handshake initiated by a module on another chain.

```
function chanOpenTry(
  order: ChannelOrder,
  connectionHops: [Identifier],
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  counterpartyPortIdentifier: Identifier,
  counterpartyChannelIdentifier: Identifier,
  version: string,
  counterpartyVersion: string,
  proofInit: CommitmentProof,
  proofHeight: uint64): CapabilityKey {
    abortTransactionUnless(validateChannelIdentifier(portIdentifier, channelIdentifier))
    abortTransactionUnless(connectionHops.length === 1) // for v1 of the IBC protocol
    previous = provableStore.get(channelPath(portIdentifier, channelIdentifier))
    abortTransactionUnless(
      (previous === null) ||
      (previous.state === INIT &&
       previous.order === order &&
       previous.counterpartyPortIdentifier === counterpartyPortIdentifier &&
       previous.counterpartyChannelIdentifier === counterpartyChannelIdentifier &&
       previous.connectionHops === connectionHops &&
       previous.version === version)
      )
    abortTransactionUnless(authenticate(privateStore.get(portPath(portIdentifier))))
    connection = provableStore.get(connectionPath(connectionHops[0]))
    abortTransactionUnless(connection !== null)
    abortTransactionUnless(connection.state === OPEN)
    expected = ChannelEnd{INIT, order, portIdentifier,
                          channelIdentifier, connectionHops.reverse(), counterpartyVersion}
    abortTransactionUnless(connection.verifyChannelState(
      proofHeight,
      proofInit,
      counterpartyPortIdentifier,
      counterpartyChannelIdentifier,
      expected
    ))
    channel = ChannelEnd{TRYOPEN, order, counterpartyPortIdentifier,
                         counterpartyChannelIdentifier, connectionHops, version}
    provableStore.set(channelPath(portIdentifier, channelIdentifier), channel)
    key = generate()
    provableStore.set(channelCapabilityPath(portIdentifier, channelIdentifier), key)
    provableStore.set(nextSequenceSendPath(portIdentifier, channelIdentifier), 1)
    provableStore.set(nextSequenceRecvPath(portIdentifier, channelIdentifier), 1)
    return key
}
```

The `chanOpenAck` is called by the handshake-originating module to acknowledge the acceptance of the initial request by the counterparty module on the other chain.

```
function chanOpenAck(
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  counterpartyVersion: string,
  proofTry: CommitmentProof,
  proofHeight: uint64) {
    channel = provableStore.get(channelPath(portIdentifier, channelIdentifier))
    abortTransactionUnless(channel.state === INIT || channel.state === TRYOPEN)
    abortTransactionUnless(authenticate(privateStore.get(channelCapabilityPath(portIdentifier,
         channelIdentifier))))
    connection = provableStore.get(connectionPath(channel.connectionHops[0]))
    abortTransactionUnless(connection !== null)
    abortTransactionUnless(connection.state === OPEN)
    expected = ChannelEnd{TRYOPEN, channel.order, portIdentifier,
                          channelIdentifier, channel.connectionHops.reverse(), counterpartyVersion}
    abortTransactionUnless(connection.verifyChannelState(
      proofHeight,
      proofTry,
      channel.counterpartyPortIdentifier,
      channel.counterpartyChannelIdentifier,
      expected
    ))
    channel.state = OPEN
    channel.version = counterpartyVersion
    provableStore.set(channelPath(portIdentifier, channelIdentifier), channel)
}
```

The `chanOpenConfirm` function is called by the handshake-accepting module to acknowledge the acknowledgement of the handshake-originating module on the other chain and finish the channel opening handshake.

```
function chanOpenConfirm(
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  proofAck: CommitmentProof,
  proofHeight: uint64) {
    channel = provableStore.get(channelPath(portIdentifier, channelIdentifier))
    abortTransactionUnless(channel !== null)
    abortTransactionUnless(channel.state === TRYOPEN)
    abortTransactionUnless(authenticate(privateStore.get(channelCapabilityPath(portIdentifier,
         channelIdentifier))))
    connection = provableStore.get(connectionPath(channel.connectionHops[0]))
    abortTransactionUnless(connection !== null)
    abortTransactionUnless(connection.state === OPEN)
    expected = ChannelEnd{OPEN, channel.order, portIdentifier,
                          channelIdentifier, channel.connectionHops.reverse(), channel.version}
    abortTransactionUnless(connection.verifyChannelState(
      proofHeight,
      proofAck,
      channel.counterpartyPortIdentifier,
      channel.counterpartyChannelIdentifier,
      expected
    ))
    channel.state = OPEN
    provableStore.set(channelPath(portIdentifier, channelIdentifier), channel)
}
```

Closing handshake The `chanCloseInit` function is called by either module to close their end of the channel. Once closed, channels cannot be reopened.

Calling modules MAY atomically execute appropriate application logic in conjunction with calling `chanCloseInit`.

Any in-flight packets can be timed-out as soon as a channel is closed.

```
function chanCloseInit(
  portIdentifier: Identifier,
  channelIdentifier: Identifier) {
    abortTransactionUnless(authenticate(privateStore.get(channelCapabilityPath(portIdentifier,
         channelIdentifier))))
    channel = provableStore.get(channelPath(portIdentifier, channelIdentifier))
    abortTransactionUnless(channel !== null)
    abortTransactionUnless(channel.state !== CLOSED)
    connection = provableStore.get(connectionPath(channel.connectionHops[0]))
    abortTransactionUnless(connection !== null)
    abortTransactionUnless(connection.state === OPEN)
    channel.state = CLOSED
    provableStore.set(channelPath(portIdentifier, channelIdentifier), channel)
}
```

The `chanCloseConfirm` function is called by the counterparty module to close their end of the channel, since the other end has been closed.

Calling modules MAY atomically execute appropriate application logic in conjunction with calling `chanCloseConfirm`.

Once closed, channels cannot be reopened.

```
function chanCloseConfirm(
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  proofInit: CommitmentProof,
  proofHeight: uint64) {
    abortTransactionUnless(authenticate(privateStore.get(channelCapabilityPath(portIdentifier,
         channelIdentifier))))
    channel = provableStore.get(channelPath(portIdentifier, channelIdentifier))
    abortTransactionUnless(channel !== null)
    abortTransactionUnless(channel.state !== CLOSED)
    connection = provableStore.get(connectionPath(channel.connectionHops[0]))
    abortTransactionUnless(connection !== null)
    abortTransactionUnless(connection.state === OPEN)
    expected = ChannelEnd{CLOSED, channel.order, portIdentifier,
                          channelIdentifier, channel.connectionHops.reverse(), channel.version}
    abortTransactionUnless(connection.verifyChannelState(
      proofHeight,
      proofInit,
      channel.counterpartyPortIdentifier,
      channel.counterpartyChannelIdentifier,
      expected
    ))
    channel.state = CLOSED
    provableStore.set(channelPath(portIdentifier, channelIdentifier), channel)
}
```

Figure 4: Packet State Machine

Packet flow & handling

A day in the life of a packet The following sequence of steps must occur for a packet to be sent from module *1* on machine *A* to module *2* on machine *B*, starting from scratch.

The module can interface with the IBC handler through ICS 25 or ICS 26.

1. Initial client & port setup, in any order
    1. Client created on *A* for *B* (see ICS 2)
    2. Client created on *B* for *A* (see ICS 2)
    3. Module *1* binds to a port (see ICS 5)
    4. Module *2* binds to a port (see ICS 5), which is communicated out-of-band to module *1*
2. Establishment of a connection & channel, optimistic send, in order
    1. Connection opening handshake started from *A* to *B* by module *1* (see ICS 3)
    2. Channel opening handshake started from *1* to *2* using the newly created connection (this ICS)
    3. Packet sent over the newly created channel from *1* to *2* (this ICS)
3. Successful completion of handshakes (if either handshake fails, the connection/channel can be closed & the packet timed-out)
    1. Connection opening handshake completes successfully (see ICS 3) (this will require participation of a relayer process)
    2. Channel opening handshake completes successfully (this ICS) (this will require participation of a relayer process)
4. Packet confirmation on machine *B*, module *2* (or packet timeout if the timeout height has passed) (this will require participation of a relayer process)
5. Acknowledgement (possibly) relayed back from module *2* on machine *B* to module *1* on machine *A*

Represented spatially, packet transit between two machines can be rendered as follows:

Figure 5: Packet Transit

Sending packets The `sendPacket` function is called by a module in order to send an IBC packet on a channel end owned by the calling module to the corresponding module on the counterparty chain.

Calling modules MUST execute application logic atomically in conjunction with calling `sendPacket`.

The IBC handler performs the following steps in order:

- Checks that the channel & connection are open to send packets
- Checks that the calling module owns the sending port
- Checks that the packet metadata matches the channel & connection information
- Checks that the timeout height specified has not already passed on the destination chain
- Increments the send sequence counter associated with the channel
- Stores a constant-size commitment to the packet data & packet timeout

Note that the full packet is not stored in the state of the chain - merely a short hash-commitment to the data & timeout value. The packet data can be calculated from the transaction execution and possibly returned as log output which relayers can index.

```
function sendPacket(packet: Packet) {
    channel = provableStore.get(channelPath(packet.sourcePort, packet.sourceChannel))

    // optimistic sends are permitted once the handshake has started
    abortTransactionUnless(channel !== null)
    abortTransactionUnless(channel.state !== CLOSED)
    abortTransactionUnless(authenticate(privateStore.get(channelCapabilityPath(packet.sourcePort,
        packet.sourceChannel))))
    abortTransactionUnless(packet.destPort === channel.counterpartyPortIdentifier)
    abortTransactionUnless(packet.destChannel === channel.counterpartyChannelIdentifier)
    connection = provableStore.get(connectionPath(channel.connectionHops[0]))

    abortTransactionUnless(connection !== null)
    abortTransactionUnless(connection.state !== CLOSED)

    consensusState = provableStore.get(consensusStatePath(connection.clientIdentifier))
    abortTransactionUnless(consensusState.getHeight() < packet.timeoutHeight)

    nextSequenceSend = provableStore.get(nextSequenceSendPath(packet.sourcePort, packet.sourceChannel))
    abortTransactionUnless(packet.sequence === nextSequenceSend)

    // all assertions passed, we can alter state

    nextSequenceSend = nextSequenceSend + 1
    provableStore.set(nextSequenceSendPath(packet.sourcePort, packet.sourceChannel), nextSequenceSend)
    provableStore.set(packetCommitmentPath(packet.sourcePort, packet.sourceChannel, packet.sequence),
        hash(packet.data, packet.timeout))

    // log that a packet has been sent
    emitLogEntry("sendPacket", {sequence: packet.sequence, data: packet.data, timeout: packet.timeout})
}
```

Receiving packets The `recvPacket` function is called by a module in order to receive & process an IBC packet sent on the corresponding channel end on the counterparty chain.

Calling modules MUST execute application logic atomically in conjunction with calling `recvPacket`, likely beforehand to calculate the acknowledgement value.

The IBC handler performs the following steps in order:

- Checks that the channel & connection are open to receive packets
- Checks that the calling module owns the receiving port
- Checks that the packet metadata matches the channel & connection information
- Checks that the packet sequence is the next sequence the channel end expects to receive (for ordered channels)
- Checks that the timeout height has not yet passed
- Checks the inclusion proof of packet data commitment in the outgoing chain's state
- Sets the opaque acknowledgement value at a store path unique to the packet (if the acknowledgement is non-empty or the channel is unordered)
- Increments the packet receive sequence associated with the channel end (ordered channels only)

```
function recvPacket(
  packet: OpaquePacket,
  proof: CommitmentProof,
  proofHeight: uint64,
  acknowledgement: bytes): Packet {

    channel = provableStore.get(channelPath(packet.destPort, packet.destChannel))
    abortTransactionUnless(channel !== null)
    abortTransactionUnless(channel.state === OPEN)
    abortTransactionUnless(authenticate(privateStore.get(channelCapabilityPath(packet.destPort, packet.
         destChannel))))
    abortTransactionUnless(packet.sourcePort === channel.counterpartyPortIdentifier)
    abortTransactionUnless(packet.sourceChannel === channel.counterpartyChannelIdentifier)

    connection = provableStore.get(connectionPath(channel.connectionHops[0]))
    abortTransactionUnless(connection !== null)
    abortTransactionUnless(connection.state === OPEN)

    abortTransactionUnless(getConsensusHeight() < packet.timeoutHeight)

    abortTransactionUnless(connection.verifyPacketCommitment(
      proofHeight,
      proof,
      packet.sourcePort,
      packet.sourceChannel,
      packet.sequence,
      hash(packet.data, packet.timeout)
    ))

    // all assertions passed (except sequence check), we can alter state

    if (acknowledgement.length > 0 || channel.order === UNORDERED)
      provableStore.set(
        packetAcknowledgementPath(packet.destPort, packet.destChannel, packet.sequence),
        hash(acknowledgement)
      )

    if (channel.order === ORDERED) {
      nextSequenceRecv = provableStore.get(nextSequenceRecvPath(packet.destPort, packet.destChannel))
      abortTransactionUnless(packet.sequence === nextSequenceRecv)
      nextSequenceRecv = nextSequenceRecv + 1
      provableStore.set(nextSequenceRecvPath(packet.destPort, packet.destChannel), nextSequenceRecv)
    }

    // log that a packet has been received & acknowledged
    emitLogEntry("recvPacket", {sequence: packet.sequence, timeout: packet.timeout, data: packet.data,
         acknowledgement})

    // return transparent packet
    return packet
}
```

Acknowledgements The `acknowledgePacket` function is called by a module to process the acknowledgement of a packet previously sent by the calling module on a channel to a counterparty module on the counterparty chain. `acknowledgePacket` also cleans up the packet commitment, which is no longer necessary since the packet has been received and acted upon.

Calling modules MAY atomically execute appropriate application acknowledgement-handling logic in conjunction with calling `acknowledgePacket`.

```
function acknowledgePacket(
  packet: OpaquePacket,
  acknowledgement: bytes,
  proof: CommitmentProof,
  proofHeight: uint64): Packet {

    // abort transaction unless that channel is open, calling module owns the associated port, and the
         packet fields match
    channel = provableStore.get(channelPath(packet.sourcePort, packet.sourceChannel))
    abortTransactionUnless(channel !== null)
    abortTransactionUnless(channel.state === OPEN)
    abortTransactionUnless(authenticate(privateStore.get(channelCapabilityPath(packet.sourcePort,
         packet.sourceChannel))))
    abortTransactionUnless(packet.sourceChannel === channel.counterpartyChannelIdentifier)

    connection = provableStore.get(connectionPath(channel.connectionHops[0]))
    abortTransactionUnless(connection !== null)
    abortTransactionUnless(connection.state === OPEN)
    abortTransactionUnless(packet.sourcePort === channel.counterpartyPortIdentifier)

    // verify we sent the packet and haven't cleared it out yet
    abortTransactionUnless(provableStore.get(packetCommitmentPath(packet.sourcePort, packet.
         sourceChannel, packet.sequence))
           === hash(packet.data, packet.timeout))

    // abort transaction unless correct acknowledgement on counterparty chain
    abortTransactionUnless(connection.verifyPacketAcknowledgement(
      proofHeight,
      proof,
      packet.destPort,
      packet.destChannel,
      packet.sequence,
      hash(acknowledgement)
    ))

    // all assertions passed, we can alter state

    // delete our commitment so we can't "acknowledge" again
    provableStore.delete(packetCommitmentPath(packet.sourcePort, packet.sourceChannel, packet.sequence)
         )

    // return transparent packet
    return packet
}
```

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.

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.

Sending end The `timeoutPacket` function is called by a module which originally attempted to send a packet to a counterparty module, where the timeout height has passed on the counterparty chain without the packet being committed, to prove that the packet can no longer be executed and to allow the calling module to safely perform appropriate state transitions.

Calling modules MAY atomically execute appropriate application timeout-handling logic in conjunction with calling `timeoutPacket`.

In the case of an ordered channel, `timeoutPacket` checks the `recvSequence` of the receiving channel end and closes the channel if a packet has timed out.

In the case of an unordered channel, `timeoutPacket` checks the absence of an acknowledgement (which will have been written if the packet was received). Unordered channels are expected to continue in the face of timed-out packets.

If relations are enforced between timeout heights of subsequent packets, safe bulk timeouts of all packets prior to a timed-out packet can be performed. This specification omits details for now.

```
function timeoutPacket(
  packet: OpaquePacket,
  proof: CommitmentProof,
  proofHeight: uint64,
  nextSequenceRecv: Maybe<uint64>): Packet {

    channel = provableStore.get(channelPath(packet.sourcePort, packet.sourceChannel))
    abortTransactionUnless(channel !== null)
    abortTransactionUnless(channel.state === OPEN)

    abortTransactionUnless(authenticate(privateStore.get(channelCapabilityPath(packet.sourcePort,
         packet.sourceChannel))))
    abortTransactionUnless(packet.destChannel === channel.counterpartyChannelIdentifier)

    connection = provableStore.get(connectionPath(channel.connectionHops[0]))
    // note: the connection may have been closed
    abortTransactionUnless(packet.destPort === channel.counterpartyPortIdentifier)

    // check that timeout height has passed on the other end
    abortTransactionUnless(proofHeight >= packet.timeoutHeight)

    // check that packet has not been received
    abortTransactionUnless(nextSequenceRecv < packet.sequence)

    // verify we actually sent this packet, check the store
    abortTransactionUnless(provableStore.get(packetCommitmentPath(packet.sourcePort, packet.
         sourceChannel, packet.sequence))
           === hash(packet.data, packet.timeout))

    if channel.order === ORDERED
      // ordered channel: check that the recv sequence is as claimed
      abortTransactionUnless(connection.verifyNextSequenceRecv(
        proofHeight,
        proof,
        packet.destPort,
        packet.destChannel,
        nextSequenceRecv
      ))
    else
      // unordered channel: verify absence of acknowledgement at packet index
      abortTransactionUnless(connection.verifyPacketAcknowledgementAbsence(
        proofHeight,
        proof,
        packet.sourcePort,
        packet.sourceChannel,
        packet.sequence
      ))

    // all assertions passed, we can alter state

    // delete our commitment
    provableStore.delete(packetCommitmentPath(packet.sourcePort, packet.sourceChannel, packet.sequence)
         )

    if channel.order === ORDERED {
      // ordered channel: close the channel
      channel.state = CLOSED
      provableStore.set(channelPath(packet.sourcePort, packet.sourceChannel), channel)
    }

    // return transparent packet
    return packet
}
```

Timing-out on close The `timeoutOnClose` function is called by a module in order to prove that the channel to which an unreceived packet was addressed has been closed, so the packet will never be received (even if the `timeoutHeight` has not yet been reached).

```
function timeoutOnClose(
  packet: Packet,
  proof: CommitmentProof,
  proofClosed: CommitmentProof,
  proofHeight: uint64,
  nextSequenceRecv: Maybe<uint64>): Packet {

    channel = provableStore.get(channelPath(packet.sourcePort, packet.sourceChannel))
    // note: the channel may have been closed
    abortTransactionUnless(authenticate(privateStore.get(channelCapabilityPath(packet.sourcePort,
         packet.sourceChannel))))
    abortTransactionUnless(packet.destChannel === channel.counterpartyChannelIdentifier)

    connection = provableStore.get(connectionPath(channel.connectionHops[0]))
    // note: the connection may have been closed
    abortTransactionUnless(packet.destPort === channel.counterpartyPortIdentifier)

    // verify we actually sent this packet, check the store
    abortTransactionUnless(provableStore.get(packetCommitmentPath(packet.sourcePort, packet.
         sourceChannel, packet.sequence))
           === hash(packet.data, packet.timeout))

    // check that the opposing channel end has closed
    expected = ChannelEnd{CLOSED, channel.order, channel.portIdentifier,
                          channel.channelIdentifier, channel.connectionHops.reverse(), channel.version}
    abortTransactionUnless(connection.verifyChannelState(
      proofHeight,
      proofClosed,
      channel.counterpartyPortIdentifier,
      channel.counterpartyChannelIdentifier,
      expected
    ))

    if channel.order === ORDERED
      // ordered channel: check that the recv sequence is as claimed
      abortTransactionUnless(connection.verifyNextSequenceRecv(
        proofHeight,
        proof,
        packet.destPort,
        packet.destChannel,
        nextSequenceRecv
      ))
    else
      // unordered channel: verify absence of acknowledgement at packet index
      abortTransactionUnless(connection.verifyPacketAcknowledgementAbsence(
        proofHeight,
        proof,
        packet.sourcePort,
        packet.sourceChannel,
        packet.sequence
      ))

    // all assertions passed, we can alter state

    // delete our commitment
    provableStore.delete(packetCommitmentPath(packet.sourcePort, packet.sourceChannel, packet.sequence)
        )

    // return transparent packet
    return packet
}
```

Cleaning up state `cleanupPacket` is called by a module to remove a received packet commitment from storage. The receiving end must have already processed the packet (whether regularly or past timeout).

In the ordered channel case, `cleanupPacket` cleans-up a packet on an ordered channel by proving that the packet has been received on the other end.

In the unordered channel case, `cleanupPacket` cleans-up a packet on an unordered channel by proving that the associated acknowledgement has been written.

```
function cleanupPacket(
  packet: OpaquePacket,
  proof: CommitmentProof,
  proofHeight: uint64,
  nextSequenceRecvOrAcknowledgement: Either<uint64, bytes>): Packet {

    channel = provableStore.get(channelPath(packet.sourcePort, packet.sourceChannel))
    abortTransactionUnless(channel !== null)
    abortTransactionUnless(channel.state === OPEN)
    abortTransactionUnless(authenticate(privateStore.get(channelCapabilityPath(packet.sourcePort,
        packet.sourceChannel))))
    abortTransactionUnless(packet.destChannel === channel.counterpartyChannelIdentifier)

    connection = provableStore.get(connectionPath(channel.connectionHops[0]))
    // note: the connection may have been closed
    abortTransactionUnless(packet.destPort === channel.counterpartyPortIdentifier)

    // abortTransactionUnless packet has been received on the other end
    abortTransactionUnless(nextSequenceRecv > packet.sequence)

    // verify we actually sent the packet, check the store
    abortTransactionUnless(provableStore.get(packetCommitmentPath(packet.sourcePort, packet.
        sourceChannel, packet.sequence))
           === hash(packet.data, packet.timeout))

    if channel.order === ORDERED
      // check that the recv sequence is as claimed
      abortTransactionUnless(connection.verifyNextSequenceRecv(
        proofHeight,
        proof,
        packet.destPort,
        packet.destChannel,
          nextSequenceRecvOrAcknowledgement
        ))
      else
        // abort transaction unless acknowledgement on the other end
        abortTransactionUnless(connection.verifyPacketAcknowledgement(
          proofHeight,
          proof,
          packet.destPort,
          packet.destChannel,
          packet.sequence,
          nextSequenceRecvOrAcknowledgement
        ))

      // all assertions passed, we can alter state

      // clear the store
      provableStore.delete(packetCommitmentPath(packet.sourcePort, packet.sourceChannel, packet.sequence)
          )

      // return transparent packet
      return packet
  }
```

Reasoning about race conditions

Simultaneous handshake attempts If two machines simultaneously initiate channel opening handshakes with each other, attempting to use the same identifiers, both will fail and new identifiers must be used.

Identifier allocation There is an unavoidable race condition on identifier allocation on the destination chain. Modules would be well-advised to utilise pseudo-random, non-valuable identifiers. Managing to claim the identifier that another module wishes to use, however, while annoying, cannot man-in-the-middle a handshake since the receiving module must already own the port to which the handshake was targeted.

Timeouts / packet confirmation There is no race condition between a packet timeout and packet confirmation, as the packet will either have passed the timeout height prior to receipt or not.

Man-in-the-middle attacks during handshakes Verification of cross-chain state prevents man-in-the-middle attacks for both connection handshakes & channel handshakes since all information (source, destination client, channel, etc.) is known by the module which starts the handshake and confirmed prior to handshake completion.

Connection / channel closure with in-flight packets If a connection or channel is closed while packets are in-flight, the packets can no longer be received on the destination chain and can be timed-out on the source chain.

Querying channels Channels can be queried with `queryChannel`:

```
function queryChannel(connId: Identifier, chanId: Identifier): ChannelEnd | void {
    return provableStore.get(channelPath(connId, chanId))
}
```

8.2.5 Properties & Invariants

- The unique combinations of channel & port identifiers are first-come-first-serve: once a pair has been allocated, only the modules owning the ports in question can send or receive on that channel.
- Packets are delivered exactly once, assuming that the chains are live within the timeout window, and in case of timeout can be timed-out exactly once on the sending chain.
- The channel handshake cannot be man-in-the-middle attacked by another module on either blockchain or another blockchain's IBC handler.

9 ICS 025 - Handler Interface

9.1 Synopsis

This document describes the interface exposed by the standard IBC implementation (referred to as the IBC handler) to modules within the same state machine, and the implementation of that interface by the IBC handler.

9.1.1 Motivation

IBC is an inter-module communication protocol, designed to facilitate reliable, authenticated message passing between modules on separate blockchains. Modules should be able to reason about the interface they interact with and the requirements they must adhere to in order to utilise the interface safely.

9.1.2 Definitions

Associated definitions are as defined in referenced prior standards (where the functions are defined), where appropriate.

9.1.3 Desired Properties

- Creation of clients, connections, and channels should be as permissionless as possible.
- The module set should be dynamic: chains should be able to add and destroy modules, which can themselves bind to and unbind from ports, at will with a persistent IBC handler.
- Modules should be able to write their own more complex abstractions on top of IBC to provide additional semantics or guarantees.

9.2 Technical Specification

> Note: If the host state machine is utilising object capability authentication (see ICS 005), all functions utilising ports take an additional capability key parameter.

9.2.1 Client lifecycle management

By default, clients are unowned: any module may create a new client, query any existing client, update any existing client, and delete any existing client not in use.

The handler interface exposes `createClient`, `updateClient`, `queryClientConsensusState`, `queryClient`, and `submitMisbehaviourToClient` as defined in ICS 2.

9.2.2 Connection lifecycle management

The handler interface exposes `connOpenInit`, `connOpenTry`, `connOpenAck`, `connOpenConfirm`, and `queryConnection`, as defined in ICS 3.

The default IBC routing module SHALL allow external calls to `connOpenTry`, `connOpenAck`, and `connOpenConfirm`.

9.2.3 Channel lifecycle management

By default, channels are owned by the creating port, meaning only the module bound to that port is allowed to inspect, close, or send on the channel. A module can create any number of channels utilising the same port.

The handler interface exposes `chanOpenInit`, `chanOpenTry`, `chanOpenAck`, `chanOpenConfirm`, `chanCloseInit`, `chanCloseConfirm`, and `queryChannel`, as defined in ICS 4.

The default IBC routing module SHALL allow external calls to `chanOpenTry`, `chanOpenAck`, `chanOpenConfirm`, and `chanCloseConfirm`
.

9.2.4 Packet relay

Packets are permissioned by channel (only a port which owns a channel can send or receive on it).

The handler interface exposes `sendPacket`, `recvPacket`, `acknowledgePacket`, `timeoutPacket`, `timeoutOnClose`, and `cleanupPacket` as defined in ICS 4.

The default IBC routing module SHALL allow external calls to `sendPacket`, `recvPacket`, `acknowledgePacket`, `timeoutPacket`, `timeoutOnClose`, and `cleanupPacket`.

9.2.5 Properties & Invariants

The IBC handler module interface as defined here inherits properties of functions as defined in their associated specifications.

10 ICS 026 - Routing Module

10.1 Synopsis

The routing module is a default implementation of a secondary module which will accept external datagrams and call into the interblockchain communication protocol handler to deal with handshakes and packet relay. The routing module keeps a lookup table of modules, which it can use to look up and call a module when a packet is received, so that external relayers need only ever relay packets to the routing module.

10.1.1 Motivation

The default IBC handler uses a receiver call pattern, where modules must individually call the IBC handler in order to bind to ports, start handshakes, accept handshakes, send and receive packets, etc. This is flexible and simple (see Design Patterns) but is a bit tricky to understand and may require extra work on the part of relayer processes, who must track the state of many modules. This standard describes an IBC "routing module" to automate most common functionality, route packets, and simplify the task of relayers.

The routing module can also play the role of the module manager as discussed in ICS 5 and implement logic to determine when modules are allowed to bind to ports and what those ports can be named.

10.1.2 Definitions

All functions provided by the IBC handler interface are defined as in ICS 25.

The functions `generate` & `authenticate` are defined as in ICS 5.

10.1.3 Desired Properties

- Modules should be able to bind to ports and own channels through the routing module.
- No overhead should be added for packet sends and receives other than the layer of call indirection.
- The routing module should call specified handler functions on modules when they need to act upon packets.

10.2 Technical Specification

> Note: If the host state machine is utilising object capability authentication (see ICS 005), all functions utilising ports take an additional capability parameter.

10.2.1 Module callback interface

Modules must expose the following function signatures to the routing module, which are called upon the receipt of various datagrams:

```
function onChanOpenInit(
  order: ChannelOrder,
  connectionHops: [Identifier],
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  counterpartyPortIdentifier: Identifier,
  counterpartyChannelIdentifier: Identifier,
  version: string) {
    // defined by the module
}

function onChanOpenTry(
  order: ChannelOrder,
  connectionHops: [Identifier],
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  counterpartyPortIdentifier: Identifier,
  counterpartyChannelIdentifier: Identifier,
  version: string,
  counterpartyVersion: string) {
    // defined by the module
}

function onChanOpenAck(
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  version: string) {
    // defined by the module
}

function onChanOpenConfirm(
  portIdentifier: Identifier,
  channelIdentifier: Identifier) {
    // defined by the module
}

function onChanCloseInit(
  portIdentifier: Identifier,
  channelIdentifier: Identifier) {
    // defined by the module
}

function onChanCloseConfirm(
  portIdentifier: Identifier,
  channelIdentifier: Identifier): void {
    // defined by the module
}

function onRecvPacket(packet: Packet): bytes {
    // defined by the module, returns acknowledgement
}

function onTimeoutPacket(packet: Packet) {
    // defined by the module
}

function onAcknowledgePacket(packet: Packet) {
    // defined by the module
}

function onTimeoutPacketClose(packet: Packet) {
    // defined by the module
}
```

Exceptions MUST be thrown to indicate failure and reject the handshake, incoming packet, etc.

These are combined together in a `ModuleCallbacks` interface:

```
interface ModuleCallbacks {
  onChanOpenInit: onChanOpenInit,
  onChanOpenTry: onChanOpenTry,
  onChanOpenAck: onChanOpenAck,
  onChanOpenConfirm: onChanOpenConfirm,
  onChanCloseConfirm: onChanCloseConfirm
  onRecvPacket: onRecvPacket
  onTimeoutPacket: onTimeoutPacket
  onAcknowledgePacket: onAcknowledgePacket,
  onTimeoutPacketClose: onTimeoutPacketClose
}
```

Callbacks are provided when the module binds to a port.

```
function callbackPath(portIdentifier: Identifier): Path {
    return "callbacks/{portIdentifier}"
}
```

The calling module identifier is also stored for future authentication should the callbacks need to be altered.

```
function authenticationPath(portIdentifier: Identifier): Path {
    return "authentication/{portIdentifier}"
}
```

10.2.2 Port binding as module manager

The IBC routing module sits in-between the handler module (ICS 25) and individual modules on the host state machine.

The routing module, acting as a module manager, differentiates between two kinds of ports:

- "Existing name" ports: e.g. "bank", with standardised prior meanings, which should not be first-come-first-serve
- "Fresh name" ports: new identity (perhaps a smart contract) w/no prior relationships, new random number port, post-generation port name can be communicated over another channel

A set of existing names are allocated, along with corresponding modules, when the routing module is instantiated by the host state machine. The routing module then allows allocation of fresh ports at any time by modules, but they must use a specific standardised prefix.

The function `bindPort` can be called by a module in order to bind to a port, through the routing module, and set up callbacks.

```
function bindPort(
  id: Identifier,
  callbacks: Callbacks) {
    abortTransactionUnless(privateStore.get(callbackPath(id)) === null)
    handler.bindPort(id)
    capability = generate()
    privateStore.set(authenticationPath(id), capability)
    privateStore.set(callbackPath(id), callbacks)
}
```

The function `updatePort` can be called by a module in order to alter the callbacks.

```
function updatePort(
  id: Identifier,
  newCallbacks: Callbacks) {
    abortTransactionUnless(authenticate(privateStore.get(authenticationPath(id))))
    privateStore.set(callbackPath(id), newCallbacks)
}
```

The function `releasePort` can be called by a module in order to release a port previously in use.

> Warning: releasing a port will allow other modules to bind to that port and possibly intercept incoming channel opening handshakes. Modules should release ports only when doing so is safe.

```
function releasePort(id: Identifier) {
    abortTransactionUnless(authenticate(privateStore.get(authenticationPath(id))))
    handler.releasePort(id)
    privateStore.delete(callbackPath(id))
    privateStore.delete(authenticationPath(id))
}
```

The function `lookupModule` can be used by the routing module to lookup the callbacks bound to a particular port.

```
function lookupModule(portId: Identifier) {
    return privateStore.get(callbackPath(portId))
}
```

10.2.3 Datagram handlers (write)

Datagrams are external data blobs accepted as transactions by the routing module. This section defines a *handler function* for each datagram, which is executed when the associated datagram is submitted to the routing module in a transaction.

All datagrams can also be safely submitted by other modules to the routing module.

No message signatures or data validity checks are assumed beyond those which are explicitly indicated.

Client lifecycle management `ClientCreate` creates a new light client with the specified identifier & consensus state.

```
interface ClientCreate {
  identifier: Identifier
  type: ClientType
  consensusState: ConsensusState
}
```

```
function handleClientCreate(datagram: ClientCreate) {
    handler.createClient(datagram.identifier, datagram.type, datagram.consensusState)
}
```

`ClientUpdate` updates an existing light client with the specified identifier & new header.

```
interface ClientUpdate {
  identifier: Identifier
  header: Header
}
```

```
function handleClientUpdate(datagram: ClientUpdate) {
    handler.updateClient(datagram.identifier, datagram.header)
}
```

`ClientSubmitMisbehaviour` submits proof-of-misbehaviour to an existing light client with the specified identifier.

```
interface ClientMisbehaviour {
  identifier: Identifier
  evidence: bytes
}
```

```
function handleClientMisbehaviour(datagram: ClientUpdate) {
    handler.submitMisbehaviourToClient(datagram.identifier, datagram.evidence)
}
```

Connection lifecycle management The `ConnOpenInit` datagram starts the connection handshake process with an IBC module on another chain.

```
interface ConnOpenInit {
  identifier: Identifier
  desiredCounterpartyIdentifier: Identifier
  clientIdentifier: Identifier
  counterpartyClientIdentifier: Identifier
  version: string
}
```

```
function handleConnOpenInit(datagram: ConnOpenInit) {
    handler.connOpenInit(
      datagram.identifier,
      datagram.desiredCounterpartyIdentifier,
      datagram.clientIdentifier,
      datagram.counterpartyClientIdentifier,
      datagram.version
    )
}
```

The `ConnOpenTry` datagram accepts a handshake request from an IBC module on another chain.

```
interface ConnOpenTry {
  desiredIdentifier: Identifier
  counterpartyConnectionIdentifier: Identifier
  counterpartyClientIdentifier: Identifier
  clientIdentifier: Identifier
  version: string
  counterpartyVersion: string
  proofInit: CommitmentProof
  proofConsensus: CommitmentProof
  proofHeight: uint64
  consensusHeight: uint64
}
```

```
function handleConnOpenTry(datagram: ConnOpenTry) {
    handler.connOpenTry(
      datagram.desiredIdentifier,
      datagram.counterpartyConnectionIdentifier,
      datagram.counterpartyClientIdentifier,
      datagram.clientIdentifier,
      datagram.version,
      datagram.counterpartyVersion,
      datagram.proofInit,
      datagram.proofConsensus,
      datagram.proofHeight,
      datagram.consensusHeight
    )
}
```

The `ConnOpenAck` datagram confirms a handshake acceptance by the IBC module on another chain.

```
interface ConnOpenAck {
  identifier: Identifier
  version: string
  proofTry: CommitmentProof
  proofConsensus: CommitmentProof
  proofHeight: uint64
  consensusHeight: uint64
}
```

```
function handleConnOpenAck(datagram: ConnOpenAck) {
    handler.connOpenAck(
      datagram.identifier,
      datagram.version,
      datagram.proofTry,
      datagram.proofConsensus,
      datagram.proofHeight,
      datagram.consensusHeight
    )
}
```

The `ConnOpenConfirm` datagram acknowledges a handshake acknowledgement by an IBC module on another chain & finalises the connection.

```
interface ConnOpenConfirm {
  identifier: Identifier
  proofAck: CommitmentProof
  proofHeight: uint64
}
```

```
function handleConnOpenConfirm(datagram: ConnOpenConfirm) {
    handler.connOpenConfirm(
      datagram.identifier,
      datagram.proofAck,
      datagram.proofHeight
    )
}
```

Channel lifecycle management

```
interface ChanOpenInit {
  order: ChannelOrder
  connectionHops: [Identifier]
  portIdentifier: Identifier
  channelIdentifier: Identifier
  counterpartyPortIdentifier: Identifier
  counterpartyChannelIdentifier: Identifier
  version: string
}
```

```
function handleChanOpenInit(datagram: ChanOpenInit) {
    module = lookupModule(datagram.portIdentifier)
    module.onChanOpenInit(
      datagram.order,
      datagram.connectionHops,
      datagram.portIdentifier,
      datagram.channelIdentifier,
      datagram.counterpartyPortIdentifier,
      datagram.counterpartyChannelIdentifier,
      datagram.version
    )
    handler.chanOpenInit(
      datagram.order,
      datagram.connectionHops,
      datagram.portIdentifier,
      datagram.channelIdentifier,
      datagram.counterpartyPortIdentifier,
      datagram.counterpartyChannelIdentifier,
      datagram.version
    )
}
```

```
interface ChanOpenTry {
  order: ChannelOrder
  connectionHops: [Identifier]
  portIdentifier: Identifier
  channelIdentifier: Identifier
  counterpartyPortIdentifier: Identifier
  counterpartyChannelIdentifier: Identifier
  version: string
  counterpartyVersion: string
  proofInit: CommitmentProof
  proofHeight: uint64
}
```

```
function handleChanOpenTry(datagram: ChanOpenTry) {
    module = lookupModule(datagram.portIdentifier)
    module.onChanOpenTry(
      datagram.order,
      datagram.connectionHops,
      datagram.portIdentifier,
      datagram.channelIdentifier,
      datagram.counterpartyPortIdentifier,
      datagram.counterpartyChannelIdentifier,
      datagram.version,
      datagram.counterpartyVersion
    )
    handler.chanOpenTry(
      datagram.order,
      datagram.connectionHops,
      datagram.portIdentifier,
      datagram.channelIdentifier,
      datagram.counterpartyPortIdentifier,
      datagram.counterpartyChannelIdentifier,
      datagram.version,
      datagram.counterpartyVersion,
      datagram.proofInit,
      datagram.proofHeight
    )
}
```

```
interface ChanOpenAck {
  portIdentifier: Identifier
  channelIdentifier: Identifier
  version: string
  proofTry: CommitmentProof
  proofHeight: uint64
}
```

```
function handleChanOpenAck(datagram: ChanOpenAck) {
    module.onChanOpenAck(
      datagram.portIdentifier,
      datagram.channelIdentifier,
      datagram.version
    )
    handler.chanOpenAck(
      datagram.portIdentifier,
      datagram.channelIdentifier,
      datagram.version,
      datagram.proofTry,
      datagram.proofHeight
    )
}
```

```
interface ChanOpenConfirm {
  portIdentifier: Identifier
  channelIdentifier: Identifier
  proofAck: CommitmentProof
  proofHeight: uint64
}
```

```
function handleChanOpenConfirm(datagram: ChanOpenConfirm) {
    module = lookupModule(datagram.portIdentifier)
    module.onChanOpenConfirm(
      datagram.portIdentifier,
      datagram.channelIdentifier
    )
    handler.chanOpenConfirm(
      datagram.portIdentifier,
      datagram.channelIdentifier,
      datagram.proofAck,
      datagram.proofHeight
    )
}
```

```
interface ChanCloseInit {
  portIdentifier: Identifier
  channelIdentifier: Identifier
}
```

```
function handleChanCloseInit(datagram: ChanCloseInit) {
    module = lookupModule(datagram.portIdentifier)
    module.onChanCloseInit(
      datagram.portIdentifier,
      datagram.channelIdentifier
    )
    handler.chanCloseInit(
      datagram.portIdentifier,
      datagram.channelIdentifier
    )
}
```

```
interface ChanCloseConfirm {
  portIdentifier: Identifier
  channelIdentifier: Identifier
  proofInit: CommitmentProof
  proofHeight: uint64
}
```

```
function handleChanCloseConfirm(datagram: ChanCloseConfirm) {
    module = lookupModule(datagram.portIdentifier)
    module.onChanCloseConfirm(
      datagram.portIdentifier,
      datagram.channelIdentifier
    )
    handler.chanCloseConfirm(
      datagram.portIdentifier,
      datagram.channelIdentifier,
      datagram.proofInit,
      datagram.proofHeight
    )
}
```

Packet relay Packets are sent by the module directly (by the module calling the IBC handler).

```
interface PacketRecv {
  packet: Packet
  proof: CommitmentProof
  proofHeight: uint64
}
```

```
function handlePacketRecv(datagram: PacketRecv) {
    module = lookupModule(datagram.packet.sourcePort)
    acknowledgement = module.onRecvPacket(datagram.packet)
    handler.recvPacket(
      datagram.packet,
      datagram.proof,
      datagram.proofHeight,
      acknowledgement
    )
}
```

```
interface PacketAcknowledgement {
  packet: Packet
  acknowledgement: string
  proof: CommitmentProof
  proofHeight: uint64
}
```

```
function handlePacketAcknowledgement(datagram: PacketAcknowledgement) {
    module = lookupModule(datagram.packet.sourcePort)
    module.onAcknowledgePacket(
      datagram.packet,
      datagram.acknowledgement
    )
    handler.acknowledgePacket(
      datagram.packet,
      datagram.acknowledgement,
      datagram.proof,
      datagram.proofHeight
    )
}
```

Packet timeouts

```
interface PacketTimeout {
  packet: Packet
  proof: CommitmentProof
  proofHeight: uint64
  nextSequenceRecv: Maybe<uint64>
}
```

```
function handlePacketTimeout(datagram: PacketTimeout) {
    module = lookupModule(datagram.packet.sourcePort)
    module.onTimeoutPacket(datagram.packet)
    handler.timeoutPacket(
      datagram.packet,
      datagram.proof,
      datagram.proofHeight,
      datagram.nextSequenceRecv
    )
}
```

```
interface PacketTimeoutOnClose {
  packet: Packet
  proof: CommitmentProof
  proofHeight: uint64
}
```

```
function handlePacketTimeoutOnClose(datagram: PacketTimeoutOnClose) {
    module = lookupModule(datagram.packet.sourcePort)
    module.onTimeoutPacket(datagram.packet)
    handler.timeoutOnClose(
      datagram.packet,
      datagram.proof,
      datagram.proofHeight
    )
}
```

Closure-by-timeout & packet cleanup

```
interface PacketCleanup {
  packet: Packet
  proof: CommitmentProof
  proofHeight: uint64
  nextSequenceRecvOrAcknowledgement: Either<uint64, bytes>
}
```

```
function handlePacketCleanup(datagram: PacketCleanup) {
    handler.cleanupPacket(
      datagram.packet,
      datagram.proof,
      datagram.proofHeight,
      datagram.nextSequenceRecvOrAcknowledgement
    )
}
```

10.2.4 Query (read-only) functions

All query functions for clients, connections, and channels should be exposed (read-only) directly by the IBC handler module.

10.2.5 Interface usage example

See ICS 20 for a usage example.

10.2.6 Properties & Invariants

- Proxy port binding is first-come-first-serve: once a module binds to a port through the IBC routing module, only that module can utilise that port until the module releases it.

11 ICS 018 - Relayer Algorithms

11.1 Synopsis

Relayer algorithms are the “physical” connection layer of IBC — off-chain processes responsible for relaying data between two chains running the IBC protocol by scanning the state of each chain, constructing appropriate datagrams, and executing them on the opposite chain as allowed by the protocol.

11.1.1 Motivation

In the IBC protocol, a blockchain can only record the *intention* to send particular data to another chain — it does not have direct access to a network transport layer. Physical datagram relay must be performed by off-chain infrastructure with access to a transport layer such as TCP/IP. This standard defines the concept of a *relayer* algorithm, executable by an off-chain process with the ability to query chain state, to perform this relay.

11.1.2 Definitions

A *relayer* is an off-chain process with the ability to read the state of and submit transactions to some set of ledgers utilising the IBC protocol.

11.1.3 Desired Properties

- No exactly-once or deliver-or-timeout safety properties of IBC should depend on relayer behaviour (assume Byzantine relayers).
- Packet relay liveness properties of IBC should depend only on the existence of at least one correct, live relayer.
- Relaying should be permissionless, all requisite verification should be performed on-chain.
- Requisite communication between the IBC user and the relayer should be minimised.
- Provision for relayer incentivisation should be possible at the application layer.

11.2 Technical Specification

11.2.1 Basic relayer algorithm

The relayer algorithm is defined over a set `C` of chains implementing the IBC protocol. Each relayer may not necessarily have access to read state from and write datagrams to all chains in the interchain network (especially in the case of permissioned or private chains) — different relayers may relay between different subsets.

`pendingDatagrams` calculates the set of all valid datagrams to be relayed from one chain to another based on the state of both chains. The relayer must possess prior knowledge of what subset of the IBC protocol is implemented by the blockchains in the set for which they are relaying (e.g. by reading the source code). An example is defined below.

`submitDatagram` is a procedure defined per-chain (submitting a transaction of some sort). Datagrams can be submitted individually as single transactions or atomically as a single transaction if the chain supports it.

`relay` is called by the relayer every so often — no more frequently than once per block on either chain, and possibly less frequently, according to how often the relayer wishes to relay.

Different relayers may relay between different chains — as long as each pair of chains has at least one correct & live relayer and the chains remain live, all packets flowing between chains in the network will eventually be relayed.

```
function relay(C: Set<Chain>) {
  for (const chain of C)
    for (const counterparty of C)
      if (counterparty !== chain) {
        const datagrams = chain.pendingDatagrams(counterparty)
        for (const localDatagram of datagrams[0])
          chain.submitDatagram(localDatagram)
        for (const counterpartyDatagram of datagrams[1])
          counterparty.submitDatagram(counterpartyDatagram)
      }
}
```

11.2.2 Pending datagrams

`pendingDatagrams` collates datagrams to be sent from one machine to another. The implementation of this function will depend on the subset of the IBC protocol supported by both machines & the state layout of the source machine. Particular relayers will likely also want to implement their own filter functions in order to relay only a subset of the datagrams which could possibly be relayed (e.g. the subset for which they have been paid to relay in some off-chain manner).

An example implementation which performs unidirectional relay between two chains is outlined below. It can be altered to perform bidirectional relay by switching `chain` and `counterparty`. Which relayer process is responsible for which datagrams is a flexible choice - in this example, the relayer process relays all handshakes which started on `chain` (sending datagrams to both chains), relays all packets sent from `chain` to `counterparty`, and relays all acknowledgements of packets sent from `counterparty` to `chain`.

```
function pendingDatagrams(chain: Chain, counterparty: Chain): List<Set<Datagram>> {
  const localDatagrams = []
  const counterpartyDatagrams = []

  // ICS2 : Clients
  // - Determine if light client needs to be updated (local & counterparty)
  height = chain.latestHeight()
  client = counterparty.queryClientConsensusState(chain)
  if client.height < height {
    header = chain.latestHeader()
    counterpartyDatagrams.push(ClientUpdate{chain, header})
  }
  counterpartyHeight = counterparty.latestHeight()
  client = chain.queryClientConsensusState(counterparty)
  if client.height < counterpartyHeight {
    header = counterparty.latestHeader()
    localDatagrams.push(ClientUpdate{counterparty, header})
  }

  // ICS3 : Connections
  // - Determine if any connection handshakes are in progress
  connections = chain.getConnectionsUsingClient(counterparty)
  for (const localEnd of connections) {
    remoteEnd = counterparty.getConnection(localEnd.counterpartyIdentifier)
    if (localEnd.state === INIT && remoteEnd === null)
      // Handshake has started locally (1 step done), relay `connOpenTry` to the remote end
      counterpartyDatagrams.push(ConnOpenTry{
        desiredIdentifier: localEnd.counterpartyConnectionIdentifier,
        counterpartyConnectionIdentifier: localEnd.identifier,
        counterpartyClientIdentifier: localEnd.clientIdentifier,
        clientIdentifier: localEnd.counterpartyClientIdentifier,
        version: localEnd.version,
        counterpartyVersion: localEnd.version,
        proofInit: localEnd.proof(),
        proofConsensus: localEnd.client.consensusState.proof(),
        proofHeight: height,
        consensusHeight: localEnd.client.height,
      })
    else if (localEnd.state === INIT && remoteEnd.state === TRYOPEN)
      // Handshake has started on the other end (2 steps done), relay `connOpenAck` to the local end
      localDatagrams.push(ConnOpenAck{
        identifier: localEnd.identifier,
        version: remoteEnd.version,
        proofTry: remoteEnd.proof(),
        proofConsensus: remoteEnd.client.consensusState.proof(),
        proofHeight: remoteEnd.client.height,
        consensusHeight: remoteEnd.client.height,
      })
    else if (localEnd.state === OPEN && remoteEnd.state === TRYOPEN)
      // Handshake has confirmed locally (3 steps done), relay `connOpenConfirm` to the remote end
      counterpartyDatagrams.push(ConnOpenConfirm{
        identifier: remoteEnd.identifier,
        proofAck: localEnd.proof(),
        proofHeight: height,
      })
  }

  // ICS4 : Channels & Packets
  // - Determine if any channel handshakes are in progress
  // - Determine if any packets, acknowledgements, or timeouts need to be relayed
  channels = chain.getChannelsUsingConnections(connections)
  for (const localEnd of channels) {
    remoteEnd = counterparty.getConnection(localEnd.counterpartyIdentifier)
    // Deal with handshakes in progress
    if (localEnd.state === INIT && remoteEnd === null)
      // Handshake has started locally (1 step done), relay `chanOpenTry` to the remote end
      counterpartyDatagrams.push(ChanOpenTry{
        order: localEnd.order,
        connectionHops: localEnd.connectionHops.reverse(),
        portIdentifier: localEnd.counterpartyPortIdentifier,
        channelIdentifier: localEnd.counterpartyChannelIdentifier,
        counterpartyPortIdentifier: localEnd.portIdentifier,
        counterpartyChannelIdentifier: localEnd.channelIdentifier,
        version: localEnd.version,
        counterpartyVersion: localEnd.version,
        proofInit: localEnd.proof(),
        proofHeight: height,
      })
    else if (localEnd.state === INIT && remoteEnd.state === TRYOPEN)
      // Handshake has started on the other end (2 steps done), relay `chanOpenAck` to the local end
      localDatagrams.push(ChanOpenAck{
        portIdentifier: localEnd.portIdentifier,
        channelIdentifier: localEnd.channelIdentifier,
        version: remoteEnd.version,
        proofTry: remoteEnd.proof(),
        proofHeight: localEnd.client.height,
      })
    else if (localEnd.state === OPEN && remoteEnd.state === TRYOPEN)
      // Handshake has confirmed locally (3 steps done), relay `chanOpenConfirm` to the remote end
      counterpartyDatagrams.push(ChanOpenConfirm{
        portIdentifier: remoteEnd.portIdentifier,
        channelIdentifier: remoteEnd.channelIdentifier,
        proofAck: localEnd.proof(),
        proofHeight: height
      })

    // Deal with packets
    // First, scan logs for sent packets and relay all of them
    sentPacketLogs = queryByTopic(height, "sendPacket")
    for (const logEntry of sentPacketLogs) {
      // relay packet with this sequence number
      packetData = Packet{logEntry.sequence, logEntry.timeout, localEnd.portIdentifier, localEnd.
```

```
              channelIdentifier,
                          remoteEnd.portIdentifier, remoteEnd.channelIdentifier, logEntry.data}
      counterpartyDatagrams.push(PacketRecv{
        packet: packetData,
        proof: packet.proof(),
        proofHeight: height,
      })
    }
    // Then, scan logs for received packets and relay acknowledgements
    recvPacketLogs = queryByTopic(height, "recvPacket")
    for (const logEntry of recvPacketLogs) {
      // relay packet acknowledgement with this sequence number
      packetData = Packet{logEntry.sequence, logEntry.timeout, localEnd.portIdentifier, localEnd.
              channelIdentifier,
                          remoteEnd.portIdentifier, remoteEnd.channelIdentifier, logEntry.data}
      counterpartyDatagrams.push(PacketAcknowledgement{
        packet: packetData,
        acknowledgement: logEntry.acknowledgement,
        proof: packet.proof(),
        proofHeight: height,
      })
    }
  }

  return [localDatagrams, counterpartyDatagrams]
}
```

Relayers may elect to filter these datagrams in order to relay particular clients, particular connections, particular channels, or even particular kinds of packets, perhaps in accordance with the fee payment model (which this document does not specify, as it may vary).

11.2.3 Ordering constraints

There are implicit ordering constraints imposed on the relayer process determining which datagrams must be submitted in what order. For example, a header must be submitted to finalise the stored consensus state & commitment root for a particular height in a light client before a packet can be relayed. The relayer process is responsible for frequently querying the state of the chains between which they are relaying in order to determine what must be relayed when.

11.2.4 Bundling

If the host state machine supports it, the relayer process can bundle many datagrams into a single transaction, which will cause them to be executed in sequence, and amortise any overhead costs (e.g. signature checks for fee payment).

11.2.5 Race conditions

Multiple relayers relaying between the same pair of modules & chains may attempt to relay the same packet (or submit the same header) at the same time. If two relayers do so, the first transaction will succeed and the second will fail. Out-of-band coordination between the relayers or between the actors who sent the original packets and the relayers is necessary to mitigate this. Further discussion is out of scope of this standard.

11.2.6 Incentivisation

The relay process must have access to accounts on both chains with sufficient balance to pay for transaction fees. Relayers may employ application-level methods to recoup these fees, such by including a small payment to themselves in the packet data — protocols for relayer fee payment will be described in future versions of this ICS or in separate ICSs.

Any number of relayer processes may be safely run in parallel (and indeed, it is expected that separate relayers will serve separate subsets of the interchain). However, they may consume unnecessary fees if they submit the same proof multiple times, so some minimal coordination may be ideal (such as assigning particular relayers to particular packets or scanning mempools for pending transactions).

12 ICS 020 - Fungible Token Transfer

12.1 Synopsis

This standard document specifies packet data structure, state machine handling logic, and encoding details for the transfer of fungible tokens over an IBC channel between two modules on separate chains. The state machine logic presented allows for safe multi-chain denomination handling with permissionless channel opening. This logic constitutes a "fungible token transfer bridge module", interfacing between the IBC routing module and an existing asset tracking module on the host state machine.

12.1.1 Motivation

Users of a set of chains connected over the IBC protocol might wish to utilise an asset issued on one chain on another chain, perhaps to make use of additional features such as exchange or privacy protection, while retaining fungibility with the original asset on the issuing chain. This application-layer standard describes a protocol for transferring fungible tokens between chains connected with IBC which preserves asset fungibility, preserves asset ownership, limits the impact of Byzantine faults, and requires no additional permissioning.

12.1.2 Definitions

The IBC handler interface & IBC routing module interface are as defined in ICS 25 and ICS 26, respectively.

12.1.3 Desired Properties

- Preservation of fungibility (two-way peg).
- Preservation of total supply (constant or inflationary on a single source chain & module).
- Permissionless token transfers, no need to whitelist connections, modules, or denominations.
- Symmetric (all chains implement the same logic, no in-protocol differentiation of hubs & zones).
- Fault containment: prevents Byzantine-inflation of tokens originating on chain `A`, as a result of chain `B`'s Byzantine behaviour (though any users who sent tokens to chain `B` may be at risk).

12.2 Technical Specification

12.2.1 Data Structures

Only one packet data type, `FungibleTokenPacketData`, which specifies the denomination, amount, sending account, receiving account, and whether the sending chain is the source of the asset, is required.

```
interface FungibleTokenPacketData {
  denomination: string
  amount: uint256
  sender: string
  receiver: string
  source: boolean
}
```

The fungible token transfer bridge module tracks escrow addresses associated with particular channels in state. Fields of the `ModuleState` are assumed to be in scope.

```
interface ModuleState {
  channelEscrowAddresses: Map<Identifier, string>
}
```

12.2.2 Sub-protocols

The sub-protocols described herein should be implemented in a "fungible token transfer bridge" module with access to a bank module and to the IBC routing module.

Port & channel setup The `setup` function must be called exactly once when the module is created (perhaps when the blockchain itself is initialised) to bind to the appropriate port and create an escrow address (owned by the module).

```
function setup() {
  routingModule.bindPort("bank", ModuleCallbacks{
    onChanOpenInit,
    onChanOpenTry,
    onChanOpenAck,
    onChanOpenConfirm,
    onChanCloseInit,
    onChanCloseConfirm,
    onRecvPacket,
    onTimeoutPacket,
    onAcknowledgePacket,
    onTimeoutPacketClose
  })
}
```

Once the `setup` function has been called, channels can be created through the IBC routing module between instances of the fungible token transfer module on separate chains.

An administrator (with the permissions to create connections & channels on the host state machine) is responsible for setting up connections to other state machines & creating channels to other instances of this module (or another module supporting this interface) on other chains. This specification defines packet handling semantics only, and defines them in such a fashion that the module itself doesn't need to worry about what connections or channels might or might not exist at any point in time.

Routing module callbacks

Channel lifecycle management Both machines `A` and `B` accept new channels from any module on another machine, if and only if:

- The other module is bound to the "bank" port.
- The channel being created is unordered.
- The version string is empty.

```
function onChanOpenInit(
  order: ChannelOrder,
  connectionHops: [Identifier],
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  counterpartyPortIdentifier: Identifier,
  counterpartyChannelIdentifier: Identifier,
  version: string) {
  // only unordered channels allowed
  abortTransactionUnless(order === UNORDERED)
  // only allow channels to "bank" port on counterparty chain
  abortTransactionUnless(counterpartyPortIdentifier === "bank")
  // version not used at present
  abortTransactionUnless(version === "")
  // allocate an escrow address
  channelEscrowAddresses[channelIdentifier] = newAddress()
}
```

```
function onChanOpenTry(
  order: ChannelOrder,
  connectionHops: [Identifier],
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  counterpartyPortIdentifier: Identifier,
  counterpartyChannelIdentifier: Identifier,
  version: string,
  counterpartyVersion: string) {
  // only unordered channels allowed
  abortTransactionUnless(order === UNORDERED)
  // version not used at present
  abortTransactionUnless(version === "")
  abortTransactionUnless(counterpartyVersion === "")
  // only allow channels to "bank" port on counterparty chain
  abortTransactionUnless(counterpartyPortIdentifier === "bank")
  // allocate an escrow address
  channelEscrowAddresses[channelIdentifier] = newAddress()
}
```

```
function onChanOpenAck(
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  version: string) {
  // version not used at present
  abortTransactionUnless(version === "")
  // port has already been validated
}
```

```
function onChanOpenConfirm(
  portIdentifier: Identifier,
  channelIdentifier: Identifier) {
  // accept channel confirmations, port has already been validated
}
```

```
function onChanCloseInit(
  portIdentifier: Identifier,
  channelIdentifier: Identifier) {
  // no action necessary
}
```

```
function onChanCloseConfirm(
  portIdentifier: Identifier,
  channelIdentifier: Identifier) {
  // no action necessary
}
```

Packet relay In plain English, between chains `A` and `B`:

- When acting as the source zone, the bridge module escrows an existing local asset denomination on the sending chain and mints vouchers on the receiving chain.
- When acting as the sink zone, the bridge module burns local vouchers on the sending chains and unescrows the local asset denomination on the receiving chain.
- When a packet times-out, local assets are unescrowed back to the sender or vouchers minted back to the sender appropriately.
- No acknowledgement data is necessary.

`createOutgoingPacket` must be called by a transaction handler in the module which performs appropriate signature checks, specific to the account owner on the host state machine.

```
function createOutgoingPacket(
  denomination: string,
  amount: uint256,
  sender: string,
  receiver: string,
  source: boolean) {
  if source {
    // sender is source chain: escrow tokens
    // determine escrow account
    escrowAccount = channelEscrowAddresses[packet.sourceChannel]
    // construct receiving denomination, check correctness
    prefix = "{packet/destPort}/{packet.destChannel}"
    abortTransactionUnless(denomination.slice(0, len(prefix)) === prefix)
    // escrow source tokens (assumed to fail if balance insufficient)
    bank.TransferCoins(sender, escrowAccount, denomination.slice(len(prefix)), amount)
  } else {
    // receiver is source chain, burn vouchers
    // construct receiving denomination, check correctness
    prefix = "{packet/sourcePort}/{packet.sourceChannel}"
    abortTransactionUnless(denomination.slice(0, len(prefix)) === prefix)
    // burn vouchers (assumed to fail if balance insufficient)
    bank.BurnCoins(sender, denomination, amount)
  }
  FungibleTokenPacketData data = FungibleTokenPacketData{denomination, amount, sender, receiver, source
      }
  handler.sendPacket(packet)
}
```

`onRecvPacket` is called by the routing module when a packet addressed to this module has been received.

```
function onRecvPacket(packet: Packet): bytes {
  FungibleTokenPacketData data = packet.data
  if data.source {
    // sender was source chain: mint vouchers
    // construct receiving denomination, check correctness
    prefix = "{packet/destPort}/{packet.destChannel}"
    abortTransactionUnless(data.denomination.slice(0, len(prefix)) === prefix)
    // mint vouchers to receiver (assumed to fail if balance insufficient)
    bank.MintCoins(data.receiver, data.denomination, data.amount)
  } else {
    // receiver is source chain: unescrow tokens
    // determine escrow account
    escrowAccount = channelEscrowAddresses[packet.destChannel]
    // construct receiving denomination, check correctness
    prefix = "{packet/sourcePort}/{packet.sourceChannel}"
    abortTransactionUnless(data.denomination.slice(0, len(prefix)) === prefix)
    // unescrow tokens to receiver (assumed to fail if balance insufficient)
    bank.TransferCoins(escrowAccount, data.receiver, data.denomination.slice(len(prefix)), data.amount)
  }
  return 0x
}
```

onAcknowledgePacket is called by the routing module when a packet sent by this module has been acknowledged.

```
function onAcknowledgePacket(
  packet: Packet,
  acknowledgement: bytes) {
  // nothing is necessary, likely this will never be called since it's a no-op
}
```

onTimeoutPacket is called by the routing module when a packet sent by this module has timed-out (such that it will not be received on the destination chain).

```
function onTimeoutPacket(packet: Packet) {
  FungibleTokenPacketData data = packet.data
  if data.source {
    // sender was source chain, unescrow tokens
    // determine escrow account
    escrowAccount = channelEscrowAddresses[packet.destChannel]
    // construct receiving denomination, check correctness
    prefix = "{packet/sourcePort}/{packet.sourceChannel}"
    abortTransactionUnless(data.denomination.slice(0, len(prefix)) === prefix)
    // unescrow tokens back to sender
    bank.TransferCoins(escrowAccount, data.sender, data.denomination.slice(len(prefix)), data.amount)
  } else {
    // receiver was source chain, mint vouchers
    // construct receiving denomination, check correctness
    prefix = "{packet/sourcePort}/{packet.sourceChannel}"
    abortTransactionUnless(data.denomination.slice(0, len(prefix)) === prefix)
    // mint vouchers back to sender
    bank.MintCoins(data.sender, data.denomination, data.amount)
  }
}
```

```
function onTimeoutPacketClose(packet: Packet) {
  // can't happen, only unordered channels allowed
}
```

Reasoning

Correctness This implementation preserves both fungibility & supply.

Fungibility: If tokens have been sent to the counterparty chain, they can be redeemed back in the same denomination & amount on the source chain.

Supply: Redefine supply as unlocked tokens. All send-recv pairs sum to net zero. Source chain can change supply.

Multi-chain notes This does not yet handle the "diamond problem", where a user sends a token originating on chain A to chain B, then to chain D, and wants to return it through D -> C -> A — since the supply is tracked as owned by chain B, chain C cannot serve as the intermediary. It is not yet clear whether that case should be dealt with in-protocol or not — it may be fine to just require the original path of redemption (and if there is frequent liquidity and some surplus on both paths the diamond path will work most of the time). Complexities arising from long redemption paths may lead to the emergence of central chains in the network topology.

Optional addenda

- Each chain, locally, could elect to keep a lookup table to use short, user-friendly local denominations in state which are translated to and from the longer denominations when sending and receiving packets.
- Additional restrictions may be imposed on which other machines may be connected to & which channels may be established.

13 ICS 027 - Interchain Accounts

13.1 Synopsis

This standard document specifies packet data structure, state machine handling logic, and encoding details for the account management system over an IBC channel between separate chains.

13.1.1 Motivation

On Ethereum, there are two types of accounts: externally owned accounts, controlled by private keys, and contract accounts, controlled by their contract code (ref). Similar to Ethereum's CA (contract accounts), interchain accounts are managed by another chain while retaining all the capabilities of a normal account (i.e. stake, send, vote, etc). While an Ethereum CA's contract logic is performed within Ethereum's EVM, interchain accounts are managed by another chain via IBC in a way such that the owner of the account retains full control over how it behaves.

13.1.2 Definitions

The IBC handler interface & IBC relayer module interface are as defined in ICS 25 and ICS 26, respectively.

13.1.3 Desired Properties

- Permissionless
- Fault containment: Interchain account must follow rules of its host chain, even in times of Byzantine behaviour by the counterparty chain (the chain that manages the account)
- The chain that controls the account must process the results asynchronously and according to the chain's logic. The result should be 0x0 if the transaction was successful and an error code other than 0x0 if the transaction failed.
- Sending and receiving transactions will be processed in an ordered channel where packets are delivered exactly in the order which they were sent.

13.2 Technical Specification

The implementation of interchain account is non-symmetric. This means that each chain can have a different way to generate an interchain account and deserialise the transaction bytes and a different set of transactions that they can execute. For example, chains that use the Cosmos SDK will deserialise tx bytes using Amino, but if the counterparty chain is a smart contract on Ethereum, it may deserialise tx bytes by an ABI that is a minimal serialisation algorithm for the smart contract. The interchain account specification defines the general way to register an interchain account and transfer tx bytes. The counterparty chain is responsible for deserialising and executing the tx bytes, and the sending chain should know how counterparty chain will handle the tx bytes in advance.

Each chain must satisfy following features to create a interchain account:

- New interchain accounts must not conflict with existing ones.
- Each chain must keep track of which counterparty chain created each new interchain account.

Also, each chain must know how the counterparty chains serialise/deserialise transaction bytes in order to send transactions via IBC. And the counterparty chain must implement the process of safely executing IBC transactions by verifying the authority of the transaction's signers.

The chain must reject the transaction and must not make a state transition in the following cases:

- The IBC transaction fails to be deserialised.
- The IBC transaction expects signers other than the interchain accounts made by the counterparty chain.

It does not restrict how you can distinguish signers that was not made by the counterparty chain. But the most common way would be to record the account in state when the interchain account is registered and to verify that signers are recorded interchain account.

13.2.1 Data Structures

Each chain must implement the below interfaces to support interchain account. `createOutgoingPacket` method in `IBCAccountModule` interface defines the way to create an outgoing packet for a specific type. Type indicates how IBC account transaction should be constructed and serialised for the host chain. Generally, type indicates what framework the host chain was built from. `generateAddress` defines the way how to determine the account's address by using identifier and salt. Using the salt to generate an address is recommended, but not required. If the chain doesn't support a deterministic way to generate an address with a salt, it can be generated by its own way. `createAccount` is used to create account with generated address. New interchain account must not conflict with existing ones, and chains should keep track of which counterparty chain created each new interchain account in order to verify the authority of transaction's signers in `authenticateTx`. `authenticateTx` validates a transaction and checks that the signers in the transaction have the right permissions. `runTx` executes a transaction after it was authenticated successfully.

```
type Tx = object

interface IBCAccountModule {
  createOutgoingPacket(chainType: Uint8Array, data: any)
  createAccount(address: Uint8Array)
  generateAddress(identifier: Identifier, salt: Uint8Array): Uint8Array
  deserialiseTx(txBytes: Uint8Array): Tx
  authenticateTx(tx: Tx): boolean
  runTx(tx: Tx): uint32
}
```

`RegisterIBCAccountPacketData` is used by the counterparty chain to register an account. An interchain account's address is defined deterministically with the channel identifier and salt. The `generateAccount` method is used to generate a new interchain account's address. It is recommended to generate address by `hash(identifier+salt)`, but other methods may be used. This function must generate a unique and deterministic address by utilising identifier and salt.

```
interface RegisterIBCAccountPacketData {
  salt: Uint8Array
}
```

`RunTxPacketData` is used to execute a transaction on an interchain account. The transaction bytes contain the transaction itself and are serialised in a manner appropriate for the destination chain.

```
interface RunTxPacketData {
  txBytes: Uint8Array
}
```

The `IBCAccountHandler` interface allows the source chain to receive results of executing transactions on an interchain account.

```
interface InterchainTxHandler {
  onAccountCreated(identifier: Identifier, address: Address)
  onTxSucceeded(identifier: Identifier, txBytes: Uint8Array)
  onTxFailed(identifier: Identifier, txBytes: Uint8Array, errorCode: Uint8Array)
}
```

13.2.2 Subprotocols

The subprotocols described herein should be implemented in a “interchain-account-bridge” module with access to a router and codec (decoder or unmarshaller) for the application and access to the IBC relayer module.

13.2.3 Port & channel setup

The `setup` function must be called exactly once when the module is created (perhaps when the blockchain itself is initialised) to bind to the appropriate port and create an escrow address (owned by the module).

```
function setup() {
  relayerModule.bindPort("interchain-account", ModuleCallbacks{
    onChanOpenInit,
    onChanOpenTry,
    onChanOpenAck,
    onChanOpenConfirm,
    onChanCloseInit,
    onChanCloseConfirm,
    onSendPacket,
    onRecvPacket,
    onTimeoutPacket,
    onAcknowledgePacket,
    onTimeoutPacketClose
  })
}
```

Once the `setup` function has been called, channels can be created through the IBC relayer module between instances of the interchain account module on separate chains.

An administrator (with the permissions to create connections & channels on the host state machine) is responsible for setting up connections to other state machines & creating channels to other instances of this module (or another module supporting this interface) on other chains. This specification defines packet handling semantics only, and defines them in such a fashion that the module itself doesn’t need to worry about what connections or channels might or might not exist at any point in time.

13.2.4 Routing module callbacks

13.2.5 Channel lifecycle management

Both machines `A` and `B` accept new channels from any module on another machine, if and only if:

- The other module is bound to the “interchain account” port.
- The channel being created is ordered.
- The version string is empty.

```
function onChanOpenInit(
  order: ChannelOrder,
  connectionHops: [Identifier],
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  counterpartyPortIdentifier: Identifier,
  counterpartyChannelIdentifier: Identifier,
  version: string) {
  // only ordered channels allowed
  abortTransactionUnless(order === ORDERED)
  // only allow channels to "interchain-account" port on counterparty chain
  abortTransactionUnless(counterpartyPortIdentifier === "interchain-account")
  // version not used at present
  abortTransactionUnless(version === "")
}
```

```
function onChanOpenTry(
  order: ChannelOrder,
  connectionHops: [Identifier],
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  counterpartyPortIdentifier: Identifier,
  counterpartyChannelIdentifier: Identifier,
  version: string,
  counterpartyVersion: string) {
  // only ordered channels allowed
  abortTransactionUnless(order === ORDERED)
  // version not used at present
  abortTransactionUnless(version === "")
  abortTransactionUnless(counterpartyVersion === "")
  // only allow channels to "interchain-account" port on counterparty chain
  abortTransactionUnless(counterpartyPortIdentifier === "interchain-account")
}
```

```
function onChanOpenAck(
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  version: string) {
  // version not used at present
  abortTransactionUnless(version === "")
  // port has already been validated
}
```

```
function onChanOpenConfirm(
  portIdentifier: Identifier,
  channelIdentifier: Identifier) {
  // accept channel confirmations, port has already been validated
}
```

```
function onChanCloseInit(
  portIdentifier: Identifier,
  channelIdentifier: Identifier) {
  // no action necessary
}
```

```
function onChanCloseConfirm(
  portIdentifier: Identifier,
  channelIdentifier: Identifier) {
  // no action necessary
}
```

13.2.6 Packet relay

In plain English, between chains `A` and `B`. It will describe only the case that chain A wants to register an Interchain account on chain B and control it. Moreover, this system can also be applied the other way around.

```
function onRecvPacket(packet: Packet): bytes {
  if (packet.data is RunTxPacketData) {
    const tx = deserialiseTx(packet.data.txBytes)
    abortTransactionUnless(authenticateTx(tx))
    return runTx(tx)
  }

  if (packet.data is RegisterIBCAccountPacketData) {
    RegisterIBCAccountPacketData data = packet.data
    identifier = "{packet/sourcePort}/{packet.sourceChannel}"
    const address = generateAddress(identifier, packet.salt)
    createAccount(address)
    // Return generated address.
    return address
  }

  return 0x
}
```

```
function onAcknowledgePacket(
  packet: Packet,
  acknowledgement: bytes) {
  if (packet.data is RegisterIBCAccountPacketData)
    if (acknowledgement !== 0x) {
      identifier = "{packet/sourcePort}/{packet.sourceChannel}"
      onAccountCreated(identifier, acknowledgement)
    }
  if (packet.data is RunTxPacketData) {
    identifier = "{packet/destPort}/{packet.destChannel}"
    if (acknowledgement === 0x)
```

```
        onTxSucceeded(identifier: Identifier, packet.data.txBytes)
    else
        onTxFailed(identifier: Identifier, packet.data.txBytes, acknowledgement)
  }
}
```

```
function onTimeoutPacket(packet: Packet) {
  // Receiving chain should handle this event as if the tx in packet has failed
  if (packet.data is RunTxPacketData) {
    identifier = "{packet/destPort}/{packet.destChannel}"
    // 0x99 error code means timeout.
    onTxFailed(identifier: Identifier, packet.data.txBytes, 0x99)
  }
}
```

```
function onTimeoutPacketClose(packet: Packet) {
  // nothing is necessary
}
```

14 ICS 006 - Solo Machine Client

14.1 Synopsis

This specification document describes a client (verification algorithm) for a solo machine with a single updateable public key which implements the ICS 2 interface.

14.1.1 Motivation

Solo machines — which might be devices such as phones, browsers, or laptops — might like to interface with other machines & replicated ledgers which speak IBC, and they can do so through the uniform client interface.

14.1.2 Definitions

Functions & terms are as defined in ICS 2.

14.1.3 Desired Properties

This specification must satisfy the client interface defined in ICS 2.

Conceptually, we assume "big table of signatures in the universe" - that signatures produced are public - and incorporate replay protection accordingly.

14.2 Technical Specification

This specification contains implementations for all of the functions defined by ICS 2.

14.2.1 Client state

The `ClientState` of a solo machine is simply whether or not the client is frozen.

```
interface ClientState {
  frozen: boolean
}
```

14.2.2 Consensus state

The `ConsensusState` of a solo machine consists of the current public key & sequence number.

```
interface ConsensusState {
  sequence: uint64
  publicKey: PublicKey
}
```

14.2.3 Headers

`Header`s must only be provided by a solo machine when the machine wishes to update the public key.

```
interface Header {
  sequence: uint64
  signature: Signature
  newPublicKey: PublicKey
}
```

14.2.4 Evidence

`Evidence` of solo machine misbehaviour consists of a sequence and two signatures over different messages at that sequence.

```
interface Evidence {
  sequence: uint64
  signatureOne: Signature
  signatureTwo: Signature
}
```

14.2.5 Client initialisation

The solo machine client `initialise` function starts an unfrozen client with the initial consensus state.

```
function initialise(consensusState: ConsensusState): ClientState {
  return {
    frozen: false,
    consensusState
  }
}
```

14.2.6 Validity predicate

The solo machine client `checkValidityAndUpdateState` function checks that the currently registered public key has signed over the new public key with the correct sequence.

```
function checkValidityAndUpdateState(
  clientState: ClientState,
  header: Header) {
  assert(sequence === clientState.consensusState.sequence)
  assert(checkSignature(header.newPublicKey, header.sequence, header.signature))
  clientState.consensusState.publicKey = header.newPublicKey
  clientState.consensusState.sequence++
}
```

14.2.7 Misbehaviour predicate

Any duplicate signature on different messages by the current public key freezes a solo machine client.

```
function checkMisbehaviourAndUpdateState(
  clientState: ClientState,
  evidence: Evidence) {
    h1 = evidence.h1
    h2 = evidence.h2
    pubkey = clientState.consensusState.publicKey
    assert(evidence.h1.signature.data !== evidence.h2.signature.data)
    assert(checkSignature(pubkey, evidence.sequence, evidence.h1.signature))
    assert(checkSignature(pubkey, evidence.sequence, evidence.h2.signature))
    clientState.frozen = true
}
```

14.2.8 State verification functions

All solo machine client state verification functions simply check a signature, which must be provided by the solo machine.

```
function verifyClientConsensusState(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  clientIdentifier: Identifier,
  consensusState: ConsensusState) {
    path = applyPrefix(prefix, "clients/{clientIdentifier}/consensusState")
    abortTransactionUnless(!clientState.frozen)
    value = clientState.consensusState.sequence + path + consensusState
    assert(checkSignature(clientState.consensusState.pubKey, value, proof))
    clientState.consensusState.sequence++
}

function verifyConnectionState(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  connectionIdentifier: Identifier,
  connectionEnd: ConnectionEnd) {
    path = applyPrefix(prefix, "connection/{connectionIdentifier}")
    abortTransactionUnless(!clientState.frozen)
    value = clientState.consensusState.sequence + path + connectionEnd
    assert(checkSignature(clientState.consensusState.pubKey, value, proof))
    clientState.consensusState.sequence++
}

function verifyChannelState(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  channelEnd: ChannelEnd) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}")
    abortTransactionUnless(!clientState.frozen)
    value = clientState.consensusState.sequence + path + channelEnd
    assert(checkSignature(clientState.consensusState.pubKey, value, proof))
    clientState.consensusState.sequence++
}

function verifyPacketCommitment(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64,
  commitment: bytes) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}/packets/{sequence}"
        )
    abortTransactionUnless(!clientState.frozen)
    value = clientState.consensusState.sequence + path + commitment
    assert(checkSignature(clientState.consensusState.pubKey, value, proof))
    clientState.consensusState.sequence++
}

function verifyPacketAcknowledgement(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64,
  acknowledgement: bytes) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}/acknowledgements/{
         sequence}")
    abortTransactionUnless(!clientState.frozen)
    value = clientState.consensusState.sequence + path + acknowledgement
    assert(checkSignature(clientState.consensusState.pubKey, value, proof))
    clientState.consensusState.sequence++
}

function verifyPacketAcknowledgementAbsence(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}/acknowledgements/{
         sequence}")
    abortTransactionUnless(!clientState.frozen)
    value = clientState.consensusState.sequence + path
    assert(checkSignature(clientState.consensusState.pubKey, value, proof))
    clientState.consensusState.sequence++
}

function verifyNextSequenceRecv(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  nextSequenceRecv: uint64) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}/nextSequenceRecv")
    abortTransactionUnless(!clientState.frozen)
    value = clientState.consensusState.sequence + path + nextSequenceRecv
    assert(checkSignature(clientState.consensusState.pubKey, value, proof))
    clientState.consensusState.sequence++
}
```

14.2.9 Properties & Invariants

Instantiates the interface defined in ICS 2.

15 ICS 007 - Tendermint Client

15.1 Synopsis

This specification document describes a client (verification algorithm) for a blockchain using Tendermint consensus.

15.1.1 Motivation

State machines of various sorts replicated using the Tendermint consensus algorithm might like to interface with other replicated state machines or solo machines over IBC.

15.1.2 Definitions

Functions & terms are as defined in ICS 2.

The Tendermint light client uses the generalised Merkle proof format as defined in ICS 8.

15.1.3 Desired Properties

This specification must satisfy the client interface defined in ICS 2.

15.2 Technical Specification

This specification depends on correct instantiation of the Tendermint consensus algorithm and light client algorithm.

15.2.1 Client state

The Tendermint client state tracks the current validator set, latest height, and a possible frozen height.

```
interface ClientState {
  validatorSet: List<Pair<Address, uint64>>
  latestHeight: uint64
  frozenHeight: Maybe<uint64>
}
```

15.2.2 Consensus state

The Tendermint client tracks the validator set hash & commitment root for all previously verified consensus states (these can be pruned after awhile).

```
interface ConsensusState {
  validatorSetHash: []byte
  commitmentRoot: []byte
}
```

15.2.3 Headers

The Tendermint client headers include a height, the commitment root, the complete validator set, and the signatures by the validators who committed the block.

```
interface Header {
  height: uint64
  commitmentRoot: []byte
  validatorSet: List<Pair<Address, uint64>>
  signatures: []Signature
}
```

15.2.4 Evidence

The `Evidence` type is used for detecting misbehaviour and freezing the client - to prevent further packet flow - if applicable. Tendermint client `Evidence` consists of two headers at the same height both of which the light client would have considered valid.

```
interface Evidence {
  fromValidatorSet: List<Pair<Address, uint64>>
  fromHeight: uint64
  h1: Header
  h2: Header
}
```

15.2.5 Client initialisation

Tendermint client initialisation requires a (subjectively chosen) latest consensus state, including the full validator set.

```
function initialize(consensusState: ConsensusState, validatorSet: List<Pair<Address, uint64>>,
    latestHeight: uint64): ClientState {
  return ClientState{
    validatorSet,
    latestHeight,
    pastHeaders: Map.singleton(latestHeight, consensusState)
  }
}
```

15.2.6 Validity predicate

Tendermint client validity checking uses the bisection algorithm described in the Tendermint spec. If the provided header is valid, the client state is updated & the newly verified commitment written to the store.

```
function checkValidityAndUpdateState(
  clientState: ClientState,
  header: Header) {
    // assert that header is newer than any we know
    assert(header.height < clientState.latestHeight)
    // call the `verify` function
    assert(verify(clientState.validatorSet, clientState.latestHeight, header))
    // update latest height
    clientState.latestHeight = header.height
    // create recorded consensus state, save it
    consensusState = ConsensusState{validatorSet.hash(), header.commitmentRoot}
    set("consensusStates/{identifier}/{header.height}", consensusState)
    // save the client
    set("clients/{identifier}", clientState)
}
```

15.2.7 Misbehaviour predicate

Tendermint client misbehaviour checking determines whether or not two conflicting headers at the same height would have convinced the light client.

```
function checkMisbehaviourAndUpdateState(
  clientState: ClientState,
  evidence: Evidence) {
    // assert that the heights are the same
    assert(h1.height === h2.height)
    // assert that the commitments are different
    assert(h1.commitmentRoot !== h2.commitmentRoot)
    // fetch the previously verified commitment root & validator set hash
    consensusState = get("consensusStates/{identifier}/{evidence.fromHeight}")
    // check that the validator set matches
    assert(consensusState.validatorSetHash === evidence.fromValidatorSet.hash())
    // check if the light client "would have been fooled"
    assert(
      verify(evidence.fromValidatorSet, evidence.fromHeight, h1) &&
      verify(evidence.fromValidatorSet, evidence.fromHeight, h2)
      )
    // set the frozen height
    clientState.frozenHeight = min(h1.height, h2.height)
    // save the client
    set("clients/{identifier}", clientState)
}
```

15.2.8 State verification functions

Tendermint client state verification functions check a Merkle proof against a previously validated commitment root.

```
function verifyClientConsensusState(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  clientIdentifier: Identifier,
  consensusState: ConsensusState) {
    path = applyPrefix(prefix, "consensusStates/{clientIdentifier}")
    // check that the client is at a sufficient height
    assert(clientState.latestHeight >= height)
    // check that the client is unfrozen or frozen at a higher height
    assert(clientState.frozenHeight === null || clientState.frozenHeight > height)
    // fetch the previously verified commitment root & verify membership
    root = get("consensusStates/{identifier}/{height}")
    // verify that the provided consensus state has been stored
    assert(root.verifyMembership(path, consensusState, proof))
}

function verifyConnectionState(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  connectionIdentifier: Identifier,
  connectionEnd: ConnectionEnd) {
    path = applyPrefix(prefix, "connection/{connectionIdentifier}")
    // check that the client is at a sufficient height
    assert(clientState.latestHeight >= height)
    // check that the client is unfrozen or frozen at a higher height
    assert(clientState.frozenHeight === null || clientState.frozenHeight > height)
    // fetch the previously verified commitment root & verify membership
    root = get("consensusStates/{identifier}/{height}")
    // verify that the provided connection end has been stored
    assert(root.verifyMembership(path, connectionEnd, proof))
}

function verifyChannelState(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  channelEnd: ChannelEnd) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}")
    // check that the client is at a sufficient height
    assert(clientState.latestHeight >= height)
    // check that the client is unfrozen or frozen at a higher height
    assert(clientState.frozenHeight === null || clientState.frozenHeight > height)
    // fetch the previously verified commitment root & verify membership
    root = get("consensusStates/{identifier}/{height}")
    // verify that the provided channel end has been stored
    assert(root.verifyMembership(path, channelEnd, proof))
}

function verifyPacketCommitment(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64,
  commitment: bytes) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}/packets/{sequence}"
         )
    // check that the client is at a sufficient height
    assert(clientState.latestHeight >= height)
    // check that the client is unfrozen or frozen at a higher height
    assert(clientState.frozenHeight === null || clientState.frozenHeight > height)
    // fetch the previously verified commitment root & verify membership
    root = get("consensusStates/{identifier}/{height}")
    // verify that the provided commitment has been stored
    assert(root.verifyMembership(path, commitment, proof))
}

function verifyPacketAcknowledgement(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64,
  acknowledgement: bytes) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}/acknowledgements/{
         sequence}")
    // check that the client is at a sufficient height
    assert(clientState.latestHeight >= height)
    // check that the client is unfrozen or frozen at a higher height
    assert(clientState.frozenHeight === null || clientState.frozenHeight > height)
    // fetch the previously verified commitment root & verify membership
    root = get("consensusStates/{identifier}/{height}")
    // verify that the provided acknowledgement has been stored
    assert(root.verifyMembership(path, acknowledgement, proof))
}

function verifyPacketAcknowledgementAbsence(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  sequence: uint64) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}/acknowledgements/{
         sequence}")
    // check that the client is at a sufficient height
    assert(clientState.latestHeight >= height)
    // check that the client is unfrozen or frozen at a higher height
    assert(clientState.frozenHeight === null || clientState.frozenHeight > height)
    // fetch the previously verified commitment root & verify membership
    root = get("consensusStates/{identifier}/{height}")
    // verify that no acknowledgement has been stored
    assert(root.verifyNonMembership(path, proof))
}

function verifyNextSequenceRecv(
  clientState: ClientState,
  height: uint64,
  prefix: CommitmentPrefix,
  proof: CommitmentProof,
  portIdentifier: Identifier,
  channelIdentifier: Identifier,
  nextSequenceRecv: uint64) {
    path = applyPrefix(prefix, "ports/{portIdentifier}/channels/{channelIdentifier}/nextSequenceRecv")
    // check that the client is at a sufficient height
    assert(clientState.latestHeight >= height)
    // check that the client is unfrozen or frozen at a higher height
    assert(clientState.frozenHeight === null || clientState.frozenHeight > height)
    // fetch the previously verified commitment root & verify membership
    root = get("consensusStates/{identifier}/{height}")
    // verify that the nextSequenceRecv is as claimed
    assert(root.verifyMembership(path, nextSequenceRecv, proof))
}
```

15.2.9 Properties & Invariants

Correctness guarantees as provided by the Tendermint light client algorithm.

16 Appendix A: Use-case Descriptions

16.1 Asset transfer

Wherever compatible native asset representations exist, IBC can be used to transfer assets between two chains.

16.1.1 Fungible tokens

IBC can be used to transfer fungible tokens between chains.

Representations Bitcoin `UTXO`, Ethereum `ERC20`, Cosmos SDK `sdk.Coins`.

Implementation Two chains elect to “peg” two semantically compatible fungible token denominations to each other, escrowing, unescrowing, minting, and burning as necessary when sending & handling IBC packets.

There may be a starting “source zone”, which starts with the entire token balance, and “target zone”, which starts with zero token balance, or two zones may both start off with nonzero balances of a token (perhaps originated on a third zone), or two zones may elect to combine the supply and render fungible two previously disparate tokens.

Invariants Fungibility of any amount across all pegged representations, constant (or formulaic, in the case of a inflationary asset) total supply cumulative across chains, and tokens only exist in a spendable form on one chain at a time.

16.1.2 Non-fungible tokens

IBC can be used to transfer non-fungible tokens between chains.

Representations Ethereum `ERC721`, Cosmos SDK `sdk.NFT`.

Implementation Two chains elect to “peg” two semantically compatible non-fungible token namespaces to each other, escrowing, unescrowing, creating, and destroying as necessary when sending & handling IBC packets.

There may be a starting “source zone” which starts with particular tokens and contains token-associated logic (e.g. breeding CryptoKitties, redeeming digital ticket), or the associated logic may be packaged along with the NFT in a format which all involved chains can understand.

Invariants Any given non-fungible token exists uniquely on one chain, owned by a particular account, at any point in time, and can always be transferred back to the “source” zone to perform associated actions (e.g. breeding a CryptoKitty) if applicable.

16.1.3 Involved zones

Vanilla payments A “vanilla payments” zone, such as the Cosmos Hub, may allow incoming & outgoing fungible and/or non-fungible token transfers through IBC. Users might elect to keep assets on such a zone due to high security or high connectivity.

Shielded payments A “shielded payments” zone, such as the Zcash blockchain (pending UITs), may allow incoming & outgoing fungible and/or non-fungible token transfers through IBC. Tokens which are transferred to such a zone could then be shielded through the zero-knowledge circuit and held, transferred, traded, etc. Once users had accomplished their anonymity-requiring purposes, they could be transferred out and back over IBC to other zones.

Decentralised exchange A “decentralised exchange” zone may allow incoming & outgoing fungible and/or non-fungible token transfers through IBC, and allow tokens stored on that zone to be traded with each other through a decentralised exchange protocol in the style of Uniswap or 0x (or future such protocols).

Decentralised finance A “decentralised finance” zone, such as the Ethereum blockchain, may allow incoming & outgoing fungible and/or non-fungible token transfers though IBC, and allow tokens stored on that zone to interact with a variety of decentralised financial products: synthetic stablecoins, collateralised loans, liquidity pools, etc.

16.2 Multichain contracts

IBC can be used to pass messages & data between contracts with logic split across several chains.

16.2.1 Cross-chain contract calls

IBC can be used to execute arbitrary contract-to-contract calls between separate smart contract platform chains, with calldata and return data.

Representations Contracts: Ethereum `EVM`, `WASM` (various), Tezos `Michelson`, Agoric `Jessie`.

Calldata: Ethereum `ABI`, generic serialisation formats such as RLP, Protobuf, or JSON.

Implementation A contract on one zone which intends to call a contract on another zone must serialise the calldata and address of the destination contract in an IBC packet, which can be relayed through an IBC connection to the IBC handler on the destination chain, which will call the specified contract, executing any associated logic, and return the result of the call (if applicable) back in a second IBC packet to the calling contract, which will need to handle it asynchronously.

Implementing chains may elect to provide a "channel" object to contract developers, with a send end, receive end, configurable buffer size, etc. much like channels in multiprocess concurrent programming in languages such as Go or Haskell.

Invariants Contract-dependent.

16.2.2 Cross-chain fee payment

Representations Same as "fungible tokens" as above.

Implementation An account holding assets on one chain can be used to pay fees on another chain by sending tokens to an account on the first chain controlled by the validator set of the second chain and including a proof that tokens were so sent (on the first chain) in the transaction submitted to the second chain.

The funds can be periodically send back over the IBC connection from the first chain to the second chain for fee disbursement.

Invariants Correct fees paid on one of two chains but not both.

16.2.3 Interchain collateralisation

A subset of the validator set on one chain can elect to validate another chain and be held accountable for equivocation faults committed on that chain submitted over an IBC connection, and the second chain can delegate its validator update logic to the first chain through the same IBC connection.

Representations ABCI `Evidence` and `ValidatorUpdate`.

Implementation `ValidatorUpdate`s for a participating subset of the primary (collateralising) chain's validator set are relayed in IBC packets to the collateralised chain, which uses them directly to set its own validator set.

`Evidence` of any equivocations is relayed back from the collateralised chain to the primary chain so that the equivocating validator(s) can be slashed.

Invariants Validators which commit an equivocation fault are slashable on at least one chain, and possibly the validator set of a collateralised chain is bound to the validator set of a primary (collateralising) chain.

16.3 Sharding

IBC can be used to migrate smart contracts & data between blockchains with mutually comprehensible virtual machines & data formats, respectively.

16.3.1 Code migration

Representations Same as "cross-chain contract calls" above, with the additional requirement that all involved code be serialisable and mutually comprehensible (executable) by the involved chains.

Implementation Participating chains migrate contracts, which they can all execute, between themselves according to a known balancing ("sharding") algorithm, perhaps designed to equalise load or achieve efficient locality for frequently-interacting contracts.

A routing system on top of core IBC will be required to correctly route cross-chain contract calls between contracts which may frequently switch chains.

Invariants Semantics of code preserved, namespacing preserved by some sort of routing system.

16.3.2 Data migration

IBC can be used to implement an arbitrary-depth multi-chain "cache" system where storage cost can be traded for access cost.

Representations Generic serialisation formats, such as Amino, RLP, Protobuf, JSON.

Implementation An arbitrary-depth IBC-connection-linked-list of chains, with the first chain optimised for compute and later chains optimised for cheaper storage, can implement a hierarchical cache, where data unused for a period of time on any chain is migrated to the next chain in the list. When data is necessary (e.g. for a contract call or storage access), if it is not stored on the chain looking it up, it must be relayed over an IBC packet back to that chain (which can then re-cache it for some period).

Invariants All data can be accessed on the primary (compute) chain when requested, with a known bound of necessary IBC hops.

17 Appendix B: Design Patterns

17.1 Verification instead of computation

Computation on distributed ledgers is expensive: any computations performed in the IBC handler must be replicated across all full nodes. Therefore, when it is possible to merely *verify* a computational result instead of performing the computation, the IBC handler should elect to do so and require extra parameters as necessary.

In some cases, there is no cost difference - adding two numbers and checking that two numbers sum to a particular value both require one addition, so the IBC handler should elect to do whatever is simpler. However, in other cases, performing the computation may be much more expensive. For example, connection and channel identifiers must be uniquely generated. This could be implemented by the IBC handler hashing the genesis state plus a nonce when a new channel is created, to create a pseudorandom identifier - but that requires computing a hash function on-chain, which is expensive. Instead, the IBC handler should require that the random identifier generation be performed off-chain and merely check that a new channel creation attempt doesn't use a previously reserved identifier.

17.2 Call receiver

Essential to the functionality of the IBC handler is an interface to other modules running on the same machine, so that it can accept requests to send packets and can route incoming packets to modules. This interface should be as minimal as possible in order to reduce implementation complexity and requirements imposed on host state machines.

For this reason, the core IBC logic uses a receive-only call pattern that differs slightly from the intuitive dataflow. As one might expect, modules call into the IBC handler to create connections, channels, and send packets. However, instead of the IBC handler, upon receipt of a packet from another chain, selecting and calling into the appropriate module, the module itself must call `recvPacket` on the IBC handler (likewise for accepting channel creation handshakes). When `recvPacket` is called, the IBC handler will check that the calling module is authorised to receive and process the packet (based on included proofs and known state of connections / channels), perform appropriate state updates (incrementing sequence numbers to prevent replay), and return control to the module or throw on error. The IBC handler never calls into modules directly.

Although a bit counterintuitive to reason about at first, this pattern has a few notable advantages:

- It minimises requirements of the host state machine, since the IBC handler need not understand how to call into other modules or store any references to them.
- It avoids the necessity of managing a module lookup table in the handler state.
- It avoids the necessity of dealing with module return data or failures. If a module does not want to receive a packet (perhaps having implemented additional authorisation on top), it simply never calls `recvPacket`. If the routing logic were implemented in the IBC handler, the handler would need to deal with the failure of the module, which is tricky to interpret.

It also has one notable disadvantage:

- Without an additional abstraction, the relayer logic becomes more complex, since off-chain relayer processes will need to track the state of multiple modules to determine when packets can be submitted.

For this reason, there is an additional IBC “routing module” which exposes a call dispatch interface.

17.3 Call dispatch

For common relay patterns, an “IBC routing module” can be implemented which maintains a module dispatch table and simplifies the job of relayers.

In the call dispatch pattern, datagrams (contained within transaction types defined by the host state machine) are relayed directly to the routing module, which then looks up the appropriate module (owning the channel & port to which the datagram was addressed) and calls an appropriate function (which must have been previously registered with the routing module). This allows modules to avoid handling datagrams directly, and makes it harder to accidentally screw-up the atomic state transition execution which must happen in conjunction with sending or receiving a packet (since the module never handles packets directly, but rather exposes functions which are called by the routing module upon receipt of a valid packet).

Additionally, the routing module can implement default logic for handshake datagram handling (accepting incoming handshakes on behalf of modules), which is convenient for modules which do not need to implement their own custom logic.

18 Appendix C: Canonical Encoding

18.0.1 Primitive types

If a value has a primitive type, it is encoded without tags.

Numbers The protocol deals only with unsigned integers.

`uint32` and `uint64` types are encoded as fixed-size little-endian, with no sign bit.

Booleans Boolean values are encoded as single bits: `0x00` (false) and `0x01` (true).

Bytes Byte arrays are encoded as-is with no length prefix or tag.

18.0.2 Structured types

Structured types with fields are encoded as proto3 `message`s with the appropriate fields.

Canonical `.proto` files are provided with the specification.

19 Appendix D: Frequently-Asked Questions

19.1 Forks & unbonding periods

What happens to all of the established IBC channels if a chain forks?

This depends on the light client algorithm. Tendermint light clients, at the moment, will halt the channel completely if a fork is detected (since it looks like equivocation) - if the fork doesn't use any sort of replay protection (e.g. change the chain ID). If one fork keeps the chain ID and the other picks a new one, the one which keeps it would be followed by the light client. If both forks change the chain ID (or validator set), they would both need new light clients.

What happens after the unbonding period passes without an IBC packet to renew the channel? Are the escrowed tokens un-recoverable without intervention?

By default, the tokens are un-recoverable. Governance intervention could alter the light client associated with the channel (there is no way to automate this that is safe). That said, it's always possible to construct light clients with different validation rules or to add the ability for a government proposal to reset the light client to a trusted header if it was previously valid and used, and if it was frozen due to the unbonding period.

19.2 Data flow & packet relay

Does Blockchain A need to know the address of a trustworthy node for Blockchain B in order to send IBC packets?

Blockchain A will know of the existence of Blockchain B after a kind of handshake takes place. This handshake is facilitated by a relayer. It is the responsibility of the relayer to access an available node of the corresponding blockchain to begin the handshake. The blockchains themselves need not know about nodes, just be able to access the transactions that are relayed between them.