ipsa: IBC v2 for Stellar

Subject. How ipsa connects Stellar to the IBC network, how each chain verifies the other's consensus in on-chain code, and what has been proven against live network data so far.

2026-11-03

© 2026 Amanda Gonsalves.

Contents

1. Executive Summary

- The problem
- What we are building
- Why this is possible now
- What it unlocks
- How this differs from a bridge
- The security model
- Evidence it works
- Where it stands

2. Questions and objections

- What is the use case?
- Does this move liquidity off Stellar?
- Why not integrate an existing bridge?
- Is IBC a poor fit for Stellar?
- What evidence is there that the verification works?
- What would have to fail for funds to be lost?
- Who operates the relayer?
- What is the delivery risk?

3. Technical Deep Dive

- 3.1 Design goals
- 3.2 System overview
- 3.3 Trust model and trust boundaries
- 3.4 Evidence model
- 3.5 How Stellar's consensus is verified
- 3.6 SCP to ledger, in detail
- 3.7 Ledger to application state
- 3.8 The Stellar IBC light client
- 3.9 Verifying the counterparty on Stellar
- 3.10 IBC v2 integration

© 2026 Amanda Gonsalves.

3.11 Relayer and historical evidence

3.12 Asset transfer model

3.13 A transfer, end to end

3.14 Failure and recovery

3.15 Security analysis

3.16 Performance

3.17 The Ethereum corridor

4. Implementation and Validation

4.1 What is built

4.2 The shape of the codebase

4.3 Mainnet verification results

4.4 Test vectors

4.5 Negative tests

4.6 End-to-end proof

5. Sources

© 2026 Amanda Gonsalves.

Subject. How ipsa connects Stellar to the IBC network, how each chain verifies the other's consensus in on-chain code, and what has been proven against live network data so far.

Updated 12 October 2026.

Two sources of truth run through this document. For anything about how Stellar's consensus works, it is the Stellar Consensus Protocol whitepaper (Mazières, draft of 25 February 2016), cited as "§x". For anything about what this project does, it is the current ipsa implementation.

© 2026 Amanda Gonsalves.

1. Executive Summary

The problem

Stellar's connection to the rest of crypto is currently capped by the weakest intermediary in the path, and for a network whose whole premise is that trust should be named, scoped and revocable, that is the wrong shape.

Stellar settles payments in five seconds for a fraction of a cent, and it holds real value doing it: roughly $820M in USDC, over $2B in tokenized real-world assets, and $5.5B in stablecoin payment volume in Q1 2026 alone. Almost all of that value can only move within Stellar.

Getting it out today means going through a custodial bridge or a signing committee: a permissioned group that watches one chain and vouches for what happened on the other. That group becomes the real security of every transfer through it, no matter how secure the two chains are. It is also the single most exploited design in the industry. Bridge hacks account for more than $2.5B in losses, and in 2022 they were roughly 70% of all crypto theft. The three largest, Ronin at $625M, Poly at $611M and Wormhole at $326M, were all the same failure: compromise the committee, take the funds.

What we are building

ipsa implements IBC v2 for Stellar natively. Two pieces:

- Soroban contracts implementing the interchain standards, so Stellar can send and receive IBC packets as a first-class member of the protocol rather than through an adapter. Among them are light clients of the counterparty chains, so Stellar verifies the other side as well.
- Stellar light clients that other chains load in order to verify Stellar for themselves. Cosmos chains load one compiled to wasm. EVM chains run the same verification inside a zero-knowledge program and check a proof of it in Solidity.

Two corridors run on Stellar testnet today, through a single router: one to the Cosmos Hub's public testnet, `provider`, and one to Ethereum Sepolia. The Hub loads the same Stellar light client any `08-wasm` chain would, stored there from a published, reproducible release.

There is no operator, committee or attestation service anywhere in the trust path. Each chain runs on-chain code that checks the other chain's consensus signatures directly. The relayer that carries the bytes has no authority at all: if it lies, the proofs fail; if it disappears, packets expire and funds return to the sender.

Why this is possible now

IBC was effectively closed to chains outside its original family for years. The protocol was too heavy to implement elsewhere, and joining meant convincing another chain to fork its own binary in order to accept you.

© 2026 Amanda Gonsalves.

Two changes removed that. IBC v2 collapsed the old multi-step connection and channel handshakes into a single counterparty registration and cut the provable state from eight paths to three, which is what makes the protocol small enough to run inside Soroban's storage and gas budget. And the `08-wasm` client module lets a host chain load a light client as data, with no fork and no coordinated upgrade on its side.

The same reasoning reaches chains with no wasm host. A zero-knowledge virtual machine can run the Stellar verification off-chain and hand the EVM a constant-size proof that it ran correctly, and Soroban's Protocol 28 BN254 host functions let a Stellar contract check that kind of proof in return. Neither side has to reimplement the other's consensus in its own VM.

This is the first moment Stellar has been technically able to join IBC as a full member rather than as a wrapped asset on someone else's bridge, and one of the first opportunities for any chain outside the protocol's original family to do so.

What it unlocks

Reach, immediately and then automatically. IBC connects more than 115 chains today. It has been in production since April 2021 with no protocol-level exploit, secures roughly $40B a year, and settles a median transfer in about 19 seconds. Connecting Stellar once reaches all of it, and every chain that joins IBC afterwards becomes reachable from Stellar at close to zero marginal cost, because the protocol layer, the light client and the relayer are all shared. A bridge connects two chains. A protocol implementation connects Stellar to a graph, including the parts of it that do not exist yet.

A two-way asset street. Stellar's regulated stablecoins, tokenized RWAs and non-USD fiat tokens become tradeable, lendable and usable as collateral elsewhere without wrapping through several bridges first. In the other direction, assets from those chains reach Stellar's anchor network, cash-out points and remittance corridors. Stellar arrives as a supplier of asset classes the graph is thin on, not just another consumer of USDC.

A claim that survives scrutiny. Every alternative route out of Stellar today relocates the trusted party rather than removing it, which means the honest version of their pitch is "trust our operator set." That claim cannot be checked by the person relying on it. This one can, in three specific ways. The evidence is public, because Stellar's consensus messages and ledger headers are published in the history archives and anyone can fetch the same bytes the light client consumes. The verifier is open and runnable, so a sceptic can point it at a ledger and watch each check pass or fail rather than taking a diagram on faith. And the trusted set is enumerable: there is exactly one assumption that a contract cannot discharge, it is written down in section 3.15, and it has a mitigation and an owner rather than a footnote. A claim survives scrutiny when the reviewer can falsify it and fails to.

Proof that Soroban carries systems work. A deterministic sparse Merkle tree maintained on-chain, ICS-23 proof verification, and consensus verification contracts are not application logic. Shipping them is an argument about the platform, made in code.

A path beyond asset transfer. Token transfer is the first application on the packet layer, not the ceiling, and IBC v2 makes that unusually cheap to exploit. Under v1 an application was bound to a channel at handshake time, so every new use case meant new plumbing. V2 carries the source and destination ports inside the

© 2026 Amanda Gonsalves.

packet payload, so a single client pairing already carries packets for any number of applications, including ones not yet written. Nothing about the trust model changes when a second application appears; it rides the verification already built.

That opens three directions in ascending order of ambition. Interchain accounts (ICS-27) let a contract or account on another chain control an account on Stellar, so a strategy running elsewhere can hold and move Stellar-native assets without a custodian in between. Cross-chain queries (ICS-31) let a contract read verified state from another chain, which is what makes a Soroban contract able to price or settle against facts it did not observe locally. And because the packet layer is symmetric, Soroban becomes a callable destination: an application on another chain can trigger Stellar execution and receive the acknowledgement, rather than only sending assets and hoping something on the far side picks them up.

The practical consequence for planning is that the expensive part is being paid once. The light client, the relayer, the proof plumbing and the operational work are shared by every application that follows, so the second and third product on this rail cost a fraction of the first.

How this differs from a bridge

Route	Who verifies a transfer	What you are trusting
ipsa (IBC)	The destination chain's on-chain light client of Stellar	Stellar's consensus and the destination chain's consensus. Nothing else.
Committee bridges	A validator set, DVN quorum or oracle network in the middle	An honest majority of a third party that neither chain controls

That difference is the product. It is why a compromised relayer costs a delay rather than the float, and why adding the next chain does not mean auditing another operator set.

The security model

One rule shapes the whole system: no component holds bridge funds, and no component vouches for events off-chain. The relayer, the gateway and the api are transport. They can slow a packet down or refuse to carry it. They cannot forge one. Neither off-chain service holds a signing key at all.

The hard engineering is proving Stellar's consensus to another chain, and it takes two separate proofs that are easy to mistake for one:

1. **A quorum of Stellar nodes agreed on a value for this ledger.** Stellar has no global validator list and no stake weighting, so there is no percentage to count to (§3.1). The client has to evaluate Stellar's own definition of agreement against a configuration it trusts.
2. **What they agreed on is this ledger, and this ledger commits to the data being proved.** SCP commits to a transaction-set hash, a close time and upgrades. It does not commit to the state root, the transaction results, or even the previous ledger's hash.

© 2026 Amanda Gonsalves.

Valid signatures on their own say nothing about which ledger or which state root arrived alongside them. A design that treats those two proofs as one ends up doing real cryptography and proving nothing useful. Section 3.5 walks through the eight checks that keep them apart, in plain language.

Evidence it works

The verification chain is implemented on-chain, and it has been run against live mainnet and testnet data by a standalone checker, `ipsa-scp-verifier`, that runs the same `stellar-consensus-verifier` crate the light clients use against public history-archive bytes. Anyone can reproduce it without deploying anything.

What was checked	Result
Signatures, quorum evaluation, agreement, ledger binding	Confirmed on real archived ledgers: 21 signers under the nested mainnet configuration, 3 under a flat testnet one
The ledger commits to its transaction results	Reconstructed for 64 of 64 mainnet ledgers in a checkpoint
The result commits to the contract state root	Reconstructed for 40 of 40 Soroban invocations
Each check rejects bad input rather than passing it	25-case negative suite, one mutation per case, all fail closed
The quorum predicate matches stellar-core	2,000 generated cases run through stellar-core's own code, agreeing case for case
An EVM contract accepts a proof of a mainnet ledger	A Groth16 proof of mainnet ledger 63,907,880 verified through a real SP1 verifier, and a header update landed on Sepolia for 316,910 gas
Packets cross both corridors	ICS-20 transfers delivered and acknowledged in both directions with the Cosmos Hub's public testnet and with Ethereum Sepolia, and an error acknowledgement from Sepolia refunded on Stellar

None of this required a change to `stellar-core`. Stellar already publishes everything the verification needs, which means this works against the network as it is, with no protocol change to negotiate and no dependency on the core team's roadmap.

Where it stands

Every light client in both corridors is implemented on-chain. The off-chain services assemble the consensus evidence, and the trust root is pinned by fingerprint rather than accepted from the network: the EVM client stores it in its client state and the proof program checks it on every update, the CLI refuses to create a Stellar client without one, and the standalone checker refuses to verify a ledger without one.

One Soroban router on Stellar testnet serves the Cosmos Hub's public testnet, a local Cosmos devnet and Ethereum Sepolia. Packets flow end to end in both directions with the Hub and Sepolia. All of them are relayed by ipsa's fork of the Cosmos IBC v2 relayer and acknowledged back to their source chain.

© 2026 Amanda Gonsalves.

Everything that runs is released. The contracts, the light client, the verifier crates and the service images each come from a tagged release whose workflow runs the repository's own checks first; the contract release also deploys what it built, and the light client is built twice from a cold cache and published with its checksum. The off-chain services, the api, the gateway, the relayer and the proof API, run hosted on AWS rather than on a developer's machine. Section 4 sets out exactly what has been proven and how.

Ahead of a production deployment:

- measuring the CosmWasm gas of the wasm light client against a host chain's limits;
- hardening the client lifecycle on both inbound clients;
- building timeouts on the Ethereum corridor, and deciding who pays for proving;
- a third-party audit of the consensus verifier before any value moves.

The quorum configuration the client trusts is a deliberate, reviewable setting and needs a named owner.

Stellar and IBC network figures are July 2026, from DeFiLlama, Map of Zones, the IBC Protocol documentation and SDF's Q1 2026 report.

© 2026 Amanda Gonsalves.

2. Questions and objections

The objections below are the ones we consider fair, and they are answered here rather than after the technical material. Each is stated at its strongest, and where a criticism holds it is conceded rather than argued around.

What is the use case?

The fair version of the criticism is that IBC is a capability rather than a use case. That is correct, so a single use case is offered here.

Stellar is the only chain carrying tokenized non-USD currency at any meaningful scale: naira, reais, pesos and shillings, issued by regional anchors and used for payments. The remainder of the IBC graph is almost entirely dollar-denominated and holds very little of this.

The significance is a question of market structure. A stablecoin that cannot be traded has limited utility. For a market to function, a participant has to hold inventory and quote prices, and holding a regional currency means carrying the risk that it moves against them. Without somewhere to offset that risk, market makers either quote poor prices or decline to participate, the market stays thin, issuance stays small, and the anchor cannot grow.

The IBC graph provides what is missing: deep spot liquidity and derivatives venues where that risk can be hedged. Stellar provides the assets. Neither side can develop the market independently, and no alternative pairing substitutes, because no other chain issues these currencies.

Does this move liquidity off Stellar?

The technical answer is that the asset never leaves, because ICS-20 escrows it on Stellar and mints a claim on the destination chain. That is accurate as far as it goes. The tokens remain locked in a Soroban contract, and issuance, custody, reserves and redemption all stay on Stellar. It is also not what the question is really asking.

What does move is trading activity. If a regional-currency pair trades on another chain's venue, that venue earns the fees and holds the depth. That is a genuine cost, and there are three reasons it is worth accepting.

First, the alternative is not that the pair trades on Stellar instead. It is that it does not trade at all. Stellar's DEX handles roughly $87M a month across all pairs, within which regional-currency markets are negligible. This does not divert an existing liquid market; it enables one that does not currently exist. If hedging becomes possible, the anchor can issue more, which expands the Stellar-side business.

Second, Stellar's economics do not rest on DEX fees. Roughly $5.5B of quarterly stablecoin payment volume compares with $87M a month of DEX volume. Issuance, settlement and the on and off ramps are where the value is captured, and all three remain on Stellar.

Third, assets already leave through less secure routes. Committee-based bridges operate on Stellar today. The question has never been whether Stellar's assets reach other chains, but whether they do so through a light client or through an operator set.

© 2026 Amanda Gonsalves.

Why not integrate an existing bridge?

Because they address a different problem.

Under a committee bridge, an external validator set observes Stellar and reports to the destination chain, and the security of the transfer rests on that set. Under IBC, the destination chain runs a light client of Stellar and verifies the proof itself, so there is no third party to trust and none to compromise. The distinction can appear academic until one observes that essentially every large bridge failure has been a compromise of that intermediate layer rather than a flaw in either chain.

The stronger argument is regulatory. A tokenized-treasury issuer or a licensed anchor operates under legal opinions governing who may hold the asset. A committee bridge introduces a custodian that those issuers have not vetted and frequently cannot accept. For them, the absence of a trusted intermediary is not a preference but the only structure that clears compliance.

There is also a difference in what Stellar acquires. Integrating a bridge provides one additional route and a lasting dependency on a commercial operator. Implementing IBC makes Stellar a chain that more than a hundred existing networks can connect to without requiring anyone's permission, including ours.

Is IBC a poor fit for Stellar?

A reasonable prior, and the question that has occupied most of this year.

The specification does not presuppose any particular consensus mechanism. It defines a client interface, of which Tendermint is one implementation, and the `08-wasm` module exists precisely so that a chain can host a light client for a consensus family that was not anticipated.

The difficulty is nonetheless real, and section 3 is largely concerned with it. A Tendermint chain provides a hash chain in which each header commits to its predecessor, and the header carries the application state root directly. Stellar's consensus provides neither. There is no global validator set to count, and the value the network agrees upon does not commit to the ledger's state. Consequently the client must reproduce Stellar's own definition of agreement, and then construct the binding from that agreement to a specific ledger and from that ledger to the state being proved, none of which a same-family integration has to do.

The conclusion is that this is harder than a same-family integration but not a poor fit. It is a different one, and the difference is now implemented and verified against live mainnet data.

What evidence is there that the verification works?

It has been executed against the live network, and it has been shown to reject invalid input rather than merely to accept valid input.

Section 4 sets out the detail. In summary, the checker runs the contract's own code against real archived consensus messages and ledger headers; the two links binding a ledger to contract state reconstructed correctly for 64 of 64 mainnet ledgers in a checkpoint and 40 of 40 Soroban invocations; and a 25-case suite

© 2026 Amanda Gonsalves.

takes verified input, alters exactly one element per case, and confirms that the corresponding check refuses it. The quorum predicate is additionally checked against stellar-core's own code, compiled from a real stellar-core tree and run over 2,000 generated cases.

The second property matters more than the first. A verifier that accepts correct input may not be doing anything at all. One that demonstrably refuses tampered signatures, incorrect network identifiers, replayed signers, forged quorum sets and mismatched headers is performing verification.

What would have to fail for funds to be lost?

Three things, of which only one is plausible.

Forging consensus evidence would require forging Ed25519 signatures for an entire quorum of Stellar validators, or finding a hash collision at one of the bindings in the chain of custody. Neither is a practical concern.

Compromising the relayer, the gateway or the api would produce delay or censorship rather than loss. None of them holds a signing key, none can produce a proof that verifies, and packets that fail to arrive time out and refund the sender.

The plausible failure is misconfiguration of the quorum set the client trusts. If that is set incorrectly, the client faithfully follows a different view of the network while every signature verifies correctly. It cannot be validated from within a contract. Today it is pinned by fingerprint so that no service can substitute an alternative: the EVM client carries the fingerprint in its client state and its proof program checks it on every update, and the CLI refuses to create a Stellar client while no fingerprint is configured. What is not yet built is the off-chain validation of the configuration itself, using SDF's quorum analysis tooling to confirm the trusted set has quorum intersection, and monitoring for drift once it is deployed. It requires a named owner. This is the principal risk, and it is organisational rather than cryptographic.

The Ethereum corridor adds one assumption of a different kind: that the SP1 proving system and its Groth16 verifier are sound. Section 3.17 covers it.

Who operates the relayer?

Anyone may, and relaying confers no authority, so the question is economic rather than architectural. A relayer pays transaction fees and receives no privileged position. If no one relays, packets stop moving and pending ones refund on timeout: nothing is lost and nothing becomes stuck.

That said, an assumption that someone will volunteer is not an operations plan. A production deployment requires at least one relayer under monitoring, and the archive publication cadence that supplies consensus evidence establishes a latency floor that product decisions need to accommodate.

The Ethereum corridor adds a second cost: every client update in either direction needs a zero-knowledge proof, bought from a prover network. A Stellar header proof cost 0.3651 PROVE in October 2026.

© 2026 Amanda Gonsalves.

What is the delivery risk?

The commit history shows a single author, so the question is fair and is better addressed here. The working arrangement is less exposed than that history suggests: one engineer writes the code, and three reviewers experienced with Stellar review it, so the design decisions and the security-critical paths have had more than one set of eyes on them throughout.

Several further factors reduce the exposure. The project is not being built in isolation, and has support from Cosmos Labs and the Cardano Foundation, whose work the relayer and light-client patterns build on. The protocol is a public standard rather than a private design, so a second engineer inherits a specification rather than having to reconstruct someone's intent. The consensus verifier exists as a standalone crate that can be read, run and audited independently of the rest of the system. The test suite and its negative cases encode the invariants that matter, so a regression fails visibly. And the planned third-party audit places external review on the security-critical paths regardless.

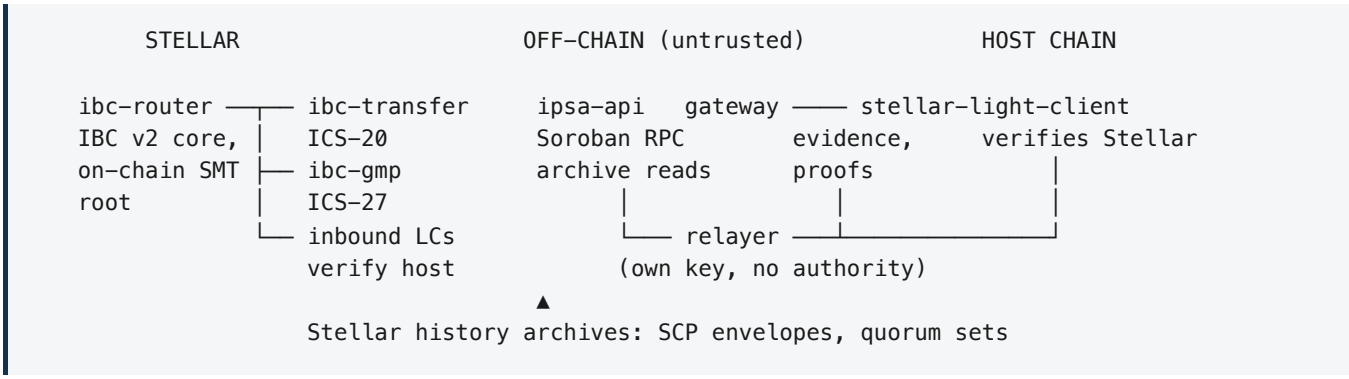
© 2026 Amanda Gonsalves.

3. Technical Deep Dive

3.1 Design goals

1. Nothing off-chain is ever believed. Every cross-chain claim is checked by on-chain code against evidence that proves its own authenticity.
2. Fail closed. Missing or malformed evidence rejects the header. The system never falls back to trusting whoever delivered it.
3. Evidence, not assertion. Off-chain components hand over bytes that carry their own signatures.
4. Stay standard. Deviate from IBC only where Stellar's consensus or execution model forces it, and write down each deviation.
5. Stay portable. The Stellar light client is one wasm artifact that any `08-wasm` host can load.

3.2 System overview



The Ethereum corridor has the same shape with one more untrusted service. The relayer asks a proof API for a zero-knowledge proof of the Stellar verification, and Sepolia's `SP1ICSStellar` contract checks that proof instead of running the verification itself. In the other direction, the proof API proves Ethereum's sync committee and storage, and Soroban's `sp1-ethereum` client checks the proof. Section 3.17 describes both.

© 2026 Amanda Gonsalves.

3.3 Trust model and trust boundaries

Component	Holds funds	Holds keys	Trusted for correctness
`ibc-router` and the light clients (on-chain)	escrow only	no	Yes. This is the root of all verification.
`ipsa-api`	no	no	No. It builds unsigned transactions and submits pre-signed ones.
`ipsa-gateway`	no	no	No. It supplies evidence that the client verifies for itself.
relayer	no	its own fee key	No. It can delay or censor, never forge.
proof API	no	a prover-network account	No. A wrong proof fails verification on-chain.
history archives	no	no	No. Every envelope carries its own signature.

The gateway is the piece exposed to the relayer's network surface, and it is built so that it cannot move value even if fully compromised: no signing key, no chain connection of its own. Every call it serves goes through the api. Transactions follow prepare, sign, submit, and whoever originates a transaction is the one who signs it.

There is exactly one boundary in the Cosmos corridor that is not cryptographic, and that is the quorum configuration the client trusts. No contract can check it from the inside. Section 3.15 treats it as the safety-critical setting it is.

3.4 Evidence model

Everything the light client consumes proves its own authenticity. SCP envelopes carry Ed25519 signatures made by named validators over exact statement bytes. Quorum sets are checked against the hashes the statements committed to. Ledger headers are checked by rehashing them.

That property is what makes the archive a file server rather than a trusted party. Any host serving the same bytes is equivalent, and a host serving different bytes gets caught by the first check.

An RPC response is a different kind of thing. It is an assertion by whoever answered the call, with no signature behind it. This design never treats one as evidence.

3.5 How Stellar's consensus is verified

This is the part of the system with no existing reference implementation to copy, so it is worth walking through slowly. The rest of section 3 assumes you have read this one.

© 2026 Amanda Gonsalves.

What the client is actually trying to prove

A contract on another chain is about to release funds because someone told it "this packet was committed on Stellar." Before it acts, it needs to be convinced of two things:

- Stellar's validators really did agree on a ledger at this position in the chain, and
- the ledger they agreed on is the one containing this packet.

Those sound like one statement. They are not, and keeping them apart is the whole design.

Picture the evidence arriving as two envelopes. The first holds 21 signed messages from Stellar validators saying "we agree." The second holds a ledger. Every signature in the first envelope can be genuine, and the ledger in the second can still be a complete fabrication, because nothing in those signatures mentions it. The signatures commit to a transaction-set hash and a timestamp. They do not commit to the ledger's state, its results, or its position relative to the ledger before it.

So the client proves the two things separately, then binds them together. Eight checks, in order. Each one answers a question a sceptic would ask, and any failure rejects the header outright.

The eight checks

1. Which validators are we even willing to listen to?

The client is configured with a quorum configuration: the set of validators it trusts and the structure describing how many of them, in what groupings, constitute agreement. Because that membership changes over time, the client stores a history of configurations, each tagged with the lowest ledger it governs, and picks the one that applies to the ledger being verified. Without this, an old ledger could be judged by validators who were not running the network when it closed.

2. Did these messages really come from the validators they name?

Each message carries an Ed25519 signature. The client checks it against the node id inside the message, over exactly the bytes that were signed: `networkID || ENVELOPE_TYPE_SCP || xdr(SCPStatement)`, where `networkID` is the SHA-256 of the network passphrase. Getting the network id wrong makes every signature fail, which is the point of including it: a testnet signature cannot be replayed on mainnet.

3. Are these the right kind of message, about the right ledger?

Stellar validators send several kinds of consensus message as they converge. Only one of them means "this is settled": `EXTERNALIZE`. The client accepts nothing else, and requires that every message names the ledger being verified, and that no validator appears twice.

The reason for accepting only `EXTERNALIZE` is a real theorem, not a preference. A node externalizes a value when it *confirms* a commit (§6.2), and Theorem 9 says two nodes outside the ill-behaved set cannot confirm contradictory statements, provided the network still has quorum intersection once those ill-behaved nodes are discounted. There is a weaker result one step earlier, Theorem 8, covering messages that merely *accept* a statement. A verifier cannot lean on it, and §5.4 explains why: accepting means nothing if the accepting node is itself compromised, and other honest nodes may never be able to accept the same thing. Confirming is the first moment at which anyone may safely act.

© 2026 Amanda Gonsalves.

Rejecting duplicate signers matters more than it looks. Without that check, one signature replayed twenty times would satisfy any threshold.

4. Is each validator's declared quorum set genuine?

Every `EXTERNALIZE` message contains a hash of the sending validator's own quorum set. The full sets travel alongside the messages, and the client hashes each one and matches it to the message that committed to it. A set that does not match its hash is rejected, so a validator cannot claim a more permissive configuration than the one it signed.

Each set is also sanity-checked the way `stellar-core` does: the threshold has to be at least 1, no greater than the number of entries, and not below the set's own blocking size; between 1 and 1000 validators; no duplicates.

5. Do these signers actually constitute agreement?

This is the check that has no equivalent on other chains, and the one most likely to be done wrong.

On a chain with a global validator set and voting power, "more than two thirds signed" is a well-defined sentence and you count to it. Stellar does not work that way. Every node publishes the groups of peers it would need to hear from, those groups nest, and agreement is defined relative to those published choices. There is no percentage to count to.

The whitepaper's definition (§3.1) is:

> A set of nodes `U` is a *quorum* if it is not empty and it contains a slice for each of its own members: for every `v` in `U`, some group that `v` would accept lies entirely inside `U`.

The implementation computes exactly that, by elimination. Start with everyone who signed. Repeatedly drop any signer whose own declared quorum set is not satisfied by whoever is left. When the set stops shrinking, what remains is a quorum in the whitepaper's sense. The client then requires that its own configured trust root has a group inside those survivors, which is what turns "a quorum somewhere in the network" into "a quorum that convinces me."

To make this concrete: on mainnet at ledger 63907880 in August 2026, the structure was 5 of 7 organisations, with each organisation satisfied by 2 of its 3 validators, and 21 signatures for the single ledger. A flat "21 signatures is enough" check would be wrong in both directions. It can pass a set of 21 that is not a quorum, and reject a smaller set that is one.

There is a second trap here worth naming. Stellar has a notion of a *blocking* set, which is enough to make a node accept something. It is not enough to confirm it, and §5.4.1 gives the sharp counterexample: if every node's only group is the entire network, then every node is blocking for every other, so any single node could convince any other node of anything at all.

6. Did they all agree on the same thing?

Every counted signer has to carry a byte-identical committed value. Ballot counters are deliberately ignored: the whitepaper defines two ballots as *compatible* when their values match (§6.2), and different counters simply mean some nodes retried after a timeout, which is normal. Requiring equal counters would be stricter

© 2026 Amanda Gonsalves.

than the theorem and would cost liveness for no security gain.

At this point the client knows consensus happened. It still does not know which ledger.

7. Is the value they agreed on this ledger?

The agreed value has to be byte-identical to the candidate ledger header's own `scpValue` field, and the header's sequence number has to match the position every message named. Then the client hashes the header to get the ledger hash.

That covers part of the header. It does not cover the rest, because SCP commits only to the transaction-set hash, the close time and upgrades. The state root, the transaction results and the previous ledger hash are all outside the signed value.

So the client reaches for the *next* ledger. Slot N+1's agreed value names a transaction set; that set hashes to the named value; and inside that set is a field pointing back at ledger N's full hash. A quorum agreeing on N+1 therefore authenticates the parts of ledger N that N's own signatures left uncovered.

This step is the weakest link in the chain and should be described as such. It relies on a `stellar-core` rule, that validators check the previous-ledger pointer before voting, rather than on anything the whitepaper guarantees. A node lacking the transaction set can return "maybe valid," and maybe-valid values are votable. The precise claim is: a quorum externalized N+1, and its honest members had fetched the transaction set.

8. Does this ledger commit to the data being proved?

Covered in full in section 3.7, because it is where Stellar differs most from the chains IBC grew up on.

What this adds up to

Checks 1 through 6 establish that consensus happened. Checks 7 and 8 establish which ledger and which state. Neither half implies the other, and a client that skips the second half is the failure mode described at the top of this section.

3.6 SCP to ledger, in detail

The mechanics behind checks 6 and 7.

`commit.value` is an XDR-encoded `StellarValue` holding `txSetHash`, `closeTime` and `upgrades`. The client decodes it, then requires that the candidate header's own `scpValue` field is byte-identical to it. Not equivalent, not semantically equal: the same bytes. That is what makes this header the one the network agreed on rather than one presented alongside unrelated signatures.

`ledgerSeq` in the header has to equal the slot index carried by every statement. In Stellar the SCP slot index *is* the ledger sequence number, which is another reason `EXTERNALIZE` is the only usable message type: it is the one that carries the slot.

The ledger hash is `SHA-256(xdr(LedgerHeader))` over the header alone. The archive wraps headers in a `LedgerHeaderHistoryEntry` with a recorded hash in front and an extension behind, and including either of those produces a hash that matches nothing. Where a recorded hash is available the client compares against

© 2026 Amanda Gonsalves.

it, but never accepts it as input.

For the next-slot binding: `sha256(xdr(GeneralizedTransactionSet))` for slot N+1 must equal the `txSetHash` inside N+1's agreed value, and that transaction set's `previousLedgerHash` must equal ledger N's hash.

On success the client stores a consensus state of `{ timestamp (from closeTime), ledger_hash, root }`, where the root is present only if the next section succeeded.

3.7 Ledger to application state

Stellar's `bucketListHash` commits to all ledger state, but the bucket list hashes whole sorted files. There is no short inclusion proof for a single entry. So the IBC state root cannot be proved as state.

It is proved instead as a *transaction result*, which the header does commit to:

```
header.txSetResultHash == sha256(xdr(TransactionResultSet))
  -> the result pair at result_index
    -> InvokeHostFunctionResult::Success(h)
      -> h == sha256(xdr(InvokeHostFunctionSuccessPreImage))
        -> a ContractEvent in that preimage whose contractID is the router
          -> the event's data: 32 bytes, the SMT root
```

Two design decisions in there are load-bearing.

The root is read from an event, not from the return value. `InvokeHostFunctionSuccessPreImage` is a pair of a return value and a list of events, and it does not record which contract was invoked. If the client took the root from the return value, it would accept a root from any contract at all: an attacker deploys their own, returns 32 bytes of their choosing, invokes it in the same ledger, and points the client at that result. `ContractEvent.contractID` is set by the host and cannot be forged by contract code, so the event is the only thing in the preimage that identifies its emitter. The client is configured with the router's contract id and the event topic, and reads the root only from the router's own event. Exactly one such event may appear, or the header is rejected as ambiguous.

The whole result set travels with the header. `txSetResultHash` is a flat hash rather than a Merkle root, so there is no logarithmic proof available. The caller supplies the set already split into per-pair blobs, and the client reassembles `u32be(len) || pair0 || pair1 || ...` and hashes it. The split is untrusted: a wrong one cannot reproduce the committed hash. The pair at `result_index` then has to decode exactly, with no trailing bytes, which pins its boundaries.

On the Stellar side, the root is correct by construction. `ibc-router` recomputes and publishes it on every provable write, committing once per invocation and publishing it as an `ibc_root` event, so a successful transaction implies a correctly derived root. Provable state is a fixed-depth-64 binary sparse Merkle tree covering exactly the three IBC v2 paths: packet commitment, receipt, acknowledgement. All counterparties share the one tree, which is capped at 512 live leaves. Packet proofs are ICS-23 commitment proofs checked against the root this chain authenticated.

Put together: section 3.7 gives "correctly derived," and 3.5 and 3.6 give "genuinely Stellar's, at ledger N."

© 2026 Amanda Gonsalves.

3.8 The Stellar IBC light client

`stellar-light-client`, running on the counterparty through ibc-go's `08-wasm` module. It is built reproducibly: a pinned Rust toolchain and a checksum-verified `wasm-opt`, with absolute paths and timestamps stripped. CI builds it twice from a cold cache and fails if the two artifacts differ, so a chain's governance can confirm that the binary it is asked to load matches the source.

Client state. Chain id, latest height, frozen height, the quorum configuration history, proof specs, network id, maximum consensus age, router contract id, the root event topic, and a generation counter. The router id and the event topic are what make the state-root binding possible; without them a header still verifies for consensus but carries no root.

Consensus state. Timestamp, ledger hash, root. A header that arrives without a state-root proof produces an empty root. That consensus state is still useful for timestamps and misbehaviour, but any membership proof against it fails with `StateRootNotBound`, which is the right outcome: no root was ever proved for that ledger.

CreateClient. The client is instantiated with a quorum configuration, a network id, the router id and the event topic. The trust root is the one input that cannot be delegated to transport. Quorum sets reach the CLI and the relayer through the gateway, which is untrusted, so serving them is a convenience and never an authority. The CLI compares them against fingerprints it is configured with, and refuses to create a client while that list is empty. The relayer carries the same check, `sha256(quorum_set_xdr)` against a configured constant, but it is not yet called on the relay path, so today the pin is enforced where clients are created rather than on every relay.

UpdateState. Runs the whole chain in 3.5 through 3.7 and stores the result. Re-submitting a header for a height that already has a consensus state is fine if it produces an identical one. Disagreement is treated as a conflict. An expired client refuses to advance at all: it has to be recovered deliberately rather than jumped forward, because the gap is exactly where a stale configuration would go unnoticed.

Misbehaviour. Two independent routes. Fork evidence is one quorum externalizing two different ledgers for the same slot, which Theorem 9 says cannot happen unless quorum intersection has been violated. The other is a verified header that contradicts a consensus state already stored. Both verify their inputs through the normal path *before* comparing anything; comparing unverified input against stored state would let anyone freeze a client by asserting a conflict that never happened. Freezing sets the frozen height, and the client reports `Frozen`. Fork evidence sent as an ordinary update is refused, so the two paths cannot be confused.

Expiry and trusting period. SCP's safety does not depend on message timing. §3.3 puts it as only termination, not safety, depending on it, so SCP finality itself needs no trusting period. That is a fact about the protocol, not permission for the client to have no time policy. A consensus state older than the configured maximum age is refused for proof verification, because a validator key compromised long after the fact could sign old ledgers and the client would have no way to find out. Status is `Frozen`, `Expired` or `Active`. The maximum age is a per-client parameter with no default, and a value of zero is rejected at creation, so a client cannot be instantiated with the check switched off.

Validator and quorum-set changes. Nothing in SCP tells a client who the validators are. The configuration comes from outside the protocol and is updated outside it. Because membership changes, the client keeps configurations as a history with validity ranges and selects the greatest `valid_from` that does not postdate

© 2026 Amanda Gonsalves.

the ledger. Rotation is a deliberate, reviewable change to both the client state and the relayer's pinned fingerprint. The whitepaper's Theorem 13 covers safety when configuration changes *within* a slot; across slots, which configuration applies is a client-side policy, and this design makes it explicit rather than implicit.

3.9 Verifying the counterparty on Stellar

Everything so far has been the outbound direction: another chain convincing itself about Stellar. The inbound direction is the mirror image, and it runs on a Soroban contract that verifies the counterparty's consensus before the router will accept a packet from it.

This half is far more familiar territory, and it is worth being clear about why. A Tendermint chain has one globally agreed validator set with voting power attached to each member, so "validators holding more than two thirds of the power signed this block" is a well-defined sentence that a contract can evaluate by counting. Better still, the block header commits to the application state root directly, in a field called the app hash. The two separate proofs that section 3.5 had to keep apart collapse into one here: verify the header, and you have the state root, because the header contains it.

What the contract checks

A client update carries a signed header, the validator set it claims signed it, and the height of the already-trusted consensus state it builds on. The contract works through six checks, and any failure rejects the update.

1. **The chain id matches** the one the client was configured with, and the header's height is strictly greater than the trusted height. A client never moves sideways or backwards.
2. **The header's validator-set hash equals the `next_validators_hash` stored in the trusted consensus state.** This is the link that chains one update to the last: the previous header said who would sign the next one, and this header has to be signed by exactly that set.
3. **The supplied validator set actually hashes to that value.** The set arrives as data from an untrusted relayer, so the contract rebuilds the Merkle root over each validator's public key and voting power and compares. A relayer cannot substitute a set of its own choosing.
4. **The header hashes to the block id the commit refers to.** Tendermint's header hash is a Merkle root over its fourteen fields, each encoded the way the reference implementation encodes it, so the contract rebuilds that root from the header it was handed and checks it against the hash the signatures actually committed to.
5. **Each precommit signature verifies**, as Ed25519 over the canonical vote. This is the fiddly part and it has to be byte-exact: the canonical vote is protobuf with the height and round written as fixed64 little-endian rather than varints, the block id, timestamp and chain id length-delimited, and the whole thing length-prefixed before signing. A single byte out of place makes every signature fail.
6. **The tallied voting power clears the threshold.** Validators without a precommit for this block are skipped rather than counted, and the comparison is done in integer arithmetic against the configured fraction, with no floating point anywhere near it.

If all six pass, the contract stores a new consensus state holding the header's timestamp, its `next_validators_hash` for the following update, and its app hash as the root. Packet proofs from that chain are then ICS-23 proofs checked against that root.

© 2026 Amanda Gonsalves.

The controls around the cryptography

A light client is defeated far more cheaply through an unguarded door than by attacking Ed25519, so the machinery around those six checks matters as much as the checks themselves.

Client updates chain. Because each header must be signed by the validator set the previous one named, an attacker cannot jump the client to an arbitrary height with a set of their own. They would have to produce a valid chain from a height the client already trusts.

Membership proofs are gated on client status. A frozen client verifies nothing, and a proof against a height with no stored consensus state fails rather than falling back to anything.

Misbehaviour freezes rather than rewrites. A header presented at a height that already has a consensus state, and disagreeing with it about the app hash, is treated as evidence that the counterparty equivocated. The response is to freeze the client rather than to overwrite the stored state, which halts the link and hands the decision to whoever operates it. Overwriting would amount to letting a submitter choose the root.

The trusting period is enforced on update. A consensus state older than the configured trusting period cannot be used as the basis for a new one, because a validator set that has since unbonded could sign an old header with nothing at stake.

Initialisation is the router's alone, and happens once. Only the router can initialise a client in the contract, and an initialisation for a client id that already exists is refused, so a live client's trusted state cannot be overwritten by a later call. An earlier build left both open.

Hardening still in progress

Being precise about the current state, because the cryptographic core and the surrounding policy are at different stages of completion.

The six checks above are real and complete: real Merkle roots over the header and the validator set, real canonical-vote construction, a real per-validator signature check, and a real power-weighted tally. What is not yet finished is the policy layer that reads the rest of the client state. The configured trust level is stored but the verifier is currently invoked with a two-thirds threshold directly, so a client instantiated with a stricter fraction would not get it. The unbonding period and maximum clock drift are likewise stored and not yet read, which means a header timestamped in the future is not rejected on those grounds. The trusting period is checked when the client updates but not when a proof is verified against an existing consensus state. The ICS-23 proof specs carried in the client state are not yet enforced during proof verification: the leaf and inner operations are fixed in code. Updates are sequential only, because a header is accepted only when its validator set is the one the trusted state named, so there is no skipping verification across a validator-set change yet. And the dedicated two-header misbehaviour message is defined but unused.

One control is in place that an earlier revision of this document listed as missing: the misbehaviour path runs a submitted header through the full six checks before comparing it with stored state, so freezing is gated on verified evidence, as it is in the outbound client.

None of these weaken the signature checking or the threshold tally. They are the difference between a client that verifies correctly and one whose security parameters and lifecycle controls are exactly what a deployment instantiated it with, and closing them is the client-lifecycle hardening named in section 1.

© 2026 Amanda Gonsalves.

One deployment-path item belongs here too. The mock client, whose proof verification returns true unconditionally, is no longer deployed or registered by the release workflow, which registers only the `07-tendermint` and `sepolia` client types. It is still built and attached to releases as an artifact alongside the attestation client, and removing both from the published artifacts is part of the same hardening pass.

3.10 IBC v2 integration

IBC v2 removes the connection and channel handshakes that v1 required. A pair of clients plus one `RegisterCounterparty` call on each side, binding a local client to its counterparty's client id and commitment prefix, is the entire setup. No connection object, no channel, no version negotiation, no port binding.

What is standard IBC here: the packet lifecycle (send, receive, acknowledge, timeout), the three provable paths, ICS-23 proofs, the ICS-20 payload as JSON `FungibleTokenPacketData` with `version = "ics20-1"`, and the `08-wasm` client interface (`VerifyClientMessage`, `UpdateState`, `VerifyMembership`, `VerifyNonMembership`, `CheckForMisbehaviour`). The host chain runs unmodified ibc-go.

What is Stellar-specific: only the inside of the light client, meaning the verification chain, the quorum configuration history and the state-root binding. Two consequences leak outward. Stellar's own provable keys carry no prefix, and the prefix used to address the counterparty's store is set per client when the counterparty is registered. And Soroban allows one `InvokeHostFunction` per transaction while `recv_packet` takes a single packet, so Stellar-bound batches are pinned to one packet until the router grows a batch entrypoint.

The router is indifferent to which chain is on the other side. A client type maps to a light client contract, and each client id carries its own light client, counterparty and prefix, so one router serves the Cosmos and Ethereum corridors side by side. The transfer app picks the payload encoding per client as well: JSON for the Cosmos chain, Solidity ABI for the EVM one.

Encodings that have to match the reference implementation byte for byte: the packet-commitment timeout is hashed big-endian, and a membership proof embeds the value hash, so the verifier compares `proof_value` against `sha256(value)`.

3.11 Relayer and historical evidence

SCP messages are not in the RPC stream. Soroban RPC carries ledger metadata. It does not carry SCP envelopes, and it has no endpoint for quorum-set preimages at all. Without both, there is nothing to verify. RPC can tell you what a ledger is; it cannot show you that the network agreed on it.

The archives carry exactly what is needed. Stellar's public history archives publish, per 64-ledger checkpoint, the SCP envelopes together with the quorum-set preimages, the ledger headers and the transaction sets. That is the evidence set sections 3.5 through 3.7 consume, and it is durable.

RPC retention is about a week, so even if RPC carried the envelopes it could not serve historical verification. Auditing an older ledger, or recovering a relayer after an outage longer than the window, has no RPC path even in principle.

© 2026 Amanda Gonsalves.

Cadence. Checkpoint publication puts a floor of roughly five and a half minutes on outbound proof latency, which the relayer accounts for when it picks packet timeouts. A non-validating watcher `stellar-core` supplies the same data in real time for deployments that need it. The contract does not care which source produced the bytes, because it verifies them either way.

Relaying is request-driven. Nothing scans the chains. A source transaction is handed to the api, which passes it to the relayer, and the relayer follows that packet through receive and acknowledgement.

The relayer is transport. It signs its own transactions with its own key and pays its own fees. It cannot forge a packet, mint, or steal. It can delay or censor, which is what timeouts and multiple relayers are for.

Recovery is re-fetching a checkpoint and resubmitting. SCP treats every slot independently (§6), so a client can be advanced to any ledger for which evidence exists. No sequential replay, no bisection, and gaps are legal.

3.12 Asset transfer model

`ibc-transfer` moves value over the SEP-41 token interface. A registered local asset is escrowed by transferring it to the transfer contract's own address using its canonical Stellar Asset Contract address, which covers native XLM and issued assets alike. There is no internal balance ledger.

Outbound, the escrowed asset appears on the host chain as an IBC voucher minted by that chain's transfer module. Inbound, a denom the contract does not recognise causes it to deploy a voucher token contract deterministically from the hash of its full traced denom and mint to the receiver. The voucher is a standard SEP-41 token contract whose only minter is the transfer app.

Sending debits: escrow for a local asset, burn for a voucher. Receiving credits: release from escrow for a local asset, mint for a voucher. A success acknowledgement settles, and a failure acknowledgement or a timeout refunds by the inverse operation. A voucher returning to its origin chain burns on the way out and releases from escrow on arrival, restoring the original asset to the sender.

Asset identity travels as the denom in the ICS-20 payload, with path prefixing as ICS-20 specifies: a denom arriving from a counterparty gains a `port/client/` prefix, and a denom returning home has its prefix stripped. The voucher address is a pure function of the traced denom, so identity is stable across hops and anyone can derive it independently.

The success acknowledgement is the ICS-20 JSON `{"result":"AQ=="}`, byte for byte. ibc-go compares it exactly, and an earlier single-byte acknowledgement left one devnet transfer that can never be acknowledged.

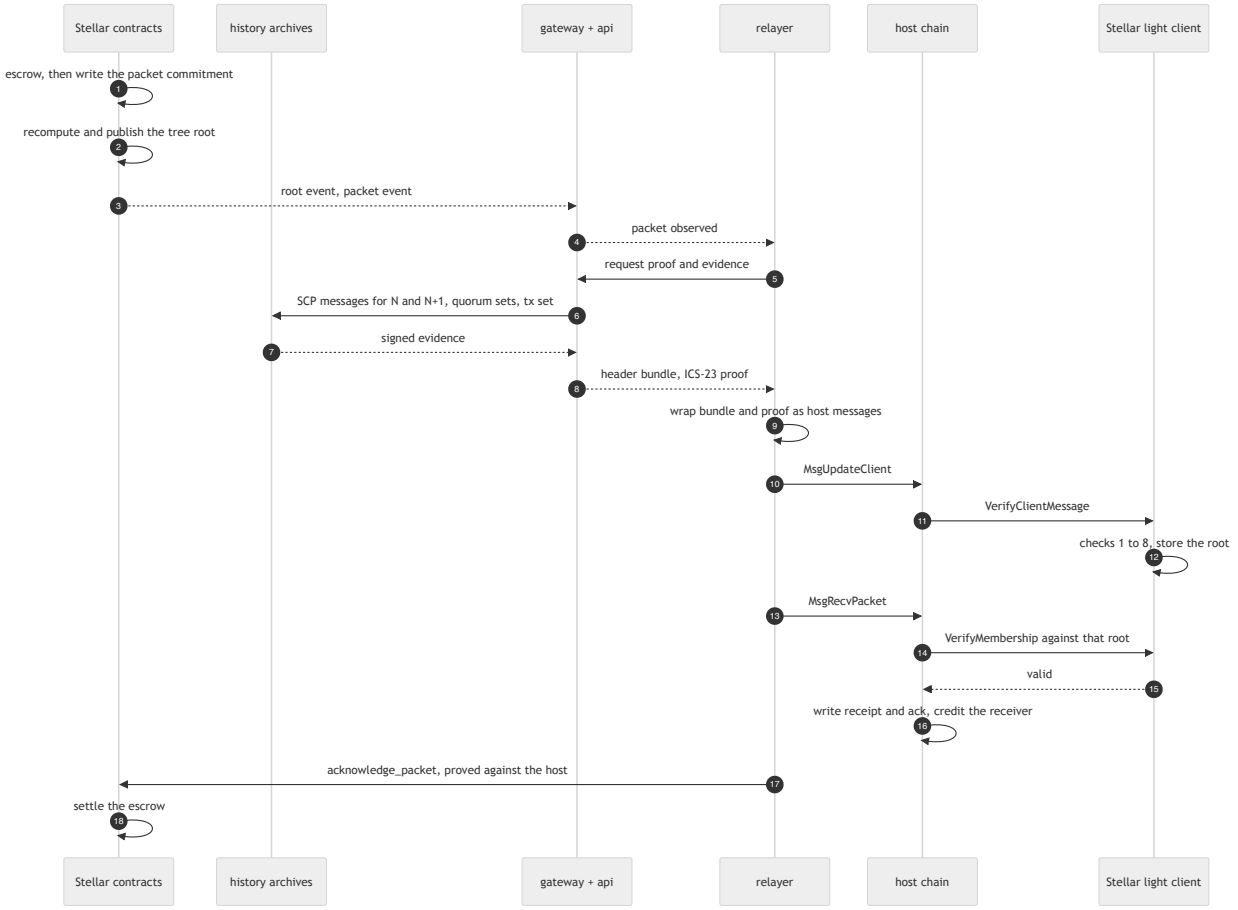
General message passing

`ibc-gmp` carries arbitrary contract calls on the same packet layer, under the port `gmpport` and version `ics27-2`. A packet names a target contract, a function and its arguments. On receive, the app deploys or reuses an account derived from the source client, the sender and a salt, and the account makes the call, so a remote sender acts through an address that is theirs alone and cannot impersonate anyone else. Calls into the router itself are refused. The acknowledgement carries the call's return value.

© 2026 Amanda Gonsalves.

3.13 A transfer, end to end

The sections above take each piece apart. This is what they look like assembled, for a transfer leaving Stellar on the implementation as it stands today.



Reading it in four movements.

Stellar commits (1 to 3). The transfer app escrows the asset and hands a packet to the router. The router assigns a sequence, writes the packet commitment into the provable tree, and recomputes and publishes the tree's root. That publication happens once per invocation, so the ledger carries exactly one router root event, and the root is correct by construction rather than asserted by anything off-chain.

Evidence is gathered (4 to 9). The relayer sees the packet and asks the off-chain services for what it needs. The gateway pulls the consensus evidence from Stellar's public history archives: the `EXTERNALIZE` messages for the ledger, the same for the following ledger, the quorum-set preimages those messages committed to, and the transaction set that binds the two ledgers together. It also produces the ICS-23 proof of the commitment. None of this is trusted. The archives are a file server and the gateway holds no key. A compromised service cannot seed the client with a validator set of its choosing either, because the configuration was fixed when the client was created and checked there against a pinned fingerprint; every header is evaluated against that stored configuration, not against anything the relayer brings.

The host chain verifies (10 to 16). The client update goes first, because a proof newer than the client is rejected. The light client walks the eight checks from section 3.5, and only then stores the authenticated root. The packet follows, and its membership proof is checked against that root. On success the host writes the

© 2026 Amanda Gonsalves.

receipt and the acknowledgement, and credits the receiver.

Then the loop closes (17 to 18). The acknowledgement is proved back to Stellar through the inbound light client of section 3.9, the router clears the commitment, and the transfer app settles the escrow. A failure acknowledgement or a timeout refunds the sender instead.

The inbound direction is the same shape with the roles swapped, and it is shorter: the host chain's header commits to its application state directly, so there is no equivalent of the evidence gathering in steps 6 and 7.

3.14 Failure and recovery

Failure	Effect	Recovery
Relayer stops	Packets stall, nothing is lost	Any relayer resumes; unrelayed packets time out and refund
Relayer misbehaves	Proofs fail on-chain	Nothing to do; the client rejects them
RPC unavailable	New packets cannot be observed or submitted	Verification evidence comes from archives, not RPC; resume when RPC returns
Archive unavailable	Outbound headers cannot be assembled	Use another archive mirror, or a watcher `stellar-core`
Invalid evidence	Header rejected, and the failing check is named	Fail closed, no state change
Insufficient quorum	Header rejected at check 5	Retry with complete evidence; a genuine shortfall means the network did not agree
Conflicting consensus	Misbehaviour path freezes the client	A governance-level decision; a frozen client accepts no packets
Stale client	Consensus state past its maximum age refuses membership proofs	Update the client with fresh evidence
Stellar halts	No new ledgers, outbound stalls	Inbound timeouts still refund once Stellar resumes
Host chain halts	Inbound stalls	Outbound packets time out and refund on Stellar
Gateway restarts	It needs the router's tree before serving proofs	The tree is persisted in Postgres and reloaded on start, then checked against the roots the router published; without a database the gateway falls back to replaying from the router's genesis ledger, which needs an RPC that still holds it
Prover network unavailable	Ethereum-corridor updates cannot be proven	Retry later; nothing is lost, and the proof can also be generated locally on a large enough machine
Ethereum client falls behind	Each missed sync committee period needs its own update proof	Catch up one period at a time while the beacon node still serves those updates, roughly five months

© 2026 Amanda Gonsalves.

3.15 Security analysis

Threat model. A malicious relayer. A malicious or compromised gateway or api. An attacker who can deploy arbitrary Soroban contracts and submit arbitrary transactions. An attacker who can serve arbitrary bytes claiming to be archive data. Out of scope: compromise of a quorum of Stellar validators, which is a Stellar-level failure, and compromise of the host chain's own consensus.

Cryptographic guarantees. Ed25519 over exact statement bytes, SHA-256 at every binding in the chain, and the quorum predicate evaluated structurally rather than as a count. Given quorum intersection despite the ill-behaved set, Theorem 9 gives that two nodes outside that set cannot confirm contradictory statements, so a valid quorum of `EXTERNALIZE` messages for a slot cannot be reproduced for a conflicting value.

Trusted assumptions, in full.

1. The configured quorum configuration is properly related to the network the ecosystem depends on. Not checkable on-chain. §7 of the whitepaper is explicit that adequate slice selection is a precondition for safety.
2. The real network has quorum intersection despite its ill-behaved nodes. §4.1: no protocol can guarantee safety without it.
3. Ed25519 and SHA-256 hold.
4. The `stellar-core` rule underpinning the next-slot binding, described in check 7.
5. On the Ethereum corridor only: the SP1 zkVM executes the programs faithfully, and its Groth16 verifier is sound. The verifying keys are pinned as immutable constructor arguments, so a change to the programs cannot reach a deployed client without a new contract and an explicit migration.

Attacks against relayers. A relayer that lies produces proofs that fail. A relayer that disappears causes timeouts and refunds. A relayer that censors is routed around by another one, because relaying is permissionless and confers no authority.

Replay protection. Packet receipts make delivery idempotent, so a second receive for the same sequence is rejected. Client updates are keyed by height and idempotent. Inside consensus verification, duplicate node ids are rejected before counting, so one signature cannot be replayed to manufacture a quorum.

Forged consensus evidence would require forging Ed25519 signatures for a whole quorum, or finding a SHA-256 collision at one of the bindings. Forging a *state root* additionally requires a success preimage whose router-emitted event carries the false root, and `contractID` is set by the host, which closes the substitute-contract path described in 3.7.

Validator-set attacks. The realistic attack is not cryptographic. It is convincing a deployment to trust a quorum configuration that is not the network's. That is why the configuration is pinned by fingerprint and versioned with validity ranges. Validating it with SDF's quorum analysis tooling and monitoring it for drift are designed but not yet built.

© 2026 Amanda Gonsalves.

3.16 Performance

The on-chain cost of the wasm light client on a Cosmos host has not been measured yet. Everything below is either bounded by inspection or marked as unknown. The Ethereum corridor's costs are measured, and they are in section 3.17.

Dimension	Position
Signature verification	21 Ed25519 checks per client update under the mainnet configuration, plus the same again for the next-slot binding. Bounded and known.
Quorum computation	A fixed-point elimination over at most 21 signers with a nested set. Iterations are bounded by the signer count, and the cost is negligible next to the signature checks.
XDR size	Envelopes and quorum sets are small. The transaction result set is the large term, because `txSetResultHash` is flat and the whole set has to travel. It grows with how busy the ledger was.
Proof size	ICS-23 over a depth-64 tree: 64 sibling hashes, roughly 2 KB, constant.
CosmWasm gas	Not measured. This is the dominant unknown, and the result-set term above is what drives it.
Memory	Bounded by the result set. The decoder enforces explicit length and depth limits throughout.
Update frequency	One client update serves every packet at or below its height, so updates are per batch rather than per packet.
Scalability	Per-slot independence (§6) means no sequential sync is ever required. The cost driver is result-set size, and the mitigations are to bind once per client update and to chain headers backwards from one authenticated header.

Measuring this against a host chain's limits, and applying the batching mitigations if the number demands it, is the immediate next step.

3.17 The Ethereum corridor

Everything above runs the Stellar verification directly inside the host chain. An EVM chain cannot reasonably do that: it has no Ed25519 precompile, and checking 21 signatures plus the quorum predicate in Solidity would be prohibitively expensive. In the other direction, Soroban could verify Ethereum's 512-member BLS sync committee natively, but only just. So the Ethereum corridor moves the verification into the SP1 zero-knowledge virtual machine, and each chain checks a short Groth16 proof that the verification ran correctly.

Nothing about what is trusted changes. The programs inside SP1 run the same checks as the direct clients, and the contracts check the proof against verifying keys pinned at deployment. The one added assumption is the soundness of SP1 and its verifier, listed in section 3.15.

The Solidity side lives in ipsa's fork of `cosmos/ibc-contracts`, the IBC v2 implementation the Cosmos ecosystem already uses to reach Ethereum. Stellar therefore uses the standard `ICS26Router` and `ICS20Transfer` on Sepolia, with only the light client added.

© 2026 Amanda Gonsalves.

Ethereum verifies Stellar

The SP1 program for a header update walks the same chain of custody as section 3.5:

1. Refuse a frozen client.
2. Check that the hash of the supplied quorum configuration equals the fingerprint pinned in the client state.
3. Require the ledger to advance.
4. For adjacent ledgers, check that the new ledger's previous-ledger hash is the trusted one.
5. Authenticate the `EXTERNALIZE` messages, the quorum sets and the quorum itself, bind the agreed value to the header, and derive the ledger hash inside the program.
6. Authenticate the next ledger's messages against the same quorum sets and bind its transaction set back to this ledger, which is check 7 of section 3.5.

These steps are calls into the same `ipsa-scp-verifier` and `stellar-consensus-verifier` crates the standalone checker runs, not a second implementation. A separate membership program proves packet state: it walks the result set down to the router's `ibc_root` event, as in section 3.7, and then proves the requested key-value pairs against that root. A third program combines an update with membership in one proof.

The contract, `SP1ICSStellar`, stores a client state of network id, trust-root fingerprint, router id, root event topic, latest height, frozen flag and proof system. For each verified ledger it stores only the keccak hash of a consensus state holding close time, ledger hash, previous ledger hash, bucket list hash and transaction result set hash. Membership proofs are anchored on the stored result set hash. A membership proof can carry many pairs, and the contract checks that the one being asked about is among them. Misbehaviour submission is not implemented; a conflicting consensus state for an existing height is refused, and freezing is a role.

The measured costs:

- A mainnet header update is 7,331,182 cycles in the zkVM. SP1's Ed25519 precompile is what makes that affordable: about 127,000 cycles per envelope against about 824,000 without it.
- A Groth16 proof of a mainnet ledger took 48.2 seconds on Succinct's prover network.
- Verifying it on-chain costs about 317,000 gas, and the cost is flat in the number of validators. 21 signers compress to the same 356-byte proof as 3.
- Deploying the client costs 2,085,120 gas.
- Each membership pair adds about 128,000 cycles, so one update pays for about 57 packets before a second update is cheaper.

Stellar closes a ledger every five seconds and has instant finality and no unbonding period, so the client never has to follow the chain. It updates only when there is a packet to prove. At 1,000 packets a day, batched 16 to a proof, that is 63 proofs a day instead of 17,280, about 214 times cheaper than tracking every ledger.

Stellar verifies Ethereum

`sp1-ethereum` is a Soroban light client of Ethereum's beacon chain. The update program wraps the upstream `ethereum-light-client` verification, sync committee signatures included, around an SP1-patched BLS backend. It runs to about 53 million cycles and proves in about 100 seconds. A second program proves

© 2026 Amanda Gonsalves.

Ethereum account and storage state with Merkle-Patricia proofs, at about 578,000 cycles. The Soroban contract checks each Groth16 proof with Protocol 28's BN254 host functions, then checks the proof's outputs against the stored client state commitment and the trusted consensus state.

The choice to prove rather than verify natively was measured before it was made, against Soroban's limit of 100 million CPU instructions per transaction:

- Verifying the 512-member BLS committee natively costs 87.2% of a transaction. It fits, with little room for anything else.
- The obvious optimisation, a multi-scalar multiplication over the 512 keys, costs 756 million instructions, seven times the limit.
- Verifying a Groth16 proof costs 27.5%, whatever the number of signers.

The client crossed a sync committee period boundary on testnet on 28 September 2026, in a single update. A client that falls behind catches up with one proof per missed period, which works as long as the beacon node still serves those periods' updates.

© 2026 Amanda Gonsalves.

4. Implementation and Validation

4.1 What is built

On Stellar

`ibc-router` is the IBC v2 core: it registers client types and counterparties, dispatches send, receive, acknowledge and timeout, and owns the provable store. It also maintains the sparse Merkle tree root on-chain, recomputing and publishing it once per invocation.

`ibc-transfer` is the ICS-20 application, moving value over the SEP-41 token interface, with `ibc-voucher-token` as the token it deploys for inbound denominations. `ibc-gmp` is the message-passing application of section 3.12, with `gmp-account` as the per-sender account it deploys. Two inbound light clients verify the counterparties: the Tendermint client of section 3.9, and the `sp1-ethereum` client of section 3.17.

All of them are built, checked and published by the `ipsa-contracts` release workflow, which also deploys them. Release `v0.1.1` deployed the router in use today, `CBA7QRXNLV2VGXNB7JALC4ZRPD4Q3SFAJRXFS3NTUJBFLLIDT6AEECSS`, and attached `deployments.json` with every address and wasm hash it produced, so the deployed contracts can be matched to the published bytes.

Two further clients exist in the repository: an attestation client and a mock client used in tests, whose proof verification returns true unconditionally. Neither is deployed or registered by the release workflow.

On the host chains

`stellar-light-client` is the Stellar light client for Cosmos chains, loaded through the `08-wasm` module. It implements the whole chain of custody in sections 3.5 to 3.7, plus the quorum configuration history, misbehaviour handling and expiry. It is the only Stellar-specific code the host chain runs, and it runs as data rather than as a fork of the host's binary.

Release `v0.1.0` is stored on the Cosmos Hub's public testnet, `provider`, under checksum `26d42704e9a6dfcf38d678419dc419aca29121b9391e72be158e8649bb55002d`, and the client created from it, `08-wasm-464`, is active there. Storing code on `provider` is reserved to its governance authority, so the release carries a ready-to-run store request alongside the artifact; anyone can confirm the stored checksum with `gaiad query ibc-wasm checksums`. The code was stored by governance proposal 328, submitted on 8 October 2026 in transaction `AB1DC4798549B14231D21F2B28A5BC6A76021F7B7789C19B3E929169F0465181` at height 19,345,920, and passed when voting closed the same day: https://www.mintscan.io/ics-testnet-provider/proposals/328.

`SP1ICSStellar` is the Stellar light client for EVM chains, with the three SP1 programs it checks proofs of. Section 3.17 describes it.

© 2026 Amanda Gonsalves.

ipsa-api

The api is the only backend service with a Soroban RPC connection, and it holds no signing key: its `/security` route fails if any key is configured. Thirty-eight HTTP routes. Chain reads serve the latest ledger, a ledger by sequence, contract events, registered clients and their state, token balances, and the corridors the deployment serves. Archive reads serve the SCP messages, ledger header and transaction set for a given ledger, which is where the consensus evidence comes from. Transaction handling builds unsigned envelopes and submits pre-signed ones, so the service can construct any router call without ever being able to authorise one. Relay routes hand a source transaction to the relayer and report per-packet progress back. Verify routes run the consensus verifier against a chosen ledger, and against deliberately mutated copies of its evidence, for the public console. An OpenAPI document is generated from the handlers, and a test asserts that every routed endpoint appears in it, so the published surface cannot silently drift from the implemented one.

stellar-watcher

A non-validating `stellar-core` node that follows consensus live. It runs as its own service, with a small reader (`ipsa-watcher`) beside it that serves its database read-only, so deploying the api never restarts it. The api queries the reader and prefers it over the history archives when serving SCP evidence, decoding every envelope again and falling back automatically when the watcher is absent, behind or slow to answer. This removes the checkpoint-cadence floor described in section 3.11 for the common case while keeping the archives as the durable source.

It changes nothing about trust. SCP envelopes are self-authenticating and quorum sets are content-addressed, so a faulty watcher can delay a transfer but cannot forge one. Like the api and the gateway, it holds no key and is treated as untrusted transport.

ipsa-gateway

The gateway is the only component the relayer talks to, and it is built so that compromising it cannot move value: no signing key, no chain connection of its own, every call fulfilled through the api. It exposes twenty gRPC methods across a query service and a message service, plus four more on the proof API described below.

Its substantive work is assembling evidence. For a given ledger it produces the ICS-23 proof of the packet commitment, and the header bundle the light client needs: the `EXTERNALIZE` messages for that ledger and its successor, the quorum-set preimages those messages committed to, the transaction set that binds the two ledgers, and the state-root proof walked out of the ledger's transaction results down to the router's own event. It also serves the quorum configuration and the router identity a client needs at creation, which the CLI and the relayer treat as a convenience rather than an authority.

To produce ICS-23 proofs it keeps the router's sparse Merkle tree, checked against every root the router published. The tree is persisted in Postgres and reloaded on start, so a restart no longer depends on public RPC, which keeps only about a week of history. Without a database configured the gateway replays the router's writes from its genesis ledger instead.

© 2026 Amanda Gonsalves.

The relayer today

Both corridors run on ipsa's fork of the Cosmos IBC v2 relayer, public at https://github.com/ipsadev/ibc-relayer. One relayer serves every counterparty, and in the staging deployment it runs hosted on AWS beside the api, the gateway and the proof API. Hermes, which carried the integration earlier, is gone from the stack, the configuration and the CLI.

The Stellar side of that relayer is a chain client of about 1,400 lines, with about 1,500 more of tests: transaction construction and submission as Soroban invocations, event-backed packet discovery, a finality rule which for Stellar is simply that a ledger is final once SCP externalizes its value, and a local signer. The quorum fingerprint check is written and tested in the relayer but not yet called on the relay path, as section 3.8 notes.

The architectural point is that no chain-specific proof logic lives in the relayer. It obtains everything it needs from a proof API over gRPC, so the Stellar-specific work sits behind a service boundary this project owns. That interface carries the packet relay and both halves of the client lifecycle, in both directions, and it is what the relayer calls to build a message it then signs and submits.

IBC v2 has no channels, so the relay is client-paired rather than channel-paired, with a proof source and a submitter for each direction independently. Batching is deliberately pinned to one packet in the Stellar direction, since Soroban allows a single host-function invocation per transaction and the router's receive entrypoint takes one packet. The acknowledgement policy relays success acknowledgements rather than dropping them, because ICS-20 escrow only settles once the success acknowledgement gets home.

The relayer next

What remains is operational rather than architectural.

- The relayer's own gas monitor cannot yet read a Stellar fee-account balance, so that account is watched from the api instead.
- Client updates ride inside a packet relay, so an idle corridor has no scheduled refresh, and a long enough gap could let a client expire. The light client refuses to advance once its trusted state is stale rather than accepting a jump forward; a refresh worker is the remaining half of that.
- The relayer has no way to give up on an acknowledgement the counterparty will never accept. A devnet transfer acknowledged before the transfer app adopted the exact ICS-20 acknowledgement was left stuck that way.
- Stellar-bound delivery is pinned to one packet per transaction and one transaction at a time. Serialising submissions on the Stellar account is written but not yet committed.
- Timeouts on the Ethereum corridor, described in section 3.17.

The CLI and the verifier

`ipsa` is a single binary that operates the whole stack: eleven top-level commands covering installation and health checks, building, bringing services up and down, deploying contracts, creating clients and registering counterparties, originating transfers and message-passing calls on both corridors, driving the relayer and the proof API, and running the live test flows. It ships the stack it runs, the compose file and its assets, so it needs no other checkout. It keeps its own settings apart from the stack's, and writes what it creates back to whichever of the two owns it: contract addresses and client ids to the stack, account keys to itself. It builds

© 2026 Amanda Gonsalves.

nothing it deploys. The contracts and the light client are downloaded from their GitHub releases and every file is checked against the published checksums before use, and no other repository's location is written into its code; where releases, images and sources live is read from configuration.

The SCP verification lives in its own crate, `stellar-consensus-verifier`, published on crates.io. It has nine modules, one question each, depends on nothing but `ed25519-dalek`, `sha2` and `thiserror`, and has no network access, storage or clock. It is the audit target. The Cosmos light client, the EVM proof programs and the standalone checker all depend on it, so there is one copy of the verification rather than several kept in step.

The checker used throughout section 4, `ipsa-scp-verifier`, is also on crates.io. It fetches the evidence for a ledger from the public archives and reports eight steps one by one, from envelope signatures to the result-set binding. Beyond verifying one ledger, it runs a single-mutation negative suite, a range sweep that classifies every ledger across a span of history instead of stopping at the first failure, and three differential checks: one against `stellar-xdr`, one that throws 20,000 random federations at the quorum predicate and compares it with an exhaustive search, and one against **stellar-core's own quorum code**, compiled from a real stellar-core tree and driven as a child process. It evaluates the quorum only against a trust root the caller supplies, and will not run without one: the quorum sets in the archive are pledged by the same validators whose signatures are being checked, so they cannot also decide who counts.

The console

https://app.ipsa.dev puts the same verifier in front of anyone with a browser, through the api's verify routes. It verifies a real mainnet ledger step by step, and lets the visitor tamper with the evidence to watch each check reject it. An explorer lists the transfers the relayer has carried, and the chains, routes and registered clients the deployment serves, with each address and transaction linked to that chain's block explorer. It is read-only: no wallet and no signing.

Releases

Every artifact in section 4.1 is released from a tag by a workflow that refuses unless the tag matches the version, runs the repository's own checks on that exact tag first, and never replaces bytes it has already published. As of 15 October 2026:

Repository	Releases	Latest
`ipsa-ibc-core`	crate	0.1.2
`stellar-consensus-verifier`	crate	1.0.1
`ipsa-scp-verifier`	crate	3.0.0
`stellar-light-client`	reproducible wasm, built twice and compared, with its checksum	v0.1.0
`ipsa-contracts`	every contract's wasm with `SHA256SUMS`, deployed to testnet by the same run	v0.1.1
`ipsa-backend`	the api and gateway images and the OpenAPI document	1.0.7

© 2026 Amanda Gonsalves.

4.2 The shape of the codebase

In September 2026 the single repository was split into separate ones, so that each artifact can be read, built, released and audited on its own. Figures are from the committed trees on 12 October 2026, except `ipsa-scp-verifier`, counted at its `v3.0.0` tag on 15 October; `ipsa-ibc-core` is counted from its published crate. Test counts are test functions in source, not a single run.

Repository	What it holds	Rust lines	Tests
`ipsa-contracts`	The Soroban contracts, including the SP1 Ethereum client and verifier	12,443	232
`stellar-consensus-verifier`	The SCP verification core, the audit target	5,008	164
`stellar-light-client`	The `08-wasm` client for Cosmos chains	5,669	115
`ipsa-scp-verifier`	The standalone checker	9,153	101
`ipsa-ibc-core`	Commitment paths, the sparse Merkle tree, ICS-23 proofs	1,846	52
`ipsa-backend`	`ipsa-api`, `ipsa-gateway`, integration tests	17,941	232
`ipsa-tooling`	The `ipsa` CLI and the stack it runs	10,267	52

Two forks sit beside them, with figures from 8 October. `ibc-relayer` carries about 2,900 lines of Go for Stellar, with 63 tests. `ibc-contracts`, the fork of `solidity-ibc-eureka`, carries the EVM client and the SP1 programs; its Stellar suite has 29 Solidity tests, and the whole Solidity suite passes, 309 tests across 38 suites.

About 785 commits across ipsa's own repositories, between 13 May and 12 October 2026. Most of the history is in `ipsa-backend`, which is where the single repository's history was kept when the split happened.

The compiled artifacts, which is what actually gets deployed, sized from their releases:

Artifact	Size	Where it runs
Stellar light client	450 KB, 145 KB compressed	Cosmos chains, loaded via `08-wasm`
`ibc-router`	27 KB	Soroban
`ibc-transfer`	23 KB	Soroban
`ibc-voucher-token`	18 KB	Soroban
`ibc-gmp`	17 KB	Soroban
Tendermint light client	18 KB	Soroban
`sp1-ethereum` light client	9 KB	Soroban
`SP1ICSStellar`	2,085,120 gas to deploy	EVM

© 2026 Amanda Gonsalves.

The Soroban contracts are small because IBC v2 is small: three provable paths, no connection or channel objects, and no handshake state machine. The Ethereum client is the smallest of all, because the heavy verification happens inside the proof and the contract only checks it. The Cosmos artifact is larger because it carries the XDR decoders, the quorum evaluation and the full verification chain.

4.3 Mainnet verification results

Checked	Result
Signatures, quorum evaluation, agreement, ledger binding, next-slot binding	Confirmed on live mainnet and testnet ledgers: 21 signers under the nested mainnet configuration, 3 under a flat testnet one
The ledger commits to its transaction results	Reconstructed for 64 of 64 mainnet ledgers in a checkpoint
The result commits to the contract state root	Reconstructed for 40 of 40 Soroban invocations across eight consecutive testnet ledgers

The mainnet quorum structure observed at ledger 63907880 in August 2026 was 5 of 7 organisations, each satisfied by 2 of its 3 validators.

The same ledger was then proven for the EVM. A Groth16 proof of mainnet ledger 63,907,880 verifies through a real SP1 Groth16 verifier in the Solidity test suite, and a header update built from mainnet data landed on Sepolia for 316,946 gas.

4.4 Test vectors

A verified run can be frozen with `--export` and replayed with `--fixture`, which gives deterministic regression tests over real consensus data with no network access. A fixture carries the envelopes, the quorum sets, the ledger header, the next-slot evidence and the expected ledger hash.

The tool also cross-checks its own XDR reader against the contract's. Both have to agree on exactly where a signed statement starts and ends, so a disagreement shows up as an explicit error instead of an unexplained signature failure.

4.5 Negative tests

A 25-case suite (`--negative`) applies exactly one mutation to verified input per case and asserts that the expected check rejects it. Twenty always run; the remaining five need the optional evidence, a result set and a next slot, to be present.

© 2026 Amanda Gonsalves.

Check	Cases
Envelope signatures	tampered signature; wrong network id; tampered highest-confirmed ballot counter; forged `CONFIRM`
Statement validation	tampered ballot counter; non-`EXTERNALIZE` statement; wrong slot; duplicate signer
Quorum-set authentication	missing preimage; preimage that hashes to something else; malformed nested set; zero threshold
Quorum check	signers outside the configured trust root; a single signer is not a quorum; no envelopes at all; every signer pledging a different value, so no value is quorum-backed
Value agreement	conflicting committed values
Ledger binding	value does not match `header.scpValue`; header for another ledger; truncated header XDR; wrong recorded ledger hash; header naming a predecessor other than the trusted one
Next-slot binding	tampered transaction set; transaction set naming another predecessor
Result-set binding	tampered result pair; result pair removed; result pairs reordered

A case that is accepted, or that is rejected by the wrong check, fails the suite.

4.6 End-to-end proof

Packets run end to end on both corridors, from the CLI, relayed by the relayer rather than by hand, with commitments, receipts and acknowledgements exercised on live Soroban testnet. The acknowledgement check fails the run when no acknowledgement appears, so a silently broken leg cannot pass as success.

The Cosmos Hub. An ICS-20 transfer in each direction between Stellar testnet and the Cosmos Hub's public testnet, `provider`, both acknowledged, on 10 October 2026, over the clients `07-tendermint-1` on Stellar and `08-wasm-464` on the Hub. Outbound, 0.1 XLM reached the Hub as an IBC voucher with the success acknowledgement `{"result":"AQ=="}`; inbound, 1,000 `uatom` reached Stellar and its acknowledgement was accepted on the Hub. Both legs ran through the hosted relayer.

A local Cosmos devnet. An ICS-20 transfer in each direction between Stellar testnet and a local ibc-go v11 devnet running the wasm light client, both acknowledged, on 4 October 2026 and again on the released contracts on 8 October. The clients on both sides were created and paired by the CLI.

Ethereum. An ICS-20 transfer in each direction between Stellar testnet and Ethereum Sepolia, both acknowledged, on 27 September 2026. The first Sepolia-to-Stellar packet of a new sync committee period crossed the period boundary on 28 September. On 7 October, after Sepolia's Gloas upgrade and a migration to a new Ethereum client, a Stellar-to-Sepolia transfer completed with its acknowledgement again.

On the released contracts the corridor was paired again from scratch. The first three Stellar-to-Sepolia transfers, on 10 and 11 October, were rejected by Sepolia's transfer app: the first because the Stellar transfer app still encoded that client's packets as JSON rather than Solidity ABI, the next two because creating the voucher token ran out of gas at the relayer's estimate, after Sepolia raised the cost of contract creation. Each

© 2026 Amanda Gonsalves.

came back as IBC v2's universal error acknowledgement and was refunded on Stellar. That exercised the failure path end to end, a rejection proved back to Stellar and the escrow returned, which no other run had. With the encoding set per client and the relayer sending 2.5 times its gas estimate, the fourth, on 12 October, was accepted on Sepolia and minted the XLM voucher. The voucher was then sent back the same day: the Sepolia-to-Stellar transfer was acknowledged, the voucher burned on Sepolia and the escrowed XLM released on Stellar. All four legs ran through the hosted relayer and the hosted proof API.

Every transaction hash, address, client id and verifying key for the Ethereum corridor is recorded in a deployment log, so a third party can check each one on both chains.

© 2026 Amanda Gonsalves.

5. Sources

1. **The Stellar Consensus Protocol**, David Mazières, Stellar Development Foundation, draft of 25 February 2016. Local copy: `scp/stellar-consensus-protocol.pdf`. Used for the protocol rules: federated Byzantine agreement systems and quorums (§3.1), safety and liveness (§3.3), quorum intersection and dispensable sets (§4), federated voting with ratify, accept and confirm and Theorems 8 through 11 (§5), SCP itself including the ballot protocol and `EXTERNALIZE` (§6), and the limitations including the configurable-parameter risk (§7).
2. `stellar/scp-proofs`, SDF's Isabelle/HOL and Ivy formalisations of SCP safety, along with the caveats those proofs state about their own abstractions: slices abstracted into axioms, nomination not modelled, a single slot, and no mechanically checked link between the Ivy axioms and the Isabelle theory. https://github.com/stellar/scp-proofs
3. `stellar-core` and `stellar-xdr`, pinned to versions. Used for wire-level facts: signing preimages, envelope type values, structure layouts, and how a ledger header is built from an externalized value. These are implementation facts, not protocol facts. https://github.com/stellar/stellar-core/blob/master/docs/architecture.md
4. **The relayer integration**: the Stellar chain endpoint and the IBC v2 packet worker. https://github.com/ipsadev/ibc-relayer
5. **IBC**, the Interchain Standards. https://ibcprotocol.dev/ and https://github.com/cosmos/ibc
6. **Cardano IBC incubator**, the earlier non-Cosmos IBC integration whose relayer and light-client patterns this work builds on. https://github.com/cardano-foundation/cardano-ibc-incubator
7. **Live Stellar data**, the public history archives and RPC on both testnet and mainnet. Used to check whether all of the above holds in practice.
8. `cosmos/solidity-ibc-eureka`, the IBC v2 implementation for EVM chains, whose router, transfer app, SP1 light client pattern and Ethereum light client this work builds on. https://github.com/cosmos/solidity-ibc-eureka
9. **SP1**, Succinct's zero-knowledge virtual machine, its Groth16 verifier and its prover network. https://github.com/succinctlabs/sp1
10. **The published crates**: https://crates.io/crates/stellar-consensus-verifier, https://crates.io/crates/ipsa-scp-verifier and https://crates.io/crates/ipsa-ibc-core.

© 2026 Amanda Gonsalves.