

The overlay bridge: an unmodified engine on the object plane

An overlay host runs an engine that admits BEEF submissions by topic, indexes the results for lookup, and propagates every admitted transaction to every other host advertising the same topics, one HTTPS request per host. This paper shows how a small landing-tier bridge lets an unmodified overlay engine run with the BEEF object plane as its transport: objects delivered by the plane enter the engine through the submit interface it already serves, publication goes through the bridge once, in place of the engine's own host-by-host propagation, and the network fans it out to every subscribed host, and the block-header lane feeds the chain tracker the engine verifies against. It gives the pre-flight checklist, the cases where stock propagation remains the right answer, the upstream changes that would tighten the fit, and how an overlay host joins the application tier of the 1BSV network. It is the companion to *The overlay object plane*, and the base on which the application blueprints build.

Author: Jeff Harris · Lightweb Inc. · jeff@lightweb.net

Paper 008 · **Version** 05

Audience: overlay application developers and the operators who run overlay hosts.

We = the 1BSV fabric and its **overlay-bridge**; you = an operator running an overlay services engine, in TypeScript or Go.

The short version

- **Overlay propagation pays two fan-out costs, and the plane removes both.** A publisher submits to every interested host, and each host that admits a transaction re-submits it to every other host it can find. Between N hosts of a topic the same object crosses the network on the order of N squared times, and a host that was unreachable simply misses it.
- **Delivery enters the engine as a normal submit.** The bridge terminates the object lane and hands each delivered object to the engine's own submit interface with the topic it matched. The engine parses, verifies, admits, and indexes exactly as it does for any client.
- **Propagation collapses to one publish.** The bridge serves the same submit interface, so a publisher submits to it once and the plane delivers to every subscribed host at the same moment. The engine's own propagation is switched off in configuration, not in code: the plane is the propagation.
- **Verification stays sovereign.** The block-header lane feeds a local chain tracker, so SPV inside the engine checks against headers the host itself received, with no third-party header service in the loop.
- **The bridge requires no fork.** The engine runs as released; the bridge speaks the interfaces the engine already speaks and holds no state of record. It is Apache-2.0; what you pay for is the delivery it lands.
- **History stays where it is.** Discovery, lookup, and catch-up remain the overlay's own protocols; the plane carries the live tail.

1. What an overlay host does today

An overlay host is an HTTP server running an engine that mounts topic managers and lookup services over a storage backend (BRC-22, BRC-24). A client sends a BEEF object and a list of topic names to the host's submit endpoint; the host verifies the object by SPV against a chain tracker, asks each named topic manager which outputs to admit and which spent inputs to retain for history, stores the admitted outputs, notifies its lookup services, and returns per-topic admittance instructions. Hosts find each other through the SHIP and SLAP overlays (BRC-88): on-chain advertisements naming which host serves which topic and which lookup service.

Two fan-outs follow from that design, both structural.

The publisher fans out. A wallet or application resolves the hosts advertising its topics and broadcasts the same object to each of them. By default the broadcast counts as successful when at least one responsive host acknowledges every requested topic; the rest are reached eventually, or not.

Every host fans out again. Inside the submit path, after admitting a transaction, the engine re-submits it to every other host advertising the affected topics. There is no marker distinguishing a peer's submission from a client's, so each receiving host verifies the object before its duplicate check discards it. Between N hosts of a topic, one object is therefore verified and discarded on the order of N squared times.

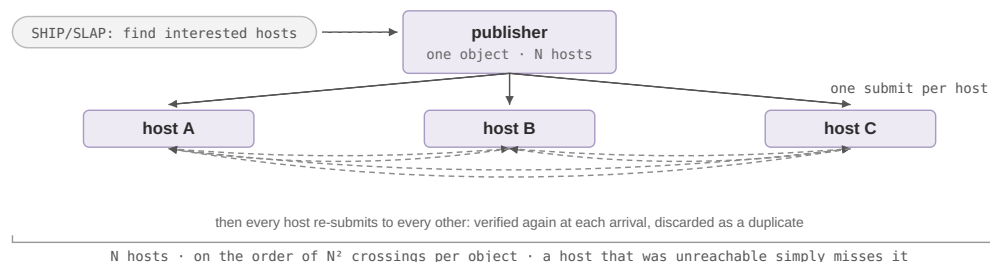


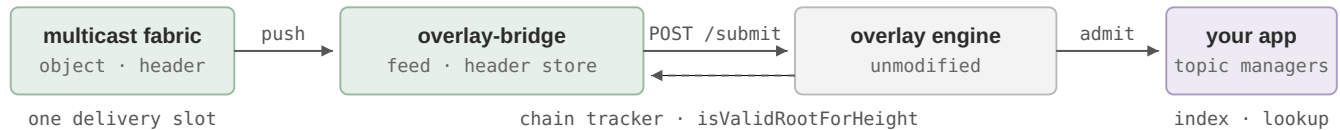
Figure 1 - propagation today: the publisher submits to every interested host, and each host re-submits to every other.

Why live propagation is best-effort, and what the object plane (BRC-148/149) does about it, is the subject of the companion paper *The overlay object plane*. In one line: a publisher emits an object once, and every subscribed host of its topic receives it, complete, with loss repaired inside the network. That paper treats the first fan-out. This one is about the second, the one only an engine operator sees, and about how an unmodified engine gets onto the plane.

2. The bridge: three parts, one landing machine

overlay-bridge runs on the landing machine beside an unmodified overlay engine. It has three independent parts (the feed, the header store, and the facade) and holds no state of record.

Delivery (down)



Publication (up)

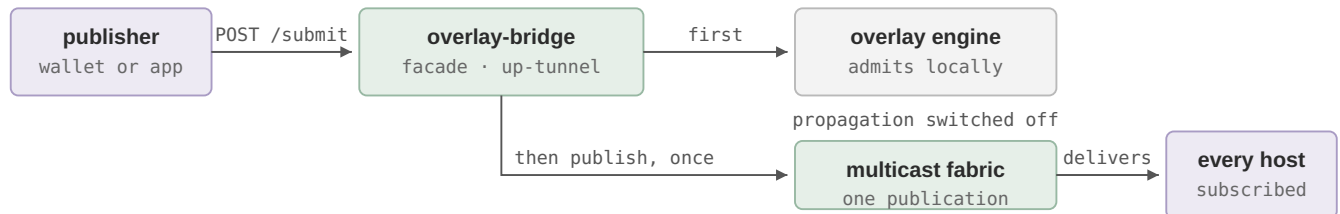


Figure 2 - the three parts. Down: delivered objects become ordinary submits, and delivered headers become the engine's chain tracker. Up: the facade serves the submit interface, so a publisher submits once through it; the engine itself no longer propagates.

overlay-bridge is the application-tier sibling of **arcade-bridge** (the broadcaster tier) and **teranode-bridge** (the miner tier), and shares their landing-tier library: lane termination, byte-order discipline, redial and failover across the tunnel's primary and standby paths. The bridge imports that library and never forks it. As with its siblings, the kind of consumer it serves is what makes it a separate binary: an overlay host is a delivery consumer of the application tier, never a miner, and nothing in the bridge carries miner-tier semantics.

3. Delivery: the lane becomes a submit

The object lane delivers one length-delimited record per object, the delivery record of BRC-149: the identifier of the topic that matched, the object's length, and the BEEF object (BRC-62) verbatim. The feed splits the stream on the explicit length, maps the topic identifier back to the name the host elected (the identifier is the hash of the name, so the map is computed from the election, never fetched), and hands the object to the engine:

```
POST /submit
Content-Type: application/octet-stream
X-Topics: ["tm_example"]

<the delivered BEEF bytes, unchanged>
```

That is the engine's own interface (BRC-22), so what happens next is what happens for any client: the engine parses the BEEF, verifies it by SPV against its chain tracker, runs the topic manager, stores admitted outputs, notifies lookup services, and returns admittance instructions. The bridge records the instructions as metrics and moves on. A duplicate (an object the host already applied for that topic) returns empty instructions, which is the engine's normal answer and costs the bridge nothing further. A submit that fails in transport, because the engine is restarting or its socket is briefly refused, is retried a bounded number of times off the lane's read loop before it is counted as an engine error, so an engine restart does not cost the host the objects delivered during it.

The lane's properties are the plane's, and the companion paper gives them: objects arrive whole, delivery is precise to the topics and encodings the host elected, and an object's identity is its bytes. The feed relies on all three and adds nothing. The last one matters to the bridge specifically: a transaction republished with a refreshed proof arrives as a distinct object, and the engine's Merkle-proof handling updates the stored transaction as it would for any proof arrival, which is how "now mined" reaches every host at once.

One property of the lane does not simplify the feed, and it is the one to design for. The plane is open to any publisher: the object a lane delivers is whatever some publisher chose to send, and the network never parses past its leading marker, by design. So any parser the feed hands those bytes to must check every length the object declares against the bytes actually present before it allocates anything of that size. This is not a theoretical concern. A BEEF reader that sizes a Merkle path straight from the wire can be made to demand an unbounded allocation by a very small object, and that failure is not catchable in the usual way: the runtime kills the process outright instead of returning an error. A host that terminates the lane with its own code should validate before it parses, and treat a parser failure as a rejected object rather than an incident. **overlay-bridge** does this for you.

The submit interface admits in the engine's default mode, which would normally include its propagation step. On the plane that step is switched off in configuration (§5), so a delivered object is admitted and goes no further: the plane has already delivered it to every other host. Engines also expose an in-process submit with an explicit historical mode. The bridge does not use it, because a separate binary speaking HTTP is what keeps the engine unmodified.

4. Headers: the chain tracker the engine already wants

Every engine verifies BEEF against a chain tracker: a component that answers whether a Merkle root is valid at a block height, and what the current height is. Stock deployments point that component at a public header service. The bridge instead feeds it from the block-header lane the network delivers beside the object lane (BRC-135): 80-byte headers as the chain advances, chained and proof-of-work checked on arrival, held in a local store, and served to the engine either through the chain-tracker adapter its language uses (an in-process interface in TypeScript and Go, a few lines each) or as the header service the engine already knows how to poll, so that pointing the engine at the bridge is one configuration line.

The consequence is sovereign verification: every object the host admits was checked against headers the host itself received, from the same network that delivered the object, with no header service to trust or to be rate-limited by. The header lane is a live tail; on start, or after a gap longer than the network's repair window, the store re-anchors from any header source the operator trusts and then follows the lane. An anchor has to answer for a header by its hash as well as for the tip, because a host that restarts below the chain tip must fetch the parents it never received; the bridge's own header store serves that route, so one bridge can anchor another and a site's second verifier needs no external header source at all. A root answered from the anchor is reported apart from one received on the lane, which is what lets an operator see that verification is sovereign rather than assume it.

5. Publication: one submit, every host

The facade serves the submit interface (the same endpoint, headers, body, and per-topic response as the engine) and turns each accepted submission into one publication on the object plane: the topic names and the BEEF bytes, submitted once to the nearest network door over the same tunnel that delivers. The network emits the object once per named topic and delivers it to every subscribed host.

Publishers submit to the facade, and the engine stops propagating.

The engine's own propagation is switched off. Both engines propagate through the same two components: an advertiser, which also maintains the host's SHIP and SLAP advertisements, and a resolver seeded from a list of SLAP trackers, which finds the peers to propagate to. Either one switched off ends propagation. A host assembled on the engine library runs with no advertiser; a host on the released Express server, which always supplies an advertiser, runs with an empty tracker list instead, so propagation resolves no peers. Both are configuration choices, so the engine is still the released engine, and both remove the second fan-out completely: an admitted object is stored and indexed, and nothing is re-submitted to anyone, because the plane has already delivered it everywhere it is wanted.

Publishers. Today a wallet or application resolves and broadcasts to N hosts. With the bridge it submits to the facade instead, as a fixed host in its configuration. The facade forwards the submission to the local engine first, so the host that received it admits it immediately and the client gets the engine's real admittance instructions, then publishes to the plane for everyone else.

One guard keeps a publication from going round twice. With the engine no longer propagating, nothing loops by construction, but the same object can still reach a facade a second time: a client configured with two facades, or a client re-sending what the plane has already delivered. The facade keeps a bounded set of the identities of objects it has recently published or received from the feed (the hash of the object's bytes, which is also how the plane identifies an object) and does not publish those again.

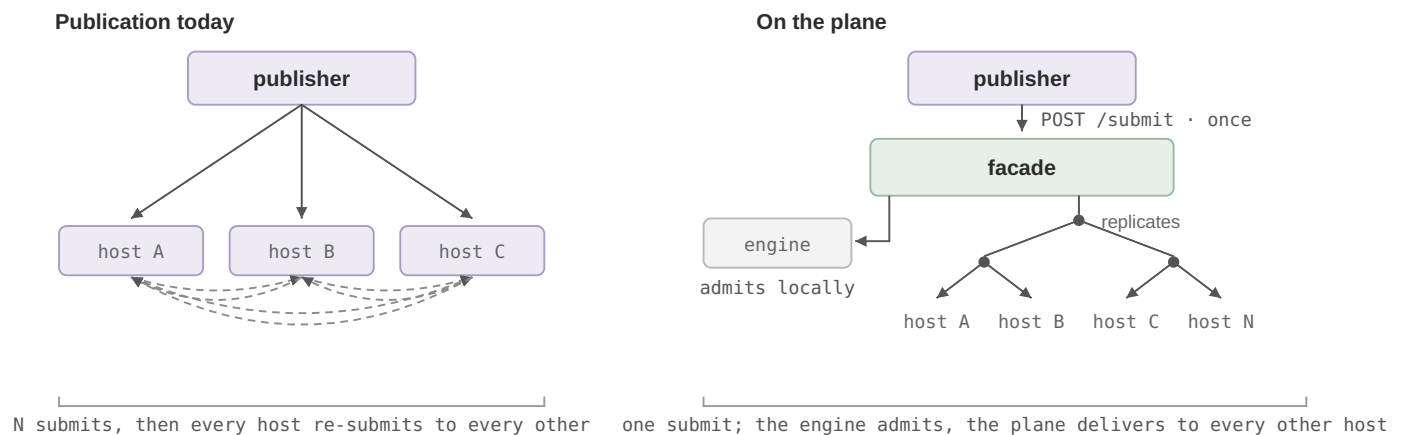


Figure 3 - publication before and after. Today the client submits to every host and every host re-submits to the others. On the plane the client submits once to the facade, the local engine admits at once, and the plane delivers to every other subscribed host.

A topic moves to the plane host by host. A host that has moved sends no HTTPS propagation, and receives it only if it still advertises (limit 5), so while a topic is mixed, the hosts on the plane and the hosts still off it meet through catch-up: each names the other as a synchronization peer (§6), and publishers that want every host immediately keep submitting to the ones that have not moved. When the last host of a topic is on the plane, one submit to any facade reaches them all.

5.1 Multi-topic publication

A single submit may name several topics, and the facade passes the list through unchanged to the engine. Toward the plane it publishes one record per topic, so every topic the client named is delivered on rather than carried as a label past the door's delivered-topic count; the plane's rule (every name travels, the public door delivers the first, an authenticated connection the first three) is set out in the companion paper. On the way down, a delivery whose record names several topics the host elected is handed to each of them, since the plane delivers an object once per subscriber.

6. What stays where it is

The plane replaces live propagation and only that; discovery, lookup, history, and catch-up remain the overlay's own protocols, as the companion paper sets out. The bridge does not touch the engine's synchronization configuration: a host names the peers it backfills from as it does today, runs its catch-up passes as it does today, and rides the stream in between. Catch-up answers "what did I miss while I was away"; the plane keeps every host of the topic in step from now on. Nothing in the bridge is a source of truth: stop it, give the engine back its propagation peers and its stock chain tracker, and it is a stock overlay host with stock propagation again.

7. What you build on top

The bridge changes nothing about how an overlay application is written. The application is still a topic manager that decides admittance, a lookup service that indexes what was admitted, and whatever serves the result to users; all three run on the stock engine as they would without the plane. What changes is that every host running them hears the same objects at the same moment. Application blueprints published separately show how particular kinds of overlay (a high-rate data stream, a low-rate directory of state) are shaped to make the most of that; none of them requires anything of the bridge beyond what this paper describes.

8. Deliver once per landing site

The scarce resource is the tunnel, not the landing machine. A site that runs several bridges (an overlay host beside the arcade installation that broadcasts its transactions, for instance) provisions one delivery slot and elects every lane it needs on it; each bridge terminates its own lanes, and the object and header lanes do not collide with the settlement lanes the other bridges take. When two bridges on one machine need the same lane, a small tier-neutral tee (follow-up 4 in §11) fans it out locally, where the copy costs nothing.

9. What to check before you start

#	Check	Notes
1	A provisioned delivery slot	The commercial step: a tunnel endpoint lands on your machine with the object lane and the header lane; the topics and encodings you subscribe to are yours to set and change after approval
2	<code>overlay-bridge</code> beside the engine	One static Go binary, Apache-2.0, built from the public repository; no engine import
3	The engine's propagation switched off	One configuration choice: no advertiser on the engine library, or an empty SLAP tracker list on the Express server. The engine admits and indexes but never propagates, because the plane does that. Publishers are pointed at the facade
4	The chain tracker pointed at the bridge	One configuration line where the engine polls a header service, or a few lines of adapter in the engine's language; or keep the stock header service and lose sovereign verification
5	Catch-up peers named explicitly	The engine's synchronization configuration lists the siblings it backfills from, as today

Only the first row ever blocks anyone; the rest is an afternoon.

10. What it costs, and when stock propagation is the right answer

`overlay-bridge` is Apache-2.0 open source; the overlay engine runs as released, in TypeScript or Go, with no fork and no patch. There is no license check in the bridge, no call home, and no metering of any kind inside it.

What is not free is fabric delivery itself: a provisioned tunnel carrying a live object lane is the paid relationship. The bridge is inert without it. The value is not access to the bridge; it is the N-squared propagation and the silent gaps you no longer carry, and what that leaves you free to build. Every host of an application hears the same objects at the same moment, each one a BEEF object that carries its own proof and that the host verified against headers it received itself. A distributed application built on that footing is easier to design, because the network keeps its hosts in step and the application need not; more accountable, because every admission rests on a proof the host checked itself; and more capable, because it can assume what host-by-host propagation never promised: that every host of the topic heard every object (§7).

Three layers of cost meet at an overlay host, and the bridge changes only one of them. Miner fees and hosting are the operator's, as before. Delivery is billed by the network on the volume delivered to the host and on nothing else (the companion paper gives the meter in full). Whether a host recovers its costs from the application it serves, from paid lookups, or not at all is the operator's business model, and nothing here constrains it.

Stock propagation over HTTPS is still the right answer in three cases:

1. **The topic has a handful of hosts.** With three hosts, N squared is nine submits and the duplicates are invisible. The costs are real but unfelt, and a tunnel cannot beat free; the case for the plane strengthens as topics and hosts multiply.
2. **N direct paths are worth their bandwidth.** Submitting to every host directly is N independent network paths and N independent acknowledgements. The plane trades those for one publication with network-side repair and a standby tunnel path behind it. If independent paths are the requirement, that trade is the wrong one.
3. **Zero external dependencies is the policy.** Declining a paid delivery relationship keeps the host self-contained. That is a procurement position as much as a technical one, and legitimate.

11. Follow-up work

1. **A public submit door.** The facade speaks the overlay submit interface on the landing machine. The same interface offered at the network's public doors would let any existing wallet or application publish to the plane with a configuration change and no bridge at all.
2. **A chain-tracker service.** The header store is embedded in the bridge and served to one engine. A site running several verifiers would prefer it as a small shared service; the header lane already delivers once per site.
3. **A replay cache.** The bridge holds delivered objects only long enough to submit them. A host that wants a replay window (to re-submit after an engine restart without waiting for catch-up) needs a cache behind the seam the landing-tier library already defines; the miner-tier bridge fills that seam in its licensed build, and the same swap applies here.
4. **A co-located deliver-once tee.** The local fan-out of \$8, for a site that runs this bridge beside an arcade or miner bridge on one machine. It arrives with the first such site.

12. Future upstream optimizations

Each of these has a workaround in the bridge, so none is a blocker; adopting one would turn its workaround into a first-class path.

1. **A submission mode on the submit interface.** The engine's in-process submit distinguishes a live submission from a historical one, and the historical mode skips propagation. Exposing that choice on the HTTP interface (a header) would let a host keep its advertiser, and with it its own SHIP and SLAP advertisements and stock propagation toward hosts that are not on the plane, while objects delivered by the plane skip the propagation step.
2. **A native plane-ingest mode.** The bridge exists to translate the delivery record into a submit. An engine that could terminate the object lane directly would make the plane a first-class source beside HTTPS.
3. **A URL scheme for the plane in SHIP advertisements.** BRC-101 already anticipates advertisements naming non-HTTPS transports and "IPv6 capabilities and bridges". A scheme identifying the object plane would let a publisher discover it the way it discovers hosts today.

13. How it fits the 1BSV solution

1BSV is one structured multicast network with several kinds of consumer on it. Miners take blocks and subtrees and submit their own; broadcasters (an arcade installation, for instance) take proofs and submit transactions; overlay hosts take the object lane and the header lane and publish objects. The fabric, the edges, the tunnels, and the addressing are shared; what differs is which lanes a consumer elects. An overlay host that relies on an arcade installation to broadcast its transactions and return their proofs can land that installation beside it through its own bridge (see *Arcade v2 on a multicast delivery fabric*); they are then two consumers of one network on one landing site.

The overlay host is an application account, whose lanes and meter the companion paper sets out. [overlay-bridge](#) places the overlay tier on that network without a new substrate: its only new surfaces are the feed that makes delivery look like a submit, the header store that makes the header lane look like a chain tracker, and the facade that makes a submit look like a publish. The tiers compound: every host the plane adds is one more subscriber every publisher reaches at no added submit cost, and every application that publishes adds demand for exactly the delivery the network already sells.

14. Assumptions and limits

Quantity	Value	Basis
Client fan-out today	one submit per interested host	structural: BRC-88 broadcast behavior
Host fan-out today	one re-submit per other host, per admitted transaction	structural: the engine's propagation step (BRC-22 step 6, BRC-88)
Network crossings per object, today	on the order of N squared for N hosts of its topic	structural: client fan-out followed by host fan-out
Submit interface	<code>POST /submit</code> , octet-stream <code>BEEF</code> , <code>X-Topics</code>	verified: BRC-22
Bridge license	Apache-2.0, no fork, no engine import	structural: speaks the engine's HTTP interfaces only
Engine propagation on the plane	off: no advertiser, or no SLAP trackers to resolve peers from	verified: both engines skip propagation entirely when no advertiser is configured (<code>src/Engine.ts</code> ; <code>pkg/core/engine/engine.go</code>). With an empty tracker list the propagation step still runs, finds no tracker to resolve peers through, and fails inside the engine's own error handling, so nothing is submitted anywhere (<code>overlay-express/src/OverlayExpress.ts</code> ; <code>sdk/src/overlay-tools/LookupResolver.ts</code>)

Limits, stated plainly:

1. **This paper claims the mechanism, not a peak rate.** The fan-outs and their removal are structural; the throughput ceilings are the engine's verification rate and the plane's per-topic rate, which are their own subjects.
2. **The engine still verifies every arrival.** Delivery through the submit interface costs the same SPV verification a client submission does. The plane removes the N-squared crossings, not the per-host verification.
3. **The loop guard keys on object identity.** The facade will not publish an object whose bytes it has recently published or received from the feed, so a deliberate republish of byte-identical content through the same host within the guard window is suppressed. A republish with a refreshed proof is a different object and passes.
4. **There is one publication path.** A tunnel outage parks client submissions on the facade's retry until the standby path takes over; the local engine has already admitted them. Nothing goes out over HTTPS in the meantime, because the engine no longer propagates.
5. **Neither way of switching propagation off leaves a host newly discoverable.** With no advertiser, the component that maintains the host's SHIP and SLAP advertisements is gone, so the host is reached by configuration rather than by discovery. Keeping the advertiser and emptying the tracker list does not recover that, for a reason worth stating plainly: an advertisement is published by submitting it, and submitting it is the step whose onward propagation the empty tracker list removes. The token is minted and admitted locally, and it goes no further. Advertisements a host published *before* it moved to the plane are held by the hosts that already received them and keep working, so a host that was already discoverable stays so; what it cannot do is become newly discoverable, or refresh what it has. A host that needs discovery should therefore treat it as a separate arrangement rather than as something either posture preserves. Wherever it does advertise, the address to advertise is the facade, because a host that discovers it will submit over HTTPS.
6. **The two ways of switching propagation off are not equally quiet.** With no advertiser the engine skips the step. With an empty tracker list it still enters the step, finds nothing to resolve through, and records the failure through its own error handling: nothing propagates either way, but the second leaves a routine error behind on every submission that admits an output. Treat that line as expected output rather than as a fault, and do not alert on it.

References

BRC-22 Overlay Network Data Synchronization

BRC-24 Overlay Network Lookup Services

BRC-88 Overlay Services Synchronization Architecture (SHIP/SLAP)

BRC-101 Diverse Facilitators and URL Protocols for SHIP and SLAP Overlay Advertisements

BRC-62 Background Evaluation Extended Format (BEEF) Transactions

BRC-135 Multicast Block Header Frame Format

BRC-148 Multicast Shard Domain Partitioning and the BEEF Object Plane

BRC-149 Multicast BEEF Object Frame Format

Overlay services engine and tooling (BSV Association): <https://github.com/bsv-blockchain/ts-stack>

Go overlay services engine (BSV Association): <https://github.com/bsv-blockchain/go-overlay-services>

overlay-bridge (Lightweb Inc.): <https://github.com/lightwebinc/overlay-bridge>

arcade-bridge (Lightweb Inc.): <https://github.com/lightwebinc/arcade-bridge>

teranode-bridge (Lightweb Inc.): <https://github.com/lightwebinc/teranode-bridge>

The overlay object plane: publish BEEF once, and every overlay hears (Lightweb Inc., Paper 006): <https://lbsv.net/papers/overlay-object-plane.pdf>

Arcade v2 on a multicast delivery fabric: one broadcast for every miner, proofs without the firehose (Lightweb Inc., Paper 007): <https://lbsv.net/papers/arcade-on-the-fabric.pdf>

1BSV User Guide (Lightweb Inc.): <https://lbsv.net/papers/1bsv-user-guide.pdf>

1BSV - BSV Layered Multicast - cash moves fast on 1BSV. · lbsv.net

Lightweb Inc. · Jeff Harris · jeff@lightweb.net · github.com/lightwebinc

Unbounded Solutions for a Small World™ · © Lightweb Inc.