INTEGRATION DESIGN · MULTICAST DELIVERY ↔ TERANODE

Pushed delivery into a Teranode cluster over unicast TCP

What 1BSV delivers to a Teranode propagation cluster, how your cluster receives it, and the Teranode-side ingest work it needs.

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

In this document, **we** = the 1BSV delivery edge and **you** = your Teranode cluster.

1. The mental model

You already know Teranode propagation: announce over libp2p, then **pull** the data from the announcing node's Asset service. This document inverts that last step - we **push** the data to you.

In Teranode today a subtree or block is announced over gossip as a hash plus a `DataHubURL`, and every other miner fetches the bytes over unicast HTTP from that node's Asset service. The serving node's egress is therefore **Σ miners × data** - an `O(N)` unicast fan-out that saturates a 10/25 Gbit NIC well before a small-world quorum is reached.

Our fabric replaces that with a **sharded multicast network**. A transaction maps deterministically to a shard group (`groupIdx = f(TxID)`); it is published **once** and the network delivers it to every subscriber. Our emitter cost is `O(1)` - independent of how many miners subscribe.

The one thing multicast **cannot** do is cross into your cluster using our tunnel solution: WireGuard has no multicast, and IPv4 GRE cannot carry IPv6 multicast. The fabric is multicast up to our edge, and we deliver to you over **unicast TCP**, addressed to a list of endpoints you provide. From Teranode's side the whole system reduces to *a reliable TCP stream of push frames - transactions, subtrees, and blocks - arrives at one or more `IP:Port` endpoints in your cluster.*

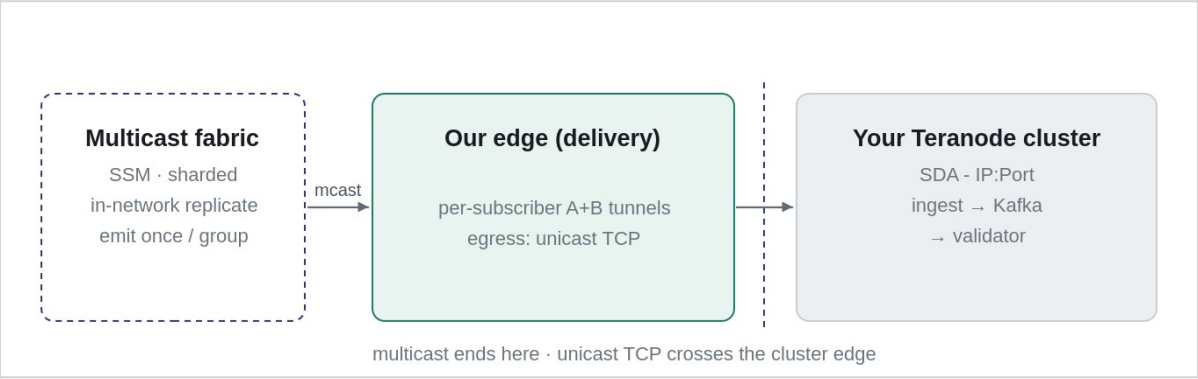

Figure 1 - the demarcation. Multicast lives entirely inside our fabric; we deliver to you over unicast TCP. This replaces libp2p-announce + Asset-pull with pushed delivery, so our emitter cost is `O(1)` in subscriber count.

2. What we deliver - the contract

- **Unicast TCP to one or more destinations.** You provide a list of service delivery addresses (SDAs, each `IP:Port`). We round-robin whole transactions across them over a reliable pool of long-lived connections - keepalive, per-connection reconnect, and redistribution to survivors on failure. A single SDA is a one-element list. Reliability and ordering are TCP's - you don't implement gap detection or retransmission.
- **Sharded tunnels.** A subscription to *k* shards is *k* tunnel pairs. Load distributes by shard, deterministically.
- **A+B for HA.** Each shard's tunnel is a pair - A active, B hot standby - sub-second failover, failover-only.
- **Whole objects only.** You always receive whole objects - never a partial transaction, subtree, or block. Any bundling (BRC-142) or fragmentation (BRC-130) is resolved before delivery.
- **Push frames - one class per lane, no type tag.** Each object class rides its own delivery lane as a self-delimiting push frame with the multicast envelope stripped: transactions as BRC-12 (raw) / BRC-30 (EF), subtrees as **BRC-143**, blocks as **BRC-144**. A receiver reads one class per stream, demultiplexing by lane rather than an in-band tag; identity the multicast header carried - a subtree's merkle root, a block's coinbase - travels in-band, since a push receiver has no pull path to fetch it out of band.

Class	Push frame	Multicast counterpart	Delivery scope
Sharded tx	BRC-12 raw / BRC-30 EF	BRC-124/128, BRC-142	only subscribed shards
Block	BRC-144 (coinbase in-band)	BRC-131 announce	own block lane
Subtree	BRC-143 (root in-band)	BRC-132 subtree data	its own lane (§4)

Two deliberate simplifications for you: because whole objects arrive over reliable TCP, reassembly, gap detection, and retransmission never live in your cluster.

3. How your cluster receives it - bare metal, load balancer optional

3.1 Distribution - round-robin SDAs, direct or via a load balancer

Distribution works either way - point your SDAs directly at per-node receiver endpoints, or at a **load balancer VIP** and let it fan out; both are first-class. You publish a list of SDAs (service delivery addresses), and our edge holds a **reliable pool of long-lived connections** (keepalive, per-connection reconnect) that round-robins whole transactions across them. What makes the load-balancer path work is the delivery shape: that pool opens **multiple long-lived flows, each a distinct 5-tuple** (the source port varies per connection) - exactly the entropy a stateful L4 load balancer, or ECMP, hashes on to spread connections across backends, so a VIP distributes the round-robin flows with no per-frame work. Going **direct** to per-node endpoints removes the extra hop and lands load exactly where your receivers are. For the direct path, prefer **stable node endpoints** (a `NodePort`/host port per node) over ephemeral pod IPs: node churn is rare, and pod churn is absorbed inside each node.

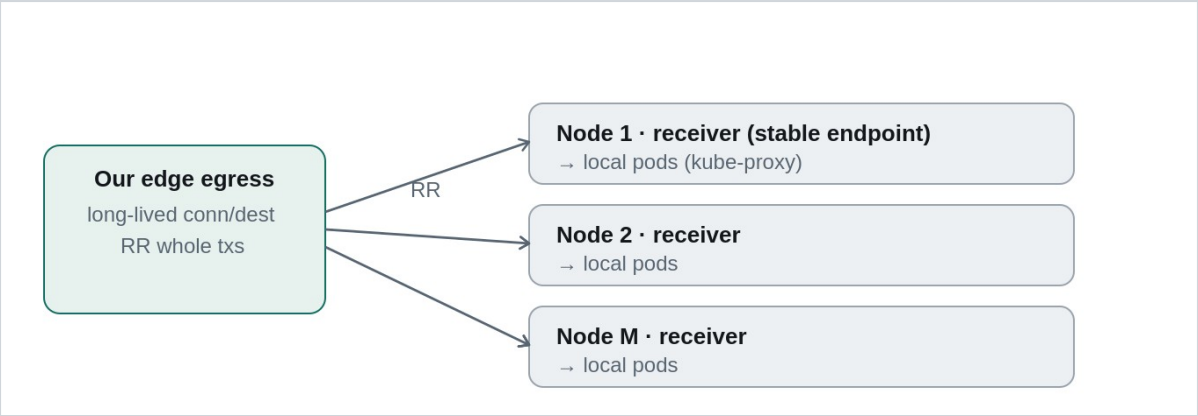

*Figure 2 - distribution, direct-to-node. Your SDA set **is** the load spread, matched to your pods/nodes. Or point the SDAs at a **load balancer VIP**: the round-robin flow variance (distinct 5-tuples) lets an L4 LB spread the connections across backends.*

3.2 Getting the tunnel in, and endpoint HA

The tunnel still has to reach those destinations, and each should survive a node failure - that is where BGP/ECMP and MetalLB earn their place: tunnel ingress and node HA, and optionally the load-balancer spreader of §3.1 itself.

Scenario A - a BGP-capable top-of-rack router. Where the environment gives you a ToR you can peer with, it is the ECMP fabric: cluster nodes run MetalLB speakers, advertise each node destination, and the ToR routes to surviving nodes on failure. Simplest when available - but not present in many dedicated-hosting environments Teranode operators use.

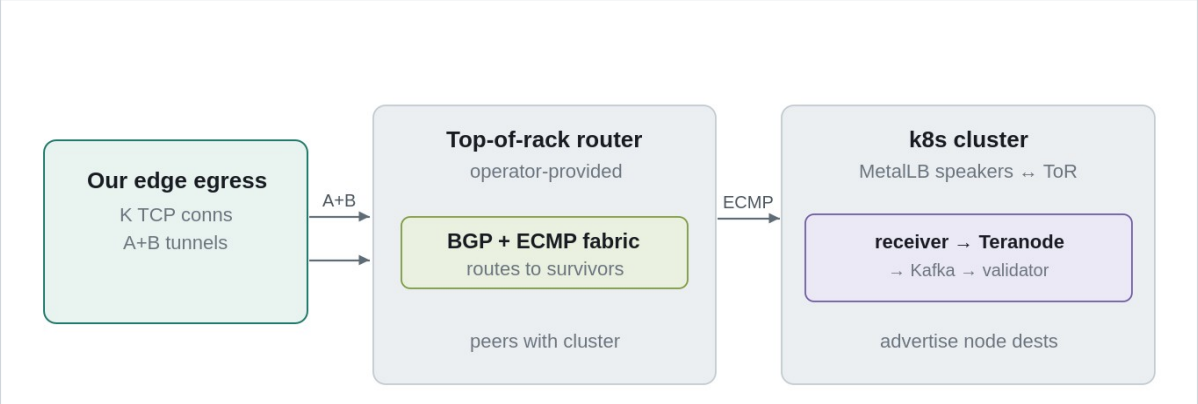

Figure 3 - Scenario A. With a BGP-capable ToR, MetalLB advertises each node destination and the ToR routes to surviving nodes on failure. The VIP is for endpoint HA, not the load spread.

Scenario B - no ToR: land tunnels on commodity BGP servers. Dedicate **one lower-power server - or a pair for HA** - as a tunnel-landing tier: it terminates the WG/GRE tunnels, runs `FRR` or `BIRD`, and peers BGP with the cluster's MetalLB speakers, taking the ECMP role a ToR otherwise fills. The A and B tunnels land on the two servers (or active/standby on one), and BGP converges the landing tier with the cluster - so tunnel ingress and node-endpoint HA live entirely on hardware the operator controls.

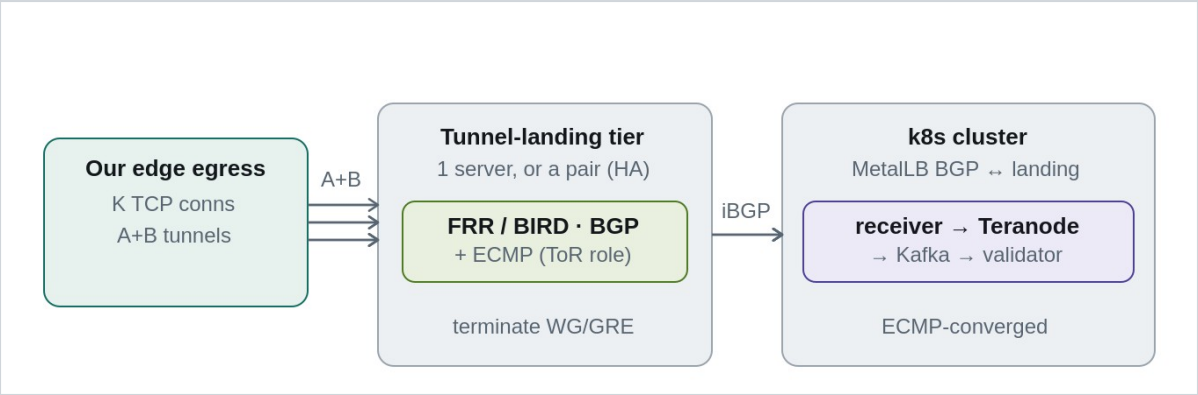

Figure 4 - Scenario B. No ToR. One or two low-power servers terminate the tunnels and run FRR/BIRD, taking the ToR's BGP + ECMP role and converging with the cluster's MetalLB speakers.

4. Content classes and where each meets Teranode

The three classes land at three different ingest points. Only the first exists in Teranode today.

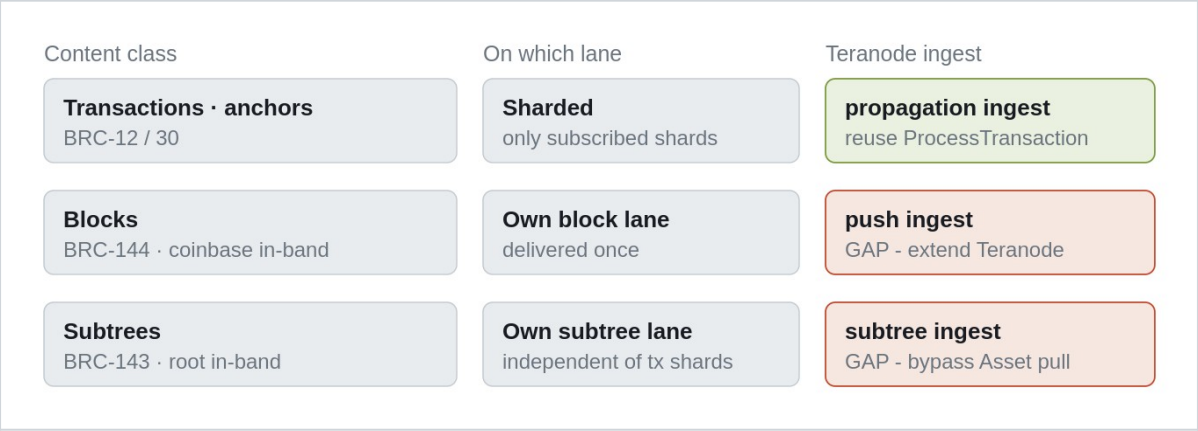

Figure 5 - sharded transactions reuse the existing propagation path; the pushed block frame and the subtree stream need new Teranode ingest.

(a) Sharded transactions. Map cleanly onto the propagation service's tx ingest: the receiver reads a BRC-12 (raw) / BRC-30 (EF) transaction and calls the same path a QUIC/gRPC submitter would - `ProcessTransaction` → validator Kafka topic. Anchors ride this lane too, as extended-format transactions. No new Teranode surface for transport; a thin adapter is enough.

(b) Blocks - BRC-144 on the block lane. A block is delivered as a single **BRC-144** push frame - 80-byte header, block counts, the ordered subtree roots, the **full coinbase carried in-band**, height, and the coinbase merkle path (BUMP) - everything `ProcessBlock` needs with no follow-up fetch. It rides its **own block lane** - a separate round-robin SDA set, delivered once like a transaction (not fanned out to every destination); your cluster propagates the block internally. This is the push counterpart of Teranode's `BlockAnnounce` (BRC-131), which carries only the coinbase *TxID* and relies on an Asset pull for the rest. Two consequences: the coinbase arrives inside the block frame, not as a loose transaction (Teranode rejects a standalone coinbase on the tx path); and a self-

contained pushed block is a new ingest path. Anchors, by contrast, are ordinary EF transactions on the tx lane (§4a).

> **Engineering gap - block ingest.** Accept a pushed BRC-144 block into block validation **without** the Asset pull (no `DataHubURL` round-trip): parse the frame, substitute the in-band coinbase into the first subtree's `0xFF` placeholder, and hand the assembled block to `ProcessBlock`. Must be **idempotent** - A/B failover or a reconnect can re-deliver the same block - so dedup by block-hash.

(c) Subtrees - BRC-143 on their own lane. Subtrees are delivered as **BRC-143** push frames on their own lane, independent of which tx shards you subscribe to. BRC-143 is the push counterpart of the multicast subtree frame (BRC-132): the same ordered node hashes, but with the **merkle root carried in-band** - on the fabric the root rode the frame header and a puller fetched `/subtree/{hash}`, whereas a push receiver has no request path, so the identifying root must travel in the frame. The first subtree of a block carries a 0xFF×32 coinbase placeholder at node 0 (detected by value, no flag); fee/size and the conflict set are omitted - the receiver recomputes them. In Teranode today a subtree is instead learned as a `SubtreeMessage{ Hash, DataHubURL }` over gossip, and `subtreevalidation` fetches `/subtree` and any missing `/txs` from the announcer's Asset service.

> **Engineering gap - subtree ingest.** `subtreevalidation` is currently pull-driven off a `DataHubURL`. It needs a push path: accept a pushed BRC-143 subtree (rebuild via the go-subtree node API - do not byte-splice its on-disk form), and resolve members from the transactions you already hold via your subscribed shards, pulling only genuinely-missing members. Because the subtree lane is independent of the tx shards, member coverage tracks your separate shard subscription - size the pull-fallback accordingly. This removes the Asset round-trip from the steady-state path.

5. Teranode gaps & requirements

ID	Item	Side	Requirement	State
G1	Delivery receiver	Deploy	Terminate the TCP stream, route each object lane (tx / subtree / block) to the right ingest. Our receiver or a thin adapter.	new
G2	Anchor ingest	Teranode	Anchors arrive as EF (BRC-30) transactions on the tx lane, but there is no anchor-transaction concept in Teranode today - build handling end to end.	net-new
G3	Block ingest	Teranode	Accept a pushed BRC-144 block (full coinbase in-band, height, BUMP) into block validation without the Asset pull; substitute the coinbase into the subtree placeholder.	gap
G4	Subtree ingest	Teranode	Accept pushed BRC-143 subtree (root in-band, hashes-only; rebuild via the go-subtree API); resolve members from held shards, pull only missing.	gap
G5	Idempotent ingest	Teranode	Dedup by TxID / subtree-hash / block-hash across repeats (A/B overlap, reconnect).	gap
G6	Reconnect tolerance	both	No contiguous-sequence assumption; tolerate TCP reconnect and A→B failover (bounded, re-delivered gap).	new
G8	Tunnel ingress + endpoint HA	Deploy	Route the tunnel in via a ToR (A) or 1–2 FRR/BIRD landing servers (B). Optional MetalLB per-node VIPs for endpoint HA, not the load spread.	new
G9	Destination set	Deploy	Publish a list of per-node destinations, not a single address; the set is tracked and updated on scale events.	new

Critical path: **G1 + G4** (receiver + subtree ingest) and **G8 + G9** (tunnel ingress + destination list). G3/G5 make block-level operation correct - note **G2 (anchor) is net-new**, not merely a new push path onto an existing concept. In the table, *Side* **Deploy** = the network and receiver you stand up around the cluster; **Teranode** = a change inside the core.

6. Reliability, HA & scale

- **Tunnel path HA.** A+B failover-only, sub-second; on A→B the standby re-establishes the destination connections. Ingest must be reconnect-tolerant (G6), not sequence-contiguous.
- **Destination failure.** A dead destination shows as a write error / RST / keepalive timeout; the connection is dropped from the round-robin, its share redistributed to survivors (any tx → any survivor is safe), and reconnected. Any frame unacked on the broken socket is lost - TCP exposes no app-ack - a bounded loss window. Teranode tolerates it (txs re-propagate); a strict SLA wants either stable per-node VIPs (MetalLB re-homes the endpoint) or app-level ack/replay. Node-granularity destinations shrink this to rare node events, with pod churn absorbed node-locally.
- **Destination-set changes.** You register your current destinations; the set is picked up and connections are added/removed to match. Scale-up/down is a config update, not a redial storm.
- **Scale axes are orthogonal.** Add shards to spread bandwidth across more tunnels/nodes; add pods for node-local processing. Shard growth is coverage-preserving.

Why the numbers favor this

The stream itself - transactions (216 B avg) plus subtree data (32 B/tx) - is **248 bytes/tx**, so its rate scales linearly with TPS. Under the pull model each miner pulls its own copy, so a serving node's egress is **N × stream**: it saturates a 25 GbE NIC at a handful of peers, and by 25M TPS the stream alone exceeds a single NIC. Multicast publishes once and replicates in-network, so source egress stays at the stream rate (and shards across nodes) - subscriber count no longer bounds it.

Aggregate TPS	Stream rate[1]	Pull - peers one 25 GbE node can feed[2]	Multicast source egress
1M	2.0 Gbit/s	~12	2.0 Gbit/s, any N
3M	6.0 Gbit/s	~4	6.0 Gbit/s, any N
5M	9.9 Gbit/s	~2	9.9 Gbit/s, any N
10M	19.8 Gbit/s	~1	19.8 Gbit/s, any N
25M	49.6 Gbit/s	**0 - stream > one NIC**	shard across nodes

1. tx (216 B) + subtree (32 B) per tx. 2. miners a single 25 GbE serving node can feed before its egress (N × stream) saturates. Pull egress rises as O(N), crossing a 10 GbE ceiling at ~5 peers and a 25 GbE NIC at ~13 (at 1M TPS); higher TPS steepens the line. Multicast stays flat - O(1).

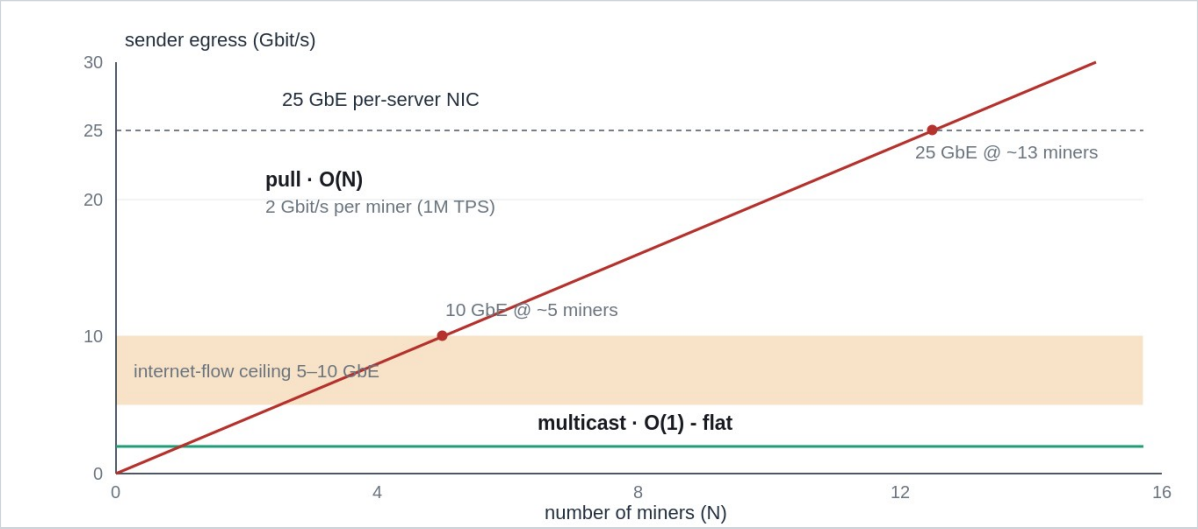

Figure 6 - where the pull model breaks down (at 1M TPS). Pull egress rises as O(N), crossing a 10 GbE flow ceiling at ~5 peers and a 25 GbE NIC at ~13; higher TPS steepens the line. Multicast stays flat - O(1) in subscriber count.

7. Checklist

- **Deploy** - Delivery receiver: TCP terminate, route each lane (tx / subtree / block) to ingest (G1).
- **Deploy** - Publish the per-node destination list; updated on scale events (G9).
- **Deploy** - Tunnel ingress: ToR (A) or 1–2 FRR/BIRD landing servers (B); optional MetalLB per-node VIPs for endpoint HA (G8).
- **Teranode** - BRC-143 subtree push ingest, Asset-pull bypass (G4).
- **Teranode** - Anchor ingest (net-new; EF tx on the tx lane), idempotent dedup (G2, G5).
- **Teranode** - BRC-144 block push ingest (coinbase in-band), `DataHubURL` bypass (G3).
- **both** - Reconnect/failover-tolerant ingest, no sequence assumption (G6).

References - Push frames: BRC-12/30 (tx), BRC-143 (subtree), BRC-144 (block). Multicast counterparts: BRC-124/128/130/131/132/133/134/142. Teranode side: services/propagation, services/subtreevalidation, services/blockvalidation, services/validator.

1BSV Layered Multicast Network - Cash moves fast on 1BSV - `1bsv.net`

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