DESIGN · KUBERNETES · PUSHED DELIVERY

Landing in Kubernetes: pushed delivery into a Teranode cluster with no routers, no BGP, and no extra boxes

Pushed delivery was first specified with a landing tier in front of the cluster: a BGP-capable top-of-rack router, or a pair of commodity servers, terminating the tunnel and forwarding into the Teranode LAN. That shape is right when the cluster sits on a LAN behind routers, and it is not the only shape. This paper describes the Kubernetes-native alternative, in which the consumer end of the tunnel terminates inside the delivery bridge's own pod and the bridge reaches Teranode over ordinary ClusterIP Services. It removes the routers, the BGP session, the load balancer, the inbound firewall rule, and the extra hardware, and it leaves Teranode itself unmodified. It closes with the cases where a separate landing box is still the better answer. Companion paper: *Pushed delivery into a Teranode cluster over unicast TCP*, which specifies the delivery contract this one deploys.

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

Audience: Teranode operators and engineers who run in Kubernetes. **We** = the 1BSV delivery edge; **you** = a Teranode operator.

The short version

- **The landing tier existed for two jobs, and neither survives contact with Kubernetes.** It gave the tunnel a reachable endpoint, and it kept node endpoints alive across failures. A pod that dials outward needs no reachable endpoint, and a pod that dies is rescheduled by machinery you already run.
- **Direction of dial is the whole unlock.** The bridge holds the tunnel from the inside with keepalives, so your cluster needs no public address, no NodePort, no LoadBalancer, and no inbound firewall rule: one outbound UDP flow, and nothing else.
- **There is no multicast anywhere in your cluster.** Multicast ends at our edge. What crosses into the pod is unicast TCP, three streams, one per object class. No Multus, no CNI multicast support, no PIM, no MLD.
- **The bridge is an unremarkable workload.** It runs nonroot and distroless, drops all capabilities, writes nothing to disk, and opens ordinary TCP sockets above 1024. Pod networking is a shipped mode of its Helm chart, not a special build.
- **Retrieval gets faster, not slower.** The bridge announces itself by its own ClusterIP name, so the pulls that follow an announcement stay inside the cluster instead of crossing a LAN to a router.
- **Scale by adding releases, one per shard, never by adding replicas.** The object cache is in-process and each bridge announces itself, so `N` replicas behind one Service send `(N-1)/N` of the cluster's pulls to a replica that never saw the object. The chart ships no autoscaler and refuses the unsafe install. This is a property of the cache backend rather than of the architecture, and §4.1 says what lifts it.
- **One capability is the only real ask, and it is confined to a sidecar.** Kernel WireGuard in a pod network namespace needs `NET_ADMIN`, which the `baseline` and `restricted` Pod Security Standards do not permit. Three clean answers follow, and the third is the original landing box.
- **Teranode changes nothing.** No fork, no new ingest path, no new consensus surface. The integration is a Helm release on your side of the demarcation.

1. The landing tier, and why Kubernetes does not need one

The companion paper puts the tunnel on a landing tier in front of the cluster, and is explicit that BGP and ECMP are there for tunnel ingress and node endpoint high availability, not for spreading load. That is worth restating, because it tells you exactly what to look for when the deployment changes shape: if something else already solves those two jobs, the routers have no remaining work.

Kubernetes solves both, and it solves them with machinery you already operate.

The landing tier's job	Why a router answered it	What answers it in Kubernetes
Give the tunnel a reachable endpoint	The cluster had no dialable address	The pod dials outward; no inbound reachability is needed at all
Survive a node failure	BGP withdraw, or a re-homed VIP	The pod is rescheduled and re-dials; no routing convergence in the path
Spread delivery across nodes	Round-robin to per-node endpoints	Shards are the spread: one release per shard tunnel, scheduled across nodes
Serve retrieval pulls to the cluster	A retrieval plane on the router	A ClusterIP Service, so pulls never leave the cluster

Nothing in that table is a new mechanism; it is the same four jobs, answered by a scheduler instead of a routing protocol.

One design rule from the fabric side is unchanged and worth saying plainly, because it is the thing people expect us to bend: consumers never join fabric multicast. There is no fabric receiver here, no `(S,G)` join, no PIM or MLD, and no gap machine running in your cluster. The tunnel carries what it always carried, which is unicast TCP.

2. What actually crosses the boundary

Multicast cannot cross into your cluster, and we do not pretend otherwise: WireGuard carries no multicast, and IPv4 GRE cannot carry IPv6 multicast. So the fabric is multicast up to our edge, and delivery to you is unicast TCP.

From your side the whole system reduces to three TCP streams, one per object class (transactions, subtrees, blocks), arriving at listeners the bridge already runs. Reliability and ordering are TCP's, so gap detection and retransmission never live in your cluster. You always receive whole objects, never a partial transaction, subtree, or block: fragmentation and bundling are resolved before delivery.

That is the entire protocol surface. Everything else in this paper is deployment.

3. The shape: one pod, dialling out

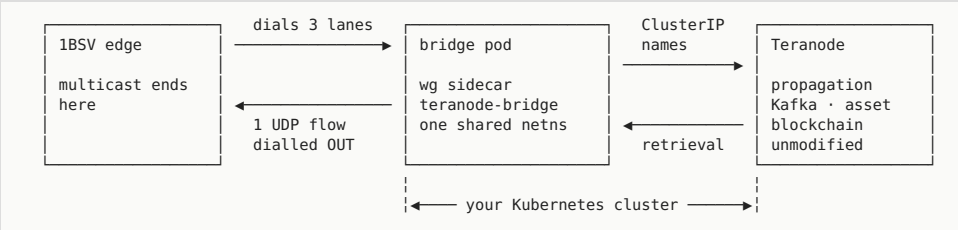

Figure 1 - the pod-attached shape. The sidecar dials outward and holds the session; both containers share one network namespace, so the bridge's lane listeners become reachable the moment the session is up.

Four things happen, in this order.

The sidecar dials out and holds the session. WireGuard is peer to peer, so either end may initiate. Having the consumer initiate, with a persistent keepalive, is what removes every inbound requirement: no public address on the cluster, no NodePort, no LoadBalancer, no firewall exception.

The two containers share a network namespace. That is ordinary pod behaviour, and it is doing real work here: once the session is up, the overlay address belongs to the pod, so the bridge's lane listeners are reachable on it with no routing, no port mapping, and no config coupling between the two containers. Bind the lanes to the wildcard address rather than pinning one, and container start order stops mattering; a re-handshake needs no restart.

The bridge talks to Teranode over ClusterIP Services. Propagation, Kafka, the blockchain service, and the asset service are reached by their ordinary names. Nothing lands on a Teranode node, and nothing about Teranode changes.

Announcements point at the bridge itself. The bridge advertises its own Service name, so when `subtreevalidation` and `blockvalidation` fetch the bytes that were already pushed, the fetch is a short hop inside the cluster. On a landing tier that same pull crossed a LAN. This is the one place where the Kubernetes shape is not merely equivalent but better.

4. Scaling and failover: releases, not replicas

This is the one part of the deployment that does not behave the way Kubernetes instincts expect, so it is worth thirty seconds of your attention rather than a discovery in production.

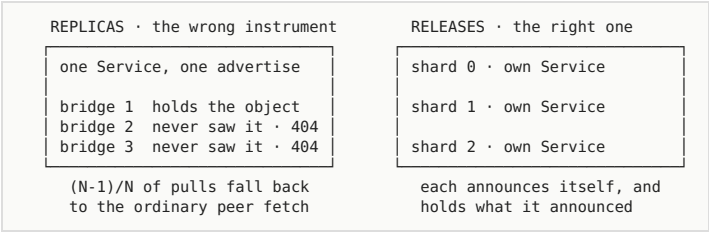

Figure 2 - why replicas fragment the cache. A pushed object lives only on the bridge that received it, and that bridge stamps itself into the announcement.

Two properties make replicas the wrong instrument:

- **The object cache is in-process, and each bridge announces itself.** Put `N` replicas behind one Service and `(N-1)/N` of the cluster's pulls land on a replica that never saw the object. The result is a 404 and a fall back to the ordinary peer announce-and-pull: slower, not lost, but it quietly deletes the benefit you deployed for.
- **Exactly one bridge per cluster may hold the reverse-path submitter role.** Two holders publish every locally produced subtree and block twice. The chart fails the install rather than creating that silently.

So the chart ships no autoscaler, on purpose. **Scale by adding releases, one per shard tunnel, each with its own Service and its own advertised URL.** That is the fabric's shard axis expressed as ordinary Kubernetes objects, and it maps cleanly onto whatever you already use to template releases.

Failover is pleasantly boring. A second release with the submitter role disabled is a hot spare: it connects, runs its origin filter, and publishes nothing. Promotion is hands-free when you point its submitter probe at the primary's readiness endpoint (a few consecutive failures promote it in place, with no restart and no operator); flipping the flag by hand remains the fallback. Demotion is deliberately slower than promotion, so a flapping primary cannot flap the role. Add

pod anti-affinity so the pair lands on different nodes. When a pod dies, it is rescheduled and dials out again; ingest is already reconnect tolerant and assumes no contiguous sequence, so a reschedule is an ordinary event rather than an incident. There is no broker re-peer, no BGP convergence, and no BFD timer in that path.

4.1 Why the limit exists, and what lifts it

Both properties above follow from one implementation choice: the object cache lives inside the bridge process. Give more than one bridge a cache they can both read and the first property goes away, which is exactly the plane split our landing-tier design describes (ingest, object cache, and retrieval as separable planes, with the announced URL always pointing at "the retrieval plane", whatever answers it).

That backend, with cache replication between bridges and a retrieval path that can serve a peer's object rather than answering 404, belongs to the commercial bridge, which is **included with an active delivery subscription** rather than sold separately. The open bridge keeps the in-process cache and the one-release-per-shard shape, and it is the version this paper describes throughout.

The distinction worth being precise about is that the two differ in **how often you exercise the fallback, not in whether you are safe**. A pull that misses returns 404 in both, and 404 sends the cluster to its ordinary peer announce-and-pull. Nothing is lost either way; the shared cache buys you the latency you deployed for during the window when a bridge has just been replaced.

5. What to check before you start

Four checks, and only the second one ever blocks anyone.

#	Check	If it is not available
1	The `wireguard` module on your nodes (in the mainline kernel since 5.6)	Use the userspace implementation; slower, and fine at current rates
2	`NET_ADMIN` permitted for the sidecar	See §6
3	MTU headroom for the tunnel	Set the WireGuard interface MTU explicitly (1420 on a 1500-byte IPv6 underlay)
4	Outbound UDP to our edge permitted	Allow the single outbound flow

On the second: kernel WireGuard has to create an interface inside the pod's network namespace, and that needs `NET_ADMIN`. The `baseline` and `restricted` Pod Security Standards permit adding only `NET_BIND_SERVICE`, so this is a real policy question rather than a formality, and it is better raised early than discovered at install time.

The good news is that the capability is confined. It belongs to the tunnel sidecar alone; the bridge container keeps its dropped capabilities, its nonroot user, and its read-only root filesystem. The usual answer is to run the bridge in its own namespace that enforces the `privileged` profile, which is a small, auditable, well-understood exception rather than a cluster-wide loosening.

Two smaller notes that save time later. Lane traffic arrives on the tunnel interface inside the pod's namespace rather than through the CNI interface, so CNI network policy does not see it; restrict the lanes with the tunnel's own allowed-IPs instead. Policy for the retrieval plane and the metrics endpoint works normally, because the cluster and your monitoring reach the pod through the CNI. And give the pod requests and limits: at present rates a bridge and a validator coexist happily, but they do compete, and a small dedicated node pool is cheap insurance before that matters.

6. When a landing box is still the right answer

We would rather deploy into your cluster, and we would rather say plainly when that is the wrong call.

1. **Policy forbids the capability, and a namespace exception is not available.** Some organisations will not grant `privileged` enforcement to any namespace, and that is a legitimate position rather than an obstacle to argue with.
2. **You would rather keep WireGuard on the node.** A systemd-managed tunnel with the overlay routed to the bridge's Service needs no privileged pod at all. The cost is node configuration that your Kubernetes team does not manage from Helm, which some teams consider worse than the exception and some consider better.
3. **Teranode is not the only thing behind the tunnel**, or the cluster genuinely sits on a LAN behind routers you run already. Then the companion paper's landing tier is not a fallback; it is simply the right topology, and BGP is earning its place.

In all three cases the delivery contract is unchanged: the same three lanes, the same whole objects, the same bridge. Only the question of where the tunnel lands has a different answer, which is the point of separating the two.

7. Assumptions and limits

Quantity	Value	Basis
WireGuard encapsulation	80 B over an IPv6 underlay	**structural**: 40 B IPv6 + 8 B UDP + 16 B header + 16 B tag
Tunnel interface MTU	1420 on a 1500 B underlay	**derived**: 1500 less the 80 B above
Kernel WireGuard availability	mainline since Linux 5.6	**stated**
Capabilities addable under `baseline`	`NET_BIND_SERVICE` only	**verified**: Kubernetes Pod Security Standards
Pull miss rate under `N` replicas	`(N-1)/N`	**structural**: one in-process cache per replica, uniform Service load balancing
Bridge retrieval-plane coverage	93.6 % of statements	**measured**: the bridge's own test suite
Bridge throughput to a sink	~1.48 M tx/s	**measured**: mock sink, no cluster behind it

Three limits, stated plainly:

1. **The pod-attached shape is specified and deployable, and the proving runs were done with the tunnel on landing hosts.** The bridge, the lanes, the announce path, the retrieval plane, and the reverse path are identical in both; what differs is where the tunnel terminates. Treat the outbound-dial path as new deployment surface and drill a pod reschedule before you rely on it.
2. **The capability question is a policy matter, not a technical one.** We have no way to know in advance which answer in §6 fits your organisation, and all three are supported.
3. **This paper describes the open bridge.** Where the commercial bridge differs (§4.1) it is stated inline rather than left for you to discover. The difference is scale-out behaviour, never a safety property: every failure path in both ends at the same 404 and the same fall back to your ordinary peer fetch.

References

Pushed delivery into a Teranode cluster over unicast TCP (Lightweb Inc.) — https://1bsv.net/papers/pushed-delivery-teranode.pdf

The Asset service at scale: why fronting `DataHubURL` with a CDN extends the gossip problem instead of solving it (Lightweb Inc.) — https://1bsv.net/papers/asset-pull-and-the-cdn-reflex.pdf

`teranode-bridge`, the delivery bridge and its Helm chart — https://github.com/lightwebinc/teranode-bridge

Kubernetes Pod Security Standards — https://kubernetes.io/docs/concepts/security/pod-security-standards/

WireGuard protocol and message framing — https://www.wireguard.com/protocol/

BRC-143 Subtree Data Frame Format (the subtree push frame)

BRC-144 Block Frame Format (the block push frame)

Teranode — https://github.com/bsv-blockchain/teranode

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

---

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