DESIGN · SHARDING · BRC-129

Shard bit extraction: the top bits of the TxID are the group

Every component in the multicast pipeline must map a transaction to an IPv6 multicast group at line rate, independently, and agree on the answer. This paper explains the one-line derivation that does it: read the first four bytes of the TxID as a big-endian word and keep the top `shardBits` bits. It walks the arithmetic bit by bit, shows why keeping the top bits beats a modulo (widening splits groups instead of reshuffling them), places the result inside the BRC-129 address layout, and closes with the BRC-148 generalisation that runs the same rule on every object plane.

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

Audience: multicast fabric operators and protocol implementers. **We** = the 1BSV fabric; **you** = an operator or implementer. All widths and defaults are verified against the open-source implementation.

The short version

- **The whole derivation is one line.** `groupIndex = binary.BigEndian.Uint32(txid[0:4]) >> (32 - shardBits)`: one 4-byte load, one shift. No division, no state, no coordination.
- **`shardBits` picks the group count: 2^N groups.** The components default to `-shard-bits 2` (4 groups); BRC-129 caps the transaction plane at 12 (4,096 groups), a cap set by the address plan, not by the arithmetic.
- **Top bits make widening a split, not a reshuffle.** Raising the width by one splits every group `g` into exactly two children, `2g` and `2g+1`. Subscribers add joins; no existing subscription becomes wrong. A modulo resize remaps roughly K/(K+1) of all traffic (80% at K = 4).
- **The index is the low 16 bits of the group address.** `FF05::B:IDX`; shard indices own `0x0000`–`0x0FFF`, object planes and the control plane own the rest.
- **Endianness is load-bearing.** The big-endian read makes `txid[0]` the most significant byte, so the group index is literally the leading bits of the TxID as it appears in the frame. At width 12 the group is the first three hex digits.
- **One rule runs every plane (BRC-148).** Each object plane applies the same top-bits rule to its own 32-byte key at its own width, plus a plane base. `shardBits = 0` is valid there and collapses a plane to one group.
- **Uniformity is statistical, not adversarial.** TxIDs are double-SHA256 outputs, so honest load spreads evenly across the 2^N groups; a sender who grinds TxIDs can aim at a group, and this function does not defend against that.

1. The derivation, bit by bit

Sharding is a sorting problem: every transaction must land in one of 2^N bins, and every component that ever touches the TxID (proxy, listener, retry endpoint, producer) must pick the same bin without asking anyone. The rule that does it:

```
groupIndex = binary.BigEndian.Uint32(txid[0:4]) >> (32 - shardBits)
```

Take the first four bytes of the 32-byte TxID field exactly as it appears in the frame, read them as one big-endian 32-bit word, and shift that word right until only the top `shardBits` bits remain. The implementation also masks with `(1 << shardBits) - 1`; for widths in `[1, 12]` the shift alone already bounds the result, and the mask earns its keep at width 0 (§5).

Worked at the cap width, `shardBits = 12`, for a TxID beginning `a3 f7 c2 e1`:

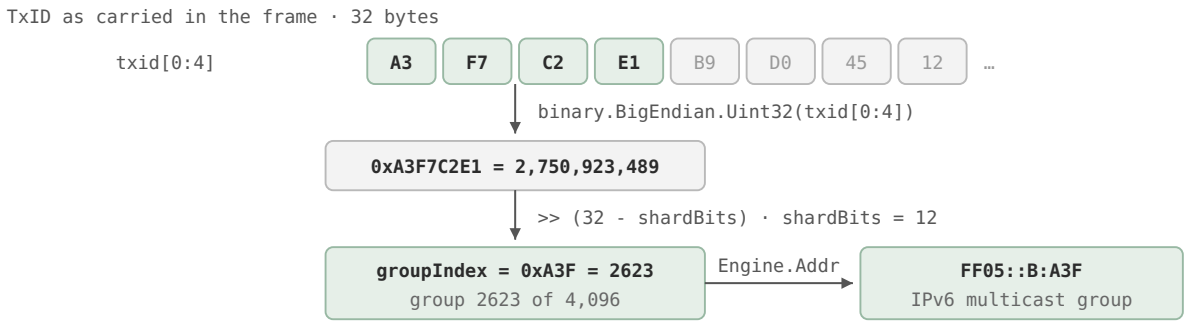

Figure 1 - the full path from TxID bytes to group address: four bytes become one big-endian word, the shift keeps the top 12 bits, and the result drops into the low 16 bits of the IPv6 group.

The shift count is `32 - shardBits`: of the 32 bits just assembled, the low `32 - shardBits` fall off the right edge and the top `shardBits` survive:

Figure 2 - the word 0xA3F7C2E1 at width 12: the kept bits read 101000111111, which is 0xA3F, group 2623.

Twelve bits is three hex digits, so at the cap width you can read the group straight off the front of the TxID: a transaction whose hex begins `a3f...` is in group `0xA3F`. At the default width 2 the group is the top two bits of the first byte: `A3` is `1010 0011`, so the group is `10` binary, group 2 of 4.

The same TxID across every width shows the deeper structure: the index at width N is the first N bits of the TxID. Widening never moves a bit; it only reveals more of them.

shardBits	Groups	Shift	Index (binary)	Index
1	2	`>> 31`	`1`	1
2	**4**	`>> 30`	`10`	**2** (default width)
3	8	`>> 29`	`101`	5
4	16	`>> 28`	`1010`	10 = `0xA`
8	256	`>> 24`	`10100011`	163 = `0xA3`
12	**4,096**	`>> 20`	`101000111111`	**2623 = `0xA3F`** (cap)

1. All rows derive from the same word 0xA3F7C2E1; the binary column is the word's leading bits.

Each index in the table is a prefix of the one below it, and that prefix property is the entire design.

2. Why the top bits and not a modulo

The obvious alternative deserves its strongest form. `idx = word % K` supports any group count, not just powers of two: a fleet of six proxies gets exactly six groups. On hash input it is exactly as uniform as the top-bits rule, and on modern cores the division is cheap. If the group count never changed, modulo would be the right answer.

But the group count is the one thing guaranteed to change: growing traffic is the reason a sharded fabric exists. Resizing a modulo from K to K+1 remaps almost every key (about K/(K+1): 80% of traffic moves at K = 4, 92% at K = 12). Every subscriber's membership is invalidated at once; the fleet needs a flag day. The top-bits rule turns the same event into a split:

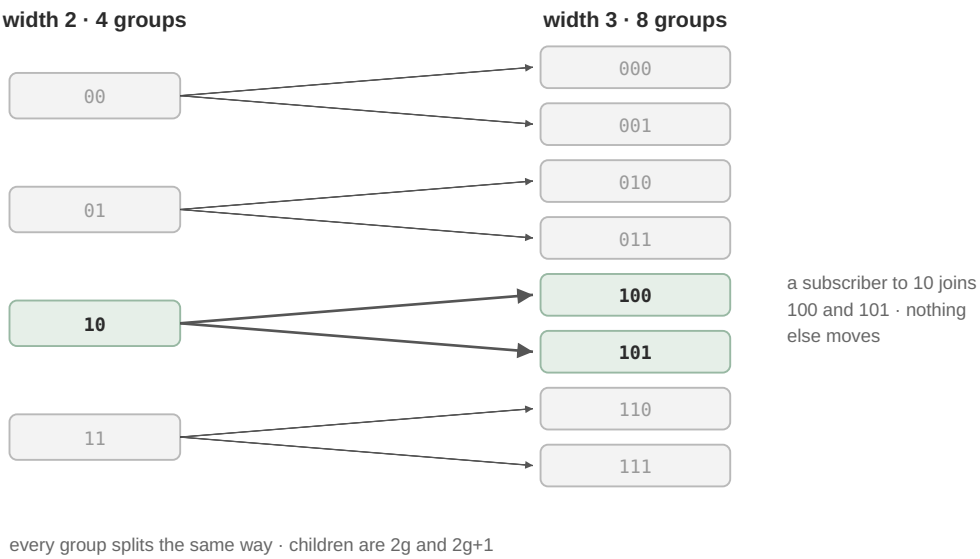

Figure 3 - widening from 2 to 3 bits: group 10 splits into children 100 and 101. A subscriber to 10 joins the two children; every other membership is untouched.

This is consistent hashing reduced to its cheapest form: the index at width N is a prefix of the index at width N+1, so raising the width refines the partition instead of replacing it. Two properties follow:

- **Subscriptions survive widening.** A consumer of group `g` becomes a consumer of `2g` and `2g+1`: two joins, zero leaves, no invalidated state anywhere else in the fleet.
- **A parent is a contiguous index range.** The children of `g` are adjacent (`2g`, `2g+1`), so any group at any width covers one unbroken block of indices. Contiguous bands are what plane reservations and band joins are built on (§5). Masking the low bits instead (`word & (2^N - 1)`) also splits cleanly, but its children land at `g` and `g + 2^N`, scattered across the space, and the index no longer reads off the front of the TxID.

What modulo legitimately buys is the exact match to a non-power-of-two fleet, and we give that up deliberately: a width-N deployment balances 2^N groups across whatever hardware exists by assigning each node a set of groups, and sets of groups are exactly what the join model makes free.

3. Where the index lands

The derived index is not an abstract bucket number; it is the low 16 bits of the IPv6 multicast group address the frame is transmitted to:

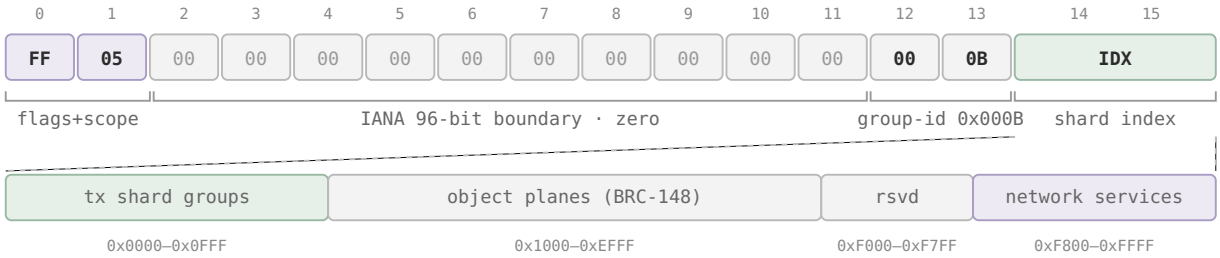

Figure 4 - BRC-129 address layout and the 16-bit index space (zone bar not to scale). The derivation fills IDX; everything else is fixed by configuration and by the IANA Bitcoin allocation FF0X::B (group-id 0x000B).

The 16-bit index space is zoned:

Index range	Zone
`0x0000`–`0x0FFF`	Transaction shard groups (this derivation)
`0x1000`–`0xEFFF`	Object planes; domain = high nibble (BRC-148)
`0xF000`–`0xF7FF`	Reserved; never joined, never transmitted to
`0xF800`–`0xFFFF`	Network services: beacon, manifest, announcements

The 12-bit cap on the transaction plane is address zoning, nothing more: 2^{12} groups exactly fill the `0x0000`–`0x0FFF` zone.

The scope and source-mode variants (`FF05` site ASM, `FF35` site SSM, `FF3E` global SSM) change only the top bytes of the address; the group-id and the derived index ride through every mode unchanged.

4. Endianness is load-bearing

`binary.BigEndian.Uint32` makes `txid[0]`, the first byte on the wire, the most significant byte of the word, and therefore the byte that drives the shard decision. A little-endian read would hand that role to `txid[3]`:

TxID begins	BE word	BE group (width 12)	LE word	LE group
`a3f7c2e1`	`0xA3F7C2E1`	2623 = `0xA3F`	`0xE1C2F7A3`	3612
`00112233`	`0x00112233`	1 = `0x001`	`0x33221100`	818
`deadbeef`	`0xDEADBEEF`	3562 = `0xDEA`	`0xEFBEADDE`	3835
`ffffffff`	`0xFFFFFFFF`	4095 = `0xFFF`	`0xFFFFFFFF`	4095

Both reads are equally uniform on hash input; only the big-endian read makes the group index equal the TxID's leading bits.

Uniformity is therefore not the argument: a hash is uniform from either end. The argument is legibility and determinism. Big-endian is network byte order; it keeps the index equal to the wire prefix, so you can read the group off the first hex digits of any TxID in a log line, and every implementation in every language lands on the same convention without a footnote. Since all components must agree on both the width and the byte order to interoperate at all, the fabric also announces each participant's `shard_bits` on the wire (BRC-139 manifests), so a divergent component is observable instead of silently mis-sharded.

5. One rule, every plane

BRC-148 partitions the index space into object planes and runs this same derivation inside each one:

```
planeBase(domain) = domain << 12
IDX               = planeBase + ((key[0:4] as uint32 BE) >> (32 - bits))
```

The transaction plane is domain `0x0` with key = TxID and width capped at 12: exactly the rule of §1, unchanged. The BEEF object plane is domain `0x1` with key = TopicID and its own width. Each plane chooses its width independently, and the split property of §2 holds inside each plane.

`shardBits = 0` is legal on object planes and is the degenerate case the implementation's mask exists for: shifting a 32-bit word right by 32 yields zero (defined behaviour in Go), the mask is zero, and every key maps to `planeBase` itself. One group carries the whole plane: the low-membership posture a plane starts in, trading fabric selectivity for minimal group-membership state. Widening it later is, again, only splits.

A plane wider than 12 bits reserves `ceil(2^bits / 4096)` contiguous `0x1000`-slots (its **SlotSpan**), and no plane's range may reach the control zone: `planeBase + 2^bits ≤ 0xF800`. Both constraints are direct consequences of §2's contiguity: a plane is always one unbroken band of indices, so reserving it and joining it is range arithmetic.

6. Assumptions and limits

Quantity	Value	Basis
Derivation	`BigEndian.Uint32(txid[0:4]) >> (32 - N)`, masked	**verified**: `shard-common shard/shard.go`
Transaction-plane cap	N ≤ 12 (4,096 groups)	**verified**: BRC-129 zone `0x0000`–`0x0FFF`
Default width	`-shard-bits 2` (4 groups)	**verified**: proxy, listener, and retry-endpoint config defaults
BEEF-plane default	`-beef-shard-bits 0` (1 group)	**verified**: same config surfaces
Group-id	`0x000B` (`FF0X::B`)	**verified**: IANA IPv6 multicast registry, Bitcoin allocation
Per-group share of load	≈ 1/2^N	**structural**: TxID is a double-SHA256 output
Split on widening	`g` becomes `2g`, `2g+1`	**structural**: the index is a key prefix
Modulo resize remap	≈ K/(K+1) of keys	**structural**: residues mod K and K+1 nearly independent

Limits, stated plainly:

1. **Uniformity is statistical, not adversarial.** TxIDs are chosen by whoever builds the transaction, and grinding contents until the leading bits land in one group is cheap. The derivation does not defend against deliberate concentration; ingress admission and rate control are the defence, and they live outside this function.
2. **Group counts are powers of two; fleets are not.** Modulo would match a six-node fleet exactly. We instead assign each node a set of groups, which restores balance but adds one indirection an operator has to manage.
3. **Agreement is operational, not enforced.** A component running the wrong width or byte order mis-routes silently at the sharding layer. BRC-139 manifests make the divergence visible on the wire, but nothing in the derivation itself rejects a misconfigured peer.
4. **Widening is cheap, not free.** Splits preserve subscriptions, but consumers still perform the new joins, and on object planes a width change is a generation transition (BRC-148) with its own choreography.

References

BRC-129: Multicast Group Addressing — https://github.com/bsv-blockchain/BRCs/blob/master/transactions/0129.md

BRC-139: Shard Manifest — https://github.com/bsv-blockchain/BRCs/blob/master/transactions/0139.md

BRC-148: Multicast Shard Domain Partitioning and the BEEF Object Plane — https://github.com/bsv-blockchain/BRCs/blob/master/transactions/0148.md

IANA, *IPv6 Multicast Address Space Registry* — https://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xhtml

Reference implementation: `shard/shard.go`, `shard/plane.go` — https://github.com/lightwebinc/shard-common

---

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

---

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

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