LVS — Protocol Specification (EN)

Version 1.0 — Final, Production-Ready

---

1. Purpose of This Document

This Protocol Specification defines the operational rules, data structures, communication flows, and behavioral requirements of the **LVS Autonomous Value Layer**. It describes how nodes interact, how value is balanced, how consensus emerges, and how the system maintains continuity without validators, miners, governance, or identity.

This document is authoritative for all implementation work.

---

2. Protocol Principles

The LVS protocol is built on five principles:

1. **Autonomy** — No human decision-making or validation.
2. **Lightweight** — Nodes must run on minimal hardware.
3. **Continuity** — System exists as long as at least one node remains.
4. **Emergent Consensus** — State converges through drift, not authority.
5. **Value Integrity** — User-driven contribution defines value.

---

3. Node Requirements

3.1 Hardware Requirements

- RAM: $\geq$ 10 MB
- CPU: low-power compatible
- Network: intermittent connectivity allowed

3.2 Software Requirements

- LVS Node Runtime (implementation-dependent)
- Transport layer module (WebRTC/UDP/TCP)
- VSE (Value State Engine) execution capability

---

4. Node Identity Model

LVS uses a **Zero-Identity Model**: - No accounts - No keys - No signatures - No persistent identity

Nodes are recognized only by: - ephemeral session IDs - dynamic connection routes - entropy exchange frequency

This eliminates identity-based attack vectors entirely.

---

5. State Model

LVS maintains a distributed, sharded state composed of:

5.1 Value Units (VU)

`VU := contribution_value(user_action)`

Positive user activity generates VU. No minting, no inflation.

5.2 Trust Credits (TC)

`TC := stability_score(node_behavior)`

TC influences drift correction.

5.3 Drift Coefficients (DC)

`DC := weight(VU, TC, entropy_variation)`

These coefficients define how each node adjusts its partial state.

5.4 State Shards

Global state S is partitioned:

```
S = {S1, S2, S3, ... Sn}
```

Each shard is redundantly stored across nodes.

---

6. Communication Model

Nodes communicate via **Entropy Packets (EP)** and **State Diff Messages (SDM)**.

6.1 Entropy Packet (EP)

Carries unpredictable micro-state:

```
EP := {
  entropy_vector: [e1, e2, ... ek],
  timestamp,
  local_node_load,
  optional_noise_flags
}
```

6.2 State Diff Message (SDM)

Sent after drift correction:

```
SDM := {
  shard_id,
  diff_vector,
  drift_weight,
  cycle_id
}
```

6.3 Transport Requirements

- messages must tolerate loss
- ordering is NOT guaranteed
- nodes must handle duplicates

---

7. Consensus Mechanism — Drift-Based Consensus (DBC)

DBC replaces traditional consensus models.

7.1 Drift Function

Each node applies:

```
new_state = old_state + f(EP, DC, local_weights)
```

Where `f()` is a deterministic correction function.

7.2 Drift Cycle (DCY)

```
1. Receive EP
2. Compute drift_weight
3. Update local shard
```

```
4. Broadcast SDM
5. Integrate SDMs from peers
```

7.3 Global Convergence

Convergence is emergent: - No leader - No authority - No finality voting

Nodes statistically converge through thousands of micro-corrections.

---

8. VaultGuard Rules

VaultGuard enforces safety invariants.

8.1 Value Protection Rule

```
VU >= 0 must always hold.
```

If drift attempts to push state negative, VG clamps the correction.

8.2 Catastrophic Loss Prevention

If sudden extreme VU drops occur:

```
trigger_recovery_protocol()
```

8.3 Malicious Drain Detection

Nodes monitor: - abnormal diff patterns - high-frequency drain attempts - entropy anomalies

Suspicious flows are down-weighted.

---

9. Node Lifecycle Specification

9.1 Initialization

- Obtain peers
- Retrieve minimum state
- Perform initial drift

9.2 Active Mode

Loops drift cycles continuously.

9.3 Low-Power Mode

Node reduces frequency of drift when: - low CPU - low battery - weak network

9.4 Exit

Node leaves silently — no penalties.

---

10. Error Handling

Nodes must gracefully handle: - EP/SDM loss - repeated packets - incomplete state - entropy oversaturation - drift divergence

Nodes correct themselves via: - redundancy - rebalancing shards - random fallback connections

---

11. Security Specification

11.1 No Identity Attacks

No keys → no identity theft.

11.2 No Governance Capture

No voting → no takeover.

11.3 No 51% Attacks

No authority → impossible to control the majority.

11.4 Drift Integrity

Drift corrections must satisfy:

```
abs(diff_vector) < max_allowed_shift
```

11.5 Node Distribution

Optimal when:

```
N >= 300 global nodes
```

But LVS remains functional at any $N \geq 1$.

---

12. Performance Constraints

Latency tolerance: high

Packet loss tolerance: high

Bandwidth usage: low (<50 KB/min avg)

CPU load: minimal

---

13. Implementation Guidelines

Recommended languages:

- Rust
- Go
- C++ (micro-devices)
- JS/TS (browser nodes)

Reference modules:

- entropy_engine
- drift_core
- state_shard
- vaultguard
- transport_layer
- node_runtime

---

14. Compliance Requirements

To be considered an LVS-compliant node, implementation must: - follow drift formula - support EP and SDM messages - enforce VaultGuard invariants - support redundancy sync - maintain partial shard storage

---

15. Final Notes

This specification is the baseline for all LVS development. Any future extensions must not break: - autonomy - drift consensus - zero-identity model - value protection invariants

LVS is engineered for long-term durability and global resilience.