LVS — Node Implementation Blueprint (EN)

Version 1.0 — Final, Implementation-Ready Specification

---

1. Purpose of This Document

This blueprint defines how to **implement an LVS-compliant node** in practice. It describes: - software architecture, - core modules and their responsibilities, - data structures, - communication logic, - pseudocode for all critical processes, - lifecycle management, - performance and safety rules.

This is the reference for all future developers, contributors, and auditors.

---

2. Node Architecture Overview

Every LVS node consists of six core modules:

1. **Runtime Engine** — manages cycles, scheduling, lifecycle.
2. **Transport Layer** — WebRTC/UDP/TCP communication engine.
3. **Entropy Engine** — generates & processes entropy packets.
4. **Drift Core** — implements DBC drift logic.
5. **State Shard Manager** — stores & manages partial state.
6. **VaultGuard Module** — enforces safety invariants.

The architecture is highly modular so it can run on: - browsers (JS/TS), - mobile apps, - microservers (Go/Rust), - IoT hardware.

---

3. Node Module Responsibilities

3.1 Runtime Engine

The master controller of the node. Manages: - initialization, - peer discovery, - drift cycles, - scheduling, - error handling, - low-power mode.

Key Functions:

```
start_node()
run_cycle()
enter_low_power()
shutdown()
```

3.2 Transport Layer

Handles all networking. Supports: - WebRTC (browser) - UDP or lightweight TCP (servers)

Message Types:

- Entropy Packets (EP)
- State Diff Messages (SDM)
- Node beacons

Requirements:

- tolerate packet loss
- tolerate reordering
- duplicate filtering

Key Functions:

```
send_EP(ep)
receive_EP()
send_SDM(sdm)
receive_SDM()
discover_peers()
```

3.3 Entropy Engine

Generates, validates, and normalizes entropy.

Key Data:

```
entropy_vector: [e1, e2, ... ek]
timestamp
node_load
```

Key Functions:

```
generate_entropy()
normalize_entropy(E)
calculate_load()
```

Entropy is the source of emergent global convergence.

3.4 Drift Core

Implements Drift-Based Consensus (DBC). Applies drift corrections using: - EP (entropy packets), - SDM (peer diffs), - local value state, - drift coefficients.

Key Functions:

```
compute_drift_from_entropy(EP)
compute_drift_from_diffs(SDMs)
apply_drift()
compute_local_diff()
```

3.5 State Shard Manager

Maintains distributed partial state.

State Structure:

```
VU: map<user_id?, value>
TC: map<node?, trust>
DC: drift_coefficients
entropy_cache
```

Even though LVS uses zero-identity, internal shard keys are ephemeral and local-only.

Key Functions:

```
load_shard()
store_shard()
rebalance_shards()
merge_diff(dv)
```

3.6 VaultGuard Module

Enforces system safety invariants.

Protects Against:

- negative VU values,
- malicious drains,
- extreme anomalies.

Key Functions:

```
validate_drift(D)
correct_anomaly(D)
trigger_recovery_mode()
```

---

4. Complete Node Lifecycle

4.1 Initialization Phase

```
start_node():
    peers = discover_peers()
    shard = load_minimum_state()
    sync_entropy()
    enter_active_mode()
```

4.2 Active Drift Cycle

```
run_cycle():
    EP   = receive_entropy()
    SDMs = receive_SDMs()

    D1 = drift_from_entropy(EP)
    D2 = drift_from_diffs(SDMs)

    D  = D1 + D2

    if VaultGuard.invalid(D):
        D = VaultGuard.correct(D)

    apply_drift(D)

    dv = compute_local_diff()
    broadcast_SDM(dv)
```

4.3 Low-Power Mode

Triggered when: - battery low, - CPU overloaded, - network weak.

```
enter_low_power():
    reduce_cycle_frequency()
    limit_entropy_speed()
```

4.4 Exit Phase

```
shutdown():
    save_state()
    close_peers()
```

---

5. Critical Data Structures

5.1 Entropy Packet (EP)

```
type EP struct {
    entropy_vector []float64
    timestamp      int64
    node_load      float32
}
```

5.2 State Diff Message (SDM)

```
type SDM struct {
    shard_id     int
    diff_vector  []float64
    drift_weight float32
    cycle_id     int64
}
```

5.3 Shard State

```
type Shard struct {
    vu_map map[string]float64
    tc_map map[string]float64
    dc     DriftCoefficients
}
```

---

6. Drift Cycle Pseudocode (Full)

```
loop:
    EP = receive_EP()
    SDMs = receive_SDMs()
```

```
    // entropy drift
    D_entropy = alpha * compute_entropy_drift(EP)

    // peer-based drift
    D_diffs = beta * compute_diff_drift(SDMs)

    // combine
    D = D_entropy + D_diffs

    // apply VaultGuard rules
    if violates_invariants(D):
        D = correct_with_vaultguard(D)

    // apply drift
    local_state = local_state + D

    // generate diff
    dv = compute_local_diff()
    broadcast_SDM(dv)
```

---

7. Performance Profile

- CPU: <5% on typical smartphone
- Memory: 5–10 MB
- Bandwidth: <50 KB/min average
- Packet loss tolerance: high (>60%)
- Latency tolerance: high

Nodes do not require real-time precision.

---

8. Security Rules (Implementation Level)

8.1 Never trust incoming diffs blindly

All SDMs must be: - validated, - weighted, - rate-limited.

8.2 Clamp unsafe drift

```
if D pushes VU < 0:
    clamp to 0
```

8.3 Detect anomaly energy

Large sudden changes must be suppressed.

8.4 No identity storage

Node must not generate or store long-term identifiers.

---

9. Recommended Tech Stack

Browsers:

- TypeScript + WebRTC

Mobile:

- Kotlin/Swift + embedded drift engine

Servers:

- Go (best balance)
- Rust (high performance)

IoT:

- C++

---

10. SDK/Library Structure

Expected open-source repo structure:

```
/transport
/drift_core
/entropy_engine
/state_shards
/vaultguard
/runtime
/examples
```

---

11. Compliance Test

An LVS node is compliant if it: - implements drift cycle exactly, - supports EP/SDM formats, - enforces VaultGuard invariants, - manages state shards, - tolerates packet loss, - remains functional in low-power mode.

---

12. Conclusion

This blueprint defines how to build a fully compliant LVS node across all platforms. It provides a unified architectural and algorithmic foundation that ensures consistent behavior, security, and performance across the global LVS micro-node swarm.