Testing Distributed Systems: From Unit Tests to Jepsen

Zach Kelling

Lux Industries

2025

Abstract

Distributed systems bugs are the hardest bugs: they only manifest under specific timing, network conditions, and failure combinations that cannot be reproduced by running the system once. This paper presents the full testing pyramid for distributed blockchain systems: unit tests for individual components, integration tests for subsystem interactions, property-based tests for algorithmic correctness, model checking for protocol design, and formal proofs for mathematical certainty. We then cover fault injection (Jepsen-style testing): how to inject network partitions, clock skew, process crashes, and disk corruption, and how to verify linearizability and consistency under these faults. We present Lux Network's testing stack: Go unit tests (12,847 tests), Playwright E2E (342 tests), Foundry smart contract tests (832 tests), Halmos symbolic tests (9 proofs), TLA+ model checking (2 protocols), and Lean 4 formal proofs (384 theorems). Each layer catches different bugs. Skipping any layer leaves a blind spot where production failures hide.

Contents

1 The Testing Pyramid for Distributed Systems **4**
- 1.1 Why a Pyramid . . . 4
- 1.2 What Each Layer Misses . . . 4

2 Layer 1: Unit Tests **4**
- 2.1 What to Test . . . 4
- 2.2 Go Test Patterns . . . 4
- 2.3 Coverage Requirements . . . 5

3 Layer 2: Integration Tests **5**
- 3.1 Simulated Network . . . 6

4 Layer 3: Property-Based and Fuzz Tests **6**

5 Layer 4: End-to-End Tests **7**
- 5.1 Playwright for Web/API E2E . . . 7
- 5.2 Network E2E Tests . . . 7

6 Layer 5: Fault Injection (Jepsen-Style Testing) **8**
- 6.1 What Is Jepsen . . . 8
- 6.2 Consistency Models . . . 8
- 6.3 The Nemesis: Fault Injection . . . 8
- 6.4 Implementing Fault Injection . . . 8
- 6.5 Linearizability Checking . . . 9
- 6.6 Bugs Only Jepsen Finds . . . 10

7 Layer 6: Model Checking (TLA+) **10**

8 Layer 7: Formal Proofs (Lean 4) **11**

9 Our Testing Stack: By the Numbers **11**

10 How to Find Bugs That Only Appear Under Network Partition **11**
- 10.1 Step 1: Identify the Invariant . . . 11
- 10.2 Step 2: Design the Partition . . . 11
- 10.3 Step 3: Run Workload During Partition . . . 11
- 10.4 Step 4: Heal and Verify . . . 12
- 10.5 Step 5: Vary Timing . . . 12

11 Practical Advice **12**
11.1 Start at the Bottom 12
11.2 Deterministic Simulation Beats Live Testing 13
11.3 Invest in Observability 13
11.4 Test the Tests 13

12 Conclusion **13**

1 The Testing Pyramid for Distributed Systems

1.1 Why a Pyramid

Different testing techniques catch different bug classes at different costs:

Layer	Technique	Cost	Speed	Catches
1	Unit tests	Low	Fast	Logic bugs
2	Integration tests	Medium	Medium	Interface bugs
3	Property/fuzz tests	Medium	Medium	Edge cases, invariant violations
4	E2E tests	High	Slow	System-level regressions
5	Fault injection	High	Slow	Concurrency, network bugs
6	Model checking	High	Medium	Protocol design bugs
7	Formal proofs	Very high	Slow	Mathematical correctness

Table 1: The testing pyramid. Lower layers are cheaper and run more frequently.

The pyramid shape means: many unit tests (cheap, fast), fewer integration tests, even fewer E2E tests, and a handful of formal proofs (expensive, slow). Every layer is necessary—no layer substitutes for another.

1.2 What Each Layer Misses

- **Unit tests** miss: interactions between components, timing dependencies, network behavior.
- **Integration tests** miss: system-level emergent behavior, rare failure modes.
- **Property tests** miss: bugs that require specific state sequences (they test properties of individual operations, not sequences).
- **E2E tests** miss: rare timing conditions (they run the happy path and a few error paths).
- **Fault injection** misses: bugs in the protocol design (it tests the implementation, not the specification).
- **Model checking** misses: implementation bugs (it verifies the model, not the code).
- **Formal proofs** miss: everything that is not in the formal model.

2 Layer 1: Unit Tests

2.1 What to Test

Unit tests verify individual functions in isolation. For a blockchain node:

- Serialization/deserialization of messages, blocks, transactions.
- Cryptographic operations (signature, hash, verification).
- State transition logic (apply block to state).
- Data structure operations (merkle tree insert/proof/verify).
- Validator set management (add, remove, weight calculation).

2.2 Go Test Patterns

Listing 1: Table-driven unit test

```
func TestBlockValidation(t *testing.T) {
    tests := []struct {
        name    string
        block   Block
        wantErr error
```

```
    }{
        {
            name:    "valid block",
            block:   validBlock(),
            wantErr: nil,
        },
        {
            name:    "empty parent hash",
            block:   blockWithEmptyParent(),
            wantErr: ErrInvalidParentHash,
        },
        {
            name:    "timestamp in future",
            block:   blockWithFutureTimestamp(),
            wantErr: ErrFutureBlock,
        },
        {
            name:    "insufficient signatures",
            block:   blockWithTooFewSigs(),
            wantErr: ErrInsufficientQuorum,
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            err := ValidateBlock(tt.block)
            if !errors.Is(err, tt.wantErr) {
                t.Errorf("ValidateBlock() = %v, want %v",
                    err, tt.wantErr)
            }
        })
    }
}
```

2.3 Coverage Requirements

We require 80% line coverage for the node codebase. More importantly, we require 100% coverage of error paths: every `if err != nil` branch must be exercised by a test that triggers that error.

Listing 2: Running Go tests with coverage

```
go test -v -race -coverprofile=coverage.out ./...
go tool cover -func=coverage.out | grep total
# total:  (statements)  83.2%
```

3 Layer 2: Integration Tests

Integration tests verify that components work together. The key difference from unit tests: multiple components are instantiated (but not the full system).

Listing 3: Integration test for consensus + networking

```
func TestConsensusWithNetworkDelay(t *testing.T) {
    // Create 4 validators with simulated network
    net := newTestNetwork(4)

    // Add latency: 50ms per message
```

```
    net.SetLatency(50 * time.Millisecond)

    // Propose a block
    net.Validators[0].Propose(testBlock)

    // Wait for consensus
    for _, v := range net.Validators {
        select {
        case decision := <-v.Decisions():
            require.Equal(t, testBlock.Hash(), decision.Hash())
        case <-time.After(5 * time.Second):
            t.Fatal("consensus did not complete within 5s")
        }
    }
}
```

3.1 Simulated Network

Our integration test framework includes a simulated network that supports:

- Configurable latency per link.
- Message reordering.
- Message loss (configurable drop rate).
- Partitions (disconnect a subset of nodes from the rest).

This is not full Jepsen-style testing (that uses real processes)—it is deterministic simulation that runs in milliseconds.

4 Layer 3: Property-Based and Fuzz Tests

Covered in detail in our companion paper on fuzzing. Key applications for distributed systems:

- **Serialization round-trip**: For every message type, serialize then deserialize and check equality.
- **State machine determinism**: Apply the same sequence of blocks to two independent state instances and verify identical final states.
- **Merkle proof soundness**: For every leaf in a tree, generate and verify the proof.

Listing 4: Fuzz test for message serialization

```
func FuzzMessageRoundTrip(f *testing.F) {
    f.Add([]byte{1, 0, 0, 0}) // PROPOSE type

    f.Fuzz(func(t *testing.T, data []byte) {
        msg, err := DecodeMessage(data)
        if err != nil {
            return // Invalid input is fine
        }

        // Re-encode
        encoded := msg.Encode()

        // Re-decode
        msg2, err := DecodeMessage(encoded)
        require.NoError(t, err)

        // Must be equal
        require.Equal(t, msg, msg2)
    })
```

```
}
```

5 Layer 4: End-to-End Tests

E2E tests run the full system (multiple node processes, real networking, real storage) and verify user-visible behavior.

5.1 Playwright for Web/API E2E

Listing 5: Playwright E2E test for transaction submission

```
import { test, expect } from '@playwright/test';

test('submit transaction and verify inclusion', async ({ request }) =>
   {
  // Submit a transfer transaction
  const txHash = await submitTransaction(request, {
    from: testAccount1,
    to: testAccount2,
    value: '1000000000000000000', // 1 LUX
  });

  // Wait for block inclusion
  const receipt = await waitForReceipt(request, txHash, {
    timeout: 30000,
  });
  expect(receipt.status).toBe(1);
  expect(receipt.blockNumber).toBeGreaterThan(0);

  // Verify balance change
  const balance = await getBalance(request, testAccount2);
  expect(BigInt(balance)).toBeGreaterThanOrEqual(
    BigInt('1000000000000000000')
  );
});
```

5.2 Network E2E Tests

Our network E2E tests use the Lux CLI to spin up local networks and verify:

- Chain bootstrapping and block production.
- Cross-chain messaging (Teleport).
- Validator addition/removal during operation.
- Node restart and state recovery.

Listing 6: E2E test via Lux CLI

```
# Start a local 5-node network
lux network start --nodes 5

# Wait for chain to produce blocks
lux network health --wait 30s

# Submit transactions
lux transaction send --amount 1 --to 0x1234...

```

```
# Verify finality
lux chain status --chain C

# Stop network
lux network stop
```

6 Layer 5: Fault Injection (Jepsen-Style Testing)

6.1 What Is Jepsen

Jepsen is a framework for testing distributed systems under adverse conditions. It:

1. Deploys the system on multiple VMs or containers.
2. Runs a workload (clients submitting operations).
3. Injects faults (partitions, crashes, clock skew).
4. Records a history of operations and results.
5. Checks the history against a consistency model (linearizability, serializability, sequential consistency).

6.2 Consistency Models

Definition 1 (Linearizability)**.** *A history H is linearizable if every operation appears to take effect at a single point between its invocation and response, and the resulting sequential history satisfies the specification.*

Linearizability is the gold standard. If a blockchain's RPC endpoint is linearizable, then for every pair of transactions, if T_1 was confirmed before T_2 was submitted, then T_1 appears before T_2 in the chain.

6.3 The Nemesis: Fault Injection

A Jepsen "nemesis" is a process that injects faults:

Fault	What It Tests
Network partition	Can the system make progress with a minority partition? Does it maintain safety?
Process kill (SIGKILL)	Does the system recover correctly after an unclean shutdown?
Process pause (SIGSTOP)	Does the system handle slow/unresponsive nodes?
Clock skew	Does time-dependent logic (timeouts, timestamps) break under clock drift?
Disk corruption	Does the system detect and recover from corrupted state?
Packet loss	Does the system handle unreliable networks?
Packet reordering	Does the system handle out-of-order message delivery?

Table 2: Nemesis fault types for blockchain testing.

6.4 Implementing Fault Injection

Listing 7: Fault injection test framework

```
type Nemesis interface {
    // Inject starts the fault
```

```
    Inject(cluster *Cluster) error
    // Recover restores normal operation
    Recover(cluster *Cluster) error
}

// Partition nemesis: splits the cluster into two groups
type PartitionNemesis struct {
    GroupA []int // Node indices in partition A
    GroupB []int // Node indices in partition B
}

func (p *PartitionNemesis) Inject(cluster *Cluster) error {
    for _, a := range p.GroupA {
        for _, b := range p.GroupB {
            if err := cluster.BlockTraffic(a, b); err != nil {
                return err
            }
        }
    }
    return nil
}

func (p *PartitionNemesis) Recover(cluster *Cluster) error {
    return cluster.UnblockAllTraffic()
}

// CrashNemesis: kills a random node
type CrashNemesis struct {
    NodeIndex int
}

func (c *CrashNemesis) Inject(cluster *Cluster) error {
    return cluster.Kill(c.NodeIndex) // SIGKILL, not SIGTERM
}

func (c *CrashNemesis) Recover(cluster *Cluster) error {
    return cluster.Restart(c.NodeIndex)
}
```

6.5 Linearizability Checking

After running a workload with fault injection, we collect the history of all operations (invocations and responses) and check linearizability using the Knossos algorithm:

Listing 8: Linearizability check

```
func TestLinearizabilityUnderPartition(t *testing.T) {
    cluster := StartCluster(5) // 5 nodes
    defer cluster.Stop()

    // Start workload: concurrent read/write operations
    history := NewHistory()
    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func(clientID int) {
            defer wg.Done()
            for j := 0; j < 100; j++ {
```

```
                op := randomOperation()
                invocation := history.RecordInvoke(clientID, op)
                result, err := cluster.Execute(op)
                history.RecordReturn(invocation, result, err)
            }
        }(i)
    }

    // Inject faults concurrently
    go func() {
        nemesis := &PartitionNemesis{
            GroupA: []int{0, 1},
            GroupB: []int{2, 3, 4},
        }
        time.Sleep(2 * time.Second)
        nemesis.Inject(cluster)
        time.Sleep(5 * time.Second)
        nemesis.Recover(cluster)
    }()

    wg.Wait()

    // Check linearizability
    ok, info := CheckLinearizable(history, RegisterModel{})
    if !ok {
        t.Fatalf("linearizability violation:\n%s", info)
    }
}
```

6.6 Bugs Only Jepsen Finds

Bug	Description	Fault Required
Stale read after partition heal	After a network partition heals, a node serves stale state for up to 2 seconds before catching up.	Network partition
Double-commit on leader change	If the leader crashes mid-commit, the new leader may re-propose the same block, causing duplicate execution.	Process crash
Timeout miscalculation	Clock skew causes a node to time out and vote for a conflicting proposal even though the original proposer is alive.	Clock skew
State corruption on crash recovery	An unclean shutdown during state database write leaves a partially written batch, causing the node to start with corrupt state.	SIGKILL during write

Table 3: Bugs found exclusively through fault injection testing.

7 Layer 6: Model Checking (TLA+)

Covered in our companion paper on TLA+. In the testing pyramid, model checking sits above fault injection because it reasons about the protocol *design*, not the implementation. It catches bugs that exist in every correct implementation of a flawed protocol.

8 Layer 7: Formal Proofs (Lean 4)

Covered in our companion paper on Lean 4. Formal proofs provide the strongest guarantees but are the most expensive to produce. We use them for the core consensus safety and liveness theorems—properties where failure means chain death.

9 Our Testing Stack: By the Numbers

Layer	Count	CI Time	Frequency	Tool
Go unit tests	12,847	4m	Every commit	`go test`
Go fuzz corpus	2,341 entries	2m (replay)	Every commit	`go test -fuzz`
Foundry contract tests	832	3m	Every commit	`forge test`
Halmos symbolic	9 proofs	8m	Every PR	`halmos`
Playwright E2E	342	12m	Every PR	`playwright test`
Fault injection	47 scenarios	45m	Nightly	Custom framework
TLA+ model check	2 protocols	38m	On protocol change	TLC
Lean 4 proofs	384 theorems	6m	On model change	`lake build`

Table 4: Lux Network's complete testing stack.

Total CI time for a full run: approximately 2 hours. Unit tests and contract tests run on every commit (7 minutes). Symbolic tests and E2E tests run on every PR (20 minutes). Fault injection runs nightly (45 minutes). Model checking and formal proofs run only when the relevant code changes.

10 How to Find Bugs That Only Appear Under Network Partition

This is the most practically useful section. Here is the methodology:

10.1 Step 1: Identify the Invariant

Before injecting faults, define what "correct" means. For consensus:

- **Safety**: No two honest nodes finalize conflicting blocks at the same height.
- **Liveness**: Every submitted transaction is eventually finalized.
- **Consistency**: After partition heals, all nodes converge to the same state.

10.2 Step 2: Design the Partition

Common partition topologies:

- **Minority isolation**: 1 node isolated from 4 (minority should not finalize).
- **Majority split**: 3 vs. 2 (majority should continue, minority should stall).
- **Ring partition**: Each node can talk to its neighbors but not distant nodes (tests message propagation assumptions).
- **Asymmetric**: A can send to B but not receive (tests connection directionality assumptions).

10.3 Step 3: Run Workload During Partition

Submit transactions to nodes on both sides of the partition. Record all operations with precise timestamps.

10.4 Step 4: Heal and Verify

Remove the partition. Wait for convergence (how long? this is itself a property to test). Verify:

- All transactions submitted to the majority partition are finalized.
- No conflicting blocks exist at any height.
- The minority partition caught up to the majority's state.
- No transactions were lost or duplicated.

10.5 Step 5: Vary Timing

The same partition with different timing (partition during proposal, during voting, during commit) may expose different bugs. Run the scenario many times with randomized timing.

Listing 9: Randomized partition timing

```
func TestPartitionAtRandomPhase(t *testing.T) {
    for trial := 0; trial < 100; trial++ {
        cluster := StartCluster(5)

        // Random delay before partition
        delay := time.Duration(rand.Intn(3000)) * time.Millisecond
        time.Sleep(delay)

        // Partition
        nemesis := RandomPartition(5)
        nemesis.Inject(cluster)

        // Hold partition for random duration
        hold := time.Duration(1000+rand.Intn(5000)) * time.Millisecond
        time.Sleep(hold)

        // Heal
        nemesis.Recover(cluster)

        // Verify
        require.Eventually(t, func() bool {
            return cluster.AllConverged()
        }, 30*time.Second, 100*time.Millisecond,
            "trial %d: nodes did not converge after partition heal",
               trial)

        require.True(t, cluster.NoForks(),
            "trial %d: fork detected after partition", trial)

        cluster.Stop()
    }
}
```

11 Practical Advice

11.1 Start at the Bottom

If you do not have unit tests, do not attempt Jepsen testing. Fix the foundation first. Most distributed systems bugs are found by good unit tests of the state machine logic, not by elaborate fault injection.

11.2 Deterministic Simulation Beats Live Testing

When possible, use a deterministic simulation of the network (FoundationDB's approach) rather than real network partitions. Deterministic simulation is faster, reproducible, and catches the same classes of bugs. Reserve live fault injection for final validation.

11.3 Invest in Observability

When a fault injection test fails, you need to understand what happened. Structured logging with causal tracing (each message carries a trace ID through the system) is essential. Without it, debugging a partition-induced bug across 5 nodes is nearly impossible.

11.4 Test the Tests

Introduce known bugs (mutation testing) and verify that your test suite catches them. If a known safety violation does not trigger a test failure, your tests have a blind spot.

12 Conclusion

Testing distributed systems is not a single activity—it is a discipline that spans seven layers, each catching different bug classes at different costs. The layers are complementary: unit tests catch logic bugs quickly, fault injection catches concurrency bugs under failure, model checking catches protocol design bugs, and formal proofs provide mathematical certainty for the core safety properties.

Lux Network's testing stack (12,847 unit tests, 832 contract tests, 9 symbolic proofs, 342 E2E tests, 47 fault injection scenarios, 2 model-checked protocols, 384 formal theorems) represents a deliberate investment in each layer. The bugs each layer has caught justify the investment: unit tests find daily regressions, fault injection found 4 bugs that only appear under network partition, model checking found 3 protocol design bugs, and formal proofs provide confidence that the core consensus is correct by construction.

The hardest part is not building the tests—it is maintaining them. Every protocol change requires updating tests at multiple layers. This is the cost of confidence.

References

[1] Kingsbury, K. "Jepsen: Distributed Systems Safety Research." jepsen.io, 2013–2024.

[2] Zhou, J. et al. "FoundationDB: A Distributed Unbundled Transactional Key-Value Store." SIGMOD, 2021.

[3] Alvaro, P. et al. "Lineage-Driven Fault Injection." SIGMOD, 2015.

[4] Herlihy, M. and Wing, J. "Linearizability: A Correctness Condition for Concurrent Objects." ACM TOPLAS, 1990.

[5] Kleppmann, M. "Designing Data-Intensive Applications." O'Reilly, 2017.

[6] Newcombe, C. et al. "How Amazon Web Services Uses Formal Methods." Communications of the ACM, 2015.