Consensus Tester Project Specification

Enhanced Version 2.0

Document Date: September 2025

Table of Contents

1. Project Scope & Vision

> **Mission:** Build a comprehensive SDK and CLI tool to stress-test, debug, and validate consensus algorithms under realistic fault conditions with deterministic reproducibility.

Core Objectives

- **Algorithm Agnostic:** Support Raft, Paxos, Multi-Paxos, EPaxos, and custom implementations
- **Comprehensive Testing:** Workload simulation, fault injection, correctness validation
- **Developer-Friendly:** Clean SDK interfaces with minimal integration overhead
- **Deterministic Debugging:** Reproducible test scenarios with seed-based randomization
- **Production-Ready Insights:** Realistic failure modes and performance characteristics

Target Audience

- Distributed systems engineers implementing consensus algorithms
- Research teams validating theoretical improvements
- Production teams stress-testing existing implementations
- Educational institutions teaching distributed systems

Success Metrics

- Integration time < 2 hours for existing implementations
- Support for 95% of common consensus algorithm patterns
- Deterministic reproduction of 100% of discovered bugs
- Performance overhead < 10% in benchmarking mode

2. Architecture Overview

> The architecture follows a layered approach with clear separation of concerns, enabling both simple integration and advanced customization.

Layer 1: SDK Interface

Provides clean, minimal interfaces for algorithm integration:

- **Node Interface:** Core consensus node abstraction
- **Message System:** Type-safe message passing with serialization
- **State Management:** Snapshotting, comparison, and validation
- **Configuration API:** Runtime parameter adjustment

Layer 2: Simulation Engine

Orchestrates test execution with precise control:

- **Event Loop:** Deterministic scheduling with configurable time models
- **Cluster Manager:** Node lifecycle and coordination
- **Transport Layer:** Network simulation with realistic fault injection
- **Workload Generator:** Configurable load patterns and client behavior
- **Fault Controller:** Systematic and chaos-based failure injection

Layer 3: Analysis & Reporting

Comprehensive validation and insight generation:

- **Property Checker:** Safety and liveness validation with custom rules
- **Metrics Engine:** Performance and behavior analysis
- **Report Generator:** Multi-format output (terminal, JSON, HTML, PDF)
- **Visualization:** Timeline and state evolution graphics

3. Project Structure

```
consensus-tester/
├── cmd/
│   ├── consensus-tester/          # Main CLI application
│   └── scenario-builder/          # Interactive scenario generator
├── pkg/
│   ├── sdk/                       # Developer-facing SDK
│   │   ├── node.go               # Core Node interface
│   │   ├── message.go            # Message types and routing
│   │   ├── state.go              # State management
│   │   ├── config.go             # Configuration API
│   │   └── adapters/             # Common algorithm adapters
│   ├── engine/                    # Simulation engine
│   │   ├── cluster.go            # Cluster management
│   │   ├── transport.go          # Network simulation
│   │   ├── scheduler.go          # Event scheduling
│   │   ├── workload.go           # Load generation
│   │   └── failures.go           # Fault injection
│   ├── checker/                   # Correctness validation
│   │   ├── safety.go             # Safety properties
│   │   ├── liveness.go           # Liveness properties
│   │   ├── custom.go             # Custom property DSL
│   │   └── linearizability.go    # Linearizability checker
│   ├── metrics/                   # Performance analysis
│   │   ├── collector.go          # Data collection
│   │   ├── analyzer.go           # Statistical analysis
│   │   └── exporters.go          # Format exporters
│   ├── report/                    # Report generation
│   │   ├── terminal.go           # Console output
│   │   ├── json.go               # JSON export
│   │   ├── html.go               # HTML reports
│   │   └── pdf.go                # PDF generation
│   └── scenarios/                 # Scenario management
│       ├── builder.go            # Programmatic scenarios
│       ├── loader.go             # YAML/JSON loading
│       └── templates/            # Predefined scenarios
├── examples/                      # Reference implementations
│   ├── raft/                     # Raft example
│   ├── paxos/                    # Paxos example
│   └── custom/                   # Custom algorithm template
├── scenarios/                     # Built-in test scenarios
│   ├── basic/                    # Simple test cases
│   ├── stress/                   # High-load scenarios
│   ├── chaos/                    # Fault-heavy scenarios
│   └── benchmarks/               # Performance baselines
├── docs/                         # Documentation
│   ├── quickstart.md             # Getting started guide
│   ├── sdk-guide.md              # SDK integration guide
│   ├── scenarios.md              # Scenario configuration
│   └── api/                      # API documentation
├── test/                         # Framework tests
│   ├── integration/              # End-to-end tests
│   ├── unit/                     # Unit tests
│   └── fixtures/                 # Test data
└── tools/                        # Development utilities
    ├── generate-docs.go          # Documentation generator
    └── validate-scenarios.go     # Scenario validator
```

4. Core Interfaces & APIs

Node Interface

```
type Node interface {
    // Core lifecycle
    ID() NodeID
    Start(ctx context.Context) error
    Stop(ctx context.Context) error

    // Message handling
    HandleMessage(msg Message) error
    SendMessage(to NodeID, msg Message) error

    // State management
    GetState() NodeState
    SetState(state NodeState) error
    CreateSnapshot() ([]byte, error)
    RestoreSnapshot(data []byte) error

    // Configuration
    UpdateConfig(config map[string]interface{}) error

    // Debugging hooks
    SetDebugger(debugger Debugger)
}
```

State Management

```
type NodeState interface {
    // State identification
    StateID() string
    StateType() string

    // Serialization
    Marshal() ([]byte, error)
    Unmarshal(data []byte) error

    // Comparison for validation
    Equals(other NodeState) bool
    Hash() uint64

    // Custom validation rules
    Validate(rules ValidationRules) []Violation
}
```

Message System

```
type Message interface {
    Type() MessageType
    Source() NodeID
    Destination() NodeID
    Payload() []byte
    Timestamp() time.Time

    // Message properties for fault injection
    SetDelay(duration time.Duration)
    SetDuplicate(count int)
    SetCorrupt(probability float64)
}
```

5. Testing Strategy

> **Critical Requirement:** The consensus tester itself must be thoroughly tested to ensure correctness of its validation logic.

Framework Self-Testing

- **Reference Implementations:** Known-correct algorithms (Raft, Paxos) as test cases
- **Intentional Bugs:** Broken implementations that should trigger specific violations
- **Property Verification:** Mathematical proofs for safety/liveness checkers
- **Fault Injection Validation:** Verify faults are actually injected as configured

Deterministic Testing

```
type TestConfig struct {
    Seed           int64           // For reproducible randomization
    TimeModel      TimeModel       // Synchronous, asynchronous, etc.
    EventOrdering  OrderingMode    // Strict, relaxed, random
    FaultSchedule  []FaultEvent    // Predetermined fault sequence
    Checkpoints    []Checkpoint    // State validation points
}
```

Test Categories

Category	Purpose	Duration	Determinism
Unit Tests	Component validation	< 1 second	Always deterministic
Integration Tests	End-to-end workflows	< 30 seconds	Seeded randomization
Stress Tests	Performance validation	1-10 minutes	Configurable
Chaos Tests	Fault discovery	Hours	Logged for reproduction

6. Network Models & Fault Injection

Supported Network Models

- **Synchronous:** Bounded message delays and processing times
- **Asynchronous:** Unbounded delays, no timing assumptions
- **Partially Synchronous:** Alternating periods of synchrony and asynchrony
- **Real-time:** Actual wall-clock timing for performance testing

Fault Types

```
type FaultType int

const (
    // Network faults
    MessageLoss       FaultType = iota
    MessageDelay
    MessageDuplication
    MessageCorruption
    NetworkPartition

    // Node faults
    NodeCrash
    NodeSlowdown
    NodeByzantine

    // System faults
    ClockSkew
    ResourceExhaustion
    DiskFailure
)
```

Fault Configuration

```
type FaultConfig struct {
    Type        FaultType
    Target      []NodeID        // Affected nodes
    Probability float64         // Fault probability
    Duration    time.Duration   // Fault duration
    Schedule    []time.Time     // Exact timing
    Parameters  map[string]interface{} // Fault-specific params
}
```

7. Metrics & Validation

Core Metrics

Category	Metrics	Purpose
Performance	Throughput, Latency, CPU/Memory usage	Scalability analysis
Consensus	Leader elections, Log entries, Agreement time	Algorithm efficiency
Fault Tolerance	Recovery time, Availability, Data loss	Resilience validation
Correctness	Safety violations, Liveness failures, Invariant breaks	Bug detection

Safety Properties

- **Agreement:** No two nodes commit different values
- **Validity:** Only proposed values can be chosen
- **Integrity:** Messages are not corrupted or duplicated
- **Non-triviality:** Progress is possible under good conditions

Liveness Properties

- **Termination:** All correct processes eventually decide
- **Progress:** The system makes forward progress
- **Responsiveness:** Timely responses under good conditions
- **Availability:** Service remains available during failures

Custom Property DSL

```
// Example custom property definition
property LeaderUniqueness {
    invariant: forall t in timeline {
        count(nodes.filter(n => n.role == LEADER)) <= 1
    }

    violation_handler: func(violation) {
        report("Multiple leaders detected", violation.timestamp)
        collect_states(violation.nodes)
    }
}
```

8. Configuration & Scenarios

Scenario Configuration

```
scenario_name: "partition-recovery-test"
description: "Test recovery after network partition"

cluster:
  nodes: 5
  algorithm: "raft"

network:
  model: "asynchronous"
  base_latency: "10ms"
  jitter: "5ms"

workload:
  type: "key-value"
  operations_per_second: 100
  duration: "5m"

faults:
  - type: "network_partition"
    start: "1m"
    duration: "30s"
    nodes: [1, 2]

validation:
  properties: ["safety", "liveness"]
  custom_checks: ["leader_uniqueness"]

reporting:
  formats: ["json", "html"]
  metrics: ["throughput", "elections"]
```

Scenario Templates

- **Basic Functionality:** Happy path testing
- **Leader Election:** Leadership change scenarios
- **Network Partitions:** Split-brain testing
- **Node Failures:** Crash and recovery patterns
- **Performance Stress:** High-load testing
- **Chaos Testing:** Random fault injection

9. Technical Stack

Core Technologies

Component	Technology	Rationale
Runtime	Go 1.21+	Excellent concurrency, fast compilation, simple deployment
CLI Framework	Cobra + Viper	Rich command-line interface with configuration management
Serialization	Protocol Buffers	Efficient, versioned message formats
Logging	Zerolog	High-performance structured logging
Testing	Testify + Ginkgo	Rich assertion library + BDD testing
Visualization	Chart.js + D3.js	Interactive charts and timeline visualization

Dependencies

```
// Core dependencies
github.com/spf13/cobra           // CLI framework
github.com/spf13/viper           // Configuration
github.com/rs/zerolog            // Logging
google.golang.org/protobuf       // Serialization
github.com/stretchr/testify      // Testing
github.com/onsi/ginkgo/v2        // BDD testing

// Optional dependencies (based on features)
github.com/prometheus/client_golang  // Metrics export
go.uber.org/zap                     // Alternative logging
github.com/hashicorp/raft           // Reference Raft impl
```

10. Implementation Phases

> **Estimated Timeline:** 8-10 weeks with 1-2 developers

Phase 1: Foundation (2-3 weeks)

- Core SDK interfaces and basic implementations
- Simple event loop and cluster management
- Basic message transport without fault injection
- Reference Raft implementation integration
- Unit tests for all core components

Phase 2: Fault Injection (2 weeks)

- Network fault injection (delays, drops, partitions)
- Node failure simulation (crashes, slowdowns)
- Deterministic scheduling with seed support
- Scenario configuration system
- Integration tests with fault scenarios

Phase 3: Validation & Metrics (2 weeks)

- Safety and liveness property checkers
- Performance metrics collection
- Custom property DSL implementation
- Linearizability checker integration
- Comprehensive test suite validation

Phase 4: Reporting & Polish (2-3 weeks)

- Multi-format reporting (terminal, JSON, HTML)
- Interactive visualization components
- CLI polish and user experience improvements
- Documentation and examples
- Performance optimization and profiling

11. Future Extensions

Advanced Features (6-month roadmap)

- **Distributed Mode:** Test real distributed nodes via gRPC
- **Formal Verification:** Integration with TLA+ or similar tools
- **Machine Learning:** Automated bug pattern detection
- **Cloud Integration:** AWS/GCP/Azure deployment scenarios
- **Language Bindings:** Python, Java, Rust SDK wrappers

Scalability Improvements

- Support for 1000+ node simulations
- Hierarchical clustering for mega-scale testing
- GPU acceleration for complex state checking
- Distributed testing across multiple machines

Algorithm Support Expansion

- Byzantine Fault Tolerant algorithms (PBFT, HotStuff)
- Blockchain consensus (Ethereum, Bitcoin)
- Specialized protocols (Multi-Paxos, Fast Paxos, EPaxos)
- Custom consensus algorithm wizard

12. Appendices

Appendix A: Integration Checklist

> **To integrate your consensus algorithm:**
>
> 1. Implement the Node interface (30 minutes)
> 2. Define your message types (15 minutes)
> 3. Implement state serialization (45 minutes)
> 4. Write a basic scenario config (15 minutes)
> 5. Run your first test (5 minutes)
>
> **Total estimated integration time: 1 hour 50 minutes**

Appendix B: Performance Characteristics

Scenario Type	Node Count	Memory Usage	CPU Overhead	Test Duration
Basic Testing	3-5	< 50MB	< 5%	1-60 seconds
Stress Testing	5-20	< 200MB	< 15%	1-10 minutes
Chaos Testing	10-50	< 1GB	< 25%	10 minutes - 24 hours

Appendix C: Limitations and Trade-offs

> **Fundamental Limitations:**
>
> - **Simulation vs Reality:** Some behaviors only emerge in production environments
> - **Performance Impact:** The testing framework itself affects the system being tested
> - **Determinism Boundaries:** Byzantine and resource faults have inherent non-determinism
> - **Integration Complexity:** Real-world implementations may require significant adaptation
> - **Coverage Gaps:** Cannot test all possible failure combinations or edge cases

Appendix D: When NOT to Use This Framework

- **Production Performance Testing:** Use actual cluster deployments instead
- **Hardware-Specific Failures:** Real disk failures, network card issues, etc.
- **Security Validation:** Framework doesn't test cryptographic or authentication layers
- **Large-Scale Performance:** >100 nodes may be better tested with actual distributed systems
- **Real-Time Systems:** Hard real-time requirements cannot be simulated accurately

Appendix E: Common Pitfalls (Updated)

> **Critical issues to avoid:**
>
> - **State Leakage:** Ensure complete state isolation between test runs
> - **Time Dependencies:** Avoid relying on wall-clock time in deterministic mode
> - **Resource Cleanup:** Properly cleanup goroutines and file handles
> - **Message Ordering:** Don't assume message delivery order
> - **Error Handling:** Properly propagate and handle all error conditions
> - **False Confidence:** Remember that passing tests don't guarantee production correctness
> - **Performance Misattribution:** Distinguish between algorithm and framework performance
> - **Integration Assumptions:** Don't assume all implementations will integrate easily

> **Contact Information:**
>
> For questions, bug reports, or feature requests, please visit our GitHub repository or contact the development team.
>
>