



## GSoC 2024 Proposal

<b>Personal Details and Contact Information.....</b>	<b>1</b>
<b>Synopsis.....</b>	<b>1</b>
<b>Current Status of the Project.....</b>	<b>1</b>
<b>Goals.....</b>	<b>3</b>
<b>Approach.....</b>	<b>4</b>
<b>Timeline.....</b>	<b>45</b>
<b>About Me.....</b>	<b>47</b>
Commitments.....	47
Planning for the Development Period:.....	47
<b>Appendix - Optional Deliverables Approach.....</b>	<b>48</b>

Harsh Raj  
harsh.raj.cd.ece21@iitbhu.ac.in  
Bihar, India

Timezone: Indian Standard Time (+5.30 UTC)

# Implementing Chaos Testing in Madsim

MadJep: Deterministic Chaos Engineering with Madsim

## Personal Details and Contact Information

Hi! I am Harsh Raj, a pre-final year student from the *Indian Institute of Technology (BHU) Varanasi, India*. I am an avid open-source enthusiast who loves making (and breaking) software.

You might better recognize me as [Harsh1s](#), as I have contributed a bit to Xline in the past few months. Here are my details and contact information:

- **Email Address:** [harsh.raj.cd.ece21@itbhu.ac.in](mailto:harsh.raj.cd.ece21@itbhu.ac.in)
- **Major:** Electronics and Communication Engineering
- **Degree:** Bachelor of Technology (B. Tech.)
- **Discord:** harsh\_raj.exe
- **Time-zone:** Indian Standard Time (+5:30 UTC)
- **LinkedIn:** <https://www.linkedin.com/in/harsh1s>
- **Resume:** <https://aquamarine-elenore-41.tiiny.site>
- **Postal Address:** Postal Park Road no. 2, Kankarbagh, Patna, Bihar (800001), India
- **Phone:** (+91) 85446 33455

## Synopsis

Distributed systems are notoriously difficult to test and debug due to their inherent complexity and non-determinism. Chaos engineering tools like Jepsen provide a powerful way to uncover potential failures by injecting faults into a real-world environment. However, the lack of determinism in these tests can make it challenging to reproduce and isolate bugs. On the other hand, deterministic simulation with Madsim offers complete control over the execution environment, but it often lacks the rich fault models needed for comprehensive chaos testing. This project aims to bridge this gap by creating MadJep, a deterministic chaos testing framework inspired by Jepsen's principles but built on top of Madsim for full reproducibility.

## Current Status of the Project

### Analysis of Xline Project

- **Functional Core:** Xline has a robust implementation of the CURP consensus protocol, enabling it to achieve strong consistency guarantees (likely linearizability) with reduced latency in high-latency WAN environments compared to traditional consensus approaches.
- **etcd Compatibility:** Xline provides an etcd-compatible API layer. This strategic decision facilitates adoption by easing integration into existing ecosystems that rely on etcd.
- **Storage Layer:** The recent introduction of a persistent storage layer (RocksDB) ensures data durability across node restarts and potential crashes.

- **Testing Practices:** Xline employs a combination of unit testing and integration testing to maintain code quality and reliability. However, the current testing approach has limitations in simulating the complex and chaotic conditions encountered in real-world distributed deployments.

## Analysis of Xline Jepsen Tests

- **Key Challenges :** The Jepsen tests exposed two fundamental challenges within Xline's design that undermine its ability to offer strict consistency guarantees:
  - **Asynchronous Storage Persistence:** The initial emphasis on performance led to asynchronous storage, which introduced a host of complexities in guaranteeing execution order and recovery behavior. It highlights the trade-off between raw performance and guarantees in distributed systems.
  - **Revision Generation:** The attempt to maintain etcd compatibility *and* achieve 1-RTT consensus proved fundamentally impossible. This incompatibility arises from the core differences between Raft's state machine approach and CURP's emphasis on commutative properties.
- **Jepsen's Value:** The Jepsen tests were instrumental in surfacing these deep-rooted issues. Without proactive chaos testing, these subtleties might have been overlooked, potentially leading to data inconsistencies in production later on.

## Lessons Learned

- **Correctness over Performance (Initially):** Xline's experience emphasizes the critical importance of prioritizing correctness in the initial design of distributed systems. Performance optimizations should be introduced cautiously, always considering their impact on the system's consistency guarantees.
- **Understanding the Consensus Protocol:** The revision issue demonstrates the need to thoroughly understand the fundamental properties and limitations of CURP. Assumptions based on other consensus protocols can be misleading.
- **Rethinking Compatibility:** The difficulties of strict etcd compatibility with CURP suggest a few possible paths:
  - **Relaxed Compatibility:** Explore a weaker form of compatibility, clearly communicating the differences to users.
  - **Divergent API:** Consider a distinct Xline-native API, potentially offering a different set of features and guarantees.

## Implications for MadJep Integration

The challenges uncovered in the Jepsen testing point to specific areas where MadJep would bring significant value in validating Xline's behavior:

- **Determinism for Asynchronous Issues:** MadJep's deterministic fault injection and simulation capabilities would be invaluable in systematically reproducing the complex asynchronous scenarios that led to bugs in Xline. This would speed up debugging and the exploration of mitigation strategies (e.g., moving towards synchronous persistence).
- **Network Simulation:** MadJep's fine-grained network fault models can precisely replicate the high-latency WAN environments where Xline aims to excel. This allows for testing under realistic conditions and verifying performance expectations.

- **Testing Evolving Consistency Models:** Depending on the decision about etcd compatibility, MadJep's flexibility can be used to define consistency checkers based on the chosen consistency model (linearizability or a relaxed serializability variant).

# Goals

## 1. Rigorous Verification of Consistency, Correctness, and Robustness

- **Deterministic Chaos:** Systematically inject a wide range of network faults, node failures, and data corruption scenarios using MadJep's fault injection models within Madsim's simulation environment. Verify that the target distributed system (Xline or others) maintains its consistency guarantees (likely linearizability or a well-defined relaxed model) under these chaotic conditions.
- **Reproducibility:** Leverage MadJep's and Madsim's deterministic approach to reliably reproduce failure scenarios. Streamline issue identification, debugging, and the verification of fixes.
- **Evolving Consistency Models:** Employ MadJep's flexible consistency checkers (Knossos-like, Elle-like) to tailor tests to the specific consistency model adopted by the system under test, given potential trade-offs revealed by previous testing (e.g., with Jepsen).
- **Realistic WAN Simulation:** Employ MadJep's network models within Madsim to create precise simulations of high-latency WAN environments with controlled packet delays, losses, and partitions.
- **WAN-specific Workloads:** Develop workload generators that mimic real-world usage patterns in WAN settings, focusing on the types of conflicts and concurrency the target system is likely to encounter.

## 2. Proactive Resilience Engineering

- **"What if?" Exploration:** Use MadJep within Madsim to proactively investigate potential failure scenarios and edge cases that might be difficult or costly to discover in traditional testing or production.
- **Confidence in Upgrades:** Rigorously test new releases or configuration changes within the MadJep/Madsim framework to identify potential regressions or unexpected behaviors before deployment.
- **Stress Testing New Use Cases:** As adoption of the system under test grows, leverage Madsim and MadJep to simulate diverse workload patterns and operational scenarios, helping expose issues related to scalability or unforeseen interactions.

## 3. Expand Madsim's Chaos Engineering Capabilities

- **Rich Fault Injection Toolkit:** Develop a comprehensive set of operations/faults generators within the Madsim framework for these failure modes:
  - **Network Faults:** Delays, partitions, packet loss, reordering.
  - **Node Faults:** Controlled crashes, restarts, and resource exhaustion.
  - **Storage Faults:** Data corruption and simulated disk failures.
- **Flexibility:** Design the generators to allow fine-grained configuration of fault parameters (e.g., delay durations, partition sizes, affected nodes), enabling the creation of highly specific and targeted chaotic scenarios.

## 4. Build Robust Consistency Verification Tools

- **Linearizability Checking (Knossos-like):** Implement a linearizability checker in Rust, modeled after Knossos, to rigorously validate the correct behavior of distributed systems under test. This checker will:

- **Operation History Tracking:** Utilize Madsim's tracing or logging mechanisms to record operation invocations, responses, and real-time timestamps across simulated nodes.
- **Sequential History Validation:** Employ appropriate algorithms to determine if the observed operation history could be produced by a valid sequential execution while respecting real-time ordering constraints.
- **Transactional Safety Checking (Elle-like):** Implement a transactional safety checker in Rust, inspired by Elle's principles, to verify the consistency of transactional operations. This checker will:
  - **Invariant Definition:** Allow users to define transactional invariants, specifying the expected properties that must hold true throughout a transaction's execution.
  - **State Tracking and Monitoring:** Integrate with Madsim to monitor relevant system state during simulation runs and flag any invariant violations induced by injected faults.

## 5. Construct a Deterministic Chaos Testing Framework

- **Chaos Test Orchestration:** Design a scripting interface or programmatic API within Madsim enabling users to:
  - **Define Test Topology:** Describe the simulated network topology and configuration of the system under test.
  - **Launch the Simulated System:** Start an instance of the target distributed system within the Madsim environment.
  - **Specify Workload Generators:** Configure the types and parameters of workloads to be executed against the simulated system.
  - **Define Fault Schedules:** Create detailed fault injection scenarios with temporal sequencing for precise control during the test execution.
  - **Capture and Analyze Results:** Collect detailed operation histories and consistency checker reports for post-test analysis.
- **Test Suite Reusability:** Promote the development of reusable and modular test suites that can be shared and adapted for various distributed systems projects, fostering a culture of deterministic chaos testing best practices.

# Approach

## 1. Expanded Madsim Fault Injection Capabilities

- **Network Fault Generators**

MadJep seamlessly integrates with Madsim's network simulation infrastructure, leveraging its deterministic environment for precise fault injection. By incorporating Madsim's existing capabilities, MadJep avoids redundant development efforts and ensures compatibility with Madsim's network interception mechanisms.

### Madsim Components Utilized:

- **Network Interception:** Leveraging Madsim's network interception capabilities provided by the `madsim::interception` module.
  - **Description:** The `NetworkInterception` module provides functionality for intercepting and manipulating network packets within the simulated environment.

- **Packet Inspection:** Utilizing Madsim's packet inspection functionality to apply fault injection selectively.
  - **Description:** Madsim's packet inspection functionality allows for inspecting packet headers and payloads to apply various operations like delay or drop based on defined rules.

#### Code Example:

```

use madsim::interception::{NetworkInterception, Packet};
use madsim::fault::{DelaySpec, LossRule};

// Implementing fault injection with Madsim's network interception
fn inject_network_fault(interception: &NetworkInterception, packet: Packet, delay_spec: &DelaySpec, loss_rules: &[LossRule]) {
    // Apply delay mechanism
    let expiration = delay_spec.calculate_delay();
    interception.enqueue_packet_with_delay(packet, expiration);

    // Apply packet loss rules
    for rule in loss_rules {
        if rule.matches(packet) {
            if rule.should_drop() {
                return; // Drop the packet
            }
        }
    }

    // Forward the packet normally if no drop rule matches
    interception.forward_packet(packet);
}

```

- **Delays:**

MadJep harnesses Madsim's threading mechanisms and high-resolution timers for efficient delay handling. By utilizing Rust's standard libraries and extending Madsim's configuration interface, MadJep optimizes delay calculations and enhances flexibility in defining delay specifications.

#### Madsim Components Utilized:

- **Threading:** Leveraging Madsim's threading capabilities for parallel delay handling.
  - **Description:** Madsim's threading capabilities provide support for managing concurrent tasks within the simulation environment.
- **High-Resolution Timers:** Utilizing Rust's `std::time::Instant` for precise delay calculations.
  - **Description:** Rust's `Instant` type provides a high-resolution timer for measuring time intervals with nanosecond precision.

#### Code Example:

```

use madsim::threading::{Thread, Instant};
use madsim::config::Configuration;

// Utilizing Madsim's threading for delay handling
fn enqueue_packet_with_delay(packet: Packet, expiration: Instant) {
    // Enqueue packet with delay expiration timestamp
    queue.push(DelayedPacket { packet, expiration });
}

// Extending Madsim's configuration interface for delay specification
fn set_delay_spec(configuration: &Configuration, node: Node, delay_spec: DelaySpec) {
    configuration.set_delay_spec(node, delay_spec);
}

```

- **Packet Loss (probabilistic or targeted):**

MadJep leverages Madsim's packet interception layer for intelligent packet loss management. By integrating with Madsim's infrastructure, MadJep efficiently implements selective packet dropping based on various criteria, enhancing realism in chaos testing scenarios.

#### **Madsim Components Utilized:**

**Packet Inspection:** Utilizing Madsim's packet inspection functionality for applying packet loss rules.

- **Description:** Madsim's packet inspection layer allows for examining packet contents and applying actions like dropping packets based on predefined rules.

#### **Code Example:**

```

use madsim::interception::{Packet, LossRule};

// Implementing selective packet dropping using Madsim's packet interception layer
fn intercept_and_apply_loss_rules(packet: Packet, loss_rules: &[LossRule]) {
    for rule in loss_rules {
        if rule.matches(packet) {
            if rule.should_drop() {
                return; // Drop the packet
            }
        }
    }
    // Forward the packet normally if no drop rule matches
    forward_packet(packet);
}

```

- **Dynamic Partition Enforcement:**  
MadJep leverages Madsim's topology representation to enforce network partitions effectively. By extending Madsim's configuration syntax, MadJep enables users to define partition rules seamlessly, ensuring precise control over network behavior during chaos testing.

#### **Madsim Components Utilized:**

**Topology Representation:** Utilizing Madsim's data structures for representing the simulated network topology.

- **Description:** Madsim's topology representation provides data structures and methods for defining and manipulating the network topology within the simulation environment.

**Configuration Interface:** Extending Madsim's configuration syntax for defining partition rules.

- **Description:** Madsim's configuration interface allows users to specify various parameters and rules to configure the behavior of the simulation environment.

#### **Code Example:**

```
use madsim::topology::{Topology, Partition};
use madsim::config::Configuration;

// Extending Madsim's configuration interface for partition definition
fn set_partition_rules(configuration: &Configuration, partitions: &[Partition]) {
    configuration.set_partition_rules(partitions);
}
```

- **Message Reordering:**  
MadJep capitalizes on Madsim's simulation capabilities to implement intelligent packet reordering strategies. By leveraging Madsim's data structures and timing mechanisms, MadJep orchestrates packet reordering with precision.

#### **Madsim Components Utilized:**

**Data Structures:** Leveraging Madsim's data structures for managing packet reordering buffers.

- **Description:** Madsim provides data structures like binary heaps for efficiently managing ordered collections of packets.

**Timing Mechanisms:** Utilizing Madsim's timing mechanisms for releasing reordered packets.

- **Description:** Madsim's timing mechanisms allow for scheduling events and actions within the simulation environment with high precision.



### Code Example:

```
use madsim::data_structures::{BinaryHeap, ReorderedPacket};
use madsim::timing::TimingMechanism;

// Managing packet reordering using Madsim's data structures
fn manage_packet_reordering(buffer: BinaryHeap<ReorderedPacket>, timing_mechanism: TimingMechanism) {
    // Apply reordering rules and forward packets accordingly
}
```

- **Node Fault Generators**

MadJep leverages Madsim's process management capabilities for reliable node fault injection. By integrating with Madsim's internals, MadJep ensures seamless coordination and timing of fault injection events.

### Madsim Components Utilized:

**Process Management:** Leveraging Madsim's process management functionalities for injecting node faults reliably.

- **Description:** Madsim's process management functionalities provide mechanisms for spawning, monitoring, and controlling processes within the simulation environment.

### Code Example:

```
use madsim::process::{Process, Signal};

// Integrating with Madsim's process management for node fault injection
fn inject_node_fault(process: &Process, signal: Signal) {
    // Inject specified fault (crash, restart, etc.) into the node
}
```

### Efficient Resource Exhaustion Simulation

MadJep utilizes Madsim's simulation environment to simulate resource exhaustion scenarios efficiently. By leveraging Madsim's process management and resource allocation mechanisms, MadJep introduces CPU and memory pressure without redundant implementations.

### Madsim Components Utilized:

**Process Management:** Utilizing Madsim's process management functionalities for simulating CPU and memory pressure.

- **Description:** Madsim's process management functionalities allow for simulating resource consumption and exhaustion within individual processes.

## Code Example:

```
use madsim::process::{Process, Resource};

// Simulating CPU load using Madsim's process management
fn simulate_cpu_load(process: &Process, load_percent: f32) {
    // Utilize process's capabilities to simulate CPU load
}
```

- **Storage Fault Generators (if applicable)**
  - **Data Corruption (bit flips, truncation, etc.):**

### 1. Integration Points

**Level of Simulation:** We'll blend the best of both worlds, using internal simulation for granular WAL faults and an external approach for broader RocksDB-level faults.

- **Internal WAL Simulation (Madsim):**
  - Intercept WAL append operations.
- **External Storage (RocksDB):**
  - **File I/O Interception (if feasible):** Target WAL files.
  - **External Tooling:** Create a tool that:
    - Understands the WAL format and how it maps to RocksDB.
    - Coordinates with Madsim to trigger faults during tests.

### 2. Interception & Modification

#### Internal WAL Simulation (Madsim)

- **Intercept:** Add an interception point in storage logic before WAL append operations.
- **Modify:**
  - Bit flips (targeting specific fields within records)
  - Truncation (at record or segment level)
  - Checksum modification
  - Reordering records
  - Invalid record types
  - Simulating delayed/failed writes

```

use rand::{distributions::Alphanumeric, Rng}; // Example randomness library

// ... Your Storage Logic ...

fn intercept_and_corrupt_wal_append(&mut self, frames: Vec<DataFrame<C>>) {
    if should_corrupt(/* Configuration */) {
        let corruption_type = match self.config.fault_type {
            CorruptionType::BitFlip => self.apply_bit_flips(&frames),
            CorruptionType::Truncation => self.truncate_data(&frames),
            CorruptionType::ChecksumMod => self.modify_checksum(&frames),
            // ... Other corruption types ...
        };
        // Write corrupted_data instead
    }
}

fn apply_bit_flips(&self, frames: &Vec<DataFrame<C>>) -> Vec<DataFrame<C>> {
    let mut corrupted_frames = frames.clone();
    let target_byte = rand::thread_rng().gen_range(0..frames.len()); // For example
    let target_bit = rand::thread_rng().gen_range(0..8);
    corrupted_frames[target_byte] ^= 1 << target_bit; // Flip a single bit
    corrupted_frames
}

// ... Implement similar functions for truncation, checksum modification, etc ...

```

## External Storage (RocksDB)

- **File I/O Interception (if feasible):**
  - Use OS-level mechanisms (e.g., LD\_PRELOAD on Linux) to intercept file operations.
  - Introduce faults on WAL files/segments.
- **External Tooling:**
  - **Fault Types:**
    - WAL record corruption
    - RocksDB SSTable corruption
    - RocksDB MANIFEST corruption
  - **Integration:**
    - Trigger faults via a control channel from Madsim using RPCs or similar.

## 3. Configuration

- **Corruption Types:**
  - Bit flips (flipping random bits within a target region)
  - Data truncation (cutting off data at specific offsets)
  - Checksum modification (introducing errors in checksum fields if applicable)
  - More advanced (pattern-based corruption)
  - WAL-Specific:
    - Simulated delays in WAL writes
    - Simulated temporary or permanent WAL unavailability
    - WAL segment deletion
    - WAL segment duplication

```

enum CorruptionType {
    BitFlip,
    Truncation,
    ChecksumMod,
    Reordering, // If the format allows record reordering
    InvalidRecordType,
    DelayWrites, // Simulate slow or stalled WAL writes
    TemporaryUnavailable, // Simulate WAL errors
    SegmentDeletion,
    SegmentDuplication,
    RocksDBSstableCorruption,
    RocksDBManifestCorruption,
    // ...
}

```

- **Targeting:**
  - File level (if the external WAL has a clear structure).
  - WAL segment level (if identifiable).
  - Record level
  - Field-level within records (for granular corruption).

## 2. Consistency Verification Tools

### Linearizability Checker (Knossos-like)

- **Operation History Tracking Integration:**
  - **Mechanism:**

#### 1. Identifying Operation Boundaries

- **Define an Operation:** Clearly determine what constitutes a single operation from the perspective of your consistency model. Examples:
  - Xline Key-Value Operation: A read, a write, a compare-and-set.
  - Distributed Transaction: A sequence of operations demarcated by begin/commit boundaries.
- **Granularity:** Decide the granularity needed:
  - Operation-level: Record the type of operation (read, write, etc.)
  - Value-level: Capture the actual data read or written if required by the checker.

#### 2. Instrumentation Mechanisms

- **Madsim Instrumentation:**
  - Intercept simulated network messages:
    - Extract operation details, relevant metadata (node IDs, logical timestamps, etc.).
  - Intercept relevant system calls or internal events of the simulated processes if Madsim has that level of access.

- **External Logging:**
  - If the system under test emits detailed logs or traces:
    - Develop a parser to extract operation information from these logs.
    - Madsim would need a mechanism to correlate these external logs with its internal simulation timeline (potentially using timestamps).

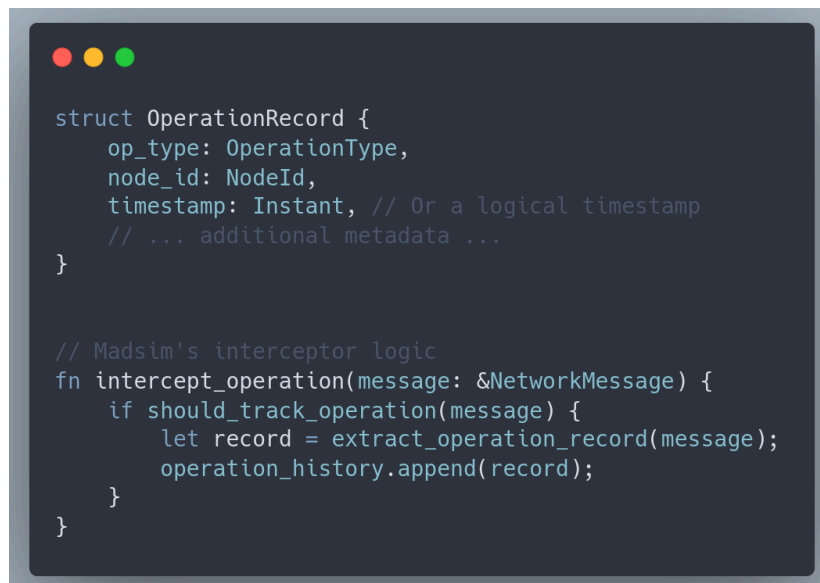
### 3. Data Structures

- **Operation History:**
  - In-memory data structure to store the operation history. Consider:
    - Append-only log for efficiency
    - Indexing or metadata association for filtering/querying during checking.

### Configuration

- **Filtering (If Necessary):** Allow users to include/exclude specific operation types or target specific simulated nodes, especially for complex tests.

### Code Example (Conceptual - Madsim Instrumentation)



```

struct OperationRecord {
    op_type: OperationType,
    node_id: NodeId,
    timestamp: Instant, // Or a logical timestamp
    // ... additional metadata ...
}

// Madsim's interceptor logic
fn intercept_operation(message: &NetworkMessage) {
    if should_track_operation(message) {
        let record = extract_operation_record(message);
        operation_history.append(record);
    }
}

```

- **Data Structures:**

#### 1. Vector Clocks

- **Representation:** For each simulated node, maintain a vector where each index corresponds to a node in the system, and the value at the index represents that node's logical clock/counter.
- **Updates:**
  - Before an operation (send/local event): Increment the logical clock of the issuing node in its vector clock.
  - Attach to Messages: Include the vector clock in simulated network messages carrying operation invocations.
  - On Receive:

- Update the receiver's vector clock by taking a component-wise maximum with the received clock
- Increment its own local clock.

## 2. Operation Logs

- **Per-Node Logs:** Maintain a log of operation events for each simulated node. Options include:
  - **Simple Append-Only:** For efficiency.
  - **Indexed Structure:** If complex queries about the history are required during analysis.
- **Event Entries:** Each entry would store:
  - Operation Type (read, write, etc.)
  - Key/Value (if applicable)
  - Vector Clock (timestamp of the operation)
  - Response (if a read operation, store the value returned)
  - Additional metadata if needed by the checker

## 3. Linearizability Checking Algorithm

- **Foundation:** Knossos-style algorithm tailored to Madsim's simulation environment.
- **Real-Time Constraints:** Consider incorporating real-time constraints into the validation logic. This is crucial, especially if Xline makes latency-related guarantees.
- **Key Validation Steps:**
  1. **Causal Consistency:** Ensure operations adhere to the partial order defined by the vector clocks (happens-before relationship).
  2. **Sequential Equivalence:**
    - Construct potential sequential histories by interleaving operations from different nodes.
    - Check if any of these sequential histories respect real-time order and produce the same results (reads observe the correct values) as the observed distributed execution.

### Code Example (Conceptual)

```
struct VectorClock {
    values: Vec<u64>,
}

struct OperationEvent {
    op_type: OperationType,
    vector_clock: VectorClock,
    // ... other fields ...
}

fn is_linearizable(history: &[OperationEvent]) -> bool {
    // ... Implementation of the checking algorithm ...
}
```

- **Validation Algorithm:**
  - **Algorithm Choice:**

Here's a breakdown of some algorithm options and factors to consider:

### 1. Original Knossos Algorithm

- **Foundation:** Constructing a Potential Real-Time Order (PRO) graph to represent potential sequential execution orders.
- **Pros:**
  - Conceptual clarity
  - Handles complex scenarios with ease
- **Cons:**
  - Computationally intensive, especially for larger operation histories. Graph operations can become a bottleneck.

#### Code Snippet: High-level (Conceptual) - Knossos

```
// Represents an execution graph node
struct PROEvent { ... }

fn is_linearizable_knossos(history: &[amp;OperationEvent]) -> bool {
    let graph = construct_pro_graph(history);
    // ... Search for cycles or other violations in the graph ...
}
```

### 2. Verification Algorithms (e.g., Adya's Algorithm)

- **Focus:** Directly verifies whether a given execution history could be produced by a sequential execution that respects real-time ordering.
- **Pros:**
  - Potentially better performance and scalability compared to graph-based approaches in many cases.
- **Cons:**
  - Can be less intuitive to understand.
  - Might have limitations in handling specific consistency models or operation types.
- **Further Details:** Adya's Algorithm is primarily an academic contribution focused on the theory and logic of linearizability verification. It is an MIT research paper, titled "Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions". The algorithm can be implemented in different ways. There's no single "standard" implementation everyone uses. In fact it is very often used without being realized that it is used. Elle itself builds upon this algorithm as you can see here: ["https://dl.acm.org/doi/pdf/10.1145/3465084.3467483"](https://dl.acm.org/doi/pdf/10.1145/3465084.3467483)

### 3. Hybrid Approaches

- **Selective Verification:** Use the original Knossos algorithm for small or straightforward test cases but switch to a faster verification algorithm for larger, more complex histories.
- **Incremental Verification with Graph Fallback:** Attempt incremental verification, but maintain a partial PRO graph in case a more detailed analysis is required for inconclusive results from the verification step.

## PREFERRED APPROACH: Porcupine-Inspired

Inspired by the efficiency gains of [Porcupine](#), let's outline a similar approach for Madsim. Key concepts from Porcupine's "P-compositionality" optimization:

- **Partitioning:** Divide histories based on independent entities (e.g., keys in a key-value store).
- **Incremental Checking:** Rather than constructing an entire graph upfront, focus on verifying linearizability within partitions and progressively compose these results.
- **Efficient Search:** Employ search techniques and data structures tailored to the structure of operation histories.

### Code Snippet: High-Level (Conceptual) - Porcupine-Inspired

```
struct Partition { ... } // Represents independent sets of operations

fn is_linearizable_porcupine_inspired(history: &[amp;OperationEvent]) -> bool {
    let partitions = partition_history(history);
    for partition in partitions {
        if !check_partition_linearizability(partition) {
            return false; // Early stopping if a partition fails
        }
    }
    // Further logic to compose results across partitions...
    return true;
}
```

## ○ Real-time Constraints:

### 1. Timestamp Integration

- **Data Structures:** Ensure that operation events in our logs store their associated real-time timestamps in addition to vector clocks (which capture logical time and causality).
- **Timestamp Sources:**
  - **Wall-clock Time:** If applicable, Madsim could use high-resolution system timers (`std::time::Instant`).
  - **Simulated Time:** If Madsim has an internal simulated clock, use that for operation timestamps.

### 2. Augmenting the Validation Logic

Here's how we'll modify the key steps of the linearizability checking algorithm (we'll assume a Knossos-style approach for illustration):



- **Causal Consistency (Unchanged):** We still need to ensure that the vector clocks accurately reflect the happens-before relationships in the observed history.
- **Sequential History Construction:** When generating potential sequential execution orders, we must respect real-time ordering:
  - The sequential order must place operations with earlier timestamps before operations with later timestamps.
  - We may need to discard or adjust the ordering of operations in the PRO graph if they violate these real-time constraints.
- **Read Validation:** Ensure that in any constructed sequential history, a read operation observes a value written by the most recent write operation *preceding* it in both real-time and logical ordering.

### Code Snippet: High-level (Conceptual)

```

struct OperationEvent {
    // ... other fields ...
    timestamp: Instant, // Real-time timestamp
}

fn is_linearizable(history: &[amp;OperationEvent]) -> bool {
    // ...
    if !respects_real_time_order(potential_history) {
        return false; // Early termination if real-time order is violated
    }
    // ... continue with other linearizability checks ...
}

```

- **Efficiency Considerations:**
  1. **Data Structures**
    - **Operation Log Representation:** While an append-only log is simple, explore more search-efficient structures (e.g., balanced trees) if your checker frequently needs to query operations based on timestamps or vector clocks.
    - **PRO Graph (if using Knossos):** Optimize the graph representation and operations (e.g., specialized data structures for cycle detection).
  2. **Early Termination**
    - **Within a Scenario:** As soon as a linearizability violation is definitively detected, stop exploring alternative sequential histories for the current operation history.
    - **Across Scenarios:** If specific patterns of operations repeatedly cause violations, consider maintaining a 'cache' of known violating patterns for quick detection in subsequent runs.
  3. **Incremental Validation**
    - **State Tracking:** Instead of fully re-analyzing the history on each new operation, maintain auxiliary data structures that allow for incremental updates to the checker state.
    - **Partial PRO Graph:** If using the Knossos approach, incrementally update the PRO graph instead of rebuilding it from scratch with each new event.
  4. **Filtering and Pruning**

- **Operation Filtering** Configure the checker to focus only on subsets of operations relevant to the specific properties under test.
  - **History Pruning:** Implement strategies to remove operations from the history that are no longer relevant for validation (e.g., keeping only a window of recent operations, or pruning operations definitively ordered by both real-time and logical time)
5. **Parallelism (if applicable)**
- **Distributed Checking** If working in a truly distributed simulation environment, segments of the operation history could be analyzed concurrently on different Madsim nodes.
  - **Multithreading** Utilize multiple threads or tasks within a single Madsim process, especially for graph-related operations in the Knossos algorithm.

#### Code Snippet: (Conceptual) - Incremental State

```

struct CheckerState {
    // Data structures to track reads, writes, potential conflicts, etc.
}

fn update_state(state: &mut CheckerState, event: &OperationEvent) {
    // ... modify state based on the event ...
}

fn is_linearizable(history: &[OperationEvent]) -> bool {
    let mut state = CheckerState::new();
    for event in history {
        update_state(&mut state, &event);
        if state.is_violation_detected() {
            return false;
        }
    }
    // ... additional checks only if no early violation detected ...
}

```

#### Transactional Safety Checker (Elle-like)

- **Invariant Definition:**
  - **Language:**

##### 1. Domain-Specific Language (DSL)

- **Syntax:** Create a concise syntax for expressing common transactional predicates.  
Examples:
  - `read(key) == value` (Read value matches expectation)
  - `balance > 0` (Account balance constraint)
  - `count(items) >= 5` (Minimum collection size)

- **Grammar:** Design a small grammar formally defining the DSL.
- **Parser:** Implement a lightweight parser to convert invariant strings into an internal representation.

### Example DSL Usage

```
invariant "account transfer": {
    old_balance(A) + old_balance(B) == new_balance(A) + new_balance(B)
}
```

## 2. Rust API

- **Predicate Functions:** Provide a set of Rust functions for building logical predicates.
  - `read(key).equals(value)`
  - `variable("balance").greater_than(0)`
  - `collection("items").size().at_least(5)`
- **Composition:** Allow combining predicates using logical operators (`and`, `or`, `not`).

### Example API Usage

```
let invariant = read("account_A").equals(100)
                .and(read("account_B").equals(50));

add_invariant("transfer_check", invariant);
```

### Internal Representation

- **Abstract Syntax Tree (AST):** Consider using an AST-like structure to represent the parsed invariants, allowing easy evaluation at runtime.

### Code Snippet: Conceptual (AST-like)

```
enum Predicate {
    Equals(String, Value), // key, value
    GreaterThan(String, Value),
    // ... other predicates ...
    And(Box<Predicate>, Box<Predicate>),
}
```

- **Invariant Scope:**

Here's a flexible way to support different scoping levels and how we might represent them:

## 1. Scope Levels

- **Object-level:** The invariant applies to a single object or entity within the simulated system (e.g., a single bank account).
- **Group-level:** The invariant involves multiple related objects (e.g., the total balance across all accounts remains constant).
- **System-wide:** The invariant expresses a constraint about the global state of the entire system (e.g., the number of items in a replicated data store must match across all replicas).

## 2. Mechanisms for Specifying Scope

### a) DSL

- **Keywords:** Introduce keywords or modifiers to the DSL:
  - `on object(account_A): balance >= 0`
  - `for group(accounts): sum(balance) == initial_total`
  - `system_wide: num_items(store) == expected_count`

### b) Rust API

- **Method Chaining:** Provide methods to scope an invariant.
  - `invariant.on_object("account_A").check(balance().greater_than(0))`
  - `invariant.on_group("accounts").check(sum("balance").equals(initial_total))`

## 3. Internal Representation

- **Scope Data Structure:** Augment the AST-like representation of invariants to include a scope specifier.

### Code Example (Conceptual)

```
enum InvariantScope {
    Object(String), // Object identifier
    Group(Vec<String>), // List of object identifiers
    SystemWide,
}
```

- **State Tracking:**
  - **Integration Points:**

### 1. Strategic Integration Points

- **Before and After Operation Execution:**
  - **Pre-operation:** Capture the system state just before an operation is applied.
  - **Post-operation:** Access the updated system state after the operation execution.
  - **Use Case:** Checking that operations uphold invariants, especially relevant for transactional behavior.
- **Fault Injection:**
  - **Before:** Capture relevant state to compare with the post-fault state.
  - **After:** Verify invariants after a fault is injected to detect unintended side effects.
- **Workload-specific Events:** Identify key events within your workloads (e.g., transaction begin/commit boundaries if working specifically with a transactional system). Check invariants at these junctures for a higher-level view of consistency.

## 2. Implementation Mechanisms

- **Madsim Interceptors:** If Madsim provides hooks into simulated network communications or operation processing, leverage these interception points to:
  - Trigger state capture.
  - Execute invariant checks.
- **State Snapshotting (If Necessary):** If you need to compare pre- and post-states, consider a state snapshotting mechanism within Madsim. Options range from full-system snapshots for maximum flexibility to targeted snapshots based on the scope of invariants.
- **User-defined Events (If Applicable):** If the simulated system supports custom events or signaling, these could be used to trigger invariant checks integrated with Madsim's mechanisms.

## Code Snippets (Conceptual)

```
// Assuming Madsim has an interceptor for operations
fn intercept_operation(op: &Operation, state_manager: &mut StateManager) {
    let pre_state = state_manager.snapshot();
    // ... apply operation ...
    let post_state = state_manager.snapshot();

    for invariant in active_invariants {
        invariant.evaluate(pre_state, post_state);
    }
}
```

- **Memory Representation:**

### 1. Understanding State

- **Data Structures:** Analyze the key data structures used by the simulated system (e.g., objects, key-value maps, collections). These will influence the design of the state snapshot mechanism.

- **Shared State:** Identify state elements shared or referenced across multiple components to avoid unnecessary duplication in snapshots.

## 2. Techniques for State Representation

- **Selective Snapshotting:** Focus on capturing only the portions of the system state actively referenced by the defined invariants. This reduces memory overhead.
- **Copy-on-Write:** For shared or large data structures:
  - Initially, use references or pointers in snapshots.
  - Perform a full copy only when a modification occurs that would affect a previously captured snapshot.
- **Delta Snapshots:** Instead of a full copy on each snapshot, consider storing incremental changes (deltas). This can be efficient if the changes between relevant snapshots are typically small.
- **Custom Serialization (if applicable):** If the system's state has a well-defined format, a compact serialized representation for snapshots could be memory efficient.

## 3. Data Structure Choices

- **Hash Maps:** For quick access to state elements by identifier (keys, object IDs).
- **Trees:** To represent hierarchical relationships within the state.
- **Versioned Data Structures:** Explore libraries providing immutable or versioned data structures, potentially simplifying the management of multiple snapshots.

### Code Example (Conceptual)

```
struct Snapshot {
    shared_objects: HashMap<ObjectId, Rc<dyn Snapshottable>>, // Shared state, reference counted
    key_values: HashMap<String, Value>,
    // ... other fields ...
}

trait Snapshottable {
    fn snapshot(&self) -> Self; // Object knows how to create its snapshot
}
```

- **Invariant Checking and Reporting:**
  - **Evaluation Frequency:**

### 1. Factors to Consider

- **Invariant Criticality:** Some invariants might be fundamental to the system's correctness and should be evaluated very frequently. Others may be secondary checks that can tolerate less frequent examination.
- **Performance Impact:** Checking invariants has an overhead. Balance the need for timely violation detection with potential performance implications.

- **Nature of the Test Scenario:**
  - **Focused Fault Testing:** More frequent checks can help pinpoint the exact operation that triggers an issue in fault-heavy scenarios.
  - **WAN Simulation:** High latencies might necessitate less frequent checks to avoid slowing down the simulation significantly.

## 2. Strategies

- **Operation-level:** After every simulated operation for maximum tightness.
  - **Best for:** Catching violations immediately, critical invariants.
- **Periodic Checkpoints:** Evaluate invariants at regular intervals within the simulation (based on real-time or a simulated time measure).
  - **Best for:** Balancing detection with overhead.
- **Event-driven:** Trigger evaluation at key events:
  - Transaction boundaries
  - Before/after fault injection
  - Workload-specific events
- **Hybrid:** Combine the above. For example, high-priority invariants on every operation, broader invariants at periodic checkpoints.

## 3. Configuration

Allow users to configure the evaluation frequency through Madsim's API or scripting interface:

```
madsim::testing::add_invariant("account_balance_check")
    .evaluate_on_every_operation();

madsim::testing::add_invariant("system_consistency_check")
    .evaluate_every(Duration::from_millis(500)); // Every 500ms of simulation time
```

### Code Snippet (Conceptual - Integration)

```
// Within Madsim's simulation loop
for operation in operations {
    apply_operation(operation);
    evaluate_relevant_invariants();
}
```

- **Reporting Format:**

## 1. Essential Report Elements

- **Invariant Identifier:** Clearly indicate which invariant was violated.
- **Violation Context:**
  - The operation that triggered the violation (type, relevant arguments).
  - Timestamp within the simulation.
  - State snapshot at the time of violation (or a way to access it).
- **Expected vs. Actual State:** Present the values or conditions that led to the violation, highlighting the discrepancy.
- **Event Trace (Optional):** If event tracing is enabled, include a relevant portion of the operation history leading up to the failure.

## 2. Reporting Formats

- **Structured Data:** Consider JSON or XML for structured violation reports. This enables easy parsing and analysis by external tools.
- **Human-Readable Text:** Generate a human-readable summary of the violation for immediate user understanding.

## 3. Reporting Mechanisms

- **Logging:** Log violation reports to files or a logging system.
- **Callbacks:** Allow users to provide custom callback functions to be triggered on violations, enabling integration with other analysis or visualization tools.
- **Integrated UI (If Applicable):** If Madsim has a graphical interface, present reports directly.

### Code Example (Conceptual)

```
struct ViolationReport {
    invariant_name: String,
    timestamp: Instant,
    operation: Operation,
    state_snapshot: StateSnapshot,
    expected: String, // e.g., "balance >= 0"
    actual: String, // e.g., "balance = -10"
}

impl ViolationReport {
    fn to_json(&self) -> String { /* ... */ }
    fn to_text(&self) -> String { /* ... */ }
}
```

## 3. Expand Madsim's Chaos Engineering Capabilities

### Rich Fault Injection Toolkit



Here's a breakdown of the fault categories, potential implementation approaches, and configuration considerations:

## Network Faults

- **Delays:**

- **Techniques:**

### 1. Thread Sleeping

- **Implementation:**

- Within Madsim's message handling logic, intercept an outgoing message.
- Calculate the desired delay
- Use `std::thread::sleep(Duration::from_millis(delay_ms))` or similar to pause the current thread.
- Resume and forward the message.

- **Advantages:**

- Simple to implement.

- **Drawbacks:**

- Blocks the message handling thread, potentially impacting throughput if you have many parallel message flows or very long delays.
- Introduces imprecision from OS thread scheduling.

### 2. Selective Queueing

- **Implementation**

- Introduce a message queue (bounded or unbounded) with associated timestamps in Madsim's network simulation.
- When intercepting a message, calculate the delay and enqueue it along with an expiration timestamp.
- Have a dedicated thread or task that periodically:
  - Checks the head of the queue.
  - Forwards messages whose expiration timestamp has passed.

- **Advantages:**

- Doesn't block the main message handling threads.
- Potential for more precise control using high-resolution timers.

- **Considerations**

- Increases implementation complexity slightly.
- Queue size management (bounded queue) if under memory constraints.

### Code Example (Conceptual - Selective Queueing)

```

struct DelayedMessage {
    message: Message,
    expiration: Instant,
}

let mut delay_queue: PriorityQueue<DelayedMessage> = PriorityQueue::new();

// Thread for processing delayed messages (simplified)
fn delay_dispatcher() {
    loop {
        let now = Instant::now();
        while let Some(msg) = delay_queue.peek() {
            if msg.expiration <= now {
                let msg = delay_queue.pop().unwrap();
                forward_message(msg.message);
            } else {
                break; // Queue is ordered by expiration
            }
        }
        // ... sleep or yield ...
    }
}

```

- **Configuration:**

### 1. Delay Distributions

- **Library Integration:** Utilize a Rust probability distribution library like `rand_distr` to generate delays from standard distributions.
- **Configuration API:**

```

madsim::fault::network_delay! {
    delay_spec: DelaySpec::Distribution(Distribution::Normal {
        mean: Duration::from_millis(500),
        std_dev: Duration::from_millis(100)
    }),
    // ... other parameters ...
}

```

### 2. Message Filtering

- **Rules:** Define a rule-matching mechanism. Potential criteria:
  - Source and destination node identifiers
  - Message types (if Madsim models different protocol messages)
  - Possibly, packet metadata if Madsim has deep packet inspection capabilities
- **Configuration API:**

```

madsim::fault::network_delay! {
  rules: [
    DelayRule::Match { source_ip: "192.168.1.10", packet_type: PacketType::TCP, ... },
    // More complex rules
  ]
}

```

### 3. Linking to Implementation

- Pass the configured delay specifications and filtering rules into Madsim's delay mechanisms (the thread sleeping or queuing approach we discussed earlier).

#### Code Example (Conceptual - Configuration Focus)

```

enum DelaySpec {
  Fixed(Duration),
  Distribution(Distribution), // From a probability distribution library
}

struct DelayRule {
  source_ip: Option<String>,
  destination_ip: Option<String>,
  // ... other criteria ...
}

impl DelayRule {
  fn matches(&self, message: &Message) -> bool { /* ... */ }
}

```

- **Partitions:**
  - **Techniques:**

#### 1. Topology Representation

- **Data Structure:** Madsim likely maintains an internal representation of the simulated network topology. This could be:
  - A graph (adjacency matrix or adjacency list)
  - A set of routing tables
- **Augmentation:** Ensure this data structure allows you to:
  - \* Dynamically mark node groups as belonging to separate partitions.
  - \* Efficiently query whether two nodes are in the same partition or not.

#### 2. Message Routing Modification

- **Interception Points:** Identify where in Madsim's message handling logic the routing decisions are made.
- **Partition Checks:** Within this routing logic, before forwarding a message:
  1. Determine the source and destination nodes.
  2. Query the topology representation to see if they belong to separate, active partitions.
  3. If partitioned, silently drop the message instead of forwarding it.

### 3. Dynamic Control

- Allow activation and deactivation of partitions through Madsim's API or scripting interface. This likely involves updating the internal topology representation.

### Code Snippet (Conceptual)

```

struct Topology {
    partitions: Vec<Vec<NodeId>>, // Nodes grouped into partitions
    // ...
}

impl Topology {
    fn are_connected(&self, node_a: NodeId, node_b: NodeId) -> bool {
        // Check if the nodes belong to the same partition
    }
}

// Inside Madsim's message routing (simplified)
fn route_message(message: Message, topology: &Topology) {
    let source = message.source_node;
    let destination = message.destination_node;

    if !topology.are_connected(source, destination) {
        return; // Drop message due to partition
    }

    // ... normal forwarding logic ...
}

```

## ○ Configuration

### 1. Static Partition Definition

- **Configuration API:** Provide a way to define partitions within Madsim's configuration or scripting.
- **Partition Specification:**
  - **Groups of nodes:** Explicit lists of node IDs or identifiers.
  - **Labeling (Optional):** Allow assigning names or labels to partitions for easier reference.

### Example Configuration

```

madsim::fault::partition! {
  groups: [ [node1, node2, node3], [node4, node5] ], // Two partitions
}

```

## 2. Internal Representation

Amend Madsim's internal topology data structure to store the defined partition information. Consider how this aligns with your chosen topology representation:

- **Graph:** Store partitions as sets of node IDs.
- **Routing Tables:** Potentially augment routing tables with partition restrictions.

## 3. Dynamic Reconfiguration (Optional)

- **API:** Extend Madsim's API to support commands like:
  - `madsim::fault::activate_partition("partition1")`
  - `madsim::fault::deactivate_partition("partition2")`
  - `madsim::fault::modify_partition("partition1",  
add_nodes=[node6], remove_nodes=[])`
- **State Management:** Update the internal topology representation based on these commands.

### Code Snippet (Conceptual - API Focus)

```

enum PartitionSpec {
  Static(Vec<Vec<NodeId>>), // For initial configuration
  Dynamic(Vec<PartitionCommand>),
}

enum PartitionCommand {
  Activate(String), // Partition name/label
  Deactivate(String),
  Modify { name: String, add_nodes: Vec<NodeId>, remove_nodes: Vec<NodeId> },
}

```

- **Packet Loss:**
  - **Techniques:**

### 1. Integration Point

Identify the appropriate location within Madsim's network handling logic where you can intercept outgoing messages. This is likely the same point as where you'd integrate delays.

### 2. Probabilistic Dropping

- **Random Number Generation:** Use Rust's `rand crate (rand::thread_rng())` to generate random numbers.
- **Drop Threshold:** In your configuration, allow specification of a drop probability (e.g., 0.2 for a 20% chance of dropping).
- **Logic:** For each intercepted message:
  1. Generate a random number between 0 and 1.
  2. If the number is less than the configured drop probability, silently discard the message.

### 3. Targeted Dropping

- **Filtering Rules:** Similar to the delay configuration, implement a mechanism for defining message filtering rules. These could be based on:
  - Source/destination node identifiers
  - Message/packet types
  - Potentially packet metadata (if Madsim allows that level of inspection)
- **Logic:** If a message matches a drop rule, discard it.

### Configuration Example

```
madsim::fault::packet_loss! {
  rules: [
    LossRule::Probabilistic { probability: 0.15 }, // 15% overall drop rate
    LossRule::Match { source_ip: "192.168.1.10", destination_ip: "192.168.2.5" }, // Targeted drop
  ]
}
```

### Code Snippet (Conceptual)

```
fn intercept_and_apply_faults(message: Message, loss_rules: &[LossRule]) {
  for rule in loss_rules {
    if rule.matches(message) {
      if rule.should_drop() {
        return; // Drop the packet
      }
    }
  }

  // ... normal forwarding
}
```

- **Configuration:**
  1. **Granular Configuration**
    - **Global Probability:** Continue supporting a global default loss probability.

- **Rule-based Overrides:** Allow users to define rules with more specific loss probabilities. Link these rules to:
  - **Source/Destination Pairs:** For simulating unreliable links between specific nodes.
  - **Message/Packet Types:** To model protocols with differing susceptibility to packet loss.

## 2. Configuration API

Example:

```
madsim::fault::packet_loss! {
  default_probability: 0.05, // 5% overall
  rules: [
    LossRule::Match { source_ip: "192.168.1.10", destination_ip: "192.168.2.5", probability: 0.2 }, // 20%
    LossRule::Match { packet_type: PacketType::UDP, probability: 0.1 },
  ]
}
```

## 3. Internal Representation

- Store the default loss probability.
- Maintain a list or lookup structure for the configured `LossRules`

## 4. Applying Loss Logic

- In your message interception, first check if a matching `LossRule` exists for the message.
- If a rule exists, use its probability. Otherwise, fall back to the default probability.

### Code Example (Conceptual)

```
// ... inside intercept_and_apply_faults function ...

let drop_probability = match find_matching_rule(message, loss_rules) {
  Some(rule) => rule.probability,
  None => default_loss_probability,
};

if rand::thread_rng().gen:::<f64>() < drop_probability {
  return; // Drop
}

// ... forward the message ...
```

- **Reordering:**
  - **Techniques:**

### 1. Reordering Buffer

- **Data Structure:** Select a suitable data structure to act as the reordering buffer on the receiving simulated nodes. Consider:
  - **Bounded Queue:** For simple reordering within a fixed window of messages.
  - **Priority Queue:** To enable more complex reordering patterns based on timestamps, logical clocks, or custom priorities.
- **Metadata:** Associate metadata with each buffered message:
  - Arrival timestamp (for time-based reordering)
  - Source node identifier
  - Potentially logical clock values (if applicable)

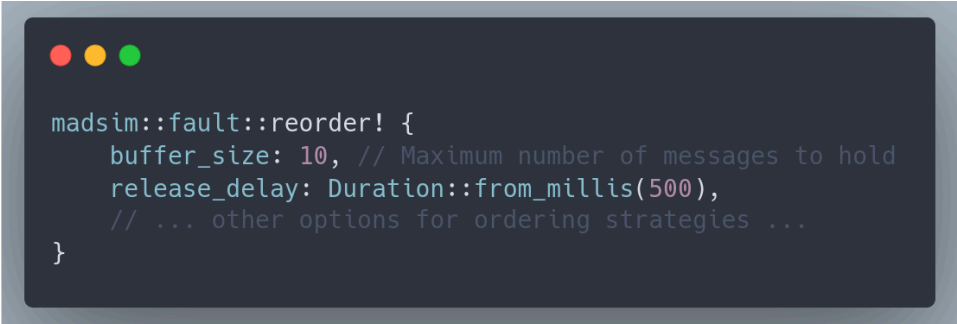
## 2. Message Interception (Receiver Side)

- Intercept incoming messages at the receiver within Madsim's network simulation.
- Place the message into the reordering buffer along with associated metadata.

## 3. Release Logic

- **Triggers:** Determine how Madsim will decide when to release messages from the buffer:
  - **Time-based:** Periodically release messages after a delay or when they exceed a configurable age.
  - **Count-based:** Release when the buffer reaches a certain count threshold.
  - **On-Demand (if applicable):** If the simulated system has a mechanism for receivers to signal readiness (e.g., polling), Madsim could integrate with that.
- **Ordering Strategy:** When releasing messages:
  - **Simple:** FIFO, potentially with a defined delay before releasing.
  - **Randomized:** Introduce random shuffling of messages within the buffer.
  - **Pattern-based:** Implement more complex reordering patterns (e.g., occasional bursts of reordering)

## Configuration Example



```
madsim::fault::reorder! {
  buffer_size: 10, // Maximum number of messages to hold
  release_delay: Duration::from_millis(500),
  // ... other options for ordering strategies ...
}
```

## Code Snippet (Conceptual)



```

struct ReorderedMessage {
    message: Message,
    arrival_time: Instant,
    // ... other metadata ...
}

// Madsim's receiver-side message handling
fn process_incoming_message(message: Message) {
    reorder_buffer.enqueue(ReorderedMessage::new(message));
    // ... logic to periodically release messages from reorder_buffer ...
}

```

- **Configuration:**

### 1. Reordering Window

- **Configuration Parameter:** Introduce a `reordering_window` parameter in Madsim's configuration.
- **Interpretation:** This window could represent:
  - **Maximum Delay:** Messages may be held for up to this duration before being released, potentially out of order.
  - **Buffer Size (Indirectly):** If using a simple bounded queue as the reordering buffer, the window size and expected message arrival rate would implicitly determine a buffer size.

### 2. Reordering Strategies

- **Configuration Syntax:** Allow the user to select a reordering strategy. Examples:

```

madsim::fault::reorder! {
    reordering_window: Duration::from_millis(500),
    strategy: ReorderingStrategy::Random,
    // ... other options ...
}

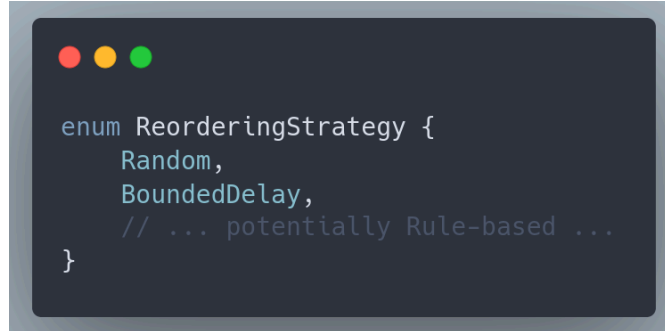
```

- **Implementations:** Support the following strategies, potentially adding more later:
  - **Random:** Random shuffling of the messages within the buffer.
  - **Bounded Delays:** Hold each message for a random delay (up to the `reordering_window`).
  - **Rule-based (if applicable):** If the simulated protocol provides enough metadata (e.g., sequence numbers) rules for complex reordering patterns become possible.

### 3. Linking to Implementation

The chosen `ReorderingStrategy` will influence the release logic within your reordering buffer implementation.

### Code Snippet (Conceptual - Configuration Focus)



```
enum ReorderingStrategy {  
    Random,  
    BoundedDelay,  
    // ... potentially Rule-based ...  
}
```

## Node Faults

- **Controlled Crashes:**
  - **Techniques:**

### 1. Node Representation

- **Understanding:** Clarify how Madsim represents simulated nodes internally:
  - Are they separate processes?
  - Threads within a single Madsim process?
  - Lightweight data structures with associated simulated state?
- **Termination Mechanism:** The crash implementation will depend on Madsim's node representation.

### 2. Integration with Lifecycle Management

- **Assumptions:** We'll assume Madsim has some form of node lifecycle management, responsible for starting and potentially tracking the health of simulated nodes.
- **Triggers:** There are two main ways to trigger crashes:
  - **Explicit Crash Command:** Madsim's API or scripting interface provides a command like `madsim::fault::crash_node("node3")`
  - **Event-Based:** Define events within Madsim that can trigger crashes (e.g., after a certain simulated time, as part of a complex scenario, or based on other fault injections).
- **Signaling:** Madsim's fault injection mechanism needs to signal the node lifecycle management component to initiate the crash.

### 3. Crash Types

- **Graceful Shutdown:** Simulate the node process sending appropriate shutdown messages (if the simulated system supports that) before termination.
- **Abrupt Termination:** Forcibly kill the node process or thread, simulating an unexpected crash.

### Code Snippet (Highly Conceptual)

```

// Madsim's internal node representation
struct SimulatedNode {
    process: Option<Child>, // If node is a separate process
    state: NodeState,
}

// Fault injection API
madsim::fault::crash_node("node3", signal: SIGKILL);

// Inside Madsim's event handling
if should_crash_node(/* ... */) {
    node_manager.terminate_node("node3");
}

```

- **Configuration**

### 1. Node Targeting

- **API:** Extend Madsim's fault injection API or scripting commands to support:
  - **Individual Nodes:** Crashing a node by its ID or identifier.
  - **Node Groups (Optional):** If convenient, allow specifying groups of nodes to crash simultaneously.
- **Configuration Example:**

```

madsim::fault::crash_node("node3");

madsim::fault::crash_nodes(group: "clusterA"); // If node grouping is supported

```

### 2. Crash Timing

- **Immediate:** Execute the `crash_node` command as soon as it is issued.
- **Scheduled:**
  - Introduce a `schedule_crash` command.
  - **Parameters:** Node identifier and a simulated timestamp representing when the crash should occur.
- **Event-Based Triggers:**
  - Define a way to link crash commands to events within the simulation. Examples:
    - `after_duration(Duration::from_secs(300)) => crash_node("node1")`
    - `on_event(EventType::NetworkPartition) => crash_nodes(group: "clusterB")`

### 3. Configuration Mechanism

- Choose between Madsim's configuration files, its scripting interface, or ideally provide both options for flexibility.

#### Code Snippet (Conceptual - API Focus)

```
madsim::fault::crash_node("node3");  
  
madsim::fault::schedule_crash("node1", at: Instant::now() + Duration::from_secs(60));  
  
madsim::fault::on_event("DatabaseFailure") => crash_node("database1"); // Assuming Madsim has an event mechanism
```

#### • Restarts:

##### 1. Integration with Node Management

- **State Persistence:** Determine if Madsim needs to persist any state for crashed nodes to enable recovery upon restart. This will likely depend on the nature of the system you're simulating.
- **Coordination:** Establish a mechanism for Madsim's fault injection subsystem to signal its node lifecycle management component to initiate a restart.

##### 2. Restart Process

- **Re-initialization:** Madsim's node lifecycle management will need the ability to launch a new instance of the node process or thread. Consider:
  - Passing any persisted state to the restarted node.
  - Potentially simulating a clean "cold start" if no state is needed.

##### 3. Configuration

- **Restart Command:** Expose a command in Madsim's API or scripting interface. Examples:
  - `madsim::fault::restart_node("node3")`
  - `madsim::fault::auto_restart_nodes(group: "clusterA")` // If applicable
- **Restart Delay:** Add a configuration parameter to control the delay.

```
madsim::fault::restart_node("node1", after: Duration::from_secs(30))
```

#### Code Snippet (Conceptual)

```

madsim::fault::restart_node("node2", after: Duration::from_secs(10));

// Inside Madsim's node lifecycle management
fn restart_node(node_id: NodeId, delay: Option<Duration>) {
    if let Some(duration) = delay {
        schedule_node_start(node_id, at: Instant::now() + duration);
    } else {
        start_node_immediately(node_id);
    }
}

```

- **Resource Exhaustion**

- **Techniques:**

1. **CPU-Bound Tasks**

- **Busy Loops:** Within the simulated node, spawn threads or tasks that execute computationally intensive loops. Avoid complex calculations if the goal is pure CPU consumption and not specific workload behavior.
- **Control:** Provide configuration on:
  - Target Node(s)
  - Intensity (percentage of a core to consume, number of threads)

2. **Memory Pressure**

- **Large Allocations:** Within the simulated node, dynamically allocate large blocks of memory (e.g., using `vec![]` with dummy data). Hold onto these allocations to maintain pressure.
- **Simulated Leaks (Less Controllable):** Introduce patterns that mimic memory leaks (forgetting to deallocate, circular references) if your simulation and memory management tools allow it.
- **Configuration:**
  - Target Node(s)
  - Amount of memory to consume or a target 'memory utilization' percentage.

3. **I/O Limits (If Applicable)**

- **Network Throttling:** If relevant, integrate with Madsim's network simulation to limit network bandwidth or selectively introduce large delays for specific nodes.
- **Storage Simulation:** If Madsim has a level of storage simulation,
  - Introduce artificial delays on disk operations.
  - Simulate disk space constraints by limiting the available space reported to the simulated node.

### Configuration (Conceptual Example)

```

madsim::fault::resource_pressure {
  node: "node2",
  cpu_utilization: 0.8, // Consume 80% of a core
  memory: {
    target_usage: 500, // In MB
  }
}

```

### Code Snippet (Highly Conceptual)

```

fn simulate_cpu_load(target_utilization: f32) {
  // ... calibrate loop iterations
  loop { /* busy work */ }
}

```

#### ○ Configuration:

##### 1. Resource Type Selection

- **Configuration Syntax:** Within Madsim's configuration or scripting language, introduce a way to specify the type of resource to target.

```

madsim::fault::resource_pressure {
  node: "node2",
  cpu_utilization: 0.8,
  memory: {
    target_usage: 500, // In MB
  }
  // Potentially add network or I/O sections if supported
}

```

##### 2. Intensity Levels

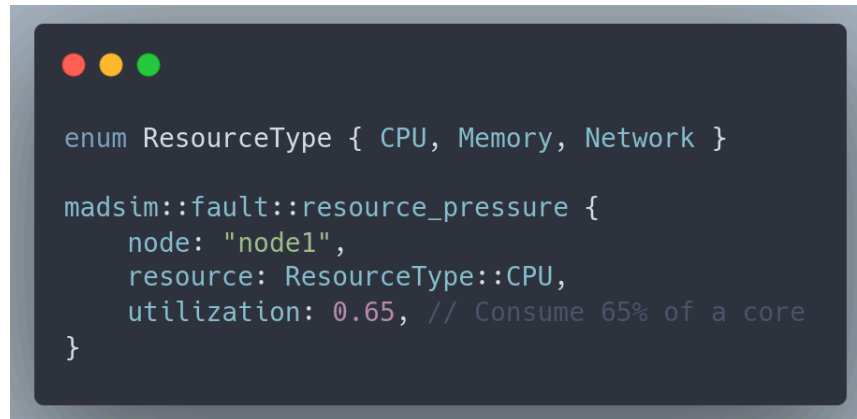
- **Parameterization:** For each supported resource type, provide parameters to control the intensity of the exhaustion. Consider:
  - **CPU:** Percentage of a core to consume, a number of threads to spawn.
  - **Memory:** Target amount of memory to consume (in bytes or MB), a percentage of simulated node memory.
  - **I/O (If Applicable):** Bandwidth limits, latency increase amounts.

- **Scaling:** Ensure the intensity parameters have sensible ranges and map well to the actual effects on the simulated system. This might require some experimentation.

### 3. Configuration Mechanism

Madsim likely already has a mechanism for defining faults (configuration files, scripting). Extend this mechanism to support resource exhaustion configuration.

#### Code Example (Conceptual - Configuration Focus)



```
enum ResourceType { CPU, Memory, Network }

madsim::fault::resource_pressure {
  node: "node1",
  resource: ResourceType::CPU,
  utilization: 0.65, // Consume 65% of a core
}
```

### Storage Faults

- **Data Corruption:**
  - **Techniques:**

#### Integration Point

1. **Storage Simulation Level:** Determine how deeply Madsim simulates storage:
  - **External Storage Simulation (e.g. RocksDB):** Madsim may interact with an external key-value store or database instance. Corruption would likely need to happen outside Madsim, via a separate tool coordinated with the test execution.
  - **Madsim-internal Storage:** If Madsim models storage structures itself, it provides a direct point for injecting corruptions.

#### Corruption Types

1. **Bit Flipping:**
  - Madsim would need the ability to intercept write operations (or read and modify data in-place).
  - Select random bits within the data to flip.
2. **Data Truncation:**
  - Intercept writes and truncate data at specific offsets.
3. **Invalid Values:**
  - If the data has a defined structure, overwrite fields with values that violate its schema or constraints.

### Configuration

- **Target:**
  - Specific files, blocks, or key-value pairs (if Madsim's storage abstraction allows this level of targeting).
- **Corruption Types:** Allow the user to select the types of corruption to apply.
- **Intensity:**
  - Frequency of corruption events during a test run.
  - Probability of a write operation being corrupted.

### Code Example (Highly Conceptual - Internal Storage)

```

struct SimulatedStorage {
    // ...
}

impl SimulatedStorage {
    fn write(&mut self, data: &[u8], offset: usize) {
        if should_corrupt(/* config */) {
            let corrupted_data = apply_corruption(data, /* config */);
            // ... write corrupted_data instead ...
        }
    }
}

```

- **Configuration:**

#### 1. Corruption Locations

- **Granularity:** Decide the level of targeting precision supported, considering Madsim's storage abstraction:
  - **File / Block-level:** If Madsim's storage simulation exposes files or block devices.
  - **Key-value:** For key-value stores (if applicable).
  - **Fine-grained:** If possible, allow targeting ranges within files or values.
- **Configuration Syntax:**

```

struct SimulatedStorage {
    // ...
}

impl SimulatedStorage {
    fn write(&mut self, data: &[u8], offset: usize) {
        if should_corrupt(/* config */) {
            let corrupted_data = apply_corruption(data, /* config */);
            // ... write corrupted_data instead ...
        }
    }
}

```



## 2. Corruption Types

- **Enumeration:** Provide a way to select from supported corruption types.

```
enum CorruptionType { BitFlip, Truncation, InvalidValue }

madsim::fault::corrupt_storage {
    // ...
    corruption_type: CorruptionType::BitFlip,
}
```

## 3. Configuration Mechanism

This will likely build upon Madsim's existing mechanisms for defining faults injections, wherever they are configured (configuration files, scripting language, etc.).

### Code Snippet (Conceptual - Configuration Focus)

```
enum StorageTarget { File(path: String), BlockDevice(id: u32), KeyValue { store: String, key: String } }
```

- **Simulated Disk Failures**

- **Techniques:**

### 1. Integration Point

- **Storage Abstraction:** Identify where Madsim interacts with its simulated storage. This could be:
  - Calls within a simulated file system layer in Madsim.
  - Interaction with a simulated block device.

### 2. Intermittent Write Failures

- **Injection Logic:** At the interception point, introduce probabilistic failure.
  - **Configuration:** Allow the user to specify a failure probability (e.g., 10% of writes fail).
  - **Randomness:** Use a random number generator ( `rand::thread_rng()` ) to determine if a write should be artificially failed.
- **Failure Simulation:** Return an appropriate error code or signal to the caller within Madsim, mimicking disk write failure.

### 3. Artificial Disk Delays

- **Injection Logic:** At the interception point (for reads and/or writes), selectively introduce delays.

- **Configuration:** Allow specification of delay distributions or fixed delays.
- **Delay Mechanism:** Use thread sleeping (`std::thread::sleep`) or similar to introduce the delay.

### Configuration Example (Conceptual)

```

madsim::fault::disk_io {
  write_failure_probability: 0.2,
  delay: DelaySpec::Distribution(Distribution::Normal {
    mean: Duration::from_millis(500),
    std_dev: Duration::from_millis(100)
  })
}

```

### Code Snippet (Highly Conceptual - Assuming File System Interception)

```

fn simulated_write(file: &mut SimulatedFile, data: &[u8]) -> Result<(), SimulatedIOError> {
  if should_fail_write() {
    return Err(SimulatedIOError::WriteFailed);
  }

  if should_introduce_delay() {
    let delay = calculate_delay();
    std::thread::sleep(delay);
  }

  // ... normal write logic ...
}

```

- **Configuration:**

#### 1. Failure Probability

- **Parameter:** Introduce a configuration parameter to set the failure probability (between 0 and 1). This could be global or disk-specific.

#### 2. Targeted Disks

- **Assumptions:** We'll assume Madsim allows you to identify or name simulated disks or volumes in some way.
- **Configuration Syntax:**

```

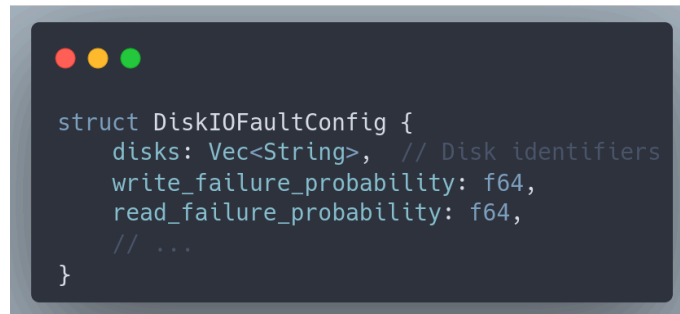
madsim::fault::disk_io {
  disks: [ "disk1", "volume_data" ], // Target these disks
  write_failure_probability: 0.15,
  // ... other options ...
}

```

### 3. Linking to Implementation

- Madsim's fault injection mechanism should store the configured failure probabilities and list of target disks.
- The I/O interception logic will need to:
  1. Check if the current operation's disk is in the target list.
  2. Use the appropriate failure probability.

#### Code Snippet (Conceptual - Configuration Focus)



```
struct DiskIOFaultConfig {  
    disks: Vec<String>, // Disk identifiers  
    write_failure_probability: f64,  
    read_failure_probability: f64,  
    // ...  
}
```

### Flexibility

- **Configuration API:**

#### 1. Configuration Methods

Consider supporting a combination of these approaches to cater to different use cases:

- **Configuration Files:**
  - Structured format (TOML, YAML, or a custom format)
  - Ideal for defining complex or pre-saved test scenarios.
  - Example: `test_scenario.toml`
- **Scripting Interface:**
  - Embed fault configuration within a scripting language supported by Madsim, if applicable.
  - Enables dynamic scenarios and potential integration with test harnesses.
- **Command-Line Options (Optional):**
  - Suitable for quick experimentation or overriding defaults.

#### 2. Fault Specification Syntax

- **Common Elements:**
  - **Fault Type:** Clearly identify the fault (e.g., `network_delay`, `node_crash`, `disk_io_failure`).
  - **Target:** Nodes, network links, storage entities, etc. (specificity depends on Madsim's capabilities).
  - **Intensity/Parameters:** Control behavior of the fault (delay distributions, failure probabilities, etc.).
  - **Timing:** Schedule the fault (immediate, after duration, event-based).

### 3. Example (Conceptual Configuration File)

```
[faults]

[faults.network_delay_1]
type = "network_delay"
target = { source_node = "node1", destination_node = "node3" }
delay = { distribution = "normal", mean = "200ms", std_dev = "50ms" }

[faults.node_crash]
type = "node_crash"
target = "node2"
timing = { after = "30s" }
```

### Code Example (Conceptual Scripting)

```
madsim.faults.network_delay(nodes=["node1", "node2"], delay="100ms")
madsim.faults.crash_node("database1")
```

- **Parameter Granularity:**

I'll discuss a few key fault types and consider relevant parameters. It's crucial to align the provided parameters with the effects these faults realistically have on systems.

#### 1. Network Delays

- **Distribution:** Support common distributions (Normal, Exponential, Uniform, etc.).
- **Parameters:**
  - Mean or median delay
  - Standard deviation or a range for variance

#### 2. Packet Loss

- **Overall Probability:** Percentage chance of a packet being dropped.
- **Targeted Rules:** Go beyond global probability, allowing filtering based on:
  - Source/destination node or IP address
  - Message type or protocol

#### 3. Node Crashes

- **Crash Types:** Distinguish between graceful and abrupt crashes (if the simulation supports it).
- **Restart Delay:** If simulating restarts, control the delay before a restart.

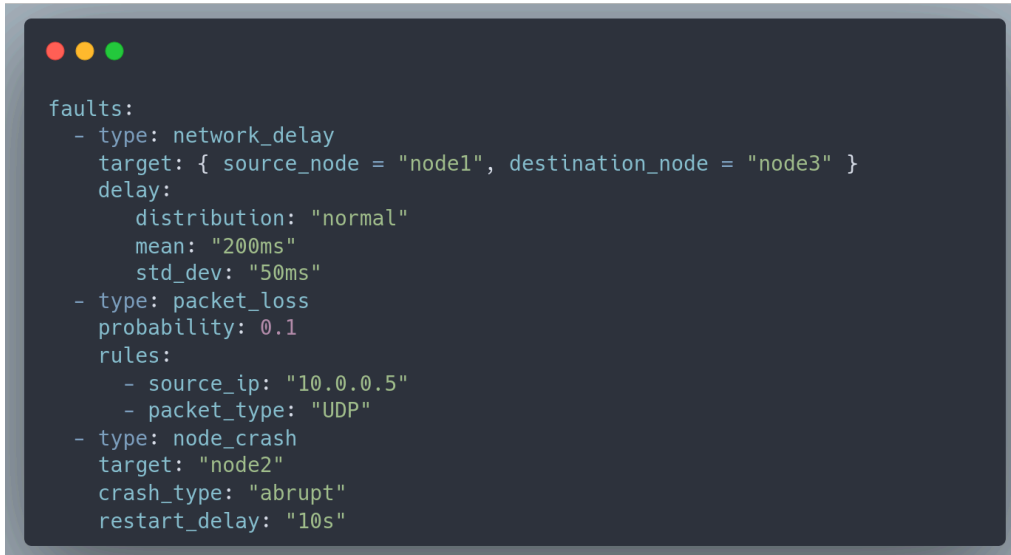
#### 4. Resource Exhaustion

- **CPU:** Target utilization percentage
- **Memory:** Amount of memory to consume
- **IO:** Limits based on throughput and/or latency increase (if applicable)

## 5. Data Corruption

- **Intensity:** Frequency or probability of corruption operations.
- **Granular Targeting:** If the storage simulation allows, target specific file ranges or key-value entries.

### Configuration Example (Conceptual)



```

faults:
  - type: network_delay
    target: { source_node = "node1", destination_node = "node3" }
    delay:
      distribution: "normal"
      mean: "200ms"
      std_dev: "50ms"
  - type: packet_loss
    probability: 0.1
    rules:
      - source_ip: "10.0.0.5"
      - packet_type: "UDP"
  - type: node_crash
    target: "node2"
    crash_type: "abrupt"
    restart_delay: "10s"
  
```

## 4. Expanded Documentation and Examples

### 4.1 User Guide

The user guide should serve as a detailed reference for developers using MadJep within Madsim. It should include the following sections:

- **MadJep for Madsim Overview:**
  - **Conceptual Foundation** A clear explanation of deterministic simulation, how MadJep achieves determinism, and its strengths and potential limitations.
  - **Key Components:** Introduce users to fault injection generators, consistency checkers, workload generators, and the test orchestration API.
- **Setting Up Chaos Experiments:**
  - **Prerequisites:** Explicitly state dependencies and system requirements.
  - **Installation:** Step-by-step installation instructions for MadJep packages within Madsim's environment.
  - **Basic Configuration:** A guide for configuring Madsim's simulation environment (network topology, node resources, etc.).
  - **Integrating the Target System** Provide instructions on how to launch an instance of the distributed system under test within Madsim. Discuss potential integration points, like system calls or network interfaces, that Madsim might need to intercept.

- **Designing Fault Injection Scenarios:**
  - **Fault Generator Syntax:** Detailed documentation on the usage and parameters of each fault generator (delays, partitions, node failures, etc.).
  - **Timing and Sequencing:** Demonstrate how to create complex fault schedules, combining multiple fault types, specifying temporal relationships, and controlling the duration and intensity of injected faults.
  - **Fault Targeting:** Explain how to target faults at specific nodes, network links, or subsystems within the simulated environment.
- **Developing Workload Generators:**
  - **MadJep Workloads API:** Introduce the interface for defining workload generators.
  - **Common Patterns:** Provide templates for basic read/write workloads, workloads with conflicts, and WAN-specific scenarios.
  - **Customization:** Guide on how to extend the framework to create highly specialized workloads tailored to a particular system.
- **Executing Chaos Tests:**
  - **Script-Based vs. Programmatic Control:** Describe both approaches for defining and running tests, highlighting the trade-offs in terms of flexibility and automation.
  - **Reproducibility:** Emphasize how to generate the seeds or deterministic inputs needed to reproduce test runs.
- **Analyzing and Interpreting Results:**
  - **Output Formats:** Specify the format of operation histories, consistency checker reports, and other data collected during test execution.
  - **Troubleshooting Tips:** Provide common errors or unexpected results that may occur, along with strategies for root cause analysis.
  - **Visualization Recommendations:** Suggest potential libraries or tools to aid in visualizing test output and understanding system behavior under chaos.

## 4.2 Example Test Suites:

- **Simple Distributed Key-Value Store:**
  - **Basic Operations:** Tests for GET, PUT, DELETE under various network partitions and node failures.
  - **Consistency Focus:** Linearizability checks for single operations and simple transactions.
- **Replicated State Machine:**
  - **Leader Election:** Tests involving node crashes and network disruptions during the leader election process.
  - **Log Replication:** Tests injecting delays and message reordering to verify correct log replication and consistency.
- **Distributed Lock Manager:**
  - **Conflicting Requests:** Scenarios where multiple nodes contend for locks, exposing potential race conditions or deadlocks.
  - **Fault Tolerance:** Tests verifying lock acquisition and release behavior under node and network failures.

## Timeline

I aim to make weekly to biweekly PRs for the project so that there is enough time for the mentors to review all the changes and suggest improvements before the next sprint cycle. This would also ensure we incrementally

develop the component continuously and do not block progress due to unstable requirements and changes for unpredictable reasons.

Period	Tasks
<b>After proposal submission</b> <b>[2 April - 26 May]</b>	<ul style="list-style-type: none"> <li>- Research additional requirements for the project</li> <li>- Convert the requirements to a technical requirement document for easier development in the later phases</li> <li>- Research other libraries and projects about how they address same concerns</li> <li>- Begin with the initial implementation of the components</li> </ul>
<b>Weeks 1-3: Foundation</b> <b>[29 May - 18 June]</b>	<ul style="list-style-type: none"> <li>- <b>Project Setup &amp; Core Mechanics:</b> <ul style="list-style-type: none"> <li>- MadJep project structure, development environment.</li> <li>- Basic Madsim integration (launching test systems, basic interception).</li> <li>- Implement core data structures for operation history tracking.</li> </ul> </li> <li>- <b>Network Faults (Delays, Partitions):</b> <ul style="list-style-type: none"> <li>- Implement delay injection mechanism (thread sleeping or queues).</li> <li>- Network topology representation for basic partitioning.</li> </ul> </li> </ul>
<b>Weeks 4-7: Expanding Fault Injection</b> <b>[19 June - 16 July]</b>	<ul style="list-style-type: none"> <li>- <b>Node Faults:</b> <ul style="list-style-type: none"> <li>- Node crash simulation (likely integration with Madsim process management).</li> <li>- Prototyping node restarts.</li> </ul> </li> <li>- <b>Resource Exhaustion (CPU, Memory):</b> <ul style="list-style-type: none"> <li>- Research Madsim's level of control over simulated resources.</li> <li>- Implement CPU exhaustion (busy loops).</li> <li>- Memory pressure (allocations, potential use of OS-level tools).</li> </ul> </li> <li>- <b>Network Faults (Refinement):</b> <ul style="list-style-type: none"> <li>- Packet loss (probabilistic, targeted).</li> <li>- Message reordering (buffering and release logic).</li> </ul> </li> </ul>
<b>Weeks 8-11: Consistency Checkers</b> <b>[17 July - 13 August]</b>	<ul style="list-style-type: none"> <li>- <b>Linearizability Checker:</b> <ul style="list-style-type: none"> <li>- Choose algorithm (Knossos-like, Adya's, or other suitable to MadJep).</li> <li>- Implement &amp; integrate with operation history.</li> <li>- Focus on single-object consistency initially.</li> </ul> </li> <li>- <b>Transactional Checker (Foundation):</b> <ul style="list-style-type: none"> <li>- DSL design for basic invariant expression.</li> <li>- State snapshot integration with Madsim.</li> </ul> </li> </ul>
<b>Weeks 12-15: Testing, Examples, Documentation</b> <b>[14 August - 10 September]</b>	<ul style="list-style-type: none"> <li>- <b>Integration Testing:</b> <ul style="list-style-type: none"> <li>- Small-scale test scenarios to validate each fault type in isolation.</li> <li>- Test cases to confirm consistency checker correctness.</li> </ul> </li> <li>- <b>User Guide (Early Version):</b> <ul style="list-style-type: none"> <li>- MadJep concepts, setup, and basic fault injection configuration.</li> <li>- Simple examples on a simulated KV store.</li> </ul> </li> <li>- <b>Expand Examples:</b> <ul style="list-style-type: none"> <li>- Basic replication, potentially leader election scenario</li> </ul> </li> </ul>

<b>Weeks 16-19: Refinement and Stretch Goals</b> <b>[11 September - 8 October]</b>	<ul style="list-style-type: none"> <li>- <b>Fault Injection Enhancements:</b> <ul style="list-style-type: none"> <li>- Rich configuration syntax, better parameterization.</li> <li>- Fault combinations with timing/triggers.</li> </ul> </li> <li>- <b>Consistency Checkers (Advanced):</b> <ul style="list-style-type: none"> <li>- Linearizability with multi-object operations.</li> <li>- Support for more complex transactional invariants.</li> </ul> </li> <li>- <b>Visualization (If Time Allows):</b> <ul style="list-style-type: none"> <li>- <b>Prototype:</b> Basic timeline visualization (D3.js) of operation history.</li> </ul> </li> </ul>
<b>Weeks 20-22: Stabilization, Documentation, Release Prep</b> <b>[9 October - 30 October]</b>	<ul style="list-style-type: none"> <li>- <b>Bug fixing &amp; Performance Tuning:</b> Address issues from earlier testing.</li> <li>- <b>Documentation Expansion:</b> <ul style="list-style-type: none"> <li>- Complete user guide and reference.</li> <li>- Example test suites (more complex)</li> </ul> </li> <li>- <b>Packaging &amp; Release Planning:</b> How MadJep will be integrated with Xline</li> </ul>
Celebrate 🎉	

## About Me

I am a **systems developer** with experience with multiple tech stacks including **back end**, **blockchain**, **distributed systems** and a particular interest in **testing and automation**. I have been fortunate enough to contribute to numerous OpenSource organizations, which have helped me learn so much and diversify my tech stack!

I am a responsible individual and a great team player who believes in the utmost importance of active communication. I have had the opportunity to work as an **SDE intern** with a wonderful team wherein I'm responsible for developing a parallelization and job scheduling algorithm to optimize performance for **MODFLOW** backend running on in-house **Supercomputer**. This piqued my interest in distributed systems particularly and **Xline** is the perfect place to learn about and contribute to this technology.

**Rust** has quickly become my go-to language for its blend of power, safety, and efficiency. I'm constantly inspired by the vibrant open-source community around Rust, and I believe contributing and learning from others is the best way to grow as a developer. In the past few months, I've actively participated in several **Rust-based projects**. **Xline** is a project I feel the most attracted towards as stated above and hence I want to contribute to the project as much as I can going forward.

Here are the links to my contributions to various organizations written in **Rust** in the **last 2 months**:

- **Xline:** [PR #682](#), [PR #676](#), [Issue #672](#)(Working currently)
- **Validating Lightning Signer:** [PR #646](#), [PR #636](#), [PR #633](#), [PR #632](#), [PR #627](#)
- **Cord:** [PR #352](#), [PR #351](#), [PR #346](#), [PR #343](#), [PR #342](#), [Issue #323](#)
- **Icu4x:** [PR #4628](#)

## Commitments

I have no other commitments for the upcoming summer and the GSOC timeline. After completing my seventh semester, I may have institute exams, but the same would be over in under ten days. I am well versed in academics and can ensure that the same would help my dedication and work schedule.



## Planning for the Development Period:

I would like to work with the mentors to decide weekly goals to align myself and work in short sprints to build the required functionality. The timeline for the project, as mentioned earlier, would help me guide the overall development.

I am a good communicator, so I want to stay in close sync with the mentors via texts, Discord, or meetings to clear all the blocking factors quickly and focus more on development. I am a night owl, so I plan to devote 150 minutes (2.5 hours) every night, seven days a week, to the project for 22 weeks, totaling more than 350 hours of work. I would happily put in extra hours if the project demands the same.

## Appendix - Optional Deliverables Approach

### Approach: Visualizing Chaos

#### 1. Data Collection

- **Instrumentation:** Enhance Madsim to emit structured data. Consider:
  - **JSON Format:** For ease of parsing and flexibility.
  - **Event Types:**
    - `operation_start, operation_end` (including operation type, node ID, timestamps, relevant metadata).
    - `metric_snapshot` (timestamp, node ID, CPU usage, memory usage, etc.)
    - `network_event` (timestamp, source node, target node, event type: delay, partition, restore)
    - `consistency_violation` (details of the violation, relevant operations)
- **Logging Mechanism:**
  - If Madsim has an event system or logging facility, integrate with that.
  - Alternatively, a dedicated log file specifically for visualization data.
- **Overhead Control:**
  - **Sampling:** Reduce the frequency of metric snapshots.
  - **Buffering:** Batch logging events to minimize I/O impact.

#### 2. Visualization Library

- **D3.js:** Primary tool due to its power and flexibility for custom visualizations.
- **Graphviz (Supplemental):** Generate initial network topology layouts, potentially with dynamic updates overlaid using JavaScript.
- **Plotting Library:** Matplotlib or Bokeh to create the synchronized metrics graphs.
- **Web-based Approach:** Develop a web interface for visualization using these libraries. Advantages:
  - Potentially tighter integration with Madsim if it has a web UI.
  - Easier for users to access.

#### 3. Visualization Components

- **Gantt-like Timeline:**
  - **D3.js Basis:** Use D3's scales and shapes to render the timeline and operation bars.
  - **Synchronization:** Align operation timestamps with a common time axis.
  - **Fault Overlays:** D3 annotations or visual markers for network disruptions and node events.

- **Consistency Checkpoints:** Consider visually indicating linearization points if applicable.
- **Metrics Graphs:**
  - **Library Choice:** Matplotlib or Bokeh, depending on desired interactivity level.
  - **Synchronization:** Share the timeline's time axis with the graphs.
  - **Visual Cues:** Potentially correlate fault events on the timeline with spikes or dips in the metric graphs.
- **Network Topology:**
  - **Graphviz Foundation:** To generate an initial network layout.
  - **D3.js Overlay:** Use D3 to add dynamic status:
    - Color-coded links (active, delayed, down).
    - Highlighting nodes under stress.
    - Animation (optional) for data flow.
- **Violation Visualization:**
  - **Conflict Graphs:** D3 or a dedicated graph library for rendering dependencies and highlighting conflicts.
  - **Invariant Visualization:**
    - Highly dependent on the nature of invariants. Might need bespoke visualizations (e.g., mini-line charts showing state variables over time within a transaction).

## 4. Integration

- **Real-time Updates:**
  - **WebSockets:** If feasible, establish a WebSocket connection between Madsim and the visualization frontend.
  - **Polling:** For less complex setups, have the visualization periodically poll Madsim or a log file.
- **Retrospective Analysis:**
  - Ensure the visualization tool can load saved log data.

## 5. Example Workflow

### Code Snippet (Conceptual - Data Structure)

```
{
  "event_type": "operation_start",
  "timestamp": 1656203300500,
  "operation": {
    "type": "read",
    "key": "customer_balance"
  },
  "node_id": "node3"
}
```

## Approach: Ensuring Reliability Through Testing

### 1. Key Principles

- **Deterministic Tests:** MadJep itself relies on determinism, so its test suite must **not** introduce additional sources of non-determinism.
- **Isolation:** Testing components individually provides granular control and helps pinpoint the origin of unexpected behaviors.
- **Ground Truth:** Having a reliable reference point to compare against is crucial for accurate assertions and reliable validation.

## 2. Test Scenarios

- **Network Faults**
  - **Delay:**
    - Set up two simulated nodes.
    - Send timestamped "ping" messages; calculate round-trip time (RTT).
    - Introduce a configured delay, verify the increase in RTT corresponds to your configuration.
  - **Packet Loss:**
    - Utilize a protocol with acknowledgements (e.g., a simplified TCP-like implementation).
    - Introduce probabilistic loss, track unacknowledged packets or retransmissions triggered by timeouts.
  - **Partitions:**
    - Divide the simulated network into groups (partitions).
    - Attempt "ping" or echo-like communication across partition boundaries. Verify that these consistently fail.
- **Node Faults**
  - **Crashes and Restarts**
    - Integrate with Madsim's process control. Issue node termination commands.
    - Either track process state directly or employ a heartbeat mechanism between the test harness and simulated nodes.
  - **Resource Exhaustion**
    - Confirm Madsim can simulate CPU/memory pressure.
    - Use system-level resource monitoring tools within Madsim's simulated environment to verify the intended effect.
- **Storage Faults**
  - **Data Corruption:**
    - If Madsim simulates storage: directly modify data at the byte level.
    - If Madsim integrates an external storage simulator: coordinate with an external tool to perform the corruption.
    - Validate that the target system has error detection or checksum mechanisms that flag the corruption.
  - **Disk Failures:**
    - Simulate disk unavailability at Madsim's storage layer (if supported).
    - Observe how the target system reacts (error codes, retries, etc.).
- **Workload Generators**
  - **Distribution Verification**
    - Run the workload generator for an extended period.
    - Log operation types and potentially relevant keys.
    - Perform statistical analysis on the logs to ensure the distribution matches your configuration.
  - **Conflict Generation**

- Design workloads with known race conditions or conflicting access patterns.
- Track the frequency of conflicts and compare it to expected rates (which might be calculated analytically).
- **Consistency Checkers**
  - **Linearizability**
    - Simple Operations: Test with non-conflicting sequences on a single object, confirm linearizability.
    - Controlled Conflicts: Devise sequences with known race conditions, the checker should reliably report violations.
  - **Transactional Safety**
    - Valid Invariants: Execute transactions where your invariant definition should always hold. The checker should not report violations.
    - Invariant Violations: Design faults and transactions which specifically trigger an invariant breach. The checker should accurately detect them.

### 3. Technical Considerations

- **Simplified Test System:** Within Madsim, set up a small-scale distributed system purely for testing purposes, offering clear, observable behavior.
- **Ground Truth**
  - **Manual Calculation:** In simpler scenarios, you might manually calculate the expected operation history or system state.
  - **"Golden" Test:** Consider having a separate, highly-trusted implementation act as an oracle to generate the ground truth for comparison.
- **Test Harness:** Rust's built-in testing framework is excellent. Define test cases, use assertions (`assert_eq!`) to compare actual vs. expected outcomes.

#### Code Snippet (Conceptual - Test Case)

```
#[test]
fn test_delay_injection() {
    madsim::setup_test_system(/* ... */);
    madsim::fault::inject_delay("nodeA", "nodeB", Duration::from_millis(500));

    let initial_rtt = measure_round_trip_time("nodeA", "nodeB");
    let rtt_with_delay = measure_round_trip_time("nodeA", "nodeB");

    assert!(rtt_with_delay - initial_rtt >= Duration::from_millis(450)); // Allow some tolerance
}
```