

Datacenter Algorithm Simulator

*Luca Fantò, Furkan Gumus,
Rai Muneeb Ullah Khan, Petar Eric*

Theme and Objectives

Distributed coordination protocols are the backbone of modern cloud services. However, evaluating these protocols on physical datacenter hardware is costly, time-consuming, and presents significant debugging challenges due to non-determinism.

The goal of this project is to develop a **Datacenter Algorithm Simulator**, a flexible and extensible software framework designed to prototype and evaluate distributed algorithms in a controlled environment. The simulator prioritizes accurate latency modeling and decoupled architecture, allowing researchers to implement protocols using simple message-passing interfaces rather than complex, hardware-specific network stacks.

As proof of concept, the framework currently supports three distinct replication protocols:

1. **Primary-Backup Replication:** A standard leader-follower model.
2. **Basic Paxos:** A consensus protocol for fault-tolerant state machine replication.
3. **Looped One-Way Imposition (LOWI):** A protocol optimized for synchronous datacenter networks.

Implementation Details

Language & tools: Python (3.13+), dependency management with [uv](#).

Simulation core: Discrete Event Simulation (**DES**) driven by a centralized min-heap scheduler. All actions (message sends, processing delays, timeouts, failures) are scheduled as events to guarantee strict chronological processing.

Determinism: The absence of real concurrency and socket-level behavior removes nondeterministic effects, allowing identical traces to be reproduced reliably when the same random seed is used.

Modular layers:

- *Config* ([src/config](#)): Handles parsing of YAML configuration file.
- *Core* ([src/core](#)): The heart of the simulator.
- *Protocols* ([src/protocols](#)): Protocol implementations inherit [Node](#); topology strategies encapsulate node wiring.
- *Services* ([src/service](#)): Orchestration elements such as `TopologyService` and `FailureService`.

Configuration: Experiments are driven by **YAML** files specifying simulation, network parameters, protocol topology and settings, workloads and failure schedules.

For further architectural details, please refer to the README in the project's root directory.

Usage Instructions

Prerequisites: Python 3.13+, [uv](#).

1. **Clone the [repository](#)** and navigate to the project root.
2. **Install dependencies** using the command: `uv sync`.
3. **Activate the environment:** `source .venv/bin/activate` (Linux/Mac) or `.venv\Scripts\activate` (Windows).

Running a Simulation To execute a simulation, the user must provide a configuration file path: `uv run main.py -c configs/target_config.yml`

Users can inspect available parameters using `uv run main.py --help`.

Output metrics are saved to CSV files as defined in the configuration, ready for post-processing analysis.

Challenges Faced

During the development of the framework, we encountered and overcame several technical challenges:

- **Abstraction vs. Flexibility:** Designing a generic [Node](#) interface that could support vastly different protocols was complex. We had to ensure the interface was simple enough for basic Primary-Backup but robust enough to handle the complex state transitions of Paxos and the timing constraints of LOWI. We solved this by using a flexible message-passing interface and the Strategy pattern for topology creation.
- **Modeling synchrony in DES:** LOWI-style synchronous interactions were simulated by enforcing tight delivery-time bounds and providing a `sync_send` primitive with a configurable probability of synchrony violation.
- **Deterministic Failure Injection:** The [FailureService](#) schedules node failures and triggers the scheduler's failure callback. It then coordinates with the [TopologyService](#), which notifies the affected node and removes it from the network, enabling precise and fully deterministic crash simulation.
- **Project Constraints:** Limited development time and coordination constraints reduced the breadth of large-scale experiments. We prioritized core functionality, correctness, and architectural cleanliness over extensive benchmarking.

Conclusions and future work

The prototype validates a DES-based approach for datacenter distributed protocol evaluation.

Next steps: richer network models (per-switch latency, queuing/congestion), integrated trace visualization, and expanded benchmarks with real workloads.