MODERN Simulator MVP Implementation and Testing Procedures

1. Introduction

This document outlines the implementation details and testing procedures for the Minimal Viable Product (MVP) of the MODERN simulator. The MVP focuses on establishing the core architecture and demonstrating basic simulation capabilities through a simple ping-pong scenario.

2. Architecture Overview

The MODERN simulator MVP is built upon a microservices architecture, leveraging gRPC for inter-service communication. The key components are:

- **Orchestrator Service:** Manages the overall simulation lifecycle, including starting and terminating simulations, and loading scenarios.
- **Kernel Service:** Implements the Discrete Event Simulation (DES) core, managing the event queue, advancing simulation time, and dispatching events to appropriate model instances.
- **Runtime Service:** Responsible for loading and executing model instances (e.g., Python models) and handling events for those models.

Communication between these services is facilitated by Protocol Buffers (protobuf) and gRPC, ensuring efficient and well-defined interfaces.

3. Implementation Details

3.1 Project Structure

The project is organized into the following main directories:

- `proto/`: Contains the Protocol Buffer definitions (`.proto` files) for all gRPC services and shared messages.
- `kernel/`: Houses the `KernelService` implementation.
- `runtime/`: Contains the `RuntimeService` implementation.
- `orchestrator/`: Contains the `OrchestratorService` implementation.

- `models/`: Stores the Python model implementations, such as `basic_node.py`.
- `scenarios/`: Contains JSON files defining simulation scenarios.
- `logs/`: Stores service logs.
- `run_simulation.py`: The main script to start and manage the simulation.

3.2 Core gRPC Interfaces

The following gRPC interfaces are defined and implemented:

- **Orchestrator <-> Kernel:**
  - `InitializeSimulation`: To set up a new simulation instance.
  - `RunSimulationStep`: To advance the simulation time.
  - `TerminateSimulation`: To end a simulation.
- **Kernel <-> Runtime:**
  - `LoadModel`: To instruct the Runtime to load a model instance.
  - `HandleEvent`: To deliver an event to a specific model instance.
  - `UnloadModel`: To instruct the Runtime to unload a model instance.

Shared message definitions (e.g., `SimulationEvent`, `LoadModelRequest`, `HandleEventResponse`) are centralized in `shared_messages.proto` to avoid duplication and ensure consistency.

3.3 Basic Simulation Kernel Service

The `KernelService` implements a Discrete Event Simulation (DES) engine. Key functionalities include:

- **Event Queue:** A min-heap (`heapq`) is used to maintain events ordered by their timestamp, ensuring events are processed chronologically.
- **Simulation Time Advancement:** The kernel advances its internal simulation time based on the timestamp of the next event in the queue.
- **Event Dispatch:** Events are dispatched to the appropriate `RuntimeService` instance based on the target model ID.

3.4 Basic Model Execution Runtime Service

The `RuntimeService` is responsible for:

- **Model Loading:** Dynamically loads Python model classes based on the `model_type` specified in the `LoadModelRequest`.
- **Event Handling:** Calls the `handle_event` method on the loaded model instances when an event is received from the Kernel.

- **Event Generation:** Collects new events generated by the models and returns them to the Kernel for scheduling.

3.5 Basic Inter-Service Communication

All inter-service communication uses gRPC. Python stubs generated from the `.proto` files (`_pb2.py` and `_pb2_grpc.py`) are used to define the service clients and servers, handling serialization, deserialization, and network communication transparently.

4. Testing Procedures

4.1 Simple Test Simulation Scenario (Ping-Pong)

The `scenario_basic.json` file defines a simple ping-pong scenario involving two `BasicNodeModel` instances (`node_1` and `node_2`).

- `node_1` is configured to send an initial `PING` event to `node_2` upon receiving a `START` signal from the Kernel.
- `node_2` is configured to respond with a `PONG` event when it receives a `PING` .
- `node_1` receives the `PONG` and the simulation completes.

4.2 Running the Simulation

The `run_simulation.py` script orchestrates the startup of all services and the execution of the simulation scenario. To run the simulation:

```
python3 run_simulation.py
```

This script will:

1. Start the `RuntimeService` , `KernelService` , and `OrchestratorService` as background processes.
2. Connect to the `OrchestratorService` .
3. Load `scenario_basic.json` .
4. Initiate the simulation, which triggers model loading and initial event scheduling.
5. Poll the simulation status until it completes or an error occurs.
6. Terminate all services.

4.3 Verifying Simulation Progression

During and after the simulation run, the following logs can be examined to verify correct behavior:

- `logs/orchestrator_service.log`: Shows the overall simulation lifecycle, including start, status polling, and termination.
- `logs/kernel_service.log`: Provides detailed insights into event scheduling, simulation time advancement, and communication with the Runtime.
- `logs/runtime_service.log`: Shows model loading, event handling by models, and events generated by models.

Successful execution will show `PING` and `PONG` events being processed, and the simulation time advancing beyond 0.0.

4.4 Unit Tests for Core Components

Unit tests are implemented for individual components to ensure their correctness in isolation. For example, `test_basic_node.py` contains tests for the `BasicNodeModel`:

```
 python3 -m unittest /home/ubuntu/modern_simulator/
test_basic_node.py
```

These tests verify:

- Model initialization.
- Correct handling of `START` events and generation of `PING` events.
- Correct handling of `PING` events and generation of `PONG` events.
- Correct handling of `PONG` events (no further events generated).
- Graceful handling of unknown event types.

5. Conclusion

The MVP of the MODERN simulator successfully demonstrates the core microservices architecture and discrete event simulation capabilities. The ping-pong scenario validates inter-service communication, model loading, event processing, and simulation time progression. The implemented unit tests ensure the reliability of individual components.