ChatGPT

Scope 4 Implementation Plan: Scheduling Engine & SBI Simulation

Overview & Goals: Building on Scopes 1–3, Scope 4 adds an **active control-plane** to the simulator. We will implement Aalyria Spacetime's *Southbound Interface* (SBI) with a basic scheduling engine so that the simulator can send time-tagged control commands (beam on/off, routing updates, etc.) to each node's agent and process telemetry returns. In practice, the simulator's **controller** module will schedule configuration changes and stream them to per-node "agent" components, which apply the commands at the right simulation times [1] [2]. The result is a closed-loop control plane: links can be *activated/deactivated* on schedule and routes can be added or removed, and agents report status back via telemetry [1] [3]. This essentially "brings the network to life": satellites will turn beams on/off on schedule and routes will appear/disappear as planned [1] [4]. Throughout, all APIs and messages will use Aalyria's open protobuf schemas to ensure full Spacetime compatibility [5].

Covered Requirements: Scope 4 fully implements the **SBI Scheduling and Telemetry** services from the requirements. In particular, we will provide the bidirectional scheduling gRPC stream (CDPI) and the Telemetry RPC so that our simulated agents can receive scheduling commands and report metrics [6] [7]. This scope **completes** the requirements for the Southbound API: it implements `ReceiveRequests` streaming RPCs (Hello, CreateEntryRequest, DeleteEntryRequest, etc.) and `ExportMetrics` calls for telemetry [8] [7]. We will also begin basic **network orchestration logic**: e.g. automatically scheduling beam activations for links in view. (More advanced planning/optimization is deferred to Scope 5.) In sum, Scope 4 meets all Scheduling/Telemetry requirements and lays the groundwork for satisfying service request routing in later phases [9] [10].

Included Implementation:

- **Agent Model:** For each `NetworkNode` (or platform hosting a node), instantiate a *Simulated Agent* as an internal component (e.g. a goroutine or thread) [11] [12]. Each agent maintains a local *schedule queue* of future actions (beam activations, deactivations, route installs, etc.). It watches the simulation clock (via the central time controller) and when a scheduled time arrives, it executes the action by updating the node's state in the KnowledgeBase. For example, an `UpdateBeam` command causes the agent to mark a particular interface's beam as active (establishing the corresponding link subject to geometry); a `DeleteBeam` turns it off [13] [14]. Executing a beam or routing command typically updates the connectivity graph: e.g. turning on a beam makes a previously potential link become *active* [14]. Agents also generate telemetry: they periodically emit metrics (interface status, counters, link quality, etc.) back to the controller (see below).

- **Scheduling Service (CDPI Stream):** Implement the Spacetime Scheduling gRPC service (`ControlDataPlaneInterface`). When simulation starts, each agent **connects** to the controller by opening the CDPI stream and sending a Hello with its agent ID (e.g. node or platform ID) [15]. The controller uses this ID to associate the stream with the correct internal node model. Then the controller (scheduler) sends `CreateEntryRequest` messages on the stream for that agent. Each request has a unique request ID, a scheduled timestamp, and a **oneof** config (e.g. `UpdateBeam`, `DeleteBeam`, `SetRoute`, `DeleteRoute`) specifying the action [16] [17]. The agent receives each command, inserts it into its local schedule, and when the simulation clock reaches the specified time, executes it. Upon execution, the agent sends back a `Response` on the stream to acknowledge success or failure of that request ID [17].

- **UpdateBeam/DeleteBeam:** An `UpdateBeam` message carries a `Beam` configuration (interface ID, target interface or coordinates, frequency, power, etc.) and a start time [13]. The controller uses this to command a node: "At time T, point interface X at target Y" [13]. The agent schedules this event; at time T, it **activates** the beam in the KB (e.g. marks interface X as pointing to Y), which in effect brings the link up in the connectivity graph [3]. A `DeleteBeam` similarly schedules turning the beam off at a given time. We will handle these gRPC message types by translating them into internal actions on the node state. (We simulate near-instant execution when the time comes; network latency is ignored.)

- **SetRoute/DeleteRoute (Static Routing):** We add support for routing commands as well. A `SetRoute` entry will include a `Route` protobuf (source, destination, next-hop, etc.) and a time. At the scheduled time, the agent adds this entry to the node's routing table in the KB [3]. Conversely, `DeleteRoute` removes it. We maintain a simple per-node route table (mapping destination to next-hop) in the KnowledgeBase. Initially, routing logic is static or one-hop: for example, when a link is up, we may install a route over it; more sophisticated routing algorithms are left for future scopes.

- **Housekeeping Messages:** Implement support for `DeleteEntryRequest`, `FinalizeRequest`, and `Reset` as defined by Spacetime's API [18]. For example, if an agent sends a `Reset` (simulating a reboot), the controller should respond by resending the current desired schedule (using the schedule manipulation token). `FinalizeRequest` can tell an agent to drop tasks before a cutoff time. Handling these ensures robustness (e.g. cleanup at scenario end).

- **Scheduling Logic (Controller/Orchestrator):** The scheduler module decides *what* to schedule. In Scope 4, this logic will be **simple**, meant to demonstrate end-to-end flow [10]. For example, the controller may automatically schedule `UpdateBeam` events whenever a satellite comes into view of a ground station, and corresponding `DeleteBeam` when it goes out of view. As one approach, the controller can subscribe to connectivity changes (from Scope 2) and, upon detecting a new potential link, create an entry to turn that beam on after a small lead time; when the link disappears, schedule it off [10]. We may also schedule a default route when a link is active. Crucially, *Scope 4 focuses on implementing the mechanics* of scheduling (controller→agent→execution→telemetry) [10]; the actual optimization of "which beams to light up" is minimal here and will be elaborated in Scope 5.

- **Telemetry Service:** Implement the Spacetime Telemetry gRPC service so agents can report status and metrics. Each agent will periodically call `ExportMetrics` with an `ExportMetricsRequest` containing interface metrics and (optionally) modem metrics [19] [20]. For this scope, we will populate basic fields: interface operational status (up/down) and traffic counters (bytes/packets sent/received) [21]. (Since we are not simulating actual traffic flows yet, counters may remain zero or reflect dummy traffic.) If feasible, we can include link quality or SNR in modem metrics. The controller's TelemetryService simply accepts these (returning empty ack) and may log or store the latest values. Providing this API closes the SBI loop and offers real-time observability of the simulated network [19] [20].

- **Data Models & Interfaces:** We will extend the data model to include scheduled entries and route tables. The KnowledgeBase (from Scope 1) must now store per-node schedules and routing state. Agent actions update the KB (e.g. an active beam may create a `NetworkLink` record between interfaces if not already present, or set a "link up" flag) [14]. All new APIs use Aalyria's protobuf definitions: for example, the `Beam`, `Route`, and telemetry messages come straight from `network_link.proto`, `service_request.proto`, and `telemetry.proto` (or compatible structs) [5]. Our implementation will expose gRPC endpoints exactly matching Spacetime's API (using their `.proto` files), ensuring any Spacetime client (or agent) could interoperate with our simulator.

Integration with Prior Scopes: Scope 4 is layered on Scopes 1–3. The **KnowledgeBase** from Scope 1–3 remains the central state store: it now also holds agent schedules and routes. Agents will update KB entries (positions and link states computed earlier by Scope 2's connectivity module) when executing commands [14]. The **Connectivity Service** (Scope 2) provides geometric LOS information that informs scheduling (e.g. to know when to schedule an `UpdateBeam`). The **NBI/Scenario loader** from Scope 3 still applies: users can define service requests (traffic demands) and network elements via gRPC or textproto, and the controller can use these to decide scheduling. For example, a `ServiceRequest` might trigger scheduling the corresponding link and route (though a full demand-driven scheduler is left for later). All existing APIs from Scopes 1–3 remain available (e.g. CreatePlatform, CreateServiceRequest, etc.), and new scheduling APIs are added alongside. Importantly, the SBI implementation coexists with the NBI layer: for instance, a CreatePlatform call (Scope 3) starts a satellite's orbit, and now that same node has an agent listening on the SBI. Thus, Scope 4 tightly integrates with prior architecture: the KnowledgeBase links nodes↔platforms and tracks interfaces (from Scopes 1–2) [22], the time-controller advances the clock (Scope 1), and the existing connectivity graph (Scope 2) is modified by agent actions. All modules remain loosely coupled: the scheduler module writes to KB, and the agent (in the same process) reads from KB and executes commands.

Edge Cases, Error Handling & Observability: We will implement robust handling of corner cases in the SBI protocol. For example, if an agent receives a `CreateEntryRequest` with a duplicate entry ID or an unknown interface ID, it should respond with an error status (gRPC error codes) and ignore the command. We will maintain the schedule manipulation token as Spacetime requires: on each `Reset`, the agent generates a new token and rejects stale requests [6]. If an agent resets or disconnects, the controller will resend pending schedules with the new token. The `Response` acknowledgments (with status) let the controller detect and log any failures. We will also handle time discrepancies gracefully: if simulation time has already passed a command's timestamp when it's received, we can choose to execute it immediately. In all cases, the system will log events and errors using structured logging (timestamps, node/agent IDs, request IDs) for traceability [23].

Observability is built-in: we will instrument key events and metrics. For instance, the scheduler will emit logs or Prometheus metrics for each `CreateEntry` sent, and agents will log when they execute or fail a task. The telemetry data (up/down counters, SNR, etc.) provides a continuous stream of network-state information. We may also add distributed tracing (e.g. OpenTelemetry) across the controller↔agent calls [23]. This level of instrumentation allows monitoring the simulator's behavior and diagnosing issues (e.g. if a link never comes up, we can inspect the logs and telemetry to see why).

Testing Strategy: A thorough test suite will validate Scope 4 features.

- *Unit Tests:* We will write tests for each new module. For the scheduling service, a test can simulate an agent stream: feed a `CreateEntryRequest` with an `UpdateBeam` and verify that the agent's schedule queue receives it correctly, and that at the scheduled time the KnowledgeBase reflects the beam as active. Similarly, tests for handling `DeleteEntryRequest`, `SetRoute`, `Reset`, etc., will verify the proper KB updates or errors. The TelemetryService will have unit tests where a fake agent calls `ExportMetrics` and we assert that the metrics are recorded or logged as expected. The agent component itself will have tests for its scheduling queue logic (e.g. adding events out of order, resetting the schedule, executing overdue events, etc.).

- *Integration Tests:* We will write end-to-end tests using the actual gRPC APIs (leveraging Spacetime's protos). For example, a test could:

- Use the NBI to create a simple scenario: one satellite with one steerable interface and one ground station, both with interfaces configured.
- Run the simulation until a known contact window (based on TLE geometry) and check that the connectivity service indicates a link is possible.
- Then simulate the scheduling: automatically or via a test controller module, issue an `UpdateBeam` for that link at time T. Advance simulation to T and assert that the KB shows the link as **active** and that `ExportMetrics` from the agent reports the interface as "up".
- Schedule a `DeleteBeam` and verify the link goes down.
- Test a `SetRoute` command by defining a flow and scheduling a route; verify packets (or dummy counters) would use that route. We may encode scenarios in textproto (as in Scope 3) and load them in tests. These integration tests confirm the scheduling pipeline (controller→agent→effect) works. Additionally, we can simulate fault conditions: e.g. send an invalid command and assert an error response, or simulate a time jump and check the agent's behavior.

- *Validation:* We will compare simulated behavior to expected timelines. For instance, we know from orbital dynamics (Scope 1) when a satellite should see a ground station. We can validate that our scheduler turns the beam on during that period. Unit tests of geometric checks (from Scope 2) will also be reused here to validate that agents only activate beams when the link is in range.

Assumptions & Simplifications: As with earlier scopes, we make deliberate simplifications. We assume **ideal agents**: they execute commands instantaneously at the scheduled time with no processing delay. Control-plane messages have no network latency since controller and agents run in one process. Scheduling logic is *naïve*: for example, we might by default activate any feasible link rather than computing optimal link assignments [10]. Traffic flows are still not simulated; interface usage in telemetry will be stubbed (typically zero) unless we later add dummy traffic. Some advanced Spacetime features (segment routing policies, DTN store-and-forward) remain out of scope here. We also assume synchronous simulation time: even if the simulation runs faster-than-real-time, we will treat event times purely as simulation timestamps (no real-time clock needed).

Throughout, we preserve Aalyria Spacetime API compatibility: all new gRPC endpoints and message formats (scheduling and telemetry) exactly match the published Spacetime protos [5]. This ensures any existing Spacetime agent or controller could interact with our simulator without changes.

Summary of Scope 4 Deliverables:

- **SBI Scheduling Service:** gRPC bidirectional stream for the controller to send `CreateEntryRequest`/`DeleteEntryRequest` (with `UpdateBeam`/`SetRoute`, etc.) and receive acks [16] [17].
- **Embedded Agent Component:** A simulated agent per node (goroutine/thread) with a local schedule queue. Agents apply commands at the right time and update KB state (beam/link status, routing table) [14] [11].

- **Telemetry Service:** gRPC `ExportMetrics` API that agents use to report interface/link status and counters [19] [20].
- **Basic Scheduling Logic:** A simple orchestrator that schedules beam on/off (and perhaps routes) based on geometry, demonstrating dynamic network behavior [10].
- **KnowledgeBase Integration:** Extensions to the data model for scheduled events and routes; updates to connectivity graph and routes as commanded.
- **Testing Suite:** Unit and integration tests covering the scheduling stream, agent logic, telemetry, and end-to-end scenario workflows.
- **Observability Hooks:** Structured logging and metrics instrumentation for controller and agent actions, as well as telemetry data, enabling monitoring of the simulator [23].

All of the above builds on the existing Core, Connectivity and NBI modules (Scopes 1–3) and ensures a full control-plane simulation in line with Aalyria Spacetime's design [1] [5].

Sources: Roadmap and architecture documents for Scope 4 planning [1] [2] [10] [5], and Aalyria Spacetime requirements for scheduling/telemetry [6] [19]. These sources outline the SBI protocol and agent model we will implement.

---

[1] [4] [8] [9] [11] [13] [15] [16] Roadmap for Spacetime-Compatible Constellation Simulator Development.pdf
file://file_0000000062f47207a2cfa34524822a5a

[2] [3] [5] [10] [12] [14] [17] [18] [20] [23] Architecture for a Spacetime-Compatible Constellation Simulator.pdf
file://file_00000000af60720785cdb52589a7113a

[6] [7] [19] [21] Requirements for an Aalyria Spacetime-Compatible Constellation Simulator.pdf
file://file_00000000ee547207b2aba490d52d866d

[22] Scope 3 Implementation Plan_ Northbound API & Scenario Configuration.pdf
file://file_00000000058c7207b3369eed90b50691