ChatGPT

Roadmap for Spacetime-Compatible Constellation Simulator Development

Scope 1: Core Entities & Orbital Dynamics (Foundational Phase)

- **Goals & Deliverables:** Establish the fundamental data models for the constellation. Implement `PlatformDefinition` (physical platform) and `NetworkNode` (logical node) entities, along with basic motion modeling. Deliver a backend that can ingest satellite orbital data (e.g. TLE strings) and propagate positions over time using an SGP4 library [1]. The simulation should maintain an internal "knowledge base" of all platforms and nodes, with their IDs, names/types, and coordinates updating as time progresses.
- **Covers Requirements:** Fully covers *Node and Platform Modeling* (all core points from that section). Partially covers related data fields from later sections (e.g. basic node fields like type tags and placeholders for power/storage).
- **Included Implementation:**
- Define data classes or database schema for PlatformDefinition and NetworkNode (including fields for IDs, names, type tags, etc.).
- Link NetworkNodes to Platforms when applicable (to inherit position) [2].
- Integrate an orbit propagation tool (such as Orekit or a Python SGP4 library) to update satellite positions periodically from TLE inputs [1].
- Support static or scripted motion for non-orbital platforms (e.g. fixed ground stations or simple trajectories for aircraft), using a uniform Earth-Centered, Earth-Fixed coordinate frame.
- Set up a simple time loop or scheduler that periodically updates each platform's position/orientation according to its motion model.
- **Assumptions & Simplifications:** In this initial scope, many advanced parameters are stubbed out or defaulted:
- **Routing config:** Store fields like router ID if provided, but no routing logic yet (just placeholders) [3].
- **SDN Agent latency:** Assume negligible control-plane latency (we simply note the field exists) [4].
- **Power budget:** Accept a node's `power_budget` definition but do not enforce it yet (just log or ignore if exceeded) [5].
- **Storage/DTN:** Include a `storage_capacity` field per node and track it, but detailed DTN logic will come later [6] [7]. For now, just note which nodes are DTN-capable.
- **Platform payloads:** A satellite's payload (e.g. bent-pipe transponder) can be defined in data, but we won't model its behavior yet (no channelization or processing limits in this scope).
- No network connectivity is computed in this phase – we treat all nodes as isolated entities with positions.
- **Dependencies:** None (this is the first phase). It establishes the base classes and time-step mechanism that all later scopes will build upon. This foundation is kept **modular** so others could contribute (e.g. adding new motion models) without altering core logic.

Scope 2: Network Interfaces & Connectivity Evaluation

- **Goals & Deliverables:** Build on the core entities by introducing `NetworkInterface` objects for each node and implementing the **wireless and wired link modeling basics**. The deliverable is a simulator that can represent each node's communication interfaces (antennas, radios, ports) and determine which links are *geometrically* possible at a given time. This scope results in a rudimentary connectivity graph that updates with platform motion.

- **Covers Requirements:** Fully covers the *Network Interfaces and Transceivers* section and partially covers *Link, Beam, and Connectivity Modeling*. It implements interface definitions (wired vs wireless, transceiver models, etc.) and computes line-of-sight link availability, laying the groundwork for dynamic beam scheduling later [8] [9].
- **Included Implementation:**
- Extend the data model to include `NetworkInterface` entries under each NetworkNode [10]. Each interface has a unique local ID (and global composite ID) and basic metadata (name, IP/MAC, etc.) [11].
- Implement support for **wired interfaces**: e.g. fiber or Ethernet links. For these, store a max data rate and (optionally) an associated platform (if modeling a specific physical cable) [12]. Assume wired links are *always up* unless manually impaired, with either a fixed latency or a configurable latency per link [13].
- Implement support for **wireless interfaces**: each with a reference to a `TransceiverModel` that defines its radio characteristics [14] [15]. Define a schema for transceiver models (frequency band, antenna gain pattern or simple beam width, transmit power/EIRP, receiver sensitivity, supported modulation modes, polarization, etc.). Provide a couple of example models (e.g. a generic LEO satellite antenna, a ground station dish) for use in scenarios. Initially use simplified models (e.g. isotropic antenna or a fixed gain) and allow extension later.
- **Line-of-Sight Calculation:** Implement geometry checks to determine if two wireless interfaces *could* communicate at a given time. This includes:
    - For satellite-to-ground: check if the satellite is above the ground station's horizon (elevation > minimum angle) [8]. Use Earth's radius and platform altitudes for this computation.
    - For satellite-to-satellite: check if the direct line between satellites is unoccluded by Earth (no Earth intersection) for inter-satellite links.
    - Optionally include a maximum range if appropriate for the frequency (though for RF LEO/GEO, line-of-sight will be the main limit). We assume full hemispherical coverage for antennas unless specified otherwise [16] (i.e. no gimbal limits yet).
- **Basic Link Budget:** Implement a rudimentary link budget evaluation to classify link quality when line-of-sight exists. For example, use Friis transmission formula to estimate receive power given distance and frequencies [17]. Compare against a threshold to decide if the link is *accessible* or marginal [18]. This can yield a simple capacity estimate or an accessibility flag (e.g., link "up" vs "down"). Initially, assume clear weather and no interference – effectively if geometry allows, the link is considered usable at some nominal bitrate.
- **Connectivity Graph:** Develop a service or function that periodically evaluates all potential wireless links between interfaces and updates their status (up/down and maybe an estimated throughput) [19]. This is akin to a **Link Evaluator** service that produces an adjacency matrix of the network in real-time. Initially run this on a fixed interval (e.g. each simulation second or minute) or upon significant movement.
- **Operational Impairments:** Implement the ability to mark interfaces (or entire links) as impaired/down via configuration or API (simulating failures). For example, an interface could be set to *DEFAULT_UNUSABLE* state with a timestamp [20]. The connectivity calculator should honor this by treating that interface as unable to form links during the impairment period.
- If wired links are configured (e.g. a terrestrial fiber between two ground nodes), include those as always-available edges in the connectivity graph (unless manually taken down).
- **Assumptions & Simplifications:**
- **Interface relationships:** We assume that any two wireless interfaces with compatible transceiver models (e.g. same band) can potentially form a link if in view. To keep it simple, we

do not require a pre-defined NetworkLink entity at this stage for two nodes to connect – all geometry-allowed pairs are considered. (Later we will allow scenario definitions to restrict or schedule these links.)

- **Single beam per interface:** At this stage, assume each wireless interface can only support one link at a time (one steerable beam) by default [21] [22]. We will treat multi-beam capabilities as an advanced feature to be added later if needed.
- **Transceiver details:** We will not fully implement complex antenna gain patterns or adaptive modulation yet. If no pattern is provided, treat antennas as isotropic (or a fixed gain). Modulation and coding schemes are kept static per link (no dynamic rate adaptation), though we will store the mode (e.g. DVB-S2X, etc.) for completeness [23] [24]. Throughput can be a fixed value or a simple function of SNR.
- **No interference modeling:** We assume each active wireless link uses an orthogonal frequency or scheduling, so no co-channel interference is considered [25] [26]. This will remain a simplification in early versions.
- **Stub NMTS fields:** Any low-level hardware chain identifiers (NMTS IDs) in the interface definitions are accepted but not used in simulation [27]. They'll be stored for API compatibility but filled with dummy values or left empty.
- **Dependencies:** Requires Scope 1 completion (platform positions must be updating correctly). This phase builds on those classes to add networking facets. The module structure here (e.g. a `ConnectivityService` for link evaluation) will be designed so that others can later contribute improvements (like more accurate RF models or integration of external libraries for link budgets) without changing core simulation loop.

Scope 3: Northbound API & Scenario Configuration

- **Goals & Deliverables:** Expose a Northbound Interface (NBI) that mirrors Aalyria Spacetime's API, allowing users (or higher-level scripts) to define and query the simulation scenario through gRPC calls or equivalent. The deliverable is a set of API endpoints to create, update, list, and delete all major entity types (platforms, nodes, interfaces, links, service requests, etc.), using the same protobuf schemas (or compatible ones) as Spacetime [28] [29]. This enables loading a scenario (network topology and traffic demands) without directly manipulating internal data structures.
- **Covers Requirements:** Fully covers *Northbound API: Entities and Operations*. It ensures that all entity types mentioned in the requirements can be managed via API calls. This scope also touches on data model aspects from earlier scopes (ensuring they align with the API definitions).
- **Included Implementation:**
- Define gRPC services (or a similar RPC mechanism) for each major NBI operation:
    - **Create/Get/List/Delete PlatformDefinition:** Accept definitions of platforms with initial position and motion source (e.g. TLE, static). On create, the simulator instantiates the platform and begins tracking its motion. (If motion source is TLE, link to an internal propagator; if manual/static, set coordinates directly.) Child components like payloads or antennas can be part of the request, but complex payload modeling is stubbed (we acknowledge the fields without deep logic) [30] [31].
    - **Create/Get/List/Delete NetworkNode:** Allow creation of a network node with its properties (node ID, name/type, etc.) and optionally an embedded list of interfaces [29] [32]. We'll support providing the interfaces in-line for simplicity (mirroring older Spacetime usage). If interfaces are not included on creation, provide separate calls to add interfaces to a node. The create call will link the node to an existing Platform if one is specified (or create a standalone node without a platform if none given).
    - **Create NetworkInterface:** (If not done via CreateNode) allow adding an interface to a node by specifying interface details (ID, type wired/wireless, transceiver model reference, etc.). In Spacetime NBI, interfaces might not be standalone entities in the older API, but

we expose this for completeness. The interface data model follows what was implemented in Scope 2.
    - **Create/Get/Delete NetworkLink:** Provide a way to define static links between two interfaces (primarily for wired or fixed links). While Spacetime's NBI doesn't treat NetworkLink as a first-class entity in all cases, our simulator will allow users to declare a link resource for clarity [33] [34]. This is useful for setting up known persistent links (like a constant ground fiber or an always-on inter-satellite laser link if desired). The create call would specify the two end interface IDs (or two unidirectional pairs for a bidirectional link). These static links will be inserted into the connectivity graph as always available (unless impaired).
    - **Create/Get/List/Delete ServiceRequest:** Allow users to specify a demand for connectivity between a source and destination node [35] [36]. The ServiceRequest includes the key fields: source node, destination node (or region, though we'll limit initial support to explicit nodes), desired bandwidth (and minimum bandwidth), latency requirements, priority, disruption-tolerant flag, and optionally time validity (if the request is only for a certain period). The API call will register this demand in the simulator's state. (It does not immediately allocate a route – that will be handled by the planning logic in the next scope.) We also implement read/list operations so that a client can query whether a service request is currently provisioned and see its status (fulfilled or not) [37].
    - **List/Get Intents (optional/debug):** Intents (LinkIntents, PathIntents, etc.) are internal artifacts created when planning how to satisfy ServiceRequests. We may maintain an internal list of Intents for each ServiceRequest to track what actions have been scheduled (beams, routes). For API completeness, we can expose a read-only view of these Intents (without requiring users to create them directly) [38] [39]. This helps mimic Spacetime's behavior for debugging, but creating Intents via NBI is not needed in our usage.
- **Data Persistence:** Implement a simple in-memory store (or database layer) for all these entities such that they can be referenced by ID. Ensure cross-references (e.g. node to platform, interfaces to node, links to interfaces, service request to node IDs) are handled and validated.
- **Query Operations:** Support Get/List for each entity type so that after creating a scenario, the user (or test scripts) can retrieve the current state (e.g. list all Platforms or all ServiceRequests and see their fields). This is important for verifying that the simulator accepted the inputs correctly.
- **API Compatibility:** Use Aalyria's published protobuf definitions whenever possible to structure our messages (e.g. use the same field names/types as in `network_element.proto`, `service_request.proto`, etc.). This will make the simulator **Spacetime-compatible** – potentially allowing existing Spacetime client code or config files (textproto scenarios) to be fed in with minimal changes [40]. If the Spacetime NBI is evolving (e.g. toward an NMTS graph model), we stick to the stable subset where a node with embedded interfaces is created, to reduce complexity.
- **Assumptions & Simplifications:**
- **Bent-Pipe Payload:** We include fields for satellite payload configuration (from `bent_pipe.proto`) in the PlatformDefinition API for completeness, but initially do not enforce any payload channel limits [41] [42]. For now, assume every satellite simply relays traffic transparently with no processing constraints. A field like `max_processed_bandwidth` can be stored and perhaps logged if exceeded, but not actively used in routing decisions in this phase.
- **Region-based requests:** If ServiceRequests specify a region (like "any satellite covering region X to ground station Y"), we will defer implementing complex region matching. The initial API will accept region fields but we focus on explicit source/destination node IDs for fulfilling flows [43].
- **No federation logic yet:** The NBI might have a flag `allow_partner_resources` in ServiceRequest for federation [44]. We will accept and store this flag, but it has no effect in this

scope (we log or ignore it, since federation will be handled in a later scope or left unimplemented initially).

- **Model vs. NBI API:** We continue to use the "entity-centric" API calls as described. We note that Spacetime has a newer model API (graph-based NMTS model), but we do **not** implement that now [45]. Our approach is sufficient to define scenarios. Adapting to the full NMTS model API is a future enhancement (the design will keep this possibility open).
- **Dependencies:** Requires Scopes 1 and 2. The internal model must be in place so that the API calls manipulate actual simulation objects. By the end of Scope 3, a solo developer can fully **configure a scenario via API** (or textproto), but the network is not yet actively doing anything beyond updating positions and recognizing which links *could* exist. This sets the stage for adding the actual scheduling and traffic handling.

Scope 4: Scheduling Engine & Southbound Interface Simulation

- **Goals & Deliverables:** Implement the **Southbound Interface (SBI)** message flow and a basic scheduling engine that uses it to control the network. In this scope, the simulator will generate time-tagged commands (beam on/off, routing updates) and simulate their execution on the nodes via an embedded agent. The deliverable is a functioning control-plane loop: the controller side of our simulator can schedule configuration changes, send them to each node's "agent" (simulated internally), and the agent applies them at the correct simulation times. Telemetry reporting from agents back to the controller is also implemented, closing the loop. This scope essentially makes the network *active*: satellites can turn beams on/off on schedule and routes can appear/disappear in response to the plan.
- **Covers Requirements:** Fully covers *Southbound API: Scheduling and Telemetry Services*. It also begins to cover *Network Planning and Orchestration Logic* (in that we must decide some actions to schedule, at least in a simple way). Essentially, this phase implements the **mechanism** of scheduling and telemetry, though the **policy/optimization** (complex orchestration decisions) will be fleshed out in Scope 5.
- **Included Implementation:**
- **Embedded Agent Model:** For each NetworkNode (or for each platform that hosts a node), instantiate a simulated "agent" component. This could be a thread or coroutine within the simulator that represents the device's control-plane agent. The agent will maintain a schedule of pending actions (beam activations, route changes) and execute them at the specified simulation times. It will also generate telemetry reports periodically.
- **Scheduling gRPC Stream:** Implement the Scheduling Service as defined by Spacetime (CDPI stream) [46] [47]. In our case, since both controller and agents are within the same process, we can simulate the bidirectional stream without network sockets:
    - The agent (client side) "connects" on startup by sending a Hello with its agent ID [47]. The controller recognizes the agent ID and links it to the corresponding node in our model.
    - The controller then sends down a series of `CreateEntryRequest` messages to that agent's stream for any scheduled tasks [48] [49]. Each request includes a unique request ID, a scheduled timestamp, and a oneof config (UpdateBeam, DeleteBeam, SetRoute, DeleteRoute, etc.) representing the action to perform [50].
    - Implement handling for each type of config:
    - **UpdateBeam:** contains a Beam configuration (which interface to activate, target interface or coordinates, frequency/power settings) and a start time [51] [52]. The controller uses this to command a node: "At time T, point interface X at target Y with given parameters." The agent will insert this into its local schedule. When simulation time reaches T, the agent marks that beam as active (meaning in our simulation, a link between those two interfaces is now considered established, subject to geometry). We simulate immediate or

timely execution since we control the sim clock (if T is in the past relative to current sim time, execute immediately).
  - **DeleteBeam:** instructs the agent to turn off a previously set beam at time T. The agent will schedule removal of the beam (freeing that interface) at that time [53] [54].
  - **SetRoute:** carries a route entry to install in the node's routing table at time T [55] [56]. The route could include a destination prefix and a next-hop or output interface (similar to a static route). The agent will apply this (we maintain a simple routing table per node). Installing the route effectively enables forwarding along a path when links are up.
  - **DeleteRoute:** removes a previously installed route at time T [57] [58].
  - **(Optional) SetSrPolicy/DeleteSrPolicy:** accept these messages if they arrive, but initially just log or store them without effect [59]. Segment routing policies (pre-defined path constraints) will be parsed and kept in the node's config, but not actively used yet (we will mostly stub this in early versions).
  - **FinalizeRequest:** agent should drop any scheduled entries before a given cutoff time [60] [61]. Implement this for completeness – the controller might send it to indicate it won't change past schedules. Our agent can then clear old executed tasks.
  - **Reset:** Have the agent send a Reset RPC on (re)startup [62] [63]. The controller handles it by clearing any pending tasks for that agent and possibly sending a fresh set of CreateEntryRequests for the current plan. In our sim, we'll simulate that agents connect at sim start, send Reset, and get their initial schedule.
  - The agent must send responses for each request ID it executes, simulating acknowledgments (Status OK or error) [64]. We will default to success (unless we simulate a failure like a beam that couldn't be established due to a missed timing – which we might consider in advanced cases). These responses don't alter simulation state except to update the status of Intents if we're tracking them.
- **Telemetry Reporting:** Implement the Telemetry Service RPC (ExportMetrics). Each agent will periodically (e.g. every N seconds of sim time) send an `ExportMetrics` message with telemetry data [65] [66]:
  - **InterfaceMetrics:** For each interface on the node, report its operational status (up/down) and basic counters [67]. We can derive "up/down" from whether the interface currently has an active link or if it's impaired. We also maintain byte/packet counters: if we have flows assigned, increment bytes based on the allocated bandwidth * time the link was active (since we're not simulating actual packets, this is an approximation) [68] [69]. If no traffic is flowing, these remain zero or constant. We will simulate at least one counter (bytes transmitted) to demonstrate that the telemetry reflects usage.
  - **ModemMetrics:** If available from our link budget calculation, include things like current SNR or modulation mode for the active link on each interface [70]. Initially, this could be stubbed or simply echo the configured mode and a notional SNR (e.g. from last link evaluation). It's optional and can be expanded in later scopes.
  - The Telemetry RPC is agent->controller; our controller will receive these and simply log or store the metrics. (In the future, we might let the user query them via NBI or subscribe to updates, but that's extra – for now, storing last known metrics per interface and node is sufficient).
- **Time Management:** Decide on a simulation time progression approach. To test SBI scheduling, we likely run the simulation in a **discrete-event** mode (so we can fast-forward through events rather than real wall-clock). Implement a simulation clock that can advance in steps or jump to next event time. The controller will schedule events (via CreateEntryRequests) with timestamps in this sim time (likely as absolute times, e.g. UNIX epoch or GPS time) [71] [72]. We might choose an arbitrary epoch (say start = 0 or a fixed date) for convenience, as long as it's consistent. The key is that the agent and controller both interpret times the same way. This mechanism allows us to run simulations faster than real time. (If we wanted real-time operation, we could also

implement that mode by having the sim clock sync with system clock, but initially, planning mode is fine.) The scheduling messages will include timestamps in epoch format for API compatibility, even if our sim uses relative time under the hood [71] [72].

- **Basic Scheduling Logic:** In this scope, we implement a **naive scheduling strategy** to drive the SBI mechanism. For example:
    - If a ServiceRequest exists for a particular source-destination, and our connectivity graph (from Scope 2) shows a link or path is currently available, schedule an UpdateBeam (or multiple) to activate those links immediately and a SetRoute to direct traffic. In simplest form, if a satellite-ground link is possible now, just turn it on continuously.
    - If a link is no longer geometrically available (satellite moved out of range), schedule a DeleteBeam for when it ends. This could be done by pre-computing contact windows for that link and scheduling off at window end.
    - Essentially, implement a trivial "keep link up whenever possible" policy. This will not optimize for multiple flows or do handovers elegantly, but it will exercise the SBI.
- **Example:** For instance, if Sat1 can see GS1 from 12:00 to 12:10 sim time, the controller will send Sat1's agent an UpdateBeam for 12:00 (point to GS1) and a DeleteBeam for 12:10 [73] [74]. If a route from a user terminal (UT1) via Sat1 to GS1 is needed, it sends SetRoute to UT1 (or Sat1) at 12:00 to route UT1->GS1 via Sat1, and likely another SetRoute to Sat1 or GS1 accordingly. All these tasks flow through the SBI stream. The agents execute them at the specified times, thereby turning the link on and installing a route in the sim's routing tables. Telemetry from Sat1's interface would start counting bytes during 12:00–12:10 if a flow is active.
- **Assumptions & Simplifications:**
- **Manual or Static Planning:** The scheduling decisions here are very rudimentary. We might initially hard-code some schedule or require the user to specify a schedule in the scenario (essentially manual intents) to drive this scope. This is acceptable in Scope 4: the focus is on getting the infrastructure (messages, timing, agent actions) correct. A more intelligent automatic scheduler comes in Scope 5.
- **Single-flow focus:** We assume scenarios with one or few ServiceRequests so that managing them is straightforward. Complex contention between flows isn't handled yet (we might just serve all flows if possible, without prioritization or capacity sharing).
- **No dynamic re-routing:** If a route goes down (beam off due to geometry), we will simply mark the ServiceRequest as not provisioned until the next window – no alternate path computation in this scope beyond the single planned route.
- **Telemetry volume:** We choose a coarse telemetry interval (e.g. 1 sim-minute) to avoid performance issues. It can be configured. The detail of telemetry (like per-second counters) is kept minimal – enough to show the feature working.
- **Agent internals:** We do not simulate any on-device delays or failures in applying commands. Every scheduled task is assumed to execute on time (unless we explicitly test a failure case). Control-plane latency is effectively zero in sim unless we choose to simulate it by offsetting execution times (not in this scope).
- **Dependencies:** Requires Scopes 1–3. By now the simulator has a fully defined scenario via NBI and knows potential links from Scope 2; Scope 4 ties it together by actively controlling those links. After this phase, the simulation can **run in time**: satellites move, come into contact, the controller issues beam commands, and we see links go up and down accordingly. This provides a basic end-to-end demonstration, suitable as an early "tech preview" (e.g. **v0.5 milestone** where basic scheduling and telemetry are proven, though not optimized). It also establishes the framework where future contributors could plug in more advanced scheduling logic without altering the API plumbing.

Scope 5: Advanced Orchestration & Routing Logic (Full Capability)

- **Goals & Deliverables:** Develop the intelligent planning algorithms that were simplified in Scope 4. In this phase, implement more comprehensive **network orchestration logic** to handle multiple ServiceRequests, dynamic topology changes, routing, and resource conflicts. The deliverable is a simulator that can automatically compute schedules and routes to satisfy as much of the demanded traffic as possible, respecting the network's time-varying constraints. This brings the simulator in line with the full set of requirements: multi-hop routing, DTN store-and-forward, priority-based allocation, and basic failure recovery are addressed.
- **Covers Requirements:** Fully covers *Network Planning and Orchestration Logic*, and revisits earlier sections to implement anything that was postponed (e.g. using node power budgets in scheduling decisions, enforcing multi-beam limits, etc.). By the end of this scope, **all requirements in the document are either implemented or explicitly stubbed**.
- **Included Implementation:**
- **Contact Window Precomputation:** Extend the connectivity service from Scope 2 to compute and store time intervals (start/end times) for which each potential link is available (ACCESS_EXISTS) based on orbit predictions [75] [76]. This can be done at scenario load or continuously updated. These contact windows form a **time-varying graph** of connectivity.
- **Dynamic Route Computation:** Implement a routing module that can find end-to-end paths through the network graph for each ServiceRequest. Use shortest-path algorithms (Dijkstra or BFS, possibly with link cost = latency or hop count) to find feasible routes [77] [78]. Because the graph changes over time, decide on a strategy:
    - For disruption-tolerant flows (DTN enabled), allow the route to be time-segmented. We might compute a *static multi-hop path* in terms of sequence of nodes, but not require it to be simultaneously connected. Instead, we plan for store-and-forward: e.g. data can hop to a satellite, wait until that satellite sees the next node, then forward [79] [80]. Implement logic to utilize node storage: if a link in the path is down, buffer outgoing data up to the node's `storage_capacity` and deliver when the link comes up [81] [82]. The ServiceRequest can still be marked *provisioned* overall if data eventually gets through, in line with DTN principles.
    - For non-DTN flows (require continuous end-to-end), ensure a route is active *in real-time*. This could involve computing a series of handovers: e.g. at time T route via Sat1->GS1, later switch to Sat1->GS2 when the satellite moves [83]. To simplify, we might choose a single path that is "best" (e.g. source->Sat1->dest) and whenever that path is broken, the flow is just down. A more advanced approach is to pre-compute multiple path options and schedule a make-before-break (this can be incremental improvement). Initially, implement at least the simplest: choose one path and stick to it, accept outages for non-DTN flows.
- **Scheduling Optimization:** Develop a scheduler that allocates **who uses each link when**. This involves resolving conflicts when, for example, two flows need the same satellite downlink simultaneously [84] [85]. Key aspects:
    - Obey interface constraints: ensure no interface (antenna) is assigned to more than one beam at the same time beyond its capacity (one beam per interface unless `max_beams > 1` was configured) [21] [86]. If a satellite has one dish and two ground stations demand it, the scheduler must split time or choose one.
    - Incorporate **flow priorities:** If capacity or time is insufficient to serve all flows, use ServiceRequest `priority` values to decide which flows get the resource first [87] [88]. Implement a simple scheme: try to fulfill higher priority requests first; lower priority ones may get skipped or partially served if there's a conflict.

    - Enforce **power and throughput limits:** Re-introduce the `power_budget` from Scope 1: if a satellite has a total transmit power cap, the scheduler should not activate more beams or links than can be powered simultaneously [5]. For example, if power budget allows 1 high-power beam or 2 low-power beams, and the scenario config indicates such, reflect that in scheduling (perhaps simply limit number of concurrent beams per node). Similarly, if we included a `max_processed_bandwidth` for a bent-pipe payload, ensure the sum of link capacities through that satellite at any time doesn't exceed it (or else schedule sequential usage).
    - Support **time-slicing**: Where simultaneous satisfaction is impossible, schedule flows in alternating time slots. For instance, if two user terminals share one satellite downlink, allocate each some percentage of each visibility pass (perhaps proportional to priority or requested bandwidth). This yields a timeline of beam activations: e.g., Terminal A gets first 30 seconds of each contact, Terminal B gets next 30 seconds, etc.
    - Use heuristic algorithms: a greedy approach can suffice (it's understood that an optimal solution is complex). For example: sort ServiceRequests by priority, then for each, find the earliest upcoming windows where a path exists and assign those windows to that flow (marking the involved links as occupied during those times), then move to the next flow. We ensure no link or interface is double-booked. This may result in suboptimal utilization but is implementable by a single developer. We will keep the scheduling code modular so that an external or more advanced optimizer can be plugged in later [89] [90].
- **Route Installation & Teardown:** Using the output of the routing and scheduling solver, generate the sequence of SBI commands needed:
    - For each link that needs to be activated at a given time, create an UpdateBeam entry for the respective agent at that time [91] [92]. Likewise, create DeleteBeam entries for when the link is no longer needed (end of window or when switching). If an interface will immediately repoint to a new target, the DeleteBeam of old and UpdateBeam of new might coincide or one replaces the other (handle P2P vs P2MP logic accordingly).
    - For each flow that gets a route through the network, issue SetRoute entries at the appropriate nodes. For simplicity, we can configure routes at the source node to use a specific next-hop (source routing), or at each intermediate hop like a traditional routing table. In Scope 5, we implement actual per-node routing tables so that multi-hop forwarding is represented: e.g. Node A gets a route for dest=Z via interface X, Node B gets route for dest=Z via interface Y, etc., for the path [55] [56]. These are timed to go in effect when the link beams are active, and DeleteRoute when links go down.
    - Maintain Intent state (if we choose to track Intents for debugging): e.g. mark an Intent as INSTALLED once all its associated tasks are acknowledged by agents. Update ServiceRequest status `is_provisioned_now` based on whether an active route exists at the current time for that request [37]. Also populate `provisioned_intervals` for each request – e.g. store the time ranges during which the request was fulfilled (we get this from our schedule).
- **DTN Flow Handling:** Ensure that if `is_disruption_tolerant=true` for a ServiceRequest, the lack of a continuous path does not mark it immediately as failed [81]. Instead, implement buffering logic: when a DTN flow's path is broken, simulate queuing data on the last node that had it. Use the node's `storage_capacity` to limit how much can be queued [93] [94]. When connectivity resumes, release the data. For telemetry, report storage utilization (optional) and for ServiceRequest status, perhaps indicate that it's *partially* provisioned (some data in transit). This aligns the simulator with Bundle Protocol concepts without implementing them fully.
- **Reactive Re-planning:** Introduce a basic mechanism to react to unexpected events. If during simulation a link fails (due to an impairment injection or perhaps a missed contact opportunity), detect that the current route for a flow is broken and attempt an alternate route if available [95] [96]. This could be as simple as: if a route was active and now no longer viable, check the contact

graph for another route at the current time. If found, send new UpdateBeam/SetRoute commands (like a quick re-route). If not, mark the flow down. This is a rudimentary failover strategy. We won't implement full continuous re-optimization, but the framework allows the controller to reschedule on the fly (for example, we could manually trigger the solver again mid-simulation). We will log such events for operator visibility.

- **Assumptions & Simplifications:**
- **Solver optimality:** We do *not* attempt a globally optimal schedule in this scope – only a reasonable one. Advanced techniques (ILPs, constraint solvers) are out of scope for initial development, but our design leaves an integration point for them [89] [90]. For example, we could have an interface `ComputeSchedule(Scenario)` that currently calls our heuristic, but could later call an external optimizer contributed by others.
- **Beam hopping:** Explicit support for rapid beam hopping patterns (as in Aalyria's ModemIntent for timeslot hopping) is not implemented initially. We assume any rapid alternation needed is handled by our time-slicing approach. A true beam hopping feature (with sub-second timeslots and repeating patterns) is noted as a future extension [97] [98], but beyond our first release.
- **Latency and protocol effects:** We maintain simple latency constants for links (e.g. add ~20 ms for LEO, 250 ms for GEO) for informational purposes, but we do not simulate packet-level effects (no TCP window, no actual queueing delay except coarse store-forward for DTN) [99] [100]. This is acceptable for planning; we assume if a route is up, it meets any latency requirement if hop count is within reason (or we avoid GEO paths if latency constraint is strict, etc.).
- **Traffic generation:** Still, no real packet traffic is simulated. The fulfillment of ServiceRequests is abstract: if a flow is provisioned with X Mbps, we consider that bandwidth "used" on the links but we don't emit packets. Metrics are computed from these abstractions. Users could integrate an external traffic simulator later if needed (our design of separating control-plane and data-plane allows that).
- **Final stubs:** By end of Scope 5, the only features explicitly not implemented are those listed as out-of-scope in the requirements summary (e.g. detailed interference modeling, live adaptive coding, full NMTS model API, full federation support). We ensure all **"Implemented"** items from the requirements document's summary are done [101] [102], and **"Stubbed"** items remain as conscious simplifications [103] [104] (with hooks to implement later).
- **Dependencies:** This builds on the scheduling framework of Scope 4. It requires all prior scopes' functionalities. This is the most complex phase, likely iterative – as a solo dev, you might develop and refine these algorithms over multiple passes. By the end of Scope 5, the simulator should reach a **v1.0** level: it can ingest a realistic constellation scenario (with LEO satellites, ground stations, and user terminals), multiple service requests, and produce a time-sequenced plan of network operations (beam scheduling and routing) that attempts to satisfy the requests using the available resources, all while exposing the standard Spacetime APIs.

Scope 6: External Integration & Federation (Extended/Future Scope)

(This scope is optional and can be pursued once the core simulator is complete. It covers the remaining open-ended requirements to maximize realism and interoperability.)

- **Goals & Deliverables:** Enhance the simulator with support for external data/protocol integration and lay the groundwork for multi-network federation. The deliverables include utilities for importing/exporting standard formats (orbits, contact plans), stubs for Federation APIs, and design provisions for hardware/software-in-the-loop testing. This scope ensures the simulator is **extensible** and compatible with external systems and standards, although a solo developer may only implement parts of it initially.

- **Covers Requirements:** Addresses *External Protocol Integration and Open Standards* and *Federation and External Networks* sections. Many aspects of these were already considered in earlier scopes (e.g. TLE support, DTN concepts), but here we make them more robust and add any missing pieces. Federation support is mostly stubbed but accounted for, so the simulator does not preclude future expansion in that direction [105] [106] .
- **Included Implementation:**
- **Orbit Data Integration:** Expand orbit input capabilities beyond TLE. For example, implement a parser for CCSDS OEM (Orbit Ephemeris Message) files so users can provide high-precision ephemerides. Integrate with SPICE toolkit or other libraries if needed for special cases (e.g. lunar or deep-space orbits) – though these are not primary, having the option adds realism [107] [108] . Also, add a feature to fetch updated TLEs from an online source (Space-Track) if the simulation runs in real-time over long periods, keeping orbits in sync with live data (subject to API access).
- **Time Standards:** Ensure the simulation can output time in standard formats (UTC, GPS) and handle leap seconds if needed [109] . For instance, provide utilities to convert the internal simulation timestamp to real UTC strings for logging, and if a user wants to align simulation start with a real date/time, allow that input.
- **Standards Awareness:** While we do not implement link-layer protocols (CCSDS TM/TC, etc.), we document how the simulator's parameters relate. For example, if a user configures a link with DVB-S2X mode, we assume certain coding overhead and can adjust the effective throughput accordingly [110] [111] . We might incorporate lookup tables for spectral efficiency of a few modulation schemes to improve our capacity estimates. This way, our simulation can produce results (throughput, latency) that are consistent with real-world expectations for those standards.
- **Contact Plan Export:** Add the ability to export the computed contact windows or full schedule to external formats. For instance, output a list of contacts (sat A – ground B from T1 to T2) as a CSV or CCSDS *CPA* (Contact Plan) format if applicable. This allows integration with external DTN routing tools (e.g. NASA's CRCMS or scheduling algorithms) [112] [113] . Similarly, we could export satellite ephemerides or link events to files that tools like GMAT or STK could read for visualization [114] .
- **Hardware/Software-in-the-Loop Hooks:** Design the system so real external agents or radios could be connected. For example, instead of an internal agent thread, allow a real agent (like Aalyria's Go agent) to connect to our Scheduling Service – this would test interoperability. Similarly, if an external modem emulator is available, our scheduling messages (UpdateBeam with frequency/power) could be sent to it. In practice, implementing this might simply mean not assuming all agents are internal: run the gRPC service for Scheduling on a port and accept real agent connections. It's a bonus feature that could attract contributions. For now, we ensure that our control messages (beam configs, etc.) use the same proto definitions as real hardware would, so integration is possible [115] [116] .
- **Federation API Stub:** Implement placeholders for Federation-related APIs. If Spacetime has a Federation Service (for east-west inter-controller communication), define those RPC endpoints in our server but have them return a "Not Implemented" status or minimal functionality [105] [106] . For example, a call like "OfferServiceRequestToPartner" might just log that federation is not supported in the open simulator. Also, if `allow_partner_resources` is true on a ServiceRequest, we could simulate a simplistic partner link by allowing usage of a special "partner" node's capacity if it was manually added to the scenario. This is purely optional – mainly, we ensure nothing in our design prevents later addition of federation.
- **Partner Network Modeling:** If desired, demonstrate a simple federation scenario by creating a second network in the simulation (with its own nodes labeled as external) and manually orchestrating a shared link. For instance, treat one ground station as belonging to a partner – our scheduler would normally skip it unless `allow_partner_resources` is true, in which case

we could allow scheduling that link. This would be a rudimentary test of cross-network usage, without implementing partner controllers.

- **Assumptions & Simplifications:**
- This scope may not be strictly necessary for a working simulator, so a solo developer might choose to document these as **future work** rather than implement all. It's included to show how the architecture will accommodate these features so that open-source contributors or later efforts can add them cleanly.
- Federation in particular might remain unimplemented in the initial release. The important part is that earlier scopes did not paint us into a corner: for example, we kept `allow_partner_resources` in the data model, we can identify links as belonging to another network if needed, etc., so a federation module could be plugged in later.
- We do not integrate actual 5G core network or other terrestrial network simulators; we simply note that our nodes can represent gateways to such networks and any high latency or throughput issues can be approximated (e.g. adding extra delay on a satellite backhaul link to account for 5G traffic considerations) [117] [118] .
- All external integration points (TLE updates, external solver, real agents) should be off by default or manually invoked, such that they do not disrupt the core offline simulation if not in use.
- **Dependencies:** This builds on a stable core (Scopes 1–5). It does not necessarily gate the v1.0 release; it could be part of a **v1.1** or later milestone once the community is involved. However, some items like improved orbit import and contact plan export are low-hanging fruits that a solo dev can add alongside final testing.

Release Milestones (Versions)

- **v0.1 – Basic Entity Modeling:** After Scope 1 (and possibly parts of Scope 2), the simulator can load a simple static scenario and propagate satellite orbits. This early version demonstrates the core data model (platforms/nodes) and time progression but lacks networking. It's mainly for validating that we can represent the constellation and read TLEs correctly.
- **v0.5 – Minimal Network Simulation:** Around the completion of Scope 4, the simulator achieves a closed-loop of control: the NBI can define a scenario, and the SBI (with a naive scheduler) can drive beams on/off and routes in a time-stepped simulation [119] [120] . This version can be used to simulate simple scenarios (perhaps one satellite, one ground, one flow) in real-time or accelerated time, with telemetry output showing activity. It's a useful checkpoint to demonstrate end-to-end compatibility with Spacetime APIs (one could connect an Aalyria agent to the scheduling service at this point to verify interoperability).
- **v1.0 – Feature-Complete Simulator:** Upon finishing Scope 5, the simulator covers all major functionalities outlined in the requirements. It supports multi-satellite constellations, dynamic connectivity, automated scheduling/routing for multiple flows, DTN buffering, priority, etc., albeit with heuristic algorithms [77] [121] . All API surfaces (NBI/SBI) are implemented. Advanced features are either implemented or explicitly stubbed as per the plan (see *Summary of Features vs. Stubs* in the document, which our implementation aligns with) [101] [103] . This version is suitable for broader use by researchers or developers, and others can build UI front-ends or integrate more sophisticated optimizations on this base.
- **Post-1.0 – Extended Capabilities:** Incorporating Scope 6 and other enhancements. For example, **v1.1** might add richer integration (live data feeds, federation experimentation) and improved modeling (interference, adaptive coding, etc.). These versions would benefit from community contributions – the groundwork laid in earlier scopes ensures that contributors can plug in improvements (like a new scheduling solver, or a better physical layer model) without a major refactor.

Each phase above is scoped to be manageable by a single developer, delivering incremental value while building on prior work. By structuring the roadmap in these logical scopes, we ensure steady progress toward a fully Spacetime-compatible constellation simulator, with clear checkpoints for testing, community collaboration, and iterative refinement. The end result is an open-source backend that mirrors Aalyria Spacetime's key behavior and APIs, enabling experimentation and development in the satellite networking domain [122] [123]. All requirements from the document are covered across the phases, with initial simplifications gradually replaced by more realism, ensuring the project is achievable solo while laying a path for future growth.

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 Requirements for an Aalyria Spacetime-Compatible Constellation Simulator.pdf

file://file_0000000012b87206951eb78067248b9a