ChatGPT

Requirements for an Aalyria Spacetime-Compatible Constellation Simulator

Overview and Context

To build a **local constellation simulation backend** that is API-compatible with Aalyria's Spacetime, we must replicate the core concepts of Spacetime's temporospatial SDN platform. Spacetime provides a Northbound Interface (NBI) for defining and orchestrating networks, and a Southbound Interface (SBI) through which network devices receive schedule updates and report telemetry [1]. Our simulation backend should implement equivalent data models and services so that it can accept the same gRPC API calls (or textproto definitions) and produce similar behavior. This includes modeling satellites, ground stations, links (including steerable beams), network schedules, and routing, while simulating time-dynamic changes (orbital motion, connectivity variation, etc.). Key open standards (orbit ephemerides, networking protocols, etc.) will be leveraged to ensure realism and openness. Below is a structured breakdown of the required components and features.

Node and Platform Modeling

Physical Platforms: The simulation must represent physical entities like satellites (LEO/MEO/GEO), ground stations, airborne relays, user terminals, etc. Each is defined by a **PlatformDefinition** with a unique ID, a human-readable name/type, and **Motion** data (position/orbit). Platforms use an Earth-centered, Earth-fixed (ECEF) coordinate frame for locations and orientations. We must support dynamic positioning: for satellites, ingest Two-Line Element sets (TLE) and use an SGP4 propagator to update orbital coordinates over time (Spacetime can automatically fetch NORAD TLE updates via a `SPACETRACK_ORG` motion source). Our backend should periodically propagate each platform's orbit to reflect its real-time position and orientation. For non-orbital platforms (e.g. fixed ground sites or moving aircraft), other motion sources can be used (e.g. live ADS-B data or predefined trajectories). Initially, we can use open libraries like **Orekit** or SGP4 implementations for orbit propagation, and stub external sources with static or scripted motion if needed.

Network Nodes: Logical networking endpoints are modeled as **NetworkNode** entities. A `NetworkNode` aggregates one or more network interfaces and defines the networking parameters of a device or subnet [2]. Each node has a unique `node_id`, an optional name/type for human reference, and can represent an entire device (like a satellite's communication subsystem, a user terminal, a ground router) or even an entire remote network. The node may optionally link to a Platform: in Spacetime's model, a node can exist **with or without an associated Platform** (e.g. a "Point of Presence" node might have no platform) [3]. In our simulation, when a node corresponds to a physical asset like a satellite or terminal, it should be associated with a PlatformDefinition (to inherit position/motion). Each NetworkNode can specify:
- **Routing configuration:** e.g. a router ID or segment routing ID for the node [4] if needed for path computation. This is mainly to integrate with IP routing or segment routing features but can be stubbed if not used initially.
- **Subnets:** IP subnet ranges the node serves or gateway prefixes [5].
- **Network interfaces:** (detailed below) representing links from this node into the network [6].
- **SDN Agent settings:** A `SdnAgent` sub-message to indicate if this node runs a Spacetime agent for

control-plane integration [7] [8]. In practice, this field (including an agent type and control-plane latency mapping) configures whether the controller can remotely manage routing on the node [9]. For our simulation, all constellation nodes that we control will likely be "SDN-enabled," meaning the simulator will accept routing/beam commands on them. The `maximum_control_plane_latency` map can be used to model communication delays in delivering control commands (e.g. if a node is only reachable via intermittent links or long RTT) [10]. Initially, we can simplify by assuming negligible control latency or a fixed nominal latency, but the structure is there to simulate delayed command propagation if needed.

- **Power constraints:** Nodes can optionally define a **signal power budget** – essentially a cap on total transmission power across all its radios [11]. This is useful for satellites or platforms with limited power. Our backend should accept a `power_budget` (with time intervals and available watts) and ensure that scheduling or link modeling does not exceed those limits (or simply log warnings if exceeded, in a stubbed manner).

- **Onboard storage (DTN support):** Each node has an optional `storage` capacity (bytes) for store-and-forward networking [12]. This directly relates to Delay/Disruption Tolerant Networking (DTN) capabilities. If `storage.available_bytes` is non-negative, the node can buffer data for later forwarding [12]. Our simulation should expose this parameter and honor it when simulating flows marked as disruption-tolerant. For example, if a **ServiceRequest** is flagged `is_disruption_tolerant=true` [13], it means the traffic can be delayed and buffered; the network should use the node's storage to hold data during link outages. Initially, we can implement a basic model: e.g. track stored bytes and ensure it doesn't exceed `available_bytes`. Full BPv7 protocol behavior (custody transfer, etc.) is beyond scope, but acknowledging the concept is important. This aligns with CCSDS DTN standards (Bundle Protocol 6/7).

Node Categories: While not explicitly enumerated in the proto, the `type` or `category_tag` fields of PlatformDefinition and NetworkNode can be used to label node roles [14] (e.g. "SATELLITE", "GROUND_STATION", "USER_TERMINAL"). In the provided scenario samples, for instance, they define *satellite_network_node*, *gateway_network_node*, *user_terminal_network_node*, etc., each presumably using a `type` or tag to classify. Our system should support these labels for scenario organization and possibly to apply domain-specific logic (e.g. a satellite node might auto-enable some orbital motion updates, a ground gateway might not move, etc.). We will include common constellation node types: - **Satellite nodes:** attached to moving orbital platforms (with altitude/orbit parameters). - **Ground station nodes (Gateways):** attached to fixed platforms at given earth coordinates. - **User terminal nodes (UT):** possibly mobile or fixed on ground, representing end-user devices. - **Network infrastructure nodes:** such as point-of-presence routers or internet gateways, which may not have a physical platform in Spacetime (modeled as NetworkNode without a Platform [3], e.g. an abstract node representing a connection to the internet backbone).

These various nodes and their relationships form the **topology graph** of the network, which in our simulator must evolve over time as platform positions change. We will treat the collection of all nodes, links, and service requests as a "Scenario" (in line with Spacetime's Simulation API concept of a Scenario [15]). The backend should maintain this graph and allow querying or exporting it via API calls (for inspection/debugging).

Network Interfaces and Transceivers

Each NetworkNode contains one or more **NetworkInterface** objects, representing the actual communication links (ports, radios, antennas) on the node [6]. We need to support modeling of both wired and wireless interfaces:

- **NetworkInterface ID:** Every interface has an ID unique within its node (and a composite global ID as `{node_id, interface_id}`) [16]. The interface can also have a human-friendly name and possibly a MAC address if relevant [17] [18]. The interface IP address (IPv4/IPv6 in CIDR notation) is stored for networking purposes [17]. Our simulation should track these for routing decisions and flow classification.

- **Wired vs Wireless Medium:** An interface specifies its medium via a oneof. A **WiredDevice** indicates a static link (e.g. an Ethernet port, fiber, or a fixed microwave link), whereas a **WirelessDevice** represents a radio transceiver [19]. For wired interfaces, we model:

- *Max data rate:* a bandwidth cap in bits per second [20]. E.g., if an inter-site fiber is 1 Gbps, that is the max throughput.
- *Associated platform (optional):* a wired link can optionally reference a `platform_id` [21] for visualization – e.g., if it's a cable connecting to a specific platform. This is just metadata; the simulator can use it to place the link on the map or skip it.
  Initially, for simplicity, we can assume any wired links (like terrestrial fiber) are always "up" and have a fixed latency (which we can configure globally or per link). We won't need to compute line-of-sight for wired links, but we may allow the user to define if a wired link is down via an impairment or downtime (see below).

For **WirelessDevice** interfaces, more complex modeling is needed:
- *Transceiver model:* Each wireless interface references a **TransceiverModelId** [22], pointing to a database of radio hardware characteristics. In Spacetime, the "SDN Store" holds transceiver models (antenna gain patterns, frequency bands, transmit power limits, sensitivity, etc.). Our open-source simulator should define a schema for transceiver models and allow loading some generic or example models (possibly drawn from the `contrib` directory, such as a Loon balloon antenna pattern [23]). At minimum, for each transceiver type, we need basic parameters: frequency bands, antenna beam pattern (gain vs angle), EIRP and G/T (gain over noise temperature) or equivalently transmitter power and receiver noise figure, supported modulation/coding options, and polarization. This will feed the link budget calculations. We can start with a simplified model (e.g. isotropic antenna or a few discrete beam modes) and allow plugging in more detailed patterns later.

- *Link establishment timeout:* A wireless interface may define a `link_establishment_timeout` duration [24]. This is the time it will attempt to lock a beam on a target before giving up. Our scheduling engine should respect this by ensuring beam pointing commands allow at most that much lead time. We can simulate this by e.g. if a beam is commanded and the link does not become viable within that timeout, mark the attempt as failed. This may be a more advanced feature, so initially we might just store the value and not actively simulate a failure unless we explicitly model acquisition times.

- *Physical chain identifiers:* Spacetime's model includes a set of **NMTS** entity IDs in the WirelessDevice: platform, port, modulator, transmitter, antenna, etc. (both TX and RX side) [25] [26]. These correspond to a detailed hardware chain – e.g., an **EK_ANTENNA** or similar in the NMTS graph. In an NMTS-enabled environment, each of those IDs would link to a specific hardware element with its own properties. For our simulator, we don't have the full NMTS

infrastructure (though it is open-sourced separately). Initially, we will treat these fields as **stubs**: we can fill them with dummy identifiers or leave empty. The key is to note that our data model should accommodate them for future compatibility. If needed, we may map the `transceiver_model_id` to some internal objects representing antenna gains, but we won't individually model each amplifier or modem as separate entities – instead, we'll roll their effects into the transceiver model performance.

- *Operational impairments:* We should allow marking an interface as down or unreliable. In the NBI, `NetworkInterface` has an `operational_impairment` list, to denote if an interface is intentionally taken out of service or suffering degraded performance [27] [28]. For example, an interface could be flagged as `DEFAULT_UNUSABLE` (completely down) by an application along with a reason and timestamp. Our simulation can expose a method to set such impairments (or schedule them over time as "downtimes"). For now, we might simulate these by simply toggling whether a given link is considered available in connectivity calculations.

NetworkInterfaceId and relationships: Every wireless interface on a node that is meant to connect via space should correspond to a **steerable antenna** or fixed beam. Interfaces are referenced in link definitions by `NetworkInterfaceId` (combining node and interface ID) [29]. For example, a *satellite* node might have a wireless interface representing an RF antenna; a *ground station* node has an interface for its dish. The simulation must maintain the mapping of interface IDs to actual endpoints. This is crucial for link establishment: when we schedule a "beam" or link, we'll refer to these interface IDs.

Link, Beam, and Connectivity Modeling

Network Links: Spacetime's NBI models connectivity via **NetworkLink** objects (directional links between interfaces) [29] and related structures. Our backend must let users or automated solvers create link objects that tie two network interfaces together. There are two primary link types:
- **Directional (unidirectional) link:** Represented by `NetworkLink` message, with a source interface and destination interface [29]. This indicates traffic can flow one-way from src to dst under the right conditions. For instance, a downlink from a satellite transmitter to a ground station receiver is one directional link, and the uplink would be a separate link (or we use a Bidirectional link container, see below).
- **Bidirectional link:** Represented by `BidirectionalLink` combining two LinkEnds (A and B) [30]. This is essentially a convenience to group a pair of opposite-direction links between two nodes. Our simulation can support both concepts. We might treat Bidirectional links as a pair of directional links that share some properties. The proto allows specifying distinct radio parameters for A→B vs B→A within a bidirectional link [31] (to handle asymmetric links).

Link radio parameters: For wireless links, there are configurable parameters that can be set either in the Intent (for dynamic control) or in the link resource itself. In `BidirectionalLink`, optional `Radio` messages can be set for each direction [32]. In a `DirectionalLink` (used in `LinkIntent`), you can attach a `Radio` config as well [33]. The **Radio** message includes frequency, bandwidth, polarization, and possibly power settings [34] [35]. In particular, `Radio.TxRadioConfiguration` can specify an EIRP or power level (with choice of units like absolute watts or EIRP spectral density) [35]. It also includes a `modulator_id` and a mode (e.g. `DVB_S2X` or `Digital_Transparent`) to describe the modem type [36]. Similarly, `RxRadioConfiguration` can specify a `demodulator_id` and mode (like `DVB_S2X`, `DVB_RCS2_TDMA`, etc.) [37]. These modes show alignment with open standards: DVB-S2X and DVB-RCS2 for satcom, as well as digital transparent processing. Our simulator should include these enumerations in the data model (as they are in the proto) [38], and use them to influence link

performance. For example, **ModCod** (modulation and coding scheme) selection could be represented by these modes or further by a `band_profile_id` in the Radio config. Initially, we might not simulate adaptive modcod in real-time, but we should allow specifying a modulation profile to compute link throughput (e.g. if mode is DVB-S2X, assume a certain spectral efficiency given the configured bandwidth and that might yield a max throughput). We can incorporate standard values from DVB-S2X (like QPSK, 16QAM rates) or use an abstract "rate table" approach (Spacetime's `RadioConfiguration` has a `rate_table_id` to model adaptive data rate tables [39]). For now, we could use a simplified model (e.g. each link has a fixed capacity or one we compute from a simple formula).

Beam Targets and Pointing: A critical concept for constellation networks is that many wireless links are not permanent fixed connections – they depend on antenna pointing (steering a beam to a target). The Spacetime model uses **BeamTarget** and **BeamUpdate** to manage this. We must implement the notion of a **steerable beam** on a wireless interface and the ability to task it toward a target at given times.

- **BeamTarget definition:** A `BeamTarget` can be specified in multiple ways [40]. It's either another network element's transceiver (`transceiver_id` referencing a TransceiverModel instance on the peer), a specific platform ID (used in legacy NMTS contexts) [41], or explicit coordinates (a Motion trajectory or coordinate point) [42]. In our simulation, the common case will be targeting another node's interface. So we'll likely identify beam targets by the destination interface's NetworkInterfaceId (node + interface). We should still handle the case of targeting *coordinates* – e.g., pointing an antenna to a specific lat/long if the other node isn't explicitly modeled (perhaps for pointing tests or communication with something not in the model). The presence of `nmts.v1.types.geophys.Motion` in the BeamTarget oneof [42] implies the system can even target a predicted position (for a moving object not fully in the model graph). We can enable this by allowing an API call that directly assigns an Az/El or coordinate for the beam (useful for external targets).

- **BeamUpdate tasks:** The SBI uses **BeamUpdate** messages to command an antenna to add or remove a beam pointing task [43]. In our scheduling model, whenever the controller wants to establish a link at a specific time, it will schedule an `UpdateBeam` action (which contains a Beam configuration) to be sent to the agent at that time. We need to represent an active beam in the simulation state. A beam is characterized by: which antenna interface (`interface_id`) is emitting/receiving, who the target is (`target_id`), and the radio parameters (frequency, etc.) [44] [45]. We should also track a unique `beam_task_id` for each beam task [46] so they can be referenced or canceled. In simulation, when a beam is "active," we consider the link between those two interfaces as *present* (subject to propagation conditions). When it's removed (either by explicit DELETE operation or by scheduling expiry), that link is no longer available.

- **Concurrent beams and exclusivity:** Spacetime differentiates point-to-point vs point-to-multipoint antennas. For P2P antennas, only one BeamUpdate can be active at a time – adding a new beam automatically removes the prior one [43]. For P2MP (multi-user) antennas, you can add multiple beams concurrently (like a phased array serving multiple terminals) [43]. Our model should allow configuration of each interface's capability: e.g., an interface might have `max_beams = N`. Initially, we might keep it simple (1 beam per interface unless explicitly stated otherwise). But it's worth noting as a requirement if multi-beam satellites are in scope, we should allow multiple target links simultaneously from one transmitter (with presumably power divided or separate channels).

- **Beam scheduling and state:** We will maintain a mapping of current active beams (perhaps per agent). A **BeamStates** message in Spacetime holds the list of active beam task IDs and possibly

their statuses [47]. Our agent simulation can similarly expose which beams are currently on, for debugging or API queries.

- **Line-of-sight and link budget:** Even if a beam is commanded, the link only exists if geometry allows. The simulation backend must determine **visibility windows** between platforms. This requires computing when two nodes have direct line-of-sight (for RF or optical links, typically no Earth obstruction) and within beam range (antenna field of view). For Earth-orbiting satellites, line-of-sight to a ground station exists when the satellite is above the horizon as seen from the ground (elevation > 0°, or some minimum elevation mask like 5-10° to account for terrain/radio horizon). Between two satellites, line-of-sight is usually assumed if no obstruction, but if we consider Earth in between for low altitude, we need to check if the straight line between them intersects Earth. We should incorporate or reference standard orbital geometry calculations to determine these windows, using each platform's propagated coordinates.

Additionally, if the beam has a limited steering range (e.g., a gimbaled antenna on a satellite might only cover a certain cone), that can be modeled either via the antenna pattern (gain goes to zero beyond some angle) or explicit constraints. As an initial approach, we will assume full hemispherical coverage for simplicity, then refine if needed.

When a link is geometrically possible, we then calculate signal quality. A **WirelessLinkBudget** model is defined in the API to represent physical-layer link metrics [48]. Key factors: transmitter power and gain, free-space path loss (frequency & distance), atmospheric attenuation (for RF) or weather (for optical), receiver gain, and noise. We plan to implement a basic link budget calculator using standard formulas (e.g., Friis transmission equation). For openness, we can utilize known models: e.g., **ITU-R P.618** for atmospheric attenuation if we go deep, or simpler clear-sky assumptions initially. The **antenna gain pattern** from the transceiver model (if provided via an AntennaPattern resource) would be used to adjust gain based on pointing angle off boresight [49]. To integrate this, our simulation might load antenna gain vs angle tables for each interface (e.g., Loon's antenna pattern textproto as an example [49]). If no pattern is given, assume isotropic or a reasonable approximation. The **Polarization** must also match (the API enumerates LHCP/RHCP [50], so we ensure both ends either have the same polarization or we account for mismatch losses).

All of this results in an estimated **received power** (and signal-to-noise). If above a threshold for the chosen modulation (e.g., threshold for QPSK), then the link can carry data at the expected rate. If below, the link is effectively down (or at least has high errors). We don't necessarily need to simulate bit-level errors, but we should decide a threshold for "link accessible" vs "not accessible". The API's **Accessibility** enum (ACCESS_EXISTS, MARGINAL, NO_ACCESS) [51] could be used to categorize links. We can define: if margin > 0, link exists; if around threshold, maybe marginal; if below, no access. These determinations could be periodically updated by a *Link Evaluation service*. In Spacetime, a "Link Evaluator" service populates InterfaceLinkReports with metrics for each possible wireless link if real measurements aren't available [52]. For our simulation, implementing a periodic **Link Evaluator** is crucial: it will examine each potential pair of wireless interfaces, compute if they can connect at the current time (and what data rate or pathloss), and expose that to the rest of the system. This can happen in real-time (every simulation tick) or whenever something changes (node moves into/out-of range). Initially, we can run it on a schedule (e.g., every few seconds of sim time).

Interference and frequency reuse: In advanced scenarios, if multiple beams use the same frequency in overlapping areas, interference could reduce capacity. The API has placeholders like `wireless_interference.proto` and interference constraints in Intent. For now, we note that our simulation doesn't initially implement detailed co-channel interference. We assume links operate on separate channels or use perfect scheduling to avoid conflict. We will however allow frequency channels

to be set in the Radio config, so that future enhancements or user experiments can simulate interference by overlapping frequencies and then perhaps manually adjusting link performance. (If needed, a simplistic interference model: if two beams within a certain angular proximity use the same frequency, mark both as degraded or disallowed. But we will treat this as an advanced feature.)

Northbound API: Entities and Operations

To the outside (e.g. user scripts or higher-level orchestration), our backend should present a **Northbound API** that mirrors Spacetime's. All major entity types should be supported with Create/Update/Delete and Get/List operations. Key NBI object types include:

- **PlatformDefinition:** Physical entity as described, created to represent each satellite, ground site, etc. The NBI would allow creating a Platform with initial coordinates, motion source (manual, TLE, etc.), and child components (like antenna payloads). The Spacetime API actually defines complex payloads (see `bent_pipe.proto` for satellite payload modeling [53] [54]), including definitions of antennas and signal processors on the platform. For completeness, we list that if the constellation has bent-pipe satellites with configurable channel plans, the simulation could implement those via **BentPipePayload** (with fixed or digital channel configs) [55] [56]. However, implementing full payload config is a deep level of detail. We will likely **stub** the payload control initially – assume satellites either transparently relay (with maybe a fixed processing delay) or have no onboard processing limitations. But the data structures exist in case we want to enforce, say, a limit on how many MHz can go through a satellite or how channels are connected. Our requirement is to expose a way to define a payload with certain transponders and possibly respect a total throughput cap (`max_processed_bandwidth_hz`) [57].

- **NetworkNode:** Created for each logical network node. When creating a node, clients will include its properties (node_id, name, type, etc., plus interfaces and agent config). We should support adding interfaces (each with their own fields as above). Possibly the NBI may require creating interfaces via separate calls or all embedded in the NetworkNode definition. In the provided API, `NetworkNode` contains the repeated `node_interface` list [6], so it seems interfaces are defined inline with the node. That means our CreateNode RPC should accept a full node spec with interfaces. Alternatively, since Spacetime is evolving to a graph model (NMTS Entity/Relationship) [58], the newer approach might treat interfaces and nodes as separate entities linked by relationships. Indeed, the **Model API** likely supersedes some of these. For our purposes, we can initially implement the simpler model (node with embedded interfaces) to get up and running, ensuring at least compatibility with older NBI usage.

- **NetworkLink (Directional and Bidirectional):** A link can be created via NBI to statically declare that two interfaces *could* be connected. In Spacetime's NBI, there isn't a distinct "NetworkLink entity" type in the enum (the EntityType list includes NETWORK_NODE, PLATFORM_DEFINITION, SERVICE_REQUEST, etc., but not an explicit network link) [59]. Instead, links are often created indirectly as part of Intents or via the **Provisioning API**. However, they do have `resources/network_link.proto` which defines the message, and the Intent messages use `NetworkLink` for path segments [60]. Also, `InterfaceLinkReport` exists as an entity type for reporting link status. In our simulation, we should allow static link definitions for things like fixed terrestrial links. For dynamic satellite-to-ground or inter-satellite, it may be more appropriate to rely on the **Intent/Beam** mechanism rather than static entities. The "Building a Scenario" tutorial likely had the user define some static links (perhaps the terrestrial fiber links as

seen in `terrestrial_link_forward.textproto`, etc.). We will support an NBI call or config to define such static links with given endpoints.

- **ServiceRequest:** Represents a demand for connectivity (network flow provisioning request) [61]. The NBI should accept creation of service requests specifying source and destination (either by node ID or by "devices in region" group) [62], along with traffic requirements. Our simulation will register these requests and then attempt to satisfy them via scheduling and routing. Key fields to implement:

- `src_node_id` and `dst_node_id` (or region-based variants) [62]. We primarily focus on node-to-node flows (region could be used if we want to say "any satellite in this region to ground" – possibly out of initial scope).
- `FlowRequirements`: list including desired bandwidth and latency over certain time intervals [63] [64]. Often there might be just one requirement covering the entire scenario timeline (best-effort or a specified period). The simulator should store min and requested bandwidth and try to allocate routes that meet at least the minimum. If latency maximum is given, our route selection or scheduling should aim to meet it (e.g. avoid too many hops or store-and-forward if latency is tight). If `is_disruption_tolerant` is true [13], we interpret that as the flow can be achieved via DTN (data can be buffered and delivered later). This means the flow can still be considered "provisioned" even if instantaneous end-to-end path doesn't always exist, as long as data is eventually moved. We will need logic to handle this: for example, if a LEO satellite can only contact a ground station intermittently, a DTN flow might be satisfied by storing data on the satellite until the next contact. Our simulator can model this by accumulating bits in the satellite's storage when out of view, then draining during passes.
- Priority: ServiceRequests have a `priority` value [65]. If multiple flows contend for limited capacity, the scheduler should use this to decide which gets served first. We'll include this in our scheduling logic design (e.g. a higher priority flow may preempt a lower one or get allocated first in a constrained link).
- `allow_partner_resources`: indicates if the flow can use Federation (other networks' assets) [66]. We likely will not implement federation initially, so we can accept this field but ignore it (or log that partner resources are not available in the open simulation unless manually added).

- Status fields: `is_provisioned_now`, `provisioned_intervals` [67] should be maintained by our simulator to indicate when the request is satisfied. Essentially, once our scheduling finds a solution (a route with enough capacity), we mark it provisioned (and possibly at specific times if it's only intermittently met). We will update this state so that an NBI client querying the ServiceRequest can see if it's currently fulfilled.

- **Intent:** In Spacetime, Intents represent compiled actions (like "set up this link or route") as a result of satisfying ServiceRequests or direct user asks [68] [69]. Intents get compiled to **ScheduledControlUpdate** tasks that go out to agents [70] [71]. We may not expose Intents directly to end users in our simulator (they might be internal artifacts of our planning process), but we should be aware of them. For instance, fulfilling a ServiceRequest might involve creating a **LinkIntent** (to schedule beams) and a **PathIntent** (to configure routing) behind the scenes [72] [73]. The outcome of those Intents (successful or failed) would determine the ServiceRequest's state. Our system could simplify by directly scheduling the needed actions without explicitly modeling the Intent state machine, or we could choose to expose a rudimentary Intent for debugging. However, for API compatibility, it might be useful to at least have the **Intent resource** type in our database and allow listing them, to mimic Spacetime's behavior. We would include fields like `state` (SCHEDULED, INSTALLED, FAILED, etc. as per the enum) [74] and references to which ServiceRequest they support [75]. We will likely not implement the full ONOS-

derived intent process (with compile/withdraw phases) in detail, but at minimum, *when* our simulator decides on a solution (route/beam schedule), we can create an Intent object representing that solution.

- **Coverage and region entities:** The protos include things like `DevicesInRegion`, `SurfaceRegion`, `StationSet`. These are used to group nodes or define geographic areas for bulk operations (for example, *DevicesInRegion* might dynamically list devices currently in a geographic region). For a first version, we can skip implementing these, unless needed for specific scenarios. ServiceRequests can use `src_devices_in_region_id` or `dst_devices_in_region_id` [62] to specify a region rather than a single node (e.g. "any satellite over region X can source the data"). If such use-cases are needed, we might implement a simple **SurfaceRegion** entity (e.g. lat-long polygon or altitude band) and allow a DevicesInRegion query to find nodes currently inside it (leveraging our coordinate propagation). This is an advanced feature; initial open-source backend could require explicit endpoints instead. We note it as a potential extension.

- **Antenna patterns and other resources:** The NBI `antenna_pattern.proto` defines how to input an antenna radiation pattern (gain vs angle) for use in link budgets. We should support ingesting these (likely via textproto or as part of transceiver model). Similarly, `band_profile.proto` might define spectral bands and allowed modcods. Incorporating these resources ensures completeness. For now, we plan to allow static configuration of antenna gain patterns and frequency bands in the transceiver model definitions, with API placeholders if needed (e.g., an AntennaPattern resource that can be created/uploaded and referenced by transceivers).

NBI RPCs: We need to implement the RPC methods to manage the above entities. This includes: - `CreateEntity`, `UpdateEntity`, `GetEntity`, `ListEntities`, `DeleteEntity` for backward compatibility with the older NetOps service [76] [77]. Although Spacetime is migrating to a Model API, we should at least support the basic CRUD for the main entity types through a similar interface. (The actual Spacetime code shows these NetOps RPCs as deprecated [77], replaced by the Model API. If we aim for future-proofing, we might implement directly the **Model API** style: i.e., treat everything as NMTS `Entity` and `Relationship` objects. That would be a graph-based approach: each Platform, Node, Interface, etc. is an Entity node, with typed relationships connecting them. The SimulationService RPCs list in the proto uses `nmts.v1.Entity` and `nmts.v1.Relationship` for network model management [78] [79]. Doing this fully is complex, but it's an eventual goal for parity. As a compromise, we could internally use a graph model, but expose a simpler CRUD for main objects to clients.)

- `ListEntitiesOverTime` might be less critical (it's used to query historical changes); since our simulator can run faster than real-time or allow time-jumps, we could omit history tracking for now or log changes in a simple way.

- `VersionInfo` RPC (to return simulation backend version) can be stubbed.

In summary, the NBI of our system will allow building the scenario: adding platforms, nodes (with interfaces), linking them, and creating service demands. It will expose their state (so that after simulation runs, one can query the final states, e.g., ServiceRequest showing provisioned intervals, or NetworkInterfaces showing stats).

Southbound API: Scheduling and Telemetry Services

To emulate Spacetime's operation, our backend must implement the **Southbound Interface (SBI)** services, primarily Scheduling and Telemetry, so that an **agent** (running on each node or simulated within the backend) can communicate with the controller. In our simulation, we might implement a simplified agent internally (e.g., as a thread or module per node) that uses these SBI APIs to receive commands and send back acknowledgments/metrics, just as a real agent would. This design allows us to test the full message flow and ensures compatibility with any existing Spacetime agents (for example, Aalyria provides a Go-based SBI agent in the repository [80] which could potentially connect to our backend). Key components:

- **Scheduling Service (CDPI – Control-Data-Plane Interface):** This is a gRPC bidirectional stream used by the controller to send time-tagged configuration updates to an agent, and for the agent to respond [81] [82]. Our backend will implement the **Scheduling RPC** `ReceiveRequests` where the server (controller) side writes requests and the client (agent) side reads them. In practice:
- The agent initiates the stream by sending a **Hello** message with its `agent_id` [83]. In our simulation, if the agent is embedded, this can be done automatically at node startup; if we run separate agent processes, they will connect and identify themselves. The controller (our backend) uses `agent_id` to map to a specific node in the model.
- The controller will then send down zero or more **requests** (oneof in `ReceiveRequestsMessageFromController`) such as `CreateEntryRequest` for a new schedule entry [84], or `DeleteEntryRequest`, etc. Each request has a `request_id` and the agent must later respond with that ID and a status [85] [86].

- We need to implement handling of these specific request types:

    - **CreateEntryRequest:** This carries a schedule entry to add [87]. It includes an `id` (entry ID), a `time` (when to execute), and a **configuration_change** oneof that will be one of: `UpdateBeam`, `DeleteBeam`, `SetRoute`, `DeleteRoute`, `SetSrPolicy`, `DeleteSrPolicy` [88]. Our backend, upon deciding to schedule an action, will send an UpdateBeam or SetRoute, etc., to the appropriate agent's stream. The agent (simulated) should apply it at the given `time`. Since we are simulating time, we have two choices: we could deliver these requests in real-time (meaning if the simulation is running at wall-clock, we send when time approaches), or, more manageably, we treat `time` as a tag and have the agent insert the event into a local schedule (list of future events). The provided `seqno` and `schedule_manipulation_token` are there to ensure consistency (we should implement basic checking: if an agent Reset occurs, the token changes and any old request must be rejected) [89] [90]. Initially, we can be forgiving (since it's local), but to mimic Spacetime exactly, we should generate a new token on each reset and attach it to scheduling messages.
    - **UpdateBeam:** Contains a `Beam` message and an update mask [91]. The `Beam` in the scheduling context likely includes which interface and target, and any changes to beam parameters. We will form this based on what link we want to activate. For example, if scheduling a satellite-to-ground link at time T, the controller sends an UpdateBeam with the satellite's interface id, target = ground interface id, and maybe frequency/power settings. The agent then knows at time T to point that antenna to the target and configure frequency/power. In our simulation, the agent will mark that beam as active in its state at time T. If our simulation time is not real-time, the agent could immediately execute if time T "has passed" in simulation or wait until sim clock reaches T. Since we

may run the sim faster or in steps, a discrete-event approach might be easier: simply queue the beam task in the agent's schedule.

- **DeleteBeam:** Tells the agent to turn off a beam (identified by ID) at a certain time. We'll implement that by removing the corresponding task from the schedule at the given time.
- **SetRoute / DeleteRoute:** These correspond to configuring IP routing rules on the node. `SetRoute` provides a route entry with fields analogous to a Linux routing table (source prefix, destination prefix, output device, next-hop) [92] [93]. The agent should install this route into the node's forwarding plane at the given time. Our simulation can maintain a routing table per node (even if just in memory) – this is important if we want to simulate multi-hop forwarding. Alternatively, for a simpler approach, we can assume the controller computes a source routing path and the agent doesn't need to store routes (i.e., we use explicit path configuration like segment routing). However, since the NBI and SBI clearly support traditional routing, we should implement it. We could simulate packet forwarding on these routes if we later incorporate actual traffic flows; initially, it may suffice to track that "Node X has a route to Node Y via interface Z" for connectivity logic. `DeleteRoute` instructs removal of such a route.

- **SetSrPolicy / DeleteSrPolicy:** These relate to segment routing policies (the provisioning API defines TE policies). We should plan to accept and store segment routing policies if provided, but in early stages, we might not simulate the full effect. A segment routing policy basically pre-defines a path (list of segments) for certain traffic. If implementing, we'd mirror the structure from `provisioning.proto`. We might choose to stub this initially unless needed for scenarios.

- **FinalizeRequest:** Informs the agent that all entries before a certain time are finalized and won't change [94]. This is used for cleanup – e.g., after time X, you can drop old schedule entries. Our simulated agent can simply acknowledge it. We may implement it to remove any stored events < X from its schedule.

- **Reset (SBI):** The Scheduling service also has an RPC `Reset` (agent -> controller) which the agent calls on startup or when it has cleared its schedule [95]. Our agent simulation should call this when it (re)connects or if we manually reset a node. The controller, upon receiving Reset, should clear any pending entries for that agent and respond (the RPC returns empty). Also, the controller will treat that agent as fresh – possibly issuing a flurry of CreateEntryRequests to reinstall the current desired state. We should implement this to keep agent/controller in sync. (During simulation initialization, after NBI scenario is loaded and schedule is computed, we might simulate that agents join and then the controller sends initial schedules.)

- **Agent responses:** The agent side of the stream sends back a **Response** message for each request ID [96] [97]. The response includes a Status (Google RPC status proto) saying OK or error. We will simulate success normally (unless we intentionally simulate a failure like link not acquired). The controller can log these or update intent state accordingly. In our closed-loop simulation, we can also directly act on the entries (since we are effectively both controller and agent), but to remain true to API, we'll go through the motions.

In summary, the **Scheduler** component of our backend must: (a) Determine what configuration changes need to happen at what times (this comes from the network planning logic – see next section), (b) For each agent (node) involved, send the appropriate scheduled tasks via the Scheduling stream,

and © handle acknowledgments. Likewise, the agent logic must receive tasks and execute them in the simulated environment (e.g., at time T, turn on beam X; install route Y).

- **Telemetry Service:** To monitor network state, the simulation should implement the Telemetry API so that agents can report metrics to the controller. The Telemetry service in Spacetime has an `ExportMetrics` RPC where the agent pushes a batch of metrics (interface metrics, modem metrics, etc.) [98] . We will do similarly: each agent (or our simulation on each node's behalf) will periodically (e.g., every simulated second or a user-defined interval) create an ExportMetricsRequest and send it. This contains:
- **InterfaceMetrics:** metrics per network interface (each identified by interface ID) [99] . We should at least populate the basics:
  - Operational status (up/down) over time [100] [101] . We can toggle this if a link goes out or an interface is impaired.
  - Standard stats counters (bytes/packets in/out, errors, drops, etc.) [102] [103] . If we simulate traffic flows (perhaps by generating dummy traffic for provisioned flows), we could increment these accordingly. Initially, we might just report that links are idle (zero traffic) unless we have a traffic generation component. But we should still implement the fields for completeness. For example, if a link is carrying a 5 Mbps flow for 60 seconds, the agent could report ~37.5 MB in `tx_bytes` on the satellite's interface over that interval. This would validate that our provisioning delivered data. We can simulate volume by integrating the allocated bandwidth over time.
- **ModemMetrics:** possibly things like signal quality, modulation used, etc., for each transceiver. The `telemetry.proto` likely defines these later (not shown in snippet). We may include, for example, current modulation/coding in use, measured SNR, etc. If our link budget engine computes an SNR, we can forward that as a metric. If not, we might skip modem metrics initially.

The Telemetry RPC is client->server (agent calls it). So our agent simulation will call ExportMetrics with the data and the controller just returns an empty ack. The controller (or any NBI client) can also retrieve telemetry via the NBI (perhaps through querying Telemetry data or via subscription). Our open backend could simply log or store the last metrics. Optionally, we could implement a minimal pub-sub or query interface for telemetry (not strictly required by Spacetime API, but useful for users).

In essence, **real-time operation** of the simulation will involve a loop or event system where: time advances, events trigger (like a satellite comes into view or a scheduled beam turns on), the agent marks links up and begins counting traffic, telemetry is sent periodically, etc. This will mimic a live network. However, we also likely want the ability to **run simulations faster-than-real-time** for planning. In that case, we might not use the gRPC streaming in wall-clock real scheduling, but rather simulate the exchange logically. We might consider a mode where the simulation engine computes the outcomes without actual asynchronous streams, then just finalizes results (i.e., a batch simulation mode). The requirements list focuses on what needs to be implemented to mirror Spacetime's capabilities; whether we run in real-time or offline mode can be a configurable detail.

Network Planning and Orchestration Logic

With the scenario defined (nodes, links, demands) and the mechanism to push configs in place, the core intelligence needed is the **orchestration logic** that decides *how to route and schedule* the service requests on the available network. In Spacetime, this logic is encapsulated in various components (Topology & Routing app, Scheduling algorithms, Solvers for beam hopping, etc.). For our backend, we need to provide at least a basic implementation of these functions, and outline the requirement to eventually support more complex algorithms or external integration.

Route computation: Given the network topology (which interfaces can potentially connect to which via beams or fixed links), we need to find paths for each ServiceRequest. This is analogous to running a routing algorithm on a time-varying graph. A straightforward requirement is to use shortest-path or max-flow algorithms to select a route that meets the bandwidth requirement. Since the topology can change over time (due to satellite movement and intermittent connectivity), routes might also change over time. There are a few approaches: - Compute a fixed path (sequence of nodes/interfaces) that is used whenever connectivity allows, and if a link in the path is down, buffer traffic (for DTN flows) or mark the service as down for that period. - Compute time-specific routes (e.g. at each time segment when satellite A is over ground station B, route via that, then switch when it moves). This is more complex but yields better continuity for non-DTN flows (akin to make-before-break handovers).

Spacetime's provisioning API supports point-to-point **SR-TE Policies** (Segment Routing Traffic Engineering) which likely encapsulate pre-planned routes through the network [104] [105]. They also have the concept of **Protection and Disjoint groups** for redundancy [106] [107]. While implementing full dynamic rerouting and protection is beyond initial scope, our backend should be structured to allow these features. Specifically: - We should design a Routing module that can take the current connectivity graph (which our link evaluator provides) and compute best routes for each active flow. We can start with a simple shortest-path (minimize hops or latency) or highest-throughput path selection. - If multiple service requests compete, incorporate the priority and possibly do some form of weighted fair share or one-by-one allocation (highest priority first, then next, etc.). This is essentially a scheduling problem combined with routing (multi-commodity flow). - For open-source, we might integrate existing solvers or libraries for network flow optimization (e.g. use ILP solvers, or simply heuristic algorithms). In absence of those, a greedy approach: pick a service request, find a route with available capacity (taking into account current beam availability schedules), reserve it, then do next.

Scheduling (time dimension): In a LEO constellation, the network changes with time, so scheduling which link is active when is crucial. For example, a satellite can only connect to one ground station at a time (if a single steerable antenna), so if it has two competing downlink demands, the scheduler might allocate time slots to each. Or if it has multiple antennas, it could serve both simultaneously on different beams. We need a **Scheduler service** (not the gRPC one, but an internal module) that, given the connectivity windows and flow requirements, produces a timeline of **beam activations and deactivations** and **routing changes** that fulfill as much demand as possible.

Requirements for the scheduling component: - **Visibility window calculation:** Determine the time intervals during which each potential link (satellite-to-satellite or satellite-to-ground) is *available*. This comes from orbit geometry. We can pre-compute contact passes between each satellite-ground pair (using e.g. SGP4 and line-of-sight check) and between satellite-satellite pairs. This yields a set of time intervals for each potential link where ACCESS_EXISTS vs NO_ACCESS [51]. These can feed into the scheduling solver as the pool of possible link opportunities. - **Conflict constraints:** A node's interfaces might have constraints: e.g., a satellite with one steerable dish cannot simultaneously downlink to two ground stations – so only one link from that interface can be active at a time. We must ensure the schedule doesn't violate these. Similarly, if a ground station has one antenna, it can only point to one satellite at a time. These constraints are naturally satisfied if we treat each wireless interface like a resource that can be occupied by at most one beam at a time (unless multi-beam capability is specified). - **Flow segmentation:** If a ServiceRequest is disruption-tolerant, we can split its data over multiple intervals (store and forward). If not, we ideally need an end-to-end path *concurrently* available to consider the flow served. So for non-DTN flows, we might restrict ourselves to finding a continuous path from source to destination existing at a given time, and schedule those time ranges for use. For DTN flows, we can plan a multi-hop store-forward: e.g., data goes up to satellite at 12:00, stored, satellite connects to ground at 12:30, delivers the data. This means the "flow" was not instant, but eventually completed by 12:30 – which is okay for DTN (and we would mark the service as provisioned over the

whole interval in a loose sense because data is not lost). This is complex to generalize; to start, we may just attempt to keep an end-to-end path always, and only if that's impossible we allow store-forward as a fallback. - **Output of scheduler:** The scheduler will output a set of **ScheduledControlUpdates** (which correspond to Intents in Spacetime terms [70]). These include beam commands (to turn on/off specific links at certain times) and route commands (to direct traffic through the appropriate sequence of nodes when links are up). For example, it might output: *At 10:00 UTC, turn on beam from Sat1 to GS1 and from UT1 to Sat1, and install route UT1->GS1 via Sat1; at 10:05, switch Sat1's downlink to GS2, update route accordingly*, etc. We will feed these into the SBI Scheduling stream as described. - **Solver extensibility:** The problem of scheduling and routing in a time-varying network can be formulated as an optimization problem (maximize sum of flow throughput, or satisfy all flows with minimum resource usage, etc.). We should design our backend so that one could plug in an external solver. For instance, one might use a MILP solver or constraint solver to maximize total throughput given our link windows. Aalyria's repository includes a `solver.proto` (likely for beam hopping optimization). We won't attempt to replicate their proprietary solver, but by matching the API, we could allow an external process to compute a schedule and feed it in. For now, we can implement a heuristic solver internally, but keep the interface modular. Possibly define an API like `ComputeSchedule(Scenario)` that either calls an internal algorithm or invokes an external tool.

- **Beam hopping:** A specific advanced scenario is when satellites rapidly switch beams among many terminals (beam hopping). The `ModemIntent` in NBI supports configuring a sequence of beams to hop through with a timeslot pattern [108]. If we needed to support that, our scheduler would output a periodic pattern for a single antenna cycling through beams. This is an edge case; initially we can ignore explicit beam hopping Intents, but it's noted in the data model. If eventually implementing, we'd treat it as splitting a time slot among multiple simultaneous demands in a round-robin, and program the modem accordingly (that would likely show up as a special scheduling entry type).

Time semantics: The system should handle **time uniformly**, probably in **GPS time** or UNIX epoch microseconds (Spacetime often uses microseconds since epoch internally [109]). The scheduling messages include both a `Timestamp` (absolute time) and a `time_gpst` (duration from GPS epoch) [110]. We should do the same. For simulation, if running accelerated, we might interpret these times in a relative sense. It might be easiest to treat *time 0* as the simulation start and manage our own timeline (especially if doing offline simulation). However, for API compatibility, we may still represent times as actual epoch timestamps (perhaps configurable epoch). A practical approach: have a simulation start time (say, 2025-01-01T00:00:00Z as epoch 0 for scenario), then all internal times are offset from that. We ensure to supply the GPST offset if needed (GPST vs UTC difference of leap seconds). This detail mostly matters if integrating with real systems; for an isolated simulation, relative times are fine as long as consistent.

Additionally, ensure the system can handle **time steps** smaller than typical network events. Some events may need sub-second precision (the protos use microsecond precision). For orbits, maybe recompute positions every, say, 1 second or so for accuracy.

Topology updates and re-planning: If the network changes beyond the known motion (e.g., an unexpected outage), our simulation should be able to react. For example, if a link impairment is introduced (maybe a ground station failure), the scheduler might need to reroute flows. We will include a basic reactive logic: if an active route fails (no connectivity), mark that service request as unprovisioned and possibly try an alternate route if available. Full automatic re-planning can be complex; for now, logging the event may suffice, or requiring manual scenario adjustment. But the framework should allow dynamic changes and another run of the solver mid-simulation.

External Protocol Integration and Open Standards

To maximize realism and interoperability, our backend should incorporate or at least be compatible with relevant protocols and data standards in the satellite networking domain. Below are key ones and how we'll address them:

- **Orbit Ephemerides (TLE / CCSDS OEM):** For orbit propagation, we will support ingesting satellite orbital data in standard formats. Two-Line Element sets (TLE) are widely used for LEO satellites; we'll utilize them via SGP4 propagation. Additionally, for more general use, we could accept CCSDS OEM (Orbit Ephemeris Messages) or propagators like SPICE kernels. The aim is to allow users to plug in real satellite ephemerides easily. The PlatformDefinition's `MotionSource` already hints at Space-Track for TLEs; we'll mirror that by providing a utility to update platform coordinates from live TLE data.

- **Time standards:** As mentioned, using GPS time or standard epoch times, making sure to account for leap seconds (GPST is continuous, 18s offset from UTC currently). We cite that Spacetime explicitly references GPST [111]. Our system should maintain correct conversions if needed, though in a closed sim environment this might not be visible externally.

- **CCSDS Communication Protocols:** The CCSDS suite covers everything from link layer (TM/TC space link protocols) to application (CFDP, BP). While our simulation does not need to implement these protocols on the wire, awareness is useful. For example, if simulating a space link, one might want to know the effective data rate after channel coding (which CCSDS defines for various coding schemes) or incorporate one-way light time for deep space. At least for LEO/GEO, light time is negligible (< 0.3s GEO RTT) compared to other delays, so we may ignore propagation delay except perhaps for GEO latency (~250ms one-way). We can note that if needed, the simulator could insert such delays into traffic. In terms of application protocols, if a user wanted to simulate file transfer via CFDP or DTN, our system would mostly just ensure the network can store and deliver data accordingly (since we're not implementing actual CFDP state machines). But we should emphasize that **disruption tolerance** is supported in our design (tie-in with BPv7 usage) [12].

- **Delay/Disruption Tolerant Networking (DTN):** We already incorporate DTN by virtue of the storage and is_disruption_tolerant flags. This aligns with the Bundle Protocol (RFC 5050, 9171). If needed, one could integrate an actual DTN stack on top of our simulation to send bundles; our job is to ensure that the network can hold and forward them. A possible future extension: simulate custody signals or bundle drop if storage is exceeded.

- **DVB-S2X / RCS2 and other waveforms:** The inclusion of DVB-S2X and DVB-RCS2 (return channel via TDMA) in the modem enums [38] [112] means our simulation should conceptually support both continuous links and TDMA bursty links. If a link is TDMA-based (like RCS2), the scheduling might involve timeslot structure. For simplicity, we may treat all links as continuously available during their window, with an equivalent bandwidth. But one might want to simulate frame structures – that's likely too granular for now. Instead, we can incorporate their performance characteristics: e.g., RCS2 (being TDMA uplink) might have a certain efficiency and slightly higher latency due to framing; we could reflect that in the latency if needed.

- **5G Non-Terrestrial Networks (NTN):** Satellite integration with 5G is emerging, involving protocols like **GTP** (GPRS Tunneling Protocol) to carry user traffic from a satellite into a 5G core network, and using the NR air interface with adaptations for satellite. While implementing a 5G

stack is outside scope, our simulation can still be relevant by allowing an "end node" to represent a 5G gNB (base station) on a satellite or an NTN terminal on ground. The traffic flowing might be encapsulated in GTP, but from the perspective of our network layer, it's IP packets either way. However, we could note requirements such as:

- If modelling a 5G backhaul via satellite, consider the impact of long RTT on the GTP/UDP tunnels and potentially the need for PEPs (Performance Enhancing Proxies). We won't simulate these explicitly, but leaving room for users to attach their own traffic simulators would be good.
- The **band profile** may correspond to 5G NR bands if using satellites as cell towers – but likely S2X covers physical layer for now.
- **Inter-satellite links and routing:** We should ensure the routing logic can handle multi-hop via satellites. Possibly integrate or compare with something like **DTN routing protocols** (contact graph routing) for dynamic links. In open-source, NASA's **ION** or **DTN2** could potentially be connected to our sim to test routing algorithms on the dynamic topology. As a requirement, our backend should allow exporting the predicted contact plan (list of ISL and downlink windows) so that external tools could use it (for example, a DTN planning tool could import the contact plan to plan routes).
- **Open Planning/Simulation Tools:** There are existing tools like GMAT (for orbit), STK (not open), and some academic simulators for network or scheduling. Our system could output scenario data in formats those understand (or vice versa). For instance, outputting satellite ephemeris and link events in a CSV or XML for external analysis is useful. Not a core requirement, but for extensibility.

Radio integration: If at some point we want to test with real or emulated radios, our backend's design should allow hooking into hardware-in-the-loop. For example, if one had software-defined radios representing a modem, the simulation could drive it (send it beam steering commands, etc.) and get real BER measurements. To be prepared, we keep the radio control interfaces clearly defined (like the control messages for beams and modems). We already have `RadioConfig` and modem config IDs in the BeamUpdate [113] [114]. We can thus imagine integrating with a modem simulator that understands those IDs. For now, these will be stubbed (we won't actually load modem firmware, etc.). Similarly, for **switching behavior**: if a node is an IP router, one could integrate Linux networking stack or a router simulator. Initially, we'll simulate routing internally (no actual packet forwarding), but an advanced user could connect our control-plane to a running router (for example, use the SetRoute commands to actually program routes on a kernel routing table and then generate traffic through it). Our requirements list that these possibilities be kept in mind: e.g., ensure the addressing scheme and interface definitions would allow binding to actual network interfaces if needed.

Federation and External Networks

While our primary focus is a single network simulation, Spacetime defines a **Federation API** for connecting multiple networks (east-west interface) [115]. This would allow, for example, our constellation network to interoperate with a partner's network by offering excess capacity or requesting resources when needed. Implementing full federation involves additional roles (provider and consumer) and possibly brokering service requests across networks. For open-source use, federation might not be immediately needed, but we should design so that it's not precluded. If `allow_partner_resources` in a service request is true [66], our scheduler might consider links that are flagged as partner-provided (if we had any). We could simulate a partner network as just another set of nodes/links with restricted

control (maybe read-only to us). Initially, we'll likely skip federation, but we mention it for completeness. Our backend might simply return "not implemented" or a stub for any Federation API calls.

Summary of Features vs. Stubs

Finally, we enumerate which complex features will be **fully implemented vs. stubbed** in the initial version of the simulation backend:

- **Implemented:**
- Core entity types (Platform, NetworkNode, NetworkInterface, ServiceRequest, etc.) with all essential fields and CRUD operations.
- Dynamic position updates for moving platforms (SGP4 propagation of TLEs, etc.).
- Visibility calculations and basic link budget evaluation for determining link availability and approximate capacity.
- Scheduling mechanism to activate/deactivate links (beams) and to configure routes according to a computed plan, delivered via SBI scheduling stream [81] [87].
- Telemetry reporting of interface status and usage counters via SBI telemetry RPC [98].
- Simple routing and flow allocation logic (e.g. pick one viable path, try to meet requested bandwidth).
- Handling of disruption-tolerant flows by allowing buffered delivery (simulate storage usage).
- Control-plane messaging (SBI) lifecycle: agent hello, resets, acknowledgments as per protocol.
- Basic failure handling: if a link or node fails (simulated via impairment or outside event), reflect that in telemetry and mark flows down.

- **Stubbed or Simplified for Now:**

- **Advanced scheduling optimization:** We will not initially have an optimal global scheduler. A heuristic (or even manual scheduling in scenarios) will be used. The integration point for a solver will be there, but perhaps not a fully automatic optimal solution.
- **Interference and spectrum reuse management:** We won't rigorously model interference between beams or manage frequency channels beyond avoiding obvious conflicts. All links might be assumed orthogonal in frequency unless scenario explicitly overlaps them, and even then we may not reduce throughput correctly.
- **Exact modem adaptation:** While we allow modem mode and modcod profiles, we will not simulate dynamic adaptation loop. A link will either operate at a fixed setting or we'll switch it manually in scenario steps.
- **Link latency & protocol effects:** We treat propagation delay simply (constant for a link type, maybe 5 ms for LEO ISL, 50-500ms for GEO, etc.). We won't simulate TCP or other transport behavior under latency, though we could note the latency for each hop. Similarly, we won't emulate GTP or other encapsulation overhead explicitly – traffic is abstract.
- **Traffic generation:** No actual user data packets will flow unless the user integrates an external traffic tool. Our metrics will be simulated based on allocated bandwidth, not by running IP packet sims (no NS-3 level detail internally).
- **Model API / NMTS Graph:** We won't fully implement the new NMTS graph-based Model API at first. We acknowledge it (entities and relationships structure [58]) and possibly use a simplified internal graph, but the external API might remain the older entity-centric calls for now. Migration to NMTS could be a future improvement, aligning with open-source NMTS library usage.
- **Federation:** Not implemented beyond config flags. We won't connect to external controllers or handle inter-network handshakes.

- **UI/Visualization hooks:** Out of scope here, but worth noting if open-sourced, providing output (like a live map of orbits and links) would be valuable. We assume textual or logging outputs for events in initial version.

In conclusion, this specification outlines a **comprehensive constellation simulation platform** that mirrors Aalyria Spacetime's key APIs and data models. It will allow users to define complex satellite network scenarios (with satellites, ground stations, links, and traffic demands), and the system will simulate the time-varying network state, orchestrate connectivity (via scheduled beam pointing and routing updates), and provide telemetry outputs. By adhering closely to the Spacetime API (NBI/SBI) and leveraging open standards for orbit and networking, this backend can serve as an open-source alternative or complement to Spacetime – suitable for research, development, or integration testing in the satellite networking field.

Sources: The requirements derived above reference the Aalyria Spacetime API definitions and documentation for accuracy. For example, the NBI entity definitions for network nodes and interfaces [2] [116], the SBI scheduling and telemetry protocols [81] [98], and support for DTN store-and-forward [12] and standardized modem modes [38] have all been incorporated. This ensures our simulator aligns with proven concepts from Spacetime's design while highlighting where open standards like TLE/SGP4 and CCSDS DTN apply. By implementing the above features, we will create a robust simulation backend ready to model a constellation and its network behavior in an open, extensible manner.

---

[1] [80] [115] GitHub - aalyria/api: The API for Spacetime, a temporospatial software defined networking platform.
https://github.com/aalyria/api

[2] [9] [58] [116] Network Node – Spacetime Knowledge Base
https://docs.spacetime.aalyria.com/api/nbi/entities/network-node/

[3] Entities – Spacetime Knowledge Base - Aalyria
https://docs.spacetime.aalyria.com/api/nbi/entities/

[4] [5] [6] [7] [8] [10] [11] [12] [14] [17] [18] [19] [20] [21] [22] [24] [25] [26] [27] [28] [52] network_element.proto
https://github.com/aalyria/api/blob/320219aaa62ccf75690e5633ccfbef77748ddd0b/api/nbi/v1alpha/resources/network_element.proto

[13] [61] [62] [63] [64] [65] [66] [67] service_request.proto
https://github.com/aalyria/api/blob/320219aaa62ccf75690e5633ccfbef77748ddd0b/api/nbi/v1alpha/resources/service_request.proto

[15] [78] [79] [104] [105] [106] [107] simulation.proto
https://github.com/aalyria/api/blob/320219aaa62ccf75690e5633ccfbef77748ddd0b/api/simulation/v1alpha/simulation.proto

[16] network.proto
https://github.com/juendi/api/blob/f0a7b6e483d504262b91a3b7e86e3a47737bb2dc/api/common/network.proto

[23] [49] README.md
https://github.com/aalyria/api/blob/320219aaa62ccf75690e5633ccfbef77748ddd0b/contrib/README.md

[29] [30] [31] [32] [33] [34] [35] [36] [37] [39] [40] [41] [42] [48] [51] [112] network_link.proto
https://github.com/juendi/api/blob/f0a7b6e483d504262b91a3b7e86e3a47737bb2dc/api/nbi/v1alpha/resources/network_link.proto

[38] [50] [60] [68] [69] [70] [71] [72] [73] [74] [75] [108] intent.proto
https://github.com/juendi/api/blob/f0a7b6e483d504262b91a3b7e86e3a47737bb2dc/api/nbi/v1alpha/resources/intent.proto

43 44 45 46 47 113 114 control_beam.proto

https://github.com/aalyria/api/blob/320219aaa62ccf75690e5633ccfbef77748ddd0b/api/common/control_beam.proto

53 54 55 56 57 bent_pipe.proto

https://github.com/aalyria/api/blob/320219aaa62ccf75690e5633ccfbef77748ddd0b/api/common/bent_pipe.proto

59 76 77 109 nbi.proto

https://github.com/aalyria/api/blob/320219aaa62ccf75690e5633ccfbef77748ddd0b/api/nbi/v1alpha/nbi.proto

81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 110 111 scheduling.proto

https://github.com/aalyria/api/blob/320219aaa62ccf75690e5633ccfbef77748ddd0b/api/scheduling/v1alpha/scheduling.proto

98 99 100 101 102 103 telemetry.proto

https://github.com/aalyria/api/blob/320219aaa62ccf75690e5633ccfbef77748ddd0b/api/telemetry/v1alpha/telemetry.proto