ChatGPT

Scope 2 Implementation Plan: Network Interfaces & Connectivity Evaluation

Overview & Scope Objectives

Scope 2 extends the constellation simulator's core (established in Scope 1) to model network interfaces on each node and evaluate connectivity in real-time. Building on the **Scope 1** foundation – which provided moving platforms, attached network nodes, and a time-stepped simulation loop – this phase introduces **NetworkInterface** objects for every network node and implements basic link modeling (wired and wireless) between interfaces. The goal is a simulator that represents each node's communication **interfaces** (antennas, radios, ports) and can determine which links are geometrically possible (line-of-sight) at any given time [1]. This will produce a dynamic **connectivity graph** that updates as satellites move, laying the groundwork for higher-layer networking logic in later scopes [2]. In summary, **Scope 2** will:

- Extend the data model to include multiple **NetworkInterface** entries per NetworkNode (wired and wireless interfaces) [3].
- Define **TransceiverModel** specifications for wireless interfaces (frequency bands, antenna parameters, power, sensitivity, etc.) [4] [5].
- Implement **wired link** support (e.g. fiber/Ethernet) with static always-on connectivity (fixed latency, no line-of-sight needed) [6] [7].
- Implement **wireless link** connectivity checks: compute line-of-sight availability for satellite-to-ground and inter-satellite links each simulation tick. Apply horizon angle constraints for ground stations and Earth occlusion checks for space-space links.
- Assume omni-directional coverage by default (no antenna pointing limits yet) and optionally enforce a max range if specified [8].
- Produce a rudimentary time-varying connectivity map (which links are **available** vs. not) that can be queried or logged. No actual traffic or scheduling is simulated yet – links are considered "up" if geometry allows, but **no control-plane actions** (beam steering or routing) are required in this scope.

This implementation plan details the new modules, data structures, interfaces, and algorithms to be added in Scope 2, and how they integrate with the Scope 1 architecture. We also outline the exhaustive tests (test-driven development) that will validate each part, and recommend libraries/protocols to use. The design emphasizes **extensibility** and alignment with Aalyria Spacetime's APIs and concepts, ensuring we maintain world-class architectural rigor as we expand the simulator.

Architectural Context & Extension of Scope 1

Scope 1 established the core simulation architecture: - **PlatformDefinition** (physical platform with orbital/or geographic motion) and **NetworkNode** (logical node attached to a platform or standalone) data models. - A central **Knowledge Base (KB)** as the state store for all entities (platforms, nodes, etc.), supporting thread-safe updates and queries. - A **Simulation Time Controller** (time-step loop) driving periodic updates (e.g. propagating satellite positions via SGP4). - **Orbit Propagation & Motion Models** (using an SGP4 library for orbital updates and supporting static or scripted motion for ground/air

nodes) [9] [10] . - Basic fields like node type tags, power budget, and storage capacity were included as placeholders (not enforced in Scope 1) [11] [12] .

Scope 2 builds directly on this foundation. The new network interface and link concepts will be integrated such that: - **Data Model**: NetworkNode is extended to contain a collection of NetworkInterfaces. We will augment the existing data classes (or database schema) with interface-related tables/fields. This preserves backward compatibility: nodes without interfaces (from Scope 1 scenarios) remain valid, and new scenarios can populate interfaces as needed. - **Knowledge Base**: Will maintain not just platform and node states, but also interface and link states. The KB's API will be extended (e.g. `KB.get_interfaces(node_id)`, `KB.get_link(link_id)` etc.) to retrieve interface details and link status. All additions will follow the concurrency and atomicity model set in Scope 1's KB design [13] – e.g. updates to link availability are done in a thread-safe manner at each tick, and can trigger notifications if a subscription mechanism exists (the subscription/pub-sub concept introduced in Scope 1 will be expanded to cover link state changes as well [14]). - **Simulation Loop Integration**: After the **time controller** advances simulation time and updates all platform positions (Scope 1 behavior), a new **Connectivity Evaluation step** will run. This step iterates over relevant interface pairs or predefined links and computes which links are currently **available** (geometrically achievable). This result is written to the Knowledge Base (updating each link's status). The time controller will maintain modularity by calling an interface of the connectivity module (e.g. `ConnectivityService.updateAllLinks(current_time)`), keeping physics propagation and connectivity logic decoupled. - **Module Structure**: We introduce a **Connectivity Service** component (or module) responsible for evaluating link conditions. This service encapsulates the geometry calculations and link state management, providing clean interfaces for the simulation loop to invoke. It uses data from PlatformDefinition (positions), NetworkInterface (types and parameters), and TransceiverModel (for any constraints) to decide link availability. By isolating this in a service, we uphold a **separation of concerns**: orbital updates (Scope 1) vs. connectivity computation (Scope 2). This also makes it easier to extend or optimize connectivity logic later (e.g. precomputing contact windows, adding interference or link budget calculations in future scopes) [15] .

Overall, Scope 2's additions are designed to be **additive** and modular. The core architecture (entities, KB, time loop) remains in place; we are plugging in new data structures and logic in a way that future scopes (which will add scheduling, traffic, etc.) can build upon without refactoring. All new classes and functions will be documented and structured for clarity, following the same coding standards and patterns established in Scope 1. This ensures consistency and a smooth transition for developers continuing from Scope 1 to implement Scope 2 features.

Data Model Enhancements

NetworkInterface Data Model

We introduce a **NetworkInterface** class (or struct) to represent each communication interface (port, antenna, radio, etc.) on a NetworkNode. Each NetworkNode can have 0 or more NetworkInterfaces [16] , allowing us to model multi-homed nodes (e.g. a satellite with multiple radios, or a ground station with both an RF antenna and a fiber backhaul). Key attributes of the NetworkInterface model:

- **Interface Identity**: Every NetworkInterface has a unique **interface_id** that is unique *within its parent node*, plus a human-readable name. The combination `{node_id, interface_id}` serves as a globally unique identifier for the interface [17] . For example, node "node_gw_1" might have interfaces "if0" (satellite dish) and "if1" (fiber port). We will maintain this composite ID for lookups and refer to interfaces in link definitions by this composite ID. In addition, we store

optional addressing info: MAC address (if applicable) and an IP address (IPv4/IPv6 in CIDR notation) for future networking use [17]. These fields allow integration with routing and flow simulation later – e.g. the IP addresses can be used by a future routing algorithm to identify subnets or classify traffic flows to interfaces.

- **Interface Type (Medium)**: Each interface is categorized as either **Wired** or **Wireless** (we model this via a oneof/enum field `medium_type`, or subclassing in an OOP language) [18]:

- **WiredInterface**: Represents a tethered link such as an Ethernet port, fiber optic cable, or any fixed point-to-point link that does not require propagation through free space. Key properties:

    - `max_data_rate` – the maximum bandwidth capacity in bits per second [19]. This is a static cap; it will be used later when simulating traffic to not exceed link capacity. For now, we store it and can use it to compute theoretical throughput or simply log it.
    - `associated_platform` (optional) – a reference to a Platform ID if this cable/port is associated with a physical platform or location [20]. This is mainly metadata for visualization (e.g., if modeling a subsea fiber, one might attach it to the two shore station platforms). In our simulation logic, this field has no effect on connectivity (wired links are assumed direct connections), but we store it for completeness and future use (e.g., export to visualization or mapping tools).
    - **Latency** – by default, we will assume a fixed propagation latency for wired links. We can allow a configuration per link or a global default (e.g. all terrestrial fiber links = 5 ms). This can be stored as a property (in the link or interface, see Link model below) and used when simulating packet delays in the future. In Scope 2, latency values are noted but since we aren't simulating actual packet flow, this will not affect any outcome; it's essentially informational or for later use.
    - **Availability** – Wired interfaces/links are assumed to be **always "up"** unless explicitly taken down. We do *not* compute line-of-sight for wired links [7]. In other words, if two nodes are connected by a wired link (like a fiber), we treat it as continuously available (100% connectivity) by default. The only exceptions could be if an *impairment* is specified (e.g. user simulating a fiber cut). We will incorporate a mechanism to mark interfaces or links as *failed* or down (operational impairment), but by default in Scope 2 no such events are generated – the link stays up indefinitely [7].

- **WirelessInterface**: Represents a radio transceiver (e.g. an RF antenna, optical comm terminal, etc.) that can form links through free-space propagation. Each Wireless interface will contain:

    - A reference to a **Transceiver Model** (see next section) that defines its RF/optical characteristics [21]. We store this as a `transceiver_model_id` or pointer, linking to a library of transceiver specifications. This allows multiple interfaces to share the same model (e.g. many satellites using a "generic LEO Ku-band antenna" model). In the Aalyria Spacetime context, transceiver hardware specs reside in an "SDN Store" database [22]; for our simulator, we will implement a simplified internal store or static list of models.
    - **Parameters from TransceiverModel** (copied or referenced): frequency band, antenna gain pattern, transmitter power, receiver sensitivity, polarization, etc. (detailed in TransceiverModel section below) [23]. Some of these may also be directly accessible via the interface for quick checks (e.g. frequency band might be needed to decide compatibility between two interfaces).
    - **Link budget status**: we will include fields to hold dynamic link metrics (signal-to-noise ratio, current data rate, etc.), though in Scope 2 these will mostly remain *unpopulated/stubbed*. We design the interface class with placeholders for link budget results so that in

Scope 3+ we can fill them. For example, a wireless interface could have a method `computeLinkBudget(other_interface)` or a field like `last_snr` or `link_quality` that the Connectivity Service can update once we implement link budget calculations. In this scope, if a link is line-of-sight, we might mark it as "available" but not calculate an SNR or throughput – those would come in later scopes. We do, however, plan to log distance or elevation angle for debug, which can later feed into link budget formulas.

- **Operational status**: A wireless interface can be **disabled or impaired**. We will support an `operational_state` or `impairment` flag/list (aligned with the Spacetime NBI's `operational_impairment` field for NetworkInterface) [24]. By default interfaces are enabled; an impairment (e.g. hardware failure) can be set via scenario or API to force the interface (and thus any link using it) to be down regardless of geometry. In Scope 2, we will not generate impairments spontaneously, but the user could specify one in scenario config for testing (or toggle via an API call if available). Our connectivity logic will honor this: if an interface is marked down, any link involving it will be treated as unavailable (even if line-of-sight is present) [25]. This gives us a manual override to simulate failures. We'll implement the impairment as a simple boolean or enum state on the interface (or an entry in a list of impairments as per the proto). A test will cover that toggling this state affects connectivity (see Testing section).

In implementing NetworkInterface, we will ensure it aligns with the Aalyria API structures. The **aalyria.api.nbi** protos define `NetworkInterface` with fields like id, name, medium (wired/wireless), and for wireless, nested `WirelessDevice` with transceiver and hardware chain IDs, etc. We will use similar field naming and typing so that integration or translation is straightforward. Notably, we include: interface `id` and `name`, `mac_address` (if needed), `ip_address`, an enum for medium type, and then either a `wired` struct or `wireless` struct. In code, this could be represented by subclasses or a union type. The **physical hardware chain IDs** (like port, antenna ID in NMTS) that Spacetime tracks [26] will be acknowledged but not deeply used – we may store them if provided, but our simulation doesn't detail internal hardware elements. The design will not preclude capturing those IDs, to stay extensible.

Additionally, the NetworkInterface will have methods or computed properties for convenience, e.g.: - `getGlobalId()` – returns a string or tuple of `{node_id, interface_id}`. - Possibly, `isWireless()` / `isWired()` helpers (or based on class type). - For Wireless: maybe a method to get its current position. Since a NetworkInterface itself is attached to a node (and hence a platform via the node), it doesn't have an independent position; it uses its platform's position. But if in future we allow positional offset (like an antenna on a large platform at a certain offset), we might incorporate that. In Scope 2, it's enough to assume interface position = node's platform position for geometry. We will note this assumption.

Multiple Interfaces per Node: Our data model and KB will be designed to handle nodes with more than one interface. For example, a **ground station node** might have one wireless interface (satellite dish) and one wired interface (backhaul fiber). We will store interfaces in a collection (list or map) within the NetworkNode object, keyed by interface_id. All interfaces on a node share the node's unique node_id namespace for interface IDs. We will implement utility functions like `node.addInterface(interface)` which registers a new interface and ensures no duplicate ID on that node. If a duplicate is attempted, it will throw an error or refuse to add (error handling as part of scenario validation). We will test that multiple interfaces can be added and retrieved correctly. This multi-interface support sets the stage for more complex networking in later scopes (e.g. a satellite acting as a bent-pipe relay might have one interface for user links and another for feeder link to a gateway).

TransceiverModel Specification

To support wireless interfaces, we define a **TransceiverModel** class (or data structure) that encapsulates the radio frequency (RF) or optical communication parameters of an interface. In Aalyria's architecture, transceiver models are stored in a central repository and referenced by ID [21]. We will implement a simplified version: a static registry of transceiver model definitions that can be loaded at startup or scenario load.

Each **TransceiverModel** will include at minimum the following fields (consistent with the requirements) [5]: - **Model ID and Name**: A unique identifier (string or GUID) for the model, and a descriptive name (e.g. "Generic Ku-Band SATCOM Antenna"). - **Frequency Band**: The operating frequency or band (e.g. "Ku band 12–14 GHz" or optical 1550 nm). This could be a simple label and/or numeric range. We might represent it as a center frequency plus bandwidth, or a set of discrete bands. For simplicity, we can store an approximate frequency range in Hz. This will be used to ensure that two wireless interfaces can only form a link if they share a common band (for now, we assume scenario designers only connect compatible interfaces, but we include the data for validation). In the future, frequency could affect propagation (e.g. atmospheric attenuation), but not in Scope 2. - **Antenna Gain Pattern**: An antenna gain profile or beamwidth. This can be represented in various ways – e.g., a simple **beamwidth angle** (half-power beamwidth) for an idealized directional antenna, or a gain vs. off-boresight angle table. Initially, we will **simplify to an omnidirectional or wide-beam assumption** for all models unless specified otherwise [8]. We may still include a field for beamwidth or gain pattern (perhaps as a reference to a file or a formula) to keep the model extensible. For example, we might have: - `beamwidth_deg` (e.g. 180° to indicate hemispherical coverage, or smaller if we want to simulate a steerable narrow beam). - A placeholder for an antenna gain pattern lookup (could be a list of gain vs angle or a formula). By default, our example models will use an isotropic pattern (0 dBi gain in all directions) or a fixed moderate gain with a wide beam (to avoid needing to simulate pointing). This means if line-of-sight exists geometrically, we assume the antenna can point and communicate (no narrow pointing constraints yet). - **Transmit Power / EIRP**: The emitted power. This can be given as EIRP (Effective Isotropic Radiated Power) which already factors antenna gain, or as a raw RF power plus separate antenna gain. We will likely store an **EIRP (dBW or dBm)** to simplify. E.g., a smallsat radio might have 30 dBw EIRP. We also store a **Receiver sensitivity/G/T**: e.g. noise figure or G/T (gain over noise temperature) which indicates how well the antenna can receive. These parameters are critical for link budget calculations (Scope 3), but we include them now in the model so that as soon as we implement link budget, the data is there [5]. For now, these values won't affect link availability – they will be used later to calculate if the link quality is sufficient. We will possibly use them in Scope 2 only to log an **estimated link margin** when a link is available, as an informational output. - **Supported Modulation/Coding Modes**: A list of supported waveform modes or modem standards (e.g. "DVB-S2X QPSK 1/2", "Custom Optical Mode1"). The Aalyria API enumerates some standard modes in RadioConfig (like DVB-S2X, DVB-RCS2, etc.) [27]. We will include a field (e.g. `supported_modes`) as a list of enums or strings representing these modes. Initially, this is not used in simulation logic, but it will allow us to select a modulation in the future to determine data rate. In Scope 2, we might simply record a default mode (e.g. assume highest rate mode) for each model and maybe an indicative max data rate (to use as link capacity). For example, if a transceiver model corresponds to a 100 MHz bandwidth QPSK link, we might say it supports up to 100 Mbps. This can double as the link's capacity if needed (similar to `max_data_rate` for wired). - **Polarization**: e.g. linear, circular, etc., if relevant. This likely won't play a role in our sim, but we note it for completeness (to avoid connecting incompatible polarizations if that were a factor; in most cases, we assume matching). - **Pointing/Steering limits**: We can have fields such as `max_pointing_angle` or `gimbal_limit` to indicate how far the antenna can steer from its boresight or if it's fixed. In Scope 2, we assume full hemispherical coverage (the antenna can cover all sky above the platform), so effectively no limit within 180° view. But we include a placeholder – for instance, if an optical terminal has a limited cone, or if a ground antenna cannot point below a certain

elevation (which we handle via min elevation on the ground side separately). We may treat ground station antenna elevation limits separately (see below). - **Link Establishment Time**: Some models might have a lock-on time (how quickly it can acquire a target). The requirements mention a `link_establishment_timeout` that should be respected by scheduling algorithms [28]. In Scope 2, we will **store** this field (in seconds) if provided, but we will not actively simulate timing delays for link acquisition. We simply assume if geometry allows, the link is considered available. This timeout will come into play in Scope 4 when we simulate beam scheduling (if a beam command is issued, it might need some lead time to lock on). For now, we keep the data to not lose sight of the feature. - **Hardware Chain IDs**: Spacetime's WirelessDevice includes fields like platform_id, antenna_id, modulator_id, etc. (references to the NMTS hardware graph) [26]. Our TransceiverModel might include analogous IDs or just ignore them. Since our simulation does not model hardware at that granularity, we will **ignore NMTS IDs** except to possibly store them if needed for API compliance. They won't affect simulation behavior.

Example Transceiver Models: We will define a few representative models to use in testing and scenarios: - *LEO_Satellite_KuBand*: Frequency ~12 GHz, beamwidth ~180° (omni), EIRP ~40 dBw, G/T ~–20 dB/K, supports QPSK/16QAM modes, polarization linear. Represents a generic low-gain omni antenna on a LEO satellite. Max range ~? (LEO to ground ~2000 km, we won't enforce range for RF beyond LoS). - *GroundStation_KuBand_Dish*: Frequency ~12 GHz (to pair with above), narrow beam (we might still set 180° for simplicity now, but note it's steerable), high gain (antenna gain ~30 dBi, so EIRP depends on power but overall maybe 60–70 dBw EIRP), G/T ~10 dB/K (larger dish), supports same modes. This represents a tracking dish on the ground. We'll give it a min elevation angle of e.g. 5°. - *Optical_ISL_Terminal*: Frequency ~193 THz (1550 nm optical), very narrow beam (say 0.1°), high data rate potential. We can use this to illustrate another interface type (though we won't fully simulate beam pointing; if used, we'd assume the scenario ensures alignment). This is mostly placeholder for future scope (Scope 5+ might incorporate optical). - *Fiber1Gbps*: This is actually a wired model, but we can include in a common registry for convenience: capacity 1 Gbps, fixed latency default 5 ms, etc. (This might belong in link config rather than transceiver, since transceiver is for wireless, but we might generalize the concept to "link model" for wired as well).

These models can be defined in a JSON or Python dictionary for easy loading. The simulator will load default models at startup (or scenario can include model definitions). The NetworkInterfaces in scenario will reference these by model_id. The design ensures that adding new models (community contributions, specific antenna patterns, etc.) is straightforward – one can extend the model list without altering core code, as long as the fields are consistent.

Network Link Modeling

In Scope 2, we introduce **NetworkLink** objects to represent connectivity between two interfaces. In Spacetime's NBI, links can be unidirectional or bidirectional [29]. We will support both, focusing on the minimal structure needed now (with geometry-based availability). The NetworkLink model will include:

- **Link ID and Type**: Each link will have a unique identifier and a type designator (unidirectional vs. bidirectional). We expect scenario input to specify links. For convenience, we might allow two notations:
- *Directional link*: specified with a source interface and a destination interface (one-way). We will model this with a `NetworkLink` class having `src_iface_id` and `dst_iface_id` fields (each being the global interface ID of each end) [30]. Directional links carry traffic one-way – e.g. a downlink from sat -> ground.
- *Bidirectional link*: a link that is logically two-way between the same pair of nodes. In implementation, we can either have a separate `BidirectionalLink` class containing two

`NetworkLink` (one for each direction), or we simply treat a bidirectional link as an umbrella that generates two internal directional links. The Aalyria proto has a BidirectionalLink message grouping two LinkEnds [31]. We will likely mirror that: our `BidirectionalLink` object will have `end_a_iface` and `end_b_iface`. For simulation purposes, we will *instantiate two directional links under the hood* (A->B and B->A) and tie them together (so they share common properties like availability – which in a symmetric environment will be the same in both directions for line-of-sight). This simplifies reuse of the connectivity logic (which can be written for a directional link). We'll ensure that if one end's geometry is blocked, both directions are marked down. Essentially, bidirectional = two directional links that *must* go up/down together in our simplified physical model (since line-of-sight is mutual).

- We assign link IDs accordingly, e.g. a bidirectional link might have an ID "link1" and internally we tag the directional components as "link1_AtoB" and "link1_BtoA" or similar. However, in output or API we may report just one link with state = available if either direction can transmit? Actually, since we assume symmetric line-of-sight, available is same both ways. We'll handle the internal representation carefully to avoid duplication in reports.

- **Endpoint references**: Each link references exactly two interface endpoints (even a directional link still implicitly involves two interfaces – one src, one dst). We will store references or IDs for these. Possibly we also store the parent node IDs for convenience (though we can derive from interface IDs). This makes it easy to query, e.g., "list all links involving node X".

- **Link Medium and Properties**: The properties of a link largely derive from the interfaces:

- If both endpoints are **wired** interfaces, the link is considered a **wired link**. For these, we may store a fixed latency and capacity. We can derive capacity as the min of the two interface's max_data_rate (assuming the link can't exceed either port's capability). We'll also allow the scenario to explicitly specify a link bandwidth or latency if needed (overriding defaults). But essentially, a wired link is up 100% unless manually taken down.

- If either endpoint is wireless (in practice, both should be wireless for a valid wireless link – we won't link a wireless directly to a wired without an intermediate node), then the link is a **wireless link** subject to line-of-sight and other RF conditions. For wireless links, we will store:

    - Frequency/band: We might define the link's operating frequency. Ideally, the link frequency should match the interfaces' frequency band. If the two interfaces have overlapping bands (e.g. both are Ku-band), we consider the link's frequency that band. If not, the link is not actually feasible; our code will check compatibility. We'll implement a **band compatibility check** on link creation: if the interfaces have totally disjoint frequency ranges, we will log a warning or error, since that link wouldn't work. (Alternatively, scenario simply wouldn't define such a link. We assume correct input but include a safeguard.)
    - Polarization or other radio config: not heavily used now, but stored if provided.
    - A placeholder for **Link budget parameters**: e.g. free-space path loss, current SNR, etc. In Scope 2, when a link is line-of-sight, we may compute the distance and possibly calculate a rudimentary path loss and SNR just for output. However, detailed link budget (including atmospheric loss, rain fade, etc.) is beyond this scope. We will mostly mark link "available" if geometry allows, and leave quality metrics for later. The architecture plan for later scopes indicates eventually computing link quality [32], so we keep fields ready (like a function to estimate received power from EIRP and distance).

- **Current status**: We maintain a state (e.g. `status` field) for the link, with values like *AVAILABLE/UNAVAILABLE* (or a boolean up/down). This state is updated each time step by the connectivity service. Because we aren't simulating the active switching of beams yet, "available" essentially means **potentially usable** (line-of-sight exists) [33]. We do not yet differentiate if the link is actively in use (that will come with scheduling in Scope 4). In future scopes, we might refine state to "idle/active" vs. "up/down", but for now **available = up** (the link *can* carry traffic).
- **Impairments**: Similar to interfaces, a link can also be explicitly impaired (down for maintenance, etc.). The Spacetime model allows scheduling downtimes. We will mirror that by allowing a link to have an `is_forced_down` flag or a time schedule of downtimes. In Scope 2, we won't implement time-scheduled impairments, but we will allow a simple manual toggle for testing. The effect is if a link is marked down due to impairment, we will override geometry and keep it down until the impairment is cleared [25].
- **Latency**: For wireless, propagation latency can be derived from distance (speed of light). We can calculate it if needed (roughly 3.33 µs per km). In LEO-to-ground (~2000 km) that's ~6–7 ms one-way. We will include this calculation in the connectivity service for completeness and possibly store it in the link's state (so that later when simulating traffic, we have an idea of propagation delay). If not used now, it can simply be logged.
- **Directional properties**: If the link is directional (say interface A transmitting, B receiving), we might attach the transmitting radio configuration to it (power, mode) as in the proto's `RadioConfig`. In our simulation's data model, since we aren't yet doing separate TX/RX simulation, we don't need to store separate configs per direction beyond what's in the TransceiverModels of each end. However, to align with the API, we note that each link direction could have distinct parameters (especially if one end is a transmitter with certain power and the other end's transmitter has different power for the opposite direction). We will ensure our model *could* support that – for example, by looking up each interface's transmit power when computing the link from that side. For now, a bidirectional link with symmetric hardware we assume roughly equal feasibility both ways (if not, one direction might have shorter range due to weaker power; but since we won't fail one side and not the other in Scope 2, this detail can be glossed over until link budgets are considered).

- **Link Creation & Storage**: We anticipate two ways links get into the system:

- **Scenario-defined Links**: The scenario description explicitly lists certain links (e.g. "satellite_A.if0 -> groundStation_X.if0" downlink, etc.). In this case, we create link objects for each definition. This is the primary method in Scope 2. We assume the user or scenario designer knows which potential links to include (based on network design). If a link is not defined, the simulator will not consider that pair of nodes as connected even if geometry is favorable. (This is important: we don't auto-connect every satellite to every ground station – that would be unrealistic. E.g., maybe only certain satellites have missions to talk to certain stations.)
- **Auto-generated Links**: We will *not* auto-generate all possible links in Scope 2, but we design the Connectivity Service such that it could handle dynamic link creation in the future. For example, a future optimization might be to generate a "potential link" for every pair of compatible wireless interfaces and then filter by geometry (similar to computing a complete visibility graph). But that could be many links and not all are needed. Instead, we stick to scenario-provided links (which likely represent the network topology the user is interested in). In future (Scope 5), an automated planner might add links (like create a new inter-satellite link on the fly), which our data model will allow via an API call to add a link. Our design will permit link addition at runtime (the KB can insert a new link safely and the connectivity check next tick will handle it).

We store all NetworkLink objects in the Knowledge Base's state. This could be in a global list of links, or indexed by node or interface for quick lookup (e.g. KB may have a map: interface_id -> list of outgoing link IDs, to quickly find what links need update when that interface moves). We likely will maintain both a global registry of links (for unique ID management and retrieval by ID) and adjacency lists (for performance in checking links – see Connectivity logic below). Each link knows its endpoints, but we may also want each interface to know which links it is part of. We will implement that relationship to optimize checks: e.g., store in each NetworkInterface a list of link IDs that originate or terminate at it.

Data Flow for Link Status: The link's status (available/unavailable) is **derived** data based on the positions of platforms (for wireless) or static assumptions (wired). We will *not* hard-code any static state for wireless links – they will be recomputed. However, we do initialize all links as "down" or "unknown" at time 0 and then run the first connectivity check to set those that are immediately available. (Alternatively, we run connectivity logic at t=0 right after scenario load to initialize states). We ensure that any API or query for a link's status will always reflect the last computed value for the current sim time.

Ground Station Horizon Limit: A particular parameter to mention is the **minimum elevation angle** for ground stations (and potentially other ground-based antennas). We incorporate this as follows: Each **WirelessInterface** that is ground-based can have a `min_elevation_deg` field (in the TransceiverModel or interface settings). This represents the lowest elevation above the horizon the antenna can or will use (often to avoid obstacles and atmospheric loss). If not specified, we default to 0° (meaning the antenna will attempt to track down to the horizon). Many real ground stations might use 5° or 10° as a limit. Our connectivity calculation will check the elevation angle of a satellite as seen from the ground and require it to be greater than min_elevation to declare the link available. We will include a default of, say, 5° for our GroundStation_KuBand model to be realistic, but scenarios can adjust this. This is a simple way to incorporate realistic coverage constraints.

Extensibility: The NetworkLink model is kept minimal in logic in Scope 2. We treat it largely as a data container for endpoints and status. Higher-level usage (like scheduling which link to activate) will come later. However, we include enough detail (frequency, capacity, etc.) so that when we implement those features we don't need to alter the data structures much. For instance, in Scope 4/5 when we schedule flows, we might mark certain links as *actively in use* vs. just available; our Link object could then get an attribute like `in_use_by_flow_id`. We won't add that now, but the object is there to be extended. Also, we keep link objects around even when not available, rather than creating/destroying them on the fly, so that we can accumulate statistics (e.g. how long was it available over a period) or easily generate contact time reports later. This design aligns with the idea of computing contact windows in the future by analyzing a link's availability over time [15].

Connectivity Evaluation Module

With the data model in place (interfaces on nodes, and defined links between interfaces), we develop the **Connectivity Service** to evaluate which links are up at a given time. This is the core functional addition of Scope 2. The service will implement geometry-based **line-of-sight (LoS)** checks and update link states accordingly, on each simulation tick or upon position changes. Key responsibilities and design decisions for this module:

Geometry Calculations for Wireless Links

For each wireless link (or each link involving wireless interfaces), we perform the following checks to determine if a valid line-of-sight path exists:

- **Satellite-to-Ground links**: We compute whether the satellite is above the ground station's horizon by checking the **elevation angle**. Using the satellite's current position (in Earth-centered coordinates) and the ground station's position (latitude, longitude, altitude on Earth), we calculate elevation as seen from the ground. If the elevation is greater than the ground interface's `min_elevation` requirement (e.g. > 5°), then the satellite is in view of the ground station. If below that angle (or below 0°, i.e. below horizon), the Earth blocks the line-of-sight. Geometrically, this can be determined by vector dot-products: we have Earth's radius (approx 6371 km) and the positions:
- Let **G** = position vector of the ground station in ECEF (Earth-Centered, Earth-Fixed coordinates), and **S** = position vector of the satellite in ECEF at the current time. We can determine the elevation angle by comparing **S - G** to the local vertical at G. One formula: the elevation angle `e` satisfies
  $\cos(90° - e) = \frac{(S - G) \cdot \hat{G}}{|S - G|}$,
  where $\hat{G}$ is the unit vector from Earth center to the ground station. We compute e and check if $e > \text{minElev}$.
- Alternatively, a simpler check: Determine if the line from ground to satellite intersects Earth. The condition for LoS is often given by: distance from Earth center to the line segment > Earth radius. We can compute the point on the line SG closest to Earth's center and check its distance. But using elevation angle is straightforward given we know ground coordinates.
- We will implement a utility function `compute_elevation(ground_position, sat_position)` that returns the elevation angle (in degrees). Then `if elevation >= min_elev_deg: link_available = True else False`.
- We will include Earth as a sphere of radius R (6371 km). Ground station altitude (if any) is added to R for position G. This yields slight differences if altitude is significant (e.g. a mountain top station), which we account for by using the actual G vector magnitude.

- Example: If a LEO satellite is overhead of the ground station, elevation ~90°. If it's near the horizon, elevation approaches 0°. Our min_elev prevents using near-horizon passes if configured.

- **Satellite-to-Satellite (Intersatellite) links**: We check if the direct line between the two spacecraft is unobstructed by Earth. For any two satellites A and B, both in orbit around Earth, the line-of-sight exists if and only if the **angle between their position vectors (from Earth center) is less than a certain threshold**. Specifically, if Earth's sphere doesn't intersect the line segment connecting A and B. A practical test:

- Let **A** and **B** be ECEF position vectors of the two satellites. We find the shortest distance from Earth center to the line AB. This can be computed by vector projection. The formula: find parameter t where the point on AB closest to the origin is $P = A + t (B - A)$. Solve $t = -\frac{A \cdot (B-A)}{|B-A|^2}$. If $0 \le t \le 1$, the closest point lies between A and B; compute distance $d = |P|$. If $d > R$, then no Earth obstruction (line passes sufficiently above Earth). If $d \le R$, then Earth intersects the line, blocking it. If $t < 0$ or >1, then one satellite is "behind" the other as seen from Earth, and one of them would be below the other's horizon.
- A simpler approach uses angles: Compute angle $\theta = \angle AOB$ where O is Earth center. Using dot product: $\cos\theta = \frac{A \cdot B}{|A||B|}$. If $\theta < \theta_{\text{max}}$ for LoS. What is θ_{max}? It's basically when the line just grazes Earth. If both satellites

are at altitudes such that their horizon half-angle is e.g. X°, then likely $\theta_{\text{max}} \approx$ (180° - 2X°). But since we can do the distance method, we'll prefer that for exactness.
- We implement `line_of_sight_space(A_pos, B_pos)` that returns true if Earth does not block. We use Earth radius R in that calc.
- We will test edge cases: if satellites are on opposite sides of Earth, $d=0 < R$, blocked. If they are very far (like GEO to GEO on opposite longitudes, likely blocked if they are ~120° apart or more).
- For simplicity, we assume no other obstacles (no large structures or other planets, etc.).

- If the link is between a satellite and another high-altitude platform (e.g. a balloon), we treat it the same – any two moving platforms with positions can use this check.

- **Ground-to-Ground wireless links**: Although not a primary focus (terrestrial wireless may be considered "wired" for our sim if it's a fixed microwave link), if we ever model a long-distance ground radio link (e.g. two distant ground stations via microwave), we could reuse the same Earth intersection check (since both are essentially at Earth's surface). However, we will likely treat fixed terrestrial links as wired (for now) due to complexity of terrain, etc. So we won't explicitly implement ground-ground LoS beyond noting that if needed, the Earth curvature formula can be applied.

- **Range Limit**: Optionally, if a transceiver model specifies a maximum range (e.g. an optical terminal might only work within 5,000 km), we will enforce that. After checking LoS, we compute the distance between the two platforms. If the distance exceeds either interface's max range specification, we mark the link unavailable [8]. For RF LEO/GEO cases, typically if LoS exists, range is not an issue since they can communicate over those distances with appropriate power. But we include this for completeness and for models like optical or low-power radios. The transceiver model could have a field `max_range_km` for this. By default (if not set), we assume no range limit except the horizon (omni coverage).

- **Antenna Field of View**: As mentioned, we assume **hemispherical coverage** for antennas by default (they can serve any target above the local horizon). We do not yet model narrow beam pointing or require explicit pointing angles. This means if satellite A and satellite B have LoS, we assume their antennas can align accordingly, even if in reality a satellite might need to rotate or have a limited field of regard. This is a simplification we document as an assumption. In the future, if a transceiver model had a limited field-of-view (say ±45° off nadir), we would incorporate that by checking the angle between the antenna's boresight (which might align with platform orientation) and the direction to the other satellite. Scope 2 will not implement this; we effectively treat all wireless interfaces as capable of connecting to any direction as long as geometry permits and multi-tasking limits (see below) are not exceeded.

The above checks yield a boolean decision for each potential link: *AVAILABLE* (if all conditions met) or *UNAVAILABLE* (if any condition fails). Specifically, for a wireless directional link from A->B, we require: - Interface A is not impaired and interface B is not impaired. - Interface A and B share a common frequency band (we'll assume yes in scenario). - If B is ground, A's platform is above B's horizon (and vice versa – but if A is satellite and B ground, vice versa is trivial since ground is basically at A's nadir if A sees it). - If both A and B are satellites, LoS unobstructed by Earth. - Range within limits (if set). - (No capacity or scheduling check in this scope).

If a link is bidirectional, we apply the same logic; effectively both directions meet or fail the criteria together, so we mark the link available only if mutual LoS.

We will implement efficient geometry calculations, possibly with the aid of a library: - We already use an SGP4 library to get satellite positions in ECEF (or we convert from TEME to ECEF). We will reuse those position outputs. - We may use **NumPy** for vector math in Python (for performance, computing dot products, etc.). - If needed, a library like `astropy` or `skyfield` could compute elevation angles, but given the straightforward math, we'll implement it directly to avoid external dependencies. - Earth parameters: use WGS-84 Earth radius for ECEF (mean radius 6371 km). If high accuracy needed, one could use ellipsoidal Earth model for elevation (slightly different horizon at high latitudes), but this complexity is probably unnecessary at this stage. We will note that we use a spherical Earth approximation for connectivity.

Connectivity Service Operation & Data Flow

The Connectivity Service (let's call it `ConnectivityService` class) will provide methods to evaluate and update link states. Its operation in the simulation loop is as follows:

1. **Initialization**: At simulator start or scenario load, the ConnectivityService is initialized with references to:
2. The KnowledgeBase (so it can access all nodes, interfaces, and links).
3. Possibly it pre-computes or caches some data: e.g. it might build adjacency lists mapping each moving platform to the links that depend on it (for performance). We can optimize by noting that only links involving at least one moving platform need continuous checking. If both endpoints are fixed (ground-ground fiber), it's always up and can be set once. If one endpoint moves, check each tick. So we might categorize links into static (no moving parts), semi-static (one side fixed, one moving), and fully dynamic (both moving). In Scope 2, given typical scenarios, the majority will be satellite-to-ground (semi-static) or sat-to-sat (dynamic). This caching is an internal optimization to avoid iterating over absolutely all links if not needed.

4. If desired, pre-fetch any constant values (like Earth radius, etc.) or references to transceiver models for quick access.

5. **Update on Time Tick**: Each simulation **tick** (e.g. each second, or whatever step chosen):

6. The Simulation Time Controller calls `ConnectivityService.updateConnectivity(current_time)` after updating all platform positions for that time.

7. The ConnectivityService will iterate through all links (or only those needing eval). For each link:

    - If link is **wired**: We do not need geometry check. If it's marked as operational (no impairment), we directly set status = **AVAILABLE** (unless we plan to simulate manual downtime – if an impairment flag is set on this link or its interfaces, we set UNAVAILABLE). Since wired links state rarely changes (except if someone toggles it off), we might even skip re-evaluating it every tick. We can set it once at init and only change if an impairment event occurs. But for simplicity, the update loop can check a wired link's impairment flag each time (which is negligible overhead given fewer wired links typically).
    - If link is **wireless**: Determine the two endpoint interfaces. Retrieve their platform positions:
    - If either endpoint's node has **no platform** (meaning a non-physical node), then this wireless link's geometry cannot be computed (the node has no position). In practice, we should not have a wireless link involving a node with no platform – that scenario is invalid because a "floating" network node can't have a radio in nowhere. We will log an error if

this is encountered and mark link unavailable. (E.g., if someone tried to link an internet backbone node to a satellite directly with a wireless link but the internet node has no coordinates, we can't simulate that – they should have had a ground platform for the internet gateway or use a wired link).
    - Otherwise, we get coordinates for both endpoints: For a ground node, that's the fixed lat/long (we convert to ECEF vector G). For a satellite or airborne node, that's the propagated ECEF vector S from the Orbit Propagator (Scope 1's output).
    - We then execute the LoS check algorithms described above:
        - Compute elevation if one end is ground, etc.
        - Compute Earth blockage for space-space.
        - Check range if applicable.
    - Check interface and link impairment states:
        - If either interface is marked down (operational_impairment), immediately mark link UNAVAILABLE (we might short-circuit other checks).
        - If the link itself has an impairment flag (like scheduled downtime), mark it down.
    - Decide availability: if all geometry and operational criteria are satisfied, set link.status = AVAILABLE (True). Otherwise, set it UNAVAILABLE (False).
    - Optionally, for each wireless link that is available, compute secondary data:
    - The slant range (distance) between the two platforms. (We will definitely compute this as it's needed for future link budgets; it's easy to compute from vectors.)
    - Propagation delay (distance / c).
    - Possibly the current geometry angles (elevation for ground link, or azimuth if needed for advanced analysis, though not needed now).
    - These values can be stored in the link object for later use or just logged.
    - We might also compute an approximate receive power or SNR just to have an idea (using a simple free-space path loss formula and the stored EIRP/G/T). This is not required by Scope 2, but including a *log output* like "Link sat1->gs1 available, range=1200 km, est. SNR = 10 dB" could be useful for debugging. We will implement a function for link budget if time permits, or simply note that it's TBD.
    - After processing, if the new status is different from the last status, we can trigger an event:
    - e.g., Log an info: "[Time 123] Link sat1->gs1 became AVAILABLE".
    - If a subscription mechanism exists (perhaps a future feature for telemetry), we would notify subscribers of link state change. (We have the structure in KB to potentially allow state subscriptions since Scope 1 considered it [14]).
    - We ensure the status update is done atomically to the KB (if multi-threaded, lock around it, but likely single-thread simulation loop, so fine).

8. Complexity: If we have N links, this loop is O(N) each tick. This is acceptable for the scale envisioned (even a large constellation might have hundreds or a few thousand possible links defined). If we had to consider every possible sat-to-sat pair it could be huge (O(n^2) for n satellites), but again, we assume scenario prunes unneeded links. In the future, Scope 5's contact plan precomputation could optimize this by calculating windows offline rather than per tick. But for now, the live per-tick check is straightforward and manageable.

9. **Data Update & Access**: The KnowledgeBase is updated with each link's new status. The link objects themselves may reside in the KB, so updating their `status` field is inherently updating the KB's state. If we maintain any separate indexing (like an "adjacency matrix" or graph representation), we update that too. For example, we might have an in-memory graph structure where nodes connect if link available. But we can generate that on the fly when needed. In

Scope 2, it suffices that if someone queries the link object from KB, they see the `status` field reflecting availability. We will implement KB query functions such as:

10. `get_link_status(link_id)` -> returns available/unavailable.
11. `get_node_connectivity(node_id)` -> returns list of other nodes that node_id currently has an available link to. This could be derived by checking all links involving the interfaces of that node. We can implement this to aid debugging (e.g. for a given time, get connectivity adjacency).

12. These queries operate on the updated data after each tick.

13. **Post-Update Actions**: Once link statuses are updated, the simulation loop might proceed to other tasks. In Scope 2, there's not much else (no routing or traffic yet). We might simply output telemetry. We will include the possibility of generating a **"connectivity snapshot"** at intervals. For example, after each tick or every N ticks, the simulator could output a summary of all active links. This is useful for verification and later for creating contact plans. We will implement a simple logger or printout when running a scenario in verbose mode to list current active links.

Handling Multiple Simultaneous Links and Constraints

An important aspect to clarify: in reality, a single wireless interface (antenna) cannot usually support multiple simultaneous independent links unless it has multi-beam capability. In Scope 2, we **do not enforce** this constraint – we treat each potential link independently. For instance, if a ground station has one antenna and two satellites are above the horizon, our simulation will mark both satellite links as "available". This means *potentially* both could be used, even though physically that might require two antennas. The reason we allow this is that scheduling which link to actually use at a time is a Scope 4 concern (the **scheduler** will later ensure only one is active if only one beam is available). We will, however, prepare for this by noting: - We can include a property like `max_beams` in the TransceiverModel (e.g. a phased array could have `max_beams = 2` meaning it can form two simultaneous links). For a typical dish, `max_beams = 1`. In Scope 2, we will store this property (default 1) but **not act on it**. The connectivity service will not enforce it (it won't arbitrarily mark one link down because another is up – that's a scheduling decision). - We will add documentation in code comments that if multiple links share an interface and are simultaneously geometrically available, they will all be flagged available. It is up to higher layers to decide which ones to actually activate. We will test a scenario where one ground station interface has two satellites in view, to ensure both links show as available, and note that this is an intentional simplification. - In later scopes, when scheduling is introduced, we will refine link state to possibly indicate "available but not active due to concurrency limits" vs. "actively used". For now, only geometry and manual downs control availability.

Example Algorithm Summary (Pseudo-code)

To illustrate, here is a pseudo-code of the connectivity update logic for one tick:

```
for link in all_links:
    if link.type == "wired":
        if link.impaired or link.interfaceA.impaired or
link.interfaceB.impaired:
            link.status = UNAVAILABLE
        else:
            link.status = AVAILABLE   # always on
        continue
```

```
    # wireless link
    A = link.interfaceA
    B = link.interfaceB
    if A.impaired or B.impaired or link.impaired:
        link.status = UNAVAILABLE
        continue

    posA = A.node.platform.position_ecef  # get ECEF coordinates
    posB = B.node.platform.position_ecef
    available = True
    # Ground horizon check
    if A.node.platform.is_ground and B.node.platform.is_space:
        elev = compute_elevation(A.node.platform, posB)
        if elev < A.min_elevation_deg: available = False
    if B.node.platform.is_ground and A.node.platform.is_space:
        elev = compute_elevation(B.node.platform, posA)
        if elev < B.min_elevation_deg: available = False

    # Earth occlusion check for space-space
    if A.node.platform.is_space and B.node.platform.is_space:
        if not line_of_sight_space(posA, posB): available = False

    # Frequency compatibility (just in case)
    if A.medium == "wireless" and B.medium == "wireless":
        if A.transceiver.freq_band != B.transceiver.freq_band:
            # (Better: check overlap)
            available = False

    # Range check
    distance = norm(posB - posA)
    if A.transceiver.max_range and distance > A.transceiver.max_range:
available = False
    if B.transceiver.max_range and distance > B.transceiver.max_range:
available = False

    # Set link status based on available flag
    link.status = AVAILABLE if available else UNAVAILABLE

    # If available, compute latency and maybe log quality
    if link.status == AVAILABLE:
        link.propagation_delay = distance / C  # speed of light
        log(f"{link.id} up: range={distance:.0f}km,
delay={link.propagation_delay*1000:.2f}ms")
    else:
        log(f"{link.id} down (geometry/impairment)")
```

(This pseudo-code assumes certain helpers and attributes. In actual implementation, we have to handle platform references carefully, and use actual Earth radius in compute_elevation, etc.)

The Connectivity Service will be unit-tested thoroughly (see Testing section) to ensure each logic branch works (horizon, occlusion, etc.).

Finally, note that **wired links** do not require such calculations, which simplifies things. They effectively behave as static edges in the connectivity graph (unless toggled off). **Wireless links** become time-varying edges.

Knowledge Base & API Considerations

After each update, the KnowledgeBase holds the new connectivity state. If the simulator provides an API (likely in Scope 3 via the Northbound Interface), it could allow queries like "get current network graph" or subscribe to link status events. We are laying the foundation for that: - We ensure that the data structures (NetworkNode with interfaces, links connecting interfaces) mirror the Spacetime API data model. E.g., a client using NBI might create a NetworkLink resource with given endpoints; our simulator would instantiate a corresponding NetworkLink object. Likewise, when our simulator updates link status, it might emit a Simulation Telemetry event (via SBI maybe) about link state changes. For now, we can simulate this by simple logging, but we keep the door open for real API hooks. - The internal representation will make it easy to produce a **contact plan** if needed – essentially a list of time intervals when each link is up. In Scope 2, we do not precompute those intervals, but we could output a log as the sim runs. By Scope 5, we plan to actually precompute and store these windows for planning algorithms [34] . Our design of keeping links persistent and updating a boolean each tick is compatible with that future improvement (we could run a separate routine to calculate future availability times using orbital predictions). - The **topology graph** is essentially formed by the nodes and links in the KB [35] . We treat the collection of all nodes, interfaces, and links as the current scenario's network topology. In Scope 2, this graph is dynamic but only due to movement; the set of nodes and links is static (no additions/deletions in runtime). We will implement functions to **export this graph** state, e.g. for debugging or visualization. For instance, output a JSON or Graphviz dot file of all nodes and which nodes have active links at a given time. This can help validate that the simulator correctly identifies connectivity (we might compare with an external STK scenario or known contact times).

In summary, the Connectivity Evaluation module continuously updates the **network connectivity graph** in the simulation. After Scope 2 implementation, the simulator will be capable of answering queries like: *"At time T, which network nodes have direct communication links, and which are isolated?"* This is a fundamental step toward later simulating routing and traffic.

Implementation Modules & Interfaces

We break down the Scope 2 implementation into distinct components/modules, each with clearly defined interfaces (methods, data flow) for integration. Below is the module breakdown and the plan for each:

- **Data Model Modules**:
- *NetworkInterface Module*: Defines the NetworkInterface class and related enums (MediumType: Wired/Wireless). Provides methods to create interfaces, validate attributes, and possibly update interface state (e.g. setting impairment). Integrates with NetworkNode (each NetworkNode will have a list of NetworkInterfaces). Will include serialization/deserialization functions if needed (e.g. from a scenario config file or to output state via API).
- *TransceiverModel Library*: Could be a simple static class or even just a JSON file + loader that populates a dictionary of TransceiverModel objects. Provides lookup by model_id. This module

may also contain utility methods, like computing antenna gain at a given angle (though initially not used, but stubbed for future). It will reside perhaps in a "config" or "models" package.

- *NetworkLink Module*: Defines the NetworkLink and BidirectionalLink classes. Provides factory methods to create directional or bidirectional links given interface references. Contains logic for linking two interfaces (with optional radio config per direction). Also includes any functions to serialize a link (e.g. to JSON or proto) and to apply impairment state. Since link status is dynamic, the link object mainly holds static info plus a mutable status field. We will ensure **thread-safe update** of that status (likely by design, all updates happen in the single simulation thread, so it's safe; if not, we'll add locking around status).

- All these data model definitions will be placed in appropriate namespaces (e.g. `simulator.core.entities` for interfaces and links, complementing PlatformDefinition and NetworkNode classes from Scope 1). Documentation strings will reference the corresponding sections of the requirements (to maintain traceability, e.g. "NetworkInterface corresponds to Spacetime NetworkInterface NBI resource [36] ").

- **Knowledge Base Integration**:
  The KnowledgeBase (perhaps implemented as a singleton, or passed around as needed) will gain:

- Storage structures: e.g. `self.network_interfaces` (if we want a global registry, but since interfaces are contained in nodes, we might not need a separate global list), and `self.network_links` (dictionary of link_id -> NetworkLink). The KB's existing structures for nodes will be leveraged: each NetworkNode object can contain its interfaces, so accessing interfaces is typically via the node (`node.interfaces`).
- Methods to add and retrieve interfaces/links: e.g. `KB.add_interface(node_id, interface_obj)`, `KB.add_link(link_obj)`, `KB.get_link(link_id)`, `KB.list_links()` etc. These allow other components (like scenario loader or connectivity service) to manipulate the KB.
- Possibly, an index for quick lookup of links by interface: e.g. `KB.links_by_interface[iface_global_id] -> [link_ids...]`. We will update this index on link creation. The Connectivity Service can use it if we optimize per-interface updates (though our initial approach is simply looping through all links, which is fine).

- Concurrency: If the simulation runs in one thread and API calls in another, KB must lock when reading/updating these structures. We will use the same concurrency mechanism from Scope 1 (perhaps a ReadWrite lock or simple mutex around the KB state) [13] . In Scope 2, since external API use is minimal, we might not fully exercise concurrent writes, but we design for it. For example, if an API call attempted to toggle an interface impairment while the simulation tick is running, we need to handle that safely (e.g. lock or queue the change). We might decide to allow impairments only at tick boundaries to simplify. We will mention in documentation that simultaneous modifications are controlled.

- **ConnectivityService Module**:
  This is the core computational module described earlier. It will likely include:

- **initialize()** – perhaps taking the KB or lists of links and interfaces. It might precompute static lists as discussed.

- **update_all_links(current_time)** – performs the iterative check for each link and updates statuses. It uses helper functions for geometry:
    - `compute_elevation(ground_platform, target_position)` – returns elevation angle. (We know ground_platform's lat/long; we can compute its ECEF vector and then do the math with target_position).
    - `line_of_sight_space(posA, posB)` – Earth occlusion check.
    - Possibly `distance(posA, posB)` – though trivial.
    - We might encapsulate these in a submodule like **GeometryUtils** for cleanliness, as they might also be used in other contexts (e.g. a future module computing contact windows or link budgets can reuse).
- The update function will clearly separate the wired vs wireless handling as outlined. It will set the status on the link objects directly. To avoid repeating checks, it may also incorporate interface-specific data:
    - We might enhance efficiency by computing some per-interface metrics each tick: e.g. which ground stations see which satellites. However, since we already iterate links, that might not be needed now.
- If performance allows, we might double-check consistency: e.g., if link A->B is set available, then if the reverse link object exists (B->A), it should also be set available. We ensure our logic either handles both together (for bidirectional) or we explicitly mirror the result. This avoids any case where one direction says up and the other says down (which physically shouldn't happen under our assumptions).

- Logging: The service will use the simulator's logging framework to output debug info about link decisions. We will likely allow a verbosity level where every link change is logged.

- **Scenario Loader / Example Setup**:
  While not a separate runtime module, writing the Scope 2 plan includes specifying how scenarios are defined now that interfaces and links exist:

- We will update the **scenario file format or builder** to allow defining interfaces under each node and to list links. For example, if using JSON, a node entry might look like:

```
{
  "node_id": "node_gw_1",
  "platform_id": "gs1",
  "...": "...",
  "interfaces": [
     { "id": "if0", "name": "GW-Ku", "type": "WIRELESS",
"transceiver_model": "GroundStation_KuBand", "ipv4": "10.0.0.1/24" },
     { "id": "if1", "name": "GW-Fiber", "type": "WIRED",
"max_data_rate": 1e9 }
  ]
}
```

And a link entry might be:

```
{ "link_id": "link1", "type": "BIDIRECTIONAL", "interface_a":
"node_leo_1:if0", "interface_b": "node_gw_1:if0" }
```

We will design accordingly. If scenario input was previously perhaps Python code or a small config, we'll extend it similarly.

- The scenario loader will create NetworkInterface objects for each interface specified and attach to the corresponding NetworkNode in KB. Then it will create NetworkLink objects for each link specified. We will validate references (e.g. does `node_leo_1:if0` exist) and throw errors if not. The loader will also likely assign default transceiver models if not explicitly given (for ease, e.g. if a node is type "SATELLITE_NODE" and no interface specified, we might auto-add a default RF interface to it in Scope 1 sample scenarios, but now we require interfaces to be explicitly listed for clarity).

- Since Scope 1's sample scenario included nodes possibly with stub interfaces [37], we might retrofit that: if those stubs exist, we can now flesh them out with real models. Otherwise, we'll define them in the scenario data.

- **Northbound API Alignment**:
  Although full NBI support (gRPC endpoints, etc.) is likely in Scope 3, we ensure our Scope 2 implementation doesn't conflict with the API. For example:

- The NBI defines operations to add links, remove links. If we were implementing them now, our modules support those actions (e.g. `KB.add_link` can be called anytime).
- The NBI data structures (proto messages) map to our classes (we have 1:1 correspondence in naming fields like interface IDs, etc.). This way, when we implement the actual API calls, it's mostly translating incoming messages to constructing these objects and vice versa for queries.
- If time permits, we might implement a rudimentary API call in Scope 2 just for demonstration, such as a CLI or function to query connectivity (which mimics a `GetNetworkTopology` NBI call).

- The *Southbound Interface (SBI)*, which is for scheduling, will come later, so nothing to implement now, but we mention that link availability will eventually feed into SBI telemetry (the simulation will report available links to a scheduler or agent).

- **Utility Modules**:

- *GeometryUtils*: As mentioned, a small collection of earth geometry functions (we can put the Earth radius constant and perhaps WGS84 ellipsoid parameters here if needed in future).
- *Logging & Metrics*: We might extend the logging utility from Scope 1 to include categories for connectivity. We might also have a metric counter for "current number of available links" for monitoring. E.g., each tick compute count of links up and log that. This is not critical, but can be useful to ensure simulation is doing something (especially in large scenarios).
- *Time management*: Scope 1 had time progression (maybe a simulation clock class). We will use that to get current time for any time-based decisions. If we later allow time-based impairments (e.g. link down from t1 to t2), we would integrate there.

All interfaces between modules will be carefully documented. For instance, the ConnectivityService will be invoked by the Simulation Controller, and it will use the KnowledgeBase interface methods to get positions and update links. The KnowledgeBase in turn uses the data model classes. This decoupling ensures that each part can be unit tested in isolation (we can test ConnectivityService with a mocked KB containing sample data, etc.).

Testing Strategy (Test-Driven Development)

We will adopt a **Test-Driven Development (TDD)** approach for Scope 2: writing tests for each piece of functionality before implementation, ensuring we capture all requirements and edge cases. The tests will be organized by module and also include high-level integration scenario tests. Below is an outline of exhaustive tests to cover the new Scope 2 features:

Unit Tests

1. NetworkInterface & TransceiverModel Tests
- *Creation and Attribute Assignment*: Create sample NetworkInterface objects (wired and wireless) and verify their fields. For wired, set max_data_rate and platform association and ensure getters return correct values. For wireless, assign a TransceiverModel (use a dummy model object) and verify the link is established. Ensure that the composite global ID is correctly formatted (e.g. `node_id:interface_id`). This might be a method like `interface.get_global_id()` that we test.
- *Unique ID Enforcement*: Attempt to add two interfaces with the same ID to one node – the second addition should raise an exception or error. Test that the KnowledgeBase or Node class rejects duplicate interface IDs (to maintain uniqueness).
- *Operational Impairment Flag*: Set an interface's impairment flag and ensure it can be read. Possibly call a helper like `interface.is_operational()` that returns False if impaired. In a standalone test, we can simulate toggling the flag (e.g. from None to "FAILED") and verify that it changes the interface's internal state. We'll later see effect in integration, but here at unit level just test the property works as intended.
- *TransceiverModel properties*: Construct a TransceiverModel for a known configuration (e.g. an example Ku-band model). Verify that all fields (frequency, beamwidth, power, etc.) are accessible and correct. If the model has any method (say to compute free-space loss or check frequency overlap), test those with known inputs. For example, if we implement a method `model.supports_frequency(freq)`, test that a frequency in band returns True and out of band False. Similarly, if we have a gain pattern placeholder (maybe we define an isotropic model returns gain 0 dBi for any angle), test that.
- *Default Values*: Ensure that for unspecified optional fields (like polarization or max_range), the model uses intended defaults (e.g. polarization default "RHCP" or None means no polarization restrictions, max_range default = infinity). Write tests that if we don't set max_range, our geometry logic treats it as no limit (we might indirectly test this via connectivity tests, but can also test a utility function that interprets `None` as infinite).

2. NetworkLink Tests
- *Directional Link Setup*: Create two dummy NetworkInterface objects (e.g. sat_if and ground_if) and then create a directional NetworkLink linking them (sat_if as src, ground_if as dst). Verify that the link stores the correct interface references and that its initial status is default (likely UNAVAILABLE until we evaluate it). We might manually call the geometry check functions with positions we choose to simulate availability (or easier: monkey-patch a position and call the connectivity update). At unit level, this might be too integration-y; instead, we test structural aspects: - The link's `src_iface_id` and `dst_iface_id` match the ones we passed. - If we call a method like `link.get_endpoints()`, it returns the interface objects or IDs properly. - If we mark the link impaired (`link.impaired = True` or via a method), verify it stores that state. - *Bidirectional Link Grouping*: Create a bidirectional link between two interfaces. Our API might create one BidirectionalLink object that in turn yields two NetworkLink instances. We test that: - The BidirectionalLink contains references to two directional links. - Each of those directional links has endpoints A->B and B->A correctly. - They possibly share an ID or have related IDs. We might decide that the BidirectionalLink's ID is a high-level identifier and each

directional link has an internal id like `link2_A` and `link2_B`. Test that naming scheme if we implement it. - Ensure the relationship is symmetric: e.g. calling a function `bidilink.get_link_A_to_B()` returns one of the internal links and vice versa for B->A. - *Frequency Compatibility Check*: If we implement a method on the link or a static function to validate that two interfaces can link (checking frequency band overlap), unit test it. For example, create two transceiver models: - Model X: frequency band [10–12 GHz]. - Model Y: frequency band [13–14 GHz]. Assign X to one interface and Y to another. Attempt to create a link. The validation should flag incompatibility. We might design `NetworkLink.create(ifaceA, ifaceB)` to raise an error if bands don't overlap. Test that this error is indeed raised. Then test a compatible case (both in 10–12 GHz) succeeds. - *Max Range in Model vs. Link*: If we allow a transceiver model to specify `max_range_km`, test that the model value is correctly recognized. Perhaps use a small max_range for a test model (e.g. 1000 km), then position two interfaces at >1000 km apart and see that connectivity logic marks the link down (the integration test will cover it, but we can test the function that checks range: feed it distance and model range and ensure it returns false if beyond). - *Latency and Capacity Assignment*: For a wired link, set a specific latency (say 10 ms) in scenario. Our link object might have a field for latency. Test that after creation, `link.latency` = 0.01 s (converted properly). Similarly, set wired interfaces with different max_data_rate (e.g. 1 Gbps and 500 Mbps) and ensure the link's effective capacity is min(1 Gbps, 500 Mbps) = 500 Mbps if we decide to compute that. If we don't auto-compute, at least test that the values are accessible. Possibly implement a method `link.get_capacity()` that does the min logic on the fly and test it.
- *Link Impairment*: Set a link's impairment flag and ensure that the connectivity service (when tested later) respects it. This might be tested at integration level primarily, but we can also directly set `link.impaired=True` and verify that `link.status` is set to UNAVAILABLE after an update (requires connectivity run or manual logic invocation, so probably integration).

3. Connectivity Geometry Utility Tests
- We will develop precise tests for the geometric algorithms: - *Elevation Angle Calculation*: We can craft a scenario: * Ground station at latitude 0°, longitude 0°, altitude 0 (so ECEF position ~ (R,0,0) on equator). * Satellite position directly above ground station at some altitude h. If the satellite is directly overhead, elevation should be 90°. For example, satellite at (R+h, 0, 0) in ECEF (which is actually colinear with ground station vector). Our function might yield 90° or very close (floating math might give 90 or we define it). * Satellite at horizon: If the satellite is just at horizon, elevation = 0. We can approximate a position just out of reach: e.g. if the satellite is on the equatorial plane 90° away in longitude: ground at (R,0,0), satellite at (0,R+h,0). That's 90° away, should be at horizon (neglecting altitude, if h small relative, maybe slight above horizon if h > 0). We can test that our calculation returns something near 0° (maybe slightly positive if h is not extremely large – could solve exactly for h that yields 0°, but easier to test a known scenario). * Satellite below horizon: e.g. satellite on opposite side of Earth: ground at (R,0,0), satellite at (-R,0,0) (with or without altitude). That should give negative elevation (or our function might just give e.g. -5° if just below). * We will test: - Elevation of overhead sat ~ 90°. - Elevation of horizon sat ~ ~0° (within a tolerance). - If we apply a min_elev, e.g. min_elev=10°, we check that our logic would mark horizon case as not available but overhead as available. * We can also test non-equatorial (to ensure formula handles general cases): ground at lat 30°, a satellite at appropriate position. But that might be overkill if our formula is robust. Possibly one test is fine for function. - *Earth Occlusion (sat-sat)*: Construct simple cases using a spherical Earth radius R: * Case 1: Two satellites both at altitude 0 (on the surface) on opposite sides of Earth: obviously no LoS. Our `line_of_sight_space(A,B)` should return False. We simulate with positions A=(R,0,0), B=(-R,0,0). The closest approach of line AB to center is 0, definitely <R, so blocked. * Case 2: Two satellites very far out (e.g. GEO altitude ~ 35786 km) but also opposite sides. Even at GEO, opposite sides probably still no LoS because Earth is in between (distance ~ *242164 km, Earth radius ~6371, line passes through Earth). We can test something like A=(R+35786,0,0), B=(-(R+35786),0,0) -> should be blocked. * Case 3: Two satellites relatively close in the sky: - e.g. same orbit plane separated by small angle. For simplicity, A=(R+500, 0, 0) (satellite just above point on equator), B rotated*

*maybe 30° around Earth: B = ((R+500)*cos30, (R+500)*sin30, 0). 30° separation might still have LoS? Actually if 30°, line likely goes ~some thousands km above Earth, likely clear. We can test and expect True. - We could also test a borderline case: find the maximum separation angle for given altitude that still has LoS. But that's complex to derive manually. Instead, pick an intuitive scenario: * For ISS-like alt (~400 km altitude), it's roughly known that two ISS satellites can see each other if within ~~30°-40° (just guess). Our test can pick 30° and see result – expecting maybe still blocked or maybe free? Actually need guess: If each at 400 km, 30° apart likely still Earth between partly... We might just pick a clearly small angle like 10° and expect LoS = True. - Or an easier scenario: same position (A = B) trivial LoS (True) – just to ensure code doesn't do something weird with divide by zero or so. * We will verify the algorithm outputs expected booleans for these cases. -* Distance and Delay*: Test the distance function (should be straightforward vector diff norm). For example, positions (0,0,0) to (1000,0,0) yields 1000. Or better, use known distance: from Earth center to horizon of ground station ~? But simpler, just test a known vector distance. Then test propagation delay = distance / c: e.g. 300,000 km => ~1 s. We can test small values easily to ensure the formula is correct.

4. ConnectivityService Logic Tests (Functional Tests)

These tests treat the connectivity update function as the unit, using mocked or simple data: - *Single Link availability*: Construct a simple scenario in code: - Create one satellite node with a wireless interface, one ground node with a wireless interface. Use known positions for the satellite (maybe manually set it in ECEF for test). - Create a link between them (sat -> ground directional). - Use the ConnectivityService's function to update that link. - Set the satellite's position such that it is directly above the ground (e.g. as per earlier, ground at (R,0,0), sat at (R+500,0,0)). Then call update. Check that link.status becomes AVAILABLE (True) because sat is in view. Check that the computed delay is reasonable (~(R+500 - R)/c = 500 km/c ~ 1.67 ms, it might store 0.0017 s). - Now set satellite position to a blocked position (e.g. sat on other side of Earth: (-R-500,0,0)). Update again (simulate time progression or just manually set pos and call update). Check that link.status = UNAVAILABLE. - This test effectively verifies horizon logic via the service. - Additionally, if ground interface has min_elev >0, we can adjust scenario: e.g. set satellite at a low elevation angle ~ 2° and min_elev = 5°, ensure link becomes UNAVAILABLE due to min elev cut (we might simulate that by picking a position just above horizon). - *Inter-satellite link toggle*: Two satellites with a link: - Create two satellite nodes A and B, each with a wireless interface. Set their positions for two times: * Time1: positions such that they have LoS (e.g. near each other). Perhaps put both at (R+700, 0, 0) and (R+700, 1000, 0) for a short separation of 1000 km along Earth's surface – likely LoS (since they are nearly in same direction from Earth, 1000 km apart horizontally at roughly same altitude). * Time2: positions such that Earth blocks them. For example, put one on opposite side relative to the other: A at (R+700, 0, 0), B at (-R-700, 0, 0). Then check not available. - Actually, to simulate time, we can just manually reposition and call update each time. - Check link.status true in first case, false in second. - If the link is bidirectional (A<->B), ensure that both directions reflect the same status. We might define it as one bidirectional link in scenario. The service logic will likely handle it once if coded well. But to be sure, we could instead define two directional links (A->B and B->A) to simulate a bidirectional. Then see that after update, both get same result. (Our service might handle each directional independently but geometry gives symmetrical result anyway). - We also test that if one satellite's interface is marked impaired = down, then even if geometry was favorable, link becomes UNAVAILABLE. Simulate by setting B.interface.impaired = True before update call; then expect link down (and ideally, perhaps skip geometry check). - *Multiple Links and Node with multiple interfaces*: - Build a scenario with: one satellite (Sat1) with a wireless interface, a ground station (GS1) with two interfaces (one wireless, one wired), and an internet node (INET) with a wired interface. - Create links: Sat1.if0 <-> GS1.if0 (wireless bidirectional link), and GS1.if1 <-> INET.if0 (wired link). - Place Sat1 in a position above GS1 for part of orbit and below horizon for another: * GS1 at some lat/long, say equator. * Sat1 at time T0: above GS (so link should be available). At time T1: below horizon (unavailable). - Execute connectivity update at T0 and T1: * At T0: Expect Sat-GS link AVAILABLE, GS-INET link AVAILABLE (wired always on). So overall, Sat can reach INET (2-hop) at T0. The test should verify link statuses individually: - Sat1-GS1 link.status == AVAILABLE. - GS1-INET link.status == AVAILABLE. * At T1: Sat moved, Sat1-GS1 link.status

== UNAVAILABLE. GS1-INET remains AVAILABLE (wired unaffected by sat). - This scenario also tests that the ground station GS1's two interfaces operate independently: one losing connectivity doesn't affect the other (besides that the overall path sat->inet is broken, but as links themselves, fiber stays up). We verify that in the data: GS1-INET link stays true. - Also test that the GS1's **multiple interfaces** are correctly handled: the wireless link uses GS1.if0 and the wired uses GS1.if1. The connectivity service should not mix these up. For instance, ensure that when Sat1-GS1 wireless goes down, it doesn't accidentally mark GS1's wired down (it shouldn't, as they are separate link objects). - If we want to test multiple simultaneous wireless connections: Add another satellite Sat2 with link to GS1.if0 as well. Now GS1.if0 has two links (to Sat1 and Sat2). Place Sat1 and Sat2 both above horizon at same time (maybe on different sides of sky but both visible). Both links should become AVAILABLE. This tests that GS1.if0 can have two available links concurrently. We expect our system to mark both as available. We'll verify: - link Sat1-GS1 = AVAILABLE, link Sat2-GS1 = AVAILABLE at T0. - If we had `max_beams=1` configured, we are not enforcing it, so still both are available. We can assert that we did not mistakenly drop one. (We might log a warning in future, but in Scope 2 no action). - This verifies that concurrency control is deferred. - Then perhaps move Sat2 below horizon leaving Sat1 up at T1: check one link goes down, the other remains up. This ensures independent evaluation. - *Impairment effects*: - Take one of the above scenarios and do: mark GS1.if0 as down (simulate hardware failure) at some time. Then run update. Even if Sat1 is overhead, link Sat1-GS1 should now be UNAVAILABLE due to GS1.if0 impairment. Check that in output. - Similarly, mark the GS1-INET wired link as down (maybe we have a method like `link.set_impaired(True)`). After update, that link shows UNAVAILABLE (even though wired and normally always up). This confirms the impairment mechanism overrides normal state. - These can be separate unit tests focusing on impairments toggling link states.

- *Edge case: Node with no Platform and wireless interface*:
- We can create a dummy scenario where a network node with no platform (e.g. an "Internet node" not attached to any platform) erroneously has a wireless interface defined. While this should be avoided, we want to ensure the system handles it gracefully. The connectivity check might find a link connecting that interface to a satellite's interface. Since the node has no position, we expect our code to mark the link unavailable and log an error.
- Test: Create Node INET (no platform), add a wireless interface to it (perhaps model doesn't matter). Link it to Sat1.if0. In update, since INET has no coordinates, we cannot compute LoS properly. Our plan was to treat this as no connectivity. We check that link.status becomes UNAVAILABLE. We also ensure no crash happens (our code should check for platform presence and handle it).
- This test ensures robustness to scenario misconfiguration. In practice, we'll discourage such links (should use a platform for any radio node), but our simulator should not fail catastrophically if it occurs.

These unit tests ensure each piece of logic works in isolation or in tightly controlled mini-scenarios.

Integration Tests (End-to-End Scenario Tests)

We will also test full scenarios that mirror realistic use-cases, combining multiple components:

1. Mixed Constellation Scenario (LEO, GEO, Ground, Internet) – *extended Scope 1 scenario with links*:
We will take the **sample scenario from Scope 1** (which had multiple satellites and ground nodes) and extend it with interfaces and links to verify end-to-end behavior: - **Scenario Description**:
- *SatLeo1* (LEO satellite) with NetworkNode *node_leo_1*. We give node_leo_1 a wireless interface "if0" using model LEO_Satellite_KuBand.
- *SatLeo2* (another LEO) with *node_leo_2*, also with wireless "if0" (same model or similar).
- *SatGeo1* (GEO satellite) with *node_geo_1*, with wireless "if0" using model (could be same KuBand or a

different band if we want to test mismatches; but better use same to allow linking).
- *GroundStation1* (GS1) with *node_gw_1*, attached to a fixed platform at, say, latitude appropriate to see SatGeo1 (e.g. equatorial). node_gw_1 gets **two interfaces**: "if0" wireless (GroundStation_KuBand model) and "if1" wired (Ethernet/fiber).
- *UserTerminal1* (UT1) with *node_ut_1*, attached to a ground platform perhaps far from GS1 (maybe not needed if we focus on SAT-UT links). UT could have a wireless interface "if0" (maybe also KuBand to talk to LEOs).
- *InternetNode1* with *node_inet_1* (no physical platform, representing a core network gateway). It gets a wired interface "if0" (to connect to GS1).

• **Links**:

◦ SatLeo1.if0 <-> GS1.if0 (bidirectional wireless link, representing that LEO can link to that ground station when in view).
◦ SatLeo2.if0 <-> GS1.if0 (maybe also allow second LEO to same ground station).
◦ SatGeo1.if0 <-> GS1.if0 (GEO to same ground station).
◦ UT1.if0 <-> SatLeo1.if0 (optional: if we want to simulate a user connecting via the LEO satellite; or UT1.if0 <-> GS1.if0 as an alternative scenario representing maybe a terrestrial wireless link – but more realistic is UT to LEO). We can include UT to LEO downlink to see how a moving LEO covers a user.
◦ GS1.if1 <-> INET1.if0 (wired link from gateway to internet).

• **Expected Behavior**:

• Initially, at simulation start (t=0), we configure satellite positions:
◦ Suppose GS1 is at lat 0, lon 0. Place SatGeo1 in GEO orbit at lon 0 (so it's always above GS1, roughly at zenith). SatGeo1->GS1 link should be **continuously AVAILABLE** throughout the sim (with minimal variation in geometry). We expect the simulator to keep that link up every tick. We will verify that over a long duration, it never drops.
◦ SatLeo1 is in LEO orbit (e.g. 500 km, inclined maybe). As time progresses (simulate e.g. 0 to 2 hours), SatLeo1 will periodically come above GS1's horizon and then set. We expect multiple access windows. The simulation, running stepwise, should mark the **Leo1-GS1 link** as available during those windows. We will verify times – e.g., by printing when it goes up/down – and ensure it matches an external prediction (like using an SGP4 to find passes). We will specifically test that at the highest point of a pass the link is up (elev > 0). If GS1 has min_elev 5°, the link might come up slightly after rise and go down before set, which we check.
◦ SatLeo2 similarly, if included, perhaps on a different orbit or offset in phase, to show possibly when SatLeo1 is out of view, SatLeo2 might come in later, etc. This tests multiple satellites scenario. The ground station can handle both but one at a time in reality. Our simulation might show times they overlap as both available. This is fine as an output; we just note it.
◦ UT1 and SatLeo1 link: UT1 might be located such that it has visibility at different times. For instance, if UT1 is far from GS1, maybe Leo1 passes over UT1 at a different time. This can demonstrate a *user terminal coverage* event. We'll verify UT1-SatLeo1 link availability changes as well.

• Wired link GS1-INET: should be always on, so regardless of satellite status, GS1 to internet is up. Thus whenever a satellite is linked to GS1, effectively the user could reach internet (two-hop path). We ensure GS1-INET link remains available in all logs. If we simulate a failure, e.g. break that link at some point, then all flows would break; but we might not do that in this normal scenario test, it's more for separate impairment test.

- **Assertions/Checks**: We will run the simulation for a full period (e.g. one orbit of LEO ~ 90 min). During the run:

- Count how many times each link changes state and ensure it's plausible (e.g. Leo1-GS1 link: maybe ~ once per orbit it should go up then down).
- Ensure no spurious toggles (e.g. not flapping too frequently – should only toggle when geometry dictates).
- Check that Geo1-GS1 link remains constantly available (no toggles at all).
- If UT1 is included: check UT1-Leo1 link toggles at times (depending on UT location).
- Also confirm that at no time is there a link declared available that shouldn't be: for example, if SatLeo1 is below horizon, link is off. We can cross-validate one event: when our sim says "Leo1-GS1 up at time T", compute roughly the elevation at T to see if > 0. These are sanity checks (or even write an assertion if we calculate it within test).
- Multi-link concurrency: if both Leo1 and Leo2 are above horizon together (rare if orbits different RAAN maybe one after other, but if by chance overlap a bit), the test would catch that both links are true simultaneously. This is fine; we just note it. If our design had incorrectly enforced single-beam, we'd see one drop – but we expect no drop.

- Performance: This test also mildly checks performance with multiple links – the simulation should handle it without lag (we might run with small time steps, ensure it iterates fine).

- **Result Logging**: For human verification, we will log statements like:

```
Time 00:30:00: link Leo1-GS1 UP (elev=45°), link Leo2-GS1 DOWN, link
Geo1-GS1 UP, link UT1-Leo1 UP, ...
Time 00:45:00: link Leo1-GS1 DOWN (sat set), ...
```

But for automated testing, we collect states and assert conditions described.

This integrated test essentially demonstrates that the **constellation connectivity graph** behaves as expected: e.g. at some time, Leo1 is connected to ground which is connected to internet, so end-to-end path exists; later that path goes away when Leo1 sets.

2. Edge Integration: Frequency Mismatch and Validation
Construct a scenario where two interfaces with incompatible frequency bands are erroneously linked, to see if our system handles it: - E.g., define SatX with a Ka-band transceiver, GS1 with Ku-band transceiver, and a link between them. Our link creation should ideally catch this and either not create the link (error) or mark it as never available. - We test that the simulator does not treat it as available at any time (since frequencies don't match). Possibly we print a warning "Link X has incompatible frequencies – will remain down". - This is more of a validation test than a physical simulation. If implemented, it's good to ensure the check works.

3. Impairment Scheduling Test (if scheduling not implemented, skip, but at least manual toggle): - We simulate a scenario with a single satellite-ground link. We then *manually* disable the ground station at a certain simulation time (maybe by calling an API or directly setting the flag in code at time T_impair). - Verify that after T_impair, the link immediately goes down (even if geometry was fine). - Then re-enable and see link come back next pass. - This can be done in a controlled loop: - Run sim until t = T_impair - δ, ensure link is up. - At t = T_impair, set interface impaired = True. - Continue sim: link goes down. - At t = T_impair + some Δ, clear impairment. - When satellite comes back in view later, link goes up again. - This

shows the simulator responds to runtime changes not just geometry. It's likely more for later scopes (failover behavior in scheduling), but we can test the mechanism now.

The **Test-Driven Development** approach means we will write the above tests first (likely using a testing framework like `pytest` or `unittest` in Python). Each test corresponds to a clearly defined requirement (traceable to our objectives list). We will then implement the functionality to make these tests pass, one by one. This ensures that we meet all requirements and catch any regression in the logic.

Libraries, Tools and Protocols

To implement Scope 2 effectively, we will leverage appropriate libraries and follow relevant protocols/ standards:

- **Orbit Propagation**: Continue using the SGP4 library integrated in Scope 1 [9] for satellite orbital updates. We already have this in place; no change needed except to ensure we can get ECEF coordinates for satellites (if not already done). We might add a utility to convert SGP4's TEME output to ECEF. For that conversion, we could use a library like **AstroPy** or **Skyfield** to get Earth's rotation (GMST) for the given time. However, to minimize dependencies, we might implement a simple Greenwich sidereal time formula ourselves for conversion. The accuracy needed is modest (a fraction of a degree error in position yields minimal impact on LoS determination).
- **Coordinate Calculations**: We will use **NumPy** for vector and matrix operations to simplify the math for checking angles and distances. NumPy's array operations will let us calculate dot products and norms cleanly, which reduces the chance of manual math errors and improves performance if computing many links at once.
- **Earth Geometry**: If high precision is desired, we might incorporate WGS-84 ellipsoid for elevation calculations. A library like **PyProj** could transform lat/long to ECEF precisely. In Scope 2, a simple spherical model is acceptable, but PyProj is a fallback if needed for coordinate conversions. We likely won't need it if we write our own given lat, lon (since formula for ECEF from lat, lon, alt is straightforward).
- **Logging**: Use Python's built-in `logging` module (as likely set up in Scope 1) to record connectivity events. We'll possibly define a logger for "ConnectivityService" and use levels INFO for link up/down events and DEBUG for more detailed geometry data. This aligns with good practice to have tunable verbosity and traceability.
- **Data Modeling**: If the project uses Python, we can use `dataclasses` for NetworkInterface, TransceiverModel, etc., to automatically generate init and repr, making the code cleaner. If using an object database or similar (maybe not, since likely in-memory), dataclasses suffice. In a static language like Java or C++, we'd define these classes normally. The key is to structure them clearly as per the design.
- **Protocols & API Alignment**: We have the Aalyria Spacetime API (`api-main.zip` provided). We should use it as a reference for field names and allowed values:
- e.g. use the same enumerations for interface medium (if possible), same structures for Link (the NBI defines `NetworkLink` and `BidirectionalLink` messages). We might even consider using the proto definitions directly by compiling them and using the generated classes in our sim. This would ensure exact alignment. For example, the proto `NetworkLink` has fields for `radio_config` etc. We can choose to either use those protos or create our own classes that mirror them. Given the simulator is our own code, it might be simpler to define our classes and just ensure similar naming. We could import constants like modulation modes from the proto to avoid retyping them.

- Using the actual protos may be more relevant when we implement the gRPC interface in Scope 3, at which time we can integrate more tightly. For Scope 2, referencing them is enough.
- **Spacetime Simulation API**: Also note Spacetime has a Simulation API for scenario definition (maybe in `simulation.proto`). We might glean some insight from it to ensure our concept of "Scenario" and how we group nodes/links aligns. Possibly the Simulation API allows posting a scenario with all nodes, links, etc. We've essentially built that concept in our KB. We remain aligned by treating our scenario as the collection of nodes/links which is what Spacetime expects [35].
- **Mathematical Libraries**: If needed, use `math` module for trig (sin, cos, etc.). For computing sidereal time, might need some constants (Earth rotation rate).
- **SGP4**: Already covered, but ensure we have latest version (some have minor differences in output).
- **Unit Testing Framework**: Use `unittest` or `pytest` in Python. We will likely go with `pytest` for its ease of writing parametric tests, etc. We will also use assertions with tolerances for floating comparisons (like elevation angle ~0).
- **Continuous Integration**: If set up, we add these tests to CI to automatically run them. (Not a library per se, but part of development rigor).

- **Performance Tools**: Not needed at this stage, but if we wanted to simulate a very large constellation, we might consider optimizing the geometry checks. Possibly using NumPy's vectorization to compute LoS for many links at once, or employing spatial indexing (like grouping satellites by visibility region). At our current scope, not necessary, but we mention it as a forward-looking note.

- **Standard Protocols**: The simulation doesn't directly interface with network protocols at this stage (since we are not sending actual packets). However, being "Spacetime-compatible" implies we respect certain standards:

- *CCSDS standards*: The requirements mention DTN (Delay Tolerant Networking) with Bundle Protocol 7 etc. [38] [39], and mention DVB-S2X, etc. These inform our field choices but we are not implementing the protocols themselves here. We ensure our design can accommodate them (e.g. storing is_disruption_tolerant in service requests, and modulation schemes in radio config).
- *Time system*: We should use a precise time representation (likely UNIX epoch or a datetime) for simulation time, which might matter if converting to real world time for orbital calcs. For consistency, we use the same epoch as SGP4 expects (likely UTC). Use of Python's `datetime` and perhaps `astropy.time` if needed for conversions (or just manual if not).
- *Units*: We will stick to SI units internally (positions in meters, time in seconds, angles in degrees for human input but convert to rad for math). Document these clearly to avoid confusion.

In summary, the main libraries we will leverage are the **SGP4 orbital library**, **NumPy for vector math**, and possibly `pyproj` or `astropy` for coordinate transforms if needed. We align with **Protobuf definitions** from Aalyria's API to ensure our implementation's data model can be directly mapped to the Spacetime API objects.

Error Handling, Logging, and Metrics

Robust error handling and clear logging are crucial to a "world-class" simulator backend:

- **Input Validation Errors**: As we parse scenario definitions and set up interfaces/links, we will validate and throw explicit errors for any inconsistencies:
- If a link references an unknown interface or node, we raise an error indicating the bad reference.

- If two interfaces of different mediums are linked (one wired, one wireless) – which is logically invalid – we either prevent it or interpret it as meaning a misconfigured scenario. We will likely treat it as invalid input and log an error. (Alternatively, one could imagine a scenario linking a wireless interface to a wired interface to simulate a "gateway" where one end is actually some modem – but that should be modeled as two interfaces on one node rather than a direct link. So we will not allow cross-medium link direct connections.)
- If frequency bands do not overlap between two wireless interfaces on a link, we log a warning and mark the link as never available. This is not a crash scenario, but we output a message so the user knows that link will never come up due to incompatible frequencies. (In later scopes, perhaps a frequency conversion payload could be modeled, but that's beyond scope.)
- If an interface is added to a node with an ID that already exists, log error and prevent duplication.
- If a ground station's min_elev is not provided, assume default. If provided and > 90°, or <0°, clamp it or warn (invalid values).
- If transceiver model ID on an interface is not found in the model registry, log error and either fail scenario load or set a default model as fallback and warn.

- All such messages will be collected and either thrown as exceptions (to not start simulation with bad config) or at least clearly logged so the user can fix the scenario.

- **Runtime Errors**: During simulation:

- The geometry calculations should be safe (no division by zero except possibly in degenerate cases like two identical satellite positions, which won't happen unless a satellite coincides, but even then our line-of-sight function would handle gracefully).
- We must guard against numerical issues: e.g. floating precision causing an elevation of -0.0001° which is essentially 0. We might treat any elevation < 0 as 0 for availability (or require strictly > minElev). We will define that clearly (maybe require elevation >= min_elev to be available, i.e. equal to threshold counts as available or not? We likely say > or >= – we can choose >= so that if min_elev = 0, horizon exactly yields available; if user wants strictly above horizon, they can set min_elev a fraction above 0).
- If the SGP4 propagation fails for some TLE (maybe TLE is expired and SGP4 returns an error or NaNs), our orbit update might produce an invalid position. We should detect that (position with NaN or absurd values) and mark any links involving that satellite as down, and log an error like "Satellite X position invalid at time Y – propagator error". The simulation should continue (other satellites unaffected) rather than crashing. This is partly Scope 1's concern, but Scope 2 will propagate the effect to connectivity (so we should ensure we handle an invalid position by not attempting to compute crazy geometry).
- The connectivity update itself is not expected to throw exceptions normally. We'll make sure to handle things like the case of missing platform (as discussed) by logging and skipping.

- If any unexpected error occurs in the update loop (e.g. due to a bug), we will catch exceptions at the top level of `update_all_links` to prevent the simulation from freezing. We can log a critical error with the stack trace and attempt to continue next tick. (Better the sim limps than dies, especially in a long run – though we prefer to fix bugs rather than rely on this.)

- **Logging**:

- We will implement **informational logs** for key events:
    - When each link becomes available or unavailable (with timestamp and possibly reason if easily determined, e.g. "Link X down (out of range)" or "Link Y up (LoS established)").

    - We can throttle logs to avoid flooding. E.g., if updating every second, logging every link every second is too much. Instead, we log on state changes. This requires storing previous state (which we have in link object). We'll do: `if new_status != old_status: log(info)`.
    - Also log once at scenario initialization listing all defined links and their parameters, so the user sees the configured network (frequencies, etc.).
    - Log any warnings as discussed (invalid configs).
- **Debug logs**: We may add a debug-level log with geometry details for one or two links if needed (but not for all links each tick, that's too verbose). Possibly allow enabling debug for a specific link ID to trace its elevation over time – useful for developers. But that might be beyond initial need; a developer can instrument manually or examine outputs.

- **Metrics**: We can keep counters such as:

    - `total_links = N`, `wired_links = X, wireless_links = Y`.
    - In each tick, count `active_links`. We could output a high-level metric like "Time T: active_links = M out of N". This can help quickly see connectivity percentage. If integrated with a monitoring system, these metrics could track network availability.
    - For each link, we could measure the fraction of time it was available during a scenario (this could be computed after the run by analyzing logs or by incremental count: e.g. each tick add dt if available). This is more of a post-simulation analytics, but we could include an option to accumulate that. For now, we might not implement in code, but we plan for it (the design can easily add it).
    - Performance metrics: how long the connectivity update takes each tick (if we wanted to optimize). We can measure time at start and end of update_all_links and log at debug if it exceeds some threshold. Given likely small N, it's fine. But as project scales, we consider it.

- **Extensibility & Rigor**:

- We maintain rigorous separation of concerns: The connectivity logic does not alter platform motions or other state beyond links. This makes debugging easier (we know a bug in connectivity won't mess orbits).
- We will double-check consistency: e.g. if a bidirectional link's one side was updated and not the other, we add a final consistency step to copy state (though if our code updates both directional links in one go, it's consistent by design).
- At the end of Scope 2 implementation, we will review if all requirements marked for Scope 2 are indeed covered or explicitly deferred:
    - Network interfaces & transceivers – fully implemented [40].
    - Link, Beam & connectivity – partially implemented: we have links and connectivity but beams (pointing/scheduling) are not yet. We document which parts are stubbed (e.g. multi-beam concurrency, adaptive modulation, link budgets beyond geometry).
    - This transparency ensures the "world-class rigor": any functionality not implemented is acknowledged and either stub-return or logged as "Not implemented in this version" when called.
- We incorporate small comments referencing future scope features, e.g. "TODO: enforce max_beams in scheduling phase" to not forget.

Testing will include intentionally bad scenarios to see that errors are caught and logged, not silently ignored (unless harmless). We will ensure the simulator fails fast on critical configuration errors, and on runtime issues it either recovers or logs clearly.

Finally, after thorough testing and debugging, we will have confidence that a developer can run the simulator with a complex scenario and get correct connectivity behavior. The documentation (this plan and code comments) will enable them to further build on it – for example, a developer implementing Scope 3 (which might involve the Northbound API and user traffic flows) can see exactly how to retrieve connectivity info, or how to add new interface types.

Conclusion

By implementing Scope 2 as outlined, we add a crucial layer to the constellation simulator: the ability to represent each node's interfaces and dynamically evaluate connectivity. The modules defined (NetworkInterface, TransceiverModel, NetworkLink, ConnectivityService, etc.) directly fulfill the Scope 2 requirements [41] [40] and set the stage for subsequent scopes. The system will now maintain a time-varying network topology graph that reflects real orbital motion and basic physics constraints (line-of-sight). This provides a solid foundation for Scope 3 and Scope 4, where we will introduce user flows, routing decisions, and active link scheduling. All design decisions have been made with **extensibility** in mind – for example, by stubbing out advanced fields (like link quality metrics) and aligning with Spacetime's API models, we ensure that adding features like link budget calculation, dynamic beam control, or routing algorithms can be done incrementally without redesign.

The implementation plan above, with its clear breakdown and rigorous detail, should enable a developer (or team) to begin coding Scope 2 immediately and systematically, using the specified tests as a guide. Upon completion of Scope 2, the simulator will achieve **Roadmap v0.1** capability – it will support a simple static scenario with orbit propagation (from Scope 1) and now also compute which links are up as satellites move [42]. This demonstrates that the simulator is progressing toward being fully Spacetime-compatible, adhering to world-class standards for modularity and accuracy. All requirements targeted in Scope 2 will be satisfied, and the system will be ready for the next phases (adding traffic handling, control-plane integration, etc.) with confidence in the robustness of the core networking model.

Sources:

- Requirements Document for Aalyria Spacetime-Compatible Simulator [21] [5] [6] [7] (network interfaces, transceivers, wired link assumptions)
- Development Roadmap for Simulator (Scope 2 goals: interface definitions and LoS connectivity)
- Architecture Specification [33] [8] (connectivity service line-of-sight logic and range/coverage assumptions)
- Scope 1 Implementation Plan [37] [12] (context of interfaces stubs in Scope 1, scenario baseline)
- Aalyria public API references (for alignment with NetworkInterface, NetworkLink definitions) [29].

---

[1] [2] [3] [4] [9] [10] [11] [15] [34] [40] [41] [42] Roadmap for Spacetime-Compatible Constellation Simulator Development.pdf
file://file_00000000541c7206a67b773409b72c28

[5] [6] [7] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [35] [36] [38] [39] Requirements for an Aalyria Spacetime-Compatible Constellation Simulator.pdf
file://file_0000000056b871faaec4ce4c5e567993

[8] [32] Architecture for a Spacetime-Compatible Constellation Simulator.pdf
file://file_000000009e44720799a541f2cf93f77a

12 Scope 1 Implementation Plan_ Core Entities & Orbital Dynamics.pdf
file://file_00000000b3c47209aa09177fed7c1ea1

13 14 37 Scope 1 Implementation Plan_ Core Entities & Orbital Dynamics.pdf
file://file_000000008d447207bd1eb280e59f4c76

16 Requirements for an Aalyria Spacetime-Compatible Constellation Simulator.pdf
file://file_0000000058407206a1f0d66f1b1e61c7

33 Architecture for a Spacetime-Compatible Constellation Simulator.pdf
file://file_000000000ac47207829eeb8ec2dd94b2