Station Arcturus: System Architecture Overview

Introduction

This document provides a comprehensive architectural overview of the Station Arcturus real-time beacon tracking system. Station Arcturus is a simulation application composed of a Python backend that generates and broadcasts dynamic beacon data, and a Unity frontend that provides a live 3D visualization of that data. This overview will detail the system's high-level architecture, provide an in-depth analysis of the backend and frontend components, and describe the communication protocols that integrate them into a cohesive, real-time system.

1.0 High-Level System Architecture

Station Arcturus employs a classic client-server model, architected to decouple the data simulation logic from the user-facing visualization. This decoupling allows the simulation engine to be iterated on—for instance, to increase its complexity or connect it to a real-world data source—without requiring any rebuild or redeployment of the visualization client. Conversely, the frontend can be ported to new platforms like VR or mobile while consuming the same stable data API. The system is comprised of two distinct, collaborating applications.

- **Python Backend Server** This component serves as the authoritative source for all beacon data. Its primary responsibilities include generating an initial set of simulated beacons, continuously updating their positions and statuses in real-time, and broadcasting these changes to all connected clients.
- **Unity Frontend Client** This application is the visualization and interaction layer. Its primary role is to establish a connection with the backend server, receive the stream of data updates, and render the beacons within an interactive 3D environment, providing a visual representation of the simulated data.

The core interaction between these components is managed through a persistent WebSocket connection, facilitated by Socket.IO. The backend server broadcasts state updates over a persistent, stateful WebSocket connection, and the Unity client listens for these events to update the 3D scene. In addition to this primary real-time channel, a stateless REST API is available for initial data retrieval and on-demand status checks. To understand how data originates and is managed, we will first examine the backend server's architecture in detail.

2.0 Backend Architecture (Python / FastAPI)

The backend is the heart of the Station Arcturus simulation, responsible for creating the dynamic and persistent data stream that gives the frontend its purpose. Its architecture is designed for performance and real-time responsiveness, leveraging a stack of modern, asynchronous Python frameworks to handle concurrent operations and client connections efficiently.

The backend's architecture is a purpose-built stack of modern, asynchronous Python frameworks, with each component selected for a specific, high-leverage role:

The backend application is organized into a modular structure within the `app/` directory, promoting separation of concerns:

- **`main.py`**: The application's main entry point. It is responsible for initializing the FastAPI application, integrating the Socket.IO server, configuring CORS middleware to allow client connections, and defining all HTTP endpoints. It also manages the application lifecycle, launching the `beacon_updater` background task on startup and ensuring its graceful cancellation on shutdown.
- **`models.py`**: Defines the Pydantic data models that structure the application's data. This includes the core `Beacon` model and the `BeaconStatus` enum, which restricts status values to `ACTIVE`, `DAMAGED`, or `OFFLINE`.
- **`beacons.py`**: Contains the core simulation logic. This file includes functions for generating the initial set of beacons (`generate_initial_beacons`), updating their positions and statuses, and the primary `beacon_updater` asynchronous background task that drives the simulation.
- **`events.py`**: Manages all Socket.IO event handlers. This module defines server responses to events like `connect`, `disconnect`, and `ping`, centralizing the real-time communication logic.
- **`run.py`**: A simple script used to launch the Uvicorn server, providing a convenient way to start the application during development.

Beacon Data Model

The `Beacon` data model is the fundamental data structure in the system. It is defined using Pydantic and includes the following attributes:

- `id`: A unique string identifier for the beacon (e.g., `BEACON-0001`).
- `x`: A float representing the beacon's X coordinate.
- `altitude`: A float representing the beacon's height.
- `z`: A float representing the beacon's Z coordinate.
- `status`: A `BeaconStatus` enum value (`ACTIVE`, `DAMAGED`, or `OFFLINE`).

The model includes a `to_unity_dict` method, which serves a critical compatibility function: it serializes the object into a dictionary and maps the `altitude` field to a `y` key, making the data directly compatible with Unity's `Vector3` coordinate system. This method effectively defines the data contract for the frontend, ensuring the payload is immediately deserializable by the corresponding `BeaconData` class in C#, which expects a `y` coordinate field.

Communication Interfaces

The backend exposes two primary interfaces for client interaction: a REST API for state retrieval and a WebSocket endpoint for real-time updates.

REST API Endpoints

The following HTTP GET endpoints are available for querying server and beacon state.

Endpoint	Method	Description
`/`	GET	A root health check that confirms the server is running.
`/status`	GET	Returns the server's status, current beacon count, and time.
`/beacons`	GET	Returns the complete list of all current beacon data.
`/ui`	GET	Serves a static HTML test client for debugging WebSockets.

WebSocket Events

The Socket.IO server manages real-time communication through a set of defined events.

Event Name	Direction	Description
`connect`	Client → Server	Triggered when a new client establishes a connection.
`disconnect`	Client → Server	Triggered when a client disconnects.

`ping`	Client → Server	A simple client-initiated event to test connectivity; the server responds with a `pong` event.
`request_beacon_data`	Client → Server	Allows a client to explicitly request the full beacon list.
`beacon_update`	Server → Client(s)	The primary broadcast event, pushed from the server, containing the complete, updated list of all beacons.

The Simulation Loop

The backend's dynamic behavior is driven by the `beacon_updater` function, which runs as a continuous background task. This function operates in an infinite `while True` loop with a two-second delay. In each cycle, it performs three key actions:

1. **`update_beacon_positions`**: Randomly adjusts the coordinates of existing beacons.
2. **`update_beacons_statuses`**: Randomly changes the status of existing beacons.
3. **`simulate_beacon_changes`**: Randomly adds a new beacon or removes an existing one.

After these modifications, the task then serializes the complete state of the `beacons_data` dictionary into a list and broadcasts it via the `beacon_update` WebSocket event to all currently connected clients, ensuring they remain synchronized with the server's state.

With the backend continuously generating and broadcasting data, the architecture of the receiving Unity client is critical for visualizing this information effectively.

3.0 Frontend Architecture (Unity / C#)

The Unity frontend is responsible for translating the raw data stream from the backend into an interactive and meaningful 3D visualization for the user. Its architecture is component-based and event-driven, revolving around a set of key C# scripts that manage application state, handle real-time communication, and control the visual representation of the beacon data.

The frontend is built with the following core technologies:

Key Scripts and Responsibilities

The frontend's logic is distributed across several key C# scripts, each with a distinct responsibility:

- **`BeaconData.cs`**: A simple C# data structure class marked as `[Serializable]`. Its sole purpose is to serve as a template for deserializing the JSON beacon data received from the server into a native C# object that the rest of the application can easily use.
- **`WebSocketClient.cs`**: This script manages the low-level connection to the Socket.IO server. It handles connection logic, registers listeners for server-side events using the `On()` method, and ensures that any Unity API calls (like creating or moving GameObjects) are safely executed on the main thread via `UnityThread.executeInUpdate()`.
- **`BeaconManager.cs`**: The `BeaconManager` script serves as the architectural nexus of the frontend. It orchestrates the entire visualization by using the `WebSocketClient` to receive data. It maintains an internal dictionary of active beacon `GameObjects` and is responsible for instantiating new beacon prefabs, updating the position and status of existing ones, or destroying them based on the data received in each `beacon_update` event.
- **`UIController.cs`**: This script manages the UI dashboard. It references the `BeaconManager` to get real-time statistics (total, active, damaged, and offline beacon counts) and updates the corresponding TextMeshPro UI elements in each frame.
- **`CameraController.cs`**: This script provides interactive camera controls, allowing the user to navigate the 3D scene using WASD/Arrow keys for movement, Q/E for rotation, and the mouse scroll wheel for zooming.
- **`PlanetController.cs`** / **`OrbitalCameraController.cs`**: These components represent a significant architectural evolution from the initial flat-plane design, as detailed in the project's 'Planetary Beacon System' implementation guide. They implement a Hybrid Radial Projection approach, mapping the backend's Cartesian coordinates onto a spherical surface within Unity. This was a deliberate design choice to enhance visualization without requiring a complex and breaking change to the backend's coordinate generation logic.

Frontend Data Flow

When a `beacon_update` event is received from the server, the following sequence of events occurs within the frontend:

1. The `SocketIOUnity` client, managed by `WebSocketClient.cs`, receives the `beacon_update` event and its associated JSON payload.
2. The event handler in `WebSocketClient.cs` deserializes the JSON string into a top-level `BeaconResponse` object, which contains an array of `BeaconData` instances.
3. `BeaconManager.cs`, which is listening for an event from the `WebSocketClient`, receives this processed array of beacon data.

4. The `BeaconManager` iterates through the received beacons and compares them against its internal dictionary of managed beacon `GameObjects` using their unique IDs.
5. Based on this comparison:
    - **New beacons** (IDs present in the server data but not in the local dictionary) cause the instantiation of a new `beaconPrefab`.
    - **Existing beacons** have their position and status data updated on their corresponding `GameObject`.
    - **Missing beacons** (IDs present in the local dictionary but no longer in the server data) are identified, and their corresponding `GameObject` is destroyed from the scene.
6. Finally, in its `Update()` loop, the `UIController.cs` script fetches the latest beacon counts from the `BeaconManager` and updates the text fields on the UI dashboard.

The seamless operation of the frontend depends entirely on the robust communication protocol established with the backend.

4.0 Communication Protocol and Data Flow

The communication protocol is the crucial link that enables Station Arcturus's client-server architecture. It defines how the system achieves its real-time synchronization by blending a stateful WebSocket connection for live updates with a stateless REST API for auxiliary requests.

Client Connection Lifecycle

When a new Unity client connects to the backend server, a specific lifecycle sequence ensures immediate and continuous synchronization:

1. The Unity client initiates a connection to the backend's Socket.IO server.
2. The backend server registers a `connect` event and logs the new client's unique session ID (`sid`).
3. Immediately upon successful connection, the server emits a `connection_status` message and the entire current list of beacons via a `beacon_update` event, specifically to the newly connected client (`to=sid`). This initial, targeted `beacon_update` is a critical design feature for user experience. It ensures the client's scene is populated and synchronized with the server's state immediately upon connection, eliminating any perceived lag or need for the user to wait for the next two-second global broadcast cycle.
4. Subsequently, the client joins the pool of connected clients and begins receiving the global `beacon_update` broadcasts that are sent every two seconds to all clients.
5. If the client disconnects for any reason, the server registers a `disconnect` event and removes it from the broadcast pool.

Primary Data Payload

The primary data payload, transmitted with the `beacon_update` event, is a JSON object containing two keys: `beacons`, a list of beacon objects conforming to the Unity-compatible data contract, and `time`, a server-side Unix timestamp for reference.

Conclusion

In summary, Station Arcturus is a well-defined client-server system that effectively separates its concerns. The Python and FastAPI backend provides an efficient and scalable engine for data simulation and real-time broadcasting. This is complemented by the Unity frontend, whose component-based C# architecture is designed to receive this data stream and translate it into a responsive and interactive 3D visualization. The synergy between these two components, linked by a clear and robust communication protocol, results in an effective and compelling real-time tracking system.