SpaceOS: Design and Implementation

Prerequisite Knowledge

For SpaceOS:

Operating System

Kernel

Spatial Computing

Type Checking

Decidability

For TCShell:

Natural Language Interfacing

Natural Language Processing

Large-Language Models

Database Interfacing

# Abstract

…

# 1. Introduction

Spatial computing is a computing paradigm that integrates the physical environment into a computational system. A prime example of spatial computing is AR/VR, where users interact with digital content that is mapped to physical space through a variety of sensors and input output devices.

Current widely-adopted implementations of spatial computing are user-focused: the system aims to enhance the computing environment of a specific user in a limited, oftentimes predefined space, using a centralized computer to handle space-contextual tasks. These implementations operate on an individual scale, such as AR devices enhancing the physical space around a user with digital models. However, by prioritizing single-user computation scenarios, these implementations treat communication services between devices as an afterthought.

By redefining spatial computing as a method of managing a distributed system of devices that operate in a singular universe, mapped to a real-world universe, inter-device communication and collaboration becomes a fundamental problem: how do we design an operating system that minimizes the communication overhead for computers operating in a real-world scenario?

# 2. SpaceOS

SpaceOS is an operating system design that prioritizes physical 2D space representation and interfacing within the operating system. Similar to how UNIX-style operating systems treat processes and other resources as files (and can be interacted with as files), SpaceOS aims to represent all system objects in a way that mirrors real-world spatial mapping, and provides computational functions with concepts similar to those found in real-time operating systems (RTOS) and distributed systems. The main focus of SpaceOS is to dictate an overarching system of a universe to facilitate low-cost communication between members residing in said universe.

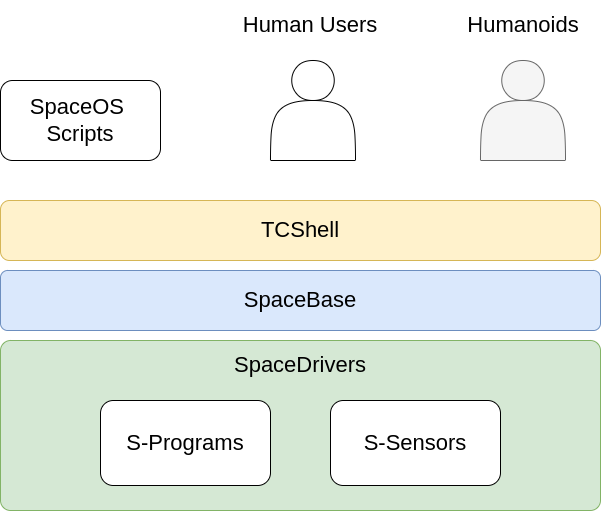


Figure X: Design of SpaceOS

### 2.0.1 Example SpaceOS Use Case

To understand how SpaceOS is organized and functions, consider the following example of a drone handling deliveries from a warehouse to a home.

In this case, the drone (an entity) delivers packages (entities) from the warehouse (a space) to a home (another space) on a preset road (a path).

## 2.1. Key Terminology

Universe: the 2-D totality of all known spaces and objects. All spaces in SpaceOS must reside in the universe.

Space: a 2-D location in the universe. There exists a set of spaces that are neighbors to said space.

Entity: an observable object acting within a space. All entities are assumed to be observable, but not controllable.

Paths: a set S of n connected spaces [s1,s2, … , sn] where each space is neighbors with the following space.

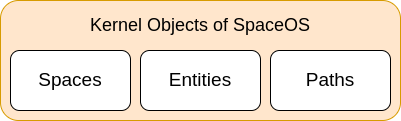
Objects: objects refer to any space, entity, or path residing in the universe.

## 2.2. Elements of Space OS

In a given operating system, objects are categorized into two main categories: those managed by the operating system at the kernel-level (native objects), and those managed at the user-level. The difference between monolithic kernels and microkernels is simply how many services the kernel handles: the more “micro” a kernel is, the less it handles.

To properly implement SpaceOS, a strong definition of native objects is required. Similar to how a kernel handles scheduling and inter-process communication by defining what a process is and handling the process life cycle, SpaceOS handles spatial organization by defining what spaces are and handling space life cycles.

The three main objects in SpaceOS are: spaces, entities, and paths. These three objects represent the core elements needed to handle the services SpaceOS provides.



A **space** in SpaceOS is a core object that represents a 2-D location in the real world, defined by a location and a dimension. A name for the space can be automatically generated by the system or assigned by the user when creating a space. Depending on the subclass of the space, it can have additional attributes which are used during runtime.

A key concept related to spaces in SpaceOS is the notion of a process running within a space. In many real-world scenarios, spaces are used to represent a location with an ongoing event, such as a fuel pump or a loading dock. In the case of the fuel pump, the pump would potentially like to put any entity that enters the space into a queue for the pump. A process continuously running within the fuel pump space would thus handle the entry and exit of entities, the queue, and any other task related to the fuel pump space. Processes in general can be related to a space and access resources related to the space.

An **entity** in SpaceOS is a virtual container for an object detected in the real world. These objects can be generated automatically by S-Sensors, or created by the user for objects that serve a more integrated purpose in SpaceOS. Objects, like spaces, can also have processes running within them to perform a continuous function.

A **path** in SpaceOS is a virtual connection between two spaces that relates to a path in the real world that can be taken by entities to move between spaces. Paths can be represented with more than two spaces, but are generally defined by a start and stop space.

All three core SpaceOS objects have subclasses with stricter typing, explored in sections 2.4 to 2.6. Furthermore, users can create user-defined subclasses with different attributes user-level type-checking. The core objects provide a foundation for users to build on top of.

## 2.3. Observability of Space

Not all real-world space is observable and mappable in SpaceOS. The operating system is limited by sensor inputs on what spaces it can detect: actions that occur outside the range of a sensor cannot be detected and updated within SpaceOS.

However, spaces can exist outside scopes of observability, and in many cases will. Consider long paths from one city center to another. WIthin the cities there likely exists multiple cameras with video feeds that can act as sensors, but outside the cities no sensors exist to provide input. The path between the cities can still be represented and manipulated in SpaceOS as it is a real-world space that can be mapped, but it cannot be observed or detected when entities enter or exit the space.

Observability as a responsibility always resides with the higher-level object: observable and controllable spaces are required to handle entities that reside within them, but entities with observable capabilities should never be required to update the space that it resides in.

## 2.4. Space Hierarchy and Typing

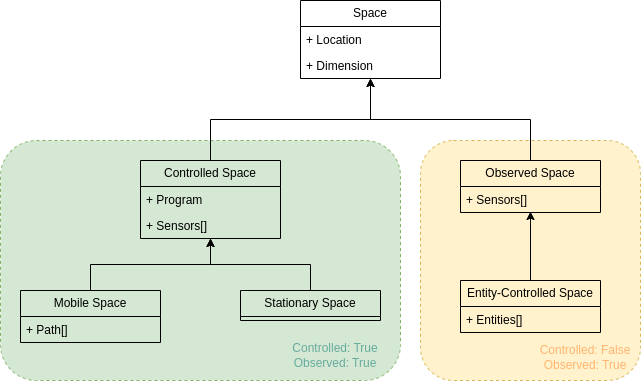


Figure X: Hierarchy of Space in SpaceOS

Spaces in SpaceOS can be categorized into classes from the hierarchy present in Figure X.

By default, all spaces are assumed to exist, as the world-mapping function is one-to-one. However, not all spaces are defined in SpaceOS. Therefore, spaces can be split into two distinct categories: controlled spaces, and non-controlled spaces.

In **controlled spaces**, a program with a predefined behavior is responsible for handling changes affecting the space. Possible changes could be the creation or entry of new entities within the space, or an updated path that now runs through the space. When a change is observed, the program applies the set behavior to the object causing the change. Controlled spaces must be observable as the program needs to observe changes to be able to handle them. Spaces themselves might also be mobile, existing in the real world as some form of mobile container for entities.

In **non-controlled spaces**, no program exists to handle changes affecting the space. The space is still programmatically observed through sensors, but any objects causing the change are not affected when they are associated with the space.

When a space is non-controlled and unobservable, the space essentially acts as only a relational mapping between the real world and the SpaceOS instance. This might occur when paths are defined with unobserved intermediate spaces, or when users are anticipating the creation of a space when linking data from two instances of SpaceOS.

## 2.5. Entity Hierarchy and Typing

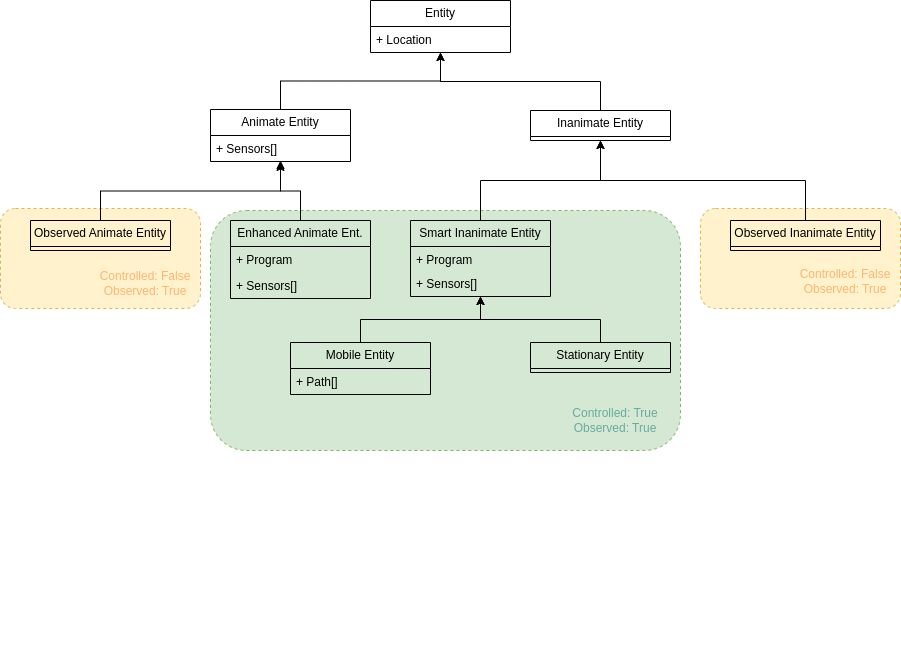


Figure X: Hierarchy of Entities in Space OS

Entities in SpaceOS can be categorized into classes shown in Figure X.

All entities defined in SpaceOS are assumed to be observed.

Observed but not controlled entities are those detected by S-Sensors and automatically created by SpaceOS. There exists no process to control such entities, but they can still affect the behavior of the system. Consider a road with multiple cars, with only one of which being a smart car (an instance of a mobile entity in SpaceOS). The other cars, which are only observed entities, prevent the SpaceOS process running on the smart car from performing certain actions, such as directly changing lanes.

Controlled entities in SpaceOS have processes within them that dictate their behavior to varying levels. A process running on an enhanced animate entity (such as a human) might not be able to tell where the human to move, but may be able to trigger events around the human to occur. Whereas a process running on a mobile entity would fully dictate its path.

All controlled mobile entities generate a trail that represents the walked pathway of the entity. The trails are used by the SpaceBase to determine popular paths that are not yet registered in the system.

## 2.6. Path Hierarchy and Typing

…

## 2.7. Handling Type Rules for all Elements

Types are important in a programming language to provide meaning to data. We use types to associate meaning to bits so we can differentiate between memory address, instruction, character, integer, float, etc. The type system allows us to elevate the bits to have higher levels of meaning.

The subclasses shown in sections 2.4 to 2.6 provide a base for all possible types that can be encountered in a use case of SpaceOS. Users can define new classes that inherit these classes if more specificity is required.

When handling objects in SpaceOS, the following type checks should be performed:

1. Assignment Compatibility: Check if a specified object can be assigned the selected type.
2. Entry and Exit Compatibility: Check if a specified object can access/leave a given space.
3. Space Resource Usage: Check if the specified space can be created based on the resource limits of the system.
4. Space Creation Compatibility: Check if a desired space can be created with the requested configuration in the current system.

# 3. SpaceBase

The main differing factor between SpaceOS and other traditional operating systems is the introduction of the SpaceBase. SpaceBase acts as a management system for spaces represented in some form within SpaceOS. Its role in SpaceOS is akin to the role of a file system in a typical OS, and provides functionality to support space creation and deletion, space organization, and space access control.

SpaceBase is a distributed system that operates on all nodes of SpaceOS. Each individual node within a SpaceOS instance has a SpaceBase running. Through a distributed model, nodes are only required to update their SpaceBases with data relevant to their computational area of interest, similar to distributed database models.

SpaceBase natively offers two main services: A **distributed visualization protocol** and a **node interaction predictor** (NIP). Both services are meant to enhance the capabilities of distributed spatial applications.

The **visualization protocol** captures real-time data on node movements and interactions in the physical world. It maps the physical world to a logical representation for browser visualization. Space Base accommodates various coordinate systems for spatial applications, ensuring compatibility with GPS and others. The protocol faces challenges in distributed provision of space-specific information. Spatial routing directs connections to relevant edge servers without centralized data storage, ensuring fault tolerance and minimal performance impact. Replication and failure recovery at edge servers prevent information loss.

The **node interaction predictor** (NIP) learns from past spatial data to predict the interactions that are likely to occur between different nodes in the system. The NIP relies on node movement history, referred to as "trails," which detail nodes' paths over time. The NIP accesses trails to predict future movements, categorized into "path trails" for mobile nodes and "disk trails" for stationary ones like traffic lights. Trail segments contain spatial data and node interactions, tagged with severity coefficients to denote positive or negative events.

Note that SpaceOS serves solely as an informational service, providing data for applications to make intelligent decisions. Distributed spatial computing applications utilize predictor results, represented as potential fields guiding their actions, including a concept of "data field" following similar principles.

## 3.1 Components of SpaceBase

The core components that make up the SpaceBase module are the world mapping function, S-Blocks, S-Block Servers (SBS), and the S-Block Server Mesh.

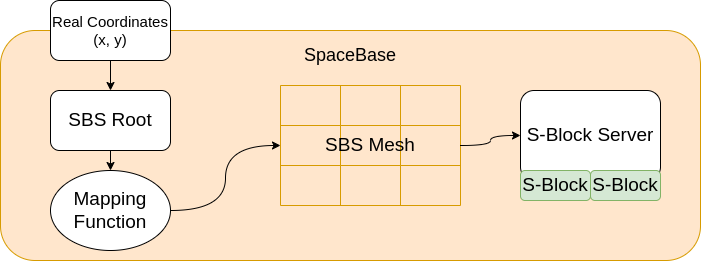


Figure X: Example access flow for SpaceBase

**World Mapping Function**: enables the conversion of physical world coordinates to logical world coordinates, ensuring a standardized world view across different coordinate systems used by applications. SpaceBase provides a generic logical view regardless of the physical coordinate system. Execution of this function occurs directly on nodes for efficiency, though SpaceBase may include a default mapping for GPS coordinates. Custom mapping functions are necessary if the world coordinate system differs from GPS.

**S-Blocks:** represent manageable-sized units of the physical world, discretizing the universe. While they may have fixed sizes within a world instance, SpaceBase adapts S-Block sizing based on context. SpaceBase enables retrieval of S-Block history and facilitates locating the specific S-Block associated with a given physical world coordinate.

**S-Block Servers (SBS):** serve as intermediaries between S-Blocks and applications needing their information. SBS provides MQTT broker channels for node data streaming and output streams for application access. SBS also services S-Blocks grouped into "S-Slabs," managing input and output streams for all assigned S-Blocks. Furthermore, SBS integrates with other SBS in a mesh network for fault tolerance.

**S-Block Server Mesh**: provides a connection between applications and S-Block Servers through linkages between individual SBSs. Applications access SBS through a SBS Root.

# 4. Space Path Graphs

…

# 5. Command Shell for Spatial Computing

Natural Language Interfacing as a concept has existed since the development of system-specific computer languages as a means to interact with computers. It is highly desirable for users to converse with a computer system without being required to understand the programming language which the system is built on or operates with. With the increasing role of computers in our everyday lives, novel methods of interaction between computers and users increase the potential for applications of computer systems.

5.1 TCShell

TCShell is a natural language interface for SpaceOS. TCShell aims to allow for both humans and humanoids to interact with SpaceOS objects and components without the need to understand lower-level SpaceBase design through the use of LLMs. Using user input and 2D-space contexts provided by S-sensors(), TCShell generates system-level code to manipulate objects and spaces within a SpaceBase universe. Conversely, TCShell responds to natural language queries using the inputted natural language.

Figure X: TCShell Design

The user communicates with TCShell via a **Command Interpreter**, which manages all user requests. By default, the system assumes that all user inputs are in natural language format. These inputs are then processed through the **LLM** (Language Understanding Model) before generating a response. However, the command interpreter is responsible for deciding if the user provided commands meant directly for the **Jamscript Command Processor** or the **Spatial Computing Engine**.