Suitability of Game Engine technologies for the implementation of a Spatial Tuple based middleware

Francesco Dente
francesco.dente@studio.unibo.it

Shapour Nemati
shapour.nemati@studio.unibo.it

Samuele Burattini
samuele.burattini@studio.unibo.it

May 2020

Distributed multi-agent-systems greatly benefit from the advantages of the Linda model, however, the latest developments of mobile Augmented Reality applications require an approach that takes space into account as a first-class citizen. More specifically, the lack of a direct and natural support for space-aware and space-based coordination in tuple spaces brought the birth of the theoretical model *Spatial Tuples*, an extension of the basic Linda model where tuples are conceptually placed in regions of the physical world and the behaviour of primitives depends on the spatial properties of the coordinating agents, tuples and the topology of space.

However a fulfilling implementation of Spatial Tuples had not been developed yet, so the goal of this project was exploring the feasibility and suitability of Game Engines as a tool for implementing the spatial abstraction, thanks to their innate capabilities of modeling virtual geometric spaces. The Game Engine of choice is Unity due to its wide use in the academic field. This report presents the development of a prototype middleware based on Spatial Tuples, along with a case study to showcase the capabilities of the solution.

Contents

1 Goal/Requirements

This project is conceived as an exploratory project aimed at better understanding how game engines' capabilities can be leveraged to implement the **Spatial Tuples** coordination model [4]. A more detailed list of **why** this project was realized is presented in the following paragraphs.

Project goals The activities undertaken in this project are aimed at the following goals:

- Evaluation of the suitability of **Game Engines**, in particular Unity[5], as services for geometric computations in implementing Spatial Tuples;
- Study of the **async-await** mechanism and its usage for coding Linda primitives;
- Design and implementation of a Spatial Tuples based middleware that is **topology independent**, which means that it is possible to use different kinds of geometries without changing the system's core.

Project requirements These functional, high-level requirements have been identified as the software that needs to be produced:

- Implementation of a **middleware** that provides a service to use Spatial Tuples as described in [4];
- Implementation of a **case study** to showcase the usage of the middleware in a real-life-like context.

Expected outcomes At the end of the project we expect to have a deeper understanding of the following topics:

- Game Engines' suitability for implementing Spatial Tuples;
- C#'s mechanisms for dealing with asynchronous code, and its usage in the implementation of Linda primitives.

1.1 Problem Definition

In order to achieve the main goals outlined in the previous paragraph there are a couple of problems that need to be investigated and solved.

In this section the key aspects of the project are presented and analyzed to introduce some concepts that will be useful for further explanation.

1.1.1 The Spatial Tuple Idea

As the Spatial Tuples model is founded upon the idea of having the capability of placing information directly onto the common notion of space that the agents share, it is pretty obvious that the first problem to be solved is how to manage the idea of a "**situated tuple container**" which should be strongly bound to space and in which agents could put or get tuples.

The easiest and most obvious way to do so is to model a concept of a space **region** in a way that keeps both the geometrical definition of its boundaries and the tuples that are stored within that region.

Thanks to this abstraction, Linda's primitives can be envisioned as a query on a space region that can be identified by its name (if the agent has a prior knowledge of the world) or by its geometrical definition. Because of the **suspensive semantic** of some interrogations, the query itself has to be stored inside the modelled concept of region: in this way the **request** can be satisfied either when the matching tuple is added onto that region or on any overlapping one.

Storing the request as well as the tuples inside a region is fundamental for another key mechanic that involves situated components **being able to move** freely the space. Their "body" can be seen as a region, so that tuples can be put onto a component. This means that the tuple can move and so should a request to express the full potential of the Spatial Tuples coordination.

Finally, a concept of "**stickiness**" is needed to specify whether or not a query on a region should follow the moving region or not.

1.1.2 Creating a middleware through the Game Engine

Since the main purpose of the project is to test the potential of Game Engine technologies when dealing with the spatial/geometrical nature of the Spatial Tuples model, a key problem to be solved is how to obtain a middleware using a technology which has not been thought for that goal.

Most Game Engines, if not all, work on the abstraction of a game-loop that keeps all the elements in the virtual world constantly updated, therefore having a virtual time that is shared inside the simulation.

As explained in the previous paragraph, the spatial nature of the model requires the definition of the concept of regions as first-class citizens of the world, that can change their features through time, like their position, and as such also interact by overlapping and exiting from each other.

This means that regions have to be modelled as entities which are updated from the aforementioned game-loop, in particular they need to be subject to the physics update in order to receive collision notifications from other entities of the system.

To sum up, independently from the Game Engine of choice, in order to work properly the middleware must map the concept of region onto a "physical" object. Doing so also means that, provided a bridge that enables this mapping, changing platform should be quite easy and this will be kept in mind when designing the system infrastructure.

1.1.3 How Unity works

The Unity Engine provides an environment for games, simulations, and other virtual experiences.

The engine is written in **C++**, which grants an highly effective implementation of the core functionalities, while application code is written in C#. In particular, Unity exposes two mechanisms for interacting with the core:

- **Lifecycle hooks**: each *Unity Script* executes a number of C# event functions in a predetermined order;
- **Function calls**: a Unity Script can call engine functionalities from within C# code.

To better understand how that works we will go through Unity's primary abstractions and its **Game Loop**.

Unity's abstractions The environment in which entities behave is called a *Scene* in Unity's terminology, and consists of a 3D context containing a set of *GameObjects*.

The GameObject is the most basic entity of Unity, characterized by a name and a set of *Components*, and organized in an hierarchical structure, with other GameObjects possibly nested beneath it. Each GameObject has at least one component: its *Transform*, which carries information about position, rotation and scale with respect to the x, y, and z axes. Many components are included in the game engine, providing a wide range of frequently used features, but most importantly developers can write their own components, also referred to as "scripts", extending the `MonoBehavior` class.

Unity's Game Loop Unity, like most game engines, runs using a game loop, which is the core **design pattern** of video-game programming. The game loop is aimed at keeping the simulation/game alive and making the game time match the real time, proactively performing some operations many times per second, instead of just reacting to the user's input. In short, the game loop is a series of operations to be carried out in a specific order, that are performed while the game is alive, separated by a configurable amount of time. Not all phases are treated the same: the physics updates are crucial for correctness, and are handled by the loop in a way that guarantees they are executed an exact amount of times per second, while others – such as rendering – can be performed a variable number of times each second.

Unity's game loop is very articulated, so here we will just give a not-too-technical overview of its main steps.

Unity has a setup phase called only once when the **Scene** is first loaded. Here, all of the GameObjects' scripts call their `Awake` and `Start` methods, where the initialization code should be put.

After that the real loop starts, performing the following steps:

1. **Physics update**: at the beginning, the `FixedUpdate` method is called on all the scripts, then physics is computed and finally collision events are fired;

2. **Input events**: the game engine buffers all input received during the other steps, and at this stage calls the appropriate handlers;

3. **Game logic**: when writing software for Unity, most of the code falls under this category, performing all the business logic exploiting the physics and input results to act on the environment, calling the `Update` method on scripts;

4. **Rendering**: the components that provide rendering functionalities are called in this step, when the new frame is build and displayed on the screen;

5. **End of cycle**: when all the actions of the loop are performed, the engine waits for a configurable amount of time to pass before it cycles back to the physics update. This decreases the high workload that the Unity application has on the host machine.

As a final note, for a good balance of performance and correctness, Unity always performs the physics updates, whereas rendering and game logic steps can be delayed, which might result in decreased smoothness without breaking the physics behavior. In the following link a more complete and detailed descriptions of the execution order of Unity's game loop is provided: `https://docs.unity3d.com/Manual/ExecutionOrder.html`

1.2 Scenarios

As the prototype middleware created for this project implements the vision and features of the Spatial Tuples model, the main application scenarios are linked to the exploration of such model for **research or application development purpose.**

The "framework" proposed is suitable for developing every application that needs a coordination system based on both space awareness and shared knowledge. To do so, the middleware models space in a virtual world which then has to be mapped onto the real one.

Potential users may want to run **simulations** with multi-agent systems that need to operate in a common space without bothering to manage the interaction with space itself.

Because of their nature, **Augmented/Mixed Reality** applications are one of the kind of software applications which could benefit more from the use of this system. The logic of the Spatial Tuples could provide a very powerful abstraction tool to model them.

The fact that the produced middleware is open to external applications with the use of **Web APIs** allows to use it with any agent modelling language that supports HTTP libraries. The software produced also includes a C# Client implementation which could be very useful for developing prototypes and simulations trough the Unity Game Engine, which was used to develop and test the system itself.

The Game Engine integration is hidden from external users, meaning that they won't need to know anything more than the agent technology and the topology of the space they want to model to use the middleware and develop their own application.

1.3 Self-assessment policy

For what concerns the assessment of the project, we expect to produce a solution that will guarantee a certain **level of quality** from a software engineering point of view. Our goal is to design an architecture that should be as **extensible**, **general** and **independent** from the current technological choices as possible, while still being a fully correct implementation of the Spatial Tuples model. Therefore, our criteria to evaluate the final outcome will consider these properties as the main targets to achieve, leaving efficiency concerns as a topic for future improvements of the system.

To measure the degree of fulfillment of these objectives, we will base our judgement on the results of unit testing as well as on the ease of development of the case study.

2 Requirements Analysis

The requirement of producing a middleware that allows its users to operate on the Spatial Tuples abstraction hides the real goal of testing Game Engines' suitability for this purpose. This additional requirement imposes a **technology-lock** that impacts on the way the system will be designed.

Another core aspect of the software is dictated by the fact that the requirements are specified through the Spatial Tuples paper, so the underlying **model** must be the one described in [4].

The one aspect left to the analysts is the programming **paradigm**, which is still strongly influenced by the previously deduced technology and model. Since Spatial Tuples is an extension of the **Tuple Space** model, a lot of the considerations about the paradigm used for many of the Tuple Space implementations hold here as well. Since an effective approach to the implementation of this model is the **Object Oriented** one, and the spatial dimension of the extension along with the abstractions used by Game Engines pair well with it, it has been chosen as the project's paradigm.

With these three elements in place some **abstraction gaps** emerge while describing the system using the available concepts. In particular, a distributed system needs to **expose its services** to clients by means of some specific technology, for example Web APIs, but these might not have a 1-on-1 correspondence with the abstractions of the Linda model. Another abstraction gap to be closed - or at least reduced - is the one between the physical world that needs to be represented and the **Virtual Space Geometry** that must model it in the middleware.

Some words about if and how the model actually meets the required coordination effectiveness would usually be spent, but since the Linda and Spatial Tuples models have already been deeply discussed in other papers - most notably [1] -, this topic can be skipped here.

An important aspect of Spatial Tuples is being able to **locate** a tuple returned from a request, but the model doesn't specify whether the tuple position should be provided as additional meta-data alongside the tuples themselves as result of some primitive, be part of the tuple, or be explicitly asked to the middleware that keeps track of the mapping.

This loose requirement must be translated to a more specific one, choosing a strategy that pairs well with the previously specified constrains.

Given these premises, the two following subsections describe the analysis of specific functional and non-functional requirements.

2.1 Functional Requirements

When compared to the Linda model, Spatial Tuples has the peculiarity of dealing with space, so the system must have the capability to model it. In particular it needs **CRUD** operations to manipulate it:

- **creating** a region;
- knowing if two separate regions are **intersecting** with each other at a given time;
- **changing** the shape and position of a region;
- **deleting** a region.

These build the foundation on which the virtual geometric space lays to achieve the behavior described in the requirements.

For a distributed system to work, there is the need of setting up some form of remote communication that can be used by different clients. This project is targeted to a **broad** range of devices and technologies, so it needs to be as open as possible, exposing its services by means of some widely adopted **standard**.

The last functional requirement that needs a more formal description is about the spatial information that is given along with the tuples as results of primitives. When performing a primitive that replies with one or more tuple, the user of the middleware expects some spatial information about the tuple(s), and that is the spatial information of the region of space where that tuple was first assigned to. This is the most accurate option since tuples are always assigned to a region that could be as simple as a single point, and in that case its exact position would be provided. Otherwise the tuple might be in a more semantically significant region, which pairs well with the philosophy of Spatial Tuples.

2.2 Non-Functional Requirements

The development of the system poses some non-functional requirements that need to be defined to have a better understanding of the capabilities of the middleware.

Since most of them are related to aspects of **performance**, and this project is focused on the creation of a first prototype, these requirements have been kept in mind when designing the overall infrastructure, but not fully addressed when implementing, leaving the improvement of performances to **future works** (Section 8.2) that should be able to operate on that with little effort.

One of the key aspects of a middleware for multi-agent systems is having an open system which should be able to **scale** with both the number of requests and agents that interact with the coordination model.

This is true for every open system, but raises some peculiar problems when dealing with the spatial nature of the model and in particular with the Game Engine underneath.

In particular, the **virtual space computation** should be executed with the minimum possible effort and maybe even be itself **distributed** between multiple nodes running instances of the Game Engine software on different areas of the same virtual environment.

3 Design

In this section we will give an overview of the high-level design of the middleware and the main components that have been developed during the project. The description is subdivided into the three major axes of design of software systems: *structure*, *behaviour* and *interaction*.

3.1 Structure

The static structure of the system can be tackled from different abstraction levels, each outlining different aspects of the produced software.

3.1.1 High-level decomposition

From a modularization point of view, the middleware has been decomposed into several components, each contributing to a specific subset of the global functionality. Each component's interface has been designed with the goal of ensuring a high level of decoupling, to allow for a great flexibility in the deployment phase. In the early phases of design, the degree of uncertainty about the final deployment structure was, in fact, pretty high, therefore a tight coupling between the core parts of the middleware could have led to an unmanageable result.

At the end of the design process, three major components have been identified (Figure 1): the `ApiService`, the `Environment` and the `PhysicalSpaceRepresentation` components. The central piece of the architecture is the `Environment`, the component responsible for the execution of primitives and the system state storage, which consists of information about tuples and regions. To that purpose, the `Environment` keeps a **virtual representation** of the space on which the middleware operates, and delegates any **physical computation** (e.g. collision detection) to an external module, hosted by the Game Engine: the `PhysicalSpaceRepresentation` component.

The satisfaction of the system openness requirement is assigned to yet another module, the `ApiService`, which acts as the main entry point for external agents to interact with the running system. Mainly, the `ApiService`'s task is to gather requests from the outside world and route them to the `Environment`, managing and abstracting from client communication concerns and handling concurrent access to the `Environment` itself.

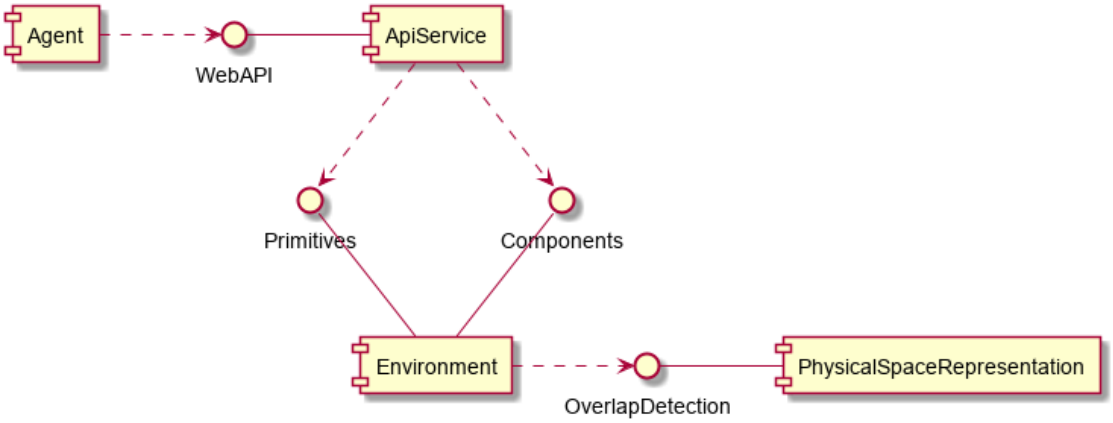

Figure 1: UML component diagram of the Spatial Tuples middleware

3.1.2 Core interfaces

We can then analyze the system structure at a deeper level by delving into the object oriented decomposition.[1] One key decision made during the interface modelling process was to have a strong separation of the concepts that needed to be visible by the clients of the middleware from those that must only exist as implementation details.

Having the `Environment` as a single point of access to the middleware for agents to interact with the world they operate in makes it possible to abstract from the distributed nature of the system and from the specific strategies adopted to implement the Spatial Tuples logic. To this purpose the same core interfaces should be used in both client and server to have a seamless interaction between nodes.

Those interfaces are captured by Figure 2, representing the public interface shared between agents and the web server to interact with the software. Due to the distributed nature of coordination services, it is important to provide the users with non-blocking access to the required data, to avoid all kinds of issues that network latency might cause. The public interface of all possibly blocking operations is, therefore, made asynchronous by using *Promises*[2] as the return type of those operations.

The heart of the system resides again in the `Environment` interface. This concept encapsulates all implementation details of the service and provides operations to work on them by means of **methods to populate space** and **primitives**. The Spatial Tuples model comes with a large set of primitives inherited from the Linda model and, on top of that, adds several ways to configure each of them, thus requiring a clean way to **select a primitive** to be run and to **setup its spatial target and behaviour**. If not done in a thoughtful way, this could lead to an explosion in the number of methods to satisfy each possible way to configure the request itself and therefore end up in a

[1] The class diagrams shown in this section aren't based on any particular programming language, therefore they do not match the presented source code.

[2] We use the Promise type since it is a concept used by many languages and paradigms to express asynchronous calls. When translating diagrams to C# code, those will be replaced by the `Task` type.

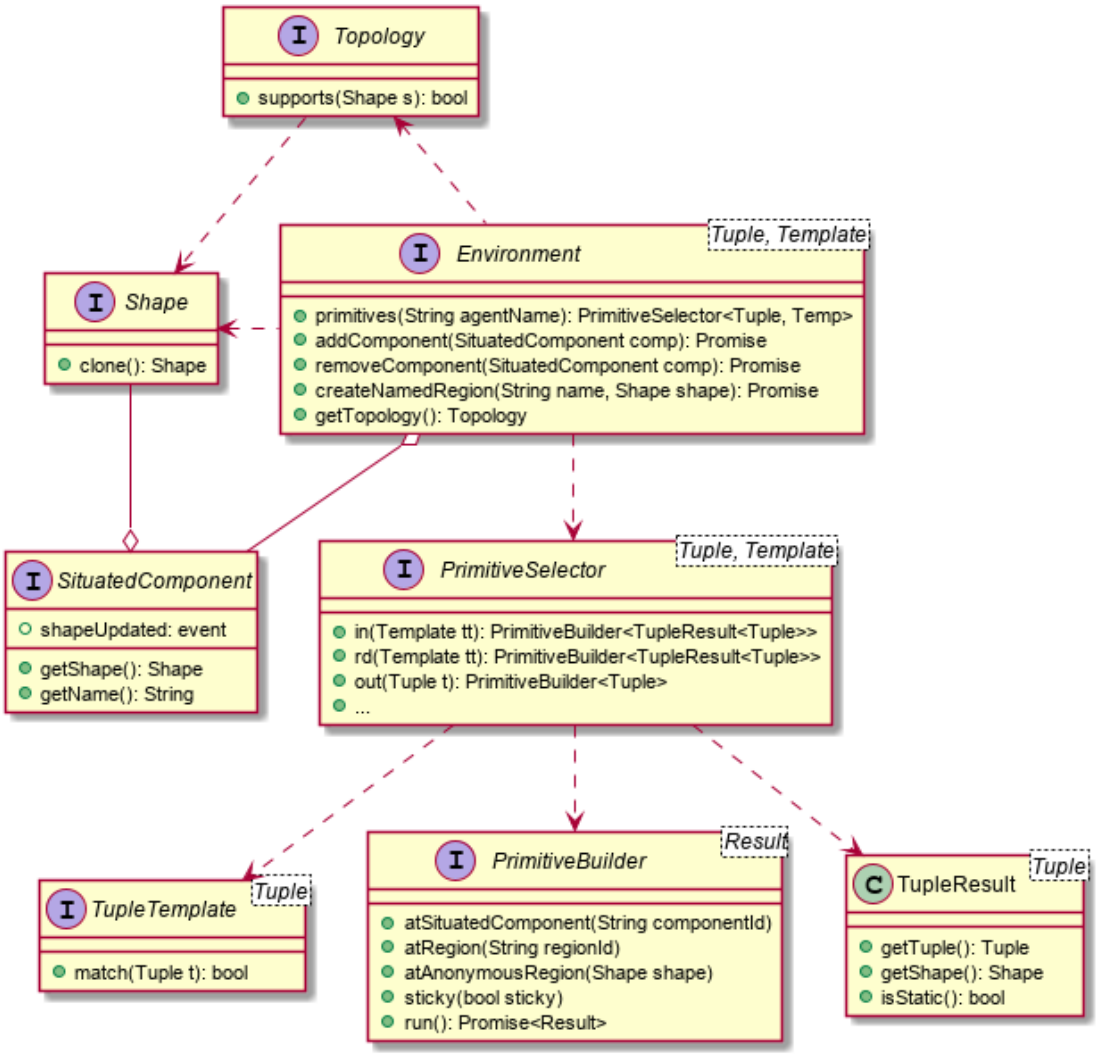

Figure 2: UML class diagram for the core interfaces of Spatial Tuples

messy code base, with lots of repetitions. Our solution to this problem leverages the builder pattern to provide a **fluent** setup of the primitive's target. To that purpose, the `PrimitiveBuilder` interface has methods to choose the target region from a certain number of sources and to decide on the stickiness of the primitive, invoking the `run()` method to actually execute it. This mechanism allows user code to perform primitives like this:

```
environment
    .primitives()
    .out("hello")
    .atSituatedComponent("myComponentName")
    .sticky(true)
    .run();
```

When developing systems like this, it is important to be as generic as possible and make no strong assumptions about the needs of an hypothetical consumer, to adapt to each individual requirement with little or no changes. There are two main aspects that were introduced to make this possible: **tuple domain independence** and **space topology independence**.

The first one can be obtained by making each interface that deals with tuples **generic** in the type of tuples themselves and in the type of template used to match them. This way, each different application scenario is free to change the tuple domain on which it operates, without altering the structure exposed by the service.

Space topology independence is instead obtained by encapsulating the geometrical definition of regions in the concept of `Shape`, which is then validated by the environment's `Topology`. By leveraging this mechanism, the `Environment` is able to accept several types of shape, while still making sure that they are the ones **supported** by the current topology specification, which might be **arbitrary**. For example, an `Environment` instance might specify that only three-dimensional shapes are allowed, rejecting any attempt to create two-dimensional regions. Using a traditional Game Engine, however, the theoretical advantage of arbitrary topology support is limited to 2D and 3D only.

On top of this geometry framework, we define the interface that represents the `SituatedComponent`. As described in the former sections, a `SituatedComponent` is an entity living in the simulated space that can change its position and shape over time, notifying each update to the environment. This is the real concept that outlines the difference between the static nature of Linda and the dynamicity of Spatial Tuples: having a component that can move freely in the "tuple space" means that its interrogations can follow it and be eventually satisfied whenever the component enters in (or exits from) a specific Region.

3.1.3 Internal server structure

While the previous section describes the design of the system from a service consumer point of view, this section is aimed at giving a deeper explanation about the internal

mechanisms of the produced software and its relation with the Spatial Tuples model. As mentioned before, the `Environment` is responsible of keeping the entire state of the system during its execution and allowing access to it via primitives. This means that it must keep track of regions that get created, destroyed or updated, of tuples that get added or removed, and of links between the two, making sure that this data is accessible whenever needed. Our approach, shown in Figure 3, uses a hierarchical structure having the `Environment` as the root.

As previously discussed, any portion of space that is referenced at runtime, being it a static `Shape` or a `SituatedComponent`, is represented as a `Region` in the system. By giving regions a strong identity in the context of Spatial Tuples and making them able to change shape over time, we are able to attach information on those instances knowing it will be persistent for the lifetime of the region itself. We took advantage of this idea to fulfill the main requirements of the model, linking tuples to changing-over-time portions of space and keeping track of suspensive primitives until they complete. In particular, `Regions` store a collection of `PendingRequests` to keep track of primitives that are waiting for some condition to be met. `PendingRequests` get notified whenever tuples are added or removed from regions they are attached to, and react to those events, possibly completing the pending primitives they represent.

The life cycle of regions is handled by an entity under the `Environment`'s responsibility, named `RegionManager`. It is responsible of maintaining the virtual representation of space introduced in Section 3.1.1, that is persisting `Regions` and keeping them up to date with any changes occurring to the linked entities. Along with this, the `RegionManager` also has to act on the physical representation, since it is not able to determine by itself if two regions are overlapping with each other or not. It does so by delegating this concerns to the Game Engine, using the `PhysicalSpaceConnection` interface for communication. It is worth noticing that this is the only point of contact between the core module and the Game Engine. As the system will need to scale in performance, multiple instances of the Game Engine might be used to split the load. Having a single point of contact with the Game Engine is an anticipation of change that allows this transition to be as smooth as possible.

3.2 Behaviour

This system operates in a way that enables the many instances of its **tuples**, **primitives**, **agents**, and **regions** to behave according to their semantic. These all come together, providing what could be defined as an **emergent behaviour**.

In this subsection we will discuss the behaviour of the components presented in 3.1.1, explaining how the system provides its essential mechanisms at an high abstraction level. Each component will be described by its possible states, based on the information it carries, and listing the possible events as transitions that the component will react to.

Agent Each agent can either be situated or non-situated. Either way an agent starts as non-situated, with the possibility of registering itself to the middleware and thus becoming a `SituatedComponent`. To do so, the agent must provide some information

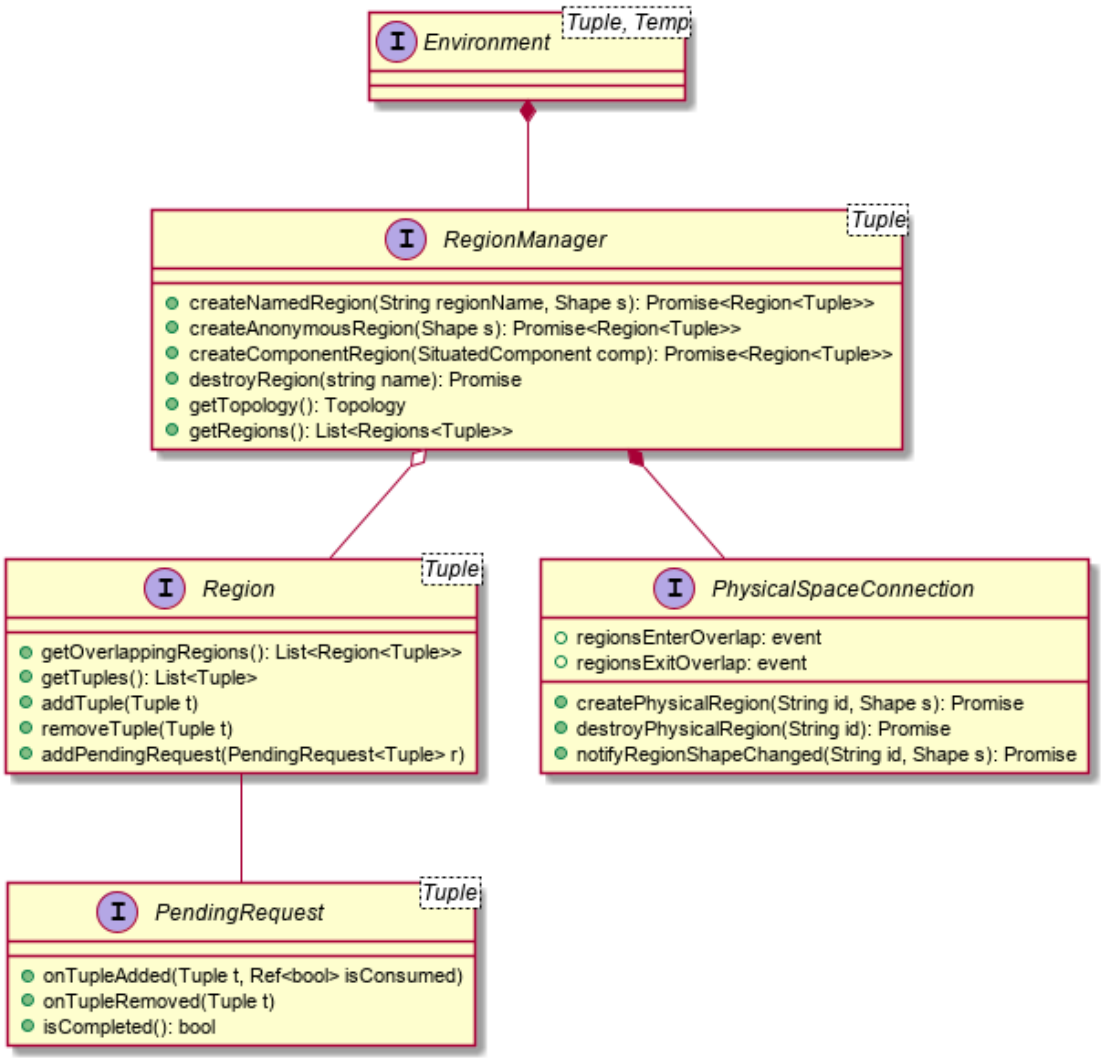

Figure 3: UML class diagram for the server implementation of Spatial Tuples

about its shape and starting position and, after that is done, it gains the possibility of updating that information with requests to the server.

At any moment an agent can perform a primitive operation, which is considered suspended until a response is received.

ApiService This component's state is pretty simple as it is only based on the state of the requests it handles. When a new request is received, it is set as suspended, and when a response is ready, the request is solved and the response forwarded to the client.

Environment and RegionManager As these two components belong to the same module they are described together. Their state is based on the sets of regions, tuples, and pending requests currently active on the system. Other than just a collection of these, the system keeps track of the relationships between these elements, in particular each region is paired to:

- **Overlapping regions**: for each region it holds the set of the regions that have a non-empty overlap;
- **Situated Tuples**: each tuple is associated to exactly one region;
- **Pending requests**: each request is associated to exactly one region.

Whenever the messages *RegionOverlapEnter* and *RegionOverlapExit* are received, the state is changed accordingly, adjusting the sets of overlapping regions. When that happens, the system checks whether the pending requests on the regions involved in the state change can be solved, and if it's the case those are removed from the pending state, and the `ApiService` is notified.

Something similar happens when a new tuple is submitted: the system checks whether there are any pending requests that could be satisfied by that tuple, either in the region it is being submitted to, or in any of its overlapping regions. If that's the case the pending request is removed and the ApiService gets notified. If it was an **in** primitive, the tuple is not added, otherwise it is assigned to the target region. When a tuple is removed by an in primitive, all the pending **no** requests are checked for possible completion.

The same behaviour is mirrored when requests are being added or removed.

Physical Space Representation The state of this component is based on the physical representation of the regions. It acts in a **reactive** way when communicating with the `RegionManager`, but has a **proactive** behaviour for the management of region collisions. In particular, the proactive behaviour is based on the *Game Loop* pattern, which consists of several updates per second in which all the physics computations are executed, instantiating new geometries and checking collisions.

When a new region is added, in the following physics update of the Game Loop the Game Engine checks whether it collides with any other region, emitting a *RegionOverlapEnter* event. Likewise, when a region is moved or removed, during the following Game

Loop step, the Game Engine checks which previously overlapping regions are no longer colliding with the moved/removed one, emitting *RegionOverlapExit* events in that case.

3.3 Interaction

Entities' interaction is a central matter within Spatial Tuples. Due to its nature, most of the computation is triggered by events coming from several sources, handling which involves multiple components that cooperate with each other. In the remainder of the section we will describe, leveraging UML sequence diagrams, some of the major scenarios showing the basics behind our implementation of the middleware. Even if this is not a comprehensive list of all the possible interactions happening within Spatial Tuples, they give a basic intuition on the inner workings of the system. In particular, we decided to show the interactions required to manage the **SituatedComponent life cycle** and to perform **out and in primitives**.

3.3.1 SituatedComponent life cycle

As mentioned before, `SituatedComponents` play a fundamental role in this context, due to their dinamicity. During their life cycle they proactively move inside the `Environment`, producing and being subject to intersections with static `Regions` or even other `SituatedComponents`. These scenarios, shown in Figure 4, start with the `SituatedComponent` registering to the `Environment` with an `addComponent()` call. This call is immediately delegated to the `RegionManager`, which creates the internal representation of the `Region` object associated with the component and takes care of sending the *create* command to the Game Engine, through the `PhysicalSpaceConnection` interface. Before this last step completes, the Game Engine is in charge of computing all possible collisions that might involve the new object and notifying them to the server. This is needed to make sure that, after the procedure completes, the `Region` object on the server is up to date with all the overlap information.

Once the registration process is over, the situated component is able to proactively move in space by notifying the `RegionManager` of **shape changes**. Upon notification, the *RegionManager* updates its virtual region representation and asynchronously[3] informs the Game Engine of the change via the *update* command. This can trigger new collisions inside the Game Engine, that must be promptly notified to the server.

When the component needs to be removed from the `Environment`, the `removeComponent()` method has to be called. Like the `addComponent()` counterpart, the invocation is delegated to `RegionManager`, which in turn instructs the Game Engine to deal with the deletion of the physical region. Once this last call terminates the `RegionManager` takes care of deleting the virtual instance of the `Region`, completing the entire process and ending the life cycle of the `SituatedComponent`.

[3] Asynchronously here means that the request is not suspended until the real change happens, but completes immediately

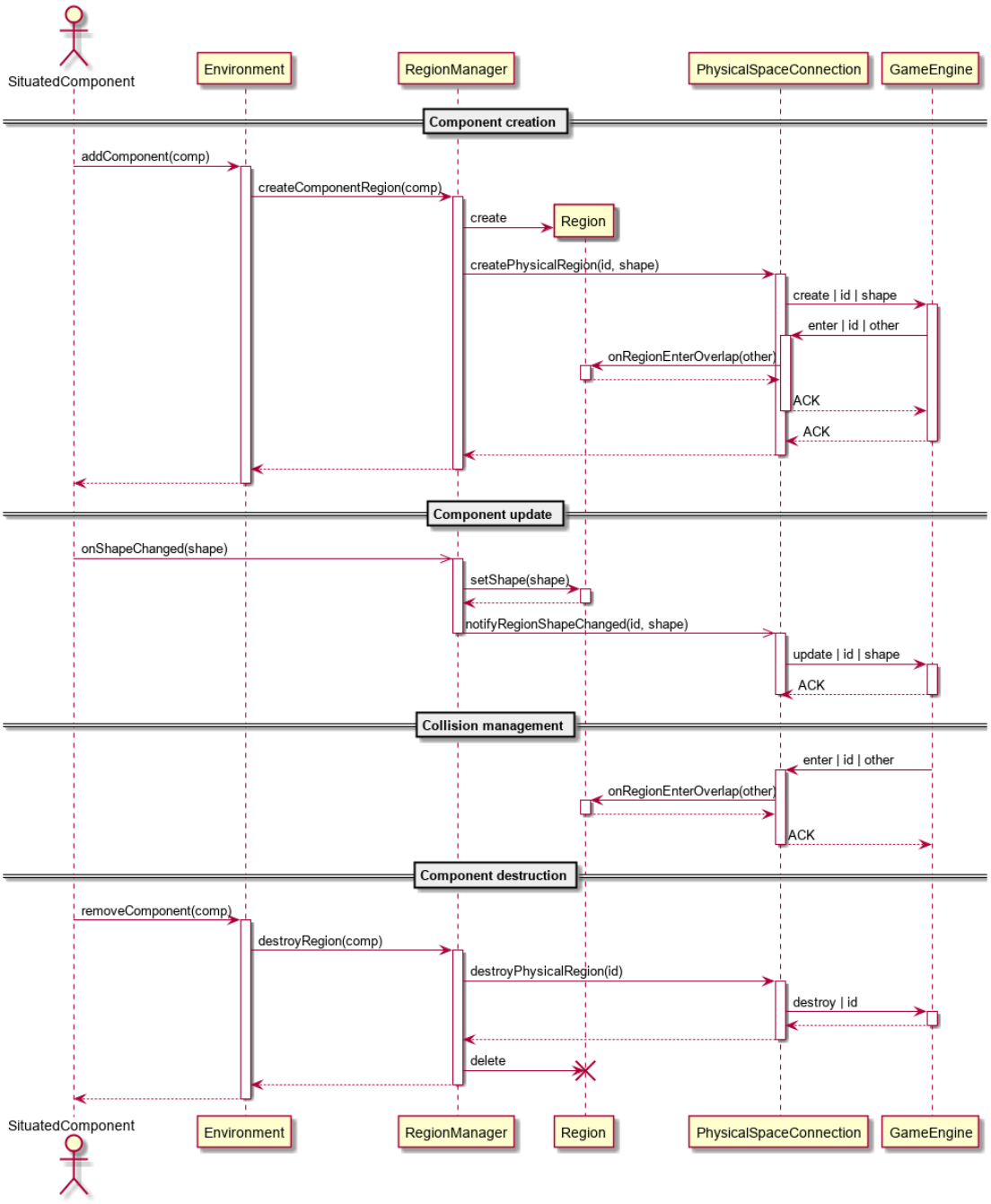

Figure 4: UML sequence diagram describing the interactions of a Situated Component with the system during its life cycle

3.3.2 Execution of an out primitive

The interactions required to perform an out primitive are shown in Figure 5.

An agent initiates the request through the `Environment` and selects the target for the primitive by using the fluent API exposed by the `Environment` itself. Once the `run()` method is called, the selected target is translated to a `Region` instance, which is created or retrieved in a different way depending on the requested configuration, then the execution is started.

The out primitive starts by passing the specified tuple to the `addTuple()` method on the target `Region`. At first, the tuple is not actually stored in the tuple collection, since that region or an overlapping one might have pending requests that could possibly consume the out'd tuple. Therefore, a notification is sent to the target region and to regions with which it overlaps. Every region (including the target) handles it by performing a check on all of its pending requests, using a shared variable indicating whether the tuple is consumed or not and possibly completing some of them.

At the end of the notification step, the tuple is actually inserted in the `Region` only if no pending request consumed it.

3.3.3 Execution of an in primitive

The logic required to perform an in primitive is slightly more complicated, due to the suspensive semantics.

The process starts when the agent queries the `Environment`, choosing the target `Region` to execute the primitive on, similarly to how it would do for an out. After that, the `Environment` asks the `Region` to retrieve a tuple matching the given template and remove it in case any is found. If this operation succeeds, the result is immediately returned and the operation completes, otherwise the process is repeated for all overlapping regions, until a tuple matches the given template.

After looping through all neighbouring regions, if none of the tuples is able to satisfy the given template, the request must be suspended. This can be done by storing a pending request inside the target `Region`, to be notified whenever an out primitive inserts a matching tuple.

4 Implementation Details

Implementing the middleware has proven to be a challenge on different levels. Even though we've gone through a long design phase, there were some non-trivial aspects that took us time to figure out, as well as some choices that needed to be taken in order to map the abstract model onto a "concrete" software project.

Overall, the implementation followed an iterative process, initially developing the main features and only later expanding the middleware structure with the design presented before.

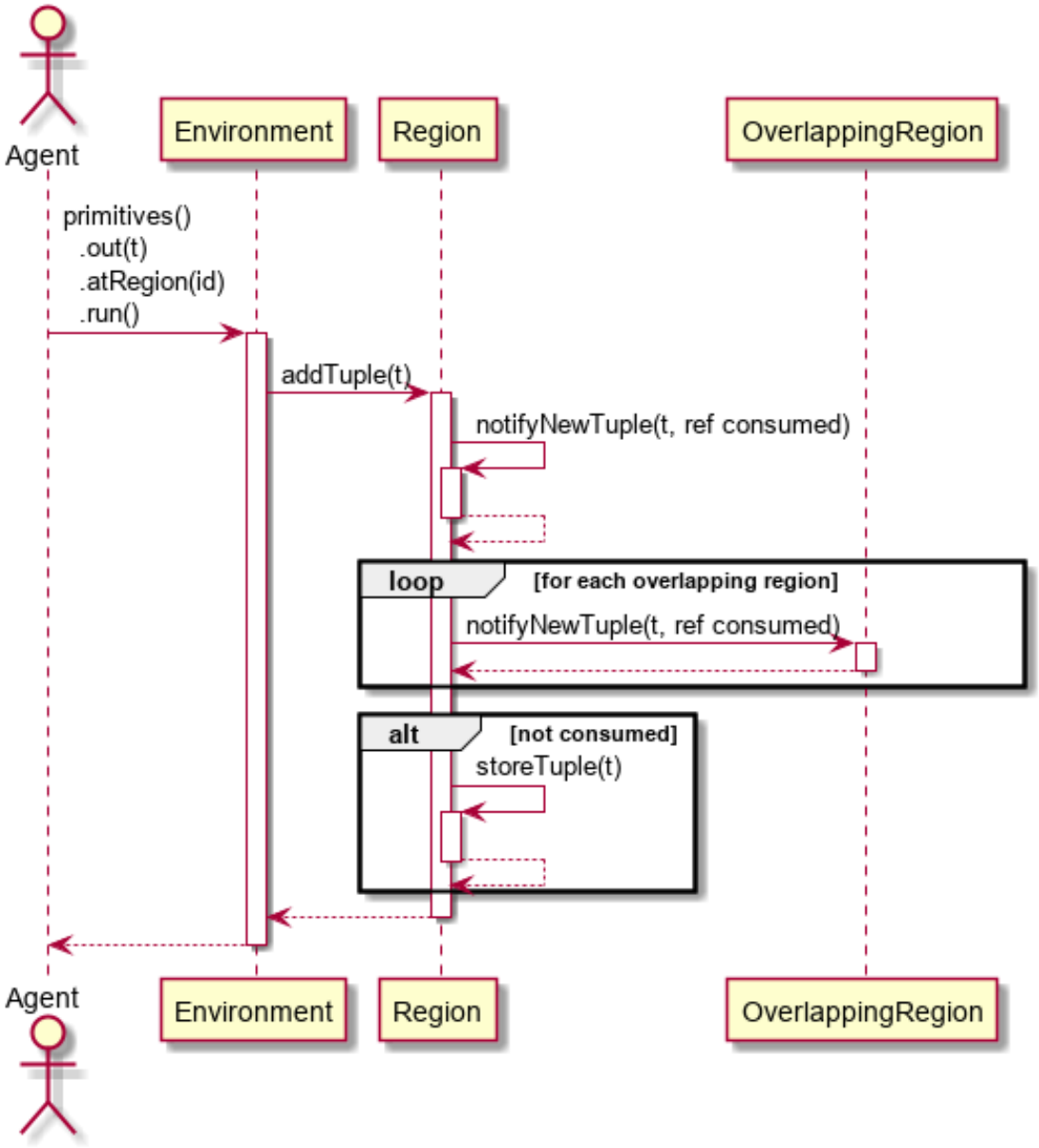

Figure 5: UML sequence diagram describing the execution of an out primitive

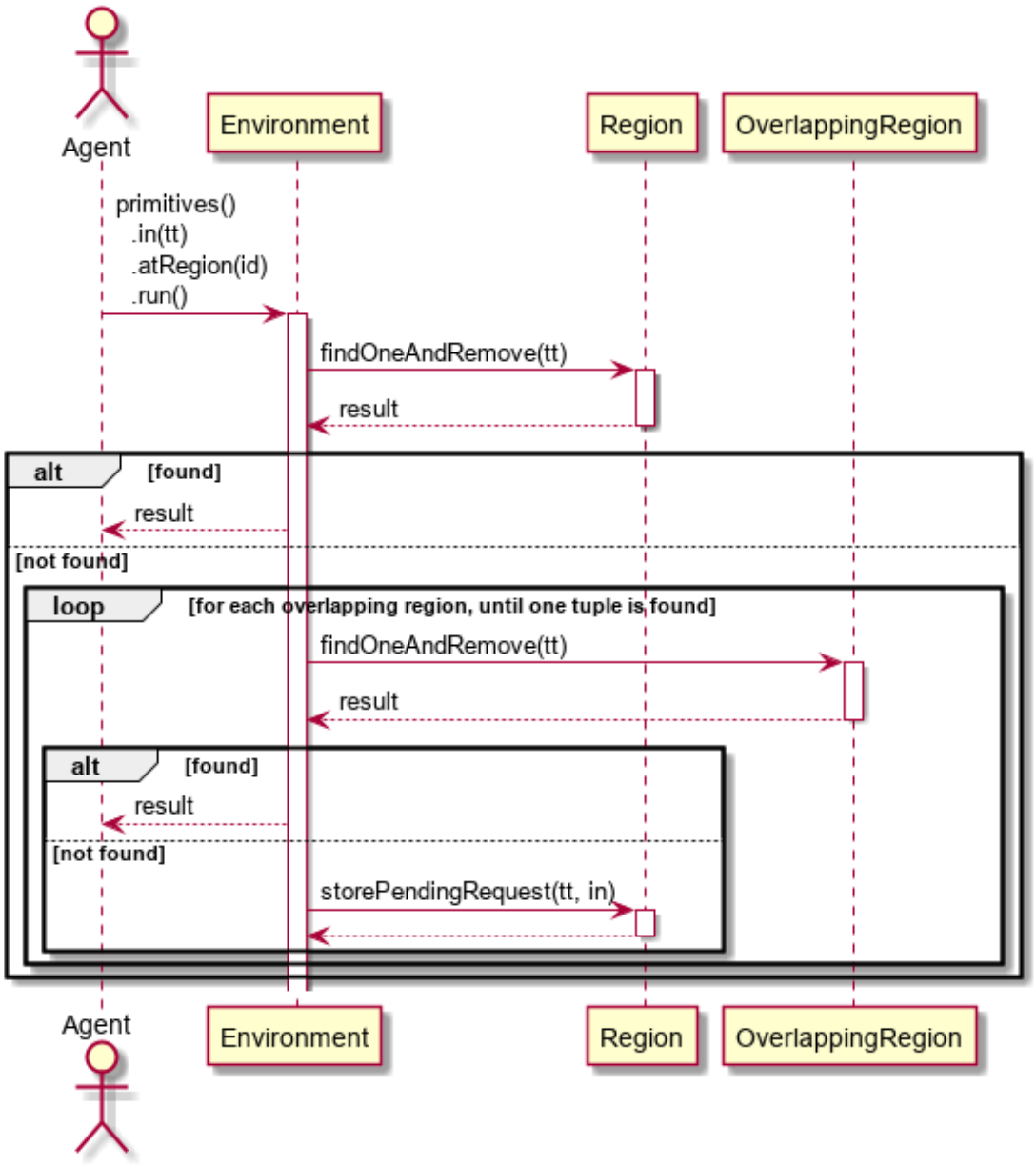

Figure 6: UML sequence diagram describing the execution of an in primitive

4.1 Web APIs

The choice of using Web APIs was made to satisfy the requirement of making an open system easy for agents to interact with. We strongly believe this is the right choice because most languages have access to some kind of library that makes HTTP requests easy and also because it's a great way to describe all the possible operations exposed by the system.

Furthermore, even if not implemented in this first version of the middleware, it could be possible to use the features of the HTTP(S) protocol to enforce an **authorization system** in order to interact with the middleware, which could be really useful in real life applications.

Although `GET`/`POST`/`DELETE` methods of the HTTP protocol can easily be mapped onto Linda's out/in/rd primitives, their "natural" behaviour in a web service environment misses the fundamental **suspensive** semantic typical of the coordination system.

To match the nature of Linda based tuple spaces in an easy and effective way, we chose to use an **infinite timeout** request. The HTTP request is stalled and waits for completion which could never arrive. This feature may need to change to improve the system performances and scalability, but it's good for now.

For pragmatic reasons, the APIs also expose additional functionality, such as the creation of a named region, as this could be useful to setup the environment with regions of which the agents may have knowledge by default. To see the full interface exposed by the system see the swagger API attachment in the documentation folder.

4.2 C# Asynchronous APIs

The Linda primitives have an asynchronous nature, so the use of some high-level APIs to approach asynchronous programming can lead to a clean implementation. Another aspect that calls for a good amount of care is the middleware need to deal with multiple users at a given time, and heavily relies on I/O operations.

In C# the reference **asynchronous pattern** is the *Task-based Asynchronous Pattern* (TAP) [3]. One particularly elegant aspect of the TAP is the `await` operator, which can be used in an `async` method before a Task invocation to suspend the current method, until a result is yielded, all without any complex asynchronous syntax. While the user of the `await` keyword writes code that looks synchronous but behaves asynchronously, the framework is setting a callback on the task by using a continuation. The callback that gets created resumes the asynchronous method at the point of suspension, and if the awaited operation was successful a result is returned, otherwise an exception is raised.

Leveraging the conciseness of the `await` operator, asynchronous code that has the same structure of synchronous code has been written, avoiding the problems of dealing with low-level synchronization mechanisms or of having code that is difficult to read because of problems like pyramids of doom, asynchronous spaghetti, etc.

Delegating all the concurrency problems to the async/await mechanism alone can lead to race conditions when accessing shared data structures, so we used a `TaskScheduler` which ensures that tasks always run exclusively. It is provided by the `ConcurrentEx-`

clusiveSchedulerPair C# class, one of the utilities offered by the language. When using it we can specify whether we want a Task to be run concurrently or exclusively, which greatly synergizes with the `async` mechanism while solving the aforementioned technical problems. Note that this might not be the most efficient strategy, but provides an high level of abstraction (opposed to low-level locking mechanisms) and doesn't require substantial changes in the code if the need of altering this behaviour rises in the future.

4.3 Unity

A significant part of the implementation relies on Unity, which needs to cooperate with the rest of the middleware in order to achieve the correct behaviour of Spatial Tuples.

Firstly, Unity has to communicate with the Core component that stores information about `Regions` and the tuples connected to those in order to provide a geometrical materialization and compute the spatial interaction. This communication was implemented making use of a **TCP socket** channel that makes the quick exchange of messages possible maintaining the order and avoiding the overhead of an application protocol. It was chosen in this scenario that needs to be extremely light because the presence of the Game Engine bridge should appear seamless to the core components. Since data is sent directly over the transport protocol, a simple syntax was created to correctly deliver messages and parsing the information needed by the two components.

With a communication channel established between the core and the Game Engine, they can now exchange data about regions, and so materialize them as Game Objects capable of physically interact within the virtual space.

We chose to create a Prefab[4] that was equipped with a `Rigidbody`, the component that makes objects being considered by the Physics engine inside Unity. The `Rigidbody` was set to be kinematic since the only interaction that needs to be checked is the collision with other objects and regions should not alter their movement when colliding.

While most of the setup described above can be done through the Unity editor and saved directly as a prefab, the `Collider` component representing the shape of the region has to be added at runtime when instantiating the Region Game Object. The specific type of `Collider`, along with its configuration, is dynamically chosen at runtime based on the information about its geometrical description given by the command. To share data about shapes we chose to pass information with JSON serialization of objects. The JSON can be deserialized by Unity and restored to an `IShape` object, that is used to configure the `Collider` and the position of the region.

Due to the topology independence constraint, the number of shape types is not known at coding time. This made it impossible to hard-code the conversion from the `IShape` to the correct `Collider` setup, so we decided to leverage the **visitor pattern**. The setup of the `Collider` is represented by an `IShapeVisitor` that knows how to deal with each concrete shape type, which is injected as a dependency of the region creation code. The details on how this was done are shown below:

[4] A Prefab is a template in Unity that could be used to instantiate new Game Objects of the same kind and with the same components.

```
public IEnumerator InstantiateRegion(string regionId, IShape regionShape)
{
    IShapeVisitor<GameObject> visitor = gameObjectUpdater(
        InstantiateGameObject());
    GameObject gameObject = regionShape.Accept(visitor);
    gameObject.name = regionId;
    ...
    yield return new WaitForFixedUpdate();
}

private GameObject InstantiateGameObject()
{
    return Object.Instantiate(prefab, parent);
}
```

Since the visitor needs to be created with the reference to the `GameObject` to be modified, what needs to be injected is not the visitor itself, but a function that maps that `GameObject` to the visitor that modifies it. That function is called `gameObjectUpdater`, which is of type `Func<GameObject, IShapeVisitor<GameObject>>`.

An implementation of the visitor required for the region `GameObject` setup is shown below:

```
public class Simple3DGameObjectUpdaterVisitor
    : Simple3DTopology.IVisitor<GameObject>
{
    private GameObject gameObject;

    public Simple3DGameObjectUpdaterVisitor(GameObject gameObject)
    {
        this.gameObject = gameObject;
    }

    private T AddOrGetComponent<T>() where T : Component
    {
        T component = gameObject.GetComponent<T>();
        return component ?? gameObject.AddComponent<T>();
    }

    public GameObject Visit(Sphere sphere)
    {
        SphereCollider collider = AddOrGetComponent<SphereCollider>();
        collider.radius = (float)sphere.Radius;
        collider.isTrigger = true;
        gameObject.transform.position = sphere.Center.ToUnityVector();
        return gameObject;
    }

    public GameObject Visit(Cylinder cylinder)
    {
        CapsuleCollider collider = AddOrGetComponent<CapsuleCollider>();
        collider.height = (float)cylinder.Height;
        collider.radius = (float)cylinder.Radius;
        collider.isTrigger = true;
```

```
32          collider.transform.rotation = Quaternion.LookRotation(cylinder.
                Orientation.ToUnityVector(), Vector3.up);
33          gameObject.transform.position = cylinder.Center.ToUnityVector();
34          return gameObject;
35      }
36
37      ...
38  }
```

The implemented interface is not `IShapeVisitor` directly, but an extension that is specific for the chosen topology and that supplies a `Visit()` method for each supported concrete shape. Each `Visit()` method is implemented by acting on the given `GameObject`, adding the correct type of `Collider`, setting up its properties and finally moving the object to the proper position.

With this setup in place it's easy to have the Game Engine serve its purpose of dealing with space computation. It is important to notice that, in order to receive the correct information about the overlapping regions, we need to wait for a physics update cycle to be computed. This is especially true for non-suspensive primitives, that return a result immediately. To achieve the correct behaviour we used the built-in `WaitForFixedUpdate` that yields when a cycle is completed.

4.4 Runtime environments

The language of choice for implementing this project is C# because it is Unity's best supported language, and some components might have been better located inside Unity instead of the web server, so having the same language would have eased the component migration. As a matter of fact though, Unity runs a stripped-down version of **.NET framework**, which is not suited for developing web APIs, while the web server is compiled on **ASP.NET core**[2], which provides a strong framework for developing web applications, and can run on a wider range of platforms. While this could seem fine since each of the two components is executed on its own runtime, the problem is that even if they both use the C# language, code that is shared between the two components must be duplicated, because it has to be compiled twice: once for the .NET framework, and the other for the .NET core.

After these considerations, one could choose a different language for the implementation of the web server.

5 Self-assessment / Validation

In subsection 1.3 we introduced our criteria for self-assessment. In particular we wanted to build an **extensible**, **general**, and **technology-independent** middleware, ensure the basic functionalities with **Unit tests**, and check the **ease of development** of the case study.

While building, fixing and extending the middleware, no need to change its high-level structure has arisen, which gives us an informal - yet important - positive judgement on

the three qualities we wanted our software to have.

When it comes to **testing**, our focus was on trying to ensure the strength of the most crucial operations of the core on which the rest of the project builds, instead of complex integration tests that might have failed because of other reasons. The tests were produced at an initial stage of the project, and the suite remained the same despite the changes we made to the rest of the code-base. An important achievement is that the tests still run successfully at the end of the project, even though the software went through various substantial changes. The tested behavior is about suspensive primitives and regions interactions. The amount of testing is relatively low, with a narrow set of features being tested inside the middleware itself, not taking the distributed aspect into account.

The case study has been developed with ease, since the Spatial Tuples model is a really powerful coordination mechanism, the APIs exposed to the clients are pretty straightforward and the C# wrapper is particularly effective to quickly develop and use Spatial Tuples' abstractions. One note needs to be made though: since the suspensive semantic of the model has been handled by not answering to the APIs until a result is found, it is not possible to always use the same communication channel, as it might be busy waiting for a suspensive primitive to yield a result. This aspect has been taken care of in the C# wrapper, but a client using a different language and performing the HTTP requests on its own must be aware of that and handle the problem.

Given all these considerations we believe this project was really successful from a **software design** perspective, but lacked a more rigorous **development process**, which would have led the team to a more efficient route.

6 Deployment Instructions

The middleware is composed of two modules that can either be deployed on a single node or two nodes, depending on how efficient it is to provide more computational resources at the cost of increased communication delay. The two deploy units are the *Unity Server* and the *Core Server*: the former packages all its dependencies in the executable, the latter requires the host machine to install the .NET core SDK with version 3.1[5]. When running on different nodes, a minimal configuration is needed to setup the communication channel between the Unity Server and the Core Server:

- For Unity, after building the project, in the `[build-dir]/SpatialTuples_Data/StreamingAssets` folder there is a file named network_config.txt where it is possible to specify the IP and port of the Core Server.
- For the Core, one can change the appsettings.json file providing the IP and port on which the Unity Server accepts connections.

If the user wants to run the middleware on one node, it is possible to do so following these steps:

[5]Microsof documentation: https://docs.microsoft.com/en-us/dotnet/core/install/sdk

1. Install .NET core SDK;
2. Download the project zip for the Operating System in use;
3. Unzip the folder;
4. Execute the script `launch.sh`

Now an instance of Spatial Tuples is running on that machine, exposing its APIs on port 5000. The console will show a Log of all the primitives and agents registrations performed by the clients.

7 Usage Examples

The most comprehensive usage example of the middleware consists in the implementation of a small software simulation that somewhat resembles the **case study** proposed in the paper [4]. This enables us to showcase how someone should approach the usage of our Spatial Tuples prototype.

The case study simulates an Augmented Reality application, which for timing and convenience reasons has been developed in Unity, using a common scene that represents the "real world" and is replicated in every instance of the simulation. Each client is an agent that interacts with the **physical environment** using Unity's APIs, and **coordinates** with other agents by means of Spatial Tuples' ones.

7.1 Basic C# Agent Example

In this section we provide an implementation of a "Spatial Ping-Pong" scenario in which 2 agents move in space and, whenever they meet, they exchange a message by retrieving the companion tuple and putting a "Ping" or "Pong" tuple on themselves. Some code examples are shown to provide a basic overview of the capabilities and modelling abstractions of the middleware's C# wrapper. It's important to say that when dealing with the implementation of an Agent inside the Unity Game Engine some further details need to be taken into consideration, and for our case study we chose to decouple the different aspects of the agents behaviour following the GameObject's Component fashion.

The code presented here is the one of the `PingAgent`, that only differs from the other agent in the example for the initialization that features the emitting of the first "Ping" tuple.

To start, a new class implementing the `ISituatedComponent` is needed. In this case we give the agent its shape by constructor and we also pass the reference to the environment on which the agent will be living.

```
1 class PingAgent : ISituatedComponent
2 {
3     private readonly IEnvironment<string, RegexTemplate> environment;
4
5     public IShape Shape { get; set; }
6
```

```
7     public string Name => "Ping";
8
9     public event Action ShapeUpdated;
10    public event Action<ISituatedComponent> ComponentRegistered;
11
12    public PingAgent(IEnvironment<string,RegexTemplate> env, Sphere shape)
13    {
14        environment = env;
15        Shape = shape;
16    }
```

We assume that after being created the `Start()` method of the agent will be invoked from outside. In this method the agent first of all register itself onto the environment waiting for his presence to be acknowledged by the system. Then it can make some setup operations (in the case of the "Pinger" it is setting up the first tuple on himself) and then it starts his own behaviour.

```
17    public async Task Start()
18    {
19        await environment.AddComponent(this);
20        ComponentRegistered?.Invoke(this);
21        await environment
22            .Primitives(Name)
23            .Out("Ping")
24            .AtSituatedComponent(Name)
25            .Sticky(true)
26            .Run();
27        PingBehaviour().FireAndForget();
28        Move().FireAndForget();
29    }
```

In this example the behaviour is composed of two Tasks that run forever so they are launched with the `FireAndForget` extension method provided by us.

The first task is the one that makes the agent move in space: since the agent is a Situated Component, it notifies with an event that fires when its own shape changes over time. The environment captures this event and makes a request to the middleware to keep track of the position of each agent.

The second task defines the agent's behaviour. It simply performs a blocking In operation on itself so that when it encounters the other agent he can remove the matching tuple, and after that put a new "opposite" tuple on himself, waiting for the other to catch him again.

```
31    private async Task Move()
32    {
33        while (true)
34        {
35            Sphere mySphere = Shape as Sphere;
36            mySphere.Center = UpdatePosition(mySphere.Center);
37            ShapeUpdated?.Invoke();
38            await Task.Delay(1000);
39        }
```

```
40      }
41
42      private async Task PingBehaviour()
43      {
44          while (true)
45          {
46              await environment
47                  .Primitives(Name)
48                  .In("Pong")
49                  .AtSituatedComponent(Name)
50                  .Sticky(true)
51                  .Run();
52              await Task.Delay(1000);
53              await environment
54                  .Primitives(Name)
55                  .Out("Ping")
56                  .AtSituatedComponent(Name)
57                  .Sticky(true)
58                  .Run();
59          }
```

7.2 Case study description

The case study mimics a software tool to help *firefighters* locate fires with the help of a *helicopter* scanning the environment and a *control room* that monitors the whole action. The firefighters make use of the spatial coordination provided by the middleware to put out fires in the most efficient ways, avoiding wasting manpower on the same threat.

7.2.1 Configuration

When developing a new application for Spatial Tuples, there are three things that need to be specified:

- Tuple domain
- Tuple templates
- Geometrical Topology

These need to be implemented for a deeper customization, but for the sake of fast prototyping and simplicity the middleware provides strings as a tuple domain, regular expressions as tuple templates, and a "Simple 3D Topology" that only consists of points, spheres, and cylinders as the geometrical shapes supported.

7.2.2 Agents

Once the configuration is set, it is possible to start implementing the agents.

In our implementation of the rescue case the helicopter patrols the main road of the city, moving back and forth while looking for fires. When it finds one, it performs two

outs, placing tuples at the region corresponding to the center of the fire: one named *FIRE* and the other *HELP*.

The firefighter is controlled by the user, is an autonomously moving entity that exploits Spatial Tuples to locate the fires to extinguish, and to coordinate with other firefighters. It first looks for *HELP* tuples in a small radius around him using the in_p primitive, then if it doesn't find any tuple it waits a bit, increases the radius, and retries. After some iterations he will eventually find an *HELP* tuple and using the in primitive he is sure that no other firefighter is going after that fire, resulting in a more balanced division of work. When the *HELP* tuple is found, it carries spatial information that the firefighter can visualize in the form of a mini-map, that will guide him to the location of the fire. Once the firefighter is close enough to the fire he is assigned to, he can extinguish it by pressing the **F** key, which causes the "physical" fire to be removed, and performs an in operation at the fire location with the *HELP* template. When a firefighter extinguishes a fire, it starts looking for new *HELP* tuples around him, and the cycle starts anew.

The control room is really simple: it just shows the current location of the tuples and the agents in the city, by periodically issuing a rd_all at the city location, and performing the correct API calls to get the agents positions.

All of these are built using the C# wrapper, but the same considerations apply to the usage of Web APIs from any programming language.

8 Conclusions

In this project we first **studied** the Spatial Tuples model, then we **modelled** and **implemented** it as a **middleware** that exposes its functionalities by means of Web APIs. During the first stages of development some **unit tests** were made, and these helped throughout the process since a significant amount of refactoring had happened in that phase. Once the middleware was finished, we developed a case study to showcase the usage of the system and to cover some more complex scenarios that the tests couldn't catch.

The middleware is fully functional, but it lacks many important features that might simplify its usage, along with some performance tweaks that might actually make it suitable to be used in a scenario with many agents interacting over long periods of time. These aspects are covered in the following subsection, with a final section presenting the learning outcomes of the group.

8.1 Suitability of Game Engines

Since this project's main goal and purpose was to have a survey of the suitability of Game Engines for implementing the Spatial Tuples model, at the end of our project we tried to evaluate the experience that we had with that kind of technology in order to answer the question: *"How good are Game Engines to reach the full potential of the model?"*

Obviously, not having explored any other possibility our judgment is far from being the last word on the matter, but we try to be as impartial as possible.

Long story short, Game Engines are **good, but not perfect**. It was clear during the whole development process that we were trying to force a technology to serve our aim and, to be fair, it was relatively easy to reach a working state because of the great usability that Game Engines in general (and Unity in particular) have when prototyping applications.

But reaching a working system in a fast time doesn't always mean that the appropriate tool is being used. Other qualities are important, especially on projects of this size and purpose that need to be **robust** in order for others to rely on them for building their own application.

Testing, and other more advanced features of languages, such as asynchronous programming, are only partially supported on this technologies and when it came to deployment we struggled to have a portable system.

We still definitely believe that Game Engines are a solid option, especially because despite having to adapt to some domain specific concepts, they provide an abstraction layer that is crucial to avoid dealing with low level issues of modelling space.

Also we think that their graphical nature could be interesting for monitoring in a more intuitive way the behaviour of a complex application, even though we did not completely explore this possibility.

8.2 Future Works

As stated multiple times, this project was an exploration of the combination of Game Engine technologies with the Spatial Tuples model and the main goal was creating a **working prototype**.

It is clear that, from this starting point, there are many possible expansions and refinements to make this middleware useful for public distribution for research (and maybe even commercial) purpose.

One of the main aspects that has to be developed to achieve that goal is focusing on making the deployment and usage easier, maybe even developing a Unity plugin that, with a configuration GUI, could allow to use the system almost as a black box.

In the subsections below we discuss some other technical details of possible improvements that may be crucial to make the leap from being a prototype to being a robust framework for developing with this modelling abstraction.

8.2.1 Performance improvement

The system was developed without worrying about performance issues, but while implementing we took note of what could make the software faster and lighter on computational resources.

The actions that could be taken in consideration to have a more performant system are:

- **Garbage Collection** of unused anonymous regions
- **Reuse** of an existing region if a new one is identical

- **Distribution** of the spatial computation between multiple Game Engine instances on different nodes.

The first two measures are tied to the way we implemented the concept of an **anonymous region** that needs to be materialized in order to receive updates from the Game Engine about collisions with other regions. In a complex scenario with multiple agents and lot of requests, the number of regions could potentially rise really fast and make it harder for the Physics Engine to take in consideration all the different interactions between each entity. In that sense both implementing a "link" between two identical logic regions with just a real one, and collecting anonymous regions that no longer have tuples nor pending requests could be really useful when trying to reduce the load on the Game Engine thus improving performance.

The possibility of distributing the space assigning different "chunks" to different nodes running the game engine is another way to avoid heavy computation. This could be done with little effort creating a border manager to deal with in-between-chunks regions and components and expose the service as whole to the Environment server.

8.2.2 Additional functionalities

While developing the case study, we came across some additional features that could be useful when creating real life applications using the Spatial Tuples middleware.

First of all, we considered the possibility of having tuples and regions **persistently saved** through different sessions. As for now the system keeps all the data in RAM and when an instance of the environment is shutdown everything is lost.

The environment should be able to save regions with their state (described as their shape, position, contained tuples and pending requests) on a file or even better on a database and have the capability to reload the system communicating with the Game Engine to recreate the world as it was. Saving the whole environment on a persistent storage would also mean that the system would probably be less responsive, but would be able to handle big scenarios.

Another feature that could be useful in real life scenarios is **managing different environments** on the same middleware. This could be done modifying the current system to keep multiple instances and adding a level of **authentication** for agents that should register to a specific environment and receive a key to log and interact with that environment.

Having an authentication system is needed when deploying the middleware in the wild. In this way agents can operate on the environment only if approved by the environment itself.

The last feature that could be implemented is more of a utility for agents that could specify a region by a function applied to another known region. This is presented in the Spatial tuple paper[4] but we didn't implement it.

8.3 What we learned

Throughout the project the team frequently addressed topics concerning the **Linda model**, which gave great insight on both the user-experienced interaction with the model (primarily thanks to the case study) and its internal workings.

Studying and building Linda primitives proved an excellent way of learning it, and on top of that we also learned about the **Spatial Tuples** model. This is really interesting from a student's point of view because *spatial computation* always felt a bit abstract when studied in theory, but now we have a better understanding of it from a practical perspective as well.

We learned how to model a system and think using the Linda primitives as the basic building blocks, treating tuples and templates as first class citizens. Using the abstractions of Linda seemed a bit complex at first, but when applied to the right problem, its high abstraction level provides some really powerful mechanisms that greatly simplify both the modeling and implementation phases.

Talking about **technologies**, one that is pervasively used in software nowadays is **Web APIs**, even though there is little coverage of the topic in our courses. Luckily with this project we had some hands-on experience on how to build some with a modern and complete framework: ASP.NET Core.

Another tool we had the chance to study is the **Unity Game Engine**, whose basics were already known by the team. However, since it had a rather technical role in the development, we ended up studying its inner working in detail, understanding the most fundamental phases of its life-cycle, and leveraging interesting options such as the possibility of running the engine with no GUI.

A key way of learning is making mistakes, detecting and analyzing them to identify how to proceed next time.

One error that was only noticed at the final stage of development is the way we handled asynchronous code. We ended up in this situation because we light-heartily used C#'s asynchronous APIs, and we considered the requirements in an iterative way without really embracing all the aspects of the various increments. Discovering that at the end of a project could have led to a lot of code changes, but luckily C# provided some interesting mechanisms to help us solve this problem. This way we learned how important it is to have concurrency in mind from the very beginning of any project that deals with some kind of parallelism.

Another mistake was not making enough tests and most importantly not making them in a distributed fashion. A system like this needs a wide set of tests that can catch bugs early on, without the need of performing some tedious and cumbersome distributed debugging.

Overall we think that this project has been really instructive though it has kept us busy for quite much time. Analyzing the cause of the long time span needed to develop the project we came to the conclusion that first of all we all became passionate about the project itself and wanted to pursue a "usable end product" in terms of quality and functionalities. Also we spent more time in cooperative sessions than developing by ourselves because of the deep connection in terms of architectural design of the various

components of the system that we developed through an iterative process.

Going back we would spend more time at the beginning defining the interface and the different modules that compose the system so that we could work in parallel. This was hard at the time because we were uncertain about how things would eventually develop.

We still think that the way we chose, although long, was useful for didactic purpose because everyone had the chance to deeply understand the behaviour of each component.

References

[1] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

[2] Microsoft. Asp.net documentation. `https://dotnet.microsoft.com/apps/aspnet`.

[3] Microsoft. Task-based asynchronous pattern (tap). `https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap`.

[4] Alessandro Ricci, Mirko Viroli, Andrea Omicini, Stefano Mariani, Angelo Croatti, and Danilo Pianini. Spatial tuples: Augmenting reality with tuples. *Expert Systems*, 35(5), October 2018. Special Issue: New trends and innovations in intelligent distributed computing.

[5] Unity Technologies. Unity user manual. `https://docs.unity3d.com/Manual/index.html`.