C# client-side interface for TuSoW

Giacomo Cavalieri
giacomo.cavalieri2@studio.unibo.it

Nicolò Di Domenico
nicolo.didomenico@studio.unibo.it

March 30, 2021

1 Abstract

This project aims to create a C# client-side library that makes it possible to interface with the TuSoW server, leveraging the asynchronous programming paradigm that comes with the chosen language.

2 Goals/Requirements

The project has two main goals:

1. since TuSoW has no documentation available, the first step is to inspect the server's REST endpoints, their semantics and their required/optional arguments; to better organize this step, a Swagger Hub specification will be created
2. properly design and implement the class library, leveraging C#'s native asynchronous programming features in order to comply with the LINDA semantics without stalling the thread that makes the request

Therefore, the library will have to perform the same set of operations that are available on the server, which are:

- `count()`: gets the number of tuples in the tuple space
- `get()`: gets all tuples in the tuple space without removing them
- `out(T)`: inserts the tuple `T` in the tuple space
- `in(TT)`: non-deterministically gets a tuple that matches the given template (`TT`); if a matching tuple is found it is removed from the tuple space and returned, otherwise the operation blocks execution until a matching tuple is added to the space

- `inp(TT)`: behaves like `in(TT)`, but if no matching tuple is found the operation fails without blocking execution
- `in_all(TT)`: gets all tuples that match the given template (`TT`) and removes them from the tuple space; if no matching tuples are found an empty collection is returned
- `rd(TT)`: behaves like `in(TT)` but does not remove any matching tuples from the space
- `rdp(TT)`: behaves like `inp(TT)` but does not remove any matching tuples from the space
- `rd_all(TT)`: behaves like `in_all(TT)` but does not remove any matching tuples from the space
- `no(TT)`: if no tuple matches the template (`TT`) then the operation succeeds, otherwise the execution is blocked until there are not any more tuples that match the template
- `nop(TT)`: behaves like `no(TT)`, but if a matching tuple is found it is returned and the operation fails without blocking execution

Also, TuSoW implements by default two kinds of tuple spaces: textual and logic. The former uses strings as tuples and their templates are regular expressions. The latter uses Prolog terms as both tuples and templates.

Since it is possible to implement any kind of server-side tuple space, the library must support it: it must be relatively straightforward for the library user to implement a client-side interface for a completely new kind of tuple space, without actually doing the heavy lifting and having to manage all interactions with the server manually. For the sake of convenience, an implementation for textual tuple spaces will be provided by default.

2.1 Scenarios

This library is meant to simplify the usage of a TuSoW remote server via C#. Any user who may need to coordinate agents written in C# can take advantage of this library which provides an intuitive way to interface with a TuSoW server without having to deal with raw HTTP requests. It also provides classes to represent tuples and templates as objects without having to worry about their serialization/deserialization in order to communicate with the server. Figure 1 shows the UML use case diagram for the library.

Moreover, this library provides many interfaces and abstract classes to make it easy to implement different kinds of tuple spaces — like logical ones — supported by the TuSoW server.

2.2 Self-assessment policy

Since the project is actually just a class library (which means it is not directly executable), the only way to test it is through unit and integration tests. As the real tuple space logic

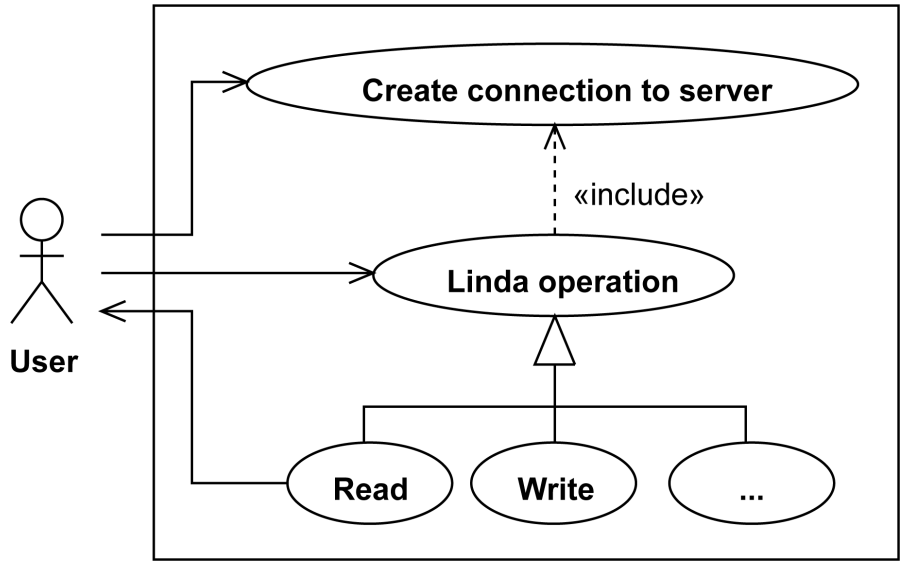

Figure 1: Use case diagram

is delegated to the server, testing aims to ensure that requests are properly formatted, responses are correctly parsed and LINDA semantics are followed.

3 Requirements Analysis

The library will have to provide the user with a simple way to interact with any remote tuple space on TuSoW, while also complying with the LINDA suspensive semantics, if applicable.

To do so, it will have to be able to correctly serialize objects into semantically correct JSON, and send the result to the server via specific HTTP endpoints. As many requests have associated responses with meaningful body, the library will have to support the inverse process (i.e. parsing the received JSON and deserializing it to create an object).

Since in TuSoW it is relatively straightforward to implement a new kind of tuple space, it should also be easy for the user to extend the library to allow interaction with any tuple space that may be supported by the server.

Lastly, the library will have to be designed with C#'s built-in asynchronous programming paradigm in mind, so that the user can choose not to stall the program's thread when making a LINDA operation with suspensive semantics.

To make sure that the library is compatible with all major desktop operating systems (Windows, macOS, Linux), .NET 5.0 will be used.

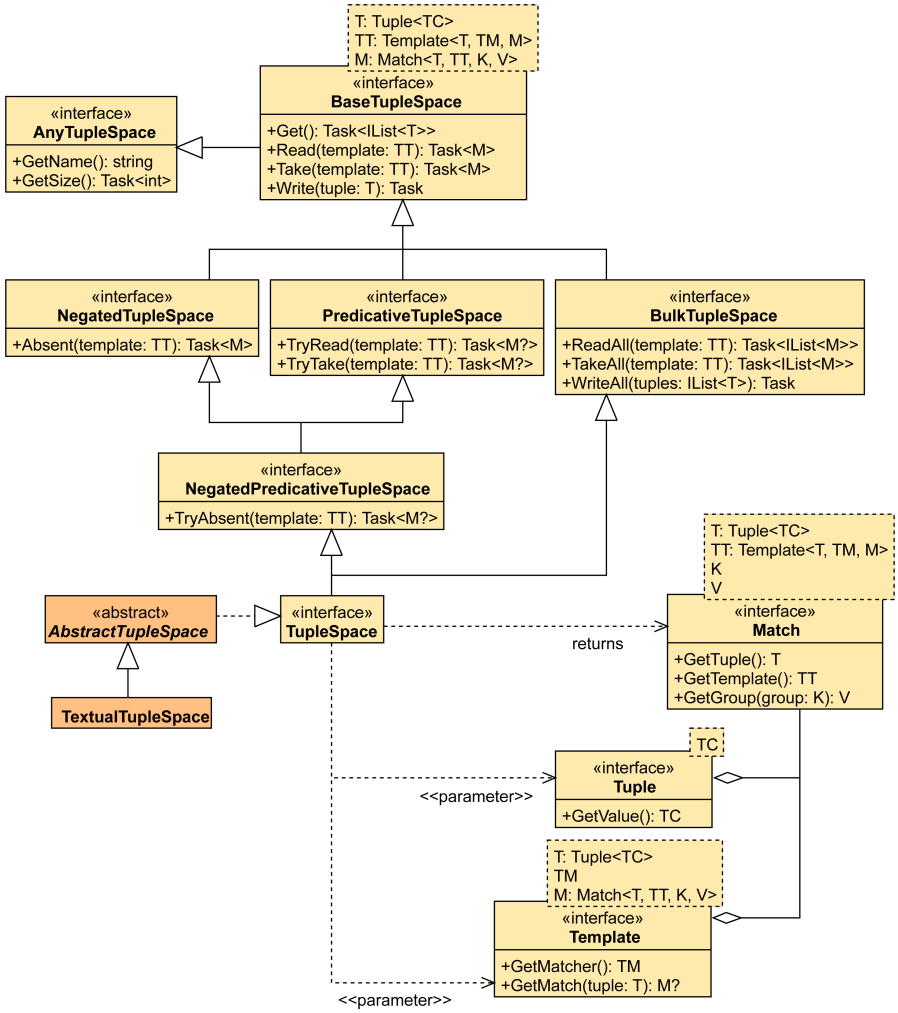

Figure 2: The main modelled entities: tuples, templates, matches and tuple spaces.

4 Design

4.1 Structure

One of the library's main goals is to provide the user with straightforward interfaces to interact with tuple spaces and their content. Therefore, the design process focused mainly on how to model these entities. As shown in Figure 2, tuple spaces are modelled in a hierarchy of interfaces, starting from the simplest kind of tuple space — i.e. defining a basic set of instructions: `get`, `read`, `take` and `write` — and building up complexity by adding different kinds of operations: bulk, predicative or negated.

The `TupleSpace` interface is used to describe a full-fledged tuple space supporting all TuSoW operations. An abstract class `AbstractTupleSpace` will take care of the aspects common to all types of tuple spaces while being completely agnostic in regard to the type of handled tuples. Figure 3 shows how the binding from generic to concrete types only takes place while defining subclasses of this abstract class.

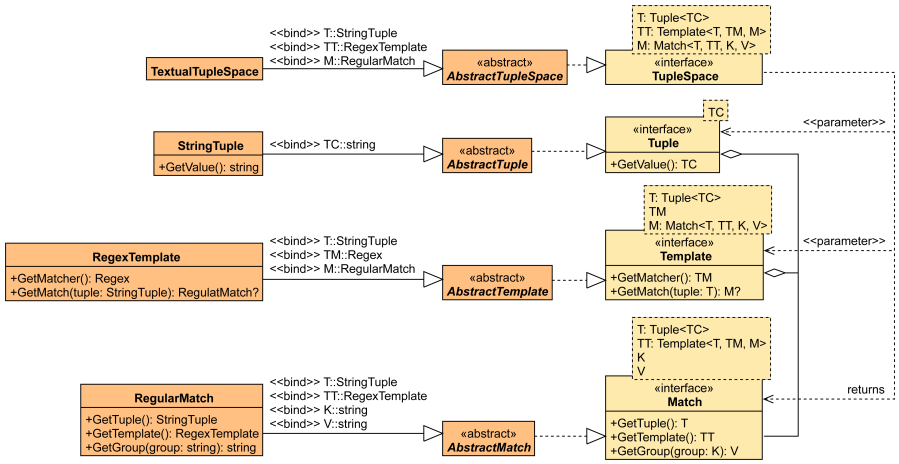

Figure 3: The implemented textual tuple space.

As shown in Figure 4, the connection to a server is modelled through the class `TuSoWConnection`, which provides a way to create tuple spaces connected to the same TuSoW server. In order to allow the user to implement and use his own tuple spaces, the class has a static method `Register()` that permits to define a new kind of supported tuple space by providing a function to generate it as needed by the `TuSoWConnection.UseTupleSpace()` method.

To prevent the user from instantiating subclasses of the `AbstractTupleSpace` — including custom ones — its constructor requires an object of type `TupleSpaceInfo` whose constructor is accessible only inside the assembly of the `TuSoWConnection`. Using this approach, only the `TuSoWConnection` can create new tuple spaces by providing them the necessary information as a `TupleSpaceInfo` object.

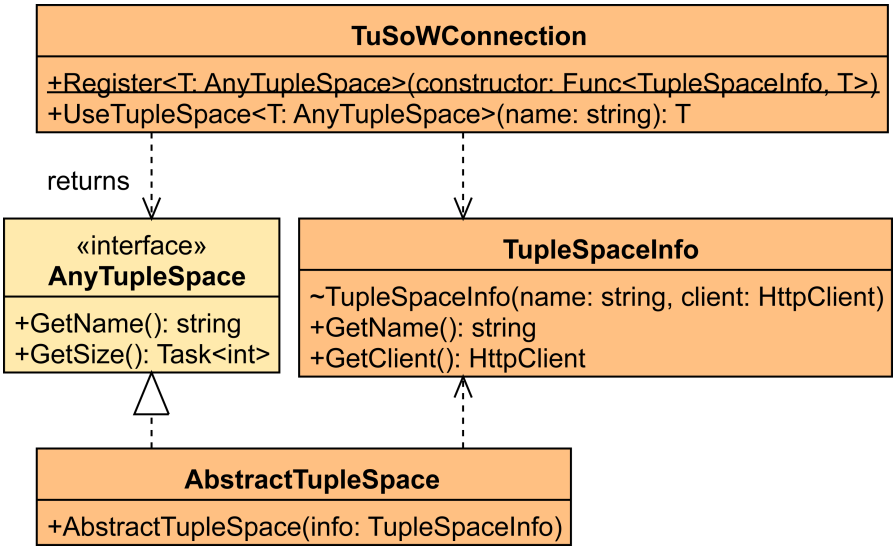

Figure 4: The `TuSoWConnection` class.

4.2 Interaction

The goal of the library is to make the interaction between the user and the TuSoW server as easy and straightforward as possible. As shown in Figure 5, the actions taken by the user in order to interact with a tuple space in a TuSoW server can be summarized to four simple steps: create a connection, create a tuple space, perform the required action and asynchronously wait for a response.

Figure 6 shows how the library takes care of the marshalling and unmarshalling processes by interposing itself between the user and the server. While the library takes care of composing correct messages and deserializing the server's answers, the user can simply use the `Tuple` and `TupleSpace` objects provided without having to worry about the technicalities of HTTP requests and responses.

5 Implementation Details

To reverse engineer the available TuSoW endpoints, we analyzed the `tusow-service` module, which contains the `head`, `get`, `post`, and `delete` functions that handle the incoming requests according to their HTTP method. The JSON schema of the various serialized objects (i.e. tuples, templates and match results) was inferred by observing the source code of each serializer: `StringTupleSerializer`, `RegexTemplateSerializer`, and `RegularMatchSerializer`.

With all these pieces of information gathered, the reverse-engineered TuSoW REST API has been formalized into a Swagger Hub specification [1], using the OpenAPI 3.0 standard. However, as TuSoW does not fully adhere to the HTTP/1.1 standard as specified in the RFC 7231 [2] and OpenAPI 3.0 explicitly forbids violating the aforementioned

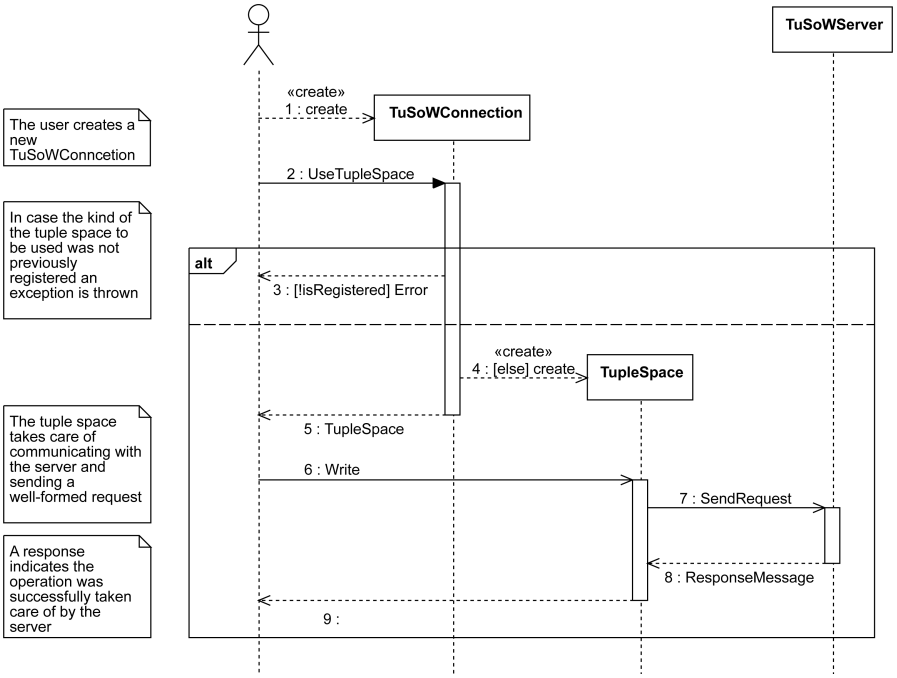

Figure 5: Interactions between a user and a tuple space when writing a tuple.

specification, some aspects of the server communication could not be correctly modelled: in particular, both `GET` and `HEAD` requests are required not to have a body, but TuSoW expects one that contains a serialized tuple template. Fortunately, many HTTP clients (including C#'s) allow a body regardless of the HTTP method; this means that while the Swagger Hub specification may not be perfect, the proper implementation does not rely on workarounds to properly communicate with TuSoW.

To implement LINDA's suspensive semantics on the client, we relied on C#'s native `async/await` features, in order to have clear and understandable asynchronous code, without dealing with any callback functions. Section 5.1 provides an overview of the usage of the `async/await` primitives.

Since `HttpClient` is intended to be instantiated once and reused throughout the lifetime of the application [3], the `TuSoWConnection` class is the one responsible for its creation. All tuple spaces created by this connection will share the same `HttpClient`. Also, this class implements a factory method that, given the remote tuple space name, instantiates an object that represents it.

Moreover, to make sure that no tuple space can be directly instantiated (forcing the user to call `TuSoWConnection.GetTupleSpace()`), when creating a tuple space object both HTTP client and the tuple space's name are passed using a `TupleSpaceInfo` opaque object, whose only constructor is marked as `internal`, thus making it impossible to create it from outside the library.

In order to make the `TuSoWConnection` aware of which types of tuple space are available

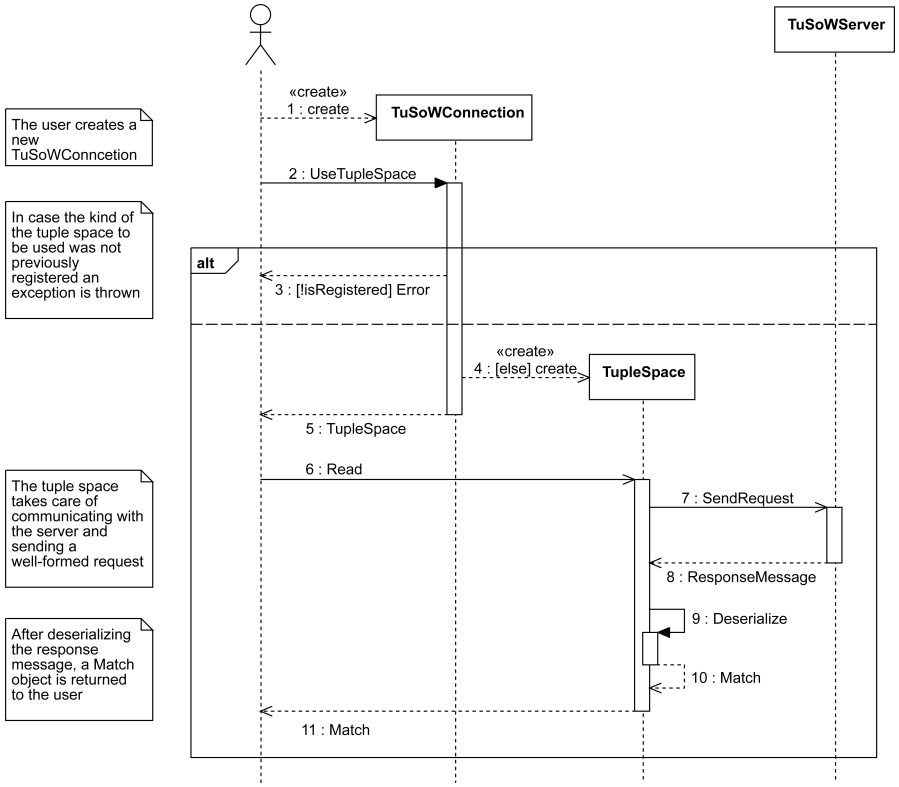

Figure 6: Interactions between a user and a tuple space when reading a tuple.

for creation, they have to be explicitly registered first by calling the `TuSoWConnection.Register()` static method. Its only argument is a function that, given a `TupleSpaceInfo` object, creates a new instance of the tuple space of the correct type. This step is not necessary for `TextualTupleSpace`, as it is already registered by default inside the connection's static constructor.

The registration phase for a specific tuple space type is necessary because in C# 9.0 there is no way to mark properties as required when using an object initializer. However, at the time of writing, there is a proposal to add such a feature to a future version of C# [4].

Another noteworthy design choice is the use of attributes to define how the tuple space's kind is named on the server (`RemoteTupleSpaceNameAttribute`). Since this property does not vary across the various tuple space's instances, it's more suited to be static. However, as C# does not allow interfaces for static members, an attribute was chosen as the cleanest solution without polluting the tuple space's interface with constant instance properties. A consequence of this choice is that it is impossible to statically check if the aforementioned attribute is present on a concrete implementation of a tuple space's interface; instead, the check is made at runtime when the tuple space type is registered via the `TuSoWConnection.Register()` static method.

Regarding serialization, we adopted the open-source `Newtonsoft.Json` library [5]. We used the `Serializable` attribute to mark tuples, templates and matches as serializable. Unfortunately, as attributes are not inherited in subclasses, each derived class must be manually marked as serializable. Also, we implemented two custom converters: the first one is responsible for converting regular expressions into strings that represent their pattern, bypassing C#'s default serialization behaviour for that class.

The other custom converter we implemented modifies the deserialization behaviour for the tuple matches: we chose to return a tuple match only if there is actually one, otherwise we return `null`. To do so, we implemented an abstract converter that is responsible for reading the `match` field in the given JSON and returns the deserialized object (falling back to the default deserialization behaviour) if that field is `true`, otherwise `null`. Since C# forbids the use of type parameters inside attributes, each class deriving from `AbstractTupleMatch` must also have a concrete attribute derived from `AbstractMatchConverter`, with all type parameters properly set.

5.1 C# asynchronous programming model

C# 5 introduces a task asynchronous programming model [6] that allows the programmer to write easier-to-read asynchronous programs, letting the compiler handle the underlying complexity. The resulting programs have a structure similar to a synchronous one and are therefore easier to understand and change over time.

An asynchronous method follows these simple rules:

- It must have the keyword `async` in its signature
- Its return type should be `Task<T>` or `Task` in case the computation does not yield any result

- It should return an object of type `T` inside its body

Listing 1 shows a simple example that encapsulates the main points of interest. Inside an asynchronous method, the keyword `await` can be used on a `Task` to wait for its completion and get the unpacked return value. If the task is not over when the `await` is performed, the control is returned to the method caller yielding a `Task` of the type specified in the signature. For example, in Listing 1, if the independent work completes before the `getStringTask` can return an actual value, when the `await` is performed the control will be returned to the caller of `GetUrlContentLengthAsync`, which will receive a `Task<int>` that can be, in turn, awaited.

```
public async Task<int> GetUrlContentLengthAsync(string url)
{
    var c = new HttpClient();
    Task<string> getStringTask = c.GetStringAsync(url);
    // Do some independent work...
    string contents = await getStringTask;
    return contents.Length;
}
```

Listing 1: An example of an async method in C# [7]

6 Self-assessment / Validation

To assess the correct working of the library, tests had to be aimed at ensuring a correct communication between the user and the TuSoW server. The main goal was to create a general enough class that could be simply used to immediately test new kinds of tuple spaces without having to change the test code. The `AbstractTupleSpaceTests` class defines the base tests common to all tuple spaces; in order to test specific tuple spaces — i.e. a textual or a logic one — the user only needs to create a subclass which provides an example tuple, a template that matches it and a template that does not. With these simple data many tests can be performed checking all available primitives for a tuple space, independently from its specific type.

Notably, we tested the proper working of the different operations — e.g. a `Take` operation actually results in the removal of the read tuple — and made sure that the library complied with their suspensive semantics — e.g. a bulk operation should return an empty list in case no matches are found instead of blocking — with a total of 12 test methods.

In particular, to test the suspensive semantics of operations we leveraged the Nunit testing framework by defining a new testing constraint [8]. This allowed us to write expressive and easy to read tests; an example is shown in Listing 2.

```
// Test operation is blocking
Assert.That(<task>, Blocks.For(<timeout>));
// Test operation is non blocking
```

```
Assert.That(<task>, Blocks.NoLongerThan(<timeout>));
```

Listing 2: An example of how to test for blocking/non-blocking methods

Another important aspect to test was the correct implementation of the serialization/deserialization mechanisms in the textual tuple space. Therefore, we defined the `TextualSerializationTests` class which makes sure that tuples and matches are handled correctly so that any message that could be sent or received from the server conforms to its expected schema.

In order to correctly run integration tests with a real TuSoW server, we created a Docker image of it and used the `Docker.DotNet` [9] library to automatically launch a container with the aforementioned image before starting the integration tests and destroy it once testing is over.

This step is not needed when testing the library via continuous integration, thanks to GitLab CI's services feature, which exposes an arbitrary container to be accessible from the code that's being tested.

7 Deployment Instructions

Importing the library is very straightforward, thanks to the continuous integration pipeline that was set up for the project: every time the `master` branch is updated, a new pipeline is launched and, if all tests pass, the library is published as a NuGet package inside the repository's registry. This means that the user only has to add the package registry as a new NuGet source to their solution, and install the library as usual. As stated on GitLab's documentation [10], a deploy token with `read_package_registry` or a personal access token with the scope set to `api` is required.

To register the NuGet source, the user has to enter the command in Listing 3.

```
nuget source Add -Name GitLab -Source "https://gitlab.com/
    api/v4/projects/23956181/packages/nuget/index.json" -
    UserName <your_username> -Password <your_token>
```

Listing 3: The command to register a new NuGet source

Then, the library can be added by using the command in Listing 4.

```
nuget install TuSoW-Cs-Client -Source "GitLab"
```

Listing 4: The command to add the package to the project

Once this is done, the user can create a new `TuSoWConnection` and connect to a TuSoW server.

To build from source, .NET 5.0 and Docker 20.04 (or upper) are required. Inside a terminal with working directory pointing at the project root, the user only has to enter `dotnet build` to build the project and `dotnet test` to test it.

8 Usage Examples

Using the library is very simple: Listing 5 shows how to create a new connection to a TuSoW server.

```
using System;
using System.Threading.Tasks;
using TuSoWClient.Linda.Remote;
using TuSoWClient.Linda.Remote.Textual;

var conn = new TuSoWConnection(
   new Uri("http://localhost:8080"));
TextualTupleSpace ts = conn.UseTupleSpace<TextualTupleSpace
   >("testTupleSpace");
```

Listing 5: How to connect to a TuSoW server

After getting a reference to a tuple space, operations can easily be done asynchronously, as shown in Listing 6.

```
// Write a tuple
await ts.Write(new StringTuple("name: John Doe; age: 23;"));
// Then make an async request to get that tuple
Task<RegularMatch> matchTask = ts.Read(
   new RegexTemplate("name: (.*?); age: (?<age>[0-9]+);"));
// ...do other slow stuff...
RegularMatch m = await matchTask;
// ...now you can use the resulting match!
Console.WriteLine($"Hello {m.Groups["1"]}!");
Console.WriteLine($"You're {m.Groups["age"]} years old!");
```

Listing 6: How to write and read a tuple from a tuple space

9 Conclusions

In conclusion, we first inspected the valid REST endpoints of the TuSoW server by analyzing its code and created a Swagger Hub specification. Then, we implemented the library's core by modelling tuple spaces, tuples, templates and matches. We tried to make it as easy as possible to implement additional tuple spaces; writing the additional code for a textual tuple space proved to be fairly straightforward. At the same time, we developed a set of tests that could be applied to any kind of tuple space and used it to test our implementation of a textual tuple space.

9.1 Future Work

Since TuSoW can currently handle both logic and textual tuple spaces, a possible future expansion of the library could consist in the addition of a default implementation of a

logic tuple space. It was not possible to add it to the project as it would have required the development of a .NET porting of tuProlog and it would have exceeded our hour limit.

9.2 What we learned

Working on this project has been interesting as well as highly educational. Because we had to reverse-engineer the TuSoW server to write a Swagger Hub specification, this allowed us to gain a better knowledge of its inner workings; we were also able to spot a bug in its implementation and propose a fix [11].

Given that the project was heavily focused on the modelling of tuple spaces, we deepened our understanding of the LINDA model, clearing up any doubts that may have arisen on the semantics of its operations.

References

[1] Giacomo Cavalieri and Nicolò Di Domenico. *TuSoW REST Specification*. URL: https://app.swaggerhub.com/apis/ndido_98/tusow-specs/1.0#/.

[2] Roy Fielding and Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. Internet Engineering Task Force (IETF), June 2014. URL: https://www.rfc-editor.org/rfc/rfc7231.txt.

[3] *HttpClient class*. Microsoft Corporation. URL: https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient?view=net-5.0.

[4] Fred Silberberg. *Nominal Parameter Lists (aka Required Properties)*. Microsoft Corporation. URL: https://github.com/dotnet/csharplang/discussions/4209.

[5] *Json.NET - Newtsonsoft*. Newtonsoft. URL: https://www.newtonsoft.com/json.

[6] *C# version 5.0 major features*. Microsoft Corporation. URL: https://docs.microsoft.com/it-it/dotnet/csharp/whats-new/csharp-version-history#c-version-50.

[7] *Task asynchronous programming model*. Microsoft Corporation. URL: https://docs.microsoft.com/it-it/dotnet/csharp/programming-guide/concepts/async/task-asynchronous-programming-model.

[8] *Nunit Constraint Model*. The NUnit Project. URL: https://docs.nunit.org/articles/nunit/writing-tests/assertions/assertion-models/constraint.html.

[9] *Docker.DotNet*. .NET Foundation. URL: https://github.com/dotnet/Docker.DotNet.

[10] *Add the Package Registry as a source for NuGet packages*. GitLab. URL: https://docs.gitlab.com/ee/user/packages/nuget_repository/index.html#add-the-package-registry-as-a-source-for-nuget-packages.

[11] Nicolò Di Domenico. *Fix TuSoW's grouping bug.* URL: https://gitlab.com/nicolo.didomenico/coordination/-/commit/30926d5998b7299531f1ef919e01b0f788e3021d.