UniRosPair

1st Daniel Chu
Department of Electrical Engineering (of Aff.)
National Chiao Tung University (of Aff.)
Hsinchu, Taiwan
daniel.chu916@gmail.com

2nd Jeremy Liu
Department of Electrical Engineering (of Aff.)
National Chiao Tung University (of Aff.)
Hsinchu, Taiwan
jremyliu09333@gmail.com

***Abstract*—This document is an introduction of using the UnityRosPair tool and how it works behind the scene.**

***Index Terms*—ros, unity, internet, connection, rosbridge, server, client, command, kit, tool, developer, oop, string**

I. Introduction

UniRosPair is a developer tool for developers to create a simple network of two unity client pairs and a python server. Unity clients can send messages to affect each other by string messages while the servers acts like the network center processing the messages.

II. Usage

A. Connecting Two Unity Programs

UniRosPair serves as a connector, facilitating direct message passage between two Unity programs.

B. Creating Two-Player Games

UniRosPair's python server can act as a server for two player multiplayer games. The server is not limited to serve only two player. It assigns new room for incoming new player.

C. String-Based Commands

UniRosPair utilizes string-type commands, offering simplicity in understanding and ease of functionality extension.

D. Scalability

This tool provides scalability by allowing the seamless addition of your own game into the server script. Moreover, the framework supplied by UniRosPair empowers users to develop their message processing algorithms.

III. Installation

A. Python Side

First, go to the link below and fork it into your repository. https://github.com/jhechengliu/siege-aoop-python. Next, clone the repository to the local side and you have installed the server side of this tool.

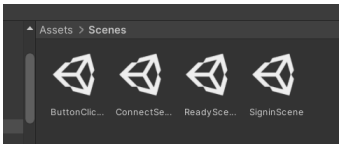

Fig. 1. Assets/Scenes Folder

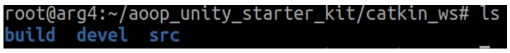

Fig. 2. Environment Setup Terminal Information

B. Unity Side

The repository of the unity client is located right here: https://github.com/dctime/unirospair-unity If you want to use this as a starter kit, fork this repository and clone it into your local, open using Unity Hub and your off to go. Make sure that the version of Unity is correct. If you are looking for using as a part of your project, download this repository and open in Unity Hub. After loading the project up, you can see four scenes in the Assets/Scenes folder (Fig. 6). ConnectServerScene, SigninScene, and ReadyScene is all what you need while ButtonClicker is just for demo purpose. For people who wants to add this tool into their system, drag the three scenes into your project and all the preparation is done.

C. Server Setup

The following steps instruct how to turn on server. The environment setup have to be done first. Use docker and build the workspace (ROS setup). Get into the repo, and run docker.

```
\$ cd ~/unirospair-python
\$ source docker_run.bash
```

In docker, type the following commands:

```
\$ cd catkin_ws/
\$ catkin_make
\$ ls
```

The terminal will be like Fig. 2.

Open three new terminals. First one is use to run ROSBridge server. Get into repo and run docker.

```
$ cd unirospair-python
\$ source docker_run.bash
```

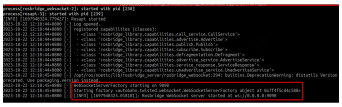

Fig. 3. Terminal 1 information

Next, type commands:

```
\$ source environment.sh
\$ roslaunch
rosbridge_server rosbridge_websocket.lau
```

If you see the three line in Fig. 3, means you are successfully turn on server.

Open second terminal to run python code with ros. Same as before, get into the repo file. Then, type the following commands:

```
\$ source docker_join.bash
(Docker)\$ source environment.sh
(Docker)\$ pip install shortuuid
(Docker)\$ chmod +x
catkin.ws/src/exmple/src/main.py
(Docker)\$ rosrun
catkin.ws/src/exmple/src/main.py
```

And your python code are running. In the last terminal, type command

```
\$ ifconfig
```

to check IP for server side for after Unity setup.

D. Unity Side Setup

Check your server's IP and update it to script RosConnector.cs under Assets/RosSharp/Scripts/RosBridgeClient/RosCommuncation/. Run unity in ConnectServerScene. Go back to python side and see if server has connected to unity successfully.

E. Run the Example

If you got "Connected to Rosbridge", now its time to go to the next step. If you got an error for this, check if the IP address is correct or not. Now shutdown the Unity runtime and hit File/Build And Run to create another client. Switch back to the unity editor and hit play, and now your ready to connect the editor one and the build one. Click the screen "...tap to continue..." to go to another screen. In the next scene, type in your name in the textbox and click Play As Attacker/Defender to get a role. After getting the role, go to another client and do the stuff again. When both clients got the role, the timer will start countdown in 5 seconds and the scene will switch to the real game. Now when you click the button at one side, the button will push down a little bit and the other side button will push up a little bit. Congratulations. Two clients were connected each other by rosbridges.

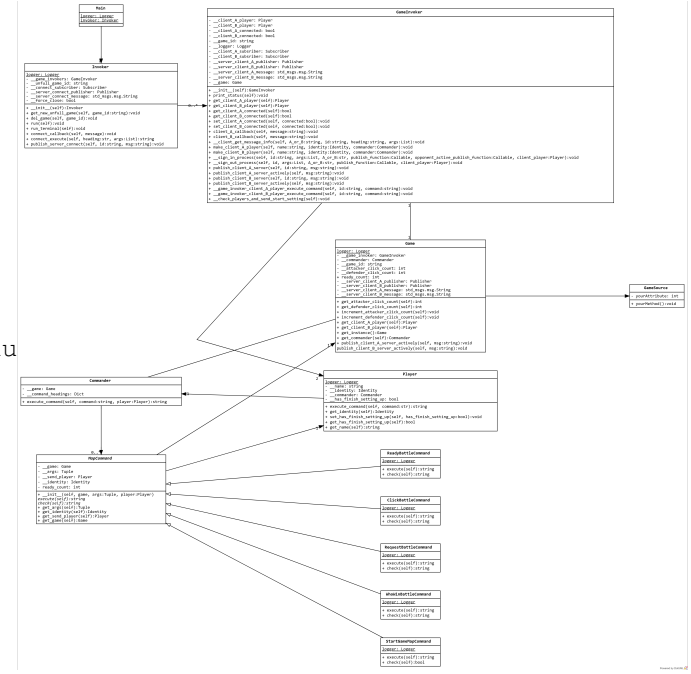

Fig. 4. Python server class diagram

IV. System Structure

A. Class Diagram of Python Server

- Fig. 4 is the Class Diagram of the python server
- When we run main.py to start the server, Invoker, GameInvoker, Game, Player, and MapCommand were initialized one by one. After initializations.
- Run method and run terminal starts to go into an endless loop in two threads. One for keeping the prgram open while the other stops all the time waiting for the user to enter server commands.
- When the program is running endlessly, subscribers and publishers begins to control the whole program. At the same time, if the subscribers in the Invoker and GameInvoker class hears something from the rosbridge passing message with the same topic as the subscriber is interested in, subscriber calls the callback function which the subscriber stores.
- The Invoker class rule is to hear the new coming clients and give the client a unique topic to keep in contact while send the topic to the GameInvoker class.
- Each GameInvoker class can only have two clients: client A and client B. When two client are registered, Invoker instance will generate another GameInvoker instance to store more incoming clients, which means that this server can generate endless pairs of clients until there are no memory left
- After connection, the client has the connection to the GameInvoker class.
- While the GameInvoker instance knows how to communicate with the client. The client still can't modify the game

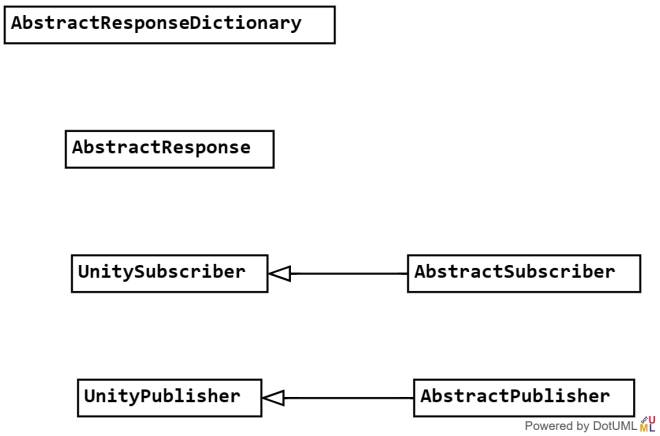

Fig. 5. Unity Client Class Diagram

yet because the GameInvoker doesn't have the ability to change data in the game.

- In order to access permission, the client must register the game. The client must provide an identity and an name in order to register. After registration, the client now got a Player class instance and the player has the commander which can build command and send directly to the game to change the games data.
- In development, we can easily modify the behavior of the client by adding new commands to the MapCommand abstract class and let the commander know the heading of the command is. We will talk about how to make custom commands later.
- Notice that Invoker, GameInvoker and Game class have publishers in it. Invoker's publisher is for sending back the topic for the newcomers while GameInvoker's publisher is for sending back success messages if the client successfully register a player or not. Lastly, Game's publisher is for commands to have a way to send game data back to the client.
- By the way, in every class, there is a logger in the class diagram to log error, info, warning, debug, and fatal logs which is great for debugging. When adding a new class to the server. use "logger = Logger("YourLoggerName")" to make a logger in your class.

B. Class Diagram of Unity Client

- Fig. 5 is the Class Diagram of Unity client.
- Utilization of Four Abstract Classes in Different Scenes. Incorporate game objects and attach the relevant scripts to them.
- Execute the build and run process in Unity. Alternatively, select the ConnectServerScene and press play to run it in the editor.
- Click on the screen to send a message to the server via the publisher. As both the server and clients share the same topic at this stage, communication can occur before pairing.
- The server will send a message back, received by the subscriber. Each client is assigned a short UUID by the server. The client records the topic name in UserRegister for subsequent topic initialization.
- Enter a name (without spaces and underscores) and click one of the buttons.
- The server detects if the opponent has been selected. If so, the button for the opponent's identity becomes unselectable.
- Upon both players signing in, the server sends a message to both subscribers. After a five-second interval, the transition to the ButtonClicker scene occurs.
- Implementation of the game takes place in this scene.

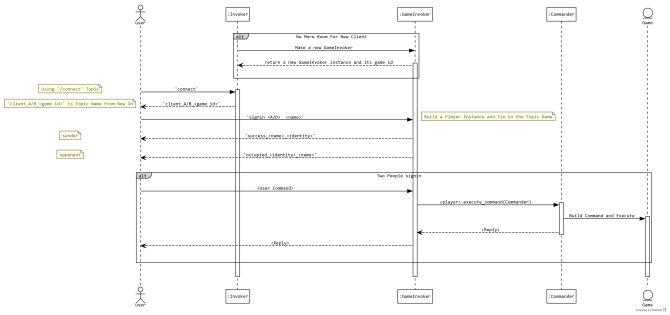

Fig. 6. Messages Sequence Diagram

V. Messages

During connection, every message in this tool is packed using the string format. The messages flow during the connection state is shown below:

A. The "connect" Command

When the client is executed, the rosbridge between them were build automatically. When the Unity client isn't communicate with the python server before, the client only holds the topic "/connect". In order to affect the game, the client should send a message "connect" with the topic "/connect" to the server. The class Invoker in the server has the responsibility to listen to the messages coming from "/connect". By the way, when there is not enough room for client to stay. Invoker must generate a new GameInvoker with a new Game ID. If Invoker hears "connect" from the "/connect" topic, it will send the topic which can be used to communicate with the already built GameInvoker instance back. And that means that the client has now have a GameInvoker instance to talk to.

B. The "signin" Command

Now, the client has the topic to communicate with the GameInvoker instance. In the Unity Example Runtime, you can see that the scene switch to a new scene with a textbox and two "Play As" buttons. When you type in your name and click one of the "Play As" buttons, the "signin" command will

be send to the GameInvoker with the topic name got from the previous command. The signin command ”signin [A/D] [name]” contains two arguments. [A/D] represents the identity of the client which the client wants to be. A is for Attacker while D is for Defender. The client will send A if the client pressed the Play As Attacker. On the other hand, if the client send D in the signin command, it means that the player click the ”Play As Defender” Button. The [name] represents the name which the player names itself. The client got the name from the textbox which the player enters their name.

When the GameInvoker instance received the signin message, it will start to build a Player Class and store into ”client A player” attribute or ”client B player” attribute depending on which client sent the message. If success, if will send ”success [name] [identity]” back to the client. Notice that [name] and [identity] is the opponent's name and identity. If the opponent client already signed in, [name] and [identity] can be used to display enemy data in the screen. If not, [name] and [identity] will return ”none” (string not None in Python).

At the same time the GameInvoker sends success message to the sender, the opponent will also get a message ”occupied [identity] [name]”. [identity] is the client who signed in for while [name] is the sender's name. If the client is in Signin Scene, one of the button will be unclickable. If the sender choose to become an attacker, the ”Play As Attacker” button won't be clickable. If the sender choose to become an defender, the ”Play As Defender” button won't be clickable.

C. Game Commands

Now the client is tied to the player in a GameInvoker instance, which means that the client is able to send messages to the GameInvoker instance and the player will help you get the message and send to the commander and the commander will execute the command you have send and modify the data in the Game Class. In the example game the repository provided have two commands ”click” and ”request”. Click command tells the server that the client pressed the button once while the Request command request the server to send back how many clicks does itself and opponent click. Other information about this topic will be written in the next section ”Customize the Game”.

VI. Customize the Game

In this section we will talk about how to add UniRosPair tool into your game.

A. Requirements

We suggest that before adding any scripts in your game. Make sure that you have read everything above. Its okay to follow the instructions but there might have a lot of problems if you don't have the knowledge you need.

Try to draw a Sequence Diagram with a server and a client. Make sure that you have drawn every message the client might send and replies the server send when the client receives the message.

B. Python Side

In this tool, Python is the server side. Hence, we have to write some scripts to let the server know that which message the server might receive. Inherited MapCommand child classes to implement your game logic. Return type of the MapCommand child classes are string. If received correct heading and arguments, the check() function return None and the execute() return what's going to send back to client. Else, check() return error message (you may see them in the terminal where you run the server). Remember to add your new headings and MapCommand classes in Commander class dictionary. It'll call the function according to the message received.

Take ReadyBattleCommand in battle.py for example. When client sent ”ready” and server received it, Commander find this heading and command class in its dictionary. If do, create and call command class with heading and args as parameters. Run check() first, then execute(). The returned message is sent back to Commander, GameInvoker, and published by the publisher.

C. Unity Side

For the Unity side, what you need to do is to write the UI and send messages through rosbridge and write the scripts for every possible reply from the server. For a new scene, you need a make a Response Dictionary and a Subscriber. Make two new C scripts and inherit the Abstract Subscriber and Abstract Response Dictionary. Make sure that these two scripts must be active in order to work. After that, its time to write some new publishers and response.

For the ButtonClicker game, check the ClickCommand object. This object is responsible for publish click command and do something when someone reply the click command. There are two scripts attached to the object: the click response and the click publisher. In the click response script, we can see that the class inherits the AbstractResponse Class, overrides the ResponseToMessage and write some scripts that do something by checking the responseMessage. In the ClickPublisher script, the class inherits the AbstractPublisher and has a method which calles the PublishMessage in the AbstractPublisher class. This method is then called by the outside when its time to publish the message.

In order to make a new command, what you need to do is to open two new scripts. One inherits the AbstractPublisher and the other inherits the AbstractResponse. Then, make a method function in the class that inherits the AbstractPublisher and write some code to process the message. After that call the PublishMessage method. Whenever the program needs to publish the new command, what you need to do is to call the function defined in this class. For the class that inherits the AbstractResponse, override the ResponseToMessage method and write code that process the reply. That's it! Remember to put them into the scene and keep them active.

VII. Conclusion and Future Work

In this work, we leveraged ROSBridge to facilitate the seamless transmission of information between Python and

Unity. While ROS proved to be a robust solution, it's worth noting that alternative approaches, such as using TCP sockets for communication, also exist. Notably, the use of [3] enables bidirectional data exchange between Unity and Python 3 through TCP. Building upon the foundation of ROSBridge, we developed a system designed for matching and pairing multiple players. Beyond the automatic allocation of two players into the same room, our system has the capability to record and process essential logic code on the server side. However, the current implementation reveals certain limitations in terms of robustness. Moving forward, our focus in future work will be on enhancing the system's robustness. Key considerations include implementing measures to ensure the stability of communication between the server and clients, guaranteeing the reliable transmission and receipt of every message. Addressing these challenges will contribute to a more resilient and dependable system.

REFERENCES

[1] Wellyowo. oop-proj-unity-ros. Available online: https://github.com/wellyowo/oop-proj-unity-ros

[2] Wellyowo. oop-proj-unity. Avaliable online: https://github.com/wellyowo/oop-proj-unity

[3] Siliconifier. Python-Unity-Socket-Communication. Avaliable online: https://github.com/Siliconifier/Python-Unity-Socket-Communication