RLereWolf – Reinforcement Learning Agent Development Framework For The Social Deduction Game Werewolf

Georgi Ventsislavov Velikov

A dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science of the University of Aberdeen.

Department of Computing Science

2021
Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed: Georgi Velikov

Date: 2021
Abstract

Unlike complete information games, which are games whose game state is visible to all players, at any given point of time, incomplete information games rely on the player’s ability to infer the “missing” information from the provided to them game state. The information inference is targeted towards other players and their personal player state, or role. Due to the inherent nature of complete information games, which do not require any assumptions to be made on the players or the game state, computers are able to beat the best players at those games, such as – Chess, Go, and Checkers by Deep Blue, AlphaGo Zero, and Chinook respectively. In order to beat the best players in incomplete information games, the software built to play them has to be able to adapt to the dynamically changing game state and “fill-out” any of the game state gaps with assumptions based on previous experience and contextual game knowledge, effectively learning what the best action to some previously encountered position. RLeerWolf aims to provide developers with the framework required to build, observe, and improve existing Agents, which are non-human players designed to play the game, as well as introduce trust, honesty, and Q-learning into the domain of beating Werewolf.
Acknowledgements

I would like to thank all who have provided me with their time, trust, and assistance during the course of the project and my degree.

Firstly, I would like to thank my project supervisor – Dr. Bruno Yun, who has not only provided me with extraordinary support, but has also made sure I followed the project timeline.

I would also like to thank my friends and family for supporting me emotionally throughout the project. Having my family, Miroslav, Militsa, Branimir, Hristiyana, and Margarita to keep up with has provided me with an incomparable “switch-off” from the on-going world events and my project work.

I would also like to extend my gratitude to everyone I have met and been close with during my degree. The friendships and knowledge exchange along the way, has made me what I am today. Thank you Anton, Emilia, Keeyan, Marcin, Mikko, and everyone else in the 2016 & 2017 cohorts.

Discussing project difficulties, possible solutions and overall project progress with the think tank consisting of Marcin, Waseem, and Siddig has been an invaluable source of design solutions for the project.

Moreover, I would also like to extend my gratitude to The Centrist Chads for providing me with countless hours of entertainment and emotional support throughout my degree and the project, whilst helping me keep in shape.

I would like to also extend my deepest gratitude to Aberdeen’s Codify. The invaluable skills and knowledge I have gained as part of my employment at Codify, both during my internship and subsequent full-time employment have been a core contributor to my technical abilities which I have fully employed in order to work on the project.

Finally, I would like to thank Simona for always being there throughout the entirety of my degree and getting me through all of it, reminding me what matters most.
List of Tables

3.1 Hardware used during the development of RLereWolf ........................................... 25
4.1 Base Werewolf game roles – possible time to act ....................................................... 37
4.2 Role-ratio values used by RLereWolf ......................................................................... 38
4.3 Base Werewolf role-point values .............................................................................. 39
4.4 Base Werewolf game roles – possible actions ............................................................. 39
4.5 RLereWolf Environment rewards .............................................................................. 47
6.1 RLereWolf role distributions for experimented Player counts ...................................... 56
6.2 Experiment win-rates, average Game length, and average invalid actions for a “run” of 1000 Games .................................................................................................................. 57
6.3 Points for the current ratio-based role-distribution counts ........................................... 57
B.1 Client Project code listings ......................................................................................... 73
B.2 Server Project code listings ........................................................................................ 73
B.3 Shared Project code listings ....................................................................................... 75
B.4 Werewolf Project code listings .................................................................................. 76
List of Figures

3.1 Pygubu Designer – the RAD graphical user interface builder tool ........................................ 22
3.2 Ray RLlib Architecture ........................................................................................................ 23
3.3 GitHub metrics – commit count and line modification (adds in green & deletes in red) counts ................................................................. 24
4.1 Client-Server-Game architecture ......................................................................................... 27
4.2 Project Layout – as seen in Visual Studio 2019’s Solution Explorer .................................. 28
4.3 Server architecture – receiving packets ............................................................................... 28
4.4 Game State Packet (GameDto) structure ............................................................................ 29
4.5 Server log – recording the connections it replies to, alongside the request packet types the server receives .......................................................... 31
4.6 Logs file structure ................................................................................................................. 31
4.7 Game log stored on the server, recording the full game state ............................................ 32
4.8 Client architecture ............................................................................................................... 33
4.9 Game Screen – Annotated elements ................................................................................... 34
4.10 Packet Updated Entity DTO ............................................................................................. 36
4.11 Communication – Conflict resolution .............................................................................. 41
4.12 Players architecture – The distinction between humans and Agents .................................. 43
4.13 Agent-environment loop ................................................................................................... 45
6.1 Experiment metrics graph – Win-rate and Game speed ....................................................... 56
6.2 Experiment metrics graph – Sum of Points for existing role distribution (inverted) ........... 58
6.3 Experiment metrics graph – Average Game time (seconds) ............................................... 59
6.4 Experiment metrics graph – Average invalid actions per Game ......................................... 60
A.1 Main menu – No Client name provided .............................................................................. 64
A.2 Main menu – Set Name dialog ............................................................................................ 64
A.3 Main menu – Client has a name, ready to connect ............................................................. 65
A.4 Main menu – Help dialog .................................................................................................... 65
A.5 Game List screen ................................................................................................................ 65
A.6 Game List – Create a Game ............................................................................................... 65
A.7 Game Lobby – Newly created Game .................................................................................. 66
A.8 Game Lobby – Non-started Game with Agents ................................................................. 66
A.9 Game Lobby – Night time as a Guard ................................................................................ 67
A.10 Game Lobby – Declare your own role as a Villager ......................................................... 67
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.11</td>
<td>Game Lobby – Send a message to accuse a Player of being a Role</td>
<td>67</td>
</tr>
<tr>
<td>A.12</td>
<td>Game Lobby – Agree with a different Player on a statement they have made</td>
<td>67</td>
</tr>
<tr>
<td>A.13</td>
<td>Game Lobby – Overview of a Started Game</td>
<td>68</td>
</tr>
<tr>
<td>A.14</td>
<td>Game Lobby – Overview of a Non-started Game</td>
<td>68</td>
</tr>
<tr>
<td>B.1</td>
<td>Project startup options in Visual Studio 2019</td>
<td>70</td>
</tr>
<tr>
<td>B.2</td>
<td>Building the Client executable</td>
<td>71</td>
</tr>
<tr>
<td>B.3</td>
<td>Successful build of the Client executable</td>
<td>71</td>
</tr>
</tbody>
</table>
List of Equations

4.1 Reward calculation for a turn and time of day whenever an Agent投票 . . . . 48

5.1 Q-learning function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

5.2 Q-learning algorithm – “weighted” Q-value . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
Chapter 1

Introduction

In this chapter, the author will go over the project subject, background information on the topic, alongside any relevant literature. Moreover, the author will define the project’s research question and provide a short description of the contributions which RLereWolf has made.

1.1 Context

The game of Werewolf is a social deduction which is based on Dimitry Davidoff’s Mafia [Braver-\text{man et al.}(2008)] where players are randomly assigned specific roles, at the start of a game, which fall under one of the general factions – evil, neutral, and good. The game is split into two times of the day – day and night. In order for a game of Werewolf to start, there needs to be at least 5 players, with a maximum of 75 players [BGG 2016]. Whilst the complete game of Werewolf provides players with the three factions, in the implemented base game, only two factions will be used – evil and good. In the base version of Werewolf, there exist four roles:

- **Villager** – The core role within the game. Part of the good faction and has no special actions. They can only vote during the day in order to execute a Werewolf suspect. Goal of the role is to kill off all evil faction players.

- **Guard** – A member of the good faction. The Guard can protect players during the night from being attacked, including himself. The goal of the role is to kill of all evil faction players.

- **Seer** – A member of the good faction. The Seer can find out the role of a player during the night time and has the ability to share or withhold any information they learn through divining.

- **Werewolf** – Villagers that turn into Werewolves during the night. They are a member of the evil faction and have the ability to attack other players during the night in their werewolf form. Their goal to kill off all good faction players.

Players must deduce and make assumptions about other players, based off of previous experiences and partial game knowledge, that is – incomplete knowledge of the game. The deduction is aided by having players question each other and communicate in the village forum, conducted during the day time, where a player is voted off for execution. During the night time, some roles either form a conclave with their team members and vote on an action, i.e. Werewolves voting
on who to attack during the night, alternatively some roles have an individual deicing process – _Guard_ deciding who to protect during the night.

### 1.2 Literature Review

To the extent of the author’s knowledge, the only existing framework that offers similar characteristics is _AiWolf_ (Toriumi et al., 2016) which hosts annual competitions amongst the developed agents, by its users (AiW, 2017, 2020). However, whilst _AiWolf_ provides a _Java_ version of the client and server, their implementation does not provide users with any analytic data, has a limited communication protocol, and a limited reach to its core audience – the machine learning community.

Moreover, although covering 50% of the in-game communications used in actual Werewolf matches (Toriumi et al., 2016), the author believes that this can be further improved upon with _conflict resolution messages_, which has proven beneficial other research which uses the Java iteration of _AiWolf_ (Wang et al., 2018). _AiWolf_ does have bindings to _Python_\(^1\) – however, at the time of writing, _Python_ is the de facto language used in the machine learning domain.

Consequently the largest machine learning community has no access to the core implementation of the framework and are unable to do any changes to the game implementation or inherent functionality, in their primary programming language. Moreover, the _AiWolf_ project’s primary natural language is Japanese, consequently not all of its functionality is accessible to English speaking users.

All “solved” or “partially-solved” games, that is – games which have an artificial intelligence agent who has mastered the game and is equal or better in skill than human players, are not classified as _incomplete information games_, but rather are known as _complete information games_ (Kline, 2015). In _incomplete information games_, also known as _Bayesian games_ (Dekel et al., 2004; Ely and Sandholm, 2005), the players do not know the entire game state and have access to a subset of the full game state, where the game state can contain information about the other players or the current player’s “view” of the game world. The reason why “solving” _Bayesian games_ is hard is because one of the core mechanics in _incomplete information games_ – the ability to infer the “missing” from your knowledge, based on partial contextual knowledge and experience, is not trivial to replicate on a computer system.

_Nagayama et al._ (2019) aims to improve upon the _AiWolf_ project, but they mostly targeted a specific role as opposed to the entire set of available roles, i.e. an _Agent_ that is proficient only as a Werewolf (Nagayama et al., 2019), whereas the author aims to create multi-role built-in _Agents_. Moreover, the proposed _Agent_ in _Nagayama et al._ (2019) does not inherently learn, and is rather a _rule-based Agent_ who tries to “swingle” other players, by communicating across to other players that they are a _good faction_ role. The rules which the “stealth werewolf” follow are based on thresholds values, forming a decision tree (Nagayama et al., 2019), which is hard-coded into the _Agent_. Whilst their _Agent_ does provide an improvement on the win-rate of the Werewolf by 65% (Nagayama et al., 2019) to the default _AiWolf Agents_ (Toriumi et al., 2016), it is not an _Agent_ that covers all roles, supported by the game and does not “learn” or adapt to the current game state.

_Wang et al._ (2018) shows an implementation of physical _Agents_, i.e. robots, which can play

---

\(^1\)AiWolfPy – https://github.com/aiwolf/AIWolfPy
the real world game of Werewolf. However, it relies on the AiWolf framework and does not provide any further improvements on the core AiWolf framework, other than the addition of conflict resolution messages – which add an agreement/disagreement option in the communication protocol. Consequently their framework is only targeted towards hardware implementations of AiWolf Agents.

Gillespie et al. (2016) used semantic natural language classifiers to try and understand the communication protocol and have a counter-action based on those player communications. However, this relies on having players consistently sending communications on what their action is. Moreover, if players are not cooperating by sending out frequent communications, then the Agent will rely on “observations” which are part of the project’s proposed future work.

1.3 Motivation

The process of deduction based on partial contextual knowledge is a relatively trivial task for humans. However, it is a difficult computing science problem which has been historically solved with complicated reasoning using various rules, which aim to translate one’s reasoning to its core – sequential if blocks (Kautz et al., 1996; Bünning and Lettmann, 1999). Whilst a framework with similar functionalities exists – AiWolf (Toriumi et al., 2016), the framework’s usability is limited specifically to artificial intelligence development on a specific iteration of the Werewolf game (see Section 1.2 for more information), which this project will improve on. Moreover, AiWolf targets individual roles with separate Agents, whereas the proposed framework will apply a single Agent to all, included in the game implementation, roles.

The goal of the framework is to expedite artificial intelligence research and development, by providing the necessary tools and foundation. Moreover, the project aims to explore the addition of trust and honesty which the agents can use to further mimic the real-life decision making process which would occur in the the context of Werewolf.

1.3.1 Research Question

The framework proposes a new framework which aims to provide analytic tools and built-in Agents which can play all supported by the game roles. Moreover, the improvements to the communication protocol and the addition of trust and honesty factors should perform better than a stochastic player approach and worse or equal to the optimised Q-learning approach. Consequently the project’s goal is to answer the following questions:

**RQ1.** Can the author offer a framework for both multiple non-expert Agents to play the game, and developers to train Agents?

**RQ2.** How can the author improve the existing communication protocol for the Agents?

**RQ3.** How do the proposed Agent behaviour models affect the winning rate?

1.3.2 Contributions

The project aims to improve on the functionality of its existing counterparts, namely AiWolf, and provide a more open API to developers and researchers alike. It does that by providing users with:

- Client – A distributable Game client which connects to a target Server and allows users to play with other human Players or Agents over the internet.
• Development Framework – A development framework which provides the Werewolf Game implementation alongside the analytic utilities which give an insight on the Players’ behaviours and Game’s metrics, i.e. turn count, day count, and winning faction. The framework also includes comprehensive Server and Game activity logging, providing users with a detailed account of the Player’s actions and the Game state. The development framework consists of four sub-systems – Client, Server, Game, and Environment.

• Built-in Agents – Three distinct built-in Agents which target three distinct behavioural models:
  – Dummy Agent – A stochastic Agent that does random valid actions.
  – Rule-based Agent – An Agent with an honesty factor who votes for the least trustworthy, according to them, Player.
  – Trainable Agent – An Agent that can learn from playing multiple games of the current Werewolf Game implementation. Has no pre-existing knowledge of the game and needs to train in order to learn the game’s rules and how to optimally play it.

The contributions of this thesis also include an evaluation of the three built-in Agent behavioural models which takes into account the speed, win-rate, and accuracy performance of the Agent over a series of 1000 Games with varying Game lobby sizes (see Chapter 6 Empirical Evaluation for more information).

Chapter I in a Nutshell

• Incomplete information games are difficult to “beat” because of the need of inference, based on partial game state knowledge, as opposed to complete information games, i.e. Chess.

• Multi-role Agents are a novelty as the primary research focus has been a single-role Agent and the communication protocol.

• RLerewolf contributes to beating the Werewolf game by providing a developmental framework which aims to expedite AI research.
Chapter 2

Requirements

The author will go over the possible scenarios that arise whenever a developer or user is using RLereWolf. The identified scenarios are the basis for both functional or non-functional requirements.

2.1 Scenarios

The Scenarios the author will frame are based on two user groups:

- **Client users** – The users whose primary focus is being able to play Werewolf with their friends and/or Agents (see more information in Subsection 4.3.4 Players).
  - Play Werewolf with other people from some network using an open Transmission Control Protocol (TCP/IP) connection.
  - Play Werewolf with Agents
  - Communicate with other Players (see more information in Subsection 4.3.4 Players) in a game of Werewolf

- **Developers** – The developers are users whose primary focus is the development of a Werewolf game & reinforcement learning (RL) artificial intelligence (AI) development for Werewolf. The developers might be researchers that investigate incomplete information games.
  - Use reinforcement learning to train “agents” to play an iteration of the Werewolf game.
  - Analyze Werewolf game outcomes with game metrics.
  - Develop expansion packs for the base Werewolf game.
  - Deploy a server that can host multiple Werewolf games.
  - Deploy a client that can connect to a server and play Werewolf.

2.2 Functional Requirements

The functional requirements the author has identified are based on the expected scenario outcomes, as well as any subsequent implied behaviours – exception handling, recovery, and analysis data generation. As RLereWolf contains four subsystems, the author has split their functional requirements into four subsections, namely: **Server**, **Client**, **Game**, **Environment**.
2.2.1 Server

The functional requirements for the Server reflect their ability to manage multiple instances of the Werewolf Game, as well as support multiple connections from Clients. The Server should be able to keep track of all connections and requests, and keep a historic log.

SFR1. **The server can host multiple Games** – RLereWolf must be able to host multiple Game lobbies of Werewolf, each independent of the other.

SFR2. **Multiple clients must be able to establish a TCP/IP connection** – Clients targeting the server should be able to connect to it without any artificial limit connection limit. However, the connection limit is based entirely on the available to the server’s available resources – internet speed and CPU speed.

SFR3. **Must accept only valid client requests** – The Server should validate any incoming requests and only accept requests which are sourced from an active TCP/IP connection. The requests must be full and non-malicious, that is – not modified by any third-party, other than the Client itself.

SFR4. **Keep track of active TCP/IP connections** – All Clients which are connected to the Server are logged and kept track of. At any given point of time, the Server is able to tell which Client is connected to the server by keeping track of their: Client name, Client identifier, IP, and Port.

SFR5. **Disconnect inactive TCP/IP connections** – The Server checks whether or not a connection it has logged is still able to receive data from it. If the connection is unable to receive data from the server, meaning it is no longer active, then the connection is disconnected and no longer tracked by the Server.

SFR6. **Log connection activity** – Every time a Client connects to the Server, an entry is logged on the Server logs, stating the IP and port pair, alongside the number of active connections.

SFR7. **Log request activity** – Every time a Client makes a request to the Server, an entry is logged on the Server logs, stating the “type of request” a Client has sent.

SFR8. **Can be built into a single executable package** – The Server can be built into a single executable as a mean of deploying it to a target host.

2.2.2 Client

The functional requirements for the Client are targeted towards the client user base, whose primary focus is being able to play the Werewolf Game, hosted on some Server.

CFR1. **Has a graphical user interface (GUI)** – The Client has a graphical user interface which provides users with the means to connect to a Server and play a Game (See Section 4.2 for more information).

CFR2. **Can connect to a Server** – The Client is able to connect to its target Server after specifying a name, which the Server and other Clients can use as identification, when paired with the Client identifier and their corresponding IP & port pair.
2.2. FUNCTIONAL REQUIREMENTS

CFR3. Can disconnect from a Server – The Client can disconnect from the Server “gracefully” at any given point of time, that is – without any run-time errors on the Client machine.

CFR4. Must be able to create a Game – Once connected to a Server, Clients will be able to create a Game lobby on that Server which other Players can join.

CFR5. Must be able to join a Game – Clients are able to join any Game hosted on the Server, as long as the maximum Player count for the Game has not been exceeded.

CFR6. Must be able to recover from any minor run-time failures – The Clients should be able to recover from a failed request to the Server, or minor Client run-time exceptions, which do not break the connection between the Server and the Client. Any Client specific exceptions related to the GUI or Game & Player state handling, can be recovered from.

CFR7. Can play Werewolf with other people – Once a Client joins a Game, other Clients can also join it and play the hosted version of the Werewolf Game together.

CFR8. Can set themselves as ready or unready in a Game – Once a Client joins a Game, their readiness state is set to unready. They can set their status to ready, whenever they would like to start the Game. All Players must be ready for the Game to start.

CFR9. Can play Werewolf with other Agents – Once a Client joins a Game, the Client can add Agents to the Game, before a Game has started. The Agent Players, which join the Game, have a readiness state of “ready”.

CFR10. Can be built into a single executable package – The Client can be built into a single executable as a means of distribution to users.

CFR11. Can communicate with other Players – The Client should be able to send preset messages from a finite set of “classified” messages to other Players within the same Game.

2.2.3 Game

The functional requirements for the Game reflect on the base Werewolf game [BGG, 2016] implementation element and what its Players can do.

GFR1. Allow multiple Players to be in a Game – The Game instances should be able to support multiple Players concurrently playing the Game, with a set maximum per instance.

GFR2. Have the base four roles in Werewolf (Villager, Guard, Seer, Werewolf) – The Game must have the base Werewolf roles included which are assigned, randomly, to Players in a Game, on its start.

GFR3. Roles have different abilities – Each of the base roles in the Game, should have their distinctive features and abilities that can be used at different times of the day/night within the Game.

GFR4. Players can only make valid actions – The Game should validate any action, such that every Player can only do permitted by the game rules action, at the current state of the Game.
GFR5. Can have humans and Agents in a single Game instance – The Game instances should be able to have humans and Agents, both referred to as just Players, in a single Game, in which they can play together.

GFR6. Log Game & Player states – Every action done by a Player is logged within a Game log, alongside the “complete” game state, that is – all information about the Game and its Players. The log must be stored on the Game host machine, if the Game is on a Server – then the logs will be on the Server machine.

2.2.4 Environment
The functional requirements for the Environment are based around the reinforcement learning aspect of the framework, and what it must provide developers with.

EFR1. Allow multiple Agents within the same environment – The RL training Environment must support multiple different Agents within it. This is because RLereWolf aims to support Agent exclusive games in which the Agents are able to train in games without the need of a human Player.

EFR2. Provide developer with game metrics, used to analyze the Player performance – The Environment must provide useful Game metrics, which can be used for Agent analytics under diverse conditions.

EFR3. Training Environment should be optional – The training Environment needs to be an optional element as the framework aims to provide a modular package for its developer user base. This means that only some Games will allow for Agent training and analysis.

EFR4. Have a default stochastic Agent – RLereWolf must come in with an integrated Agent, which does random actions.

EFR5. Have a default rule-based Agent – RLereWolf must come in with an integrated Agent, which does deterministic actions, based on some arbitrary rules.

EFR6. Have a default trainable Agent – RLereWolf must come in with an integrated Agent, which can learn to play the Werewolf Game iteration that the Agent is in.

2.3 Non-functional Requirements
The non-functional requirements describe the inherently expected behaviour from the system with regards to features such as usability, scalability, and the performance of RLereWolf’s components.

NFR1. Server and Game reliability – Run-time errors on the Server & Game must be handled and logged, to avoid the possibility of having a crash whenever Clients are connected to the Server and/or Game. Both Server and Game will always try and recover from run-time errors.

NFR2. Client reliability – Whilst the reliability of the Client is not nearly as important as the Server’s or the Game’s, all run-time errors should be either prompted to the user or recovered from.
NFR3. Maintainability of all modules – Each module should have well documented code, with easy to navigate classes and structures so that it is easily expandable on.

NFR4. RLereWolf must work on modern operating systems – As RLereWolf is written entirely in Python 3.7, the framework’s compatibility with operating systems is equivalent to that of Python – capable of supporting modern and legacy systems – Linux, Windows Vista and newer, FreeBSD 10 and newer, and MacOS 10.6 and newer (PythonDev, 2018).

NFR5. Portability of the Client & Server – The Client and Server should be in a portable format that is compatible with all, supported by Python 3.7, operating systems.

NFR6. Games should never get stuck other than on awaiting Players’ actions and should conclude after some finite amount of time – There should be no situation in which the Game gets stuck in a state of awaiting someone who is no longer viable for an action with a game turn. That is, the Game should reliably track the Player actions and record each one, so that it never gets stuck.

Chapter 2 in a Nutshell

- RLereWolf provides a framework with four comprehensive and independent subsystems – Client, Server, Game, Environment.
- RLereWolf’s target audience is Werewolf players and Werewolf/Incomplete information game AI researchers.
- All four subsystems must have their core functionality implemented, as well as be resistant to run-time issues.
Chapter 3

Technologies

In this chapter, the author will go over the technologies and methodologies used for RLereWolf, accompanied by a discussion for each technological decision, its implications, and future.

3.1 Programming Language

In the beginning of the project, the author had two primary candidates for a programming language with which the initial iteration of RLereWolf would be built with, namely:

- **C#** – A compiled, general-purpose, type-safe programming language, which has access to the .NET framework and its multitude of GUI, asynchronous programming and language integrated query (LINQ) utilities (Hejlsberg et al., 2003, 2008). As a result, the author would have access to highly optimised frameworks for building a Server-Client infrastructure, a GUI client and would be assisted with the aforementioned type-safety and immutability, which – in the author’s experience, is essential for large projects, such as RLereWolf. Moreover, the author has had 2 years of professional experience with C# and the .NET stack.

- **Python** – An interpreted, general-purpose, dynamically-typed programming language whose philosophy is heavily based around code readability, by sticking as close to natural languages as possible (Van Rossum and Drake Jr, 1995; Oliphant, 2007). Although Python’s “out of the box” frameworks and packages are enough for the development of RLereWolf, it would need to greatly rely on open-source frameworks to aide the author in the framework’s development. Furthermore, Python’s vast selection of machine learning frameworks allows for the framework’s use in multiple machine learning fields, other than the targeted reinforcement learning field (Pedregosa et al., 2011; Raschka, 2015). Although the author has no professional experience with Python, they have 4 years of academic experience which mostly revolved around Python’s usage.

Even though there is the possibility of using both languages for the different modules of RLereWolf, the author decided to stick with a single language for consistency in the initial iteration of the framework. Given the two potential language candidates, the author’s decision for an initial programming language was Python.

Although, the performance of Python is sub-par to C#’s (KARACI, 2015; Fourment and Gillings, 2008; Srinath, 2017), the author’s decision to go with it is entirely based on the potential user base and their focus – machine learning. More specifically, the author has chosen Python
3.2 Packages

In this section, the author will discuss the “direct” packages used in the implementation of RLereWolf and their role within the project. By “direct” the author means that they will only look at packages as a whole, without any of their inherent dependencies.

3.2.1 Serialization

A big part of RLereWolf is the client-server infrastructure and the “communication” between the two subsystems. Consequently, the need for a standardised & reliable byte serialization framework which supports the usage of classes, functions, and lambda functions arose, as both client and server subsystems have significant overlap in the used by them data transfer objects (DTOS) (see Chapter 4 [RLereWolf Architecture] for more information). An example of such a data transfer object is the game state information, which contains the known to some Player information for the current state of the game. For the task, the author chose cloudpickle, which is an extension of the Python integrated pickle object serialization framework.

3.2.2 Names & Locale

The framework’s game implementation allows for multiple Players to play together or with Agents. As the game is heavily focused around communication and assumptions for other Players, the author decided to use a random name generation package, which is based off of the locale of the game host, typically this would be the server, which would generate names whose players can easily remember and uniquely identify within a game. The names, generated by Faker, are assigned to Agents which have been added to some game lobby. Whilst in the current iteration RLereWolf does not use Faker for anything more than name generation, the author has planned, at a later stage, the addition of “profiles”. The “profiles” can be generated by the Faker library for the Agents. The Agent “profiles” will give an estimate, to other Players, what their game personality might be, that is – how they are likely to behave in the game.

3.2.3 Graphical User Interface

As the time scale of the project did not allow for a fully-fledged “game-like” graphical user interface, the author decided to go with the standard in Python GUI framework – TKinter (see Section 4.2 for more information).

However, the author has additionally relied on Pygubu, which is a Rapid Application Development (RAD) tool whose purpose is to

---

1PyTorch issue with Python 3.9 – This is due to the lack of wheels support for Python 3.9 [https://github.com/pytorch/pytorch/issues/47776]
increase implementation speed by providing a GUI builder tool, which creates descriptions of the
graphical user interface as Extensible Markup Language (XML) [Bray et al. (2000)] files. The au-
thor has also additionally supplemented Pygubu, by adding in renderers for the generated by it
XML files. Moreover, Pygubu allows the majority of the design to be primarily designed and laid
out in its builder (see Figure 3.1), whilst functionality and complex behaviours are defined in the
aforementioned renderers (see Section 4.2 for more details).

![Figure 3.1: Pygubu Designer – the RAD graphical user interface builder tool](image)

3.2.4 Machine Learning

Whilst RLereWolf comes with the tools for further development of the Werewolf game and a
client-server infrastructure for it, the primary target user base are researchers and machine learning
developers. Whilst RLereWolf could be used with other machine learning techniques, due to
its modular design (see Chapter 4 [RLereWolf Architecture] for more information), its target is
reinforcement learning (RL).

As such, the author is including a reinforcement learning training environment and a trainable
agent (see Sections 4.4 and 5.3 for more information), which can be used as an introduction to the
framework, by machine learning enthusiasts. Consequently, the author’s choice for a reinforce-
ment learning framework is OpenAI Gym and its open-source extension Ray RLlib [RLlib (2020)].
They provide the author with the necessary environment-agent loop functionalities, alongside the
ability to create multi-agent training environment in which multiple Agents can play against each
other and optimise against their individually learned styles of play.

Moreover, the RLlib application support can easily be integrated in the framework’s built-
in reinforcement learning algorithms, as well as allow for the further development of RL-based
custom algorithms for some multi-agent environment (see Figure 3.2).
3.3 Methodologies

The development approach chosen by the author was an iterative approach with a focus on providing the minimal viable product (MVP) (Moogk, 2012; Lenarduzzi and Taibi, 2016). Consequently, the author’s focus was to provide a framework which covers the functional requirements, on an at least partial state, such that none of the non-functional requirements. This means effort has been put into all of the functional requirements and are concurrently addressed, which was managed with the assistance of the author’s supervisor.

The author proposed a loosely agile-based approach which consisted of week-long sprints in which the author would focus on a particular RLereWolf sub-system and its core functionalities which implementation details were split onto several tickets (Schwaber and Beedle, 2002; Schwaber, 2004). Each ticket had well-defined & small task which combined with the other sprint tickets, would complete parts of the functional requirements for some sub-system.

The author relied on git version control to keep a record of all changes and GitHub’s metrics to keep track of their progress. The repository, owned by the author, which RLereWolf is hosted on can be found on the following address: https://github.com/GeorgeVelikov/RLereWolf

The metrics provided by GitHub were an important part of making sure the project’s development timeline was followed. Furthermore, they provide a complete list of the design decisions the author has made, as a frequent commit principle was followed – i.e. the author would split up tasks into small sub-tasks and make commits for each one. The separation of the tasks inherently leads to numerous small commits (see Figure 3.3 where the orange graph represents the commit count over the period of 17th January to 2nd May).

3PyInstaller – https://www.pyinstaller.org/
Alongside detailed and frequent commits, the author has also employed the software engineering industry standard method of *verbose coding* ([Buse and Weimer, 2009](#); [dos Santos and Gerosa, 2018](#)). This standard states that the code written by a developer should be clear and coherent enough to be understood as functionality, whereas any code comments should act as a documentation for the reasoning behind the design choices. However, employing *verbose coding* does lead to some possible issues – whilst easily readable and understandable by developers, it does mean that developing will take longer and code files will require more disk space in comparison to a more “minimal” or *cryptic* approach. That being said, the author’s decision on employing *verbose coding* was not only a personal choice, but an inherent design choice, as the code within RLereWolf itself, is the tool that is offered to developers.

### 3.4 Development Hardware & Software

The development of the framework was done on a single machine with the aide of the preferred by the developer [Integrated Development Environment (IDE)](https://github.com/GeorgeVelikov/RLereWolf) – Visual Studio, and a graphing tool – [Draw.io](https://drawio-app.com/). Whilst, the framework has no inherent system requirements other than the requirements for Python 3.7 and *Tkinter*, development speed with RLereWolf is highly dependent on the CPU of the host machine.

As development of RLereWolf was done with an IDE, maintenance on the framework would be done, ideally, with the same configuration as the author (see Table 3.1) which is described in-depth both in Appendix B [Developer Manual](https://github.com/GeorgeVelikov/RLereWolf) and on the project’s [GitHub repository](https://github.com/GeorgeVelikov/RLereWolf).

---

4 Draw.io – https://drawio-app.com/
### Component Description

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware</strong></td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Core i7 6700, 4 cores, 3.4 GHz (up to 4.00 GHz)</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia Quadro P2000, 5 GB GDDR5</td>
</tr>
<tr>
<td>RAM</td>
<td>4x8 DDR4 2133 MHz</td>
</tr>
<tr>
<td>OS</td>
<td>Windows 10 Pro</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td></td>
</tr>
<tr>
<td>IDE</td>
<td>Visual Studio Professional 2019 16.8.6</td>
</tr>
<tr>
<td>Graphing Tool</td>
<td>Draw.io</td>
</tr>
</tbody>
</table>

**Table 3.1:** Hardware used during the development of RLereWolf

---

**Chapter 3 in a Nutshell**

- **RLereWolf** is a Python 3.7 based framework which employs a loose interpretation of the Model-View-ViewModel design pattern.

- **Client** relies on Tkinter for its graphical user interface (GUI) which has a dedicated GUI development pipeline.

- The framework has employs multi-agent environment as opposed to the commonly used single-agent environments.

- The development of RLereWolf was focused on a minimal viable product and weekly iteration showcasing.
Chapter 4

RLereWolf Architecture

In this chapter, the author will describe the architecture of RLereWolf, its logical separation into components and how the components interact with another, to create a request-response loop (Mozilla 2015; FMC 2004) between the server and the client.

The overall architecture of the framework (represented in Figures 4.1 and 4.2) consists of four major components, which create a request-response loop, namely:

- **Server** – The Controller and middle-man between clients and the game who validates that the actions specified by the client are valid and obey the game rules. As a middle-man, the server accepts data from the clients, checks its validity and passes it to the game. Once the game replies to the server, the server serializes the reply and sends it back to the client (see Section 4.1 for more details about the server).

- **Client** – The Graphical User Interface (GUI) which acts as the input for the user to send requests to the server in order to play a game hosted on the server (see Section 4.2).

- **Game** – An implementation of the base Werewolf game which is moderated by the server. Multiple Werewolf games can be hosted on the server (see Section 4.3).

- **Environment** – The training environment and agents supported by the framework (see Section 4.4). The training environment is an extra layer wrapped around the Game. The environment keeps track of the game state and assigns rewards to Agent Players. It is an additional feature that any game hosted on the server can have.

The request-response loop of RLereWolf starts with the client sending a Packet to the server, which is either redirected to the packet handlers (bottom of Figure 4.3) or discarded by the Packet Handler Context (bottom of Figure 4.3), depending on whether the Packet is successfully validated. Each color in Figure 4.1 represents the request-response flow for some client, e.g. orange is the packet transfer of Client Beta, blue is the packet transfer of Client Alpha, and magenta is the packet transfer of Client Gamma. Once validated, the Packet gets redirected to its corresponding Packet Handler and is unwrapped by it. Once the Packet is processed, it is targeted as an Action on the Game, the client is in. For example, Client Beta can be seen wanting to send a message in Game Y (the orange lines in Figure 4.1).

The Client connects to a specifically targeted server which hosts multiple Games and any associated Environments with them. Each of the aforementioned components is treated as a separate, self-contained project. By project, the author means the code package separation of RLereWolf.
This is done with the intention of providing developers with an easy way to just use parts of the framework that they are interested in, e.g. someone purely interested in AI development would opt to only use the Werewolf project, which contains the Game and Environment modules.

The Client-Server architecture uses TCP/IP sockets and the communicated data is wrapped in serialized “Packets” which contain various metadata about the TCP/IP request and response, which includes data for the validation and redirection of the request (see packets after the Entry in Figure 4.1).

The packets also contain Data Transfer Objects (DTOs), which are miniature models that are serialized and contain the minimum amount of required shared data between some sender and receiver, whose purpose is the decoupling of the sender and its data source, i.e. some data store/database [MSDN 2014]. The DTOs are shared between the server and client and are the minimum functionally required data by either side in order to handle a request/response. The usage of DTOs allows the shared data to be used as a read-only entity (see Figure 4.4 where each
4.1 Server

The RLeReWolf server instance can moderate multiple Werewolf games which are concurrently ran with diverse combinations of players – be it entirely human players (Game Y, Figure 4.1), entirely “Agent” players (Game Z, Figure 4.1), or a mixture of the two (Game X, Figure 4.1).

The server’s abstraction of received data and the Handler Context allows it to only work with “valid” data and reduce the potential for any sort of corrupt or invalid requests being processed. Consequently, this reduces the possibility of run-time failure and increases server stability.

Moreover, the server instance keeps track of its connections and automatically disconnects dead connections which may be a result of some unhandled run-time client exception or a communication failure – be it loss of internet connection or loss of a disconnect packet.

Figure 4.2: Project Layout – as seen in Visual Studio 2019’s Solution Explorer.

Figure 4.3: Server architecture – receiving packets.
4.1.1 Packets

The data passed between the client and the server is wrapped in *Packets* which allows for easy expansion and debugging, as tracking the "movement" of each of the data packet is well defined and easily traceable. Since each packet is identified by a *PacketTypeEnum* (see Figure 4.4), the framework can log what is received by the server (see Figure 4.7).

The wrappers allow for consistency in terms of validating and redirecting specific packets in the server. This is needed as the framework aims to provide developers with an easily modifiable and expendable implementation of the Werewolf game. *Packets* usually contain the source (Client) connection identifier, the target (server) connection identifier and the packet type, e.g. *VotePlayer Packet* (as seen in Figure 4.5), as they are primarily used to encapsulate data from the Client, to the server.

Moreover, the encapsulation in Packets, inherently adds the support for backwards compatibility in versions. To elaborate, version X+1 of the server will inherently support version X of the client, as long as the Packet type has the same data properties, where any new properties will be defaulted to their default type value upon deserialising and validating on the server.

![Figure 4.4: Game State Packet (GameDto) structure.](image)

```
Packet
  x PacketType: PacketTypeEnum
  x Data: byte[]
```

```
PacketTypeEnum
  x Connect: int
  x GetGamesList: int
  x CreateGame: int
  x JoinGame: int
  x LeaveGame: int
  x GameLobby: int
  x AddAgent: int
  x RemoveAgent: int
  x VoteStart: int
  x Talk: int
  x VotePlayer: int
  x Whisper: int
  x AttackPlayer: int
  x DivinePlayer: int
  x GuardPlayer: int
```

```
GameDto
  x Identifier: Guid
  x HasStarted: bool
  x Name: string
  x Messages: MessageDto[]
  x Votes: VoteDto[]
  x Players: PlayerDto[]
  x Turn: int
  x TimeOfDay: TimeOfDayEnum
  x PlayersCount: int
```

4.1.2 Handler Context

The *Handler Context* is a wrapper that contains all of the registered server packet handlers and is also responsible for the validation and redirection of packets received by the server. It can be described as the traffic controller for a machine that is running a RLerewolf server instance.

The registered packet handlers within are "business logic" units which are separated by general functionality. Whilst the hierarchy can span multiple levels, the author has implemented a two-level hierarchy. The hierarchy consists of the Packet Handlers and Handler Context (as seen in Figure 4.3) for RLerewolf, as further abstraction and nesting does not provide a huge benefit, due to the relative small scale of the needed packet handlers for the functionality of the base Werewolf game. Expanding this into a three-level hierarchy will result in creating sub
packet handlers, which will divide the functionality of one specific packet handler. The implemented structure, closely resembles a tree data structure, where the Handler Context is the root node.

The handlers also exist in the service in a global space, meaning that all the hosted games use the same handlers. This is done because of performance reasons, such that creating a handler context for each game will greatly reduce the load times. The trade-off is that the packets sent by the clients must contain valid metadata which hold a valid game identifier they want to do some action in. Furthermore, handlers are coupled with the server and its handler context. This coupling servers as a means of “registering” handlers to a specific server-context pair. As a result, expanding the framework with additional handlers is a trivial task.

4.1.3 Packet Handlers

The packet handlers, as previously mentioned, handle the validated and redirected packets. Each packet handler then processes each packet by “breaking” it down and acting upon the specified metadata and value data.

The server contains a “Handler Context” which is the formal entry point for the server and it serves as a validation and redirection point to the various specific handlers, i.e. “Game Lobby Handler” or “Game Action Handler”. The modular design of the server allows for a formal separation of concerns, similar to how a client would have separate.

Each packet type is redirected to a specific method in some Packet Handler. However, there could be the scenario where the framework can also accept the use case of having various PacketTypeEnum (see Figure 4.4) going to a single method within a Packet Handler. An example of this in the GameActionHandler, where a Wait action is dealt as, the appropriate for the player type, an empty Action. This enables the framework to react dynamically to some Packet, as the framework could be easily programmed to redirect packets differently, based on some variable – be it the game state or the player state.

4.1.4 Logging

Due to the nature of the tools included in the framework, the author has also developed a basic logging infrastructure to record any errors and general operations on the server and any of its hosted games. The logging tool has a few default distinctions in the messages it logs, namely:

- **Error** – Logs created by the server whenever a run-time exception is hit which contains a timestamp and a server stack trace. An error message can also be manually logged on specific events within the server.

- **Warning** – Logs which warn the developer that there might be some, potentially instance breaking, error. As of the time of writing, the framework’s only warning messages are logged whenever the intrinsic game rules are broken, e.g. having no villagers in a game. Another warning message is whenever some user has tried to join a full game.

- **Information** – Logs which are used to just record the state of either game or server. These are mostly used to keep track of dynamically changing values, e.g. the active connections counter (see Figure 4.5).
4.1. Server

- Request – Logs which keep track of the Packet types received by the server at any given time. These messages can be quite chatty and can easily be disabled in the server’s initialization calls. However, they are useful in tracking user behaviours and having a rough log of steps in order to recreate some exception.

- Message – This log message type is treated individually and it records any messages sent from either server or any of the players, be it Agent or human (see Figure 4.7).

The logging is split up into folders within the server instance directory. The server creates a folder “Logs” if there does not exist one already and creates a directory for the day it is logging for, e.g. “17-03-2021” (see Figure 4.6). Each folder contains the log files of the server, which records general connect/disconnect calls, any handled and unhandled run-time exceptions, as well as a count of all of the active connections to the server.

![Figure 4.5: Server log – recording the connections it replies to, alongside the request packet types the server receives.](image)

![Figure 4.6: Logs file structure](image)
Furthermore, alongside storing the general server operations in a log file, the framework also logs all of the actions and effective game states, that is - all known game information that has been distributed amongst players. Each log file consists of the game name followed by the unique game identifier e.g. “Game Beta - f56974e2c1dc4e1da363d1d6f6eae9e9”. It must be noted that the game identifier does not change on a game restart, it is an effective constant that is created once a game lobby is created by either clients or the server itself.

Using identifiers allows for multiple games with the same name to be created, without having any identification issues, as the game names are only used for display purposes. Internally the framework refers to players and games by their assigned universality unique identification (UUID) (Leach et al., 2005), also known as a globally unique identifier (GUID) (MSDN, 2017) (see Figure 4.6).

The logging functionality built-in to the framework aims to ease development work by giving a verbose description of any game actions to ease state recreation, and any run-time exceptions will also be logged with their appropriate full stack-trace. Although the server should continue working on some run-time exception, it might be the case that it breaks the normal state flow of the server. Having the verbose logs helps development by pinpointing the exact failure point.

4.2 Client

The RLerWolf client provides end users with a Graphical User Interface to the base game implementation and relies on the supported functionality by the server. The client operates on a “screen-swapping” basis, such that there exists a main window to the client with a content frame which we can regard as the display buffer (see Figure 4.8). Every view of the client is then displayed on the screen by swapping out any existing view on the display.

Figure 4.7: Game log stored on the server, recording the full game state.

The logging functionality built-in to the framework aims to ease development work by giving a verbose description of any game actions to ease state recreation, and any run-time exceptions will also be logged with their appropriate full stack-trace. Although the server should continue working on some run-time exception, it might be the case that it breaks the normal state flow of the server. Having the verbose logs helps development by pinpointing the exact failure point.

4.2 Client

The RLerWolf client provides end users with a Graphical User Interface to the base game implementation and relies on the supported functionality by the server. The client operates on a “screen-swapping” basis, such that there exists a main window to the client with a content frame which we can regard as the display buffer (see Figure 4.8). Every view of the client is then displayed on the screen by swapping out any existing view on the display.
buffer. This buffer approach allows for an instant transition between the various client pages as a transition only happens once the new page has been fully initialised.

![Client architecture diagram](image)

**Figure 4.8:** Client architecture.

### 4.2.1 Graphical User Interface

The client’s graphical user interface is built with the help of Pygubu\(^1\) an extension of Python’s integrated bindings to the open-source, cross-platform Tk Graphical User Interface (GUI) kit – Tkinter (Python, 2012). The Tk toolkit is based on widgets\(^2\) and it comes with various, commonly-used, widgets, e.g. button\(^3\), label\(^4\), listbox\(^5\) etc., alongside a layout hierarchy, which allows the construction of various grid, list or absolute layouts.

The use of Pygubu helps the mimicking of the Model-View-ViewModel (MVVM) (MSDN, 2009) design pattern as it separates any User Interface (UI) definition elements from the code files,

---

2. Widget – a small self-contained component, usually digital.
3. Tkinter Button – [https://tkdocs.com/widgets/button](https://tkdocs.com/widgets/button)
by defining the user interface in a .ui file, whilst providing basic data binding to most of the UI controls.

These UI files are then processed and rendered on run-time by the Pygubu Builder and the accompanied infrastructure created by the author and act as the View. The infrastructure consists of renderer classes, acting as ViewModels, and the Models are the appropriate client-server shared DTOs, i.e. the game state or individual player state.

The written by the author renderers, create the GUI during the framework’s runtime, based off of the preemptively created .ui files (see Appendix A User Manual for more information on the GUI). The design of the graphical user interface pipeline is based on Windows Presentation Foundation’s (WPF) functionality, where the MVVM pattern is preferred (Sells and Griffiths 2007; Sorensen and Mikaelsc 2010).

The GUI consists of three major screens – Main Menu, Game List, and Game Lobby (see Appendix A User Manual for more information). The Game Lobby screen is the actual Game which users can play (see Figure 4.9 for the layout and its elements).

![Figure 4.9: Game Screen – Annotated elements](image)

All of the navigation and GUI manipulation is abstracted through a UI Context which serves as the effective controller for the client. The UI Context itself is part of the overall ViewModel Context (top part of Figure 4.8) of the application. For readability purposes, the author will refer to the ViewModel Context as Context.

### 4.2.2 ViewModel Context

The client’s Context is similar to the aforementioned server’s Handler Context (4.1.3 Packet Handlers), in the sense that it aims to separate the various concerns into contexts. The defined by the
4.2. Client

author contexts, accessible to all ViewModels, so far are as follows:

- UI Context – GUI navigation
- Service Context – server API calls

The architecture is such that it can easily be broken down into multiple child levels (Figure 4.8). The service context can grow to a larger scale and be separated into multiple services which are instantiated in the service context. This allows for a verbosely targeted and coherent API structure. Whilst it does not inherently provide any performance benefits, it does provide the framework with a modular design so that it is easy to plug and unplug various context functionalities.

The Context is shared between all screens of the client and is one of the first entities to be created once a client run, as the context holds all of the logic, neatly separated into corresponding sub-contexts.

4.2.3 Identification

At the current state of the framework, the clients are not remembering their identifier and subsequently create a new one on each launch. The identifiers are a crucial part in the server-client communication protocol.

Once a client connects to the server, the server keeps a reference of the client connection details and their identifier, which is how the client and server can effectively recover from minor run-time errors. A particularly important handling was the failure of complete packet transmission on either client-to-server or server-to-client route. This is because there exists the possibility of a request-response loop getting preemptively broken as a result of some run-time exception and the receiving side would end up being stuck on awaiting some Packet.

All awaited calls are done on background threads which are interrupted if they exceed their allocated time frame. This allocated time frame in which a request-response loop should succeed is based on the poll rate. The reason for awaited calls as opposed to “fire and forget” type calls is because the receiver requires the data for some functionality.

4.2.4 Polling

Due to the nature of the client, there needs to be a clear separation between main (UI) thread actions and background (child) thread actions. Most actions are done on the main thread as the usual client expectation is as follows:

1. Press button
2. Start event
3. Optionally get a result by awaiting the event handler

However, whenever the client asks for updates on the game state or the player state, those updates should be done in the background as they are not explicitly started from the user. As a result, the update calls are scheduled by default to happen once every second in order to keep the client synced with the server, which acts as the game moderator.
Consequently, due to having to poll at a specific time delta, the clients are prone to desynchronising (Cronin et al., 2001; McAnlis et al., 2014). This can be a direct result from an unhandled run-time error, temporary connection loss or corrupt Packets. However, desynchronisation in RLereWolf is not a huge issue in its current iteration, as the client interface is not complex, in the sense that it does not prompt any real-time dynamic animations as a result of the data transfer. The only noticeable effect of a desync is that the client will skip an update frame, only to catch up on the next successful sync call.

The game state polling retrieves data to the user that they do not have. This is done with a timestamp of the last sync time. Every time a client polls for data, the server serializes and packages all game state within a GameDto (see Figure 4.4) with data that is created after the provided by the client UTC timestamp, and grabs the current server UTC time as soon as the GameDto is created. The server then packages the GameDto, the game state snapshot timestamp and other miscellaneous meta data, e.g. target client identifier, and sends it back to the client. Once the client receives the UpdateEntityDto (see Figure 4.10), which is a layer between the Packet and GameDto, it sets a client global value of a timestamp, which records the last time the client has received a snapshot of the game state, for their current game. Then once a client has to poll for data again, it sends off a Packet with that timestamp, which is used by the server to package the game/player state into the minimal possible package.

![Figure 4.10: Packet Updated Entity DTO.](image)

Keeping a track of the time where the last snapshot was created allows for the server to send the bare minimum of data required to the client. This increases performance and increases general server stability, as the server wouldn’t need to send the entire history of messages, as the client is responsible for storing those in their game lobby instance. This means that once a client leaves the

---

6Desyncing - Originates from desynchronisation (desync) and describes the state where a client does not hold the most up-to-date data.
game lobby, that message history is lost.

Moreover, all regular polling events need to have a timestamp which indicates when that particular entity is form – be it the game state or player state. Having a generic \texttt{UpdatedEntityDto} wrapper allows for its usage with multiple types of data objects that are part of a polling process.

## 4.3 Game

The \textit{Game} is the implementation of Werewolf that comes in the current iteration of RLereWolf. \textit{Games} can either be hosted on the server, or played locally with agents. This was an important part of the architecture design, because the author’s target was to create an easily modifiable game structure, which can be \textit{played} offline, online and used for training, which would usually be done \textit{offline} as to reduce overheads and training time from some client-server connection. The \textit{Game} is the base Werewolf game, which consists of four roles, namely:

- Villager – A villager in the town, with no special abilities.
- Guard – A villager in the town, capable of protecting someone during the Night from an attack. However, \textit{Guards} have a limited amount of uses on their special ability. RLereWolf has defaulted it to one.
- Seer – A villager in the town with the capabilities to look into their crystal ball at night, and tell whether a certain villager is a werewolf.
- Werewolf – A villager in the town who transforms into a werewolf at night and attacks other villagers.

Each role contains the basic description for it in the form of flags and counters. The flags specify whether a role can do a specific action, usually separated by \textit{day} and \textit{night} actions. However, there exist roles which have a limit to their actions. An example of such roles is the Guard, who can only protect other players in the night time a fixed amount of time, usually just once. This behaviour is managed through each of the roles classes as they serve as a configuration for the role itself in that specific implementation of Werewolf.

<table>
<thead>
<tr>
<th>Role</th>
<th>Day Action</th>
<th>Night Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Villager</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Guard</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Seer</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Werewolf</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

\textbf{Table 4.1:} Base Werewolf game roles – possible time to act

Moreover, the game’s default configuration is to distribute roles and allow a \textit{start} whenever there are at least 5 players and at most 75 players, as per the game rules (BGG [2016]). The role distribution and player count can be modified by the game constants configuration file, which defines the minimum, maximum amount of players, alongside the villager : $X$ ratio, where $X$ is a non-villager role (Guard, Seer, Werewolf).
Whilst this could introduce role distribution issues, such as a distribution in which there would be no villagers due to overlapping ratios, this is entirely down to the developer to consider, as the current iteration of the framework does not support the standard point-based role allocation technique, commonly used in Werewolf games (BGG, 2016).

### 4.3.1 Roles

The roles in the framework are built using a common `Role` class. This allows for easy expansion and addition of new roles, as the only requirement is adding in the role specific definition “rules”, which are inherited from the common `Role` class as abstract behaviour. As briefly mentioned previously, RLereWolf does not support a point-based role distribution system in its current iteration.

This could be a problem as the role type count for some game will not be as diverse. This is because RLereWolf will have a consistent role distribution mapping for some player count and some role distribution ratio configuration – which can be modified within the `GameConstants` and `GameRules` files. However, due to the target of the framework – the development and training of reinforcement learning agents, the author has decided to use the consistent approach of using a ratio-based system. That is, each role will have an additional member from it, based on some overall player count (see Table 4.4 for currently used ratios). It must be noted that the `Villager` has no inherent ratio, as it is assigned to the remainder of the non-special role assigned `Players` in the `Game`.

<table>
<thead>
<tr>
<th>Role</th>
<th>Ratio (Role:Players)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Villager</td>
<td>N/A</td>
</tr>
<tr>
<td>Guard</td>
<td>1:10</td>
</tr>
<tr>
<td>Seer</td>
<td>1:15</td>
</tr>
<tr>
<td>Werewolf</td>
<td>1:6</td>
</tr>
</tbody>
</table>

*Table 4.2: Role-ratio values used by RLereWolf*

Using a point-based distribution system will mean that summing up the distributed points a role gives, has to be as close to 0 as possible, to insure a theoretical state of equilibrium between the **bad** and **good** roles, where they are categorised as **negative** and **positive** points respectively (see Table 4.3). Reaching 0 points is not always possible with a role-point distribution, particularly with a small set of playable roles, due to the nature of the point values, each of the roles has. For example:

- **Scenario 1** – A Game with two **Werewolves**, one **Guard**, and one **Seer** is **valid** provides the Game with a point sum of $-2$, meaning the **Werewolves** will have a slight advantage;

- **Scenario 2** – Alternatively, a Game with two **Werewolves**, two **Guards**, and one **Seer** is also **valid**, with a point sum of $+1$, with an even slighter advantage for the **Villagers**.

As there is no way to reach 0 points in this scenario, the Game will pick the option with the smallest advantage to either side, in this example – with **Scenario 2**.

With a point-based system, the role distribution can vary greatly, as all roles that do not have a 0 value, provide some form of effect that helps either **bad** or **good** sides. Whilst the Villager role is part of the **good** side, it does not provide any ability that aides either side.
4.3. Game

<table>
<thead>
<tr>
<th>Role</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Villager</td>
<td>0</td>
</tr>
<tr>
<td>Guard</td>
<td>+3</td>
</tr>
<tr>
<td>Seer</td>
<td>+7</td>
</tr>
<tr>
<td>Werewolf</td>
<td>-6</td>
</tr>
</tbody>
</table>

Table 4.3: Base Werewolf role-point values

4.3.2 Actions

All of the possible actions a Player can do are encapsulated as an action. This allows for the handling messages, votes, attacks etc. as a common entity – an action. This results in code simplicity and ease of expansion. The game specific actions currently supported by the framework are:

- **Vote** – Vote for a player to be executed during the day time.
- **Talk** – Send a preset communication message to everyone.
- **Whisper** – Send a preset communication message only to your teammates. Currently accessible only by Werewolves, as they are the only role that concretely know who their teammates are.
- **Attack** – Attack a player during the night as a Werewolf.
- **Divine** – Reveal whether or not a specific player is a Werewolf to yourself as a Seer, during the night time.
- **Guard** – Protect a player during the night time from an attacking Werewolf.
- **Wait** – An empty action in either day or night time which skips the user’s turn.

A limitation the framework currently has is that each role will have at most one special action per time of day phase in the game. By special action, the author identifies actions which are required for the game to evaluate the next game state. The special actions in this iteration of Werewolf are **Attack**, **Divine**, **Guard**, and **Vote** (see Table 4.4).

However, the action **Wait** is an exception, as it is treated as the Empty action, meaning player \( X \) targets no player \( Y \) for the current time of day. To elaborate, a **Wait** action is registered if the player targets no one with an **Attack**, **Divine**, **Guard**, or **Vote** action.

<table>
<thead>
<tr>
<th>Role</th>
<th>Vote</th>
<th>Talk</th>
<th>Wait</th>
<th>Guard</th>
<th>Divine</th>
<th>Whisper</th>
<th>Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Villager</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Guard</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Seer</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Werewolf</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.4: Base Werewolf game roles – possible actions
4.3.3 Communication

RLereWolf’s implementation of Werewolf also supports a range of preset communication messages which players can send within the game. These messages can be broken down to four message types, namely:

- Conflict resolution messages – Agreement or disagreement with a statement or observation made by a different player, about a specific event or scenario, e.g. whether or not a player is some role $X$.

- Asserting messages – Messages that declare whether a player is a particular role. These can be both certain and uncertain, e.g. “I know player is a role” and “I think player is a role” respectively.

- Declarative messages – Used by players to announce what their action or state is, e.g. “I am a role”.

- Special messages – Additional messages that are available to particular roles. These would usually be used as a message between known teammates, e.g. a werewolf can say who they will attack during the night as a Whisper.

The communication protocol proposed by the author aims to improve on the minimalist communication protocol created in AiWolf (Toriumi et al., 2016), which covers 50% of all possible messages used in the tested games, by adding conflict resolution messages. The conflict resolution messages can be used to sway votes into a particular direction if there are uncertainties who to vote for. The ability to agree and disagree with players, openly to everyone, can create a scenario, similar to a chain of trust (see Figure 4.11), where trust can be split into two forms of trust:

- Local trust – An instance trust factor, based on the actions solely in the current game’s context.

- Global trust – A trust factor which takes into account all previous interactions with some player.

In this chain of trust structure, you could players that can get swayed to do a specific action, if someone they trust has also done it. Taking a look at figure 4.11 the author proposes a scenario in which Cindy is a seer and knows that Diego is a werewolf. Ben trusts Cindy a lot and is consequently swayed to do the same action as Cindy, because they were uncertain what to do in this round. This chain of trust can proceed to go indefinitely or can break when some player at the end of chain, in this case Alex, is certain about their action, or just does not trust Ben enough to get swayed.
Figure 4.11: Communication – Conflict resolution.

Whilst the framework does not provide with an “out of the box” Agent experiment with swaying and conflict resolution messages, the author provides the reader with the framework, client and data to implement the aforementioned scenario.

The provided by the author communication protocol is broken down into multiple DTOs which are created by the server, either by prompted by a client request to send a message, or an intrinsic message that is created by the server, e.g. sending a message of the game state to all players. Each MessageDto contains a MessageMetaDataDto, which is information about the message itself and is used when determining who the target for some message is. This replaces the need for Natural Language Processing (NLP), as all of the messages are strictly defined presets based on three variables:

- The source – The entity that created the message. The entity can be a player or the server in the current iteration of RLereWolf.

- The target – The target player who the message is directed to.

- The message type – This is the identifier of the message preset. It represents the key value if we represent the communication presets as a key-value pair.

Moreover, the implemented by the author communication protocol is easily expandable by adding the preset in the CommunicationPresetConstants file and classifying it. That is, describing which role can use it and when it can be used, which is done in the TalkMessageUtility. At the current iteration of RLereWolf, the supported classification filters are based on the time of day and the player’s role.
4.3.4 Players

A Player in the framework is regarded as either Agent or human player and is a common modelling object between the actions of an agent and a human player. Players are the entities that play the games hosted on RLereWolf’s servers and are tightly coupled with the game implementation. A Player can is an entity that is used exclusive within the context of a Werewolf game and cannot exist when not in used by some game. Furthermore, a Player can only exist within the context of a single game. Generally, in the current implementation of Werewolf within the framework, a player is characterised by the following features:

- Identifier – The UUID that is either generated or assigned by the Client. Players can be instantiated without a reference to an identifier, in which case they just generate a new UUID. The identifier is often used in the framework as a reference of individuals, particularly when handled by a Client, as the identifier does not over-expose information about some Player, which might reveal their status within the game’s context, e.g. their role.

- Name – All Players must have a name to be assigned one. Clients will provide the name they assign within the Client instance itself, whereas Agents will have a randomly generated name.

- Role – The role a Player has within a game. This is defaulted to Python’s null value – None, until the game starts and the role distribution process has finished.

- IsReady – Indicates whether or not the Player is ready to play the game. By default, all Agents which are used in the server hosted games are always ready, as to remove the possibility of getting a game lobby getting stuck in a state where it cannot ever start, as the Agents cannot toggle their readiness state.

- IsAlive – Indicates whether a player has been killed or executed within the game.

In RLereWolf, the author has separated Players into two groups (see Figure 4.12), namely:

- Agent – A stochastic, rule-based or artificial intelligence (AI) who is not directly controlled by a human player. However, the actions of other players may influence the agent’s overall behaviour. Agents can be the following types (see Figure 4.12):
  - Dummy – An agent which does stochastic actions with minor action validation, in order to make sure valid game moves are done within the current game state, without doing any logical decision making (for further information see Section 5.1 Dummy Agent).
  - Rule-based – An agent which does deterministic actions based on a trust factor it keeps for each other Player in the current game. (for further information see Section 5.2 Rule-based Agent).
  - Trainable – An agent which learns from playing multiple games of Werewolf and bases their actions on “previous experiences” (for further information see Section 5.3 Trainable Agent). A Trainable Agent has no inherent knowledge of the game, and as such will attempt all possible actions within the game space until they have learned the rules of the game by considering the potential reward.
• Human – A human player that is running an instance of the RLereWolf client. This can be used interchangeably with Client as their relationship is one-to-one (see Figure 4.12).

![Figure 4.12: Players architecture – The distinction between humans and Agents.](image)

Whilst having a common derivative interface for both Agents and Clients can lead to shotgun surgery (Li and Shatnawi [2007], Olbrich et al. [2010]), that is – the change or addition of features of child classes being a direct result by some change on a parent class, its effect is drastically mitigated by enforcing abstract classes. Shotgun surgery is a very likely scenario due to Python’s lack of type safety and lack of strictly checked abstract class usages, this is because Python is an interpreted language and those mistakes can only be recognised during the framework’s run-time.

The enforcement of abstract classes is done through having the derivative class have no functionality whatsoever, but only serve as an interface which throws exceptions whenever the base class is called. Alternatively, users can use Python’s Abstract Base Class[7](ABC). However, its behaviour and interface is variable on the Python version at use and could have massive implementation implications. As a result, the author has decided to use self-governed abstract classes by, as previously mentioned, throwing exceptions when the base functionality is called. Consequently, this enforces any children classes to override the base methods if they are to be used at any given point.

Having a shared Player derivative for both Agents and Clients allows for the framework’s game implementation to easily be expanded with additional features which may or may not be included in their children classes. This also inherently applies to any of the AgentPlayer derivatives – where more Agents can be added, or their functionality can be easily modified to target all or just some of the Agents (for further information see Chapter 5 Agents).

4.4 Environment

In RLereWolf’s architecture, the Environment is the “training ground” for some Agent. The environment is a wrapper that can optionally be put around a game. This keeps the Werewolf implementation lightweight and implies that not all games will serve a game that trains Agents (see top

---

[Abstract Base Class –](https://docs.python.org/3/library/abc)
of Figure 4.1. The environment uses RLlib (RLlib, 2020), which uses OpenAI Gym (Gym, 2016) and is an effective wrapper, encapsulating OpenAI Gym. The author’s decision to use RLlib is due its integrated support for the creation of a multi-agent environment, as opposed to plainly using OpenAI Gym. The only supported environment types by OpenAI Gym is a single-agent environment. This means that in a game, you can only have one single Agent entity, this could mean that you have 10 Agents in a single game – however, they would all be using the same brain.

Having a multi-agent environment allows for the training of multiple Agents in a single game. Each Agent is responsible for individually taking a decision in each step of the game, which in RLereWolf’s case, a step is a time of day within some game turn, where each turn consists of a day and night time. Having a multi-agent environment also allows for the definition of individual policies, which are the intrinsic rules that define an Agent and its consideration of the various potential rewards for some action. In the current iteration of RLereWolf, the TrainableAgent aims to be the “jack of all trades” within the game’s context – that is, be able to learn all four currently supported roles. An immediate consequences of the author’s decision is that the training process will be longer as a consequence. The alternative approach is to have a role-specific TrainableAgent, e.g. WerewolfTrainableAgent and VillagerTrainableAgent. Although not provided in the default configuration of RLereWolf, creating TrainableAgents for each individual role is possible, due to the hierarchical structure used in the framework (see Figure 4.12).

The training environment in the framework is regarded as the TrainableEnvironmentWrapper and it can hold any combination of Agents in it. However, its primary target in the current iteration of RLereWolf is the TrainablePlayer. However, the environment can also be used in multiplayer games which have multiple players in them, as the Client allows for the addition of various Agent types to the game lobby. Moreover, the environment contains the auto-play service which makes sure training never gets stuck by some invalid action, by keeping a close track of the Agents which have attempted an invalid game action. Once an invalid game action is attempted, the auto-play service makes sure to discount their identifier as a Player who needs to make a play in the current turn.

The environment’s functionality heavily relies on the game implementation, as each of the game actions a Player can do, will have a return value which indicates the result of an action. A limitation to the framework’s action request results is that there can only ever be one result. A scenario in which this could be a problem is whenever a werewolf attacks another werewolf, which happens to be dead. RLereWolf’s training environment keeps track of the following action results:

- Wait action – A wait action result is treated individually as the author can see the importance of recording the only passive action, as Agents could get into a state where they have learned that waiting would be a viable strategy. Moreover, the incentive to wait should be different than the incentive to actively vote, attack, guard and divine someone.

- Successful action – A successful action result is given whenever an Agent has decided to do a game action, e.g. vote, attack, guard or divine, and their request has gone through successfully.

- Cannot act during time of day – A result given whenever an Agent with a particular role that cannot act during the current time of the day, attempts to do an action during that time (see
In the current iteration of the game, this could be Villagers attempting an action during the night time or a Guard attempting to guard someone during the night once they have reached their maximum guard limit.

- Dead player targeted – A result given whenever an Agent targets a dead Player. This could be a result of either day or night game actions. However, Seers can target dead Players during the night, as it is a valid game action.

- Werewolf cannibalism – A result given whenever a werewolf targets another werewolf of their attack. However, this will be preceded by the Dead player targeted result, as only one action result can be retrieved in the current iteration of RLereWolf.

- Invalid action – An invalid action result is used whenever an Agent attempts to do an action that their role does not allow them to. This differs from the “cannot act during time of day” result, as an attack and guard options are both possible during the night. However, they are possible to the werewolf and guard, respectively.

Each of the action results are used in the observation space generation and are an important part which determines which reward an Agent will receive. The action results are part of the description of the observation space, that is – the game state and all of its known, by some player, parameters. Each Agent action prompts a reaction from the Environment, by providing the Agent with an observation of the current game state, alongside a reward – which is based off of their action and its consequences. This is also known as the “agent-environment loop” (Gym, 2016; Kong and Mettler, 2013), which is also known as the Markov Decision Process (MDP) (Littman, 1994). The loop can be seen in Figure 4.13 in which the Environments receives some action ($A_x$) from an Agent, to which the Environment replies with an observation space ($S_{x+1}$) and a reward ($R_{x+1}$).

![Figure 4.13: Agent-environment loop](image)

### 4.4.1 Observations

The observation space for each Agent is calculated on each Agent step. Due to the nature of the game and being able to work only with incomplete game state knowledge, the observation space is constructed based on information the Agent knows at the current point of some Player acting. Whenever an action is made in an Environment-wrapped game, the observation space is generated and distributed to each Agent. Without the complete game state information it is impossible to
reach pure Nash equilibrium \cite{Fabrikant2004, Gottlob2005} in Werewolf as it is a social deduction, finite Bayesian game \cite{Cheon2008}.

However, the aim of the observations is to provide Agents with enough information to reach Bayesian Nash equilibrium \cite{Christodoulou2008} – that is, the state in which a Player’s action is based around the highest probable payoff, given their knowledge and beliefs of the game and player states. To elaborate, this means that every Player’s decision for their current turn will not be based on some predetermined strategy, but would instead be based on all available to them information at the current state of the game.

The knowledge of the game state is included in the observation, whereas the player state observations are based entirely off of the Agent in the current iteration of RLereWolf (see Section 5.3 for more information). The game state information given to each Agent in the current iteration of RLereWolf’s Environment consists of:

- Role – The current player role, this is provided in the observation as it can be changed by the game whenever a game is restarted.
- Players – The list of players currently in the game. The information for the Players is minimal, as it only contains the identifiers and the names of the Players.
- Votes – The visible by everyone actions for the current time of day and turn in Werewolf. An example of an invisible action is the seer’s divine ability.
- Messages – The messages created by other Players or the game itself, which are descriptions of the game state accompanied by message metadata, in order to aide the developer in recognising what the message is and who it is targeted to.

RLereWolf’s observation size for some turn is, at the moment, uncapped. As a result there is the possibility of having an infinitely long observation for some turn. What this means is that the framework will get stuck in a state where the Players are generating observation data indefinitely. A solution to this problem would be to limit the amount of possible messages an Agent can generate within a given turn. Consequently, this would result in a limited observation space which removes the potential for infinite loops within the framework. The reason for allowing infinite observation spaces within RLereWolf is because the author aimed at having the Agents define their own game logic – resulting in more freedom when defining the training environment as some Agents could be more “chatty” than others.

Furthermore, it mimics the possible behaviour in a real Werewolf game, where some Players’ behaviour is variable, all whilst obeying the core game rule. In the current iteration of RLereWolf, the observation space not represented in exclusively discrete values, this is a by-product of leaving the Agents to process the game state provided to them.

### 4.4.2 Rewards

Alongside the observation space, the Agents receive a reward at the end of each turn. The rewards are based on various action results, game state conditions and general game motion, that is – the advancement of the game in its turn value (see Table 4.5). The values of the rewards, represent whether or not a result is deemed favourable or unfavourable (see Section 4.4 for more information on results).
4.4. Environment

<table>
<thead>
<tr>
<th>Result</th>
<th>Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day Passed</td>
<td>-1.0</td>
</tr>
<tr>
<td>Vote</td>
<td>+1.0</td>
</tr>
<tr>
<td>Wait</td>
<td>+0.5</td>
</tr>
<tr>
<td>Incorrect Vote Multiplier</td>
<td>-0.5</td>
</tr>
<tr>
<td>Death</td>
<td>-5.0</td>
</tr>
<tr>
<td>Victory</td>
<td>+20.0</td>
</tr>
<tr>
<td>Lost</td>
<td>-20.0</td>
</tr>
<tr>
<td>Werewolf Cannibalism</td>
<td>-7.5</td>
</tr>
<tr>
<td>Dead Player Targeted</td>
<td>-35.0</td>
</tr>
<tr>
<td>Incorrect Action</td>
<td>-35.0</td>
</tr>
</tbody>
</table>

Table 4.5: RLereWolf Environment rewards

The default RLereWolf rewards are loosely based on AiWolf (Toriumi et al., 2016) which the author has expanded on, due to the higher amount of possible action results. The design for the rewards take into account a possible Agent who does not have any understanding of the intrinsic game rules whatsoever and is able to “break” the game rules by doing some invalid or malicious action. The author describes an invalid action as the action which the current Player’s state is unable to do, e.g. A Villager attacking another Player at day time; or A Guard divining another Player at night. A malicious action can be seen as an action in which an Agent actively acts against their own role’s nature, e.g. a Werewolf attacking another Werewolf. The result-reward justification is as follows:

- **Day Passed** – At the end of each turn (day & night), Agents gets a −1.0 penalty. The reasoning behind the penalty is so that Agents can get trained to finish the game as quickly as possible.

- **Vote** – Each time an Agent votes (this includes attacking, guarding, divining), and that Player is the most voted one, the Agent gets a +1.0 reward. In the case that the vote the Agent has cast is on the non-executed Player, an Incorrect Vote Multiplier is applied. This logic will be based on internal Agent design estimations (see Section 5.3 for more information), as this might not have to apply to a Werewolf, as they would inherently be targeting the opposite player group.

- **Wait** – There is a need for an explicit reward for waiting, as it is a “delaying” tactic, which accompanied with the Day Passed reward should lead to either a neutral or negative reward value. In the current iteration of RLereWolf, the author has chosen a reward of +0.5, as this pushes Agents into making valid and non-malicious actions on their turn. Whilst usually actively voting will lead to a higher overall reward for the Agent, sometimes a wait action might be lead to the higher overall reward. This is because the voted Player might not be the executed one and consequently lead to a higher penalty, as given by the Incorrect Vote Multiplier.
• Incorrect Vote Multiplier – At the end of each time of day, the Agents’ rewards are applied a $-0.5$ multiplier, based on how far from the most voted Player the Agents were. This reward acts as a meta-communication protocol, which will eventually align Agents to unanimously target the same Players. The multiplier gets added on to the final reward by taking into account the order of the most-voted Players in some time of day, e.g. Agents who have voted for the second most-voted Player will receive an additional reward of $1 \times (-0.5)$, Players who vote for the third most-voted Player will receive an additional reward of $2 \times (-0.5)$. Generally speaking, the act of voting will lead to the following reward ($R$) for some turn and time of day ($R_{Turn\_Time\_of\_Day}$):

$$R_{Turn\_Time\_of\_Day} = R_{Vote} + (Index_{Voted\_Players} \times R_{Incorrect\_Vote\_Multiplier})$$

Equation 4.1: Reward calculation for a turn and time of day whenever an Agent votes

• Death – The death of an Agent, although sometimes can be seen as a viable strategy, is generally regarded as a negative result which should be avoided. However, the death result is not something an Agent will directly have control over. As a result, the author has determined that a penalty of $-5.0$ will be sufficient in the standard use case, where suicidal strategies are to be generally avoided.

• Victory/Lost – Both the Victory and Lost rewards are applied to all Agents within a game, independently of their current Player state – dead or alive. The reward/penalty is $\pm 20.0$, depending weather a positive or negative outcome is achieved by the Players role archetype. The reward is relatively high as it keeps Agents focused on winning.

• Werewolf Cannibalism – In the current version of the framework, the only recognised malicious action is the act of a Werewolf attacking another Werewolf. Whilst this could be a viable strategy to confuse other players and cast doubt on self-proclaimed Seers, the author has set the penalty for cannibalism as $-7.5$, as to reduce cannibalism actions, due to their niche viability.

• Dead Player Targeted – Targeting dead players is considered an invalid action by most roles in Werewolf. In the base game of Werewolf, the only role that can target dead players is the Seer and this penalty does not apply to them. However, the other currently supported roles (Villager, Werewolf, and Guard) should not be able to target dead players and are consequently penalised with $-35.0$ points. The need for such a high penalty is because all game breaking actions must be put into the highest priority, as having a higher reward for a Victory will prompt Agents to “cheat” in the game in order to achieve a Victory.

• Incorrect Action – The attempt of acting as other roles than the currently assigned one, or doing an action that is not possible in the current time of day of the turn, will prompt an Incorrect Action penalty of $-35.0$ points. The reasoning is the same as the Dead Player Targeted reward – not allowing Agents to “cheat” to achieve a Victory.

The rewards in RLereWolf are configurable within a single file. However, expanding the possible rewards will prompt the creation of more action-results which the game implementation
will need to recognise. Once the game implementation recognised the result of a specific Player action and replies with the newly implemented result, then the Environment will be able to use the result-reward when going over the step for some Agent.

4.4.3 Auto-play

The Environment in RLereWolf is an optional wrapper that can be used for a game instance (see Game Z in Figure 4.1). As the framework can support games with and without human Players, the need for an automated playing service arose. The auto-play service aims at simulating games which consist only of Agents and ensures the validity of the game state at any given point of time. The auto-play service can be present both in server hosted games and in locally instantiated games, whose current target is games dedicated for training Agents.

This service is not manageable by Clients within the GUI, but rather by the server administrator’s preference, reflected in the server instance configuration. This design choice was due to the fact that a malicious Client can get the server into an infinite training loop which can only be handler by a server restart or a computational resource threshold on each game. Consequently, the author has decided to keep the auto-play configuration, primarily targeted for offline training games as the server hosted games will use the training data generated from the offline ones.

4.4.4 Metrics

The Environment keeps a log of all of its games under a csv file format and keeps a record of the following metrics within each game:

- **Number of players** – Shows the number of Players in the game. These Players can be either humans or Agents.

- **Dead Players voted** – The number of times a Dead Player Targeted action-result has been achieved during the day.

- **Dead Players attacked** – The number of times a Dead Player Targeted action-result has been achieved during the night.

- **Teammate attacked** – The number of times a Werewolf Cannibalism action-result has been achieved.

- **Incorrect action** – The number of times an Incorrect Action action-result has been achieved.

- **Werewolf wins** – Used as an indicator whether the game was won by the Werewolf role archetype. This is a boolean value, displayed as a number (0 or 1).

- **Villager wins** – Used as an indicator whether the game was won by the Villager role archetype – Villager, Guard, and Seer in the current iteration. This is a boolean value, displayed as a number (0 or 1).

- **Total turns** – The number of time of day phases that occurred in the game (day & night).

- **Total days** – The total number of days that have passed in the game. Combined with the total turns metric, it is possible to determine which time of day the game has ended, as an odd total turns value implies that the game has ended during the day, where as an even total turns value implies that the game has ended during the night.
4.4. Environment

- **Total games** – Used as an indicator whether a game has successfully finished, or something has gone wrong within the game, which has prompted the auto-play service to continue. This is a boolean value, displayed as a number (0 or 1).

- **Game time** – How much overall time the Game has lasted. This metric is important when designing Agents whose primary target is to have shorter Games, preferably classified as wins.

The metrics are stored within the root Werewolf project folder (see Figure 4.2). Each of the metrics get stored within the same “Statistics.csv” file. An issue with this design is that it would be hard to differentiate between the metrics across different games which are concurrently running games with the Environment wrapper on either a server, or locally. This is a result of the time constraints the author was limited to, and a suggested architecture would be one similar to logging (see Section 4.1.4 and Figure 4.6), in which each game with an Environment wrapper will create a csv file within a “Metrics” folder which will be named after the game name and its identifier.

---

**Chapter 4 in a Nutshell**

- Server can accept only legitimate packets sent from the Client.

- Server keeps track of all traffic and logs it.

- Client has a graphical user interface which can easily be extended with the help of the GUI development pipeline.

- Clients are uniquely identified.

- Werewolf game implementation supports four roles – Villager, Guard, Seer, and Werewolf.

- Each role has special “abilities” that distinguish them.

- The players are able to communicate with a “closed” communication protocol, that is – able to communicate with predefined messages.

- The Environment is a wrapper over a Game instance.

- The built-in Environment provides rewards to the players.

- The built-in Environment generates analytic and training data.

- The built-in Environment ensures that a Game between Agents is never “stuck”.
Chapter 5

Agents

Agents are the artificial Players which humans can play Werewolf with. In this chapter, the author will go over the built-in Agents within RLereWolf and what purpose they serve within the context of the framework. Users of the framework can develop additional Agents, independently of their decision making capabilities – stochastic, rule-based or machine learning. Whilst the framework can support various machine learning methods, the built-in example provided by the author is using reinforcement learning (see Section 5.3 for more information).

5.1 Dummy Agent

The Dummy Agent is the first built-in Agent which was introduced into the framework and primarily serves as a means to test the framework and the supported Game functionality. The Dummy Agent does stochastic actions during both day and night time, which are valid within the game rules. To put the Dummy Agent into perspective:

- If a Dummy Agent is a Villager, they will only randomly vote for other Players (excluding themselves) during the day time.
- If a Dummy Agent is a Werewolf, they will randomly vote for other, non-Werewolf, Players during the day time and attack a random non-Werewolf Player during the night time.

Dummy Agents are not swayed by any of the communications and they themselves are not capable of sending Messages or Whispers. The Dummy Agent serves as a baseline for a Player doing random actions within the games’ rules and its behaviour serves as a starting point for future Agents’ performance.

5.2 Rule-based Agent

The Rule-based Agent in RLereWolf has a simple understanding of trust (Marcus and Davis, 2019) it keeps towards all the players they play against. Each Rule-based Agent keeps a record of global trust and local trust which reflects how trustworthy a Player is throughout all of the games played with them and how trustworthy they are in the current game instance. Moreover, each Rule-based Agent has an “honesty factor” which is a value from −1 to 1, which provides the Agent with a deterministic approach as to what communication the Agent will convey to Players, if any.

In the current iteration of RLereWolf the Rule-based Agent is not fully implemented and behaves similarly to the Dummy Agent, such that all of the actions it takes are stochastic with minor validity and “common sense” checks. However, the Rule-based Agent is not purely stochastic
5.3. **Trainable Agent**

and does select the least trustworthy Player to vote for in the current game. This trust factor, however, is not fully implemented and is expected to behave similarly to the purely stochastic approach. The validity of an action is determined by the game implementation and means that the action “makes sense” within the context of the game, e.g. A Guard cannot divine. As such, in the current implementation of the framework, the Rule-based Agent will perform similarly to the Dummy Agent when observing the metrics provided by RLereWolf (see Section 4.4.4 for more information).

However, the aim of the Rule-based Agent is to observe communications made by other Players in the game and get swayed or potentially sway other players (see Figure 4.11), based on their observed trust factor, which gets recalculated on every game event, in which the Agent has made an assumption on.

The global trust each Rule-based Agent keeps gets recalculated at the end of every game and is a rolling average of the local trust for all games played with that Player. Due to the limited time of the project, the author could not fully implement this feature. However, the author suggests a similar approach to Q-learning (Watkins and Dayan, 1992; Hasselt, 2010) where a discount factor is used for the trust factor. The role of the discount factor is such that the more recent local trust values hold more “weight” than older local trust values when determining the global trust for some other Player. Semantically this means that Players’ behaviour can change over time, and Rule-based Agents should be able to adapt to it.

The Rule-based Agents currently do not persist the trust factor they observe about other Players, and are instead stored in the game host machine’s memory. However, this can be easily implemented as each Player in the framework has a corresponding UUID (see Section 4.12 for more information) which can be used to reference Players across multiple Game instances and game lobbies. The persistence model will be similar to the logging architecture (see Section 4.1.4 and Figure 4.6 for more information), in which the host machine will have a folder of files named under the following format: “Agent Name – Agent Identifier”.

**5.3 Trainable Agent**

RLereWolf comes with a built-in reinforcement learning Agent alongside its multi-agent environment (see Section 4.4 for more information). The Trainable Agent, unlike the Dummy Agent and the Rule-based Agent, has no knowledge of the intrinsic game rules and is capable of making any move within the context of the game, be it valid or invalid. However, the Trainable Agent can learn strategies based on rewards (see Section 4.5 for more information on Rewards).

The Quality Learning (Q-learning) algorithm is used for the Trainable Agent as the Werewolf game has a finite action space and will eventually lead to an “optimized” agent for some set of Players and their corresponding behaviours. In the current iteration of RLereWolf, the Trainable Agent is not fully implemented and can consequently not exhibit any emergent behaviours, as the Trainable Agent does not formally “learn” in the current iteration. However, the Trainable Agent should have a Q-table, which represents a map of some unique tuple consisting of the game State \((S_T)\) and the Action \((A_T)\) which leads to some Reward \((R_T)\) at some turn/time \((T)\) within the Game’s context (see Equation 5.1), which can be as the following function, for some time \(T\):

\[ R_T = Q(S_T, A_T) \]
5.3. Trainable Agent

\[ Q : S \times A \rightarrow \mathbb{R} \]

**Equation 5.1: Q-learning function**

To elaborate that, the goal of a **Trainable Agent** is to do an **Action** for the current game **State**, such that the highest expected **Reward** “route” is picked, as the **Agent** will try to score the highest **Reward** value it can, for some game instance (see Section 4.3 for more information on **Rewards**). The **Q-table** is the effective data a **Trainable Agent** will acquire after playing a sufficient, finite amount of Werewolf games. The “learned” data will be highly dependent on the game implementation, game rules and **Player** role distribution mechanics.

Consequently, the **Trainable Agents’** perception of the potential **Reward** for some **Action** and game **State** will change over time, which can be further supplemented by a **learning rate** (\( \alpha \)) and a **discount factor** (\( \gamma \)) (see Equation 5.2).

\[
Q_{\text{new}}(S_T, A_T) = Q_{\text{old}}(S_T, A_T) + \alpha \sum \text{learning rate} \times \\
\left( R_T + \gamma \sum \text{discount factor} \times \max \text{optimal future value for all actions} \right) - Q_{\text{old}}(S_T, A_T)
\]

**Equation 5.2: Q-learning algorithm – “weighted” Q-value**

The **Trainable Agent** is able to play as any role which is implemented in the Werewolf game, as a result the training time for the built-in **Trainable Agent** will be substantially longer that of a role-specific **Agent**. That is – an **Agent** model designed to play as a single role, as opposed to the full set of supported roles: Villager, Guard, Seer, and Werewolf.

Moreover, as the author has briefly mentioned, the **Trainable Agent** has no understanding of the intrinsic **Game** rules and will, consequently, make **invalid** actions at the beginning of the training process. Over time, the **Trainable Agent** will correlate **invalid** actions with a very high negative reward and will get discouraged from attempting the specific \( Q(S_T, A_T) \) tuple which will consistently provide it with a negative reward. Furthermore, the **Trainable Agent** has no understanding of trust and does not keep **honesty values** for the **Players** they have played with, instead all the observable behaviours, exhibited from the **Trainable Agent**, are learned with **Q-learning**.

In the current iteration of RLereWolf, the “weighted” **Q-learning** (see Equation 5.2) is not fully implemented as the author’s allocated time slot for the development of the built-in **Agents** was not sufficient enough. Consequently, each **Agent** has partial completion, but not fully implemented for the initial iteration of the framework. This will be observed in Chapter 6 **Empirical Evaluation** where the author will go over a set of tests with the built-in **Agents** and discuss their results.

**Trainable Agents** can be “used” both in **Agent-only** games, which are designed for simulating games and training, as well as games with human **Players** in them (see Figure 4.1). A limitation to **Trainable Agents** in the current iteration of RLereWolf is that they cannot be trained in server hosted games as it would require the usage of the auto-play service and **Environment** (see...
Section 4.4 for more information), which cannot be dynamically added to a Game instance.

Chapter 5 in a Nutshell

- The RLereWolf framework provides three built-in Agents.
- The Dummy Agent does random actions which are considered valid by the game and its rules.
- The Rule-based Agents employs honesty and trust factors which it uses to determine who to vote/attack next. A fall-back is the stochastic approach used in Dummy Agents. All actions considered valid by the game and its rules.
- The Trainable Agent uses Q-learning in order to learn on what the best action for some game state is.
Chapter 6

Empirical Evaluation

In this chapter the author will evaluate the built-in Agents’ performance with the help of the provided by the Environment metrics. The experiments the author will carry out are multiple “runs” of the Werewolf Game with an Environment wrapper around them. Once the set of experiments are complete, the author will discuss the results with the aide of graphs and tables.

6.1 Experiment Design

The experiments the author will carry over a set of Games, each consisting entirely of a single type of Agent (Dummy, Rule-based or Trainable). Each of the Game “lobbies” will be wrapped with the Environment, which not only “trains” the Trainable Agents, but it also provides the metrics, which the author will use for the experiment analysis.

The games will be a populated with a variable range of Players for each set of “runs”. For the purposes of the experiment, each “run” will consist of 1000 games. And the Agent count to be observed at minimum, 25%, 50%, 75%, maximum and AiWolf equivalent capacities, namely:

- **5 Agents** (minimum capacity) – The minimum amount of Players required in a Game of the base version of Werewolf, as stated in the rule book [BGG 2016].

- **15 Agents** (roughly 25% capacity) – A small-sized party, the same Player count used in the AiWolf conferences [Toriumi et al. 2016].

- **20 Agents** (AiWolf capacity) – The one quarter point between the minimum and maximum amount of Players that can play in a Game of Werewolf.

- **35 Agents** (roughly 50% capacity) – The mid-point between the minimum and maximum amount of Players that can be in a Game of Werewolf.

- **55 Agents** (roughly 75% capacity) – The three quarter point between the minimum and maximum amount of Players that can play in a game of Werewolf.

- **75 Agents** (maximum capacity) – The maximum amount of Players allowed in a Game of Werewolf, as stated in the rule book [BGG 2016].

The amount of Players was, such that, a wide range of possible capacities is tested and analysed, alongside the Player capacity tested in AiWolf – 15 Agents.

As the current iteration of RLereWolf is using a ratio-based role distribution system, the roles for some number of Players in a Game will be consistent. The role counts for the currently
implemented ratios (see Table 4.2) can be seen in Table 6.1. The hardware the author used for the experiments is the same as the development machine (see Table 3.1).

<table>
<thead>
<tr>
<th>Player count</th>
<th>Villagers</th>
<th>Guards</th>
<th>Seers</th>
<th>Werewolves</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>35</td>
<td>25</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>55</td>
<td>38</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>75</td>
<td>51</td>
<td>7</td>
<td>5</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.1: RLereWolf role distributions for experimented Player counts

The experiments were carried out on the author’s development machine (see Section 3.4 for more information on the hardware used) and took a total of 16 hours to complete. The complete recorded data from the $3 \times 6 \times 1000$ ($AgentCount \times PlayerQuantityVariants \times RunGameCount$) set of games, can be found on the project’s public repository.

6.2 Results Discussion

From the metrics provided to us by the Environment we can extract the general win-rate and “speed” performance of each Agent (see Table 6.1 and Figure 6.2). However, these results abstract from the validity of the Agents’ actions in the Games which were part of the experiment. The validity of the Agent actions is measured with the amount of times an Dead Agent Voted, Dead Agent Attacked, Teammate Attacked, and Incorrect Action action-results (see Section 4.4.2 for more information on action-results and rewards) are given to the Agents throughout an entire experiment “run” (see Table 6.2).

Figure 6.1: Experiment metrics graph – Win-rate and Game speed

6.2. Results Discussion

From the results (see Table 6.2), it is visible that the Agents have a different win-rate as Werewolves, consequently as Villagers as well (see Figure 6.1). This is attributed to the different behaviours all three built-in Agents exhibit. However, if we take a look at the Dummy and Rule-based Agents’ win-rate as a Werewolf (see Figure 6.1), it is visible that both Agents exhibit similar behaviour, such that with the rise of Player count within the Game, their performance as Werewolves increases, even reaching 100% win-rate as Werewolves. The author believes that this

<table>
<thead>
<tr>
<th>AT</th>
<th>P#</th>
<th>EWR</th>
<th>GWR</th>
<th>AGL</th>
<th>AGTL</th>
<th>AGTC</th>
<th>DAV</th>
<th>DAA</th>
<th>TA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>5</td>
<td>56.2</td>
<td>43.8</td>
<td>0.038</td>
<td>0.014</td>
<td>2.74</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.07</td>
</tr>
<tr>
<td>D</td>
<td>15</td>
<td>62.8</td>
<td>37.2</td>
<td>0.248</td>
<td>0.022</td>
<td>11.52</td>
<td>0.00</td>
<td>0.07</td>
<td>0.00</td>
<td>2.23</td>
</tr>
<tr>
<td>D</td>
<td>20</td>
<td>84.4</td>
<td>15.6</td>
<td>0.511</td>
<td>0.033</td>
<td>15.71</td>
<td>0.00</td>
<td>0.13</td>
<td>0.00</td>
<td>6.38</td>
</tr>
<tr>
<td>D</td>
<td>35</td>
<td>93.1</td>
<td>6.9</td>
<td>1.245</td>
<td>0.044</td>
<td>28.64</td>
<td>0.00</td>
<td>0.17</td>
<td>0.00</td>
<td>18.55</td>
</tr>
<tr>
<td>D</td>
<td>55</td>
<td>99.1</td>
<td>0.9</td>
<td>3.292</td>
<td>0.079</td>
<td>41.90</td>
<td>0.00</td>
<td>0.16</td>
<td>0.00</td>
<td>50.93</td>
</tr>
<tr>
<td>D</td>
<td>75</td>
<td>100.0</td>
<td>0.0</td>
<td>7.741</td>
<td>0.136</td>
<td>57.14</td>
<td>0.00</td>
<td>0.16</td>
<td>0.00</td>
<td>99.95</td>
</tr>
</tbody>
</table>

| RB | 5 | 47.9 | 52.1 | 0.037| 0.013| 2.72 | 0.00| 0.00| 0.00| 0.07|
| RB | 15| 54.6 | 45.4 | 0.179| 0.016| 11.55| 0.00| 0.07| 0.00| 2.21|
| RB | 20| 77.1 | 22.9 | 0.458| 0.028| 16.24| 0.00| 0.16| 0.00| 6.45|
| RB | 35| 76.3 | 23.7 | 1.573| 0.051| 30.64| 0.00| 0.19| 0.00| 19.20|
| RB | 55| 89.1 | 10.9 | 4.570| 0.095| 48.22| 0.00| 0.23| 0.00| 54.93|
| RB | 75| 92.8 | 7.2  | 8.777| 0.132| 66.65| 0.00| 0.24| 0.00| 107.50|

| T  | 5 | 40.2 | 59.8 | 0.045| 0.012| 3.78 | 0.63| 0.38| 0.15| 2.07|
| T  | 15| 29.1 | 70.9 | 0.292| 0.017| 17.27| 11.71| 4.50| 0.56| 48.10|
| T  | 20| 28.0 | 72.0 | 0.601| 0.024| 25.13| 22.91| 8.25| 1.11| 89.68|
| T  | 35| 26.9 | 73.1 | 1.627| 0.033| 48.66| 73.10| 20.62| 2.68| 282.80|
| T  | 55| 32.4 | 67.6 | 3.525| 0.043| 82.28| 73.10| 20.62| 2.68| 282.80|
| T  | 75| 32.2 | 67.8 | 6.560| 0.058| 114.92| 319.30| 70.72| 13.10| 1224.63|

Table 6.2: Experiment win-rates, average Game length, and average invalid actions for a “run” of 1000 Games

AT = Agent type; D = Dummy Agent; RB = Rule-based Agent; T = Trainable Agent; P# = Player count; EWR = Evil (Werewolf) win-rate percentage; GWR = Good (Villager, Guard, and Seer) win-rate percentage; AGL = Average Game length (seconds); AGTL = Average Game turn length (seconds); AGTC = Average Game turn count; DAV = Average Dead Agents voted per Game; DAA = Average Dead Agents attacked per Game; TA = Average Teammates attacked per Game; IA = Average Incorrect actions per Game

<table>
<thead>
<tr>
<th>Player count</th>
<th>Villagers</th>
<th>Guards</th>
<th>Seers</th>
<th>Werewolves</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>+4</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>-2</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>-5</td>
</tr>
<tr>
<td>35</td>
<td>25</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>-7</td>
</tr>
<tr>
<td>55</td>
<td>38</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>-12</td>
</tr>
<tr>
<td>75</td>
<td>51</td>
<td>7</td>
<td>5</td>
<td>12</td>
<td>-16</td>
</tr>
</tbody>
</table>

Table 6.3: Points for the current ratio-based role-distribution counts
is an inherent flaw of using a ratio-based distribution approach as the intrinsic role-distribution is unbalanced in the current iteration of the Game (see Tables 4.3, 6.1, and 6.3).

The upward trend of having Werewolves winning as the Agent count increase, can also be seen in Table 6.2 which shows the inverted points advantage for both factions, that is – the points each role provides is multiplied with by $-1$. However, examining the Werewolf win-rate of the Trainable Player shows that the opposite trend occurs – with the increase of the Werewolf advantage, the Villagers’ win-rate rises, albeit not as drastically as the change occurring for the Dummy and Rule-based Agents. This is likely a result of having Trainable Players being able to Wait, as opposed to Dummy Agents and Rule-based Agents – who must always act.

![FACTION POINTS ADVANTAGE (INVERTED)](image)

Figure 6.2: Experiment metrics graph – Sum of Points for existing role distribution (inverted). The bad and good faction have inverted points, that is – their advantage is multiplied by $-1$.

It must be noted that the Trainable Agent iteration used in the experiments has no pre-existing training data as it is not complete in this iteration. Consequently, a lot of its Game actions will be invalid (see Figure 6.4), which will in hand increase the experiment time. This can be seen in all of the experiment “runs” where the Trainable Agent took from 35% to 100% more turns for a single Game on average (see Table 6.2 column AGTC).

The Dummy and Rule-based Agents have some invalid actions recorded, which are likely a result of faulty action validation checks as the DAA and IA values for both are roughly the same with minor differences which are attributed to the stochastic, but mostly valid, nature of the Agents’ actions. Moreover, both Agents share the same action validation logic, which further justifies their validity performance.

The Trainable Agent “runs” have a higher invalid actions count, due to them having no inherent action validation whatsoever.

However, observing the average Game length in time, we can see that the timings are relatively consistent (see Figure 6.3). The Rule-based Agent took the most time for a Game, which
was expected by the author, as the Rule-based Agent will need to go the trust and honesty values it keeps, in order to make a decision on their action. Surprisingly to the author, the Trainable Agent had the lowest average Game time, although the longest Games in turns, were the Trainable Agent ones (see Table 6.2 – low AGL, high AGTC).

Moreover, it can be seen that the Dummy Agent ends up being slower than the Trainable Agent after the increase of Agents from 55 to 75 (see Figure 6.3), this could be attributed to the extra action validity checks a Dummy Agent must do, in order to complete their turn. This is further supported by the different in average Game turn length (AGTL) in Table 6.2 as the Trainable Agent “runs” have an equal or smaller AGTL value when compared to the Dummy or Rule-Based Agent “runs”. As the time delta between the Games ran by the built-in Agents follows the same curve and any variations they have can be attributed to their decision making process, the author states that the framework’s performance is consistent, independently of the Agent type used. However, these minor variations on an average Game length in seconds can add up to the overall “run” time. To put into perspective, the time difference between the Rule-based and the Trainable Agent “runs” at a Game Player count of 75 is 2,217 seconds – roughly equating to a 37 minutes difference for a “run” of 1000 Games.

Figure 6.3: Experiment metrics graph – Average Game time (seconds)

Whilst RLereWolf does have an improved communication protocol over AiWolf (Toriumi et al., 2016; Nakamura et al., 2017) (see Section 4.3.3 for more information), RLereWolf’s communication protocol has had no use in the carried out experiment, as the functionality of swaying other Agents is not implemented in the framework’s current iteration (see Section 4.3.3 for more information). However, observing at the win-rate for the Villager and Werewolf faction, its average value is closely matched to the final results of the 2nd AiWolf competition (aiW, 2020).
6.2. Results Discussion

Chapter 6 in a Nutshell

- The experiments consist of 18,000 games which are only with non-human players.

- The recorded data shows a fault within the game implementation’s ratio-based role distribution.

- The Dummy and Rule-based Agents have an inherent validation fault which could not be observed during initial testing.

- The Dummy and Rule-based Agents behave differently, although the same performance trajectory is followed.

- The average game time stays roughly equivalent until the 55 player game lobby mark. Implies the stability of the framework and its correlation with the Agent implementation.

- Trainable Agent does not learn from their mistakes and commits the most invalid actions, due to the lack of the action validation, present in both Dummy and Rule-based Agents.

- The ability to Wait as an action has a big detrimental impact on the overall win-rate of the Trainable Agent when playing as a Werewolf.
Chapter 7

Conclusion & Future Work

In this chapter, the author will present the formal conclusion they have made from the experiment results and inherent work with RLereWolf, alongside a detailed list of future work elements, which they author has planned for future iterations of the framework, which aim to improve on the GUI, Agents, overall analysis tools provided by the Environment and accessibility to developers.

7.1 Conclusion

The framework, RLereWolf, has shown its potential for the development of Agents whose performance is not entirely bound on the Game implementation. Furthermore, the metrics and analysis tools the framework provides are not currently provided by any existing framework. Moreover, RLereWolf provides users with:

- **Client** – An access point for users to play the game Werewolf with other humans or Agents on the targeted RLereWolf Server. The Client has a an easily expandable Tk based GUI which is rendered on run-time by the built-in Pygubu builders and renderers.

- Development Framework – The development framework consists of the code base for the Werewolf game implementation, the analytic utilities provided by the built-in training Environment, comprehensive server & game activity logging, and the modular implementation of the four subsystems: Client, Server, Game, and Environment.

- Built-in Agents – Three built-in Agents which allow the framework’s users to use them as a foundation for future Agents. The three Agents are:
  
  - **Dummy Agent** – A stochastic Agent that does random valid actions.
  
  - **Rule-based Agent** – An Agent with an honesty factor who votes for the least trustworthy, according to them, Player.
  
  - **Trainable Agent** – An Agent that can learn from playing multiple games of the current Werewolf Game implementation. Has no pre-existing knowledge of the game and needs to train in order to learn the game’s rules and how to optimally play it.

The importance of RLereWolf is that it will expedite the further development and research of artificial intelligence in Bayesian games whose real-life application can be easily translated to. This is due to the problem’s core nature – working with incomplete information about the environment and making a decision on what the best course of action is, based off of your limited contextual knowledge and past experiences.
The current three, distinct, built-in Agents provided the author with experiment results which, unfortunately, prove that the RLereWolf Agents do not reach the performance of AiWolf’s Agents. Consequently, the project could not manage to achieve its goals of providing the entirety of the promised framework, as the three built-in Agents are not in a finished state (see Chapters 5 and 6 for more information).

Whilst, the RLereWolf Agents bring similar results as previous research (Toriumi et al., 2016; aiW, 2017, 2020), as well as theoretical improvements over its competitor – AiWolf. The current framework iteration does not provide the functioning communication channel improvements and are unused by any of the Agents.

Moreover, the current framework has a faulty Game implementation, which has resulted in non-balanced Game lobbies, which is an inherent issue with using ratio-based distribution systems, as opposed to the standard point-based system.

The negative result of project has largely been a result due to the scope of the project and its allocated time. The author aims to complete the framework and add additional improvements in future releases. However, the the machine learning and development concepts learned by the author during the development of the RLereWolf framework will aide them with the planned future work and provide useful in both academic and career contexts.

### 7.2 Future Work

In this section, the author will discuss some of the future work that has been planned for the framework, be it as a result of truncated functionality, due to the size of the project and its deadline, or – work that was eventually perceived as beneficial to the project.

#### 7.2.1 Agents

In the current iteration of RLereWolf, not all of the Agents are fully completed. The author has planned the completion of the Rule-based Agent and the Trainable Agent, alongside the addition of more Agents, namely a Stochastic Agents – An Agent that does completely random actions, similarly to the Dummy Agent. However, the Stochastic Agent is not guaranteed to make valid Game actions.

Consequently, there should be a base for some untrained Agent – the Stochastic Agent, and a base for some trained Agent that only does valid actions – the Dummy Agent. As a result, Agent performance analysis should prove easier, due to the additional performance reference point.

#### 7.2.2 Multiple Game Action Results

In its current state, RLereWolf can only record one Game action result, per Player action. This can be improved upon by returning a set of results as there is the possibility of overlapping action results. This can be beneficial when training Agents which have violated more than one game rule, e.g. an Agent whose role is a Villager, attacks another Player, during the day time. This would prompt two InvalidAction rewards – one for the attempt at attacking as a Villager and one for attempting a Werewolf action as a Villager (see Section 4.4.2 for more information on rewards).

#### 7.2.3 Point-Based Role Distribution

As briefly mentioned in Section 4.3, the current version of the framework employs a ratio-based approach when distributing the roles, at the start of a Game (see Section 4.3.1 for more information...
on existing role distribution).

The author intends to implement the standard point-based role distribution, as per the Werewolf game \cite{BGG2016} (see Section \ref{section:point-based Role Distribution} for more information on point-based role distribution). Users of the RLereWolf framework will have the option to choose between the standard point-based role distribution, or the bespoke ratio-based role distribution.

\textbf{7.2.4 RLereWolfSharp (C# Iteration)}

As mentioned, the author’s other programming language choice was C#, which they are proficient in, and is gaining popularity in the field of machine learning with the release of \textit{ML.NET} \cite{Csh2018} – an open source machine learning framework for the .NET stack. Furthermore, C#’s package manager – \textit{NuGet}, has access to popular machine learning frameworks such as TensorFlow and Keras.

The immediate challenge with having a C# iteration of RLereWolf is the possibility of code duplication and its avoidance. The author proposes a shared code C# project which uses \textit{Python.NET}\footnote{Python.NET — \url{http://pythonnet.github.io/}} which allows Python code to run within the \textit{Common Language Runtime} (CLR) \cite{Meijer2001,Box2003}, in order to run Python code within the C# implementation. That way the initial iteration of RLereWolf will become the primary code base which can get reused and referenced from other, future, implementations – in this case RLereWolfSharp.

\textbf{7.2.5 Expand Communication Protocol & Natural Language Processing}

Although the currently implemented communication protocol covers more than 50% of the real world communications \cite{Toriumi2016}, the author aims at improving the communication protocol by adding additional messages into the already established message classifiers (see Section \ref{section:Existing Communication Protocol} for more information on existing communication protocol) by expanding the Conflict resolution and Asserting messages. This will be done with the intention of adding a spectrum of certainty, i.e. the messages for Conflict resolution could have presets with adjectives which enforce or minimise the agreement/disagreement.

Moreover, the author plans on having an “open” communication protocol for the human \textit{Players}. This would inherently mean that there needs to be some sort of mechanism to allow for Agents to comprehend the free text, entered by the human \textit{Player} – i.e. with \textit{Natural Language Processing} (NLP) \cite{Liddy2001,Chowdhury2003,Indurkhya2010}.

However, problem of \textit{Natural Language Processing} is complex and as such, the author proposes the “open” communication protocol, only being applicable to human \textit{Players}. This is in accordance to popular \textit{Games} which use preset communications, in order to communicate with Agents, commonly referred to as \textit{Bots}. This separation of “open” and preset communications can be seen in games in the \textit{Counter-Strike} series, where \textit{Bots} would only react to the “radio commands”, which are preset communication messages \cite{CSGO2021}. Consequently, human \textit{Players} would need to use the preset communication messages, whenever they intend to convey a message to the \textit{Agents}, present in their \textit{Game}. 
Appendix A

User Manual

This appendix will go over the functionality of the Client and serves as a guide as to how its users should interact with the graphical user interface. In its current iteration – RLereWolf consists of only three major screens – the Main Menu, Game List, and Game Lobby.

A.1 Main Menu

Once a user starts up the Client, either by double-clicking on the distributed Client executable, or by debugging the Client (see Appendix B for more information) – they will be greeted with the main menu screen which shows them the options to:

- Connect – Establish a connection with the Server, which will change the Client screen to the game list screen – showing all Games which the Client can join.

- Set Name – This is a mandatory action before connecting to the Server. This sets the displayed to other Clients name. It serves as a user alias, as the Clients are identified by their unique Client Identifier.

- Help – Shows a dialog with basic information on how to use the Client (see Figure A.2).

- Quit Game – Closes the Client.

Before a user is able to Connect to the Server, they must set their name (see Figures A.1 to A.3).

![Figure A.1: Main menu – No Client name provided](image1)

![Figure A.2: Main menu – Set Name dialog](image2)
Once a user has set their name, they are able to connect to the Server. When the user is successfully connected to the Server, that is – whenever they get a successful connect response back from the Server, they will be redirected to the Game List screen.

A.2 Game List

The Game List screen contains all “joinable” by the Client Games. The “joinable” Games are defined as Games which have not started. The “unjoinable” Games are those which will not be visible in the list in first place, or Games which are full. In this screen, the Client user has the following options:

- Disconnect – Disconnects from the Server and returns the Client back to the Main Menu.
- Create – Opens up a dialog (see Figure A.6) so that a Client can create a new Game. Once a Game name is specified, the Client automatically joins the newly created Game.
- Join – Joins the selected (see Figure A.5 highlighted Game) by the Client user Game, provided it is “joinable”. If no Game is selected, then the Client will stay on the Game List screen. Once a Game is joined, the Client changes the screen to the Game Lobby screen.
A.3 Game Lobby

The Game Lobby screen is the effective Game the Clients will be able to play. This screen has two modes:

- **Non-started Game** – A non-started Game is a Game in which at least one Player has not set their status as Ready (see Figures A.7 and A.8). The state of readiness of a Player is noted as a “+” and “−” on the left of the Player’s name for being Ready and being not Ready respectively (see Figures A.7, A.8, and A.14). These Games are visible in the Game List screen and can be joined by other Players, as long as the Player limit is not reached. During this phase, all human Players in the lobby are able to add Agents from a set of three possible Agent types – Dummy Agent, Rule-based Agent, or Trainable Agent.

- **Started Game** – A started Game is when all the Players within the lobby have marked their status as Ready, after which the Game initiates, by randomly distributing roles to the Players in the Game (see Figures A.9, A.11, and A.13).

Once a Game has started, the Players are provided with a random role from the set of currently supported roles – Villager, Guard, Seer, and Werewolf. The Game starts off with the day time, where Players must Vote to execute or Wait, that is – skip their turn and do nothing (see Figure A.9).

Moreover, once a Game has started, all Players are able to Talk, using a set of preset messages, during the day time. In order to Talk, a Player must select another Player, whose “targeted” message is intended to. A “targeted” message is that which references another Player, i.e. “Accuse a Player of” (see Figure A.11).

Whenever the time of day is changed – day or night, certain Game actions the Player can do become unavailable. This is because as a Player of your specific role, you are incapable of doing the disabled action within that turn, i.e. Voting during the night as any role. Furthermore, a Player can only see the Actions their role can do (see Figure A.9 for a Guard and Figure A.11 for a Villager).

Once a Game has finished, then the lobby will automatically “reset” and mark all human Players as not Ready. Players can then mark themselves as Ready and start another Game of
Werewolf with the same Players, or alternatively, add more Agents, or wait for new human Players. See Figures A.9, A.10, A.11, A.12, and A.13 for more examples on playing Werewolf.

Figure A.9: Game Lobby – Night time as a Guard

Figure A.10: Game Lobby – Declare your own role as a Villager

Figure A.11: Game Lobby – Send a message to accuse a Player of being a Role

Figure A.12: Game Lobby – Agree with a different Player on a statement they have made
Figure A.13: Game Lobby – Overview of a Started Game

Figure A.14: Game Lobby – Overview of a Non-started Game
Appendix B

Developer Manual

The development of RLereWolf has been done with Visual Studio 2019 and Python 3.7, as such – the author recommends the usage of the Visual Studio IDE for the maintenance and further development of RLerewolf. The Visual Studio solution comes with integrated helpers, which aide the development work. An example of such a helper is SwitchStartupProject\(^1\), which has been set up with the various project combinations that can be launched (see Figure B.1 for the currently set up project combinations). The framework has a few development rules, which are as follows:

- **Maximum line length** – All code files of RLereWolf are to have a line length less of 120 characters when not a comment, and less than 125 characters when a comment. This aims to support users of 3:4 monitor users and increase code readability. A useful Visual Studio Package to keep this in mind is Editor Guidelines\(^2\).

- **Code style** – The code style applied by developers much match that of the existing RLere-Wolf code base. This implies that classes should always have an individual file for them, alongside the continued practice of defining getters and setters for the appropriate properties.

- **Naming convention** – Developers should be mindful of the naming conventions they apply. The in-use naming convention is camelCase whereas public getters and all methods should be using UpperCamelCase.

- **Naming philosophy** – The developers should aim for the coding philosophy in non-documented code should be enough to convey the functionality, whereas any comment blocks should only convey the reason why a certain piece of functionality works the way it does.

Once a developer has a copy of RLereWolf’s solution & Visual Studio installed, they can open up the *Werewolf.sln* file, which contains the various sub-systems, separated as different Visual Studio projects.

Within the root framework folder, there is a *requirements.txt* file, which contains the needed by Python 3.7 packages & their dependencies, in order to run the framework in its entirety. To

\(^1\)SwitchStartupProject – https://marketplace.visualstudio.com/items?itemName=vs-publisher-141975.SwitchStartupProject

install the requirements, the developer should run one of the following commands, where pip targets the developer’s Python 3.7 environment:

\[
pip \text{ install } -r \text{ requirements.txt}
\]

\[
\text{python3.7} -m pip \text{ install } -r \text{ requirements.txt}
\]

The currently set up startup configurations can be seen on Figure B.1 and are defined as follows:

- **Client** – Starts a single instance of the `Client.ClientInstance.py` file, which brings up the Tkinter GUI.

- **Server** – Starts a single instance of the `Server.ServerInstance.py` file, which is a black box, containing a log of all actions & requests. The default `ServerInstance` is configured at localhost:26011, details of this can be seen in `Shared.constants.NetConstants.py`.

- **Client + Server** – Starts off both `Client & Server` projects.

- **Werewolf** – Starts off a configured Game, entirely populated with Agents, which is set in `Werewolf.Environment.py`. The Game has an Environment wrapper around it, which saves game metrics and allows for the “training” of Agents.

![Figure B.1: Project startup options in Visual Studio 2019](image)

**B.1 Deployment**

RLereWolf comes with a set of “build scripts” for the `Client` and `Server`. They are located in the root project directory and are called `BuildClient.py` and `BuildServer.py` respectively. Each file contains calls to `PyInstaller` which “freezes” and packages the project as executables for some target operating system, which is defaulted to the current host machine OS.

In its current form, the build scripts are not complete as they do not include all of the necessary files. However, these should manually call any non-Python files which are inherently required by the project. An example of this can be the `.ui` files which are used in the `Client`.

Once the scripts are finished, developers should be able to run either of the “Build” files and have an executable provided to them which they can distribute to either `Client` users or a host machine, to act as a `Server` (see Figures B.2 and B.3 on building the executables).
B.2 Client

The Client Python Project consists of the GUI implementation (see Section 4.2 for information on functionality). The Client project holds references to the Shared project, and contains usage of Pygubu and a bespoke .ui rendering framework, whose purpose is to abstract away the GUI creation and abide by the Rapid Application Development methodology (Beynon-Davies et al., 1999). The basic GUI development pipeline is as follows:

1. Create a View:

   • Through writing XML in a .ui file; or
   • Through the Pygubu GUI editor.
2. Use the `ScreenBase` class, contained in `Client.screens.ScreenBase.py`, which is the formal “renderer” of the .ui file. An example of `ScreenBase`’s usage can be seen in `Client.screens.GameListScreen.py` and `Client.screens.GameLobbyScreen.py`.

3. Define any handler methods, i.e. Click, Drag, Scroll events, within the new renderer class, which inherits `ScreenBase`.

The descriptions of the code files contained in the `Client` Project, can be seen in Table B.1.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientInstance.py</td>
<td>The entry point for the <code>Client</code> GUI</td>
</tr>
<tr>
<td>MainWindow.py</td>
<td>The root Tkinter Window, used by all screens of the <code>Client</code></td>
</tr>
<tr>
<td><code>constants/ClientConstants.py</code></td>
<td>A configuration file, containing global constants to be used by all <code>Clients</code></td>
</tr>
<tr>
<td><code>context/ServiceContext.py</code></td>
<td>The set of all service calls to the <code>Server</code>, which can be made by the <code>Client</code></td>
</tr>
<tr>
<td><code>context/UIContext.py</code></td>
<td>The set of all navigation manipulating calls, which can be made on the <code>Client</code>, e.g. showing a specific screen or showing a dialog</td>
</tr>
<tr>
<td><code>context/ViewModelContext.py</code></td>
<td>The collection of all “contexts”, supported by the <code>Client</code>, “contexts” are defined as logic units which have some common greater purpose, e.g. UIContext is concerned with UI elements, ServiceContext is concerned with service calls</td>
</tr>
<tr>
<td><code>models/TalkMessage.py</code></td>
<td>The abstraction of a message, seen in the <code>Game</code>. It can be sent by the <code>Server</code> or other <code>Clients</code></td>
</tr>
<tr>
<td><code>screens/GameListScreen.py</code></td>
<td>The code-behind file for the Game List Screen. A list of games a <code>Client</code> can join, with the relevant <code>join</code> and <code>create Game</code> functionalities</td>
</tr>
<tr>
<td><code>screens/GameLobbyScreen.py</code></td>
<td>The de facto <code>Game</code> screen. This can be referred as a game lobby which has a single <code>Game</code> running in it, at any given point of time. Once a <code>Game</code> has finished, it is reset</td>
</tr>
<tr>
<td><code>screens/MainMenuScreen.py</code></td>
<td>The landing screen of the <code>Client</code>, this is the main menu where users can set their username and connect to the binded <code>Server</code></td>
</tr>
<tr>
<td><code>screens/ScreenBase.py</code></td>
<td>The base class for all screens, this is responsible for rendering the .ui files</td>
</tr>
<tr>
<td><code>utility/PacketUtility.py</code></td>
<td>A utility class which provides short-hand methods for creating all necessary <code>Packets</code> to send to the <code>Server</code> with the <code>ServiceContext</code></td>
</tr>
<tr>
<td><code>utility/TalkMessageUtility.py</code></td>
<td>The class responsible for classifying all of the communication preset messages to their respective <code>roles</code>, which can be sent through a <code>Client</code></td>
</tr>
</tbody>
</table>
### B.4. Shared

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>views/GameListScreen.ui</td>
<td>The XML definition of the game list screen, created with Pygubu</td>
</tr>
<tr>
<td>views/GameLobbyScreen.ui</td>
<td>The XML definition of the de facto Game screen, created with Pygubu</td>
</tr>
<tr>
<td>views/MainMenuScreen.ui</td>
<td>The XML definition of the mainmenu screen, created with Pygubu</td>
</tr>
</tbody>
</table>

**Table B.1:** Client Project code listings

### B.3 Server

The **Server** project contains the implementation of the *Game* host, which multiple *Clients* can connect to. The **Server** is also responsible for logging requests, both to the **Server** itself and any *Games* hosted onto it.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HandlerContext.py</td>
<td>The handler &quot;context&quot; contains a reference to all handlers the <strong>Server</strong> has access to. Handlers are the logic units of the server, which are split by general functionality – similarly to the <strong>Client</strong> &quot;contexts&quot;</td>
</tr>
<tr>
<td>ServerInstance.py</td>
<td>The <strong>Server</strong> entry-point</td>
</tr>
<tr>
<td>handlers/GameActionHandler.py</td>
<td>The handler containing all logic concerned with <em>Game</em> actions, i.e. <em>Vote</em>, <em>Whisper</em>, <em>Attack</em> etc.</td>
</tr>
<tr>
<td>handlers/GameLobbyHandler.py</td>
<td>The handler containing all logic concerned with the game lobby list, that is – the ability to get the list of available <em>Games</em>, joining a <em>Game</em> etc.</td>
</tr>
<tr>
<td>handlers/HandlerBase.py</td>
<td>The base class for all handlers. Contains boilerplate code and helper references</td>
</tr>
<tr>
<td>utility/ConversionHelper.py</td>
<td>A utility class to convert specific models to DTOs which can be passed to the <strong>Client</strong>. Currently used to translate a <em>Game</em> to a <em>GameDto</em></td>
</tr>
</tbody>
</table>

**Table B.2:** Server Project code listings

### B.4 Shared

The **Shared** Project contains code files which are used in more than one Project, from the set of currently existing Projects – **Client**, **Server**, and **Werewolf**. The most prominent usage for the **Shared** Project is for the data transmission files and common data files that both the **Client** and **Server** use, in order to communicate with each other.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
</table>

**Shared source code listings**
<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet.py</td>
<td>The base communication “unit” that the Client and Server use to communicate to each other. A Packet has data and a type</td>
</tr>
<tr>
<td>constants/CommunicationPresentConstants.py</td>
<td>Contains the definitions of all preset communications that can be sent from the Client</td>
</tr>
<tr>
<td>constants/GameConstants.py</td>
<td>The configuration for each Game – min/max Player count and the role distribution ratios</td>
</tr>
<tr>
<td>constants/LogConstants.py</td>
<td>Tags used for logging, used to classify messages which are stored in the logs</td>
</tr>
<tr>
<td>constants/NetConstants.py</td>
<td>The Server configuration for the host IP, port, and formats for logging. Contains constants used in the socket connection established between a Client and the Server</td>
</tr>
<tr>
<td>dtos/AddAgentGameDto.py</td>
<td>The DTO sent by the Client whenever they want to join a Game</td>
</tr>
<tr>
<td>dtos/ConnectDto.py</td>
<td>The DTO sent by the Client whenever they want to connect to the binded Server</td>
</tr>
<tr>
<td>dtos/CreateGameDto.py</td>
<td>The DTO sent by the Client whenever they want to create a Game on the Server they are connected to</td>
</tr>
<tr>
<td>dtos/GameActionDto.py</td>
<td>The DTO sent by the Client whenever they are doing a specific Game action within a Game they are in, e.g. Whisper, Attack, Vote</td>
</tr>
<tr>
<td>dtos/GameDto.py</td>
<td>The DTO which represents a minimal version of the current Game state for some Player. Not all GameDtos given by the Server are the same</td>
</tr>
<tr>
<td>dtos/GameListDto.py</td>
<td>The DTO which contains all of the Games which a Client can see in the game list screen</td>
</tr>
<tr>
<td>dtos/MessageDto.py</td>
<td>A DTO which represents a minimal version of the Messages within a Game. Clients do not get all Messages</td>
</tr>
<tr>
<td>dtos/MessageMetaDataDto.py</td>
<td>The DTO which contains various meta data for some Message with details to who the Message is targeted to and what the message type is. Used as a means to defer the need of Natural Language Processing</td>
</tr>
<tr>
<td>dtos/PlayerGameDto.py</td>
<td>The DTO which contains the Player identifier and the Game state they have requested. This is used by the Server to pass back the Game state to the Client</td>
</tr>
<tr>
<td>dtos/PlayerGameIdentifierDto.py</td>
<td>The DTO which a Player passes to a Server, in order to get the latest Game state</td>
</tr>
</tbody>
</table>
### B.5 Werewolf

The author’s recommendation is that logging is disabled whenever using the “auto-play service” \((\text{TrainableEnvironmentWrapper})\) or training as it will decrease the average game time by 2.5-3 times. All metrics provided by the \(\text{Environment}\) will be stored in “Statistics.csv”, stored within the

<table>
<thead>
<tr>
<th>File Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dtos/UpdatedEntityDto.py</strong></td>
<td>A DTO which is used as a wrapper over some other DTO, providing a datetime, created by the \textit{Server}, which provides the “birthdate” of that child DTO. This is used as a minor version control for \textit{Messages} and \textit{Game} states.</td>
</tr>
<tr>
<td><strong>enums/AgentTypeEnum.py</strong></td>
<td>An enum class containing all \textit{Agent} types.</td>
</tr>
<tr>
<td><strong>enums/FactionTypeEnum.py</strong></td>
<td>An enum class containing all “factions” supported by the \textit{Game}.</td>
</tr>
<tr>
<td><strong>enums/MessageTypeEnum.py</strong></td>
<td>An enum class containing all types of \textit{Messages} that can be created.</td>
</tr>
<tr>
<td><strong>enums/PacketTypeEnum.py</strong></td>
<td>An enum class containing all types of \textit{Packets}.</td>
</tr>
<tr>
<td><strong>enums/PlayerTypeEnum.py</strong></td>
<td>An enum class containing all of the roles, supported by the \textit{Game}.</td>
</tr>
<tr>
<td><strong>enums/TimeOfDayEnum.py</strong></td>
<td>An enum class containing all of the times of day, which can be played through in a turn of the \textit{Game}.</td>
</tr>
<tr>
<td><strong>enums/TurnPhaseTypeEnum.py</strong></td>
<td>An enum class which contains all game turn phases – \textit{Introduction}, \textit{Discussion}, \textit{Event}, \textit{Accusation}, and \textit{Voting}.</td>
</tr>
<tr>
<td><strong>enums/VoteResultTypeEnum.py</strong></td>
<td>An enum class containing all possible results from some \textit{Game} action.</td>
</tr>
<tr>
<td><strong>exceptions/GameException.py</strong></td>
<td>A specific type of exception that is handled by both \textit{Server} and \textit{Client}.</td>
</tr>
<tr>
<td><strong>utility/DateTimeUtility.py</strong></td>
<td>A utility class which deals with the conversion of date time’s from UTC to Local and Local to UTC.</td>
</tr>
<tr>
<td><strong>utility/Helpers.py</strong></td>
<td>A generic helpers method container – contains references to third-party packages who have useful utility methods, i.e. \textit{Faker’s} name generation, and \textit{varname’s} \texttt{nameof} utility.</td>
</tr>
<tr>
<td><strong>utility/KillableThread.py</strong></td>
<td>A custom thread which can \textit{gracefully} be awaited/killed off on a secondary thread with minor exception handling. Used extensively in the \textit{Client} to clean off any recursive polling threads.</td>
</tr>
<tr>
<td><strong>utility/LogUtility.py</strong></td>
<td>Utility methods for logging \textit{Messages}, \textit{Actions} or some string. Used primarily in the \textit{Werewolf} and \textit{Server} Projects. The logging functionality creates logs on the host machine the utility methods are called on.</td>
</tr>
</tbody>
</table>

| Table B.3: \textit{Shared} Project code listings | |

---

The author’s recommendation is that logging is disabled whenever using the “auto-play service” \((\text{TrainableEnvironmentWrapper})\) or training as it will decrease the average game time by 2.5-3 times. All metrics provided by the \textit{Environment} will be stored in “Statistics.csv”, stored within the
root Werewolf Project folder (see Section 4.4 for more information on metrics).

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment.py</td>
<td>The entry-point for the training process, doubles as the configuration file for what the training Game will have in terms of Agents and how many games the Environment &amp; auto-play service must go through.</td>
</tr>
<tr>
<td>TrainableEnvironmentWrapper.py</td>
<td>A bridge class between the Environment &amp; the Trainable-Player.</td>
</tr>
<tr>
<td>agents/AgentPlayer.py</td>
<td>The base Agent class which is used by all subsequent Agent definitions. Used as an abstract class to make sure the APIs RLereWolf expects are there.</td>
</tr>
<tr>
<td>agents/DummyPlayer.py</td>
<td>The most basic built-in Agent, does stochastic, valid Actions.</td>
</tr>
<tr>
<td>agents/RuleBasedPlayer.py</td>
<td>An intermediate built-in Agent, has a basic understanding of honesty and trust. Does mostly stochastic actions with minor reasoning, all actions are valid.</td>
</tr>
<tr>
<td>agents/TrainablePlayer.py</td>
<td>The built-in Reinforcement Learning Agent. Has no understanding of the Game and can do invalid actions. Learns how to play Werewolf by playing multiple Games with an Environment around it.</td>
</tr>
<tr>
<td>environment/Observation.py</td>
<td>All information about the Game state an Agent can “see”.</td>
</tr>
<tr>
<td>environment/Statistics.py</td>
<td>The metrics recorded by the Environment for the Game in it. The metrics recorded are local and global.</td>
</tr>
<tr>
<td>environment/TrainingRewards.py</td>
<td>The definition of all rewards/penalties, given to Agents.</td>
</tr>
<tr>
<td>environment/WerewolfEnvironment.py</td>
<td>The actual Environment which wraps over the Game which “listens” for results from the Game for some Agent action. Has the auto-play service integrated which just makes sure the Game within the Environment is never stuck.</td>
</tr>
</tbody>
</table>

Table B.4: Werewolf Project code listings


Bray, T., Paoli, J., Sperber-McQueen, C. M., Maler, E., Yergeau, F., et al. (2000). Extensible markup language (xml) 1.0.


Python (2012). Tkinter – python interface to tcl/tk – python 3.9.5 documentation. [https://](https://)


