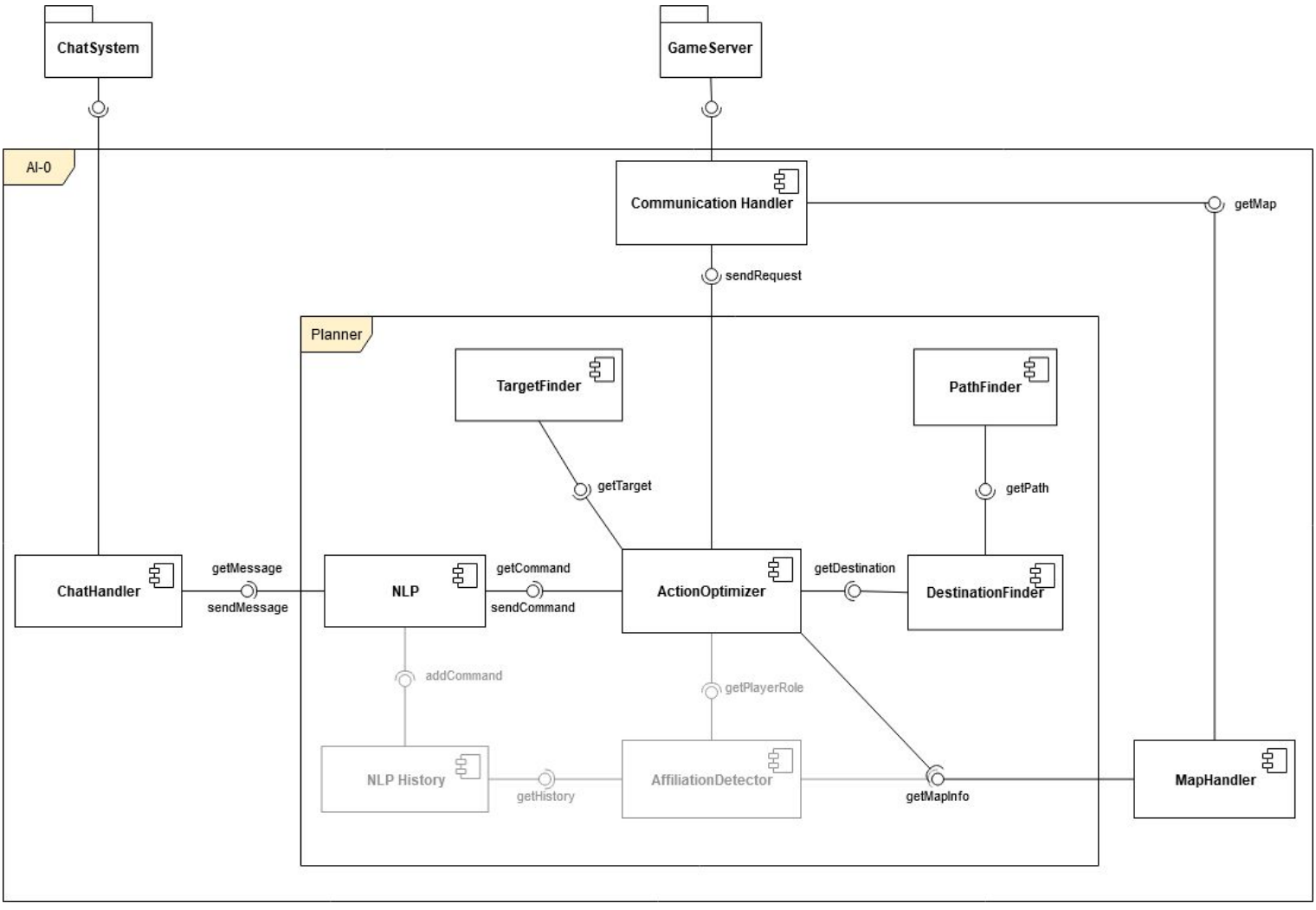
AI-0 Architecture - Beta

Gabriele Tenucci, Stefan Motoc, Luigi Quarantiello

Architectural Diagrams

We developed two kinds of agents: the first one based on a pure algorithmic approach and the second one on ML techniques. Since the results provided by the ML approach were underwhelming, we have chosen the other one as our main agent implementation.
To better present the chosen architecture, we decided to show its *Component* and *State* diagrams, with a detailed description of the way our agent works.

Component Diagram



Components

- **ChatHandler**:
  This component interacts with the *ChatSystem* API and it retrieves and sends messages in the chat channel of the match. It offers an interface to the *NLP* module to allow the AI to interact with the chat.

- **CommunicationHandler**:
  It handles the network communication with the *GameServer*, managing the timings according to the game protocol specifics.
  It also deals with the packages arriving from the *GameServer*, transforming them into messages to be passed to:
  - *ActionOptimizer*: to which it sends the responses to the requests
  - *MapHandler*: which receives the current state of the map

  This component also transforms the messages received from the *ActionOptimizer* (which are actions to be performed by the agent) into packages, in order to send them to the *GameServer*.

- **Map Handler**:
  It receives the most updated version of the map from the *CommunicationHandler* and it extracts information about:
  - general map configuration, such as walls, terrain type and objects
  - the position of other players

  It keeps the history of the players' movements; it then sends these information to the *AffiliationDetector* and *ActionOptimizer*.
  The *MapHandler* is also able to approximate the current state of the map by using an updated version of the status and an old configuration of the map.

- **Target Finder**:
  This component receives from the *ActionOptimizer* information about the position of the enemies and the map configuration; then it estimates the best target to shoot to (if any), according to its distance from the agent.
  Its results will be used by the *ActionOptimizer*, which will decide whether shooting the target is the best option or not.

- **Destination Finder**:
  It receives from the *ActionOptimizer* information about the position of the enemies, the map configuration and the position of the objects.

Then it estimates the best areas where to move to, according to the position of objects and enemies, with their estimated paths and shooting lines.
The results of this component will be used by the *ActionOptimizer*, which will decide whether moving to the destination is the best option or not.

- **PathFinder**:
  This component receives the best locations where to move to from *DestinationFinder*, and it computes the best path to reach them. In particular, it tries to find the safest route while keeping optimality in the number of steps.
  Its results will be used by the *DestinationFinder*, which will then send them to the *ActionOptimizer*.

- **NLP**:
  The *NLP* module interacts with *ChatHandler* by extracting information from the received chat messages.
  The final version of this component was meant to send the extracted information to the *NLPHistory* and *ActionOptimizer,* to receive new commands from the *ActionOptimizer* and then translate them into natural language.
  The current version (beta) is only able to identify a few patterns in the received messages (standard server messages) and to send preconfigured chat messages from a pool of premade phrases, according to the action performed by the agent. For example, if the agent is defending its position, the module will randomly choose one of the defensive phrases and send it to the *ChatHandler*.

- **NLP History**:
  This module receives the extracted information from the *NLP* module and it stores it, in order to be used by the *AffiliationDetector*.

- **Affiliation Detector**:
  This module receives the chat history from *NLPHistory* to get the recent messages from each player, and it performs sentiment analysis on it.
  It receives the map history from *MapHandler*, in order to reconstruct the recent movements of each player.
  It then uses the available information to estimate whether each player is an impostor or not and passes this information to the *ActionOptimizer* module.

- **Action Optimizer**:

  This module receives information from *NLP* and *MapHandler* and it sends it to *TargetFinder* and *DestinationFinder*.

  It decides the next operation to be performed by the agent, based on the results from the other modules. For example, if it decides that the best operation is to shoot an approaching enemy, it waits until the enemy is in shooting range and then it fires. Alternatively, if the decision is to move, it decides how many steps to perform on the path to the target location, in order to be sure that it will not cross any enemy shooting lines.

  The *ActionOptimizer* interacts directly with the *CommunicationHandler* to send requests to the *GameServer,* in order to perform actions such as move, shoot or accuse a player.

  It also decides if and what to write onto the chat, taking care of the timings so that it can look more human-like.

The NLP module we have developed so far is a crippled version of the one we had planned to implement in the final stage. In particular, it does not do any actual “understanding” of the messages, but is only able to interpret pre-defined server messages, such as the ones informing us that the game has started or that a player has been killed.

The grayed-out modules of the Component Diagram (*NLPHistory* and *AffiliationDetector*) are not yet fully developed, since they were scheduled for the final version of the game. We have decided to focus on other aspects of the agent as we have noticed that users seldom used the chat to send messages while playing, given that matches are short and human players are focussed on moving and shooting.

General State Diagram

This is the state diagram for the moving and shooting logic of our agent, explaining its underlying strategy.

- **Lobby**: upon entering the lobby of the match, the agent requests the map of the match and the status; then, it updates the status periodically, to discover new players and to find out if the game has started. It also uses this initial time to initialize everyone as AIs, since that is the simplest strategy to be applied to play the Turing game. Once the game has started, the agent moves to the *Initial Phase* state.
- **Initial Phase**: this phase only lasts for the first 5 seconds of the game, in which the players can only move and they can't shoot enemies.
  The agent uses this initial time to find an area of the map where the enemies cannot shoot it, then it moves in such a safe area until the initial phase is over. After that, it shifts to the successive state, according to its loyalty.

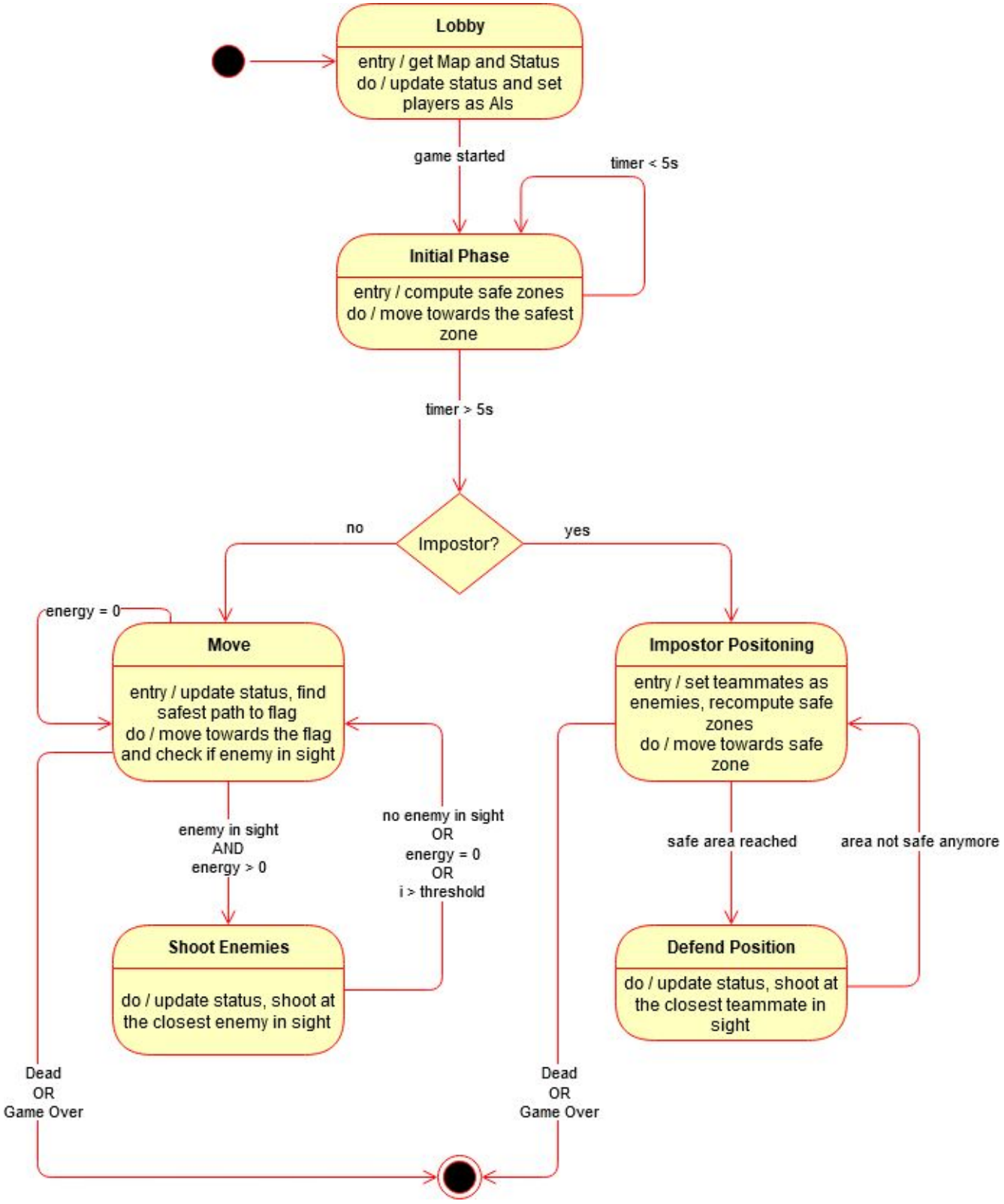
Loyal agent case

- **Move**: Initially, the agent updates the game status, and uses the new information on its opponents' position to find a path towards the enemy flag. In particular, this path changes at every status update, as it tries to avoid the enemy shooting lines.
  After that, it moves by *n* steps along this new path, where *n* is the maximum number of steps the agent can take without crossing enemy lines. In this way, it is sure that no one is going to shoot him while moving. It repeats this process until it cannot move anymore in a safe way, because an enemy is in sight; at this point, it moves to the *Shoot Enemies* state.
  The only exception is when the agent has no energy left; in this case, it remains in this state and it tries to reach the opponent flag.
- **Shoot Enemies**: At this point, the agent is close to an opponent and has energy to shoot. It keeps requesting the status, and it shoots to the closest enemy as soon as his path crosses its fire line.
  After hitting the target, or if its energy becomes 0, it returns to the *Move* state.
  Additionally, to avoid infinite loops, we have implemented a *threshold* variable, so that after *threshold* cycles are passed it goes back to the *Move* state.

Impostor agent case

- **Impostor Positioning**: The agent starts by identifying its teammates as enemies and recomputing the safest areas accordingly. After that, it reaches the most secure area among the ones nearby. Once it arrives there, it moves to the *Defend Position* state.

- **Defend Position**: In this state, the agent holds its position and it updates the status, so that it can notice when a teammate is approaching its shooting range and it can try to kill him.
  The agent returns to the *Impostor Positioning* state as soon as the current area is not safe anymore, meaning that it is close to an enemy shooting line.



Machine Learning Approach

The other agent we developed is based on Machine Learning techniques. In particular, we used a Reinforcement Learning approach, in order to train our model by letting it play hundreds of matches against itself, and giving positive or negative reinforcements according to the actions performed by the agents.
To do that we created a training environment, which is an abstraction that allows the agents to observe the world and interact with it.
In our approach, the observation corresponds to the current configuration of the map and the information about the current player extracted from the game status, while the actions are the ones that can be done by a basic player (i.e. move, shoot).

We have standardized the training phase by playing 5v5 balanced training matches, on a small rectangular sized map (with the idea of incrementing the map size after obtaining good training results). Since the agents were obviously performing random actions in the beginning, we had to set a maximum amount of actions per player, in order not to reach the server timeout for the match length.

We first tried to use a basic Q-Learning agent, but it was clear it wasn't able to learn properly given the high complexity of the game.
Because of this, we have decided to increase the complexity of our agent by mixing a neural network approach with Q-Learning. This technique is generally called Deep Q-Learning.
We created a network of 3 densely connected layers: 64*128, 32*64, 11 nodes respectively, with the output layer representing the mapping of the possible actions the agent can take.
We have trained this agent for hundreds of matches, and it showed visible improvements compared to the simpler Q-Learning one. Still, this agent was not even remotely close to the strength level of the AI we developed with much simpler techniques.

We think that one of the possible solutions could be to use a deeper network and train for a much longer amount of time. Unfortunately, we were not able to do so because of the lack of computational power and time.
Given those limitations, we have decided to stop developing this kind of AI and focus on the simpler approach, which looked more feasible for us.

Had we kept developing this agent, the next step could have been to use data from previous matches to implement an LSTM neural network, to predict the next state of the map.

Technological Choices

The language we chose to develop the agent is Python, since it is both simple and powerful, as it includes a large amount of Artificial Intelligence libraries.

To communicate with the servers, we have used the library *telnetlib* which provided a quick way to establish a Telnet connection and to send and receive messages through the sockets.

For the pathfinding part, we have used the *A* algorithm,* included in the external library *python-pathfinding*. We have chosen this algorithm because it offers the right tradeoff between optimality and time complexity.

For the Machine Learning implementation of the agent, we have used both a *Q-Learning* approach, using the *stable baselines* library, and a *Deep Q-Learning* approach, running on *Tensorflow* with *Keras*. The reason why we have chosen these frameworks is they already provided state of the art tools for dealing with our type of problem.
We have also used *Gym* to simplify the procedure by allowing us to create a training environment for the agents to learn.

We have used *Google Colab* to develop our code, as it provides a common playground to share among the whole team, allowing us to test the code on the fly without requiring any additional software installed locally.

Thanks to machine-learning algorithms,
the robot apocalypse was short-lived.

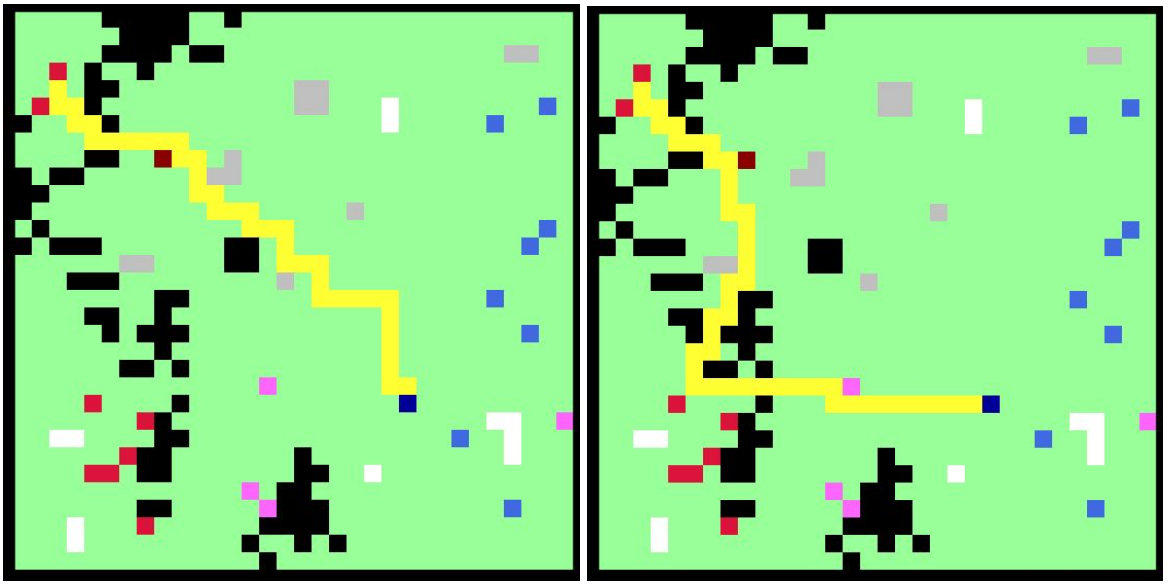
Peculiarities and Code

Safe Zones

To implement the general strategy of the agent we have developed a way to compute the dangerousness of different areas of the map, according to the enemies presence and their shooting lines. In order to do so, starting from the original map, we calculate a weighted map by assigning a bigger cost to the rows and columns reachable by enemy fire. If multiple agents can directly shoot at a specific cell, the cost adds up.
We then divide the map into smaller subsections and for each one of those we compute its danger level, by calculating the average value of every cell in that area. If the agent is not an impostor we also add to that value the distance from the flag, so that our player will prefer areas closer to it, while traveling in a safe way.

Smart Pathfinding

To find a path to the flag, we use the *A* algorithm* computed on a modified weighted version of the map. In this version, we take into consideration the enemy shooting lines, to compute how safe a cell is; the dangerousness decreases as the distance between the shooting enemy and the current cell increases. We do that in order to always choose the farthest dangerous cell, so that if the enemy got the timing right, it still has to consume more energy to hit us.



In the two figures above, the yellow lines correspond to the path from the top red cell (our agent) to the dark blue one (enemy flag). In the one on the left, we have the path found by *A** computed on the default weighted map, which is slightly shorter but

goes through several shooting lines. The one on the right, on the other hand, is the result of our smart pathfinding, which produces a slightly longer but much safer path.

Shooting

When we are in shooting mode, we keep updating the status to see if an enemy is approaching. If we find one directly in our fire line, we simply shoot. Otherwise we try to anticipate the approaching enemy's movements by shooting just before it enters our range.
Before shooting, we also check the fire line between us and the target to find if there are any obstacles, in order to avoid wasting energy.

Map Updates

To avoid using too many map update requests, we have implemented a way to update the map by parsing the result of the status requests, since it contains the updated coordinates of each player.
Every time we do a status request to the server, we modify our stored map by moving each player to their new positions and substituting the old coordinates with grass. Because of this, the updated map is only an approximation of the real one, especially in the case when the old coordinates contained a river tile or if a barrier was moved in the meantime. Nonetheless we found this to be a good tradeoff since it allows us not to waste time on map requests.

Timers

In order to avoid wasting more time than needed between one command and the other, we have used a timer to store the last time we have sent a command. When a new command arrives to the *CommunicationHandler*, we calculate if enough time has passed, according to the protocol specifications. If not, we wait the exact amount of time, calculated with a difference between the current time and the last time we have sent a command.

We also use some of these "dead times" to read and write on the chat. For writing on the chat, we have another, randomized, timer, which allows the agent to send chat messages in a non-systematic way, in order to look more "human".

User Manual

To run the agent, simply download the *ai0-agent.py* file from the Github repository, and run it in a terminal with
`>python ai0-agent.py <matchname> <playername>`
The agent will start playing the game as soon as the game is started. You can monitor the actions taken by the AI player by reading the output console.

We also provide a simple “runner” agent, which only runs towards the flag trying to avoid enemies without shooting.
You can run this agent by downloading the *ai0-runner.py* file and running it with
`>python ai0-runner.py <matchname> <playername>`

You may be required to install the *python-pathfinding* library. In that case, you need to run in a terminal
`>pip install pathfinding`

Reflection on the development

At first we experienced some problems due to the distance learning, but after the first period of adaptation we started seeing the interesting aspects of the development.
It was our first experience with a *“big”* project, in which we went through all the stages of development, from defining the requirements to the actual implementation and end-to-end evaluation. We found this challenging at first, since we were not used to coordinating with so many people and to depend on the results of other groups.
We also had the chance to put our hands on algorithms we had only seen in theory before.

Despite all of the difficulties we experienced during the course, we enjoyed the ride, which overall turned out to be an enriching experience.