MAW: Multi Agent Werewolf

SENG 696 - University of Calgary

Design Document

Version 0.2

Olga Koldachenko

November 10, 2024

Contents

1 Werewolf: A Party Game

1.1 A Bloody History

Mafia (sometimes called Werewolf, which is the name I'll use going forward) is a game for seven or more players originally designed by Moscow University psychology major Dimitry ("Dimma") Davidoff in the year 1986 [1]. At its core, this game is about deception, where "honest" players are infiltrated by killers. This game has since gone though many iterations, for example through Andrew Plotkin's renaming of the game to Werewolf and his addition of a new role [2]. During the COVID-19 pandemic, a variation of the game became popular under the name Among Us, which gives it a science fiction spin [3].

While Werewolf is originally played in person, with the lights used to control the phases of the game and cards to denote players' roles, it has translated well to online chatroom environments. With the rise of large language models capable of interpreting and generating text, there is the potential to create autonomous agents who can play this game in a way that would mimic humans.

There is, in fact, a competition for developing such agents in Japan called AIWolf [4], where the goal is to have the AI exclusively play as the role of the wolves, while human players attempt to find the imposter. Wang et al. have demonstrated the potential of deep reinforcement learning to create such agents not only on the werewolf side, but also that of the honest villagers [5]. It's also possible to take the models trained on games of Werewolf and apply them to more practical applications like botnet detection, as shown by Javadpour et al [6].

A web archive of the original rules as written by Davidoff can be found via the Wayback Machine [1], which I'll be using as the basis for this project. I'll be modifying the rules for my own purposes, with this iteration of the game being referenced as MAW (Multi-Agent Wolf).

1.2 Rules

1.2.1 Sunrise Phase

The players are randomly split into two roles: Werewolf and Villager, with one third of the players granted the Werewolf role. The Werewolves are notified of which other players are on their team, while the Villagers are left ignorant. Once the setup is complete, the game will cycle between the Day and Night phases, until either the Villagers have executed all of the Werewolves, or the Werewolves have murdered all of the Villagers.

Dead players can observe the game, but not participate.

1.2.2 Day Phase

A timer is set for the remainder of the phase.

If a player has been murdered in the previous phase, then the system announces the name of the murdered player.

Here, all players are placed into a day phase chatroom (the "Village"), and are left to freely talk. Each player has a public vote. Voting for another player is an accusation of the

target's guilt. Votes can be changed, as long as the phase hasn't ended yet.

If there's a majority vote for any given target, then that player is executed and the phase ends early. The name of that player is anounced by the GM, but their role is not revealed to the Day Phase chatroom, and the chatroom is closed (will not accept messages) until the next Day phase.

If the timer runs out before a majority vote can be decided, then the player with the most votes is executed instead.

If no votes are made, then no player is executed, and the next phase continues as normal.

1.2.3 Night Phase

A timer is set for the remainder of this phase.

Here, all Werewolf players are placed into a night phase chatroom (the "Hideout"), and are left to freely talk. The role of the player executed in the previous phase, if any, is announced by the system. Like in the Village, each player is given a vote. However here, a unanimous vote is needed to decide on which Villager to murder.

Once a unanimous vote has been made, then the phase ends early and the target is murdered.

If the timer runs out without a unanimous vote, then no Villager is murdered at the end of this night.

1.2.4 The End

At the end of the game, if the Werewolves win, they receive points equal to the initial number of Villagers. If the Villagers win, they receive points equal to the number of remaning Villagers.

The game can be played over multiple rounds, with points accumulating over the rounds.

The player with the most points at the end wins!

1.3 Enter the MAW

The first version of the MAW system will be playable through a command line interface, with the newest messages from any player showing up as they arrive. Messages are sent to and forwarded by a centralized server, acting as Game Master (GM), which creates an agreed-upon, unified game state between the players.

What sets the MAW system apart is its reliance on one or more large language models (LLM), which will allow the agents to communicate among each other as if they were human players. Rather than using heuristics or hard-coded decision logic to determine how to act at any moment, the agents will follow a *guiding prompt*, which will inform the LLM of the format of the expected output. The prompt can include:

- Decisions available to the agent (speak, vote, do nothing).
- The maximum length for any dialogue.
- Expected syntax for voting commands.
- A list of potential vote targets.

As part of this project will be to determine what specific commands are appropriate in order to get the intended behaviour out of the bot, the previous points are liable to change. If no success is seen with this method, it may be required to fall back onto heuristic methods or reinforcement learning instead. As it's possible for LLMs to output code, I hypothesize that even a relatively small model run on a local machine should be able to handle this trinary decision space.

Algorithm 1 An internal dialogue.

M is the set of strings in an incoming message queue from the game server
g is a guiding prompt
LLM is a large language model API
out is the outgoing message queue to the game server
n is the max number of seconds to wait for M to accumulate messages

repeat
  wait $rand(n)$ seconds *//to prevent overwhelming the server*
  post $M \cup \{g\}$ to LLM
  clear M
  $r \leftarrow LLM.response$
  if r contains "*listen*" **then**
    do nothing
  else
    forward r to *out*
  end if
until the game is over

1.4 Nice-to-Haves

These features, while not core to the game, would enhance the software if time permits. They are not currently included in the schemas within the rest of the document, but may be amended later.

- A webpage designed for a human to monitor the game and view the internal state of any given agent while the game is running. This could also allow the human to more naturally play the game, rather than through a command line interface.
- A "whisper" or "private message" feature between the players. This could be implemented as direct peer-to-peer communication between the players, which would require extra checks to ensure that dead players cannot communicate with live ones.
- Expanding the game rules to allow for additional roles, such as the Seer designed by Plotkin, who can, once per day phase, query another player's role [2].
- Modifying the system to align with AIWolf's protocols, enabling these agents to compete with others. As their version of the game enables features that are out of scope for this project, this will be saved for a later time.

2 System Specifications

2.1 Roles

Role Schema: PLAYER
Description: The control scheme with which a human or agent can interact with the game.
Protocols and Activities: ChooseName, JoinLobby, GetNewMessages, GetPlayerInput, Speak, Vote, PrintNextMessage, Listen
Permissions: **reads** supplied *role* *//villager or werewolf* supplied *status* *//dead or alive* **changes** *name* *//the name to send to the lobby*
Responsibilities Liveness: PLAYER = ChooseName. JoinLobby. (SENSE$^{\omega}$ || ACT$^{\omega}$) SENSE = GetNewMessages. [PrintNextMessage]* ACT = GetPlayerInput. [Speak | Vote | Listen] Safety: a successful connection to the lobby

Table 1: Role schema for a player

Role Schema: USERINTERFACE
Description: A space for a human player to input text.
Protocols and Activities: ReadInput, SendMessage
Permissions: **reads** supplied *inputStream* *//CLI to read from*
Responsibilities Liveness: USERINTERFACE = (ReadInput. SendMessage)$^{\omega}$ Safety: successful interface with the player client

Table 2: Role schema for user input

Role Schema: ModelInterface		
Description: Runs the connection between the large language model and the game server		
Protocols and Activities: QueryLLM, SendMessage		
Permissions:		
reads	supplied *modelAPI*	*//API for the LLM*
	strategy	*//Influences decison making*
changes	*guidePrompt*	*//varies based on world state*
	messageQueue	*//contains chat and system messages to process*
Responsibilities		
Liveness: ModelInterface = (QueryLLM. SendMessage)$^{\omega}$		
Safety: successful connection to the LLM's API and to the game server		

Table 3: Role schema for LLM input

Role Schema: Lobby		
Description: The outermost connection to the game world. Coordinates the game's rounds.		
Protocols and Activities: ConfirmPlayer, StartRound, AnnounceScores, AwaitPlayer, AwaitRoundEnd		
Permissions:		
reads	supplied *maxPlayers*	*//number of players to wait for*
changes	*playerData*	*//contains name, IP, points*
	roundNumber	*//tracks the current round of the game*
Responsibilities		
Liveness: Lobby = (AwaitPlayer. ConfirmPlayer)+. Round+ Round = StartRound. AwaitRoundEnd. AnnounceScores		
Safety: successful connection to all players, safely handling disconnections		

Table 4: Role schema for a game lobby

Role Schema: MODERATOR	
Description:	
Assigns game roles to players, declares phases, and counts votes.	
Protocols and Activities:	
AssignRole, OpenRoom, CloseRoom, AnnounceVoteResult, AnnounceWinningTeam, AwaitPhaseEnd,	
Permissions:	
changes	supplied *playerData* //*status of current players* *phaseTimer* //*controls time limit for phases* *currentPhase* //*influences vote logic*
Responsibilities	
Liveness:	
	MODERATOR = AssignRole+. PHASE+. AnnounceWinningTeam. PHASE = OpenRoom. AwaitPhaseEnd. AnnounceVoteResult. CloseRoom
Safety:	
	enforced synchronization of the current phase among players

Table 5: Role schema for a moderator

Role Schema: CHATROOM	
Description:	
Contains a subset of the total playerbase, facilitating group discussion.	
Protocols and Activities:	
AnnounceOpen, AwaitMessage, Verify, AddToLog	
Permissions:	
reads	supplied *roomName* //*lobby, day, or night* supplied *playerData* //*players who can view the chat*
changes	*chatLog* //*can be retrieved on request* *roundNumber* //*tracks the current round of the game*
Responsibilities	
Liveness:	
	CHATROOM = AnnounceOpen+. (AwaitMessage. Verify. AddToLog)$^{\omega}$
Safety:	
	successful connection to the lobby

Table 6: Role schema for a chatroom

2.2 Interactions

Protocol	Purpose	Initiator(s)	Responder(s)	Processing
ChooseName	Requests a name from the player or model	Player	UserInterface, ModelInterface	Used when connecting to the lobby
JoinLobby	Connects to a game as a new player	Player	Lobby	Game starts when all players have connected
GetNewMessages	Requests unread messages from a chatroom	Player	ChatRoom	Only succeeds if the player has rights
GetPlayerInput	Requests the next input from the human or LLM	Player	UserInterface, ModelInterface	Can be dialogue or a vote
Speak	Sends dialogue to a chatroom	Player	ChatRoom	Will be seen by all others in the room
Vote	Sends a vote to the moderator	Player	Moderator	Votes can be parameterized to be silent or announced by the system
SendMessage	Sends a message from the human or LLM to a player interface to be further processed and forwarded	UserInterface, ModelInterface	Player	Can be any generated text
ConfirmPlayer	Informs a player that they have successfully joined a game	Lobby	Player	The player's name must be valid

Table 7: Interaction model

Protocol	Purpose	Initiator(s)	Responder(s)	Processing
StartRound	Initiates the start of a round of the game	Lobby	Moderator	The first phase will be day
AnnounceScores	Sends the leaderboard	Lobby	ChatRoom	Calculated at the end of each round
AssignRole	Tells a player what their role is (Villager/Werewolf)	Moderator	Player	One third of players should be wolves
OpenRoom	Notifies a room that it should open and to which players	Moderator	ChatRoom	Only one room should be open at a time
CloseRoom	Notifies a room that it should reset	Moderator	ChatRoom	The room should refuse all messages
AnnounceVoteResult	Notifies a room of the result of a vote	Moderator	Chatroom	The target player will be considered dead
AnnounceWinningTeam	Notifies the room and the lobby of the winning team	Moderator	ChatRoom, Lobby	Signals the end of a round of the game
AnnounceOpen	Lets a player know when they've been placed into a new chatroom.	ChatRoom	Player	Signals a change of phase

Table 7: Interaction model (continued)

2.3 Agents

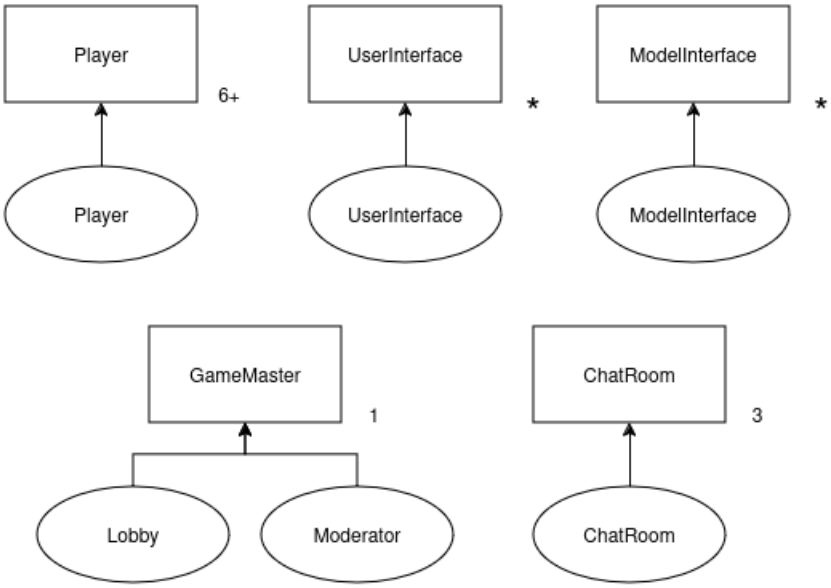

Figure 1: Agent model

2.4 Acquantances

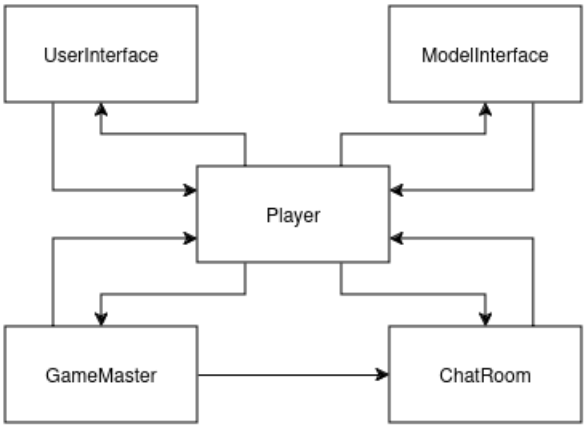

Figure 2: Aquantance model

2.5 Services

GameMaster

Service	Inputs	Outputs	Pre-condition	Post-condition
join game	*name*	*confirmation message*	the player has not yet connected to the game, and has chosen a valid (unique) name for themselves	the player is inserted into *playerData*
vote	*vote target*	*confirmation message*	the player is eligible to vote	the vote is stored in *playerData*, and the round is completed if enough votes have been cast
show score	*request*	*list of players and their scores*	a round of the game has completed	the details of the game are shared in the lobby

Table 8: Services model for the GameMaster

Player

Service	Inputs	Outputs	Pre-condition	Post-condition
assign role	*role*		notifies the player of their role	the player can use this in deciion making
kill			the player is alive	the player cannot participate for the rest of the round

Table 9: Service model for the Player

ModelInterface

Service	Inputs	Outputs	Pre-condition	Post-condition
query LLM	*prompt*	*response*	the player is a bot	a decision is made to progress the game (listen, speak, or vote)

Table 10: Service model for the ModelInterface

UserInterface

Service	Inputs	Outputs	Pre-condition	Post-condition
get next user input		*message*	the player is a human	the message will influence the game space

Table 11: Service model for the UserInterface

ChatRoom

Service	Inputs	Outputs	Pre-condition	Post-condition
open room	*list of players*	*confirmation message*	the room is closed	indicates start of phase, players may now send dialogue and commands
close room	*request*	*confirmation message*	the room is open	the room will no longer accept messages
send message	*message*		the room is open and the player is eligible to send messages	updates the *chatLog* with the message
retrieve unread messages	*index of last message received*	*list of unread messages*	the player is participating in the room	verifies the index of the last message that all players have received, and purges the *chatLog* of read messages

Table 12: Service model for the ChatRoom

3 Implementation Details

3.1 General

This software will use SPADE 3.3.3 [7] as the development environment for the multi agent system. The XMPP server is hosted locally using Ejabberd [8], while the LLM will be powered by Llama 3.1 via Ollama [9].

The following section contains information on all the classes implemented for this project, alongside message information passed between the agents.

3.1.1 Message Formatting

All inter-agent communication in the MAW system is conducted using JSON-formatted messages through SPADE message passing, which include specific fields to ensure consistency across the system. Each message type is associated with a particular interaction model as outlined in Section 2.2, and message fields are designed to be processed by the receiving agent's behaviors. They're associated with specific FIPA communicative acts according to FIPA SC00037 [10].

Field	Description
to	The JID of the recipient.
metadata	A (key, value) dictionary desribing the FIPA attributes of the message.
body	A JSON string containing the body of the message, which will allow the receiving agent to parse the intent of the message (for example, whether it's a vote, a chat, or a name request)

Table 13: Standard Fields for Messages

3.2 PlayerAgent

The central class for either the user or an AI-controlled player, corresponding to the Player agent in the agent model (see 1). This agent sends the same messages requesting user input regardless of whether or not the player is a human or an AI, only with different destinations for the messages.

Attribute	Description
$player\_interface$	JID of an agent of either a $UserInterface$ or $ModelInterface$.
name	Stores the name of the player, which must be custom chosen on initialization, by querying the $player\_interface$.
role	Assigned by the GameMasterAgent at the beginning of a round.
status	Either alive or dead, set to *alive* at the beginning of a round, then possibly set to *dead* at the end of a phase of the game.

Table 14: Attributes of PlayerAgent

3.2.1 Behaviours and Methods

Behaviour	Type	Description
$PlayerBehaviour$	FSMBehaviour	See the following table for states.
$SetupRole$	OneShotBehaviour	Receives the player's role from the GameMaster, and resets the status to *alive*.
$Kill$	OneShotBehaviour	When receiving a *kill* message from the GameMaster, influences the state of the FSM.
$PhaseChange$	OneShotBehaviour	Processes a notification from the GameMaster of a change in phase, along with the results.
$RoundEnd$	OneShotBehaviour	Processes a notification from the GameMaster that the current round of the game has ended.

Table 15: Behaviours of PlayerAgent

State	Description
GetNameState	Sends a message *get_input* to the *player_interface*, then receives a message *player_action* that holds the name of this player.
JoinLobbyState	Sends a *join_game* message to the *GameMasterAgent* until successful or a timeout.
LobbyState	Waits for a *start_game* message from the *GameMasterAgent*.
AwakeState	While *alive*, the agent participates in a gameplay phase where they send *get_new_messages* messages and forward *player_action* messages to the currently active *ChatRoomAgent*.
AsleepState	The player is in this state when *alive*, a *villager*, and the game is in the *night* phase. They cannot recieve or send messages to the *ChatRoomAgent*, and will wait for a *phase_change* or *kill* message from the *GameMasterAgent*.
DeadState	The player has received a *kill* command and can no longer send messages.

Table 16: States of PlayerAgent

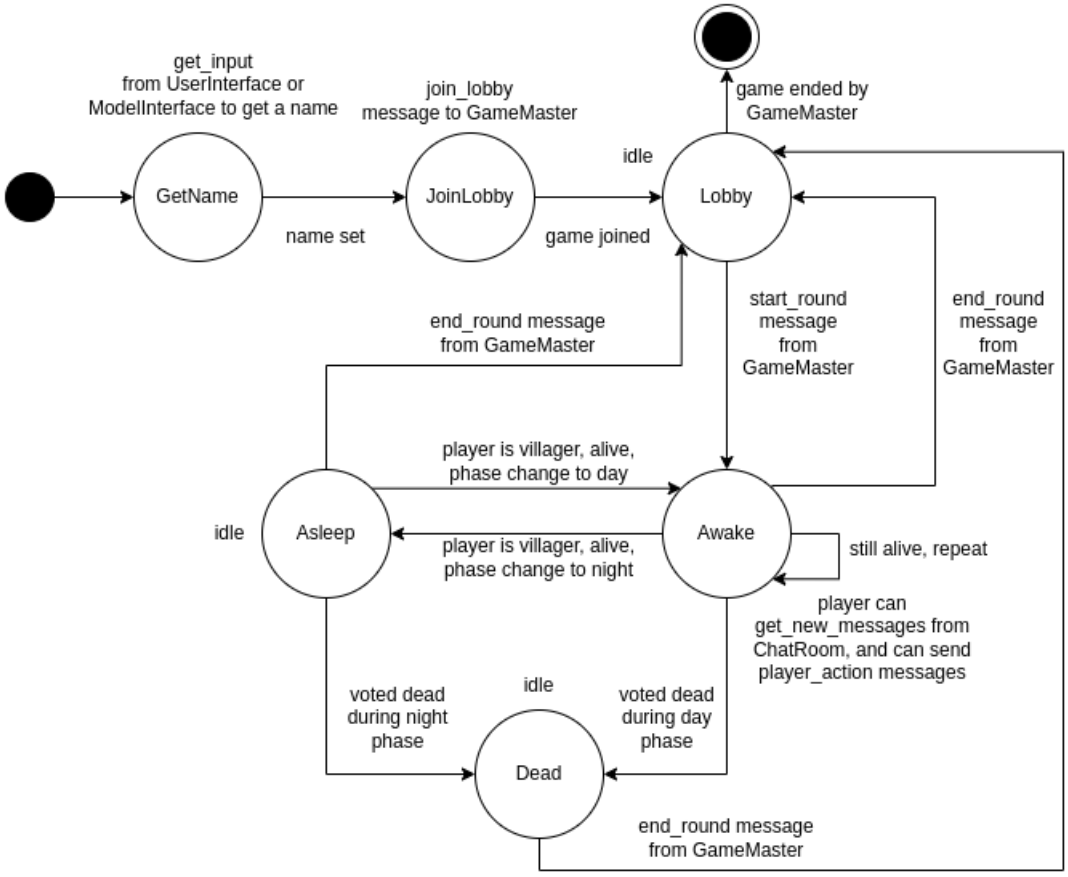

Figure 3: Finite State Machine of PlayerAgent

3.2.2 Messages

Join Game Message Sent by a PlayerAgent to the GameMasterAgent to request entry into a game lobby.

FIPA Attribute: *request*

```
{
"type": "join_lobby",
"player_name": "Player1" }
}
```

Get Input Message Sent by the PlayerAgent to the UserInterfaceAgent or ModelInterfaceAgent to prompt for the next action, which could be a dialogue message or a vote.

When sent to a ModelInterfaceAgent, the agent will be supplied with a chat completion which can then be forwarded to the LLM.

FIPA Attribute: *request*

```
{
"type": "get_input",
"completions": {
model: "llama3.1",
messages: [
{ role: "system", content: "<<GUIDING PROMPT AS PER ALGORITHM 1>>" },
{
role: "user",
content: "<<MESSAGES IN THE CHAT LOG>>",
}]}}
```

Player Action Message Sent by a PlayerAgent to the GameMasterAgent to cast a vote during a game phase.

FIPA Attribute: *request*

```
{
"type": "vote",
"target": "player2@localhost"
}

{
"type": "chat",
"message": "I think Player3 is guilty!"
}
```

3.3 UserInterfaceAgent

Represents the user's command line interface. Upon receiving a message of type *get_action*, it will read the next user input from the console, and send that as a *player_action* message back to the *PlayerAgent* class that requested it.

This class has no attributes, as for this version of the project, only the command line will be used, and as such Python's standard *input*() function will be used to read from the keyboard.

Behaviour	Type	Description
GetAction	CyclicBehaviour	Responds to a *get_action* message, by retrieving the user input and sending a *player_action* message to the requesting *PlayerAgent*.

Table 17: Behaviours of UserInterfaceAgent

3.4 ModelInterfaceAgent

The equivalent of a *UserInterfaceAgent*, but for an AI player. Receives the same *get_action* messages, but instead of prompting the user, will forward the prompts to an LLM.

Attribute	Description
model	An instance of an LLM class (see next class).

Table 18: Attributes of ModelInterfaceAgent

3.4.1 Behaviours and Methods

Behaviour	Type	Description
SendInput	CyclicBehaviour	Responds to a *get_action* message, by sending an API request to the LLM and forwarding the *player_action* message to the requesting *PlayerAgent*.

Table 19: Behaviours of ModelInterfaceAgent

3.4.2 Messages

Messages generated by the UserInterfaceAgent and ModelInterfaceAgent will be in the same format as Player Action Messages above.

3.5 LLM

This non-agent class stores the connection information and context for a given AI agent's LLM. For simple use, the messages sent to and from this agent can be used to create the context. However, time permitting, this class can be extended to optimize the context for low-powered devices. If an API key for chatGPT is available, it can be plugged into here for seamless use.

The LLM will be prompted to respond using JSON formatting, so that the messages can be passed to agents immediately without further processing.

Attribute	Description
base_url	Address of the LLM to query.
api_key	Required when using an online API, however can be left simply as "ollama" for use on localhost
model	The name of the model to use on the server.
context	The context stored for the individual agent, so that memories are not shared between different players, in the form of a set of chat completions.

Table 20: Attributes of LLM

3.5.1 Methods

Method	Description
prompt	Prompts the LLM for a response, then stores the resulting context.
save	Saves the current context to the disk as a text file, for reuse and logging purposes.
load	Can be optionally used to load an existing context.
$truncate_context$	A simple way to reduce the processing time of the LLM, however at the risk of causing amnesia.

Table 21: Methods of LLM

3.6 GameMasterAgent

The GameMasterAgent orchestrates the game flow by managing player connections, assigning roles, tracking the game state across phases, and declaring the winning team at the end of each game round. It serves as the central authority ensuring consistent gameplay.

Attribute	Description
num_players	Set on initialization, the number of players to expect in the game, so that the game can start when the threshold is reached. Default 6.
player_data	Holds information on all connected players, including names, roles, and status (alive or dead).
phase_timer	A timer controlling the duration of each phase (day, night).
current_phase	Tracks the active game phase and influences voting and communication permissions.
state_timer	The timer that will

Table 22: Attributes of GameMasterAgent

3.6.1 Behaviours and Methods

Behaviour	Type	Description
GameMasterBehaviour	FSMBehaviour	See the following table for states.

Table 23: Behaviour of GameMasterAgent

State	Description
SetupState	Connects to players.
AssignRolesState	Assigns roles to players.
DayPhaseState	Opens the day phase ChatRoomAgent, resets the timer, and notifies players of the phase change.
DayPhaseVoteCount	Upon the end of the day phase, counts votes and optionally kills a player.
NightPhaseState	Opens the night phase ChatRoomAgent, resets the timer, and notifies players of the phase change.
NightPhaseVoteCount	Upon the end of the night phase, counts votes and optionally kills a player.
EndPhaseState	Declares the game outcome based on the remaining players and sends a summary of the game to all players.

Table 24: States of GameMasterAgent

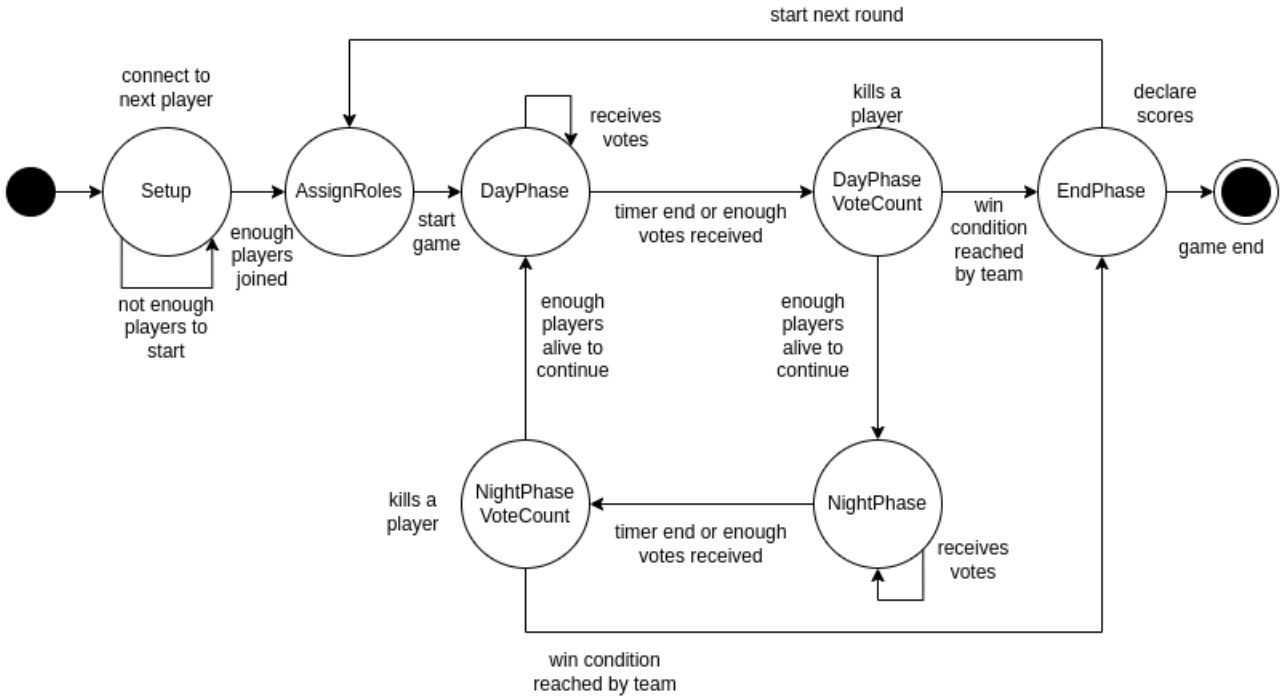

Figure 4: Finite State Machine of GameMasterAgent

3.6.2 Messages

Start Game Message Sent by the GameMasterAgent to all PlayerAgents to announce the beginning of a game round. This message includes the total number of players and signals the transition from the lobby to the game setup.

FIPA Attribute: *inform*

```
{
"type": "start_game",
"total_players": 6
}
```

Assign Role Message Sent by the GameMasterAgent to each PlayerAgent to assign their role (e.g., Villager, Werewolf) at the beginning of the game.

FIPA Attribute: *inform*

```
{
"type": "assign_role",
"role": "Werewolf"
}
```

Phase Change Message Sent by the GameMasterAgent to all PlayerAgents and ChatRoomAgents to indicate a transition between phases (e.g., from day to night). This message signals agents to adjust their permissions and state.

FIPA Attribute: *inform*

```
{
"type": "phase_change",
"new_phase": "night"
}
```

Vote Result Message Sent by the GameMasterAgent to all PlayerAgents and ChatRoomAgents to announce the result of a vote at the end of the day phase, specifying the player who has been executed.

FIPA Attribute: *inform*

```
{
"type": "vote_result",
"target": "player2@localhost",
"status": "executed"
}
```

End Game Message Sent by the GameMasterAgent to all PlayerAgents and ChatRoomAgents to declare the end of the game and announce the winning team.

FIPA Attribute: *inform*

```
{
"type": "end_game",
"winning_team": "Villagers"
}
```

3.7 ChatRoomAgent

The ChatRoomAgent manages communication within specific groups of players during each phase. It handles the message flow and voting within each chatroom (e.g., day and night chatrooms) and ensures phase-appropriate access control for players.

Attribute	Description
room_name	Specifies the name of the chatroom, indicating the phase (e.g., "Village" for day, "Hideout" for night).
player_data	Stores details of all players allowed in the room, including status (alive or dead), to make sure that any agent sending messages are allowed to do so.
chat_log	Maintains a record of all messages sent during the active phase, which can be retrieved on request.
round_number	Tracks the current round of the game for logging purposes.

Table 25: Attributes of ChatRoomAgent

3.7.1 Behaviours and Methods

Behaviour	Type	Description
ChatRoomBehaviour	FSMBehaviour	See the following table for states.

Table 26: Behaviour of ChatRoomAgent

State	Description
SetupState	Receives its name and initial status from the GameMasterAgent.
OpenState	Confirms each incoming message meets the chatroom requirements (e.g., only live players can send messages). Logs validated messages into the *chatLog*, ensuring messages can be retrieved later if needed.
CloseState	Ends message acceptance, prevents further communication, and clears and archives the *chat_log* at the end of the phase by writing to the disk.

Table 27: States of ChatRoomAgent

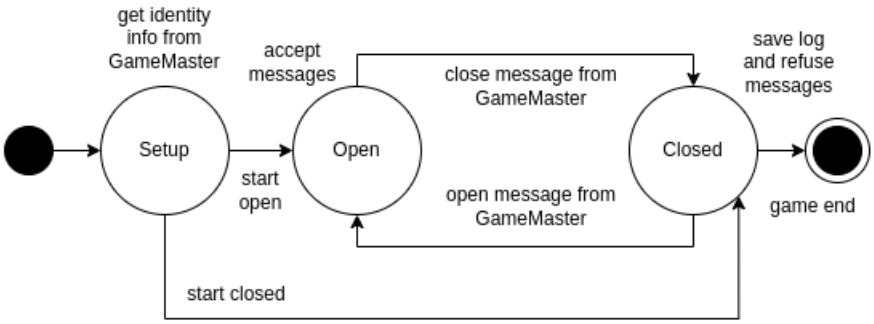

Figure 5: Finite State Machine of ChatRoomAgent

3.7.2 Messages

Announce Open Message Sent by the ChatRoomAgent to all PlayerAgents allowed in the room to notify them that the chatroom is now open for communication during the current phase.

FIPA Attribute: *inform*

```
{
"type": "announce_open",
"room_name": "Village",
"players": ["player1@localhost", "player2@localhost"]
}
```

Send New Messages Sent by a PlayerAgent to the ChatRoomAgent to request any unread messages from the chat log. The ChatRoomAgent responds with a list of messages the player has not yet seen.

FIPA Attribute: *inform*

```
{
"type": "new_messages",
"messages": ["Player1: I think Player3 is guilty.",
"Player4: I agree.",
"Player3: I'm innocent!"]
}
```

References

[1] D. Davidoff. (1998) The original mafia rules. [Online]. Available: http://web.archive.org/web/19990302082118/http://members.theglobe.com/mafia_rules/

[2] A. Plotkin, "Werewolf," 2010. [Online]. Available: https://www.eblong.com/zarf/werewolf.html

[3] (2024) Among us. [Online]. Available: https://www.innersloth.com/games/among-us/

[4] (2024) Artificial intelligence based werewolf. [Online]. Available: https://aiwolf.org/

[5] T. Wang and T. Kaneko, "Application of deep reinforcement learning in werewolf game agents," in *2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*. Taichung, Taiwan: IEEE, 2018, pp. 28–33.

[6] A. Javadpour, F. Ja'fari, T. Taleb, H. Ahmadi, and C. Benzaïd, "Cybersecurity fusion: Leveraging mafia game tactics and reinforcement learning for botnet detection," in *GLOBECOM 2023 - 2023 IEEE Global Communications Conference*. Kuala Lumpur, Malaysia: IEEE, 2023, pp. 6005–6011.

[7] "Smart python agent development environment," 2024. [Online]. Available: https://pypi.org/project/spade/

[8] "Ejabberd," 2024. [Online]. Available: https://www.ejabberd.im/index.html

[9] "llama3.1," 2024. [Online]. Available: https://ollama.com/library/llama3.1

[10] "Fipa acl message struture specification," 2002. [Online]. Available: http://www.fipa.org/specs/fipa00061/SC00061G.html