CS7650:Autonomous systems

Design and Implementation a Multiagent System

Submitted by:

Avanti Mittal - B22ES021
arohi Dharmadhikarii -B22AI001
Khushi Bhardwaj - B22AI026
Siramsetty Indusri - B22AI039

Submitted to:

Prof. [Gourav Harit]

September 28, 2025

Contents

Abstract

This report presents the design and implementation of a multi-agent system (MAS) to solve a dynamic resource-sharing problem. The system models a real-world scenario where students exiting multiple classrooms create congestion at a shared bottleneck. We have developed a custom, rule-based framework in Python where autonomous agents, representing each classroom, negotiate to deconflict their exit schedules. The agents employ a sophisticated, utility-driven decision-making process, engage in multi-step negotiations involving counter-offers, and utilize a trust and reputation model to penalize uncooperative behavior. The simulation and the accompanying .interactive web-based demonstration demonstrate that this autonomous system can consistently and effectively resolve congestion, adapting its strategy based on agent personalities and a persistent memory of inter-agent commitments.

1 Introduction

1.1 Problem Statement

Coordinating access to a shared and limited resource is a fundamental challenge in distributed systems. When multiple autonomous entities require access simultaneously, the demand can exceed the resource's capacity, leading to system-wide failure or congestion. Our project models this challenge in the context of pedestrian traffic flow outside a lecture hall complex. If all lectures conclude at the same time, the sudden influx of 290 students onto a narrow road with a capacity of 80 students per 2-minute slot will cause a significant jam. The goal is to design a system that can manage this recurring problem autonomously.

1.2 Chosen Approach: A Custom MAS Framework

To address this, we designed a decentralized multi-agent system where each classroom is represented by an autonomous agent. While the assignment suggested frameworks like AutoGen, we made a deliberate design choice to build a custom, procedural simulation in Python. This approach provided us with direct and precise control over the agents' strategic logic, state management, and the enforcement of the negotiation protocol. It allowed us to finely tune the heuristics, such as the utility functions and reputation model, to directly model the specific resource-coordination problem defined in the assignment.

2 System Design and Agent Architecture

2.1 Agent Roles

Our system comprises two types of agents:

- **Bottleneck Agent (Agent B):** A singular, observational agent that monitors the overall state of the shared resource (the road). At the start of each simulation episode, it broadcasts the initial congestion state to all classroom agents, initiating the negotiation process.

- **Classroom Agents (C1, C2,...):** Each classroom is represented by a Classroom Agent. These are the primary actors in the system, designed with a set of beliefs, desires, and intentions (BDI) to guide their autonomous behavior.

2.2 Classroom Agent Architecture (BDI Model)

We designed each classroom agent using the principles of the Beliefs, Desires, Intentions (BDI) architecture.

- **Beliefs:** The agent's knowledge of the world. This includes its own attendance, the bottleneck capacity, the current schedule of all agents ('slot_map'), and a record of other agents' reputations.
- **Desires:** The agent's high-level goals. These include (1) ensuring its students exit without congestion, (2) fulfilling its commitments to other agents, and (3) acting in accordance with its assigned 'personality'.
- **Intentions:** The agent's chosen course of action. This represents the agent's commitment to a course of action, such as executing a propose$_{s}hiftorformulate_{c}ounter_{o}fferbasedonitscurrentbeliefsan$

3 Core Agent Intelligence & Negotiation Protocol

3.1 Agent Personalities and Utility Functions

To model the assignment's constraint that professors may have different preferences, each agent is randomly assigned a 'personality' (`prefers_early`, `prefers_late`, or `flexible`). Decisions are then made not by chance, but by a **utility function** that calculates a rational score for any given offer. An agent accepts an offer only if its utility exceeds a minimum threshold. The utility U for an offer is modeled as:

$$U(\text{offer}) = w_1 \cdot \text{PersonalityMatch} + w_2 \cdot \text{CommitmentGain}$$

Where PersonalityMatch is positive if the offer aligns with the agent's preference and CommitmentGain is positive if the deal results in another agent owing a future favor. This ensures agents are rational and act in their own self-interest.

3.2 Strategic Negotiation and Counter-Offers

The negotiation protocol is designed to be efficient and realistic.

- **Strategic Partner Selection:** The agent with the most students in a congested slot initiates the negotiation, targeting the agent with the second-most students. This heuristic ensures the most impactful agents negotiate first.

- **Dynamic Proposals:** Agents survey the environment (‘slot$_{m}ap$‘$)andproposeshiftstotheleastcongestedava$

- **Rational Counter-Offers:** If an offer is rejected due to low utility, the receiving agent formulates a **counter-offer** that aligns with its own personality and utility. This creates a dynamic, multi-step negotiation that is highly effective at finding mutually acceptable solutions.

3.3 Trust and Reputation Model

To ensure long-term cooperation, the system implements a memory of past actions.

- **Commitments:** When a deal is made, a ‘Commitment‘ object is created and stored in a global ledger.
- **Violations:** If an agent fails to fulfill a due commitment (as demonstrated by our ”stubborn” agent, C4), a ‘miss‘ is recorded. If the number of misses exceeds the ‘violation$_{t}hreshold$‘, $aformal*$ $*violationevent * *istriggered.$
- **Reputation Penalty:** Upon a violation, the offending agent’s ‘reputation‘ score is penalized. Other agents may then refuse to negotiate with agents that have a low reputation, effectively isolating uncooperative members.

4 Simulation and Results

4.1 Simulation Philosophy: The Recurring Problem

A key design choice of our simulation is the episodic nature of the problem. The **physical problem is reset from scratch** at the start of every episode, while the **agents’ social memory is persistent**. At the beginning of each episode, the schedule is reset to the worst-case scenario where all 290 students are in the ‘t=0‘ slot, as evidenced by the log entry ‘[Initial slot map] 0: 290‘. This design is crucial for three reasons:

1. **To Test Robustness:** It tests the agents’ negotiation *process*, proving they can solve the problem autonomously every time, not just follow a single pre-computed schedule.
2. **To Model a Dynamic World:** A static schedule is fragile. Resetting the problem simulates a real-world scenario where attendance or professor preferences might change weekly.
3. **To Make Social Memory Matter:** The recurring problem forces agents to use their persistent memory of commitments and reputations. Fulfilling a promise from a previous episode becomes a meaningful action taken to navigate the challenges of the current episode.

4.2 Analysis of Negotiation Results

The simulation log and visualizations provide clear evidence of the system's effectiveness. In Episode 1 (Figure 1), a complex negotiation unfolds, including a successful counter-offer from C6 to C4.

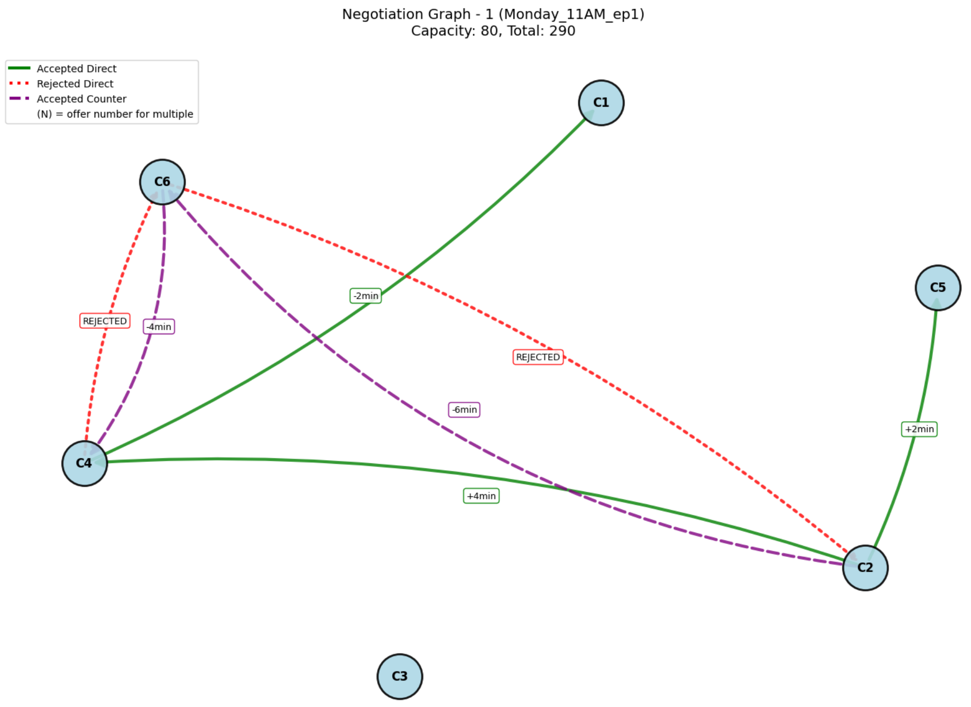

Figure 1: Negotiation graph for Episode 1, showing accepted (green), rejected (red), and counter-offered (purple) deals.

The result is a successfully de-conflicted schedule (Figure 2), transforming the initial congestion into a manageable distribution.

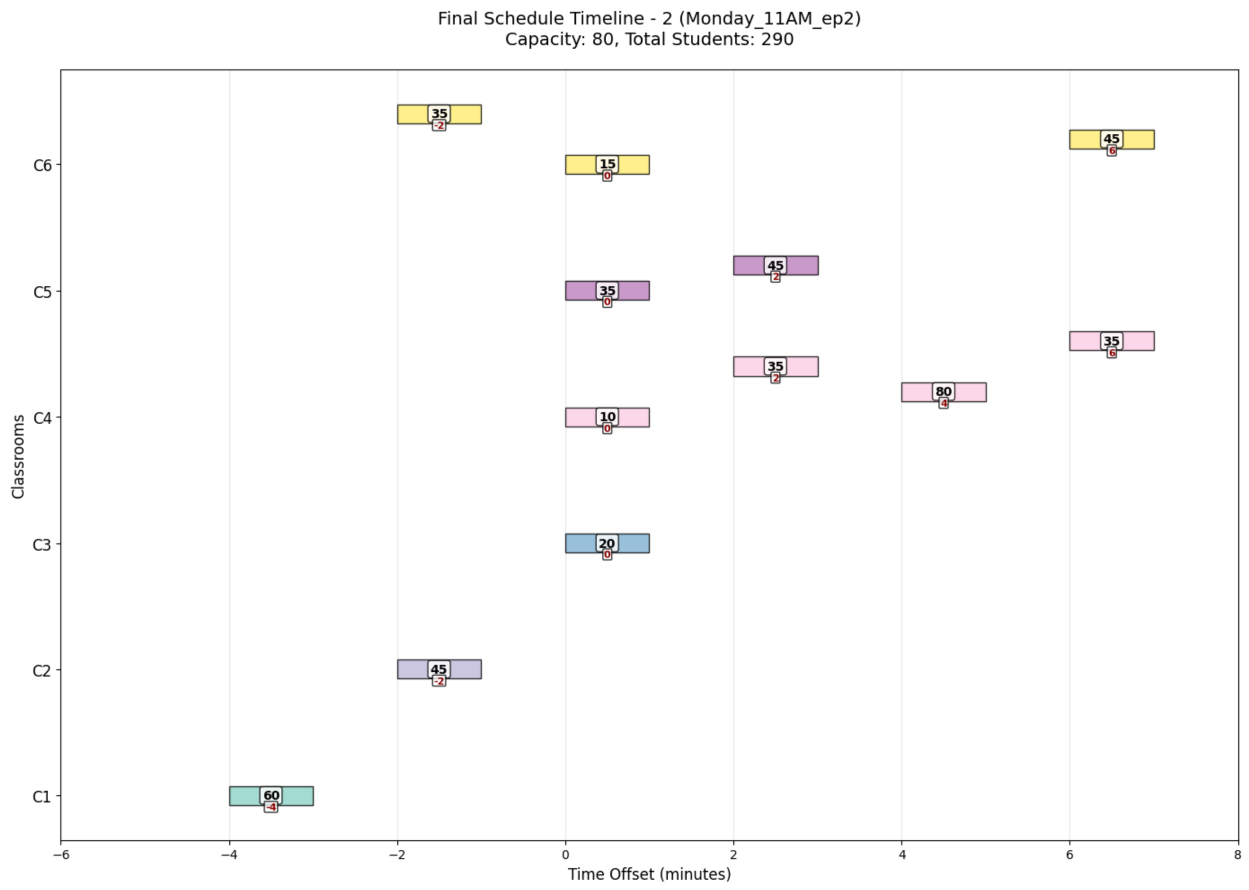

Figure 2: Final balanced schedule after the negotiations in Episode 1.

4.3 Demonstration of the Reputation System

Agent C4 was configured to be ”stubborn”. The system successfully identified this behavior. In Episode 2, C4 fails its commitment and is penalized:

```
[FULFILL FAILED] C4 couldn't fulfill com_offer_C4_to_C1_ep1 (missed 1)
[VIOLATION] C4 exceeded violation threshold ... Reputation penalized.
```

Over subsequent episodes (e.g., Episode 4, Figure 3), C4 accumulates multiple violations, cementing its status as an unreliable agent.

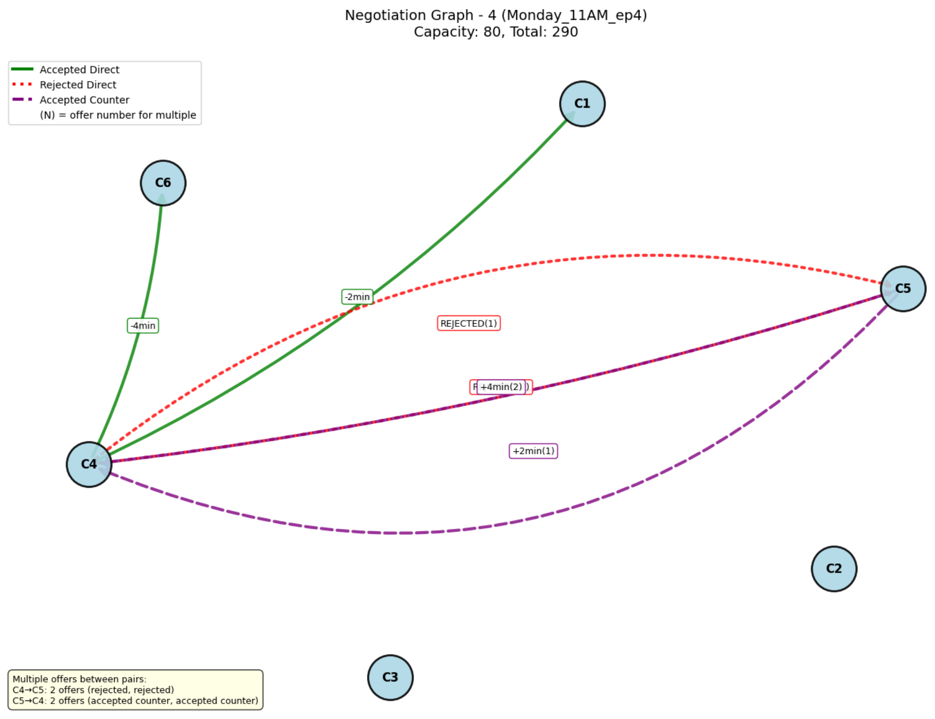
Negotiation Graph - 4 (Monday_11AM_ep4)
Capacity: 80, Total: 290
Accepted Direct
Rejected Direct
Accepted Counter
(N) = offer number for multiple
C1
C6
C5
C4
C2
C3
-4min
-2min
REJECTED(1)
+4min(2)
+2min(1)
Multiple offers between pairs:
C4→C5: 2 offers (rejected, rejected)
C5→C4: 2 offers (accepted counter, accepted counter)

Figure 3: Negotiation graph for Episode 4, where C4 continues to negotiate despite its failing reputation.

5 Conclusion

This project resulted in the successful implementation of a decentralized, autonomous multi-agent system in solving a complex resource-sharing problem. By equipping agents with intelligent, utility-driven behaviors like strategic negotiation, counter-offers, and a trust and reputation model, we created a robust system that can consistently cooperate to achieve a global goal. The simulation results and interactive demonstration confirm that this approach is highly effective at managing congestion and adapting to the behaviors of its constituent agents.

Group Member Contributions

- **B22AI026**
  Lead Developer & Visualization Engineer. Initiated the project by setting up the initial file structure and basic agent framework. Developed the comprehensive visualization module for creating negotiation graphs and schedule timelines. Integrated the final counter-offer logic and its corresponding visualizations.

- **B22AI039**
  Lead Strategist & Intelligence Designer. Implemented the core intelligence of the agents. This included upgrading the negotiation to a strategic model with heuristics (smarter agent selection, dynamic proposals) and designing and implementing the entire trust and reputation system (violations, reputation scores, and social sanctions).

- **B22ES021**
  Core Logic Developer. Worked on the foundational logic of the simulation. This involved developing and updating the core agent classes, implementing the initial state management, and building the basic simulation loop and commitment-handling logic upon which the more advanced features were later built.

- **B22AI001**
  Demonstration Lead & System Integrator. Focused on the final presentation and validation of the system. Developed the interactive 'demo.py' application, integrated the core simulation logic into the web server, and was responsible for ensuring all components worked together as a cohesive whole for the final demonstration.

6 Interactive Demonstration ('demo.py')

To supplement the static simulation, an interactive web-based demonstration was developed using Flask and Socket.IO. This 'demo.py' application allows a user to:

- Start, stop, and reset the multi-episode simulation in real-time.
- Observe the agent interaction logs as they occur.
- View live-updating charts that visualize the traffic distribution, classroom schedules, and commitment status.

This provides a dynamic and engaging way to observe the autonomous agents in action, making the complex negotiation processes easy to follow and understand.

A screen recording of this interactive demonstration running through a complete 5-episode simulation has been submitted separately as part of the project deliverables.