Project 13A: Decision Making in Robot Swarm via Majority Voting

Team Members: Dmitriy Lukiyanov, Ramil Shakirzyanov, Asgat Keruly, Yusuf Abdughafforzoda, Anvar Gilmiev

Introduction

Majority voting is a fundamental distributed consensus approach where a collective decision is reached based on the option receiving the most support among the agents. Each agent proposes an option, shares it with others, collects proposals from the swarm, and independently determines the winning option based on the majority count. This approach enhances robustness, as the system doesn't rely on a single point of failure.

This project implements a simulation of a robot swarm performing distributed decision-making using a majority voting protocol. The robots communicate asynchronously via a message broker (RabbitMQ) to share their proposals and converge on a single decision from a predefined set of options ("Go Left", "Go Right", "Stay Put", "Go Forward"). The objective is to demonstrate a functional decentralized decision system and validate its correctness, handling of ties, and basic performance characteristics. This simulation is relevant for understanding coordination patterns in multi-agent systems and distributed computing protocols.

Methods

System Architecture

The system consists of two main components:

1. **Robot Agents:** Independent Python processes (`robot.py`), each representing a single robot in the swarm. Each robot is initialized with a unique ID, a specific proposal, and the total expected size of the swarm.
2. **Message Broker:** A RabbitMQ server (run via Docker Compose) facilitates communication between the robot agents.

Communication relies on a `fanout` exchange (`robot_exchange`) in RabbitMQ. This ensures that messages published to the exchange are broadcast to all connected robots. Each robot creates its own exclusive, temporary queue bound to this exchange, allowing it to receive all broadcast messages without interfering with others.

The decision-making process follows these steps:

1. **Initialization:** Each robot process starts, connecting to RabbitMQ and declaring the exchange and its unique queue.
2. **Synchronization (Readiness Check):** Before exchanging proposals, robots ensure all participants are ready. Each robot broadcasts a `{"type": "ready", "robot_id": "..."}` message. Robots listen for these messages until they have received one from every expected member of the swarm (matching the `--swarm-size`). This prevents premature proposal exchange.
3. **Proposal Exchange:** Once all robots are ready, each robot broadcasts its own proposal in a `{"type": "proposal", "robot_id": "...", "proposal": "..."}` message. Messages are marked persistent for basic reliability, although the primary handling is in-memory.
4. **Proposal Collection:** Each robot consumes messages from its queue, collecting incoming proposals until it has received exactly `swarm_size` proposals.
5. **Decision Calculation:** Upon receiving all proposals, each robot independently calculates the result:
    - It counts the occurrences of each unique proposal using `collections.Counter` .
    - It identifies the proposals with the maximum vote count.
    - **Tie-Breaking:** If multiple proposals share the highest vote count (a tie), the proposals are sorted alphabetically, and the first one in the sorted list

is chosen as the final decision. This provides a deterministic outcome.

6. **Logging:** The robot logs its final decision and the time taken for the proposal exchange and decision phase (from starting consumption after readiness to calculating the final result) to a shared `results.csv` file. Console output provides verbose logging of received messages, vote counts, and the final decision.

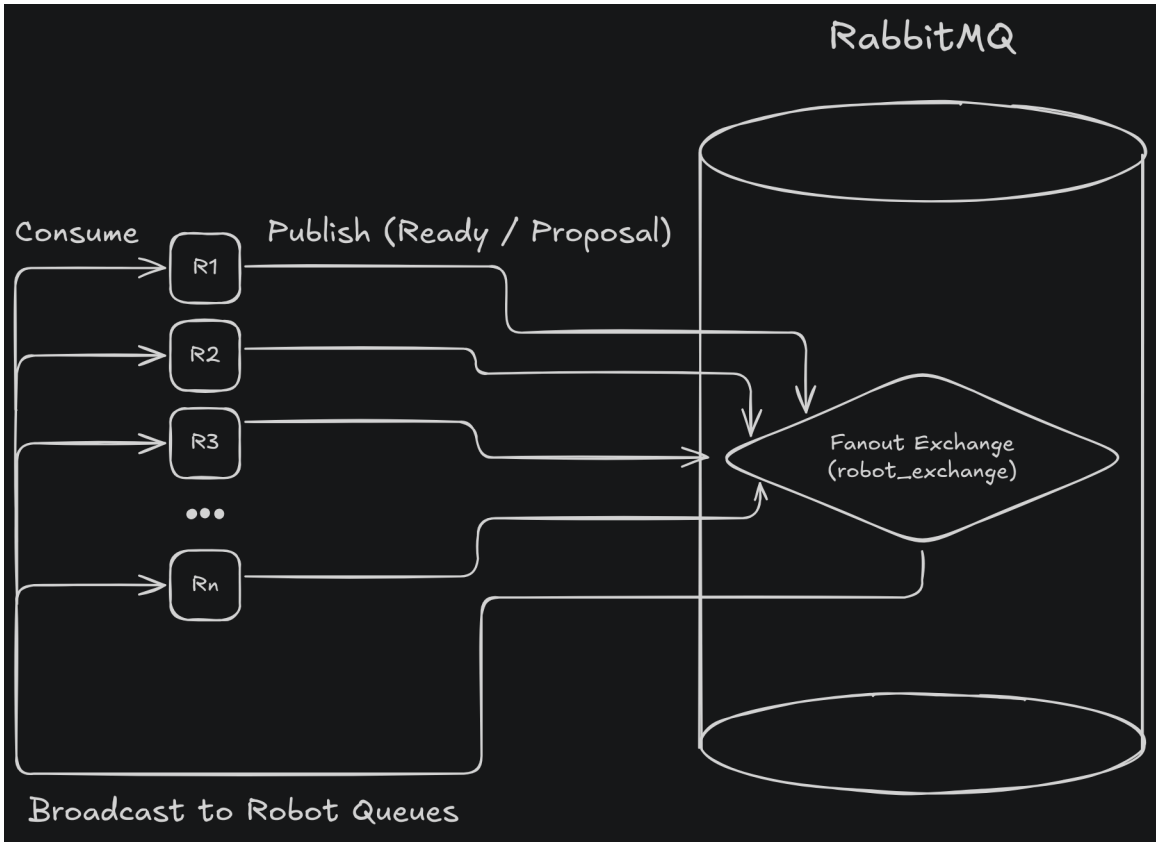


Diagram Description: The diagram above illustrates the system architecture. Robot agents (Python processes) communicate indirectly via a RabbitMQ message broker. Each robot publishes "ready" and "proposal" messages to a central fanout exchange. The exchange broadcasts these messages to temporary, exclusive queues, one for each robot. Robots consume messages from their respective queues to gather readiness signals and proposals from the entire swarm, enabling decentralized decision-making.

Implementation Details

- **Language:** Python 3

- **Libraries:**
  - `pika`: For RabbitMQ communication.
  - `json`: For serializing/deserializing messages.
  - `argparse`: For handling command-line arguments (robot ID, proposal, swarm size).
  - `uuid`: For generating default unique robot IDs.
  - `random`: For assigning a random default proposal if none is specified.
  - `collections.Counter`: For efficient vote counting.
  - `time`: For performance measurement (convergence time).
  - `os`: For path manipulation (results directory).
- **Environment:** The RabbitMQ broker is managed using `docker-compose`. Robot processes are run directly on the host machine, interacting with the containerized broker.
- **Configuration:** Key parameters like RabbitMQ host (`RMQ_HOST`), credentials (`RMQ_USER`, `RMQ_PASS`), exchange name (`EXCHANGE_NAME`), possible proposals list, and results directory are managed by a `constants.py` file. Swarm size and individual proposals are passed via command-line arguments.
- **Variant Implemented:** This implementation corresponds to **Variant A (One-time vote with preloaded proposals across agents)**. Each robot starts with a fixed proposal and participates in a single round of voting.

Setup and Execution

The system is set up using the provided `README.md` instructions:

1. Initialize a Python virtual environment (`.venv`).
2. Install dependencies from `requirements.txt`.
3. Start the RabbitMQ container using `docker-compose up -d`.
4. Launch the swarm using the provided shell script (`launch_swarm.sh` or `.bat`) or manually run individual `robot.py` processes, ensuring the `--swarm-size` argument matches the total number of robots being launched.

Results

The implemented system successfully simulates the distributed decision-making process within the robot swarm. Multiple robot instances were launched, communicating via RabbitMQ to achieve consensus based on majority voting.

System Functionality

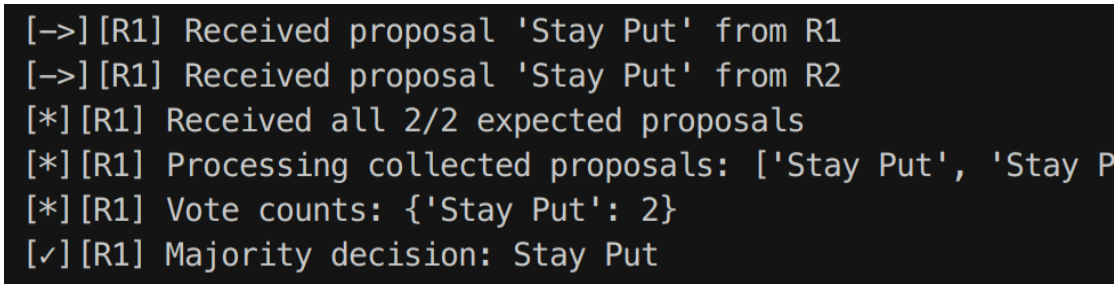
1. **Log decision messages and proposal counts per round:**
   - The console output of each robot clearly shows the reception of proposals from peers, the calculated vote counts for each proposal, and the final determined decision, including any tie-breaking logic applied.
   - The final decision for each robot, along with its ID, is logged persistently in `results.csv`.
   - 

     Figure 1: Console output from a single robot showing received proposals, the aggregated vote counts, and the final computed decision for a swarm of size 2.
2. **Simulate tie votes and show decision fallback:**
   - Ties were simulated by manually launching robots with proposals designed to create a tie and also observed naturally during runs with random proposals.
   - The system correctly identifies tie situations and applies the deterministic fallback mechanism (alphabetical sort and select first). The console output explicitly logs when a tie is detected and the resulting final decision.

```
[->][R1] Received proposal 'Stay Put' from R1
[->][R1] Received proposal 'Go Right' from R2
[*][R1] Received all 2/2 expected proposals
[*][R1] Processing collected proposals: ['Stay Put', 'Go Right']
[*][R1] Vote counts: {'Stay Put': 1, 'Go Right': 1}
[*][R1] Tie detected between: ['Go Right', 'Stay Put'], applying fallback...
[✓][R1] Final decision after tie-break: Go Right
```

Figure 2: Console output demonstrating the detection of a tie between "Stay Put" and "Go Right" (with a swarm size of 2) and the application of the **alphabetical tie-breaking rule** resulting in "Go Right".

3. **Track time-to-convergence across multiple swarm sizes:**
   - The time elapsed between a robot being ready to process proposals (after synchronization) and determining the final decision is measured (`end_time - start_time`). This represents the core convergence time for proposal exchange and voting logic.

Swarm Size (N)	Average Convergence Time (ms)
3	1.07
5	1.34
10	8.19
20	46.64
30	76.16
40	99.32

Table 1: Average convergence time (proposal exchange and decision phase) recorded for different swarm sizes.

results > results.csv > data

```
1   R2,Stay Put,0.0192
2   R4,Stay Put,0.0056
3   R9,Stay Put,0.0064
4   R3,Stay Put,0.0184
5   R7,Stay Put,0.0201
6   R8,Stay Put,0.0222
7   R10,Stay Put,0.0068
8   R1,Stay Put,0.0146
9   R5,Stay Put,0.0137
10  R6,Stay Put,0.0187
```

Figure 3: Sample content of the `results.csv` file generated after a

simulation run, showing the recorded final decision and convergence time for each participating robot.

Performance Observations

The simulation runs effectively for the tested swarm sizes (N=3 to N=40). Performance was measured by tracking the average convergence time, defined as the duration from the completion of the readiness synchronization to the calculation of the final decision. As shown in Table 1, the average convergence time consistently increases with the swarm size. This is expected, as each robot must process proposals from all N peers, and the communication overhead via the message broker also grows with N.

Discussion

The implemented system successfully demonstrates a distributed majority voting mechanism for a simulated robot swarm using RabbitMQ. The results confirm that the agents can collectively reach a decision based on the most frequent proposal without a central coordinator, fulfilling the core project requirement. The validation checklist items were met: decision messages and counts are logged, tie situations are handled deterministically, and convergence time is tracked and can be analyzed across different swarm sizes.

The observed increase in convergence time with swarm size (Table 1) is expected in a system where each node must receive information from all other nodes. While RabbitMQ handles the broadcast efficiently, the processing load on each client increases linearly with N (number of proposals to receive and process). The initial synchronization phase also contributes to the overall time but is crucial for the correctness of this specific one-shot voting protocol.

Challenges Faced:

- **Synchronization:** Ensuring all robots start the proposal exchange phase simultaneously (or at least only after all are ready) was critical. The implemented readiness check addresses this for the start, but assumes

robots don't fail mid-execution. Real-world network latency could also introduce minor timing variations.

- **Debugging:** Debugging distributed behavior across multiple processes required careful logging and observation of RabbitMQ message queues to trace message flow.
- **Scalability:** While functional, the approach of every robot receiving every message might face limitations at very large scales. The console output also becomes much less readable and the distributed synchronization mechanism becomes less stable.

Limitations:

- **Fault Tolerance:** The current implementation lacks robustness against robot failures or significant network issues. If a robot crashes before sending its proposal, the others might wait indefinitely. There's no mechanism for robots to detect and handle peer failures during the voting round.
- **Network Model:** The simulation relies on a reliable local network connection to the RabbitMQ broker. Real-world swarm communication would introduce latency, loss, and intermittent connectivity not modeled here.
- **Tie-Breaking:** The alphabetical tie-breaking is simple and deterministic but arbitrary. In a real application, a more meaningful tie-breaking strategy might be preferred (e.g., fallback to a default "safe" state or triggering a new round).

Potential Improvements/Optimizations:

- **Fault Tolerance:** Implement heartbeats or timeouts to detect non-responsive robots and potentially proceed with a decision based on available votes after a certain period.
- **Scalability:** For larger swarms, alternative communication patterns (e.g., gossip protocols) or hierarchical structures (e.g., small groups with leaders) could reduce the communication overhead on individual agents and increase overall reliability.
- **Metrics:** Expand logging to include message counts per robot or other relevant performance metrics. Maybe make the logging robot-specific for the inspection to be more manageable.

In conclusion, this project provides a functional baseline for distributed majority voting in a simulated robot swarm, successfully meeting the specified requirements and validation criteria using Python and RabbitMQ. While limitations exist regarding fault tolerance and scalability, it serves as a valuable educational implementation of decentralized coordination.

References

- RabbitMQ Documentation: https://www.rabbitmq.com/docs
- Pika Python Client Documentation: https://pika.readthedocs.io/en/stable/
- IU DNP Course: https://moodle.innopolis.university/course/view.php?id=3062