# Implementation of a Decentralized Cloning Controller for Heterogeneous Mobile Agents

***An M.Tech Project Report Submitted in Fulfillment of the Requirements for the Degree of***

**Master of Technology**

*by*

**Harsh Bijwe**
(214101020)

*under the guidance of*

**Prof. Shivashankar B. Nair**

**to the**

**COMPUTER SCIENCE AND ENGINEERING PROGRAMME**

**INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI**
**GUWAHATI - 781039, ASSAM**

# CERTIFICATE

*This is to certify that the work contained in this thesis entitled “**Implementation of a Decentralized Cloning Controller for Heterogeneous Mobile Agents**” is a bonafide work of **Harsh Bijwe (Roll No. 214101020)**, carried out in the Computer Science Programme, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Prof. Shivashankar B. Nair**

November, 2023

Department of Computer Science and Engineering

IIT Guwahati, Assam

# Abstract

The performance of systems connected in a network that relies on mobile agents for code and data transfer, and providing services to the requesting nodes, can be substantially enhanced through cloning. Uncontrolled cloning can cause an excessive number of agents to be generated, which could result in network congestion. This congestion may prevent on-demand agents from reaching their destination, causing delays in servicing requests. This report describes an implementation of **Cloning Controller Mechanism** stated in [1], which prescribes a population control mechanism for heterogeneous mobile agents that operates on demand. The model's effectiveness is ensured through a population control mechanism with heterogeneous mobile agents, which involves stigmergic sensing and estimation of network conditions within a platform. This mechanism allows the mobile agents to control their cloning rates using various equations, as explained in later sections.

# Contents

# Chapter 1

# Introduction

Mobile agents are the entities that carry both execution state and data as they migrate through a network. They can be quite useful in a multi-node environment for transferring information in the form of code, payload, and services to the desired node by traveling to that node and executing them. These agents visit the nodes in the network in some manner defined either implicitly or explicitly and, if needed, may service the nodes.

In a decentralized setting, Mobile Agents are useful entities. The code and data carried by the Mobile Agents may or may not be useful at some nodes. When working with a system of these agents which will roam around in a network, servicing the node's requests, it is important to determine the optimal placement of the agents and the order in which they will service the nodes. For instance, relying on a single agent to service the entire node's requests is not practical because it may take too long for agents to move between nodes that are far apart and require the same type of agent. In addition, there needs to be a mechanism in place to ensure that agents reach the desired node as soon as possible rather than visiting random nodes that may not have generated the request.

The most effective solution to the above problem is **Cloning**. Now that there are multiple copies of the same code moving around in the network, which will eventually serve more requests, the second question is the order of visits. Some of the earlier works like [2] is based on the assumption that once visited, the platform will not generate requests for the same service again. Also, active patrolling mechanisms published in [3] do not take care of the on-demand service request. The paper [4] talks about Active Patrolling, in which already visited nodes might generate the request for the same agent again and needs to be serviced.

Also, note that there can be multiple such agents who are carrying different types of services. We call them **Heterogeneous agents**. The platforms can generate requests for any of the agent types. We may further use the term node and platform interchangeably.

## 1.1 Tartarus

***Tartarus*** [5], is a multi-agent platform based on *SWI-Prolog*[6]. It has the capability to create a network of nodes or platforms. Each of the platforms or nodes can host Mobile Agents. They can be programmed to do certain tasks.

Tartarus can run on Windows, Ubuntu, and Raspbian OS. Mobility and Cloning is an essential features of Tartarus. Cloning can help maintain the degree of parallelism in the network, which in turn helps improve efficiency.

The cloning of Mobile Agents must not come at the cost of compromising the bandwidth of the network and utilizing more resources when a lesser number of agents and resources can do the same job. So there is a need for an inherent mechanism that can deliver the solution to this problem. An optimal solution has to be such that parallelism is achieved since parallelism will help service requests faster and resources are utilized minimally.

## 1.2 Cloning Controller

Agents whose services are frequently requested should be cloned in proportion to the demand to ensure timely service delivery. However, excessive cloning can result in network congestion, which may increase migration times and degrade the system's performance. Hence, the replication of mobile agents must be performed based on the self-contained network based on circumstances and requirements. In the event of network congestion, mobile agents are expected to die and release the resources so as to make space for the agents whose services are more relevant.

The decision to clone or not is made individually by each mobile agent based on network conditions and demand. While more clones in the network can lead to decreased service times, this improvement is not indefinite since excessive clones can congest the network, leading to performance degradation. Additionally, agents who are required more may end up stagnant at other platforms due to congestion.

The following graph from Cloning Controller Journal [1] shows what can happen if we perform unbounded cloning.

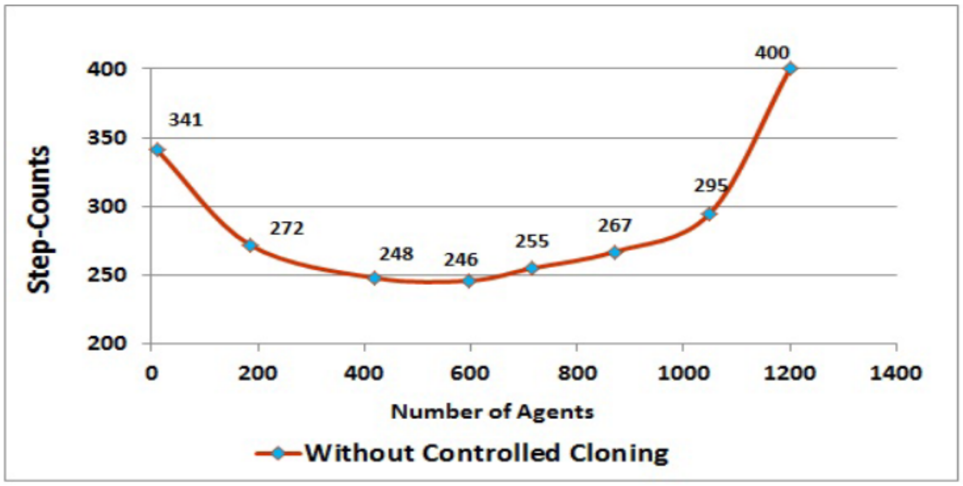


**Fig. 1.1**: Source: Cloning Controller Journal [1]

The graph presented in Figure(1.1) illustrates the relationship between the number of Step-Counts (time) required to service 100 platforms and the population of agents (including their clones). Initially, as the number of agents increases, the Step-Counts decrease, indicating improved efficiency and faster service. However, beyond a certain threshold, the Step-Counts start to increase due to network cluttering and congestion. This observation highlights the impact of network overcrowding on the performance of the system.

## 1.3 Pheromone-Conscientious Strategy

To improve this active patrolling approach, the Pheromone-Conscientious or *PherCon* Approach has been introduced. This approach employs the use of pheromones and a conscientious strategy to assist agents in migration toward the target nodes as quickly as possible. Each robotic node in the network is equipped with the ability to diffuse virtual pheromones, which attract suitable mobile agents to the robotic nodes requesting a service (RRS). When an agent detects a pheromone trail, it follows the trail to reach the node and provide the necessary service. Otherwise, if the agent does not stumble upon a pheromone trail at any network node, it follows a Conscientious strategy in which it avoids visiting the recently visited nodes.

In conclusion, Heterogeneous mobile agents perform conscientious patrols of the network, avoiding visits to nodes that were recently visited. When a pheromone trail is detected, the agents follow it to reach the destination node via the shortest possible path. It is a sort-of bi-directional search approach where Mobile Agents look for pheromone trails, and Pheromones are diffused in a network to guide Mobile Agents.

## 1.4 Organization of The Report

In this introductory chapter, we presented an overview of the topics covered in this report. We introduced the concept of Mobile Agents and explained why cloning is essential in a multi-node environment. Additionally, we provided a brief introduction to the key factors that must be considered when developing a viable model for controlling the population of clones and discussed their significance. We also mentioned Tartarus, which will be used for building this feasible model, and why it requires an inherent cloning controller mechanism.

Moving forward to Chapter 3, we will delve into our implementation of a feasible model for controlling the population of clones and discuss the details of experimentation and results of the implemented model. Finally, we conclude this report by highlighting potential areas for future work.

# Chapter 2

# Related Work

## 2.1 Related works on the importance of Cloning

Various approaches have been proposed to reduce service time and improve response times in satisfying requests. One such approach is the use of mobile agents, which has been found to be more effective than traditional client-server-based architectures, as discussed in [7]; increasing the number of agents has also been shown to reduce response time further and enhance system performance. Another approach is the use of grid computing-based routing models using mobile agents, as proposed in [8]. Mobile agents have also been successfully utilized in the domain of cloud computing, as highlighted in [9]. By incorporating cloning in these models, their performance can be further improved.

## 2.2 Why controlling the population of Agents is necessary?

Creating multiple copies of an agent can lead to decreased error and failure rates, as another agent of the same type can execute the job. Additionally, having more copies of the same code allows for handling multiple requests. However, this approach also has its drawbacks. The increased population of mobile agents can occupy more network resources, such as bandwidth, leading to slower service times. Furthermore, insignificant agents may take up valuable network bandwidth, leaving little space for more important agents to move around.

## 2.3 Related Works on controlling population of Agents

According to [10], controlling the population of mobile agents in dynamic networks can be achieved by using the amount of food an agent receives to determine the rate of cloning. The food is generated at each node in a span of regular intervals, and if an agent does not receive enough food, it starves and dies. Other approaches, such as the probabilistic algorithm described by [11] and [12], involve each node having a copy of an agent and cloning occurring with a certain probability 'p.' In contrast, the agents use a random migration policy to travel in the network. [13] suggests a biologically-inspired, homogeneous mobile agent-based approach using pheromones, where the frequency of visits to a particular node is used to calculate an inter-arrival time, and a larger value implies that there are fewer agents, meaning that cloning is needed. In comparison, a value greater than a specific threshold indicates that agents need to be purged. Similarly, [14] also proposes a similar approach for controlling the population of mobile agents. There may be communication between mobile agents, causing overheads on the network.

## 2.4 Controlling Population of Heterogeneous Mobile Agents

Our model allows each mobile agent to control its own cloning, eliminating the need for communication with other agents across the network. This approach is ideal for distributed systems, as it minimizes communication overhead.

### 2.4.1 Architecture

Figure (2.1) illustrates the cloning controller that is present on each node in the network. Before migrating to another node, all mobile agents are queued up in a designated queue at each node. In addition to the code for execution on other platforms, a mobile agent carries with it a record of its current lifetime, cloning resources, and a number of rewards.

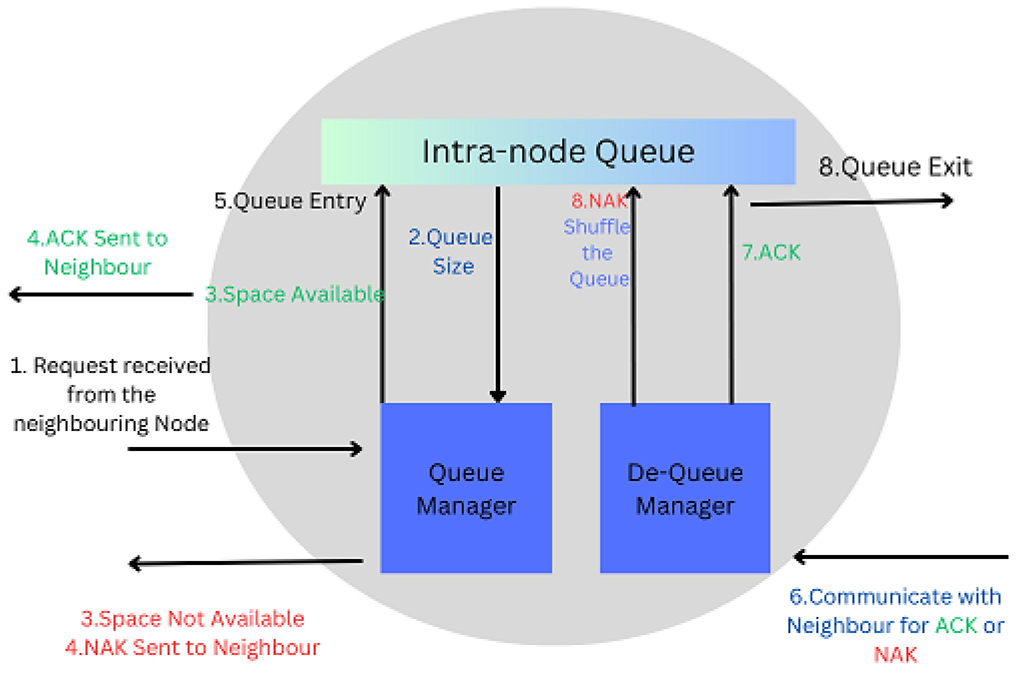


**Fig. 2.1**: Platform Architecture

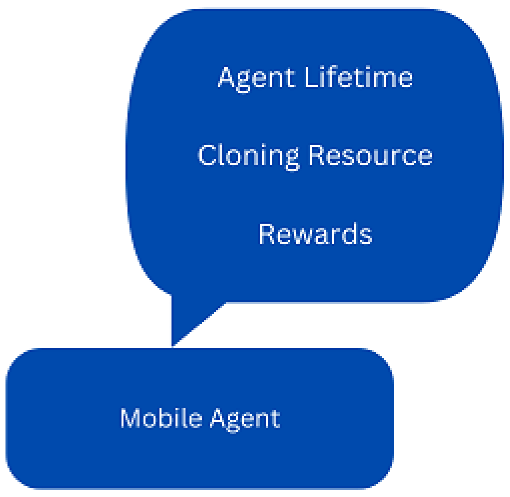


**Fig. 2.2**: Mobile Agent Architecture

The intra-node queue is managed by a **Queue Manager**. The **Dequeue Manager** is responsible for communicating with neighboring nodes within one hop by sending migration requests. It receives a positive or negative acknowledgment depending on the availability of a vacant slot in the destination node's queue. Migrations are performed only if there is an empty slot in the destination node's queue. The Lifetime decrement unit decrements the agent's lifetimes and updates the queue when the agent dies within it. The decision to clone is made based on the *Cloning Pressure* $\rho$, which is calculated using the population of the intra-node queue. Further details about the calculation will be provided subsequently.

### 2.4.2 Stigmergic Sensing

Stigmergic sensing is a communication method that utilizes the environment rather than traditional connection establishment or handshaking. When a mobile agent enters a node, it executes its payload to provide its service and then gets enqueued to the intra-node queue for migration to a neighboring node. By observing the queue, a mobile agent can determine the appropriate number of clones to generate.
Additionally, a queue threshold is defined to limit the capacity of each node's intra-node queue. Migration will not occur unless a request is generated to a vacant neighbor or an agent dies due to the expiration of its lifetime.

### 2.4.3 Cloning Resource

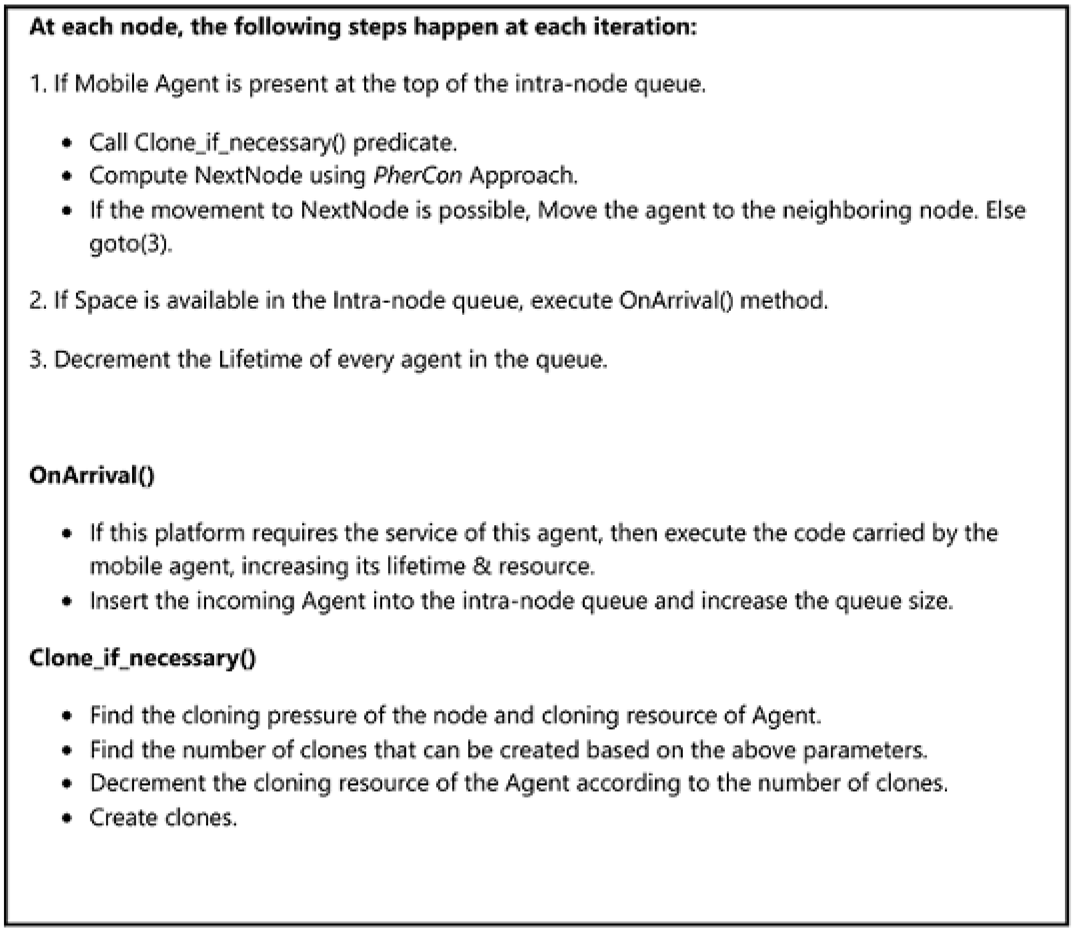
Cloning of mobile agents depends on several parameters, including the current size of the intra-node queue, the amount of available cloning resources, and the calculated cloning pressure. The cloning pressure denoted by $\sigma$ is determined by the intra-node queue threshold and the number of mobile agents currently in the intra-node queue. If the cloning pressure increases, it will lead to a higher probability of cloning. The amount of cloning resources also play a role, as a mobile agent with a higher amount of cloning resources has a greater chance of creating more mobile agents.

1. Cloning Resource: Added as payload to the agent.

2. Rewards: Added as payload specifying the reward it got after servicing any RRS.

3. Cloning Pressure: It varies with the number of available positions in the intra-node queue of that particular platform. In other words, it depends on the number of agents currently in the queue. The more the number of agents less will be the cloning pressure.

Cloning Resources and Lifetimes are recharged as and when the agent services the RRS and by another inherent charging mechanism.
Algorithm 1 shows this Mechanism.



**At each node, the following steps happen at each iteration:**

1. If Mobile Agent is present at the top of the intra-node queue.

- Call Clone_if_necessary() predicate.
- Compute NextNode using *PherCon* Approach.
- If the movement to NextNode is possible, Move the agent to the neighboring node. Else goto(3).

2. If Space is available in the Intra-node queue, execute OnArrival() method.

3. Decrement the Lifetime of every agent in the queue.

**OnArrival()**

- If this platform requires the service of this agent, then execute the code carried by the mobile agent, increasing its lifetime & resource.
- Insert the incoming Agent into the intra-node queue and increase the queue size.

**Clone_if_necessary()**

- Find the cloning pressure of the node and cloning resource of Agent.
- Find the number of clones that can be created based on the above parameters.
- Decrement the cloning resource of the Agent according to the number of clones.
- Create clones.

**Fig. 2.3**: Algorithm 1

### 2.4.4 Dynamics

Initially, only the parents are present in the queue; parent agents are the agents that always exist and never die. They populate the network with their clones. The decision to whether or not to clone is made by Cloning Pressure($\rho$) by following equation(2.1).

$$\rho_c^N(t) = \begin{cases} Q_{Th} - Q_N(t) & \text{for } \rho_c(t) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

where $Q_{Th}$ is the Queue Threshold and $Q_N(t)$ is the number of mobile agents in the queue.

The number of clones that should be generated is given by equation(2.2):-

$$C^M(t) = \rho_c^N(t)\left\{R_{av}^M(t)/R_{\max}\right\} \tag{2.2}$$

where $R_{av}^M$ is the available cloning resource within the agent $M$ and $R_{max}$ is the maximum cloning resource. Initially, all parents have a maximum resource, defined by a hyper-parameter named *Agent Max Resource.*

The cloning resource is decremented as per the following equation(2.3):-

$$R_{av}^M(t+1) = R_{av}^M(t) - C^M(t)R_{\min} \tag{2.3}$$

where $R_{min}$ is the minimum cloning resource required to generate a clone.

The Cloning resource is recharged based on the following equation(2.4):-

$$R^M_{av}(t+1) = \begin{cases} R^M_{av}(t) + \tau_c e^{-1/R_{av}(t)} + \tau_r \operatorname{Rew}(t) & \text{for } R^M_{av}(t) \geq 1, \rho^N_c(t) \leq 1 \\ R^M_{av}(t) + \tau_c + \tau_r \operatorname{Rew}(t) & \text{for } R^M_{av}(t) < 1, \rho^N_c(t) < 1 \\ R^M_{av}(t) + \tau_c e^{(1-1/x)} + \tau_r \operatorname{Rew}(t) & \text{for } \rho^N_c(t) > 1 \text{ where } x = \rho^N_c(t) \\ R_{\max} & \text{if } R^M_{av}(t+1) > R_{\max} \end{cases} \tag{2.4}$$

Agent's *Rew(t)* is 1 if the agent is able to service the RRS, else it is 0 at time *t*. $\tau_c$ and $\tau_r$ are positive constants.

The Lifetime is decremented as per the following equation(2.5):-

$$L(t+1) = L(t) - 1 \tag{2.5}$$

*Lifetimes* of the agents are increased when it services an RRS

$$L(t+1) = L(t) + \sigma \operatorname{Rew}(t) \tag{2.6}$$

where *L(t)* is the lifetime of an agent at time $t$ and $\sigma$ is a non-zero positive integer constant.

## 2.5 Conclusion

In this chapter, we reviewed previous works related to cloning and discussed its significance in a multi-node environment. The importance of judicious cloning was also highlighted, along with how our cloning controller model ensures it. Furthermore, we introduced various equations that serve as building blocks for the cloning controller.

# Chapter 3

# Contribution

## 3.1 Methodology

The cloning controller mechanism, as stated in the previous section, has been implemented in *SWI-Prolog*[6]. The code uses the functionalities of ***Tartarus*** [5], which is a multi-agent platform based on *SWI-Prolog*[6]. Multiple ***Tartarus*** platforms are instantiated on computers or on local systems, which provides an environment for agents to move around in a network of nodes. The goal is to service the platform's **Needs** as soon as possible with the help of cloning and then controlling the population so as to serve the request faster without congestion in the network. The implementation details for each functional unit are as follows:-

1. The ***Tartarus*** platforms can be instantiated on either local systems or computers connected via a wired or wireless transmission medium. Each platform acts as an RRS, as discussed in the previous chapter, and has a cloning controller started on it. Within each node, there is an intra-node queue implemented as a list data structure in *SWI-Prolog*.

2. The user has the flexibility to set several hyper-parameters based on their preferences, such as *queue threshold, clone lifetime, clone max-resource, clone resource, agent*

*lifetime, rewards, and other constants.*

3. The *q_manager handler* is responsible for managing the insertion of an agent into the intra-node queue. It checks whether there is enough space available in the queue for the agent. If there is sufficient space, it sends a positive acknowledgment. Otherwise, it sends a negative acknowledgment indicating that the queue is full.

4. At every iteration, both insertion and deletion occur at each node in the intra-node queue. The process of insertion has been explained in the previous point. For the deletion process, we implement the Pheromone-Conscientious (*PherCon*) approach.

5. The*PherCon* Approach involves the use of a Pheromone Database at each node, which is capable of diffusing pheromones when there is a need for a particular service. The pheromones are released to neighboring nodes, and then to their neighbors, and so on. The concentration of pheromones is highest at the originating node and gradually becomes lower as they spread to other nodes.

6. By default, outgoing agents exhibit Conscientious movement, which means they avoid revisiting previously visited nodes. However, when a pheromone trail is detected at a specific node, agents begin to follow the trail, which guides them to the requesting node. This bi-directional search approach results in faster service times for the nodes.

7. The *clone_if_necessary* predicate is invoked before sending to the neighbor to determine whether cloning should be done or not, based on the resource a mobile agent possesses and intra-node queue size, using the equations outlined in the previous chapter. The cloning functionality is provided by the *agent_clone* predicate in the ***Tartarus*** platform. The *deduct_cloning_resource* predicate manages the deduction of resources, and based on this, the *create_clone* predicate generates the specified number of clones.

8. After the generation of clones, it is time for them to migrate to other nodes.

The migration is based on *PherCon* approach. The *dequeue_manager* calls the *dequeue_manager_handle* to carry out this process. *The dequeue_manager_handle* communicates with the neighboring node via the *post_agent* predicate (defined in ***Tartarus***) to know if there is a slot available in the queue. If the slot is available, the agent moves to the corresponding node using the *move_agent* predicate defined in ***Tartarus***. If it is not available, it stays within the same queue.

9. The *lifetime* of each agent is decremented after each time step; the *Agent's lifetime* is a hyper-parameter that the user can set as per his application needs. The agent is purged as soon as the *lifetime* reaches zero.

10. There are other auxiliary functions written to help provide assistance in agent transfer as well as cloning.

## 3.2 Experimental Setup and Results

The testing & experiments were conducted on different network topologies, including Grid, Line, Mesh, Random, Star, and Tree topologies.

For the experiments, I have considered 1 Iteration = 10 seconds.

The values of common hyper-parameters for all the above topologies are as follows:

| **Serial Number** | **Column 1** | **Column 2** |
|---|---|---|
| 1 | Queue Size | 5 |
| 2 | Agent Max. Resource | 100 |
| 3 | Number of Nodes | 50 |
| 4 | $\tau_c$ | 0.1 |
| 5 | $\tau_r$ | 5 |
| 6 | Agent Min. Resource | 10 |
| 7 | Number of Agents | 3 |
| 8 | Max. Number of platforms requesting service at any time | 48 |

The values of other hyper-parameters were adjusted based on the various topologies. Following are the details of the testing phase.
The testing was performed on 4 cases with iteration numbers ranging from 1 to 300.

1. Normal case: The agents are requested randomly according to the **Need** of the platform.
2. **Extreme Case 1**: All platforms will require the service of Agent 1 & Agent 2 between Iteration 1 to 50, Agent 2 & Agent 3 between Iteration 51 to 100, and Agent 3 & Agent 1 between 101 to 150. Rest time, they will request randomly.
3. **Extreme Case 2**: All platforms will require the service of Agent 1 between Iteration 1 to 50, Agent 2 between Iteration 51 to 100, and Agent 3 between 101 to 150. Rest time, they will request randomly.
4. **Extreme Case 3**: All platforms will require the service of Agent 1, Agent 2, and Agent 3 between Iteration 1 to 150. Rest time, they will request randomly.

These four cases are tested against various network topologies, namely Grid, Mesh, Tree, Line, Random, and Star topology. We plotted the Agent-wise population of respective agents along with the total number of agents. Note that the graphs are only for extreme cases since the normal ones were bound to execute correctly if extreme ones are in place.

1. Grid Topology.

   A random grid connection of 5 x 10 nodes was generated using a Python script. In the case of Grid topology, the hyper-parameter $\sigma$ ranged from 3 to 7. The Agents were allowed to clone based on a Cloning resource which ranges from 20 to 50 initially. The graphs were plotted on extreme cases, and the following is the result.

   (a) Extreme Case 1: Total count of agents and serviced platforms with $\sigma = 3$:

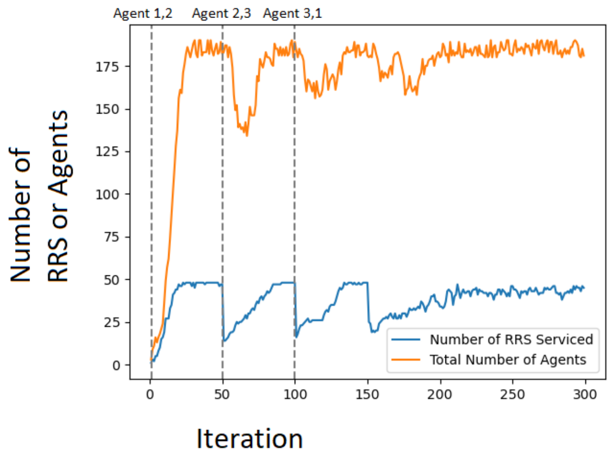


**Fig. 3.1**: Total count of agents and serviced platforms with $\sigma = 3$ (Grid), 1 iteration = 10 seconds

(b) Extreme Case 1: Agent-wise population $\sigma = 3$:

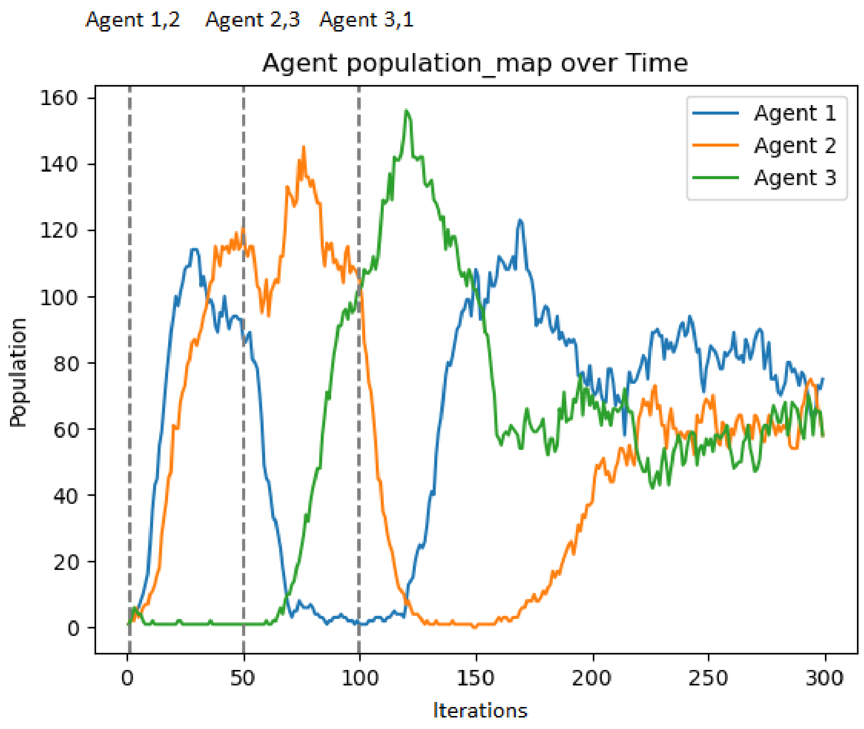


**Fig. 3.2**: Agent-wise Population $\sigma = 3$ (Grid), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 3$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These

two graphs were for the case when agent1 & agent2 were requested at 0, agent2 & agent3 were requested heavily at 50, and agent3 & agent1 were requested heavily at 100.

(c) Extreme Case 1: Total count of agents and serviced platforms with $\sigma = 7$:

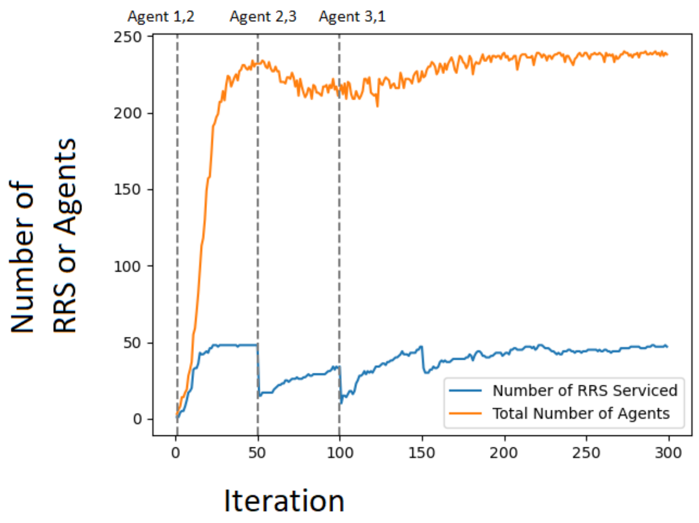


**Fig. 3.3**: Total count of agents and serviced platforms with $\sigma = 7$ (Grid), 1 iteration = 10 seconds

(d) Extreme Case 1: Agent-wise population $\sigma = 7$:

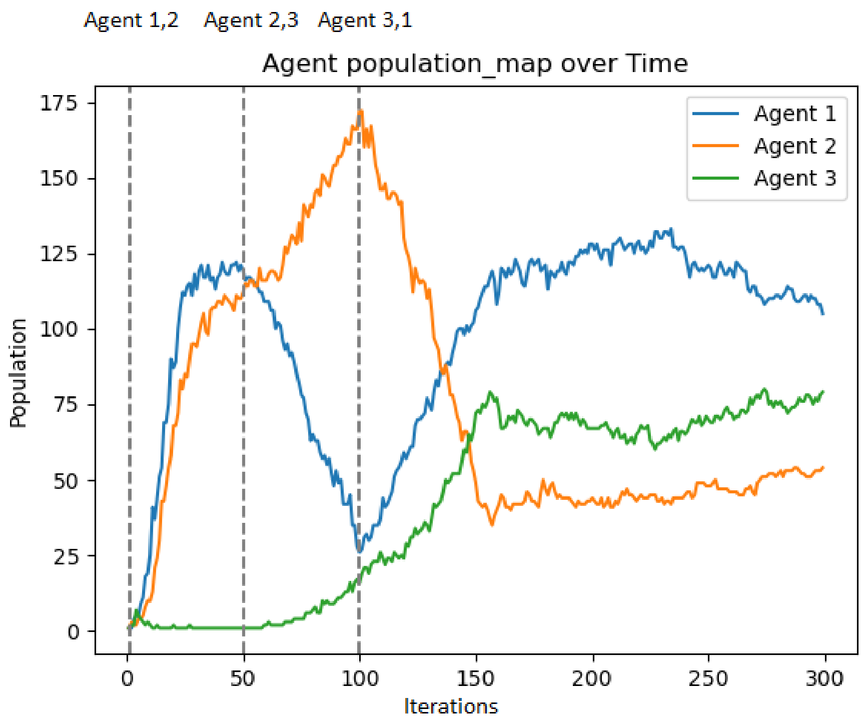


**Fig. 3.4**: Agent-wise Population $\sigma = 7$ (Grid), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 7$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1 & agent2 were requested at 0, agent2 & agent3 were requested heavily at 50, and agent3 & agent1 were requested heavily at 100.

(e) Extreme Case 2: Total count of agents and serviced platforms with $\sigma = 3$:

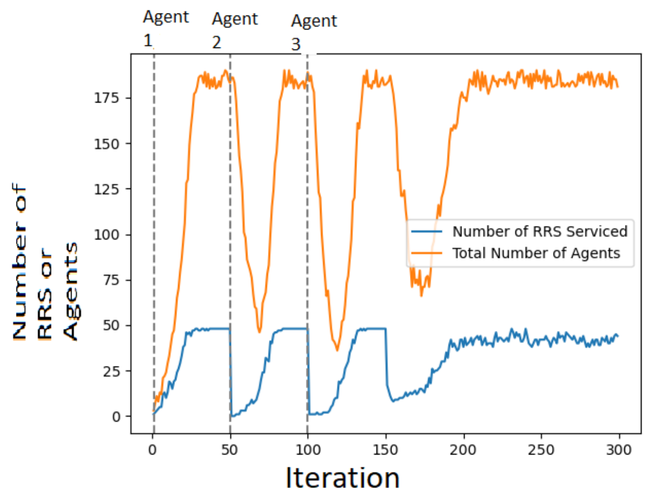


**Fig. 3.5**: Total count of agents and serviced platforms with $\sigma = 3$ (Grid), 1 iteration = 10 seconds

(f) Extreme Case 2: Agent-wise population $\sigma = 3$:

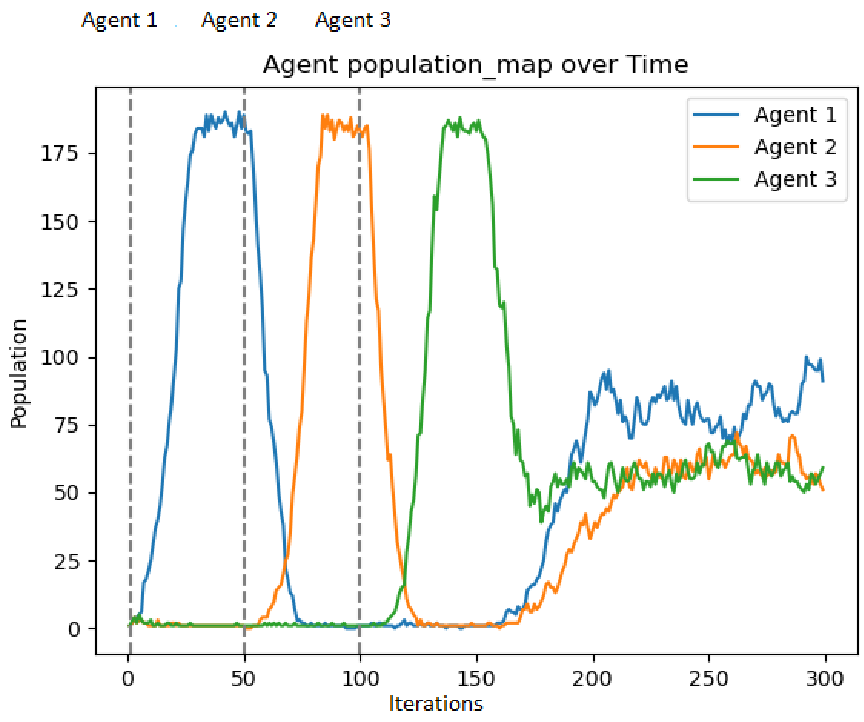


**Fig. 3.6**: Agent-wise Population $\sigma = 3$ (Grid), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 3$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These

two graphs were for the case when agent1 was requested at 0, agent2 was requested heavily at 50, and agent3 was requested heavily at 100.

(g) Extreme Case 3: Total count of agents and serviced platforms with $\sigma = 3$:

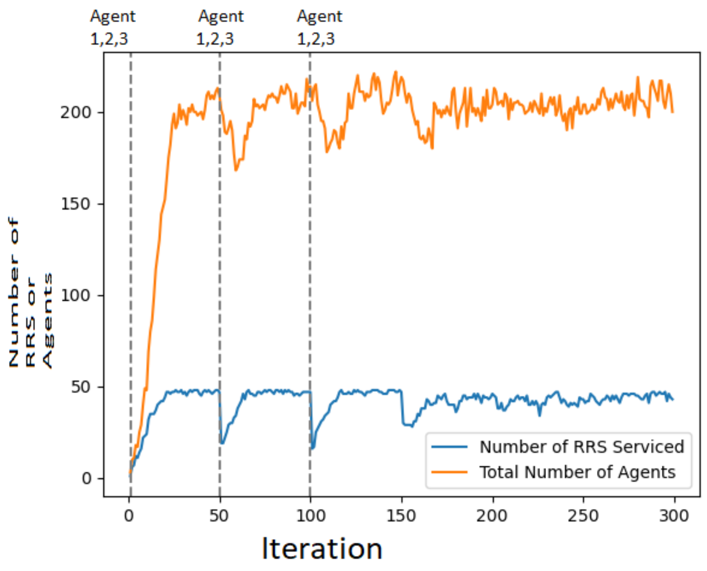


**Fig. 3.7**: Total count of agents and serviced platforms with $\sigma = 3$ (Grid), 1 iteration = 10 seconds

(h) Extreme Case 3: Agent-wise Population $\sigma = 3$:

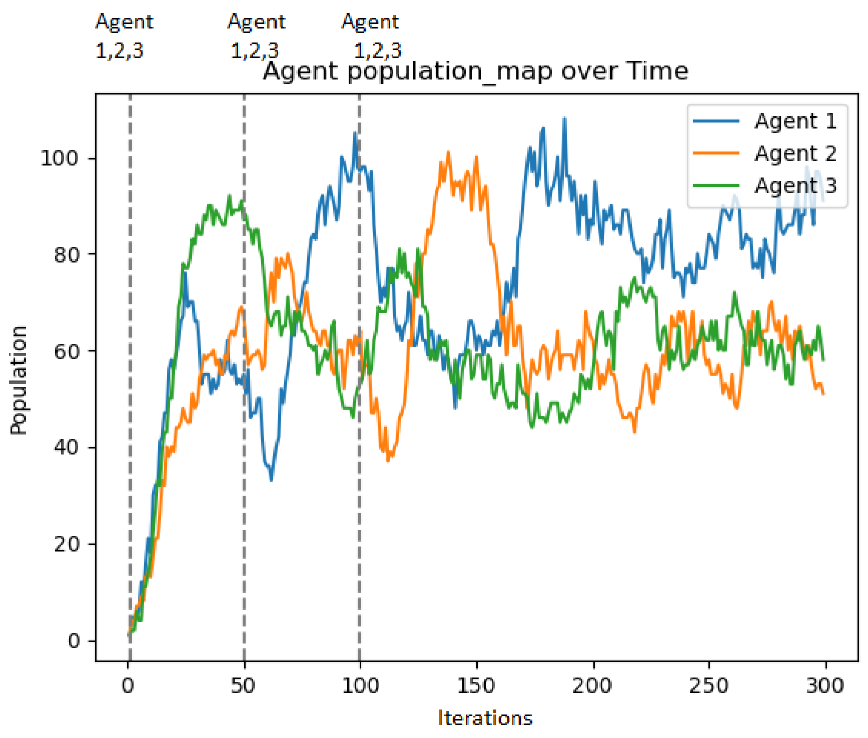


**Fig. 3.8**: Agent-wise Population $\sigma = 3$ (Grid), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 3$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1, agent2 & agent3 were requested at 50, agent1, agent2 & agent3 were requested heavily at 50, and agent1, agent2 & agent3 were requested heavily at 100.

2. Line Topology.

   A random line connection was generated using a Python script. In the case of the Line topology, we plotted graphs for $\sigma = 7$ only because agents were rapidly dying in the Line topology when $\sigma = 3$ since we have a narrow space and more hops.

   (a) Extreme Case 1: Total count of agents and serviced platforms with $\sigma = 7$:

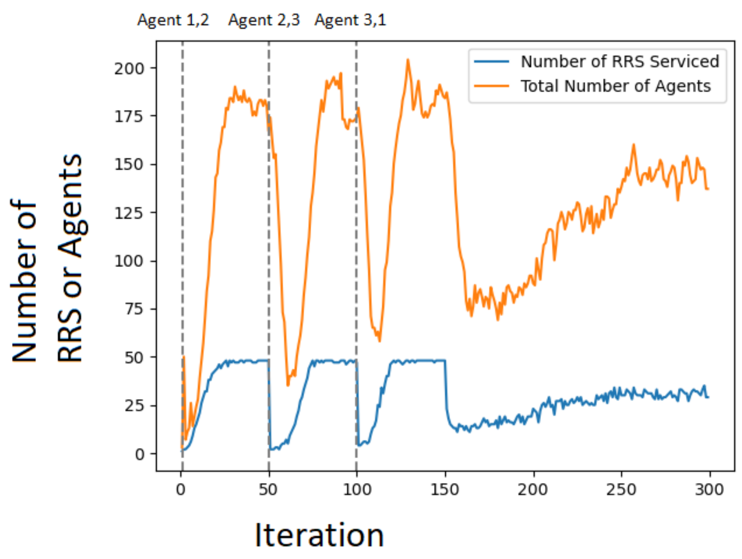


**Fig. 3.9**: Total count of agents and serviced platforms with $\sigma = 7$(Line), 1 iteration = 10 seconds

(b) Extreme Case 1: Agent-wise Population of Agents with $\sigma = 7$:

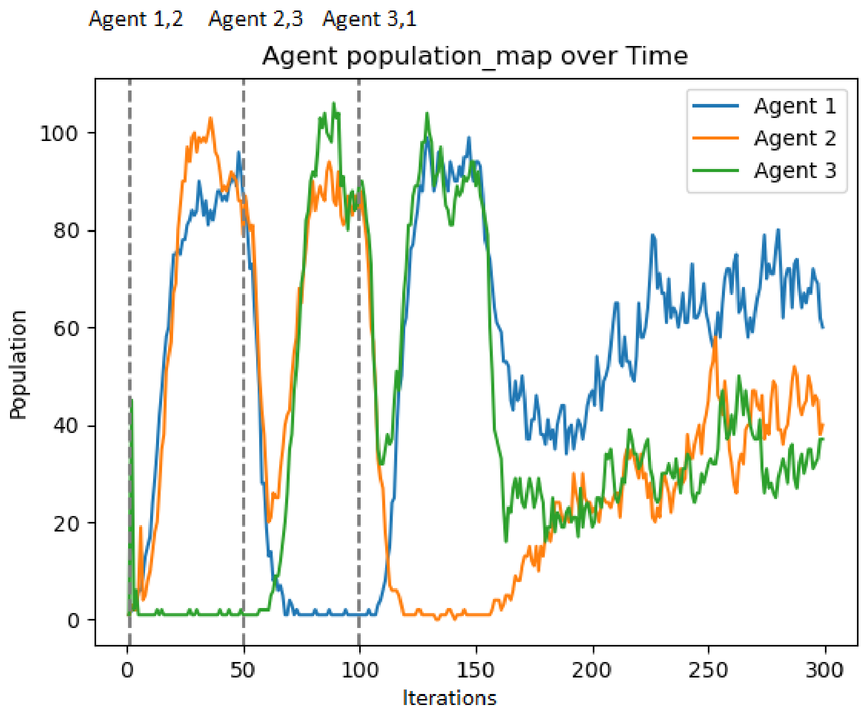


**Fig. 3.10**: Agent-wise population with $\sigma = 7$ (Line), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 7$; they show the total number of

agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1 & agent2 were requested at 50, agent2 & agent3 were requested heavily at 50, and agent1 & agent3 were requested heavily at 100.

(c) Extreme Case 2: Total count of agents and serviced platforms with $\sigma = 7$:

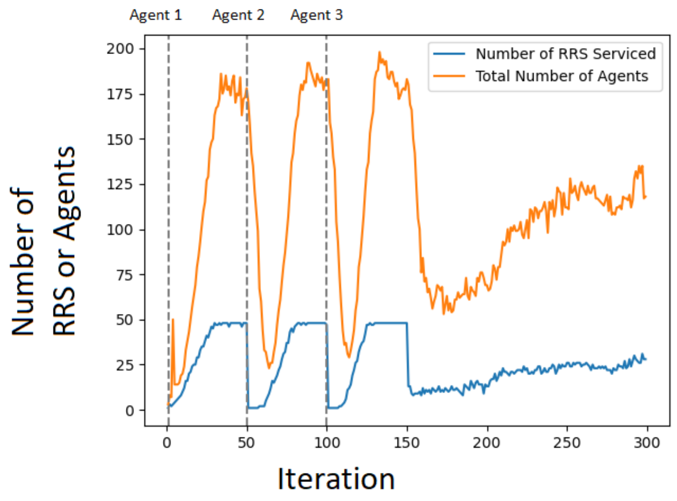


**Fig. 3.11**: Total count of agents and serviced platforms with $\sigma = 7$ (Line), 1 iteration = 10 seconds

(d) Extreme Case 2: Agent-wise Population of Agents with $\sigma = 7$:

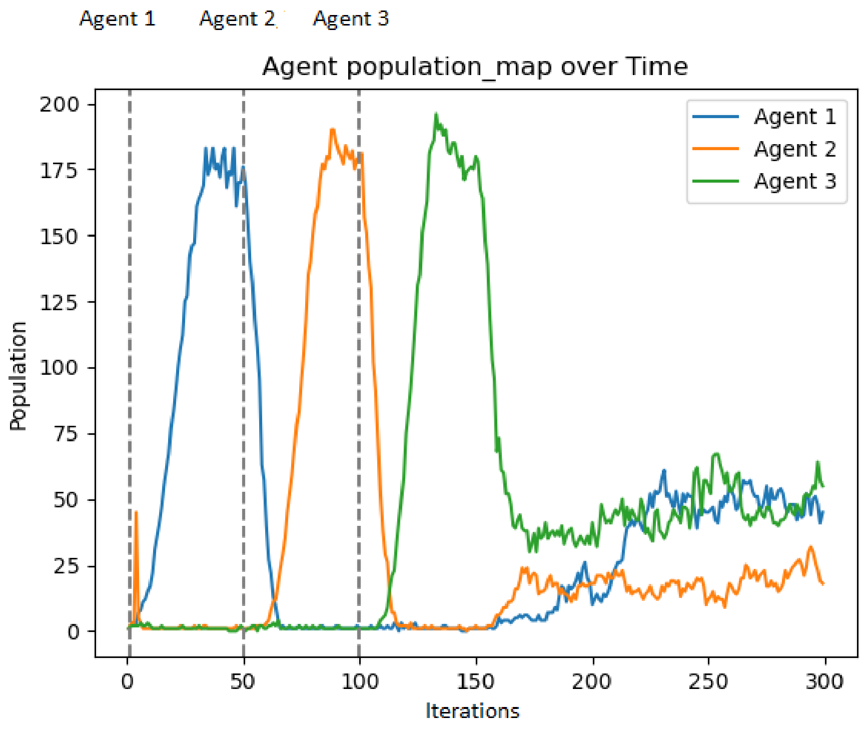


**Fig. 3.12**: Agent-wise population with $\sigma = 7$ (Line), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 7$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1 was requested at 50, agent2 was requested heavily at 50, and agent3 was requested heavily at 100.

(e) Extreme Case 3: Total count of agents and serviced platforms with $\sigma = 7$:

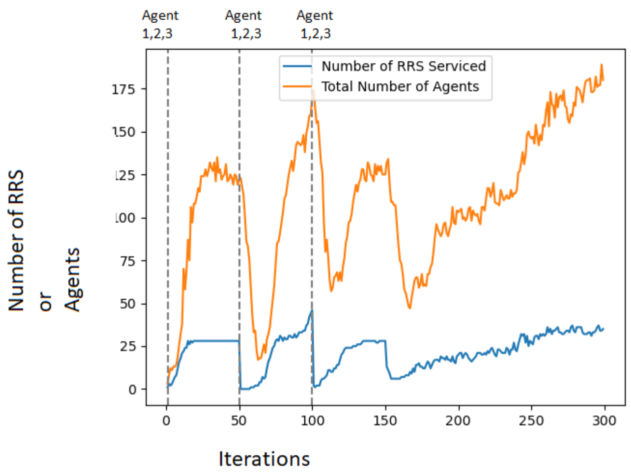


**Fig. 3.13**: Total count of agents and serviced platforms with $\sigma = 7$ (Line), 1 iteration = 10 seconds

(f) Extreme Case 3: Agent-wise population with $\sigma = 7$:

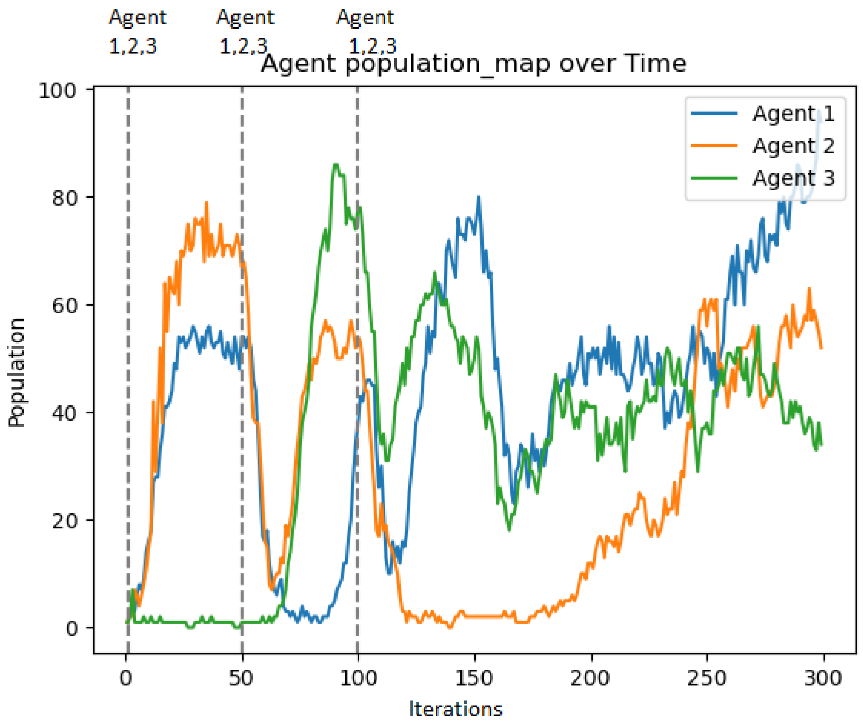


**Fig. 3.14**: Agent-wise population with $\sigma = 7$ (Line), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 7$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These

two graphs were for the case when agent1, agent2 & agent3 were requested at 50, agent1, agent2 & agent3 were requested heavily at 50, and agent1, agent2 & agent3 were requested heavily at 100.

3. Mesh Topology.

   A Mesh connection was generated using a Python script. In contrast to the Line Topology, Mesh is a fully connected topology, which means platforms are just one hop distance from each other. The $\sigma$ value must be accordingly small because if the Agents persist for long, they will congest the network. We took $\sigma = 3$ for this case.

   (a) Extreme Case 1: Total count of agents and serviced platforms with $\sigma = 3$:

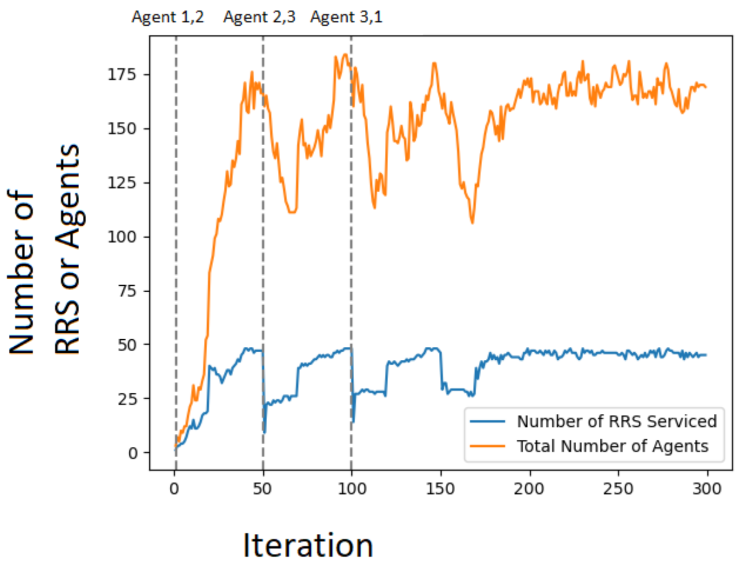


**Fig. 3.15**: Total count of agents and serviced platforms with $\sigma = 3$ (Mesh), 1 iteration = 10 seconds

   (b) Extreme Case 1: Agent-wise Population of Agents with $\sigma = 3$:

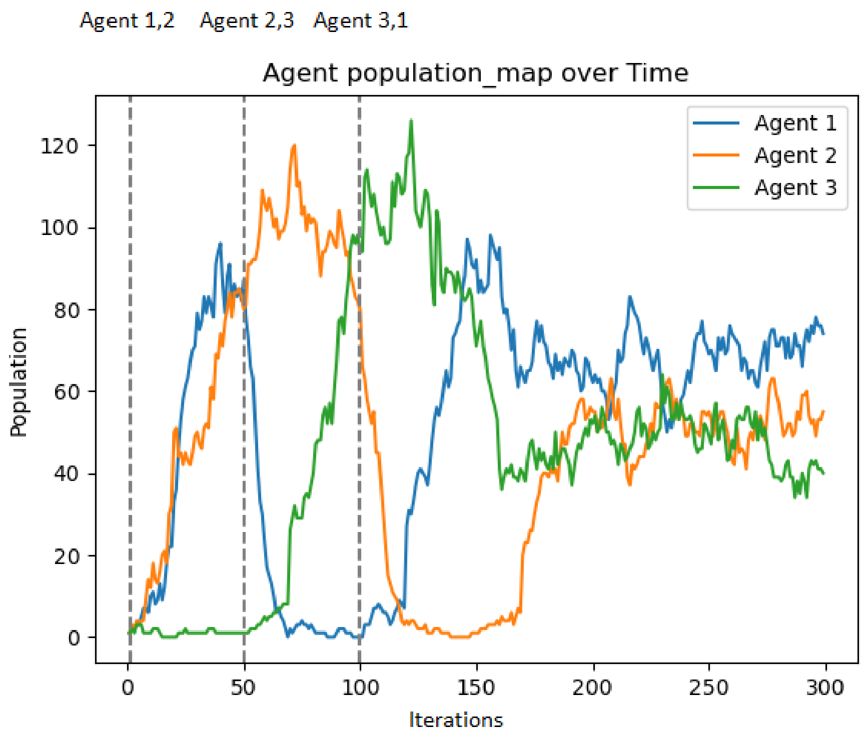


**Fig. 3.16**: Agent-wise Population with $\sigma = 3$ (Mesh), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 3$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1 & agent2 were requested at 50, agent2 & agent3 were requested heavily at 50, and agent1 & agent3 were requested heavily at 100.

(c) Extreme Case 2: Total count of agents and serviced platforms with $\sigma = 3$:

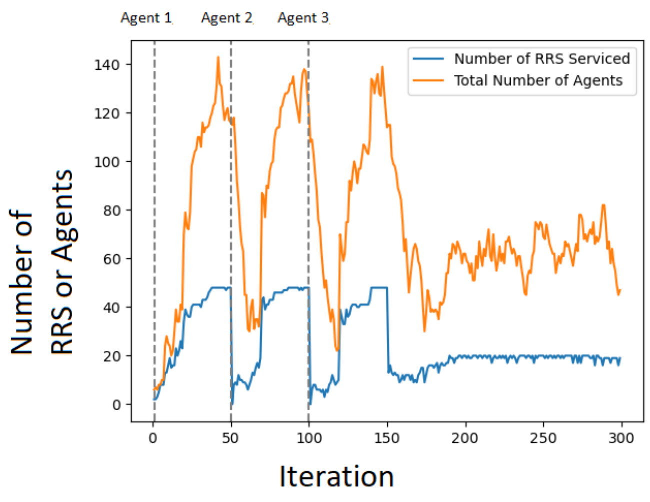


**Fig. 3.17**: Total count of agents and serviced platforms with $\sigma = 3$ (Mesh), 1 iteration = 10 seconds

(d) Extreme Case 2: Agent-wise Population of Agents with $\sigma = 3$:

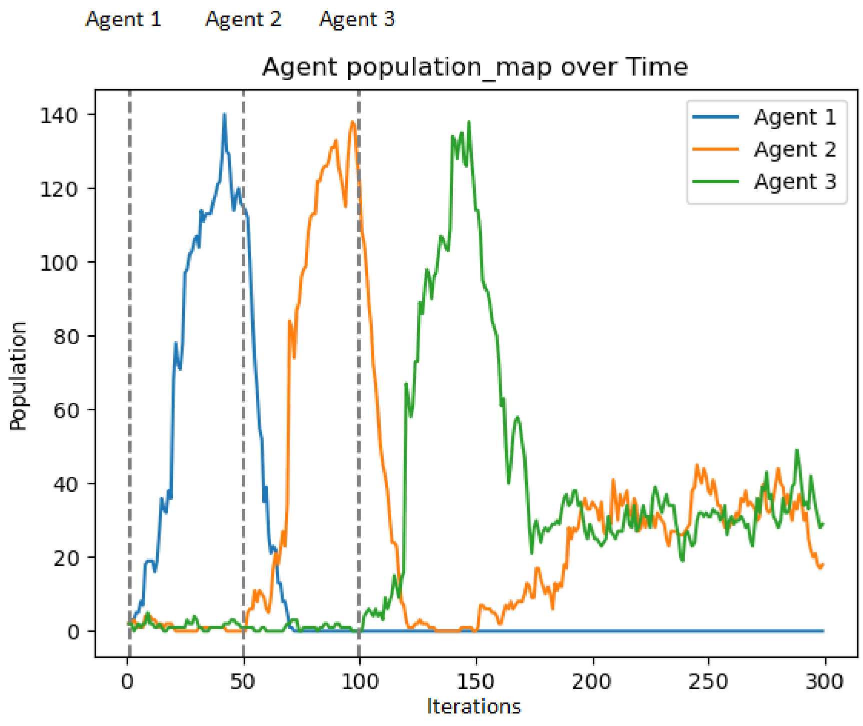


**Fig. 3.18**: Agent-wise Population with $\sigma = 3$ (Mesh), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 3$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These

two graphs were for the case when agent1 was requested at 50, agent2 was requested heavily at 50, and agent3 was requested heavily at 100.

(e) Extreme Case 3: Total count of agents and serviced platforms with $\sigma = 3$:

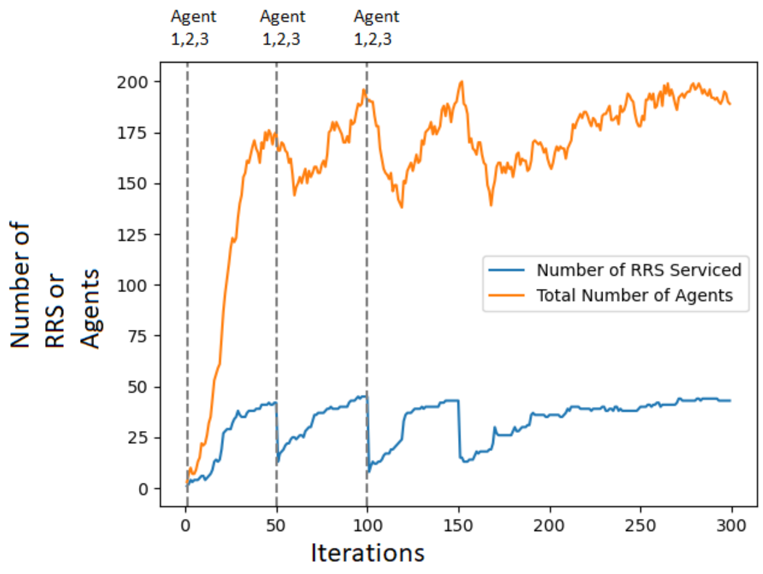


**Fig. 3.19**: Total count of agents and serviced platforms with $\sigma = 3$ (Mesh), 1 iteration = 10 seconds

(f) Extreme Case 3: Agent-wise Population of Agents with $\sigma = 3$:

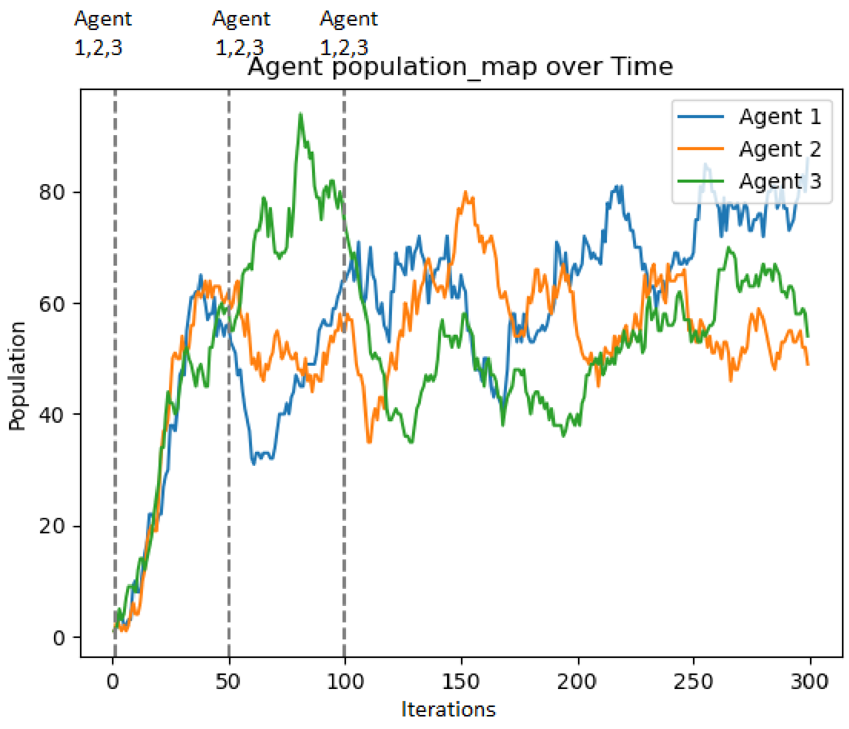


**Fig. 3.20**: Agent-wise Population with $\sigma = 3$ (Mesh), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 3$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1, agent2 & agent3 were requested at 50, agent1, agent2 & agent3 were requested heavily at 50, and agent1, agent2 & agent3 were requested heavily at 100.

4. Random Topology.

   A random connection was generated using a Python script. The random connection had cycles, pendant nodes, and bridges. The $\sigma$ value was taken to be 4. Note that a value equal to three or five may also suffice. It mostly depends on the application of what we choose.

   (a) Extreme Case 1: Total count of agents and serviced platforms with $\sigma = 4$:

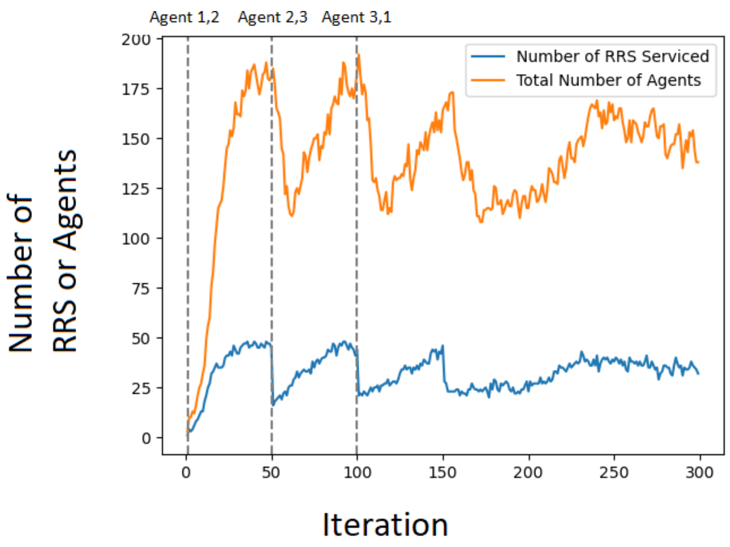


**Fig. 3.21**: Total count of agents and serviced platforms with $\sigma = 4$ (Random), 1 iteration = 10 seconds

(b) Extreme Case 1: Agent-wise Population of Agents with $\sigma = 4$:

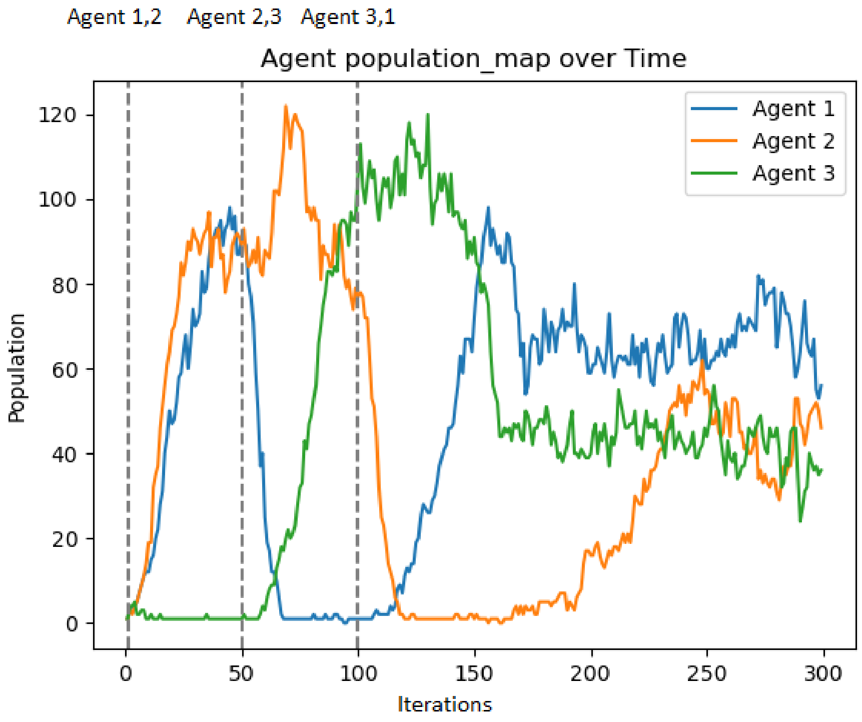


**Fig. 3.22**: Agent-wise Population with $\sigma = 4$ (Random), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 4$; they show the total number of

agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1 & agent2 were requested at 50, agent2 & agent3 were requested heavily at 50, and agent1 & agent3 were requested heavily at 100.

(c) Extreme Case 2: Total count of agents and serviced platforms with $\sigma = 4$:

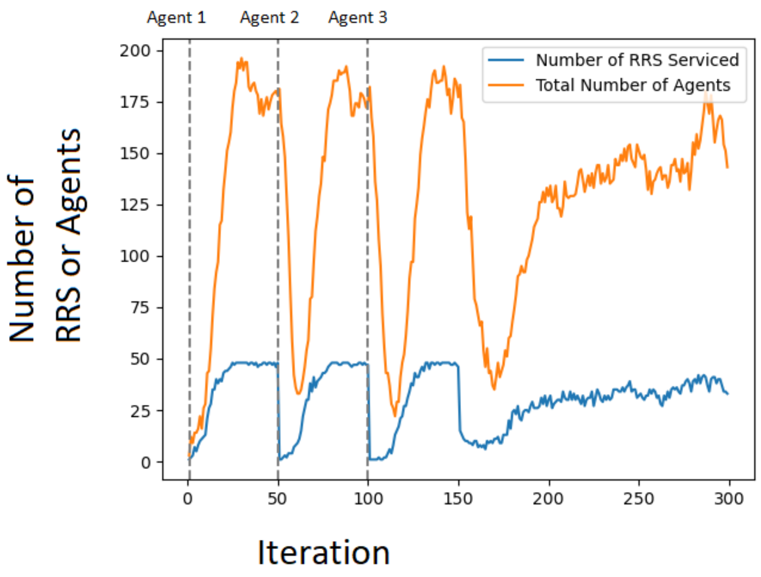


**Fig. 3.23**: Total count of agents and serviced platforms with $\sigma = 4$ (Random), 1 iteration = 10 seconds

(d) Extreme Case 2: Agent-wise Population of Agents with $\sigma = 4$:

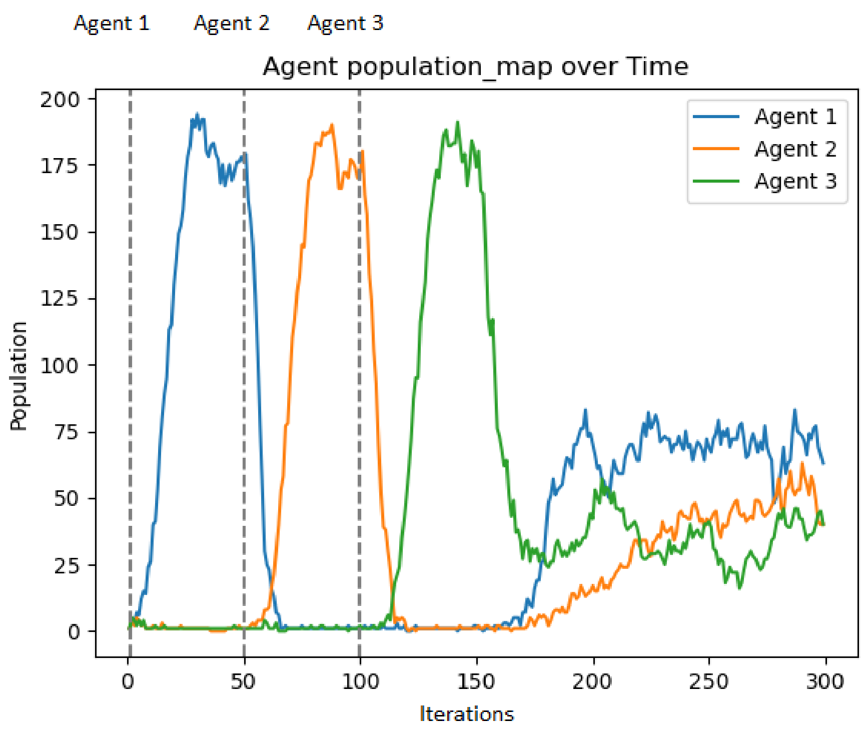


**Fig. 3.24**: Agent-wise Population with $\sigma$ = 4 (Random), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma$ = 4; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1 was requested at 50, agent2 was requested heavily at 50, and agent3 was requested heavily at 100.

(e) Extreme Case 3: Total count of agents and serviced platforms with $\sigma$ = 4:

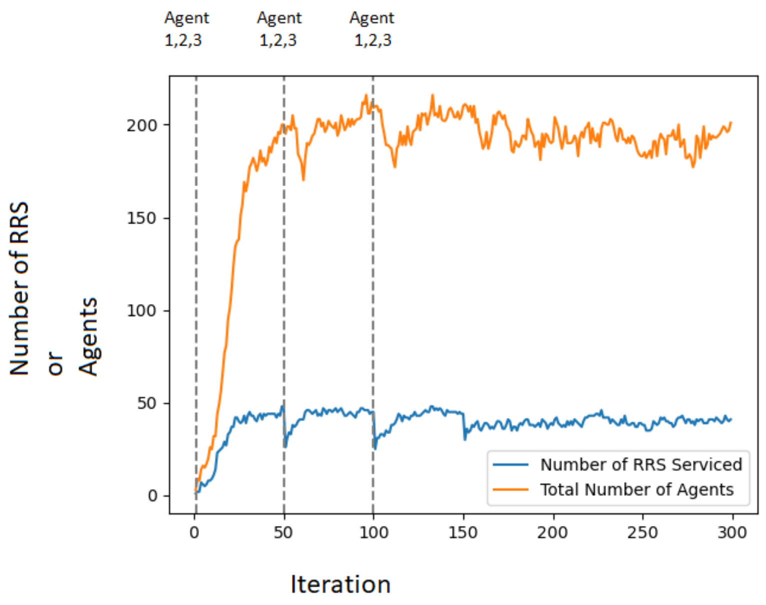


**Fig. 3.25**: Total count of agents and serviced platforms with $\sigma = 4$ (Random), 1 iteration = 10 seconds

(f) Extreme Case 3: Agent-wise Population of Agents with $\sigma = 4$:

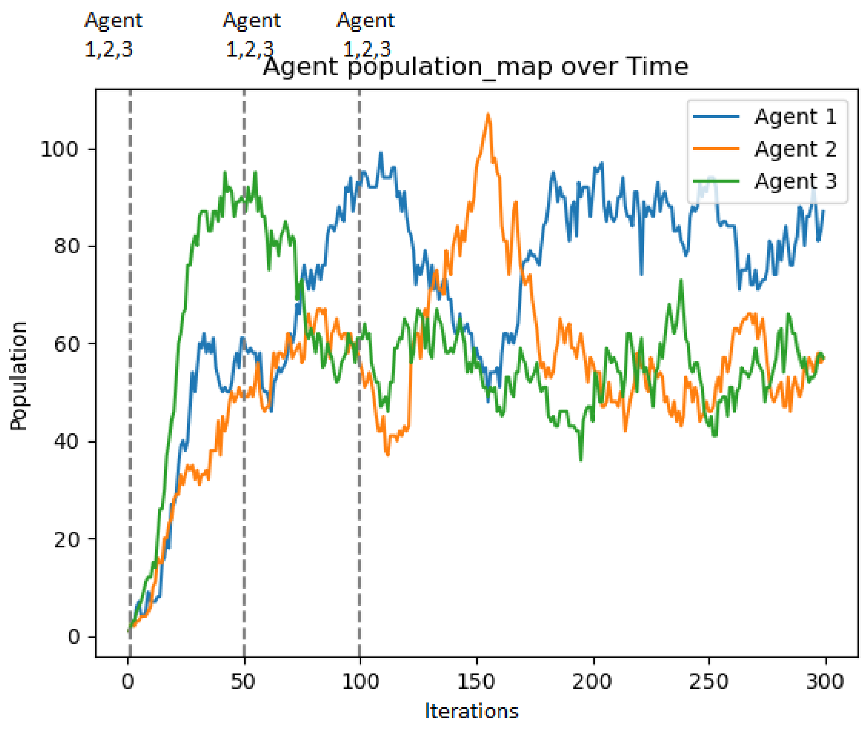


**Fig. 3.26**: Agent-wise Population with $\sigma = 4$ (Random), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 4$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1, agent2 & agent3 were requested at 50, agent1, agent2 & agent3 were requested heavily at 50, and agent1, agent2 & agent3 were requested heavily at 100.

5. Star Topology.

A Star connection was generated randomly using a Python script. The Star connection is the one in which there is a single node in the center, and all other nodes are only the center node's neighbors. In this case, the number of platforms serviced is low. The reason is congestion at the middle node. Rest nodes do not have much population of agents since the middle node has to do all the processing. Since it is a dense topology, the $\sigma$ value should be low, we took it as 3.

(a) Extreme Case 1: Total count of agents and serviced platforms with $\sigma = 3$:

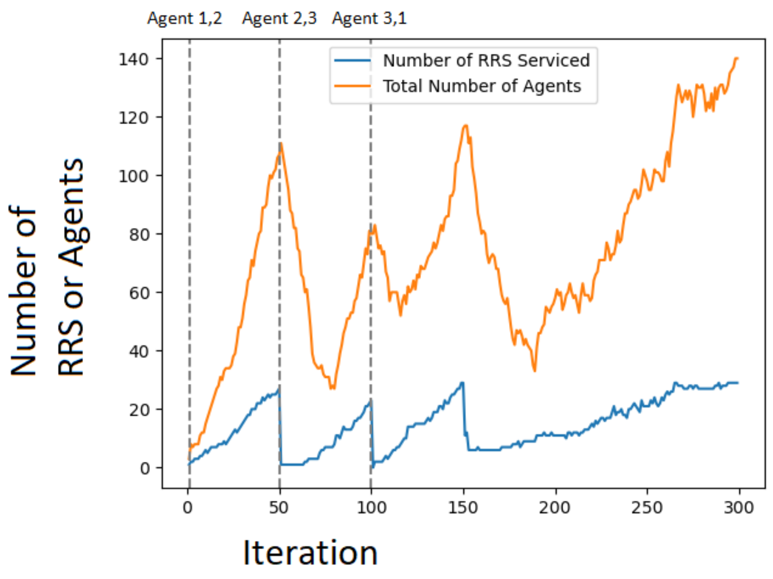


**Fig. 3.27**: Total count of agents and serviced platforms with $\sigma = 3$ (Star), 1 iteration = 10 seconds

(b) Extreme Case 1: Agent-wise Population of Agents with $\sigma = 3$:

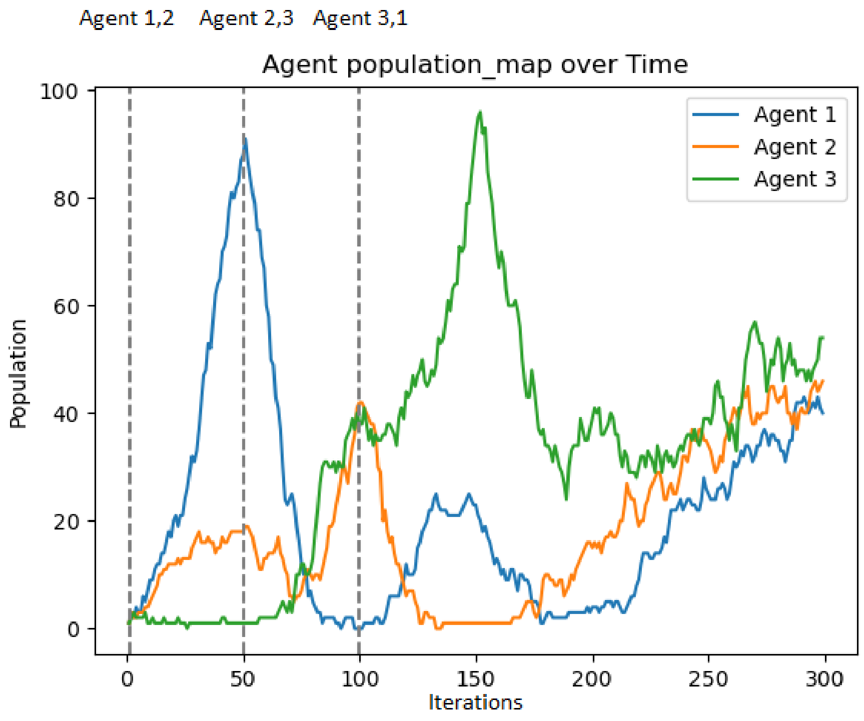


**Fig. 3.28**: Agent-wise Population with $\sigma = 3$ (Star), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 3$; they show the total number of

agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1 & agent2 were requested at 50, agent2 & agent3 were requested heavily at 50, and agent1 & agent3 were requested heavily at 100.

(c) Extreme Case 2: Total count of agents and serviced platforms with $\sigma = 3$:

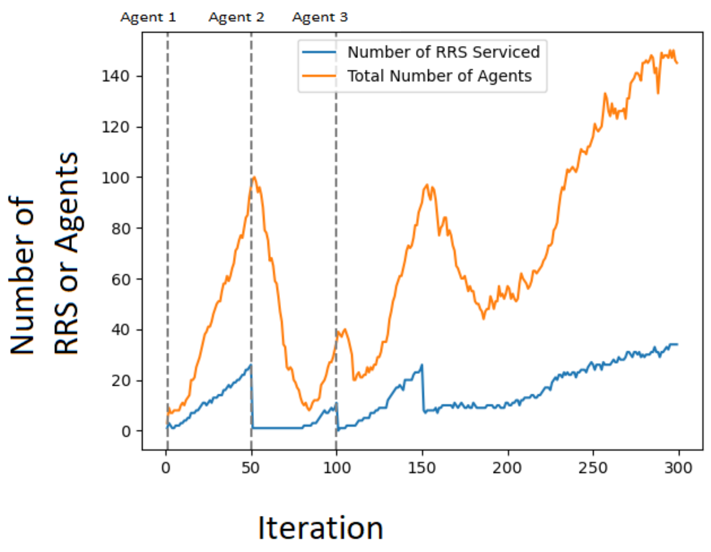


**Fig. 3.29**: Total count of agents and serviced platforms with $\sigma = 3$ (Star), 1 iteration = 10 seconds

(d) Extreme Case 2: Agent-wise Population of Agents with $\sigma = 3$:

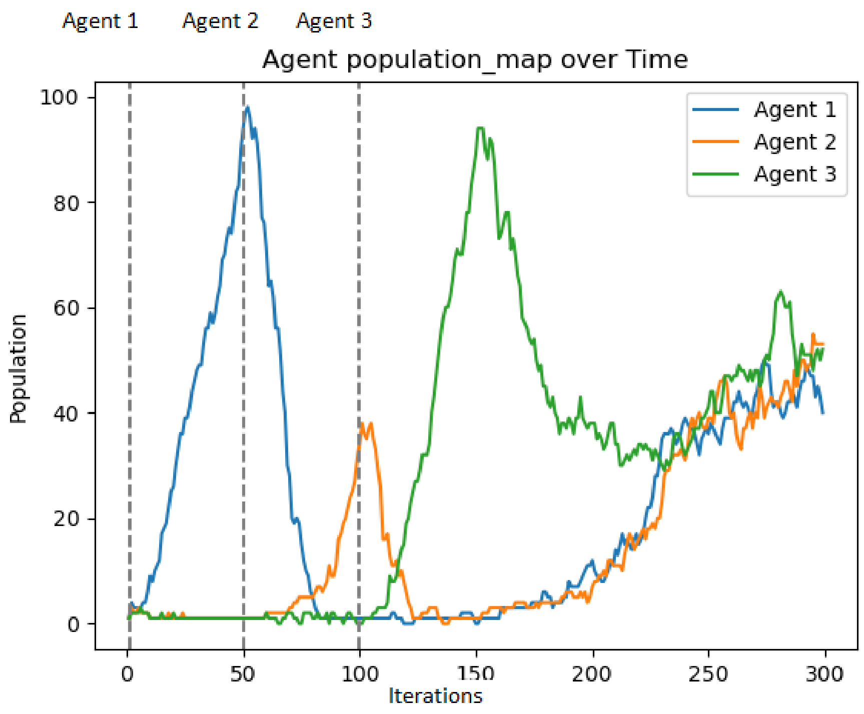


**Fig. 3.30**: Agent-wise Population with $\sigma = 3$ (Star), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 3$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1 was requested at 50, agent2 was requested heavily at 50, and agent3 was requested heavily at 100.

(e) Extreme Case 3: Total count of agents and serviced platforms with $\sigma = 3$:

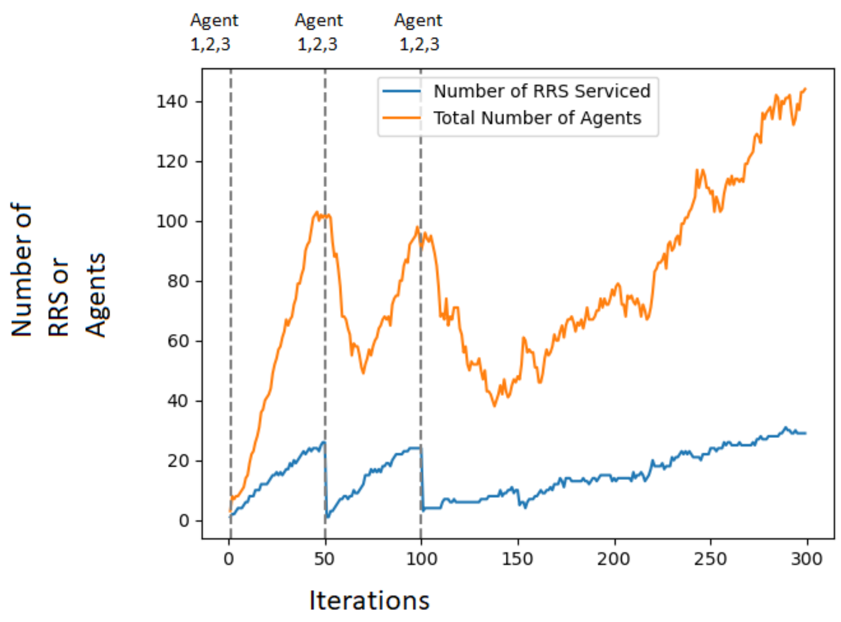


**Fig. 3.31**: Total count of agents and serviced platforms with $\sigma = 3$ (Star), 1 iteration = 10 seconds

(f) Extreme Case 3: Agent-wise Population of Agents with $\sigma = 3$:

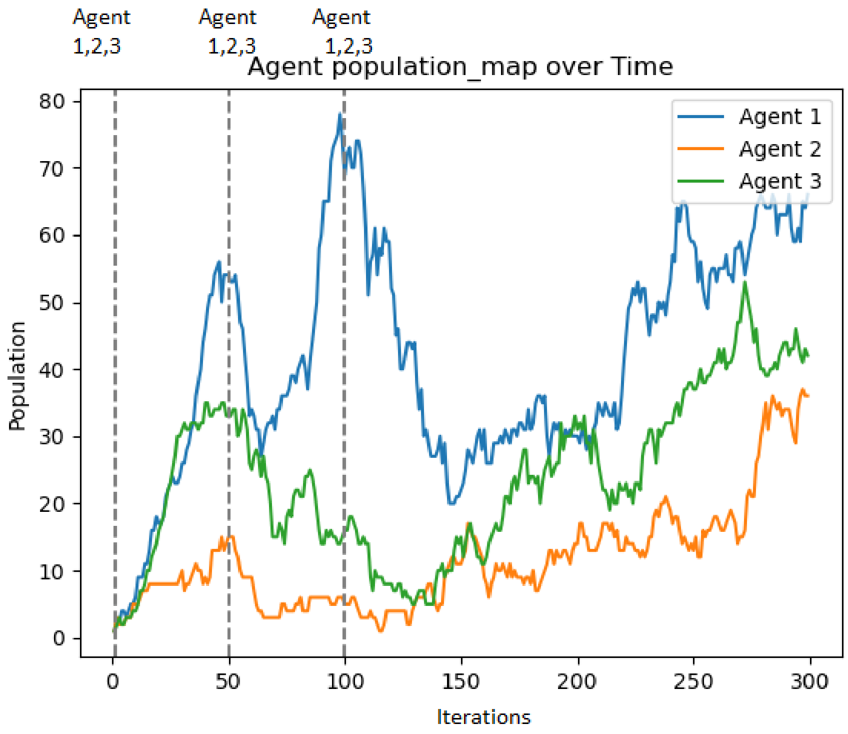


**Fig. 3.32**: Agent-wise Population with $\sigma = 3$ (Star), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 3$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1, agent2 & agent3 were requested at 50, agent1, agent2 & agent3 were requested heavily at 50, and agent1, agent2 & agent3 were requested heavily at 100.

6. Tree Topology.

   A random Tree connection was made using a Python script. It had a root node, and that node had some offspring; the offspring further had their own offspring, and so on, until leaves were reached. This case also has a somewhat less number of requests satisfied. It may be because of the sparsity of the number of edges. And we have only a single way to travel between 2 nodes, which may also be a reason for less number of satisfied requests. Since it is a somewhat sparse connection, we tested it with $\sigma = 3$ and 7.

   (a) Extreme Case 1: Total count of agents and serviced platforms with $\sigma = 3$:

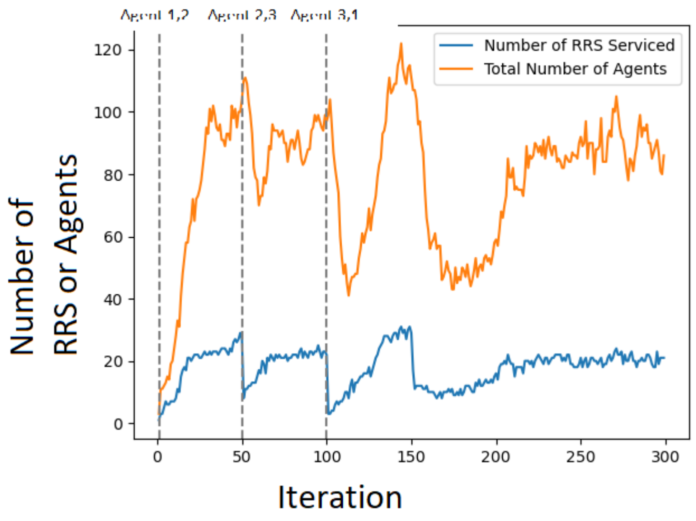


**Fig. 3.33**: Total count of agents and serviced platforms with $\sigma$ = 3 (Tree), 1 iteration = 10 seconds

(b) Extreme Case 1: Agent-wise population $\sigma$ = 3:

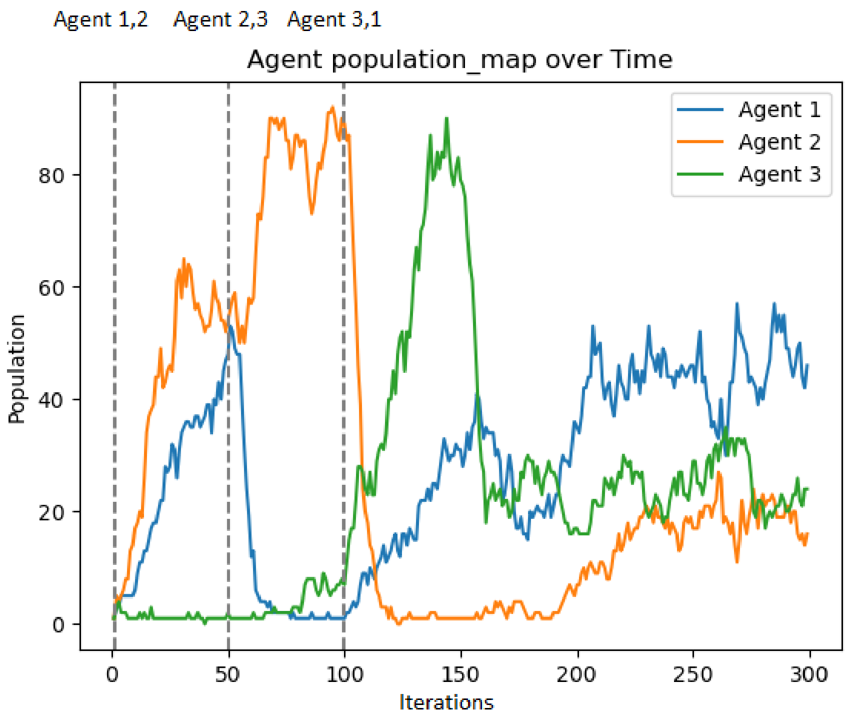


**Fig. 3.34**: Agent-wise Population $\sigma$ = 3 (Tree), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma$ = 3; they show the total number of

agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1 & agent2 were requested at 50, agent2 & agent3 were requested heavily at 50, and agent1 & agent3 were requested heavily at 100.

(c) Extreme Case 1: Total count of agents and serviced platforms with $\sigma = 7$:

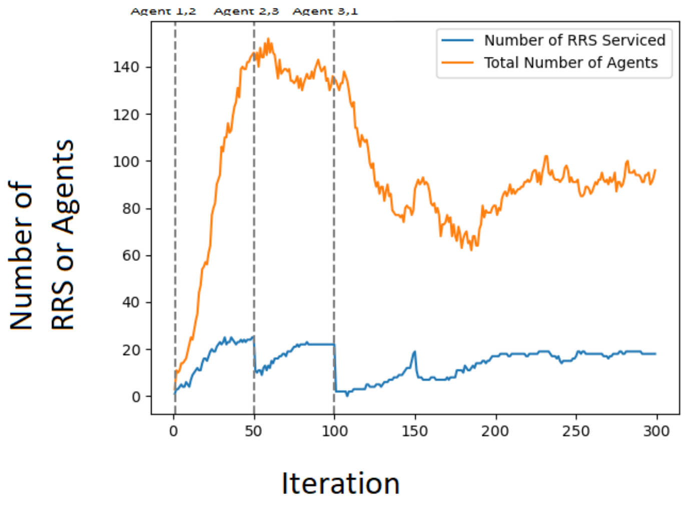


**Fig. 3.35**: Total count of agents and serviced platforms with $\sigma = 7$ (Tree), 1 iteration = 10 seconds

(d) Extreme Case 1: Agent-wise population $\sigma = 7$:

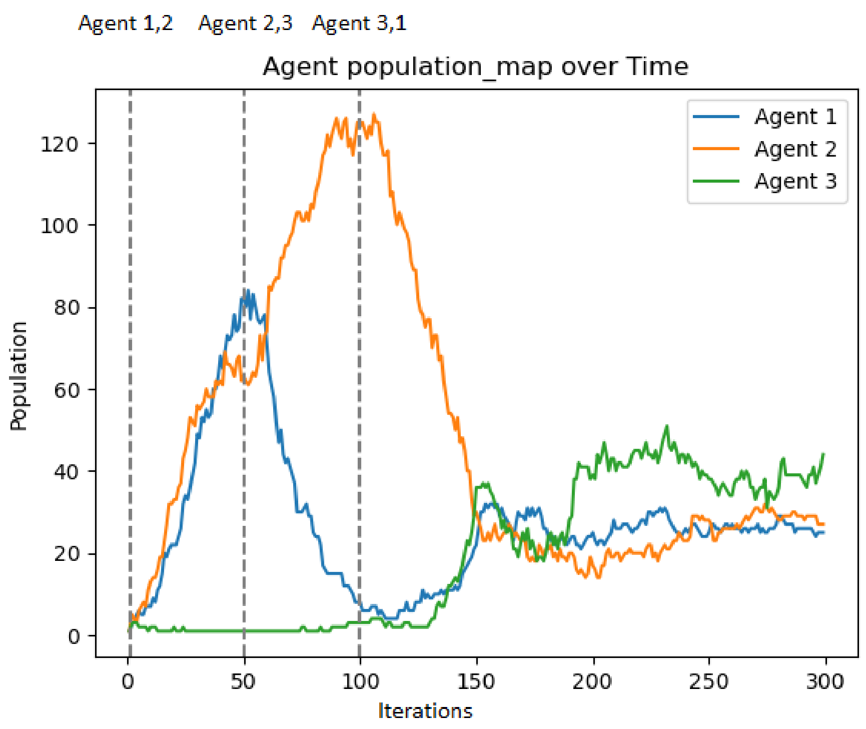


**Fig. 3.36**: Agent-wise Population $\sigma = 7$ (Tree), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 7$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1 & agent2 were requested at 50, agent2 & agent3 were requested heavily at 50, and agent1 & agent3 were requested heavily at 100.

(e) Extreme Case 2: Total count of agents and serviced platforms with $\sigma = 3$:

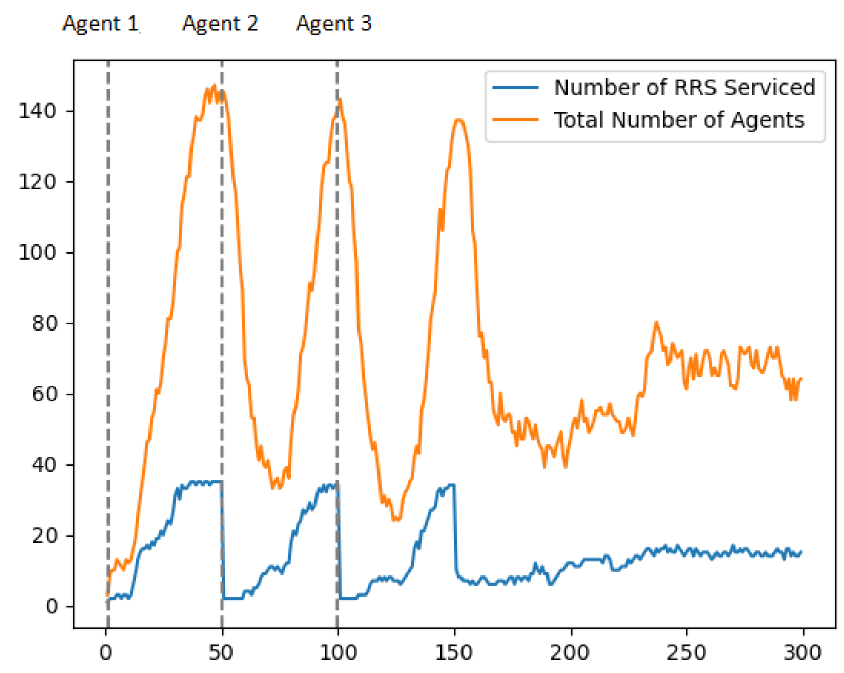


**Fig. 3.37**: Total count of agents and serviced platforms with $\sigma = 3$ (Tree), 1 iteration = 10 seconds

(f) Extreme Case 2: Agent-wise population $\sigma = 3$:

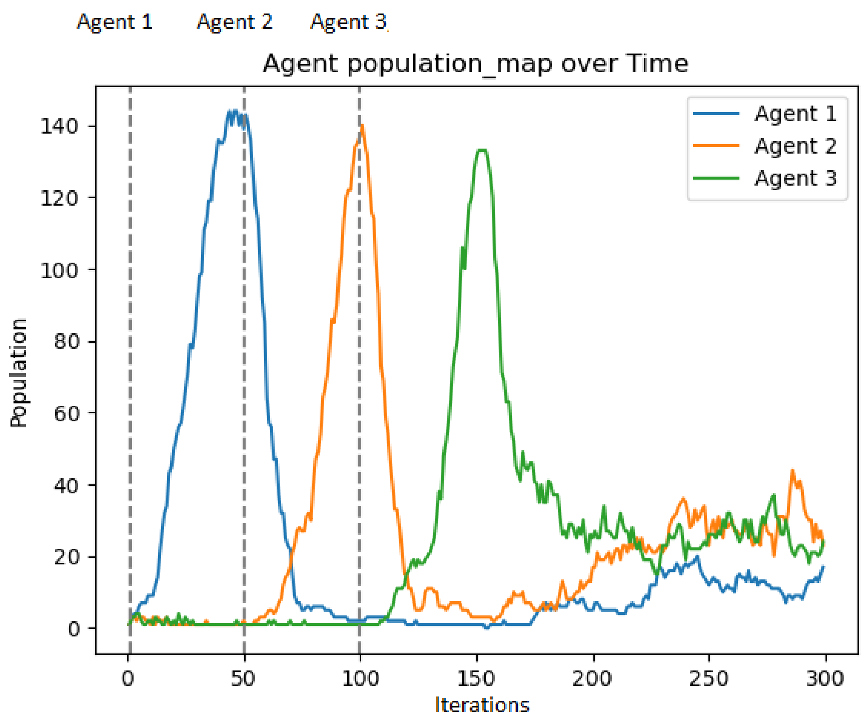


**Fig. 3.38**: Agent-wise Population $\sigma = 3$ (Tree), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 3$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1 was requested at 50, agent2 was requested heavily at 50, and agent3 was requested heavily at 100.

(g) Extreme Case 3: Total count of agents and serviced platforms with $\sigma = 7$:

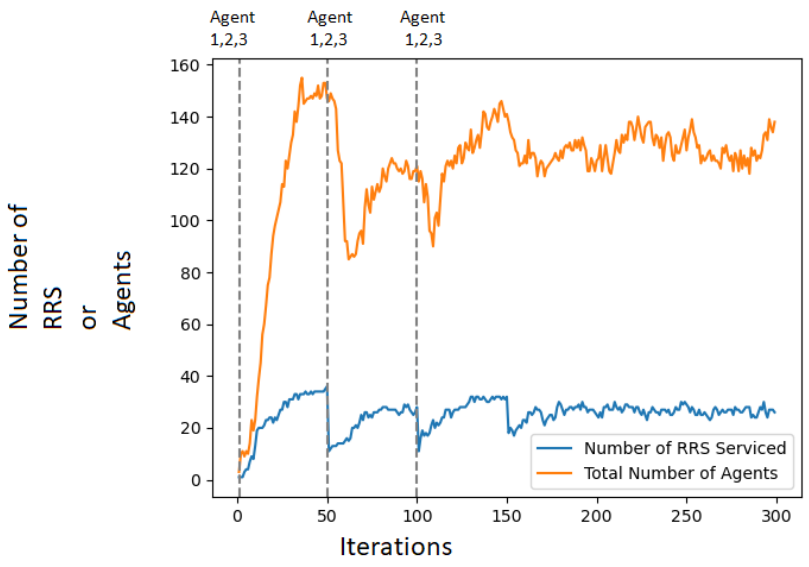


**Fig. 3.39**: Total count of agents and serviced platforms with $\sigma = 7$ (Tree), 1 iteration = 10 seconds

(h) Extreme Case 3: Agent-wise population $\sigma = 3$:

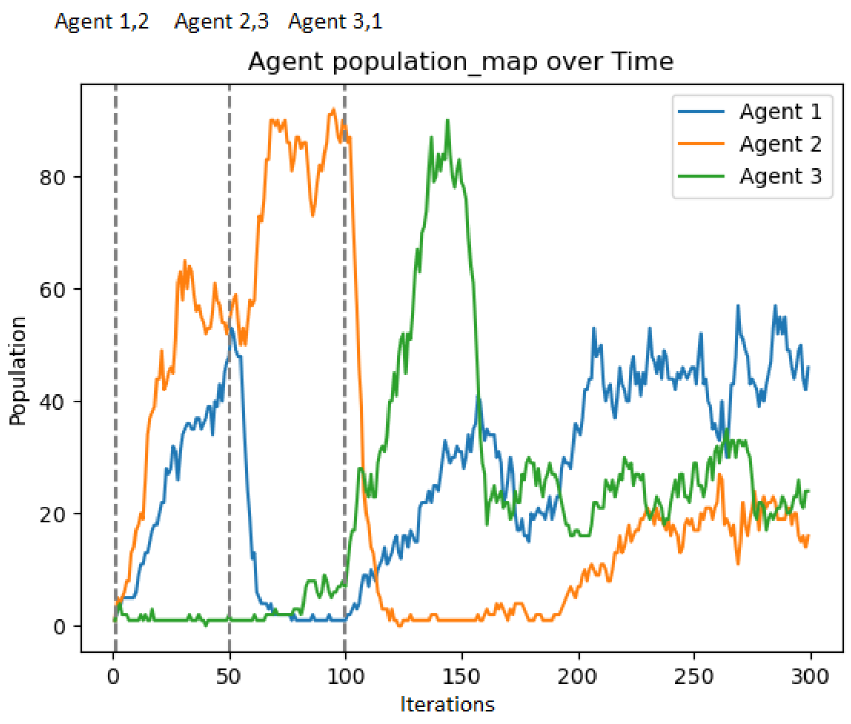


**Fig. 3.40**: Agent-wise Population $\sigma = 3$ (Tree), 1 iteration = 10 seconds

The above 2 graphs were obtained for $\sigma = 3$; they show the total number of agents, the platforms service, and the agent-wise population, respectively. These two graphs were for the case when agent1, agent2 & agent3 were requested at 50, agent1, agent2 & agent3 were requested heavily at 50, and agent1, agent2 & agent3 were requested heavily at 100.

Experiments were performed with changing queue size as well. As the queue size decreases, the service times are observed to be increasing.

The following are the results obtained:-

1. Topology = Grid, Service time graphs:

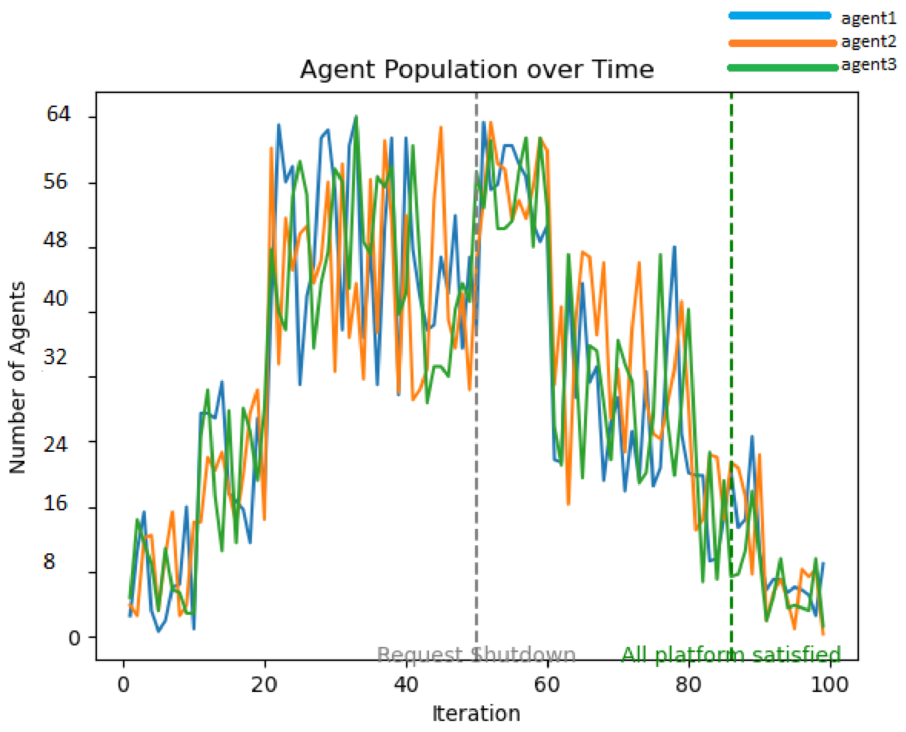


**Fig. 3.41**: Service Time with Queue size = 5, (Grid), 1 iteration = 10 seconds

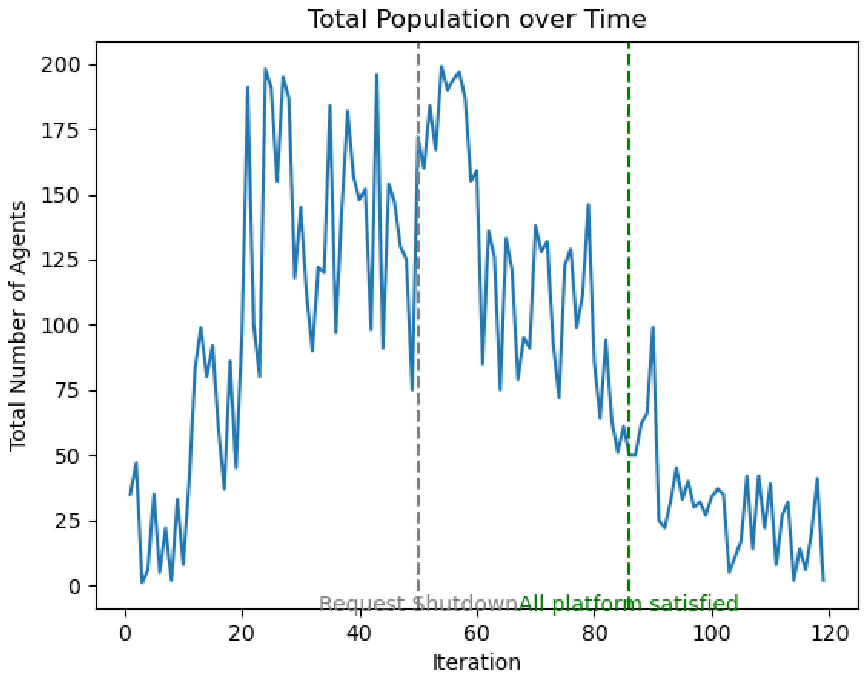


**Fig. 3.42**: Total number of agents and Agent Service Time with Queue size = 5, (Grid), 1 iteration = 10 seconds

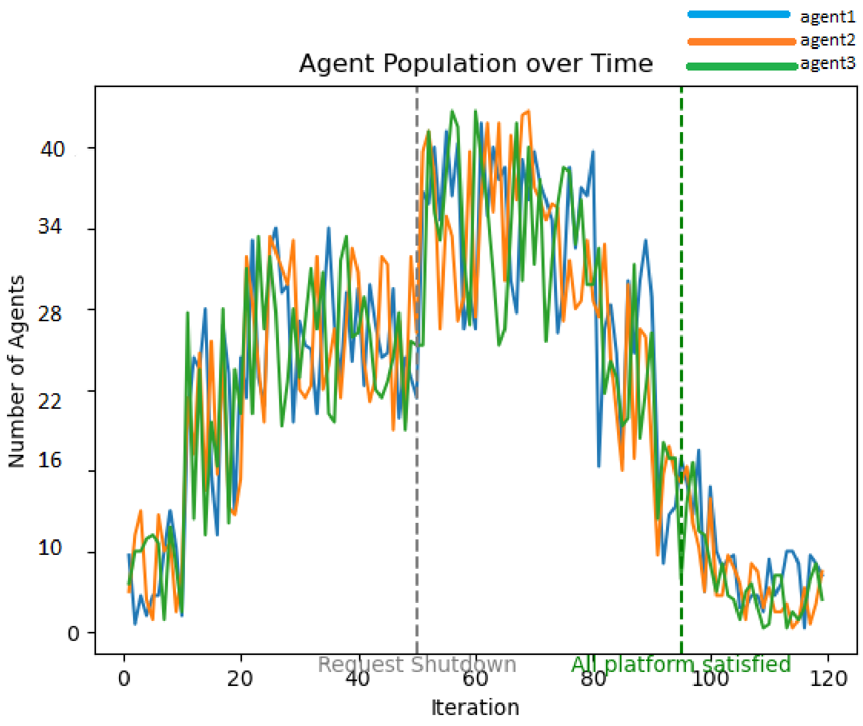


**Fig. 3.43**: Service Time with Queue size = 3 (Grid), 1 iteration = 10 seconds

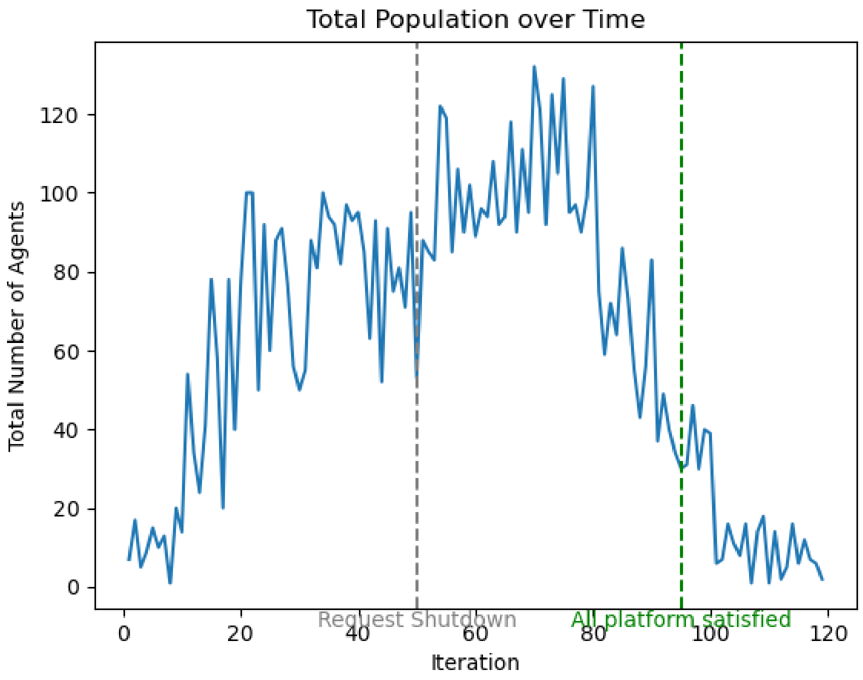


**Fig. 3.44**: Total number of agents and Agent Service Time with Queue size = 3, (Grid), 1 iteration = 10 seconds

It is clear from the graphs that it is important to choose queue size wisely based on the network. If a network size is big, using a slightly bigger queue is recommended.

In this topology, we generated requests for the service of agent1, agent2, and agent3 from iteration 0 to 50. Then we abruptly stop generating requests. The queue size = 3 and 5 have service times of 95 and 86 iterations, respectively. So we are servicing requests quickly if the queue size is slightly bigger.

2. Topology = Line, Average Waiting Time Graph:

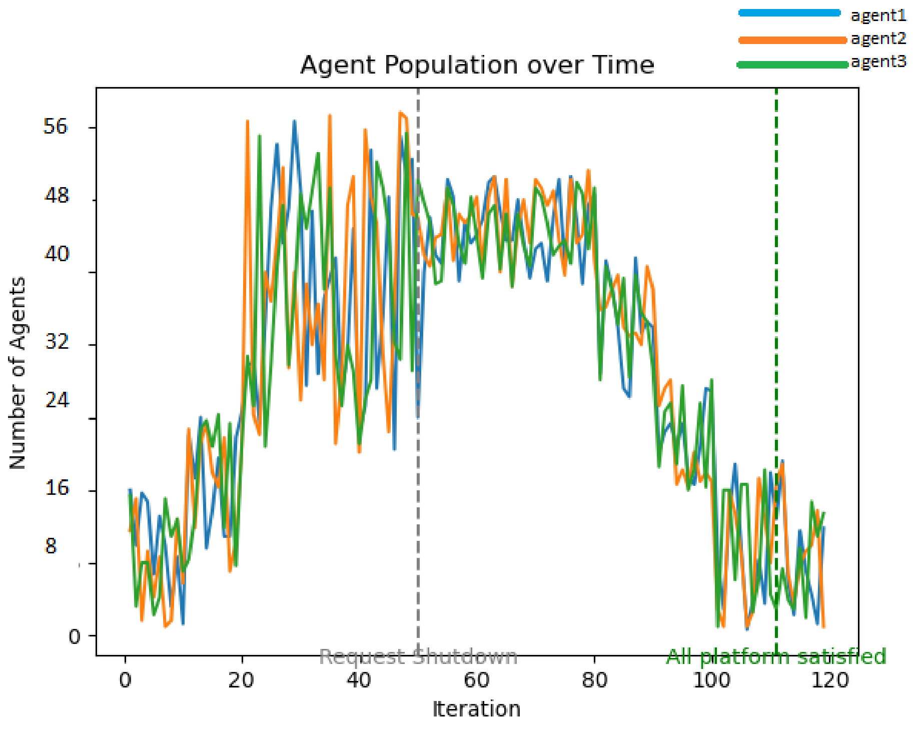


**Fig. 3.45**: Service Time with Queue size = 5, (Line), 1 iteration = 10 seconds

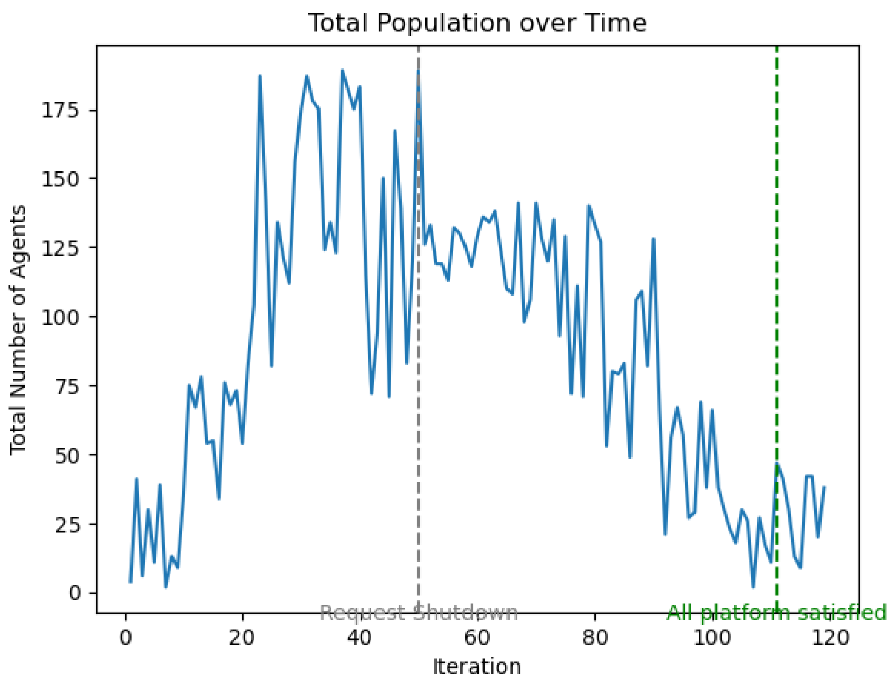


**Fig. 3.46**: Total number of agents and Agent Service Time with Queue size = 5, (Line), 1 iteration = 10 seconds

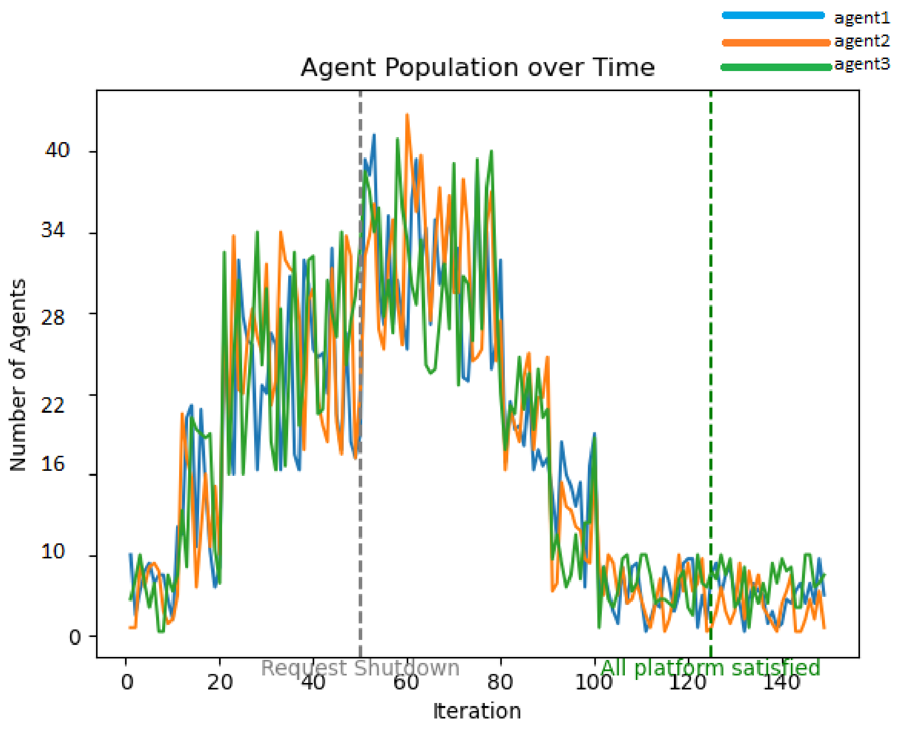


**Fig. 3.47**: Service Time with Queue size = 3 (Line), 1 iteration = 10 seconds

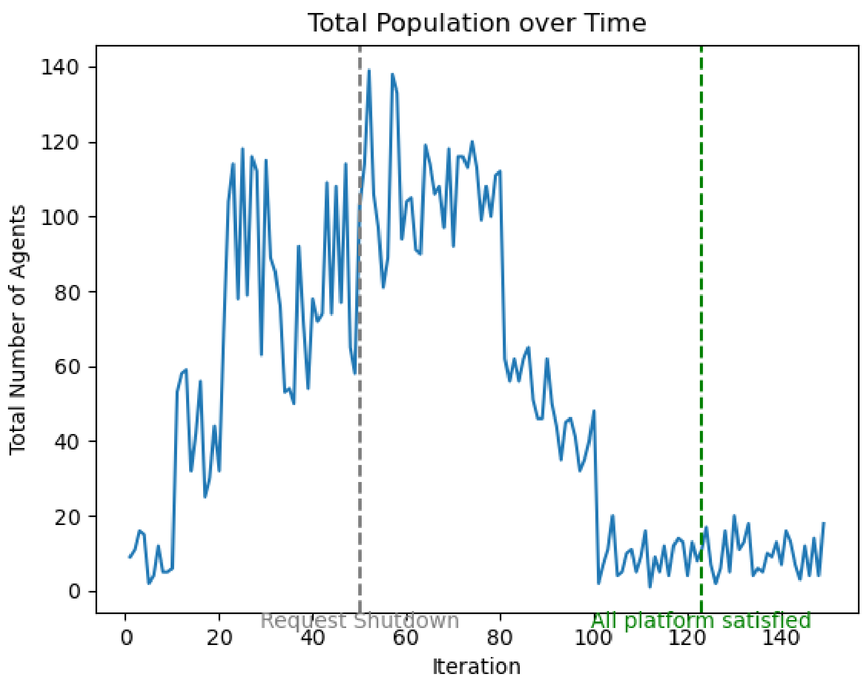


**Fig. 3.48**: Total number of agents and Agent Service Time with Queue size = 3 (Line), 1 iteration = 10 seconds

In this topology, we generated requests for the service of agent1, agent2, and agent3 from iteration 0 to 50. Then we abruptly stop generating requests. The queue size = 3 and 5 have service times of 123 and 116 iterations, respectively. So we are servicing requests quickly if the queue size is slightly bigger.

3. Topology = Mesh, Average Waiting Time Graph:

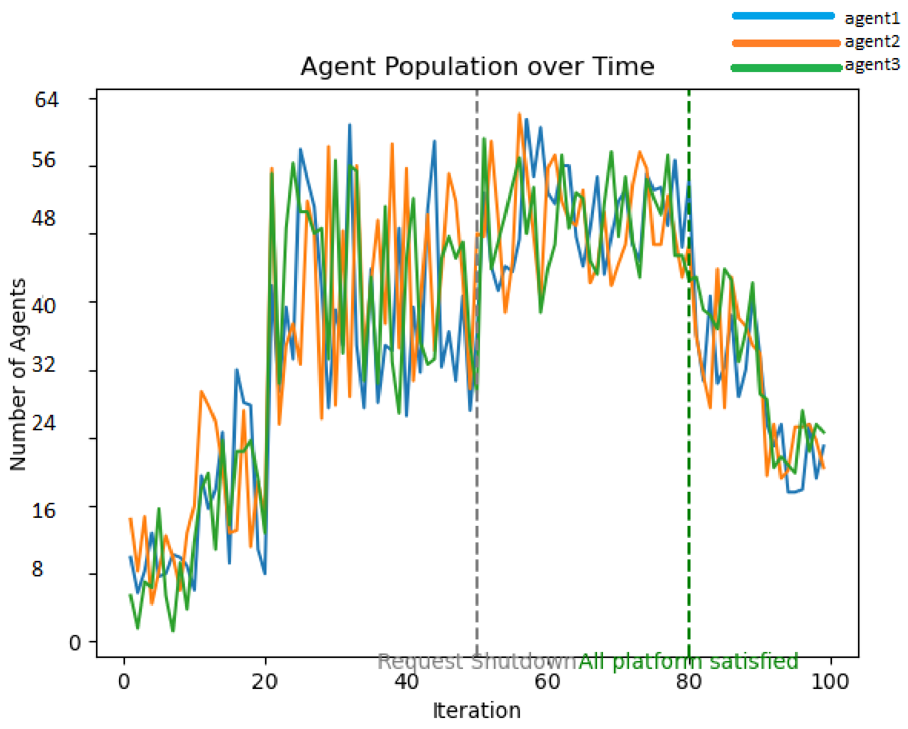


**Fig. 3.49**: Service Time with Queue size = 5 (Mesh), 1 iteration = 10 seconds

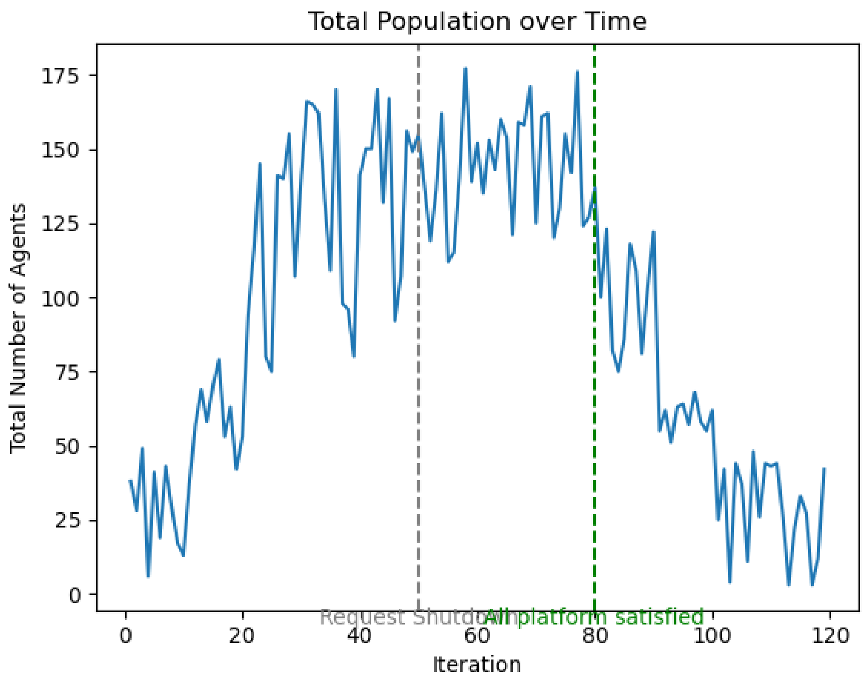


**Fig. 3.50**: Total number of agents and Agent Service Time with Queue size = 5 (Mesh), 1 iteration = 10 seconds

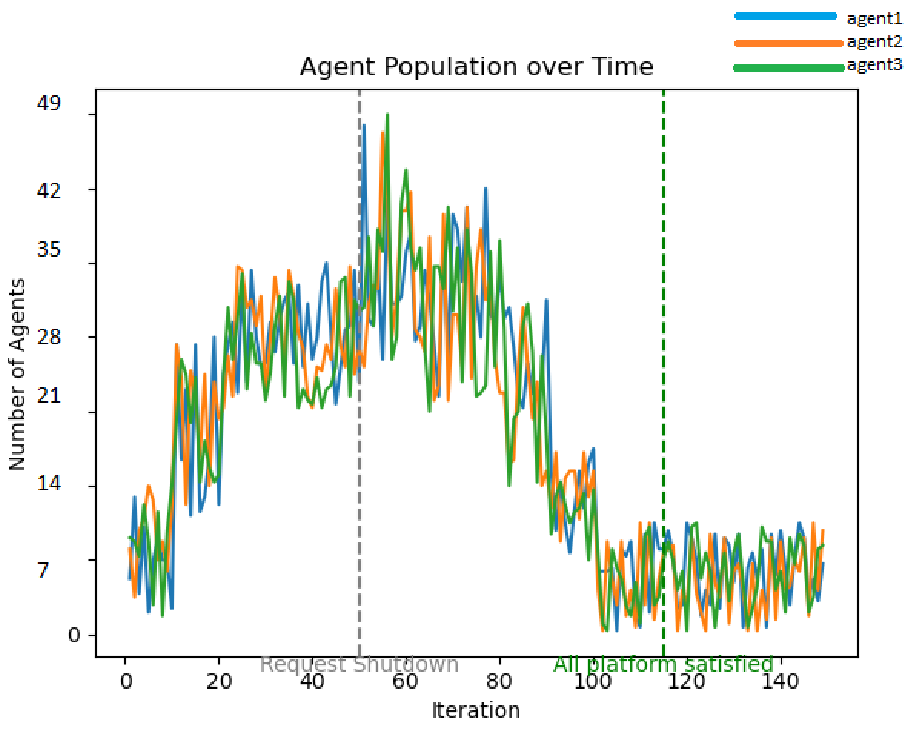


**Fig. 3.51**: Service Time with Queue size = 3 (Mesh), 1 iteration = 10 seconds

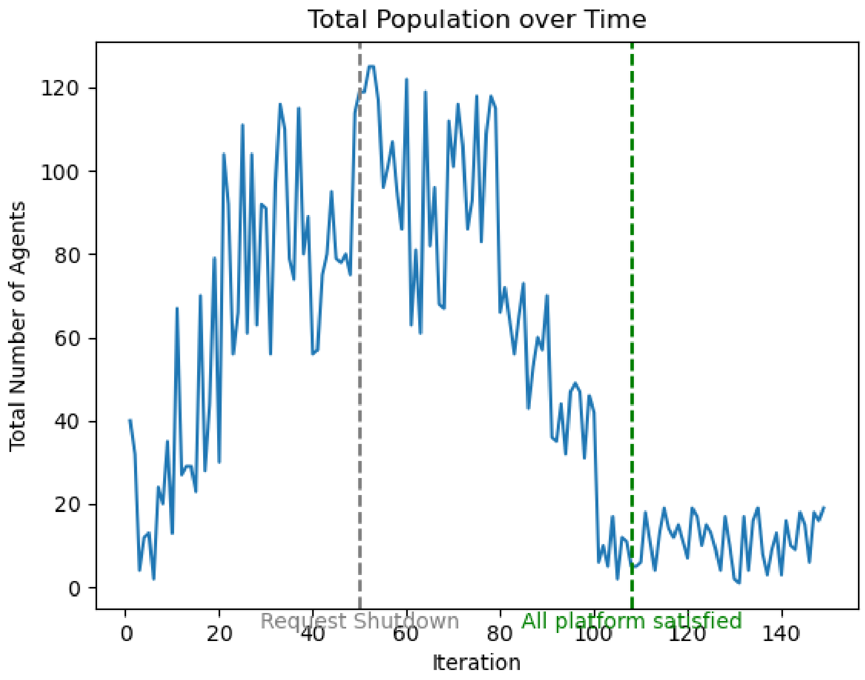


**Fig. 3.52**: Total number of agents and Agent Service Time with Queue size = 3 (Mesh), 1 iteration = 10 seconds

In this topology, we generated requests for the service of agent1, agent2, and agent3 from iteration 0 to 50. Then we abruptly stop generating requests. The queue size = 3 and 5 have service times of 108 and 80 iterations, respectively. So we are servicing requests quickly if the queue size is slightly bigger.

4. Topology = Random, Average Waiting Time Graph:

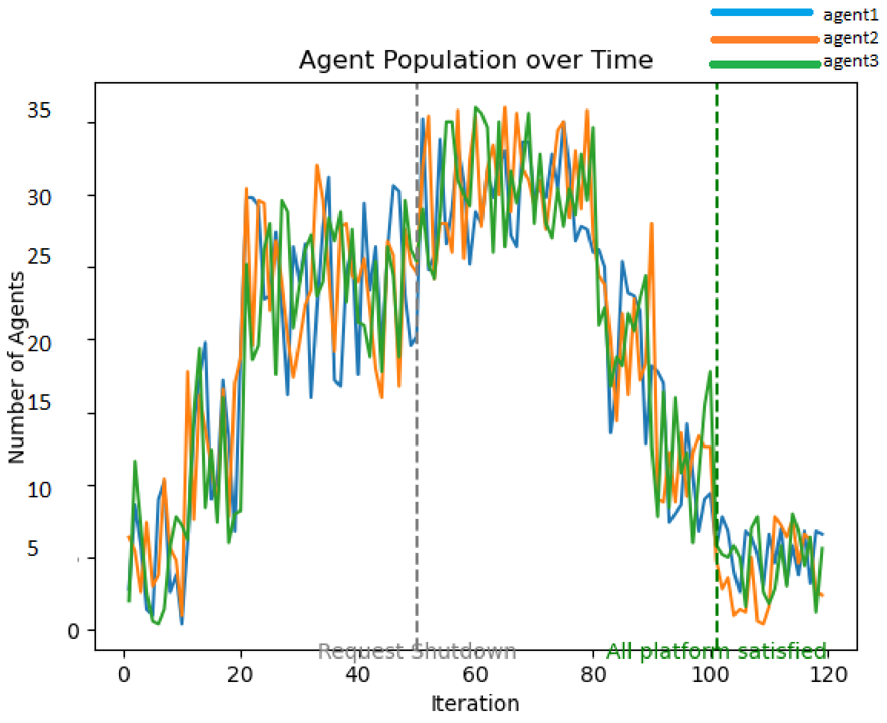


**Fig. 3.53**: Service Time with Queue size = 5 (Random), 1 iteration = 10 seconds

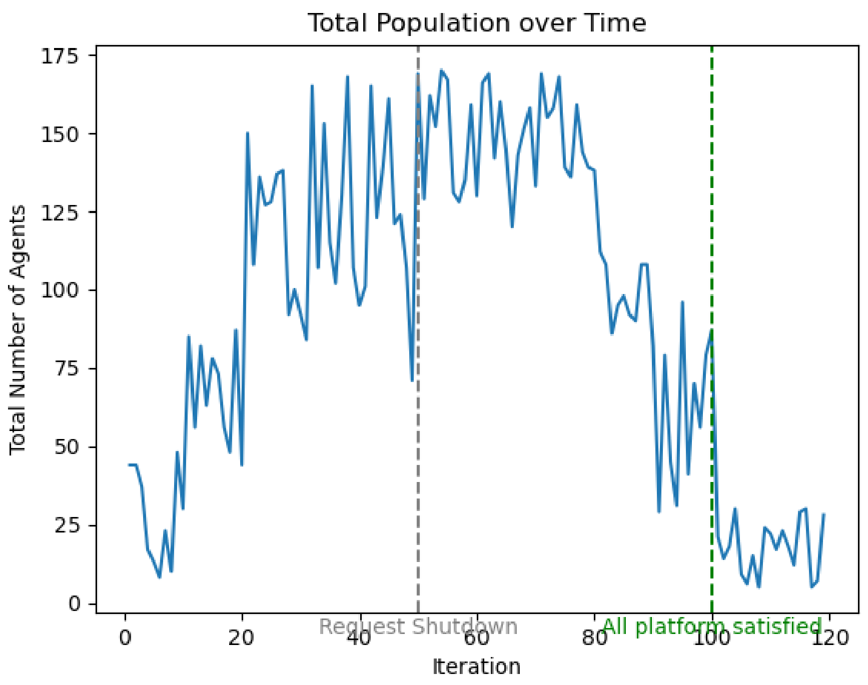


**Fig. 3.54**: Total number of agents and Agent Service Time with Queue size = 5 (Random), 1 iteration = 10 seconds

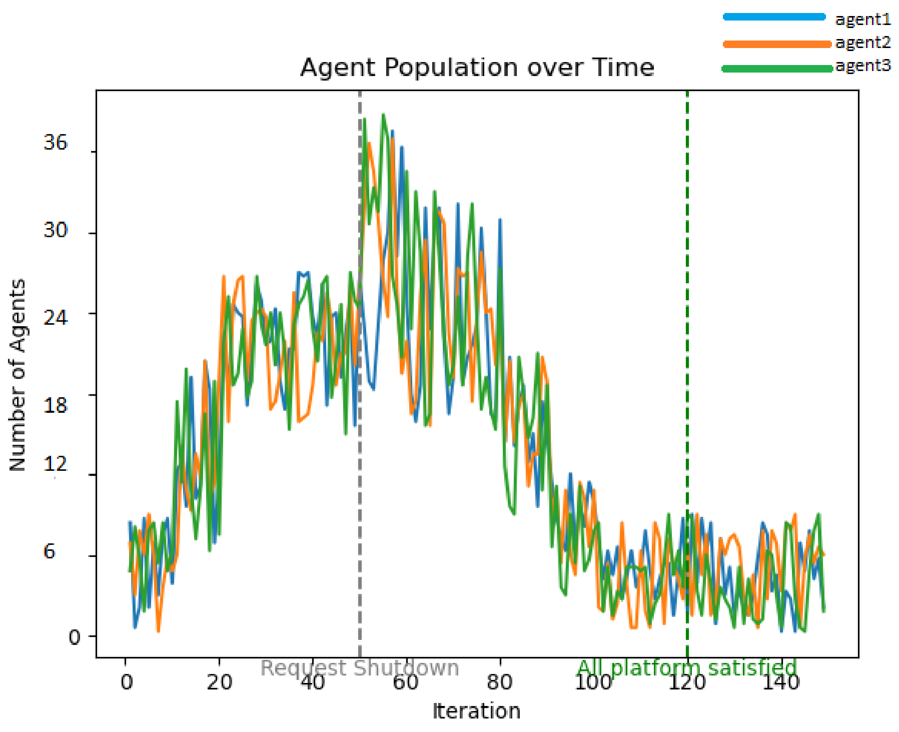


**Fig. 3.55**: Service Time with Queue size = 3 (Random), 1 iteration = 10 seconds

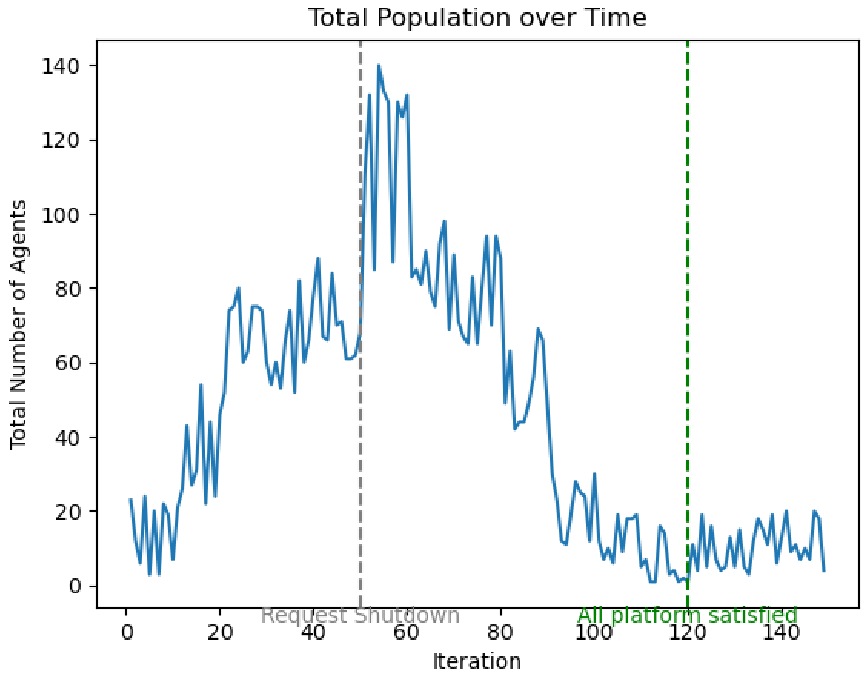


**Fig. 3.56**: Total number of agents and Agent Service Time with Queue size = 3 (Random), 1 iteration = 10 seconds

In this topology, we generated requests for the service of agent1, agent2, and agent3 from iteration 0 to 50. Then we abruptly stop generating requests. The queue size = 3 and 5 have service times of 120 and 100 iterations, respectively. So we are servicing requests quickly if the queue size is slightly bigger.

5. Topology = Star, Average Waiting Time Graph:

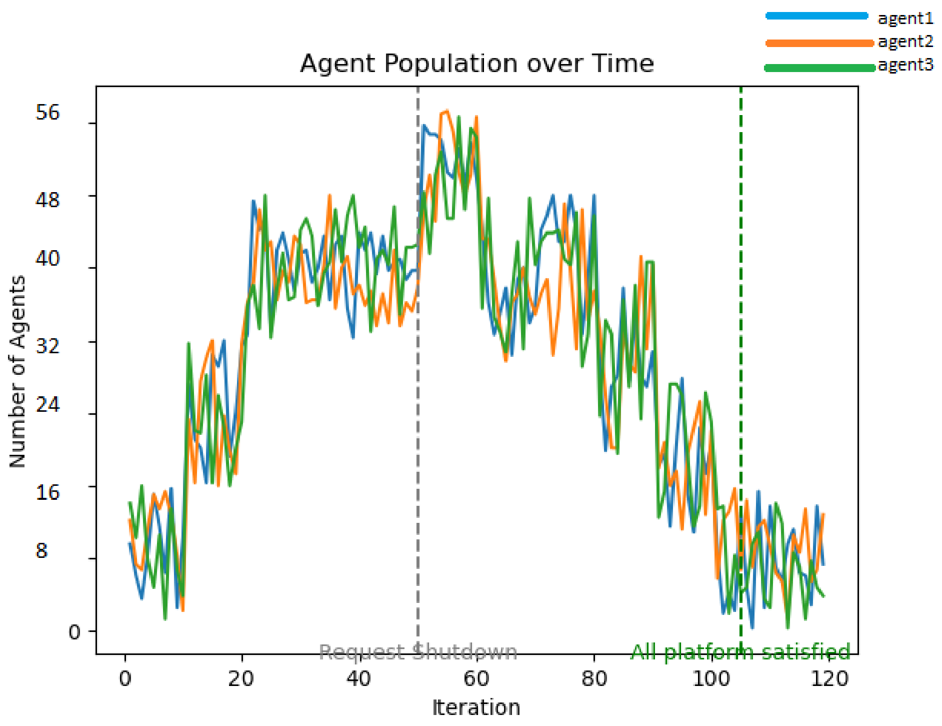


**Fig. 3.57**: Service Time with Queue size = 5 (Star), 1 iteration = 10 seconds

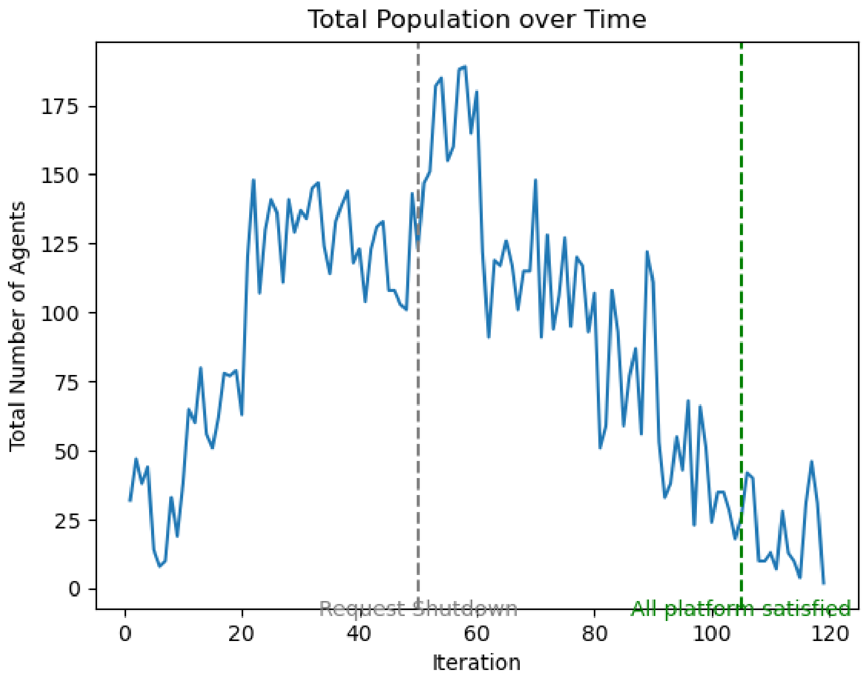


**Fig. 3.58**: Total number of agents and Agent Service Time with Queue size = 5 (Star), 1 iteration = 10 seconds

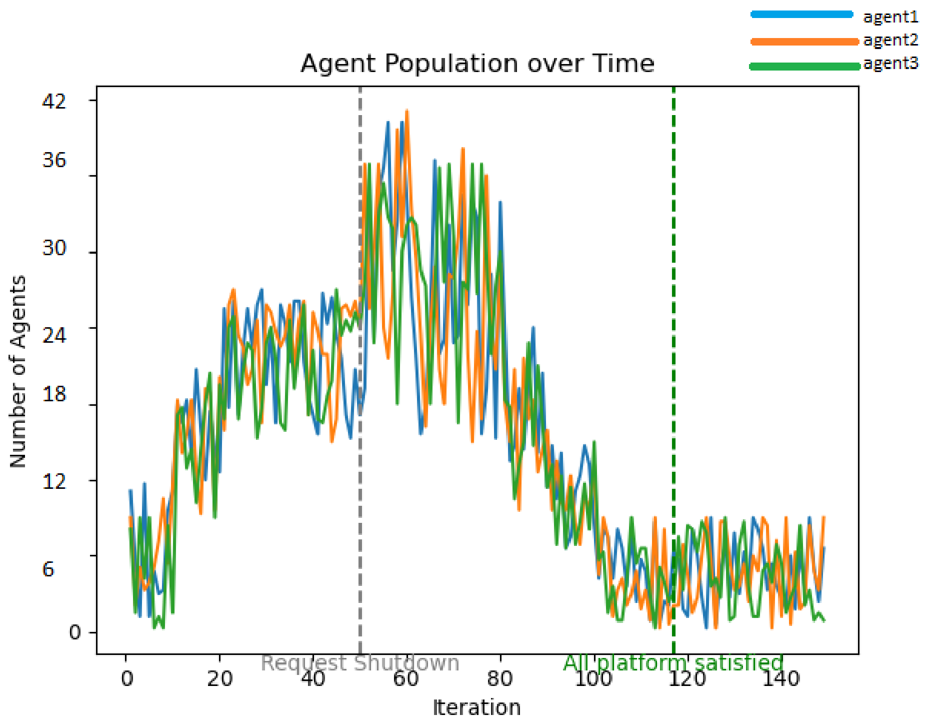


**Fig. 3.59**: Service Time with Queue size = 3 (Star), 1 iteration = 10 seconds

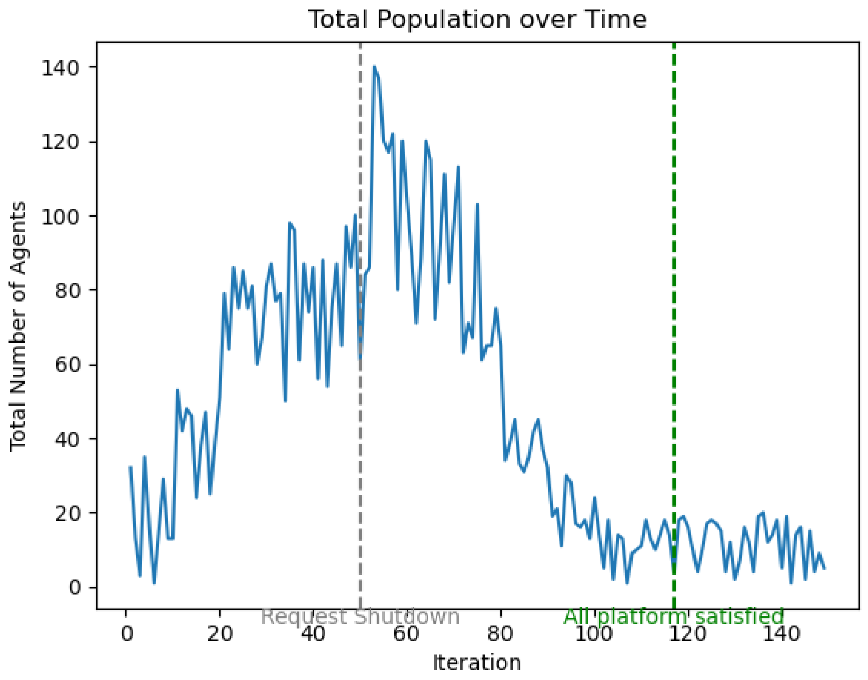


**Fig. 3.60**: Total number of agents and Agent Service Time with Queue size = 3 (Star), 1 iteration = 10 seconds

In this topology, we generated requests for the service of agent1, agent2, and agent3 from iteration 0 to 50. Then we abruptly stop generating requests. The queue size = 3 and 5 have service times of 117 and 111 iterations, respectively. So we are servicing requests quickly if the queue size is slightly bigger.

6. Topology = Tree, Average Waiting Time Graph:

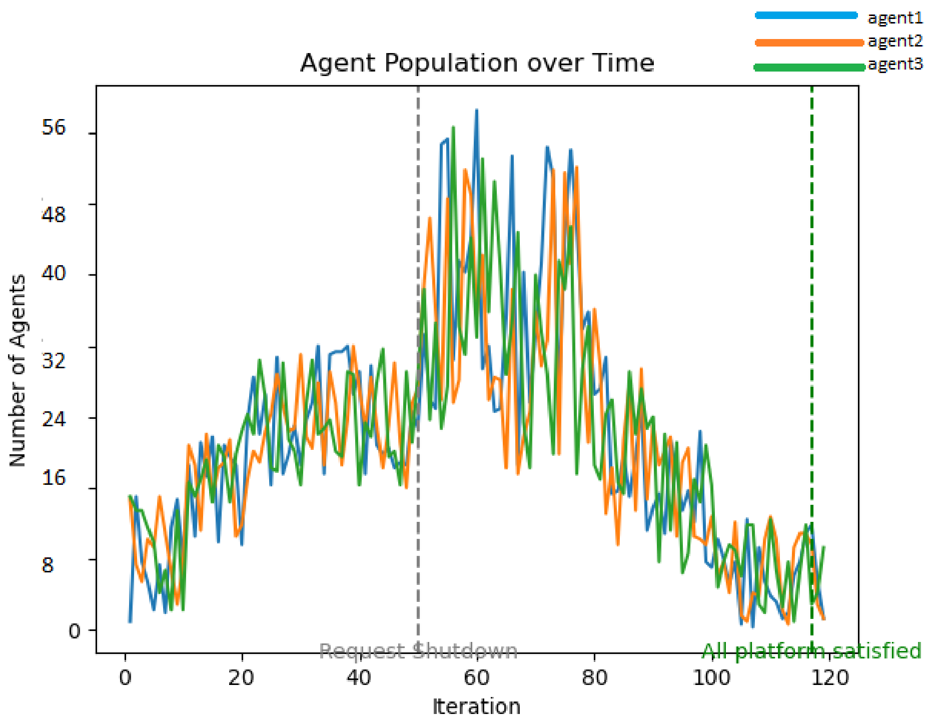


**Fig. 3.61**: Service Time with Queue size = 5 (Tree), 1 iteration = 10 seconds

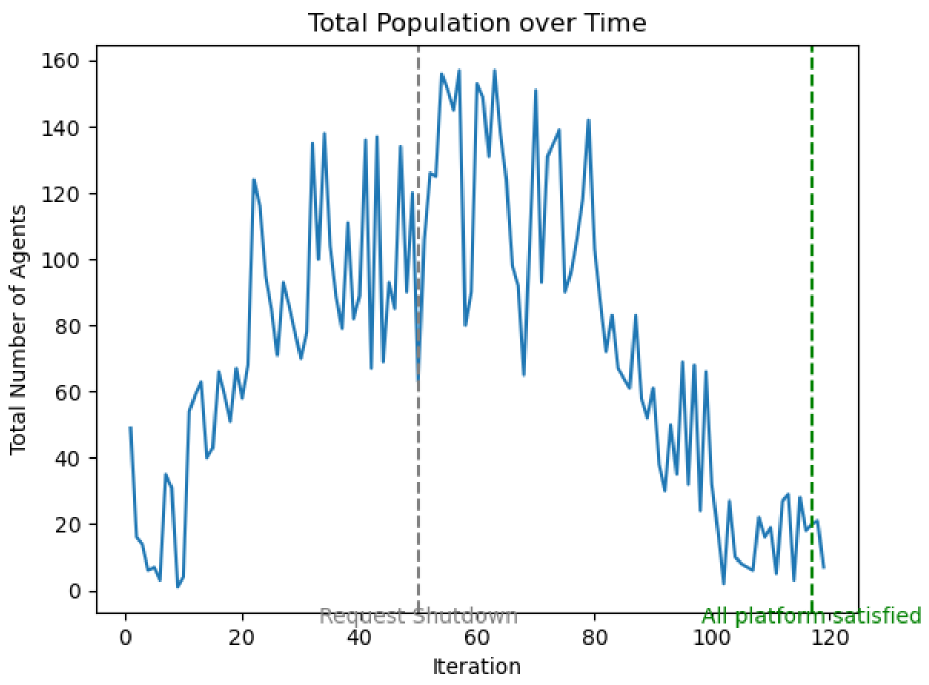


**Fig. 3.62**: Total number of agents and Agent Service Time with Queue size = 5 (Tree), 1 iteration = 10 seconds

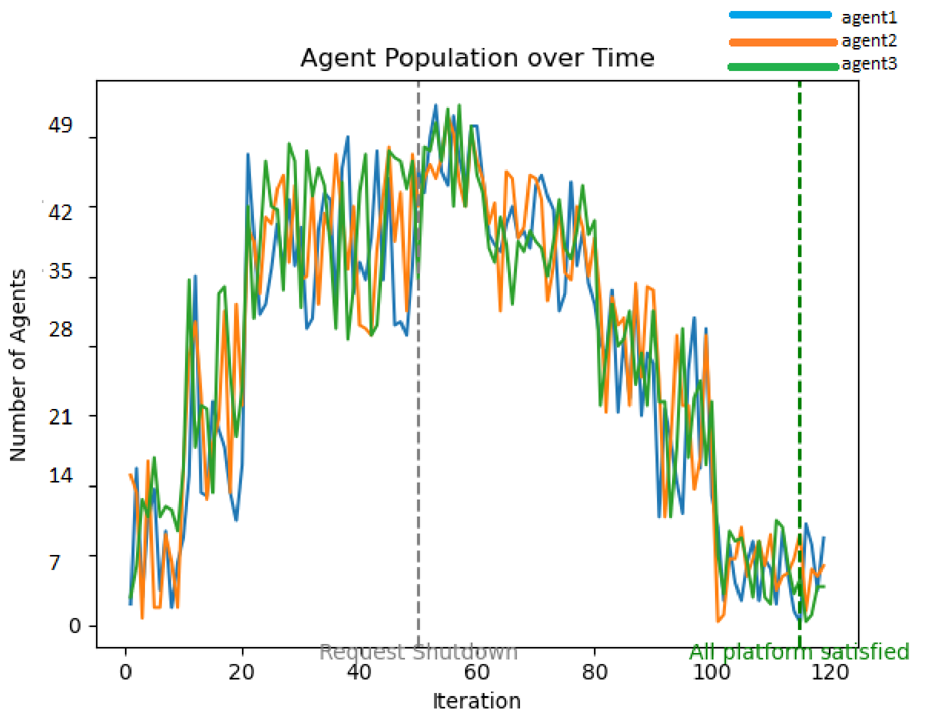


**Fig. 3.63**: Service Time with Queue size = 3 (Tree), 1 iteration = 10 seconds

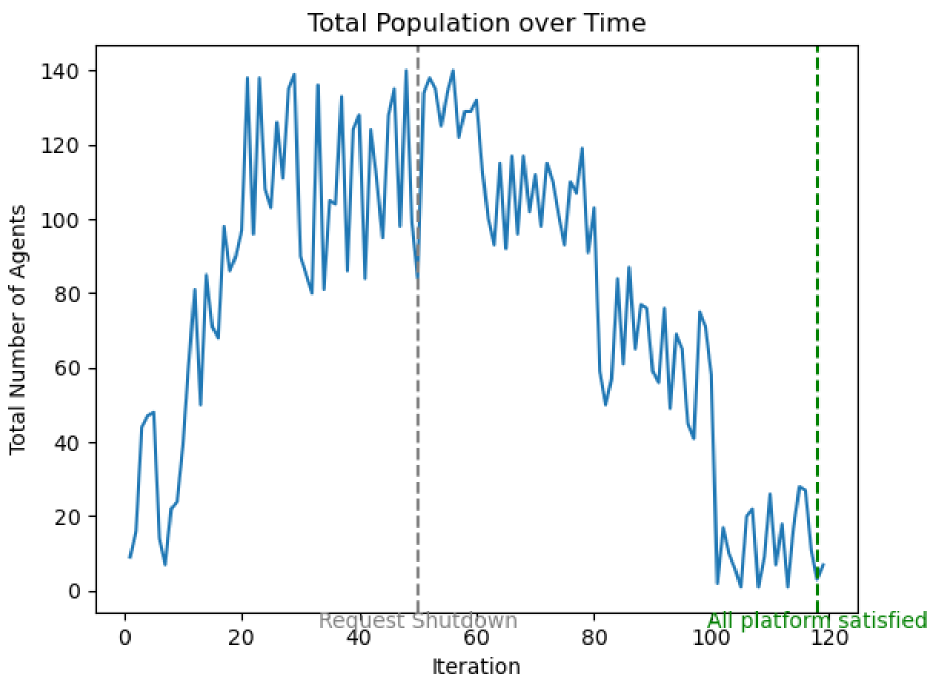


**Fig. 3.64**: Total number of agents and Agent Service Time with Queue size = 3 (Tree), 1 iteration = 10 seconds

In this topology, we generated requests for the service of agent1, agent2, and agent3 from iteration 0 to 50. Then we abruptly stop generating requests. The queue size = 3 and 5 have service times of 119 and 115 iterations, respectively. So we are servicing requests quickly if the queue size is slightly bigger.

## 3.3 Results and Remarks

In conclusion, it has been observed that certain hyper-parameters, such as $\sigma$, $\tau_r$, *and Agent Resource*, need to be adjusted according to the network topology and application demands. A higher value of $\sigma$ is recommended for sparser networks, while denser networks would benefit from a lower value. In turn, the Agent Resource value should also be adapted accordingly, that is, lower cloning resources for denser networks and higher for sparser ones.

For a network with 50 nodes, for a sparser network topology like Line or Tree, we considered the range of values for the parameter $\sigma$ to be between 4 and 7 and the initial cloning resource to be between 40 and 50. It was determined that these values were sufficient to fulfill requests effectively. However, for denser topologies such as Mesh or Star, we adjusted the range of $\sigma$ to be between 3 and 4 and the initial cloning resource to be between 20 and 30. It is important to note that as the number of nodes increases, the hyper-parameters must be appropriately adjusted to ensure optimal performance.

Mesh topologies often experience high delay times due to multiple incoming requests and the execution of rigorous PherCon predicates. Consequently, the cloning controller may take longer to execute.

Tree topologies, on the other hand, tend to satisfy fewer requests at each iteration due to the presence of a single path between any two nodes. As a result, delays in satisfying requests can be significant, especially if the request originates from a leaf node on one side of the tree and the nodes requesting services are on the other side.

In the case of Star topology, the number of satisfied requests at nodes is also relatively low. The agent-wise population graphs indicate that congestion around the center-most node may initially prevent agents from transmitting. Still, their population increases later on, and so does the number of platforms whose requests are satisfied at each iteration.

Finally, with changing values of the queue threshold, we saw how it could affect the waiting time of a node for any particular service. The queue size should be adjusted accordingly. In all the topologies, it is evident that the queue size plays an important role; the more the queue size more the number of satisfied requests. But this queue size should not be increased beyond a threshold. Otherwise, we will get a parabolic case, as the introduction mentions.

# Chapter 4

# Conclusion and Future Work

Chapter 1 provided an introduction to the topics discussed in this report, including Mobile Agents and the significance of cloning in a multi-node environment. Key considerations for building a feasible model were also highlighted.

In Chapter 2, we explored relevant published works that underscored the importance of controlled cloning and the limitations of unregulated cloning models. We introduced the concept of a cloning mechanism, which is detailed in Chapter 3.

Chapter 3 presented our implementation of a viable cloning controller model, offering an architectural overview of the process and the necessary steps to be followed. The implementation was carried out using Tartarus, a framework written in SWI Prolog. The functionalities of the cloning controller were specifically developed in SWI Prolog. Additionally, we presented the experimental setup and graphs depicting the performance of the cloning controller on different network topologies.

The potential of the Cloning Controller extends to robotic environments where groups of robots collaborate to accomplish tasks such as foraging. As the population of agents in the network increases with the addition of new tasks, the Cloning Controller can dynamically adapt the node's functionality based on the environment. Relevant agents can be retained while others are purged to create space. This aspect can be explored further in future work.

This report establishes the importance of controlled cloning and presents a practical implementation of the Cloning Controller, with potential applications in robotic systems and beyond.

# References

[1] W Wilfred Godfrey, Shashi Shekhar Jha, and Shivashankar B Nair. On stigmergically controlling a population of heterogeneous mobile agents using cloning resource. *Transactions on Computational Collective Intelligence XIV*, pages 49–70, 2014.

[2] Hoang Nam Chu, Arnaud Glad, Olivier Simonin, François Sempé, Alexis Drogoul, and François Charpillet. Swarm approaches for the patrolling problem, information propagation vs. pheromone evaporation. In *19th IEEE international conference on tools with artificial intelligence (ICTAI 2007)*, volume 1, pages 442–449. IEEE, 2007.

[3] François Sempé and Alexis Drogoul. Adaptive patrol for a group of robots. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453)*, volume 3, pages 2865–2869. IEEE, 2003.

[4] W Wilfred Godfrey and Shivashankar B Nair. An immune system based multi-robot mobile agent network. In *Artificial Immune Systems: 7th International Conference, ICARIS 2008, Phuket, Thailand, August 10-13, 2008. Proceedings 7*, pages 424–433. Springer, 2008.

[5] Tushar Semwal, Manoj Bode, Vivek Singh, Shashi Shekhar Jha, and Shivashankar B Nair. Tartarus: a multi-agent platform for integrating cyber-physical systems and robots. In *Proceedings of the 2015 Conference on Advances in Robotics*, pages 1–6, 2015.

[6] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *arXiv preprint arXiv:1011.5332*, 2010.

[7] Roch H Glitho, Edgar Olougouna, and Samuel Pierre. Mobile agents and their use for information retrieval: a brief overview and an elaborate case study. *IEEE network*, 16(1):34–41, 2002.

[8] Yingwei Jin, Wenyu Qu, Yong Zhang, and Yong Wang. A mobile agent-based routing model for grid computing. *The Journal of Supercomputing*, 63:431–442, 2013.

[9] Saouli Hamza, Benharkat Aïcha-Nabila, Kazar Okba, and Amghar Youssef. A cloud computing approach based on mobile agents for web services discovery. In *Second International Conference on the Innovative Computing Technology (INTECH 2012)*, pages 297–304. IEEE, 2012.

[10] Tomoko Suzuki, Taisuke Izumi, Fukuhito Ooshita, and Toshimitsu Masuzawa. Biologically inspired self-adaptation of mobile agent population. In *16th International Workshop on Database and Expert Systems Applications (DEXA'05)*, pages 170–174. IEEE, 2005.

[11] Justin Ma, Geoffrey M Voelker, and Stefan Savage. Self-stopping worms. In *Proceedings of the 2005 ACM workshop on Rapid malcode*, pages 12–21, 2005.

[12] Zbigniew Goebiewski, Miroslaw Kutylowski, Tomasz Luczak, and Filip Zagórski. Self-stabilizing population of mobile agents. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[13] Kaizar A Amin, Armin R Mikler, and Venkatesan Iyengar Prasanna. Dynamic agent population in agent-based distance vector routing. *Neural, Parallel & Scientific Computations*, 11(1 & 2):127–142, 2003.

[14] Mohamed Bakhouya and Jaafar Gaber. Adaptive approach for the regulation of a mobile agent population in a distributed network. In *2006 Fifth International Symposium on Parallel and Distributed Computing*, pages 360–366. IEEE, 2006.