The CSI Journal on Computer Science & Engineering

The CSI Journal on
Computer Science and Engineering
Vol. 19, No. 1, 2025
Pages 70-86
doi.org/10.22034/jcse.2025.182048
Regular Paper

DiMAS: A micro-service based approach to create distributed agent-based simulations

Talayeh Riahi[1] Mehrdad Ashtiani[1]

[1]School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

Abstract

An agent-based simulation is an efficient approach to simulate real-world entities that naturally can be mapped to agents and their interactions. This approach compared to the traditional approaches is more flexible and extendable. A lot of centralized simulation toolkits and frameworks exist in which we can create autonomous, self-contained, and self-directed agents interacting with each other as well as their environment. But, besides their numerous advantages, they usually face a serious challenge in simulating large-scale models. In other words, these frameworks cannot provide timely responses to users when the number of agents and their interactions is massive. Also, centralized simulation frameworks cannot execute the simulation scenarios in a distributed form. In this paper, we have designed and implemented a distributed agent-based simulation framework (DiMAS) leveraging the principles of micro-service design. In the proposed framework, we have implemented every agent in the simulation as a micro-service. This will allow the users of the framework to create and execute flexible, reusable, extendable, and large-scale models by easily distributing them to multiple nodes and migrating them later on if it is required. We have performed several evaluation scenarios and the results of these evaluations conclude that models that are simulated by DiMAS are more extendable with more fault tolerance. Also, the simulation is very robust against the existence of a massive number of agents.
Keywords: Agent-based simulation, distributed simulation, micro-service architecture

1. Introduction

As we move forward in the scientific domain, researchers' requirements for modeling large scale and complicated systems grow bigger and bigger. Therefore, we need to design flexible and scalable frameworks for simulating such scientific models. An agent-based simulation is one of the effective solutions for such a flexible and extensible design. As many of the models we need to simulate are concepts that can be mapped to a set of agents that have attributes and behaviors interacting with one another, hence agent-based simulations have gained widespread recognition and application in the simulation domain.

There are many tools and frameworks for designing agent-based models with specific advantages. For example, users can model a system without knowing any programming languages, or they can see simulation results visually through the graphical tools provided. But usually, a considerable drawback is present when using these tools which is their centralized simulation scheme. Agents should be capable of running in parallel and independent of one another. But, in most of the existing frameworks, this parallelism does not exist because the agents are running on a single machine, most of the time using only one of the CPU cores. This problem can also affect the performance and the delivery time of the simulation results. Another problem with the centralized simulation frameworks is their considerable performance drop when running models with a huge number of agents and interactions. These frameworks usually face exponential growth in the running time when executing such models.

For solving the stated problems, distributed simulation schemes are proposed. Distributed simulation decreases simulation time by running parts of the simulation concurrently. It can help increase the system fault tolerance and also the results of simulations are more reliable.

On the other hand, the micro-service architecture is becoming more and more popular these days. In this architecture, large systems are divided into small modules called micro-services. Each micro-service can be implemented differently and run

independently. They can run on multiple nodes and interact with each other in a completely loosely coupled manner. It seems that designing a distributed agent-based simulation, based on the principles of micro-service architecture can potentially be a suitable mechanism to challenge the performance and scalability of massive simulation models. The rationale behind the idea is that we can easily implement each agent as a self-contained micro-service being executed in a container (i.e., Docker) and the whole multi-agent system will become a set of microservices communicating with each other through their APIs. Beyond that, we can use several advantages of micro-service architecture such as the ability of auto migrations of micro-services which leads to the proper load balancing of nodes in case of huge workload.
To examine such a possibility, in this paper, we present an agent-based simulation framework implemented based on the micro-service architecture. This framework is a library that users should integrate into their projects for creating the simulation systems they require. The simulation system is divided into several micro-services, each micro-service containing one or more agents. Every agent also has behaviors, data, and a communication bus that must be declared. Every agent sends and receives messages through the RabbitMQ [1] bus. Agents, as well as their attributes and behaviors, must be implemented and the proposed framework provides a set of interfaces and templates for the user's convenience to do so. Distributed simulation has its challenges, such as the simulation clock synchronization, managing the state of the whole system, queuing behaviors, and incoming messages. DiMAS has built-in functionalities to manage these aspects. DiMAS is divided into five Components based on their responsibilities:

1. The core component contains the main classes and interfaces,
2. The time manager manages the simulation clock,
3. The messenger is responsible for sending and receiving messages,
4. The schedule component handles the actions that must be executed and
5. The controller is a component for gathering the state of each agent.

For system evaluation, we simulate a variety of models and monitor the usage of micro-services
by increasing the number of agents. We also compare these models by two different frameworks and evaluate the results. The results conclude that for models with a large number of agents DiMAS can provide better performance. But, for models with a small number of agents, other approaches are more appropriate. The summary of the major contributions of the proposed simulation framework are:

1. Agents inside the container can migrate to another node in runtime.
2. Usage monitoring of each micro-service is easy. We can acquire and save information in a stream-like fashion.
3. Micro-services are reusable. Therefore, we can implement one agent and its dependencies and run it many times in multiple nodes.
4. The simulation system is extendable. We can add agents in the form of a micro-service at runtime.
5. The simulation system's fault tolerance increases. Failing one agent does not cause the failure of the whole system.

The rest of this paper is organized as follows. Section 2, gives the definitions for the background knowledge required to understand the challenges in implementing distributed systems and their solutions. Section 3, provides a brief overview of the currently used agent-based simulation frameworks, their designs, and their pros and cons. In section 4, we first introduce the general overview of DiMAS's modules and the general abstract design. Then, we explain the details of the architecture and the existing components. In section 5, we evaluate DiMAS by comparing the implemented examples with other frameworks. In section 6, we discuss the advantages, disadvantages, and the practical limitations of DiMAS. Finally, the paper concludes in Section 7.

2. Background

DiMAS uses the concepts and algorithms that we describe in this section. In subsection 2.1, an agent is defined, features of an agent-based model, and communications between agents are explained. In subsection 2.2, we will review distributed simulations and their advantages and synchronization challenges. Subsection 2.3 gives definitions regarding micro-service architecture and its advantages.

2.1. Agent-based simulation

This type of simulation is becoming more and more popular in recent years. In this type of simulation, the system has one or more agents that communicate with the environment and with each other. Agents have various features. The main features of an agent are as follows [2]:

1. Autonomy: every agent has its state and makes the decisions by itself.
2. Reactivity: agents can perceive events from the environment and respond to such events in a planned way.
3. Proactiveness: agents can change their behavior and their responses to the environment based on their goals.
4. Intractability: every agent can interact with other agents based on a communication language and protocol. This communication is performed for sending and receiving messages from other agents in the environment.

Agents have different types. They are grouped by various features such as motivations, role, and goals [3].

Figure 1 shows the typology of an agent. Collaborative agents emphasize cooperation to perform their tasks. Interface agents monitor and observe actions to learn and suggest better actions. Mobile agents are computational software processes that can interact with foreign hosts to gather information. Information agents perform the role of managing, manipulating, or collecting information from many distributed resources. Reactive agents do not possess internal, symbolic models of their environments, instead, act in a stimulus-response manner to present the state of the environment. Hybrids agents are a mixture of other agents and have features of multiple agents.

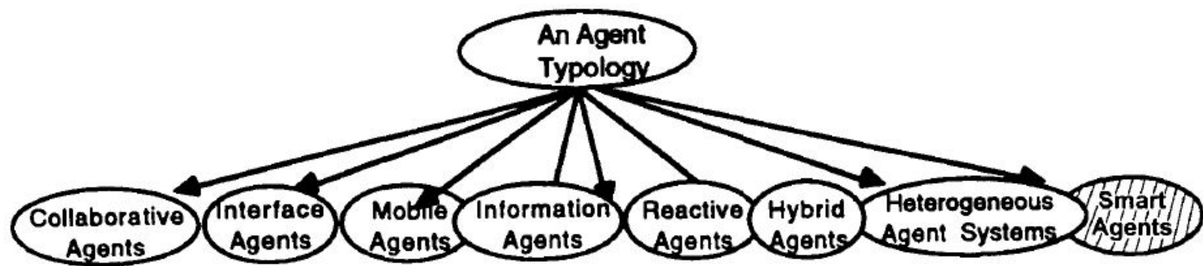


Figure 1. typology of an agent [3]

An agent-based model has three elements [4]:
1. A set of agents and their attributes and behaviors.
2. A set of agent relationships and methods of interaction.
3. The agents' environment.

As stated before, agents interact with each other based on a communication language. So, a
protocol is needed for sending and receiving messages. FIPA [5] is a standard protocol for communication between agents. As shown in Figure 2, this protocol provides a standard structure for sending and receiving messages.

Frame Ontology	envelope FIPA-Agent-Management		
Parameter	**Description**	**Presence**	**Type**
to	This contains the names of the primary recipients of the message.	Mandatory	Sequence of agent-identifier
from	This is the name of the agent who actually sent the message.	Mandatory	agent-identifier
comments	This is a comment in the message envelope.	Optional	String
acl-representation	This is the name of the syntax representation of the message body.	Mandatory	String

2.2. Distributed simulation

To run a distributed simulation, the system must be divided into several independent parts. Each part can be executed on a different node. There are four benefits to executing a simulation program across multiple nodes [6]:

1. Reduced execution time: execution time is reduced by subdividing a large simulation computation into many sub-computations that execute concurrently.
2. Geographical distribution: the execution of the simulation on a set of geographically distributed computer systems is an enabling factor to create virtual worlds with participants physically located on different sites.
3. Interoperability: the possibility of integrating various simulators that execute on machines from different manufacturers running distinct software and hardware specifications.
4. Fault tolerance: the failure of one process does not cause the failure of the whole system.

On the other hand, clock synchronization is one of the major challenges of distributed simulations. In a distributed simulation, there must be an algorithm to guarantee that events are executed in the right order. For doing this, there exist two main approaches [6]:
1. Conservative synchronization algorithm: The logic of these algorithms constructed in a way to preserve the system from executing events out of sort. These algorithms ensure that an event with t_i timestamp is processed only if all the events with t_i-1-time stamps are processed before that. Deadlock avoidance, deadlock detection, and recovery as well as centralized barriers are a few examples of algorithms that can be classified in this category.
2. Optimistic synchronization algorithm: In this approach, algorithms do not guarantee that events will be processed in the order. But they identify and fix the problem when it happens. Anti-message, time wrap, and global virtual time are some of the algorithms in this category. Algorithms in the second approach most of the time demonstrate more efficiency and better performance in general compared to the first. But they are hard to implement and make the system more complicated. DiMAS uses deadlock avoidance using the null message algorithm which is described below.

A. Deadlock avoidance using null messages

This algorithm belongs to the first category of approaches. This algorithm ensures that no deadlock occurs in the system. Assume that each logical process (LP) communicates with other LPs and all the messages that we send, will reach the destination in order. Therefore, every time LP's simulation clock increases it should send a null message to the neighbors. The LP can increase its simulation clock to ti+1, only if all the neighbors have sent their null messages related to ti [6]. The huge number of sent messages is the biggest problem of this algorithm. Although, there are some approaches such as shared memory [7] to optimize the number of messages being sent across the LPs.

2.3. Micro-service architecture

Micro-services are modules that run as distinct processes. A micro-service architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability [8]. The followings are the primary characteristics of micro-services which makes the resulting architecture unique in its nature [9]:

1. The purpose of using micro-services is to divide large software systems into smaller components.
2. Micro-services can be deployed independently of each other.
3. Micro-services have their own data storage. This usually is in the form of a private database or a separate schema in a shared database.
4. Micro-services are self-contained processes or virtual machines.

5. Micro-services communicate via networks based on various protocols such as representational state transfer (REST).

Each micro-service can be implemented with a different programming language and even a technology stack. So, programmers can choose their implementation mechanisms independently of other micro-services. Whenever they decide to change the used technologies, they can perform the change without any changes in other micro-services. Hence, using micro-services make the application modular and modules can be replaced easily. The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack [10].
As stated before, micro-services are self-contained. Therefore, they should be containerized with all their dependencies. The container is a part of the host resources that contain a system with all its dependencies. Docker [11] is a container engine that manages all the containers. Containers created by Docker can communicate with each other via their network ports. Docker images are available in the docker registry [9]. Each system has multiple micro-services. Docker-compose file contains a name, address, and other information about its micro-services. Also, and to enable the micro-services to communicate with one another, there should exist a message broker. Message brokers usually use message queues to manage the incoming messages and their responses. RabbitMQ is one of such message brokers. Exchange queues and their protocols should be defined in RabbitMQ. For sending a message, one micro-service publishes its message on the corresponding queue and another micro-service consumes the message from the queue and responses to it.

3. Related work

There exist a lot of frameworks for agent-based modeling and simulation. Some of them are centralized and some of them are designed to run in a distributed fashion. Centralized frameworks have more popularity and simplicity than distributed ones. But they are not suitable for large scale simulations with a massive number of agents. In the following, we have briefly reviewed some of these frameworks.

3.1. NetLogo

NetLogo is one of the most prominent tools for agent-based modeling and simulation [12]. The simplicity of usage, graphical interface, and visual simulation results are the most important reasons to use this tool. In return, models with complicated concepts are hard to implement. NetLogo interprets the code that is written with its scripting language and runs them. NetLogo is written using Scala [13] and leverages the dependency injection design pattern for better handling the dependencies between packages. The NetLogo engine is single-threaded and users cannot take advantage of multiple processors. But, they provide a Java API for programmers to extend this tool based on their needs [14].

3.2. RePastSimphony

The RePastSimphony framework is also as popular as NetLogo because of the usage simplicity. It is open source and has comprehensive documentation [15]. The old versions of this tool also suffer from all the drawbacks mentioned for NetLogo, but recently, in RepastHPC [16] version, the limit in the number of agents has been improved [17]. RepastHPC distributes agents to multiple processes that are running in parallel and memory is not shared across processes [18]. So, for implementing interactions between agents, they use cross-process communication and synchronization. RepastHPC moves an agent to another process or makes a copy of it to another process to synchronize the state of an environment in a process. This approach makes the whole system complicated. This framework also uses the time warp algorithm for time synchronization that belongs to optimistic synchronization algorithms described in section 2.2.

3.3. GAMA

The GAMA framework is supported by Google. It provides a better graphical appearance and using it is very simple. To use the tool, one has to be familiar with its special scripting language named GMAL, which can be learned using the given tutorial and documentation for this open-source tool [19]. This tool offers a wide range of facilities to users such as creating 3D visualization of models, connecting a database to your application, and generating charts from the results of the simulations. GAMA is made of several eclipse projects which are designed as a plugin or as a feature. A plugin can be seen as a module and a feature is a group of one or several modules that can be loaded. Some plugins like Core, Gmal, GmalUi, and Application are the main plugins. Whereas some plugins like Fipa, Database, and Math are optional [20].

3.4. Zeus

Zeus is an open-source distributed simulation framework [21]. Defining a model in this framework has several steps. At the initial stage, the developer should find candidate agents. In Zeus, agents are task-oriented. So, finding agents is difficult. After that, the programmer should define tasks, attributes of each candidate agents, the relations between agents as well as the protocol of the interaction. After defining all these configuration parameters in Zeus's editor, the code generator component generates the simulation codes automatically.
Zeus uses FIPA for interaction between agents. In the beginning, agents have to register themselves in a name server for saving their name and IP address. The agents' simulation clock must be synchronized with the global clock. This causes the agents to stay idle for a specific period.

3.5. MCMAS

Unlike the CPU, the Graphical processing unit (GPU) has thousands of small cores. So, developing programs with many little tasks that should run in parallel using the computational capacity of GPU will result in a better performance. CUDA [22] and Open-CL [23] are two programming language frameworks that are used for developing programs based on

GPU. The MCMAS framework uses the system's GPU cores to execute the agent's tasks concurrently [24]. The graphical user interface of MCMAS is written in Java. But, Open-CL has been used to run processes on GPU. There is no explanation for transferring messages between agents in the official documentation. But it seems that MCMAS is not a suitable choice for simulating models that need data storage for saving their information. But, for models that deal with a huge number of agents with a limited number of resources is an appropriate framework.

3.6. AgentService

AgentService is a distributed framework that is capable of running simulation models on multiple nodes [25]. The framework is developed using the .NET framework. Each node interacts with other nodes using web services [26]. The agent management system, directory facilitator, message transport system, and system core are the main components of this framework.

The agent management system is the supervisor and the controller of agents and their life-cycle. The directory facilitator maintains a registry of all services offered by agents. The message system provides agents with a communication channel. Message transport system routes exchanged messages and the message that needs to be sent to a different instance of AgentService in another peer. Inter-platform messages are sent using the simple object access (SOAP) protocol and have to be serialized in the XML format. Although the web service-based design makes the simulation more extendable, agents are still unable to migrate to another node at runtime for load balancing purposes.

3.7. D-Mason

Mason is a centralized agent-based framework written by java [27]. Mason has three layers. The utility layer which contains the simulation logic and can be used for implementing the simulation logic. The Model layer consists of a discrete-event schedule and schedule utilities. The Visualization layer permits GUI-based visualization. As stated before, centralized frameworks have problems with large-scale simulations. Thus, a distributed expansion of Mason is released called (D-Mason) [28]. D-Mason is based on a master/worker's paradigm. Master assigns a portion of the whole computation to each worker. Each worker simulates the agents assigned to it and sends back the results of the computation to the caller master. D-Mason partitions the space to be simulated into regions and assigns each region with agents contained in it to a worker. Each agent can migrate only between neighboring regions. D-Mason uses the Java management system for transferring messages between agents [29].

3.8. Jade

Jade is a powerful and popular framework for distributed simulations [30]. This framework uses JVM for running simulations. Jade allows you to create multiple containers on multiple nodes and distribute the execution on multiple nodes. But only one instance of JVM can be executed on each node. So, the framework cannot use all the resources of a node for the sake of distribution. Jade uses FIPA for transferring messages. Jade consists of a primary main container that can hold agents like other containers. Also, it contains two special agents [31]: (1) AMS which is the only agent able to manage the platform itself such as starting and killing agents or shutting down whole platform, (2) DF agent provides the yellow pages service where agents can publish the services they provide and find other agents providing the services they need.

3.9. SASSY

SASSY provides a middleware between an agent-based API and a parallel discrete event simulation kernel which is written in Java [32]. SASSY provides a user interface for implementing the agents as logical processes (LP). The simulation kernel is implemented as a Java package. The kernel has two major processing elements: (1) The first one is MasterPE which handles the whole system process and (2) a processing element which is called WorkerPE. Each WorkerPE is responsible for receiving and sending messages. MasterPE creates the WorkerPE, processes configuration information, and allocates LPs to WorkerPEs. So, it has the main role and if it fails, the whole system fails. SASSY has a built-in monitoring and steering architecture for controlling and monitoring the system (S/M). Users can control and observe the kernel features such as load balancing, cloning, and merging of simulations. SASSY uses an optimistic time synchronization algorithm which can improve the performance of the simulation.

3.10. HLA Grid RePast

High-Level Architecture (HLA) [33] is a standard for distributed simulations which defines a software architecture for modeling and simulation. HLA is designed to provide reusability and interoperability of simulation components. Grids are a form of distributed computing that provides an opportunity for large-scale distributed simulations. Grid technologies allow for the management of distributed computing resources. HLA_Grid is an infrastructure designed to extend the HLA to the Grid environment and is written in Java. HLA_RePast is a middleware that permits the integration through an HLA federation of multiple instances of RePast [34]. HLA_Grid_RePast is a middleware for executing distributed large-scale RePast models on the Grid. These frameworks use conservative synchronization time algorithms which can potentially limit their performance. Based on the performed experiments, it is evident that the time spent for establishing all the TCP connections on a WAN is more than half of the entire simulation time. But, by using a message aggregation approach the authors have tried to improve the overall performance.

3.11. Cluster of Workstations

The work of Popov et al. represents an approach for the parallel implementation of discrete-time agent-based simulation models which allows for the adaptation of a sequential simulator for performing large-scale simulations on a cluster of workstations [35]. Each simulation can have one or many threads and each thread communicates via synchronous message passing through different ports. Oz [36]

is a concurrent, dynamically typed, and multi-paradigm programming language that offers symbolic computation using a variety of data types and automatic memory management. Mozart [36] is an implementation of the Oz. Authors have used Oz and Mozart to run parallel simulations. Each simulation in their work has one or many workers. Each of the workers holds different data partitions. This approach can be used in a single computer or multiple connected computers. However, it cannot be used in multiple distributed computers through the internet.

In Table 1, a summarization for the comparison of the related work is given. The reviewed literature is compared in terms of the capability for parallel execution, running on distributed networks, using load balancing, the type of messaging infrastructure they are using and reusability (i.e. how easily the framework or its independent components can be integrated within a new context and a respective simulation environment).

	Parallel Execution	Network Distribution	Load Balancing	Messaging	Reusable
NetLogo	No	No	No	-	No
Repast symphony	No	No	No	-	No
GAMAM	No	No	No	-	No
Zeus	Yes	Yes	No	TCP connection	No
MCMAS	Yes	No	No	Not Mentioned	Yes
AgentService	Yes	Yes	Yes	SOAP	Yes
D-Mason	Yes	Yes	Yes	Configurable	No
Jade	Yes	Yes	Yes	IIOP - HTTP	Yes
SASSY	Yes	Yes	Yes	HTTP protocol	No
RePast HPC	Yes	No	No	Shared-memory	Yes
HLA Grid Repast	Yes	Yes	Yes	TCP connection	Yes
Cluster of workstations	Yes	No	No	Mozart system	Yes
DiMAS	Yes	Yes	Yes	Configurable	Yes

4. The proposed simulation framework

Designing a framework has its unique challenges. The design of the framework should be performed in a way in which it provides the required facilities for users making their work easier while being flexible enough so that users do not feel unnecessary limitations. One of the first decisions to implement the framework is to choose the programming language. The latest version of the visual studio supports Docker and micro-service-based architecture. Besides, to leverage the advantages of object-oriented programming, C# is used alongside.Net core framework.

4.1. The general framework

In Figure 3, the general overview of the primary components of DiMAS is displayed. As mentioned, each agent is defined as an independent micro-service. The agents have to interact with each other by a message broker transmitting their messages. In this framework, we use RabbitMQ for transferring the messages based on a publish/subscribe pattern [37]. Also, a separate micro-service namely the controller is designed for gathering information and the results of the execution of other micro-services.

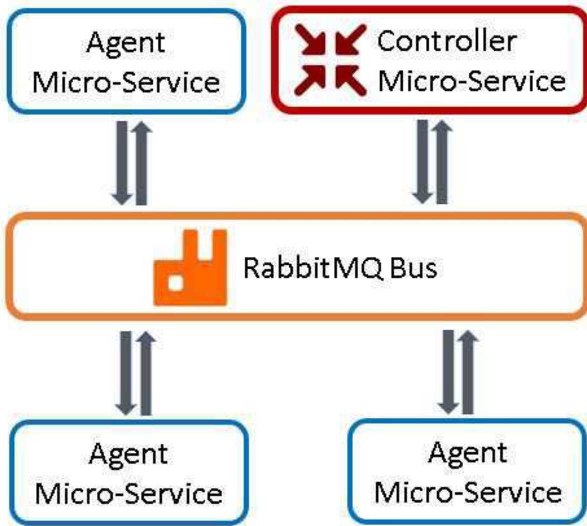
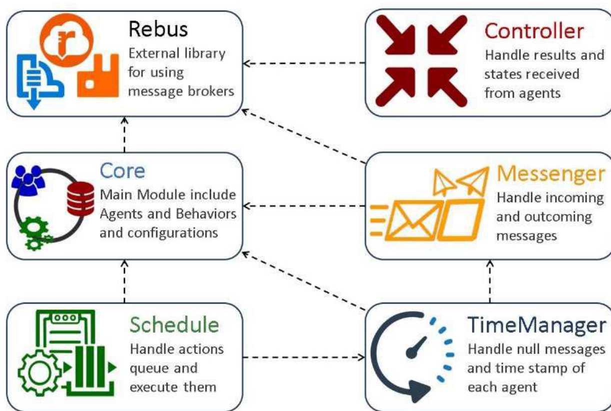


Figure 3. The general structure of the system

It is worth mentioning that the RabbitMQ library is not used directly in DiMAS. Instead, we use the Rebus [38] library that contains a multi-message broker's library such as RabbitMQ and Azure service bus. Therefore, every time we decide to change our message broker, we can do it flexibly. Figure 4 shows DiMAS components and dependencies between them. As can be seen, we have tried to keep the coupling among the components as low as possible to make it more extendable and flexible.



Fi Figure 4. The DiMAS's components

As it is shown in figure 4, DiMAS has five components (excluding the Rebus external library used for message brokers). The core is the main component, enabling the users to implement agents and their behaviors as well as the corresponding data structures. Messenger component is responsible for publishing messages to RabbitMQ queues through the Rebus library while receiving and handling the related messages from the queues. The job of the schedule component is to schedule the actions currently in the queue waiting to be executed at the right time. Finally, the TimeManager component handles the simulation clock and manages and synchronizes the time across the whole simulation. For gathering information about each agent, users should implement the interfaces provided in the Controller component.

Figure 5 shows the activity flow of an agent. Every agent will perform its behaviors based on the following state cycle. In the meantime, messages might be received from other agents that affect the flow of execution in the agent. If the incoming message is a null message, TimeManager will be responsible for handling the message. Otherwise, the message goes to queue to be executed at the right time and the scheduler component will be responsible for the correct scheduling. In the following, we will describe all the introduced components in detail.

4.2. Core component

The SystemConfiguration class is in the Core component. It contains the default configuration of the system, such as RabbitMQ connection and bus initialization. Therefore, this class is the first class that must be instantiated and initialized. Every agent that the user wants to create must implement the IAgent interface which is shown in Figure 6. This interface has the default attributes and method signatures. AgentBase is a class that implements the IAgent interface and its methods. So, users' Agents can inherit from AgentBase class or implement the IAgent interface directly. Each agent has a data class. This class has only one instance for each agent and contains all the data that the agent needs. The IBehavior interface provides the execute method which can be overridden to implement the logic for execution. Each behavior of the agent should implement this method. Behaviors go to the scheduling queue as an action, and when the time is right, the execute method is invoked. The proposed framework uses the publish/subscribe pattern for sending messages. Therefore, we need a class for subscribing and unsubscribing agents' interests toward other agents or events. Subscription class is responsible for this operation.

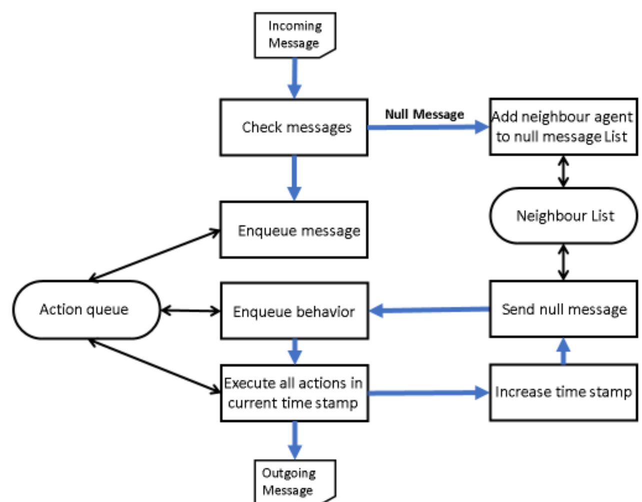


Figure 5. The activity flow of an agent

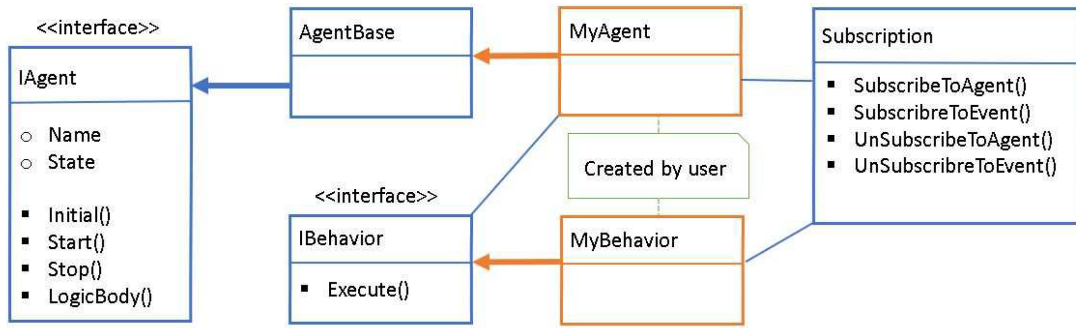


Figure 6. The relations between the Agent classes and the behavior interface

4.3. TimeManager Component

The algorithm used for handling the simulation clock in DiMAS is deadlock avoidance using null messages which are discussed in section 2. TimeSpan class is holding the current value of the simulation clock. TimeController class is responsible for managing the simulation clock. There is a list of neighbors in this class. Every time a null message arrives, this class with the help of the neighbor's list decides about increasing the simulation clock. Every time the simulation clock is increased, TimeController publishes a null message to inform other neighbors. Also, the Scheduler component executes all the actions that are queued for execution in this time step.

4.4. Messenger Component

This component is responsible for all the actions about sending and receiving messages. As seen in figure 7, IMessage is the interface that shows the structure of the messages. This interface has a publish method that can be implemented in a customized way. MessageBase class implements this method for implementing the default algorithm. If this implementation is not enough for users, they can create a class that inherits from the MessageBase and change everything they need. Another approach will be to create a new class that implements the IMessage interface with any attribute that is required and write a customized algorithm for publishing messages in the publish method. When a message has arrived, the Handle method from EventListener class acts as the callback. This method checks the message and if it is a null message then sends it to the TimeManager to be handled. Otherwise, it will Enqueue the message to be executed at the right time. In the time of execution, the Handle method from IMessageHandler implemented by the user is invoked.

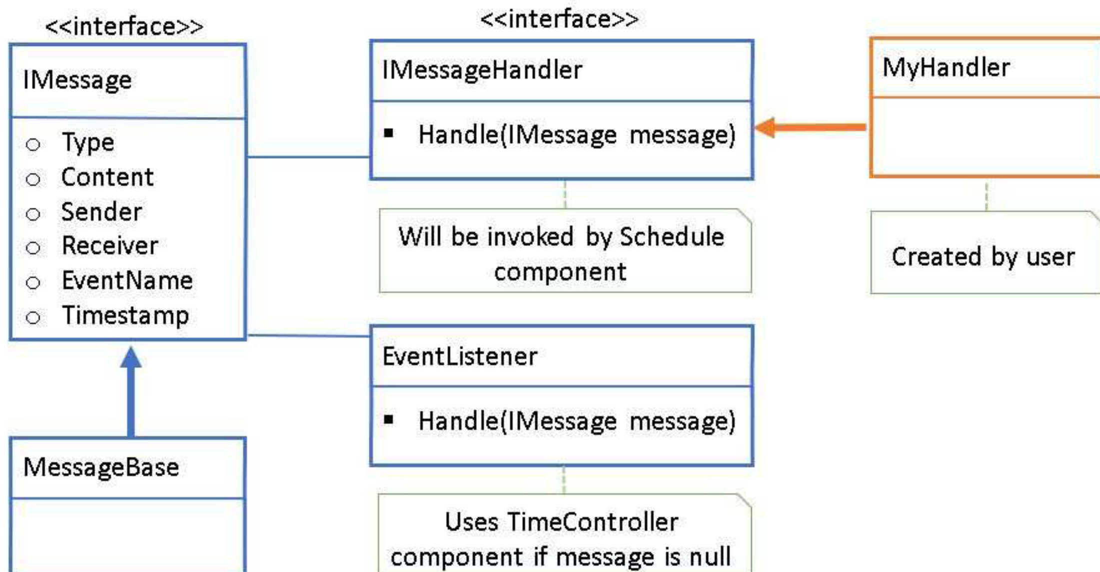


Figure 7. The simplified diagram of Messenger Component

4.5. Schedule Component

The scheduler component has a queue that holds actions ordered by the Execution period shown in Figure 8. When the

correct time comes, certain actions need to be executed. Thus, if the action is a behavior, then its execution method is invoked. But, if the action is a message, then the handler method is invoked. After all the actions were executed and all the neighbors were sent null messages, the TimeManager component increases the time stamp and the cycle continues.

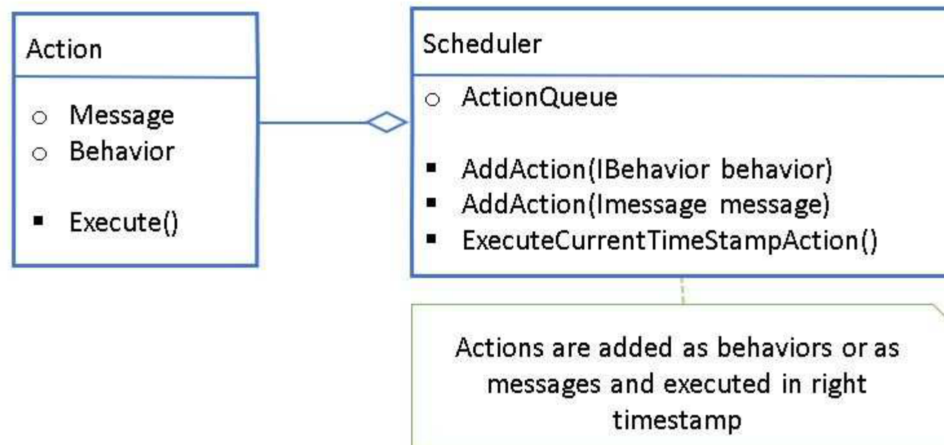


Figure 8. The action queue in the Schedule component

4.6. Controller Component

This component is independent of other DiMAS components. Its responsibility is handling the state of the whole system. StateController class performs the initial configurations and establishing the required connections. In this component, there exists an IMessageController interface that provides a handle method. This method is invoked when a message arrives. The method should be implemented by the user.

5. Evaluations and comparisons

To evaluate the proposed framework, an agent-based simulation model has been implemented using DiMAS. Two models were simulated for 6 times with an increase in the number of agents. For each simulation, system usages per micro-service were captured. Also, for comparing the results of DiMAS with other frameworks, the same simulation model was implemented by three other frameworks and the results were compared and reported.

5.1. Monty Hall

The implementation contains eight micro-services in which one of them is related to RabbitMQ and one of them is the controller micro-service. The model consists of 50 agents which are divided into six micro-services. As can be seen in figure 9, the simulation runtime is 17 seconds. Also, the CPU usage of controller micro-service and RabbitMQ is higher than the agent micro-services. Also as seen in figure 10, the memory usage of RabbitMQ is higher than other micro-services.

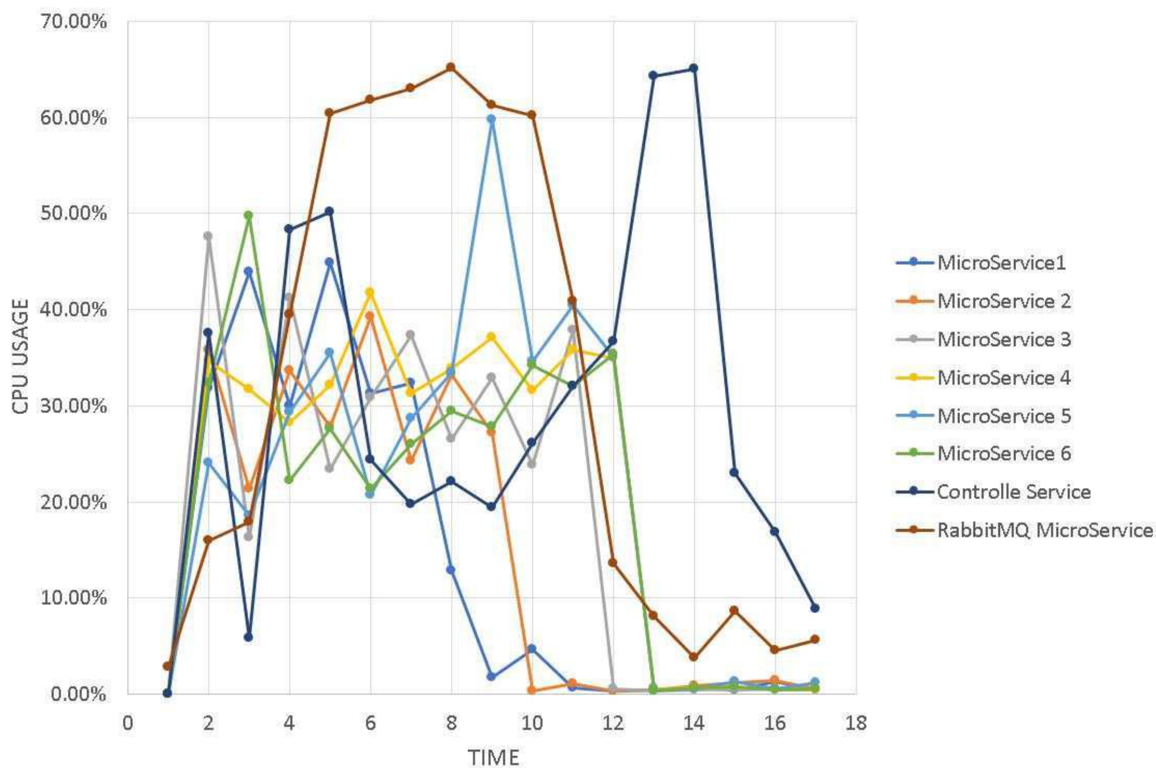


Figure 9. The CPU usage diagram for each micro-service

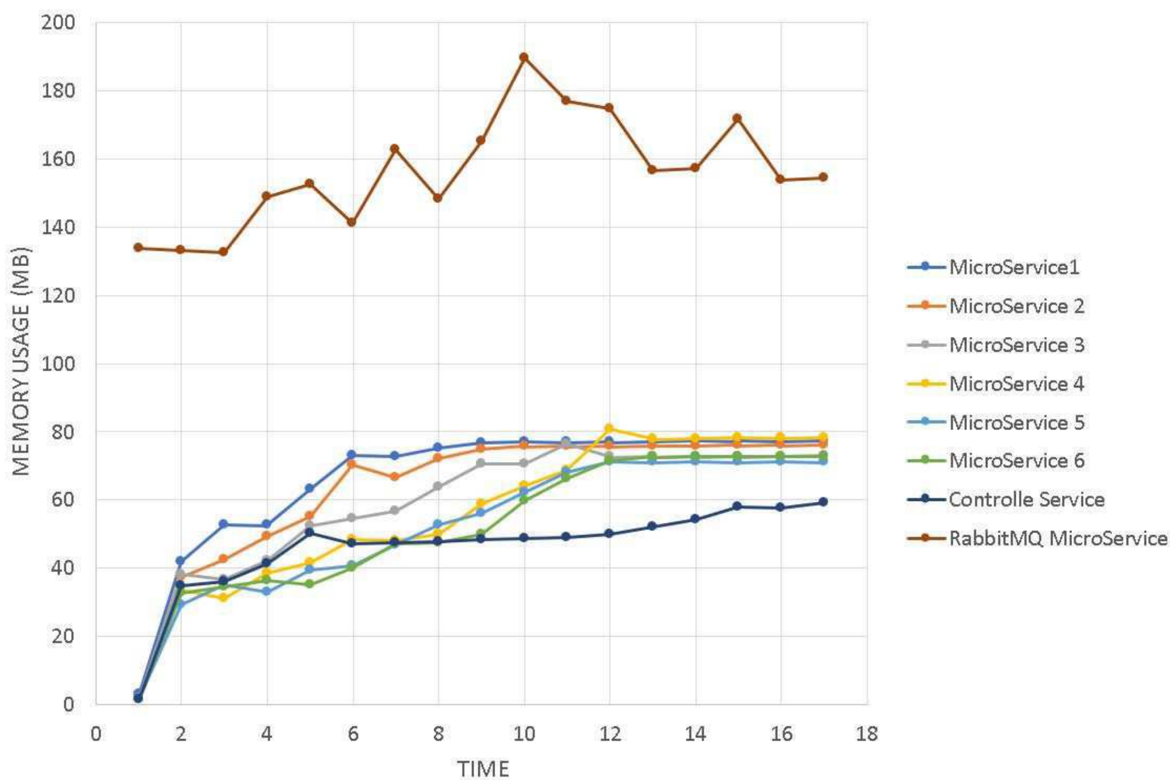


Figure 10. The memory usage diagram for each micro-service

Mont Hall model was implemented by NetLogo [39], GAMA [40], and Jade. In the proposed framework, at least from a theoretical point of view, there is no limit for the number of agents. Jade is a framework that can run a simulation on multiple

containers with the help of JVM. As mentioned in [41], only one JVM can be executed at each time step in a node. So, in a single node, Jade does not differ from other centralized frameworks. On the other hand, the Docker technology can use all CPU cores.

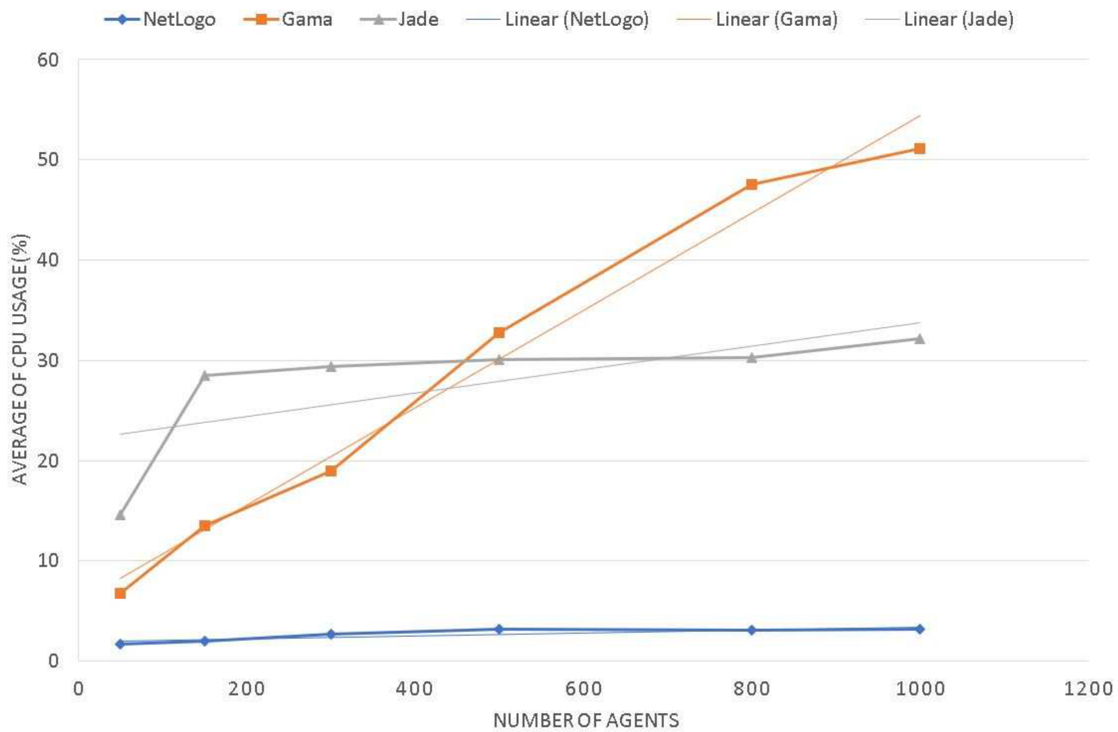


Figure 11. The average CPU usage per number of agents for the compared frameworks

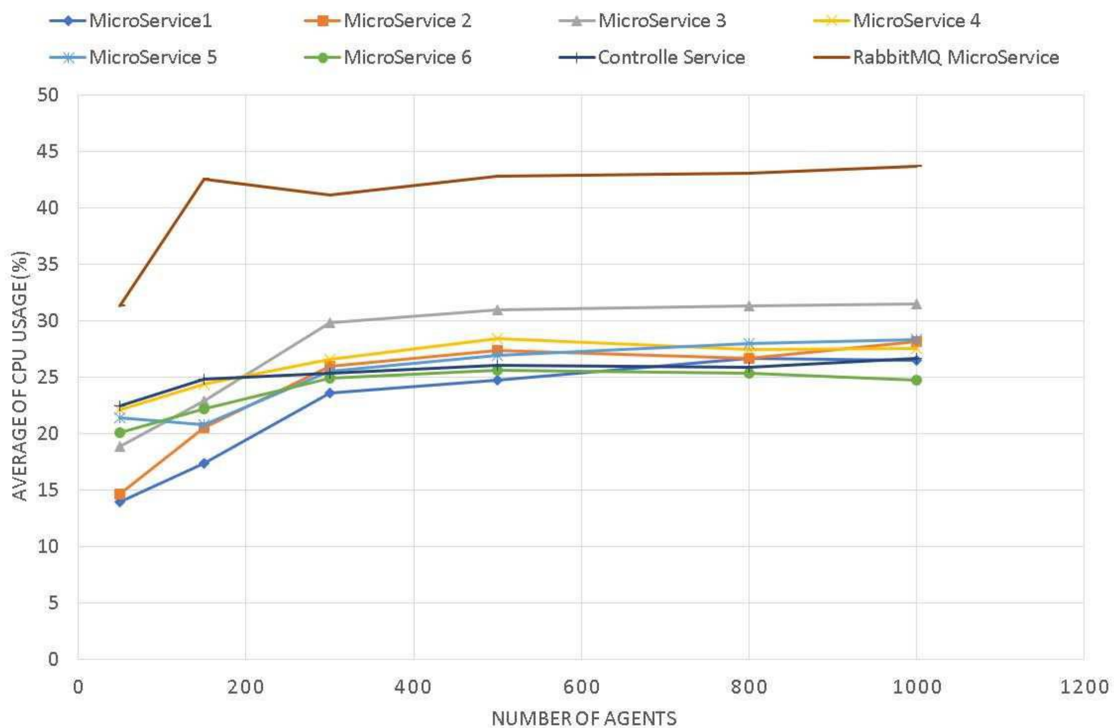


Figure 12. The average of CPU usage per number of agents in DiMAS

The Monty Hall model is simulated by NetLogo, GAMA, and Jade. We have repeated the simulation with a different number of agents including 50, 150, 300, 500, 800, and 1000 agents. Figure 11 shows the CPU usage of these frameworks during the simulation. NetLogo provides the best performance among these frameworks. The maximum CPU usage of NetLogo is 3.14%. GAMA has the lowest performance. The minimum CPU Usage of GAMA is 6.79% and the maximum is 51.11 %. Also, the Maximum CPU usage of Jade is 32.15%.

Figure 12 shows the results of the same experiments with DiMAS. CPU Usage of Agent's micro-service is almost fixed while increasing the number of agents. But, RabbitMQ micro-service has the lowest performance among these micro-services and it even gets lower when the number of agents increases. In the worst-case scenario, the average CPU usage of micro-services is 29.34%.

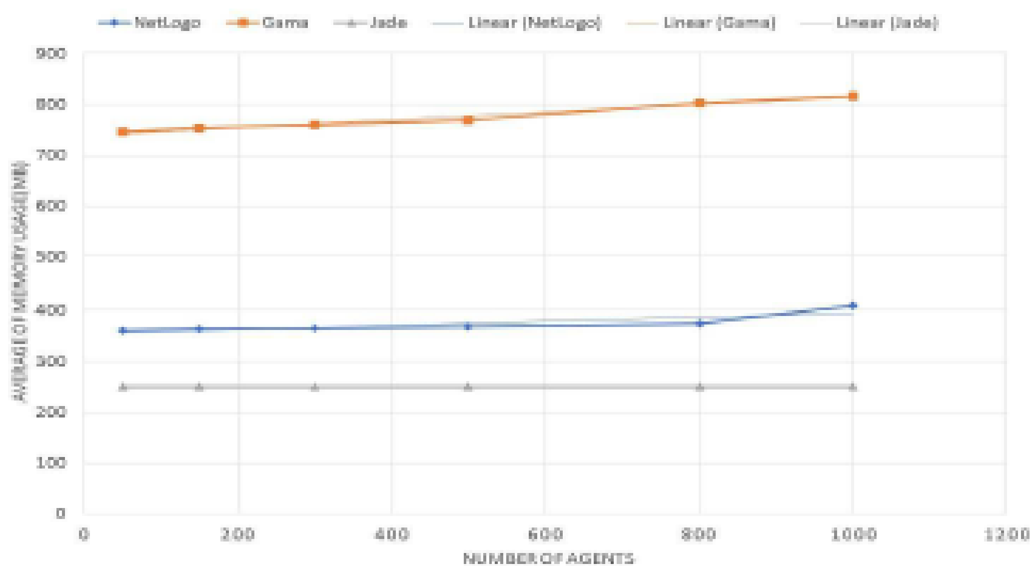


Figure 13. The average memory usage per number of agents for the compared frameworks

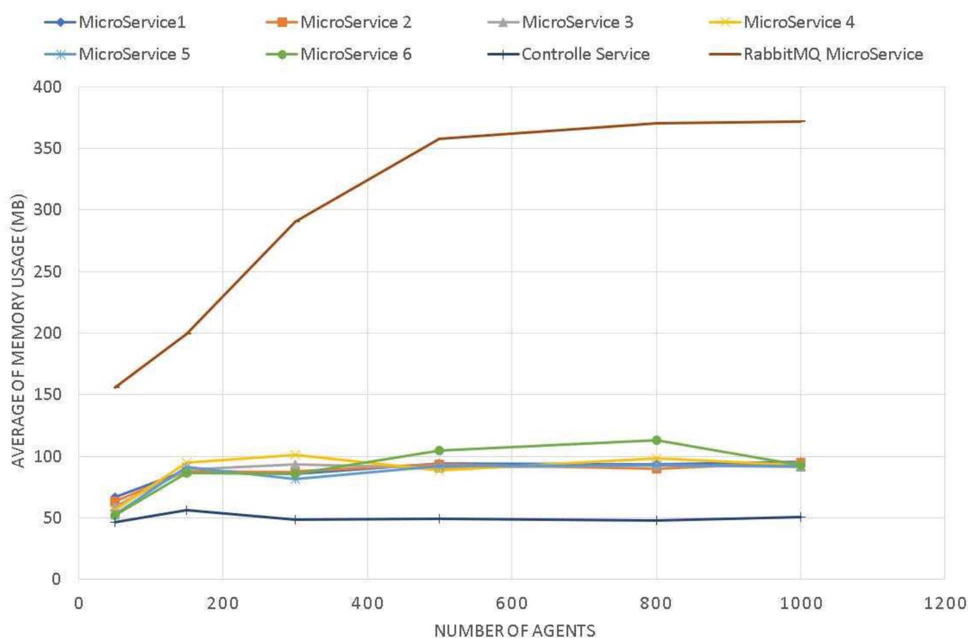


Figure 14. The average memory usage per number of agents for DiMAS

In the mentioned experiment, the memory usage of NetLogo, GAMA, and Jade was captured. Figure 13 shows these results. Jade's memory usage is fixed during all the simulations with 249.561 MB of usage. Maximum memory usage of GAMA and NetLogo is 815.5 MB and 409.21 MB respectively. Jade has the lowest memory usage among these frameworks. Figure 14 shows the result of the same experiments with DiMAS. Again, RabbitMQ micro-service has the most memory usage among micro-services and the micro-services containing agents use an almost fixed memory amount during the simulations. The average maximum memory usage of micro-services is 110.875 MB which is the lowest usage between these 4 frameworks.

5.2. Virus spreading simulation scenario

This model shows the spread of a virus on a graph with an average degree of 6. This model is simulated with 50, 150, 300, 500, 800, and 1000 agents by NetLogo, GAMA, Jade, and DiMAS. The results of these experiments are shown in Figures 15 and 16. This model is simulated by NetLogo, Jade, and GAMA. Figure 15 shows the results. NetLogo has the best performance with 4.1% of minimum usage and 16.71% of maximum usage. But, unlike the previous experiment, the usage of CPU increases. GAMA has the lowest performance with 16.79% of minimum usage and 53.11% of maximum usage.

Figure 16 shows the result of the same experiment with DiMAS. Micro-services containing agents have almost a fixed CPU usage amount. The CPU usage of Controller micro-service and RabbitMQ micro-service is higher than others with the maximum usage of 28.37% and 46.39%. The average for the maximum amount of CPU usage for all the micro-services is 21.387% which is better than GAMA and Jade. Figure 17 shows that the memory usage of the three frameworks is almost fixed. Jade with 251.102 MB memory usage has the best performance among the three. On the other hand, GAMA with 901.5 MB maximum memory usage has the lowest performance. Figure 18 shows the performance of DiMAS. Unlike other frameworks, the memory usage of micro-services increases by increasing the number of agents. The average memory usage for the micro-services is 304.52 MB which is lower than NetLogo and GAMA.

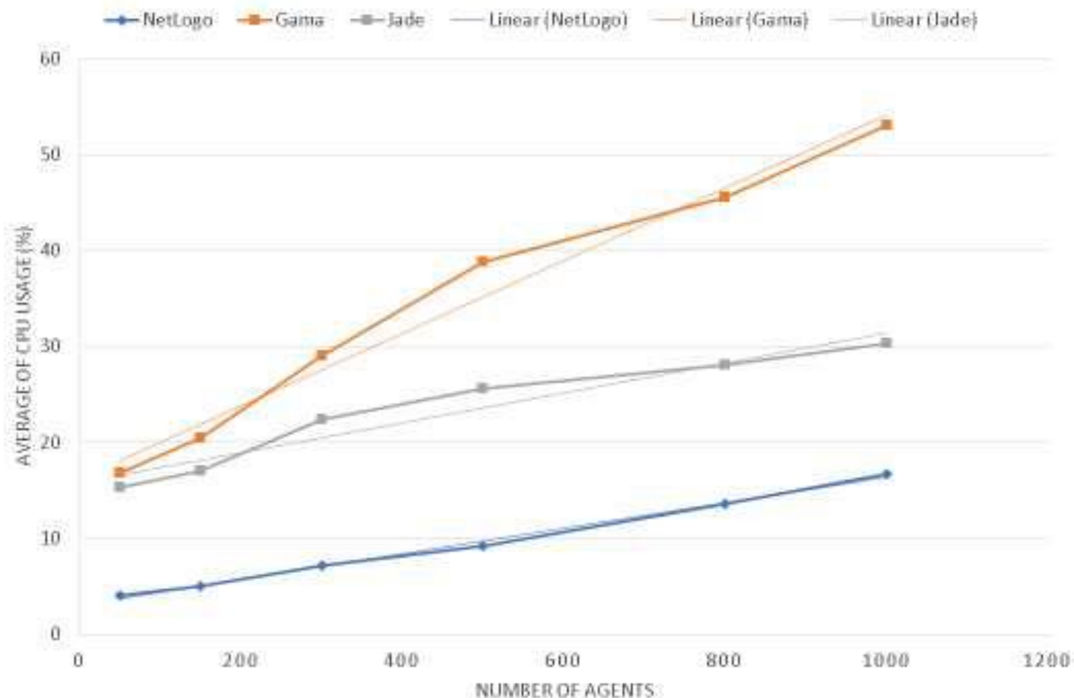


Figure 15. The average CPU usage of NetLogo, GAMA, and Jade

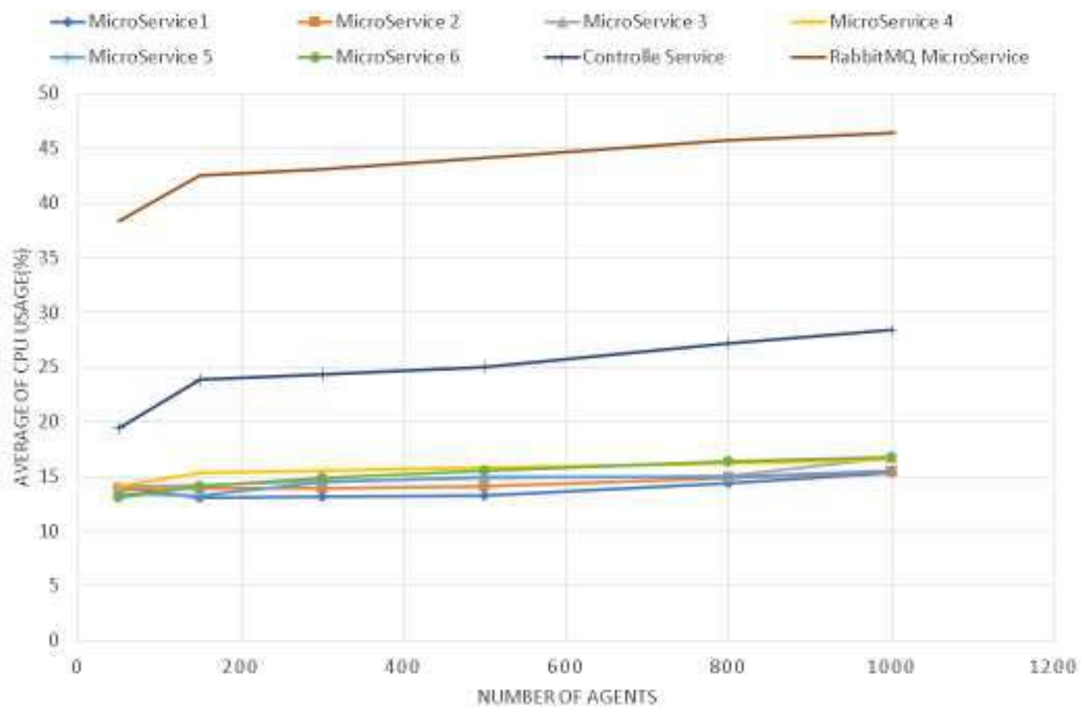


Figure 16. The average CPU usage of DiMAS

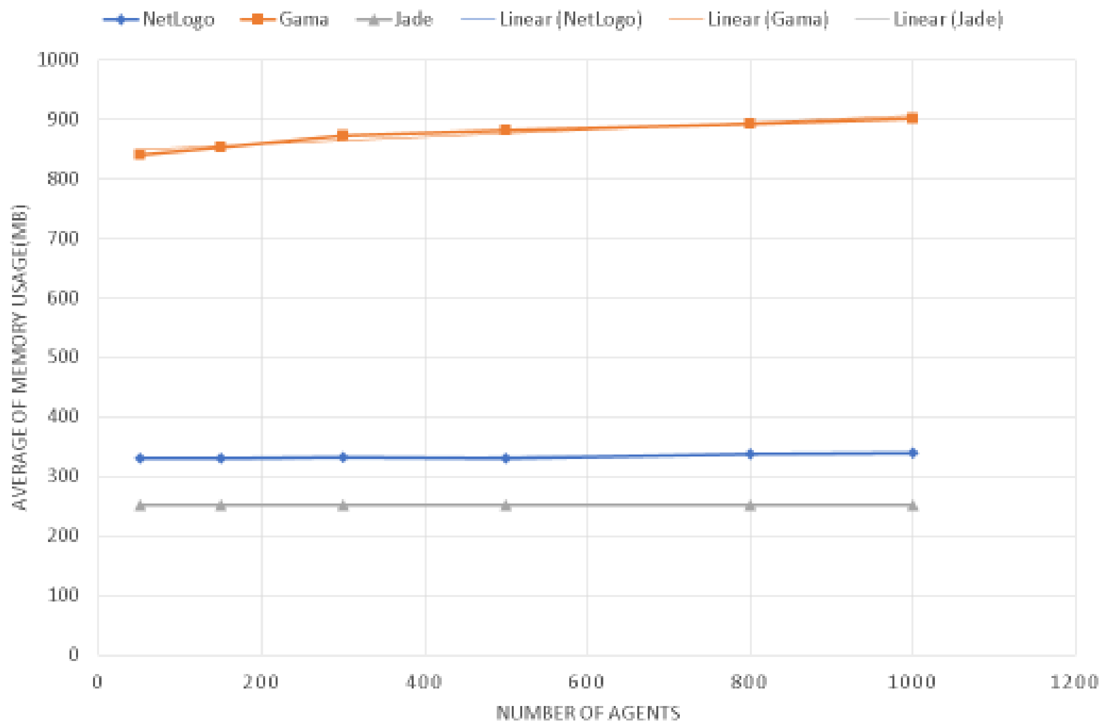


Figure 17. Average of memory usage of NetLogo, GAMA, and Jade

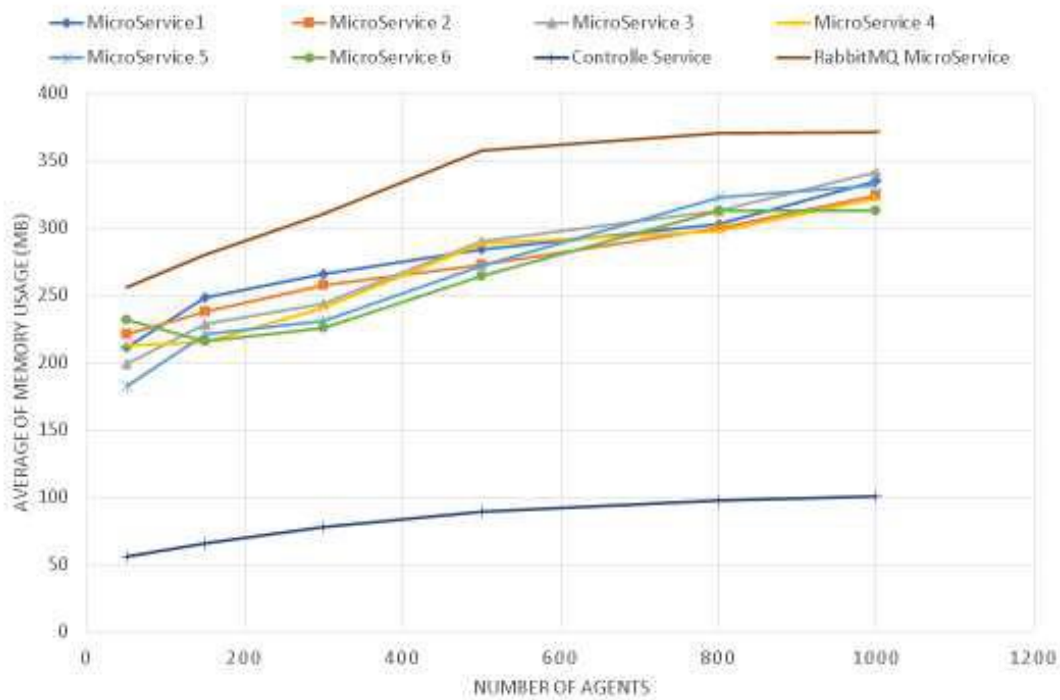


Figure 18. Average memory usage of DiMAS

6. Discussions

Centralized simulation is not enough for some of the large-scale models. Longer simulation times, lack of resources, incorrect results, and most importantly system failure are the inherent challenges of executing a simulation model in a centralized scheme. Hence, as the simulation models become increasingly more complex, we need to leverage the advantages of distributed execution. Implementing a distributed simulation framework is inherently more complex compared to the centralized versions. Synchronizing simulation clock, gathering results from nodes, handling messages, and scheduling actions are just a few of this essential complexity.

Distributed simulations paradigms decrease the simulation time, makes the system more extendable and flexible. Changing or failing one part of the system does not affect other parts of the system. Also, using micro-service architecture makes the system modular, and using containers help us to better maintain the system. Also, a container engine like Docker helps us to maintain and monitor containers. Therefore, developing a distributed simulation with the help of micro-service architecture will be effective in tackling some of the major challenges we have mentioned for the centralized simulation schemes. Currently, the transferring message queue is the main limitation. Each agent should send its state to the Controller micro-service for each timestamp. Also, each agent can send a message to another agent. The number of these messages can be variable based on the model's logical structure. So, for models with high transmission rate or models which have connected agents, using DiMAS does not produce any notable advantages even compared to the centralized simulation frameworks.

7. Conclusions

In this paper, we have presented a distributed agent-based simulation framework named DiMAS. A framework for large scale simulations that need to be distributed on multiple nodes. DiMAS is designed based on the micro-service architecture. Micro-services make the system more modular. Each module can be replaced easily and this makes the system more flexible and extendable. Using a container engine like Docker helps us to manage the load balancing of our nodes and maximize the performance of our system. In the current micro-service scheme, one or more agents are located in a micro-service and interact with one another using the RabbitMQ messaging system based on their APIs. Each agent has its simulation clock and synchronization is handled by deadlock avoidance using the null message algorithm.

DiMAS is divided into several components based on the main responsibility and primary scope of operation. The core is the main component and it must be used to define and implement agents and their behaviors. The messenger component handles the incoming messages and sends messages to another agent. Scheduler component manages and queues actions needed to be executed. TimeController component holds and increases the simulation clock and handles null messages. Users can leverage the interfaces and base classes for implementing their customized models. They must define agents and their behaviors by realizing these interfaces. They also must define micro-services as well as the corresponding configurations to distribute their simulation agents. The resulting simulation system is extendable and flexible and is capable of using the hardware resources efficiently.

As for the future works of this research, we intend to improve DiMAS's performance by decreasing the number of

exchanged messages. Optimizing or changing the time synchronization algorithm is a major part of this improvement. Null messages that are sent by agents are increased by increasing the number of neighbors. Optimizing the algorithm can have a significant impact on reducing the number of exchanged messages. Other solutions for decreasing the number of exchanged messages are eliminating messages that are added to the RabbitMQ queues for agents that are located in the same micro-service. Another solution for improving the performance is to distribute messages queues themselves. RabbitMQ micro-service is the system's primary point of failure at the moment. Every message is first added to the queue. So, if this micro-service fails, the whole system would fail. Distributing exchanges queues to multiple micro-services will decrease the length of queues, then messages are delivered quickly. Also, it reduces the failure potential of the whole system. Finally, we have in mind to design a graphical user interface for the framework and graphically show the results of the simulation scenarios. Creating a customized scripting language for defining the simulation model to make it more convenient for the users to create their simulation models is also a possible future work.

Compliance with Ethical Standards
This study has received no funding from any organization.
Conflict of Interest
All of the authors declare that they have no conflict of interest.

Compliance with Ethical Standards

This study has received no funding from any organization.

Conflict of Interest

All of the authors declare that they have no conflict of interest.

Ethical approval

This article does not contain any studies with human participants or animals performed by any of the authors.

References

[1] RabbitMQ. "RabbitMq tutorial and document." [Online]. Available: https://www.rabbitmq.com. [Accessed: Nov. 12, 2018].
[2] M. Wooldridge, "Agent-based Software Engineering," *J. Software Eng.*, vol. 144, no. 1, pp. 26–37, 1997.
[3] H. S. Nwana, "Software agents: an overview," *Knowl. Eng. Rev.*, vol. 11, no. 3, pp. 205–244, 1996.
[4] C. M. Macal and M. J. North, "Tutorial on agent-based modeling and simulation," *J. Simul.*, vol. 4, no. 3, pp. 151–162, 2010.
[5] S. Poslad and P. Charlton, "Standardizing agent interoperability: the FIPA approach," *Multi-agent Syst. Appl.*, vol. 2086, pp. 89–117, 2001.
[6] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. Wiley Interscience, Jan. 3, 2000.
[7] Y. M. Teo, Y. K. Ng, and B. S. S. Onggo, "Conservative simulation using distributed-shared memory," in *Proc. 6th Workshop on Parallel and Distributed Simulation*, May 2002, Washington, DC, USA, pp. 3–10.
[8] Microsoft. "Introduction to micro-service architecture." [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/micro-services/introduction. [Accessed: May 27, 2019].
[9] E. Wolff, *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, Nov. 11, 2015.
[10] "Micro-service definition." [Online]. Available: https://micro-services.io/. [Accessed: May 27, 2019].
[11] Docker. "Docker tutorial and document." [Online]. Available: https://www.docker.com. [Accessed: Nov. 12, 2018].
[12] NetLogo. "NetLogo tutorial and document." [Online]. Available: https://ccl.northwestern.edu/netlogo/. [Accessed: Nov. 12, 2018].
[13] Scala. "Scala programming language." [Online]. Available: https://www.scala-lang.org/. [Accessed: May 27, 2019].
[14] NetLogo. "NetLogo FAQ." [Online]. Available: http://ccl.northwestern.edu/netlogo/2.1/docs/faq.html. [Accessed: May 27, 2019].
[15] RePast. "RePast tutorial and document." [Online]. Available: https://repast.github.io/. [Accessed: Nov. 12, 2018].
[16] N. Collier and M. North, "Parallel agent-based simulation with Repast for high-performance computing," *J. Simul.*, vol. 89, no. 10, pp. 1215–1235, 2013.
[17] G. Chao and W. Xiong, "Parallel agent-based simulation of a complex system based on repast HPC," in *Proc. 2nd Int. Symp. Instrumentation and Measurement, Sensor Network and Automation (IMSNA)*, Feb. 2014, Toronto, Canada, pp. 808–812.
[18] RePast. "RepastHTC documentation PDF." [Online]. Available: https://repast.github.io/docs/repast_hpc.pdf. [Accessed: May 27, 2019].
[19] GAMA. "GAMA tutorial and document." [Online]. Available: https://GAMA-platform.github.io/. [Accessed: Nov. 12, 2018].
[20] GAMA. "GAMA Architecture." [Online]. Available: https://github.com/GAMA-platform/GAMA/wiki/GAMAArchitecture. [Accessed: May 27, 2019].
[21] N. H. S. Divine, T. Ndumu, L. C. Lee, and J. C. Collis, "ZEUS: A Toolkit and Approach for Building Distributed Multi-Agent Systems," in *Proc. 3rd Annu. Conf. Autonomous Agents*, Apr. 1999, Washington, USA, pp. 360–361.
[22] NVIDIA. "CUDA programming language." [Online]. Available: https://developer.nvidia.com/cuda-zone. [Accessed: May 27, 2019].
[23] NVIDIA. "Open-CL programming language." [Online]. Available: https://developer.nvidia.com/opencl. [Accessed: May 27, 2019].
[24] L. Guillaume *et al.*, "MCMAS: A Toolkit to Benefit from Many-Core Architecture in Agent-Based Simulation," in *Proc. Parallel Processing Workshops*, Aug. 2013, Aachen, Germany, pp. 544–554.
[25] C. Vecchiola *et al.*, "AgentService: a framework to develop distributed multi-agent systems," *Int. J. Agent-Oriented Softw. Eng. (IJAOSE)*, vol. 2, no. 3, pp. 290–323, 2008.
[26] W3C. "Web Service Architecture." [Online]. Available: https://www.w3.org/TR/ws-arch/. [Accessed: May 27, 2019].

[27] MASON. “Mason tutorial and document.” [Online]. Available: https://cs.gmu.edu/~eclab/projects/mason/. [Accessed: Nov. 12, 2018].

[28] D-Mason. “D-Mason tutorial and document.” [Online]. Available: https://sites.google.com/site/distributedmason/. [Accessed: Nov. 12, 2018].

[29] S. Luke *et al.*, “Mason: A multiagent simulation environment,” *J. Simul.*, vol. 8, no. 7, pp. 517–527, 2015.

[30] JADE. “Jade tutorial and document.” [Online]. Available: http://jade.tilab.com. [Accessed: Nov. 12, 2018].

[31] JADE. “Jade Architecture.” [Online]. Available: https://jade.tilab.com/documentation/tutorials-guides/jade-administrationtutorial/architecture-overview. [Accessed: May 5, 2019].

[32] M. Hybinette *et al.*, “SASSY: A Design for a Scalable Agent-Based Simulation System using a Distributed Discrete Event Infrastructure,” in *Proc. Winter Simulation Conf.*, Winter 2006, Monterey, CA, USA, pp. 926–933.

[33] J. S. Dahmann, “High-Level Architecture for simulation,” in *Proc. 1st Int. Workshop Distributed Interactive Simulation and Real-Time Applications*, Jan. 1997, Eilat, Israel, pp. 9–14.

[34] D. Chen *et al.*, “Large scale agent-based simulation on the grid,” *Future Gener. Comput. Syst.*, vol. 24, pp. 658–671, 2008.

[35] K. Popov *et al.*, “Parallel Agent-Based Simulation on a Cluster of Workstations,” *Parallel Process. Lett.*, vol. 13, no. 4, pp. 629–641, 2003.

[36] P. Van Roy, “Logic programming in Oz with Mozart,” in *Proc. 1999 Int. Conf. Logic Programming*, Nov. 1999, MIT, USA, pp. 38–51.

[37] P. T. Eugster *et al.*, “The many faces of publish/subscribe,” *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[38] Rebus. “Rebus Library.” [Online]. Available: https://rebus.fm/. [Accessed: Nov. 12, 2018].

[39] U. Wilensky, “NetLogo Three Doors model.” [Online]. Available: http://ccl.northwestern.edu/netlogo/models/ThreeDoors. [Accessed: Nov. 12, 2018].

[40] A. Grignard *et al.*, “GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation,” in *PRIMA 2013: Principles and Practice of Multi-Agent Systems*, Lecture Notes in Computer Science, vol. 8291, Springer, pp. 117–131.

[41] JADE. “Jade technical description.” [Online]. Available: http://jade.tilab.com/technical-description/. [Accessed: Feb. 14, 2019].