Scalable and Reactive Multi Micro-Agents System Middleware for Massively Distributed Systems

EZZRHARI Fatima Ezzahra, EL ABID AMRANI Noureddine, YOUSSFI Mohamed, BOUATTANE Omar
SSDIA Laboratory, ENSET, Hassan II University
Casablanca, Morocco

***Abstract*—IT transformation has revolutionized the business landscape and changed most of organizations business model into digital and innovation driven firms. To fully take advantage of this digitalization and the exponential growth of data, organizations need to rely on resilient, scalable, extremely connected, highly available & very performant systems. To meet this need, this paper presents a model of middleware for multi micro-agents system based on reactive programming and designed for massively distributed systems and High-Performance Computing, especially to face big data challenges. This middleware is based on multi-agents systems (MAS) which are known as a reliable solution for High Performance Computing. This proposal framework is built on abstraction and modularity principles through a multi-layered architecture. The design choices aim to ensure cooperation between heterogeneous distributed systems by decoupling the communication model and the cognitive pattern of micro agents. To ensure high scalability and to overcome networks latency, the proposal architecture uses distribution model of data & computing, that allows an adaptation of the grid size as needed. The resilience problem is addressed by adopting the same mechanism as Hazelcast middleware, thanks to his peer-to-peer architecture with no single point of failure.**

Keywords*—Massively distributed system; multi agent system (MAS); high performance computing; reactive programming; hazelcast

I. INTRODUCTION

Information technologies have faced breakthrough changes during the last decades: a huge acceleration of artificial intelligence, the invasion of cloud computing, an exponential growth of data with the appearance of 5G, the emergence of the big Data & IoT [1], and the birth of the blockchain.

This revolution is a real catalyst for the different fields and industries. It is the trend changing the future and requiring each organization to boost innovation, to ensure the performance and to improve the time to market so that it remains competitive and differentiated. So, companies need new information systems able to connect permanently with many objects, while executing treatments, analyzing huge quantities of data & making various decisions.

To meet these expectations, we need to establish new IT applications allowing data exploitation and enabling collective intelligence. The massive quantity of data received from all connected objects and social media need to be stored and analyzed differently with suitable strategies. Moreover, the systems need an acceleration of computing and are adopting more and more massively distributed machines such as GPU architecture (Graphic Processing Units) [2] to perform their treatments more efficiently. Even so, the use of massively distributed machine unitary is not enough efficient to process a very large amount of data and perform the needed processing quickly, so the use of massively distributed systems [3] has become very common, with the deployment of several heterogeneous systems to allow faster data processing and more efficient data storage and analysis, it is today the real solution for High Performance Computing [4].

This solution has been approved with the development of new middlewares offering the possibility of cooperating several heterogeneous hardware and software systems: mobile devices, servers, PCs, electronic cards, embedded systems, etc. However, challenges for this type of architecture remain relevant: limitation in terms of network latency, load balancing, scalability, maintenance & fault tolerance.

To design a such complex system, we must use a paradigm capable of integrating these different constraints and providing a complete solution, promoting cooperation, interaction & scalability. This is the case of Multi-Agent Systems [5] which have proven their usefulness for this type of high complexity problem.

This article proposes a new model of multi micro-agent middleware for massively distributed systems based on reactive programming and applied to big data applications. The proposal framework is built on several abstraction levels to ensure modularity, scalability, load balancing and fault tolerance. It allows to cooperate different micro-agents that can be deployed in heterogenous IT infrastructure with different communication channels and various learning models. This middleware offers several technological implementations & interfaces for each layer and it is also open to extension by new implementations. To ensure a good performance level and to deal with fault tolerance challenge, we chose to use the mechanism of Hazelcast in term of data and computing distribution. So, the present model ensures resilience by guaranteed replication, a peer-to-peer architecture for the distribution of processing operations, and fault tolerance with the absence of Single Point Of Failure (SPOF).

We have organized the rest of this paper into six sections. The following section II is a description of the overall middleware architecture. Section III details the micro agent structure and kinematics. In the fourth section, we carried a deep dive of the data distribution model. The fifth section describes the computing distribution model of the middleware.

Section VI present some performance measurement of the proposal framework. And last section concludes with highlighting advantages and improvement areas of the present work.

II. Global Architecture of the Proposed Middelware

We have designed this framework to ensure a high level of abstraction and modularity [6], it is composed of several abstraction layers that are 7 APIs:

- An agent API for easy creation and deployment of micro-agents allowing different implementations and using multiple programming languages. This API defines the lifecycle of a micro-agent such as instantiation, initialization, deployment, serialization, deserialization, and destruction;
- A Communication API, that allows clear and transparent communication between micro-agents by adopting semantic messages ACL compliant;
- A cognitive API to implement & assign learning models to micro-agents with both supervised models and/or reinforcement learning models;
- A data distribution API: allowing to the middleware a balanced and transparent distribution of massive data. It uses distributed collections to dispatch data across cluster's nodes of heterogeneous computers.
- A data computing that enables a transparent distributed computing among the cluster nodes.
- A monitoring API to scan the status of the MAS.
- An API to build the cluster by defining the infrastructure to use for the distributed system.

Figure 1 illustrates the architecture and the different layers of a multi micro-agent system built by three member nodes.

A. Cluster Builder API

To create a Multi micro-Agent System using this middleware, we need first to identify the soft & hard infrastructure by launching a cluster of nodes. These infrastructures enable the distribution of data & computing for massive data applications or for computationally intensive applications.

A cluster [7] is a network of machines where each machine executes a member Instance. Each member automatically joins the others to form the cluster in a decentralized model while still having instances fully connected to each other's. The cluster's instances represent the hard core of the infrastructure allowing the nodes of the cluster to accommodate the data and the distributed computing over the micro-agents of the application.

To ensure the junction between the members of the cluster, different discovery mechanisms can be used by members to find each other, namely:

- Multicast mode: This mode uses the multicast mechanism with UDP protocol. It is useful when the cluster instances belong to the same local network.
- TCP mode: This mode requires the specification of the IP address of one of the active nodes of the cluster when a new member joins the cluster.
- Cloud Discovery: The proposal framework allows the use of cloud discovery services such as: AWS Cloud Discovery, ZooJeeper, Apache jclouds, GCP Cloud Discovery.

After establishing the junction between the members of the cluster, any communication between these members is carried out exclusively by a TCP / IP mode.

B. Monitoring API

In order to monitor the state of the cluster, we suggest starting a special instance in the cluster. Once it joins the cluster, this instance receives real-time notifications from all instances in the cluster whenever the state of an instance changes. Therefore, this instance will allow real-time monitoring of the distribution of data and computing at the cluster level.

C. Data Distribution API

To allow data distribution, this layer provides the default interfaces and implementations to represent data in standard structures and collections such as List, Map, Queue, Set, etc.

D. Computing Distribution API

This layer allows to distribute the execution of massive tasks of an application among the nodes of the cluster. It provides various interfaces & implementations allowing to submit complex jobs for distributed execution.

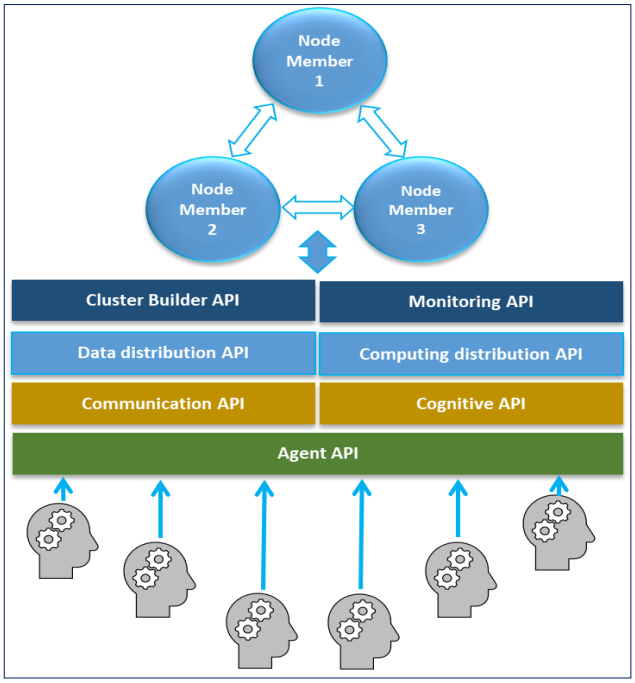

Fig. 1. Overall Architecture of the Proposal Middleware.

E. Communication API

We define at this layer the communication mechanisms between micro-agents. After each micro-agent deployment, the framework provides to this new agent a subscription to a Topic at the cluster level, then allows him to receive messages from other platform's micro-agents. This API also provides an implementation of Agent Communication Language (ACL) which allows agents to exchange semantic messages compliant to the FIPA [8] ACL standard and therefore ensuring interoperability with other MAS platforms.

F. Cognitive API

This API provides the interfaces and implementations for machine and deep learning models. It defines supervised, unsupervised and reinforcement learning models.

G. Agent API

This is the layer deploying interfaces & implementations to easily create micro agents by using and extending the functionalities offered by the other layers of the framework. This API also provides the mechanisms for managing the agent life cycle.

III. Micro Agent Structure and Kinematics

In this section, we will have a deep dive on the Agent layer by presenting its static structure, its ecosystem, and its interactions with the other layers. We will detail the life cycle of a micro-agent by its deployment and migration processes.

A. Agent API Description

To create a micro-agent, the developer has just to extend the abstract class "Agent" and to redefine the operators that composes the agent's life cycle at its container level.

The created micro-agent inherits all operators allowing:

- Creating and configuring ACL messages;
- Sending messages to a micro-agent or to a community of agents by choosing a communication strategy by the developer. In fact, the present framework is open to extension by using any communication mechanism as by external brokers such as KAFKA, RabbitMQ or ActiveMQ based on several messaging protocols as MQTT, AMQP or STOMP. If the developer does not have a preference, the system is based by default on an internal communication system as a broker directly using the messaging functionalities offered by the middleware cluster [9].

The figure 2 focuses on the principle of micro-agent's communication of the model.

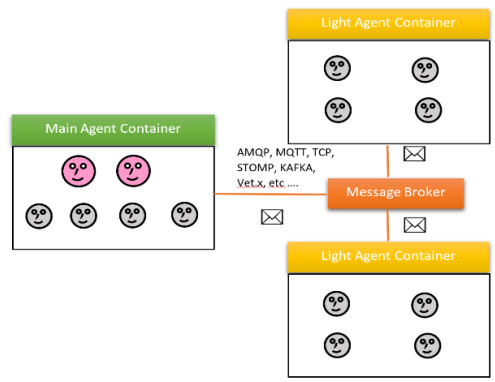

Fig. 2. Micro Agents Communication Model.

- Assign learning behavior to the micro-agent using one of the possible strategies. The framework implements 3 interfaces representing respectively:
  - supervised learning strategy with different implementations of machine and deep learning techniques based on neural networks.
  - unsupervised learning strategy with various implementations: the k-means clustering algorithm, fuzzy-cmeans, ...
  - reinforcement learning strategy with several possible implementations such as the Qlearning algorithm.

  The developer is free to choose the appropriate learning strategy among these three implementations, to assign to his micro-agent according to the context of his application.
- Create and access the data collections distributed over the nodes of the cluster. We have defined interfaces and implementations based on classic distributed structures like Queue, Map, Topic.
- Submit distributed tasks for cluster-level executions. To create a distributed task, the developer must create a class that inherits from the abstract DistributedCallableTask class and then redefine the call method by implementing the code of the task to be distributed. The micro-agent can submit this distributed task to a cluster node transparently for remote execution returning the result asynchronously. Once deployed in a node, the task becomes bound to the instance, allowing it to transparently access the functionality and data distributed in the cluster.

Figure 3 illustrates the core class diagram of this API.

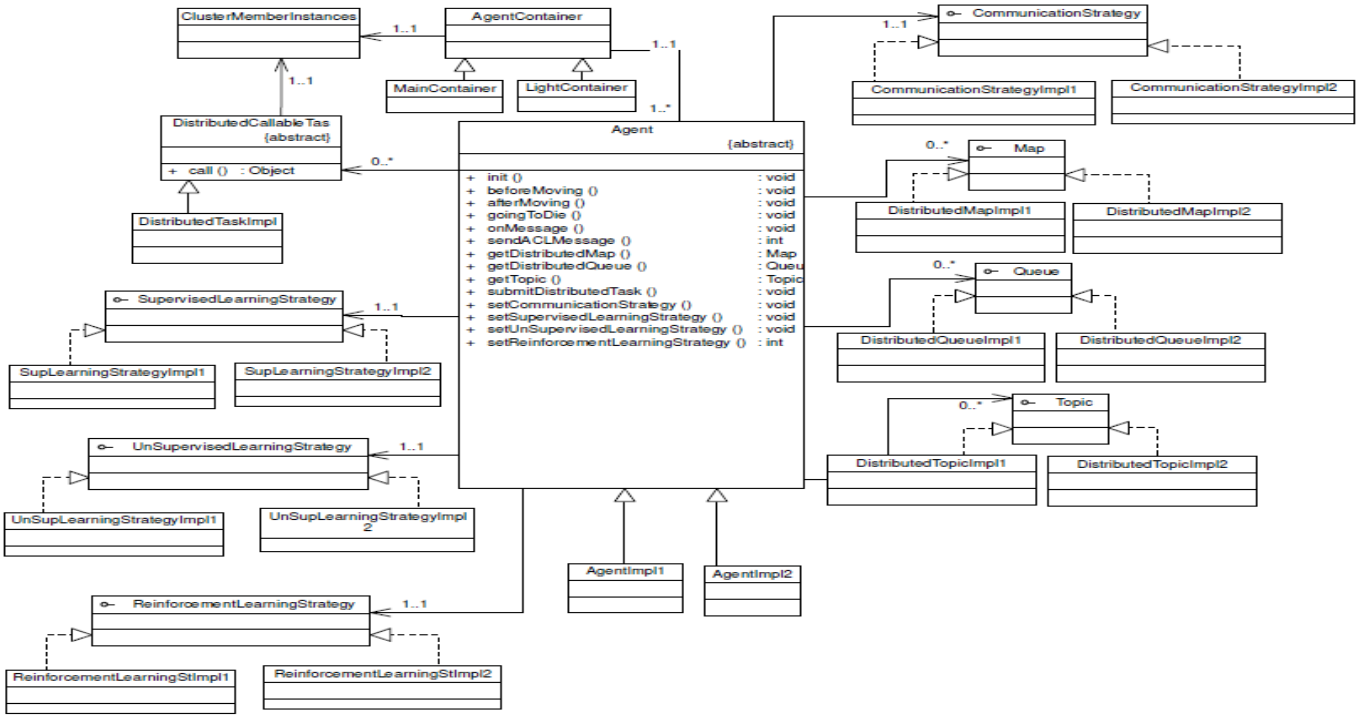

Fig. 3. Summary Class Diagram of the Agent API.

B. *Agent Container*

A micro-agent is systematically deployed in a container where it lives and finds the various techniques for managing its life cycle. An agent is systematically deployed in a container where it lives and finds the various techniques for managing its life cycle.

The present model is FIPA Compliant, it deploys two types of container: MainContainer which is deployed in a single instance of the MAS platform, and several LightContainers which allow the deployment of the agents of the developed MAS platform.

The MainContainer essentially deploys technical agents in accordance with the specifications of the FIPA:

- Agent Management System (AMS): used to manage the identity of agents and the communication system between agents.
- Directory Facilitator (DF) Agent: which defines the directory of yellow pages allowing agents to publish their services and discover the services offered by other agents of the MAS platform.

Once the MainContainer instance is started, the following operations are automatically performed:

- the launch of the first instance of the cluster for distributed computing.
- the deployment of AMS and DF agents that each subscribes their own mailbox as a Topic, at the level of the messaging service provided by the cluster.
- The subscription to a topic specific to the MainContainer.

  Indeed, each time a container is created, the system must create a specific mailbox for this container, which is used in different agent operations, notably when a migration of an agent is requested to this container, by retrieving the code of the migrant agent in the mailbox.
- The start of the MainContainer graphical interface. This interface has a graphical component representing each agent deployed in the container to easily and visually identify the agents deployed and their location/status.

Figure 4 shows the sequence diagram illustrating the deployment of the MainContainer.

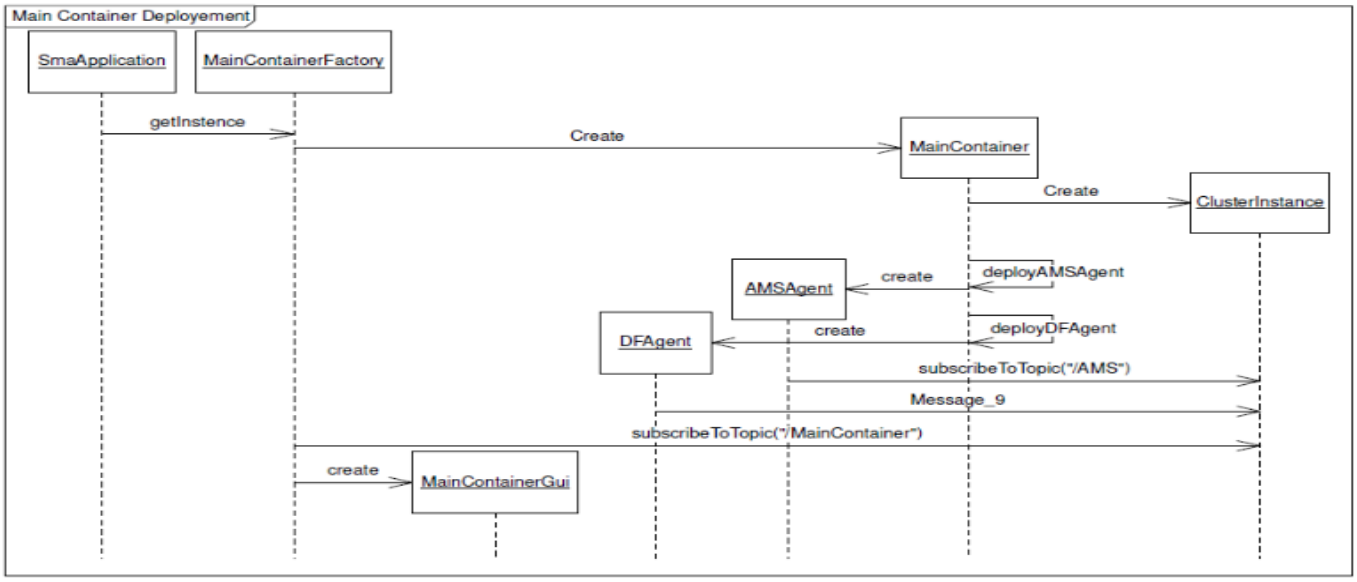

Fig. 4. Sequence Diagram for MainContainer Deployment Process.

To deploy a MainContainer with its Graphical interface, the developer must use just one code line:

```
MainContainer mainContainer=MainContainer.getInstance(true);
```

Figure 5 is a screenshot of the MainContainer graphical interface.

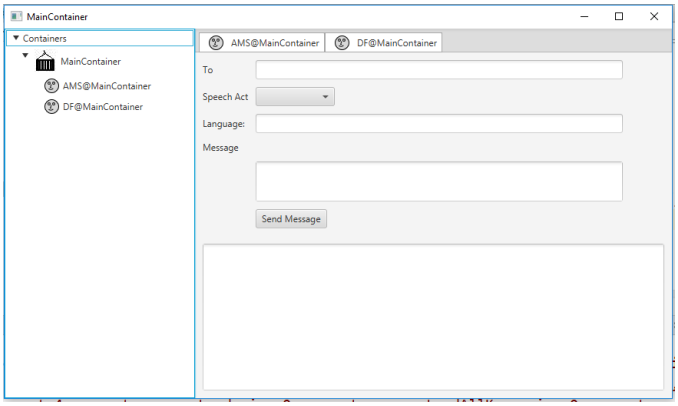

Fig. 5. Default Graphical Interface for the MainContainer.

C. Agent Deployement

The creation of an agent goes through an extension of the abstract class "Agent", then a redefinition of the various methods of the agent lifecycle management, in particular:

- init() method: is called by the container while its deploying just after instantiation. This method allows to the developer to initialize the agent and assign it its behaviors.
- onMessage() method: is invoked by the container every time a message is received by the agent topic.
- beforeMoving(): is called just before activating the agent migration process to another container.
- afterMoving() method: is invoked after the agent migration process.
- goingToDie() method: is performed just before the agent destruction.

The listing 1 represents an example code for a java implementation of an agent. It shows the main methods of the agent lifecycle.

```
public class SampleAgent extends Agent {
 @Override
 public void init() {
 }
 @Override
 public void onMessage(ACLMessage aclMessage) {
 }
 @Override
 public void beforeMoving(String from, String to) {
 }
 @Override
 public void afterMoving(String from, String to) {
 }
 @Override
 public void goingToDie() {

 }
}
```

To deploy an agent, we have first to create a LightContainer where the agent will live, then deploy the agent using the deployAgent () method. Figure 6 illustrates the deployment process of an agent.

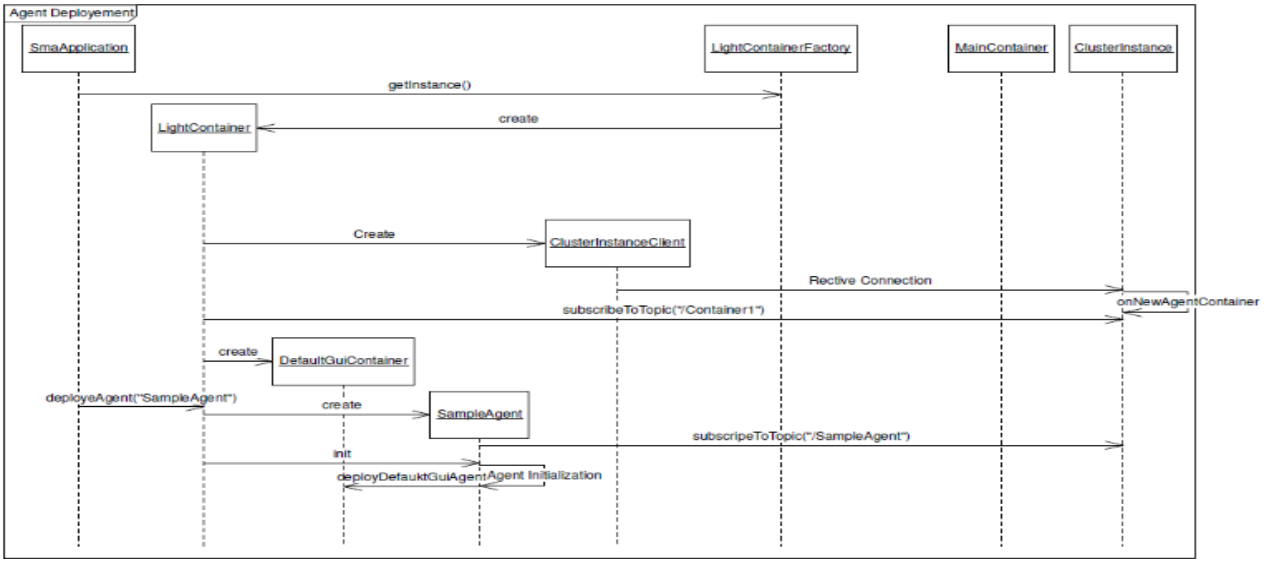

Fig. 6. Sequence Diagram for the Agent Deployment.

The deployment process begins by calling the container factory "LightContainerFactory" that creates an instance of LightContainer. This container will connect to the distributed computing cluster by Launching an instance of "ClusterInstanceClient". This instance establishes a permanent and transparent connection to the container through its instance linked to the MainContainer. The created lightContainer subscribes to his own topic at cluster level, which constitutes a reception box for agents requesting to migrate to this container. Subsequently, if the developer wishes, a default graphical interface for this container is displayed.

Once the container is ready, the agent is deployed using the deployAgent () method - which is an instantiation of the agent class -, then the agent asks the cluster to create its own mailbox by creating its own topic. After this initialization, the agent deploys its default graphical interface inside the graphical interface of its container. these graphical interfaces are very useful to allow the developer to graphically visualize the different agents of the platform without having to develop code for this purpose; it can send messages to agents, activate the migration of an agent to another container or even display the messages received by the various agents.

These two code lines below represent the creation of a LightContainter and the deployment of an agent with default graphical interface.

```
LightContainer
lightContainer=LightContainer.getInstance("Container1",true);
lightContainer.deployNewAgent("SampleAgent",
SampleAgent.class,true);
```

The following screenshots show the graphical interfaces of two containers MainContainer and LightContainer (Figure 7 & figure 8).

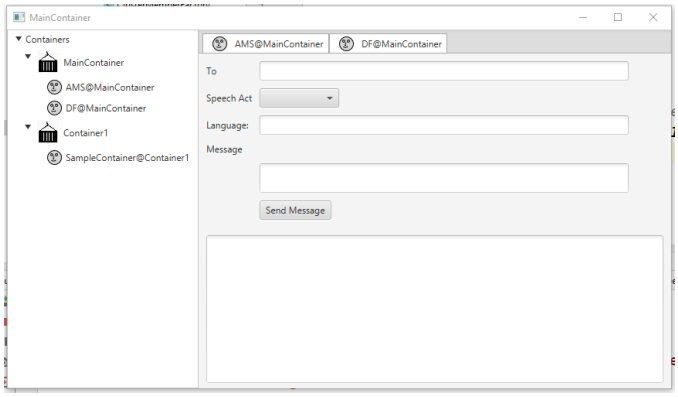

Fig. 7. MainContainer Graphical Interface.

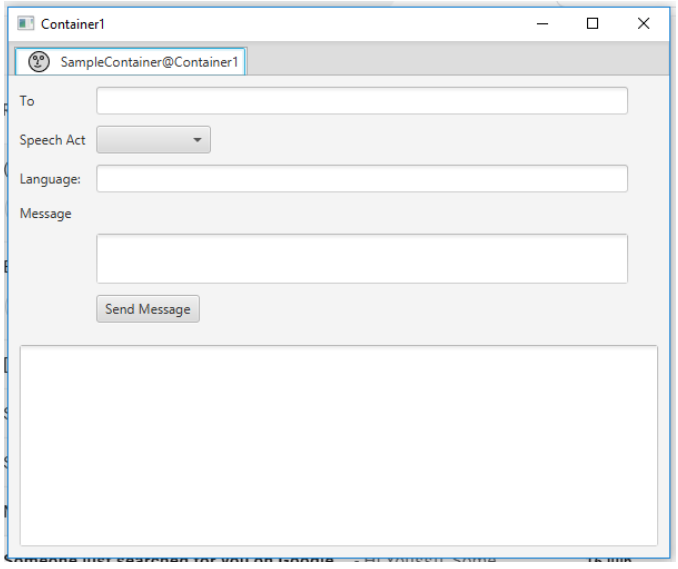

Fig. 8. LightContainer Graphical Interface.

D. Agent Migration

The agent mobility or migration is an essential asset of Multi Agents Systems, it provides the ability to agents to migrate from their initial container to other containers for many reasons: load balancing to overcome problems of overloading resources, requirements or constraints of applications requesting agent relocation.

To migrate an agent, we must send to the agent an ACL message with the communication act is "MIGRATE" and the content is the address of the destination container. Figure 9 shows the sequence diagram of an agent migration process.

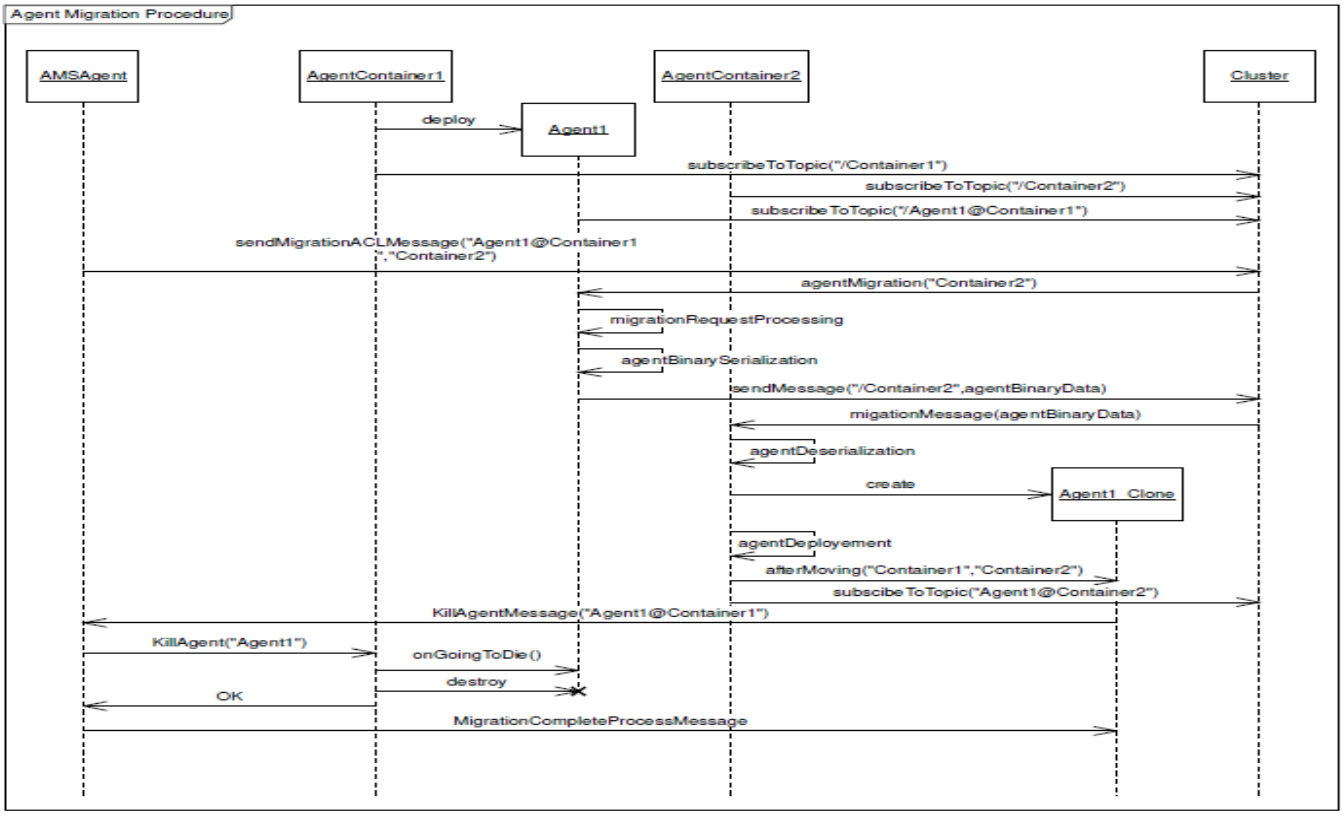

Fig. 9. Sequence Diagram of an Agent Migrating between Two Containers.

As explained previously, each container subscribes to a reception topic for migrant agents.

The sequence diagram above illustrates an agent "Agent1" initially deployed in "AgentContainer1". This agent had its own topic "Agent1@Container1". Once "Agent1" received from AMS agent an ACL message requesting him to migrate to "Container2", he studies the possibility of this migration according to his state, then, if the migration is possible, he auto-serializes into a byte array. Afterwards, the agent sends his clone by message to the Container2 topic. This latter deserializes the Agent1 clone and deploy it. The agent method "afterMigration" is invoked by Container2 and a notification is sent to the AMS agent requesting him to kill the original agent. The AMS agent sends to the Container1 an ACL message with the act of KILL and the content is the Agent1 address. Container1 runs the onGoingToDie() method, kill the original agent and sends a notification to AMS agent. AMS agent updates his context and the graphical interface of the MainContainer with current localization of agents, then sends to Agent1 his new localization using the afterMoving() method.

The following figures show some screenshot of containers graphical interface before and after migration. We can see the change of location of the agent "SampleContainer" that moved from Container1 to Container2. All graphical interfaces of MainContainer, Container1 & Container2 tracked this migration (Figure 10 to figure 15).

1) Before migration:

a) MainContainer

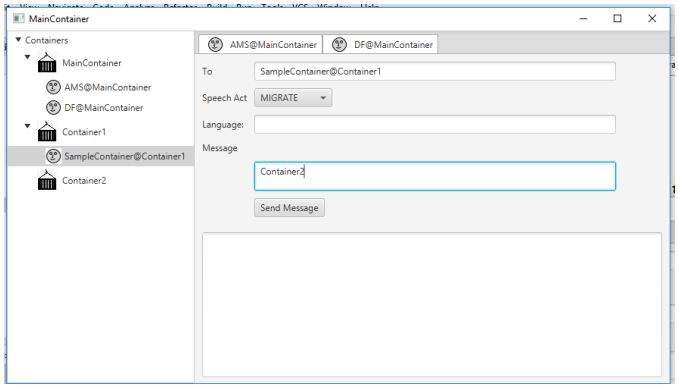

Fig. 10. MainContainer before Migration.

b) Container1

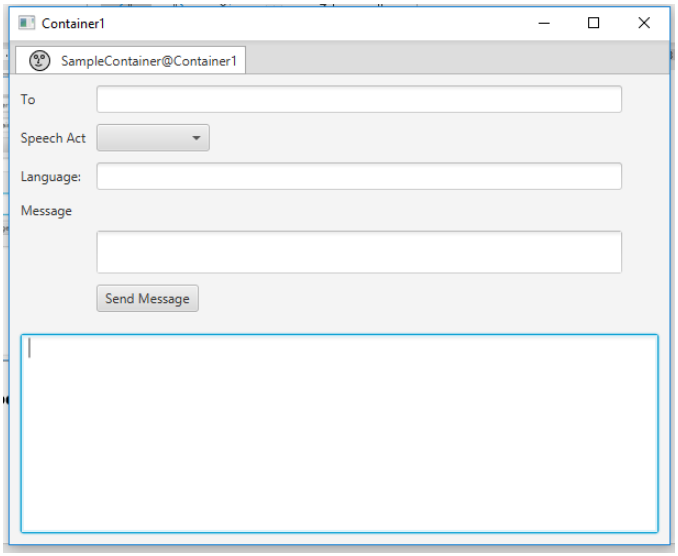

Fig. 11. Container1 before Migration.

c) Container2

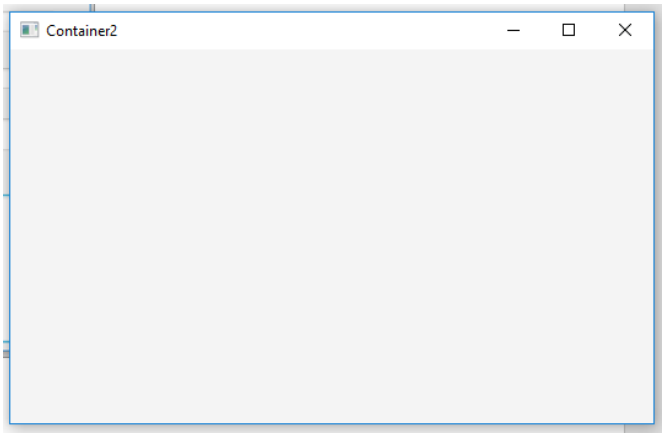

Fig. 12. Container2 before Migration.

2) After migration:

a) MainContainer

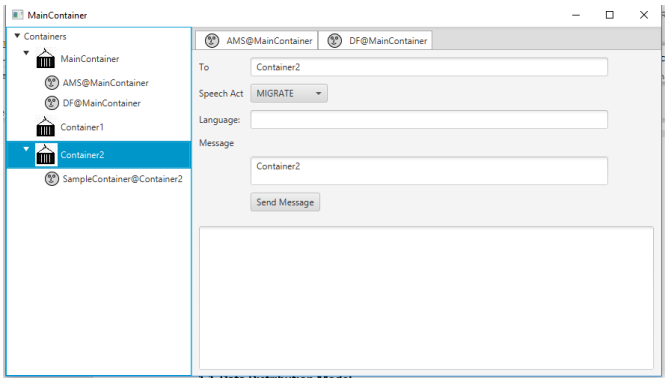

Fig. 13. MainContainer after Migration.

b) Container1

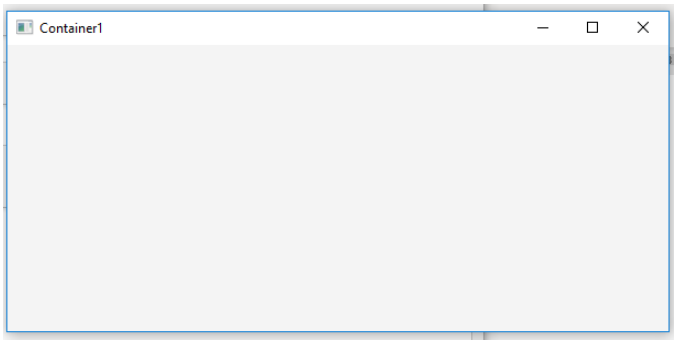

Fig. 14. Container1 after Migration.

c) Container2

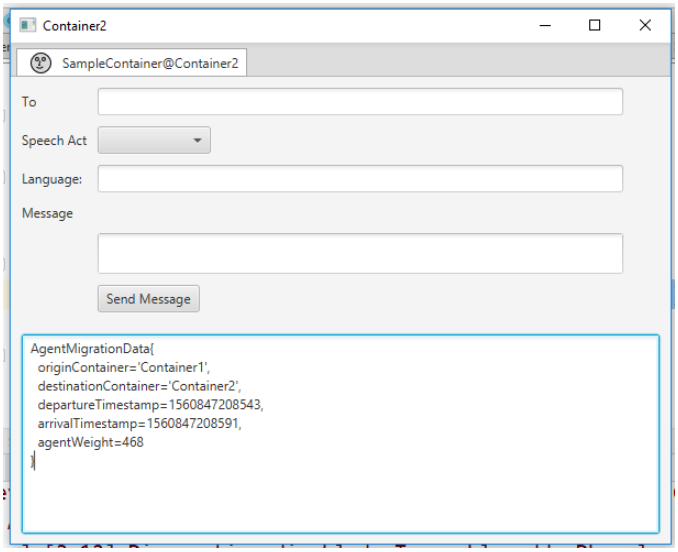

Fig. 15. Container2 after Migration.

It is important to retrieve the migration information, including the duration of the migration and the size in bytes of the agent. to take advantage of this logging functionality, just run the getLastMigration () method of the agent class. It allows us to log in:

- The original container;
- the destination container;
- the start time of migration;
- the end time of migration;
- the size of the agent.

IV. Data Distribution Model

Data distribution requires the definition of data structures as collections. For the management of these distributed collections, we chose the same mechanism used by the Hazelcast middleware [10].

A. Hazelcast

Hazelcast is an In Memory Distributed Grid (IMDG), it is a java open-source middleware allowing to create distributed memory cache.

In a Hazelcast grid, data are distributed evenly among the nodes of a group of computers to ensure:

- Scalable distributed storage (distributed memory cache).
- Scalable distributed computing.
- Replication of data on several nodes for fault tolerance.

These three Hazelcast principles reduce the load of database queries and improve the performance of distributed systems.

The following figure 16 shows the Hazelcast architecture. It consists of a cluster of nodes which host the distributed data (3 nodes in the example diagram below), a memory cache distribution layer which performs dispatching and ensures compliant addressing of the data, and various client APIs of various and varied programming languages to allow communication with other components of the system which could be heterogeneous.

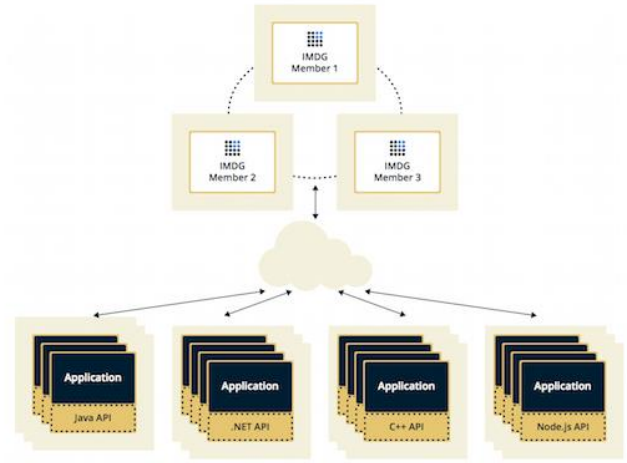

Fig. 16. Hazelcast Architecture.

B. The Distribution Model Description

The creation of a distributed collection can simply be done using one of these methods: getMap(), gestQueue(), getTopic() of a cluster's instance.

For example:

```
Map<String, String> data=memberInstance.getMap("myData");
```

The expression above allows to implicitly return an instance of DistributedMapImpl. Then, we can easily manipulate the inputs of this collection using methods "put" & "get" as follows:

```
data.put("key1","Item 1") ;
data.put("key2","Item 2") ;
data.put("key3","Item 3") ;
String value=data.get("key1");
```

A distributed collection is splited into several partitions. A partition is a memory segment able to contain hundreds or even thousands of data inputs depending to the capacity of the system memory.

Each partition can have several backups that are distributed over the cluster nodes. One of these partitions becomes the main replica and others are secondary. The cluster member that has the main replica becomes the owner. When we need to read or write a specific data entry, we address transparently the owner of the partition containing that entry.

By default, the system proposes a number n of partitions to create. When we start the cluster with one member, it owns all n partitions. For example, if n=100, we will have:

P 1
P 2
P 3
…
P 98
P 99
P 100

Node 1

Once we launch a second member of the cluster, the partitions are distributed over the two nodes as follows:

P 1
P 2
…
P 50
P 51
P 52
…
P 100

Node 1

P 51
P 52
…
P 100
P 1
P 2
…
P 50

Node 2

The 50 first partitions remain at node 1 that is the owner, while partitions 51 to 100 are sent implicitly to node 2 which will be their owner. A backup (in red) of the 50 partitions of node 1 is created in node 2, and vice-versa.

If we start or stop other cluster members, the same distribution mechanism happens again. For example, if the cluster has 4 nodes, we will have:

P 1
P 2
…
P 25
P 76
P 77
…
P 100

Node 1

P 26
P 27
…
P 50
P 1
P 2
…
P 25

Node 2

P 51
P 52
…
P 75
P 26
P 27
…
P 50

Node 3

P 76
P 77
…
P 100
P 51
P 52
…
P 75

Node 4

Thereby, we distribute main and secondary partitions equally among cluster members. backup replicas of partitions are kept for redundancy.

For data partitioning, we use the following algorithm:

- When a cluster member starts up, a partition table is created in that member.
- This partition table records the IDs of the partitions and the cluster members to which these partitions belong. This allows each member to know where the data is.
- The oldest member of the cluster (the one that started first) periodically sends the partition table to all members. This way, every member of the cluster is informed of any change in partition ownership.
- Repartitioning is carried out each time a new member joins or leaves the cluster.

C. Advantages of this Distribution Model

This mechanism offers to us solutions to 3 main problems:

- First, spread the data over several nodes of the cluster, which overcomes the problem of storage limit of the memories of the physical units representing the nodes (scalability).
- Second, face the challenge of fault tolerance, because if a node goes down, the data is not lost (replication).
- Third, in the case of distributed computing, the execution of each node can perform the elementary task using the part of the elementary data fragments of the local node (performance).

V. Distributed Computing Model

A. Static Model

To create a distributed task, you would have to create a class that extends the abstract generic DistributedCallableTask <T> class, then implement the code to be executed in the generic public T call () method.

This class implementing the two interfaces Callable <T> and Serializable is linked to the instance of the cluster that hosts this task. This allows access to data distributed in the cluster grid.

To deploy a distributed task, we define in this layer an ExecutorService with several implementations allowing this task to be submitted to an instance, a group of instances or to all instances of the cluster.

Figure 17 shows the main part of the class diagram of this API.

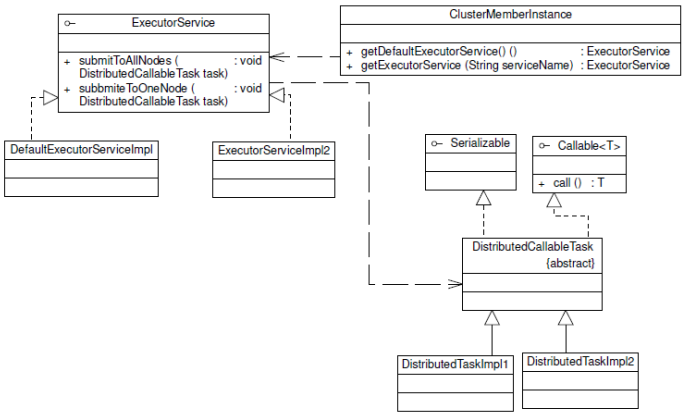

Fig. 17. Class Diagram of the Task Distribution Layer.

B. Sequence Model

Figure 18 illustrates the sequence diagram which describes an example of deployment of a distributed task in the cluster.

The agent begins by soliciting the local cluster client instance to retrieve the Distributed Task Execution service. This operation returns a DefaultExecutorService object configured and linked to the local cluster instance.

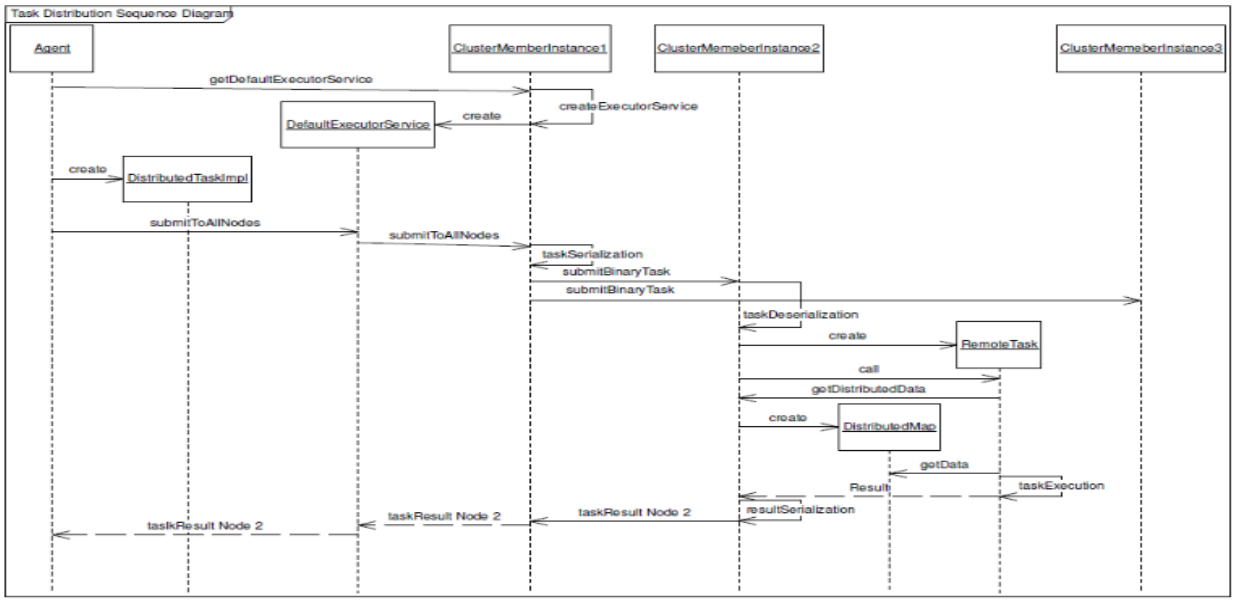

Fig. 18. Sequence Diagram of a Distribution Task Example.

After instantiating the implementation of the distributed task, the agent calls the ExecutorService object to submit the code for that task to all nodes in the cluster. The latter relies on the local cluster instance to do this. The DistributedTaskImpl object is first serialized in binary format before submitting this binary code to all instances in the cluster.

Each remote cluster instance that receives this code deserializes the object implementing the task code and then configures it by binding it to the local cluster instance. Then the remote cluster instance executes the call method of the distributed task.

Often in the distributed task code, we need to retrieve the data to be processed that is distributed in the grid as shared memory by calling the getDistributedData () operations of the local cluster instance. Then the result of the execution is returned to the cluster instance that submits this task. This result is returned to the agent who created the task.

VI. Highlight on the Middelware Performance

The present middleware aims to optimize usage performance to verify this objective, we have monitored and carried out several measurements during the execution of the multi-agent system.

A. System Status after Maincontainer Launch

Figure 19 indicates an optimization of threads number just after the launch of the platform with a reduction of CPU use.

B. System Status after Container1 Creation

The following figure 20 shows an increase of the threats number after the deployment of a lightContainer “Container1” with its graphical interface. All usage of the system increase: CPU and memory occupation.

C. System Status after Container2 Deployment

Figure 21 present an evolution of the performance status after the creation of a second LightContainer “Container2”. At the creation moment, there is a pic of threats that is optimized just after the launch by eliminating all inactive threats.

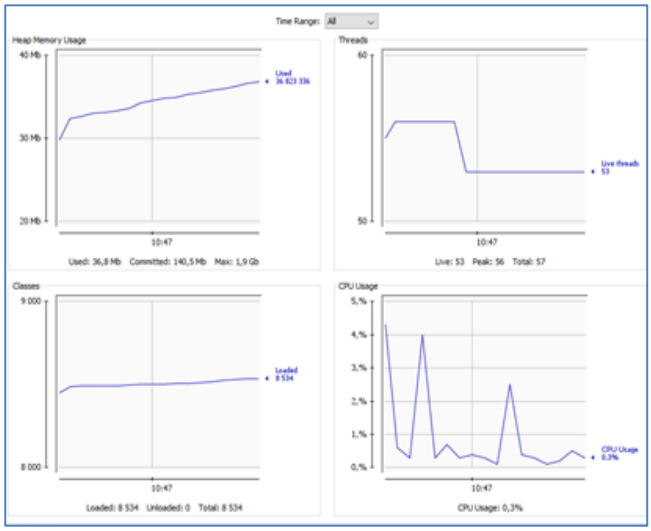

Fig. 19. Overview Performance Measurement – After MainContainer Launch.

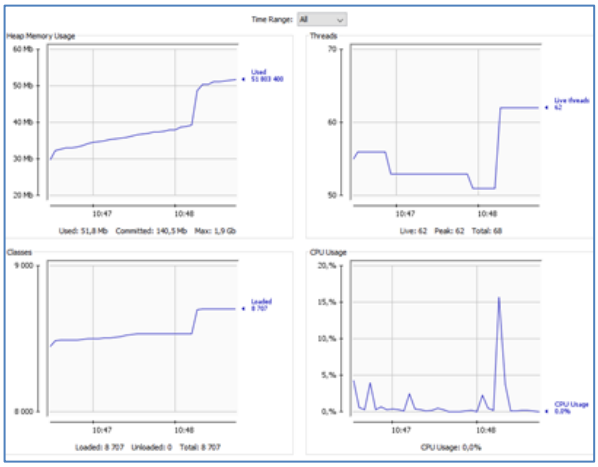

Fig. 20. Performance Measurement – After Container1 Deployment.

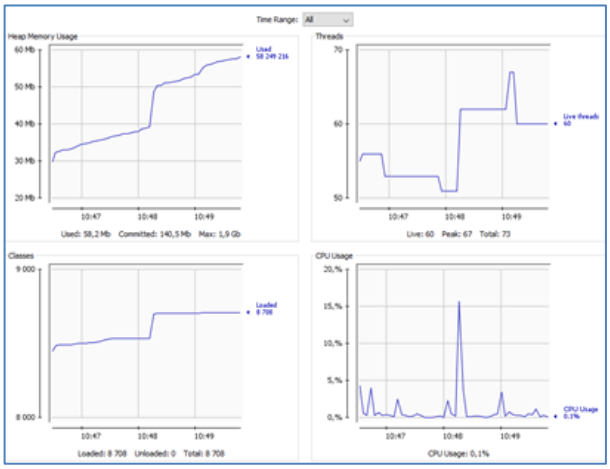

Fig. 21. Performance Measurement – After Container2 Deployment.

D. System Status after Migration Precess from Container1 to Container2

The figure 22 illustrates a liberation of many threats and memory after the agent migrates.

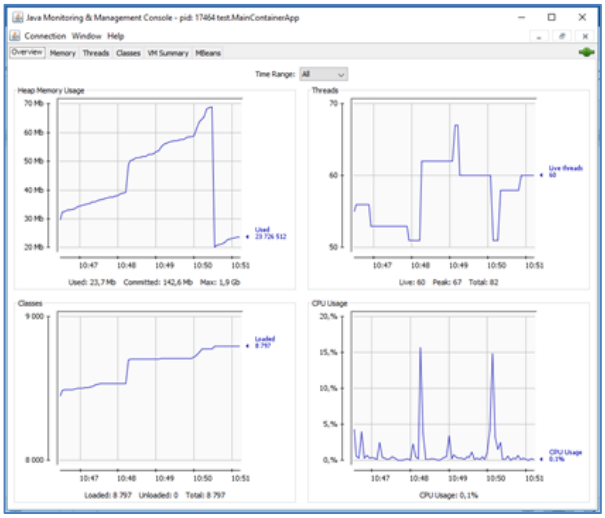

Fig. 22. Performance Measurement – After Agent Migration.

E. System Status after an Agent Communication

Figure 23 visualizes that in case of an agent communication act, some threats are used to send the message and are destructed just after optimizing also the CPU and memory usage.

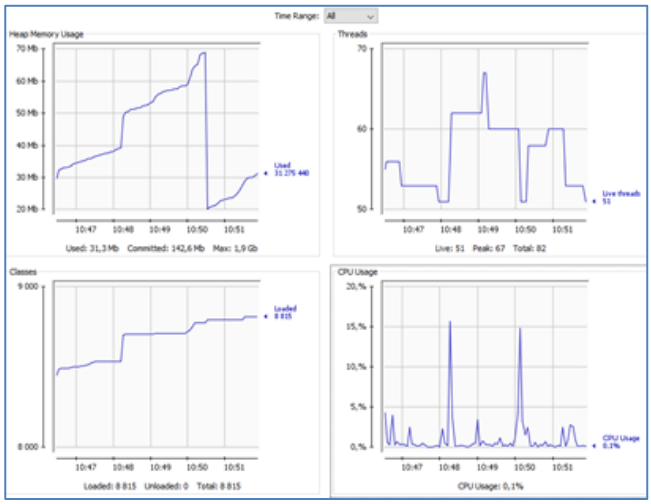

Fig. 23. Performance Measurement – After Agent Communication.

All these use cases show that the present middleware mobilize threats, CPU and memory just as needed, and optimizes these performance metrics dynamically.

VII. Conclusion

We presented in this article a scalable multi micro agent middleware based on reactive programming and designed for massively distributed systems and High-Performance Computing. This middleware is modular, based on seven APIs separating the creation/management of micro-agent, the communication pattern, the learning pattern, the data & distribution models, and the creation of the cluster and its monitoring.

The main objective is to find a reliable solution to design and build applications of massively distributed systems enabling cooperation, scalability, communication efficiency, resilience, and fault tolerance. We based the data & computing distribution approach on Hazelcast mechanism, which is efficient in term of data storage, cache computing structure & tasks distribution. Several performance metrics were explored after implementing the proposal middleware, that show optimization of CPU usage, memory allocation and threads mobilization.

To confirm the performance of this solution, we are going to implement it in a big data context with a massively distributed architecture. The results of this implementation and the performance measurements will be published in a future article.

Acknowledgment

This project has received funding from the European nion's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No 777720.

References

[1] E. Ahmed, I. Yaqoob, I. A. Targio Hashem & al. :"The role of big data analytics in Internet of Things". In: Computer Networks, vol. 129, pp 459-471, December 2017.

[2] R. Todd Evans, M. Cawood, S. Lien Harrell & al. : "Optimizing GPU-enhanced HPC system and cloud procurements for scientific workloads". In: High Performance Computing, pp. 313-331, June 2021.

[3] M. Youssfi, O. Bouattane, J. Bakkoury & M. O. Bensalah "A new massively parallel and distributed virtual machine model using mobile agents". In: 2014 International Conference on Multimedia Computing and Systems (ICMCS), pp. 407-414, April 2014.

[4] Youssfi M., Bouattane O., Bensalah M. : "A parallel computational model based on mobile agents for high performance computing". In: Contemporary Engineering Sciences, vol. 8, no. 15, pp. 677- 698, 2015.

[5] J. M. Alberola, J. M. Such, V. Botti, A. Espinosa, A. Garcia-Fornes : "A scalable multiagent platform for large systems". In Computer Science and Information Systems Journal, vol. 10, N. 1, 2013.

[6] F. Ezzrhari, M. Youssfi, O. Bouattane, V. Kaburlasos : "Scalable multi agent system middleware for HPC of big data applications". In : 2020 International Conference on Intelligent Systems and Computer Vision (ISCV), June 2020.

[7] P. Rathore, D. Kumar, J. C. Bezdek, S. Rajasegarar, M. Palaniswami "A rapid hybrid clustering algorithm for large volumes of high-dimensional data". In: IEEE transactions on knowledge and data engineering, 2018.

[8] FIPA Agent Communication Language Specifications. http://www.fipa.org/repository/aclspecs.html, last accessed October 2021.

[9] F. Ezzrhari, H. Bensag, M. Youssfi, O. Bouattane & O. K. Abra : "Towards a New Micro Agents Middleware for Massively Distributed Systems". In: 2018 6th International Conference on Multimedia Computing and Systems (ICMCS), May 2018.

[10] HAZELCAST. https://docs.hazelcast.com/hazelcast/latest/index.html, last accessed October 2021.