International Journal of Engineering Trends and Technology
ISSN: 2231–5381 / https://doi.org/10.14445/22315381/IJETT-V72I11P131

Volume 72 Issue 11, 322-332, November 2024
© 2024 Seventh Sense Research Group®

Original Article

Towards Secure Agent Migration in Mobile Cloud Computing Using JADE Framework

Khadija El miloudi[1], Yassine Aggagui[2], Abderrahim Abdellaoui[3]

[1,2,3]*Systems Engineering Laboratory, ENSAK, Ibn Tofail University, Kenitra, Morocco.*

[1]*Corresponding Author : elmiloudi.khadija@gmail.com*

Received: 03 April 2024 Revised: 09 August 2024 Accepted: 17 August 2024 Published: 29 November 2024

Abstract - *Mobile Cloud Computing (MCC) has achieved significant success in recent years by delivering resources to mobile devices and making them available to any device for computation and execution in the cloud. Speaking of computing the execution, the Multi-Agent System (MAS) allows mobile devices to run an agent-based application in which agents can be migrated from one node to another to continue the execution within a system. The goal is to combine the advantages of MCC and MAS to compute application agents between mobiles to reduce power consumption and improve application performance. This paper provides a novel agent migration architecture for mobile cloud computing built on the JADE platform and considers security concerns using the ECDH, ECDSA, and SIMON algorithms.*

Keywords - *Agent migration, ECDH, ECDSA, JADE, Mobile cloud computing, SIMON.*

1. Introduction

Mobile Cloud Computing (MCC) has become increasingly important in the mobile environment, thanks to the vast services and benefits provided by Cloud Service Providers (CSPs) such as Google, Amazon, and Microsoft. According to [1], mobile cloud computing is "a rich mobile computing technology that leverages unified elastic resources of varied clouds and network technologies toward unrestricted functionality, storage, and mobility to serve a multitude of mobile devices anywhere, anytime through the channel of Ethernet or Internet regardless of heterogeneous environments and platforms based on the pay-as-you-use principle". The advancement of MCC can be observed in the enhanced performance of mobile devices and mobile applications, whereby the hosting of mobile execution has migrated to cloud-based resources.

This can have a significant impact, particularly for mobile devices with limited resources. Such improvements include increased processing and data storage capacity, longer battery life, and increased reliability [2]. On the other hand, mobile users can benefit from a wide selection of services that can reduce power usage and energy consumption while running applications on mobile devices. These services are divided into several groups, including software, platforms, and infrastructure [3], [4]. As a result, consumers will benefit from using servers, storage, applications, and processing power with minimal management effort. Three approaches can be used to implement MCC. The first is the conventional Client-Server design, where the mobile device serves as a client, and application processing is conducted in the cloud servers. The second is called a cloudlet, a solution meant to help cloud servers by giving customers access to resources with quicker response times [5]. The third is a virtual resource cloud created by collaborating with mobile devices on the same network, with peer-to-peer communication between nodes [6]. This study focuses on the third approach of MCC implementation. Several solutions have been presented on this subject, including migrating Agents based on Multi-Agent Systems (MAS). Niu Haichun and Liu Yong [4] introduce a mobile agent-based task seamless migration approach for mobile cloud computing. It is a solution designed to overcome the limitations of standard mobile agent migration by monitoring the migration of Agent tasks over the network and specifying each task's purpose with regard to time delay.

However, security is crucial to secure Agent data across the network. As described in ref [7], a method based on Elliptic Curve Cryptography is proposed to secure Agent data exchange. In order to increase processing power and decrease mobile phone resource consumption, this paper presents a new agent migration scheme that complies with mobile cloud computing security standards. This scheme provides a framework for making decisions about agent migration and optimizing agent transport. It also provides a technique for monitoring agent migration and securing data flow between agents in the platform. The scheme is based on the Java Agent Development Framework (JADE) and Elliptic curve Diffie Hellman algorithm with lightweight encryption using the SIMON algorithm.

2. Review of Related Works

This section presents some of the research literature around the agent migration and agents task migration models used in mobile cloud computing and multi-agent systems. A few instances of current agent/agent-task mobility works will be introduced. Then, a collection of papers that offer several approaches to enhance the security and privacy of agent data will be examined. In ref [8], Ametller et al. present a mechanism for agent mobility using the ACL communication language proposed by FIPA. The "mobile-agent-description" concept outlined by FIPA handles the agent's description during migration. The transmission of the agent consists of establishing the transmission protocol and the ontology that will be utilized for the movement and extraction of the agent between two platforms. The migration model used is handled directly by the platform specified by the FIPA specification.

In reference [9], Vedran Vyroubal and Mario Kušek present a novel migration agent model for the Android platform, which allows agents to be sent between the J2SE and Android platforms. A pair of techniques have been introduced, one based on encapsulating the agent (including its code and state) within an ACL message that FIPA defines. The second step is to migrate each agent's function. By converting JVM byte codes to DALVIK VM bytes-codes on both platforms, they can solve the byte code mismatches between the J2SE and Android platforms. A framework for improving mobile device resource usage is presented by Angin and Bhargava [10] as an example of how to implement agent migration. In this example, mobile application agents migrate to a designated cloud server to be executed. Based on each server's computing power and communication link speed, the migration manager selects which server will be utilized to migrate the mobile agent.

In [11], the asynchronous migration technique is employed to enable migrations to occur almost anywhere in the user codes, while the Twin Method Hierarchy that is suggested reduces the overhead that arises from state-restoration codes during regular execution. Another method for migrating agents between incompatible platforms is described in ref [12]. The problem was resolved by creating platform-independent agents with the ability to migrate between incompatible systems using a special agent architecture. Kamouri et al. [13] offer a novel method based on a cryptography trace to identify the agent and the home platform to ensure integrity during the agent transmission in order to enforce the security requirement during the agent migration. Additionally, an agent named SOS is mentioned as a potential option for monitoring the migration of Lightweight agents to new platforms. This agent is in charge of defending the LW agent from malicious attacks, including DOS attacks. Another method that delivers two benefits is referenced [14]. The first is the agent's robust anonymous authentication using elliptic curve cryptography. The second one includes a tracing tool for monitoring the execution of agent code migrating across several platforms. As a result, when a new platform receives an agent, this method can aid in detecting harmful code. Angin et al. [15] demonstrate an effective way for securing mobile agent mobility in mobile cloud computing, where mobile agents move from a mobile device to a remote container on the cloud server side. The suggested approach enables the mobile agent to use integrity checkpoints to defend itself against tampering while the agent runs in the container to identify tampering attacks.

3. Theoretical background

Before presenting an agent migration framework based on the JADE platform that considers security needs, a quick overview of the JADE platform and the multiple security algorithms employed in the proposed approach below is provided.

3.1. JADE Platform

Java Agent Development Framework (JADE) is a software framework that is compliant with the standard FIPA for developing multi-agent systems. It is fully implemented in Java and facilitates the development of intelligent agents in heterogeneous environments under the FIPA specifications. JADE provides a distributed agent platform across multiple machines (which is unnecessary to execute the same OS), besides a peer-to-peer communication between agents based on the message-passing paradigm. Figure 1 demonstrates the main architecture of JADE. The JADE platform consists of containers, each of which can execute zero or more agents simultaneously and is identifiable by a unique name. Agents communicate with one another via ACL messages. As seen in Figure 1, the platform allows containers to be executed across several hosts in a distributed system. The JADE architecture includes a specific container named Main Container, which comprises two fundamental agents responsible for managing the platform: The DF (Directory Facilitator) agent offers a yellow page where agents can list their services and provide a directory of the agents available on the platform. The AMS (Agent Management System) agent is a platform controller; aside from registration control and agent authentication, it is the only one able to start, kill, suspend, and resume the agent, container, or hall platform [10].

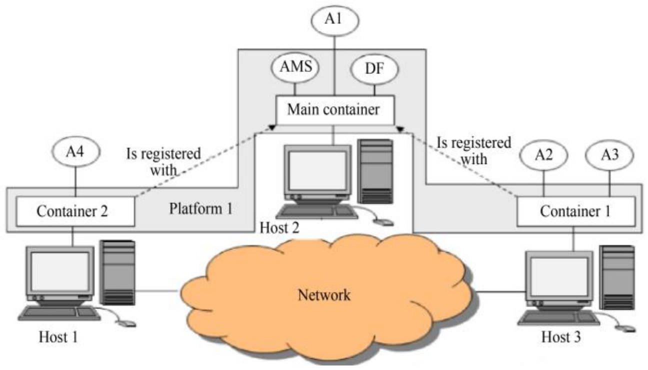


Fig. 1 The JADE architecture [16]

Table 1. ECC parameters

Parameter	Description
p	The field in which the curve is defined over
a, b	The values used to define the curve
G	The generator point
n	The prime order of G
h	Cofactor

3.2. Elliptic Curve Diffie Helman

3.2.1. Elliptic Curve Cryptography (ECC)

Definition

ECC is an asymmetric algorithm designed as an alternative to RSA, commonly used in SSL certificates. Compared to other algorithms, ECC delivers smaller keys while maintaining the same level of security for devices with limited resources. The ECC is used for key exchange, digital signature, and encryption via key exchange with other symmetric algorithms (in this study, the Elliptic curve with Simon and Speck algorithms is used).

The following equation defines the ECC:

$$y^2 = x^3 + ax + b \tag{1}$$

a and b are two arbitrary constant points on the elliptic curve, which must satisfy the following condition:

$$4a^3 + 27b^2 \neq 0. \tag{2}$$

An Elliptic curve [5] is a curve passing from a point that solves the equation below:

$$E:(x, y) \mid y^2 = x^2 + ax + b \tag{3}$$

Domain Parameters

They are a set of parameters to be agreed upon between two parties to use the ECC, which are:

3.2.2. Diffie Helman (DH)

Definition

The Diffie Helman (DH) is a public key protocol to exchange cryptographic keys across the network. It is an interesting method to securely compute keys between two parties to generate a shared key for encryption purposes.

Proposition

Let Bob and Alice be the two persons who wish to exchange a secret key. The sharing procedure cannot begin until all parties agree on the parameters. The protocol applies to the multiplicative group of integers modulo p, where g is a primitive root modulo p and p is a prime number. p and g are necessary for determining the shared secret, which must be between 1 and p-1. Below is an example:

- Bob and Alice agreed on a specific p = 24 and g = 4 to use while sharing secrets.
- Alice generates a private integer a = 5, then computes $A = g^a mod\, p$: $A = 4^5 mod\, 24 = 16$
- Bob generates a private integer b = 4, then computes $B = g^b mod\, p$: $B = 4^3 mod\, 24 = 16$
- Alice sends $S = B^a mod\, p$:
  $$S = 16^5 mod\, 24 = 16$$
- Bob sends $S = A^b mod\, p$:
  $$S = 16^4 mod\, 24 = 16$$

As a result, Alice and Bob have the same secret, which is 16.

3.3. Simon Lightweight Algorithm

SIMON is a lightweight encryption scheme from the Feistel-based block ciphers' family, where each block is divided into two halves [17]. SIMON was introduced by the National Security Agency (NSA) in June 2013. It supports different blocks and key sizes and provides high performance on software and hardware.

Consequently, SIMON can be suitable for various platforms and light for hardware-based devices (Table 2). As illustrated in Figure 2, the round function uses the key schedule to run several rounds on the same block data. The round function combines two blocks of n bits to generate a single encrypted word. The block logic comprises logical functions of AND, XOR, and left rolls.

The following describes the steps of round function used to encrypt the message.

- The first step is to generate a bit field utilizing the next block via a left roll of 1 (S^1) and 8 (S^8) with a logical AND operating on the current block as an XOR.
- In the second step, a left roll of 2 (S^2) is utilized to create a bit field in the next block. This field is then used as an XOR against the outcome of the previous operation.
- The final step is to XOR the keyword with the previous result. After that, the two blocks are switched, and the procedure is repeated.

The block and key sizes of the function determine the number of rounds.

3.4. Elliptic Curve Digital Signature Algorithm (ECDSA)

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a Digital Signature Algorithm (DSA) that employs keys generated from elliptic curve cryptography. It is a remarkably effective algorithm based on public key cryptography. ECDSA has three primary functions.

Table 2. The combination of block size, key size, and number of rounds using the Simon algorithm

Block size (bits)	Key size (bits)	Rounds
32	64	32
48	72	36
	96	36
64	96	42
	128	44
96	96	52
	144	54

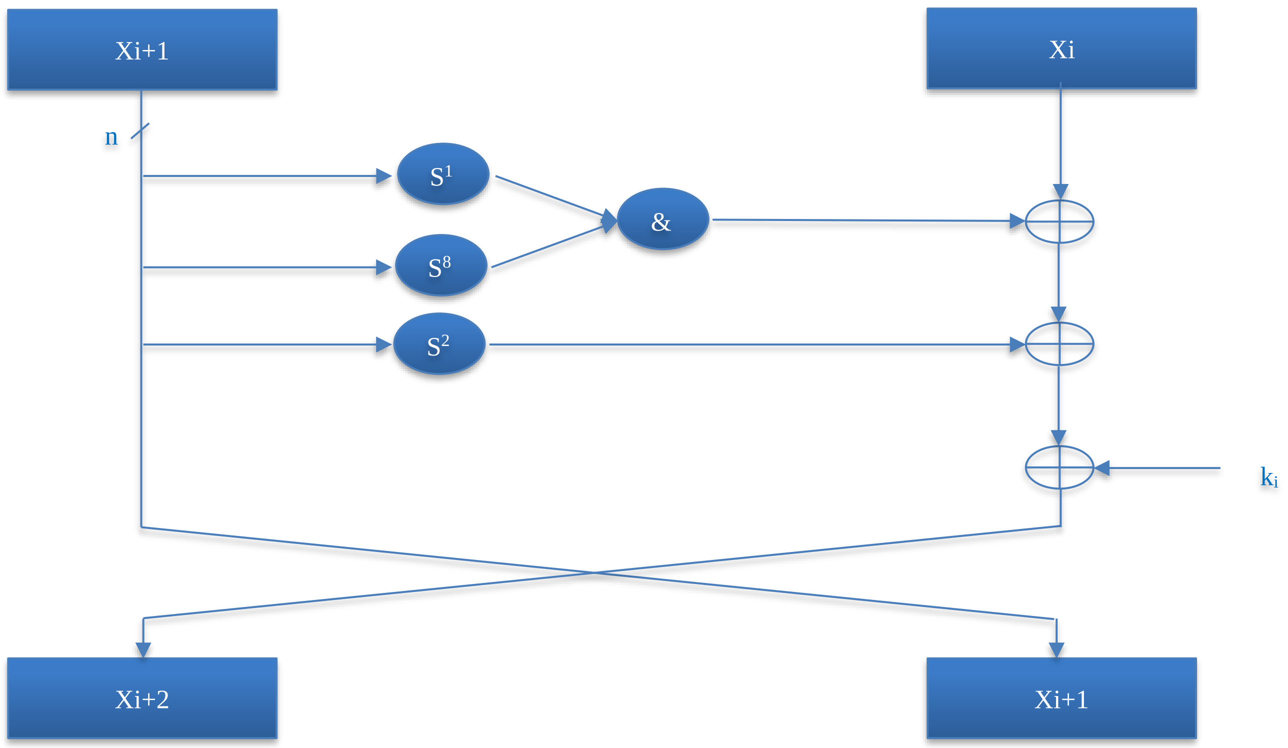


Fig. 2 The SIMON round function [18]

3.4.1. Key Pair Generation

The ECDSA key pair comprises

- Private key (privkey): it is a random integer in the range of 0 to n-1.
- Public key (pubKey): is the point on the elliptic curve that results from multiplying the generator point G by the private key.

pubKey = privKey * G

3.4.2. Signature

To sign a message, the following actions are required:

- Randomly choose a number k between 1 and n-1.
- Calculate $(i, j) = k \times G$ where G is an elliptic curve base point.
- Calculate $= i \; mod \; n$; if $x = 0$, return to step 1.
- Calculate $y = k^{-1} \times (H(m) + d) \; mod \; n$ where $H(m)$ is the result of a cryptographic hash on the message m to be signed, and d is the private key.
- If $y = 0$, return to step 1.

The signature is (x, y).

3.4.3. Verification

The signature verification technique uses the message and signature (x, y) as inputs to establish the signature's validity. Here is a summarised version of the verification procedure. The algorithm authenticates the sender by verifying the digital signature of the message following steps below:

- Verify if Q is on the curve where Q is the sender's public key.
- Check that x and y are between 1 and n-1.
- Calculate
  $(i, j) = (H(m)y^{-1} mod \; n) \; G + (xy^{-1} mod \; n)Q$

The signature is valid if $x = i \; mod \; n$, invalid otherwise.

4. The Proposed Approach

Multi-Agent Systems (MAS) is a robust paradigm that offers numerous benefits for creating complex systems and applications. The presence of a software agent in MAS aids in resolving system issues by simplifying system activity. Each node in an agent-based system can have either a static agent that runs locally on the node or a mobile agent that can be migrated to another node in the system [9]. In addition to the advancement of mobile devices, the evolution of MAS provides the opportunity to create agent-based smartphone applications. The concept is to use the migration of agents in MAS to compute agents between mobile devices for mobile cloud computing, allowing us to reduce smartphone power consumption. This research provides a new dynamic framework for migrating mobile agent-based application partitions to mobile cloud computing while maintaining security needs. A computing agent is used to connect mobile devices inside the same network. The migration of agents, which distributes the execution of agents over several mobile devices (servers offer resources to the client), aids in the reduction of CPU, memory, and battery usage on mobile devices. Our strategy relies on the JADE platform because of its compatibility with the FIPA standard.The architectural elements will be first described to enhance comprehension of the suggested methodology. Then, every component of our approach will be presented, along with an overview of each component. Next, the entire process that the mobile agent goes through to migrate from the client to the servers will be outlined, including the steps of device discovery, migration decision, ECDH authentication, registration, agent migration, and migration monitoring.

4.1. Architecture of the Proposed Agent Migration Security Framework

Figure 3 illustrates our proposed agent migration security framework based on the JADE platform. It displays every component of our approach on both the client-side and server-side.

The client-side consists of the following components

- The client-interface is in charge of initiating communication with the server, identifying and authenticating devices and transferring agents to the server (i.e., other mobile devices).
- The client manager decides which agents are candidates for migration and which server will receive the transmission (migration decision).
- The Main Container (provided by the JADE framework) connects and allows all remote containers (i.e. those executed on other mobile devices) to join the client on the same platform. The main container always runs on the client side.
- The App Agents are the application agents running in local containers on the client side and execute all application related tasks.

The following are the components offered on the server-side:

- The server-interface performs the same function as the client-interface and receives the agent transferred by the client-manager.
- The Server Manager collects the address information of the main container from the client and uses it to create a remote container on the same platform as the client. He also receives agents from the server interface and adds them to the remote container.
- The remote container where migrated agents will be executed.

4.2. Mobile Agent Mobility Lifecycle

In order to achieve mobile agent mobility between containers through the network, the following phases must be performed (Figure 4): Every phase is defined to fulfill the purpose of the subsequent phase, and each has a specific role. The following section delves further into each component.

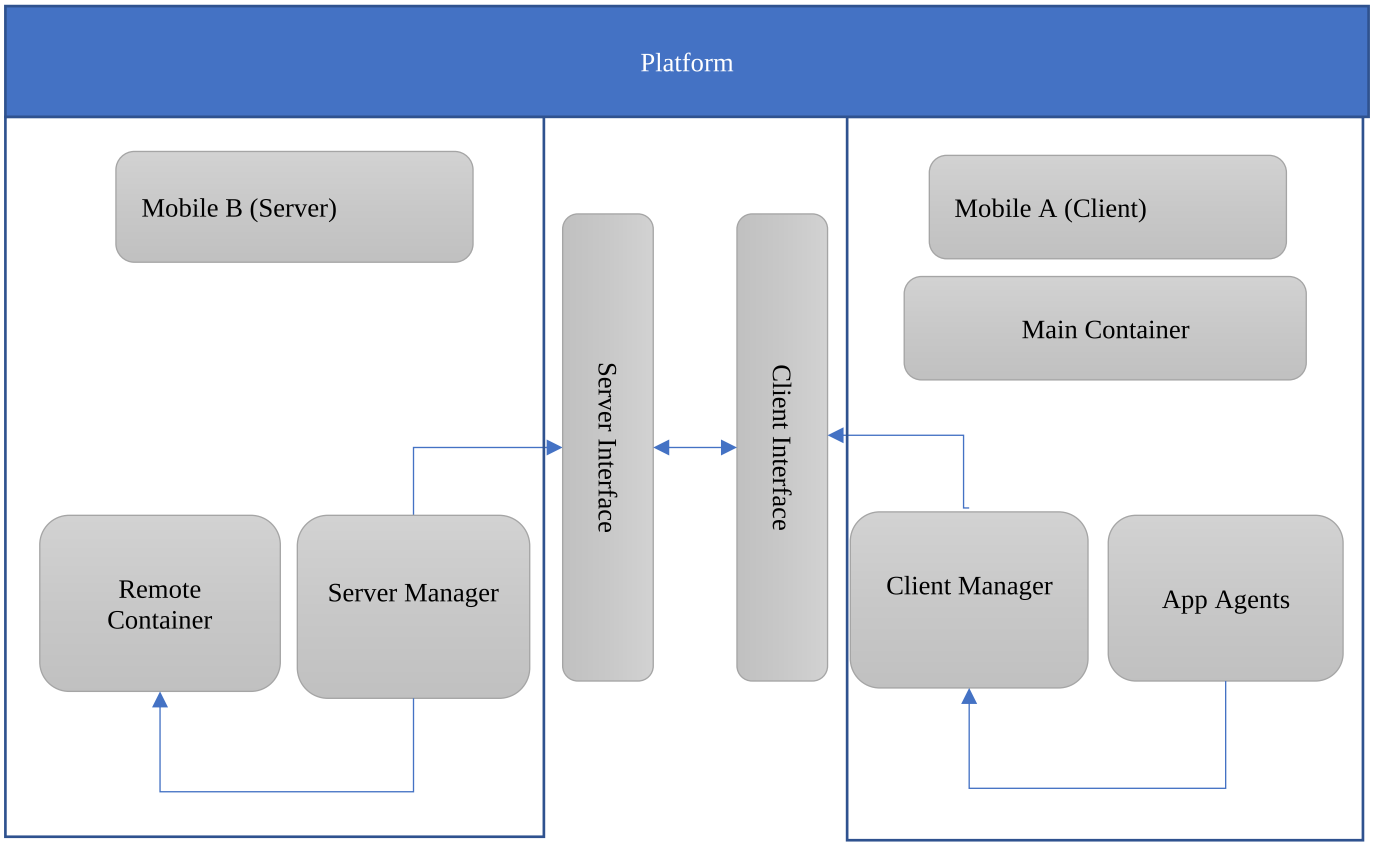


Fig. 3 Overview of the suggested agent migration architecture

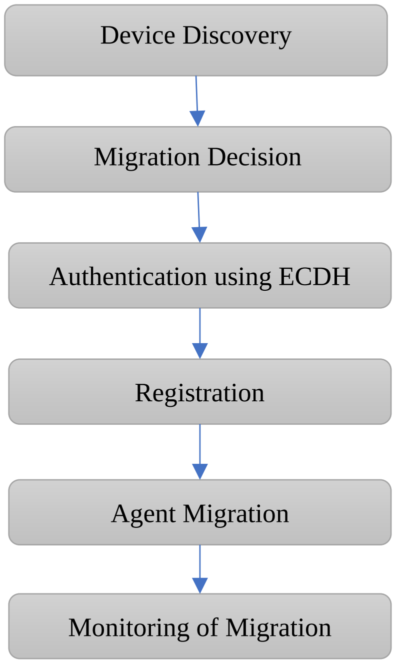


Fig. 4 Mobile agent migration lifecycle

4.2.1. Device Discovery

In the device discovery phase, the purpose is to detect all mobile devices that operate as servers and provide resources to clients in the same network. In order to accomplish this, a broadcast request is sent to every node connected to the network using the UDP protocol.

All servers listen on the same port, allowing us to distinguish the servers from other connected nodes. Upon receipt of a request from the client, the server returns the server's IP address along with details about the phone's total free memory, battery life, and CPU usage (Figure 5).

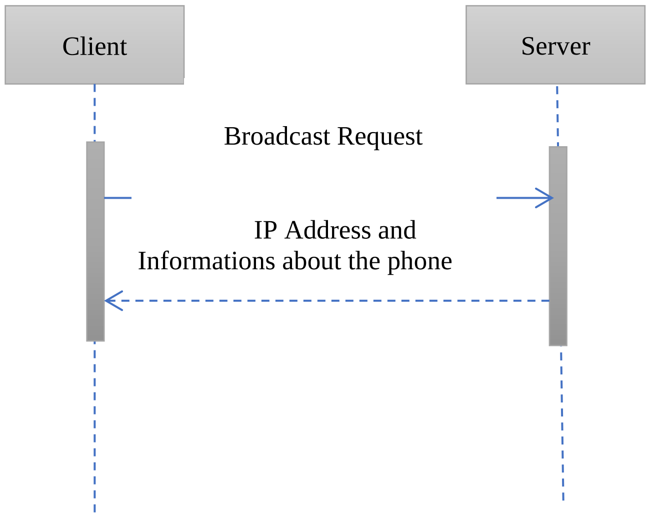


Fig. 5 Sequence diagram of communication between server and client

4.2.2. Migration Decision

As previously stated, the client receives the IP addresses of all servers within the network along with the phone's current health (memory, CPU, and battery life). These details allow for the classification of the devices and the determination of the agent migration destination (Figure 6). Each agent's CPU and memory usage is classified on the agent side. The memory consumed before and after creating each agent in the container is calculated to achieve that.

In Figure 8, The ObjectMemoryBefore method returns the total consumed memory before creating the agent. The ObjectMemoryAfter method returns the memory size of the agents after creation. It is calculated by subtracting the total memory used before and after creating the agent. Therefore, the average CPU time of each behavior in the agent is calculated (Figure 7):

n: number of behaviors in the agent.

m: number of executions of behavior.

a: The CPU usage is based on the behavior during execution.

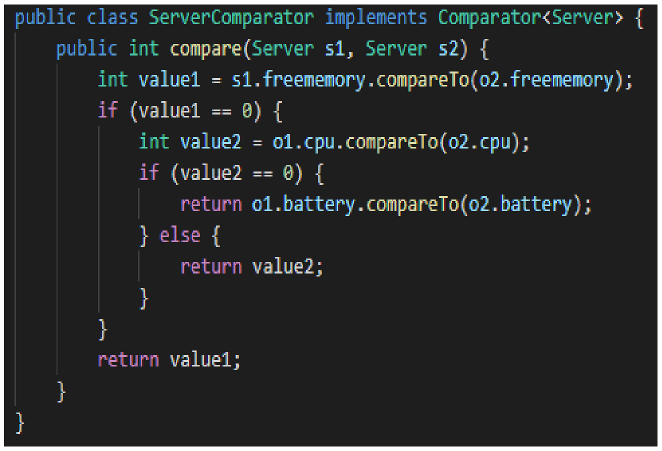
d: The total CPU usage by the behavior.

c: The range of CPU usage by the agent.

$$\mathrm{d} = \sum_{j=1}^{m} (a_j) \tag{4}$$

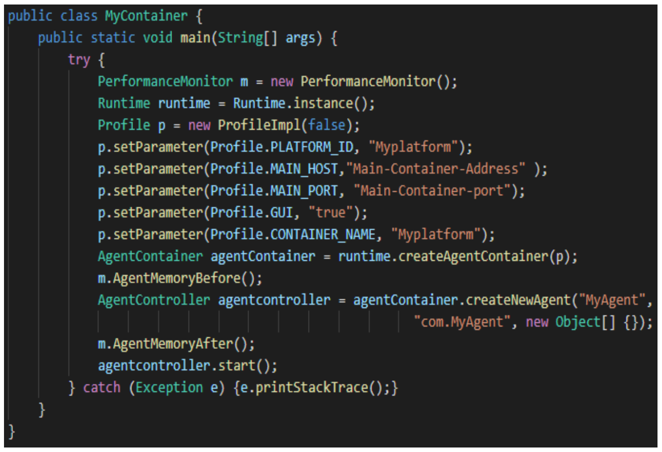
$$\mathrm{c} = \frac{(\sum_{i=0}^{n} (d_i))}{n} \tag{5}$$

As seen in Equation (4), the overall CPU usage is obtained as the sum of the total CPU usage by the behavior during execution which allows calculation of the range of CPU for all behaviors in the agent (Equation (5)).



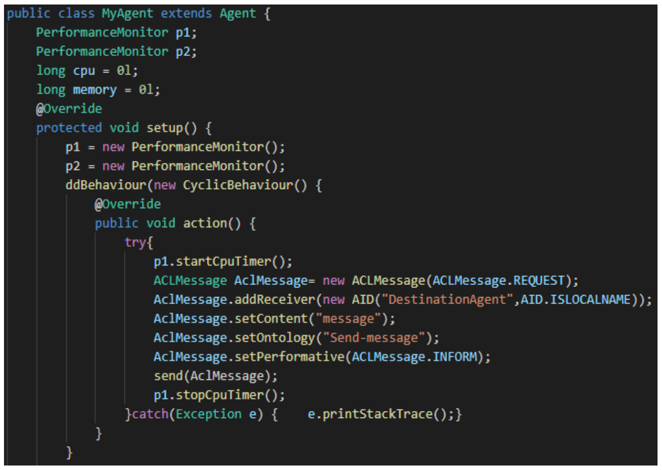
```
public class ServerComparator implements Comparator<Server> {
    public int compare(Server s1, Server s2) {
        int value1 = s1.freememory.compareTo(o2.freememory);
        if (value1 == 0) {
            int value2 = o1.cpu.compareTo(o2.cpu);
            if (value2 == 0) {
                return o1.battery.compareTo(o2.battery);
            } else {
                return value2;
            }
        }
        return value1;
    }
}
```

Fig. 6 Code segment of the Servercomparator class

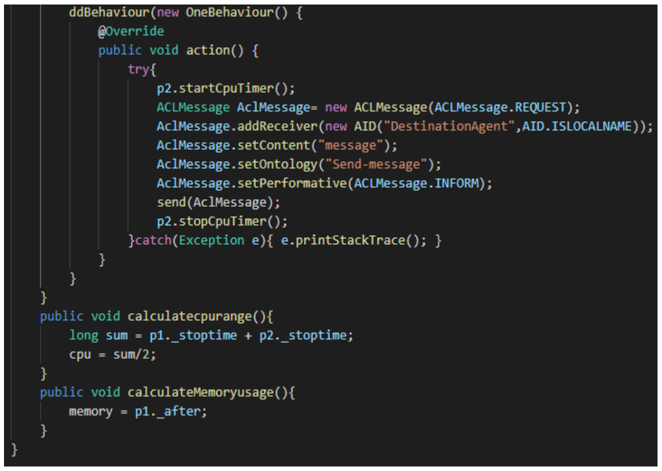


```java
public class MyContainer {
    public static void main(String[] args) {
        try {
            PerformanceMonitor m = new PerformanceMonitor();
            Runtime runtime = Runtime.instance();
            Profile p = new ProfileImpl(false);
            p.setParameter(Profile.PLATFORM_ID, "Myplatform");
            p.setParameter(Profile.MAIN_HOST,"Main-Container-Address" );
            p.setParameter(Profile.MAIN_PORT, "Main-Container-port");
            p.setParameter(Profile.GUI, "true");
            p.setParameter(Profile.CONTAINER_NAME, "Myplatform");
            AgentContainer agentContainer = runtime.createAgentContainer(p);
            m.AgentMemoryBefore();
            AgentController agentcontroller = agentContainer.createNewAgent("MyAgent",
                                                "com.MyAgent", new Object[] {});
            m.AgentMemoryAfter();
            agentcontroller.start();
        } catch (Exception e) {e.printStackTrace();}
    }
}
```

Fig. 7 Code segment of the MyContainer class that displays the calculation memory of an agent during the creation



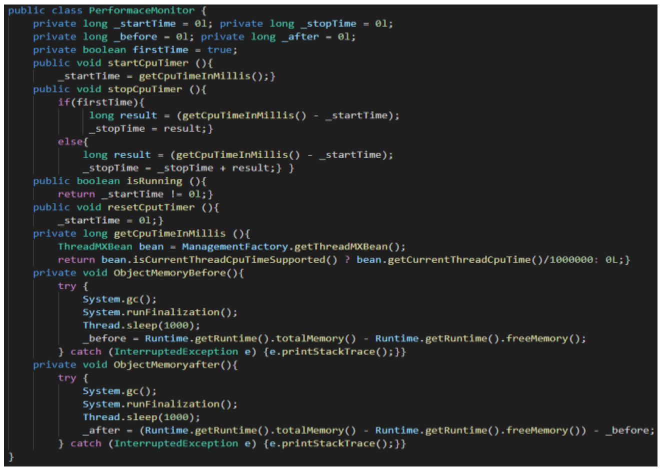
```java
public class MyAgent extends Agent {
    PerformanceMonitor p1;
    PerformanceMonitor p2;
    long cpu = 0l;
    long memory = 0l;
    @Override
    protected void setup() {
        p1 = new PerformanceMonitor();
        p2 = new PerformanceMonitor();
        ddBehaviour(new CyclicBehaviour() {
            @Override
            public void action() {
                try{
                    p1.startCpuTimer();
                    ACLMessage AclMessage= new ACLMessage(ACLMessage.REQUEST);
                    AclMessage.addReceiver(new AID("DestinationAgent",AID.ISLOCALNAME));
                    AclMessage.setContent("message");
                    AclMessage.setOntology("Send-message");
                    AclMessage.setPerformative(ACLMessage.INFORM);
                    send(AclMessage);
                    p1.stopCpuTimer();
                }catch(Exception e) {    e.printStackTrace();}
            }
        }
```



```java
        ddBehaviour(new OneBehaviour() {
            @Override
            public void action() {
                try{
                    p2.startCpuTimer();
                    ACLMessage AclMessage= new ACLMessage(ACLMessage.REQUEST);
                    AclMessage.addReceiver(new AID("DestinationAgent",AID.ISLOCALNAME));
                    AclMessage.setContent("message");
                    AclMessage.setOntology("Send-message");
                    AclMessage.setPerformative(ACLMessage.INFORM);
                    send(AclMessage);
                    p2.stopCpuTimer();
                }catch(Exception e){ e.printStackTrace(); }
            }
        }
    }
    public void calculatecpurange(){
        long sum = p1._stoptime + p2._stoptime;
        cpu = sum/2;
    }
    public void calculateMemoryusage(){
        memory = p1._after;
    }
}
```

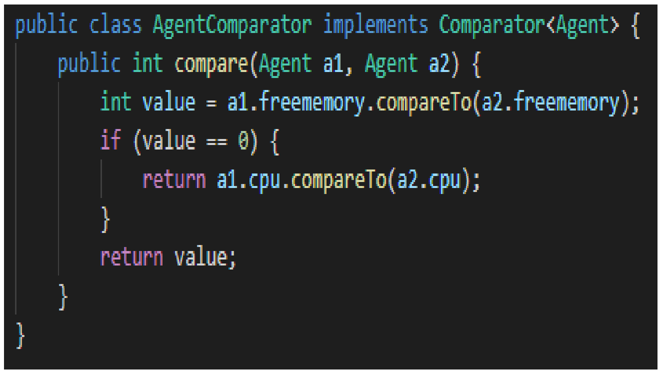
Fig. 8 Code segment of the MyAgent class displays the calculation of CPU range and memory of an agent

As a result, the agent's memory capacity and CPU use can be determined (Figures 7 and 8). The Performance Monitor class (Figure 9) is used by the classes Mycontainer and MyAgent to do the computation. There are methods in this class that facilitate memory and CPU monitoring. These factors were utilized to categorize the agents and determine each agent's final destination (Figure 10). The computation is accurate for a given period of time at the moment of agent migration. There are two goals in classifying the server's power and resource consumption by the agent. The first is to choose which servers are going to authenticate. The second is determining which agent consumes the most resources, necessitating migration to a specific server from among the available servers in the network.



```java
public class PerformaceMonitor {
    private long _startTime = 0l; private long _stopTime = 0l;
    private long _before = 0l; private long _after = 0l;
    private boolean firstTime = true;
    public void startCpuTimer (){
        _startTime = getCpuTimeInMillis();}
    public void stopCpuTimer (){
        if(firstTime){
            long result = (getCpuTimeInMillis() - _startTime);
            _stopTime = result;}
        else{
            long result = (getCpuTimeInMillis() - _startTime);
            _stopTime = _stopTime + result;} }
    public boolean isRunning (){
        return _startTime != 0l;}
    public void resetCputTimer (){
        _startTime = 0l;}
    private long getCpuTimeInMillis (){
        ThreadMXBean bean = ManagementFactory.getThreadMXBean();
        return bean.isCurrentThreadCpuTimeSupported() ? bean.getCurrentThreadCpuTime()/1000000: 0L;}
    private void ObjectMemoryBefore(){
        try {
            System.gc();
            System.runFinalization();
            Thread.sleep(1000);
            _before = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
        } catch (InterruptedException e) {e.printStackTrace();}}
    private void ObjectMemoryafter(){
        try {
            System.gc();
            System.runFinalization();
            Thread.sleep(1000);
            _after = (Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()) - _before;
        } catch (InterruptedException e) {e.printStackTrace();}}
}
```

Fig. 9 Code segment of the PerformanceMonitor class



```java
public class AgentComparator implements Comparator<Agent> {
    public int compare(Agent a1, Agent a2) {
        int value = a1.freememory.compareTo(a2.freememory);
        if (value == 0) {
            return a1.cpu.compareTo(a2.cpu);
        }
        return value;
    }
}
```

Fig. 10 2Code segment of the AgentComparator class

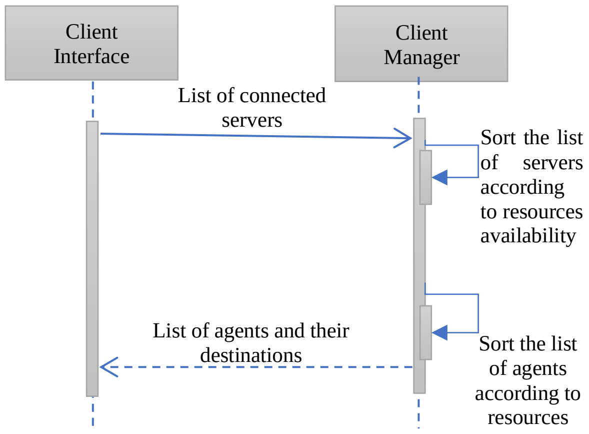

Client Interface
Client Manager
List of connected servers
Sort the list of servers according to resources availability
List of agents and their destinations
Sort the list of agents according to resources

Fig. 11 Sequence diagram of migration decision

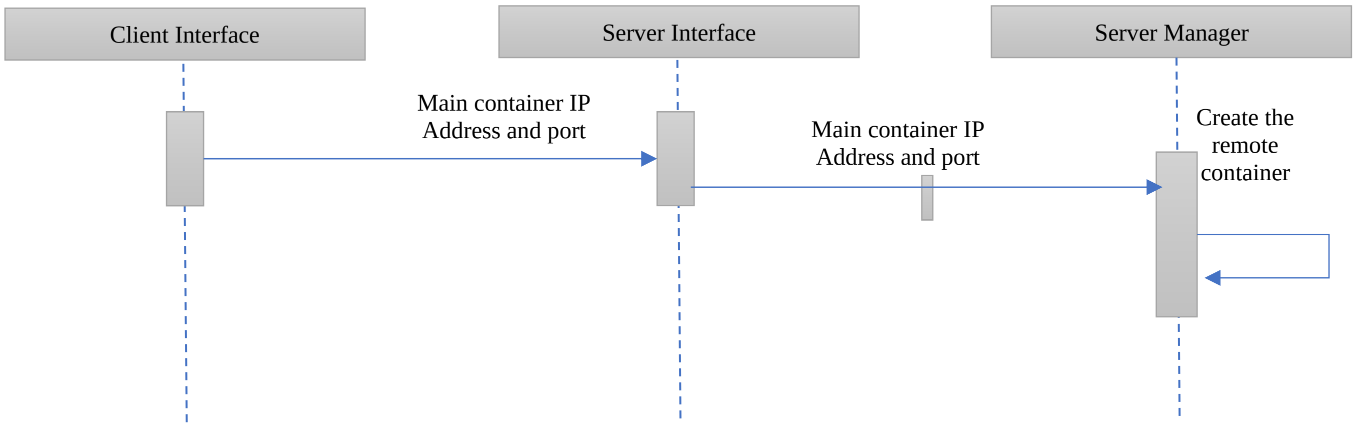

Client Interface
Server Interface
Server Manager
Main container IP Address and port
Main container IP Address and port
Create the remote container

Fig. 12 Sequence diagram of the creation and addition of a container to the platform

4.2.3. Authentication Using ECDH

As previously stated, the client manager determines which agents will participate in the migration and which will be the destination. A list based on server-side resource availability and agent-side resource use is the outcome of this classification. The authentication phase focuses on creating a secure tunnel between the client and the server that will be used to transfer agents. A secure connection to the servers is established using the list of servers created by the client-manager.

To accomplish this purpose, the Elliptic curve Diffie Helman key exchange is employed between the client and server to ensure secure communication:

Step 1: To perform the key exchange, the domain parameters (p, G, a, b, n, and h) must match on both the client and the server.

Step 2: A random private key, d, that must be between 1 and n-1, is generated by the client and server.

Step 3: The point Q is calculated, the public key:

- For the client: $Q_c = d_c \times G$ (The generator point).
- For the server: $Q_S = d_S \times G$ (The generator point).

Step 4: To calculate the shared key, the client and the server compute their public keys.

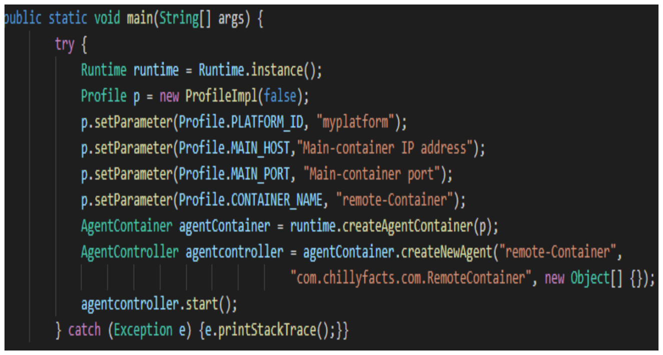
For the client:

- Confirm that Qs is on the curve and that $Qs \times n = \infty$.
- Calculate the shared secret $R, Rc = Qs \times dc$.

For the server:

- Verify if the Qc is on the curve, and check if $Qc \times n = \infty$.
- Calculate the shared secret $R, Rs = Qc \times ds$.

Since the client and server share the same key, they can safely communicate data.



```
public static void main(String[] args) {
    try {
        Runtime runtime = Runtime.instance();
        Profile p = new ProfileImpl(false);
        p.setParameter(Profile.PLATFORM_ID, "myplatform");
        p.setParameter(Profile.MAIN_HOST,"Main-container IP address");
        p.setParameter(Profile.MAIN_PORT, "Main-container port");
        p.setParameter(Profile.CONTAINER_NAME, "remote-Container");
        AgentContainer agentContainer = runtime.createAgentContainer(p);
        AgentController agentcontroller = agentContainer.createNewAgent("remote-Container",
                                      "com.chillyfacts.com.RemoteContainer", new Object[] {});
        agentcontroller.start();
    } catch (Exception e) {e.printStackTrace();}}
```

Fig. 13 Code segment of adding a container to the platform

4.2.4. Registration

During this phase, the remote container is launched on the same platform as the client. In order to create the remote container, the server interface obtains the IP address and port of the main container on the client side upon the client's authentication with the server. To create and integrate the remote container into the platform, the server interface provides the server manager with the IP address and port of the main container (Figure 12).

The JADE framework permits the addition of a remote main container's IP address and port to any container during the creation phase. The process of adding a container to a platform is shown in Figure 13.

4.2.5. Agent Migration

Loading Agent

The migration process consists of transmitting the agent from local containers on the client side to a remote container on the server, accompanied by the agent's ability to execute its tasks. For this, the client manager uses ACL messages to retrieve the agent object directly from the agent. For example, MyAgent (MA) requests the binary and status of the agent via an ACL message sent by the client-manager to the agent. As a result of the MA's response, the client manager sends the MA to the client interface, which initiates the migration via Java Socket.

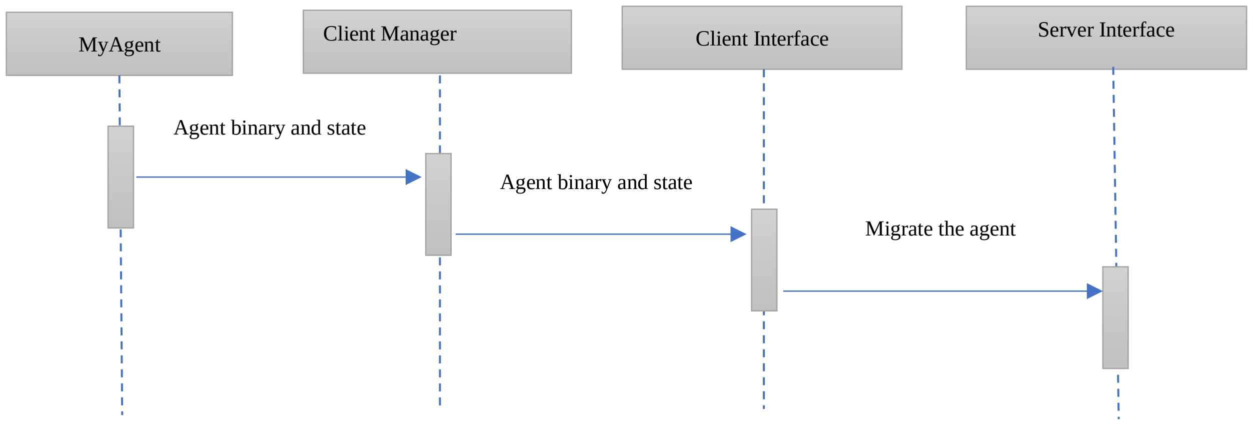
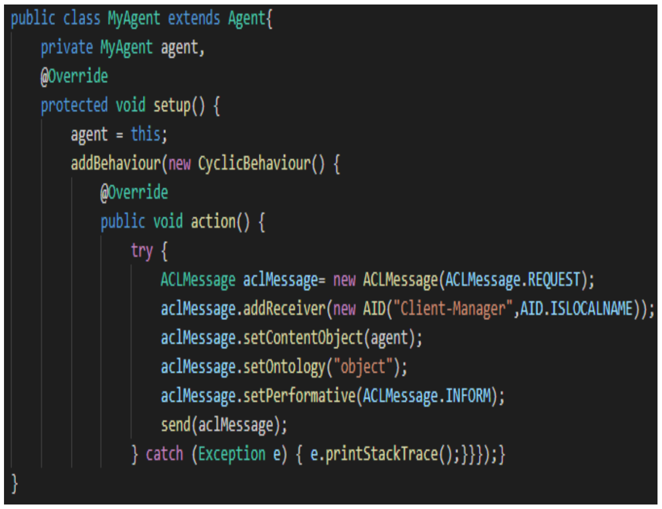


Fig. 14 Sequence diagram of agent migration

The figure below shows how to send agent binary and state from the agent itself.



```
public class MyAgent extends Agent{
    private MyAgent agent,
    @Override
    protected void setup() {
        agent = this;
        addBehaviour(new CyclicBehaviour() {
            @Override
            public void action() {
                try {
                    ACLMessage aclMessage= new ACLMessage(ACLMessage.REQUEST);
                    aclMessage.addReceiver(new AID("Client-Manager",AID.ISLOCALNAME));
                    aclMessage.setContentObject(agent);
                    aclMessage.setOntology("object");
                    aclMessage.setPerformative(ACLMessage.INFORM);
                    send(aclMessage);
                } catch (Exception e) { e.printStackTrace();}}});}
}
```

Fig. 15 Code segment that shows how to send agent state

Serialization and Encryption

Serialization is employed because it allows data to be read by other devices. Agents in the JADE framework are serializable by default because they extend the Agent class, which is already serializable. Upon receiving the agent binary and its current state, the client manager sends it to the client interface.

After authentication, the client interface prepares the agent for migration. Agent a is encrypted using the SIMON algorithm and the shared keys. The cyphertext c, $c = E(s, b, a)$, is gated. The block size b and key size n need to be manually defined on both sides (a and c are two binary arrays), after which a random k is chosen from $[1, n-1]$.

Next, the ECDSA algorithm is used to sign the message to retrieve the point e before the migration: $e \mid (x, y) = H(d, c, k)$.

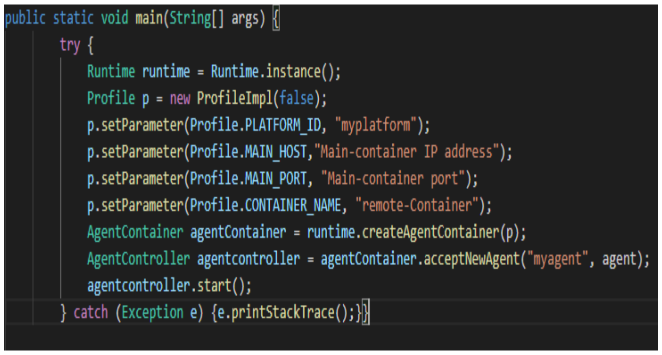
Table 3. The proposed approach parameters

Parameter	Description
E	Encryption method
D	Decryption method
H	Signature method
V	Signature verification method
a	The agent that we want to migrate
b	The block size used for encryption
c	The cyphertext is the result of encryption method E.
k	The random value used to sign c.
e	The signature value of the signing method.
Q	The public key of the client.
d	The private key of the client

In the end, c and e are computed on the server. On the server side, it is verified if x and y are between 1 and n-1. It is also checked if Q is a point from the curve E. Then, the point v is obtained by verifying c using the ECDSA algorithm, $v = V(c, e, Q)$. The agent a is obtained by decrypting c using the SIMON Algorithm, $a = D(k, b, c)$.

Adding Agent to the Remote Container

With the JADE Framework, an agent can be added to a container by providing the agent binary to the containers that have the agent's name assigned to them (Figure 16).



```
public static void main(String[] args) {
    try {
        Runtime runtime = Runtime.instance();
        Profile p = new ProfileImpl(false);
        p.setParameter(Profile.PLATFORM_ID, "myplatform");
        p.setParameter(Profile.MAIN_HOST,"Main-container IP address");
        p.setParameter(Profile.MAIN_PORT, "Main-container port");
        p.setParameter(Profile.CONTAINER_NAME, "remote-Container");
        AgentContainer agentContainer = runtime.createAgentContainer(p);
        AgentController agentcontroller = agentContainer.acceptNewAgent("myagent", agent);
        agentcontroller.start();
    } catch (Exception e) {e.printStackTrace();}}
```

Fig. 16 Code segment that shows how to add an agent to a container

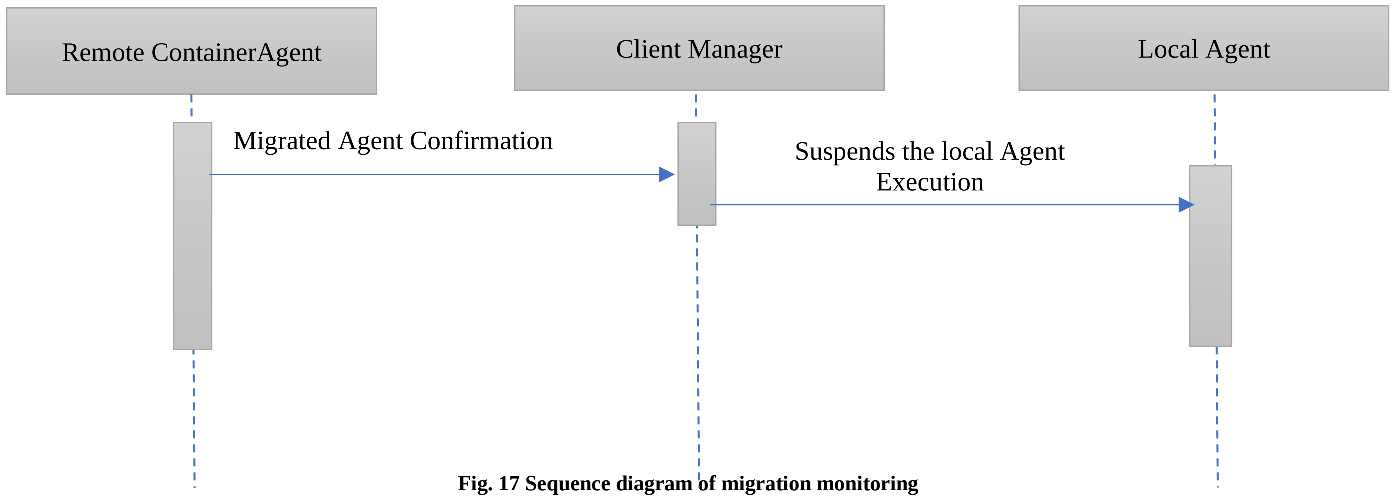

Remote ContainerAgent
Client Manager
Local Agent
Migrated Agent Confirmation
Suspends the local Agent Execution

Fig. 17 Sequence diagram of migration monitoring

4.2.6. Migration Monitoring

When the agent arrives in the remote container, it sends an ACL message to the client manager to confirm the migration. Afterwards, the client manager pauses the agents' execution within local containers. If no confirmation is detected, the agent continues the execution normally.

5. Conclusion and Future Work

In order to minimize the consumption of CPU, memory, and battery life consumption of mobile devices, this paper provides a novel mechanism for computing agents across mobile phones. Agents can be moved between incompatible platforms thanks to our method, which is based on the Jade platform. Additionally, an elliptic curve digital signature is employed to guarantee the integrity of the migration, and an elliptic curve Diffie Helman is employed for authentication. The SIMON cipher algorithm is used to encrypt data sent between agents. This method determines each agent's destination before migration by using migration decisions. The decision is made based on server resources and agent consumption. The benefits of this study include the proposal of a system that offers authentication, confidentiality, and integrity for an agent migration while simultaneously considering mobile device resource consumption. Cloud resource management optimization [19] is one of the research fields that has lately received notice and deserves further exploration.

A critical field for further research is code injection. A primary issue encountered by mobile agents is the possibility of third parties manipulating the way the agents execute. Successful code injection might have potentially harmful consequences, such as enabling the client to receive untrusted data.

References

[1] Dijiang Huang, Huijun Wu, *Mobile Cloud Computing Taxonomy*, Mobile Cloud Computing, pp. 5-29, 2018. [Google Scholar] [Publisher Link]

[2] Parkavi Ravi, Priyanka Chinnaiah, and Sheik Adullah Abbas, *Cloud Computing Technologies for Green Enterprises: Fundamentals of Cloud Computing for Green Enterprises*, IGI Global Scientific Publishing Platform, pp. 395-414, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[3] Jason Carolan et al., *Introduction to cloud computing architecture*, White Paper, 1st ed., 2009. [Google Scholar] [Publisher Link]

[4] N. Haichun, and L. Yong, "A Mobile Agent-Based Task Seamless Migration Model for Mobile Cloud Computing," *Proceeding 2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA)*, Ottawa, ON, Canada, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[5] Layth Muwafaq et al., "A Survey on Cloudlet Computation Optimization in the Mobile Edge Computing Environment," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 14, no. 1, PP. 1-13, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[6] Niroshinie Fernando, Seng W. Loke, and Wenny Rahayu, "Mobile Cloud Computing: A Survey," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 84-106, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[7] Yousra Berguig et al., "Mobile Agent Security Based on Mutual Authentication and Elliptic Curve Cryptography," *International Journal of Innovative Technology and Exploring Engineering*, vol. 8, no. 12, pp. 2509-2517, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[8] Joan Ametller, Sergi Robles, and Joan Borrell, "Agent Migration over FIPA ACL Messages," *Proceedings International Workshop on Mobile Agents for Telecommunication Applications*, Springer, Berlin, Heidelberg, vol. 2881, pp. 210-219, 2003. [CrossRef] [Google Scholar] [Publisher Link]

[9] Vedran Vyroubal, and Mario Kušek, "Task Migration of JADE Agents on Android Platform," *Proceedings of the 12th International Conference on Telecommunications*, Zagreb, Croatia, pp. 123-130, 2013. [Google Scholar] [Publisher Link]

[10] P. Angin, and B.K. Bhargava, "An Agent-Based Optimization Framework for Mobile-Cloud Computing," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, vol. 4, no. 2, pp. 1-17, 2013. [Google Scholar] [Publisher Link]

[11] Ricky K.K. Ma, and Cho-Li Wang, "Lightweight Application-Level Task Migration for Mobile Cloud Computing," *Proceedings of the IEEE 26th International Conference on Advanced Information Networking and Applications*, Fukuoka, Japan, pp. 550-557, 2012. [CrossRef] [Google Scholar] [Publisher Link]

[12] P. Misikangas, and K. Raatikainen, "Agent Migration Between Incompatible Agent Platforms", *In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, Taipei, Taiwan, pp. 4-10, 2000. [CrossRef] [Google Scholar] [Publisher Link]

[13] Sophia Alami-Kamouri et al., "Mobile Agent Security Based on Cryptographic Trace and SOS Agent Mechanisms," *Journal of Communications*, vol. 15, no. 3, pp. 221-230, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[14] Hind Idrissi, "Anonymous ECC-Authentication and Intrusion Detection Based on Execution Tracing for Mobile Agent Security," *Wireless Personal Communications*, vol. 94, no. 3, pp. 1799-1824, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[15] Pelin Angin, Bharat Bhargava, and Rohit Ranchal, "A Self-Protecting Agents Based Model for High-Performance Mobile-Cloud Computing," *Computers & Security*, vol. 77, pp. 380-396, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[16] David Grimshaw, JADE Administration Tutorial, Jade Platform, Ryerson University, 2010. [Online]. Available: https://jade.tilab.com/documentation/tutorials-guides/jade-administration-tutorial/

[17] Ashutosh Dhar Dwivedi, and Gautam Srivastava, "Security Analysis of Lightweight IoT Encryption Algorithms: SIMON and SIMECK," *Internet of Things*, vol. 21, pp. 1-11, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[18] Ray Beaulieu et al., "The SIMON and SPECK Lightweight Block Ciphers," *Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, San Francisco, CA, pp. 1-6, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[19] Aldo H.D. Mendes et al., "MAS-Cloud+: A Novel Multi-Agent Architecture With Reasoning Models for Resource Management in Multiple Providers," *Future Generation Computer Systems*, vol. 154, pp. 16-34, 2024. [CrossRef] [Google Scholar] [Publisher Link]