# Mobile agents management, modeling and implementation

Mohamed Khamis, Ibrahim Al-Bedewi
*Faculty of Computing and Information Technology, KAAU, Saudi Arabia*

To gain large amount of processing power in an inexpensive way, the computational power of a group of small computers can be accumulated rather than the power of a single expensive supercomputer. Internet, in this case, represents the communication media that bridge the distance gaps between these small computers and bring them in a single computing paradigm. A scheme is, therefore, required to manage these distributed computers. However, the required communication bandwidth can represent an obstacle for achieving this target in an efficient way, especially when many messages need to be exchanged among the different computing nodes. Mobile agent technology represents a viable solution of this problem, but a framework is needed to allow agents to communicate, migrate and execute in a secure environment. This paper presents an agent platform that provides applications with these services using a simple interface. Both the modeled platform and its implementation are described and a demo is given to show how to integrate this platform with agent applications.

للحصول على قدرة حاسوبية كبيرة بطريقة غير مكلفة يمكن أن يتم ذلك بواسطة تجميع القدرة الحاسوبية لمجموعة من الحواسيب الصغيرة بدلا من استخدام حواسيب عملاقة ذات قدرة عالية ولكنها باهظة الثمن. شبكة الإنترنت في هذه الحالة قد تمثل الوسط الذي يجمع هذه الحواسيب المتفرقة معا في بيئة حاسوبية واحدة. يتطلب هذا بناء نظام لإدارة الحسابات الموزعة ولكن الحيز الترددي الكبير المطلوب لانجاز التواصل سيكون عقبة في سبيل تحقيق ذلك خاصة عند وجود مجموعة كبيرة من الحواسيب التي تتبادل العديد من الرسائل بين بعضها البعض. تكنولوجيا الوكيل المتنقل تمثل أحد الحلول البديلة المرئية لهذه المشكلة وفي مثل هذه الحالة فإن الوكيل المتنقل سيحتاج إلى بيئة تسمح للتطبيقات المختلفة بإنشاء وكلائها واتصال هذه الوكلاء فيما بينها والتمكن من الهجرة من ماكينة إلى أخرى لتنفيذ الكود الخاص بهذه العملاء على الماكينة الأخرى إما لأنها هي الوحيدة التي تملك المصادر اللازمة لذلك أو لإمكانية توزيع الحمل على الحواسيب المختلفة للحصول على أداء أفضل. تنفيذ هذا الحل يحتاج إلى بناء إطار لعمل الوكيل المتنقل يسمح للتطبيقات المختلفة بإنشاء وإرسال الوكيل من ماكينة إلى ماكينة أخرى ثم تنفيذ الكود الخاص بها وكل هذا يجب أن يتم في بيئة آمنة ذات شفافية. في هذه الورقة البحثية نعطى نموذج لخدمتي الهجرة وإدارة الوكلاء المتنقلين والخوارزميات الخاصة بهما وقد تم تنفيذ وبناء الخدمات الأساسية لهذا الإطار واختبارهما بواسطة مثال في نهاية هذه الورقة يوضح كيفية استخدام هذا الإطار بواسطة التطبيقات المختلفة.

**Keywords:** Distributed application, Agent framework, Modeling, Mobile agent, Platform

## 1. Introduction

In order for mobile agents to work and perform their tasks properly they need an environment that provides the execution for agents, this environment is called Agent Server, Agent Framework, Agent Platform, or Agency. Agent environment hosts all agents which are running in parallel on a machine, and enables agents to access its resources. Also, this environment allows agents to communicate with each other locally or remotely and to migrate to other environments. The environment manages the creation, execution, suspension, transfer, and termination of mobile agents [1]. It provides a directory service to look up for agents' locations [2]. This service is required to allow agents to communicate and to migrate. Agent Security [3] is one of the most important issues, since agent is an executable code which could cause damage for the machine it migrates to. The environment, like any other application, is subject to errors and corruption, so it has to adopt a fault tolerant technique to be reliable. Below, several of the existing mobile agent environments are briefly described.

## 2. Background

Aglet [4] is an agent platform that allows java objects to move from one host to another spanning network. Aglet system does not offer strong mobility or migration. For security reasons, all Aglet agents communicate with each other through a proxy object. The Aglet proxy object acts as a shield to protect the agent against malicious agents. The proxy consults the agent security manager to check the permission. An Aglet' agent has a unique identifier used through the life cycle of the agent.

Agent Remote Action (ARA) [5] is another platform that facilitates the execution of mobile agents in a heterogeneous network. ARA mobile agent is a code that can migrate to another machine during its execution to resume its execution there. This means that, the ARA system can capture the execution state and sends it with the agent to the destination host, i.e. ARA system provides strong mobility. ARA system deals with the agent independently of its programming language since it uses different language run-time interpreters in order to execute multiple mobile agents programmed with multiple interpreted languages. ARA core is the part of the system which provides different services such as mobility, communication, security, and resource management.

Concordia [[6]] is a java-based mobile agent environment to develop mobile agents. It introduces many services for executing mobile agents, these services are implemented in many components known as managers. The mobility of mobile agents is performed by an agent manager, which also provides the agent execution. This environment introduces inter-agent communication in two forms: distributed events and agent collaboration. This environment allows developers to support agents by adding new services to Concordia servers using Service Bridge components. Concordia introduces a Directory Manager to register agents, which in turn enables locating the desired servers. Concordia provides security model to protect agents from access or tampering while they have been stored in persistent storage. Further, it protects agents during transmission via network. Also, Concordia protects server resources from harmful and unauthorized access or modification. One of the important services introduced by Concordia is the recovery from failure service that protects both agents and server against system crash by saving them in persistent storage; thereby recovering the original state is feasible. Concordia uses a message Queuing system, which works as "store and foreword". This system allows the saving of all transmitted agents locally until they successfully reach the remote server. Concordia Queue Manager is a component that performs the latter technique, while it uses java RMI in its communication.

TACOMA [[7],[8]] agent is a code that has the ability to migrate to remote host across the network, during its execution. TACOMA project focuses on the support of operating system to agent based computing, and how to structure applications based on the mobile agent model. TACOMA consists of interpreter for TCL scripting and C programming languages to provide the access to its features Tracy [[9]] is a java-based mobile agent that allows building and managing stationary and mobile agents. Tracy introduces basic services required to manage and transmit mobile agents.

Mobile Object and Agent (MOA) [[10]] is an Agent Environment (AE) implemented in Java Virtual Machine (JVM) without any modification to JVM. It is developed to achieve four goals; the first goal is to provide collaboration among agents or their users. The second goal is to provide resource control in order to protect agents and hosts against denial of service attack. The third goal is to provide a standard way to configure and customize the system's components. The fourth goal is to increase the base in which agents can visit other agent systems. The architecture of MOA consists of components that support naming and locating agents, mobility, communication, and resource management. MOA does not support strong mobility. Many security issues were not implemented, such as authentication, authorization and integrity checking.

Grasshopper [[12]] is the first mobile agent platform, which conforms to MASIF standards

in addition to FIPA. Grasshopper was developed by GMD FOKUS and IKV++. It is based on JVM, and it represents the actual runtime environment for both kind of agents (mobile and stationary). It enables the programmers to develop multi-agent systems with mobile agents and provides an Agent Communication Language (ACL) to support the communication and cooperation among agents. Grasshopper's Distributed Agent Environment (DAE) is composed of regions, places, agencies, and agents. Grasshopper agency is composed of two parts: the first one is the core agency, which support the minimal services required for executing the agents. Whereas, the second parts is one or more places, which are the container in which the agents are executed. The core agency provides many services such as communication service, registration service, management service, security service, and persistence service. The communication in grasshopper is supported by using multiple protocols to achieve remote interactions, such as, CORBA, RMI, and plain sockets. Optionally the last two protocols can be protected by SSL. Also, this environment introduces four modes of communications, such as synchronous, asynchronous, dynamic, and multicast communication. The security in grasshopper is implemented in two levels: the external and the internal. The external level protects the interactions between the distributed agencies and region registries by using SSL protocol while the internal level protects the agencies resources against unauthorized access, and protects the agents from each others, this level is based on java security mechanism.

MAPNET [[14]] is another alternative to java-based environments; it is based on .net framework and conforms to MASIF specification. This environment is written in C#, and supports both migration and inter-agent communication.

## 3. Agent migration service

The available agent platforms don't provide all the necessary component modules. Building a mobile agent platform to provide the different modules is a project funded at the KAAU University. The main goals of this project is improving the performance and efficiency by adopting efficient algorithms for the different services and introducing all the necessary modules in a single platform as well as building a prototype of this platform. In this paper a model and implementation for the communication, migration and management services are given. These services represent the core functionality of any platform. The migration model considers a layering concept to facilitate developing the framework. This model is described below.

### *3.1 . Agent migration model architecture*

The migration service is the module responsible for transmitting agents to their destination. The migration model is introduced to support migration of agents between agencies. The proposed model considers the previously stated migration design issues in order to fulfill the dedicated tasks. It adopts the concept of layers. As the layering makes the design functionality modular, so the system could be evolved easily.

The proposed model is called Four-Layers Model (FLM). It consists of four layers as depicted in fig. 1. This modularity of the system facilitates the developing and upgrading for each layer separately. Further, it enables discovering the errors of the system easily and in turn minimizes the time required to debug and fix the problems.

The functions of the different layers are summarized as follows:

- *Communication Layer:* it opens TCP channel to migrate agent from the sender to the receiver.
- *Mapping Layer:* it queries the DNS server in order to resolve the host symbolic name when an agent is required to migrate to other servers.
- *Migration Layer:* it guarantees reliable migration for mobile agents.
- *Compacting Layer:* it is to transform agents to a form that allows them to migrate rather than the form of execution (marshalling and un-marshalling operations). Agent Migration Block (AMB)

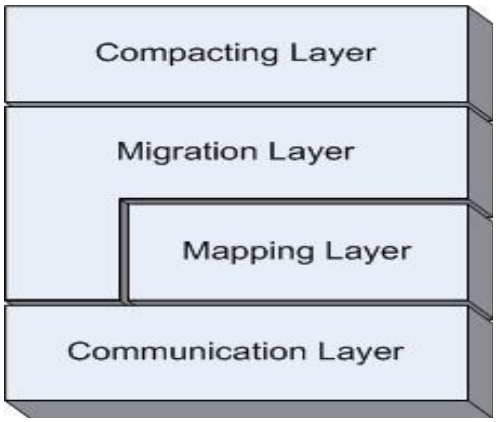


Fig. 1. The four layers migration model

A mobile agent within migration service is represented by a data structure known as *Agent Migration Block* (AMB). The AMB holds information about particular agent. This information includes identifier, agent' name, and message of the agent which in turn includes the state, code and agent itinerary, beside the format' information of the message and other values as shown in fig. 2.

For reliable migration, two AMB are required. The first in the sender, whereas the second in the receiver site. As soon as the agent migration is accomplished successfully, the AMB in sender will be removed, whereas the AMB in receiver will be maintained in the compacting layer.

*3.2 . Agent migration scenario*

The agents in the system start their itinerary in the management service on the machine they are residing on. Once an agent needs to migrate to another remote machine after finishing its execution task on the current machine, the management service will launch agent for migration through calling the service of the compacting layer, of the migration service fig. 3. Sequence of operations has to be taken in order to complete the migration process for the agent as follows:

1. The compacting layer builds the Agent Migration Block and calls the migration layer to add AMB to its outgoing list.
2. The mapping layer resolves the destination name to corresponding IP address then, the mapping layer will handle both of the IP address and the marshaled agent to the sender component of the communication layer.
3. The sender component will send the agent to the destination via network using TCP protocol.
4. At the destination site, the AMB is received by the receiver object in the communication layer.
5. The received AMB is accumulated in incoming list within the migration layer.

Once the agent server is completely accumulated in incoming list, it will be sent to the upper layer. The agent is un-marshaled in the compact layer and is handled to the management service, which immediately starts by putting the agent in its execution cycle.

*3.3 . Platform characteristics*

Weak mobility is used in this platform, This kind of implementation offers many advantages to applications such as network information gathering, remote filtering and any application that doesn't require capturing the agent state as in case of strong mobility. Strong mobility is important only when there is a need to build reliable and dynamic load balancing systems.

Routing technique in this platform is static, where all desired hosts (itinerary) to be visited by the agent, are listed in a table with the tasks which are needed to be executed at each host. While the agent roams according to its itinerary, the platform manager fetches this table to determine the task to be executed by the agent at the current host as well as the next host of the agent' itinerary. (The Implication of the Migration Policy).

The current migration service adopts *push-all-to-next* strategy. This strategy transmits the entire agent code as well the agent state at once to next destination. It suffers from a considerable drawback. The drawback appears when the agent code refers to many dependencies and not all of these dependencies are used in every destination during the agent journey. Hence there are superfluous code that will be transmitted, which in turn affect the bandwidth of the

underlying network. However, if the agent code doesn't have these many dependencies, this policy will be relatively faster, since agents are transmitted with all of their parts (object states, and code) at once.

To alleviate the above problem, the developer can divide the single agent with multi functions into multi agents, each with only a single function. The whole functionality of the big agent is then gathered from these many small agents. This process is known as *Agent Normalization*. Using such normalization technique the required bandwidth is reduced and the fastness is obtained.

Once agent migration process is started, the entire agent information is marshaled into a migration message, as shown in fig. 2.

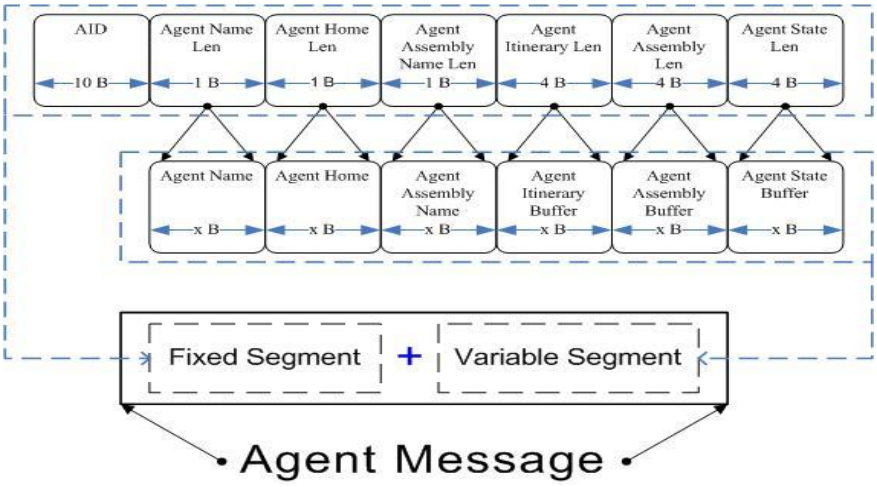


Fig. 2. Agent message format.

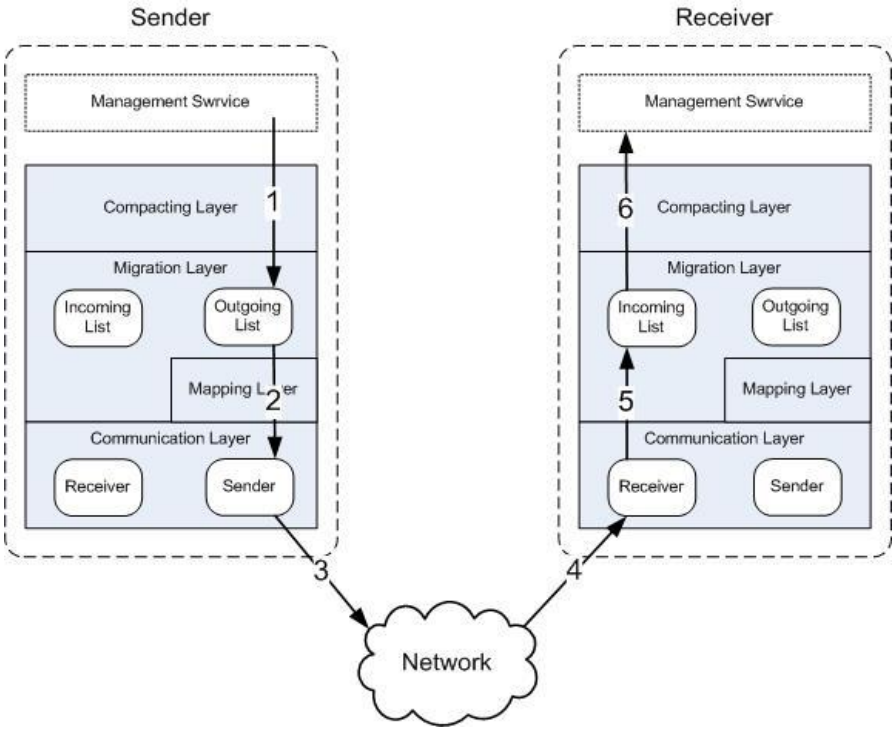


Fig. 3. Agent migration.

## 4. Agent management service

The Agent Management Service (AMS) includes creation, deletion, and execution of agents. It is important to note that, AMS has the authority to manage the agent life cycle. The agent life cycle model consists of set of states that an agent can exist in one of them during its lifetime. The transition of an agent from state to the other will depend on the arisen events.

The agent life cycle model shows the states of the agents during their life and the different events on which an agent transit from any state to another one. According to FIPA agent management specification the agent life cycle model is defined explicitly by introducing a state diagram, which consists of five states, they are *initiated, active, suspended, waiting,* and *transit* states, where any agent must exist in one of these states. FIPA also define ten transitions events, which can be described as: *create, invoke, destroy, quit, suspend, resume, wait, wake up, move,* and *execute* transitions. With regard to FIPA the active state is the central state, through which an agent has to pass during moving to another state.

### *4.1 . Agent Life-cycle model*

For the sake of efficiency, the management service of the platform here, embeds few of the above states in one, and adopts the agent life cycle model shown in fig. 4, which consists of three states and seven transitions. The three basic states are:

1. *The suspension state*: the agent has resources allocated but it is not executing. The agent enters in this state in one of three cases. The first case, when the agent enters the system for first time; the second, when the agent finishes its execution and is ready to migrate; the third, when the agent is in the migration state after finishing the current task but needs to execute another task on the current machine.
2. *The execution state*: the manager assign an agent from the suspension queue to run by allocating a thread for this agent to perform its task.
3. *The migration state*: in which the management system checks up whether the next task of an agent, is to be executed on the local host or on a remote one according to the agent itinerary. On the local host, the management system will push back the agent to the suspension state, otherwise it call the migration service to send the agent to next hop in the itinerary (if any), or to send it back to its home then de-allocates the resources used by the agent afterward.

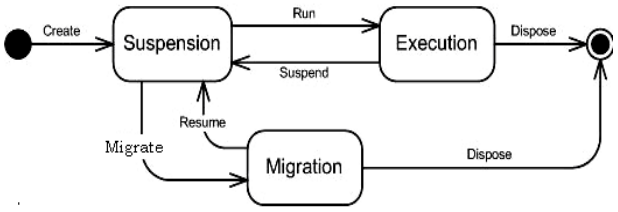


Fig. 4. Mobile agent life-cycle model.

Some important notes have to be noticed here, the proposed agent life cycle model explained considers the suspension state as the central state, i.e. when an agent wants to move between two states it must pass through the suspension state first. Also, another important note is that, the management system is not concerned about the internal state of agents or even their specific goals.

As mentioned earlier, this model has six transitions, which describe how agents can change their state. The state transition of agents can be listed as:

1. *Create;* it causes creation of an agent by representing it in the system with *Agent Control block* (ACB). This transition assigns the agent in suspension state.
2. *Run;* it brings an agent to the execution in an independent thread, hence, the agent state is changed accordingly.
3. *Suspend;* it makes an agent to be in its passive form after its task has been fulfilled.
4. *Migrate;* it brings the agent into migration state to check up for the next hop in the agent itinerary.
5. *Resume;* it resumes the agent back from migration state to suspension state in order to execute the next task on the current machine.
6. *Dispose;* it causes the termination of the agent and the release of its resources, either after fulfilling its tasks or after sending the agent to the migration service to continue the remaining tasks on other machines.

## 5. Queuing service

The management service is implemented in the server to control and to coordinate the execution of all agents within the environment using three queues: Suspension Queue (SQ), Execution Queue (EQ), and Migration Queue (MQ). Any agent within the system has to join one of these queues. Further, the agent is represented in these queues using the structure of its ACB. The ACB is composed of six fields:

1. *The identifier;* it is a unique string that is used to access the agent during its life time.
2. *The state;* it is a value that reflects the state of the agent. When an agent joins the SQ, the agent will take one of two values "Execute" or "Migrate". If the agent joins other queues, this value will be “NONE”
3. *The code path;* it is a string that indicate the path of the agent source file.
4. *The Itinerary;* it is an object that maintains the itinerary of this agent.
5. *The agent-object;* it is an object that represent the real instance of the agent in memory.
6. *The agent thread;* it is an object thread in which the agent object executes its task.

In the implementation of the management service, each state in the model is represented by a queue and each transition in the life-cycle model is represented by a method.

### *5.1 . Agent queuing algorithm*

The queuing algorithm is used to judge and coordinate the execution of agents within the environment. It is explained in the *activity diagram* depicted in fig. 5. When an agent is received by the migration service, it will be sent to the management service by invoking *the Create* method. The *Create* method constructs the ACB for the agent and adds it to SQ. At this point, the suspension state (SS) is set to "Execute" value. If the thread pool has thread available, then the management service will send the ACB of this agent to EQ and invokes the *Run* method that take the ACB as an argument. The *Run* method executes the agent task in a thread from the thread pool. When the agent completes its task, the management service invokes the *Suspend* method to change the ACB suspension state to "Migrate" value, removes the ACB from EQ, and adds it to SQ. once the ACB is added to SQ, the management service invokes *Migrate* method to remove the ACB from SQ and add it to MQ. In the MQ, the management service checks the next task. If the next task is a local one, then the ACB will be removed from MQ, its suspension state will be changed to execute value, and finally it will be added to SQ, otherwise, if the next task has to be executed remotely then, the agent information will be sent to management service in order to rebuild AMB to send the agent to the remote machine through the migration service. Finally, the ACB is disposed.

## 6. Application interaction with the Framework

The server (framework) which has been built provides the basic services, which are required to execute mobile agents and control their life-cycles. The server is built as a library that has an interface which allows applications to access the services of this server. Like many other mobile agent servers such as Aglets, Grasshopper, Concordia, Mapnet, etc, any agent class has to extend a predefined base class to be considered as a mobile agent.

### *6.1. Using system API library*

Developers need to access and benefit from the services of this mobile agent server. So, the server has an API that enables developers to write mobile agents easily. Further, these API make developers able to map out the movement and the tasks for their mobile agents in simple manner. The API is located in a library which has two classes; the first one is *MobileAgent* class, whereas the second one is *Itinerary* class.

The *MobileAgent* class is the super class for all mobile agents' classes. The key issue from extending this class is to give the mobility ability for user-agent classes. It has three attributes and one method. The attributes are:

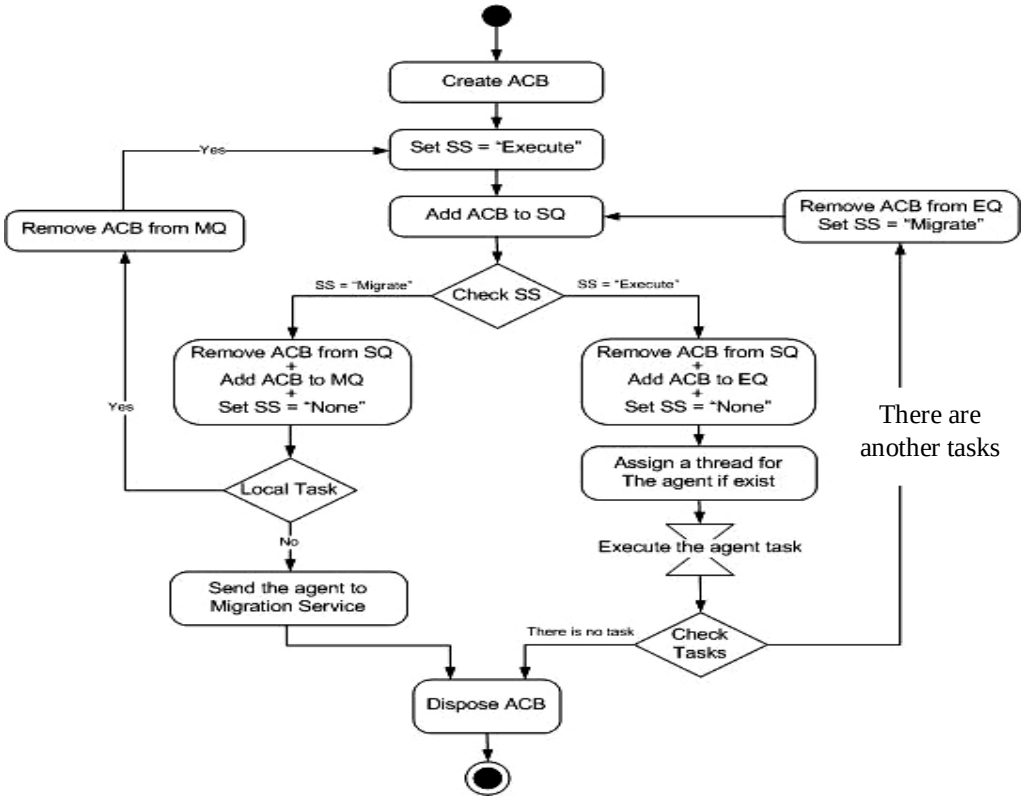


Fig. 5. The queuing algorithm.

1. The agentname: to assign a user-symbolic name for the agent.
2. The agentassembly: to specify the agent assembly file path.
3. The agentitinerary: to assign a itinerary object to the agent.

The only method of this class is the *GO* method. When the *GO* method is invoked, the agent will migrate to the local mobile agent server to start its journey. This method builds the agent message, described previously in Migration service.

The *Itinerary* class is the class that allows developers from to determine the journey of their mobile agents, in addition to the tasks at each station of the journey. To do so, developers have to initiate an object from this class and invoke its *SetSiteTask* method. This method takes two string parameters; the first one is the name of the destination or IP address to where the agent migrate, whereas the second parameter is the name of the user's agent method that has to be executed at that destination. Eventually, the itinerary object has to be assigned to a user mobile agent using its AgentItinerary property.

### *6.2. Lunching mobile agent*

In order to create a mobile agent, developers need to extend the *MobileAgent* base class, which belongs to the *MicroLib* library. Moreover, they also need to flag the mobile agent class with the *serializable* attribute.

```
[Serializable]
public class CDAgent: MobileAgent
{
  //Implement the Agent code //here!
  // in our demo we assume two
  //methods are defned for the //agent
  // they are compare() and  //print()
}
```

To minimize the size of the transferred code, it is preferred to write the agent class in separated assembly. The assembly file is the unit of transferable code for agents.

After writing mobile agent class in a separate assembly file, the agent needs to be initiated and launched to the target machines in order to start execution of its tasks. So, in the application assembly, the developer initiate a new itinerary object from itinerary class and uses its *SetSiteTask* method to plan out the agent journey. After that a new instance from the user mobile agent class is required to be created. In this example, the developer can set AgentName property optionally, whereas the AgentAssembly property must be set mandatory, in addition to AgentItinerary which in this example takes the previously defined itinerary object. Eventually, the last step required to launch the mobile agent is to invoke the GO method with this agent, which in turn enforces the migration of agent. The following piece of code show how the developer can launch the mobile agent.

```
private void SendAgent_Click(object
sender, EventArgs e)
{
      Itinerary itnr = new
Itinerary();
      itnr.SetSiteTask("LocalSite",
"");
      itnr.SetSiteTask("Site#1",
"Compare");
      itnr.SetSiteTask("Site#2",
"Compare");
      itnr.SetSiteTask("Site#3",
"Compare");
      itnr.SetSiteTask("LocalSite",
"Print");

      CDAgent _agent = new CDAgent();
      _agent.AgentName = "aaa";
      _agent.AgentAssembly = "/*The
agent assembly path*/";
      _agent.AgentItinerary = itnr;
      // set the agent' local data
here
      _agent.GO();
}
```

After launching the mobile agent, it starts its tasks as planed out in the itinerary object. At the end of the agent life-cycle, the agent comes back to the first site and provides the gathered data to the owner.

## 7. Conclusions

This paper presented and described the basic services (core) of an agent platform. These services are migration and management services.

The strength of this framework stems from the fact that agents can be written with a variety of computer languages, while all other frameworks are designed to support mobile agents based only on Java except the MAPNET framework. The CLR framework supports all agent-based distributed applications that are written in programming languages such as C#, C++, VB or Java.

The framework designed is based on modified algorithms that ensure better performance. This framework is implemented using C# and can be incorporated with any application as a library as described in section 6.

The services presented here are designed and implemented in partial fulfillment of a complete mobile agent framework which includes other services such as directory, security and consistency besides many other services (not core).

## References

[1] D.G.A. Mobach, B.J. Overeinder, N.J.E. Wijngaards and F.M.T. Brazier, "Managing Agent Life Cycles in Open Distributed Systems", SAC Melbourne, Florida, USA, ACM (2003).

[2] Mohamed Khamis, Ibrahim Al Bedwi, Ihab Sroogy, "Directory and Migration Services for Mobile Agent Framework", JAUES, Vol. 2 (6), January (2008).

[3] Haeryong Park, Haksoo Ju, Kilsoo Chun and Jaeil Lee, "The Algorithm to Enhance the Security of Multi-Agent in Distributed Computing Environment", Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS'06), IEEE (2006).

[4] http://www.trl.ibm.com/aglets/

[5] H. Peine and T. Stolpmann, "The Architecture of the Ara Platform for Mobile Agents", First International Workshop on Mobile Agents (1997).

[6] T. Walsh, N. Paciorek and D. Wong, "Security and Reliability in Concordia", Thirty-First Hawaii International Conference on System Sciences, IEEE Press, (1998).
[7] http://www.tacoma.cs.uit.no/
[8] D. Johansen, R. Renesse and F. Schneider, "An Introduction to the TACOMA Distributed System", Technical Report 95-23, Department of Computer Science, University of Tromso, Norway, (1995).
[9] P. Braun and W. Rossak, Mobile Agents: Basic Concepts, Mobility Models and the Tracy Toolkit, Morgan Kaufman, 2005.
[10] D. Milojicic, W. LaForge and D. Chauhan, "Mobile Objects and Agents (MOA)", Proceedings of USENIX Conference on Object Oriented Technologies and Systems (1998).
[11] J. Baumann, F. Hohl, M. Straßer and K. Rothermel, "Mole - Concepts of a Mobile Agent System", University of Stuttgart, Institute for Parallel and Distributed High-Performance Computers (1997).
[12] C. Baumer, M. Breugst and S. Choy, "GRASSHOPPER – A UNIVERSAL AGENT PLATFORM", Magedanz (2000).
[13] L. Silva, P. Simoes, G. Soares, P. Martins, V. Batista, C. Renato, L. Almeida, and N. Stohr, "JAMES: A Platform of Mobile Agents for the Management of Telecommunication Network", 3rd International Workshop on Intelligent Agents for Telecommunication Applications, Stockholm, Sweden (1999).
[14] D. Staneva and D. Dobreva, "MAPNET: A .NET-Based Mobile-Agent Platform", International Conference on Computer Systems and Technologies, ACM Press, (2004).

Received July 30, 2008
Accepted September 7, 2008