# Mobile Agent Synchronizer for a Real-time Architecture

Ali Esserhir and Fabrice Mourlin
Algorithmic, Complexity and Logic Laboratory
Paris-Est University (U-PEC)
Créteil, France
{ali.esserhir, fabrice.mourlin}@u-pec.fr

*Abstract*—**In this paper, we use mobile agents in order to synchronize real-time distributed system. We present a way to exchange mobile agents from an agent server to agent hosts. Imported agents are used to apply actions in their host context. We solve security problems by a negotiation step between the host and the server. Locally, we use a specific thread to execute the imported agent. Its mission uses time vector for event synchronization. When the mission of the mobile agent is ended, it goes back to the server and waits for a future demand. We apply this strategy to implement mobile agents, which import the time of the server, and synchronize the actions between source and target hosts and keeps the real-time properties.**

***Keywords-agents; mobile code; real-time system; task synchronization***

## I. INTRODUCTION

Mobile agents provide a new approach of distributed systems. They give a solution for moving part of code from an agent host to another one where the data are installed. If developers have the habit to transfer data from a software layer to another one, this technique has limits. When the size of data exceeds a limit, the exchange cannot be possible. Again, security reasons can filter data transfers. It depends on the semantics of the data: results of a company, hybrid mesh for numerical simulation, etc.

Many works are about the subject of mobile agents. Java Agent DEvelopment Framework (JADE) toolkit [1] is one of the most famous solutions for the development of software agents. Jade is a middleware that facilitates the development of Agent Management System (AMS). The need of mobility becomes greater in network management and other platforms exist, such as Tracy toolkit [2] or Mobile Computation Architecture (MCA) platform [3]. They allow moving agents from one node to another node of a network. MCA is designed for building a fault tolerance layer into a numerical application. When a distributed application is deployed over a set of processors, events can occur during the runtime. This set of computing resources is not fixed, the new processors can be free and few nodes can fail. Also, the execution should continue despite these events. MCA allows moving a part of the execution if the new computing resource appears or disappears. The mobile agent can be considered as a vehicle which leads a piece of code towards a computing resource. This approach can be done because the parts of the application are loosely coupled. The exchanges of data are few and predictive.

When the exchanges are based on time synchronization, this means that a time control algorithm is applied to a business application, such as a distributed numerical simulation. The main constraint of a real-time application is to respond to an event at a scheduled time. We did not find an existing work which groups mobile agent and real-time. For example, this requirement occurs when specific events happen, such: hardware interrupts, local clock, communication, etc. The traditional operating systems cannot guarantee that the latency is below a certain threshold. In fact, during some very resource-intensive operations (such as inputs / outputs), these systems can be temporarily blocked. This leads to the use of real-time operating systems, such as QNX [4], which ensures latency known by its micro-kernel structure. Deployment of applications on to such operating system allows users to exploit real-time features.

Our work is about distributed architecture and the use of mobile agents for synchronizing tasks in real-time context. Section 2 is about the requirements of our work. Section 3, we present our approach of mobile agent under the development constraints. In Section 4, we describe our algorithm and our implementation. Section 5 is about our measures and results. Finally, we review the main features of our contribution and the future directions.

## II. REQUIREMENTS AND CONSTRAINTS

### A. *Operating system and libraries*

A real-time system is a combination software / hardware where the software allows adequate management of material resources to complete certain tasks or functions in very specific time limits. Real-time applications are often embedded applications. Constraints of standard software, add the notions of response time, latency, clock, timing tasks,f etc. We use an operating system with a real-time kernel. A real-time kernel is the minimal implementation to make real-time scheduler, task management, and inter-task communication.

There are differences between a true and a real-time system. The first kind of system has completed a real-time kernel with modules and libraries to facilitate the design of real-time application: memory management, management of input/output management, timer management, network access, file management. A real-time operating system is the special case where the host system communicates with the target system. So, here we have chosen a development environment natively based on RT-Linux [5].

### B. *Programming languages*

Our experience about mobile agent development has shown that object interpreted languages are useful for exchanging between two parts of code: a requestor and a server. Many programming languages exist, which are interpreted with object paradigm. We have already developed projects based on mobile agent architecture. Our approach has been validated in the distinct domain, such: web server monitoring, numerical analysis, business process management, etc. We have chosen Java language because it possesses useful features such: serialization, garbage collection, class loading, network, Just-in-time compilation, and thread scheduling. But, traditional implementations are incapable of running applications with real-time behavior.

Fortunately, real-time extensions to Java technology, based on the real-time specification for Java (RTSJ) [6], enable JVMs with real-time features. The RTSJ provides an API is enabling real-time scheduling, advanced memory management, high-resolution timers, asynchronous event handling, and asynchronous interruption of threads. Of course, standard Java applications can run without modification in a real-time JVM, but some APIs ,such as the threads and timers APIs, APIs are enriched. Also, when developers would like to create a new project with the real-time Java development kit [7], they can set their own configuration.

Because mobile agents move from a Java virtual machine towards another one, network protocols are necessary. Default implementation (provided by Sun, Oracle) of RTSJ does not provide any solution. Research works exist about real-time Remote Method Invocation (RMI) framework, but the libraries are not maintained or are not available. For instance, the RT-RMI framework supports timely invocation of remote objects. The thread classes defined by the RTSJ are used to provide the client and server threading mechanisms in the invocation process [8]. But, this framework is not available for four years

Also, we have selected a Jamaica virtual machine [9] because it provides a complete implementation of the RMI protocol. This software product is frequently updated, its roadmap is public and reference documentation is available. The use of RMI protocol means a rich protocol where object can be serialized on an RMI socket. Also, this feature allows developers to separate the concerns: agent server, agent host and mobile agent.

In the next section, we give details about our mobile agent design and implementation with this restricted development context.

## III. Mobile Agent in an RT Context

The software architecture of a distributed mobile system is based on two main strategies. In the first strategy, agent hosts provide a remote service for reaching them. Next, an agent server configures a mobile agent to visit a set of hosts with these remote stubs. Then, the mobile agent starts its visits by using the remote services. By the end of its mission, it comes back to the server through a call to a remote service on the server.

A second strategy consists in the creation of mobile agents which have a remote call. Then, the agent hosts can send a demand to a specific mobile agent if is needed. Because, we believe that a mobile agent has to be autonomous, we have selected the first approach. Thus, the agent host is passive until the agent decides to reach it. Then, the host will allow the mobile agent to realize its mission in the context of the agent host.

### A. *Necessary conditions*

Whatever the strategy is, it is necessary to have technical skills. The migration of agent can be realized onto the network with the following concerns:

- Common execution in virtual machine on all the nodes,
- Process persistence: saving and spawning
- Communication mechanism between agent hosts
- Security to protect mobile agents and agent hosts

In a heterogeneous environment, many different system architectures can be connected. An interpreted language that is capable of executing machine code solves the problem of a common execution language. A Java virtual machine answers to that constraint even if the versions are not the same. The use of timers is a first need to settle task management. Existing *Timer* class in standard JDK is not powerful enough for clock synchronization. RTSJ provides new timer with a richer API.

For agents to migrate to remote machines, they must be capable of saving their execution state, or spawning a new process whose execution state will be saved. This persistently means the conversion of the object's state into a data form suitable for transmission over a network. Mobile agents should not have to be responsible for achieving this themselves. Again, Java language takes into account the serialization concern and of course the deserialization.

Some communication mechanisms exist to transfer mobile agents across networks. A mobile agent might be transferred using TCP/IP, or by using a higher level of communication such as RMI, Internet Inter-ORB Protocol (IIOP), Simple Mail Transfer Protocol (SMTP) or even HyperText Transfer Protocol (HTTP). RMI is a sub part of Java language since its first version. This is also true for the real-time implementation.

Security is critical when mobile agent is transferred across a network. Depending on the mission of the agent, it could write local resources of the agent host. Mobile agents themselves need protection against hostile hosts. Java runtime forces the creation of a security manager which observes and checks all the actions of incoming agents. So, by default, a mobile agent has no permission at all and when an agent leaves for a new host, extreme care must be taken to prevent unauthorized modification or analysis of the agent. This can be done during the code loading of mobile agents.

Under this set of constraints, we propose into the next section an object approach of RT mobile code. We stress the importance of the state of the agent which ensures that a mission starts with a first agent host and continues on the next one and so on.

## *B. Object description*

### *1) Architecture descriptions*.

Our approach is based on the use of the object model. First, we provide a high level deployment diagram, where main nodes are presented. The Server node plays the role of the agent factory. It supports component called Agent Server in the following section. The Host nodes play the role of agent client, which waits for the visit of mobile agents. Figure 1 gives a picture where the links support RMI protocol and nodes support Jamaica virtual machine.

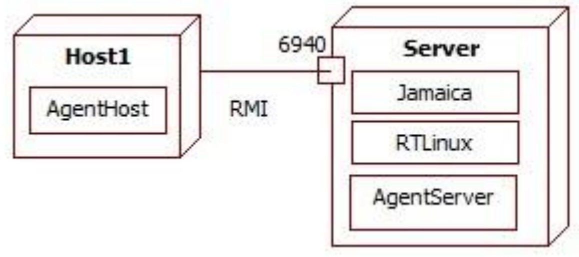


Figure 1. deployment diagram for a minimal network

This hardware architecture supports several components. For our design, all Host nodes have the same deployed components called *AgentHost*. This choice can evolve whether an agent host filters mobile agents through its type. The server node receives an agent server at deployment step. It will create mobile agents as demand. It configures them by injection of remote accesses and mission description. Then, mobile agents start and are autonomous during their whole activity. Figure 2 shows the dependencies of a mobile agent component. It depends on the provided interfaces of all the agent host components and also the remote interface of the agent server for the end of its mission.

The mobile agent provides two local interfaces. At first one, called *IAgent* is used by the agent server to configure locally the agent. Another one is called *IMobileAgent*, which is called by the agent host to start or to suspend the activity of the mobile agent locally to the host. This software architecture leads to the creation of three artifacts which will be assigned to the nodes of the Figure 1. Globally, all nodes which wish mobile agents needs to support an *AgentHost* component.

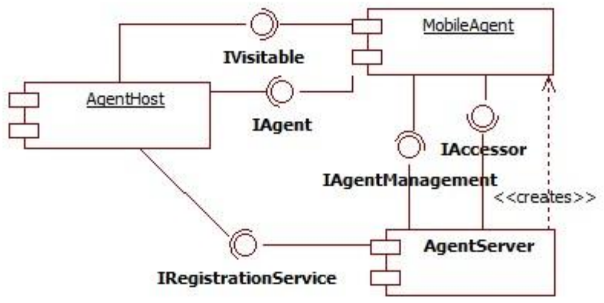


Figure 2. component diagram with provided interfaces

### *2) Class structure*.

#### *a) Mobile agent component structure*.

Minimal structure of *the Mobile Agent component* contains a class which implements two main interfaces (*IMobileAgent* and *IAgent*) (Figure 3). This class has a behavior called *configure* method: it allows an agent server to provide remote stubs about the hosts to visit, a set of properties and a task. The task describes the activity of a mobile agent. This can be a data collection. For instance, at the end of a distributed application, a mobile collector can gather the partial results on each node. Each new task type is defined by a subset of classes. The main one implements *Task* interface. All tasks are placed on the agent server in the workspace directory. The properties are useful to provide a specific working context. So, the description of the features of agent *host1* is set into a file called *host1.properties* on the agent server. In the configuration step, the URL of this input stream is provided by the server. Next, the mobile agent extracts the data set into a *Properties* object. These data will be used during the activity of the mobile agent.

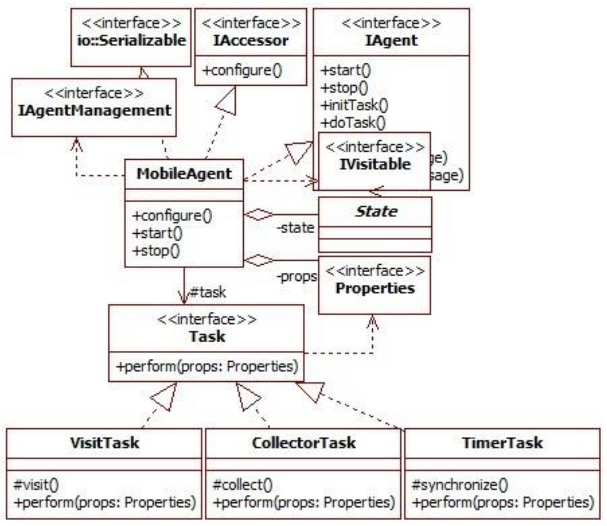


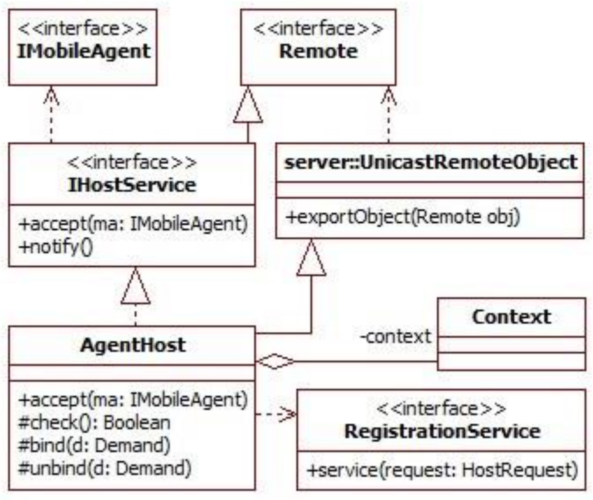
Figure 3. minimal class diagram of *MobileAgent* component

A mobile agent is composed with a *State*instance, which manages local data during its activity. This state is a data structure which is encoded during a transfer from on the agent host to another one or the agent server.

#### *b) Agent host component structure.*

The *AgentHost* component plays the role of the requestor. Its interaction sequence of the demand is as follows. First, the agent hosts are registered in the locate registry of the agent server. Then, it configures one or more mobile agent with the remote stubs. Next, the mobile agents use the remote access to move onto the agent hosts through a serialization step. It invokes a remote call through RMI protocol. This is why the *IHostService* interface is a generalization of *Remote* interface. In Figure 4, the implementation, called *AgentHost* is an extension of a technical RMI class. It generates automatically a remote stub for the publication into the server registry. So, it exposes a remote interface called *IHostService*, for mobile agents.

When the mobile agent is received by the agent host, it performs a check of the byte code and permissions useful to execute its activity. Then, it loads the code of the mobile agent and assigns needed permissions. It launches its mission. By the end of its job, the host invokes the *stop method* which triggers the continuation of the mission onto another agent host.



<<interface>> IMobileAgent
<<interface>> Remote
<<interface>> IHostService
+accept(ma: IMobileAgent)
+notify()
server::UnicastRemoteObject
+exportObject(Remote obj)
AgentHost
+accept(ma: IMobileAgent)
#check(): Boolean
#bind(d: Demand)
#unbind(d: Demand)
-context
Context
<<interface>> RegistrationService
+service(request: HostRequest)

Figure 4. class diagram of *AgentHost* component

The runtime context of a mobile agent is restricted by the agent host itself. It does not run into another virtual machine. So the security is managed by an access controller, which is created by the host. This one is also responsible of the permissions which are defined locally to the host with a code base and a signature. They are not downloaded from the agent server. Each agent host has a context where the activities of the mobile agents are interpreted. So, after the execution of a mobile agent, the agent host suppresses its demand from the registry of the agent server.

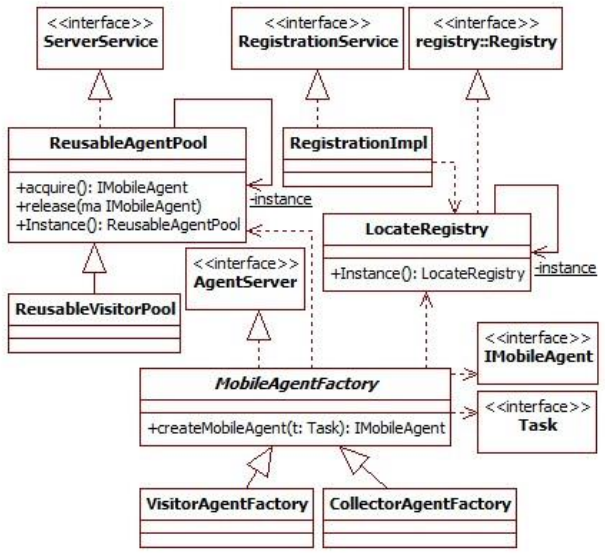
*c) Agent server component structure.*

The agent server is a clever agent factory. It exposes two remote interfaces, one for the agent hosts and another for the mobile agents. The first interface (*Registration Service*) is used to collect or remove the demands of agent hosts. They are stored in a local registry.

The second interface (*Server Service*) is used by mobile agents in two situations; (i) on the one hand, when their mission is

completed and they return to their starting point, and on the other hand, when a mobile agent needs additional resources, it can ask the server that created it.

The interaction sequence for a mobile agent creation is as follows. First, a mobile agent is instantiated by the use of an extension of *MobileAgentFactory* class. A task is assigned to the new mobile agent. Next, it is configured with remote stubs of agent hosts. Then, its behavior is launched. So, it can begin its mission by moving onto the first agent host.



<<interface>> ServerService
<<interface>> RegistrationService
<<interface>> registry::Registry
ReusableAgentPool
+acquire(): IMobileAgent
+release(ma IMobileAgent)
+Instance(): ReusableAgentPool
-instance
RegistrationImpl
LocateRegistry
+Instance(): LocateRegistry
-instance
<<interface>> AgentServer
ReusableVisitorPool
<<interface>> IMobileAgent
MobileAgentFactory
+createMobileAgent(t: Task): IMobileAgent
<<interface>> Task
VisitorAgentFactory
CollectorAgentFactory

Figure 5. class diagram of *AgentServer* component

The agent server tracks the activities of mobile agents through incoming messages. By the end of the mission, the mobile agent uses the interface called *Service Server*. So, it returns to the pool of mobile agents pending mission. In a next request of an agent host, this mobile agent can be used to receive a new task and receive new information and thus make a new mission.

Because agents returning to the pool are already preconfigured with a business task, this can be useful to select an agent in the pool among all agents. Thus, the pool is indexed to the task of the agent. This is illustrated by the class, called *ReusableVisitorPool* that collects *Visitor* agents to be reused.

The creation and management of objects are a tricky business. It is often useful to create agents only when necessary. There are situations where it is useful to reuse objects (reusable agents), in other situations the duration of activity of an object is important to keep its state. This is the case of the services offered by the agent server where the concept of threads is crucial. We develop this notion in the next section.

## C. Real-time description

A real-time system must meet certain performance constraints expressed in the execution time of tasks, even in the worst case. A real-time system is not as fast overall system, but a system fast enough or not too slow.

### 1) Management of server activity

*a) Thread strategy*

The RTSJ specification defines two categories of real-time threads. The difference between these two categories is the use of memory and whether or not a thread can be interrupted by the garbage collector. In the first category, defined by the *RealTimeThread* class, threads can use the heap allocation and regions to create objects. Therefore, these threads can be interrupted, but the garbage collector is not responsible for the automatic is responsible for the automatic management of allocated objects. The second category is composed of real-time threads where the worst execution time must be deterministic. These threads are defined by the class *NoHeapRealTimeThread* have a higher priority than the garbage collector to prevent any interference with its performance. Therefore, they are not allowed to access the heap allocations. They are forced to use regions for the allocation of objects.

The agent server component is structured into three parts which are independent. So, we have developed three classes which extend *RealTimeThread* because they can prevent the garbage collector from running, but would not preempt the collector if it is already running. These are the reusable agent pool, the registration implementation and the mobile agent factory. They run at a priority higher than the garbage collector. The three classes implement indirectly *Schedulable* interface. This means that they use the concepts of cost or duration, maturity period (for periodic tasks) overrun handler, if exceeded runtime and miss handler if the deadline is exceeded. The default scheduler is a priority scheduler and

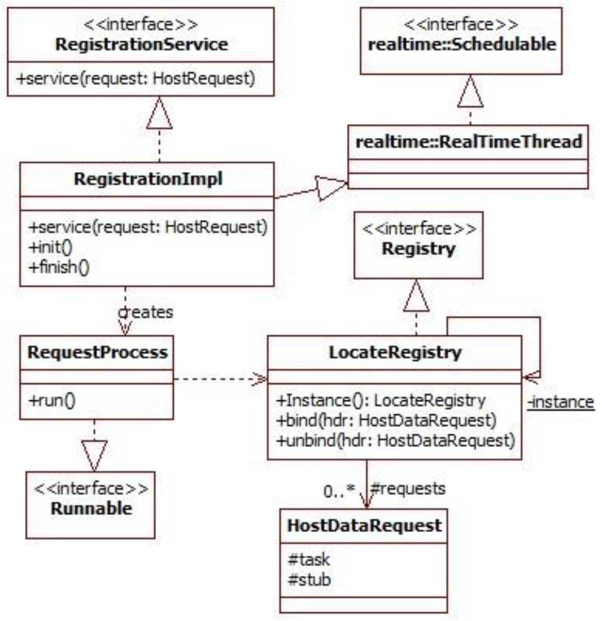


Figure 6. advanced class diagram of registration service

each real-time thread has its own priority

We have decided that requests from the agent hosts have to be registered first, next the return of mobile agents, and finally, the creation of new mobile agents. Also, we have assigned decreasing priorities. We use an aperiodic parameter to specify the cost and the deadline of threads. This set also a deadline missed handler to the schedulable servers. Then, these three threads are added to the scheduler to determine whether the system is schedulable. The scheduler is responsible of the whole runtime as a conductor in front of an orchestra. By the end of the registration thread, its callback (*finish* method) is invoked and this triggers the end of the whole agent server (figure 6). It means that the reusable pool of mobile agents is cleaned and the agent factory is stopped.

#### *b) Memory management strategy*

Each of the three threads (registration, agent pool, and agent factory) has its own memory area. A standard Java distributed application uses a distributed memory area that is hidden from the programmer and managed by the garbage collectors. In our application using real-time extensions, object creation is performed in the exact same manner, but now we can control which memory area the new operator uses to construct objects, and in turn, how that memory is reclaimed. We use subclasses of *ScopedMemory* for the registration requests. They indicate how the memory area will behave during allocations.

We use the *LTMemory* class which ensures that the allocation time will be linear. Moreover, the initial memory size, specified when the memory area is constructed, is contiguous. As a result, *LTMemory* areas pre allocates the minimum size specified when constructed. This memory area is not subject to garbage collection.

As an example, consider a request from an agent host about a future data collection. This object contains a data record with: the type of the wished task, reference of the remote agent host and a validity period for this demand. The registration is received by the registration implementation server. As a consequence, a *LTMemory* area is defined for this data record. In traditional object programming, this data record is available for garbage collection when the request is satisfied. This could perturb the behavior of the whole server component. The registration implementation is written so that it can reclaim the space for the temporary request without garbage collection. First, a scope memory area is needed; in our case *LTMemory* is used. Next, the code that is going to create and use the temporary request is constructed into a class that implements the *Runnable* interface; then, that class is passed as a parameter to *enter* method. To avoid creating a public class, a named inner class is used (called *Request Process*).

Our approach reclaims the temporary memory every time the *enter* method of scope memory is finished. However, as mentioned above, the creation of a named inner class is better than the creation of an anonymous class because it will still use memory in the surrounding memory area. With our solution, there is just the initial cost of creating the *HostDataRequest* object. We can remark that the only way to pass parameters to the run method (*Request Process* class) is by setting attributes of the object from the registration implementation. This main real-time thread dispatches the request process with the creation of an instance of *ProcessRequest* class which is runnable. So, it is available to treat next event.

#### *c) Event management strategy*

We use an approach to keep plain RMI and follow the event notification model similar to AWT/Swing. In those models, an object implements a simple listener interface and is then added to an appropriate event notification list. When that event occurs, every object in the list becomes notified of the event. The object takes whatever action necessary in response to the event. Such a model obviously is a better solution than polling, and solves the problems that polling has. There is no unnecessary network traffic and clients are notified instantly when a change in the data has occurred.

Applying this model to RMI is not trivial. Consider the RMI-based agent server as an example, where agent hosts registered with this agent server (interface *Registration Service*). After creating and configuring a mobile agent, the RMI agent server fires an event (called *AgentCreationEvent*) to inform the agent hosts of the arrival of a new mobile agent. To successfully do this, first, we enrich the remote interface named *IHostService* (Figure 3) with a method called *notify()*. The remote interface (called *RegistrationService*) with the *service()* method is analogous to the *setActionListener()* methods in Swing only instead of implementing the *ActionListener* interface, an object that implements the *IHostService* interface is needed. Just as the *ActionListener* interface serves to link an event with the application code that processes the event.

The *HostRequest* parameter serializes to a byte stream and is transmitted over the network to the server. The agent server needs to know how to deserialize the stream and reconstitute the object. Second, the *RequestProcess* instance registers the object by calling the *bind()* method of the *LocateRegistry*. Then the request is saved as a *HostDataRequest* instance.

When the same request comes from several agent hosts, only one mobile agent is created and configured with the set of remote stubs. It will execute its mission as far as the server has received corresponding request. All the hosts are notified by the server, before receiving the mobile agent. Now two cases can be considered: an aperiodic task as mentioned in the visit task (Figure 3), a periodic task as a data collection.

### 2) *Configuration of periodic task*

When mobile agents are prepared by a task and properties, some of them are executed periodically at different rates for most agents. Some of the agents may be scheduled for simultaneous execution though. Also, a mobile agent may need to start executing while another is currently executing. Many constraints are taken into account when scheduling, real-time mobile agents.

The first constraint is the amount of available resources; whether there are mobile agents into an agent host using existing resources. The second constraint is task precedence; to avoid ambiguous state. In the present work, all the tasks are considered independent but a task graph could be useful to organize structured activities. Certain tasks may need to be executed before others. The third constraint is timing; each task has its own deadline, some tasks execute longer than others, some may execute on a steady period, and others may vary between release times. Each constraint contains many of its own factors that need to be considered further when scheduling tasks.

The availability of system resources, such as the processor, is important to a scheduling multi thread application. But this goes beyond just the processor; other resources, such as shared objects that require synchronization across mobile agents, are to be considered. On agent host, we need to lock shared resources to avoid errors due to concurrent access. This limits a resource to being updated atomically, by one mobile agent at a time. Since resource locking synchronizes access to a shared resource, one mobile agent may become blocked when attempting to access a resourced that is currently locked by another mobile agent or host itself. In distributed real-time application, we ensure that high-priority (*RegistrationService*), real-time threads continue to make progress towards completion. Resource locking is commonly a problem since priority inversion can cause threads to execute out of order. When a request is received from a host, this host request is treated with care of resource lock.

A mobile agent is also defined by timed properties. The three main features are deadline, period and execution time. The deadline is requested by an agent host; its value is critical to determining if it can be scheduled feasibly. It is expressed with a timer local to the host. The host can request real-time tasks at regular time intervals. The mobile agent is configured with a value (periodic or aperiodic) along with the timing characteristics that might apply. The execution time of the request is the task cost used to know the interval execution times for a task. All these features are kept for the configuration step. The periodic feature is particularly useful by the end of the mission of a mobile agent. It means that the mobile agent has to start again the tour and performs its task. The controls are done to know whether the duration of the tour is permitted by the agent hosts.

### 3) *Exchange management*

All RMI services are sources of unbounded interference and their interaction with the real-time Java virtual machine and runtime libraries is not clearly defined. Also, delays occur without reason. Jamaica-RMI implementation avoids some of these constraints, but RMI communication is always a source of unpredicted measure. The non-deterministic behavior has induced many researchers to avoid or forbid their use in specific environments like high-integrity applications. Other researchers have developed their own implementation. First, we can cite the work of A. Ahern and N. Yoshida [10]. They present an object-oriented, Java-like core language with primitives for distributed programming and explicit code mobility. But, this is a formulation to prove the correctness of distributed programs; it does not provide an extension a RTSJ specification.

D. Holmes explained in [11] the contributions of the RTSJ specification, particularly the sequencing, memory management and asynchronous events. More specifically, it describes the importance of managing priorities and resources between threads introduced in [12]. In [13], S. Rho explains the design a real-time Java remote method invocation. In their work, remote method invocations are modeled as sporadic events. So, the overhead of their employment seems to be negligible. J. S. Iyilade provides a comparative study of the performance of the modern system, multi agents with the same criteria to try to limit the cost of RMI [14].

P. Basanta-Val describes the spectrum of approaches for distributed real-time [15]. Many projects around RMI and real-time are born, but often without reaching frameworks used by people other than the authors. The approach seems to Jamaica by far the most successful [16].

We used the results of S. Rho for the management of data exchange, especially for event notifications. Remote method invocations are modeled sporadic events and treated by a specific configured server. The latency of real-time RMI becomes stable without interference.

## IV. Synchronization Algorithm

### A. Need of the control algorithm

In a distributed application, a question remains often: how to time the execution of a Java program. Several answers are possible, depending on the used platform. There are some issues where *System.nanoTime()* cannot be reliably used on multi-core CPU's to record elapsed time. Each core maintains its own Time Stamp Counter (TSC): this counter is used to obtain the nano time (it is really the number of ticks since the CPU booted).

Hence, unless the OS does some TSC time warping to keep the cores in sync, then if a thread gets scheduled on one core when the initial time reading is taken, then switched to a different core, the relative time can sporadically appear to jump backwards and forwards. Because, a simple use of timers is not allowed because of the distribution of code over the network, we developed an approach based on timer vector. Our approach is implemented through the development of a new task called *TimerTask* as an extension of the Task (Figure 3).

### B. Distributed task synchronizer

In this section, we present our implementation of task synchronizer based on clock vector. This approach is not new, for conflict resolution. Some NoSQL databases use this strategy too. L. Lamport defined the main concerns of this approach [17]. C.J. Fridge provided an application in the domain of the message passing system [18]. F. Mattern developed the clock vector idea for distributed applications. He proved correctness properties about partial ordering.

#### 1) Initial step

In our context, agent hosts need to synchronize local activities. So, they need to receive mobile agents to do that. Mobile agents are initialized with a *TimerTask* instance and input data about one agent host. This *TimerTask* instance, will be the behavior of mobile agent after its migration onto the agent host. First mobile agents use accept service of their agent host. This host controls as mentioned previously. Next, it starts its mobile agent which starts its task. Each agent host has its own identifier. Each mobile agent has its local timer to manage.

#### 2) Core step

The objective of the deployment is to apply a scheduling to the distributed host. Our scenario is the simplest: it is a step by step execution. Each time an agent host fires a local event, it calls the increment method of the mobile agent. Next, its own logical clock is incremented in the vector by one. Because, the role of the mobile agent is to synchronize the set of hosts, it needs to send its data to the other mobile agents.

Before sending the messages, it increments its own logical clock in the vector by one and then sends its entire vector. Each time a mobile agent receives a message, it increments its own logical clock in its vector by one and updates each element in its vector by computing the maximum of the value in its own vector clock and the value in the vector in the received message (Figure 6, arrows 11 and 12). Then, when the agent host (called *agentHost2*) fires a local event, its mobile agent (called *maB*) uses the updated vector to add a tick to its clock (Figure 6, arrows 15 and 16). As before, this time data is sent to all active mobile agents (only *maA* on Figure 6)

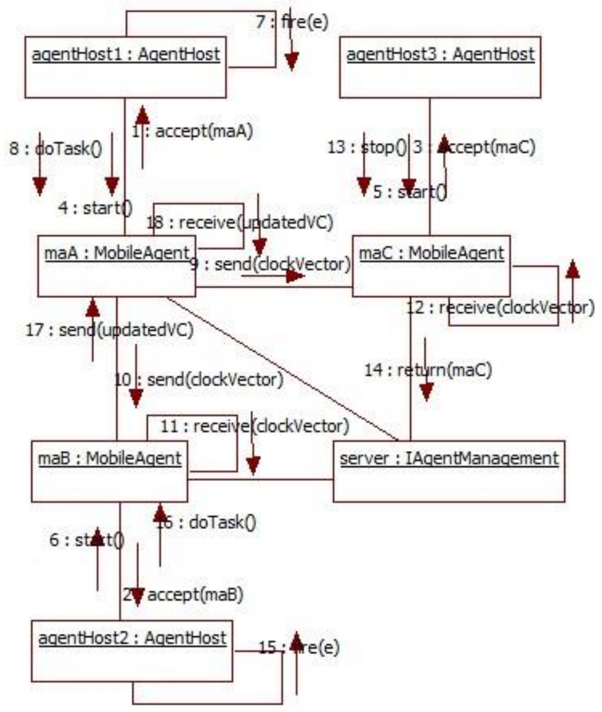


Figure 6. collaboration diagram of Task Synchronization

#### 3) End step

The pilots of the distributed synchronization are the agent hosts. When a host decides to stop its partnership with the group, it just invokes the *stop* method of the mobile agent. Then, the mobile agent informs all the other that it does not accept future clock vectors. Next, it uses the *ServerService* interface of the agent server to go back to the server.

Figure 6 shows an agent host (called *agentHost3*) which stops the behavior of mobile agent called *maC*. So, it won't receive the next update from the other mobile agents. All steps of a task are released into the class called *TimerTask, which* implements the interface Task. This one requires three methods called: *init*, *doWork*, *end*. All the methods are not visible in Figure 6 because of visibility. But, this design corresponds to a Template design pattern which is wrapped into the class *MobileAgent*. In the next section, we present the results of this scenario and how it is possible to realize more complex interaction sequence.

## V. Observations and Results

This experiment mixes real-time constraints and test construction. Because a distributed system is a set of local properties, it is not easy to discover global properties. Any observation involves perturbation on the whole program.

### A. Information description

The great thing is that this time control extends to any number of agent hosts. It's just easier to understand in a context with two agent hosts (Figure 1), but the same mechanism works for any number of vector dimensions. Missing agent host is assumed to have version 0. The clock vectors can be used to describe temporal relations between events in a distributed system. A vector clock can be considered as a list of (place, transition) pairs, in which each place occurs at most once.

The clock vectors have a partial ordering, which captures the idea of before/after a given point in clock vector. It's partially because there exist pairs of clock vectors so that one is neither before nor after the other; they are "concurrent".

But this partial order stresses only on test path. The *TimerTask* instances can be created to stress other test paths. The initial properties of a mobile agent have consequences onto the interaction with its agent host. So, test coverage can be designed by setting precise properties for mobile agents before migration. After the execution, the observation of the state of each host allows the developer to validate or not the test. For instance, a given execution can achieve in a blocking state because a resource is not available. Such timer task is a helper class to detect anomalies into a distributed system.

### B. More complex interaction development

This paper presented a simple scenario of test construction, but the more specific timer task is defined. This is particularly crucial when data injections are useful for the agent hosts. This can be occurred when the data environment of the host is set by the mobile agent.

In the previous example, the task is simply an automaton with three states called *init*, *doWork* and *end*. New kinds of task classes define such infinite loop behavior or with a specific skeleton based on a given interaction with en agent host. For instance, a message format can be forced or a volume of data.

## VI. Conclusion

In this paper, we designed and implemented a real-time distributed application based on mobile agents. Java Remote Method Invocation is a valid transport protocol for mobile agent. By using a server centric approach with developer task classes, we achieved efficient and effective real-time test construction based on event orders. We believe that this work is an important and interesting step toward distributed real-time systems. We defined open frameworks, which can be completed by new contributors, especially who have experience with real-time concepts. The clock vectors are results which can be collected through the use of other mobile agents configured with *CollectorTask* class (Figure 3).

## References

[1] N. Spanoudakis and P. Moraitis: "Modular JADE Agents Design and Implementation using ASEME", IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'10), Toronto, Canada, 2010.

[2] P. Braun and W. Rossak. “From Client-Server to Mobile Agents. Mobile Agents” Basic Concepts, Mobility Models, and the Tracy Toolkit. Heidelberg: Morgan Kaufmann Publishers, 2005, Germany.

[3] C. Dumont and F. Mourlin. Space Based Architecture for numerical solving. In IEEE, editor, International Conferences on Computational Intelligence for Modelling, Control and Automation; Intelligent Agents, Web Technologies and Internet Commerce; and Innovation in Software Engineering, IEEE Digital Library, Vienna Austria, pages 309-314, 2008.

[4] F. Kolnick, The QNX 4 Real-time Operating System . Ottawa, Ont., Canada: Basis Computer Systems, 2000.

[5] L. M. Voy, “Scaling Linux with (partially) cc clusters,” July 2002, http://ww.bitmover.com/cc-pitch/, Retrieved: January 2014.

[6] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. "The Real-Time Specification for Java". Addison-Wesley, 2000.

[7] P. C. Dibble. Real-time Java Platform Programming. Prentice-Hall, 2002.

[8] A. Borg and A. J. Wellings: A Real-Time RMI Framework for the RTSJ. ECRTS 2003: 238-246.

[9] Siebert, "The Jamaica VM," available on http://www.aicas.com. 2004, Retrieved: January 2014.

[10] A. Ahern and N. Yoshida: Formalizing Java RMI with explicit code mobility. OOPSLA 2005: 403-422

[11] D. Holmes, "Introduction to the Real-Time Specification for Java", Brisbane, Purdue University, Australia, 2002.

[12] S. Rho, B. Choi, and R. Bettati: Design Real-Time Java Remote Method Invocation: A Server-Centric Approach. IASTED PDCS 2005: pp. 269-276

[13] J. S. Iyilade, G. A. Aderounmu, and M. O. Adigun: Performance Comparison of Multi-Agent Grid Job Scheduler Organizations. IASTED PDCS 2005: pp. 1-6

[14] Sun Microsystems, "Java Remote Method Invocation," online at http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf. 2004, Retrieved: January 2014.

[15] P. Basanta-Val, M. García-Valls, and I. Estévez-Ayres: An architecture for distributed real-time Java based on RMI and RTSJ. ETFA 2010: pp. 1-8

[16] L. Lamport and P. M. Melliar-Smith: Byzantine Clock Synchronization. Operating Systems Review 20 (3): 10-16 (1986)

[17] C. J. Fidge. "Timestamps in Message-Passing Systems That Preserve the Partial Ordering". In K. Raymond (Ed.). Proc. of the 11th Australian Computer Science Conference (ACSC'88). pp. 56–66 (February 1988).

[18] F. Mattern, "Virtual Time and Global States of Distributed Systems", in Cosnard, M., Proc. Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France: Elsevier, pp. 215–226, (October 1988)