Native Mobile Agents for Embedded Systems

Mohamed Ali Ibrahim and Philippe Mabilleau
University of Sherbrooke
Canada

1. Introduction

Mobile agent technology can be viewed as an extension, refinement, or replacement of the traditional client-server paradigm (Dilyana & Petya, 2002). Client-server technology relies on remote procedure calls running across a network. Taking advantage of local interactions, the mobile agent can at any time decide to migrate from host to host in a network and to which location. In this way, several benefits can be obtained, such as decreasing network traffic, reducing dependency on network availability, and an increasing flexibility and autonomy.

Since the emergence of the concept of mobile agent, many platforms have been developed to facilitate the programming of mobile agent applications. Noteworthy is that these platforms exclusively use an interpreted language (virtual machine) to support the heterogeneous systems. The first language supporting the paradigm of mobile agents was Telescript (Domel, 1996) followed by many others such Obliq (Cardelli, 1995), Safe-Tcl (Borenstein, 1994), etc. Because it is a widespread virtual machine, Java has become the language of choice for distributed applications programming in diverse environments by allowing independence from networks and operating systems.

Java provides a mechanism for serialization and dynamic class loading which are directly used to implement agent migration in the platform such as JADE (Bellifemine et al., 2007), and Aglets (Lange & Mitsru, 1998). The conceptual model of a Java-based mobile-agent platform is shown in Fig. 1. However, Java and languages using virtual machines are too big in terms of required memory space for many embedded systems. A key feature of embedded systems is that they run on machines with limited resources. This limitation is generally spatial (limited size) and energetic (restricted consumption). In order to solve this problem, we propose a mobile agent platform conceived for homogeneous embedded systems. Embedded systems perform predefined tasks and constraints that have to be respected:

- Embedded systems addressing the strict need to avoid an additional cost;
- Computing power required just to meet the predefined task avoiding an additional cost of the device and an excess consumption of energy;
- Keeping energy consumption as low as possible.

In order to meet the constraints mentioned above, we propose a mobile agent platform for homogeneous embedded systems, called *µC/MAS (Microcontroller/Mobile Agent System)*. In a homogeneous environment, not only the type of the operating system needs to be identical, but also the processor type has to match. To our knowledge, this is the first attempt to engineer a mobile agent platform on microcontrollers with memories (both RAM and ROM) as small as one megabyte.

The applications targeted by this platform pertain to the field of pervasive computing and, more particularly, to the "smart space" paradigm. In this context, mobile agents can move between embedded systems implanted in physical objects of the smart space by taking into account limited resources (memory, power consumption, bandwidth, and so on). Each agent uses a native code and operates on homogeneous platforms. Thus, an agent can move only towards a microcontroller having the same physical architecture and supported by the same operating system.

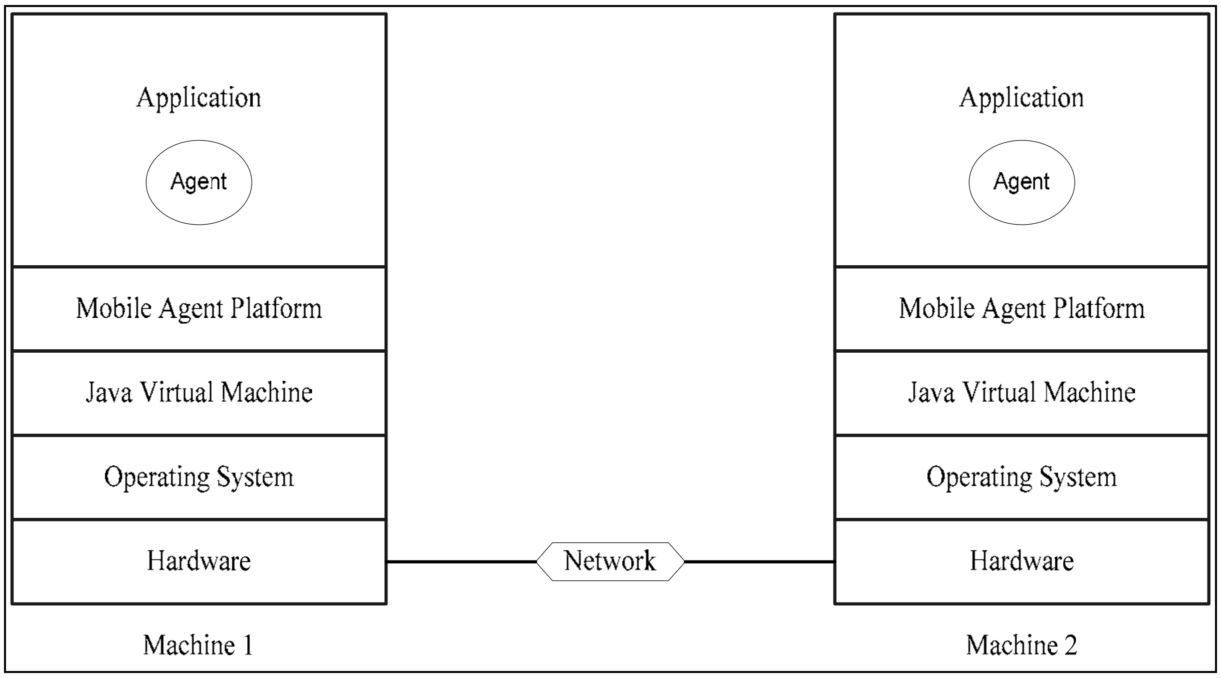


Fig. 1. Java-Based Mobile-Agent Platform.

2. Concept agents

The definition of an agent raises numerous debates in both fundamental and applied research. By simplifying as much as possible, there are on the one hand those who view agents almost like human beings, and on the other hand are those who assimilate agents to simple software. Ferber (Ferber, 1999) defines an agent as a physical or virtual entity which has the following properties:

- is capable of acting in an environment;
- can communicate directly with other agents;
- is driven by a set of tendencies (in the form of individual objective or of a satisfaction/survival function which it tries to optimize);
- possesses resources of its own;
- is capable of perceiving its environment (but to a limited extent);
- has only a partial representation of this environment (and perhaps none at all);

- possesses skills and can offer services;
- may be able to reproduce itself;
- behavior tends towards satisfying its objective, taking account of the resources and skills available to it and depending of its perception, its representation and the communication it receives.

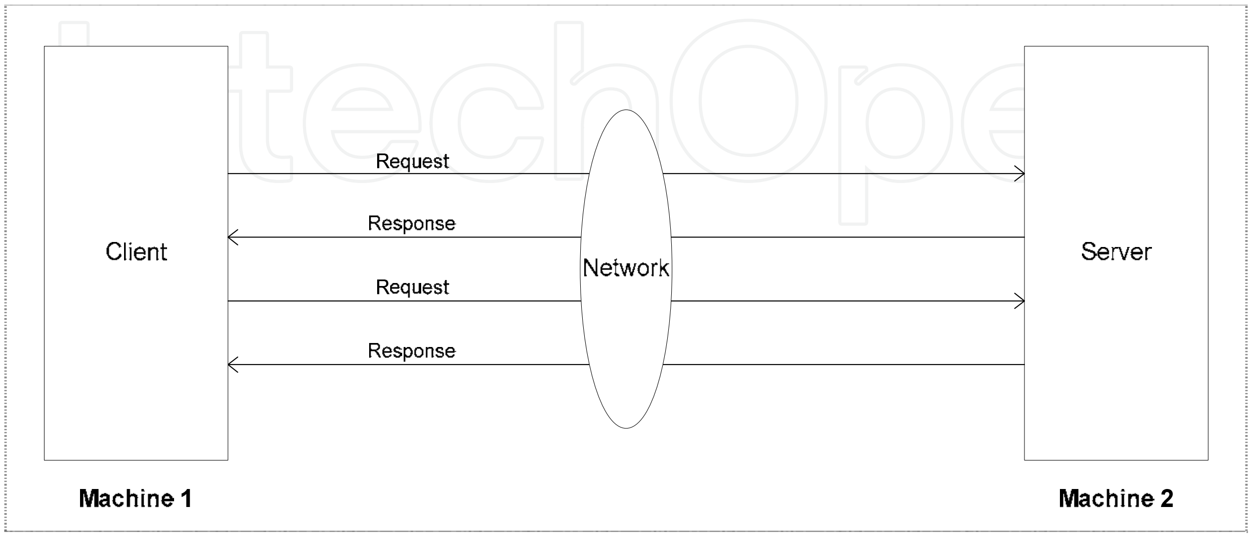


Fig. 2. Client-Server Model.

In contrast, many people consider an agent as an "entity authorized to act on behalf of someone else." According to such a definition an intelligent agent, a police officer, a security guard or a sales agent belong to the same category. As a result, the distinction between an intelligent agent and simple software is very fuzzy. Despite its limitations, this view is a starting point for a definition that is realistic enough without being simplistic. We can thus assert that an intelligent agent is a software entity that has specific attributes and acts in order to perform certain tasks on behalf of another entity (another agent or person). The problem now is to define the attributes appropriate for an agent and, on this point, the debates are ferocious. The main characteristics of an agent are:

- Autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state.
- Interactivity: agents interact with other agents and with humans.
- Responsiveness: agents perceive their environment which can be either the physical world, a user via a GUI or the Internet or even all at once, and respond to changes that occur.
- Intentional behavior: agents do not simply act in response to their environment, they are able to perform goal-directed behavior and take initiatives where appropriate.
- Ability to learn: the agent is able to adapt to the needs of its user by analyzing its past actions.
- Flexibility: the actions of an agent are not entirely predetermined; the agent is indeed able to choose what actions it will choose and what order, depending on the external environment.
- Self-starting: unlike traditional software, an agent can decide, according to the external environment, when to initiate a specific action.

- Mobility: some agents may be stationary as the traditional client-server shown in Fig. 2. Agents reside either on the user's machine or on the server. Other agents may be mobile, as shown in Fig. 3, i.e., they travel on the network. They can move from one machine to another during their execution, carrying with them their execution environment. These agents may meet other agents that can provide certain services, or serve as a meeting point between different agents.

For some researchers, particularly those working in the field of artificial intelligence (AI), the term “agent” has a stronger and more specific meaning. To these researchers, an agent is a computer system that, in addition to the above properties, is conceived as having properties that are most commonly attributed to humans. For example, it is common in artificial intelligence to characterize an agent by purely mental concepts such as knowledge, belief, intent or obligation. Some researchers have gone further and talk about emotional agents. But at what point can one speak of intelligent an agent? Should it have some or all of these attributes? The debate is endless, perhaps insoluble, and certainly without much interest for the end user.

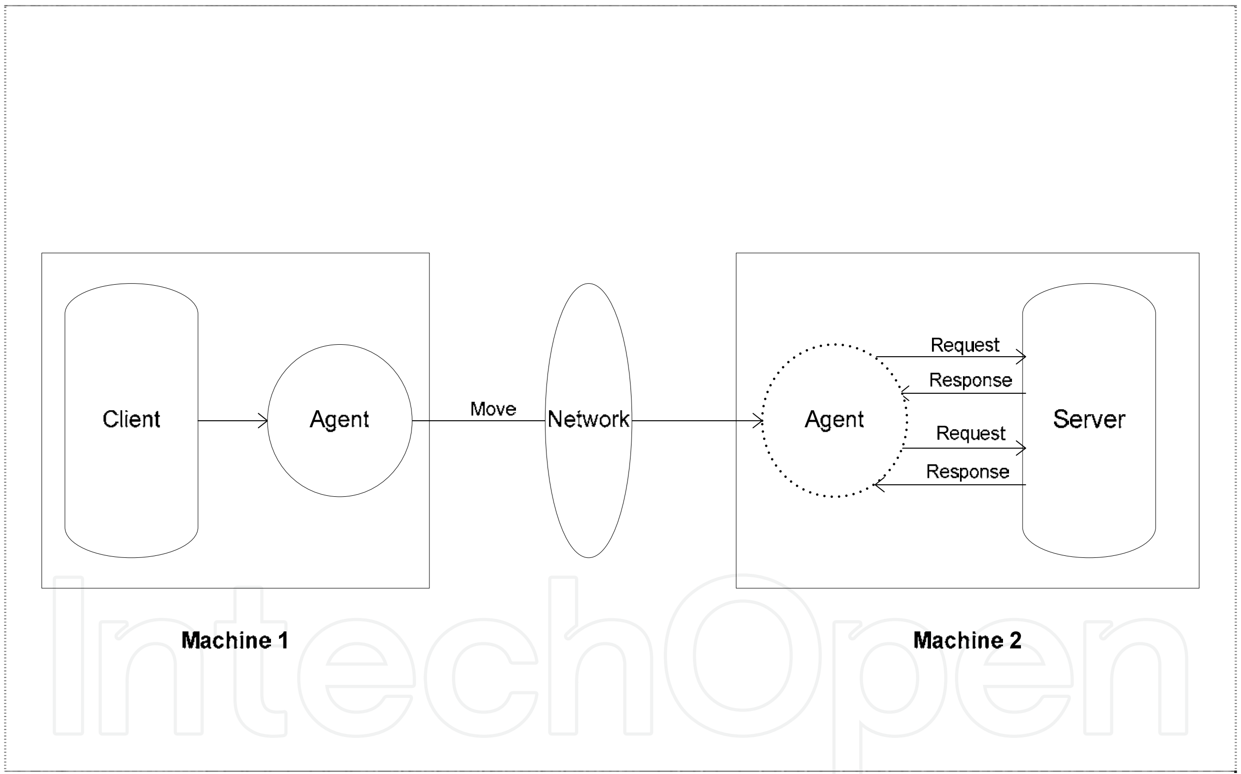


Fig. 3. Mobile Agent Model.

A mobile agent is generally defined as a computer entity capable of reasoning, use the network infrastructure to run in remote locations, search and gather results, cooperate with other agents and return to its original site after completing the assigned task. Its main feature is the ability to travel in an autonomous way between multiple machines such as presented in Fig. 4. In most systems, by virtue of the principle of autonomy, the agent decides when and where to go. This agent can interact with other agents; provide services and use of local resources.

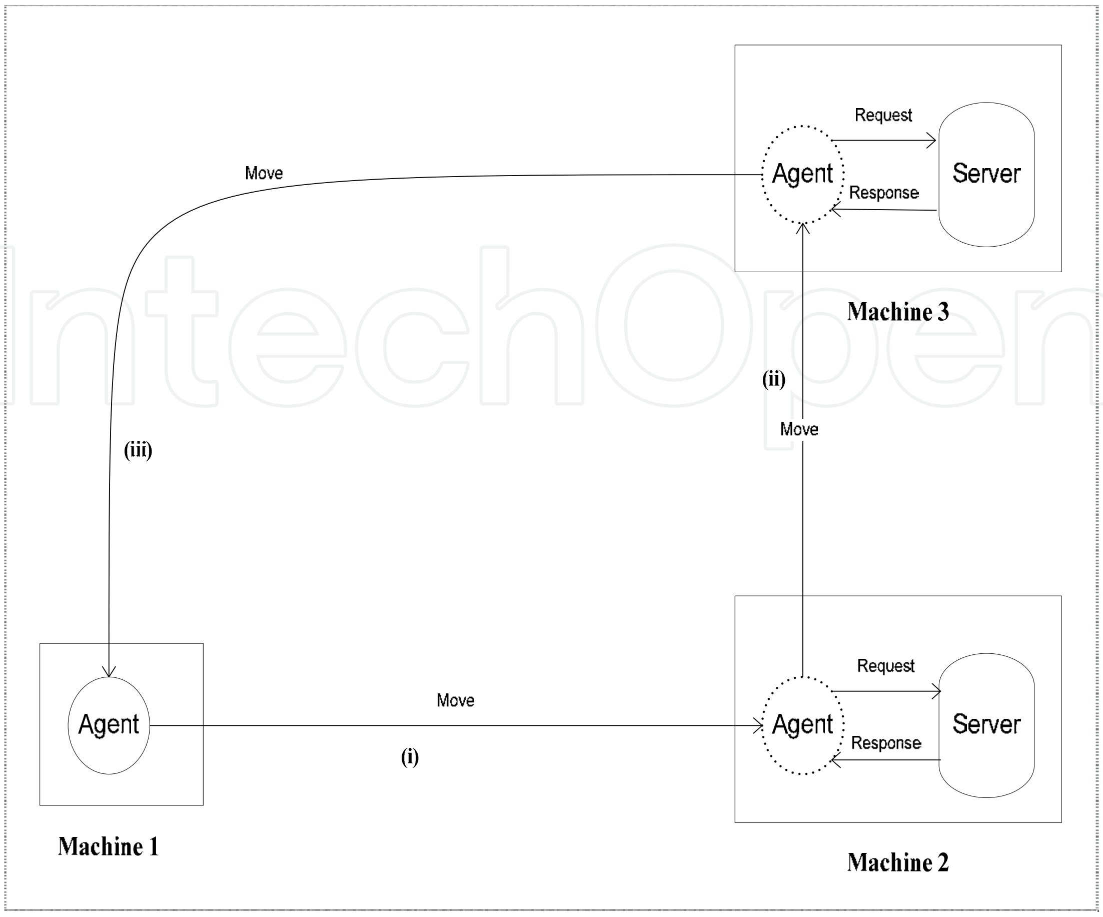


Fig. 4. Model of an Agent visiting two Servers.

2.1 Taxonomy of mobility

There are two degrees of mobility as shown in Fig. 5:

- Weak migration: code + current data.
- Strong migration: code + data + current execution state.

Weak migration is transferring the execution of the application from the source machine to a destination machine, through interruption of the execution of the application on the source site. Then the code and the current data from the application of the source site are transferred to the destination site. Finally, arriving at the destination host, the mobile application resumes execution from the beginning, while having the updated values of its data.

In addition to information taken into account by weak migration (code + current data), strong migration also takes into account the current execution state of the application. Thus, an application with strong mobility that moves during its execution from a source site to a destination site can resume its execution from the point where it left off on the start site. The mobility of an application results in the interruption of the execution of the application on the source site. Then the code, the current data used and the current state of the application running on the source are transferred to the destination site. Finally, arriving at the destination site, the mobile application continues execution where it left off on the start site.

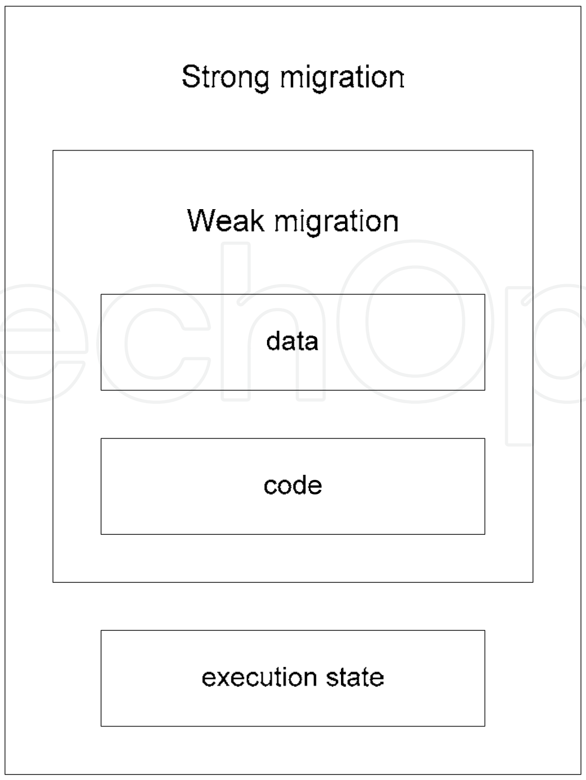


Fig. 5. Degrees of mobility.

3. Architecture of µC/MAS

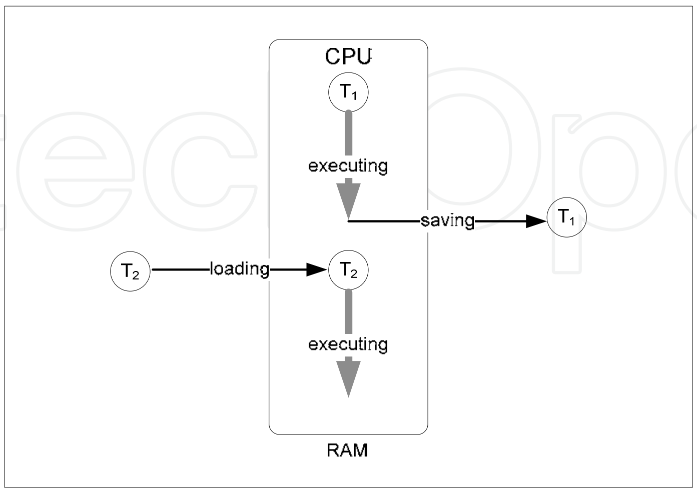
In order to deploy an application based on mobile agents, it is necessary to have an appropriate platform. There are three approaches for designing and implementing a platform for mobile agents. The first is using a programming language that includes instructions for mobile agents. The second approach is implementing mobile agents as extensions of the operating system. Finally, the last approach builds the platform as a specialized application that runs on top of an operating system.

µC/MAS is based on the extension of a real-time kernel by exploiting the similarity between the tasks' context switching and the agents' mobility. The Fig. 6 shows a context switching and a task migration from one node to another. The concept of task is fundamental in a real-time kernel. The task execution is done sequentially: the instructions that compose it are loaded into the processor and executed one after the other. A task is characterized at a given time by the data, the stack, the heap, the value of the program counter, register contents, etc. A program is a static entity like the contents of a file stored on a disk while a task is an active entity with a program counter specifying the address of the next instruction to execute and related resources. A task is dynamic as opposed to a program that is static.

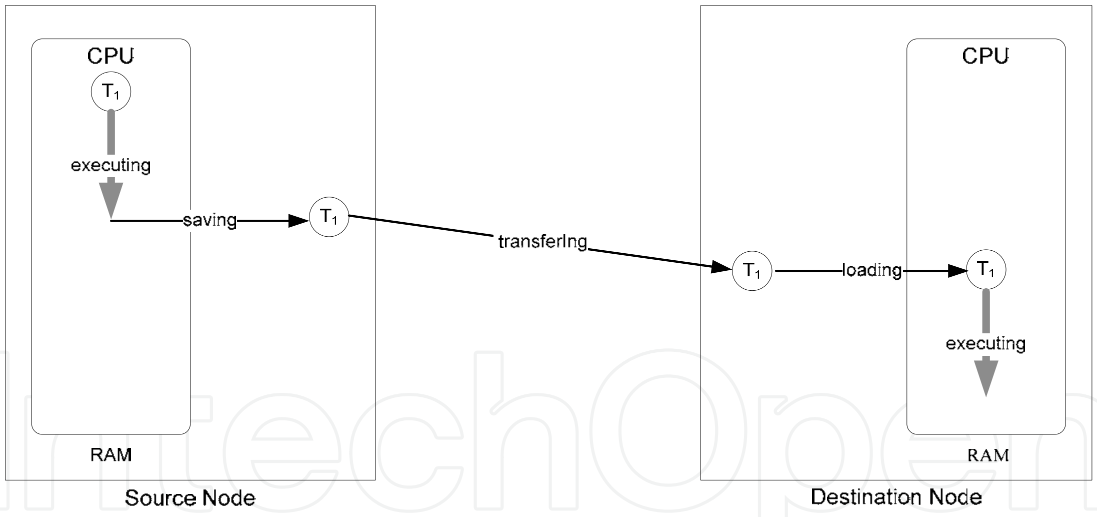
A task is typically an infinite loop that must necessarily be in one of the following five basic states as shown in Fig. 7 (Labrosse, 2002):

1. Dormant: the task resides in memory but is not available for scheduling
2. Ready: the task is waiting to be assigned to the CPU
3. Running: the task is one whose instructions are being executed by the CPU

4. Waiting: the task is waiting for a signal or a resource to continue its execution.
5. Interrupted: the task is interrupted when an interrupt has occurred and the CPU is in the process of servicing the interrupt.



a) CPU switch from task T_1 to task T_2.



b) Migration of the task T_1 from the source node to the destination node

Fig. 6. a) Context switching and b) a task migration from one node to another.

In the context of the µC/MAS, an agent is a task that is able to migrate from one node to another. When an agent decides to migrate, it suspends execution of the current node, the source node. Then, if the agent code is not already at the destination, it will be loaded from a server. Next, the data representing the state of the agent are transferred from source node to destination node. Once the agent reaches the destination node, it resumes execution where it left off on the source node. The Fig. 8 shows the migration of various components of a mobile agent.

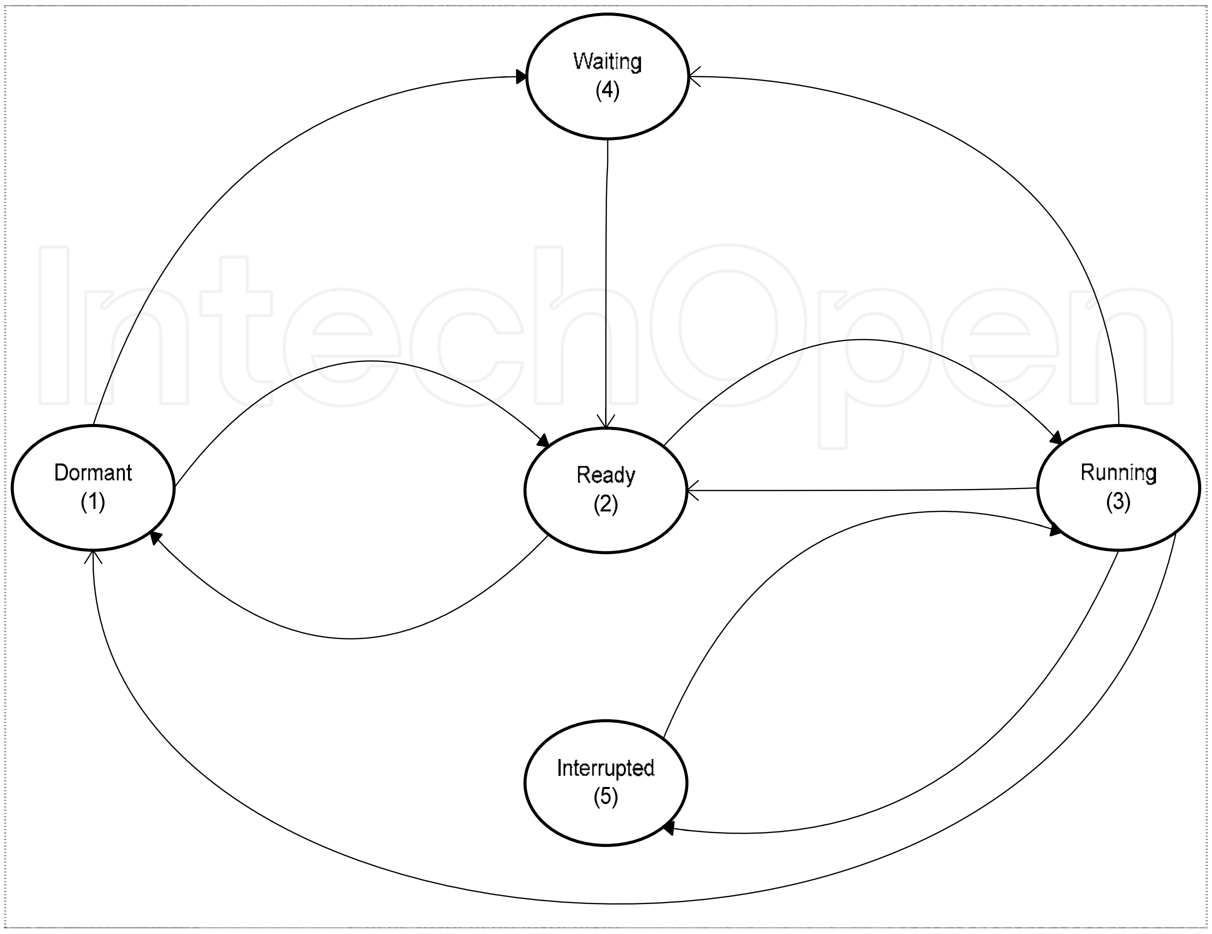


Fig. 7. Task states.

A real-time kernel provides two main functions: scheduling and context switching of tasks. Scheduling determines the ready task having the highest priority. When a task of highest priority is ready, the kernel saves the context of the current task (CPU registers) on the stack to allow a possible resumption. The context of the new task is retrieved from the memory area (the storage area, its stack) and execution resumes where it left off. Note that in a multitasking environment as a real-time kernel, each task has its own stack as shown in Fig. 9.

The μC/MAS is based on the extension of context switching mechanism (suspension and resumption) that exists in multitasking systems and especially in real-time kernels. From the context switching to the agent migration, we exploit the similarity between the CPU preemption by the real-time kernel and the agent mobility. This mechanism is integrated into the features of a real-time kernel allowing mobile agent based software to be implemented in the homogeneous embedded systems. The Fig. 10 shows the structure of the μC/MAS.

The mobile agent system differs from the migration process system in the sense that the agent moves at the time it chooses by means of a primitive while in a migration process, the system will decides when and where to go. The agents of the μC/MAS use a native code (C and Assembler) and can move as well in a wired network as in a wireless network. The agents operate on homogeneous platforms. Consequently, a mobile agent can move only towards a microcontroller of same physical architecture as well as the same environment of execution.

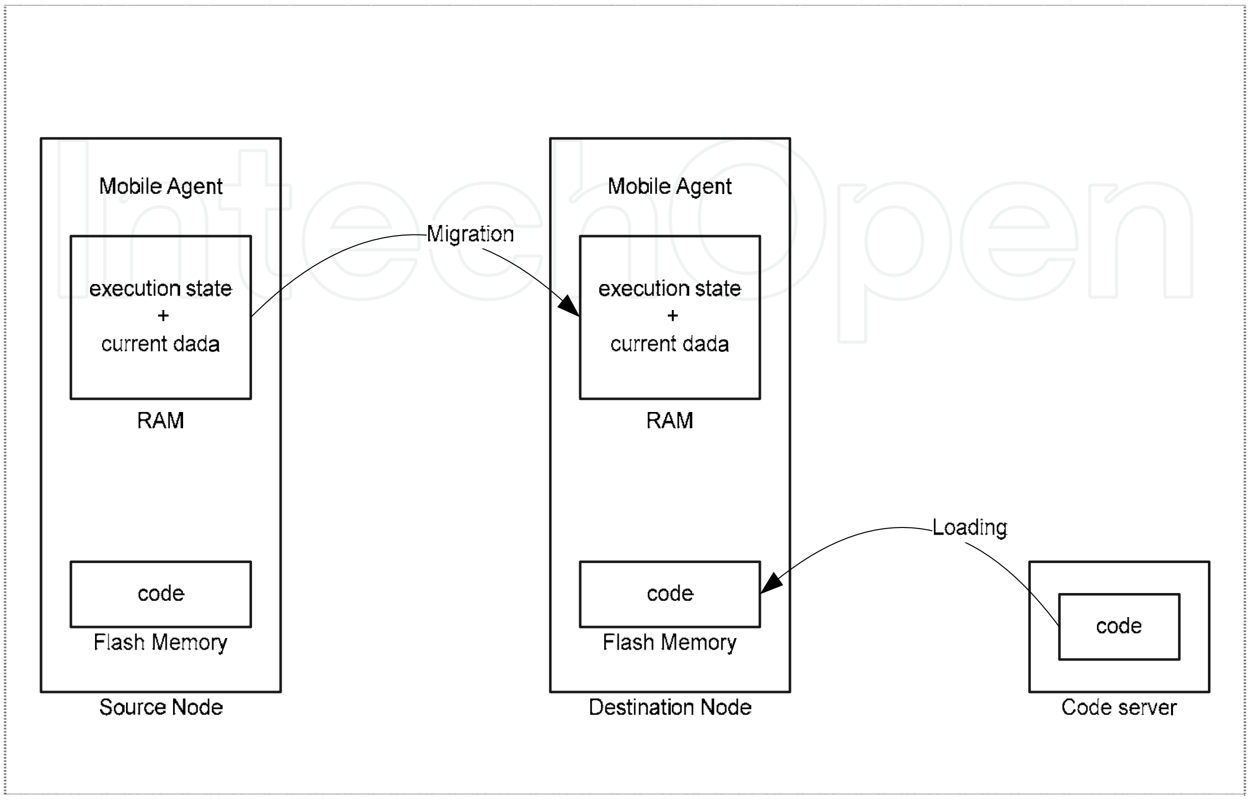


Fig. 8. Migration of various components of a mobile agent.

From the conceptual point of view, a mobile agent is a task that can autonomously migrate from one machine to another. As described above, there are two types of migration: strong migration and weak migration. The µC/MAS supports strong migration. There are very few agents platforms which support the strong mobility. These platforms do not use a native code. These platforms allow only weak migration: the mobile agent resumes execution from the beginning when it reaches the destination.

3.1 Migration of agent

Migration allows the transfer of a running agent from one node to another through a network. A platform supporting strong mobility must be able to capture and restore the structure of the agent in memory. As shown in Fig. 11, this structure consists of the following segments:

1. *Stack*: stores the function calls with their parameters and local variables. When a function return, parameters and variables are popped.
2. *Heap*: reserved for dynamic memory allocation.
3. *BSS (Block Started by Symbol)*: contains all global variables and static variables that are initialized to zero.
4. *Data*: contains global and static variables used by the program that are initialized
5. *Text*: contains executable instructions.

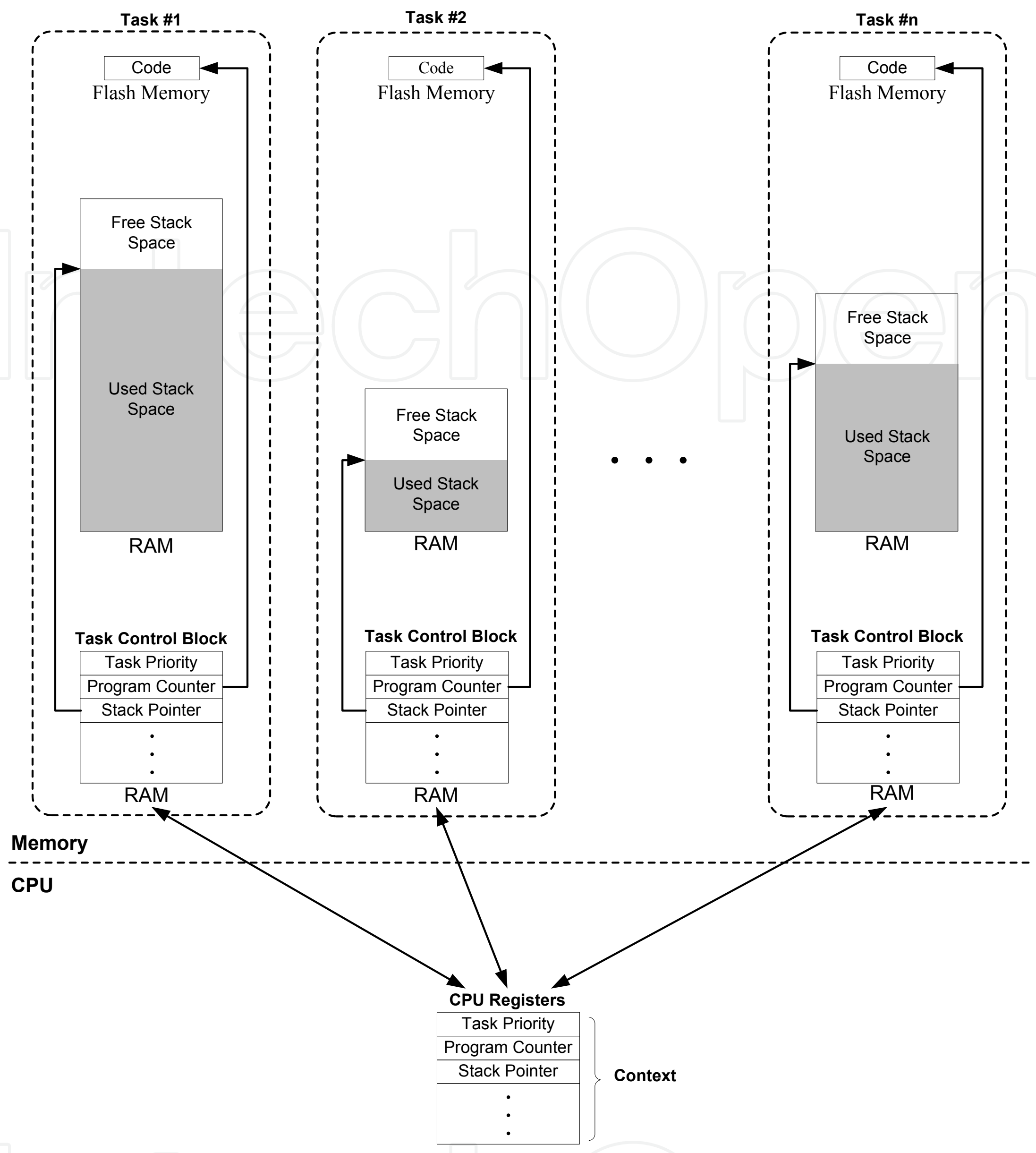


Fig. 9. Multiple tasks.

During migration in the source node, the task *agent* sends a transfer request containing its name and the address of the destination where it wants to go. Once the destination system accepts, the source system makes the following steps:

1. Capture of the current data. These are classified in two categories: the migrant data and the non-migrant data. The captured migrant data are formatted to transport used. Then, they are transferred towards the destination node.
2. Interruption of the task *agent* by generating a context switching. This saves the execution context of the task *agent* on the stack.
3. Capture of the stack and the task control block (TCB). The latter is a data structure that maintains the state of the task when it is preempted. When the task *agent* arrives at the destination node, the task control block allows the task to resume execution exactly where it left off.

4. Format the stack and the task control block to transport used.
5. Transfer of the stack and the task control block.

Before an agent is accepted into a destination system, it must be authenticated. In µC/MAS platform, the source node and destination node mutually authenticate by using passwords. Once mutual authentication is performed, the destination system makes the following steps:

1. Receiving, decoding and restoring of migrant data.
2. Receiving of the stack and the task control block.
3. Decoding the stack and the task control block.
4. Restoration of the stack and the task control block.
5. Resume execution of the task *agent*.

The Fig. 12 shows the algorithm of task *agent* migration.

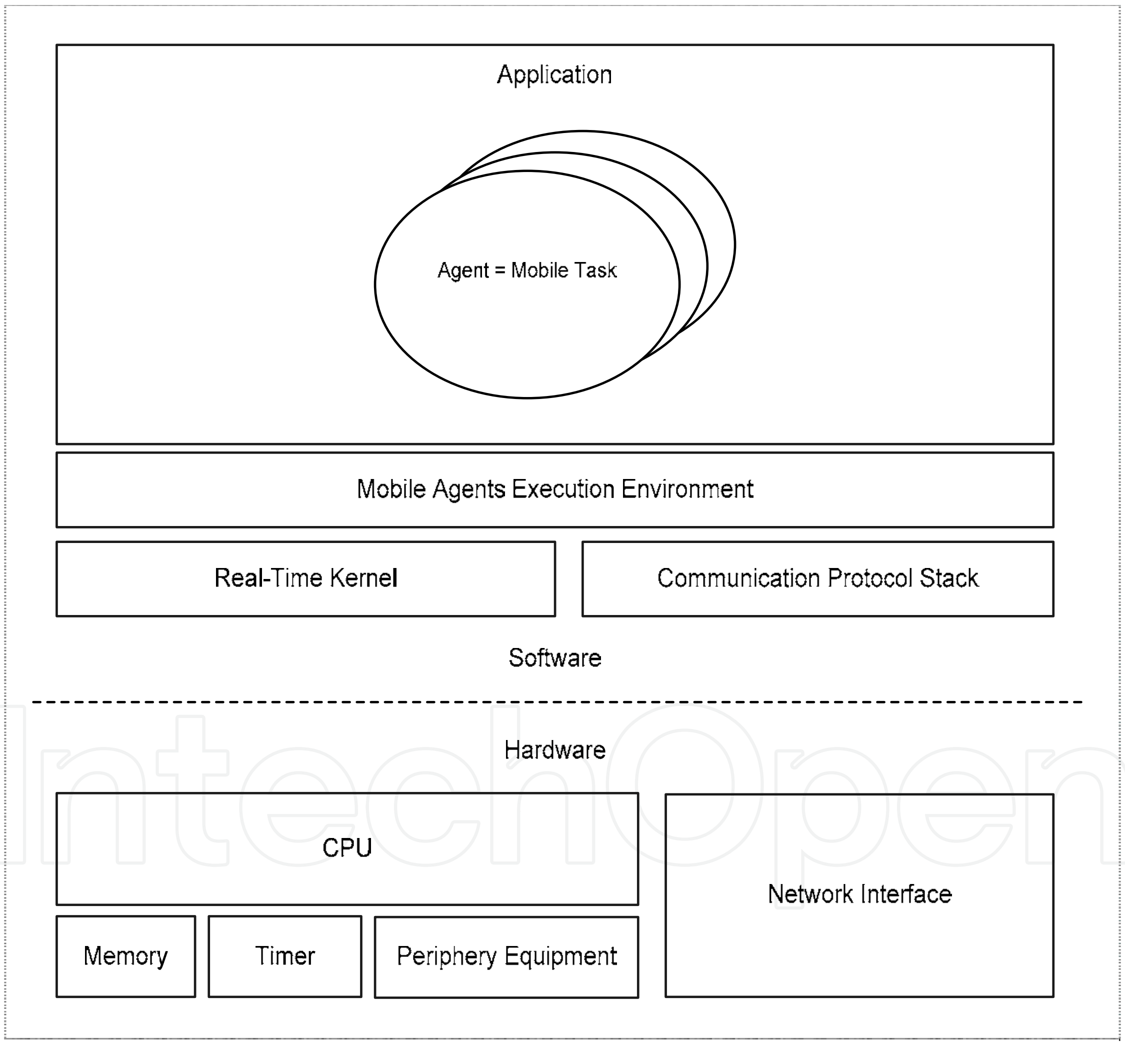


Fig. 10. Structure of the µC/MAS.

3.1.1 Directive migration and data types

Such as defined previously, an agent is a migrant task which could use local and/or global variables. This raises the following question: is it necessary migrating all the variables with the task that it used in the source node? The answer is no because some variables could be used by other tasks because they are not specific to the agent. In its life cycle, an agent uses different data types. In order to manage migration, we classify the data into different categories based on the classification of existing variables in C/C++. We may also apply this classification to other languages. In C/C++, the variables are classified into different categories according to how they are created and how they may be used. The different aspects that can take variables constitute what is called their storage class. The storage class of a variable allows specifying its life cycle and its place in memory.

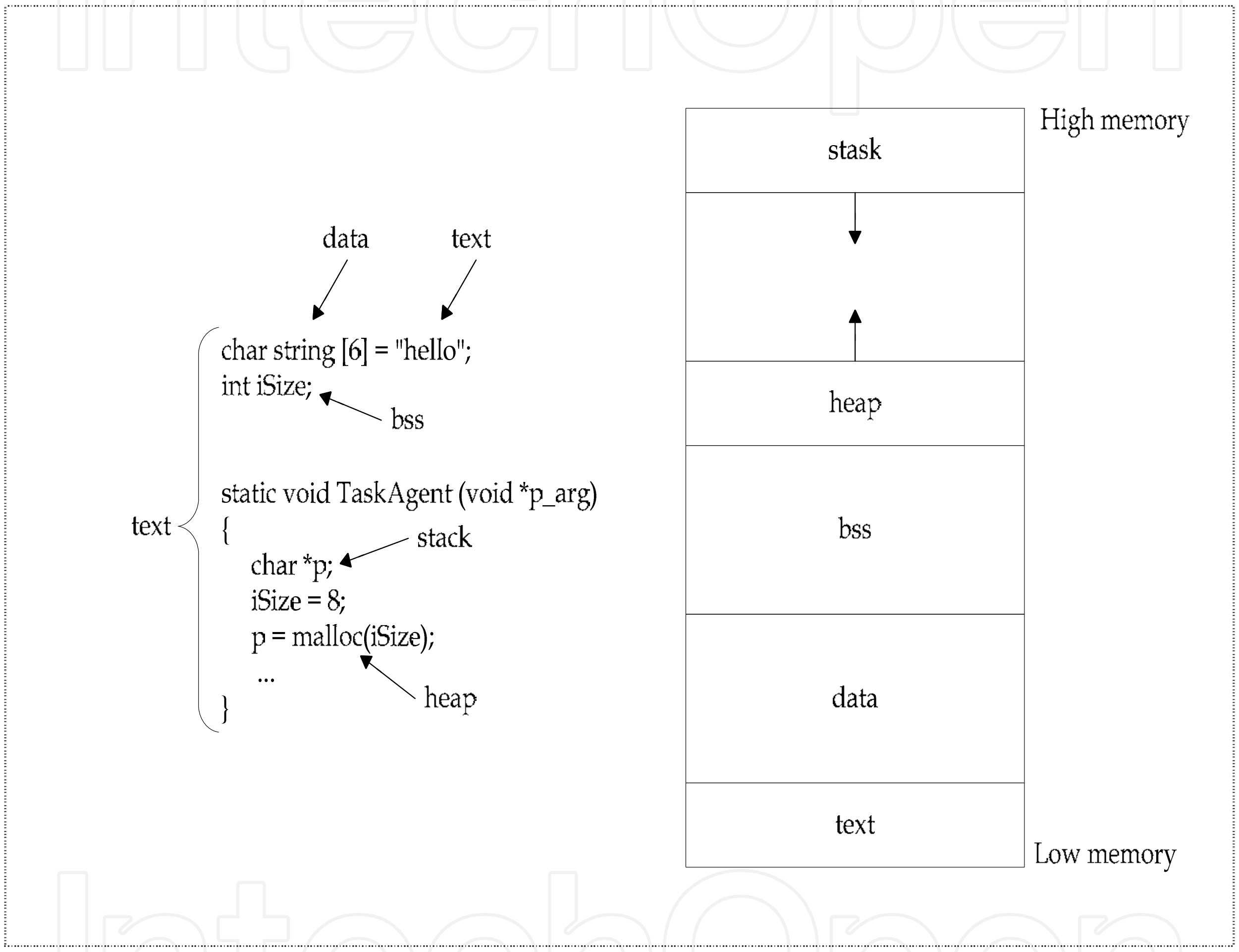


Fig. 11. Structure of the task *agent* in memory.

In order to identify the data that migrate with the agent, we first classify local and global variables. The local variables are created inside a block of instructions, in the case of the µC/MAS in the task *agent*. However, global variables are declared outside of any block of instructions in the zone of global declaration of the program. The local and global variables have different life cycles and different scopes according to their locations in memory. The variable scope is the program area in which it is accessible. The scope of global variables is the program while the scope of local variables is the block of instructions in which they were created.

The C/C++ has a range of storage classes for specifying the type of variables that you want to use:

- *auto*: the scope of an *auto* (automatic) variable is the function or block in which it is defined. The variable exists in memory during execution of the function or block in which it is defined. When all the instructions of the block are executed, the variable is removed from memory and its value is automatically lost. If the block is executed again, the variable is recreated. An *auto* variable has no initial default value.
- *static*: this storage class is used to create variables whose scope is the function or block of instructions in progress, but, unlike the *auto*, the *static* variables are not destroyed when the exit of this block. Every time that we enter this function or this block of instructions, the *static* variables exist and have value to those they had before we left. Their life cycle is that of the program, and they retain their values. If it is initialized at its declaration, it will not be reset by a subsequent call. A file can be viewed as a block. Thus, a *static* variable of a file cannot be accessed from another file. This is useful to separate compilation.
- *register* variables obey the same rules as *auto* variables, but they are not always stored in working memory. If the compiler can, it stores them in registers i.e. in memory areas included in the processor. If no register is available, the variable will receive the *auto* class. The & operator cannot be used on register variables. The advantage of having a variable stored in a register is the reduction of access time to this variable compared to the access time to a variable located in RAM. This can be useful when a variable is often requested.
- *volatile*: this class of variable is used in the programming system. It indicates that a variable can be changed in the background by another program (for example an interruption by a thread, by another process, the operating system or by another processor in a parallel machine). This requires reloading the variable every time the system refers to a processor register, even if the variable is already stored in one of these registers (which can happen if the compiler is asked to optimize the program).
- *extern*: this class is used to indicate that the variable can be defined in another file. It is used in the context of separate compilation.

There are also modifiers that may apply to a variable in order to specify its constancy:

- *const*: this keyword is used to make the contents of a variable unchangeable. In a way, the variable becomes a read-only variable. Warning, this variable is not necessarily a constant: it can be modified either through another identifier, or by an external program (such as volatile variables). When this keyword is applied to a structure, no structure field is writable.
- *mutable*: only available in C++, this keyword is used only for members of structures. It helps overcome the constancy of a possible structure for this member. Thus, a structure field declared mutable can be modified even if the structure is declared *const*.

In order to declare a particular storage class, it is sufficient to place one of the following keywords: *auto, static, register, etc*., before or after the variable. You can only use the not contradictory storage classes. For example, *register* and *extern* are incompatible, as well as *register* and *volatile*, and *const* and *mutable*. On the other hand, *static* and *const*, as well as *const* and *volatile*, can be simultaneously used. Global variables that are defined without the *const* keyword is processed by the compiler as variables of *extern* storage class by default. These variables are accessible from any program files. However, this rule is not valid for the variables defined with the *const* keyword. These variables are automatically declared *static* by the compiler, which means they are only available in the file in which they were declared. In order to make them accessible to other files, it is imperative to declare the extern keyword before defining them.

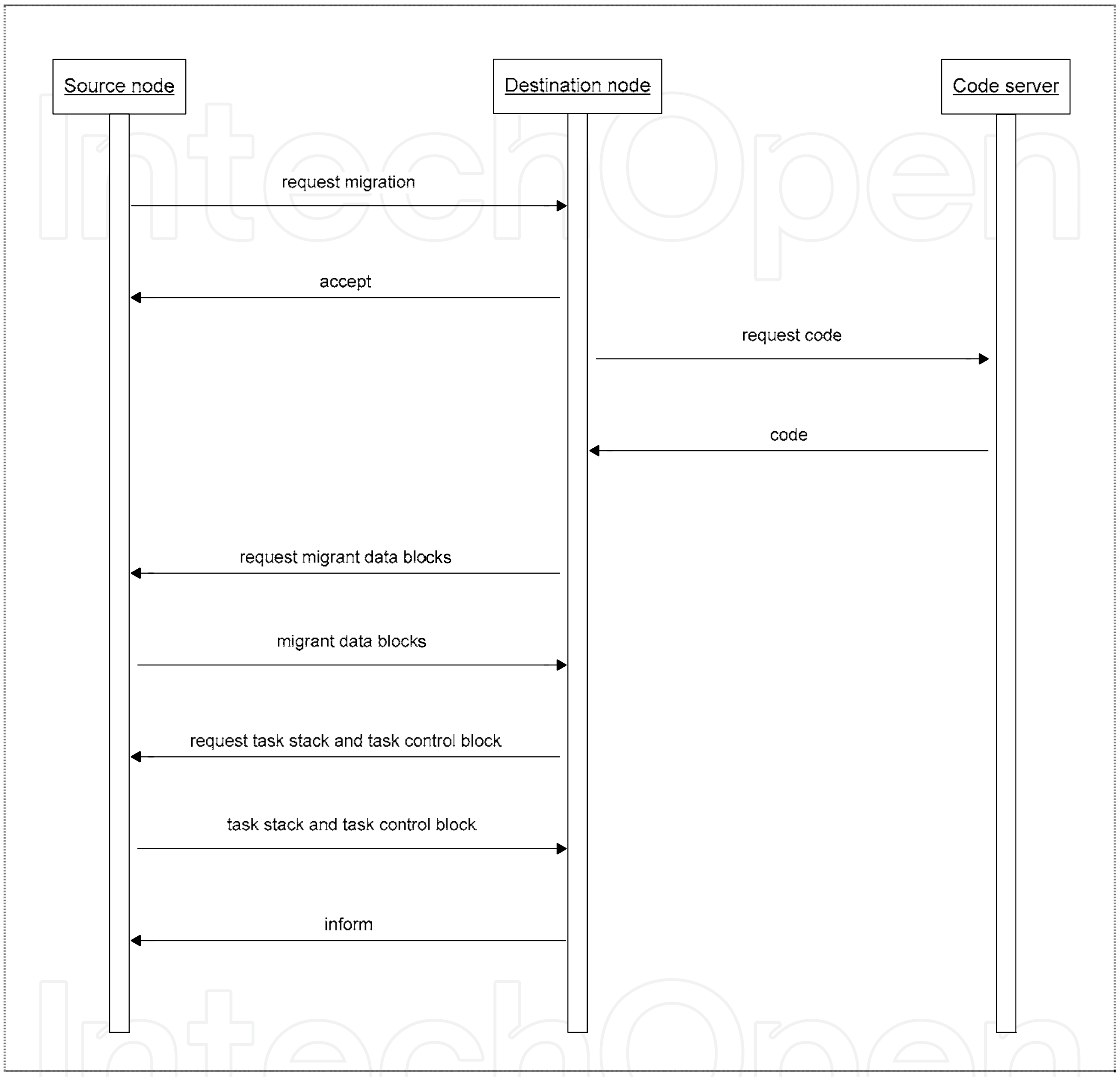


Fig. 12. Algorithm of task *agent* migration.

Second, we classify the local variables in automatic or dynamic. The area containing automatic variables is managed by the stack. For dynamic variables, we must first allocate memory using *malloc()* and subsequently liberate memory by using *free()*. The use of these functions in a real-time embedded system is dangerous because it is not always possible to obtain an area of contiguous memory due to the inherent fragmentation. The mechanism of memory partition proposed by the real-time kernels such as µC/OS-II (Labrosse, 2002) and µC/OS-III (Labrosse, 2010) provides alternatives to *malloc()* and *free()*. This mechanism allows obtaining fixed-sized memory blocks from a partition made of a contiguous memory area as illustrated in Fig. 13. Allocation and de-allocation of these memory blocks are done in constant time and is deterministic. The partition is usually allocated statically (as an array), but can also be dynamically allocated without being freed (never used the *free()* function). In an application, there may be multiple memory partitions as shown in Fig. 14. However, each specific memory block must always be returned to the partition where it originated. This type of memory management is not subject to fragmentation. Before using a partition of memory blocks, you must first create it. This allows the kernel to get the partition of memory blocks in order to manage their allocation and de-allocation. These containing blocks of the dynamic variables of the tasks *agent*, we call migrant blocks of data.

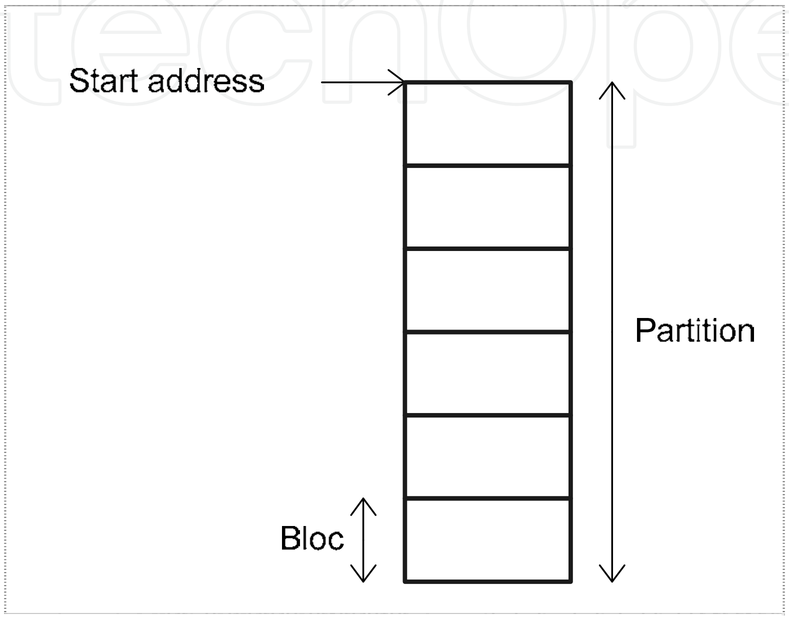


Fig. 13. Memory partition (Labrosse, 2002).

The Fig. 15 presents a classification of data as a binary tree in which each branch indicates the migration or not of these data with the agent:

1. *Automatic local variables* automatically migrate with the agent.
2. *Dynamically allocated local variables* migrate if the agent plans to use them during its travel. In the source node, the memory blocks containing these variables must be returned to their partition after use. The *dynamically allocated local variables* are stored in the stack but the spaces pointed by these variables are stored in heap.
3. *Dynamically allocated local variables* do not migrate when the agent does not use in its route. In the source node, the memory blocks containing these variables must be returned to their partition after use.
4. *Non-dynamically allocated global variables* do not migrate with the agent because the system has been designed to that effect in order to avoid that these variables would be used by other tasks.
5. *Dynamically allocated non-shared global variables* migrate if the agent plans to use during its travel. As in (2), the memory blocks containing these variables must be returned to their partition once the migration is performed.
6. *Dynamically allocated non-shared global variables* do not migrate when the agent does not use in its route. As in (3), the memory blocks containing these variables must be returned to their partition once the migration is performed.
7. *Dynamically allocated shared global variables* do not migrate with the agent.

3.1.2 Transfer format of the agent

The XML format in combination with the Intel HEX is used to transfer the task *agent* from one node to another. As shown in Fig. 8, a mobile agent consists of a code, execution state and current data. Note that the executable code of the agent does not undergo any transformation. It is loaded into the flash memory as produced by the compiler. The compiler used in this project produces an Intel HEX format object file.

The execution state of the agent is composed of the stack and task *agent* control block (Task Control Block, TCB). The stack contains the local variables of the task *agent*. The Current data are global variables specific to the task *agent*. The TCB is a data structure that is used by the real-time kernel to maintain the execution state of the task *agent* when it is preempted (Labrosse, 2002). The TCB contains the stack pointer, the priority of the task *agent*, the stack size, etc.

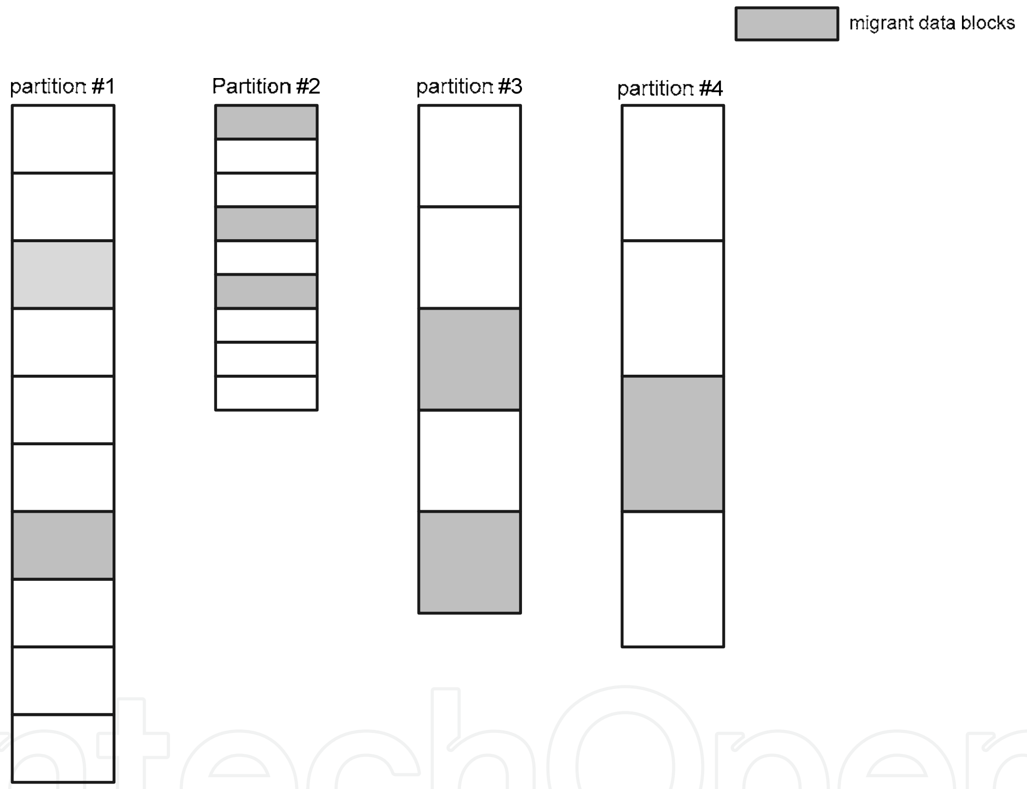


Fig. 14. Multiple memory partitions (Labrosse, 2002).

During migration of a task *agent*, the memory space used by the stack of the source node is not always available at the destination. We must therefore relocate the stack of each task as it appears at the destination. For the same reasons as previously mentioned, we use again the mechanism of the partitions to allocate the stack a memory space. In µC/MAS, we opt for static allocations. This consists in fixing the number and size of partitions of memory blocks. This approach has advantages in the context of embedded systems. Indeed, the use of fixed blocks of memory allows the allocation and de-allocation of this in a unitary manner, thus avoiding memory fragmentation. The number of memory blocks is directly dependent on the number of agents on the node at any given time. The memory requirements can be determined in advance depending on the structure and the number of agents circulating in the network. Thus, it becomes possible to produce a reliable mobile agent platform for homogeneous embedded systems. The Fig. 16 shows the relocation of a stack of tasks from one node to another.

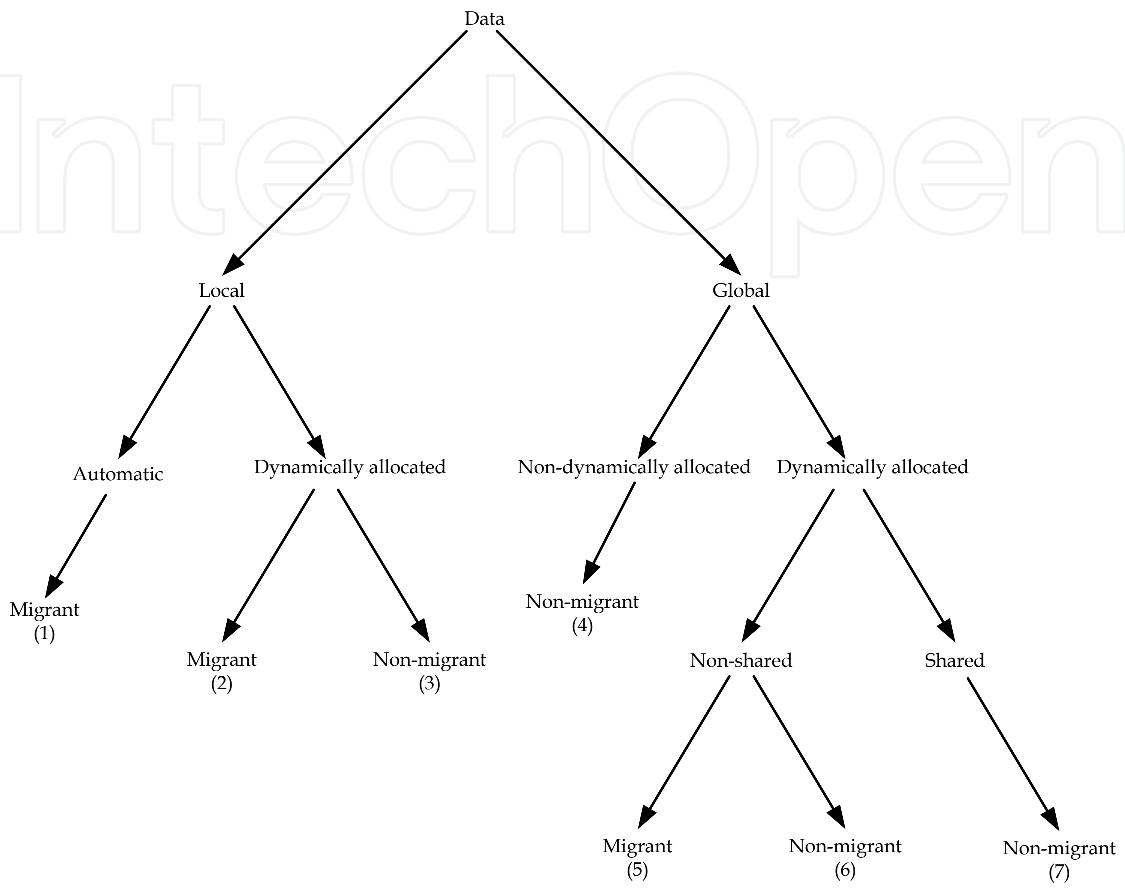


Fig. 15. Classification of data interacting with the agent.

XML presents the task *agent* (execution state and current data) as a document, and XML parser manages this document. The parser structures the document and the way the document is accessed and manipulated. It provides the following functionality: build documents, navigate, add, modify, or delete elements and their content. Unlike the TCB, the stack and the current data are first encoded in Intel HEX format before being incorporated into the XML document. As described in [Intel Corporation, 1988], the Intel Hex file is an ASCII text file that encodes and represents a binary file. Each line in an Intel HEX file contains one HEX record. These HEX records are made up of hexadecimal numbers. Data records appear as follows:

:10800000140000EA34F09FE534F09FE534F09FE57A. As shown in Fig. 17, the record is decoded as follows:

1. starts every Intel HEX record.
2. is the number of data bytes in the record.
3. is the address where the data are to be located in memory.
4. is the record type 00 (a data record).

5. is the data.
6. is the checksum of the record.

An Intel HEX file must end with the following record:

:00000001FF

The Fig. 18 shows a model of interconnection between a source node and destination node.

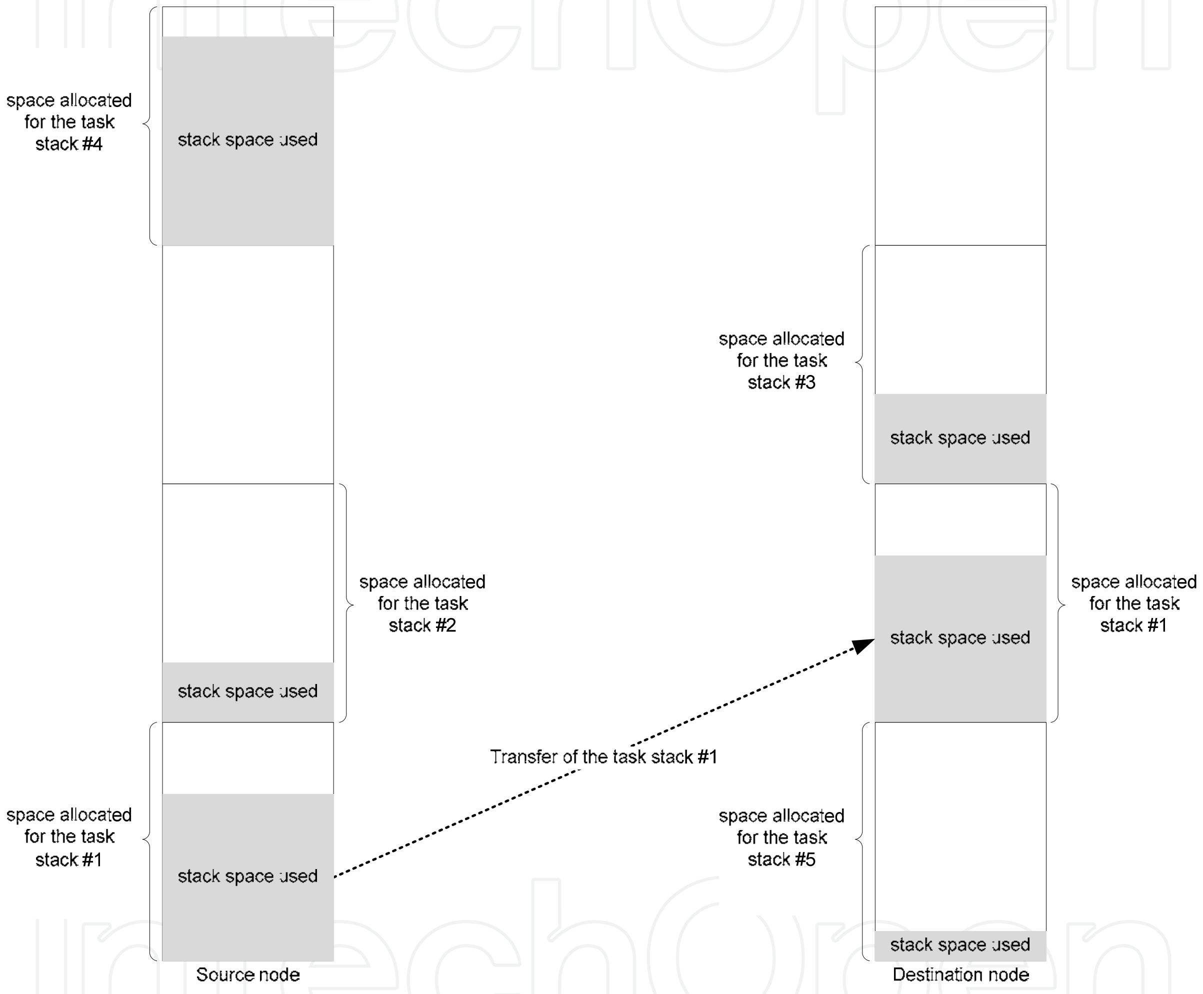

space allocated for the task stack #4
stack space used
space allocated for the task stack #2
stack space used
space allocated for the task stack #1
stack space used
Source node
space allocated for the task stack #3
stack space used
space allocated for the task stack #1
stack space used
Transfer of the task stack #1
space allocated for the task stack #5
stack space used
Destination node

Fig. 16 Relocation of a task stack from one node to another.

```
(1)   (2)    (3)     (4)    (5)                                   (6)
 |     |      |       |      |                                     |
 :     10    8000     00     140000EA34F09FE534F09FE534F09FE5      7A
```

Fig. 17 Intel HEX record.

4. Implementation of µC/MAS

µC/MAS platform is based on the extension of a real-time kernel called µC/OS-II Kernel. µC/OS-II is a portable, ROMable, scalable, preemptive, real-time deterministic multitasking kernel for microprocessors, microcontrollers and DSPs. µC/OS-II manages up to 250 application tasks and provides the following services (Labrosse, 2002): semaphores, event flags, mutual-exclusion semaphores that eliminate unbounded priority inversions, message mailboxes and queues; task, time and timer management; and fixed sized memory block management. µC/OS-II's footprint can be scaled (between 5 Kbytes to 24 Kbytes) to only contain the features required for a specific application. The execution time for most services provided by µC/OS-II is both constant and deterministic; execution times do not depend on the number of tasks running in the application. µC/OS-II comes with all the source code, written in portable ANSI C. However, this kernel cannot support mobile agents without modifications.

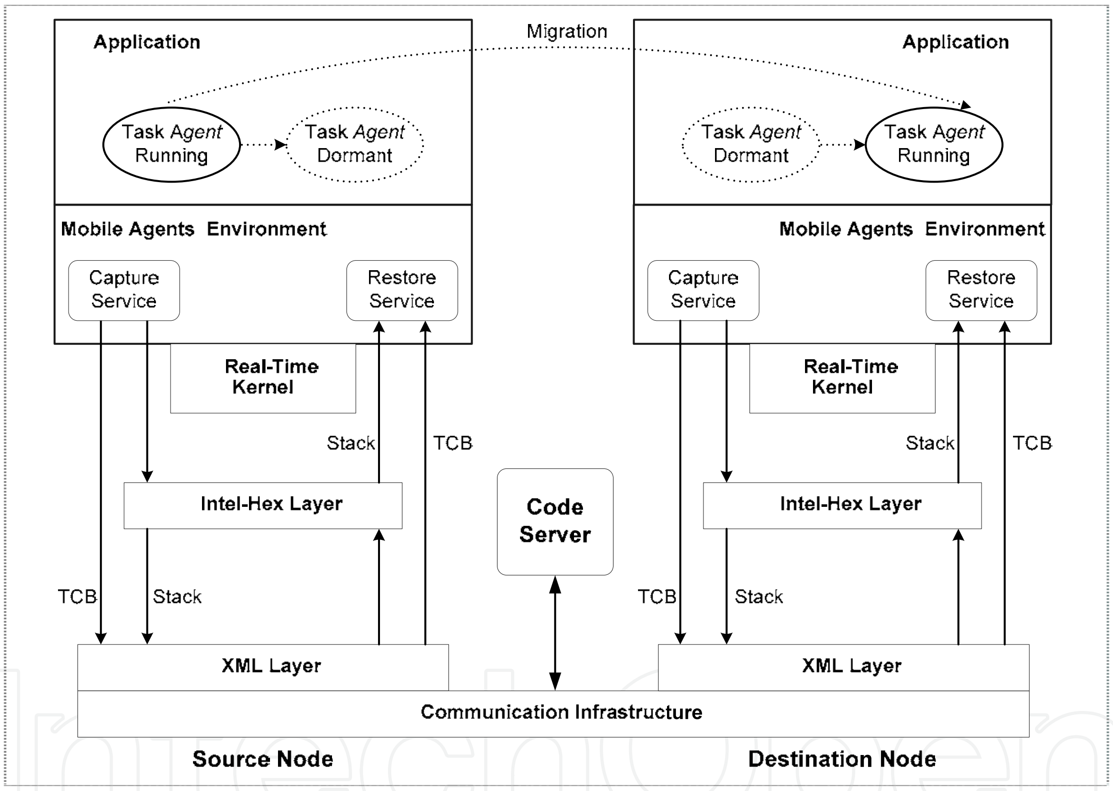


Fig. 18. Mobile Agents Platform Interconnection.

The choice of µC/OS-II is motivated by the need to use a small real-time kernel, because of the limitations incurred by the computing resource constraints of the embedded system. µC/OS-II has two major advantages: it requires not only a small memory but also it is open-source software. Thus, we can extend its code to implement a mechanism to capture and restore the task's state.

4.1 µC/MAS API

µC/MAS platform is implemented as an API. µC/MAS API is designed to integrate the features of an existing real-time kernel, in the case of this platform, µC/OS-II but also of other similar types such as µC/OS-III and µClinux. It is primarily intended for very small environments built around microcontrollers. Choice of kernel, i.e. µ/COS-II, should be written in ANSI C and its source code must be available to allow access to mechanisms for capturing and restoring the execution context that will be used by the mobile agents' platform. While most of µC/OS-II is written in C, for portability, it is still necessary to write some code in assembly language. For example, code for the context switching is written in assembly language, as it is not possible to access CPU registers directly from C. The device driver, code which initializes the hardware, also needs to be written in assembly language. The platform API is built on a modular architecture that allows integration with other services using a real-time kernel and different means of transport for agents. It includes the following modules at the base of its integrability with these services:

- Capturing/restoring module that interfaces with the real-time kernel;
- Encoding/decoding that allows formatting the agent (data + current execution state) according to the transport service;
- Module offers various transport services (TCP/IP, ZigBee, RFID, etc.) that provide the interface with the means of transport. Transport can be used synchronously as in the case of TCP/IP or ZigBee or asynchronously as in the case of using smart card storage (RFID).

```
<Doc>
        <TCB>
                <OSTCBStkPtr            INT32U="8105A760" />
                <OSTCBStkBottom         INT32U="81050A0C" />
                <OSTCBStkSize           INT32U="2800" />
                <OSTCBOpt               INT32U="03" />
                <OSTCBId                INT32U="0A" />
                <OSTCBPrio              INT32U="0A" />
                <OSTCBTaskName          string="TaskAgent" />
        </TCB>
        <Stack string=":10A76000200000D320000013000000008000000007B4
                       :10A770008105AE1404040404050505050606060655
                       :10A78000070707070808080809090909101010101029
                       :10A790008105A7B48105A7B88000C6608000C660A7
                       :10A7A0000100F0BC200000138105A7D48105A7B8E3
                       :10A7B0008000A8EC8000C5808105A7F00000000A99
                       :10A7C000010000026000000138105AA088105A7D8D6
                       :10A7D00080008CCC8000A7E8000000000A21300DD
                       :10A7E0005E093E4013005E353E4000A2973B59098A
                       :10A7F00000A21300A22F0A401300686D0A4000A2B5
                       :10A80000FAB9B42F2065685469626F6D6120656C78
                       :10A81000746E6567207369206E2074612065646FB3
                       :10A820006F6E20340D0A2E770000000000000000003B
                       :10A830000000000000685400006F6D2065656C69625F
                                           •
                                           •
                                           •
                       :10AA00008105AA0C1414141480008964000000014C
                       :00000001FF" />
</Doc>
```

Fig. 19. Transfer format of the execution state.

The mechanism of migration of the agents takes place in the following way: When the agent decides to migrate to another node, the execution thread (data and current state) of the task is saved. The execution thread is transported to the destination node using the means of transport (TCP/IP, ZigBee, RFID, etc.). At the destination node, a monitoring mechanism is to listen and identify the transport used to receive the agents arrive. This mechanism is implemented by a task that is part of the µC/MAS platform. The incoming agent is used by the task (for restoration) to create a new task that will host the execution thread of this agent. Note that in the source node, the task migrant is deleted.

```
<Bloc>
        <BlocSize>value</BlocSize>
        <BlocIntelHex =":XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                       :XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                             •
                             •
                             •
                       :00000001FF" />

</Bloc>
```

Fig. 20. Transfer format of migrant data blocks.

4.1.1 Capturing and restoring the execution context

In order to capture the execution context of a task *agent*, it is necessary to generate a context switching. To do so, we implemented the primitive *OSTaskMoveTo()*. The latter is called when the task *agent* decides to migrate to another node. Calling this primitive *OSTaskMoveTo()* generates the context switching of the task *agent*. *OSTaskMoveTo()* requires two additional arguments: *DestNodeAddr* and *MediumType*. The variable *DestNodeAddr* contains an address in the mechanism of transport such as an IP address, a MAC (Media Access Control address) to a Zigbee network, or an identifier of an RFID chip. The variable *MediumType* is the type of transport protocol used: TCP/IP, Zigbee, RFID, etc. After the context switching is done, the function *OSTaskContextCapture()* captures and encodes the stack and task control block (TCB) of the agent in a suitable for transport format. In order to capture the stack, the platform uses the value stored in the TCB *(stack pointer)* which is the pointer to the top of the stack and the starting address of the stack. From this information we can identify the stack space used to extract the addresses and data. Then, the addresses and data from the stack and task control block are encoded in a format suitable for transfer. For example, you can use a combination of two standards for data exchange: the Intel Hex format for binary elements as the content of the stack and XML for structured data such as the TCB. The Fig. 19 shows the transfer format of the execution state.

Once the execution context of the agent is encoded, *OSTaskContextTransfert()* is called for the transfer to the destination node. This transfer is transparent to the user. However, to check the status of the agent's transfer, the user can use *OSTaskTransfertStatus()* which return one of the following:

- *OS_NO_ERR* : the transfer of the agent is successful;
- *OS_PRIO_INVALID*: the priority associated with the agent is not available on the destination node;
- *OS_ADDR_INVALID*: the destination address is not valid;
- *OS_TASK_TIME_OUT*: the destination node does not respond;
- *OS_MIGR_ERR*: An error occurred during the transfer operation.

In cases where the migration fails, the agent continues to run in the node where it is located.

```
Static void TaskAgent (void *p_arg)
{
        ... // The Instructions before moving data
        OSMemBlocMove(DataBlocPtr, DestNodeAddr, DestDataBlocAddr, err);
        if(err == OS_NO_ERR)
        {
                // The data blocks transfert is successful, execution continues HERE
                OSTaskMoveTo(DestNodeAddr, MediumType);
                if(OSTaskTransfertStatus())
                {
                        // The task agent transfert is successful, execution continues HERE
                        ...
                }
                else
                {
                        //The task agent transfert is failed, execution continues HERE
                        ...
                }
        }
        else
        {
                //The data blocks transfert is  failed, execution continues HERE
                ...

        }
}
```

Fig. 21. Sequence using primitive *OSMemBlocMove()* and *OSTaskMoveTo().*

In the destination node, the data representing the execution context of the agent are received by a monitoring task that acts as an agents' server for the node. This task decodes and restores the context of agents received and starts executing. The received data is decoded to extract the stack and the task control block, and is restored to a new task to the destination node. To do so, we modified *OSTaskCreate()* that is the task creation function of the real-time kernel μC/OS-II in order to implement *OSTaskAgentCreate()*. This function uses *OSTaskContextRestore()* that we designed to restore the execution context of the task *agent*. The task of running the agent is recreated under the same conditions as the source node, especially with the same memory card and the same executable code. The data in the stack of the task *agent* are identical to those that were present on the source node. The task *agent* then resumes execution where it left off on the source node.

Note that a task *agent* consists of an execution context and current data. For migration of current data other than the stack, partitions of memory blocks offered by the μC/OS-II can be extended to design transfer mechanisms. The Fig. 20 shows the transfer format of migrant data blocks. When migrating a data block, the task *agent* uses the function *OSMemBlocMove()*. This allows extracting data from the memory block to save and then encode into a suitable transport format. The parameters of the function *OSMemBlocMove()* are:

- *DataBlocPtr* pointing the data block to be transferred to the destination node;
- *DestNodeAddr* that contains the network address of the destination node;
- *DestDataBlocAddr* that returns the address of the data block in the destination node;

- *err* that returns one of the following:
  - OS_NO_ERR: the transfer of data block is successful migrants;
  - OS_ADDR_INVALID: the destination address is not valid;
  - OS_TASK_TIME_OUT: the destination node does not respond;
  - OS_MIGR_ERR: An error occurred during the transfer operation.

The primitive *OSMemBlocMove()* calls the function *OSMemBlocCapture()*. This captures and encodes the data block. Then, the data block must be deleted from the source node if the transfer is successful. In the destination node, the block of migrant data is received by the task that acts as an agents' server for the node. This task decodes and restores the data block in the same memory partition than the source node. To do so, it implements the function *OSMemBlocRestore()* that we designed to restore the data blocks of the task *agent*. Restoring a data block takes place according to the following sequence: (a) a data block of the same type and same size as the source node is created in the destination node, (b) the data received from the source node are copied to the newly created block, (c) the address of the block and a code indicating that the restoration is successful are sent in the source node, (d) the source node, the address of the received data block is copied to a local variable, in this case in *DestDataBlocAddr* for later use in a possible migration of the task *agent*. The Fig. 21 shows a sequence of primitives using *OSMemBlocMove()* and *OSTaskMoveTo()*.

5. Evaluation of µC/MAS

In order to ensure the proper functioning of µC/MAS, we made certain numbers of the tests on various networks.

5.1 Wired network

In order to build the wired network, we set up a system composed of two ARM7-based microcontrollers (LPC-H2214/LPC-H2294) each one being connected with a PC. Each PC serves as a code server as well as an interface displaying the results. As shown in Fig. 22, the communication between PCs and microcontrollers is done via UART0 using the serial communication protocol. The UART1 of the source node is connected directly to the UART1 of the destination node to communicate between the two microcontrollers.

In order to test µC/MAS platform on a wired network, we implemented a sample application of a mobile agent that performs the following steps to test the platform on the wired network:

1. The agent initializes a number of variables of different types (char, short, int, float, double) in a node 1;
2. It performs operations on these variables and displays their contents, and then it moves to a node 2.
3. The agent continues operations, displays the contents of variables and returns to the starting node, node 1;

In this experiment, the agent performs different operations. For example, the agent performs arithmetic operations such as addition, subtraction, etc. It also performs concatenation of strings contained in variables from a remote node.

This experience has allowed us to validate the strong migration. The agent interrupts its own execution on the source node, and then the code is transferred from the server to the destination node. Then, the data representing the state of the agent is transferred from the source node to the destination node. Finally, when the agent arrived at the destination node, it continues to run where it left off on the source node. The execution thread of this experiment is displayed on the screen of the PC that is connected to the microcontroller of each node.

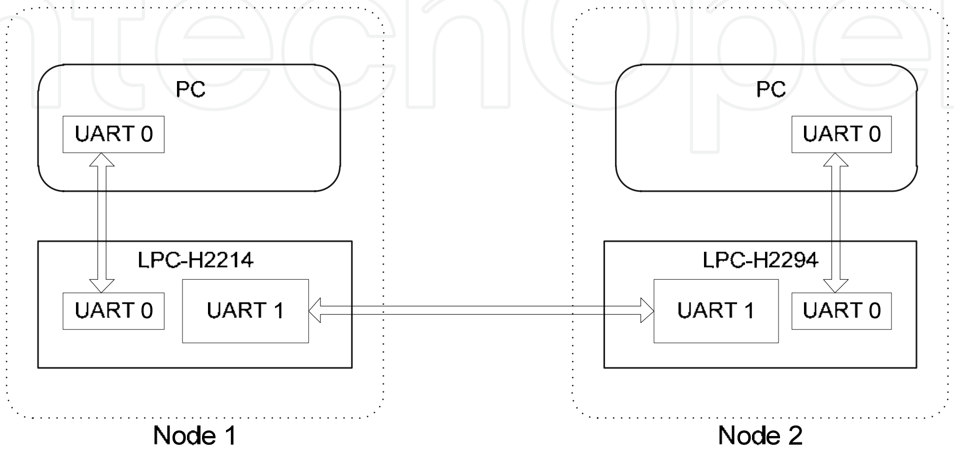


Fig. 22 Model of the wired network.

5.2 Zigbee network

In order to build the Zigbee network, we set up a system composed of six nodes. Each network node consists of:

1. ARM7-based microcontrollers (LPC-H2214/LPC-H2294);
2. Module X-bee;
3. PC to follow the thread of execution.

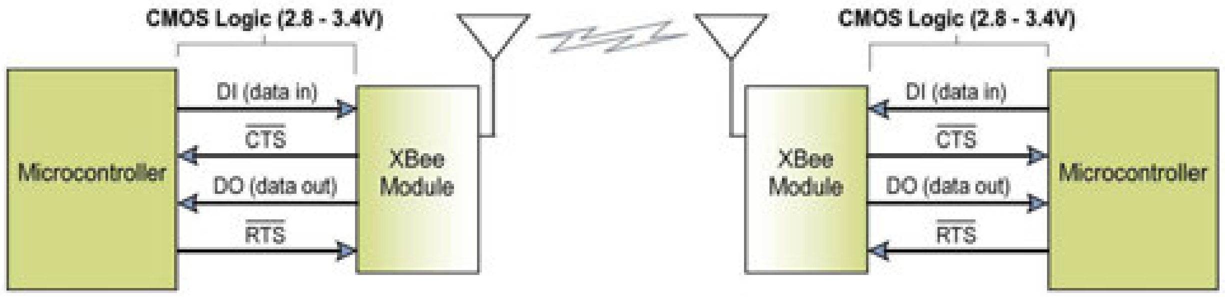


Fig. 23 Interconnection between XBee module and microcontroller (Digi International, 2009)

In order to test µC/MAS platform on the Zigbee network, we conducted a sample application of mobile agent similar to the wired network. The difference between the two experiences is the network communication infrastructure, the number of nodes used and the number of laps completed by the agent. In this experiment, as previously mentioned, we used six nodes. As shown in Fig. 23, the XBee module has a serial interface that connects directly to a microcontroller.

Furthermore, contrary to the previous experiment where the agent makes only a single iteration, in this one the agent makes five times the tour of the six nodes. As the previous experiment, this one allowed us to validate the strong mobility supported by this platform of mobile agents. The Fig. 24 illustrates the model of the Zigbee network where a mobile agent makes the tour of six nodes by beginning his route of the node 1.

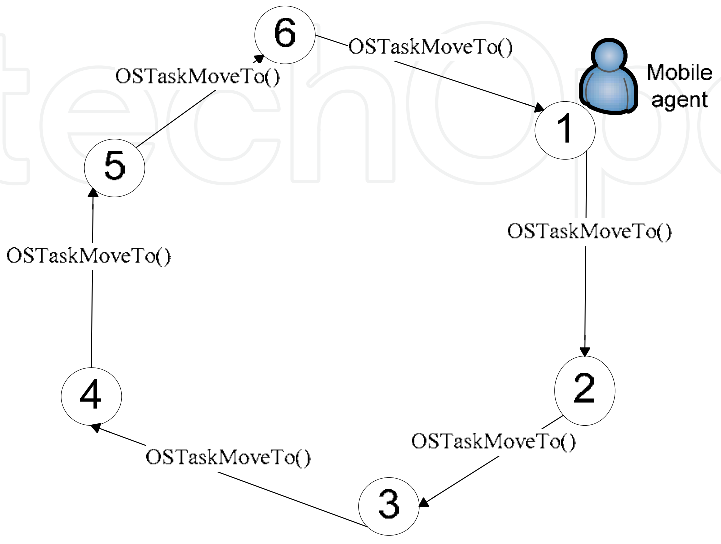


Fig. 24 Mobile Agent on Zigbee network.

In both networks we have experimented, the transfer speed of the agent depends on the communication link between the nodes. The Table 1 summarizes the quantitative evaluation of the experience we have done on the wired network and the Zigbee network. In order to calculate the transfer speed of a frame, we calculate the transfer rate of a bit. A frame is composed of 8 bit data, 1 stop bit and 1 start bit. The LPC-H2214/ LPC-H2294 can operate up to 60 MHz CPU frequency. For baud rate of 115,200 bps and CPU frequency of 60 MHz, the calculation of the constant C is as follows: C = frequency / (baud rate x 16) = 60 x 10^6/(115200 x 16) = 32.552. This can be approximated to C = 32. The actual baud rate = 60 x 10^6/ (32 x 16) = 117187.5 bps. This gives a period of 8.53 microseconds. The transfer time of a frame can be calculated as follows: the transfer time of a frame is 10 bits x 8.53 microseconds = 85.3 microseconds. Thus, we can calculate the transfer time of the execution state in the wired network like this: 34309 bytes x 85.3 microseconds = 2.926 seconds. This calculation does not take into account a possible loss of frames.

	Wired network	Zigbee network
Baud rate (bps)	115200	115200
CPU frequency (in MHz)	60	60
Used stack size (bytes)	1024	1024
Size of execution state in transfer format (in bytes)	3011	3011
Transfer time of execution state in transfer format (in seconds)	1.41	2.88

Table 1. Quantitative evaluation of wired and Zigbee network.

6. Conclusion

In this chapter, we presented the stages of the development of a mobile agent platform for embedded systems. By utilizing a context switch mechanism which already exists in the multitasking system, we have designed a mobile agent migration method and implemented it inside a real-time kernel. We have also designed and implemented a transfer format and a communication protocol stack for mobile agents' migration on a wired or wireless network. For the wireless network, the code of the agent was not transferred because of limited bandwidth.

The advantage of our mobile agent migration method, in regard to the other projects, is that it can be implemented in any multitask system, because we employ mechanisms that exist in these systems. The main advantage of our platform of mobile agents is its small size, which allows it to run on machines having a memory as small as one megabyte.

7. References

Bellifemine, F., Caire, G. & Greenwood, D. (2007) *Developing Multi-Agent Systems with JADE* In: John Wiley & Sons Ltd, ISBN 978-0-470-05747-6, Chichester, England.

Borenstein, N. S. (1994) *E-mail with a mind of its own: The Safe-Tcl language for enabled mail,* IFIP Transactions C (25):389–402.

Cardelli, L. (1995) *A language with distributed scope*, Annual Symposium on Principles of Programming Languages, Proc. 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, San Francisco, CA, pp. 286 – 297.

Digi International (2009) *Product Manual v1.xEx - 802.15.4 Protocol, XBee®/XBee-PRO® RF Modules* http://ftp1.digi.com/support/documentation/90000982_B.pdf (The last time accessed, November 18, 2011)

Dilyana, S. & Petya, G. (2002) *Building distributed applications with Java mobile agents*, International workshop NGNT, pp. 103 - 109

Domel, P. (1996) *Mobile Telescript agents and the Web*, In Digest of Papers, COMPCON '96, Technologies for the Information Superhighway, 41st IEEE Computer Society International Conference, pp. 52–57, IEEE Computer Society Press.

Ferber, J. (1999) *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, In: Addison Wesley Professional, ISBN 9780201360486.

Intel Corporation (1988) *Hexadecimal Object File Format Specification*, http://microsym.com/editor/assets/intelhex.pdf (The last time accessed, October 26, 2011).

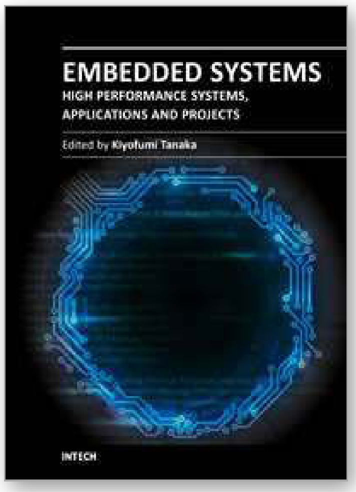
Lange, D. B. & Mitsru, O. (1998) *Programming and Deploying Java Mobile Agents Aglets, 1st edition,* In: Addison-Wesley Longman Publishing Co., Inc., ISBN 0201325829 Boston, MA, USA.

Lange, D. B. & Mitsru, O. (1998) *Programming and Deploying Java Mobile Agents Aglets, 1st edition,* In: Addison-Wesley Longman Publishing Co., Inc., ISBN 0201325829 Boston, MA, USA.

Labrosse, J. J. (2002) *Micro/OS-II The Real-Time Kernel, Second Edition,* In: CMP Books, ISBN 1578201039, San Francisco, CA, USA.

http://micrium.com/page/products/rtos/os-ii (Last time accessed, October 26, 2011)

Labrosse, J. J. (2010) µ*C/OS-III: The Real-Time Kernel and the NXP LPC1700,* In: Micriµm Press, ISBN 9780982337554, Weston, FL, USA.


EMBEDDED SYSTEMS
HIGH PERFORMANCE SYSTEMS, APPLICATIONS AND PROJECTS
Edited by Kiyofumi Tanaka
INTECH

Embedded Systems - High Performance Systems, Applications and Projects
Edited by Dr. Kiyofumi Tanaka

ISBN 978-953-51-0350-9
Hard cover, 278 pages
Publisher InTech
Published online 16, March, 2012
Published in print edition March, 2012

Nowadays, embedded systems - computer systems that are embedded in various kinds of devices and play an important role of specific control functions, have permeated various scenes of industry. Therefore, we can hardly discuss our life or society from now onwards without referring to embedded systems. For wide-ranging embedded systems to continue their growth, a number of high-quality fundamental and applied researches are indispensable. This book contains 13 excellent chapters and addresses a wide spectrum of research topics of embedded systems, including parallel computing, communication architecture, application-specific systems, and embedded systems projects. Embedded systems can be made only after fusing miscellaneous technologies together. Various technologies condensed in this book as well as in the complementary book "Embedded Systems - Theory and Design Methodology", will be helpful to researchers and engineers around the world.