9

Designing secure agents with O.O. technologies for user's mobility

David Carlier
RD2P - Recherche & Développement Dossier Portable
CHR Calmette, 59037 Lille Cédex - France
tel: +33 20 44 60 46, fax: +33 20 44 60 45, email: david@rd2p.lifl.fr

Patrick Trane
TIT - Tokyo Institute of Technology
Ookayama 2-12-1 Meguro-ku Tokyo 152 - Japan
tel: +81 35499 7001, fax: +81 35734 2817, email: patrick@cs.titech.ac.jp

Abstract

Using different kinds of computers from different locations has become a classical phenomenon. A user is said to be mobile when he does not always communicate with the outside from the same location. More and more people are being included in this category, such as people using a workstation linked to the Internet at their office, a microcomputer equipped with a telephone modem at home or a portable computer communicating via a wireless link. Features of these stations can be drastically different [GC94] especially for mobile computers due to numerous constraints such as weight, size, low communication flow and energy consumption [FZ94], [IV94]. Communication with a mobile user depends on the terminal used. This paper proposes an agent-based system in which one agent is associated to one user in order to simplify his tasks. Object-oriented technologies will be used as a way to guarantee agent's data privacy, protect agent acceptor's sites from intrusions and, in case of loss, make agent's recovery easier to perform.

Keywords

Mobile Computing, Oriented-Object Technology, Fault Tolerance

1 DESCRIPTION OF THE PROPOSED SYSTEM

This proposed system aims at enabling anybody to use the same services and data whatever the kind of terminal. To do so, a *representation agent* is widely used. A representation agent is a piece of software able to travel throughout the wired network and excecute itself on acceptor sites [CT95].

J. L. Encarnação et al. (eds.), *Mobile Communications*

A unique representation agent is associated with a given user. An agent takes in charge three main tasks :

- The agent is the user's representation, able to act on his behalf even when he is logged out.
- The representation agent manages data sent from/to a mobile user. This data must be specific to the user's current environment in order to allow him to be able to use his applications from different terminals. For instance, if the communication flow and the resolution of the current user's computer screen are weak, a large image may be reduced.
- The representation agents are located at acceptor sites providing an execution environment for agents. An agent is a mandatory intermediary between a user and all his interlocutors. The agent remains fixed during the whole communication even when the user is moving.

The last property allows an easier integration of mobile computing into current systems. However, when there is no external communication and when the agent is too far from its associated user, the agent is allowed to *migrate* to get closer to the user. For example, if an European person travels to Australia, his associated agent must follow him so as to avoid extra use of the network and to reduce message latency; if the user is communicating with an Australian station, it would be very inappropiate to have messages travelling from Australia to Europe and vice-versa. This step is called the agent migration. Using an agent is particularly suited to address the following problems for mobile computing described in [MDC93] :

- *Task delegation* : It allows the user to save energy and get more processing power. The delegation is performed by sending a small program or script to the user's agent. The computation results are returned to the user.
- *Asynchronism management* : As the user is often disconnected, the representation agent can be viewed as an ever active dedicated component on the wired network. Its behavior can be defined with the help of scripts.
- *Communication strategies* : All data sent to the user are first filtered by the agent with respect to both the communication flow and the device features. For example, it is no use transferring a 16-million-color picture to the user if his mobile station is only equipped with a black-and-white color screen. On the other hand, emitting data requires more energy than receiving data. Instead of sending a request each time the user wants to access a document, it is more adapted to send a short script describing data he wants to receive regularly. For example, if he is used to consulting news relative to a specific newsgroup, each time the user is connected, new items are automatically sent.

2 AN OBJECT ORIENTED APPROACH

The use of representation agents on acceptor sites involves several security problems. Some requirements must be taken into account so that, on the one hand, an agent cannot

voluntarily or otherwise damage an acceptor site and, on the other hand, an agent is not able to read or modify another agent's data located on the same site.

Some constraints are therefore applied on the agent processing domain. Two methods are proposed :

- The first one is to interpret the agent code. The user defines his own agent and sends it to an acceptor site. An agent interpreter on this acceptor site verifies that all instructions are correctly processed. The main drawback is the slow execution speed : running interpreted code is much slower than running native code.
- The other possibility is to predefine the code of the agent. If a user wants to get an agent on an acceptor site, he requests it to create one from a predefined code. This code can be written in native code, so the execution speed is improved in comparison with the first method. The features and the behavior are, however, predefined, that is, the agent cannot damage an acceptor site and cannot use another agent's content, as will be indicated hereafter.

The first method allows a user to define his agent the way he wants, according to the interpreted language. In the second one, the code is predefined. This code mainly enables the agent to communicate with the user and the outside world. It is also used to manage data and scripts sent by the user. The agent consists of (1) the code part *ie* the agent code and (2) its individual part, *ie* personal data and scripts. Thanks to the data and scripts, the user can define its own behavior. Moreover, as the agent code is the same for all agents, only the agent data and scripts are affected by the migration phase.

The use of object-oriented concepts enable an easy design of the system. An object is composed of two sets of elements : the interface (code), and the structure (data). Creating an object is achieved by sending a *create* message to an object server.

A representation agent can then be viewed as an object. On each acceptor site, an agent objet server is responsible for the agent creation. To create an agent within an acceptor site, a *create* message must be sent to the agent server on the site. Once a blank agent is created, the user personalizes it by sending a *personalize* message with a set of data related to the user and a set of scripts defined by him.

To process an agent migration, a *create* message is sent to the target acceptor site by the agent that wants to migrate. It then personalizes it and transfers its own personal data and scripts. When these operations are correctly performed, it destroys the copy remaining on the user site.

The agent code itself is not important. It may be different according to acceptor sites. However, the interface and the behavior of an agent must be independant of the site.

3 REPRESENTATION AGENT OBJECT STRUCTURE AND SCRIPT OBJECT

3.1 Agent structure

The user can add a script into his associated representation agent by sending it the message *AddScript*. When receiving such a message, the agent creates a new instance

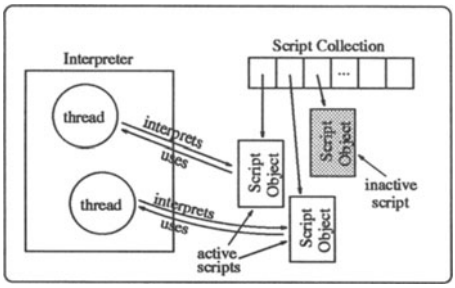

Figure 1 Representation agent object structure

of the class *ScriptObject* and adds it to its *ScriptCollection* as shown in Figure 1. One schedule is associated to each script sent to the agent to define the time and the frequency of its execution. The *AddScript* message is followed by two arguments : the script itself and the associated execution schedule. Once this method is executed, a script identifier is returned to the user. This identifier allows him to modify or remove the script from the agent. When the execution of a script is no longer planned in the schedule, the script can automatically be removed from the *ScriptCollection* of the agent object.

When a script is activated according to the schedule, a thread of the agent's script interpreter is created. Its role is to manage the execution of a given script during its running time. After the end of the execution, this thread is removed and the script object returns to an inactive mode.

3.2 Script structure and script execution

As mentioned in the previous part, a script can be in two different states : inactive or active. In the former case, the script object can be just considered as data. It is only made of two elements : one script text and one schedule. On the other hand, the script object structure is extended by an execution block. This one is used as a memory dedicated to the execution data of the script. This memory is divided into three elements : a data stack, a set of registers (such as an ordinal counter) and a data space for the variables storage as represented in Figure 2.

A method *Start* run on a script object enables to create an interpreter thread associated to this object and to create an execution data space. The thread is then activated by the script object. The method *Abort* kills the thread associated to the script and frees the execution data space.

3.3 Agent object migration

A representation agent can migrate only when no communication is in progress. All its active scripts must not be communicating. The migration must allow the representation agent to suspend both its execution and those of its scripts, to travel accross a network

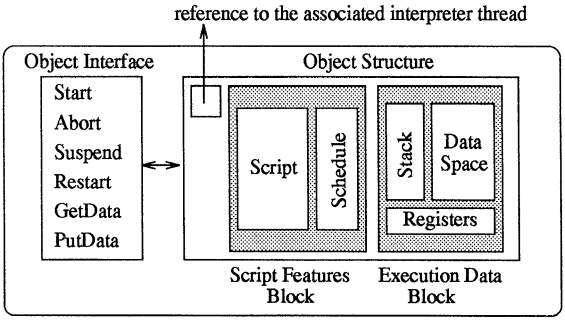

Figure 2 Active script object structure

from the current agent's acceptor site to another one and finally to end up with the same execution environment as before the migration.

It is necessary to identify all data to be transferred from one site to another. Minimizing the quantity of transferred data is an important issue. As it is entirely re-created according to the new site hardware and software features, the agent interface does not need to be sent to the target site. Even if the agent code may differ, both its interface and its behavior do not change according to their location on various acceptor sites. However, the data contained in an agent object must migrate to the target site. The scripts can be considered as data. Since the running scripts contain data required by an interpreter to continue their execution, the formerly defined script structure allows them to keep their execution context even after a migration. For all scripts for which the execution has been suspended, a new interpreter thread will be created and will use the execution data contained by the script object. Four methods must be added to the script object interface :

- *Suspend* which stops the execution of a script before a migration.
- *GetData* which allows the migrating agent to get data of a script object in order to send them to the new script object on the target acceptor site.
- *PutData* which is used to put script data from the old agent script object into the corresponding new agent's script object.
- *Restart* which enables the script object to continue its execution from the instruction where the script was suspended.

4 LOSS AND RECOVERY OF AGENTS

4.1 Identification of problems

The system presented here entirely relies upon the presence of the agents. As the agents are located on acceptor sites and may migrate from one site to another, if an acceptor site fails, two main problems may be identified : (1) all agents located on this acceptor site are lost, and (2) all agents migrating towards this faulty are will not reach their destination. Although the problem of acceptor sites diagnosis is beyond the scope of this paper [TC96], it is important to mention that it provides a useful tool to quickly warns the system to forbid any further migration towards faulty sites and initiate a recovery process for lost agents.

Recovery of agents must be done with care. Two main problems are identified. Reconstruction of an agent after its acceptor site has failed and restarting from a consistent global state without any message losses. The solution proposed also focuses on low-overhead dynamic reconfiguration strategies. Problems are as follow :

Checkpointing [AB94] must be performed with care so that the saved states form a consistent global state. Agent A sends a message M_1 to agent B. A has taken a checkpoint C_a before sending this message. B takes a checkpoint C_b right after having received this message as shown in Figure 3. Subsequently, A fails and restarts from its last checkpoint C_a. At this stage, the system global state is inconsistent for A's local state shows no message sent to B while B's local state indicates that a message has been received from A. This remains true even if B restarts from C_b.

Rollback-recovery [KT87] from consistent checkpoints may also cause message losses. In Figure 3 again, B sends a message M_2 to A. A receives it and subsequently fails. Both A and B rollbacks to their respective last checkpoints. B's local state shows that it has already sent M_2. A's local state indicates that it has not been received. The sytem recovers from a consistent state. However, the channel from B to A is empty. Consequently, M_2 is lost.

4.2 Proposed solution

The O.O. agent approach, that is, code and data (refer to part 2), is particularly suited to provide fault-tolerance taking into account the problems mentioned this above. However, to ensure consistency and integrity, an agent has to be modelled as follow :
code: a block of code
data: (a) a data block, (b) a queue of incoming messages and in case the agent was active when the failure occured, (c) its history since last message reception
As the agent classes located on acceptor sites make the agent code always available the code reconstruction problem is directly eliminated. To be able to restart properly, a counter of outgoing messages is also required. A change in an agent's local state occurs if either (1) the agent sends a message *i.e.* the outgoing message counter increases, or (2) the agent receives a message, *i.e.* its incoming queue changes, or (3) the agent completes a task, *i.e.* a new checkpoint is performed.

To be able to recover properly from a sudden failure, two messages are sent together with the original one. A *shadow* of the original message is sent to the shadow of the recipient.

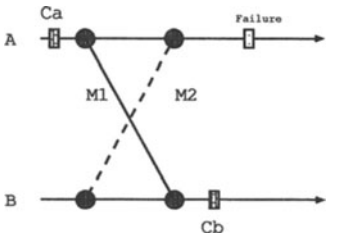

Figure 3 Identification of problems

A message informing the shadow of the sender that the sender has sent a message is also sent so as to increase the message counter. To ensure fault-tolerance, four actors are required. The sender (Agent A), its shadow (Agent A'), the receiver (Agent B) and its shadow (Agent B'). The process is ordered as follow : *(first)* Agent A sends a copy of the message he wants to send to Agent B', *(second)* Agent A sends a message to its shadow A' to increment its message counter and finally *(third)* Agent A sends the message to the receiver B. All transactions are simultaneous and atomic. When these operations are completed, Agent A checkpoints and sends a copy of its state to its shadow. Agent A' then replaces its data with the copy it has received. Agent A also keeps a copy in case it has to regenerate the shadow. In case the shadow has to be generated, it just takes both the checkpointed data block and the queue of incoming messages of the agent and transforms itself into a real agent.

4.3 Implementation

The agent is a mandatory intermediary between the user and the external world. The agent is responsible to update data within its shadow when necessary, that is, after a transaction or after a long enough computation modifying the agent data. As an agent script is processed by a interpreter thread associated to it, the shadow data management can be delegated to the interpreter and to the script object itself.

Two kinds of script object modifications implying a change in the shadow state are identified. The first one is induced by a call of certain script object methods. For instance, the call of the *Start* method involves the creation of the execution data block within the script object. In the same way, it must create a similar block in the script shadow of the shadow of the agent. The *Abort* method, which ends the script running, frees this part both in the agent script and in the agent shadow script. The *PutData* method must set the execution data block of both the agent script and its shadow. The above shadow changes can be taken into account by the code of the method itself. The second kind of modification is due to the interpreter running. It changes the script execution data block values consequently changes the agent state. It must periodically and just after a transaction update the shadow.

5 CONCLUSION

This paper defines a system enabling a mobile user to obtain facilities from an associated assistant : the representation agent. Its structure takes advantage of the object-oriented technology. The object instanciation and behavioral notions enable the creation of secure agents respecting rules such as privacy of the agents located on the same site and acceptor site integrity. This object structure also allows to split agents into two kinds of data : static data and dynamic data. The last one are very sensitive and each update must be taken into account so that even if an agent is lost, it is possible to restore this agent with the same state as before the failure.
Future works include the implementation of a prototype allowing the management of small representation agents responding to the requirements described in this paper.

ACKNOWLEDGEMENTS

The authors wish to express their sincere appreciation to members of their respective laboratory and more especially to Professor Vincent Cordonnier, Pierre Paradinas from RD2P - University of Lille 1, and Professor Nanya from Tokyo Institute of Technology who have made this collaboration possible. The authors also would like to thank Sylvain Lecomte for lively discussions regarding this research.

REFERENCES

[AB94] A. Acharya, B. Badrinath, *"Checkpointing Distributed Applications on Mobile Computers"*, in proceedings of the 3rd International Conference on Parallel and Distributed Information Systems, September 94

[CT95] D. Carlier, P. Trane, *"Security Requirements for Mobile Computing Systems"*, Technical Report of IEICE, FTS 95-70, pp 57-65, Tokyo, December 1995

[FZ94] G.H. Forman, J. Zahorjan, *"The Challenges of Mobile Computing"*, in IEEE Computer, pp 38-47, April 1994

[GC94] S. Gadol, M. Clary, *"Nomadics Tenets - A User's Perspective"*, Sun Microsystems Laboratories Inc. Technical Report, SMLI-TR-94-24, June 1994

[MDC93] B. Marsh, F. Douglis, R. Cáceres, *"System Issues in Mobile Computing"*, Technical Report TR94-020, Matsushita Information Technology Laboratory (Princeton), February 1993

[IV94] T. Imielinski, S. Viswanathan, *"Adaptative Wireless Information Systems"*, in proceedings of SIGDBS Conference, Tokyo, October 1994

[KT87] R. Koo, S. Toueg, *"Checkpointing and Rollback-Recovery for Distributed Systems"*, in proceedings of IEEE Transactions on Software Engineering, Vol. SE-13, No 1, pp 23-31, January 1987

[TC96] P. Trane, D. Carlier, *"Diagnosis Algorithm for Mobility Oriented System"*, in proceedings of the 2nd International Conference on Application-specific Systems, Architectures and Processors, IEEE, Chicago, USA, August 1996 (to appear)