A multi agent system for the concurrent execution of simulation replicas

Miguel A. Mateo; Julio Pons; Roman García, Lenin Lemus
Departamento de Informática de Sistemas y Computadores
Universidad Politécnica de Valencia (UPV), Spain
{mimateo,jpons,roman,lemus}@disca.upv.es

Keywords
Mobile agent, distributed simulation, Java simulation, high performance simulation systems.

ABSTRACT
The design and validation of simulation models are common tasks for many researchers. Frequently, the simulation execution is the phase where more time the researchers spent. In order to improve the response time of simulation execution, a network of workstations (NOW) is commonly used.

Although these systems have proved their usefulness, they do not include mechanisms to easily and transparently distribute the application code to hosts.

In this paper, a distributed multi agent framework for concurrent simulations executions is presented. This framework is MADEX (Mobile Agent for Distributed Execution), which facilitates a) the dynamic management of computing resources, b) the application of various methods for code distribution and c) the design of fault tolerant mechanisms.

MADEX has been developed in Java, which allows using heterogeneous hosts. The framework has been implemented on top of Grasshopper agent system. Although the framework overhead, the tests show that it is not meaningful when the number of concurrent simulations is increased.

INTRODUCTION
There are two main concepts that are relevant to MADEX: a) the concurrent execution of simulations and b) the main characteristics of multi agent systems.

Concurrent Execution of Simulations
Many researchers use discrete event-based simulation in their work. The simulation process has four phases: model creation, simulation implementation, simulation execution and model validation. It seems obvious that model creation and validation are the main tasks of the simulation process.

On one hand, design environments exist that create a model implementation from a high-level representation of the systems. On the other hand, the simulation execution time can be reduced using better machines and mainframes.

Furthermore, NOWs are one emerging and cheaper alternative to mainframes. NOW systems use idle CPU of network-connected workstations to execute distributed and concurrent applications. CONDOR, PVM and MPI are examples of well known NOW software. There are simulation executions systems and techniques that can be used on top of one of this NOW software systems.

One of the techniques used in these systems is replication. This technique consists in the concurrent execution of statistically independent simulation applications. The results of these simulations are combined to obtain the confidence interval of measured variables. The CSX (Perles et al. 1999) is an example of a replica execution system. This tool has been developed in the Department of Computer Engineering (DISCA) of the Technical University of Valencia (UPV), Spain. In addition, environments like CONDOR and PVM are used for the distributed execution of applications (Lemus et al. 1999). The basic components of this kind of system and their relationships are presented in Figure 1.

Figure 1: Replica execution framework components

The different components use the communication services of NOW software to transfer data and control information. The three main components of the system are:

1. Control module. This module initializes the simulation process, starts the simulations in the hosts that compose the NOW and decides when the simulation process ends. To accomplish these tasks, the control module
needs data from the other modules and the NOW software.
2. Local-execution modules. There is one of these modules in each execution node. The objective of this module is to execute and monitor the simulation and to communicate the measures to the control module.
3. Load monitor modules. The NOWs are usually formed by heterogeneous workstations. It is necessary to apply load-balancing techniques to adapt the simulation process to changing capabilities of execution modules.

Although the NOW software spawn the execution through the network, it usually needs that the code to be executed was allocated in the target hosts. When the NOW software does not provide a method to distribute the code, the researcher has to design one. In many cases this is satisfied using a distributed file system. When this solution is not available, the code has to be copied to hosts. In all of these methods, problems with the configuration, the number of hosts and the version of code arise. Moreover, it could be necessary to maintain two different copies of the simulation implementation: one to be executed standalone and one to be executed inside the simulation system.

**Multi Agent System**

A multi agent system is the software system that offers the minimal services that a set of software agents needs to execute and communicate (Ghanea-Hercock 1980). Mobile agents are agents with the ability of changing the host where they are executing. The main differences of mobile agent systems from other approaches of distributed systems are a) the mobility of the agents, b) the proactiveness of agents and execution systems and c) the rich communications mechanism among components.

Major advantages of mobile agents are seen in a) the possibility of reducing global communication costs by moving computation to data, b) to easily distribute computation onto several hosts and c) the increase in system fault tolerant characteristics.

The basic components of multi agent systems, as shown in Figure 2, are:
1. Execution place. The components provide agents the computational resources they need to be executed. These resources include the access to the other agent system services.
2. Communication facilities. The basic communication system used in agent systems is based on messages. These messages are transfer among agents or between an agent and a service or place.
3. Migration service. If an agent has to be transferred from one execution place to other, the migration systems transfer the data and execution state of the agent. This service includes systems to check if the agent code has to be transferred.
4. Naming system. Each component of an agent system has a unique identifier. This identifier or name allows the system to locate agents and services in order to perform communications in a location aware way.

![Figure 2: Multi agent system structure](image)

In this paper, we present MADEX (Mobile Agent Distributed Execution) a multi agent system for the distributed execution of simulation replicas. This system has been tested in the execution of discrete-event simulations. The simulation applications were built with a Java version of SMPL called JSMPL (Mac Dougal 1980). However, the system could be easily adapted to use other Java simulation libraries and tools.

The rest of the paper is organized as follows: Initially, we propose MADEX structure as a basic scheme to create mobile agent based simulation execution systems. The components and their relationships are compared with NOW structure. We then go on to present the implemented system: the tools used, their basic characteristics and the influence of these tools in the final structure. Then, we analyze the results obtained from practical cases. Finally, we present the conclusions obtained from our experience working with MADEX system.

**MADEX**

As mentioned above, the replica execution system is a viable alternative to use a NOW as execution environment of simulations. This approximation consists basically in spawning a fixed quantity of replications in a NOW and let them run until the wished confidence interval is achieved for the measured variables.

One way to solve this problem is to use a multi agent system with support for mobile agents. Whenever a mobile agent changes of host, it needs its code in the destination node. This problem is similar to the distribution of code of NOW systems, but the agent systems resolve the problem automatic and transparently.

**MADEX Structure**

Now, the question is how the service offered by multi-agent system can be used to create a simulation execution system. Our proposal (Figure 3) is based on the interaction
among a set of agents that performs the functions of the basic components of a simulation execution system: simulation execution, data collection, control and monitor of simulations and user interface.

![Multiagent Framework](image)

**Figure 3: MADEX basic components and communications**

Once the agents have been developed it can be used in the execution of different simulation models. The underlying multi-agent system covers the demands on communication, process creation and code distribution. The implementation of these services depends on the selected multi agent system.

MADEX system has several advantages over the traditional execution system:

1. **Transparent and automatic code distribution.** To create a remote execution, an agent is created in the node where the code exists. Once the agent is created it migrates to the node where the execution is desired. As mentioned previously, the agent system migration service takes upon this process.

2. **Easy load balancing.** The implementation of a load balancing system has been simplified. The load balancing can be accomplished by moving, creating, deleting or stopping simulation agents. These tasks are directly supported by multi agent system. All that we need to implement is a module, maybe an agent, that decides when and where it is necessary a load balancing task.

3. **Open system development.** Once a place is created, the agent in the system can migrate to it. Adding a place to an agent system is an easy task that only needs that the host where the place will be located can access the naming service.

4. **Fault tolerance.** If a place or an agent fails, the multi agent system continues working. The entities that try to communicate with the faulty element will receive timeouts or error messages.

Additionally, the use of a Java based agent system facilitates the inclusion of new machines with independence of their characteristics (operating system, computational power, network bandwidth...). Java also simplifies the task of adding web based services to our system.

In our tests, three kinds of agents were implemented: Monitor and Control Agent (MOCOA), Execution Agent (EXAG) and Interface Agents (INA). The MOCOA agent controls and monitors the whole system. The EXAG agents are mobile agents that contain the simulation application to run. The INA agents implement the user interface. If an automatic load balancing system is desired, Load Monitor Agents (LMA) will also be need. Users can perform load-balancing tasks using INA.

Typical execution consists of the following tasks:

1. **MOCOA start.** The first created agent is the MOCOA. The MOCOA needs a set of parameters to start the system. These parameters include the EXAG to run and the simulation model each EXAG will handle.

2. **The MOCOA creates a model of the underlying agent system structure.** This model will be used to decide how many EXAG will be started and where they have to move.

3. **MOCOA creates the EXAG and informs them where to move.** Each EXAG has its own random number generator and simulation model to execute. The EXAG moves to the execution place and starts the simulation.

4. **The MOCOA extracts information from EXAG and, based on the confidence interval achieved, determines if the process has ended or if more simulation execution are needed.**

5. **Once the desired confidence is achieved, the MOCOA writes all the information extracted from EXAG to a file.**

In any moment, the user can access MOCOA and EXAG using specialized user interface agents. These INA agents can be executed from web servers and clients, allowing the researchers to control the simulation through standard web browsers.

MADEX is flexible enough to change this mechanism. Configurations without a MOCOA have been tested during the system development. In this situation, the user is responsible of deciding when to stop the simulation process, taking into account the information collected with a very simple INA.

**ISimulation Interface**

The EXAGs need to control the simulation and to access those variables that are relevant for the researcher. An interface has been designed to allow doing that. We reached an agreement with researchers on what are the main actions they do with simulation applications.

This interface allows starting, stopping and monitoring the simulation. Our EXAGs are constructed to use the ISimulation interface to access its internal simulation. Any
application implementing this interface can be used in the system.

However, if a simulation application implemented in Java is available, it can be used without recoding it. Java patterns design techniques like Adapter or Bridge can be used to build a wrapper class that enables our agents to control that simulation. We use this approach to use existing applications written in JSMPL.

This approach has the following advantage: the simulation application can be built and tested outside the agent system. This means that application designers do not have to worry about the target environment. Another advantage is that multiples implementations of the same model can be used to improve the confidence of results. The only requirement is that all implementations have the same behavior for the ISimulation interface calls.

**IMPLEMENTATION AND RESULTS**

The current implementation of the system is based on Grasshopper 2.2.1 (Grasshopper web page); a multi agent system that can be executed in any Java Virtual Machine that supports JDK 1.3 platform. The different agents have been constructed using these tools. To implement simulation models we used JSMPL. The model implemented is the well-known M/M/1 queue model, with 8 and 10 as time parameters.

Three different machines have been used in the different tests. Each machine has different operating systems and configurations. More over, in the case of Unix-like hosts, the host is shared with other users.

Because Java has a bad reputation as a slow execution environment, the goal of our tests is measuring the agent system overhead and not to compare our system with other simulation execution systems like CSX.

The test that we carry out can be classified in three main areas: a) JSMPL simulation results outside the agent system, b) one machine execution and c) a distributed execution test. The firsts tests try to simulate what a researcher would do if a distributed execution environment were not available.

The test consists in executing the simulation implementation as an independent Java application. The user can set how many independent replications to execute. Each simulation has its own execution thread, all inside the same JVM. This approach emulates the agent system work. The Figure 4 presents the measured response time on three different hosts. The machine labels corresponds with:

- Machine_A. A personal computer with MS Windows 98. Equipped with an AMD K6-2 CPU at 450 MHz, 348 MB RAM. The JVM version is SUN JDK 1.3.1.
- Machine_B. A Linux Linux SuSe 7.2 server. Equipped with two Pentium III at 1GHz, 1GB RAM. The JVM version is SUN JDK 1.3.1.
- Machine C: A Linux Linux SuSe 7.1 server. Equipped with two Pentium II at 350 MHz, 512 MB RAM. The JVM version is SUN JRE 1.3.0.

The simulation stops when the simulation time is almost equal to 1E7 units.

![Figure 4: Response time of N simulations](image)

These results are a bit surprising. The best machine has the slower response times. This is in part due to the Java Virtual Machine scheduling and how it is translated into the native operating system. However, the Machine B has show in next tests its computational power.

The second test consists in evaluating the overhead that agent system introduces. The same simulation application is used and the same number or replication started, but this time inside the multi agent system. The Figure 5 compares the different response times when the number of replicas increases.

The overhead introduced doubles the time needed to execute the same number of replicas without the agent system. This overhead, while being high, it is not as big as with previous prototypes (Mateo et al. 1999). In the worst case, the replicas executes three times slower in agent system. This means that the system will useful if we use at least four machines like Mach A.

The worst machine is the fastest. Now we can conclude that implementation of Linux Java machines are not well suited for our system (at least using Grasshopper).

The last test is the more important. In this test, we simulate the real system. Once simulation execution is started, the MOCAO executes an EXAG for each execution place. The MOCAO periodically monitors the EXAG and measures one of the system variables, the machine clock time (in milliseconds) and the simulation time (the measure unit depends on the model).
For this test, the simulation code only resides in the machine where the simulation process was started. This implies that the code has to be copied on remote machines before they started its execution.

The tests show that MADEX is a tool to consider although it introduces overhead. The main benefits of MADEX are: a) a transparent and automatic code distribution, b) an easy mechanism to adapt existing Java simulation, c) configuration flexibility, d) easy to made load balancing and e) fault tolerance.

The system has fault-tolerant characteristics based on its agent system characteristics. Only if the host running the naming service fails the whole system comes down. The system tolerates the individual fail of agents and places. If this occurs the damaged entity is not available any more, but the rest of the system continues its work.

**CONCLUSIONS**

In this paper, the MADEX system is presented. The mobile agent paradigm is used in order to distribute the processes in a network of workstations. The agent system is responsible of the execution of agents, and it provides the communication and the code distribution infrastructures.

Once a set of execution agents is developed, it can be used with any simulation that implements a specific interface. More over, it is not necessary to maintain different versions of simulation application, one for use inside the system and one independent to rapid and individual tests.

The ISimulation interface and the use of well-known Java patterns, allow the rapid conversion of existing Java simulation application to allow it to be executed with MADEX infrastructure.

The tests show that MADEX is a tool to consider although it introduces overhead. The main benefits of MADEX are: a) a transparent and automatic code distribution, b) an easy mechanism to adapt existing Java simulation, c) configuration flexibility, d) easy to made load balancing and e) fault tolerance.

The system has fault-tolerant characteristics based on its agent system characteristics. Only if the host running the naming service fails the whole system comes down. The system tolerates the individual fail of agents and places. If this occurs the damaged entity is not available any more, but the rest of the system continues its work.

**FUTURE WORK**

The obtained results encourage us to continue with the use of agent system to support distributed executions. As far as the tests showed that system overhead is higher than expected, we plan to test other systems and, if necessary, implement our own agent system.

It is also possible that the bad response times were due to Java use. Consequently, other non-Java based system will be evaluated.

The INA and MOCOA used in our work are very simple. More agents have to be developed. These new agents will be used with more complicated simulations, even models used in current research.

The use of JSMPL has simplified our tests because we are the its designers, so we can change it if we need it. We plan to test the system using other simulation tools like CSIM. Finally, our goal is to design a complete graphical design and execution environment. This environment will use a graphical interface to help researchers in developing simulations. The models created in this environment will be executed in MADEX system.

**References**


