A High Performance Core for OSI Management Agents: Implementation, Simulation and Performance Evaluation

ABSTRACT

This article presents a proposal of a *multithreaded* core for an OSI/ISO management agents including its modeling, simulation and performance evaluation. The proposed model is compared against another one (*unthreaded* core) which is the commonly adopted model by the most of the network management systems. All the modeling and simulation process were done at the agent core level. Therefore the performance evaluation do not consider any effect of the communication layers, dealing with the performance of the activities related to the core of the agents' applications.

Keywords

OSI Network Management, Telecommunication Management Network, Agent Core Simulation, Multithreading, Active Objects, Performance Evaluation.

1 - Introduction

The goals of network management are immutable, however, the process by which these goals are reached have changed in a manner that makes use of the current technological advances. As new technologies and services are being introduced, network management systems must keep up with new innovations, executing monitoring in real time and offering support for the execution of automatic corrective actions, based on the events and on the current state of the network [2].

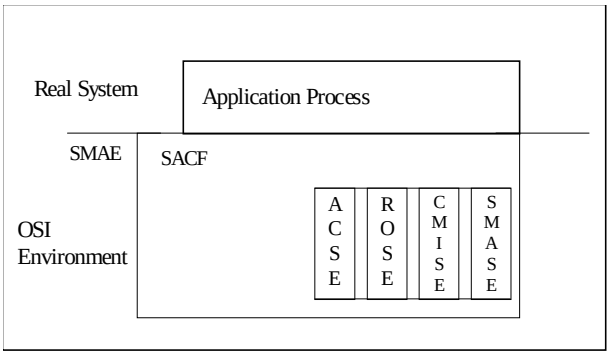
Currently, due to advances in operating systems, program languages and environments that support the development of applications, the implementation of *multithreaded* applications have become frequent in various fields (SGDBs, GUIs, Web Servers, etc.).

This article presents a *multithreaded* core for the implementation of OSI management agents, which are part of a larger project that seeks the definition and implementation of a network management platform.

With the goal of offering a basic core to the agent processes in the platform, the proposed model will allow managed objects to be implemented as active objects, given their *multithreaded* characteristics. This core offers the agent processes capabilities that provide better performance and security when in the support of the execution of its managed objects, and in the performance of intra-agent activities (Management Information Tree (MIT) maintenance, dissemination of notifications, etc.).

2 - Agent Applications

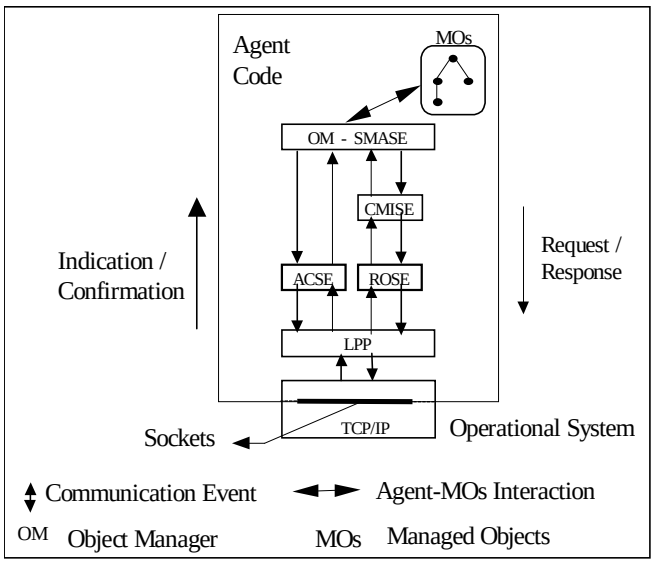
The standardization documents in the field of network management do not define the internal organization of the management processes, which is handled locally in each implementation. Conceptually, management application processes are those applications that utilize the services provided by the service element of the Systems Management Application Element. (SMAE). This concept, defined in [3], can be illustrated in Figure 1.



- the service elements ACSE[6], ROSE[6] and CMISE[7];
- the protocol LPP[6];

In the approach adopted for the implementation of the agent processes, when loaded in memory, the agent carries in its code all of the functions needed for its execution. Figure 2 presents the structure of the agent processes in the platform.

Figure 2: Structure of the agent processes



3 - Agents and Managed Objects

Each process agent is set to contain all of the definitions of the classes of managed objects; this will allow the definitions to be instantiated at the time of execution. The implementation of the managed objects, was basically conducted representing them as instances of classes of C ++ Language.

In order to denominate their managed objects, the agent processes implement an internal structure for the representation of MIT [8]. This structure allows the representation of objects to follow the *containment* hierarchy concept, defined in the OSI management model. The implementation of MIT is not included in the implementation of the classes of the managed objects, this being an independent implementation, which offers greater flexibility for their maintenance (creation, consultation, deletion, etc.).

A support object called OM (*Object Manager*), was defined to conduct the MIT management and the maintenance of the set of instances of classes of managed objects supported by the agent. A more detailed description of the behavior of OM will be presented in section 4.3.

4 - The *Multithreaded* Core

As indicated above, managed objects are an integral part of the code of the agent processes. Therefore, agent processes must support the basic requirements for the execution and maintenance of these objects. The *multithreaded* core that will support the execution of the managed objects, takes into account two basic aspects:

- Features of the handling of management operations and the issuing of notifications;
- Dynamics (behavior) of the managed objects.

Concerning the handling of the management operations and the issuing of notifications, a serialization policy of these operations was defined. The servicing of the operations requested by the manager is done in a serialized manner, in accord with the order of the arrival of these operations (FCFS - *First-Come-First-Served*).

As the *multithreaded* core is inserted in the structure presented in Figure 2, the serialization of the management operations takes place at the moment that PDU's of ACSE/CMISE arrive. At this point, the organization of the OM input structure is in charge of grouping the operations that arrive, according to the FIFO (*first-in-first-out*) policy. The OM support object has the goal of providing MIT maintenance, distributing the operations that arrive from the manager to their respective managed objects, conduct the support operations for *multithreading*, implement access control to the MIB objects, and other functions of infrastructure management.

Concerning the treatment of operations issued by the manager, some form of structuring process agents (servers) were studied, being that one of the most utilized to enhance the performance of these processes, is similar to the implementation of a RPC *multithreading* server [9]. Within this approach, for each operation which arrives a *thread* is created to execute this operation. This type of organization was not adopted, given that the serialization of the operations is not guaranteed, and in the cases of the operations conducted on the same object, the CMIP protocol [10] requires the serialization of these operations, contrary to the non-synchronized operations on different objects, which can be executed without any need for serialization, when it comes to the order in which operations are issued by the manger.

An example that invalidates the utilization of the approach adopted for the RPC *multithreading* server, can be the execution of non-serialized operations on the same managed object. A manager requesting a SET (*Replace-value*) operation on the attributes of various managed objects, utilizing scope and filter, then issues a GET operation to retrieve the value of an attribute that was altered in one of the managed objects affected by the SET operation. The *thread* created for the SET operation can lose the processor before its conclusion, allowing the *thread* of the GET operation to pick up the processor and possibly return the value of the attribute, which was still not altered by the SET operation, creating an error situation.

Due to the demands of the serialization of the CMIP protocol, it was considered that the management operations are treated sequentially, both by the OM as well as by the managed objects, given that both types of objects have operation queues that follow a FIFO policy. Figure 3 shows the model proposed for the multithreaded core.

After messages arrive in its input structure, the OM does not need to be invoked, for it possesses its own control *thread* and is constantly consuming messages in its queue. When there are no messages to be consumed, the OM continues to execute parts of its behavior (change of priority between the *threads*, synchronization of operations, etc.), relative to the management of various managed objects.

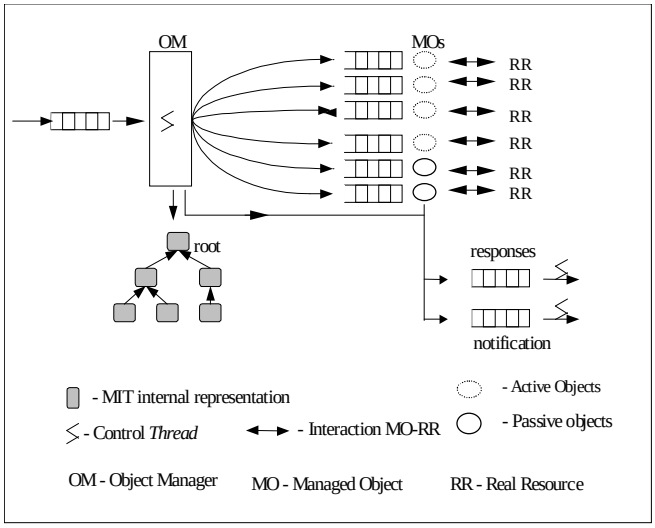


Figure 3

Each object that possesses its own control *thread* and operations queue, allows its execution to be conducted independently and concurrently with other activities of the agent, enhancing the performance of this process. To maintain independence and concurrence between the managed objects, the system adopts the concept of active objects (section 4.1) for the implementation of the managed objects.

Figure 4 presents the location of the multi-threaded infrastructure in the code of the process agent, according to the structure presented in Figure 2.2. In section 5, an evaluation of the performance of the two agent core implementations will be presented, one utilizing active objects and another that implements passive objects.

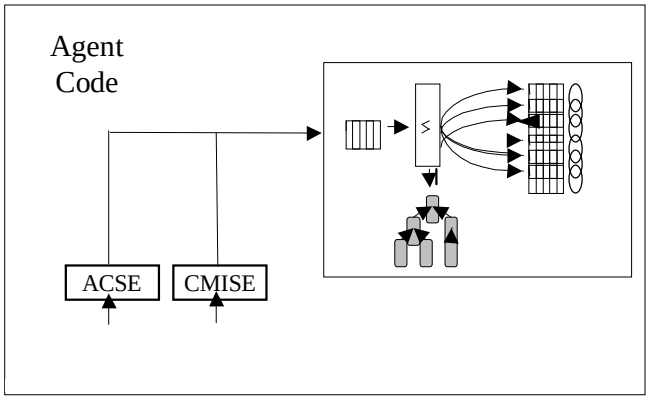


Figure 4

4.1 - Active Objects

In the object-oriented paradigm, we can classify the execution states of the objects as either **Active** or **Passive** [11]. According to Booch [12], OOP concurrence is the property that distinguishes an active object from a non-active (passive) one. In the presence of concurrence, it is appropriate that the objects are viewed as autonomous active entities, which execute their behavior independently from the external invocations [13].

Utilizing these concepts, it can be said that active objects are those that encapsulate their own control *thread*, in order to have autonomy to change their internal state without the need for external invocations. Passive objects do not encapsulate their own control *thread*, and can only change their state in the presence of an external invocation, which can come both from another active object or from the control *thread*, of the process of which the passive object is a part. After being invoked, a passive object becomes active, conducting some computation (its behavior) and returns to the passive state.

Some object-oriented languages [14], offer support for the implementation of active objects, however C++ language, which is still being utilized in the implementation of the platform, only implements the concept of passive objects. Due to this limitation, threads that are associated with C ++ language were used, in order to resolve this constraint and permit the active object concept to be implemented.

4.2 Objects Managed as Active Objects

With the goal of better attend the requirements of management, the utilization of active objects to implement managed objects was adopted, given that the representation of real resources through the active objects is better suited to the abstraction of system resources (layer entities, carrier connections, hubs, etc.).

In traditional passive objects approaches, are limited to the implementation of such abstractions, due to the lack of dynamics and independence between the execution of objects. Problems with objects (for example: a failure of interaction with a real resource), compromise the execution of other objects, because the control *thread* of the process is dedicated to the execution of the object that failed, causing the blocking of the entire agent process. To resolve this problem, some platforms limit the implementation of the objects, or even execute a more complex computation, because blockages in external events are not permitted. An example of these platforms is the OSIMIS management platforms [15], which define that implementations of managed objects cannot be blocked in external communication points, causing the blockage of the entire process agent.

Restrictions such as this make the implementation of the behavior of objects more complex, given that the entire process agent can be compromised by a system error in one of its objects. This complexity can be increased, when there are managed objects representing system resources which are weakly coupled, demanding greater complexity in the communication between the real resource and the managed object.

By implementing managed objects as active objects, the object that became blocked does not affect the other objects or the activities of the process agent itself, given that its individual *threads* are free to execute their computations independently from the object that became blocked.

Active objects reflect with greater fidelity the principal function of the managed objects; to represent real resources [16].

The proposed core (Figure 3) does not impede the utilization of passive objects, because in some cases managed objects have a static behavior and will be better implemented as passive objects. An example of managed object implemented as non-active entities, is the *log* class objects, which only conduct their functions through external invocations (writing, reading, etc.). In these cases, it is desirable that the object be implemented as passive, because its constant activity, will only consume processing resources (CPU) unnecessarily.

As said earlier, passive objects can become blocked

and compromise the operation of other passive objects or the control *thread* that invoked it. To do this, an implementation of passive objects is proposed, which ensures that a blockage in the functioning of its behavior does not interfere with the execution of other activities in the agent process. This implementation was possible, thanks to the *multithreaded* features of the proposed core and will be considered in section 4.5.

4.3 - Dynamics of the Objects

The concurrent execution of the managed objects, offers the system manager less response time in relation to the requested operations, and a more precise vision of the MIB, concerning the states of the managed objects and the states of the resources that they represent. [17]. Basically, managed objects can obtain information from their resources, accessing them in three ways:

- through a request from the manager (access on demand);
- through periodic *polling*;
- by receiving *traps* from resources with built-in management functions.

In the first case, traditional *single-threaded* organizations would cause a temporary blocking of the entire agent application, because the object is interacting with a resource at that given instant. In case this resource was weakly coupled to the system, various later requests coming from the manager could remain pending for a long time, waiting for the completion of communication between the managed object and the resource. By utilizing the *multithreaded* core, all of the agent activities are independent, offering greater fairness in their executions and not allowing active computations to monopolize the utilization of the processor at any given moment.

In the second case of interaction (access by *polling*), traditional, as well as *multithreaded* organizations offer desirable support for their implementation. Nevertheless, in the *multithreaded* approach, there is a capacity to offer certain managed objects greater priorities in relation to their executions, causing a greater frequency of their polling operations. This is desirable because greater precision is needed for them to support real time features. In application acting in fault management, this feature is desirable in order to provide more precision in the detection of problems that may occur with certain resources that are vital for the perfect functioning of the system.

In the third interaction (*trap* receptions), similar to the earlier case, certain resources that demand special attention (for ex. access control service), can be privileged in having their notifications given priority over other managed objects that demand less attention in relation to their notifications.

The function of altering priorities between the managed objects is delegated to the OM, which interferes directly in the priorities of the set of *threads* utilized by the managed objects. The designer of the application agent will be in charge of defining which priorities will be assigned to each object.

In order to improve the performance of the process agent, objects can pass their turn of execution to other active entities (*threads*) of the system, verifying that at a given instant, their behavior would not perform any computation. Empty queues, attributes of the *disabled* operational state, and managed resources which are temporarily unavailable, are some examples of conditions in which objects pass the processor on to other activities, in order not to consume their entire share of execution time without any computation.

The serialization of the operations takes place at the operations level on the same object. Requests that utilize the scope, filter and synchronization, must previously be handled by the OM, because in addition to managing the structure that represents the MIT (*scope*, *filter*), this supplies mechanisms for the implementation of synchronization between objects, among other functions.

In the model presented by Figure 3, a *thread* is dedicated to the execution of the OM, and two others for each of the response and notification queues, which are dedicated to the issue of these messages. It is the function of the OM to perform the maintenance (create, delete, suspend, reactivate, alter priorities, etc.) of these *threads* and the *threads* of each managed object.

In the creation of an object, the OM associates a control thread to the created object, updates the MIT, initializes the values of the attributes of this object with some reference object (if specified in the *Name Binding* [18]), creates an operations queue for the object, issues an event report notifying the creation, and updates its internal state in order to support the new object created. The *thread* associated to the new object will execute the "*behavior*" method of this object, being that part of this behavior is defined by the platform and the rest will be that defined in the constructor of the managed object class *template* [18], which will be implemented by the application designer. Part of the behavior of the object, supplied by the platform, is related to the handling of the operations queue of the managed object and to interactions with synchronization objects (*S*). This part of the implementation is called "active objects support code". The functioning of the object (*S*) will be described in section 4.4.

4.4 Operations that Demand Synchronization

Each object that has its own operations queue and that executes in an independent and concurrent manner with other system objects, creates a problem in situations where an operation must be executed upon multiple objects (*scope*) and this requests atomic synchronization. The CMIP protocol allows two forms of synchronization:

- *atomic*;
- *best effort*.

In the *best effort* synchronization, a failure in the execution of an operation by an object in the group does not interfere in the execution of the other objects. The *atomic* synchronization demands that all of the objects of the group successfully execute the operation; in case some object fails, all of the objects involved, abort the operation, even if they are capable of executing them. The problem with this type of operation can be illustrated in the following example.

Suppose that for a SET management operation (S), requested with scope and filter, three objects were selected, and that this operation was requested with *atomic* synchronization. Figure 5 illustrates the setting of the problem.

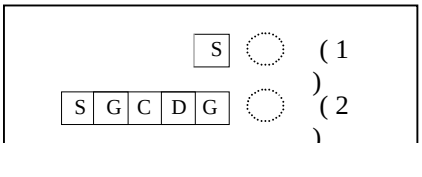


Figure 5

As object (1) does not possess operation messages in its queue, the SET operation will be the only one in the operations queue of this object. In the case of object (2), there are remaining operations, and SET will be the fifth operation of the queue. The third object has only two operations in its queue, and the SET operation will be the third to be attended. In this scenario, it is found that the problem of operation (S) can be executed by the objects at different times, given that the operation queue of each object has different sizes. To resolve the problem an application support object called synchronizer object (S), will coordinate operations that demand atomic synchronism. Figure 6 demonstrates the solution.

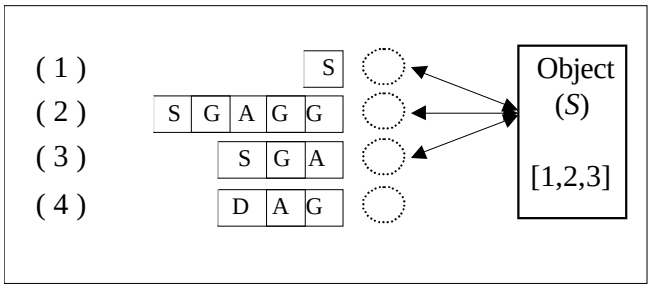


Figure 6

When operations that demand *atomic* synchronism are submitted to the managed objects, the OM will create an object (S) which will contain the object identifiers (OID) that are in the operation group. In this example, only objects 1,2 and 3 are submitted to atomic synchronization, given that their identifiers are contained in the object (S). When the managed object removes an operation from its queue which demands synchronization with other objects, this will notify its object (S), that it is ready to execute the operation. From this moment the managed object waits for a message from the object (S), requesting the operation to be executed. This message only will be issued by the object (S) when all of the objects belonging to the group are ready to execute the operation. Thus, new operations that arrive in the object queue, are not handled, until it is released by the object (S). Other activities can be executed by the managed object, in order to keep the real resource's image always updated.

After the object (S) receives from all of the objects of the group notifications that they are ready, (S) sends a message to all of the objects to execute the operation. After they execute the operation, the managed objects must send a response to object (S) indicating that the operation was successfully realized or that it failed, because in the execution of the operation all of the managed resources (real resources), problems can arise in the interaction between the managed object and resource. In case a failure response is notified, the transaction is canceled for all of the objects of the group, which once they return to their earlier state, are released to continue to serve their operation queues.

4.5 - Non-Blocking Passive Objects

In order to allow that the implementation of some managed objects (*log*, *log* register, etc.), be modeled as passive objects, does not compromise other activities of the system in case these are blocked, it was defined that each passive object has its own control *thread*. These objects only become active in the presence of external invocations, preserving the semantics of passive objects.

In this implementation, the OM does not directly access the passive object method, because if the method is blocked for some reason, the OM *thread* will be blocked and will not handle the new PDU's that arrive from the CMISE/ACSE. Therefore, the OM requests that the passive object *thread*, invoke its methods to attend the queue operations, thus not affecting the execution of other agent activities, in case the passive object is blocked. Passive objects execute their behavior by invoking the OM, which is represented by the stepping up (*sema_post()*) of a semaphore variable (OM_Signal), utilized by the passive managed object, differently from the active objects that are constantly executing their behavior.

5 - Modeling, Simulation and Performance Evaluation

This section presents the results of a performance comparison (benchmarking) of the two agent core models. One model adopts the implementation of the managed objects as passive objects, this being the form adopted by the majority of the management platforms implemented today. In the second model, the implementation of the managed objects follows the approach of the active objects, which is proposed in this paper.

The performance evaluation of the management agents requires the construction of models, and their simulation, which offers results concerning the behavior of the variables that measure their performance. Such models, when exposed to artificial loads, analogous to those in the real world, allow the comparison of the performance of the systems being investigated. The main components of the models are:

- Sources of emission of management operations (manager)
- Managed system (agent)
  - Managed Object (MO)
  - Object Manager (OM)
  - Event Report (Notifications)

5.1 - Parameters Utilized

The parameters utilized for the modeling of the above components, were obtained from the measurements conducted on prototypes, which were implemented for each of the evaluated models. The management operations (requests) are responsible for 85% of the traffic of the entities to be processed by the agent. The remaining 15% are related to the notifications generated by the managed objects. It is assumed that the arrival of both requests and notifications adhere to a Poisson process. In this way, to model the processes of notifications and request's generation, exponential distributions were utilized.

For the OM processing times, according to the measurements conducted in the two implementations, values were adopted of 0.073 and 0.075 seconds, respectively, for the Passive Object and the Active Object models.

In the implementation of both of the prototypes, it was defined that the processing time of the MO's has a duration of 2 seconds. The measurements of both models, resulted in the total time of 2.0094 seconds for the MO's implemented in a passive form, and 2.0508 for the MO's implemented as active objects.

5.2 - Design of the Experiments

The performance of the modeled systems depends,

fundamentally, on four principal factors: the number of managers, the number of objects, the forms of interaction between the agent and the manager and the level of the system load. Each of these factors can assume two possible levels, thus characterizing a complete factorial type experiments design [19], with the number of experiments equal to 2^4. The four factors and their respective levels are presented in Table 1.

Factor	Level 1 (+)	Level 2 (-)
Managers	5	1
Objects	30	10
WorkLoad	2,2 sec (act obj)	1,1 sec. (act obj.)
	12,44 sec (pass obj)	24,88 sec. (pass obj)
Interaction	Asynchronous	Synchronous

Table1

As can be seen in the above table, the workload allocation, within a single level, is differentiated, depending on the model adopted. The values utilized reflect a compatibility between the bottleneck points distinguished in the two models, OM in the model of active objects and MO in the model of passive objects. In this way, to characterize the models in situations of high and low rates of processing of their critical activities, the parameters shown in Table 1 were adopted. To evaluate the performance of the modeled systems, the following response variables were adopted:

- Total time spent by a request in the system;
- Number of requests attended in both models;
- Total number of notifications accounted for.

5.3 - Results of the Simulations

The results presented here were obtained from the simulations of the models of the systems (active and passive). The models were developed using the simulation environment ARENA 2.0 [20]. For a statistical analysis of the results of the simulations the ARENA *Output Analyzer* was utilized. The results presented come from the value obtained from 10 replications of each of the experiments. This number proved to be sufficient for the measurements obtained to have the acceptable confidence interval at an α level of 0.05. The simulations were conducted considering intervals of 1,000 seconds. Since they were non-terminal systems, their analyses must be conducted within the steady-state of the systems. For that purpose, after proper statistical procedures, the adoption of a warm-up period equivalent to 10% of the total time of each replication was considered. In this way, periods of 1,100 seconds were simulated, where the observations referring to the initial 100 seconds were ruled out.

5.3.1 - Analysis of the Results of the Passive Object Model

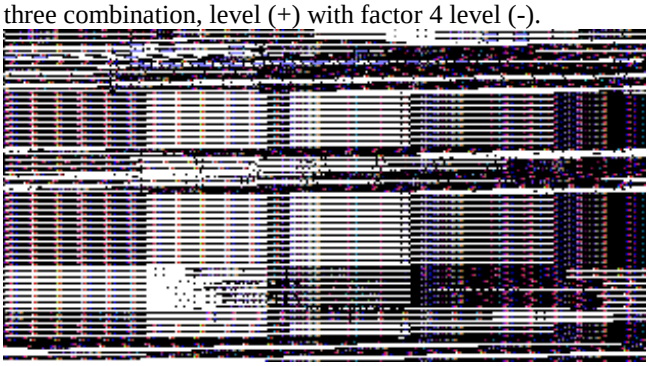
The results of the simulations, for the passive objects model, is found in Table 2.

	Factors/Levels						
Experim.	1	2	3	4	NRA	NNA	TTR
1	+	-	+	+	**383**	29	2.11
2	+	+	+	+	**397**	**75**	2.18
3	+	+	-	+	223	59	2.37
4	+	-	+	-	**430**	27	13.63
5	+	+	+	-	**403**	**69**	**64.62**
6	+	+	-	-	223	59	4.5
7	-	+	+	+	70	**67**	2.11
8	-	+	+	-	70	**67**	2.23
9	+	-	-	-	206	39	3.03
10	+	-	-	+	183	26	2.11
11	-	-	-	-	31	24	2.26
12	-	-	-	+	31	24	2.11
13	-	-	+	-	70	18	2.26
14	-	-	+	+	70	18	2.11
15	-	+	-	+	104	40	2.11
16	-	+	-	-	104	40	2.16

Table 2

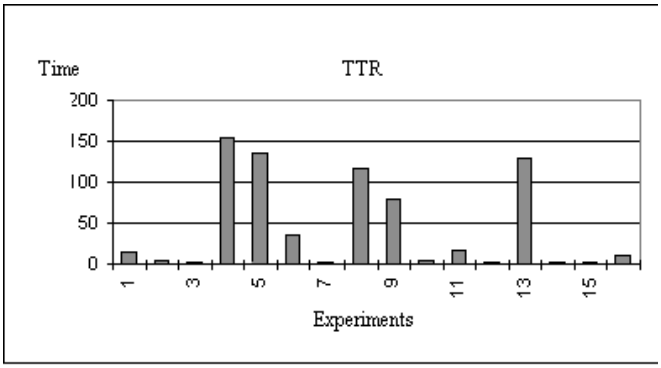
For each of the experiments the values adopted for each factor are presented, as well as the results obtained by the response variables. In the table, NRA refers to the number of requests attended by the OM, NNA is the number of notifications attended by the OM, and TTR is the total time used by a request in the system.

Considering the observations in Table 2 and Graph 1, it is found that in experiment 5, there is a considerable increase (64.62 seconds) in the response time. This increase is justified by the levels adopted by the various factors. The response time to a request for the synchronous passive object model, is dependent on the wait time for the execution by the system of the other requests in transit. In this case, due to the high level of requests generated by the five managers, because of the large number of notifications generated by the 30 objects and by the form of interaction between agent and manager (synchronous form) the number of operations waiting to be attended by the OM reaches it maximum point, causing a high response time for the operation. In the other cases, the system presents itself as less congested, principally due to the absence of the factor three combination, level (+) with factor 4 level (-).



Graph 1

Concerning the number of requests attended, it is observed in Table 2 and in graph 2, that tests 1,2,4 and 5



present respective maximums of (383, 397, 430 and 403). These values are mainly due to the high number of requests generated by the 5 managers with high rates of creation of operations (factors 1 and 3 (+)). However, it is also observed that especially in experiment 5, due to the large number (69) of

notifications generated by the 30 objects, the processing time of the operations is also the highest of all of the experiments.

Graph 2

5.3.2 - Analysis of the Results of the Active Object Model

The results of the simulations, for the active object model, are found in Table 3.

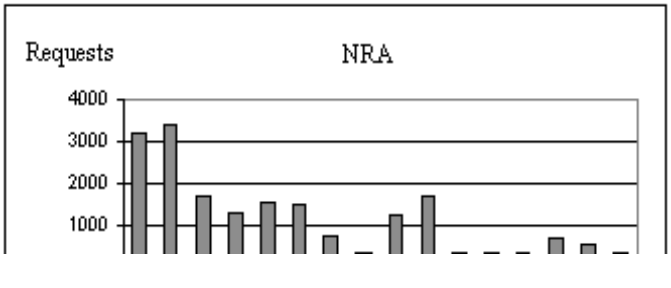
	Factors/Levels						
Experim	**1**	**2**	**3**	**4**	**NRA**	**NNA**	**TTR**
1	+	-	+	+	**3219**	34	15.21
2	+	+	+	+	**3420**	97	3.13
3	+	+	-	+	1656	98	2.57
4	+	-	+	-	1283	34	**157.79**
5	+	+	+	-	1529	97	**135.96**
6	+	+	-	-	1512	98	34.849
7	-	+	+	+	699	97	**2.39**
8	-	+	+	-	344	97	117.42
9	+	-	-	-	1220	102	79.39
10	+	-	-	+	1677	37	3.79
11	-	-	-	-	341	35	16.041
12	-	-	-	+	350	35	**2.41**
13	-	-	+	-	342	32	128.19
14	-	-	+	+	683	32	2.70
15	-	+	-	+	489	**107**	**2.28**
16	-	+	-	-	321	**107**	9.67

Table 3

For each of the experiments values adopted for each factor are presented, as well as the results obtained by the response variable. In the table, NRA refers to the number of requests attended by the OM, NNA is the number of notifications attended by the OM and TTR, the total time spent by a request in the system.

Observing Table 3 and Graphs 3 and 4, it is recognized that in general, the average number of requests attended grows in relation to the earlier model (passive). Examining the results more accurately it can be recognized which factors most influence the general performance of the system, considering the combination of response variables NRA and TTR. In this sense, experiment 2 (NRA = 3,420 requests and TTR = 3.13 seconds) can be considered to have the best performance. In this experiment, there is a system with the maximum utilization rate of the MO's (due to the levels adopted in various factors) and acting in a asynchronous form.

Graph 3



Graph 4

6 - Conclusions

The *multithreaded* core presented in this paper allows the introduction of the concept of active objects in the implementation of managed objects. One advantage of this core over the implementations of cores that utilize the *single-thread* approach is the great independence between the executions of the behavior of the managed objects, which offers a higher degree of fault-tolerance for the process as a whole, protecting it from faults that can occur in the scope of an object and which interfere in other activities of the agent. The best performance in the execution of the intra-agent activities is another important factor present in the *multithreaded* proposal. The results achieved from the simulated models prove that this model has the best performance when compared with the *traditional* model that implements passive objects.

In multiprocessed machines, the nature of the intra-agent activities allows the parallel execution of the various control *threads* causing a high performance level of its executions.

This (core) is currently implemented as part of an *Agent Toolkit*, the goal of which is to automate the development process of the agent processes, with the user only responsible for the implementation of the *behavior* of the managed objects. This *Toolkit* is part of a larger project, which seeks the implementation of an OSI network management platform. The implementation of the platform is being developed for Solaris 2 (SunOS 5.5). Various resources (mutex, semaphore, LWPs, etc.) related to this environment are being utilized. The entire handling of the *threads* is being conducted with a *pthreads library* [21] in order to make the implementation compatible with the POSIX P1003.4a. standard. Future papers will seek the integration of this core into agents that utilize XOM/XMP interfaces given the widespread utilization of these API's in commercial management platforms.

References

[1] ISO/IEC 10040, Information Technology - Open Systems Interconnection - Systems management overview, 1992.

[2] Bean, A.; Wood, D.; Fairclough, W. "Specifying Goal-Oriented Network Management Systems", IEEE Communications Magazine, 1993.

[3] ISO/IEC 7498-4, Information Technology - Open Systems Interconnection - Management framework, 1992.

[4] Mansouri-Samani, M.; Sloman, M. "Monitoring Distributed Systems (A Survey)", Imperial College Research Report N°. DOC92/23, 1992.

[5] Bach, M. J. "The Design of the UNIX Operating System", 1990.

[6] Rose, M. T. "The Open Book: A practical perspective on OSI", Prentice-Hall, 1990.

[7] ISO/IEC 9595, Information Technology - Open Systems Interconnection - Common management information service definition, 1991.

[8] ISO/IEC 10165-1, Information Technology - Open Systems Interconnection - Structure of management information: Management information model, 1992.

[9] Tanenbaum, A. S. "Distributed Operating Systems", Prentice-Hall, 1995.

[10] ISO/IEC 9596, Information Technology - Open Systems Interconnection - Common management information protocol - part 1: Specification, 1991.
[11] Tsichritzis, D.; Nierstrasz, O.; Gibbs, S. "Beyond Objects: Objects ", IJICIS, vol. 1 no. 1, pp. 43-60, 1992.
[12] Booch, G. "Object-Oriented Analysis and Design with Applications - Second Edition", The Benjamin/Cummings Publishing Company, 1994.
[13] Löhr, K. "Concurrency Annotations", OOPSLA'92, pp 327-340, 1992.
[14] Takashio, K.; Tokoro, M. "DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems", OOPSLA'92, pp. 276-294, 1992.
[15] Pavlou, G.; McCarthy, K.; Bhatti, S.; Knight. G. "The OSIMIS Platform: Making OSI Management Simple",1994.
[16] Matias Jr., R.; Specialski, E. S. "Managed Objects as Active Objects: A Multithreaded Approach.", IS&N'97 (4th International Conference on Intelligence in Services & Networks), Maio/1997, Como, Italia.
[17] Matias Jr., R.; Specialski, E. S. "A Multithreaded Core for Network Management Agents", ISCC'97 (IEEE Symposium on Computers and Communications), Julho/1997, Alexandria, Egito.
[18] I SO/IEC 10165-4, Information Technology - Open Systems Interconnection - Structure of management information: Guidelines for the definition of managed objects, 1992.
[19] JAIN, R. 1991. *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, Inc.
[20] PEGDEN, C.D., Shannon, R.E. Sadowski, P.P. 1995. *Introduction to Simulation Using SIMAN*, 2nd Ed, McGrall-Hill
[21] SunSoft "Pthreads and Solaris threads: A comparison of two user level threads APIs", Sun Microsystems, 1994.