Analysis of embedded CORBA middleware performance on urban distributed transportation equipments

S.L. Toral *, F. Barrero, F. Cortés, D. Gregor

Departamento de Ingeniería Electrónica, Universidad de Sevilla, Avda. Camino de los Descurbimientos s/n, 41092, Seville, Spain

A B S T R A C T

The increasing number of ITS (Intelligent Transportation Systems) equipment spread across cities offers tremendous possibilities in the development of distributed smart environments. A middleware layer located between the operating system and the final application can be used for the communication among the equipment to spontaneously act and cooperate among themselves. However, this middleware layer has also a computational cost that should be quantified as it can affect the main application. This paper defines a methodology for such quantification using as case example a modern ITS equipment related to vehicle tracking using artificial vision. Experimental results illustrate the proposed methodology.

© 2012 Published by Elsevier B.V.

1. Introduction

Today, a public infrastructure is distributed through the city and consists of hundreds of cameras for surveillance purposes, traffic light regulators, panels and systems for traffic parameters estimation, usually connected to an urban data network, capable of disseminating comprehensive information, including real-time data and video [1]. Transportation equipments have quickly evolved during the last years as a result of the continuing progress of embedded devices and multimedia processors. Modern Intelligent Transportation Systems (ITS) equipments are based on embedded processors able to run complex computational tasks and provided with multithreading and networking capabilities. Consequently, they can expand their typical functionality to offer distributed services both to the traffic control centers and to citizens [2]. The final aim consists of transforming urban environments into smart urban environments, that is, "a small world where all kinds of smart devices are continuously working to make inhabitants' lives more comfortable" [1]. Smart cameras constitute today a clear example of this innovation. They are equipped with a high-performance onboard computing and communication infrastructure, combining video sensing, processing, and communications in a single embedded device [4,3]. Consequently, they can make use of urban data networks and, by including software for distributed services, they can potentially support more complex and challenging applications, such as smart rooms, surveillance, tracking, and motion analysis [5].

Middleware is system's software that resides between the applications and the underlying operating systems and networks to provide reusable services that can be composed, configured, and deployed to create distributed real-time and embedded (DRE) applications rapidly and robustly [6]. Middleware provides critical capabilities for such distributed systems like encapsulating native OS communications and concurrency mechanisms, avoiding hardware and software dependencies or allowing the management of distributed services. However, middleware has also several drawbacks like resource consumption. Evaluation of these middleware technologies has often reached the conclusion that middleware is a large and complex system, requiring careful coupling to, and tuning for, the specified application domain. In this sense, The Middleware Technology Evaluation (MTE) project has defined a procedure for evaluating middleware components and technologies [7]. They consider a laboratory-based generic product evaluation and they evaluate load-balancing mechanism for client requests as one of the key middleware components. The aim of this paper consists of defining a method for analyzing and quantifying embedded middleware performance in terms of transactions processed per second. The obtained results have important implications on the performance of the main functionality of embedded processors. The outline of this paper is as follows: Section 2 will provide an overview about embedded middleware. Then, a generic method for the measurement of embedded middleware performance is proposed. Section 4 describes the implementation of the embedded middleware layer responsible of the efficient deployment of distributed services and its application to urban scenarios. Two open source middleware software have been chosen as the layer between the operating system and the final applications. The proposed methodology is applied to this case study in Section 5,

* Corresponding author at: Dept. Ingeniería Electrónica, E. S. Ingenieros, Avda. Camino de los Descurbimientos s/n, 41092, Seville, Spain. Tel.: +34 954481293; fax: +34 954487373.

E-mail address: toral@esi.us.es (S.L. Toral).

obtaining experimental results about middleware performance. Conclusions are finally detailed in Section 6.

2. Embedded middleware

During the last years, CORBA has become more and more popular as a reference middleware for distributed computation [8,9]. CORBA is a framework of standards and concepts for open systems defined by the OMG [10]. In this architecture, methods of remote objects can be invoked transparently in a distributed and heterogeneous environment through an ORB (Object Request Broker). The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request [11,12].

CORBA can be considered as an open distributed object computing infrastructure with the objective of automating many common network programming tasks such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshalling and un-marshalling; and operation dispatching [13]. One of the most important advantages of CORBA is its ability to support heterogeneous environments, different vendors' products, and several popular programming languages. This feature justifies the use of CORBA for urban environments where many different embedded systems were installed by different vendors through the time.

Several implementations of the CORBA specification exist today, both in the commercial and the open source domain. A detailed comparison of several CORBA implementations can be found in Puder [14]. When choosing a CORBA implementation, several considerations must be taken into account:

2.1. Supported platforms

When working with heterogeneous embedded systems, it is important to choose a CORBA implementation able to work properly in the majority of them. This is particularly true for urban environments, where ITS equipments from different vendors have been installed during a long period of time. In particular, System-on-Chip (SoC) architectures support is very interesting as they have been popularized for multimedia processors typically used in surveillance and monitoring applications.

2.2. Supported programming languages and prerequisites

The supported programming languages and prerequisites to successfully compile the CORBA implementation must be also taken into account. This is particularly true for ITS equipments usually based on a great variety of processors and operating systems. In this case, the heterogeneity of installed equipments requires the software to be adapted to the existing software.

2.3. Supported features

Depending on the final application, additional features like objects passed by value, asynchronous messaging, CORBA component model, etc.

2.4. Licenses

Commercial and open source CORBA implementations can be found nowadays, and the licenses under which they are released can affect the development of the project. In the case of urban environments, open source licenses are preferred by public authorities to achieve vendor independence.

MICO (www.mico.org) and TAO (http://www.theaceorb.com/) are fully standard-compliant implementation of CORBA that satisfies the previous detailed criteria. Both of them are completely written in C++, and supports the most widely used platforms, including SoC architectures based on ARM and MIPS processors. They are available under the "open source software" model. That means the source is freely downloadable, open for inspection, review and comment. Finally, they include most of the novel features of CORBA [14].

3. Performance evaluation

The following method was used to quantify the effects of a middleware service on an embedded system running a heavy application which demands most of processor's resources. The method takes advantage of the proc files available in GNU/Linux systems to retrieve data about CPU time assigned to the processes by the scheduler.

At first sight, the solution might be to measure the CPU load using the top command. This alternative could be enough in some simple cases, but in many others it could distort the measurements as a consequence of its features and its own CPU load. Some examples of situations where the top command does not offer accurate measurements are described below:

- When it is needed to synchronize the beginning and the end of the measuring period. This is very important in tests where the client requests are simulated with bursts. Since top cannot be synchronized to these bursts, it can consider a measuring period wider or narrower leading to false results.
- Some servers usually replicate themselves in order to process many client requests simultaneously. These replications are carried out by creating child processes which, of course, have their own PIDs and their own associated file in /proc. If the number of child processes is high, top has to read many files each time it refreshes and the information shown becomes complex to analyze since the actual CPU load is the sum of all the children's CPU loads. There are other cases where this problem appears although the servers do not create child processes. In old versions of GNU/Linux systems, multi-thread programs often appear as different processes since the kernel assigns a PID for each thread although it treats them in a different way. For our purposes, the server must be considered as a multi-process application in these cases. This problem is solved in recent kernel versions and a multi-thread server can be seen as a single process.
- When the refresh rate is high or when there are many processes being monitored, top may demand a significant amount of CPU time and this can produce unrealistic measurements.

When the use of top is not possible due to its limitations, a slighter and more selective method is required. A suitable method for measuring the average CPU load of middleware services has been developed for systems meeting these conditions:

- The middleware server may create many child processes.
- The Linux kernel version considers multi-threads programs as processes.
- The client requests can be modeled as bursts.
- The measuring period can be configured and must be synchronized to the burst.

To measure the effects of middleware on the main application running in an embedded system, we should compare the average CPU load in two cases: without and with the middleware server.

Measurements in the first case can be done directly with the top command by watching the percentage of CPU time the scheduler is assigning to the main application. The undesirable effects of top command load can be minimized forcing top to retrieve only the main application CPU time. Fig. 1 shows the average and the actual accumulated CPU time assigned to the main application without the middleware. This main application usually gets most processor's resources, so its

curve is very close to the total accumulated CPU time. Note that all these curves have always a slope between 0º and 45º. The steeper the curve, the higher the percentage of CPU use.

In the second case, top cannot be used due to the problems mentioned before but, since all the information gathered by top is available in the special files inside the /proc folder, we can insert some lightweight code in our middleware server in order to read these files and extract only the relevant information at certain moments. When there are only a few processes to consider, this solution may be suitable and its effects on the measurement (in terms of extra CPU load) are insignificant. For example, if both the main application and the middleware server are executed by single processes, the measurement code would only open three files: the one regarding the main application, the one regarding the middleware layer and the one that contains the total CPU time. Unfortunately, this is not always true as some middleware servers, such as Mico, create multiple threads on demand and the number of PIDs may be quite long, thus taking a significant time to read all the associated files.

To solve this problem we took advantage of the fact that the middleware server hardly consumes CPU time when the server is not receiving client requests. Then we can assume its CPU time curve is flat in these periods, as depicted in Fig. 2. Since the values contained in the proc files associated to the server do not change during the inactivity periods (before and after the burst of client requests) we can take the measurements within these periods (t_{m1}, t_{m2}) despite their long delays and we will get the same values as if we would have taken the measurements at the beginning and at the end of the burst.

$$\text{TCPU}_{srv}(t_{m1}) = \text{TCPU}_{srv}(t_s)$$

$$\text{TCPU}_{srv}(t_{m2}) = \text{TCPU}_{srv}(t_e) \tag{1}$$

These lengthy measurements should be taken at any time before and after the burst by a simple application. This application has to get first the list of PIDs related to the server and then retrieve the accumulated CPU times for each proc file related to the list of PIDs. The server's total accumulated CPU time is the sum of all these values. This task can be accomplished easily by a bash script.

In order to calculate the main process' load we only need to take the measurements at the beginning and at the end of the burst. This can be done by inserting some code in the service routine so that it retrieves information from the proc files related to the main process and the system when the server receives the first and the last request. But this solution has two problems:

1. How to determine the first and the last request.
2. Even though the code could determine the first and the last request, the server process may be working before and after executing
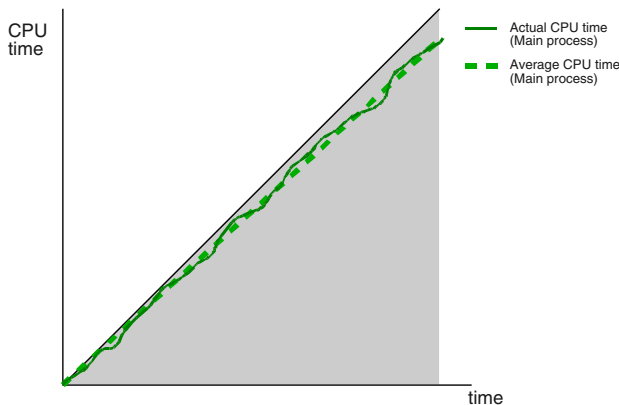

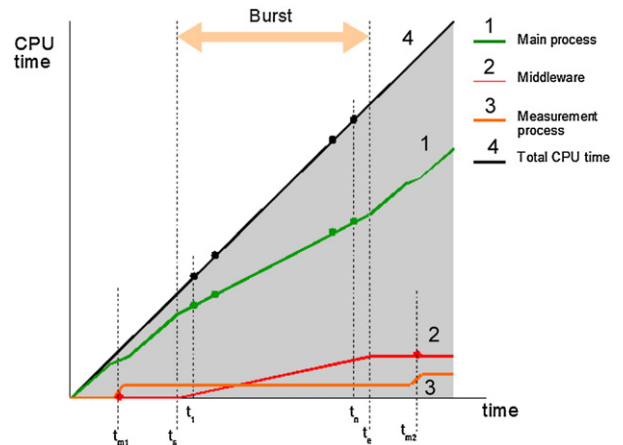**Fig. 1.** Average and actual accumulated CPU time without middleware server.

Fig. 2. Accumulated CPU time with middleware server.

the measurement code, thus consuming a little more time than estimated.

A straightforward way to solve the first problem is to mark the first and the last request from the client side but this may fail in some circumstances. This is particularly true when the request rate (requests per minute) is high and the server launches several child processes or threads which do not share any memory space. The scheduler at the server side might hold the thread serving the first request so that the actual request being first served would not be the marked one. One simple solution when communication between processes/threads is not possible is to mark the first and the last n requests, being n a small number, and, after finishing the burst, keeping only the lower and the higher values.

The second problem is difficult to be avoided, but it can be mitigated if the burst is long enough. Since we are interested in the average behavior, we can make the burst as long as we need provided the average request rate remains the same. The longer the burst, the smaller the effect. The following expressions show this from a mathematical point of view. The average CPU load of the main process (in %) is defined as:

$$L_{\text{MAIN}} = 100\left(\frac{\text{TCPU}_{MAIN}(t_e) - \text{TCPU}_{MAIN}(t_s)}{\text{TCPU}_{SYS}(t_e) - \text{TCPU}_{SYS}(t_s)}\right) =$$

$$= 100\left(\frac{\text{TCPU}_{MAIN}(t_e) - \text{TCPU}_{MAIN}(t_s)}{t_e - t_s}\right) \tag{2}$$

Assuming t_1 and t_n as the time of the first and the last measurement, respectively, the average CPU load of the main process according to the proposed method is:

$$L'_{\text{MAIN}} = 100\left(\frac{\text{TCPU}_{MAIN}(t_n) - \text{TCPU}_{MAIN}(t_1)}{\text{TCPU}_{SYS}(t_n) - \text{TCPU}_{SYS}(t_1)}\right) =$$

$$= 100\left(\frac{\text{TCPU}_{MAIN}(t_n) - \text{TCPU}_{MAIN}(t_1)}{t_n - t_1}\right) \tag{3}$$

The differences between the expected and the actual measurement times are:

$$\Delta t_s = t_1 - t_s; \Delta t_e = t_e - t_n \tag{4}$$

The errors in the measurements due to the problem explained before can be defined as:

$$e_{Ms} = \text{TCPU}_{MAIN}(t_1) - \text{TCPU}_{MAIN}(t_s)$$

$$e_{Me} = \text{TCPU}_{MAIN}(t_n) - \text{TCPU}_{MAIN}(t_e) \tag{5}$$

Substituting the previous definitions in the expression of the estimated average CPU load the result is given by Eq. (6).

$$L'_{\text{MAIN}} = 100\left(\frac{\text{TCPU}_{MAIN}(t_e) + e_{\text{Me}} - \text{TCPU}_{MAIN}(t_s) - e_{\text{Ms}}}{t_e - t_s - (\Delta t_s + \Delta t_e)}\right) \qquad (6)$$

Δt_e and Δt_s do not depend from the burst's length, so if the burst is long enough we can assume the following hypotheses:

$$t_e - t_s >> \Delta t_s + \Delta t_e \qquad (7)$$

$$L'_{\text{MAIN}} \approx 100\left(\frac{\text{TCPU}_{MAIN}(t_e) + e_{\text{Me}} - \text{TCPU}_{MAIN}(t_s) - e_{\text{Ms}}}{t_e - t_s}\right)$$

Since the CPU load of any running process is always under 100%, e_{Me} and e_{Ms} are lower or equal to Δt_e and Δt_s, respectively. These errors are also independent from the burst's length, so:

$$L'_{\text{MAIN}} \approx 100\left(\frac{\text{TCPU}_{MAIN}(t_e) - \text{TCPU}_{MAIN}(t_s)}{t_e - t_s}\right) = L_{\text{MAIN}} \qquad (8)$$

since $t_e - t_s >> e_{\text{Me}} - e_{\text{Ms}}$.

The CPU load of the middleware server can be estimated using Eq. (9).

$$L_{\text{SRN}} = 100\left(\frac{\text{TCPU}_{\text{SRV}}(t_e) - \text{TCPU}_{\text{SRV}}(t_s)}{\text{TCPU}_{\text{SYS}}(t_e) - \text{TCPU}_{\text{SYS}}(t_s)}\right) =$$
$$\left(\frac{\text{TCPU}_{\text{SRV}}(t_{m2}) - \text{TCPU}_{\text{SRV}}(t_{m1})}{t_e - t_s}\right) \qquad (9)$$

In addition to CPU load measurements, some useful information, such as memory consumption can also be extracted from the proc files without a significant additional load.

The proposed method takes advantage of some standards to accomplish the measurements. Particularly, it uses some shell utilities and system calls defined in the POSIX family of Standards (IEEE 1003). POSIX standards cover a wide range of aspects regarding the operating systems behavior, such as inter-process communication, multi-threading, exceptions, etc. From these standardized group of functions and utilities, the method uses those related to process management.

In order to retrieve detailed process information in a fast way, the method uses the procfs (proc filesystem). This pseudo-filesystem standardizes the way to access some important information about the operating system state, including a detailed report for every process running in the machine, which contains useful data for the method, such as the CPU time assigned to the process or the memory consumption. Although the procfs is not covered by a official standard, it is widely used in several UNIX-like operating systems such as GNU/Linux, BSD, Solaris, QNX, etc. Therefore, the proposed method can be easily ported to other embedded systems running operating systems which use these standard features.

4. Smart urban environment application

Smart environments and ambient intelligence can be considered today one of the possible instantiations of the emerging Information Society. New initiatives like ARTEMIS (Advanced Research & Technology for Embedded Intelligence and Systems, http://www.artemis-office.org/) are focused on making ambient intelligence a reality, to improve competitiveness, and to create opportunities for a new industry to flourish [16,17]. One of the fields highlighted by ARTEMIS is the traffic and transport application area, which is related to public infrastructure application context. Urban traffic management systems have evolved in recent years into a fully integrated architecture of Intelligent Transportation Systems (ITS) interconnected through a data network with the traffic control center. The ITS equipments themselves have also evolved from simple equipments with basic functionalities to modern equipments provided with high processing and connection capabilities. As they have developed a local intelligence, they can also provide local services to other equipments or citizens.

Urban traffic equipments are usually based on embedded computer platforms running an embedded operating system [18,15]. Among the different possibilities of available operating systems, Linux is firmly in first place as the operating system of choice for smart gadgets and embedded systems [19,20]. Several commercial ITS equipments have been used to implement a middleware layer able to offer services to citizens and traffic control centers (www.visioway.com). These equipments are used for traffic parameter estimation using artificial vision [5,21]. Fig. 3 is a picture of the main board. It includes the Freescale i.MX21 video processor based on an ARM926EJ-S core, memory cards and USB controllers, CMOS sensor interfaces, and an Ethernet interface [22].

The processor is running under ARM Linux (kernel 2.4.20), which provides access to a wide variety of open source software modules as well as the support of a large community of developers. MICO and TAO were cross compiled using arm-linux gcc-3.3.2. The size of the resulting library files after compilation is below 7.2 MB. Although this size could be large for certain embedded systems, this is not the case of ITS equipments related to artificial vision, which need a great amount of memory to deal with this kind of applications.

Fig. 4 shows the structure of the object request interface. The Object Request Broker (ORB) is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect that is not reflected in the object's interface.

Object invocations in CORBA are based on the client server paradigm. The client is the entity that wishes to perform an operation on the object and the server is the code and data that actually implement the object. CORBA objects are defined as interfaces in Interface Definition Language (IDL). This language defines the types of objects according to the operations that may be performed on them and the parameters to those operations.

A client application may invoke a method of a CORBA object through the IDL compiler-generated client stub, which is the local representation of a CORBA server. This local representation makes the invocations location transparent, as stubs may be representing co-located objects or remote objects. To make an invocation a client needs to get an object reference to the server, the process of binding the client to a reference creates the stub, which becomes object's entry point to the ORB. Client stub is responsible for marshalling requests to server, and demarshalling replies back to the client. Receiving requests and preparing replies in the server side are similar to the client side, with the request/reply marshalling and demarshalling taking place through the IDL compiler-generated server skeletons. However, the server process is responsible for implementing the interface as a servant object and activating the servant, if idle, upon receiving requests for the implemented interface. The ORB operations and policies needed to control and manage server behavior are aggregated in a Portable Object Adaptor (POA) interface.

CORBA assigns each object a unique IOR (Interoperable Object Reference) for the client application to find the remote object. An IOR contains a string of characters used to locate and identify remote objects. Although the OMG defines the Naming Service as the preferred way for a client to obtain an IOR of a remote object, many CORBA implementations pass the IOR of the server's object to the client in a common file that will be read by the client during initialization. This alternative facilitates services registration and the possibility of obtaining
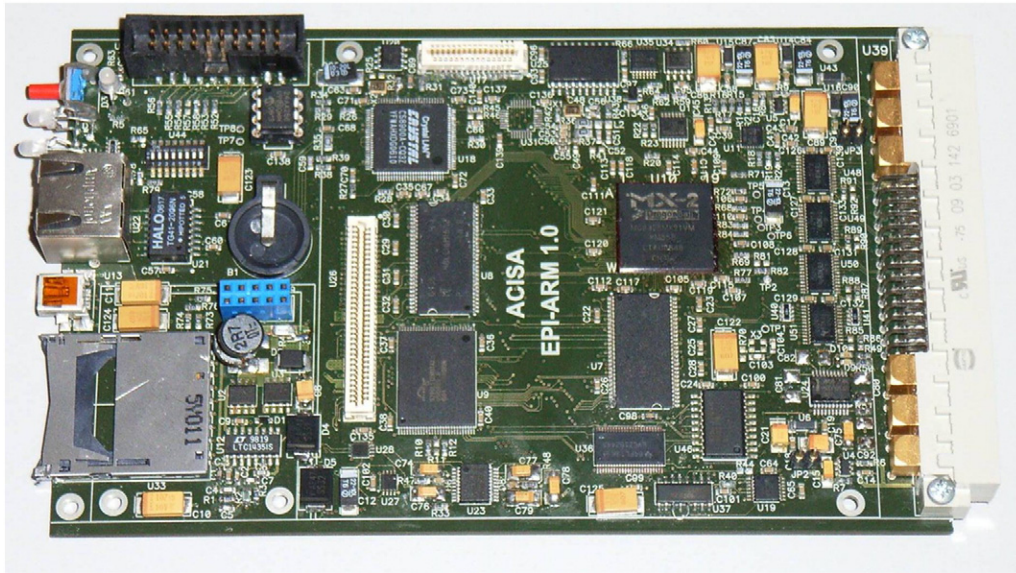
Fig. 3. Main board of ITS equipment for traffic parameter estimation using artificial vision.

a directory of services. Clients may also use a dynamic invocation in the case that the server is not running.

The idea of an urban smart environment is illustrated in Fig. 5. Several embedded systems interconnected through an Ethernet network spread over the city are located at the physical layer. Each one is developing a particular task, for instance, detecting the queue length in front of a traffic light (scenario 1), counting vehicles using artificial vision techniques (scenario 2) or just controlling a traffic panel (scenario 3). The CORBA specifications define the General Inter-ORB Protocol (GIOP) as its basic interoperability framework. GIOP realization over TCP/IP is Internet-IOP (IIOP) and for an ORB to be CORBA compliant, IIOP must be supported. This is the case of the two selected CORBA implementations, MICO and TAO.

The implemented ORB is responsible of connecting services provided by these equipments to local users (citizens) or the traffic control center. The client for the available services is a web server containing a directory that is continuously updated, as different equipments start or finish their operations. This web service can be remotely accessed by citizens or driver assistance systems. The detailed list of services included in the proposed platform is the following:

• Vehicle queue detector. The equipment detects whenever the queue of vehicles in front of a traffic light goes beyond a fixed threshold. The

limits of the queue region can be remotely configured by drawing virtual detectors on the image of the scene, as shown in Fig. 5, scenario 1. The blue and red regions in front of the traffic light delimit the queue detection regions. The service consists of delivering the current state of the queue of vehicles in the selected traffic light [21].
• Traffic density estimator. The equipment is based on a vehicle detector system using artificial vision techniques (Fig. 5, scenario 2). The service provides data about traffic flow and incident detection [5].
• Panel messages. Panels can be automatically updated using services provided by other equipments connected to the urban network. For instance, a traffic panel can automatically show an incident detection or traffic delay requiring services like the ones provided by previous equipments.

The first two considered equipments have to deal with several computer vision algorithms. The most prominent among those implemented are the background subtraction and the shadow removal algorithms. Both of them require rather complex heuristics and intensive floating-point computation.

The background subtraction technique allows extracting a moving object from an image sequence obtained using a static camera. This should be accomplished even with objects permanently moving around in the scene. It is based on the estimation of the so called background-model of the scene. This model is used to obtain a reference image which is compared to each recorded image. Consequently, the background-model must be a representation of the scene after removing all the non-stationary elements, and must be permanently updated to take into account the changing lighting conditions or any change in the background texture [5,21]. Surveys and comparisons of different algorithms for background subtraction can be found in Jacques et al. [23]. Shadow removal is a main concern when dealing with images in outdoor or unstructured environments, like traffic scenes where the shadows of vehicles are cast over neighbor regions. Shadows have some particular properties that can be exploited in order to eliminate them or, at least, to reduce its presence in the image [24,25]. The adopted solution for shadow detection and removal is inspired by the work described in [26].

5. Experimental results

In this section we have applied the proposed method for quantifying the computational cost of including a middleware layer like MICO
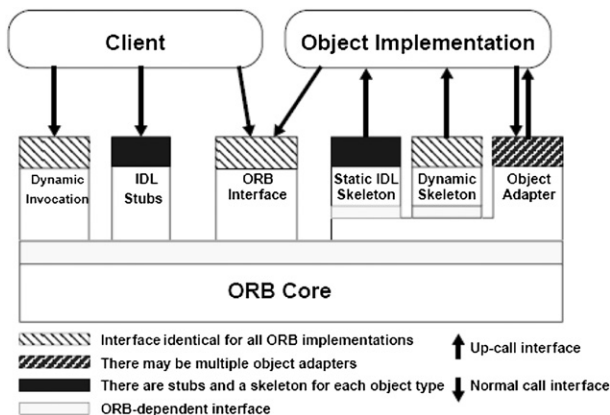

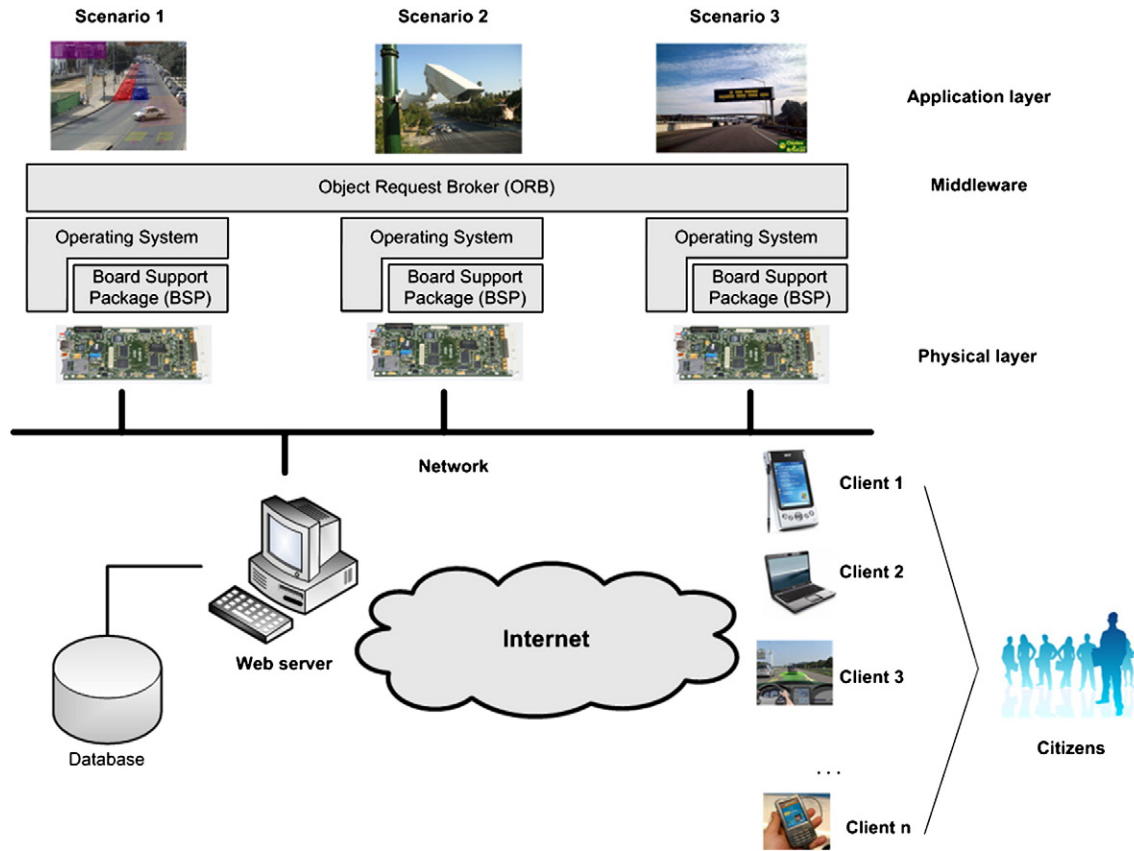**Fig. 4.** Structure of the MICO object request interfaces.

Fig. 5. Urban smart environment scheme.

and TAO from the point of view of transaction processed per second. This computational cost can be important when dealing with computational intensive equipments like vision-based ITS equipments. We have focused on one of this artificial vision-based equipments to test how its functionality is affected by the embedded middleware implementation. In particular, the traffic density estimator of Fig. 5,

Fig. 6. CPU time and memory requirements of concurrent processes executed in the selected traffic equipment.

scenario 2 has been employed. Fig. 6 details the running processes on the equipment and their CPU time and memory requirements.

The running processes correspond to the video processing algorithms (epi.out) and the SSH server for remote logging. In particular, the first one consumes the majority of the CPU time (98,8%), but just a small amount of memory (4,1%). Notice that the equipment is provided with a large amount of memory (64 MB) to deal with video applications.

Whenever several clients request the provided service, the middleware application creates several threads on the server side to achieve load balancing. Fig. 7 illustrates a situation where up to seven threads have been released to process several clients request. As a Linux kernel 2.4.20 is being used, the server threads are identified like processes with PIDs. Notice that the main process epi.out is still using the majority of the CPU time as all the servers are sleeping. But when the clients are attended, the performance of the main process is affected.

The proposed methodology has been applied to evaluate load-balancing mechanism for client requests. In particular, Fig. 8 details how the main process is affected when the implemented middleware is attending external demands. From the client side, the number of clients for 1 min is specified as an input parameter. From the server side, an estimation of the CPU load due to the main application (epi.out) and the server (MICO and TAO) can be obtained following the proposed methodology. Experiments have been repeated to average the variability of results due to the fact that the video processing algorithms are not always demanding exactly the same amount of CPU time in each individual experiment. As expected, the obtained results show that the CPU time remaining for the epi.out process decreases, while the number of server threads increases with the number of clients. It can be observed that the CPU load of the main process remains always above the 80% in the case of MICO, even in those cases with more than 1000 clients demanding services. The behavior of TAO is slightly better. The obtained results can help to determine how the selected processor must be oversized to support its normal operation as well as the middleware layer. A 20% is enough for the illustrated case examples.

6. Conclusions

An embedded middleware platform of smart urban environments has been presented. The possibility of connecting spread urban equipments using a middleware layer can contribute to offer new added-value services integrated in a smart environment. However, this possibility has also a cost in terms of the computational consumption due to the embedded software necessary to implement the middleware layer. A methodology for measuring the impact of a middleware layer in normal processor operation has been proposed and two open source middleware like MICO and TAO have been selected to measure its performance in video-based urban equipments. The proposed method can provide useful hints for optimizing a middleware service and minimizing its impact on the main application.

```
12:37am  up 29 min,  0 users,  load average: 2.00, 1.97, 1.68
50 processes: 46 sleeping, 3 running, 1 zombie, 0 stopped
CPU states: 71.7% user, 28.2% system,  0.0% nice,  0.0% idle
Mem:    62992K av,   27068K used,   35924K free,      0K shrd,      0K buff
Swap:      0K av,       0K used,      0K free                   14092K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
  154 root      25   0  2988 2728  1064 R    97.0  4.3  28:59 epi.out
  367 root      16   0   960  960   776 R     2.1  1.5   0:00 top
  109 root      15   0   240  240   208 S     0.5  0.3   0:03 EPIWatchdog.out
  138 root      15   0  1300  536   456 S     0.3  0.8   0:05 mpegServer.out
    1 root      15   0   528  528   492 S     0.0  0.8   0:06 init
    2 root      0K   0     0    0     0 SW    0.0  0.0   0:00 keventd
    3 root      34  19     0    0     0 SWN   0.0  0.0   0:00 ksoftirqd_CPU0
    4 root      25   0     0    0     0 SW    0.0  0.0   0:00 events/0
    5 root      25   0     0    0     0 SW    0.0  0.0   0:00 kswapd
    6 root      25   0     0    0     0 SW    0.0  0.0   0:00 bdflush
    7 root      15   0     0    0     0 SW    0.0  0.0   0:00 kupdated
    9 root      25   0     0    0     0 SW    0.0  0.0   0:00 mtdblockd
   10 root      25   0     0    0     0 SW    0.0  0.0   0:00 khubd
   11 root      25   0     0    0     0 DW    0.0  0.0   0:00 mx2otg_id
   12 root      35  10     0    0     0 Z N   0.0  0.0   0:00 jffs2_gcd_mtd2 <defun
   29 root      35  10     0    0     0 SWN   0.0  0.0   0:03 jffs2_gcd_mtd2
   44 root      15   0  1152 1152  1060 S     0.0  1.8   0:00 sshd
  139 root      15   0  1300  536   456 S     0.0  0.8   0:00 mpegServer.out
  141 root      17   0  1300  536   456 S     0.0  0.8   0:00 mpegServer.out
  143 root      16   0   528  528   492 S     0.0  0.8   0:00 init
  144 root      15   0  2244 2240  2132 S     0.0  3.5   0:00 httpd
  145 daemon    24   0  2244 2240  2136 S     0.0  3.5   0:00 httpd
  146 daemon    24   0  2244 2240  2136 S     0.0  3.5   0:00 httpd
  147 daemon    24   0  2244 2240  2136 S     0.0  3.5   0:00 httpd
  148 daemon    24   0  2244 2240  2136 S     0.0  3.5   0:00 httpd
  149 daemon    24   0  2244 2240  2136 S     0.0  3.5   0:00 httpd
  150 root      15   0  1544 1544  1240 S     0.0  2.4   0:01 sshd
  152 root      15   0  1060 1060   852 S     0.0  1.6   0:00 sh
  156 root      15   0  1548 1548  1240 R     0.0  2.4   0:00 sshd
  158 root      15   0  1068 1068   856 S     0.0  1.6   0:00 sh
  159 root      15   0   468  464   408 S     0.0  0.7   0:00 webCommandServe
  240 root      15   0  4844 4840  4020 S     0.0  7.6   0:00 server
  244 root      15   0  4844 4840  4020 S     0.0  7.6   0:00 server
  245 root      15   0  4844 4840  4020 S     0.0  7.6   0:00 server
  246 root      15   0  4844 4840  4020 S     0.0  7.6   0:00 server
  247 root      15   0  4844 4840  4020 S     0.0  7.6   0:00 server
  270 root      15   0  4844 4840  4020 S     0.0  7.6   0:00 server
  271 root      15   0  4844 4840  4020 S     0.0  7.6   0:00 server
  280 root      15   0  4844 4840  4020 S     0.0  7.6   0:00 server
```

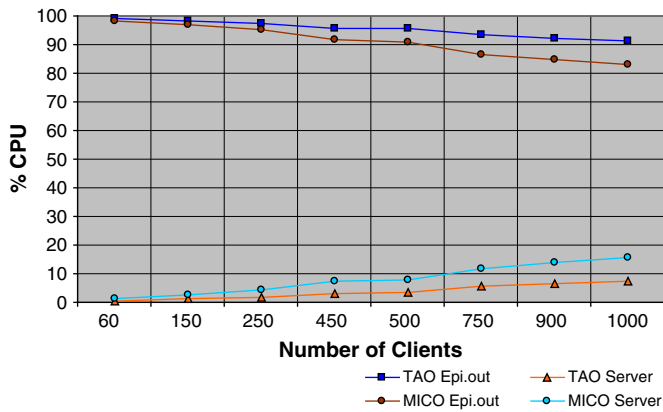**Fig. 7.** CPU time and memory requirements including several server processes.

Fig. 8. Evolution of the CPU load with the number of clients.

References

[1] D. Cook, S. Das, Smart Environments: Technology, Protocols and Applications, Wiley-Interscience, 2004..

[2] K. Plößl, H. Federrath, A privacy aware and efficient security infrastructure for vehicular ad hoc networks, Computer Standards & Interfaces 30 (6) (2008) 390–397.

[3] F. Barrero, S.L. Toral, S. Gallardo, EDSPLAB: remote laboratory for experiments on DSP applications, Internet Research 18 (1) (2009) 79–92.

[4] M. Bramberger, A. Doblander, A. Maier, B. Rinner, H. Schwabach, Distributed embedded smart cameras for surveillance applications, Computer 39 (2) (2006) 68–75.

[5] S.L. Toral, M. Vargas, F. Barrero, Embedded multimedia processors for road-traffic parameter estimation, Computer 42 (12) (2009) 61–68.

[6] R. Schantz, D. Schmidt, Middleware for Distrib-uted Systems: Evolving the Common Structure for Network-centric Applications, Encyclopedia of Software Engineering, Wiley and Sons, 2002.

[7] I. Gorton, A. Liu, P. Brebner, Rigorous evaluation of COTS middleware technology, Computer 36 (3) (2003) 50–55.

[8] M. Henning, A new approach to object-oriented middleware, IEEE Internet Computing Magazine 1 (2004) 66–75.

[9] A. Tripathi, Challenges designing next-generation middleware systems, Communications of the ACM 6 (2002) 39–42.

[10] OMG, Common Object Request Broker Architecture: Core Specifications, version 3.0.3, http://www.omg.org/technology/documents/formal/corba_iiop.htm Dec. 2006.

[11] U. Mutlu, R. Edwards, P. Coulton, QoS aware CORBA Middleware for Bluetooth, In: 2006 IEEE Tenth International Symposium on Consumer Electronics, ISCE '06, 2006, pp. 1–7.

[12] K.-C. Liang, D. Chyan, Y.-S. Chang, W.-T. Lo, S.-M. Yuan, Integration of CORBA and object relational databases, Computer Standards & Interfaces 25 (4) (2003) 373–389.

[13] Z. Tari, O. Bukhres, Fundamentals of Distributed Object Systems: The CORBA Perspective, John Wiley & Sons, Inc., 2001.

[14] A. Puder, MICO: an open source CORBA implementation, IEEE Software 21 (4) (2004) 17–19.

[15] T.-L. Tseng, W.-Y. Liang, C.-C. Huang, T.-Y. Chian, Applying genetic algorithm for the development of the components-based embedded system, Computer Standards & Interfaces 27 (6) (2005) 621–635.

[16] C. Gill, V. Subrarnonian, J. Parsons, H.M. Huang, S. Torri, D. Niehaus, D. Stuart, ORB middleware evolution for networked embedded systems, In: Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems, 2003. (WORDS 2003), Vol. 15–17, 2003, pp. 169–176.

[17] ARTEMIS Strategic Research Agenda Working Group, Strategic Research Agenda, First Edition Ideo Ltd., 2006.

[18] C. Arth, H. Bischof, C. Leistner, TRICam—An Embedded Platform for Remote Traffic Surveillance, In: 2006 Conference on Computer Vision and Pattern Recognition Workshop, Vol. 17–22, 2006.

[19] S.L. Toral, M.R. Martínez-Torres, F. Barrero, Virtual communities as a resource for the development of OSS projects: the case of Linux ports to embedded processors, Behavior and Information Technology 28 (5) (2009) 405–419.

[20] S.L. Toral, M.R. Martínez-Torres, F. Barrero, F. Cortés, An empirical study of the driving forces behind online communities, Internet Research 19 (4) (2009) 378–392.

[21] S. Toral, M. Vargas, F. Barrero, M.G. Ortega, Improved Sigma-Delta background estimation for vehicle detection, Electronics Letters 45 (1) (2009) 32–34.

[22] M. Vargas, J.M. Milla, S.L. Toral, F. Barrero, An enhanced background estimation algorithm for vehicle detection in urban traffic scenes, IEEE Transactions on Vehicular Technology 59 (8) (2010) 3694–3709.

[23] M. Piccardi, Background Subtraction Techniques: A Review, In: IEEE International Conference on. Systems, Man and Cybernetics, 2004, Vol. 4, 2004, pp. 3099–3104.

[24] A. Prati, I. Mikic, M.M. Trivedi, R. Cucchiara, Detecting moving shadows: algorithms and evaluation, Transactions on Pattern Analysis and Machine Intelligence 25 (7) (2003) 918–923.

[25] R. Cucchiara, C. Grana, M. Piccardi, A. Prati, Detecting moving objects, ghosts, and shadows in video streams, Transactions on Pattern Analysis and Machine Intelligence 25 (10) (2003) 1337–1342.

[26] J.C.S. Jacques Jr., C.R. Jung, S.R. Musse, Background subtraction and shadow detection in grayscale video sequences, In: Proc. of the XVIII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'05), 2005, pp. 189–196.
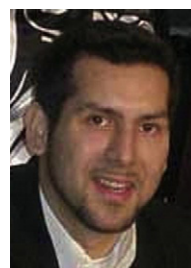
S. L. Toral received the M.Sc. and Ph.D. degrees in Electrical and Electronic Engineering from the University of Seville, Spain, in 1995 and 1999, respectively. He is currently a Full Professor of the Department of Electronic Engineering, University of Seville. His research interests include embedded systems, distributed systems and ad hoc networks.

F. Barrero received the M.Sc. and Ph.D. degrees in Electrical and Electronic Engineering from the University of Seville, Spain, in 1992 and 1998, respectively. In 1992, he joined the Electronic Engineering Department at the University of Seville, where he is currently an Associate Professor.

F. Cortés received the M.Sc. and Ph.D. degrees in Electrical and Electronic Engineering from the University of Seville, Spain, in 2007 and 2011, respectively. He is currently an Assistant Professor of the Department of Electronic Engineering, University of Seville.

D. Gregor received the M.Sc. degree in Electrical and Electronic Engineering from the University of Seville, Spain, in 2009. He is currently a researcher of the Department of Electronic Engineering, University of Seville.