Real-time Dynamic Guarantee in Component-based Middleware

Cássia Yuri Tatibana Carlos Montez Rômulo Silva de Oliveira
Pós-Graduação em Engenharia Elétrica
Universidade Federal de Santa Catarina
{cytatiba,montez,romulo}@das.ufsc.br

Abstract

This paper describes an extension to the component-based programming model to support real-time dynamic guarantee for distributed applications. The extended model aims to include an acceptance tests to component-based servers at bind time. We present the mapping of our model to the CORBA Component Model, the architecture to support the dynamic guarantee in component-based middleware, the implementation of this architecture in CIAO (Component Integrated ACE ORB) and the result of experiments run to evaluate the cost of the mechanisms used.

1. Introduction

Middleware is applied to a wide range of distributed applications. As it becomes more pervasive, more capabilities are developed and added to this technology to address the specific needs of different classes of applications. The consequence is sophisticated technologies with such a high complexity level that keeps it from being widely applied.

Component-based middleware was developed in response to the need to cope with this increasing complexity, shortening software development cycles and reducing costs. The component-based approach defines a separation of concerns that can potentially leverage the complexity inherent to large scale real-time systems by detaching application functionality from real-time behavior. As a result, real-time applications no longer need to be tightly coupled to the underlying platform; the application real-time behavior can be configured instead of hard coded.

Real-time software components can be reused in different compositions and timing configurations. Reusability allows systems faster development by alleviating the maintainability, upgrade and extension processes.

Standard component-based models like Enterprise Java Beans and CORBA Component Model [9] deal successfully with business oriented requirements, but provide insufficient support to real-time applications [17].

In the literature, many component models, frameworks and tools were developed for different classes of real-time systems. In the class of embedded systems, memory constraints and deterministic behavior are usually essential. There are many component-based projects and products addressing embedded systems in the industry and in research [3], [15]. Instead of runtime adaptability or reconfigurability, the effectiveness of this class of components is measured by properties like memory consumption, testability and understandability [6].

Distributed real-time applications ranging from air traffic control to some classes of videogames (massively multiplayer online games) operate under varying workload in heterogeneous environment. For different reasons all these applications must provide a certain quality of service level for clients. In the case of videogames, servers over the Internet provide the required quality level for an unpredictable number of paying players. They require availability of enough resources (processor or bandwidth, for example) in all times or at least alternatives to deal with overload conditions maintaining a limit for the degradation of the service. For other types of systems, like factory automation, surveillance and target tracking systems [13], requirements are more critical. Their design involves issues like interoperability across different platforms, delays from network communication and the capability to adapt to changes of operational conditions since the workload and resources availability can not be characterized accurately `a priori`.

This paper describes the DGC (Dynamic Guarantee for Components) model that applies mechanisms to enable an acceptance test in component-based real-time systems at bind time. The mechanisms are generic enough to accept different algorithms for schedulability analysis at the acceptance test. We describe a CCM proof of concept implementation and the results of some experiments executed to measure the mechanism overhead.

The paper is organized as follows: in the next section it is presented an overview of the model. In section 3 we present the main elements that characterize the CORBA Component Model. The architecture for dynamic guaran-

tee is described in section 4. In section 5, we describe the implementation of the architecture. The experiments with the architecture are documented in section 6. Some of the main related works are briefly described in section 7 and section 8 presents our concluding remarks.

2. Dynamic Guarantee in Component-based Applications: The DGC Model

The provision of dynamic guarantee in application servers requires the knowledge of the workload that an invocation incurs and the current workload condition of the server domain. This is not a simple task specially if we have to cope with the distribution of the server domain over heterogeneous platforms. This section presents our model for dynamic guarantee in component-based real-time systems. This model evolved from the work presented in [14].

Basic Definitions The model was elaborated assuming applications structured in N ($N > 1$) components deployed in distributed nodes. The server components use synchronous communication, the communication delays are accounted in one of the components execution time. The application client side is not explicity addressed, although real-time constraints are expected from the client, the time taken for the communication between client and server is not accounted. The server could receive connection attempts from any number of clients at all times.

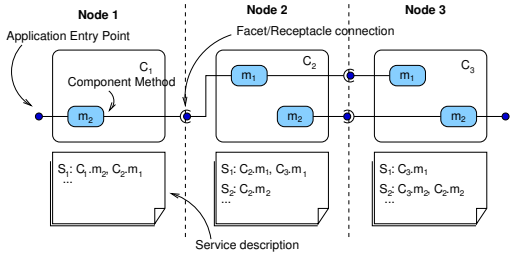

Figure 1. Execution flow: service mapping.

In the definition of our model, a **service** is the provision of a server functionality through the execution of a sequence of methods. These methods may be implemented by different components that are deployed in different nodes. Figure 1 illustrates the local representation of service S_1 (the local method to be executed and the reference to the remote method to be called) in each node.

As the execution of a service may not always mean a simple sequential execution of methods, a task abstraction was adopted. Each task in the server application corresponds to the execution of one block of code that does not involve the invocation of remote methods. When a local method m_1 invokes a remote method m_2, for example, m_1 description is composed of two sets of information; (i) the task that has an arrival time at the moment m_1 was invoked and that ends at the moment m_2 is invoked; and (ii) the task that starts at the moment m_2 returns and ends at the end of m_1.

The component facets or exposed interfaces that can be acessed by application clients are called **entry points**. Each entry point gives clients the access to a set of methods that are the first on the sequence that implements a service. Figure 1 illustrates two services provided by the server, their entry points are at components C_1 and C_3.

Clients issue periodic calls to the server. At a client first invocation it is required the following information: the functionality it requires, the frequency for the invocations, the deadline it needs the functionality to be accomplished and the time interval during which it will issue the requests (**session time**). If the activation of the tasks related to the periodic calls allows a feasible schedule at the server, the client is said to be approved in the system, it passed the schedulability or **acceptance test**. It receives the expected response from the server concerning the functionality requested and proceed with the subsequent periodic calls. For these subsequent calls, only the client identification must be provided as access control to the server is managed through client identification. When the activation of tasks related to the client calls do not allow a feasible schedule at the server, the client receives an exception and cannot issue subsequent periodic calls. It has failed the acceptance test.

Managing Workload As shown in Figure 1 each component may participate in providing different services. The component C_1 is involved in providing service S_1 only, while component C_2 and C_3 are involved in services S_1 and S_2. From the Node 2 perspective, the local workload involves the execution of tasks from methods m_1 and m_2 that must maintain the precedence relationship with the other remote methods from service S_1 and S_2. These relationships are determinant when building the schedule of tasks that share resources. Considering that each node may have different resources capacity, domain heterogeneity must also be considered to provide dynamic guarantee. This model addresses the server domain heterogeneity by allowing temporal protection between nodes and enforcing local real-time behavior.

Deadline partitioning mechanisms split the deadline requested by the client for one specific service into M partial deadlines. Assuming M as the number of tasks that composes the service requested. Once all local tasks deadlines are available the local schedule of tasks can be built. The partial deadline of a task is used as the arrival time for the next task in the service. Therefore, the real-time constraints of each task is translated in every node and the local work-

load can be planned assuming independent periodic tasks (jitter cancelation). After deadline partitioning and the adjust of tasks real-time parameters, local tasks schedulability test determines the acceptance or the rejection of the new client to the system.

Real-time guarantees A schedulability test can be used to ensure that the real-time constraints of previously accepted tasks will be respected and it does not allow new tasks into the system if that acceptance would jeopardize the real-time constraints already guaranteed.

For each request, the acceptance test is applied at the nodes that hosts the components that participate in providing the service requested. The approval of the new client depends on the agreement of all the nodes that applied the schedulability test. If all that nodes could reserve local resources for the client, then server access is granted (Figure 2). If at least one node does not have enough resources, the request is rejected.

The real-time guarantee provided reflects what the local operating system, communication mechanisms used and algorithms for deadline partitioning and schedulability tests are capable to support. If deterministic operating system and communication mechanisms are used the system allow the proposed architecture to provide deterministic guarantee. On the other hand, if the operating system and communication support have probabilistic timing behavior, the architecture will only provide an overload detection mechanism.

3. The CORBA Component Model

The model presented in section 2 could be implemented in different component platforms. We developed an implementation of the model for CORBA Component Model (CCM) [9]. This section provides an overview of CCM emphasizing the elements used for the description of our model.

The CCM specification extends the CORBA object model to support the concept of components and establishes standards for implementing, packaging, assembling, and deploying component implementations. From a client perspective, a CCM component is an extended CORBA object that encapsulates various interaction models via different interfaces and connection operations. From a server perspective, components are units of implementation that can be installed and instantiated independently in standard application server runtime environments defined by the CCM specification [12].

Components are connected to each other through ports, that are named interfaces and connection points: (i) **Facets** define a named interface that services method invocations from other components; (ii) **Receptacles** provide named connection points to facets provided by other components; (iii) **Event sources and event sinks** are used to exchange event messages between components.

Facets and Receptacles are synchronous ports while event sources and event sinks are assynchronous ones.

The component **container** provides the server runtime environment for component implementations called executors. It contains various pre-defined hooks and operations that give components access to non-functional services, such as persistence, event notification, transaction, replication, load balancing, and security. These services can be configured at deployment time according to the application requirements [12].

Two types of clients are supported by CCM: component-aware and component-unaware clients. Component-aware clients know they are making the request to a component and can choose to interact with a component through one or more CORBA interfaces: the equivalent interface implied by the component IDL declaration, supported interfaces declared on the component specification, facets or home interface supported operations. Component-unaware clients see a single supported interface of the component.

4. Providing Dynamic Guarantee in CCM

In this section we describe the mapping of the definitions used to describe the DGC model to the elements in CCM.

4.1. Mapping Dynamic Guarantee over CCM

Only synchronous ports were considered in the model proposed, all the components for which dynamic guarantee is provided communicate through facets and receptacles. The client access to server components is provided through a component facet or a supported interface; the **entry point**. The same XML descriptors used for server components deployment are extended to provide pre-runtime information for the dynamic guarantee mechanism. It must be included in the descriptors the specification of the components that must be monitored and which of them provides entry points.

We call the **bind time** between client and server, the time of the first invocation of a service from a client to the server. At bind time, an acceptance test is applied. For component-unaware clients, it corresponds to the first invocation to the server. For component-aware clients the bind time is the invocation of a service provider facet. The period specified by the client is treated by the server as the minimal interarrival time of the requests. The interarrival time is used to keep clients from issuing requests at a rate higher than specified.

4.2. Architecting Dynamic Guarantee in CCM

The dynamic guarantee for components will be implemented as a Dynamic Guarantee Service provided by the container. Just as the transaction and persistence services, it must be configured pre-runtime.

A manager for real-time aspects is collocated in every server component node to calculate the feasibility to schedule the local tasks activated by the client request. To do that, the real-time constraints sent with the request must be adjusted to the nodes involved in the service provision. This architecture manager is responsible for adjusting clients constraints into local tasks real-time parameters, verify the feasibility to schedule these tasks, allocate the necessary resources and activate the managers from other nodes involved in providing the requested service. When all the nodes involved in the provision of the service requested confirms the allocation of resources, the client is accepted into the server and its request proceeds normal execution. The managers located at the server domain nodes execute the same steps recursively. For the application example in Figure 1, the interactions of managers are described in Figure 2.

To implement all the functionalities of the Dynamic Guarantee Service we used the following elements: (i) client interceptor, that sends clients real-time constraints with the requests; (ii) server interceptor that calls the DGManager when clients invoke an application entry point; and (iv) the DGManager, that controls clients access to the server functionalities. In the following section we describe how the architecture was implemented in CIAO.

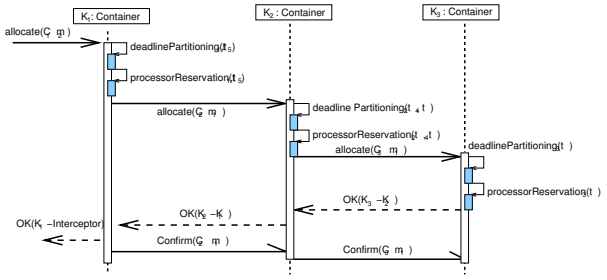

Figure 2. Runtime interactions example.

5. Implementing Dynamic Guarantee on CCM

The architecture developed to support our model is composed by DGManagers, CIAO's `Node Application` and CORBA portable interceptors. The role of the main modules that composes the model are described in the subsections bellow.

5.1. Node Application

The `Node Application` is part of the OMG Deployment and Configuration Specification [8]. It plays the role of a component server process that provides the computing resources for the components it hosts. Based on metadata provided by deployment tools, the `NodeApplication` creates the initial containers that provide an environment for creating and instanciating application components [2].

The `Node Application` is started before the application is deployed in the nodes and executes as long as the application is running. The only modification needed into the `Node Application` is the addition of the code to install the server `portable interceptor` (see next section). This is made through usual mechanisms for installing interceptors in CORBA ORBs [10].

5.2. Interceptors

CORBA portable interceptors [10] were used at the client and the server side of the application. Interception points at the components container level would be ideal for the purpose of this work. Unfortunately, there is no suport yet for this in CIAO and neither the specification for container interceptors is finished. CORBA Portable Interceptors are used at application clients and server `Node Applications`. All the remaining code created by this work is placed into CORBA portable interceptors and CIAO standard components.

Client Interceptor At the client side, the request interceptor is used to add the real-time constraints into the requests to the application server. The values for period and deadline required by a client are defined in a specific XML descriptor to be read into the interceptor at the moment the client invokes the server. For now, the following information is used: client identification, period, deadline, service to be invoked and session time. Once the client request is accepted at the server side, subsequent requests will include only the client identification field.

Server Interceptor After deployment, the request interceptor at the server side locates the DGManager facet reference at the Naming Service and connects to it. As the interceptors are placed in the ORB, all requests directed to any application located at the node are intercepted, the interceptor must filter the invocations to the application entry points. If the component does not provide any entry point, its respective DGManager is not invoked by the request interceptor, but by other DGManagers.

When an entry point is invoked, the interceptor invokes the DGManager `access` method that is explained in the

next subsection. Depending on the method returned value, the DGManager `allocate` method is also invoked.

5.3. DGManager

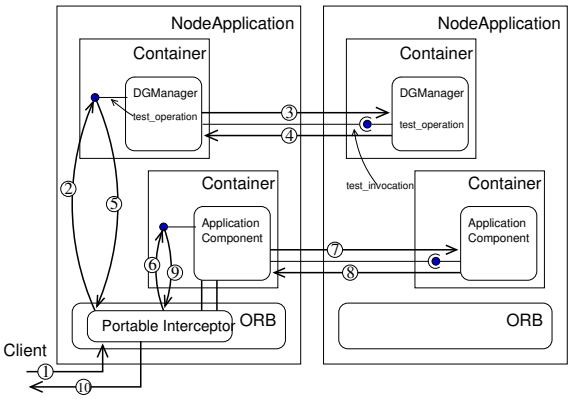

Figure 3. Accepted bind request.

The DGManager encapsulates most of the functionality of the real-time aspects manager from the Dynamic Guarantee service. It is implemented as a CIAO component which instances are deployed in the server side nodes (Figure 3). Each DGManager instance monitors one or more components collocated at the same node. Its functionalities are exposed through a facet named `test_operations` and it accesses other DGManagers through a receptacle named `test_invocation`. When two components involved in one application service are hosted in two different nodes, the DGManagers from these nodes are connected to each other.

The facet `test_operation` provides two methods:

- `access`: requires the client identification and the method intended to be called as parameters and returns a boolean value. It simply verifies if one client that is trying to access the server has already gained permission for it. If the client identification is stored in the DGManager, that means it has passed through a acceptance test and is still in its session period. So the resources it needs are still allocated to it. This method also aims to ensure that the client respects the minimal interarrival time: the time of the last request is stored and compared to the following request. If the interarrival time is smaller than the period specified by the client at the acceptance test, the request is not processed, and a `CORBA::NO_RESOURCES` exception is issued. For new clients or for clients that exceeded its session period, `access` returns *false*.
- `allocate`: requires the client identification and timing constraints. This method is called when `access` returned *false*. `allocate` is the method through which the mechanism for an acceptance test is launched. It executes: (i) the deadline partitioning algorithm; (ii) the local acceptance test; (iii) the local resources reservation; (iv) the `allocate` method invocation in the other DGManagers that monitor the dependencies of the method requested; (v) allocate local resources upon the confirmation from the other DGManagers concerning the availability of resources for the client call (Figure 2).

The communication between DGManagers (steps 3 and 4 from Figure 3) also pass through the ORB, just like the communication between the application components (steps 7 and 8). But they are not illustrated in the Figure because in this communication no verification is executed at the interceptor.

The connection between DGManagers as well as the applications components is made as usual at deployment time, when it is already known which components are involved in each service and where the application components are deployed in the network. After application components and DGManagers deployment is finished, the DGManagers configuration is made through XML descriptors. The descriptors define which components each DGManager is going to monitor and what are the methods to be executed as part of which service. This information allows the DGManagers adjust the end-to-end real-time constraints provided by the client in constraints for each local method.

The *DGManager* class manages a list of *MonitoredComponents* instances. The *MonitoredComponent* class represents the application component whose timing behavior is intended to be controlled. New instances of this class are created according to the inclusion of new components to be monitored. Currently it is done once at the deployment of the application: however, it can be extended to be made at runtime in the case of re-deployment of components.

Each *MonitoredComponent* instance manages two lists; one for the *Operation* class and one for the *AcceptedClient* class. The *Operation* class represents each of the methods that the monitored component provides. Instances of this class are also added at the deployment of the components when the information concerning the tasks that composes the operation are provided to the DGManager. Two kinds of tasks are described in the *Operation* class; local tasks which will execute at the local node, and remote tasks that includes all the calls invoked by the local task. The task description consists of: arrival time and estimated execution time. The *AcceptedClient* class represents the clients connected to the server. Instances of this class are created at runtime and added to the *MonitoredComponent* list or removed from the list according to the session time requested by each client.

6. Experimental Evaluation

The model described in section 2 was implemented using CIAO, an implementation of the Lightweight CORBA Component Model (CCM) and Real-time CORBA [7] built on top of TAO (The ACE ORB). The implementation described in this section is an example, we chose a simple deadline partitioning and utilization based acceptance test for Earlier Deadline First scheduling approach.

The algorithm for deadline partitioning implemented in the DGManagers `allocate` method for the implementation described in this paper is the Equal Flexibility (EQF) [4]. It assumes EDF as the scheduling policy in every node. EQF aims to provide equal flexibility to all subtasks for which the deadline is partitioned.

It is important to notice that the architecture does not impose an algorithm for the deadline partitioning. As the schedulability test, this mechanism depends on the processor scheduling capacity.

For the acceptance test, a simple utilization based algorithm was applied. The processor utilization U of each node is given by the sum of the utilization of each periodic task.

The starting point for the acceptance test to be launched in the DGManagers is always a server *request interceptor* (Figure 3). Whenever the application component provides an entry point, its requests are being monitored by an interceptor that invokes the DGManager if necessary. For the DGManagers that monitor components with no entry points, the invocation is made by other DGManagers (Figure 3). The execution time of DGManager and server *request interceptor* is accounted as an aperiodic local task managed by an aperiodic server.

When the client session time is over, the client identification is removed from the accepted client list, all the resources allocated for it are deallocated and for posterior call, the clients request will have to be submited to the acceptance test again.

6.1. Overhead Evaluation

Experiments were run to determine the overhead caused by the mechanisms in applications with different number of components. The execution environment is composed by a set of five Athlon 2.4GHz PCs with 512 Mb of memory, running Linux (kernel 2.6.12). These machines are connected by a 100Mbps switched Ethernet network. The component platform used is CIAO 0.5.3 (TAO 1.5.3).

For all the experiments we executed applications with different number of components (from 1 to 4 components). Each application component was deployed in a different node together with one DGManager component. The fifth node was used to launch clients requests.

The experiments measured: (i)the overhead resulting from the use the deadline partitioning and acceptance test algorithm previously described at the first clients request to the server and (ii) the overhead caused by the use of interceptors in the calls of an accepted client.

The objective of the first experiment was to evaluate the overhead caused by the mechanisms used by the model, sophisticated deadline partitioning and acceptance tests algorithms were avoided. This test is not tight in the sense that it may keep schedulable tasks from being accepted, but it can ensure that overload condition will not be reached. The graph in Figure 4 shows the mean of acceptance test response times and the corresponding standard deviation in microseconds (us) from 1000 invocations from different clients to applications with one to four components. In all the experiments the clients invocations were dispatched with a period of 1000 ms.

From the graph it is possible to observe the increase of the mean response time with the increase of the number of components in the application. The difference in the lattency experienced by the clients of these applications is of about 1ms higher for each component added to the application. Observing the difference of response times from the application with one component and the application composed by two components one could infer that difference is caused by: the communication between components and the deadline partitioning that inexist in the application with only one component. These experiments demonstrated that the architecture for dynamic guarantee implies in a cost of about 1ms per node in this configuration. It must be noticed that these measures are dependent of the algorithms used for deadline partitioning and acceptance test. Higher values would be reached for more sophisticated algorithms.

The graph in Figure **??** illustrates the mean response times of the applications using the mechanisms implemented for DGC model implementation and CIAO. For this experiment, 1000 invocations from a single client were dispatched. The overhead of the first client request is not illustrated in the graph as the 10% of lowest and highest values of response times were disregarded.

From the graph in Figure **??** it can be observed that the overhead caused by the interceptor in DGC in comparisson with CIAO is of about 200us. The overhead is caused by the verifications made at every client request at the server components interceptors.

7. Related Work

The Component Integrated ACE (Adaptive Common Environment) ORB, CIAO [16], is a QoS enabled CORBA 3.X CCM implementation built atop TAO, The ACE ORB. TAO is a C++ ORB that is compliant with most of the features and services defined in the CORBA 3.x specification which

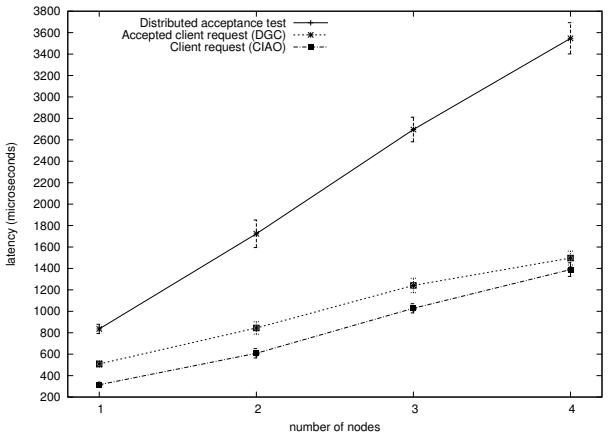

Figure 4. Acceptance test overhead.

includes the Real-time CORBA specification. CIAO, alone, does not support dynamic guarantee as it is not part of any specification that CIAO implements.

The Bulls-Eye Target Manager [11] is a resource provisioning service integrated to CIAO. It enables the retrieval of a list of available resources in a target domain, allocates resources for a particular deployment plan, releases resources when components are removed, obtains runtime resources available and updates resource consumption data. The Target Manager monitors a specified domain of resources and when over utilization is detected, it tears down the component application with the lower priority.

In the context of [11], overload results from the deployment of other applications (component assemblies) in the same domain (sharing the same resources). In our model, over utilizaton is a consequence from the amount of clients that may connect to the server application at runtime. Overload situations are avoided by controling the access of clients to the server instead of aborting one entire application.

An integrated architecture for managing dependencies in distributed component-based systems is proposed in [5]. The authors developed a resource management service that uses distributed monitors to acquire local status information and aggregates the information on a central server. Periodic updates are sent from local resource managers to the global resource manager that encompasses an instance of the standard OMG Object Trading Service. In the future, this work will be integrated to CCM, but currently it is implemented for CORBA objects.

The works described above centralize the mechanisms that actualy make a decision about accepting a new application to the system or tearing down lower priority applications. They apply a combination of local resource monitors to ensure the translation of information from the system domain heterogeneous platforms and a global resource monitor that will analyse the data for decision making. The model we propose uses an alternative approach for access control, it deals with the platform heterogeneity by implementing temporal protection [1] among the nodes and allowing each node to decide if it will be able to accomplish the request or not according to its own resources capacity.

Quality Objects (QuO) [18] is a framework for providing quality of service (QoS) in network-centric distributed applications developed by BBN. QuO can be used in conjunction to CIAO, as encapsulated units, to provide dynamic QoS and adaptability. It allows QoS constraints specification, system monitoring and adaptation to requirement changes at runtime.

In [13] the authors present HiDRA - Hierarchical Distributed Resource-management Architecture, a control-based multi-resource management framework. HiDRA employs feedback controllers to prevent over-utilization of processor and network bandwidth. The approach adopted in that work suits applications like target tracking systems, in which images of moving target(s) must be processed and the quality of them may vary according to the resources available, the application itself admits some flexibility so that it can cope with a smooth degradation in the quality of the image being transmited. HiDRA is built atop of TAO, meant for distributed objects and do not address component-based applications so far. The main difference between HiDRA and QuO and our work is that they assume that the client requirements are negotiable, which is not always the case. And both addresses applications whose requirements can be balanced to reach a predefined utilization resources. Our model by the other hand, ensures that, if a client obtains access to some service, there will be enough resources to respect its real-time constraints. If the access is not granted, the client could, for instance, execute some exception handling procedure until there are resources available or it could try to access other server.

In [17], an admission control for real-time component-based systems mechanism is presented. The clients must fit into one of the pre-defined categories of services. Each service category specifies the set of methods that can be invoked, the deadline for these methods and the arrival function that describes the maximum number of method invocations for an interval time. Different from DGC model, the admission control is centralized and the time for components comunication and the processing of admission control itself are not accounted.

8. Concluding Remarks

This paper described an architecture for real-time dynamic guarantee provision in component-based distributed applications. We presented the main elements that com-

poses the architecture and how it was implemented in an existing component-based platform. The architecture can be implemented in different technologies, other than CIAO components, since its mechanisms are based on specifications like CCM and CORBA portable interceptors. We presented an example of implementation in CIAO to illustrate the feasibility of the architecture and provided a glance at the overhead caused by it when used with applications composed by different number of components.

For the next steps, further investigation will complement this work. Two important issues must be considered: the execution time is very dependend on the environment in which it executes and components are assumed to be deployed in different environments. To acquire more accurate execution time for the application tasks, the **self inspection of real-time** behaviour must be provided. The container itself should obtain the components methods execution time at the end of the deployment step of the application.

We must also incorporate **priorities** in the architecture that would relate importance to tasks according to the entry point used by clients. According to the interface accessed, the requests would be treated with different priorities inside the server side. Clients with lower priorities would have the resources deallocated by the DGManagers when a higher priority client access the server.

It is also possible to use the architecture as a base for load balancing: Once a client does not pass the acceptance test it could be redirected to other server with a lower workload. The test would be combined with load balancing schemes in scenarios with replicated servers. QoS negotiation approaches could also be associated to the test so that rejected clients have other chances to be executed.

Acknowledgment: This work is supported by CNPq - The Brazilian National Counsel of Technological and Scientific Development and CAPES. We want to thank Alysson Neves Bessani for his essential contribution and Jeff Parsons and Jaiganesh Balasubramanian for all the help.

References

[1] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*. Springer, 2005.

[2] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale. Dance: A qos-enabled component deployment and conguration engine. In *Proc. of the 3rd Working Conference on Component Deployment*, Grenoble, France, November 2005.

[3] D. Isovic and C. Norstrm. Components in Real-time Systems. In *Proc. of the 8th International Conference on Real-Time Computing Systems and Applications*, Tokyo, Japan, March 2002.

[4] B. Kao and H. Garcia-Molina. Deadline Assignment in a Distributed Soft Real-Time System. *IEEE Transactions on Parallel and Distributed Systems*, 8:1268–1274, December 1997. No 12.

[5] F. Kon, T. Yamane, C. Hess, R. Campbell, and M. D. Mickunas. Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. In *Proc. of the 6th USENIX Conference on Object-Oriented Technologies and Systems* , pages 15–30, San Antonio, Texas, February 2001.

[6] A. Möller, M. Åkerholm, J. Fröberg, J. Fredriksson, and M. Nolin. Industrial Requirements on Component Technologies for Vehicular Control Systems. Technical report, Mälardalen Real-Time Research Centre Mälardalen University, February 2006.

[7] Object Management Group. Real-time CORBA Specification. OMG Document formal/02-08-02, August 2002.

[8] Object Management Group. Deployment and Configuration Specification. OMG Document ptc/03-07-08, 2003.

[9] Object Management Group. Light Weight CORBA Component Model. OMG Document realtime/03-05-05, May 2003.

[10] Object Management Group. Common Object Request Broker Architecture. OMG Document formal/2004-03-01, 2004.

[11] N. Roy, N. Shankaran, and D. C. Schmidt. A Resource Provisioning Service for Enterprise Distributed Real-time and Embedded Systems. In *Proc. of the International Symposium on Distributed Objects and Applications (DOA)*, Montpellier, France, November 2006.

[12] D. C. Schmidt and S. Vinoski. Object Interconnections: The CORBA Component Model: Part 1, Evolving Towards Component Middleware. *C/C++ Users Journal*, February 2004.

[13] N. Shankaran, X. Koutsoukos, C. Lu, D. C. Schmidt, and Y. Xue. Hierarchical control of multiple resources in distributed real-time and embedded systems. In *Proc. of the 18th Euromicro Conference on Real-Time Systems*, Dresden, Germany, July 2006.

[14] C. Y. Tatibana, R. S. de Oliveira, and C. Montez. Dynamic guarantee in component-based distributed real-time systems. In *10th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 19–22, Catania, Italy, September 2005.

[15] R. van Ommering, F. van der Linden, and J. Kramer. The Koala Component Model for Consumer Eletronics Software. *IEEE Computer*, March 2000.

[16] N. Wang and C. D. Gill. Improving Real-Time System Configuration via a QoS-aware CORBA Component Model. In *In Proc. of Hawaii International Conference on System Sciences (HICSS)*, Honolulu, Hawaii, January 2004.

[17] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao. Real-Time Component-Based Systems. In *Proc. of IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 428–437, San Francisco, CA, USA, March 2005.

[18] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1), 1997.