Dynamic Support for Nonfunctional Requirements in Distributed Applications

Ana Lúcia de Moura [1], Cristina Ururahy [2], Renato Cerqueira [3] and Noemi Rodriguez [4]
Computer Science Department – PUC-Rio
Rua M. S. Vicente 225, 22453-900
Rio de Janeiro-RJ, Brazil
{*ana,ururahy,rcerq,noemi*}*@inf.puc-rio.br*

PUC-RioInf.MCC34/01 October, 2001

Abstract: This work presents an environment that allows distributed applications to adapt dynamically to nonfunctional properties of their components. This environment, based on the programming language Lua and on CORBA, allows applications to select dynamically the components that best suit their requirements, to verify whether the system is satisfying these requirements, and to react, when appropriate, to variations in the nonfunctional properties of the services in use. The paper also describes an example application based on the proposed environment.

Keywords: middleware, dynamic adaptation, CORBA, components, nonfunctional requirements.

Resumo: Este trabalho apresenta uma ambiente que permite a adaptação dinâmica de aplicações distribuídas às propriedades não funcionais de seus componentes. Esse ambiente, baseado na linguagem Lua e em CORBA, permite que as aplicações selecionem dinamicamente os componentes adequados a seus requisitos, verifiquem se esses requisitos são atendidos ao longo do tempo, e reajam, quando necessário, a variações nas propriedades não funcionais dos serviços utilizados. O trabalho descreve também uma aplicação exemplo que utiliza o ambiente proposto.

Palavras-chave: *middleware*, adaptação dinâmica, CORBA, componentes, requisitos não funcionais.

[1] This author was supported by a grant from CAPES-Brazil
[2] This author is supported by a grant from CNPq-Brazil
[3] This author is supported by a grant from CAPES-Brazil
[4] This author is supported by a grant from CNPq-Brazil

1 Introduction

Component-based programming is nowadays an important trend in the development of distributed applications. One of the main characteristics of component systems is the strict separation between interface and implementation. The components of a distributed application implement specific services, offered through *functional interfaces*. Component systems (or middleware platforms), such as CORBA, DCOM, and Enterprise JavaBeans, provide the required abstractions and facilities for the definition of interfaces and the invocation of their operations.

Initially, middleware platforms were supposed to hide from their users (and applications) matters related to heterogeneity and, more generically, to specific details of the execution environment. This was in some way a heritage of the RPC approach, which aimed to make transparent to an invoker whether an invoked function was local or remote [BN84]. However, with the increasing demand for multimedia applications, collaborative systems, real-time applications, and with the growth of mobile computing, the scenario has changed. Distributed applications sometimes have very specific requirements about their execution coditions, possibly relating to bandwith, availability, or security. It becomes infeasible to handle these requirements in a transparent way inside the middleware platform, since uniform decisions would have to be taken for all kinds of applications. Conditions associated with nonfunctional requirements such as availability or performance often present variations along time, and applications must be able to determine how they can best adapt to changes in their execution environment.

In this context, many works have been proposing extensions to middleware platforms, more specifically, to CORBA, supporting dynamic adaption of distributed applications. These works have been using the concept of *Quality of Service* (*QoS*) in relation to any characteristics, or properties, that the application's execution environment may present (in contrast to the more restricted use of the concept by the computer network community). The proposed extensions allow distributed applications to adapt to variations on the *quality of the service* that their components provide [ZBS97, BG97, KK98].

An important issue is how to specify the reaction of an application to changes in its execution environment. This reaction may vary not only from application to application, but also along time. New services (with new interfaces, for example) may become available to replace a component that is not satisfying the requirements of a specific application anymore. The adaptation mechanism must be flexible enough to incorporate such variations [AW99].

In this work, we address the problem of dynamic evolution of execution conditions and applications, in the context of a research project that investigates the flexibility that an interpreted language can bring to component-based programming. This project has been in development for a few years at PUC-Rio, and has produced, as one of its major results, the LuaOrb system [Cer00]: The implementation of a mechanism for dynamic composition among component systems. The implementation of dynamic bindings between the Lua language [IFC96, FIC96], which this mechanism uses like a composition element, and the systems CORBA, COM, and Java allows us to incorporate into an application, on the fly, components from each of these systems. The CORBA binding, called *LuaCORBA*, is of particular interest to this work.

We have been exploring the flexibility that both the Lua language and LuaCORBA provide to implement a variety of dynamic configuration mechanisms for distributed applications based on components [RIC98, MRI99, RI99, BCR00]. As an extension to these works, the goal of the proposed environment is to offer a set of mechanisms to support the dynamic adaptation of distributed applications to the quality of the service that their components offer. These mechanisms allow applications to select dynamically the components that best suit their requirements; to verify the compliance with such requirements along the time; and to react, when appropriate, to changes in the quality of the service provided by the components currently in use.

In the next section we give an overview of the LuaCorba binding. Section 3 presents the architecture of the environment proposed in this work and describes the provided services. Next, in Section 4, we show an example of the use of the proposed environment. Finally, Section 5 contains some closing remarks.

2 LuaCorba

LuaCorba is the part of the LuaOrb system responsible for the binding between CORBA and the Lua programming language, an interpreted language developed at PUC-Rio [IFC96]. Lua is an *extension* language, implemented as a library. Through its API, it is possible to call Lua functions from C code, and also to register C functions so that Lua code can call them.

The CORBA standard [OMG99] defines an *Object Request Brocker* (ORB) that is responsible for the communication between client objects and distributed servers. The programmer describes each server interface in IDL, a technology-independent syntax. Typically, such descriptions are compiled to generate *stubs* for clients and *skeletons* for servers. The client program must simply be compiled and linked with its stub; the server program must implement the methods declared in the skeleton.

Such approach, commonly used in most CORBA bindings (specially in the most popular bindings for Java and C++), requires that we recompile the client whenever we change the server's interface or whenever we want to access a new type of server. The CORBA architecture provides two mechanisms to avoid recompilation: the *Dynamic Invocation Interface* (DII), which allows a client to invoke any operation with an argument list defined on the fly; and the *Dynamic Skeleton Interface* (DSI), which supports the implementation of objects that do not know, in advance (at compilation time), what interfaces they will implement.

LuaCorba uses the DII and DSI interfaces, along with other reflexive features of CORBA, and also dynamic and reflexive features of Lua, to create a dynamic binding [ICR98, CCI99, Cer00] for clients and servers.

Lua is a dynamically typed language: type checking takes place at run-time. There are no declarations of variables or functions in Lua. Objects (implemented by *tables*) are not associated with classes: Each object can have any methods and instance variables.

The client binding uses the DII, and represents CORBA objects in Lua programs through proxies. The result is that a CORBA client written in Lua uses a CORBA object in the same way it uses any Lua object: without declarations and with dynamic type checking.

A proxy is a common Lua object; LuaCorba uses *tag methods*, a reflexive mechanism of Lua, to change the default behavior of this object. When a Lua program makes a call to a proxy's method, the tag method intercepts this call and redirects it to LuaCorba; the binding then (dynamically) maps the parameters' Lua types to IDL, invokes the remote method, and maps possible results back to Lua.

We will now illustrate the use of LuaCorba proxies, using the IDL interface defined in Figure 1.

```
struct book {
   string author;
   string title;
};

interface library {
   boolean add_book(book abook);
   ...
};
```

Figure 1: IDL interface example.

The `lo_createproxy` constructor creates a proxy of a remote object that implements the `library` interface. This constructor must receive the name of the interface of the remote object and a reference (IOR) to it. Once we create the proxy, we can request the remote services as if we were invoking local methods, as shown in Figure 2.

```
a_foo = lo_createproxy(an_ior_string, "library")
a_book = {author="Mr. X", title="New Book"}
a_foo:add_book(a_book)
```

Figure 2: Accessing a CORBA object through a LuaCorba proxy.

On the server side, the LuaCorba binding exploits the DSI interface to allow the use of Lua objects as CORBA servers. In short, the idea behind DSI is to implement all the calls to one specific object, which we call here a *dynamic server*, through calls to one single routine, the *dynamic implementation routine* (DIR). This routine is responsible for unpacking the received arguments and for activating the appropriate code.

Each Lua object that acts as a CORBA server is represented by a LuaCorba *adapter* (a C++ object). This object handles all the requests delivered to the Lua object, implementing the DIR routine. The code that the DIR activates is the Lua method corresponding to the invoked operation.

We next present a simple Lua server, based on the IDL shown in Figure 3. In this example, the `generator` object generates numbers (e.g., random numbers) and calls, for each number it generates, the `put` method of its `listener` object.

```
interface listener {
   oneway void put(long i);
};
interface generator {
   void set_listener(listener r);
};
```

Figure 3: `generator` and `listener` IDL interfaces.

We can write a LuaCorba `listener` as shown in Figure 4. In the first assignment, we create a Lua object with a single method, called `put`. Next, we create a proxy for the generator, passing as an argument to `set_listener` the previously created Lua object. When LuaCorba detects that the formal parameter is a `listener` CORBA object, and that the real argument is a Lua object, the system automatically creates an adapter for this object, allowing this object to work as a CORBA server.

```
l = { put = function(self, i) print(i) end }

gen = lo_createproxy(gen_ior, "generator")
gen:set_listener(l)
```

Figure 4: Implementing a CORBA object in Lua.

We say that this way of creating CORBA servers is *implicit*, because the mere use of the Lua object as an argument determines the creation of the server. LuaCorba also supports the *explicit* creation of CORBA servers through function `lo_createservant` [CIR00].

3 An Environment for Adapting Distributed Applications Dynamically

The main goal of the environment we propose is to support the dynamic adaptation of distributed applications to the quality of service that their components provide. For this, the environment provides facilities and mechanisms that enable applications to:

- dynamically select components that best suit their requirements. We express these requirements as a set of *nonfunctional properties* that characterizes the quality of the services that these components provide.
- monitor the properties associated with these requirements, so as to verify whether the system satisfies them along the time.
- react to relevant changes in these properties through the activation of appropriate adaptation strategies.

To provide these facilities in a way that is transparent to the functional behavior of applications, we applied the *smart proxy* approach [ZBS97, BG97]. In this approach, a client application is not bound to specific servers, but instead accesses services through smart proxies. In our environment, the smart proxy provides a representation of a type of service and incorporates the transparent treatment of the *nonfunctional* requirements demanded by the application. The smart proxy encapsulates the stragegies for dynamic component selection, monitoring, and adaptation, allowing the code of the application to concentrate on functional issues. The same smart proxy can activate different components along time, trying to fulfill the application's requirements (Figure 5).

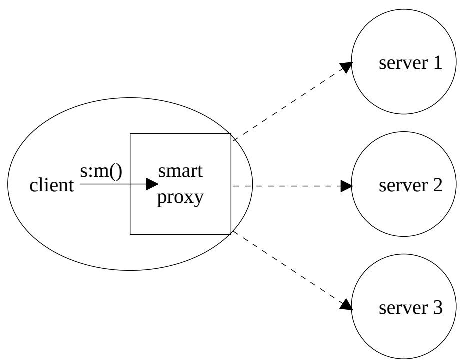

Figure 5: The smart proxy.

The Trading service [OMG00, HV99], defined by the CORBA architecture, is the basis of our support for dynamic selection of components. The Trading service is an important mechanism for obtaining dynamic information about available objects. By interacting with a trader, clients can obtain sets of references for implementations of given interfaces, and select, from this set, the ones that best suit their requirements. The selection is based on the characteristics, or *nonfunctional* properties, offered by these implementations. To facilitate the use of the Trading service in our environment, we developed a Lua library that provides a simplified interface to it, called *LuaTrading*. Due to lack of space, we will not describe the LuaTrading library in this work.

Besides locating objects that fulfill the application's requirements at a given time, a smart proxy must be able to verify and guarantee the maintainance of these requirements along time. To this end, it is essential to have a mechanism that allows monitoring the properties that characterize the quality of the represented service. Because we are interested in supporting the specific needs of any application, it is important for this mechanism to be configurable and extensible. In Section 3.2, we describe *LuaMonitor*, the extensible monitoring mechanism we included in our environment.

To make the use of the Trading service effective, we must guarantee that the trader has access to information about all available objects. In our environment, *service agents* are the elements responsible for announcing service offers to a trader. Besides managing the service offers of one or more server components, these service agents, typically implemented as Lua scripts, provide, when needed, the configuration of remote monitoring environments.

We illustrate the architecture of the proposed environment in Figure 6. In the next sections we describe briefly the facilities and mechanisms available in this environment. A comprehensive description of these mechanisms is available in [dM00].

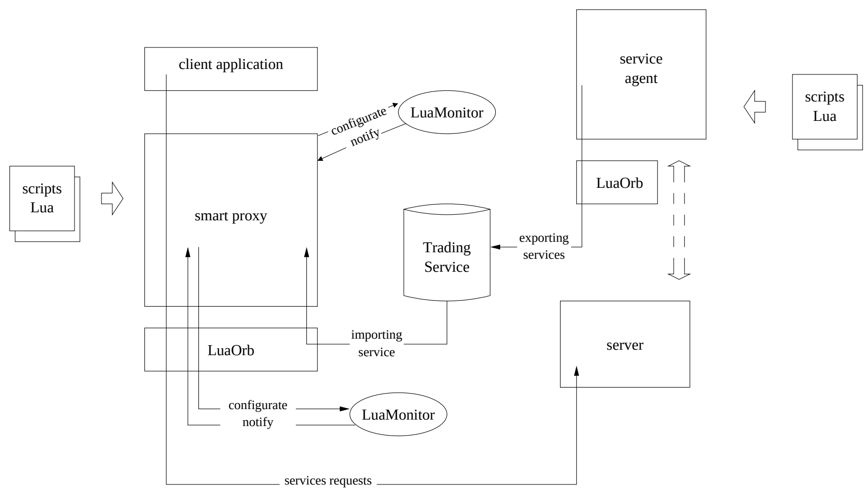

Figure 6: An Environment for Adapting Distributed Applications Dynamically.

3.1 The CORBA Trading Service

Basically, a CORBA *trader* object is a repository of *service offers*. A service offer contains a reference to a provider of a specific IDL interface and describes non-functional characteristics, or attributes, of this provider through a set of tuples <name,value> called *properties*. Each offer is associated with a *service type* description that contains an IDL interface and a property set. Descriptions of service types are stored in a *Service Types Repository*, also maintained by the trader.

Server objects publish their service offers interacting with a trader through *export* operations. When exporting a service offer to a trader, a server object informs the associated service type, the reference to the implementation of the corresponding IDL interface, and the values of the properties offered by this implementation. Besides the export operation, the trading service provides operations that allow modification and removal of service offers.

Clients obtain information about available services through *import* operations. In these operations, requirements may be specified using the *Trader Constraint Language*. This language allows the programmer to use boolean expressions that specify requirements on the property values of the available offers.

The Trading service defines an important concept for dynamic adaptation: *dynamic properties*. Instead of storing a constant value, a dynamic property stores a reference to an object that, when required, provides the trader with the current value of the property. Dynamic properties can reflect execution conditions that evolve dynamically (such as the size of a CPU ready queue) and, for this reason, have an important role in our work.

The LuaTrading library simplifies access to the *Trading* service by mapping the repositories that this service maintains into local Lua objects. Once a Lua object is initialized to reflect a repository, either of service offers or of service types, operations on this object are automatically reflected in the remote trader repository. The LuaTrading library is described in detail in [dM00], while the technique of using Lua objects as mirrors of remote CORBA services is discussed in [Nog01].

3.2 An Extensible Monitoring Mechanism

A number of requirements that distributed applications demand can be related to properties that evolve along the time. Therefore, effective adaptation strategies require that the applications be capable of not only selecting the adequate server, but also detecting and reacting to changes in the fulfillment of their requirements.

To allow the development of flexible monitors, capable of supporting arbitrary requirements and monitoring strategies, this work proposes a configurable and extensible monitoring mechanism, called LuaMonitor. As in [ZBS97] and [AW99], the LuaMonitor mechanism defines a special type of object — a *monitor* — that represents a specific observed property, such as the response time associated with an operation invocation over a server, or a policy adopted by a service provider.

The monitor object provides a generic interface that supports the definition of specific mechanisms for observation and control. This interface allows the specification of application requirements through a set of properties.

A predefined Lua object, `BasicMonitor`, implements the basic monitor functionality. It provides two methods (`getValue` and `setValue`), through which we can obtain and modify the current value of the represented property.

In many cases an application may be interested not only in the specific value of a property, but also in statistics or profiling the evolution of some condition. To accomodate these interests, we added to LuaMonitor facilities for defining and obtaining *aspects* and *qualifiers* of a property [FK98, GCS99]. We offer these facilities through the `AspectsManager` interface, defined in Figure 7. All monitor objects implement this interface. Because we implemented monitors as Lua objects, we can define the function that performs the evaluation of an aspect (as argument `updatedef` in method `defineAspect`) at run-time. This flexibility allows monitors to be dynamically extended according to user requirements. LuaCorba's facilities for implementing CORBA servers with Lua objects allowed us to implement this dynamic extensibility in LuaMonitor.

```
interface AspectsManager {
   PropertyValue getAspectValue(in Aspectname name);
   AspectList definedAspects();
   void defineAspect(in AspectName name, in string updatef);
};
```

Figure 7: `LuaMonitor::AspectsManager` Interface.

To facilitate the implementation of event-driven monitoring strategies, we defined an extension of the monitor object, based on the `Observer` [GHJV95] pattern. This pattern defines a relationship between a subject that represents the state of whatever it is modeling (for example, memory usage) and a set of observers that are interested in changes in this state. When the subject's state changes, it informs its observers that it has changed. In our context, an *event monitor* represents some state and the applications act as observers.

An event monitor implements interface `EventMonitor` shown in Figure 8. This interface allows observers to register themselves (through invocations to method `attachEventObserver`) as observers, specifying specific events in which they are interested. In this way, applications can be notified only when specific changes in the state occur, instead of at every modification in the value of the monitored property. In the monitor's implementation, an internal timing mechanism supports the generation of notifications. It triggers updates of the property value and activates the mechanism for event detection. The transfer of event detection to monitors allows a reduction in the number of iterations between smart proxies and these objects; this is particularly interesting in environments that use remote monitors.

```
interface EventObserver {
   oneway void notifyEvent(in EventID evID);
};
interface EventMonitor:BasicMonitor {
   EventObserverID attachEventObserver(in EventObserver obj,
                                       in EventID evID,
                                       in string notifyf);
   void detachEventObserver(in EventObserverID id);
};
```

Figure 8: `LuaMonitor::EventMonitor` Interface.

Again, we take advantage of the interpreted nature of Lua to define events through strings of Lua code, avoiding the need of previously defining the type of events that may be observed. Method `attachEventObserver` receives as parameters a reference to the observer object, an identification of the observed event, and a string containing the definition of the Lua function that determines whether the observed event has occurred.

The parameters that the event monitor passes to this function include the current value of the monitored property and a reference to the implementation of the `AspectManager` interface, through which we can obtain the values of any aspect of the observed property. The function will return true if the observed event has been detected. In this case, the event monitor will send a notification to the corresponding observer.

In Figure 9, we show the definition of property observer `CPUstat`. The observed property in this case represents the percentage of idle system time. Lua object `eventObserver` implements this observer and we register it in the property monitor (`mon`). The monitor will notify this observer if the current property value and its average go beyond the minimum defined boundaries.

```
eventObserver = { notifyEvent = function(self, event) ... end }

mon:attachEventObserver(eventObserver, "CPUstat",
  [[ function(self, currval, aspectmgr)
       return currval < 90 and aspectmgr:getAspectValue("Mean") < 80
     end
  ]])
```

Figure 9: Defining an Event Observer.

In several situations, the property that a monitor object represents, or a few of its aspects, could correspond to one or more dynamic properties associated with services offers registered with the trading service. To allow a single representation of this type of property, the LuaMonitor mechanism offers a function that dynamically incorporates an implementation of the interface `DynamicPropEval` to any monitor object. The implementation of this interface, which the Trading service defines, allows the trader to obtain, when needed, the current value of the represented property, and also its aspects.

3.3 Smart Proxies

We implemented the smart proxies of our environment as Lua objects that encapsulate LuaCorba proxies. The use of LuaCorba proxies to access server components implies that components with new types can be incorporated to applications dynamically. The programmer can define, in the smart proxy, the behavior that best suits the application requirements. Examples of such behaviors are: component substitution according to execution conditions, choice of different components for different requested operations, use of alternative methods, and management of control mechanisms. Moreover, the facilities that the Lua language offers allows the trivial implementation of service invocation interceptors, with the activation of the adequate adaptable behavior.

As we have said before, our smart proxy uses the CORBA Trading service to locate server components that not only implement the required functional interface, but also fulfill the *nonfunctional* requirements of the application. The programmer expresses these requirements in the smart proxy as constraints over a set of nonfunctional properties associated with the represented service type; thus, the smart proxy will retrieve service offers that fulfill the specified restrictions.

To verify and guarantee the fulfillment of the requirements along time, the smart proxy can observe and control the properties associated with these requirements. For this, it uses local or remote monitor services, which implement the LuaMonitor mechanism, described in Section 3.2. The use of this mechanism allows the environment to inform the smart proxies about relevant changes in the observed properties, and allows them to activate, when appropriate, the programmed adaptation strategies.

The programmer defines the relevant changes in the properties that a smart proxy observes as *events of interest* registered with the corresponding monitors. The detection of one of these events will cause the monitor to notify the smart proxy. To communicate with an event monitor, the proxy must implement the `EventObserver` callback interface, defined in Figure 8. When the smart proxy receives a notification, it can either immediately adapt, or register the new event for later treatment.

A smart proxy can also explicitly activate the adaptation strategies that it implements, independently of received events (e.g., through on-demand acquisition of the observed properties' values). Moreover, a single event may not be sufficient to completely determine the *state* of the current quality of service. For a final identification of this state, the treatment of an event can use other properties' values accessing the correspondent monitors.

The implementation of a smart proxy is specific not only to the represented service type, but also to the adaptation requirements and strategies of a specific type of client. Thus, it is up to the programmer or administrator of a distributed application to program the smart proxy, using the facilities the environment provides. To facilitate the construction of smart proxies, we offer a few pre-defined functions. As an example, function `createEventObserver` associates a new *event observer* to the mart proxy it receives as an argument. This standard event observer registers notified events in a global queue of events.

4 An Application of the Proposed Environment

This section describes a simple example we developed to validate the proposed environment. The example deals with load sharing among several servers that offer the same functional interface (we suppose these servers are stateless). Sharing the load among servers is the resposibility of clients: They dynamically locate the least loaded servers, and address their requests to them.

The idea of our example was derived from [BKKD99], which describes a system with similar characteristics. This system uses the CORBA Trading service to announce service offers associated with a dynamic property that represents the average response time that the servers present. The client uses this support to determine which server it is going to use. However, once it makes this selection, the system does not allow it to change servers. Thus, if the client-server interactions are long, the system may become unbalanced.

Using the environment proposed in this work, we developed a system similar to the one described in [BKKD99], but allowing dynamic changes of servers.

To evaluate the load at each server, we used two parameters: the server's average response time (`ResponseTime`), and the average load in a host system (`LoadAvg`). We developed monitors for each of these properties.

The first monitor registers a server's average response time to the requests its clients issue. This monitor was installed in the server's addressing space and makes use of specific LuaCorba facilities for intercepting invocations.

The second monitor executes as an independent process in the server machine and computes the average number of processes in the ready queue observed in the past one, five and fifteen minutes. In this second monitor, we defined an *aspect* that indicates tendencies in the load submitted to the system.

Figure 10 presents function `LoadAverageMonitor`, that creates a load monitor for a specific host[5]. Lines 3–10 contain the code that creates the event monitor. The code passed to `EventMonitor:new` first reads system information from `/proc/loadavg` and then returns a Lua *table* containing the values for the last one, five, and fifteen minutes. Lua tables are associative arrays, and can be created by a constructor such as the one in line 8. When a table constructor contains a simple list of expressions, these expressions are automatically associated to integer indices.

[5]This load monitor was developed for a Linux environment.

After creating the event monitor, function `LoadAverageMonitor` defines an aspect that indicates whether an increase in the load submitted to the system has occurred (lines 14–21). As discussed in section 3.2, method `defineAspect` receives as parameters the name of the new aspect and a string defining the Lua function that should be (repeatedly) invoked to compute the new aspect. This function, when called, receives as parameters the event monitor to which it is associated (`self`) and the current value of the monitored property (`currval`). In this case, as we just discussed, the value of the monitored property is a Lua table containing three positions. The Lua function defined in lines 16–21 tests whether the average load value in the last one minute was greater that the average load value in the last five minutes as a simple way to detect an increase in the load submitted to the system.

```
 1 function LoadAverageMonitor()
 2   local lmon
 3   lmon = EventMonitor:new("LoadAvg",
 4                         function()
 5                           readfrom("/proc/loadavg")
 6                           local nj1,nj5,nj15 = read("*n","*n","*n")
 7                           readfrom()
 8                           return {nj1,nj5,nj15}
 9                         end,
10                         60)  -- update values every minute
11
12   -- create an aspect that represents the tendency
13   -- to increase the load in the host
14   lmon:defineAspect("Increasing",
15                   [[function(self,currval)
16                       if currval[1] > currval[2] then
17                         return "yes"
18                       else
19                         return "no"
20                       end
21                     end]])
22   return lmon
23 end
```

Figure 10: Creating the `LoadAverage` Event Monitor.

We implemented the service agents of our application as a Lua script, which executes on the host systems of the server components. This script creates the monitors for properties `ResponseTime` and `LoadAvg`, and exports the service offers to the trader. This service offer will contain the two dynamic properties and two properties of type `Object Reference`, through which the mechanism provides smart proxies with references to the monitors.

Finally, we defined a smart proxy that uses our infrastructure to implement the load sharing. In general, smart proxy implementations are specific to some service type and to the adaptation policy that the client application adopts. However, we can apply the basic schema that we used to implement the smart proxy for this sample application to other situations. When a client application creates an instance of our load sharing smart proxy, the proxy constructor selects the server component that has the best performance and availability at the moment. It does this selection through a query to the trader, using the average response time of the server components as a sorting criterion for the service offers, and eliminating the components hosted on the system that show a tendency for load increase. If no offer suits the imposed restriction, the smart proxy issues an alternative query, where it specifies only offer sorting, and no filtering.

When the smart proxy selects a specific server component, it registers itself as an event observer for the response time monitor associated with the selected component. The monitor will then inform the smart proxy if a relevant performance takes place. We defined this event as an increase on the average response time larger than would be expected with two clients.

Figure 11 shows how the smart proxy registers the observer in the monitor.

```
function_code = format("function(self,value) return value > %d end",
                  (smart_proxy._response_time + (2 * ONECLIENT_RT)))
smart_proxy._rt_monitor:attachEventObserver(smart_proxy._observer,
                                    "ResponseTimeDecrease",
                                    function_code)
```

Figure 11: Registering an Observer.

The predefined Lua function `format` returns a formatted version of its variable number of arguments following the description given by its first argument, with rules similar to those of the `printf` function in standard C. In this chunk of code, `format` is used to create a string defining a function that is then passed as an argument to method `attachEventObserver`, defining the function that will detect the desired event.

When the smart proxy receives an event, it inserts this event in a queue and postpones its handling until the next service invocation. At a later moment, when the client application requests some service through the smart proxy, the proxy checks whether it has events to handle, and activates the adaptation strategy associated to each received event, immediately before redirecting the request to its represented component.

We show how we defined the strategy to handle `ResponseTimeDecrease` events in Figure 12. Smart proxies have a an attribute called `_strategies` that contains functions for handling events of interest. `_strategies` is a Lua table indexed by event names. In our case, this table contains a single field, since this smart proxy is interested only in the `ResponseTimeDecrease` event. To adapt itself to a performance decrease of the represented component, the smart proxy tries to locate a more adequate server component, performing a new query on the trader. The new component must have an average response time that, added to the load expected for one more client, is at least 20 percent less than that of the currently selected component (lines 7–9). The `_select` method performs the new query and replaces the current component for a new one (line 10). If this method does not find any component that suits the specified requirement, the smart proxy keeps the current component and relaxes the performance requirements over it (lines 11–17).

```
 1 smart_proxy._strategies = {
 2   ResponseTimeDecrease = function(self)
 3     -- get the current response time
 4     self._response_time = self._rt_monitor:getValue()
 5
 6     -- look for an alternative server
 7     local query
 8     query = format("ResponseTime < %d and LoadAvgIncreasing == 'no'",
 9                    (0.8 * self._response_time - ONECLIENT_RT))
10     if not self:_select(query) then
11        local f_code
12        f_code = format("function(self,value) return value > %d end",
13                        (self._response_time + ONECLIENT_RT))
14        self._rt_monitor:attachEventObserver(
15                  self._observer,
16                  "ResponseTimeDecrease",
17                  f_code)
18     end
19   end }
```

Figure 12: The adaptation strategy for `ResponseTimeDecrease` events.

It is worth noting that, as pointed out in section 3, the reconfiguration facilities are transparent to the applications' functional behavior, i.e., the reconfiguration code is not mixed with the application code. We can apply the reconfiguration solution described in this section to different applications and to components with different functional interfaces.

5 Final Remarks

The concepts and mechanisms that we used to support dynamic adaptation of applications to que quality of service of their components were based, to some extent, on a set of projects developed in the past few years. Specifically, we used many ideas from proposals of extensions to the CORBA architecture that deal with generic requirements of quality of service [ZBS97, BG97, KK98]. These proposals, however, are based on compiled languages and on extensions of functional interfaces. In their approach, applications must be recompiled to incorporate new services and requirements.

Our proposal, on the other hand, is based on an interpreted language, and on dynamic bindings of this language to component systems. [PLS+00] points out the similarity between the goals of *aspect languages* [KIL+96], used in QoS, and of *scripting languages*. That work also remarks that scripting languages generally lack adaptive and distributed characteristics necessary to the treatment of QoS. The Lua programming language and the LuaOrb system, however, have shown themselves adequate for use in this context, thus eliminating the need for defining a specific aspect language for our environment. In addition, the dynamism offered by the Lua language and the LuaOrb system allows our environment to not only support the dynamic evolution of applications, through the incorporation of new services, but also dynamically extend itself, through on-the-fly configuration and extension of its own mechanisms.

It is still necessary to verify the applicability of the proposed environment in more realistic environments, dealing with more complex applications with different requirements and adaptation strategies. This verification will allow us to analyze whether the benefits of introducing dynamic adaptation capability to distributed applications compensate for the probable performance loss that these applications will suffer.

Both Lua and LuaCorba implementations are open source, and you can obtain them through [Luaa] and [Luab] respectively. The implementation of the mechanisms that compose the environment described in this work can be obtained through contacting its authors.

References

[AW99] N. Amano and T. Watanabe. An Approach for Constructing Dynamically Adaptable Component-based Software Systems using LEAD++. In *OOPSLA'99 International Workshop on Object Oriented Reflection and Software Engineering*, November 1999.

[BCR00] T. Batista, C. Chavez, and N. Rodriguez. Dynamic Reconfiguration through a Generic Connector. In *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, Las Vegas, Nevada, USA, June 2000.

[BG97] C. Becker and K. Geihs. MAQS - Management for Adaptive QoS-enabled Services. In *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, CA, September 1997.

[BKKD99] E. Badidi, R. Keller, P. Kropf, and V. Dongen. The Design of a Trader-based CORBA Load Sharing Service. In *Proceedings of the Twelfth In-*

ternational Conference on Parallel and Distributed Computing Systems (PDCS'99), Fort Lauderdale, FL, August 1999.

[BN84] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Trans. on Computer Systems*, 2(1):39–59, February 1984.

[CCI99] R. Cerqueira, C. Cassino, and R. Ierusalimschy. Dynamic Component Gluing Across Different Componentware Systems. In *International Symposium on Distributed Objects (DOA'99)*, Edinburgh, Scotland, 1999. IEEE Press.

[Cer00] Renato Cerqueira. *Um Modelo de Composição Dinâmica entre Sistemas de Componentes de Software*. PhD thesis, Computer Science Department, PUC-Rio, Rio de Janeiro, Brazil, 2000.

[CIR00] R. Cerqueira, R. Ierusalimschy, and N. Rodriguez. *The LuaOrb Manual*, November 2000.
`http://www.tecgraf.puc-rio.br/luaorb/pub/luaorb.pdf`.

[dM00] Ana Lúcia de Moura. Um Ambiente de Suporte à Adaptação Dinâmica de Aplicações Distribuídas. Master's thesis, Computer Science Department, PUC-Rio, Rio de Janeiro, Brazil, 2000.

[FIC96] L. Figueiredo, R. Ierusalimschy, and W. Celes. Lua: an extensible embedded language. *Dr. Dobb's Journal*, 21(12):26–33, December 1996.

[FK98] S. Frolund and J. Koistinen. Quality of Service Specification in Distributed Object Systems Design. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98)*, April 1998.

[GCS99] A. Gomes, S. Colcher, and L. Soares. Um Framework para Provisão de QoS em Ambientes Genéricos de Processamento e Comunicação. In *Anais do XVII Simpósio Brasileiro de Redes de Computadores (SBRC'99)*, Salvador, Bahia, Brazil, May 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*, chapter 19, pages 827–921. Addison Wesley, 1999.

[ICR98] R. Ierusalimschy, R. Cerqueira, and N. Rodriguez. Using reflexivity to interface with CORBA. In *International Conference on Computer Languages 1998*, pages 39–46, Chicago, IL, 1998. IEEE, IEEE.

[IFC96] R. Ierusalimschy, L. Figueiredo, and W. Celes. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[KIL+96] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es), 1996.

[KK98] T. Kramp and R. Koster. A service-centred approach to QoS-supporting middleware, September 1998. Work-in-Progress Paper at the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98).

[Luaa] The Programming Language Lua. `http://www.lua.org/`.

[Luab] LuaOrb Online. `http://www.tecgraf.puc-rio.br/luaorb/`.

[MRI99] Marco Martins, Noemi Rodriguez, and Roberto Ierusalimschy. Dynamic extension of CORBA servers. In *Euro-Par'99 Parallel Processing*, pages 1369–1376, Toulouse, France, September 1999. Springer-Verlag. (LNCS 1685).

[Nog01] Letícia Nogueira. Um Ambiente de Gerência de Aplicações CORBA. Master's thesis, Computer Science Department, PUC-Rio, Rio de Janeiro, Brazil, 2001.

[OMG99] Object Management Group. *The Common Object Request Broker: Architecture and Specification v2.3.1*, October 1999. OMG Document formal/99-10-07.

[OMG00] Object Management Group. *Trading Object Service Specification*, May 2000. OMG Document formal/00-06-27.

[PLS+00] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration. In *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, Newport Beach, CA, March 2000.

[RI99] N. Rodriguez and R. Ierusalimschy. Dynamic Reconfiguration of CORBA-Based Applications. In *SOFSEM'99: 26th Conference on Current Trends in Theory and Practice of Informatics*, pages 95–111, Milovy, Czech Republic, 1999. Springer-Verlag. (LNCS 1725).

[RIC98] N. Rodriguez, R. Ierusalimschy, and R. Cerqueira. Dynamic Configuration with CORBA Components. In *4th International Conference on Configurable Distributed Systems*, pages 27–34, Annapolis, MD, 1998.

[ZBS97] J. Zinky, D. Bakken, and R. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.