Dynamic Extension of CORBA Servers

Marco Catunda, Noemi Rodriguez, and Roberto Ierusalimschy

Departamento de Informática, PUC-Rio
22453-900, Rio de Janeiro, Brazil
`catunda,noemi,roberto@inf.puc-rio.br`

Abstract. This paper describes LuaDSI, a system for implementing CORBA servers with the Lua scripting language. An object written in LuaDSI can be dynamically modified and extended without stopping its service. We also describe LuaRep, an extension to Lua which allows clients to have transparent access to CORBAs interface repository. In conjunction with LuaDSI, LuaRep allows new CORBA services to be dynamically defined and installed.

1 Introduction

The wide acceptance of the CORBA architecture has shown the importance of componentware as a current trend in software development. Currently, most CORBA bindings direct their design to support servers written in a statically compiled language, such as C++, using a statically compiled skeleton; although CORBA supports a dynamic interface for servers (DSI, the Dynamic Skeleton Interface), this interface is very low level and difficult to use. Static implementations are quite acceptable and even desirable in many cases, but they have shortcomings that can be serious in some contexts. First, static implementations make rapid prototyping, which is another important trend in current software development, difficult. Second, they make remote updates to servers very hard. Third, the updating of an existing service implemented as a static server usually requires its interruption for some period of time.

A scripting language brings an interesting design alternative to this static nature of CORBA components. Servers implemented with a scripting (and interpreted) language can be dynamically modified and extended without compiling or linking phases, and so, without interrupting their services. With an interpreted language, it is easy to send code across a network, which allows the system to do remote or interactive modifications and extensions to the server. An interpreted language greatly improves the support for rapid prototyping, as we can load and test new design alternatives for a system in a quick and simple way.

The OMG recognizes the relevance of scripting languages to manipulate CORBA components [6]. Although the primary use of such scripts is for CORBA clients, they must be able to handle events generated by the components. Usually, the callbacks to handle events are specified as methods of an object, a *listener*, that is passed to the component. Therefore, even a scripting language used only for writing client code should support the creation of server objects, to act as listeners.

Our work on dynamically extensible servers has been preceded by a study of the flexibility which a binding to an interpreted language can bring to a CORBA client. For that task we chose the scripting language Lua [4, 2]; Lua was developed at PUC-Rio in 1994, and since then it has been used in hundreds of places, both in Brazil and abroad; so it was a natural choice for us. We then implemented a binding between CORBA and Lua, called *LuaOrb* [5]. LuaOrb, which is based on CORBA's Dynamic Invocation Interface, allows clients to access any available CORBA server, independently of stubs; servers are represented by proxy Lua objects, and operations on these proxies map dynamically to remote method calls.

In this work we describe LuaDSI, a system which allows CORBA servers to be implemented in the language Lua. A server written in LuaDSI can be dynamically modified and extended without stopping its service; we can also change its interface without having to create and link new skeletons to the server. All these modifications are made through a CORBA interface, so that you can modify a dynamic server from any remote system with CORBA access; as a particular example, you can manage a server from a remote console running LuaOrb.

LuaDSI hides the complexity of using the DSI, CORBAs Dynamic Skeleton Interface. LuaDSI dynamically receives all calls to a server object, unmarshals the arguments according to their descriptions in the Repository Interface, calls the appropriate Lua method to handle the call, and marshals any results back to the ORB.

In this work we also describe LuaRep, an extension to Lua which allows clients to have transparent access to CORBAs interface repository. The Interface Repository, which contains interface definitions, acts as a regular CORBA object, and can be dynamically queried and updated. LuaRep facilitates these operations and, in conjunction with LuaDSI, allows new services to be dynamically defined and installed. If you need only to modify a server implementation, LuaDSI is enough. If you also have to modify the server interface, then you can use LuaRep for that. With both systems, you can have a console wherein you have complete control over a server.

2 The Dynamic Skeleton Interface and LuaDSI

The Dynamic Skeleton Interface is an interface for writing object implementations that do not have compile time knowledge of the type of the object they are implementing [7]. The basic idea of the DSI is to implement all calls to a particular object (which we will call a *dynamic server*) by invocations to a single upcall routine, the *Dynamic Implementation Routine* (DIR). This routine is responsible for unmarshalling the arguments and for dispatching the call to the appropriate code. To our knowledge, the most frequent application of DSI

```
class ServerRequest {
  public:
  Identifier op_name();
  OperationDef_ptr op_def();
  Context_ptr ctx();
  void params (NVList_ptr parameters);
  void result (Any *value);
  void exception (Any *value);
}
```

Fig. 1. `ServerRequest` interface

```
interface Hello {
  void Print (in string Hello)
}
```

Fig. 2. Hello interface.

has been to implement bridges between different ORBs [11]. In this context, the dynamic server acts as a proxy for an object in some other ORB.

To make a request to a dynamic server, the ORB invokes the corresponding DIR with a single argument, a `ServerRequest` object. Figure 1 shows the `ServerRequest` interface for C++: The `op_name` attribute identifies the method being invoked. Method `params` gets the list of parameters. The other methods are related to setting the invocation results and exception signalling, and to retrieving the context information specified in IDL for the operation.

To illustrate the use of the DSI in C++, we present a simple IDL interface, in Fig. 2, and a dynamic skeleton implementation for it, in Fig. 3. The C++ implementation of a dynamic server must inherit from class `DynamicImplementation`. This class implements the DIR through a method `invoke`, which must be reimplemented in each dynamic server implementation. In our example, the implementation first checks whether the called operation is `Print` (line 4); it should be, in any type correct invocation. Then it creates and initializes an `NVList` (lines 5–9), to get the operation arguments with correct types from the ORB (line 10). Finally, the code gets the only argument from the list (lines 11–12), converts it to a C++ string (lines 13–14), and prints it (line 15). Because C++ is statically typed and has no garbage collection, while the DSI is inherently dynamic, the code is quite complex.

Since DSI allows a server to implement requests to objects about which it has no compile time knowledge, it is natural to consider this mechanism as a basis for the dynamic extension of servers. This section describes LuaDSI, a facility which uses DSI to allow remote dynamic creation and updating of CORBA servers written in Lua. LuaDSI offers an IDL interface for server update. Servers with this interface are implemented as DSI servers, and are collections of Lua

```
class ImplHello: CORBA::DynamicImplementation {
  public:
  void invoke (CORBA::ServerRequest request) {
    if (strcmp(request->op_name(), "Print") == 0) {
       CORBA::NVList nv;
       orb->create_list(0, nv);
       CORBA::NamedValue *namedValue = nv->add(ARG_IN);
       CORBA::Any *any = namedValue->value();
       any->replace(CORBA::_tc_string, 0);
       request->params(nv);
       namedValue = nv->item(0);
       any = namedValue->value();
       char *str;
       (*any) >> str;
       printf("%s", str);
    }
  }
}
```

Fig. 3. A Dynamic Implementation Routine in C++.

```
interface LuaDSIObject {
  readonly attribute Object obj;
  void InstallImplementation (in string opname, in string luaCode);
}

interface ServerLua {
  LuaDSIObject Instance (in CORBA::InterfaceDef intf);
}
```

Fig. 4. `ServerLua` interface.

objects, each representing an instance of some IDL interface. A dynamically extensible server maps each request coming from the ORB (except requests for object instantiation) to an operation on the corresponding Lua object.

A LuaDSI dynamic object is encapsulated inside a `LuaDSIObject` interface, presented in Fig. 4. They are created through the interface `ServerLua`. To create a new instance at a dynamically extensible server, a client first invokes method `Instance`. The single parameter of this method is a reference to an interface definition in the interface repository (see Sect. 3). This reference is used by `Instance` to retrieve information about the attributes and methods of the new object. The new object is returned as the attribute `obj` inside a new `LuaDSIObject`. The manager client can then invoke the method `InstallImplementation` to install or modify each of the objects methods.

To illustrate the use of LuaDSI, we will again use the `Hello` interface presented in Fig. 2. Figure 5 presents a Lua chunk that installs an instance of

```
-- gets the desired interface from the Interface Repository
interfaceDefHello = IR:lookup("Hello")
-- creates a new LuaDSIObject
dsiobj = serverlua:Instance(interfaceDefHello)
-- installs a "Print" method
dsiobj:IntallImplementation("Print", "function (s) print(s) end")
newobj = dsiobj.obj            -- gets the object
newobj:Print("Hello World!")   -- uses it
```

Fig. 5. Creation of a new server in Lua.

this object. We assume that the user already has the bindings to `serverLua` and to the Interface Repository. The presented sequence of commands can be interactively issued from a simple LuaOrb console.

3 The Interface Repository and LuaRep

The interface repository (IR), defined as a component of the ORB, provides dynamic access to object interfaces. The IR is itself a CORBA object, and can thus be accessed through method invocations. In general, these methods can be used by any program, allowing the user, for instance, to browse through available interfaces. However, the IR is specially important for the dynamic interfaces of CORBA, DII and DSI. DII allows programs to invoke CORBA servers for which they have no precompiled stub. In order to build dynamic invocations, the program must possess information about available methods and their parameters; the interface repository provides this information. On the server side, DSI allows a server to handle requests for which it has no precompiled skeleton. Again, the correct signature of these requests must be obtained from the interface repository.

The possibility of dynamically updating the IR extends the flexibility obtained with LuaDSI. It allows a manager client to install not only new implementations for existing interfaces, but also unforeseen services in the server, by first adding their definitions to the interface repository.

Eight interfaces are defined in CORBA for interaction with the IR. The `Repository` interface represents the root object, the IR itself. The `ModuleDef`, `InterfaceDef`, `AttributeDef`, `OperationDef`, `TypedefDef`, `ConstantDef`, and `ExceptionDef` interfaces represent the definitions of a module, interface, attribute, operation, typedef, constant and exception, respectively. All these interfaces inherit from the `IRObject` interface, which contains operations `destroy`, for destroying an object in the repository, and `def_kind`, which identifies an object.

Although these interfaces provide any CORBA client with the possibility of querying and updating the repository interface, using them in a conventional binding, such as the one to C++, is quite complex. The goal of the LuaRep library is to simplify access to the IR. LuaRep makes extensive use of Lua data

```
-- creates an object representing the Hello Interface
HelloInt= CORBA_createinterface{
            Print = CORBA_createoperation{
                      result = CORBA_void,
                      params = {CORBA_string}}}

-- binds to the IR
repository = OM_BindRepository{}
-- Installs Hello Interface in the repository
repository.Hello = HelloInt
```

Fig. 6. Lua code for defining interface `Hello`.

description facilities, so as to describe with Lua structures the desired interfaces. When such a structure is "assigned" to a field of a repository, LuaRep automatically installs the described interface in the repository.

Basic IDL types are represented by Lua constants, and structured IDL types are represented by Lua constructors; the names of these constants and constructors are formed by prefixing the IDL type name with the string `'CORBA_'`. To make descriptions simpler, both parameter names and parameter modes are optional, with mode `PARAM_IN` as default. As an example, Fig. 6 contains the Lua code needed to publish the IDL interface `Hello` (Fig. 2):

To create a Lua object representing an IDL type, we use a constructor. The name of the constructor is again formed by prefixing the IDL type name, in this case with the string 'CORBA_create'. Execution of the first assignment in Fig. 6 creates a Lua object with the description on the `Hello` interface. Such object has a single field, named `Print`, to represent the single method of the interface. The method description is again given by a Lua object, with two fields, `result` and `params`, that represent the result type (`CORBA_void`) and the parameter types (a list with a single element, `CORBA_string`) of the method.

To navigate in a CORBA repository, the Lua program must first create a local reference to the repository object. This is done in the second assignment in Fig. 6; function `OM_BindRepository` returns an object that acts as a proxy for the IR. The fields of this proxy object are themselves objects that represent type definitions contained in the IR. Operations on this proxy object are transparently reflected on the repository itself. Thus, the last assignment in Fig. 6 updates the IR by creating a new "field" and adding to it the new `Hello` interface. After adding the `Hello` interface to the IR, the client is free to install an implementation for it in the extensible server.

The reflexivity implemented by LuaRep allows not only updates to the repository, but queries as well. Again, expressions written in Lua are automatically translated to operations on the IR. For instance, to know the type of the third argument of a method `Foo` from interface `Iface`, you just write

```
argtype = repository.Iface.Foo.params[3]
```

4 Final Remarks

Although the OMG's Request for Proposals for a CORBA Scripting Language [6] has generated several responses, none of them offers facilities similar to LuaDSI.

CorbaScript [8] supports the creation of server classes written in the language. This allows easy development of servers. However, you must write the complete class code, which must be type compatible with the IDL description, before it is bound to the ORB. You can neither use partial implementations for tests nor add new methods on the fly.

The Python proposal [3] also offers a conventional support for servers. Either you write the server as a complete class, as in CorbaScript, or you can use the raw DSI. For that, you must implement a function `invoke`, that must do an explicit dispatching and type conversions, like in our C++ example (Fig. 3).

The work presented in this paper is part of a larger project wherein we study the flexibility that can be brought to distributed componentware with the use of an interpreted language [5, 10]. The use of a standard CORBA mechanism, the DSI, allows a dynamically extensible server to be accessed by any client, either through stubs or through the DII. Although the CORBA specification cites support for "monitors that want to dynamically interpose on objects" [7] as one of the possible applications for DSI, we believe this work has the important function of showing how this can be done in practice.

When we started work on LuaDSI, our main goal was to dynamically implement and reimplement existing interfaces. Since the implementation of a dynamic skeleton must rely on queries to the interface repository, this led us to a deeper understanding of the IR, and to the idea of developing LuaRep. LuaRep made our working environment much more flexible than we had anticipated, since now the manager client can install and implement new interfaces. This can be done either interactively or through programmed scripts, and seems to be an important facility in the context of CORBA application management.

Currently, we are working on the application of the tools we described in network management. In the initial network management model [1], the network management application concentrated all statistics and tests, collecting only raw data from the management agents. However, this model is neither appropriate to the current size of networks nor necessary, since nowadays most of the managed devices have enough memory and CPU resources to carry on part of the management processing. Thus, it is interesting that the agents be programmed to include tests and statistics collection. However, since the management application typically retains the role of maintaining a global view, it should be able to modify agent behavior according to the global state of the network. Using the CORBA framework which we implemented, the management application would be able not only to change parameters and limits, but also to carry out complex modifications; as an example, the administrator using the management application could dynamically define new statistical functions to be applied by the agents on the raw data.

Although we have used the language Lua in this work, our approach can be applied to other languages with similar characteristics, such as Tcl or Python.

The LuaOrb system, which includes both LuaDSI and LuaRep, is available at `http://www.tecgraf.puc-rio.br/~rcerq/luaorb/`.

Acknowledgments

We would like to thank Renato Cerqueira, who implemented the main part of LuaOrb (the client side of the binding between Lua and CORBA); and LES (Laboratório de Engenharia de Software — PUC/Rio), which hosted the project. This work has been partially supported by CNPq (The Brazilian Research Council), as part of PROTEM III.

References

[1] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol, 1990. RFC 1157.

[2] Luiz H. Figueiredo, Roberto Ierusalimschy, and Waldemar Celes. Lua: An extensible embedded language. *Dr. Dobb's Journal*, 21(12):26–33, December 1996.

[3] GMD-Fokus. *CORBA Scripting with Python*, 1998. OMG Document: orbos/98-12-19.

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

[5] Roberto Ierusalimschy, Renato Cerqueira, and Noemi Rodriguez. Using reflexivity to interface with CORBA. In *IEEE International Conference on Computer Languages (ICCL'98)*, pages 39–46, Chicago, IL, May 1998. IEEE Computer Society.

[6] Object Management Group. *CORBA Scripting Language Request for Proposal*, 1997. OMG Document: orbos/96-06-12.

[7] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1998. revision 2.2.

[8] Object-Oriented Concepts, Inc. *CORBA Scripting Language*, 1998. OMG Document: orbos/98-12-09.

[9] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[10] Noemi Rodriguez, Roberto Ierusalimschy, and Renato Cerqueira. Dynamic configuration with CORBA components. In *4th International Conference on Configurable Distributed Systems (ICCDS'98)*, pages 27–34, Annapolis, MD, May 1998. IEEE Computer Society.

[11] N. M. S. Zuquello and E. R. M. Madeira. A mechanism to provide interoperability between orbs with relocation transparency. In *IEEE Third International Symposium on Autonomous Decentralized Systems (ISADS'97)*, pages 195–202, Berlin, Germany, April 1997. IEEE.