

OiL: An Object Request Broker in The Lua Language

Renato Maia¹, Renato Cerqueira¹, Ricardo Cosme²

¹ Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
Rua Marquês de São Vicente, 225 RDC – Rio de Janeiro – Brasil

² Grupo de Tecnologia em Computação Gráfica (Tecgraf)

Rua Marquês de São Vicente, 225 Belisário Velloso – Rio de Janeiro – Brasil

{maia,rcerq}@inf.puc-rio.br, rcosme@tecgraf.puc-rio.br

Abstract. *In this paper, we present OiL, a small and efficient CORBA implementation completely developed in the interpreted language Lua. OiL was designed to fit a wide range of platforms, from server machines to PDAs and mobile phones, and provide a flexible implementation for experimentation in different applications scenarios. Although it is an ongoing project, OiL is already used in a couple of research and industrial projects.*

Resumo. *Neste artigo, apresentamos OiL, uma implementação de CORBA pequena e eficiente totalmente desenvolvida na linguagem interpretada Lua. OiL foi projetado para caber numa grande variedade de plataformas, desde máquinas servidoras a PDAs e telefones celulares, além de oferecer uma implementação flexível adequada a experimentação em diferentes cenários de aplicação. Apesar de ser um projeto em andamento, OiL já é utilizado em alguns projetos de pesquisa e comercialmente.*

1. Introduction

OiL (ORB in Lua) is an implementation of the CORBA specification [OMG 2002a] in the Lua scripting language [Ierusalimschy 2003]. OiL shares some of the main characteristics of Lua, like simplicity, small footprint and efficiency. Besides that, the dynamic nature of a scripting language like Lua enables a more flexible implementation of the CORBA specification. Another important feature of OiL is portability because the Lua interpreter is entirely written in ANSI C, so it can be compiled in virtually any platform, from microcontrollers [Marinescu 2005] and cellular phones to PCs and mainframes.

As a matter of fact, OiL is a result of a long-term research on the use of a scripting language in the development of flexible middleware. Previously, we developed the Lua-Orb [Cerqueira et al. 1999] system that enables the use of Lua to manipulate components of different models, including CORBA objects. However, LuaOrb was build on top of an off-the-shelf C++ ORB that constrained the flexibility of LuaOrb applications in some ways. As an alternative approach, we developed the OiL ORB from the scratch using the Lua language with the purpose of using it as a base platform for experimentation with dynamic adaptation models, middleware implementation techniques as well as middleware for mobile and embedded systems.

Currently, OiL is been used as a tool in various research projects on dynamic adaptation [Maia et al. 2005], security [Theophilo et al. 2005], grid computing [Goldchleger et al. 2004, Lima et al. 2005] and scheduling policies [de Mello 2005].

However, OiL also presents some characteristics that promote its use in industrial projects. For example, a couple of applications for management of heterogeneous clusters of the Brazilian Oil Company (PETROBRAS) also make use of the OiL ORB due to the easy of deployment over the various platforms that compose the clusters.

2. Overview

Similarly to other ORBs, OiL manipulates messages encoded in the CDR (Common Data Representation) format that are sent and received through sockets channels according to the GIOP (General Inter-ORB Protocol) defined by CORBA [OMG 2002a]. All this is almost completely implemented in Lua. Only socket and bit manipulation support is provided by fairly portable C libraries.

However, unlike other CORBA implementations, OiL does not provide ORB support by means of generated glue code such as stubs and skeletons. Instead, all support is created at runtime, including method invocation and dispatching. This simplifies enormously the development process by removing compilation steps. Additionally, this dynamically created infrastructure is more flexible than statically generated ones like in generated code approaches, providing a better support for prototyping exploratory programming, and dynamic adaptation.

2.1. The Lua Language

OiL is a Lua ORB, therefore OiL applications are typically written in Lua. However, OiL can be embedded into applications written in different languages, such as C++ and Java because Lua is a language devised to be embedded into applications.

Although Lua is not an object oriented language, it provides mechanisms for definition of object-like structures. An object in Lua is a table (*i.e.* an associative array) that maps the names of attributes to their current values and name of methods to the functions that implement them. This is possible because Lua functions are first-class values. The method implementation functions receive an implicit parameter (named `self`) that is the table that represents the object. Additionally, Lua provides syntactic sugar to ease this object-oriented programming style. For illustrative purposes consider the implementation shown on listing 1 that provides an object (lines 1-10) for registration and retrieval of user messenger objects (lines 12-13) that are used to send messages to users.

Listing 1. Object oriented Lua application.

```
1 CentralServer = { users = {} }
2 function CentralServer:register(name, messenger)
3   self.users[name] = messenger
4 end
5 function CentralServer:unregister(name)
6   self.users[name] = nil
7 end
8 function CentralServer:isonline(name)
9   return self.users[name]
10 end
11
12 Messenger = {}
13 function Messenger:show(message) print(message) end
```

2.2. Interface Definition

In order to build up the GIOP message correctly, it is necessary to know precisely the operations and attributes available in each object interface. This information is available

as an internal interface repository that contains data structures describing IDL definitions. Code on listing 2 shows the description of an interface definition created with IDL description constructors provided by OiL.

Listing 2. IDL description in Lua.

```
1 hellointerface = oil.idl.interface{
2   name = "Helloiface",
3   members = {
4     hello = oil.idl.operation{
5       result = oil.idl.string,
6       parameters = {{mode = "PARAM.IN", type = oil.idl.string, name = "name"}},
7     },
8   },
9 }
```

Conveniently, OiL provides a compiler that translates IDL specifications into Lua data structures and registers them in the OiL internal interface repository by means of operations `oil.loadidl` and `oil.loadidfile`.

Alternatively, OiL also may retrieve interface definitions from a remote Interface Repository that is defined using the operation `oil.setIR`. In that case, a proxy to the remote Interface Repository must be supplied. See section 2.3. for information on creating proxies for remote objects.

2.3. Distributed Objects

CORBA servants are created using the `oil.newobject` function that receive a Lua object, which provides the implementation of the object operations and attributes (*i.e.* a table containing functions for operations and values for attributes), and the name of the IDL interface supported by the servant. All method dispatch is done accordingly to this interface. Whenever a new request is addressed to a particular servant, the associated interface definition is used to retrieve the operation signature that is then used to figure out the actual kind of data marshaled in the message. Code on listing 3 shows a script that creates a CORBA object from the `CentralServer` object defined on listing 1.

Listing 3. Creation of CORBA objects with OiL

```
1 require "oil"
2
3 oil.loadidl [[
4   interface IMessenger {
5     void show(in string msg);
6   };
7   interface IServer {
8     void register(in string user, in IMessenger msgr);
9     void unregister(in string user);
10    IMessenger isonline(in string user);
11  };
12 ]]
13
14 oil.writeIOR(oil.newobject(CentralServer, "IServer"), "server.ior")
15
16 oil.run()
```

Call to function `oil.loadidl` (line 3) loads IDL definitions compatible to the interface provided by objects defined on listing 1. Function `oil.writeIOR` (line 14) is an auxiliary function used to write the IOR of a servant into a file specified as a second argument. Call of function `oil.run` at the end of the script starts the processing of

requests by the OiL ORB. Therefore, every request to CORBA servants will be processed and dispatched to its proper implementation.

All method invocation is done by object proxies, which work like dynamic stubs that perform method invocations accordingly to the interface definition provided. Object proxies are created when an object reference is received as the return value or output parameter of a method call. Proxies can also be created from textual references like stringfied *IOR* or *corbaloc* references by the use of `oil.newproxy` function. An object proxy behaves like the remote object, so all method invocations and attribute manipulation made through the object proxy is forwarded to the actual remote object. Similarly to method dispatch, all method invocation is done accordingly to interface definitions stored at the OiL internal interface repository. Code on listing 4 illustrates how to create a proxy for the remote object created on listing 3 using the reference stored in file `server.ior`.

Listing 4. Use of CORBA objects through object proxies.

```
1 require "oil"
2
3 oil.loadidfile("ifaces.idl") — same IDL on listing 3
4
5 local server = oil.newproxy(oil.readIOR("server.ior"), "IServer")
6
7 server:register("Me", Messenger)
8
9 local user = server:isonline("John Doe")
10 if user
11 then user:show("Hello there!")
12 else print("User 'John Doe' is offline")
13 end
14
15 oil.run()
```

Call of function `oil.readIOR` (line 5) reads the contents of file `server.ior` and return it as the first argument of function `oil.newproxy` that creates a proxy for the remote object with the interface `IServer`. The definition of the proxy interface may be omitted. In that case, the interface used is the interface specified by the IOR or the interface supplied by the operation `get_interface` of CORBA objects.

It is worth noticing that OiL allows the use of simple Lua objects as argument of method invocations that take object parameters, like the use of object `Messenger` in invocation of method `register` of interface `IServer` (line 7). In that case, the Lua object is implicitly used as the implementation of a CORBA object that is used as the actual parameter of the invocation. Finally, operation `isonline` (line 9) is used to retrieve the messenger object for user `John Doe` that is then used to send a message to that user (line 11).

2.4. Cooperative Concurrency

By default, OiL process one request at time. This mean that while a request is pending no other request is processed. This can lead to many undesired situations that may freeze the application. To avoid such situations, OiL provides the ability to process different requests concurrently using the native support for co-routines in Lua. This enhances portability because Lua co-routines are implemented inside the virtual machine that is entirely written in ANSI C. Lua co-routines are used to create independent execution threads that switch execution at explicitly defined points. This kind of multiprogramming is called cooperative concurrency. Differently from preemptive models, synchronization between different

threads is much simpler in cooperative models. However, fairness assurance is almost entirely handled by the programmer.

In order to enable OiL concurrency support, it is necessary to load an additional package of the OiL distribution package, called `scheduler`, before loading the OiL package. This package implements the co-routine scheduler that controls the execution of OiL co-routines. When using OiL in concurrent mode, some constraints must be taken into account: first, every OiL code must be executed by a co-routine registered in the scheduler; and second, the scheduler main loop must be executed continuously by the function `scheduler.run()` or periodically by calls to function `scheduler.step()`. As an example, consider the code on listing 5 that is equivalent to code on listing 4 but with concurrent support enabled by the use of package `scheduler` (line 1). In this case, all OiL communication is performed inside a co-routine created with function `scheduler.new` (line 6). Additionally, OiL main loop is also performed by a different co-routine created in line 16.

Listing 5. OiL application with concurrent support

```
1 require "scheduler"
2 require "oil"
3
4 oil.loadidfile("ifaces.idl") — same IDL on listing 3
5
6 scheduler.new(function() — create new co-routine to execute client code
7   local server = oil.newproxy(oil.readIOR("server.ior"), "IServer")
8   server:register("Me", Messenger)
9   local user = server:isonline("John Doe")
10  if user
11    then user:show("Hello there!")
12    else print("User 'John Doe' is offline")
13  end
14 end)
15
16 scheduler.new(oil.run) — create co-routine to process requests
17
18 scheduler.run() — run scheduler main loop
```

Every concurrent task performed by OiL, including method dispatching, is done by a co-routine that explicitly switch execution at well known points, avoiding potential race conditions. Cooperative concurrency has many advantages, like easier programming and debugging, since thread synchronization is often trivial and execution is much more deterministic. Additionally, cooperative concurrency can be more efficient than preemptive models. This is mainly because the developer is able to program execution switching properly, avoiding unnecessary synchronization calls and undesired execution switching that leads to inevitably inefficient situations when using some automatic scheduling policy like those adopted in preemptive models. On the other hand, cooperative models of concurrency may not cope very well to some requirements. For example, execution switching between independently developed objects (or components) may be difficult to be achieved fairly. However, besides the current limitations of cooperative models, OiL concurrency model simplify enormously the complexity of development resulting in more efficient and maintainable applications.

2.5. Dynamic Adaptation

The ability to change an application implementation without stopping is called dynamic adaptation. Since Lua is a dynamic language, it is easy to change Lua implementations

dynamically, including OiL. Additionally, cooperative concurrency copes well with dynamic adaptation. This is due mainly by the almost absence of race conditions, since the programmer defines the execution switching points. Therefore, the programmer can define the points where adaptations can take place, avoiding potential inconsistencies. The use of a cooperative concurrency model in the implementation of OiL enables us to easily create simple mechanisms for experimenting with dynamic adaptation, because adaptation operations are always atomic, as long as they do not switch execution. For example, we can define an operation that changes an object implementation and its respective interface atomically with no need for synchronization mechanisms.

Any CORBA object created with OiL can change its implementation dynamically by the replacement of its functions (*i.e.* method implementations) so any future request received by OiL is dispatched using the new implementation. Besides that, OiL can adapt itself accordingly to changes on interface definitions, so if a method needs an additional parameter, its interface and implementation can be changed to reflect that necessity. For example, consider the messenger application described earlier, now suppose that we want to add an operation to the `IMessenger` interface for transferring files. In that case, besides adding the operation `store` to the `IMessenger` interface, we also need to change the implementation of the `Messenger` object. Code on listing 6 shows the script used to adapt the `Messenger` object implementation and interface.

Listing 6. Adaptation script for object Messenger.

```

1 — update interface definition
2 oil.loadidl [[
3   interface IMessenger {
4     void show(in string msg);
5     boolean store(in string file , in sequence<octet> data);
6   };
7 ]]
8 — add function to object implementation
9 function Messenger:store(file , data)
10  local file = io.open(file , "w")
11  if file then
12    file:write(data)
13    file:close()
14  return true
15 end
16 end

```

OiL currently provides very basic support for dynamic adaptation. However, we are planning to extend that support. Particularly, we are using it as a platform for experimentation on different mechanisms and aspects pertaining to dynamic adaptation [Maia et al. 2005] that may be incorporated into newer versions.

3. Final Remarks

In this paper we presented OiL, an ORB implemented in the Lua scripting language. Despite of its dynamic and interpreted nature, the current implementation of OiL presents promising performance results that measures overheads around 20% when compared to high-performance CORBA implementations in C++ [Maia et al. 2005]. The latest version of OiL can be found at <http://oil.luaforge.net>.

Traditionally, scripting languages are underestimated due to poor performance when compared to compiled languages. However, when it comes to the realms of distributed applications the overhead introduced by the use of such languages gets less evident.

Additionally, scripting languages are generally dynamic and usually provide introspection mechanisms for checking runtime conditions, which is particularly useful for distributed applications where errors at runtime are common due to failures on deployment.

Scripting languages are also suitable to make programming easier by the use of higher levels constructs and semantics. OiL makes use of this and simplifies the development of distributed applications by providing a simple IDL type mapping, removing the necessity of a compilation process for generation of supporting code and creating servants implicitly from objects whenever necessary.

OiL turns out to be a good platform for experimentation due to its simple and flexible implementation. For example, it was used as a platform for evaluation of different techniques for firewall/NAT transversal in CORBA applications as presented in [Theophilo et al. 2005]. Currently, we are using OiL as a platform for experimentation on dynamic adaptation because we claim that the use of a scripting language simplifies the development of such mechanisms due to its dynamic nature. Additionally, Lua data description and extension facilities are particularly useful for implementation of programming abstractions. Basically, one can trivially create data structures for describing object classes, interceptors or component ports just like the description of IDL interfaces presented previously. Besides that, Lua extension features are provided by means of computational reflection and therefore are implemented using the very same mechanisms available in the language. This way, there is no need to change the language implementation to introduce new abstractions.

Aside from its usage in research areas, OiL has proving a useful tool in real projects as well. The reason for this is its portability and small footprint. The InteGrade project is an object-oriented grid middleware that focus on the use of idle computing power of the nodes for opportunistic computing [Goldchleger et al. 2004]. One of the concerns of the InteGrade middleware is to minimize the amount of resources required for deployment on the nodes of the grid to reduce impact on its usage. Therefore, in order to reduce memory consumption, the node management component of the InteGrade middleware is implemented using OiL.

Other important use of OiL is in development of management tools for high-performance heterogeneous clusters of the Brazilian Oil Company. In this case, the choice for the OiL ORB is justified by the simplicity of deploying OiL and its libraries in the various platforms (*e.g.* Microsoft Windows, Sun Solaris, IRIX Unix, AIX Unix and various Linux flavors) that compose the clusters of hundreds nodes resulting in a much simpler deployment process when compared to deployment of Java Virtual Machine and ORB.

Although we got some initial results, OiL is still an ongoing project. Mainly, we plan to extend its features to support more wide spread usage. Currently, OiL does not implement all the CORBA specification and we are very selective about the features that are incorporated. Basically, we intend to evaluate different approaches for the problems addressed by CORBA that can make use of the features of the Lua language in order to keep OiL as simple and efficient as possible (*e.g.* POA policies).

Currently, we are restructuring the OiL internal implementation prior to make it more flexible. One reason for this is to add support for other distributed object protocols like ICE [ZeroC 2005], SOAP and XML-RPC.

OiL adaptation features are very low level and lack mechanisms for adapting the application properly. However, we are working on extensions that provides more elaborated abstractions that impose alternative programming paradigms to facilitate the implementation of dynamic adaptations, like the abstractions based on the CORBA Component Model [OMG 2002b] we previously implemented with the LuaOrb as presented in [Maia et al. 2004].

Referências

- Cerqueira, R., Cassino, C., and Ierusalimschy, R. (1999). Dynamic component gluing across different componentware systems. In *Proceedings of DOA'99*, pages 362–373, Edinburg, Scotland. IEEE Press.
- de Mello, R. X. (2005). Um modelo de escalonamento colaborativo de eventos baseado em corrotinas. Master's thesis, Catholic University of Rio de Janeiro, PUC-Rio, Rio de Janeiro, Brazil. (in Portuguese).
- Goldchleger, A., Kon, F., Goldman, A., Finger, M., and Bezerra, G. C. (2004). Inte-Grade: Object-oriented grid middleware leveraging idle computing power of desktop machines. *Concurrency and Computation: Practice and Experience*, 16:449–459.
- Ierusalimschy, R. (2003). *Programming in Lua*. Lua.org.
- Lima, M. J., Melcop, T., Cerqueira, R., Cassino, C., Silvestre, B., Mery, M., and Ururahy, C. (2005). Csgriid: um sistema para integração de aplicações em grades computacionais. In *Proceedings of the 23th Brazilian Symposium on Computer Networks*, Porto Alegre, Brazil. Brazilian Computing Society. (in Portuguese).
- Maia, R., Cerqueira, R., and Kon, F. (2005). A middleware for experimentation on dynamic adaptation. In *Proceedings of the Workshop of Reflective and Adaptive Middleware 2005*, Grenoble, France. ACM Press.
- Maia, R., Cerqueira, R., and Rodriguez, N. (2004). An infrastructure for development of dynamically adaptable distributed components. In Meersman, R. and Tari, Z., editors, *Proceedings of DOA'04*, volume 3292 of *Lecture Notes in Computer Science*, pages 1285–1302, Agya Napa, Cyprus. OTM 2004, Springer-Verlag Heidelberg.
- Marinescu, B. (2005). ReVaLuaTe. <http://www.circuitcellar.com/renesas2005m16c/winners/1685.htm>. (visited in 09/02/2006).
- OMG (2002a). *Common Object Request Broker Architecture: Core Specification - Version 3.0*. Object Management Group, Needham, EUA. document: formal/2002-12-06.
- OMG (2002b). *CORBA Components*. Object Management Group, Needham, EUA. document: formal/2002-06-65.
- Theophilo, A., Endler, M., and Cerqueira, R. (2005). Evaluation of three approaches for CORBA firewall/NAT traversal. In *Proceedings of DOA'05*. Springer-Verlag Heidelberg.
- ZeroC (2005). ZeroC - the internet communication engine. <http://www.zeroc.com/ice.html>. (visited in 09/02/2006).