

A Middleware for Experimentation on Dynamic Adaptation *

Renato Maia[†]
Department of Informatics
PUC-Rio, Brazil
maia@inf.puc-rio.br

Renato Cerqueira
Department of Informatics
PUC-Rio, Brazil
rcerq@inf.puc-rio.br

Fabio Kon
Depart. of Computer Science
University of São Paulo, Brazil
kon@ime.usp.br

ABSTRACT

This paper presents OiL, an adaptive middleware that aims at supporting experimentation with different models and techniques for dynamic adaptation of distributed systems. OiL is a CORBA implementation written completely in Lua, an interpreted language with several reflective and data-description facilities. In addition to the support for dynamic adaptation, OiL was designed to be deployed on a wide range of platforms, from server machines to PDAs and mobile phones, enabling experimentation in different applications scenarios. In this paper, we report the results already achieved in this project and discuss future directions.

Categories and Subject Descriptors

D.1.3 [Software]: Programming Techniques—*Concurrent Programming, Distributed Programming*

Keywords

Reflective Middleware, Adaptive Middleware, Dynamic Adaptation, Programming Abstractions, CORBA

1. INTRODUCTION

The ability to adapt computer systems without interrupting the provided services is an ever-growing necessity in many different areas of Computer Science. Indeed, almost every computer system provides means to be changed on the fly, considering that both data and instructions are usually stored in memory that can be modified. However, modifications at that level can become extremely complex and error prone in many practical situations. Therefore, current research on dynamic adaptation is more concerned with mechanisms that provide means to introduce these changes in a simple, organized and safe way.

*This work is supported by Tecgraf/PUC-Rio and CNPq, Brazil, process #55.0094/2005-9.

[†]Sponsored by a CNPq doctoral fellowship

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ARM'05 November 28–December 2, 2005 Grenoble, France
Copyright 2005 ACM 1-59593-270-4/05/11 ...\$5.00.

Although many approaches have been proposed to deal with dynamic adaptation, it is unclear how to compare these approaches and in which situations one approach is better than another. Such comparison requires a better understanding of dynamic adaptation and its underpinnings, what demands more experience with dynamic adaptable systems and the mechanisms necessary to support them.

We have been involved for some years in investigating abstractions and programming tools to develop dynamically adaptable component-based applications. Initially, we developed the LuaOrb system, which uses a scripting language as a unifying compositional language, wherein one can write code at runtime that freely uses and mixes components from different component systems, such as CORBA, COM, and Java [2]. LuaOrb uses the scripting language Lua [6], which is a very dynamic language with several reflective and data-description facilities.

Based on LuaOrb, we have already investigated some higher-level mechanisms to support dynamic adaptation, such as a dynamic extension of CORBA servers [10], smart proxies and an extensible distributed monitoring mechanism [12], and a dynamic adaptable container for the CORBA Component Model (CCM) [9]. We have also evaluated [9] the use of the *role* and *protocol* abstractions [14] to adapt CCM-based applications. To evaluate these tools and mechanisms, we developed some experiments in different application areas, such as Collaborative Computer Aided Design [3], Distributed Visualization [4], and Ubiquitous Computing [15].

A lesson learned from all these experiments was that the current middleware technology already provides several mechanisms to support dynamic adaptation at the application level, but more suitable abstractions and programming tools are fundamental to help using and understanding these mechanisms, and therefore enabling the development of dynamic adaptable applications [1]. However, in these experiments we could not exploit adaptations at the middleware level, since LuaOrb is implemented using CORBA's DII (Dynamic Invocation Interface) of an off-the-shelf C++ ORB, which typically does not provide means to dynamically adapt the middleware. In many cases, this was a strong limitation to the exploration of alternative adaptation strategies.

In this paper, we present OiL (*ORB in Lua*), an adaptive CORBA implementation completely developed using Lua that replaces the underlying middleware implementation used in our research. OiL aims at achieving, at the middleware level, the same degree of flexibility provided by LuaOrb to implement adaptation mechanisms at the application level. In addition to the support for dynamic adap-

tation, due to the small footprint of Lua, we were able to design OiL to fit a wide range of platforms, from server machines to PDAs and mobile phones, enabling experiments in different application scenarios.

This paper is organized as follows: Section 2 describes the main characteristics of OiL. Section 3 presents an example of how to implement a higher-level adaptation abstraction over OiL. Then Section 4 presents some results already achieved with OiL and Section 5 discusses related work. Finally, Section 6 presents some final remarks.

2. OIL OVERVIEW

OiL stands for *ORB in Lua* and is an implementation of CORBA (Common Object Request Broker Architecture) that shares many of the characteristics of the Lua language, such as simplicity, flexibility and portability. Unlike other CORBA implementations, OiL does not provide ORB support by means of generated glue code such as stubs and skeletons. Instead, similarly to the approach used by LuaOrb, all support is created at runtime, including method invocation and dispatching. Roughly, we can divide the implementation of the OiL ORB into four main modules, responsible for CORBA protocol support, interface definition description, method invocation, and method dispatch.

Basically, all communication performed by the ORB is done through messages written into sockets according to the GIOP (General Inter-ORB Protocol) and containing Lua values encoded into the CDR (Common Data Representation) defined by CORBA [13]. This communication is almost completely implemented in Lua (only socket and bit manipulation support is provided by C libraries) and therefore can be easily adjusted at runtime to certain criteria.

However, 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 data structures that describe IDL definitions and can be modified or replaced dynamically.

Conveniently, OiL provides a compiler that translates IDL specifications into Lua data structures and registers them at the OiL internal interface repository by means of the operation `oil.loadidl`, as illustrated in the following examples.

All method invocation is done by object proxies, which work like dynamic stubs that perform method invocations according to a given interface definition. Each object proxy is an instance of a class associated with a particular interface. Every time an interface definition is changed, its associated proxy class is changed as well. As a consequence, every object proxy of that class is adapted to reflect the new interface. The code on Listing 1 shows a script that initially accesses a remote object using a particular interface definition. Later, when the application provides a new definition for the same interface, the ORB is adapted to match the new definition.

Listing 1: Client code adaptation.

```

1 oil.loadidl[[
2   interface Hello {
3     void hello();
4   };
5 ]]
6 proxy = oil.newproxy(readIOR("ior.txt"), "Hello")
7 proxy:hello()
8
9 ...

```

```

10 oil.loadidl[[
11   interface Hello {
12     string hello(in string name);
13   };
14 ]]
15 print(proxy:hello("World"))

```

Similarly, method dispatch is done according to an interface definition provided at object creation. Whenever a new request is addressed to a particular object, the associated interface definition is used to retrieve the operation signature, which is then used to figure out the actual kind of data marshaled in the message. This way, whenever an interface definition is changed, every subsequent request received by an object with that interface will be matched against the new definition. The code on Listing 2 initially creates an object with a given interface. However, after this interface is changed, every new request is matched to the new definition. Therefore, the object implementation can be changed to properly fit into the new interface.

Listing 2: Server code adaptation.

```

1 impl = { } — object is an empty table
2 function impl:hello()
3   print("Hello World!")
4 end
5 oil.loadidl[[
6   interface Hello { void hello(); };
7 ]]
8 object = oil.newobject(impl, "Hello");
9 writelOR(object, "ior.txt")
10
11 ...
12
13 oil.loadidl[[
14   interface Hello {
15     string hello(in string name);
16   };
17 ]]
18 function impl:hello(name)
19   return "Hello "..name.."!"
20 end

```

However, besides being able to adapt the middleware to certain criteria, such as a new interface definition, it is crucial to introduce these changes at adequate moments to avoid inconsistencies. This becomes especially cumbersome in concurrent systems, where adaptation may interfere with many different tasks that are performed at the same time.

2.1 Cooperative Concurrency

Lua provides native support for concurrency by means of co-routines that 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, thread synchronization is much simpler in cooperative models. However, fairness conditions are almost entirely handled by the programmer.

In order to enhance portability and simplicity, the OiL concurrency model is based on Lua co-routines. Therefore, every concurrent task performed by OiL, including method dispatching, is done by a co-routine that explicitly switches execution at well-known points, thus avoiding potential race conditions. Cooperative concurrency has many advantages, such as 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 undesirable execution switches that lead to inefficient situations (which normally occur in preemptive models when automatic scheduling policies are used).

Particularly, cooperative concurrency copes well with dynamic adaptation. This is due mainly to the virtual 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 class implementation and its respective interface with no need for synchronization mechanisms.

On the other hand, cooperative models of concurrency may not cope very well with some requirements. For example, execution switching between independently developed objects (or components) may be difficult to be achieved fairly. It is possible to introduce switching policies in cooperative models to ensure fairness constrains, but this kind of analysis is not included in our current research. Even though we are aware of current limitations of cooperative models, we believe that its use simplifies enormously the complexity of dynamically adaptable systems and therefore is an interesting alternative for experimentation on this subject. Anyway, preemptive concurrency models are equivalent to their cooperative counterparts. Therefore we are very confident that the same results valid for experimentations on our simplified cooperative platforms can be adapted to other platforms that use preemptive models with proper synchronization mechanisms.

2.2 Object Models

Although Lua is not an object-oriented language it can be extended to provide object-oriented features such as behavior sharing through prototype-based or class-based approaches. This lack of built-in support for OOP (object-oriented programming) enables the use of custom-made OOP models that address specific requirements, like dynamic adaptation features for example. Additionally, Lua provides elementary support for a sort of OOP by means of some syntactic sugar and conventions. This elementary support can be used as an integration point between different custom-made models.

Almost all the adaptation support provided by OiL is implemented over a set of dynamically adaptable object-class models called LOOP (Lua Object-Oriented Programming) that is implemented using the extension mechanisms provided by Lua. LOOP models define object classes like a kind of virtual table of operations (*i.e.* VTable) that contains all operations available for objects of that class. This table of operations is shared by all objects of a particular class. On the other hand, object state data is stored at each object-instance structure, which is implemented by a Lua table (*i.e.* an associative array).

This kind of separation between state and operation data allows us to create adaptation mechanisms that change the implementation of systems without any object (or component) replacement or state transfer. For example, let us consider the code on Listing 3 that shows part of the implementation of the Dining Philosophers problem.

Listing 3: Dining Philosophers on LOOP.

```

1 Fork = oo.class{ inuse = false }
2
3 function Fork:get()
4     local ok = not self.inuse
5     if ok then self.inuse = true end
6     return ok
7 end
8
9 function Fork:release()
10    assert(self.inuse,
11           "attempt to release an unused fork")
12    self.inuse = false
13 end
14
15 Philosopher = oo.class{ -- default attributes
16    name = "unnamed",
17    hunger = 0,
18    has_left = false,
19    has_right = false,
20 }
21
22 function Philosopher:update()
23     if
24         self.has_left and
25         self.has_right
26     then
27         if self:is_hungry()
28             then self:eat_some()
29             else self:release_forks()
30         end
31     else
32         self:get_more_hungry()
33         if self:is_hungry() then
34             if not self.has_left then
35                 if self:try_get_fork("left") then
36                     return
37                 end
38             end
39             if not self.has_right then
40                 if self:try_get_fork("right") then
41                     return
42                 end
43             end
44         end
45     end
46 end
47
48 ...

```

Suppose now that, after the deployment of an application made up of three philosophers and three fork instances, the system reaches a deadlock because at a given time each philosopher has a fork. We can update the system's implementation in order to avoid deadlocks if the application provides an entry point for the execution of adaptation code, for example by using an object that implements an operation that compiles and executes a chunk of Lua code received as a parameter.

As an example, we have applied the change illustrated at the code on Listing 4 to add proper deadlock prevention.

Listing 4: Adaptation to avoid deadlocks.

```

1 adaptor = oil.newproxy(readIOR("adaptor.iior"))
2
3 adaptor:execute[[
4     -- adds new operation to class Philosopher
5     function Philosopher:avoid_deadlock()
6         if (
7             (self.has_left and not self.has_right)
8             or
9             (not self.has_left and self.has_right)
10            ) and (math.random(3) == 1)
11        then
12            self:release_forks()
13        end
14    end
15

```

```

16 — change existing operation update
17 function Philosopher: update()
18     if self.has_left and self.has_right then
19         if self.is_hungry()
20             then self.eat_some()
21             else self.release_forks()
22             end
23         else
24             self.get_more_hungry()
25             if self.is_hungry() then
26                 if not self.has_left then
27                     if self.try_get_fork("left") then
28                         return
29                     end
30                 end
31                 if not self.has_right then
32                     if self.try_get_fork("right") then
33                         return
34                     end
35                 end
36                 self.avoid_deadlock()
37             end
38         end
39     end
40 ]]

```

Actually, the whole implementation of OiL is based on LOOP classes. However, the implementation of dynamic proxies may be the best example to illustrate the use of LOOP's dynamic adaptation features in OiL. A new class of proxy objects is created for each interface handled by OiL and this class is then used to create proxies of objects of that particular interface. These proxies are objects that contain state data that describe each object's interface and reference such as host, port or object id. However, the whole method invocation is implemented by a set of operations stored at the proxy class that are created according to each object's interface. Whenever the interface is changed (*e.g.* a new operation is added), the list of operations available for these proxies is changed accordingly. In fact, proxy classes are implemented as a cache of stub operations created on-demand that actually implement method invocation. This way, when an interface is changed, this cache is emptied to allow the recreation of stub functions for new operations.

3. ADAPTATION ABSTRACTIONS

In order to safely introduce changes into a running program, aside from being able to change the program's implementation and middleware support, one needs to organize these changes properly. This should be done by means of proper programming abstractions specifically devised for dynamic adaptation. LOOP classes already provide an adaptation abstraction through the notion of dynamic classes, which can be used to change the implementation of a whole set of related objects, as seen previously. However, in some situations other aspects must be addressed as well, for instance in the case of an adaptation that requires changes in system data besides its implementation. As an example, let us suppose the implementation of a component that collects the e-mails of authors when a conference paper is submitted and notifies previously registered authors, as illustrated by the code on Listing 5.

Listing 5: Mail collector naïve implementation

```

1 oil.loadidl [[
2     struct Author {
3         string name;
4         string email;
5     };
6     typedef sequence<Author> AuthorSeq;

```

```

7     struct Paper {
8         string title;
9         AuthorSeq authors;
10    };
11    interface Collector {
12        void submit(in Paper paper);
13    }; ]]
14 Collector = oo.class{}
15 ...
16 ...
17 ...
18 function Collector:submit(paper)
19     for _, email in ipairs(self.emails) do
20         send_to(email, "New paper: "..paper.title)
21     end
22     for _, author in ipairs(paper.authors) do
23         table.insert(self.emails, author.email)
24     end
25 end

```

After some time running the conference website we would notice that some authors receive duplicated notifications due to the submission of papers with common authors. Now we want to correct this problem and use this information to obtain the name of the author with more papers submitted to the conference as a result of the operation `submit`. To do so, we must change the component's implementation and interface, as well as adapt its state to remove duplicated e-mails. Supposing that we have many components running for different conferences in the same website, this adaptation may be more complicated.

Although LOOP classes can add new attributes with default values to object instances, it is not possible to change object attributes through the class. However, if the new implementation is defined as a new class (which can be a subclass of the original one), then objects can be adapted on-demand by means of a triggering operation call that switches the object class to the new one. This way, when the class of an object is changed, its data can be modified accordingly.

On the other hand, the introduction of these changes can be very error-prone. Therefore, we have defined an abstraction that is used to properly introduce a combined change in a component's interface, implementation and data. This abstraction consists of a description that contains the new implementation of component operations, its new interface definition, a procedure to adapt the instance state and a list of the operations that should trigger the adaptation. Suppose that for each object class we have an object that implements the interface illustrated on Listing 6, which receives a complete definition of an adaptation that defines the change in state, implementation and interface of that particular object class.

Listing 6: IDL for combined adaptor component

```

1 interface ComponentAdaptor {
2     void apply_change(
3         in StringSequence triggers,
4         in string state_adaptation_code,
5         in string impl_adaptation_code,
6         in string new_interface_def
7     ) raises (CompileError);
8 };

```

Whenever a new change is applied, the current class is modified so that each one of the specified triggering operations is replaced by an operation that executes the state adaptation code over the object instance and then changes the class of the object to a new one that is a derived class of the original one and contains the changes defined by the implementation adaptation code.

Such approach enables us to adapt object implementations on-demand whenever a particular set of triggering operations is invoked. For the example presented above, we can define an adaptation like the one showed on Listing 7, which specifies the changes on the state of each object (lines 8-13) and eventually removes duplicated e-mails. This adaptation also specifies changes on the implementation of operation `submit` so that it avoids duplications and returns the name of the author with more submitted papers (lines 18-33). Consequently, this adaptation also defines the interface that matches the new implementation as shown (lines 38-40).

Listing 7: Adaptation for the Collector component

```

1 Adaptor = oil.newproxy(readIOR("adaptor.ior"))
2 Adaptor:apply_change(
3   -- triggering operations
4   {"submit"},
5
6   -- object state adaptation
7   [[
8     local emailset = {}
9     for _, email in ipairs(self.emails) do
10      local count = emailset[email] or 0
11      emailset[email] = count + 1
12    end
13    self.emails = emailset
14  ]],
15
16  -- class implementation adaptation
17  [[
18    function Collector:submit(paper)
19      for email in pairs(self.emails) do
20        send_to(email, "New paper: "..paper.title)
21      end
22      local result
23      local maxcount = 0
24      for _, author in ipairs(paper.authors) do
25        local count = self.emails[author.email] or 0
26        if count >= maxcount then
27          maxcount = count
28          result = author.name
29        end
30        self.emails[author.email] = count + 1
31      end
32      return result
33    end
34  ]],
35
36  -- new object interface definition
37  [[
38    interface Collector {
39      string submit(in Paper paper);
40    };
41  ]])
42 )

```

Many other adaptation abstractions have been proposed to address different situations [11]. Examples include the use of interceptors to add functionality rather than replacing it, or the description of architecture changes to adapt component connections rather than internal state. We intend to use the infrastructure provided by OiL to experiment with such abstractions in order to evaluate its applicability and feasibility [9].

4. FIRST RESULTS

Although OiL development is still in its early stages, we have already produced some initial results by using it in some applications. Currently, most applications using OiL do it as an alternative to other larger CORBA implementations. That is the case of the InteGrade system for grid

	Orbacus	OiL
No parameters	57.5906	70.5835
String inout parameter	58.3726	71.3613
User exception	83.4109	78.1850

Table 1: Time to perform 100,000 calls (in seconds).

management [5]. InteGrade provides lightweight node management by using OiL to implement CORBA support in the software running on the nodes of the grid.

Similarly, OiL is being used by the Tecgraf computer graphics laboratory at PUC-Rio in an application developed for the Brazilian Oil Company that controls the execution of applications on heterogeneous clusters. Lua is extremely portable, because it is entirely implemented in standard C. As a result, OiL turns out to be an equally portable implementation of CORBA. Additional socket support is provided by fairly portable code that lies mostly in BSD sockets. Due to the large number of different platforms composing the clusters, the portability provided by OiL has simplified the deployment of the code running in clusters.

Aside from its usage on real projects, OiL has also been used in some experimental projects. One example that illustrates well the use of OiL to analyze the applicability of middleware implementation techniques is a work [16] that evaluates three different techniques to traverse firewalls. In another experiment we were able to run OiL over resource-limited devices, such as mobile phones, using a port of the Lua language for the BREW platform.

Initial performance evaluations also show promising results even with our current implementation with no optimizations. For measuring performance we used a CORBA object, with an implementation in Lua and another in C++, which only receives requests but does no processing. We compared OiL's performance with Orbacus for C++ compiled with optimizations and using statically linked libraries. The tests were performed by a client implemented in C++ with Orbacus that measured the time taken to perform 100K operation calls over a server. Table 1 shows the average time taken to perform the 100K calls of three different operations: one with no parameters; another with an inout parameter of type string; and a last one that raises a user exception.

These results show that the current implementation of OiL adds a small overhead of only 20% over the performance of a static C++ implementation. Most of this overhead is due to the marshalling process that is done at every call and is mostly implemented in Lua interpreted code. On the other hand, this performance proximity is mostly justified by the network delay inherent to distributed applications. However, the performance of OiL with more complex types such as structures or sequences should be worse than these initial results.

5. RELATED WORK

DynamicTAO and Open ORB are among the first implementations of reflective middleware [7]. Both of them provide support for introspection and the dynamic replacement of both ORB and application components. Since dynamicTAO is implemented in C++, it has to provide all support for introspection and dynamic reconfiguration from scratch. Open ORB, on the other hand, was prototyped in Python, a dynamic language that naturally includes this kind of sup-

port. Unlike Open ORB's Python prototype, OiL features a lightweight implementation with a performance that is close to the ones of C++ ORBs. A later prototype of Open ORB, written in C++ and based on the COM model, traded flexibility for performance, achieving high throughputs.

Another work similar to ours is OpenCORBA, a reflective ORB implemented in a Smalltalk-like reflective language based on the concept of meta-classes [8]. Similarly to our approach, in OpenCORBA the behavior of CORBA services is modified by replacing the meta-class object of the class defining that service.

6. FINAL REMARKS

In this paper we presented OiL, an ORB implemented in the scripting language Lua, and discussed how its features can be used to implement dynamic adaptation mechanisms. Despite its dynamic and interpreted nature, the current implementation of OiL presents a small performance penalty compared to CORBA implementations in C++. OiL can be downloaded from <http://oil.luaforge.net/>.

We intend to use OiL as a platform for experimentation on dynamic adaptation because our claim is that the use of a scripting language simplifies the development of such mechanisms due to its dynamic nature. However, OiL's adaptation features are very low-level and lack mechanisms for adapting the application properly. Although LOOP models already provide some support for changing application implementation, we are particularly interested in investigating more elaborated abstractions that impose different programming paradigms to facilitate the implementation of dynamic adaptations.

Traditionally, scripting languages are underestimated due to their 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 becomes less evident. Additionally, scripting languages are generally dynamic and usually provide introspection mechanisms to check runtime conditions, which is particularly useful for distributed applications where errors at runtime are common due to deployment failures.

Finally, we intend to run experiments using our platform that implement and evaluate techniques and abstractions for dynamic adaptations currently proposed by the community. In addition to evaluating the performance impact introduced by the different dynamic adaptation approaches, we are particularly interested in measuring the degree of flexibility, usability and feasibility of these approaches. However, the measurement of these subjective aspects is a long-term research goal of our group, since there is no generally accepted set of metrics to evaluate these aspects.

7. REFERENCES

- [1] T. Batista, R. Cerqueira, and N. Rodriguez. Enabling reflection and reconfiguration in CORBA. In *Proceedings of ARM'03*, pages 125–129, Rio de Janeiro, Brazil, June 2003.
- [2] R. Cerqueira, C. Cassino, and R. Ierusalimsky. Dynamic component gluing across different componentware systems. In *Proceedings of DOA'99*, pages 362–373, Edinburg, Scotland, September 1999. IEEE Press.
- [3] B. Feijó, P. Rodacki, J. Bento, S. Scheer, and R. Cerqueira. Reactive design agents in solid modelling. In J. Gero and F. Sudweeks, editors, *Artificial Intelligence in Design'98*, pages 557–577. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998.
- [4] A. Ferreira, R. Cerqueira, W. Celes, and M. Gattass. Multiple display viewing architecture for virtual environments over heterogeneous networks. In *Proceedings of SIBGRAP'99*, pages 83–92, Campinas, Brazil, 1999. SBC, IEEE Computer Society.
- [5] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. Integrate: Object-oriented grid middleware leveraging idle computing power of desktop machines. *Concurrency and Computation: Practice and Experience*, 16:449–459, March 2004.
- [6] R. Ierusalimsky. *Programming in Lua*. Lua.org, 2003.
- [7] F. Kon, F. Costa, R. Campbell, and G. Blair. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, June 2002.
- [8] T. Ledoux. Opencorba: A reflective open broker. In *Proceedings of Reflection'99*, number 1616 in Lecture Notes in Computer Science, pages 197–214, St. Malo, France, July 1999. Springer-Verlag Heidelberg.
- [9] R. Maia, R. Cerqueira, and N. Rodriguez. An infrastructure for development of dynamically adaptable distributed components. In R. Meersman and Z. Tari, editors, *Proceedings of DOA'04*, volume 3292 of *Lecture Notes in Computer Science*, pages 1285–1302, Agya Napa, Cyprus, October 2004. OTM 2004, Springer-Verlag Heidelberg.
- [10] M. Martins, N. Rodriguez, and R. Ierusalimsky. Dynamic extension of CORBA servers. In *Proceedings of Euro-Par'99*, Lecture Notes in Computer Science, pages 1369–1376, Toulouse, France, September 1999. Springer-Verlag.
- [11] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng. A taxonomy of compositional adaptation. Technical Report MSU-CSE-04-17, Software Engineering and Network Systems Laboratory, Michigan State University, East Lansing, Michigan, July 2004.
- [12] A. L. Moura, C. Ururahy, R. Cerqueira, and N. Rodriguez. Dynamic support for distributed auto-adaptive applications. In R. Wagner, editor, *Proceedings of ICDCS'02*, pages 451–458, Viena, Austria, July 2002. IEEE Press.
- [13] Object Management Group, Needham, EUA. *Common Object Request Broker Architecture: Core Specification - Version 3.0*, December 2002. document: formal/2002-12-06.
- [14] F. Peschanski, J.-P. Briot, and A. Yonezawa. Fine-grained dynamic adaptation of distributed components. In M. Endler and D. Schmidt, editors, *Proceedings of Middleware 2003*, volume 2672 of *Lecture Notes in Computer Science*, pages 123–142, Rio de Janeiro, Brazil, June 2003. Springer-Verlag.
- [15] M. Román, C. K. Hess, R. Cerqueira, A. Ranganat, R. H. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.
- [16] A. Theophilo, M. Endler, and R. Cerqueira. Evaluation of three approaches for CORBA firewall/NAT traversal. In *Proceedings of DOA'05*. Springer-Verlag Heidelberg, 2005. (to appear).