Automated Testing for a CORBA-Based Distributed System

Renato Cerqueira
Roberto Ierusalimschy

Computer Science Department — PUC-Rio, Brazil
rcerq,roberto@inf.puc-rio.br

Abstract

In this paper, we present a test infrastructure used to automate the testing of a CORBA-based distributed system. This infrastructure is based on a flexible programming environment for CORBA systems, called LuaOrb. LuaOrb uses the interpreted language Lua to access and implement CORBA objects, and adopts a dynamic composition approach that, combined to the interactive facilities offered by an interpreted language, makes it a quite suitable platform for test automation.

Keywords: Testing, Scripting Languages, CORBA.

1 Introduction

Testing is often a complex task, more so if the tested application is distributed. For instance, the execution of a simple test for a distributed system may require to launch many servers and clients in different hosts following a specific order. The high complexity for a test execution may be an obstacle to adopt software development methodologies that apply testing more frequently during the development cycles, such as the Extreme Programming approach [Bec99a, Bec99b].

This paper describes an infrastructure used to automate the testing of a CORBA-based distributed system. This infrastructure was developed during a project with the Brazilian Navy, when our group developed a distributed sonar simulator that should work integrated with another distributed simulation system developed for training warship commanders. In this development, we used many practices that are common to flexible processes like Extreme Programming: planning game, simple design, testing centered, continuous refactoring, and continuous integration [Bec99a, Bec99b].

The test infrastructure used in this project is based on a scripting tool for CORBA objects, called LuaOrb [ICR98, CCI99]. LuaOrb implements a binding between the interpreted language Lua [IdFC96, FIC96, IFC99] and CORBA [OMG98], and uses the dynamic and reflexive facilities of both systems to support the use of new IDL interfaces and the implementation of CORBA objects at runtime, without the need of precompiled stubs.

Interpreted languages like Tcl [Ous94] can take all the effort out of test implementations. Unit tests can be supported by a set of small scripts that validates different aspects of the system components. With high level abstraction mechanisms and a simple syntax, an interpreted language can be also a powerful tool for an end user—the system customer—to implement functional tests. Features like interactivity, automatic memory management, and no need for recompilations and relinking after each change are very useful for building little test scripts (specially for end user programming) and allow that several design options are implemented and tested in a short period of time.

Lua fits all those characteristics: It has a very simple syntax and a high level mechanism for data description. Besides, with LuaOrb, we can use Lua to implement scripts to test either each CORBA object independently or the whole integrated system. By keeping all scripts during the development process, we can easily support automated regression testing [KP99] of our system. Besides the scripting tool, the test infrastructure offers some CORBA services to support remote installation of CORBA servers and remote execution of Lua scripts.

Although scripting languages like Lua provide many facilities for testing, the CORBA infrastructure makes testing easier: With the CORBA's reflexive facilities, such as the Dynamic Invocation Interface (DII) and the Interface Repository, test tools can have access to the application components without the previous creation of specific test stubs. Without this kind of reflexive facilities, the test stubs must be generated either with the help of some stub generator, such as SWIG [Bea98] and toLua [toL], or manually by the developer.

The next section describes the sonar and the Navy's tactical simulators. The LuaOrb system is presented in more details in Section 3, while its extensions to support the test infrastructure are described in Section 4. The last section is reserved for some final remarks.

2 The Sonar Simulator

The Distributed Sonar Simulator (DSS), wherein we have applied the distributed test infrastructure, is part of a bigger project to develop a distributed tactical simulator for training warship commanders of the Brazilian Navy.

The whole tactical simulator, called SSTT (*Sistema de Simulação Tática e Treinamento*), is responsible to manage and integrate all entities in the simulated environment, such as the vehicles (ships, aircrafts, submarines, etc.), their sensors (radars, sonars, GPSs, gyros, links, etc.), and user consoles. Besides these components, which represent real entities, there are also some control components, such as a collision detection module.

The users interact with the system through graphical consoles where they have navigation instruments and the sensor displays of a vehicle. The navigation instruments allow changes in the vehicle's course and velocity. A sensor display shows the signal generated by a sensor, such as a radar or sonar.

The objects that represent vehicles have all information about their relevant mechanical characteristics: position, velocity, course, turn rate, etc. With these characteristics, the vehicle acts as a deterministic entity, whose position is a function of the time.

This system adopts the *dead reckoning* approach [IEE93, MZP$^+$94], that is, a sensor object extrapolates the vehicles' positions in function of the time. When a specific vehicle has its kinetic state changed, all sensors must be updated. To perform this update, the observer design pattern [GHJV95] is used: A vehicle has a collection with the objects interested in listening the changes in its state and, when any change occurs, all these objects (listeners) are notified. Figure 1 illustrates the pattern behavior. We use this pattern in many other situations that require a notification mechanism.

Since the main goal of the sonar simulator DSS is to provide only the information more relevant to the tactical procedures, many simplifications could be assumed in its specification. For instance, the detection algorithm and the graphic and sound representations of the sonar contacts could be very simplified. Basically, the sonar simulator monitors the simulated vehicles, applies a simplified detection algorithm, and presents in a graphic console the sonar contacts that correspond to the detected targets. The sonar console operator can also create synthetic contacts, classify them, and select some to inform the tactical graphic console.

Due to constrains in the project schedule, both systems, DSS and SSTT, were developed in parallel. Therefore, the two development teams worked together to define the IDL interfaces of all components that are common for both systems:

- *Target interface*: This interface provides methods to get and set the target attributes, besides offering operations to register listener objects that are interested in changes in the target state. A target

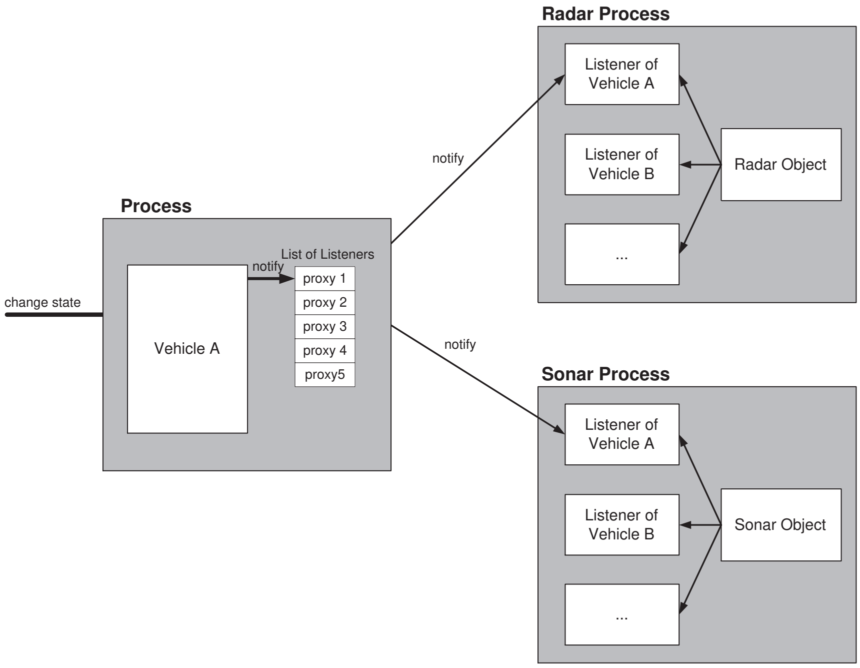

Figure 1: The basic observer pattern.

can be a vehicle, a petroleum offshore platform, a shoal, a whale, etc. The sonar simulator, like other simulated entities, uses this interface to monitor the state of all targets.

- *Vehicle interface*: This interface is a specialization of the target interface that provides additional operations specific for vehicle entities. The sonar simulator uses this interface to interact with its own vehicle.
- *Target collection interface*: This interface represents a distributed collection, with operations to insert and remove elements, and iterators for traversing elements. All targets in the simulated environment are registered in a main target collection.
- *Tactical console interface*: This interface provides operations to update console information. The console sonar uses this interface to inform the contacts that are selected by its operator.

During the development process, many integration tests were performed and the common IDL interfaces were refined. Figure 2 shows a simplified version of the `Target` and `TargetCollection` interfaces.

The sonar simulator was designed as a set of distributed CORBA objects, where each object is responsible for one main task. To illustrate how the sonar simulator works, Figure 3 presents a simplified version of the actual sonar interfaces (note that the module `Sonar` refers to some definitions presented in Figure 2). The `Detector` component interacts with the target collection to access the available targets, and applies the detection algorithms periodically. After each detection period, this component sends a list of the detected targets' descriptions to the `Display` component, which is responsible for the graphic and sound presentation.

3 The LuaOrb System

LuaOrb is a programming tool for CORBA systems that uses the interpreted language Lua [IdFC96, FIC96, IFC99] as its programming language. LuaOrb is based on a language binding that, as usual,

```
struct TargetDescription {
   unsigned long id;
   PlatformDescription platform;
   float x, y, z, vx, vy, vz;
};

interface StateListener {
   void notify(in TargetDescription target);
};

interface Target {
   TargetDescription getDescription();
   void destroy();
   void addStateListener(in StateListener listener);
};

interface Iterator {
   Target first();
   Target next();
   void destroy();
};

interface TargetCollection {
   void insert(in Target obj);
   void remove(in Target obj);
   Iterator createIterator();
};
```

Figure 2: A simplified version of `Target` and `TargetCollection` interfaces.

defines mappings between Lua and IDL types. However, unlike other bindings, LuaOrb relies on the reflexive facilities of CORBA and the dynamic nature of Lua.

Currently, most CORBA language bindings direct their design to support clients and servers written in a statically compiled language, such as C++, using statically compiled stubs and skeletons. Although CORBA supports dynamic interfaces, such as the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI), these interfaces are 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, such as rapid prototyping, test stub implementation, scripting for component configuration and testing, and on-the-fly and remote server updates.

A scripting language brings an interesting design alternative to this static nature of CORBA systems: It greatly improves the support for testing, rapid prototyping, and dynamic configuration, as we can load and test new design alternatives for a system in a quick and simple way. For instance, a CORBA object can be tested by a set of scripts, while other services that are not implemented yet, but are required by this object, can be provided by prototypes implemented with the scripting language. Servers implemented with a scripting 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.

The main goal of LuaOrb is to offer a more suitable support for developing flexible applications, with the dynamic incorporation of CORBA objects. Using the CORBA's features for introspection (Interface Repository) and dynamic construction of method calls (DII), LuaOrb offers a binding where scripts can incorporate and make effective use of new object interfaces at run time. Moreover, as in other bindings, Lua can use external CORBA servers transparently, in the same way it uses internal objects. Using the

```
module Sonar {
   typedef sequence<TargetDescription> TargetDescrList;

   interface Display {
      void detect( in TargetDescrList targets);
   };

   interface Detector {
      void activate();
      void deactivate();
      void registerDisplay(in Display display);
      void registerTargetCollection(in TargetCollection collection);
   };
};
```

Figure 3: The IDL definition for the sonar interfaces.

CORBA's DSI, LuaOrb also supports dynamic implementation of new CORBA object servers.

The OMG recognizes the relevance of scripting languages to manipulate CORBA objects [OMG96, C^+97]. At first sight, it seems that this kind of binding can be one-way: the scripting language acts as a client of CORBA objects to coordinate and test them, but it does not need to implement new *object servers*. However, besides the motivation of rapid prototyping, the binding must work in both directions to support callback objects—such as *listeners* in JavaBeans [Jav96]—since those callback objects must be called from the external components. Because the use of callback objects is a pervasive technique in object-oriented programming (for instance, most patterns presented in [GHJV95] use callback objects), a language without such support would have quite a limited use.

LuaOrb uses *proxies* for each external CORBA object to be handled by a program, following the design pattern *proxy* [GHJV95]. A proxy is an internal object created to represent an external one; its basic functionality is to delegate to the external object it represents any operation performed over itself. To implement this interface, LuaOrb uses the reflexive mechanisms of both CORBA and Lua.

Lua has a mechanism such that operations applied to the proxy, made through the regular language syntax, can be detected and handled by the binding that created the proxy. Moreover, when the binding is evoked, it can access any information concerning the original operation (the "receiver" proxy, the operation name, eventual arguments, etc). As an example, let us use the following IDL interface:

```
interface Calc {
  float add(in float a, in float b);
};
```

Once you have a proxy to a CORBA object with that interface, the method `add` can be called exactly like any method invocation over native objects. For instance, we can test a `Calc` object in the following way:

```
assert(calc:add(2,3) == 5)
assert(calc:add(0,3) == 3)
assert(calc:add(2,0) == 2)
assert(calc:add(-2,3) == 1)
...
```

Because LuaOrb redefines the behavior of all proxy tables, this method call generates an event, which calls the corresponding handler (called *tag method* in Lua). This event handler, which is part of the binding, gets the external component that the proxy represents; then it uses introspection to check whether this component has the desired method. The final result is a closure encapsulating the information about the found method. When the closure is called with the given arguments, it executes a sequence of steps. First, for each formal parameter, it checks whether the corresponding Lua argument has a compatible type and converts this value to the corresponding CORBA value. Second, it dynamically builds the method call and performs the call. Finally, it converts back any results (or `out` parameters), again using the method description and the compatibility rules. (For more details about the binding, see [ICR98].)

To provide dynamic object implementations with Lua, LuaOrb uses *generic adaptors*. A generic adaptor is registered in the CORBA Object Adaptor as an object servant that provides a specific interface, and delegates the real method implementations to an object defined in Lua, in a way similar to proxies: Any operation performed over the adaptor is applied over the real implementation. Because these implementations are written in Lua, they can be dynamically created and linked to adaptors. The linking between the adaptor and the real object uses the same reflexive mechanisms used by our client proxy, but this time these mechanisms are provided by the language interface (the Lua API also allows the dynamic construction of method calls).

As an example of how to implement CORBA servants in Lua, consider the following Lua object that could be used as an implementation of the `Calc` IDL interface:

```
obj = { add = function(self, a, b)
            return a+b
        end
      }
```

With LuaOrb, a `Calc` servant can be created as shown below:

```
servant = CreateDSIServer("Calc", obj)
```

In this example, the `servant` object is an instance of a generic adaptor that can be exported to some client application. The `CreateDSIServer` function receives the IDL interface name and the Lua object that should provide the real service implementation.

The key point in this language binding is its dynamic nature. Every step, from type checking to method identification and invocation, is done at run time. Therefore, a program can incorporate new types and classes without any additional declarations and without stopping for recompilations. The binding also allows some coercions between data types. Therefore, many changes in an IDL interface do not affect its uses in Lua. Examples of such modifications are reordering and removing of struct fields, and changes between IDL types that have similar representations in Lua, such as `short` and `long`, or `array` and `sequence`. This coercion works recursively, so a list of Lua tables can be automatically converted to an array of records, for instance.

To illustrate how we can use LuaOrb to implement simple unit tests, let us consider the IDL definitions in Figure 4. We can implement simple scripts to test different aspects of the `Clipping` interface. For instance, a script can create a Lua table that is compatible with a `Region` sequence and use it to test the `setRegion` and `getRegion` operations:

```
region = {{{x = 0.0, y = 0.0}, {x = 1.0, y = 1.0}},
          {{x = -2.0, y = -2.0}, {x = -1.0, y = -1.0}}
         }
clip_obj:setRegion(region)
assert(isEqual(region, clip_obj:getRegion()))
```

And then it can use a set of points to test the operation `isInside`:

```
struct Point {
   double x, y;
};
typedef Point Rect[2];
typedef sequence<Rect> Region;

interface Clipping {
   void setRegion(in Region reg);
   Region getRegion();
   boolean isInside(in Point pt);
};
```

Figure 4: The IDL definition for the Clipping service.

```
assert(clip_obj:isInside({x = 0.5, y = 0.5})
assert(clip_obj:isInside({x = -1.5, y = -1.0})
assert(not clip_obj:isInside({x = 5, y = -0.5})
assert(not clip_obj:isInside({x = -0.5, y = -1.0})
```

(A real test script could declare lists of regions and points to be tested, and use a loop to perform all tests.) Readers that are familiar with the binding between CORBA and C++ should consider how many lines are needed to write a client program to do this test in CORBA/C++.

Because of its dynamic nature, LuaOrb can present a run-time performance penalty, when compared with statically compiled stubs and skeletons. This performance penalty has been greatly reduced with the minimization of the number of queries to the Interface Repository. This has been achieved simply by caching the most recent queries. The current performance penalty, when compared to C++, is about 50%, which is quite acceptable for testing.

4 The Testing Infrastructure

The previous section showed some examples that illustrate how to use LuaOrb to implement unit test scripts. Another facility that LuaOrb provides for test support is the possibility of implementing, in a simple way, prototypes of components that are required to test another component. For the sonar simulator, this feature was very important, since this system required components that would be developed by another group. In the many integration tests that were performed, those prototypes were replaced with the final or intermediate component implementations. Although the interfaces of the shared components evolved during the development process, the prototype components could easily evolve together. Even after the final component implementation had been available, the prototype continued to be used, because the final component was typically much more complex than what concerned the sonar simulator. Therefore, we could maintain the execution environment of the sonar simulator as simple as possible during its development. Note that, besides the use of an interpreted language, much of this flexibility is a consequence of the strict separation between interfaces and their implementations that CORBA provides.

To make the test automation easier, we extended the test infrastructure with services that support remote installation of CORBA servers and remote execution of Lua scripts. Figure 5 shows the IDL interfaces for these services:

- The interface `Lua::DSIObject` allows the remote installation of CORBA servers. This interface encapsulates an object reference (the attribute `obj`) to a LuaOrb's dynamic server implementation, and offers the operation `installOperation` to install operation implementations in the dynamic server. This method receives the operation name and its Lua source code.

```
module Lua {
   interface DSIObject {
      readonly attribute Object obj;
      void installOperation(in string op_name, in string lua_code);
   };

   interface Factory {
      DSIObject createInstance(in string interface_name);
   };

   interface Interpreter {
      void doString(in string lua_code);
   };
};
```

Figure 5: The IDL definition for the test utilities.

- The interface `Lua::Factory` is used to create new instances of `Lua::DSIObject`. Note that the method `createInstance` receives the name of the interface that the new dynamic server should use as its own interface (you can find more details about these two interfaces in [MRI96]).
- The interface `Lua::Interpreter` allows you to send Lua code across a network to be executed in another host process. Note that the Lua code that the remote process executes can change its global Lua environment, that is, it can define new functions and variables or change previous definitions.

These services are very useful for scripts that need to manage different hosts to execute their test cases. Moreover, the scripts can be implemented such that we can change the hosts where each component runs without changing the test scripts.

As an example of how we used these services to test the sonar simulator, we will use the sonar detector component with the script fragment in Figure 6, in accordance with the IDL definitions in Figures 2 and 3. The sonar display component's prototype is created in the same process that runs the test script (lines 1-4). This test stub of the display component receives the list of detected targets, and checks if this list is equal to the expected result (line 2). A `Lua::Factory` object is used to install the target collection prototype in `host1` (line 5). To simplify the remote installation procedure, we have defined the function `installService` that receives an implementation identifier and a factory object (`host1.factory`). In the same way, we have defined the function `runScript` that receives a script name and a `Lua::Interpreter` object, and sends the script to be executed in the specified remote interpreter. This function is used to create the target objects in `host2` (line 6) and to execute the sonar detector component in `host3` (line7). After we have created all components, we can start to connect them: First, we create a LuaOrb proxy of the detector server (line 9), and then we link it to the target collection and the display component prototypes (lines 10–11). Finally, we activate the detector component (line 12).

The file `createTargets.lua` and the list `expected_targets` (used in lines 6 and 2 respectively) define a test case. So, we can apply this script to different test cases, only changing the file used to create the targets and the list of targets that we expect as result of the detection algorithm.

This test script can use up to four process to run: one for the sonar detector component, which we are testing, and other three processes for the sonar display component, the target collection service, and the target objects. Note that we can test different configurations for `host1`, `host2`, and `host3` using the same test script.

```
display = { detect = function(self, targets)
                assert(isEqual(targets,expected_targets)
            end
          }
collection = installService("TargetCollectionService", host1.factory)
runScript("createTargets.lua", host2.interpreter)
runScript("detector.lua", host3.interpreter)
wait(WAIT_TIME)
detector = createproxy{interface="Sonar::Detector",ior=detector_ior}
detector:registerTargetCollection(collection)
detector:registerDisplay(display)
detector:activate()
```

Figure 6: A simple automated test script.

5 Final Remarks

The infrastructure used to test the sonar simulator system has presented good results: We have automated most of the unit tests, with support to regression testing too. Moreover, we easily implemented integration tests that required to manage many process in different hosts. We have achieved the same results in other CORBA-based projects, such as a distributed 3D visualization system [FCCG99].

However, we did not reach the same success when we tried to apply the infrastructure to test some aspects of the system's graphical user interface (GUI), such as the sonar's graphic and sound presentation. Even test tools like *screen scrapers*, which are based on GUI event recording and replaying, did not work, because the GUI layout changed too often. So, we performed this kind of functional testing manually. We are planning to improve the test infrastructure to provide better support for GUI testing in another project that we have just started.

In the sonar simulator and other CORBA-based projects [FCCG99, RIC98, GFCI98, RMIC97], LuaOrb has proved to be a powerful programming tool to support flexible development processes, like Extreme Programming: It provides many facilities for testing and rapid implementation. Often, a component prototype implemented with LuaOrb becomes the component's final version, if the performance penalty is not relevant. Otherwise, the component implementation always can be refactored to a more efficient implementation (typically a C/C++ version).

References

[Bea98] David Beazley. SWIG and automated C/C++ scripting extensions. *Dr. Dobb's Journal*, 23(2):30–36, 1998.

[Bec99a] Kent Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, 1999.

[Bec99b] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999.

[C+97] IBM Corporation et al. CORBA Component Imperatives. Technical Report orbos/97-05-25, OMG, 1997. `http://www.omg.org`.

[CCI99] Renato Cerqueira, Carlos Cassino, and Roberto Ierusalimschy. Dynamic component gluing across different componentware systems. In *International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, 1999. OMG, IEEE Press.

[FCCG99] Alexandre Ferreira, Renato Cerqueira, Waldemar Celes, and Marcelo Gattass. Multiple display viewing architecture for virtual environments over heterogeneous networks. In *XII Simpósio Brasileiro de Computação Gráfica*, Campinas, Brasil, 1999. SBC, IEEE Press.

[FIC96] Luiz H. Figueiredo, Roberto Ierusalimschy, and Waldemar Celes. Lua: An extensible embedded language. *Dr. Dobb's Journal*, 21(12):26–33, December 1996.

[GFCI98] Paulo Gomes, Bruno Feijó, Renato Cerqueira, and Roberto Ierusalimschy. Reactivity and proactiveness in virtual prototyping. In *2th International Symposium on Tools and Methods for Concurrent Engineering*, pages 242–253, Manchester, U.K., April 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[ICR98] Roberto Ierusalimschy, Renato Cerqueira, and Noemi Rodriguez. Using reflexivity to interface with CORBA. In *International Conference on Computer Languages 1998*, Chicago, 1998. IEEE.

[IdFC96] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[IEE93] Institute of Electrical and Electronics Engineers, International Standard. *Standard for Information Technology, Protocols for Distributed Interactive Simulation*, March 1993. ANSI/IEEE Std 1278.

[IFC99] R. Ierusalimschy, L. H. Figueiredo, and W. Celes. *Reference Manual of the Programming Language Lua version 3.2*. Rio de Janeiro, Brazil, 1999. (available by ftp at `ftp://ftp.tecgraf.puc-rio.br/pub/lua/refman.ps.gz`).

[Jav96] JavaSoft. *JavaBeans, Version 1.00-A*, December 1996. `http://java.sun.com/beans`.

[KP99] Brian Kernighan and Rob Pike. *The Practice of Programming*. Addison Wesley, 1999.

[MRI96] Marco Martins, Noemi Rodriguez, and Roberto Ierusalimschy. Dynamic extension of CORBA servers. In *Euro-Par'99 Parallel Processing*, pages 1369–1376, Toulouse, France, September 1996. Springer-Verlag. (LNCS 1685).

[MZP+94] Michael Macedonia, Michael Zyda, David Pratt, Paul Barham, and Seven Zeswitz. NPSNET: A network software architecture for large scale virtual environments. *Presence*, 3(4), 1994.

[OMG96] OMG. CORBA Scripting Language — Request For Proposal. Technical Report orbos/96-06-13, OMG, 1996. `http://www.omg.org`.

[OMG98] OMG, Framingham, MA. *The Common Object Request Broker Architecture and Specification; Revision 2.2*, February 1998. `http://www.omg.org`.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[RIC98] Noemi Rodriguez, Roberto Ierusalimschy, and Renato Cerqueira. Dynamic configuration with CORBA components. In *4th International Conference on Configurable Distributed Systems*, Annapolis, 1998.

[RMIC97] N. Rodriguez, A. Moura, R. Ierusalimschy, and R. Cerqueira. Aplicações de gerência com comportamento dinâmico. In *Anais do SFBSID'97*, pages 476–487, Fortaleza, Brasil, 1997.

[toL] Waldemar Celes. *toLua — accessing C/C++ code from Lua*. `http://www.tecgraf.puc-rio.br/~celes/tolua/tolua.html`.