PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 06/07

Combining Programming Paradigms in Asynchronous Systems

Ricardo Gomes Leal Costa
Pedro Martelletto de Alvarenga Bastos
Bruno Oliveira Silvestre
Noemi de La Rocque Rodriguez

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 06/07 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena May, 2007

Combining Programming Paradigms in Asynchronous Systems

Ricardo Gomes Leal Costa, Pedro Martelletto de Alvarenga Bastos, Bruno Oliveira Silvestre and Noemi de La Rocque Rodriguez

{rcosta,pbastos,brunoos,noemi}@inf.puc-rio.br

Abstract. Event-based systems are becoming more and more popular. However, event orientation imposes a state-machine view of the program which is not always natural or easy for the programmer. In this work we discuss how programming language features can facilitate the construction and integration of different high-level programming abstractions, allowing the programmer to view the code in the way that is most appropriate to each situation, even inside one single application.

Keywords: Event-based Programming, Distributed Programming, Programming Language

Resumo. Sistemas orientados a eventos vêm se tornando cada vez mais populares. Entretanto, orientação a eventos impõe um estilo de programação baseado em máquina de estados, o que nem sempre é uma forma natural e fácil para o programador. Neste trabalho nós discutimos como características da linguagem de programação podem facilitar a construção e integração de diferentes níveis de abstrações de programação, permitindo ao programador escolher qual é o mais apropriado para cada situação, mesmo dentro de uma única aplicação.

Palavras-chave: Programação Orientada a Eventos, Programação Distribuída, Linguagens de Programação

In charge for publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introduction

In the last few years the focus of distributed computing has shifted from local area to wide-area networks. Much work has been devoted to studying programming paradigms that are more adequate to this new target environment. It is now widely accepted that asynchronous and event-based systems are more convenient for wide-area distributed applications than synchronous programming models.

Event-oriented programming systems originally became popular in the setting of graphical interfaces. In this context, the developer codes the program as a series of responses to user actions, such as mouse clicks or menu selections. When we extend event-oriented programming to distributed programming, the arrival of a message is viewed as an event, and the developer again codes the program in terms of responses to events. In both cases, the program takes the form of a state machine, in which the current state, coupled with the incoming event, determines what the next state will be. In some situations, creating code in this way is extremely natural. However, in other scenarios, such as requesting a service from a remote node and acting upon its reply, it can be quite unnatural to create code with this model.

In this work we discuss how, with the appropriate language support, one can have the best of both worlds, seamlessly combining programming paradigms in one single application. Our discussion is based on the Lua programming language, which offers some reflective facilities and also some features typical of functional languages, such as functions as first order values. However, our focus is on the effect these features have, which could be obtained in other systems with similar properties.

This work is organized as follows. In Section 2 we give a brief introduction to the ALua system, an event-based distributed programming system based on Lua. Section 3 describes two libraries which support programming abstractions: one for writing event-oriented distributed algorithms and the other one for remote procedure calls. In Section 4, we discuss how both libraries can be integrated seamlessly in one application. Finally, Section 5 contains some final remarks.

2 The ALua System

Lua [Ierusalimschy, Figueiredo e Celes 1996] is an interpreted programming language designed to be used in conjunction with C. It has a simple Pascal-like syntax and a set of functional features, such as first-order function values and closures. Lua implements dynamic typing. Types are associated to values, not to variables or formal arguments. Lua's main data-structuring facility is the *table* type. Tables implement associative arrays, i.e., arrays that can be indexed with any value in the language. When programming in Lua, tables are used to implement different data structures, from ordinary arrays to sets or linked lists, and also to support objects and modules.

Metatables add reflective facilities to Lua. They allow us to change the normal behavior of tables. Of specific interest to us is the possibility of changing language behavior when querying fields that are absent in a table. When a program accesses an absent field in a table T, the interpreter looks for the `__index` method in T's metatable. If there is such a method, it is invoked to provide a result.

The ALua system [Ururahy, Rodriguez e Ierusalimschy 2002] uses Lua to support event-oriented distributed programming. An important feature of interpreted languages in general is the support for executing dynamically created chunks of code. In ALua, messages are chunks of code that will be executed by the recipient. There is only one asynchronous communication primitive in

ALua, `alua.send`, that sends a chunk of Lua code to another process. There is no equivalent to a `receive` primitive. ALua uses an event-driven programming model, where the arrival of a message is treated as an event. Whenever an ALua process becomes idle, the event loop activates a handler for pending events, which may be generated by the user, through the console, or by a remote process, by sending a message. If an ALua process receives a message from another process, it executes this message in the environment of the receiver. The result is an event-driven programming model, compatible with the character of interpreted languages — not very secure, but highly flexible: A programmer can use it to perform simple tasks, such as calling a remote function, but she can also use it for much more complex tasks, such as remotely changing the algorithm a process is executing.

ALua is provided as a library of Lua functions [Rodriguez 2005]. An ALua *daemon* must run on each machine in which ALua processes execute. Besides the *send* primitive, the ALua library provides functions for starting and connecting to daemons, starting or joining an application, and for spawning new processes. An application works as a group of processes: when a process joins an existing application, it receives a list of current processes in this application, and can from that point on send messages to them. Most ALua functions are asynchronous, and can take as an optional argument a callback to be invoked with the result as an argument. So, for example, the programmer can associate a callback to the asynchronous invocation of `alua.send`, and have this callback handle the case of failure in communication.

As we discuss in [Ururahy, Rodriguez e Ierusalimschy 2002], the use of the event-driven paradigm, as of any other programming paradigm, leads programmers to create specific program structures. An important characteristic of ALua is that it treats each message as an atomic chunk of code. It handles each event to completion before starting the next one. Messages must typically be small, non-blocking chunks of code.

3 Programming Abstractions

The ALua basic programming model, in which chunks of code are sent as messages and executed upon receipt, is very flexible, and can be used to construct different interaction paradigms, as discussed in [Ururahy, Rodriguez e Ierusalimschy 2002]. However, programming distributed applications directly on this programming interface keeps the programmer at a very low level, handling large strings containing chunks of code. On the other hand, using features of Lua, we are able to create libraries that offer higher-level communication abstractions and integrate them into the language.

In the next two subsections, we describe two such libraries. The first one implements an event-driven model, very similar to the basic ALua model, plus a small set of services which are convenient for programming distributed algorithms. The other library is LuaRPC, which combines ALua's asynchronous nature with the well-known remote procedure call abstraction. In Section 4, we discuss how both of them can become useful in one same application.

3.1 Distributed Algorithms

The DAlua library was designed as a support tool for teaching distributed algorithms. Its main goal is to bridge the gap between the notation used in classical Distributed Algorithms books and their implementation. This notation is quite similar in spirit to the basic ALua event-driven programming model, but allows a more direct transcription of the distributed algorithms as described

in technical books [Barbosa 1996, Raynal 1988]. So the DAlua library mainly creates new and simplified versions of ALua's functions. The *send* function is redefined as:

```
DAlua.send (dest, messagetype, ...)
```

which takes as arguments `dest`, the destination process, `messagetype`, a string describing the type of message being sent, and a sequence of optional arguments.

Figure 1 shows code using the DAlua library, and also works as an example of the ALua programming model. This is part of the (simplified) code for implementing a classical algorithm for distributed mutual exclusion [Ricart e Agrawala 1981] based on logical clocks (Many improvements were proposed to the algorithm since it was first described, but we use the original algorithm for the sake of simplicity.). In this algorithm, a process desiring to enter its critical region sends a request to all processes in the group and then wait for all of their replies. When a process receives a request for mutual exclusion, it either immediately replies (a reply in this case always means agreement), if it is not interested in entering the critical section or if the other process issued a request with a timestamp prior to the one of its pending request, or defers its reply until it leaves the critical section. To keep the code small, we handle in this case only one request per process at a time. In this simple implementation, we store in global variable `criticalSec` a string containing the code to be executed when it is ok for the current process to enter its critical region.

Function `enterCS` sends *requests* to all other processes in the application and sets some global variables: `waiting` indicates that the process is now in a new state, awaiting authorizations; `timeoflastrequest` records the logical time of this request, and `missing` keeps track of the number of processes which have not yet sent their authorization. Function `DAlua.processes` returns a table with all processes currently in the application, and the call to Lua function `table.getn` returns the size of this table. Function `loadstring` compiles a Lua chunk from a string and returns the compiled chunk as a function, in this case, to be stored in global variable `critictalSec`.

Sending a *request* amounts to having the receiver execute function `request`. This function checks whether the process is in its critical region or is awaiting authorizations to enter it: in this last case, logical times of requests are compared, and if there is a draw, ids of both processes (function `DAlua.self` returns the current's process id) are compared. In the situations in which the receiver must grant authorization, it sends an *oktogo* message back, which, on its turn, makes its receiver execute `oktogo`. Function `oktogo` decrements the count of missing authorizations, and, if this count is zero, executes the critical region.

3.2 Remote Procedure Calls

Organizing code as a series of responses to external events can become quite cumbersome for the programmer, particularly when one conceptual "task" involves several events. As discussed in [Adya et al. 2002], the problem is that the control flow for a single task and the task-specific state is broken across several language procedures, discarding the effectiveness of language scoping features. The classical RPC mechanism [Birrell e Nelson 1984] allows communication with a peer (request and reply) to be handled inside the scope of a single language procedure, but imposes unacceptable synchronism. In [Rossetto e Rodriguez 2005] we discuss the LuaRPC library, in which we bring the benefits of RPC to asynchronous systems, using an asynchronous remote invocation as a basis for communication.

LuaRPC provides a simple interface for issuing asynchronous and synchronous remote procedure calls between ALua processes. Function `rpc.async()` receives as arguments the identifier

```
-- global variables
logicalclock = 0
busy = false
waiting = false
deferred = {} -- constructs empty table

function enterCS(code)
  waiting = true
  timeoflastrequest = logicalclock
  local peers = DAlua.processes("myapp")
  missing = table.getn(peers)
  criticalSec = loadstring(code)
  -- requests to all peers:
  DAlua.send(peers, "request",
             DAlua.self, logicalclock)
end

function request (id, timestamp)
  if not busy and not waiting then
    DAlua.send(id, "oktogo", DAlua.self)
  elseif waiting then
    if timestamp < timeoflastrequest or
      (timestamp == timeoflastrequest and
       id <=  DAlua.self) then
      DAlua.send(id, "oktogo", DAlua.self)
    else
      table.insert(deferred, id)
    end
  elseif busy then
    table.insert(deferred, id)
  end
  -- keep logical clock consistent
  if timestamp < logicalclock then
    logicalclock = logicalclock + 1
  else
    logicalclock = timestamp + 1
  end
end

function oktogo (id)
  missing = missing - 1
  if missing == 0 then
    busy = true
    waiting = false
    -- enters critical region:
    criticalSec()
  end
end

function leaveCS ()
  busy = false
  -- function pairs "traverses" a table:
  for k, pid in pairs (deferred) do
    DAlua.send(pid, "oktogo", DAlua.self)
  end
  deferred = {} -- resets empty list
end
```

Figure 1: Code outline for Ricart-Agrawal

of a remote process, the name of the procedure to be called, and a callback function to be executed when the reply arrives. The returned value is a function — a dynamically created function, whose construction relies on Lua's closures — which can be called one or more times, passed as an argument or assigned to a variable, exactly like an ordinary local function. However, whenever this function is invoked, execution will proceed immediately, and the associated callback will be executed on function return.

The LuaRPC library uses the `rpc.async()` construct to build an alternative, synchronous remote procedure call. The main idea is that the callback function, in the synchronous case, is the "continuation" [Friedman, Wand e Haynes 2001] of the current computation. To implement this idea, we used the *coroutine* [de Moura, Rodriguez e Ierusalimschy 2004].

Coroutines introduce multitasking in a cooperative fashion: each coroutine has its own execution stack, as a thread does, but control is transferred only though the use of explicit control transfer primitives. Using coroutines, a process can maintain several execution lines but only one can run at a time, and the switch between two of them is explicit in the program. This allows applications to improve their availability without the context-switching weight and complexity that may come together with the multithreading solution.

With *LuaRPC*, each invoked procedure executes inside a new coroutine — when a synchronous call occurs, this coroutine yields, returning control to ALua's main event loop. When the reply to the remote call arrives, its callback is a procedure that resumes the suspended coroutine. This allows us to maintain the single-threaded structure of the ALua system while avoiding that a process remains blocked when it issues a synchronous call. Because context switches occur only at explicit points in the code, many issues related to race conditions are eliminated.

A chunk of code using LuaRPC could be something like:

```
local sum = 0
f = rpc.sync(serverid, "foo")
for i=1,limit do
  -- remote invocation (will yield):
  sum = sum + f(i)
end
```

4 Hybrid Programming

Because each of the abstractions we discuss in the previous section is implemented by an independent library, we can mix and match them as we please. This is interesting because it means a programmer need not be restricted to a single communication paradigm when he writes an application. To illustrate this, we discuss in this section an example which combines the use of the LuaRPC and the DAlua libraries. In this example, we assume we are offering a set of services under the form of remote procedures, and that the implementation of these services contains critical sections, which require mutual exclusion among processes running on different machines. As we discussed in Section 3.1, a distributed mutual exclusion algorithm can be easily implemented using the DAlua library. Figure 2 shows how we could combine directly both models in one application, if we encapsulate the implementation of the critical section in function *critical*. In this case, when *remoteService* is invoked, it will first execute a non-critical section, and then use the event-based interface to activate the mutual exclusion algorithm and guarantee that the critical section is executed with mutual exclusion.

This solution is not elegant at all. If we are implementing a remote service using RPC, we would probably like to maintain this programming style along the implementation of the service,

```
function remoteService (args)
  -- non-critical code here
  -- ask for mutual exclusion:
  DAlua.send (DAlua.self,
              "enterCS", "critical()")
end
function critical ()
  -- critical region here
  leave_cs()
  -- possibly more non-critical
end
```

Figure 2: Applying an event-oriented algorithm directly

```
function rpc.enterCS ()
  pendingCS = rpc.getcoroutine()
  DAlua.send(alua.id, "enterCS",
          "pendingCS()")
  rpc.yield()
end
```

Figure 3: Combining the RPC abstraction with event-oriented algorithms

that is, program the whole service in a procedural style and not have to break it across handlers. So, we would probably like to write something like:

```
  rpc.enterCS()
```

and from that point on write code that executes with the guarantee of mutual exclusion.

We can write this call and have *rpc.enterCS* trigger the execution of the mutual exclusion code with the guarantee that the next lines will only execute under the correct condition by writing a few extra lines of code that will provide the bridge beween styles. Figure 3 contains the necessary code.

Function `rpc.getcoroutine` returns a function that, when invoked, will resume the currently running coroutine. `rpc.enterCS` stores this function in a global variable, `pendingCS`, and sends an asynchronous `DAlua` message requesting the execution of this function when mutual exclusion is guaranteed.

As in Section 3.1, this code works only if a process issues only one request for mutual exclusion at a time. This of course is not realistic in a distributed setting. Figure 4 illustrates the need for several ongoing requests for mutual exclusion. In this figure, we can see a process and its event loop. When this process starts handling a remote procedure call, a second one arrives. The handling of the first call involves a critical section, and so the process sends itself an asynchronous message invoking `enterCS`. However, the second remote call is already pending, and is now handled by the event loop. Processing of this second call also involves a critical section, and so we have a new asynchronous call to `enterCS`, and from that point on two mutual exclusion requests running simultaneously.

The complete code for handling simultaneous requests for mutual exclusion is not much larger, in number of lines, than that shown in this work, but demands a little more familiarity with Lua, and for this reason we have chosen to discuss the simplified versions here. We also have not

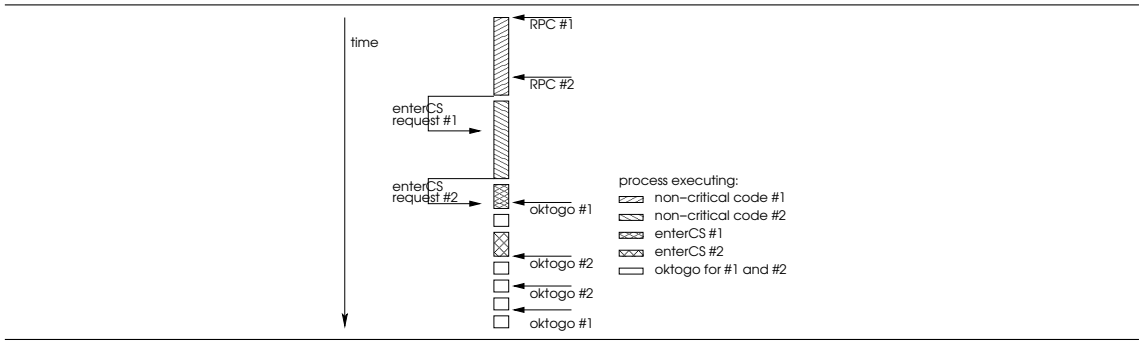

Figure 4: Several requests for mutual exclusion in progress

discussed the proper packaging of the discussed functions into different modules, supported in Lua also by tables.

5 Final Remarks

We discussed in this work how programming distributed asynchronous-oriented systems can become easier with appropriate programming abstractions. This is not a new idea: from the beginning of work on distributed systems, researchers have proposed the use of abstractions such as RPC [Birrell e Nelson 1984]. Many recent systems also provide abstractions for distributed programs [Caromel, Klauser e Vayssiere 1998, Nieuwpoort et al. 2005]. Our main point is that, given appropriate language features, the programmer can combine the advantages of different programming abstractions in one single application.

[Briot, Guerraoui e Lohr 1998] discusses three different approaches to supporting concurrency and distribution in object oriented languages: the *library* approach, the *integrative* approach, and the *reflective* approach. The library approach would be more adapted to system builders, allowing more general software architectures but with harder-to-use interfaces. The integrative approach aims at defining a high-level programming language with few, unified concepts, and would be targeted at application developers. Finally, we discuss how reflection can provide a bridge between both approaches, helping to integrate transparently different libraries within a programming system. Although we are not in an object-oriented context, we believe the same notions apply here. Reflective facilities of Lua, mainly metatables, combined with other programming language features, such as closures, allow us to integrate abstractions very easily into ALua. The advantage of this approach is that we can offer the higher level of abstraction usually provided by languages with built-in communication models and still provide support for different paradigms, allowing the developer to combine different communication abstractions in a single program.

Thanks

We would like to thank Silvana Rossetto, who designed the LuaRPC library, and CNPq, for partial funding of this work.

References

[Adya et al. 2002]ADYA, A. et al. Cooperative task management without manual stack management. In: *USENIX Annual Technical Conference*. Berkeley: [s.n.], 2002. p. 289–302.

[Barbosa 1996]BARBOSA, V. *An Introduction to Distributed Algorithms*. [S.l.]: MIT Press, 1996.

[Birrell e Nelson 1984]BIRRELL, A.; NELSON, B. Implementing remote procedure calls. *ACM Trans. on Computer Systems*, v. 2, n. 1, p. 39–59, fev. 1984.

[Briot, Guerraoui e Lohr 1998]BRIOT, J.; GUERRAOUI, R.; LOHR, K. Concurrency and Distribution in Object-Oriented Programming. *ACM Computing Surveys*, v. 30, n. 3, 1998.

[Caromel, Klauser e Vayssiere 1998]CAROMEL, D.; KLAUSER, W.; VAYSSIERE, J. Towards seamless computing and metacomputing in java. *Concurrency Practice and Experience*, Wiley & Sons, Ltd., v. 10, n. 11–13, p. 1043–1061, Sep-Nov 1998.

[de Moura, Rodriguez e Ierusalimschy 2004]de Moura, A.; RODRIGUEZ, N.; IERUSALIMSCHY, R. Coroutines in Lua. *Journal of Universal Computer Science*, v. 10, n. 7, p. 910–925, jul 2004.

[Friedman, Wand e Haynes 2001]FRIEDMAN, D. P.; WAND, M.; HAYNES, C. T. *Essentials of Programming Languages*. 2nd. ed. [S.l.]: MIT Press, 2001.

[Ierusalimschy, Figueiredo e Celes 1996]IERUSALIMSCHY, R.; FIGUEIREDO, L.; CELES, W. Lua - an extensible extension language. *Software: Practice and Experience*, v. 26, n. 6, p. 635–652, 1996.

[Nieuwpoort et al. 2005]NIEUWPOORT, R. van et al. Ibis: a flexible and efficient Java based grid programming environment. *Concurrency and Computation: Practice and Experience*, v. 17, n. 7-8, p. 1079–1107, June 2005.

[Raynal 1988]RAYNAL, M. *Distributed Algorithms and Protocols*. [S.l.]: Wiley, 1988.

[Ricart e Agrawala 1981]RICART, G.; AGRAWALA, A. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, v. 24, n. 1, p. 9–17, 1981.

[Rodriguez 2005]RODRIGUEZ, N. *ALua: asynchronous distributed programming in Lua*. 2005. `http://alua.inf.puc-rio.br`.

[Rossetto e Rodriguez 2005]ROSSETTO, S.; RODRIGUEZ, N. Integrating remote invocations with asynchronism and cooperative multitasking. In: *Third International Workshop on High-level Parallel Programming and Applications (HLPP 2005)*. Warwick, UK: [s.n.], 2005.

[Ururahy, Rodriguez e Ierusalimschy 2002]URURAHY, C.; RODRIGUEZ, N.; IERUSALIMSCHY, R. ALua: Flexibility for parallel programming. *Computer Languages*, Elsevier Science Ltd., v. 28, n. 2, p. 155–180, 2002.