PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 09/08

Flexibility and Coordination in Event-Based, Loosely-Coupled, Distributed Systems

Bruno Oliveira Silvestre
Silvana Rossetto
Noemi de La Rocque Rodriguez
Jean-Pierre Briot

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 09/08 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena February, 2008

Flexibility and Coordination in Event-Based, Loosely-Coupled, Distributed Systems

Bruno Oliveira Silvestre, Silvana Rossetto[1], Noemi de La Rocque Rodriguez, Jean-Pierre Briot[2]

[1] Universidade Federal Fluminense [2] Laboratoire d'Informatique de Paris 6

brunoos@inf.puc-rio.br, silvana@ic.uff.br, noemi@inf.puc-rio.br, jean-pierre.briot@lip6.fr

Abstract. Coordination languages are tools to manager the interactions among the parts of distributed systems. However, applications with a large scaled distribution and loosely coupled interaction demand flexible coordination mechanisms instead of pre-defined models. We believe that characteristics of dynamic programming languages and coordination libraries can provide more appropriate control on the application interactions.

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

Resumo. Linguagens de coordenação são usadas como ferramentas para gerenciar as interações entre as partes de um sistema distribuído. Mas, devido à diversidade dessas interações em aplicações largamente distribuídas e fracamente acopladas de hoje, modelos de coordenação pré-definidos não conseguem atender os objetivo. Acreditamos que algumas características de linguagens de programação dynâmicas podem contribuir para o uso de bibliotecas de coordenação, que podem ser combinadas para controlar as interações da aplicação de forma mais flexível.

Palavras-chave: Programação Orientada a Eventos, Programação Distriuída, Linguagem de Programação

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introduction

Over the last years, the focus of distributed computing has shifted from local to wide-area networks. In these new environments, because of unknown latencies and of the need to allow systems to scale up to large numbers of participants, loosely coupled, asynchronous systems have gained popularity.

In this setting, a lot of attention has been directed to event-driven programming. Instead of viewing applications as sequential programs with remote invocations to servers, it is now common to see processes in distributed applications as having a reactive behaviour: handlers are executed as reactions to different incoming events. Event-based programs often bear a strong resemblance to state machines. Each incoming event leads the program to a new state, which typically must be recorded in global variables.

However, understanding a program that is coded as a series of responses to different events is not easy when the number of interacting peers is large. One single process may, at any moment, be interacting with dozens of other processes, and each of these interactions may require its own state information. This highlights the need of abstractions to coordinate interacting processes.

The concept of coordination languages as tools to describe the interactions among the parts in a distributed program was much discussed in the nineties [Gelernter e Carriero 1992, Papadopoulos e Arbab 1998]. At the time, most discussion focused on the idea of using a predefined set of coordination primitives, such as Linda's [Carriero e Gelernter 1989] tuple space manipulation operations, to define the communication and synchronization of the application. Coordination models proposed at that time were often focused on tightly-coupled application models. Today, the way in which coordination abstractions are programmed must be evaluated in light of new requirements posed by event-orientation, loose coupling, and dynamic execution environments. These seem to demand more flexible mechanisms, that may be redefined and combined even at runtime.

Primitives for communication and synchronization have classically been offered either through special-purpose programming languages or through libraries. Languages that are designed from scratch to support distribution usually provide a consistent programming model, but are bound to predefined communication patterns designed by the language architects. Communication libraries for conventional languages have the advantage that they can be freely combined in a single application, allowing the programmer to choose the preferred pattern for each set of interactions. However, it is generally the case that libraries must deal with the gaps between the model they implement and that of the host programming language, and that, besides different communication models, each library also imposes a different programming model, making it awkward for the programmer to deal with several of them simultaneously.

The scale and diversity of interactions in Internet-based applications points to the idea that no single pre-defined set of interaction patterns will suffice, even considering specific classes of applications. This indicates the need for environments in which different abstractions and coordination patterns can be built and combined. However, it would be nice to be free to experiment and combine abstractions without having to resort to the relatively low-level programming interfaces traditionally offered by libraries.

We believe programming languages features could make a significant contribution to this issue. Using appropriate language constructs, it is possible to build an arbitrary number of coordination mechanisms from a very small set of primitives. Dynamically typed languages, due to flexible type systems and extension facilities, can allow libraries to be seamlessly added to them, creating environments in which different coordination techniques can be used and combined to compose

new mechanisms. So, instead of looking for the specific coordination mechanism which will be better than others for event-driven programs, we should concentrate on such environments, allowing programmers to create and combine coordination constructs easily.

To evaluate this claim, over the last few years, we have developed a series of coordination libraries in Lua, a dynamically typed interpreted language with a procedural syntax but with several functional characteristics. In this paper, we discuss the results of this development and the role of language features in allowing these libraries to be easily combined. Specifically, we highlight the role of first-class functions, closures, and coroutines. The distributed environment which we analyse is composed by several Lua processes. These could perfectly well incorporate compiled parts written in C or C++, but this is irrelevant for our discussion. We also consider that each of these processes has a single-threaded structure. This has considerable weight in our discussion and is a decision based on the several complexity issues associated to preemptive multithreading [Ousterhout 1996].

Our main contribution is not to present yet another system for distributed programming, but to show the role of language features in allowing different coordination mechanisms to be constructed out of a very small set of primitives, and to be easily mixed and combined.

The rest of this paper is organized as follows. In the next Section, we present a brief introduction to Lua and to the event-driven library we use as a low-level communication mechanism. Section 3 presents the design and implementation of an asynchronous RPC primitive which is the basis for the discussion in the rest of the work. In Section 4, we discuss different coordination patterns that can be built over this asynchronous primitive. Section 5 presents some remarks on performance. In Section 6, we discuss related work and, finally, in Section 7, we present some final remarks.

2 Event-driven distributed programming in Lua

Over the last years, we have been investigating the advantages and limitations of creating distributed programs with a simple event-driven model, based on the Lua programming language [Ierusalimschy, Figueiredo e Celes 1996]. Lua 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. Lua implements dynamic typing. Types are associated to values, not to variables or formal arguments. Functions are first-class values and Lua provides lexical scoping and closures. 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. Tables are used to represent ordinary arrays, records, queues, and other data structures, and also to represent modules, packages, and objects.

The Lua programming language has no embedded support for distributed programming. ALua [Ururahy, Rodriguez e Ierusalimschy 2002] is our basic library for creating distributed event-based applications in Lua. ALua applications are composed of processes that communicate through the network using asynchronous messages. Processes use the `alua.send` primitive to transmit the messages.

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. This avoids race conditions leading to inconsistencies in the internal state of the process. However, it is important that the code in a message does not contain blocking calls, for these would block the whole process, precluding it from handling new messages.

The ALua basic programming model, in which chunks of code are sent as messages and exe-

cuted 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 strings containing chunks of code.

As an example, consider the distributed implementation of the sieve of Erathostenes method for generating primes between 1 an N described in [Magee, Dulay e Kramer 1994], in which each process in the application is responsible for testing divisibility by one specific prime. In this implementation, each process acts as a filter, receiving a stream of numbers. The filter ALua program of Figure 1 prints the first value it receives and stores it in variable `myprime`, for this is the value it will use to test if the remaining numbers are divisible. Subsequent received values are passed on to the next filter if they are not multiples of the first value received. To illustrate Lua's handling of functions as first class values, all numbers are passed from one candidate to the next through calls to `handleCandidate`: Initially, this is the function that stores the received number in `myprime`, but after this first execution, `handleCandidate` is assigned a new value.

The major complexity in the code in Figure 1 is due to the fact that the next filter is created dynamically when the first value is sent to it. This makes sense because we don't know a priori how many processes will be needed. To create a new process, the program invokes function `alua.spawn`, which executes asynchronously, invoking a callback when it is completed. Because of this asynchronous nature, when the second non-divisible number is found we cannot be sure that the next filter is already in place. That is why we test if variable `nextinpipe` exists. If it does not yet exist, the prime numbers are pushed into a list (`numbers2send`). This list is emptied in when the callback for `alua.spawn` is invoked.

Some remarks are in place about string manipulation in Lua. Both quotes, double quotes, and double square brackets may be used as string delimiters. Strings delimited by double square brackets may run for several lines. The `..` is the string concatenation operator in Lua.

3 Asynchronous RPC

Although this basic message-oriented event-driven programming model is powerful, it can be quite error-prone and hard to use. Programmers need tools that allow them to model high-level interaction patterns. This is where programming abstractions come into play. To allow the programmer to deal with higher-level concepts, we have implemented several communication libraries over the last few years, providing support for tuple spaces [Leal, Rodriguez e Ierusalimschy 2003], publish-subscribe [Rossetto, Rodriguez e Ierusalimschy 2004], and remote procedure call [Rossetto e Rodriguez 2005], among others. In this work, we use remote procedure call as the basic communication mechanism, and so we discuss it next.

The RPC abstraction has been adopted in systems ranging from CORBA [Siegel 1996] to .NET [Common Language Infrastructure (CLI) 2006] and SOAP [Mitra e Lafon 2007]. From its inception, however, critiques to the paradigm were made, [Tanenbaum e Renesse 1988] [Birman e Renessee 1994], mostly discussing the imposition of a synchronous structure on the client application and the difficulty of matching RPC with fault tolerance, partly due to its one-to-one architecture. These critiques gain further importance in the context of wide-area networks. However, the familiarity that programmers have with the model must not be ignored. If the synchronous nature of the original proposal is somewhat incompatible with the loose coupling we need in wide-area distribution, we can resort to an asynchronous RPC model. Asynchronous invocations have been long discussed as an alternative [Ananda, Tay e Koh 1992] but the fact is that

```
local myprime
local creatednext = false
local numbers2send = {}
local nextinpipe

function handleCandidate (cand)
  print (alua.id .. ": " .. cand .. " is a prime number")
  myprime = cand
  handleCandidate = testNumber
end

function send2next (cand)
  alua.send (nextinpipe, "handleCandidate (" .. cand .. ")")
end

function spawn_callback(reply)
    for id, proc in pairs(reply.processes) do
      alua.send(id, [[dofile("primop.lua")]])
      nextinpipe = id
      for _, cand in ipairs (numbers2send) do
        send2next (cand)
      end
    end
end

function testNumber (cand)
  if (cand%myprime ~= 0) then
    if not creatednext then
      alua.spawn(1, spawn_callback)
      table.insert(numbers2send, cand)
      creatednext = true
    elseif nextinpipe then
        send2next (cand)
    else
      -- necessary because it is possible that a new process has
      -- been created but the spawn callback has not yet been called
      table.insert(numbers2send, cand)
    end
  end
end
```

Figure 1: ALua code for process in distributed Erathostenes sieve

they are not comfortable to use in traditional sequential programs. When the program is event-based, however, asynchronous invocations are natural, and can be associated to callback functions to be executed upon the completion of the remote invocation.

Our remote procedure mechanism [Rossetto e Rodriguez 2005], provided by the `rpc` library, explores this idea, associating an asynchronous invocation with callback functions over an event-driven model. The basic execution model remains the one we described in the last section with a process handling each incoming message at a time, with the difference that now messages are function invocations.

To provide the same flexibility as we have with normal function values in Lua, the `rpc.async` primitive does not directly implement the invocation, but, instead, returns a function that calls

the remote method (with its appropriate arguments). Figure 2 illustrates the use of `rpc.async` to invoke a remote function `CurrentValue`.

Mandatory parameters for `rpc.async` are the remote process (`server`) and the remote function identifier (`CurrentValue`). An optional argument is a callback function (`register`). In Figure 2, the function returned by `rpc.async` is stored in variable `getCurrentRemValue`. When function `getCurrentRemValue` is invoked (with `"foo"` as the single argument to `CurrentValue`)„ control returns immediately to the caller. At some later point, when the program returns to the event loop and receives the result of the remote function, callback `register` will be invoked, receiving this result as an argument. `register` then assigns the received value to global value `currentRemValue`. (In a more realistic setting, it could also schedule the next update of this value.)

```
-- callback function
function register (value)
   currentRemValue = value
end

-- asynchronous remote function
getCurrentRemValue = rpc.async(server, CurrentValue, register)
   ...
-- remote invocation
getCurrentRemValue("foo")
```

Figure 2: Using `rpc.async`.

Two language features are specially important for allowing the `rpc.async` primitive to return a function that can be manipulated as any other value: (i) functions as first-class values; and (ii) closures.

Having functions as first-class values means they can be passed as arguments to or be used as return values from other functions. A *closure* is a semantic concept combining a function with an environment. The distinguishing feature of closures is the ability to associate an intermediate set of data with a function where the data set is distinct from both the global data of the program and data local to the function itself. As a consequence, closures can be used to hide state, implement higher-order functions and defer evaluation. With these two mechanisms, a function can return a nested function, and the new function has full access to variables and arguments from the enclosing function.

Figure 3 shows the (complete) implementation of the `rpc.async` primitive. Basically, it creates a function (called `f`) that encapsulates the remote invocation. This function receives a variable number of arguments (the `...`, captured in table `args`), which are serialized and sent to the remote process. The callback function (`cb`) is registered to handle the results when they arrive. The request is sent to the remote process through a call to `alua.send`. We believe the concision of this implementation reflects the importance of using a programming language with appropriate flexibility and support for extension.

Thus, the `rpc.async` primitive returns a function, defined inside it, which depends on values passed as arguments on each specific invocation of this operation. Each time the returned function is invoked, a new remote call is performed, which uses the same values for the remote process, remote function, and callback function (a *closure*), but different actual arguments to the remote function.

```
function rpc.async(dest, func, cb)
   local f = function (...)
       -- get the function arguments
       local args = {...}
       -- register the callback
       local idx = set_pending(cb)
       -- process the arguments
       marshal(args)
       local chunk = string.format ("rpc.request(%q, %s, %q, %q)",
                                    func, tostring(args), localId, idx)
       -- send the request
       alua.send(dest, chunk)
   end
   return f
end
```

Figure 3: Implementation of `rpc.async`

Asynchronous interaction and event-driven models

The `rpc.async` primitive allows us to program in an event-driven style with the syntax of function calls for communication.

The event-driven programming model is convenient in that it mirrors asynchronous interactions among processes. However, because we have only one execution line, whenever a process needs to receive an event before continuing execution, the current action must be finalized to wait for the message (that is, the process must return to the event loop to be able to handle the next message). Moreover, to maintain the interactivity, the system must make sure that no message handler takes too long to execute. So, event handlers must run quickly, i.e., long tasks must also be broken into small pieces, between which the system saves the current state and returns to the main loop. In order to do that, the event handler can post a request and schedule the remainder of the current computation to be executed later, as explored in [Welsh, Culler e Brewer 2001]. Typically, to maintain state between the function that is being executed and that which will be executed later, the programmer must resort to global variables, because the current locals will not exist any more at this future point. This process, referred by Adya et al. as *stack ripping* [Adya et al. 2002], is one of the main difficulties for developing applications using the event-driven programming style [Behren et al. 2003].

The closure mechanism can once again come into play to reduce this stack ripping process, by allowing local variables to be maintained in nested functions. When a process makes a remote request and needs to register a *continuation* (or callback) to be executed when the request reply is received, the closure mechanism can be used to encapsulate the values that need to be kept during the request manipulation.

The example presented in Figure 4 illustrates this idea. In this example, the `request` function computes the average of a set of values provided by several processes. For that, the primitive `rpc.async` is used to build asynchronous requests to take values in each remote process and the `avrg` function is defined as the callback function. This function is a closure of `request`, and is thus able to keep the values of the variables `acc` and `repl` (used to compute the average) even when the process returns to the main event loop.

```
-- Array of remote processes
servers = {"ProcA","ProcB","ProcC"}

function request(servers)
  local acc, expected = servers.n, repl = 0

  function avrg (ret)
    repl = repl + 1
    acc = acc + ret.results
    if (repl == expected) then
      print ("Current Value: ", acc/repl)
    end
  end

  -- Request the remote values
  for i = 1, expected do
    -- Create the asynchronous function
    local get = rpc.async(servers[i], getValue, avrg)
    -- Invoke the remote function
    get()
  end
end
```

Figure 4: Exploring the closure mechanism do avoid *stack ripping*.

4 Coordinating Concurrent Activities

The model we described in the previous sections avoids many synchronization issues. Because each event is handled to completion, the fine-grained kind of synchronism one needs with preemptive multithreading, due to the possibility of arbitrary execution interleavings, is not required. However, support is still needed for a number of synchronization and communication issues.

Gelernter and Carriero [Gelernter e Carriero 1992] discuss the advantages of viewing communication and synchronization primitives as means of *coordinating* a concurrent or distributed application. In this section, we adopt this approach and discuss how different coordination abstractions can be provided by libraries that can be combined, either as building blocks, to create further abstractions, or simply as alternative to be used inside an application, as needed. We again focus on language features that allow libraries with these abstractions to be seamlessly integrated into the language.

The issues we discuss can be classified in two major lines. The first of them is the need for different communication abstractions. Programmers do not always want to deal directly with the asynchronous programming model we introduced, based on asynchronous invocations and callbacks. This model is interesting when there is an inherent asynchronism in the interaction itself, as is the case, for instance, in the example presented in Figure 4, in which the contacted peers can reply in arbitrary order. Some other interactions, on the other hand, are inherently synchronous. Consider the case of a client contacting a server for a file which is to be viewed by the user. It may well be more natural for the programmer to code this interaction as a synchronous invocation. Inside a single application, the programmer will typically need to code different interactions, and it would be nice for him to be able to code each of these in the most convenient way. In Sections 4.1 and 4.2 we discuss support for different interaction models.

The second class of abstractions we discuss is the one related to classical synchronization

among concurrent processes, for mutual exclusion and cooperation [Andrews 2000]. Even if, in our model, fine-grained synchronization problems, such as interleaved accesses to global variables, are avoided, we can still have problems occurring at a coarser granularity. Subsequent calls to one process may need to occur with the guarantee that no events were handled between the two (for instance, to guarantee an atomic view of a set of operations). Also, because we are in a distributed setting, we may need to synchronize actions occurring at different processes. Sections 4.3 and 4.4 discuss support for classical synchronization.

4.1 Synchronous RPC

With asynchronous invocations, the programmer must turn control flow upside down, using callback functions to code the *continuation* of the computation after the results of the invocation are available. This directly reflects the event-driven nature of a program, but can not be the best model for the programmer to work with. In this section, we discuss function `rpc.sync`, that creates functions that make synchronous calls to other processes over the same asynchronous communication model. As function `rpc.async`, function `rpc.sync` receives as parameters the process identification and the remote function name Because it is synchronous, the callback parameter does not make sense (in fact, a callback which resumes the current computation will be implicitly built by `rpc.sync`).

We illustrate the use of `rpc.sync` with the code in Figure 5 that repeatedly retrieves a value from a remote process *procA*, uses this value to perform a calculation, and updates the remote process.

```
get = rpc.sync("procA", "getValue")
set = rpc.sync("procA", "setValue")

while true do
   oldvalue = get()
   newvalue = transform(oldvalue)
   set(newvalue)
end
```

Figure 5: Example using `rpc.sync`

Because ALua runs on a single thread, suspending the execution during a synchronous call would block the ALua event loop as well. As a consequence, a process would not receive new requests until the invocation is completed. The most frequent solution to this would be to introduce multithreading. However, we would like to avoid the burden of performance and complexity added by preemptive multithreading [Rossetto e Rodriguez 2005]. For the implementation of `rpc.sync`, we rely on yet another language mechanism: the cooperative multitasking facility offered by Lua *coroutines*.

A coroutine is similar to a thread in that it maintains its own execution stack, but it must issue an explicit control primitive for control to be transferred to any other coroutine. This is interesting because it allows applications to maintain different execution lines while avoiding the complexities of race conditions, but on the other hand it leaves the responsibility of managing the control transfer to the programmer. In the case of syncronous RPC, the control transfer is automatically encapsulated in the remote invocation. Each new computation is handled in a new coroutine and, when a synchronous call is performed, the current coroutine is suspended and

execution flow returns to the ALua loop.

To implement `rpc.sync`, we use `rpc.async` as a basis and again the mechanisms of functions as first-class values and closures. Figure 6 contains a sketch of this implementation.

```
function sync(proc, func)
   -- Create a function to perform the remote invocation
   -- The '...' refers to the variable parameters
   local remote = function (...)
       -- Reference to current coroutine
       local co = coroutine.running()
       -- Create a callback that will resume the execution
       local callback = function (...)
           coroutine.resume(co, ...)
       end

       -- Make the remote invocation
       local aux = rpc.async(proc, func, callback)
       aux(...)

       -- Suspend the current coroutine execution before
       -- returning
       return coroutine.yield()
   end
   return remote
end
```

Figure 6: Implementation of `rpc.sync`

At this point it is convenient to explain some features of Lua coroutines. `yield` and `resume`, respectively, suspend and resume a coroutine execution. `coroutine.resume` receives as its first argument the coroutine to be resumed; any extra argument will be returned by `coroutine.yield`. In the same fashion, any argument passed to `coroutine.yield` will be returned by `coroutine.resume`. This provides a communication channel among coroutines.

Back to the implementation of `rpc.sync`: when function `remote` is called, it first creates a callback that will be responsible for resuming the current coroutine. Then, `remote` invokes `rpc.async` to perform the remote communication (passing the internal callback) and suspends the current execution (`coroutine.yield`). When the results arrive, `rpc.async` calls the internal callback passing these results, which are forwarded to `coroutine.resume`. The coroutine is then resumed and the results are returned to the caller of `remote`. Figure 7 illustrates this behavior.

4.2 Futures

As yet another example of building communication abstractions, we can also implement support for *futures* [Lieberman 1987]. In some cases, the programmer may know, at a certain point of execution, that he needs to schedule a computation whose result will be needed only later. Futures allow the programmer to synchronize actions between processes in a looser relationship. This mechanism can be implemented using `rpc.async` as a basis and the remote call in this case returns a *promise* [Liskov e Shrira 1988], which can be used to retrieve the results later. When the promise is invoked, we use coroutines to suspend execution if the results are not yet available — in a similar way to what we did in the implementation of `rpc.sync`.

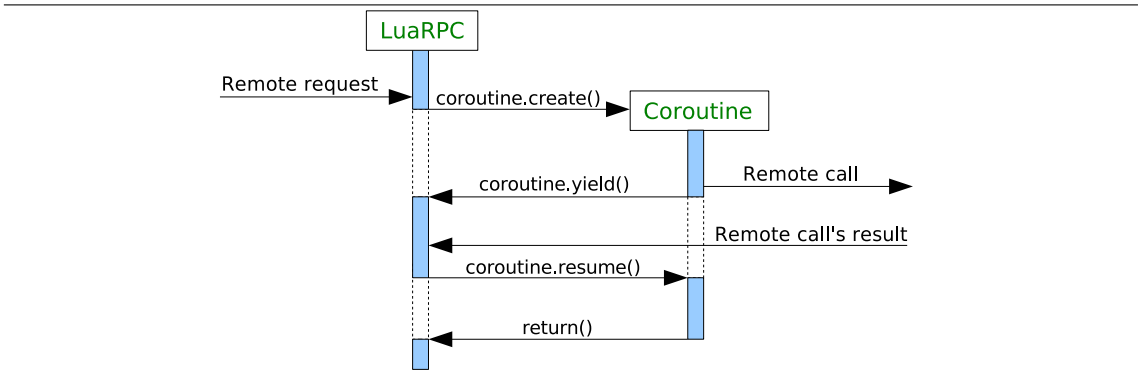

Figure 7: Coroutines in a synchronous call.

Figure 8 shows the implementation of `rpc.future`. The function builds and returns an internal function, which is very similar to the one returned by `rpc.async`, except that this time, when the returned function is invoked, besides calling the remote function asynchronously, it returns the promise, another closure, which may be invoked to synchronize on the results of the asynchronous invocation. The future mechanism uses an internal structure, `result`, to control if the results for the remote call were received. As in `rpc.sync`, a callback is created to handle the results of the asynchronous invocation. This callback fills the future structure with the results and verifies whether the process is blocked waiting for them. The `co` field in the `result` structure indicates that the process has called the promise to retrieve the results, but they were not yet available. (In this case, the promise sets the `co` field and suspends execution of the running coroutine.) As the results are now available, the callback returns them to the suspended coroutine.

4.3 Monitors

In this section we discuss an implementation of monitors [Hoare 1974]. Monitors described here are different from the classic proposals in that they are dynamic: functions may be added to a monitor at any point in execution.

Our implementation for monitors is based on synchronous calls to acquire a lock, that suspends the execution until this lock is acquired. We implement a *monitor* as a structure containing a boolean *lock*, which indicates if the monitor is free, an entrance queue, and the identity of its creator. `monitor.create` creates a new monitor (with no enclosed functions) and returns a reference to it. After an "empty" monitor is created, arbitrary functions can be placed under its protection by calling function `monitor.doWhenFree`, such as:

```
local function set_internal(value)
   -- Do some activities here
end
-- Creates a monitor
local mnt = monitor.create()
set = monitor.doWhenFree(mnt, set_internal)
```

Figure 9 shows the implementation of function `monitor.doWhenFree` This function again creates and returns a new function that encapsulates the one received as a parameter. This new function uses the lock to guarantee the execution in mutual exclusion in relation to other functions in the monitor. `monitor.doWhenFree` also deals with the input parameters and the results. The `pack`

```
function future(proc, func)
   local f = function(...)
      -- Future structure to store the results
      local result = {}
      -- This callback is responsible to receive the results and
      --  put them into the future structure above
      local callback = function(...)
         result.ready = true
         result.values = {...}
         -- If the 'co' field exists, the process is blocked
         if result.co then
            coroutine.resume(result.co, ...)
         end
      end
      -- Create a promise for the invocation
      local promise = function()
         -- If the results are not available, suspend the execution
         if not result.ready then
            result.co = coroutine.running()
            coroutine.yield()
         end
         -- Extract the result from the Lua's table and return them
         return unpack(result.values)
      end
      -- Send the remote request
      async(proc, func, callback)(...)
      -- Return the promise
      return promise
   end
   return f
end
```

Figure 8: Implementation of `rpc.future`

function captures the results in a Lua table that is stored in `rets` variable. After releasing the lock, the result is unpacked and returned.

Functions `monitor.take` and `monitor.release` control lock acquisition as follows. `monitor.take` tries to acquire the lock on a given monitor. If the lock is free, this function switches its value and execution continues normally. If the lock is taken, `monitor.take` puts the current coroutine in the lock's waiting queue and yields. Function `monitor.release` symmetrically, releases the lock on a monitor. It verifies whether there is any coroutine in the monitor entrance queue, and, if so, resumes the first waiting coroutine. Otherwise, `monitor.release` marks the lock as free.

This mechanism for mutual exclusion is different from most classic language proposals in that it does not provide direct syntactic encapsulation of the protected functions. This makes the monitor a dynamic mechanism, allowing functions to be added to the monitor only as needed. This dynamic idea is in some ways captured in the *pthread* API [Butenhof 1997] and in the *lock* interface in the concurrency API of JDK5.0 [Mahmoud 2005]. However, once again, the fact of handling functions as first-class values allows us to create these protected functions and have them behave similarly to the classic, syntactically protected ones. This allows the flexibility of dynamism in a less error-prone environment.

The implementation of `monitor.doWheFree`, based on remote calls, creates the possibility of

```
-- mnt: monitor created to protect the function
-- func: function to execute in mutual exclusion
function doWhenFree(mnt, func)
   -- Reference to the monitor structure
   local idx = mnt.idx
   -- 'from' points to the monitor creator
   local take = rpc.sync(mnt.from, "monitor.take")
   local release = rpc.async(mnt.from, "monitor.release")
   return function (...)
             take(idx)
             -- Invokes the function and captures its results
             local rets = pack(func(...))
             release(idx)
             return unpack(rets)
          end
end
```

Figure 9: Implementation of function `monitor.doWhenFree`.

having a single monitor protecting functions from different processes, supporting distributed mutual exclusion. For instance, a process could create a monitor and add functions to it. Next, the process could pass this monitor to another process, which adds new functions. At the time the monitored functions are invoked, they make remote calls to acquire the lock. However, only one of them will succeed and the others will wait in the queue for the lock to be released. Figure 10 illustrates the use of a distributed monitor. In this example, several distributed processes could receive, upon initialization, calls to a function such as `init`, all of them with the same monitor being received as an argument. Each of the processes could then protect, using this monitor, functions that manipulate a shared state.

```
local isOn = false
local function _off()
   -- Only turns off if the neighbor is 'on'
   if neighbor_state() then isOn = false  end
end
function init(mnt, neighbor)
   -- 'mnt' is a monitor and 'neighbor' is other process
   neighbor_state = rpc.sync(neighbor, "get_state")
   off = monitor.doWhenFree(mnt, _off)
end
```

Figure 10: A distributed monitor.

Our monitor mechanism also offers support for waiting and signalling condition variables, as traditional monitors do. Due to limitations of space, and because it does not introduce new issues, we do not describe this support here.

4.4 Synchronization Constraints and Synchronizers

In this section, we discuss a simple architecture, based on handlers, that permits us to define different handlers to process incoming requests. The handler is selected according to the invoked function, so we can define per-request handlers. If the function does not have an associated han-

dler, a default handler is used. This simply calls the function passing the arguments received in the request and returns the results to the caller. Using this architecture we can explore how alternative coordination policies can be created for a distributed application.

As an example, we turn our attention to the possibility of defining conditions for a function call to be executed. These conditions will allow us to model both intra and inter-process coordination. We defined a handler that uses a queue and a scheduler policy to reimplement the coordination abstractions proposed by Frϕlund and Agha [Frϕlund 1996, Agha et al. 1993]. We chose this proposal because it is one of the few that provides support for distributed as well as for concurrent synchronization.

Intra-object synchronization in [Frϕlund 1996, Agha et al. 1993] is supported by *synchronization constraints*. As with *guards* [Riveill 1995, Briot 2000], the idea is to associate constraints or expressions to a function to determine whether or not its execution should be allowed in a certain state. This kind of mechanism allows the developer to separate the specification of synchronization policies from the basic algorithms that manipulate his data structures, as opposed to monitors, in which synchronization must be hardcoded into the algorithms.

As an example taken from [Frϕlund 1996], consider a radio button object with methods `on` and `off`. To ensure that these methods are invoked in strict alternations, the programmer can define a state variable `isOn`, that indicates whether or not the button is turned on. Synchronization constraints can be defined disabling method `on` when `isOn` is true, and disabling method `off` when it is false.

For inter-object synchronization, Frϕlund proposes the use of *synchronizers*. Synchronizers are separate objects that maintain integrity constraints on groups of objects. They keep information about the global state of groups and permit or prohibit the execution of methods according to this global state.

Keeping the rules in a central point, instead of scattering them among the processes, facilitates modifications and allows using synchronizers in an overlapping fashion. Consider again the example of radio buttons. Besides the individual integrity constraint of alternate invocation, a set of radio buttons must satisfy the constraint that at most one button is on at any time. For this situation, Frϕlund proposes the following solution. A synchronizer keeps the global group state in variable `activated`, whose value is true if any radio button in the set is on. A disable clause in the synchonizer states that, for any button in the set, method `on` is disabled if the value of this variable is true. To ensure the maintance of the global state, synchronizers also support *triggers*: code that is associated to the execution of methods in the individual members of the group controlled by the synchronizer. In the case of our example, a trigger is associated to the execution of method `on`, setting `activated` to true, and to method `off`, setting it to false.

We provide support for these mechanisms through modules `sc` (synchronization constraints) and `synchronizer`, that interact with the handler. Both modules introduce constraints that must be checked by function calls. In the case of the `sc` module, these are local calls that verify the internal state, whereas `synchronizer` permits processes to register themselves as synchronizers of each remote object (or process, in our case) they coordinate.

Figure 11 illustrates how a program could use synchronization constraints to ensure the properties of the radio button example taken from [Frϕlund 1996]. `sc.add_constraint` associates guard functions to the RPC visible functions — those not defined as local. It receives as arguments the name of function to be guarded and the function that implements verification. The latter receives as arguments the request information and must return *true* if the guarded function can be executed or *false* otherwise.

```
local isOn = false
local function can_turn_on(request)
    return not isOn
end
local function can_turn_off(request)
    return isOn
end
function on()
    isOn = true
end
function off()
    isOn = false
end

sc.add_constraint("on", can_turn_on)
sc.add_constraint("off", can_turn_off)
```

Figure 11: Defining synchronization constraints.

Figure 12 illustrates the creation of a synchronizer that coordinates a set of distributed processes that represent radio buttons, enforcing that at most one of them is activated at any time. When one of the buttons receives a request, it contacts the synchronizer in order to verify the remote constraints, which allow or not the button to execute the function according the global state information.

```
local activated = false
local function can_turn_on(request)
   return not activated
end
local function trigger_on(request)
   activated = true
end
local function trigger_off(request)
   activated = false
end
-- defines constraint and triggers for each distributed button
for _, bt in ipairs(buttons) do
   synchronizer.set_trigger(bt, "on", trigger_on)
   synchronizer.set_trigger(bt, "off", trigger_off)
   synchronizer.add_constraint(bt, "on", can_turn_on)
end
```

Figure 12: A synchronizer defining a remote constraint and triggers for a set of distributed buttons.

Both `synchronizer.set_trigger` and `synchronizer.add_constraint` receive, as their argument, the remote process identification, the name of the function in this process, and the function to be executed once the synchronizer is contacted. For triggers, this function typically updates the global state, and for constraints, it must return, respectively, *true* or *false* to permit or prohibit the constrained function execution. Moreover, the function to be executed receives, as a parameter, information about the constrained function into the variable `request`.

To implement synchronization constraints and synchronizers, we developed a handler that manipulates a queue of requests. The handler processes the queue until either it is empty or the

remaining requests cannot be executed due to the synchronization rules. A request is executed only if all of its local constraints and remote verifications evaluate to true. However, when a requested function is executed, we need to restart the queue evaluation because this function can have modified the internal or global states, making some request newly eligible for execution.

We first implemented the scheme as described in [Frϕlund 1996], that is, verifying and executing the requests in a sequential fashion — a new request is handled only after the previous one is completed. However, evaluating remote constraints is expensive because the process communicates with the synchronizer and blocks until the answer arrives. Figure 13-a shows a diagram illustrating this scheme. Although the synchronizer imposes constraints only on function `set`, the process waits the synchronizer's answer arrives before executing the request `get`.

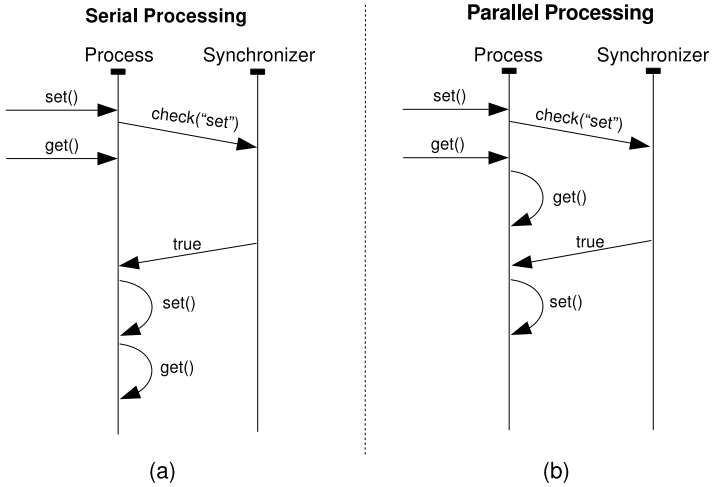

Figure 13: Diagram of (a) the original and (b) our proposals for constraints evaluation.

Using coroutines once again, we implemented an alternative scheme to further explore concurrency in this system. Since we know that the verification of remote constraints verification will block the process, we encapsulated this verification inside a new coroutine, so the process can suspend it while the synchronizer analyzes the constraints, and the process is free to handle other request. The new scheme is shown in Figure 13-b.

Our implementation checks the synchronization constraints before contacting the synchronizers, avoiding the network communication if the local verification fails. However, because new function calls can now modify the internal state during the remote constraints checking, it verifies the synchronization constraints again when it receives all replies from synchronizers in order to guarantee that the internal states still allows the request execution.

5 Performance Results

In this section, we discuss execution times of the mechanisms we described along the paper. Our goal is to evaluate the minimal cost that these mechanisms add to the basic RPC mechanism. We measured the average time for a client to execute an asynchronous RPC request to a remote function in the following cases:

- **RPC:** the basic client/server scheme that is used as reference.

- **Monitor:** the remote function protected by a monitor.
- **Queue Handler:** the same basic scheme used in the case of RPC, but with the server using the queue infrastructure to handle the client requests. This infrastructure is the basis for implementing local and remote constraints.
- **Local Constraint:** the remote function is protected by a local constraint.
- **Remote Constraint:** the function has a remote constraint, which is evaluated by a synchronizer.

In the tests, the client, server, and synchronizer processes execute in different hosts. Moreover, we implemented remote functions that do not receive any argument and return no value (*void* value), and the local and remote constraints are functions that just return *true*.

Table 1 shows the average time (in miliseconds) to execute an asynchronous remote call in each case described above. As a reference, we show the average time for a Java RMI request (using Sun JDK 6), with the same criteria. We performed two tests with RMI, enabling and disabling the just in time (JIT) Java compiler. With JIT disabled, Java runs in interpreted mode only, ans does not generate machine native code. We performed the tests in machines equipped with a Pentium 4 1.7GHz, 256MB RAM and Ethernet 100Mb/s, executing Linux (kernel 2.4.20).

Mechanism	Time
Java RMI – JIT enabled	0.426 ms
Java RMI – JIT disabled	0.803 ms
RPC	1.134 ms
Monitor	1.758 ms
Queue Handler	1.328 ms
Local Constraint	1.338 ms
Remote Constraint	4.281 ms

Table 1: Execution times for different constructs

Our implementation of RPC was not specially performance-oriented, so when compared to RMI, which is built as part of Java's architecture, execution times don't look bad. Here we are focusing specifically on the time to execute a remote call, but in an application this would be part of a mix of activities, which, when using a dynamic language for coordination, should include parts in C or C++ for te hard processing tasks.

The queue handler adds a little overhead to the basic RPC because the request is put in a queue before its execution. On the other hand, a local constraint has almost no cost in our test case, since it is just a function call that returns *true*.

In the remote constraint test, the server must also contact another process, the synchronizer, in order to execute the request. However, the synchronizer implementation is more complex than the distributed monitor, which explains their different performance. Besides the time spent in the network communication with the server, the synchronizer must implement mechanisms to guarantee a consistent view of global state and prevent deadlocks. So the synchronizer uses a transactional protocol with the server.

We performed a second experiment to measure the time to the server to process a set of requests in a serial and parallel fashion, as described in Section 4.4. In this experiment, the server exports

two functions, `get` and `set`, and receives requests of two clients, each one invoking a specific function (Figure 13). The `set` function is protected by a remote constraint, whereas `get` has no protection. First, we configured the server to execute the requests in a serial fashion, i.e., the server processes the next request only when it finishes processing the previous one. We then changed the server behavior to execute a new request while the synchronizer evaluates the remote constraint. Table 2 shows the times (in milliseconds) for each scenario. In the serial case, the request for the `get` function is limited by the evaluation of the remote constraint, so the requests for each function take almost the same amount of time. However, in the parallel configuration, the server can process the `get` requests while the synchronizer evaluates the `set` remote constraint, so that average time of the `get` request is much lower.

Request	Serial Processing	Parallel Processing
`get`	5.138 ms	2.858 ms
`set`	5.247 ms	5.314 ms

Table 2: Serial and Parallel Processing of Requests

6 Related Work

Over the years, support for synchronization and communication abstractions has been implemented in several programming languages, such as Emerald [Black et al. 1987], Orca [H., Kaashoek e A. 1992], Java [Gosling et al. 2005], Erlang [Armstrong et al. 1996], and E [Miller, Tribble e Shapiro 2005], and in a number of libraries, such as SunRPC [Sun Microsystems 1988], Linda [Carriero e Gelernter 1989], JXTA [Gong 2001], ProActive [Caromel, Klauser e Vayssiere 1998], and Chord [Stoica et al. 2001]. Our goal in presenting the implementation of different coordination abstractions is not to discuss the mechanisms themselves, but to support the argument that programming language features can help bridge the gap between the simplicity and flexibility offered, respectively, by the language and library approaches [Briot, Guerraoui e Lohr 1998].

Arbab and Papadopoulos [Papadopoulos e Arbab 1998] present a survey of coordination models and languages, and classify the surveyed works in two major categories. *Data-driven models* offer coordination primitives which can be freely mixed with the computational code. Proposals such as Linda and the synchronizers we discussed in Section 4.4 are placed by the authors in this category. This is also the approach we follow in our model. The other category, *control-driven models*, encompasses models in which the coordinating entities are separate from the computational ones. Typical examples of this category are *configuration* languages [Magee, Dulay e Kramer 1994, Arbab, Herman e Spilling 1993], which focus on describing interconnections between independent processes or components.

Adya and others [Adya et al. 2002] discuss advantages and disadvantages of multithreading and event-based programming, considering the distinction between *manual* and *automatic* stack management. One point that further links their work to ours is the emphasis on allowing the programmer to freely combine both models when coding an application. Eugster and others [Eugster, Guerraoui e Damm 2001] also discuss the importance of combining different interaction paradigms (RMI and publish/subscribe) in a single distributed programming tool.

Many authors have highlighted the importance of object oriented concepts, such as encapsulation and the message passing paradigm, for concurrent and distributed programming [Briot, Guerraoui e Lohr 1998, Agha 1990, Nicol, Wilkes e Manola 1993]. We believe this discussion is orthogonal to ours and do not pursue it in this paper.

There appears to be little work exploring how language features interact with libraries for concurrency and distribution. The work of Varella and Agha [Varela e Agha 2001] and that of Eugster et al. [Eugster, Guerraoui e Damm 2001] are among the few that approach the discussion of distributed programming libraries from a language point of view. Varella and Agha point out the need for extensions to Java, indicating that the original set of language features is insufficient, and on the other hand defends that only a few extensions can suffice to provide a simple distributed programming model. Eugster et al. identify a set of mechanisms which enforce language support for publish/subscribe. Specifically, they identify the concept of closures as one of these mechanisms. Their work also intends to contribute to the discussion about whether to use the integrative or library approach to provide publish/subscribe in object-oriented languages.

The idea of providing synchronous communication facilities over event-driven systems is explored in some worksi [Schmidt e Cranor 1996, Lea, Vinoski e Vogels 2006, Vinoski 2005]. Haller and Odersky [Haller e Odersky 2006] are explicitly interested in avoiding the problems associated to the typical "control inversion" of event-driven systems. However, in the case of their work, the "continuation" of the current computation must be explicitly coded in a receive clause.

7 Final remarks

In this work, we explore some special programming language features — namely, dynamic execution environment, functions as first-class values, closures and coroutines — to build different coordination mechanisms for distributed asynchronous computing. Although we are particularly interested in exploring these features in the Lua programming language, our goal is not to promote this language specifically, but, instead, to contribute the discussion about the role of language features in bridging the gap between the integrative and library approaches for distributed programming. The specific programming system we describe is simply an environment we used to demonstrate that programming abstractions which simplify distributed applications can be easily implemented and combined given a small set of language features.

We defined a basic asynchronous primitive, called `async`, with a callback function, which allows programming in a direct event-driven style with the syntax of function calls for communication among interacting processes. Over this primitive we explored the system's flexibility by building differente well-known coordination abstractions.

By using cooperative multitasking (provided by coroutines) combined with the asynchronous primitive, we implemented the synchronous primitive `sync`, allowing the programmer to have a synchronous view of interactions without having to deal with the classic shared memory issues posed by preemptive multithreading. We also discussed the `future` primitive, which supports deferred invocations. Operations provided by the `async`, `sync`, and `future` primitives allow the programmer to combine different communication paradigms in the same application: one can explore, simultaneously, the active style of the remote call model, or the reactive style of the event-driven model. By considering the diversity of interactions possible in distributed applications, specially in Internet-based applications, it becomes important to allow different programming abstractions to be built and combined into the same application.

For the classical synchronization problems of mutual exclusion and cooperation, we chose to

implement monitors and synchronizers. In both cases, the abstractions we built can be used both for intra and inter-process synchonization. Both our monitors and synchronizers, in line with the dynamic character of our environment, allow synchoniztion constraints to be defined dynamically, at any point in the life cycle of objects and applications.

Finally, languages with the features we emphasize, such as dynamic execution model and support to functions as first-class values and closures, are often interpreted languages. Normally, interpreted languages are not adequate for computing-intensive systems. As discussed in [Ururahy, Rodriguez e Ierusalimschy 2002], the idea is that one should use a dual programming model in which the interpreted language coordinates the application and a traditional compiled language handles the computing-intensive parts.

Acknowledgments

We would like to thank CAPES and CNPq (Brazilian government research agencies) to have been partially supported 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.

[Agha 1990]AGHA, G. Concurrent object-oriented programming. *Commun. ACM*, v. 33, n. 9, p. 125–141, 1990.

[Agha et al. 1993]AGHA, G. et al. Abstraction and modularity mechanisms for concurrent computing. *IEEE parallel and distributed technology: systems and applications*, v. 1, n. 2, p. 3–14, 1993.

[Ananda, Tay e Koh 1992]ANANDA, A. L.; TAY, B. H.; KOH, E. K. A survey of asynchronous remote procedure calls. *SIGOPS Oper. Syst. Rev.*, v. 26, n. 2, p. 92–109, 1992.

[Andrews 2000]ANDREWS, G. *Foundations of Multithreaded, Parallel, and Distributed Programming*. [S.l.]: Addison Wesley, 2000.

[Arbab, Herman e Spilling 1993]ARBAB, F.; HERMAN, I.; SPILLING, P. An overview of manifold and its implementation. *Concurrency: Pract. Exper.*, v. 5, n. 1, p. 23–70, 1993.

[Armstrong et al. 1996]ARMSTRONG, J. et al. *Concurrent Programming in Erlang*. [S.l.]: Prentice Hall, 1996.

[Behren et al. 2003]BEHREN, R. von et al. Capriccio: scalable threads for internet services. In: *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. [S.l.]: ACM Press, 2003. p. 268–281.

[Birman e Renessee 1994]Reliable distributed computing with the Isis toolkit. In: BIRMAN, K.; RENESSEE, R. van (Ed.). [S.l.]: IEEE Computer Society Press, 1994. cap. RPC considered inadequate, p. 68–78.

[Black et al. 1987]BLACK, A. et al. Distribution and data types in Emerald. *IEEE Trans. on Software Engineering*, SE-13, n. 1, jan. 1987.

[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.

[Briot 2000]BRIOT, J.-P. Actalk: A framework for object-oriented concurrent programming - design and experience. In: BAHSOUN, J.-P. et al. (Ed.). *Object-Oriented Parallel and Distributed Programming*. [S.l.]: Hermès Science Publications, Paris, France, 2000. p. 209–231.

[Butenhof 1997]BUTENHOF, D. *Programming with POSIX Threads*. [S.l.]: Addison-Wesley, 1997.

[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.

[Carriero e Gelernter 1989]CARRIERO, N.; GELERNTER, D. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, v. 21, n. 3, p. 323–357, set. 1989.

[Common Language Infrastructure (CLI) 2006]COMMON Language Infrastructure (CLI). 2006. Standard ECMA-335.

[Eugster, Guerraoui e Damm 2001]EUGSTER, P.; GUERRAOUI, R.; DAMM, C. On objects and events. In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. [S.l.: s.n.], 2001. p. 254–269.

[Frϕlund 1996]FRϕLUND, S. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*. [S.l.]: The MIT Press, 1996.

[Gelernter e Carriero 1992]GELERNTER, D.; CARRIERO, N. Coordination languages and their significance. *Commun. ACM*, v. 35, n. 2, p. 97–107, 1992.

[Gong 2001]GONG, L. JXTA: A network programming environment. *IEEE Internet Computing*, p. 88–95, May 2001.

[Gosling et al. 2005]GOSLING, J. et al. *The Java language specification*. third. [S.l.]: Addison-Wesley, 2005.

[H., Kaashoek e A. 1992]H., B.; KAASHOEK, M.; A., T. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, v. 18, n. 3, p. 190–205, mar. 1992.

[Haller e Odersky 2006]HALLER, P.; ODERSKY, M. Event-based programming without inversion of control. In: *Proc. 7th Joint Modular Languages Conference (JMLC 2006)*. Oxford, UK: [s.n.], 2006.

[Hoare 1974]HOARE, C. A. R. Monitors: an operating system structuring concept. *Commun. ACM*, ACM Press, New York, NY, USA, v. 17, n. 10, p. 549–557, 1974. ISSN 0001-0782.

[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.

[Lea, Vinoski e Vogels 2006]LEA, D.; VINOSKI, S.; VOGELS, W. Asynchronous middleware and services. *IEEE Internet Computing*, v. 10, n. 1, p. 14–17, 2006.

[Leal, Rodriguez e Ierusalimschy 2003]LEAL, M.; RODRIGUEZ, N.; IERUSALIMSCHY, R. LuaTS - A Reactive Event-Driven Tuple Space. *Journal of Universal Computer Science*, v. 9, n. 8, p. 730–744, ago. 2003.

[Lieberman 1987]LIEBERMAN, H. Object-oriented concurrent programming. In: ____. [S.l.]: The MIT Press, 1987. cap. Concurrent Object-Oriented Programming in Act 1, p. 9–36.

[Liskov e Shrira 1988]LISKOV, B.; SHRIRA, L. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: [s.n.], 1988. p. 260–267.

[Magee, Dulay e Kramer 1994]MAGEE, J.; DULAY, N.; KRAMER, J. A constructive development environment for parallel and distributed programs. *IEE/IOP/BCS Distributed Systems Engineering*, v. 1, n. 5, 1994.

[Mahmoud 2005]MAHMOUD, Q. *Concurrent Programming with J2SE 5.0*. 2005. Sun Developer Network Article.

[Miller, Tribble e Shapiro 2005]MILLER, M. S.; TRIBBLE, E.; SHAPIRO, J. Concurrency among strangers — Programming in E as plan coordination. In: *Symposium on Trustworthy Global Computing (European Joint Conference on Theory and Practice of Software*. [S.l.: s.n.], 2005. LNCS 3705.

[Mitra e Lafon 2007]MITRA, N.; LAFON, Y. *SOAP Version 1.2 Part 0: Primer*. 2007. W3C Recommendation.

[Nicol, Wilkes e Manola 1993]NICOL, J.; WILKES, C.; MANOLA, F. Object orientation in heterogeneous distributed computing systems. *IEEE Computer*, v. 26, n. 6, p. 57–67, 1993.

[Ousterhout 1996]OUSTERHOUT, J. *Why threads are a bad idea (for most purposes)*. 1996. Invited talk at the 1996 USENIX Technical Conference.

[Papadopoulos e Arbab 1998]PAPADOPOULOS, G.; ARBAB, F. Coordination models and languages. In: *Advances in Computers*. [S.l.]: Academic Press, 1998. v. 46, p. 329–400.

[Riveill 1995]RIVEILL, M. Synchronising shared objects. *Distributed Systems Engineering Journal*, v. 2, n. 2, p. 112–125, June 1995.

[Rossetto e Rodriguez 2005]ROSSETTO, S.; RODRIGUEZ, N. Integrating remote invocations with asynchronism and cooperative multitasking. In: *3rd International Workshop on High-level Parallel Programming and Applications (HLPP)*. [S.l.: s.n.], 2005.

[Rossetto, Rodriguez e Ierusalimschy 2004]ROSSETTO, S.; RODRIGUEZ, N.; IERUSALIMSCHY, R. Abstrações para o desenvolvimento de aplicações distribuídas em ambientes com mobilidade. In: *VIII Simpósio Brasileiro de Linguagens de Programação*. [S.l.: s.n.], 2004. p. 143–156.

[Schmidt e Cranor 1996]SCHMIDT, D.; CRANOR, C. Half-sync/half-async: an architectural pattern for efficient and well-structured concurrent I/O. *Pattern languages of program design 2*, Boston, MA, USA, p. 437–459, 1996.

[Siegel 1996]SIEGEL, J. *CORBA Fundamentals and Programming*. [S.l.]: John Wiley & Sons, 1996.

[Stoica et al. 2001]STOICA, I. et al. Chord: A scalable peer-to-peer lookup service for internet applications. In: *ACM SIGCOMM*. San Diego, CA: [s.n.], 2001. p. 149–160. Code available from `pdos.csail.mit.edu/chord/`.

[Sun Microsystems 1988]SUN MICROSYSTEMS. *RPC: Remote Procedure Call, Protocol Specification, Version 2*. [S.l.], jun 1988. RFC 1057.

[Tanenbaum e Renesse 1988]TANENBAUM, A.; RENESSE, R. van. A critique of the remote procedure call paradigm. In: *EUTECO'88 Conf.* Vienna: [s.n.], 1988. p. 775–783. Participants Edition.

[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, dez. 2002.

[Varela e Agha 2001]VARELA, C.; AGHA, G. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, v. 36, n. 12, p. 20–34, dez. 2001.

[Vinoski 2005]VINOSKI, S. RPC under fire. *IEEE Internet Computing*, v. 9, n. 5, p. 93–95, 2005.

[Welsh, Culler e Brewer 2001]WELSH, M.; CULLER, D.; BREWER, E. Seda: an architecture for well-conditioned, scalable internet services. In: *18th Symp. on Operating Systems Princ. (SOSP-18)*. Banff, Canada: ACM, 2001. p. 230–243.