PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 23/07

Flexibility for Synchronization Abstractions in Distributed Programming

Bruno Oliveira Silvestre
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. 23/07 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena August, 2007

Flexibility for Synchronization Abstractions in Distributed Programming

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

[1] Laboratoire d'Informatique de Paris 6

{brunoos,noemi}@inf.puc-rio.br, jean-pierre.briot@lip6.fr

Abstract. Most proposals for synchronization mechanisms in concurrent and distributed systems have privileged the idea of a single abstraction that can solve all synchronization issues. However, in the context of loosely coupled wide-area applications, it is interesting to have available different synchronization mechanisms, so that the developer may choose the most suitable mechanism for each part of his application. Besides, it is important that synchronization facilities guarantee the desired constraints while imposing as little blocking as possible. In this work, we discuss how the use of dynamic programming language can help us deal with this problem, providing support for building different building blocks and abstractions. We base our discussion on the Lua programming language, and show how we can use it to implement different intra and inter-object synchronization mechanisms proposed in the literature.

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

Resumo. Diversas propostas de mecanismos de sincronização para sistemas concorrentes e distribuídos têm privilegiado a idéia de uma única abstração como solução para todas as questões relacionadas à sincronização. No entanto, em se tratando de aplicações altamente distribuídas com baixo acoplamento entre suas partes, é interessante que haja diferentes mecanismos de sincronizações disponíveis para que o desenvolvedor possa escolher qual deles é o mais apropriado para cada uma das partes da aplicação. Além disso, é importante que tais mecanismos garantam as restrições necessárias impondo o mínimo possível de bloqueio. Neste trabalho nós discutimos como o uso de uma linguagem de programação dinâmica pode ajudar na construção de mecanismos reutilizáveis e abstrações sincronização. Baseamos nossa discussão na linguagem de programação Lua e mostramos como podemos usá-la para implementar diferentes mecanismos propostos na literatura para sincronização intra e inter-objetos.

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

Synchronization mechanisms for concurrent and distributed environments have been widely investigated in the eighties and nineties [Briot, Guerraoui e Lohr 1998]. Since that, interest in the matter seems to have waned, although some groups continue to work in specific niches [Harris et al. 2005, Miller, Tribble e Shapiro 2005]. This may have been at least partially due to an understanding that many of the issues were resolved.

At the time these works were developed, in the eighties and early nineties, interest in distributed applications was centered in local area network applications, mostly modeled as strongly coupled systems. Most of the synchronization techniques proposed in these works were based on this model. In the late nineties and early 00's, however, the focus of interest shifted to wide area applications. Besides interest in Internet applications, we are also witnessing the growth of scenarios such as grid and mobile computing, which impose dealing with high levels of heterogeneity and dynamism. In these new environments, loosely coupled, asynchronous systems have gained popularity. Instead of viewing applications as sequential programs with remote invocations to servers, it is now common to see a distributed application as a passive loop which must handle different incoming events. In this new scenario of loosely coupled applications, some level of synchronization is often necessary. It is still the case that some operations may require mutual exclusion or other types of intra and inter object synchronization. Indeed, the current growth of systems with a peer-to-peer architecture tends to generate new interest in synchronization techniques. While on client-server systems many issues could be dealt with inside a single process, with more symmetrical peer-to-peer approaches, there is often the need to synchronize the activities of the peer processes. However, the way in which these synchronization constraints are programmed must be evaluated in the 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, and less "hard synchronization", in the sense of keeping less execution threads blocked.

We believe dynamic languages have an important contribution to the understanding and investigation of synchronization in these new scenarios. Interpreted and dynamic languages have often been discarded, in the past, due to the performance overhead they impose when compared to more traditional, compiled languages. However, it is now common to use dual programming models, in which a traditional language may be used for the harder computing chores, and an interpreted language for coordination and communication. Besides, in wide-area distributed systems, communication itself imposes costs that often minimize the relative impact of using interpreted languages. On the positive side, these languages, due to flexible type systems and extension facilities, can allow synchronization libraries to be seamlessly added to them, creating environments in which different synchronization techniques can be used and even combined to compose new mechanisms.

In this work, we use the Lua programming language [Ierusalimschy, Figueiredo e Celes 1996, Ierusalimschy 2006] to investigate synchronization in distributed, event-based settings. We discuss how different classic intra and inter-object synchronization mechanisms can be integrated into the language, resulting in an environment in which the programmer can easily apply different synchronization mechanisms. Our emphasis is on the use language features to implement synchronization mechanisms using a few basic facilities, and on the integration of these implemented mechanisms with an asynchronous communication basis. We also discuss how we can remove some blocking from some of the previously proposed mechanisms, in line with the asynchronous nature of many current applications.

This work is organized as follows. Section 2 describes our basic environment, composed by Lua and asynchronous distributed programming libraries. Section 3 discusses how we can implement distributed monitors on this environment. In Sect. 4 we describe how a request queue can be added to our basic environment to implement a scheme for intra and inter-object synchronization. Finally, in Sect. 5 we conclude with a discussion of the presented work.

2 Distributed Programming in Lua

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. 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.

The Lua programming language has no support for distributed programming. However, Lua is easily extended through libraries that can be seamlessly integrated into the language. We have, over the last few years, experimented with several such extensions for distributed programming in Lua, which allow us to combine different programming model in one single environment. In our work with synchronization mechanisms, we have used the ALua and LuaRPC libraries.

ALua [Ururahy, Rodriguez e Ierusalimschy 2002] is a 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.

A message is, by default, a piece of Lua code. The arrival of a message is treated as an event, and the default handler for this event is to execute the received code. Because Lua is an interpreted language, the messages can redefine functions and change the application behavior.

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. For cases in which the application must include blocking calls, such as direct socket manipulation for transfer of byte streams, we provide the *channel* API [Pfeifer et al. 2002].

In line with the asynchronous nature of the ALua, most of its functions return immediately, and receive as one of their parameters a *callback*, that is activated when execution of the requested function is complete.

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 new libraries that offer higher-level communication abstractions and integrate them easier into the language. One of the abstractions we have built is the LuaRPC library [Rossetto e Rodriguez 2005] (implemented by the `rpc` module, which provides support for Remote Procedure Call [Birrell e Nelson 1984].)

Remote procedure calls are typically synchronous. In fact, this is one of the reasons for which they have often been deemed inappropriate for wide-area environments [Saif e Greaves 2001]. In our specific case, the classic blocking structure of an RPC system would also be incompatible with ALua's event-based nature. To deal with this, LuaRPC uses as its basic primitive `rpc.async`.

This function takes as arguments the ALua process identifier where the function must be invoked, the remote function name; and a callback, which will receive as a parameter the results of the invocation (Lua allows multiple returns). However, `rpc.async` does not directly invoke the desired remote function: it returns a new function which, when called, will send the invocation and return the control immediately. The callback will be activated upon completion of the remote call. This design option allows the program to build remote functions and manipulate them as it would any other function value. LuaRPC also maintains Lua's treatment of functions as first-class values across invocations. Remote functions may be passed as arguments in local and remote calls.

Asynchronous call is interesting because it reflects the possibility of immediately proceeding with other calls: the order in which the results are received is not important. However, if the program needs a result to proceed with a thread of activity, synchronous calls may be more convenient for the programmer. Function `rpc.sync` creates functions that make synchronous calls to other processes. It receives as a parameter a process identification and the remote function name.

Function `rpc.sync` uses `rpc.async` for its implementation, allowing us to combine the idea of suspending a computation for a synchronous call with the asynchronous nature of our environment. In this case, the callback that must be executed when the remote function completes is the "continuation" of the invoking function. To implement this idea, LuaRPC uses Lua's coroutines. Each new remote request is executed inside a new coroutine. When a function created by `rpc.sync` makes a remote call, what in fact occurs is an asynchronous call followed by a `yield` from the running coroutine. The callback associated to the asynchronous call is a function that resumes the suspended coroutine when the result arrives. Figure 1 illustrates this behavior.

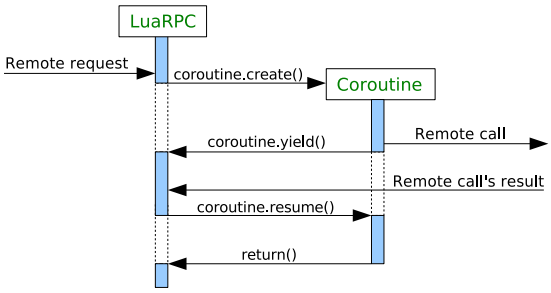

Figure 1: Flow of control with LuaRPC and coroutines

Because LuaRPC was designed as an abstraction layer over ALua, it also maintains the basic ALua model of allowing handler redefinition. On the server side, the requests for execution are processed by handlers (functions) that are selected according to the invoked function. A server can define new handlers for its exported functions, allowing it, for instance, to execute logging or auditing code before invoking the function itself. However, if the function does not have a handler defined, LuaRPC uses a default handler that creates a new coroutine that simply executes the requested function and returns the results to the client.

Synchronization The option for an event-based system, in which each event is handled to completion, avoids many synchronization issues. Even when we introduce context-switching, with coroutines to support the use of synchronous remote calls, the fact that there is no preemption avoids many classic race conditions. However, there are situations in which we may still need to enforce synchronization. Because LuaRPC does not block the entire execution while the answer for a remote call is pending, it potentially creates an internal state consistency problem, even if the

developer knows the possible interruption points. Besides, because we are in a distributed setting, we may need to synchronize actions occurring at different processes. The next sections discuss how we can apply different synchronization mechanisms in this environment.

3 Monitors

In this section we discuss an implementation of monitors [Hoare 1974]. Our implementation for monitors in LuaRPC is based on synchronous calls to acquire and release a lock. The synchronous call suspends the execution until the lock is acquired.

Monitors provide support for mutual exclusion and for condition synchronization. We next discuss in detail how we can implement mutual exclusion.

We define 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. Functions `monitor.take` and `monitor.release` control lock acquisition – again using coroutines. `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, it marks the lock as free. Finally, after an "empty" monitor is created, arbitrary functions can be placed under its protection by calling function `monitor.doWhenFree`:

```
local function _off()
   -- contains synchronous RPC calls
end
-- Creates a monitor
local mnt = monitor.create()
off = monitor.doWhenFree(mnt, _off)
```

Figure 2 shows the implementation of function `monitor.doWhenFree`, to illustrate how language mechanisms facilitate the integration of abstractions into the language. It 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. It also deals with the input parameters and the results. The variable number of the arguments indicated by the "`...`" in function definition is passed to the encapsulated function. The `pack` function captures the results in a Lua table that is stored in `rets` variable. After releasing the lock, the result is unpacked and returned.

The mechanism we have described for mutual exclusion is different from most classic proposals in that it does not provide syntactic encapsulation of the protected functions. This makes the monitor a dynamic mechanism, allowing functions to be protected only when needed. It also creates the possibility of having a single monitor protecting functions from different processes, supporting distributed mutual exclusion. For instance, a process could create a monitor and, add one of its functions to the monitor. Next, it could pass the monitor to another process that adds new functions to the monitor. 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 release. Figure 3 illustrates the use of a distributed monitor.

Our monitor mechanism also offers support for waiting and signaling condition variables, as traditional monitors do. We do not describe this here due to space constraints.

```
module("monitor")

-- mnt: monitor created to protect the function
-- func: function to execute in mutual exclusion
function doWhenFree(mnt, func)
   -- index to the monitor structure
   local idx = mnt.idx
   -- 'from' points to the process that creates the monitor
   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 the results
             local rets = pack(func(...))
             release(idx)
             return unpack(rets)
          end
end
```

Figure 2: Implementation of function `monitor.doWhenFree`.

```
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 = lock.doWhenFree(mnt, _off)
end
```

Figure 3: A distributed monitor.

4 Synchronization Constraints and Synchronizers

In the previous section we described how support for monitors can be added to LuaRPC. In this section we discuss how the handler mechanism in LuaRPC can be used to define yet another building block to facilitate the implementation of other synchronization and coordination abstractions.

Using this handler mechanism, we reimplemented the proposal described in [Frϕlund 1996, Agha et al. 1993] for intra and inter-object synchronization. We chose this proposal because it 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 programmer 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.

For inter-object synchronization, [Frϕlund 1996] and [Agha et al. 1993] propose the use of

synchronizers. Synchronizers are separate objects that maintain integrity constraints on groups of objects. They keep information about the global state of the groups and permit or prohibit the objects method execution according with the global state. To ensure the maintenance of theirs information, synchronizers also support *triggers*: code that is associated to the execution of methods in the individual members of the group.

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

As an example taken from [Frϕlund 1996], Fig. 4 illustrates how a program could use synchronization constraints to ensure that the exported functions `on` and `off` of a radio button are remotely invoked in strict alternations. `sc.add_constraint` associates guard functions to the exported functions. It receives as arguments the name of function to be guarded and the function that implements verification. The latter receives as arguments the request, containing the name and arguments of the invocation.

```
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 4: Defining synchronization constraints.

Figure 5 illustrates the creation of a synchronizer that coordinates a set of distributed 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, allowing or not the button to execute the function, and execute the appropriate triggers.

```
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 5: 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 new handler to LuaRPC that manipulates a queue of requests. The handler processes the queue until this 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 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 6-a shows a diagram illustrating this scheme. The synchronizer `S` imposes constraints on the function `off` and `get_state` does not have any constraint. However, the process `P` executes the request `get_state` after the synchronizer's answer arrives.

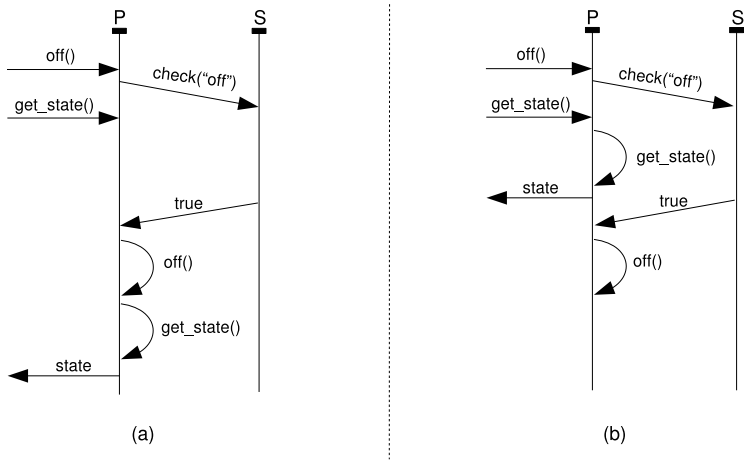

Figure 6: Diagram of (a) the original and (b) our proposals for constraints evaluation.

To explore more concurrency in this system, we implemented an alternative scheme. In this, each request is verified inside a new coroutine, so the process can suspend it while the synchronizer verifies the constraints, and handle other request. This is shown in the Fig. 6-b. The two implementations differ only in two lines of code: one line to create a new coroutine and other line to start its execution.

Our implementation checks the synchronization constraints before contact 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.

As future work, we intend to evolve this implementation to a generic queue processing model, which may be used as a base to create new scheduling and synchronization policies. Synchronizers and synchronization constraints would become one possible policy. The idea is that this

model built over LuaRPC provides for the developer a set of primitives to easily define new synchronization policies — this is somewhat similar to the redefinition of scheduling policies in Converse[Kale et al. 1996] and Proactive [Caromel, Klauser e Vayssiere 1998]. Moreover, the policies could be dynamically changed.

5 Final Remarks

In this work we explored the possibility of building different synchronization mechanisms for concurrent and distributed threads of execution upon a single basis. As opposed to the idea of finding a general solution for the issue of synchronization and embedding it into a programming language, which has been a constant goal for many years [Hoare 1978], we believe each application and even each part of an application may have different synchronization requirements. For example, the constraints in Sect. 4 guarantee that functions start executing only in allowed states, but they do not provide mutual exclusion. However, the application can combine constraints with other abstraction, such as monitor, to assure the mutual exclusion.

So it is important to explore language features that can help us create building blocks that facilitate the creation of the needed synchronization abstractions. Lua is an interpreted programming language and provides functions as first-class values. We use these characteristics in LuaRPC to offer a remote function calls abstraction over the ALua and the handler redefinition mechanism, which is used to implement the synchronization constraints and synchronizers. Furthermore, coroutine allows us to implement synchronous remote calls using asynchronous primitive, and the monitor. It also permits us to introduce more concurrency in the constraints evaluation.

The coupling of synchronization requirements with environments that are event-oriented at their basis is an interesting problem for wide-area computing. The model we explored is one in which coroutines and asynchronous calls allow us to combine synchronization abstractions, while at the same time avoiding the hardships of dealing with preemptive threads. We believe this is a direction that should be further explored.

References

[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.

[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.

[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.

[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.

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

[Harris et al. 2005]HARRIS, T. et al. Composable memory transactions. In: *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. [S.l.]: ACM Press, 2005. p. 48–60.

[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.

[Hoare 1978]HOARE, C. A. R. Communicating sequential processes. *Commun. ACM*, v. 21, n. 8, p. 666–677, 1978.

[Ierusalimschy 2006]IERUSALIMSCHY, R. *Programming in Lua*. second. [S.l.]: lua.org, 2006.

[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.

[Kale et al. 1996]KALE, L. et al. Converse: an interoperable framework for parallel programming. In: *Proc. of IPPS'96: 10th Intl Parallel Processing Symposium*. [S.l.: s.n.], 1996. p. 212–217.

[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.

[Pfeifer et al. 2002]PFEIFER, A. et al. An event-driven system for distributed multimedia applications. In: *Proceedings of DEBS'02 – International Workshop on Distributed Event-Based Systems (held in conjunction with IEEE ICDCS 2002)*. Vienna: [s.n.], 2002. p. 583–584.

[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: *Third International Workshop on High-level Parallel Programming and Applications*. [S.l.: s.n.], 2005.

[Saif e Greaves 2001]SAIF, U.; GREAVES, D. Communication primitives for ubiquitous systems or RPC considered harmful. In: *Workshop on Smart Appliances and Wearable Computing (in conj. with ICDCS'01)*. Mesa, AZ: [s.n.], 2001.

[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.