PUC

ISSN 0103-9741

Monografias em Ciência da Computação

nº 13/11

Embedding Concurrency: A Lua Case Study

Alexandre Rupert Arpini Skyrme

Noemi de La Rocque Rodriguez

Pablo Martins Musa

Roberto Ierusalimschy

Bruno Oliveira Silvestre

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. 13/11
Editor: Prof. Carlos José Pereira de Lucena

ISSN: 0103-9741
September, 2011

Embedding Concurrency: A Lua Case Study

Alexandre Rupert Arpini Skyrme Noemi de La Rocque Rodriguez
Pablo Martins Musa Roberto Ierusalimschy Bruno Oliveira Silvestre[1]

[1] Informatics Institute – Federal University of Goias (UFG)

askyrme@inf.puc-rio.br , noemi@inf.puc-rio.br , pmusa@inf.puc-rio.br , roberto@inf.puc-rio.br ,
brunosilvestre@inf.ufg.br

Resumo. O suporte à concorrência pode ser considerado no projeto de uma linguagem de programação ou provido por construções incluídas, frequentemente por meio de bibliotecas, a uma linguagem sem suporte ou com suporte limitado a funcionalidades de concorrência. A escolha entre essas duas abordagens não é simples: linguagens com suporte nativo à concorrência oferecem eficiência e elegância de sintaxe, enquanto bibliotecas oferecem mais flexibilidade. Neste artigo discutimos uma terceira abordagem, disponível em linguagens de script: embutir a concorrência. Nós utilizamos a linguagem de programação Lua e explicamos os mecanismos que ela oferece para suportar essa abordagem. Em seguida, utilizando dois sistemas concorrentes como exemplos, demonstramos como esses mecanismos podem ser úteis na criação de modelos leves de concorrência.

Palavras-chave: concorrência, Lua, embutir, estender, scripting, threads, multithreading

Abstract. Concurrency support can be considered in the design of a programming language or provided by constructs added, often by means of libraries, to a language with limited or lacking concurrency features. The choice between these approaches is not an easy one: explicitly concurrent languages offer efficiency and syntax elegance, while libraries offer greater flexibility. In this paper we discuss yet another approach, available to scripting languages: embedding concurrency. We take the Lua programming language and explain the mechanisms it offers to support embedding. Then, using two concurrent systems as examples, we show how these mechanisms can be useful for creating lightweight concurrency models.

Keywords: concurrency, Lua, embed, extend, scripting, threads, multithreading

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/

Contents

1 Introduction 1

2 Embedding in Lua 1

3 luaproc 3

4 ALua// 3

5 Performance 4

5.1 Armstrong Challenge 4

5.2 HTTP Server 5

6 Related Work 6

7 Final Remarks 7

References 7

1 Introduction

Traditionally, two approaches can be used in order to provide concurrency support in a programming language: either the language is designed from scratch to incorporate this support, or concurrency constructs are added by means of libraries to a language with limited or lacking concurrency features. Considering support for concurrency in the language design often results in elegant formulations but, on the other hand, in limited scope of use. Such languages are frequently useful only for the type of application originally envisaged by the designers. Libraries generally require a smaller effort to develop and offer more flexibility, allowing the programmer to choose the best concurrency model for each application at hand and even to combine different models in a single application [1]. However, it can be hard to provide arbitrary abstractions over languages that have no support whatsoever for concurrency. How can a library, for instance, hide access to shared memory?

Scripting languages can offer a third approach. In order to explore it, we must recall that there are two standard ways to integrate a scripting language with another language: *extending* and *embedding* [2, 3]. To ease the discussion, let us assume that the scripting language is Lua [4] and the other language is C.

Extending Lua means using C to write new libraries. The main loop of the application runs in a Lua script, which calls functions written in C for extra functionality. For instance, a library written in C can export the POSIX threads (pthreads) model to Lua scripts. This conforms to the second approach mentioned earlier. A common problem with this approach is synchronizing the interpreter: a single main loop usually requires a *global interpeter lock* (GIL) [5, 6], which limits true concurrency.

Embedding Lua means using it to write new functions that can be called from C. The main loop of the application runs in C, which calls Lua scripts for extra functionality. For instance, a C program may use pthreads to run Lua scripts concurrently. This gives us the third approach. With this approach, any global lock is restricted to the C code. Because the real work is done by the scripts, such a global lock creates much less contention.

In this paper we discuss the use of this third alternative to implement two different concurrency abstractions for Lua. We also discuss how some specific Lua mechanisms aimed at embedding are particularly suitable for building concurrency abstractions.

This paper is organized as follows. In section 2 we discuss the mechanisms that Lua offers for embedding and how they can be used for concurrency. In sections 3 and 4 we present two concurrency models based on message-passing and discuss each of their implementations. In section 5 we present some performance analysis results of the previously discussed implementations. In section 6 we present a brief survey of related work. Lastly, in section 7 we draw some final remarks.

2 Embedding in Lua

Unlike most other scripting languages, the design of Lua puts great emphasis on embedding [4, 7]. Among the Lua mechanisms aimed at embedding, here we are particularly interested in *coroutines* and *Lua states*.

Coroutines are similar to cooperative threads, which explicitly request transfers of control. Lua supports *asymmetric* coroutines [8]. Besides a primitive for creating new coroutines (threads), this model offers two other relevant primitives: *resume*, which (re)starts

the execution of a coroutine; and *yield*, which suspends its execution. Coroutines are useful in embedding because they allow a script to return the control to the main loop in the application and continue its execution later. A typical example is a character in a game: for each turn, the engine resumes the character script, which runs some predefined actions and yields.

Lua coroutines are implemented completely inside the interpreter. There is no relation between Lua coroutines and operating system processes or threads. It is easy to write a simple scheduler in Lua to emulate non-preemptive concurrency using coroutines [9]. Nevertheless, coroutines by themselves do not support simultaneous execution of multiple flows in multi-processor or multi-core machines.

Most scripting language interpreters maintain state in C global (extern) variables. For extending, this design is not a problem. However, embedding means using the interpreter as a library, and libraries should not keep internal, static, state. The Lua interpreter maintains its whole state in an instance of a structure called *Lua state*.

Before any interaction with Lua, an application must create a Lua state. Thereafter, any call to the Lua-C API gets this state as its first argument. When the application is done with a state, it can close the state, so that all its resources are freed.

An application may create and manipulate multiple states. Each state is completely independent; any communication between states must be provided by the application, through C code. Because the state data structure is quite light, a single C program running in an off-the-shell desktop can create and manipulate hundreds of thousands of Lua states.

The programming models we describe in this paper use multiple independent states in Lua in order to explore the concept of *Lua processes*, lightweight execution flows of Lua code able to communicate only by message passing. Each Lua process runs in its own Lua state, without any form of memory sharing. This implementation results in a behavior similar to that of operating system (OS) processes, but lighter.

A set of *workers*, kernel threads implemented in C with the POSIX Threads library (pthreads) [10], are responsible for effectively executing Lua processes. Workers repeatedly take a Lua process from a ready queue and run it either to completion or until a blocking operation is performed. Potentially blocking operations *yield* control back to C code; the worker then can enqueue the Lua process if it should block. When the requested operation is completed, if the Lua process was blocked, it returns to the ready queue to have its execution resumed.

Both the systems we will discuss are based on message passing. There is of course no single solution for process communication, but in general the option for no memory sharing simplifies parallel programming, avoiding fine-grained race conditions. Message-passing primitives are implemented in C and exported to Lua (therefore doing a small *extension* to Lua). Messages are managed through a shared C data structure, accessible only to the internal worker code and controlled through conventional locking operations.

Lua states are fundamental for guaranteeing that Lua processes communicate only through messages. This approach is in contrast with work such as the one described in [11], which had to resort to separate OS processes communicating through sockets to create actors that communicate only though message passing.

3 luaproc

The programming model we present in this section combines concurrent execution of Lua code with a deterministic, synchronous, communication scheme. It offers Lua processes through a library called *luaproc* [12] and is further detailed in [13]. As discussed on the previous section, independent Lua states are used and thus there is no shared memory between Lua processes, which rely exclusively on message passing for communication.

In luaproc, processes have no identifiers, and messages are sent to and received from *channels*. Channels are entities on their own, without direct relation to specific processes, and must be explicitly created. A channel is identified by a name (an arbitrary string specified when the channel is created). Lua processes only need to know the name of a channel in order to use it.

Each message carries a tuple of values with basic Lua data types. More complex or structured data can be transmitted either by serializing it beforehand, as detailed in [9], or by encoding it as a string of Lua code which will be later executed by the receiving process.

Sending a message is a synchronous operation, which means control will return to the sending process only after the message is delivered. Receiving a message, on the other hand, can be either a synchronous or asynchronous operation, depending on a parameter passed to the *receive* function.

4 ALua//

We have also been working with concurrency in event-driven systems [14]. ALua// is a system which provides a basis for the creation of distributed, event-driven applications written in Lua.

In ALua//, each event is processed to completion before the next one can be handled. Lua processes send messages that are chunks of Lua code. Each process has a unique identifier in the system, and these identifiers are used for specifying the destination of a message. ALua//'s *main loop* is responsible for receiving an event and dispatching it to the respective handler — the default handler is the execution of the enclosed code. Control returns to the loop only after the handler has finished. This creates a simple model in which concurrency issues can be ignored, but that also limits the tasks that may be executed inside a handler to non-blocking, short chunks of code.

We are also investigating, with ALua//, the possibility of combining multithreading with events. Its architecture allows us to employ threads in order to deal with several event handlers inside the same processes, in addition to the distributed behavior.

The ALua// model again uses Lua processes with corresponding Lua states. However, each ALua// process has its own event loop. When creating a new process, the programmer can choose whether to host it in the current operating system (OS) process or in a different one.

The ALua// API offers a single function for Lua processes to send events. This function abstracts the location of the destination process, which may be in the same OS process, in another OS process in the same machine, or in a different machine in the network. The implementation can choose the appropriate way to deliver messages according to destination.

5 Performance

In this section we present the results of some performance tests with luaproc and ALua//. We also present a comparative view to evaluate them against Erlang [15]. Since Erlang has built-in support for concurrency and is well regarded for its massive concurrency support and efficient implementation of lightweight processes, we believe this comparison is a good benchmark for a library implementation of concurrency.

5.1 Armstrong Challenge

Joe Armstrong proposes a challenge [16] that can be useful to determine the number of processes that a language or library can support and the extent to which increasing the number of processes degrades performance. The challenge consists of putting N processes in a ring, sending a message around this ring M times, and increasing N until the system crashes. It also suggests answering a few questions after running the challenge, such as how long did it take to start the ring, how long did it take to send a message and when did the ring crash.

In our implementation of the challenge, both in Lua and in Erlang, we created rings of up to 100,000 processes and passed a simple message 100 times through the ring. We measured, in seconds, both the time taken to create the processes and the time taken to complete passing the message around the ring for the specified number of times. In order to do so, we used the `statistics( wall_clock )` call in Erlang and the `os.time` function in Lua. Code in Erlang was executed in interpreted (escript) and compiled (erl) modes, both with SMP support enabled. We ran the test on a machine with an Intel Core 2 Quad Q6700 processor with four cores of 2.66GHz and 3GB of RAM, running Fedora 8 kernel 2.6.26.8-57.fc8 #1 SMP.

Figure 1 shows the time taken to create up to 100,000 processes using luaproc and Erlang. It is worth noticing that times measured for luaproc account both for creating the processes and for creating a communication channel for each of them. Also, we were not able to create 80,000 processes or more using interpreted Erlang code, as our ring implementation would exhibit an error indicating not enough memory was available to complete its execution.

It is clear that compiled Erlang shows the best performance with constant and very small times (under one second) for process creation. However, when using interpreted code, luaproc shows better performance results.

Figure 2 shows the average time taken to send a single message. We calculated this time by dividing the total time required to pass the message around the ring for the specified number of times by the total number of actual send operations (100 * N).

Once again, compiled Erlang shows the best performance, with very small times to send messages. Also once more, when running interpreted code, luaproc shows better results than escript. As it can be observed, both luaproc and Erlang show almost constant times to send messages, despite the increasing number of processes.

Lastly, we observed the ring would crash with around 300,000 processes in luaproc, around 80,000 processes in interpreted Erlang and around 2,500,000 processes in compiled Erlang.

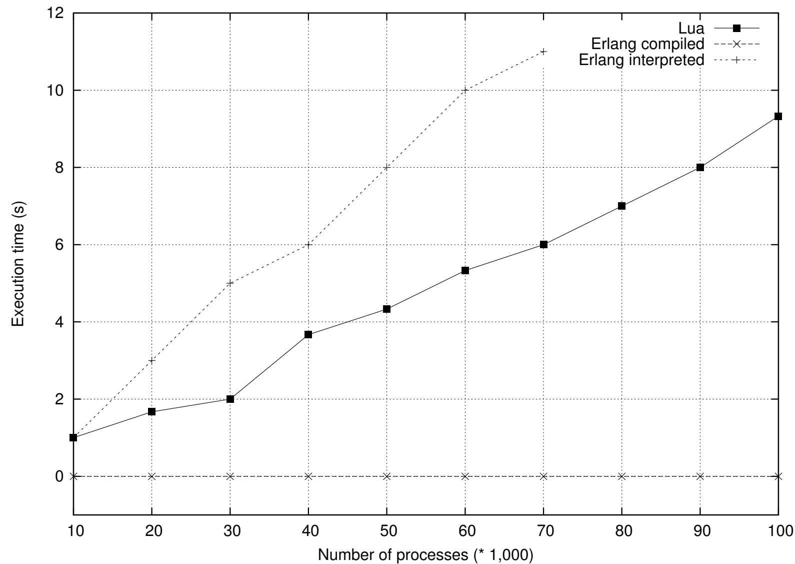

Figure 1: Time taken to create up to 100,000 processes in our ring implementation.

5.2 HTTP Server

In this section we present an HTTP benchmark in order to test both an Erlang based webserver and a Lua based webserver enhanced with ALua//. We performed some tests using the latest version of the Yaws HTTP server [17], implemented in Erlang, and a variant of Xavante HTTP server [18], which is implemented in Lua. We modified Xavante in order to leave an ALua// process responsible for listening at the server socket and dispatching the new connections to a set of ALua// processes that handle the HTTP requests.

We ran the tests with a client sending multiple simultaneous requests to the server. We measured the average number of serviced requests per second for dynamic content (an HTML page where the numbers from 1 to 5,000 are generated). We used the Apache HTTP server benchmarking tool (ab) [19] as a client. The HTTP client was executed in a machine with an Intel Core 2 Duo E6750 processor with two cores of 2.66GHz and 1GB of RAM, running Fedora 8 kernel 2.6.26.8-57.fc8. The HTTP servers were executed on the same machine where the luaproc tests were performed.

Yaws ran with its default configuration, but we tested Xavante with differents parameters. For example, we used 100, 200, 400, and 800 pre-created ALua// processes that handle the connections. Also, we experimented with 2, 4, and 8 workers to execute the ALua// processes.

The number of pre-created ALua// processes did not cause significant difference in the benchmark, so we present the results for 800 ALua// processes in figure 3. The concurrency level refers to the number of simultaneous requests launched by the client.

We can observe that the Yaws webserver was able to serve more requests than Xavante with 2 threads as workers, since Yaws used the four available cores. On the other hand, it is interesting to notice the obvious speedup in Xavante when the number of workers increased to 4 and 8. We can also notice that Yaws served less requests when the client augmented the number of requests per second.

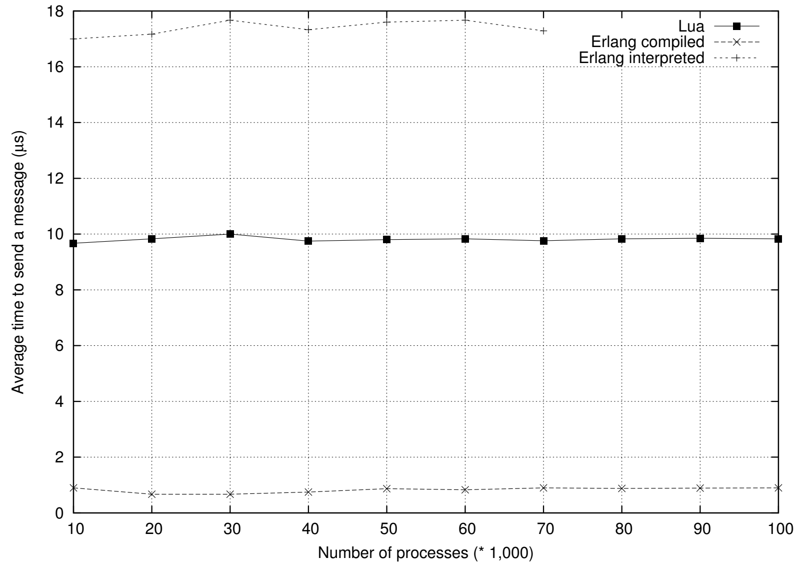

Figure 2: Average time required to send a message in our ring implementation.

6 Related Work

Concurrency in scripting languages is yet to be thoroughly explored. However, there are some implementations focused on providing concurrency support in scripting languages, both by extending the languages and by creating concurrency libraries. *LuaLanes* [20], for instance, is another library for multithreating in Lua which is similar to the luaproc library presented in this paper. It allows multiple independent execution flows (or *lanes*) to run in parallel and offers a tuple space inspired on Linda [21] to allow for communication between them. Other popular scripting languages such as Python and Ruby, both of which include some native support for concurrency, also have alternative concurrency implementations.

Native concurrency in standard Python relies on the operating system to supply (kernel) thread support and is built on top of a conventional preemptive multithreading with shared memory model. However, despite being able to use kernel threads, the potential for parallel execution is hampered by its *global interpreter lock* (GIL) [5, 22]. Alternative implementations for Python such as *Stackless Python* [23] offer a complete Python distribution which includes an enhanced interpreter. Stackless Python supports *microthreads* (or *tasklets*) which are user threads managed by the interpreter and scheduled by a built-in scheduler; it also supports communication channels for inter-process communication. The *greenlets* package [24] is derived from Stackless Python and can be used with the standard Python interpreter as a C extension. It supports greenlets, which work as microthreads without implicit scheduling and are regarded as a coroutine implementation. *Stage* [25] is an actor model [26] programming language based on Python. It extends and modifies the Python programming language constructs to provide a new language that presents abstractions that make the actor model more consistent with object oriented methodologies. Here we can observe both language-based (Stackless Python, Stage) and library-based approaches (greenlets).

Native concurrency in standard Ruby relies on user threads managed by the interpreter. In earlier versions these threads were refered to as *Ruby threads* and were serviced by a single kernel thread. Ruby threads were preemptive and could use shared memory to communicate. In later versions, however, Ruby introduced *fibers*, which are user

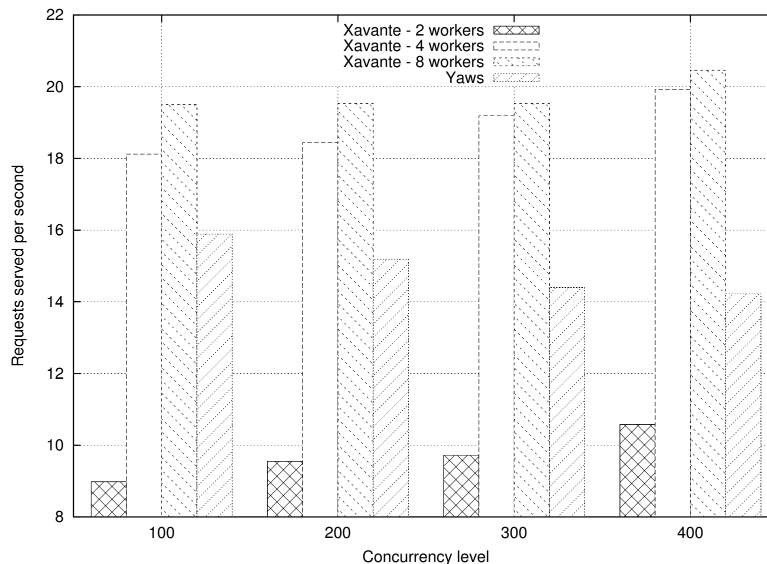

Figure 3: HTTP requests served per second in Yaws and Xavante with 2, 4 and 8 workers.

threads that rely on cooperative scheduling. Fibers are *semi-coroutines*, which are like asymmetric coroutines but can only transfer control back to their caller. Regardless, the potential for parallel execution in Ruby is still hampered, as in Python, by its global interpreter lock (GIL) [6]. Ruby was used to create an implementation of the actor model, influenced by the Erlang programming language, which is also named *Stage* [11], despite holding no direct relationship with the Python implementation of the same name. Another alternative implementation for concurrency in Ruby is *JRuby* [27], a pure Java implementation of the Ruby programming language. Concurrency in JRuby is based on Java threads, which means Ruby threads can be mapped to kernel threads without a global interpreter lock between them [6]. Here both approaches are language-based.

7 Final Remarks

The discussion about library-based versus language-based approaches to concurrency is an old one [28]. In this paper, we discussed a third approach to building concurrency support into a programming language, based on embedding. We showed how mechanisms originally created to support embedding, such as states and coroutines, can be useful for integrating lightweight concurrency models into Lua.

In the conventional choice between language-based and library-based concurrency, libraries usually result in more cumbersome notations and programming [28]. One aspect we did not discuss in this work is the level of elegance we can attain when using embedding to provide concurrency to scripting languages. This aspect should be the object of further investigation. However, features such as dynamic typing and first-class functions with lexical scoping, available in Lua, tend to facilitate seamless integration between libraries and language [29].

References

[1] ODERSKY, M.. **Tackling Concurreny – Language or Library?** Presentation given at Intel Berkeley Research Center, November 2006.

[2] LEFKOWITZ, G.. **Extending vs. Embedding**. Website, 2003. http://twistedmatrix.com/users/glyph/rant/extendit.html.

[3] Python Software Foundation. **Python v3.1.3 documentation**, January 2011. http://docs.python.org/py3k/extending/index.html.

[4] IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; FILHO, W. C.. **Lua – an Extensible Extension Language**. Software – Practice and Experience, 26(6):635–652, 1996.

[5] BEAZLEY, D.. **Understanding the Python GIL**. In: PRESENTATION AT PYCON 2010 - ATLANTA, GEORGIA, 2010. http://www.dabeaz.com/python/UnderstandingGIL.pdf.

[6] **Concurrency is a Myth in Ruby**. Website, 2008. http://www.igvita.com/2008/11/13/concurrency-is-a-myth-in-ruby/.

[7] IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; CELES, W.. **The Evolution of Lua**. In: HOPL III: PROCEEDINGS OF THE THIRD ACM SIGPLAN CONFERENCE ON HISTORY OF PROGRAMMING LANGUAGES, p. 2–1–2–26, New York, NY, USA, 2007. ACM.

[8] MOURA, A. L. D.; IERUSALIMSCHY, R.. **Revisiting Coroutines**. ACM Trans. Program. Lang. Syst., 31:6:1–6:31, February 2009.

[9] IERUSALIMSCHY, R.. **Programming in Lua, Second Edition**. Lua.org, 2006.

[10] IEEE. **IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)**. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995.

[11] SILLITO, J.. **Stage: Exploring Erlang style concurrency in Ruby**. In: INTERNATIONAL WORKSHOP ON MULTICORE SOFTWARE ENGINEERING (IWMSE), p. 33–40, New York, NY, USA, 2008. ACM.

[12] SKYRME, A.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **luaproc: a concurrent programming library for Lua**. Website, 2009. http://luaforge.net/projects/luaproc/.

[13] SKYRME, A.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **Exploring Lua for Concurrent Programming**. JUCS, 14(21):3556–3572, 2008.

[14] URURAHY, C.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **ALua: Flexibility for parallel programming**. Computer Languages, 28(2):155–180, Dec. 2002.

[15] ARMSTRONG, J.. **Programming Erlang**. Pragmatic Bookshelf, City, 2007.

[16] ARMSTRONG, J.. **Concurrency Oriented Programming in Erlang**. SICS Talk, 1992. http://www.sics.se/~joe/talks/ll2_2002.pdf.

[17] **Yaws – an HTTP high perfomance 1.1 webserver written in Erlang**. Website, 2010. http://yaws.hyber.org.

[18] **The Xavante Lua Web Server**. Website, 2010. http://keplerproject.github.com/xavante/.

[19] The Apache Software Foundation. **ab – Apache HTTP server benchmarking tool**, 2009. http://httpd.apache.org/docs/2.0/programs/ab.html.

[20] KAUPPI, A.. **Lua Lanes – Multithreading in Lua**. Website, 2009. http://kotisivu.dnainternet.net/askok/bin/lanes/.

[21] GELERNTER, D.; CARRIERO, N.. **Coordination Languages and their Significance**. Commun. ACM, 35(2):97–107, 1992.

[22] BEAZLEY, D.. **Inside the Python GIL**. In: PRESENTATION AT PYTHON CONCURRENCY WORKSHOP - CHICAGO, 2009. http://www.dabeaz.com/python/GIL.pdf.

[23] **Stackless Python**. Website, 2010. http://www.stackless.com.

[24] **greenlet**. Website, 2010. http://pypi.python.org/pypi/greenlet.

[25] AYRES, J.; EISENBACH, S.. **Stage: Python with Actors**. In: INTERNATIONAL WORKSHOP ON MULTICORE SOFTWARE ENGINEERING (IWMSE), May 2009.

[26] AGHA, G.. **Actors: A Model of Concurrent Computation in Distributed Systems**. MIT Press, Cambridge, MA, USA, 1986.

[27] **JRuby**. Website, 2010. http://jruby.org.

[28] BRIOT, J.; GUERRAOUI, R. ; LOHR, K.. **Concurrency and Distribution in Object-Oriented Programming**. ACM Computing Surveys, 30(3), 1998.

[29] SILVESTRE, B.; ROSSETTO, S.; RODRIGUEZ, N. ; BRIOT, J. P.. **Flexibility and Coordination in Event-based, Loosely Coupled, Distributed Systems**. Comput. Lang. Syst. Struct., 36(2):142–157, 2010.