Pontifícia Universidade Católica
do Rio de Janeiro

Liander Millán Fernández

Concurrent Programming in Lua - revisiting the Luaproc library

Dissertação de Mestrado

Dissertation presented to the Programa de Pós-Graduação em Informática, of the Departamento de Informática da PUC-Rio, as partial fulfillment of the requirements for the degree of Mestre.

Advisor: Prof. Noemi de La Rocque Rodriguez

Rio de Janeiro
December 2016

Pontifícia Universidade Católica
do Rio de Janeiro

Liander Millán Fernández

Concurrent Programming in Lua - revisiting the Luaproc library

Dissertation presented to the Programa de Pós-Graduação em Informática, of the Departamento de Informática do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requirements for the degree of Mestre.

Prof. Noemi de La Rocque Rodriguez
Advisor
Departamento de Informática — PUC-Rio

Prof. Ana Lúcia de Moura
Departamento de Informática — PUC-Rio

Prof. Roberto Ierusalimschy
Departamento de Informática — PUC-Rio

Prof. Márcio da Silveira Carvalho
Coordinator of the Centro Técnico Científico da PUC-Rio

Rio de Janeiro, December 12th, 2016

Liander Millán Fernández

The author graduated in Computer Science from University of Havana in 2011. His main interests are in Distributed Algorithms, and in Concurrent and Parallel Computing.

Bibliographic data

Fernández, Liander Millán

Concurrent Programming in Lua - revisiting the Luaproc library / Liander Millán Fernández ; Advisor: Noemi de La Rocque Rodriguez. — 2016.
68 f. : il. ; 30 cm

Dissertação (Mestrado em Informática)-Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2016.
Inclui bibliografia

1. Informática – Teses. 2. Concorrência. 3. Paralelismo. 4. Lua. 5. Troca de mensagens. 6. Luaproc. I. Rodriguez, Noemi de La Rocque. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

To my advisor Noemi Rodriguez, for all the transmitted knowledge; for the support and understanding before the difficulties that arose in the way, for each advice that made possible to accomplish this work.

To PUC-Rio, for the generous scholarship.

To my friends, for always supporting me.

To my family, without them it would not have been possible to make it so far. I will always be in debt to them.

To Jenny Padron and my little Flavia, for always being there for me, giving me so much support, confidence, happiness and love.

For you all, my sincere thank you!

Abstract

Fernández, Liander Millán; Rodriguez, Noemi de La Rocque (Advisor). **Concurrent Programming in Lua - revisiting the Luaproc library**. Rio de Janeiro, 2016. 68p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

In recent years, the tendency to increase the performance of a microprocessor, as an alternative solution to the increasing demand for computational resources of both applications and systems, has decreased significantly. This has led to an increase of the interest in employing multiprocessing environments. Although many models and libraries have been developed to offer support for concurrent programming, ensuring that several execution flows access shared resources in a controlled way remains a complex task. The Luaproc library, which provides support for concurrency in Lua, has shown some promise in terms of performance and cases of use. In this thesis, we study the Luaproc library and incorporate to it new functionalities in order to make it more user friendly and extend its use to new scenarios. First, we introduce the motivations to our extensions to Luaproc, discussing alternative ways of dealing with the existing limitations. Then, we present requirements, characteristics of the implementation, and limitations associated to each of the mechanisms implemented as alternative solutions to these limitations. Finally, we employ the incorporated functionalities in implementing some concurrent applications, in order to evaluate the performance and test the proper functioning of such mechanisms.

Keywords

Concurrency; Parallelism; Lua; Messages exchange; Luaproc;

Resumo

Fernández, Liander Millán; Rodriguez, Noemi de La Rocque. **Programação Concorrente em Lua - revisitando a biblioteca Luaproc**. Rio de Janeiro, 2016. 68p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Nos últimos anos, a tendência por aumentar o desempenho de um microprocessador, como uma solução alternativa para a crescente demanda por recursos computacionais de aplicações e sistemas, diminuiu significativamente. Isto levou a um aumento do interesse em utilizar ambientes multi-processados. Embora muitos modelos e bibliotecas tenham sido desenvolvidos para oferecer suporte à programação concorrente, garantir que vários fluxos de execução acessem recursos compartilhados de forma controlada continua a ser uma tarefa complexa. A biblioteca Luaproc, que oferece suporte para a concorrência em Lua, mostrou alguma promessa em termos de desempenho e casos de uso. Nesta tese, nós estudamos a biblioteca Luaproc e incorporamos-lhe novas funcionalidades a fim de torná-la mais amigável e estender o seu uso a novos cenários. Primeiro, nós apresentamos as motivações para nossas extensões a Luaproc, discutindo formas alternativas de lidar com as limitações existentes. Em seguida, nós apresentamos requisitos, características da implementação e limitações associadas a cada um dos mecanismos desenvolvidos como soluções alternativas a essas limitações. Finalmente, nós utilizamos as funcionalidades incorporadas na implementação de algumas aplicações concorrentes, a fim de avaliar o desempenho e testar o funcionamento adequado de tais mecanismos.

Palavras-chave

Concorrência; Paralelismo; Lua; Troca de mensagens; Luaproc;

Contents

List of figures

List of tables

1
Introduction

In the search for the necessary computing power to execute increasingly demanding applications, microprocessors maintained, during many years, a constant evolution [1], mainly by increasing their processing speed. However, in recent years that evolution has stopped [5], mainly due to either high consumption of energy or the inability to dissipate the generated heat. This has led to an increase of the interest on employing multiprocessing environments.

The use of multiprocessing environments introduces the concept of multithreading [10], in which two or more threads can coexist simultaneously. However, this does not mean they are necessarily executed in parallel, as they can also be executed concurrently. *Parallelism* refers to existence of two or more threads running simultaneously on different cores, while *concurrency* is used to denote the context where two or more threads are active in a same time interval, but not necessarily executed at the same time.

Concurrency occurs in both single-core and multi-core environments. This is possible because modern operating systems support multithreading by switching context of existing threads, that is, they offer mechanisms to execute threads alternately, even if only one of them at a time can be processed by systems with a single CPU. Depending on how the context switch is made, we will be in the presence of a *preemptive* or *cooperative* multithreading.

One of the biggest advantages of multithreading relies on the ability to maintain the execution of more than one thread at a same time interval, without the need to wait until execution of one of them is completed to proceed with the execution of some another thread. This allows us to implement applications in which certain processes are executed in background without the risk of getting blocked due to the call to blocking functions in another thread (e.g. a process interacting with an user through I/O operations). Multithreading allows an optimal use of CPU processing capacity, because a blocked thread would not cause CPU to remain idle, having other threads waiting to be executed.

Many approaches, models and languages [17], [18], [19], [20], [6], [14] have been proposed to provide support for concurrent programming. How-

ever, concurrent programming still presents several challenges to the programmer, mostly related to ensuring proper interaction between threads sharing resources. The Luaproc library[21], which provides support for concurrency in Lua, has shown some promise in terms of performance and ease of use. It adopts messages exchange as the only means for communication between execution flows, so mechanisms synchronizing accesses on shared memory are not required.

The goal of our work is the study of the Luaproc library in order to provide alternative solutions to the limitations that the library still presents and incorporate to it mechanisms for exploring parallelism in scientific computing, thus extending its use to new scenarios. Our work is structured as follows: Chapter 2 presents several related works, Chapter 3 discusses the extensions incorporated to the Luaproc library, Chapter 4 presents the experiments and their corresponding results, and Chapter 5 presents the conclusion and future works.

2
Related Work

In this chapter, we first present a brief description of support for concurrency in some scripting languages. The main features of the concurrency models they adopt and some of their corresponding disadvantages are presented. We then focus on the exploration of concurrency in Lua, by introducing some libraries created with this purpose. For each of them, both the implemented concurrency model and its main characteristics are presented.

2.1
Concurrency in scripting languages

Scripting languages have become widely used in the implementation of applications with different purposes and several models have been designed in order to support concurrency in them. In this section, we describe how Javascript, Perl and Python support concurrency.

Javascript

JavaScript is a dynamic language mostly used in web-oriented programming. It provides support for concurrency [8] through *web workers*, thus allowing expensive tasks to be executed concurrently by web workers in background and without blocking the UI thread of the web application. When instantiating a web worker, we are instantiating a new JS VM which creates a new kernel thread in order to run the code assigned to that web worker. Many browsers limit the number of web workers that can be created.

In the Javascript concurrency model, Web workers share no data at all, and communication among them takes place either by *message cloning* or *ownership transfer*. The first option clones all data to be exchanged between web workers, which results in a significant overhead when sending large objects. In scenarios where the exchange of large volumes of data between web workers is necessary, ownership transfer may be appropriate. In this case, a reference to the object is sent, but the source web worker looses access to this object. This allows a message to transfer references while avoiding memory references to

be shared among web workers. In the exchange of data between web workers, basic types of JavaScript, JSON objects, TypedArrays and recently ImageData types are supported.

The concurrency model adopted by Javascript, although allowing costly tasks to be executed in background, has some disadvantages. For example, the parallel execution of a set of tasks working on an shared object can not be performed by using ownership transfer, as only one web worker could access the object at a time. On the other hand, the use of message cloning for such purposes results in an additional overload when creating new copies of the object to be sent. Furthermore by using message cloning, the tasks are executed on different copies of an object, so it is necessary to create mechanisms to unify the different results obtained by each web worker into a single object.

Perl

Perl provides support for concurrency through its *thread model* [9], in which each thread runs in a separate virtual machine. By default, no data is shared. Threads may only share access to data explicitly marked as shared. Perl allows scalars, arrays and hashes to be shared.

Shared data are subject to data races, so Perl provides the *lock* function as a synchronization mechanism. This functionality places a kind of lock on a particular object, although it does not forbid other threads to access the object, unless they also request the same lock.

Perl 6 [31] introduces the `Lock` class as part of its low-level API for concurrency. The *protect* method defined in this class allows us to encapsulate code blocks, ensuring that they will be executed by only one thread at a time (as a kind of critical section). On the other hand, as part of its high-level API for concurrency, Perl 6 defines the `Channel` class, whose objects represent *thread-safe queues*. Objects sent to a channel are delivered to readers in the same order in which they arrived at it. The channels are used as high-level mechanisms for communication between threads. In fact, the Perl 6 documentation recommends using them instead of locks.

One of the disadvantages of Perl thread model is the high cost associated with creating threads, since for each new thread a new instance of the interpreter is created and the existing data in the current thread copied to the new instance.

Python

Python supports concurrency through its *threading library* [16], which supports the creation of threads. These are mapped to kernel threads in a 1:1 relation and represent a concurrent execution flow, sharing all data in the process where they are created. The threading library provides synchronization primitives (locks, semaphores, conditions, among others) for guaranteeing controlled accesses to shared data.

Although Python threads can communicate through shared memory, the language provides a library for handling thread-safe queues. These are mostly aimed at concurrent programs structured in a providers/consumers model, but can be used as a generic communication mechanism between Python threads.

One of the disadvantages of the Python concurrency model is the inability to execute threads in parallel, due to the existence of the *GIL* (Global Interpreter Lock). This is a kind of lock ensuring that only one Python thread can be executed in the interpreter, at a time.

The disadvantages identified above and the complexity of using synchronization mechanisms in preemptive multithreading models demonstrate the need to search for models for concurrent programming in dynamic languages.

Lua [4] provides support for concurrent programming by using coroutines, a mechanism for cooperative multhreading. Skyrme, Rodriguez and Ierusalimschy [21] designed a model for concurrent programming, which allows multiple execution flows to be executed concurrently, communicating only through *messages exchange*. They implemented this model as a Lua module, resulting in the *Luaproc* library [7]. Subsequently, they structured a communication model intended to safely share data in shared memory environments. The implementation of this model in Lua resulted in the *Luashare* library [22]. These models and libraries are described in next section.

2.2 Message exchange in Lua libraries

Lua is a scripting language, characterized by being simple, efficient, portable and extensible. The language was designed for integration with software written in C, so it is possible both to call Lua from C and to create modules in C in order to extend Lua. Much of Lua's power comes from its extensible nature, as many of the functionalities that enable it to be used for numerous purposes come from libraries.

Lua *states* encapsulate an execution environment. Each Lua state has

its own global variables and structures. Two or more Lua states can coexist without the risk that uncontrolled accesses to global variables occur.

When creating a C module in order to extend Lua, generally we implement a set of functions that will subsequently be called from Lua applications. These functions must be registered previously in the Lua state where the module is loaded. Moreover, these functions may have input parameters, which will be provided upon their invocation in the Lua code, and return values, which must be moved from C to Lua.

In order to provide communication between C and Lua, regardless of which way it needs to do so, each Lua state has a virtual stack that acts as a bridge in that communication. When we call a function defined in C from Lua code, its input parameters are placed in this stack and at the end of its execution, possible return values are obtained from this stack. By using a set of functions (C API) of the Lua library, it is possible to manipulate (e.g. push, insert, remove, etc.) items in this stack. In this way, functions in C modules can obtain from and place back to the stack its input parameters and return values respectively. Some libraries supporting concurrency in Lua allow creating execution flows from Lua states, and establish the message transfer as a means of communication between them. To transfer a message between Lua states, such libraries use their corresponding virtual stacks. Next, we present some examples of such libraries.

Lanes

Lanes[29] is a C module providing concurrency in Lua. It allows for executing several Lua states in parallel, thus enabling us to exploit parallelism on multicore machines. The architecture supporting Lanes is composed of the following elements: *lanes*, which are basically Lua states executing in parallel; *universes*, which are sets of lanes; and *keeper states*, which are Lua states responsible for storing the data sent through *Linda*[3] objects.

Lanes are created and executed by a call to a *generating function*, which receives the function (let's call it *func*) to be executed in parallel. The lanes execute the code of *func*, sharing their upvalues and global variables. When calling the generating function, the values passed to *func* allow each lane to follow its own execution path. The concept of universe organizes the lanes into groups and establishes limitations about how a memory space can be shared among them. Universes share no data with each other, that is, there can be data exchange only between two lanes belonging to a same universe.

Two lanes can exchange data through Linda objects. Lanes implements

a Linda object as a gateway to a tuple space, through which the lanes can send and receive tuples (messages). Furthermore, it defines a set of atomic operations on Linda objects, so that different lanes can access a same tuple space, in a controlled manner. The *func* function can access a Linda object as upvalue, parameter or as part of a message.

Luaproc

Luaproc defines a hybrid architecture that employs both kernel threads and user threads. The use of kernel threads allows the exploration of parallelism in multiprocessing environments, which would not be possible through user threads. On the other hand, the use of user threads enables the creation of a large number of concurrent execution flows which would be too costly using only kernel threads. User threads are scheduled by an user-level scheduler following an MxN model. The combined use of kernel threads and user threads by Luaproc decreases scalability restrictions when exploring multithreading.

Luaproc user threads are called *Lua processes* and communicate only through messages. The model includes functions for sending and receiving messages, which are used for communication and synchronization. Messages are exchanged through *channels*, identified by a *name*.

Lua processes execute Lua code fragments. Each Lua process corresponds to a separate Lua state. Lua processes do not share memory at all, and there is no need to worry about uncontrolled accesses to the same object from different Lua processes. This characteristic facilitates the development of concurrent applications because it avoids the need for synchronization mechanisms to control accesses to shared memory.

The Luaproc library allows the creation of an arbitrary number of Lua processes and a different number of *worker* threads, which are kernel threads created by using *POSIX Thread* library [11]. The Luaproc scheduler distributes the pending Lua processes among worker threads. Worker threads draw Lua processes, one at a time, from a ready queue in order to start or resume their execution.

Early versions of Luaproc supported only strings as message data. Any other value had to be serialized to be exchanged. The current version of the Luaproc library extends message transfer to booleans and numbers without previously converting them to strings. The authors argue that the remaining restrictions can be overcome by sending Lua code in the form of messages. A Lua process can send a string containing a function with arbitrary return values.

Message sending is synchronous, that is, the call to the *send* function only returns when a Lua process is ready to receive this message. So, when sending a message, a Lua process remains blocked until the message is delivered.

Receiving a message is also a synchronous operation. However, you can also perform an asynchronous receipt, in which the call to the *receive* function returns control to the Lua code immediately. In this case, a message is received from the specified channel if there is one pending. Otherwise, the `nil` value and a specific status indication are received.

Luaproc implements sending/receiving of messages basically through the copy of values from the stack of the Lua state associated with the sender Lua process to the stack of the Lua state associated with the Lua process that receives them.

Luashare

Luashare is another library for concurrent programming in Lua. The model implemented by this library enables safe sharing of data in shared memory environments. This model results from making several changes in the Luaproc model. It was implemented through changes to the Luaproc library but also required making changes to the Lua interpreter. The Luashare library is composed by a Lua module and another one written in C, plus a modified Lua interpreter.

The model implemented by Luashare is the result of the combination of three concurrency patterns [9]. The first pattern taken into account was *no-default sharing*. In this pattern all data are by default local to an execution flow, and the only way to share them is through *shared objects*. The second pattern considered was *data ownership*, which allows only one execution flow at a time to have read/write permission on a shared object. Finally, Luashare offers shared access to read-only, *immutable*, data.

Luashare uses *capabilities* to enforce the concept of read-only and read-write shareable objects. Capabilities represent the permissions (*read-write* or *read-only*) through which a shared object can be shared. Shared objects subject to future modifications must be shared in read-write mode, while those ones that are to be only read by multiple execution flows must be shared as read-only.

Sharing a shared object as read-write implies the loss of access by the execution flow that sends it; at the same time, the execution flow that receives it gains a read-write permission on this shared object. On the other hand, an execution flow sharing a shared object as read-only loses write access to

this, and the receiver execution flow obtains a read-only access to this shared object. Shared objects shared as read-only become immutable objects without the possibility of being shared as read-write again.

Similar to Lua processes in Luaproc, *Lua threads* in Luashare represent parallel execution flows of Lua code fragments. However, unlike Lua processes, which are created from different Lua states, Lua threads are created from co-routines within the same Lua state. This feature enables Lua threads to have access to the global state of the Lua state through which they share data.

Luashare provides functions for sending (sharing) and receiving shared objects through channels, as Luaproc does. As in Luaproc, the functions intended to send/receive shared objects are blocking. We can also call the *receive* function asynchronously.

A verification is performed when sharing a shared object, including both the types associated to the encapsulated values and the proper correspondence between the capabilities of each of these values and the capability specified in the call to the sending function.

In Luaproc, Lua processes are associated to different Lua states, so a Lua state can be accessed by only one worker thread at a time. However, in Luashare, Lua threads are created in the same Lua state, thus sharing its global state. This implies that the worker threads executing them have access to this global state simultaneously, which might introduce data races on the data stored in it. Modifications had to be made to the Lua interpreter, in order to ensure controlled access to these data.

Comparison between Luaproc and Luashare

A previous work [22]established comparisons between both libraries. In terms of time spent for transferring a data set between execution flows, Luaproc showed, in most cases, a better performance than Luashare. This is mainly due to the cost associated with the creation and transfer of shared objects. However, with respect to required memory resources, Luashare showed better results than Luaproc, since only references to immutable or shared objects are transferred between execution flows. In contrast, Luaproc must perform a copy of each sent message, making use, therefore, of a greater amount of memory resources.

We performed a test in which we implemented a parallel solution to the knapsack problem (see Appendix A), employing both the Luaproc and Luashare library. The purpose of the test was to determine which of the two libraries provided better performance. The *fragments* defined in that solu-

tion are actually tables, which are transferred between execution flows. Because Luaproc does not support the transfer of tables between Lua processes, the implementation using Luaproc employs the Luabins library for serialization/deserialization.

As a measure of performance, we used the time it takes an implementation to compute the solution of the problem. We executed each implementation a total of 4 times with a specific number of threads (kernel threads created by using the *POSIX Thread* library) and workers (processes or Lua threads employed for computing the solution), taking as an execution time (T) of the implementation the average of times resulting from the runs. We also computed the standard deviation (SD) associated to these times in order to know how irregular they are. The instance used in this test consists of: a knapsack with $12x10^3$ units of capacity; and $12x10^2$ objects, each of them with an associated weight of 20 units. Objects with odd and even identifiers have an associated value of 50 and 2 units, respectively. All the runs were executed on a desktop computer with the following features: Intel(R) Core(TM) processor (i5-4210U, 1.70 GHz, 2 Cores), 8 GB (RAM) and a 64-bits architecture. Table 2.1 shows the execution times (in seconds) obtained by both implementations.

Threads	**Workers**	**Luashare**		**Luaproc**	
		T(s)	SD	T(s)	SD
2	2	60.24	0.83	15.38	0.16
	4	94.29	0.17	18.10	0.45
	8	163.52	0.90	23.92	0.22
4	2	60.63	0.60	15.42	0.35
	4	90.94	0.85	19.46	0.11
	8	163.62	0.20	24.37	0.16
8	2	60.72	0.79	15.58	0.26
	4	91.15	0.67	18.44	0.17
	8	158.39	0.65	24.59	0.33

Table 2.1: Comparing the Luaproc and Luashare libraries.

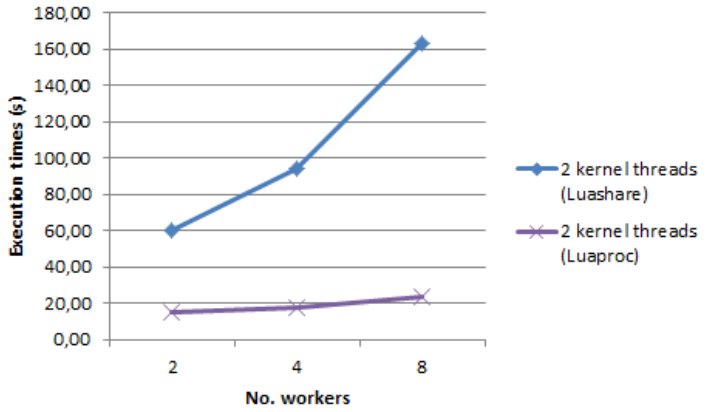

Figure 2.1: Comparing the Luaproc and Luashare libraries.

The best performance achieved by each library is shown in Figure 2.1. This confirmed the pattern we have observed in previous comparisons between the two libraries: Luaproc employs a lesser time for transferring data between execution flows, regarding Luashare. Therefore, we focused on making the use of Luaproc more suitable to new scenarios by incorporating new functionalities.

3
Extensions to Luaproc

In this chapter, we first introduce the motivations to our extensions to Luaproc, discussing alternative ways of dealing with the existing limitations. We then describe in more details how these solutions were implemented. Finally, we present a brief discussion about the benefits that the implemented solutions may offer to Luaproc.

3.1
Motivations and goals

After studying both the model and the implementation of Luaproc library, we identified a set of limitations. The first of these is that Luaproc does not support the *table*, *userdata* and *function* basic types on messages transfer, although these data types are commonly used in the implementation of Lua applications. Another limitation we found was the absence of support for either asynchronous messages sending or the implementation of collective communication operations. Applications employing simulating mechanisms for such purposes may not reach their maximum performance in certain scenarios.

Next, we present the motivations that led us to provide solutions to these limitations, as well as the goals defined in our work for such purposes. For some limitations we consider a simulating mechanism providing a solution. In that case, we discuss the main disadvantages of using that mechanism.

3.1.1
Tables

Tables are a key element in any implementation with some degree of complexity, because they represent the only data structuring mechanism in Lua. They are associative arrays of variable size, which can be indexed by values of any types excepting `nil`, and can have values of any types. Tables support the traditional operations (e.g. insertion, removal, access and updating of elements) performed on arrays. In the current Luaproc version, tables must be serialized before sending them (see Figure 3.1). It would be useful, and more efficient, if tables could be transferred between Lua processes without having to

serialize them previously. As part of our work, we incorporate support for the direct transfer of tables between Lua processes. Alves[34] implemented a basic scheme to support *table* data type on message transfer, but the communication costs were higher than expected. We used this implementation as a basis.

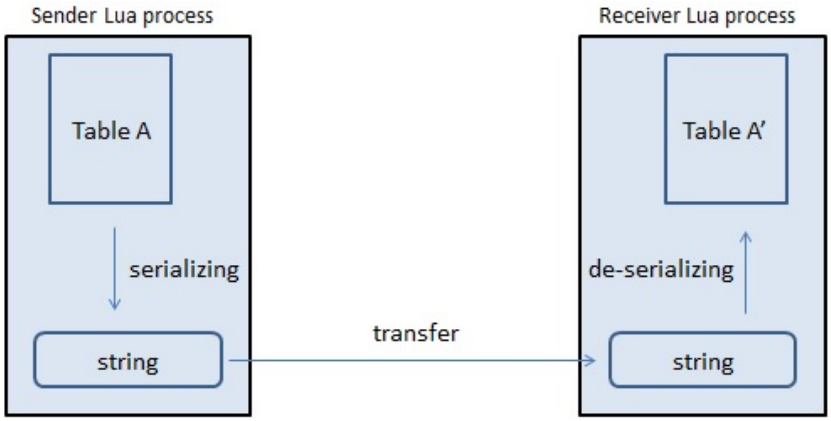

Figure 3.1: Transferring a table between Lua processes, by serializing it previously.

3.1.2 Userdata

In Lua, an userdata value represents a raw memory area, normally used for storing user-defined types in C. An userdata has no predefined operations in Lua, so operations for handling it must be implemented in C and stored in a metatable associated to it. There are several ways of creating an userdata, but usually, we use two of them, either a pointer to the structure to be manipulated or the structure itself is stored in an userdata.

Using userdata gives us some benefits. This data type can be used in the implementation of applications in which data structures store a large amount of elements and the required memory resources could affect performance. Tests have been performed in order to compare the memory resources required when representing a same collection of elements by using tables in Lua, as well as by using instances of user-defined types in C. The results show a lower demand for memory resources by the second representation in comparison with demand required by the first one[15]. Userdata is also useful when dealing with external objects such as sockets or files, allowing such objects to be stored and manipulated from Lua. Figure 3.2 shows a Luasocket's socket as an example of an userdata allowing us to manipulate a system's socket.

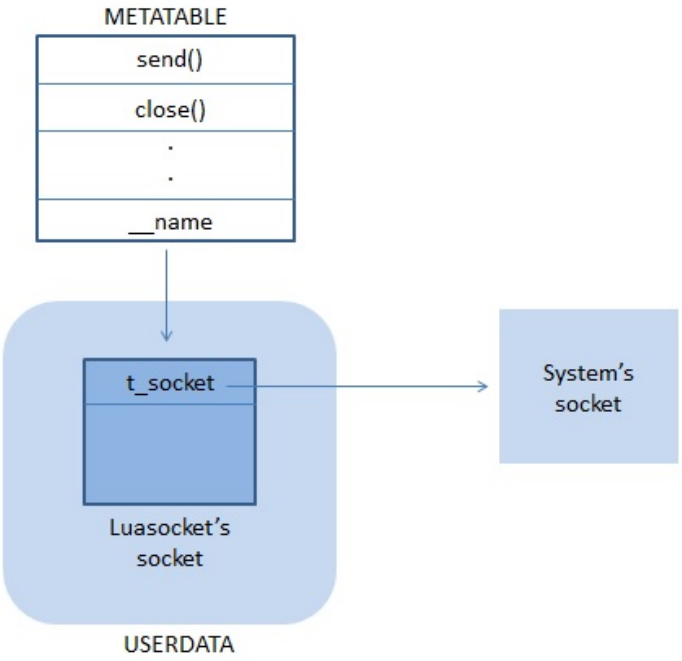

Figure 3.2: Luasocket's socket as an example of an userdata value.

A second goal of our work is thus to include support for transferring userdata between Lua processes. For this, we may copy either the pointer or structure stored in the userdata to the receiver Lua process. However, without taking appropriate measures, this would introduce shared memory in Luaproc, because two or more Lua processes would be able to access the same memory block in an uncontrolled manner. Furthermore, if one of the copies were reached by the garbage collector, the structure associated to the userdata would be finalized. This would cause the remaining copies to become invalid. Figure 3.3 illustrates the consequences from transferring userdata without certain cares. Having this in mind and, in line with the Luaproc model, we must design and implement a mechanism that transfers userdata between Lua processes but disallows shared accesses to these data. A similar behaviour to that we desire is implemented by the Luashare library and Javascript language. When transferring a shared object in read-write mode, the sender Lua thread no longer has access to this. Similarly, transferring data between web workers by using the *ownership transfer* pattern causes the source to loose access to these data.

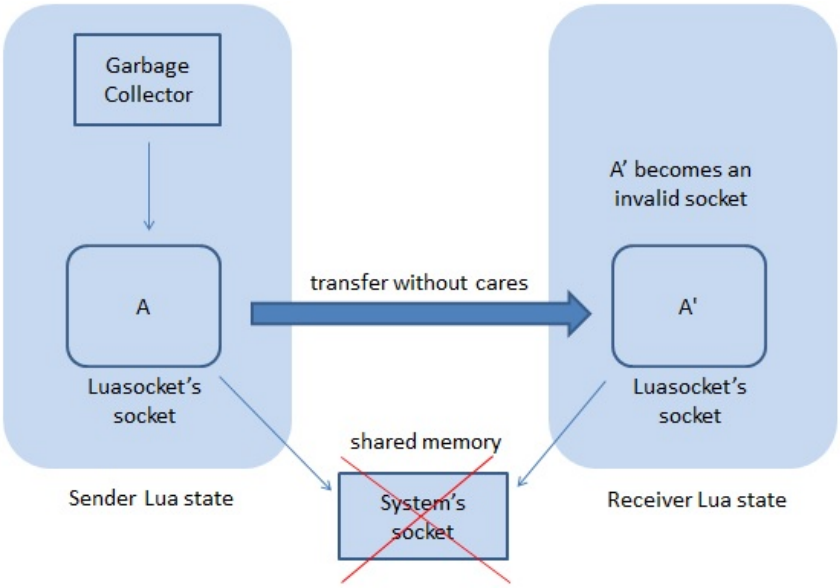

Figure 3.3: Consequences from transferring an userdata without taking the proper cares.

3.1.3 Lua functions

Lua provides *first-class* functions with lexical scoping. Lua functions are in fact closures[4] with their own environment. The environment of a function is a table storing values used by it.

Following our desire to extend the support to basic Lua types on messages transfer, our next goal is to allow the transfer of functions in Luaproc. It would be useful to extend the Lua semantics of *first-class* functions to messages transfer, allowing the transfer of functions between Lua processes.

Transferring a closure between Lua processes could be useful in implementing some applications. For example, a Lua process defining and distributing tasks processing to a set of Lua processes could encapsulate one or more tasks in a closure, and transfer them to a Lua process, in which these tasks would be executed concurrently.

3.1.4 Asynchronous message sending

There are several advantages and disadvantages of using either synchronous or asynchronous sending in concurrent and distributed applications[2]. As one of the advantages of using synchronous sending, a sender process can ensure that a sent message has been received, before resuming its execution. However, a disadvantage is the limitation imposed to the concurrency degree that the execution flows can reach. In scenarios in which parallelism is em-

ployed for high-performance computing, the use of synchronous sending is often considered a limitation to the exploitation of parallelism.

Despite the synchronous nature of messages sending in Luaproc, we can simulate asynchronous message sending to approach such scenarios (see Figure 3.4), to facilitate the exploration of parallelism using Luaproc. Let's consider a sender and receiver Lua processes that exchange messages continuously. According to the semantics of the *send* function, when sending a message, the sender Lua process will remain blocked until the receiver Lua process receives it. For each message, the sender Lua process A can create a temporary Lua process C and, send the message to C. At this point A can proceed execution and C will remain blocked waiting for the receiver Lua process B. However, A will still block until C finishes receiving the message. Furthermore, creating a temporary Lua process for each asynchronous sending increases the demand for memory resources. To avoid this, we have opted for providing an asynchronous sending in Luaproc.

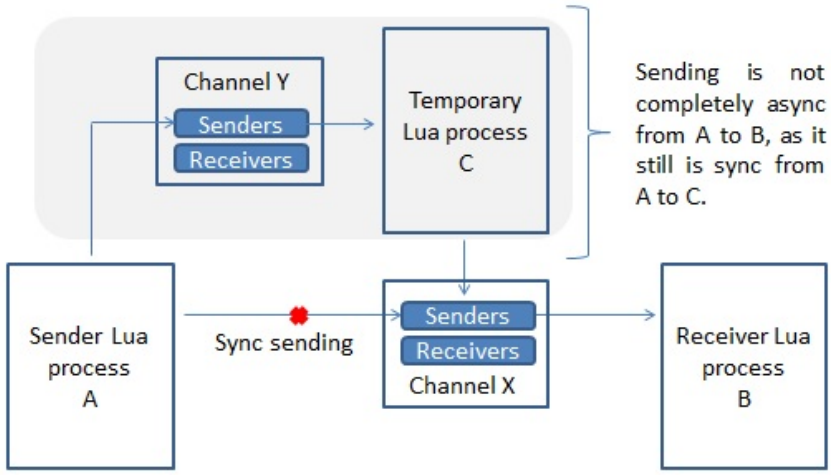

Figure 3.4: Simulating an asynchronous message sending.

3.1.5 Support for collective communication operations

Messages exchanged among processes may be either point to point, from a source to a destination process, or collective, in which two or more processes participate[12]. Collective communication operations are communication patterns often used for exchanging data and establishing synchronization between execution flows in parallel and distributed applications. These may be useful in several scenarios, such as that of parallel scientific computing. The *PVM*[26] and *MPI*[27] interfaces include a set of collective communication operations. Next, we describe some of them.

- **Broadcast**: broadcasts data from a single sender process to a set of receiver processes.
- **Reduce**: computes a single value out of the data sent from different sender processes and stores it in a single receiver process.
- **Gather**: gathers data sent by different sender processes and stores them in a single receiver process.
- **Scatter**: distributes different data from a single sender process on a set of receiver processes.
- **Barrier**: synchronizes the execution of multiple processes.

The broadcast, reduce, gather and scatter collective communication operations have in common the establishment of a point of synchronization among the participating execution flows, so that none of them can continue their execution unless the operation is completed. This characteristic is present in most collective communication operations. As an example of these operations, Figure 3.5 illustrates the functioning logic of the scatter operation. Establishing a synchronization point corresponds to the behavior defined by the barrier operation, so we may use this operation as a basis for implementing collective communication operations.

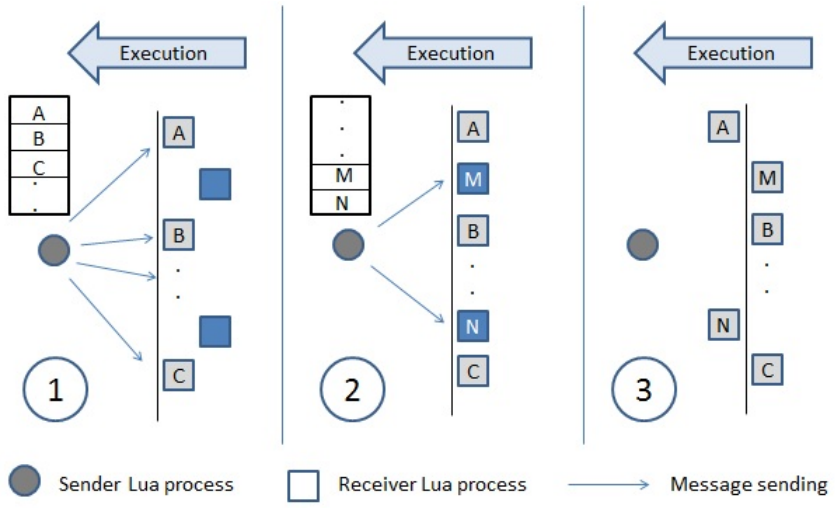

Figure 3.5: Basic steps of a scatter operation.

Luaproc supports point-to-point communication operations through the *send/receive* routines. It would be useful for Luaproc to offer support for the implementation of collective communication operations. This would allow extending the set of communication operations that can be executed on Lua processes and, consequently, its usage scenarios. To offer support for the implementation of collective communication operations, a key task is to

incorporate the barrier operation to Luaproc. In addition to serving as a basis for the implementation of collective communication operations between Lua processes, the use of the barrier operation as part of Luaproc does not restrict the implementation of these operations to a specific set.

Without extending Luaproc, we could simulate a barrier by using synchronous channels and a coordinator Lua process (see Figure 3.6). A Lua process involved in a barrier operation uses a first channel for sending a notification to the coordinator, indicating that it is participating in the operation, it then remains waiting for the coordinator to send a notification through a second channel. When the coordinator receives notifications from all the Lua processes involved in the operation, it sends as many notifications as it has previously received to these Lua processes, through the channel where they are waiting.

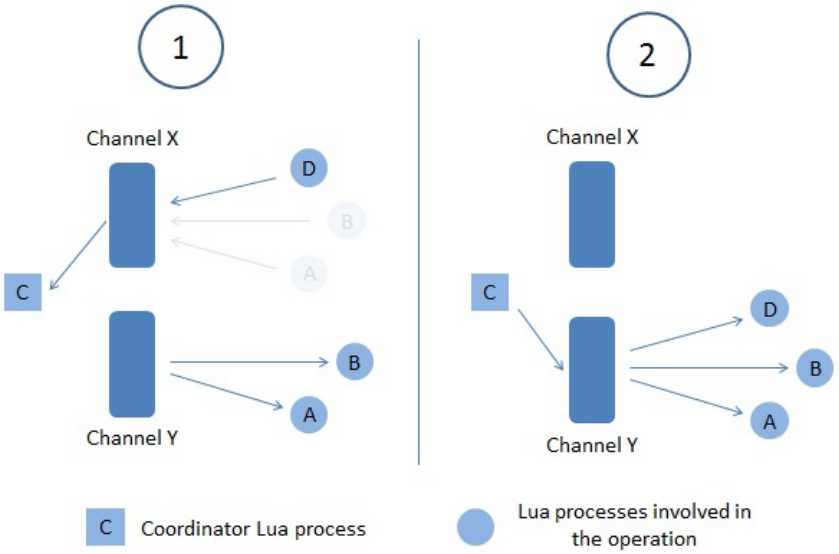

Figure 3.6: Simulating a barrier operation.

This mechanism includes several messages exchanges between the involved Lua processes, potentially increasing resource usage and execution time. Moreover, the channels used when simulating a barrier operation should not be used for sending messages through *send/receive* communication routine, as sending and receptions from different communication operations could match and lead to an undesired behavior of the application. This would imply the creation of a large number of channels, if the application includes Lua processes involved in both communication routines.

Therefore, we opted for implementing a new *barrier* function. This is called by all the Lua processes involved in the operation, and accepts the following parameters: *chn*, the channel through which the operation will be executed; and *num*, the number of involved Lua processes. As a result, *barrier*

blocks the execution of the Lua process executing it, returning after *num* Lua processes have executed this function on *chn* channel. Unlike the simulating mechanism, this function does not require messages exchanges between the Lua processes involved in the operation, and allows using both the *send/receive* communication routine and barrier collective operation on a same channel.

3.2 Implementing the extensions

In this section, we present requirements, characteristics of the implementation, and limitations associated to each functionality incorporated to Luaproc. When possible, we present mechanisms implemented by other libraries or languages for similar purposes. When applicable, we discuss the differences between a simulating mechanism and that we implemented. Furthermore, we mention modifications on the API of the library, resulting from implementing these functionalities.

3.2.1 Tables

Currently, we must employ a mechanism to serialze Lua values for transferring tables between Lua processes. Luabins[30] is one of the libraries developed for such purpose. When transferring a table *t* by using Luabins, we must first call the *save* function offered by this library. This function first allocates a memory block to store the string resulting from serializing *t*. It then builds that string by copying each key-value pair in *t* to the allocated memory block. After transferring this string between Lua processes, we must employ Luabins's *load* function for performing the deserialization process in the receiver Lua process. This function creates a table *t'* equivalent to *t*, by travelling the transferred string and inserting into *t'* the key-value pairs previously stored in that string. As mentioned in Section 3.1.1, the serialization process may affect the time it takes for transferring a table. Therefore, we implemented a mechanism for transferring tables directly between Lua processes, which presents a similar behaviour to that recently described but avoids the use of serialization/deserialization processes.

Our mechanism traverses a table *t*, and at the same time, builds an equivalent one in the receiver Lua process by copying to it each key-value pair in *t* (see Figure 3.7). This mechanism includes both a recursive traversal of the table to be sent, if at least one of its values is a table in turn, and detection of values for which messages transfer is not supported.

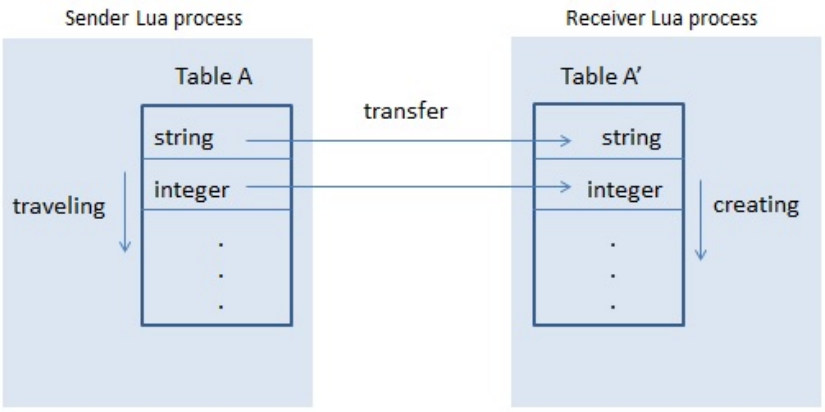

Figure 3.7: Direct transfer of tables between Luaproc processes.

The tables to be transferred are subject to certain limitations on their composition. Both table keys and values must be supported basic types on messages transfer and the table itself must be acyclic. Furthermore, our mechanism limits the number of nesting levels that a table may have: tables going further that limit will not be transferred. However, restrictions on supported types in the serialization process, cyclic character of tables and number of allowed nesting levels also apply among existing mechanisms for serialization/deserialization of Lua values. Therefore, the limitations do not represent a loss in flexibility.

3.2.2 Userdata

Some Lua libraries supporting concurrency have implemented mechanisms to transfer userdata between execution flows. Before describing the mechanism we implemented, let's see briefly how the Lanes and *Leda*[24] libraries transfer userdata between Lua states.

The concept of *deep userdata* is introduced by Lanes in order to allow multiple lanes to access a same Linda object. The deep userdata are a kind of proxy or wrapper userdata for a memory block shared among several Lua states. This memory block stores the structure of an object (e.g. Linda objects in Lanes) to be shared and must have no links with Lua states. The Lanes mechanism implementing this concept shares an object among Lua states by creating an userdata (the deep userdata) in the receiver Lua state, which points to its memory block. As a result, this object can be accessed from several Lua states.

This mechanism requires the programmer to provide an identity function (let's call *idfunc*) for each object to be shared, whose main tasks are creating and finalizing the memory block storing the object. Lanes uses this function for

creating an object to be shared, and then creates the deep userdata through which said object can be accessed. This allows using the mechanism for sharing userdata among Lua states in both Lanes itself and other applications. In that case, the *idfunc* function associated to an userdata is responsible for creating the memory block to be shared and storing in it the structure of this userdata.

This mechanism ensures a shared userdata to be finalized just in case the garbage collector reaches the last deep userdata pointing to it. However, Lanes guarantees synchronization on accesses to a shared userdata by a different mechanism to the one implementing the deep userdata concept. Therefore, using this mechanism outside Lanes requires the programmer to guarantee a controlled access on the shared userdata.

Leda is a library intended to add support for multi-threaded processing based on *SEDA*[33] principles to the Lua programming language. It provides Lua interfaces to explicitly define the event workflow of an application. Each workflow's step is defined by a *Stage*, which basically is a Lua state, and is implemented by an event handler. Each event is tied to a stage through generic interfaces called *Connectors*. Stages implement part of the application logic and decide which are the next events fired from there. Thus, workflow is defined as a stage graph which are bonded together through connectors.

This library allows userdata to be transferred between stages through triggered events. For this, Leda requires the programmer to provide a *wrapper* and a *re-builder* function for an userdata to be transferred. The first one is responsible for both disabling access to the userdata from the sender stage and returning the re-builder function. When executed, this last function must allow access to the userdata previously wrapped from the receiver stage. The wrapper function must be stored in the metatable of the userdata to be transferred. Thus, Leda is able to execute this function in the sender stage and, in representation of the userdata, place the returned re-builder function in the event to be triggered. Once the event is received, Lanes proceeds to execute the sent re-builder function. Unlike Lanes, Leda allows only one Lua state to access a transferred userdata.

When transferring some kinds of userdata between Lua processes, it is necessary to know their structures, because these require some updates once transferred. However, the structure of an userdata may vary from one to another, thus impeding to design a mechanism for transferring any userdata without having information about its structure. That is the main reason why both Lanes and Leda establish some requirements on the userdata to be supported on the message transfer. Lanes requires an *identity* function that is able to handle the structure of the userdata to be shared. In addition, it creates

said userdata only once and, sharing it turns into transferring a pointer to the userdata between the involved Lua states. On the other hand, Leda requires two functions, which must also be able to interact with the structure of the userdata to be transferred.

Having this in mind, the mechanism we implemented for userdata transfer optionally allows the programmer to define new transfer functions. For each type of userdata to be supported on the messages transfer, both a sender and receiver transfer function may be provided. These functions are responsible for transferring an userdata to and from a Lua process, taking into account a set of requirements we will address later. If no transfer functions are provided for an userdata, the mechanism tries to transfer it in a predefined way. The implemented mechanism ensures that a transferred userdata is accessed by only one Lua state, in compliance with the characteristic of no-sharing adopted by Luaproc. Regarding to requirements and guarantees, our mechanism shows certain similarity only with that implemented by Leda, as Lanes basically uses the *identity* function to create/finalize the userdata to be shared, and allows this userdata to be accessed by several Lua states.

Message exchange in Luaproc may occur in two different scenarios: a sender Lua process sends the message to a receiver Lua process r or a receiver Lua process receives a message from a sender Lua process s. Both r and s are Lua processes blocked in a channel, while waiting for matching Lua processes. When transferring an userdata with associated transfer functions, our mechanism uses either the sender or receiver one, depending on which of these scenarios the transfer is carried out. In that case, we must previously register these functions in the Lua process, by using the *regudata* function.

This function accepts, as a parameter, a table storing a *transf_funcs* function. When invoked, *transf_funcs* must return a table whose indexes are names of userdata metatables, and its values, tables storing the corresponding transfer functions. In each Lua process, *regudata* registers the types of those userdata which can be transferred by that Lua process and their transfer functions in a *transferable_udata* table (see Figure 3.8). The *regudata* function executes *transf_funcs* and checks whether any of the entries in the returned table have previously been registered in the *transferable_udata* table. If so, *regudata* registers none of them and returns a notification. Otherwise, it adds each of these entries to the *transferable_udata* table.

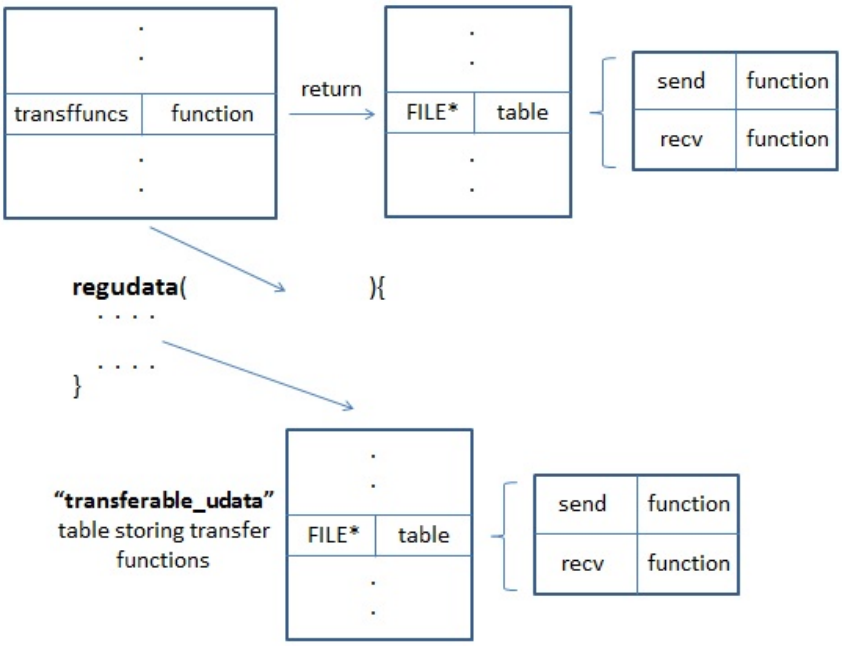

Figure 3.8: Characteristics of the *regudata* function.

The transfer functions must be C functions following a behaviour indicated by our mechanism. When executed, a sender transfer function receives, as parameters: a *transfer_type* integer indicating a behaviour to follow, the userdata to be transferred and the receiver Lua process. The receiver transfer function receives, as parameters: a transfer_type integer; the sender Lua process; and the index, within the stack of that Lua process, in which the userdata to be received is stored (for more details about these functions and their parameters, see Appendix B). Both the sender and receiver transfer functions must first transfer the userdata to the receiver Lua process, resulting in an equivalent userdata with no links to the sender Lua process. They must then associate the corresponding metatable to the resulting userdata, therefore the library from which the transferred userdata is created must be loaded in the receiver Lua process. If the transfer is successfully completed, leaving the resulting userdata in the receiver Lua process should be the only modification made by these functions on the Lua processes participating in the transfer. Otherwise, both the sender and receiver transfer function must leave a `nil` value plus an error message, in each of these Lua processes. Figure 3.9 illustrates a behaviour these transfer functions could follow.

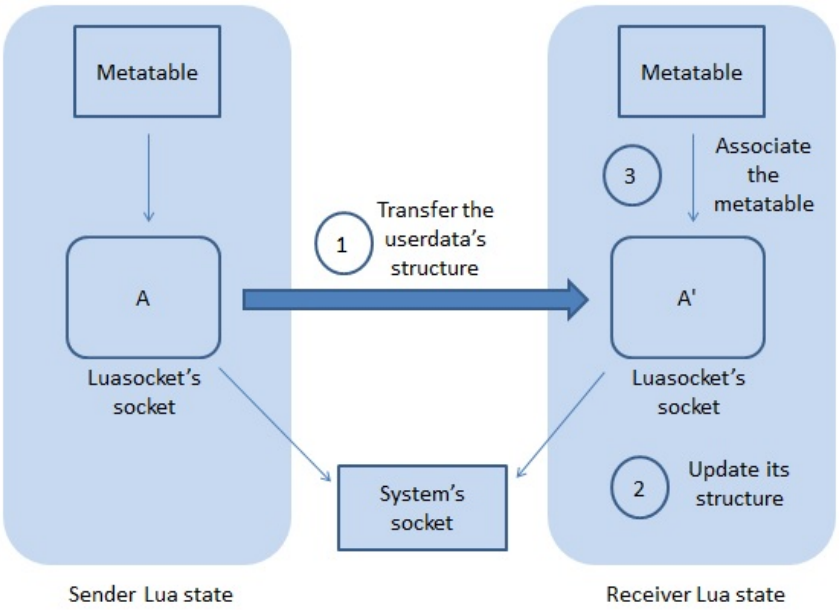

Figure 3.9: An alternative behaviour for userdata transfer functions.

For those userdata with no associated transfer functions, the mechanism performs the above routine by default. However, it is not able to perform updates on the structure of the resulting userdata. Therefore, not to provide transfer functions for userdata with these needs results in an impediment for transferring them successfully, as the structures of the resulting userdata become inconsistent.

When a Lua process sends or receives an userdata, our mechanism checks whether the type of the userdata is registered in that Lua process [1]. If so, it gets either the sender or receiver transfer function from the *transferable_udata* table. It then executes this function and, if no errors occur during the transfer process, associates a blocking metatable as the new metatable of the transferred userdata in the sender Lua process (see Figure 3.10). This blocking metatable inhibits further use of the object, returning an error: when a call to a method on the userdata occurs. The blocking metatable prevents the userdata structure both to be finalized when the garbage collector reaches the userdata and to be accessed by two or more Lua processes. Our mechanism returns a `nil` value plus an error message, if during the transfer process an error occurs.

[1] In versions prior to Lua 5.3, the implemented mechanism requires the metatable of an userdata to store its name in a *_name* field. This allows our mechanism to check whether the type of the userdata is registered in the *transferable_udata* table of a Lua process.

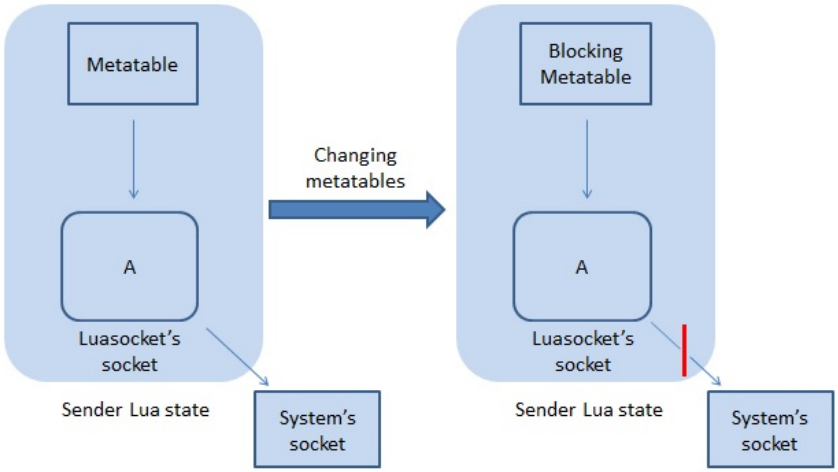

Figure 3.10: Restricting access to a transferred userdata.

The support for asynchronous messages sending in Luaproc, which we will address later, incorporates two new scenarios to the messages exchange. In each of them, both the mechanism and transfer functions must adopt a different behavior to those described above. Depending on how a message must be transferred, our mechanism adopts a behavior and indicates to a transfer function the behavior it must take, through the *transfer_type* parameter. We will address these scenarios in more detail in Section 3.2.4.

To evaluate the proper functioning of our mechanism, we implemented transfer functions for files, and sockets (see Appendix B) created by the Luasocket library[32]. We incorporated these functions to the Luaproc library, so that transferring these userdata does not require the programmers to provide transfer functions.

3.2.3 Lua functions

Before describing the mechanism we implemented for transferring Lua functions, we present a brief description of the implementation of the same mechanism in Lanes, as both of them have several characteristics in common.

When transferring a *funcLua* Lua function between two lanes, Lanes generates its corresponding binary code and loads it into the receiver lane, thus resulting in an equivalent *funcLua'* Lua function. Then, it gets the funcLua's upvalues, transfers them to the receiver lane, and finally establishes them as upvalues of funcLua'. Figure 3.11 shows the two main steps of this mechanism. The upvalues can take as values: tables, deep userdata, functions, booleans, string, integers and nil. When transferring an upvalue, Lanes checks whether its value matches the global environment of the sender lane. If so, the result

of the transfer is the global environment of the receiver lane; otherwise, Lanes transfers this value by using its transfer mechanism.

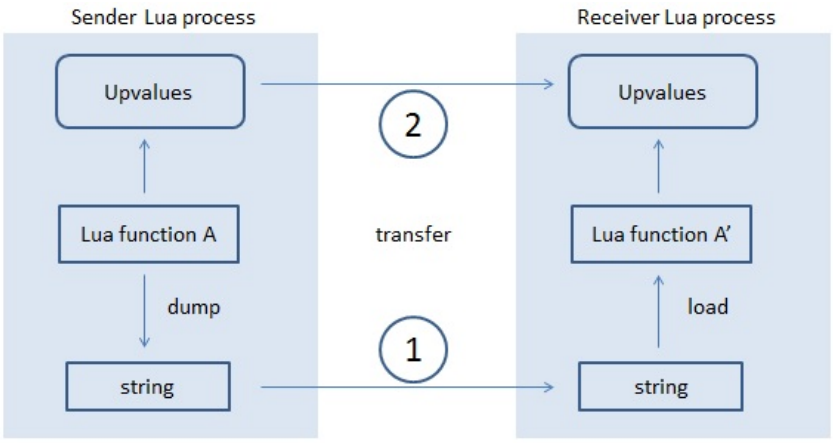

Figure 3.11: Transferring Lua functions between Lua processes.

In order to allow the transfer of C functions as upvalues, Lanes implements the following mechanism. As mentioned in Section 2.2, creating and executing a lane requires the execution of a generating function. This function accepts, as optional parameters, two lists storing the names of the standard Lua libraries and C modules to be loaded. Lanes uses a customized *require* function for such purposes, which after loading a C module by employing the Lua *require* function scans all the functions that this module registers, and stores them in the *path_func* and *func_path* tables. Each function in *path_func* has, as a key, a string (let's call it *path*) composed of the set of keys by which the function can be accessed (e.g."io.read" is associated to the *read* function of the *io* Lua library). The *func_path* table is the result of swapping roles between keys and values in the *path_func* table. When a C function is transferred between lanes, Lanes first searches for its associated string in the *func_path* table of the sender lane. It then uses this string for retrieving the equivalent function from the *path_func* table of the receiver lane. Lanes transfers C functions between lanes, as long as the sequences of keys providing access to them are similar in both lanes. Moreover, Lanes will not allow those C functions that have not been registered when creating a lane to be transferred (e.g. functions belonging to C modules loaded by the body of a lane).

Each lane has a cache table storing all the Lua functions it has received so far. Lanes uses this table for preventing transfers on already transferred functions and allowing the transfer of a Lua function with upvalues referencing it, recursively. Despite these benefits, the use of the cache table has the following disadvantage: when transferring again the funcLua function, Lanes will not be able to preserve the equivalence between the upvalues of the funcLua

and funcLua' functions. This may cause the result of a transfer not to be an equivalent function to the one that was transferred.

The mechanism we propose for supporting the transfer of Lua functions between Lua processes presents some difference when compared to that implemented by Lanes. With respect to the transfer of upvalues storing C functions, our mechanism locates and stores only those functions to be transferred, instead of storing all the C functions registered in a Lua state. Therefore, there is no need to implement a behavior similar to the customized *require* of Lanes. As a consequence, our mechanism must employ the *package.loaded* tables of the involved Lua states for transferring a C function for the first time. This implies that the mechanism takes more time for performing such a transfer in comparison to Lanes. However, it does not invest time to store those C functions that will not be transferred. Unlike Lanes, our mechanism is able to transfer all the registered C functions regardless of when their corresponding libraries have been loaded.

When loading a C module, usually we get a table storing the functions it exports. To transfer upvalues storing such tables, our mechanism does not perform a function-by-function transfer. In that case, it first searches for the equivalent table in the *package.loaded* table of the receiver Lua process, by using the name employed by the C module for registering that table in the sender Lua process. Next, it places the equivalent table onto the stack of the receiver Lua process, as a result of the transfer. Therefore, our mechanism requires that the names employed by the C module for registering its corresponding table in the Lua processes involved in a transfer match.

Similarly to Lanes, we employ a cache table for transferring a Lua function with upvalues referencing it, recursively. However, we create this table for each transfer. Although this implies our mechanism transfers a function over and over again, regardless of it having already been transferred, we guarantee with this, an equivalence between the function to be transferred and that created in the receiver Lua process.

The Lua functions that our mechanism transfers may have upvalues storing values of type string, nil, boolean, integer, userdata or function; and tables, as long as they store transferable values. Upvalues storing the global environment are not transferred. In that case, the global environment of the receiver Lua state is placed onto its stack, as a result of the transfer. Therefore, both environments must define the global variables used by the transferred functions, in a similar way. Moreover, the alternative employed by our mechanism for transferring C functions requires the libraries associated to them to be loaded in both sender and receiver Lua process.

3.2.4
Asynchronous message sending

Many approaches can be considered when designing a mechanism for transferring messages asynchronously, as this is a functionality supported by several languages and libraries providing support for concurrency. Let's see how some of them implement the asynchronous message sending between their concurrent execution flows. We then describe our implementation.

Perl6 adopts messages passing as model of communication between the different threads of a process. As we mentioned in Section 2.1, Perl6 defines the `Channel` class, whose objects are thread-safe queues providing support mostly to the provider/consumer programming style. The call to the *send* method on a *ch* Channel's instance causes the message passed as a parameter to be placed in the queue represented by *ch*. The execution of this method does not block the thread executing it regardless of whether the sent message is consumed or not. When the *receive* method is invoked on *ch*, it extracts and removes the message at the front of the queue represented by *ch*. If there are no messages in ch, *receive* will block the current thread until a message is queued in this channel. Although the receipt of messages in Perl6 is blocking, the use of the channels allows sending messages asynchronously between different threads of a process.

The Lanes library, mentioned in Section 2.2, allows exchanging messages between lanes through the Linda objects. The messages exchange mechanism implemented by this library employs Lua states (defined as keeper states) for storing the tuple spaces associated to the Linda objects. When creating a Linda object A, the *group* identifier of a keeper state can be optionally passed as a parameter. In this way, Lanes creates a t table representing the tuple space associated to A, in the keeper state corresponding to *group*. Each entry in t is indexed by a specific key and has a queue storing all the tuples associated to that key, as a value.

The *send* method of a Linda object accepts, as parameters, the tuple of values to be sent and the k key associated to it. This method places a copy of the tuple passed as a parameter in the queue associated to k, within the table corresponding to the Linda object on which it is invoked. Moreover, the tuples stored under a v key within the table corresponding to a B Linda object can be retrieved by calling the *receive* method on B, with v as a parameter. Copying the values composing a sent tuple to a keeper state allows the asynchronous message sending between lanes.

As we mentioned in Section 2.2, Luaproc implements the synchronous message sending between Lua processes by using channels. These hold two

queues, in which those Lua processes blocked while sending or receiving messages through a channel wait for a matching Lua process to arrive.

The support for asynchronous sending requires buffering those sent messages that have not yet been received. So, we created a new kind of channel. When creating a new channel, the programmer must now specify whether it supports synchronous or asynchronous messages. We now denominate a channel as either synchronous or asynchronous, depending on how messages are transferred through it.

Asynchronous channels must provide support for storing pending messages. These channels are composed of a Lua state (the *container Lua state*), storing messages in transit, and a queue (the *receivers queue*), holding those receiver Lua processes waiting for asynchronous sending (see Figure 3.12). An asynchronous message will be stored in the container Lua state as a table, so that each of the values of the message is stored in an entry. Messages in the container Lua state will be delivered to the receiver Lua processes in the same order in which they were stored. Because they are used for storing messages temporarily, container Lua states have some similarity with the keeper states in the Lanes library and the thread-safe queues in Perl6.

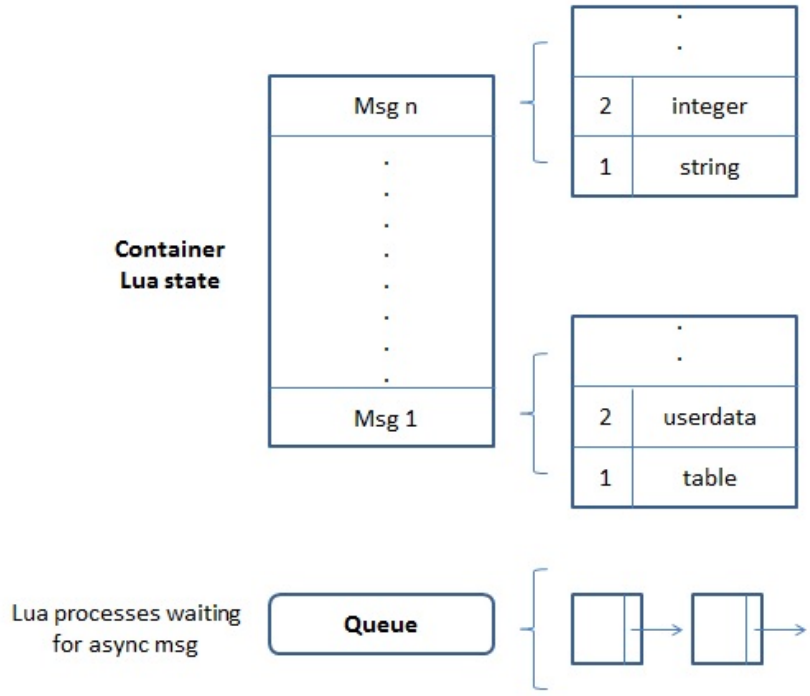

Figure 3.12: Structure of an asynchronous channel.

The support for asynchronous sending also requires making certain modifications in both API and implementation of some functionalities of the Luaproc library. We incorporate a mechanism for creating asynchronous

channels to the *newchannel* function. This now accepts as an optional second parameter, a boolean value indicating the kind of channel to be created. We also added a mechanism for messages transfer through asynchronous channels to *send* and *receive* functions. When invoked on an asynchronous channel, *send* verifies the existence of at least one Lua process in the *receivers queue*. If so, it draws that Lua process at the beginning of this queue and copies to it the sent message. Otherwise, *send* stores the message in the container Lua state. On the other hand, when the *receive* function is called on an asynchronous channel, it checks whether the container Lua state stores at least one message. If so, *receive* draws it and copies the corresponding values to the receiver Lua process. Otherwise, *receive* places the Lua process executing it in the *receivers queue* and releases the corresponding worker thread. Similarly to what is done in *newchannel*, we incorporate a mechanism for destroying asynchronous channels to the *delchannel* function.

The use of container Lua states for storing messages sent asynchronously implies taking some care when transferring userdata and certain Lua functions. The mechanisms implemented for transferring these values require the libraries associated to these values to be loaded in the receiver Lua process. Although this requirement can be guaranteed by the programmer, messages sent asynchronously may be first stored in the container Lua states, in which these libraries will not be loaded. Therefore, transfer mechanisms must adopt a different behavior when transferring userdata or Lua functions asynchronously.

As we mentioned in Section 3.2.2, the support for asynchronous message sending introduces two new scenarios to the messages exchange in Luaproc, so that a message may be sent from a Lua process to either another Lua process or a container Lua state, or received by a Lua process from either another Lua process or a container Lua state. Because the behaviours of the userdata and Lua functions transfer mechanisms, both described in Sections 3.2.2 and 3.2.3 respectively, correspond to the transfer of messages between Lua processes, we focus on their behaviours when transferring a value from or to a container Lua state.

Due to the impossibility of associating the corresponding metatable to an userdata transferred to a container Lua state, a sender transfer function must associate to it a metatable, whose name equals the name associated to the userdata's metatable. When receiving an userdata from a container Lua state, its metatable will allow our transfer mechanism to check whether the type of the userdata is registered in the receiver Lua process, and if so, obtain the corresponding receiver transfer function. Even if an error occurs while transferring an userdata from and to a container Lua state, both the sender

an receiver transfer function should leave no error message in it. In versions prior to Lua 5.3 the metatables do not store their names. So, in those versions, sender transfer functions must associate to an userdata in a container Lua state, a metatable storing its name.

Regarding the transfer of Lua functions, as a result from transferring an upvalue storing a table registered by a C module to a container Lua state, the transfer mechanism creates a specific table, in this Lua state, indicating that this upvalue must store a table registered by a C module once transferred to a Lua process. This table stores the name used by the C module for registering the table to be transferred in the sender Lua process. The transfer mechanism uses this name to search for the equivalent table in the *package.loaded* table of a Lua process receiving this upvalue from a container Lua state. To transfer upvalues storing C functions through container Lua states, we proceed in a similar way.

Because in the container Lua states no libraries are loaded, it is not possible to assign to the userdata stored in them their corresponding metatables. Therefore, if the garbage collector reached these userdata, the structures associated to them would not be finalized. This required us to make changes in the implementation of Luaproc, in order to ensure that both an application does not close successfully unless all the messages sent asynchronously have been received and asynchronous channels cannot be destroyed if its container Lua state still stores at least one message.

3.2.5 Support for collective communication operations

Implementing the *barrier* function requires certain modifications on the structure defining a channel. The channels involved in barrier operations will now able to hold an additional queue (let's call it *barrier queue*), storing Lua processes that perform such operations. When executed on a *chn* channel, *barrier* verifies whether the *barrier queue* of *chn* stores $num - 1$ Lua processes. If so, it puts them all in the queue of Lua processes that are ready for execution; otherwise, *barrier* enqueues the Lua process executing it in this *barrier queue* and releases the corresponding worker thread.

After knowing how the *barrier* function was implemented, we may analyse in more details the differences it presents with the mechanism simulating this operation. The latter performs several calls to *send* and *receive* functions for exchanging notifications between the involved Lua processes. This could cause a larger number of context switches; that is, there is a greater possibility that a Lua process calling these functions does not match any correspond-

ing receiver or sender Lua process and proceeds to release the worker thread executing it. Resuming its execution, after the corresponding operation (receive or send) is carried out by another Lua process, implies placing it in the queue of Lua processes that are ready for execution and waiting for a worker thread to execute it. This characteristic may increase the execution time of this simulating mechanism.

On the other hand, executing the barrier operation by using the *barrier* function requires at most one context switch by each of the involved Lua processes, because these processes do not have to send notifications to a coordinator Lua process neither do they have to wait for worker threads to execute them in order to remain blocked through a second channel, while waiting for a notification to be sent by the coordinator. Therefore, the implementation of the *barrier* function decreases the time it takes synchronizing the executions of the Lua processes participating in the operation, in comparison with the simulating mechanism.

3.3 Discussion

We began this chapter by defining some limitations associated to the Luaproc library. In order to offer solutions to these limitations, we extended Luaproc with new functionalities. In this section, we will discuss the benefits provided by each of these to Luaproc. In some cases, we address requirements or limitations associated to these functionalities.

Programmers are no longer required to serialize a table before sending it. In applications employing libraries for serializing tables, using the implemented mechanism removes the dependencies associated to them, thus facilitating the portability of such applications. Another advantage offered by this mechanism is the reduction of the time it takes for transferring a table, since it is not necessary to perform serialization/deserialization on it.

By using either a serializer mechanism or the direct transfer of tables, we must not employ tables as keys of tables to be transferred; every time a table is transferred, an equivalent but not equal under comparison is created. Moreover, we established a limit on the number of nesting levels that a table may have, to prevent an application transferring a table to crash due to memory issues. The direct transfer of tables does not allow cyclic tables to be transferred, although in future works, the mechanism we implemented to transfer Lua functions could be used as a model to implement a solution that removes this limitation. Because the limitations of the mechanism do not represent a loss in flexibility, its incorporation to Luaproc facilities the transfer of tables between

Lua processes.

We next implemented a mechanism for transferring userdata between Lua processes, ensuring that they are only accessed by the receiver Lua process. If the userdata requires specific handling, the programmer can register the transfer functions to be used. Otherwise, the mechanism transfers any userdata in a predefined way.

The requirements defined by *regudata* on the table it accepts as an argument allow all the transfer functions provided by a library to be registered in a simple way. The author of the library may put together these functions in a table, and export a *transf_udata* function that returns this table. Moreover, *regudata* allows a programmer to take transfer functions separately and make up a table fitting these requirements with them. This introduces flexibility in registering transfer functions.

The transfer functions must transfer an userdata maintaining no links with the sender Lua process, thus avoiding that the sender's lifecycle affects the userdata's structure. In order to allow the proper manipulation of a transferred userdata, these functions associate the userdata metatable to the transferred object, thus requiring the library creating the userdata to be loaded in the receiver Lua process. The use of different transfer functions for sending and receiving userdata allows the programmer to define different transfer logics for each of these operations.

The use of transfer functions incorporates flexibility to our mechanism, as any userdata can be transferred regardless of its structure. To transfer an userdata, the transfer mechanism uses the name of its metatable. Because in versions prior to Lua 5.3 the metatables do not store its name, we require that the metatables of userdata to be transferred in such versions store its name.

Transferring Lua functions was the next goal in our work. The mechanism we implemented allows transferring most basic Lua types as upvalues, including those which perform recursive calls to the corresponding function. As we cannot serialize C functions, we decided to look for the equivalent functions in the receiver Lua state, as an alternative for transferring them as upvalues. This gives rise to the main requirement of this mechanism: the libraries associated to those C functions which are employed by the transferred Lua function must be loaded in the Lua processes involved in the transfer. This requirement does not represent a disadvantage, as it is a characteristic of mechanisms transferring those values that maintain a link with libraries loaded in the Lua states where they are stored.

The alternative we considered for transferring C functions as upvalues, mentioned in Section 3.2.3, also allows us to transfer them as values of a

message. Therefore, we do not necessarily have to transfer C functions as upvalues of a Lua function.

By using our mechanism in the introducing scenario mentioned in Section 3.1.3, we may define routines in the distributor Lua process and send them to the worker Lua processes, which simply execute the received routine. Furthermore, if it were necessary to customize the execution of a routine by changing the values stored in some of its upvalues, this alternative allows us to do so without the intervention of the worker Lua process responsible for executing it.

In such scenarios, the implemented mechanism allows us to define most of the functioning logic of the application in the distributor Lua process, facilitating modifications. For this sample application, the utility of this mechanism is not to provide performance improvements, but to provide flexibility.

The mechanism for asynchronous message sending stores the messages in transit in a Lua state, thus allowing a sender Lua process to continue its execution without having to wait for the message to be consumed. We tried to incorporate this functionality to Luaproc making the fewest possible changes to its API. If we want to create an asynchronous channel, we only have to pass an extra parameter to the *newchannel* function. The *send*/*receive* and *delchannel* functions can be employed to send/receive messages and destroy asynchronous channels respectively, with no modifications on their use. The programmer only has to choose between a synchronous or asynchronous channel to define the kind of sending to be performed. Furthermore, not to make changes on the library's API allows using this version of Luaproc in already implemented applications without causing compatibility issues.

In Luaproc, the application will remain active until all Lua processes have terminated; similarly, with the asynchronous extension, an application will remain active until all the messages sent asynchronously have been received. This is important because such messages may store userdata, which need to be finalized properly.

Moreover, this mechanism demands less memory resources than the simulator one, because it employs a single Lua state for storing messages in each asynchronous channel, instead of creating a new Lua state for each sending. However, the fact that no other Lua process can interact with an asynchronous channel while a Lua process accesses its corresponding container Lua state may affect the communication through asynchronous channels, because the longer the time it takes for *send* to perform the copy of the sent message, the longer will be the time in which all the Lua processes attempting to interact with this

channel will remain blocked.

One final objective defined in our work consists of providing support for the collective communication operations among Lua processes, by incorporating the *barrier* function to the Luaproc library. In order to evaluate the benefits provided by this extension when implementing such operations, we established a comparison between implementations of the *scatter* operation using both the mechanism simulating a barrier and the *barrier* function itself.

For this, we implemented two functions (let's call them *scattersend* and *scatterrecv*) simulating sending and receive scatter operations. The *scattersend* function divides the collection of items to send into n groups, and sends them to the n receiver Lua processes. It then sends n notifications to a barrier channel; that is, a channel where the receiver Lua processes wait for the arrival of the remaining ones, after receiving their corresponding elements. Moreover, *scatterrecv* receives the sent items and waits for a notification to arrive, through the barrier channel, indicating that the Lua process executing this function may resume its execution. This description corresponds to the implementation of the scatter operation, which employs the mechanism simulating a barrier. Instead of sending and receiving notifications through a barrier channel, the *scattersend* and *scatterrecv* functions in the second implementation execute the *barrier* function on this channel.

We performed a test by employing both implementations. In this test, the application consists of a Lua process distributing the elements of a table on a set of Lua processes, by using the scatter operation. The goal of the test was to establish a comparison between the execution times obtained by both implementations, after distributing a specific table several times. As a result, the implementation using the *barrier* function achieved a better performance regarding to that achieved by the simulating mechanism. As we mentioned in Section 3.2.5, the fact that this mechanism performs several calls to the *send* and *receive* functions for exchanging notifications could imply a greater number of context switches in comparison to those required by the *barrier* function, thus influencing in the results obtained in this test.

This extension basically minimizes the number of messages exchanges required for synchronizing the executions of a set of Lua processes, thus constituting an efficient synchronization mechanism on which collective communication operations can be implemented. Furthermore, the implementation of the *barrier* function avoids creating specific channels for performing this operation, thus allowing greater simplicity and flexibility in designing the logic of a concurrent application using collective communication operations.

4
Experiments and Results

In this chapter, we will use the extensions incorporated to Luaproc in implementing a set of test applications to evaluate to what extent these extensions make the use of Luaproc in such scenarios more appropriate. We will first see how the direct transfer of tables allows concurrent applications performing large amounts of table exchanges between Lua processes to achieve good performance. Next, we see how the asynchronous message sending allows concurrent applications to take advantage of the benefits provided by parallelism. Finally, we test the proper functioning of the mechanisms implemented for transferring userdata and Lua functions.

We performed all the tests presented in this chapter on a desktop computer with the following features: Intel(R) Core(TM) processor (i7-4790, 3.60 GHz, 4 Cores), 8 GB (RAM) and a 64-bit architecture. For those tests in which we analyze demand for memory resources, we employ the *lmprof* [23] tool. This is an automatic memory profiler that helps Lua programmers deal with memory bloat. It provides detailed reports regarding the amount of memory allocated for each of the functions executed within a chunk of Lua code.

4.1
Direct transfer of tables

We conducted a test in order to evaluate to what extent the incorporation of the direct transfer of tables to Luaproc may improve the performance of a concurrent application. Using the current version of Luaproc and that resulting from our modifications, we developed two implementations of a concurrent solution to the knapsack problem (see Appendix A). The large number of table exchanges between Lua processes required by this solution motivated us to use it in our test. Because the current version of Luaproc does not support the transfer of tables between Lua processes, the implementation using it employs the Luabins library for serializing/deserializing them.

We compared the two implementation measuring total execution time for each of them with different number of workers (Lua processes employed

for computing the solution) and threads (kernel threads created by using the *POSIX Thread* library).

We executed each run 3 times, taking as the execution time (T) of the run the average result. For all the executions we employed an instance consisting of: a knapsack with $12x10^3$ units of capacity; and $8x10^3$ objects, each of them with an associated weight of 3 units. Objects with odd and even identifiers have an associated value of 50 and 5 units, respectively. In order to know how irregular each group of executions is, we computed the standard deviation (SD) associated with their resulting times. Table 4.1 shows the execution times (in seconds) obtained by these implementations in each run. The implementation employing the current Luaproc library is denoted by *Current*, while *Modified* is used for that employing the Luaproc's version modified by us. The "%" column shows, in percent, the improvement provided by the mechanism we implemented relatively to the *Current* implementation.

Threads	**Workers**	**Current**		**Modified**		**%**
		T(s)	SD	T(s)	SD	
4	4	30.33	0.47	18.33	0.47	39.5
	8	29.0	0.82	20.0	0.82	31.0
	16	40.33	0.47	23.33	0.47	42.1
	32	76.33	0.94	40.33	0.47	47.1
8	4	34.0	0.82	23.0	0.82	32.3
	8	31.33	0.47	20.67	0.47	34.0
	16	41.33	0.94	23.0	0.82	44.3
	32	67.33	0.47	38.0	0.82	43.5
16	4	32.67	0.47	20.33	0.47	37.7
	8	33.33	0.94	21.67	0.94	35.0
	16	44.67	0.47	25.33	0.47	43.2
	32	75.67	0.94	43.33	0.94	42.7

Table 4.1: Improvement provided by the direct transfer of tables.

Regardless of the implementation employed for obtaining them, the results shown in Table 4.1 show the existence of two patterns regarding the number of both threads and workers to be used in a run. The first one shows us that an increase in the number of threads does not improve the resulting performance. In preemptive multithreading, the system performs context switches among threads in order to alternate their executions. Therefore, creating a large number of threads introduces a greater number of context switches, which represents an overhead for the performance of an application. Furthermore, this implies for some threads, a greater time interval between

two consecutive executions. This causes the execution of the corresponding process to be slower, also affecting the performance of the application. Usually, a number of threads near to the number of cores in the system offers the best performance.

The second pattern indicates that an increase in the number of workers affects the performance achieved on a run. Something similar happens here, but the context switches to alternate the executions of the workers are now controlled by the scheduler implemented in Luaproc. In addition, a greater number of workers introduces a greater amount of message exchanges to the application, which also affects its performance. Because increasing the number of either threads or workers does not provide better performance for any of the implementations, we can establish the comparison between them by using the results shown in Table 4.1. By taking into account the best performance achieved by each implementation, we obtain Figure 4.1. It shows how the *Current* implementation takes, as a average, almost twice the time required by the *Modified* implementation to compute the solution of the problem.

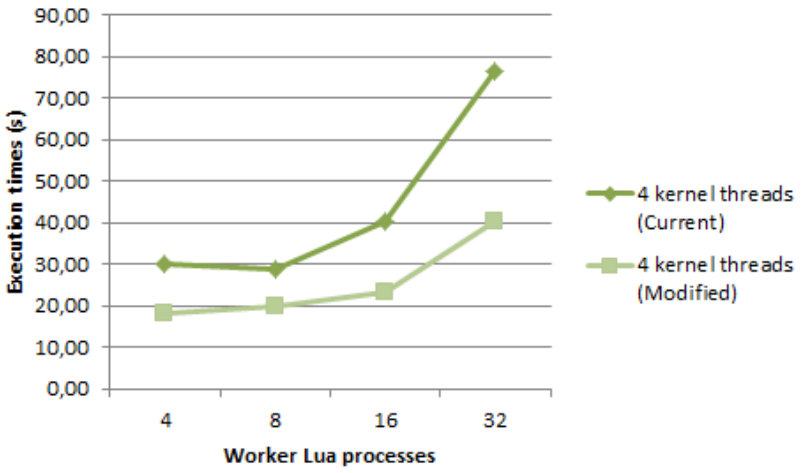

Figure 4.1: Performance of Luaproc when transferring tables.

When implementing an application, ensuring the minimum possible execution time is not the only point to consider. Demand for memory resources may restrict the usage scenarios of an application, since in those with insufficient memory the application will not reach the expected performance and could even crash. Because the serialization process performed by Luabins allocates a memory block to store the string representation of a table, the demand for memory resources associated to the *Current* implementation could suffer an increase in comparison to that associated to the *Modified* implementation. Therefore, we established a comparison between the amount of memory

required by each of these implementations when executed on the instances shown in Table 4.2.

Instances	**Knapsack's capacity**	**Amount of objects**	**Weight of each object**
A	1000	500	4
B	3000	2000	3
C	6000	4000	3
D	9000	6000	3

Table 4.2: Instances of the knapsack problem.

For all these instances, the values associated to the objects are the same as those we established for the instance we defined initially.

As the table exchanges between Lua processes occur when computing the matrix A, we focus on measuring the memory required by each implementation from the moment where workers begin computing the first row of A to the moment where each of them sends its corresponding collection *rows* to the main Lua process. We executed each implementation a total of 3 times on each of the above instances, and took the average required memory as a measure. Table 4.3 shows the demand for memory resources associated to each implementation. The "%" column shows the percent that the amount of memory required by the *Modified* implementation represents regarding to that associated to the *Current* implementation.

Instances	**Current** (Mb)	**Modified** (Mb)	**%**
A	53.6	20.6	38.42
B	670.7	316.7	47.22
C	2675.9	1258.4	47.03
D	6387.8	3762.6	58.9

Table 4.3: Memory required by Luaproc when transferring tables.

The results in Table 4.3 show that, as we had assumed, the serialization process increases the demand for memory resources in comparison to that required by the mechanism we implemented, when transferring a table. In the *Current* implementation, the serialization process is performed by calling the *save* function. For instance D, this function requires approximately 3004488840 bytes (2865.3 Mb) of memory in a total of 36004 calls. This amount of memory approximates the difference between those required by both implementations when processing the instance D, which corresponds to the fact that such implementations only differ on whether the serialization process is employed.

The improvement shown by the direct transfer of tables, regarding to both execution time and amount of required memory associated to an instance may vary depending on the alternative employed for serializing/deserializing a table. However, beyond the fact that direct transfer of tables has achieved better results when compared with those associated to Luabins, this test shows that the use of serialization/deserialization processes for transferring a table implies an increase in both the time interval and the amount of memory resources.

4.2 Sending messages asynchronously

In this section, we will evaluate the impact that asynchronous message sending has on those applications which are implemented following a provider-/consumer style. For this, we defined a scenario and established comparisons between different implementations of a concurrent solution described below.

In this scenario we want to achieve two different objectives. The first is to sort a set of integer arrays, which are previously stored in files. Each file stores an array per line, varying the number of lines from one file to another. The second objective is to obtain information about the set of arrays. In this test, the amount of arrays to be processed, the amount of arrays with a same length, minimum and maximum integer found globally in such arrays and amount of both odd and even numbers in the set, make up the summary we want to obtain from the set to be processed. We would also like to obtain this summary as soon as possible, that is, without having to wait for the whole set of arrays to be processed.

We could define an arbitrary number of Lua processes processing the arrays stored in such files and gathering the required information. In this way, we will not be able to achieve our second goal. We need to separate the task of gathering the information from that of sorting arrays. We then implemented a concurrent solution, by following a provider/consumer style.

In this solution, each file is handled by one *provider*. This provider reads a line from a file and draws from it the information required to make up the summary. It then sends this line, which actually is an array, to an available consumer. Once the consumer receives an array, it is responsible for sorting it and storing the result in a file.

The application routine defines an arbitrary number of providers and consumers, in addition to a main Lua process which assigns the reading of a file to a provider. This provider reads a file line by line, sending each of them along with the name of a file to consumers. A provider draws the information

required to make up the summary at each reading, and once it has finished reading a file, performs the same routine with the next one. Once all files have been read, the providers send the data they were able to gather to the main Lua process, which composes the required summary. On the other hand, a consumer receives a message and sorts the corresponding array. Once it has stored the resulting array in a file, the consumer becomes available to process the next one.

Although the provider/consumer style allows us to achieve our goals separately, the semantics of message sending may influence the time it takes to obtain the summary. Therefore, we developed several implementations of the above routine by employing both a synchronous and an asynchronous message sending, and compared the time each of them takes. For the asynchronous sending, we consider both the simulating mechanism and the one implemented.

Each run of the concurrent solution is defined by the number of providers and consumers to be employed. In all the runs we employed a same data set, which is composed of 4 files. Each of them stores between 31505 and 31510 arrays, each of which in turn stores between 595 and 600 integers. In each case, we considered the time that it takes to obtain the summary. We took this time as a performance measure for an implementation. In order to measure a time interval, we employed the *time* function that the *os* Lua library provides. We again took the average result (P) of three executions. We employed 4 kernel threads in all the executions, as usually employing an amount of these close to the number of cores in the system provides the best results. Table 4.4 shows the performance achieved by these implementations in each of the defined runs. We employ *Sync*, *Async sim*, and *Async* for denoting the implementation using a synchronous, simulated asynchronous or asynchronous sending, respectively.

This table also shows the time (T) that an implementation took for sorting the set of arrays, and in "%" column, the improvement provided by the mechanism we implemented regarding the best performance obtained by employing either the synchronous or simulated asynchronous sending. In this way, we can see how far in advance we can get the summary information in relation to the time the implementation takes for sorting all the arrays.

We observe that for both synchronous and asynchronous sending a continuous increase in the number of consumers does not provide better performance. Increasing the number of consumers increases the number of context switches that the Luaproc's scheduler must perform, which has a negative influence on the gain we could get from processing a greater number of arrays concurrently. Increasing the number of providers would not be useful to us, because in this test we employ only 4 files. Therefore, in the comparison, we

Providers	Consumers	Sync		Async sim		Async		%
		P(s)	T(s)	P(s)	T(s)	P(s)	T(s)	
1	2	53.0	53.0	28.33	40.33	20.33	39.33	28.2
	4	32.33	32.33	31.67	31.67	22.67	25.66	28.4
	8	29.33	29.33	37.33	37.33	29.33	29.33	0.0
	16	29.67	29.67	38.33	38.33	29.33	29.33	1.1
2	2	37.33	37.33	13.67	38.66	11.33	41.33	17.0
	4	33.67	33.67	22.67	25.66	19.33	25.33	14.7
	8	24.67	24.67	24.67	24.67	20.33	24.33	17.5
	16	24.67	24.67	25.0	25.0	20.67	24.66	16.2
4	2	38.33	38.33	10.33	41.33	9.67	40.66	6.4
	4	23.67	23.67	13.67	15.66	11.33	26.33	17.0
	8	24.67	24.67	25.67	25.67	11.67	26.66	52.7
	16	25.0	25.0	26.0	26.0	12.67	27.66	49.3

Table 4.4: Synchronous and asynchronous sending in Luaproc.

take into account those runs in which each of these implementations achieves its best performance. Figure 4.2 illustrates the performances achieved by each implementation in those runs where 4 providers are employed.

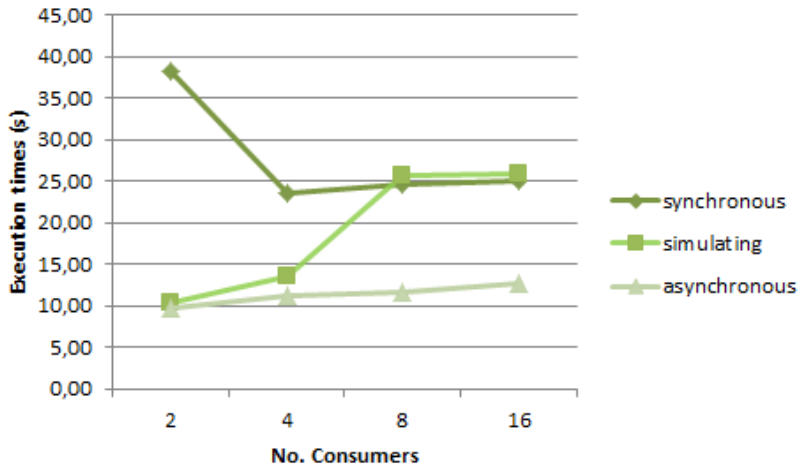

Figure 4.2: Synchronous and asynchronous sending in Luaproc.

This figure shows us that employing a synchronous sending in such a scenario, and generally in those where an independence among the executions of two different tasks communicating with each other is required, is not suitable. In this scenario, sorting the set of arrays takes a grater time interval in comparison to drawing the required information from them. As we mentioned in Section 3.1.4, two Lua processes communicating through a synchronous sending must synchronize to transfer a message. This is the reason why the

implementation sending messages synchronously takes a greater time interval for making up the summary in comparison to the others.

The use of an asynchronous sending is ideal for these scenarios. It allows two Lua processes to communicate without affecting the executions of their corresponding tasks. The fact that messages sent asynchronously are stored in a channel or received by temporary Lua processes, allows providers to send an array to consumers and immediately begin processing the next one. The obtained results reflect the benefits of this characteristic of the asynchronous sending.

Figure 4.2 shows that both implementations sending messages asynchronously achieve a similar performance regarding to the times they take for making up the summary. However, the mechanism simulating an asynchronous sending creates temporary Lua processes continuously, which could increase the demand for memory resources by the implementation employing it. Moreover, storing a message in transit in a container Lua state may cause the asynchronous messages sending to require a greater amount of memory resources in comparison to the synchronous sending. We then established a comparison between these implementations, regarding to the demand for memory resources associated to each of them when sending messages. In order to evaluate how the corresponding demands vary regarding to the number of arrays to be sorted, we defined the data sets shown in Table 4.5. Each of them only differs on the number of arrays stored in each file, regarding to the data set we defined initially.

Data sets	**Number of arrays stored in each file**
A	In range [5505..5515]
B	In range [10505..10515]
C	In range [20505..20515]
D	In range [31505..31515]

Table 4.5: Sets of integer arrays.

We executed each implementation a total of 3 times on each of the above data sets, taking the demand for memory resources resulting in each execution. The average amount of memory required for processing each data set is shown in Table 4.6. The "%" column shows the percent that the amount of memory required by the implementation using the mechanism we implemented represents, regarding to that associated to the simulating mechanism.

As we had assumed, the continuous creation of temporary Lua processes by the simulating mechanism increases the demand for memory resources

Sets	**Synchronous** (Mb)	**Simulating** (Mb)	**Asynchronous** (Mb)	%
A	40.0	89.3	50.7	53.84
B	80.3	170.4	96.8	53.83
C	153.0	332.5	189.0	53.83
D	223.5	510.9	290.3	53.83

Table 4.6: Memory required by Luaproc when sending messages.

associated to the implementation employing it. A Lua process is created by calling the *newproc* function, which requires approximately 1828 bytes (1.78 KB) of memory in its execution. When the number of calls to this function is large, the amount of memory required by this becomes remarkable. According to reports provided by the *lmprof* tool, for the data set *D*, the *newproc* function required about 230391980 bytes (219.7 Mb) of memory in a total of 126035 calls. This amount of memory approximates the difference between those required for both implementations when processing the data set *D*, which corresponds to the fact that such implementations only differ on whether this function is employed. Moreover, the mechanism we implemented creates only one Lua state in each asynchronous channel, to store messages in transit.

Table 4.6 also shows that buffering messages in the asynchronous messages sending implies an extra cost. When there are no receiver Lua processes waiting for messages in an asynchronous channel, the sender process stores the message in the container Lua state. When any receiver Lua process eventually reaches this channel, Luaproc again copies the message from the container Lua state to this process. This does not happen in the synchronous sending, as sent messages are always stored in the receiver Lua states. That different is the reason why the asynchronous messages sending tends to require a greater amount of memory resources than the synchronous one.

Employing this scenario for the experiment allowed us to test the proper functioning of our mechanism and observe how the asynchronous sending facilitates the use of the parallelism in multiprocessing environments. Moreover, the study performed on the amount of memory required by the simulating mechanism showed us that an alternative initially considered for implementing our mechanism was not adequate. Similarly to the simulating mechanism, this alternative creates temporary Lua states for each asynchronous sending.

4.3 Distributed web server

We employed the mechanisms described in Sections 3.2.2 and 3.2.3, which are intended to transfer userdata and Lua functions respectively, for implementing a test application. Our main purposes with this application were to evaluate the proper functioning of these mechanisms and to illustrate their utility.

The test application simulates a distributed web server, which is composed of a *controller* Lua process and a set of *worker* Lua processes. This server offers a set of services, which are defined in the controller and implemented by using Lua functions. The controller is responsible for establishing a connection with a client application and, depending on the service requested by it, transferring both the corresponding function and the established connection to one of the workers. Moreover, a worker executes the received function as many times as required by the client. In each execution, a worker employs the transferred connection for either receiving the input data of this function or sending the corresponding response.

The set of services offered by this server is composed of two algorithms for sorting arrays of integers. Because such algorithms may use different criteria for comparing two integers, we also defined two Lua functions representing comparison criteria. In this way, we can employ a same algorithm for sorting a set of integers according to either the parity or the sign that these have. Defining comparison criteria for the sorting algorithms allowed us to check that Lua functions with at least one upvalue were transferred in a proper way.

When a connection attempt is accepted by the controller, the client application selects both the sorting algorithm and the comparison criteria it needs to employ. The controller then customizes the execution of the corresponding function, by updating the upvalue storing the function used for comparison, and sends it along with the established connection to an available worker. This worker can then receive an array to be sorted through the transferred connection, execute the sorting algorithm on it, and send back the sorted array. Once it has finished processing all the client's requests, a worker notifies to the controller about its availability.

The distributed web server allows established connections to be handled concurrently by the workers. These client-server connections are established through Luasocket's sockets, which are instances of userdata. Therefore, we need to employ the userdata transfer mechanism for transferring the established connections between coordinator and workers. If userdata could not be transferred, instead of having a distributed web server we would have a sequen-

tial one, in which only one connection could be processed at a time regardless of whether the server was executed in a multiprocessing environment or not. Despite executing the services requested by a set of clients concurrently, a sequential server can only receive a request or send a response at a time. Therefore, a benefit provided by the userdata transfer for this application is to make use of the processing capacity of multiprocessing environments in order to provide better response times to the client applications.

We executed the distributed web server and established connections with some client applications, which are modified instances of the application described in the previous section. We performed some modifications on that application, so that the consumers it defines employ the services provided by this server, for sorting arrays of integers. Each of the client applications selected the sorting algorithm to employ and, consequently, sent several arrays to be sorted. As a result, these arrays were sorted by the workers and sent back to these applications. In this way, we checked that both userdata and Lua functions were transferred between Lua processes successfully.

5
Conclusion

In this work we incorporated a set of functionalities to Luaproc[25], as alternative solutions to limitations that the library still presents. Some of these functionalities are aimed at expanding the support to basic Lua types on messages transfer, while others are intended to make the use of Luaproc more suitable for the exploration of parallelism.

Because the table, userdata and function data types are often employed in the implementation of Lua applications, we incorporated mechanisms to support them on messages transfer to Luaproc. On the other hand, providing support for both an asynchronous message sending and the implementation of collective communication operations allows for a better use of the processing capacity of multiprocessor environments by concurrent applications.

The direct transfer of tables between Lua processes removes the need to serialize a table before transferring it. By avoiding the serialization/deserialization processes, this mechanism reduces both the time interval and the amount of memory resources that transferring a table requires, in comparison to using a serializer mechanism.

We implemented a mechanism to transfer userdata, which ensures that these are accessed by only one Lua process at a time. The fact that the mechanism optionally employs transfer functions allows any userdata to be transferred regardless of its structure, as well as employing different transfer logics when sending and receiving an userdata. For those userdata that do not require updates on their structure once transferred, a predefined transfer function is available. This exempts the programmer from providing transfer functions in that cases. The mechanism also defines simple interfaces for both the implementation of transfer functions and registration of these, thus making it easier for programmers to transfer userdata between Lua processes.

A function to be transferred can have all those basic Lua types supported in the messages exchange so far as upvalues. We implemented an alternative to transfer upvalues storing C functions, since we cannot serialize these functions. As a consequence, we may now transfer C functions as part of messages, under certain conditions.

To support asynchronous message sending, we created a new kind of channel in Luaproc. This channel acts as a container for messages in transit, so that the Lua processes sending messages through this channel may continue their executions without having to wait for receiver Lua processes to arrive. This removes the limitation that the synchronous sending often imposes on the concurrency realizable by an application, and therefore, facilitates the use of parallelism in multiprocessing environments. We tried to make as few modifications as possible in the Luaproc's API when incorporating the corresponding mechanism, avoiding compatibility issues in the use of the library for already implemented applications. The fact that the mechanism does not remove a message in transit until it has received a confirmation of receipt by the receiver Lua process gives guarantees on a safe delivery of those messages sent asynchronously.

The incorporation of the *barrier* function to Luaproc provides an efficient mechanism for establishing synchronization points among Lua processes, which is a key piece for the implementation of collective communication operations. Instead of employing synchronous channels and coordinator Lua processes, using this function decreases the number of message exchanges, and therefore the number of context switches, that synchronizing the executions of several Lua processes requires. Moreover, the characteristics of the implementation of this functionality allow us to transfer messages and establish synchronization points on a same channel, without generating conflicts in the communication of the involved Lua processes.

Each of the implemented functionalities, with its corresponding requirements and limitations, contributes to expanding the use of Luaproc to new scenarios, and therefore, the support for concurrency that it offers to the Lua language. However, we can keep increasing the power of Luaproc. Incorporating functionalities aimed at making it easier for programmers to implement concurrent applications or improve the performance of those functionalities already defined are some of the tasks we may perform for such a purpose. Next, we present briefly some insights to explore.

Because a Lua process is an user-defined C type storing basically a Lua state, we could use userdata for handling its execution from the application level. This control may include associating priority levels to the execution of a Lua process. An alternative to do so is to incorporate a functionality to such userdata (through its metatable) allowing us to assign a certain priority to the execution of a Lua process. This may involve making some changes to the Luaproc's scheduler, so that it can take these priorities into account when executing Lua processes. The fact that we can convey to

Luaproc our preferences on the execution of several sets of tasks would increase even more the flexibility provided by Luaproc when implementing concurrent applications.

We could also take control over the lifecycle of a Lua process, that is, make Lua processes cancellable. Currently, if we wanted a Lua process to finish its execution due to factors external to its execution scope (e.g. a task to be performed by several Lua processes, this process included, is accomplished by some other), we should implement stop conditions using messages exchange. Having control of a Lua process through an userdata, an alternative that we could consider is to incorporate a functionality to such userdata to notify a Lua process that it must finish its execution. Since a Lua process must employ a way for finishing its execution (if necessary), we could provide a mechanism for such a purpose as a new function or as a part of those functions already defined by Luaproc. In this way, the execution of a cancelled Lua process would end when it executes some of the functions registered by Luaproc (e.g. *send*, *receive*). In this case, no messages exchange to finish the execution of a Lua process from outside is required.

Once these functionalities have been implemented, we would get a kind of ownership on the execution of a Lua process. Considering that we are now able to transfer userdata, it would be interesting to investigate whether both handling the execution of a Lua process and transferring such an ownership offer some benefit (e.g. flexibility, simplicity) in implementing concurrent applications.

Moreover, we could re-use a Lua state for several asynchronous channels. This may allow an application sending messages asynchronously to require less amount of memory resources than that it may require using the current asynchronous sending.

Finally, we could evaluate the performance of the mechanism implemented to support the asynchronous sending on larger scale applications. This would allow us to study the impact the fact that no other Lua process can interact with an asynchronous channel while a Lua process accesses its corresponding container Lua state may cause on that performance.

Bibliography

[1] D. Sima. Decisive aspects in the evolution of microprocessors. **Proceedings of the IEEE**, 92(12):1896–1926, 2004.

[2] S. M. Shatz. Communication Mechanisms for Programming Distributed Systems. **Computing**, 21–27, June 1984.

[3] N. J. Carriero; D. Gelernter; T. G. Mattson; A. H. Sherman. The Linda alternative to message-passing systems. **Parallel Computing**, 20(4):633–655, 1994.

[4] R. Ierusalimschy; L. H. De Figueiredo; W. Celes Filho. Lua-an extensible extension language. **Software – Practice and Experience**, 26(6):635–652, 1996.

[5] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. **Dr. Dobb's journal**, 30(3):202–210, 2005.

[6] N. Benton, L. Cardelli; C. Fournet. Modern concurrency abstractions for C#. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, 26(5):769–804, 2004.

[7] A. Skyrme; N. de La Rocque Rodriguez; R. Ierusalimschy. Exploring Lua for Concurrent Programming. **Journal of Universal Computer Science**, 14(21):3556–3572, 2008.

[8] D. Namiot; V. Sukhomlin. JavaScript Concurrency Models. **International Journal of Open Information Technologies**, 3(6):21–24, 2015.

[9] A. Skyrme; N. Rodriguez; R. Ierusalimschy. Scripting Multiple CPUs with Safe Data Sharing. **Software, IEEE**, 31(5):44–51, 2014.

[10] A. Lenart. An Introduction to Multithreading with Programming Examples in C for the Windows NT Platform. **The Design of Operating Systems**, 1–15, April 1998.

[11] U. Drepper; I. Molnar. The native POSIX thread library for Linux. **White Paper, Red Hat Inc**. 2003.

[12] I. Foster. **Designing and building parallel programs**. Addison Wesley Publishing Company Reading, 1995.

[13] S. Martello; P. Toth. **Knapsack problems: algorithms and computer implementations**. John Wiley & Sons, Inc. 1990.

[14] J. Armstrong. **Programming Erlang: software for a concurrent world**. Pragmatic Bookshelf, 2007.

[15] R. Ierusalimschy. **Programming in Lua, Fourth Edition**. Lua.org, 2016.

[16] D. M. Beazley. **Python essential reference**. Addison-Wesley Professional, 2009.

[17] D. Caromel; L. Mateu; E. Tanter. **Sequential object monitors**. In: Proceedings of the European Conference on Object Oriented Programming, ECOOP '04, pages 317–341, 2004. Springer.

[18] G. Chrysanthakopoulos; S. Singh. **An asynchronous messaging library for C#**. In: Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages, OOPSLA, pages 89–97, San Diego, CA, USA, 2005.

[19] J. Armstrong. **Erlang—a Survey of the Language and its Industrial Applications**. In: The 9th Exhibitions and Symposium on Industrial Applications of Prolog, INAP '96, pages 16–18, Hino, Tokyo, Japan, 1996.

[20] P. Haller; M. Odersky. **Event-based programming without inversion of control**. In: Modular Programming Languages, pages 4–22, 2006. Springer.

[21] A. Skyrme. **Um modelo alternativo para programação concorrente em Lua**. Master's thesis, PUC-Rio, 2008.

[22] A. Skyrme. **Safe Record Sharing in Dynamic Programming Languages**. PhD thesis, PUC-Rio, 2015.

[23] P. Musa. **Profiling memory in Lua**. Master's thesis, PUC-Rio, 2015.

[24] T. Salmito. **Leda**. `https://github.com/salmito/leda`. 2012.

[25] L. Millan. **An extension to the Luaproc library**. `https://github.com/lmillanfdez/luaproc-master`. 2016.

[26] Oak Ridge National Laboratory. **Parallel Virtual Machine**. `http://www.csm.ornl.gov/pvm/`. 2011.

[27] The Open MPI Project. **Open MPI: Open Source High Performance Computing**. `https://www.open-mpi.org/`. 2016.

[28] OpenMP. **The OpenMP API specification for parallel programming**. `http://www.openmp.org/specifications/`. 2016.

[29] A. Kauppi; B. Germain. **Lua Lanes - multithreading in Lua**. `http://lualanes.github.io/lanes/`. 2015.

[30] A. Gladysh. **Luabins. Lua Binary Serialization Library**. `https://github.com/agladysh/luabins`. 2016.

[31] Perl 6. **Perl 6 Documentation. Concurrency and Asynchronous Programming**. `https://docs.perl6.org/language/concurrency`. 2016.

[32] D. Nehab. **Luasocket: Network support for the Lua language**. `http://www.tecgraf.puc-rio.br/diego/professional/luasocket`. 2004.

[33] M. Welsh. **SEDA: Staged Event-driven Architecture**. `http://www.eecs.harvard.edu/ mdw/proj/seda/`. 2002.

[34] F. Alves. **Concorrência e compartilhamento em Lua, estudo do aprimoramento da biblioteca Luaproc**. Trabalho Final de Curso, PUC-Rio, 2015.

A
Knapsack Problem

The knapsack problem[13] is a well known NP-Complete problem in combinatorial optimization. We may define it as follows: given a set of items, each of them with an associated identifier, weight and value; the goal is to determine the identifiers of the objects conforming a collection so that its total weight is less than or equal to a given limit and its total value, gained by means of each object, is as large as possible.

By using dynamic programming, we may implement a solution for the knapsack problem that runs in pseudo-polynomial time. Let's assume that the weights $w_1, w_2, w_3, \ldots, w_n$, and the values $v_1, v_2, v_3, \ldots, v_n$, are strictly positive integers. An nxC matrix A (named solution matrix) is defined, where: C is the capacity of the knapsack; n, the amount of objects; and $A[i, c]$, represents the maximum value that can be gained using items whose identifiers are less than or equal to i and leaving the knapsack with weight less than or equal to c.

$A[i, c]$ is defined recursively as follows:

$$A[i, 0] = \left\{ \begin{array}{c} \text{-1, if i}\neq 0 \\ \text{0, if i=0} \end{array} \right\}$$

$$A[i, c] = max(A[i-1, c], A[i-1, c-w_i] + v_i)$$

The solution can then be found by computing $A[i, c]$ for all items and possible capacities.

Based on this solution, we designed a concurrent one to solve the knapsack problem. Next, we briefly describe it.

Similarly to the dynamic solution, we execute a set of steps, in each of which a row of the matrix A is computed as defined above. However, the solution involves an arbitrary number of worker Lua processes (let's call them *workers*), on which a *block distribution* [12] of matrix A is applied. In this way, each worker is responsible for computing the values associated with a set of cells (let's call it *fragment*), in each step. In order to determine a fragment in step i, a worker must access the row computed in the previous step. Therefore, each worker must share the fragment it computed in step $i-1$ before beginning

the step i. Each worker defines a collection *prevrow* for storing in each step, all those shared fragments to which it must access. A worker also defines a collection *rows* storing the fragments that it computed in each step, which will be used for rebuilding the matrix A, once all the rows have been analyzed.

A worker starts the processing corresponding to step i, by checking whether at least one row of A must be analyzed. If so, it shares (sends) the fragment computed in the previous step through a *chshare* channel, which is created specifically for each worker. A worker shares a fragment as many times as workers requiring access it there. It then receives and stores in *prevrow* those fragments shared by workers responsible for processing the columns previous to theirs. A worker uses both fragments in *prevrow* and that it computed in step $i - 1$ for computing the fragment corresponding to step i. Once it has computed such a fragment, a worker stores the fragment it computed in step $i - 1$ in *rows* and continues with the step $i + 1$.

If all the rows of the matrix A have been computed, a worker adds the fragment computed in the last step to the collection *rows* and sends this to the main Lua process. This uses the collections *rows* sent by all the workers for rebuilding the matrix A, which it then employs for determining the set of objects that generates the greater possible gain without exceeding the capacity of the knapsack.

B
Functions for transferring userdata

B.1
An example of a sender transfer function for Luasocket's sockets

```c
static int luaproc_send_socket(lua_State *L){

/* userdata's metatable name */
const char *mt_name = NULL;

size_t len;

/* size of the memory block associated to the userdata to
   be transferred */
int size_udata = 0;

/* validating the function's parameters */
luaL_checktype(L, 2, LUA_TUSERDATA);
luaL_checktype(L, 3, LUA_TLIGHTUSERDATA);

/* getting the function's parameters */

/* "transfer_type" may be either "1" (indicates the
   userdata is being transferred to another Lua processs)
or "3" (indicates the userdata is being transferred to a
   container Lua state)*/
int transfer_type = luaL_checkinteger(L, 1);

/* getting a pointer to the userdata to be transferred */
const void *udata_pointer = lua_touserdata(L, 2);

/* getting a pointer to the receiver Lua state */
lua_State *Lto = (lua_State *)lua_touserdata(L, 3);

/* getting the userdata's metatable name in the sender Lua
    state */
lua_getmetatable(L, 2);
lua_pushstring(L, "__name");
lua_rawget(L, -2);
```

```
31
32 mt_name = lua_tolstring(L, -1, &len);
33
34 /* getting the userdata's metatable in the receiver Lua
       state */
35 luaL_getmetatable(Lto, mt_name);
36
37 if(lua_isnil(Lto, -1)){
38   /* userdata's metatable is not registered in the
         receiver Lua state  */
39
40   if(transfer_type == 1){
41     /* userdata is being transferred to another Lua state
         */
42     /* this function must leave error messages in both Lua
          states */
43     lua_pushnil(L);
44     lua_pushstring(L, "userdata's metatable must be
         registered in the receiver Lua process");
45
46     lua_settop(Lto, 1);
47     lua_pushnil(Lto);
48     lua_pushstring(Lto, "userdata's metatable must be
         registered in the receiver Lua process");
49
50     return 2;
51   }
52   else{
53     /* userdata is being transferred to a container Lua
         state */
54     /* this function must create (or get) the metatable to
          be associated to the resulting userdata */
55     lua_pop(Lto, 1);
56     luaL_newmetatable(Lto, mt_name);
57
58     /* storing the metatable's name, in versions prior to
         Lua 5.3 */
59     #if (LUA_VERSION_NUM < 503)
60       lua_pushstring(Lto, "__name");
61       lua_pushlstring(Lto, mt_name, len);
62       lua_rawset(Lto, -3);
63     #endif
64   }
65 }
66
67 /* getting the size of the memory block associated to the
       userdata, in the sender Lua state */
```

```
68 size_udata = lua_rawlen(L, 2);
69
70 /* creating the resulting userdata in the receiver Lua
      state */
71 p_tcp tcp = (p_tcp)lua_newuserdata(Lto, size_udata);
72
73 /* copying the userdata's structure to the receiver Lua
      state */
74 memcpy((void *)tcp, udata_pointer, size_udata);
75
76 /* updating the structure of the resulting userdata */
77 p_io io = &tcp->io;
78 io->ctx = &tcp->sock;
79
80 p_buffer buf = &tcp->buf;
81 buf->io = io;
82 buf->tm = &tcp->tm;
83
84 /* associating the corresponding metatable to the
      resulting userdata  */
85 lua_pushvalue(Lto, -2);
86 lua_setmetatable(Lto, -2);
87 lua_remove(Lto, -2);
88
89 /* if no error occurs, a sender transfer function returns
      no values */
90 return 0;
91 }
```

B.2
An example of a receiver transfer function for Luasocket's sockets

```
1  static int luaproc_recv_socket(lua_State *L){
2
3  /* userdata's metatable name */
4  const char *mt_name = NULL;
5
6  /* size of the memory block associated to the userdata to
      be transferred */
7  int size_udata = 0;
8
9  size_t len;
10
11 /* validating the function's parameters */
12 luaL_checktype(L, 3, LUA_TLIGHTUSERDATA);
13
14 /* getting the function's parameters */
```

```
15
16 /* "transfer_type" may be either "2" (indicates the
       userdata is being received from another Lua processs)
17 or "4" (indicates the userdata is being received from a
       container Lua state) */
18 int transfer_type = luaL_checkinteger(L, 1);
19
20 /* getting the index within the stack of the sender Lua
       state in which the userdata to be transferred is stored
        */
21 int udata_idx = luaL_checkinteger(L, 2);
22
23 /* getting a pointer to the sender Lua state */
24 lua_State *Lfrom  = (lua_State *)lua_touserdata(L, 3);
25
26 /* getting a pointer to the userdata to be transferred */
27 const void *udata_pointer = lua_touserdata(Lfrom,
       udata_idx);
28
29 /* getting the userdata's metatable name in the sender Lua
        state */
30 lua_getmetatable(Lfrom, udata_idx);
31 lua_pushstring(Lfrom, "__name");
32 lua_rawget(Lfrom, -2);
33
34 mt_name = lua_tolstring(Lfrom, -1, &len);
35
36 /* getting the userdata's metatable in the receiver Lua
       state */
37 luaL_getmetatable(L, mt_name);
38
39 if(lua_isnil(L, -1)){
40   /* userdata's metatable is not registered in the
        receiver Lua state */
41
42   if(transfer_type == 2){
43     /* userdata is being received from another Lua process
          */
44     /* this function must leave error messages in both Lua
          processes */
45     lua_pushnil(Lfrom);
46     lua_pushstring(Lfrom, "userdata's metatable must be
         registered in the receiver Lua process");
47   }
48
49   /* if the userdata is being received from a container
        Lua state */
```

```
50    /* this function cannot leave an error message in the
          container Lua state */
51
52    lua_settop(L, 1);
53    lua_pushnil(L);
54    lua_pushstring(L, "userdata's metatable must be
          registered");
55
56    return 2;
57  }
58
59  /* getting the size of the memory block associated to the
       userdata, in the sender Lua state */
60  size_udata = lua_rawlen(Lfrom, udata_idx);
61
62  /* creating the resulting userdata in the receiver Lua
       state */
63  p_tcp tcp = (p_tcp)lua_newuserdata(L, size_udata);
64
65  /* copying the userdata's structure to the receiver Lua
       state */
66  memcpy((void *)tcp, udata_pointer, size_udata);
67
68  /* updating the structure of the resulting userdata */
69  p_io io = &tcp->io;
70  io->ctx = &tcp->sock;
71
72  p_buffer buf = &tcp->buf;
73  buf->io = io;
74  buf->tm = &tcp->tm;
75
76  /* associating the corresponding metatable to the
       resulting userdata */
77  lua_pushvalue(L, -2);
78  lua_setmetatable(L, -2);
79  lua_remove(L, -2);
80
81  /* leaving nothing in the sender Lua state */
82  lua_pop(Lfrom, 2);
83
84  /* if no error occurs, a receiver transfer function
       returns the resulting userdata */
85  return 1;
86  }
```