Concurrency Oriented Programming in Lua

Eleftherios Chatzimparmpas

Master of Science Thesis
Stockholm, Sweden 2007

ECS/ICT-2007-138

Abstract

This thesis report presents a project that brings the Erlang model of concurrency to interpreted scripting languages.

A system that provides Erlang-style concurrency has been designed and implemented for the Lua programming language. This model of concurrency brings an alternative to what scripting languages have to offer today for concurrent and distributed programming. The implementation is based on the ideas that make Erlang, not only an inspiration, but a reference in the concurrent programming paradigm. The resulting system will help further change the view on the way to achieve concurrency.

Contents

List of Figures

Chapter 1

Introduction

One of the problems that the software engineering field has to work out today is that of efficient concurrent and parallel programming. Even though several approaches have been suggested, the situation today cannot be considered satisfactory. And this comes, because most programming languages and development platforms in general, are embracing solutions that are based on preemptive multithreading, which in turn requires further solutions to the race hazards problem.

Nevertheless, there are other models of concurrency that have been successfully used by some programming languages. One of these models is the message passing model, upon which Erlang, a declarative language targeted for concurrent and distributed programming, is based.

An interesting direction in the field of programming languages today are interpreted scripting languages. Lua, a dynamic general purpose, embeddable and extensible language, is one of the scripting languages that is becoming more and more widespread, due to some fresh ideas and a unique feature set.

This thesis will answer the question of how the actor model can be combined with and how it will benefit interpreted scripting languages. More specifically, how it will bring the Lua programming language closer to the goal of concurrent programming. This report will present the design and implementation of a concurrency solution for the Lua programming language. A system that has been designed by drawing inspiration from concepts and ideas used in Erlang, and implemented as to be integrated to Lua.

The main architectural design decisions of the system comprise some of the problems that will be solved during this thesis. The main functionality of the system includes processes that can execute concurrently and that share data by communicating with each other, and this functionality is also supported for distributed processes. The problems that stem from these system requirements are such as: how will the processes be implemented as independent threads of execution, how will these threads send and receive messages, what will messages be, and how all these concepts can be applied in a distributed system. In addition, how will this system be integrated seamlessly with the target language.

By solving these very problems it will then be possible to ripe some of the benefits for concurrency oriented programming. At this point questions about the importance of this system, its usefulness and its future prospects will be also answered. This evaluation of the system will prove the accomplishment of the initial objectives of this thesis.

The next chapters will be organized as follows. Chapter 2 offers background information that are needed to better understand the main idea and concepts of the project, the problem that it tries to solve. Chapter 3 provides the requirements and the specification of the system that was implemented. Chapter 4 includes details on the architectural and design decisions that had to be taken and a summary of the implementation work done. Chapter 5 presents the implemented system and discusses the results, possible uses of the system, along with a theoretical approach to multiprocessing. Chapter 7 concludes the report and suggests directions for future work.

Chapter 2

Background

A general introduction to the area, the Lua programming language, concurrency, multithreading, along with an investigation on previous related work, will comprise the contents of this chapter.

2.1 Scripting languages

In the recent years, there has been a paradigm shift towards a higher level programming approach, partially represented by the so called scripting languages [1], which are on the rise today.

Scripting languages are typically interpreted, while in some cases the *just-in-time compilation* or *dynamic translation* technique is used. Scripting languages are also characterized by dynamic typing and automatic type conversions, which altogether guarantees a better suited environment for rapid application development and a higher level of programming.

There is a specific type of scripting languages that are particularly interesting. These languages, which are considered *multi-paradigm programming languages*, have the following properties:

- They can be used as dynamic general-purpose stand-alone languages
- They can play the role of a "glue" language between applications and components
- They can be embedded into other applications and extend their functionality (extension languages)
- They can extended with other languages (extensible languages)

2.2 The Lua programming language

One of the most interesting scripting languages, that has gained more popularity the last years, is the Lua [2][3][4] programming language.

Lua is a dynamic general-purpose, embeddable and extensible, interpreted, scripting language, that is simple, powerful, fast, portable and lightweight. Lua combines a simple procedural syntax, powerful data description constructs and extensible semantics. This ability to extend its semantics is realized with the so called *meta-mechanisms*: *dynamic associative arrays*, *reflexive facilities*, *fallbacks*.

Lua's runtime system is also unique across the board, as it is based on a register-based Virtual Machine (VM), while source code is not directly interpreted, but is first compiled into bytecode for the VM. Lua also has dynamic data typing and includes automatic memory management facilities with garbage collection.

2.3 Concurrency models

It has become more than evident the last years that concurrency will lead the future of computing. Today with the first multi-core systems already inside most of the modern personal computers, it is very important for programming solutions to take advantage of this computational power.

Concurrent programming can be *explicit* or *implicit*. In explicit concurrency a set of primitives is provided to the application developer, by the use of which parallelism can be achieved. In implicit concurrency the developer does not need to take special action in order to exploit parallelism, as this is done transparently by some language constructs. Most of the mainstream programming languages today offer explicit concurrency programming.

In concurrent systems their components need to interact and communicate. Based on how the components communicate, they can be divided in two different classes: *shared memory* and *message passing*. In shared memory systems communication takes place by memory that is shared between the communicating components. In message passing systems the components communicate by exchanging messages.

2.3.1 Shared memory communication

The shared memory communication model is the most commonly used approach to exploit parallelism, as most programming languages include some kind of light-weight processes, usually called *threads*, that share memory space. Although this allows the most optimal access to the the data, data through which the threads communicate, it can bring the system in an undesirable state, due to race conditions.

Race conditions in multithreaded systems make the use of synchronization techniques mandatory. There have been suggested many mechanisms in order to guarantee that a concurrent system is free from race conditions. The most common mechanisms used, are based on some form of locking, such as mutexes, semaphores and monitors.

2.3.2 Message passing communication

In the message passing communication model, there is no shared memory between the components, and instead the components rely on some kind of communication mechanism in order to move data across the components' memory space. Most programming languages have external libraries to support message passing, while few languages provide support by built-in facilities.

There are different models of message passing based on the characteristics of the communication channels: *synchronous message passing* and *asynchronous message passing*. In the synchronous message passing model, *blocking* semantics are used for sending and receiving messages; the *send* operation completes only if the data have been sent and the *receive* operation completes only if the data have been received. In the asynchronous message passing model, the receive operation is blocking, but the send operation is non-blocking, and this makes sending asynchronous with regard to receiving.

2.4 Threads

A *thread* in computer languages represents a thread of execution, an executable unit of code within a program that executes independently from other parts of code inside the program. Threads are considered light-weight processes that are the smallest unit of code that can be scheduled to execute by the operating system.

2.4.1 Multithreading

Multithreading is to threads what multitasking is to processes, and gives to threads the ability to run concurrently or in parallel. Most programming languages provide some kind of threads and multithreading in order to support concurrent or parallel programming. It is the obvious solution to the need for parallelism that can be added easily upon a programming language and is supported by almost all of the modern operating systems today.

Apart from the usual threads that are distinguished as *native* or *operating system* or *kernel-mode* threads, there is another type of threads, the *interpreter* or *VM* or *user-mode* threads. Interpreter threads are implemented by most interpreted languages, instead of native threads, primarily because they are more portable and simple to implement in regards to synchronization with the interpreter itself.

2.4.2 Scheduling

There are two main scheduling policies for threads: *preemptive multithreading* and *cooperative multithreading*. The preemptive scheduling approach is the one that is generally preferred and considered superior. In preemptive multithreading the scheduler decides when each thread gets to run and can suspend and resume execution of threads at any time. In cooperative multithreading the threads themselves relinquish control and thus give the chance to other threads to run.

2.4.3 Problems

Threads and preemptive multithreading, with their race hazards and synchronization problems, and the various locking-based mechanisms that were designed to overcome these very problems, is the dominant model today. Threads and preemption are so popular because of the operating systems support for this model and the simplicity of adding the necessary language constructs to take advantage of this capability. Unfortunately, while it seems that it is straight-forward to add support for this model in any language, this practice opens a whole world of problems that the developers are called to face.

Threads *discard the most essential properties of sequential computation: understandability, determinism and predictability.* They are *a non-deterministic model of computation* that results in *non-trivial multithreaded programs being incomprehensible to humans* [5]. It is becoming more and more evident, especially with today's multi-core systems, that multithreaded programs are difficult to understand, hard to implement correctly in the first place and virtually impossible to debug.

It is more than clear than a different route will have to be followed in order to take advantage of the computing power and the parallelism of today's systems, and different models of concurrency have to be looked into.

2.5 Concurrent Programming

A brief description of the main concurrency facilities that are available in Lua and Erlang will follow.

2.5.1 Concurrency in Lua

Lua supports concurrency through *coroutines* [6], also called *collaborative* or *cooperative multithreading*. Lua includes a full implementation of *asymmetric coroutines*, also called called *semi-symmetric coroutines* or *semi-coroutines*.

Coroutines are generalized subroutines that allow multiple entry points, suspending and resuming of their execution, and persistence of data local to them. Coroutines represent an independent thread of execution, but because they need to cooperate and suspend when explicitly requested to, they can avoid locking, since they are resumed only in specific points inside a program.

2.5.2 Concurrency in Erlang

Erlang [7][8][9] is a declarative programming language for programming concurrent, real-time, distributed fault-tolerant systems. Erlang was designed with concurrent and distributed programming in mind, and has built-in facilities to support concurrency oriented programming.

Erlang has as a set of primitives to create light-weight VM threads, which are also known as *microthreads* or *green threads*. These threads, that are called *processes* in Erlang, communicate using a *share-nothing asynchronous message passing* system. They send and receive *messages* which are stored in and consumed from message queues named *mailboxes*. Any type of Erlang data structure can be passed as a message.

Distribution is supported transparently; Erlang processes map naturally onto distributed systems. An Erlang *node*, which represents an Erlang runtime system, is a host of Erlang processes. Processes in remote nodes can communicate with the same primitives that are used for communication between local processes.

In one of its relatively recent releases, Erlang has also gained support for "true" Symmetric Multi-Processing (SMP) parallelism, by adding multithreading support to the Erlang VM. This was implemented by running more than one Erlang process schedulers inside the Erlang VM, where each scheduler runs on a separate native thread, and thus the application developer does not have to handle multithreading explicitly.

2.6 Previous work

There have been some efforts to bring ideas either from Erlang directly or other solutions that are based on the message passing or actor model. Some of the most popular scripting languages have made attempts to overcome the problems of concurrent programming, either through new components or rewrites of their interpreter. Some of most important of these efforts, those that resulted in more complete and usable solutions, will be described.

2.6.1 ALua

ALua [10][11][12] is an event-driven environment with mobile code. Alua implements a communication mechanism that takes advantage of the interpreted nature of Lua in order make it possible to exchange messages that are actually chunks of code. These messages of code can be then executed by the receiver providing what is called *weak code mobility.* In ALua the role of processes play components called agents that run on different hosts and communicate through a network. ALua agents are implemented as operating system processes.

2.6.2 Io

Io [13] is a small, embeddable, prototype-based, pure object-oriented programming language. Io features coroutines, *actor-based concurrency* and *futures.* Actor-based concurrency is based on objects being able to send asynchronous messages that end up in another object's message queue. These objects which are processing message queues, are the actors, and they use coroutines and asynchronous I/O for their implementation. In addition, when a message is sent a transparent future object is returned, which will become the returned value, when that value is ready.

2.6.3 Stackless

The Stackless Python project [14] is an enhanced version of the Python interpreter, that targets to bring an easier model of concurrency to the Python programming language. It tries to take advantage of what multi-threading has to offer but with minimizing the complexity problems at the same time. Stackless Python features light-weight interpreter threads, called *microthreads* or *tasklets*, bidirectional communication objects between the tasklets, called *channels*, a cooperative and a preemptive tasklet scheduler and serialization of the tasklets.

Chapter 3

Method

A formal description of the system's requirements and specification will follow, and will serve as an introduction to what has been implemented.

3.1 Requirements

The system targets to bring concurrency oriented programming to the Lua programming language. This includes the ability to:

- Create new processes.
- Send messages to processes and correspondingly receive messages from processes.
- Use aliases instead of process identifiers for easier access to the processes themselves.
- Monitor processes and receive notifications when their status changes.
- Link processes together and receive signals when they terminate abnormally.

The second part of the system is the distributed programming functionality. This should extend some of the aforementioned properties:

- Processes on different network hosts can communicate, again by sending and receiving messages.
- Processes can be created on demand in remote network hosts.

- Global aliases for the processes that are shared between all network hosts can be set.
- Processes monitor remote processes and receive notifications when their status changes.
- Processes link to remote processes and receive signals when the remote processes terminate abnormally.
- A security mechanism takes care of the authentication of the remote network systems, before any kind of communication is allowed.

Furthermore, the following non-functional requirements are important:

- The system should follow one of the main principles of Lua, that of simplicity. In these lines, distribution should transparently work in the same way as concurrency, regardless of the location of the processes.
- Processes that terminate abnormally should not take the whole system down. When a process dies nothing else is affected, and the system continues its operation.
- The processes should be as light-weight as possible, so as to make it possible to achieve satisfactory scalability.
- The system is intended to be a prototype, but it should serve as the basis for a more complete system that can be extended easily.
- The system should be reasonably portable and run under most major platforms today.
- It would be desired for the system, to be based upon the infrastructure that Lua provides, instead of modifying Lua or adding large subsystems in Lua, in order to reimplement part of its functionality.

3.2 Specification

As is obvious from the system's requirements the system can be divided in two main subsystems, the one implements concurrency and the other distribution.

One of the main elements of the concurrency functionality supported by the system is the *process* (Figure 3.1). A process is defined as an interpreter thread, or else a microthread or a tasklet. The process in this system is like processes in an operating system; they don't share any memory, and

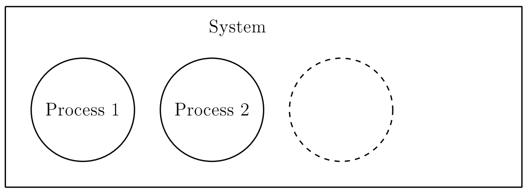

Figure 3.1: System processes and process creation

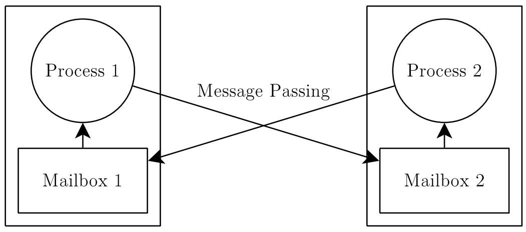

Figure 3.2: Message exchange between processes

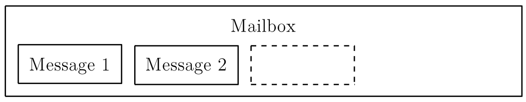

Figure 3.3: A process message queue

therefore they have to communicate with some interprocess communication mechanism. Processes can be created and destroyed on demand. A simple scheduler takes care of running the processes using a round-robin scheduling policy.

Each process has a *mailbox* associated with it (Figure 3.2). The mailbox is a message queue and plays the role of a temporary storage for messages that were sent to the process (Figure 3.3). When a process receives a message it checks for any messages available in the mailbox, and if the mailbox is not empty, the older message is removed from the mailbox and processed by the process.

Each process is identified by a unique *process identifier*, or else a process ID or PID. Instead of PIDs, which are difficult to remember, an alias can be set for the PID. This process name is resolved back to the PID each time it is used. The role of the central repository of names and PIDs plays a *registry* (Figure 3.4). Processes can query the registry or edit the registry, by adding, deleting or updating entries.

Monitors are a mechanism for error handling (Figure 3.5). A process can monitor the status of other processes. When a monitored process dies, all the processes monitoring it are notified via a special message. A process can

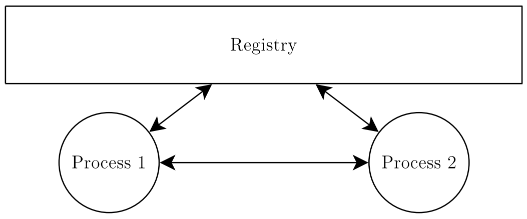

Figure 3.4: The process name registry

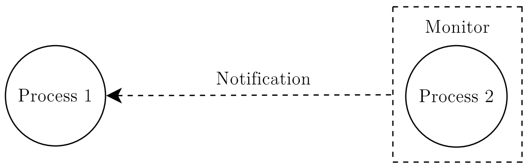

Figure 3.5: Process monitoring

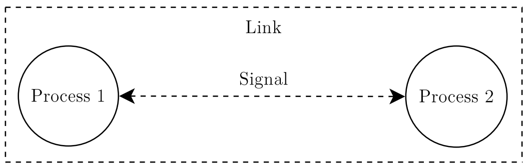

Figure 3.6: Process linking

stop monitoring another process at any point in time.

Links are also a mechanism for error handling (Figure 3.6). Two processes can be linked together, and when one of them dies, the other one is signalled. This signal causes the second process to die, too. The two processes can stop being linked together at any point in time.

Distribution properties are based on a component of the system that is called the *node* (Figure 3.7). A node represents a specific runtime of the system, which may have an arbitrary number of processes running inside it.

Nodes have the ability to communicate with each other. The nodes form a *virtual network* that the processes in every node can use in order to exchange messages with each other (Figure 3.8). In this way processes can exchange messages transparently with processes residing in remote nodes.

Nodes are identified by their name. In order for nodes to discover one another, a daemon that acts as a nameserver is used. This daemon is called the *port mapper daemon*. Exactly one daemon is running for each network host. The daemon contains details about the nodes that are running in the host of its responsibility. This information is vital for nodes to establish connections between them (Figure 3.9).

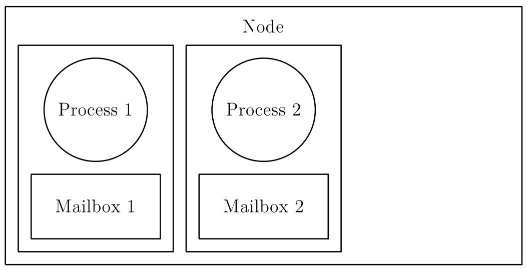

Figure 3.7: A node includes an arbitrary number of processes

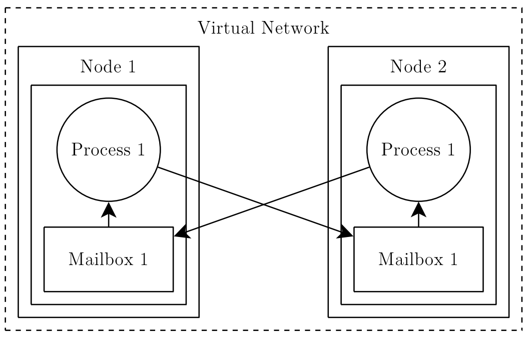

Figure 3.8: Distributed nodes form a virtual network

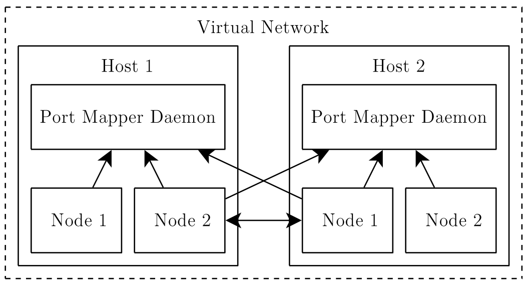

Figure 3.9: Nodes connect after communicating with the port mapper

As processes can be created on the local node, it is also possible to request the creation of processes in remote nodes. In this case, the PID of the created process is an object that allows access to the remote process, as if it was a local process.

The virtual network that nodes form has some additional properties. If the network is fully connected, where each node is connected bidirectionally to each other node, it is possible to support global aliases. Nodes can negotiate and keep a *virtual global registry* of process names that are resolved to PID handles (Figure 3.10), while they also keep local copies of the global registry.

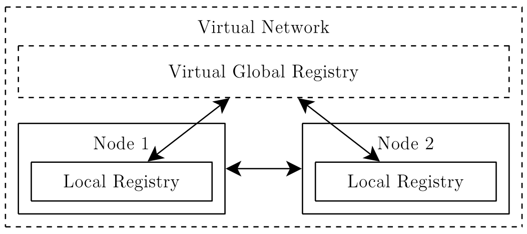

Figure 3.10: Nodes maintain a virtual global registry

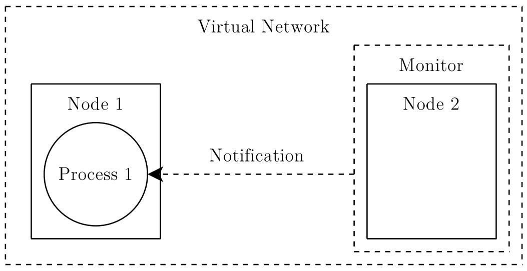

Figure 3.11: Node monitoring

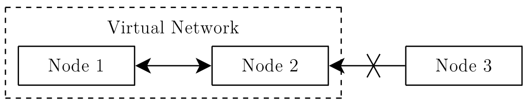

Figure 3.12: Node authentication

Monitors and links are supported transparently for remote processes. Nodes take care of the necessary communication and coordination in order to support remote monitors and links. Also supported is the capability for a process to monitor nodes as a whole, an additional error handling mechanism (Figure 3.11).

Nodes are required to authenticate before any further communication is allowed. Thus, a node cannot enter a network of nodes unless it has first authenticated. A security mechanism transparently takes care of all this work, each time a node communicates with another node (Figure 3.12).

Chapter 4

Implementation

An overview of the architectural design of the system that was implemented, along with more detailed design decisions for each component of the system, and some details on the implementation, are described in this chapter.

4.1 Architecture

As is already obvious by now, the system is targeted for integration with the Lua programming language. This left two choices for the implementation language: Lua and C. The choice of which of the two was to be used in preference, was in fact partially dependant on the architectural pattern that would be followed. There were two strategies that could be used in this regard:

- Modification of the Lua VM in order to create a enhanced version of Lua, probably with backwards compatibility to the standard distribution

- Implementation based on the Lua standard component system, in order to create a system that could be loaded from any standard Lua VM

The first approach should be only preferred in the case where the desired system and its functionality could not be implemented otherwise. The second approach is the simple and more correct solution, in order to produce a component that can be integrated with Lua. The implementation language in the first case would be almost entirely C, while in the second solution, Lua could be used for most parts of the system. The second approach has been followed.

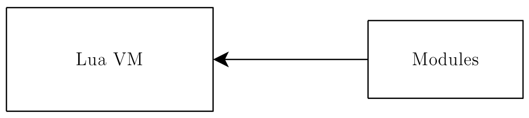

Figure 4.1: Lua and the implemented modules

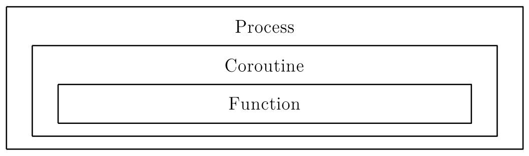

Figure 4.2: A system process is a coroutine which in turn is a function

The Lua component system is based on *modules* (Figure 4.1). Lua modules are provided by a package library that is responsible for building and loading of the modules. Lua modules can be written in Lua and C, thus it is a fully satisfactory solution for the target system. Ideally, the implemented system is broken in a set of submodules, that some of them can be loaded stand-alone, and other submodules can be left out if the functionality they provide is not needed.

4.2 Design

Design decisions on some of the most important aspects of the system will be analyzed in the following sections.

4.2.1 Processes

The processes in the system are implemented with Lua coroutines. Each process is actually a Lua coroutine, and a coroutine itself is a Lua function (Figure 4.2). This means that each process has a set of commands that comprise its functionality, and these commands are bound together inside a function.

A special case of usually short-lived processes is also used internally. These processes are targeted to accomplish different system function. Mostly distributed functions, that require communication and coordination between remote nodes, are implemented with these type of processes.

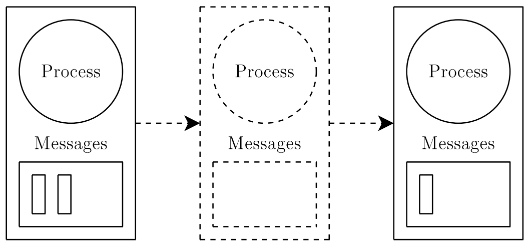

Figure 4.3: A process suspends and resumes when a message arrives

4.2.2 Scheduler

The scheduling policy used for the processes is still based on cooperative multithreading. Processes do not get preempted but instead they voluntarily pass control to another process. This suspending and resuming of processes is hidden under a higher level mechanism.

A process suspends its execution when waiting for a message to arrive and its mailbox is empty. On the other hand, a process resumes execution, when at least one messages has arrived in its mailbox (Figure 4.3). A simple round-robin type of scheduler is responsible for the role of resuming of suspended processes.

4.2.3 Messages and data

When processes exchange messages, one can wonder what is the Lua data type of the messages, or what data values can be included in messages. In this implementation, messages can be of any data type:

- Booleans
- Numbers
- Strings
- Tables
- Functions

The only exception to what data can be included are memory references. Furthermore, all of the data that can be sent inside a message, are marshalled on sent and unmarshalled on receive from the other end (Figure 4.4). This serialization and deserialization of the data is done transparently.

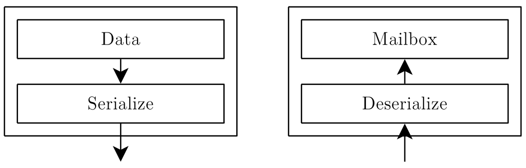

Figure 4.4: Data are serialized on sent and deserialized on receive

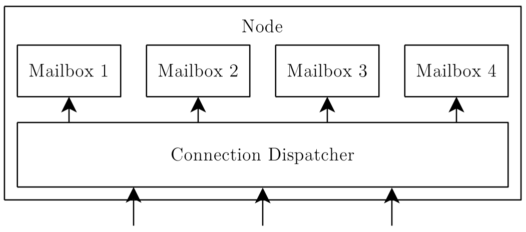

Figure 4.5: A network dispatcher handles all incoming connections

4.2.4 Interprocess communication

The interprocess communication model of the implemented system is based on an asynchronous socket handler (Figure 4.5). This results in a model of asynchronous communication between nodes and between processes, non-blocking socket calls and periodic polling.

This approach has the advantage of not requiring native threads, because socket calls return immediately even if no data are available. Lua's socket model is heading towards this model of asynchronous non-blocking calls, partly due to its multithreading model, so this approach integrates better with Lua.

4.3 Development

The system is divided in two main subsystems, organized as modules (Figure 4.6). The first of these modules is responsible for concurrency oriented programming. The second module uses the facilities provided by the concurrency module, and extends it by providing support for distributed oriented programming. As has been already mentioned, both modules are further comprised by a number of submodules. In addition, the port mapper daemon has been implemented as a stand-alone application.

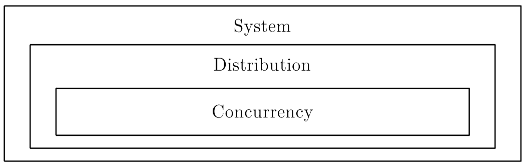

Figure 4.6: The architecture of the implemented system

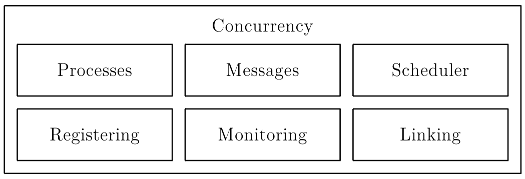

Figure 4.7: The architecture of the concurrency module

4.3.1 Concurrency

The concurrency module's (Figure 4.7) foremost role is that of handling the processes that make a concurrent system. Processes can be created and destroyed at any time. In addition, the module schedules when each of these processes gets to run. It has the ability to resume processes that went to idle state, for example when a process is waiting for some condition to change in order to be able to run.

The module also is responsible for maintaining the process mailboxes, one for each process. Messages that can be sent by any process, are delivered to the mailbox of the target process.

There is functionality for keeping a registry for registered names, instead of process unique identifiers. Aliases in the registry can be added or deleted.

The module also takes care of links between processes, along with monitoring of processes by other processes. It signals linked processes in case of error, and notifies the monitoring process about errors in the monitored process.

4.3.2 Distribution

The distribution module (Figure 4.8), which can be optionally loaded if needed, is based on the core functionality of the concurrency module. The first task that the distribution component completes is initialization of a node. Initialization is realized by setting a name for the node and by publishing the network port that the node is listening along with its name in the local port mapper.

The distribution module is responsible for establishing connections with

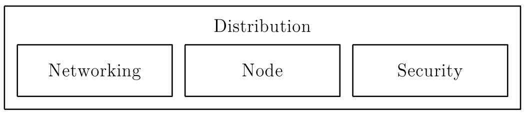

Figure 4.8: The architecture of the distribution module

remote nodes and closing these connections when this is necessary. Both when initializing connections and for every communication that takes place between nodes, an authentication scheme is applied. Any messages that are sent to remote processes are handled by the distribution module, in order to transparently forward them to the destination node.

The distribution module is able to implement monitoring and linking of remote processes. This is accomplished by the exchange of special control messages with the nodes that the monitored or linked processes reside in.

The distribution module has the ability to maintain a fully connected network of nodes. This has been used in order to implement a global registry for process aliases between all the nodes that are part of this virtual network.

4.3.3 Port mapper daemon

The port mapper daemon's functionality is rather simple. It features a simple database, that stores pairs of node names and network ports. The daemon listens on a standard port for connections from the nodes running on the local host. Thus, nodes can register or unregister their name and listening port, while remote nodes can find out the listening port of a specific node they want to connect to.

Chapter 5

Analysis

A presentation of the system, by the use of examples, will provide an overview of what the system can do in comparison with Erlang. This will play the role of an evaluation of the implementation in regards with the original requirements. Also, a discussion of the results, the possible uses of the system along with an brief investigation on multiprocessing will follow.

5.1 Evaluation

A number of simple examples will present some of the most essential properties of the implemented system, from process creation and message passing to distributed programming and error handling.

This presentation will cover only a part of the system's functionality, even though the major properties will be described. Nonetheless, an extensive and complete set of test cases has been written in order to test the full extend of the functionality that the system provides.

5.1.1 Creating processes

The basic function for creating processes is `spawn()`. The `spawn()` function takes at least one argument, the Lua function that will be the command set of the new process. Any additional arguments are passed directly to the process.

The following program demonstrates how processes can be created, by first defining the code that the process will execute, and then spawning the process itself. The program spawns a process that will print a string to the standard output as many times as it has been specified when creating the process (Figure 5.1).

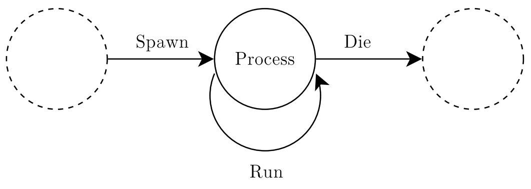

Figure 5.1: A process is created, executes and terminates

```
require 'concurrent'

function hello_world(times)
    for i = 1, times do print('hello world') end
    print('done')
end

concurrent.spawn(hello_world, 3)

concurrent.loop()
```

The output of the program when executed is:

```
hello world
hello world
hello world
done
```

First the implemented system is loaded:

```
require 'concurrent'
```

The function that the process will execute is defined next:

```
function hello_world(times)
    for i = 1, times do print('hello world') end
    print('done')
end
```

The process is created:

```
concurrent.spawn(hello_world, 3)
```

And last the infinite loop, in order for the system to run, is called:

```
concurrent.loop()
```

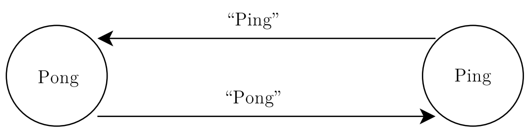

Figure 5.2: Two processes exchange messages

5.1.2 Exchanging messages

This second program makes use of message exchange between two processes. Specifically, the `send()` and `receive()` functions are used to send and receive messages. Also, the `self()` function is used to get the PID of the process that is calling the function. This program implements a process that sends messages and receives replies from another process (Figure 5.2).

```
require 'concurrent'

function pong()
    while true do
        local msg = concurrent.receive()
        if msg.body == 'finished' then
            break
        elseif msg.body == 'ping' then
            print('pong received ping')
            concurrent.send(msg.from, { body = 'pong' })
        end
    end
    print('pong finished')
end

function ping(n, pid)
    for i = 1, n do
        concurrent.send(pid, {
            from = concurrent.self(),
            body = 'ping'
        })
        local msg = concurrent.receive()
        if msg.body == 'pong' then
            print('ping received pong')
        end
    end
    concurrent.send(pid, {
        from = concurrent.self(),
        body = 'finished'
    })
    print('ping finished')
end

pid = concurrent.spawn(pong)
concurrent.spawn(ping, 3, pid)

concurrent.loop()
```

This is the output of the program:

```
pong received ping
ping received pong
pong received ping
ping received pong
pong received ping
ping received pong
pong finished
ping finished
```

The *pong* process is first created, and then, when the *ping* process is created, it is supplied with the PID of *pong*.

```
pid = concurrent.spawn(pong)
concurrent.spawn(ping, 3, pid)
```

The first message is sent by *ping*:

```
concurrent.send(pid, {
    from = concurrent.self(),
    body = 'ping'
})
```

The *pong* process waits for a message to become available in its mailbox, and saves it in a variable when it comes:

```
local msg = concurrent.receive()
```

The *pong* process replies back with a message:

```
concurrent.send(msg.from, { body = 'pong' })
```

The *pong* process may complete its operation, after the *ping* process has already finished and sent a notification message to *pong*.

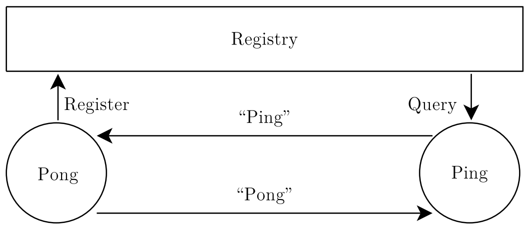

Figure 5.3: Two processes with registered names exchange messages

5.1.3 Registering process names

Instead of using PIDs as a destination when sending messages, a process can send messages by some process name. The `register()` function can be used to add a record in the process names registry. The previous example could be rewritten in order to take advantage of this feature (Figure 5.3).

```
require 'concurrent'

function pong()
    while true do
        local msg = concurrent.receive()
        if msg.body == 'finished' then
            break
        elseif msg.body == 'ping' then
            print('pong received ping')
            concurrent.send(msg.from, { body = 'pong' })
        end
    end
    print('pong finished')
end

function ping(n)
    for i = 1, n do
        concurrent.send('pong', {
            from = concurrent.self(),
            body = 'ping'
        })
        local msg = concurrent.receive()
        if msg.body == 'pong' then
            print('ping received pong')
        end
    end
    concurrent.send('pong', {
        from = concurrent.self(),
        body = 'finished'
    })
    print('ping finished')
end

pid = concurrent.spawn(pong)
concurrent.register('pong', pid)
concurrent.spawn(ping, 3)

concurrent.loop()
```

The output is the same as the previous example, and the only changes from the previous example is the destination that the *ping* process now sends the messages to:

```
concurrent.send('pong', {
    from = concurrent.self(),
    body = 'ping'
})
```

And:

```
concurrent.send('pong', {
    from = concurrent.self(),
    body = 'finished'
})
```

Also, the *pong* process now registers its PID:

```
concurrent.register('pong', pid)
```

Therefore, there is no reason to supply the PID of the *pong* process to the *ping* process.

5.1.4 Distributed message passing

Processes that are part of a different node can still communicate with the same message passing primitives. The only difference is that the destination process has to be referenced not only by its PID or process name, but also by the node that it is running on. If the two processes, *ping* and *pong* were

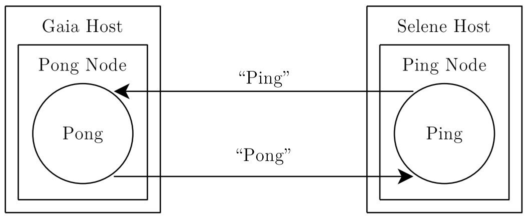

Figure 5.4: Two process on different nodes exchange messages

running on different nodes, and each of these nodes were in different network hosts, the previous example could be broken in two separate programs, one for each distributed process (Figure 5.4).

The code for the *pong* process:

```
require 'concurrent'

function pong()
    while true do
        local msg = concurrent.receive()
        if msg.body == 'finished' then
            break
        elseif msg.body == 'ping' then
            print('pong received ping')
            concurrent.send(msg.from, { body = 'pong' })
        end
    end
    print('pong finished')
end

concurrent.init('pong@gaia')

pid = concurrent.spawn(pong)

concurrent.register('pong', pid)
concurrent.loop()
concurrent.shutdown()
```

And the code for the *ping* process:

```
require 'concurrent'

function ping(n)
```

```
    for i = 1, n do
        concurrent.send({ 'pong', 'pong@gaia' }, {
            from = { concurrent.self(), concurrent.node() },
            body = 'ping'
        })
        local msg = concurrent.receive()
        if msg.body == 'pong' then
            print('ping received pong')
        end
    end
    concurrent.send({ 'pong', 'pong@gaia' }, {
        from = { concurrent.self(), concurrent.node() },
        body = 'finished'
    })
    print('ping finished')
end

concurrent.spawn(ping, 3)

concurrent.init('ping@selene')
concurrent.loop()
concurrent.shutdown()
```

The output for the *pong* process:

```
pong received ping
pong received ping
pong received ping
pong finished
```

And the output for the *ping* process:

```
ping received pong
ping received pong
ping received pong
ping finished
```

What has changed in this example is that now the runtime system is running in distributed or networked mode. Also, the port mapper daemon, one for each network host, has to be running.

The code that initializes the node that *pong* is running on:

```
concurrent.init('pong@gaia')
```

And the initialization for the *ping* node:

```
concurrent.init('ping@selene')
```

During initialization of the nodes a port is registered in the port mapper daemon. The nodes have to unregister this port when they stop running. This is done with the following command:

```
concurrent.shutdown()
```

Apart from the initialization and finalization of the nodes, the only other change is that processes are denoted both by their PID or process name and the name of the node they are running on. So when the *ping* process sends a message to the *pong* process it does it like this:

```
concurrent.send({ 'pong', 'pong@gaia' }, {
    from = { concurrent.self(), concurrent.node() },
    body = 'ping'
})
```

And later on again:

```
concurrent.send({ 'pong', 'pong@gaia' }, {
    from = { concurrent.self(), concurrent.node() },
    body = 'finished'
})
```

Something new in the above code is also the `node()` function which returns the name of the node that the executing process is running on.

5.1.5 Handling error

Another interesting property of the system is how to handle errors in processes. Here comes the notion of linked processes, where two processes are bound together, and when one of them dies, the other is taken down, too. The `link()` function provides this functionality (Figure 5.5).

```
require 'concurrent'

function ping(n, pid)
    concurrent.link(pid)
```

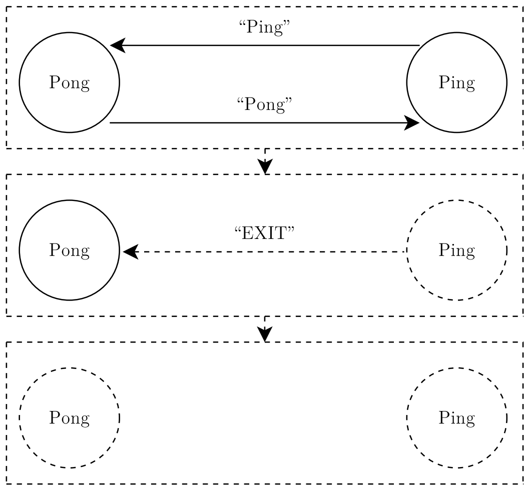

Figure 5.5: One of the two linked processes exits

```
    for i = 1, n do
        concurrent.send(pid, {
            from = concurrent.self(),
            body = 'ping'
        })
        local msg = concurrent.receive()
        if msg.body == 'pong' then
            print('ping received pong')
        end
    end
    print('ping finished')
    concurrent.exit('finished')
end

function pong()
    while true do
        local msg = concurrent.receive()
        if msg.body == 'ping' then
            print('pong received ping')
            concurrent.send(msg.from, { body = 'pong' })
        end
    end
    print('pong finished')
end
```

```
pid = concurrent.spawn(pong)
concurrent.spawn(ping, 3, pid)

concurrent.loop()
```

The output of this example is:

```
pong received ping
ping received pong
pong received ping
ping received pong
pong received ping
ping received pong
ping finished
```

The *pong* process doesn't ever reach its last line, because when the *ping* process dies, *pong* dies immediately with it.

The new code in this example is the linking of the two processes:

```
concurrent.link(pid)
```

Along with the use of the `exit()` function, in order to cause an abnormal exit of the *ping* process:

```
concurrent.exit('finished')
```

There is also the capability to trap the exit signal, and instead of dying, a special message is received by the linked process:

```
require 'concurrent'

concurrent.setoption('trapexit', true)

function pong()
    while true do
        local msg = concurrent.receive()
        if msg.signal == 'EXIT' then
            break
        elseif msg.body == 'ping' then
            print('pong received ping')
            concurrent.send(msg.from, { body = 'pong' })
        end
    end
    print('pong finished')
end

function ping(n, pid)
    concurrent.link(pid)
    for i = 1, n do
        concurrent.send(pid, {
            from = concurrent.self(),
            body = 'ping'
        })
        local msg = concurrent.receive()
        if msg.body == 'pong' then
            print('ping received pong')
        end
    end
    print('ping finished')
    concurrent.exit('finished')
end

pid = concurrent.spawn(pong)
concurrent.spawn(ping, 3, pid)

concurrent.loop()
```

The output of this example is:

```
pong received ping
ping received pong
pong received ping
ping received pong
pong received ping
ping received pong
ping finished
pong finished
```

The `setoption()` function changes the behaviour of process linking, specifically the `trapexit` option:

```
concurrent.setoption('trapexit', true)
```

While the *pong* process checks for an exit message with:

```
    if msg.signal == 'EXIT' then
        break
```

5.2 Discussion

The presentation of the system provides a satisfactory overview of the resulting system, which will be evaluated here. A discussion on possible uses and some further ideas will also follow.

5.2.1 Results

Several important properties of the system have been showcased through examples. It is obvious that the desired functionality that the requirements defined has been achieved. All of the goals that have been originally set, have been finally met.

Process creation and the message passing primitives are straight-forward and flexible, similar to what Erlang provides. One difference on the interface is that instead of the matching of messages during receive that is used Erlang, tables with named fields are preferred in order to mark parts of the messages. This approach is closer to what scripting languages such as Lua support.

Distributed programming is supported transparently and in the same way as concurrent programming. Processes can communicate regardless of their location. The functions provided for this purpose have the same syntax, and every necessary action to be taken by the system in order to support distribution is done transparently. Several mechanisms for error handling are also available for the application programmer.

And last but not least, the final system is strongly integrated to Lua. It can be loaded and become ready to be used from any Lua program. Modularization and good portability also help in the direction of developing cross-platform distributed applications.

5.2.2 Uses

The resulting system could be used as the base for development of diverse applications. But some of the more obvious uses could be:

- A higher level mechanism instead of the lower level coroutines

  Coroutines are the concurrency mechanism provided by Lua. They are based on cooperative multithreading and so the developer has to manually transfer control to each coroutine, replacing the role of a scheduler. Instead, with the implemented system all this is hidden under processes that just send messages to each other and based on this communication, scheduling is automatically taken care of. Also, the developer does not have to program based on what global data the processes have to share, but focuses on the functionality of each process and whenever any data are needed by another process, they are just simply passed to it.

- A development platform for distributed systems and general purpose networked systems

  While one can do socket programming in Lua, in many cases when developing distributed systems, the message passing mechanism is simpler but still adequate as an alternative. The message passing system is built upon the socket interface and can be considered a higher level mechanism.

- A prototype-based programming solution in the context of distributed programming

  Prototype-based programming is becoming more and more widespread, and interpreted languages, due to their dynamic nature, can fit this role well. The ability to exchange message that contain code, can provide an environment for experimenting on prototype-based programming in distributed systems.

- An infrastructure for the development of applications with code mobility characteristics

  The system can also be used to create applications that support weak code mobility. As mentioned before, code can be easily transferred across the network. In addition, it would be also possible to extend the system in order to support strong code mobility, where whole processes could be transferred. This would include use of more advanced serialization and deserialization routines.

- An infrastructure for the development of agent-oriented programming solutions

  It would be also possible to support mobile agents, either with weak code mobility and by programming the restoring of the agent's state explicitly, or better yet, by first extending the system to support strong code mobility.

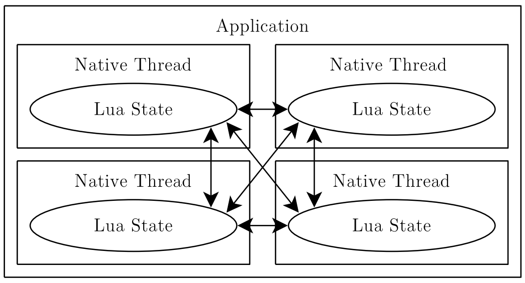

Figure 5.6: Lua VMs running on different native threads

5.2.3 Multiprocessing

Another aspect of the system that has not been investigated at all, is that while it brings a model for concurrency to the Lua programming language, it doesn't address parallelism. This aspect will be discussed here from a theoretical viewpoint.

One of Lua's distinguished properties is that it is targeted to be embedded into other applications. This embedding is done in the form of a reentrant interpreter, the Lua VM, that is represented by an opaque structure, the *Lua state*.

Having all these in mind, a way to implement SMP support, would be by having multiple interpreters embedded inside an application. Each of these interpreters would run in its own separate native thread. The interpreters would then communicate by using the message passing system that has been implemented and each interpreter would be represented by a node (Figure 5.6).

Of course, one might ask that interpreters could communicate by sharing data through the application that embeds them. While that is true, and it would have its advantages as a solution, as has been explained in previous chapters, this approach opens a whole world of complexity and this makes it extremely error-prone. Instead, the aforementioned solution is suggested as a viable alternative, that could make it possible to do multiprocessing programming simpler.

Chapter 6

Conclusions

General concluding remarks and an insight into future work and hints for further analysis will complete this report.

6.1 Conclusion

This thesis began with questions related to the message passing concurrency and the benefits of this model in modern scripting languages. The answers to these questions, required solutions to some of the architectural design problems, problems which have been solved during this thesis.

The implemented system required some kind of processes or independent threads of execution, something that has been addressed by the use of coroutines and cooperative multithreading. Messages are stored in message queues, and thus when messages are sent or received, they are appended or removed from this queues. Messages can be of any data type that Lua supports, and this is accomplished by serialization and deserialization of data. Distribution is realized transparently by the use of nodes that communicate through a virtual network that they form, and interprocess communication is implemented by the use of asynchronous non-blocking sockets. Finally, the implementation is based on Lua modules in order for it to be strongly integrated with Lua.

A presentation of the aforementioned design decisions of the system, by the use of examples, provided an inner look on the capabilities of the system. Some of the benefits and possible uses of such an architecture have been also suggested, but maybe most importantly, a theoretical approach to the problem of multiprocessing has been included. This approach is based on the notion of multiple runtimes running on different native threads, but instead of communicating by sharing data, the race conditions problem is solved by the utilization of the message passing system.

6.2 Future work

The implementation that has been presented in this report is first of all a prototype. This does not mean that it cannot be used in production but that mainly it is a proof of concept. One of the things that has been left undone and should be subject of more work in the future are optimizations and stress testing. Some of the areas that would first benefit from optimizations would be the process scheduler and the message mailbox subsystem.

From there on it would be also very important for different uses of this system to be investigated. Some of the possible uses of the system have already been suggested, and it is through extensive testing that the system would mature even more and the extend of its capabilities would become more evident. This more thorough analysis and evaluation of the system will be possible after it has been used as a component for other projects.

Another aspect of the system that can prove very important in the future is multiprocessing support. Although the system can already be used in order to achieve SMP inside an application, with an approach that has been described previously, what would be more interesting for the future would be to generalize this concept. This means that somehow programs should benefit from SMP support available by the hardware without requiring from the programmer to do anything explicitly. There are two approaches that will be suggested here.

The first approach is similar to what is done in Erlang, where multiple process schedulers are running on different native threads, and communicate by sharing data. In this case the application programmer will not have to worry about race conditions, because they will be taken care just once inside the interpreter utility.

The second approach is based on the idea of *multiple VMs*. This concept is similar to the multiple interpreters inside an application approach discussed before, but in this case a Lua interpreter utility is the application that will interpret any Lua code and parallelize process execution by distributing processes to the interpreters.

Bibliography

[1] J. K. Ousterhout, *Scripting: Higher-Level Programming for the 21st Century*, IEEE Computer, Vol 31, No. 3, p. 23-30, 1998.

[2] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, *Lua - an extensible extension language*, Software: Practice & Experience 26 #6 (1996) 635-652.

[3] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, *The implementation of Lua 5.0*, Journal of Universal Computer Science 11 #7 (2005) 1159-1176.

[4] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, *The evolution of Lua*, Proceedings of ACM HOPL III (2007).

[5] Edward A. Lee, *The Problem With Threads*, IEEE Computer, vol. 36, no. 5, May 2006, pp. 33-42.

[6] Ana Lcia de Moura, Noemi Rodriguez, Roberto Ierusalimschy, *Coroutines in Lua*, Journal of Universal Computer Science 10 #7 (2004) 910-925.

[7] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams, *Concurrent Programming in Erlang*, Prentice-Hall, 1996.

[8] Joe Armstrong, *Concurrency Oriented Programming in Erlang*, Distributed Systems Laboratory, Swedish Institute of Computer Science, 2003.

[9] Joe Armstrong, *Making reliable distributed systems in the presence of software errors*, PhD thesis, Royal Institute of Technology, Swedish Institute of Computer Science (SICS), Stockholm, December 2003.

[10] Cristina Ururahy and Noemi Rodriguez, *ALua: an event-driven communication mechanism for parallel and distributed programming*, Proceedings of ISCA 12th International Conference on Parallel and Distributed

Computing Systems (PDCS-99), Fort Lauderdale, USA, August 1999, pages 108-113.

[11] Alsio Leoni Pfeifer, Cristina Ururahy, Noemi Rodriguez, Roberto Ierusalimschy, *An event-driven system for distributed multimedia applications*, Proceedings of DEBS'02 - International Workshop on Distributed Event-Based Systems (held in conjunction with IEEE ICDCS 2002), Vienna, Austria, July 2002, pages 583-584.

[12] Cristina Ururahy, Noemi Rodriguez, Roberto Ierusalimschy, *ALua: flexibility for parallel programming*, Computer Languages 28 #2 (2002) 155-180.

[13] Steve Dekorte, *Io: a small programming language*, OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (2005), 166-167.

[14] Christian Tismer, *Continuations and Stackless Python*, In Proceedings of the 8th International Python Conference, Arlington, VA, 2000.

Appendix A

User's manual

All of the functionality of the implemented system becomes available by loading the "concurrent" module. The following functions are exported by the module.

Processes

spawn(body, ...)

Creates a process which will execute the `body` function. Any extra arguments can be passed to the executing function. The PID of the new process is returned. In case of error `nil` and an error message are returned.

spawn(node, body, ...)

Creates a process in a remote `node` which is a string in the format `'node-name@hostname'` and the new process will execute the `body` function. The PID of the new process is returned. In case of error `nil` and an error message are returned.

self()

Returns the PID of the calling process.

isprocessalive(process)

Checks if the `process`, which can be specified by PID or by its registered string name, is alive. Returns `true` if the process is alive, and `false` otherwise.

exit(reason)

Exits abnormally the calling process with the specified `reason` string as a cause of exit.

Messages

receive([timeout])

Receives the next message in the mailbox of the calling process. If the mailbox is empty it waits indefinitely for a message to arrive, unless a `timeout` number in milliseconds is specified. A message of any type, that depends on what was sent, is returned.

send(process, message)

Sends to the destination `process` a `message` which can be one of: boolean, number, string, table, function. Returns `true` if the message was send successfully, and `false` if not.

Scheduling

sleep(time)

Suspends implicitly the calling process for the specified `time`, the number of milliseconds.

loop([timeout])

Calls the system's infinite loop which executes the process scheduler until all the processes have terminated, or unless the specified `timeout`, the number of milliseconds, has expired.

interrupt()

Interrupts the infinite loop of the process scheduler.

step([timeout])

Executes one step of the process scheduler unless the specified `timeout`, the number of milliseconds, has expired.

tick()

Forwards the system's clock by one tick.

Options

setoption(key, value)

Sets the `key` string option to the specified `value`, the type of which depends on the option.

getoption(key)

Returns the value of the `key` string option.

Node

init(node)

Makes the runtime system a distributed node. The first argument is the name string of the `node`, which can be either in `'nodename'` or `'nodename@host-name'` format.

If the `'shortnames'` option is set to `true`, then short names are used instead of fully qualified domain names. If the `'connectall'` option is set to `false`, then a fully connected virtual network between the nodes will not be maintained.

shutdown()

Makes the runtime system stop being a distributed node.

node()

Returns the name of the node the calling process is running on.

nodes()

Returns a table with the nodes that the node the calling process is running on is connected to.

isalive(node)

Returns `true` if the specified `node` is alive, and `false` otherwise.

monitornode(node)

The calling process starts monitoring the specified `node`, which is a string of the format `'nodename@hostname'`.

demonitornode(node)

The calling process stops monitoring the specified `node`, which is a string of the format `'nodename@hostname'`.

Security

setcookie(secret)

Sets the pre-shared `secret` key, a string, also known as the *magic cookie*, that will be used for node authentication.

getcookie()

Returns the pre-shared secret key, also known as the *magic cookie*, that is being used for node authentication.

Registering

register(name, pid)

Registers the `name` string for the given process `pid`.

unregister(name)

Unregisters the process with the `name` string.

whereis(name)

Returns the PID of the process with the registered `name` string.

registered()

Returns a table with all the registered process names.

Linking

link(process)

The calling process gets linked with the specified `process`, which can be either a PID, a registered name, or a remote process. A remote process is a table with two elements, the remote process PID or registered name and the node's name in the format `'nodename@hostname'`.

The `'trapexit'` option can be set to `true`, if exit signals between linked processes are to be trapped.

unlink(process)

The calling process gets unlinked with the specified `process`, which can be either a PID, a registered name, or a remote process. A remote process is a table with two elements, the remote process PID or registered name and the node's name in the format `'nodename@hostname'`.

spawnlink(body, ...)

Creates a process which will execute the `body` function and the calling function also gets linked to the new process. Any extra arguments can be passed to the executing function. The PID of the new process is returned. In case of error `nil` and an error message are returned.

The `'trapexit'` option can be set to `true`, if exit signals between linked processes are to be trapped.

spawnlink(node, body, ...)

Creates a process in a remote `node` which is a string in the format `'nodename@hostname'`, the new process will execute the `body` function, and also the calling process gets linked to the newly created process. The PID of the new process is returned. In case of error `nil` and an error message are returned.

The `'trapexit'` option can set to `true`, if exit signals between linked processes are to be trapped.

Monitoring

monitor(process)

The calling process starts monitoring the specified `process`, which can be either a PID, a registered name, or a remote process. A remote process is a table with two elements, the remote process PID or registered name and the node's name in the format `'nodename@hostname'`.

demonitor(process)

The calling process stops monitoring the specified `process`, which can be either a PID, a registered name, or a remote process. A remote process is a table with two elements, the remote process PID or registered name and the node's name in the format `'nodename@hostname'`.

spawnmonitor(body, ...)

Creates a process which will execute the `body` function and the calling function also starts monitoring the new process. Any extra arguments can be passed to the executing function. The PID of the new process is returned. In case of error `nil` and an error message are returned.

spawnmonitor(node, body, ...)

Creates a process in a remote `node` which is a string in the format `'nodename@hostname'`, the new process will execute the `body` function, and also the calling process starts monitoring the newly created process. The PID of the new process is returned. In case of error `nil` and an error message are returned.