Computer Languages, Systems & Structures 36 (2010) 142 – 157

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

# Flexibility and coordination in event-based, loosely coupled, distributed systems

B. Silvestre[a], S. Rossetto[c], N. Rodriguez[a,*], J.-P. Briot[a,b]

[a]*Depto Informática, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), 22451-900, Brazil*
[b]*LIP6-Paris6, France*
[c]*Universidade Federal Fluminense (UFF), Brazil*

ARTICLE INFO

*Article history:*
Received 9 July 2008
Received in revised form
28 April 2009
Accepted 10 June 2009

*Keywords:*
Coordination
Synchronization
Distributed programming
Interpreted language
Closures
Coroutines
First-class functions
Asynchronous remote procedure call

ABSTRACT

The scale and diversity of interactions in current wide-area distributed programming environments, specially in Internet-based applications, point to the fact that there is no single solution for coordinating distributed applications. Instead, what is needed is the ability to easily build and combine different coordination abstractions. In this paper, we discuss the role of some language features, such as first-class function values, closures, and coroutines, in allowing different coordination mechanisms to be constructed out of a small set of communication primitives, and to be easily mixed and combined. Using the Lua programming language, we define a basic asynchronous primitive, which allows programming in a direct event-driven style with the syntax of function calls, and, based on this primitive, we build different well-known coordination abstractions for distributed computing.

## 1. Introduction

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

In this setting, a lot of attention has been directed to event-driven programming. As opposed to conventional programming, in which a program is written as a sequence of actions, in event-based programming the developer writes a set of event handlers, which are activated by incoming events. However, understanding a program that is coded as a series of responses to different events is not easy when the number of interacting peers is large. One single process may, at any moment, be interacting with dozens of other processes, and each of these interactions may require its own state information. This highlights the need of abstractions to coordinate interacting processes.

The concept of a coordination language as a tool to describe the interactions among the parts in a distributed program was much discussed in the nineties [22,40]. At the time, most discussion focused on the idea of using a pre-defined set of coordination primitives, such as Linda's [17] tuple space manipulation operations, to define the communication and synchronization of the application. Coordination models proposed at that time were often focused on tightly coupled application models. Typical distributed applications today rely on complex communication patterns among machines placed in different

* Corresponding author. Tel.: +55 21 3527 1500; fax: +55 21 3527 1530.
*E-mail address:* noemi@inf.puc-rio.br (N. Rodriguez).

geographical and administrative domains. To deal with the uncertainty of communication across domains, communication abstractions must support loosely coupled interactions, often supplied by event-oriented application models. Many interactions are in fact naturally captured by the event-based paradigm. However, many applications include interactions which would be best captured by other communication abstractions. (Imagine, for instance, an application for news dissemination based on publish–subscribe. Enrollment of new users may require that they be authenticated or registered, and this may best be done using RPC.)

It is important to allow the programmer the flexibility of choosing the most appropriate abstraction for each interaction.

Primitives for communication and synchronization have classically been offered either through special-purpose programming languages or through libraries. Languages that are designed from scratch to support distribution usually provide a consistent programming model, typically because language architects designed them with a chosen communication pattern in mind. This makes them easy to master for the programming tasks for which they were imagined, but can lead to little maleability when dealing with new interaction patterns. Communication libraries for conventional languages, in contrast, can be freely combined in a single application, allowing the programmer to choose the preferred pattern for each set of interactions. On the other hand, gaps between the host programming language and the model the libraries implement often result in awkward APIs.

These classical approaches must be reviewed in light of the requirements for flexibility discussed above. No single predefined set of interaction patterns will be appropriate for designing wide-area and large-scale applications. What we need are environments in which different abstractions and coordination patterns can be built and combined. The programmer should be free to experiment and combine abstractions without having to resort to the relatively low-level programming interfaces traditionally offered by libraries.

We believe programming language features could make a significant contribution to this issue. Using appropriate language constructs, it is possible to build an arbitrary number of coordination mechanisms from a small set of primitives. Furthermore, these mechanisms can be smoothly integrated with the programming language, even if they are implemented as extensions. This is not a new claim. In [14], the authors discuss how reflection can help integrate protocol libraries intimately within a programming language or system, providing a bridge between the library and language approaches. Besides reflection, there are a number of other programming language features that can help in this integration, creating environments in which different coordination techniques can be used and combined to compose new mechanisms. So, instead of looking for the specific coordination mechanism which will be better than others for event-driven programs, we should concentrate on such environments, allowing programmers to create and combine coordination constructs easily.

To evaluate this claim, over the last few years, we have developed a series of coordination libraries in Lua [31], a dynamically typed interpreted language with a procedural syntax but with several functional characteristics. In this paper, we discuss the results of this development and the role of language features in allowing these libraries to be easily combined. We emphasize the role of first-class functions, closures, and coroutines. The distributed environment which we analyze is composed by several Lua processes. We also consider that each of these processes has a single-threaded structure. This has considerable weight in our discussion and is a decision based on the several complexity issues associated to preemptive multithreading [39].

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

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

## 2. Event-driven distributed programming in Lua

Over the last years, we have been investigating the advantages and limitations of creating distributed programs with a simple event-driven model, based on the Lua programming language [31,29,30]. Lua is an interpreted programming language designed to be used in conjunction with C. It has a simple Pascal-like syntax and a set of functional features. Lua implements dynamic typing. Types are associated to values, not to variables or formal arguments. Functions are first-class values and Lua provides lexical scoping and closures. Lua's main data-structuring facility is the *table* type. Tables implement associative arrays, i.e., arrays that can be indexed with any value in the language. Tables are used to represent ordinary arrays, records, queues, and other data structures. Tables are also used to represent modules, packages, and objects.

The Lua programming language has no embedded support for distributed programming. ALua [50] is our basic library for creating distributed event-based applications in Lua. ALua applications are composed of processes that communicate through the network using asynchronous messages. Processes use the `alua.send` primitive to transmit the messages.

An important characteristic of ALua is that it treats each message as an atomic chunk of code. It handles each event to completion before starting the next one. This avoids race conditions leading to inconsistencies in the internal state of the process. However, it is important that the code in a message does not contain blocking calls, as it would block the whole process, precluding it from handling new messages.

```
 1 -- Initialize global variables
 2 acc = 0
 3 repl = 0
 4 expected = 0
 5
 6 function avrg (ret)
 7   repl = repl + 1
 8   acc = acc + ret
 9   if (repl == expected) then
10     print ("Average: ", acc/repl)
11   end
12 end
13
14 function request(peers)
15   -- Save the number of peers in the array
16   expected = peers.n
17   -- Create string to be executed by remote peers
18   local req = "alua.send (" .. alua.id ..
19                               ", ' avrg(' .. valueHere .. ')' )"
20   -- Request the remote values
21   for i = 1, expected do
22     alua.send (peers[i], req)
23   end
24 end
```

**Fig. 1.** ALua code for evaluating average returned value.

The ALua basic programming model, in which chunks of code are sent as messages and executed upon receipt, is very flexible, and can be used to construct different interaction paradigms, as discussed in [50]. However, programming distributed applications directly on this programming interface keeps the programmer at a very low level, handling strings containing chunks of code.

As an example, consider a simple situation in which a process wants to evaluate an average of values retrieved from a list of peers. Fig. 1 presents the ALua code for this example. Function `request` receives as an argument a list of peers, `peers`, stored as a Lua table, and sends the string stored in variable `req` to each process in this list. This string contains code that, when executed in the destination process, will make it send back to the current process a new string, containing a call to function `avrg` with the current value of variable `valueHere` in the remote process passed as argument.

Some remarks are in place about string manipulation in Lua. Both quotes and double quotes may be used as string delimiters. The `..` is the string concatenation operator in Lua. So, supposing that the value of `alua.id` (a variable containing addressing information for the current process) is `‘139.82.20.1:2100:1’`, the resulting string in lines 18 and 19 will be:

```
‘‘alua.send (‘139.82.20.1:2100:1’, ‘avrg(‘.. valueHere .. ’)’)’’
```

After receiving this message, each peer process will execute:

```
alua.send (‘139.82.20.1:2100:1’, ‘avrg(‘.. valueHere .. ’)’)
```

which will make it evaluate the current value of `valueHere` in its environment, creating a new string (such as `‘avrg(3)’`) which will be sent back to the original caller.

This example illustrates a programming model that is typical of event-oriented programming. After sending the requests, global variables `acc`, `repl`, and `expected` maintain the state of the application, which behaves as a state machine, moving between states upon handling each new event.

## 3. Asynchronous RPC

Although this basic message-oriented event-driven programming model is powerful, it can be quite error-prone and hard to use. Programmers need tools that allow them to model high-level interaction patterns. This is where programming abstractions come into play. To allow the programmer to deal with higher-level concepts, we have implemented several communication libraries over the last few years, providing support for tuple spaces [33], publish–subscribe [43], and remote procedure call [42], among others. In this work, we use remote procedure call as the basic communication mechanism, and so we discuss it next.

```
-- Initialize global variables
acc = 0
repl = 0
expected = 0

-- Callback function
function avrg (ret)
  repl = repl + 1
  acc = acc + ret
  if (repl == expected) then
    print ("Average: ", acc/repl)
  end
end

function request(peers)
  -- Save the number of peers in the array
  expected = peers.n
  -- Request the remote values
  for i = 1, expected do
    -- Create asynchronous remote function
    local sendRequest = rpc.async(peers[i], "getCurrentValue", avrg)
    -- Remote invocation
    sendRequest("valueHere")
  end
end
```

**Fig. 2.** Using `rpc.async`.

The RPC abstraction has been adopted in systems ranging from CORBA [46] to .NET [19] and SOAP [38]. From its inception, however, critiques to the paradigm were made [49,11]. They mostly discuss the imposition of a synchronous structure on the client application and the difficulty of matching RPC with fault tolerance, partly due to its point-to-point architecture. These critiques gain further importance in the context of wide-area networks. However, the familiarity that programmers have with the model must not be ignored. If the synchronous nature of the original proposal is somewhat incompatible with the loose coupling we need in wide-area distribution, we can resort to an asynchronous RPC model. Asynchronous invocations have been long discussed as an alternative [3], but the fact is that they are not comfortable to use in traditional sequential programs, specially when return values are involved. (Because the program may be in any arbitrary point of execution when these return values become available, they must be handled either in a new thread of execution or though the use of special primitives.) When the program is event-based, however, asynchronous invocations are natural, and can be associated to callback functions to be executed upon the completion of the remote invocation.

Our remote procedure mechanism [42], provided by the `rpc` library, explores this idea, associating asynchronous invocations with callback functions over an event-driven model. The basic execution model remains the one we described in the last section with a process handling each incoming message at a time, with the difference that now messages are function invocations.

To provide the same flexibility as we have with normal function values in Lua, the `rpc.async` primitive does not directly implement the invocation, but, instead, returns a function that calls the remote method (with its appropriate arguments). As an example, consider that a process offers remote access to values of its global variables through a function called `getCurrentValue`, which sends the value of a global variable to the process specified as its first argument. Fig. 2 illustrates the use of `rpc.async` to implement the same example of Section 2, which evaluates the average of values provided by a set of peers.

Mandatory parameters for `rpc.async` are the remote process (`peers[i]`) and the remote function name (`''getCurrentValue''`). The third argument is an optional callback function (`avrg`). In Fig. 2, the function returned by `rpc.async` is stored in variable `sendRequest` (remember that `rpc.async` does not invoke the remote function, but instead, creates a function that invokes it). When function `sendRequest` is called (with `''valueHere''` as the single argument to `getCurrentValue`, so that it can return the required value), control returns immediately to the caller. At some later point, when the program returns to the event loop and receives the result of the remote function, callback function `avrg` will be invoked, with the received value as an argument. `avrg` then proceeds as before, keeping current state in global variables `repl`, `acc`, and `expected`.

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

Having functions as first-class values means they can be passed as arguments or be used as return values from other functions. A *closure* is a semantic concept combining a function with a set of data that is neither local to this function nor global to the

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

**Fig. 3.** Implementation of `rpc.async`.

program. These are typically variables defined in the new function's lexical scope. Closures can be used to hide state, implement higher-order functions and defer evaluation. With these two mechanisms, a function can return a nested function, and the new function has full access to variables and arguments from the enclosing function.

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

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

### 3.1. *Maintaining state in event-driven programs*

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

The event-driven programming model is convenient in that it mirrors asynchronous interactions among processes. However, because we have only one execution line, whenever a process needs to receive an event before continuing execution, the current action must be finalized to wait for the message (that is, the process must return to the event loop to be able to handle the next message). Moreover, to maintain the interactivity, the system must make sure that no message handler takes too long to execute. So, event handlers must run quickly, i.e., long tasks must also be broken into small pieces, between which the system saves the current state and returns to the main loop. In order to do that, the event handler can post a request and schedule the remainder of the current computation to be executed later, as explored in [54]. Typically, to maintain state information between the function that is being executed and the one that will be executed later, the programmer must resort to global variables, because the current locals will not exist any more at this future point. This process, illustrated by the use of global variables `repl`, `acc`, and `expected` in our example, and referred by Adya et al. as *stack ripping* [1], is one of the main difficulties for developing applications using the event-driven programming style [53].

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

To illustrate this idea, Fig. 4 presents once again the example of evaluating an average value, now using a closure. `rpc.async` is again used to build asynchronous requests to take values in each remote process and the `avrg` function is defined as the callback function. The main difference is that `avrg` is a closure of `request`, and it is thus able to keep the values of (local) variables `acc`, `repl`, and `expected` (used to compute the average) even when the process returns to the main event loop.

The pattern used in this example can be implemented in any system supporting closures, and is useful in a number of situations. Because closures capture state, they are able to preserve the relevant part of the activation stack from the moment in which the asynchronous invocation is issued until the activation of the callback.

```
function request(peers)
  local acc = 0
  local repl = 0
  local expected = peers.n

  local function avrg (ret)
    repl = repl + 1
    acc = acc + ret
    if (repl == expected) then
      print ("Average: ", acc/repl)
    end
  end

  -- Request the remote values
  for i = 1, expected do
    -- Create the asynchronous function
    local sendRequest = rpc.async(peers[i], "getCurrentValue", avrg)
    -- Invoke the remote function
    sendRequest("valueHere")
  end
end
```

**Fig. 4.** Exploring the closure mechanism do avoid *stack ripping*.

## 4. Coordinating concurrent activities

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

Gelernter and Carriero [22] discuss the advantages of viewing communication and synchronization primitives as means of *coordinating* a concurrent or distributed application. In this section, we adopt this approach and discuss how different coordination abstractions can be provided by libraries that can be combined, either as building blocks, to create further abstractions, or simply as alternative to be used inside an application, as needed. We again focus on language features that allow libraries with these abstractions to be seamlessly integrated into the language.

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

The second class of abstractions we discuss is the one related to classical synchronization among concurrent processes, for mutual exclusion and cooperation [4]. Even if, in our model, fine-grained synchronization problems, such as interleaved accesses to global variables, are avoided, we can still have problems occurring at a coarser granularity. Subsequent calls to one process may need to occur with the guarantee that no events were handled between the two (for instance, to guarantee an atomic view of a set of operations). Also, because we are in a distributed setting, we may need to synchronize actions occurring at different processes. Sections 4.3 and 4.4 discuss support for classical synchronization.

### 4.1. Synchronous RPC

With asynchronous invocations, the programmer must turn control flow upside down, using callback functions to code the *continuation* of the computation after the results of the invocation are available. This directly reflects the event-driven nature of a program, but may not be the best model for the programmer to work with. In this section, we discuss function `rpc.sync`, that creates functions that make synchronous calls to other processes over the same asynchronous communication model. Function `rpc.sync`, like `rpc.async`, receives as parameters the process identification and the remote function name. Because

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

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

**Fig. 5.** Example using `rpc.sync`.

it is synchronous, the callback parameter does not make sense (in fact, a callback which resumes the current computation will be implicitly built by `rpc.sync`).

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

Because ALua runs on a single thread, suspending the execution during a synchronous call would block the ALua event loop as well. As a consequence, a process would not receive new requests until the invocation is completed. One solution to this is to introduce a multitasking mechanism, where computation can be suspended and resumed later in the same thread of control. The most popular mechanism for multitasking is preemptive multithreading—as offered, for instance, by Posix threads [15]. Preemptive multithreading guarantees automatic scheduling of the different active threads by suspending (*preempting*) running threads at arbitrary points of execution—typically after a *time slice* has elapsed—and scheduling other ready threads. Context switching, however, has a cost, and in preemptive multithreading this cost must be paid by the application even if response time and fairness are not important for it. Besides, as a result of the suspension of control at arbitrary points of execution, actions in different threads may be interleaved also in arbitrary ways, some of which may lead to undesirable interferences. To deal with this, programmers typically must resort to synchronization primitives originally designed for programming operating systems, like mutexes and semaphores. These low-level constructs often make programs hard to understand and to debug.

In order to avoid the complexity added by preemptive multithreading, we rely on yet another language mechanism for the implementation of `rpc.sync`: the cooperative multitasking facility offered by Lua *coroutines*. A coroutine is similar to a thread in that it maintains its own execution stack, local variables and a program pointer. The main difference is that the transfer of control among coroutines is explicit, i.e., one must issue an explicit control primitive for control to be transferred to any other coroutine and the execution of a coroutine is only suspended when that coroutine yields control. In essence, coroutines are concurrent processes where the control transfer is completely described in the application algorithm [5]. This is interesting because it allows applications to maintain different execution lines while avoiding the complexities of race conditions. Besides, the control transfer points are pre-defined in the application, so context switching occurs only when it is in fact needed, minimizing computational cost.

However, one disadvantage of cooperative multitasking is that the management of scheduling is left to the programmer, in contrast to the automatic management in preemptive multithreading. Even if this is a serious drawback only for applications which need response time guarantees, it can be cumbersome for the programmer in any case. To avoid this burden, one alternative is to encapsulate control transfers in wrappers for possibly blocking operations. This is what we do in the implementation of `rpc.sync`. Each new computation is handled in a new coroutine and, when a synchronous call is performed, the current coroutine is suspended and execution flow returns to the ALua loop.

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

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

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

## 4.2. Futures

As yet another example of building communication abstractions, we can also implement support for *futures* [34]. In some cases, the programmer may know, at a certain point of execution, that he needs to schedule a computation whose result will be needed only later. Futures allow the programmer to synchronize actions between processes in a looser relationship. This

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

     -- invoke the remote function
     local aux = rpc.async(proc, func, callback)
     aux(...)

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

**Fig. 6.** Implementation of `rpc.sync`.

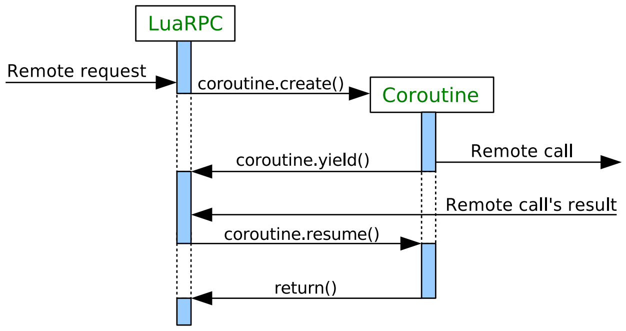

**Fig. 7.** Coroutines in a synchronous call.

mechanism can be implemented using `rpc.async` as a basis and the remote call in this case returns a *promise* [35], which can be used to retrieve the results later. When the promise is invoked, we use coroutines to suspend execution if the results are not yet available—in a similar way to what we did in the implementation of `rpc.sync`.

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

## 4.3. Monitors

When using synchronous invocations and futures, we allow coroutines to block themselves while waiting for the result of an invocation. While such a coroutine is blocked, other invocations may arrive and modify shared globals, leaving them in a state different from that which the blocked coroutine expects upon being resumed. We thus introduce the possibility of race conditions, although in a far coarser grain than that of preemptive multithreading (with coroutines, "context switches" may only

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

**Fig. 8.** Implementation of `rpc.future`.

occur at explicit points in the code), and the eventual need for mutual exclusion mechanisms. Instead of providing a built-in mechanism, we argue that the preferred abstraction can be easily implemented over the existing primitives and integrated into a given application, either as a library or as an application module. As an example, in this section we discuss an implementation of monitors [28]. Monitors described here are different from the classic proposals in that they are dynamic: functions may be added to a monitor at any point in execution.

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

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

Fig. 9 shows the implementation of function `monitor.doWhenFree`. This function creates and returns a new function that encapsulates the one received as a parameter. This new function uses the lock to guarantee the execution in mutual exclusion with respect to other functions in the same monitor. Function `monitor.doWhenFree` also deals with the input parameters and

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

**Fig. 9.** Implementation of function `monitor.doWhenFree`.

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

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

This mechanism for mutual exclusion is different from most classic language proposals in that it does not provide direct syntactic encapsulation of the protected functions. This makes the monitor a dynamic mechanism, allowing functions to be added to the monitor only as needed. This idea is possible using the fact that functions are first-class values, closures, and dynamic function creation. As shown in Fig. 9, a new function, `f`, is dynamically created to wrap the unprotected `func`, passed as parameter to `doWhenFree`. The closure mechanism allows `f` to keep a reference to `func`, so that `f` can invoke it in a protected section.

In object-oriented languages like C++ or Java, we can achieve a similar mechanism instantiating a wrapper to intercept the calls and impose the ordered entrance in the critical section. However, the fact of handling function as first-class values gives us a fine-grained control compared to the object interception because we can redefine only a function and leave all the rest untouched. The wrapper must keep the same object's interface, add code to protect the desired methods, and forward all the calls to the original object.

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

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

## *4.4. Synchronization constraints and synchronizers*

In this section, we continue our discussion on how we can integrate coordination mechanisms into our basic communication model. To illustrate the flexibility we offer in this integration, we turn our attention to the possibility of defining conditions for function execution. These conditions will allow us to model both intra and inter-process coordination. Among existing proposals

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

**Fig. 10.** A distributed monitor.

in this direction, we chose to reimplement the mechanism proposed by Frølund and Agha [21,2], because it is one of the few that provides support for distributed as well as for concurrent synchronization.

To implement support for this abstraction, we explored the possibility of redefining event handlers for remote events. Upon arrival of a function invocation, our implementation directs it to a handler. The default handler simply calls the requested function passing the arguments received in the request, and returns the results to the caller. However, we can define different handlers for different functions or even for different requests for a same function.

Intra-object synchronization in [21,2] is supported by *synchronization constraints*. As with *guards* [41,13], the idea is to associate constraints or expressions to a function to determine whether or not its execution should be allowed in a certain state. This kind of mechanism allows the developer to separate the specification of synchronization policies from the basic algorithms that manipulate his data structures, as opposed to monitors, in which synchronization must be hardcoded into the algorithms.

As an example taken from [21], consider a radio button object with methods `on` and `off`. To ensure that these methods are invoked in strict alternations, the programmer can define a state variable `isOn`, that indicates whether or not the button is turned on. Synchronization constraints can be defined disabling method `on` when `isOn` is true, and disabling method `off` when it is false.

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

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

We provide support for these mechanisms through modules `sc` (synchronization constraints) and `synchronizer`, that interact with a new handler we defined for function invocation. Modules `sc` and `synchronizer` introduce constraints that must be checked by this handler. In the case of the `sc` module, these are local calls that verify the internal state, whereas `synchronizer` permits processes to register themselves as synchronizers of each remote object (or process, in our case) they coordinate. If the constraints are not satisfied, the handler places the request in a queue which is reexamined after any other invocation is executed.

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

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

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

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

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

**Fig. 11.** Defining synchronization constraints.

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

**Fig. 12.** A synchronizer defining a remote constraint and triggers for a set of distributed buttons.

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

We first implemented the scheme as described in [21], that is, verifying and executing the requests in a sequential fashion—a new request is handled only after the previous one is completed. However, evaluating remote constraints is expensive because the process communicates with the synchronizer and blocks until the answer arrives. Fig. 13a shows a diagram illustrating this scheme. Although the synchronizer imposes constraints only on function `set`, the process waits for the synchronizer's answer arrival before executing the request `get`.

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

Our implementation checks the synchronization constraints before contacting the synchronizers, avoiding the network communication if the local verification fails. However, because new function calls can now modify the internal state during the remote

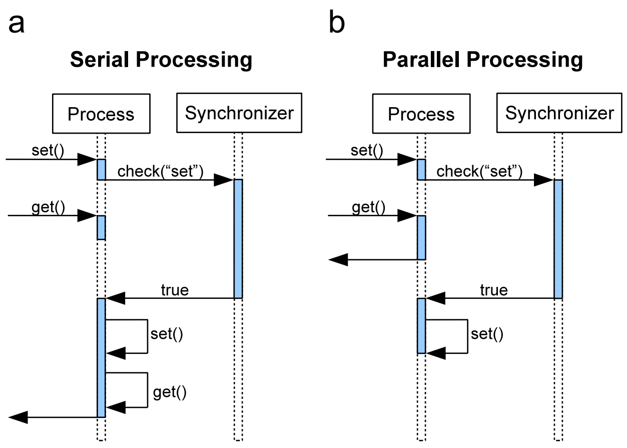

**Fig. 13.** Diagram of (a) the original and (b) our proposal for constraints evaluation.

**Table 1**
Execution times for different constructs.

| Mechanism | Time (ms) |
|---|---|
| Java RMI–JIT enabled | 0.426 |
| Java RMI–JIT disabled | 0.803 |
| RPC | 1.134 |
| Monitor | 1.758 |
| Queue handler | 1.328 |
| Local constraint | 1.338 |
| Remote constraint | 4.281 |

constraints checking, it verifies the synchronization constraints again when it receives all replies from synchronizers in order to guarantee that the internal states still allows the request execution.

## 5. Performance results

In this section, we discuss performance of the mechanisms we described along the paper.[1] Our goal is to evaluate the minimal cost that these mechanisms add to the basic RPC mechanism. We measured the average time for a client to execute an asynchronous RPC request to a remote function in the following cases:

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

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

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

Because we were interested in exploring the aspect of being able to build different abstractions, we did not dedicate ourselves to optimizing the implementation of RPC, so, when compared to RMI, which is built as part of Java's architecture, execution times seem quite acceptable.

[1] The code and results we discuss in this section are available at alua.inf.puc-rio.br/ftp/papers/CLSS09-perf/.

**Table 2**
Serial and parallel processing of requests.

| Request | Serial processing (ms) | Parallel processing (ms) |
|---|---|---|
| `get` | 5.138 | 2.858 |
| `set` | 5.247 | 5.314 |

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

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

We performed a second experiment to measure the time for the server to process a set of requests in a serial and parallel fashion, as described in Section 4.4. In this experiment, the server exports two functions, `get` and `set`, and receives requests of two clients, each one invoking a specific function (Fig. 13). The `set` function is protected by a remote constraint, whereas `get` has no protection. First, we configured the server to execute the requests in a serial fashion, i.e., the server processes the next request only when it finishes processing the previous one. We then changed the server behavior to execute a new request while the synchronizer evaluates the remote constraint. Table 2 shows the times (in milliseconds) for each scenario. In the serial case, the request for the `get` function is limited by the evaluation of the remote constraint, so the requests for each function take almost the same amount of time. However, in the parallel configuration, the server can process the `get` requests while the synchronizer evaluates the `set` remote constraint, so that average time of the `get` request is much lower.

## 6. Related work

Over the years, support for synchronization and communication abstractions has been implemented in several programming languages, such as Emerald [12], Orca [10], Java [24], Erlang [7], and E [37], and in a number of libraries, such as SunRPC [48], Linda [17], JXTA [23], ProActive [16], and Chord [47]. Our goal in presenting the implementation of different coordination abstractions is not to discuss the mechanisms themselves, but to support the argument that programming language features can help bridge the gap between the simplicity and flexibility offered, respectively, by the language and library approaches [14].

We have been investigating event-driven model and with asynchronous message-passing in the development of distributed systems. Some other works explore synchronous communication facilities over event-driven systems [44,32,52]. Haller and Odersky [26] are explicitly interested in avoiding the problems associated to the typical "control inversion" of event-driven systems. However, in the case of their work, the "continuation" of the current computation must be explicitly coded in a receive clause.

Adya et al. [1] discuss advantages and disadvantages of multithreading and event-based programming, considering the distinction between *manual* and *automatic* stack management. One point that further links their work to ours is the emphasis on allowing the programmer to freely combine both models when coding an application. Eugster et al. [20] also discuss the importance of combining different interaction paradigms (RMI and publish/subscribe) in a single distributed programming tool.

On top of our event-driven model, we investigated how characteristics of a language can help to create and compose abstractions of communication and coordination in order to provide more suitable mechanisms to aid the development of distributed systems. Arbab and Papadopoulos [40] present a survey of coordination models and languages, and classify the surveyed works in two major categories. *Data-driven models* offer coordination primitives which can be freely mixed with the computational code. Proposals such as Linda and the synchronizers we discussed in Section 4.4 are placed by the authors in this category. The other category, *control-driven models*, encompasses models in which the coordinating entities are separate from the computational ones. Typical examples of this category are *configuration* languages [36,6], which focus on describing interconnections between independent processes or components.

As far back as in 1978, Backus [8] argued that programming languages in general should provide an expressive core with powerful changeable parts and combining forms, instead of incorporating a large set of features. In the specific domain of concurrency and distribution, Briot et al. [14] discussed how reflective features can bridge the gap between the specific domain programming language (*integrative*) and *library* approaches. However, there appears to be little recent work exploring how language features interact with libraries in order to allow uniform forms for concurrency and distribution in the language.

One good example of language extension exploring language features is ProActive [9], a library for parallel and distributed programming in Java. Its architecture is based on *active objects* that are entities with their own thread of control, and method calls on them are asynchronous. The library is implemented in standard Java though use of reflection, and the result is very similar to a specific purpose language. Thanks to polymorphism, handling of remote objects and also of the future objects that are returned by asynchronous invocations is transparent. However, papers on ProActive do not emphasize the role of language features in these extensions. On the other hand, Sewell et al. [45], in their presentation of Acute, argue that a language should offer explicit primitives for distribution and communication and provide support for libraries to implement high-level abstractions, but do not give examples of such abstractions.

The work of Varella and Agha [51] and that of Eugster et al. [20] are among the few that approach the discussion of distributed programming libraries from a language point of view. Varella and Agha [51] point out the need for extensions to Java, indicating that the original set of language features is insufficient, and on the other hand defend that only a few extensions can suffice to provide a simple distributed programming model. Eugster et al. [20] identify a set of mechanisms which enforce language support for publish/subscribe. Specifically, they identify the concept of closures as one of these mechanisms. Their work also intends to contribute to the discussion about whether support for publish/subscribe communication in object-oriented languages should be provided as a library or as part of the programming language.

With the evolution of managed run-time environments (MRTE), such as .NET and JVM, more and more compilers are being developed to generate code for this kind of platform [27,25,18]. MRTEs provide facilities such as portability, automatic memory management, large sets of libraries, and language interoperability. The idea behind the multilanguage platform is to use the most suitable tool to accomplish each task, allowing languages to export their capabilities and libraries to the others. We think this can represent another level of integration, that will have to deal with the resolution of external capability integration and programming model mismatching.

## 7. Final remarks

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

It is interesting to summarize how we explored language features along the paper. We started with a very simple event-driven programming model. The ALua model derives its flexibility from the support for executing dynamically created chunks of code. This support is typical of interpreted languages. We then developed, over this model, an abstraction for asynchronous RPC which allows the programmer to specify a callback function to be executed when the invocation is completed. The use of closures and first-class function values allowed us to create new remote functions that act exactly like the local ones. We discussed how closures could be used to encapsulate state that must be remembered when a callback of an asynchronous invocation is executed. Next, we went on to build different coordination mechanisms over this basis. Combined with the coroutine construct, the closure/callback pattern allowed us to build a synchronous RPC abstraction, which has the advantages of cooperative multithreading while hiding the details of control transfer inside the remote invocations. Closures and first-class functions were again used to build a monitor-like mechanism with a dynamic behavior. Finally, to experiment with abstractions that support the definition of necessary state conditions for functions to be executed, we implemented a mechanism similar to Frølund's synchronizers. The dynamic nature of ALua was important in allowing us to modify handlers for incoming function invocations. Synchronization was implemented by defining a handler that manages a queue with blocked requests, but no changes to the syntax of invocations were needed.

With this step-by-step development of a set of communication and synchronization abstractions, we hope to have made the case that (sequential) programming languages with appropriate extension features allow us to combine the advantages of the library and language approaches for concurrent and distributed programming. Languages designed from scratch for concurrency and distribution are usually elegant and simple to understand for a given coordination model. Features such as first-order functions and closures allow libraries to be integrated seamlessly into the programming language, resulting in environments that can be seen as specific-domain programming languages. However, these environments are not bound to one single coordination model, and allow the programmer the flexibility of choosing appropriate abstractions for each task.

Languages with the features we emphasize, such as dynamic execution model and support for functions as first-class values and closures, are often interpreted languages. Although interpreted languages, in general, are not as efficient as compiled languages for computing intensive systems, we can use a dual programming model as discussed in [50]. The idea is that the interpreted language can be used to coordinate the application, while a traditional compiled language handles the computing-intensive parts.

## Acknowledgments

We would like to thank CAPES and CNPq (Brazilian government research agencies) for having partially supported this work.

## References

[1] Adya A, Howell J, Theimer M, Bolosky W, Douceur J. Cooperative task management without manual stack management. In: USENIX annual technical conference, Berkeley, 2002. p. 289–302.
[2] Agha G, Frolund S, Kim W, Panwar R, Patterson A, Sturman D. Abstraction and modularity mechanisms for concurrent computing. IEEE Parallel and Distributed Technology: Systems and Applications 1993;1(2):3–14.
[3] Ananda AL, Tay BH, Koh EK. A survey of asynchronous remote procedure calls. SIGOPS Operating Systems Review 1992;26(2):92–109.
[4] Andrews G. Foundations of multithreaded, parallel, and distributed programming. Reading, MA: Addison-Wesley; 2000.
[5] Andrews G, Schneider F. Concepts and notations for concurrent programming. ACM Computing Surveys 1983;15:3–43.

[6] Arbab F, Herman I, Spilling P. An overview of Manifold and its implementation. Concurrency: Practice and Experience 1993;5(1):23–70.
[7] Armstrong J, Virding R, Wikstrom C, Williams M. Concurrent programming in Erlang. Englewood Cliffs, NJ: Prentice-Hall; 1996.
[8] Backus J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM 1978;21(8):613–41.
[9] Baduel L, Baude F, Caromel D, Contes A, Huet F, Morel M. et al. Programming, deploying, composing, for the grid. Grid computing: software environments and tools. Berlin: Springer; 2006.
[10] Bal H, Kaashoek M, Tanenbaum A. Orca: a language for parallel programming of distributed systems. IEEE Transactions on Software Engineering March 1992;18(3):190–205.
[11] Birman K, van Renessee R, editors. RPC considered inadequate. Reliable distributed computing with the Isis toolkit. Silver Spring, MD: IEEE Computer Society Press; 1994. p. 68–78.
[12] Black A. et al. Distribution and data types in Emerald. IEEE Transactions on Software Engineering 1987;SE-13(1).
[13] Briot J-P. Actalk: a framework for object-oriented concurrent programming—design and experience. In: Bahsoun J-P, Baba T, Briot J-P, Yonezawa A, editors. Object-oriented parallel and distributed programming. Paris, France: Hermès Science Publications; 2000. p. 209–31.
[14] Briot JP, Guerraoui R, Lohr KP. Concurrency and distribution in object-oriented programming. ACM Computing Surveys 1998;30(3).
[15] Butenhof D. Programming with POSIX threads. Reading, MA: Addison-Wesley; 1997.
[16] Caromel D, Klauser W, Vayssiere J. Towards seamless computing and metacomputing in Java. Concurrency Practice and Experience 1998;10(11–13):1043–61.
[17] Carriero N, Gelernter D. How to write parallel programs: a guide to the perplexed. ACM Computing Surveys 1989;21(3):323–57.
[18] Clojure programming language ⟨http://clojure.org/⟩.
[19] Common Language Infrastructure (CLI), 2006, Standard ECMA-335.
[20] Eugster P, Guerraoui R, Damm C. On objects and events. In: Conference on object-oriented programming, systems, languages, and applications, 2001. p. 254–69.
[21] Frølund S. Coordinating distributed objects: an actor-based approach to synchronization. Cambridge, MA: The MIT Press; 1996.
[22] Gelernter D, Carriero N. Coordination languages and their significance. Communications of the ACM 1992;35(2):97–107.
[23] Gong L. JXTA: a network programming environment, IEEE Internet Computing, May 2001. p. 88–95.
[24] Gosling J, Joy B, Steele G, Bracha G. The Java language specification. third ed., Reading, MA: Addison-Wesley; 2005.
[25] Groovy programming language ⟨http://groovy.codehaus.org/⟩.
[26] Haller P, Odersky M. Event-based programming without inversion of control. In: Proceedings of the 7th joint modular languages conference (JMLC 2006), Oxford, UK, 2006.
[27] Haller P, Odersky M. Scala actors: unifying thread-based and event-based programming. Theoretical Computer Science 2008;410(2–3):202–20.
[28] Hoare CAR. Monitors: an operating system structuring concept. Communications of the ACM 1974;17(10):549–57.
[29] Ierusalimschy R. Programming in Lua, Lua.org, 2nd ed., 2006.
[30] Ierusalimschy R, Figueiredo L, Celes W. The Lua programming language ⟨http://www.lua.org⟩.
[31] Ierusalimschy R, Figueiredo L, Celes W. Lua—an extensible extension language. Software: Practice and Experience 1996;26(6):635–52.
[32] Lea D, Vinoski S, Vogels W. Asynchronous middleware and services. IEEE Internet Computing 2006;10(1):14–7.
[33] Leal MA, Rodriguez N, Ierusalimschy R. LuaTS—a reactive event-driven tuple space. Journal of Universal Computer Science 2003;9(8):730–44.
[34] Lieberman H. Concurrent object-oriented programming in Act 1. Object-oriented concurrent programming. Cambridge, MA: The MIT Press; 1987 p. 9–36.
[35] Liskov B, Shrira L. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: ACM SIGPLAN conference on programming language design and implementation, New York, NY, USA, 1988. p. 260–7.
[36] Magee J, Dulay N, Kramer J. A constructive development environment for parallel and distributed programs. IEE/IOP/BCS Distributed Systems Engineering 1994;1(5).
[37] Miller MS, Tribble E, Shapiro J. Concurrency among strangers—programming in E as plan coordination. In: Symposium on trustworthy global computing (European joint conference on theory and practice of software), Lecture notes in computer science, vol. 3705. Berlin: Springer; 2005.
[38] Mitra N, Lafon Y. SOAP version 1.2 part 0: Primer, 2007, W3C Recommendation.
[39] Ousterhout J. Why threads are a bad idea (for most purposes). In: USENIX technical conference, January 1996, Invited talk.
[40] Papadopoulos G, Arbab F. Coordination models and languages. Advances in computers, vol. 46. New York: Academic Press; 1998. p. 329–400.
[41] Riveill M. Synchronising shared objects. Distributed Systems Engineering Journal 1995;2(2):112–25.
[42] Rodriguez N, Rossetto S. Integrating remote invocations with asynchronism and cooperative multitasking. Parallel Processing Letters 2008;18(1):71–85.
[43] Rossetto S, Rodriguez N, Ierusalimschy R. Abstraç oes para o desenvolvimento de aplicações distribuídas em ambientes com mobilidade. In: VIII Simpósio Brasileiro de Linguagens de Programação; 5:2004. p. 143–56.
[44] Schmidt D, Cranor C. Half-sync/half-async: an architectural pattern for efficient and well-structured concurrent I/O. Pattern Languages of Program Design 1996;2:437–59.
[45] Sewell P, Leifer J, Wansbrough K, Nardelli F, Allen-Williams M, Habouzit P. et al. Acute: high-level programming language design for distributed computation. Journal of Functional Programming 2007;17(4–5):547–612.
[46] Siegel J. CORBA fundamentals and programming. New York: Wiley; 1996.
[47] Stoica I, Morris R, Karger D, Kaashoek M, Balakrishnan H. Chord: a scalable peer-to-peer lookup service for internet applications. In: ACM SIGCOMM, San Diego, CA, August 2001. p. 149–60, code available from ⟨pdos.csail.mit.edu/chord/⟩.
[48] Sun Microsystems, RPC: remote procedure call, protocol specification, Version 2, June 1988, RFC 1057.
[49] Tanenbaum A, van Renesse R. A critique of the remote procedure call paradigm. In EUTECO'88 conference, Vienna, 1988. p. 775–83, Participants Edition.
[50] Ururahy C, Rodriguez N, Ierusalimschy R. ALua: flexibility for parallel programming. Computer Languages 2002;28(2):155–80.
[51] Varela C, Agha G. Programming dynamically reconfigurable open systems with SALSA. ACM SIGPLAN notices. OOPSLA'2001 intriguing technology track proceedings, vol. 36(12), December 2001. p. 20–34.
[52] Vinoski S. RPC under fire. IEEE Internet Computing 2005;9(5):93–5.
[53] von Behren R, Condit J, Zhou F, Necula G, Brewer E. Capriccio: scalable threads for internet services. In: SOSP '03: proceedings of the nineteenth ACM symposium on operating systems principles. New York: ACM Press; 2003. p. 268–81.
[54] Welsh M, Culler D, Brewer E. Seda: an architecture for well-conditioned scalable internet services. In: 18th symposium on operating systems principles (SOSP-18). Banff, Canada: ACM; 2001. p. 230–43.