INTEGRATING REMOTE INVOCATIONS WITH ASYNCHRONISM AND COOPERATIVE MULTITASKING

NOEMI RODRIGUEZ SILVANA ROSSETTO
Departamento de Informatica, PUC-Rio, Rua Marques de Sao Vicente, 225, Gavea
Rio de Janeiro, RJ, 22453-900, Brazil
noemi,silvana@inf.puc-rio.br

ABSTRACT

In this paper we argue that it is possible to couple the advantages of programming with the well-known abstraction of RPC with asynchronous programming models adequate for wide-area programming environments such as grids. We discuss how some programming language features can help create different programming abstractions over a basic asynchronous invocation primitive. The paper also discusses how coroutines (cooperative multitasking) can be used to allow computation to proceed while a client is waiting for the result of a remote invocation, avoiding the pitfalls of programming with threads.

1. Introduction

The direct use of low-level send-receive operations usually available at operating system and library level is complicated and often results in programs that are hard to understand and to debug. Thus, when distributed computing became a practical reality, in the early eighties, researchers and developers started searching for appropriate programming abstractions. The Remote Procedure Call (RPC) mechanism [1], which extended to distributed programming what is probably the most basic abstraction for sequential programming, received a lot of attention, and was later promoted to an ad-hoc standard by Sun's RPC tools [2] In the early nineties, remote procedure invocations were replaced by remote object invocations [3,4], but the main ideas remained much the same.

As the focus of distributed programming shifted to wide-area networks, in the late nineties, limitations of the classical, synchronous, RPC mechanisms for this new environment triggered research and development of new communication models. Many developers turned their attention to message-oriented communication, which had been regarded as too low level for application programming in the previous decade.

In this paper we argue that, with appropriate language support, it is possible to couple the benefits of programming with a well-known abstraction such as RPC with the asynchronous execution model that is needed for wide-area platforms such as grids. We propose asynchronous function calls as the basis for all communication. The programmer can associate a callback to an asynchronous invocation, allowing results to be handled in an asynchronous way. But, because this is not a natural model for programmers, we also provide synchronous invocations, which are built over the asynchronous basic primitive. We discuss how the concepts of functions as first-class values, coroutines, and closures can help the construction of programming abstractions. To allow the computation to proceed at a process that has issued a remote invocation, we use cooperative multitasking, implemented through Lua coroutines [5]. This eliminates many of the problems associated to multithreading, which is the concurrency mechanism traditionally used in this setting. We use the Lua programming language because it offers the necessary support and because of our involvement with Lua for distributed programming [6,7], but the approach we propose could be perfectly applied in other languages.

The paper is organized as follows. Section 2 discusses requirements for programming models in wide-area distributed platforms. In Section 3 we describe *LuaRPC*, an event-driven library providing asynchronous invocations as a basic inter-process communication primitive. This section also describes the synchronous invocation and *future* mechanisms built over this basic communication mechanism. Section 4 contains a discussion about cooperative multitasking as an alternative to multithreading. In Section 5 we discuss how we implemented *LuaRPC*. Section 6 compares this work to related works. Finally, Section 7 contains some closing remarks.

2. Wide-area parallel and distributed computing and asynchronism

Parallel and distributed programming communities have traditionally stayed apart, even though they share many common problems. This separation has been specially true in what regards the study of programming paradigms. Research in distributed programming has usually explored alternative models more freely, while parallel programmers, maybe due to their tight focus on performance, have mostly stuck to variants of message passing libraries coupled with C or FORTRAN. (Exceptions to this general trend exist: notably the tuple-space proposal, which was born in the parallel programming community [8] and later adopted as a distributed programming model [9,10].)

The popularity of grid computing is changing this picture. One reason for this is that when using resources from different geographical and administrative domains the programmer is forced to deal with issues that could be ignored in dedicated, homogeneous clusters. Another aspect in grid computing is that the philosophy of resource usage, in many cases, is to take advantage of idle resources in different institutions. Because programs must adapt to dynamic execution conditions, the focus on performance must be relaxed to take into account other sets of programming constraints. Besides, when considering homogeneous settings such as a cluster, programmers can deal with the complexity of message passing by writing strongly-coupled SPMD programs. In an heterogeneous and dynamic environment, pro-

gramming abstractions become more important. These issues have brought many groups who historically worked with distributed computing to develop projects and proposals for grid, bridging the gap between the two communities.

In distributed programming history, the remote procedure call paradigm can be classified as a successful programming model. Besides the classical Sun RPC implementation, many other mainstream systems, such as CORBA [3] and Jini [4], have been implemented using its basic ideas. The RPC paradigm was probably the most popular one for client-server applications running in local-area networks. From the beginning, however, critiques to the RPC paradigm were made [11,12], mostly discussing the imposition of a synchronous structure on the client application and the difficulty of matching RPC with fault tolerance, partly due to its one-to-one architecture.

As the scope of distributed systems grew, from local-area to geographical networks, both of these aspects gained importance. On the one hand, communication across geographical networks means more variation in response times. On the other hand, the probability of failures in communication peers is larger when messages are exchanged across administrative domains. New critiques to the one-to-one and synchronous RPC model have also appeared from novel application areas, such as ubiquitous computing [13]. All of these considerations led researchers and developers to consider alternative programming paradigms, and, more specifically, led to emphasis on asynchronism.

Typical asynchronous programming systems provide support for event-based or message-oriented programming. In this model, processes register interest in classes of events, and are notified when these events occur, often through the use of *callback functions*. Processes and operations are not coded as a sequence of steps but as a passive state machine: the arrival of an event is handled by a function whose execution leads the program to a new state.

Although this "inverted" structure can be quite convenient for many applications, it can be awkward to use in other cases, such as some parallel applications, where there is an established set of programming patterns, or even applications which exhibit an obvious client-server behaviour. This has led us to reconsider the remote procedure call mechanism and to analyse the extent to which it is really inappropriate for wide-area distribution. What is clear is that the classical RPC mechanism imposes much more synchronism than is reasonable. We ponder that maybe this has led many researchers to discard the whole RPC proposal, and that maybe the invocation abstraction can still be useful in asynchronous environments.

In the next section, we propose an RPC mechanism for asynchronous environments. The basic idea is to implement asynchronous functions calls as the basis for all communication, instead of synchronous calls as in the classical RPC abstraction. In order to receive the result of an invocation and integrate it into the ongoing computation, the caller can define a callback function that will be executed when the result is available.

3. Asynchronous and Synchronous Remote Procedure Calls

In this section, we describe an asynchronous procedure call mechanism that we implemented using the Lua programming language [5]. Using Lua, we are able to

take advantage of facilities such as closures and functions as first-class values to implement high level communication abstractions, but, at the same time, because Lua is primarily intended for use as an embedded language, we can keep the hard processing in C or C++ [6].

Lua is an interpreted language with a Pascal-like syntax. One of the main design aspects of Lua is its simplicity and flexibility. Two types in Lua deserve special attention in our context: *table* and *function*. Tables can grow and shrink dynamically and implement associative arrays, that is, their indices can be values of arbitrary types. A table can thus be used as a general-purpose container. To create a table, one must use a *constructor expression*, which can be a simple {}, creating an empty table:

```
mytable={}     -- creates a table
mytable["x"]=3 -- new entry, with index "x" and value 3
if (mytable.x==3) then ... -- syntactic sugar for mytable["x"]
```

Functions, in turn, are first-order values, i.e. they can be passed as arguments to or be used as return values from other functions, or be stored in a table.

We use these facilities for creating a *LuaRPC* module. This module imposes an event-oriented structure on a program: The program, after initial actions, typically issues a call to `luarpc.loop`, which will make it continuously wait for new communication events. The main primitive in this module is `createAsyncCall`, which creates a function value which, when invoked, issues an asynchronous remote procedure call:

```
f = luarpc.createAsyncCall(funcId, host, port, callback)
...
f(arg1, ..., argn) -- remote invocation
```

Function `luarpc.createAsyncCall` is relatively low-level, in that it takes as parameters a host (IP number) and port of the remote process. Its first argument, `funcId`, is a string identifying the function as exported by the remote process (we will return to this point later). Finally, the last argument is a callback function, to be executed when the remote call is completed.

The function returned by `luarpc.createAsyncCall` is a regular Lua function, and, as such, can be assigned to variables or fields, passed as an argument, and so on. In Lua, functions can take variable number of arguments, and so the programmer can use the function returned by `luarpc.createAsyncCall` as a local function, passing the necessary arguments in the invocation. (The role of the `luarpc.createAsyncCall` is similar to that of a stub generator in a traditional RPC system.)

When the remote call is completed, the callback function is invoked in the calling process. This callback function will receive as its argument a table with possible fields `error` and `results`. If the error field exists, its value will be a string indicating the kind of error that has occurred.

Figure 1 shows the outline of a program that distributes tasks, in a round-robin fashion, to be executed by a set of *worker* processes. Table *workers* stores the *ip* and *port* numbers of each available worker. After sending all tasks to remote workers, the program enters the event loop. Function `luarpc.loop` takes as an argument a

```
-- initial settings
publicFunctions = {}
function publicFunctions.handleresults (ret)
  if ret.error then
    -- handle error
  else
    total = total + ret.results.value
    workleft = workleft - 1
    if (workleft==0) then
      -- show results and exit
    end
  end
end
-- create remote function values
remotework = {}
for k in pairs(workers) do  -- iterates over table workers
                            -- to create table remotework
  table.insert(remotework, luarpc.createAsyncCall("processtask",
                workers[k].host, workers[k].port, "handleresults")
end
-- distribute tasks, table tasks contains a bag of tasks
-- in this simple example each task is described by a single number
host = next(remotework, nil)  -- iterates over table remotework
workleft = 0
for i,task in pairs(tasks) do -- iterates over table tasks
  remotework[host](task)      -- sends a task to a remote worker
  workleft = workleft + 1
  host = next(remotework, host) or next(remotework, nil)
end
luarpc.loop(publicFunctions)
```

Fig. 1. Code outline for round-robin task distribution

table containing the functions “exported” by this process, i.e., the functions that can be called remotely. The callback function collects results that are sent back. (An on-demand task distribution strategy could be implemented by sending only one task to each worker and, in function *handleresults*, sending a new task.)

In this example, because the remote invocations are asynchronous, we rely completely on the passive, machine-state style of programming which we mentioned in Section 2. Programs written with *LuaRPC* have an event-driven structure, in which incoming communication events trigger the execution of functions. These may be either functions that are being invoked remotely or callbacks associated to some completed remote invocation. Each process is single-threaded, that is, all incoming events must wait for this single thread to finish handling previous events.

This is a convenient structure for programs executing in wide-area environments, but may not be a convenient way for the programmer to design her code. In the next section, we propose a synchronous view of the remote function call that does not interfere with the asynchronous nature of the program as a whole.

```
...
publicFunctions = {}
function publicFunctions.availableWorker(mst)
  getWork = luarpc.createSyncCall(mst.newTask, mst.host, mst.port)
  putResult = luarpc.createSyncCall(mst.taskCompleted, mst.host, mst.port)
  whenFree = publicFunctions.availableWorker
  publicFunctions.availableWorker = nil
  work(getWork, putResult)
end
function work(get, put)
  local w = get()
  while w do
    put(localwork(w))
    w = get()
  end
  publicFunctions.availableWorker = whenFree
end
luarpc.loop(publicFunctions)
```

Fig. 2. Worker process in pool-of-workers application

3.1. Synchronous Invocations

With asynchronous invocations and event-driven programs, 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. Using coroutines, we encapsulate this common behavior in a new function, called `luarpc.createSyncCall`. When a programmer writes something like:

```
f = luarpc.createSyncCall(funcId, host, port)
...
r = f()
```

the assignment to variable `r` only takes place after the remote invocation has been completed.

This behavior must be coupled with the basic event-driven and single-threaded structure of the process. So the process cannot effectively remain blocked while the remote function is being executed. To deal with this, the main event loop is coded in a main coroutine, and all other processing occurs in secondary coroutines. A synchronous call encapsulates a *yield* from a secondary coroutine to the main one, allowing other events to be processed while the remote invocation proceeds. The implementation will be discussed in further detail in Section 5. This coupling of synchronous communication abstractions, with which programmers are more familiar, with an asynchronous program structure, more appropriate for wide-area applications, tries to extract the advantages of each model.

As an example, consider again the problem of distributing tasks. Figures 2 and 3 sketch worker and master processes in a new program structure. We now imagine worker processes that continuously offer their processing services. For this, they offer a remote function called `availableWorker`, that triggers the execution

```
...
publicFunctions = {}
function publicFunctions.workUnit()
  -- if there are work units left then return unit
  -- else return nil
  -- end
end
function publicFunctions.result(newresult)
  -- store newresult
end
master = {taskCompleted="result", newTask="workUnit",
          host=myHost, port=myPort}
luarpc.createBcastAsyncCall("availableWorker", nil)(master)
luarpc.loop(publicFunctions)
```

Fig. 3. Master process in pool-of-workers application

of a loop in which the worker repeatedly asks the master for a task description, executes the task, and returns results. Workers now request tasks and return results in synchronous calls to the master, inverting the direction of the invocations in Figure 1.

The invocation of `availableWorker` triggers calls to `newTask` and `taskCompleted`, assigning values to `getWork` and `putResult`, using the argument it received, and sets the value of `availableWorker` to nil. This is equivalent to exiting the pool of available workers. Finally, `availableWorker` invokes `work`, which simply contains a loop for retrieving tasks from the master and sending results. When the master has no more work to be done, the worker sets the value of `availableWorker` back to its original one, thus returning to the processing pool.

On its side, a master process (Figure 3) begins execution by broadcasting a call to `availableWorker`. *LuaRPC* offers the `createBcastAsyncCall` primitive for sending an invocation to a set of hosts (this set is initialised in the configuration file). This is just an example, the list of possible workers could have been obtained in a more application-specific way. When a new master process issues this invocation, some of the workers will be available while some of them will possibly hold no value for function `availableWorker` at the moment (because they are working for some other master). The available workers will begin processing the new application.

Other issues could be explored such as what would happen if no workers are available. We could insert in `availableWorker` a call to a function in the master process indicating whether this worker will join the work pool. Thus, the master would be able to react to situations in which the size of the work pool is too small.

3.2. Futures

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. Another interesting abstraction for synchronizing actions of clients and servers in a looser relationship than with synchronous invocations is that of *futures* [14,15]. Futures represent a promise of a value that will be eventually available. Suppose that a

```
futureQuery = luarpc.createFutureCall("newQuery", host, port)
getResult = futureQuery(arguments) -- invocation returns "future"
... do other activities
res = getResult() -- now wait for the result
```

Fig. 4. Use of the future transformation

program knows it will need a result from function `f`, but only at some later point in execution. If the program invokes `f` synchronously, computation will not proceed (in that line of execution) while `f` is executed. Support for futures allows a special call to `f` to proceed asynchronously but return a *future object*, that is, a function that, when called, synchronizes the remaining computation with the termination of `f`, and returns its results.

Figure 4 shows a simple example of the use of futures. Note that the "other activities" could include other (asynchronous or synchronous) remote procedure calls. Only when the program invokes the returned `getResult` function will execution be suspended, if the result is not already available.

4. Concurrency, Threads and Coroutines

The traditional way of dealing with the excessive synchronism of remote procedure call is to introduce multithreading in the client. However, as discussed in [13,12], this adds a burden of thread management: because thread implementations are mostly preemptive, control switches among different threads can occur not only when one of them blocks on an operation (such as a remote invocation), but at any point in execution. This results in race conditions and in the need for synchronizing primitives such as monitors and semaphores, which can make the program hard to understand and to debug.

Coroutines introduce multitasking in a cooperative fashion: each coroutine has its own execution stack, as a thread does, but control is transferred only through the use of explicit control transfer primitives. Using coroutines, a process can maintain several execution lines but only one at each time can run and the switch between two of them is explicit in the program. This allows applications to improve their availability without the weight that may come together with the multithreading solution. On the other hand, managing the transfer of control between coroutines is a burden for the programmer. But if we are able to build libraries in which possibly blocking calls encapsulate the transfer of control between coroutines, we obtain systems in which the potential points of context switching are explicit in the code, but in which the control of coroutines is hidden from the programmer. This is what we propose to do in the *LuaRPC* library.

Many parallel applications need multitasking only to make sure that each processor is used as much as it can be; they often do not have scheduling constraints that must be met by timesharing. Because context switches are time consuming, the timesharing nature of multithreading may impose a performance overhead on systems which do not need it. As one initial attempt at performance analysis, we conducted an experiment comparing a classical producer-consumer example, written in C, using coroutines and using threads. We used the pthread library for threads

and a library developed specifically for providing coroutines in C [16]. The program consisted of a simple producer-consumer loop with 500,000 iterations, with no processing other than assigning an integer to a buffer (at the producer) and adding this value to an accumulator (at the consumer). In out tests, the version with coroutines always executed in less than 7% of the time taken by the version with threads.

Coroutines were first introduced in programming languages in the seventies, with Simula [17], but have not been included in many programming languages. As discussed in [18], this may be due to a lack of a uniform view of the concept and also to the complexity of the first implementations. Recently, however, interest in this concept has reappeared, both in the area of multitasking applications [19] and in the area of scripting languages. Both Python and Perl currently provide restricted forms of coroutines [20,21].

It is important to note that there are several classes of applications for which response time must be limited for each line of execution. In this case multithreading may be an excellent solution. However, there are many other situations in which cooperative multitasking constitutes a lightweight alternative.

5. Implementation

The *LuaRPC* module was implemented in Lua and uses `LuaSocket` [22], a library offering a simplified view of Berkeley sockets. One advantage of the `LuaSocket` implementation is its flexible timeout control mechanism. As in C, all I/O operations are blocking by default. `LuaSocket` provides a *settimeout* method through which an application can specify upper limits on the time it can be blocked. It facilitates the use of sockets to implement asynchronous communication.

The key idea in *LuaRPC* is that the main behavior of all processes is the typical "server" behavior in socket language: a typical application starts by executing some initial code and then falls into a main loop where it waits for connections. When a connection is completed, the received message is completely handled and, after that, control goes back to the main loop to wait for new connections. When an application needs to send the results of an invocation to its caller or call a remote method offered by other processes, it starts a connection request to a specific (or to a set of) process.

The set of operations provided by *LuaRPC* includes:

- `createAsyncCall(funcId, host, port, <callback>)`: This operation returns a function that calls a remote method asynchronously. When the returned function is invoked, control returns immediately to the caller. Required parameters are the method identification and the host and port where the remote process is running. To receive and handle the results of this invocation, a callback function must be defined which will receive as arguments the requested results.

- `createSyncCall(funcId, host, port)`: This is a variant of `createAsyncCall` that also returns a function to invoke a remote method. The main difference is that, when this function is invoked, the effect is that of a synchronous call: execution flow will proceed only when the call is completed.

- `createBcastAsyncCall(funcId, <callback>)`: This is a variant of operation

```
function luarpc.createAsyncCall(funcId, host, port, callback)
  local callbackId = getcbId(callback)
  -- return a function
  return function(...)
    local msg = net.createMessage(funcId, callbackId, ...)
    return net.sendMessage(msg, host, port)
  end
end
```

Fig. 5. Skeleton implementation of `createAsyncCall`

`luarpc.createAsyncCall` that sends a request to a set of pre-defined processes.

- `createFutureCall(funcId, host, port)`: This is yet another variant of operation `createAsyncCall`. However, in this case, when the new function is invoked, it returns a function that, when called, synchronizes the remaining computation with the termination of the remote invocation, and returns its results.

As discussed (and explored in our examples) in Section 3, `createAsyncCall`, `createSyncCall` and `createBcastAsyncCall` do not directly make the remote invocations, but instead return a function that can be called a number of times after its creation, and that can be manipulated as any other value. This is possible because Lua supports functions as first-class values and lexical scoping. In Lua, a function can be enclosed in another function, and, in this situation, it has full access to variables from the enclosing function. This allows `createAsyncCall` (and its variants) to return a function, defined inside it, which depends on values passed as arguments on each specific invocation of `createAsyncCall`. To illustrate this, Figure 5 shows a sketch of our implementation of `createAsyncCall`. This implementation creates and returns an anonymous function which uses the values of variables `funcId`, `host`, `port`, and `callbackId`, which are local to the enclosing context. By the time the programmer uses the anonymous function returned by `createAsyncCall`, these variables will be out of scope. Nevertheless, they will be encapsulated in the definition of the new function. This feature of Lua implements the notion of *closures*, common in functional languages.

LuaRPC applications have a single line of control. A typical application starts by executing some initial code and then falls into a main loop, where it waits for public or callback functions activations. In this way, the application logic is normally put into functions, which are executed asynchronously. Considering only asynchronous operations, this model is sufficient to support the operations described above. However, when a synchronous operation is invoked, the next command must be executed only after the response to that request is available. Thus, the current execution line must be suspended and resumed later by the callback related to this request.

In order to support this architecture we use coroutines to model each function activation as a new coroutine. In the case of public method invocation, *LuaRPC* checks the public function table. If the current application offers the requested method, a new coroutine is created to execute it. On the other hand, in the case of callback invocation, the local function table is checked and the specified callback

is executed as a new coroutine. Each coroutine will execute to completion unless it contains a synchronous invocation. If the coroutine ends, control returns to the main loop. If it invokes a synchronous operation, a callback function is implicitly created (again using lexical scoping facilities) which is implemented as a function that will restore and resume the current coroutine when the result of the synchronous operation is available. Next, the control line is transferred back (*yielded*) to the main event loop. Hence, it is possible to suspend an execution line in order to wait for a remote invocation and yet allow the application to receive and handle other events.

Figure 6 illustrates the process of control transfer between coroutines and the main loop. The application starts by executing its initial code as a coroutine. After that, the loop waits for functions activations. When this occurs, one coroutine – A – is created (and immediately resumed) in order to execute the invoked method code. Control is transferred to this coroutine, which executes to completion. When the next message arrives, the main loop creates coroutine B. This coroutine transfers the control back to the main loop before it reaches its end (invokes a synchronous function). This case is again illustrated by coroutine C. When the result to the synchronous operation is available, the appropriate coroutine is resumed.

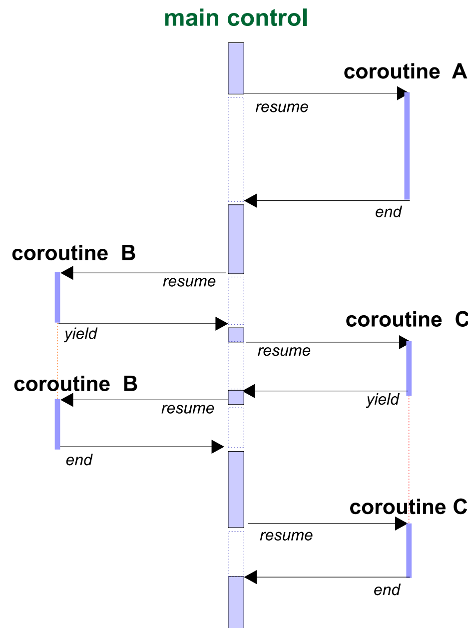

Fig. 6. Transferring control among coroutines.

In another simple performance experiment, we compared our implementation with coroutines with a variant using multithreading. For this, we implemented a threaded variant of the *LuaRPC* library: in this implementation, the arrival of a new function activation implies in the creation of a new thread. If a thread executes a synchronous call, it is suspended (on a condition variable) until the reply is available. In this version, we obviously loose the non-preemptive advantages

of the coroutine version. However, we were interested in the performance difference. We ran a test in which processes A, B, and C, each of them on a different machine, executed the following actions. Process A invoked (asynchronously) two times function f, on process B. Execution of f involved one synchronous call to g, on C, and a heavy processing loop (involving operations with floats). Execution of g involved only a processing loop. We ran this test on Pentium II machines executing Linux. The coroutine version took an average of 67,46 seconds while the multithreaded version took an average of 87,83 seconds (both on five runs). This is again a preliminary performance study, and we inserted the heavy processing loops in f and g exactly to extract the advantages of the non-preemptive version. However, we believe these results are promising and show that coroutines can help both in simplifying programming and in diminishing context-switching penalties.

6. Related work

A number of works propose alternatives to the multithreading model [23,24,25]. Specific approaches have been investigated in order to to combine concurrent programming with synchronisation and cooperative multitasking.

Fair Threads [26] is a framework that combines cooperative and preemptive scheduling of tasks in order to exploit some advantages of these two strategies (ease of programming and able to exploit hardware parallelism). The framework combines *user threads* and *services threads* which can execute concurrently and in parallel when the hardware supports it. Although our model is completely based on cooperative multitasking, it is possible to explore the hardware parallelism since distinct processes can be started at different pairs ip/ports in the same machine.

Future-based RMI [27] is a recent implementation of futures in procedural languages, with the purpose of optimizing the remote method invocation mechanism (RMI). The central idea is to support the composition of remote methods where one method uses the result of another method in order to reduce the data traffic on the network. When a server receives a remote invocation it runs the invoked method and returns an *object reference* that can be used as argument to other method invocations. This object implements a special method, *getdata*, which can be called by all clients interested in these results at the execution point that these values are required.

Our invocation abstractions are similar to abstractions offered in the *ProActive* [28] system, a Java middleware for parallel, distributed and concurrent programming. The central idea is that method calls are always asynchronous and *future objects* are transparently provided to collect the results of the call. The resulting programming model is reactive, as in our work. ProActive is 100% Java, and does not modify the language's basic multithreading model. In *LuaRPC*, synchronous calls (and futures) are coupled to a cooperative multitasking concurrency model.

7. Final Remarks

In this paper, we tried to show that it is possible to couple the simplicity of programming with RPC with the asynchronous requirements of wide-area distributed

computing. We used the *LuaRPC* module as a tool to experiment with these ideas. Although we are interested in exploring the possibilities of *LuaRPC*, our objective here was not to promote it specifically, but to create a small environment with adequate abstractions for distributed computing.

The basic asynchronous RPC primitive with a callback function allows programming in the direct event-driven style with the syntax of function calls for communication. Asynchronous calls can also be extended for one-to-many communication, as was shown with the broadcasting primitive. The judicious use of callbacks in this case allows the programmer to either explicitly acknowledge results returned by different hosts or to use the broadcast invocation in an event-publishing style. The use of synchronous calls and futures allied to cooperative multitasking allows the programmer to insert synchronization points in her code without having to deal with the classic shared memory issues posed by multithreading.

Languages with the features we emphasize, such as support for first-class functions and closures, are often interpreted. In the parallel programming community, this automatically leads to performance worries. However, as we discussed in [6], if we use a dual programming model, in which the interpreted language coordinates the application and a traditional compiled language handles the computing-intensive parts, results can be very good, specially considering the flexibility that can be achieved.

The programming examples we discussed were very simple. We intend to extend them, investigating other common frameworks for distributed applications, and exploring issues such as node failures and disconnections in the cases we have begun to discuss.

Acknowledgements

We would like to thank Ana Lucia de Moura for her support with performance experiments and CNPq (the Brazilian government research agency) for financial support.

References

[1] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Trans. on Computer Systems*, 2(1):39–59, February 1984.

[2] Sun Microsystems. RPC: Remote procedure call protocol specification, 1988.

[3] The common object request broker: Architecture and specification, 1999. OMG document/99-10-07, v2.3.1.

[4] W. Edwards. *Core JINI*. Prentice-Hall, 1999.

[5] R. Ierusalimschy. *Programming in Lua*. lua.org, second edition, 2006.

[6] C. Ururahy, N. Rodriguez, and R. Ierusalimschy. ALua: Flexibility for parallel programming. *Computer Languages*, 28(2):155–180, December 2002.

[7] M.A. Leal, N. Rodriguez, and R. Ierusalimschy. LuaTS - A Reactive Event-Driven Tuple Space. *Journal of Universal Computer Science*, 9(8):730–744, August 2003.

[8] N. Carriero and D. Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[9] G. P. Picco, A. Murphy, and G. Roman. LIME: Linda meets mobility. In *21st Intl Conf. on Software Engineering (ICSE'99)*, pages 368–377, Los Angeles, CA, 1999.

[10] C. Fok, G. Roman, and G. Hackmann. A lightweight coordination middleware for

mobile computing. In *6th Intl Conf. on Coordination Models and Languages (COORD'04)*, Pisa, Italy, 2004.

[11] A. Tanenbaum and R. van Renesse. A critique of the remote procedure call paradigm. In *EUTECO'88 Conf.*, pages 775–783, Vienna, 1988. Participants Edition.

[12] K. Birman and R. van Renessee, editors. *Reliable Distributed Computing with the Isis Toolkit*, chapter RPC considered inadequate, pages 68–78. IEEE Computer Society Press, 1994.

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

[14] H. Lieberman. *Object-Oriented Concurrent Programming*, chapter Concurrent Object-Oriented Programming in Act 1, pages 9–36. The MIT Press, 1987.

[15] A. Yonezawa, J. P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. *SIGPLAN Notices (OOPSLA'86 Proceedings)*, 21(11):258–268, November 1986.

[16] E. Toernig. C Coroutines(coro library), 2000. `www.goron.de/~froese/coro/coro.html`.

[17] G. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Petrocelli Charter, 1975.

[18] A.L. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, July 2004.

[19] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, 2003. Usenix.

[20] N. Schemenauer, T. Peters, and M. Hetland. PEP 255 Simple Generators, 2001. `www.python.org/peps/pep-0255.html`.

[21] D. Conway. RFC 31: Subroutines: Co-routines, 2000. `dev.perl.org/perl6/rfc/31.html`.

[22] D. Nehab. Luasocket: Networking support for lua, 2004. `luaforge.net/projects/luasocket/`.

[23] A. Adya, J. Howell, M. Theimer, W. Bolosky, and J. Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference*, pages 289–302, Berkeley, 2002.

[24] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In *Intl Conf. on Functional Programming*, pages 18–27, Paris, 1999.

[25] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *18th Symp. on Operating Systems Princ. (SOSP-18)*, pages 230–243, Banff, Canada, 2001. ACM.

[26] M. Serrano, F. Boussinot, and B. Serpette. Scheme fair threads. In *6th ACM-SIGPLAN Intl Conf. on Princ. and Practice of Declarative Programming*, pages 203–214, Verona, 2004. ACM.

[27] Martin Alt and Sergei Gorlatch. Future-based rmi: optimizing compositions of remote method calls on the grid. In *Euro-Par 2003*, volume 2790 of *Lecture Notes in Computer Science*, pages 682–693, Klagenfurt, Austria, 2003.

[28] D. Caromel. Toward a method of object-oriented concurrent programming. *Comm. of the ACM*, 36(9):90–102, September 1993.