Event-Driven Programming for Distributed Multimedia Applications

Alésio Pfeifer, Cristina Ururahy, Noemi Rodriguez, Roberto Ierusalimschy
{alesio, ururahy, noemi, roberto}@inf.puc-rio.br

Departamento de Informática – PUC-Rio
R. Marquês de São Vicente, 225, Rio de Janeiro – RJ, 22453-900, Brazil
Tel.: +55 (21) 3114-1500 r. 3504, Fax.: +55 (21) 3114-1530

Resumo. O interesse atual em computação distribuída para redes geográficas vem destacando a necessidade de um modelo de programação adequado a este ambiente. Devido à sua natureza assíncrona, a programação dirigida a eventos fornece um modelo apropriado ao tratamento de falhas e retardos, que são frequentes neste contexto. Neste trabalho nós propomos uma arquitetura para aplicações multimídia distribuídas baseada em um modelo de programação orientado a eventos. Para evitar problemas de sincronização, inerentes à programação com múltiplas *threads*, optamos por uma abordagem *single thread*, orientada a eventos, aliada ao uso de múltiplos canais de comunicação. Com essa abordagem, o programador pode definir procedimentos de manipulação apropriados para cada canal, permitindo que a aplicação processe concorrentemente fluxos de controle e de dados. Neste trabalho, discutimos este modelo de programação, apresentamos o sistema implementado sobre ele e descrevemos experiências com este sistema.

Abstract. Current interest in wide-area distributed computing has highlighted the need for an adequate programming model for this environment. Because of its asynchronous nature, event-driven programming provides a suitable model for dealing with the failures and delays that are frequent in this context. In this work we propose an architecture for distributed multimedia applications based on an event-driven programming model. To avoid the synchronization problems that are inherent to multi-threaded programming, the proposed architecture is based on a single-threaded structure. Instead of multithreading, we opted for the event-oriented approach allied to multiple communication channels with user-defined handling procedures to allow the application to deal concurrently with control and data streams. We discuss this programming model, present the system we have implemented based on this model, and describe the experience we have had with this system.

keywords: multimidia distributed systems, mobile code, asynchronous communication, event-driven programming, interpreted language

1 Introduction

Over the last years, we have seen a shift in the focus of distributed computing from local-area to large-area networks. However, the programming model that has been largely successful in the context of NOWs (networks of workstations), which is the client-server model, does not adapt itself well to wide-area distributed systems. In geographically distributed networks, there is no guarantee about the state of the resources, and the synchronous nature of client-server

computing becomes inconvenient. This has led to a renewal of the discussion about alternative paradigms for distributed programming [1, 2, 3].

The event-oriented programming paradigm has been gaining importance over the last years, due, specially, to the growth in systems with graphical interfaces. It is now common for applications to be written in an interface-driven style, which has shifted the control of application flow from the program to the user: Instead of using code that dictates the flow of execution, the programmer writes code to react to user input, letting the user direct the flow of control.

Several groups have investigated the applicability of the event-oriented paradigm to concurrent and distributed programming. Early work on Tcl/Tk discusses this communication model as a technique to exchange information between different applications running on a single machine [4]. In this case, events are chunks of code that are executed upon being received. The Glish system uses scripts as a glue between compiled components, and these components send information to the controlling scripts through events [5]. More recently, reactive coordination models make it possible for the programmer to define activities (reactions) to be triggered by the occurrence of a communication event on a Linda-style tuple space [6, 7, 8].

Work with the ALua system has investigated the extent to which the event-oriented model can be used as a basic model for distributed computing [9]. Messages exchanged by ALua processes are regarded as chunks of code to be executed by the receiver. The arrival of a message through the network is regarded as an event that can trigger state transitions. An ALua process contains a single thread that handles each event to completion before it receives the next event; thus, as in a program with a graphical interface, the structure of an ALua program is a set of event-handling functions that modify the current execution state.

This architecture has been successfully employed to build different distributed and parallel applications [9]. In most cases, ALua applications use a dual programming language model, in which a scripting language, Lua [10], is used to coordinate the interaction between components written in C. Lua code handles all communication among processes (and therefore defines the architecture of the application), while C functions handle the CPU-intensive tasks in each process.

We have identified a new class of applications that can benefit from ALua's programming model: multimedia applications such as VoD (Video-on-Demand), videoconference, or videotelephone. Leite and colleagues describe a distributed video service that has been implemented with ALua [11]. In this kind of application, the transmission of Lua code is useful for transmitting control information, allowing agents to asynchronously receive commands and to exchange code. However, in these applications agents also need to exchange large streams of data. This typically involves synchronous (blocking) operations which do not match well with ALua's original architecture: If an agent becomes blocked on an synchronous I/O operation, it cannot complete the execution of the current event, and pending control events will not get a chance to be processed. Thus, an agent may deadlock waiting for data that will never arrive, or react too slowly to user commands.

One possible solution to this problem would have been to introduce multithreading in ALua, reserving a separate thread for control commands. However, this would conflict with ALua's basic design choices.

In this paper, we propose an event-driven architecture that allows us to deal with both kinds of messages — control and data — in one single asynchronous programming model. We have extended the original ALua system according to this new model. We opted for a solution that maintains the single-threaded structure of ALua agents, but at the same time allows agents to

handle different communication channels concurrently. This structure fits in well with the requirements of distributed multimedia applications, since it allows the agents to asynchronously receive different kinds of information. At the same time, we avoid the synchronization problems that are inherent to multi-threaded programming.

The remainder of this work has the following structure. In the next Section, we describe the original ALua system. Then, in Section 3, we discuss our solution for dealing concurrently with multiple types of messages. Section 4 highlights some characteristics of the multiple channel ALua architecture and presents a simple example program. Section 5 describes a simple multimedia application using the new system. Next, Section 6 presents some preliminary performance measures. Finally, Section 7 contains some concluding remarks.

2 ALua — Basic Design and Examples

A program in the ALua system is composed of several processes (also called *agents*), which may run in different hosts. ALua follows the event-driven design first advanced by GUI systems: Each agent runs an event loop, and only acts as a reaction to some event.

There is only one communication primitive between agents, a *send* operation, which generates an event in the receiver process (this is similar to the communication model used in [12], for instance). There is no explicit operation to receive messages. The receiver process gets its messages through its intrinsic event loop, like any other event (such as a mouse-move or a key-press event).

What makes ALua quite flexible is what is inside each message, and how receivers react to communication events. In ALua, each message is a piece of code (written in an interpreted language). When an agent receives a message, it immediately runs its contents. This results in a programming model with the same character of interpreted languages: Not very secure, but highly flexible. Each agent runs a single thread. The event-driven architecture provides responsiveness, while avoiding internal concurrency and its inherent problems.

Next we will see how to implement some common tasks using this model. In our first example, agent `A` simply sets a value in agent `B`:

```
alua.send("B", "n = 50")
```

This chunk of code will send the message `n = 50` to agent `B`. When `B` receives this message, it will execute it, setting its global variable `n` to 50.

Now, if `A` wants to run some function `foo` over the value of `B`'s variable `n`, it does the following:

```
alua.send("B", "alua.send('A', format('foo(%d)', n))")
```

The message that `B` gets is

```
alua.send('A', format('foo(%d)', n))
```

Assuming that `n` is still 50, the string `'foo(50)'` will be the result of the `format` command, and this is the message `B` will send to `A`. Finally, when `A` gets the message and runs it, it will call function `foo` with 50 as an argument.

It is interesting to note that, in our previous example, `foo` plays the role of a *continuation* for process `A`; that is, it encompasses what `A` wants to do upon receiving the value of `n`. This continuation-passing style of programming is very frequent in ALua. It allows `A` to wait for its answer without blocking. Between the time `A` sends its message to `B` and the time it gets its result, `A` is free to handle any other incoming event.

Another useful way to structure a program in ALua is as a state machine. To illustrate this technique, let us suppose agent A needs to send a message to two other agents, B and C, and then terminates its own execution when it receives acknowledgment from both of them. A script for that task follows:

```
function firstAnswer ()
  answer = secondAnswer
end

function secondAnswer ()
  print("The End")
  exit()
end

answer = firstAnswer

alua.send({"B", "C"}, "alua.send('A', 'answer()')")
```

When A runs this code, it sends to B and C the message `alua.send('A','answer()')` which makes both B and C send the message `answer()` back to A. When A receives the first answer, it will call function `firstAnswer`, which is the current value of variable `answer`. This function changes the value of the variable `answer`, so that the next message will call `secondAnswer`, and finally finish the execution of A.

Typically, when we run an ALua program, we create a "console agent", that simply opens an interactive console to the user, and executes any command typed into the console. Through this console, a user has the means to dynamically add or change functionality in other processes. For instance, let us suppose we have several agents running a program, and they all send their output through function `print`. Now, the user decides to redirect the output from all processes into its own console, prefixing each output message with an identification of its origin. All she has to do is to type the following code into her console:

```
prog = [[
  print = function (msg)
    alua.send("A",
          format("print('%s: %s\n')", alua.myname, msg))
  end
]]

alua.send(all, prog)
```

(The double square brackets `[[...]]` work as literal-string delimiters, mostly like double or single quotes, but they allow the string to span several lines.) Each process will eventually receive and run the code `prog`, which will redefine its `print` function. This new `print` function, when called, sends to process A (which runs the console) a message like `print('processname: msg')`.

The beauty of this facility is that the user does not need to anticipate such a change, and does not need to stop the program. The interpreted language allows her to change the program on the fly, with new code created on the fly.

3 Multiple Communication Channels

In the original ALua architecture, a single callback function is employed for any message arriving for a given agent. Because messages contain arbitrary Lua code, the use of a single

callback that executes this code has proved to be quite flexible in a range of applications [9]. However, looking at the demands of multimedia distributed applications, we identified the need for an agent to handle different messages in different ways. In such applications, processes typically exchange large data streams, and it would not be feasible to handle this exchange though messages containing Lua code. On the other hand, many of these applications would greatly benefit from ALua's facilities for code distribution and runtime configuration. We thus decided to extend the basic ALua model and allow it to define different handlers for different messages.

One important characteristic of the ALua model is its single-threaded execution model. This model imposes a programming discipline on the ALua programmer: The code invoked when a message is received must not execute blocking operations. If it does, the system will become blocked and will not handle new incoming events. When extending the ALua model to deal with data streams, this was an important consideration. The new architecture should maintain an event-oriented approach throughout the handling of all communication.

Once we had identified the need to handle different messages in different ways, the obvious question was how to associate messages to handlers. One convenient way to specify that a message is to receive a certain treatment (or service) is through the use of *communication channels* [13, 14]. In the new ALua architecture, communication channels became objects that the ALua programmer can manipulate. An agent may now define as many communication channels as it needs and associate different callback functions to each of them. The event loop now observes the status of a set of communication channels, and triggers the callback functions associated to the ones that need handling. One channel — called the *control* channel — is regarded as having a distinguished status, maintaining the original function of exchange of Lua code. We will use the expression *Multi-channel ALua* to distinguish the new ALua architecture from the original one. Figure 1 sketches a possible configuration for an application running on Multi-channel ALua.

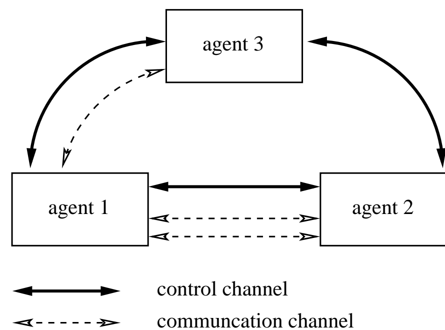

Figure 1: Agents using several communication channels.

The communication channels are implemented through LuaSocket [15]. LuaSocket is a library that exports Berkeley sockets to Lua. Integration with the LuaSocket library allows Lua code to manipulate sockets directly through a simplified interface. Our implementation of the ALua agent is itself mostly written in Lua with calls to LuaSocket, and the ALua programmer may also directly access the library if she so wishes.

A set of new ALua functions provides the programmer with the ability to manipulate channels. Function `alua.new_srvchannel` creates a new TCP server channel. It receives as arguments the port to which this channel must be bound and four callback functions: `onConnect`, `onRead`, `onWrite`, and `onClose`. When a new connection arrives at this channel, a new communication channel will be set up between the contacted and contacting agents. Function

`onConnect` will then be triggered. The three other functions passed to `alua.new_srvchannel` are automatically defined as callbacks for the new communication channel.

Function `alua.new_channel` creates a client connection to a TCP server. Besides a port and host, this function receives as parameters three callback functions — `onRead`, `onWrite` and `onClose` — that are to be invoked whenever the new channel is ready for reading, writing or when it is about to be closed. Function `alua.set_handler` allows the programmer to change the callbacks associated with a given communication channel at any point of execution. Function `alua.close_channel` will trigger the `onClose` function associated with the communication channel passed as a parameter, and close its connection. Finally, function `alua.send` has been changed to accept not only an agent's name, but also a communication channel as its first parameter.

In order to illustrate the use of communication channels, Figure 2 presents a simple multi-channel ALua program.

In this example, we show a client/server application, where the server basically sends the string "Hello!" through a communication channel. The ALua server creates the `Client` agent and sends through the control channel a message initializing some of its variables, followed by the code the client is going to execute (lines 24-29). Function `alua.spawn` is asynchronous and its second parameter is a callback function that will be executed only after the new agent is created and properly running. This prevents that messages are sent before the new agent is ready to receive them, without making the server wait for the client's creation to continue executing its code.

The server will then create a server TCP channel and define function `ConnectFunc` as the connect callback on this channel (lines 32-37). When the client creates a new connection, this will triggered function `ConnectFunc` on the server. When this happens, a new communication channel will be set up between server and client, and the server will finally send the string "Hello!" through this channel. After this, the server will again wait for a communication or control event.

On the other side, when the client receives its code (lines 4-13), it will create a TCP connection to the server and associate function `readFunc` as the read callback on this channel. When the communication channel is ready for reading, function `readFunc` will be called and the string *Client received Hello!* will be displayed on the screen (line 6) and, using the control channel, the client will send the code `Exit()` to the server (line 8), so that the server can terminate the program execution.

The new architecture makes ALua appropriate for applications which must transfer large streams of data because the receiver can invoke potentially blocking input operations inside a callback that is triggered only when there is data available to be read. This avoids the need for blocking on IO operations.

4 USING MULTIPLE CHANNELS

ALua's communication model, in which messages are regarded as chunks of Lua code, provides support for what is sometimes called *weak* code mobility [2]: the ability of an execution unit in a site to be bound dynamically to code coming from a different site.

A major asset provided by code mobility is service *customization* [16]. In conventional distributed systems that follow the client-server paradigm, servers provide an a priori fixed set of services accessible through a statically defined interface. It is often the case that these sets of services, or their interfaces, are not suitable for unforeseen client needs. A common solution to

```
 1 PORT = 6020
 2
 3 client_code = [[
 4 do
 5    local readFunc = function(ch, buffer)
 6       print("Client received " ..  buffer)
 7       alua.close_channel(cchannel)
 8       alua.send(alua.myparent,"Exit()")
 9    end
10    cchannel = alua.new_channel(HOSTNAME, PORT,
11                                readFunc, nil)
12    print("Client is ready.")
13 end
14 ]]
15
16 function Exit( hosts)
17   alua.close_channel(schannel)
18   alua.exit_all()
19   print('The End')
20   exit()
21 end
22
23 do
24    alua.spawn({"Client"}, function (hosts)
25       local defines=format("PORT = %d; HOSTNAME = '%s'",
26                              PORT, alua.localhost)
27       alua.send("Client", defines)
28       alua.send("Client", client_code)
29    end)
30
31    -- Server Code
32    local ConnectFunc = function(ch)
33       alua.send(ch,"Hello!")
34       alua.close_channel(ch)
35    end
36    schannel = alua.new_srvchannel(PORT, ConnectFunc,
37                                   nil, nil, nil)
38    print("Server is ready.")
39 end
```

Figure 2: A simple multi-channel ALua program.

this problem is to upgrade the server with new functionality, thus increasing both its complexity and its size without increasing its flexibility.

The ability to request the remote execution of code helps to increase server flexibility without affecting the size and complexity of the server. In this case, in fact, the server actually provides very simple and low-level services that seldom need to be changed. The client can compose services to obtain a customized high-level functionality that meets its specific needs [16].

In the same way, components that are able to link code dynamically can extend the types of interaction they support, increasing system flexibility. The *Code on Demand* paradigm enables the components to retrieve code from other remote components, providing a flexible way to extend dynamically the behavior of a component [2].

Active networks is one area that can benefit from this type of flexibility [17]. One of its primary goals is to facilitate network evolution [18]. The ability to inject new code into devices, possibly built on top of a set of existing services, provides systems with the flexibility of evolving in unanticipated ways.

In ALua, the agents are able to link code dynamically, extending the types of interaction they support. As we discussed in the previous section, when a channel is created in ALua, the user can associate callbacks with some events. Any agent can dynamically change these callbacks. This ability adds to the system a flexible and powerful configuration mechanism. One agent can send to other agent(s) a chunk of code, defining a function, and the code to associate this new function with a channel callback. In this way, it can change the behavior of the channel to which it is connected, according to its needs.

In order to illustrate dynamic redefinition of callbacks in ALua, we present in Figures 3 and 4 a client-server application that sorts a sequence of numbers. Due to space restrictions, we left the initialization code of the agents out.

In Figure 3, the server creates a TCP channel and defines functions `ConnectFunc`, `ReadFunc`, and `WriteFunc` as the connect, read and write callbacks on this channel (lines 40-41). A new client connection will trigger function `ConnectFunc` on the server. When this happens, a new communication channel is set up between server and client, and the server initializes some data structures to manage the data exchange in that channel (lines 2-3). The server also defines a default sort function that will be used to sort the data exchanged via this channel (line 4). The client can change this function at any moment (as we will show later).

Whenever the client sends some data over the new channel, function `ReadFunc` will be triggered on the server. This function will insert the received data in the set associated with this channel (`ch.dataSet`), unless an EOS (end of sequence) terminator is received. In this case, the server will sort the data received in that channel and set a flag (`ch.haveData`) indicating that the sequence is already ordered (lines 7-15).

Whenever a channel is immediately available for writing, the function associated with write events is called. In our example, function `WriteFunc` will start to send the sorted sequence back to the client, one number at time, after the flag `ch.haveData` is set. When there is no number left to be sent, the server closes the channel (lines 17-25).

Figure 4 presents the client code. The client creates a TCP connection to the server and defines functions `ReadFunc`, `WriteFunc`, and `FinalFunc` as the read, write and close callbacks on this channel. The client also sets the sorting algorithm that will be used on the server. In our implementation, we use three sorting algorithms. The first one, implemented in the server, is an exchange sort implementation. The two others are mergesort and quicksort

```
 1 function ConnectFunc(ch)
 2   ch.haveData = nil
 3   ch.dataSet = {}
 4   ch.Sort = sort
 5 end
 6
 7 function ReadFunc(ch, data)
 8   local value = tonumber(data)
 9   if (value ~= EOS) then
10     tinsert(ch.dataSet, value)
11   else
12     ch.Sort(ch.dataSet, 1, getn(ch.dataSet))
13     ch.haveData = 1
14   end
15 end
16
17 function WriteFunc(ch)
18   if (ch.haveData) then
19     alua.send(ch, tostring(ch.dataSet[1]))
20     tremove(ch.dataSet, 1)
21     if (getn(ch.dataSet) == 0) then
22       alua.close_channel(ch)
23     end
24   end
25 end
26
27 function sort(a, low, high)
28   local minv, pos
29   for i = 1, high do
30     minv = a[i]; pos = i
31     for j = i, high do
32       if (a[j] < minv) then
33         minv = a[j]; pos = j
34       end
35     end
36     a[i], a[pos] = a[pos], a[i]
37   end
38 end
39
40 server = alua.new_srvchannel(port, ConnectFunc,
41                              ReadFunc, WriteFunc, nil)
42 print("Server is ready")
```

Figure 3: Sorting a sequence of numbers: Server code

[19, 20]. The client chooses the sorting algorithm based on the value of the `sortalg` variable, which receives a random value when the master agent creates each client. The `dataSet` variable contains the sequence to be sorted, while the `ordSet` variable will contain the ordered sequence sent by the server.

Before sending the sequence of numbers to the server, the client decides whether it is going to redefine the server's default sorting algorithm or not (based on `sortalg`). Depending on the value of the variable `sortalg`, the client will choose a new sort algorithm among mergesort, quicksort in ascending order or quicksort in descending order (lines 30, 33, and 36).

The quicksort algorithm is implemented to sort the sequence in ascending order, but the client can redefine the `compare` function to sort the sequence in descending order (lines 37-39).

To redefine the sorting function of a channel, the client uses function `alua.addFunction` (lines 31, 34, and 40). This function is to be executed in the server and receives four parameters. The first two are the Internet address (IP) and the port that identifies this channel. The third parameter is the field of this channel that is to be redefined, and the last parameter is a string that defines the new function.

If the client decides to change the sorting algorithm for its channel, it will send a control message to the server with the corresponding code (line 43-45).

When a write event happens, the `WriteFunc` function will be trigged and the client will send a number of the sequence to the server (lines 5-14). After the whole sequence has been sent (line 8), the client sends a EOS (end of sequence) value to the server (line 9) and sets a terminator indicating the end of the sequence (line 10). When the communication channel is ready for reading, function `ReadFunc` will be trigged and the value received from the channel will be saved in the `ordSet` variable (line 2). Finally, when the communication channel is about to be closed, function `FinalFunc` will be trigged and the values kept in the `ordSet` variable will be printed (lines 16-23).

We can extend the ideas used in this small example to a more realistic setting in the context of a multimedia application. To do this, we report to the work of Baldi and colleagues [21], who investigate the benefits that mobile code and active networks may bring to the application domain of videoconferences. The architecture they propose relies on three points in order to provide customization and scalability. The first of these points, and the one most related to mobile code facilities, is enabling the user to "upload" application code into the server, thus changing its behavior and *customizing* it to her needs. This facility would be directly available in any application developed with ALua: The multiple channel architecture allows different code and behavior to be defined by each client that connects to the server. The possibility of defining new functions to be used for reading and writing data to sockets allows even buffering and discarding to be defined on a per-channel basis.

Although the other points in Baldi's architecture are not strongly related to the focus of this paper, it is interesting to observe how ALua stands in regard to them. The second point, more related to active networks, is running the server on intermediate nodes of the network, where it can use the information managed by the device to become aware of the status of the network and *adapt* to it; doing this in ALua would, of course, depend on support for running Lua at intermediate nodes. Lua would be an appropriate tool for this purpose, since it is small and has a small memory footprint. Besides, Lua is extremely easy to interface with existing services, and can be used as a glue language between predefined components [10]. Also, the event-driven nature of ALua makes it convenient to write applications that receive notifications

```
 1 function ReadFunc(ch, data)
 2   tinsert(ordSet, data)
 3 end
 4
 5 function WriteFunc(ch)
 6   if (haveData) then
 7     alua.send(ch, tostring(dataSet[i]))
 8     if (i == getn(dataSet)) then
 9       alua.send(ch, tostring(EOS))
10       haveData = nil
11     end
12     i = i + 1
13   end
14 end
15
16 function FinalFunc()
17   local str = format("%d", ordSet[1])
18   for i = 2, getn(ordSet) do
19     str = format("%s, %d", str, ordSet[i])
20   end
21   print(format("sorted array [%s]", str))
22   alua.exit()
23 end
24
25 dataSet = {310, 285, 179, 652, 351, 423, 861, 254, 450, 520, 310}
26 i = 1; ordSet = {}; haveData = 1
27 ch = alua.new_channel(host, port, ReadFunc, WriteFunc, FinalFunc)
28
29 local code; local ip, p = ch:getsockname()
30 if (sortalg == "quicksort") then
31   code = format("alua.addFunction('%s', %d, 'Sort', %s)",
32                 ip, p, quicksortDef)
33 elseif (sortalg == "mergesort") then
34   code = format("alua.addFunction('%s', %d, 'Sort', %s)",
35                 ip, p, mergesortDef)
36 elseif (sortalg == "qsm") then
37   code = [[ compare = function(a, b)
38                return a < b
39              end ]]
40   code = code..format("alua.addFunction('%s', %d, 'Sort', %s)",
41                       ip, p, quicksortDef)
42 end
43 if (code) then
44   alua.send(server, code)
45 end
```

Figure 4: Sorting a sequence of numbers: Client code

about necessary adaptations. The third point in Baldi's design principles is enabling the server to *migrate* to a different node as a consequence to adaptation. Although ALua processes cannot explicitly migrate, it is easy to write an application in which the server sends its state and code to another ALua process.

5 Experience with RTP

To experiment with Multi-channel ALua, we have integrated an RTP library to it. RTP (Real-time Transport Protocol) was developed in the Audio-Video Working Group of the IETF, and has been published as an RFC [22]. RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data, over multicast or unicast network services. RTP does not address resource reservation and does not guarantee quality-of-service for real-time services. The data transport is augmented by a control protocol (RTCP) to allow monitoring of the data delivery in a manner scalable to large multicast networks, and to provide minimal control and identification functionality [22].

We used an RTP library that was developed at Lucent Technologies and is freely distributed [23]. The library provides a high level interface for developing applications that make use of RTP and offers a large number of functions. Only a few of those have been incorporated in this first experiment with Multi-channel ALua.

The RTP functions we added to ALua include functions for initializing the library, setting up addresses on which RTP data should be sent and received, and actually sending and receiving data. The core functionality of the protocol is implemented by functions which receive as a parameter the data that has been read from a socket. In our ALua+RTP agent, the read callback function associated with an RTP socket reads the data from the socket and then invokes the appropriate RTP library function. In this way, the RTP library functions, originally designed to be invoked in a synchronous environment, and the ALua programming model were integrated.

This integration of RTP into ALua environment enables the agents to exchange real-time data with each other. To experiment with Multi-channel ALua and RTP, we have set up a sample video application. This application involves servers, which hand out JPEG streams obtained in real time, clients, which receive such streams and display them on the screen, and proxies, which are clients that are capable of forwarding the received data to other clients.

A master ALua agent controls the application. As execution progresses, this agent spawns ALua agents on participating machines and configures them, using the control channel, to act as servers, clients, or proxies.

The servers obtain real time JPEG streams from a SunVideo subsystem [24]. The SunVideo subsystem is a real-time video capture and compression subsystem for Sun SPARCstations, which captures, digitizes, and compresses video signals from video sources such as video cameras. It is designed to work closely with applications that use the facilities of the XIL Imaging Library, which in turn provides functions to decompress and display video [25].

In our experiments we are using JPEG baseline sequential. The agents are running in Sun and Linux platforms. The agent configured as server captures the video, compresses it and sends it across the network to other agents. The clients receive the data and use XIL routines to decompress and display the data, reconstructing the image. Besides displaying the received data, the clients can forward the data, acting as proxies. We are using unicast communication to transfer the data flows between the agents.

Figure 5 shows a diagram of the architecture used in this experiment. We have not shown the control channels, as they are always available between any two agents. We have yet to

investigate this application further, tackling situations that demand more of the flexibility it offers and measuring performance. However, this initial experiment showed that the new multi-channel architecture results in an easy-to-use distributed programming tool that fits in well with real time video applications.

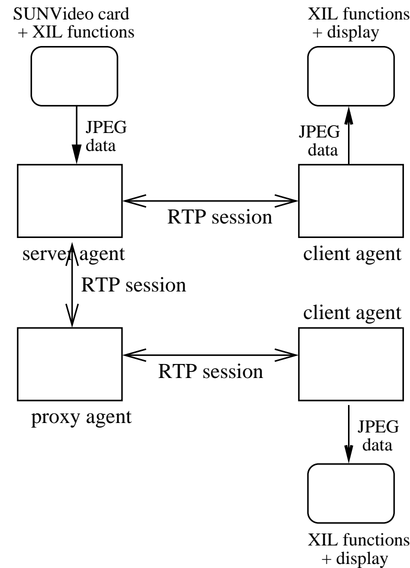

Figure 5: Agents exchanging control information and RTP messages.

6 Performance of Multi-channel ALua

To get a rough estimate of the performance of Multi-channel ALua, we ran a very simple experiment, in which an ALua program downloads a file from a server written in C and writes the received data to a new file. We compared the execution time of this ALua program to that of an equivalent program written in C and also to that of a third program written in standard (sequential) Lua.

We measured the execution time for this program and its C and Lua counterparts, downloading a file with approximately 80 megabytes. The results are shown in Table 1.

The observed performance of the ALua agent was very good. The average running time of the ALua program is less than 10% more than that of its C counterpart. The overhead of the event-driven model, in which a callback function is invoked each time new data is detected on the socket, is also very small, as can be seen by comparing the execution times of the sequential Lua program to that of the ALua agent.

	Average (sec)	Standard Deviation
C	26.3	0.3
Sequential Lua	27.9	0.2
ALua	28.7	0.3

Table 1: Execution times for clients downloading an 80Mb file.

As mentioned in the beginning of this section, this is a simple experiment and can only give us a general idea of Multi-channel ALua's performance. We must now extend our performance measures, mainly conducting experiments with more realistic settings, involving multimedia applications such as the one we described in Section 5.

7 FINAL REMARKS

In this work we discussed an architecture that allows a single-threaded application to deal concurrently with different streams of information and control. The implementation of the Multi-channel ALua system follows this architecture, associating the flexibility of transmitting Lua code through a main control channel to that of allowing the programmer to define functions for handling data streams on secondary channels.

The ability of an ALua agent to link code dynamically and redefine callbacks associated with channels adds to the system a flexible and poweful configuration mechanism. The user can change channel behavior and, in this way, customize the service according to her necessities. The customization can take place at different levels: The programmer may define a specific behavior for each channel or a single behavior to be used by the agent in any of the channels it establishes.

The integration of Multi-channel ALua with an RTP library allowed us to experiment with the architecture we proposed and observe its suitability in a multimedia application. The performance of Multi-channel ALua, although observed only in a preliminary experiment, appears to be quite promising.

We are now working on more elaborate experiments involving Multi-channel ALua and RTP. The development of different multimedia applications exploiting the potential of code mobility in terms of user customization will allow us both to conduct more elaborate performance measures and to evaluate the current programming interface.

REFERENCES

[1] A. Oram, Ed., *Peer-to-Peer : Harnessing the Power of Disruptive Technologies*, O'Reilly, 2001.

[2] A. Carzaniga, G. Picco, and G. Vigna, "Designing distributed applications with a mobile code paradigm," in *Proceedings of the 19th International Conference on Software Engineering*, 1997.

[3] U. Saif and D. Greaves, "Communication primitives for ubiquitous systems or RPC considered harmful," in *Proceedings of the International Workshop on Smart Appliances and Wearable Computing (IWSAWC)*, Mesa, Arizona, 2001, Held in conjunction with the 21st IEEE Intl Conf. on Dist. Computing Systems.

[4] J. Ousterhout, "Tcl: an embeddable command language," in *Proc. of the Winter 1990 USENIX Conference*. USENIX Association, 1990.

[5] V. Paxson and C. Saltmarsh, "Glish: a user-level software bus for loosely-coupled distributed systems," in *1993 Winter USENIX Technical Conference*, 1993.

[6] E. Denti, A. Natali, and A. Omicini, "On the expressive power of a language for programmable coordination media," 1998.

[7] G. Cabri, L. Leonardi, and F. Zambonelli, "Reactive Tuple Spaces for Mobile Agent Coordination," in *Proceedings of the 2nd International Workshop on Mobile Agents*, K. Rothermel and F. Hohl, Eds. 1998, vol. 1477, pp. 237–248, Springer-Verlag: Heidelberg, Germany, citeseer.nj.nec.com/cabri98reactive.html.

[8] B. Johanson and A. Fox, "Tuplespaces as coordination infrastructure for interactive workspaces," in *Proceedings of UbiTools'01 Workshop*, 2001, Held in conjunction with the Ubiquitous Computing Conference (UBICOMP).

[9] C. Ururahy and N. Rodriguez, "ALua: An event-driven communication mechanism for parallel and distributed programming," in *PDCS'99*, Fort Lauderdale, Florida, 1999.

[10] R. Ierusalimschy, L. Figueiredo, and W. Celes, "Lua—an extensible extension language," *Software: Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996.

[11] L. Leite, R. Alves, G. Lemos, and T. Batista, "DynaVideo - a dynamic video distribution service," in *6th Eurographics Workshop on Multimedia*, Manchester, UK, 2001.

[12] V. Barbosa, *An Introduction to Distributed Algorithms*, The MIT Press, 1996.

[13] G. Andrews, "Paradigms for process interaction in distributed programs," *Computing Surveys*, vol. 23, no. 1, pp. 49–90, 1991.

[14] G. Andrews and A. Schneider, "Concepts and notations for concurrent programming," 1983.

[15] D. Nehab, "Luasocket: IPv4 sockets support for the lua language," `http://www.tecgraf.puc-rio.br/~diego/luasocket/`.

[16] G. Vigna, *Mobile Code Technologies, Paradigms, and Applications*, Ph.D. thesis, Politecnico di Milano, Milano, Italy, Feb 1998.

[17] K. Psounis, "Active networks: Applications, security, safety, and architectures," *IEEE Communications Surveys*, pp. 1–16, First Quarter 1999.

[18] M. Hicks and S. Nettles, "Active networking means evolution (or enhanced extensibility required)," in *Second International Working Conference on Active Networks (IWAN 2000)*, Tokyo, Japan, Oct. 2000, pp. 16–32, Springer-Verlag, LNCS 1942.

[19] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1997.

[20] D. Knuth, *The Art of Computer Programming - v. 3 Sorting and Searching*, Addison-Wesley, 1998.

[21] M. Baldi, G. Picco, and F. Risso, "Designing a Videoconference System for Active Networks," in *Proceedings of Mobile Agents: 2nd International Workshop MA'98*. 1998, pp. 273–284, Springer-Verlag, LNCS 1477.

[22] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacoson, "RTP: A transport protocol for real-time applications," RFC 1889, Jan 1996.

[23] "Rtplib," `http://www.bell-labs.com/project/RTPlib/`.

[24] Sun Microsystems Computer Corporation, *Sun Video User's Guide*, Aug 1994.

[25] "Xil imaging library," `http://www.sun.com/software/imaging/XIL/xil.html`.