

ALua 3.0b

PUC-Rio, Abril de 2004

1 Introdução

O ALua [1] é uma extensão da linguagem Lua [2] cujo objetivo é avaliar, no desenvolvimento de aplicações distribuídas, a flexibilidade obtida pela combinação do paradigma de orientação a eventos com uma linguagem de programação interpretada.

Sua implementação se constitui de um mecanismo de comunicação entre processos remotos baseado em eventos. A comunicação se dá através da troca de mensagens, que são trechos de código na linguagem Lua, e são enviadas de forma assíncrona e recebidas como eventos de comunicação. Cada evento é tratado de forma atômica, ou seja, a mensagem recebida é completamente processada antes que a próxima seja examinada, eliminando assim a concorrência entre eventos. Um exemplo de uso do ALua é o espaço de tuplas reativo LuaTS [3] que visa avaliar a utilização de espaços de tuplas no contexto de aplicações orientadas a eventos.

Esse documento apresenta uma descrição sucinta sobre o sistema ALua e sobre como instalar e usar o ALua 3.0b, uma versão atualizada do ALua para o Lua 5.0. Essa versão ainda deve sofrer algumas mudanças estruturais e está em fase de teste. Por isso pedimos que qualquer problema que venha a ocorrer, seja na instalação ou no desenvolvimento de aplicações usando essa versão do ALua, seja remetido para *alua@inf.puc-rio.br*.

Gostaríamos de salientar a importância de estarmos trabalhando com um sistema completamente desenvolvido aqui na PUC-RIO, o que nos permite experimentar novas idéias e aperfeiçoá-las com o seu uso prático. Você que pretende usar o ALua é convidado a participar deste esforço rementendo-nos eventuais problemas ou dificuldades no seu uso e também possíveis sugestões de aprimoramento.

2 O Sistema ALua

Uma aplicação ALua é composta por um grupo de processos que rodam em múltiplas máquinas e se comunicam através de uma rede usando a biblioteca LuaSocket [4].

Um daemon chamado **alud** deve rodar em cada máquina. Este daemon é um servidor que age como um intermediário na troca de mensagens entre os processos de uma aplicação. Os processos existentes em uma máquina, que podem ser vários, se comunicam com o daemon através de um *canal de controle*. Ao **alud** cabe:

- Criar, configurar e finalizar os processos;
- Prover um espaço de nomes onde cada processo tem um nome único;
- Permitir que os processos troquem mensagens;
- Manter e fornecer as informações de status dos processos rodando na máquina;
- Criar uma sessão diferente para cada computação iniciada.

Cada processo possui um interpretador Lua e um loop de eventos que gerencia os eventos da rede de comunicação e da interface do usuário. Os processos se comunicam usando a operação “send” assíncrona. A chegada de uma mensagem é tratada como um evento, isto é, não há uma operação “receive” explícita.

O ALua trata as mensagens como pedaços de código Lua e os executa. Essa execução é feita de forma atômica, ou seja, cada mensagem recebida é executada por completo antes que a próxima mensagem seja tratada, eliminando a existência de concorrência em um mesmo processo. Em consequência disso, a estrutura do programa precisa ser especial: os trechos de código devem ser tipicamente pequenos e não bloqueantes para permitir o tratamento das outras mensagens que chegam e evitar a ocorrência de deadlock. Se uma ação maior for necessária, o processo pode quebrá-la em partes menores e enviá-las para ele mesmo por meio de novas mensagens. A ausência de concorrência simplifica aspectos da programação distribuída, uma vez que não há a necessidade de tratar questões de sincronização dentro de um mesmo processo.

2.1 Modos de comunicação entre os processos

Há três maneiras básicas de implementar a comunicação entre dois processos A e B de uma aplicação ALua:

1. O processo A envia uma mensagem que define um valor no processo B;
2. O processo A envia uma mensagem para B especificando a função que deverá ser executada em A quando B retornar a mensagem. De forma similar, outra técnica comum é incluir na mensagem um código que faça o processo enviar de volta o resultado de um processamento. O processo solicitante não fica bloqueado enquanto a resposta não chega. Dessa forma é possível simular um loop de espera sem bloquear o loop de eventos;
3. A cada mensagem recebida o processo A pode alterar a função de recebimento. Um exemplo é quando A precisa terminar uma execução depois de receber mensagens de um conjunto de outros processos.

Veja alguns exemplos:

No processo A:

```
tabA = {a, b, c}
msg = string.format("tabB=%s", alua.toString(tabA))
alua.send("B", msg)
```

O processo A irá criar no processo B a variável “tabB” com o valor da sua variável “tabA”. A função *alua.toString()* faz parte da API do ALua e permite serializar dados. Ela se faz necessária uma vez que as mensagens transmitidas entre os processos devem ser strings.

No processo A:

```
msg = [[alua.send ("A", "print(..processa(x)..)")]
alua.send("B", msg)
```

B irá executar *processa(x)* e enviará o resultado para A imprimir.

Exemplos mais completos podem ser encontrados em [5] e na própria distribuição do ALua, no diretório *alua/samples*. Uma aplicação exemplo é mostrada no final desse documento.

2.2 Canais de comunicação

Uma importante extensão feita no ALua foi a implementação de múltiplos canais de comunicação [5]. Originalmente existia apenas um canal de comunicação - o canal de controle entre um processo e o daemon - com um único tratador para as mensagens recebidas por um processo. Desse modo, todas as mensagens deviam ser tratadas da mesma forma o que, como já foi dito, consiste em executá-las como pedaços de código Lua.

Quando uma aplicação precisa manipular mensagens de diferentes modos essa extensão permite criar canais adicionais, associando a cada um deles o tratador apropriado. O canal original (canal de controle) ligado ao daemon local foi mantido com a função de trocar código Lua. Os canais adicionais estabelecem comunicação direta entre dois processos que podem estar ou não na mesma máquina; a comunicação através desse canal não é intermediada pelo daemon.

A implementação dos canais de comunicação segue o modelo cliente/servidor. Desse modo cada canal possui de um lado um servidor e de outro um cliente. Nos dois processos associados a um canal é possível definir manipuladores para os eventos de leitura, escrita e fechamento do canal. Do lado do servidor é possível ainda definir um manipulador para o evento de conexão de um cliente. Ao implementar as funções que serão tratadores de eventos é necessário considerar os argumentos que serão passados pelo gerente de eventos quando o mesmo chamar essas funções na ocorrência dos respectivos eventos. Nos eventos de escrita de dados, conexão de clientes e fechamento de um canal, o gerente passa como argumento o canal. No caso do evento de leitura, o canal e os dados

recebidos (caso um padrão para a recepção dos dados tenha sido especificado, como será mostrado na descrição da função *alua.setReadPattern*).

Uma observação importante deve ser feita com relação ao evento de escrita. Enquanto existir um manipulador definido para esse evento, o gerenciador de eventos irá chamá-lo, mesmo que isso implique em reescrever dados no canal. Isso significa que, se os mesmos dados não devem ser reescritos, o tratador do evento deve garantir isso, disponibilizando novos dados ou cancelando o tratamento do evento através da função *alua.setHandler*, que será descrita na seção 4.1.

2.3 Temporizadores

Outra extensão do ALua foi a inclusão do pacote *luaTimer* e a consequente adição de duas novas funcionalidades na API do ALua: inclusão e remoção de temporizadores associados a strings de código. O pacote *luaTimer* oferece funções que permitem manipular a criação, remoção e avaliação de temporizadores. Em linhas gerais, um temporizador consiste da especificação de uma frequência que define intervalos de tempo de expiração. O *luaTimer* permite criar, armazenar e remover temporizadores e oferece funções para checar os intervalos de expiração e o intervalo de tempo para a expiração mais próxima.

Usando as funcionalidades do pacote *luaTimer*, o ALua oferece uma interface que permite associar strings de código a frequências de tempo, de modo que essas strings serão executadas em intervalos regulares até que o temporizador correspondente seja removido.

3 Como instalar o ALua 3.0b

Para instalar o ALua 3.0b é necessário primeiro instalar a linguagem Lua 5.0 e o pacote *LuaSocket 2.0-alpha*. Para usar as funções de temporização é necessário instalar o pacote *luaTimer-1.0*. Os arquivos-fonte destes pacotes podem ser obtidos em <http://www.inf.puc-rio.br/alua/sources>.

3.1 Instalando o Lua 5.0

Informações detalhadas sobre o processo de instalação do Lua podem ser encontradas no arquivo 'INSTALL' fornecido em conjunto com a linguagem. Deve-se habilitar a opção de carga dinâmica de bibliotecas, alterando o arquivo 'config' conforme a própria instalação da linguagem orienta.

Mais adiante, será necessário o uso da ferramenta 'bin2c', que deve ser construída separadamente. Para gerá-la, vá ao diretório 'etc/' dentro da distribuição da linguagem e efetue 'make bin2c' na shell. Recomenda-se tomar nota das localizações dos arquivos de cabeçalho (headers) e das bibliotecas do Lua.

3.2 Instalando o LuaSocket 2.0 alpha

Primeiramente ajuste, no arquivo *Makefile*, as localizações da instalação do Lua 5.0 em seu sistema. Faça **make**, **make dyn** e **make install**. No diretório *luasocket* criado (default em *lua-5.0/luasocket*), edite o arquivo *luasocket.lua* e substitua 'luasocket' pelo caminho completo da biblioteca *libluasocket.so*.

Para testar a instalação do LuaSocket, chame o interpretador Lua e digite os comandos mostrados abaixo. Se nenhum erro for exibido, a instalação foi efetuada com sucesso.

```
[::~]lua
Lua 5.0 Copyright (C) 1994-2003 Tecgraf, PUC-Rio
> require (os.getenv("LUADIR").."/luasocket/luasocket")
> soc = socket.tcp()
> soc:close()
> os.exit()
```

3.3 Instalando o luaTimer

Edite o arquivo *Makefile* e altere a variável *LUA_DIR* para armazenar o caminho do diretório de instalação do Lua 5.0 em seu sistema. Faça *make*. Defina a variável *INSTALL_LUATIMER* com o local em que se desejar instalar o pacote faça *make install*. O local de instalação oficial é */jdiretório do Lua/luaTimer*.

Edite *luaTimer/src/luatimer.lua* (o script de carga dinâmica da biblioteca) e defina a variável *libname* com o caminho da biblioteca C dinâmica do *luatimer*, *libluatimer.so*.

3.4 Instalando o ALua 3.0b

Inicialmente, ajuste as variáveis contidas no arquivo *config* para descrever corretamente o ambiente de compilação. Neste arquivo devem constar as localizações da instalação do Lua e do LuaSocket. Faça *make*, e defina a variável de ambiente **ALUADIR** com a localização de *alua/bin*.

Na subseção 6.4 é mostrado um exemplo de execução de uma aplicação ALua. Esse exemplo serve para testar se o ALua está corretamente instalado.

4 A API do ALua 3.0b

Essa seção descreve as funções básicas da API do ALua 3.0b.

4.1 API geral

- **alua.spawn(nagents, completedFunc)**: Inicia os processos que serão usados pela aplicação. O primeiro argumento define o número de processos que deverão ser criados (pode ser um número ou uma tabela com o

nome dos processos). O segundo argumento é uma função que será executada quando todos os processos estiverem prontos para receber mensagens. Quando essa função é chamada, a tabela com todos os processos criados é passada como argumento.

- **alua.send_agentid(to, table)**: Quando os processos são criados, eles não conhecem os demais processos e só podem enviar mensagens para o processo 'master' (primeiro processo criado pelo ALua quando a aplicação é iniciada). Em alguns casos pode ser necessário que um processo conheça os demais. Essa função permite enviar a identificação dos processos listados no segundo argumento para os processos listados no primeiro argumento. Essa função é tipicamente chamada dentro da função passada como argumento para *alua.spawn* quando é o caso dos processos precisarem se conhecer mutuamente.
- **alua.setHandler(soc, mode, func)**: Define um tratador para um evento do canal passado como primeiro argumento. O evento, que pode ser : de chegada de dados ('Read'), escrita de dados ('Write'), fechamento de canal ('Close') ou de conexão de um cliente ('Connect'). é especificado pelo segundo argumento. O terceiro argumento é a função que implementa o tratador do evento. Para cancelar o tratamento de um evento deve-se passar como tratador o valor 'nil'.
- **alua.send(to, msg)**: Envia a mensagem 'msg' para um processo 'to'.
- **alua.mcast(table, msg)**: Envia a mensagem 'msg' para todos os processos na tabela 'table'.
- **alua.broadcast(msg)**: Envia a mensagem 'msg' para todos os processos criados pelo processo que faz a chamada da função.
- **alua.close(soc)**: Executa o tratador do evento 'Close' do canal passado e em seguida o fecha.
- **alua.exit(proc)**: Finaliza a execução do(s) processo(s) passado(s). O argumento 'proc' pode ser uma tabela, uma string ou ser omitido. No primeiro caso, o argumento deverá conter os nomes dos agentes que deverão ser finalizados. No segundo, conterà o nome do único processo que será finalizado e no último caso finalizará o próprio processo.
- **alua.exit_all()**: Finaliza a execução de todos os processos criados através da chamada de *alua.spawn*.
- **alua.toString(arg)**: Prepara o argumento 'arg' para ser enviado para outro processo. Esse argumento pode ser uma tabela, uma string (envolve a string com os delimitadores '[' e ']') para tratar o caso de uma string com mais de uma linha), ou um número.

Cada processo criado tem associado a ele as seguintes variáveis:

- **alua.mytid**: índice do processo;
- **alua.myname**: nome do processo;
- **alua.myhost**: máquina onde o processo está rodando;
- **alua.myparent**: nome do processo pai (não é definido para o processo master)
- **alua.myparenthost**: nome da máquina do processo pai (não é definido para o processo master)

4.2 API para canais TCP

- **alua.setReadPattern(soc, pattern)**: Define o padrão que será usado pelo gerente de eventos do ALua para receber os dados do canal passado. O argumento 'pattern' pode receber um dos seguintes valores: 'a' (recebe os dados escritos em um canal quando a conexão é fechada); 'l' (recebe cada linha de texto escrita no canal); um número (recebe o número especificado de bytes escritos no canal); 'nil' (o programador deverá implementar na função que trata o evento de leitura a recepção explícita dos dados no canal usando a API do LuaSocket). O valor default definido pelo ALua quando um canal é criado é 'l'.

Com exceção do último caso, o gerente de eventos irá receber os dados do canal de acordo com o padrão especificado e em seguida irá chamar a função definida pelo programador para manipular os eventos de leitura, caso ela tenha sido definida. Ao chamá-la, o gerenciador passará como argumento o canal e os dados recebidos. No caso do padrão ser 'nil', a função será chamada passando apenas o canal como argumento.

- **alua.getReadPattern(soc)**: Retorna o padrão usado para receber os dados em um canal.
- **alua.newServerTCP(port, readFunc, writeFunc, connFunc, closeFunc)**: Cria um canal servidor TCP na máquina local. O primeiro argumento especifica a porta onde o servidor ficará escutando as conexões dos clientes. Os demais argumentos definem as funções para tratar os eventos de leitura, escrita, conexão e fechamento do canal, respectivamente. Caso não se deseje tratar algum destes eventos, o argumento pode ser omitido ou preenchido com 'nil', caso devam ser seguidos por um argumento válido.
- **alua.newClientTCP(host, port, readFunc, writeFunc, closeFunc)**: Cria uma nova conexão cliente TCP e define os manipuladores dos eventos do canal, que como no caso do servidor podem ser omitidos ou preenchidos com 'nil'. Os dois primeiros argumentos especificam a máquina e a porta do canal servidor respectivamente.

4.3 API para uso de temporizadores

- **alua.insertTimer(cmd, freq)**: Insere um novo temporizador com a frequência informada, associando a ele o comando também passado como argumento. Retorna uma referência para o temporizador criado.
- **alua.removeTimer(timer)**: Remove o temporizador passado como argumento.

5 Um exemplo ALua

O código abaixo ilustra um exemplo de uma aplicação ALua.

```
PORT = 6020

-- Client code
client_code = [[
do
  -- Callback to "read" event
  local readFunc = function(ch, buffer)
    if buffer then
      io.write(alua.myname.." received " .. buffer.." \n")
    elseif err then
      error(err)
    end
    if buffer == "exit" then
      alua.close(ch)
      alua.send(alua.myparent, "Exit()\n")
    else
      alua.setHandler(ch, "Write", writeFunc)
    end
  end
end

  -- Callback to "write" event
  writeFunc = function(ch)
    alua.send(ch, alua.myname.. ": ACK \n")
    alua.setHandler(ch, "Write", nil)
  end
end

  -- Creates a new client side channel
  cchannel = alua.newClientTCP(HOSTNAME, PORT, readFunc, nil, nil)
  io.write(alua.myname.." is ready.\n")
end
]]

-- First exit function
function First_Exit()
  Exit = Last_Exit
end

-- Last exit function
function Last_Exit()
  alua.close(schannel)
  alua.exit_all()
  io.write('The End\n')
```

```

    alua.exit()
end

Exit = First_Exit

-- Callback function to be called after all process are
-- ready to send and receive messages
spawn_completed = function(nodes)
    local defines = string.format("PORT = %d; HOSTNAME = '%s'",
                                PORT, alua.getHostName())
    alua.mcast({" ClientA ", " ClientB "}, defines)
    alua.mcast({" ClientA ", " ClientB "}, client_code)
end

-- Start the process to be used in the application
alua.spawn({" ClientA ", " ClientB "}, spawn_completed)

-- Callback to "connect" event
local ConnectFunc = function(ch)
    alua.send(ch, "Ciao, ciao!!!\n")
end

-- Callback to "read" event
local ReadFunc = function(ch, data)
    io.write(alua.myname .. ":" .. data .. "\n")
    alua.send(ch, "exit\n")
    alua.close(ch)
end

-- Creates a new server side channel
schannel = alua.newServerTCP(PORT, ReadFunc, nil, ConnectFunc, nil)
io.write("Server is ready.\n")

```

Listing 1: Código ALua usando canais de comunicação.

6 Rodando uma aplicação ALua

Para rodar uma aplicação ALua é preciso especificar em qual(is) máquina(s) ela deverá rodar, iniciar o daemon *alua*d em cada máquina e por final a própria aplicação. O pacote ALua traz três arquivos de script que facilitam essas tarefas.

6.1 config

config [d daemonportnumber] master [slave1 slave2 ...]

Cria o arquivo de configuração *alua*cfg.lua especificando a porta que será usada para criar o daemon e a(s) máquina(s) que será(ão) usada(s) pela aplicação. Deve-se especificar o nome completo de cada máquina. A primeira máquina da lista deve ser a máquina a partir da qual a aplicação será iniciada. A porta padrão é 6070.

6.2 startd

startd [-v]

Inicia ou checa os daemons ALua. A opção **-v** checa o estado do(s) daemon(s) na(s) máquinas(s) listada(s) no arquivo de configuração *aluaconf.lua*. Todos os comandos recebidos e enviados pelo daemon são devidamente armazenados junto ao *system logger*.

6.3 run

run [d application_directory] [s startfile] [e execfile]

Inicia uma aplicação ALua. O argumento *application_directory* define o diretório com os arquivos da aplicação. O padrão é o diretório corrente, considerando-se assim que esse script está sendo chamado de dentro do diretório da aplicação. O argumento *startfile* é o arquivo executado depois que o agente principal é criado. Esse arquivo contém a aplicação ALua e o padrão é *master.lua*. O argumento *execfile* é o arquivo executado depois que o daemon cria um novo processo e define o tipo de agente que este irá rodar (o padrão é *agent*). Esse script deve ser chamado da máquina definida para o agente principal (primeira máquina da lista de máquinas).

6.4 Exemplos de execução

Definidos os arquivos com a aplicação ALua, o usuário deve executar o script *config* definindo as máquinas que serão usadas pela aplicação. Lembre-se de escrever o nome completo das máquinas. Em seguida deve-se iniciar os daemons em cada uma das máquinas usando o script *startd*. Por fim, o script *run* deve ser executado - a partir da primeira máquina listada - para iniciar a aplicação.

A seguir é mostrado um exemplo de configuração do ambiente ALua e a execução da aplicação mostrada na seção 5.

```
-----
-- Configurando seu ambiente...
-----
[verdi.inf.puc-rio.br:~] setenv LUADIR ~/lua-5.0
[verdi.inf.puc-rio.br:~] setenv ALUADIR ~/alua/bin

obs: Isso pode ser colocado no seu arquivo .cshrc ou .bashrc

-----
-- Verificando se a biblioteca luasocket está OK...
-----
[verdi.inf.puc-rio.br:~] lua
Lua 5.0 Copyright (C) 1994-2003 Tecgraf, PUC-Rio
> require (os.getenv("LUADIR").."/luasocket/luasocket")
> soc = socket.tcp()
> soc:close()
```

```

> os.exit()
[verdi.inf.puc-rio.br:~]

-----
-- Configurando a execução na máquina verdi.inf.puc-rio.br...
-----
[verdi.inf.puc-rio.br:~/alua/bin] config
config [d daemonportnumber] master [slave1 slave2 ...]
[verdi.inf.puc-rio.br:~/alua/bin] config verdi.inf.puc-rio.br
Creating file /home/b/silvana/alua/bin/aluacfg.lua ...

-----
-- Iniciando o daemon
-----
[verdi.inf.puc-rio.br:~/alua/bin] startd
Starting daemons at: verdi.inf.puc-rio.br
Please, wait...
OK: the daemon in verdi.inf.puc-rio.br:6070 is alive!!!
[verdi.inf.puc-rio.br:~/alua/bin]

-----
-- Rodando a aplicação ALua (~alua/samples/channel/master.lua)...
-----
[verdi.inf.puc-rio.br:~/alua/bin] run d $ALUADIR/./samples/channel/
[verdi.inf.puc-rio.br:~] Server is ready.
ClientB is ready.
ClientB received Ciao, ciao!!!
master:ClientB: ACK
ClientB received exit
ClientA is ready.
ClientA received Ciao, ciao!!!
master:ClientA: ACK
ClientA received exit
The End

```

6.5 Exemplo de aplicação usando a API de temporizadores

O código listado em 46 é um exemplo de uso da API de temporizadores do ALua.

```

-- Client code
client_code = [[
do
  io.write(alua.myname.." is ready.\n")
  -- Insert a timer with frequency 2 to send a message to parent
  timer = alua.insertTimer("alua.removeTimer(timer);

```

```

        alua.send(alua.myparent, 'Exit()')", 2)
end
]]

-- First exit function
function First_Exit()
    Exit = Last_Exit
end

-- Last exit function
function Last_Exit()
    alua.exit_all()
    --alua.removeTimer(timer)
    io.write('The End\n')
    alua.exit()
end

Exit = First_Exit

-- Callback function to be called after all process are ready
-- to send and receive messages
spawn_completed = function(nodes)
    local defines = string.format("HOSTNAME = '%s'", alua.getHostName())
    alua.mcast({"ClientA", "ClientB"}, defines)
    alua.broadcast(client_code)
    alua.insertTimer([[io.write("waiting message from clients to exit...\n")]], 1)
end

-- Insert a timer with frequency 3 to start clients
t_start = alua.insertTimer("alua.spawn({'ClientA', 'ClientB'},
    spawn_completed); alua.removeTimer(t_start); alua.removeTimer(t_wait)", 3)

-- Insert a timer with frequency 1 to show a message while the
-- clients are not started
t_wait = alua.insertTimer([[io.write("waiting to start clients...\n")]], 1)

io.write("\nServer is ready.\n")
-- Insert a timer with frequency 0.4 to show the message 'Hello!'
t_msg = alua.insertTimer([[io.write("Hello!\n")]], 0.4)

```

Listing 2: Código ALua usando temporizadores.

Executando este código, temos:

```

[verdi.inf.puc-rio.br:~] $ALUADIR/config verdi.inf.puc-rio.br
Creating file /home/b/alua/alua/bin/aluacfg.lua ...
[verdi.inf.puc-rio.br:~] $ALUADIR/startd
Starting daemons at: verdi.inf.puc-rio.br
Please, wait...
OK: the daemon in verdi.inf.puc-rio.br:6070 is alive!!!
[verdi.inf.puc-rio.br:~] $ALUADIR/run d $ALUADIR/../samples/timer/
[verdi.inf.puc-rio.br:~]
Server is ready.
Hello!

```

```
Hello!
waiting to start clients...
Hello!
Hello!
waiting to start clients...
Hello!
Hello!
ClientB is ready.
ClientA is ready.
Hello!
Hello!
Hello!
waiting message from clients to exit...
Hello!
Hello!
waiting message from clients to exit...
Hello!
The End
```

Referências Bibliográficas

- [1] C. Ururahy, N. Rodriguez, and R. Ierusalimschy. Alua: flexibility for parallel programming. *Computer Languages*, 28(2):155–180, Dezembro 2002.
- [2] R. Ierusalimschy, L. H. Figueiredo, and W. Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.
- [3] M.A. Leal, N. Rodriguez, and R. Ierusalimschy. Luats - a reactive event-driven tuple space. *J.UCS - Journal of Universal Computer Science*, 9(8):730–744, Agosto 2003.
- [4] D. Nehab. Luasocket-2.0-alpha. <http://www.tecgraf.puc-rio.br/diego/luasocket/new/>.
- [5] A. Pfeifer, C. Ururahy, N. Rodriguez, and R. Ierusalimschy. Event-driven programming for distributed multimedia applications. *SBRC/02*, Maio 2002.