4
Comunicação de Controle e Dados

O mecanismo de comunicação ALua, apresentado no capítulo anterior, traz uma contribuição para aplicações paralelas distribuídas. Primeiro, ele traz ao domínio de computação distribuída uma linguagem poderosa como Lua, que como uma linguagem de extensão pode ser usada para estender e configurar aplicações e permitir que o usuário final tenha a sua disposição uma linguagem de programação completa que o permite interagir com aplicações em tempo de execução. Além disso, ele também oferece um mecanismo de comunicação que pode ser visto como um modelo de coordenação bastante flexível e com fronteiras menos rígidas no que diz respeito a estrutura da aplicação e de seus componentes.

Mas enquanto o fato de poder comunicar código entre os componentes de uma aplicação traz poder e flexibilidade, o ALua, em sua arquitetura original, não é adequado para aplicações que também precisam comunicar grande quantidade de dados.

4.1
O ALua Multicanal

Após termos empregado o ALua com sucesso na construção de diferentes aplicações paralelas e distribuídas [35, 36, 15], resolvemos estender este modelo para lidar com dois tipos de mensagens – controle e dados – em um único modelo de programação assíncrono. O ALua multicanal [49, 18, 14] mantém a estrutura *single-threaded* dos processos, mas permite que os processos tratem diferentes canais de comunicação concorrentemente. Esta estrutura permite que os processos recebam assincronamente diferentes tipos de informações. Ao mesmo tempo, evitamos os problemas de sincronismo inerentes a programação *multi-threaded*.

O ALua multicanal é apropriado para aplicações que precisam transferir

grandes *streams* de dados, pois o receptor pode invocar operações de entrada potencialmente bloqueantes dentro de uma *callback* que só é disparada quando existem dados para serem lidos. Isto evita a necessidade de bloqueio em operações de E/S.

Uma vez que identificamos a necessidade de tratar mensagens diferentes de forma diferente, a pergunta óbvia era como associar as mensagens a seus "tratadores". Um modo conveniente de especificar que uma mensagem deve receber um certo tratamento (ou serviço) é através do uso de *canais de comunicação* [42, 50]. No ALua multicanal, canais de comunicação tornam-se objetos que o programador ALua pode manipular. Um processo agora pode definir tantos canais de comunicação quantos ele achar necessário e associar diferentes funções de *callback* a cada um deles. O *loop* de eventos passa a observar o estado de um conjunto de canais e dispara as funções de *callback* associadas aos canais que precisam de tratamento. Um canal – chamado canal de *controle* – tem um *status* diferenciado, mantendo a função original de troca de código Lua. Os demais canais são tratados como canais de dados. A Figura 4.1 mostra a estrutura de um processo ALua multicanal.

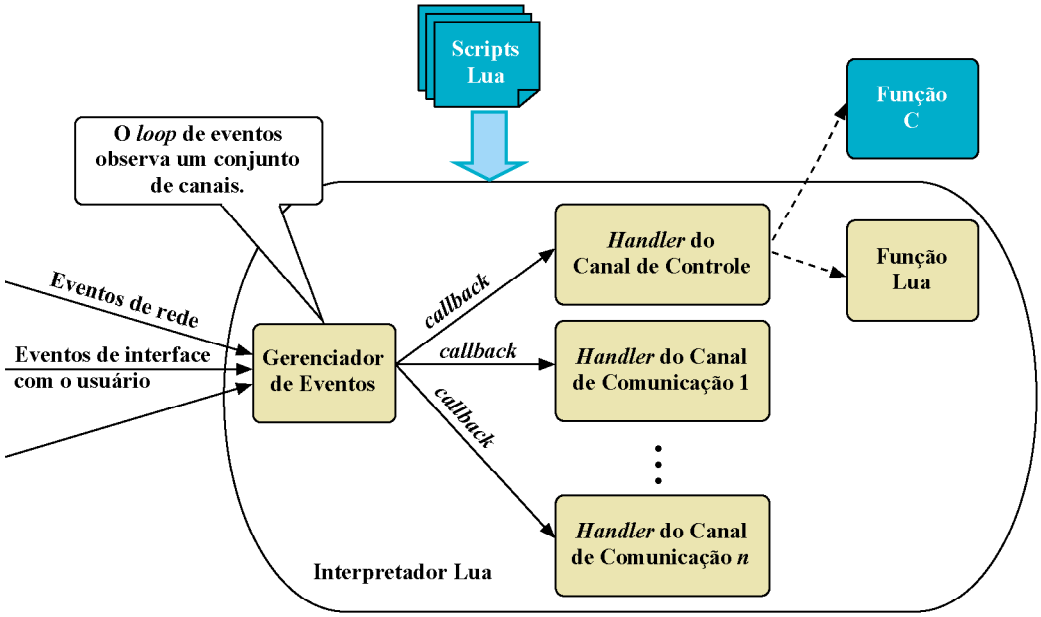

Figura 4.1: A estrutura de um processo ALua multicanal.

Um conjunto de novas funções ALua oferece ao programador a possibilidade de manipular *canais*. A função `alua.new_srvchannel` cria um canal servidor TCP. Ela recebe como argumentos a porta na qual se deseja receber conexões e quatro funções de *callback*: `onConnect`, `onRead`, `onWrite`, e `onClose`. Quando um pedido de conexão chega a este canal, um novo canal de comunicação será criado e a função `onConnect`

será então acionada. As três outras funções são automaticamente definidas como *callbacks* para os canais de comunicação criados.

A função `alua.new_channel` cria uma conexão cliente para um servidor TCP. Além de uma máquina e uma porta, esta função recebe como parâmetros as duas funções de *callback* — `onRead` e `onWrite` — que são chamadas sempre que um novo canal está pronto para ler ou escrever. É possível alterar as *callbacks* associadas a um determinado canal de comunicação em qualquer ponto da execução (ver Apêndice A). Finalmente, a função `alua.close_channel` irá acionar a função `onClose` associada ao canal de comunicação passado como parâmetro e fechará sua conexão.

Nos exemplos de uso do ALua, apresentados no Capítulo 3, pode-se observar que cada canal de controle é representado por um nome, i.e. uma *string*, que identifica o processo com o qual aquele canal está ligado. Quando se deseja transmitir uma mensagem de controle a um processo, esse nome é passado como primeiro parâmetro para a função `alua.send`. Já os canais de dados são representados por uma estrutura mais complexa, através de uma tabela Lua que contém funções de *callback*, um ponteiro para a estrutura C que representa um soquete, etc. Apesar das estruturas que representam canais de dados e de controle serem distintas, o fato de Lua ser uma linguagem com tipagem dinâmica nos permite criar uma API simplificada, onde tanto a string que representa um canal de controle quanto a tabela Lua que representa um canal de dados podem ser passadas como parâmetro para uma única função: `alua.send`.

Por outro lado, enquanto nos canais de controle não existe uma primitiva de recebimento de mensagens, dado que estas mensagens contêm código Lua e que o tratamento padrão é executá-las através do interpretador, nos canais de dados é muito difícil definir um comportamento único para o recebimento de mensagens. Por isso, a API do ALua multicanal oferece duas alternativas para o recebimento de mensagens dos canais de comunicação de dados. A primeira alternativa é ligar o flag `mustReceive`, que é um campo da tabela que define um canal de dados, para que o ALua se encarregue do recebimento de dados lendo uma linha de texto de cada vez. Neste caso, a linha de texto recebida é passada como segundo parâmetro para a *callback* `onRead`.

A segunda alternativa é utilizar a função `alua.receive` para ler dados diretamente dos canais de comunicação. Nesse caso é possível definir o padrão de leitura de dados que se deseja receber. Esse padrão pode ser uma única linha de texto, uma palavra (seqüência de caracteres não brancos) ou até mesmo um determinado número de bytes. Mas o programa deve chamar esta função apenas quando está certo

de que há dados para serem lidos, e que, conseqüentemente, o processo não ficará bloqueado numa chamada de `receive`. Tipicamente, a função de recebimento de dados `alua.receive` deve ser chamada de uma *callback* `onRead`, que é responsável pelo de tratamento de dados.

Para ilustrar o uso de canais de comunicação, a Figura 4.2 mostra um programa ALua multicanal simples. Neste exemplo, mostramos uma aplicação cliente/servidor, onde o servidor basicamente envia a string "Hello!" através de um canal de comunicação.

O servidor ALua cria um processo `Client` e envia, através do canal de controle, uma mensagem inicializando algumas de suas variáveis, seguida de outra mensagem com o código que o cliente irá executar (linhas 29–34). A função `alua.spawn` é assíncrona e seu segundo parâmetro é uma função *callback* que será executada somente quando o novo processo tiver sido criado e estiver executando devidamente. Isto evita que mensagens sejam enviadas antes que o novo processo esteja pronto para recebê-las, sem fazer com que o servidor espere a criação do cliente para continuar sua execução.

O servidor então criará o canal servidor TCP e associará a função `ConnectFunc` como a *callback* de conexão deste canal (linhas 36–41). Os parâmetros `nil` indicam que nenhuma *callback* para leitura e escrita foram definidas neste canal. Quando o cliente pedir uma conexão, um novo canal de comunicação será estabelecido entre o cliente e o servidor, o que acionará a função `ConnectFunc` no servidor. Quando isso acontecer, o servidor finalmente enviará a string "Hello!" através deste canal e em seguida o fechará. Depois disso, o servidor esperará novamente por um evento de comunicação ou de controle.

Do outro lado, quando o cliente receber seu código (linhas 4–17), ele criará uma conexão TCP para o servidor e associará a função `readFunc` como a *callback* de leitura neste canal. Quando o canal de comunicação estiver pronto para leitura, a função `readFunc` será chamada e a string "Hello!" será lida desde canal. A string *Client received Hello!* será exibida na tela (linha 8) e, usando o canal de controle, o cliente enviará o código `Exit()` para o servidor (linha 13), de modo que o servidor possa terminar a execução do programa.

A habilidade de configurar as *callbacks* adiciona ao sistema um mecanismo de configuração e reconfiguração dinâmica poderoso e flexível. Processos podem enviar a outros processos trechos de código, definindo uma função, e o código para associar esta função a *callback* de um canal. Desde modo, eles podem alterar o comportamento de um canal com o qual eles estão conectados, de acordo com suas necessidades.

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

Figura 4.2: Um programa ALua multicanal simples.

4.2 Desempenho do ALua Multicanal

Em [14], fizemos uma avaliação simplificada de desempenho do ALua multicanal. Desenvolvemos um experimento simples, onde um programa ALua faz o *download* de um arquivo de um servidor escrito em C e escreve os dados recebidos em um novo arquivo. Comparamos o tempo de execução deste programa ALua com um programa equivalente escrito em C e ainda com um terceiro programa escrito em Lua (seqüencial)[1]. Medimos o tempo de execução para este programa e seus equivalentes em C e Lua, fazendo *download* de um arquivo de 80MB. O desempenho do processo ALua foi muito bom. A média do tempo de execução do programa ALua foi menos de 10% maior que seu equivalente em C e menos de 5% maior que seu equivalente em Lua.

Neste trabalho, fizemos uma outra avaliação do desempenho de um programa ALua que utiliza múltiplos canais concorrentemente, fazendo o *download* concorrente de arquivos de 150Mb. Comparamos o desempenho deste programa com o seu equivalente em C, que usa uma thread para gerenciar cada um destes *downloads*, em contraposição ao modelo do ALua que utiliza uma única *thread*.

Ambas as versões conectam-se a servidores escritos em C e escrevem os dados recebidos em cada um dos canais (ou threads) em arquivos separados. Cada servidor é executado em uma máquina diferente e é responsável por transmitir um único arquivo. A Figura 4.3 apresenta o código ALua para este programa. Esta aplicação é muito simples e envolve um único processo ALua. Entretanto, mesmo assim, é interessante notar que a Figura 4.3 mostra o código completo que é necessário para executar a aplicação[2].

O programa ALua começará executando as linhas 49 – 58 da Figura 4.3. A função `alua.new_channel` criará `N` (linha 9) conexões TCP com `N` diferentes servidores e associará a função `readFunc` como a *callback* de leitura de cada um destes canais. Estes canais serão usados apenas para leitura. A chamada a `writeto` abrirá o arquivo `FNAME[i]` para escrita. Em seguida o programa fica passivo. Deste ponto em diante, a chegada de eventos irá dirigir a execução do programa.

Toda vez que o loop de eventos ALua detecta dados para serem lidos em um dos canais de `mychannels`, a função `readFunc` é chamada e irá ler um bloco de dados do canal e escrevê-lo em um arquivo. Em seguida, o programa irá novamente esperar por um evento de comunicação ou de controle. Quando o servidor termina o envio

[1] O código completo do servidor C e dos clientes C, Lua e ALua estão listados no Apêndice C.1.
[2] O código completo do servidor C e dos clientes C e ALua estão listados no Apêndice C.1.

```
 7 t1=time()           -- As linhas 1 a 6 contém a definição da função time.
 8
 9 N=4; PORT=6068
10 BLOCKSIZE = 2^13    -- 8K
11
12 FNAME  = {"/tmp/mchannel00",
13           "/tmp/mchannel01",
14           "/tmp/mchannel02",
15           "/tmp/mchannel03",
16          }
17 HOSTNAME  = {"n00.par.inf.puc-rio.br",
18              "n01.par.inf.puc-rio.br",
19              "n02.par.inf.puc-rio.br",
20              "n03.par.inf.puc-rio.br",
21             }
22
23 function final()
24    if nchannels == nil then nchannels = 0 end
25    nchannels = nchannels + 1
26    if nchannels == N then
27       print(format("Total time = %d seconds.", time() - t1))
28       exit()
29    end
30 end
31
32 do
33    local readFunc = function(ch)
34       local fd=ch.fdescr
35       local buffer, err = alua.receive(ch, BLOCKSIZE)
36       if buffer then
37          write(fd, buffer)
38       end
39       if err then
40          if err == "closed" then
41             writeto(fd)  -- close output file
42             alua.close_channel(ch)
43             alua.send(alua.myname,"final()")
44          else error(err)
45          end
46       end
47    end
48
49    mychannels={}
50    for i=1,N do
51       mychannels[i] = alua.new_channel(HOSTNAME[i], PORT, readFunc)
52       if mychannels[i] then
53          mychannels[i].mustReceive = nil
54          mychannels[i].fdescr = writeto(FNAME[i])
55       else
56          print("Could not create channel.")
57       end
58    end
59 end
```

Figura 4.3: Código ALua do programa de teste de desempenho.

dos dados do arquivo e fecha a conexão, o programa ALua termina executando a função `final`.

Neste exemplo, a função `final` poderia ser chamada diretamente de `readFunc`. Entretanto, para ilustrar o modelo de programação do ALua, escolhemos ter este processo enviando uma mensagem para si mesmo quando ele detecta o fim da conexão. A linha

```
alua.send(alua.myname,"final()")
```

envia a string `"final()"` através do canal de controle, fazendo com que ela seja executada quando recebida.

Medimos o tempo de execução para este programa e para o seu equivalente em C, usando múltiplas *threads*, fazendo o *download* de arquivos com 150 megabytes. Os resultados são mostrados na Figura 4.4.

O desempenho observado de um processo ALua foi muito bom. O tempo de execução médio de um programa ALua multicanal é menos de 1,5% a mais que o seu equivalente em C, para o *download* concorrente de 4 arquivos e menos de 4% para 20 arquivos.

	Nº de Arquivos	Média (sec)	Desvio Padrão
C	4	53.2	0.2
ALua	4	53.8	0.2
C	20	267.4	0.4
ALua	20	278.0	0.6

Figura 4.4: Tempo de execução para clientes fazendo *download* de arquivos de 150Mb em uma rede de 100Mbits/s.

Como mencionado no início desta seção, este experimento é simples e nos fornece apenas uma idéia geral do desempenho do ALua multicanal. No futuro pretendemos estender nossas medidas de desempenho, principalmente conduzindo experimentos com ambientes mais realistas, envolvendo aplicações multimídia, como a descrita em [14].

4.3 Aplicações Multimídia

O ALua multicanal permite que uma nova classe de aplicações possa se beneficiar do seu modelo de programação: aplicações multimídia como vídeo sob

demanda (VOD – *video on demand*), vídeo-conferência e vídeo-telefonia. Neste tipo de aplicações, a transmissão de código Lua é útil para transmitir informações de controle, permitindo que processos recebam comandos assincronamente e que troquem código. Entretanto, nessas aplicações processos também necessitam trocar grandes volumes de dados de forma contínua, o que tipicamente envolve operações síncronas e bloqueantes, que não combinam bem com a arquitetura original do ALua: Se um processo fica bloqueado em operações de E/S síncronas, ele não consegue terminar a execução do evento corrente, e eventos de controle pendentes não terão chance de serem processados. Portanto, um processo poderia entrar em *deadlock* esperando por dados que nunca iriam chegar, ou reagir de forma muito lenta a comandos do usuário.

Mas, ao contrário do que se espera a primeira vista, a estrutura *single-threaded* dos processos ALua aliada a capacidade de tratar diferentes canais de comunicação concorrentemente, combina bem com os requisitos de aplicações multimídia distribuídas, uma vez que permite que os processos recebam assincronamente diferentes tipos de informações.

O trabalho [51] descreve o DynaVideo, um serviço dinâmico de distribuição de vídeo que utiliza o ALua. Este serviço foi projetado para distribuir vídeo de um modo independente do formato do vídeo e para interagir com diferentes tipos de clientes. Sua principal característica é a possibilidade de configuração dinâmica do serviço para uma demanda específica.

A Figura 4.5 mostra a arquitetura do DynaVideo de forma simplificada. Ela é composta por um gerente, que controla a execução do serviço, servidores primários (SP) e servidores secundários (SS). Quando os clientes solicitam uma conexão, o gerente procura por um servidor com capacidade para atendê-los. Caso um servidor não seja encontrado, um servidor secundário é iniciado.

Servidores primários têm acesso direto às fontes de vídeo. Eles são capazes de capturar um fluxo de vídeo de uma fonte e transmiti-lo para o cliente. Por outro lado, servidores secundários não têm acesso direto às fontes de vídeo. Eles recebem os fluxos de dados de um servidor primário e os enviam para os clientes, servindo como refletores.

Enquanto os servidores primários foram implementados em C, os servidores secundários foram implementados em três versões: ALua (fazendo envio de código), Aglets (Java) [52], CORBA (LuaORB) [53]. Segundo a experiência descrita em [51], é fácil modificar os servidores secundários implementados em ALua sempre que necessário, sem interferir com outros módulos do DynaVideo, o que introduz flexibilidade à implementação desses servidores.

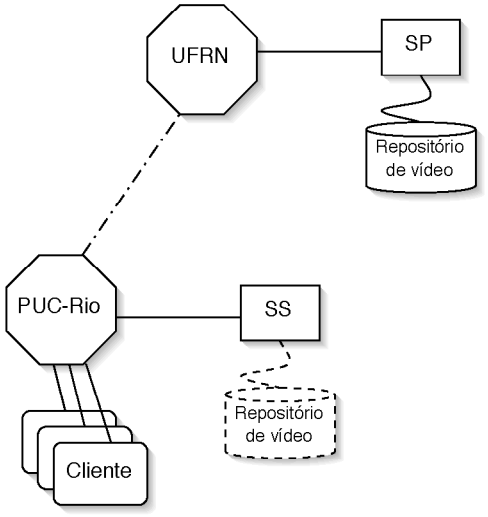

Figura 4.5: A arquitetura do DynaVideo.

A seguir, apresentamos pequenos trechos de código das implementações ALua e Aglets desenvolvidas no trabalho [51]. Essas implementações não foram desenvolvidas dentro do escopo dessa tese e são apresentadas apenas para efeitos comparativos. As conclusões apresentadas no final desta seção são referentes aos exemplos a seguir e não aos mecanismos oferecidos pelo ALua e por Aglets de uma forma mais genérica.

4.3.1
Versão ALua

O trecho de código a seguir é executado pelo servidor secundário ALua no momento em que ele cria uma conexão com o servidor primário. Todo fluxo de dados recebido será enviado para seus clientes.

```
function readData(soc)
   data = alua.receive(soc)
   if data then
      for i=1, getn(clients) do
         alua.send(clients[i], data)
      end
   end
end

src_soc, e = alua.new_udpsrv(server_port, readData)
if e then
   trata_erro(format("ERRO (%s) AO CRIAR CANAL UDP", e))
```

```
end

print ("Servidor Inicializado")
```

Através de uma mensagem de controle, o gerente pode solicitar a inserção de mais um cliente na lista do servidor secundário. Para isso, ele envia para o servidor secundário a mensagem:

`insert_client(<máquina do cliente>, <port>)`

Quando este código é recebido pelo servidor secundário, a função `insert_client`, apresentada a seguir, é executada.

```
function insert_client(client, port)
   local dst_soc, e = alua.new_udpclt(client, port)
   if e then
      trata_erro(format("ERRO (%s) AO CRIAR CANAL UDP", e))
   end
   dst_soc:setpeername(client, port)

   tinsert(clients, dst_soc)
end
```

4.3.2 Versão Aglets

A seguir, mostramos o trecho de código utilizado para iniciar o servidor secundário do DynaVideo na versão Aglets. Uma nova *thread* é criada para fazer a conexão com o servidor primário e receber os fluxos de dados que deverão ser retransmitidos para seus clientes.

```
public void iniciar_Servidor(){
   ...
   executando=true;
   Thread t=new Thread(){
      public void run(){
         DatagramSocket s=null;
         DatagramPacket p = null;
         executando=true;
         ...
            s = new DatagramSocket(4445);
         ...
            s.setReceiveBufferSize(64000000);
```

```
        ...
        byte[] bufin = new byte[1024];
        p = new DatagramPacket(bufin, bufin.length);
        ////////////////////////////////////////////////////////////
        // Após estabelecer a conexão com o servidor primário, passa //
        // a esperar as mensagens  que deverão ser retransmitidas.   //
        ////////////////////////////////////////////////////////////
        while(executando){
           ...
              s.receive(p);       // recebimento bloqueante.
           ...
           if(!(ips.isEmpty())){  // ips contém os clientes.
              Integer inteiro;
              InetAddress aux2;
              for(int i=0;i<ips.size();i++){
                 inteiro =(Integer)portas.elementAt(i);
                 aux2=(InetAddress)ips.elementAt(i);
                 p.setAddress(aux2);
                 p.setPort(inteiro.intValue());
                 ...
                    s.send(p);
                 ...
              }
           }
        }
        s.close();
        ...
     }
  };
  t.start();
}//EOF iniciar servidor
```

Para solicitar a inserção de mais um cliente, o gerente envia uma mensagem para o servidor, que será tratada pelo trecho de código abaixo. O servidor secundário Aglet fica em um *loop* a espera de comandos enviados pelo gerente, representados por um caractere.

```
////////////////////////////////////////
// loop que espera comandos do gerente //
////////////////////////////////////////
...
   if(x.charAt(0)=='1'){
   acrescentar_Cliente(buf);
   enviar_mensagem();
   }
...
```

```
   if(x.charAt(0)=='3'){
   iniciar_Servidor();
   enviar_mensagem();
   }
...
```

A função `acrescentar_Cliente`, que insere um cliente na lista de clientes para os quais o servidor secundário deve retransmitir os dados que recebe, é apresentada a seguir.

```
public void acrescentar_Cliente(byte[] b){
   ...
   InetAddress ipCliente=null;
   int i=2;
   String t = new String (b);
   byte[] buf_temp=new byte[20];
   while(t.charAt(i)!=' '){
       buf_temp[i-2]=b[i];
       i++;
   }
   String cliente=new String(buf_temp);
   i++;
   int y=0;
   byte[] buf_temp2=new byte[4];

   while(t.charAt(i)!=' '){
       buf_temp2[y]=b[i];
       i++;
       y++;
   }
   String porta=new String(buf_temp2);
   ...
       ipCliente=ipCliente.getByName(cliente);
   ...
   ips.addElement(ipCliente);
   portas.addElement(new Integer(porta));
   enviar_mensagem();
}//EOF Acrescentar_Cliente
```

Os trechos de código apresentados na Seção 4.3.1 mostram as funções completas `readData` e `insert_client` da versão ALua do servidor secundário. Na Seção 4.3.2, omitimos algumas partes das funções devido a sua complexidade.

A versão ALua oferece uma flexibilidade muito maior para a aplicação, uma vez que os comandos enviados pelo gerente não precisam ser conhecidos a priori pelo

servidor secundário. Ao iniciar o servidor secundário o gerente envia a string com o código que define a função `insert_client` e a qualquer momento o gerente pode redefinir esta função como melhor lhe convier.

Neste capítulo, mostramos que o ALua permite a uma aplicação *single-threaded* lidar concorrentemente com diferentes fluxos de dados e controle, associando a flexibilidade de se transmitir código Lua através de um canal principal de controle à possibilidade de permitir que o programador defina funções para tratar os fluxos de dados nos canais secundários.