Uma Experiência em Programação Distribuída Dirigida por Eventos

Renato Cerqueira
Noemi Rodriguez
Roberto Ierusalimschy

Departamento de Informática – PUC-Rio
Rua Marquês de São Vicente 225, Gávea
22453-900 Rio de Janeiro, RJ, Brasil

rcerq, noemi, roberto@inf.puc-rio.br

Resumo

Este documento apresenta um mecanismo de programação distribuída dirigida por eventos para a linguagem de extensão *Lua*. Procurou-se implementar um modelo simples, porém poderoso, de programação, que tirasse proveito de todas as facilidades oferecidas por uma linguagem de extensão como Lua. Para validar o mecanismo e experimentar a adequação do paradigma de orientação a eventos à programação distribuída, foram implementados alguns algoritmos clássicos de programação distribuída usando-o. Como a linguagem Lua não oferece originalmente nenhum suporte especial para programação orientada a eventos, o mecanismo foi implementado sobre o ambiente *Tche*, um ambiente de programação para esta linguagem que possui todo um suporte para programação orientada a eventos.

Palavras-Chave: Programação Orientada a Eventos, Programação Distribuída, Linguagens de Extensão

Abstract

This paper presents an event-driven distributed programming mechanism for the extension language *Lua*. We have focused on a simple, yet powerful, programming model, which can take advantage of the specific facilities of an extension language like Lua. In order to evaluate the mechanism and the adequacy of the event-driven approach to distributed programming, we have implemented some classical distributed algorithms using it. Because Lua has no specific support for event-driven programming, the mechanism has been implemented using *Tche*, an event-driven environment for Lua.

Keywords: Event-driven Programming, Distributed Programming, Extension Language

Introdução

Com a proliferação das redes de computadores e a sofisticação dos ambientes multitarefas, torna-se inquestionável a importância dos mecanismos de comunicação entre aplicações para garantir a interoperabilidade destas. No entanto, dependendo do tipo de interação desejada entre duas aplicações, a sua implementação adequada pode representar uma tarefa árdua para o desenvolvedor. Para que este problema seja solucionado, tem havido um grande esforço no desenvolvimento de *toolkits* para linguagens convencionais voltados à comunicação entre processos [Sun88], [Sud90], assim como na definição de novas linguagens que dêem suporte à programação paralela e/ou distribuída [BST89].

Por outro lado, temos o reconhecimento do uso de linguagens para extensão de aplicações como uma importante técnica de projeto, permitindo o desenvolvimento de aplicações com um elevado grau de adaptação e expansão pelo usuário [FIF94], [Ous94].

Neste artigo, é apresentado um mecanismo para programação distribuída que usa a linguagem de extensão Lua [IFF95], visando oferecer às aplicações que usam esta linguagem embutida um simples, no entanto poderoso, mecanismo de comunicação. No projeto do mecanismo, procurou-se tirar proveito de todas as facilidades oferecidas por uma linguagem de extensão como Lua.

Além disto, o mecanismo procura se adequar a outra característica marcante das aplicações atuais: a orientação a eventos. Para isto, as mensagens enviadas de uma aplicação para outra são tratadas pela aplicação receptora da mesma forma que um evento de interface com o usuário é tratado. Como a linguagem Lua não oferece nenhum suporte especial à programação orientada a eventos, usou-se o ambiente Tche, um ambiente gráfico para esse paradigma de programação com a linguagem Lua [Car95].

Também é relatada a experiência de implementação de alguns algoritmos clássicos de programação distribuída, usando o mecanismo apresentado. Para isto, foram escolhidos os algoritmos de filtros, *heartbeat* e *probe/echo* [And91]. Com isso, além de se validar o mecanismo, é possível uma melhor avaliação deste, facilitando uma análise comparativa com outros mecanismos. Cabe destacar que os algoritmos escolhidos são independentes do modelo de orientação a eventos e, a princípio, eles não tiram nenhuma vantagem deste paradigma.

Na próxima seção, apresentamos o mecanismo de comunicação proposto, assim como um breve resumo da linguagem Lua e do ambiente de programação Tche. Na seção seguinte, apresentamos o relato da experiência de implementação dos algoritmos escolhidos. Finalmente, a última seção é dedicada a algumas considerações a respeito do resultado do trabalho e sobre trabalhos futuros.

Comunicação em Lua

A Linguagem Lua

Lua é uma linguagem de extensão que, embora procedural, apresenta facilidades para descrição de dados. Em sua versão 2.1, Lua apresenta como principais extensões à sua versão original facilidades para orientação a objetos, e um mecanismo de *fallbacks* [IFF95]. Lua tem sido largamente utilizada em produção para: configuração pelo usuário, entrada de dados, descrição de interfaces, descrição de objetos da aplicação e armazenamento de *metafiles* gráficos estruturados.

Sendo uma linguagem embutida (*embedded language*), Lua não tem a noção de programa principal; ela só trabalha embutida em uma aplicação cliente. Lua é implementada como uma biblioteca de funções C para ser ligada aos seus clientes. Assim, uma aplicação que a utilize pode invocar funções para executar um trecho de código Lua, pode manipular variáveis Lua, e pode registrar funções C para serem chamadas por um código Lua.

Através do uso de funções C, Lua pode ser especializada para diferentes domínios, criando assim linguagens de programação especializadas que compartilham a mesma estrutura sintática.

Todos os comandos em Lua são executados em um ambiente *global*. Este ambiente, que mantém todas as variáveis globais e funções, é construído no início da aplicação cliente e persiste até o seu término. O ambiente global pode ser manipulado através de código Lua ou através do programa hospedeiro, que pode alterar ou consultar variáveis globais usando funções da biblioteca que implementa Lua.

Variáveis globais não precisam ser declaradas. Assume-se que qualquer variável é global, a menos que seja explicitamente declarada como local. Antes da primeira atribuição, o valor de uma variável é nulo.

A unidade de execução de Lua é chamada *módulo*. Um módulo pode conter comandos e definições de funções, e pode ser um arquivo ou uma cadeia de caracteres do programa hospedeiro. Quando um módulo é executado, primeiro todas as suas funções e comandos são compilados, e as funções são adicionadas ao ambiente global; só então os comandos são executados em ordem seqüencial. Todas as modificações que um módulo efetua no ambiente global persistem após o seu término.

Quanto ao seu sistema de tipos, Lua é uma linguagem dinamicamente tipada. Variáveis não possuem tipos; somente valores têm tipo.

Existem sete tipos básicos em Lua: valor nulo (nil), números de ponto flutuante (number), cadeias de caracteres (string), funções (function), dado do usuário (userdata) e tabelas (table).

Funções são tratadas como qualquer outro tipo de valor. Assim, funções podem ser atribuídas a variáveis, passadas como argumento para outras funções ou retornadas como resultado.

Quando uma função é definida em Lua, seu corpo é compilado e armazenado em uma determinada variável. Funções escritas em Lua podem ser chamadas e manipuladas da mesma forma que funções C. O tipo userdata é oferecido para permitir que ponteiros de C sejam armazenados em variáveis de Lua.

O tipo table implementa arrays associativos, isto é, arrays que podem ser indexados tanto por números quanto por strings. Portanto, este tipo pode ser usado não somente para representar arrays convencionais, como também tabelas de símbolos, conjuntos, estruturas (*records*), etc. Para representar uma estrutura, Lua usa o nome do campo como um índice. A linguagem suporta esta representação tratando a.name como *syntactic sugar* para a["name"]. Tabelas também podem possuir métodos. Como funções são tratadas como qualquer outro tipo de valor, um campo de uma tabela pode referenciar uma função. A forma t:f(x) é *syntactic sugar* para t.f(t,x), que chama o método f da tabela t passando esta como o primeiro parâmetro.

É importante destacar que tabelas são objetos, e não valores. Variáveis não podem conter tabelas, somente referências para elas. Atribuição, passagem de parâmetro e retorno de função sempre manipulam referências para tabelas, e não implicam em nenhum tipo de cópia. Além do mais, tabelas têm que ser explicitamente criadas antes de serem usadas.

Com relação a funções, cabe destacar que a passagem de parâmetros é feita por valor, e funções podem receber um número variável de parâmetros e retornar mais de um valor. Isto torna desnecessária a passagem de parâmetro por referência quando mais de um resultado precisa ser retornado por uma função.

O Sistema Tche

O sistema Tche (*Tiny Canvas-Hosted Environment*) é um ambiente visual para a criação de aplicativos Lua, que está incluído neste conjunto de ferramentas para dar suporte a programação orientada a eventos [Car95]. O sistema oferece diversos recursos para a utilização de objetos visuais, que podem ser manipulados diretamente pelo usuário, de forma a se alterar suas características e seu comportamento. Por ser escrito em Lua, o sistema é facilmente modificável, incentivando a programação exploratória.

Por se tratar de um ambiente baseado em eventos, Tche não conta com um programa principal, mas sim com diferentes objetos que passam a interagir entre si e com o usuário, a partir do momento em que são criados. A arquitetura baseada em tratadores de eventos de Tche permite ainda que se crie novos tipos de eventos (*dispatchers*) e os associem ao sistema.

O Mecanismo Proposto

Seria muito interessante ter disponível um mecanismo de comunicação entre processos que se enquadrasse com o modelo de orientação a eventos, dada a grande difusão das aplicações

desenvolvidas segundo este paradigma. Tendo isto como motivação, foi desenvolvido um mecanismo de programação distribuída dirigida por eventos para o ambiente Lua/Tche.

Como se trata de um mecanismo de comunicação a ser usado por uma linguagem de extensão, ele estará disponível para uso tanto pelo desenvolvedor da aplicação quanto pelo usuário final. Sendo assim, o mecanismo deve ser o mais simples possível para que se enquadre com o princípio básico de simplicidade de uma linguagem de extensão.

A idéia básica do mecanismo é tratar toda mensagem enviada de uma aplicação para outra da mesma forma que um evento gerado pelo usuário seria tratado. Assim, mensagens enviadas pela rede são encaradas como um outro evento qualquer.

Para que um processo envie uma mensagem, ou melhor, gere um evento de comunicação, foi criada a função send. Esta função recebe dois parâmetros:

- uma string com o nome que identifica o processo a receber a mensagem;
- uma string que representa a mensagem - o seu significado vai depender do tratamento dado a ela pelo processo que recebe o evento.

Com esta função, uma aplicação já pode gerar um evento para uma outra. Porém, para que uma aplicação possa tratar esses eventos, é necessário que ela defina a *callback* Receive. A função que a aplicação definir como sendo essa *callback*, será chamada pelo sistema gerenciador de eventos de Tche sempre que o *dispatcher* responsável pelos eventos de comunicação via rede identificar uma nova mensagem. O sistema, quando chamar esta função, também vai fornecer dois parâmetros: um identificando o processo que gerou o evento, e o outro contendo a mensagem enviada.

Cabe destacar que a função Receive é uma função *callback*, que só é chamada pelo sistema quando este identifica um evento de comunicação; a aplicação nunca chama esta função explicitamente. Desta forma as aplicações envolvidas no processo de troca de eventos não ficam bloqueadas, estando sempre prontas a responder a qualquer evento, ao contrário do que acontece na maioria dos outros mecanismos de comunicação entre processos.

Tanto a função send quanto o *dispatcher* responsável pelos eventos de comunicação via rede foram implementados em C, usando o mecanismo de *sockets* para envio de datagramas [CS93], [Ste90].

Já a *callback* Receive pode ser implementada pela aplicação como uma função em Lua ou em C. Desta forma, a aplicação que recebe a mensagem pode dar o tratamento que desejar a esta. Porém, o tratamento sugerido e usado em todos os exemplos aqui apresentados é encarar a string da mensagem como sendo um trecho de código Lua que será executado pelo interpretador Lua do processo receptor do evento. O trecho de código a seguir apresenta a definição da *callback* Receive sugerida e usada em todos os exemplos:

```
function Receive( sender, msg )
   dostring(msg)
end
```

A função dostring é uma função predefinida de Lua, que recebe uma string com um trecho de código Lua como parâmetro e executa este código. Cabe destacar que este código será executado no ambiente global de Lua. Este código pode alterar valores de variáveis e chamar funções já existentes no ambiente do processo, assim como pode definir novas variáveis e funções e usá-las. Isto dá ao mecanismo uma grande flexibilidade, já que uma aplicação pode usar outros processos para ajudá-la em alguma tarefa sem que esses outros processos tenham conhecimento prévio do código necessário para isto. Programas inteiros podem ser enviados de uma aplicação para outra desta maneira. Este mecanismo também permite que uma aplicação controle outra remotamente, através de algum protocolo conhecido pelas duas.

No caso de mensagens enviadas por uma aplicação que precisem de alguma resposta, a mensagem pode conter um comando explícito, ou melhor, uma chamada da função send, para retornar a resposta da mensagem.

Esse recurso de tratar a mensagem do evento como sendo um trecho de código Lua a ser executado torna o mecanismo muito poderoso e flexível. Porém, para determinados usos, seria necessário adicionar algum controle sobre o código a ser executado para garantir a integridade do sistema. Mas a idéia do mecanismo é que uma aplicação só deixe disponível através de Lua a funcionalidade que lhe interesse exportar.

Um mecanismo de comunicação entre aplicações através de uma linguagem de extensão baseado em uma função do estilo de send foi proposto anteriormente para a linguagem de extensão TCL [Ous90]. Neste mecanismo de TCL, a mensagem também é encarada como um trecho de código a ser executado pelo processo receptor. No entanto, esse mecanismo só trata a comunicação entre processos em uma mesma máquina, sem se preocupar com o aspecto de programação distribuída.

Avaliação do Mecanismo

Como forma de avaliação do mecanismo proposto, foram escolhidos alguns algoritmos clássicos de programação distribuída para serem implementados usando o mecanismo.

Os algoritmos escolhidos foram tirados do artigo "*Paradigms for Process Interaction in Distributed Programs*" de Gregory R. Andrews [And91]. Este artigo apresenta sete algoritmos clássicos em programação distribuída que representam classes de algoritmos freqüentemente aplicados a problemas reais. Desses algoritmos foram escolhidos três para serem implementados com o mecanismo proposto. Procurou-se escolher os algoritmos que mais pudessem contribuir

para a análise, seja pela sua complexidade ou por, a princípio, não se adequarem ao mecanismo proposto. Os três algoritmos escolhidos foram:

- algoritmo *probe/echo* (versão do algoritmo de busca em profundidade em grafos para programação concorrente);
- algoritmo *heartbeat*;
- algoritmo de filtros.

Na implementação do *heartbeat*, só o processo que inicia o algoritmo conhece previamente as funções utilizadas pela aplicação. Essas funções são passadas para os demais processos através das mensagens dos eventos de comunicação. Já para as outras duas implementações, cada instância do sistema Tche já tem registradas funções de suporte para o algoritmo, como um serviço oferecido pelo sistema. Neste caso, a aplicação só precisa fornecer o código necessário para que os algoritmos sejam aplicados corretamente em uma determinada tarefa.

Por problemas de espaço não é possível apresentar a descrição de todos os três algoritmos e suas respectivas implementações. Assim, só será apresentado o algoritmo *probe/echo*. Dado que sua implementação é a mais simples dos três, ele é mais adequado para ser usado como um primeiro exemplo da aplicação do método. As implementações dos outros dois algoritmos podem ser encontradas na referência [CRI95].

O algoritmo *probe/echo*

Algoritmos aplicáveis a árvores e grafos são de grande importância para a programação distribuída, desde que a estrutura da maioria dos sistemas distribuídos é um grafo onde processos são os nós e conexões entre processos são as arestas.

O algoritmo *probe/echo* é a versão para programação concorrente do algoritmo de busca em profundidade em grafos. Um *probe* é uma mensagem enviada por um nó a seus sucessores; um *echo* é a subsequente resposta. Como os processos executam concorrentemente, *probes* são enviados em paralelo a todos os sucessores.

Quando um nó recebe um *probe*, ele envia *probes* aos seus demais vizinhos e fica esperando as suas respostas (*echos*). Quando tiver recebido todos os *echos*, o processo poderá enviar a mensagem de *echo* para o seu vizinho que lhe enviou o *probe*. No caso de grafos com ciclos, pode ocorrer a situação de dois nós já visitados trocarem *probes* entre si. Neste caso, os dois processos podem enviar suas mensagens *echo* correspondentes a este segundo *probe* imediatamente. Um nó que receba um *probe* de sua única conexão à rede, isto é, um nó *folha*, também pode responder imediatamente. Os *probes* e *echos* trocados por processos já visitados são chamados de mensagens *idle* neste texto.

Para implementar este algoritmo, foi feito um conjunto de funções que vão estar disponíveis em cada instância do sistema Tche como um serviço de suporte ao uso deste algoritmo. Sendo assim, toda aplicação que desejar usar o algoritmo para alguma tarefa poderá usar estas funções

de suporte. Para que esse mecanismo de suporte possa ser usado em aplicações diferentes, são passadas nas mensagens de *probe* e *echo* strings com código Lua que serão executadas quando sua mensagem chegar a um determinado nó.

A implementação do algoritmo usa as seguintes variáveis globais:

```
probemsg = nil      --mensagem de probe recebida
idlemsg = nil       --mensagem de probe idle recebida
target = nil        --nome do processo ao qual deve-se enviar o echo
waitechos = 0       --indica quantos echos devem ser esperados
echosreceived = 0   --contador dos echos já recebidos
idleechomsg = ""    --valor default para a mensagem de echo idle
echomsg = ""        --valor default para a mensagem de echo
```

Todos os processos envolvidos na computação de alguma tarefa com esse algoritmo, terão essas variáveis definidas em seu ambiente global. Apesar de não ser necessária a inicialização de variáveis com o valor `nil`, este recurso foi usado como um mecanismo de reinicialização do algoritmo. As variáveis `echomsg` e `idleechomsg` representam os códigos que vão ser enviados com as mensagens de *echo* e de *idle-echo*, respectivamente. O uso dessas variáveis vai ficar mais claro com a apresentação das funções que dão suporte ao algoritmo.

Antes da apresentação da implementação continuar, é importante chamar a atenção para alguns detalhes de Lua. A linguagem Lua tem como delimitadores de strings os caracteres `"` e `'`, além do par `[[` e `]]`. Este par permite a divisão de uma string em várias linhas e o aninhamento de delimitadores. Outro detalhe da linguagem é que funções podem retornar mais de um valor. Um exemplo disto é a função predefinida `next`, que é usada para a iteração sobre uma tabela. Esta função recebe dois parâmetros. O primeiro é uma tabela e o segundo é um índice desta. Este índice pode ser um número ou uma string. Os seus valores de retorno são o próximo índice da tabela e o valor associado a este. Quando chamada com um valor `nil` como segundo argumento, a função retorna o primeiro índice da tabela e seu valor correspondente. Quando chamada com o último índice da tabela, ou com uma tabela vazia (igual a `nil`), esta função retorna `nil`.

Na implementação da função `probe` apresentada a seguir, os três parâmetros recebidos são armazenados em variáveis globais. O parâmetro `sender` é armazenado na variável global `target` para que a função `sendecho` possa enviar o *echo* para o processo correto. Já os outros dois parâmetros são armazenados em variáveis globais para que o código presente no parâmetro `msg`, que vai ser executado sobre o escopo global de Lua, possa alterar o valor das mensagens enviadas nos *probes* para os nós seguintes.

```
--a tabela info contem informações a respeito do processo, por
--exemplo: o nome que identifica o processo e uma tabela com os nós
--adjacentes da rede
--ex.: info = {name="p4", adjacents = {"p1","p2","p5"}}

--o operador de concatenacao de strings e' o par de caracteres ..

--funcoes para serem chamadas pela aplicacao
function probe(sender, msg, imsg)
  local i, host = next(info.adjacents, nil)
  local wait
  probemsg = msg
  idlemsg = imsg
  target = sender
  dostring(msg)
  while i do
    if host ~= sender then
      send(host,"probe('"..info.name.."',[["..probemsg.."]],[["
                      ..idlemsg.."]])")
      waitechos = waitechos + 1
    end
    i, host = next(info.adjacents, I)
  end
  probe = idleprobe       --depois do primeiro, os possíveis demais
                          --probes são tratados pela função idleprobe
  if waitechos == 0 then  sendecho()  end
end

function echo(sender, msg)
  echosreceived = echosreceived + 1
  dostring(msg)
  if echosreceived == waitechos then  sendecho()  end
end

--funcoes para uso interno
function idleprobe(sender, msg, imsg)
  dostring(imsg)
  send(sender, "echo('"..info.name.."',[["..idleechomsg.."]])" )
end

function sendecho()
  send( target, "echo('"..info.name.."',[["..echomsg.."]])" )
end
```

Um exemplo da aplicação desse algoritmo seria determinar a topologia de uma rede. Para isso ela poderia usar o seguinte código, que usa a variável `network` para armazenar a tabela com a topologia da rede:

```
pmsg = [[ network = {}                          --constroi uma tabela
          network[info.name] = info.adjacents --inicializa a tabela
          function addinfo(str)
             dostring("adjs="..str)
             updatetable(network,adjs)
          end
          doecho = [[
            echomsg="addinfo([["..table2string(network)..
                     "]]);dostring(doecho)"
                    ]]
          dostring(doecho)
]]

imsg = [[idleechomsg = ""]] --no caso de um probe idle, é retornado
                            --um echo com uma string vazia

--esta função é parecida com a função probe, só que é um pouco mais
--simples, já que a raiz da árvore não precisa retornar um echo
function findtop()
   local i, host = next(info.adjacents, nil)
   dostring(pmsg)
   while i do
      send(host,"probe('"..info.name.."',[["..pmsg.."]],[["
                ..imsg.."]])")
      waitechos =waitechos + 1
      i, host = next(info.adjacents,i)
   end
   sendecho = fakesendecho     --quando todos os echos para o
                               --processo raiz já tiverem sido
end                            --recebidos, a função fakesendecho
                               --vai ser chamada


function fakesendecho()
   --esta funcao vai ser chamada quando o processo já tiver recebido
   --todos os echos de seus processos adjacentes
   ...
end
```

Note que o código da string `pmsg` define a regra para determinar o valor da variável `echomsg` e o código da string `imsg` define o valor da variável `idleechomsg`. O mecanismo de suporte ao algoritmo usa estas variáveis, e toda a aplicação que quiser usá-lo terá que definir em suas mensagens os valores dessas variáveis. Um detalhe interessante é como o valor de `echomsg` é determinado. É usada a string auxiliar `doecho` para definir a regra para o cálculo de `echomsg`, e toda vez que a mensagem `echomsg` for executada o seu valor é recalculado através da nova execução da string `doecho`. A função `addinfo` usa a função utilitária `updatetable` que recebe duas tabelas como parâmetros e atualiza a primeira com os campos da segunda que ela não possui.

Considerações Finais

Com a implementação do mecanismo apresentado, conseguiu-se criar um suporte para comunicação entre processos que junta a flexibilidade e simplicidade da linguagem de extensão Lua com o assincronismo do modelo de programação dirigida por eventos. A união destas características é muito interessante para aplicações interativas, onde a necessidade de interoperabilidade a cada dia se torna mais importante e o assincronismo é fundamental.

Por outro lado, com os algoritmos que foram implementados, constatou-se a adequação do mecanismo à implementação de algoritmos clássicos de programação distribuída. Até mesmo para algoritmos onde o uso do mecanismo não parecia ser conveniente devido a sua característica de assincronismo, obteve-se resultados de implementação interessantes. O ganho de flexibilidade nas implementações, devido à possibilidade de transferência de código entre processos, compensou a necessidade de algum controle extra para "sincronizar" o algoritmo.

Além da flexibilidade apresentada, o mecanismo também atende ao requisito de simplicidade para que possa ser usado pelos usuários finais de uma aplicação. Desta forma, ele se enquadra perfeitamente à filosofia de Lua.

O próximo caminho a ser seguido é aprofundar o uso do mecanismo em aplicações interativas, onde espera-se que ele seja de grande valia. Junto a isto, será estudado um mecanismo para implementação de objetos distribuídos que tire proveito das facilidades de Lua para programação orientada a objetos.

Referências Bibliográficas

[And91] G. R. Andrews. Paradigms for Process Interaction in Distributed Programs. Em *ACM Computing Surveys*, 23(1):49-90, 1991.

[BST89] H. E. Bal, J. G. Steiner, e A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. Em *ACM Computing Surveys*, 21(3):261-322, 1989.

[Car95] A. F. L. Carregal. *Tche: Um Ambiente Visual Lua*. Dissertação de Mestrado, Departamento de Informática, PUC-Rio, março 1995.

[CRI95] R. Cerqueira, N. Rodriguez, e R. Ierusalimschy. *An Experiment on Event-driven Distributed Programming*. Monografias em Ciencia da Computacao ?/95, PUC-Rio, Rio de Janeiro, Brasil, 1995.(a ser publicado)

[CS93] D. E. Comer e D. L. Stevens. *Internetworking With TCP/IP Volume III: Client-server Programming and Applications BSD Socket Version*. Prentice Hall, 1993.

[FIF94] L. H. Figueiredo, R. Ierusalimschy, e W. Celes Filho. The design and implementation of a language for extending applications. Em *XXI Semish*, páginas 273-284, 1994.

[IFF95] R. Ierusalimschy, L. H. Figueiredo, e W. Celes Filho. *Reference Manual of the Programming Language Lua version 2.1*. Monografias em Ciencia da Computacao ?/95, PUC-Rio, Rio de Janeiro, Brasil, 1995.(a ser publicado) (disponível por ftp em `ftp.inf.puc-rio.br/pub/docs/techreports`).

[Ous90] J. Ousterhout. Tcl: an Embeddable Command Language. Em *Proceedings of the Winter 1990 USENIX Conference*, USENIX Association, 1990.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Ste90] W. R. Stevens. *Unix Network Programming*. Prentice Hall, 1990.

[Sud90] V. Suderman. A Framework for Parallel Distributed Computing. Em *Concurrency: Practice and Experience*, dez 1990.

[Sun88] Sun Microsystems 1988. *RPC: Remote Procedure Call, Protocol Specification, Version 2*, RFC 1057, jun 1988.