Funções Remotas como Valores de Primeira Classe em um Sistema de Objetos Distribuídos

Maria Julia de Lima
mjulia@inf.puc-rio.br
Noemi Rodriguez
noemi@inf.puc-rio.br
Roberto Ierusalimschy
roberto@inf.puc-rio.br

PUC-RioInf.MCC10/00
Fevereiro 2000

Abstract: This paper describes a system for distributed programming based on the interpreted language Lua. Like functional languages, Lua supports functions as first-class values. The system proposed in this work extends this feature to remote functions. Processes communicate using remote procedure calls which accept functions as arguments or returned values. The solution combines expressive power with transparency in the manipulation of distributed resources. The paper also shows how distributed objects can be implemented using remote functions references and properties already provided by Lua, emphasizing the flexibility of the system.

Keywords: Distributed Programming, Distributed Objects, Event-driven Programming, Remote Procedure Call.

Resumo: O trabalho descreve um sistema de programação distribuída baseado na linguagem de programação Lua. Em Lua, funções são valores de primeira classe. O sistema apresentado estende esse tratamento a funções remotas, oferendo comunicação via chamada remota de procedimentos onde funções podem ser passadas como parâmetros e retornadas em chamadas entre processos. O resultado é um ambiente que une expressividade com transparência na manipulação de recursos distribuídos. O trabalho mostra também como a manipulação de referências a funções remotas pode ser usada para implementar objetos distribuídos de uma forma bastante direta, sem necessidade de suporte explícito da linguagem.

Palavras-chave: Programação Distribuída, Objetos Distribuídos, Programação Orientada a Eventos, Chamada Remota de Procedimentos.

1 Introdução

Várias soluções têm sido propostas para atender às necessidades de desenvolvimento de aplicações distribuídas. Nessa direção, encontramos novas linguagens de programação ou extensões de linguagens já existentes, que incorporam primitivas para tratar aspectos de comunicação e sincronismo. Entretanto, um problema que ocorre em várias das soluções propostas é que a semântica dos mecanismos para programação distribuída não adere bem às abstrações já existentes nas linguagens.

O objetivo principal deste trabalho é investigar como podemos empregar algumas abstrações oferecidas pela linguagem Lua [IFC96] para a definição de um sistema de programação distribuída. A base desse sistema é a utilização de referências a funções remotas como valores de primeira classe. O resultado, diferente da maioria dos sistemas existentes, une a expressividade de linguagens que tratam funções como valores de primeira classe com a transparência na manipulação de recursos distribuídos entre vários processos da aplicação. Além disso, a capacidade de comunicarmos funções em aplicações distribuídas favorece a extensibilidade e a criação dinâmica de novos protocolos de interação.

Como valores de primeira classe, referências a funções remotas podem ser guardadas em variáveis ou passadas como parâmetro e retorno de funções. Além disso, referências a funções remotas podem ser invocadas da mesma forma que qualquer função local. Essa última propriedade reproduz um mecanismo baseado em chamada remota de procedimentos (RPC) para fazer a comunicação entre processos distintos.

Neste trabalho também propomos utilizar funções remotas na implementação de objetos distribuídos. Nessa implementação, objetos são representados remotamente através de sua interface, que é um conjunto de referências a funções remotas. Cada uma dessas funções, por sua vez, é um fecho (*closure*) que encapsula o objeto real (o *self*) sobre o qual a função opera. Como veremos mais adiante, tal representação permite a manipulação de objetos remotos com a mesma sintaxe usada para objetos locais.

Para desenvolver o sistema proposto neste trabalho, usamos o sistema Agentes Lua. O sistema Agentes Lua (ALua) [UR99] é uma extensão da linguagem Lua que oferece mecanismos de comunicação baseados no modelo de programação dirigido por eventos. Nesse modelo de comunicação, o envio de mensagens é feito de forma assíncrona, e o recebimento de mensagens gera um evento para o qual existe uma função registrada encarregada de fazer o tratamento adequado.

O restante deste trabalho está organizado da seguinte forma. Na seção 2, mostramos a solução proposta para comunicar funções remotas como valores de primeira classe. Na seção 3, discutimos como o uso de fechos pode ser utilizado na implementação de objetos distribuídos. Na seção 4 discutimos a concorrência que o nosso sistema suporta na execução de processos. Finalmente, na seção 5 apresentamos algumas conclusões observadas.

2 Funções remotas como valores de primeira classe

Funções em Lua são valores de primeira classe. Portanto, pode-se fazer com funções tudo o que a linguagem permite fazer com qualquer outro valor. Funções podem ser atribuídas a variáveis, passadas como parâmetros ou devolvidas como resultado de outras funções. O código em Lua mostrado abaixo exemplifica como um valor do tipo função pode ser criado, atribuído a uma variável global e passado como parâmetro para uma outra função.

```
f = function(n) ... end
g(f)
```

No sistema aqui proposto, o mecanismo básico de comunicação é a chamada a funções remotas. Um processo `A` pode possuir em uma variável `foo` uma referência para uma função localizada em um outro processo `B`. Quando `A` chama a função `foo`, o sistema gera um evento para o processo `B` executá-la. A princípio, os valores que poderiam ser passados como parâmetros ou retornados em tais chamadas são números, strings, e tabelas (arrays associativos). Tabelas são transmitidas por cópia profunda, de modo que tabelas contidas em tabelas são também copiadas. Um ponto fundamental de nosso sistema, no entanto, é que ele estende às funções remotas o status de valor de primeira classe que é dado em Lua a funções comuns. Com isso, obtemos um ambiente de programação distribuída onde funções também podem ser passadas como parâmetros ou retornadas pelas chamadas remotas. Sempre que uma função é transmitida para um processo diferente daquele onde foi definida, é criado um *proxy* para representar essa função remota. A criação de proxies é transparente para o desenvolvedor: quando uma função é usada como parâmetro ou retorno de uma outra função remota, o sistema automaticamente cria um proxy.

A inclusão de funções como valores de primeira classe em um sistema de programação distribuída oferece um mecanismo através do qual podemos simular diversos outros mecanismos de comunicação em aplicações distribuídas; chamada remota de procedimentos [BN84], avaliação remota [SG90] e extensão dinâmica de componentes [CRI99] são exemplos de mecanismos que podem ser simulados.

O exemplo a seguir ilustra o uso de funções remotas como valores de primeira classe. Vamos assumir que um processo `A` possui uma referência remota para a função `setglobal` em um processo `B`. A função `setglobal` recebe um nome (como uma string) e um valor, e atribui o valor à variável global com o nome dado. Digamos que essa referência remota em `A` esteja armazenada na variável `setBglobal`. Então, após `A` executar o código abaixo, o processo `B` passa a contar com uma função `setAglobal`, que atribui valores a globais no processo `A`.

```
  setBglobal("setAglobal", setglobal)
```

Observe que a função `setglobal`, usada como argumento de `setBglobal`, é uma função local do processo `A` (em particular, é uma função pré-definida da linguagem Lua).

Ao passar valores do tipo *function* para uma função remota ou retornar resultados desse mesmo tipo, o processo passa a dispor do próprio mecanismo para exportar dinamicamente novas referências de funções remotas. Ainda usando o exemplo anterior, suponha que o processo `B` execute o código abaixo:

```
function le_relogio ()
  readfrom("/proc/uptime")       -- abre pseudo-arquivo para leitura
  local s = read()               -- le primeira linha
  readfrom()                     -- fecha arquivo
  return s
end

function foo1 ()
  ...
end

function foo2 ()
  ...
end

setAglobal("le_relogioB", le_relogio)
setAglobal("foo1", foo1)
setAglobal("foo2", foo2)
```

Agora, o processo `A` passa a contar com acesso direto às funções `le_relogio`, `foo1` e `foo2` de `B`. Por exemplo, quando `A` executa `le_relogioB`, recebe como resultado da chamada o valor do relógio interno da máquina onde `B` está executando.

Uma outra característica do sistema é que, assim como qualquer outro valor, a referência a uma função remota pode ser re-transmitida de um processo para o outro. Ao exportar referências de funções, usando passagem de parâmetro ou retorno de funções, é indiferente para o desenvolvedor se a referência está sendo criada naquele momento ou se uma referência está sendo re-transmitida. O próprio sistema cuida de tratar essas situações, incluindo casos onde o processo exporta uma mesma função mais de uma vez, ou onde um processo recebe de um outro uma referência a uma função remota definida originalmente nele próprio.

Implementação

As referências a funções exportadas de um processo para outro foram representadas por objetos em Lua. Esses objetos funcionam como *proxies* das funções representadas. Ao exportar uma função de um processo para o outro, o sistema cria um objeto que possui, entre outras informações, um identificador da função original e um identificador do processo onde a função está definida. Com essas informações, invisíveis para o programador, o sistema reconhece as referências a funções remotas, dando um tratamento diferente das funções locais. A execução das funções remotas é o principal exemplo de como o sistema faz uso dessas informações para prover um comportamento semelhante à execução de uma função local.

Para implementar a chamada a uma função remota, foi utilizado o mecanismo de *fallback* (ou *tag methods*) presente na linguagem Lua. *Fallback* é um mecanismo de Lua que permite especificar funções para serem executadas em situações onde o interpretador não sabe como prosseguir, como por exemplo, em condições anormais que ocorrem em tempo de execução [IFC96]. Semanticamente, esse mecanismo pode ser classificado como um recurso para tratamento de exceção com retomada [YB85]; pragmaticamente, ele não é usado apenas em situações de "exceção", mas como um mecanismo geral de extensão da linguagem.

Quando um processo `A` chama uma função remota `foo`, o que ocorre é que `foo` na verdade não é uma função, mas um objeto (o proxy da função). Essa é uma das condições "anormais" que geram uma fallback em Lua: tentar chamar um valor que não é uma função. Quando esse evento ocorre, Lua chama um tratador especificado pelo programa, ao invés de dar um erro (o erro só é gerado na ausência de um tratador para o evento). No nosso caso, o tratador registrado para esses casos é responsável por empacotar a identificação da função chamada e seus parâmetros, e enviá-los para o processo onde a função está localizada. Após a execução da função original, seus resultados são retornados pelo tratador, como os resultados da chamada original feita pelo programa.

É interessante observarmos que nenhuma alteração foi feita na linguagem Lua. Todo o sitema opera através da interface padrão que a linguagem oferece.

3 Objetos Distribuídos

Cada vez mais é reconhecida a importância do paradigma de objetos na construção de aplicações distribuídas. A definição de objeto baseada nos princípios de modularidade e encapsulamento provê um modelo poderoso e flexível que atende à heterogeneidade de diversas arquiteturas cliente/servidor. O mecanismo de comunicação adotado na maior parte dos modelos de objetos distribuídos é baseado em chamada remota de procedimentos. A interface das operações disponibilizadas por um objeto é publicada para ser utilizada no estilo RPC, tornando transparente para as aplicações a localização dos objetos envolvidos na computação distribuída. A notoriedade de modelos de distribuição baseados em objetos, como CORBA [Obj95], comprovam a importância desses conceitos no contexto de programação distribuída.

Neste trabalho, propomos uma distribuição de objetos que segue a mesma semântica de *Obliq* [Car94] e de *Network Objects* [BNOW94]. Um objeto no nosso sistema nunca é copiado para um outro processo, não existindo, portanto, duplicação de seu estado. No entanto, referências a funções que acessam esse objeto podem ser livremente transmitidas de um processo para o outro. Os valores que representam as referências a funções remotas são manipulados como valores de primeira classe e fazem parte da interface que o objeto disponibiliza para os seus clientes.

A maioria dos sistemas que possuem facilidades para representação de objetos distribuídos oferece suporte explícito para manipular remotamente esses objetos. No entanto, para representar e manipular objetos distribuídos no nosso sistema não foi necessário definir primitivas especiais. Utilizamos os próprios recursos de que a linguagem Lua já dispõe, tornando a solução proposta mais aderente à linguagem de programação utilizada.

Uma função em Lua pode ser definida dentro do corpo de uma outra função (*enclosing function*). Como Lua mantém suas variáveis locais em uma pilha, uma função aninhada não pode referenciar diretamente uma variável definida na função que a engloba, já que essa variável pode não existir mais quando a função aninhada for executada. Para resolver esse problema, Lua introduziu o conceito de *upvalues*. Um upvalue representa o valor de uma variável não local no momento em que a função é instanciada. Uma instância da função com seus upvalues fixados é chamada de *fecho*. Assim, apesar da função não poder acessar variáveis não locais, ela pode acessar os valores dessas variáveis. Para evitar confusão entre variáveis e valores, upvalues em Lua são prefixados com o sinal `%`; por exemplo:

```
function cria_inc (x)
  local inc = function (y) return %x+y end
  return inc
end

inc5 = cria_inc(5)
inc2 = cria_inc(2)
print(inc2(10))       --> 12
print(inc5(10))       --> 15
```

Note como a função `inc` não referencia `x` diretamente, mas usa um upvalue de `x`. Cada vez que a função `cria_inc` é executada, a função `inc` é instanciada, congelando em `%x` o valor corrente de `x`.

O exemplo abaixo, mais complexo, mostra como podemos usar upvalues para definir uma função de fatorial sem usar loops nem recursão, através de um operador de ponto fixo `Y`, também definido em Lua:

```
Y = function (le)
      local a = function (f)
                  return %le(function (x) return %f(%f)(x) end)
                end
      return a(a)
    end

F = function (f)
      return function (n)
```

```
                if n == 0 then return 1
                else return n*%f(n-1) end
              end
    end

fat = Y(F)         -- fat e' o ponto fixo de F
print(fat(10))     --> 3628800
```

Na solução que adotamos para representar objetos, o *proxy* remoto de um objeto é uma tabela Lua, isso é, um array associativo que associa cada nome de método a uma função. Quando esse proxy é exportado para outro processo, a tabela é copiada no outro processo; como a cópia é profunda, cada elemento da tabela também é exportado. Em particular, as funções originais são substituidas por referências remotas.

Para criarmos esses proxies, definimos para cada classe uma função que fabrica objetos daquela classe, e retorna um proxy para o objeto criado. A função que representa a fábrica, ao ser executada, cria uma instância do objeto e das funções que definem sua interface. Cada uma dessas funções se refere ao objeto original via upvalue, de modo que não precisamos passar o objeto original (nem ter acesso a ele) quando invocamos a função. O exemplo a seguir ilustra uma função fábrica de um objeto bastante simples, com um único atributo e duas funções para consultar e modificar o valor do atributo:

```
-- Defines the object class
function NewObject ()
  local self = {attribute = nil}
  local set_attribute = function (x)
                          %self.attribute = x
                        end
  local get_attribute = function ()
                          return %self.attribute
                        end
  local proxy = { get_attribute = get_attribute,
                  set_attribute = set_attribute
                }
  return proxy
end

-- Creating objects
obj1 = NewObject()
obj2 = NewObject()

-- Using object methods
obj1.set_attribute("obj1")
x = obj2.get_attribute()
```

A função `NewObject`, ao ser chamada, cria um objeto local `self`, define as operações sobre esse objeto, e retorna uma tabela (`proxy`) cujos campos `get_attribute` e `set_attribute` contêm referências às funções locais definidas com os mesmos nomes.

A tabela retornada por `NewObject` representa a interface dos objetos criados. Quando essa interface é transmitida para um outro processo, o objeto correspondente pode ser acessado remotamente. As funções `set_attribute` e `get_attribute` utilizam *upvalues* para ter acesso a instância de `self`, fixada quando `Object` é executada.

Note que, nessa solução, apenas tabelas e referências a funções remotas são transmitidas de um processo para o outro; não foi necessário estender o modelo proposto com nenhum mecanismo adicional aos já existentes na linguagem.

4 Concorrência

A comunicação por chamadas a funções remotas no nosso sistema é uma abstração construída sobre o sistema ALua, que utiliza um protocolo de comunicação dirigido por eventos. Em ALua, tanto o envio como o recebimento das mensagens ocorrem de maneira assíncrona. A chegada de uma mensagem é tratada como a chegada de um evento de comunicação. Dessa forma, não existe uma operação explícita para receber uma mensagem. Um *handler* é responsável por tratar os eventos de comunicação e sua ação default pode ser, por exemplo, a execução das mensagens recebidas. Assim como faz para outros tipos de eventos, o desenvolvedor pode mudar a ação default, definindo e registrando um novo handler para tratar os eventos de comunicação.

No nosso sistema, os eventos são sempre requisições para execução de funções locais. Quando um processo chama uma função remota, o sistema se encarrega de gerar um evento com o pedido de execução da função chamada. Para poder continuar seu fluxo normal de execução, o processo que originou a chamada deve aguardar que esse evento seja tratado e produza uma resposta. Enquanto isso, novos eventos podem estar chegando. Como tratar esses eventos?

Uma decisão possível seria fazer com que cada chamada remota fosse executada de maneira atômica, ou seja, seu processamento fosse sempre finalizado antes de se iniciar o tratamento de um novo evento. Essa abordagem corresponde a proposta de monitores segundo sua definição original [Hoa74], onde cada processo corresponderia a um monitor. A exclusão mútua entre as funções dos monitores garante que regiões críticas sejam executadas atomicamente, evitando interferências perigosas e preservando os dados compartilhados. No entanto, esse estilo de sincronismo pode criar situações de *deadlock*, se um processo `A` chama `B` e `B` precisa chamar `A` novamente. Em particular, em sistemas orientados a objetos esse tipo de deadlock se torna bastante relevante, pelo uso frequente de callbacks, *listeners*, e outras construções similares.

Na solução que adotamos, os eventos não são tratados de maneira atômica. Enquanto está bloqueado aguardando o retorno de uma chamada, o processo pode atender a novos eventos. Dessa forma, a execução de funções locais é iniciada e, posteriormente, o processo retorna para aguardar a resposta pendente da chamada anterior.

A figura 1 exemplifica a situação descrita acima, onde um mesmo processo atende a mais de um evento simultaneamente. Tudo começa quando o processo `A` faz uma chamada a uma função remota `f`. Como consequência, o sistema bloqueia a execução de `A` até o processo `B` executar e enviar o retorno da função `f`. Enquanto isso, um outro processo `C` faz uma chamada para uma função `g` definida em `A`. Mesmo estando bloqueado pela chamada a uma função remota, o processo `A` pode tratar esse novo evento, executando a função `g` e retornando, em seguida, para o estado anterior onde aguarda o retorno de `f`.

É interessante notar que a execução de uma função em resposta a um novo evento recebido pode fazer com que o processo chame novamente uma função remota, entrando em um outro ponto de bloqueio diferente do anterior. Logo, o sistema permite que um processo tenha uma lista de espera pelo retorno de mais de uma chamada a funções remotas. No exemplo anterior, a execução de `g` poderia fazer uma chamada a uma função remota `e`. Como consequência, o processo `A` teria dois pontos de bloqueio na fila aguardando resposta: um pela chamada à função `f` e outro pela chamada à função `e`.

A lista de espera pelo retorno de chamadas a funções remotas, criada como consequência do tratamento concorrente de mais de um evento, adota uma política de tratamento das respostas recebidas baseada na pilha de execução corrente. O sistema somente trata a resposta da última chamada remota, que está no topo da pilha. Se a mensagem for a resposta de uma outra função previamente chamada, o sistema guarda a resposta para ser tratada quando a pilha de execução voltar ao ponto de bloqueio da chamada correspondente.

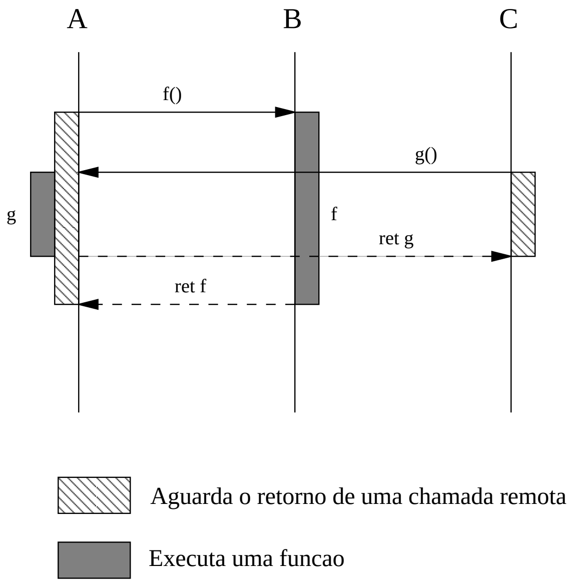

Figura 1: Execução reentrante em um mesmo processo.

A figura 2 ilustra um caso onde duas respostas de chamadas remotas são aguardadas em pontos de execução diferentes. Somente após ter recebido o retorno de `f` o processo pode tratar o retorno de `e`. No caso do sistema prosseguir sua execução sem erro em nenhum dos processos, ambas as respostas serão recebidas e tratadas, independente de sua ordem de chegada.

O tratamento de mais de um evento ao mesmo tempo permite que um processo esteja simultaneamente atendendo a várias sequências de execução que compartilham memória e recursos alocados para esse mesmo processo. Esse tipo de concorrência exige um controle no acesso de globais para garantir que o estado do processo seja preservado na execução de cada função. Para evitar a necessidade desse tipo de controle de acesso, analisamos um outro possível comportamento para o sistema. Nessa alternativa, ao chegar uma solicitação para execução de uma função local, o agente somente executa essa função se ela for parte da linha de controle (*thread of control*) corrente. A Figura 3 ilustra esse cenário.

Suponha que a definição da função `f` inclua uma chamada à função `g`. Portanto, durante a execução de `f` e após a chamada a `g`, a linha de controle corrente inclui as chamadas a `f` e `g`, apesar destas funções estarem sendo executadas em processos diferentes. Mesmo bloqueado pela espera do retorno da chamada a `f`, o processo `A` pode executar a função `g`, já que a sua invocação faz parte da linha de controle corrente. Se chegar uma segunda chamada a `g` proveniente de um outro processo, ela será processada somente depois que `A` receber a resposta de `f` e terminar a linha de controle corrente.

A linguagem Java [AG96] utiliza essa mesma solução na construção de monitores com o uso de blocos de código sincronizados. Quando um método é definido como *synchronized*, sua execução detém o *lock* associado ao objeto, o que impede que qualquer outro método do mesmo objeto seja executado, a não ser que a chamada seja feita pela mesma *thread* de execução corrente. Essa solução, entretanto, também pode criar situações de *deadlock* como exemplificado na Figura 4.

Os processos `A` e `C` fazem chamadas às funções `e` e `f`, respectivamente. Se `C` receber a chamada de `e` após já ter chamado `f`, a execução de `e` ficará pendente, já que `C` está aguardando

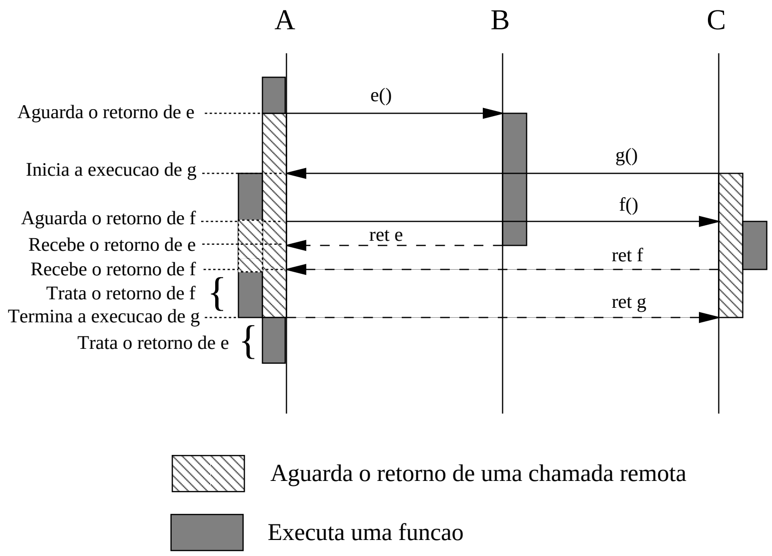

Figura 2: Um mesmo processo aguardando o retorno de duas chamadas remotas.

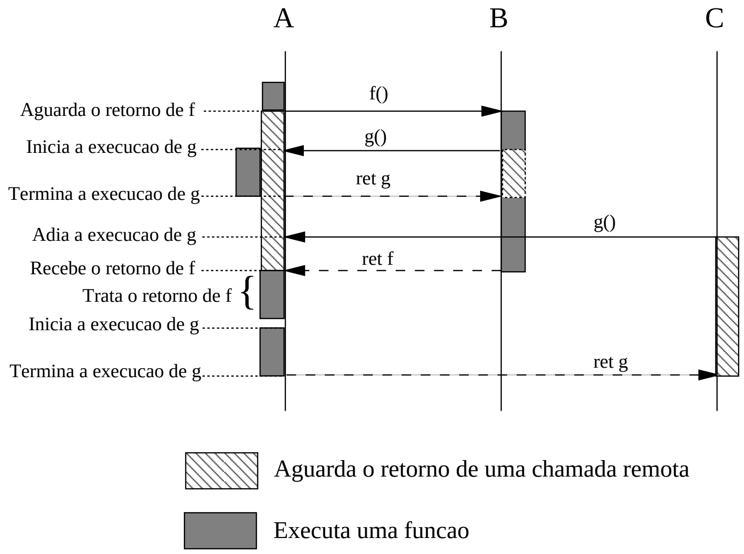

Figura 3: Controle da linha de execução corrente.

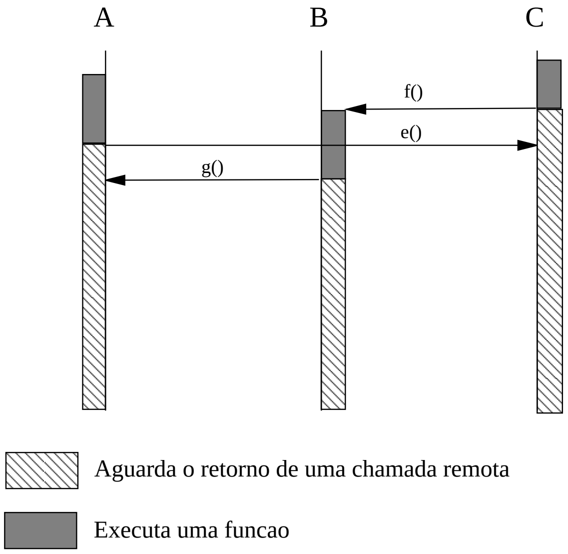

Figura 4: Exemplo de *deadlock*.

o retorno de `f` e a chamada de `e` não faz parte da linha de controle corrente. Por sua vez, a execução de `f` precisa do retorno da chamada de `g` para poder finalizar. Se o evento com a chamada de `g` chegar depois de `A` ter feito a chamada a `e`, temos novamente uma situação em que o controle da linha de execução corrente impede a execução de `g` até que chegue o retorno de `e`. Note que `C` somente pode executar `e` se `A` executar `g` e, da mesma forma, `A` somente pode executar `g` se `C` executar `e`, causando um deadlock na aplicação.

Em última análise, as duas alternativas investigadas caracterizam graus diferentes de controle de concorrência, com diferentes compromissos entre *liveness* e *safeness*. A primeira solução, que não oferece controle de execução de funções concorrentes enquanto o processo aguarda pelo retorno de uma chamada, não cria situações de deadlock, embora dificulte o controle de exclusão mútua. Já a segunda solução, baseada no controle da linha de execução corrente, embora garanta a exclusão mútua com a sincronização de *threads* diferentes, possibilita mais situações com a ocorrência de deadlock. Nesse trabalho, por razões de flexibilidade, preferimos adotar a primeira solução. Atualmente estamos investigando outros mecanismos que o desenvolvedor possa explicitamente empregar para garantir a sincronização necessária.

5 Conclusões

O trabalho aqui apresentado segue a mesma linha de [HA92] e [HC95], que investigam o uso de abstrações de alto nível para programação distribuída. HAL [HA92] e Kali [HC95] são linguagens de programação funcionais, com uma sintaxe no estilo LISP, que também adotam um modelo de objetos distribuídos. Usando continuações e um mecanismo básico de comunicação assíncrona, HAL permite o envio de mensagens síncronas no estilo RPC. Em Kali, funções e continuações são valores de primeira classe e, assim como qualquer outro objeto na linguagem, podem ser enviados nas mensagens entre processos. Essa facilidade é o fundamento utilizado pelas aplicações em Kali para desenvolver recursos de concorrência e comunicação.

Nesse trabalho, apresentamos um sistema de programação distribuída que se baseia em referências a funções remotas como valores de primeira classe. O sistema adota a chamada remota de procedimentos como mecanismo de comunicação entre os processos e provê suporte a objetos distribuídos cujas interfaces são disponibilizada como referências a funções remotas. O trabalho desenvolvido mostra como algumas abstrações oferecidas pela linguagem Lua, em especial fechos e funções como valores de primeira classe, facilitam a construção de um sistema com essas características. A utilização dessas abstrações e de um ambiente de comunicação dirigido por eventos também permitiu a implementação e estudo de diferentes alternativas para controle de concorrência no sistema. Estamos atualmente trabalhando no estudo de mecanismos de sincronização que evitem inconsistências decorrentes de acessos concorrentes a variáveis globais.

Como resultado desse trabalho, conseguimos oferecer um ambiente de programação distribuída bastante natural e aderente à semântica já existente em uma linguagem de programação. Ao contrário do mecanismo tradicional de RPC, o programador pode exportar interfaces de funções usando os próprios recursos da linguagem para passagem de parâmetros e retorno de funções. Da mesma forma, objetos distribuídos são representados através de fechos e manipulados remotamente usando referências as funções exportadas como parte de sua interface. A solução obtida torna o mecanismo flexível e transparente para o programador. O sistema descrito foi implementado em ALua, permitindo que as novas facilidades sejam utilizadas dentro de um ambiente orientado por eventos. O uso de uma linguagem interpretada como Lua com um modelo de programação distribuída dirigido por eventos tem sido investigado em diversos trabalhos [RUIC96, CRI95] e os resultados são bastante favoráveis em função da expressividade e flexibilidade alcançadas.

Referências Bibliográficas

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[BN84] A. Birrel and B. Nelson. Implementing remote procedure calls. *ACM Trans. on Computer Systems*, 2(1):39–59, feb 1984.

[BNOW94] A. Birrel, G. Nelson, S. Owicki, and E. Wobbe. Network objects. Technical Report 115, Digital Equipment Corporation Systems Research Center, 1994.

[Car94] L. Cardelli. Obliq: a language with distributed scope. Technical Report 121, Digital Equipment Corporation Systems Research Center, jun 1994.

[CRI95] R. Cerqueira, N. Rodriguez, and R. Ierusalimschy. Uma experiência em programação distribuída dirigida por eventos. In *Anais do PANEL95*. SBC, 1995.

[CRI99] M. Catunda, N. Rodriguez, and R. Ierusalimschy. Dynamic extension of CORBA servers. In *Euro-Par'99 Parallel Processing*, pages 1369–1376, Toulouse, France, aug 1999. Springer-Verlag. LNCS 1685.

[HA92] C. Houck and G. Agha. Hal: A high-level actor language and its distributed implementation. In *Proceedings of th 21st International Conference on Parallel Processing (ICPP '92)*, volume II, pages 158–165, St. Charles, IL, 1992.

[HC95] S. Jagannathan e R. Kelsey H. Cetjin. Higher-order distributed objects. *ACM Trans. on Programming Languages and Systems*, 17(5):704–739, sep 1995.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communication of the ACM*, 17(10):549–557, October 1974.

[IFC96] R. Ierusalimschy, L. H. Figueiredo, and W. Celes. Lua–an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[Obj95] Object Management Group, Inc. *The Common Object Request Broker Architecture and Specification*, revision 2.0 edition, jul 1995.

[RUIC96] N. Rodriguez, C. Ururahy, R. Ierusalimschy, and R. Cerqueira. The use of interpreted languages for implementing parallel algorithms on distributed systems. In *Euro-Par'96 Parallel Processing*, pages 597–600, Tyon, France, aug 1996. Springer-Verlag. LNCS 1123.

[SG90] J. W. Stamos and D. K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[UR99] C. Ururahy and N. Rodriguez. ALua: An event-driven communication mechanism for parallel and distributed programming. In *PDCS'99 (12th Intl Conf. on Parallel and Distributed Computing Systems*, pages 108–113, Fort Lauderdale, Florida, 1999.

[YB85] S. Yemini and D. Berry. A modular verifiable exception handling mechanism. *ACM Trans. on Programming Languages and Systems*, 7(2):214–243, April 1985.