3
Integrando invocações remotas com gerência cooperativa de tarefas

O modelo de programação discutido no capítulo anterior — uma combinação de co-rotinas e comunicação baseada em eventos, com elementos que encapsulam a transferência de controle entre as linhas de execução de uma aplicação — possibilita a construção de sistemas assíncronos com a interface síncrona da programação tradicional. Neste capítulo descrevemos uma implementação desse modelo para ser usada no desenvolvimento de aplicações que requerem a interação entre processos localizados em máquinas distintas (Rodriguez/05). Em particular, discutimos como essa implementação pode ser aplicada para o desenvolvimento de aplicações paralelas que usam a estrutura das redes geográficas.

Tradicionalmente, a pesquisa nos campos da computação paralela e da computação distribuída tem sido desenvolvida separadamente, embora os dois campos lidem com problemas comuns. Essa separação é especialmente clara quando se refere ao desenvolvimento de paradigmas e modelos de programação. Na computação paralela, o foco direcionado fortemente para o desempenho das aplicações limita o uso de modelos mais flexíveis de interação entre processos. Entretanto, com a popularidade da computação em grade (Foster/01), esse cenário está mudando. Uma razão para isso é que a necessidade de usar recursos de diferentes domínios geográficos e administrativos força o desenvolvedor de aplicações paralelas a lidar com questões que poderiam ser ignoradas em contextos mais restritos, como é o caso dos *clusters*. Outra razão para a mudança de comportamento é a filosofia da computação em grade que visa aproveitar os recursos ociosos em diferentes instituições. Isso torna o ambiente de execução mais dinâmico e requer que o foco em desempenho seja atenuado para ceder lugar ao tratamento de outras restrições do contexto. Oferecer abstrações de programação adequadas para cenários como esse tem adquirido mais importância (Getov/01).

Neste capítulo, aprofundamos a discussão sobre essa linha de investigação. Considerando um conjunto de processos localizados em máquinas distintas que cooperam para a realização de uma atividade global, descrevemos a

construção de um ambiente de computação para a execução desses processos tomando como base o modelo discutido no capítulo anterior. Implementamos um conjunto de operações que oferecem possibilidades distintas de interação entre os processos: o programador pode explorar, simultaneamente, o estilo ativo do modelo de chamadas remotas e o estilo reativo da notificação de eventos. Exemplificamos como esses estilos de programação favorecem a construção de aplicações que refletem o comportamento mestre/trabalhador — característico das aplicações paralelas — dentro de contextos mais dinâmicos, como é o caso das redes geograficamente distribuídas. Para implementar as operações usamos a linguagem Lua (Ierusalimschy/03) aproveitando, em particular, as facilidades de funções como valores de primeira classe, o conceito de *closure* e a construção de co-rotinas providos pela linguagem.

3.1 Um ambiente para interação entre processos de uma aplicação

A interação entre processos que se comunicam via uma rede de computadores requer consideração especial sobre a possibilidade de atrasos na comunicação, falhas nos processos remotos ou desconexões temporárias, assim como estratégias para o roteamento de mensagens e para a manutenção de informações sobre a configuração da rede. As operações que implementamos auxiliam a lidar com essas características no escopo de desenvolvimento das aplicações, considerando a necessidade de dois ou mais processos interagirem e cooperarem para a execução de uma atividade global. Não faz parte do escopo deste trabalho tratar questões específicas da estrutura de comunicação da rede. Desse modo, assumimos a existência de uma camada básica, responsável por encaminhar apropriadamente as mensagens trocadas entre dois ou mais processos. Sobre essa camada implementamos o modelo que estamos propondo.

Dois paradigmas de programação são comummente usados para codificar a interação entre os processos de uma aplicação: um paradigma de natureza *ativa*, no qual um processo requisita a execução de um serviço ou de uma operação a outro processo quando necessário; e outro paradigma de natureza *reativa*, através do qual um processo avisa outro processo que deseja ser notificado sempre que um determinado evento ocorrer. Os modelos arquiteturais mais usuais incorporam um desses paradigmas. Na arquitetura cliente/servidor, um processo *cliente* envia uma requisição para um processo *servidor* e espera pela resposta para continuar o seu processamento, assumindo o estilo ativo de execução. Na arquitetura produtor/consumidor, um processo *consumidor* avisa um processo *produtor* sobre o seu interesse em ser notificado sobre a ocorrência de um evento gerado por ele, assumindo o estilo reativo de

execução. Nos dois casos, um mesmo processo pode assumir os dois papéis: um processo pode agir como servidor e cliente de serviços, ou como produtor e consumidor de eventos.

As abstrações de programação se adequam a uma dessas arquiteturas. A abstração de *chamada remota de procedimento* (RPC) (Birrell/83) e a sua variante, *invocação remota de método* (RMI) (Edwards/99) são normalmente usadas para modelar aplicações que refletem a arquitetura cliente/servidor; enquanto a abstração *publish/subscribe* (Bates/98) é normalmente usada quando a aplicação é modelada com base na reação à ocorrência de eventos. No conjunto de operações que implementamos, oferecemos ao programador a possibilidade de usar, dentro de uma mesma aplicação, esses dois tipos de abstração: invocações remotas síncronas e assíncronas e anúncio de eventos.

Definimos que o comportamento de cada processo deve refletir o comportamento típico de um "servidor" no contexto dos protocolos TCP/IP (Comer/96): uma aplicação executa algum código de inicialização e depois permanece em um laço através do qual espera por conexões de outros processos. Quando uma conexão é estabelecida, a mensagem recebida é imediatamente tratada e então o controle volta para o laço principal para esperar por novas conexões. Toda mensagem destinada a um processo está associada à execução de uma função local que pode corresponder: (a) a um serviço oferecido publicamente pelo processo, (b) a uma função de retorno para tratar os resultados de uma requisição anterior. Além da informação sobre a função que deve ser executada, a mensagem carrega também os argumentos necessários para a execução da função. No caso da mensagem conter a resposta de uma requisição, os argumentos são exatamente os resultados do processamento remoto.

Nesta seção descrevemos a implementação desse ambiente, denominado LuaRPC. Apresentamos algumas características da linguagem Lua, descrevemos o conjunto de operações de interação providas pelo ambiente e exemplificamos o uso dessas operações em pequenos trechos de código.

3.1.1 A linguagem Lua

Lua (Ierusalimschy/03) é uma linguagem interpretada com uma sintaxe similar a da linguagem Pascal. Uma das principais características de Lua é sua simplicidade e flexibilidade. Usando Lua, é possível explorar as vantagens oferecidas por conceitos como *closures* e funções como valores de primeira classe para implementar abstrações de programação de nível mais alto. Outra vantagem de Lua é o fato de poder ser usada como uma linguagem de

cola (Ousterhout/98), permitindo que a computação mais pesada seja escrita em outras linguagens, como C ou C++ (Ururahy/02). Essa característica é particularmente útil no desenvolvimento de aplicações paralelas.

Dois tipos de dados usados em Lua merecem especial atenção para a nossa implementação: *table* e *function*. Tabelas podem crescer e diminuir dinamicamente e implementam vetores associativos, i.e., seus índices podem ser valores de tipos arbitrários. Para criar uma tabela é necessário usar um *construtor* cuja forma básica corresponde a um simples par de chaves {}. Esse construtor cria uma tabela vazia, como mostra o exemplo abaixo:

```
mytable = {}     -- cria uma tabela
mytable["x"] = 3 -- nova entrada, com indice "x" e valor 3
if (mytable.x == 3) then ... -- o mesmo que mytable["x"]
```

Funções em Lua são valores de primeira classe, ou seja, são tratadas da mesma forma que valores básicos, como números e strings. Assim, uma função pode ser armazenada em variáveis ou tabelas, passada como argumento ou devolvida por outras funções. O *construtor* de funções corresponde a uma sentença que cria um valor do tipo *function*, como ilustrado a seguir:

```
f = function (x) return x+1 end
```

O mesmo construtor pode ser usado com uma sintaxe mais convencional, como mostra o exemplo abaixo:

```
function f (x) return x + 1 end
```

As funções podem receber um número variável de argumentos. Para indicar essa possibilidade, a lista de parâmetros é substituída por três pontos (...), como no exemplo abaixo:

```
function f (...)
 for i,v in ipairs(arg) do print(v) end
end
```

Quando a função `f` é chamada, todos os argumentos passados são inseridos em uma tabela denominada *arg*.

Em Lua, as funções possuem escopo léxico, ou seja, podem ser embutidas dentro de outras funções ganhando acesso total às variáveis da função mais externa. Essa facilidade, somada à característica de funções como valores de primeira classe, permite construções como a seguinte (extraída do livro (Ierusalimschy/03)):

```
function criaContador ()
 local x = 0
 return function ()
  x = x + 1
  return x
 end
end
```

A variável `x` é denominada *variável local externa* e é usada pela função interna para manter o valor de um contador. Quando a função interna (que é devolvida pela função `criaContador`) é chamada, a variável `x` está fora de escopo porque a função `criaContador` já terminou sua execução. Essa situação é tratada por Lua através do conceito de *closure*. O conceito de *closure* engloba a noção de uma função e tudo o que é necessário para acessar as variáveis locais externas apropriadamente. No trecho de código que segue, a função `criaContador` é usada para criar dois contadores:

```
contadorA = criaContador()
print(contadorA()) --> 1
print(contadorA()) --> 2
contadorB = criaContador()
print(contadorB()) --> 1
print(contadorA()) --> 3
```

Nesse exemplo, `contadorA` e `contadorB` são *closures* da função `criaContador`. A possibilidade de usar *closures* simplifica a construção de outras facilidades de programação, como funções de retorno (*callbacks*) e redefinição de funções.

A linguagem Lua implementa co-rotinas *completas* e *assimétricas* (Moura/04, Moura/04). Isso significa que as co-rotinas em Lua são valores de primeira classe e podem ser suspensas e retomadas dentro de funções aninhadas (característica de co-rotinas completas); e que a transferência de controle é feita por meio de duas operações distintas: uma para invocar uma co-rotina, e outra para suspendê-la, retornando o controle implicitamente para a co-rotina anterior (característica de co-rotinas assimétricas).

As operações de co-rotina em Lua são implementadas pelo pacote `coroutine`. A operação `coroutine.resume` transfere o controle para a co-rotina passada como argumento; e a operação `coroutine.yield` devolve o controle para a co-rotina anterior. Lua permite trocar dados através dessas operações de transferência de controle. Uma co-rotina é criada através da operação `coroutine.create` que recebe como argumento uma função. Quando o controle é transferido para a co-rotina pela primeira vez, ela inicia executando a

```
co = coroutine.create(function ()
       for i=1,10 do
         print(i)
         coroutine.yield()
       end
     end)
...
coroutine.resume(co) --> 1
coroutine.resume(co) --> 2
...
coroutine.resume(co) --> 3
```

Figura 3.1: Exemplo de uso de co-rotina em Lua.

primeira sentença dessa função. Se a operação `coroutine.yield` é chamada, ou a função executada pela co-rotina finaliza, o controle volta para a sentença imediatamente após a chamada da co-rotina. Se o controle foi devolvido através da operação `coroutine.yield`, a co-rotina ainda pode ser retomada. Quando isso acontecer, sua execução continuará a partir da última chamada de `coroutine.yield`.

O uso de co-rotinas em Lua é ilustrado na Figura 3.1 (através de um exemplo extraído do livro (Ierusalimschy/03)). A função executada pela co-rotina implementa um laço que imprime um valor e devolve o controle. Cada vez que a co-rotina é retomada, o próximo valor é impresso.

3.1.2 O ambiente LuaRPC

Usamos as facilidades de Lua para construir o ambiente LuaRPC: um ambiente de execução de processos com um conjunto de operações para a interação entre eles. Em linhas gerais, o laço principal de cada processo e as funções ativadas pelas mensagens recebidas são tratadas como co-rotinas. Quando um processo recebe uma mensagem para executar uma função oferecida por ele ou uma função de retorno, uma nova co-rotina é criada e ativada. Caso contrário, se a mensagem contém a resposta de uma chamada síncrona, a co-rotina apropriada é restaurada. Quando uma operação síncrona é invocada, uma função de retorno é implicitamente criada para restaurar e retomar a co-rotina corrente quando o resultado da chamada síncrona for recebido. Em seguida, a linha de controle é devolvida para o laço principal. Se o processo precisa enviar os resultados de uma invocação ou chamar uma operação remota, uma requisição de conexão é iniciada para o processo específico (ou para um conjunto de processos).

Para construir o ambiente de experimentação usamos a biblioteca `LuaSocket` (Nehab/04) — uma versão simplificada da definição de *sockets* de

Berkeley. A biblioteca `LuaSocket` oferece uma implementação do mecanismo de *timeout* que permite especificar limites superiores para o tempo em que um processo espera para estabelecer uma conexão ou receber uma mensagem. Essa característica facilita o uso de *sockets* como mecanismo base para implementar uma estrutura de comunicação não-bloqueante.

Com base nesse ambiente de execução, as sequintes operações são oferecidas:

- `luarpc.start(processCode, localHost, localPort)`: Inicia um processo especificando o seu código, a máquina e a porta onde ela irá executar.
- `luarpc.SyncCall(funcId, host, port)`: Devolve uma função que faz uma invocação remota síncrona. Quando a função devolvida é chamada o efeito é o de uma chamada síncrona, i.e., o fluxo de controle irá continuar apenas quando a chamada tiver sido completada. O controle volta imediatamanente para o laço principal que poderá tratar outras requisições.
- `luarpc.AsyncCall(funcId, host, port, <callback>)`: Devolve uma função que faz uma invocação remota assíncrona. Quando a função devolvida é chamada, o fluxo de controle corrente prossegue executando a próxima sentença do código, enquanto a mensagem com a invocação é enviada pela rede e tratada pelo processo remoto. Os parâmetros para a operação incluem uma identificação da função remota e opcionalmente a especificação da função que deverá ser executada no processo local para tratar os resultados da operação requisitada. Se a função de retorno não é informada, nenhum resultado será devolvido. Essa opção é útil quando o processo requisitante precisa que uma computação remota seja executada mas não precisa tratar os seus resultados (similar às chamadas *oneway* do padrão CORBA (Yang/96)), ou quando um processo precisa anunciar um evento.
- `luarpc.BcastAsyncCall(funcId, <callback>)`: Essa operação é uma variante particular de `luarpc.AsyncCall` e é usada para interagir simultaneamente com um conjunto de processos. A função de retorno é executada para cada resultado recebido. Usamos um mecanismo simplificado, baseado em um arquivo de configuração com a lista de endereços dos processos, para eliminar a necessidade de uma estrutura de registro expansiva.
- `luarpc.FutureCall(funcId, host, port)`: Permite um tipo de "sincronização postergada", como será discutido na seção 3.1.4. Devolve uma função que faz uma invocação remota assíncrona. Entretanto, ao

invés de definir uma função de retorno, como no caso da operação `luarpc.AsyncCall`, outra abordagem é adotada. Quando a função devolvida é chamada, ela devolve outra funcão que poderá ser invocada posteriormente para receber e tratar os resultados da invocação remota dentro do mesmo contexto de execução ou em um contexto distinto.

Para que os processos em diferentes máquinas possam interagir usando *sockets*, é preciso conhecer a identificação da máquina e da porta onde cada processo executa. Por isso, as operações implementadas incluem na lista de parâmetros essa informação. Como simplificação, consideramos que os possíveis endereços dos processos de uma aplicação são pré-definidos e estabelecidos na implantação da aplicação.

A operação `luarpc.FutureCall` reproduz a noção da promessa de um valor que será recebido e tratado no futuro (Lisbov/88, Walker/90, Gorlatch/03), adiando a sincronização para o ponto da execução onde o valor requisitado é de fato necessário. Essa abordagem pode ser usada pelo programador quando a lógica do programa define que um determinado valor será necessário em uma etapa mais adiante do processamento. A sincronização adiada permite minimizar os efeitos do retardo da comunicação.

As operações `luarpc.AsyncCall` e `luarpc.BcastAsyncCall` podem ser usadas para implementar lógicas distintas de interação entre os processos, por exemplo: (a) requisitar a execução de um serviço cuja resposta, quando houver, será recebida e processada pela função de retorno; (b) divulgar um serviço — nesse caso a função de retorno será chamada sempre que o serviço for solicitado; ou (c) registrar interesse em um evento — nesse caso a função de retorno pode ser vista como um tratador de evento que será chamado sempre que uma notificação do evento for recebida.

Como descrito, as operações do ambiente LuaRPC devolvem uma função ao invés de proceder com a invocação remota diretamente. Assim, a função que de fato faz a invocação remota pode ser criada uma única vez e invocada várias vezes, alterando apenas os valores dos argumentos passados. Com essa facilidade, o programador pode fazer as chamadas remotas com a mesma semântica das chamadas locais. Para prover essa simplificação, exploramos o conceito de *closure* e de funções como valores de primeira classe. Usando esses conceitos, as operações do ambiente LuaRPC devolvem funções construídas dentro delas. Essas funções encapsulam os valores dos argumentos passados para a operação que as criou, e cada vez que são chamadas elas podem receber seus próprios argumentos.

Para exemplificar, a Figura 3.2 ilustra o uso da operação `luarpc.AsyncCall`. O parâmetro `remoteFunc` é uma string que identifica

```
function callback(ret)
 if not ret.error then print(ret.results) end
end
f = luarpc.AsyncCall(remoteFunc, host, port, callback)
f(a, b) -- chamada remota
f(x, y) -- chamada remota
```

Figura 3.2: Exemplo de uso da operação `luarpc.AsyncCall`.

```
function luarpc.AsyncCall(funcId, host, port, callback)
  callbackId = getId(callback)
  -- retorno
  return function(...)
    return net.sendMsg(funcId, host, port, callbackId, arg)
  end
end
```

Figura 3.3: Esqueleto da operação `luarpc.AsyncCall`.

a operação remota, e o parâmetro `callback` é uma função local que será executada para tratar os resultados da invocação. O valor devolvido por `luarpc.AsyncCall` é uma função regular da linguagem Lua que pode ser atribuída a uma variável e invocada quantas vezes for necessário. A cada invocação dessa função, a mesma operação remota é chamada com argumentos diferentes. Quando a chamada remota é concluída, a função de retorno é executada recebendo como argumento uma tabela com os campos `error` e `results`. O campo `error` permite sinalizar a ocorrência de erros e o campo `results` armazena os resultados da operação.

A Figura 3.3 mostra o esqueleto da implementação da operação `luarpc.AsyncCall`. A função de retorno é armazenada localmente e um identificador dessa função (`callbackId`) é enviado com a requisição remota para ser associado aos resultados que serão devolvidos posteriormente. A operação `net.sendMsg` interage com a biblioteca de *sockets* para enviar a mensagem para o processo destino. Usando o mecanismo de *closure*, a operação `luarpc.AsyncCall` cria e retorna uma função anônima que usa os valores das variáveis `funcId`, `host`, `port` e `callbackId`, específicos do escopo léxico local. Sempre que o programador chama a função anônima retornada por `luarpc.AsyncCall`, essas variáveis estarão fora de escopo, mas encapsuladas na definição da nova função.

Linguagens convencionais (como C e C++), que não oferecem suporte para *closure*, exigiriam um esforço adicional para construir um mecanismo como esse. Como salientam Adya et al. (Adya/02), sem a noção de *closure* é

```
function request()
  local acc, repl = 0, 0
  function avrg (ret)
    repl = repl + 1
    acc = acc + ret.results
    if (repl == expected) then
      print ("Current Value: ", acc/repl)
    end
  end
  remoteCall = luarpc.BcastAsyncCall("currValue", avrg)
  remoteCall()
end
```

Figura 3.4: Usando o mecanismo de *closure* no processo de *stack ripping*.

preciso gerenciar diretamente características procedurais da linguagem. Nesse caso, seria preciso que duas ou mais funções da linguagem pudessem representar uma função conceitual única; e as variáveis automáticas precisariam ser armazenadas em um espaço apropriado na memória para serem preservadas ao longo dos possíveis pontos de invocação das funções.

O mecanismo de *closure* permite diminuir a dificuldade com o processo de *stack ripping*, caracterizado por Adya et al. (Adya/02). Quando um tratador de evento faz uma requisição e precisa registrar uma *continuação* para ser executada quando a resposta da requisição estiver disponível, o mecanismo de *closure* pode ser usado para encapsular os valores que precisam ser preservados ao longo do tratamento dessa requisição. O exemplo apresentado na Figura 3.4 ilustra essa idéia. Nesse exemplo, a função `request` deve calcular a média dos valores fornecidos por um conjunto de processos. Para isso, a operação `luarpc.BcastAsyncCall` requisita a leitura do valor em cada processo e define a função `avrg` como função de retorno. Essa função é um *closure* da função `request` e por isso é capaz de manter os valores das variáveis `acc` e `repl` (usados para calcular a média) enquanto processa a informação devolvida pelos processos remotos.

3.1.3 Visão síncrona usando comunicação não bloqueante

Com as chamadas assíncronas, a invocação de uma função remota é explicitamente particionada em duas fases: a primeira com a invocação e a segunda com o tratamento dos resultados (usando as funções de retorno para codificar a continuação da computação depois que os resultados se tornam disponíveis). Em alguns casos, esse comportamento reflete naturalmente a

```
function luarpc.SyncCall(funcId, host, port)
  -- retorna uma funcao
  return function(...)

     -- gerador da funcao de retorno
     function callback_sync(currentCo)
       return function(...)
         luarpc.currentCo = currentCo
         -- restaura a co-rotina que foi suspensa
         coroutine.resume(currentCo, unpack(arg))
       end
     end

     -- cria a funcao de retorno
     callback = callback_sync(luarpc.currentCo)
     callbackId = getId(callback)

     -- envia a mensagem de requisicao
     if net.sendMsg(funcId, callbackId, host, port, arg) then
       -- suspende a co-rotina atual
       return coroutine.yield()
     else return nil end
  end
end
```

Figura 3.5: Esqueleto da operação `luarpc.SyncCall`.

lógica do problema tratado. Por exemplo, quando os resultados de uma requisição podem ser recebidos e processados em momentos ou etapas distintas da aplicação. Em outros casos, uma visão síncrona é mais apropriada, especialmente se a continuidade do fluxo de execução depende dos resultados esperados.

Usando co-rotinas, encapsulamos o comportamento das chamadas assíncronas com funções de retorno dentro da função `luarpc.SyncCall`. A função de retorno é implicitamente construída e representa a continuação da computação corrente. Assim, quando o programador escreve o trecho de código abaixo, a atribuição para a variável `ret` ocorrerá apenas depois que a invocação remota tiver sido concluída.

```
f = luarpc.SyncCall(funcId, host, port)
-- executa outras atividades
...
ret = f()
```

A Figura 3.5 mostra o esqueleto da implementação da operação `luarpc.SyncCall`. A função interna `callback_sync` cria a função de retorno que, quando chamada, restaura a co-rotina corrente, passando os resultados recebidos como argumentos para a co-rotina. Assim que a requisição é

enviada pela rede, a co-rotina corrente é suspensa executando a sentença `coroutine.yield()`. A variável `luarpc.currentCo` é usada para armazenar a informação sobre a co-rotina ativa no momento.

3.1.4 Sincronização postergada

Um caminho intermediário para implementar a interação síncrona entre processos baseia-se nos mecanismos de *Futures* (Walker/90) e *Promises* (Lisbov/88). *Future/Promise* são mecanismos linguísticos que permitem armazenar uma referência para a "promessa" de um valor que será recebido e tratado no futuro. A idéia básica é que o processamento local continue normalmente e em paralelo com a interação remota (como uma invocação assíncrona). O ponto de sincronizaão ocorre em um passo mais adiante do código do processo requisitante. Quando o processo chega nesse ponto da execução é que a tarefa corrente é suspensa para esperar pelos resultados da interação, caso eles ainda não estejam disponíveis. Esse mecanismo favorece, em especial, a sobreposição da computação local com a comunicação entre as partes de uma aplicação. Além disso, permite tratar de forma mais simples casos em que várias respostas são esperadas para uma única interação, ou os resultados da computação remota são recebidos de forma incremental.

Incluímos na implementação de LuaRPC um mecanismo similar de *sincronização postergarda* por meio da operação `luarpc.FutureCall`. Quando o programador sabe, em certo ponto da execução, que é preciso escalonar uma computação para obter uma informação que será usada mais tarde, a opção pela sincronização postergada é uma opção razoável para aumentar o paralelismo da aplicação. Nesse caso, uma chamada especial para uma função remota irá proceder assincronamente e em paralelo com a computação local. O valor devolvido localmente será uma função que poderá ser invocada posteriormente, dentro do mesmo contexto de execução ou em um contexto distinto. Apenas nesse ponto é que a sincronização necessária irá de fato ocorrer.

A Figura 3.6 ilustra um exemplo de sincronização postergada. Quando a função designada por `futureQuery` é chamada, o envio da invocação remota e a execução da função `newQuery` poderá acontecer em paralelo com o restante da computação local até o ponto de sincronização, definido na sentença que chama a função `getResult`. Quando `getResult` é invocada, o tratamento é o mesmo de uma invocação síncrona, ou seja, a co-rotina é suspensa e uma função de retorno é implicitamente criada para restaurar essa co-rotina quando o resultado for recebido. A sentença *"outras atividades"* é destacada para

```
futureQuery = luarpc.FutureCall("newQuery", host, port)
getResult = futureQuery(arguments) -- devolve a promessa de valor futuro
-- ! outras atividades !
res = getResult() -- agora espera pelos resultados
```

Figura 3.6: Usando o mecanismo de sincronização postergada.

indicar que a aplicação poderia incluir outras invocações remotas síncronas ou assíncronas antes de requisitar a sincronização e esperar pelos resultados da função `getResult`.

3.2 Exemplos de uso do ambiente LuaRPC

Usamos as operações oferecidas pelo ambiente LuaRPC para experimentar a construção de pequenos exemplos de aplicações que refletem o relacionamento mestre/trabalhador, onde um processo *mestre* distribui a execução de um conjunto de tarefas entre vários processos *trabalhadores*. A Figura 3.7 mostra o esqueleto de um programa que distribui tarefas no estilo *round-robin* para serem executadas por um conjunto de processos trabalhadores. A tabela `publicFunctions` é usada para armazenar as funções que implementam os serviços oferecidos publicamente por cada processo. A tabela `workers` armazena a identificação da máquina e da porta onde cada processo trabalhador espera por tarefas.

Depois de enviar todas as tarefas para os trabalhadores remotos, o programa entra em um laço à espera de eventos de comunicação. A tabela com as funções públicas (`publicFunctions`) é passada como argumento para a função `luarpc.loop`. Nesse exemplo, essa tabela contém apenas a função `handleresults`, que é invocada remotamente pelos processos trabalhadores para enviar os resultados das tarefas processadas. O processo mestre distribui todas as tarefas inicialmente e define a função `handleresults` para receber e tratar os resultados esperados. Uma outra estratégia poderia ser implementada para permitir a distribuição de tarefas por demanda. Nesse caso, uma única tarefa seria enviada inicialmente e a função `handleresults` trataria o envio das demais tarefas.

Em um outro exemplo, consideremos novamente o problema de distribuir tarefas, mas desta vez os processos trabalhadores continuamente oferecem seus serviços de processamento — explorando o estilo reativo de execução. Para isso, eles definem uma função remota, denominada `availableWorker`, que inicia a execução de um laço. Dentro desse laço principal, o trabalhador solicita repetidamente ao mestre a descrição de uma nova tarefa, executa a

```
publicFunctions = {}

-- funcao para tratar os resultados das tarefas remotas
function publicFunctions.handleresults (ret)
  if ret.error then -- tratador de erros
  else
    total = total + ret.results
    workleft = workleft - 1
    if (workleft==0) then
      -- mostra os resultados e termina
    end
  end
end

-- tabela para armazenar as tarefas que devem ser executadas
remotework = {}

for _,worker in pairs(workers) do  -- percorre a tabela 'workers'
  table.insert(remotework, luarpc.AsyncCall("processtask",
                worker.host, worker.port, handleresults)
end

-- distribui as tarefas, a tabela 'tasks' contem as tarefas
work = next(remotework, nil)    -- percorre a tabela 'remotework'
  workleft = 0
  for _,task in pairs(tasks) do -- percorre a tabela 'tasks'
    remotework[work](task) -- envia uma tarefa para um trabalhador remoto
    workleft = workleft + 1
    work = next(remotework, work) or next(remotework, nil)
  end

luarpc.loop(publicFunctions)
```

Figura 3.7: Código para uma distribuição de tarefas do tipo *round-robin*.

tarefa recebida e retorna os resultados. As Figuras 3.8 e 3.9 mostram o código principal dos processos que executam o programa trabalhador e o programa mestre, respectivamente.

Quando a função `availableWorker` é invocada, os valores `getWork` e `putResult` são criados usando o argumento recebido. `availableWorker` chama a função `work`, a qual implementa um laço para receber tarefas do mestre e enviar os resultados. Do lado do mestre, como mostra a Figura 3.9, a execução começa com uma chamada à função remota `availableWorker`. A operação `luarpc.BcastAsyncCall` é usada para enviar a invocação para todos os trabalhadores ativos no momento.

3.3

```
publicFunctions = {}

function publicFunctions.availableWorker(master)
  local getWork = luarpc.SyncCall
                      (master.newTask, master.host, master.port)
  local putResult = luarpc.SyncCall
                      (master.taskCompleted, master.host, master.port)
  work(getWork, putResult)
end

function work(get, put)
  local w = get()
  while w do
    put(localwork(w))
    w = get()
  end
end

luarpc.loop(publicFunctions)
```

Figura 3.8: Processo trabalhador.

```
publicFunctions = {}

function publicFunctions.workUnit()
  -- if there are work units left
  --   return one work unit
  -- else
  --   return nil
  -- end
end

function publicFunctions.result(newresult)
  -- armazena o resultado
end
master = {taskCompleted="result", newTask="workUnit",
                  host=myHost, port=myPort}
bcast = luarpc.BcastAsyncCall("availableWorker", nil)
bcast(master)

luarpc.loop(publicFunctions)
```

Figura 3.9: Processo mestre.

Trabalhos relacionados

Plataformas para o desenvolvimento de aplicações distribuídas (como CORBA (Yang/96), DCOM (Redmond/97), JavaRMI (JavaRMI/99), ICE (ICE), etc.) visam facilitar a tarefa do programador modelando as chamadas de métodos remotos de modo a serem tão simples de usar quanto as chamadas de métodos locais. Um dos aspectos fundamentais nessa direção consiste em preservar a visão da programação síncrona, onde a linha de execução bloqueia até que a resposta da invocação remota esteja disponível. Tipicamente, essas plataformas simulam o comportamento síncrono para beneficiar os programadores: quando uma aplicação faz uma invocação síncrona, a mensagem correspondente é enviada e a *thread* de execução é bloqueada até que a resposta esteja disponível. Para os casos em que a programação síncrona é custosa, i.e., quando a aplicação pode executar outras tarefas enquanto espera pelos resultados de uma invocação remota, um modelo para invocação assíncrona de métodos é normalmente usado.

O ambiente de execução proposto por LuaRPC segue a mesma direção: oferecer uma interface de programação familiar para o programador favorecendo a otimização do uso dos recursos computacionais disponíveis. A diferença está na opção explícita por explorar a gerência cooperativa de tarefas e de trazê-la para o nível da aplicação usando co-rotinas. Com essa opção eliminamos a possibilidade de explorar o paralelismo real entre linhas de execução independentes (nos casos em que ele é possível), mas por outro lado eliminamos também a dificuldade de sincronizar essas linhas de execução quando é necessário tratar o acesso a recursos compartilhados.

O padrão AMI(*Asynchronous Method Invocation*) (OMG/AMI, Vinoski/98), usado para a invocação assíncrona de métodos remotos na plataforma CORBA, permite implementar um comportamento assíncrono dentro dos pares requisição/resposta ao invés de usar um serviço CORBA adicional, como Eventos ou Serviço de Notificação. A definição CORBA AMI define dois modelos básicos: (i) *polling* e (ii) *callback*. No modelo *polling* cada invocação AMI retorna uma entidade com dados e métodos. O cliente pode usar os métodos dessa entidade (os quais executam localmente) para verificar o estado da invocação remota e então obter os resultados enviados pelo servidor. Se o servidor ainda não respondeu à requisição, o cliente pode: (i) bloquear sua execução até que os resultados estejam disponíveis; ou (ii) continuar sua execução e fazer uma nova verificação posteriormente. O modelo *polling* provê uma funcionalidade similar aos mecanismos *Futures* (Walker/90) e *Promises* (Lisbov/88). No modelo *callback*, o cliente passa como parâmetro a referência de um objeto que irá tratar a resposta da invocação remota. Essa

referência não é passada para o servidor, mas armazenada localmente pelo cliente ORB. Quando o servidor responde, o cliente ORB ativa a função de retorno correspondente à mensagem recebida.

Arulanthu et. al. (Arulanthu/00) implementaram o o padrão CORBA AMI no ORB TAO (*CORBA complient ORB*), usando o modelo de chamadas de retorno. Os autores destacam algumas pontos importantes para implementar o padrão AMI, entre eles a decisão sobre como tratar as respostas das requisições, por exemplo, usando uma única *thread* e serializando o tratamento das respostas, ou usando um conjunto de *threads*. A opção por um conjunto de *threads* exige a sincronização do acesso aos recursos compartilhados. No ambiente LuaRPC o tratamento das respostas das invocações assíncronas é sempre serial e o programador pode optar pelo modelo *polling* (com bloqueio), usando a operação `luarpc.FutureCall` ou pelo modelo *callback*, usando a operação `luarpc.AsyncCall`.

A plataforma ICE (*Internet Communication Engine*) (ICE) — uma plataforma de *middleware* assíncrona para o desenvolvimento de aplicações cliente/servidor em ambientes heterogêneos — define um modelo de invocação de método assíncrono (AMI) e permite que o programador especifique o seu uso em dois níveis: para uma interface ou classe de objetos; ou para uma operação isoladamente. A especificação é feita no nível da linguagem de metadados e o código gerado para a aplicação inclui sempre a versão para invocação síncrona de todos os métodos remotos e acrescenta a versão assíncrona nos casos especificados pelo programador. Além do *proxy* para a invocação síncrona do método, o gerador de código cria um *proxy* para a invocação assíncrona e uma classe para a função de retorno: os parâmetros de retorno das invocações síncronas são removidos, e a aplicação informa o objeto de retorno (uma entidade local da aplicação) que será invocado com os resultados da operação. Para permitir invocações assíncronas do lado do cliente de uma aplicação, a plataforma ICE define um conjunto de *threads* cuja finalidade principal é processar as mensagens de retorno recebidas pela aplicação. O número pré-definido de *threads* é um, o que significa que as funções de retorno são tratadas de forma serial. Se esse número for aumentado, algum tipo de sincronização pode ser necessário para tratar o caso de diferentes funções de retorno acessarem recursos comuns.

A plataforama ICE define ainda um *despachador de métodos assíncronos* (AMD - *Asynchronous Method Dispatch*) como o equivalente ao método de invocação assíncrona (AMI) no lado do servidor. A idéia básica é que o servidor pode receber uma requisição e suspender o seu processamento para tratá-lo assim que for possível (quando um *thread* de execução torna-se disponível).

Quando o processamento é concluído, o servidor envia a resposta usando uma função de retorno. O objetivo é permitir que o servidor minimize o número de *threads* e ao mesmo tempo permaneça apto para atender vários clientes simultaneamente. Diferente do mapeamento usado para as invocações assíncronas do lado co cliente (o qual sempre permite que a aplicação continue a usar as invocações síncronas quando necessário), o mapeamento para operações AMD não permite o uso dos dois modelos de despachador: se a definição na linguagem de metadados especifica o uso do modelo AMD, o despachador síncrono do método é completamente substituído pelo despachador assíncrono. A assinatura do método inclui um objeto de retorno e os argumentos da operação, como nos métodos AMI. A diferença é que o objeto de retorno é definido pelo próprio ICE (ao invés de pela aplicação) e contém métodos para devolver os resultados da operação ou sinalizar a ocorrência de uma exceção.

A abordagem adotada na implementação do ambiente LuaRPC prioriza fortemente a flexibilidade oferecida ao programador. Do lado do cliente ou do lado do servidor é possível adotar estratégias distintas dentro da mesma aplicação, como por exemplo, postergar a execução de uma requisição. Um mesmo método remoto pode ser invocado de forma síncrona ou assíncrona, ou através de uma invocação síncrona postergada. Não é necessário gerar código adicional para permitir as diferentes opções, bastando usar as operações providas pelo ambiente da forma mais apropriada para cada estado da aplicação.

Nossa abordagem é similar ao projeto *ProActive* (Caromel/93), um *middleware* Java projetado para permitir programação concorrente, distribuída e paralela. As chamadas de métodos são sempre assíncronas e *objetos futuros* são transparentemente providos para coletar os resultados de uma chamada. Entretanto, o modelo do *ProActive* explora facilidades estáticas de verificação de tipos, enquanto em nosso trabalho exploramos as vantagens de tipos dinâmicos. Além disso, o *ProActive* usa multi-tarefa preemptiva, enquanto o nosso trabalho explora multi-tarefa cooperativa.