3
Abstrações e Coordenação

Ao contrário da programação convencional, na qual um programa é escrito como uma seqüência de ações, na programação orientada a eventos o desenvolvedor escreve um conjunto de tratadores, os quais são ativados pela chegada de eventos. No entanto, entender um programa que é escrito como uma série de respostas para diferentes eventos não é fácil quando o número de interações é grande. Um único processo pode, a qualquer momento, estar interagindo com dezenas de outros processos, e cada uma dessas interações pode requerer seu próprio estado. Isso reforça a necessidade de abstrações para coordenar a interação de processos.

O conceito de linguagem de coordenação como ferramenta para descrever a interação entre as partes de um programa distribuído foi muito discutido no anos noventa (Gelernter e Carriero, 1992; Papadopoulos e Arbab, 1998). Naquele tempo, a maior parte da discussão foi focada sobre a idéia de usar um conjunto pré-definido de primitivas de coordenação, como manipulação de espaço de tuplas de Linda (Carriero e Gelernter, 1989), para definir a comunicação e sincronização da aplicação. Modelos de coordenação propostos naquele tempo eram geralmente focados em modelos de aplicação fortemente acoplados. Hoje, a forma na qual abstrações de coordenação são programadas deve ser avaliada à luz das novas exigências impostas por orientação a eventos, fraco acoplamento e ambientes de execução dinâmica. São necessários mecanismos mais flexíveis, que possam ser redefinidos e combinados até mesmo em tempo de execução.

Primitivas para comunicação e sincronização têm classicamente sido oferecidas através de linguagens de programação de propósito geral ou bibliotecas. Linguagens criadas com o objetivo de dar suporte à programação distribuída geralmente provêem um modelo de programação consistente, mas são atreladas a padrões de comunicação pré-definidos pelos arquitetos da linguagem. Bibliotecas de comunicação para linguagens convencionais têm a vantagem de poderem ser livremente combinadas em uma única aplicação, permitindo ao programador escolher o melhor padrão para cada conjunto de interações. No entanto, há casos em que as bibliotecas devem lidar com uma lacuna entre o modelo que elas implementam e o modelo da linguagem em que estão sendo utilizadas. E cada biblioteca, além dos diferentes modelos de comunicação, também impõe um modelo de programação diferente,

tornando complicado para o programador lidar com várias delas ao mesmo tempo.

A escala e a diversidade de interações em aplicações para Internet aponta para a idéia que nenhum conjunto pré-definido de padrões de interação será suficiente, mesmo considerando classes específicas de aplicações. Isso indica a necessidade de ambientes nos quais diferentes padrões de abstrações e coordenação podem ser construídos e combinados. No entanto, seria interessante poder experimentar e combinar livremente abstrações sem ter que recair em interfaces de programação de nível relativamente baixo como as tradicionalmente oferecidas pelas bibliotecas.

Nós acreditamos que características das linguagens de programação podem dar uma contribuição significante nesse aspecto. Usando as construções apropriadas da linguagem, seria possível construir um número arbitrário de mecanismos de coordenação através de um pequeno conjunto de primitivas. Linguagens dinamicamente tipadas, devido aos seus sistemas de tipos flexíveis e facilidades de extensão, podem permitir que bibliotecas sejam incorporadas uniformemente, criando ambientes nos quais diferentes técnicas de coordenação podem ser usadas e combinadas para compor novos mecanismos. Assim, em vez de procurarmos um mecanismo de coordenação específico para programas orientados a eventos o qual seria melhor que os demais, deveríamos nos concentrar nesses ambientes, permitindo aos programadores criarem e combinarem ferramentas de coordenação facilmente.

Neste capítulo, defendemos esse argumento apresentando bibliotecas para coordenação em Lua e discutindo as características da linguagem que permitem que essas bibliotecas sejam facilmente construídas com APIs bem integradas com a própria linguagem. Especificamente, nós destacamos o papel de funções como valores de primeira classe, escopo léxico e co-rotinas. Trabalharemos em um ambiente distribuído, composto por vários processos executando o ALua. Consideramos que cada um desses processos tem uma única linha de execução (*single-threaded*).

A principal contribuição deste capítulo é mostrar o papel das características da linguagem em permitir que diferentes mecanismos de coordenação sejam construídos sobre um pequeno conjunto de primitivas, e sejam facilmente misturados e combinados.

3.1 RPC Assíncrono

Apesar do modelo básico de programação orientada a eventos do ALua, utilizando passagem de mensagens com trechos de código Lua, ser poderoso, ele pode ser bastante propenso a erro e difícil de se utilizar. Os programadores precisam de ferramentas que os permitam modelar padrões de interação de mais alto nível. Isso é onde as abstrações de programação entram. Para permitir que o programador lide com conceitos de mais alto nível, várias bibliotecas de comunicação foram

implementadas nos últimos anos, provendo suporte a espaços de tuplas (Leal et al., 2003), publish-subscribe (Rossetto et al., 2004) e chamadas remotas de procedimentos (Rodriguez e Rossetto, 2008), dentre outras. Neste trabalho, usamos o mecanismo de chamadas remotas de procedimento proposto em (Rodriguez e Rossetto, 2008) como mecanismo base, sendo discutido a seguir.

Desde sua concepção, o mecanismo de chamada remota de procedimento (RPC) foi alvo de uma série de críticas (Tanenbaum e Renesse, 1988; Birman e Renessee, 1994), em especial devido a sua natureza síncrona e conseqüente forte acoplamento entre cliente e servidor. Invocações assíncronas foram largamente discutidas como uma alternativa (Ananda et al., 1992), mas o fato é que elas não são confortáveis para o uso tradicional dos programas seqüenciais. Quando o programa é orientado a evento, no entanto, chamadas assíncronas são naturais, e podem ser associadas a funções de callback que serão executadas no ato da término da chamada remota.

O mecanismo de procedimento remoto provido pela biblioteca `rpc` (Rodriguez e Rossetto, 2008) explora essa idéia, associando chamadas assíncronas com funções de callback sobre um modelo orientado a eventos. O funcionamento básico de execução se mantém o mesmo descrito anteriormente, com um processo tratando cada chegada de mensagem por vez, com a diferença que agora a mensagem faz apenas uma chamada de função em vez de conter trecho código arbitrário.

Para prover a mesma flexibilidade que temos com funções sendo valores normais em Lua, a primitiva `rpc.async` não realiza diretamente a chamada remota. Em vez disso, ela retorna uma função que chama o método remoto (com seus argumentos apropriados). Como exemplo, voltemos ao exemplo de cálculo da média da figura 2.2. O laço `for` que realiza a requisição aos valores remotos pode ser reescrito utilizando `rpc.async` para as chamadas assíncronas da seguinte forma:

```
for i = 1, expected do
  local sendRequest = rpc.async(peers[i], "getvalue", cb)
  sendRequest()
end
```

Os parâmetros obrigatórios para `rpc.async` são o processo remoto, `peers[i]`, e o nome da função remota, `"getvalue"`. O terceiro argumento é opcional e se refere a uma função de callback para tratar o valor de retorno da função remota. No exemplo acima, a função retornada por `rpc.async` é armazenada na variável `sendRequest` (lembrando que `rpc.async` não invoca a função remota, em vez disso ela cria uma função que fará a chamada). Quando a função `sendRequest` é chamada, a requisição remota é enviada e o controle retorna imediatamente para o chamador. Futuramente, quando o programa retornar para o loop de evento e receber o resultado da função remota, a cal1lback `cb` será invocada, recebendo o resultado

como argumento.

Duas características da linguagem são especialmente importantes para permitir que a primitiva `rpc.async` retorne a função que pode ser manipulada como qualquer outro valor: função como valor de primeira classe e escopo léxico. Com essas características, uma função pode retornar uma outra função aninhada, a qual tem total acesso às variáveis e argumentos de sua função criadora.

A figura 3.1 mostra a implementação (completa) da primitiva `rpc.async`. Basicamente, ela cria uma função (chamada `f`) que encapsula a invocação remota. Essa função recebe um número variável de argumentos, que são capturados na tabela `args`, serializados e enviados para o processo remoto. A função de callback (`cb`) é registrada para tratar os resultados quando eles chegam. A requisição é enviada para o processo remoto através da chamada à `alua.send`. Acreditamos que a concisão desta implementação reflete a importância de utilizar uma linguagem de programação com características apropriadas.

```
function rpc.async(dest, func, cb)
  local function f(...)
     -- Guarda os argumentos da função
     local args = {...}
     -- Registra a callback
     local idx = set_pending(cb)
     -- Serializa os argumentos
     args = marshal(args)
     local chunk = string.format("rpc.request(%q, %s, %q, %q)",
        func, args, localId, idx)
     -- Envia a requisição ao processo remoto
     alua.send(dest, chunk)
  end
  return f
end
```

Figura 3.1: Implementação da primitiva `rpc.async`.

A primitiva `rpc.async` retorna uma função, definida internamente, a qual depende dos valores passados como argumento a cada chamada. Cada vez que a função retornada é chamada, uma nova chamada remota é realizada, a qual utiliza os mesmos valores para o processo remoto, função remota e função callback, com diferentes argumentos para a função remota.

3.1.1 RPC Síncrono

Com chamadas assíncronas, o programador deve inverter o fluxo de controle, usando funções de callback para codificar a continuação da computação depois dos resultados da chamada estarem disponíveis. Isso reflete diretamente a natureza de um programa orientado a eventos, mas pode não ser o melhor modelo para

o programador trabalhar. Nesta seção, discutimos a função `rpc.sync`, que cria funções que fazem chamadas síncronas a outros processos sobre o mesmo modelo de comunicação assíncrona. Assim como a função `rpc.async`, a função `rpc.sync` recebe como parâmetros a identificação do processo e o nome da função remota. Por ser síncrona, o parâmetro de callback não faz sentido. De fato, uma callback que reinicia a computação corrente será construída implicitamente por `rpc.sync`.

Ilustramos a utilização de `rpc.sync` com o código da figura 3.2 que recupera repetidamente um valor de um processo remoto *procA*, realiza uma computação com esse valor e atualiza o processo remoto.

```
get = rpc.sync("procA", "getValue")
set = rpc.sync("procA", "setValue")

while true do
   oldvalue = get()
   newvalue = transform(oldvalue)
   set(newvalue)
end
```

Figura 3.2: Exemplo da utilização de `rpc.sync`

Para a implementação de `rpc.sync`, recaímos sobre a facilidade de multithreading cooperativa oferecida pelas co-rotinas de Lua. A co-rotina mantém sua própria pilha de execução, mas ela deve utilizar primitivas para a transferência explícita do controle para outra co-rotina. Isso é interessante porque evita a complexidade de condições de corrida, mas por outro lado ele deixa para o programador a responsabilidade de gerenciar a transferência de controle. No caso do RPC síncrono, a transferência de controle é encapsulada automaticamente na chamada remota. Cada nova computação é tratada em uma nova co-rotina e, quando a chamada síncrona é realizada, a co-rotina corrente é suspensa e o fluxo de execução retorna para o loop do ALua.

Para implementar `rpc.sync`, nós utilizamos `rpc.async` como base e novamente o mecanismo de funções como valor de primeira classe acessando as variáveis do escopo léxico. A figura 3.3 contém um esboço dessa implementação.

Neste ponto é conveniente explicar algumas características das co-rotinas de Lua. As funções `coroutine.yield` e `coroutine.resume`, respectivamente, suspendem e retomam a execução da co-rotina. `coroutine.resume` recebe como primeiro parâmetro a co-rotina a ser retomada e qualquer outro valor recebido será retornado por `coroutine.yield`. Da mesma forma, qualquer argumento passado para `coroutine.yield` será retornado por `coroutine.resume`. Isso cria um canal de comunicação entre as co-rotinas.

De volta à implementação de `rpc.sync`: quando a função `remote` é chamada, ela primeiro cria uma callback que será responsável por reiniciar a co-rotina

```
function rpc.sync(proc, func)
  -- Cria uma função para realizar a chamada remota.
  local function remote(...)
     -- Referência para a co-rotina corrente.
     local co = coroutine.running()
     -- Cria uma callback que retomará a execução uma vez
     --  que a chamada remota tenha terminado.
     local function callback(...)
         coroutine.resume(co, ...)
     end
     -- Efetua a chamada remota.
     local aux = rpc.async(proc, func, callback)
     aux(...)
     -- Suspende a execução da co-rotina corrente
     --  antes de retornar.
      return coroutine.yield()
  end
  return remote
end
```

Figura 3.3: Implementação de `rpc.sync`

corrente. Então, `remote` invoca `rpc.async` para realizar a comunicação remota (passando a callback interna) e suspende a execução com `coroutine.yield`. Quando os resultados chegam, a callback interna é chamada com os valores recebidos, os quais são repassados para `coroutine.resume`. A co-rotina é então reiniciada e os resultados são retornados para o chamador de `remote`.

Os modelos de RPC síncrono e assíncrono foram inicialmente propostos para Lua por Rossetto (Rodriguez e Rossetto, 2008), e as implementações foram feitas diretamente sobre socket. Reimplementamos os RPCs sobre o modelo de eventos do ALua e os utilizaremo em nossos estudos, descritos a seguir, sobre a criação e combinação de mecanimos para a coordenação e sincronização, explorando as características da linguagem de programação Lua.

3.2 Coordenando Atividades Concorrentes

O modelo que nós descrevemos nas seções anteriores evitam muitas das questões de sincronização. Como cada evento é tratado por completo, a sincronização de granularidade fina necessária para multithreading, devido à possibilidade de interferência arbitrária na execução, não é requerida. No entanto, em certos casos, ainda temos necessidade de um certo nível de sincronização no tratamento dos eventos.

Gelernter e Carriero (Gelernter e Carriero, 1992) discutem a vantagem de ver as primitivas de comunicação e sincronização como meios de *coordenação* de uma aplicação concorrente ou distribuída. Nesta seção, adotamos essa abordagem e discutimos como diferentes abstrações de coordenação podem ser providas por

bibliotecas que podem ser combinadas, mesmo como blocos de montar, para criar outras abstrações, ou simplesmente como alternativa para ser utilizada dentro de uma aplicação, quando necessário.

As abstrações que discutimos são aquelas relacionadas com a sincronização clássica entre processos concorrentes, para exclusão mútua e coordenação (Andrews, 2000). Mesmo se, em nosso modelo, os problemas da granularidade fina de sincronização, tal como a interferência de acesso a variáveis globais, são evitados, podemos ainda ter problemas ocorridos na granularidade grossa. Chamadas subseqüentes a um processo necessitam ocorrer com a garantia de que nenhum evento foi tratado entre elas (por exemplo, para garantir uma visão atômica de um conjunto de operações). Além disso, podemos ter a necessidade de sincronizar ações ocorrendo em diferentes processos. As seções 3.2.1 e 3.2.2 discutem o suporte à sincronização e coordenação.

3.2.1 Monitores

Quando utilizamos chamadas síncronas, permitimos que as co-rotinas fiquem bloqueadas enquanto esperam pelos resultados da chamada. Enquanto tais co-rotinas estão bloqueadas, outras chamadas podem modificar as globais compartilhadas, as deixando em um estado diferente do qual a co-rotina bloqueada espera estar quando for reiniciada. Nós introduzimos então a possibilidade de condições de corrida, mesmo que em uma granularidade muito maior que a de multithreading (com co-rotinas, a "troca de contexto" pode somente ocorrer em pontos específicos do código), e a eventual necessidade de mecanismos de exclusão mútua. Em vez de prover um mecanismo diretamente na linguagem, acreditamos que abstrações podem ser facilmente implementadas sobre as primitivas existentes e integradas dentro de uma dada aplicação, como biblioteca ou como um módulo da aplicação. Por exemplo, nesta seção discutiremos uma implementação de monitores (Hoare, 1974). Monitores aqui descritos são diferentes da proposta clássica, pois eles são dinâmicos: funções podem ser adicionadas ao monitor em qualquer ponto da execução.

Nossa implementação para os monitores é baseada em uma chamada síncrona para adquirir um lock, que suspende a execução até que esse lock seja adquirido. Implementamos um *monitor* como uma estrutura contendo um lock booleano, que indica se o monitor está livre, uma fila de entrada e o identificador de seu criador. A função `monitor.create` cria um novo monitor (sem funções incluídas) e retorna uma referência para o mesmo. Depois que um monitor "vazio" é criado, funções arbitrárias podem ser postas sob sua proteção através da chamada de função `monitor.doWhenfree`. Esse mecanismo foi inicialmente proposto em (Lima, 1992) e

utilizava *continuação* em Lua, descrito no mesmo trabalho – nossa implementação é baseada co-rotinas e chamadas remotas. Como exemplo de nossa modelo de monitor, temos:

```
local function set_internal(value)
   -- Faz alguma coisa
end
-- Cria o monitor
local mnt = monitor.create()
set = monitor.doWhenFree(mnt, set_internal)
```

A figura 3.4 mostra a implementação da função `monitor.dowhenfree`. Esta função cria e retorna uma nova função que encapsula a recebida como parâmetro. Essa nova função utiliza o lock para garantir que a execução é em exclusão mútua em relação às demais funções no monitor. `monitor.doWhenFree` também lida com os parâmetros de entrada e os resultados. A função `pack` captura os resultados em uma tabela Lua que é armazenada na variável `rets`. Após liberar o lock, o resultado é desempacotado e retornado.

```
-- mnt: monitor criado para proteger a função
-- func: função que será executada em exclusão mútua
function doWhenFree(mnt, func)
   -- Recupera referência para a estrutura de lock
   local idx = mnt.idx
   -- 'from' aponta para o criador do monitor
   local take = rpc.sync(mnt.from, "monitor.take")
   local release = rpc.async(mnt.from, "monitor.release")
   function f(...)
      take(idx)
      -- Invoca a função original e captura os valores
      local rets = pack(func(...))
      release(idx)
      return unpack(rets)
   end
   return f
end
```

Figura 3.4: Implementação da função `monitor.doWhenFree`.

As funções `monitor.take` e `monitor.release` controlam a aquisição do lock da seguinte forma: `monitor.take` tenta adquirir o lock com o dado monitor. Se o lock está livre, essa função muda o seu valor e continua normalmente. Se o lock já foi adquirido, `monitor.take` coloca a co-rotina corrente na fila de espera do lock e a suspende. A função `monitor.release`, simetricamente, libera o lock do monitor. Ela verifica se existe alguma co-rotina na fila de entrada do monitor, e, se existir, reinicia a primeira co-rotina que está esperando. Caso contrário, `monitor.release` marca o lock como livre.

Esse mecanismo para exclusão mútua é diferente das propostas de linguagens mais clássicas, pois ela não provê um encapsulamento sintático direto da função protegida. Isso faz do monitor um mecanismo dinâmico, permitindo que funções sejam adicionadas ao monitor somente quando necessário. Essa idéia é possível através da criação dinâmica de funções, escopo léxico e funções como valores de primeira classe. Como mostrado na figura 3.4, uma nova função, `f`, é dinamicamente criada para englobar a função desprotegida `func`, passada como parâmetro para `monitor.doWhenFree`. O escopo léxico permite que `f` mantenha a referência para `func`, podendo assim invocá-la na seção protegida.

Em linguagens orientadas a objetos como C++ e Java, nós podemos alcançar um mecanismo similar instanciando um encapsulador para interceptar as chamadas e impor a entrada ordenada na seção crítica. No entanto, o fato de tratar funções como valores de primeira classe nos dá um controle mais fino comparado com a interceptação de objetos porque nós podemos redefinir somente uma função e deixar as demais intocadas. O encapsulador precisa manter a mesma interface do objeto encapsulado, adicionar código para proteger os métodos desejados e repassar todas as chamadas para o objeto original.

A estrutura do monitor e a implementação de `monitor.doWhenFree`, baseada em chamadas remotas, criam a possibilidade de termos um único monitor protegendo funções de diferentes processos, dando suporte exclusão mútua distribuída. Por exemplo, um processo poderia criar um monitor e adicionar funções a ele. Em seguida, o processo passaria o monitor para outro processo, o qual adiciona novas funções. Quando as funções monitoradas são invocadas, elas fazem chamadas remotas para adquirir o lock. No entanto, somente uma delas irá ter sucesso e as demais ficarão esperando na fila pela liberação do lock. A figura 3.5 ilustra o uso do monitor distribuído. Neste exemplo, vários processos distribuídos poderiam receber, na inicialização, chamadas para uma função como `init`, todos eles recebendo o mesmo monitor como argumento. Cada processo poderia então proteger, usando o monitor, funções que manipulam o estado compartilhado.

3.2.2
Restrições de Sincronização e Sincronizadores

Nesta seção, continuamos nossa discussão sobre como podemos integrar mecanismos de coordenação dentro do nosso modelo básico de comunicação. Para ilustrar a flexibilidade obtida com Lua, voltamos nossa atenção para a possibilidade de definir condições para a execução da função. Essas condições nos permitirão modelar a coordenação tanto intra como inter-processo. Entre as propostas existentes nessa direção, escolhemos reimplementar o mecanismo proposto por Frølund e Agha (Frølund, 1996; Agha et al., 1993), porque ele é um dos poucos que provê

```
local isOn = false

local function _off()
   -- Desliga se seu vizinho está ligado
   if neighbor_state() then
      isOn = false
   end
end

-- 'mnt' é um monitor e 'neighbor' é um processo vizinho
function init(mnt, neighbor)
   neighbor_state = rpc.sync(neighbor, "get_state")
   off = monitor.doWhenFree(mnt, _off)
end
```

Figura 3.5: Exemplo de um monitor distribuído.

suporte tanto para sincronização distribuída como para sincronização concorrente.

Frølund e Agha propõem *restrições de sincronização* para dar suporte à sincronização intra-objetos (Frølund, 1996; Agha et al., 1993). Assim como guardas (Riveill, 1995; Briot, 1999), a idéia é associar restrições ou expressões a uma função para determinar se sua execução é permitida em um certo estado. Esse tipo de mecanismo permite ao desenvolvedor separar a especificação de políticas de sincronização do algoritmo básico que manipula suas estruturas de dados, ao contrário dos monitores, nos quais a sincronização deve ser embutida dentro do algoritmo.

Como um exemplo, retirado de (Frølund, 1996), considere um objeto botão de rádio com um método `on` e `off` para ligar e desligar, respectivamente. Para garantir que esses métodos são invocados estritamente em alternância, o programador pode definir a variável de estado `isOn`, que indica se o botão está ligado. Restrições de sincronização podem ser definidas para desabilitar o método `on` quando `isOn` é verdadeiro, e desabilitar o método `off` quando ele é falso.

Para sincronização inter-objetos, Frølund propõe o uso de *sincronizadores*. Um sincronizador é um objeto separado que mantém restrições de integridade sobre um grupo de objetos. Ele mantém informações sobre o estado global do grupo e permite ou proíbe a execução de métodos de acordo com esse estado global.

Manter as regras em um ponto central, em vez de espalhá-las entre os processos, facilita a modificação e permite o uso dos sincronizadores em sobreposição. Considere novamente o exemplo dos botões de rádio. Além da restrição de integridade individual da chamada alternada, um conjunto de botões de radio deve satisfazer a restrição de que no máximo um botão está ativo por vez. Para essa situação, Frølund propõe a seguinte solução: um sincronizador mantém o estado global do grupo na variável `activated`, cujo valor é verdadeiro se algum botão de rádio no grupo está ligado. Uma cláusula de desativação no sincronizador indica que, para qualquer botão do conjunto, o método `on` está desabilitado se o valor dessa variável

é verdadeiro. Para garantir a consistência global do estado, sincronizadores também permitem a definição de *gatilhos*: código que é associado com a execução de um método em um membro do grupo controlado pelo sincronizador. No caso do nosso exemplo, um gatilho é associado com a execução do método `on`, alterando `activate` para verdadeiro, e com o método `off`, alterando para falso.

Provemos suporte a esses mecanismos através dos módulos `sc` (para as restrições de sincronização) e `synchronizer`. No caso do módulo `sc`, as restrições são verificadas localmente pelo estado interno do processo, enquanto `synchronizer` permite que processos se registrem como sincronizadores nos objeto remotos (ou processo, no nosso caso) que eles coordenam.

A figura 3.6 ilustra como um programa poderia empregar restrições de sincronização para garantir as propriedades no exemplo do botão de rádio. `sc.add_constraint` associa funções de guarda para as funções acessíveis via RPC — aquelas que não são definidas como locais. Ele recebe como argumento o nome da função a ser protegida e a função que implementa a verificação. Esta, por sua vez, recebe como argumento as informações da requisição e deve retornar verdadeiro se a função protegida pode ser executada e falso caso contrário.

```
local isOn = false
local function can_turn_on(request)
    return not isOn
end

local function can_turn_off(request)
    return isOn
end

function on()
    isOn = true
end

function off()
    isOn = false
end

sc.add_constraint("on", can_turn_on)
sc.add_constraint("off", can_turn_off)
```

Figura 3.6: Definição de restrições de sincronização.

Na figura 3.7 temos a criação de um sincronizador que coordena um conjunto de processos os quais representam botões de rádio, restringindo que no máximo um deles esteja ativo por vez. Quando um dos botões recebe uma requisição, ele contacta o sincronizador para verificar as restrições remotas, as quais permitem ou não que o botão execute a função de acordo com as informações do estado global.

```
local activated = false
local function can_turn_on(request)
   return not activated
end

local function trigger_on(request)
   activated = true
end

local function trigger_off(request)
   activated = false
end

-- Define as restrições e gatilhos para cada botão
for bt in pairs(buttons) do
   synchronizer.add_trigger(bt, "on", trigger_on)
   synchronizer.add_trigger(bt, "off", trigger_off)
   synchronizer.add_constraint(bt, "on", can_turn_on)
end
```

Figura 3.7: Um sincronizador que define restrições remotas e gatilhos para conjunto de botões.

Ambos `synchronizer.add_trigger` e `synchronizer.add_constraint` (respectivamente para os gatilhos e as restrições) recebem, como seus argumentos, o identificador do processo, o nome da função nesse processo e a função que será executada uma vez que o sincronizador é contactado. No caso dos gatilhos, as funções tipicamente atualizam o estado global, e para as restrições, elas devem retornar, respectivamente, verdadeiro ou falso para permitir ou proibir a execução da função protegida. Além disso, a função que será executada recebe, como parâmetro, informações sobre a função protegida dentro da variável `request`.

Para implementar restrições de sincronização e sincronizadores, fizemos um alteração no módulo `rpc` que permite definirmos um tratador padrão para todas as requisições. Assim, os módulos `sc` e `synchronizer` são construídos sobre o `rpc` instalando um novo tratador padrão. Esse tratador, em vez de despachar imediatamente a chamada remota, coloca a requisição em um fila de espera.

O tratador então processa a fila, verificando se cada requisição pode ser atendida de acordo com as regras de sincronização impostas. Uma requisição é executada somente se todas as restrições locais e remotas forem avaliadas como verdadeiras. No entanto, quando uma função requisitada é executada, precisamos reiniciar a avaliação da fila porque essa função pode ter modificado o estado interno ou global, fazendo com que alguma requisição se torne elegível para execução. A fila é processada até estar vazia ou restarem apenas requisições que estão impedidas de executar.

Implementamos primeiro o esquema como descrito em (Frølund, 1996), que se baseia na verificação e execução das requisições de forma seqüencial — uma

nova requisição é tratada somente depois da anterior terminar. No entanto, avaliar requisições remotas é caro porque o processo se comunica com o sincronizador e bloqueia até que a resposta chegue. A figura 3.8-a mostra um diagrama ilustrando esse esquema. Embora o sincronizador imponha restrições somente sobre a função `set`, o processo espera que a resposta do sincronizador chegue antes de executar a requisição `get`.

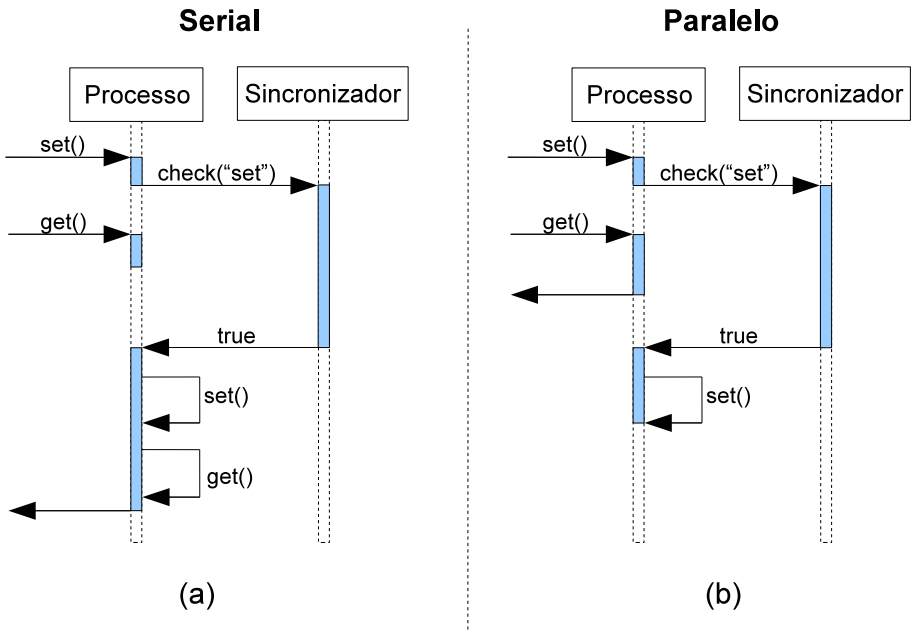

Figura 3.8: Diagramas da (a) proposta original e da (b) nossa proposta da avaliação das restrições.

Usando co-rotinas uma vez mais, implementamos um esquema alternativo para explorar mais a concorrência nesse sistema. Como sabemos que a verificação das restrições remotas irá bloquear o processo, encapsulamos essa verificação dentro de uma nova co-rotina, assim o processo pode suspendê-la enquanto o sincronizador analisa as restrições, e o processo está livre para tratar outra requisição. O novo esquema é mostrado na figura 3.8-b.

A figura 3.9 mostra o código do tratador implementado como alternativa ao modelo original de Frølund. O tratador varre a fila de mensagens em espera até que esta esteja vazia. A função `process_message` é responsável por verificar as restrições locais e do sincronizador. Se a requisição não for aceita, ela é posta em uma fila de pedidos bloqueados e uma nova requisição será avaliada. Caso ela seja aceita, ela é processada e os pedidos na fila de bloqueados devem ser avaliados novamente, pois o estado interno pode ter sido alterado. A variável `pending` controla a movimentação dos pedidos da fila de bloqueados de volta à fila de espera, procedimento realizado pela função `reload`.

As linhas de 11 a 13 criam e iniciam a execução da rotina `process_message`

```
1  function process_queue()
2    -- Move todas as requisições bloqueadas para a fila de espera.
3    reload()
4    while #waiting_queue > 0 do
5       local msg = waiting_queue[1]
6       if msg then
7          table.remove(waiting_queue, 1)
8
9          -- Cria um co-rotina para verificar as restrições
10         --  e processar a mensagem.
11         local co = coroutine.create(process_message)
12         processing_queue[co] = true
13         coroutine.resume(co, msg)
14
15         -- Se a mensagem foi executada, mover todas as mensagens
16         --  bloqueadas de volta para a fila de espera para nova
17         --  avaliação.
18         if pending then
19            reload()
20            pending = false
21         end
22      end
23   end
24 end
```

Figura 3.9: Tratador de fila alternativo que dá suporte ao sincronizador.

dentro de uma co-rotina. Essa rotina envia um pedido de verificação ao sincronizador e suspende, retornando o controle de volta ao tratador da fila. Este por sua vez pode iniciar a verificação de uma nova requisição. Quando o sincronizador responde, a verificação suspensa é então retomada.

Nossa implementação verifica as restrições de sincronização antes de contactar o sincronizador, evitando a comunicação pela rede se a verificação local falha. No entanto, como as chamadas de função agora podem modificar o estado interno durante a verificação das restrições remotas, as restrições de sincronização são verificas novamente após todos os sincronizadores terem respondido para garantir que o estado interno ainda permite a execução da requisição.

3.3 Avaliação de Desempenho

Nesta seção, discutiremos o desempenho dos mecanismos que descrevemos ao longo deste capítulo. Nosso objetivo é avaliar o custo mínimo que esses mecanismos adicionam ao modelo básico de RPC. Medimos o tempo médio para um cliente executar uma requisição com RPC assíncrono para uma função remota nos seguintes casos:

- **RPC:** esquema básico cliente/servidor que é usado como referência.
- **Monitor:** a função remota protegida por um monitor.

– **Tratador com Fila:** o mesmo esquema básico usado no caso do RPC, mas com um servidor usando uma infra-estrutura de fila para tratar as requisições do cliente. Essa infra-estrutura é a base para a implementação de restrições locais e remotas.
– **Restrições Locais:** a função remota é protegida por restrições de sincronização.
– **Restrições Remotas:** a função remota é protegida por um sincronizador.

Nos testes, os processos cliente, servidor e sincronizador executam em diferentes máquinas, com canais diretos entre o cliente e o servidor, e entre o servidor e o sincronizador. Além disso, implementamos funções remotas que não recebem qualquer parâmetro e que não retornam valores, e as restrições locais e remotas são funções que só retornam verdadeiro. No caso do monitor, não há chamadas concorrentes, dado que estamos interessados em medir o custo mínimo introduzido pela proteção.

A tabela 3.1 mostra o tempo médio (em milissegundos) para executar uma chamada remota assíncrona em cada um dos casos acima. Para efeito de comparação, mostramos o tempo médio para uma chamada Java RMI (usando Sun JDK 1.6), com os mesmos critérios de implementação das funções remotas. Realizamos os testes em máquinas equipadas com Pentium Core 2 Duo 2.6GHz, 1GB de RAM e Ethernet 100Mb/s, executando Linux (kernel 2.6.26).

Mecanismo	**Tempo**
Java RMI	0,612 ms
RPC	0,863 ms
Monitor	0,987 ms
Tratador com fila	0,972 ms
Restrições locais	0,974 ms
Restrições remotas	2,125 ms

Tabela 3.1: Tempos de execução dos diferentes mecanismos.

Por estarmos interessados em explorar o aspecto de construir diferentes abstrações, não nos dedicamos em obter uma implementação otimizada do RPC. Mesmo assim, quando comparado com RMI, o qual é construído como parte da arquitetura de Java, os tempos de execução do RPC parecem razoáveis.

O tratador de fila adiciona um pequeno custo para o RPC básico porque a requisição é posta na fila antes de ser executada. Por outro lado, uma restrição local quase não tem custo no nosso caso, já que ela é só uma chamada de função que retorna verdadeiro.

No teste de restrições remotas, o servidor deve contactar outro processo, o sincronizador, para executar a requisição. Além do tempo perdido na comunicação de

rede com o servidor, o sincronizador deve implementar um mecanismo que garante a visão consistente do estado global para evitar deadlocks. Assim, o sincronizador utiliza um esquema de transação com o servidor.

Realizamos um segundo experimento para medir o tempo para o servidor processar um conjunto de requisições de forma serial e paralela, como mostrado na figura 3.8 da seção 3.2.2. Nesse experimento, temos um servidor, um sincronizador e dois clientes. Todos esses processos executam em máquinas diferentes.

O servidor exporta duas funções, `get` e `set`, e recebe requisições dos dois clientes, um invocando somente a função `get` e outro somente a `set`. A função `set` é protegida pelo sincronizador, enquanto `get` não possui proteção.

Cada cliente realiza um total de 1.000 chamadas em seqüência e calcula o tempo médio de uma chamada. A figura 3.10 mostra o exemplo do código de um cliente. `get` é uma função remota assíncrona e `cb` é a callback responsável por receber o valor de retorno. O cliente para a função `set` é construído de forma semelhante.

```
max = 1000
count = 0

-- Recebe valor de retorno da chamada
function cb(val)
   count = count + 1
   if count < max then
      get()
   else
      -- Fim do teste, calcular o tempo
      t2 = clock.mark()
      local totaol = (t2 - t1) / 1E3
      print(total/max)
      alua.exit()
   end
end

-- Início do teste
function start()
   -- Cria a função remota para a 'get'
   get = crpc.async(server, "get", cb)
   -- Marca o tempo inicial
   t1 = clock.mark()
   get()
end
```

Figura 3.10: Código do cliente para a função `get`.

No lado do servidor, primeiramente, configuramos o mesmo para tratar as requisições de forma serial, isto é, o servidor processa a próxima requisição somente quando ele termina o processamento da atual. Alteramos então o comportamento do servidor para executar novas requisições enquanto o sincronizador avalia a restrição

remota. A tabela 3.2 mostra os tempos de cada chamada (em milissegundos) para cada cenário.

Requisição	**Processamento Serial**	**Processamento Paralelo**
`get`	1,217 ms	0,979 ms
`set`	2,204 ms	2,114 ms

Tabela 3.2: Processamento serial e paralelo das requisições.

No caso serial, a requisição para a função `get` é limitada pela avaliação da restrição remota, o que eleva o tempo de atendimento. No entanto, na configuração paralela, o servidor pode processar as requisições para `get` enquanto o sincronizador avalia as restrições remotas de `set`, assim, o tempo médio para as requisições `get` é menor.