PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 30/06

Sincronização em um Modelo RPC Orientado a Eventos

Bruno Oliveira Silvestre
Noemi de La Rocque Rodriguez
Jean-Pierre Briot

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 30/06 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Agosto, 2006

Sincronização em um Modelo RPC Orientado a Eventos

Bruno Oliveira Silvestre, Noemi de La Rocque Rodriguez, Jean-Pierre Briot[1]

[1] Laboratoire d'Informatique de Paris VI

{brunoos,noemi}@inf.puc-rio.br, jean-pierre.briot@lip6.fr

Abstract. The growth of the Internet has caused distributed programmers to face highly geographically distributed scenarios. The ALua system offers a base for developing event-based applications, using a callback-based programming model to process events. We developed a RPC library for ALua that offers the programmer a more friendly abstraction to create event-based applications. However, the RPC model introduces concurrent event processing, even if only in explicit program points. In this work we analyze several inter and intra-process synchronization mechanisms which have been proposed in the literatures and implement some of them on our RPC library.

Keywords: Event-based Programming, Synchronization Mecanisms, RPC, ALua

Resumo. Com a ampliação do uso da Internet, as aplicações distribuídas vêm atuando em cenários geográficos mais amplos. A programação orientada a evento tem sido apontada como um bom modelo de desenvolvimento para lidar com as característica desse ambiente altamente distribuído. O ALua provê uma base para o desenvolvimento de aplicações orientadas a eventos, fornecendo um modelo de programação via *callbacks* para o tratamento dos eventos. Desenvolvemos uma biblioteca RPC sobre o ALua que oferece uma abstração mais amigável de programação, mantendo a base assíncrona de comunicação. No entanto, a abstração de chamada síncrona que descrevemos introduz concorrência no tratamento dos eventos, ainda que em pontos bem localizados. Neste trabalho analisamos alguns mecanismos de sincronização intra e inter processos propostos na literatura, e implementamos alguns deles para o nosso modelo de RPC.

Palavras-chave: Programação Orientada a Eventos, Mecanismos de Sincronização, RPC, ALua

Responsável por publicações:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Sumário

1 Introdução

Com popularização da Internet nos últimos anos, aplicações distribuídas estão atuando em um cenário geograficamente amplo, envolvendo computadores de todas as partes do mundo. Devido às características desse novo ambiente, as chamadas síncronas, comumente usadas pelos *middlewares* para computação distribuída, não vêm se mostrando tão eficiente. A programação orientada a eventos tem sido apontada como uma boa alternativa nesses casos.

O ALua [Ururahy, Rodriguez e Ierusalimschy 2002, Rodriguez 2006] é um sistema que provê uma base para o desenvolvimento de aplicações orientadas a eventos. Mas, apesar dele fornecer um mecanismo flexível para o desenvolvimento de aplicações orientadas a eventos, a manipulação direta de mensagens dificulta o trabalho do desenvolvedor. Além disso, certas tarefas demandam diversas interações, o que eleva a uma divisão do processamento em vários passos de tratamentos de *callback*, tornando a programação menos intuitiva.

Usando como base o modelo apresentado em [Rossetto e Rodriguez 2005], desenvolvemos uma abstração de chamadas remotas de procedimento (RPC) sobre o ALua que fornecesse ao programador uma forma mais amigável, aos moldes da invocação tradicional de funções, para a criação das aplicações. No entanto, diferente do ALua que processa apenas um evento por vez, nosso modelo de RPC introduz a execução concorrente de eventos.

O propósito deste trabalho é estudar mecanimos de sincronização entre processos e obter um bom entendimento implementando-os sobre nossa base e ao mesmo tempo testá-la.

Na seção 2 apresentamos o ALua e suas característica. Na seção 3 mostramos como foi criado nosso mecanismo de RPC. Apresentamos na seção 4 o estudo de alguns mecanismos de sincronização tanto intra-processo, como inter-processos. Na seção 5 implementação de um mecanismo de monitor, sincronização via guardas e restrições remotas. Na seção 6 apresentamos as considerações finais sobre este trabalho.

2 ALua

ALua [Ururahy, Rodriguez e Ierusalimschy 2002, Rodriguez 2006] é um sistema desenvolvido na linguagem Lua [Ierusalimschy 2006] com o objetivo de fornecer uma infra-estrutura para a criação de aplicações distribuídas orientadas a eventos. As aplicações em ALua são compostas por diversos processos distribuídos pela rede que se comunicam através da troca assíncrona de mensagens, usando a primitiva `alua.send`.

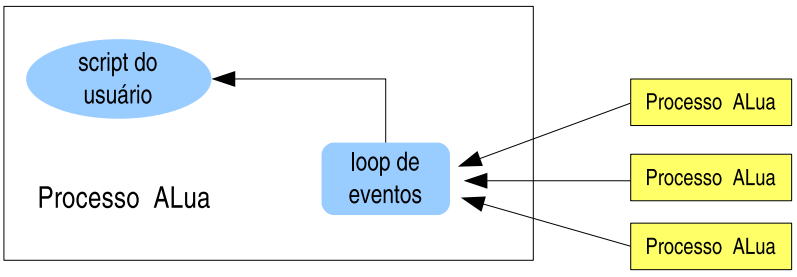

Figura 1: Recebimento de mensagens em um processo ALua.

Uma mensagem é um trecho de código em Lua que é executado no receptor. Como Lua é uma linguagem interpretada, as mensagem podem, além de disparar os eventos, alterar o com-

portamento da aplicação através da redefinição de funções. Além disso a aplicação não necessita invocar nenhuma função específica para o recebimento dessas mensagens. O ALua possui um *loop* de eventos que é reponsável por recebê-las e executá-las. Do ponto de vista da aplicação, ela só deve lidar com os eventos que estão sendo disparados pelas mensagens. A figura 1 mostra um esquema do *loop* de eventos na interação entre processos ALua.

Uma das características do ALua é não permitir o processamento concorrente de eventos, o que elimina problema de incosistência interna. Quando um evento é recebido pelo *loop* e tem sua execução iniciada, outro evento só será tratado quando o primeiro terminar por completo. Por isso, é importante que as mensagem não possuam trecho de código bloqueantes, o que evitaria o processamento de novas mensagens.

Cada uma das máquinas que faz parte de uma aplicação devem executar um *daemon* do ALua, possibilitando que os processos executem nessas máquinas. Na figura 2 é mostrada a interação entre os processos e *daemon* no ALua, distribuídos em várias máquinas pela rede. O *daemon* é responsável por dispachar as mesagens enviadas com `alua.send` entre os processos. Caso o processo esteja executando em outra máquina, a mensagem deve ser enviada ao *daemon* responsável por aquela máquina, para que ele então repasse a mensagem ao processo correspondente. O ALua fornece também um conjunto de funções que permite a abertura de canais diretos entre os processos, possibilitando uma comunicação livre da interação com o *daemon*.

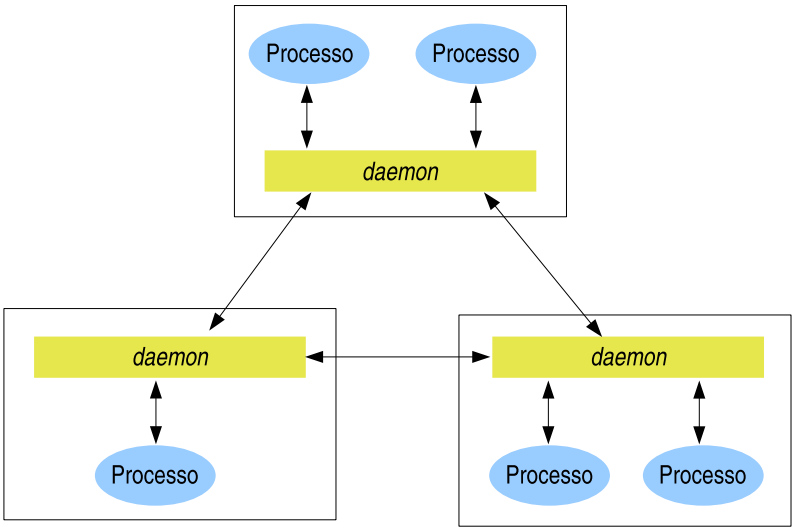

Figura 2: Interação entre processos e *daemons* no ALua.

O ALua disponibiliza um conjunto de funções para a manipulação de processos e *daemons*. Essas funções geralmente funcionam de forma assíncrona, assim, elas recebem um *callback* como parâmetro, a qual permite manipular o retorno da chamada e, por exemplo, realizar um tratamento de erro apropriado em caso de falha. Na figura 3 mostramos um programa simples usando o ALua. Criamos um *daemon* através da chamada `alua.open`, e então, iniciamos uma nova aplicação com `alua.start`. Esta função recebe uma *callback* que é executada após a criação da aplicação, a qual invoca `alua.spawn` para iniciar três novos processos. A *callback* `spawn_cb`, executada após a criação dos processos, envia um trecho de código para cada um dos novos processos, que imprime os identificadores dos mesmos. A primitiva `alua.loop` inicia o *loop* de eventos e deve ser invocada no fim do programa, pois ela fica em um laço infinito recebendo e processando as

mensagens.

```
function spawn_cb(reply)
   local cmd = "print(alua.id)"
   for p in pairs(reply.processes) do
      alua.send(p, cmd)
   end
end

function start_cb(reply)
   alua.spawn("ALuaSample", 3, spawn_cb)
end

alua.open()
alua.start("ALuaSample", start_cb)
alua.loop()
```

Figura 3: Exemplo de uso das funções de manipulação de processos e *daemons*.

3 RPC – Remote Procedure Call

O uso de chamada remota de procedimento (RPC) é uma abstração já bem difundida, que torna transparente para o desenvolvedor todo o processo de envio da solicitação de execução e a recuperação do retorno produzido. Entretanto, esse tipo de chamada é geralmente realizada de forma síncrona, levando ao bloqueio da aplicação até o recebimento da resposta. Em um cenário distribuído, principalmente em uma ampla distribuição geográfica, o tempo de resposta pode ser alto. Para determinados tipos de aplicações, isso significa uma perda significativa de desempenho. O uso de chamadas assíncronas nesses cenários deixa a aplicação liberada para a realização de outras tarefas, enquanto a resposta não esteja disponível.

Nesta seção apresentamos um RPC desenvolvido sobre o ALua que possibilita tanto o uso de chamadas síncronas bem como assíncronas. Fazemos uso das características da linguagem Lua que permite criar funções dinamicamente e manipulá-las como valor de primeira ordem, podendo armazená-las em variáveis ou passá-las como parâmetros para outras funções.

```
func = "foo"
process = "Bar"

function callback(v1, v2)
   print(v1 + v2)
end

f = rpc.async(func, process, callback)
f(arg1, ..., argn)
```

Figura 4: Criação de uma função de chamada remota assíncrona

A figura 4 mostra a criação de uma função `f` que realiza chamadas remotas assíncronas. A primitiva `rpc.async` recebe como parâmetro o nome da função remota, a identificação do processo ALua onde a função está definida e uma *callback* que será invocada quando o valor de retorno da

chamada estiver disponível. Como Lua suporta mais de um valor de retorno, a *callback* receberá como parâmetro todos os valores retornados pela função remota.

O uso de RPC assíncrono aumenta o nível de abstração sobre o uso direto da primitiva do ALua, mas a programação via *callback* recai novamente na divisão do processamento em várias etapas distintas, devido à forma como o resultado das chamadas remotas é obtido. `rpc.sync` cria funções que realizam chamadas síncronas a outros processos. Assim como o `rpc.async`, ela recebe como parâmetro o nome da função remota e o processo, mas por se tratar de invocação síncrona, o uso de função de *callback* não é necessário. A figura 5 apresenta um exemplo da criação e invocação síncrona de uma função.

```
func = "foo"
process = "Bar"

f = rpc.sync(func, process)
v1, v2 = f(arg1, ..., argn)

print(v1 + v2)
```

Figura 5: Criação de uma função chamada remota síncrona

Quando `f` é invocada, o processo ficará bloqueado até que os valores de retorno sejam recebidos, os quais serão armazenados em `v1` e `v2`. O bloqueio é feito suspendendo a linha de execução atual. Ao receber o retorno, é dada continuidade à execução. Isso é possível através o uso do mecanismo de co-rotinas de Lua, que, assim como *threads*, possuem a sua própria pilha de execução. No entanto, co-rotina emprega um mecanismo cooperativo de troca de contexto, ou seja, o desenvolvedor deve explicitamente efetuar chamadas para suspender a execução de uma co-rotina ou colocá-la novamente em processamento. Isso deixa claro onde o processo é interrompido, diferente da programação com *thread* que pode suspender a execução em qualquer ponto. Com os pontos de interrupção definidos, torna-se mais fácil manter a consistência interna pois tem-se a certeza de que não ocorrerá uma suspensão involuntária do processamento entre esses pontos.

A função retornada por `rpc.sync` utiliza `rpc.async` para realizar a invocação ao processo remoto e logo em seguida, suspende a linha de execução, bloqueando o processamento. A *callback* passada para o `rpc.async` fica incumbida de restaurar a execução assim que a resposta da chamada remota for recebida.

O modelo de RPC sobre Lua já havia sido proposto em [Rossetto e Rodriguez 2005], que além das chamadas síncronas e assíncronas, possibilitava a criação de objetos futuros [Lieberman 1987]. Nesta nossa implementação sobre o ALua, não contemplamos o uso de futuros, mas estendemos as chamadas remotas, permitindo a passagem de funções como parâmetros.

Como dito, em Lua as funções são tratadas como valores de primeira ordem, podendo ser armazenadas em variáveis e passadas como parâmetros. Para prover esse mesmo mecanismo, toda vez que uma função é passada como parâmetro para uma chamada remota, a implementação do RPC cria uma referência para a função e a envia como parâmetro da invocação. A figura 6 mostra um exemplo da passagem de uma função como parâmetro entre dois processos.

A parte RPC do receptor da chamada modifica a referência remota recebida, fazendo com que ela se comporte como uma função (usando os mecanismos de meta-métodos de Lua), ou seja, ela pode ser invocada da mesma forma que uma função local. No entanto, toda a invocação dispara uma chamada síncrona ao processo que possui a definição da função. No exemplo da figura 6, o processo `B` recebe a função `soma` como segundo parâmetro e realiza uma chamada, disparando

uma chamada remota de volta ao processo A.

```
-- Processo A --
function soma(v)
  return v + 1
end

f = rpc.sync("B", "bar")
f("Hello World", soma)
```

```
-- Processo B --
function bar(str, func)
  print(str)
  -- chamada remota para "A"
  v = func(123)
end
```

Figura 6: Exemplo da passagem de uma função como parâmetro.

A referência remota criada também pode ser passada como parâmetro para funções remotas. Isso não gera impacto no funcionamento do mecanismo – o receptor da referência continua endereçando o processo que define a função.

4 Mecanismos de Sincronização

O modelo de tratamento de eventos do ALua só permite que um novo evento seja tratado quando o anterior terminar. No entanto, a implementação do RPC realizada sobre o ALua introduz a possibilidade de tratamento concorrente de eventos. Se um evento bloquear em uma chamada síncrona, a linha de execução retorna para o ALua, abrindo a possibilidade de recebimento de novos eventos.

Se por um lado evitamos o bloqueio de execução enquanto não há resposta da chamada remota, por outro, criamos o problema da consistência interna não ser mantida - ainda que em pontos explícitos. É necessário que mecanismos de sincronização estejam disponíveis para que se possa realizar a coordenação na execução dos eventos.

4.1 Sincronização Intra-objeto

Nesta seção apresentamos três mecanismo clássicos utilizados para a sincronização do acesso concorrente dentro de um objeto: *enabled sets*, monitores e guardas.

4.1.1 Enabled Set

O mecanismo de *enabled set* define um conjunto de operações que podem ser executadas de acordo com o estado atual do objeto [Briot 1996, Frϕlund 1996, Tomlinson e Singh 1989]. A mudança de estado do objeto pode acarretar na mudança do conjunto de operações disponíveis.

Como exemplo, suponha uma fila de capacidade limitado que dispões das operações `get` e `put` para, respectivamente, a retirada e inserção de elementos. Quando a fila está vazia não é possível efetuar a retirada de um item, por outro lado, se a fila estiver cheia, não é possível inserir novos elementos. Em um estado intermediário, ou seja, com a fila nem completamente cheia, nem vazia, elementos podem ser retirados ou adicionados.

Pode-se notar três estados bem definidos para a fila: vazio, cheio e parcial. Assim, definimos para o estado vazio apenas a operação `put` como disponível; para o estado cheio, apenas `get`; e ambas as operações para o estado parcial.

Nas implementações encontradas em [Briot 1996] e [Tomlinson e Singh 1989], o mecanismo utilizado para invocação das operações é a passagem de mensagens. *Enabled sets* permitem que

apenas as mensagens que invocam as operações permitidas para o estado atual do objeto são processadas. As demais são postergadas até que o objeto mude para um estado onde o processamento das mesmas é permitido.

4.1.2 Monitor

O monitor é uma abstração que tenta reduzir o impacto no desenvolvimento, provendo um mecanismo de exclusão mútua entre as operações. Por definição, o monitor só permite que apenas uma das operações que ele controla esteja ativa por vez, não importando o estado do objeto [Andrews 200].

Apesar de semáforos e *locks* serem abstrações que permitem ao desenvolvedor definir com um grau menor a sincronização de acesso aos dados compartilhados, o uso desses mecanismo embutem no código da operação todo o mecanismo de sincronização. Com monitores a dificuldade no controle do acesso é minimizado, sendo boa parte do trabalho realizada de forma transparente.

Em certas ocasiões não é possível continuar a execução de uma operação devido ao estado interno do objeto. Seguindo o exemplo da fila, não é possível retirar um elemento se a fila estiver vazia. Para esses casos, a execução pode ser suspensa por meio de variáveis de condição.

Quando um processo sinaliza a espera em uma variável de condição dentro do monitor, ele é suspenso e o monitor fica livre para que outras operações sejam executadas. Vários processos podem ser bloqueados sobre uma mesma variável. Ao sinalizar uma variável variável de condição, um ou mais processos suspensos (dependendo do tipo de sinalização disponível) são postos de volta em execução, mas submetidos à coordenação do monitor, ou seja, apenas um entrará no monitor e os demais serão novamente bloqueados.

```
Monitor Fila {
   List elements;
   cond empty;

   procedure get() {
      while(isEmpty(elements)) wait(empty);
      e = first(elements);
      remove(elements, 1);
      return e
   }

   procedure put(e) {
      append(elements, e);
      signal(empty);
   }
}
```

Figura 7: Exemplo de uma fila com monitor.

Para exemplificar, seja o pseudo-código mostrado na figura 7, onde é declarada uma fila de capacidade infinita com duas operações, `get` e `put`. Devido ao monitor, as operações não podem ser executadas simultaneamente, garantindo a consistência interna. No entanto, a operação de `get` só é válida quando houver elementos a serem retornados. Caso contrário, a operação é suspensa (primitiva `wait`) esperando por um sinal na variável `empty`. Esse sinal ocorre toda vez que a operação `put` é executada (primitiva `signal`).

4.1.3 Guarda

Um guarda é uma expressão associada a um método de um objeto. Antes de invocar o método, a expressão é avaliada e, caso seu resultado seja verdadeiro, o método é então executado. Com isso, é possível evitar a execução de certas operações em um estado inadequado.

As expressões geralmente se referem a variáveis internas do objeto. Guide [Riveill 1995] e Actalk [Briot 2000], além do estado interno do objeto, permitem utilizar os parâmetros que estão sendo passados para o método na construção das expressões. Ambos também disponibilizam contadores por método que indicam, por exemplo, o número total de invocações, de invocações aceitas, de execuções completadas, em andamento e pendentes.

```
CLASS FixedSizeBuffer IMPLEMENTS ProducerConsumer IS
   CONST size = 100;
   buffer: ARRAY[0 .. size-1] OF Element;
   METHOD Put(IN e:Element);
      // Implementação do método Put
   METHOD Get(OUT e:Element);
      // Implementação do método Get
   CONTROL
      Put: (completed(Put) - completed(Get) < size) AND
           current(Put) = 0;
      Get: (completed(Get) < completed(Put)) AND
           current(Get) = 0;
END FixedSizeBuffer
```

Figura 8: Exemplo de um *buffer* com guarda.

A figura 8 mostra um trecho da implementação de um *buffer* de capacidade limitada no Guide, que utiliza os contadores para controle de sincronização. Os guardas são definidos na seção `CONTROL`. A execução do método `Put` só é permitida se a capacidade total do *buffer* não foi atingida (diferença entre o número de invocações completadas de `Put` e `Get` é menor que a capacidade) e nenhum processo esteja executando a operação. Para o método `Get`, a restrição é que haja elementos disponíveis (maior número de invocação de métodos `Get`) e nenhum processo esteja executando a operação.

4.1.4 Considerações

O que vale destacar desta seção é a forma como a sincronização é definida. Com o uso de *locks* e semáforos, o controle é mesclado à implementação das operações. Diferente ocorre nos mecanismos de *enabled set* e guardas. Os monitores ficam em um ponto intermediário, pois se ao mesmo tempo em que a exclusão mútua provida por definição libera o desenvolvedor do controle da sincronização, as variáveis de condição se espalham pela implementação.

4.2 Sincronização Inter-objetos

Nosso objetivo é estender os mecanismos de coordenação para o nível de interação entre os elementos distribuídos da aplicação. Da mesma forma que é necessário manter um estado consistente dentro de um processos, certas aplicações necessitam manter um estado global consistente, evitando que determinadas operações sejam executadas de acordo com o estado de alguns elementos.

4.2.1 Restrições Remotas

Em [Frϕlund 1996], Frϕlund descreve um sistema de objetos distribuídos com comunicação assíncrona. Como parte desse sistema, é definido um objeto sincronizador que tem por finalidade coordenar a interação de um grupo objetos, levando toda a responsabilidade da consistência global para um único elemento, em vez de espalhá-lo por vários objetos. Esse modelo facilita mudanças na coordenação global, pois apenas os sincronizadores necessitam de ser alterados, e simplifica o desenvolvimento, livrando os demais objetos da tarefa de cooperar explicitamente com a sincronização.

Um dos mecanismos adotados pelo sincronizador é a implantação de restrições remotas (*remote constraints*) nos objetos a serem coordenados. Restrições remotas funcionam como guardas que inibem o processamento dos eventos, mantendo as mensagens que disparam esses eventos em uma fila interna aos objetos, até que possam ser processadas. A principal diferença é que a avaliação dos guardas é feita remotamente, no sincronizador. Com isso, é possível compor regras de sincronização para vários objetos, como por exemplo, um recurso compartilhado. O sincronizador pode impedir o uso concorrente do recurso, postergando nos objetos o processamento dos eventos que usam esse recurso, de forma a ordenar o acesso. A figura 9 ilustra o funcionamento interno do objeto. A função `verify` consulta o sincronizador sobre as restrições externas, e se a resquisição passar pela avaliação ela é então executada, caso contrário ela permanece na fila.

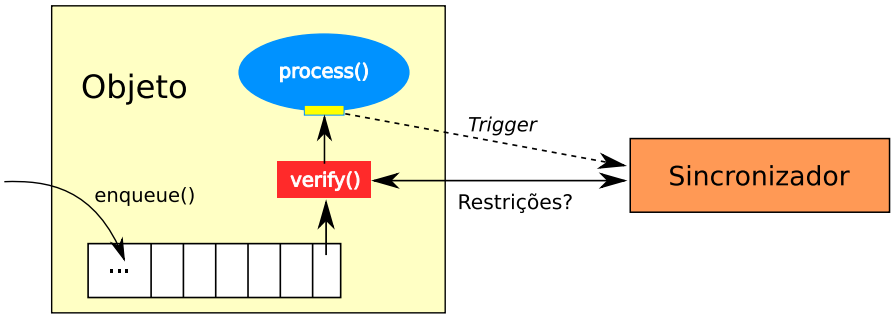

Figura 9: Processamento de restrições remotas.

O modelo de sincronização permite a instanciação de vários objetos sincronizadores, criando uma esquema de sobreposição, o qual dá origem a uma evolução por composição de funcionalidades. Cada sincronizador coordenaria uma determinada característica, fazendo com que um evento em um objeto seja processado somente se o mesmo passar pelas restrições de todos os sincronizadores que atuam sobre o objeto.

Ao instanciar um sincronizador, este entra em contato com os devidos objetos a serem sincronizados e informa quais eventos possuem restrições. Antes de processar um evento, o objeto verifica se há alguma restrição a ser verifica para o evento. Caso haja, o objeto consulta o sincronizador, que de acordo com as suas informações sobre o estado global do sistema e as regras de coordenação, responde positivamente ou não a execução do evento.

Como exemplo, o sincronizador mostrado na figura 10 controla a ativação de um recurso compartilhado (evento `on` em um objeto). Os objetos `Foo` e `Bar` passados como parâmetro só podem ativar o recurso se o mesmo estiver desativado. As regras são definidas na seção `disable`, a qual bloqueia o processamento do evento se a expressão após o `if` for avaliada como verdadeira.

Por esse exemplo também podemos notar a seção `trigger`, usada para manter o conhecimento

```
synchonizer sharedResource(Foo, Bar)
   activated := false;

   trigger
     Foo.on -> { activated := true };
     Bar.on -> { activated := true };
     Foo.off -> { activated := false };
     Bar.off -> { activated := false };

   disable
     Foo.on if activated;
     Bar.on if activated;
end sharedResource;
```

Figura 10: Exemplo de um sincronizador.

do estado global atualizado. Um *trigger* define um observador de eventos no objeto, ou seja, toda vez que um dos eventos descritos nesta seção (parte antes da seta) for executado, o sincronizador é notificado (figura 9) e a expressão contida entre as chaves é processada. Na figura 10, o sincronizador é notificado toda vez que o recurso é ativado ou desativado (`on` e `off`) por `Foo` ou `Bar`, fazendo com que sincronizador atualize suas informações sobre o estado do sistema.

4.2.2 Eventos Atômicos

As restrições remotas lidam com os eventos de forma isolada, não descrevendo nenhuma relação entre eles. Em certos casos, para manter o estado do sistema consistente, a execução de um evento em um objeto deve ocorrer em conjunto com o processamento de eventos em outros objetos. Por isso Frϕlund estende a coordenação entre objetos e introduz mais um mecanismo de controle que possibilita definir quais eventos devem ser processados atomicamente no sistema. No modelo proposto em [Frϕlund 1996], as regras são definidas no sincronizador, que informa os objetos sobre a necessidade de execução atômica. Entretanto, todo o processo de verificação e controle é de responsabilidade dos próprios objetos envolvidos.

Uma restrição imposta à definição das regras de atomicidade é que apenas eventos em objetos distintos podem ser coordenados, ou seja, a ocorrência de dois eventos ou mais eventos atomicamente em um mesmo objeto não é suportada. Isso endereça o fato de que se fosse possível esse tipo de execução, um evento poderia causar a mudança do estado interno, de forma a invalidar o processamento dos eventos seguintes. Mas, como os eventos já foram avaliados e aceitos pela regra de atomicidade, os demais eventos seriam processados, tornando o objeto inconsistente.

Uma solução para esse caso era detectar tal situação e desfazer as ações dos eventos, retornando o objeto a um estado novamente válido. Essa noção de atomicidade e recuperação de estados é bem conhecida na área de banco de dados, na forma de controle de transação. Ela se aplica bem a sistemas que se baseiam em operações para a mudança de estados. Entretanto, quando as operações desencadeiam a execução de uma ação, como por exemplo, a movimentação de um braço de robô ou a criação de uma série de novos eventos, desfazer tais operações pode se tornar uma tarefa impossível ou requerer uma complexidade com alta penalidade na eficiência.

Como exemplo da definição de atomicidade em eventos, seja o problema clássico do jantar dos filósofos. Pela definição, um filósofo só pode comer se o mesmo obter dois garfos (lembrando que existem *n* filósofos e *n* garfos). A figura 11 mostra um trecho do sincronizador que coordena cada

um dos filósofos. A regras de atomicidade previne o caso em que cada filósofo obtenha apenas um garfo, gerando uma situação de *deadlock*; um filósofo deve obter os dois garfos de uma única vez (atomicamente).

```
synchonizer indivisiblePickup(fork1, fork2, phil)
   atomic
      (fork1.pickup(p1) if p1 = phil,
       fork2.pickup(p2) if p2 = phil)
end indivisiblePickup;
```

Figura 11: Exemplo de um sincronizador para o jantar dos filósofos.

5 Implementação

Como parte do estudo da abstrações, implementamos alguns mecanismos de sincronização. As implementações também visam analisar o modelo de RPC descrito na seção 3 como base para o desenvolvimento de modelos com suporte à coordenação de processos.

5.1 Monitor

Nossa implementação do monitor sobre o RPC se baseia no uso de chamadas síncronas para requisição e liberação de um *lock*, o qual ordena a execução das chamadas remotas. A implementação também faz uso do fato de que as funções são executadas remotamente dentro de co-rotinas, permitindo o bloqueio dos processos até que o *lock* esteja disponível. Esse esquema foi baseado no monitor do ORFEO [Lima 2001], o qual emprega continuações no lugar co-rotinas.

O *lock* é uma tabela Lua, criado via `lock.create`, que guarda informações sobre o identificador do processo onde ele foi criado e um índice para uma tabela nesse mesmo processo, a qual mantém a informação se o *lock* está em uso ou não e fila de processos esperando a liberação, ou seja, o *lock* na realidade é um apontador para a uma estrutura mais complexa.

```
-- lck: lock criado para proteger as funções
-- func: função a ser executada em exclusão mútua
function doWhenFree(lck, func)
   -- índice do lock na tabela de locks
   local idx = lck.idx
   -- 'from' é o identificador do processo detentor do lock
   local take = rpc.sync(lck.from, "lock.take")
   local release = rpc.sync(lck.from, "lock.release")
   return function (...)
      take(idx)
      -- invoca a função e captura os valores de retorno
      local rets = pack(func(...))
      release(idx)
      return unpack(rets)
   end
end
```

Figura 12: Implementação da função `lock.doWhenFree`.

A criação do monitor utiliza a função `lock.doWhenFree`, que recebe como parâmetros um *lock* e a função que fará parte do monitor. O objetivo é usar um mesmo *lock* para proteger várias funções. O retorno de `lock.doWhenFree` é uma nova função que encapsula a função passada como parâmetro. Ao ser executada, a nova função retornada tenta adquirir o *lock* e, em caso de sucesso, passa o processamento para a função original. Quando a função original termina, todos os valores de retorno são capturados, para serem retornados logo após a liberação do *lock*. A figura 12 mostra a definição de `lock.doWhenFree`.

A implementação de `lock.take` e `lock.release` é mostrada na figura 13. Quando um processo solicita o *lock*, este é verificado para ver se está disponível. Caso esteja, seu estado é alterado para indicar que está em uso e a função termina normalmente, o que libera o processo chamador. Por outro lado, se o *lock* não estiver disponível, a co-rotina atual é colocada em uma fila de espera e então ela suspende a execução com uma chamada explícita ao método `coroutine.yield`. Com a suspensão da linha de execução da função `lock.take`, o processo chamador não obterá resposta, ficando bloqueado.

A liberação do *lock* se dá da seguinte forma parecida com a técnica de passagem de bastão [Andrews 200]. A função `lock.release` verifica a existência de alguma co-rotina suspensa à espera da liberação do *lock*. Neste caso, ela retira a co-rotina da fila e passa o controle para ela. A responsabilidade da co-rotina agora é simplesmente enviar uma resposta ao processo chamador, desbloqueando-o da chamada síncrona. Logo após a co-rotina enviar a resposta, o fluxo de execução retorna a `lock.release` que termina sua execução. Se a fila de espera estiver vazia, o *lock* é apenas sinalizado como livre.

```
function take(idx)
  -- busca o lock na tabela de locks
  local lck = locks[idx]
  if lck.busy then
     -- vai para a fila de espera e suspende
     local co = coroutine.running()
     table.insert(lck.waiting, co)
     coroutine.yield()
  end
  lck.busy = true
end

function release(idx)
  local lck = locks[idx]
  -- passa o controle para o próximo processo
  local co = lck.waiting[1]
  if co then
    table.remove(lck.waiting, 1)
    coroutine.resume(co)
  else
    lck.busy = false
  end
end
```

Figura 13: Funções `lock.take` e `lock.release`.

Na figura 14 temos um exemplo simples do uso desse mecanismo, que impede que as funções `foo` e `bar` sejam executadas concorrentemente – `_foo` e `_bar` não são visíveis externamente pois são declaradas como locais.

```
local function _foo(str)
   print("foo:", str)
end

local function _bar(str)
   print("bar:", str)
end

-- Cria um lock
local lck = lock.create()

foo = lock.doWhenFree(lck, _foo)
bar = lock.doWhenFree(lck, _bar)
```

Figura 14: Uso do *lock* para a criação de monitores.

Um outro fato que vale destacar sobre o *lock* é que a sua atual estrutura permite que ele seja compartilhado com outros processos, possibilitando a criação de um monitor distribuído. Esse é o efeito causado pelo uso de chamadas remotas na `lock.doWhenFree` para obter e liberar o *lock*. A figura 15 mostra um exemplo com dois processos, `A` e `B`, usam um *lock* compartilhado para ordenar o acesso às funções internas. O processo `A` cria o *lock* e o utiliza na criação de `foo`, que é a versão sincronizada. `A` então invoca a função remota de inicialização de `B`, passando o *lock*; o processo `B` cria a função sincronizada `bar`. Qualquer chamada a `foo` ou `bar` origina invocações remotas ao processo `A` para avaliação do *lock*.

```
-- Processo A --
local function _foo(str)
  print("foo:", str)
end

local lck = lock.create()
-- Cria uma função protegida
foo = lock.doWhenFree(lck, _foo)
-- Compartilha o lock
initB = rpc.sync("B", "init")
initB(lck)
```

```
-- Processo B --
local function _bar(str)
  print("bar:", str)
end

-- Cria a função protegida
function init(lck)
   bar = lock.doWhenFree(lck, _bar)
end
```

Figura 15: Exemplo de um monitor distribuído.

5.2 C-RPC – Constrainted Remote Procedure Call

C-RPC estende o RPC da seção 3, adicionando três mecanismos de sincronização: guardas, restrições remotas e *triggers* – estes não geram restrições, mas ajudam a manter o conhecimento sobre estado global. O C-RPC foi desenvolvido como uma camada acima do RPC e não em substituição do mesmo.

Na implementação do RPC sobre o ALua, as solicitações de execução são atendidas assim que o sistema as recebe. Isso é devido ao processamento atômico de eventos do ALua. Para a implementação dos mecanismos de sincronização, o primeiro passo foi criar uma abstração de fila de solicitações, que é transparente ao desenvolvedor. O C-RPC disponibiliza as função `crpc.async` e `crpc.sync` que tem o funcionamento análogo às suas respectivas do RPC. A diferença é que

elas têm por objetivo enviar solicitações de execução para a fila do processo remoto.

Da mesma forma que `rpc.sync` foi construída usando `rpc.async` como base, `crpc.sync` usa `crpc.async`. Esta por sua vez, também emprega `rpc.async` como base para a comunicação. Na figura 16 mostramos as implementações das funções. Perceba que `crpc.async` invoca a função remota `crpc.enqueue` que insere a solicitação de execução em uma fila. A criação da mensagem em `crpc.async` foi retirada por motivo de simplicidade.

```
function enqueue(msg)
   table.insert(queue, msg)
   -- processa a fila
   process()
end

function async(dst, func, callback)
   local send = rpc.async(dst, "crpc.enqueue")
   return function(...)

      -- monta a mensagem 'msg' (omitido, por simplicidade)

      send(msg)
   end
end

function sync(proc, func)
   return function (...)
      local co = coroutine.running()
      -- callback para acordar a co-rotina atual quando
      -- a resposta estiver disponível
      local callback = function (...)
          coroutine.resume(co, ...)
      end

      async(proc, func, callback)(...)
      -- bloqueia antes do retorno
      return coroutine.yield()
   end
 end
```

Figura 16: Implementação de `crpc.sync` e `crpc.async`.

O C-RPC não possui uma *thread* ou co-rotina dedicada ao processamento da fila de solicitações. A função `crpc.enqueue`, após colocar a solicitação na fila, desvia o fluxo de execução para o processamento da fila, invocando `process`. Esta função percorre a fila, executando as solicitações que são aceitas pelos mecanismos de sincronização.

O processamento dessas requisições é feita de forma ininterrupta devido ao esquema de multitarefa cooperativa das co-rotinas. No entanto, se a execução de alguma função for suspensa por uma chamada remota, o processo pode receber novos pedidos de enfileiramento de requisições. Nesse caso, mantemos uma indicação interna de que a fila já está sendo processada. A função `process` verifica essa indicação está desativada antes de iniciar o processamento da fila.

Quando não houver mais requisições na fila ou elas estão sendo bloqueadas pelas regras de sincronização, o processamento da fila termina, sendo retomado somente se uma nova solicitação ou uma notificação de mudança do estado global é recebida.

No entanto, antes que uma requisição seja executada é necessário que ela passe por dois mecanismos de sincronização. O primeiro, implementado com guardas, analisa se o estado interno do processo é válido para a execução. O segundo verifica se o estado global permite, através dos processos sincronizadores e as restrições remotas. A seguir descrevemos como foram implementados esses dois mecanismos no C-RPC.

5.2.1 Guardas

No C-RPC, guardas são funções associada aos métodos e que, de acordo com o estado interno do processo e dos parâmetros da solicitação, avaliam se os métodos podem ou não serem executados. Os guardas são chamados no sistema como restrições locais (*local constraints*).

Antes de executar uma solicitação, a função de processamento da fila verifica se há alguma restrição associada com o método solicitado. As funções de guarda retornam verdadeiro para indicando que a solicitação passou pelas regras de validação impostas. Caso um guarda retorne falso, a solicitação é mantida na fila e outra passa a ser avaliada. As funções de guarda recebem como parâmetro a solicitação de execução, com isso, elas podem acessar os parâmetros que foram enviados para a execução do método.

A figura 17 mostra o exemplo de um *buffer* com capacidade para um elemento. A função `get` só pode ser invocada quando há um elemento a ser retornado. Da mesma forma, `put` não pode ser invocado quando o *buffer* estiver armazenando apenas um elemento. A função `crpc.add_local_constraint` vincula uma função de guarda a um método, ela recebe como parâmetro o nome do método a ser protegido e a função de avaliação. De acordo com o exemplo, `check_get` e `check_put` verificam o estado do *buffer* e impedem a execução de `get` e `put`, respectivamente, caso necessário.

```
local empty = true
local value = nil

local function check_get(msg)
  return not empty
end

local function check_put(msg)
  return empty
end

function put(v)
  value = v
  empty = false
end

function get()
  empty = true
  return value
end

crpc.add_local_constraint("put", check_put)
crpc.add_local_constraint("get", check_get)
```

Figura 17: Exemplo do uso de guardas no C-RPC.

O *buffer* pode ser usado na implementação do problema do produtor e consumidor. A figura 18 mostra o trecho de código do processo produtor, que coloca o valor de um contador no *buffer*, e do processo consumidor, que retira esse valor e o exibe.

```
-- Produtor --
put = crpc.sync("Buffer", "put")
while true do
   count = count + 1
   put(count)
end
```

```
-- Consumidor --
get = crpc.sync("Buffer", "get")
while true do
   value = get()
   print(value)
end
```

Figura 18: Exemplo do uso do C-RPC – produtor e consumidor.

Quando um método é executado, o processamento da fila deve ser reiniciado para que as solicitações sejam novamente avaliadas. Isso porque o método recém executado pode ter alterado o estado interno do processo, fazendo com que solicitações que antes não eram aceitas pelas restrições locais, possam ser executadas.

5.2.2 Restrições Remotas

Restrições remotas são implementadas usando a idéia de processos sincronizadores, cujo papel é coordenar a interação entre outros processos – nossa versão dos objetos sincronizadores da seção 4.2.1. Quando o sincronizador é criado, ele entra em contato com os processos que são afetados pelas regras de coordenação, informando que determinados métodos possuem restrições remotas a serem satisfeitas.

O processo coordenado não sabe quantas nem quais restrições estão sendo aplicadas, mas apenas quais sincronizadores estão atuando sobre um método. Quando o processo avalia uma das solicitações da fila e verifica que o método requisitado está sob restrições, ele envia a solicitação para cada um dos sincronizadores que atuam sobre o método, para que eles possam avaliá-la. A verificação é feita de acordo com as regras de coordenação descritas no sincronizador e o estado global do sistema. Enquanto isso, o processo fica bloqueado esperando resposta dos sincronizadores. Se novas requisições de execução forem enviadas para tal processo, elas são colocadas na fila de solicitações.

A avaliação pode resultar em três respostas: *ok*, *failed* ou *aborted*. Se o sincronizador retornar *ok*, isso significa que a solicitação não infringiu as regras impostas. No entanto, para que solicitação seja processada é necessário que todos os sincronizadores respondam *ok*. Caso um retorno seja *failed*, indicando que a solicitação não é válida, esta permanece na fila para uma avaliação posterior. *aborted* sinaliza que a avaliação foi cancelada pelo mecanismo de controle de *deadlock*. Neste caso, é preciso resubmeter a solicitação aos sincronizadores para que os mesmos façam uma nova avaliação, pois o recebimento de uma resposta *aborted* torna a verificação das restrições inconclusiva.

Como exemplo, seja um sincronizador cuja responsabilidade é coordenar um conjunto de recursos compartilhados. Esses recursos podem ser ativados (método `turn_on`) e desativados (método `turn_off`), mas apenas um recurso pode estar ativo por vez. A figura 19 mostra o código do sincronizador.

Da mesma forma que na implementação dos guardas, a verificação é feita através de funções que devem retornar verdadeiro, se a solicitação é aceita, ou falso, caso contrário. Se uma das função retornar falso, o solicitação é tida como inválida e uma resposta *failed* é retornada ao processo.

```
local on = false

local function check_on(id, msg)
  return not on
end

local function trigger_on(id, msg)
  on = true
end

local function trigger_off(id, msg)
  on = false
end

function init(resources)
  for _, rc in ipairs(resources) do
    synchronizer.set_trigger(rc, "turn_on", trigger_on)
    synchronizer.set_trigger(rc, "turn_off", trigger_off)
    synchronizer.add_constraint(rc, "turn_on", check_on)
  end
end
```

Figura 19: Exemplo de um sincronizador de recursos.

A função `synchronizer.add_constraint` associa uma restrição remota a um método. Ela recebe como parâmetro o identificador do processo que contém o método, o nome do método e a função de avaliação. O processo é então informado de que uma restrição foi instalada no método, e qualquer requisição de execução do mesmo deve ser avaliada remotamente. Podemos ver na função `init` mostrada na figura 19, a definição de restrições remotas sobre o método `turn_on` em cada um dos recursos.

O C-RPC também disponibiliza o mecanismo de *trigger*, que pode ser usado para manter os sincronizadores informados do estado do global. Ao associar um *trigger* a um método, o sincronizador é informado toda vez que o método é executado. O *trigger* é implementado da seguinte forma: toda vez que o sincronizador recebe uma solicitação para ser avaliada, ele a armazena até que todos os sincronizadores avaliem a solicitação. Se a verificação for bem sucedida, ou seja, a solicitação foi aceita por todos os sincronizadores, o *trigger* correspondente é executado.

Entretanto, há situações em que um método não possui uma restrição associada, mas somente um *trigger*. Por exemplo, na figura 19, é definido um *trigger* na função `turn_off`, para que o sincronizador saiba que o recurso não está mais em uso. Usando o fato de que o processo coordenado não sabe quais restrições atuam sobre ele, mas apenas quais sincronizadores, ao associarmos um *trigger* a um método, o processo é notificado de que uma restrição remota foi instalada. Dessa forma, ele passa a consultar o sincronizador, mas como não há uma restrição para avaliar, a resposta *ok* é sempre retornada, o que não afeta a avaliação dos outros sincronizadores. E, ao se constatar que a verificação foi bem sucedida, o *trigger* é então disparado.

Controle de Transação

A garantia de que os sincronizadores saberão quando uma verificação foi bem sucedida é dada através do uso de transações. O fato de que uma função em um processo pode estar sendo coordenada por diversos sincronizadores, somado com o assincronismo da troca de mensagens, introduz

uma incerteza se a execução foi mesmo realizada. Por exemplo, um sincronizador pode avaliar como *ok* uma solicitação, mas outro pode retornar *failed*, evitando que a execução seja feita. Como os sincronizadores são elementos independentes, o primeiro deve ser notificado da falha.

Ao receber um pedido de verificação por parte de um processo, o sincronizador realiza a análise e envia o resultado de volta, ficando bloqueado até que o processo recolha as respostas dos demais sincronizadores. Caso todas as respostas sejam positivas, os sincronizadores recebem a notificação de sucesso e têm a certeza da execução da solicitação. Com isso, eles podem alterar as informações sobre o estado global – através da execução dos *triggers*. Se a avaliação falhar, os sincronizadores finalizam a transação sem nenhuma alteração adicional.

Devido a esse bloqueio do sincronizador, há a possibilidade de que o sistema entre em *deadlock*. Suponha os processos `A` e `B` sendo coordenados pelos sincronizadores S_1 e S_2, simultaneamente. Se `A` envia um pedido de avaliação para S_1 ao mesmo tempo que `B` envia um pedido para S_2, os sincronizadores ficarão bloqueados. Então, `A` tenta entrar em contato com S_2 e `B` com S_1. Como os sincronizadores estão bloqueados, nenhuma resposta é obtida e o sistema pára.

Seguindo o que foi proposto por Frϕlund em [Frϕlund 1996], utilizamos um identificador para as transações, que serve como um critério de desempate. Na implementação atual, esse identificador é baseado em um contador de transações já efetuadas pelo processo requisitante e o identificador do mesmo. Processos com contadores mais baixos possuem prioridade, e no caso dos contadores serem iguais, o identificador do processo é comparado. Esse modelo tenta igualar as chances de sucesso nas avaliações, evitando *starvation* de algum processo.

Quando um sincronizador bloqueado recebe um novo pedido de avaliação, ele confronta o identificador da transação atual com o do novo pedido. Caso a prioridade seja da transação atual, o sincronizador envia *aborted* como resposta ao pedido. Como descrito anteriormente, uma resposta *aborted* faz com que o processo tenha que reiniciar a avaliação. Se o novo pedido possuir a prioridade, ele irá para uma fila e será processado posteriormente, não interrompendo o andamento da transação atual.

A garantia do funcionamento desse esquema é que ou o pedido de maior prioridade foi para a fila de espera em todos os sincronizadores, caracterizando que o pedido de menor prioridade atingiu os sincronizadores primeiro; ou há um sincronizador onde o pedido de maior prioridade foi aceito, fazendo com que o de menor prioridade cancele todas as transações, liberando os sincronizadores. É importante destacar esse esquema visa evitar apenas a ocorrência de *deadlock*: o sistema pode se comportar de maneira diferente de acordo com a ordem em que as restrições são avaliadas, mas não depende dela para funcionar.

5.2.3 Composição de Sincronizadores

A instanciação de diferentes processos sincronizadores nos permite dividir a tarefa de coordenação em elementos menores, e usá-los como componentes que se sobrepõem para garantir a consistência global. O objetivo é trabalhar com sincronizadores menores, na tentativa de reduzir o esforço de manutenção, e poder reusá-los em diversos cenários.

Para mostrar um exemplo da composição, voltemos ao sincronizador da figura 19, que controla a ativação de um recurso compartilhado por vez. Suponha agora que uma nova restrição é imposta ao sistema, fazendo com que um recurso só possa ser ativado uma vez dentro de um passo de processamento do sistema. A marcação dos passos do sistema é dada pela ativação de todos os recursos (não simultaneamente), ou seja, um novo passo é iniciado assim que todos os recursos tenham sido ativados uma vez no passo atual. O código do novo sincronizador é mostrado no figura 20.

```
local done = 0
local nRes = 0
local current = 0
local step = {}

local function check_on(rc, msg)
  return step[rc] == current
end

local function trigger_on(rc, msg)
  -- inverte o indicador do passo: 1->0 ou 0->1
  step[rc] = (step[rc] == 0) and 1 or 0
  done = done + 1
  -- novo passo: todos os recursos foram ativados
  if done == nRes then
     done = 0
     -- inverte o indicador do passo: 1->0 ou 0->1
     current = (current == 0) and 1 or 0
  end
end

function init(resources)
  for _, rc in ipairs(resources) do
    synchronizer.set_trigger(rc, "turn_on", trigger_on)
    synchronizer.add_constraint(rc, "turn_on", check_on)
    nRes = nRes + 1
    step[rc] = 0
  end
end
```

Figura 20: Exemplo da composição de sincronizadores.

A figura 21 mostra o código do processo que inicializa os recursos e os sincronizadores das figuras 19 e 20. Uma nova aplicação é iniciada com `alua.start`, que passa uma *callback* (`start_cb`), a qual cria os procesos dos sincronizadores e dos controladores dos recursos – usando a primitiva `alua.spawn`. A função de *callback* `spawn_cb` envia trechos de código Lua para os processos, fazendo com que eles carreguem os seus arquivos de código correspodentes. Então, é feita uma chamada RPC para si próprio, invocando a função `start`, para que seja feita a inicialização dos componentes, informando aos sincronizadores quais os recursos que eles devem coordenar. Essa chamada a `start` é necessária pois a função retornada por `rpc.sync` só pode ser invocada dentro de uma chamada RPC, devido ao uso de co-rotinas para o bloqueio. Por isso, uma aplicação usando RPC geralmente inicia com uma chamada via `rpc.async`.

Por simplicidade, os passos são marcados com dois valores (0 e 1), indicando o passo atual e o próximo. O novo sincronizador instala uma restrição remota nos recursos para garantir que os mesmo só serão ativados uma vez a cada passo. Para saber se o recurso já foi ativado, o sincronizador instala um *trigger* sobre a função `turn_on` que marca em que passo o recurso foi ativado. Após todos os recursos serem ativados dentro do mesmo passo, o sincronizador passa para o passo seguinte.

```
local resources = {"Res_A", "Res_B", "Res_C", "Res_D", "Res_E"}
local procs = {"Res_A", "Res_B", "Res_C", "Res_D",
               "Res_E", "Sync_A", "Sync_B"}

function start()
  -- Inicializa os sincronizadores
  local init = rpc.sync("Sync_A", "init")
  init(resources)

  init = rpc.sync("Sync_B", "init")
  init(resources)

  -- outras inicializações
end

function spawn_cb(reply)
  -- carrega o código dos controladores dos recursos
  for _, r in ipairs(resources) do
    alua.send(r, "dofile('resource.lua')")
  end

  -- carrega o código dos sincronizadores
  alua.send("Sync_A", "dofile('sync_on_off.lua')")
  alua.send("Sync_B", "dofile('sync_round.lua')")

  -- invoca a função start em si próprio via rpc.async,
  -- requisito para uso do rpc.sync
  local start = rpc.async(alua.id, "start")
  start()
end

function start_cb(reply)
  alua.spawn("SyncTest", procs, spawn_cb)
end

alua.open()
alua.start("SyncTest", start_cb)
alua.loop()
```

Figura 21: Criação dos recursos e dos sincronizadores.

6 Conclusão

Neste trabalho, realizamos a implementação de um modelo de chamadas remotas de procedimentos sobre o ALua, como forma de simplificar o desenvolvimento de aplicações orientadas a eventos sobre o ALua. A camada RPC encapsula o envio dos eventos e o tratamento do retorno, fornecendo uma abstração mais fácil baseada em chamadas de funções. Com isso o desenvolvedor não necessita dividir suas tarefas em vários tratadores de eventos – essa divisão geralmente torna mais difícil o entendimento e a criação dos programas.

No entanto, dessa implementação surge a demanda por mecanismos de sincronização, pois os eventos que antes eram processados ininterruptamente no ALua, passaram a ser executados concorrentemente. Mesmo com o uso de co-rotinas, o que leva a uma concorrência cooperativa de pontos de troca de contexto bem conhecidos, não há como garantir a não interferência entre os

eventos.

Os nossos estudos realizados sobre alguns mecanismos de sincronização intra-objetos mostram duas abordagem, do ponto de vista do desenvolvedor, na descrição das regras de acesso. Na primeira abordagem, podemos considerar a introdução da lógica de sincronização dentro da escrita da própria função, mesclando funcionalidades com controle de acesso. Na segunda, essa lógica é transferida para elementos separadas, que apesar de necessitarem de acesso ao estado interno do processo, separam o que é funcionalidade do que é coordenação de acesso. Uma linha de investigação poderia ser traçada para determinar um comparativo de expressividade no controle do sincronismo entre essas duas abordagens.

Os guardas e os monitores foram os dois mecanismo implementados para a manutenção do estado interno dos processos. O monitor atende a requisitos mais simples de acesso simultâneo, permitindo que a execução dos eventos dentro do monitor seja de maneira exclusiva. No entanto, nossa implementação não possibilita o uso de variáveis de condição que suspendem o processamento e liberam o monitor.

Com os guardas é possível elaborar verificações mais elaboradas, considerando o estado interno do processo, o método invocado e até os parâmetros. Entretanto, os guardas controlam o sincronismo antes da execução da solicitação, e não podem ser usado no caso de que a decisão do acesso é tomada dentro da função, após um pré-processamento.

O uso dos processos sincronizadores possibilita que as regras de coordenação da interação entre os processos da apliação sejam retirados desses processos e levadas a um elemento separada. Além de facilitar o desenvolvimento dos processos, os sincronizadores podem ser compostos e até reutilizados em diferentes aplicações, afetando de maneira positiva a evolução dos sistemas.

Os próximos passos deste trabalho é analisar formas de implementar o mecanismo de execução atômica de eventos sobre o C-RPC, e a realizar testes de desempenho que possam mostrar como esses mecanismos de sincronização estão impactando no tempo de processamento dos eventos.

Referências

[Andrews 200]ANDREWS, G. R. *Foundation of Multithreaded, Parallel, and Distributed Programming*. [S.l.]: Addison Wesley, 200.

[Briot 1996]BRIOT, J.-P. Experience in classification and reuse of synchronization schemes. In: FUTATSUGI, K.; MATSUOKA, S. (Ed.). *Object Technologies for Advanced Software (ISOTAS'96)*. Kanazawa, Japan: Springer-Verlag, 1996. (LNCS, 1049), p. 227–249.

[Briot 2000]BRIOT, J.-P. Actalk: A framework for object-oriented concurrent programming - design and experience. In: BAHSOUN, J.-P. et al. (Ed.). *Object-Oriented Parallel and Distributed Programming*. [S.l.]: Hermès Science Publications, Paris, France, 2000. p. 209–231.

[Frϕlund 1996]FRϕLUND, S. *Coordinationg Distributed Objects: An Actor-Based Approach to Synchronization*. [S.l.]: The MIT Press, 1996.

[Ierusalimschy 2006]IERUSALIMSCHY, R. *Programming in Lua*. 2. ed. [S.l.]: Lua.org, 2006.

[Lieberman 1987]LIEBERMAN, H. Concurrent object oriented programming in act 1. In: YONEZAWA, A.; TOKORO, M. (Ed.). *Object Oriented Concurrent Programming*. [S.l.]: MIT Press, 1987.

[Lima 2001]LIMA, M. *ORFEO: Programação Distribuída Orientada a Eventos com Funções e Continuações como Valores de Primeira Classe*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, Julho 2001.

[Riveill 1995]RIVEILL, M. Synchronising shared objects. *Distributed Systems Engineering Journal*, v. 2, n. 2, p. 112–125, June 1995.

[Rodriguez 2006]RODRIGUEZ, N. *ALua: asynchronous distributed programming in Lua*. 2006. http://alua.inf.puc-rio.br.

[Rossetto e Rodriguez 2005]ROSSETTO, S.; RODRIGUEZ, N. Integrating remote invocations with asynchronism and cooperative multitasking. In: *Third International Workshop on High-level Parallel Programming and Applications*. [S.l.: s.n.], 2005.

[Tomlinson e Singh 1989]TOMLINSON, C.; SINGH, V. Inheritance and synchronization with enabled-sets. In: *Object-oriented programming systems, languages and applications*. New York, NY, USA: ACM Press, 1989. p. 103–112. ISBN 0-89791-333-7.

[Ururahy, Rodriguez e Ierusalimschy 2002]URURAHY, C.; RODRIGUEZ, N.; IERUSALIMSCHY, R. Alua: Flexibility for parallel programming. *Computer Languages*, v. 28, n. 2, p. 155–180, December 2002.