Abstrações para o desenvolvimento de aplicações distribuídas em ambientes com mobilidade

Silvana Rossetto , Noemi Rodriguez , Roberto Ierusalimschy

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

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

Abstract. *In this paper, we discuss adequate abstractions and mechanisms for developing distributed applications in mobile environments. We argue that coordination and event orientation are important technologies for these environments. To show that these technologies can be used in conjunction with well-known, conventional programming models, we describe a system with synchronous and asynchronous remote procedure call that we implemented over an event-oriented tuple space. To implement synchronous calls in an asynchonous environment, we used the coroutines provided by the Lua programming language.*

Resumo. *Neste artigo investigamos mecanismos e abstrações de programação adequadas para o desenvolvimento de aplicações distribuídas em ambientes com mobilidade. Exploramos o uso da programação orientada a eventos e da noção de coordenação por se adaptarem bem aos requisitos desses ambientes. Para prover uma abstração de programação mais convencional, construímos em cima de um espaço de tuplas orientado a eventos uma interface de chamada de procedimentos remotos que dá suporte a chamadas síncronas e assíncronas. Para implementar chamadas síncronas utilizamos o mecanismo de corrotinas da linguagem Lua.*

1. Introdução

O avanço da computação móvel tem permitido que dispositivos físicos com diferentes características e finalidades realizem alguma computação e estabeleçam conexão com outros dispositivos, de forma independente ou parcialmente independente de uma estrutura física. Também é possível mover uma computação de um dispositivo físico para outro com o objetivo de atender a uma necessidade de continuidade ou desempenho na execução de uma aplicação. Associado a essas novas possibilidades os ambientes com mobilidade trazem novos desafios para o desenvolvimento de aplicações distribuídas, entre os quais, a necessidade de lidar com a intermitência das conexões dos dispositivos físicos e a consequente exigência de adaptação às mudanças no contexto.

Nesse cenário o desacoplamento é uma característica importante. A adoção de princípios arquiteturais que garantem a separação entre computação e comunicação facilitam o tratamento dos desafios trazidos pela computação móvel. Essa separação permite

a substituição, atualização ou monitoração dos componentes sem afetar a lógica que determina como devem interagir. Por outro lado, também é possível modificar essa lógica de interação sem afetar a computação interna de cada componente. Da mesma forma um componente individual que migra de um local para outro pode precisar adaptar-se ao novo contexto alterando a sua computação interna sem afetar a sua interface de comunicação.

Os conceitos de coordenação e de programação orientada a eventos favorecem o desacoplamento. A coordenação enfatiza uma disciplina de programação que privilegia o desacoplamento buscando uma separação clara entre computação e comunicação para permitir que as aplicações possam ser compostas e recompostas a partir de componentes distintos. O conceito de coordenação é historicamente ligado ao modelo de espaço de tuplas [Gelernter and Carriero, 1992], o qual possui ainda outras características que se ajustam bem a ambientes com mobilidade. A orientação a eventos, por outro lado, é um paradigma de programação que favorece o desacoplamento entre as partes de uma computação. Nele a programação de baseia na reação à ocorrência de eventos que podem ser emitidos e recebidos sem que as partes da comunicação precisem ter consciência da presença umas das outras.

No entanto, embora o assincronismo da programação orientada a eventos seja extremamente conveniente para os ambientes que estamos considerando, o modelo de programação associado a ele nem sempre é fácil de assimilar pelos programadores. Uma aplicação puramente orientada a eventos assume um pouco a forma de uma máquina de estados — como discutido em [Ururahy et al., 2002] — onde os tratadores de eventos são responsáveis por transições que dependem do evento que chegou e do estado anterior da aplicação. Por isso consideramos importante oferecer ao programador alguma abstração que facilite a tarefa de programação.

Nesse trabalho exploramos os conceitos de orientação a eventos e coordenação com espaços de tuplas como base para um ambiente de desenvolvimento de aplicações distribuídas que leve em conta as características de contextos com mobilidade. Fazemos uso da linguagem Lua [Ierusalimschy et al., 1996] como linguagem de configuração que além de permitir especificar a interação entre os componentes também permite alterá-la dinamicamente — como foi explorado por [Batista and Rodriguez, 2002] na reconfiguração dinâmica de aplicações baseadas em componentes. Para facilitar a tarefa de programação investigamos a construção de um modelo de programação convencional, o de chamada remota de procedimentos, sobre essa plataforma de espaço de tuplas e orientação a eventos.

O restante do texto está organizado da seguinte forma: na seção 2 mostramos as características da programação orientada a eventos e da tecnologia de coordenação e justificamos porque constituem bons paradigmas para o desenvolvimento de aplicações em ambientes com mobilidade. Na seção 3 descrevemos o LuaTS, uma implementação assíncrona de espaços de tuplas desenvolvida para Lua. Na seção 4 propomos um ambiente para o desenvolvimento de aplicações distribuídas implementado sobre o espaço de tuplas. Esse ambiente define uma camada de coordenação que usa chamada remota de procedimento assíncrona e síncrona. Por fim, na seção 5 mostramos um exemplo de aplicação desenvolvida nesse ambiente e na seção 6 avaliamos os resultados desse trabalho.

2. Orientação a eventos e coordenação: boas abstrações para mobilidade

A abordagem baseada em eventos é adequada para a computação distribuída que precisa interagir assincronamente e preservar a anonimidade das partes envolvidas. A coordenação, por outro lado, privilegia a desvinculação entre a computação interna de um componente e sua interação com outros componentes, oferecendo uma característica de flexibilidade desejável em ambientes com conexão intermitente. Nessa seção destacamos as características dessas duas abordagens e apontamos algumas das vantangens de usá-las para o desenvolvimento de aplicações em ambientes com mobilidade.

2.1. Orientação a eventos

Como caracterizado por [Carzaniga et al., 2001], na arquitetura baseada em eventos os componentes de software distribuídos interagem gerando e consumindo eventos. Um evento é publicado para o mundo externo por meio de uma mensagem, sem informação sobre o receptor. De forma geral, quem gera um evento o envia para um componente encarregado de distribuí-lo para todos os demais componentes interessados em recebê-lo. Esse componente é quem faz o desacoplamento entre os emissores e os receptores de eventos. [Fiege et al., 2002] destacam o uso de sistemas baseados em eventos como modelos de coordenação para integrar componentes fracamente acoplados.

A troca de mensagens é o mecanismo básico de comunicação e é feita de forma assíncrona, de modo que uma aplicação orientada a eventos executa ações em decorrência da chegada de mensagens. Assim, o termo computação distribuída orientada a eventos refere-se ao estilo de execução reativa de processos que se baseia no envio e recebimento assíncrono de eventos. Esse estilo de programação favorece o assincronismo e o não-determinismo dos acontecimentos além de permitir que as partes de uma comunicação não precisem ter consciência da existência umas das outras e de estabelecer uma conexão direta entre si para se comunicarem. Como consequência é possível associar e desassociar componentes em uma aplicação sem afetar outros componentes diretamente.

As características da programação orientada a eventos se ajustam bem aos requisitos dos ambientes com mobilidade, entre os quais a necessidade de permitir que duas ou mais partes possam interagir sem precisar estar diretamente conectadas ou saber a priori da existência de cada uma. Outro requisito dos ambientes com mobilidade é a possibilidade de condicionar a execução de uma tarefa à localização dos componentes, definindo que informações precisam ser enviadas ou tratadas dependendo da localização dos mesmos. Essa necessidade pode ser satisfeita através da flexibilidade dos componentes poderem reagir a determinados eventos, como a mudança de localização das partes envolvidas.

2.2. Coordenação e espaço de tuplas

Modelos de coordenação enfatizam a separação entre a definição do componente e a forma como ele interage com outros componentes [Gelernter and Carriero, 1992]. A abordagem de coordenação incentiva o desacoplamento entre processos, e através dela o código associado aos componentes carrega um número reduzido de mecanismos explícitos de interação.

A mobilidade pode se beneficiar da perspectiva de coordenação[Picco et al., 2001] explorando esse desacoplamento entre as partes de forma a poder estabelecer e desestabe-

lecer conexões entre componentes dinamicamente, sem precisar alterar a estrutura interna de um componente. Por outro lado também pode permitir a reestruturação interna de um componente sem interferir na maneira como ele se relaciona com o meio externo, respondendo assim às necessidades de adaptação do meio.

Linda [Carriero and Gelernter, 1989] foi um dos primeiros trabalhos a discutir o conceito de coordenação, introduzindo a idéia de *espaços de tuplas* como uma estrutura de dados adequada para o desacoplamento sugerido pela coordenação. Um espaço de tuplas consiste de uma memória virtual compartilhada representada por uma estrutura de dados global, persistente e de conteúdo endereçável, cujo acesso se dá de forma associativa, através das próprias características dos dados. É composto basicamente por duas partes distintas: o *espaço* que é a estrutura usada para armazenar os dados, e as *tuplas* que são as estruturas de dados básicas armazenadas no espaço de tuplas. A comunicação por meio de um espaço de tuplas se dá através da inserção, leitura ou remoção de tuplas. Esse estilo de comunicação garante o desacoplamento espacial obtido pelo fato de que quem deposita uma tupla não precisa conhecer quem irá usá-la e vice-versa. Além disso garante também o desacoplamento temporal, ou seja, uma tupla pode ficar armazenada até que a parte interessada em usá-la esteja disponível. Isso simplifica a tarefa de garantir a comunicação entre as partes quando elas não estão disponíveis ao mesmo tempo.

Espaços de tuplas podem ser implementados de formas distintas, adicionando características que favoreçam sua aplicação para lidar com problemas específicos. Um exemplo é a transição do modelo de armazenamento centralizado para um modelo distribuído, mais adequado para ambientes móveis. [Picco et al., 2001] descrevem um sistema denominado Lime, que refina o modelo persistente e globalmente acessível, originalmente proposto por Linda, para um modelo com compartilhamento temporário e vários espaços nomeados e associados permanentemente a uma unidade móvel. O Lime introduz regras para um compartilhamento transitório, baseado na conectividade entre as partes, encapsulando dessa forma um tratamento para a mobilidade. Além disso acrescenta a possibilidade de nomear os espaços de tuplas como uma abstração usual para separar dados relacionados a aplicações distintas.

3. LuaTS: um espaço de tuplas orientado a eventos

LuaTS [Leal et al., 2003] é um implementação de espaço de tuplas construída sobre o sistema *ALua* [Ururahy et al., 2002] – uma extensão de Lua para programação baseada em eventos – com o objetivo de explorar o assincronismo desse ambiente.

O ALua é um sistema baseado em eventos que trata as mensagens como pedaços de código Lua e os executa. Essa execução é feita de forma atômica, ou seja, cada mensagem recebida é executada por completo antes que a próxima mensagem seja tratada, eliminando assim a existência de concorrência em um mesmo processo. Uma aplicação ALua é composta por um grupo de processos que podem rodar em múltiplas máquinas. Cada processo possui um interpretador Lua e um loop de eventos que gerencia os eventos da rede de comunicação e da interface do usuário. Os processos comunicam-se usando a operação "send" assíncrona. A chegada de uma mensagem é tratada como um evento, isto é, não há uma operação "receive" explícita. Assim, usando o LuaTS podemos associar, em uma mesma estrutura de comunicação, características de orientação a eventos e

de coordenação, além de características próprias dos espaços de tuplas, para obter boas soluções para mobilidade.

As operações para ler ou retirar uma tupla no LuaTS são associadas a funções de retorno (chamadas *callback*) que são executadas quando a operação é realizada, ou seja, quando o padrão (*template*) de tupla procurado é encontrado no espaço de tuplas. Essas funções são definidas e executadas no componente que registra o interesse pela tupla e são chamadas por meio de um evento de comunicação. As primitivas básicas oferecidas pelo LuaTS incluem:

- **write**: insere uma tupla;
- **read**: recupera uma tupla associada a um padrão sem removê-la do espaço de tuplas;
- **take**: o mesmo que *read* mas remove a tupla do espaço de tuplas;

Se as operações *take* e *read* não forem servidas imediatamente elas ficam ativas até que alguma tupla associada ao padrão procurado seja inserida no espaço de tuplas. O final da execução é sempre indicado pela execução da função de retorno (uma *callback*) associada à operação, ou a um *timeout*.

Através desse tratamento dado pelo LuaTS às primitivas de acesso ao espaço de tuplas é possível abstrair toda a lógica da comunicação por eventos por meio de um espaço de tuplas. Um evento pode ser emitido através da inserção de uma tupla; e o interesse por algum evento pode ser registrado através das operações de leitura ou retirada de um padrão de tupla. Como essas operações são associadas a funções de retorno, os componentes interessados em um evento serão avisados da sua ocorrência sem que precisem ficar presos aguardando a sua chegada. Adicionalmente a implementação da comunicação por eventos ganha do espaço de tuplas a simplicidade para gerenciar o armazenamento e entrega de eventos quando as partes envolvidas não estão disponíveis ao mesmo tempo.

O LuaTS oferece ainda uma facilidade extra para a comunicação por eventos: a possibilidade de registrar o interesse por padrões de eventos de forma mais acurada, por meio de uma função de busca que caracteriza os padrões de tuplas procurados. Usando uma função, a busca por tuplas pode conter algo mais do que a simples combinação dos seus campos, é possível definir um código capaz de estabelecer relações mais sofisticadas entre os valores carregados por uma tupla. Essa característica satisfaz a um aspecto desejado em sistemas baseados em eventos que é dispor de uma linguagem de inscrição por meio da qual os interesses por eventos são registrados como expressões que serão avaliadas sobre o conteúdo dos eventos [Carzaniga et al., 2001].

4. O ambiente de desenvolvimento de aplicações distribuídas proposto

Com base nos aspectos levantados e discutidos nas seções anteriores, propomos um ambiente de desenvolvimento de aplicações distribuídas que usa o espaço de tuplas LuaTS como meio de comunicação e associa a cada componente um script de configuração que coordena sua interação com o meio, seguindo o paradigma da programação orientado a eventos. Como uma abstração de programação para essa camada de coordenação definimos primitivas que implementam um modelo de chamadas remotas de procedimento assíncronas e síncronas usando corrotinas. Nessa seção apresentamos esse ambiente e sua implementação.

4.1. O modelo de coordenação adotado

A tarefa de coordenar componentes em um ambiente de computação tem por finalidade fazer emergir propriedades globais resultantes da interação entre eles. Dessa forma podemos pensar a computação de um componente em dois níveis: o primeiro responsável pela funcionalidade básica do componente, que serviços ele pode oferecer e como os implementa; o segundo responsável por determinar sobre quais condições e como deverá interagir com outros componentes para compor as aplicações ou executar tarefas em conjunto. [Lopes et al., 2002] propõem um ambiente no qual os componentes individuais provêem um serviço básico e se relacionam por meio de conectores para obter algum resultado desejado. A execução de um conector implica na instanciação dos componentes que satisfazem à sua definição e na sincronização das suas ações, o que os obriga a estarem disponíveis simultaneamente. Em um ambiente com mobilidade consideramos que a necessidade de sincronizar dois ou mais componentes para realizar uma tarefa global não é muito adequada. O melhor é que cada componente saiba como deve interagir com os outros componentes e possa fazer isso com certa independência da presença dos demais. Por isso propomos um modelo de coordenação através do qual associamos a cada componente uma segunda camada de computação, responsável exclusivamente por determinar como se dará a sua interação com o meio externo. Mais especificamente, propomos o uso de uma linguagem de programação interpretada para permitir que a computação nessa camada possa ser alterada dinamicamente oferecendo uma flexibilidade ainda maior para lidar com as mudanças do meio.

Nesse contexto de programação consideramos importante oferecer abstrações para garantir certo grau de simplicidade à tarefa de programação. Como estamos usando um espaço de tuplas, a interação entre os componentes deve ocorrer estabelecendo-se algum padrão para as tuplas que devem ser inseridas e lidas. Embora as operações sobre um espaço de tuplas sejam intuitivas, a necessidade de estabelecer formatos para padrões de tuplas gera um esforço adicional de programação. Além disso, o uso das primitivas do espaço de tuplas faz com que fique explícita no código a diferença entre a comunicação entre e intra processos, o que nem sempre é desejável. Optamos, então, por oferecer ao programador o modelo bem conhecido de chamada remota de procedimento, popularizado nos modelos de componentes. Através dele cada componente especifica sua interação com o meio externo oferecendo métodos que poderão ser chamados externamente e fazendo chamadas a métodos oferecidos por outros componentes. O uso dessa abstração em conjunto com Lua é especialmente conveniente pelo fato de Lua tratar funções como valores de primeira classe, status que podemos estender a funções remotas.

Assim, as aplicações nesse ambiente são constituídas por componentes distintos, que oferecem alguma funcionalidade básica, e estão associados a scripts de configuração que coordenam a interação com o meio seguindo o paradigma de programação orientada a eventos. Cada componente pode estar associado a mais de um script, caracterizando interfaces distintas de comunicação.

4.2. Comunicação via espaço de tuplas

A comunicação via espaço de tuplas foi implementada estabelecendo padrões de tuplas que são depositados e lidos no espaço de tuplas. A Figura 1 ilustra esse processo. O componente que solicita um serviço deposita no espaço de tuplas uma tupla com a *tag*

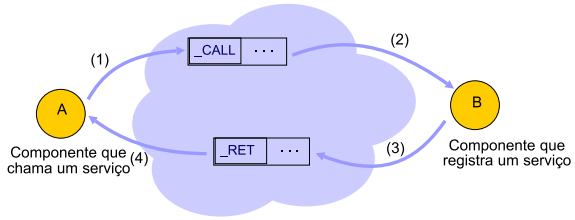

Figura 1: Comunicação através do espaço de tuplas

tag	serviço	args	solicitante
_CALL	"servico x"	{...}	"comp_y"

Figura 2: Padrão da tupla de requisição de serviço

_CALL e com os demais dados necessários para a execução do serviço (1). O componente que oferece um serviço insere no espaço de tupla um padrão com uma função de busca que combina com as tuplas que correspondem às solicitações do serviço oferecido (2). Assim podemos ver a requisição de um serviço como a geração de um evento, cujo conteúdo caracteriza o serviço desejado; e a oferta de um serviço como o registro de interesse por eventos que correspondem ao serviço disponibilizado.

Para retornar o resultado de uma requisição os componentes inserem no espaço de tuplas uma tupla com a *tag* *_RET* e o resultado da requisição (3). O componente que solicita um serviço insere um padrão com uma função de busca que deverá combinar com a tupla de retorno do serviço (4).

A Figura 2 mostra a estrutura do padrão de tupla usado para solicitar um serviço. Além da *tag* *_CALL*, a tupla contém os campos que informam o serviço requisitado, os argumentos necessários e a identificação do componente requisitante para que a tupla de retorno caracterize o componente que requisitou o serviço.

A Figura 3 mostra a estrutura da tupla usada para retornar os resultados da requisição de um serviço. Essa tupla contém, além da *tag* *_RET*, os campos para identificar o componente requisitante, o serviço solicitado e o retorno da requisição.

Essa abordagem de oferta e solicitação de serviços por meio da inserção e leitura de tuplas oferece a possibilidade de definir por quanto tempo um serviço ficará disponível, associando um *timeout* ao padrão que aguarda requisições ao serviço. Com essa característica podemos, por exemplo, implementar um serviço que seja oferecido por um período de tempo experimental com uma interface reduzida e depois seja associado a algum encargo para ser disponibilizado por completo.

tag	solicitante	serviço	retorno
_RET	"comp_y"	"servico x"	{...}

Figura 3: Padrão de tupla de retorno de requisição de serviço

4.3. As primitivas de comunicação oferecidas

As seguintes primitivas foram implementadas para oferecer uma interface de chamada remota de procedimentos:

- luarpc.configura()
- luarpc.inicia()
- luarpc.registra_servico()
- luarpc.cancela_servico()
- luarpc.cria_chamada_servico()
- luarpc.cria_chamada_sinc_servico()

As primitivas *luarpc.configura()* e *luarpc.inicia()* são usadas para configurar e iniciar a interação de um componente com o espaço de tuplas. A primitiva *luarpc.registra_servico()* é usada para disponibilizar um serviço para os demais componentes e consiste em inserir no espaço de tuplas um padrão que aguarda requisições para o serviço oferecido. Como vimos anteriormente, no LuaTS as operações para ler ou retirar tuplas do espaço de tuplas são associadas a funções de retorno, chamadas funções de *callback*. Essas funções são executadas no componente que requisitou a operação quando a tupla procurada torna-se disponível no espaço de tuplas. Assim, a primitiva de registro de serviço recebe como parâmetros valores que caracterizam o serviço oferecido e uma função (a *callback*) que será executada quando uma tupla de requisição para esse serviço for inserida no espaço de tuplas. Essa função poderá chamar diretamente a interface principal do componente ou realizar alguma computação preliminar, chamando algum serviço de outro componente (por exemplo, para computar autorizações do requisitante).

A primitiva *luarpc.cancela_servico()* cancela a oferta de um serviço retirando do espaço de tuplas o padrão que combina requisições ao serviço. Por fim, as primitivas *luarpc.cria_chamada_servico()* e *luarpc.cria_chamada_sinc_servico()* permitem requisitar serviços oferecidos por outros componentes de forma assíncrona e síncrona respectivamente. Elas depositam no espaço de tuplas o padrão de tupla correspondente a chamada do serviço desejado.

Para o caso das chamadas assíncronas, no qual não é preciso sincronizar o retorno da chamada com a sequência de execução do componente, o resultado da requisição pode ser manipulado depois que estiver disponível definindo-se, de forma similar ao caso do registro de serviço, uma função de retorno que será executada quando a tupla com o resultado da execução do serviço estiver disponível no espaço de tuplas. Para o caso síncrono, no entanto, o componente só deve continuar sua seqüência de execução depois que o retorno da chamada estiver disponível. Para garantir que isso aconteça é necessário fazer com que o componente interrompa a sua execução e aguarde a chegada da tupla com o resultado. Porém é importante considerar a possibilidade de um retardo indeterminado na chegada dessa resposta. Pode ocorrer, por exemplo, que o componente responsável pela sua execução esteja temporariamente desconectado. Se simplesmente deixarmos esse componente interrompido aguardando o resultado da chamada, ele não poderá tratar eventuais chamadas aos seus métodos. Portanto, nesse contexto, o ideal é podermos interromper a execução até que o resultado da chamada esteja disponível e enquanto isso permitir que o componente esteja disponível para interagir com outros componentes. Quando o resultado chegar, o componente deverá retomar a execução do ponto em que estava.

Uma solução para esse problema seria adotar um mecanismo de *multithreading*, através do qual toda chamada síncrona fosse tratada como uma nova *thread* de execução. A principal dificuldade dessa solução, nesse caso, é que *threads* tipicamente sofrem escalonamento preemptivo, podendo ser interrompidas em momentos arbitrários. Desse modo precisaríamos implementar um tratamento adicional, acima do escalonador, para gerenciar a execução das *threads* controlando quando devem ser suspensas ou retomadas. Uma solução alternativa é usar *corrotinas*.

Uma corrotina é similar a uma *thread* de execução pois tem uma linha de execução com sua pilha, suas variáveis locais e seu ponteiro de instruções próprios; mas compartilha variáveis globais com outras corrotinas. A diferença é que em corrotinas a execução é sempre suspensa explicitamente, ou seja, corrotinas são um tipo colaborativo de *multithreading* não-preemptivo porque enquanto estão rodando não podem ser suspensas externamente. Essa característica se ajusta bem à necessidade explícita de controlar a suspensão e a retomada da execução das chamadas síncronas. Além disso, quando utilizamos um único processador, a questão da sincronização do acesso a espaços de memória compartilhados é também simplificada pois em muitos casos pode-se programar a corrotina para só ceder a execução quando estiver fora da seção crítica. [de Moura et al., 2004] discute algumas das facilidades que podem ser exploradas usando corrotinas e os contextos nos quais são mais adequadas.

4.4. Implementação de chamada remota de procedimento usando corrotinas

Da mesma forma que um mecanismo de continuação [Kelsey et al., 1998], uma corrotina é representada por um endereço de código e por um ambiente de referência para o qual pode-se saltar por meio de uma função de transferência especial, conhecida como *transfer*. A principal diferença é que uma continuação é uma constante — não se modifica depois de criada — enquanto a corrotina se modifica sempre que é executada. Quando uma continuação é restaurada o estado do programa anterior é perdido, a menos que seja criada explicitamente uma nova continuação para armazená-lo. Quando muda-se de uma corrotina para outra, o estado atual é armazenado e a corrotina corrente é atualizada para refletir isso. A operação *tranfer* salva o contexto corrente do programa na corrotina atual e retoma a corrotina especificada como parâmetro, o programador é quem define a alternância entre as corrotinas. A abstração de corrotinas faz, portanto, o trabalho mais simples de prover uma operação de transferência que elimina a necessidade de salvar e restaurar o estado de execução explicitamente.

A linguagem Lua oferece uma implementação assimétrica de corrotinas com duas funções de transferência: *resume* — que recebe como parâmetro a corrotina a ser retomada — e *yield*, que retorna o controle para a corrotina que ativou a corrotina corrente. Uma característica particularmente interessante da implementação de corrotinas em Lua é a possibilidade de transferir dados entre as operações de transferência e retomada de uma corrotina [Ierusalimschy, 2003] [de Moura et al., 2004].

Em nossa implementação transformamos inicialmente todo o script de comunicação de um componente em uma corrotina. Se nenhuma chamada síncrona for solicitada a corrotina executará até terminar e a aplicação cairá no loop de espera de eventos. Se uma operação síncrona for solicitada, a tupla correspondente é depositada no espaço de tuplas juntamente com o padrão que espera o resultado da requisição — como no caso

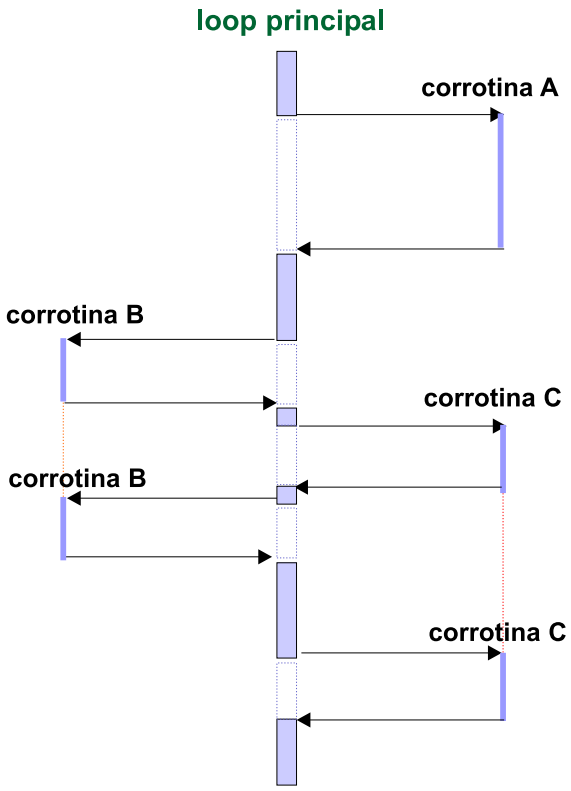

Figura 4: Loop de chegada de eventos associado à criação de corrotinas

das chamadas assíncronas — mas em seguida a linha de execução é tranferida para o ponto onde a corrotina foi iniciada (usando *yield*). Isso fará a aplicação voltar ao loop de eventos. Para que a execução anterior seja concluída ela deverá ser explicitamente retomada quando o resultado da requisição estiver disponível no espaço de tuplas. Para isso associamos à *callback* de retorno do padrão de resultado uma função que retoma a corrotina interrompida. Dessa forma garantimos a interrupção da linha de execução do componente até que o resultado da chamada síncrona chegue sem impedir que ele possa receber e tratar outros eventos enquanto aguarda.

Quando a linha de execução do componente volta para o loop de eventos, o componente pode receber eventos em resposta aos padrões depositados no espaço de tuplas. Esses padrões podem corresponder à requisição de um serviço oferecido pelo componente, à resposta de uma chamada assíncrona ou à resposta de uma chamada síncrona. No último caso, como discutimos no parágrafo anterior, a *callback* associada ao padrão depositado retoma a corrotina suspensa na chamada. Nos demais casos, os eventos estão associados a funções definidas no código do componente dentro das quais podemos ter chamadas síncronas. Para tratar essa possibilidade, as funções de retorno criam novas corrotinas para execução das funções requisitadas Assim, elas serão executadas por completo se não houver na sua implementação chamadas síncronas; ou serão interrompidas e retomadas quando o evento correspondente a chegada do retorno da requisição estiver disponível.

A Figura 4 ilustra esse processo de transferência e retomada da linha de execução de uma corrotina a partir do loop de chegada de eventos. A partir do loop de eventos a

corrotina A é criada e a linha de execução é transferida para ela, que executa por completo, devolvendo o controle para o loop principal. Em seguida a corrotina B é iniciada mas sua execução é transferida antes do seu término. No loop principal, a chegada de um novo evento faz com que a corrotina C seja criada e iniciada. Essa corrotina também é suspensa durante a sua execução devolvendo o controle para o loop principal. Esse recebe o evento que causa a retomada de B e conclui sua execução. Por fim, o evento que causa a retomada da corrotina C também é recebido e a corrotina finaliza a sua execução.

5. Exemplo de aplicação

Para mostrar o uso das primitivas de comunicação implementadas, descrevemos nessa seção uma aplicação ilustrativa, composta por três componentes. O primeiro componente (A) faz a interface com uma biblioteca de acesso a um servidor de diretório para prover um serviço de autenticação. O segundo componente (B) é responsável por interagir com uma biblioteca que disponibiliza métodos de temporização e o último componente (C) disponibiliza um serviço de registro de informações em um arquivo de texto.

A Figura 5 mostra a estrutura da aplicação. O serviço disponibilizado por A é chamado "AutenticaDiretorio"; recebe como entrada o diretório que será consultado, a entrada do requisitante nesse diretório e a sua credencial de autenticação; e retorna uma string que informa o resultado da autenticação. O componente B oferece o serviço "InformaTempo" que retorna o tempo corrente. Por fim, o componente C disponibiliza um serviço chamado "RegistraPresença". Esse serviço autentica o requisitante, toma o tempo corrente e escreve essas informações, junto com a identificação do requisitante, em um arquivo de log.

Assim, como ilustrado na figura 5, o componente A registra interesse pelo evento nomeado "AutenticaDiretorio". O componente B registra interesse pelo evento "InformaTempo" e gera o evento "RegistraPresenca". O componente C, por sua vez, registra interesse pelo evento "RegistraPresenca" e gera os eventos "AutenticaDiretorio" e "InformaTempo".

Para destacar o uso das primitivas de comunicação apresentamos o código principal dos componentes B e C. O código do componente B é mostrado abaixo. Além de registrar o seu serviço de informação do tempo corrente, B faz também uma requisição para registrar sua presença.

```
1  function tempo()
2    return luatimer.gettime()
3  end
4
5  rpc.registra_servico('InformaTempo', "informa_o_tempo_corrente", tempo)
6
7  function recebeConf(ret)
8     print(ret)
9  end
10
11 f = rpc.cria_chamada_servico("RegistraPresenca", recebeConf)
12 f("cTempo", "dir.server.com", "cn=root", "senha")
```

A linha 5 registra o serviço oferecido por B e as linhas 1 a 3 implementam a função que

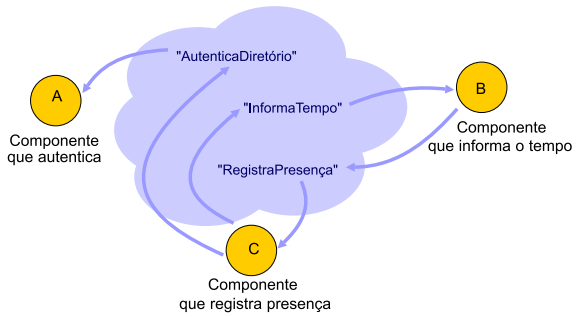

Figura 5: Estrutura da aplicação exemplo

é executada quando uma requisição a esse serviço é atendida. É na implementação dessa função que o componente comunica-se com a biblioteca que contém a implementação do serviço oferecido, nesse caso a biblioteca *luatimer*. As linhas 11 e 12 criam e executam uma chamada assíncrona ao serviço de registro de presença e as linhas 7 a 9 implementam a função que será executada quando o retorno do registro chegar.

O componente C faz requisições a outros serviços para executar o serviço de registro oferecido por ele. O código a seguir mostra a sua implementação.

```
1  function registra(nome, diretorio, dn, senha)
2      f = rpc.cria_chamada_sinc_servico("AutenticaDiretorio")
3      ret = f(diretorio, dn, senha)
4      tempo = rpc.cria_chamada_sinc_servico("InformaTempo")()
5      file = io.open("log", "a+")
6      file:write(tempo.."; " ..nome.."; "..ret.."\n")
7      return ret
8  end
9
10 rpc.registra_servico('RegistraPresenca', "registra a presenca", registra)
```

Entre as linhas 1 e 8 é implementada a função que trata as solicitações ao serviço de registro de presença. Nessa implementação C requisita o serviço de autenticação (linha 2) e o serviço de tomada de tempo (linha 4), ambos de forma síncrona, para que na linha 6 ele possa escrever as informações necessárias no arquivo de log. Por fim, na linha 10, C registra o serviço oferecido.

Além de ilustrar o uso das primitivas implementadas, essa aplicação nos permite destacar uma característica importante do modelo de comunicação adotado, que é permitir que um componente possa tratar novas requisições enquanto aguarda a resposta das suas próprias requisições, sejam elas síncronas ou assíncronas. Por meio desse modelo um componente pode ser chamado para executar parte de uma tarefa requisitada por ele mesmo, é o que acontece quando B requisita seu registro e é solicitado por C para informar o tempo corrente.

6. Conclusões e trabalhos futuros

Neste trabalho discutimos as vantagens de utilizar os modelos de orientação a eventos e de coordenação como base para a programação em ambientes com mobilidade. Mostramos como, em cima desses modelos, podemos explorar abstrações mais conhecidas dos

programadores, particularmente chamadas remotas de procedimentos. Para isso construímos, sobre um espaços de tuplas orientado a eventos, um sistema de chamadas remotas síncronas e assíncronas. As chamadas síncronas são construídas de forma que, mesmo quando um processo aguarda o retorno de uma dessas chamadas, ele não fica bloqueado e pode continuar a receber e tratar novas requisições.

Utilizamos como base da implementação o espaço de tuplas LuaTS, um espaço com implementação centralizada, e não adequado para aplicações móveis. No entanto, existem hoje sistemas que implementam espaços de tuplas com suporte à mobilidade [Picco et al., 2001]. Usando um sistema como esse, o que propomos neste trabalho poderia ser implementado num cenário real de aplicações móveis. Pretendemos explorar essa linha de desenvolvimento.

No sistema que descrevemos, o espaço de tuplas foi utilizado para espelhar um modelo convencional de componentes ou objetos, onde servidores *registram* os serviços que oferecem e cada cliente requisita a um servidor específico a execução do serviço que esse anunciou. No entanto, acreditamos que podemos utilizar uma interface semelhante, de fácil assimiliação pelo programador, para implementar uma série de outros modelos. A operação que chamamos de registro de serviço poderia ser interpretada como o registro de interesse em um evento, criando a base para um sistema do tipo *publish/subscribe* em que o casamento de padrões poderia ser explorado como linguagem de filtragem de eventos. Restaria reimplementarmos a interface de chamada de procedimentos para expandir uma chamada para um "anúncio de evento", invocando não apenas um, mas todos os componentes que houvessem registrado interesse em determinado evento.

Agradecimentos: Agradecemos ao CNPq por financiar bolsas de pesquisa a todos os autores desse trabalho.

Referências

Batista, T. and Rodriguez, N. (2002). Using a scripting language to dynamically interconnect component-based applications. In *VI Simposio Brasileiro de Linguagens de Programação*, pages 180–194.

Carriero, N. and Gelernter, D. (1989). How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357.

Carzaniga, A., Rosenblum, D., and Wolf, A. (2001). Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383.

de Moura, A. L., Rodriguez, N., and Ierusalimschy, R. (2004). Coroutines in lua. In *8th Brazilian Symposium on Programming Languages*, Niteroi, RJ, Brasil. a publicar.

Fiege, L., Muhl, G., and Gartner, F. C. (2002). A modular approach to build structured event-based systems. In *SAC*, Madrid. ACM.

Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Comm. ACM*, 32(2):97–107.

Ierusalimschy, R. (2003). *Programming in Lua*. Lua.org.

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

Kelsey, R., Clinger, W., and Rees, J. (1998). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9).

Leal, M., Rodriguez, N., and Ierusalimschy, R. (2003). Luats - a reactive event-driven tuple space. *J.UCS - Journal of Universal Computer Science*, 9(8):730–744. `http://www.jucs.org/jucs_9_8/luats_a_reactive_event`.

Lopes, A., Fiadeiro, J. L., and Wermelinger, M. (2002). Architectural primitives for distribution and mobility. *SIGSOFT*, pages 18–22.

Picco, G. P., Murphy, A. L., and Roman, G. (2001). Lime: A middleware for physical and logical mobility. In Golshani, F., Dasgupta, P., and Zhao, W., editors, *21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 524–533, MO, USA. Phoenix.

Ururahy, C., Rodriguez, N., and Ierusalimschy, R. (2002). Alua: flexibility for parallel programming. *Computer Languages*, 28(2):155–180.