2
Eventos e Concorrência

Nos últimos anos, o foco da computação distribuída mudou de redes locais para redes de grande escala. Nesse novo cenário, sistemas assíncronos e fracamente acoplados ganharam popularidade devido a características como a latência na comunicação ou a necessidade de permitir que os sistemas comportem um grande número de participantes. Com isso, muita atenção vem sendo dada à programação orientada a eventos para a criação de aplicações distribuídas.

A programação orientada a eventos define um modelo onde as tarefas são codificadas como resposta a eventos recebidos, como o clique em um botão em uma janela ou a chegada de dados em uma conexão de rede (Ousterhout, 1996; Lee, 2006). Orientação a eventos tem um comportamento reativo, onde os eventos são disparados de forma assíncrona e são tratados pelo destinatário no ato do recebimento.

Um conceito relacionado, mas que às vezes é confundido com orientação a eventos, é o de serviço de eventos. Este age geralmente como um mediador entre o produtor do evento e seus consumidores, agregando algumas funcionalidades. Orientação a eventos é um modelo de desenvolvimento, não propriamente um serviço. Por exemplo, o Serviço de Eventos de CORBA (Bolton, 2002) fornece o conceito de canais onde os participantes podem se inscrever para o recebimento de eventos publicados através desses canais. Esse serviço desacopla o emissor dos eventos de seus receptores e possui políticas para armazenamento e distribuição dos eventos. Neste trabalho não tratamos de serviços de eventos.

Os eventos são, geralmente, recebidos e processados isoladamente e ininterruptamente. Os sistemas baseados em eventos dispõem de uma rotina principal denominada *loop de eventos*, que é encarregada de receber um evento e despachá-lo para o seu respectivo tratador. Quando o loop invoca o tratador, o fluxo é desviado para processar o evento e só é retornado ao loop quando o tratador terminar. Nenhum novo evento será recebido e processado enquanto o loop não estiver ativo novamente.

Como um exemplo de programação orientada a eventos, considere a realização do cálculo de uma média de valores que estão espalhados por nós da rede. A figura 2.1 mostra esse exemplo usando a notação do ALua. No ALua, eventos

são chegadas de mensagens contendo uma string de código Lua, e o tratamento dos eventos é a execução do código recebido. A função `request` recebe uma lista de identificadores de processos remotos e envia a cada um desses processos um evento requisitando o valor desejado. O evento de requisição, codificado na string `req`, quando é executado no destinário, faz com este envie de volta ao requisitante o evento `avrg` com o valor para a média.

```
-- Inicializa as variáveis globais
acc = 0
repl = 0
expected = 0

-- Cálculo da média dos valores
function avrg(ret)
  repl = repl + 1
  acc = acc + ret
  if repl == expected then
    print ("Média: ", acc/repl)
  end
end

-- Envia as requisições ao nós
function request(peers)
  -- Guarda o número de nós
  expected = #peers
  local req = string.format("getvalue(%q)", alua.id)
  for i = 1, expected do
    alua.send(peers[i], req)
  end
end

-- Envia o valor para o nó 'id'
function getvalue(id)
  local reply = string.format("avrg(%d)", value)
  alua.send(id, reply)
end
```

Figura 2.1: Exemplo de orientação a eventos com ALua.

A interatividade dos sistemas orientados a eventos depende do tempo de execução de cada evento. Por exemplo, bibliotecas para interface gráfica que adotam o modelo de eventos recomendam que os tratadores sejam o mais breve possível para evitarem a paralisação da interface e darem a impressão que o programa não está funcionando corretamente (Goetz et al., 2006).

Apesar das recomendações, existem tarefas que demandam um tempo maior de processamento. Nesse caso, uma solução é quebrar a tarefa em partes menores. O processamento que até então era realizado em resposta a um único evento, passa a ser dividido em vários tratadores. A tarefa será concluída após a ocorrência de uma seqüência específica de eventos. Isso abrirá espaço para que o loop de evento esteja ativo mais vezes e outros eventos possam ser recebidos ao longo da execução

da seqüência. Se por um lado essa abordagem permite o aumento na interatividade do sistema, por outro temos que lidar com a continuidade da tarefa.

Além das tarefas longas, que podem causar a impressão de travamentos momentâneos da execução, sistemas baseados em eventos também podem sofrer travamentos momentâneos devido a chamadas bloqueantes. Como os eventos são recebidos e processados em seqüência, há apenas um fluxo de execução. Se funções bloqueantes são chamadas dentro de um tratador, todo o processamento fica bloqueado, esperando a conclusão da operação. Isso impede que o loop de eventos seja executado, não abrindo espaço para novos eventos. Por exemplo, se um evento necessita enviar dados por uma conexão TCP e o *buffer* de escrita estiver cheio, o fluxo ficará suspenso.

Assim, dentro dos tratadores, devemos user funções que atuam de forma assíncronas. Com isso, o controle retorna imediatamente ao tratador, que pode realizar outras atividades ou simplesmente retornar ao loop de eventos. Por exemplo, no caso da conexão TCP, poderíamos configurá-la para um modo não bloqueante. Em caso de falta de espaço no buffer, recebemos um código de retorno específico e agendamos uma tentativa futura, evitando o travamento do fluxo de execução.

Funções assíncronas empregam mecanismos de notificação ou consulta para obtenção do resultado do processamento. Na notificação, é comum o uso de funções de *callback*, que são passadas como parâmetro no ato da chamada da função assíncrona. Ao término da operação, a função assíncrona invoca a callback para informar que a requisição foi concluída (podendo passar o resultado como parâmetro).

Na consulta, ao chamarmos uma função assíncrona, esta retorna uma estrutura de controle que é usada para verificar se o resultado da computação já está disponível. Esse caso requer uma abordagem mais ativa, pois temos que, de tempos em tempos, verificar o estado da operação. No cenário de eventos, podemos agendar algum temporizador para realizar essa verificação. Após o término da operação assíncrona, a tarefa deve ser retomada pela callback ou pela consulta explícita.

2.1 Estado, Closures e Multithreading Cooperativa

No modelo de eventos, os tratadores são geralmente implementados como funções, e dessa forma, no fim do processamento de um evento a pilha de execução é desfeita e dados intermediários que estavam lá armazenados são perdidos. Considere, por exemplo, um sistema de compras, onde o produto é escolhido (um computador), seguido da customização (mais memória RAM ou disco) e então o pagamento. Se a compra for dividida nas três subtarefas descritas, a cada passo temos que manter os dados para que o pedido seja fechado. Devemos então mover as informações para o estado global a fim de mantê-las acessíveis aos próximos passos. Em

princípio, isso pode parecer simples quando olhamos a macro divisão acima. Mas a continuidade da tarefa pode requerer divisões em etapas de tamanho bem menor. Então, a cada evento temos que selecionar quais dados devem permanecer ativos no estado global. Esse efeito é conhecido como *stack ripping* (Adya et al., 2002).

Por exemplo, na figura 2.1, tanto `request` quanto `avrg` precisam manter os valores intermediários em globais até que todos os processos tenham respondido. Se, no exemplo, mais de uma média fosse calculada ao mesmo tempo, os valores intermediários seriam sobrescritos, tornando os resultados inválidos. Nesse caso, o programa deveria ser modificado para poder acomodar vários conjuntos de globais e usar alguma forma de diferenciar cada uma das respostas.

Com os tratadores atualizando o estado da aplicação, a programação orientada a eventos se assemelha com a programação em máquina de estados. A aplicação se encontra em um estado, que é modificado com a chegada de um evento, e é levada a um novo estado. Dependendo do nível de granularidade das tarefas, uma aplicação pode ter muitos tratadores. Considerando ainda várias instâncias da mesma tarefa em andamento, a compreensão do funcionamento e a manutenção da aplicação se tornam cada vez mais difícil com tantos estados e tratadores.

Linguagens que implementam o conceito de *closure* (Sussman e Steele, 1998) podem usá-lo para reduzir o esforço em manter o controle do estado da tarefa. Uma closure permite, em tempo de execução, criar funções que capturam o estado da computação, mais precisamente fazendo referência a variáveis no escopo léxico (por exemplo, funções aninhadas acessando variáveis definidas em funções de nível mais externo de aninhamento). Acopladas ao uso de funções de callback, permitem capturar as variáveis de estado, referenciando-as diretamente ao retomar a tarefa, o que evita o salvamento das mesmas em um estado global. A figura 2.2 apresenta um exemplo usando uma closure e callbacks para o cálculo da média apresentado anteriormente. A função `sendRequest` é responsável por fazer as requisições aos nós remotos. Ela recebe como argumentos o identificador do nó e uma função de callback, a qual tratará o valor retornado e calculará a média.

Após o término da operação assíncrona, a tarefa deve ser retomada pela callback. Nesse ponto podemos visualizar um maior efeito do stack ripping. Mesmo tratadores pequenos podem ser obrigados a quebrar a tarefa em várias partes.

Podemos ver no exemplo que os valores são todos mantidos internamente, como variáveis locais da função `avrg`. A função `cb` de callback, criada dinamicamente, faz referência a essas variáveis, que estão em seu escopo léxico, e à medida que os nós vão respondendo à chamada remota, a callback vai atualizando-as. Nessa nova implementação, é possível iniciar vários cálculos de média sem que um interfira no outro, dado que os valores não são compartilhados entre eles.

Outra forma de facilitar a programação das aplicações orientadas a eventos

```
function avrg(peers)
  -- Estado da computação (não global)
  local acc = 0
  local repl = 0
  local expected = #peers

  -- Callback para as requisições remotas
  local cb = function(ret)
    repl = repl + 1
    acc = acc + ret
    if (repl == expected) then
      print ("Média: ", acc/repl)
    end
  end

  for i = 1, expected do
    -- Faz a requisição ao nó remoto
    sendRequest(peers[i], cb)
  end
end
```

Figura 2.2: Utilização de closure para capturar o estado nas callbacks.

é com o uso de multithreading cooperativa. Diferentemente de multithreading preemptiva, onde a troca de contexto entre as threads ocorre de forma transparente, na cooperativa existem funções explícitas para a passagem do controle. No modelo cooperativo também não há paralelismo, apenas uma thread cooperativa está em execução por vez. A gerência da pilha é feita de forma automática, ou seja, quando uma thread cooperativa passa o controle, sua pilha de execução é preservada, não sendo preciso recorrer a um estado global para a retomada da execução.

Neste ponto, gostaríamos de deixar claro o uso dos termos *thread* que será feito no restante deste trabalho. Adotaremos os termos *thread cooperativa* e *multithreading cooperativa* para indicar o modelo de concorrência cooperativa, onde apenas uma thread está ativa por vez e o programador é responsável por efetuar a troca de contexto manualmente. Nesse modelo as threads não se beneficiam de paralelismo real em computadores com vários núcleos de processamento. Ao utilizarmos somente *thread* ou *multithreading*, estaremos nos referindo ao mecanismo de threads na linha de threads POSIX, com preempção e suporte dado pelo sistema operacional para programação multitarefa. Essas threads têm um funcionamento parecido com processos, sendo que a principal diferença é o compartilhamento de memória entre as threads de um mesmo processo. Com os sistemas operacionais modernos dando suporte à multithreading, as threads desse modelo podem ser escalonadas em processadores diferentes ao mesmo tempo, tirando proveito do paralelismo.

As características de multithreading cooperativa se enquadram bem no modelo de orientação a eventos. Os tratadores agora podem ser implementados como

threads cooperativas e têm como opção devolver o fluxo de execução ao loop de eventos sem necessariamente terminar o tratamento completo do evento – mantendo os valores intermediários na pilha. Durante o tratamento dos eventos, eles conseguem transferir o controle de volta ao loop e esperar a ocorrência de um evento requerido para que a execução venha a ser retomada. A figura 2.3 mostra esse comportamento.

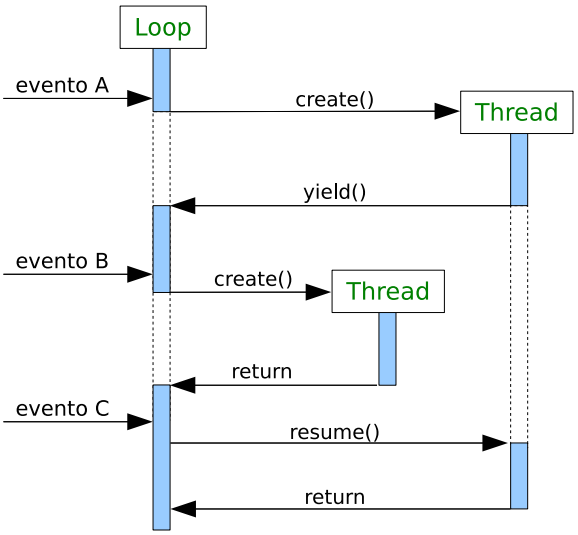

Figura 2.3: Tratamento de eventos com multithreading cooperativa.

O loop de eventos cria uma nova thread cooperativa para tratar o evento *A* e transfere o controle para a mesma. Após algum processamento, a thread termina o processamento do evento *A*, mas a tarefa ainda não está completa. O controle é devolvido ao loop de eventos que recebe um novo evento B. Este é processado por completo por uma nova thread cooperativa. Novamente com o controle, o loop recebe um terceiro evento, *C*, que dá continuidade ao tratamento da tarefa iniciada pelo evento *A*. Desta vez, a tarefa é concluída e a thread termina e retorna o fluxo ao loop.

Além de facilitar a programação dos eventos, multithreading cooperativa introduz um nível de concorrência oportunista na orientação a eventos. Os pontos onde uma thread cooperativa pode suspender são bem conhecidos. Por exemplo, uma aplicação de rede utilizando TCP pode empregar a facilidade de manutenção de pilha de multithreading cooperativa para suspender a tarefa de recebimento ou envio de informações pela rede no caso de *time out* causado pelo controle de fluxo (buffer de envio cheio ou de recebimento vazio), podendo retomar a operação posteriormente.

Por outro lado, esse oportunismo depende das características da tarefa que está sendo executada. Por exemplo, em aplicações que fazem muita entrada e saída, e que possuem suporte a operações não bloqueantes, tomar proveito desses pontos para a interrupção é a melhor escolha. Mas em aplicações com computação

intensa, como aplicações matemáticas ou gráficas, esses pontos nem sempre existem. Multithreading cooperativa também não permite explorar mais de um núcleo de processamento, assim, devemos recorrer a algum mecanismo de multitarefa (ou multiprogramação), como por exemplo, multiplos processos ou threads. De fato, multithreading é o modelo mais adotado pelos programadores devido ao acesso direto aos recursos. Por serem isolados uns dos outros, os processos precisam recair sobre canais de comunicação oferecidos pelo sistema operacional para a troca de dados. Mesmo assim, alguns recursos só fazem sentido dentro de um processo e não podem ser compartilhados com outros.

Apesar das vantagens que multithreading tem no compartilhamento de recursos, a falha na coordenação das threads é apontada como uma das grandes fontes de erros nas aplicações. Multithreading cooperativa não está livre da interferência, pois se uma thread cooperativa é suspensa e depois volta a executar, é possível que o estado global tenha sido modificado por outra thread. No entanto, é mais fácil verificar a consistência no modelo cooperativo devido ao fato de que as interrupções acontecem em pontos bem conhecidos, diferentemente de multithreading, onde a troca de contexto pode ocorrer em qualquer parte do programa.

Ousterhout (Ousterhout, 1996) apresentou vários argumentos contra o uso de multithreading, sugerindo que eventos seriam uma boa alternativa – idéia compartilhada por outros (Lee, 2006; Fischer et al., 2007). Dentre os problemas levantados, podemos citar a dificuldade de programação e depuração, quebra de modularidade, problema em alcançar bom desempenho (granularidade das regiões críticas), bibliotecas de terceiros não preparadas para multithreading e portabilidade. Apesar disso, ele não descartou definitivamente o emprego de threads, pois há certos tipos de aplicações que necessitam de uma alta concorrência ou paralelismo. Ousterhout sugeriu que as aplicações, ao usarem threads, confinassem a concorrência em um núcleo e que o restante fosse construído de tal forma que não fosse preciso lidar com os aspectos de concorrência diretamente.

Behren *et al.* (von Behren et al., 2003) defendem o modelo de multithreading, argumentando que as críticas sobre o desempenho e usabilidade de threads foram motivadas por problemas encontrados em bibliotecas ou pacotes que ofereciam multithreading, e não pelo conceito em si. Eles debatem pontos onde supostamente orientação a eventos leva vantagens, comparando com o modelo de multithreading.

Em termos de concorrência e escalabilidade, a linguagem Erlang (Wikstrom, 1994; Armstrong et al., 1996) é reconhecida pelo seu modelo de processos leves. Erlang implementa o seu próprio conceito de processo o qual não é mapeado diretamente para processo ou thread do sistema operacional. Sistemas operacionais geralmente impõem um limite na criação de threads e processos, por decisão de arquitetura ou por limite de recursos. Com um esquema de processos leves, Erlang

consegue alcançar a criação de um grande número deles.

Cada processo Erlang é isolado dos demais, não havendo compartilhamento de memória, e a comunicação entre eles é feita através de troca de mensagens. Isso evita o problema de concorrência por estruturas compartilhadas mutáveis. Johansson *et al.* (Johansson et al., 2002) discutem arquiteturas de memória empregadas na implementação de Erlang para melhorar o desempenho e o consumo de memória na passagem de mensagem. Dado que os processos de Erlang compartilham o mesmo endereçamento de memória no nível de sistema operacional, juntamente com o fato das estruturas de dados da linguagem serem imutáveis, foi possível organizar a arquitetura interna para otimizar o coletor de lixo e o armazenamento dos valores das mensagens.

2.2 Desempenho e Estruturas Lock-Free

Nos últimos anos, vários trabalhos, linguagens e bibliotecas começaram a explorar a concorrência com threads e o compartilhamento de memória em busca de maior desempenho e escalabilidade usando estruturas chamadas *lock-free* (Barnes, 1993; Goetz et al., 2006; Discolo et al., 2006; Reindeers, 2007). O modelo "clássico" de programação concorrente incentiva a adoção de locks ou semáforos para proteger regiões críticas do programa. O desempenho, escalabilidade e dificuldade de programação desses programas ficam dependentes da granularidade da proteção – quanto menos retenções, maior ganho de desempenho.

Estruturas lock-free se baseiam em atualizações atômicas, geralmente usando instruções de hardware como *compare-and-swap*, para evitar o uso de mecanismos de bloqueio (lock ou semáforo). Os algoritmos empregados para manipular essas estruturas são considerados otimistas, pois eles tentam aplicar a mudança assumindo que a estrutura não foi alterada por outra thread. Caso a alteração falhe, o algoritmo deve recalcular a mudança para aplicá-la novamente.

Por exemplo, considere uma fila implementada com o uso clássico de lock. Quando duas threads tentam adicionar um novo elemento, elas devem primeiro requerer o lock, o que bloqueia todo o acesso à fila. Uma delas conseguirá acesso, adicionará um novo elemento ao final da fila e liberá o lock. Em seguida, a outra thread ganha o acesso e adiciona o seu elemento.

Em uma implementação lock-free da fila, não há um lock de proteção e ambas as threads tentarão colocar um novo elemento no fim de uma fila. Pela atomicidade, apenas uma delas terá sucesso em alterar o final da fila, o que invalidará o cenário esperado pela outra thread. Esta terá que verificar novamente onde é o final da fila para tentar mais uma vez adicionar o seu elemento.

Enquanto as duas threads estão tentando inserir um elemento, outras threads

podem estar consumindo elementos da fila, sem a necessidade de proteção. Essa ausência de mecanismos de bloqueio é o que torna as estruturas lock-free mais eficientes que o modelo clássico (Michael e Scott, 1996; Goetz et al., 2006).

2.3 Eventos, Distribuição e Concorrência em Lua

O GSD-PUC tem investigado as vantagens e limitações da criação de aplicações distribuídas com o modelo de orientação a eventos, principalmente usando a linguagem de programação Lua.

Lua é uma linguagem dinâmica, com uma sintaxe próxima de Pascal, e que possui uma fácil interação com C (Ierusalimschy, 2006). Ela tem se destacado por sua flexibilidade e portabilidade, bem como por seu pequeno tamanho. Lua possui algumas características de linguagens funcionais, como funções como valores de primeira classe, ou seja, funções em Lua podem ser armazenadas em variáveis, passadas a outras funções ou retornadas pelas mesmas, e closure. A *tabela* é a principal estrutura de dados da linguagem, sendo um array associativo capaz de indexar quaisquer valores de Lua. Com essa estrutura básica, é possível criar estruturas mais complexas como filas, árvores, registros, etc.

Para investigar o desenvolvimento de programas distribuídos orientados a eventos, o GSD-PUC construiu o sistema ALua (Ururahy e Rodriguez, 1999; ALua: asynchronous distributed programming in Lua, 2004). Os eventos são mensagens assíncronas contendo trechos de código Lua que são executados pelo receptor no ato da chegada. A primitiva `alua.send` é usada para enviar os eventos e não há uma primitiva para o recebimento explícito. O ALua trata cada evento até o fim antes de iniciar o próximo. Isso evita condições de corrida que levam a inconsistências no estado interno do processo. É importante que os eventos não contenham chamadas bloqueantes, caso contrário todo o processamento ficará bloqueado.

Lua oferece multithreading cooperativa através de co-rotinas (Moura et al., 2004; Moura e Ierusalimschy, 2009). As funções `coroutine.yield` e `coroutine.resume` permitem, respectivamente, suspender uma co-rotina e colocá-la de volta em execução. Esse mecanismo foi usado no ALua para auxiliar a programação dos tratadores de eventos, como descrito anteriormente neste capítulo. Com co-rotinas criamos várias linhas de execução entrelaçadas através de suspensões e retomadas.

No entanto, gerenciar essas várias linhas de execução quando o número de tratadores cresce pode se tornar uma tarefa complexa. Além disso, existe a interação entre os processos remotos. Isso destaca a necessidade de abstrações para coordenar e sincronizar a interação entre linhas de execução. Classicamente, cada sistema oferece o seu próprio conjunto de primitivas de coordenação. Mas nos cenários

onde orientação a eventos se destaca (Ousterhout, 1996), o dinamismo e fraco acoplamento demandam mecanismos mais flexíveis de coordenação, que possam ser redefinidos de acordo com as exigências das aplicações.

A utilização de multithreading cooperativa permitiu combinar a programação orientada a eventos com linhas de execução diferentes sem trazer a complexidade de programação concorrente e preemptiva. No entanto, notamos que o suporte de multitarefa facilitaria a codificação de determinadas tarefas. Além disso, o modelo de concorrência cooperativo não permite explorar as arquiteturas multicore que vêem se popularizando. Neste trabalho, investigamos a questão de multitarefa em Lua em conjunto com orientação a eventos.