3
Trabalhos Relacionados

As limitações do *multithreading* preemptivo, em particular quando associado ao compartilhamento de memória, fomentaram o desenvolvimento de diversos trabalhos orientados à estruturação de alternativas para a programação concorrente. Nas próximas seções são apresentados alguns desses trabalhos, que têm em comum a utilização comedida de *kernel threads* em prol da exploração de *user threads*.

3.1
Sequential Object Monitors

A complexidade na programação com monitores, a ineficiência atribuída à implementação de monitores em Java e a dificuldade em isolar os códigos-fonte associados à sincronização e à aplicação, foram os principais motivadores dos *Sequential Object Monitors* (9), uma alternativa aos monitores padrão de Java.

Os Sequential Object Monitors são um modelo para abstração de concorrência, implementado através de uma biblioteca desenvolvida totalmente em Java, facilmente portável. A denominação "seqüencial" se origina do fato de não ocorrer execução intercalada de métodos, ou seja, uma vez que a execução de um método seja iniciada, ela prosseguirá até a sua conclusão antes que outro método possa ser executado.

O modelo pode ser usado como base para a implementação de outros mecanismos de sincronização de alto nível, e a sua natureza seqüencial se propõe a facilitar o entendimento do código, além de representar ganho de desempenho devido à redução na quantidade de trocas de contexto. O modelo também facilita a separação entre código-fonte associado à sincronização e código-fonte associado à aplicação.

No entanto, a natureza seqüencial descrita anteriormente também é responsável por desestimular o paralelismo, o que pode implicar queda de desempenho em algumas situações e põe em xeque o potencial de comunicação assíncrona. Essa limitação é reconhecida pelos próprios autores da proposta, que defendem que os benefícios da simplicidade justificam sua escolha.

Um Sequential Object Monitor, na prática, é um objeto Java padrão, implementado sem preocupação explícita com questões de sincronização, com um escalonador anexado. O escalonador é responsável por implementar um método de escalonamento que determina como requisições concorrentes serão tratadas. As principais características do escalonador são o seu baixo consumo de recursos, o que implica impacto reduzido no desempenho, e a ausência de *threads* próprias.

Sempre que uma *thread* chama um método de um monitor, a chamada é consolidada como um objeto de requisição. Os objetos de requisição correspondentes às diversas chamadas realizadas para os métodos do monitor são devidamente armazenados em uma fila, enquanto aguardam o escalonamento. Enquanto houver tarefas que podem ser escalonadas, o método de escalonamento é executado e pode selecionar uma ou mais tarefas para execução. As tarefas selecionadas são executadas sucessivamente, em regime de exclusão mútua.

A API associada ao escalonamento é flexível e permite a definição de estratégias distintas de escalonamento. Além de incluir diversos métodos que refletem estratégias comumente utilizadas, a API permite a criação de filtros capazes de alterar a seleção das tarefas, bem como possibilita a fácil adaptação e o reaproveitamento de estratégias de escalonamento.

3.2 Polyphonic C#

O interesse na incorporação de funcionalidades dedicadas à programação concorrente diretamente em linguagens de programação, em conjunto com a necessidade de modelos mais bem adaptados à comunicação assíncrona, resultou na linguagem de programação *Polyphonic C#* (8).

O Polyphonic C# é uma extensão da linguagem C# com novas construções voltadas à comunicação assíncrona, implementadas com base no *join calculus* (13). Na realidade, idéias propostas no *join calculus* foram adaptadas para uma linguagem de programação orientada a objetos que já possuía funcionalidades para programação concorrente, representadas por mecanismos tradicionais de sincronização incluídos no *framework* .NET.

Os métodos, em Polyphonic C#, podem ser declarados como síncronos ou assíncronos. As chamadas a métodos síncronos ficam bloqueadas até que o método retorne, enquanto que as chamadas a métodos assíncronos resultam em retorno imediato. Ainda que o .NET inclua bibliotecas para permitir chamadas assíncronas a métodos, a determinação do assincronismo é realizada pelo responsável pela chamada. Já no Polyphonic C#, o assincronismo é

determinado pelo método chamado.

A principal inovação proporcionada por essa extensão à linguagem C# está na definição de um novo padrão de sincronização denominado *chord* (ou *synchronization pattern* ou *join pattern*). Os chords integram as definições de classes e são compostos por um cabeçalho, que inclui um conjunto de declarações de métodos, e um corpo, que somente é executado após todos os métodos definidos no cabeçalho serem chamados.

As chamadas a métodos integrantes de um chord que ainda não foi habilitado são simplesmente colocadas em uma fila, até que os demais métodos que compõem o cabeçalho do chord sejam chamados. No caso de métodos síncronos, ocorre o bloqueio, no caso de métodos assíncronos, ocorre o retorno imediato. A ordem na qual a fila de métodos é processada não é especificada.

Os cabeçalhos dos chords admitem múltiplos métodos assíncronos e, no máximo, um método síncrono. Na prática, sempre que houver um método síncrono no cabeçalho, o corpo do chord é executado na *thread* associada à chamada desse método. Quando o cabeçalho for composto exclusivamente por métodos assíncronos, uma nova *thread* é disparada para executar o corpo do chord. Essa estratégia implica uso comedido de *threads* e reduz a necessidade de criação de novos fluxos de execução.

Operações atômicas são usadas para determinar se um chord está habilitado e escalonar a execução de seu corpo, o que previne inconsistências na execução concorrente de chamadas aos métodos definidos nos cabeçalhos. O próprio compilador se encarrega de implementar essa atomicidade. Contudo, não há garantia de exclusão mútua durante a execução dos corpos dos chords.

O desempenho da programação concorrente com Polyphonic C# está intrinsecamente atrelado ao desempenho do .NET, uma vez que a abstração de concorrência na linguagem depende de funcionalidades disponibilizadas pelo *framework*. Essa característica, ao mesmo tempo que desonera o desenvolvedor da preocupação com desempenho, configura uma barreira bem definida para otimizações.

3.3 Concurrency and Coordination Runtime

A biblioteca intitulada *Concurrency and Coordination Runtime* (10) representa uma tentativa de facilitar a programação dirigida a eventos por meio de uma proposta para programação concorrente com troca de mensagens. Seu principal objetivo é permitir a comunicação assíncrona por troca de mensagens em uma linguagem de programação orientada a objetos (C#), visando oferecer uma alternativa de concorrência equiparável à utilização de *threads*, mas com

melhor desempenho.

Os eventos, na biblioteca, correspondem ao recebimento de mensagens em uma ou mais portas. Uma vez recebidas, as mensagens disparam código específico para tratar o evento. A arquitetura da biblioteca é baseada na existência de uma quantidade pequena de árbitros, que podem ser compostos para implementar construções de alto nível para sincronização, em particular *join patterns*.

Ainda que possua inspiração no *join calculus*, como o Polyphonic C#, a Concurrency and Coordination Runtime distingue-se por utilizar um modelo de programação baseado em portas. A programação baseada em portas, que pode ser comparada à programação com métodos assíncronos de Polyphonic C#, tem como diferencial o freqüente envio de portas de resposta destinadas à transmissão de retornos de operações assíncronas. Modelagem similar é possível em Polyphonic C#, entretanto é necessária abordagem menos direta e conseqüentemente mais incomum. Além disso, os *join patterns* usados no Polyphonic C# são de natureza estática, ou seja, o código executado quando um padrão é habilitado permanece inalterado. Já na Concurrency and Coordination Runtime, a proposta é que seja possível construir *join patterns* de natureza dinâmica.

As principais entidades da biblioteca representam portas, árbitros e receptores. As portas são parametrizadas de acordo com os tipos de dados que podem receber. Cada porta pode ser associada a um ou mais tipos de dados. Os árbitros são responsáveis por regular o fluxo de dados entre as portas e os receptores, bem como por decidir se mensagens recebidas devem ser consumidas.

Os receptores, por sua vez, representam o código que deve ser executado em decorrência do recebimento de mensagens. A execução é intermediada por um disparador, que mantém um conjunto de tarefas que são progressivamente enviadas para as *threads* responsáveis pela execução. A ativação de um receptor é uma operação assíncrona e é possível enviar mensagens para portas sem receptores. Quando um receptor é associado a uma porta, ele é capaz de processar as mensagens anteriormente recebidas naquela porta, desde que não tenham sido consumidas, e as novas mensagens recebidas naquela porta após a associação.

Alguns dos principais árbitros que já foram desenvolvidos são:

- Receptores de um único item, que definem o tratamento de mensagens em uma única porta;
- Um árbitro que é capaz de usar receptores distintos de acordo com a disponibilidade de mensagens, o que possibilita, por exemplo, a utilização

de receptores diferentes para processar dados de tipos variados;

– Um árbitro de *join* que pode consumir atomicamente mensagens recebidas em mais de um canal;

– Árbitros capazes de realizar operações dinâmicas de *join* através do método *joinvariable*, que cria um *join* englobando um conjunto de portas que pode ser especificado em tempo de execução;

– Um árbitro de intercalação, criado especialmente para possibilitar a expressão, menos suscetível a erros, de cálculos concorrentes que poderiam ser descritos com *join patterns*.

A implementação de uma biblioteca, em oposição à incorporação direta em uma linguagem de programação, foi proposital, ainda que os próprios autores reconheçam a possibilidade de não ser essa a melhor forma de disponibilizar funcionalidades de concorrência. Entre as justificativas estão a adaptabilidade à experimentação e o estágio incipiente das funcionalidades de sincronização. No entanto, essa escolha impõe algumas limitações, como por exemplo a limitação do compilador em analisar e otimizar o código.

A biblioteca ainda carece de testes de desempenho mais expressivos. Entretanto, a indicação preliminar é de que o desempenho é superior à utilização de *threads* ou programação seqüencial, em particular com relação ao consumo de recursos.

3.4 Erlang

A *linguagem de programação Erlang* (6, 7) é uma linguagem funcional, com tipagem dinâmica, desenvolvida para a programação de sistemas de controle em tempo real, como sistemas responsáveis pela comutação em centrais telefônicas, simuladores de redes e controladores de recursos distribuídos. Nesses sistemas é comum a execução concorrente de grandes quantidades de fluxos de execução e é difícil prever com exatidão os consumos de memória e processamento.

Um dos principais atrativos de Erlang é o suporte à concorrência e à programação distribuída, modelado por processos, direto na própria linguagem e independente dos mecanismos disponibilizados pelo sistema operacional. As principais primitivas de concorrência oferecidas pela linguagem são a instanciação de novos processos, o envio de mensagens e o recebimento de mensagens.

A função *spawn* é usada para instanciar novos processos e iniciar a sua execução. Seus parâmetros são um identificador de módulo (similar a

uma classe em programação orientada a objetos), um nome de função e uma lista de parâmetros. Uma vez disparado, o novo processo executa a função especificada, com os parâmetros também especificados. As chamadas à função *spawn* retornam o controle imediatamente e não aguardam a avaliação da função. Seu retorno é o identificador (*process identifier* ou PID) atribuído ao processo recém-disparado.

A principal forma de comunicação entre processos em Erlang é a troca de mensagens. A função *send*, representada pelo ponto de exclamação, é responsável pelo envio de mensagens para outros processos. Os identificadores dos processos são utilizados para endereçar as mensagens. Entretanto, Erlang permite a associação de identificadores de processos a nomes, abstração denominada *registro de processos*. O conteúdo da mensagem pode ser qualquer tipo de dado suportado por Erlang. O retorno da função *send* é a mensagem que foi enviada, e sua execução implica avaliação prévia dos parâmetros, o que significa que é válido utilizar chamadas de funções como destinatário ou conteúdo de mensagens.

O envio de mensagens é uma operação assíncrona, ou seja, não há espera pela entrega e nem confirmação de recebimento. Cabe à aplicação implementar mecanismos para realizar esse tipo de verificação. Entretanto, a linguagem garante a entrega das mensagens caso o destinatário exista e esteja em execução.

A função *receive* é usada para recuperar mensagens enviadas por processos. Cada processo possui um repositório de mensagens, comumente chamado de *caixa postal* (ou *mailbox*), usado para armazenar as mensagens recebidas, na ordem em que foram enviadas. A seleção das mensagens para retirada da caixa postal é implementada com casamento de padrões.

Quando uma mensagem na caixa postal combina com um padrão definido pela função *receive*, a mensagem é retirada da caixa postal e as instruções associadas ao recebimento daquele tipo de mensagem são executadas. Cada padrão definido pela função *receive* pode ainda ter a execução das instruções correspondentes controlada por guardas.

O casamento de padrões é verificado, sucessivamente, a partir das mensagens na caixa postal, o que significa que a ordem na qual os padrões são especificados na função *receive* é irrelevante. Se nenhuma mensagem na caixa postal puder ser casada com os padrões definidos, a execução do processo é suspensa até que seja recebida uma mensagem cujo conteúdo case com algum dos padrões. Mensagens que não são retiradas continuam na caixa postal indeterminadamente.

O casamento de padrões permite, por exemplo, que um processo defina

que deseja receber apenas mensagens originadas de outro processo específico. Nesse caso, bastaria especificar na função *receive* um padrão de mensagem composto por uma tupla formada pelo identificador do processo remetente e pela mensagem propriamente dita. O remetente, por sua vez, envia seu próprio identificador, obtido através de função inclusa na linguagem, juntamente com a mensagem. Contudo, esse mecanismo é baseado na premissa de que o identificador do processo informado na mensagem é verdadeiro.

O recebimento de mensagens pode ainda ser flexibilizado mediante utilização de temporizadores. Um temporizador, em Erlang, é uma expressão definida por um valor inteiro que é interpretado como um período em milisegundos. Se o prazo definido pelo temporizador expirar, sem que nenhuma mensagem tenha sido recebida ou sem que o conteúdo de nenhuma mensagem tenha casado com os padrões definidos, então as instruções associadas ao temporizador são executadas.

Os temporizadores admitem dois argumentos especiais: *infinity* e zero. O zero indica que a expiração do prazo deve ocorrer imediatamente após a primeira tentativa de casar o conteúdo das mensagens na caixa postal com os padrões definidos. O *infinity* indica que o prazo nunca deve expirar.

A programação distribuída com Erlang utiliza essencialmente as mesmas primitivas de concorrência descritas anteriormente. No entanto, a sintaxe das primitivas de instanciação de processos e envio de mensagens é levemente alterada. No caso da função *spawn*, um novo parâmetro representa as referências para o nó no qual o processo deve ser disparado. Já na função *send*, a especificação do destinatário da mensagem inclui o nó no qual o processo destino está em execução.

São disponibilizadas também primitivas para listar os nomes dos nós conhecidos e o nome do nó onde o processo está sendo executado, entre outras voltadas à interação com nós. O nome de cada nó é único e composto por duas partes separadas pelo símbolo de arroba. Normalmente, a primeira parte representa o nome da máquina (ou *hostname*) onde o nó está em execução e a segunda parte representa o domínio da rede.

Além das primitivas focadas nos nós, a programação distribuída em Erlang conta com uma primitiva para monitoração de nós. Essa primitiva recebe como parâmetros um identificador de nó e uma variável booleana. Se o valor da variável for verdadeiro, então o processo que executou a primitiva passa a monitorar o nó passado como parâmetro e, em caso de falha no nó monitorado ou na comunicação pela rede com o nó monitorado, receberá uma mensagem indicativa da falha.

O escalonamento de processos em Erlang é variável de acordo com a

implementação. Qualquer algoritmo desenvolvido para este fim deve observar dois critérios básicos: ser justo, ou seja, qualquer processo que pode ser executado deve ser executado; e não bloquear a máquina por muito tempo.

A gerência de memória, em Erlang, é totalmente transparente para o programador e tanto a alocação quanto a liberação de memória são feitas automaticamente.

Erlang é amplamente reconhecida por proporcionar boa escalabilidade, característica decorrente do foco da linguagem em sistemas de concorrência em larga escala. Alguns exemplos de aplicações que exploram a escalabilidade de Erlang são o servidor *web* de alto desempenho **yaws** (3) e o servidor de *instant messaging* **ejabberd** (2).

3.5 Programação Dirigida a Eventos sem Inversão de Controle

O trabalho de Haller e Odersky (14), apresentado nessa seção, tem por objetivo estruturar uma solução capaz de associar os benefícios do *modelo de atores* (4) e da *programação dirigida a eventos*, o que representaria uma abstração propícia para concorrência, sem ocasionar inversão de controle. Os autores do trabalho sugerem a implementação de atores, sem *threads* e dirigidos a eventos, com o objetivo de concretizar essa solução e permitir a sua utilização em máquinas virtuais que não provêem mecanismos para realizar gerenciamento explícito do estado de execução de software.

O modelo de atores propõe uma arquitetura para processamento concorrente onde cada processo (ou *thread*) é um ator, capaz de se comunicar por troca assíncrona de mensagens e de reagir ao recebimento de mensagens. A utilização desse modelo, em conjunto com reconhecimento de padrões para mensagens, mostrou-se efetiva na linguagem de programação Erlang, descrita em mais detalhes na seção anterior.

O ônus de desempenho, associado ao uso de *threads* implementadas em sistemas operacionais ou máquinas virtuais, é proibitivo para sistemas de controle em tempo real, para os quais Erlang foi projetada. O elevado consumo de recursos de processamento, e especialmente de memória, também é um fator que restringe a utilização de *threads* implementadas em sistemas operacionais ou máquinas virtuais. Essa restrição é particularmente severa no caso de dispositivos móveis de computação, onde a quantidade de memória e o poder de processamento são limitados.

A programação dirigida a eventos, um modelo no qual o fluxo de execução de um processo é determinado por ações do usuário ou mensagens recebidas de outros processos, representa uma alternativa à utilização de *threads*. Um

dos benefícios associados à programação dirigida a eventos é o seu melhor desempenho em comparação ao *multithreading*.

No entanto, o desenvolvimento de software dirigido a eventos é considerado complexo, e até mesmo mais difícil que o desenvolvimento com *multithreading* (33), em grande parte devido à mudança de paradigma. O software passa a ser composto por declarações de interesse em eventos específicos, em oposição a seqüências de chamadas de operações. Na prática, os eventos são responsáveis por disparar a execução de operações e não a seqüência do software em si. A inversão de controle fragmenta a lógica do software e pode requerer que o fluxo de controle de operações voltadas ao tratamento de eventos seja realizado com o auxílio de estados compartilhados. A utilização do modelo de atores, proposta pelos autores do trabalho, tem por objetivo eliminar essa inversão.

Na proposta de Haller e Odersky, a espera de um ator por mensagens não é implementada por meio do bloqueio de *threads*, mas através de *closures* que englobam o processamento de outras atividades dos próprios atores. O *closure* é executado quando um ator envia uma mensagem para outro ator que esteja aguardando mensagens e essa mensagem combina com algum dos padrões definidos pelo destinatário para o recebimento de mensagens. A execução em si é realizada na *thread* do ator que envia a mensagem, o que reduz as criações e destruições de *threads*. O controle é retornado ao ator que enviou a mensagem após a execução, no *closure*, de tentativa não correspondida de recebimento de mensagem ou, se não houver tentativas não correspondidas de recebimento de mensagem, após a conclusão da execução do *closure*. Caso a mensagem enviada não combine com nenhum padrão definido pelo destinatário, ela é simplesmente colocada em uma caixa postal e fica disponível para ser recuperada posteriormente.

Uma limitação desse modelo é a necessidade de estruturar o código de modo que os atores não dependam da conclusão ou dos resultados de um bloco destinado a receber mensagens. Os autores do trabalho alegam que essa não é uma restrição severa. Essa limitação, no entanto, parece intrínseca a qualquer comunicação assíncrona.

A implementação consiste em dois protótipos, ambos desenvolvidos com base em *Scala*, uma extensão à linguagem de programação Java com suporte a construções de concorrência, em consonância com o modelo de atores. A biblioteca se propõe a disponibilizar as funcionalidades essenciais oferecidas por Erlang.

Scala possui uma biblioteca que implementa abstrações similares a processos em Erlang. Nessa biblioteca, a comunicação é realizada através de troca

assíncrona de mensagens e o recebimento de mensagens utiliza casamento de padrões. Os atores são capazes de receber mensagens de qualquer tipo, uma vez que as mensagens são objetos abstratos. No primeiro protótipo, em linha com o funcionamento padrão dessa biblioteca, a criação de um ator é implementada por meio da criação de uma *thread*, o que ocasiona desvantagens decorrentes da utilização do *multithreading*. Por esse motivo foi desenvolvida uma versão adaptada da biblioteca.

Na versão adaptada da biblioteca, que evita a inversão de controle, o envio de mensagens é responsável por inserir a mensagem na caixa postal de destino e verificar se o destinatário está bloqueado, aguardando a mensagem recém-recebida. Em caso afirmativo, a execução do destinatário é retomada, na própria *thread* que executou o envio, até que seja concluída ou até que haja tentativa de receber mensagem.

O recebimento de mensagens é responsável por recuperar mensagens a partir das caixas postais dos atores e, se não houver mensagens disponíveis ou se novo recebimento for executado, suspender a execução. Nesse caso, apenas o recebimento em si, e não a *thread* como um todo, tem a sua execução suspensa. A suspensão da execução é implementada através da sinalização de uma exceção que permite o retorno do controle ao ponto onde foi executado o envio correspondente.

A biblioteca é capaz de distribuir a execução de atores em mais de um processador (ou em mais de um núcleo de processador) mediante utilização de um escalonador que determina a quantidade de *threads* necessárias à execução concorrente dos atores. As retomadas de execução de atores bloqueados são encapsuladas em tarefas, que são submetidas ao escalonador, e posteriormente enviadas para as *threads* responsáveis pela execução. Quando um ator é bloqueado, sua *thread* é liberada para execução de outro ator, o que significa que é possível executar aplicações com quantidade reduzida de *threads*. Novas *threads* somente são criadas quando é constatado que uma *thread* existente foi bloqueada, o que é determinado através de heurística desenvolvida para este fim.

A avaliação de desempenho da biblioteca de atores dirigida a eventos é escassa de comparativos e o trabalho apresenta apenas comparação com a *Simple Actor Language System and Architecture (SALSA)* (32), uma linguagem de programação orientada para o modelo de atores. Nessa comparação, a biblioteca apresentou resultados superiores. Entre os dois protótipos desenvolvidos com Scala, a biblioteca adaptada apresentou escalabilidade superior à biblioteca que utiliza uma *thread* por ator, provavelmente em decorrência do uso comedido de *multithreading*.

Em trabalho subseqüente ao trabalho recém-apresentado (15), os mesmos autores evoluem sua proposta por meio de uma abstração para unificar o modelo de atores baseado em *threads* e o modelo de atores baseado em eventos.