7
Conclusão

A preempção dá ao desenvolvedor uma visão de uma execução ininterrupta, e threads têm ainda o favorecimento do compartilhamento de recursos. Os programadores usam processos ou threads para dispararem diversas tarefas executando simultaneamente, como forma de aproveitar o poder de processamento ou apenas como organização lógica. A construção de aplicações utilizando threads com memória compartilhada é atualmente muito incentivada, principalmente com a popularização de computadores com vários núcleos.

No entanto, essa facilidade tem um preço a ser pago. Coordenar a utilização correta dos recursos compartilhados demanda cada vez mais esforço à medida que a aplicação e o número de tarefas concorrentes crescem. A notoriedade dos problemas de concorrência com threads, devido à dificuldade de lidar corretamente com mecanismos de sincronização e identificar as interações entre as partes das aplicações, favorece a adoção do modelo orientado a eventos (Ousterhout, 1996; Lee, 2006).

7.1
Trabalhos Relacionados

A discussão sobre a predominância de eventos ou multithreading parece não ter um consenso, havendo vantagens e desvantagens atribuídas a cada um deles (Haller e Odersky, 2007). Ousterhout (Ousterhout, 1996), por exemplo, descreveu vários problemas referentes à utilização de multithreading, defendendo a adoção do modelo orientado a evento. Por outro lado, Behren *et al.* (von Behren et al., 2003) defendem o modelo de multithreading, discutindo os argumentos contra esse modelo de desenvolvimento.

Alguns modelos tentam explorar os dois lados, para prover flexibilidade no desenvolvimento das aplicações, podendo empregar a abordagem que melhor se encaixa nos requisitos da aplicação. Por exemplo, SEDA (Welsh et al., 2001) usa um modelo misto com fila de eventos e multithreading. A computação é dividida em estágios, e cada estágio possui sua própria fila de eventos que são tratados por um conjunto de threads. Cada estágio dispara eventos para o próximo, formando um *pipeline*. Scala (Haller e Odersky, 2008; Odersky et al., 2008) é uma linguagem que

permite definir *actors* (Agha, 1986) baseados em threads ou orientados a eventos, atendendo aos dois estilos de desenvolvimento.

Adya *et al.* (Adya et al., 2002) discutem as vantagens e desvantagens de programação com multithreading e orientação a eventos, considerando a distinção entre gerenciamento de pilha *manual* ou *automático*. Um ponto que mais conecta esse trabalho com o nosso é a ênfase sobre permitir que o programador combine livremente ambos os modelos quando codificar uma aplicação. Eugster *et al.* (Eugster et al., 2001) também discutem a importância de combinar diferentes paradigmas de interação (RMI e publish/subscribe) em uma ferramenta simples para programação distribuída.

Arbab e Papadopoulos (Papadopoulos e Arbab, 1998) apresentam uma visão geral sobre modelos de coordenação e linguagens, e classificam os trabalhos analisados em categorias maiores. *Modelos orientados a dados* oferecem primitivas de coordenação que podem ser livremente misturadas com código computacional. Propostas como Linda e os sincronizadores que discutimos na seção 3.2.2 são postas pelos autores nessa categoria. A outra categoria, *modelos orientados a controle*, engloba modelos nos quais as entidades de coordenação são separadas das entidades computacionais. Um exemplo típico dessa categoria são as linguagens de *configuração* (Magee et al., 1994; Arbab et al., 1993), que focam na descrição das interconexões entre processos ou componentes independentes.

Em (Backus, 1978), Backus argumenta que linguagens de programação em geral deveriam prover um núcleo expressivo, e partes acopláveis poderosas para construir novos procedimentos e formas de combiná-los, em vez de incorporar um grande conjunto de características e tornar a linguagem menos flexível. E no domínio específico de concorrência e distribuição, Briot (Briot et al., 1998) discute as abordagens integrativa e por biblioteca, onde a primeira provê mais coerência e a última mais extensibilidade para a linguagem. No entanto, parece que poucos trabalhos recentemente exploram como as características das linguagem interagem com bibliotecas para permitir maneiras uniformes para concorrência e distribuição dentro da linguagem.

Acute (Sewell et al., 2007) é uma linguagem baseada em ML que tem o objetivo de apoiar o desenvolvimento distribuído, permitindo uma interação segura de tipos entre as partes, carga dinâmica de código, amarração de recursos locais, globais, nome, etc. Os autores citam que prover todos os requisitos de distribuição em uma única linguagem de propósito geral pode ser uma tarefa sem fim. Em vez disso, eles argumentam que a linguagem deve tornar distribuição e comunicação explícitas e permitir que bibliotecas sejam implementadas como abstrações de alta ordem.

O trabalho de Varella e Agha (Varela e Agha, 2001) e o de Eugster *et*

al. (Eugster et al., 2001) são dos poucos que abordam a discussão de bibliotecas para programação distribuída do ponto de vista da linguagem. Varella e Agha apontam a necessidade de extensões para Java, indicando que o conjunto original de características da linguagem é insuficiente, mas por outro lado, defendem que somente umas poucas extensões podem ser suficientes para prover um modelo simples de programação distribuída. Eugster *et al.* identificam um conjunto de mecanismos que reforçam o suporte da linguagem à publish/subscribe. Mais especificamente, eles identificam o conceito de closures como um desses mecanismos. O trabalho deles também tem a intenção de contribuir para a discussão se o suporte para a comunicação publish/subscribe em linguagens orientadas a objetos deve ser provido como uma biblioteca ou como uma parte da linguagem de programação.

7.2 Contribuições do Trabalho

Uma das contribuições deste trabalho é mostrar como algumas características da linguagem Lua podem ser usadas para construir diferentes mecanismos de coordenação. Embora estejamos particularmente interessados em explorar essas características em Lua, nosso objetivo é contribuir na discussão sobre o papel das características da linguagem para preencher a lacuna entre as abordagens de linguagem ou biblioteca na programação.

O sistema de programação específico que descrevemos é simplesmente um ambiente usado para demonstrar que as abstrações de programação, as quais simplificam as aplicações, podem ser facilmente implementadas e combinadas dado um pequeno conjunto de características da linguagem.

As funções como valores de primeira ordem cumpriram um papel importante na construção dos mecanismos e abstrações. Por serem manipuladas como valores, elas permitiram uma incorporação mais fácil e uniforme das funcionalidades na linguagem. O programador lida com as funções criadas dinamicamente para efetuar chamadas remotas ou fornecer a proteção do monitor da mesma forma que ele lida com as funções que já são providas pela linguagem. Além disso, a criação dinâmica de funções com a contribuição do escopo léxico para acesso direto às variáveis simplificou a implementação interna dos módulos. Pudemos capturar mais facilmente o estado atual da computação e definir continuações utilizando essas duas características da linguagem.

Empregamos a criação dinâmica e passagem de funções em todas as abstrações apresentadas no capítulo 3. Na construção do RPC sobre o ALua, utilizamos a passagem de callback para `rpc.async` e a construção interna de uma callback para o `rpc.sync`. O monitor recebe a função a ser protegida e retorna uma nova versão da mesma, provendo a construção dinâmica da proteção. Nas restrições de sincro-

nização e no sincronizador, os guardas são funções que devem ser consultadas para liberar a execução das mensagens. Em todos os casos, o escopo léxico permitiu o acesso direto à funções e variáveis, evitando criar e gerenciar estruturas auxiliares para manter essas referências, simplificando a montagem da arquitetura interna.

Co-rotina foi outra característica de Lua que empregamos na contrução de nossas abstrações. A utilização de multithreading cooperativa no auxílio à programação dos tratadores dos eventos é conhecida. Por meio das co-rotinas foi possível implementar a suspensão do RPC síncrono nos eventos, dando uma visão mais linear à programação, em vez de lidar somente com inversão de controle das callbacks. Essa suspensão permitiu também a criação do lock empregado no monitor, construído sobre as chamadas síncronas para efetuar o bloqueio. Além disso, a otimização no sincronizador foi implementada com as co-rotinas, por meio das quais pudemos salvar automaticamente todo o contexto da chamada ao sincronizador para retomá-la posteriormente apenas resumindo a co-rotina.

De forma geral, essas características da linguagem contribuiram não só na montagem das abstrações, mas também do modelo e das aplicações do capítulo 6.

Uma segunda contribuição deste trabalho foi a investigação de um modelo de eventos para Lua, que disponibiliza linhas de execução concorrentes através de threads ou processos. O objetivo do modelo é manter as características de orientação a eventos, tentando fornecer ao desenvolvedor um ambiente onde ele possa explorar a paralelização com multitarefa, mas que ao mesmo tempo, não seja exposto a todos os detalhes de concorrência.

Optamos por estender o ALua, introduzindo o conceito de *processos Lua*. O programador tem a visão de que esses processos são a base do desenvolvimento, e que a interação entre eles se dá de forma similar independentemente deles estarem dentro do mesmo processo do sistema operacional (executando com threads) ou em processos distintos.

A linguagem Lua não foi desenvolvida para tirar o máximo proveito de ambientes multithread. Com isso, há alguns desafios em oferecer um modelo consistente para concorrência. Por outro lado, a linguagem é muito flexível e pode estender e ser estensível em C.

Além das abstrações, empregamos o mecanismo de co-rotina de Lua na construção do nosso modelo de escalonamento que permitiu desacoplar as threads dos estados Lua. Lua disponibiliza para a linguagem C funções de manipulação de co-rotinas. Utilizamos essas funções para iniciar um programa Lua de tal forma que fosse possível suspender e retomar a sua execução de acordo com a chegada de eventos. Com esse mecanismo foi possível construir um escalonador em C que aloca threads para processarem os estados Lua. A thread executa o estado Lua até que este, por meio do mecanismo de co-rotina, suspenda a execução esperando um evento.

O controle retorna ao escalonador, que aloca a thread para outro estado Lua. Mas vale ressaltar que seria interessante dispormos de uma primitiva que suspendesse imediatamente a execução do estado, retornando ao lado C diretamente, pois vimos que há alguns casos em que essa suspensão não pode se dar.

Na construção do modelo, exploramos a localização dos processos Lua para a criação de mecanismos de comunicação que pudessem aproveitar as características do ambiente para oferecer mais eficiência. Exploramos, por exemplo, o compartilhamento de memória com a utilização de estruturas *lock-free* para auxiliar na troca direta de mensagens entre os processos Lua. Essas estruturas empregam operações atômicas de memória, evitando o mecanismo de lock tradicional. Por outro lado, trabalhar com essas estruturas altamente concorrentes requer abordagens diferentes do que estávamos acostumados.

Implementamos dois casos de uso para validar algumas das características do modelo proposto. No primeiro, adaptamos o servidor web Xavante, permitindo ver o efeito de multithreading na paralelização do atentimento das requisições e o nosso mecanismo de desacoplamento de estados. No segundo caso de uso, criamos uma versão do executor de fluxos do MPA. Utilizamos os mecanismos de RPC e a abstração de monitor discutida anteriormente com os processos Lua. Isso mostra que apesar desses mecanismos e abstrações terem sido inicialmente propostos para um ambiente distribuído, eles podem ser explorados (e combinados) para a construção de aplicações com os processos Lua.

Finalmente, apesar do esforço em manter a visão uniforme de interação entre os processos Lua, o desenvolvedor deve estar ciente do funcionamento e características do modelo implementado. Isso interfere na escolha de como montar a aplicação e quais implicações essas decisões terão: velocidade de comunicação, número de threads disponíveis para os processos Lua, isolamento das partes, etc.

7.3 Trabalhos Futuros

O modelo que propomos dá ao desenvolvedor a flexibilidade de lidar com eventos longos ou bloqueantes em orientação a eventos. Por outro lado, é responsabilidade do desenvolvedor dimensionar o número de threads necessárias para atender os processos Lua. Uma linha de pesquisa é traçar estratégias para que o próprio sistema regule de forma automatizada a relação processos Lua e threads. Seria interessante dispor de fatores que apontem, por exemplo, relações de utilização de recursos dos processos Lua e o número de threads necessárias para manter o sistema com tempo de resposta aceitável. E com a transparência na comunicação, um modelo de balanceamento de carga também poderia ser investigado.

Atualmente, o modelo adota um único conjunto de threads para atender todos

os processos Lua. SEDA e Helper Threads permitem criar conjuntos independentes de threads e atribuir as tarefas a eles. Isso permite definir o poder de processamento, podendo refletir no tempo de resposta para tipos diferentes de tarefas. Seria interessante fornecer esse nível de controle ao desenvolvedor e analisar o impacto na abstração dos processos Lua.

Em um caráter mais prático, reavaliar a forma de funcionamento das bibliotecas adicionais de Lua, levando em consideração a utilização em diversos estados. No desenvolvimento do trabalho, foi necessário alterar, por exemplo, a biblioteca de socket e criar módulos que estendiam ou contornavam situações devido ao forte acoplamento com um único estado.

Outro ponto seria rever a relação dos processos com o daemon, e até reavaliar a necessidade do mesmo. Poderíamos explorar os conceitos e tecnologias *peer-to-peer* para dar mais flexibilidade à estrutura do ALua e até mais desempenho.