5
Implementação do Modelo

Em sua configuração padrão, Lua inclui suporte à programação concorrente por meio de co-rotinas. As co-rotinas são fluxos de execução distintos no espaço do usuário (*user threads*). O controle da execução segue modelo cooperativo e depende de chamadas explícitas às funções `yield`, que resulta na cessão da vez à outra co-rotina, e `resume`, que resulta na retomada da execução de uma co-rotina a partir de onde foi suspensa.

A API de Lua para programação em C inclui funções para criação de co-rotinas, bem como para suspensão e retomada da execução de co-rotinas. Essa facilidade, aliada à flexibilidade que a API oferece para interação com o interpretador Lua a partir de código C e à dissociação entre co-rotinas e *kernel threads*, torna a linguagem de programação Lua propícia à exploração de modelos alternativos de concorrência nos moldes desejados.

A utilização de código Lua a partir de aplicações desenvolvidas em C normalmente é precedida pela criação de um estado Lua, representado em C por uma variável do tipo `lua_State`. Um estado Lua define o estado do interpretador Lua e mantém registro de funções e variáveis globais, entre outras informações relacionadas ao interpretador. É possível criar múltiplos estados Lua a partir de código C, o que viabiliza a existência de múltiplos códigos Lua independentes.

Uma vez que o código Lua tenha sido carregado em um estado Lua, é possível controlar a sua execução por meio de funções disponibilizadas na API de Lua para programação em C. Esse controle é realizado como se o código Lua fosse executado a partir de uma co-rotina. Dessa forma, mesmo que o código Lua não inclua chamadas explícitas às funções padrão de Lua para manipulação de co-rotinas, é possível suspender e retomar a sua execução a partir de funções implementadas em C. Essa característica é fundamental para permitir o controle da execução dos processos Lua.

Como parte deste trabalho, foi desenvolvida, em C, uma biblioteca para programação concorrente que pode ser utilizada diretamente a partir de código Lua. Essa biblioteca utiliza co-rotinas e estados Lua para implementar os processos Lua. Cada processo Lua é executado como uma co-rotina exclusiva

dentro de um estado Lua independente. A execução efetiva dos processos Lua é realizada por trabalhadores, *kernel threads* implementadas com a biblioteca POSIX Threads. Não há relação fixa entre trabalhadores e processos Lua. Apesar de usar *kernel threads*, como cada processo Lua tem um estado Lua independente, não há memória compartilhada entre processos Lua.

Nos moldes de Erlang, onde a criação de novos processos é estimulada e tem ônus reduzido, a biblioteca foi implementada com o intuito de oferecer boa escalabilidade, em particular com relação aos modelos que utilizam exclusivamente *multithreading* preemptivo no nível do *kernel*. A API da biblioteca é apresentada no Apêndice A.

As seções a seguir apresentam uma descrição detalhada dos componentes e da estrutura da biblioteca.

5.1 Estrutura dos Processos Lua

Cada processo Lua é composto por um estado Lua independente, onde é carregado o código Lua definido como parâmetro para a criação do processo. A independência de estados Lua significa que não há compartilhamento de memória entre os processos Lua, conseqüentemente a comunicação entre processos Lua somente pode ser realizada através dos mecanismos providos pela própria biblioteca.

A estrutura utilizada na implementação dos processos Lua é enxuta e possui poucos membros além do estado Lua associado ao processo. Entre eles destacam-se o estado do processo Lua (ocioso, pronto para execução, bloqueado ou terminado) e a quantidade de parâmetros que devem ser passados na retomada da execução do processo após bloqueio. Nenhum identificador único de processos é utilizado.

Apesar da criação de estados Lua ser uma operação considerada leve, a carga de todas as bibliotecas padrão de Lua requer mais de dez vezes o tempo necessário à criação de um estado (21). Dessa forma, com o objetivo de desonerar a criação de processos Lua, apenas as bibliotecas `basic` e `package`, além da própria biblioteca para programação concorrente com processos Lua, são carregadas automaticamente nos estados criados em decorrência da criação de processos. As demais bibliotecas (`io`, `os`, `table`, `string`, `math` e `debug`) são pré-registradas e carregadas somente mediante chamada explícita à função padrão `require`.

Ainda no sentido de desonerar a criação de processos Lua, a biblioteca para programação concorrente oferece uma funcionalidade, de ativação opcional, para reciclagem de processos. A reciclagem consiste no reaproveitamento

de processos Lua concluídos para execução de novos processos. Em vez de ser destruído após a sua conclusão, o processo Lua pode ser armazenado e ter os membros de sua estrutura restaurados para os valores iniciais. Assim, a criação de um processo Lua pode se resumir apenas a carregar o código Lua no estado Lua de um processo reciclado, o que elimina os custos de criação de um novo estado e carga de bibliotecas.

Avaliações quantitativas referentes ao custo de carga das bibliotecas padrão e aos ganhos ocasionados pela reciclagem de processos Lua são apresentadas no capítulo 6.

5.2 Escalonador

O escalonador é responsável por ordenar a execução dos processos Lua e efetivamente executar os códigos Lua associados aos processos. A inicialização do escalonador é realizada de forma implícita durante a carga da biblioteca e ocorre no contexto do processo do sistema operacional responsável pela execução do código que resultou na carga da biblioteca. Durante essa inicialização, é criada uma *kernel thread* implementada com a biblioteca POSIX Threads.

Quando um novo processo Lua é criado, o que envolve a criação de um estado Lua e a carga de código Lua nesse estado, é responsabilidade do escalonador inserir o processo recém-criado no final de uma fila (FIFO) de processos prontos para execução. A ordenação dos processos Lua nessa fila determina a ordem na qual eles são executados.

A *kernel thread* criada durante a inicialização do escalonador representa um trabalhador cujo único propósito é executar o código associado aos processos Lua. A *thread* executa um ciclo que consiste em retirar o primeiro processo Lua da fila de processos prontos para execução, executar o código Lua associado ao processo e tomar as providências cabíveis de acordo com o resultado da execução.

Se a execução terminar em conseqüência da conclusão do código Lua associado ao processo, cabe ao trabalhador encerrar o estado correspondente e destruir o processo Lua. Se a execução resultar em bloqueio do processo Lua por chamada explícita no código à função padrão `yield`, cabe ao trabalhador simplesmente reinserir o processo no final da fila de processos prontos para execução. Se a execução resultar em erro inesperado, cabe ao trabalhador emitir um aviso de erro, encerrar o estado correspondente e destruir o processo Lua.

A fila de processos Lua prontos para execução é única, ou seja, todos os trabalhadores retiram processos da mesma fila. Tal fato demanda a utilização de mecanismos de sincronização baseados em memória compartilhada para

serializar as operações de acesso e manipulação de processos Lua na fila. Nesse sentido, são utilizadas variáveis condicionais e exclusão mútua, ambas funcionalidades suportadas diretamente pela biblioteca POSIX Threads.

A biblioteca oferece funções para criação e destruição de trabalhadores em tempo de execução. Como os trabalhadores são associados univocamente a *kernel threads*, a cada trabalhador criado é criada uma nova *kernel thread*. Analogamente, a cada trabalhador destruído é destruída uma *kernel thread*.

5.3 Comunicação entre Processos

A passagem de dados entre códigos C e Lua é realizada com o auxílio de uma pilha virtual cujos elementos representam tipos definidos em Lua. Chamadas em Lua a funções implementadas em C usam a pilha para passar os argumentos necessários à execução das funções. Analogamente, funções implementadas em C usam a pilha para retornar resultados para Lua. Dessa forma, a troca de mensagens em si consiste apenas na cópia de dados entre as pilhas dos estados associados aos processos Lua envolvidos na comunicação.

5.4 Estratégia de Bloqueio

O bloqueio de processos Lua pode ocorrer em duas situações distintas:

1. quando um processo Lua faz uma chamada à função de recebimento síncrono de mensagens e passa como parâmetro um canal no qual não há envios pendentes, ou seja, quando ocorre uma tentativa de recebimento de mensagem sem prévia tentativa de envio correspondente;

2. quando um processo Lua faz uma chamada à função de envio de mensagens e passa como parâmetro um canal no qual não há recebimentos pendentes, ou seja, quando ocorre uma tentativa de envio de mensagem sem prévia tentativa de recebimento correspondente; ou

As chamadas explícitas à função `yield` em código Lua resultam na suspensão da execução do processo Lua. Nesse caso o trabalhador apenas reinsere o processo Lua na fila de processos prontos para execução e retira o processo Lua no início da fila para executar. O processo Lua recém-suspenso tem a sua execução retomada tão logo atinja o início da fila de processos prontos para execução e seja retirado por um trabalhador.

Nos casos de tentativa de envio de mensagem para canal no qual já houve tentativa de recebimento ou tentativa de recebimento síncrono a partir de canal para o qual já houve tentativa de envio de mensagem, o fluxo de execução

do código Lua procede normalmente e não ocorre bloqueio. A ausência de chamadas à função `yield` e às funções de envio e recebimento de mensagens resulta em fluxos de código Lua que são executados sequëncialmente até a sua conclusão.

As tentativas não correspondidas de envio ou recebimento de mensagens ocasionam o bloqueio do processo Lua. O bloqueio implica alteração do estado do processo Lua, interrupção da execução do código Lua e inclusão do processo em uma fila associada ao canal referenciado na comunicação.

Cada canal possui duas filas: uma destinada aos processos bloqueados por tentativas de envio de mensagens para o canal e outra destinada aos processos bloqueados por tentativas de recebimento de mensagens a partir do canal. Como não é possível que haja processos bloqueados simultaneamente por realizarem tentativas complementares de envio e recebimento no mesmo canal, no máximo uma dessas filas pode não estar vazia em um dado instante.

Quando um processo Lua é bloqueado durante uma tentativa de envio ou recebimento de mensagens, a *kernel thread* responsável por sua execução é liberada e pode passar a executar outro processo Lua da fila de processos prontos para execução. Nesse caso, o desbloqueio do processo Lua somente ocorre mediante chamada à função complementar de comunicação. Assim, quando um processo Lua faz uma chamada à função de envio para canal onde já houve tentativa de recebimento, ou à função de recebimento a partir de canal para o qual já houve tentativa de envio, a própria *kernel thread* responsável pela execução da chamada também se encarrega de remover o processo Lua bloqueado da fila correspondente do canal, copiar os dados entre as pilhas virtuais e colocar o processo recém-desbloqueado na fila de processos prontos para execução.

5.5 Ambientes Distribuídos

A biblioteca para programação concorrente em Lua, desenvolvida conforme o modelo proposto, não inclui os mecanismos necessários à utilização em ambientes distribuídos. A interação com o escalonador somente é possível a partir do processo do sistema operacional a partir do qual é executado, ou seja, a criação de novos processos Lua somente pode ser comandada a partir do código Lua executado inicialmente ou de processos Lua criados a partir dele.

Contudo, essa limitação pode ser facilmente superada com o auxílio de uma biblioteca que ofereça mecanismos para comunicação através da rede a partir de código Lua, como a biblioteca *LuaSocket* (25), por exemplo. A combinação das duas bibliotecas viabiliza a exploração do modelo proposto

para execução de processos Lua em ambientes distribuídos.

Uma das formas mais simples de fazê-lo consiste na implementação, em Lua, de um serviço de escalonamento remoto, composto por um componente servidor e um componente cliente. O servidor tem o objetivo de disponibilizar um escalonador local e um *socket* com o qual podem ser estabelecidas conexões para a transmissão de código Lua. Os códigos Lua recebidos a cada conexão são posteriormente utilizados para criar novos processos Lua, submetidos ao escalonador local para execução. O funcionamento do serviço consiste apenas em um eterno ciclo de recebimento de conexões e criação de novos processos Lua.

O cliente é voltado à criação de novos processos Lua. Ele fica encarregado de estabelecer uma conexão de rede com o *socket* disponibilizado pelo serviço de escalonamento, transmitir o código Lua lido a partir de um arquivo e encerrar a conexão após a transmissão. A implementação dos componentes servidor e cliente do serviço de escalonamento remoto é apresentada no Apêndice B.

A implementação contempla apenas as funcionalidades básicas para exemplificar a disponibilização de um escalonador em ambiente distribuído. Entretanto, é possível estendê-la para torná-la mais robusta, segura ou mesmo oferecer outras funcionalidades.

A preocupação com segurança, em particular, é patente em ambientes distribuídos. A ausência de mecanismos de controle de acesso possibilita a execução remota e arbitrária de código Lua, com os privilégios do usuário remoto responsável pela execução do processo do sistema operacional, que efetuará a execução do código Lua – o usuário que executa o componente servidor, no caso do serviço de escalonamento remoto.

O risco de execução remota e arbitrária de código Lua pode ser mitigado de inúmeras formas. Uma infra-estrutura de chaves públicas poderia ser utilizada visando emitir certificados digitais para as partes envolvidas na comunicação. Os certificados poderiam ser utilizados tanto para assinar os códigos Lua submetidos à execução, assegurando que apenas os códigos autorizados seriam executados, como para resguardar a confidencialidade e a integridade dos dados em trânsito, mediante utilização de canais de comunicação cifrados.

Outras formas menos elaboradas poderiam ser adotadas como substitutas ou complementares, de acordo com o nível de segurança desejado. Entre elas estão a utilização de uma camada de filtragem, anterior à execução de código Lua, que ficaria encarregada de impedir chamadas a funções específicas ou criticar argumentos a essas funções. Outra alternativa seria a utilização de segredos compartilhados entre as partes envolvidas na comunicação. A execução de código Lua somente seria permitida mediante apresentação do

segredo correto. Essa última alternativa, no entanto, requer cuidado com relação à preservação dos segredos durante o trânsito através da rede.

A programação distribuída em Lua já foi alvo de estudos mais minuciosos (31), que resultaram na implementação de ambientes de execução mais estruturados para este fim.