1
Motivação

Sistemas Distribuídos possibilitam e normalmente demandam que diversas tarefas sejam realizadas simultaneamente. Novas tarefas podem ser requeridas mesmo quando tarefas requeridas anteriormente ainda não terminaram de ser atendidas.

Em muitos casos pode-se aumentar a escalabilidade de um sistema, ou seja, a quantidade de tarefas que ele consegue processar, aumentando o número de máquinas, o número de processadores, o tamanho da memória, aumentando a velocidade desses *hardwares*, ou utilizando esquemas de comunicação mais rápidos. Porém os modelos de programação utilizados na implementação podem também ter um impacto no desempenho desses recursos, muitas vezes permitindo o uso de máquinas mais simples e em menor quantidade. O uso de diferentes técnicas de programação, onde se aproveite melhor a memória, os ciclos de máquina, se minimize a espera por E/S, ou se minimize a espera para executar trechos que precisem acesso exclusivo a algum recurso compartilhado, pode resultar em desempenhos significativamente diferentes.

Um dos contextos onde a escalabilidade é reconhecidamente um requisito importante é o de servidores web. Nesse trabalho estudamos o impacto da escolha de diferentes modelos de concorrência e de isolamento de tarefas por *sandbox* no desempenho de um servidor web. Utilizamos servidores baseados na linguagem Lua (ry96-4), que nos permite explorar tanto modelos de programação concorrente e isolamento de dados mais comuns, como outros, pouco utilizados em outras linguagens, como facilidades para implementar concorrência colaborativa usando co-rotinas e isolamento de dados entre máquinas virtuais em um mesmo processo.

O modelo de concorrência pode afetar drasticamente a escalabilidade de um sistema, pelo uso de ciclos de máquina, pelo uso de espaço em memória ou pelo aproveitamento simultâneo de múltiplas CPUs. Existem poucos trabalhos comparando o desempenho de modelos de concorrência usando co-rotinas, especialmente em Lua. Gostaríamos de ter medidas comparativas de desempenho envolvendo diferentes modelos de concorrência.

O uso de um determinado modelo de programação concorrente normal-

mente também impacta em restrições nos modelos de isolamento de dados disponíveis. Cada modelo oferece vantagens e desvantagens em relação aos outros. Por exemplo, modelos de concorrência colaborativos tendem a oferecer alta escalabilidade ao custo de perda de isolamento entre as tarefas. Procuramos nesse trabalho observar essas relações.

Existem diversos servidores web disponíveis no mercado. Esses servidores utilizam diversos modelos de concorrência e isolamento de dados. Portanto, apesar do nosso trabalho se concentrar em servidores escritos em Lua, ele pode fornecer insumos para a compreensão do desempenho e vantagens relativas de diversos outros sistemas.

1.1 Termos utilizados

Na Ciência da Computação, autores diferentes e até mesmo o mesmo autor, muitas vezes utilizam os mesmos termos para descrever conceitos diferentes. Por isso alguns termos, que poderiam receber interpretações diferentes, têm a sua utilização nesse trabalho explicada aqui.

1.1.1 SO

Utilizaremos a sigla SO como abreviatura para sistema operacional. Existem diversos sistemas operacionais disponíveis no mercado porém aqui nos concentraremos nos SOs da família Windows e os da família Unix, com ênfase no SO Linux.

1.1.2 Processos

Termo utilizado nesse trabalho com o sentido de processos como normalmente são implementados pelos sistemas operacionais multitarefa em oposição ao termo processo Lua, usado no contexto da linguagem de programação Lua. Descreve um espaço de endereçamento independente, logicamente dividido em regiões de código, dados globais e pilha de execução. Processos diferentes podem até compartilhar a mesma região de código se um for filho do outro, porém a região de memória onde se escrevem os dados e as pilhas são necessariamente independentes. O escalonamento entre processos é feito pelo SO de forma preemptiva, isto é com a perda involuntária do controle do processador. O termo processo Lua que evitamos nesse trabalho é usado em (Ierusalimschy06) se referindo a um *thread* do SO executando uma máquina virtual Lua exclusiva para esse *thread*.

1.1.3 Threads

O termo thread é usado para descrever linhas de execução em diversos modelos. Existem *threads* a nível de usuário, a nível de co-rotina e a nível de núcleo do SO. Em algumas situações esse termo se confunde com processos leves (powell91sunos). No caso dos *threads* a nível de usuário, os *threads* são suportados pela aplicação, sem conhecimento do Kernel e geralmente são implementados por pacotes de rotinas fornecidas por uma determinada biblioteca de uma linguagem.

No contexto da linguagem Lua, um objeto do tipo *thread* representa uma co-rotina(ierusalimschy+05), porém esse termo aqui é utilizado com o sentido de *threads* como normalmente são implementados pelos SOs, ou seja, com o sentido de modelo de execução concorrente com escalonamento preemptivo disponibilizado por chamadas do SO, e controlado pelo SO, onde vários *threads* podem compartilhar os recursos do mesmo processo, obrigatoriamente compartilhando os mesmos segmentos de código e dados mas cada um com a sua pilha de execução independente.

1.1.4 Orientação a eventos

Adya et. al. (adya02cooperative) classifica os modelos de programação quanto ao gerenciamento da pilha e quanto ao gerenciamento de concorrência das tarefas. O gerenciamento de concorrência das tarefas pode ser serial, cooperativo e preemptivo. O gerenciamento da pilha pode ser manual ou automática. O modelo *multi-threads* é classificado como preemptivo com gerenciamento automático de pilha. O modelo orientado a eventos é classificado como cooperativo com gerenciamento manual de pilha, ou seja, um único processo ou thread atende concorrentemente diversas requisições usando apenas uma pilha de execução. Nesse modelo, geralmente o programa em é dividido em trechos com apenas uma requisição de entrada ou saída (E/S). Quando é necessário fazer uma E/S, ela é feita de forma assíncrona. O trecho que executa a continuação da tarefa, juntamente com os dados necessários para a continuação, deve ser registrado em um disparador. A tarefa é interrompida e o controle da CPU é passado para o disparador. Quando a requisição de E/S é completada o disparador é avisado. Depois que o aviso é recebido, o disparador executa a continuação da tarefa. Se a continuação envolver nova operação de E/S, o procedimento de registrar outra continuação é repetido.

O disparador fica bloqueado em uma chamada do sistema (ex: *select*, *poll* ou *WaitForMultipleObjects*) e, à medida que cada operação de E/S

assíncrona se completa, chama a continuação da tarefa correspondente. Como o programador precisa passar explicitamente os dados que serão usados na continuação do processamento, classifica-se o gerenciamento de pilha como manual.

1.1.5 Co-rotina

É uma generalização do conceito de sub-rotina que permite múltiplos pontos de entrada e suspensão (revisitandoco-rotinas). Sempre que uma co-rotina previamente suspensa é invocada, ela volta a executar no ponto onde foi suspensa anteriormente.

Nesse trabalho co-rotinas são interessantes pois cada uma tem sua própria pilha de execução e suas próprias variáveis locais (Ierusalimschy03). Isso permite a implementação de um modelo de concorrência cooperativo sem que o programador precise se preocupar com a persistência dos dados entre invocações da mesma co-rotina, como ocorre na orientação a eventos. O próprio programa controla a troca de contextos de execução, podendo interromper e desviar de um contexto de execução para outro. Posteriormente outro contexto de execução pode comandar a volta para o contexto inicial.

Co-rotinas são usadas nesse trabalho para criar um modelo de programação cooperativo com gerenciamento automático de pilha. Com esse gerenciamento automático, o modelo com co-rotinas fica mais parecido com o modelo multi-thread do que o modelo orientado a eventos. O código fica organizado por tarefas e não por segmentos de tarefas.

O modelo *multi-thread*, que é preemptivo, exige o uso de mecanismos de sincronização para garantir a consistência de dados compartilhados entre as tarefas. Esses mecanismos de sincronização podem ser por exemplo travas, semáforos ou monitores (Monitors). O uso de mecanismos de sincronização não é uma tarefa trivial e seu mau uso pode gerar condições adversas ao prosseguimento das tarefas como por exemplo *deadlocks* ou dados corrompidos (Lea99). Diferente do modelo *multi-thread,* o gerenciamento cooperativo garante a atomicidade dos trechos de código contido entre chamadas explicitas de escalonamento, permitindo o acesso exclusivo aos dados compartilhados e dispensando os mecanismos de sincronização. Por outro lado, no gerenciamento cooperativo uma tarefa pode monopolizar o processamento não executando o código de passagem de controle. Caso isso ocorra, as outras tarefas entram numa condição conhecida como *starvation*, isto é, não recebem sua chance de executar.

Normalmente as co-rotinas são suportadas pelas linguagens de pro-

gramação ou por bibliotecas específicas, enquanto processos e *threads* costumam ser recursos oferecidos pelo SO.

1.1.6 Servidor de aplicação

Nesse trabalho nos referimos a servidor de aplicação como um servidor dedicado a executar aplicações diversas, em oposição a servidores de serviços específicos como servidores de arquivo, correio e impressão.

Os servidores de aplicação oferecem recursos que transcendem a funcionalidade de um servidor web como por exemplo controle de seção, balanço de carga, controle de transações e facilidades de segurança.

Um exemplo largamente difundido de servidor de aplicação é o padrão J2EE implementado por diversos fabricantes (BEA, BES, Geronimo, JBoss, JOnAS, JRun, OracleAS, OrionAS, Pramati, SunAS, WebSphere, WebObjects). Esses servidores de aplicação oferecem uma API Java para executar aplicações e fornecem diversos mecanismos de comunicação e encapsulamento, sendo capazes de servir funcionalidades não só pelo protocolo HTTP como através de outros protocolos.

1.1.7 Sandbox

O termo *sandbox* normalmente é utilizado com dois sentidos diferentes, um no contexto de desenvolvimento de software e outro no contexto de segurança em computadores. No contexto de desenvolvimento de software, o termo é usado para descrever um ambiente de testes, onde se isola o ambiente de produção, normalmente se referindo a infra-estruturas de *hardware* separadas.

No contexto de segurança, *sandbox* é um mecanismo para executar programas com segurança. Normalmente utilizado para executar código não testado, programas de terceiros que não tem verificação, ou de usuários não confiáveis, compartilhando uma mesma infra-estrutura de hardware. O contexto em que aplicamos o termo *sandbox* nesse trabalho é o de segurança

A *sandbox* tipicamente oferece um conjunto de recursos bem controlado para executar os programas visitantes, limitando por exemplo o acesso a disco, memória ou rede. Normalmente se limita a possibilidade de inspecionar o sistema hospedeiro ou de ler dispositivos de entrada. Pode-se dizer que *sandboxes* são um exemplo de virtualização.

Sandboxes podem encapsular desde funções e módulos de um programa, passando pelo encapsulamento de um processo inteiro, até encapsular um SO

completo, controlando seu acesso a recursos de hardware.

Um exemplo de *sandbox* para máquina virtual de código intermediário bastante difundido é o ambiente para execução de *applets* Java em navegadores web. Segundo McGraw et. al. (McGraw96) quando executado em um navegador web o *applet* não pode:

- ler, escrever, remover, renomear arquivos no sistema de arquivos do cliente
- criar e listar diretórios no sistema de arquivos do cliente
- verificar a existência ou obter informações sobre o arquivo, incluindo tamanho, tipo e data de atualização
- criar uma conexão de rede para algum computador que não seja o mesmo de onde o *applet* se originou
- esperar ou aceitar conexões de rede em qualquer porta do sistema cliente
- criar uma janela sem o ícone de inseguro
- obter o nome ou diretório pessoal do usuário
- definir qualquer propriedade do sistema
- executar qualquer programa no sistema cliente
- terminar a execução da máquina virtual
- carregar bibliotecas dinâmicas no sistema cliente
- criar ou manipular qualquer thread que não seja parte do mesmo grupo de *threads* do *applet*
- criar um *Class Loader* (carregador de classes) ou um *Security Manager* (gerenciador de segurança)
- especificar qualquer função de controle de rede
- definir classes que sejam parte de pacotes do sistema cliente.

1.1.8 Interferência entre tarefas

Diversos tipos de comunicação entre tarefas são desejáveis, como troca de dados e sinalizações. Porém muitos dos recursos utilizados para essas comunicações podem resultar em interferências indesejáveis.

Como interferência indesejada podemos exemplificar: impedir que uma tarefa receba uma fatia de tempo para processamento, alterar indevidamente a entrada ou a saída do processamento, alterar a lógica do código sendo executado, acessar dados que deveriam ser restritos àquele código e deixar resquícios, como arquivos, *locks* ou processos em execução que não terão mais serventia para nenhum outro sistema que venha a executar naquele ambiente.

No caso dos servidores de aplicação, a execução de códigos com finalidades diversas (muitas vezes sem nenhum relacionamento entre elas) em ambientes compartilhados, demandam um ambiente de execução protegido. Essa proteção é desejável tanto para impedir que uma aplicação interfira de maneira indesejada com a máquina e com as outras aplicações que executam nessa máquina e vice versa.

1.2 Objetivos e Organização

Já existem diversas iniciativas de utilização da linguagem Lua em aplicações web, as quais oferecem um contexto interessante para realizar medidas de desempenho de modelos de concorrência e de isolamento de tarefas por sandbox. O objetivo desse trabalho é quantificar o peso de diferentes modelos de concorrência e modelos de *sandbox* em servidores web, especialmente no que se refere à linguagem Lua. Nos primeiros capítulos descrevemos como chegamos ao sistema a ser testado. Para isso, no capítulo 2 discutimos os diversos modelos de programação possíveis para concorrência, *sandbox*, persistência e E/S e como esses modelos se relacionam entre si. No capítulo 3 descrevemos as tecnologias utilizadas como infra-estrutura nesse trabalho e no capítulo 4 descrevemos os sistemas que foram implementados ou modificados para a realização desse trabalho.

Depois de definir e implementar sistemas representativos dos principais modelos de nosso interesse, elaboramos e realizamos testes de desempenho nesses sistemas. No capítulo 5 descrevemos como chegamos aos fatores que mais influenciam nos nossos resultados, quantificamos os valores a serem testados e descrevemos como diminuir a influência de blocos de variáveis que normalmente estão fora do nosso controle. No capítulo 6 descrevemos quais eram as nossas expectativas de resultados antes da realização das medidas. No

capítulo 7 relatamos os resultados obtidos nas medidas e comparamos com o que esperávamos antes. No capítulo 8 fazemos algumas considerações finais.