2
Modelos de Implementação

Os modelos de concorrência definem como uma aplicação atende às requisições concorrentes. Os modelos de *sandboxes* definem como o ambiente das aplicações são criados. Os modelos de concorrência e os modelos de *sandboxes* podem ser considerados ortogonais. Já a escolha de modelos de E/S e de persistência não podem ser considerados independentes dos modelos de concorrência e *sandbox*.

Nesse capítulo mostramos modelos de concorrência e sandboxes usados em uma variedade de sistemas, selecionamos os de maior interesse e descrevemos como integrá-los à linguagem Lua. Discutimos também o impacto do uso desses modelos sobre modelos que não são o foco principal desse trabalho.

2.1
Modelos de sandbox

Diversos sistemas podem ser citados como exemplo de uso de *sandboxes*. A seguir destacamos alguns deles.

Goldberg et. al. (goldberg96secure) citam o trabalho de Wahbe et. al. (wahbe93efficient) como o primeiro a introduzir o termo *sandboxing* para descrever o confinamento de uma aplicação em um ambiente controlado. Wahbe et. al. descrevem mecanismos de compilação e carga de módulos binários para isolamento de módulos de software que colaboram entre si.

O sistema Janus (goldberg96secure) monitora as chamadas ao SO, permitindo ou bloqueando o seu procedimento de acordo com arquivos de regras que levam em conta os argumentos recebidos e retornados pela chamada.

Keahey et. al. (KeaheyDF04) descrevem 3 tipos de tecnologia para criar ambientes virtuais dinâmicos que sirvam para códigos remotos em uma grade computacional: contas Unix, servidores virtuais no mesmo kernel Linux e emulação de computadores completos através do software VMware. Nas contas Unix são usados os recursos do controle de usuários do SO para controlar o ambiente. No servidor virtual, o SO é estendido para se comportar como se fosse vários SOs. Nos computadores virtuais, um sistema emula um computador

hospedeiro completo onde um SO pode ser executado, controlando o acesso ao hardware real, podendo multiplexar o uso do hardware para vários SOs.

Prevelakis et. al. (ps01) apresenta o sistema FMAC para criar automaticamente um sistema de arquivos somente com os arquivos necessários para uma execução segura do sistema e depois limitar o acesso da aplicação somente a esse sistema de arquivos. O FMAC substitui um serviço oferecido pelo SO por um com controle.

Esses modelos de *sandbox* presam a generalidade de tecnologia e linguagem, sendo muito voltados para a execução de código de máquina binário. No caso do VMware, chega-se ao ponto de criar uma máquina virtual capaz de executar um SO sem modificações.

No modelo de *sandbox* dos *applets* Java apresentado na seção 1.1.7 são utilizados basicamente mecanismos da própria linguagem.

De forma resumida, os diversos *sandboxes* de alguma forma conseguem:

- criar separação entre domínios
- substituir funcionalidades por similares controladas ou mais restritas
- remover funcionalidades
- prover transferência de dados entre domínios.

Segundo os exemplos acima, o domínio independente pode se parecer com um computador, com um programa, com um SO, com um sistema de arquivos, com uma área de memória ou com uma máquina virtual . As funcionalidades podem ser dispositivos de hardware, chamadas de SO, arquivos, bibliotecas ou objetos de uma linguagem.

A comunicação entre *sandboxes* pode ser feita através da emulação de uma rede, de mecanismos de IPC (*Inter Process Comunication*) fornecidos pelo SO, por mecanismos da linguagem, ou extensões à linguagem. Algumas das soluções de *sandbox* apresentadas são bastante genéricas e independentes de linguagem, porém nesse trabalho nos concentramos em aplicações Lua. Nossa maior preocupação na escolha dos modelos de *sandbox* foi como esses modelos implementam a separação de domínios. A forma como cada modelo faz a comunicação entre domínios foi escolhida de acordo com a natureza da implementação dos mecanismos de criação de domínios.

Para se resguardar contra problemas de segurança, normalmente se procura oferecer em um ambiente controlado somente funcionalidades realmente necessárias para a execução da tarefa desejada. Porém essa abordagem faz com que cada aplicação tenha seus requisitos mínimos. Como não nos fixamos em aplicações específicas, não entramos no âmbito de definir e preparar ambientes

com funcionalidades controladas. Não tivemos preocupação de efetivamente remover ou substituir funcionalidades porém, para manipular as funcionalidades na linguagem Lua, em qualquer modelo, pode-se valer do fato de funções serem objetos de primeira classe, então as funcionalidades podem ser eliminadas ou redefinidas como simples atribuições a variáveis do sistema.

Nesse trabalho apresentaremos modelos que criam novos domínios criando novas máquinas virtuais Lua no mesmo processo, criando novos processos e criando novos espaços de variáveis usando recursos da própria linguagem Lua.

2.2 Modelos de Concorrência

Nesse trabalho estamos preocupados com questões de concorrência em um servidor web e tarefas disparadas por ele. Procuramos não abranger a questão de tarefas externas que possam vir a concorrer com as tarefas atreladas ao servidor web.

J. Hu et. al.. (hu99jaws) descrevem 4 modelos de concorrência que se aplicam ao contexto dos servidores web: *thread* por requisição, *thread* por sessão, *pool* de *threads* e *thread* único. No *thread* por requisição, cada nova requisição dispara um novo *thread*. No *thread* por sessão, cada conexão, que pode conter múltiplas requisições, dispara um novo *thread*. No *pool* de *threads*, um número pré-determinado de *threads* é previamente disparado e à medida que novas requisições chegam, são colocadas em uma fila. Esses *threads* retiram as requisições da fila e atendem essas requisições. No *thread* único, uma única seqüência de execução atende a todas as requisições. Nesse caso para aumentar o desempenho desse modelo, não se atende as requisições seqüencialmente com chamadas de E/S síncronas, o atendimento é multiplexado usando mecanismos como a orientação a eventos. Dependendo da sofisticação da implementação, o desempenho do modelo *thread* único em máquinas com um único processador pode ser melhor que os outros modelos *multi-threaded* (hu97measuring).

Complementando os modelos propostos pode-se considerar a possibilidade da utilização de co-rotinas e de processos.

A. Moura (revisitandoco-rotinas) propõe o escalonamento de tarefas, com gerência cooperativa baseada em co-rotinas completas e assimétricas, evitando-se assim a preempção. Esse mesmo trabalho sugere a utilização de uma biblioteca auxiliar de funções de E/S para evitar que todos as tarefas fiquem bloqueadas na entrada ou saída de uma única tarefa. Essa biblioteca deve utilizar os mecanismos oferecidos por diversos SO de *timeout* nas chamadas de E/S e em todo caso de *timeout* deve forçar o escalonamento das tarefas. Caso

todas as tarefas estejam esperando por operações de E/S, o escalonador deve se bloquear até que um dos recursos envolvidos mude de estado.

Uma forma de minimizar a sobrecarga gerada pela criação de mecanismos como *threads* e processos é a criação de *pools*. Se considerarmos que se pode utilizar *pools* de co-rotinas e *pools* de processos, podemos considerar que os *pools* são a combinação de um modelo seqüencial (sem concorrência) com um modelo com concorrência.

Destacamos os seguintes modelos básicos com os quais podemos criar os *pools* ou outros modelos mais complexos de concorrência:

- seqüencial (servidor TCP simples),
- com co-rotinas,
- multi-thread
- multi-processos

Exemplos de formas combinadas seriam:

- Em uma máquina com dois processadores, para melhorar a utilização dos processadores, pode-se disparar um *pool* de 2 *threads* onde cada *thread* atenderia as requisições atribuídas a ele de forma seqüencial.
- Um *pool* de *threads* poderia atender mais requisições simultaneamente se cada um utilizasse um modelo com co-rotinas ao invés de seqüencial.

2.2.1 Comparação dos modelos de concorrência

Uma vez que haja a possibilidade de realizar tarefas concorrentemente na mesma máquina, diversos modelos de concorrencia podem ser utilizados. Cada modelo tem suas proprias características e implicações. Nessa seção descrevemos os modelos utilizados nesse trabalho.

Modelo seqüencial

O modelo seqüencial, ou seja, que trata as requisições uma a uma mesmo que clientes estejam competindo para serem atendidos, é o mais simples de se programar. É implementado como um *loop* que aceita uma nova conexão, trata de respondê-la e só então reinicia o *loop* para atender outra conexão. Todos os dados são acessados por apenas uma tarefa.

O problema de executar tarefas seqüencialmente é que isso diminui a interatividade dos sistemas, o que em alguns casos pode ser crítico. Por exemplo: Em um servidor web seqüencial a requisição é processada e enviada

para o cliente toda de uma vez. Se existirem 100 requisições na fila esperando para serem atendidas, a última terá que esperar o servidor terminar de processar as outras 99 para começar a ser atendida. Um usuário que solicite a um servidor web um texto muito longo, provavelmente ficará muito mais satisfeito se receber imediatamente o começo do texto e for recebendo o resto do texto enquanto ele lê o começo do que esperar um longo tempo e receber o texto de uma só vez. Isso porque o servidor teve que atender diversas outras requisições por completo antes. A falta de interatividade pode até criar a sensação de dúvida se o sistema está funcionando. Por outro lado, para um robô de indexação de máquina de busca, que tipicamente realiza suas tarefas em *batch*, não faz diferença se a requisição é respondida toda de uma vez ou em várias rajadas. Provavelmente ele só processará o conteúdo da página quando ela for totalmente recebida.

Modelo de concorrência com co-rotinas

O modelo de concorrência com co-rotinas é o que exige menos funcionalidades do SO e do *hardware*, já que o escalonamento é feito pelo próprio programa e não depende de interrupções de *hardware* ou *software*. Por essa independencia, é indicado para dispositivos com recursos limitados como atualmente são alguns sistemas embarcados, celulares e PDAs. Para o programador, os momentos de escalonamento são explícitos, então a troca de dados por memória compartilhada é bastante simples quando comparado com modelos preemptivos.

Um trecho de código que precisa acessar dados em memória compartilhada por diversas tarefas é chamado de região crítica. Esse trecho recebe esse nome porque se duas ou mais tarefas alteram simultaneamente um trecho de memória, ou ocorre uma leitura no meio de uma alteração, o resultado final pode ser inesperado. Para garantir a consistência dos dados usando co-rotinas como modelo de concorrência, basta não intercalar uma região crítica com uma tranferência de controle entre co-rotinas. Assim está garantido que somente uma tarefa acessa a região de memória por vez.

A questão da interatividade pode ser regulada pelo próprio programador, escolhendo momentos adequados para realizar o escalonamento. Porém se o programador não tomar cuidado, pode criar situações onde o escalonamento nunca aconteça, impedindo que outras tarefas executem. A condição onde uma tarefa não recebe oportunidade de executar é conhecida como *starvation*.

Modelo de concorrência multi-processos

O modelo de concorrência multi-processos é a solução mais tradicional para endereçar a questão da interatividade. O SO controla o tempo que cada processo executa, interrompendo o processo sempre que ele chega ao seu limite de tempo, e retomando o processamento do ponto onde parou quando for a vez do processo novamente, permitindo que todos os processos executem seqüencialmente, cada um por um período de tempo. Essa interrupção involuntária do processo é conhecida como preempção e garante que um processo não monopolize a CPU caso haja processos com prioridades de execução semelhantes, evitando que um processo entre em *starvation* (ntscheduler1, ntscheduler2).

Normalmente os processos quando são criados não compartilham dados com outros processos. Diversos mecanismos costumam ser oferecidos pelo SO para isso e são conhecidos como IPC. Exemplos de IPC são: envio de sinais, troca de mensagens, *streams* de dados e até mesmo memória compartilhada.

O uso de memória compartilhada exige a utilização de mecanismos de sincronização para evitar que regiões críticas de dois processos se intercalem. O uso desses mecanismos é essencial para garantir a corretude de um sistema ao longo de diversas execuções. O problema é que a falta ou mau uso desses mecanismos muitas vezes só geram erros depois de muitas execuções e que por isso mesmo são difíceis de serem identificados e consertados. Os próprios mecanismos de sincronização aumentam um pouco o processamento necessário para executar as regiões criticas, porém é comum os programadores criarem travas abrangendo trechos da aplicação que não necessariamente precisariam estar protegidos, afetando negativamente o desempenho da aplicação.

Aparentemente, utilizar outros mecanismos de IPC evita o uso de mecanismos de sincronização, porém o uso desses IPCs apenas abstraem que alguma forma de sincronização está sendo usada. Para permitir que duas tarefas, que podem ter suas execuções intercaladas de forma imprevisível possam manipular dados em comum é necessário utilizar mecanismos de sincronização.

Nesse trabalho utilizamos processos para criar uma nova máquina virtual Lua independente da máquina virtual onde foi criada. No caso do nosso servidor web, o modelo multi-processos é o que esperamos apresentar o pior desempenho. Apesar disso é o modelo que oferece o melhor isolamento, pois cada servidor roda em uma máquina virtual Lua totalmente independente, minimizando assim a possibilidade de comunicações. Esse modelo é o mais indicado para aplicações que demandam maior robustez e dispõem de capacidade de processamento sobrando.

Modelo de concorrência multi-thread

O modelo de concorrência multi-thread, do ponto de vista da concorrência, é muito parecido com o modelo multi-processos, pois ambos utilizam escalonamento preemptivo. Em algumas implementações, a inicialização de um *thread* é mais rápida que a de um processo, por isso *threads* também são conhecidos como processos leves. A grande diferença é que um *thread* filho compartilha a área de dados com o pai enquanto o processo recebe uma área de dados exclusiva.

Como *threads* compartilham memória, são necessários mecanismos de sincronização complementando a API de *threads*. Apesar de existirem abstrações simplificadoras como travas, monitores e semáforos, o mau uso desses mecanismos, muitas vezes, continua sendo difícil de ser identificado e consertado. Além disso, assim como no modelo multi-processo, também pode-se degradar o desempenho devido ao tempo de espera por sincronizações.

Também com no caso dos processos, utilizamos *threads* nesse trabalho para criar máquinas virtuais Lua independentes daquela que as criou.

2.3 Modelos de persistência

O estado envolvido em uma execução remota pode ser criado e destruído de acordo com uma série de critérios. Alguns desses critérios podem ser impostos pelo sistema que dá suporte a essa execução remota e outros pela aplicação. O estado pode ser criado a cada requisição e destruído ao fim dessa requisição ou até resistir a paradas e reinicializações do sistema.

Web services (WebServicesHandbook), por exemplo, oferecem três modelos de persistência:

1. Aplicação: o serviço é criado na inicialização do servidor e permanece ativo enquanto o servidor estiver executando.

2. Requisição: o serviço é instanciado quando uma mensagem é recebida e eliminado quando a requisição termina de ser respondida.

3. Sessão: o tempo de vida do serviço dura o período que dura a sessão da camada de transporte.

Em modelos de objetos remotos como CORBA, o objeto remoto pode ser criado ou destruído pela aplicação em qualquer momento. No CORBA, podem ser criados inclusive esquemas que criam os objetos sob demanda, ou objetos que se comportam como um objeto que responde por todos os outros. Cada objeto remoto recebe um endereço único. Por isso é possível fazer com que

um objeto em uma execução atual responda como se fosse um objeto criado em uma execução anterior, dando a impressão ao cliente que o objeto resistiu à parada do sistema, bastando para isso fazer que ele responda no endereço utilizado anteriormente (POA98).

Fazendo uma classificação bastante abrangente, podemos enumerar os seguintes modelos de persistência no servidor:

1. sem persistência: o estado resiste apenas a uma requisição

2. persistência a diversas requisições na mesma execução: o estado pode ser retomado desde que na mesma execução do serviço que o está armazenando

3. persistência entre execuções: o estado é armazenado e reconstruído entre uma execução e outra do serviço que o está armazenando.

No caso de aplicações web, quando temos persistência, é comum o cliente enxergar 2 níveis de persistência.

1. persistência do estado global da aplicação: o estado da aplicação normalmente fica armazenado em memória (persiste na mesma execução do servidor), arquivos e em um banco de dados (persistem entre execuções do servidor).

2. persistência do estado da seção de um usuário: o estado normalmente fica armazenado em memória, como no framework J2EE (persiste na mesma execução do servidor), ou em arquivos temporários, como na linguagem PHP para web (Niederauer04), e em *cookies* ou URLs que trafegam junto com os dados da conexão e ficam armazenados no cliente (persistem entre execuções do servidor).

2.3.1 Relação entre modelo de persistência e concorrência

Nesse trabalho não estudamos o desempenho dos modelos de persistência, porém a escolha do modelo de concorrência é muitas vezes influenciado pelos modelos de persistência desejados no sistema.

A persistência utilizando arquivos em disco e banco de dados é ortogonal ao modelo de concorrência. Porém a persistência em memória é fortemente influenciada pelo modelo de concorrência. Nos modelos onde a tarefa é criada e destruída por requisição ou seção da camada de transporte não é possível armazenar o estado da aplicação diretamente na memória exclusiva da tarefa,

pois a memória é apagada sempre que a tarefa é finalizada. No caso da utilização de um *pool* de tarefas, armazenar o estado em uma das tarefas do *pool* também não resolve, pois normalmente qualquer uma das tarefas do *pool* pode atender a requisição. Quando se usa um *pool*, em função do acaso, duas requisições podem ser atendidas pela mesma tarefa e encontrar o mesmo estado ou serem atendidas por tarefas diferentes e não encontrarem o mesmo estado. Em ambos os casos, para manter persistência em memória é necessário a utilização de mecanismos de memória compartilhada. Dependendo do modelo de concorrência, o uso de memória compartilhada é facilitado ou não. Para o caso de mecanismos do SO, *threads* compartilham memória desde sua criação, enquanto processos, além de demandarem a utilização de chamadas de sistema mais complexas, demandam mecanismos de comunicação auxiliares para encontrar a memória compartilhada. Ambos os modelos demandam a utilização de mecanismos de sincronização para garantir a integridade dos dados armazenados na memória compartilhada.

Para o caso mais específico da linguagem Lua, o mecanismo padrão de concorrência é a utilização de co-rotinas para criar escalonamento colaborativo. A memória é naturalmente compartilhada entre as tarefas e tipicamente não existe a necessidade de mecanismos de sincronização para garantir a integridade dos dados.

Como a linguagem Lua oferece mecanismos para ser estendida, é possível utilizar os mecanismos do SO para implementar novos modelos de concorrência para a linguagem. A máquina virtual Lua não está preparada para suportar concorrência preemptiva[1], então pode-se utilizar o mecanismo de *threads* do SO para criar máquinas virtuais concorrentes no mesmo processo do SO e facilitar um pouco a implementação de comunicação entre essas máquinas virtuais. É possível simplesmente utilizar máquinas virtuais em processos separados e utilizar métodos de IPC oferecidos pelo SO. Ambos os casos se comportam de maneira similar quando pensamos na possibilidade de utilização de memória compartilhada.

2.3.2 Relação entre modelo de persistência e sandbox

Nesse trabalho nos limitaremos a testar o desempenho sem persistência, pois optamos por nos limitar a levantar o custo computacional de criar novos domínios para aplicações. Porém um caminho natural para o alto desempenho

[1]Existe um remendo para a máquina virtual Lua chamado de LuaThread(luathread). Essa variação da linguagem inclui chamadas de sincronização na máquina virtual Lua e oferece uma API com chamadas de sincronização para programação concorrente preemptiva com memória compartilhada baseada nos threads do SO

na persistência da aplicação é criar mecanismos para a persistência do ambiente da aplicação. Uma *sandbox* pode encapsular o ambiente da aplicação, fazendo a *sandbox* persistir, ela passa a ser sinônimo da aplicação.

A criação do ambiente de execução pode demandar diversas tarefas custosas, como por exemplo: carga da sua parte executável, carga de dados iniciais, abertura de conexão com banco de dados ou autenticação em outros sistemas. O conjunto dessas tarefas tendem a ser um procedimento custoso. Se uma requisição puder aproveitar um ambiente criado em requisições anteriores, provavelmente demandará um processamento menor que o necessário para criar um novo ambiente. O reaproveitamento do ambiente certamente demanda a coleta de lixo de objetos criados exclusivamente para uma requisição mas essa coleta de lixo parcial do ambiente tende a ser menos custosa que a criação total do ambiente.

Um caminho natural para o alto desempenho na persistência da aplicação é criar mecanismos para a persistência do ambiente da aplicação. Uma *sandbox* pode encapsular o ambiente da aplicação, fazendo a *sandbox* persistir, ela passa a ser sinônimo da aplicação persistente. Porém nesse trabalho nos limitaremos a testar o desempenho sem persistência, pois optamos por nos limitar a levantar o custo computacional de criar novos domínios para aplicações.

2.4 Modelos de E/S

Nos SOs multitarefa modernos toda E/S é gerenciada pelo próprio SO. Para evitar que se desperdice tempo de processamento simplesmente esperando que a E/S se conclua, toda E/S é executada de forma assíncrona (não bloqueante) pelo SO.

Apesar das operações de E/S serem internamente assíncronas, para as aplicações, o SO pode oferecer diferentes níveis de sincronismo (hu99jaws). Pode oferecer chamadas síncronas (bloqueantes), assíncronas, onde ele se encarrega da transferência de um grande bloco de informação sem interferência adicional da aplicação que a solicitou, e finalmente, assíncrona, onde a aplicação cuida dos *buffers* e do fluxo dos dados.

Nas chamadas síncronas, a tarefa fica bloqueada esperando que a E/S se complete e os dados estejam disponíveis, enquanto o SO permite que a CPU seja utilizada por outras tarefas. Chamadas de E/S bloqueantes são as mais comumente encontradas para acessar os mais diversos dispositivos e certamente facilitam bastante a vida do programador.

2.4.1

Relação entre modelo de E/S e concorrência

Normalmente, quando se usa um modelo de concorrência preemptivo, também se utilizam chamadas de E/S síncronas. O uso de chamadas síncronas se dá porque se espera que, caso elas não sejam prontamente atendidas, o escalonador passe a vez para uma outra tarefa e só retome a execução da tarefa interrompida em E/S quando essa chamada puder ser atendida. E/S assíncrona pode ser usada caso essa E/S não seja crítica e possa ser realizada em qualquer outro momento, ou se existirem outras ações mais críticas a serem executadas que não podem ficar esperando E/S.

E/S assíncrono é utilizado nos modelos de concorrência colaborativos como na orientação a eventos e no modelo multi-corrotina sendo inclusive as chamadas de E/S assíncronas um ponto natural de escalonamento.

No modelo seqüencial, somente a E/S síncrona faz sentido. A tarefa não tem a opção de fazer mais nada caso os mecanismos de E/S não possam atender a tarefa prontamente.