3
Tecnologias envolvidas

Foram utilizadas, nesse trabalho, ferramentas de uso público que envolvem a linguagem de programação Lua, ao Projeto Kepler e servidores web em geral. Nesse capítulo descrevemos brevemente essas tecnologias.

3.1
Lua

Lua (ry96-4, Ierusalimschy03, Ierusalimschy06) é uma linguagem de programação interpretada com uma máquina virtual toda escrita em ANSI C, o que a torna altamente portável entre plataformas. Foi desenvolvida inicialmente para ser embutida em aplicações escritas em outras linguagens, de forma que pudesse estendê-las. Pode ser usada para descrever dados, definir configurações e comportamentos do núcleo da aplicação escrito em outra linguagem. Como linguagem concebida para extensão, Lua também fornece mecanismos para ser estendida. Com a evolução da linguagem, Lua passou a ser vista também como uma linguagem de *script* auto-suficiente e passou a ser usada para desenvolver diversas aplicações, mantendo as características de oferecer recursos poderosos para descrever dados e ser facilmente extensível.

Esse trabalho desperta um especial interesse na forma com que Lua permite manipular os diversos objetos que formam o estado do programa que está executando. O estado de um programa Lua simples consiste no seu ambiente global e sua pilha de execução. O ambiente global de Lua armazena não apenas estruturas de dados mas também funções, que são objetos de primeira classe da linguagem. O ambiente global é uma tabela Lua, que por sua vez também é um objeto de primeira classe e pode ser facilmente manipulado. O mecanismo de co-rotinas oferecido nativamente em Lua permite o chaveamento entre pilhas de execução. Uma co-rotina também é um objeto de primeira classe da linguagem.

Manipulando objetos de primeira classe é possível criar, na mesma máquina virtual Lua, um ambiente multitarefa, isto é: deixar programas suspensos, implementar modelos de concorrência e executar programas Lua de forma independente (ou com comunicação regulada).

A versão da linguagem Lua utilizada nesse trabalho é a 5.0.2.

3.2 Kepler

O Projeto Kepler (kepler) se propõe a disponibilizar uma plataforma de desenvolvimento para a web baseada na linguagem Lua, mantendo a característica multi-plataforma de Lua. Esse projeto desenvolveu uma arquitetura onde aplicações para a Web podem ser escritas utilizando uma API única denominada CGILua. Seguindo a linha multi-plataforma, Kepler disponibiliza disparadores de interpretadores Lua para as tecnologias web mais difundidas, entre elas: CGI, Módulos do Apache, módulos no IIS (*Internet Information Server*), FastCGI e Servlet Java. O desempenho dessas tecnologias é analisada em (Killelea98). Além dos disparadores para tecnologias amplamente difundidas, o Kepler também oferece um servidor web escrito em Lua, chamado Xavante. Esse servidor também permite a execução de *scripts* Lua, entre eles os que usam a API CGILua.

3.2.1 Sandboxes

O uso de *sandboxes* faz parte da arquitetura de alguns disparadores do Kepler. Os módulos que suportam essas funcionalidades são utilizados sem modificações nesse trabalho. Os *sandboxes* são implementados como módulos Lua e denominados Venv e Rings.

Ambas implementações de *sandbox* inicialmente foram desenvolvidas para limpar os vestígios da execução de uma tarefa ao seu fim. O objetivo era simular um ambiente semelhante ao padrão CGI (*Common Gateway Interface*) utilizado para executar aplicações em servidores web. Nesse padrão, quando o servidor web recebe uma requisição que aponte para um arquivo executável CGI, ele executa o arquivo em um novo processo, passando dados por variáveis de ambiente e pela entrada de dados padrão do processo. Os dados são recebidos da saída padrão do processo e repassados ao cliente.

Como o arquivo executa em um processo separado ao processo do servidor web, ele usufrui dos mecanismos que o SO oferece aos processos. Cada processo executa em seu próprio domínio. Quando o processo é finalizado, o SO limpa os vestígios da execução desalocando todos os recursos alocados para o domínio. O mecanismo de processos fornece um certo nível de isolamento entre tarefas que dificulta comunicações e interferências. Estas só são permitidas através de mecanismos oferecidos pelo próprio SO, como por exemplo sinais, mensagens e memória compartilhada.

Ambas as *sandboxes* do projeto Kepler têm uma semântica parecida a um processo do SO, porém, além de serem totalmente independentes do SO, são criadas dentro do próprio processo que as solicitou. Em comum notamos que elas oferecem uma separação entre domínios que só pode ser ultrapassada usando mecanismos específicos das *sandboxes* e que, quando finalizados os domínios, os vestígios de sua execução são removidos da memória. De diferente notamos que no caso dos processos (a princípio) a maioria dos mecanismos de comunicação, chamados de IPC, estão desabilitados, porém quando habilitados permitem vários tipos de comunicação com qualquer processo que esteja executando, inclusive em outras máquinas. No caso das *sandboxes* do projeto Kepler, os mecanismos de comunicação já começam habilitados quando um novo domínio é criado e só permitem a comunicação entre domínios pai e filho. Os mecanismos de comunicação basicos são poderosos o suficiente para permitir que a partir de um domínio se possa executar funções no outro domínio.

Os mecanismos de comunicação iniciais disponíveis nas *sandboxes* do projeto Kepler deixam a questão da segurança totalmente aberta. Para implementar segurança entre domínios, a idéia é criar mecanismos de comunicação que encapsulem os mecanismos iniciais, oferecendo somente as funcionalidades desejadas e depois dessa criação, apagar os mecanismos iniciais, limitando o acesso ao outro domínio.

Outra característica de segurança comum em *sandboxes* é o acesso controlado a recursos da máquina. Normalmente o SO controla o acesso que um processo tem sobre os recursos da máquina através de listas de controle chamadas de ACL (*Access Control List*) que relacionam usuários, grupos de usuários, recursos e o tipo de acesso que um tem ao outro. No caso de *sandboxes* em Lua, esse tipo de controle pode ser realizado redefinindo ou eliminando funções, já que funções são objetos de primeira classe da linguagem. Para embutir o controle de acesso em uma função, basta atribuir uma nova função ao nome da função original. Essa nova função pode, por exemplo, ser um *proxy* para a função antiga que antes de chamar a função antiga verifica uma ACL. Para remover uma funcionalidade basta remover da aplicação o nome da função original.

Uma deficiência dessas *sandboxes* é que elas não oferecem recursos para garantir a separação de domínios em bibliotecas C. Uma variável global dentro de uma biblioteca C é a mesma para todos os ambientes que a estiverem chamando. Além disso, em C é possível encontrar referências para os outros domínios e modificar domínios que deveriam estar isolados.

A principal diferença entre os dois modelos de *sandbox* é que no Venv

todos os domínios executam na mesma máquina virtual, já a Rings cria uma máquina virtual para cada domínio.

Venv

O Venv é uma *sandbox* que oferece separação de domínios por criação de ambientes globais separados, porém deixa vazar por herança para o domínio filho o domínio do pai, de tal modo que os objetos do domínio do pai podem ser acessados mas não podem ser modificados a partir do domínio do filho, somente podem ser substituídos por objetos locais no domínio filho. Esse mecanismo tem a semântica parecida com o mecanismo de clonagem de processos do Linux que mapeia as páginas da memória virtual do processo pai para o filho e só cópia as páginas para uma página própria quando vai realizar uma operação de escrita.

A API do módulo Venv oferece apenas uma função:

`venv(func)` Cria um novo ambiente e retorna uma função que executa a função `func`, no novo ambiente.

O mecanismo Lua básico que permite a implementação do Venv é a função *setfenv* (*set function environment*)(Ierusalimschy03). O Venv encapsula uma função em uma nova tabela global que acessa por herança a tabela global do ambiente que criou a função (ambiente global pai). No disparo do ambiente filho, algumas funções e tabelas (que manipulam a carga de módulos) da nova tabela global são recriadas. Depois que a função encapsulada pelo Venv é disparada, qualquer leitura a objetos globais que existirem na nova tabela global são buscados na tabela global pai. Caso existam na tabela global pai, é feita uma cópia para a nova tabela global. Para evitar que um objeto destruído na nova tabela global volte a herdar o valor da tabela global pai, o Venv guarda o nome dos objetos que alguma vez existiram nos dois ambientes, evitando que a herança volte a ocorrer.

Quando se executa uma função herdada do ambiente pai, mesmo a função tendo sido copiada para a nova tabela global, ela executa acessando a tabela global pai onde a função foi criada. Então os valores globais que a função altera são os valores globais do pai e não do filho que a chamou. Funções herdadas são o mecanismo de comunicação oferecido pelo Venv. Para evitar que se possa acessar mais que o desejado depois da criação do ambiente, somente funções que executam tarefas necessárias no novo ambiente devem ser deixadas. As outras devem ser removidas do ambiente.

Quanto ao acesso de ambientes a partir de bibliotecas C, pouco se pode fazer para proteger os ambientes de acesso indevido vindo de outro ambiente.

Como todos os ambientes estão na mesma máquina virtual, a própria API da máquina virtual oferece recursos para encontrar e alterar outros ambientes.

Rings

O Rings cria novos domínios instanciando novas máquinas virtuais. Ele permite a comunicação entre máquinas virtuais pai e filha através do envio de trechos executáveis de código Lua. O Rings envia trechos de código entre diferentes máquinas virtuais no mesmo processo, compila a *string* e a executa. Esse modelo de comunicação baseado em transferência de código é parecido com o modelo do ALua (elsevier). O ALua oferece um mecanismo para enviar por TCP *strings* que são trechos executáveis de código a agentes remotos (que embutem uma máquina virtual Lua). Um agente envia asincronamente códigos executáveis para outro agente. Quando um agente recebe um código desses, compila a *string* e a executa. Diferentemente do ALua, a execução remota de código é síncrona e além disso a API do Rings além de receber como argumento uma *string* com um trecho de código, permite a passagem de objetos Lua como argumentos adicionais, além de receber objetos Lua como retorno.

A API do módulo Rings é a seguinte:

`rings.new()` cria e retorna uma nova máquina virtual Lua escrava (`state`). Nessa nova máquina virtual, cria a função `remotedostring()`

`state:close()` destrói a máquina virtual.

`state:dostring(string, ...)` Executa a `string` na máquina virtual escrava. Os argumentos são armazenados na tabela `arg` da máquina virtual escrava (`state`). Retorna um booleano indicando o status da operação seguido dos valores retornados ou da mensagem de erro.

`remotedostring(string, ...)` Função criada pela `rings.new()` na máquina virtual Lua escrava. Executa a `string` na máquina virtual mestre. Os argumentos são armazenados na tabela `arg` da máquina virtual mestre. Retorna um booleano indicando o status da operação seguido dos valores retornados ou da mensagem de erro.

Os objetos Lua passados entre domínios podem ser: booleanos, números, *strings* e *userdata* (ponteiros controlados pela máquina virtual Lua). *O userdata* é convertido para *lightuserdata* (ponteiros sem o controle da máquina virtual). Esses são objetos Lua que podem ser copiados com uma simples cópia de

área de memória, já que as áreas de dados das máquinas virtuais são independentes. Não existe provisão na linguagem para manter referências consistentes entre máquinas virtuais. Referenciar um objeto diretamente no outro estado não é previsto pela API Lua. Um dos problemas mais simples que utilizar um objeto criado em outro estado pode causar é que esse objeto pode ser coletado pelo coletor de lixo da outra máquina virtual e a referência mantida na outra máquina virtual passa a apontar para uma região de memória inválida. Podemos exemplificar com um objeto LuaSocket. Ele tem partes em Lua e partes em objetos C, por isso não pode ser copiado entre máquinas virtuais, a não ser que a função da parte em C do módulo que faz a cópia entre os domínios conheça a estrutura e replique os objetos Lua e C que criam o LuaSocket na outra máquina virtual. Outro exemplo é uma função que armazene algum dos argumentos recebidos numa tabela global. Se um outro estado chama essa função com um dos seus objetos, a tabela global passa a armazenar uma referência pertencente a outra máquina virtual. Qualquer coisa pode acontecer com essa referência, até desaparecer porque uma das máquinas virtuais concluiu que ela devia ser coletada.

Tenta-se diminuir o impacto do processamento para compilar uma nova *string* a cada *dostring()* fazendo uso de um *cache* de todas as *strings* de código que já foram compiladas. Assim, na segunda vez que se tenta executar uma mesma *string*, o *bytecode* previamente compilado é reutilizado. Para melhorar a eficácia do *cache*, o *string* de código compilado é transformada em uma função, que recebe os argumentos adicionais em uma tabela de forma análoga a funções com número de argumentos variáveis. Assim é possível criar *templates* de *strings* de código que recebem valores como argumentos.

Os mecanismos de *dostring* do módulo Rings são ainda mais poderosos que a herança de funções do Venv. Eles permitem que se defina qualquer objeto ou se execute qualquer função ou trecho de código no outro ambiente. A etapa de encapsulamento e eliminação da função *remotedostring()* é ainda mais crítica para se proteger o ambiente.

Quanto ao acesso de ambientes a partir de bibliotecas C, o acesso indevido é mais complicado que no modelo Venv, porém também é possível. Supondo que na criação da *sandbox*, por segurança, a *remotedostring()* tenha sido removida do ambiente global, seria necessário encontrar alguma referência à máquina virtual mestre na *heap* da máquina virtual escrava ou até mesmo na *heap* do processo para poder manipular a máquina virtual mestre a partir da escrava.

3.2.2

Concorrência

A linguagem Lua oferece, como mecanismo nativo de suporte a concorrência, mecanismos de suporte a co-rotinas. Porém como a linguagem oferece recursos para ser facilmente estendida usando a linguagem C e os mecanismos de concorrência disponibilizados pelo SO normalmente são apresentados através de uma API C, é possível utilizar essas APIs C para criar novos mecanismos de concorrência similares as disponíbilizadas pelo SO para a linguagem Lua.

Copas

Também faz parte do projeto Kepler o módulo Copas (*Coroutine Oriented Portable Asynchronous Services for Lua*) (copas). Esse módulo segue o modelo proposto por A. Moura (revisitandoco-rotinas). Trata-se de um módulo que implementa um escalonador de tarefas baseado em co-rotinas que oferece chamadas de E/S em *sockets* TCP/IP. Quando essas chamadas não podem ser imediatamente executadas, elas passam o controle para o escalonador. Este, por sua vez, troca a co-rotina que está aguardando a E/S em um *socket* por uma que tenha a E/S em *socket* pronta para ser executada. Na falta de uma co-rotina que possa ser executada, o escalonador fica bloqueado até que alguma das operações de E/S solicitadas possa ser completada. Esse modelo permite que sejam abstraídos os detalhes da troca de tarefas, permitindo uma abstração de programação seqüencial usando chamadas de E/S que parecem ser síncronas mas encapsulam as transferências de controle.

3.3 Xavante

Outra parte crucial desse trabalho, oferecida pela plataforma Kepler, é o servidor web Xavante. Esse servidor implementa o protocolo HTTP 1.1 utilizando o módulo Copas como modelo de concorrência. Estender o Xavante foi a base de todo desenvolvimento realizado nesse trabalho.

3.4 LuaSocket

LuaSocket (luasocket) é um conjunto de módulos que oferecem suporte multi-plataformas as camadas de transporte TCP e UDP além de diversas funcionalidades comumente encontradas em aplicações que usam esses protocolos. Entre os módulos oferecidos estão o suporte a clientes SMTP, HTTP e FTP além de módulos para tratar MIME e URL.

O módulo TCP do LuaSocket precisou de pequenas modificações e extensões para atender a esse trabalho. Essas são discutidas na seção 4.3

3.5 Apachebench

Quando se examina o desempenho de um servidor web, diferentes fatores podem ser medidos. Midgley em (Midgley01) classifica esses fatores em:

taxa de respostas - taxa de requisições por unidade de tempo que o servidor consegue manter;

conexões simultâneas - número de conexões simultâneas que o servidor consegue atender sem erros;

throughput de dados - volume de dados transferidos por unidade de tempo;

número de conexões adequadas - número de conexões simultâneas que o servidor consegue atender mantendo um *throughput* mínimo;

Existem disponíveis algumas ferramentas para teste de desempenho de servidores HTTP. Um exemplo de ferramenta adequada para a medição dos últimos três itens é o httperf (mosberger98httperf, Midgley01). Já para medir o primeiro item encontramos a ferramenta Apachebench(ab, Bekman03).

O Apachebench é uma ferramenta para teste de desempenho de servidores HTTP e HTTPS distribuída juntamente com o servidor web Apache. Essa ferramenta permite, entre outras coisas, que seja aguardado um número determinado de respostas a requisições HTTP enquanto é mantido um número de conexões simultâneas. Por essa razão, utilizaremos essa ferramenta para gerar requisições concorrentes.

A saída dessa ferramenta pode apresentar diversos dados de forma discreta e consolidada.

Os principais fatores medidos por requisição são:

- tamanho do documento
- tempo total que os testes levaram para executar
- número de requisições completadas
- número de requisições que falharam e onde falharam (conexão, tamanho ou exceção)
- número de erros de escrita
- total de bytes transferidos

- total de bytes contendo HTML transferidos
- tempo de conexão (tempo decorrido para estabelecer uma conexão)
- tempo de espera (tempo decorrido para receber os primeiros bits de resposta da conexão)
- tempo de processamento (tempo decorrido desde o envio da requisição até o recebimento completo da resposta)

Como exemplo de dados consolidados temos:

- tempo total da requisição (tempo de conexão + tempo de espera)
- tempos mínimo, máximo, média e mediana dos tempos de conexão, espera, processamento e total
- Requisições por segundo (requisições completadas / tempo que os testes levaram para executar)
- Tempo médio por requisição considerando o todas as requisições (tempo que os testes levaram para executar / requisições completadas)
- Tempo médio por requisição considerando a concorrência (concorrência * tempo que os testes levaram para executar / requisições completadas)

A própria ferramenta já oferece por padrão dados consolidados na saída. Como o volume de requisições, e por conseguinte o volume de dados é muito grande, optamos por utilizar somente o dado consolidado referente ao número de requisições atendidas por segundo.