4
Modelos Alternativos de Concorrência e Sandbox

Para evitar inconsistências nas comparações de medidas devido ao efeito de executar testes em plataformas de software diversas, optamos por utilizar uma plataforma única, que facilitasse a implementação de extensões para modificar o comportamento do sistema. Foi escolhido o servidor web Xavante como plataforma base. Devido à sua arquitetura modularizada, foi possível criar novos módulos para estender suas funcionalidades e alterar seu comportamento.

J. Hu et. al.(hu99jaws) propõem a utilização de *frameworks* e padrões de projeto em projetos orientados a objeto para software de comunicação. O objetivo é conseguir alta adaptabilidade e desempenho, permitindo assim que através de pré-configurações se possa alterar o comportamento do sistema, carregando estaticamente ou dinamicamente diferentes estratégias dependendo da carga e infra-estrutura de software e hardware.

Ainda que o código do Xavante não refletisse diretamente uma arquitetura orientada a objetos, procuramos aplicar padrões de projeto para estender a implementação do Xavante com novos modelos de concorrência e de *sandbox*.

O Xavante original utiliza a máquina virtual Lua padrão, como módulo de E/S em *sockets* o LuaSocket e como módulo de concorrência o Copas. O módulo servidor HTTP se chama "httpd" e implementa o protocolo HTTP 1.1. Existem diversos tratadores de pedidos implementados: para servir arquivos (páginas web estáticas), redirecionamentos de URLs e gerar páginas criadas dinamicamente em Lua com a API CGILua.

Nesse capítulo descrevemos como criamos novos módulos e como modificamos módulos já existentes para atenderem aos objetivos desse trabalho.

4.1
Alternativas escolhidas

O Xavante original utiliza co-rotinas como modelo de concorrência e já utiliza, em versões diferentes, dois modelos de *sandbox*.

Mantivemos os modelos já utilizados pelo Xavante e acrescentamos aos modelos de concorrência um modelo baseado em atendimento seqüencial de requisições, um modelo *multi-threads* e um modelo multi-processo. Quanto

aos modelos de *sandbox*, incluímos a possibilidade de não usar *sandboxes* e a possibilidade de usar como *sandbox* um processo ligado por *sockets* ao servidor.

O modelo de concorrência seqüencial e o modelo sem *sandbox* foram incluídos na intenção de servirem de referência.

Como resultado, os modelos de concorrência selecionados foram: seqüencial, co-rotinas, *threads*, processos disparados do zero e processos clonados pela chamada de sistema *fork* do padrão POSIX (*Portable Operating System Interface*) normalmente implementada nos SOs da família Unix. Os modelos de *sandbox* selecionados foram: sem *sandbox*, Venv, Rings e processos.

As figuras 4.1 e4.2 sumarizam os modelos de concorrência e *sandbox* adotados.

4.1.1 Processos como mecanismo de concorrência versus processos como mecanismo de sandbox

Utilizamos multi-processos como mecanismo de concorrência porque o SO garante a execução concorrente de seus processos através de mecanismos de preempção. Dois processos na mesma máquina naturalmente já concorrem por tempo de execução, além de outros recursos em comum que queiram utilizar.

Quando utilizamos processos como mecanismo de *sandbox* estamos interessados no controle que o SO faz do domínio de cada processo. Mais especificamente, estamos interessados na garantia de uma área de memória exclusiva, na liberação de estruturas de E/S e áreas de memória ao fim da execução do processo, nos mecanismos de comunicação com outros processos, além do controle de acesso por ACL aos recursos da máquina.

A questão da preempção só tem impacto no comportamento do processo como mecanismo de concorrência e não tem maiores impactos no comportamento do processo como *sandbox*. O processo ser interrompido pelo SO não compromete os seus dados nem o seu emcasulamento mas compromete o proseguimento do processamento.

Já as questões de área de memória protegida e liberação de recursos ao final do processamento, ao contrário da preempção, tem impacto no comportamento do processo como *sandbox* e não têm maiores impactos no comportamento do processo como mecanismo de concorrência.

Os mecanismos de comunicação entre processos e o controle de acesso a recursos compartilhados influem nos dois mecanismos. A comunicação entre processos ou com recursos da máquina quase sempre implicam em sincronização. A sincronização por sua vez influi no comportamento da concorrência. Além disso, o mecanismo de comunicação é a forma de ultrapassar a barreira da

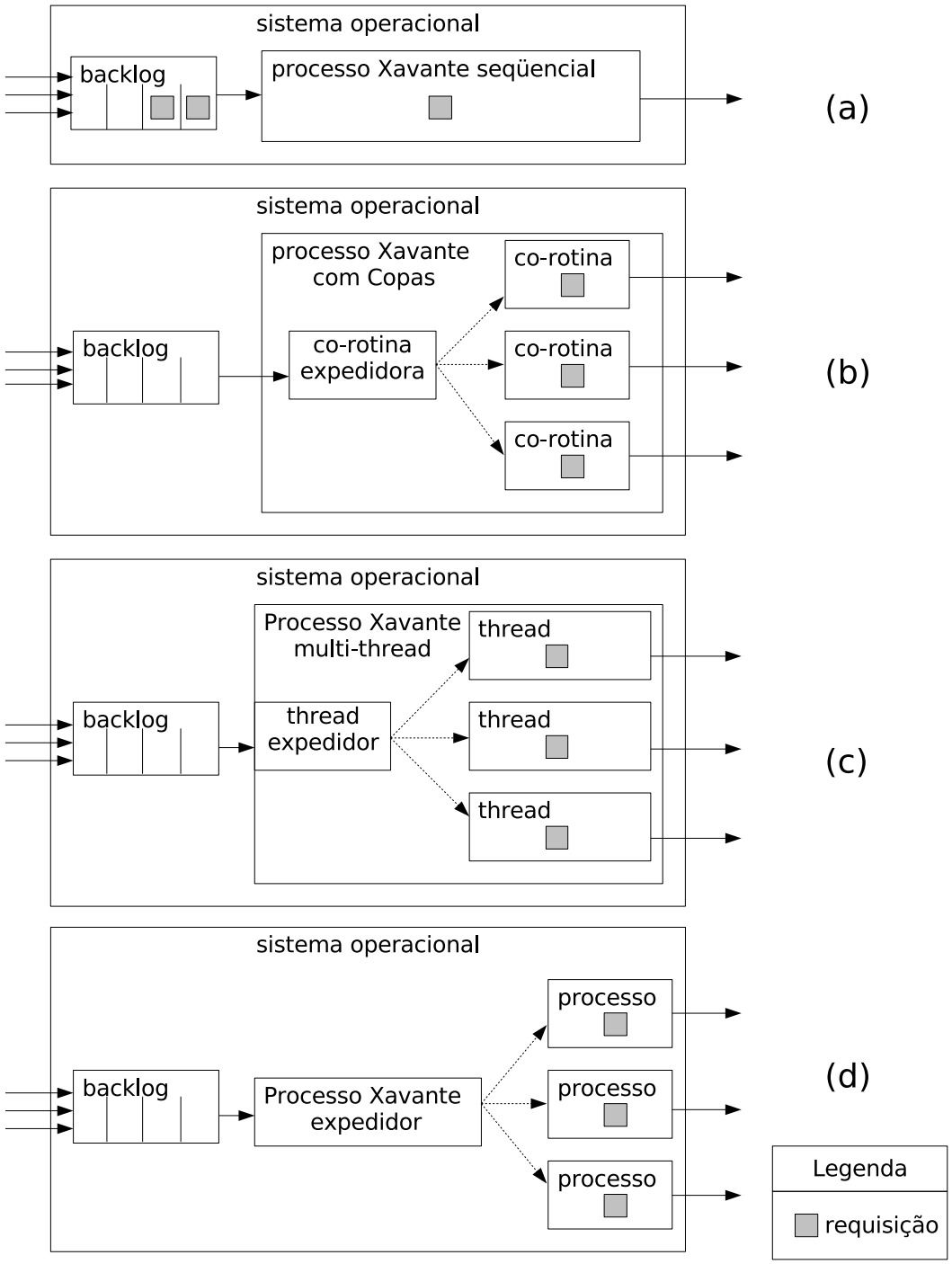

Figura 4.1: Diagramas dos 4 modelos de concorrência selecionados. (a) seqüencial, (b)com co-rotinas, (c)*multi-threads*, (d) multi-processos e multi-clones.

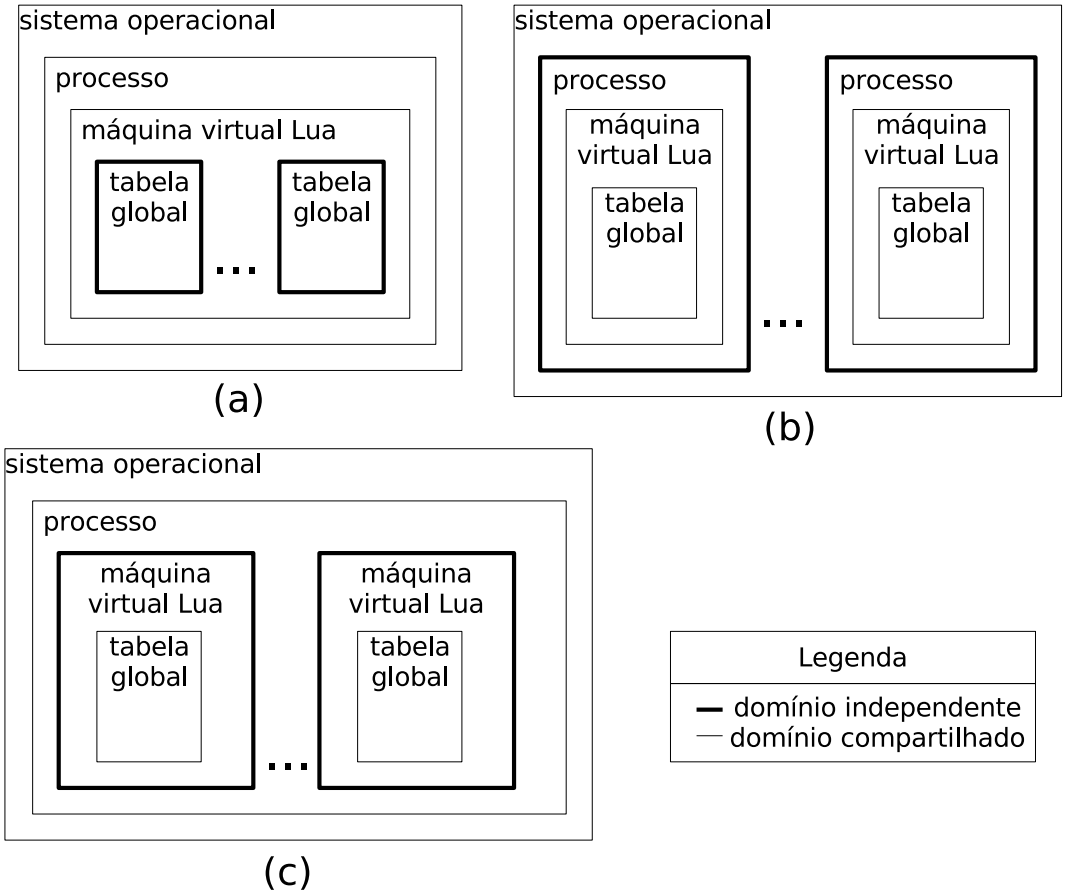

Figura 4.2: Diagramas ilustrando o isolamento de domínio nas diversas sandboxes. (a) Venv - tabela global Lua como domínio independente, (b) processo do SO como domínio independente, (c) Rings - máquina virtual em um mesmo processo do SO como domínio independente.

sandbox. Para se comunicar com outro, um processo precisa de alguma forma escrever em uma área de memória que outro possa de alguma forma ler, isso já implica em uma interceção entre os domínios dos processos. Supondo que os processos executam concorrentemente (se intercalando ou simultâneamente em processadores diferentes), se mais de um processo quer acessar ou modificar uma região de memória, somente um dos processos prossegue o seu processamento enquanto os outros aguardam até que a operação deixe a região de memória em uma condição consistente, influenciando a fluidez do mecanismo de concorrência.

4.2 Implementação

Todo o desenvolvimento foi realizado a partir do código do servidor web Xavante. Inicialmente, para auxiliar o entendimento da arquitetura, foi levantado um modelo UML do Servidor. Para realizar essa tarefa foi

necessário abstrair o fato do servidor não estar implementado usando classes e objetos. O servidor utiliza basicamente estruturas nativas da linguagem Lua como variáveis locais e globais, módulos, funções e tabelas. Modelamos os módulos Lua como objetos *singleton* (GOF) e funções retornadas por fábricas como objetos. As interfaces são implícitas, ou seja, a linguagem não força a sua implementação. O arquivo de configuração não possui apenas atribuições, possui também chamadas de funções, então a configuração também foi modelada em UML.

A partir do modelo, notou-se que configurações estavam sendo feitas fora do arquivo de configuração, e poderiam ser facilmente movidas. Mostrou-se também que o servidor já disponibilizava como ponto flexível o tratador do conteúdo das requisições HTTP, seguindo o padrão de projeto *factory method* (GOF) Esse tratador permite que se mapeie URLs para tratadores específicos.

A implementação consistiu em modificar o servidor para permitir que o modelo de concorrência fosse mais um ponto flexível. Foram criados novos modelos de concorrência, novos tratadores de requisições e uma aplicação para testes de desempenho.

O Xavante original trabalha apenas com o modelo de concorrência de co-rotinas com E/S assíncrona (Copas). Criando módulos com a mesma interface e semântica semelhante, como no padrão de projeto *strategy* (GOF), foi possível alterar o servidor de modo que o modelo de concorrência possa ser definido alterando o arquivo de configuração ou na linha de comando quando se dispara a aplicação. O diagrama UML do sistema modificado, com os padrões de projeto identificados por uma etiqueta, pode ser visto na figura 4.3.

Nos servidores web onde não há preocupação com persistência de dados, as requisições podem ser tratadas de uma forma completamente independente pois toda a informação sobre a requisição é fornecida através do canal de comunicação. Uma tarefa fica encarregada de esperar novas requisições. Quando essa requisição chega, ela passa o canal de comunicação para outra tarefa e volta a esperar por novas requisições, a não ser no modelo seqüencial, onde ela mesma trata a requisição. Com exceção do modelo seqüencial, a tarefa passa uma referência ao canal de comunicação para a tarefa que processa a requisição HTTP. No caso de co-rotinas, essa referência é objeto Lua e no caso de *threads* ou processos, um identificador numérico (*file descriptor* ou *file handler*).

A aplicação para testes implementada parseia a URL procurando por 2 valores: o tamanho da resposta em Kbytes e o número de ciclos de *loop* vazios a serem executados. Depois disso, ela intercala o envio de uma *string* de 1Kbyte e a execução de um *loop* vazio. Já que esse *loop* vazio executa para cada *string* enviada, ele tem o número ciclos igual ao total de ciclos vazios solicitado na

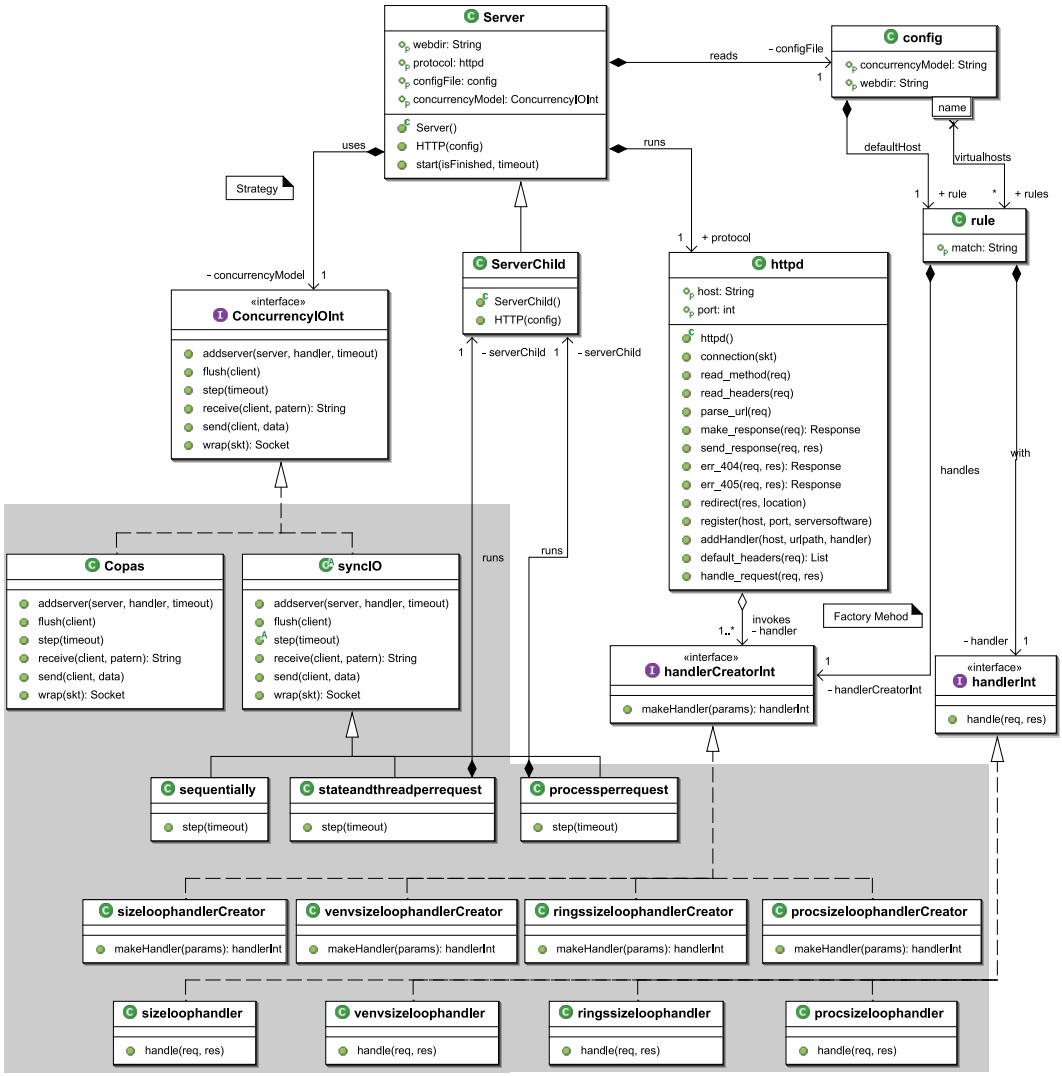

Figura 4.3: Diagrama UML do framework Xavante modificado, com as extensões implementadas na área cinza.

URL dividido pelo tamanho da resposta em Kbytes.

Seria possível oferecer uma API única para a mesma aplicação funcionar em todos os modelos de *sandbox*, porém como a aplicação é muito simples, optamos por implementá-la diretamente para cada *sandbox*, usando os recursos de cada arquitetura da melhor forma possível. Nos modelos sem *sandbox*, Venv e Rings foi possível embutir a aplicação diretamente no tratador de cada *sandbox*. No caso do modelo de *sandbox* por processos independentes, foi necessário criar um arquivo separado com a aplicação.

4.3 Módulo de sockets TCP

O módulo TCP do LuaSocket original oferece todos os recursos necessários para implementar um servidor web seqüencial ou em múltiplas co-

rotinas, porém oferece algumas barreiras para os modelos *multi-threads* e multi-processos.

A primeira barreira é que, apesar do LuaSocket oferecer uma chamada para informar o identificador numérico de um *socket*, não oferece uma chamada que a partir de um identificador numérico consiga criar um objeto LuaSocket que manipule esse *socket*.

A outra barreira é semântica. Quando um objeto LuaSocket é coletado por não ter mais referência para ele, o *socket* é automaticamente fechado. No caso de processos filhos que herdam o *socket*, o próprio SO se encarrega de manter o *socket* aberto em um processo mesmo que o outro o tenha fechado. Porém no caso de *threads*, se qualquer *thread* fechar um *socket*, ele é fechado para todos os *threads* que compartilham os recursos do mesmo processo. No caso onde criamos *threads* com máquinas virtuais Lua separadas, se cada máquina virtual possui um objeto LuaSocket encapsulando o mesmo *socket*, caso o objeto LuaSocket de uma das máquinas virtuais seja coletado, o *socket* é fechado para todas as máquinas virtuais.

Foram implementadas então duas modificações. Foi criada uma chamada que cria um LuaSocket a partir do identificador de um *socket* e foi removido do meta-método de coleta de lixo do objeto LuaSocket o trecho que fecha o *socket*. Depois dessa última modificação, os *sockets* só são fechados quando é explicitamente realizada a chamada de fechamento. A solução adotada para manter a conexão TCP viva demanda alguma disciplina do programador a fim de evitar que seja necessário implementar mecanismos de sincronização entre máquinas virtuais Lua somente para controlar a consistência do *socket*.

No Xavante tomamos o cuidado de só realizar a chamada de fechamento ao fim da tarefa que trata a requisição pois o fechamento indevido da conexão causaria erros de E/S.

4.4 Módulo de threads e processos

O modelo de *sandbox* baseado em processos segue a linha dos processos CGI em servidores web. As APIs de criação de processos no Windows e nos sistemas POSIX são bastante diferentes, além de possuírem semânticas diferenciadas em vários pontos. Em ambos os casos os SOs oferecem uma fatia de memória protegida e ciclicamente uma fatia de tempo de processamento para cada processo. Processos podem criar processos filhos e manter referências para os eles, enquanto os processos filhos podem herdar as referências para os canais de E/S do pai. De diferença temos que nos SOs POSIX, um processo filho é criado como uma cópia idêntica do processo pai, enquanto no Windows

um novo processo precisa carregar o código binário que irá executar, lendo do sistema de arquivos.

Para permitir uma semântica única seguimos o modelo do Windows porque implementar a semântica da chamada de sistema *createProcess* do Windows utilizando as chamadas *fork* e *execve* do POSIX é uma tarefa relativamente simples se comparada com implementar a clonagem de processos usando chamadas de sistema nativas do Windows. A biblioteca Cygwin (cygwin) oferece essa extensão porém ao custo de um desempenho muito inferior.

Inicialmente implementamos separadamente uma função de criação de processos que podia receber e enviar dados para o processo pai através de *pipes* anônimos na intenção de usá-la para a criação de sandboxes, porém, com o decorrer do projeto, essa técnica de comunicação entre processos foi abandonada. A questão que levou a esse abandono foi que, para o modelo de co-rotinas não ficar bloqueado desnecessariamente, só podemos usar comunicação assíncrona. Seria possível implementar facilmente a comunicação assíncrona parando apenas na chamada de sistema *select* em ambientes POSIX, porém no Windows o *select* só funciona com *sockets*, e seria necessário mudar completamente a arquitetura do sistema para conseguir se comunicar assincronamente com alguns dos modelo de *sandbox*. No lugar dos *pipes* foram utilizados *sockets*. Para utilizar *sockets* foi necessário modificar o módulo Copas, como será visto na próxima seção.

Foi implementada uma função de criação de processos com a semântica do Windows tanto para Windows como para Linux e uma outra com a semântica POSIX somente para Linux. Também foi criada uma função para a criação de máquinas virtuais Lua em um novo *thread* do SO. A semântica Windows serviu para implementar o modelo multi-processos, a semântica POSIX serviu para implementar o modelo multi-clones e a que utiliza *threads* serviu para implementar o modelo *multi-threads*.

A API desse módulo ficou assim:

`luaprocess.fork()` cria um novo processo com a semântica de um processo POSIX desatrelado do pai.

`luaprocess.start(string, ...)` cria uma nova máquina virtual Lua em um *thread* separado depois executa a `string` na nova máquina virtual. Os argumentos são armazenados na tabela `arg` da nova máquina virtual.

`luaprocess.startsoprocess(exec, args)` executa o arquivo `exec` em um novo processo, passando o conteúdo da tabela `args` como parâmetros para esse novo processo.

A mesma função que cria processos para sandbox, cria processos para o modelo de concorrência.

4.5 Módulo Copas

Originalmente esse módulo associava um *socket* a uma co-rotina. Assim cada tarefa consistia de uma co-rotina que era escalonada nas operação de E/S do seu *socket*. O problema era que cada tarefa só podia transmitir e receber dados em um único *socket*. Para permitir que uma tarefa abra uma nova conexão TCP, no caso, para se comunicar com outro processo, e aproveite essa conexão para realizar escalonamento nos *timeouts*, cada *socket* passou a ser associado a uma pilha FIFO de tarefas a serem atendidas quando a sua leitura estiver disponível e outra pilha FIFO com tarefas a serem atendidas quando sua escrita estiver disponível. Essa nova arquitetura é bem mais flexível e permite inclusive que tarefas diferentes manipulem o mesmo *socket*. Por exemplo, uma tarefa pode processar apenas os dados recebidos em um *socket* enquanto outra somente transmite nesse *socket*.

4.6 Testes automatizados

Para testar o módulo de *threads* e processos e o módulo TCP modificado foram criados alguns *scripts* Lua que testam todas as funções criadas ou modificadas.

Os testes são organizados de forma que qualquer erro não tratável interrompe a execução do *script* e os erros tratáveis geram mensagens texto na saída padrão do tipo :

- testando a criação de novo processo do sistema operacional [OK]
- testando a criação de novo processo do sistema operacional [FALHA]

Caso o teste resulte em sucesso ou falha, respectivamente.

Os testes foram agrupados por *script* da seguinte forma:

- Teste de todas as funções da API do módulo de *threads* e processos.
- Testes da API de duplicação de *socket* no mesmo estado Lua.
- Testes da API de duplicação de *socket* em estado Lua diferente usando o módulo Rings.
- Testes da API de duplicação de *socket* em processo diferente usando o módulo de *threads* e processos.

4.7 Bateria de testes de desempenho

Para testar o servidor web Xavante seria adequado o uso de testes de unidade onde cada classe é testada como um sistema separado, porém o objetivo desse trabalho não é a robustez do servidor web e sim o desempenho sob diferentes condições.

Foi criado um par de *shell scripts* que automatizaram os testes de desempenho. Um *script* fica na máquina onde são executados os servidores e reinicializa 5 instâncias do servidor web Xavante, cada uma com um modelo de concorrência. O segundo *script* fica em uma máquina cliente. Esse *script* realiza um teste de desempenho para cada combinação de fatores desejados. Para cada combinação esse *script* executa todos os passos descritos mais adiante para minimizar as interferências, entre eles executar via ssh o *script* de reinicialização do Xavante na máquina servidor e depois chamar o Apachebench no cliente com os parâmetros para realizar o teste condizente com a combinação a ser testada. Depois da execução, o *script* armazena o número de requisições atendidas por segundo em um arquivo texto.

4.8 Ajustes para o sistema executar no SO Windows

O *script* de reinicialização do Xavante precisou ser reescrito para um arquivo de lote do Windows (.bat) e o acesso remoto, que no Linux era realisado por ssh, passou a ser realizado por uma nova instância do servidor Xavante que tinha a função de reinicializar os 4 servidores que sofriam o teste (não há o modelo de clones no Windows).

Quanto ao servidor web, desenvolvemos a parte Lua do Xavante e aplicação de teste inicialmente no SO Linux. Havia a esperança de que essa parte do sistema não precisaria de nenhuma modificação para ser executada no S.O Windows, porém foram encontrados alguns comportamentos diferentes no módulo TCP do LuaSocket. Aparentemente no Linux não é necessário a atribuição de um tamanho para o *backlog* na chamada *listen*. O tamanho do *backlog* parece ser configurado globalmente no sistema e não conseguimos atingir o valor limite através de testes que só puderam chegar a 1021 (número de identificadores por processo suportados no sistema cliente). Quanto ao Windows, os primeiros testes foram realizados sem definir o *backlog* assim como no Linux, resultando em erros do tipo "*Connection refused*" quando se tentava realizar mais de 32 conexões simultâneas. Então incluiu-se o argumento *backlog* na chamada *listen*.

Durante os testes de desempenho notou-se ainda mais dois erros. A bateria de requisições que chamavam o modelo de *sandbox* baseado em processos separados não chegavam ao final quando o volume de informações era grande. No meio da caminho apresentavam o erro "*Connection reset by peer*". Após analisar os pacotes gerados nessas conexões através de um analisador de pacotes, notou-se que ao fim da transmissão, o servidor estava enviando um pacote com a *flag* RST (pedido de *reset*) ativada. Resolveu-se isso incluindo uma chamada *shutdown* imediatamente antes da chamada *close*.

O outro erro só aparecia quando o cliente requisitava sequencialmente d(sem concorrência) uma resposta de 1Mbyte para o modelo de concorrência baseado em co-rotinas. O erro era "*The timeout specified has expired*". O manipulador de *sandboxes* por processo estava lendo todos os dados recebidos do processo de uma única vez e enviando também em uma única vez. O código foi alterado para ler o canal TCP da *sandbox* em blocos de 1024 bytes e renviar para o cliente a cada bloco recebido.

4.9 Outras verificações

Nessa seção discutimos dois pontos aos quais dedicamos atenção especial. O primeiro deles diz respeito à concorrência entre os pedidos de conexão e o outro se realmente está ocorrendo escalonamento no atendimento das requisições.

Como somente um pacote passa pela rede de cada vez, supondo que não ocorressem colisões na rede, mesmo que 100 máquinas diferentes disparem simultaneamente uma conexão com uma determinada máquina, esses pedidos de conexão chegarão seqüencialmente nessa máquina, provavelmente na forma de uma rajada de pacotes vindos de 100 origens diferentes. Então uma única máquina gerando 100 pedidos de conexão conseguiria emular 100 pedidos de origem diferentes desde que conseguisse gerar uma rajada que gerasse uma carga semelhante a que viria de 100 máquinas. Isso pode significar usar 100% da banda passante ou ao menos encher o *backlog* tanto quanto as 100 máquinas fariam. Na prática, comprovamos que o cliente estava emulando corretamente pedidos concorrentes quando tentamos disparar mais de 200 conexões concorrentes contra um servidor Windows. Até 200 conexões concorrentes, que é o tamanho máximo do *backlog* do Windows, nenhum erro era apresentado. Quando tentamos 201 conexões concorrentes, imediatamente recebemos uma mensagem de conexão rejeitada. Isso significa que o cliente está sendo capaz de encher o *backlog* mais rápido que o servidor aceita as conexões. No caso do Linux atingimos o número máximo de descritores de arquivo abertos pela

ferramenta de teste (*stdin,stdout,stderr* + 1021 *sockets*) antes de saturar o *backlog* do servidor.

Para verificar se as tarefas estavam escalonando durante a implementação dos modelos de concorrência, a técnica utilizada foi abrir diversas instâncias de um navegador Web numa mesma URL e recarregar as páginas simultaneamente, verificando visualmente se as páginas carregavam simultaneamente. O Navegador utilizado foi o Firefox que precisou ser configurado para trabalhar sem *cache* e permitir mais de 2 conexões simultâneas ao mesmo servidor (limitação da especificação do protocolo HTTP) . Posteriormente, durante os testes de carga, muitas vezes as URLs testadas geravam muito pouco dado para a verificação visual então outra técnica precisou ser empregada. No modelo seqüencial, contamos que não haverá escalonamento baseado no fato de utilizarmos apenas um processo e chamadas de sistema síncronas. No modelo de co-rotinas, aproveitamos que as tarefas só enxergam chamadas de E/S síncronas que na realidade encapsulam chamadas de E/S assíncronas e a lógica de passagem de controle ao escalonador, para acrescentar nesse encapsulador mecanismos de *log*. Assim é possível registrar sempre que uma tarefa passa o controle ao escalonador devido a timeout de E/S.

Já nos outros modelos (*multi-threads*, multi-processos e multi-clones) somente a análise da distribuição de tempos de resposta das requisições concorrentes pode dar uma pista. Baseado nas respostas dos modelos seqüencial e de co-rotinas notamos que, quanto mais escalonamento, menor o desvio padrão do tempo de resposta de conexões concorrentes. Essa informação já aparece consolidada na saída da ferramenta Apachebench. Para requisições com volumes de processamento e transferência baixas a comparação de desvio padrão não oferece margens conclusivas, porém para volumes de processamento e transferência altos essa técnica mostrou que realmente estava havendo escalonamento. O tempo de resposta entre as respostas é bastante parecido. Isso indica que as requisições foram tratadas concorrentemente pois se isso não ocorresse as últimas requisições a serem respondidas estariam acumulando no seu tempo de resposta o tempo de todas as requisições que foram atendidas anteriormente (atendimento seqüencial).