5
Planejamento dos experimentos

Depois de escolher e implementar 5 modelos de concorrência e 4 modelos de sandboxes que consideramos representativos para um servidor web Lua, procuramos os principais fatores que podemos variar para exercitar os modelos.

Para comparar o desempenho dos diversos modelos de concorrência com diversos modelos *sandboxes* é preciso levar em conta que diversos fatores geram influência no desempenho dos modelos. Entre eles podemos citar o SO, o volume de chamadas concorrentes, o tamanho da janela TCP, o volume de dados, a banda passante, o volume de processamento, o número de processadores, a velocidade dos processadores, a latência da rede e o tráfego da rede.

Comparar as combinações dos modelos foco desse trabalho com todas as possibilidades dos fatores que podem gerar influência geraria um volume de combinações que não caberiam nesse trabalho. Por isso selecionamos combinações que consideramos representativas.

Nesse capítulo descrevemos os fatores que consideramos relevantes para as medidas e como minimizamos o efeito dos outros fatores.

5.1
Escolha de Fatores

A criação das nossas *sandboxes* e a comunicação entre tarefas de diferentes *sandboxes* demandam processamento além do exigido pelo modelo sem *sandbox*. Estamos interessados no peso da criação de cada tipo de *sandbox* e na sobrecarga relativa à comunicação entre cada tipo de *sandbox* e o servidor.

Um dos objetivos dos modelos de concorrência é aproveitar o tempo que uma execução seqüencial ficaria parada esperando por operações de E/S. Esse tempo é aproveitado para processar outras tarefas. Gostaríamos de medir o quanto ganhamos aproveitando esse tempo de espera com diferentes técnicas de implementar concorrência. Para aproveitar o tempo de espera fazendo outra tarefa, é necessário que existam outras tarefas que possam ser realizadas concorrentemente. Gostaríamos de saber o efeito do aumento do número de tarefas concorrentes. No nosso caso, essas tarefas consistem em

atender requisições HTTP. Essas tarefas podem ser processadas em paralelo sem maiores problemas.

Mesmo aproveitando o tempo de espera para processar outras tarefas, é comum ocorrer o fato de todas as tarefas acabarem ficando paradas em espera por operações de E/S ao mesmo tempo. Como cada aplicação tem suas próprias demandas de processamento e E/S, gostaríamos de testar como esse tempo é aproveitado variando o volume de E/S e de processamento.

5.1.1 Plataformas de teste

Gostaríamos de poder desprezar as variáveis com influência no desempenho introduzidas pela rede supondo que esta seria uma rede ideal, com banda passante infinita e latência zero. Supomos que, se uma máquina não consegue gerar pacotes rápido o suficiente para saturar a rede, podemos considerar que essa máquina está conectada em uma rede com banda passante infinita. Idealmente estamos considerando que se não há espera nas chamadas de E/S, o gargalo é a capacidade de processamento. Se o gargalo é o processamento, também podemos considerar que a latência da rede é desprezível.

No nosso parque computacional tínhamos disponíveis diversos computadores mono-processadores, com a arquitetura IA-32 de 32 bits (IA-32) em diversas configurações e velocidades; e um grupo de 3 computadores com 4 processadores de 1,5 GHz na arquitetura Itanium de 64 bits (Itanium).

Tínhamos disponíveis 2 tipos de rede. As duas eram Ethernet, sendo a primeira de 100Mbits e a segunda de 1Gbits. A rede de 1Gbits seria a mais indicada para emular uma rede ideal já que sabemos que os computadores modernos conseguem facilmente saturar uma rede de 100Mbits. O problema é que somente as máquinas Itanium estavam ligadas a rede de 1Gbit. Selecionamos duas dessas máquinas que estavam ligadas em um mesmo *switch* para testar os sistemas em um ambiente multi-processador.

Para testar os modelos em um ambiente mono-processador ficamos limitados a rede 100Mbits. Para tentar não sobrecarregar a rede, escolhemos para servidor o equipamento mais lento disponível, um Celeron de 466MHz e para cliente o equipamento mais rápido disponível, um Pentium 4 HT de 3GHz. Tivemos acesso total às máquinas onde pudemos instalar, configurar e administrar os SOs, inclusive conectando as duas máquinas em um *switch* exclusivo para elas, criando uma rede com apenas dois computadores.

Número de processadores

Para testar o efeito do número de processadores seria interessante a comparação dos resultados de uma máquina mono-processador com uma máquina multi-processador com o mesmo tipo de CPU e velocidade. Porém, conforme descrevemos, tínhamos disponíveis nos parques computacionais apenas as opções 1 ou 4 processadores com arquiteturas diferentes. Sendo assim, as comparações devem ser feitas qualitativamente.

5.1.2 Conexões concorrentes

O tamanho do *backlog* da porta TCP influencia o momento que um servidor TCP vai começar a rejeitar conexões. *O backlog* é uma fila onde o SO coloca os pedidos de conexão recebidos pela pilha TCP e de onde o servidor retira esses pedidos quando executa uma chamada *accept*. Se o servidor retira os pedidos numa taxa mais lenta do que esses pedidos entram na fila, o fila enche e o SO começa a rejeitar conexões nessa porta.

Nos nossos sistemas o número de tarefas que podem ser realizadas em paralelo, no caso, as requisições HTTP, podem ser contabilizadas pelo número de conexões concorrentes. Para começar, podemos medir os sistemas sob carga de apenas uma conexão (concorrência igual 1, ou sem concorrência) e ir aumentando a concorrência até que alguma parte do sistema sature. No Linux atingimos o número máximo de conexões do nosso cliente Apachebench (1021) sem aparentemente saturar o servidor, porém, notamos que o número máximo de conexões esperando para serem aceitas (*backlog*) no SO Windows é 200.

A comparação entre alto e baixo volume de eventos normalmente é melhor visualizada usando uma escala logarítmica. Para trabalhar em uma faixa que funcione bem nos dois SOs e poder usar uma escala logarítmica, escolhemos trabalhar com 1, 10 e 100 conexões concorrentes.

5.1.3 Volume de dados

No nosso sistema existem basicamente 2 tipos de E/S, o de disco para a carga do código a ser executado e o de rede que são os dados transferidos entre cliente e servidor. Poderíamos ler os dados a serem transmitidos do disco, porém, no modelo de concorrência baseado em co-rotinas somente implementamos escalonamento em E/S de rede, então para obter medidas comparativas mais consistentes, tivemos que evitar ao máximo a E/S em disco durante as medições. Para mostrar o efeito da variação do volume de E/S, variamos o tamanho das transferências de dados na rede.

Como nos modelos com *threads* e processos o servidor carrega a cada requisição os seus executáveis do disco, eles sofrem a influência do tempo em E/S para acesso a arquivos em disco. Para reduzir esse efeito, tentou-se minimizar o acesso em disco, passando trechos de código como parâmetro, juntando-se códigos no mesmo arquivo e gerando dados para a transmissão diretamente no servidor, sem acessar disco.

Escolhemos empiricamente 10Kbytes, 100Kbytes e 1Mbyte porque além de estarem em uma escala logarítmica, também consideramos que enquadram uma boa parte das transferências HTTP comuns na Internet hoje em dia. Apesar de que, com a proliferação de pontos de acesso internet com banda larga, já é mais comum fazer *download* de arquivos de 600Mbytes por HTTP. Por causa da Web 2.0 pequenos trechos de menos que 1K com dados descritos em XML também estão se tornando comuns.

Esperamos que a variação do volume de E/S dê uma idéia da eficiência dos diferentes mecanismos de escalonamento para evitar que o sistema fique bloqueado em E/S. Esperamos também que dê uma idéia do peso da comunicação do servidor com a *sandbox*.

5.1.4 Volume de processamento

Para emular processamento de uma maneira facilmente controlada decidimos usar um *loop* vazio em Lua onde podemos variar o número de ciclos do *loop*.

Testes iniciais com o volume de processamento no sistema Celeron mostraram que *loops* com menos de 10.000 ciclos não mostravam alteração perceptível no *throughput* do sistema base. Então testamos como referência o caso de não executar nenhum ciclo adicional fora a E/S e, além disso, 100.000 e 1.000.000 ciclos.

5.1.5 Sistemas operacionais

O algoritmo de escalonamento de *threads* e processos, a implementação dos mecanismos de E/S (entre outras coisas) variam entre os tipos de SOs.

Nesse trabalho consideramos que são representativas 2 famílias de SO, a família dos sistemas estilo Unix que seguem o padrão POSIX e a família Windows. Para representar a família Unix foi escolhido o Linux 2.6 e para representar a família Windows foi escolhido o Windows 2000 profissional. Assim ficamos com 2 opções de SOs. Porém, devido ao fato da família Windows não oferecer uma chamada de sistema para clonagem de processos, foram

eliminados os teste do modelo de concorrência baseado em clonagem no Windows 2000.

5.1.6 Janela TCP

O tempo que uma tarefa fica esperando uma operação de E/S que usa o protocolo TCP inclui não apenas o tempo de transmissão do dado mas também o tempo de confirmação de que o dado foi entregue corretamente. Esse protocolo tem algumas otimizações para deixar as operações de E/S mais fluidas, isto é, não precisar parar esperando por confirmação a cada trecho de dado enviado. Entre as otimizações estão a utilização de buffers de transmissão e de janelas de transmissão de dados sem confirmação. Os buffers permitem que os dados fiquem armazenados em locais intermediários enquanto não estão na rede ou não são lidos pela aplicação. Enquanto os *buffers* não estiverem cheios, a aplicação consegue transmitir. A janela permite que uma seqüencia de dados seja transmitida sem esperar por confirmação imediata. Se a confirmação não chega em um determinado intervalo, o protocolo considera que o dado foi perdido.

Os SO normalmente implementam uma pilha TCP que adapta dinamicamente a janela TCP de acordo com as condições do enlace de dados e dos *buffers* de cada parceiro em cada momento. Conexões com banda limitada ou alto índice de perdas de pacote funcionam melhor com uma janela pequena. Já conexões com alta banda passante e poucos erros funcionam melhor com janelas maiores. O tamanho da janela afeta a taxa com que se consegue transmitir dados. Janelas pequenas reduzem a banda passante. A variação dinâmica dos pacotes pode mascarar diversas medidas, então medir extremos de um intervalo de janelas pode dar uma idéia melhor de como o sistema pode se comportar em diferentes situações. Assim, a princípio adotamos 2 opções de janelas TCP, porém, esse tipo de configuração normalmente é um parâmetro global do SO. Para modificar esse tipo de parâmetro é necessário acesso de administrador a esses sistemas. Como não temos acesso administrativo às máquinas com 4 processadores, os testes tiveram que ser feitos com janelas dinamicamente variáveis. No SO Windows, por dificuldade de documentação, também usou-se janelas TCP dinamicamente variáveis.

No sistema Linux-Celeron foram realizados testes fixando a janela em 4Kbytes, tanto no cliente como no servidor, e depois repetimos os testes fixando para 128Kbytes a fim de termos bases para comparação.

5.2

Sistemas Implementados

A partir das limitações técnicas foi escolhido um sistema para servir de base nas comparações. Esse sistema escolhido foi o Linux-Celeron, com janela TCP de 4k, e nele foram testados 5 modelos de concorrência, 4 de *sandbox*, 3 volumes de dados e 3 volumes de processamento, totalizando uma combinação de 180 pontos. Repetimos o teste no mesmo sistema configurando a janela TCP para 128k, com o objetivo de mostrar o efeito de menos E/S bloqueante no sistema, também resultando em 180 pontos. Executamos testes semelhantes na mesma máquina porém com SO Windows com janela TCP variável. Foram testados 4 modelos de concorrência, 4 de *sandbox*, 3 volumes de dados e 3 volumes de processamento, totalizando uma combinação de 144 pontos. Para verificarmos o efeito do uso de múltiplos processadores, repetimos os testes no Linux-4processadores, acrescentando assim mais 180 pontos de teste.

Depois da análise dos resultados acima achamos razoável fazer mais uma bateria de testes, no sistema Linux-Celeron sem controle do tamanho da janela TCP, desta vez inserindo uma latência de 50 mili-segundos na transmisão de todos os pacotes. Acrescentamos assim mais 180 pontos de teste.

Um fato que atrapalhou bastante a evolução dessa bateria de testes foi o tempo que ela levava para completar uma execução no sistema base. Uma bateria no sistema base com 180 combinações de teste exige aproximadamente 7 horas. No sistema com 4 processadores esse tempo cai para 1 hora. Introduzindo a latência de 50ms na comunicação do sistema Linux-Celeron sem controle do tamanho da janela TCP passa a exigir aproximadamente 16 horas.

5.3 Ações para minimizar interferências nas medidas

Para minimizar o efeito de ruídos nas medições para cada ponto estudado foram realizadas 10 amostras. Ou seja, para cada uma das plataformas testadas, foram executadas 10 baterias de testes idênticos.

O objetivo dos testes é medir a eficiência dos modelos de concorrência e *sandbox*, tendo como métrica o número de requisições atendidas por unidade de tempo. Gostaríamos que os valores medidos fossem limitados pela capacidade de processamento da CPU e não pela banda passante do canal de comunicação que, se saturada, pode deixar todas as tarefas simultaneamente bloqueadas em E/S. Porém, diversos fatores podem influenciar nessas medidas degradando o *throughput*, entre eles: a existência de outros processos rodando na máquina competindo por CPU e E/S, a mudança dinâmica de tamanho da janela TCP, o tamanho das filas de transmissão da interface de rede, o tamanho do *backlog* do TCP, a capacidade de transmissão e recepção da placa de rede, a velocidade

dos *hubs* e *switches*, o tráfego de rede além do gerado no teste, o volume de colisões na rede, o volume de pacotes que chegam na máquina e precisam ser descartados, a capacidade de processamento e E/S da máquina cliente que realiza os testes, a eficiência do *cache* de disco do SO, a periodicidade de coleta de recursos do SO.

5.3.1 Modo administrativo

No sistema Linux-Celeron e Windows-Celeron, onde foi possível isolar as máquinas envolvidas no teste em uma rede particular, tínhamos também o acesso como administrador. Com esse tipo de acesso, colocamos os sistemas em modo de manutenção, ativamos a interface de rede e, para o caso do Linux, ativamos o acesso remoto por terminal seguro pois utilizamos este para reiniciar automaticamente os servidores web. Assim garantiu-se que a máquina não estava executando serviços que não fossem essenciais para os testes. Essas medidas foram tomadas para minimizar existência de outras tarefas rodando na máquina competindo por CPU e E/S.

No sistema Linux-Celeron, aumentamos a fila de transmissão da interface de rede para um valor que permitisse com folga que o sistema atingisse a capacidade de transmissão máxima da rede (10000) pacotes. Nos outros sistemas deixamos as configurações originais.

No Linux-4processadores, não foi possível o acesso como administrador ou tampouco garantir o uso exclusivo da máquina, porém, esperamos que, ao repetir a bateria de testes 10 vezes, tenhamos amortizado o efeito de eventuais interferências.

5.3.2 Rede

Não se espera que se consiga utilizar 100% da banda em uma rede ethernet já que esse tipo de rede está sujeita a colisões, perdas e retransmissões de pacotes. Para garantir que o hardware pudesse suportar um volume de pelo menos 70% do máximo da rede, rodou-se o servidor web gerando uma página de 100 Mbyte e verificou-se a taxa de transmissão.

Para minimizar a influência do tráfego além do gerado pelo teste na rede, o volume de colisões na rede e o volume de pacotes que chegam na máquina e precisam ser descartados, o sistema Celeron e a máquina cliente foram ligados em um *switch* exclusivamente para eles. No caso do Linux-4processadores só foi possível garantir que as máquinas estavam ligadas no mesmo *switch*. Não foi possível garantir que pacotes não solicitados chegassem à máquina.

5.3.3
Capacidade dos clientes

Utilizou-se a máquina mais rápida disponível em cada rede como cliente dos testes. Para garantir que a máquina cliente tivesse capacidade de processamento e E/S para realizar os testes sem saturar, realizamos medições independentes de alguns pontos para monitorar a utilização de CPU do cliente durante essas medições. Verificou-se que o processo que realiza efetivamente o teste (o Apachebench) consome apenas uma pequena porcentagem da CPU e sobra processamento na máquina. Além disso, suprimiu-se toda a E/S durante o teste senão a E/S de rede necessária.

5.3.4
Coleta de lixo

Lua usa mecanismos de coleta de lixo para desalocar a memória que não está sendo mais usada. O algoritmo de coleta de lixo executa a coleta baseado em limites de tamanho de memória utilizada, calculados dinamicamente. Como o momento de coleta muda dinamicamente, esse tipo de mecanismo pode gerar momentos de muito processamento, difíceis de prever. Para evitar que a coleta de lixo influencie nas medidas, configuramos o sistema para realizar a coleta quando a quantidade de lixo atingir um valor que sabemos não será atingido (1Gbyte).

Com essa alteração o servidor não desaloca mais memória entretanto outros recursos podem ficar alocados sem necessidade, como por exemplo sockets. Os sockets que não são fechados graciosamente podem ficar ocupando recursos eternamente, até mesmo os *sockets* programados para verificarem se a conexão está realmente ativa, podem durar até duas horas sem que nenhum pacote seja transmitido. Assim como os *sockets,* outros recursos podem ficar alocados sem necessidade e comprometerem o desempenho da máquina. Por isso, a cada valor medido finalizamos e reiniciamos o servidor web para a medição de um novo valor.

5.3.5
Preparação antes da medição

Logo após remover os processos dos servidores e reiniciá-los, aguardamos um tempo (5 segundos) para o sistema coletar, sem competir processamento, todos os recursos estejam para ser liberados. Esperamos assim que o sistema se estabilize sem tarefas pendendes de execuções que não sejam as tarefas dos novos processos servidores. Imediatamente antes da medição propriamente dita realizamos alguns acessos ao servidor (30) na intenção do sistema estar pronto

e tudo o que for possível já colocado em cache e otimizado pelo servidor web e pelo SO. A *sandbox* Rings, por exemplo, faz *cache* de *strings* já compiladas em chamadas anteriores.

5.3.6 Tratamento dos dados

As medidas descritas acima visam minimizar o efeito de um bloco de variáveis que não são foco de interesse na análise dos sistemas, porém, existe um outro bloco de variáveis que não são facilmente influenciáveis e aparecem de forma aleatória nos resultados.

Cada bateria de testes levou de 1 a 16 horas, dependendo do sistema sendo testado. No total 4 sistemas foram testados. Devido à limitação de tempo para a obtenção das amostras, optou-se por gerar uma amostra de 10 pontos (somente os testes realizados com sucesso levaram aproximadamente 380 horas de teste). Essa amostra se mostrou pequena para uma análise com médias aritméticas pois o desvio padrão em alguns pontos chegou a 33% do valor da média aritmética.

Analisando qualitativamente os resultados, notamos que na maior parte dos pontos as amostras são bem parecidas. Numa tentativa de isolar os pontos com amostras consistentes, notamos que apesar do desvio padrão atingir valores acima de 6% do valor da média aritmética em 155 pontos, nos outros 385 pontos ele aparece bem comportado. Consideramos que os ruídos que geram o aumento do desvio padrão sempre aumentam o tempo de resposta das requisições, diminuindo o *throughput*. Logo podemos ver os maiores *throughputs* como os casos que sofreram pouco ruído e tiveram um comportamento próximo do ideal. A partir daí, para prosseguir a análise, geramos gráficos usando a amostra de maior valor para cada ponto. Esperamos que com esse ponto os gráficos traçados se aproximem de um comportamento ideal do sistema.