Uma análise do sucesso financeiro de gêneros de jogos para o design de jogos de sucesso

Luiz Fernando Bueno Rosa - RA: 221197
Instituto de Computação
Unicamp
Campinas, Brazil
l221197@dac.unicamp.br

I. Introdução

O projeto consistiu da simulação de tráfego para um balanceador de carga usando 3 diferentes políticas de balanceamento de carga para 3 servidores com capacidades de processamento diferentes de forma a avaliar a eficiência de cada política usada. As políticas usadas foram a *Round Robin*(variando o servidor escolhido de tal forma que cada servidor recebe a mesma quantidade de requisições), *Random Queue*(escolha de servidores aleatória) e *Shortest Queue*(escolha do servidor com menor fila de requisições).

a vazão (*throughput*) e o de resposta médio foram medidos e colidos ao final de cada simulação de tráfego e foram medidos a relação desses parâmetros com o número médio de pacotes(requisições) por ciclo e com o número médio de bytes recebidos por ciclo de tal forma a obter *insights* desses dados. Foram utiilzadas técnicas de linearização de variáveis do eixo x e regressão linear para observar tendências lineares nos dados obtidos.

II. Geração do tráfego

Cada tráfego é composto por $ciclos$, onde em cada ciclo foram geradas um conjunto de requisições utilizando-se de uma distribuição exponencial. O tamanho de cada requisição também foi gerado utilizando-se de uma distribuição exponencial, variando de 1 a 64kbytes (máximo da conexão TCP). A geração do tráfego está implementada como a função $trafficSimulator$ em no arquivo load-balancer.js.

Cada pacote gerado é criado pela função $createRequest(requestId, percentage, packetSize, cycle)$ que gera um pacote com os atributos $id, type, size$ e $loadReceiveTime$, sendo o primeiro para guardar seu identificador, $type$ é o percentual da requisição que envolve operações I/O, $size$ é o seu tamanho em bytes e $loadReceiveTime$ é o ciclo em que a requisição chegou no balanceador de carga. A distribuição exponencial é gerada pela função $generateExponentialSample$ em $ExponentialDistribution$ em sample.js.

III. Balanceador de Carga

O balanceador de carga é o responsável pelo balanceamento do tráfego, enviando cada pacote a um servidor de acordo com uma política de balanceamento. O balanceador de carga está implementado na função $LoadBalancer(load\_balancing\_policy, servers)$ do JavaScript em load-balancer.js. Para a sua construção, são necessários uma política de balanceamento $load\_balancing\_policy$ e dos servidores a terem suas cargas balanceadas $servers$. O balanceador possui um método $balanceLoad$ que distribui a carga para algum servidor e retorna o índice do servidor para log.

IV. Simulador do servidor

Cada servidor é instanciado através da classe $ServerSimulator$ em load-balancer.js que necessita de um $processor$ (processador) para ser construído. O processador é gerado pela função $processorTypes()$ load-balancer.js, que possuem alguns benchmarks de processadores reais, com suas taxas de processamento escaladas para baixo para que os tráfegos/logs pudessem ser pequenos.

A classe $ServerSimulator$ possui um contador global $ServerSimulator.cycle$ que conta o ciclo atual em que os servidores estão. A partir da taxa de processamento do processador do servidor, é calculado a quantidade de bytes que pode ser processada a cada ciclo multiplicando-se 4 pela taxa de processamento do serivdor.

A classe $ServerSimulator$ possui o método $receiveLoad(load)$ que enfileira o pacote na sua fila interna e $processLoad()$ que processa todas as requisições possíveis na fila (em order first-in-first-out) considerando seus bytes. O atraso é gerado através da fórmula $processing_size = 3 * first.size * first.type + 2 * first.size * (1 - first.type)$, onde o tempo de processamento é três vezes maior para a parte da requisição I/O e 2* maior para a parte da requisição de processamento. Enquanto há capacidade de processamento de bytes faltantes, o servidor continua processando as requisições da fila, até que uma requisição não possa ser totalmente processada, nesse caso o servidor guarda a quantidade de bytes que faltaram ser processados da requisição no seu campo $size$ (veja a linha 104 de load-balancer.js), ou que não há mais capacidade de processamento de bytes, fazendo com que o servidor encerre o seu ciclo.

A função $processLoad()$ também retorna uma tupla: o número de pacotes processados naquele ciclo e uma lista de todos os ids de pacotes processados naquele ciclo.

V. Simulador de testes

O simulador de testes é uma classe em JavaScript $TestSimulator$ em load-balancer.js que necessita de um balanceador de carga para ser construído. Ele possui um método $beginSimulation(traffic, servers)$ que recebe um tráfego e um servidor e realiza uma simulação, mantendo os atributo $throughput$, que guarda o número de requisições realizadas e $response\_times$, um objecto que guarda o ciclo de chegada e saída de cada pacote, identificado por um id.

O método $beginSimulation(traffic, servers)$ realiza o teste de uma política de balanceamento em um tráfego. Ele gera e retorna um log, um objeto que salva, para cada ciclo, para qual servidor cada pacote identificado pelo seu id foi enviado. Esse método roda num while que para quando todos os 100 ciclos forem executados, incrementando $ServerSimulator.cycle$ no final de cada ciclo para sincronizar os servidores. E para cada pacote no tráfego sendo recebido no ciclo em execução, o método chama o balanceador de carga para balancear a carga e chama a função $processLoad()$ de cada um dos servidores, atualizando $response\_times$, o $throughput$ e o log. No final essa função retorna log.

VI. Execução do programa - main

O programa executa a função $generateTrafficInstances()$, gerando 1000 instâncias de tráfego e as salvando na pasta traffic/. Após isso, os servidores e os balanceadores de carga são instanciados e 3 fors são executados, cada um executando a simulação em cada um dos tráfegos gerados. Cada log é retornado da simulação do tráfego, sendo appendados a eles o throughput e o tempo de resposta médio. Eles são salvos na pasta log, identificando a política e o tráfego usados.

No fim, o cache de cada servidor é limpado através do método $cleanQueue()$.

VII. Análise dos dados

A. Extração dos dados

Os dados de cada simulação são extraídos. Os dados usados no eixo x são a média de pacotes por ciclo e a média de bytes por ciclo de cada tráfego. Esses dados são utilizados igualmente em todas as análises de política. Os dados do eixo y são o throughput e o tempo de resposta médio de cada simulação para cada política utilizada.

B. Análise dos dados

Primeiramente, os dados foram plotadaos normalmente utilizando-se uma regressão linear e o scatter do pyplot da biblioteca matplotlib e depois uma versão foi plotada usando-se a escala logarítimica. A implementação da geração dos gráficos está na função $main()$ em data-analysis/data-analysis.py.

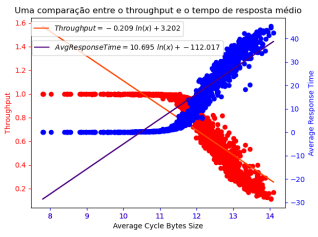
Após ser identificado uma relação logarítmica entre os dados, percebeu-se que existia uma relação linear na escala log(x) por Y. Assim, foi feita a aglomeração dos gráficos linearizados no eixo X do throughput e do tempo de resposta médio (veja figuras na seção Imagens) geradas na pasta data-analysis/plots/xy/. Para gera-las, basta executar o código data-analysis/data-analysis.py pelo terminal (chamar na pasta root).

Foi observado que a tendência era a mesma sobre ambos os domínios em x. A partir de um certo ponto, o throughput (em vermelho) começa a diminuir e o Tempo de resposta médio (em azul) começa a aumentar.

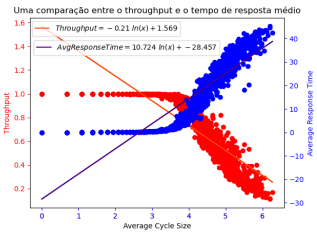
Os dados de linearização abaixo não forneceram muitos insights sobre a política mais eficiente. Os coeficientes estão muito próximos das outras políticas e a política de escolha aleatória foi a vitoriosa, por uma pequena margem, quando se esperava que a política de menor fila fosse a mais eficiente.

Y	Política	x	A	B
Throughput	rq	Média pacotes	-0.21	1.569
Avg. Response Time	rq	Média pacotes	10.724	-28.457
Throughput	rq	Média bytes	-0.209	3.202
Avg. Response Time	rq	Média bytes	10.695	-112.017
Throughput	rr	Média pacotes	-0.211	1.575
Avg. Response Time	rr	Média pacotes	10.824	-28.779
Throughput	rr	Média bytes	-0.21	3.216
Avg. Response Time	rr	Média bytes	10.794	-113.113
Throughput	sq	Média pacotes	-0.21	1.574
Avg. Response Time	sq	Média pacotes	10.765	-28.778
Throughput	sq	Média bytes	-0.209	3.208
Avg. Response Time	sq	Média bytes	10.736	-112.656

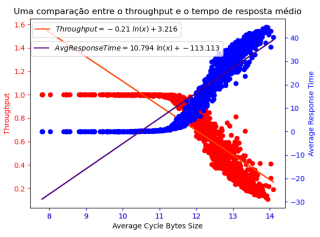
Imagens



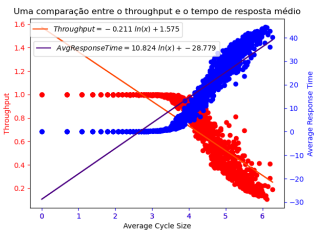
Random Queue sobre média de bytes



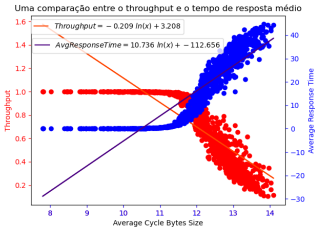
Random Queue sobre média de pacotes



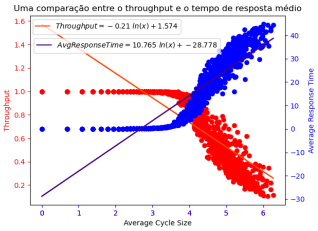
Round Robin sobre média de bytes



Round Robin sobre média de pacotes



Shortest Queue sobre média de bytes



Shortest Queue sobre média de pacotes