Sistema de alocação de Máquinas Virtuais em pool de recursos

José Pirangaba da Silva Neto
Ciência da Computação
UFAPE
Garanhuns, Brazil
pirangabaneto@gmail.com

Laisy Cristina Ferreira Silva
Ciência da Computação
UFAPE
Garanhuns, Brazil
laisyferreira2011@gmail.com

Manoel Natalicio Cavalcante Braga Filho
Ciência da Computação
UFAPE
Garanhuns, Brazil
nataliciocavalcantebraga@gmail.com

Marcos Vinicius Valério Silva Filho
Ciência da Computação
UFAPE
Garanhuns, Brazil
ortigol2010@gmail.com

Matheus da Silva Noronha
Ciência da Computação
UFAPE
Garanhuns, Brazil
noronhamatheus07@gmail.com

Resumo*—Este projeto propõe a criação de um algoritmo de alocação de máquinas virtuais em um pool de recursos, os quais considera um servidor, para cada cliente que se conecta a esse servidor criamos um novo processo, para assim ele atender a vários clientes. Cada cliente tem uma lista com todos os servidores disponíveis (estático), eles podem pegar uma tarefa (soma de todos os itens de uma matriz, por exemplo), escolhe servidores de forma aleatória e divide as partes dessa tarefa entre os servidores.

Palavras-chave*—alocação de Máquinas Virtuais, Sistemas Operacionais, Computação em Nuvem.

I Introdução

O uso máquinas virtuais tem sido grandemente empregado nos últimos anos objetivando, principalmente, a economia de recursos computacionais e o isolamento entre usuários e aplicações, para fins de segurança. O principal contexto de emprego de máquinas virtuais (VMs) é na arquitetura de sistemas operacionais.

A virtualização possibilita otimizar o uso da infraestrutura de TI, incluindo servidores, armazenamento e os dispositivos e rede. O centro de processamento de dados virtual pode ser provisionado para o negócio com o conceito de infraestrutura compartilhada virtualmente. Os principais recursos dessas arquitetura são: pool de recursos, recursos de armazenamento e recursos de conectividade.

Os servidores virtuais representam os recursos virtuais de processamento e memória de um servidor físico rodando o software. O servidor é onde estão alojadas as máquinas virtuais. Os recursos de processamento e memórias incluídos em servidores são particionados em uma hierarquia de pool de recursos.

As máquinas virtuais (Vms) são associadas a um servidor virtual aleatoriamente, um pool de recursos. Os recursos são provisionados para as Vms baseados nas políticas definidas no sistema.

II Objetivos

Temos como objetivo criar um sistema de alocação de máquinas virtuais em um pool de jobs (recursos), utilizando conceitos de threads e/ou processos. Onde a conexão para submissão desses jobs, deve ocorrer de forma randômica e com menor tempo de processamento possível. Utilizando a linguagem de programação python.

III Processo X Threads

Quando trabalha-se com tecnologia, compreender o significado dos termos threads e processos, pode ser de uma ajuda significativa. Tais conhecimentos auxiliam no entendimento dos gerenciadores de atividade do sistema operacional, resolvem quais programas estão causando problemas no computador e se existe a necessidade de instalar mais memória para fazer o sistema funcionar melhor.

A. Processo

Um processo em execução requer memória e alguns recursos do sistema operacional para ser executado. Ele é carregado na memória com todos os recursos de que precisa para ser executado. O sistema operacional é o "cérebro" que aloca todos esses recursos, e possui diferentes estilos. O sistema operacional é responsável pela tarefa de gerenciar os recursos necessários para tornar o programa um processo em execução.

Um programa pode ter várias instâncias, e cada instância do programa em execução é um processo. Cada processo possui um espaço de endereço de memória separado, o que significa que um processo é executado de forma independente e isolado de outros processos. Ele não pode acessar diretamente os dados compartilhados em outros processos. Mover de um processo para outro requer tempo para salvar e carregar registradores, mapeamento de memória e outros recursos.

A independência do processo é muito útil, porque o sistema operacional fará o possível para isolar o processo, de forma que o problema de um processo não danifique ou cause danos a outro.

B. Thread

Uma thread é uma unidade de execução dentro de um processo. Um processo pode ter vários threads. Após o início do processo, memória e recursos serão alocados. Cada thread no processo compartilha essa memória e recursos. Em um processo de thread único, o processo contém uma thread, ambos são os mesmos e apenas uma coisa está acontecendo. Em um programa multithread, o processo contém vários threads e executa muitas operações ao mesmo tempo.

Cada thread terá sua própria pilha, mas todos os threads no processo compartilharão o heap, memória alocada dinamicamente.

Eventualmente, Threads são chamados de processos leves porque têm sua própria pilha, mas podem acessar

dados compartilhados. Como a thread compartilha o mesmo espaço de endereço com o processo e outras threads no processo, o custo operacional de comunicação entre as threads é menor, o que é uma vantagem. A desvantagem é que um problema com uma thread no processo afetará definitivamente a viabilidade de outros threads e do próprio processo.

C. *Multi Processamento X Multithreading*

Multiprocessamento e Multithreading diferente na forma como as instruções do programa são processados. Em multiprocessamento, duas ou mais instruções podem ser executadas de forma simultânea, através de um processo chamado "processamento paralelo". Multithreading, no entanto, abre "caminhos" individuais para cada instrução, e distribui o poder de processamento entre esses caminhos periodicamente.

Multithreading usa várias threads de um único processo, assim pode-se particionar a tarefa, para melhorar a computação, diminuindo a alocação de recursos do sistema e o tempo de troca de execução é menor que um processo.

A utilização de threads traz diversas vantagens para o usuário, já que podem aumentar o desempenho das aplicações. Permitindo que as tarefas possam ser executadas em paralelo ao fluxo principal, assim possibilitando que a tarefa continue acessível ao usuário.

No entanto, no multi processamento abre processo para executar múltiplas funções simultâneas e cada processo deste obtém sua própria parcela de memória. Quando um único processo falhar, neste ambiente, todos os outros processos são poupados.

IV Relação Cliente-Servidor

Explicaremos alguns conceitos importantes relativos à comunicação de processos via rede de computadores, especificamente a tecnologia de soquetes, a qual utilizamos durante a implementação deste trabalho.

A. *Sockets*

Um Socket provê a comunicação entre dois dispositivos conectados a rede, conhecido também como two-way communication, essa comunicação pode ser feita entre dois processos que estejam na mesma máquina (Unix Socket) ou na mesma rede (TCP/IP). Na rede, a representação de um socket se dá por ip:porta.

Os sockets TCP/IP, são uma abstração para endereços de comunicação nos quais os processos se comunicam através de um identificador único composto pelo endereço da máquina e a porta que será usada para entrada do processo, Essa porta é utilizada para mapear os dados recebidos pela máquina para os processos específicos.

A conexão com o servidor é feita colocando o servidor em loop, ouvindo às requisições que chegam até ele através de uma determinada porta, o cliente precisa conhecer o endereço do servidor previamente, então o cliente e o servidor estabelecem uma conexão direta entre eles em uma nova porta designada pelo servidor, criando assim um canal de comunicação entre o cliente e servidor.Um Socket provê a comunicação entre dois dispositivos conectados a rede, conhecido também como two-way communication, essa comunicação pode ser feita entre dois processos que estejam na mesma máquina (Unix Socket) ou na mesma rede (TCP/IP). Na rede, a representação de um socket se dá por ip:porta.

Os sockets TCP/IP, são uma abstração para endereços de comunicação nos quais os processos se comunicam através de um identificador único composto pelo endereço da máquina e a porta que será usada para entrada do processo, Essa porta é utilizada para mapear os dados recebidos pela máquina para os processos específicos.

A conexão com o servidor é feita colocando o servidor em loop, ouvindo às requisições que chegam até ele através de uma determinada porta, o cliente precisa conhecer o endereço do servidor previamente, então o cliente e o servidor estabelecem uma conexão direta entre eles em uma nova porta designada pelo servidor, criando assim um canal de comunicação entre o cliente e servidor.

V Problemas Enfrentados

Aqui explicaremos as tentativas feitas, os problemas enfrentados e o que aproveitamos disso.

A. *Thread do lado do Servidor*

Utilizamos threads do lado do servidor, para que vários clientes possam se conectar a um único servidor.

B. *Thread do lado do Cliente*

Tentamos utilizar threads do lado do cliente, porém não funcionou. Para criar threads do lado do cliente, para um cliente se conectar a vários servidores, precisamos de sub clientes, pois cliente é um socket e cada socket possui somente uma porta, e como estamos utilizando TCP IP, só temos conexão de um para um, ou seja, um cliente só poderia se conectar a um servidor. Para, superar este problema, criamos sub clientes dentro do cliente principal, os quais se conectam a servidores distintos.

C. *Multiprocessos sem servidores*

A biblioteca de multiprocessamento do python nos permite criar processos em paralelo, utilizando diferentes núcleos do cliente. Executando este código, utilizando multiprocessos do python sem servidores, conseguimos ter um código rápido e que funciona, porém não é o objetivo do projeto.

D. *Divisão de job para diferentes Clientes*

Utilizar a mesma conexão de um servidor conectado a múltiplos clientes e esse servidor pegando um job e separando para os diferentes clientes que estão conectados.

VI Funcionamento do sistema

O funcionamento do sistema se dá da seguinte forma: o cliente tem uma função f_n que calcula a soma de 1 a um número muito grande (1 bi), tem uma função que chama essa função f_n e calcula 3 vezes, em seguida retorna o tempo de processamento que demorou. Temos um array estático que contém os dados de todos os servidores da rede. Criamos novos processos e adicionamos nesse array.

Cada processo chama a função que escolhe o servidor, ela cria um novo socket e pega o número do servidor, associa cada servidor a um processo. Ele dá set no host e na porta, e cria uma conexão com o servidor.

A. *Servidor Multi Threads*

Afim de criar uma rede de comunicação capaz de suportar a relação "um servidor - vários clientes", utilizamos threads. Uma vez que o servidor esteja ativo, temos um processo em execução, como mostra a figura 01, o servidor está sempre à espera da conexão com um cliente e, assim que o cliente realiza uma requisição de conexão, o servidor inicia uma nova thread, a qual é adicionada ao espaço de memória do processo deste servidor. Desta forma possibilitamos que um servidor, de forma concorrente, mantenha conexão com vários clientes.

```
#loop que torda o servidor sempre disponivel pra toda solicitacao
while True:
    #sempre aceita uma coneccao
    Client, address = ServerSideSocket.accept()
    print('Connected to: ' + address[0] + ':' + str(address[1]))

    #comeca uma nova thread pra esse novo cliente conectado
    start_new_thread(multi_threaded_client, (Client, ))

    #conta quantos clientes esetao conectados
    ThreadCount += 1
    print('Thread Number: ' + str(ThreadCount))
ServerSideSocket.close()
```

Figura 01. trecho do código servidor multi threads

B. *Cliente Multi Threads*

De forma similar ao servidor multi threads, para que um cliente possa se conectar a vários servidores, a fim de dividir uma tarefa que, localmente, requer um espaço de tempo significante, foram implementadas threads do lado do cliente, as quais, são responsáveis por dividir essa atividade em atividades menores e, teoricamente deveriam executar de forma mais rápida se comparadas a execução sequencial.

A execução de um sistema multi thread executa de forma concorrente e não paralela, tal concorrência, quando somada aos delays de conexão, que precisa ser estabelecida, resultaram em um resultado insatisfatório, o qual, por fim, requer mais tempo que a execução sequencial no próprio cliente. Com o objetivo de averiguar se o delay de conexão era o que tornava a utilização de threads mais lenta que a execução sequencial, foram implementadas, também, threads somente no cliente, como mostra a figura 02 e, no melhor dos casos obteve-se tempo igual a uma execução sequencial, raramente obtendo-se resultado com tempo menor.

```
def TEST_THREADS():
    new_thread1 = threading.Thread(target=fn, args=())
    new_thread2 = threading.Thread(target=fn, args=())
    new_thread3 = threading.Thread(target=fn, args=())
    new_thread1.start()
    new_thread2.start()
    new_thread3.start()
    new_thread1.join()
    new_thread2.join()
    new_thread3.join()
```

Figura 02. trecho do código cliente multi threads

C. *Multiprocessos do lado do Cliente*

Implementamos a utilização de multiprocessos no cliente, como mostra a figura 03.

Para cada servidor - aqui identificados no array nomeado "dados_conecao" - é criado um processo, o qual chama o método "escolherServidor" - figura 04 - onde instancia um *socket* que é conectado a um servidor disponível. A criação de um novo socket para cada processo se faz necessária pois uma instância de socket que utiliza protocolo TCP pode conectar-se com apenas 1 servidor, logo, a necessidade de vários sockets.

```
def TEST_MULTIPROCESSING():
    threads = []
    for i in range(len(dados_conecao)):
        threads.append(multiprocessing.Process(target=escolherServidor, args=(i,)))

    for t in threads:
        t.start()

    for t in threads:
        t.join()
```

Figura 03. Multiprocessos no cliente

```
def escolherServidor(n):
    global dados_conecao
    ClientMultiSocket = socket.socket()
    servidor = dados_conecao[n]

    host = str(servidor[0])
    port = servidor[1]

    ClientMultiSocket.connect((host, port))

    ClientMultiSocket.recv(1024)
    ClientMultiSocket.close()
```

Figura 04. chamada socket

VII RESULTADOS

Para comparar a eficiência das abordagens aqui apresentadas, executamos o algoritmo apresentado na figura 05 de 4 maneiras distintas (figura 06). Primeiro executamos o método de forma isolada, para saber o tempo de duração do método - linha 66 - que durou 4.3s ; 3 vezes de forma sequencial - linha 67 - com duração de 11.55s; multithreads - linha 68 - 10.62s; e, finalmente, multiprocessos com 3 servidores - linha 69 - que teve tempo total de duração de 5.83s.

Com estes dados foi possível mensurar o quão mais rápido conseguimos executar esta tarefa utilizando multiprocessos. Conseguimos, com multiprocessos, executar esta tarefa (figura 05) 49.52% vezes mais rápido se comparado a execução sequencial.

```
def fn():
    x = 0
    while x < 10000000:
        if (x == 9999999):
            print("chegou ao topo")
        x += 1
```

figura 05. trecho do código que compara eficiência das abordagens.

```
print("Time to Run 1x: %0.2fs" % (timeit.timeit(fn, number=1),))
print("NORMAL:%0.2fs" % (timeit.timeit(TEST_NORMAL, number=1),))
print("Threaded: %0.2fs" % (timeit.timeit(TEST_THREADS, number=1),))
print("Multiprocessing: %0.2fs" % (timeit.timeit(TEST_MULTIPROCESSING, number=1),))
```

Figura 06. formas de execução do código

VIII CONCLUSÃO

Este projeto investigou o problema de alocação de VMs em um pool de recursos, considerando políticas

definidas no sistemas e seguindo protocolo TCP-IP, com a utilização de multiprocessos e multithreads.

Embora encontrada dificuldades para finalizar o sistema e não tendo utilizado máquina virtuais, conseguimos obter um bom resultado, com rapidez de execução nas tarefas, utilizando multiprocessos.

IX Trabalhos Futuros

A utilização de execução paralela de processos se mostrou ser de uma eficiência indiscutível. Visamos, em um projeto futuro, implementar a gerência automática de servidores, que hoje se encontra estática; testar compartilhamento de memória entre os processos; e multiprocessamento no servidor, ao invés de threads.

Referencias

Veras. M., Vitualização: tecnologia central do Datacenter. 2° edição. Brasport. 2015.

Gerenciamento de regras de afinidade de Host da VM. VMWARE Docs. 31-05-2019. Disponível em: <https://docs.vmware.com/br/VMware-Cloud-Director/9.7/com.vmware.vcloud.admin.doc/GUID-1462624A-E0F5-4040-BC60-CD6A5FA80439.html> Acesso em: 06-10-2020.

Synchronization and Pooling of processes in Python. GeeksforGees. 13-02-2018. Disponível em: <https://www.geeksforgeeks.org/synchronization-pooling-processes-python/>. Acesso em: 19-10-2020.

Oakes, E. How to scale Python multiprocessing to a cluster with one line of code. Distributed Computing with Ray. Feb 18. Disponível em: <https://medium.com/distributed-computing-with-ray/how-to-scale-python-multiprocessing-to-a-cluster-with-one-line-of-code-d19f242f60ff>. Acesso em: 08-10-2020.

Stegh Camati, R. Novos Algoritmos para Alocação de Máquinas Virtuais e um Novo Método de Avaliação de Desempenho. Dissertação de Mestrado, Ciência da Computação, Universidade Católica do Paraná. Curitiba. 131 pág. 2013.

Kramer, L. Configerando um ambiente de desenvolvimento com Vagrant. Ship it! 07-07-2015. Disponível em: <http://shipit.resultadosdigitais.com.br/blog/configurando-um-ambiente-de-desenvolvimento-com-vagrant/>. Acesso em: 27-10-2020.

Socket - Low-Level networking interface. Python.org. Disponível em <https://docs.python.org/3/library/socket.html>. Acesso em: 13-10-2020.

Back, R. What's the Diff: Programs, Processes, and Threads. BackBlaze. 16-08-2017. Disponível em:<https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/>. Acesso em: 13-10-2020.

Difference Between Multithreading vs Multiprocessing in Python. GeeksforGeeks. 10-05-2020. Disponível em:<https://www.geeksforgeeks.org/difference-between-multithreading-vs-multiprocessing-in-python/>. Acesso em: 19-10-2020.

Multiprocessing - Process-based parallelism. Python.org. Disponível em: <https://docs.python.org/3.4/library/multiprocessing.html?highlight=process>. Acesso em: 13-10-2020.