Virtualização em *datacenters* utilizando o *Xen*

Vinícius Lopes da Silva

***Abstract*—Em tempos onde a sustentabilidade torna-se palavra-chave para usar os recursos disponíveis de maneira eficiente, a virtualização ressurge como uma forma de aumentar o uso do *hardware* sem que isso acarrete em mais uso de recursos energéticos. O conceito de virtualização é antigo e data desde a década de 60, mas suas bases teóricas só foram estabelecidas em 1974, quando Robert P. Goldberg e Gerald J. Popek publicaram um artigo intitulado *Formal requirements for virtualizable third generation architectures*. Neste artigo, serão apresentadas as diferentes formas de virtualização, como a virtualização funciona, como a virtualização tem sido adotada dentro dos *datacenters* e como ela tem melhorado a eficiência dos *datacenters*. Existem diversos virtualizadores disponíveis para uso, mas neste artigo, será focado no uso do virtualizador Xen para a arquitetura x86.**

***Palavras-chave*—Virtualização, Xen, Datacenter, Eficiência energética, Sustentabilidade, Sistemas operacionais, Sistemas distribuídos, Máquina Virtual, Xenoserver.**

I. INTRODUÇÃO

A VIRTUALIZAÇÃO teve suas bases teóricas definidas em 1974. O conceito de virtualização é o de criar uma versão virtual de alguma coisa. Computadores modernos implementam a virtualização de uma maneira mais simplificada. Um computador *single-core* deve executar um processo por vez para dar a ilusão para o usuário de que existe mais de um programa em execução, nesse caso, pode-se dizer que o processador é virtualizado. Um processo também tem a ilusão de que ele possui toda a memória do computador disponível para ele, mas na verdade, existem vários outros processos que estão usando a memória, neste caso, também pode-se dizer que a memória é virtualizada.

O desejo de usar os recursos computacionais ao máximo existe desde a década de 60, assim como os *datacenters* de hoje, os computadores (tipicamente *mainframes*) daquela época sofriam com uma má utilização de seus recursos, que dificilmente alcançavam um pico de uso. Para tirar maior proveito dos *mainframes* a IBM passou a investir em técnicas de virtualização, de forma a tornar seus *mainframes* multitarefa e consequentemente usando o máximo de seus recursos computacionais por mais tempo.

Mesmo sendo um conceito antigo, a virtualização nunca esteve tão na moda como atualmente. Com o surgimento da *cloud computing*, surgiram os *datacenters* que são conjuntos de máquinas de alto desempenho, que geralmente são usadas por empresas de TI para fornecer algum tipo de serviço na *web*. Como se pode imaginar, gerenciar um *datacenter* não é uma das tarefas mais fáceis de se fazer e não é apenas a parte gerencial que é complicada, *datacenters* consomem uma enorme quantidade de energia. Por isso, é interessante que seja possível tirar o maior proveito possível dos recursos que se tem disponíveis, ou seja, ao invés de ter apenas um único servidor em uma única máquina, seria melhor ter o maior número possível de servidores nessa única máquina. Através da virtualização, esse e outros cenários são possíveis de ser realizados, por isso a virtualização tornou-se uma tecnologia fundamental dentro dos *datacenters*.

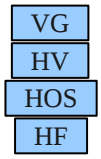
II. CONCEITOS DE VIRTUALIZAÇÃO

A virtualização possui dois conceitos chave, o de máquina virtual (MV) e o de Monitor de Máquina Virtual (MMV). De acordo com Goldberg e Popek em *Formal Requirements for virtualizable third generation architectures*, uma MV é uma réplica isolada e eficiente de uma máquina real. Por isolada, deve-se entender que a MV tem a ilusão de que ela detém os recursos do *hardware* somente para ela, ou seja, ela deve ter a ilusão de que é a única que está sendo executada na máquina real. Por eficiente, deve-se entender que a MV deve apresentar um desempenho próximo ao desempenho de um processo que é executado diretamente na máquina real (veremos posteriormente que a segunda condição é mais difícil de ser atingida). A MV é um ambiente criado pelo MMV. O MMV é uma camada intermediária entre a MV e o *hardware*. O MMV constrói um ambiente onde os *Virtual Guests* (tipicamente sistemas operacionais) podem usar o recurso computacional como se fossem os únicos donos daquele recurso e como se o acesso a esse recurso fosse feito de maneira direta, sem intermediários.

III. FORMAS DE VIRTUALIZAÇÃO

Existem diferentes abordagens adotadas para se alcançar a virtualização de um sistema. Apesar de diferentes, essas abordagens possuem algumas semelhanças. Atualmente, existem quatro tipos de virtualização capazes de garantir a execução de mais de um sistema operacional na mesma máquina. Apesar da virtualização não se restringir a executar mais de um sistema operacional em uma mesma máquina, neste artigo a virtualização será tratada de tal forma. Segue abaixo uma descrição dos quatro tipos de virtualização:

1. Emulação: Na emulação, a máquina virtual simula um *hardware* completamente diferente daquele em que ela se encontra para poder executar um *Virtual Guest* (VG) no *hardware* que ela implementa sem qualquer tipo de modificação no VG. A figura 1 apresenta uma hierarquia de como funciona a emulação.



VG: Virtual guest
HV: Hardware virtual
HOS: Host operating system
HF: Hardware físico

Figura 1.

Nem o VG e nem o HV tem consciência do HF ou do

[] Este trabalho foi desenvolvido para a disciplina de Tópicos Avançados em Redes de Computadores.

Vinícius Lopes da Silva é estudante do sétimo semestre da UFSCar – *campus* Sorocaba.

Professor da disciplina: Fabio Verdi.

HOS. Exemplos de emulador são o Bochs e a versão não-acelerada do QEMU.

2. Virtualização completa: A virtualização completa é similar a emulação. A única diferença entre a emulação e a virtualização completa é que a máquina virtual simula o mesmo *hardware* da máquina em que ela se encontra. Isso aumenta o desempenho da máquina pois permite que o sistema que está na máquina virtual possa executar as instruções diretamente no *hardware* (com exceção das instruções privilegiadas, que precisam ser emuladas pela própria máquina virtual). A figura 2 mostra um esquema de como funciona a virtualização completa.

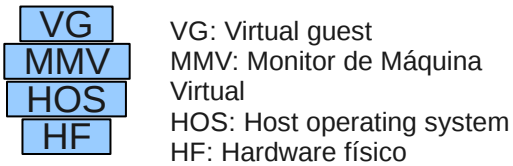


Figura 2.

É importante ressaltar que apesar do MMV rodar em cima do HOS, o VG executa as instruções não-privilegiadas diretamente no HF, o MMV serve apenas para verificar se a instrução que o VG deseja executar é privilegiada e se for, ele vai emular essa instrução. Exemplos de sistemas que implementam a virtualização completa são os sistemas da VMware e a versão acelerada do QEMU (KQEMU).

3. Paravirtualização: A paravirtualização adota uma abordagem diferente das utilizadas pela virtualização completa e pela emulação, nelas, o VG executa na máquina virtual sem sofrer qualquer tipo de modificação, ou seja, para eles é transparente o uso do *hardware.* Na paravirtualização é necessário que o sistema operacional (potencial VG) seja modificado para que ele não possa executar instruções privilegiadas diretamente no *hardware*, assim, é necessário solicitar o uso da instrução privilegiada para o MMV. A figura 3 mostra um esquema de como funciona a paravirtualização.

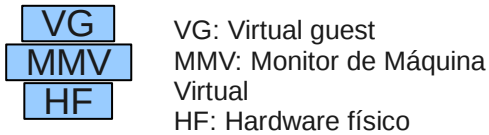


Figura 3.

É bom ressaltar que na paravirtualização não existe um HOS, permitindo que a MMV execute diretamente no *hardware*, diferente da emulação e da virtualização completa, em que se faz necessário um HOS. Como o MMV tem contato direto com o HF e ele também monitora todos os VGs que estão em execução, o MMV na paravirtualização também é chamado de *hypervisor* (hipervisor). Um exemplo de sistema que implementa a paravirtualização é o Xen, que será discutido em maiores detalhes neste artigo, expondo mais conceitos da paravirtualização.

4. Virtualização a nível de sistema operacional: Neste tipo de virtualização, não existe um MMV propriamente dito, ao invés disso, o MMV é substituído por um sistema operacional. Os sistemas operacionais usados para garantir esse tipo de virtualização devem prover um sistema de tempo compartilhado, ter propósito geral e garantir um forte isolamento de recurso e de espaço de nomes. A figura 4 mostra um esquema de como funciona a virtualização a nível de sistema operacional.

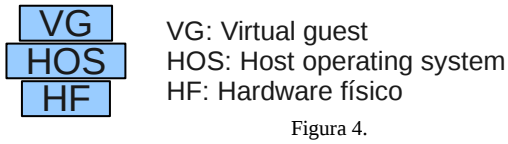


Figura 4.

Os VGs criados nesse tipo de virtualização utilizam o *kernel* do próprio sistema operacional. Neste tipo de virtualização os VGs consomem menos recursos do que nos outros tipos, por outro lado, é necessário que o VG utilize o mesmo *kernel* que o do sistema operacional, tirando um pouco da flexibilidade da virtualização. Exemplo de sistema que implementa a virtualização a nível de sistema operacional é o FreeBSD Jails.

Além desses quatro tipos de virtualização, existem outros dois, que não permitem a execução de um sistema operacional, mas são dignos de menção devido a serem implementadas em sistemas usados com uma certa frequência. Segue abaixo uma descrição desses dois tipos de virtualização:

1. Virtualização de biblioteca: Neste tipo de virtualização ocorre a emulação de partes de um sistema operacional, permitindo que aplicações desenvolvidas para um sistema operacional possam ser executadas em outro sistema operacional. Um exemplo de sistema que realiza esse tipo de virtualização é o Wine.

2. Virtualização de aplicação: Neste tipo de virtualização a aplicação é executada em um ambiente virtual, esse ambiente é montado de forma a parecer com o ambiente de execução de processos, criado por um sistema operacional. Um exemplo de sistema que realiza esse tipo de virtualização é a Máquina Virtual Java.

IV. Por que o Xen ?

Como dito anteriormente, este artigo tem o propósito de explicar a virtualização em *datacenters* através do virtualizador Xen. Existem diversos virtualizadores disponíveis, mas o Xen é o que tem sido o mais adotado nas grandes empresas de *cloud computing*. Entre elas pode-se citar *Amazon*, *Cloud.com*, *GoGrid* e *Rackspace*. Essas empresas escolheram o Xen devido a sua simplicidade e eficiência. O Xen separa muito bem mecanismos de políticas administrativas, ele não possui *drivers* de dispositivo e usa apenas 2MB de memória, ele é uma solução gratuita e *open source.* Diferente da maioria dos virtualizadores, o Xen adota a abordagem da paravirtualização, isso permite que seu desempenho seja em média 95% daquele obtido com um sistema nativo. Por esses motivos, o Xen foi escolhido para ser explicado neste artigo. Devido o Xen ser um

virtualizador, ele possui muitos detalhes, o que o torna muito extenso. Este artigo não tem a intenção de explicar todos os detalhes de funcionamento do Xen, apenas os aspectos relevantes para que ele seja usado em *datacenters*.

V. As origens do xen

O Xen tem suas origens em 2001, como parte do projeto Xenoserver, idealizado no laboratório de computação da Universidade de Cambridge e ainda em fase de desenvolvimento. O projeto Xenoserver tem como objetivo construir uma infraestrutura pública que possa ser utilizada para a computação distribuída. Essa infraestrutura será sustentada pelas plataformas de execução do Xenoserver que estarão espalhadas pelo mundo inteiro e poderão ser usadas por qualquer pessoa que esteja conectada a internet. Quando o projeto estiver completo, os usuários da plataforma montada poderão submeter seu código para rodar na plataforma e serão cobrados pelos recursos que o programa consumiu. Para garantir que os nós do Xenoserver sejam utilizados ao máximo, foi construído o Xen *Hypervisor*. *O* Xen dentro do projeto Xenoserver não apenas garantiria a execução de múltiplos sistemas nos nós, mas também poderia controlar os recursos consumidos por cada sistema. A princípio, os nós utilizados no projeto Xenoserver possuíam arquitetura x86 e portanto o Xen foi concebido para essa arquitetura, entretanto, com o crescimento do Xen, ele foi portado para as arquiteturas x86-64, Itanium, Power PC e ARM. O Xen também suporta a execução de diversos sistemas operacionais, entre eles, pode-se citar Linux, FreeBSD, NetBSD, Solaris e Windows. Neste artigo será focado o uso do Xen para a arquitetura x86, não apenas por ser a arquitetura mais utilizada em servidores e *desktops*, mas também por ele ter sido desenvolvido inicialmente para essa arquitetura.

VI. O Xen Hypervisor

O Xen é um MMV que implementa os conceitos da paravirtualização. O Xen se situa entre o *hardware* da máquina real e os VG, que na terminologia usada pelo Xen são chamados de *Guest Domains* (GD) ou simplesmente de *Domains*. Os GDs são sistemas operacionais modificados para poder executar no ambiente virtual criado pelo Xen. Em uma arquitetura x86, existem quatro níveis de execução, representadas por anéis. Na figura 5 é apresentado como esses níveis são representados.

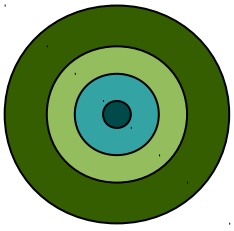


Figura 5.

O círculo mais interno situa-se no nível 0 e garante acesso total ao *hardware*, esse nível é também chamado de *real mode*. O círculo mais externo situa-se no nível 3 e ele chama-se *protected mode*, pois não sofre interferências dos outros níveis. Usualmente, o sistema operacional é executado no *real mode* (nível 0) e todo o resto é executado no nível 3. Quando se utiliza a paravirtualização na arquitetura x86, é necessário que o sistema operacional seja modificado para que ele deixe de residir no anel 0 e passe para o anel 1, quem assume o lugar no anel 0 é o virtualizador, no caso, o Xen.

VII. Os *Guest Domains*

Por executar no nível 0, pode-se imaginar que a camada que o Xen implementa é equivalente ao de um sistema operacional, mas na verdade ele é bem menor do que isso. Isso ocorre graças a um GD especial, chamado de *Domain0* (*Dom0*). O *Dom0* é o primeiro GD carregado durante a inicialização do Xen. O *Dom0* é o único GD que possui privilégios, pois ele é responsável por apresentar uma interface para realizar tarefas administrativas (como criar e configurar outros GDs) e também por ter acesso direto sobre o *hardware* da máquina. Apesar de ser o único GD privilegiado, o *Dom0* pode delegar a função de controlar um dispositivo específico para um outro GD, esse GD é chamado de *Driver Domain* (DD). Os outros GDs são referenciados por *DomU* onde o U é abreviação para *Unprivileged* (sem privilégio). Em algumas situações, é interessante que os GDs possam se comunicar um com o outro, para conseguir isso é utilizado o *XenStore*, que será explicado na próxima seção.

VIII. Controle de Dispositivos no Xen

Em um sistema paravirtualizado, os sistemas operacionais não podem mais controlar o *hardware*, portanto, quando eles desejam realizar alguma operação de E/S é necessário comunicar o Xen da operação a ser feita. O Xen, diferente dos sistemas operacionais, não implementa *drivers*, consequentemente, não tem como ele atender a requisições de E/S. A abordagem adotada pelo Xen para que os GDs tenham acesso ao dispositivo chama-se de *split-driver model*. Essa abordagem deixa a função de controlar o dispositivo para o *Dom0* ou para um DD e caso algum GD queira realizar alguma operação de E/S, ele deve comunicar o GD que controla aquele dispositivo. É responsabilidade do *Dom0* ou do DD dizer para os outros GDs quais operações eles podem solicitar para o dispositivo de E/S, isso é realizado através do *backend driver*. Os GDs por outro lado devem saber usar *interface* criada pelo *backend driver* e como recuperar as respostas das requisições feitas, isso é feito através do *frontend driver*. A figura 6 apresenta um esquema de como funciona a abordagem *split-driver*.

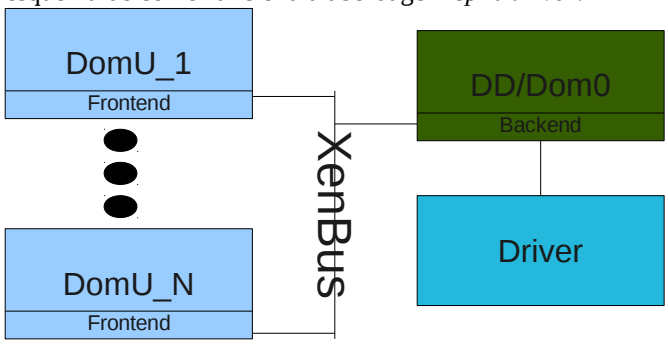


Figura 6.

É necessário também um mecanismo que realize a comunicação entre o *backend* e o *frontend driver*, isso é feito através do *XenBus*, que por sua vez usa o *XenStore*. O *XenStore* é um sistema de armazenamento que é compartilhado entre os GDs e é mantido pelo *Dom0*. Para se comunicar, os GDs devem ler/escrever no *XenStore*. O *XenStore* é muito usado para o controle de dispositivos. Segue abaixo uma descrição mais detalhada dos componentes fundamentais do *split-driver model*.

Driver: É o driver já existente no sistema operacional, utilizado para controlar os dispositivos de E/S. Esse *driver* vai ser utilizado para realizar o acesso ao dispositivo de E/S, quem controla ele é o DD ou o *Dom0* (depende das opções de configuração).

Backend driver: É o componente que garante a multiplexação do *driver*, permitindo que o dispositivo de E/S possa ser utilizado por mais de um GD. É o *backend driver* que recebe as requisições dos outros GDs e as envia para o *driver*. Essas requisições são feitas através de um *frontend driver*.

XenBus: É um protocolo que permite o *backend* e *frontend driver* se comunicar e foi construído em cima do *XenStore*. O *XenBus* utiliza um canal de eventos compartilhado e uma estrutura de dados em anel que implementa o modelo produtor/consumidor, ambos implementados pela *XenStore*. O sistema de eventos compartilhados serve para avisar o *Dom0*/DD de que existe uma requisição sendo feita para o dispositivo de E/S em específico e também para avisar o *DomU* de que a operação solicitada já foi realizada. O anel produtor/consumidor é onde o *Dom0*/DD buscam as operações de E/S que serão realizadas e os GDs solicitam a operação a ser feita. É importante ressaltar que a função do *XenStore* é apenas garantir a comunicação entre os GDs e que o *XenBus* apenas utiliza esse esquema de comunicação para permitir que os GDs possam solicitar uma operação de E/S. A figura 7 mostra um esquema de como os GDs realizam o processo de solicitar a operação de E/S e receber a resposta.

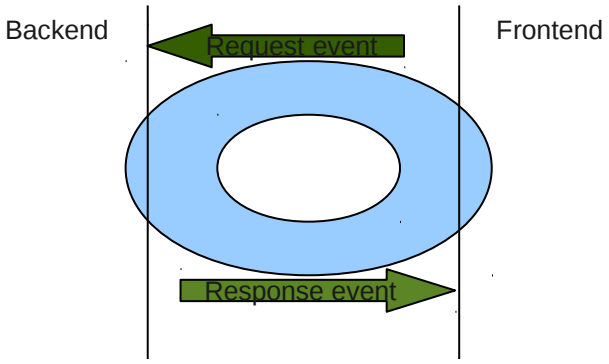


Figura 7.

Frontend driver: É o componente utilizado pelo GD para solicitar algum tipo de operação do dispositivo de E/S que se deseja. O *frontend driver* é o responsável por inicializar a estrutura de dados em anel e por estabelecer um canal de eventos para realizar solicitações para o *backend driver* do *Dom0*/DD.

Como o modelo *split-driver* não exige que os GDs que desejam usar o dispositivo de E/S tenham aquele *driver* em específico, esse é um modelo que garante que o GD possa se comunicar com vários dispositivos de uma mesma classe. Por exemplo, se o GD deseja acessar o HD para escrever em um bloco, ele precisa apenas dizer que deseja escrever no bloco, é responsabilidade do *Dom0*/DD que possui aquele *driver* realizar aquela operação e não do GD. Caso o GD migre de máquina, ele não precisa se preocupar com que dispositivos estão na máquina em que se encontra, pois ele sabe que vai ter alguém que realizará as operações de E/S por ele, isso torna o modelo *split-driver* muito flexível e útil em *datacenters*.

IX. Armazenamento de *Guest Domains*

Tipicamente os *datacenters* devem ter diversos VGs em execução e portanto ficam a maior parte do seu tempo na memória volátil. Eventualmente é necessário que esse VG armazene dados de forma persistente. O *Xen* permite que os GDs tenham acesso a qualquer dispositivo de armazenamento, desde que haja um *driver* que realize a comunicação entre o GD e o dispositivo. O *Xen* mascara o acesso ao dispositivo criando um *virtual block device* que é dado para o GD e esse por sua vez acha que está acessando um dispositivo de armazenamento real. A maneira mais comum de se utilizar dispositivos de armazenamento local é através do *Logical Volume Manager* (LVM). O conceito do LVM não é específico do *Xen*, ele foi criado pela IBM e tem sido utilizado em sistemas operacionais (principalmente Linux). O LVM gerencia Volumes Físicos (VFs), o que para o LVM podem ser blocos de discos, RAIDs, partições e até mesmo arquivos. Os VFs são quebrados em pedaços menores chamados de Extensões Físicas (EF), que definem o tamanho do bloco a ser usado em um VF. Os VFs podem ser agrupados para formar um Volume de Grupo (VdG), que por sua vez são usados para criar Volumes Lógicos (VLs) e são os VLs quem são efetivamente utilizados para o armazenamento. A figura 8 mostra um esquema de como um VdG é composto.

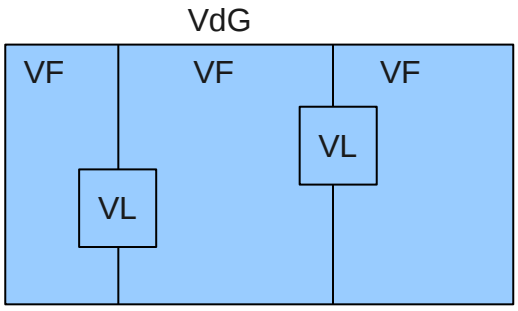


Figura 8.

O LVM é uma camada que abstrai os dispositivos de armazenamento, o que garante uma maior flexibilidade para realizar tarefas corriqueiras não apenas de *datacenters* como também de usuários comuns, como por exemplo

redimensionar o espaço que um GD ou sistema operacional pode ocupar. Outra vantagem do LVM é permitir o uso de *snapshots* com o modelo *copy-on-write* (CoW). A *snapshot* é o ato de salvar no disco o estado de um GD e isso pode ser feito mesmo quando ele está em execução, o que facilita a migração de máquinas virtuais (explicado na seção XI) e também a recuperação de falhas. O CoW é bom pois ele reduz o espaço de armazenamento necessário, criando cópias novas da *snapshot* apenas quando um GD realiza uma operação de escrita.

Além de dispositivos locais, o *Xen* também permite o armazenamento de GDs em sistemas de arquivos distribuídos. Um exemplo é o NFS, que pode suportar vários clientes escrevendo ao mesmo tempo, algo que não ocorre com dispositivos de armazenamento local pois se vários clientes desejam escrever ao mesmo tempo, muito possivelmente ocorrerá corrupção do sistema de arquivos, outro benefício é quando é necessário realizar alguma modificação em um arquivo de configuração e essa modificação deve ser visualizada pelos outros GDs.

X. Redes no *Xen*

Os GDs, assim como computadores comuns também necessitam de endereços IP, mas a máquina física pode ter um número de GDs que acessam a internet maior do que o número de *interfaces* físicas de rede (IF), por isso, é necessário que cada GD possua uma *interface* virtual de rede (IV), assim, o *Xen* multiplexa a IF entre os diversos GDs que desejam usa-la, analogamente, o *Xen* também demultiplexa as mensagens que foram recebidas na IF e entrega a mensagem para a IV do GD correspondente. Para que os GDs possam acessar a rede, é necessário que eles implementem o modelo *split-driver*, onde o *backend driver* está no *Dom0*/DD que controla a IF e esse *backend driver* é quem cria a IV. O *frontend driver* está nos GDs e é ele quem cria uma *interface* de rede virtual. Também são utilizadas duas estruturas em anel que implementam o modelo produtor/consumidor, uma serve para colocar o pacote que o GD deseja enviar e a outra serve para armazenar o pacote enviado para o GD. Os tipos de conexão que o *Xen* suporta são descritos abaixo:

Bridge: Em uma rede *bridged* os GDs são conectados a uma rede virtual (só é permitida uma rede virtual para cada IF), e a rede virtual é conectada a uma rede física. Uma rede *bridged* não encaminha pacotes IP, somente pacotes que contenham MAC *addresses* de origem e destino, que é gerado pela IV ou especificado pelo administrador. Esse tipo de rede também é utilizada para criar redes sem conexão com a internet. Para fazer isso é necessário estabelecer uma rede virtual que não seja conectada a uma rede física. Para um GD obter um IP nesse tipo de rede vai depender do administrador.

Roteador: Em uma rede roteada, o *Dom0*/DD é responsável por encaminhar os pacotes para a rede mais próxima da máquina de destino. Diferente de uma rede *bridged*, a comunicação ocorre através de um endereço IP. Neste tipo de rede, o IP do GD pode ser decidido pelo administrador ou por um servidor DHCP externo (que fica for a da máquina física) ou interno (que fica dentro da máquina fisica).

NAT: Quando o *Xen* usa o NAT o GD se esconde atrás do *Dom0*/DD como se ele estivesse por trás de uma NAT física. Por trás do *Dom0*/DD, os GDs estão em uma rede privada e cada um possui um endereço IP único, os nós que estão na internet não conseguem contactar os GDs usando o IP da rede privada. Quando um pacote IP chega no *Dom0*/DD é necessário que ele seja traduzido, mudando o IP e a porta. O NAT permite que diversos GDs compartilhem um único IP público, deixando mais faixas de IP disponíveis para serem utilizadas no *datacenter*.

XI. Migração de máquinas virtuais

Podem ocorrer casos em que uma máquina real será incapaz de carregar o GD desejado devido a motivos variados como por exemplo manutenção ou sobrecarga da máquina. Mas em um *datacenter* isso não deve impedir que o cliente tenha acesso ao serviço que lhe é disponibilizado, nessas situações, é necessários transferir para outra máquina o GD em específico, a essa transferência da-se o nome de migração. O *Xen* suporta três tipos de migração, descritas abaixo:

Cold Migration: Este é o tipo mais básico de migração de GDs. É necessário que um GD seja desligado e que a sua imagem gravada no disco seja copiada do dispositivo de armazenamento para a máquina em que a imagem será carregada.

Warm Migration: Este tipo de migração não necessita que o GD seja desligado mas sim pausado e então ele é transferido para a máquina destino. Terminando a transferência, o GD pausado volta a executar. Esse tipo de migração interrompe as possíveis modificações na memória que poderiam ocorrer caso o GD continuasse em execução. O *Xen* não suporta o espelhamento do sistema de arquivos do GD, para que ambas as réplicas tenham acesso ao mesmo sistema de arquivos, é necessário compartilha-lo pela rede.

Live Migration: Este tipo de migração permite que uma GD seja transferida para uma outra máquina física de forma transparente e sem interromper a execução da GD. Como se pode imaginar, esse tipo de migração é mais complicada pois o estado da máquina que será transferida continua sendo alterado, para resolver isso é necessário transferir a GD e checar o que mudou na GD durante esse tempo e então copiar o que foi mudado. O acesso ao sistema de arquivos é feito da mesma forma que no *Warm Migration*.

De todos os métodos descritos, o *live migration* é o melhor pois realiza a migração de dados de uma maneira muito mais transparente do que os outros dois tipos, entretanto, é também a forma mais complicada.

XII. Gerência de Recursos para *Guest Domains*

Uma máquina física usualmente executará mais de um GD, isso significa que eles deverão compartilhar entre si

recursos da máquina e mais ainda, dependendo do recurso, ele não deve ser monopolizado por uma única máquina. Neste seção serão abordados alguns problemas que surgem na gerência de recursos para os GDs e qual a abordagem adotada pelo *Xen*.

Um dos recursos que deve ser compartilhado por todos os GDs é a memória, o *Xen* utiliza a abstração de memória virtual (assim como a maior parte dos sistemas operacionais modernos). Quando o *Xen* é inicializado, ele reserva uma porção da memória virtual no início do espaço de endereço e todo o resto é disponibilizado para os GDs. O uso da abstração de memória virtual garante segurança, tornando impossível os GDs referenciar a memória que não foi reservada para ele e também garante o uso de mais memória do que é disponível na máquina. Apesar de benéfico, isso pode levar a uma situação de *thrashing*, o que reduz a performance do sistema. Para resolver isso, o *Xen* impõe um limite na quantidade de memória que cada GD pode usar, essa solução não apenas resolve a situação de *thrashing* como também evita que um GD use mais memória do que outros. Além da memória, outro recurso importante é a CPU. O *Xen* virtualiza as CPUs, assim elas podem ser compartilhadas entre vários GDs. Cada GD pode ter até 32 ou 64 CPUs virtuais (CPUV) em um sistema de 32 bits ou 64 bits, respectivamente. As CPUVs possuem o mesmo conjunto de instruções das CPUs reais, portanto, os GDs não percebem diferença entre elas. Da mesma forma que um sistema operacional necessita de um escalonador de processos, o *Xen* necessita de um escalonador de GDs. Assim como o escalonador do sistema operacional, o escalonador do *Xen* necessita definir um critério para decidir qual o próximo GD a usar a CPU e por quanto tempo. O escalonador do *Xen* adota o critério de *weight and cap* onde o *weight* (peso) define quanto tempo um GD tem em relação aos outros, por exemplo, um GD cujo weight é 1024 terá oito vezes mais tempo de CPU do que um GD com peso de 128. O *cap* (boné) é um limite definido em porcentagem sobre a quantidade de CPU que um GD pode usar, um *cap* de 50%, 100% e 200% representam respectivamente metade de uma CPU, uma única CPU e duas CPUs. O algoritmo do escalonador coloca as CPUVs em uma fila por CPU em uma prioridade associada. A prioridade pode ser “over” (acima) ou “under” (abaixo), representando se a CPUV excedeu ou não o seu tempo de uso da CPU, de tempos em tempos as prioridades de todas as CPUVs é recalculada.

XIII. Conclusão

A virtualização é um conceito antigo mas que ganhou popularidade apenas recentemente, principalmente devido a sua ampla adoção por parte de empresas que fornecem serviços de *cloud computing*, pois as mesmas desejam otimizar ao máximo o uso energético de seus *datacenters*. Através da virtualização, é possível ter uma única máquina física capaz de rodar uma ampla variedade de sistemas ao mesmo tempo, o que elimina gastos da empresa com vários computadores e também consumindo menos energia. Diversos modelos de virtualização de sistemas operacionais foram propostos, mas a paravirtualização foi um dos que chamou mais atenção devido a sua abordagem que garante um bom desempenho das máquinas virtuais, similar ao de uma máquina real. Esse modelo foi implementado pelo *Xen*, que é um virtualizador *opensource* e gratuito, capaz de virtualizar diversas arquiteturas e possui suporte para vários sistemas operacionais, o que o torna uma alternativa atraente para empresas que não desejam ter gastos com virtualizadores pagos e desejam fornecer um serviço de virtualização de alta qualidade para seus clientes. Além disso, o *Xen* permite facilidades administrativas para o administrador, como por exemplo a migração de máquinas virtuais, algo essencial no ambiente de *cloud computing*.

Bibliografia

Running Xen: A Hands-On Guide to the Art of Virtualization; Jeanna N. Matthews, Eli M. Dow, Todd Deshane, Wenjin Hu, Jeremy Bongio, Patrick F. Wilbur, Brendan Johnson; Editora: Prentice Hall.

The Definitive Guide to the Xen Hypervisor; David Chisnall; Editora: Prentice Hall.

The Book of Xen: A practical Guide for the System Administrator; Chris Takemura, Luke S. Crawford. Editora: No Starch Press.

Professional Xen Virtualization; William von Hagen; Editora: Wrox.

Formal Requirements for Virtualizable Third Generation Architectures; Gerald J. Popek, Robert P. Goldberg.

Xen and the Art of Virtualization; Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield.