UNIVERSIDADE DE CAXIAS DO SUL
CENTRO DE CIÊNCIAS EXATAS E DA TECNOLOGIA
CIÊNCIA DA COMPUTAÇÃO

BRUNO EMER

Implementação de alta disponibilidade em uma empresa prestadora de serviços para Internet

André Luis Martinotto
Orientador

Caxias do Sul
Maio de 2016

Implementação de alta disponibilidade em uma empresa prestadora de serviços para Internet

por

Bruno Emer

Projeto de Diplomação submetido ao curso de Bacharelado em Ciência da Computação do Centro de Ciências Exatas e da Tecnologia da Universidade de Caxias do Sul, como requisito obrigatório para graduação.

Projeto de Diplomação

Orientador: André Luis Martinotto
Banca examinadora:

Maria de Fatima Webber do Prado Lima
CCTI/UCS
Ricardo Vargas Dorneles
CCTI/UCS

Projeto de Diplomação apresentado em
x de x de 2016

Daniel Luís Notari
Coordenador

SUMÁRIO

LISTA DE SIGLAS

AMD-V	*AMD virtualization*
ECC	*Error correction code*
ISA	*Instruction set architecture*
IVT	*Intel virtualization technology*
JVM	*Java virtual machine*
KVM	*Kernel-based virtual machine*
MTBF	*Mean time between failures*
MTTR	*Mean time to repair*
PC	*Personal computer*
RAID	*Redundant array of independent disks*
SLA	*Service level agreement*
SPOF	*Single point of failure*
TI	Tecnologia da informação
VM	*Virtual machine*
VMM	*Virtual machine monitor*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Palavras-chave: .

1 INTRODUÇÃO

O crescente avanço tecnológico e o desenvolvimento da internet, provocou um aumento no número de aplicações ou serviços que dependem da infraestrutura de Tecnologia da informação (TI). Além disso, percebe-se um aumento significativo no número de operações e negócios *on-line* que são realizados, tanto por organizações públicas ou privadas, quanto por grande parte da população.

Desta forma, a sociedade está cada vez mais dependente da tecnologia, de computadores e de sistemas. De fato, pode-se observar sistemas computacionais desde em uma farmácia, até em uma grande indústria. Sendo assim, a estabilidade e a disponibilidade destes sistemas apresentar um grande impacto em nosso dia-a-dia, pois um grande número de atividades cotidianas dependem deles.

Uma interrupção imprevista em um ambiente computacional poderá causar um prejuízo financeiro para a empresa que fornece o serviço, além de interferir na vida das pessoas que dependem de forma direta ou indireta deste serviço. Essa interrupção terá maior relevância para corporações cujo o serviço ou produto final é fornecido através da internet, como por exemplo, o comércio eletrônico, *web sites*, sistemas corporativos, entre outros. Em um ambiente extremo, pode-se imaginar o caos e o possível risco de perda de vidas que ocorreria em caso de uma falha em um sistema de controle aéreo (COSTA, 2009).

Para essas empresas um plano de contingência é fundamental para garantir uma boa qualidade de serviço, além de otimizar o desempenho das atividades, e também para fazer uma prevenção de falhas e uma recuperação rápida caso essas ocorram (COSTA, 2009). De fato, hoje em dia a confiança em um serviço é um grande diferencial para a empresa fornecedora deste serviço, sendo que a alta disponibilidade é fundamental para atingir este objetivo.

A alta disponibilidade consiste em manter um sistema disponível por meio da tolerância a falhas, isto é, utilizando mecanismos que fazem a detecção, mascaramento e a recuperação de falhas, sendo que esses mecanismos podem ser implementados a nível de *software* ou de *hardware* (REIS, 2009). Para que um sistema seja altamente disponível ele deve ser tolerante a falhas, sendo que a tolerância a falhas é implementada, frequentemente, utilizando redundância. No caso de uma falha em um dos componentes evita-se a interrupção do sistema, uma vez que o sistema poderá continuar funcionando, utilizando o outro componente (BATISTA, 2007).

Neste trabalho será realizado um estudo sobre a implementação de um sistema de alta disponibilidade em uma empresa de hospedagens. Essa empresa oferece serviços pela internet, como por exemplo hospedagens de sites, *e-mail*, sistemas de gestão, *e-mail marketing*, entre outros. A empresa possui aproximadamente 55 servidores físicos e virtuais, e aproximadamente 9000 clientes, sendo que em períodos de pico

atende em torno de 1000 requisições por segundo.

Atualmente, a empresa possui redundância de conexões de acesso a internet, refrigeração e energia, com *nobreaks* e geradores. Porém, essa empresa não possui nenhuma redundância nos serviços que estão sendo executados nos servidores. Desta forma, caso ocorra uma falha de *software* ou de *hardware*, os serviços ficarão indisponíveis. Neste trabalho será realizada uma análise dos serviços oferecidos pela empresa, sendo que mecanismos de alta disponibilidade serão desenvolvidos para os serviços mais críticos. Para a redução dos custos serão utilizadas ferramentas gratuitas e de código aberto.

1.1 Objetivos

Atualmente a empresa estudada não possui nenhuma solução de alta disponibilidade para seus serviços críticos. Desta forma, neste trabalho será desenvolvida uma solução de alta disponibilidade para estes serviços, sendo que essa solução será baseada no uso de ferramentas de código aberto e de baixo custo. Para que o objetivo geral seja atendido os seguintes objetivos específicos deverão ser realizados:

- Identificar os serviços críticos a serem integrados ao ambiente de alta disponibilidade;
- Definir as ferramentas a serem utilizadas para implementar tolerância a falhas;
- Realizar testes para a validação do sistema de alta disponibilidade que foi desenvolvido.

1.2 Estrutura do trabalho

O trabalho foi estruturado em ??? capítulos, que estão detalhados a seguir:

- O Capítulo 2 apresenta o conceito de alta disponibilidade e outros conceitos relacionados;
- No Capítulo 3 é descrito um breve histórico da virtualização, bem como o conceito de máquinas virtuais e também as suas classificações.

2 ALTA DISPONIBILIDADE

Alta disponibilidade é um termo muito conhecido, sendo cada vez mais empregada em ambientes computacionais. O objetivo de promover alta disponibilidade resume-se em garantir que um serviço esteja sempre disponível quando o cliente solicitar ou acessar (COSTA, 2009). A alta disponibilidade frequentemente é implementada com uma redundância de *hardware* ou de *software* para que o serviço fique mais tempo disponível, sendo que quanto maior for a disponibilidade desejada maior deverá ser a redundância no ambiente, assim reduzindo os pontos únicos de falha, que em inglês são chamados de *Single point of failure* (SPOF). A alta disponibilidade está diretamente relacionada aos conceitos de:

- Dependabilidade: indica a qualidade do serviço fornecido e a confiança depositada neste serviço. A dependabilidade envolve atributos como segurança de funcionamento, segurança de acesso, manutenabilidade, testabilidade e comprometimento do desempenho (WEBER, 2002);
- Confiabilidade: é o atributo mais importante, pois transmite a ideia de continuidade de serviço (PANKAJ, 1994). A confiabilidade refere-se a probabilidade de um serviço funcionar corretamente durante um dado intervalo de tempo;
- Disponibilidade: é a probabilidade de um serviço estar operacional no instante em que for solicitado (COSTA, 2009);
- Tolerância a falhas: procura garantir a disponibilidade de um serviço utilizando mecanismos capazes de detectar, mascarar e recuperar falhas, e seu objetivo é alcançar a dependabilidade, assim indicando uma boa qualidade de serviço (COSTA, 2009). A tolerância a falhas é um dos principais conceitos da alta disponibilidade, sendo melhor apresentada na Seção 2.1.

2.1 Tolerância a falhas

Sabe-se que o *hardware* tende a falhar, principalmente devido a fatores físicos, por isso utiliza-se métodos para a prevenção de falhas. A abordagem de prevenção de falhas é realizada na etapa de projeto, ou seja, consiste na criação de mecanismos que impeçam que que as falhas ocorram. Além disso, a prevenção de falhas melhora a disponibilidade e a confiabilidade de um serviço, uma vez que essa tem como objetivo prever e eliminar o maior número de falhas possíveis antes de colocar o sistema em uso.

A prevenção de falhas não resolverá todas as possíveis falhas. Sendo assim, a tolerância a falhas procura fornecer a disponibilidade de um serviço mesmo com

a presença de falhas. De fato, enquanto a prevenção de falhas tem foco nas fases de projeto, teste e validação, a tolerância a falhas tem como foco na utilização de componentes replicados para mascarar as falhas (PANKAJ, 1994).

O objetivo da tolerância a falhas é aumentar a disponibilidade de um sistema, ou seja, aumentar o intervalo de tempo em que os serviços fornecidos estão disponíveis aos usuários. Um sistema é dito tolerante a falhas se ele for capaz de mascarar a presença de falhas ou recuperar-se de uma falha sem afetar o funcionamento do sistema. A tolerância a falhas é implementada utilizando redundância (Seção 2.2). Um exemplo muito utilizado para tornar um sistema tolerante a falhas é a virtualização. Nestes ambientes normalmente existem dois servidores físicos onde máquinas virtuais são executadas, sendo que no caso de um dos servidores falhar, o *software* de monitoramento fará a transferência das máquinas virtuais para o outro servidor, de forma transparente aos usuários, evitando assim a indisponibilidade do serviço. Os principais conceitos de virtualização, são apresentados no Capítulo 3.

A tolerância a falhas pode ser dividida em dois tipos. O primeiro tipo, o mascaramento, não se manifesta na forma de erro sendo assim não necessita que o sistema trate este erro, pois as falhas são tratadas na origem. O mascaramento é utilizado principalmente em sistemas de tempo real crítico. Um exemplo são os códigos de correção de erros, em inglês *Error correction code* (ECC), que são utilizados em memórias para detecção e correção de erros.

O segundo tipo consiste em detectar, localizar a falha e reconfigurar o *software* ou *hardware* de forma a corrigir a falha. Esse tipo de tolerância a falha é dividido nas seguintes etapas (WEBER, 2002).

- Detecção: realiza o monitoramento e aguarda uma falha se manifestar na forma de erro, para então passar para a próxima fase. Um exemplo de detecção de erro é o cão de guarda (*watchdog timer*), que recebe um sinal do programa ou serviço que esta sendo monitorado e caso este sinal não seja recebido, o *watchdog* irá se manifestar na forma de erro. Um outro exemplo é o esquema de duplicação e comparação, onde são realizadas operações em componentes replicados com mesmos dados de entrada, e então os dados de saída são comparados. No caso de diferenças nos dados de saída um erro é gerado.
- Confinamento: responsável pela restrição de um erro para que dados inválidos não se propaguem para todo o sistema, pois entre a falha e a detecção do erro há um intervalo de tempo. Neste intervalo pode ocorrer a propagação do erro para outros componentes do sistema, sendo assim antes de executar medidas corretivas é necessário definir os limites da propagação. Na fase de projeto essas restrições devem ser previstas e tratadas. Um exemplo de confinamento é o isolamento dos processos de um sistema operacional, sendo que esse sistema faz o gerenciamento dos processos para isolar e impedir que as falhas de um processo gerem problemas em outros processos, com isso as falhas de um programa estão restritas ao processo.
- Recuperação: após a detecção de um erro ocorre a recuperação, onde o estado de erro é alterado para estado livre de erros. A recuperação pode ser feita de duas formas, que são:

  - *forward error recovery* (recuperação por avanço): ocorre uma condução para um estado que ainda não ocorreu. É a forma de recuperação mais

eficiente, porém mais complexo de ser implementado.
  - *backward error recovery* (recuperação por retorno): ocorre um retorno para um estado anterior livre de erros. Para retornar ao estado anterior pode ser utilizados pontos de recuperação (*checkpoints*). Assim quando ocorrer um erro, um *rollback* é executado, ou seja, o sistema retornará a um estado anterior a falha.

- Tratamento: procura prevenir que futuros erros aconteçam. Nesta fase ocorre a localização da falha para descobrir o componente que originou a falha. A substituição do componente danificado pode ser feita de forma manual ou automática. O reparo manual é feito por um operador que é responsável pelo reparo ou substituição de um componente. Como exemplo pode-se citar a troca de um disco rígido de um servidor. Já o reparo automático é utilizado quando existe um componente em espera para substituição, como por exemplo, um disco configurado como *hot spare*, ou seja, um componente de *backup* que assumirá o lugar do outro imediatamente após o componente principal falhar. Em *storages* ou servidores, o *hot spare* pode ser configurado através de um *Redundant array of independent disks* (RAID) (ROUSE, 2013).

2.2 Redundância

A redundância pode ser implementada através da replicação de componentes, e tem como objetivo reduzir o número de SPOF e garantir o mascaramento de falhas. Na prática, se um componente falhar ele deve ser reparado ou substituído por um novo, sem que haja uma interrupção no serviço. Além disso redundância pode ser implementada através do envio de sinais ou *bits* de controle junto aos dados, servindo assim para detecção e correção de erros (WEBER, 2002). Segundo (NøRVåG, 2000) existem quatro tipos diferentes de redundância que são:

- *Hardware*: utiliza-se a replicação de componentes, sendo que caso um falhe outro possa assumir seu lugar. Para fazer a detecção de erros a saída de cada componente é constantemente monitorada e comparada à saída do outro componente. Um exemplo prático de redundância de *hardware* são os servidores com fontes redundantes. Nestes são utilizadas duas fontes ligadas em paralelo, sendo que caso uma falhe a outra suprirá a necessidade de todo o servidor;
- Informação: ocorre quando uma informação extra é enviada ou armazenada para possibilitar a detecção e a correção de erros. Um exemplo são os *checksums* (soma de verificação). Esses são calculados antes da transmissão ou armazenamento dos dados e recalculados ao recebê-los ou recuperá-los, assim sendo possível verificar a integridade dos dados. Outro exemplo bastante comum são os *bits* de paridade que são utilizados para detectar falhas que afetam apenas um *bit* (WEBER, 2002);
- *Software*: pode-se definir redundância de *software* como a configuração de um serviço ou *software* em dois ou mais locais diferentes. Pode-se citar como exemplo, um sistema gerenciador de banco de dados *MySQL*, que pode ser configurado com um modelo de replicação do tipo *master-slave*, onde um servidor principal (*master*) grava as operações em um arquivo, para então os servidores *slaves* recuperarem e executarem essas operações, com isso mantendo os dados

Tabela 2.1: Níveis de alta disponibilidade e exemplos de sistemas

Nível	Uptime	Downtime por ano	Exemplos
1	90%	36.5 dias	computadores pessoais
2	98%	7.3 dias	
3	99%	3.65 dias	sistemas de acesso
4	99.8%	17 horas e 30 minutos	
5	99.9%	8 horas e 45 minutos	provedores de acesso
6	99.99%	52.5 minutos	CPD, sistemas de negócios
7	99.999%	5.25 minutos	sistemas de telefonia ou bancários
8	99.9999%	31.5 minutos	sistemas de defesa militar

sincronizados. Neste caso, tanto o servidor *master* quanto os *slaves* executam o serviço *MySQL*, caracterizando uma redundância (SILVA VIANA, 2015). A redundância de *software* também pode ser implementada com o objetivo de tolerar falhas e *bugs* em um *software* crítico. Existem algumas técnicas que podem ser utilizadas para essa implementação, como por exemplo, a programação de n-versões, que consiste em criar n versões para um mesmo *software*, desta forma, possibilita-se o aumento da disponibilidade, uma vez que elas provavelmente não apresentarão os mesmos erros. Por outro lado a programação de n-versões possui um custo muito elevado, não sendo muito utilizada.

- Tempo: este é feito através da execução de um conjunto de instruções repetidas vezes em um mesmo componente, assim detectando uma falha caso ocorra. Essa técnica necessita tempo adicional, e é utilizada em sistemas onde o tempo não é crítico. Por exemplo, um *software* de monitoramento de serviços em servidores que faz um teste em cada serviço. No caso de ocorrência de alguma falha, uma ação corretiva será executada para reestabelecer este serviço. Essa técnica, diferentemente da redundância de *hardware*, não requer um *hardware* extra para sua implementação (COSTA, 2009).

2.3 Cálculo da alta disponibilidade

Um aspecto importante sobre alta disponibilidade é como medi-la. Para isso são utilizados os valores de *uptime* e *downtime*, que são respectivamente, o tempo em que os serviços estão em execução e o tempo em que não estão executando. A alta disponibilidade pode ser expressa pela quantidade de "noves", isto é, se um serviço possui quatro noves de disponibilidade, este possui uma disponibilidade de 99,99% (PEREIRA FILHO, 2004).

A Tabela 2.1 apresenta alguns níveis de disponibilidade, a suas porcentagens de *Uptime*, os *Downtime* por ano. Já na última coluna tem-se alguns exemplos de serviços relacionados ao nível de disponibilidade. Pode-se observar que para alguns serviços, como por exemplo, sistemas bancários ou sistemas militares é necessário um alto nível de disponibilidade (PEREIRA FILHO, 2004).

A porcentagem de disponibilidade (d) pode ser calculada através da equação

$$d = \frac{MTBF}{(MTBF + MTTR)} \tag{2.1}$$

onde *Mean time between failures* (MTBF) é o tempo médio entre falhas, ou seja, corresponde ao tempo médio entre as paradas de um serviço. Já o *Mean time to repair* (MTTR) é o tempo médio de recuperação, isto é, o tempo entre a queda e a recuperação de um serviço (GONçALVES, 2009).

A alta disponibilidade é um dos principais fatores que fornece confiança aos clientes ou usuários, sendo extremante importante em empresas que fornecem serviços *on-line*. Por isso, as empresas desenvolveram o *Service level agreement* (SLA), que é um acordo de nível de serviço, que garante que o serviço fornecido atenda as expectativas dos clientes. Um SLA é um documento contendo uma descrição e uma definição das características mais importantes do serviço que será fornecido. Esse acordo também deverá conter a porcentagem de disponibilidade exigida pelo negócio, sendo que esta deve ser minuciosamente definida. Por exemplo, um SLA deverá conter descrição do serviço, requerimentos, horário de funcionamento, *uptime* do serviço, *downtime* máximo do serviço, entre outros (SMITH, 2010).

3 VIRTUALIZAÇÃO

O conceito virtualização surgiu na década de 60, onde muitas vezes era necessário que um usuário utilizasse um ambiente individual, com suas próprias aplicações e totalmente isolado dos demais usuários. Esse foi um dos principais motivos para a criação de máquinas virtuais, mais conhecida como *Virtual machine* (VM), que apresentaram uma forte expansão com o sistema operacional *370* que foi desenvolvido pela *IBM* e foi um dos principais sistemas comerciais com suporte a virtualização da época. Esse sistema operacional executava sobre *mainframes*, que na época eram grandes servidores capazes de processar um grande volume de informações (LAUREANO; MAZIERO, 2008).

Na década de 80, houve uma redução da utilização da virtualização devido a popularização do *Personal computer* (PC). Na época era mais vantajoso disponibilizar um PC para cada usuário, do que investir em *mainframes*. Devido ao crescente avanço e o melhor desempenho do PC e ao surgimento da linguagem *Java*, no início da década de 90, a tecnologia de virtualização retornou com o conceito de virtualização de aplicação.

A virtualização foi definida nos anos 60 e 70 como uma camada entre o *hardware* e o sistema operacional que possibilitava a divisão e proteção dos recursos físicos. Porém, atualmente ela abrange outros conceitos, como por exemplo a *Java virtual machine* (JVM), que não virtualiza necessariamente um *hardware*. Ela proporciona que uma aplicação convidada execute em diferentes tipos de sistemas operacionais.

Atualmente define-se virtualização como uma camada de *software* que utiliza os serviços fornecidos de uma determinada interface de sistema para criar outra interface de mesmo nível. Essa camada irá permitir a comunicação entre interfaces distintas, de forma que uma aplicação desenvolvida para uma plataforma X possa também executar em uma plataforma Y (LAUREANO; MAZIERO, 2008).

Nesse Capítulo será apresentado as diferentes classificações de máquinas virtuais, porém antes disso deve-se conhecer os diferentes tipos de interfaces existentes em sistemas de computação:

- Conjunto de instruções ou *Instruction set architecture* (ISA): é a interface básica, que fica entre o *software* e o *hardware*, e é composta por instruções de código de máquina. Esta interface é dividida em dois grupos:
  - Instruções de usuário ou *User ISA*: são instruções de *hardware* disponíveis à aplicações de usuários. Essas executam em modo não-privilegiado. Um exemplo de instrução de usuário é uma instrução do processador executadas diretamente sobre a memória alocada para o programa ????;

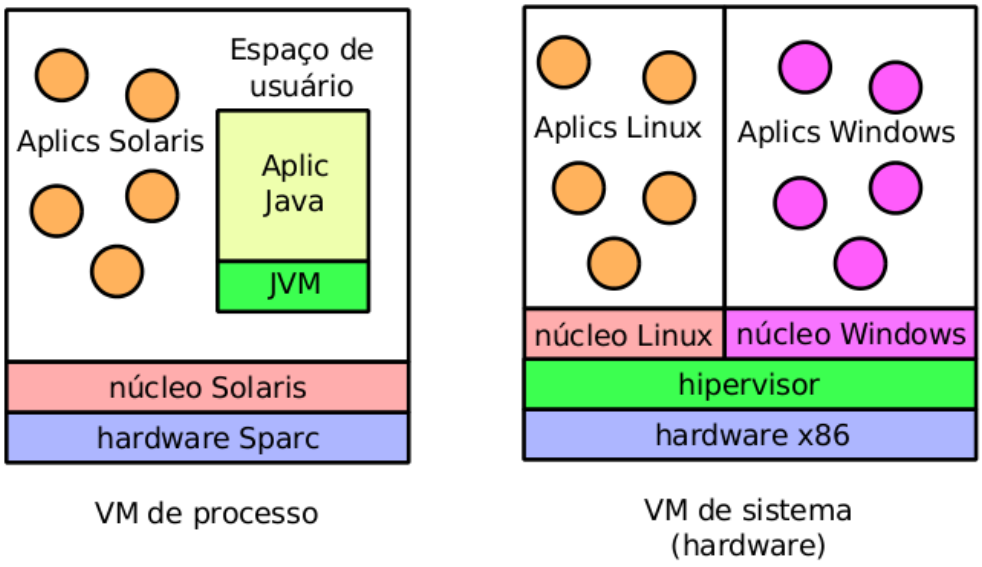

Figura 3.1: Máquinas virtuais de aplicação e de sistema.
Fonte: LAUREANO; MAZIERO (2008)

  - Instruções de sistema ou *System ISA*: são instruções exclusivamente acessíveis ao núcleo do sistema operacional. Essas são executadas em modo privilegiado. Pode-se citar as instruções de entrada e saída (E/S) como exemplo de instruções de sistema;

- Chamadas de sistema ou *syscalls*: são operações oferecidas pelo núcleo do sistema operacional para as aplicações dos usuários. Essas operações permitem o acesso controlado aos dispositivos, memória e processador. Um exemplo de chamada de sistema é uma escrita em disco.

Máquinas virtuais podem ser divididas em dois grupos principais, que são: as máquinas virtuais de aplicação, na (Seção 3.1), e máquinas virtuais de sistema, na (Seção 3.2). As máquinas virtuais de aplicação fazem a virtualização de uma aplicação e suportam apenas um processo ou aplicação. Um exemplo de máquina virtual de aplicação é a JVM. Já uma máquina virtual de sistema suporta sistemas operacionais convidados, com suas aplicações executando sobre ele (Figura 3.1). Uma máquina virtual executando com *Kernel-based virtual machine* (KVM) é um exemplo de máquina virtual de aplicação (LAUREANO; MAZIERO, 2008).

3.1 Máquinas virtuais de aplicação

As máquinas virtuais de aplicação, também chamadas de máquinas virtuais de processos são responsáveis por prover um ambiente onde um sistema operacional suporte uma aplicação convidada, sendo que esta aplicação possui um conjunto de instruções ou de chamadas do sistema diferentes da arquitetura do sistema hospedeiro. De fato quando temos chamadas do sistema operacional ou instruções de máquina que são diferentes das oferecidas pela máquina real, será necessário uma tradução dessas interfaces que será feita pela camada de virtualização. Abaixo são detalhados dois tipos de máquinas virtuais de aplicação:

- Máquinas virtuais de linguagem de alto nível: esse tipo de máquina virtual foi criado levando em consideração uma linguagem de programação e seu compilador. O código compilado gera um código binário que não pode ser executado em uma arquitetura real, mas pode ser executada em uma máquina virtual. Sendo assim para cada arquitetura ou sistema operacional deve existir uma máquina virtual que permita a execução da aplicação nesse ambiente. Como exemplo deste tipo de máquina virtual pode-se citar a: *Microsoft Common Language Infrastructure*, base do *.Net* e a máquina virtual Java (JVM) (CARISSIMI, 2008);
- Emulação no sistema operacional: nesse caos é feito um mapeamento entre as chamadas de sistema que são utilizadas pela aplicação e as chamadas do sistema operacional real. A virtualização de aplicação pode ser encontrado em ferramentas que emulam uma aplicação desenvolvida para uma plataforma em outra plataforma distinta, como por exemplo o *Wine*, que permite executar aplicações *Windows* em plataformas *Linux*.

3.2 Máquinas virtuais de sistema

As máquinas virtuais de sistema também chamadas de hipervisor ou *Virtual machine monitor* (VMM), são uma camada de *software* que possibilita que um ou mais sistemas operacionais convidados executem independentemente sobre um memsmo computador físico. O hipervisor prove uma interface ISA virtual que pode ou não ser igual a interface real, e virtualiza outros componentes de *hardware*, para que cada máquina virtual convidada possa ter seus próprios recursos isolados.

Nesse modelo, um ambiente de virtualização de sistema é composto basicamente por três componentes:

- Sistema real: também pode ser chamado de hospedeiro, que é o *hardware* onde o sistema de virtualização irá executar;
- Camada de virtualização: é conhecida como hipervisor ou também chamado de VMM e tem como função criar interfaces virtuais a partir de interfaces físicas, para a comunicação do sistema virtual com o sistema real;
- Sistema virtual: também conhecido como *guest*, ou sistema convidado, que executa sobre o sistema real. Geralmente existem vários sistemas virtuais executando simultaneamente sobre o sistema real.

Virtualização de sistema utiliza abstração em sua arquitetura, por exemplo, ela transforma um disco rígidos físico em dois discos rígidos virtuais menores, sendo que esses discos virtuais são arquivos armazenados no disco físico. Sabendo que arquivos são uma abstração em um disco rígidos físico, pode-se dizer que virtualização não é apenas uma camada de abstração do *hardware*, ela faz a reprodução do *hardware* (SMITH; NAIR, 2005).

Existem basicamente duas arquiteturas de hipervisor de sistema, apresentados na Figura 3.2 e detalhados abaixo (MAZIERO, 2013):

- Hipervisores nativos: esse hipervisor executa diretamente sobre o *hardware*, ou seja, sem um sistema operacional hospedeiro. O hipervisor nativo faz a multiplexação dos recursos do *hardware* como memória, disco rígido, interface de rede, entre outros, e diponibiliza esses recursos para as máquinas virtuais.

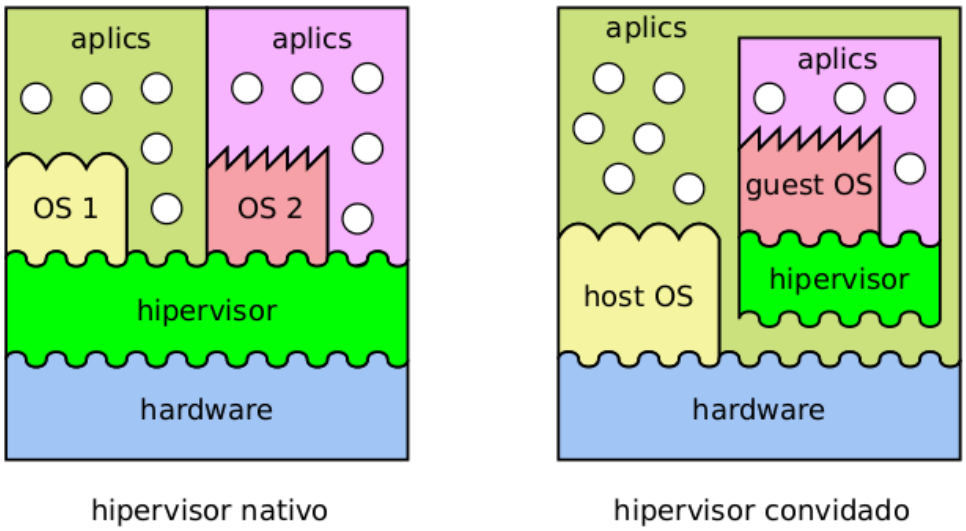

Figura 3.2: Arquiteturas de máquinas virtuais de sistema.
Fonte: MAZIERO (2013)

Alguns exemplos que utilizam esse hipervisor são *IBM 370*, o *Xen* e o *VMware ESX Server*;

- Hipervisores convidados: esse tipo de hipervisor executa sobre um sistema operacional hospedeiro, e utiliza os recursos desse sistema para gerar recursos virtuais para as máquinas virtuais. Normalmente esse tipo suporta apenas um sistema operacional convidado para cada hipervisor. Exemplos deste tipo de sistema são: o *VirtualBox* e o *QEmu*.

Sabendo essas definições das arquiteturas pode-se concluir que hipervisor convidados são mais flexíveis que os nativos, pois podem ser implementados em diversos sistemas operacionais e *hardwares*. Já o hipervisor nativo possui melhor desempenho pois acessa o *hardware* diretamente.

3.2.1 Estratégias de virtualização

As máquinas virtuais de sistema podem ser classificadas em diferentes tipos de estratégias. Atualmente as estratégias mais utilizadas são a virtualização total e a paravirtualização. Antes disso deve-se compreender uma técnica conhecida como tradução dinâmica. Nesta técnica o hipervisor adapta as instruções geradas pelo sistema convidado para a interface ISA do sistema real. Outra função da tradução dinâmica é analisar e reoganizar as instruções do sistema convidado, com o objetivo de melhorar o desempenho da execução do sistema convidado.

- Virtualização total: nesta estratégia todas as interfaces de acesso ao *hardware* são virtualizadas. Desta forma possibilita-se que sistemas operacionais convidados executem como se estivessem diretamente sobre o *hardware*. Na virtualização total o conjunto de instruções do processador é acessível somente pelo hipervisor, sendo que essa estratégia utiliza tradução dinâmica para traduzir as instruções do sistema convidado. A grande vantagem dessa virtualização é a possibilidade de um sistema convidado executar sem a necessidade de ser

modificado. Porém, existe uma redução significativa no desempenho devido ao hipervisor intermediar todas as chamada de sistemas e operações do sistema convidado. Um exemplo de virtualização total é o *QEmu*;

- Paravirtualização: esta utiliza arquitetura de hipervisor nativo, prove um melhor acoplamento entre os sistemas operacionais convidados e o hipervisor. Para isso o sistema convidado deve ser adaptado para o hipervisor no qual executará, ou seja, a interface de sistema (*system ISA*) será acessada diretamente pelo sistema convidado, resultando em um desempenho melhor. São exemplos de sistema que implementa a paravirtualização: o *Xen* e o *VMware ESX Server*.

A paravirtualização possui um desempenho melhor se comparada a virtualização total, pois acessa alguns recursos diretamente, sendo que o hipervisor é reponsável somente por impedir que o sistema convidado faça operações indevidas. Por exemplo, o gerenciamento da memória, na virtualização total o hipervisor reserva um espaço para cada sistema convidado, que por sua vez acessa a memória como se fosse a memória de uma máquina física iniciando seu endereçamento em zero. Sendo assim cada vez que o sistema convidado acessar a memória, o hipervisor precisará converter os endereços do sistema convidado para endereços reais. Na paravirtualização o hipervisor informa ao sistema convidado a área de memória que ele pode utilizar, otimizando assim as operações.

Por outro lado, a virtualização total possui maior portabilidade, ou seja, permite que sistemas operacionais convencionais executem como convidados, sem serem modificados. Pode-se, por exemplo, transferir um sistema operacional instalado diretamente em uma máquina física para um ambiente virtual, sem a necessidade de reinstalar e reconfigurar o sistema operacional.

A virtualização total obteve uma grande ganho de performance com a incorporação da virtualização aos processadores, através das tecnologias *Intel virtualization technology* (IVT) da *Intel* e *AMD virtualization* (AMD-V) da *AMD*. Elas possuem dois modos, um para execuções normais e para hipervisor, e outro específico para máquinas virtuais.

3.2.2 Vantagens

Em muitos casos empresas utilizam serviços distribuídos entre servidores físicos, como, por exemplo, servidores de e-mail, hospedagens e banco de dados, com isso existe uma ociosidade grande de recursos. Portanto uma das grandes vantagens da virtualização de sistema é um melhor aproveitamento destes recursos, alocando vários serviços em um único servidor físico e assim gerando um melhor aproveitamento do *hardware* (MOREIRA, 2006). Além disso, pode-se ter uma redução de custos com a administração e a manutenção dos servidores. Em um ambiente heterogênio pode-se também utilizar virtualização, pois ela permite a instalação de diversos sistemas operacionais em um único servidor. Considerando isto, esse tipo de virtualização favorece a implementação do conceito um servidor por serviço, que consiste em ter um servidor para cada serviço. Outro fator relevante que favorece a implementação desse conceito é o isolamento de serviços, ou seja, se ocorrer uma falha de segurança em um serviço, comprometerá todo o sistema e seus respectivos serviços (CARISSIMI, 2008).

Uma outra motivação para a utilização de virtualização do sistema consiste no

custo da energia elétrica. A economia de energia pode ser obtida através da implantação de servidores mais robustos para substituir dezenas de servidores comuns. Outros fatores como refrigeração do ambiente e espaço físico utilizado também podem ser reduzidos com a implantação de virtualização de servidores, e consequentemente, reduzem os custos de energia.

A portabilidade é outra vantagem da virtualização, que pode ser aplicada em *desktops*, pode-se citar como exemplo o desenvolvimento de *software* para diversos sistemas operacionais sem a necessidade de um computador para cada sistema operacional. Assim, máquinas virtuais em *desktops* são essenciais para ambientes de desenvolvimento, pois não comprometem o sistema operacional original (CARISSIMI, 2008).

4 ESTUDO DE CASO

5 PROPOSTA DE SOLUÇÃO

REFERÊNCIAS

BATISTA, A. C. **Estudo teórico sobre cluster linux**. 2007. Pós-Graduação(Administração em Redes Linux) — Universidade Federal de Lavras, Minas Gerais.

CARISSIMI, A. Virtualização: da teoria a soluções. In: **Minicursos do Simpósio Brasileiro de Redes de Computadores**. Porto Alegre: XXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, 2008.

COSTA, H. L. A. **Alta disponibilidade e balanceamento de carga para melhoria de sistemas computacionais críticos usando software livre**: um estudo de caso. 2009. Pós-Graduação em Ciência da Computação — Universidade Federal de Viçosa, Minas Gerais.

GONçALVES, E. M. **Implementação de Alta disponibilidade em máquinas virtuais utilizando Software Livre**. 2009. Trabalho de Conclusão (Curso de Engenharia da Computação) — Faculdade de Tecnologia e Ciências Sociais Aplicadas, Brasilia.

LAUREANO, M. A. P.; MAZIERO, C. A. Virtualização: conceitos e aplicações em segurança. In: **Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais**. Gramado - Rio Grande do Sul: VIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, 2008.

MAZIERO, C. A. **Sistemas Operacionais**: conceitos e mecanismos. 2013. Dissertação (Mestrado em Ciência da Computação) — DAInf UTFPR, Paraná.

MOREIRA, D. **Virtualização**: rode vários sistemas operacionais na mesma máquina. <Disponível em: `http://idgnow.com.br/ti-corporativa/2006/08/01/idgnoticia.2006-07-31.7918579158/#&panel1-3`>. Acesso em 5 de abril de 2016.

NøRVåG, K. **An Introduction to Fault-Tolerant Systems**. 2000. IDI Technical Report 6/99 — Norwegian University of Science and Technology, Trondheim, Noruega.

PANKAJ, J. **Fault tolerance in distributed system**. Nova Jérsei, Estados Unidos: P T R Prentice Hall, 1994.

PEREIRA FILHO, N. A. **Serviço de pertinência para clusters de alta disponibilidade**. 2004. Dissertação para Mestrado em Ciência da Computação — Universidade de São Paulo, São Paulo.

REIS, W. S. dos. **Virtualização de serviços baseado em contêineres**: uma proposta para alta disponibilidade de serviços em redes linux de pequeno porte. 2009. Monografia Pós-Graduação(Administração em Redes Linux) — Apresentada ao Departamento de Ciência da Computação, Minas Gerais.

ROUSE, M. **Hot spare**. <Disponível em: `http://searchstorage.techtarget.com/definition/hot-spare`>. Acesso em 12 de abril de 2016.

SILVA VIANA, A. L. da. **MySQL**: replicação de dados. <Disponível em: `http://www.devmedia.com.br/mysql-replicacao-de-dados/22923`>. Acesso em 21 de abril de 2016.

SMITH, J. E.; NAIR, R. The architecture of virtual machines. **IEEE Computer**, [S.l.], v.38, p.32–38, 2005.

SMITH, R. **Gerenciamento de Nível de Serviço**. <Disponível em: `http://blogs.technet.com/b/ronaldosjr/archive/2010/05/25/gerenciamento-de-n-237-vel-de-servi-231-o.aspx/`>. Acesso em 25 de março de 2016.

WEBER, T. S. **Um roteiro para exploração dos conceitos básicos de tolerância a falhas**. 2002. Curso de Especialização em Redes e Sistemas Distribuídos — UFRGS, Rio Grande do Sul.