
**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* |
| 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 tem uma grande importância 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 ou em um sistema 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

# 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 é definida como a redundância de *hardware* ou de *software* para que o serviço fique mais tempo disponível. 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 discutida 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 e para a tolerância a falhas. A abordagem de prevenção de falhas define-se como um projeto feito na criação de mecanismos para impedir que 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 fornece disponibilidade de um serviço mesmo com a presença de

falhas. 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 que os serviços fornecidos ficam disponíveis aos usuários. Um sistema é dito tolerante a falhas se ele pode 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 nos quais máquinas virtuais estão sendo 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. Este tema, virtualização, será detalhado no Capítulo 3.

A tolerância a falhas pode ser dividida em dois tipos que são: mascaramento; detecção, localização e reconfiguração. 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* e por fim resolver a falha (WEBER, 2002). O segundo tipo de tolerância a falahas será decrito na Seção 2.1.1.

### 2.1.1 Fases da tolerância a falhas

A classificação das fases de tolerância a falhas mais comuns são (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, devido alguma falha, 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 de processos de um sistema operacional, onde existe restrições para que um processo não gere problemas em outro processo, com isso as falhas de um programa estão restritas ao processo. COMO É FEITO ???
- 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 novo estado não ocorrido anteriormente. 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 que deve estar livre de erros. Para retornar ao estado anterior pode ser utilizados pontos de recuperação (*checkpoints*), e 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, e o automático quando existe um componente em espera para substituição. Exemplo de um reparo manual é um operador que efetua a troca de um disco rígido de um servidor. E um exemplo de reparo automático é 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. A redundância pode ser implementada ainda através do envio de sinais ou *bits* de controle junto aos dados, servindo assim para detecção de erros e até para correção (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 um exemplo de um sistema gerenciador de banco de dados *MySQL*, que pode ser configurado com um modelo de replicação do tipo *master-slave*, onde um servidor

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 |

principal (*master*) grava as operações em um arquivo, para então os servidores *slaves* recuperarem e executarem essas operações, com isso mantendo os dados sincronizados. Neste caso, tanto o servidor *master* quanto os *slaves* executam o serviço *MySQL* caracterizando assim uma redundância (SILVA VIANA, 2015). A redundância de *software* também pode ser implementada com o objetivo de tolerar falhas e *bugs* de um *software* crítico, sendo que este *software* é replicado em diferentes locais. Existem algumas técnicas que podem ser utilizadas para essa implementação. 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 por isso não é 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 e caso ocorra 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 que os serviços estão em execução e o tempo 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. A última coluna possui 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 pode conter descrição do serviço, requerimentos, horário de funcionamento, entre outros (SMITH, 2010).

# 3 VIRTUALIZAÇÃO

O conceito virtualização surgiu na década de 60, onde muitas vezes havia a necessidade de um usuário utilizar um ambiente individual, com suas próprias aplicações e totalmente isolado dos outros usuários. Este foi um dos principais motivos para a criação de máquinas virtuais, mais conhecida como *Virtual machine* (VM), que teve forte expansão com um dos principais sistemas comerciais com suporte a virtualização, sistema operacional *370* que foi desenvolvido pela *IBM*. Este 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 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*.

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, para suprir as necessidades dos componentes do sistema, de forma que uma aplicação desenvolvida para uma plataforma $X$ possa também executar em uma plataforma $Y$ (LAUREANO; MAZIERO, 2008).

Deve-se entender os diferentes tipos de interfaces existentes em sistemas de computação:

- Conjunto de instruções ou *Instruction set architecture* (ISA): é a interface básica, 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. Executam em modo não-privilegiado;
  - Instruções de sistema ou *System ISA*: são instruções exclusivamente acessíveis ao núcleo do sistema operacional. São executadas em modo privilegiado;

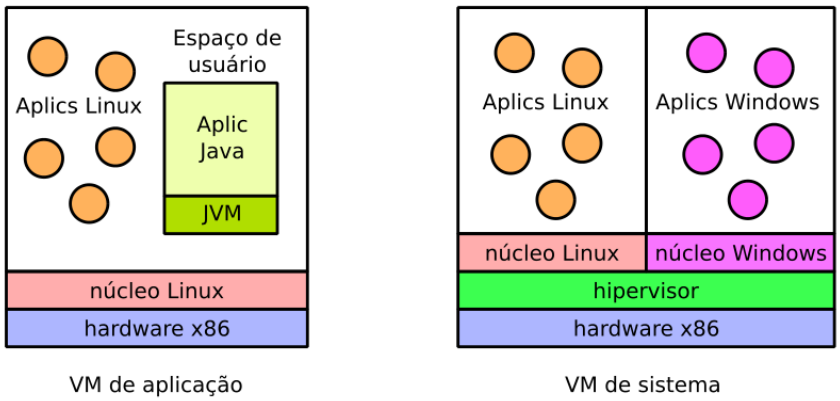


Figura 3.1: Máquinas virtuais de aplicação e de sistema.
Fonte: LAUREANO; MAZIERO (2008)

- Chamadas de sistema ou *syscalls*: são operações oferecidas pelo núcleo do sistema operacional para as aplicações dos usuários. Sendo que essas chamadas permitem acesso controlado aos dispositivos, memória e processador.

Máquinas virtuais podem ser divididas em dois grupos principais: máquinas virtuais de aplicação, detalhado na seção 3.1, e máquinas virtuais de sistema, detalhado na seção 3.2. A primeira faz a virtualização de uma aplicação e suporta apenas um processo ou aplicação. Já a máquina virtual de sistema suporta sistemas operacionais convidados, com suas aplicações executando sobre ele (Figura 3.1) (LAUREANO; MAZIERO, 2008).

## 3.1 Máquinas virtuais de aplicação

As máquinas virtuais de aplicação, também chamadas de máquinas virtuais de processo, é geralmente conhecida 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 do sistema hospedeiro.

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. Exemplos dessa definição são:

- Máquinas virtuais de linguagem de alto nível: este 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 nenhuma arquitetura real, mas pode ser executada em uma máquina virtual. Sendo assim para cada arquitetura ou sistema operacional deve existir uma máquina virtual que faça esse intermédio permitindo a execução de aplicações em sistemas operacionais diversos. Exemplos são: *Microsoft Common Language Infrastructure*, base do *.Net* e a máquina virtual Java (JVM) (CARISSIMI, 2008);
- Emulação no sistema operacional: ocorre quando as interfaces restringe-se às chamadas do sistema operacional, ou seja, é feito um mapeamento das

chamadas utilizada pela aplicação com as chamadas do sistema operacional da máquina real. Este tipo de 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 resumidamente permite executar aplicações *Windows* em plataformas *Unix*.

Na virtualização de aplicação também existem máquinas virtuais que utilizam as mesmas interfaces ISA da máquina real, com isso uma grande parte das instruções podem ser executadas diretamente, com exceção de instruções sensíveis, que serão devidamente tratadas. Alguns tipos de máquinas virtuais de aplicação que utilizam as interfaces da máquina real são detalhadas abaixo:

- Sistemas operacionais multi-tarefas: sistemas operacionais que suportam simultaneamente mais de um usuário também podem ser vistos como máquinas virtuais. Em um sistema multi-tarefa cada processo possui um "processador virtual" (devido a rápida troca de contextos do processador real), uma "memória virtual" (memória alocada para o processo) e outros recursos que podem ser acessados através de chamadas de sistema;
- Tradutores dinâmicos: esses tradutores analisam e otimizam o código de máquina para torná-lo mais eficiente. Essa otimização pode ser feita durante a carga do processo na memória ou durante a execução das instruções;
- Depuradores de memória: são sistemas de depuração de memória que detectam erros decorrentes do uso incorreto da memória. Um exemplo de depurador é o sistema *Valgrind*, que faz a execução em uma máquina virtual.

## 3.2 Máquinas virtuais de sistema

Máquina virtual de sistema também pode ser chamada de hipervisor ou monitor (*Virtual machine monitor* (VMM)), que é uma camada de *software* que possibilita que um ou mais sistemas operacionais convidados executem independentemente sobre uma mesma máquina real. O hipervisor prove uma interface ISA virtual que pode ou não ser igual a interface real, e virtualiza outros componentes de *hardware*, para então cada máquina virtual convidada ter seus próprios recursos isolados.

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 real com o sistema virtual;
- 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 físico em dois discos virtuais menores, sendo que esses discos virtuais são arquivos armazenados no disco físico. Sabendo que arquivos são uma abstração em um disco físico, pode-se dizer que virtualização não é apenas uma

camada de abstração do *hardware*, ela faz a reprodução do *hardware* (SMITH; NAIR, 2005).

Pode-se classificar ambientes de máquinas virtuais de sistema de duas formas, pela arquitetura de virtualização (Seção 3.2.1) ou pelo nível de virtualização (Seção 3.2.2), ambos serão vistos nas próximas seções.

### 3.2.1 Arquiteturas de virtualização

- 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, entre outros, e diponibiliza esses recursos para as máquinas virtuais. 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 este tipo suporta apenas um sistema operacional convidado para cada hipervisor. Exemplos de sistemas são o *VirtualBox* e o *QEmu*.

Sabendo essas definições 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.

### 3.2.2 Níveis de virtualização

- Virtualização de recursos: neste tipo de virtualização os recursos como memória e disco, além das instruções privilegiadas (*system ISA*) são virtualizadas. Somente a interface ISA de usuário é utilizada diretamente, por isso o desempenho do sistema convidado é mais próximo a um sistema executando diretamente sobre um *hardware*. O *VirtualBox* e o *VirtualPC da Microsoft* são exemplos de virtualização de recursos;
- Virtualização completa: na virtualização completa todas interfaces são virtualizadas. Sendo assim o hipervisor fornece uma interface distinta ao sistema operacional convidado. Esse tipo de virtualização possui um eficiência menor, por outro lado ele permite executar sistemas operacionais em plataformas distintas a qual foram projetadas inicialmente. Por exemplo, o *MS Virtual PC for MAC*, que permite executar o sistema *Windows* sobre plataforma de *hardware PowerPC*.

Tendo essas classificações, pode-se combiná-las para se obter quatro maneiras diferentes de implementar virtualização. Na Figura 3.2 tem-se essas combinações com seus respectivos exemplos.

### 3.2.3 Estratégias de virtualização

Pode-se dividir máquinas virtuais de sistema em dois tipos de estratégias distintas, são elas:

- Virtualização total: nesta estratégia todas as interfaces de acesso ao *hardware* são virtualizadas. Desta forma possibilita-se que sistemas operacionais convi-

dados executem como se estivessem diretamente sobre o *hardware*. A grande vantagem dessa virtualização é a possibilidade de um sistema convidado executar normalmente sem precisar ser modificado. Porém existe uma redução significativa no desempenho devido ao hipervisor intermediar todas as chamadas e operações do sistema convidado. Exemplo de virtualização total é o *QEmu*;

- Paravirtualização: esta prove um melhor acoplamento entre o sistema operacional 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, com isso haverá um melhor desempenho. Um exemplo de paravirtualização é o *Xen*.

A paravirtualização possui um desempenho melhor comparado a virtualização total pois a primeira acessa alguns recursos diretamente, sendo que o hipervisor apenas monitora informando os limites do convidado. 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.

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.

### 3.2.4 Um servidor por serviço

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 é um melhor aproveitamento destes recursos, alocando vários serviços em um único servidor físico e 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.

Uma outra motivação para a utilização de virtualização 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 virtualização favorece a implementação do conceito um servidor por serviço, que consiste em ter um servidor para cada serviço. Mas porque não colocar todos serviços em um único servidor? Muitas vezes com uma variedade de serviços é necessário diferentes sistemas operacionais, ou os serviços necessitam rodar nas

mesmas portas, portanto isto se torna inviável. Outro fator relevante que também favorece a implementação de um servidor por serviço é, caso exista uma falha de segurança em apenas um serviço, essa vulnerabilidade poderá comprometer todos os outros serviços (CARISSIMI, 2008).

## 3.3 Emulação

Emulação é a capacidade de uma aplicação ou um dispositivo imitar um determinado conjunto de *hardware*. Normalmente é executado como um aplicativo em um sistema operacional. A emulação é uma camada de *software* que possibilita a execução de uma plataforma em outra plataforma distinta, com isso obtem-se a portabilidade das máquinas virtuais hospedeiras (SILVA, 2009).

Este tipo de virtualização possui uma considerável redução de desempenho, pelo fato de toda comunicação com o *hardware* ser intermediada pela camada de *software* que faz a sua tradução. Por outro lado ela não visa desempenho, mas sim flexibilidade, por exemplo ela permite que desenvolvedores de *firmware* e de *hardware* simulem e validem aplicativos sem a presença do *hardware* real.

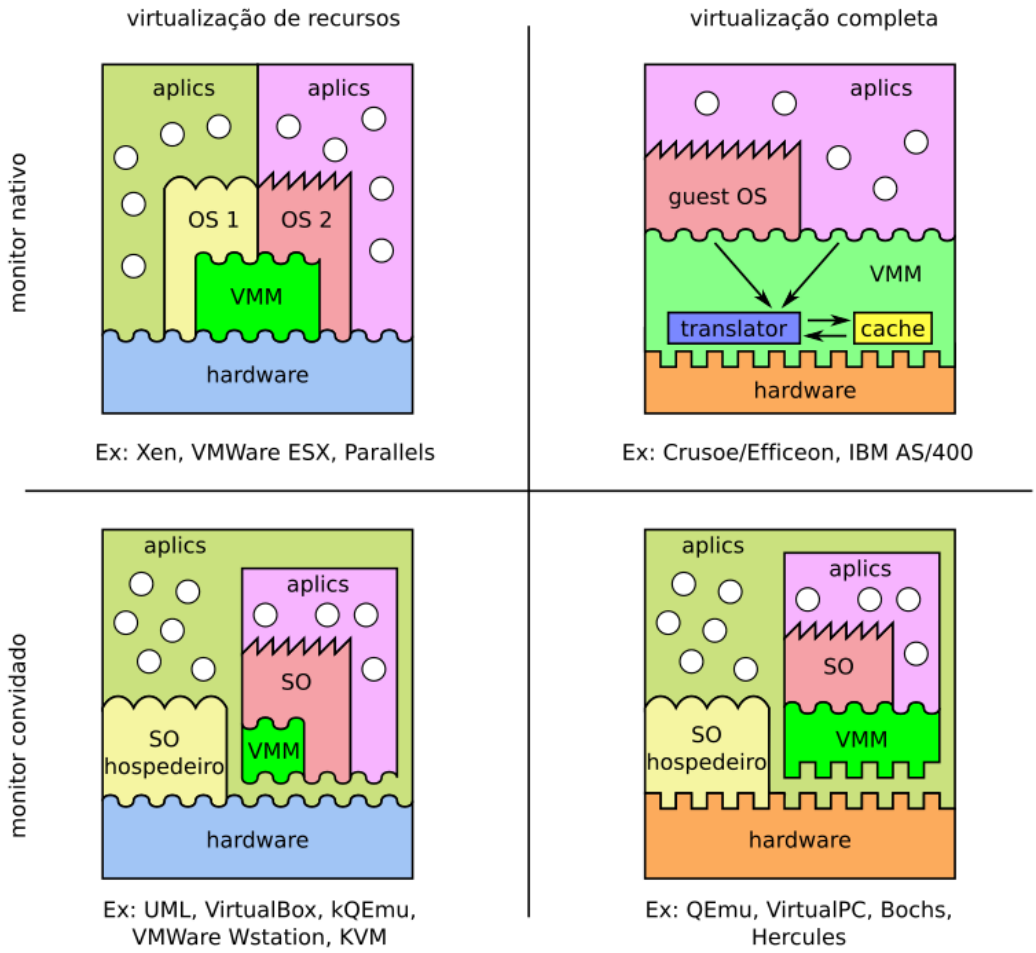


Figura 3.2: Classificação de máquinas virtuais de sistema.
Fonte: LAUREANO; MAZIERO (2008)

# 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.

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.

SILVA, Y. F. da. **Uma Avaliação sobre a Viabilidade do Uso de Técnicas de Virtualização em Ambientes de Alto Desempenho**. 2009. Trabalho de Conclusão (Curso de Ciência da Computação) — Universidade de Caxias do Sul, Caxias do Sul - Rio Grande do Sul.

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.