MARCELO PEREIRA DA SILVA

Adaptive Remus: Replicação de Máquinas Virtuais Xen com *Checkpointing* Adaptável

Dissertação apresentada ao Programa de Pós-Graduação em Computação Aplicada da Universidade do Estado de Santa Catarina, como requisito parcial para obtenção do grau de Mestre em Computação Aplicada.

Orientador: Guilherme Piêgas Koslovski
Coorientador: Rafael Rodrigues Obelheiro

JOINVILLE, SC
2015

S586a Silva, Marcelo Pereira da
Adaptive Remus: replicação de máquinas virtuais Xen com Checkpointing adaptável / Marcelo Pereira Silva. – 2015.
94 p. : il. ; 21 cm

Orientador: Guilherme Piêgas Koslovski
Coorientador: Rafael Rodrigues Obelheiro
Bibliografia: 89-94 p.
Dissertação (mestrado) – Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada, Joinville, 2015.

1.Computação aplicada. 2. Maquinas virtuais Xen. 3. Checkpointing . 4. Adaptive Remus .I. Koslovski, Guilherme Piêgas. II. Obelheiro, Rafael Rodrigues. III. Universidade do Estado de Santa Catarina. Programa de Pós-Graduação Computação Aplicada. IV. Título.

CDD 005. 43 – 23.ed.

Ficha catalográfica elaborada pela biblioteca setorial do CCT

MARCELO PEREIRA DA SILVA

ADAPTIVE REMUS: REPLICAÇÃO DE MÁQUINAS VIRTUAIS XEN COM CHECKPOINTING ADAPTÁVEL

Dissertação apresentada ao Curso de Mestrado Acadêmico Computação Aplicada como requisito parcial para obtenção do título de Mestre em Computação Aplicada na área de concentração "Ciência da Computação".

Banca Examinadora

Orientador:

Prof. Dr. Guilherme Piêgas Koslovski
CCT/UDESC

Membros

Prof. Dr. Maurício Aronne Pillon
CCT/UDESC

Profa. Dra. Andrea Schwertner Charao
UFSM

Coorientador:

Prof. Dr. Rafael Rodrigues Obelheiro
CCT/UDESC

Joinville,SC, 03 de julho de 2015.

Este trabalho é dedicado aos meus pais, irmãos, minha filha e minha esposa.

Agradecimentos

Agradeço primeiramente a Deus. Aos meus pais, Ari e Zenaide, por todo acolhimento e incentivo que sempre me proporcionaram. Minha esposa Taiza e minha filha Mariana, pela paciência e compreensão nos diversos momentos em que tive que estar ausente durante esta jornada. Aos colegas de trabalho e aos professores do Departamento de Ciências da Computação da UDESC Joinville, em especial meus orientadores, Guilherme e Rafael, pelo aprendizado e pela confiança depositada.

"Sobreviver é escolher,
escolher é renunciar."
(Olavo de Carvalho)

RESUMO

Com a dependência cada vez maior de computadores e redes, muitos sistemas precisam estar continuamente disponíveis para cumprir sua missão. A tecnologia de virtualização permite prover alta disponibilidade de forma conveniente e a um custo razoável: com o encapsulamento oferecido pelas máquinas virtuais (MVs), sistemas inteiros podem ser replicados em *software*, de forma transparente, eliminando a necessidade de *hardware* tolerante a faltas dispendioso. Remus é um mecanismo de replicação de MVs que fornece alta disponibilidade diante de faltas de parada. A replicação é realizada através de *checkpointing*, seguindo um intervalo fixo de tempo predeterminado. Todavia, existe um antagonismo entre processamento e comunicação em relação ao intervalo ideal entre *checkpoints*: enquanto intervalos maiores beneficiam aplicações com processamento intensivo, intervalos menores favorecem as aplicações cujo desempenho é dominado pela rede. Logo, o intervalo utilizado nem sempre é o adequado para as características de uso de recursos da aplicação em execução na MV, limitando a aplicabilidade de Remus em determinados cenários. Este trabalho apresenta Adaptive Remus, uma proposta de *checkpointing* adaptativo para Remus, que ajusta dinamicamente a frequência de replicação de acordo com as características das aplicações em execução. Os resultados indicam que a proposta obtém um melhor desempenho de aplicações que utilizam tanto recursos de processamento como de comunicação, sem prejudicar aplicações que usam apenas um dos tipos de recursos.

Palavras-chaves: Replicação, tolerância, *checkpointing*, frequência, adaptativo, Xen, Remus, MV.

ABSTRACT

With the ever-increasing dependence on computers and networks, many systems are required to be continuously available in order to fulfill their mission. Virtualization technology enables high availability to be offered in a convenient, cost-effective manner: with the encapsulation provided by virtual machines (VMs), entire systems can be replicated transparently in software, obviating the need for expensive fault-tolerant hardware. Remus is a VM replication mechanism for the Xen hypervisor that provides high availability despite crash failures. Replication is performed by checkpointing the VM at fixed intervals. However, there is an antagonism between processing and communication regarding the optimal checkpointing interval: while longer intervals benefit processor-intensive applications, shorter intervals favor network-intensive applications. Thus, any chosen interval may not always be suitable for the hosted applications, limiting Remus usage in many scenarios. This work introduces Adaptive Remus, a proposal for adaptive checkpointing in Remus that dynamically adjusts the replication frequency according to the characteristics of running applications. Experimental results indicate that our proposal improves performance for applications that require both processing and communication, without harming applications that use only one type of resource.

Key-words: Replication, tolerance, checkpointing, frequency, adaptive, Xen, Remus, VM.

Lista de ilustrações

Figura 1 – Máquinas virtuais de processo e de sistema. Adaptado de (CARISSIMI, 2008). 30
Figura 2 – Arquitetura de máquinas virtuais de sistema. Adaptado de (LAUREANO; MAZIERO, 2008). 31
Figura 3 – A MV0, ou domínio 0, é responsável por fornecer acesso aos recursos de *hardware* para as outras máquinas virtuais. Adaptado de (LAUREANO; MAZIERO, 2008). 33
Figura 4 – A relação entre falta, erro e falha. Adaptado de (OBELHEIRO, 2003). 36
Figura 5 – Taxonomia da segurança do funcionamento. Adaptado de (OBELHEIRO, 2003). 38
Figura 6 – No ambiente de execução de Remus, os hospedeiros são interligados por uma interface exclusiva de rede. 44
Figura 7 – O tráfego de saída de rede da MV protegida somente será liberado após o estado replicado ser confirmado pelo hospedeiro de *backup*. Enquanto esse evento não ocorre, os pacotes são armazenados em um *buffer* temporário no hospedeiro primário. 48
Figura 8 – O processo de replicação do mecanismo Remus. . . . 49
Figura 9 – Em Remus, o tempo de execução em modo especulativo varia de acordo com o tempo de replicação sempre que o intervalo definido for ultrapassado. . . 49
Figura 10 – Média da vazão entre os hospedeiros primário e *backup* com a MV ociosa. 52

Figura 11 – Tempo de execução e vazão média durante o *benchmark* de CPU. 52
Figura 12 – Tempo de execução e vazão média durante o *benchmark* de memória. 53
Figura 13 – Tempo de execução e vazão média durante o *benchmark* de E/S. 54
Figura 14 – Tempo de execução e vazão média durante a compilação do BIND. 55
Figura 15 – Tempo de transferência para *upload* e *download* . . . 56
Figura 16 – Tempo de transferência para *upload* e *download* sem o *buffer* de rede ativo. 56
Figura 17 – Número de requisições atendidas em 60 segundos e vazão média entre os hospedeiros durante a execução do *benchmark* Ab. 58
Figura 18 – A latência média percebida pelo cliente para cada operação de rede em um servidor de arquivos durante 120 segundos. 60
Figura 19 – Fluxograma do mecanismo proposto para *checkpointing* adaptável em Remus. 68
Figura 20 – Tempo de compilação do BIND para as quatro versões de Remus. 78
Figura 21 – Resultados obtidos com a execução do *benchmark* DaCapo Tomcat para as quatro versões de mecanismo Remus. 79
Figura 22 – Tempo para transferência de um arquivo de 50MB da MV para a máquina cliente. 80
Figura 23 – Para a aplicação NAS-MG, uma baixa frequência de *checkpointing*, como 100 ms, resulta em uma alta sobrecarga que afeta o desempenho da aplicação. . . . 81
Figura 24 – Resultado do *benchmark* RUBiS para as quatro versões do Remus em relação ao número de operações realizadas pelos clientes. 82
Figura 25 – Análise do uso de CPU imposta ao hospedeiro primário com a MV ociosa. 83
Figura 26 – Análise do uso de CPU no hospedeiro primário com o *benchmark* Dacapo Tomcat em execução. 84

Lista de tabelas

Tabela 1 – Número de operações executadas em um servidor SMB durante 120 segundos. 59
Tabela 2 – Latência média (em milissegundos) percebida pelo cliente para cada operação durante 120 segundos. . 59
Tabela 3 – Variáveis utilizadas no fluxograma da Figura 19. . . 66
Tabela 4 – Resumo dos testes executados. 77

Lista de abreviaturas e siglas

ACK	Abreviatura de *Acknowledgement*
API	*Application Programming Interface*
ARP	*Address Resolution Protocol*
AR	*Adaptive* Remus
AR+	*Adaptive* Remus fixado em modo rede
BF	Baixa Frequência
DRBD	*Distributed Replicated Block Device*
FSR	Fluxo de Saída de Rede
FTP	*File Transfer Protocol*
IP	Internet *Protocol*
LAN	*Local Area Network*
MMV	Monitor de Máquina Virtual
MV	Máquina Virtual
R25	Remus com intervalo fixo de 25 ms
R100	Remus com intervalo fixo de 100 ms
RF	Remus Flutuante

RX	Fluxo de Pacotes de Rede Recebidos
SCP	*Secure Copy*
SMB	*Server Message Block*
SO	Sistema Operacional
SSH	*Secure Shell*
TCP	*Transmission Control Protocol*
TDC	Tempo de Duração de um *Checkpointing*
TEMV	Tempo de Execução da Máquina Virtual
TX	Fluxo de Pacotes de Rede Transmitidos
VCPU	CPU Virtual
WAN	*Wide Area Network*
WWS	*Writable Working Set*

Sumário

1	**INTRODUÇÃO**	**23**
1.1	Objetivos do Trabalho	25
1.1.1	Objetivos Específicos	25
1.2	Principais Contribuições	25
1.3	Organização do Texto	26
2	**REVISÃO DA LITERATURA**	**27**
2.1	Virtualização de Recursos Computacionais	27
2.1.1	Tipos de Máquinas Virtuais	29
2.1.2	Monitor de Máquinas Virtuais Xen	32
2.1.3	Migração de MVs no Xen	33
2.2	Segurança do Funcionamento	35
2.2.1	Conceitos e Definições	36
2.2.2	Tolerância a Faltas em Sistemas Distribuídos	38
2.2.3	Técnicas de Replicação	39
2.3	Considerações Parciais	42
3	**O MECANISMO DE REPLICAÇÃO REMUS**	**43**
3.1	Funcionamento	43
3.1.1	Visão Geral	43
3.1.2	Replicação da Memória e CPU	45
3.1.3	Replicação do Disco	46
3.2	Questões de Projeto	46
3.2.1	Linearizabilidade	47
3.2.2	Frequência de *Checkpointing*	47
3.3	Avaliação Experimental da Sobrecarga Imposta pelo Remus	50
3.3.1	Aplicações sem Uso de Rede	51
3.3.1.1	Aplicação com Uso Intensivo de CPU	52
3.3.1.2	Aplicações com Uso Intensivo de Memória	53
3.3.1.3	Aplicação com Uso Intensivo de E/S	54
3.3.1.4	Compilação	54

3.3.2 Aplicações com Uso de Rede 55
3.3.2.1 Transferência de Dados 55
3.3.2.2 *Benchmark* Ab 57
3.3.2.3 *Benchmark* tbench 58
3.3.3 Discussão dos Resultados 58
3.4 Considerações Parciais 60

4 REPLICAÇÃO COM *CHECKPOINTING* ADAPTÁVEL 63
4.1 Visão Geral do Mecanismo 63
4.2 Modos de Operação 64
4.3 Alternância entre os Modos de Operação 64
4.4 Implementação do Mecanismo 67
4.5 Trabalhos Relacionados 69
4.5.1 Ambientes não Virtualizados 70
4.5.2 Ambientes Virtualizados 71
4.5.2.1 Migração de Máquinas Virtuais 71
4.5.2.2 Replicação de Máquinas Virtuais 71
4.6 Considerações Parciais 73

5 ANÁLISE EXPERIMENTAL 75
5.1 Visão Geral 75
5.2 Resultados Experimentais 76
5.2.1 Métricas 76
5.2.2 Aplicações Mistas (CPU e E/S) 77
5.2.2.1 Compilação 77
5.2.2.2 DaCapo Tomcat 78
5.2.3 Aplicações Sensíveis a Latência de Rede 79
5.2.4 Aplicações Mistas (CPU e Rede) 80
5.2.4.1 *Benchmark* NAS 80
5.2.4.2 RUBiS 81
5.2.5 Sobrecarga no Hospedeiro Primário 83
5.2.6 Discussão dos Resultados 84
5.3 Considerações Parciais 85

6 CONSIDERAÇÕES FINAIS 87
6.1 Trabalhos Futuros 89
6.2 Trabalhos Publicados 89

Referências 91

1

INTRODUÇÃO

Aplicações computacionais podem ter sua execução parcial ou totalmente comprometida pela ocorrência de faltas nos recursos de processamento, comunicação ou armazenamento por elas utilizadas. Para superar esse cenário, aplicações críticas recorrem a soluções de alta disponibilidade (algoritmos, arquiteturas, protocolos e infraestruturas) para garantir o seu funcionamento correto mesmo na ocorrência de faltas.

Paralelamente, os centros de processamento de dados têm explorado a virtualização de recursos computacionais para consolidação de servidores, economia no consumo de energia, e gerenciamento facilitado. Em um ambiente virtualizado, os usuários executam suas aplicações sobre um conjunto de máquinas virtuais (MVs), através da introdução de uma abstração entre o *hardware* físico (que hospeda a MV) e a aplicação final. A camada de abstração, chamada de monitor de máquinas virtuais (MMV) ou hipervisor, tem acesso a todo o estado de execução de uma MV, e por isso pode ser explorada para introduzir tolerância a faltas de forma transparente para a aplicação hospedada na máquina virtual. Motivado por esta oportunidade, novos mecanismo para tolerância a faltas foram desenvolvidos para diferentes hipervisores, como KVM Kemari (TAMURA et al., 2008), VMware vSphere Fault Tolerance (VMware, 2012a) e High Availability (VMware, 2012b), Paratus (DU; YU, 2009) e Remus (CULLY et al., 2008). Dentre esses mecanismos, Remus destaca-se por ser uma implementação de código-aberto, com o maior número de referências em trabalhos científicos da área (aproximadamente 10 vezes mais referência que Kemari).

Remus, um mecanismo para replicação de MVs residente no hi-

pervisor Xen (BARHAM et al., 2003) baseado no modelo de replicação primário-*backup* (BUDHIRAJA et al., 1993), encapsula as aplicações e o sistema operacional do usuário em uma MV, salvando dezenas de *checkpoints* por segundo. De forma assíncrona, todos os *checkpoints* salvos no hospedeiro primário são transferidos para um segundo hospedeiro, chamado de *backup*. Normalmente, os usuários interagem com a MV hospedada no primário, porém, quando uma falha de parada ocorre neste hospedeiro, a MV residente no *backup* é ativada e passa a responder pelo serviço em poucos milissegundos.

O processo de replicação de uma MV para atingir tolerância a faltas de parada pode ser interpretado como uma alta frequência de sincronização entre as réplicas existentes nos hospedeiros primário e *backup*. Cada *checkpoint* replicado compreende um conjunto de alterações no estado da MV desde o último *checkpoint* salvo. Salvar um *checkpoint* e posteriormente transferi-lo para um armazenamento estável, no caso o hospedeiro *backup*, é compreendido como *checkpointing* (LAMPSON; STURGIS, 1979). Uma alta frequência de replicação reduz o trabalho que deverá ser refeito no *backup* quando sua réplica, por necessidade, tornar-se ativa (ELNOZAHY et al., 2002).

A sobrecarga causada pelo processo de replicação no desempenho das aplicações hospedadas não é desprezível (SILVA; KOSLOVSKI; OBELHEIRO, 2014) (GEROFI; ISHIKAWA, 2011). Em suma, quanto mais alta a frequência de *checkpointing*, menor a latência na comunicação para o cliente que interage com a MV protegida. Em relação a aplicações dependentes apenas de processamento, uma baixa frequência de *checkpointing* representa menos interrupções à sua execução (intervalos maiores para *checkpointing* representam uma baixa frequência, enquanto intervalos menores uma alta frequência). Frente a estas características antagônicas, soma-se o fato que o intervalo para *checkpointing* utilizado por Remus é fixo. Embora as características das aplicações hospedadas em relação ao uso de recursos, como comunicação ou processamento, possam ser conhecidas, não é trivial a escolha de um intervalo de *checkpointing* ideal.

Aplicações com características mistas de uso de recursos aumentam a complexidade da escolha. Uma frequência alta de *checkpointing* priorizará parte da aplicação voltada a comunicação, porém, prejudicará a outra que necessita de mais tempo de processamento. A escolha de um intervalo médio não representa uma boa opção, uma vez que a aplicação hospedada nunca experimentará uma frequência que seja a melhor possível de acordo com suas características de uso de recursos. Ainda, dependendo da carga de processamento a qual a MV está

submetida, é possível que o intervalo predeterminado para *checkpointing* seja superado, uma vez que um novo *checkpoint* somente será salvo após o atual estiver completamente armazenado no hospedeiro *backup*. Durante este período, a MV se mantém em execução pelo tempo necessário, privilegiando o tempo de processamento e consequentemente aumentando a latência da comunicação na visão do cliente. O aumento da latência é inerente a sistemas de tolerância a faltas, uma vez que uma requisição deve primeiramente ser replicada no *backup* para posteriormente ser respondida, o que garante para o cliente uma visão idêntica de estado em caso de uma falta no *primário*.

1.1 Objetivos do Trabalho

Frente as limitações especificadas, o presente trabalho propõe uma abordagem para diminuir a sobrecarga imposta às aplicações protegidas pelo *software* Remus durante seu período livre de faltas.

1.1.1 Objetivos Específicos

- Analisar o sobrecarga imposta pelo mecanismo de replicação Remus;
- Introduzir um mecanismo de *checkpointing* adaptativo, ajustando seu intervalo em função do comportamento das aplicações hospedadas;
- Diminuir a sobrecarga nas aplicações comunicantes.

1.2 Principais Contribuições

Os resultados experimentais indicam que o uso de *checkpointing* adaptativo para a replicação de MVs com Remus, principalmente para aplicações comunicantes, é justificado. Em aplicações unicamente comunicantes, obteve-se uma diminuição de aproximadamente 93% no tempo de transferência devido à diminuição da sua latência, se comparado a Remus com um intervalo fixo de 25 ms. Para aplicações dependentes exclusivamente de tempo de processamento, o desempenho foi sempre semelhante a versão nativa de Remus sob uma baixa frequência de *checkpointing* (100 ms). Em relação a aplicações mistas, que alternam entre o uso de recursos como rede e processamento, a sobrecarga

imposta pela replicação diminuiu aproximadamente 38%, quando comparado a Remus com 25 ms de intervalo. Os resultados serão analisados no Capítulo 5.

1.3 Organização do Texto

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 traz uma revisão da literatura em relação a virtualização e tolerância a faltas em sistemas distribuidos. O Capítulo 3 detalha o mecanismo de replicação de MVs implementado pelo Remus e salienta suas limitações. O Capítulo 4 apresenta a versão *Adaptive Remus* e os trabalhos relacionados, enquanto os resultados experimentais são discutidos no Capítulo 5. Por fim, o Capítulo 6 conclui o trabalho, apresentando perspectivas para trabalhos futuros.

2

REVISÃO DA LITERATURA

Este capítulo apresenta uma revisão literária a respeito de virtualização e segurança de funcionamento *(dependability)*. Inicialmente são abordadas as propriedades básicas para um sistema computacional virtualizado, como os tipos e monitores de máquinas virtuais existentes, além de técnicas de migração e suas principais características. Posteriormente, uma taxonomia sobre a segurança de funcionamento de um sistema, que aborda seus atributos, meios e ameaças, é apresentada. Destacam-se no texto técnicas de tolerância a faltas e replicação, uma vez que são de fundamental importância para o entendimento deste trabalho. Por fim, uma seção com considerações parciais em relação as técnicas utilizados pelo Remus, conclui o capítulo.

2.1 Virtualização de Recursos Computacionais

Apesar de atual no mundo da tecnologia da informação, o termo virtualização não é exatamente uma novidade. Em meados da década de 70 era comum que os computadores de grande porte *(mainframes)* de diferentes modelos, ainda que do mesmo fabricante, tivessem cada um seu próprio sistema operacional. A tendência naquela época era fornecer a cada usuário um ambiente monousuário completo, com sistema operacional e aplicações completamente independentes dos ambientes dos demais usuários. Como na época o *software* não incluía apenas a aplicação, mas sim todo o ambiente operacional no qual ele operava, era necessário permitir a migração e execução de *software* legado entre os *mainframes*. Introduziu-se o termo máquina virtual (MV), implantando inicialmente na linha 370 da IBM, onde havia um modelo

de MV portado para cada uma de suas plataformas. Na década de 80, com a popularização do computador pessoal através da diminuição dos custos com *hardware*, o foco em tecnologias de virtualização diminuiu. Afinal era menos oneroso adquirir um computador pessoal para cada usuário do que investir em sistemas de grande porte, caros e complexos. Atualmente, com a disseminação das redes e dos sistemas distribuídos, além do enorme poder computacional dos processadores, a virtualização ressurgiu (LAUREANO; MAZIERO, 2008) (CARISSIMI, 2008).

Uma MV é uma camada de *software* que oferece um ambiente completo similar a uma máquina física, possibilitando a execução de sistemas operacionais distintos e independentes, com acesso a rede, memória, disco, CPU e outros dispositivos físicos de forma compartilhada. Como é possível executar várias MVs em uma mesma máquina física, também chamada de hospedeiro, possibilita-se usufruir ao máximo seu poder de processamento. Essa condição, conhecida como consolidação de servidores, resulta na diminuição de máquinas físicas e consequentemente na redução de custos como refrigeração, infra-estrutura de cabeamento, espaço físico, consumo de energia elétrica e manutenção. Além disso, a virtualização permite a portabilidade, possibilitando que uma MV seja migrada de um hospedeiro para outro, seja por questões de desempenho, *backup*, segurança ou simplesmente por faltas ou manutenções de *hardware*.

Um ambiente de MV é composto por três partes básicas:

- **o hospedeiro** (máquina física) que contém os recursos reais de *hardware* e *software* do sistema;
- **o sistema virtual**, ou sistema convidado, que executa sobre o sistema virtualizado;
- **o monitor de máquinas virtuais (MMV)** que constrói as interfaces virtuais a partir da interface real.

Em (POPEK; GOLDBERG, 1974), foram introduzidas três propriedades primárias para eficiência de um sistema computacional virtualizado:

- **Equivalência**: Um programa executando em uma MV deve apresentar o mesmo comportamento de quando está em execução diretamente sobre a mesma máquina física. Porém, há duas exceções permitidas, como eventualmente um maior tempo para execução de algumas instruções e a possibilidade de conflito de recursos quando houver outra MV compartilhando a mesma máquina física.

- **Eficiência**: todas as instruções de máquina enviadas ao processador virtual (provido pelo MMV) devem ser executadas diretamente no processador real da máquina, sem qualquer intervenção do MMV.

- **Controle de Recursos**: o MMV deve possuir controle total sobre os recursos da máquina física. Nenhum programa em execução em uma MV deve acessar recursos que não foram previamente disponibilizados pelo MMV. O MMV também tem o poder de resgatar recursos já alocados a qualquer momento.

2.1.1 Tipos de Máquinas Virtuais

Há diversas possibilidades de implementação de sistemas virtualizados, cada uma com diferentes características em relação a eficiência e equivalência. Os ambientes de MVs podem ser divididos em duas grandes famílias (LAUREANO; MAZIERO, 2008) (CARISSIMI, 2008):

- MV de processo: o conjunto MMV e aplicação é visto como um único processo dentro do sistema operacional subjacente. Um aplicação Java, por exemplo, tem seu próprio MMV, e é tratado como um processo dentro do sistema operacional Linux (sistema operacional subjacente que dispõe de acesso direto ao *hardware* da máquina física).

- MV de sistema: foco deste trabalho, a MV de sistema possui seu próprio sistema operacional e tem a ilusão de estar executando diretamente sobre uma plataforma de *hardware*. Porém, há uma camada de *software* entre o sistema operacional e o *hardware* para proteger o acesso direto deste aos recursos físicos.

A Figura 1 apresenta a composição dos tipos de MVs.

Na Figura 1(a), a aplicação executa sobre sua própria MV de processo. A MV utiliza as funcionalidades fornecidas pelo sistema operacional nativo da máquina física através de instruções não privilegiadas. Já a Figura 1(b) mostra um MMV com uma única MV responsável por prover recursos físicos e de CPU aos sistemas convidados. É possível executar mais de uma MV simultaneamente. Este ambiente é conhecido também como virtualização em nível de *hardware*, uma vez que o MMV se localiza imediatamente sobre o *hardware* para abstrair recursos para as MVs.

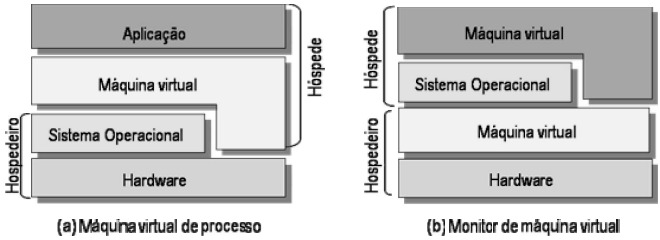


Figura 1 – Máquinas virtuais de processo e de sistema. Adaptado de (CARISSIMI, 2008).

Outros atributos associados aos MMVs incluem (ROSENBLUM, 2004):

- **Isolamento**: um *software* em execução em uma MV não pode ver, influenciar ou modificar outro *software* em execução no MMV ou em outra MV. Erros de *software* ou aplicações maliciosas devem ficar contidas na própria MV.

- **Compatibilidade**: todos os sistemas operacionais ou aplicações desenvolvidas para um determinado *hardware* físico deverão também ser executados em MMVs que abstraem o mesmo tipo de *hardware*.

- **Performance**: o MMV gerencia os recursos físicos de modo que o *hardware* virtual da MV seja direcionado diretamente para o *hardware* da máquina física. Isso permite que o desempenho de uma aplicação hospedada seja próximo de outra que execute diretamente sobre *hardware*.

- **Encapsulamento**: como o MMV tem acesso e controle sobre o estado interno de cada MV em execução, é possível que ele salve *checkpoints* de cada uma periodicamente ou apenas em situações especiais, como antes de alguma atualização de *software* (atualização do sistema operacional de uma MV, por exemplo). Além de *backups*, salvar *checkpoints* também é necessário para migrar uma MV entre MMVs em hospedeiros distintos.

Em relação a arquitetura, existem basicamente dois tipos de MMVs (LAUREANO; MAZIERO, 2008):

- **Nativo**: também conhecido como tipo 1. O MMV atua diretamente sobre o *hardware* da máquina física (ou real) sem um sistema operacional subjacente. Sua função é multiplexar os recursos como memória, discos e interfaces de rede, de forma que a MV veja um conjunto de recursos próprio e independente. São exemplos desta abordagem IBM OS/370, VMware ESX Server e Xen.

- **Convidado**: também conhecido como tipo 2. O MMV executa como um processo de um sistema operacional subjacente que tem acesso direto ao *hardware* da máquina física. O MMV utiliza os recursos oferecidos pelo sistema operacional nativo para oferecer recursos virtuais para o sistema operacional de seu hospedeiro. Normalmente, apenas uma MV pode ser ser executada em cada MMV hospedado. Para mais MVs, mais monitores devem ser lançados. QEMU, VirtualBox e VMware Workstation adotam esta estrutura.

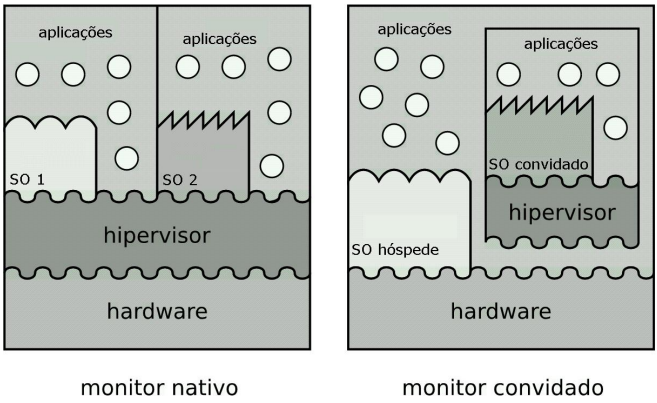


Figura 2 – Arquitetura de máquinas virtuais de sistema. Adaptado de (LAUREANO; MAZIERO, 2008).

A Figura 2 ilustra a diferença entre os modos nativo e convidado. Enquanto no primeiro há um único MMV (hipervisor) para diversas MVs, no segundo há apenas um sistema operacional com acesso direto aos recursos de *hardware* da máquina física. Este é responsável por prover os recursos a outros MMVs que executam seus próprios sistemas operacionais. Em relação a desempenho, segundo (POPEK; GOLDBERG, 1974), é aconselhável que o MMV execute diretamente sobre o *hardware* sempre que possível.

Existem diversas estratégias e técnicas de virtualização, como a virtualização total e paravirtualização, aonde a forma com que os recursos físicos são abstraídos para a MV pelo MMV se diferem. Porém, é necessário ressaltar que face o atual suporte de *hardware* à virtualização presente nos processadores Intel e AMD, ambas tem apresentado desempenhos semelhantes (CARISSIMI, 2008). Além destas, existe também a tradução dinâmica, em que partes do código binário do sistema convidado e suas aplicações são traduzidas dinamicamente pelo monitor. As sequências de instruções emitidas pelo sistema convidado são analisadas, reorganizadas e traduzidas em novas sequências de instruções, objetivando desempenho (através de um *cache* para instruções similares) e adaptabilidade (LAUREANO; MAZIERO, 2008).

2.1.2 Monitor de Máquinas Virtuais Xen

Dentre os hipervisores mais difundidos, pode-se citar *VMware*, Xen, QEMU, JVM, KVM e Microsoft Hyper-V e Virtual PC. Como o foco deste trabalho restringe-se ao MMV Xen, uma vez que é o monitor no qual Remus executa, este será o único a ser abordado nesta seção. Mais detalhes a respeito dos hipervisores citados podem ser obtidos em (LAUREANO; MAZIERO, 2008) e (CARISSIMI, 2008), ou nas documentações originais.

Atualmente o MMV Xen oferece suporte tanto para a virtualização completa quanto para a paravirtualização. Trata-se de um monitor do modo Nativo, capaz de controlar os recursos de *hardware* disponíveis para cada MV, como memória, processamento e comunicação. Uma MV principal, única, chamada de domínio 0 (ou MV0), é composta por um sistema operacional Linux modificado, dotado de *drivers* de dispositivos da máquina física e de acesso privilegiado ao *hardware*. É necessária para criar, iniciar, terminar e fornecer acesso aos recursos físicos para outras MVs hospedadas. Todas as máquinas virtuais hospedadas, exceto a MV0, são chamadas de domínios U (*unprivileged*), ou convidados.

Os domínios U podem ser classificados como paravirtualizados (U-PV) e virtualizados (U-HVM - *Hosted Virtual Machines*). Domínios U-PV, por terem um núcleo modificado com suporte a virtualização, possuem *drivers* para acesso a rede e a disco, têm consciência de que são virtualizados e controlados por um MMV e sabem da existência de outras MVs. Domínios U-HVM não são modificados e tem a ilusão de que estão executando diretamente sobre o *hardware* de uma máquina física, e sequer reconhecem a existência de outras MVs. O acesso a

disco e rede é realizado através de um emulador (QEMU), em execução no Domínio 0, vinculado a cada MV (U-HVM). U-HVM requer um processador habilitado para virtualização. Ainda, também é possível instalar *drivers* otimizados para domínios U-HVM para melhorar seu desempenho. Neste caso, são classificados como domínios PV-HVM.

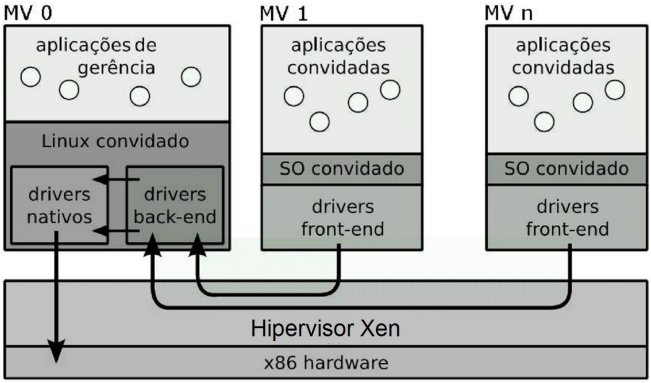


Figura 3 – A MV0, ou domínio 0, é responsável por fornecer acesso aos recursos de *hardware* para as outras máquinas virtuais. Adaptado de (LAUREANO; MAZIERO, 2008).

A Figura 3 ilustra o funcionamento de um domínio U-PV no Xen. A MV0 é a única MV hospedada que interage e detém os *drivers* nativos do hospedeiro. As demais MVs, 1 e n, são do tipo U-PV, portanto já possuem *drivers* de rede e disco para comunicação com o MMV. Uma MV U-HVM, por não ter seu sistema operacional previamente modificado para executar de maneira virtualizada, necessita de um *firmware*, emulado pelo Xen para simular uma BIOS, para ter a ilusão de que está diretamente em uma máquina física. Além disso, como não possui *drivers* específicos para rede e disco, uma instância QEMU é criada para cada MV na MV0.

2.1.3 Migração de MVs no Xen

Devido a propriedade de encapsulamento que permite ao MMV salvar um *checkpoint* completo do estado de uma MV, destaca-se a facilidade e o baixo custo para clonar (ou mover) uma MV entre hospedeiros. Em casos em que não há necessidade de alta disponibilidade para as aplicações hospedadas, basta que a MV seja copiada para um

novo hospedeiro para posteriormente, quando necessário, ser ativada. Também é possível ativar manualmente o clone de uma MV em um segundo hospedeiro, seja devido a falhas de *software* ou *hardware* no hospedeiro de origem, ou simplesmente para sua manutenção preventiva. São exemplos simples e de baixo custo, não tolerantes a faltas de parada, que geralmente servem como uma estratégia de *backup*.

Para ambientes em que os sistemas hospedados exigem alta disponibilidade de execução, a migração de uma MV também pode ser executada, seja para balanceamento de carga, consolidação de servidores, realocação de recursos ou até mesmo para manutenção preventiva no hospedeiro. Para tanto, o processo da migração deve passar ao sistema operacional convidado e suas aplicações, bem como para seus clientes remotos, a ilusão de que o hospedeiro não foi alterado. Contudo, é possível que breves períodos de indisponibilidade durante a migração sejam perceptíveis para o cliente, assim como a melhora no desempenho das aplicações executadas na MV após a migração completa (desde que esta tenha sido movida para um hospedeiro com mais recursos) (MEDINA; GARCÍA, 2014).

O processo de migração para MVs que demandam alta disponibilidade de execução exige que o estado completo da MV, incluindo disco, rede e memória, seja transferido para o hospedeiro de destino. Num ambiente local, a migração do estado do disco geralmente não é considerada, uma vez que é comum o uso de discos compartilhados. Para manter as conexões de rede após a migração de uma MV dentro de uma mesma rede local, é necessário que a MV mantenha seu mesmo endereço IP, para isso soluções baseadas em ARP são empregadas. Em relação a memória, há diversos estudos já conhecidos e basicamente dividem-se em *precopy* (pré-cópia) e *postcopy* (pós-cópia) (MEDINA; GARCÍA, 2014). Na pós-cópia, a execução da MV no hospedeiro de origem sofre uma pausa. Apenas o estado do processador é transferido ao hospedeiro de destino, onde posteriormente a MV retoma sua execução sem qualquer conteúdo na memória. Caso necessite acessar alguma página na memória que ainda não tenha sido transferida, a MV sofre uma pausa temporária e o conteúdo faltante é então transferido do hospedeiro original. A pré-copia, por sua vez, considera basicamente três fases:

- ***Push***: com a MV no hospedeiro de origem em execução, inicialmente todo conteúdo da memória é transferido ao hospedeiro de destino. Rodadas posteriores transferem apenas o conteúdo da memória que tenha sido alterado desde a última rodada. Este

procedimento, que visa a consistência da memória no destino, é conhecido por cópia iterativa.

- ***Stop-and-copy***: a execução da MV no hospedeiro primário sofre uma pausa e todo conteúdo da memória ainda não migrado é transferido ao hospedeiro de destino. A MV retoma sua execução no hospedeiro de destino.

- ***Pull***: já em execução no novo hospedeiro, caso a MV necessite acessar alguma página ainda não copiada, a mesma pode ser transferida do hospedeiro de origem.

Por sua vez, *Live Migration* é um mecanismo de migração de MVs que garante um tempo mínimo de indisponibilidade e interrupção dos serviços na visão do cliente. Implementada por (CLARK et al., 2005) para migração de MV no MMV Xen, considera apenas duas fases; *push* e *stop-and-copy*. Na fase inicial, (*Push*), é utilizada uma reserva de largura de banda, que controla tanto o fluxo de rede das aplicações hospedadas quanto o necessário para a migração. A cópia iterativa, por sua vez, também diminui a quantidade de dados transferidos a cada rodada. Por fim, o número de rodadas que precedem a fase *stop-and-copy* deve ser limitado (através de análises do *WWS: writable working set*). Além disso, simultaneamente, também há o controle de recursos da CPU durante o período da migração. Fatores como o tempo total de migração de uma MV, bem como seu período de indisponibilidade, devem ser considerados antes da escolha de um processo de migração. Uma migração efetuada por demanda, como é o caso da pós-cópia, pode levar a um tempo total de migração muito alto, além do fato de que o hospedeiro de origem precisa estar em execução até o final da migração. Por outro lado, uma única etapa *stop-and-copy*, pode levar a um tempo de indisponibilidade muito elevado. A fase *stop-and-copy* é periodicamente utilizada pelo Remus para salvar *checkpoints* em um hospedeiro *backup*, conforme será explicado no Capítulo 3.

2.2 Segurança do Funcionamento

A operação de sistemas computacionais pode ser ameaçada por diversos fatores, incluindo problemas de *hardware* e *software*, desastres naturais e violação de segurança. O domínio de segurança do funcionamento (*dependability*) tem como foco a concepção e análise de sistemas computacionais nos quais se possa depositar uma justificada confiança

diante de ameaças que ponham em risco o seu funcionamento correto (AVIZIENIS et al., 2004).

2.2.1 Conceitos e Definições

As ameaças à segurança do funcionamento de um sistema podem ser classificadas em **faltas**, **erros** e **falhas**. Um **erro** pode ser definido como uma parte do estado do sistema que pode levar a uma falha, e uma **falha** ocorre quando um erro afeta o funcionamento do sistema. A causa de um erro é chamada de **falta**. A falha de um subsistema pode gerar uma falta no sistema que o contém. Por exemplo, um capacitor de uma fonte de energia de um computador pode sofrer um dano. Esta falta pode levar a uma falha, comprometendo ou não seu funcionamento. Para o computador, a falha de uma fonte é interpretada como uma falta, que pode ou não comprometer ser funcionamento, uma vez que podem haver fontes de energia redundantes. A Figura 4 ilustra a relação entre falta, erro e falha em relação as interfaces de serviço, ou sistemas e subsistemas.

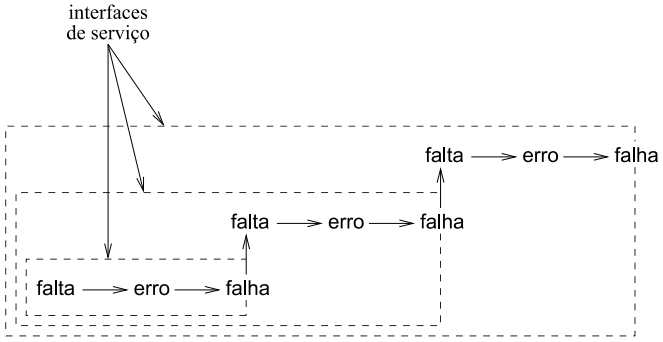


Figura 4 – A relação entre falta, erro e falha. Adaptado de (OBELHEIRO, 2003).

A segurança do funcionamento compreende os seguintes atributos:

- **disponibilidade**: capacidade de fornecer o serviço correto;
- **confiabilidade**: continuidade do serviço correto;
- **segurança contra riscos**: ausência de consequências catastróficas para os usuários e para o ambiente;

- **confidencialidade**: ausência de revelação não autorizada de informações;
- **integridade**: ausência de alterações indevidas no estado do sistema;
- **manutenibilidade**: capacidade de sofrer reparos e modificações.

Segundo (OBELHEIRO, 2003), "Nem sempre um sistema possui as mesmas exigências em relação a todos estes atributos: por exemplo, todos os sistemas necessitam de um nível mínimo de disponibilidade, mas nem todos possuem requisitos de confidencialidade ou de segurança contra riscos. A ênfase dada a cada um dos atributos depende da aplicação a qual se destina o sistema".

Os meios para se atingir a segurança do funcionamento de um sistema podem ser agrupados em quatro maiores categorias (AVIZIENIS et al., 2004):

- **prevenção de faltas**: meios para prevenir a ocorrência ou a introdução de faltas;
- **tolerância a faltas**: meios para evitar falhas de serviço na presença de faltas;
- **remoção de faltas**: meios para reduzir a quantidade ou a gravidade das faltas;
- **previsão de faltas**: meios para estimar a quantidade atual, a incidência futura e as prováveis consequências de faltas no sistema.

A semântica de falhas de um sistema tolerante a faltas define as maneiras pelas quais um sistema pode falhar. A hipótese de faltas define o tipo e a frequência das faltas que serão toleráveis. Segundo (Nacamura Jr., 1996) (AVIZIENIS et al., 2004), os tipos de faltas são classificadas em:

- **faltas por valor**: fazem com que o sistema forneça respostas incorretas;
- **faltas de temporização ou de desempenho**: fazem com que o sistema responda fora do intervalo de tempo especificado. As faltas de temporização podem ser divididas em **omissão**: o sistema deixa de responder um ou mais requisições; e **parada *(crash)***: o sistema deixa de responder as requisições sistematicamente até que o sistema volte ao seu estado normal (livre de falhas);

- **faltas bizantinas**: o sistema fornece respostas incorretas para um conjunto de requisições. Este tipo de falta também é conhecido como **arbitrária**.

Em suma, a taxonomia da segurança de funcionamento de um sistema pode ser observada na Figura 5. Uma exposição mais abrangente em relação a classificação de erros, falhas e faltas pode ser encontrada em (AVIZIENIS et al., 2004).

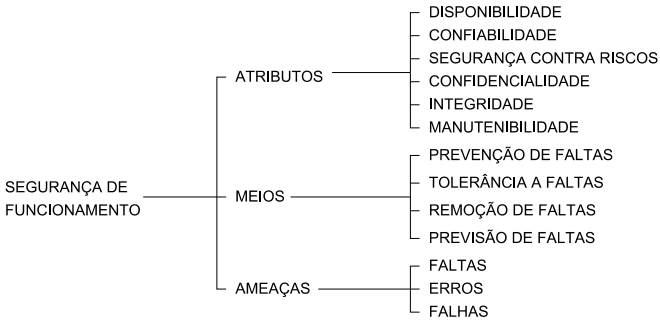


Figura 5 – Taxonomia da segurança do funcionamento. Adaptado de (OBELHEIRO, 2003).

2.2.2 Tolerância a Faltas em Sistemas Distribuídos

Segundo (GUERRAOUI; SCHIPER, 1997), um modelo de sistema distribuído pode ser generalizado como um sistema que consiste em um conjunto de processos comunicantes com troca de mensagens. Um processo pode estar em um estado correto ou incorreto. Um processo correto se comporta de acordo com suas especificações, enquanto um incorreto é um processo que ou sofreu uma falta de parada ou se comporta de forma maliciosa ou simplesmente está em desacordo com as suas especificações.

Um sistema tolerante a faltas requer o uso de redundância para manter o fornecimento do serviço de acordo com sua especificação, mesmo na presença de faltas. A redundância pode ser por *software*, *hardware* ou temporal. A implementação de um sistema tolerante a faltas envolve quatro fases (Nacamura Jr., 1996):

- **detecção de erros**: uma falta só pode ser detectada após ter gerado um erro. É neste momento que os mecanismos de tolerância a faltas devem ser acionados;

- **confinamento**: intervalo entre a manifestação do erro e sua detecção. Neste período outros componentes podem ser afetados, se faz necessário estabelecer limites para a propagação do erro;

- **processamento de erros**: trata-se do período após a detecção de um erro, cujo objetivo é removê-lo e retornar para um sistema livre de faltas. Duas técnica são empregadas no que se refere a processamento de erros:

  - **compensação de erros:** também conhecida como mascaramento de faltas, estas técnicas envolvem redundâncias de informação ou processamento. Consiste em mascarar os efeitos de elementos faltosos, garantindo a resposta correta mesmo sob a presença de faltas.

  - **recuperação de erros:** consiste em substituir um estado errôneo por um estado sem erro. Há dois tipos de recuperação de erros: i) *recuperação em retrocesso*: informações sobre o estado do sistema são regularmente armazenadas em pontos específicos, chamados de pontos de restauração ou *checkpoints*. Após a detecção de um erro, o estado do sistema pode ser restaurado com os valores do último ponto de restauração salvo; e ii) *recuperação por avanço*: consiste em transformar um estado errôneo em um novo estado, livre de erros. A recuperação por avanço não utiliza pontos de restauração.

- **tratamento de faltas**: consiste no uso de procedimentos para impedir que uma falta seja reativada, levando o sistema novamente para um estado errôneo. A diagnose de faltas, primeira etapa do tratamento, consiste em determinar a localização e a natureza das faltas. A segunda etapa, passivação de faltas, consiste em prevenir que as faltas sejam novamente ativadas.

2.2.3 Técnicas de Replicação

As técnicas de replicação são utilizadas para duas finalidades. A primeira é desempenho, já que permitem reduzir a carga imposta a um servidor único centralizado, além de proporcionar uma melhor escalabilidade. A segunda é aumentar a disponibilidade de um serviço, eliminando um ponto único de falha, já que o serviço continuará a ser fornecido mesmo em caso de falha em algum nó. As técnicas de

replicação variam de acordo com o grau de sincronismo, e podem ser classificadas em modelos de implementação (Nacamura Jr., 1996):

- **replicação ativa**: de forma paralela, todas as réplicas recebem e processam as requisições, produzindo respostas iguais na ausência de faltas. Em relação ao processamento de erros, utilizam o modelo de compensação de erros. Em relação a semântica de falhas, são comumente utilizadas para tratar faltas bizantinas, mas também suportam faltas de parada. Segundo (GUERRAOUI; SCHIPER, 1997) um cliente jamais terá que reenviar uma solicitação devido a uma falta de parada, já que todas as réplicas operam no mesmo estado ao mesmo tempo. Uma falta de parada neste modelo de replicação não significa um aumento da latência na percepção do cliente, uma vez que, ao contrário do que ocorre no modelo passivo, todas as réplicas processam todas as requisições simultaneamente.

- **replicação passiva**: ou primário - *backup*. Nesse modelo apenas uma réplica (privilegiada) recebe, processa e responde as requisições. As demais réplicas são passivas e atualizam seu estado periodicamente através de *checkpoints* oriundos da réplica privilegiada. Uma réplica passiva nunca irá responder uma requisição, a não ser quando assumir o papel de réplica privilegiada. Em relação a semântica, atua principalmente sobre as faltas de parada.

  Um resumo sobre seu funcionamento pode ser observado em quatro fases (GUERRAOUI; SCHIPER, 1997):

  1. O primário recebe uma requisição (req) de um cliente juntamente com uma identificação única (IDreq);
  2. A resposta é processada (res) e o estado do primário (*state-update*) é atualizado. Em seguida é enviada ao *backup* uma mensagem contendo IDreq.res.state-update;
  3. Uma vez recebida completamente a mensagem, o *backup* atualiza seu estado e confirma ao primário via um reconhecimento (ACK);
  4. Somente após o recebimento do ACK, o primário envia a resposta (res) ao cliente.

  Em relação a uma falta de parada no primário, três situações serão possíves; antes da mensagem com seu estado ser transferida

ao *backup*; após ou durante o envio - porém antes do cliente receber sua resposta; e após o cliente já ter recebido sua resposta. Em todos os casos uma réplica deverá assumir o papel de primário. O terceiro caso é o mais simples, pois futuras requisições serão feitas diretamente para o novo primário. Já no primeiro e segundo casos, o cliente não receberá sua resposta até que reenvie suas requisições ao novo primário (que pode levá-lo a suspeitar de uma falta). A atomicidade é garantida quando todas as replicas de *backup* estão no mesmo estado, ou seja, ou receberam a mensagem completa do primário antes da falta ou não. Quando todas receberam a mensagem completa, para evitar que a requisição anterior seja processada novamente, o novo primário imediatamente responde a (res) já confirmada. A técnica de replicação primário - *backup* pode levar o cliente a erroneamente suspeitar de uma falha no primário, já que suas respostas dependem do tempo da transferência da mensagem de estado para o *backup* (GUERRAOUI; SCHIPER, 1997).

- **replicação semi-ativa**: trata-se de um híbrido entre replicação ativa e passiva. Apenas uma réplica é encarregada de responder as requisições, porém todas as recebem e as processam em paralelo. Não há recuperação em retrocesso, como na replicação passiva.

A frequência de *checkpointing* é um fator importante a ser considerado, uma vez que em caso de falta no primário, o *checkpoint* mais recente salvo no *backup* (ou armazenamento estável) será recuperado para a restauração do sistema. A consistência da réplica pode variar de acordo com a frequência em que os *checkpoints* são salvos. Duas técnicas são abordadas em (AYARI et al., 2008):

- incremental: mais agressiva, visa maximizar a consistência da réplica, salvando um novo *checkpoint* cada vez que ocorrer uma alteração crítica no hospedeiro primário. Sua desvantagem é o alto custo de processamento envolvido no primário;

- periódica: um novo *checkpoint* é salvo a cada intervalo (período) de tempo predeterminado. Intervalos curtos levam a uma alta frequência de *checkpointing*, elevando o custo de processamento no hospedeiro primário. A consistência da réplica representa o número de operações de reversão que deverão ser executadas no caso de impedimento do hospedeiro primário. Uma outra abordagem sugere o uso randômico para determinação dos intervalos,

que objetiva a diminuição da sobrecarga imposta pela replicação por *checkpointing* em relação as aplicações hospedadas.

Ainda, em relação a replicação, é importante salientar o critério de correção da linearizabilidade. Além de preservar a semântica das aplicações hospedadas, também passa para o cliente/usuário a ilusão de estar interagindo com uma máquina não replicada, onde as respostas as suas solicitações serão as mesmas que uma não replicada responderia (GUERRAOUI; SCHIPER, 1997).

2.3 Considerações Parciais

Seja pelo aumento da capacidade dos processadores atuais, por questões relativas a manutenção, consumo de energia ou econômicas, a virtualização de recursos computacionais torna-se cada vez mais comum nos dias de hoje. Enquanto propriedades como isolamento, compatibilidade e performance garantem controle de recursos e segurança para execução de uma MV, o encapsulamento possibilita o uso de técnicas de migração e replicação.

Remus, um sistema que visa tolerar faltas de parada para execução de uma MV, será explicado no próximo capítulo. Baseado na técnica de replicação primário-*backup* e aproveitando-se da propriedade de encapsulamento, Remus (que executa no MMV Xen), é capaz de salvar e replicar o estado inteiro de uma MV, incluindo disco, CPU e memória, através de dezenas, ou até centenas, de *checkpoints* periodicamente. Em relação aos atributos da segurança do funcionamento, pode-se dizer que disponibilidade e confiabilidade sejam os mais relevantes.

3

O MECANISMO DE REPLICAÇÃO REMUS

A virtualização de recursos computacionais permite o encapsulamento de uma MV (incluindo a pilha de aplicativos em execução). Explorando essa oportunidade, o mecanismo de replicação Remus fornece tolerância a faltas de parada às aplicações hospedadas, de forma transparente e com um baixo custo financeiro (CULLY et al., 2008). Além disso, seu mecanismo é focado na utilização de *hardware* de prateleira como hospedeiros das MVs. Este capítulo descreve as particularidades de Remus, apresentando detalhes de sua implementação e discutindo algumas limitações que comprometem o desempenho das aplicações finais.

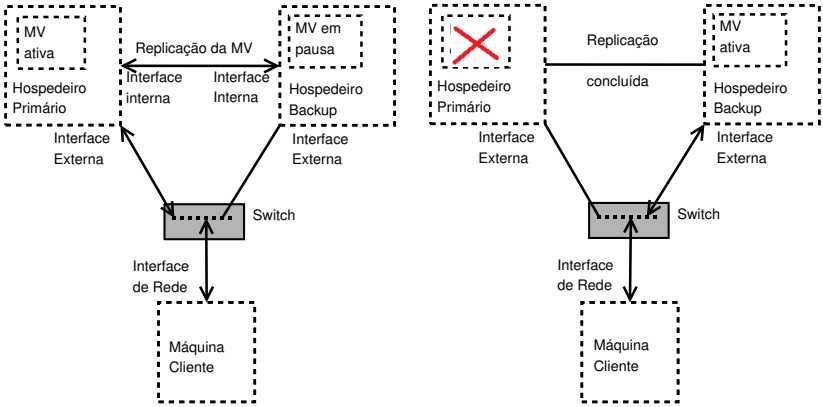
3.1 Funcionamento

Remus é totalmente agnóstico em relação ao sistema operacional em execução na MV, o qual é executado sem precisar de configuração adicional. Seu mecanismo de replicação salva dezenas (ou centenas) de *checkpoints* por segundo, sendo que para cada *checkpoint* é feito um procedimento *stop-and-copy*, semelhante ao utilizado pelo mecanismo de migração de MVs Xen (CLARK et al., 2005), descrito no Capítulo 2.1.3.

3.1.1 Visão Geral

A Figura 6a ilustra o cenário de execução original de Remus entre os hospedeiros primário e *backup*. Cada hospedeiro possui duas interfaces de rede: uma para acesso externo, para interação com os

clientes, e outra exclusiva para o tráfego de replicação. É preciso que os discos da MV estejam sincronizados nos dois hospedeiros antes do processo de replicação ser iniciado. Com os discos sincronizados, uma MV em estado de pausa é criada no hospedeiro *backup*, e uma cópia inteira da memória da MV em execução no primário é então transferida para o *backup*.



(a) Remus em execução - estado livre de faltas.

(b) Estado após uma falta de parada no primário.

Figura 6 – No ambiente de execução de Remus, os hospedeiros são interligados por uma interface exclusiva de rede.

Após esse processo inicial, inicia-se o processo de *checkpointing*. Periodicamente, a cada intervalo predeterminado, a MV ativa sofre uma pausa e salva um novo *checkpoint* em um *buffer* local. Nesta etapa, somente o conteúdo da memória que foi alterado desde o último *checkpoint* é salvo. Após esta fase, a MV retoma sua execução enquanto o *buffer* é transferido para o *backup*. Assim que o *buffer* estiver salvo no *backup*, é enviada uma confirmação para o primário, que pode ou não salvar um novo *checkpoint* imediatamente.

Durante o período que compreende a retomada da execução até a próxima pausa para salvar um novo *checkpoint*, e até que este estado seja completamente transferido, não há garantia de que o que foi ou está sendo processado na MV esteja protegido. Ocorrendo uma eventual falta de parada no primário, os dados processados na MV antes ou durante a transferência de um *checkpoint* serão perdidos, ou seja, o *backup* retoma a execução da MV considerando apenas o último *checkpoint* completamente recebido. Devido a este risco, diz-se que a

execução da MV está constantemente em modo especulativo.

Um *timeout* de um segundo no recebimento de um *checkpoint* é interpretado pelo hospedeiro *backup* como uma falta de parada no primário, levando-o a ativar sua MV, como ilustra a Figura 6b. Nesse caso, um ARP não solicitado é enviado para a rede local com o intuito de informar que o endereço físico referente ao IP que antes pertencia ao hospedeiro primário foi alterado (STEVENS, 1993).

O modelo de faltas implementado por Remus segue as seguintes propriedades:

1. tolerância a falta de parada em um único hospedeiro;

2. hospedeiros primário e *backup*, quando sofrem simultaneamente faltas de parada, mantêm os dados da MV protegida num estado imediatamente anterior à ocorrência;

3. nenhuma resposta às requisições solicitadas será respondida até que o estado da MV em execução esteja completamente salvo no hospedeiro de *backup*.

Além disso, Remus não visa a recuperação de faltas de *software*, ou qualquer outro tipo de falta que não seja a de parada.

3.1.2 Replicação da Memória e CPU

Diferentemente do que ocorre com Xen *Live Migration*, Remus utiliza periodicamente, e não como uma única etapa, o processo *stop-and-copy* (CLARK et al., 2005). Ao migrar uma MV, nesta fase, Xen suspende a MV, transfere ao destino o conteúdo de sua memória ainda não migrado, retoma sua execução no hospedeiro de destino e a destrói no hospedeiro de origem. Remus, por sua vez, periodicamente suspende a MV, salva o conteúdo da memória e o estado da CPU alterados desde o último *stop-and-copy* executado em um *buffer* local, e retoma sua execução no hospedeiro primário enquanto transfere o *buffer* para o *backup*.

Como em Remus a MV retoma sua execução no próprio hospedeiro de origem e não depende do tempo de transferência do *buffer* ao *backup*, o tempo de pausa torna-se menor, beneficiando seu tempo de processamento. Neste cenário, a repetição periódica deste processo é chamada de *checkpointing*.

Algumas modificações foram necessárias para diminuir a latência imposta pelo mecanismo de replicação, uma vez que a maioria do

tempo gasto na fase *stop-and-copy* ocorre devido ao excesso de instruções na comunicação entre a MV e o hipervisor. Um *event channel*, ou interrupção virtual, implementado para hóspedes paravirtualizados, diminui o número de etapas necessárias para suspender e retomar a execução da MV protegida. Além disso, Remus identifica rapidamente as páginas de memória a serem salvas no *buffer*, ignorando as que não foram utilizadas desde o último salvamento. O tempo gasto para o mapeamento total da memória da MV, que ocorre no início de cada *checkpoint*, é aproximadamente o mesmo gasto para salvar o *buffer* local. Para diminuir o tempo gasto com a replicação entre primário e *backup*, Remus previamente aplica uma compressão de dados no *buffer* local (RAJAGOPALAN et al., 2012).

3.1.3 Replicação do Disco

A replicação do disco local da MV é realizada por uma adaptação da ferramenta DRBD – *Distributed Replicated Block Device* (REISNER; ELLENBERG, 2005). A cada *checkpointing* o disco da MV residente no hospedeiro primário é sincronizado com o disco local da MV no *backup*. Um novo protocolo, chamado de D, foi adicionado ao DRBD para suportar a replicação assíncrona por *checkpoint*, uma vez que os protocolos previamente existentes (A, B e C), suportam apenas os modos assíncrono, semi-síncrono e síncrono, respectivamente. Outra vantagem do uso da ferramenta DRBD é a ressincronização automática dos discos. Quando o hospedeiro primário retoma sua execução após uma falta de parada, por exemplo, o disco local da sua MV é sincronizado com o disco da MV que se manteve em execução no *backup* automaticamente. Com os discos idênticos e sincronizados, é possível inverter os papéis de primário e *backup*, e retornar a MV protegida ao hospedeiro de origem sem que haja interrupção dos serviços em execução.

3.2 Questões de Projeto

Duas características importantes presentes no projeto de Remus são linearizabilidade e frequência de *checkpointing*. A linearizabilidade passa a ilusão ao cliente de que o mesmo não está interagindo com um servidor replicado, tanto nos períodos livres, quanto após uma falta. A frequência de *checkpointing* controla o tempo de execução de uma MV antes do salvamento de um novo *checkpoint*. Tanto a linearizabilidade, quanto a frequência de *checkpointing*, influenciam diretamente

na latência da comunicação e no tempo de processamento da MV protegida.

3.2.1 Linearizabilidade

O modelo de replicação utilizado por Remus permite que os clientes das aplicações hospedadas percebam um comportamento consistente do sistema em caso de faltas, passando a ilusão de estarem interagindo com um servidor não replicado. Uma maneira de garantir essa consistência, chamada de linearizabilidade, define que somente será processado no primário o que for previamente salvo no armazenamento estável (GUERRAOUI; SCHIPER, 1997) (ELNOZAHY et al., 2002), no caso o disco e memória do hospedeiro *backup*.

Remus controla apenas o tráfego de saída de rede como forma de garantir a linearizabilidade. A MV recebe e processa o tráfego de entrada de rede sem qualquer restrição, o que, além de reduzir a sobrecarga de processamento, também preserva a semântica das aplicações em execução na MV. Durante o período de execução, o tráfego de rede que sai da MV fica retido em um *buffer* no primário, o chamado *buffer* de rede. Somente após o *backup* confirmar o recebimento do *checkpoint* (ou seja, o *buffer* local ter sido totalmente recebido e o disco sincronizado) é que este tráfego é liberado. O *buffer* de rede se faz necessário para garantir, na visão do cliente, um estado idêntico em relação às conexões de rede caso ocorra uma falta de parada do primário (CULLY et al., 2008). Essa técnica, que leva a uma sobrecarga no desempenho das aplicações protegidas, é classificada como um problema de *output commit* (ELNOZAHY et al., 2002) (ZHU et al., 2010).

A Figura 7 ilustra o funcionamento do *buffer* de rede no primário. Qualquer tráfego de saída de rede originado na MV (TX) é bloqueado até que o *backup* envie uma confirmação de recebimento do último *checkpoint* (a linha vermelha representa este evento). O tráfego de entrada (RX) não sofre influência deste *buffer*, ou seja, a MV está apta a receber dados sempre que estiver em execução.

3.2.2 Frequência de *Checkpointing*

Em Remus, o intervalo entre *checkpoints* é fixo, sendo definido antes do início do processo de replicação. Esse intervalo tem influência direta no tempo de duração do *checkpointing* (TDC), que compreende o tempo para salvar o estado no *buffer* e o tempo de transferência do *checkpoint* para o *backup*, e, devido ao *buffer* de rede, pode prejudicar as aplicações comunicantes, aumentando a latência da comunicação.

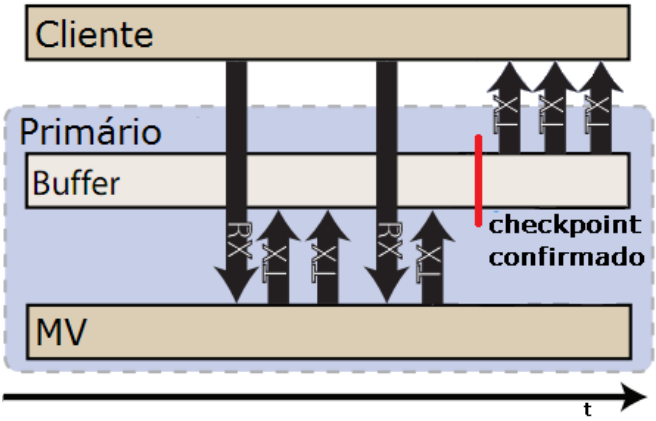


Figura 7 – O tráfego de saída de rede da MV protegida somente será liberado após o estado replicado ser confirmado pelo hospedeiro de *backup*. Enquanto esse evento não ocorre, os pacotes são armazenados em um *buffer* temporário no hospedeiro primário.

Como o TDC depende da carga de trabalho em execução na MV, seu valor é sempre variável. Como será discutido da Seção 3.3, quanto mais alta r a frequência de *checkpointing*, menor a latência na comunicação.

A Figura 8 apresenta o mecanismo de replicação Remus com um intervalo fixo predefinido em 50 ms. Tomando como exemplo o período 3, observa-se que o tempo de execução da MV antes de salvar um novo *checkpoint* foi exatamente de 50 ms. Isso ocorre porque o tempo de transferência do *checkpoint* referente ao período 2 foi inferior ao intervalo estipulado (50 ms). O mecanismo então aguarda o intervalo estipulado ser atingido, garantindo o tempo de execução de 50 ms, e inicia o salvamento de um novo *checkpoint*. Com o estado confirmado no *backup*, um ACK é enviado ao primário e o *buffer* de rede é finalmente liberado. A seta vertical número 3 indica a visão do cliente em relação ao que foi executado na MV durante o período 3. Nota-se, portanto, que o cliente tem uma visão atrasada em relação ao que foi processado na MV.

Ademais, a Figura 8 ilustra o intervalo de *checkpointing* sendo respeitado em todos os períodos, porém, nem sempre é o que ocorre: quando o TDC for maior que o intervalo predeterminado, o intervalo passará a ser o TDC. Esta condição leva a um atraso no salvamento

do próximo *checkpoint*, diminuindo a frequência predeterminada (PETROVIC; SCHIPER, 2012), e aumentando a latência na comunicação. Essa característica pode ser observada na Figura 9, na qual o intervalo determinado de 50 ms foi superado nos períodos 2, 3 e 6. Por exemplo, o período 2 teve seu tempo de execução aumentado para 200 ms porque a transferência do *checkpoint* anterior (período 1) foi exatamente de 200 ms. Quanto maior o tempo de execução da MV (TEMV), maiores as chances que o próximo *checkpointing* também exceda o intervalo de predeterminado, uma vez que a quantidade de dados a serem replicados aumenta. Como pode ser visto na Figura 9, o tempo de execução da MV do período i acompanha o TDC do período (i - 1). Sempre que o intervalo predeterminado for superado, maior será a latência na comunicação, uma vez que o *buffer* de rede depende do ACK de recebimento do *checkpoint* para ser liberado.

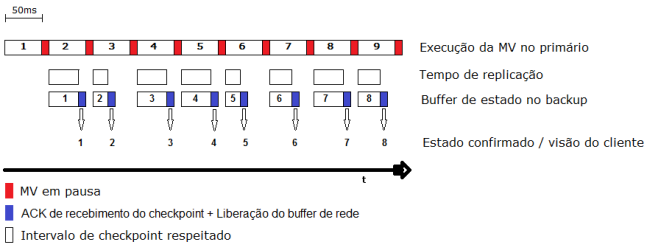


Figura 8 – O processo de replicação do mecanismo Remus.

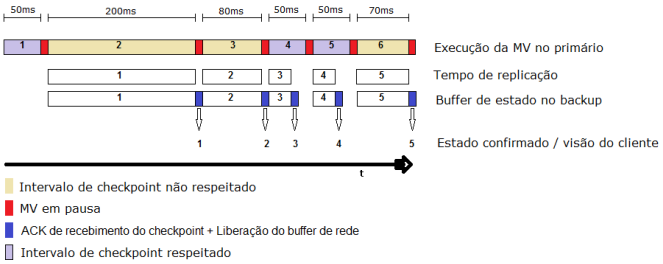


Figura 9 – Em Remus, o tempo de execução em modo especulativo varia de acordo com o tempo de replicação sempre que o intervalo definido for ultrapassado.

Conforme descrito em (PETROVIC; SCHIPER, 2012), dependendo do momento em que um pacote de saída é gerado na MV, a latência na comunicação com o cliente pode ser maior ou menor. Por exemplo, se um pacote a ser transferido para um cliente for gerado próximo à pausa para salvar um novo *checkpoint*, a latência será o somatório dos tempos de pausa e de transferência. Por outro lado, se o pacote for gerado imediatamente após um novo *checkpoint* salvo, ou seja, logo no início de um período de execução em modo especulativo, a latência será o resultado do somatório do tempo de execução até a próxima pausa, pela pausa e pelo tempo de transferência deste *checkpoint*. Este pode ser caracterizado como o pior caso em relação à latência na comunicação percebida pelo cliente. Em suma, é difícil predizer a latência que um pacote de saída pode sofrer, uma vez que a liberação do *buffer* de rede está diretamente conectado com o TDC, que nativamente em Remus é variável: o processo de replicação em sua configuração padrão pode não obedecer a frequência predeterminada para *checkpointing*, como visto na Figura 9.

3.3 Avaliação Experimental da Sobrecarga Imposta pelo Remus

O mecanismo de replicação impõe uma sobrecarga na aplicação hospedada e protegida. Conforme discutido, esta sobrecarga pode ser mais perceptível em aplicações comunicantes, devido à linearizabilidade do sistema. Nesta seção, quantificamos a sobrecarga imposta em diversas aplicações.

Os *benchmarks* executados na análise experimental de Remus quantificaram o impacto causado pelo mecanismo de replicação na execução de aplicações considerando a variação de latência no substrato físico. Neste cenário original, os autores não investigaram a frequência de *checkpointing* apropriada para aplicações comunicantes, ou seja, aquelas sensíveis à latência da comunicação, e sugerem a possível existência de sobrecarga, porém sem quantificar esta afirmação. Assim, realizou-se uma análise da sobrecarga imposta por Remus levando em consideração dois cenários: aplicações sensíveis e não sensíveis à latência imposta pelo uso da rede.

Para todos os testes realizados, a frequência foi definida em 10, 20, 30 ou 40 *checkpoints*/s, seguindo os estudos previamente realizados (CULLY et al., 2008), (RAJAGOPALAN et al., 2012), além de zero (0), que indica a realização do mesmo teste com uma MV não protegida. Para cada experimento foram realizadas trinta rodadas. Os resultados

apresentam a média e o intervalo de confiança de 95% (representado pelas barras de erro).

O ambiente de testes compreendeu três computadores com processador AMD Phenom II X4 (4 cores) de 2,8 GHz, memória RAM de 4 GB, disco 500 GB SATA, executando o sistema operacional Ubuntu 12.10 (kernel 3.5.0-17-generic), e interconectados em uma rede local, conforme mostrado na Figura 6a. O equipamento identificado como "Máquina Cliente" foi utilizado como origem das requisições para as aplicações que utilizam rede, contando com uma interface de 100Mbps. Os hospedeiros primário e *backup* possuem duas interfaces de rede, sendo uma para a conexão com a máquina cliente através de um comutador de 1 Gbps, e outra exclusiva para a replicação da MV (1 Gbps via cabo *crossover*). Cada hospedeiro é virtualizado com o MMV Xen 4.2.1 e protegido pelo mecanismo Remus nativo, executando DRBD versão 8.3.11-remus. A MV a ser replicada é composta por duas VCPUs, 20 GB de disco e 1 GB de memória. Seu sistema operacional é o OpenSUSE 12.2 (x86_64).

O tempo de execução total de cada tarefa foi medido durante os testes e utilizado como métrica para avaliar a sobrecarga imposta à aplicação hospedada na MV durante a execução do Remus. Complementarmente, a vazão de dados, em MB/s, entre os hospedeiros primário e *backup* também foi mensurada. O objetivo de medir esta vazão foi observar se a capacidade deste enlace, exclusivo para a replicação, influencia no tempo de processamento ou na latência das aplicações hospedadas na MV protegida.

Para indicar uma base de comparação, a Figura 10 mostra a média da vazão entre os hospedeiros com a MV ociosa. O tempo de captura foi de 10 segundos para cada intervalo. Os resultados mostram que quanto mais baixa a frequência, menor a vazão média da replicação. Neste caso, a eficácia da compressão do *checkpoint*, implementada por (CULLY et al., 2008), tende a melhorar conforme aumenta o volume dados a ser replicado.

3.3.1 Aplicações sem Uso de Rede

As aplicações sem o uso de rede não têm seu desempenho afetado pela latência da comunicação imposta pelo *buffer* de rede. Neste cenário, a sobrecarga imposta às aplicações hospedadas ocorre somente pela necessidade de parar e retomar a execução da MV periodicamente para salvar um novo *checkpoint*.

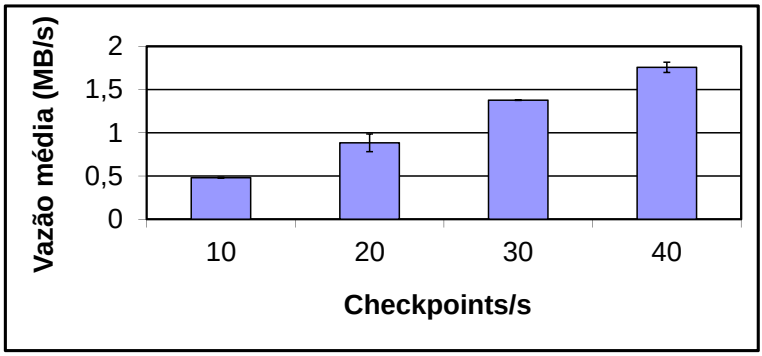
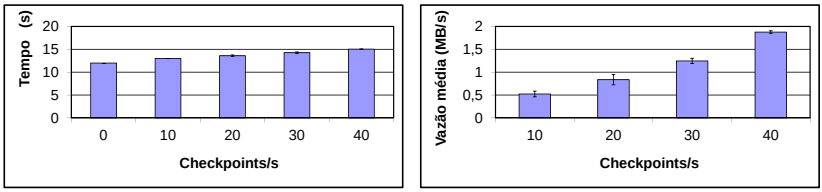


Figura 10 – Média da vazão entre os hospedeiros primário e *backup* com a MV ociosa.

3.3.1.1 Aplicação com Uso Intensivo de CPU

Para investigar a sobrecarga do mecanismo de replicação em aplicações CPU-*bound*, utilizamos o *benchmark sysbench* [1], especificamente o módulo que calcula os números primos possíveis menores do que 10000. O tempo de execução pode ser comparado à execução sem Remus (identificado pela legenda 0).



(a) Tempo de execução. (b) Vazão média de replicação.

Figura 11 – Tempo de execução e vazão média durante o *benchmark* de CPU.

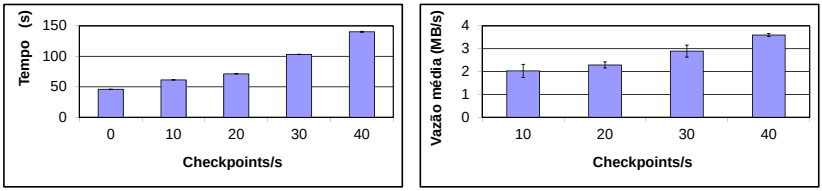
Como mostra a Figura 11a, a sobrecarga imposta por Remus variou pouco em relação à frequência de replicação. Enquanto sem replicação o tempo foi de 12 s, com 10 *checkpoints*/s foi de 13 s, resultando em 8% de sobrecarga no desempenho da MV. No pior caso, com 40 *checkpoints*/s, a sobrecarga foi de 25%. Ou seja, conforme já observado por (CULLY et al., 2008), uma maior frequência na replicação

[1] Disponível em <https://launchpad.net/sysbench>

induz uma maior sobrecarga no desempenho de aplicações CPU-*bound*. A Figura 11b ilustra a média da vazão de dados entre os hospedeiros primário e *backup* durante a execução do teste. Assim como ocorre quando a MV está ociosa, percebe-se que quanto menor a frequência, menor o fluxo de dados transferido.

3.3.1.2 Aplicações com Uso Intensivo de Memória

Este cenário avalia o uso intensivo de operações na memória principal. Uma aplicação foi desenvolvida para alocar 200.000 páginas de 4 KB, totalizando o uso de 781,25 MB na memória da MV, e escrever dados aleatórios a uma frequência de 10000 páginas por segundo. Quanto maior o tempo para a alocação total, maior a sobrecarga imposta pela frequência da replicação. Os resultados são comparados com uma execução sem Remus (barra rotulada com 0). Como mostra a Figura 12a, a sobrecarga causada por Remus, em seu melhor resultado (com 10 *checkpoints*/s), foi em torno de 32% superior em relação à execução em uma MV desprotegida. Para 40 *checkpoints/s*, esta sobrecarga é superior a 200%. Novamente, observa-se que, quanto mais baixa a frequência de *checkpoints*, menor a sobrecarga, e menor o fluxo de dados entre os hospedeiros, como mostra a Figura 12b.



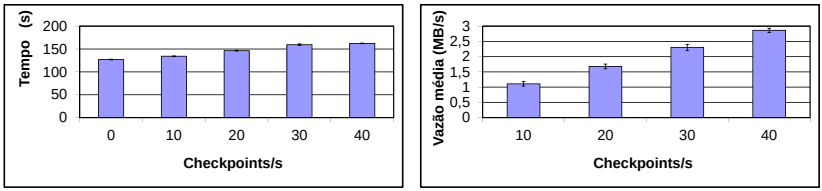
(a) Tempo de execução. (b) Vazão média de replicação.

Figura 12 – Tempo de execução e vazão média durante o *benchmark* de memória.

A sobrecarga é justificada pela natureza do mecanismo de replicação, que transfere, constantemente, todas páginas de memória alteradas desde o último *checkpoint* confirmado pelo *backup*. Embora o volume de memória modificado pela aplicação varie entre 250 e 1000 páginas (1 a 4 MB) por *checkpoint*, a vazão média entre os hospedeiros foi insignificante perante a capacidade do enlace.

3.3.1.3 Aplicação com Uso Intensivo de E/S

O teste executado realiza operações de leitura e escrita (volume de dados total de 3 GB), aleatoriamente, no disco local da MV, sincronizado pelo mecanismo DRBD. Assim como nos testes anteriores, quanto mais baixa a frequência do *checkpointing*, melhor o desempenho da MV (Figura 13a) e menor o fluxo de dados transmitido entre os hospedeiros primário e *backup*, como mostra a Figura 13b. O pior caso identificado foi com 40 *checkpoints*/s, sendo 27% a sobrecarga observada. A pequena variação observada entre 30 e 40 *checkpoints*/s segue o comportamento observado em alguns experimentos originais do Remus. Se comparado com o valor base (sem Remus), observa-se que a sobrecarga de operações de E/S é tolerável para este tipo de aplicação. A vazão de dados entre os hospedeiros foi medida para cada frequência (Figura 13b) e segue o padrão dos demais experimentos.



(a) Tempo de execução. (b) Vazão média de replicação.

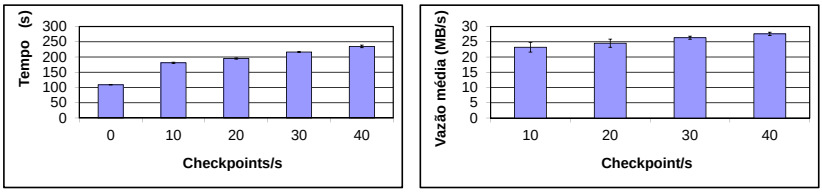
Figura 13 – Tempo de execução e vazão média durante o *benchmark* de E/S.

3.3.1.4 Compilação

Neste experimento foi efetuada a compilação do BIND[2] versão 9.9.4.p2 em uma MV protegida. Este processo envolve uso de CPU, memória e acesso a disco (360 novos arquivos são criados). A Figura 14a apresenta o tempo total de execução do teste.

A sobrecarga imposta pela replicação foi de 66% para o melhor caso: 10 *checkpoints*/s. Observa-se que, quanto maior o intervalo (mais baixa a frequência), menor a sobrecarga imposta. Isto se deve ao fato de Remus permitir a execução das aplicações hospedadas em modo especulativo, ou seja, seu tempo de processamento não depende da confirmação de recebimento pelo *backup* do *checkpoint* anterior. Especificamente para este cenário combinado, observa-se que a vazão média

[2] <http://www.isc.org/software/bind>



(a) Tempo de execução. (b) Vazão média de replicação.

Figura 14 – Tempo de execução e vazão média durante a compilação do BIND.

entre primário e *backup* foi maior em relação aos outros testes, como mostra a Figura 14b. Isso se deve pela diversificação dos dados a serem replicados durante a compilação, que acaba por diminuir a taxa de compressão do *checkpoint*. No pior caso, com 40 *checkpoints/s*, a média da vazão ficou abaixo de 30 MB/s, mostrando novamente que a capacidade máxima do enlace não foi atingida.

3.3.2 Aplicações com Uso de Rede

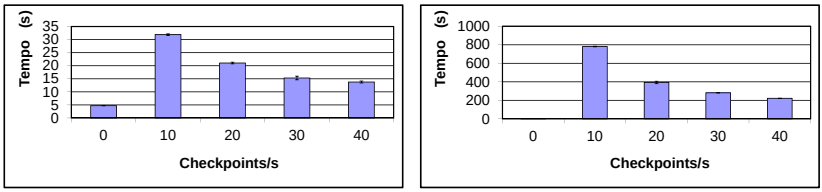
As aplicações com uso de rede sofrem um impacto causado pela atuação do *buffer*. Este cenário de testes avalia a sobrecarga imposta às aplicações comunicantes sob diferentes frequências de *checkpoints*/s. Para avaliar este impacto serão utilizadas aplicações baseadas no TCP, uma vez que este protocolo é confiável, capaz de se recuperar de eventuais perdas induzidas por indisponibilidade (*downtime*) da MV.

3.3.2.1 Transferência de Dados

Este experimento avalia a latência percebida por um cliente conectado a uma MV protegida por Remus. Através da ferramenta SCP (transferência sobre SSH)[3], mediu-se o tempo necessário para cada transferência de 50 MB entre o cliente e a MV, sob diferentes frequências. A Figura 15a ilustra os resultados do teste de *upload*, onde a transferência do arquivo parte do cliente para a MV. A Figura 15b representa o teste de *download*, onde a transferência do arquivo parte da MV para o cliente.

Ao contrário do que mostram os testes sem o uso da rede, a sobrecarga maior sobre as aplicações hospedadas se dá com uma baixa

[3] <https://en.opensuse.org/SDB:SCP_usage>



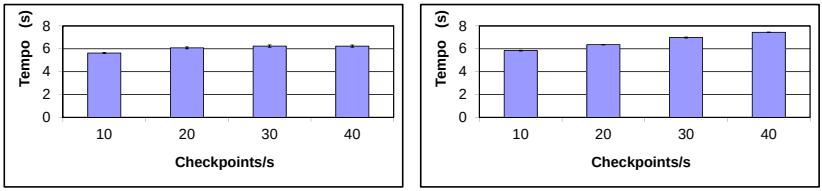
(a) *Upload* - transferência de um arquivo de 50MB do cliente para a MV.

(b) *Download* - transferência de um arquivo de 50MB da MV para o cliente.

Figura 15 – Tempo de transferência para *upload* e *download*

frequência de *checkpointing*. A Figura 15a mostra que no melhor resultado, salvando 40 *checkpoints*/s, a sobrecarga é de 218% em relação à mesma transferência para uma MV sem proteção. No segundo caso, onde a transferência parte da MV (Figura 15b), a menor sobrecarga é de aproximadamente 4900% (4,4 segundos sem Remus, contra 222 segundos com 40 *checkpoints*/s. No pior caso, 10 *checkpoints*/s, a sobrecarga está próxima a 18.000%. Quanto mais alta a frequência, menor a latência na comunicação percebida pelo cliente, contudo, mesmo no melhor caso, o impacto sobre a latência é acentuado, podendo inviabilizar o uso de Remus.

A diferença da latência na comunicação percebida entre *upload* e *download* pode ser explicada pela atuação do *buffer* de rede. A MV, mesmo em modo especulativo, nunca deixa de receber requisições. O que ocorre é um bloqueio das respostas destas requisições até que o *checkpoint* esteja totalmente salvo no *backup*. Não há transmissão de dados partindo da MV enquanto o hospedeiro *backup* não receber por completo o *checkpoint* em questão.



(a) *Upload* - transferência de um arquivo de 50MB do cliente para a MV.

(b) *Download* - transferência de um arquivo de 50MB da MV para o cliente.

Figura 16 – Tempo de transferência para *upload* e *download* sem o *buffer* de rede ativo.

Para fins de análise de desempenho, Remus permite a execução sem o *buffer* de rede ativo (o que fere a linearizabilidade em caso de faltas, conforme discutido na Seção 3.2.1). As Figuras 16a e 16b mostram o resultado para transferências de 50 MB entre o cliente e a MV, tanto para *upload* quanto para *download*, respectivamente, sem a utilização do *buffer* de rede. Enquanto com *buffer* de rede ativo o tempo para transferência de um arquivo de 50 MB para a MV foi de 13,7 segundos, sem o *buffer* foi de 6,2 segundos com 40 *checkpoints*/s.

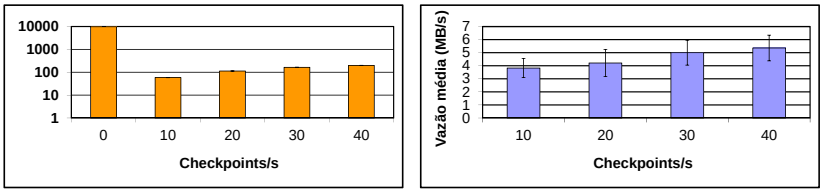
No *download*, também a 40 *checkpoints*/s, o tempo necessário foi 222 segundos, contra 7,4 segundos sem o *buffer* ativo. Neste caso, pode-se afirmar que o *buffer* de rede foi responsável em aumentar a latência percebida pelo cliente em 3.000%. Outra característica dos testes sem o *buffer* ativo é que o melhor desempenho na execução das MVs se dá com frequências menores, a exemplo do que ocorre com aplicações que não utilizam a rede. Há de se ressaltar que esta é uma opção apenas para *benchmark*, já que o *buffer* de rede é essencial para que o estado das MVs esteja sempre igual, na visão do cliente, na ocorrência de uma falta de parada no hospedeiro primário.

3.3.2.2 *Benchmark* Ab

O *benchmark* Ab[4] permite a quantificação do número de requisições por segundo suportadas por um servidor Apache. Nesse cenário, avaliou-se o desempenho de um servidor Apache versão 2.2.22 (com a configuração padrão do OpenSUSE) em execução na MV protegida, atendendo a 100 conexões simultâneas durante 60 segundos. O arquivo a ser lido constantemente pelos clientes compreende uma página de 72950 bytes (a saber, phpinfo.php).

A Figura 17a mostra o total de requisições atendidas. No pior caso, com 10 *checkpoints*/s, o servidor web respondeu a apenas 59 requisições durante o período. No melhor caso, com 40 *checkpoint*/s, 199 requisições foram atendidas. Porém, se comparado ao teste sem Remus em execução, onde o servidor respondeu a 9770 requisições durante os mesmos 60 segundos, houve uma sobrecarga no desempenho superior a 4.800%. A média da vazão de replicação entre o primário e o *backup* (Figura 17b) seguiu o padrão dos testes anteriores. Novamente, observa-se que esta vazão não foi responsável pela sobrecarga imposta às aplicações hospedadas.

[4] Disponível em <http://httpd.apache.org/docs/2.2/programs/ab.html>



(a) Requisições atendidas. (b) Vazão média de replicação.

Figura 17 – Número de requisições atendidas em 60 segundos e vazão média entre os hospedeiros durante a execução do *benchmark* Ab.

3.3.2.3 *Benchmark* tbench

Este experimento analisa o comportamento de um servidor de arquivos em uma MV protegida por Remus. Para tal, o *benchmark* tbench[5] simula as operações comumente executadas em um servidor de arquivos SMB (*Server Message Block*)[6] durante 120 segundos. Este *benchmark* simula somente a percepção do usuário, ou seja, somente as mensagens enviadas, sem executar operações de leitura e escrita em disco. Neste teste foram utilizadas as configurações padrões sugeridas na documentação do tbench.

A Tabela 1 apresenta o número de operações para cada operação executada, enquanto a Tabela 2 mostra a latência média percebida pelo cliente. Assim como no *benchmark* Ab, a latência da comunicação aumenta conforme aumenta o intervalo entre os *checkpoints*, como pode ser observado na Figura 18, que apresenta o gráfico da latência para cada operação.

3.3.3 Discussão dos Resultados

A análise dos resultados da Seção 3.3 mostra que o mecanismo de replicação implementado por Remus induz uma sobrecarga não desprezível ao desempenho das aplicações hospedadas nas MVs. Aplicações com uso intensivo de operações em memória tendem a sofrer um maior impacto devido à natureza do mecanismo de replicação, já que todo conteúdo alterado desde o *checkpoint* anterior deve ser transferido ao *backup*. Ainda, quanto mais alta a frequência com que a MV é interrompida para que novos *checkpoints* sejam salvos, maior a sobrecarga

[5] Disponível em <https://dbench.samba.org>
[6] Disponível em <https://www.samba.org/cifs/docs/what-is-smb.html>

Operação	Sem Remus	40 cp/s	30 cp/s	20 cp/s	10 cp/s
Close	35836	1330	1447	1386	956
Rename	2092	99	97	95	22
Unlink	9916	169	166	161	58
Find	17142	420	413	404	162
Write	24168	2751	3333	3140	3356
Read	77339	1819	1575	1425	618
Flush	3437	239	239	238	186

Tabela 1 – Número de operações executadas em um servidor SMB durante 120 segundos.

Operação	Sem Remus	40 cp/s	30 cp/s	20 cp/s	10 cp/s
Close	2,13	29	37,8	55,17	110,8
Rename	2,96	27,4	34,3	52	101,7
Unlink	2,16	26,9	34,8	52,4	101,6
Find	3	36,1	41	61,4	103
Write	5	77	110,6	106,3	186,1
Read	6	459,2	394	398,7	332,7
Flush	3	27,2	34,3	52,3	102,3

Tabela 2 – Latência média (em milissegundos) percebida pelo cliente para cada operação durante 120 segundos.

identificada nas aplicações exclusivamente dependentes de tempo de processamento e de operações de E/S. Em geral, para aplicações sem comunicação, observou-se que é preferível uma frequência mais baixa da replicação, porém o oposto pode ser dito em relação a aplicações sensíveis a latência (ou que envolvem transferência contínua de dados entre cliente e MV).

Para as aplicações comunicantes, o fato da MV estar constantemente em modo de execução especulativo (agravado pela influência do *buffer* de rede) resulta no aumento da latência na comunicação com o cliente. Em suma, quanto mais alta a frequência de *checkpoints*/s, menor o tamanho do *checkpoint* a ser transferido e mais rápida a liberação do *buffer* de rede.

Independente do intervalo utilizado para reger a frequência de *checkpoint*, Remus sempre prioriza o tempo de processamento em detrimento da latência na comunicação. O modo especulativo permite que

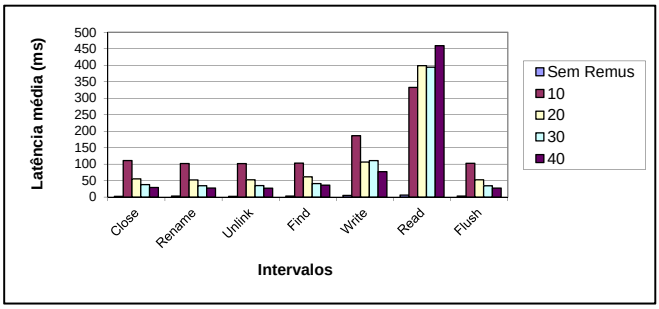


Figura 18 – A latência média percebida pelo cliente para cada operação de rede em um servidor de arquivos durante 120 segundos.

o tempo de transferência do *checkpoint* para o hospedeiro *backup* não prejudique o tempo de processamento das aplicações hospedadas na MV. Contudo, o tráfego de saída de rede é retido no *buffer* de rede até que o *checkpoint* em questão esteja completamente salvo no hospedeiro *backup*. Essas característica, necessária para a garantia da linearizabilidade, colabora com o aumento da latência na comunicação entre o cliente e a MV.

A vazão média observada para a transferência dos *checkpoints* entre os hospedeiros primário e *backup* mostrou-se irrelevante em relação à capacidade total do enlace (1 Gb/s) exclusivo para esta função. A compilação, por exemplo, apresentou a maior média, aproximadamente 25 MB/s para os 4 intervalos testados. Em ambientes geograficamente distribuídos, esta vazão pode ser um fator limitante (com os hospedeiros primário e *backup* em *data centers* distintos). Nesses casos, os enlaces que interconectam os *data centers* apresentam uma menor largura de banda, além de uma latência maior, em relação a uma rede local.

3.4 Considerações Parciais

Nativamente, Remus utiliza intervalos curtos, na grandeza de milissegundos, para salvar e replicar *checkpoints* de uma MV em um hospedeiro de *backup*. Essa característica permite que qualquer aplicação hospedada, inclusive o sistema operacional da MV, tornem-se tolerantes a uma falta de parada.

A análise experimental revela o antagonismo existente entre tempo de processamento e latência de comunicação com relação à frequência de *checkpointing* adotada. O fato da frequência ser regida por um

intervalo predeterminado dificulta o problema. Uma MV pode hospedar uma ou mais aplicações, simultaneamente, com comportamentos distintos em relação ao uso de recursos, como uso intensivo de memória ou rede. Ou então, uma única aplicação pode ser sensível ao tempo de processamento e à latência de comunicação em momentos distintos durante sua execução. Neste caso, nem sempre é possível saber qual dos recursos é mais relevante.

No próximo capítulo é apresentada uma proposta para adaptação dinâmica do intervalo para *checkpointing* em Remus. Baseado na existência de fluxo de saída de rede gerado pela MV, a frequência é dinamicamente adaptada, dispensando a necessidade de um intervalo mínimo predeterminado, e acelerando a liberação do *buffer* de rede. Em contrapartida, quando não há necessidade de priorizar a diminuição da latência, o intervalo é adaptado para diminuir a frequência da replicação, beneficiando o tempo de processamento na MV.

4

Adaptive Remus: Replicação com *Checkpointing* Adaptável

Este capítulo detalha as modificações realizadas no mecanismo de replicação de MVs Remus para introdução do suporte a *checkpointing* adaptativo, chamado *Adaptive Remus*. O mecanismo alterna entre dois modos de operação (rede e processamento), sendo que a alternância é guiada pelo fluxo de rede de saída da MV. Por padrão, a solução opera no modo processamento, alternando somente quando a aplicação hospedada estiver comunicando.

Por fim, são revisados os trabalhos relacionados com a proposta, inclusive aqueles voltados ao uso de *checkpoint* em ambientes não virtualizados.

4.1 Visão Geral do Mecanismo

Enquanto um tempo de execução maior em modo especulativo beneficia o processamento das aplicações hospedadas na MV protegida, já que intervalos maiores entre as pausas resultam em menos interrupções, por outro lado, as aplicações comunicantes são prejudicadas pela existência do *buffer* de rede. Esse antagonismo dificulta a escolha de um intervalo adequado, principalmente quando a MV reúne uma carga mista de uso de recursos, como uso variável de CPU, memória, E/S e rede. Baseado no fluxo de saída de rede da MV, e visando priorizar a diminuição da latência na comunicação, *Adaptive Remus* apresenta uma proposta para adaptação dinâmica do intervalo que rege a frequência de *checkpointing* em Remus.

Através do tempo de duração de um *checkpointing* (TDC), a carga de processamento da MV pode ser quantificada. Por exemplo, um *checkpoint* que dure 80 ms para ser salvo e transferido para o *backup* indica que a MV está sob uma carga maior em comparação a outro de apenas 30 ms. Assim como TDC, o fluxo de saída de rede (FSR) da MV também pode ser quantificado, observando o número de bytes gerados em sua interface virtual a cada *checkpointing*. Com estas variáveis é possível identificar o comportamento temporário da aplicação hospedada, e assim adaptar a frequência de *checkpointing* ideal para cada situação.

4.2 Modos de Operação

A proposta deste trabalho visa adaptar dinamicamente o intervalo da replicação em função da carga de comunicação na qual a MV está submetida. Para isso, dois modos de operação são propostos, *Rede* e de *Processamento*:

- **Modo Rede**: aumenta a frequência de *checkpointing* sempre que for detectado tráfego de saída de rede na interface da MV, visando a diminuição da latência na comunicação com o cliente.
- **Modo Processamento**: quando não houver fluxo de saída de rede (FSR) na interface da MV, o mecanismo deve diminuir a frequência de *checkpointing*, aumentando o tempo de execução da MV (TEMV) para priorizar seu tempo de processamento.

Modificações como compressão de *checkpoint*, mapeamento das áreas de escrita da memória virtual e replicação de disco não são abordadas como parte da proposta, uma vez que Remus já possui seus próprios mecanismos para reduzir este tipo de sobrecarga durante a replicação. A proposta prioriza a diminuição da latência na comunicação da MV, quando houver necessidade, adaptando dinamicamente a frequência de *checkpointing*. Na ausência de FSR, o tempo de execução da MV será sempre priorizado, ou seja, o mecanismo irá operar no modo de processamento, sob uma baixa frequência.

4.3 Alternância entre os Modos de Operação

Uma aplicação comunicante requer uma alta frequência de *checkpointing* para acelerar a saída dos pacotes do *buffer* de rede. Como o oposto, uma baixa frequência, é desejada àquelas aplicações sensíveis ao

tempo de processamento. A proposta guiada pelo fluxo de saída de rede atende as duas condições, porém a latência na comunicação é sempre prioritária.

O mecanismo é capaz de detectar o tráfego de rede originado pela MV monitorando sua interface no hospedeiro primário. O FSR é contabilizado pela coleta do número de bytes de saída de rede na interface de comunicação da MV, e ocorre ao final de cada *checkpointing*. Quando o FSR for maior que o contabilizado no *checkpoint* anterior, Remus tem seu intervalo adaptado para o modo rede, e passa a priorizar a diminuição da latência envolvida entre o cliente da aplicação hospedada e a MV, mantendo-se neste estado por um número predeterminado de rodadas (NCR - número de *checkpoints* em modo rede), quando então, uma nova verificação do FSR é realizada.

Não é uma tarefa trivial determinar o tempo em que o modo rede permanecerá em execução, uma vez que a verificação constante de bytes na interface da MV incorre uma sobrecarga, aumentando o TEMV, atrasando o salvamento de um novo *checkpoint* e consequentemente a liberação do *buffer* de rede.

Desta maneira, a verificação de bytes ocorre frequentemente no estado padrão, ou seja, quando o mecanismo estiver executando em modo de processamento. No modo rede, não há um tempo mínimo estipulado para que um novo *checkpoint* seja salvo, ou seja, um novo *checkpoint* será salvo imediatamente após o anterior estar confirmado no *backup*. Neste modo, o TEMV passa a ser exatamente o tempo de transmissão do *checkpoint* entre o primário e o *backup*. Esta condição acelera a liberação do *buffer* de rede ao máximo possível, diminuindo a latência na comunicação entre cliente e a MV protegida.

O intervalo utilizado no modo de processamento deve priorizar o TEMV, fazendo com que a execução das aplicações hospedadas sofram menos interrupções para o salvamento de novos *checkpoints*. Para este modo, o intervalo deve diminuir a frequência de *checkpointing*. Contudo, dependendo da carga de processamento na MV, frequências muito baixas podem sobrecarregar o tempo gasto para identificar os dados que devem ser salvos no *buffer* local, podendo levar o *backup* a interpretar erroneamente uma falta de parada no primário. Como será visto no próximo capítulo, utilizamos um intervalo de 100 ms, ou 10 *checkpoints*/s, em nossos testes, para o modo de processamento. Apesar de Remus utilizar um intervalo padrão de 200 ms, 100 ms foi o valor utilizado como frequência mais baixa em seus testes (CULLY et al., 2008).

Ressalta-se que, para os dois modos de operação, sempre que

o TDC superar o intervalo utilizado no modo de processamento, o mecanismo perderá sua capacidade de adaptação. Nesse caso, o intervalo para reger a frequência de *checkpointing* passa a ser o próprio TDC. Por exemplo, um TDC de 120 ms, mesmo quando o intervalo predeterminado para o modo de processamento é de 100 ms, fará com que um novo *checkpoint* seja salvo somente com 120 ms. Trata-se do modo padrão de operação do Remus, que indiretamente prioriza o TEMV em detrimento da latência na comunicação.

O fluxograma da Figura 19 ilustra o funcionamento do algoritmo para *checkpointing* adaptável implementado. A adaptação do intervalo ocorre sempre com base em $FSR_{(t-1)}$ que verifica a existência ou não de FSR para determinar o modo de operação do mecanismo. A Tabela 3 mostra as variáveis envolvidas no fluxograma:

BF	tempo do intervalo usado no modo processamento
NCR	Número de *checkpoints* em modo rede
x	contador utilizado para informar que NCR foi atingido
t	contabiliza o número de *checkpoints* efetuados
FSR	fluxo de saída de rede
TR_{ini}	armazena o tempo de início do TDC
TR_{fim}	armazena o tempo de conclusão do TDC

Tabela 3 – Variáveis utilizadas no fluxograma da Figura 19.

- Modo processamento: neste modo, ativado quando $FSR_{(t-1)} == 0$, o algoritmo comporta-se de maneira semelhante ao Remus original. O valor de *BF* (baixa frequência), que determina o intervalo de *checkpointing* a ser usado, busca garantir um tempo mínimo para a execução da MV e assim priorizar seu tempo de processamento. O TDC, que representa o tempo total de duração de um *checkpoint*, TDC = ($TR_{fim} - TR_{ini}$), indica se o TEMV mínimo foi atingido ou não. Quando for maior que *BF*, indica-se o salvamento de um novo *checkpoint* imediatamente, uma vez que o intervalo predeterminado já foi superado. Caso contrário, a MV deve permanecer em execução especulativa até que *BF* seja atingido (aguarda($BF - TDC$)).

  A verificação de bytes de saída de rede da MV ocorre frequentemente, a cada *checkpointing*, no modo de processamento (contabiliza $FSR_{(t)}$). Nesta etapa, o FSR do *checkpointing* atual é comparado ao anterior. Sua diferença indica se há ou não necessidade de adaptação para o modo rede.

- Modo rede: o mecanismo opera neste modo sempre que $FSR_{(t-1)}$ for maior que zero. Para evitar a sobrecarga e o atraso do salvamento de um novo *checkpoint*, uma vez que neste modo o objetivo é aumentar a frequência de *checkpointing* para acelerar a liberação do *buffer* de rede, a verificação do FSR da MV ocorre somente após um número mínimo de *checkpoints* ser atingido (x < NCR). O valor para o número de *checkpoints* em modo rede (NCR), assim como BF para o modo processamento, deve ser previamente estipulado.

  Para voltar ao modo de processamento, é necessário que o valor de NCR seja atingido (x == NCR). Quando x, que atua como um simples contador, for igual ou maior que NCR, uma nova coleta de bytes é executada na interface da MV, ainda em modo rede. Dependendo deste valor, o mecanismo pode manter-se no modo rede ou alternar para o modo processamento.

No modelo de *checkpointing* adaptativo proposto, a variável NCR tem papel fundamental para a diminuição da latência na comunicação entre o cliente e a MV. Determinar seu valor, uma vez que Remus não tem conhecimento algum sobre as características das aplicações em execução, torna-se uma tarefa de difícil solução. Um valor muito alto para NCR, pode prejudicar o tempo de processamento da MV quando uma transferência originada pela MV terminar logo nas primeiras rodadas. Porém, com valores muito baixos, a latência na comunicação seria prejudicada, uma vez que a verificação de bytes na interface da MV aumenta o TDC, atrasando a liberação do *buffer* de rede.

Para relaxar tal condição, duas rodadas adicionais ao valor de NCR foram adicionadas ao modo rede. Nesta fase, ainda no modo rede, o mecanismo executa apenas a checagem de bytes na interface da MV (são necessárias duas rodadas para que o mecanismo determine o FSR), sem utilizar o intervalo predeterminado pela variável BF. Essa condição evita que a replicação retorne ao modo de processamento para tomar uma decisão, prejudicando a latência de rede quando uma transferência ainda estiver em execução.

4.4 Implementação do Mecanismo

Remus é implementado em duas linguagens de programação, C e Python. Enquanto em C estão as funções responsáveis pelo gerenciamento do *checkpoint*, como pausa e retomada da MV, em Python estão as APIs de alto nível, que permitem o acionamento do mecanismo. A

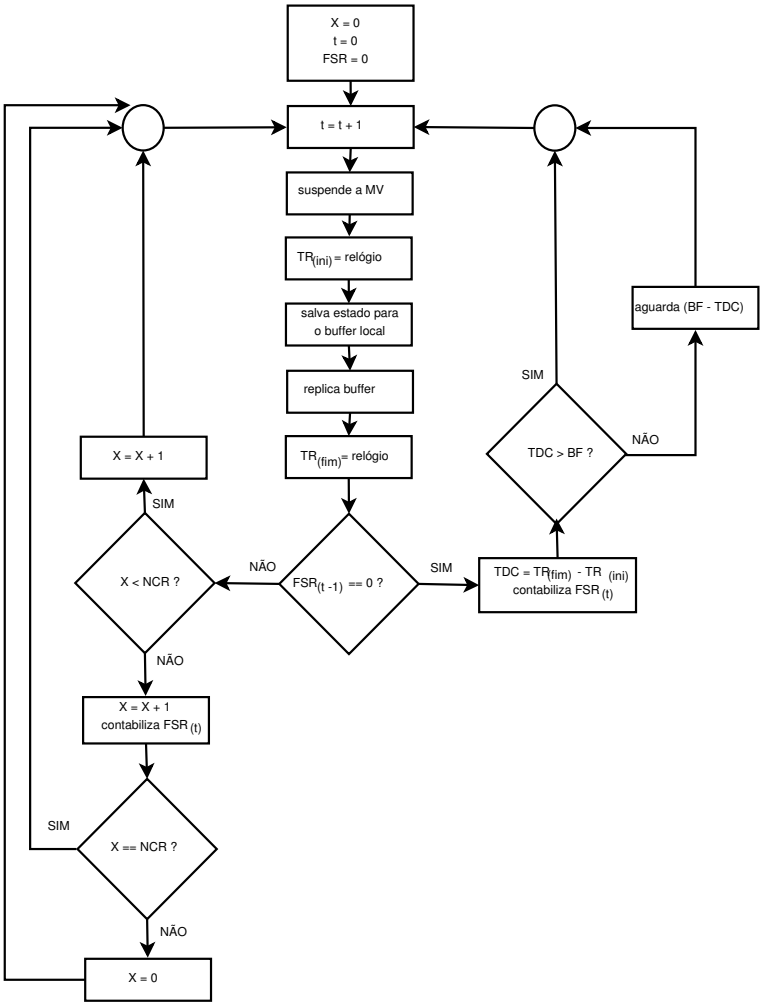


Figura 19 – Fluxograma do mecanismo proposto para *checkpointing* adaptável em Remus.

integração entre as funcionalidades implementadas nessas linguagens ocorre através de uma API específica para utilização de código C em Python [1].

O arquivo de execução do Remus, implementado em Python,

[1] https://docs.python.org/2.7//c-api/index.html

integra os códigos escritos em C, permitindo que estes sejam tratados como módulos de extensão, ou bibliotecas, em seu código original.

Originalmente, a chamada para início do mecanismo de replicação recebe o nome da MV replicada, o endereço do hospedeiro de *backup* e o intervalo de *checkpointing* desejado. Para a versão *Adaptive Remus*, algumas alterações na API foram realizadas. Além do nome da MV, intervalo e endereço do hospedeiro de *backup*, é também necessário informar o valor da variável NCR utilizada no modo rede. Para o modo processamento, nenhuma alteração é requerida, uma vez que o intervalo estipulado é diretamente atribuído para a variável BF.

Quanto as versões de *software*, a solução proposta foi implementada sobre as versões 2.7.3 do Python e 4.2.1 do Xen/Remus. As alterações para adaptação do intervalo ocorreram apenas no código Python, especificamente em seu arquivo de execução. Aproximadamente 25 linhas de código foram inseridas ao código original. Trata-se de uma sequência de instruções, otimizadas para diminuir a sobrecarga computacional, que se repete a cada *checkpointing*. Uma nova execução dessas instruções ocorre somente após o *checkpoint* anterior estar confirmado no hospedeiro *backup*. Cada instrução resulta numa pequena sobrecarga na CPU do hospedeiro primário, principalmente devido ao fato de Remus realizar dezenas, ou até centenas, de *checkpoints* por segundo. A análise experimental será discutida do 5.

A decisão da adaptação entre rede e processamento ocorre imediatamente após o *checkpoint* estar devidamente salvo no hospedeiro *backup*. Para a coleta de bytes na interface de rede da MV, que determina o FSR, novas instruções foram inseridas ao código Python de Remus. Trata-se de uma leitura na interface de rede da MV protegida existente no hospedeiro primário.

Remus, em sua versão original, requer que o intervalo determinado para a frequência de *checkpointing* seja fixo e informado uma única vez no início de sua execução. *Adaptive Remus* alterou esta condição implementando uma estrutura de seleção após o salvamento de cada *checkpoint* no *backup*. Baseado na leitura do FSR, o mecanismo pode ou não determinar o momento exato para o término e início de uma nova rodada.

4.5 Trabalhos Relacionados

Os trabalhos relacionados com a proposta estão organizados em duas categorias: na primeira estão referenciados trabalhos sobre o uso de *checkpointing* adaptativo em ambientes não virtualizados. Na

segunda categoria são abordados trabalhos relativos a migração e replicação de MV.

4.5.1 Ambientes não Virtualizados

Embora inovador em Remus, a adaptação do intervalo de *checkpointing* já foi explorado na literatura especializada. Por exemplo, (ZIV; BRUCK, 1997) busca minimizar a sobrecarga de *checkpointing* de um programa aumentando a frequência quando o estado a ser salvo é pequeno e reduzindo-a quando o custo de *checkpointing* aumenta. (ZHANG; CHAKRABARTY, 2003) adapta a frequência de *checkpointing* em sistemas embarcados de tempo real brando, minimizando o consumo de energia. A proposta tolera um número fixo de faltas para uma dada tarefa. Em comum, esses trabalhos têm como foco controlar o tempo de execução limitando o tempo de recuperação após uma falha, sem se preocupar diretamente com a responsividade durante seu estado livre de faltas.

(EGWUTUOHA et al., 2013) apresenta uma pesquisa sobre tolerância a faltas e implementação de *checkpointing* para sistemas de computação de alto desempenho. O trabalho descrito em (CHTEPEN et al., 2009) propõe o uso de *checkpointing* adaptativo em grades computacionais, ajustando o intervalo entre *checkpoints* de acordo com duas variáveis: o tempo estimado de execução de uma tarefa e a frequência de falhas de recursos. O objetivo é balancear a sobrecarga de *checkpointing* e diminuir o custo de reiniciar uma tarefa interrompida. Dentre as técnicas abordadas para diminuir a sobrecarga causada pelo processo em seu período livre de faltas, duas são semelhantes às utilizadas no Remus: o *checkpointing* incremental e a compressão de dados (PLANK et al., 1995) (PLANK; LI, 1994). Contudo, o método incremental, pode prejudicar o tempo de recuperação de uma falta devido ao excesso de arquivos criados. Similarmente, a descompressão de *checkpoints* eleva o tempo de recuperação de uma aplicação. Ainda, alguns modelos são baseados em métodos de previsão de faltas. Por exemplo, (LIANG et al., 2006) indica ser possível detectar aproximadamente 80% de falhas de rede e memória, e 47% de E/S, antes que elas efetivamente ocorram. Desta maneira é possível migrar um processo antes da ocorrência de uma falha, eliminando a sobrecarga imposta pelo processo de *checkpointing*.

Em Remus, o estado completo da MV é salvo e replicado numa frequência de dezenas ou até centenas de *checkpoints* por segundo, garantindo um tempo mínimo em relação a recuperação de uma falta de

parada. Ainda, a sobrecarga causada pelo processo de compactação e descompactação de um *checkpoint* fica a cargo dos hospedeiros físicos, não prejudicando o tempo de execução das aplicações protegidas.

4.5.2 Ambientes Virtualizados

Neste contexto, os trabalhos podem ser divididos em migração e replicação de MVs.

4.5.2.1 Migração de Máquinas Virtuais

O objetivo da migração em tempo real é minimizar o tempo de interrupção de serviço quando uma MV precisa ser movida para outro hospedeiro físico, seja por motivos de manutenção no hospedeiro, balanceamento de carga ou até por questão de economia de energia.

CloudNet (WOOD et al., 2011) propõe a migração de MVs sobre uma WAN (*wide-area network*) visando o balanceamento de carga. A tomada de decisão para tal migração depende da latência na comunicação observada pelo cliente final e/ou limites de capacidade. O processo final de migração da MV pode levar o cliente a perceber um alto período de indisponibilidade, ferindo a linearizabilidade.

O trabalho realizado por (HU et al., 2013) aborda o desempenho de diferentes MMVs (KVM, XenServer, VMware e Hyper-V) para a migração em tempo real. Dentre os resultados observados, o tempo total de migração e de indisponibilidade da MV e o tráfego de rede entre os hospedeiros são considerados. O mecanismo apresentado em nosso trabalho pouco contribuiria com soluções de migração em tempo real, já que a latência na comunicação percebida pelo cliente ocorrerá somente durante o período de indisponibilidade da MV, quando a migração entra na fase *stop-and-copy*, e não em seu estado livre de falhas.

4.5.2.2 Replicação de Máquinas Virtuais

(KOPPOL et al., 2011) fez um estudo sobre Remus avaliando a qualidade de serviços que utilizam pacotes de voz. O atraso causado pelo uso do *buffer* de rede, somado ao fato de que a liberação ocorre de uma só vez ao final de um *checkpoint*, levou a uma explosão de tráfego, degradando a qualidade do áudio. *Adaptive Remus* poderia acelerar a liberação do *buffer* de rede para essa aplicação.

Por sua vez, SecondSite (RAJAGOPALAN et al., 2012) utiliza as mesmas técnicas de replicação do Remus diferenciando o ambiente alvo: propõe a replicação completa de um *site* entre *data centers* dis-

tribuídos. O objetivo é migrar as MVs em caso de catástrofes naturais, faltas de *hardware* ou energia para outro *datacenter*. Nesta solução, a manutenção da conectividade das MVs é obtida através de configurações de roteamento previamente realizadas. Assim como no Remus, o tamanho do *checkpoint* a ser transferido para o *backup* é o fator determinante para o tempo de execução da MV. Técnicas de compressão para diminuir o tamanho de um *checkpoint* foram utilizadas, porém dependendo da aplicação, nem sempre são suficientes. SecondSite pode se beneficiar da proposta deste trabalho, uma vez que o enlace WAN existente entre os hospedeiros primário e *backup* pode aumentar a latência na comunicação percebida pelo cliente da MV.

(TAMURA et al., 2008) implementou uma solução para replicação de MVs que visa tolerância a faltas de parada utilizando o modelo primário-*backup*. O mecanismo de replicação não utiliza *buffer* de rede, replicando o estado da MV apenas na ocorrência de eventos de leitura ou escrita em disco. Um mecanismo para tolerar faltas de parada no MMV KVM foi implementado em (CUI et al., 2009), realizando uma migração contínua entre dois hospedeiros. Diferentemente de Remus, a MV utiliza um disco compartilhado e não há um *buffer* de rede implementado no hospedeiro. Essas abordagens diminuem a sobrecarga imposta ao tempo de processamento na MV, porém não garantem a linearizabilidade do sistema.

Uma solução para diminuir a sobrecarga do processo de *checkpointing* é apresentada em (ZHU et al., 2010). Nesse estudo, o tempo para salvar um *checkpoint* é guiado pelo comportamento de acesso a memória no MMV. Embora o procedimento possa diminuir o tempo de transferência de um *checkpoint*, a alteração dinâmica da frequência não é abordada.

Em outra solução, porém baseada no MMV VMware, a replicação do estado da MV não é baseada em *checkpoints*, e sim em *logs*, que são enviados ao *backup* imediatamente após uma instrução ser executada no primário (SCALES; NELSON; VENKITACHALAM, 2010). Semelhante ao Remus, o sistema utiliza um *buffer* de rede, inovando através da busca automática por um servidor com recursos disponíveis para atuar como *backup* (para tal, a solução utiliza armazenamento em disco compartilhado). Nesse cenário, o tempo de processamento no primário varia de acordo com a capacidade de processamento do hospedeiro *backup* selecionado. Além de ser uma solução fechada, aplica-se a servidores e não a máquinas comuns que utilizam *hardware* de prateleira, e limita a utilização de apenas um processador por MV.

(GEROFI; ISHIKAWA, 2011) mescla duas abordagens: adap-

tação do intervalo e diminuição do volume transferido. Em Remus, o volume (memória e disco) é otimizado com técnicas de compressão (CULLY et al., 2008) (RAJAGOPALAN et al., 2012). Enquanto esta proposta abordou 3 variáveis em sua formulação (memória, largura de banda e intervalo) o modelo proposto neste trabalho confia na otimização já existente em Remus, simplificando o processo de adaptação, considerando apenas o FSR para a tomada de decisão. Ainda, o limite superior para o intervalo de *checkpointing* foi definido pelos autores em 2 s, ou seja, 20 vezes o limite adotado pela presente proposta (os limiares adotados em *Adaptive Remus* visam a manutenção da confiabilidade da MV protegida).

4.6 Considerações Parciais

A proposta de *checkpointing* adaptativo se contrapõe aos modelos utilizados para replicação em ambientes não virtualizados no que diz respeito a responsividade. Uma vez que Remus não tem conhecimento algum sobre as características das aplicações em execução na MV, não é possível prever o custo da recuperação de uma falta de parada ou sequer mapear seus pontos críticos. Ainda, a proposta apresentada procura diminuir a sobrecarga imposta pelo modelo de replicação primário-*backup* em seu estado livre de faltas.

Adaptive Remus difere-se da implementação original de Remus em relação ao intervalo de *checkpointing* utilizado. Enquanto Remus utiliza o TDC e um intervalo fixo para reger a frequência da replicação, *Adaptive Remus* baseia-se no FSR da MV para possibilitar sua adaptação entre dois modos, rede e processamento. Enquanto no primeiro a alta frequência de replicação acelera a liberação do *buffer* de rede para diminuir a latência entre cliente e MV, no segundo a prioridade é o tempo de processamento das aplicações (não comunicantes) hospedadas.

5

ANÁLISE EXPERIMENTAL

Este capítulo apresenta a análise experimental realizada para verificar a eficácia e aplicabilidade de *Adaptive Remus*. Inicialmente, um conjunto de *benchmarks* e aplicações foi executado comparando diferentes configurações do mecanismo nativo de Remus com a solução proposta. A última etapa da análise investiga a sobrecarga de processamento introduzida por *Adaptive Remus*.

5.1 Visão Geral

Os experimentos compreendem aplicações sensíveis ao tempo de processamento, mistas e comunicantes. Os testes executados no Capítulo 3 indicaram que operações de E/S não interferem significativamente na definição do intervalo de *checkpointing*, e por isso não há um cenário exclusivo para este tipo de operação. Ainda, a vazão média causada pela replicação entre os hospedeiros primário e *backup* apresentou valores insignificantes perante a capacidade do enlace exclusivo para este fim. Assim, os resultados dos experimentos não discutem essa métrica. Exceto para o experimento NAS, que necessitou do acréscimo de três computadores, o ambiente de testes é semelhante ao utilizado no Capítulo 3. Atribuiu-se 30 para o valor da variável NCR, utilizado exclusivamente pela versão AR, em todos os experimentos.

5.2 Resultados Experimentais

Para verificar a eficácia da versão *Adaptive Remus* (AR), as linhas de base para comparação foram definidas pela execução da versão original de Remus com intervalos fixos de 25 ms e 100 ms (respectivamente chamadas de R25 e R100). Uma terceira abordagem, guiada somente pelo tempo de processamento da MV, sem considerar o FSR, foi implementada. Esta abordagem – Remus Flutuante (RF) – alterna entre dois intervalos de *checkpointing*, 25 e 100 ms. A alternância é definida pelo TDC calculado na própria rodada: se inferior a 25 ms, o intervalo para a próxima rodada deve se manter em 25 ms; se igual ou superior a 25 ms, deve ser alternado para 100 ms. O uso destes limiares foi baseado em testes de desempenho efetuados no Capítulo 3. A verificação do TDC ocorre ao final de cada *checkpointing* sendo o fator responsável por aumentar ou diminuir a frequência da replicação.

Assim, quatro versões de Remus foram consideradas:

- *Adaptive Remus*: identificada como AR nos gráficos. Trata-se da versão proposta neste trabalho0;
- Remus Flutuante: identificada como RF. Trata-se de uma versão implementada exclusivamente para fins de comparação;
- Remus: identificada como R25 e R100. Trata-se na versão original de Remus, com intervalos predeterminados e fixos de 25 ms e 100 ms, respectivamente.

5.2.1 Métricas

O tempo de execução total de cada tarefa foi medido durante os testes e utilizado como parâmetro para avaliar a sobrecarga imposta à aplicação hospedada durante a replicação da MV. Quanto maior o tempo para a execução completa de uma tarefa, maior a sobrecarga. Complementarmente, a utilização de CPU no hospedeiro primário foi monitorada para fim de comparação entre as versões abordadas.

Os experimentos foram divididos de acordo com o uso de recursos pela MV, como mostra a Tabela 4. Exceto para o teste de utilização de CPU, que monitora o consumo de CPU no hospedeiro primário, o tempo de execução é contabilizado como métrica de comparação para avaliar as quatro versões de Remus.

Teste	Recursos Utilizados	Objetivos
Compilação, Da-Capo Tomcat	Mista (CPU e E/S)	avaliar tempo de processamento
Transferência de um arquivo	Rede	avaliar latência na comunicação
NAS MG, RUBiS	Mista (CPU e Rede)	avaliar tempo de processamento e rede
Utilização de CPU	CPU	avaliar a sobrecarga de cada mecanismo

Tabela 4 – Resumo dos testes executados.

5.2.2 Aplicações Mistas (CPU e E/S)

Este experimento avalia a sobrecarga imposta pelas quatro versões de Remus para aplicações mistas, sensíveis ao tempo de processamento e operações de E/S. O tempo de execução das tarefas, medido em segundos, é usado como métrica para a análise da sobrecarga imposta em cada versão.

5.2.2.1 Compilação

Neste experimento foi efetuada a compilação do BIND em uma MV protegida pelo Remus. Este processo envolve uso de CPU e memória, ou seja, simula uma aplicação dependente principalmente do tempo de processamento, já que não envolve comunicação entre cliente e servidor. Ainda, operações de E/S também estão presentes neste cenário, uma vez que dezenas de novos arquivos, que serão replicados a cada *checkpoint*, são criados e salvos no disco local da MV. A Figura 20 apresenta o comparativo entre as quatro versões.

Aplicações dependentes exclusivamente de processamento apresentam um melhor desempenho quando a frequência de *checkpointing* é mais baixa, uma vez que intervalos maiores diminuem a interferência na execução da MV. As versões RF e AR se adaptaram à natureza do experimento, obtendo resultados semelhantes a R100, que serve como base de melhor desempenho. Analisando AR e R25, pode-se afirmar que AR teve um tempo médio 25% menor. Para R25, sempre que o TDC for maior que 25 ms, o intervalo passa a ser o próprio TDC. Tomando como exemplo um TDC de 30 ms, em R25 o intervalo utilizado para

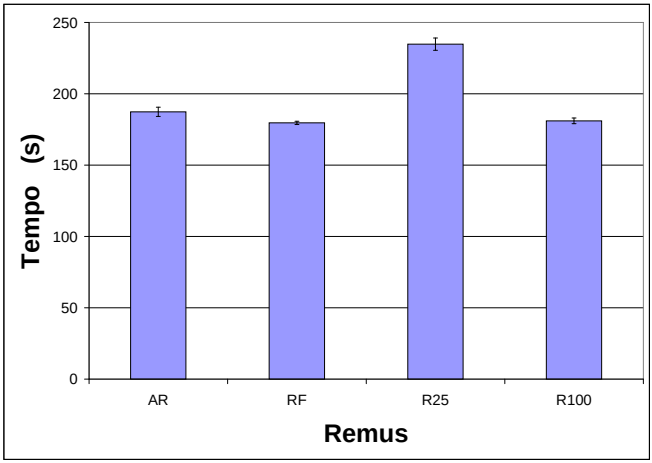


Figura 20 – Tempo de compilação do BIND para as quatro versões de Remus.

salvar o próximo *checkpoint* será de 30 ms, enquanto que, para AR, RF e R100, será de 100 ms. Ou seja, essas versões garantem o intervalo determinado para o TEMV.

Em suma, para este tipo de aplicação, quanto mais baixa a frequência de *checkpointing*, maior o tempo de execução da MV a cada *checkpoint*, e menos interrupções na execução da MV.

5.2.2.2 DaCapo Tomcat

O *benchmark* DaCapo Tomcat (BLACKBURN et al., 2006) simula a execução de um conjunto de consultas a um servidor Tomcat[1]. Este teste não envolve uma conexão entre clientes e servidores, sendo totalmente dependente do tempo de processamento da MV. A Figura 21 mostra os resultados obtidos para as quatro versões de Remus.

Em resumo, todas a versões apresentaram um desempenho semelhante. Isso se deve pela carga de processamento constante imposta pelo *benchmark*, fazendo com que o TDC superasse o limite de 100 ms em praticamente todos os *checkpoints*. A captura dos bytes na interface da MV a cada *checkpoint* elevou ligeiramente o TDC na versão AR, aumentando igualmente o TEMV a cada rodada. Essa característica explica a pequena vantagem de desempenho da versão AR em

[1] Disponível em <http://tomcat.apache.org>

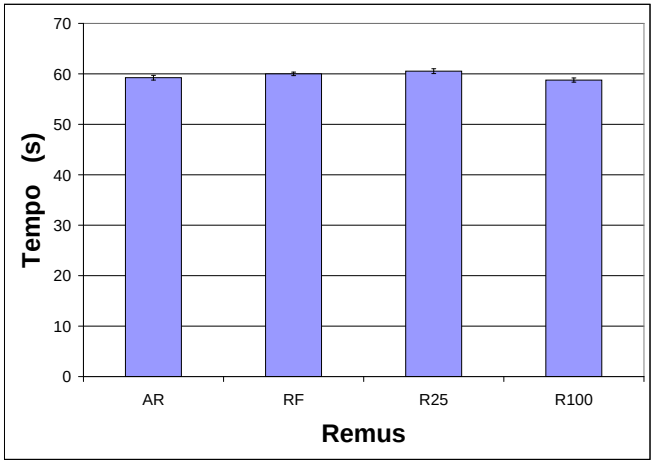


Figura 21 – Resultados obtidos com a execução do *benchmark* DaCapo Tomcat para as quatro versões de mecanismo Remus.

relação a R25. A sobrecarga no hospedeiro primário será discutida na Seção 5.2.5.

5.2.3 Aplicações Sensíveis a Latência de Rede

Para avaliar o desempenho das quatro versões de Remus em relação a latência e seu impacto nas aplicações comunicantes, foi efetuada a transferência de um arquivo de 50 MB da MV para a máquina cliente. O tempo de transferência deste arquivo é de aproximadamente 4,5 segundos quando Remus não está em execução. O tempo de execução da transferência também é usado como base para a análise da sobrecarga.

A Figura 22 apresenta os resultados obtidos. Um intervalo de 100 ms entre cada *checkpoint* é desejável para aplicações que necessitam de tempo de processamento, mas o mesmo não ocorre com as aplicações comunicantes, ou seja, a versão R100 torna-se impraticável neste cenário. Como esperado, os resultados de R25 e RF foram semelhantes (já que ambas têm como intervalo padrão 25 ms). AR mostrou-se a melhor opção, uma vez que não espera um tempo mínimo para iniciar o salvamento de um novo *checkpoint*. Os dados existentes no *buffer* de rede são liberados na maior frequência possível, ou seja, um novo *checkpoint* é salvo assim que o anterior estiver completamente salvo no *backup*. O

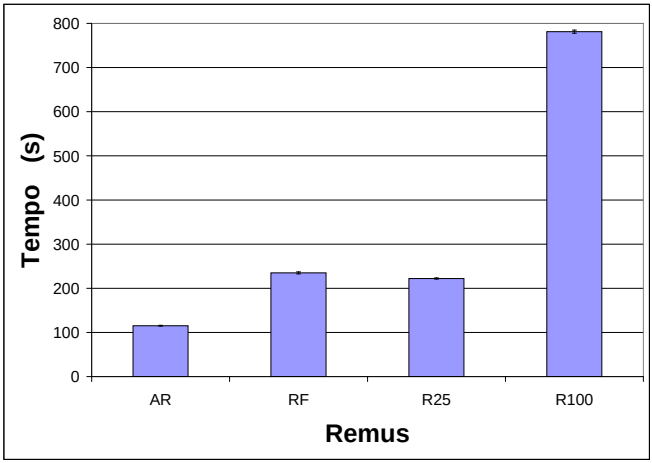


Figura 22 – Tempo para transferência de um arquivo de 50MB da MV para a máquina cliente.

tempo de transferência de AR foi aproximadamente 93% menor que o de R25 e RF.

5.2.4 Aplicações Mistas (CPU e Rede)

Duas aplicações mistas, sensíveis ao tempo de processamento e ao uso da rede, são avaliadas. Assim como na Seção 5.2.2, o tempo de execução das tarefas é adotado como base para a análise da sobrecarga imposta para cada versão.

5.2.4.1 *Benchmark* NAS

O *benchmark* NAS pode ser definido como um conjunto de aplicações destinadas a avaliar o desempenho em arquiteturas em computação paralela (BAILEY et al., 1991). Para este experimento, além da MV residente no hospedeiro primário, outros três hospedeiros continham uma MV configurada com o *benchmark* NAS, totalizando quatro MVs para o teste. Das quatro, a única MV protegida por Remus era a residente no hospedeiro primário.

Dentre as aplicações disponíveis no *benchmark* (versão NPB3.3-MPI), escolheu-se o MG (*multi-grid*), por possuir um comportamento misto em relação a uso de recursos (intervalos definidos de processamento e comunicação). Além disso, MG apresenta o maior consumo de

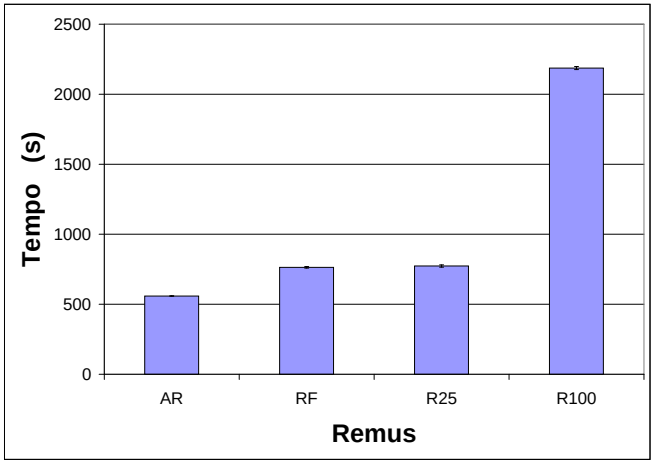


Figura 23 – Para a aplicação NAS-MG, uma baixa frequência de *checkpointing*, como 100 ms, resulta em uma alta sobrecarga que afeta o desempenho da aplicação.

memória, com tempo de processamento médio de 59,6%, e o terceiro menor tráfego de rede entre os nós (MVs) hospedados (SUBHLOK; VENKATARAMAIAH; SINGH, 2002). Como observado na Figura 23, AR apresentou o melhor resultado em relação ao tempo de execução do experimento. Os resultados indicam que a latência na comunicação é o fator preponderante. RF e R25 apresentaram resultados semelhantes: em aplicações que envolvem transferência de rede, é preferível que a frequência de *checkpointing* seja regida pelo TDC, como faz AR, e não por intervalos mínimos ou fixos, como é o caso de RF, R25 e R100. Sempre que o TDC for menor que o intervalo predeterminado, maior a latência na comunicação, uma vez que Remus atrasa o salvamento de um novo *checkpoint* para atingir o intervalo predeterminado. R25 apresentou uma sobrecarga de 38% em relação a AR.

5.2.4.2 RUBiS

Assim como NAS, este experimento visa avaliar as quatro versões do Remus quando a aplicação em execução na MV requer características mistas de recursos, ou seja, processamento e rede, mas em proporções distintas. Para execução do RUBiS[2], o cenário foi dividido

[2] Disponível em <http://rubis.ow2.org/>

em i) *Servidor*: simula as funcionalidades básicas de um *site* de leilão como o Ebay, como navegação, busca e negociações de compra e venda; e ii) *Cliente*: trata-se de um emulador de clientes para servidor, sendo que cada cliente gera um sequência de interações com o *site*. O servidor RUBiS está em execução na MV residente no hospedeiro primário, enquanto o emulador executa na máquina cliente. A métrica escolhida para o analisar o comportamento da aplicação foi o número de operações atendidas pelo *site*. Cada operação compreende a execução de pesquisas (por categoria, região) e a visualização dos itens encontrados. No total, foi simulada a interação de 300 clientes com o *site* de leilão.

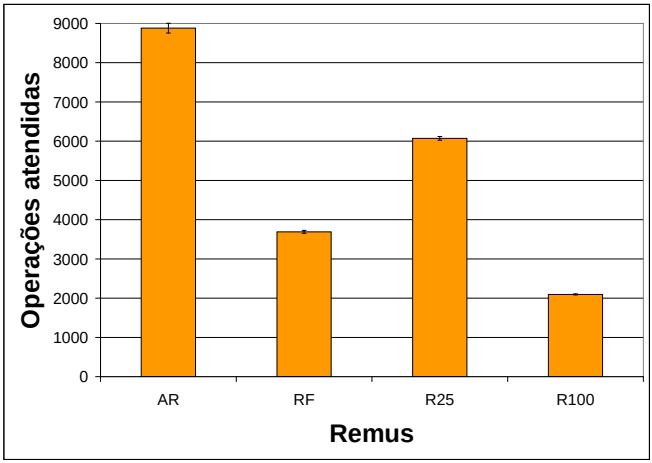


Figura 24 – Resultado do *benchmark* RUBiS para as quatro versões do Remus em relação ao número de operações realizadas pelos clientes.

A Figura 24 mostra que a latência na comunicação, assim como observado na execução do *benchmark* NAS MG, é o fator responsável pelo desempenho final da aplicação. AR obteve um desempenho aproximadamente 350% superior a R100, uma vez que o intervalo para R100 salvar um novo *checkpoint* é sempre de, no mínimo, 100 ms, o que aumenta a latência na comunicação. RF, que utiliza um intervalo de 100 ms sempre que o TDC for maior que 25 ms, alcança o segundo pior desempenho. Já R25 obteve uma sobrecarga em torno de 31% se comparado a AR. Assim como no NAS MG, é preferível que a frequência de *checkpointing* seja regida pelo TDC, como faz AR.

5.2.5 Sobrecarga no Hospedeiro Primário

Para verificar a aplicabilidade da solução proposta, é necessário quantificar a sobrecarga imposta pelas abordagens implementadas. Assim, observou-se a utilização média de processador do hospedeiro primário, uma vez que é o hospedeiro responsável por salvar e transferir os *checkpoints* para o *backup*.

Dois experimentos foram realizados: o primeiro com a MV ociosa, conforme ilustra a Figura 25, e o segundo com o Dacapo Tomcat em execução, conforme ilustra a Figura 26. Para fins de comparação entre os dois modos de operação de AR, AR+ foi adicionado aos experimentos para representar a versão Adapative Remus propositadamente fixada no modo rede. A utilização de CPU foi capturada com intervalos de 1 segundo com a ferramenta mpstat [3].

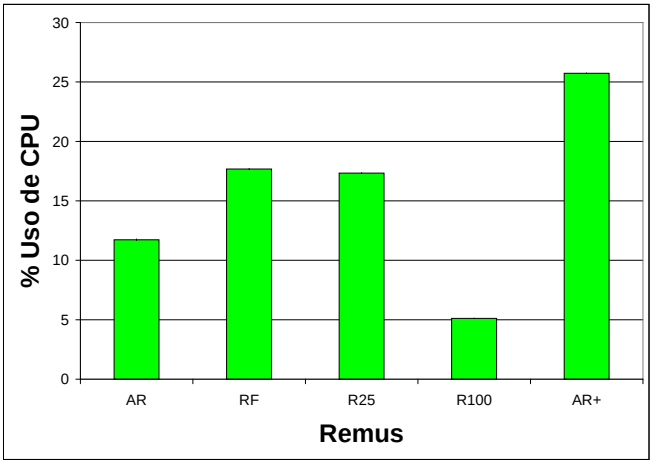


Figura 25 – Análise do uso de CPU imposta ao hospedeiro primário com a MV ociosa.

Verifica-se que o percentual de uso da CPU aumenta conforme aumenta a frequência de *checkpointing*, como mostra a Figura 25. Exceto para AR+, a taxa de utilização durante o teste não chegou a 20% em nenhuma das versões. R25 e RF apresentam um percentual semelhante devido ao seu intervalo padrão, predeterminado em 25 ms. Entre R100 e AR (que utilizam 100 ms como intervalo padrão), a diferença é motivada pela coleta de métricas realizada por AR para identificar o

[3] Disponível em <http://sebastien.godard.pagesperso-orange.fr/>

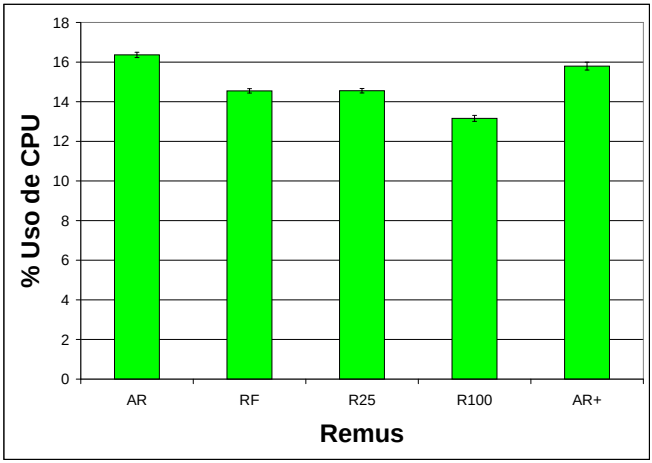


Figura 26 – Análise do uso de CPU no hospedeiro primário com o *benchmark* Dacapo Tomcat em execução.

FSR da MV a cada *checkpoint*. AR+ ilustra o pior caso pelo fato de praticamente não haver dados a serem replicados quando a MV está ociosa, fazendo com que o TDC seja extremamente curto e consequentemente elevando a frequência da replicação e o consumo de CPU. Mesmo assim, o uso de CPU observado foi inferior a 30%.

A diferença do percentual de processamento entre as versões diminui quando a MV sofre uma carga de uso de processamento maior e constante, como ilustra a Figura 26. Isto é, quando o TDC ultrapassa constantemente 100 ms (limiar superior para AR, RF e R100 e AR+) ou quatro vezes o predeterminado para R25, todas as versões passam a ter o mesmo TDC, e consequentemente a mesma frequência de *checkpointing*.

5.2.6 Discussão dos Resultados

O modo de adaptação AR obteve um desempenho superior, ou bem próximo ao melhor, em todos os experimentos, adaptando-se às características da aplicação que está em execução na MV. Ainda, ao contrário de R25, R100 e RF, AR tem a capacidade de aumentar, ao máximo possível, a frequência de *checkpointing*, salvando um novo *checkpoint* imediatamente após o anterior estar devidamente salvo e confirmado no hospedeiro *backup*. Em relação ao tempo de processa-

mento, como AR adota um intervalo de 100 ms quando não há comunicação envolvida, seu desempenho mostra-se semelhante a R100 e superior a R25 para as aplicações que necessitam de maior tempo de execução da MV. A justificativa é dada pela variação do TDC: por exemplo, um TDC de 26 ms fará com que o TEMV em R25 seja exatamente de 26 ms. Já em AR, o TEMV será de 100 ms. Ademais, sempre que o TDC superar o intervalo estipulado, Remus original (R25 e R100) adota TDC como intervalo padrão, e passa a indiretamente priorizar o processamento em detrimento da comunicação. Quando a aplicação é mista e dependente tanto de rede quanto de processamento, como é o caso de RUBiS e NAS MG, aguardar seguidamente 100 ms para salvar um novo *checkpoint* torna-se prejudicial. Nesses casos, o aumento da frequência de *checkpointing* resulta em um melhor desempenho.

5.3 Considerações Parciais

O uso de um intervalo fixo, mesmo que de poucos milissegundos, pode ser prejudicial para a latência na comunicação da MV. *Adaptive Remus* utiliza somente o TDC como intervalo de *checkpointing* quando está no modo rede, ou seja, inicia o salvamento de um novo *checkpoint* assim que o anterior estiver salvo no hospedeiro *backup*. Mesmo que isso implique uma diminuição considerável do TEMV em relação às outras três versões, os resultados mostram uma menor sobrecarga em aplicações mistas, que envolvem rede e processamento, ou exclusivamente rede. Além disso, na ausência de FSR, *Adaptive Remus* também é capaz de aumentar o TEMV, diminuindo a frequência de *checkpoints* para priorizar o tempo de processamento na MV.

6

CONSIDERAÇÕES FINAIS

Infraestruturas computacionais estão sujeitas a interrupções. Usuários de aplicações críticas recorrem a soluções para implementação de alta disponibilidade, geralmente investindo recursos na aquisição de *hardware* ou *software* especializados. Entretanto, com o advento da virtualização, técnicas para provimento de alta disponibilidade tornaram-se economicamente acessíveis. Nesse contexto, destaca-se Remus (CULLY et al., 2008), um mecanismo de replicação de MVs residente no hipervisor Xen (BARHAM et al., 2003) que proporciona tolerância a faltas de parada. Baseado no modelo de replicação primário-*backup*, Remus mantém uma cópia atualizada da MV em execução, encapsulando as aplicações por ela hospedadas.

Através de uma sincronização constante (dezenas ou centenas de *checkpoints* por segundo), a MV replicada no segundo hospedeiro é mantida em estado de pausa, estando pronta para assumir a execução na ocorrência de uma falta de parada no hospedeiro primário. Essa substituição ocorre em poucos milissegundos, sem que haja perda de conexão entre as aplicações hospedas e seus clientes. Em suma, o processo de replicação da memória da MV assemelha-se ao *stop-and-copy* utilizado na migração de MVs (CLARK et al., 2005), enquanto a replicação de seu disco utiliza uma versão adaptada da ferramenta DRBD (REISNER; ELLENBERG, 2005).

Embora promissora, a aplicação de Remus em cenários de produção enfrenta algumas limitações:

1. Introdução de sobrecarga computacional que influencia o desempenho da aplicações hospedadas;

2. Por ser guiado por um intervalo de *checkpoiting* fixo, Remus não trabalha bem com aplicações mistas, sensíveis a comunicação e tempo de processamento;

3. O tempo de duração de um *checkpoint* atrasa o salvamento de um novo *checkpoint* sempre que supera o intervalo predeterminado.

Adaptive Remus busca aliviar a sobrecarga imposta relativa ao item 2, uma vez que os itens 1 e 3 são inerentes a uma característica primordial do mecanismo: a linearizabilidade. Sem ferir a linearizabilidade, *Adaptive Remus* é capaz de operar em dois modos, rede e processamento. Baseado no fluxo de saída de rede da MV, e não mais em um único intervalo de tempo fixo e predeterminado, o mecanismo é capaz de acelerar a liberação do *buffer* de rede, aumentando ao máximo possível a frequência de *checkpointing* para diminuir a latência na comunicação da MV. Na ausência deste fluxo, o mecanismo volta a seu modo padrão de operação, processamento, garantindo o tempo de mínimo de execução predeterminado pelo usuário, diminuindo a periodicidade da replicação.

Para aplicações comunicantes, que envolvem exclusivamente a transferência de rede, os resultados mostram que *Adaptive Remus* obteve um desempenho 93% superior quando comparado a Remus sob um intervalo de 25 ms para *checkpointing*. A principal diferença que justifica este resultado está diretamente ligada ao fato de *Adapative Remus* não utilizar um intervalo mínimo para reger a frequência de *checkpointing* quando está operando no modo rede, ou seja, não há nenhum incremento no tempo de duração de um *checkpointing* para atingir um tempo mínimo de execução da MV. Aplicações mistas, que envolvem rede e tempo de processamento, beneficiaram-se igualmente da menor sobrecarga imposta pelo intervalo de replicação adaptável, porém num menor percentual. Em relação a aplicações exclusivamente sensíveis ao tempo de processamento, como a compilação, *Adaptive Remus* obteve resultados similares a Remus sob um intervalo fixo de 100 ms. Isso mostra a capacidade do mecanismo proposto de adaptar frequências mais baixas para beneficiar este tipo de aplicação.

O percentual do consumo de CPU no hospedeiro primário mostra que, quanto mais alta a frequência da replicação, maior sua carga de processamento. Contudo, quanto maior a carga na MV, menor o consumo de CPU observado no hospedeiro primário. Em suma, quanto maior o tempo de duração de *checkpointing*, maior o tempo de execução da MV, e menor o consumo de CPU no hospedeiro primário.

Adaptive Remus é capaz de diminuir a sobrecarga imposta pela replicação às aplicações hospedadas. A adaptabilidade do intervalo, que permite variar a frequência de *checkpointing* entre alta e baixa de acordo com fluxo de saída de rede da MV, faz-se necessária principalmente para aplicações mistas sensíveis a latência de rede e tempo de processamento, ou simplesmente comunicantes, uma vez que o uso de um intervalo fixo pode atrasar, desnecessariamente, a liberação do *buffer* de rede.

6.1 Trabalhos Futuros

Como perspectiva para continuação do trabalho, identificamos duas linhas:

- o fato de Remus não respeitar a frequência estipulada para *checkpointing* é prejudicial principalmente para aplicações comunicantes. Mesmo diminuindo o tempo de execução da MV ao máximo possível, não há garantia que o tempo de transferência do *checkpoint* será inferior ao intervalo estipulado. Uma solução possível seria a fixação do início de um *checkpoint*, exatamente conforme estipulado pelo usuário, independente de o *checkpoint* anterior já ter sido salvo no *backup* ou não. A versão atual de Remus permite esta execução em modo experimental e não há ainda documentação disponível sobre seu uso. Em suma, propõe-se a combinação de *Adaptive Remus* com o modo experimental existente.

- os experimentos discutidos neste trabalho referem-se à execução do Remus em uma rede local, sendo os hospedeiro primário e *backup* interligados por um cabo de rede *crossover*. Surge um questionamento quanto a aplicabilidade da solução proposta em ambientes geograficamente distribuídos, conectados por enlaces WAN (*Wide-Area Network*). O objetivo seria verificar a aplicabilidade do *checkpointing* adaptativo neste ambiente.

6.2 Trabalhos Publicados

Durante o desenvolvimento deste trabalho, alguns resultados parciais foram publicados no XV e XVI Workshop de Testes e Tolerância a Falhas (WTF 2014/2015), organizado em conjunto com o Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC). O artigo intitulado *Uma Análise da Sobrecarga Imposta pelo Mecanismo de Replicação de Máquinas Virtuais Remus* (SILVA;

KOSLOVSKI; OBELHEIRO, 2014) apresentou uma avaliação em relação ao intervalo de *checkpointing* utilizado por Remus. O estudo revela o antagonismo existente entre aplicações voltadas exclusivamente ao tempo de processamento *versus* as sensíveis a latência na comunicação. O segundo artigo, publicado em 2015, intitulado *Replicação de Máquinas Virtuais Xen com Checkpointing Adaptável* (SILVA; OBELHEIRO; KOSLOVSKI, 2015), apresenta a diminuição da sobrecarga imposta pela variação do intervalo para *checkpoint* no Remus.

Referências

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 1, n. 1, p. 11–33, jan. 2004. ISSN 1545-5971. Disponível em: <http://dx.doi.org/10.1109/TDSC.2004.2>.

AYARI, N.; BARBARON, D.; LEFEVRE, L.; PRIMET, P. Fault tolerance for highly available internet services: concepts, approaches, and issues. *Communications Surveys Tutorials, IEEE*, v. 10, n. 2, p. 34–46, Second 2008. ISSN 1553-877X.

BAILEY, D. H.; BARSZCZ, E.; BARTON, J. T.; BROWNING, D. S.; CARTER, R. L.; DAGUM, D.; FATOOHI, R. A.; FREDERICKSON, P. O.; LASINSKI, T. A.; SCHREIBER, R. S.; SIMON, H. D.; VENKATAKRISHNAN, V.; WEERATUNGA, S. K. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, v. 5, n. 3, p. 63–73, Fall 1991. Disponível em: <citeseer.ist.psu.edu/article/bailey94nas.html>.

BARHAM, P.; DRAGOVIC, B.; FRASER, K.; HAND, S.; HARRIS, T.; HO, A.; NEUGEBAUER, R.; PRATT, I.; WARFIELD, A. Xen and the art of virtualization. *SIGOPS Operating Systems Review*, ACM, New York, NY, USA, v. 37, n. 5, p. 164–177, out. 2003. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/1165389.945462>.

BLACKBURN, S. M.; GARNER, R.; HOFFMAN, C.; KHAN, A. M.; MCKINLEY, K. S.; BENTZUR, R.; DIWAN, A.; FEINBERG,

D.; FRAMPTON, D.; GUYER, S. Z.; HIRZEL, M.; HOSKING, A.; JUMP, M.; LEE, H.; MOSS, J. E. B.; PHANSALKAR, A.; STEFANOVIć, D.; VanDrunen, T.; DINCKLAGE, D. von; WIEDERMANN, B. The DaCapo benchmarks: Java benchmarking development and analysis. In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, 2006. p. 169–190.

BUDHIRAJA, N.; MARZULLO, K.; SCHNEIDER, F. B.; TOUEG, S. The primary-backup approach. In: MULLENDER, S. (Ed.). *Distributed Systems (2Nd Ed.)*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993. p. 199–216. ISBN 0-201-62427-3. Disponível em: <http://dl.acm.org/citation.cfm?id=302430.302438>.

CARISSIMI, A. *Virtualização: da teoria a soluções*. [S.l.]: Minicursos do XXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, 2008. 173-207 p.

CHTEPEN, M.; DHOEDT, B.; TURCK, F. D.; DEMEESTER, P.; CLAEYS, F.; VANROLLEGHEM, P. Adaptive checkpointing in dynamic grids for uncertain job durations. In: *Information Technology Interfaces, 2009. ITI '09. Proceedings of the ITI 2009 31st International Conference on*. [S.l.: s.n.], 2009. p. 585–590. ISSN 1330-1012.

CLARK, C.; FRASER, K.; HAND, S.; HANSEN, J. G.; JUL, E.; LIMPACH, C.; PRATT, I.; WARFIELD, A. Live migration of virtual machines. In: *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*. Berkeley, CA, USA: USENIX Association, 2005. (NSDI'05), p. 273–286. Disponível em: <http://dl.acm.org/citation.cfm?id=1251203.1251223>.

CUI, W.; MA, D.; WO, T.; LI, Q. Enhancing reliability for virtual machines via continual migration. In: *Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2009. (ICPADS '09), p. 937–942. ISBN 978-0-7695-3900-3. Disponível em: <http://dx.doi.org/10.1109/ICPADS.2009.20>.

CULLY, B.; LEFEBVRE, G.; MEYER, D.; FEELEY, M.; HUTCHINSON, N.; WARFIELD, A. Remus: High availability via asynchronous virtual machine replication. In: *Proceedings of*

the 5th USENIX Symposium on Networked Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, 2008. (NSDI'08), p. 161–174. ISBN 111-999-5555-22-1. Disponível em: <http://dl.acm.org/citation.cfm?id=1387589.1387601>.

DU, Y.; YU, H. Paratus: Instantaneous failover via virtual machine replication. In: *Grid and Cooperative Computing, 2009. GCC '09. Eighth International Conference on*. [S.l.: s.n.], 2009. p. 307–312.

EGWUTUOHA, I. P.; LEVY, D.; SELIC, B.; CHEN, S. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *J. Supercomput.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 65, n. 3, p. 1302–1326, set. 2013. ISSN 0920-8542. Disponível em: <http://dx.doi.org/10.1007/s11227-013-0884-0>.

ELNOZAHY, E. N. M.; ALVISI, L.; WANG, Y.-M.; JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, ACM, New York, NY, USA, v. 34, n. 3, p. 375–408, set. 2002. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/568522.568525>.

GEROFI, B.; ISHIKAWA, Y. Workload adaptive checkpoint scheduling of virtual machine replication. In: *Dependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on*. [S.l.: s.n.], 2011. p. 204–213.

GUERRAOUI, R.; SCHIPER, A. Software-based replication for fault tolerance. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 30, n. 4, p. 68–74, abr. 1997. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/2.585156>.

HU, W.; HICKS, A.; ZHANG, L.; DOW, E. M.; SONI, V.; JIANG, H.; BULL, R.; MATTHEWS, J. N. A quantitative study of virtual machine live migration. In: *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. New York, NY, USA: ACM, 2013. (CAC '13), p. 11:1–11:10. ISBN 978-1-4503-2172-3. Disponível em: <http://doi.acm.org/10.1145/2494621.2494622>.

KOPPOL, P.; NAMJOSHI, K.; STATHOPOULOS, T.; WILFONG, G. The inherent difficulty of timely primary-backup replication. In: *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM,

2011. (PODC '11), p. 349–350. ISBN 978-1-4503-0719-2. Disponível em: <http://doi.acm.org/10.1145/1993806.1993878>.

LAMPSON, B. W.; STURGIS, H. E. *Crash Recovery in a Distributed Data Storage System*. 1979. Xerox Palo Alto Research Center.

LAUREANO, M. A. P.; MAZIERO, C. A. *Virtualização: Conceitos e Aplicações em Segurança*. [S.l.]: Minicursos do VIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, 2008. 139-187 p.

LIANG, Y.; ZHANG, Y.; JETTE, M.; SIVASUBRAMANIAM, A.; SAHOO, R. Bluegene/l failure analysis and prediction models. In: *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*. [S.l.: s.n.], 2006. p. 425–434.

MEDINA, V.; GARCÍA, J. M. A survey of migration mechanisms of virtual machines. *ACM Computing Surveys*, ACM, New York, NY, USA, v. 46, n. 3, p. 30:1–30:33, jan. 2014. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/2492705>.

Nacamura Jr., L. *Proposta de um Esquema de Tolerância a Faltas Baseado em Múltiplas Replicações de Processos: Esquema MR*. Tese (Tese de doutorado) — Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal da Santa Catarina, Florianópolis, SC, novembro de 1996.

OBELHEIRO, R. R. *Tolerância a Intrusões como Abordagem de Segurança em Sistemas Distribuídos*. Tese (Monografia de qualificação de doutorado) — Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis, SC, setembro de 2003.

PETROVIC, D.; SCHIPER, A. Implementing virtual machine replication: A case study using Xen and KVM. In: *Proceedings of the 2012 IEEE 26th International Conference on Advanced Information Networking and Applications*. Washington, DC, USA: IEEE Computer Society, 2012. (AINA '12), p. 73–80. ISBN 978-0-7695-4651-3. Disponível em: <http://dx.doi.org/10.1109/AINA.2012.50>.

PLANK, J.; LI, K. ickp: a consistent checkpointer for multicomputers. *Parallel Distributed Technology: Systems Applications, IEEE*, v. 2, n. 2, p. 62–67, Summer 1994. ISSN 1063-6552.

PLANK, J. S.; BECK, M.; KINGSLEY, G.; LI, K. Libckpt: Transparent checkpointing under unix. In: *Proceedings of the USENIX 1995 Technical Conference Proceedings*. Berkeley, CA, USA: USENIX Association, 1995. (TCON'95), p. 18–18. Disponível em: <http://dl.acm.org/citation.cfm?id=1267411.1267429>.

POPEK, G. J.; GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, ACM, New York, NY, USA, v. 17, n. 7, p. 412–421, jul. 1974. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/361011.361073>.

RAJAGOPALAN, S.; CULLY, B.; O'CONNOR, R.; WARFIELD, A. SecondSite: Disaster tolerance as a service. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 47, n. 7, p. 97–108, mar. 2012. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/2365864.2151039>.

REISNER, P.; ELLENBERG. DRBD v8 – replicated storage with shared disk semantics. In: *Proceedings of the 12th International Linux System Technology Conference*. [S.l.: s.n.], 2005.

ROSENBLUM, M. The reincarnation of virtual machines. *Queue*, ACM, New York, NY, USA, v. 2, n. 5, p. 34–40, jul. 2004. ISSN 1542-7730. Disponível em: <http://doi.acm.org/10.1145/1016998.1017000>.

SCALES, D. J.; NELSON, M.; VENKITACHALAM, G. The design of a practical system for fault-tolerant virtual machines. *SIGOPS Operating Systems Review*, ACM, New York, NY, USA, v. 44, n. 4, p. 30–39, dez. 2010. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/1899928.1899932>.

SILVA, M. da; KOSLOVSKI, G.; OBELHEIRO, R. Uma análise da sobrecarga imposta pelo mecanismo de replicação de máquinas virtuais Remus. In: *XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC) - Workshop de Testes e Tolerância a Falhas (WTF)*. Florianópolis, Brasil: [s.n.], 2014.

SILVA, M. da; OBELHEIRO, R.; KOSLOVSKI, G. Replicação de máquinas virtuais xen com checkpointing adaptável. In: *XXXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC) - Workshop de Testes e Tolerância a Falhas (WTF)*. Vitória, Brasil: [s.n.], 2015.

STEVENS, W. R. *TCP/IP Illustrated (Vol. 1): The Protocols*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1993. ISBN 0-201-63346-9.

SUBHLOK, J.; VENKATARAMAIAH, S.; SINGH, A. Characterizing NAS benchmark performance on shared heterogeneous networks. In: *Proceedings of the 16th International Parallel and Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2002. (IPDPS '02), p. 91–. ISBN 0-7695-1573-8. Disponível em: <http://dl.acm.org/citation.cfm?id=645610.661710>.

TAMURA, Y.; SATO, K.; KIHARA, S.; MORIAI, S. Kemari: VM Synchronization for Fault Tolerance. In: *USENIX '08 Poster Session*. [S.l.: s.n.], 2008.

VMware. *VMware vSphere Fault Tolerance (FT)*. 2012. Available: <http://www.vmware.com/products/vsphere/features/fault-tolerance>.

VMware. *VMware vSphere High Availability (FT)*. 2012. Available: <http://www.vmware.com/products/vsphere/features/high-availability.html>.

WOOD, T.; RAMAKRISHNAN, K. K.; SHENOY, P.; MERWE, J. van der. CloudNet: Dynamic pooling of cloud resources by live wan migration of virtual machines. In: *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. New York, NY, USA: ACM, 2011. (VEE '11), p. 121–132. ISBN 978-1-4503-0687-4. Disponível em: <http://doi.acm.org/10.1145/1952682.1952699>.

ZHANG, Y.; CHAKRABARTY, K. Energy-aware adaptive checkpointing in embedded real-time systems. In: *Design, Automation and Test in Europe Conference and Exhibition, 2003*. [S.l.: s.n.], 2003. p. 918–923. ISSN 1530-1591.

ZHU, J.; DONG, W.; JIANG, Z.; SHI, X.; XIAO, Z.; LI, X. Improving the performance of hypervisor-based fault tolerance. *Parallel and Distributed Processing Symposium, International*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 1–10, 2010.

ZIV, A.; BRUCK, J. An on-line algorithm for checkpoint placement. *Computers, IEEE Transactions on*, v. 46, n. 9, p. 976–985, Sep 1997. ISSN 0018-9340.