Uma Análise da Sobrecarga Imposta pelo Mecanismo de Replicação de Máquinas Virtuais Remus

Marcelo Pereira da Silva, Guilherme Koslovski*, Rafael R. Obelheiro*

Programa de Pós-Graduação em Computação Aplicada (PPGCA)
Universidade do Estado de Santa Catarina (UDESC) – Joinville, SC – Brasil

pereira@joinville.udesc.br, *{firstname.lastname}@udesc.br

Abstract. *Remus is a primary-backup replication mechanism for the Xen hypervisor, providing high availability to virtual machines by frequent checkpointing (on the order of tens of checkpoints per second). Despite providing good fault tolerance for crash and omission failures, the performance implications of using Remus are not well understood. We experimentally characterize Remus performance under different scenarios and show that (i) the checkpointing frequency that provides the best performance is highly dependent on application behavior, and (ii) the overhead might be prohibitive for latency-sensitive network applications.*

Resumo. *Remus é um mecanismo de replicação primário-backup para o hipervisor Xen, que fornece alta disponibilidade a máquinas virtuais realizando checkpoints com frequência (na ordem de dezenas de checkpoints por segundo). Apesar de oferecer boa tolerância a faltas para falhas de parada e omissão, as implicações de desempenho do Remus não são bem compreendidas. Este trabalho caracteriza experimentalmente o desempenho do Remus em diferentes cenários e mostra que (i) a frequência de checkpointing que resulta no melhor desempenho é altamente dependente do comportamento das aplicações e (ii) o overhead pode ser proibitivo para aplicações de rede sensíveis a latência.*

1. Introdução

Eventuais faltas em recursos computacionais, de comunicação e armazenamento podem comprometer a execução de uma aplicação ou até mesmo interrompê-la. Este fato motiva a constante busca por soluções computacionais (algoritmos, arquiteturas, protocolos e infraestruturas) com alta disponibilidade. Pesquisadores e projetistas de *software* têm criando soluções para o desenvolvimento de aplicações confiáveis, capazes de sobreviver a interrupções no substrato computacional. Infelizmente, esta tarefa é complexa e usualmente as soluções encontradas aumentam o tempo de desenvolvimento (inclusão de verificações de estado, *checkpoints*, entre outros), impondo ainda uma sobrecarga de processamento perceptível ao usuário da aplicação.

Paralelamente, os centros de processamento de dados têm explorado as vantagens introduzidas pela virtualização de recursos computacionais, tais como consolidação de servidores, economia no consumo de energia, e gerenciamento facilitado [Pearce et al., 2013]. Neste contexto, um usuário executa sua aplicação sobre

um conjunto de máquinas virtuais (MVs), ou seja, ocorre a 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 o *software* na máquina virtual [Bressoud e Schneider, 1996]. Essa proposta pioneira possuía diversas restrições de natureza prática, como fraco desempenho e disponibilidade limitada da tecnologia de virtualização na época. Mais recentemente, na esteira da ressurgência da virtualização, surgiu o Remus [Cully et al., 2008], um mecanismo para replicação de máquinas virtuais baseado no hipervisor Xen [Barham et al, 2003]. O Remus encapsula a aplicação do usuário em uma MV e realiza *checkpoints* assíncronos entre o hospedeiro principal e a cópia de segurança, com uma frequência na ordem de dezenas de *checkpoints (cps)* por segundo. Quando um problema ocorre no hospedeiro principal, a MV no hospedeiro secundário é automaticamente iniciada, o que induz um baixo tempo de indisponibilidade.

O grande atrativo do uso de replicação de MVs para prover tolerância a faltas é a transparência, uma vez que as aplicações e sistemas operacionais existentes podem executar sem modificações. Entretanto, é latente a necessidade de investigar qual a sobrecarga imposta por este mecanismo nas aplicações hospedadas. Cully et al. (2008) observaram que o tempo de indisponibilidade depende do comportamento da aplicação em execução, já que o estado da MV é constantemente migrado (migração ativa) entre os hospedeiros [Clark et al., 2005]. Complementarmente, o presente trabalho destaca duas variáveis visando contribuir com esta caracterização: i) o intervalo de *checkpoint* necessário para manter o nível de confiabilidade do serviço, e ii) o consumo de recursos computacionais e de comunicação. O impacto dessas variáveis pode ser percebido na visão do usuário (tempo de execução e latência na resposta a uma solicitação) e do administrador (custo de provisionamento e largura de banda necessária). *Através de uma análise experimental em um ambiente controlado, este trabalho caracteriza o comportamento destas variáveis para um conjunto de aplicações executadas em uma infraestrutura virtualizada na qual a alta disponibilidade é garantida pelo Remus*. Mais especificamente, contribuímos demonstrando experimentalmente que não existe um intervalo entre *checkpoints* eficiente para todas as aplicações, sendo este o principal responsável pela latência percebida pelo cliente. Em nosso entendimento, este trabalho apresenta a primeira análise experimental que quantifica estas variáveis.

O restante deste artigo está organizado da seguinte forma: a Seção 2 detalha o mecanismo de replicação de MVs implementado pelo Remus. A Seção 3 apresenta a metodologia experimental e os resultados obtidos. A Seção 4 discute os trabalhos relacionados. Por fim, a Seção 5 conclui o artigo.

2. O Mecanismo de Replicação Remus

Remus é um mecanismo para replicação de MVs que visa fornecer alta disponibilidade a um baixo custo. Basicamente, Remus efetua uma migração constante de uma MV entre hospedeiros físicos, de forma transparente ao usuário final. Quando uma falta ocorre no hospedeiro principal (faltas como de *hardware*, rede, queda de energia, entre outros), o mecanismo ativa a MV secundária sem perda de conexão na visão do serviço hospedado na MV. O Remus é totalmente agnóstico em relação ao sistema operacional em execução na MV, o qual é executado sem precisar de configuração adicional.

Nativo nas versões atuais do MMV Xen [Barham et al., 2003], Remus utiliza a técnica de replicar o estado/disco da MV entre hospedeiros físicos a uma alta frequência (dezenas de *cps*/*s*). Cada transferência contém a variação das informações da MV desde o último *checkpoint*, ou seja, mantém o estado de conexões e monitora as transações em memória e disco, bem como o estado atual da CPU virtual. O Remus segue o modelo de replicação passiva ou primário-*backup* [Budhiraja et al, 1993]: normalmente, os clientes interagem com a MV hospedada no primário, a qual é replicada no hospedeiro *backup*. Caso ocorra a falha do primário, a MV no *backup* passa a responder pelo serviço. No Remus, os papéis de primário e *backup* são estáticos: quando um primário que havia falhado é restaurado, ele é atualizado a partir do *backup* e reassume o papel de primário. Por definição, Remus oferece alta disponibilidade para MVs sujeitas a faltas de parada (*crash*) e omissão de recursos virtuais ou físicos; em caso de parada simultânea do primário e do *backup*, as MVs são deixadas em um estado consistente.

O mecanismo de replicação pode ser visto como uma versão modificada do procedimento para migração ativa (*live migration)* [Clark et al., 2005], nativamente oferecido pelo Xen. A migração permite mover uma MV de um hospedeiro para outro, sem reiniciá-la, com um pequeno tempo de inatividade. Este procedimento é comumente utilizado para facilitar o gerenciamento e o balanceamento de carga em ambientes virtualizados, mas não oferece alta disponibilidade. Há semelhanças nos processos de replicação e migração. O *live migration* realiza uma transferência inicial da memória da MV em execução e, posteriormente, realiza cópias iterativas, copiando somente o que foi alterado na memória desde a cópia anterior. Esse ciclo, que ocorre diversas vezes, é repetido até que exista uma quantidade pequena de memória a ser transferida. Neste momento, ocorre a etapa chamada de *stop-and-copy*: a MV, que ainda está em execução no hospedeiro original, é parada definitivamente, e todo o restante da memória ainda não migrada é então transferido para o hospedeiro de destino. Na próxima fase a MV é finalmente iniciada no novo hospedeiro e uma resposta ARP não solicitada [Stevens, 1994] é difundida na rede local, permitindo que ela assuma o endereço IP da MV no hospedeiro original. *Live migration* e Remus confiam em protocolos de camadas superiores para reenvio de tráfego de rede eventualmente perdido durante esse processo.

O processo de replicação do Remus, baseado em *live migration*, é composto por vários *cps*/*s*, sendo que cada *checkpoint* realiza um procedimento *stop-and-copy*. Mais especificamente, a cada *checkpoint* a MV sofre uma pausa e o conteúdo da memória é transferido não diretamente para o hospedeiro de *backup*, mas para um *buffer* local. Em seguida, a MV retoma sua execução, e o *buffer* é então transferido para o hospedeiro de *backup*. Neste momento, o estado de execução na MV é especulativo no hospedeiro primário: a MV permanece em execução, porém o tráfego de rede que sai da MV (presumivelmente contendo respostas às requisições dos clientes) fica retido em outro *buffer*, o de rede. Somente depois que o hospedeiro de *backup* confirmar o recebimento dos dados (ou seja, o *buffer* de memória foi totalmente recebido e o disco está sincronizado) é que o tráfego de saída é liberado para deixar o *buffer* de rede. A Figura 1 ilustra este cenário. As requisições (RX) enviadas pelo cliente sempre são recebidas pela MV, enquanto as respostas (TX) são enviadas somente após a confirmação de que as réplicas estão em um estado consistente, sincronizadas entre os dois hospedeiros. É importante ressaltar que os dados temporariamente armazenados nos *buffers* local e de

rede serão descartados pelo hospedeiro *backup* caso ocorra uma falha no primário antes do término de um *checkpoint*.

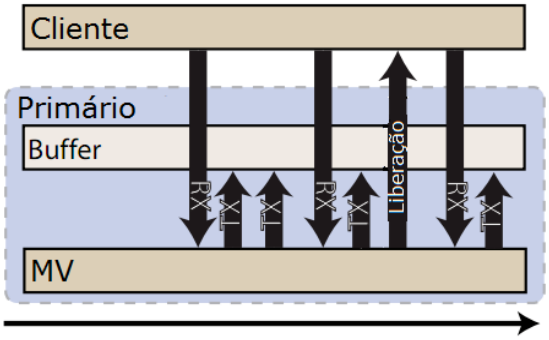

Figura 1. Funcionamento do *buffer* de rede. Adaptado de [Cully et al., 2008].

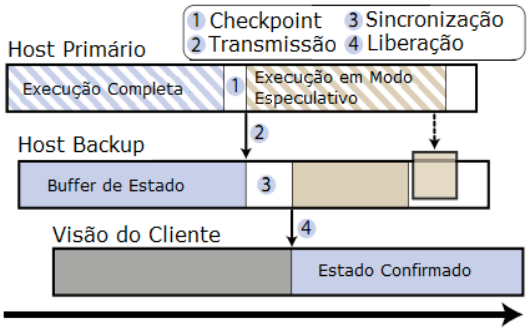

Figura 2. Modo especulativo e assíncrono. Adaptado de [Cully et al., 2008].

O procedimento de replicação alterna o estado de execução da MV entre especulativo e completo. A Figura 2 ilustra este processo: a interação com o cliente ocorre sempre de uma forma atrasada. Desta forma, o mecanismo garante que a memória e o disco da MV estão em estado consistente nos hospedeiros primário e de *backup*. A MV no hospedeiro *backup* mantém-se constantemente em estado de pausa (com execução suspensa), e somente executa se houver uma falha do primário. Isso difere da proposta de Bressoud e Schneider. (1996), onde o *backup* está sempre ativo, executando as mesmas instruções do primário (o que dificulta a garantia do determinismo das réplicas).

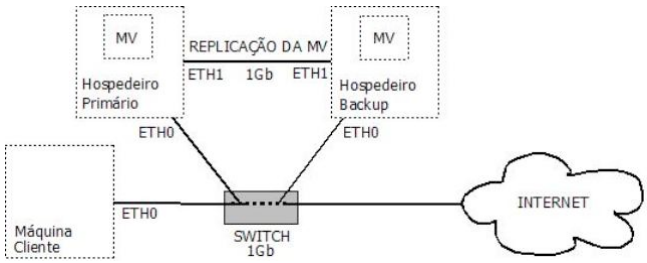

Figura 3. Remus em execução em uma rede local.

A replicação do disco é auxiliada pela ferramenta DRBD [Reisner e Ellenberg, 2005], configurada nos dois hospedeiros. O DRBD fornece um dispositivo virtual para replicação de blocos de disco, atuando entre a área de escalonamento de E/S e a camada de rede, sem conhecimento ou interação com o sistema de arquivos do disco. Na prática, ele oferece espelhamento de discos (RAID-1) via rede. Remus estendeu o protocolo utilizado pelo DRBD para suportar replicação com *cps* assíncronos. Sob Remus, DRBD opera em modo *dual-primary*, permitindo que o mecanismo de Remus decida qual hospedeiro é o primário.

A Figura 3 ilustra o funcionamento do Remus em uma rede local. Neste cenário, quando o primário falha, imediatamente o *backup* ativa a sua cópia da MV, atualizada até o último *checkpoint*. O *backup* detecta a falha do primário por *timeout* na recepção de *cps*; o primário, por sua vez, detecta a falha do backup por *timeout* no reconhecimento de *cps* (quando a falha do *backup* é detectada, o primário desativa a replicação). Assim, os *cps* e reconhecimentos funcionam como mensagens de *heartbeat*.

Após o problema com o hospedeiro primário ser resolvido, o cenário pode retornar à sua configuração inicial, migrando a MV do *backup* para o primário, reestabelecendo o mecanismo de proteção.

3. Análise experimental

Os *benchmarks* executados na análise experimental de Remus [Cully et al., 2008] 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 qual o intervalo de *checkpointing* apropriado para aplicações interativas e não interativas. Especificamente considerando aplicações interativas (sensíveis a latência), os autores sugerem a possível existência de sobrecarga, mas não quantificaram esta afirmação. Ainda, é importante ressaltar que algumas das otimizações listadas como trabalhos futuros no artigo original já foram implementadas (*e.g.*, compressão de páginas), o que sugere uma alteração (menor fluxo de dados entre primário e *backup*) no mecanismo de replicação. Mesmo que a compressão de dados resulte em um maior uso da CPU, ela ocorre no hospedeiro, não na MV protegida. Assim, analisamos o impacto no desempenho do uso do mecanismo de replicação Remus em sua versão atual, dividindo os testes em dois cenários: A) serviços executando isoladamente em uma MV; e B) e serviços que mantêm uma conexão permanente entre a MV e clientes externos.

Cenário A – Nesse caso, a sobrecarga imposta pela necessidade de parar e reiniciar a MV constantemente é relevante. Analisamos o tempo de execução de testes com operações específicas sobre a memória, dispositivos de E/S e CPU. Além da medição de desempenho, o fluxo de rede entre os hospedeiros físicos foi monitorado para identificar a utilização de largura de banda. Em suma, *o objetivo deste experimento é investigar o impacto causado pelo Remus considerando diferentes intervalos de checkpointing*. Inicialmente a ferramenta de *benchmark* Sysbench[1] foi utilizada para gerar cargas de teste para memória, disco e operações de entrada e saída. Em um segundo momento, um processo de compilação foi executado.

Cenário B – *Esse experimento investiga o impacto da replicação na latência observada pelos clientes de serviços em execução na MV*. Foram executados testes para descobrir qual o intervalo de *checkpoint/s* que causa um menor impacto à MV considerando aplicações com conexões TCP de longa duração.

3.1 Ambiente de testes

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 500GB SATA, executando o sistema operacional Ubuntu 12.10 (*kernel* 3.5.0-17-generic), e interconectados em uma rede local, conforme descrito pela Figura 3. O equipamento identificado como "Máquina Cliente" foi utilizado com origem das requisições para o cenário B. Os hospedeiros primário e de *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

[1] Disponível em http://sysbench.sourceforge.net.

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).

Para todos os testes realizados, a frequência de *checkpointing* foi definida em 10, 20, 30 ou 40 *cps/s*, seguindo estudos previamente realizados [Cully et al., 2008], [Rajagopalan et al., 2012]. Para cada experimento foram realizadas cinco rodadas, com coeficiente de variação máximo (entre todos os experimentos) de 6,4%.

3.2 Cenário A: CPU

Para investigar a sobrecarga do mecanismo de replicação em aplicações *CPU-bound*, utilizamos o *benchmark sysbench*, 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).

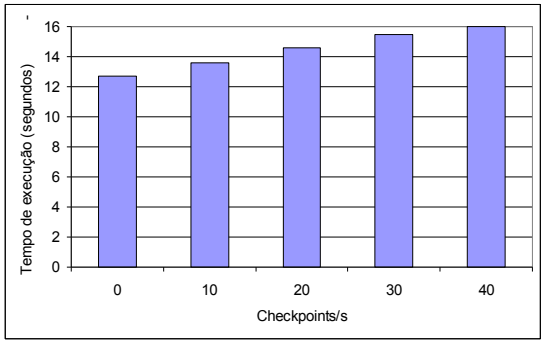

Figura 4. Tempo de execução do *benchmark* de CPU.

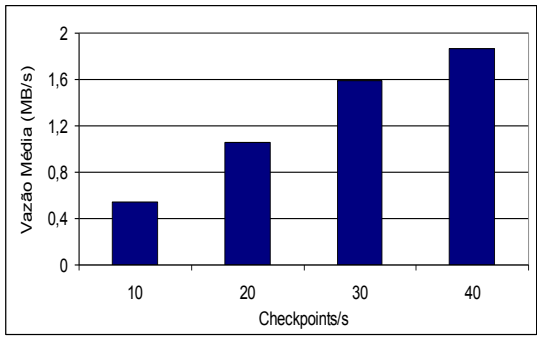

Figura 5. Vazão média de dados entre os hospedeiros durante a execução do *benchmark* de CPU.

Como mostra a Figura 4, o *overhead* causado pela replicação variou pouco em relação ao número de *cps*. Enquanto sem replicação o tempo foi de 12,7 s, com 10 *cps/s* foi de 13,6 s, resultando em 7% de sobrecarga no desempenho da MV. No pior caso, com 40 *cps/s*, a sobrecarga foi de 25,9%. Ou seja, conforme já observado por Cully et al. (2008), uma maior frequência de *cps* induz uma maior sobrecarga no desempenho de aplicações *CPU-bound*. Complementarmente, a Figura 5 ilustra a média da vazão de dados entre os hospedeiros primário e *backup* durante a replicação. Percebe-se que quanto maior o intervalo entre *cps*, melhor o desempenho (Fig. 4), e menor o fluxo de dados durante a replicação (Fig. 5). A baixa vazão é reflexo da otimização de compressão de dados de *checkpoint* implementada nas versões atuais do Remus.

3.3 Cenário A: memória

Este teste simula uma aplicação com uso intensivo de operações na memória principal. Para tal, configuramos o módulo de memória do *benchmark* sysbench para realizar escrita em memória até 10 GB, como blocos de 1 KB. Os resultados são comparados com uma execução sem Remus (barra rotulada com 0 *cps/s*).

Como mostra a Figura 6, a sobrecarga causada pelo Remus, em seu melhor resultado (com 10 *cps/s*), foi superior a 100% em relação à execução em uma MV desprotegida. É fato que esta sobrecarga pode ser um fator impeditivo para este tipo de aplicações. Novamente, observa-se que, quanto maior o intervalo entre *cps*, menor a sobrecarga, e

menor o fluxo de dados entre os hospedeiros (Figura 7). A sobrecarga é justificada pela natureza do mecanismo de replicação, que transfere, constantemente, todas páginas de memória alteradas desde o último *checkpoint* confirmado.

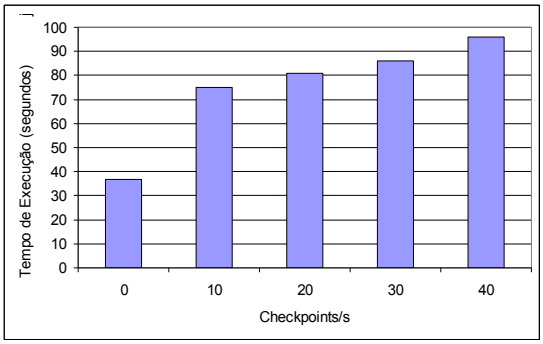

Figura 6. Tempo de execução do *benchmark* de memória.

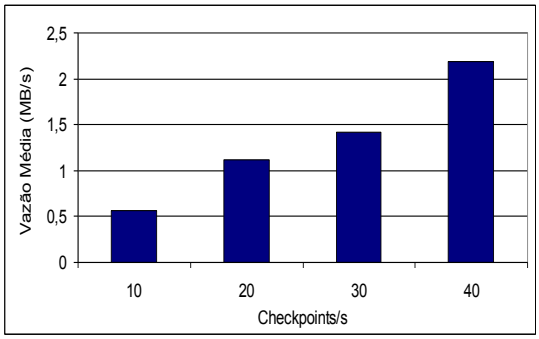

Figura 7. Vazão média de dados entre os hospedeiros durante a execução do *benchmark* de memória.

3.4 Cenário A: operações de entrada e saída

O teste executado realiza operações de leitura e escrita, aleatoriamente, no disco local da MV, sincronizado pelo mecanismo DRBD. Assim como nos testes anteriores, quanto menor a frequência de *checkpointing*, melhor o desempenho da MV (Figura 8) e menor o fluxo de dados transmitido entre os hospedeiros primário e *backup*, como mostra a Figura 9. O pior caso identificado foi com 30 *cps/s,* sendo 28,7% a sobrecarga observada (a pequena variação observada entre 30 e 40 *cps/s* segue o comportamento observado em alguns experimentos originais do Remus [Cully et al., 2008]). Se comparado com o valor base (sem Remus) e sobretudo com o *benchmark* de memória, onde o melhor intervalo de *cps/s* causou uma sobrecarga superior a 100%, observa-se que a sobrecarga de operações de E/S é tolerável para este tipo de aplicação. A vazão de dados entre os hospedeiros (Figura 9) segue o padrão dos demais experimentos.

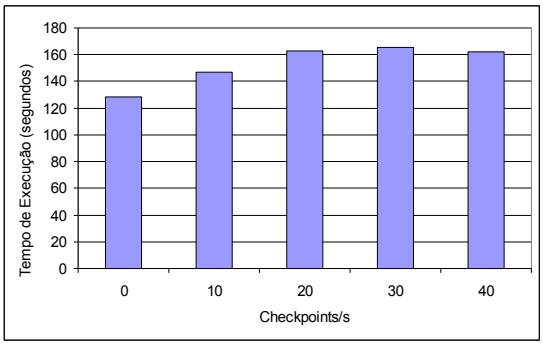

Figura 8. Tempo de execução do *benchmark* de operações de entrada e saída.

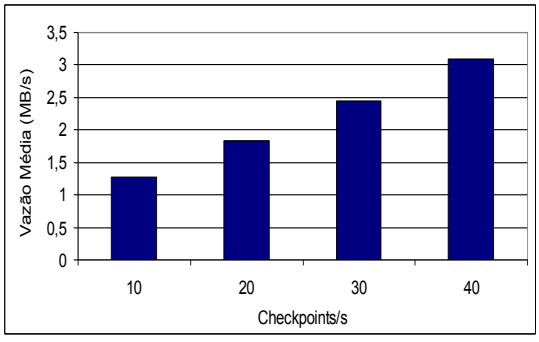

Figura 9. Vazão média de dados entre os hospedeiros durante a execução do *benchmark* de E/S.

3.5 Cenário A: compilação

Neste experimento efetuamos a compilação do BIND versão 9.9.4.p2[2] sobre uma MV replicada. Este processo envolve uso de CPU, memória, acesso a disco e rede (tráfego entre hospedeiros), ou seja, é uma combinação dos cenários individualmente analisados nas seções anteriores. Durante a compilação, 360 novos arquivos serão criados, totalizando 235MB, que deverão ser replicados para o *backup*. A Figura 10 apresenta o tempo de execução enquanto a Figura 11 mostra a vazão média de dados transferidos entre os hospedeiros.

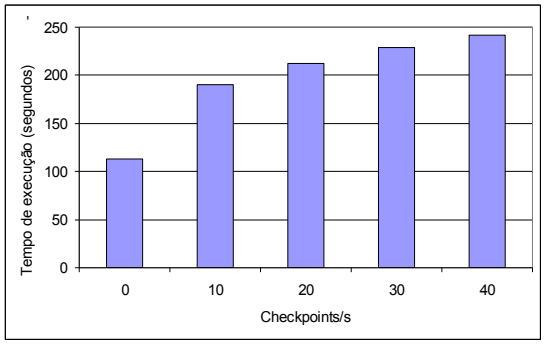

Figura 10. Tempo de execução da compilação do BIND.

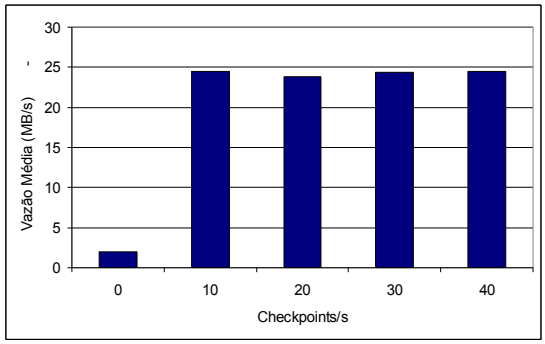

Figura 11. Média da vazão de dados entre os hospedeiros primário e *backup* durante a compilação

A sobrecarga imposta pela replicação foi de 68% para o melhor caso (10 *cps/s*). A sobrecarga identificada segue o comportamento dos testes anteriores, mas percebe-se que, embora a compilação seja uma aplicação com carga combinada de CPU, memória e entrada/saída, a memória é o fator limitante. Especificamente para este cenário combinado, observa-se que o consumo de rede entre primário e *backup* foi semelhante para todos os intervalos de *checkpoints* (Figura 11).

3.6 Cenário B: transferência de dados via rede

Este experimento avalia a latência percebida por um cliente conectado a uma MV protegida por Remus. Foi usada a ferramenta Netcat[3] para minimizar a sobrecarga de aplicação que pudesse influenciar nos resultados, já que o Netcat apenas envia os dados por um *socket* TCP, sem um protocolo de aplicação. Medimos o tempo necessário para cada transferência de 50 MB entre o cliente e a MV, sob diferentes intervalos de *checkpoints*. Testes de *upload* representam transferências do cliente para a MV, enquanto *download* representa o sentido oposto. Desta forma, caracterizamos o impacto causado pelo *buffer* de rede e pela execução da MV em modo especulativo.

[2] Disponível em https://www.isc.org/downloads/bind/

[3] Disponível em http://netcat.sourceforge.net

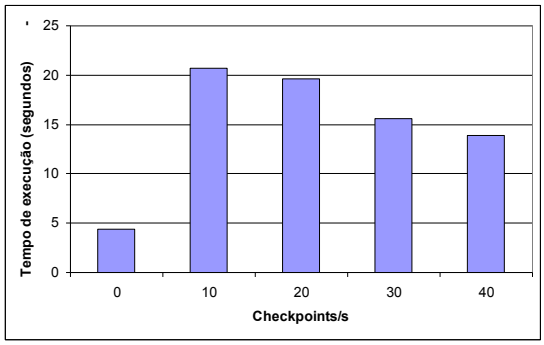

Figura 12. Tempo de transferência de um arquivo de 50 MB do cliente para a MV (*upload*).

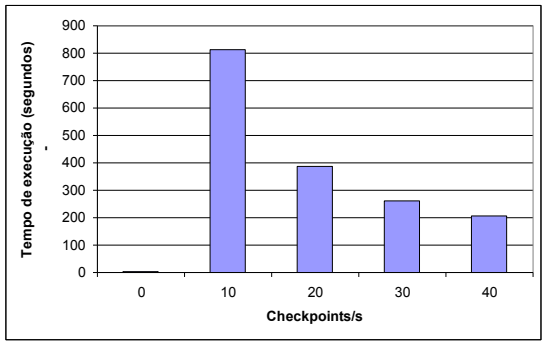

Figura 13. Tempo de transferência de um arquivo de 50 MB da MV para o cliente (*download*).

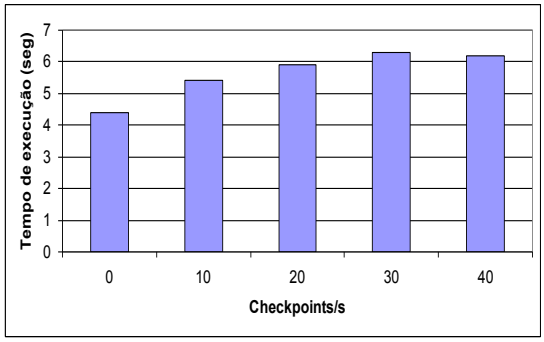

Figura 14. Tempo de transferência de um arquivo de 50 MB para a MV (*upload*), sem o *buffer* de rede ativo.

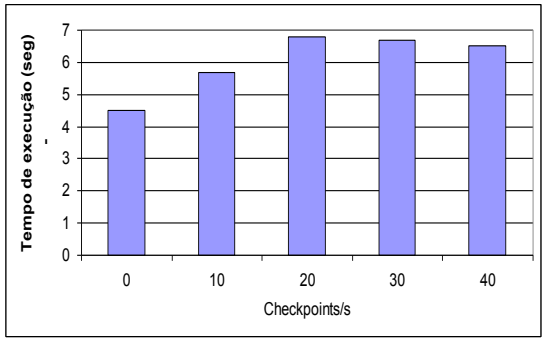

Figura 15. Tempo de transferência de um arquivo de 50 MB partindo da MV para o hospedeiro cliente (*download*), sem o *buffer* de rede ativo.

Ao contrário do que mostram os testes anteriores (cenário A), onde não há percepção de latência, a sobrecarga maior sobre a MV se dá com maiores intervalos de *cps*/*s*. A Figura 12 mostra que no melhor resultado, 40 *cps*/*s*, a sobrecarga é de 215% em relação à mesma transferência para uma MV sem proteção. No segundo caso, onde a transferência parte da MV (Figura 13), a menor sobrecarga é de aproximadamente 4500% (4,5 s sem Remus, contra 202 s com 40 *cps*/s). No pior caso, utilizando uma replicação de 10 *cps*/*s*, a sobrecarga está próxima a 18.000%. Novamente, quanto mais *cps*/*s*, menor a latência percebida pelo cliente. Porém, mesmo no melhor caso, o impacto no desempenho de *download* é acentuado, podendo inviabilizar o uso do Remus.

A diferença da latência percebida entre os dois testes (*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 há é um bloqueio às respostas das requisições até que o período de *checkpoint* esteja completo. Não haverá transmissão de dados partindo da MV enquanto o hospedeiro *backup* não devolver uma confirmação de recebimento completo relativo ao *checkpoint* em questão.

Para fins de análise de desempenho, Remus permite a execução sem o *buffer* de rede ativo (o que reduz o nível de proteção). As Figuras 14 e 15 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 13,9 s, sem o *buffer* foi de 6,2 s com 40 *cps*/*s*. No caso de *download*, também a 40 *cps*/*s*, o tempo necessário é 202 s, contra 6,5 s sem o *buffer*. Neste último caso, pode-se afirmar que o *buffer* de rede foi responsável em aumentar a latência percebida pelo cliente em mais de 3.000%. Outra característica dos testes sem *buffer* de rede ativo é que o melhor desempenho na execução das MVs se dá com intervalos maiores entre *checkpoints*, a exemplo do que ocorre com aplicações não sensíveis a latência (cenário A). 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 em caso de ocorrência de faltas.

Em suma, este experimento indica que o *buffer* de rede é o principal responsável pela latência percebida pelo cliente quando o Remus está em execução. O tempo necessário para transferir o *buffer* local para o hospedeiro *backup*, somado ao tempo em que a MV fica suspensa até que os dados sejam transferidos para o *buffer* local, são as principais causas. Independente do intervalo de *cps*, a latência percebida pelo cliente é sempre alta, principalmente quando é a MV a origem das transferências. *Checkpoints* mais frequentes, como 40 *cps*/*s*, fazem com que o *buffer* a ser transferido entre primário e *backup* seja menor, o que leva a uma resposta mais rápida às requisições solicitadas pelos clientes. Porém, frequências de *checkpointing* menores resultam em menos suspensões, deixando a MV livre para processamento por mais tempo (conforme discutido nos experimentos do cenário A).

3.7 Cenário B: *Benchmark* ab

Este experimento avalia o desempenho de um servidor web Apache[4] versão 2.2.22 (com a configuração padrão do OpenSUSE) executando na MV, atendendo a 100 conexões simultâneas durante 60 segundos. O arquivo a ser lido constantemente pelo cliente compreende uma página de 72950 bytes (phpinfo.php).

A Figura 16 mostra o percentual das requisições respondidas por segundo. Com 5,645ms, 50% das requisições sob 40 *cps*/*s* já haviam sido respondidas, contra 14,428ms para a mesma porcentagem sob 10 *cps*/*s*. Em relação ao número de requisições atendidas, a diferença entre o intervalo de *checkpoints* é ainda maior. No pior caso, com 10 *cps*/*s*, o servidor web respondeu a apenas 53 requisições durante o período. No melhor caso, com 40 *cps*/*s*, 200 requisições foram atendidas. Porém, se comparado ao teste sem Remus, no qual o servidor respondeu a 9549 requisições durante os mesmos 60s, houve uma sobrecarga no desempenho superior a 4.600% conforme mostra a Figura 17. A média da vazão de dados entre o primário e o *backup* (Figura 19) variou pouco em relação a 40 ou 30 *cps*/*s*. Em suma, os resultados mostram que quanto menor o intervalo de *checkpoints*, melhor o desempenho, menor a latência e maior o fluxo de rede utilizado para a replicação.

[4] Disponível em http://www.apache.org.

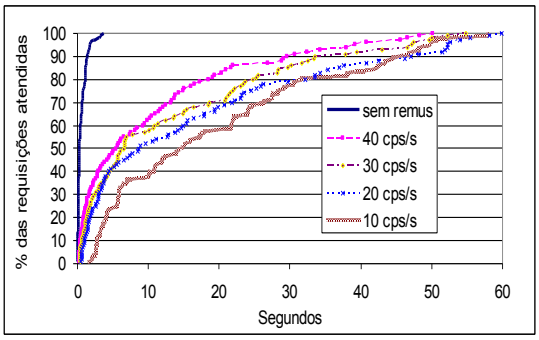

Figura 16. Percentual das requisições atendidas em 60 segundos.

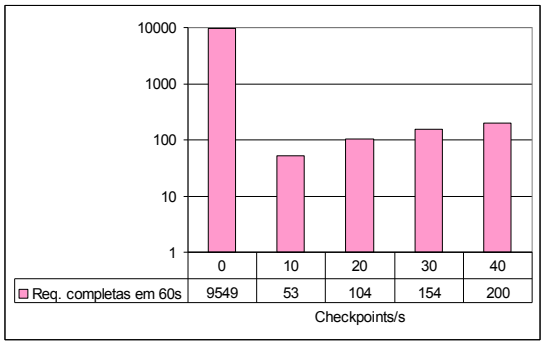

Figura 17. Número de requisições atendidas em 60s sob diferentes *cps/s.*

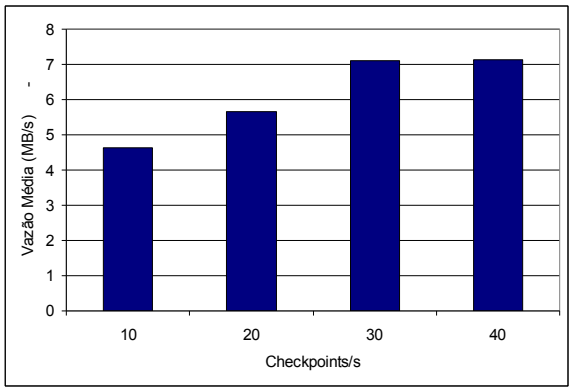

Figura 18. Média da vazão de dados entre primário e *backup* durante a execução do *benchmark* ab.

Nas operações em que não há uma percepção de latência, como no caso da compilação, a média de tráfego entre primário e *backup* se mantém constante, mesmo com diferentes intervalos de *checkpoints*. Em relação ao *benchmark ab*, onde há percepção de latência, tal característica não se repete.

3.8 Cenário B: *Benchmark* tbench

Este experimento analisa o comportamento de um servidor de arquivos replicado com Remus. Para tal, o *benchmark* tbench[5] simula as operações comumente executadas em um servidor de arquivos SMB (*Server Message Block*). 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 I apresenta o número de operações executadas e a latência observada pelo usuário do serviço.

[5] Disponível em http://linux.die.net/man/1/tbench e http://dbench.samba.org.

Operação	Sem Remus		40 *cps/s*		30 *cps/s*		20 *cps/s*		10 *cps/s*	
	ops	lat (ms)	ops	lat (ms)	ops	lat (ms)	ops	lat (ms)	ops	lat (ms)
Close	181968	3,08	5755	26,36	4105	35,19	2404	52,09	2447	101,83
Rename	10504	3,25	293	26,83	220	33,07	122	51,11	139	101,54
Unlink	50139	3,08	1368	25,91	1062	35,52	355	51,01	321	101,24
Find	86873	3,34	2606	51,44	1936	56,56	1188	84,45	944	133,18
Write	122390	5,55	3742	45,46	2724	59,54	1643	83,18	1713	151,54
Read	388191	6,13	11525	438,11	8891	575,19	5975	889,59	3536	1309,33
Flush	17376	3,57	491	25,93	370	37,11	150	50,82	298	102,05

Tabela I – Número de operações executadas em um servidor SMB durante 600 segundos e a média da latência percebida pelo cliente para cada operação.

Assim como no experimento da Seção 3.7, a latência aumenta conforme aumenta o intervalo entre *checkpoints*. A Figura 19 mostra a média de latência para cada operação.

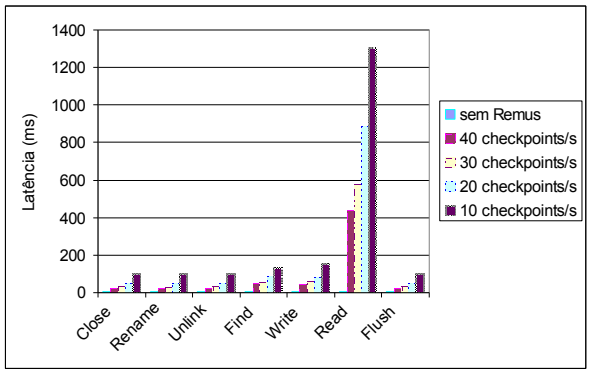

Figura 19. A latência média percebida pelo cliente para cada operação de rede em um servidor de arquivos.

4. Trabalhos relacionados

A migração e replicação de máquinas virtuais são temas promissores, já abordados em outros contextos. Enquanto algumas soluções preocupam-se com ambiente LAN, outras são voltadas para WAN, onde a instabilidade e alta latência são grandes desafios. Além disso, nem todas abordagens objetivam alta disponibilidade: o processo de migração pode ser realizado por motivos de desempenho ou simplesmente manutenção de um hospedeiro físico. Por exemplo, o mecanismo nativo de *live migration* [Clark et. al, 2005] implementado pelo hipervisor Xen atua na migração de uma MV em uma rede local. O conteúdo da memória e o estado da CPU virtual são transferidos para outro hospedeiro sem que haja perda de conexão entre cliente e serviços hospedados na MV, ou seja, não ocorre um alto intervalo de indisponibilidade. Como não há replicação do disco, o mesmo deve ser compartilhado ou copiado para o hospedeiro físico de destino antes da migração. CloudNet [Wood et al, 2011] propõe a migração de MVs sobre uma WAN buscando um balanceamento de carga. A tomada de decisão para tal migração é motivada pela latência observada pelo cliente final e/ou limites de capacidade. A replicação de disco fica a cargo do DRBD [Reisner e Ellenberg, 2005].

O mecanismo proposto pelo SecondSite [Rajagopalan et. al, 2012] utiliza as mesmas técnicas de replicação do Remus, como migração assíncrona e *checkpoints*, diferenciando o ambiente alvo: SecondSite propõe a replicação completa de um *site* entre *datacenters* geograficamente distribuídos. Para tal, utiliza um servidor externo para detecção de falhas, responsável por ativar o *site* secundário quando necessário. O objetivo é migrar as MVs em caso de catástrofes naturais, faltas de *hardware*, *links* ou energia, para outro *datacenter*. Nesta solução, a manutenção da conectividade das MVs é obtida através de configurações de roteamento previamente informadas nos roteadores BGP sob um único Sistema Autonômo. Assim como Remus, SecondSite utiliza o DRBD para replicar o disco, de uma forma assíncrona.

Considerando a implementação destes mecanismos, existe uma crescente busca em relação ao desempenho final. Por exemplo, *live migration* migra apenas o que foi alterado na memória em relação à rodada anterior. Este processo, conhecido como cópia iterativa, inicia-se logo após uma cópia inteira da memória ser transmitida para o hospedeiro *backup*. Se na fase *stop-and-copy* ainda restar um número grande de dados na memória a serem copiados, a MV poderá ficar indisponível por um longo período. SecondSite e Remus, além da mesma técnica utilizada pelo *live migration* (replicar a cada *checkpoint* apenas o que foi alterado na memória em relação ao *checkpoint* anterior), também utilizam outras duas técnicas para diminuir a sobrecarga causada pela constante replicação: compressão do tamanho do *checkpoint* e utilização de um *buffer* local para armazenar o estado da MV antes de transferi-lo. CloudNet, por sua vez, utiliza *live migration* para migrar uma MV em um ambiente WAN. Para diminuir, ou otimizar, o número de rodadas necessárias para a transferência, localiza dados repetidos na memória e no disco, transferindo-os apenas uma vez para o hospedeiro de destino. Os estudos para diminuir a indisponibilidade e a sobrecarga estão sempre voltados à quantidade de dados transferida do hospedeiro primário para o *backup*, mas na replicação há mais um fator relevante, que é a frequência com que as cópias ocorrem. Remus replica a MV inteira para o hospedeiro *backup* a uma taxa de vários *cps*/*s*. Como demonstrado na Seção 3, a frequência de *checkpoints* influencia no desempenho da MV, já que cada *checkpoint* representa um período de indisponibilidade.

Especificamente considerando o desempenho das aplicações hospedadas nestas arquiteturas, [Cully et al, 2008] aborda um processo de compilação como *benchmark* para avaliar o desempenho da MV em relação ao número de *cps*/*s*, porém não há estudos isolados em relação a CPU, memória e E/S. Ainda, os autores simularam a percepção de um usuário através do *benchmark* SPECweb, não considerando aplicações interativas como abordado em nossa análise experimental. [Magalhães et al, 2011] analisaram o desempenho de dois tipos de migração de MVs hospedadas em Xen: *stop-and-copy* e pré-cópia. Em sua análise, observaram que o tempo de indisponibilidade é diretamente relacionado com o tamanho da memória ocupada pela MV e com o tipo da aplicação em execução. Esta situação é semelhante no contexto de nosso trabalho dada a proximidade existente entre o Remus e o mecanismo de *live migration*.

5. Conclusão

A análise dos resultados mostra que o mecanismo de replicação Remus induz uma sobrecarga não desprezível quanto ao desempenho das aplicações hospedadas nas MVs protegidas. Em relação a CPU, memória e disco é preferível um intervalo maior entre

checkpoints, ou seja, menos *checkpoints* por segundo, porém o oposto pode ser dito em relação a aplicações sensíveis a latência, como um servidor web ou de arquivos. Sobretudo, aplicações com uso intensivo de operações em memória tendem a sofrer um maior impacto devido à natureza do mecanismo de replicação. A vazão média de replicação entre primário e *backup* varia conforme a natureza das operações executadas na MV e a freqüência de *checkpoints*. Quanto maior o intervalo entre *checkpoints*, menor a vazão média, pois o processo de compressão do *checkpoint* opera com *buffers* maiores, em menor quantidade, diminuindo o volume de dados que são transferidos.

Interromper uma MV frequentemente é o principal responsável pela sobrecarga identificada nas aplicações que não respondem a requisições externas. Por outro lado, nas aplicações onde há uma percepção de latência, além dos *checkpoints*, há também o fato da MV estar constantemente em modo de execução especulativo, onde as requisições somente são respondidas e entregues ao requisitante após o hospedeiro *backup* acolher completamente o *checkpoint* anterior. Nesse cenário, observa-se que a replicação de MVs utilizando o mecanismo de proteção Remus pode comprometer o desempenho final da aplicação hospedada.

Referências

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. “Xen and the art of virtualization”. Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP), 2003.

Bressoud, T. C., and Schneider, F. B. “Hypervisor-based fault tolerance”. *ACM Trans. on Computer Systems*, 14(1), 80-107, 1996.

Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. “The primary-backup approach”. In *Distributed Systems (2nd Ed.)*, Cap. 8. Sape Mullender (Ed.), ACM Press, 1993.

Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. “Live migration of virtual machines”. Proc. of the 2nd USENIX Symp. on Networked Systems Design and Implementation (NSDI), 2005.

Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. “Remus: High availability via asynchronous virtual machine replication”. Proc. of the 5th USENIX Symp. on Networked Systems Design and Implementation (NSDI), 2008.

Magalhães, D. M. V.; Soares, J. M.; Gomes, D. G. “Análise do Impacto de Migração de Máquinas Virtuais em Ambiente Computacional Virtualizado”. Anais do XXIX SBRC, 2011.

Pearce, M., Zeadally, Sherali., and Hunt, R. “Virtualization: Issues, security threats, and solutions”. *ACM Computing Surveys*, 45(2), 2013.

Rajagopalan, S., Cully, B., O’Connor, R., and Warfield, A. “Secondsite: Disaster tolerance as a service”. Proc. of the 8th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments (VEE), 2012.

Reisner, P., and Ellenberg, L. “DRBD v8 – replicated storage with shared disk semantics”. Proc. of the 12th International Linux System Technology Conference, 2005.

Stevens, W. R. *TCP/Illustrated, Vol. 1: The Protocols*. Addison-Wesley, 1994.

Wood, T., Ramakrishnan, K. K., Shenoy, P., and van der Merwe, J. “Cloudnet: Dynamic pooling of cloud resources by live WAN migration of virtual machines”. Proc. of the 7th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments (VEE), 2011.