O USO DE VIRTUALIZAÇÃO NA CONSTRUÇÃO DE SERVIÇOS DE STREAMING ESCALÁVEIS

Lauro Luis Armondi Whately

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Maria Clícia Stelling de Castro, D.Sc.

Prof. Noemi de La Rocque Rodriguez, D.Sc.

Prof. Eugene Francis Vinod Rebello, Ph.D.

RIO DE JANEIRO – RJ, BRASIL

JULHO DE 2008

WHATELY, LAURO LUIS ARMONDI

O Uso de Virtualização na Construção de Serviços de Streaming Escaláveis [Rio de Janeiro] 2008

XI, 68 p. 29, 7cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 2008)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1. Streaming
2. Monitor de Máquinas Virtuais
3. Serviços Web Escaláveis
4. Gerenciamento de Recursos
5. Sistema Operacional

I. COPPE/UFRJ II. Título (série)

Para Isabela e João

Agradecimentos

Aos meus pais, os primeiros e maiores agradecimentos pelo esforço que sempre fizeram para que eu pudesse estar, hoje, escrevendo estas linhas.

Quero agradecer ao meu orientador Claudio Luis de Amorim, pela dedicação e esforço para que seus alunos sempre realizem bons trabalhos. É constante sua luta para manter o Laboratório de Computação Paralela sempre vibrante e cheios de projetos. Espero que muitas pessoas possam ter o mesmo prazer que tenho em trabalhar no LCP até hoje.

Esta tese representa para mim mais do que apenas o período que passei trabalhando em seus problemas. Representa um período muito rico de experiências que se iniciou quando vim trabalhar no Laboratório de Computação Paralela. Só tenho a agradecer o convívio que tive com todos que estiveram de alguma forma envolvidos com o LCP. Em especial, a amizade do Leonardo Bidese Pinho é muito gratificante.

No decorrer do desenvolvimento da tese, surgiram idéias para outros trabalhos e alguns resolveram se arriscar comigo. Devo muito ao Diego Dutra, sua colaboração na implementação dos experimentos foi ímpar. Arthur Granado, Almir Fernandes e Daniel Schmidt também participaram de muitas discussões. Obrigado pessoal!

Também não posso esquecer de agradecer ao João, à Isabela e à Vera. João e Isabela cederam muitas horas de brincadeiras no computador para o papai. Vera, te agradeço muito por todo o apoio nos momentos mais difíceis.

Agradeço às seguintes instituições pelo suporte financeiro no período desse trabalho: à CAPES e o Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ pela bolsa de doutorado e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), pela bolsa de Desenvolvimento Tecnológico e Inovação, nos últimos anos.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

O USO DE VIRTUALIZAÇÃO NA CONSTRUÇÃO DE SERVIÇOS DE STREAMING ESCALÁVEIS

Lauro Luis Armondi Whately

Julho/2008

Orientador: Claudio Luis de Amorim
Programa: Engenharia de Sistemas e Computação

Os últimos anos foram marcados pelo surgimento do modelo de computação orientada por serviços na Web, como simulações virtuais de ambientes 3D e vídeo-sob-demanda. Estes serviços web trouxeram novas demandas de desempenho, distintas das aplicações tradicionais, como serviços de pesquisa e mapas. Por exemplo, servidores de jogos precisam de tempos de resposta pequenos para que possam apresentar bom desempenho de interatividade, e serviços de transmissão de vídeo requerem garantias de desempenho de tempo-real. Além disso, estes serviços precisam prover garantias de confiabilidade e escalabilidade para seus usuários. Entretanto, as técnicas existentes para a implementação de serviços com desempenho de tempo-real são significativamente ineficientes. A tese demonstra através de experimentos em um estudo de caso que usando os mecanismos de virtualização de um servidor é possível construir uma plataforma escalável e confiável, baseada em cluster de computadores, que garanta probabilisticamente o compartilhamento global dos recursos com QoS por muitos serviços de *streaming*.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

USING VIRTUALIZATION TO BUILD SCALABLE STREAMING SERVICES

Lauro Luis Armondi Whately

July/2008

Advisor: Claudio Luis de Amorim

Department: Systems Engineering and Computer Science

Recent years have been marked by the appearance of the model of computing driven by Web services, such as simulations of virtual 3D environments and video-on-demand. These web services have brought new performance requirements, distinct from traditional applications such as maps and search services. For instance, game servers need good interactive performance and thus low average response times, and streaming media services require real-time performance guarantees. Moreover, these services still need provide guarantees of reliability and scalability to its users. However, the existing techniques for hosting cluster-based scalable real-time web services are significantly inefficient. The thesis demonstrates through experiments in a case study that using mechanisms of a server virtualization you can build a reliable and scalable platform, based on cluster of computers, which assures probabilistically guarantees of global sharing of resources with QoS for many streaming media services.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Rápidos avanços no poder de processamento, na capacidade de armazenamento em memória e disco e na largura de banda das redes estão nos conduzindo para uma era em que teremos acesso a recursos computacionais virtualmente ilimitados e distribuídos globalmente [1, 2, 3, 4]. No modelo de computação ainda corrente, usuários tem acesso a poderosos computadores pessoais, onde a maioria do processamento é feito no hardware local ou em alguns casos em servidores em uma sala próxima. Hoje, já começamos a assistir a transferência da computação pessoal de computadores de mesa para servidores remotos na Internet. A *Web* é o melhor exemplo desta tendência, onde cada vez mais a computação em benefício do usuário é feita por servidores anônimos na Internet. Exemplos incluem serviços de busca de documentos, enciclopédias digitais *online*, simuladores financeiros e comércio eletrônico [5, 6, 7, 8, 9]. Como resultado, nosso ambiente computacional está passando por uma mudança radical. A mudança de uma visão de computação centrada no hardware para uma computação orientada por serviços [10, 11, 12, 13, 14]. Nesse ambiente, a rede poderá ser vista como um repositório de computadores virtuais escalável e confiável que estará disponível para os usuários ou provedores de serviços e que pouco precisarão se preocupar com os incômodos da administração e manutenção de um sistema [15, 16].

O melhor exemplo na *Web* para essa mudança é o crescimento acelerado de serviços de *streaming* de conteúdo multimídia na Internet [17]. Serviços de *streaming*

implicam na transmissão e na reprodução imediata[1] de vídeo e áudio de notícias, eventos esportivos, entretenimento e sítios educacionais.

Estes serviços diferem dos serviços web tradicionais em muitos aspectos, apresentando um número de desafios para desenvolvedores de sistemas e os provedores de serviços multimídia [18]. Por exemplo, o *streaming* de vídeo por um servidor requer mais capacidade do processador, largura de banda de rede, armazenamento em disco e também é mais sensível ao *jitter* na rede do que as páginas ou imagens web estáticas. Além disso, os acesso aos serviços *streaming* ocupam os recursos do sistema por um período indefinido de tempo, oposto dos pedidos de objetos web que tipicamente duram milisegundos. Além disso, estes serviços precisam prover garantias de confiabilidade e escalabilidade para seus usuários. Entretanto, as técnicas existentes para a implementação de serviços web escaláveis com desempenho de tempo-real são significativamente ineficientes [19]. Essa tese se procupa em encontrar mecanismos que possibilitam a criação de sistemas eficientes - oferecendo garantias de QoS no compartilhamento dos recursos existentes -, escaláveis e confiáveis para o *streaming* de vídeo e áudio. Na próxima seção são analisadas as soluções para a implementação de serviços web atuais e as razões de suas ineficiências para serviços de *streaming*.

1.2 Sistemas Atuais

O sucesso dos serviços web na Internet trouxe desafios para o projeto de sistemas servidores em termos de escalabilidade, confiabilidade e gerenciamento. A escalabilidade requer que o aumento de recursos de hardware e software tragam um aumento correspondente na capacidade de processamento e armazenamento no sistema. Em termos de confiabilidade, o sistema deve assegurar que as falhas ocorridas em suas operações internas não interrompam (ou apenas degradem suavemente) a execução do serviço. O gerenciamento deve permitir a reconfiguração e manutenção do sistema sem afetar o desempenho do serviço. Um modelo para a construção de sistemas escaláveis é muito bem descrito em [20].

Por ser uma solução comprovadamente custo-efetiva para obter alta confiabilidade e escalabilidade incremental, *cluster* de computadores (ou apenas, cluster)

[1]Diferente de *download*, que recebe todo o conteúdo localmente antes da reprodução, ou a transmissão em *pseudo-streaming* que envia o conteúdo em uma taxa diferente da reprodução.

[21] tornou-se uma solução reconhecida para sistemas web escaláveis, especialmente quando os sistemas devem esperar o crescimento abrupto na demanda do serviço e no número de usuários [22, 23, 24]. Mas, nos sistemas propostos inicialmente, como Porcupine [23], por serem dedicados a apenas um serviço web ou multiplexados com a granularidade de um servidor, como em SNS [22], foi dada pouca atenção para o controle de recursos voltado para a garantia de desempenho e da segurança intra-cluster.

Atualmente, as soluções para sistemas web evitam qualquer compartilhamento, usando um modelo de cluster dedicado. Por exemplo, os vários serviços (busca, correio eletrônico etc) oferecidos pelo Yahoo! [25] são hospedados em clusters dedicados para cada serviço. Em alguns casos, os serviços compartilham os recursos sem oferecer qualquer garantia para o desempenho do serviço, em um modo conhecido como "melhor-esforço". Uma rede de computadores ou serviço no modo "melhor-esforço" não suporta qualidade de serviço (QoS)[2]. Uma alternativa para mecanismos complexos de controle de QoS é sobreprovisionar a capacidade do sistema para suporte o pico de carga esperado. Um exemplo é o serviço de *streaming* que provê a ilusão que cada provedor de conteúdo possui seu próprio servidor, quando na realidade múltiplos servidores lógicos podem compartilhar um servidor físico. Enquanto o tráfego de cada serviço é baixo para a capacidade do sistema, o compartilhamento é feito no modo "melhor-esforço", sem isolamento de desempenho ou qualquer garantia de QoS para os serviços. Normalmente, a capacidade do servidor está sobreprovisionada para os serviços oferecidos (o que "mantém" o tráfego baixo). A alternativa quando o tráfego de um serviço em particular ultrapassa a capacidade provisionada no servidor, esse serviço é movido para um servidor dedicado.

Outras razões, no cenário econômico atual, reforçam o compartilhamento das máquinas por vários serviços, tais como o uso racional do espaço físico alocado, o alto custo da energia elétrica consumida pelo sistema, e a grande quantidade de calor gerado pelos servidores. O compartilhamento dos nós do cluster permite o

[2]QoS é um termo originado na indústria de telecomunicações, sendo definido como a medida coletiva de vários parâmetros para o nível de serviço entregue ao usuário. No contexto dessa tese, a atenção está voltada para os parâmetros relacionados com confiabilidade e desempenho, como degradação suave, tempo de resposta, vazão, pedidos perdidos, tempo de conexão, entre outros relacionados com serviços de *streaming* [26].

uso mais efetivo dos recursos por multiplexação e oferecem economia de escala na administração e manutenção da plataforma [27].

Partindo da necessidade de um número grande de serviços compartilharem um pequeno número de máquinas, o gerenciamento de recursos torna-se a questão central para uma solução em cluster. A habilidade para reservar recursos dinamicamente para cada serviço, a habilidade para isolar os serviços uns dos outros e a necessidade de gerenciar os requerimentos heterôgenos de desempenho são alguns dos novos desafios que devem ser enfrentados. Um problema nos sistemas operacionais (SO) correntes é que o modelo de contabilidade dos recursos não necessariamente reflete o padrão de uso real. A execução no processador contabilizada para um usuário pode ter sido consumida em um tratamento de interrupção, por exemplo. Ainda mais, um serviço pode incluir vários processos (ou *threads*), tornando assim muito difícil a contabilidade em um modelo baseado em processos. Sem um modelo acurado para a contabilidade do uso dos recursos, não é viável oferecer garantias de QoS.

Por estas razões, serviços comerciais de *streaming* não possuem um controle de admissão interno que possa prevenir a sobrecarga no servidor, ou alocar uma fração predefinida de recursos do servidor para um serviço em particular. O servidor de *streaming* não conhece qual recurso em especial deve ser monitorado para avaliar a disponibilidade (ou o uso) corrente da capacidade do sistema: a utilização do processador e memória, como também a banda da rede são altamente dependentes das características do serviço [19].

Soluções para o compartilhamento de recursos propostas até o momento, como *SODA* [28], alcançam um índice alto de uso dos recursos, mas comprometem o isolamento. Plataformas de cluster "dedicado" (por exemplo, *Oceano* [29] e *Cluster-On-Demand* [30]) por outro lado, apresentam o isolamento, mas não multiplexam - com granularidade fina no tempo -, os recursos. O uso da técnica de escalonamento *Proportional Share* foi proposta por Arpaci-Dusseau e Culler em [31] para a multiplexação de aplicações distribuídas em um cluster. Outro trabalho - Sharc [32] - estuda mecanismos de alocação de recursos através de reservas no controle de admissão e usa escalonamento *Proportional Share* para a distribuição da banda de rede.

Cluster Reservers [33] emprega *Resource Containers* [34] para isolar o consumo de recursos por serviços web em um cluster de computadores. *Resources Containers* busca resolver o problema do gerenciamento e a contabilidade acurada do uso dos recursos em servidores monoprocessados. Um *resource container* é uma nova abstração do SO, que engloba todos os recursos consumidos por uma classe de serviço. Por exemplo, para uma dada conexão HTTP associada a uma classe de serviço e gerenciada por um servidor web, os recursos incluem tempo de processador dedicado à conexão, objetos de kernel como soquetes, buffers de rede e memória reservada para dados, usados pela conexão. Podem estar associados a um *resource container*, os processos do servidor web e do CGI e a conexão do cliente, por exemplo, que representam uma classe de serviço. Um escalonador deve disponibilizar os recursos para o *resource container* e não para os processos ou *threads* possuídos. Os processos são escolhidos dentro do *container*. *Resources Containers* são voltados para o isolamento de desempenho de serviços web, onde uma classe de serviço é identificada por um pedido http ou pelo endereço fonte. Apesar de a maioria dos serviços na Internet estarem nessa categoria, no futuro devemos esperar um variedade maior de interfaces com os serviços. Estes sistemas não apresentam uma solução completa quando assumem que o perfil de uso dos recursos pelas aplicações é previamente conhecido. Além disso, praticamente não se adaptam a alguma variação inesperada da demanda no serviço, ou na ocorrência de alguma modificação (falha, entrada ou saída de um nó no cluster, por exemplo) no sistema .

Uma abordagem diferente para a garantia da QoS no compartilhamento de recursos é fazer uso de uma máquina virtual (MV). Cada serviço é executado em um ambiente computacional exclusivo, criado em um MV exclusiva. Um máquina física é multiplexada entre várias MVs. O monitor de máquina virtual (MMV) é o responsável por gerenciar todos os recursos físicos no sistema. O MMV pode contabilizar o uso do recursos (tempo de execução no processador, quantidade de memória reservada, buffers de recebimento e transmissão na rede, entre outros) para MV. Esta solução está abaixo das abstrações do SO e é mais abrangente e precisa, comparando com *Resource Containers*.

1.3 Tese

Esta tese afirma que explorando os seguintes mecanismos encontrados na virtualização de um servidor:

- a garantia de isolamento e segurança através da multiplexação fina de recursos;
- monitoramento dinâmico de recursos consumidos por múltiplas MVs;
- *checkpointing* e migração de uma máquina virtual;

é possível estender os benefícios para uma plataforma escalável e confiável, baseada em cluster de computadores, que garanta probabilisticamente o compartilhamento global dos recursos com QoS por muitos serviços de *streaming*.

Para essa plataforma, um número de questões precisam ser investigadas para a implementação de uma arquitetura escalável baseada em MMV. Os benefícios esperados são altos quando muitas MVs podem ser co-executadas em um mesmo servidor. Assim, é importante encontrar um MMV que seja escalável o suficiente para rapidamente retirar e entregar os recursos necessários para as MVs [35]. Os benefícios de multiplexação podem ser aproveitados ao máximo se as aplicações executando simultaneamente em diferentes MVs tenham requerimentos complementares, ou melhor, os picos de demanda de recursos não se sobreponham no tempo. Para isso, é importante que um algoritmo efetivo de admissão possa escolher dinamicamente o melhor servidor para iniciar MVs com necessidades complementares. Finalmente, uma falha de hardware no cluster pode afetar muitos serviços, comparando com a arquitetura dedicada onde apenas um serviço é afetado, normalmente. Essa desvantagem pode ser minimizada utilizando o mecanismo de alocação dinâmica dos recursos no cluster ao detectar alguma falha em um nó do cluster. A alocação dinâmica explora as facilidades oferecidas pelo MMV como a migração da MVs entre os servidores do cluster para balancear carga entre os nós do cluster. Ao acionar a alocação dinâmica, os serviços no nó que está apresentando falhas pode ser movido para outros nós que disponham dos recursos necessários.

Na investigação dessa tese, foram obtidas as seguintes contribuições:

1. A formulação de uma metodologia para o mapeamento da QoS, desejada por um serviço, em um perfil do uso dos recursos.

2. Uma arquitetura para a construção de uma nova plataforma para serviços de *streaming*, baseada em virtualização. A arquitetura SMART (*Small Machines Are Rather good Tools*) oferece a multiplexação de recursos em um cluster para muitos serviços, mantendo o isolamento encontrado em plataformas de cluster dedicado. SMART permite descobrir os recursos e a alocação necessária para manter o QoS desejado pelo serviço. Através da migração das MVs, o sistema provê a alocação dinâmica de recursos para serviços e melhora balanceamento de carga no sistema.

3. A apresentação de uma solução escalável e confiável para um sistema real de distribuição de vídeos na Internet, baseado na arquitetura SMART;

1.4 Organização

Esta tese está estruturada da seguinte forma. O próximo capítulo aborda os trabalhos que antecederam, inspiraram ou são próximos a esta tese. No Capítulo 3, são descritas as tecnologias existentes de virtualização de recursos e discute-se as funcionalidades e compromissos apresentados por dois modelos de virtualização. SMART, uma plataforma escalável que explora os benefícios dos mecanismos de virtualização para obter garantias de QoS em serviços de *streaming* é apresentado no Capítulo 4. No Capítulo 5, demonstramos uma solução baseada em SMART para um sistema real de distribuição de vídeos na Internet. O Capítulo 6 apresenta as conclusões e os trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Muito esforço tem sido dedicado nas últimas duas décadas para aperfeiçoar os mecanismos de gerenciamento e escalonamento de recursos em sistemas operacionais, primeiro voltados para atender a demanda de aplicações multimídia em computadores pessoais, e depois em sistemas baseados em cluster de computadores, para prover escalabilidade e tolerância a falhas com qualidade de serviço aos serviços web, principalmente. Este capítulo apresenta uma visão dos trabalhos que antecederam, inspiraram ou são próximos a esta tese.

A Seção 2.1 discute os trabalhos relacionados com o desenvolvimento de mecanismos de gerenciamento de recursos no contexto de sistemas operacionais para multimídia e tempo-real. A Seção 2.2 apresenta os trabalhos voltados para a alocação de recursos em clusters de computadores. Sistemas que utilizam virtualização para o isolamento e alocação eficiente de recursos no sistema, são apresentados na Seção 2.3.

2.1 Gerenciamento de Recursos no Sistema Operacional

Várias técnicas para a alocação previsível de recursos em uma máquina foram desenvolvidas na última década. Os escalonadores podem ser divididos em escalonadores do tipo *proportional-share*, tais como *Start-Time Fair Queueing* [36] e *Lottery* [37] ou escalonadores baseados em *reservas*, por exemplo, os escalonadores propostos nos sistemas Rialto [38] e Nemesis [39].

Lottery propõe gerenciar recursos usando tíquetes, onde cada classe de serviço obtém os recursos proporcionalmente ao número de tíquetes que possui. Como o algoritmo de escalonamento é probabilístico, as proporções das alocações não têm a garantia de ser exatamente as proporções esperadas. Entretanto, a disparidade entre as alocações diminui de acordo com o número de vezes que o recurso for escalonado. Isso significa que, a curto prazo o *Lottery Scheduling* não é preciso. O algoritmo *Stride Scheduling* [40] é uma versão determística de *Lottery*, que elimina a variabilidade no curto prazo. Basicamente, o cliente recebe um "intervalo" (*stride*) - um número de passos entre execuções do cliente -, do escalonador, que é inversamente proporcional ao número de tíquetes possuído. Assim, processos com muitos tíquetes possui um *stride* pequeno e é executado mais vezes. Por exemplo, um cliente com o dobro de tíquetes do outro, possui metade do intervalo e é alocado em uma freqüência duas vezes maior.

O escalonador *Fair Queueing* (FQ) [41] foi desenvolvido para distribuir a banda passante da rede entre vários fluxos que competem em um *switch*. Isso envolve manter filas separadas para cada fluxo e serví-los em um modo *round-robin*. Se alguma fila "estoura", os pacotes seguintes para aquela fila são descartados e os outros fluxos se mantém intactos. Uma vantagem desse algoritmo é sua simplicidade, pois ele não monitora as rajadas dos fluxos, e nem tenta controlar o tráfego dos fluxos. Uma desvantagem é que ele não provê qualquer garantia de taxa. Apesar de a banda ser distribuída igualmente, a proporção recebida por cada fluxo depende da quantidade de fluxos compartilhando a conexão. O escalonador *Weighted Fair Queuing* (WFQ) [41] é uma variação de FQ, onde cada fluxo recebe pesos específicos. Isto permite uma distribuição da banda proporcional ao peso designado a cada fluxo. Ainda, no escalonamento da transmissão na interface de rede, *Traffic Shaping* é um termo genérico dado para uma variedade de técnicas voltadas para moldar o tráfego de rede de acordo com algum comportamento desejado. O esquema *Leaky Bucket* [42] regula as rajadas do tráfego transmitido usando um *bucket* de capacidade finita, que gera permissões a uma taxa fixa. Pacotes da rede são transmitidos apenas quando existe uma permissão disponível. Assim, rajadas de pacotes são limitadas pela capacidade do *bucket*, enquanto a taxa média de transmissão é governada pela taxa de geração de permissões.

O escalonador *Start-time Fair Queueing* (SFQ) [43] pertence à classe de algoritmos derivados de WFQ. Um tempo virtual de cada pacote determina o escalonamento em WFQ. SFQ modifica WFQ, relacionando o tempo virtual com o momento inicial de cada cliente, e alocando o recurso para o menor tempo inicial. O escalonador multimídia *SMART* [44] integra prioridades e WFQ para alcançar os requerimentos de tempo-real, enquanto simultaneamente suporta aplicações comuns.

YFQ [45] é um escalonador *proportional-share* voltado para multiplexar eficientemente os recursos de disco. YFQ estende o algoritmo WFQ para obter justiça e alta banda nos acessos ao disco por muitos clientes. Os pedidos de acesso são escalonados de acordo com o algoritmo WFQ, mas ao invés de escolhar apenas um pedido, YFQ escolha n pedidos. Depois, um algoritmo clássico de acesso ao disco (como C-SCAN ou *shortest seek time first*) ordena os pedidos selecionados para obter o máximo de banda de acesso. Um escalonador de acessos ao disco rígido, chamado *Cello*, proposto por Shenoy e Vin [46], busca atender os diversos requerimentos de desempenho das aplicações correntes. Cello distribui pesos para classes de aplicações e aloca banda de acesso ao disco rígido proporcionalmente aos pesos. O escalonador busca o equilíbrio entre a menor latência no conjunto de acessos com a distribuição da banda de acesso entre as aplicações. Comparando com uma implementação de virtualização em SO - OpenVZ, por exemplo -, cada container recebe uma prioridade de E/S, e o escalonador distribui a banda disponível de acordo com as prioridades existentes. Assim, nenhum container pode saturar um canal de E/S. Em um segundo nível, é usado o escalonador CFQ para E/S no Linux.

O escalonamento *Borrowed Virtual Time* (BVT) [47] assemelha-se ao SFQ no que concerne o uso proporcional do recurso através do estabelecimento do tempo virtual para cada cliente. Entretanto, BVT é voltado para prover baixa latência para aplicações interativas e de tempo-real. O conceito principal é o empréstimo do tempo virtual reservado para o uso do recurso no futuro, assim o cliente pode ser alocado mais cedo, e em troca cede o uso do recurso que foi adiantado.

O SO Nemesis [39] segue uma abordagem diferente para o gerenciamento dos recursos. A aplicação e a maior parte da funcionalidade do kernel é executada no espaço de endereços do usuário. Desta forma é possível criar domínios de execução isolados - e gerenciar recursos a partir destes -, para processos correlacionados.

Novas abstrações para o gerenciamento de recursos, que vão além de simplesmente lidar com processos e threads, foram propostas. Entre muitos trabalhos, destacam-se *Resource Container*, *Software Performance Units* e *Virtual Service*. *Resource Container* [34] é uma abstração no sistema operacional que possibilita a alocação com granularidade fina e a contabilidade do consumo de recursos. Acoplado com *Lazy Receiver Processing* (LRP) [48], que permite a integração do processamento do protocolo de rede como gerenciamento de recursos, esta abstração é capaz de garantir o isolamento de desempenho entre aplicações em uma máquina. *Resource Container* emprega o escalonador *Lottery* para alcançar a alocação proporcional do processador.

Verghese e outros [49] apresentaram o gerenciamento de recursos em um multiprocessador de memória compartilhada. A técnica de gerenciamento de recurso, chamada de *Performance Isolation*, foi implementada para três recursos do sistema: processador, memória e banda de acesso ao disco rígido. *Performance Isolation* provê garantia de desempenho para uma *Software Performance Unit (SPUs)*, que é um grupo de processos de uma aplicação. Destaca-se, em *Performance Isolation*, o mecanismo proposto para garantia de desempenho em relação à memória física, que funciona provendo dinamicamente limites ajustáveis para o número de páginas usadas por diferentes SPUs.

Waldspurger [50] introduziu vários mecanismos novos para o gerenciamento de memória no monitor de máquina virtual VMWare ESX. Esses mecanismos permitem que o servidor superestime a quantidade de memória para alcançar melhor escalabilidade do que um simples particionamento estático da memória permitiria. Um algoritmo baseado na combinação de alocação proporcional de memória e no uso das páginas é usado para obter garantia de isolamento e alta taxa de acerto nos acessos à memória.

Reumann e outros criaram *Virtual Service (VS)* [51], uma abstração no sistema operacional que busca eliminar a interferência no desempenho causada por serviços compartilhados como *DNS*, serviços *cache proxy*, sistema de arquivo distribuído e banco de dados distribuído. Entretanto, oferece garantias mais fracas de isolamento e desafios adicionais para eliminar a interferência na gerência dos recursos.

No trabalho de Sullivan e Seltzer [52], o escalonador *Lottery* é estendido para

permitir variações na alocação de acordo com as reais necessidades da aplicação. A técnica é aplicada em três recursos do sistema: tempo de processador, memória física e banda de acesso ao disco rígido.

2.2 Gerenciamento de Recursos em Cluster

2.2.1 Escalonamento de Tarefas e Balanceamento de Carga

Várias técnicas foram propostas para obter a melhor alocação de recursos em um cluster. Sistemas como *Condor* [53], investigaram técnicas para a alocação de processadores em um cluster de máquinas para executar aplicações em *batch*. As técnicas como *gang scheduling* [54] e *implicit coscheduling* [55] foram propostas para coordenar o escalonamento de aplicações paralelas em sistemas distribuídos, entretanto apresentam garantias fracas de reserva de recursos.

Mosix [56] provê algoritmos para o compartilhamento dinâmico de recursos, junto com um mecanismo de migração de processos. Estes algoritmos respondem dinamicamente às variações no uso dos recursos entre as máquinas e através da migração de processos de uma máquina para outra buscam melhorar o desempenho das aplicações globalmente no cluster. Mosix busca apenas balancear o uso dos recursos entre as máquinas no cluster, não oferece nenhuma garantia de qualidade de serviço para as aplicações. Em [57], foi proposto um algoritmo para a alocação dinâmica *fair-share* de máquinas em um cluster, também baseado em migração de processos.

2.2.2 Qualidade de Serviço em Cluster de Computadores

Na década passada, surgiram muitas propostas de sistemas web escaláveis. Fox e outros [22] demonstraram um servidor escalável em cluster, usado para o serviço de busca *Inktomi*. Em *Porcupine* [23] foi desenvolvido um serviço de correio eletrônico altamente escalável, baseado em cluster. Esses trabalhos focaram os aspectos de escalabilidade e confiabilidade, através da alocação estática de recursos com granularidade de uma máquina. *Multispace* [24] apresentou uma infraestrutura de serviços, baseada em cluster, para o desenvolvimento de aplicação web. Porém não foi discutido o compartilhamento de recursos e o isolamento para mais de uma

aplicação. Shen e outros [58] discutiram o gerenciamento estático de recursos para um cluster "dedicado" a um serviço web. Mais recentemente, *Oceano* [29] e *Cluster-on-Demand (COD)* [30] descreveram uma infraestrutura de gerenciamento dinâmico de um cluster, onde subdomínios podem ser criados e reservados para um aplicação. O gerenciamento dos subdomínios é dinâmico, adicionando ou removendo máquinas de acordo com a demanda da aplicação. *Cellular Disco* [59] criou clusters virtuais, utilizando um monitor de máquina virtual, para gerenciar recursos e contenção de falhas em um multiprocessador de memória compartilhada.

Trabalhos mais recentes [33, 60, 61, 32] focaram na questão específica do gerenciamento de recursos em cluster compartilhado. Todos os trabalhos buscam resolver a alocação de recursos em cluster, usando um modelo de granularidade mais fina com múltiplas aplicações competindo pelos recursos em uma mesma máquina ou em alguns casos englobando todo o cluster.

Cluster Reservers [33] oferece o isolamento de desempenho entre múltiplos serviços web, que compartilham um cluster, baseado no conteúdo do pedido que chega, no cliente que realiza o pedido ou ambos. Enquanto a implementação apresentada é focada no tempo de processador, o mecanismo pode ser estendido para outros recursos, como memória, banda de acesso ao disco rígido e à rede. Partições (*shares*) fixas de recursos são entregues para aplicações distribuídas em vários nós, enquanto em cada máquina, as partições são ajustadas dinamicamente baseada no uso local dos recursos. A abstração *Resource Containers* [34] é empregada para o gerenciamento em cada máquina e um algoritmo de programação linear - com complexidade polinomial no tempo - para a alocação de recursos no cluster.

Muse [60] emprega um modelo econômico para o aprovisionamento dinâmico de recursos para múltiplas aplicações. O objetivo principal é incorporar o consumo da energia elétrica ao gerenciamento dos recursos. A técnica empregada busca melhorar o consumo de energia, adaptando-se à flutuação do uso dos recursos pelas aplicações no cluster e respondendo aos transientes de energia com o mínimo de degradação no desempenho das aplicações no cluster.

Arpaci-Dusseau e Culler [31] mostraram um escalonador *proportional-share* para aplicações seqüenciais, iterativas e paralelas em um ambiente distribuído. *Stride Scheduling*, com pequenas modificações, foi usado em cada máquina para escalonar

as aplicações.

Sharc [32] gerencia o tempo de processador e banda de rede em um cluster compartilhado. O modelo de uma aplicação em Sharc, composto de um conjunto de cápsulas, é genérico o suficiente para permitir que funcione com qualquer das técnicas mencionadas acima, como *reservations* [39, 38] e *shares* [36]. Mesmo usando um escalonador *proportional-share*, Sharc provê uma alocação absoluta de recursos usando *reservations* através de um controle de admissão - o controle de admissão garante os recursos para as aplicações e restringe o escalonador *proportional-share* a fazer uma distribuição justa dos recursos não usados. Sharc não permite a acumulação de créditos e as trocas são restritas pela reserva agregada de recursos de uma aplicação.

O MMV provê emulação em software de máquinas físicas, que são idênticas ao hardware subjacente. A virtualização, através do MMV, oferece vários benefícios: isolamento total entre MVs, independência de hardware, migração e configuração dinâmica de uma ambiente de execução. A abstração de um MV, contendo o SO e seus processos, serve a um propósito semelhante ao apresentado pelas abstrações *Resource Container*, Cápsulas (em Sharc) e *Virtual Service*: o encapsulamento de um ambiente de computação ativo. Estando o SO todo contido na MV, evita-se o problema do compartilhamento do espaço de nomes e existe total isolamento entre as instâncias. O MMV atribui os recursos aos SOs convidados, provendo uma forma de gerenciamento de recurso hierárquico.

Como vimos, alguns trabalhos para uma máquina [50, 51] ou multiprocessadores [59] já mostraram vantagens no uso de virtualização para o gerenciamento de recursos. Comparando com os mecanismos normalmente empregados no gerenciamento de recursos no cluster (*resource container*, *software performance units* e *Virtual Service*), o uso de máquinas virtuais oferece garantias mais fortes de isolamento e elimina o *crosstalk* na qualidade de serviço desejada.

2.3 Virtualização

Figueiredo et alli [62] mostraram a viabilidade do uso de máquinas virtuais no ambiente de computação em *grid*. O uso de virtualização suporta mais facilmente a abstração, oferecida pelo ambiente em *grid*, de desacoplamento da computação

dos recursos físicos existentes. O software Globus, para a implementação de infra-estrutura de computação em *grid*, oferece o serviço *Virtual Workspaces* [63]. Este serviço permite ao cliente especificar a alocação de recurso para uma MV específica. No versão corrente, apenas memória e o tempo de início de execução são gerenciados.

PlanetLab é uma rede *overlay* de máquinas distribuidas geograficamente, por vários países, para o desenvolvimento e acesso de novos serviços para a Internet. PlanetLab permite múltiplos serviços executarem simultaneamente e continuamente, cada um usando sua "fatia" do sistema. A peça central da arquitetura é o conceito de "fatia", uma rede de máquinas virtuais, que aloca partições dos recursos locais da máquina real em que estão executando. Os recursos de cada máquina real são escalonados em *fair-share*, mas limitados no uso máximo, como no tráfego de rede por exemplo. PlanetLab é descrito em [1], e usa Linux VServers com a adição de funcionalidades de controle e proteção de SO Scout [64].

2.3.1 Migração

O suporte para a migração de grupos de processos entre máquinas foi apresentado em [65], mas a aplicação tem que ser suspensa e não resolve a questão das conexões de rede ativas. A migração de máquinas virtuais tem sido usada para a alocação dinâmica de recursos em ambientes *grid*. Um sistema empregando migrações automáticas de MVs para aplicações científicas em um ambiente *grid* foi empregado em [66]. O sistema Shirako [67] provê infra-estrutura para o "empréstimo" (*leasing*) de recursos e usa a migração de MVs para adequar as diferenças entre as políticas de alocação do "gerente de empréstimo" e as políticas do provedor do cluster. O *Escalonador de Recursos Distribuído* [68] da VMware uso migração para balancear automaticamente o cluster de acordo com o uso do processador e da memória.

Capítulo 3

Virtualização - Fundamentos e Tecnologia

O monitor de máquina virtual (MMV), também conhecido como *hypervisor*, foi introduzido no início dos anos 70 e alcançou o sucesso comercial com a série de *mainframes* IBM 370. O conceito de virtualização, trazido pelo MMV, permitiu que os *mainframes* executassem múltiplos sistemas operacionais (SO) simultaneamente, tornando possível o compartilhamento no tempo de uma máquina grande e cara sem precisar modificar nenhum software de sistemas legados, includindo os SOs mono-usuários. Com o advento de servidores de baixo custo e computadores pessoais, a necessidade de MMVs, para essa finalidade, diminuiu nas décadas seguintes.

Mas, ainda em 1974, Goldberg identificou (em [69]) como vantagens das máquinas virtuais: a melhor segurança, confiabilidade e a consolidação de servidores. As motivações identificadas para a virtualização continuam importantes para os sistemas de computação atuais. No decorrer de três décadas, os computadores se tornaram mais rápidos e baratos, mas também cada vez mais conectados com outros computadores, tanto em redes locais, como em redes de grande distância. Novas oportunidades para aplicar a virtualização de recursos surgem neste contexto, onde um grande número de máquinas estão conectadas à Internet. A virtualização de recursos provê um veículo para por em prática muitos dos resultados das pesquisas em sistemas de computação distribuídos nos últimos anos. Por exemplo, as técnicas de migração, balanceamento de carga, confiabilidade e segurança podem ser mais facilmente aplicadas através de máquinas virtuais (MVs), do que nas abstrações

correntes – como processos, sistemas de arquivo e soquetes TCP/IP, entre outros –, por razões como o encapsulamento implícito do estado da computação e a interface mais simples encontradas na MV.

Este capítulo apresenta o conceito de virtualização e descreve as diversas tecnologias de monitor de máquina virtual (MMV). O nosso objetivo é entender como os mecanismos para o gerenciamento de recursos, entre eles o isolamento das MVs, o monitoramento e o controle fino dos recursos e a migração das MVs, podem trazer os benefícios destacados anteriormente.

3.1 Fundamentos

As atuais tecnologias de virtualização foram propostas por diversos grupos, com diferentes objetivos, gerando alguma confusão nos conceitos relacionados. Nesta seção consideramos as diversas arquiteturas de máquinas virtuais e descrevemos as arquiteturas de uma forma unificada, de acordo com o modelo proposto por Smith e Nair [70], colocando o conceito de virtualização e os tipos de MVs em perspectiva.

3.1.1 Abstração e Virtualização

Os sistemas de computadores são compostos de hierarquias de camadas com interfaces bem definidas por níveis de abstração. A Figura 3.1 [70] mostra algumas interfaces e camadas importantes em um típico computador. O uso de interfaces bem definidas facilita o projeto independente de subsistemas por ambos os grupos de desenvolvimento de software e hardware. As abstrações simplificam e escondem os detalhes de implementação do subsistema na camada subjacente, conseqüentemente reduzindo a complexidade do projeto. O entendimento destas interfaces é importante para a virtualização do sistema.

Interface ISA (*Instruction Set Architecture*) A arquitetura do conjunto de instruções marca a divisão entre o hardware e o software e consiste das interfaces 3 e 4 na Figura 3.1. A interface 4 representa a interface no nível do usuário e inclui as instruções visíveis a uma aplicação. A interface 3 é um super-conjunto da interface 4 e inclui também as instruções vistas apenas pelo SO, responsável pelo gerenciamento dos recursos do hardware.

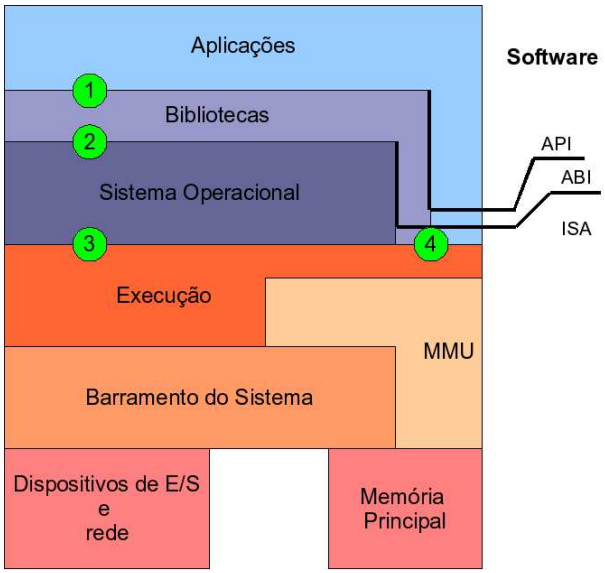

Figura 3.1: Arquitetura de um computador. As camadas se comunicam verticalmente através da arquitetura do conjunto de instruções (em inglês, ISA), da interface binária da aplicação (em inglês, ABI) e da interface de programação da aplicação (em inglês, API)

.

Interface ABI (*Application Binary Interface*) A ABI provê a um programa o acesso aos recursos do hardware e aos serviços disponíveis em um sistema através do subconjunto de instruções do usuário (interface 4) e as chamadas de sistema (*system calls*) na interface 2. A ABI não permite o controle direto do sistema, mas todos os programas interagem com os recursos de hardware indiretamente através das abstrações criadas pelo SO. As chamadas de sistema executam serviços oferecidos pelo SO ao usuário, após a validação da autorização e segurança.

Interface API (*Application Programming Interface*) A API apresenta basicamente as mesmas funcionalidades da ABI, mas é implementada com bibliotecas desenvolvidas com linguagens de alto nível. Ela permite executar os programas em diferentes sistemas, depois de recompilados.

A Figura 3.2 [70] mostra um exemplo de abstração aplicada a um disco rígido. O sistema operacional abstrai os detalhes de endereçamento no disco rígido - por exemplo, que é composto por trilhas e setores - apresentando uma interface para

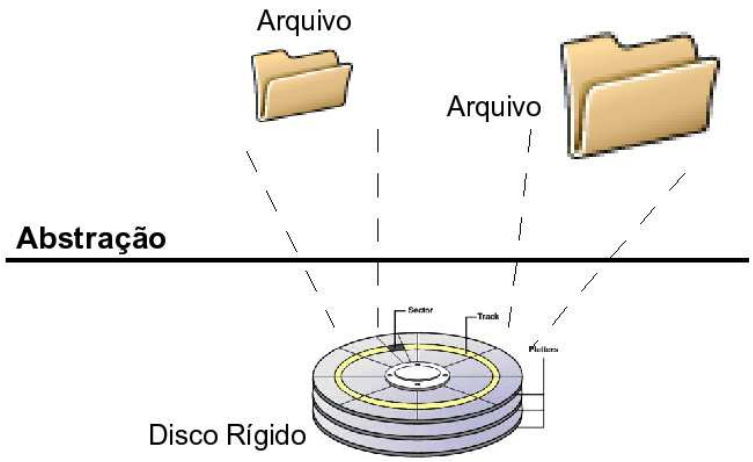

Figura 3.2: Abstração aplicada a um disco. Abstração provê uma interface simplificada para os recursos subjacentes

.

a manipulação de arquivos e diretórios com tamanhos variáveis. Programadores podem usar a abstração para criar, ler e escrever arquivos sem conhecer o funcionamento e a organização física do disco rígido.

A virtualização de um sistema ou componente - como um processador, memória, ou um dispositivo de entrada/saída - em um certo nível de abstração mapeia sua interface e recursos visíveis em uma outra interface e recursos de um sistema real subjacente. Conseqüentemente, o sistema real aparece como um sistema virtual ou mesmo múltiplos sistemas virtuais. O sistema virtual pode ser um clone idêntico ao sistema real ou possivelmente o mapeamento pode criar características diferentes para a interface e recursos virtuais.

Diferentemente da abstração, a virtualização não necessariamente tem como objetivo simplificar ou esconder os detalhes de implementação do sistema e componentes. Por exemplo, na Figura 3.3 [70], a virtualização transforma um disco rígido em dois discos virtuais menores, cada qual com seus setores e trilhas próprios. O software de virtualização usa a abstração de um arquivo como o passo intermediário para prover o mapeamento necessário entre os discos real e virtual. Uma escrita para o disco virtual é convertida para uma escrita no arquivo, e conseqüentemente em uma escrita no disco real. Note que o nível de detalhe provido na interface de disco virtual - o endereçamento através de setores e trilhas - não é diferente do

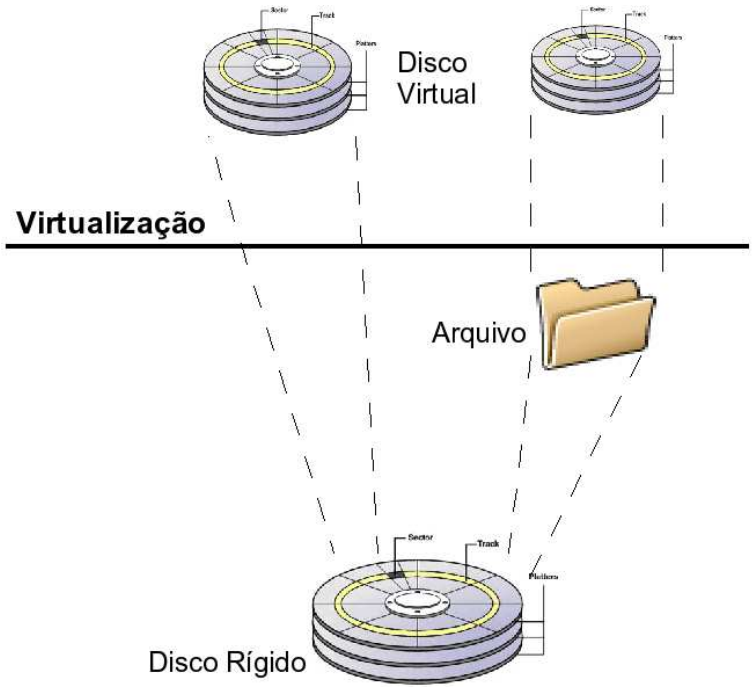

Figura 3.3: Virtualização aplicada à um disco. A virtualização provê uma réplica do recurso no mesmo nível da abstração. A interface virtual pode ser idêntica à interface do hardware real ou pode implementar um recurso com características diferentes

.

endereçamento do disco real, nenhuma abstração é usada no disco virtual.

3.1.2 Conceitos de Virtualização

A virtualização pode ser aplicada não apenas para subsistemas como discos mas para todo o sistema em uma máquina. Para implementar uma máquina virtual, os desenvolvedores adicionam uma camada de software à máquina real para suportar a arquitetura desejada. Através do mapeamento implementado pela camada de virtualização, a máquina virtual pode oferecer uma interface diferente e esconder restrições do hardware na máquina real. Para entendermos o que é uma máquina real, é necessário primeiro considerar o significado de "máquina" do ponto de vista de um processo e do sistema.

Do ponto de vista de um processo do usuário, a máquina consiste de um espaço lógico de endereços de memória reservado para o processo contendo as instruções no nível do usuário e os registradores que permitem a execução do código e armazenar

o estado pertencente ao processo. Os dispositivos de entrada/saída são visíveis apenas através das abstrações criadas pelo SO. A interface binária da aplicação (*ABI*), ver Figura 3.1, define a máquina vista pelo processo, assim como a interface de programação de aplicação (*API*) especifica as características da máquina em uma linguagem de alto nível.

Do ponto de vista do SO, a máquina é todo o hardware subjacente, onde o sistema inteiro é executado. O sistema é um ambiente de execução completo que pode atender múltiplos processos simultaneamente. Estes processos compartilham o processador, um sistema de arquivos, páginas de memória física e outros dispositivos do hardware. O ambiente do SO persiste no tempo independente do início e término dos processos executados. O sistema aloca tempo de execução no processador, memória real e recursos de E/S para os processos, e permite que eles interajam com seus recursos. Da perspectiva do SO, então, as características do hardware apenas definem a máquina; é a arquitetura do conjunto de instruções (*ISA*) que provê a interface entre o sistema e a máquina.

Assim como existem perspectivas do processo e do sistema para a "máquina", existem também dois tipos de máquinas virtuais, uma para o processo e outra para o sistema. Uma MV de processo é uma plataforma virtual que executa um único processo. Essa MV existe apenas para suportar o processo; é criada junto com o processo e termina quando o processo termina. A MV de processo mais comum é tão ubíqua que poucos a consideram uma MV. A maioria dos SOs podem simultaneamente suportar múltiplos processos de usuário através da multiprogramação, que proporciona para cada processo a ilusão de possuir uma máquina completa para ele. O SO compartilha no tempo o hardware e gerencia os recursos para tornar isso possível. Outro exemplo de MV de processo bastante conhecido hoje, é a MV implementada no nível da linguagem de alto nível, como são a arquitetura de MV Java de Sun Microsystems [71] e a *Common Language Infrastructure* da Microsoft [72]. Este tipo de MV não corresponde diretamente a nenhuma máquina real, mas é projetada para permitir a portabilidade do código para qualquer plataforma de hardware que execute a MV e explorar as características da respectiva linguagem de alto nível. Em contraste, a MV de sistema provê um ambiente persistente e completo que suporta um SO junto com seus muitos processos de usuários. Ela permite

ao SO "convidado" o acesso ao hardware virtual, como processador, memória, rede e outros recursos de entrada e saída (E/S). O processo ou sistema que executa sobre a MV é o componente "convidado", enquanto a plataforma subjacente que suporta a MV é o "hospedeiro". O software de virtualização, que cria as MVs, é normalmente chamado de monitor de máquina virtual (MMV) ou *hypervisor*.

Um meio termo aos dois tipos de MVs descritas anteriormente (a de processo e a de sistema), é a virtualização no nível do SO. Através da *ABI*, o SO virtualiza diferentes ambientes de execução (tempo de processador, sistema de arquivos, páginas de memória e outras abstrações), criando uma MV para um conjunto de processos. Neste tipo de virtualização, normalmente a MV é chamada de "compartimento" (*container*) ou ambiente virtual (AV). Além de criar um compartimento para um conjunto de recursos e processos, a MV define a API para a qual os programas são escritos. O kernel do SO é compartilhado pela MVs, mas as abstrações criadas são específicas para cada conjunto de processos compartimentados. O isolamento entre as MVs é obtido através de uma combinação de contexto de segurança do processo e identificadores no espaço de nomes. Na execução de uma chamada de sistema, os privilégios de acesso de controle são verificados e é decidido que recurso deve ser exposto ao processo.

Com a exceção dos casos onde se faça necessário usar a denominação específica da técnica de virtualização, optamos por simplificar a terminologia para manter a clareza do texto. É mantido o uso geral do termo MMV para o software que provê a virtualização, no lugar de "virtualização no SO" ou *hypervisor*. Para o ambiente de execução virtual, seguimos com o uso de MV, no lugar "ambiente virtual", compartimento, domínios ou processos.

3.2 Tipos de Virtualização e Técnicas de Implementação

Como explicado anteriormente, uma MV provê um ambiente completo no qual o SO e muitos processos, possivelmente pertencendo a muitos usuários, podem coexistir. Através do MMV, uma única máquina pode suportar múltiplos SOs convidados executando isoladamente uns dos outros e simultaneamente. A virtualização pode

ser aplicada em diferentes níveis do sistema: no nível da aplicação, no sistema operacional ou no nível do hardware. Operando nos níveis do SO e do hardware, o MMV deve executar algumas das funções de um SO normal, como controlar e arbitrar o acesso ao processador, aos recursos de memória e outros dispositivos de hardware. Estas operações exigem que o MMV execute no modo privilegiado e substitua a execução pelo SO nas operações privilegiadas. A seguir descrevemos os tipos de virtualização e implementações no nível do hardware e do SO. Finalizamos com uma discussão das vantagens e desvantagens da virtualização nos dois níveis.

3.2.1 Implementações no Nível do Hardware

De acordo com Popek e Goldberg [73], uma MMV tem três características. Primeira, uma MMV provê **um ambiente de execução muito próximo ao da máquina original**; qualquer processo em uma MV deve ser executado como se estivesse na máquina real. Exceções à essa regra advém das diferenças na disponibilidade de recursos do sistema, na dependência da temporização e nos dispositivos de E/S oferecidos. Se a disponibilidade do recurso, por exemplo, espaço físico de memória, é diferente, o programa pode tomar decisões diferentes. A temporização esperada pelo programa ser alterada porque o MMV pode intervir e executar um conjunto diferente de instruções quando uma instrução privilegiada é executada na MV. Finalmente, se a MV não está configurada com todos os dispositivos de E/S disponíveis na máquina real, o comportamento do programa executado pode não ser o esperado.

A segunda característica obriga a MMV a **ter o total controle dos recursos da máquina real**. Nenhum processo na MV pode ter acesso a qualquer recurso do hardware sem que seja explicitamente alocado pelo MMV. Como também, o MMV pode retomar o controle de um recurso alocado previamente para uma MV.

A **eficiência** é a terceira característica. Um grande percentual de instruções do processador virtual deve ser executado pelo processador real sem a intervenção do MMV. As instruções que não podem ser executadas diretamente pelo processador real são tratadas pelo MMV. O desempenho do sistema depende do tamanho do subconjunto de instruções geradas pelo MMV.

Diferentes técnicas podem ser usadas para obter as características citadas, cada uma oferecendo diferentes resultados na implementação. A avaliação dos resultados

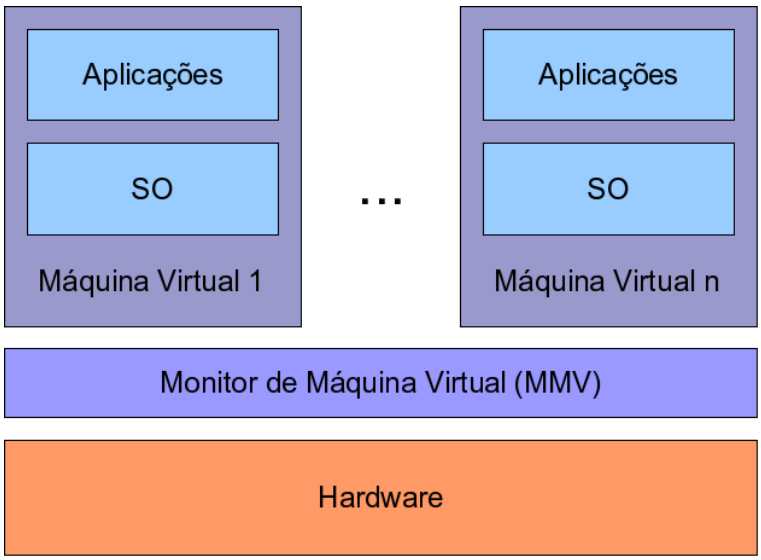

Figura 3.4: Virtualização Clássica ou tipo I

deve levar em conta os seguintes objetivos: **compatibilidade**, **desempenho** e **simplicidade**. A compatibilidade é importante para manter a habilidade de executar software legado. O segundo objetivo procura tornar a execução do MMV o mais transparente possível para o desempenho do SO e seus processos na MV. A simplicidade é mantida para evitar falhas no MMV, que causariam problemas para todas as MVs executando no sistema. Em particular, para prover a segurança na execução da MV é necessário que o MMV esteja livre de falhas que possam ser exploradas por ataques externos.

Existem duas variações para a virtualização em hardware [74]:

Virtualização Clássica ou do Tipo I. Do ponto de vista do usuário, a maioria das MVs provê essencialmente a mesma funcionalidade, mas difere nos detalhes de implementação. A abordagem clássica (ver Figura 3.4) coloca o MMV diretamente sobre o hardware e as MVs são criadas na camada acima. O MMV executa com a maior prioridade, enquanto os sistemas convidados executam no modo de usuário, de forma que o MMV pode interceptar e emular todos as operações do SO convidado que acessam ou manipulam os recursos de hardware. Exemplos deste tipo são VMware ESX server [75], Sun xVM [76] e o MMV Xen [77].

Virtualização Hospedada ou do Tipo II. Uma implementação alternativa exe-

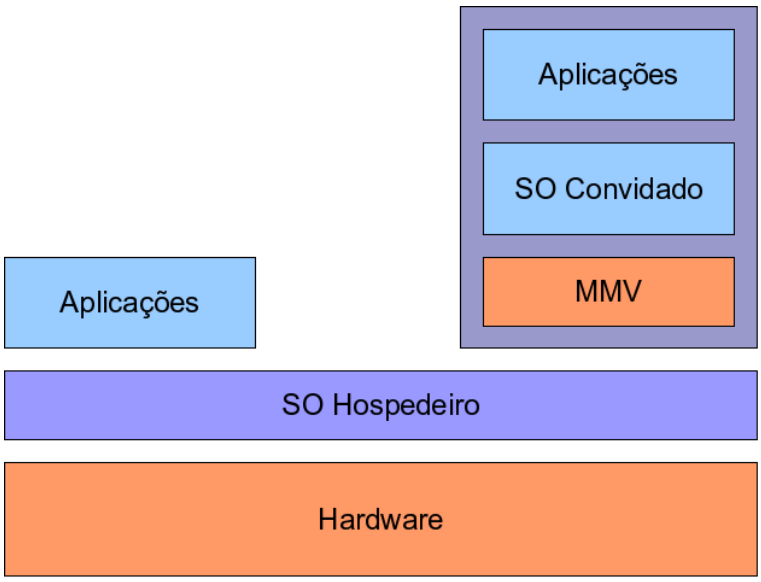

Figura 3.5: Virtualização hospedada ou tipo II

cuta a camada de virtualização acima de um SO hospedeiro, resultando em uma MV hospedada, ver Figura 3.5. Uma vantagem dessa abordagem é que o usuário instala a MV como uma aplicação típica. Mais ainda, a MV pode alcançar o hardware através dos controladores (*drivers*) de dispositivos do SO, além de outros serviços de sistema, sem precisar de um MMV. Exemplos da implementação de uma MV hospedada são o servidor VMware GSX para plataforma de hardware Intel IA-32, e o VirtualBox [78]. Outro exemplo é o Virtual PC, da Microsoft, que executa um sistema Windows em uma plataforma Macintosh. Nesse caso, como o *ISA* difere entre o sistema convidado e o hospedeiro, a MV virtualiza todo o software, emulando tanto o código das aplicações como do SO convidado.

Descrevemos a seguir as implementações de virtualização em hardware para o processador, memória e E/S, em geral:

Virtualização do Processador

Uma técnica para implementar as características básicas de um MMV é a "execução direta", que permite a execução da MV diretamente na máquina real, mas sob o controle total do MMV. Essa técnica transfere o modo de execução do SO convidado para o modo não-privilegiado do processador, enquanto o MMV executa no modo

privilegiado. Quando algum código na MV tenta executar uma operação privilegiada, o processador gera uma exceção e passa a executar o código respectivo no MMV. Assim, o MMV pode controlar os acessos privilegiados aos recursos feitos pela MV.

O tratamento de uma instrução que desabilita as interrupções provê um bom exemplo. Se o SO convidado puder desabilitar realmente as interrupções, o MMV não mais receberá o controle da execução. A solução é o MMV registrar que as interrupções foram desabilitadas para aquela MV e adiar a entrega das próximas interrupções até que sejam habilidadas novamente.

Infelizmente, quase todos os processadores atuais não estão aptos à técnica "execução direta", incluindo a arquitetura mais popular - a Intel IA-32. Por exemplo, algumas instruções x86 não-privilegiadas referenciam ou podem modificar a configuração de recursos (memória), e podem acessar registradores com informações do modo de privilégio.

Por exemplo, um código executado na MV pode ler o registrador CS [1] para determinar o nível de privilégio corrente. A "execução direta" espera que o processador gere uma exceção para que o MMV possa informar o nível de privilégio correto da máquina virtual. Um processador x86, entretanto, não interrompe a execução da instrução e o software recebe a informação de privilégio incorreta. Robin and Irvine [79] identificaram várias instruções x86 que estão nesta categoria. Estas instruções ao serem executadas por um SO convidado podem afetar o comportamento de outras VMs ou inadvertidamente, prejudicar a sua própria operação. Estas instruções devem ser substituídas por serviços do MMV.

Várias técnicas alternativas podem ser usadas para implementar um MMV em processadores que não permitem a "execução direta". As mais usadas são a "paravirtualização" e a "execução direta" combinada com a "tradução binária rápida". Na paravirtualização, a interface da MV não é exatamente a interface da máquina real. Em termos do processador, o SO convidado deve ser modificado para remover as instruções que não permitem a virtualização. Estas intruções devem ser substituídas por trechos de código equivalentes, que respeitem a "execução direta". As modificações no SO convidado são transparentes para as aplicações do usuário.

[1] registrador CS - Code Segment - possue nos bits 0 e 1 o nível corrente da execução.

Disco[80] é um MMV implementado para o processador MIPS, usando a técnica da paravirtualização. Foi usada uma abordagem de substituir os registradores privilegiados no processador por posições especiais de memória. Foram feitas modificações no código do SO para trocar as instruções problemáticas, como a leitura do registrador CS, por trechos de código que usam aquelas posições especiais na memória. Estas modificações também permitem evitar as interrupções em instruções privilegiadas, resultando em um incremento do desempenho do sistema.

A maior desvantagem da paravirtualização é a incompatibilidade com SOs que não foram modificados para a arquitetura da MV. Mas mesmo com esse inconveniente, muitos projetos de pesquisa [35, 4, 63] escolhem trabalhar com a paravirtualização, pois as implementações obtidas têm apresentado bons resultados de desempenho.

O MMV VMware ESX desenvolveu uma nova técnica de virtualização que combina a "execução direta" com a tradução binária em tempo de execução. Basicamente, o código do kernel (privilegiado) é executado sob o controle de um tradutor binário. O tradutor troca o código que contém instruções problemáticas por outro equivalente que pode ser executado diretamente no processador. A tradução é armazenada em uma *trace cache* para otimizar o desempenho. A tradução binária é feita também para otimizar a "execução direta", eliminando muitas das interrupções geradas por instruções privilegiadas.

Uma abordagem adotada pelos fabricantes de processadores foi criar um novo modo nos processadores para executar o MMV. As extensões *Hyperprivileged mode* dos processadores Sun UltraSparc T1 e T2 [81], *VMX* dos processadores Intel [82] e *SVM* [83] dos processadores AMD, permitem o MMV executar o SO convidado sem modificaçoes e emulação do código (virtualização "completa").

O escalonamento das MVs, pelo MMV, normalmente segue alguma variação do algoritmo *fair-share* [84]. O acesso ao processador, dado a uma MV, é baseado na fração de "créditos" (ou *shares*) possuídos do total de créditos existentes (por toda MVs) no sistema. O termo crédito define o quanto será alocado na execução do processador para a respectiva MV. Se um número grande de créditos é designado para uma MV, em relação às outras MVs, então esta MV é escalonada mais vezes pelo MMV. Os créditos não são equivalentes a porcentagens de recursos do proces-

sador. Os créditos são usados para definir o peso relativo do uso do processador por uma MV em relação às outras MVs. A seguinte fórmula mostra como o escalonador calcula a alocação do processador por MV:

$$Alocacao_{MV^i} = \frac{Creditos_{MV^i}}{TotalCreditos}$$

O escalonador também pode especificar o uso mínimo (reserva) e máximo (limite) do processador por cada MV. Uma reserva garante que a MV receberá pelo menos esta porcentagem de alocação de tempo do processador, não importando o número total de créditos. O limite assegura que a MV nunca use mais que o tempo máximo determinado, mesmo que o processador esteja disponível. O algoritmo *fair-share* só é aplicado se a utilização do processador estiver na faixa determinada pelo limite e reserva.

Virtualização da Memória

O gerenciamento de memória pelo MMV envolve duas tarefas: o particionamento da memória física entre as MVs e o suporte à tradução dos endereços na MV.

A virtualização do espaço de endereços é criada no MMV através de uma outra camada de endereçamento - o endereço de máquina (EM). Dentro do SO convidado, o endereço virtual (EV) é usado pelas aplicações, e o endereço físico (EF) é usados pelo SO em operações de *DMA* e nas tabelas de páginas. O MMV mapeia o EF de uma MV para o EM, que é usado diretamente no hardware.

Normalmente, o MMV ?mantém uma tabela de páginas (para cada MV) para a tradução do EF para o EM - a tabela de página "sombra". Assim, o MMV pode manter o controle preciso das páginas de memória da máquina real disponíveis para uma MV. Quando o SO convidado estabelece um mapeamento novo em sua tabela de página, o MMV detecta a modificação e atualiza a respectiva entrada na tabela de página sombra, que aponta para a localização real da página de memória no hardware. Quando a MV está executando, o hardware usa a tabela de página sombra para a tradução do endereço, de forma que o MMV sempre tem o controle das páginas que cada MV está usando.

O subsistema de memória do MMV constantemente controla a quantidade de memória usada por uma MV. Ele pode periodicamente recuperar as páginas de memória transferindo uma parte do espaço de memória da MV para o disco. Essa

funcionalidade conflita com o gerenciamento de memória feito pelo SO convidado, o que pode resultar, por exemplo, na anomalia da paginação dupla, quando o SO convidado força o MMV a trazer uma página do disco, simplesmente para no momento seguinte retirá-la do seu espaço de memória.

O MMV Xen usa a paravirtualização para os processadores x86. Diferente do método tradicional, ele não trabalha com tabela de página sombra. Para obter melhor desempenho, Xen apenas interfere nas atualizações feitas pelo SO convidado na tabela de página para verificar a validade das modificações. Para isso, Xen permite o registro da tabela de página do SO convidado diretamente com a *MMU*, mas restringe os acessos a apenas leituras. As operações de atualizações são passadas para Xen, através de chamadas especiais inseridas no SO - as *hypercalls* - que valida e aplica as modificações nas entradas da tabela de página.

VMware e Xen mantém o controle das quantidade de páginas usadas através de uma técnica conhecida como "*ballooning*" [50]. Esse mecanismo permite que o MMV controle a quantidade de páginas físicas usadas pela MV. Quando Xen quer retomar algumas páginas de memória, ele pede ao *balloon driver* para alocar mais memória. As páginas virtuais reservadas pelo SO convidado são então "presas" às respectivas páginas físicas. As páginas físicas podem assim ser retiradas da MV pelo MMV. As decisões de alocação e transferência de páginas de memória para o disco é feita apenas pelo SO convidado, evitando qualquer redundância no gerenciamento de memória, como descrito acima para o método tradicional.

Virtualização de Entrada e Saída

Os sistemas de computação atuais, com seus diversos dispositivos de hardware tornam a virtualização de E/S muito difícil. Os sistemas baseados na arquitetura IA-32 suportam uma quantidade muito grande de dispositivos de diferentes fabricantes, com diferentes interfaces de programação. Conseqüentemente, o MMV virtualiza todo o hardware em alguns controladores (*devices drivers*) apenas ou deixa o controle dos dispositivos para o SO convidado. No último caso, por causa das limitações, apresentadas anteriormente, os dispositivos só podem ser controlados exclusivamente por uma MV.

Em geral, existem três opções para a virtualização de E/S, como ilustradas na

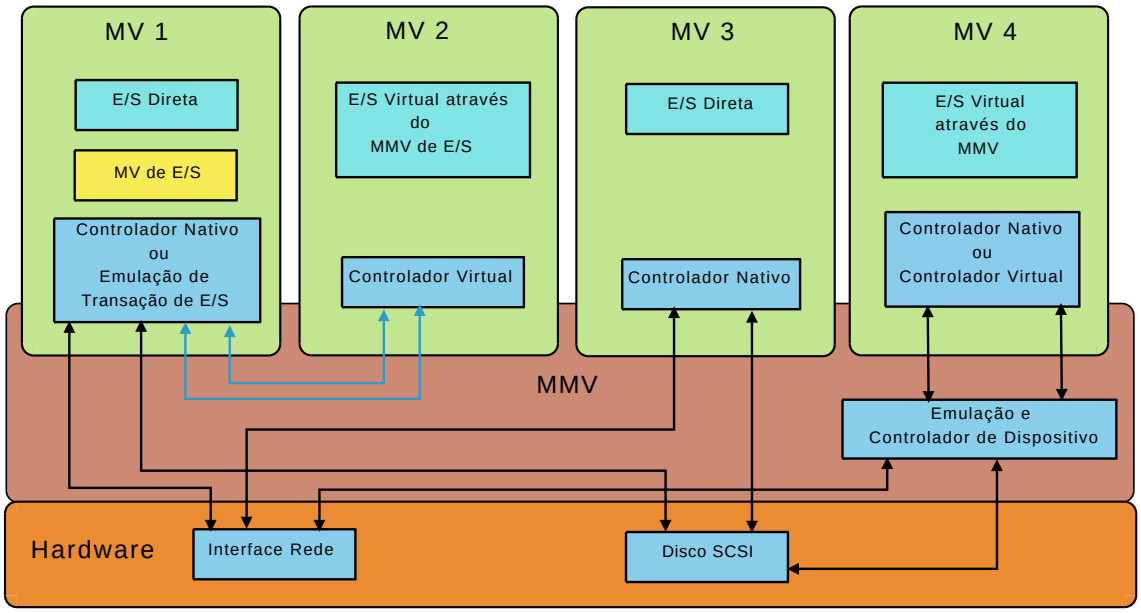

Figura 3.6: Diferentes técnicas de virtualização de E/S, usadas por monitores de máquina virtual.

Figura 3.6:

1. E/S Direta (MV1 e MV3);
2. E/S Virtual usando transação emulada (MV2);
3. E/S usando emulação de dispositivo (MV4).

As máquinas MV1 e MV3 utilizam do modelo E/S direta. Neste modelo, o MMV exporta o uso e a gerência de todos ou alguns dispositivos para a MV. O software controlador (*device driver*) encontrado no respectivo SO é usado pela MV para a comunicação direta com o dispositivo. Na Figura 3.6, a MV1 é uma MV especial que provê o acesso de E/S para outras MVs. O controlador virtual de dispositivo da MV2, na Figura 3.6, recebe pedidos de E/S das aplicações e os re-envia, através do MMV, para o respectivo controlador nativo no servidor de dispositivos (MV1). O MMV1 pode então aplicar o pedido diretamente no dispositivo. Esse modelo de "transação de E/S emulada" é tipicamente usada em implementações paravirtualizadas porque o SO do lado do cliente precisa incluir os controladores especiais para a comunicação com o seu controlador correspondente no SO do servidor, através de um canal no MMV.

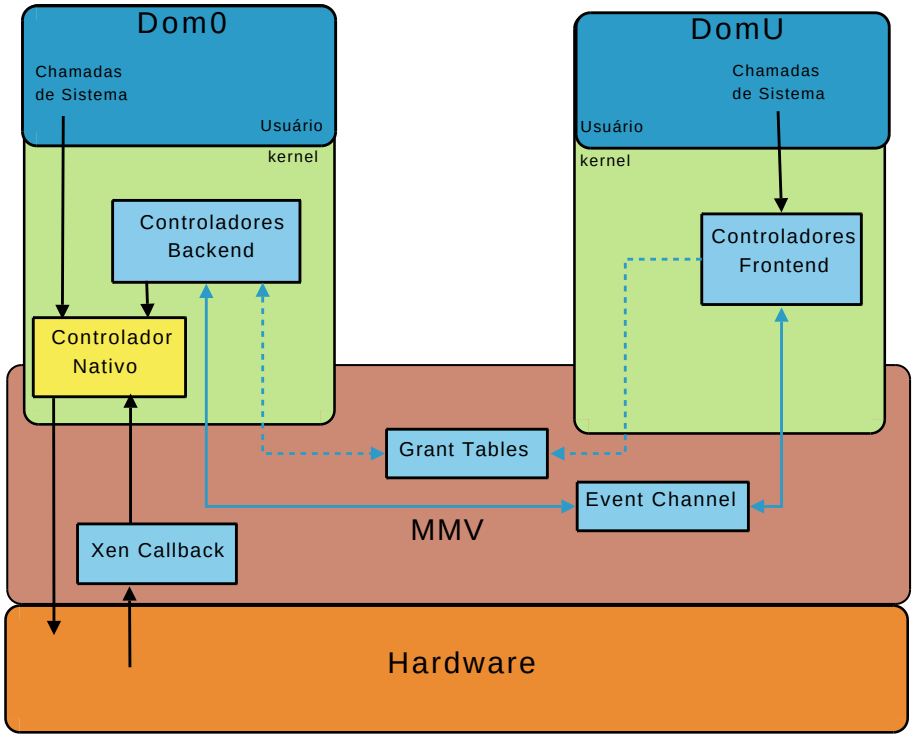

Figura 3.7: Xen - Safe Hardware Interface.

Em sua última versão, Xen implementa a arquitetura *Safe Hardware Interface*, ver Figura 3.7, que permite os controladores de dispositivo, sem modificação, serem compartilhados por MVs isoladas, enquanto protege cada instância de SO e o sistema como um todo de falhas nos controladores.

Os domínios de dispositivos, na terminologia Xen, são MVs específicas para executar controladores nativos de dispositivos. Isso permite a mesma compatibilidade alcançada pela arquitetura "hospedada", mas sem a sobrecarga do SO hospedeiro. As MVs executam um controlador *front-end* simples, que se comunica através da *grant table*, implementado no MMV, com o controlador *back-end* no domínio de dispositivo. Os domínios de dispositivo acessam diretamente os dispositivos possuídos. Entretanto as interrupções vindas dos dispositivos são tratadas primeiro pelo MMV, que então notifica o domínio de dispositivo correspondente através do *xen callback*. Os eventos e interrupções entre o domínio de dispositivo e o domínio da aplicação são gerados através do *event channel*.

Ainda é possível que os sistemas não-paravirtualizados usem controladores especiais, como os citados anteriormente, para um melhor desempenho nas transações de E/S. A emulação de transações de E/S pode causar problemas de compatibilidade

nas aplicações, se o controlador virtual não provê todas as funções de controle e dados que o controlador real possui.

A "emulação do dispositivo" provê a emulação de um tipo de dispositivo, permitindo que o controlador do dispositivo emulado seja usado no SO convidado. O MMV exporta dispositivos emulados para uma MV de forma que os controladores existentes dos dispositivos do SO possam ser usados. Os acessos aos registradores e portas de E/S pelo controlador no SO convidado irão gerar falhas devido aos endereços inválidos. Estas falhas são capturadas pelo MMV e transformadas em acessos reais ao dispositivo de hardware real. A MV4, na Figura 3.6, usa controladores nativos ao SO para dispositivos emulados exportados pelo MMV.

A emulação do dispositivo é normalmente menos eficiente e mais limitada do que a emulação das transações de E/S. A emulação de dispositivo não exige modificações no SO convidado e, assim, é tipicamente usada para prover a virtualização completa em hardware.

Para superar estes problemas, o MMV VMware Workstation utiliza a arquitetura [85] "hospedada" (*hosted architecture*) mostrada na Figura 3.5. Nessa arquitetura, o MMV é instalado como uma aplicação do SO hospedeiro, como Windows NT ou Linux, e usa os *drivers* do SO hospedeiro para ter acesso aos dispositivos de E/S. Como esses SOs possuem os *drivers* para os dispositivos, a camada de virtualização pode suportar qualquer dispositivo de E/S. Essa técnica funciona bem para *desktops*, mas a sobrecarga do SO hospedeiro é muito alta para a virtualização dos dispositivos de E/S em servidores.

3.2.2 Virtualização no Nível do SO

Como foi explicado anteriormente, o SO cria uma ilusão de uma máquina inteira dedicada para o processo. Estendendo esse conceito, é possivel particionar os recursos do sistema de maneira a criar múltiplos contextos de execução, onde os processos e recursos são isolados, provendo a ilusão de múltiplos, independentemente gerenciados, ambientes de execução virtuais (AVs) executando em uma única máquina. Um AV possui seu próprio conjunto de processos (iniciados do *init*), sistema de arquivos, usuários (incluindo *root*), interfaces de rede com endereços IP, tabela de rotas, regras de firewall etc. AVs podem possuir versões diferentes do SO (bibliote-

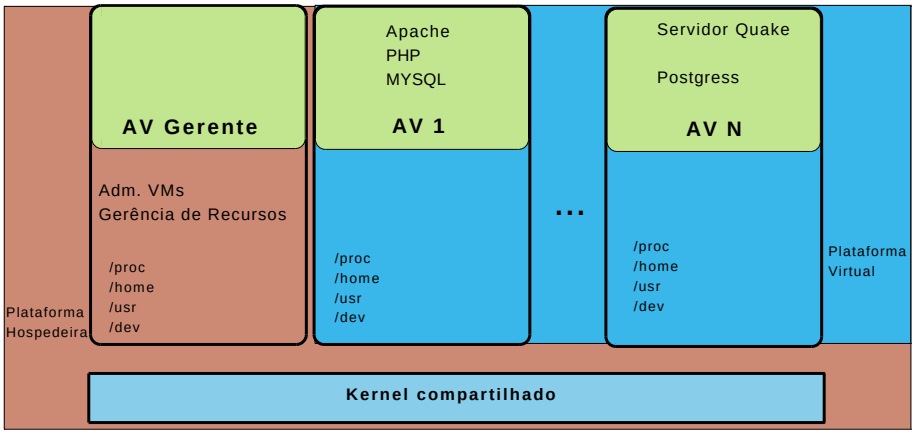

Figura 3.8: Virtualização no nível do SO

cas, serviços, compiladores, interface gráfica, entre outros serviços), mas executam sob o mesmo kernel. Exemplos de virtualização no SO incluem Solaris Containers [86], Linux VServer [87], FreeBSD Jails [88], Parallels Virtuozzo [89] e a sua versão "software livre" OpenVZ [90].

Como mostrado na Figura 3.8, existem três componentes básicos para a virtualização do SO. A plataforma hospedeira consiste essencialmente do kernel compartilhado e do AV gerente (AVG). Este é o AV que possui todos os privilégios para modificar parâmetros do kernel quando necessário e gerenciar as outras AVs. Cada AV hóspede pode ser criada e destruída como um SO normal. As AVs mantém a API e ABI do SO original, assim as aplicações executam sem modificações.

Neste nível, não existe conceitualmente nenhuma diferença entre a virtualização feita no hardware ou no SO. As AVs podem ser vistas como MVs e a virtualização do kernel possui todas as funcionalidades de um MMV. Entretanto, as duas versões diferem fundamentalmente nas técnicas usadas para implementar o isolamento entre as MVs. Discutimos estas diferenças mais adiante, mas antes apresentamos um exemplo da implementação de um SO virtualizado.

OpenVZ

O OpenVZ é uma implementação de virtualização no kernel do Linux, que oferece as seguintes funcionalidades: virtualização e isolamento de vários subsistemas, gerenciamento de recursos e *checkpointing*. Estes componentes são descritos a seguir:

Virtualização e Isolamento. Cada AV tem seu próprio conjunto de recursos pro-

vido pelo kernel. Dentro do kernel, estes recursos são virtualizados ou isolados:

- Arquivos - bibliotecas de sistema, /proc e /sys virtualizados, *locks* virtualizados;
- Usuários e grupos - Cada MV possui seu usuário root, como também usuários e grupos específicos;
- Processos - uma MV enxerga apenas seu próprio conjunto de processos, começando pelo *init*. *PIDs* são virtualizados, de forma que o *PID* do *init* é um em qualquer MV;
- Rede - o subsistema de rede é virtualizado de forma que a MV possui seus próprios endereços IP, como também um conjunto de regras de *firewall* e roteamento;
- Dispositivos - Alguns dispositivos são virtualizados. Ainda, se for necessário, qualquer MV pode receber o acesso exclusivo ao dispositivo real como interface de rede, porta serial, partições de disco etc;
- Objetos de comunicação entre processos (*IPC*) - memória compartilhada, semáforos e mensagens.

Gerenciamento de recursos O controle do consumo de recursos é de extrema importância para uma solução de virtualização no nível do SO. Existe um conjunto finito de recursos dentro do kernel que são compartilhados entre as MVs. O gerenciamento se divide em:

- Quota de disco - a quota pode ser configurada em dois níveis. Quotas para o uso de espaço de disco e *inodes* por MV, no primeiro nível. A configuração, no segundo nível, do uso do disco por usuário e grupo, dentro da MV;
- Escalonamento *Fair-Share* do processador - funciona basicamente como explicado na "Virtualização do Processador". Pode ser configurado os créditos e o limite de uso do processador para cada MV. No primeiro nível, o MMV escalona a MV a ser executada baseada na quantidade de créditos possuídos. No segundo nível, o escalonador do Linux escolhe qual processo deve executar, de acordo com as filas de execução da MV.

Além dos créditos, pode ser estabelecido um limite máximo de execução em um percentual do tempo de processador disponível;

- Escalonamento de E/S - similar ao escalonamento do processador, é também um escalonador de dois níveis, utilizando o escalonador CFQ [91] no segundo nível. Cada MV recebe uma prioridade de E/S. O escalonador distribui a banda disponível de acordo com as prioridades de cada MV. Assim, nenhuma MV pode saturar o canal de E/S;
- *Beancounters* do usuário - são parâmetros de controle (contadores, limites e garantias de uso) dos recursos de cada MV. São 20 parâmetros que buscam cobrir todos os aspectos da operação da MV. O uso dos recursos pode ser garantido, contabilizado e/ou controlado através dos parâmetros. Os recursos são principalmente memória, objetos de *IPC* e *buffers* de rede. Cada parâmetro pode ser utilizado através do sistema de arquivos */proc* do Linux e possui cinco valores associados: *uso corrente*, *uso máximo* (no decorrer da execução da MV), *barreira*, *limite* e *contador de falhas*. O significado de *barreira* e *limite* depende do parâmetro, mas eles podem ser vistos como um limite da garantia (*barreira*), onde o MMV estabelece algumas restrições de uso do recurso quando esse limite é alcançado e o uso máximo (*limite*), onde é especificado que nenhum recurso a mais pode ser alocado acima dessa quantidade. Por exemplo, na configuração do parâmetro *kmemsize*, se o uso do recurso - memória alocada para o kernel - excede a *barreira*, mas não ultrapassa o *limite*, operações vitais, como a expansão da pilha do processo, podem ainda alocar o novos recursos, mas outras operações serão impedidas. A diferença entre os valores de *barreira* e *limite* oferece a chance para que as aplicações possam se adaptar ao uso máximo do recurso, já que freqüentemente é devido aos picos de uso e não a uma demanda média de uso do recurso. O contador de falhas mostra o número de vezes que o respectivo parâmetro ultrapassou o limite configurado. A configuração do parâmetro pode ser alterada em tempo de execução e se torna efetiva para a MV logo após a modificação.

Checkpointing - assim como na virtualização em hardware, o isolamento dos re-

cursos e objetos do SO em uma MV, permite facilmente suspender e salvar seu estado para o disco. A MV pode ser reiniciada no mesmo ponto em que foi suspensa, mais tarde.

3.3 Migração da Máquina Virtual

Existe um outro benefício da virtualização, além de suportar múltiplos SOs convidados executando isoladamente uns dos outros e permitir o gerenciamento eficiente dos recursos do sistema, que é a migração da máquina virtual. A migração do SO e todas as suas aplicações, como uma unidade, consegue evitar muitas das dificuldades encontradas nas técnicas de migração de processo. A migração no nível da MV significa que todo o ambiente de execução, em memória, pode ser transferido de modo consistente e eficiente. Isto se aplica para o estado interno do kernel (ex. bloco de controle do TCP para uma conexão ativa), como também para os estados no nível da aplicação, mesmo quando está compartilhado entre vários processos. Em termos práticos, por exemplo, isto significa que podemos migrar um servidor de *chat* ou um servidor de transmissão de vídeo sem a necessidade de os clientes refazerem a conexão. Com exceção de algumas dependências do hardware local, como a leitura do contador de ciclos do processador em máquinas de um cluster heterogêneo, a migração da MV deve ser transparente para as aplicações.

A migração do SO é extremamente útil em sistemas baseados em cluster de computadores, pois ao tornar independente o software do hardware, expande as funcionalidades como tolerância a falhas e balanceamento de carga. Por exemplo, se uma máquina real começa apresentar problemas, as MVs podem ser transferidas para outras máquinas do cluster. Assim, a máquina que apresenta falhas pode ser retirada para a manuntenção. De modo similar, as MVs podem ser re-distribuídas no cluster para aliviar aquelas máquinas reais que se encontram sobrecarregadas. Nestas situações, a combinação de virtualização e migração incrementa significativamente a visão de um sistema consistente e único para um conjunto de unidades de hardware.

A transferência do conteúdo da memória de uma MV entre duas máquinas reais pode ser alcançada de várias maneiras. Entretanto, em alguns casos deve se evitar

a suspensão das aplicações que estão executando na MV. Por exemplo, evitar a reconexão dos muitos usuários de um servidor de jogos, e em outros a migração deve ocorrer no menor tempo possível, mesmo precisando suspender as aplicações, como seria o caso de ocorrência de falhas em uma máquina real. Assim, a técnica usada para a migração da MV pode buscar minimizar apenas o *downtime*, ou também o "tempo total de migração". No segundo caso, durante todo o intervalo de migração, a execução da MV é suspensa, conseqüentemente qualquer serviço naquela máquina permanece indisponível. Esse período de tempo é potencialmente visível para os clientes, como uma interrupção do serviços. No caso do *downtime*, a MV e o serviço são suspensos apenas no intervalo de tempo necessário nas últimas operações da migração, ainda necessárias antes de reiniciar a MV na máquinas remota. A implementação da migração de MVs, em quase todos os casos, usa uma das seguintes técnicas descritas a seguir:

- Pré-cópia - A MV continua a executar enquanto as páginas são enviadas através da rede para a máquinas remota. Para assegurar a coerência, as páginas modificadas, durante esse processo, devem ser re-enviadas. Como pode ser visto, essa técnica busca minimizar o *downtime*. Descrevemos o algoritmo mais detalhadamente a seguir:

  1. Reserva os recursos necessários para a MV na máquina remota;
  2. Iterativamente copia as páginas da memória alocada pela MV, enquanto o SO e as aplicações executam. O primeiro passo copia toda as páginas físicas usadas pela MV. Antes de cada página ser transferida, ela é protegida contra escrita, para que qualquer modificação em uma destas páginas seja detectada pelo MMV. Quando o primeiro passo é completado, aquelas páginas que foram modificadas na MV são copiadas novamente para a máquinas remota, enquanto a MV continua a execução do SO e aplicações;
  3. Quando a maior parte da memória, ou o número de modificações se mantém constante por muitas iterações, a MV é suspensa e as páginas restantes são transferidas para a máquinas remota;
  4. reinicia a MV na máquina remota.

O VMWare VirtualCenter [92] e o MMV Xen são exemplos de implementações de migração usando a técnica pré-cópia.

- *Suspend/Resume* - a MV é suspensa e todo o estado de execução, na memória, é salvo em uma imagem no disco. Esta imagem é transferida para outra máquina real e a MV é restaurada. Como todo o estado do SO, incluindo as conexões de rede, é salvo, da visão do cliente, a migração parece apenas como um atraso na resposta do sistema. Nesto modo, o tempo de transferência é proporcional à quantidade de memória física alocada para a MV. Na transferência, o SO e as aplicações estão suspensos, assim o tempo total pode ser menor do que no modo pré-cópia, mas o *downtime* é significativamente maior.

Outro desafio na implementação da migração das MVs é como transferir os recursos ligados a outros dispositivos de hardware. Enquanto a memória pode ser copiada diretamente para a nova máquina real, as conexões para os dispositivos locais como interfaces de rede e discos rígidos requerem considerações adicionais. A simples cópia da imagem no disco (o sistema de arquivos da MV) pode incrementar muito o intervalo de migração. Em muitos casos assume-se que o sistema de discos é compartilhado entre as máquinas reais, sendo necessário apenas reconectar a imagem à nova máquina real.

Para os recursos de rede, deseja-se que o SO mantenha as conexões ativas sem precisar de mecanismos de reenvio na máquina fonte (que pode ser removida do cluster após a migração). A migração deve incluir todo o estado dos protocolos de rede usados no SO e os endereços IP ativos. Para atender estes requisitos, observa-se que no cluster, as interfaces de rede são conectadas em uma simples rede local (uma mesma subrede), implementada por *switches*. Uma solução prática é gerar um pacote não-solicitado de resposta *ARP* na máquina destino, avisando que aquele endereço IP foi transferido para um novo local. O recebimento deste pacote reconfigura as outras interfaces para enviar os pacotes para o novo endereço físico. Enquanto alguns pacotes são perdidos, o serviço é capaz de continuar usando as conexões existentes sem nenhuma interferência explícita.

Finalmente, uma preocupação adicional com a migração é o impacto no desempenho dos outros serviços ativos na máquina real. Por exemplo, a transferência iterativa das páginas de memória entre duas máquinas em um cluster pode facil-

mente consumir a capacidade da rede disponível e assim interferir na comunicação de outros serviços. Esta degradação de desempenho pode ocorrer para qualquer técnica de migração se não houver algum controle dos recursos usados no processo de transferência. Tanto a virtualização em hardware, como no SO, oferecem mecanismos para limitar o consumo dos recursos pelo responsável pela migração.

3.4 Desempenho e Isolamento

Em termos de desempenho, as implementações de virtualização tanto no nível do hardware e como no SO são semelhantes para aplicações *cpu-bound*. Enquanto, para as aplicações focadas em E/S, a virtualização no SO é mais eficiente na utilização dos recursos do sistema e por isso alcança melhor desempenho. A virtualização no nível do SO introduz sobrecarga mínima e permite a execução de um número de MVs em uma ordem de grandeza maior que as implementações em hardware. Os MMVs como VMware, Xen e UML não conseguem alcançar esta densidade por causa da sobrecarga (consumo de memória, falhas de cache L2 etc.) causada na execução de múltiplos kernels. Comparações entre os dois níveis de virtualização podem ser encontradas nos trabalhos de S. Soltesz et alli [93] e P. Padala et alli [94],[95].

Observa-se que a virtualização em hardware privilegia o isolamento das MVs. Entretanto, quando cada MV está executando o mesmo kernel e distribuições de SO similares, o grau de isolamento oferecido pelo MMV pode custar a eficiência relativa de executar todas as aplicações no mesmo kernel. A virtualização define três tipo diferentes de isolamento: o isolamento de falha, o isolamento de recursos e o isolamento de segurança. A seguinte discussão ilustra as diferenças entre a virtualização em hardware e SO em relação ao isolamento.

Isolamento de Falha. Corresponde à habilidade do MMV de isolar a falha ocorrida em um MV de modo a não afetar as outras MVs na mesma máquina real. Para assegurar o completo isolamento de falha é necessário que não exista o compartilhamento de código e dados entre as MVs. Entretanto, a virtualização no SO é aplicada diretamente no kernel compartilhado por todas as MVs. Assim, o MMV neste caso não pode prover o isolamento no caso de uma falha no kernel. O mesmo não ocorre com as MVs, que possuem seu próprio kernel

cada uma, na virtualização em hardware.

Isolamento de Recursos. Corresponde à habilidade da MMV de isolar o uso de recursos para evitar interações indesejadas entre as MVs. O isolamento, provido pela MMV, involve o escalonamento e a alocação de recursos da máquina real (ex.: ciclos de processador, espaço em disco), como também os recursos lógicos (*buffers* de transmissão, por exemplo) compartilhados pelas MVs.

Isolamento de segurança. Se refere à extensão ao qual o MMV limita o acesso aos recursos lógicos, como arquivos, endereços de memória, portas de E/S ou identificações de processos e outros. Esse isolamento deve promover (1) independência de nomes, onde os identificadores (ex.: arquivos, processos, locks) de uma MV não conflitam com os identificadores de outras MVs e (2) segurança, para que uma MV não seja capaz de ver ou modificar dados ou código pertencente à outra MV, de modo que qualquer falha de segurança (ex.: vírus, quebra de senhas) em uma MV, não afete outra MV na mesma máquina real. Os dois tipos de virtualização implementam os dois requisitos de segurança, escondendo os recursos lógicos em uma MV das outras MVs.

3.5 Sumário

Esse capítulo mostrou os conceitos sobre virtualização e criou uma base para que possamos entender em profundidade as diversas implementações de MMVs. A tecnologia de virtualização oferece mecanismos poderosos para o gerenciamento de recursos, entre eles o isolamento das MVs, o monitoramento e controle fino das recursos e migração das MVs. Mas como pode um cluster explorar estes mecanismos? Um desafio é encontrar mecanismos e políticas práticas para controlar estas mecanismos de forma autônoma, sem o auxílio de operadores humanos. Uma outra questão surge ao sabermos que um número de diferentes implementações existem, e a escolha do MMV para um sistema particular é claramente motivada pelo compromisso entre a eficiência e o grau de isolamento oferecido. A virtualização traz benefícios para uma grande variedade de cenários. Para alguns, como serviços web para jogos e transmissão de vídeos por exemplo, o balanceamento entre o isolamento e a eficiência é de fundamental importância. Experimentos [93] indicam que implementações de

virtualização no SO provêm, para certas aplicações, um desempenho até duas vezes melhor que a virtualização em hardware.

Capítulo 4

A Arquitetura SMART

Neste capítulo, é apresentada a arquitetura SMART para uma plataforma de serviços streaming, baseada em cluster de computadores. A arquitetura consiste de três componentes principais: [1] a) *Control*, que controla a admissão de novos pedidos de serviço no cluster, a localização da MV, as decisões de migração e as operações de tolerância a falhas; b) *Agente 86*, que monitora o uso dos recursos no nó de serviço, executa a migração e realiza operações de tolerância a falhas; c) *Agente 99*, que mapeia os requisitos de QoS apresentados pelo serviço no respectivo uso de recursos.

A Figura 4.1 apresenta a arquitetura do sistema SMART, com os componentes descritos a seguir.

Front-End de Serviço: Um componente ligado ao serviço. Pode existir um *Front-End* para cada tipo de serviço. Tipicamente realiza as funções: a) recebe os pedidos de serviço; b) distribui o pedido entre as diversas MVs executando o serviço. Pode existir mais de um nó dedicado ao componente Front-End. Esse componente pode trabalhar, através de uma API exportada, em conjunto com componente *Control*. Por exemplo, para pedir a execução de uma nova instância de um tipo de serviço, com a chegada ou saída de novos usuários que demandam variação no uso de recursos. Esses pedidos devem passar pelo algoritmo de admissão e conseqüentemente reservar novos (ou liberar) recursos no nó de serviço.

[1] A *Control* e seus melhores agentes 86 e 99 lutam para não deixar a *KAOS* dominar o mundo, no seriado Agente 86 (título original *Get Smart!*).

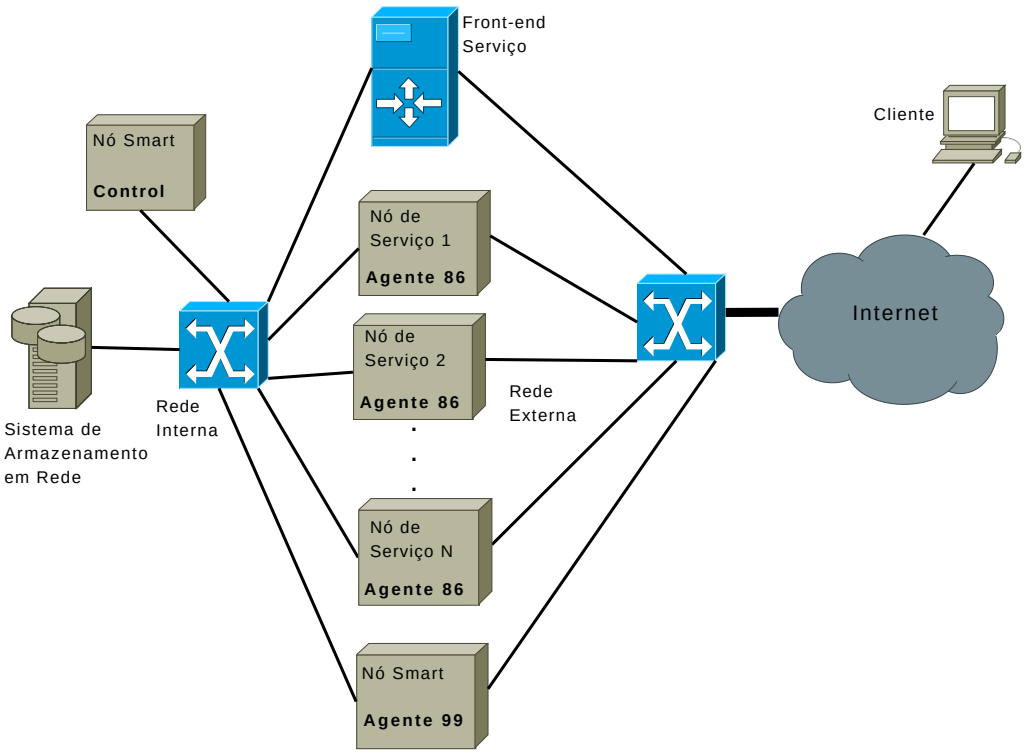

Figura 4.1: Arquitetura SMART

Nó de Serviço: Executa os múltiplos serviços isolados e monitorados através das MVs. A MV de gerência executa o Agente 86.

***Control*:** Este componente executa o algoritmo SMART para a admissão de novos serviços ao sistema, para isso, (i) assegura que exista recursos suficientes para cada novo serviço, (ii) determina em que nó de serviço a MV irá executar e (iii) realoca dinamicamente novos recursos para uma MV, através da migração. O módulo de tolerância a falhas é responsável por detectar falhas e recuperar os outros componentes do SMART. Recebe aviso de problemas no nó de serviço e tenta evitar a falha através da migração dos serviços para outros nós.

Agente 86: Monitora o comportamento e o uso de recursos no nó de serviço. Executa operações para o *Control*, como: (i) executar e remover a MV no nó de serviço; (ii) alocar os recursos necessários para a execução da MV e (iii) executa uma migração pedida pelo Control. Seu módulo de tolerância a falha, detecta a ocorrência de problemas no nó de serviço e avisa ao *Control*. Também, monitora o funcionamento do *Control*. Em caso de falha, reinicializa uma nova instância em um novo nó.

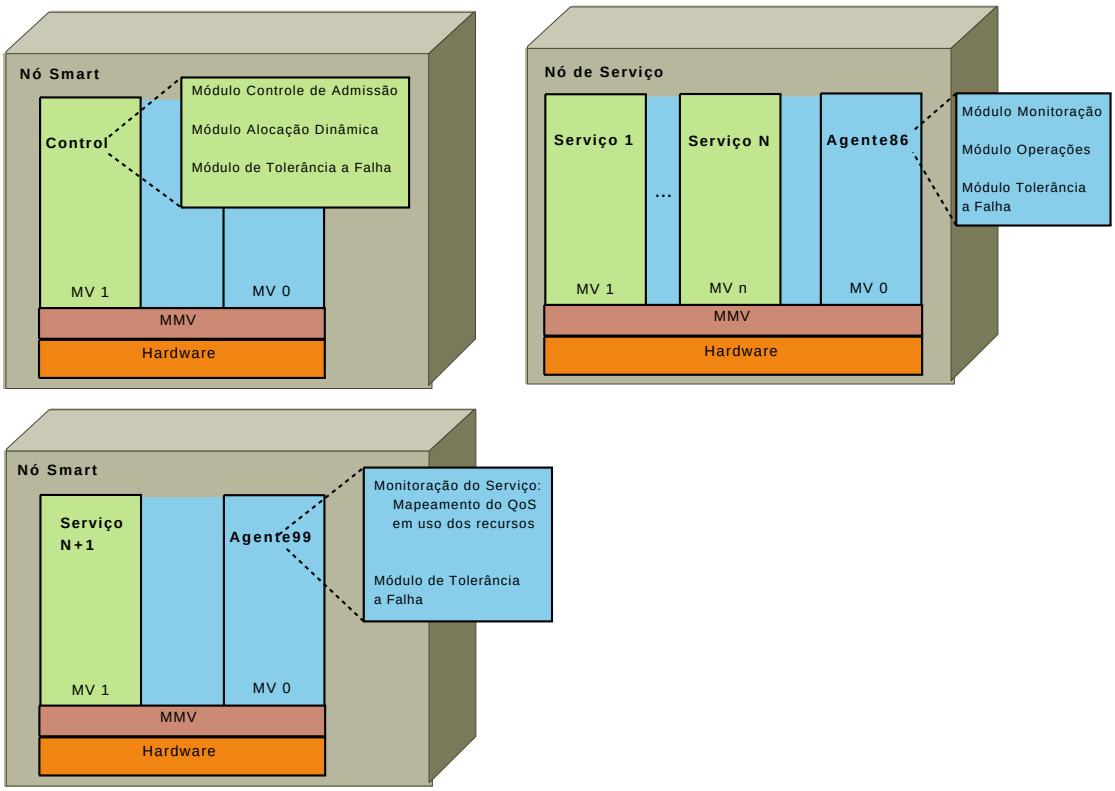

Figura 4.2: Composição dos Componentes Principais

Agente 99: Descobre o uso dos recursos necessários para manter a QoS desejada para o serviço.

4.1 Mecanismos e Políticas em SMART

Nesta seção descrevemos e discutimos os mecanismos e políticas utilizadas nos agentes descritos anteriormente.

4.1.1 Descobrindo o Perfil do Uso dos Recursos

Antes de alocar recursos para o serviço, o operador deve determinar os recursos necessários para atender os parâmetros de QoS[2]. Os parâmetros de QoS não quantificam diretamente o uso dos recursos. Assim, para os parâmetros dados, o perfil de uso dos recursos deve ser encontrado antes de admitir o serviço no sistema. A

[2]Indicadores de desempenho do serviço e a qualidade de serviço que é oferecida para os usuários. Exemplos de parâmetros são: taxa de pedidos atendidos, tempo de resposta e transmissão em tempo-real.

demanda dos recursos pode variar de acordo com o número de clientes no sistema no decorrer do tempo ou de acordo com a quantidade de processamento necessária para gerar um conteúdo dinâmico. Por exemplo, na transmissão de um vídeo, o uso dos recursos depende do número de clientes assistindo a um mesmo vídeo, o que varia no tempo, ou de acordo com o vídeo, diferentes taxas de decodificação exigem diferentes taxas de transmissão de pacotes e uso de buffer de memória.

Para mapear os parâmetros no uso dos recursos, o serviço é submetido a uma fase de "caracterização do perfil" de uso de recursos. Nesta fase, o serviço é executado em um sistema específico, onde os recursos estão totalmente livres (em um modo de alocação por "melhor esforço") para o uso do serviço. Assume-se que o nó utilizado tenha as mesmas características que os nós de serviço, ou que seja possível o mapeamento do uso dos recursos entre os nós heterôgeneos. O serviço é então submetido ao uso real pelo clientes. A utilização dos recursos e os parâmetros de QoS são monitorados pelo Agente 99 por um período suficiente para capturar um padrão de comportamento do serviço.

Na implementação corrente, estamos utilizando mecanismos de caracterização do perfil de uso embarcados no kernel do SO. Optamos por estes mecanismos por duas razões fortes: primeiro, sendo embarcado no kernel, este mecanismo funciona com qualquer aplicação e não requer qualquer alteração no código fonte ou binário do software. O que é bastante importante se o serviço executar software fornecido por terceiros. Em segundo lugar, a estimativa precisa dos recursos necessários pelo serviço requer uma informação detalhada sobre quando e quanto recursos são usados pela aplicação em um granularidade fina no tempo. Técnicas embarcadas no kernel podem prover informações precisas de eventos como intervalos de escalonamento do processador e de transmissão de pacotes na rede. Apesar das vantagens citadas, sistemas embarcados no kernel, são difíceis de configurar e instalar, o que deve ser feito a cada atualização do SO. Um outro problema com essa técnica é a tradução dos dados obtidos em uma alocação de recursos na MV. A análise dos dados e respectivo perfil de uso dos recursos são realizados pelo operador. Alternativamente, pode-se obter o uso dos recursos através da monitoração oferecida pelo MMV, mas não na mesma granularidade obtida através do kernel. A vantagem deste método é a tradução automática dos dados para a configuração da MV.

O mecanismo de caracterização de perfil usado é o implementado no kernel do Linux pelo *Linux Trace Toolkit* [96]. Este software provê uma implementação flexível e de baixa sobrecarga no kernel para gerar informações sobre uma variedade de eventos no kernel, tais como chamadas de sistema, processos, memória, sistema de arquivo e operações de rede. O usuário pode especificar os eventos desejados, como também os processos que irão gerá-los. Para o nosso propósito, especificamos as atividades do processador e da rede. Monitoramos os instantes em que processos da aplicação são escalonados e a duração do uso do processador, como também os períodos de transmissão na rede e os tamanhos dos pacotes.

Determinação Empírica do Perfil de Uso dos Recursos

Na caracterização do perfil de uso, assume-se que o parâmetro de QoS apresentado tem uma relação linear aproximada com o uso do recurso encontrado na fase de monitoração. Se um único recurso, como o tempo de processador, por exemplo, foi o limitador para um determinado tempo de resposta desejado, é esperado que o incremento da quantidade de tempo do processador permita proporcionalmente ajustar o tempo de resposta ao intervalo desejado. Observa-se que essa premissa é verdadeira para os tipos de recursos alocados no tempo, mas para aqueles com alocação espacial (como memória e espaço em disco) a relação não é aproximadamente linear. Por exemplo, alocando mais memória que o conjunto de trabalho pode aumentar o desempenho da aplicação, mas não atinge uma escala linear. Por outro lado, a alocação de um conjunto menor, pode impor um penalidade drástica ao desempenho, caso o SO entre no estado de "paginação sem fim" (ou *thrashing*). Nenhum sistema de virtualização oferece atualmente um mecanismo de controle fino de uso da memória, apesar de algumas propostas [52, 50] já existirem. A solução adotada é baseada no particionamento da memória principal entre os conjuntos de trabalho dos serviços. Durante a fase de caracterização do perfil, o tamanho do conjunto de trabalho é monitorado. Para as aplicações com a alocação estática, o tamanho obtido é usado. Para outras aplicações, com o conjunto dinâmico, é obtido um percentual (normalmente maior que 90%) da quantidade que não ofereça a paginação no disco. Estes serviços podem ser admitidos no sistema, se o tamanho do conjunto de trabalho adicionado não ultrapassar a capacidade de memória do nó de serviço. A monitoração no nó de serviço (pelo Agente 86) permite o controle de

picos de até 10%. Caso a demanda ultrapasse esta variação, podem ser alocadas mais páginas (se existir o recurso no nó de serviço) ou iniciar uma migração para um nó com mais recursos disponíveis.

4.1.2 Controle de Admissão

Utilizamos os "traços" dos eventos do kernel obtidos na caracterização do perfil para modelar o uso do processador e da rede. Isso é obtido observando o uso do recurso a cada intervalo t, durante um período de monitoração. Esse conjunto de "traços" é usado para derivar a distribuição probabilística do uso do recurso e a série de uso no tempo. Através dessa caracterização dos recursos, a decisão de controle de admissão é feita pela seguinte heurística:

1. **Assegure que a admissão do novo serviço não excede a capacidade do cluster.** Para n nós de serviços, m serviços existentes e $U_i\,(t)$ denominando a porcentagem da capacidade total de cada recurso usada pelo serviço i no intervalo t, a seguinte condição deve ser verdadeira tanto para o processador, como para a rede:

$$P(\sum_{i=1}^{m+1} U_i(t) < n * VR) \geq C \tag{4.1}$$

   Ou seja, a probabilidade que a soma do uso do recurso durante qualquer intervalo t para todos os serviços admitidos, mais o uso do recurso do novo serviço, seja menor que o valor de referência VR (em todo o cluster) deve ser mantida alta (maior que um valor de confiança C, onde $0 < C < 1$). A Equação 4.1 assegura que as reservas de recursos feitas para todos os serviços devem ser mantidas com alta probabilidade após a admissão do novo serviço no cluster. O valor de referência usado na Equação 4.1 é uma porcentagem da capacidade total de cada nó. Essa *margem de segurança* permite ao Agente 86 no nó de serviço: (i) aceitar picos de uso do recurso pelo serviço e (ii) trabalhar com variações no escalonamento dos recurso pelo MMV. A probabilidade na Equação 4.1 é estimada usando a distribuição de probabilidades do uso de recurso $U\,(t) = \sum_i U_i\,(t)$ no lugar da série temporal dos valores amostrados de uso de recursos. Por exemplo, a probabilidade $P\,(U\,(t) < n * 90\%)$ é

aproximada para o percentil de amostras que são menores que 90%.

2. **Encontre o um nó de serviço com recursos disponíveis para executar o novo serviço.**

   O nó deve ser "adequado" para cada tipo de recurso (tempo de processador, espaço em memória e banda passante de rede) necessário para o serviço. A política utilizada define o número total de serviços atendidos pelo SMART. Inicialmente, analisando o problema de alocação de recursos do tipo memória apenas, sabe-se que as soluções mais utilizadas no Problema de Alocação de Memória Dinâmica [97] são o *first-fit* - aloca o primeiro bloco de memória grande o suficiente -, *best-fit* - aloca o menor bloco de memória grande o suficiente, e *worst-fit* - aloca o maior bloco de memória. As simulações demonstraram que *first-fit* e *best-fit* são melhores do que *worst-fit* em termos decisão mais rápida e utilização do recurso, respectivamente.

   O algoritmo, adotado em SMART, procura inicialmente o nó pelo recurso de memória e para cada nó encontrado, verifica a disponibilidade para os outros dois recursos (processador e rede). Caso não exista a quantidade de recurso suficiente, um novo nó é procurado para o primeiro recurso. Para diminuir o tempo da busca, usa-se a estratégia "first-fit", mas procura-se manter, para cada tipo de recurso, uma estrutura de dados com os nós de serviço, indexada na ordem decrescente da quantidade de recursos disponível.

   Caso não encontre nenhum nó que tenha a disponibilidade de todos os recursos, é usado o algoritmo de alocação dinâmica, que explora a migração da MV entre os nós de serviço. Esse algoritmo é inspirado no algoritmo MS (*Migrate the Smallest process*) apresentado em [98]. O serviço é submetido ao primeiro nó que possuir mais espaço em memória disponível e suficiente para a admissão do serviço. Antes da admissão, o algoritmo deve escolher um serviço do conjunto que está sendo executado no nó. Esse serviço deve possuir o menor conjunto de trabalho em memória (que representa o menor *overhead* de migração) e ao mesmo tempo que libere uma quantidade de recursos de processador e rede suficientes para a admissão do novo serviço. O algoritmo de admissão é aplicado ao serviço escolhido para sair do nó.

4.1.3 Migração

Em razão das migrações requererem tempo de processador e banda de rede, elas interferem no alocação dos recursos no sistema. Se esses efeitos não forem considerados, podem resultar na perda do isolamento de desempenho e degradação da QoS dos serviços. Os recursos para a execução das operações do Agente 86 também precisam ser controlados.

4.2 Tolerância a Falhas

Um cluster está sujeito a falhas de hardware e software, mas por usar máquinas independentes é possível construir um sistema que isole estas falhas, de modo a obter uma degradação suave dos serviços hospedados. Esse sistema deve buscar a detecção de falhas comuns e se recuperar com o mínimo ou nenhuma intervenção humana. Para obter o isolamento das falhas, deve-se aproveitar a redundância natural do cluster para maximizar a confiabilidade e mascarar as falhas transientes no sistema. Assim como cada máquina no cluster é independente das outras, busca-se obter o mesmo com os componentes em software.

Cada componente em software deve ser independente, o que significa, poder restabelecer o estado a qualquer momento e em qualquer outro nó do sistema. Por exemplo, o componente *Control* executa em uma MV. Se algum problema for detectado no nó, Control pode migrar para qualquer outro nó (SMART ou de Serviço) disponível no sistema. A migração do Control deve manter o seu estado, caso seja necessário reiniciar o agente, o estado corrente pode ser reconstruído consultando os outros componentes do sistema. O componente Agente 99 pode ser iniciado em qualquer outra máquina existente e sua MV, executando o serviço monitorado pode ser migrada para a nova máquina. O componente Agente 86 existe em todos os nós de serviço e apresenta redundância no sistema.

Sistemas tradicionais utilizam a técnica de tolerância a falha "ativo/passivo" [99] para assegurar a disponibilidade do sistema. Esta técnica envolve replicar cada componente do sistema: um dos clones atua como um processo primário e participa ativamente no sistema, enquanto o outro é o secundário e simplesmente monitora o primário para espelhar seu estado. Se o ativo falhar de alguma forma,

o passivo assume as suas tarefas e cria seu próprio secundário. No lugar de uma configuração "ativo/passivo" para tolerância a falhas, todos os componentes da arquitetura SMART são pares para tolerância a falhas, e cada um monitora o outro, através de *heartbeat*, para assegurar que todos os componentes necessários estejam sempre ativos. O componente *Control* é o gerente central para a monitoração do funcionamento do sistema. Ele age como um par para os outros, mas monitora e reinicia os nós de serviço, o nó Agente 99 e o *Front-End de serviço* em caso de uma falha total. De forma similar, as instâncias do Agente 86 monitoram o *Control* e o reinicializa em caso de falha. Enquanto houver pelo menos um componente ativo no sistema, ele pode eventualmente regenerar todos os outros componentes.

4.2.1 Falha do *Control*

Todas as instâncias do Agente 86 colaboram para monitorar o *Control*. Quando o *heartbeat* do *Control* é perdido, um algoritmo distribuido é iniciado. Cada Agente 86 inicia um espera randômica. O Agente que terminar primeiro envia uma mensagem de aviso, em um canal multicast, para todos os outros Agentes 86, liberando estes da espera. O *Control* deve ser reinicializado temporariamente no nó Agente 99, preferencialmente, mas também pode executar em qualquer nó de serviço que tenha recursos disponíveis. Enquanto o *Control* estiver morto, o sistema não recebe nenhum pedido de serviço novo, mas os restantes que estão executando não são afetados. Quando o *Control* se torna ativo novamente, todos os Agentes 86 recebem o *heartbeat* e se registram com ele e reenviam as informações do estado do nó. Caso seja criado mais de um *Control* no sistema, ao receber o *heartbeat* do outro, o mais novo (relativo ao *timestamp* do *heartbeat*) comete suicídio.

4.2.2 Falhas no Nó de Serviço

O Agente 86 monitora o hardware do nó de serviço. Ao ocorrer qualquer anormalidade (elevação da temperatura, falha em alguma ventoinha, falhas excessivas no consumo de recursos de uma MV) fora de uma referência específica para o tipo de recurso, um aviso é enviado ao *Control* e um processo de migração das MVs para outros nós de serviço é iniciado. Esse processo é implementado utilizando o algoritmo de alocação dinâmica dos recursos no cluster. A alocação dinâmica explora

as facilidades oferecidas pelo MMV como a migração da MVs entre os servidores do cluster para balancear carga entre os nós do cluster. Ao acionar a alocação dinâmica, os serviços no nó que está apresentando falhas pode ser movido para outros nós que disponham dos recursos necessários.

Se ocorrer uma falha não detectada no nó de serviço, tão logo Control percebe a falta do *heartbeat* daquele nó, ele tentará restabelecer a execução das respectivas MVs em outros nós (. No caso de serviços *stateful*, uma alternativa é manter um *checkpoint* da imagem da memória da MV em intervalos pré-definidos. Essa configuração pode ser controlada pelo *front-end* do serviço, através de uma API oferecida pelo Control.

4.3 Sumário

Nesse capítulo foi apresentada uma arquitetura para a construção de uma nova plataforma para serviços *streaming*, baseada em virtualização. A arquitetura SMART oferece a multiplexação de recursos em um cluster para muitos serviços, mantendo o isolamento encontrado em plataformas de cluster dedicado. SMART permite descobrir os recursos e a alocação necessária para manter o QoS desejado pelo serviço. Através da migração das MVs, o sistema permite a alocação dinâmica de recursos para serviços. SMART explora a redundância dos nós do cluster para oferecer tolerância a falhas e estende esta propriedade usando a capacidade de migração das MVs. No próximo capítulo é demonstrado um estudo de caso - o Smart-GloVE, um serviço de streaming de vídeo para Internet, implementado tendo como base a arquitetura proposta. Além das vantagens apresentadas nesse capítulo, Smart-GloVE demonstra como a arquitetura SMART pode também prover a escalabilidade em um cluster para uma aplicação que originalmente não foi projetada para executar em cluster de computadores.

Capítulo 5

Smart-GloVE

A arquitetura SMART, apresentada no capítulo anterior, provê uma estrutura que pode ser usada para a construção de servidores escaláveis para serviços que necessitam de garantias de desempenho de tempo-real. Nesse capítulo é demonstrado o uso de SMART para implementar um serviço escalável de distribuição de vídeo sob demanda na rede.

5.1 GloVE

O sistema GloVE [100, 101, 102] oferece uma solução de distribuição de vídeo sob demanda para atender grandes audiências. A principal contribuição do GloVE é criar uma cache de vídeos em memória e gerenciar os blocos de vídeo, armazenados nessa cache, de forma eficiente. Uma solução de distribuição de vídeo procura reduzir a demanda por vazão na rede do servidor de vídeo. São utilizados *distribuidores*, que atuam como *proxies* dinâmicos para os fluxos de vídeo entre o servidor e os clientes. A Figura 5.1 mostra um conjunto de distribuidores GloVE atendendo os clientes de uma rede de acesso xDSL. Cada distribuidor recebe o vídeo do servidor, agindo como um cliente e executa o serviço de *streaming* para vários clientes reais. O conteúdo armazenado no distribuidor é determinado de acordo com as requisições que chegam dos clientes, mudando dinamicamente. O papel de um distribuidor é evitar que o vídeo selecionado por um cliente tenha que trafegar pela dorsal da Internet desde o servidor até o cliente final. O distribuidor mantém em memória parte do vídeo que está sendo transmitido para o cliente. Os próximo pedidos, de outros clientes para

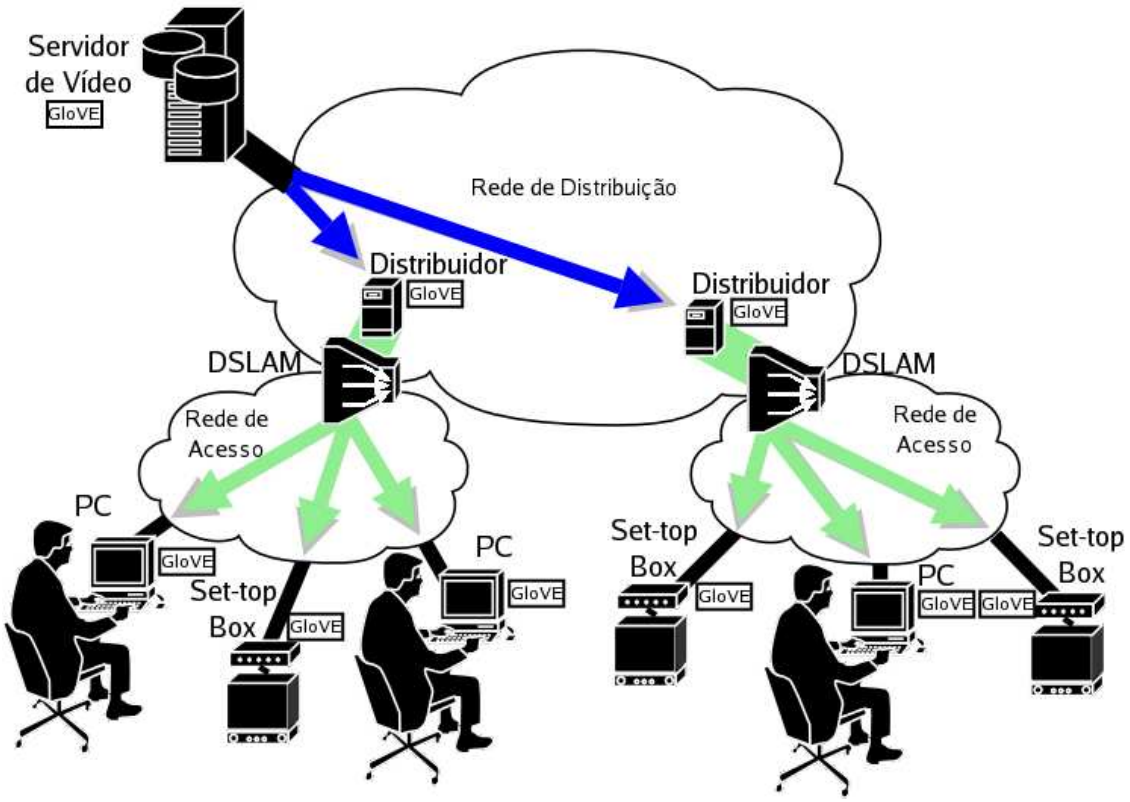

Figura 5.1: Arquitetura GloVE

o mesmo vídeo, podem ser atendidos através dos blocos armazenados em memória, evitando criar novos fluxos de vídeo entre o servidor e o distribuidor.

O distribuidor opera armazenando em memória e reciclando os fluxos de vídeos recebidos do servidor de maneira tal que quanto mais clientes assistirem a um mesmo vídeo, mais eficiente será o uso da rede. Além disso, o distribuidor não utiliza buffers dedicados para cada cliente. Todos os clientes compartilham a mesma estrutura de dados que organiza os blocos de vídeo armazenados em memória. O gerenciamento dos blocos de vídeo, em memória, pelo algoritmo de compartilhamento do GloVE permite que cada cliente possa interagir individualmente com o vídeo assistido.

No modelo adotado em GloVE, vídeos populares aumentam a eficiência do sistema, uma vez que concentram clientes assistindo o vídeo em pontos próximos. Assim, aumenta a probabilidade do bloco de vídeo já estar disponível em um *buffer* do distribuidor, levando a uma redução do tráfego entre o distribuidor e o servidor e a demanda pelos recursos de rede do servidor. A demanda transfere-se para os

recursos (processador, memória, vazão de rede) encontrados no distribuidor. O Distribuidor GloVE possui um algoritmo bastante eficiente para manter um serviço de *streaming* para cada cliente conectado. Além disso, ele permite o *streaming* de vídeos com taxas de codificação e duração distintas. A alocação da memória é dinâmica - apenas estabelecendo um limite máximo de *cache* em memória para cada vídeo na configuração inicial -, possibilitando o armazenamento de novos blocos ou reciclagem de blocos de vídeo existentes no *streaming* de cada cliente. Os clientes não armazenam o vídeo localmente, apenas possuem um pequeno buffer para esconder o *jitter* da rede.

A versão original do GloVE é uma implementação monolítica de um software que gerencia o compartilhamento de fluxos de vários vídeos entre seus clientes. Essa implementação garante a QoS para os clientes de cada vídeo transmitido, mas não é eficiente ao ser usada em um cluster de computadores. A criação de um novo distribuidor em uma máquina com mais recursos, para adicionar mais clientes a um vídeo já atendido em uma máquina sobrecarregada, gera pelo menos um novo fluxo entre o servidor e o distribuidor. Outra alternativa é migrar o vídeo, e seus clientes, para a nova máquina. Na Figura 5.2 mostrar um gráfico para o tempo necessário para a migração em função do número de clientes assistindo o vídeo. Na migração, os clientes perdem a conexão de rede com o distribuidor. Quando todos os clientes tentam se reconectar no novo distribuidor cria um contenção na interface de rede, o que contribui para aumentar ainda mais a duração da migração.

A arquitetura SMART provê a escalabilidade ao sistema GloVE sem criar novos fluxos de vídeo. Através do mapeamento da QoS exigida por cada distribuidor (de acordo com os parâmetros do vídeo como taxa de transmissão e duração do vídeo) no perfil de uso dos recursos de uma MV, SMART consegue executar vários distribuidores em um servidor, aproximando-se da mesma quantidade de vídeos e clientes atendida na solução original do GloVE. O MMV realisa a multiplexação fina das várias instâncias do distribuidor, garantindo o desempenho de tempo-real. A capacidade de migrar o distribuidor viabiliza a alocação dinâmica dos recursos, junto com o balanceamento de carga, em todo o cluster.

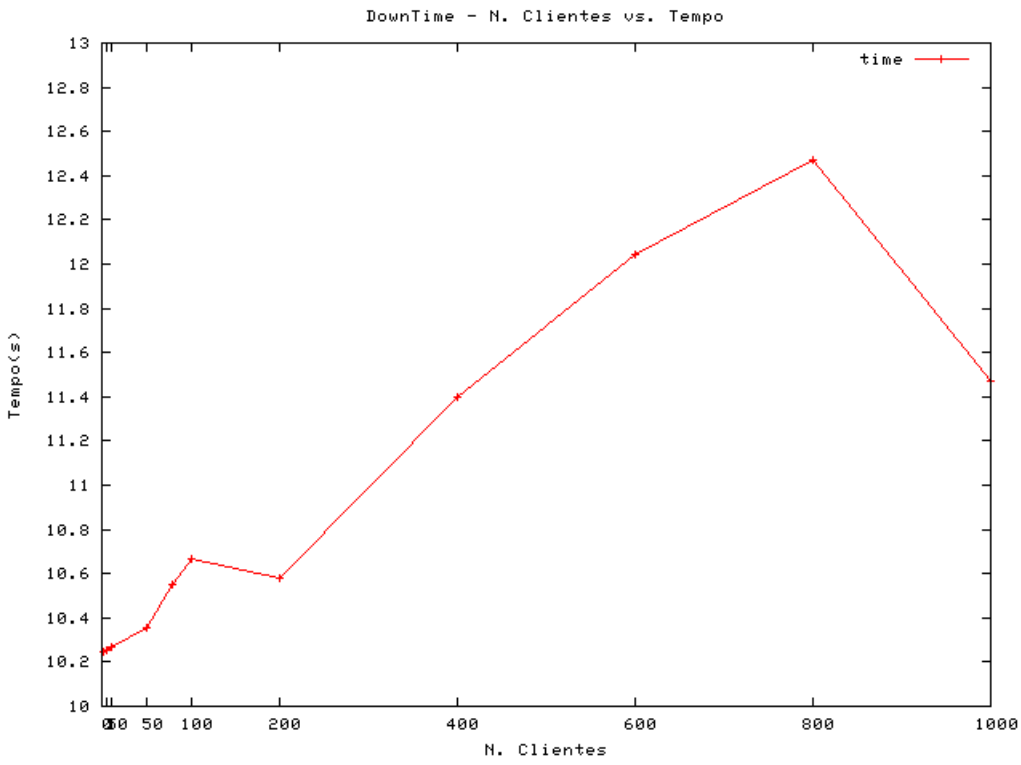

Figura 5.2: Duração da Migração de um Distribuidor GloVE

A cada novo pedido de um cliente o controle de admissão encontra um nó para executar o distribuidor, caso seja o primeiro pedido para um vídeo ou encontra o nó de onde o distribuidor correspondente está recebendo o vídeo. Caso o nó esteja no limite da capacidade (mas se ainda existir recursos globalmente no cluster), o algoritmo de alocação dinâmica pode migrar uma instância do distribuidor para uma outra máquina com mais recursos disponíveis. A migração ocorre de forma transparente para o serviço, a menos de alguns segundos de suspensão do *streaming* (dependendo principalmente do tamanho da cache de vídeo). A conexões dos clientes com o distribuidor se mantém. Veja a Seção 3.3 no Capítulo 3 para a descrição de como as conexões se mantém na migração.

5.2 Implementação

Nessa seção, é analisada a implementação de Smart-GloVE. O sistema GloVE possui um componente que aloca o distribuidor mais adequado (por proximidade na rede ou menor carga, entre outras métricas) para atender a requisição do cliente. O Supervisor GloVE, como é chamado, também realiza outras tarefas gerenciais no

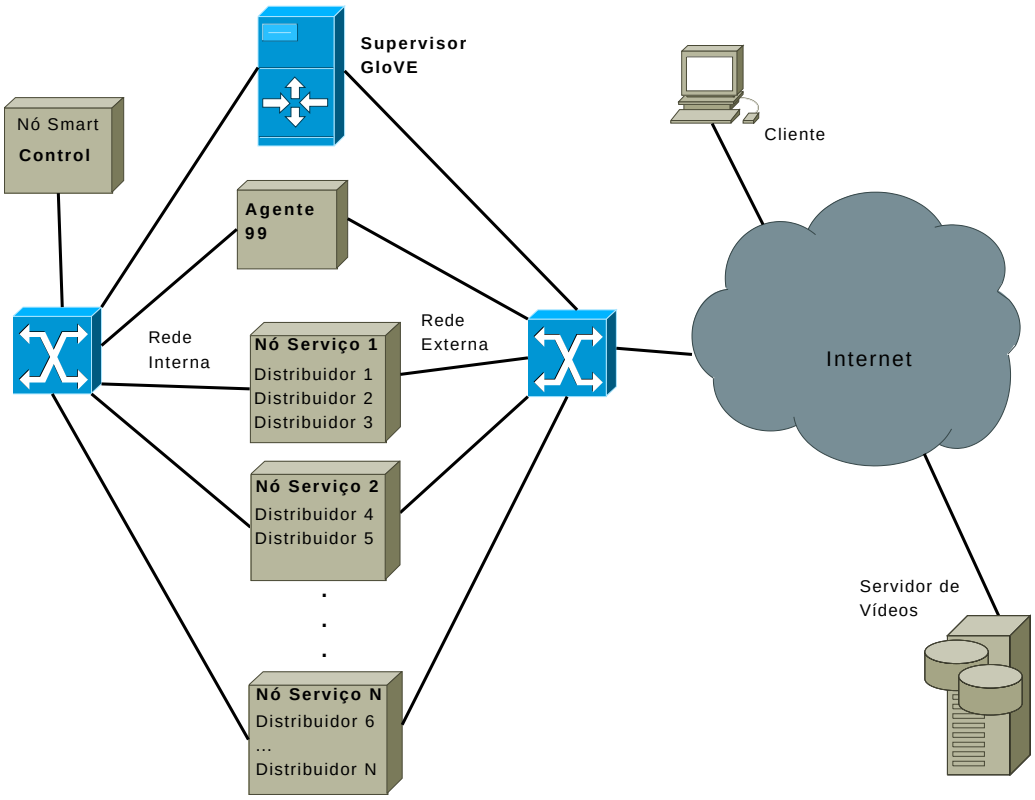

Figura 5.3: Arquitetura Smart-GloVE

sistema, mas a alocação do distribuidor é o mais importante para o sistema Smart-GloVE. Assim, o componente *Front-End* de Serviço na arquitetura SMART recebe as funções do Supervisor, ver Figura 5.3. Ao receber uma requisição de cliente, o Supervisor após realizar o processamento necessário, repassa para o *Control*. Nessa etapa, o *Control* descobre se existem os recursos necessários para atender o pedido e envia ao Supervisor o resultado. O Supervisor também informa ao *Control* sobre a saída de clientes do sistema e o término da operação de uma distribuidor. Estas informações são necessárias ao *Control* para a admissão dos clientes ao serviço e alocação dos recursos no sistema. Nenhuma outra alteração no sistema GloVE é necessária, além da troca de informações entre o Supervisor e *Control*.

5.2.1 Monitor de Máquina Virtual

A arquitetura proposta em SMART não depende de um modelo de virtualização específico, tanto a virtualização em hardware, como no SO podem ser usadas. Como

foi analisado na Seção 3.4, em termos de desempenho, as implementações de virtualização tanto no nível do hardware e como no SO são semelhantes para aplicações *cpu-bound*. Enquanto, para as aplicações focadas em E/S, a virtualização no SO é mais eficiente na utilização dos recursos do sistema e por isso alcança melhor desempenho. A virtualização no nível do SO permite a execução de um número maior de MVs que as implementações em hardware. Os MMVs como VMware e Xen apresentam uma densidade de MVs por causa da sobrecarga (consumo de memória, falhas de cache L2 etc) causada na execução de múltiplos kernels. Os sistemas Xen (representa o modelo de virtualização em hardware) e OpenVZ (virtualização em SO) foram testados para o uso na implementação de Smart-GloVE.

Para todos os testes a serem analisados a seguir, o ambiente experimental está descrito na Tabela 5.1.

Software - SO e MMV	
SO	CentOS 5.1
OpenVZ	2.6.18-53 SMP
Xen	3.0.3-41.el5
Hardware - Servidores	
Processador	dual Intel Pentium D 3.0 GHz
Memória	2 GB
Rede Interna	1 Gb Ethernet (efetivo: 800 Mbps)
Rede Externa	100 Mb Ethernet (efetivo: 80 Mbps)
Vídeo (20% em memória)	
A Marca do Zorro (ação)	taxa: 400 Kbps; duração: 2 h

Tabela 5.1: Ambiente Experimental

Para o mesmo ambiente experimental, com os distribuidores transmitindo o vídeo para um cliente, é possível executar 12 MVs Xen. Enquanto o MMV OpenVZ permite a execução de 24 MVs. A razão principal para essa diferença é tamanho da imagem da MV Xen, pois para a execução de uma distribuição simples do SO CentOS é necessário pelo menos 64MB de memória. O tamanho da imagem chega a 140MB, quando adicionado o conteúdo da cache do vídeo e outras estruturas de dados do GloVE. O OpenVZ compartilha o mesmo kernel entre todas as MVs, assim

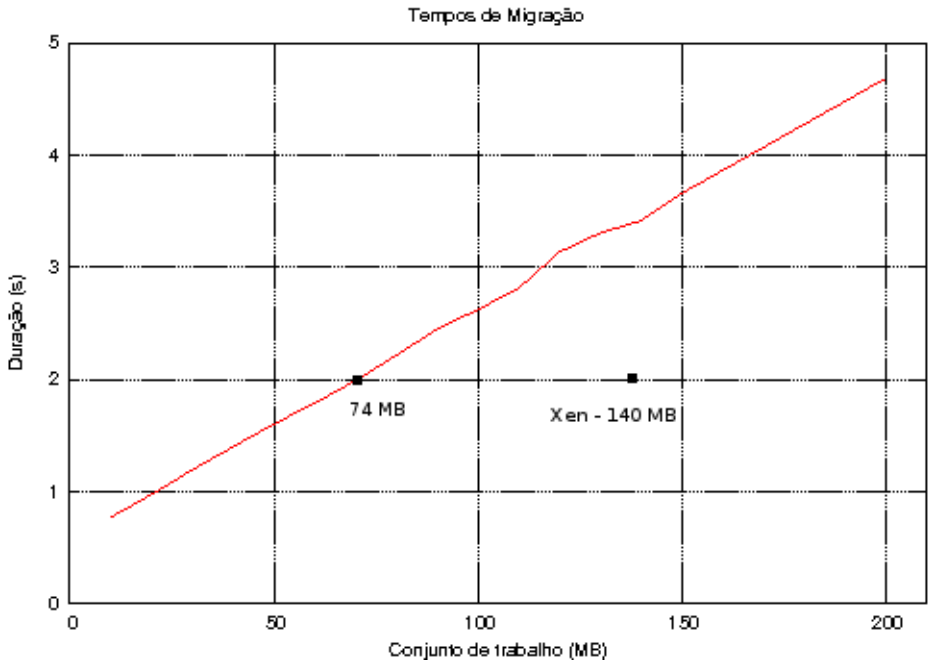

Figura 5.4: Duração da Migração x Tamanho da Imagem no OpenVZ

o tamanho da imagem para cada MV chega próximo de 74MB (cache de vídeo + estrutura de dados do distribuidor GloVE e SO).

O gráfico da Figura 5.4 mostra a duração da migração para vários tamanhos de imagem de uma MV OpenVZ. A migração de MV Xen (com 140MB) está assinalada com um ponto no gráfico, dura 2 segundos, e é 23% menor que a duração da migração para o mesmo tamanho de imagem no OpenVZ. Mas, para o mesmo vídeo, o tamanho da imagem no OpenVZ é menor. Assim, como podemos ver no gráfico, OpenVZ gasta também 2 segundos para migrar um distribuidor transmitindo o mesmo vídeo. A migração mais rápida no Xen, é devida à técnica *Pré-Cópia*, que é mais eficiente do que o *Suspend/Resume* usado pelo OpenVZ (ver Seção 3.3 para uma descrição da técnicas de migração). A migração original no OpenVZ envolve a cópia da imagem em disco entres os nós de serviço. Para otimizar a cópia, usa-se um partição montada em NFS em uma terceira máquina comum aos nós de serviço.

5.3 Determinação do Perfil de Uso dos Recursos

Através do Agente 99, foi feita a caracterização do uso dos recursos em uma execução real do distribuidor, transmitindo um vídeo de 400 Kbps, com 2 horas de duração, para 1 cliente e 10 clientes. O conjunto de trabalho, em memória principal, para esse vídeo está relacionado com a cache de 20% da duração do vídeo, configurada no distribuidor. Para o vídeo transmitido, o tamanho da cache é aproximadamente 72 MB. A versão atual do GloVE trabalha com a configuração estática da cache. A quantidade de vídeo cacheado em memória não varia com a quantidade de clientes, ou com as diferentes partes que são transmitidas do vídeo para cada cliente.

Usando as informações encontradas na fase de caracterização do uso do processador, mostradas nos gráficos das Figuras 5.5 e 5.6, e da rede, nos gráficos das Figuras 5.7 e 5.8, podemos estabelecer a quantidade de recursos necessária para a transmissão do um vídeo e a para a admissão de um cliente ao distribuidor. A caracterização de uso foi feita usando intervalos de 1 segundo. Podemos observar que na transmissão do vídeo ocorrem rajadas de uso, tanto no processador como na rede. Mas, pela distribuição de probabilidade do uso (ver Tabela 5.2, referente ao uso do processador e Tabela 5.3, referente ao uso da rede), sabemos que para 1 cliente, o uso do processador não passa de 1% em aproximadamente 94% da transmissão do vídeo, e o uso da rede não ultrapassa 1% em 90% dos intervalos medidos. O uso da rede aumenta linearmente para os 10 clientes, ver o gráfico na Figura 5.8. O consumo do processador não aumenta significativamente com os 10 clientes, conforme podemos observar no gráfico da Figura 5.6.

O escalonamento do processador no OpenVZ utiliza um algoritmo *fair-share*, estabelecendo garantia mínima de uso do processador de acordo com a proporção configurada para cada distribuidor. Uma MV esta livre para consumir além da quota estabelecida se não existe nenhuma outra MV mais consumindo o recurso. Um limite de uso máximo pode ser estabelecido no escalonamento do recurso para os distribuidores, permitindo a obediência ao limite estabelecido na admissão do cliente (ver a Equação 4.1, para a admissão do serviço).

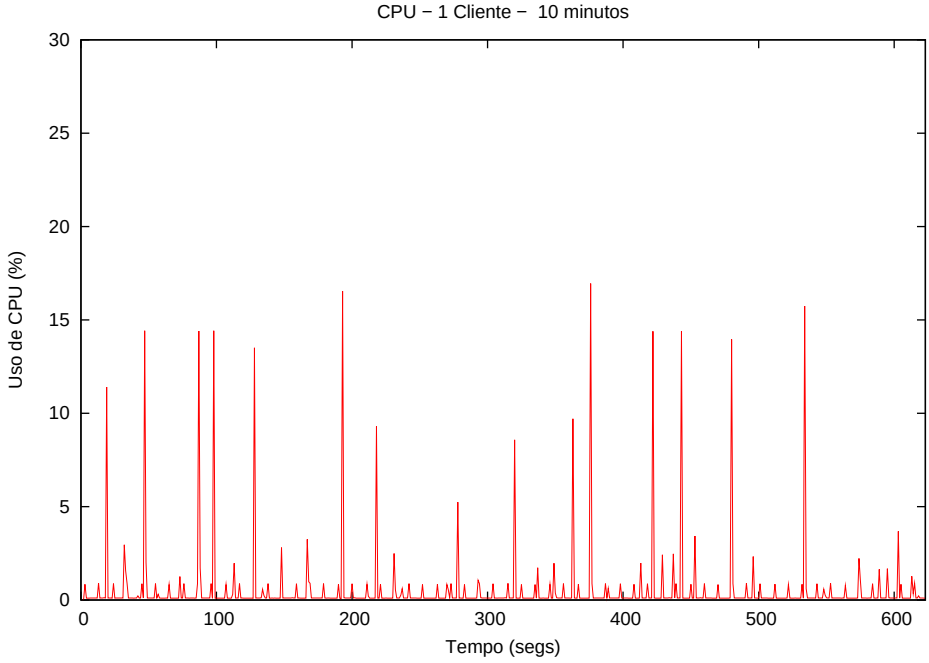

Figura 5.5: Uso do Processador - 1 Cliente

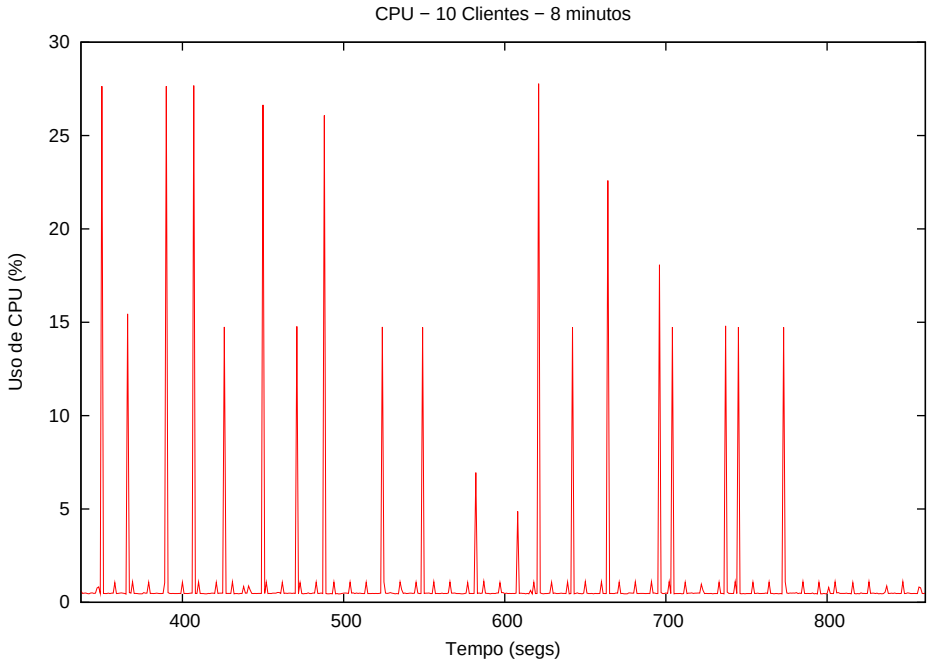

Figura 5.6: Uso do Processador - 10 Clientes

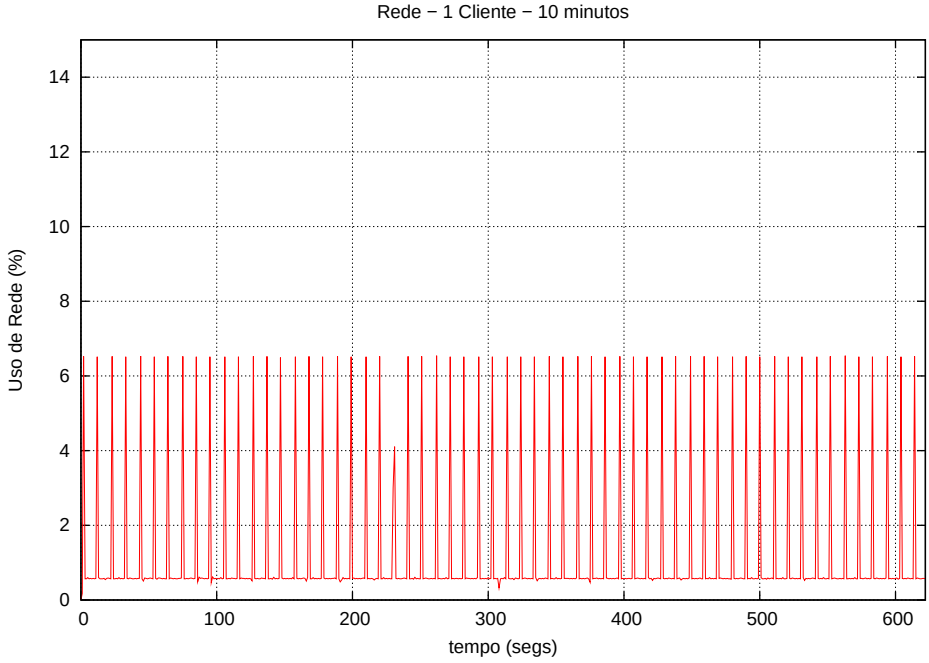

Figura 5.7: Uso de Rede - 1 Cliente

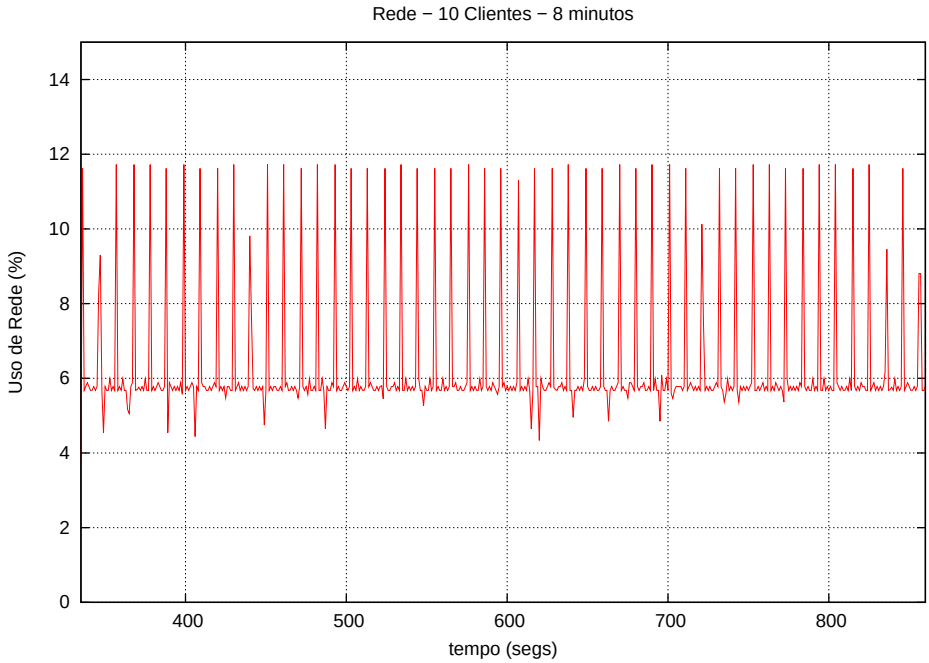

Figura 5.8: Uso de Rede - 10 Clientes

Distribuição - Processador	
1 Cliente	
0 - 1%	94%
1 - 5%	4%
5 - 20%	2%
10 Clientes	
0 - 1%	93%
1 - 5%	5%
5 - 30%	2%

Tabela 5.2: Distribuição do Uso do Processador

Distribuição - Rede	
1 Cliente	
0 - 1%	90%
5 - 10%	9%
10 Clientes	
0 - 1%	38%
5 - 10%	54%

Tabela 5.3: Distribuição do Uso da Rede

OpenVZ não oferece um escalonador proporcional para o escalonamento do recurso de rede. O controle de admissão é feito contabilizando a proporção de uso de banda passante na interface de rede para cada cliente admitido e também configurando a quantidade de *buffers* de transmissão para cada distribuidor de acordo com o número de clientes possuído. A quantidade de *buffers* possuída por cada distribuidor estabelece a proporção de uso da interface de rede.

5.4 Migração

Por ser uma aplicação *soft realtime*, o serviço de VoD coloca um desafio extra para a solução com migração. O intervalo de tempo entre a suspensão do serviço e o

re-início no outro nó não pode ser muito longo. Se um distribuidor falha, todos os clientes atendido por ele deixarão de receber os blocos de vídeo. A migração deve ocorrer em um intervalo suficientemente pequeno de maneira que não afete o conteúdo do *buffer* no cliente, necessário para manter a exibição do vídeo.

A duração do *prefetch*[1] e a taxa de transferência do vídeo recebido são atualmente determinantes para o tamanho do *buffer* no cliente. A duração da migração pode também ser um dos fatores a serem considerados no cálculo do *prefetch* necessário. Nos experimentos realizados foi usado o tamanho de buffer no cliente original (2 min de vídeo), não levando em conta o intervalo de migração no cálculo do tamanho, apenas os parâmetros do vídeo e da rede.

A Figura 5.4 mostra o gráfico da duração da migração para diverso tamanhos de cache de vídeo. Por exemplo, OpenVZ consegue migrar um distribuidor transmitindo um vídeo de 400 Kbps e 2 horas (72 MB em cache) em 2 segundos. Esse tempo é suficiente para manter a QoS da transmissão para o cliente, sem precisar alterar o tamanho padrão do *buffer* encontrado no cliente. Foi observado que um intervalo de até 3 segundos é permitido pelo buffer do cliente, para a taxa de 400 Kbps. Como discutido acima, o cálculo do tamanho do *buffer* também pode ser estabelecido adicionando a duração da migração como mais um parâmetro. Devemos ressaltar que a maioria dos vídeos assistidos, atualmente, nos serviços mais populares na internet, estão abaixo de 2 horas de duração. No site *YouTube* [2], por exemplo, 99% dos vídeos assistidos são menores que 30 MB (em média 8,4 MB) e duram menos de 700 segundos [103].

O distribuidor GloVE pode atender vários clientes assistindo o vídeo em momentos diferentes. A quantidade de clientes atendidos implica na quantidade de fluxos transmitidos na interface de rede e no trabalho exigido do distribuidor para manter a temporização necessária na transmissão do vídeo para cada cliente (pois a transmissão deve obedecer a taxa do vídeo). Assim, procuramos conhecer o efeito da variação da quantidade de clientes, atendidos pelo distribuidor, na duração da migração. Observamos no gráfico da Figura 5.9 que é praticamente nula a interferência da quantidade de clientes na duração da migração.

[1] quantidade de vídeo armazenada no buffer do cliente antes do início da reprodução

[2] www.youtube.com

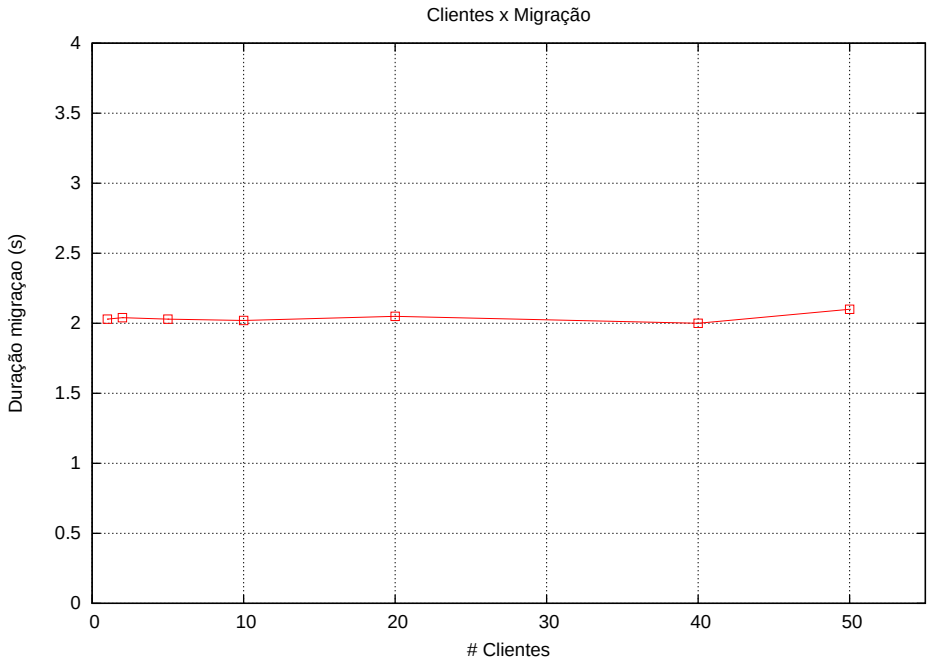

Figura 5.9: Número de Clientes X Duração da Migração

5.5 Isolamento

O isolamento de desempenho entre os distribuidores foi medido em dois testes. No primeiro teste, executamos dois distribuidores em um nó de serviço. O primeiro distribuidor, transmitindo um vídeo para um cliente, foi migrado com tamanhos de cache variados, enquanto o segundo distribuidor transmitia o vídeo para 20 clientes. Entre os tamanhos do sistema migrado (imagem da memória) de 70 MB e 150 MB não ocorreu nenhuma interferência entre os distribuidores.

Em outro teste, observou-se o comportamento do isolamento de desempenho ao executar o máximo de distribuidores e atender o máximo de clientes em um nó de serviço. O número máximo de distribuidores é determinado pelo quantidade de MVs (SO + distribuidor) suportada em um nó de serviço. O número máximo de clientes atendidos é determinado pela capacidade da interface de rede usada. Na configuração adotada para o nó de serviço (descrita na Tabela 5.1, é possível executar 24 distribuidores com 72 MB de vídeo na cache cada um. Esse mesmo nó, usando uma interface de rede Ethernet de 100 Mbps, atende até 200 clientes de um vídeo

de 400 Kbps. Utilizando uma interface de rede Ethernet de 1 Gbps, foi possível migrar um distribuidor entre dois nós de serviço em um intervalo de 2 segundos. Através da monitoração das mensagens de status dos clientes e com um cliente real assistindo a transmissão do vídeo, confirmamos os devidos escalonamento e alocação dos recursos, entre os distribuidores, necessários para manter a reprodução correta do vídeo.

5.6 Algoritmo de Admissão

O algoritmo de admissão de SMART foi implementado com as seguintes adaptações:

1. Se o vídeo ainda não está sendo transmitido por nenhum distribuidor, executa-se normalmente o algoritmo descrito na Seção 4.1.2;

2. Se o vídeo está sendo transmitido por algum distribuidor, executa-se o algoritmo MS, onde o cliente é aceito no distribuidor que está transmitindo o vídeo, e um outro distribuidor (com a menor cache de vídeo, e que esteja usando recursos suficientes para a admissão do novo cliente) é escolhido e transferido para outro nó de serviço.

Essa implementação utiliza uma árvore de perdedores, para o recurso de memória (escolhido como referência principal), que apresenta uma seqüência decrescente destes recursos disponíveis em cada nó de serviço. A árvore serve como um oráculo para o recurso, que apresenta o melhor nó para o algoritmo *worst-fit*. Caso o nó não coincida com a disponibilidade dos outros recursos (processador, rede), o próximo nó na árvore é escolhido. No pior caso, a busca por todos os nós candidatos será O(n). A atualização da árvore é feita em $O(\log n)$.

5.7 Sumário

Esse capítulo demonstrou a de um serviço real de distribuição de vídeos na Internet usando a arquitetura SMART. A principal contribuições do SMART para o sistema GloVE é prover a escalabilidade no cluster através da virtualização dos recursos sem criar novos fluxos de vídeo. Quando uma máquina real estiver sobrecarregada, uma

instância do distribuidor (em uma MV) pode migrar para uma outra máquina com mais recursos disponíveis.

Foi demonstrado como obter o perfil de uso dos recursos do serviço de *streaming*. A configuração da MV de acordo com o perfil obtido ainda é feita pelo operador. Um trabalho futuro é obter os mesmos dados através do próprio MMV.

Foi demonstrado que o MMV mantém o isolamento de desempenho necessário para garantir a QoS desejada no *streaming* do vídeo.

Capítulo 6

Conclusões e Trabalhos Futuros

Esta tese demonstrou que é possível desenvolver um sistema que através de mecanismos de virtualização ofereça qualidade de serviço necessária para a hospedagem compartilhada de serviços de *streaming* de conteúdo multimídia. O sistema provê o provisionamento e alocação eficiente e dinâmica de recursos em um conjunto de máquinas simples. A seguir, apresentamos as contribuições da tese e uma discussão sobre trabalhos futuros.

6.1 Contribuições

No desenvolvimento do trabalho dessa tese, podemos citar as seguintes contribuições:

1. A formulação de uma metodologia para o mapeamento da QoS, desejada por um serviço, em um perfil do uso dos recursos.

2. Uma arquitetura para a construção de uma nova plataforma para serviços de *streaming*, baseada em virtualização. A arquitetura SMART oferece a multiplexação de recursos em um cluster para muitos serviços, mantendo o isolamento encontrado em plataformas de cluster dedicado. SMART permite descobrir os recursos e a alocação necessária para manter o QoS desejado pelo serviço. Através da migração das MVs, o sistema provê a alocação dinâmica de recursos para serviços e melhora balanceamento de carga no sistema.

3. A apresentação de uma solução escalável e confiável para um sistema real de distribuição de vídeos na Internet, baseado na arquitetura SMART;

6.2 Direções Futuras

Deve-se testar SMART para outros serviços, além da distribuição de vídeos. Uma plataforma para servir jogos foi avaliada. A duração da migração com a técnica usada em OpenVZ é uma barreira para obter um sistema com QoS para os jogos avaliados. Duas alternativas são possíveis: (i) desenvolver uma nova técnica de migração para OpenVZ e (ii) avaliar outros sistemas de virtualização, como KVM, por exemplo.

Apesar do experimentos feitos na implementação do sistema Smart-GloVE, comprovarem a viabilidade do uso da virtualização para a construção de uma plataforma de serviço escaláveis, não foi testado o algoritmo de admissão proposto. O próximo passo é executar a simulação e analisar o comportamento do algoritmo de admissão para vídeo com taxas e durações variadas.

O uso da energia elétrica é um fator muito importante, que deve ser considerado em todas as plataformas de serviços na Internet atualmente. O controle do consumo de energia em SMART deve ser um aspecto tão importante quanto a escalabilidade e a tolerância a falhas.

O uso de MVs em muitos aspectos se assemelha com as propostas de microkernels. O MMV pode finalmente tornar o microkernel eficiente? Com certeza, uma investigação nesse caminho pode trazer contribuições relevantes.

Referências Bibliográficas

[1] CHUN, B., CULLER, D., ROSCOE, T., et al., "PlanetLab: An Overlay Testbed for Broad-Coverage Services", *ACM SIGCOMM Computer Communication Review*, v. 33, n. 3, pp. 205–225, 2003.

[2] FOSTER, I., KESSELMAN, C., NICK, J., et al., *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*, Tech. rep., Global Grid Forum, junho 2002.

[3] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., et al., "OceanStore: An Architecture for Global-Scale Persistent Storage". In: *Proc. 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS2000)*, pp. 190–201, 2000.

[4] KOTSOVINOS, E., MORETON, T., PRATT, I., et al., "Global-scale service deployment in the XenoServer platform". In: *Proc. of First Workshop on Real, Large Distributed Systems (WORLDS '04)*, pp. 56–65, 2004.

[5] "Wikipedia, the free encyclopedia", http://en.wikipedia.org, Junho, 2008.

[6] "Ask Jeeves - ask.com", http://www.ask.com/, Junho, 2008.

[7] "ebay online auctions", http://www.ebay.com/, Junho, 2008.

[8] "Google Maps", http://maps.google.com/, Junho, 2008.

[9] "webUnRisk: Interactive Online Tool", http://library.wolfram.com, Junho, 2008.

[10] BARATTO, R., POTTER, S., SU, G., et al., "MobiDesk: Mobile Virtual Desktop Computing". In: *Proc. of 10th Annual ACM International Conference on Mobile Computing and Networking (MobiCom 2004)*, pp. 23–31, 2004.

[11] SAPUNTZAKIS, C., LAM, M. S., "Virtual Appliances in the Collective: A Road to Hassle-Free Computing". In: *Proc. 9th Workshop on Hot Topics in Operating System*, pp. 110–122, 2003.

[12] CHASE, J., VAHDAT, A., WILKES, J., "Back to the Future: dependable computing = dependable services". In: *Proc. 10th European SIGOPS Workshop*, pp. 432–440, 2002.

[13] "Showcase for tomorrow's computing utility", http://www.hpl.hp.com/se3d/se3d-background.html, Junho, 2008.

[14] "The N1 Grid Service - A True Computing Utility", http://blogs.sun.com/roller/page/jonathan, Junho, 2008.

[15] "Amazon Web Services", http://aws.amazon.com, Julho, 2008.

[16] "Google App engine", http://code.google.com/appengine/, Julho, 2008.

[17] CHENG, X., DALE, C., LIU, J., *Understanting the Characteristics of Internet Short Video Sharing: Youtube as a Case Study*, [cs.NI] arXiv:0707.3670v1, School of Computing Science, Simon Fraser University, Canada, Julho 2007.

[18] TANG, W., FUN, Y., CHERKASOVA, L., et al., "Modeling and Generating Realistic Streaming Media Server Workloads", *Computer Networks: The International Journal of Computer and Telecommunications Networking*, v. 51, n. 1, pp. 336–356, Janeiro 2007.

[19] CHERKASOVA, L., TANG, W., "Providing Resource Allocation and Performance Isolation in a Shared Streaming-Media Hosting Service". In: *Proc. of the 2004 ACM Symposium on Applied Computing*, pp. 1213–1218, 2004.

[20] BREWER, E. A., "Lessons from Giant-Scale Services", *IEEE Internet Computing*, v. 5, n. 4, pp. 46–55, 2001.

[21] ANDERSON, T., CULLER, D., PATTERSON, D., "A Case for NOWS (Network of Workstations)", *IEEE Micro*, v. 12, n. 1, pp. 54–64, fevereiro 1995.

[22] FOX, A., GRIBBLE, S., CHAWATHE, Y., et al., "Cluster-Based Scalable Network Services". In: *Proc. 16th ACM Symp. on Operating Systems Principles (SOSP'97)*, pp. 78–91, 1997.

[23] SAITO, Y., BERSHAD, B. N., LEVY, H. M., "Manageability, Availability and Performance in Porcupine: a Highly Scalable, Cluster-based Mail Service". In: *Proc. 17th ACM Symp. on Operating Systems Principles (SOSP'99)*, pp. 1–15, 1999.

[24] GRIBBLE, S. D., WELSH, M., BREWER, E. A., et al., "The Multispace: an Evolutionary Platform for Infrastructural Services". In: *Proc. 1999 USENIX Annual Technical Conference*, pp. 12–22, 1999.

[25] "Yahoo! Everything", http://everything.yahoo.com/, Julho, 2008.

[26] MANI, A., NAGARAJAN, A., "Understanding Quality of Service for Web Services", http://www.ibm.com/developerworks/library/ws-quality.html, Janeiro 2002.

[27] FAN, X., WEBER, W., BARROSO, L. A., "Power Provisioning for a Warehouse-sized Computer". In: *Proceedings of the 34th Annual International Symposium on Computer architecture*, pp. 13–23, ACM, 2007.

[28] JIANG, X., XU, D., "SODA: a Service-On-Demand Architecture for Application Service Hosting Utility Plataforms". In: *Proc. of the 12th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-12)*, p. 174, 2003.

[29] APPLEBY, K., FAKHOURI, S., FONG, L., et al., "Oceano - SLA Based Management of a Computing Utility". In: *Proc. 7th IFIP/IEEE Int'l Symp. on Integrated Network Management*, pp. 855–868, 2001.

[30] CHASE, J. S., IRWIN, D. E., GRIT, L. E., et al., "Dynamic Virtual Clusters in a Grid Site Manager". In: *Proc. 12th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC'03)*, p. 90, 2003.

[31] ARPACI-DUSSEAU, A., CULLER, D. E., "Extending Proportional-Share Scheduling to a Network of Workstations". In: *Proc. Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pp. 957–973, 1997.

[32] URGAONKAR, B., SHENOY, P., "Sharc: Managing CPU and Network Bandwidth in Shared Clusters", *IEEE Trans. on Parallel and Distributed Systems*, v. 15, n. 1, pp. 2–17, 2004.

[33] ARON, M., DRUSCHEL, P., ZWAENEPOEL, W., "Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers". In: *Proc. ACM Sigmetrics - Int'l Conference on Measurement and Modeling of Computer Systems*, pp. 456–472, 2000.

[34] BANGA, G., DRUSCHEL, P., MOGUL, J. C., "Resource Containers: a new facility for resource management in server systems". In: *Proc. USENIX 3rd Symp. on Operating Systems Design and Implementation (OSDI'99)*, pp. 211–223, 1999.

[35] WHITAKER, A., SHAW, M., GRIBBLE, S. D., "Scale and Performance in the Denali Isolation Kernel". In: *Proc. USENIX 5th Symp. on Operating Systems Design and Implementation (OSDI'02)*, pp. 195–209, 2002.

[36] GOYAL, P., VIM, H., CHENG, H., "Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks". In: *Proc. ACM SIGCOMM*, p. 294, 1996.

[37] WALDSPURGER, C., WEIHL, W., "Lottery Scheduling: Flexible Proportional-Share Resource Management". In: *Proc. USENIX 1st Symp. on Operating Systems Design and Implementation (OSDI'94)*, pp. 653–665, 1994.

[38] JONES, M., ROSU, D., ROSU, M., "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities". In: *Proc. 16th ACM Symp. on Operating Systems Principles (SOSP'97)*, pp. 25–35, 1997.

[39] LESLIE, I., MCAULEY, D., BLACK, R., et al., "The Design and Implementation of an Operating system to Support Distributed Multimedia Applications", *IEEE J. Selected Areas in Communication*, v. 14, n. 7, pp. 1280–1297, 1996.

[40] WALDSPURGER, C., WEIHL, W., *Stride Scheduling: Deterministic Proportional-Share Resource Management*, Tech. Rep. TM-528, Massachusetts Institute of Technology, MIT Laboratory for Computer Science, 1995.

[41] DEMERS, A., KESHAV, S., SHENKER, S., "Analysis and Simulation of a Fair Queueing Algorithm". In: *Proc. of SIGCOMM Symposium*, pp. 1–12, 1989.

[42] BERTSEKAS, D., GALLAGER, R., *Data Networks*. Prentice-Hall, 1987.

[43] GOYAL, P., GUO, X., VIN, H. M., "A Hierarchical CPU Scheduler for Multimedia Operating System". In: *Proc. USENIX 5th Symp. on Operating Systems Design and Implementation (OSDI'02)*, pp. 6–18, 1996.

[44] NIEH, J., LAM, M., "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications". In: *Proc. 16th ACM Symp. on Operating Systems Principles (SOSP'97)*, pp. 117–163, 1997.

[45] BRUNO, J., BRUSTOLONI, J., GABBER, E., et al., "Disk Scheduling with Quality of Service Guarantees". In: *Proc. of IEEE ICMCS Conference*, p. 400, 1999.

[46] SHENOY, P., VIN, H., "Cello: A Disk Scheduling Framework for Next Generation Operating Systems". In: *Proc. ACM SIGMETRICS - Int'l Conference on Measurement and Modeling of Computer Systems*, pp. 203–219, 1998.

[47] DUDA, K., CHERITON, D., "Borrowed Virtual Time (BVT) Scheduling: Supporting Latency-sensitive Threads in a General-Purpose Scheduler". In: *Proc. 17th ACM Symp. on Operating Systems Principles (SOSP'99)*, pp. 543–561, 1999.

[48] DRUSCHEL, P., BANGA, G., "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems". In: *Proc. USENIX 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pp. 20–32, 1996.

[49] VERGHESE, B., GUPTA, A., ROSENBLUM, M., "Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors". In: *Proc. 8th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pp. 181–192, 1998.

[50] WALDSPURGER, C., "Memory Resource Management in VMWare ESX Server". In: *Proc. USENIX 5th Symp. on Operating Systems Design and Implementation (OSDI'02)*, pp. 181 – 194, 2002.

[51] REUMANN, J., MEHRA, A., SHIN, K., et al., "Virtual Services: A New Abstraction for Server Consolidation". In: *Proc. USENIX Ann. Technical Conf.*, p. 623, 2000.

[52] SULLIVAN, D., SELTZER, M., "Isolation with Flexibility: A Resource Management Framework for Central Servers". In: *Proc. 2000 USENIX Technical Conf.*, pp. 27–27, 2000.

[53] LITZKOW, M., LIVNY, M., MUTKA, M., "Condor - A Hunter of Idle Workstations". In: *Proc. 8th International Conf. of Distributed Computing Systems*, pp. 18–29, 1988.

[54] HORI, A., TEZUKA, H., ISHIKAWA, Y., et al., "Implementation of Gang Scheduling on a Workstation Cluster". In: *Proc. Workshop Job Scheduling Strategies for Parallel Processing*, pp. 126–139, 1996.

[55] DUSSEAU, A., ARPACI, R., CULLER, D., "Effective Distributed Scheduling of Parallel Workloads". In: *Proc. ACM SIGMETRICS - Int'l Conference on Measurement and Modeling of Computer Systems*, pp. 25–36, 1996.

[56] BARAK, A., LA'ADAN, O., "The MOSIX Multicomputer Operating System for High Performance Cluster Computing", *Future Generation Computer Systems*, v. 13, n. 4–5, pp. 361–372, 1998.

[57] AMAR, L., BARAK, A., LEVY, E., et al., "An On-line Algorithm for Fair-Share Node Allocations in a Cluster". In: *Proc. 7-th IEEE Int. Symposium on Cluster Computing and the Grid (CCGrid '07)*, pp. 83–91, 2007.

[58] SHEN, K., TANG, H., YANG, T., et al., "Integrated Resource Management for Cluster-based Internet Services". In: *Proc. USENIX 5th Symp. on Operating Systems Design and Implementation (OSDI'02)*, pp. 225 – 238, 2002.

[59] GOVIL, K., TEODOSIU, D., HUANG, Y., et al., "Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors". In: *Proc. 17th ACM Symp. on Operating Systems Principles (SOSP'99)*, pp. 229 – 262, 1999.

[60] CHASE, J., ANDERSON, D., THAKAR, P., et al., "Managing Energy and Server Resources in Hosting Centers". In: *Proc. 18th ACM Symp. on Operating Systems Principles (SOSP'01)*, pp. 103–116, 2001.

[61] URGAONKAR, B., SHENOY, P., ROSCOE, T., "Resource Overbooking and Application Profiling in Shared Hosting Plataforms". In: *Proc. USENIX 5th Symp. on Operating Systems Design and Implementation (OSDI'02)*, pp. 239 – 254, 2002.

[62] FIGUEIREDO, R. J., DINDA, P. A., FORTES, J. A. B., "A Case For Grid Computing On Virtual Machines". In: *Proc. of the 23rd International Conference on Distributed Computing Systems*, pp. 550–560, 2003.

[63] FREEMAN, T., KEAHEY, K., FOSTER, I., et al., "Division of Labor: Tools for Growth and Scalability of Grids". In: *Proc. of 4th International Conference on Service-Oriented Computing*, pp. 63–71, 2006.

[64] MOSBERG, D., PETERSON, L. L., "Making Paths Explicit in The Scout Operating System". In: *Proc. USENIX 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pp. 153–167, 1996.

[65] OSMAR, S., SUBHRAVETI, D., G.SU, et al., "The Design and Implementation of Zap: A System for Migration Computing Environments". In: *Proc.*

USENIX 5th Symp. on Operating Systems Design and Implementation (OSDI'02), pp. 78–90, 2002.

[66] RUTH, P., RHEE, J., XU, D., et al., "Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure". In: *Proc. of The 3rd IEEE International Conference on Autonomic Computing*, pp. 32–44, 2006.

[67] GRIT, L., IRWIN, D., YUMEREFENDI, A., et al., "Virtual Machine Hosting for Networked Clusters: Building the Foundations for Autonomic Orchestration". In: *Proc. of Second International Workshop on Virtualization Technology in Distributed Computing*, pp. 203–213, 2006.

[68] "Dynamic Resource Scheduler", http://www.vmware.com/, Julho, 2008.

[69] GOLDBERG, R., "Survey of Virtual Machine Research", *IEEE Computer*, pp. 34–35, Junho 1974.

[70] SMITH, J., NAIR, R., "The Architecture of Virtual Machines", *IEEE Computer*, v. 38, n. 5, pp. 32–38, 2005.

[71] "Java Virtual Machine Specification", http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html, Maio, 2008.

[72] "Standard ECMA-335, Common Language Infrastructure (CLI)", http://www.ecma-international.org/publications/standards/Ecma-335.htm, Julho, 2008.

[73] POPEK, G., GOLDBERG, R., "Formal Requirements for Virtualizable 3rd Generation Architectures", *Communications of the ACM*, v. 17, n. 7, pp. 412–421, 1974.

[74] GOLDBERG, R., *Architectural Principles for Virtual Computer Systems*, Ph.D. Thesis, Harvard University, 1972.

[75] "VMware ESX", http://www.vmware.com/products/vi/esx/, Abril, 2008.

[76] "Sun xVM", http://www.sun.com/software/products/xvm/index.jsp, Abril, 2008.

[77] BARHAM, P., DRAGOVIC, B., FRASER, K., et al., "Xen and the Art of Virtualization". In: *Proc. 19th ACM Symp. on Operating Systems Principles (SOSP'03)*, pp. 164–177, 2003.

[78] "VirtualBox", http://www.virtualbox.org/wiki/VirtualBox_architecture, Abril, 2008.

[79] ROBIN, J., IRVINE, C., "Analysis of Intel Pentium's Ability to Support a Secure Virtual Machine Monitor". In: *Usenix Security Symposium*, p. 450, 2000.

[80] BUGNION, E., DEVINE, S., ROSENBLUM, M., "Disco: Running Commodity Operating Systems on Scalable Multiprocessors." In: *Proc. 16th ACM Symp. on Operating Systems Principles (SOSP'97)*, pp. 412–447, 1997.

[81] Sun Microsystems, *UltraSparc Virtual Machine Specification, Revision 1.0*, janeiro 2006.

[82] Intel, *IA-31 Intel Architecture Software Developer's Manual*, março 2006.

[83] AMD, *AMD64 Architecture Programmer's Manual, Volume 2: System Programming, Rev. 3.12*, setembro 2006.

[84] KAY, J., LAUDER, P., "A Fair Share Scheduler", *Communications of ACM*, v. 31, n. 1, pp. 44–55, janeiro 1988.

[85] SUGERMAN, J., VENKITACHALAM, G., LIM, B., "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor". In: *Proc. of Usenix Annual Technical Conference*, pp. 1–14, 2002.

[86] VICTOR, J., *Solaris Containers Technology Architecture Guide*, Sun Microsystems, rev 1.1 ed., maio 2006.

[87] "Linux VServer Paper", http://linux-vserver.org/Paper, Maio, 2008.

[88] "FreeBSD Jails", http://www.freebsd.org, Maio, 2008.

[89] "Virtuozzo Containers 4", http://www.parallels.com/en/products/virtuozzo/, Maio, 2008.

[90] Parallels, *OpenVZ User's Guide*, version 2.7.0-8 ed., 2005.

[91] "CFQ IO Scheduler", http://en.wikipedia.org/wiki/CFQ, Maio, 2008, author: Jens Axboe.

[92] NELSON, M., LIM, B.-H., HUTCHINS, G., "Fast Transparent Migration for Virtual Machines". In: *USENIX Annual Technical Conference*, pp. 391–394, 2005.

[93] SOLTESZ, S., POTZL, H., FIUCZYNSKI, M. E., et al., "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors". In: *Proc. of the 2nd EuroSys Conference*, pp. 275–288, 2007.

[94] PADALA, P., ZHU, X., WANG, Z., et al., *Performance Evaluation of Virtualization Technologies for Server Consolidation*, Tech. Rep. 59, HP Laboratories Palo Alto, 2007.

[95] WANG, Z., ZHU, X., PADALA, P., et al., "Capacity and Performance Overhead in Dynamic Resource Allocation to Virtual Containers". In: *Proc. of the IFIP/IEEE Symposium on Integrated Management*, p. 226, 2007.

[96] "Linux Trace Toolkit Next Generation", http://ltt.polymtl.ca, Julho, 2008, Ecóle Polytechnique de Montreal.

[97] KNUTH, D. E., *The Art of Computer Programming, Volume1 : Fundamental Algorithms*. Segunda edição ed. Addison-Wesley, Reading, 1973.

[98] BARAK, A., BRAVERMAN, A., "Memory ushering in a scalable computing cluster". In: *Proc. IEEE Third Int. Conf. on Algorithms and Architecture for Parallel Processing*, pp. 897 –907, 1997.

[99] EVAN MARCUS, H. S., *Blueprints for High Availability: Designing Resilient Distributed Systems*. John Wiley & Sons, 2000.

[100] ISHIKAWA, E., *Memória Cooperativa Distribuída para Sistemas de VoD*, Ph.D. Thesis, COPPE/UFRJ, outubro 2003.

[101] ISHIKAWA, E., AMORIM, C. L., "Collapsed Cooperative Video Cache for Content Distribution Networks". In: *Proc. of the Brazilian Simposium on Computer Networks*, pp. 249–264, maio 2003.

[102] BRAGATO, L. S., *Implementação e Avaliação de um Sistema de Vídeo sob Demanda Baseado em Cache Cooperativa Colapsada de Vídeo*, Master's Thesis, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, setembro 2006.

[103] GILL, P., ARLITT, M., LI, Z., et al., "Youtube traffic characterization: a view from the edge". In: *Proc. of the 7th ACM SIGCOMM Conference on Internet Measurement*, pp. 15–28, ACM, 2007.