Máquinas Virtuais como Suporte ao Desenvolvimento de Aplicações Tolerantes a Falhas

Tórgan F. de Siqueira [1] , Taisy S. Weber [1] , Reynaldo Novaes [2] , Ingrid Jansch-Porto [1] *

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul – Porto Alegre, RS

[2]Hewlett-Packard do Brasil – Porto Alegre, RS

{torgan,taisy,ingrid}@inf.ufrgs.br, reynaldo.novaes@hp.com

Resumo. *Máquinas virtuais têm sido empregadas no compartilhamento de recursos de* hardware *em servidores e estações de desenvolvimento. Suas características incluem alto desempenho, suporte a múltiplos sistemas operacionais virtualizados, estabilidade, segurança e possibilidade de injeção de falhas. Visando acelerar o ciclo de desenvolvimento de aplicações tolerantes a falhas, analisa-se as características das máquinas, UML, FAUmachine e XEN, e sua adequação quanto a desempenho, transparência, portabilidade, intrusividade, instrumentabilidade, monitorabilidade e transportabilidade.*

1. Introdução

O conceito e as primeiras implementações de máquinas virtuais (MV) são conhecidos desde meados da década de 60, primeiro como cenário de desenvolvimento, e posteriormente levados à indústria através dos conhecidos IBM System 370 (S/370) e IBM System 390 (S/390) [Kohlbrenner et al., 2004]. Atualmente, são utilizadas como servidores e estações de desenvolvimento, tendo impulso tanto na indústria [VMware Inc., 2004] quanto na academia ([Dike, 2004], [FAUmachine, 2004], [XEN, 2004], [Plex86, 2004], [Liang et al., 2003] e [Thain and Livny, 2003], por exemplo). Como servidores, as MVs destinam-se a compartilhar recursos de *hardware* entre diversos sistemas operacionais (SO), ou cópias destes, e cada instância executa um serviço específico. Entretanto, muitas máquinas podem e são usadas como ferramentas de suporte ao desenvolvimento e depuração, tanto de SOs quanto de aplicações.

O objetivo deste artigo é identificar as características de algumas máquinas virtuais e sua adequação ao desenvolvimento de aplicações tolerantes a falhas para Grids. No âmbito do projeto DependableGrid (UFRGS/ HP Brasil P&D), uma das metas é salvar e restaurar o estado de aplicações, seja no próprio computador após *crash* seguido de recuperação, ou em outro computador do Grid. Neste contexto, as propriedades avaliadas foram escolhidas considerando todo o ciclo de desenvolvimento das aplicações e de seus mecanismos de tolerância a falhas e validação, o que abrange níveis distintos, indo do kernel do SO a aplicações Java distribuídas.

Na seção 2, são descritas as propriedades mencionadas, em função do ambiente de desenvolvimento. Na seção 3, são apresentadas as características gerais e de três máquinas virtuais: UML, FAUmachine e XEN. A seção 4 apresenta uma comparação entre as MVs de acordo com as propriedades, e as conclusões são apresentadas na seção 5.

2. Ambientes de Desenvolvimento de Aplicações Tolerantes a Falhas

O objetivo do uso de MVs para desenvolver de aplicações tolerantes a falhas é acelerar o ciclo de desenvolvimento. Isto se alcança, por exemplo, permitindo o estudo de vários

*Este trabalho foi desenvolvido conjuntamente com a HP Brasil P&D

cenários sem precisar reconfigurá-los a cada vez; utilizando uma infraestrutura de *hardware* comum e variando o SO; reaproveitando instalações de distribuições; confinando testes de rede e assim eliminando a interferência entre sistemas não-relacionados.

Dentre as características que um ambiente de desenvolvimento de aplicações tolerantes a falhas apresenta, algumas recebem mais atenção quando da avaliação de MVs: desempenho, transparência, portabilidade, intrusividade, instrumentabilidade, monitorabilidade e transportabilidade. Estas características foram sendo anotadas ao longo de experiências de desenvolvimento, acrescidas recentemente pela necessidade de distribuição de aplicações na plataforma *Grid*.

O **desempenho** é necessário para não penalizar a máquina hospedeira. As medidas principais são o tempo de carga da MV e o de execução das aplicações dentro desta, além do impacto sobre as demais instâncias em execução.

A **transparência** diz respeito à percepção da MV pela aplicação em desenvolvimento. A inserção da MV não deve alterar o desenvolvimento e/ou características da aplicação. Neste caso, por exemplo, injeta-se falhas na MV e estuda-se o comportamento/resposta da aplicação, evitando a injeção de falhas no sistema hospedeiro. Injetar falhas na MV também permite que execuções concorrentes não só continuem, mas também sejam avaliadas em caso de falha daquela que foi abortada, particularmente útil em sistemas distribuídos sob teste em ambiente confinado.

A **portabilidade** de uma MV é desejável, pois permite o desenvolvimento de aplicações portáveis desde os estágios iniciais, quando houver esta necessidade. A portabilidade diz respeito à plataforma hospedeira em termos de *hardware*/SO nativo.

A **intrusividade** refere-se ao custo de inserir a MV no desenvolvimento da aplicação. Idealmente, a MV não deve ser intrusiva, isto é, a aplicação deve ter seu desenvolvimento independente do nível inferior ser um sistema nativo ou uma MV.

A **instrumentabilidade** e **monitorabilidade** da MV referem-se ao grau em que se pode injetar falhas e monitorar o estado do sistema. Estão entre as mais importantes, pois determinam em que extensão uma MV é utilizável em um ciclo de desenvolvimento.

Por fim, a **transportabilidade** é um item a ser considerado em caso de desenvolvimento de sistemas distribuídos, *Grids* e *middleware*. Espera-se que, se a MV não for transportável entre dois hospedeiros, que pelo menos no nível de aplicações (ou processos) isto seja possível.

3. Máquinas Virtuais

Máquinas virtuais podem ser classificadas em dois grupos: as que virtualizam o conjunto completo de instruções da arquitetura, denominadas ISA-VM (*Instruction Set Architecture Virtual Machine*), e as máquinas que suportam uma interface binária para aplicações, denominadas ABI-VM (*Application Binary Interface Virtual Machine*), que virtualizam chamadas de sistema [Figueiredo et al., 2003].

Máquinas que usam o mesmo ISA têm melhor desempenho do que as que têm diferentes ISA, pois não precisam traduzir código em tempo de execução ou modificar o código binário da aplicação. As máquinas virtuais clássicas são ISA-VMs que suportam a execução de sistemas operacionais completos, como IBM S/390, VMware e Plex86. Estas máquinas suportam a execução concorrente de vários sistemas SOs sobre o mesmo *hardware*. Assim, três máquinas foram escolhidas: a XEN, que embora faça paravirtualização, se aproxima de uma ISA, e a FAUmachine e UML, do segundo grupo. Máquinas comerciais foram descartadas, pois procura-se uma solução em *software* livre.

3.1. UML - User-Mode Linux

O UML é uma MV que executa sobre o SO hospedeiro permitindo que um usuário não-privilegiado a instancie e que se torne privilegiado dentro desta. A virtualização ocorre através da interceptação das chamadas de sistema, e a máquina não tem acesso ao *hardware* nativo, exceto se permitido. Do ponto de vista do usuário, o UML é um sistema Linux completo, e do ponto de vista do hospedeiro, é uma aplicação de usuário. Correntemente, o UML executa apenas sobre Linux, mas há planos de portá-lo para outras plataformas [Buchacker and Sieh, 2001], [Höxer et al., 2002], e [Dike, 2004].

O UML executa os binários do hospedeiro sem necessidade de recompilação ou religação. Tudo que a máquina cria fica confinado a ela, não se propagando para outras instâncias ou hospedeiro. Seus parâmetros de memória e disco podem ser modificados dinamicamente, e possui escalonador e sistema de memória virtual próprios, usando o sistema hospedeiro somente para acessos ao *hardware*. O escalonador nativo somente implementa a decisão feita no correspondente do UML. Os dispositivos UML são mapeados para os correspondentes reais no sistema hospedeiro. Com UML, é possível virtualizar um rede inteira dentro de um hospedeiro, inclusive isolando-a da rede física, se desejado. A gerência da máquina pode ser feita localmente, remotamente ou via *daemons*.

Cada instância UML utiliza um espaço de endereçamento próprio, na forma de arquivos temporários no hospedeiro. Estes arquivos podem ser mapeados para memória via *tmpfs*. A utilização de memória apresenta alguns problemas, já que cada instância UML se comporta exatamente como um *kernel*, isto é, não libera memória a não ser que seja solicitado a fazê-lo, mantendo dados em *cache*. Já existe uma alternativa que usa *mmap* e um dispositivo virtual, */dev/anon*, reduzindo a utilização de memória física do hospedeiro, mas ainda não é totalmente estável.

Uma máquina UML é vista como um arquivo no sistema nativo, e corresponde a uma imagem de um disco, contendo partições e arquivos da MV. Se um hospedeiro precisa instanciar um número grande de máquinas, este esquema é oneroso em termos de armazenamento, já que as imagens de discos são da ordem de gigabytes (3 a 4, para um sistema razoavelmente completo). A solução é utilizar *copy-on-write* (COW), com um sistema-base comum a todas as instâncias e mantendo as escritas individuais de cada uma em arquivos separados. Se necessário, pode-se incorporar estas cópias separadas ao sistema-base posteriormente. Como desvantagens, a instanciação de várias máquinas UML pode provocar instabilidade no sistema nativo, e a queda do hospedeiro provoca a queda de todas as instâncias. Processos dentro da UML executam com desempenho levemente inferior se comparado à execução no hospedeiro. Devido à construção da UML, uma instância pode se apropriar dos ciclos de processador, degradando o desempenho das demais.

O UML foi construído originalmente para trabalhar com *tracing threads* (TT), sendo cada processo da MV mapeado para um processo no hospedeiro. A TT rastreia chamadas de sistema nos processos da UML, intercepta-as e desvia o fluxo de execução para o *kernel* UML. O problema com este esquema é a possibilidade dos processos poderem acessar o hospedeiro, uma vez que estão no seu espaço de endereçamento. Além disso, este mecanismo é lento, pois há quatro chaveamentos de contexto, uma entrega e um retorno de sinal. Para resolver este problema, um novo mecanismo foi implementado, o *Separate Kernel Address Space* (SKAS). Este mecanismo requer modificação no *ptrace* do *kernel* hospedeiro, mas é pouco intrusivo, segundo a documentação. O SKAS utiliza dois chaveamentos de contexto e é mais rápido do que as TTs, e a abertura de */proc/mm* inicia um espaço de endereçamento novo e vazio para os processos UML [Dike, 2004]. Do ponto de vista da UML, o SKAS melhora o desempenho, mas não modifica sua funcio-

nalidade, enquanto que do ponto de vista do hospedeiro, os processos UML desaparecem, restando apenas os quatro essenciais, e há redução na carga do processador.

3.2. FAUmachine

O projeto FAUmachine derivou do UML, e é semelhante ao VMware ou VirtualPC. Esta máquina também executa como processo de usuário, e é restrita à arquitetura i386. Atualmente executa sobre o Linux, mas portes para OpenBSD e Windows já estão em progresso. Esta máquina é utilizada como plataforma experimental no projeto europeu DBench, que pesquisa *benchmark* de dependabilidade [FAUmachine, 2004], [Sieh and Buchacker, 2002].

Esta MV é iniciada por um carregador adaptado e executa um *kernel* Linux ligeiramente modificado. As características de memória, discos, cdrom e rede são configuráveis no momento da criação da máquina. Assim como no UML, é possível instalar uma distribuição Linux sobre a MV. Existe suporte à rede e a conexão pode ser feita transparentemente. A compatibilidade binária com respeito ao hospedeiro é mantida. A FAUmachine é acompanhada do *Experiment Controller Expect* (ECE), que pode executar experimentos completos a partir de um *script* e injetar falhas no *hardware* virtual.

A injeção de falhas pode ocorrer no núcleo de processamento, periféricos ou rede. Para efeitos da MV, o nível de *hardware* é o *kernel* hospedeiro, e a injeção de falhas é feita via interface *ptrace*. As possíveis falhas ocorrem: no núcleo de processamento, nos periféricos, nos componentes de rede externos, na configuração do sistema, no ambiente, na interação e na operação. Este conjunto permite avaliar a resposta do sistema sob carga pesada de processamento, e evita que se tenha que injetar falhas no SO nativo. O código do injetor e o da MV são separados.

O acesso ao processador é direto, isto é, não simula uma CPU. A memória RAM é simulada via arquivo mapeado em memória, e a unidade de gerência de memória é simulada via *mmap* e *munmap*. Os periféricos, incluso os de rede, também são simulados, e os *drivers* de dispositivos são construídos sobre chamadas de sistema ao SO nativo. O teclado e a console são implementados por um *xterm*, enquanto que dispositivos de rede são implementados por chamadas *socket*. A rede o o roteamento são configurados à parte, permitindo interação transparente com a rede real. *Traps* (interrupções e exceções) são implementados por sinais, e o relógio de tempo real usa *gettimeofday()*.

A injeção de falhas é disparada temporalmente, de forma pré-programada ou aleatória, e as falhas são configuradas em um arquivo descritivo. Utilizando-se o ECE, não é necessário interação humana com o injetor, e uma instância do ECE é suficiente para todas as máquinas virtuais de um hospedeiro. A FAUmachine não implementa proteção de memória do *kernel*, e o processo *Tracer* é também o injetor.

3.3. XEN

O XEN é um monitor para máquinas virtuais (*Virtual Machine Monitor* - VMM), que suporta a arquitetura x86. Ele foi concebido para suportar múltiplos SOs virtuais, e atualmente suporta Linux, com portes para Windows XP e NetBSD parcialmente concluídos [XEN, 2004], [Barham et al., 2003] e [Fraser et al., 2003].

Diferente das duas MVs anteriores, este VMM foi construído visando alto desempenho e forte isolação entre SOs virtuais. Também neste caso um usuário instancia um VMM, que exporta uma interface para binários. O XEN não utiliza virtualização total da máquina hospedeira, mas um mecanismo denominado paravirtualização, no qual a abstração de *hardware* oferecida aos SOs virtualizados é similar, mas não igual ao *hardware* real. A virtualização total imporia penalidades maiores ao desempenho. Em vista disto, os SOs virtualizados precisam ter seu código-fonte portado para o XEN.

O mecanismo de funcionamento do XEN retira o *kernel* virtual do nível 0 (onde os SOs normalmente executam) e o transfere para o nível 1, passando ele próprio a executar neste nível. As aplicações continuam a executar no nível 3 (menos privilegiado). Cada SO virtualizado está contido e executa em um domínio. O XEN, propriamente, é denominado *hypervisor* e executa num domínio especial, Domínio0, criado na inicialização da máquina hospedeira. A virtualização oferecida pelo XEN divide-se em três áreas: gerência de memória, processador e periféricos. A gerência de memória é a parte mais complexa de tratar, tanto no *hypervisor* quanto no SO portado, em função de idiossincrasias da arquitetura x86. A virtualização do processador requer que o SO portado execute num nível menos privilegiado, neste caso, o nível 1. Instruções privilegiadas que o SO tenta executar são validadas pelo XEN, e falham caso o SO tente executá-las diretamente. Exceções, incluso falha de memória e *traps* de *software*, são virtualizadas por tabelas de manipuladores. A maior penalidade imposta ao desempenho ocorre nas chamadas de sistema e nas falhas de páginas de memória. Os dispositivos de entrada e saída (E/S) não são emulados, mas abstraídos. Esta interface melhora o isolamento entre os SOs virtualizados, e é construída usando *buffers* assíncronos em memória compartilhada.

Assim como interrupções de *hardware*, o XEN suporta um mecanismo de entrega de interrupções de *software*, ou seja, notificações aos domínios, baseado em mapas de bits. O *hypervisor* permite apenas o controle básico de operações, e o controle e gerência do *hypervisor* e domínios são feitos por um *software* de controle que executa no Domínio0. Este *software* controla os outros domínios, inclusive sua criação e terminação, bem como interfaces virtuais de rede e dispositivos.

4. Comparativo

Quanto ao **desempenho**, dentre as MVs discutidas, o XEN é a que apresenta melhor resultado, haja visto sua construção. O UML e a FAUmachine apresentam desempenhos semelhantes, com leve vantagem do primeiro. Possivelmente, o que torna a FAUmachine mais lenta que o UML são suas extensões para injeção de falhas e monitoramento.

A **transparência** é dependente da aplicação em desenvolvimento: se estiver no nível do usuário, todas as MV são elegíveis. Porém, se a aplicação precisar interagir com o SO, dependendo do nível a que se deve chegar, deve-se considerar características individuais das MVs. De modo geral, o UML é menos intrusivo, a não ser pelo SKAS. A FAUmachine se posiciona num meio-termo e o XEN é o mais intrusivo de todos. A **portabilidade** depende da continuidade do desenvolvimento das próprias máquinas.

O UML é uma espécie de meta-arquitetura suportada no Linux, e deve ser possível executá-lo nas arquiteturas de *hardware* suportadas pelo Linux. O XEN foi desenvolvido para a arquitetura x86. Quanto ao SO virtualizado, UML é restrito ao Linux, o mesmo acontecendo com a FAUmachine, enquanto que o XEN já está portando NetBSD e Windows XP. Com relação ao SO hospedeiro, o UML prevê um futuro porte para outros SOs. Quanto ao XEN, ele é derivado de um *kernel* Linux, mas por ser um VMM esta característica não se aplica estritamente.

Referente à **intrusividade** em termos de aplicações, nenhuma MV é intrusiva. Com respeito ao *kernel*, o XEN é o mais intrusivo de todos.

Todas as MVs têm código-fonte aberto, e portanto a **instrumentabilidade** e **monitorabilidade** podem ser acrescidas às máquinas, se já não existirem. Em particular, a FAUmachine, por seus propósitos, já oferece um grau de instrumentabilidade maior.

Finalmente, na **transportabilidade** deve-se avaliar em que nível a operação deve ocorrer. Se for apenas a imagem da MV, a FAUmachine é transportável, assim como o

UML, ainda que o novo hospedeiro tenha que oferecer suporte a elas. O XEN não é facilmente transportável, pois envolveria o transporte do carregador e a partição hospedeira.

5. Conclusão

Dadas às características das máquinas, pode-se afirmar que são utilizáveis como ferramentas de suporte ao desenvolvimento de aplicações tolerantes a falhas. A escolha entre as máquinas apresentadas depende em parte da natureza da aplicação. Testes estão sendo conduzidos para aprofundar o conhecimento sobre as capacidades e limitações, bem como a adequação de cada máquina em função dos tipos de aplicações. A viabilidade de outras máquinas virtuais também está sendo considerada, e será direcionada, caso necessário, pelas limitações encontradas nas MVs aqui apresentadas.

Referências

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *Proceedings of the 19th ACM symposium on Operating systems principles*, pages 164–177.

Buchacker, K. and Sieh, V. (2001). Framework for testing the fault-tolerance of systems including os and network aspects. In *The 6th IEEE International Symposium on High-Assurance Systems Engineering*, pages 95–105.

Dike, J. (2004). The user-mode linux kernel home page. URL: http://user-mode-linux.sourceforge.net/.

FAUmachine (2004). Faumachine project. URL: http://www3.informatik.uni-erlangen.de/Research/UMLinux.

Figueiredo, R. J., Dinda, P. A., and Fortes, J. A. B. (2003). In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 550–559.

Fraser, K. A., Hand, S. M., Harris, T. L., Leslie, I. M., and Pratt, I. A. (2003). The xenoserver computing infraestructure. Technical Report 552, Computer Laboratory, University of Cambridge, Cambridge, United Kingdom.

Höxer, H.-J., Buchacker, K., and Sieh, V. (2002). Implementing a user-mode linux with minimal changes from original kernel. In *9th International Linux System Technology Conference*, pages 72–82, Köln, Germany.

Kohlbrenner, E., Morris, D., and Morris, B. (2004). The history of virtual machines. URL: http://cne.gmu.edu/itcore/virtualmachine/history.htm.

Liang, Z., Venkatakrishnan, V. N., and Sekar, R. (2003). Isolated program execution: An application transparent approach for executing untrusted programs. In *Proceedings of the 19th Annual Computer Security Applications Conference*, pages 182–191.

Plex86 (2004). The plex86 x86 virtual machine project. URL: http://www.plex86.org.

Sieh, V. and Buchacker, K. (2002). Umlinux - a versatile swifi tool. In *Proceedings of the 4th European Dependable Computing Conference on Dependable Computing*, pages 159–171.

Thain, D. and Livny, M. (2003). Parrot: Transparent user-level middleware for data-intensive computing. In *Workshop on Adaptive Grid Middleware – AGridM 2003*.

VMware Inc. (2004). Vmware. URL: http://www.vmware.com.

XEN (2004). Xen: The xen virtual machine monitor. URL: http://www.cl.cam.ac.uk/Research/SRG/netos/xen/.