Checkpoint, *Restore*, Migração de VM's OO

Tiago Filipe da Silva Garrochinho
tiago.garrochinho@ist.utl.pt
Número de aluno 57688

Instituto Superior Técnico, Portugal

Resumo Este documento apresenta uma solução para os mecanismos de *checkpoint/restore* e migração para as aplicações que se executam sobre máquinas virtuais para linguagens de programação orientadas a objectos (e.g. Java VM). Esta solução tem como objectivo oferecer um conjunto de propriedades que nenhuma outra solução fornece conjuntamente, das quais se destacam transparência, completude, portabilidade e eficiência. Os dados salvaguardados no conteúdo dos *checkpoints* e enviados em mensagens durante a migração devem ser representados de forma eficiente. Em particular, devem privilegiar o máximo de informação respectiva à aplicação em execução propriamente dita e conterem o mínimo possível de informação exclusiva da maquina virtual (que poderá ser reconstruída). Este documento propõe uma arquitectura como primeira visão da solução para este trabalho, que retrata os pontos anteriormente referidos. O estado de arte dá uma visão detalhada das soluções existentes, que estão divididas aos diferentes níveis de implementação e, as metodologias de avaliação apresentam indicadores que irão ser usados para medir o *overhead* específico dos mecanismos implementados e o impacto global de execução das aplicações.

1 Introdução

As máquinas virtuais de nível sistema actuais (e.g. VMWare, Xen, entre outros) fornecem mecanismos de *Checkpoint/Restore* & Migração, ou C/R&M.

Estes mecanismos permitem que essas máquinas virtuais possam guardar o seu estado de execução de forma persistente, para ser recuperado mais tarde ou para ser transportado entre sistemas distintos. Contudo, nas máquinas virtuais para linguagens de programação orientadas a objectos (*Virtual Machines Object Oriented* ou VM OO), tal mecanismo não é oferecido ao nível de uma aplicação. Uma vez que muitas aplicações utilizadas hoje em dia são executadas sobre estas máquinas virtuais, esta seria uma extensão útil.

Para muitas soluções, estes mecanismos C/R&M são vistos como um requisito e vêm obter:

- Melhor desempenho, pois possibilita migrar a execução para máquinas com menor carga;
- Disponibilidade, porque permite diminuir os tempos de interrupção de um sistema/aplicação;
- Tolerância a falhas, isto é, permite que o estado de execução seja guardado ao longo do tempo para posterior recuperação no caso de uma falha.

As VM OO têm alguns mecanismos que suportam persistência e migração de objectos. No entanto, não existem mecanismos para extrair o estado de execução de um processo, que neste contexto, corresponde às unidades de execução denominadas por *threads*. As *threads* dependem do seu ambiente de execução e não podem ser guardadas de forma persistente como qualquer outro objecto (o efeito produzido não é o desejado: a mera salvaguarda do conteúdo do objecto *thread* da máquina virtual não salvaguarda o seu contexto de execução real que é suportado por *threads* do sistema operativo subjacente).

Existe um conjunto de investigações e projectos relacionados com persistência e migração de *threads* ao nível das VM OO. A grande maioria destas soluções não são completas, possuem vários problemas, e passam algumas das responsabilidades para o programador de forma a tomar decisões, o que limita a transparência para o programador e prejudica a portabilidade das

aplicações (propriedade dada como certa no âmbito das VM OO). No entanto este problema encontra-se ultrapassado num universo bastante restrito de soluções [8,12].

Embora exista muita investigação relacionada com persistência e migração ao nível de uma *thread*, existe também um conjunto de sistemas que têm como objectivo atingir C/R&M ao nível de um processo do sistema operativo. Esses sistemas têm como limitação principal uma *thread* por processo, ou seja, os mecanismos só podem ser aplicados a processos que tenham apenas uma *thread* a correr. Muitos deles impõem um custo significativo de performance durante a execução das aplicações. Existem outros que nem sequer se preocupam com ligações externas existentes ao ambiente de execução (e.g. sockets, ficheiros, entre outros), além disso, não oferecem desempenho ideal no âmbito das VM OO porque implicam salvaguardar todo o código em memória da própria máquina virtual e não apenas da aplicação que se executa sobre ela.

Este documento está organizado da seguinte forma: na Secção 2 apresentaremos os objectivos deste trabalho através da descrição de um conjunto de propriedades a cumprir; na Secção 3 retrataremos o trabalho relacionado; na Secção 4 abordaremos a arquitectura como primeira visão da solução para este trabalho; na Secção 5 mostraremos as metodologias de avaliação de forma a obter indicadores que possam analisar a adequação e eficiência da solução final; por fim na Secção 6 apresentaremos a conclusão.

2 Objectivos

Pretende-se criar um sistema de C/R&M ao nível de VM OO, que ofereça as seguintes propriedades:

- **Transparência**: Os mecanismos de C/R&M não devem ser construídos de forma a atribuir responsabilidades ao programador. É desejável que exista um método de controlo externo realizado sobre as aplicações, para usufruir destes mecanismos (que pode ser usado por outros utilizadores que não o próprio programador). As aplicações não devem aperceber-se da mudança de ambiente, nem que foram recuperadas usando *checkpoint* ou transportadas para outro ambiente utilizando migração.
- **Flexibilidade**: Apesar de não haver imposição de atribuição de responsabilidades ao programador, propõe-se uma API que permita ao mesmo controlar esses mecanismos a partir da sua aplicação.
- **Consistência**: O estado de uma aplicação mantém-se inalterável, livre de inconsistências, mesmo após uma operação de recuperação por *checkpoint* ou migração. Em termos funcionais a aplicação prossegue a sua execução como se o *checkpoint* ou migração nunca tivesse ocorrido (não incluí questões temporais).
- **Completude**: Os mecanismos de *checkpoint* e migração têm de persistir todo o estado de uma aplicação: código, dados, estado de execução e ligações externas (ficheiros e sockets).
- **Portabilidade**: Em parte assegurada por abordagem baseada em VM OO, mas é necessário ter disponível a VM modificada/estendida em todos os máquinas. É também desejável que a mesma *source* da VM compilada num dado sistema operativo e numa dada arquitectura seja capaz de utilizar *checkpoints* criados pela mesma VM compilada noutros sistemas.
- **Eficiência**: O custo adicional de performance durante a execução da aplicação deve ser mínimo ou nenhum. O custo de performance durante a execução dos mecanismos deve ser proporcional às aplicações que as vão usar.
- **Robustez**: Os mecanismos de C/R&M no mínimo, não devem prejudicar a aplicação, nem ser fonte de novas excepções que não foram antevistas pelo programador (e.g. quando não é possível fazer *checkpoint* ou migração, a aplicação deve continuar normalmente).

3 Trabalho relacionado

Este tema está subdividido em três grandes áreas que estão directamente relacionadas com o seu título: *Checkpoint* e *Restore*; Migração; *Log e Replay*.

Checkpoint e Restore: Checkpoint é o mecanismo que permite persistir um subconjunto da informação pertencente a um processo ou a uma *thread*. *Restore* é o mecanismo que permite efectuar a sua recuperação e recomeçar a execução a partir do ponto em que foi realizado o *checkpoint*.

Migração: O mecanismo de migração permite transportar durante a execução um subconjunto da informação pertencente a um processo ou *thread* entre duas máquinas (entre nós origem e destino).

Log e Replay: Logging, no contexto de *checkpoint*, é um mecanismo que permite guardar acções para serem re-executadas mais tarde. Essas acções são re-executadas de forma sequencial sobre um estado consistente do sistema (e.g. obtido num *checkpoint* anterior).

Vão ser apresentados de seguida alguns dos sistemas mais importantes, na sua secção respectiva.

3.1 Checkpoint e Restore

Em relação ao trabalho existente nesta área, temos implementações aos diferentes níveis:

- **Nível de processo** (quer desencadeado pela aplicação, com código próprio ou através de bibliotecas específicas, quer como um mecanismo oferecido pelo sistema operativo modificado ou estendido);
- **Máquina virtual de sistema** (VM sistema);
- **Máquina virtual OO** (VM OO).

Eric Roman [39], mostra o resultado da sua pesquisa sobre sistemas que suportam *checkpoint* para o sistema operativo Linux. Esse artigo apresenta implementações ao nível do sistema operativo [51,18] e, ao nível das aplicações [22,35,14]. Existem também outras soluções interessantes feitas ao nível do sistema operativo [19,42,36,49]. Essas soluções exigem homogeneidade, na qual dependem da adopção global de uma arquitectura ou sistema operativo específico. Existe algum esforço científico de transportar esse mecanismo para as VM sistema [21], pelo facto de suportarem heterogeneidade (podendo executar a mesma instância de máquina virtual sobre sistemas operativos e arquitecturas físicas diferentes). Porém, como as VM OO encontram-se num nível de abstracção mais elevado e como também conseguem oferecer essa vantagem, existe alguma investigação na persistência de *threads* (cujas soluções estão contextualizadas na categoria de migração, pois também a suportam) e processos [45,2] a esse nível.

Existem ainda sistemas mundialmente conhecidos que suportam *checkpoint*, cuja solução é fechada pois não existe nenhuma documentação no domínio público sobre como é que esses sistemas são implementados. Temos o SGI para o sistema operativo IRIX, e o VMWare para as arquitecturas Intel.

Nível de processo. A maior parte das soluções para *checkpoint* guardam o estado de execução de forma persistente em dispositivos não voláteis, como por exemplo no disco rígido. Para muitos sistemas, este tipo de soluções é um requisito porque conseguem atingir tolerância a falhas. Porém, existem sistemas que por terem outro tipo de requisitos, não precisam de manter o *checkpoint* em disco.

Flashback [42] é um exemplo de uma solução que permite fazer *checkpoint* do estado de um processo para a memória e está feita ao nível do sistema operativo Linux. Os seus autores tiveram como motivação principal o principio de que já existem soluções boas para tentar capturar a maior parte das falhas das aplicações. Contudo é impossível apanhar todas as falhas o que faz com que os programadores precisem de utilitários para poder fazer depuração às suas aplicações, como por exemplo o `gdb`. Segundo os autores tal não é suficiente, porque há falhas que só ocorrem

passado muito tempo, ou com determinadas combinações ou configurações. O que criaram foi um método que permite um processo voltar atrás, para poderem analisar o estado da aplicação num determinado ponto de execução. Basicamente são retirados vários *checkpoints* ao longo do tempo e são mantidas todas as interacções com o ambiente externo (eventos de recepção de mensagens pela rede, pedidos de *I/O*), de forma a permitir avanços mais curtos a partir de um dado *checkpoint*.

Em termos de implementação, os *checkpoints* são guardados em processos *shadow* que capturam o estado de um processo num determinado ponto de execução através da chamada de sistema `fork`, isto é, a partir do momento que um processo *shadow* é criado, fica automaticamente suspenso. Para fazer *restore*, a execução do processo principal é interrompida e é criado um novo processo activo através de um processo *shadow*.

Esta solução permite ao programador ter controlo sobre os mecanismos de *checkpoint* e *restore*. É disponibilizada uma interface que permite ao programador definir no seu programa quando quer fazer *checkpoint* ou fazer *restore* do mesmo.

Já aqui, nas implementações ao nível do sistema operativo, é usada uma técnica conhecida com o objectivo de melhorar a performance do mecanismo de *checkpoint* designada por *copy-on-write*, também conhecida como *checkpoint* incremental. Esta técnica faz com que apenas seja copiada a informação que foi modificada entre *checkpoints*. Com isto os *checkpoints* ocupam menos espaço e o tempo para criar um *checkpoint* fica mais reduzido, porque se copia menos informação.

Existem duas soluções conhecidas, Rx [36] e Triage [49], que estendem o trabalho realizado em Flashback. Rx é um sistema que consegue recuperar automaticamente falhas duma aplicação em produção com eventos determinísticos e não determinísticos. Segundo os autores, existe uma quantidade enorme de aplicações que falham devido ao seu ambiente de execução. A solução passa por múltiplas tentativas de re-execução sobre os *checkpoints* retirados anteriormente, pela ordem inversa pelos quais foram retirados, e efectuando alterações no ambiente de execução até conseguir resolver o problema. Este sistema consegue aprender e, no pior caso, a recuperação de uma aplicação pode não ter sucesso. Um detalhe importante é que consegue guardar estado externo à aplicação tal como os ficheiros. É guardado o conteúdo dos ficheiros e seus descritores, dos ficheiros que foram acedidos ao longo do tempo.

Enquanto que Rx tem como objectivo tentar recuperar as aplicações, Triage tem como objectivo apenas diagnosticar automaticamente o problema da falha. Comparativamente ao mecanismo de *checkpoint*, é muito semelhante, só que para afins diferentes.

libckpt [35] é uma solução de *checkpoint* ao nível da aplicação que aplica *checkpoint* para memória antes de o fazer para disco. A ideia principal está em não criar *overhead* no processo no qual se faz *checkpoint*. A razão principal pela qual muitos mecanismos de *checkpoint* sofrem um grande custo de performance, é porque ficam à espera que as escritas no disco sejam efectuadas. Portanto copia-se o estado do processo para memória, e depois o *checkpoint* para disco é efectuado sobre esse estado. Contudo, embora pareça que esta técnica não vá criar nenhum custo de performance sobre o processo principal, os resultados mostram que qualquer escrita que é feita no disco tem tendência para abrandar o processamento das aplicações. Ao longo deste documento vão ser apresentados outros sistemas que fazem uso desta técnica. Essencialmente pode-se constatar que não resolve o problema por completo, mas melhora-o significativamente.

A maior parte das soluções de *checkpoint* a estes níveis de implementação conseguem abordar bem o problema de guardar o estado interno de um processo, sendo este: *signals* pendentes, espaço de endereçamento (que contém grande parte do estado de um processo: *heap*, *stack* e qualquer região mapeada) e registos de *cpu*. Contrariamente, em relação ao estado externo (descritores de ficheiro, o conteúdo dos ficheiros e sockets) já não é muito bem suportado, com uma pequena excepção sobre os descritores de ficheiro. Respectivamente ao *checkpoint* e *restore* de sockets apenas é suportado no sistema CRAK [51]. Finalmente em relação ao conteúdo dos ficheiros, apenas as soluções Rx, Triage e libckpt é que o suportam. libckpt guarda apenas o conteúdo

dos ficheiros que estão abertos. Um detalhe importante é que o suporte a *multi-thread* só está disponível nos sistemas Flashback, Rx, Triage e Jon Howell [19].

Jon Howell [19] tentou aplicar um mecanismo de *checkpoint* existente a estes níveis de implementação (sistema operativo/aplicação) à Java VM. De realçar que na altura não havia a maioria das soluções apresentadas. Mas o importante a reter desta experiência é que não é qualquer mecanismo de *checkpoint* que consegue persistir o estado da Java VM, isto porque, há estado relacionado com *multi-threading* e ligações externas às aplicações, tais como ficheiros e sockets, que tem de ser suportado pelo mecanismo de *checkpoint*.

Na tabela 1 é apresentado um resumo das soluções a estes níveis.

Nome	**Nível implementação**	**Estado interno**, referente à aplicação	**Descritores de ficheiro**	**Conteúdo dos ficheiros**	**Sockets**	***Multi-thread***
libckp [22]	Aplicação (App)	Incompleto (*signals*)	Sim	Não	Não	Não
libckpt [35]	App	Incompleto (*signals*)	Sim	Sim	Não	Não
libtckpt [14]	App	Completo	Sim	Não	Não	Não
CRAK [51]	Sistema operativo (SO)	Completo	Sim	Não	Sim	Não
BProc [18]	SO	Completo	Não	Não	Não	Não
Flashback [42]	SO	Completo	Sim	Não	Não	Sim
Rx [36]	SO	Completo	Sim	Sim	Não	Sim
Triage [49]	SO	Completo	Sim	Sim	Não	Sim
Jon Howell [19]	SO	Completo	Sim	Não	Não	Sim

Tabela 1. Resumo das soluções de *checkpoint/restore* ao nível de processo.

Máquina virtual de sistema. Anteriormente foi vista uma solução de *checkpoint* ao nível do sistema operativo que permitia pôr uma aplicação em qualquer estado de execução, com o objectivo de fazer depuração Flashback [42]. Existe um outro sistema que tem a mesma motivação, mas em vez de fazer depuração às aplicações, pretende fazer depuração aos sistemas operativos. Samuel T. King et. al [21] criaram uma solução ao nível da VM sistema UML[1], que permite fazer esse tipo de depuração. Essa solução precisa de criar *checkpoints* sucessivos para que a reposição do estado de execução seja mais eficiente.

A partir deste artigo consegue-se perceber qual é o estado de execução que tem de ser persistido por um mecanismo de *checkpoint* a este nível, sendo este: registos de *cpu*, a memória física da máquina virtual, a imagem virtual em disco, e qualquer estado que esteja dentro da máquina virtual ou do núcleo do sistema operativo *host* respectivo que afecte a execução dessa mesma máquina virtual.

A este nível também são usadas técnicas para melhorar a performance dos mecanismos de *checkpoint* e *restore*. A maneira mais fácil de retirar um *checkpoint* é através da persistência do estado completo da máquina virtual. Isso é ineficiente. Como tal, delimitaram apenas o estado que é necessário (já foi referido), e usam técnicas de *copy-on-write* assim como um sistema de controlo de versões para reduzir o tempo e o espaço ocupado por cada *checkpoint*.

Máquina virtual OO. Ao nível das VM OO existem duas soluções de *checkpoint* de processos interessantes, que **irão ser detalhadas pois apresentam implementações ao mesmo nível da solução abrangida neste documento**:

MERPATI [45] fornece uma implementação dos mecanismos de *checkpoint* e *restore* para uma máquina virtual Java em execução. Este sistema considera persistência ao nível dos objectos e

[1] User-mode Linux: http://user-mode-linux.sourceforge.net/

das *threads*, incluindo a seu estado de execução: Java *stack*, Java *frames*, *local operands* e *operand stacks*.

O estado de execução que permite a Java VM recuperar de um *checkpoint* tem o nome de *snapshot* e contém: *heap*, todas as *threads* e o conteúdo dos ficheiros *class*. De notar que os ficheiros *class* que pertencem à Java VM não precisam de ser guardados.

Quando um mecanismo de *checkpoint* é activado, a Java VM imediatamente cria uma *thread* denominada `SnapshotGenerator`. Esta *thread* é responsável por monitorizar o número de *threads* que conseguem atingir um ponto de preempção (para serem suspensas) e escrever toda a informação importante de execução do *snapshot*. É preciso ter cuidado com algumas *threads*, visto que pode acontecer que uma determinada *thread* não consiga chegar a um ponto de preempção. Uma *thread* está num ponto de preempção se já acabou de executar a instrução Java corrente e ainda não foi aceder à seguinte. Os motivos pelos quais esta situação pode ocorrer, devem-se essencialmente ao facto da *thread* estar bloqueada, ou de estar a executar código nativo (não está a executar *bytecode* e a Java VM não tem controlo sobre essa execução), ou ainda por outras razões. Para rectificar este detalhe foi acrescentado um temporizador que limita o tempo de espera até todas as *threads* chegarem ao ponto de preempção. Se o tempo exceder o limite, o *checkpoint* é cancelado.

Um dos passos dos mecanismos de *checkpoint* é gerar informação importante de execução do *snapshot*. Para tal é preciso inferir o tipo das variáveis locais e do *operand stack*, isto porque a Java VM em *run-time* não sabe nada sobre esses tipos. O único local onde esses tipos estão especificados é no *bytecode*. Existem duas formas de inferir esses tipos: actualizando em *run-time* ou *on-demand*. Por motivos de performance, optaram pela segunda solução. Essa solução requer uma análise do fluxo da informação para inferir os tipos, que segundo o artigo é uma solução já conhecida.

Para recuperar um programa Java é preciso criar uma nova instância da Java VM e inicializa-la com o *snapshot*. É fornecido um programa para o efeito.

MERPATI é um sistema flexível ao ponto de fornecer uma API para usufruir do mecanismo de *checkpoint*, porém não é transparente: a aplicação tem de ser programada para invocar esse mecanismo. Tem como desvantagens não preocupar-se com o ambiente externo (ficheiros, sockets, entre outros) e as *threads* não podem ser suspensas quando estão a executar código nativo.

Adnan Agbaria, Roy Friedman [2] têm como motivação o seguinte: anteriormente foram vistos muitos mecanismos de *checkpoint* e *restore* que assumem a homogeneidade do sistema, ou seja, os computadores vão ter sempre a mesma arquitectura e o mesmo sistema operativo. Contudo é desejável para determinados sistemas construir mecanismos de *checkpoint* e *restore* que consigam funcionar entre diferentes plataformas e sistemas operativos. A solução proposta transpõe estes mecanismos ao nível da VM OO em vez de ser ao nível de sistema. Desta forma conseguem resolver mais facilmente o problema de transporte para diferentes plataformas e sistemas operativos. Estes autores criaram mecanismos de *checkpoint* e *restore* para a máquina virtual OCaml (OCVM).

O mecanismo de *checkpoint* corre num processo `fork` para que não haja bloqueio durante o *checkpoint*. É pretendido que um *checkpoint* guarde de forma consistente o estado da aplicação. Como tal, são definidos pontos *safe*. Um ponto *safe* pode ser encontrado entre pares de instruções (*bytecode*) consecutivas ou durante uma instrução que não muda o estado do sistema. Em relação a *multi-thread*, para garantir consistência no *checkpoint*, no pior caso é necessário parar todas as *threads*, tirar o *checkpoint*, e continuar a execução dessas mesmas *threads*.

Existe um detalhe importante que nenhuma solução tinha referenciado ainda: no *restore*, é recuperada toda a informação de um *checkpoint* e colocada novamente em memória tal e qual como no processo original. Contudo, não é possível garantir que toda a informação fique nos mesmos endereços de memória que estavam anteriormente, sendo necessário neste caso um reajustamento dos apontadores da aplicação.

As vantagens desta solução em relação às existentes são:

– Portabilidade para sistemas diferentes, não precisando de guardar informação dependente do sistema operativo, o que faz com que o *checkpoint* fique mais reduzido em tamanho.
– O modelo de programação e o processo de compilação não são alterados.

Como desvantagens, podem ser apontadas as seguintes:

– O nível de transparência é contestável: os mecanismos são flexíveis mas o mecanismo de *checkpoint* tem de ser invocado dentro do programa.
– O mesmo mecanismo não funciona quando está a ser executado código nativo. Apenas é possível fazer *checkpoint* da memória que é controlável pela máquina virtual. Embora haja esta restrição, não existe sequer um esforço de adiar o *checkpoint* para um momento controlável.
– Em relação às ligações externas não são tratados sockets, e ficheiros têm alguns problemas.

Em relação ao *overhead* criado pelos mecanismos de *checkpoint* e *restore*, foi verificado que guardar o estado em disco, mesmo que seja feito num processo `fork` do original, consome muitos recursos (*cpu* e *I/O bandwidth*) do sistema e faz com que o processo original sofra também de algum atraso.

Embora isto aconteça, o *overhead* adicional é dependente apenas do tamanho da aplicação a fazer checkpoint.

3.2 Migração

O trabalho relacionado com esta área mantém os mesmos níveis de implementação referidos em *checkpoint*. Grande parte dos sistemas que propõem migração também suportam *checkpoint*. Na verdade existe uma ligação forte entre ambas as áreas, pois para migrar uma aplicação é necessário primeiro obter um estado consistente da sua execução.

Dejan Milojicic et. al [29], apresentam um estudo sobre soluções existentes para um conjunto diverso de sistemas operativos. Esse estudo é muito vasto e apresenta implementações ao nível do sistema operativo [50,4,32,3,37,47,43,13,25,28,53,27] e das aplicações [24,52,17,6].

São retratadas ainda soluções para objectos e agentes móveis, que basicamente estão inseridas nas soluções que vão ser apresentadas sobre VM OO.

Eric Roman [39] mostra também duas soluções importantes [44,34] a estes níveis. Tal como se viu em *checkpoint*, e pelas mesmas motivações, existem soluções ao nível das VM sistema [30,10] e das VM OO tanto para *threads* [12,8,38,20,1,9,46,7,23,40,48,41] como para processos [16,26].

Nível de processo. No estudo de migração de processos efectuado em [29] são apontadas um conjunto de soluções que suportam migração de processos ou *threads*. Dois pontos importantes têm de ser referenciados:

1. São definidas estratégias que são usadas para transferir o espaço de endereçamento de um processo:
   – *eager all* (Condor [24], LSF [52] e todas as soluções ao nível da aplicação): o espaço de endereçamento é todo copiado quando é feita a migração. Não fica nenhuma dependência para trás.
   – *eager dirty* (Mosix [4], Locus [50]): apenas as páginas modificadas no espaço de endereçamento é que são copiadas.
   – *copy-on-reference* (Accent [37], Mach [28]): as páginas do espaço de endereçamento só são transferidas para o nó destino, quando referenciadas.
   – *flushing* (Sprite [32]): apenas as páginas modificadas no espaço de endereçamento são copiadas para o disco de um servidor central, onde todos conseguem aceder.
   – *precopy* (System V [47]): o processo que está a ser migrado, fica a correr no nó origem enquanto o seu espaço de endereçamento está a ser copiado. Qualquer alteração sobre o espaço de endereçamento durante este processo, tem de ser enviado numa segunda fase.

Cada estratégia está desenhada para diferentes objectivos, mas obviamente cada uma tem os seus custos. Sistemas que implementam *eager all*, eliminam dependências do nó origem, mas têm um custo de migração altíssimo. *Eager dirty* reduz esse custo, mas pode criar algum *overhead* durante a execução dos sistemas que a adoptem, isto porque, têm de controlar as páginas que foram modificadas em execução. *Copy-on-reference* é a solução que mais dependências tem com o nó origem, porém, é também a que tem menor custo de migração. *Flushing* elimina as dependências com o nó origem, mas tem sempre dependências com um servidor central. Dessa forma já se pode descartar o nó origem sem problemas. O custo de migração é muito parecido com o *eager dirty*. Finalmente, *precopy* pode ter um custo adicional de migração em relação ao *eager all* caso haja alterações ao espaço de endereçamento, mas esta solução tem a vantagem de diminuir o tempo de interrupção dos serviços que possam estar a correr.

2. A implementação do mecanismo de migração, seja esta efectuada ao nível do sistema operativo ou da aplicação, influencia os seguintes factores: portabilidade, transparência e completude (Secção 2). Uma implementação ao nível do sistema operativo pode ser transparente e completa porque tem acesso a todo o sistema, mas em termos de portabilidade está dependente que todas as alterações efectuadas estejam disponíveis em todos os nós.
   Ao nível das aplicações acontece exactamente o contrário: por um lado não consegue ser transparente nem completo, mas como são inseridas instruções nas aplicações que suportam os mecanismos, então conseguem executar-se em sistemas que não têm suporte para tais. Veremos mais tarde que o mesmo tem lugar ao nível das VM OO.

Máquina virtual de sistema. Ao nível de uma VM sistema, é introduzido o conceito de *Live Migration*: tem como objectivo permitir transportar uma máquina virtual entre nós, sem interromper os pedidos (e.g. pedidos num servidor *web* ou aplicacional) que já estão a ser processados. Uma solução que explora bem esse conceito é [10] dos autores Robert Bradford et al.:

Muitas das soluções de migração das VM sistema existentes focam-se principalmente em capturar o estado de execução da máquina virtual, mas não se preocupam com o estado do sistema de ficheiros local (denominado por estado de persistência local). Muitas vezes, essas soluções quando tratam do sistema de ficheiros, contam com sistemas de ficheiros distribuídos no âmbito de uma rede local. Quando existe uma acção de migração, no destino existe apenas um tratamento de realocação dos descritores de ficheiros. Quando se trata de sistemas em larga escala, o tempo de execução pode ser dispendioso, ou ainda incompatível para determinadas soluções utilizarem sistemas de ficheiros distribuídos. Como tal, no acto da migração é preciso transportar também o estado do sistema de ficheiros entre os nós respectivos. Os autores deste artigo propõem explorar alternativas eficientes para esse efeito. Pretendem também manter os serviços activos durante o processo de migração para minimizar o tempo de interrupção devido ao acto de migração.

A solução desenhada baseou-se na máquina virtual Xen [5]. Aproveitaram o mecanismo de migração existente, que permite transportar o estado de execução de uma VM sistema entre nós diferentes.

A migração do sistema de ficheiros pode ser feita de duas maneiras: *on-demand* ou usando *pre-copying*. Na primeira os blocos são transmitidos apenas quando são precisos, o que faz com que haja alguma degradação da performance das aplicações ao longo do tempo. No segundo todo o sistema de ficheiros é transmitido antes da migração terminar, demorando mais tempo a finalizar, mas para este sistema o mais importante é reduzir o *overhead* criado pelo mecanismo de migração a longo prazo. Portanto decidiram usar a segunda solução.

Durante o processo de migração, a VM sistema no nó origem é mantida em execução. Desta forma os serviços desse nó ainda conseguem responder a pedidos. Há que ter em atenção que durante esse processo a imagem do disco da máquina virtual (referenciado como *bulk*) está a

ser enviada para o nó destino, mas no entanto, podem ser feitas alterações aos ficheiros no nó origem.

Para manter a imagem do disco actualizada no destino, essas alterações são interceptadas no nó origem (referenciadas com *deltas*), e enviadas num canal à parte para serem aplicadas no destino. Portanto, durante o processo de migração são enviados em paralelo o *bulk*, os *deltas* e ainda o estado de execução da máquina virtual que já é suportado de base no Xen.

A partir do momento que a cópia do *bulk* e do estado de execução é dada por terminada, são aplicados os *deltas* por ordem de chegada no nó destino. Quando essa operação terminar, o mecanismo de migração arranca a máquina virtual nesse nó. Seguidamente ajusta o endereço IP no serviço de DNS, para o endereço IP da máquina destino, de forma a que novos pedidos sejam encaminhados para essa máquina. Quando no nó origem não existirem mais pedidos para processar, a execução da máquina virtual nesse nó é terminada. Não esquecer que a aplicação dos 'deltas' só termina quando o nó origem terminar a execução do último pedido recebido por si. Desta forma mantém-se consistência.

Máquina virtual OO. O mecanismo de migração ao nível da VM OO está maioritariamente focado na migração de *threads*. Primeiro de tudo é preciso perceber os conceitos de *weak/strong mobility*. *Weak mobility* apenas transfere código e alguma informação (usando mecanismos de serialização de objectos, *dynamic class loading*, entre outros). *Strong mobility* transfere também o estado de execução de uma *thread*. Muitos sistemas não usam *strong mobility* por razões de complexidade de implementação própria, devido à falta de suporte em guardar o estado de execução nas VM OO existentes. Um exemplo conhecido de *weak mobility* são os *mobile agents*: o código depois de ser transportado para outra máquina, arranca sempre do início ou num método específico designado antes da migração.

As implementações feitas ao nível da VM OO podem estar ainda subdivididas em mais dois níveis:

- **Ao nível da própria VM OO**: verificam o requisito de completude, isto é, ter acesso a todo o estado de execução das *threads*. No entanto têm problemas de eficiência e portabilidade (as outras máquinas virtuais também têm de ter as alterações de código para o sistema de migração fazer efeito). Normalmente modificam ou estendem o código da VM OO, introduzindo API's para permitir a migração. Temos como exemplos: CIA [20], Sumatra [1], JavaThread [9], Nomads [46], ITS [7], Jessica2 [23].
- **Ao nível das aplicações que correm na VM OO**: são portáveis, mas têm problemas de eficiência e não cumprem o requisito de completude. Neste nível, o código (*source code* ou *bytecode*) é transformado por um pré-processador que adiciona novas instruções ao código da aplicação (instruções estas que servem para capturar ou restaurar o estado). Temos como exemplos: WASP [16], JavaGoX [40], Brakes [48], JavaGo [41].

Até à data do projecto CIA [20], existiam outras implementações de *strong mobility* que modificavam o código fonte ou alteravam o *bytecode* ou ainda o interpretador. Todas introduziam complexidade adicional e são consideradas pesadas. A solução base que os autores deste artigo propõem, essencialmente explora as funcionalidades do JPDA (*Java Platform Debugger Architecture*), que faz parte da máquina virtual Java, para retirar informação em tempo de execução de um programa para poder oferecer migração transparente (ou *strong mobility*).

Reflectindo um pouco como é que a migração poderia ser feita em Java, temos um conjunto de mecanismos que suportam migração de objectos, através de *dynamic class loading* e *serialization*. Contudo a migração de *program counter*, *multi-threads*, *stack*, *resources* e *user interface* não é directamente suportada pela Java VM.

Este artigo analisa bem o problema de *strong migration*, porém apenas é resolvida uma parte do problema. Os autores deste artigo, focam-se apenas nos problemas de migração do *program counter* e da *stack*, ou seja, assumem que o programa é *single-threaded*, e não está ligado a nenhum sistema aberto de *resources* (ou seja um agente de facto e não uma aplicação).

Para capturar o estado de uma Java *thread* são usadas funções do JPDA, para percorrer a *stack* da *thread* e guardar cada *frame* da *stack* (incluindo os valores das variáveis locais e o *program counter*). No *restore*, é usado um mecanismo de *callback* fornecido pelo JPDA, para apanhar eventos quando se entra nos métodos. A aplicação é reiniciada, e para cada método que é executado, o *callback* reconstrói as variáveis locais e o *program counter* respectivos.

Em relação à implementação, sobressaem-se dois problemas: não se consegue aceder ao *operand stack* através do JPDA; não existe nenhuma funcionalidade para definir o *program counter* (conseguem ler, mas não conseguem escrever). Para resolver o primeiro problema, não é permitido que o programador use *nested calls* nos métodos que podem invocar o mecanismo de migração, obrigando este a guardar os resultados intermédios de invocação de métodos explicitamente em variáveis temporárias locais. O segundo problema pode ser resolvido de duas maneiras: ou modificando directamente a Java VM (criaram um método `SetFrameLocation`, que pode ser acedido pelo programador), ou então fazer alguma instrumentação do *bytecode*.

A desvantagem mais clara desta solução é que tem de correr em modo de debug, o que impõe algum *overhead* significativo.

Giacomo Cabri et al. [12] focam-se em criar uma solução para o problema da mobilidade de computação em Java, usando a máquina virtual Jikes RVM. No Java é possível persistir um objecto e transferi-lo para outras máquinas através de *serialization*. Contudo, esta funcionalidade não tem o comportamento desejável quando se trata de *threads*. Serializar o objecto `java.lang.Thread` não guarda o estado de execução de uma *thread*. O problema é que as *threads* dependem do seu ambiente de execução e a única informação que é serializada são os atributos da *thread*. A informação respectiva à *call stack*, entre outros, que correspondem à execução em cada ambiente, não é mantida. Existem sistemas que suportam migração de threads ao contrário da linguagem Java, como por exemplo o Mosix [4], já referenciado anteriormente nas soluções de migração ao nível do sistema operativo.

Esta solução para garantir *strong mobility* utiliza algumas técnicas bem definidas: *On-Stack Replacement* (OSR), *type-accurate garbage collectors*, *quasi-preemptive Java thread scheduler*, entre outras.

A implementação pode ser classificada como uma solução intermédia entre a Java VM e as aplicações que correm na Java VM. O mecanismo de migração de uma *thread* executa as seguintes acções: põe a *thread* numa lista de espera, depois quando poder ser atendida, retira-se da lista, faz-se a ligação ao destino, extrai-se o estado da *thread*, e serializa-se para enviar. No destino o processo é o inverso.

Esta solução suporta dois métodos de migração:

- Migração Reactiva: uma *thread* pode ser interrompida para migração sem qualquer previsão (a *thread* não invoca nada, é-lhe pedido, e ela tem de tratar isso a qualquer altura). Obviamente vai ter de ser tomada uma acção de migração apenas quando for possível, isto é, não é imediata porque senão podia ser criado um estado inconsistente do sistema.
- Migração Proactiva: é sincronizado por si mesmo (a *thread* invoca o método de migração quando pretende migrar, ou seja já foi programado dessa forma).

Este artigo fornece muitos detalhes de implementação. De seguida vão ser retratados os mais importantes:

- *Yield points*: São pontos no código em que as *threads* verificam se podem ficar em execução. São usados para parar uma *thread*. Os *yield points* são inseridos automaticamente pelo compilador JIT[2], quando compila os métodos pela primeira vez. Estes *yield points* são postos no início e no fim de cada método e também no início dos ciclos.
- Pontos de migração: subconjunto dos *yield points*. São pontos num programa que permitem fazer migração sem problemas de inconsistência. Se uma thread recebe um pedido de migração, ela vai aguardar até chegar a um ponto de migração para poder migrar.

[2] *Just-In-Time Compiler* que transforma *bytecodes* em código executável nativo

- Para capturar o estado de uma *thread*, apenas os *frames* que foram carregados pelo código da aplicação (*user-pushed frames*) é que precisam de ser capturados. Todos os outros *frames* estão disponíveis ou podem ser substituídos por informação no destino.
- A classe `FrameExtractor` captura todos os *user-pushed frames* de uma *thread*. O método `extractSingleFrame` captura o estado de um único *frame*, através de uma versão modificada do OSR *extractor*. O OSR *extractor* original serve para atribuir um novo *stack frame* e iniciá-lo num *program counter* desejado. Um `MobileFrame` tem toda a informação capturada de um *frame*: `methodName`, `methodDescritor`, `methodClass`, `bytecode index`, `locals` e `operands`. Um `MobileThread` é composto por vários `MobileFrame`'s.
- Para recomeçar uma *thread* após ter sido migrada, efectuam-se os seguintes passos: cria-se uma nova *thread*, e suspende-se; transporta-se o estado da `MobileThread` migrada para uma nova *stack*; troca-se a *stack* da *thread* suspensa com a anterior; a execução da *thread* pode então ser retomada. Desta forma consegue-se criar uma nova *thread* com o estado que está guardado na `MobileThread`.
- Em relação às referências para objectos, são retratadas soluções para os seguintes problemas:
  - identificar referências dos objectos na *call stack* da *thread*. O importante é efectuar este passo sem criar *overhead* na execução da aplicação. Em vez de inferirem as referências em *run-time*, usam *type-accurate garbage collectors* que constroem em tempo de compilação uma tabela com as posições da stack e atributos de objectos que constituem referências;
  - como é que se consegue identificar, marcar para migração e reciclar alguns desses objectos;
  - realocação de objectos: objectos como o `File` podem ser serializados, mas quando transportados para outro nó, perdem a ligação com o ficheiro. Para resolver este problema, pode ser usado um mecanismo de serialização adicional fornecido de base do Java, conhecido como *externalizable*, que permite definir tratamento específico para quando um objecto é persistido e recuperado.
- Suporte para a migração de grupos de *threads*: (classe `ThreadGroup`). O processo de migração só é iniciado quando todas as *threads* do grupo estiverem preparadas.

Segundo Sara Bouchenak et al. [8], Java é uma boa linguagem exemplo para explorar migração de processos. É portável e já tem algumas funções que suportam transporte de computação, tais como Java *serialization*: permite capturar e restaurar o estado de um objecto, possibilitando a migração desse mesmo objecto entre máquinas diferentes. Contudo, Java não permite migração de *threads*. Este artigo propõe uma solução para migração de *threads* Java, que não impõe nenhum *overhead* à aplicação durante a sua execução. O custo de execução é transferido para o ponto em que a migração é efectuada. Portanto, esta solução é boa para sistemas que raramente fazem migração.

Os requisitos de eficiência e completude (Secção 2) fizeram com que fosse criada uma solução ao nível da Java VM. Têm acesso a todo o estado de execução das *threads* Java (requisito completude), e conseguiram criar uma solução, que como foi referido, não impõe nenhum *overhead* durante a execução da aplicação (requisito eficiência).

Segundo este artigo, o estado de execução de uma *thread* Java é composto por três estruturas de dados:

- Java *stack* associada com a *thread*;
- Um subconjunto da *heap* de objectos (inclui todos os objectos usados pela thread);
- Um subconjunto da *method area* (inclui também as classes usadas pela *thread*).

O primeiro problema na migração de *threads*, está na estrutura de dados que não pode ser transferida para outra Java VM, porque não gere qualquer tipo de informação que está guardada na *stack* Java (uma *thread* é considerada um objecto nativo, que não é suposto ser transportado entre máquinas).

Existe dificuldade em saber qual é o tipo de dados que estão guardados na *stack* Java. O único sítio onde estes tipos são conhecidos é no *bytecode* de cada método, que coloca dados na *stack*.

Portanto, esta solução baseia-se numa abordagem de *type inference*. Para evitar qualquer *overhead* durante execução, este *type inference* apenas é efectuado no ponto de migração.

Type inference: Para cada *frame* na *stack* de uma *thread*, o *bytecode* do método associado é analisado do início até ao ponto de saída do método. O objectivo pretendido é associar o tipo de cada valor (variável local ou resultado parcial) de cada *frame* e guardá-lo no respectivo *type frame* (análogo ao *operand stack tracking*, na verificação de código feita pela Java VM). Adicionalmente existe uma questão que tiveram de contornar, e que está relacionada com o caminho que deve ser seguido quando existem vários caminhos entre o ponto de entrada e saída de um método.

O segundo problema está relacionado com a performence da Java VM. A execução dos métodos normalmente está associada à *stack* da *thread*, quando produzidos pelo interpretador Java, mas deixa de ser verdade quando um método é dinamicamente compilado pelo JIT *compiler*. Este método passa a partir desse momento a ser gerido na *native stack* associada com a *thread*. De forma a usar *type inference* nos métodos compilados dinamicamente pelo JIT *compiler*, é invocado um método de *dynamic deoptimization* para recriar os *frames* Java tal e qual como se fossem produzidos pelo interpretador Java. Assim já é possível aplicar o *type inference* a estes *frames*. No fim, é possível recompilar de novo dinamicamente os métodos com o JIT *compiler* para ficarem como estavam. Esta implementação consegue fazer *checkpoint*, *restore* e migração ao nível de uma *thread*, através da invocação de métodos sobre cada *thread*.

Esta solução consegue mover todos os custos de performance para o mecanismo de migração, seja o tempo de execução assim como o tempo de migração, ou seja o *overhead* na execução do código e a latência da migração, respectivamente.

Enquanto projectos anteriores distribuiam relativamente bem estes dois *overheads* (execução e migração), esta solução elimina o *overhead* de execução mas tem como consequência um *overhead* adicional na migração. Esta característica pode ser vantajosa em sistemas que raramente necessitam de migrar *threads*.

Johnston Stewart et al. [38] propõem estender o modelo de *threads* do SSCLI (*Shared Source Common Language Infrastructure*) para permitir capturar e guardar a *stack* de uma *thread* para ser enviada e retomada noutro nó.

O SSCLI, mais conhecido como ROTOR, é uma implementação aberta e portável das ferramentas de programação e bibliotecas que estão de acordo com o *standard* ECMA-335 CLI. Este *standard* define que as aplicações escritas em linguagens de alto nível possam ser executadas em diferentes ambientes sem terem de ser re-escritas especificamente por causa de determinadas características existentes nesses ambientes.

No *Shared Source CLI* uma nova instância da classe `System.Threading.Thread` é denominada por *managed thread*. Uma *thread* que tem origem fora do SSCLI denomina-se por *unmanaged thread*.

Managed threads estão implementadas por cima das *threads* suportadas pela PAL (*Platform Adaptation Layer*). PAL faz com que SSCLI seja facilmente portável para outros sistemas operativos. *Threads* PAL têm como objectivo abstrair detalhes e semânticas das diferentes implementações que têm de ser feitas do modelo de *threading*, para os diversos sistemas operativos onde o SSCLI se pode executar.

Threads PAL têm uma relação de 1 para 1 com *threads* SO (sistema operativo). *Managed threads* estão sempre associadas a uma *thread* PAL. Contudo uma *thread* PAL nem sempre está associada a uma *managed thread*. A ideia base é que uma *thread* SO está relacionada com uma *thread* PAL, que por si pode estar ou não relacionada com uma *managed thread*.

Cada aplicação corre dentro de um AppDomain (*Application Domain*) que isola as aplicações umas das outras. Uma *thread* SO só tem no máximo uma *managed thread* associada a si em cada AppDomain.

De realçar que uma *managed thread* pode estar, ou não, associada a uma *thread* SO. As *threads* que foram paradas, ou que foram criadas mas não iniciadas, podem não ter uma *thread* SO associada.

No SSCLI só existe uma *stack*, e todas as *threads* partilham a mesma *stack*. Cada *thread* é composta por uma lista interligada de estados, que correspondem aos métodos que cada *thread* está a executar. Estes estados denominam-se por *activation records*.

Agora que se entende como é que o modelo de *threading* está implementado no SSCLI, já é possível entender a solução proposta. Para poderem criar uma estrutura única que contenha toda a informação de uma *thread*, utilizam-se *containers* para conseguir associar os objectos (da *heap*) às *threads* respectivas. Contudo, o grande desafio está em capturar o estado de execução de uma *thread*. Como as *threads* do SSCLI estão relacionadas com as *threads* PAL, e consequentemente com as *threads* SO, então não são facilmente serializáveis. A solução passa por descobrir através da *thread* do SSCLI qual é a *thread* SO correspondente, e com isto guardar o estado dessa *thread* SO.

Este artigo apresenta algumas teorias interessantes para implementar *strong mobility* no SSCLI. Contudo, não são apresentados nenhuns resultados, nem explicam como é que se pode recuperar uma *thread* depois da sua serialização ou migração.

Embora haja muito trabalho científico sobre migração de *threads*, **existe algum que se foca também em migração de processos**:

WASP [16] é uma solução que implementa *strong migration* para a máquina virtual Java, instrumentando o código fonte através de um pré-compilador que insere código para guardar e restaurar o estado de um programa.

Segundo o autor deste artigo, o estado de um programa Java consiste no seguinte:

- Código do programa
- Dados do programa (localizado nas suas variáveis)
- Informação de execução (*program counter*, *call stack*, entre outros).

Um problema para capturar o estado do programa é que a informação está localizada em sítios diferentes: as variáveis do programa podem ser acedidas no próprio programa, enquanto que o restante encontra-se em níveis mais baixos. A Java VM suporta capturar o estado de todos os objectos usando *serialization*, mas não suporta capturar o *method call stack* que contem os valores das variáveis locais a cada método, nem o *program counter*.

Portanto *strong migration* ainda não é suportado pela Java VM como já foi visto noutras soluções, e os autores deste artigo propõem capturar o estado de um programa Java ao nível da linguagem.

Object serialization captura uma grande parte da informação de um processo, mas fica a faltar a informação que está localizada na máquina virtual (*method call stack* e *program counter*). Para guardarem esse estado, inserem código através de um pré-processador que guarda os valores locais a todos os métodos inclusive o *program counter*.

O *checkpoint* e *restore* do *program counter* e *stack* é feito da seguinte forma:

- Para capturar o estado da *stack*, usam um tipo de *method call stack serialization*: usam o mecanismo de erros do Java, semelhante a *exceptions*, com a diferença de que os erros do Java não têm de estar declarados na assinatura de um método. Basicamente quando existe um pedido de migração é lançada uma excepção, e em cada método que esteja invocado, existe um *exception handler* instrumentado que trata a excepção e guarda o seu estado. Esta instrumentação foi realizada pelo pré-compilador, e insere o código *try-catch* do *exception handler* em todos os métodos que eventualmente possam iniciar a captura do estado. Como este mecanismo é baseado em *exceptions*, o estado de todos os métodos é capturado, isto porque as *exceptions* são sempre relançadas para cima até todos os métodos activos serem percorridos e o estado ser capturado na totalidade.
- Para o *program counter*, foi necessário garantir a não re-execução de código que já tivesse sido executado, até à captura do estado. Para tal, definiram formas de permitir saltar partes de código em cada método: *artificial program counter* e *code regions*. Basicamente, são definidas regiões de código, que estão protegidas com instruções *if*, que específica se uma parte do código deve ou não ser pulada na re-execução.

– No *restore* executam os métodos pela ordem inversa pela qual foram guardados (*stack*), de forma a garantir que as variáveis locais a cada método sejam restauradas com os valores correctos. A reconstrução precisa de alguma instrumentação porque as variáveis locais podem ter dois valores: valor de restauro ou o valor normal do programa.

Esta solução também suporta *multi-thread*:

– Capturam o estado do *method call information* de cada *thread*.
– Uma *thread* que inicie o processo de guardar o estado, tem de esperar que as outras *threads* estejam também prontas para guardar o seu estado, e ainda, que todas as *threads* se sincronizem nesse processo. Também serve para garantir que o sistema é migrado apenas quando tudo está parado. É uma verificação necessária. Uma questão interessante é 'quando é que as *threads* estão prontas para guardar o estado?'. A resposta foi simplificada: para minimizar o custo de pôr em cada instrução uma *thread* pronta para a migração, são dadas responsabilidades ao programador para definir intervalos no código para resolver esse problema. Este é um ponto fraco desta solução. Existe ainda outra questão que não está bem retratada: 'o que é acontece no caso de haver *threads* que já estão anteriormente bloqueadas?'. Basicamente nunca vão estar prontas para guardar o estado, e todo o processo fica bloqueado.

Relativamente ao ambiente externo, como é o caso dos ficheiros, estes estão acessíveis através de um sistema de ficheiros distribuído. Como tal, só tem de ser resolvido um problema de realocação.

Existe ainda outra pequena limitação associada a esta solução que é não conseguir instrumentar o código das bibliotecas para as quais não é fornecido o código fonte.

M-JavaMPI [26] é uma solução que corre em cima da máquina virtual Java, que suporta migração de processos e serviços de comunicação com localização transparente (através do *Message Passing Interface*, a partir de agora MPI).

Esta solução utiliza a interface de debug do Java (JVMDI), que faz parte do referido JPDA, para capturar o estado de execução de um processo. Como esta interface está disponível de base em qualquer Java VM, é potencialmente mais portável que algumas soluções existentes, e a solução é mais simples, visto que não é preciso modificar a Java VM.

A arquitectura desta solução está dividida em três camadas:

– Camada de pré-processamento, é usada para modificar o *bytecode* de uma aplicação Java, para inserir funções que permitem restaurar o Java *stack* e continuar a execução durante a migração. Estas funções, em mais detalhe, são vistas como tratamento de excepções na forma de *try-catch* e só são activadas quando uma excepção de *restoration* ocorrer.
– Camada da API Java-MPI, fornece API's aos mecanismos para ter acesso aos serviços de comunicação com localização transparente. Estas API's comunicam sempre com um serviço local (MPI *daemon*) a cada máquina para falar com os outros. Desta forma, após migração, a comunicação entre nós diferentes pode efectuar-se como se a migração não tivesse ocorrido. Não existe o problema de recolocação.
– Camada de migração, é responsável por capturar o estado de execução de um processo, e restaurar esse estado no nó destino, incluindo os canais de comunicação.

Como já foi referido, JVMDI, é usado para capturar o estado de um processo. Contudo, existem algumas limitações para que o JVMDI consiga restaurar esse estado. Como tal, os autores tiveram de usar técnicas de pré-processamento para resolver o problema.

Um dos obstáculos que teve de ser ultrapassado foi o seguinte: com JVMDI é difícil extrair e reconstruir os *operand stacks* de um *frame*. Para o resolver, produzem os resultados intermédios (em *bytecode*) para variáveis temporárias auxiliares.

No *restore*, existem dois detalhes importantes: o tratamento de excepções captura a já referida excepção *restoration*, e para cada método que intercepte essa excepção, as variáveis locais são

repostas e é executado um comando que produz um salto para a posição de execução que foi guardada quando o processo foi capturado.

Resumindo apenas o fluxo do mecanismo de migração: quando existe um pedido de migração, o processo a migrar é suspenso. De seguida é enviado um pedido de migração ao MPI *daemon* que está a correr localmente. Este trata de capturar o estado de execução do processo e envia-lo. Todas as mensagens que sejam enviadas durante o processo de migração, são mantidas no nó origem, e enviadas no fim da migração para o nó destino.

Existem três desvantagens conhecidas desta solução: o mecanismo de migração impõe que as aplicações corram em modo de debug, por causa do JVMDI; apenas consegue trabalhar com processos que tenham apenas uma *thread*; e que tenham sido desenvolvidas aplicações sobre MPI, o que não é de forma alguma geral.

Das soluções apresentadas e detalhadas ao nível da VM OO, quer para os mecanismos de migração, quer para os mecanismos de *checkpoint/restore*, pode-se constatar que existem soluções semelhantes para cada tipo de problema:

- Guardar e transportar objectos (análogo a quase todas as soluções): são usados mecanismos de serialização e carregamento dinâmico de classes, já suportados de base nas VM OO correntes.
- JPDA *operand stack* (CIA [20], M-JavaMPI [26]): não é possível reconstruir o *operand stack* associado a cada método. A solução passa por obrigar o programador a guardar os resultados intermédios de invocação de métodos em variáveis temporárias auxiliares, ou através de instrumentação automática, que gere automaticamente esses resultados intermédios para as tais variáveis auxiliares.
- *Type inference* (MERPATI [45], Sara Bouchenak et al. [8]): a VM OO, não tem conhecimento sobre os tipos das variáveis em execução. A solução é análoga ao *operand stack tracking*, na verificação de código feita pela VM OO, com ligeiras modificações.
- *Exception handling* no *restore* (WASP [16], M-JavaMPI [26]): é usado na recuperação de uma aplicação um mecanismo de tratamento de excepções que percorre todos os métodos e, repõe toda a informação local a um método.

Na tabela 2 é apresentado um resumo das soluções anteriormente detalhadas dos mecanismos de C/R&M ao nível das VM OO.

3.3 Log e Replay

Esta área está relacionada com o tema deste trabalho porque existe um conjunto de sistemas [11,31] que conseguem atingir *checkpoint* através de *log e replay*. Essencialmente, abordam o problema de uma forma diferente.

No Hyperviser [11], replica-se o estado de execução de um sistema, entre várias máquinas. Basicamente os sistemas evoluem globalmente, passando instruções entre si, para ficarem todos no mesmo estado. Estas instruções são interceptadas no sistema principal e entregues aos sistemas secundários. Se o sistema principal falhar, é eleito um secundário para o seu lugar. Desta forma existe sempre disponibilidade. Neste artigo são descritos um conjunto de protocolos para manter o sistema global consistente.

Esta solução atinge *checkpoint*, porque através das técnicas *log e replay*, o estado de execução de um sistema é replicado, e essas réplicas representam o *checkpoint* do sistema principal.

Os autores desta solução tiveram de resolver alguns detalhes interessantes, estando estes relacionados com as interacções com o ambiente:

- *Output* para o ambiente: Só o sistema primário é que passa a informação para o ambiente.
- *Input* do ambiente: O sistema primário é que faz todas as escolhas e decisões não determinísticas. Quando um primário falha, o secundário eleito tem de efectuar as mesmas escolhas. Não é desejável que um secundário dê a hora do dia mais atrasada do que o antigo primário, isto é, o relógio tem de se manter contínuo. Este problema foi resolvido recorrendo a

Nome	**Nível implementação**	**Contexto**	**Estado interno**, referente à aplicação	**Descritores de ficheiro**	**Conteúdo dos ficheiros**	**Sockets**	***Multi-thread***	**Transparência**	**Portabilidade**	**Eficiência**
MERPATI [45]	VM OO (modificações internas)	*Checkpoint* Processo	Completo	Não	Não	Não	Sim	Semi-transparente (invocação directa)	Semi-portável	Semi-eficiente
Adnan Agbaria, Roy Friedman [2] (OCVM)	VM OO (modificações internas)	*Checkpoint* Processo	Completo	Sim	Não	Não	Sim	Semi-transparente (invocação directa)	Semi-portável	Semi-eficiente
CIA [20]	VM OO (JPDA, modificações internas)	Migração *Thread*	Incompleto (objectos da *heap*)	Não	Não	Não	Não	Semi-transparente (invocação directa)	Semi-portável	Ineficiente (JPDA)
Giacomo Cabri et al. [12] (Jikes RVM)	VM OO (extensão)	Migração *Thread*	Completo	Sim	Parcial (Sistema de ficheiros distribuído)	Não	Parcial (ThreadGroup)	Semi-transparente (adoptar classes estendidas e invocação directa)	Semi-portável	Eficiente
Sara Bouchenak et al. [8] (*Zero overhead*)	VM OO (modificações internas)	Migração *Thread*	Completo	Não	Não	Não	Não	Semi-transparente (invocação directa)	Semi-portável	Durante execução é bom, durante migração é pesado
Johnston Stewart et al. [38] (SSCLI)	VM OO	Migração *Thread*	Completo (apenas em teoria)	Não	Não	Não	Sim	Desconhecido	Semi-portável	Sem resultados
WASP [16]	VM OO App (*pre-compiler*)	Migração Processo	Completo	Sim	Parcial (Sistema de ficheiros distribuído)	Não	Sim	Não transparente (responsabilidades sobre o programador)	Semi-portável	Semi-eficiente
M-JavaMPI [26]	VM OO App (JPDA, instrumentação *bytecode*)	Migração Processo	Completo	Não	Não	Sim	Não	Semi-transparente (pedidos efectuados por MPI)	Portável	Ineficiente (JPDA)

Tabela 2. Resumo das soluções de C/R&M detalhadas neste documento ao nível da VM OO.

um mecanismo de sincronização de relógios, usando *piggyback* nas mensagens trocadas entre sistemas.

Jeff Napper et al. [31] transpõem a mesma ideia, mas agora ao nível da máquina virtual Java. Tal como no Hypervisor, construiu-se um sistema de replicação com uma arquitectura de backup primário - secundários. Para além dos problemas que já tinham sido encontrados (efeitos não determinísticos, excepções assíncronas, output para o ambiente, entre outros), também teve de ser resolvido o problema do não determinismo relacionado com o *multi-threading*. Mais uma vez é uma solução baseada em máquinas de estado, cujas alterações são efectuadas à custa de comandos ou instruções, que são transportados entre sistemas.

Por outro lado, podemos verificar que é frequente existirem sistemas de *log e replay* [15,33] que usam mecanismos de *checkpoint* para melhorar a sua performance. É importante reter que se essas soluções usassem apenas *log e replay*, teriam um custo de performance elevado. A razão pela qual isto acontece é porque o *logging* guarda comandos ou instruções que provocam avanços muito pequenos na recuperação. Em vez de avançar um sistema, executando todas as instruções guardadas a partir de um ponto inicial, com *checkpoint* pode-se aproximar mais eficientemente para o ponto pretendido, e a partir deste, aplicar as instruções guardadas através de *logging*.

As soluções anteriormente apresentadas: Flashback [42], Rx [36], Triage [49], Samuel T. King et. al [21], são de facto soluções que fazem também uso de técnicas de *log e replay* em conjunto com *checkpoints*, pelas razões anteriormente apontadas. Essas soluções como tratavam de aspectos importantes respectivos a *checkpoint*, foram antecipadamente retratadas nessa categoria. Em [15,33] são apresentadas soluções mais recentes, que incidem mais apenas em questões relacionadas com *log e replay* e, que por essa razão, valem a pena serem detalhadas aqui.

SMP-ReVirt [15] foi a primeira solução a fazer *log e replay* em VM sistema com múltiplos processadores. Para detectar corridas de acesso à informação entre os vários processadores virtuais é usado o mecanismo de protecção de páginas do hardware, que está disponível em qualquer computador e processador moderno. Através disso, é possível fazer *log e replay* de toda a máquina virtual, incluindo do núcleo do sistema e de todas as aplicações, sem requerer modificações no software. Usar o mecanismo de protecção de páginas do hardware evita o *overhead* das soluções que instrumentam todas as instruções de leitura e escrita ao nível do software, mas requer um controlo adicional e com alguma cautela das zonas de informação partilhadas pelos diversos processadores.

Soyeon Park et al. [33] exploram o conceito de que a maior parte das técnicas de *log e replay* têm um problema grave, que está associado com a tentativa de reprodução de uma falha logo na primeira tentativa de *replay*. Como resultado os sistemas têm um custo de performance elevado (10-100 vezes mais lentos), isto porque, têm de guardar todos os eventos não determinísticos, enquanto as aplicações estão em execução. Estes autores propõem como solução, apenas guardar os eventos não determinísticos essenciais, que consigam reconstruir o estado de um sistema. Mesmo que o estado reconstruído não seja exactamente igual ao estado que originou a falha, isso não constitui problema, porque o interesse nem é reconstruir um estado exactamente igual, mas sim reproduzir a falha. A ideia está bem explorada, e a partir do momento em que se consegue reproduzir a falha, este consegue reproduzi-la sempre da mesma forma, em qualquer altura.

4 Arquitectura

Na Figura 1 é apresentada uma visão inicial e geral da arquitectura prevista para este trabalho. A figura descreve um conjunto de situações exemplificativas da utilização dos mecanismos que este trabalho visa oferecer. Três casos englobam as funcionalidades pretendidas:

- *Checkpoint* para o disco local e seu respectivo *restore* posterior. Na Máquina 1 (M1), o *checkpoint* da AppA (Aplicação A) é efectuado e salvaguardado para disco local e recuperado mais tarde.

- *Checkpoint* para um sistema de ficheiros distribuído, e qualquer nó que esteja ligado a esse sistema, consegue fazer *restore* do estado persistido. Pode ser considerada como uma migração indirecta e/ou replicação indirecta, pois podem ser lançadas mais instâncias com o *restore* e suas réplicas dos ficheiros. Na M2, é efectuado o *checkpoint* da AppB para o sistema de ficheiros distribuído, e mais tarde recuperado na M4, através do mesmo.
- Migração entre dois nós. A migração é feita de forma directa entre dois nós. Na M3, o *checkpoint* da AppC é directamente transportado para a M4 e aí reiniciado.

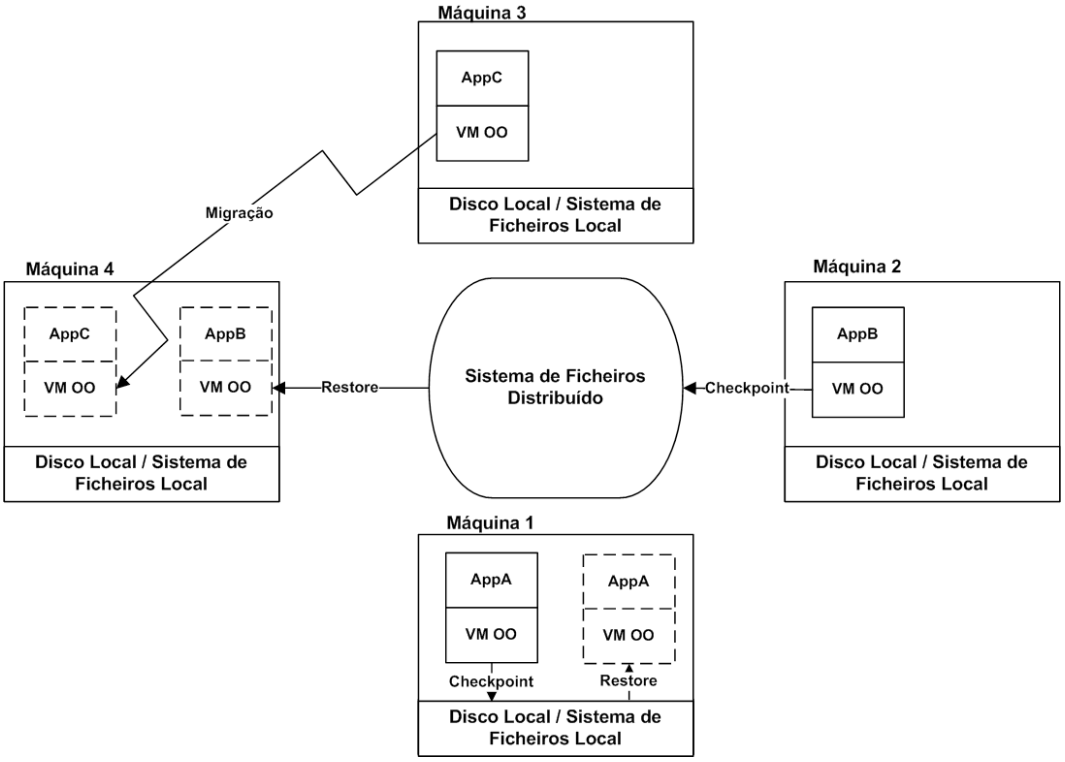

Figura 1. Visão geral dos mecanismos de C/R&M.

As aplicações a tracejado, representam novas instâncias criadas no momento em que se efectua *restore* ou migração.

A VM OO escolhida para implementar os mecanismos de C/R&M foi a Jikes RVM, pelas seguintes razões que relevam da linguagem suportada e da sua arquitectura interna:

- *Yield points*, podem ser usados para parar a execução de cada *thread* num ponto correcto (para garantir consistência no funcionamento dos mecanismos, do estado de execução salvaguardado pelos mesmos).
- *On stack replacement*, permite substituir ou adicionar *stack frames* (métodos em execução de cada *thread*) e continuar a execução através de um ponto de execução especificado.
- Suporta extrair o estado dos *stack frames* de uma *thread*. A classe que representa o estado de um *frame* é `ExecutionState`.

É importante referir também que Jikes RVM é uma VM OO que consegue executar programas Java e, ao contrário de muitas máquinas virtuais Java existentes, está implementada em Java.

Isto é visto como uma vantagem, porque Java é uma das linguagens mais utilizadas e suportada em ambientes Linux, Windows e Macintosh.

Na Figura 2 é apresentada uma visão geral da arquitectura de uma VM OO (Java). Através desta arquitectura, é possível entender qual é o estado de execução de uma aplicação a executar-se sobre essa VM OO:

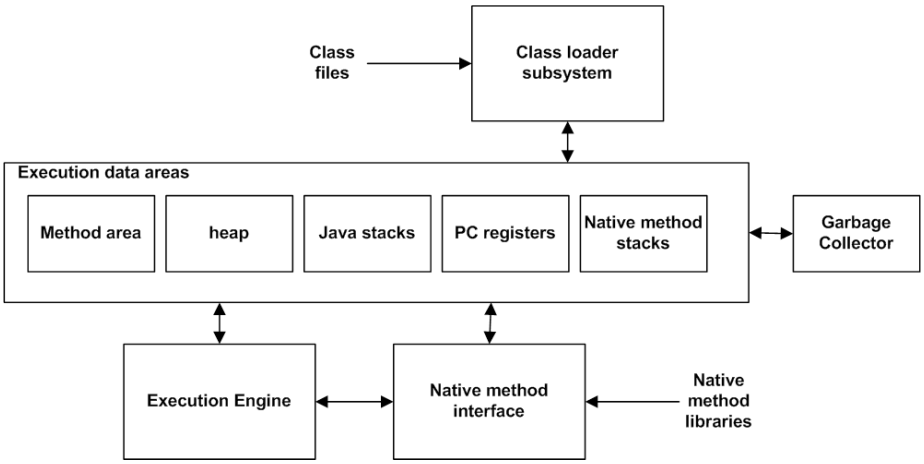

Figura 2. Visão geral da arquitectura de uma VM OO (Java).

- *Method area*: guarda o código das classes do programa.
- *Heap*: zona de memória global que guarda objectos e *arrays* que são dinamicamente alocados.
- Java *stacks*: representa o estado de todos os métodos (não nativos) por cada *thread*, que estão em execução. Guarda as variáveis locais, resultados intermédios (operandos) e argumentos a cada método.
- Registos PC (contadores de programa): cada nova *thread* que é criada tem associada automaticamente um contador de programa próprio. Se uma *thread* está a executar um método não nativo, então esse contador de programa aponta para a próxima instrução (*bytecode*) que tem de ser executada.
- *Native method stacks*: o estado de execução de métodos nativos é guardado numa forma dependente da implementação nesta zona.

Todo o estado externo à aplicação, sendo este conteúdo dos ficheiros (incluindo os descritores de ficheiro) e sockets, também tem de ser devidamente tratado. O conteúdo dos ficheiros pode trazer algumas complicações. Primeiro de tudo não se sabe que ficheiros é que uma aplicação tem efectivamente controlo e poderá utilizar no futuro. Apenas se pode conhecer aqueles presentemente abertos e os que já foram manipulados no passado. Para minimizar o custo de performance, é fornecida a opção ao utilizador que usufrui dos mecanismos de C/R&M, de incorporar no *checkpoint* o conteúdo actual dos ficheiros que estão dentro da pasta da aplicação ou noutra(s) a indicar. No mínimo são sempre guardados os ficheiros que estão abertos.

Em termos de implementação apresentamos já uma primeira abordagem à intervenção referente a alguns promenores/detalhes, resultante da análise da VM OO em causa:

- Embora haja mecanismos que suportem persistência de objectos (*heap*), tais como *serialization*, é preciso dar uma atenção especial aos seguintes tipos de objectos quando numa acção de *restore*:

  - Ficheiros: os ficheiros podem estar localizados numa nova localização, como tal, é preciso redefinir o caminho para onde está localizado um ficheiro. Pode ser também necessário reajustar o cursor do ficheiro. Este cursor pode estar guardado ou no objecto Java, ou implicitamente no `libc` (que já está fora da própria máquina virtual).
  - Sincronização de *threads*: tem que ser acautelada a situação das *threads* que já estavam bloqueadas no momento de *checkpoint*, devem ser recuperadas nesse estado.
  - Sockets: em cada *restore*, é preciso reabrir de novo as ligações que estavam estabelecidas anteriormente.
- Métodos nativos: A VM OO não tem controlo sobre este estado (incluí JNI[3] e JIT). Em relação ao JNI, esta solução apenas se compromete em suportar aplicações Java *standard*. Em relação ao JIT, a máquina virtual é que decide quando é que deve optimizar ou não um método. Se houver métodos optimizados, pode ser usado um mecanismo de desoptimização para guardar o estado respectivo.
- Identificação das *threads*: os mecanismos de *checkpoint* e migração vão sempre precisar de parar todas as *threads*. A identificação das *threads* é então um ponto de partida chave. No Jikes RVM uma *thread* tem internamente apontadores para todas as outras *threads*. Essa informação está representada na classe `RVMThread`.
- *Yield points*: é preciso limitar os pedidos de pausa das *threads* com um temporizador, isto porque, as *threads* podem não conseguir atingir um ponto correcto. No caso do temporizador expirar, então cancelam-se ou reagendam-se os mecanismos de *checkpoint* ou migração (com possível notificação do utilizador).
- Reajustamento dos apontadores dos objectos: o código interpretado/re-JIT vai ter de reconsultar as *object handles* (identificadores internos) e utilizar os novos endereços dos objectos recriados na recuperação.

Na Figura 3 são introduzidos dois componentes importantes desta solução, para explicar como é que vai ser fornecida transparência: Controlador; Serviço para receber pedidos de migração. O Controlador permite a um utilizador (seja este o próprio programador, ou outro), usufruir dos mecanismos de C/R&M (pode ser visto como uma interface de interacção). Cada aplicação tem uma *thread* especial, que está em escuta num determinado socket para receber acções do Controlador. O Serviço para receber pedidos de migração é auto-explicativo como se pode ver na figura. Nesta figura é ainda representado um exemplo da informação empacotada de um *checkpoint*.

Finalmente, para obter melhores resultados de performance, são propostas como possíveis optimizações as seguintes:

- Compressão em memória e do conteúdo dos ficheiros;
- *Checkpoints* diferenciais: as alterações efectuadas entre *checkpoints* têm de ser mantidas. Tem de se manter também as dependências entre *checkpoints*, e no *restore* fazer as recuperações sucessivas.
- Redução do caminho crítico numa migração: enviar e recuperar a informação o mais rápido possível, para que quando o último byte ou bloco de memória for recebido, a aplicação possa continuar de imediato a execução.

5 Metodologia de Avaliação

Micro-Benchmarks (*overhead* específico dos mecanismos implementados)

- Tempo para realizar *checkpoint* (pausa da aplicação), em função da memória ocupada e número de *threads* em execução (pode ser feito em conjunto ou em separado);
- Dimensão do *checkpoint*, também em função da memória ocupada e do número de *threads* em execução;

[3] *Java Native Interface*

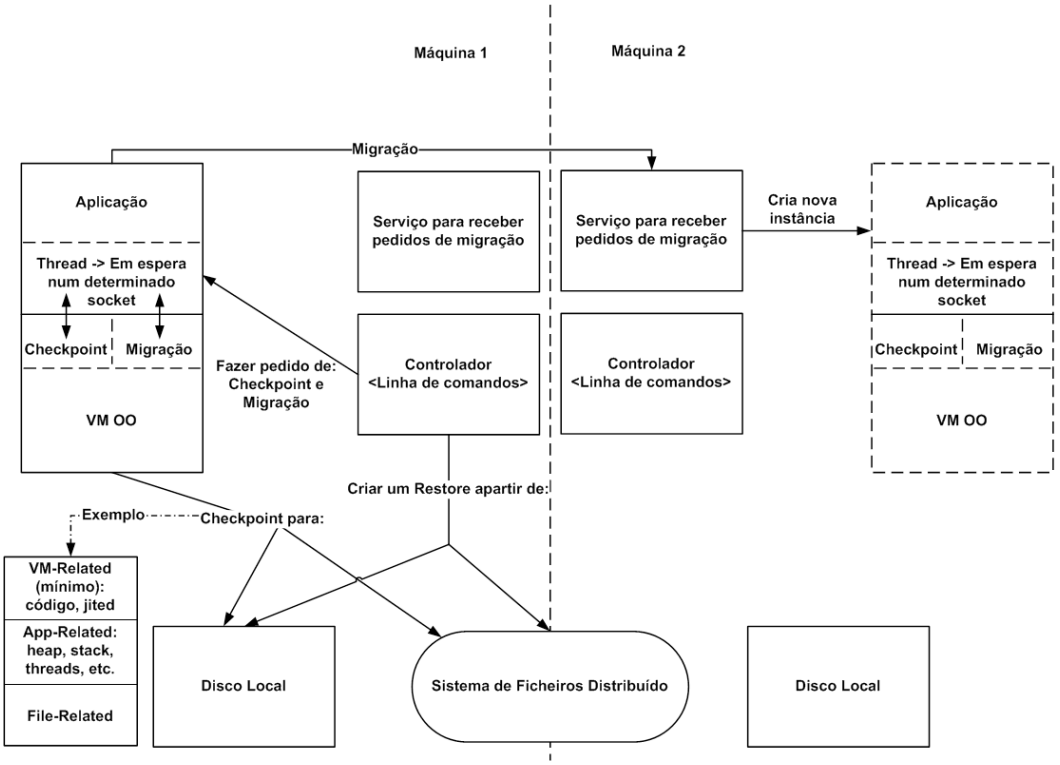

Figura 3. Visão em mais detalhe dos mecanismos de C/R&M.

- Reduções de tamanho do *checkpoint* (compressão do tamanho *VS* tempo adicional);
- Tempos de *restore* (latência prévia à re-execução), em função da memória e número de *threads*;
- Tempos de migração com e sem contribuição da transferência na rede;
- Migração em simultâneo com criação de *checkpoint* (*overlap*). Tentar esconder os tempos de pausa o mais possível, isto é, reduzir o caminho crítico.

Macro-Benchmarks (medem o impacto global de execução das aplicações)

- Tempos globais de aplicações com suporte para *checkpoint/restore*, estando este ligado ou desligado (com o objectivo de medir o *overhead* latente);
- Tempo global da aplicação com N migrações (ver se demora mais tempo);
- Simular falhas e, recolher tempos de execução com *checkpoints* (somar tempo de execução com recuperação, mais tempos de *checkpoint*) *VS* execução do início em cada falha em que apenas a última chega ao final (somar todos os tempos de execução com os reinícios).

6 Conclusões

Este documento apresenta um conjunto alargado de soluções que suportam C/R&M de *threads*, processos e até mesmo de sistemas operativos. As implementações surgem aos diferentes níveis: nível de processo, quer desencadeado pela aplicação, com código próprio ou através de bibliotecas específicas, quer como um mecanismo oferecido pelo sistema operativo modificado ou estendido; VM sistema e VM OO. Deu-se alguma importância às soluções que se encontram ao nível de uma VM OO porque a solução deste documento localiza-se nesse mesmo nível.

Esta solução define um conjunto de propriedades que, como foi reportado no trabalho relacionado, nenhum outro sistema ao nível de uma VM OO fornece conjuntamente, das quais se destacam: transparência, completude, portabilidade e eficiência.

A VM OO escolhida para implementar os mecanismos de C/R&M foi a Jikes RVM, por razões que relevam da linguagem suportada e da sua arquitectura interna. Em termos de implementação, foi já apresentada uma primeira abordagem respectiva à intervenção referente a alguns promenores/detalhes resultantes da análise da VM OO em causa. A ideia geral associada ao estado que tem de ser retratado pelos mecanismos de C/R&M é a seguinte: tentar guardar o mínimo de estado relacionado com os mecanismos internos da VM OO em si mesma (código da própria VM, código da aplicação em execução, eventualmente já JITed), guardar todo o estado relacionado com a própria aplicação, sendo este *heap*, *stacks* e *threads*, contando também com todo o estado externo à mesma, tais como, ficheiros e sockets.

Referências

1. A Acharya, M Ranganathan, and J Saltz. Sumatra: A language for resource-aware mobile programs. *Lecture Notes in Computer Science*, 1222:111–130, 1997.
2. Adnan Agbaria and Roy Friedman. Virtual-machine-based heterogeneous checkpointing. *Software: Practice and Experience*, 32(12):1175–1192, Outubro 2002.
3. Y. Artsy and R. Finkel. Designing a process migration facility: the Charlotte experience. *Computer*, 22(9):47–56, 1989.
4. A Barak and O La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4-5):361–372, Março 1998.
5. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.
6. Krishna a. Bharat and Luca Cardelli. Migratory applications. *Proceedings of the 8th annual ACM symposium on User interface and software technology - UIST '95*, pages 132–142, 1995.
7. S Bouchenak and D Hagimont. Pickling threads state in the Java system. In *Third European Research Seminar on Advances in Distributed Systems*, 1999.
8. S. Bouchenak and D. Hagimont. Zero overhead Java thread migration. *Rapport technique de l'INRIA-Rhone-Alpes*, page 33, 2002.
9. S. Bouchenak, D. Hagimont, S. Krakowiak, N. De Palma, and F. Boyer. Experiences implementing efficient Java thread serialization, mobility and persistence. *Software: Practice and Experience*, 34(4):355–393, 2004.
10. Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. *Proceedings of the 3rd international conference on Virtual execution environments - VEE '07*, pages 169–179, 2007.
11. T.C. Bressoud and F.B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.
12. G. Cabri, L. Leonardi, and R. Quitadamo. Enabling Java mobile computing on the IBM Jikes research virtual machine. *Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 62–71, 2006.
13. David L Cohn, William P Delaney, and Karen M Tracey. ARCADE: A Platform for Heterogeneous Distributed Operating Systems. In *Proceedings of USENIX Workshop on Distributed and Multiprocessor Systems, Fort Lauderdale, FL*, pages 373–390. Usenix Association, 1989.
14. W R Dieter and J E Lumpp Jr. User-level checkpointing for LinuxThreads programs.
15. G.W. Dunlap, D.G. Lucchetti, M.A. Fetterman, and P.M. Chen. Execution replay of multiprocessor virtual machines. *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, 2008.
16. S. Ffinfrocken. Transparent migration of Java-based mobile agents. *Springer*, Volume 147:26–37, 1998.
17. D Freedman. Experience Building a Process Migration Subsystem for UNIX. In *USENIX Winter*, pages 349–356, 1991.

18. Erik Hendriks. BProc: the Beowulf distributed process space. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 129–136, New York, NY, USA, 2002. ACM.
19. J. Howell. Straightforward Java persistence through checkpointing. *Advances in Persistent Object Systems: Proceedings of the Int'l Workshop on Persistent Object Systems (POS) and the Int'l Workshop on Persistence \& Java (PJAVA)*, pages 322–334, 1999.
20. T. Illmann, T. Krueger, F. Kargl, and M. Weber. Transparent migration of mobile agents using the java platform debugger architecture. *Lecture Notes in Computer Science*, pages 198–212, 2001.
21. S.T. King, G.W. Dunlap, and P.M. Chen. Debugging operating systems with time-traveling virtual machines. *Proc. USENIX Annual Technical Conference*, pages 1–15, 2005.
22. C. Kintala. Checkpointing and its applications. *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, (June):22–31, 1995.
23. F.C.M. Lau. JESSICA2: a distributed Java Virtual Machine with transparent thread migration support. *Proceedings. IEEE International Conference on Cluster Computing*, pages 381–388.
24. M Litzkow and M Solomon. Supporting checkpointing and process migration outside the UNIX kernel. In *In Proceedings of the Winter 1992 USENIX Conference*, 1992.
25. Wolfgang Lux. Adaptable object migration: concept and implementation. *SIGOPS Oper. Syst. Rev.*, 29(2):54–69, 1995.
26. R.K.K. Ma, C.L. Wang, and F.C.M. Lau. M-JavaMPI: A Java-MPI binding with process migration support. *The Second IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 1–9, 2002.
27. B P Miller, D L Presotto, and M L Powell. DEMOS/MP: the development of a distributed operating system. *Softw. Pract. Exper.*, 17(4):277–290, 1987.
28. D S Milojicic, W Zint, A Dangel, and P Giese. Task Migration on the top of the Mach Microkernel. In *USENIX MACH III Symposium table of contents*, pages 273–290. USENIX Association Berkeley, CA, USA, 1993.
29. DS Milojicic, F Douglis, Y Paindaveine, and R. Process migration. *ACM Computing Surveys*, 2000.
30. A. Mirkin, A. Kuznetsov, and K. Kolyshkin. Containers checkpointing and live migration. *Ottawa Linux Symposium*, 2008.
31. J. Napper, L. Alvisi, and H. Vin. A fault-tolerant java virtual machine. *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003), DCC Symposium*, pages 425–434, 2003.
32. J.K. Ousterhout, a.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite network operating system. *Computer*, 21(2):23–36, Fevereiro 1988.
33. S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K.H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 177–192, 2009.
34. E Pinheiro. Truly-transparent checkpointing of parallel applications. *citeseer. ist. psu. edu/434768. html*.
35. James S Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. *In USENIX Winter 1995 Technical Conference, January*, 1995.
36. Feng Qin, Joseph Tucek, Yuanyuan Zhou, and Jagadeesan Sundaresan. Rx: Treating bugs as allergies. *ACM Transactions on Computer Systems*, 25(3):7, 2007.
37. R F Rashid and G G Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the eighth ACM symposium on Operating systems principles*, pages 64–75. ACM New York, NY, USA, 1981.
38. J.B.H. Roebbers, J. Simler, P. Welch, and D. Wood. Towards Strong Mobility in the Shared Source CLI. *Communicating process architectures 2005: WoTUG-28: proceedings of the 28th WoTUG Technical Meeting, 18-21 September 2005, Technische Universiteit Eindhoven, The Netherlands*, pages 363–373, 2005.
39. E. Roman. A survey of checkpoint/restart implementations. *Citeseer*, pages 1–9, 2002.
40. T Sakamoto, T Sekiguchi, and A Yonezawa. Bytecode transformation for portable thread migration in Java. *Lecture Notes in Computer Science*, pages 16–28, 2000.
41. T Sekiguchi, H Masuhara, and A Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. *in Coordination Models and Languages*, 1999.
42. S.M. Srinivasan, S. Kandula, C.R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. *usenix.org*, 2004.

43. E. Steketee and P. Moseley. Implementation of process migration in Amoeba. *14th International Conference on Distributed Computing Systems*, pages 194–201.
44. G. Stellner. CoCheck: checkpointing and process migration for MPI. *Proceedings of International Conference on Parallel Processing*, pages 526–531.
45. T. Suezawa. Persistent execution state of a Java virtual machine. *Proceedings of the ACM 2000 conference on Java Grande*, pages 160–167, 2000.
46. N Suri, J M Bradshaw, M R Breedy, P T Groth, G A Hill, and R Jeffers. Strong mobility and fine-grained resource control in NOMADS. *Lecture Notes in Computer Science*, pages 2–15, 2000.
47. M Theimer, K Lantz, and D Cheriton. Preemptable remote execution facilities for the V-System. *ACM SIGOPS Operating Systems Review*, 19(5):12, 1985.
48. E Truyen, B Robben, B Vanhaute, T Coninx, W Joosen, and P Verbaeten. Portable support for transparent thread migration in Java. *Lecture notes in computer science*, pages 29–43, 2000.
49. J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. *ACM SIGOPS Operating Systems Review*, 41(6):131–144, 2007.
50. B Walker, G Popek, R English, C Kline, and G Thiel. The LOCUS distributed operating system. In *Proceedings of the ninth ACM symposium on Operating systems principles*, page 70. Acm, 1983.
51. H Zhong and J Nieh. CRAK: Linux checkpoint/restart as a kernel module. *Technical Report CUCS-014-01, Department of Computer Science, Columbia University*, pages 1–18, 2001.
52. Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pierre Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. *Software: Practice and Experience*, 23(12):1305–1336, Dezembro 1993.
53. W Zhu. The Development of an Environment to Study Load Balancing Algorithms, Process Migration and Load Data Collection. *PhD thesis, University of New South Wales*, 1992.