Depuração simbólica extensível para sistemas de objetos distribuídos

Giuliano Mega

Dissertação apresentada
ao Instituto de Matemática e Estatística
da Universidade de São Paulo
para obtenção
do grau de Mestre em Ciências

Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. Fabio Kon

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES/IBM/Google

São Paulo, Março de 2008

Depuração simbólica extensível para sistemas de objetos distribuídos

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Giuliano Mega
e aprovada pela Comissão Julgadora.

Banca Examinadora:

- Prof. Dr. Fabio Kon (orientador) - IME-USP.
- Prof. Dr. Alan Mitchell Durham - IME-USP.
- Prof. Dr. Renato Fontoura de Gusmão Cerqueira - PUC-RIO.

à minha mãe Cristina, ao meu pai Élio
ao meu irmão Stefano e à minha namorada Juliana

Agradecimentos

O professor Fabio Kon é um orientador no sentido mais verdadeiro da palavra. Sua sábia orientação teve um efeito inegável, tanto sobre o alcance deste trabalho, quanto sobre as minhas visões da ciência da computação e de como conduzir uma pesquisa científica, de modo geral. Agradeço imensamente por todos esses ensinamentos, pelo apoio durante este longo mestrado e, é claro, pela amizade. Sinto que meu futuro como pesquisador será muito mais rico por causa disso.

O trabalho de servir em uma banca é muitas vezes uma tarefa árdua e sem agradecimentos. Gostaria, portanto, de agradecer ao professor Alan Mitchell Durham, de quem aprendi muito durante a graduação, e ao professor Renato Cerqueira, colaborador de longa data do nosso grupo de pesquisa, pela gentileza de participarem em minha banca de mestrado. Tenho certeza de que ambos farão contribuições importantes a este trabalho.

Não seria justo deixar de reconhecer as contribuições dos professores Marco Dimas Gubitoso e Francisco Reverbel para a formação e o amadurecimento de muitas das principais idéias deste trabalho. Agradeço pelas horas de discussões estimulantes, pela generosidade e pela amizade. Agradeço duplamente ao Professor Marco Dimas Gubitoso, por ter aceito ser meu orientador de programa no início deste mestrado. Seu apoio e conselhos foram essenciais para que eu sentisse segurança ao me decidir por este tópico.

Agradeço aos meus pais, Cristina e Élio Mega, e ao meu irmão, Stefano Mega, pela paciência e pelo apoio, não só durante o decorrer deste mestrado, mas sempre.

Agradeço à minha namorada e amiga Juliana Teixeira Fiquer pela paciência inesgotável nos finais-de-samana e pelo carinho, sem os quais minha vida teria sido muito mais difícil durante esses últimos anos.

Agradeço a todos os amigos que tornaram essa minha passagem pelo IME tão especial, interessante e divertida. Agradeço principalmente aos companheiros de laboratório Andrei Goldchleger, Alexandre

Vidal, Arlindo da Conceição, José Braga, Eduardo Guerra, Marco Netto, Raphael Camargo, Ricardo Abrantes, Vinicius Pinheiro e Vladimir Rocha, bem como aos amigos Danilo Sato, Hugo Corbucci, Igor Sucupira, Julian Monteiro, Kelly Braghetto, Mariana Bravo, Pedro Losco e, é claro, ao professor Alfredo Goldman. Sua presença e ajuda, direta e indireta, foram muito importantes para mim. Finalmente, agradeço ao meu grande amigo Daniel Cordeiro, pela eterna disponibilidade em ler os manuscritos e versões finais dos meus artigos, pelas sugestões dadas na versão preliminar deste texto e, é claro, pela amizade.

Muito Obrigado!

Resumo

Depurar sistemas distribuídos continua uma tarefa difícil, mesmo após 30 anos de pesquisa intensa. Embora essa situação possa ser parcialmente atribuída à complexidade das execuções concorrentes, o rápido passo de desenvolvimento das plataformas e tecnologias para computação distribuída também carrega a sua parcela de culpa, por encurtar a vida de muitas ferramentas potencialmente úteis. Neste trabalho, apresentamos uma análise dos principais problemas, técnicas e ferramentas ligados à depuração de sistemas concorrentes e discutidos na literatura. Baseados nessa análise, desenvolvemos e apresentamos uma nova técnica, simples e portátil, que pode ser aplicada a sistemas distribuídos que utilizam chamadas síncronas e bloqueantes. Essa técnica, concebida para sobreviver à heterogeneidade, é validada por meio da implementação de um arcabouço escrito para plataforma Eclipse e instanciado para sistemas de objetos distribuídos baseados em Java/CORBA.

Abstract

After over thirty years of intense research, debugging distributed systems is still regarded as a difficult task. While this situation could be partially blamed on the fact that concurrent executions are complex, the fast pace of evolution witnessed with distributed computing technologies have also played its by shortening the lifespan of many potentially useful debugging tools. This work presents an analysis of the main issues, techniques and tools in the field of parallel, distributed, and concurrent debugging in general. Based on this analysis, we develop and present a simple and portable technique targeted at synchronous-call-based distributed systems. This technique, designed for portability, is validated through the implementation of an Eclipse-based framework that is instantiated for Java/CORBA distributed object systems.

Sumário

RESUMO 1

ABSTRACT 2

1 INTRODUÇÃO 1

1.1 Sistemas paralelos e distribuídos 3

1.2 Desenvolvimento de sistemas distribuídos 6
1.2.1 Sistemas de middleware 7
1.2.2 Ciclo de desenvolvimento e execuções distribuídas 11

1.3 Sumário 15

2 EXECUÇÕES CONCORRENTES E DISTRIBUÍDAS 16

2.1 Um modelo para execuções distribuídas 17

2.2 Impacto em atividades de teste e depuração 19

2.3 Relógios escalares e vetoriais 23

2.4 Cortes consistentes e observações corretas 27

2.5 Prelúdio para as condições de corrida: sistemas de memória compartilhada 28

2.6 Condições de corrida – o cerne do indeterminismo em sistemas concorrentes 30
2.6.1 Condições de corrida em sistemas de memória compartilhada 32
2.6.2 Condições de corrida em sistemas baseados em passagem de mensagens 37

2.7 Os efeitos do observador e labirinto 39

2.8 Sumário 39

3 DEPURAÇÃO DE SISTEMAS DISTRIBUÍDOS 42

3.1 Sistemas de depuração e depuradores para sistemas distribuídos 43

3.2 Trabalhos relacionados 48
3.2.1 Execução reversa 48
3.2.2 Re-execução determinística (*replay debugging*) 51
3.2.3 Re-execução em sistemas de memória compartilhada e a detecção de condições de corrida 57
3.2.4 Re-execução de entrada 68
3.2.5 Re-execução de interrupções 75
3.2.6 Mais sobre monitoramento e análise 79
3.2.7 Análise e visualização 83

3.3 Heterogeneidade – a grande vilã 87

3.4 Sumário 92

4 DEPURAÇÃO DE SISTEMAS DE OBJETOS DISTRIBUÍDOS 95

4.1 Visão geral de um sistema de objetos distribuídos 95

4.2 Ferramentas usuais 99
4.2.1 O comando *print* 99
4.2.2 Depuradores simbólicos 100

4.3 Depuração e *threads* distribuídos 102
4.3.1 Proposta deste trabalho 103
4.3.2 Caracterização do *thread* distribuído 103
4.3.3 Visualização e sinergia com depuradores simbólicos 112
4.3.4 Valor dos *threads* distribuídos como ferramenta de depuração 116
4.3.5 Re-execução determinística 117

4.4 Atividades de apoio – gerenciamento de processos e interação remota 120

4.5 Sumário 121

5 O *GLOBAL ON-LINE DEBUGGER* (G.O.D.) 123

5.1 Arquitetura 123

5.2 Representação dos *threads* distribuídos 124
5.2.1 Adendo sobre re-execução e desempenho 127

5.3 Rastreio 127
5.3.1 Reflexão e técnicas interativas 128
5.3.2 Interação e técnicas *on-line* limitadas 130
5.3.3 Rastreio síncrono e assíncrono: s*uspend* em *threads* distribuídos 133
5.3.4 Dividindo *threads* distribuídos na presença de falhas 139

5.4 Os agentes locais 141
5.4.1 Implementação Java/CORBA 142
5.4.2 Interceptadores, eventos e o modo de execução passo-a-passo 149
5.4.3 Viabilidade em outras linguagens e sistemas de middleware 152

5.5 O agente central 156
5.5.1 Extensibilidade: mais sobre meta-sistemas 156
5.5.2 O meta-modelo de depuração do Eclipse 157
5.5.3 Mais sobre o modo de execução passo-a-passo 163
5.5.4 Particularidades na integração com o depurador da JPDA 166
5.5.5 Portátil, interoperável ou extensível? 169
5.5.6 Considerações sobre escalabilidade 171
5.5.7 Mecanismos de análise automática 178

5.6 O controlador de processos remotos 181

5.7 Limitações e trabalhos futuros 189

5.8 Sumário 194

6 CONCLUSÕES 196

6.1 Sumário de contribuições **197**

6.2 Publicações **198**

1 Introdução

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

-- *Leslie Lamport*

A ampla aceitação dos padrões e tecnologias da Internet presenciada nos dias de hoje tornam a importância dos sistemas distribuídos algo difícil de contestar. Comunidades científicas utilizam o poder de milhares de computadores espalhados pelo mundo inteiro, trabalhando em conjunto na solução de problemas computacionalmente intensos. Empresas desenvolvem software de forma mais eficiente por meio de processos de desenvolvimento distribuídos (tornados viáveis por ferramentas de desenvolvimento colaborativo e distribuído), ampliam seu campo de vendas através do comércio eletrônico e se tornam capazes de operar mais eficientemente pela integração de informações em suas subsidiárias. Indivíduos se beneficiam da rede quase ilimitada de informações e serviços disponibilizados na *World Wide Web*.

Apesar de todos os benefícios, no entanto, os sistemas distribuídos continuam vinculados a um desagradável estigma: o de serem difíceis de construir corretamente. Esse estigma pode ser parcialmente justificado pelas dificuldades técnicas enfrentadas pelos pioneiros, em especial no que diz respeito à programação em ambientes heterogêneos. Nesse sentido, a introdução dos sistemas de middleware e das máquinas virtuais representou um importante avanço nas técnicas de construção de sistemas distribuídos – a contribuição trazida pela especificação e pelas implementações do padrão CORBA [151] e pela introdução da linguagem Java [74], na década de 90, são inegáveis.

A geração atual de sistemas de middleware para sistemas distribuídos não trata, no entanto, de algumas questões intrínsecas ao desenvolvimento desses sistemas; questões que representam empecilhos tão grandes quanto a própria heterogeneidade e que transcendem o seu caráter técnico. O não-determinismo característico das execuções distribuídas impacta de forma negativa as atividades de desenvolvimento mais fundamentais, como teste e depuração. As dificuldades ligadas à captura e análise de estados globais tornam a reconstrução das execuções distribuídas difícil, limitando as possibilidades de detecção de comportamentos incorretos. Além disso, o grande volume de dados produzidos pela execução de um sistema distribuído – ainda que de pequeno porte – pode rapidamente sobrepujar os esforços de análise de um desenvolvedor em busca de pistas a respeito das causas de um comportamento.

Um fato intrigante é que uma busca não muito ampla na literatura que aborda a depuração e teste de sistemas concorrentes (paralelos, distribuídos e pseudo-paralelos [213]) revela uma riqueza muito grande de técnicas, abordagens e implementações. Uma boa parte das ferramentas produzidas, no entanto, tiveram a sua vida útil abreviada por estarem acopladas a sistemas operacionais ou ambientes de desenvolvimento de pouca difusão, cuja aplicabilidade é restrita, muitas vezes, aos grupos de pesquisa que as desenvolveram. Outras são restritas à comunidade de sistemas paralelos, encontrando pouco ou nenhum uso na comunidade de sistemas distribuídos. O resultado é que esse conhecimento acumulado ao longo de mais de 30 anos de pesquisa intensa muitas vezes não encontra lugar nas tecnologias modernas – especialmente quando o assunto são sistemas distribuídos.

O foco deste trabalho é na depuração (e, em menor grau, no teste) de sistemas distribuídos, mais especificamente de sistemas de objetos distribuídos baseados em chamadas síncronas e bloqueantes (RPC/RMI). A escolha pelas chamadas síncronas e bloqueantes se deve à importância do paradigma, tanto historicamente quanto nos tempos atuais. Conforme veremos mais adiante, nossa análise também revela que um dos principais problemas enfrentados pelas ferramentas de depuração produzidas até então é a rápida evolução das tecnologias (tanto de hardware, quanto middleware e sistemas operacionais). Os principais objetivos e, por conseqüência, as principais contribuições deste trabalho são, portanto:

1. Apresentar um estudo dos problemas intrínsecos à depuração de sistemas paralelos e distribuídos, com enfoque especial nas questões ligadas ao não-determinismo e aos problemas de observabilidade.
2. Analisar as principais técnicas e ferramentas desenvolvidas para a depuração de sistemas paralelos e distribuídos ao longo das últimas duas décadas.
3. O desenvolvimento de uma técnica de depuração simples, portátil e amplamente aplicável a uma classe de sistemas distribuídos relevante (os sistemas baseados em chamadas síncronas e bloqueantes).
4. A construção de uma ferramenta de depuração útil, simples e portátil que seja baseada nessa técnica.

É importante frisar que, embora uma das propostas deste trabalho seja o estudo, em alguns pontos aprofundado, dos principais problemas, ferramentas e técnicas ligados à depuração e ao teste de sistemas distribuídos (e concorrentes em geral), não é objetivo deste trabalho apresentar soluções para todos os problemas estudados. Em particular, tanto a técnica de depuração desenvolvida quanto a ferramenta implementada cobrem apenas uma pequena parte dos problemas estudados. Uma

apresentação adequada, tanto das questões envolvidas quanto dos motivos que tornam essas questões de difícil abordagem, será conduzida no Capítulo 3.

O restante deste capítulo discute os seguintes tópicos:

- **Sistemas paralelos e distribuídos:** a Seção 1.1 procura contextualizar a área de aplicação do nosso trabalho, colocando em evidência algumas das diferenças entre sistemas paralelos e distribuídos, no sentido usual da aplicação desses termos.
- **Desenvolvimento de sistemas distribuídos:** a Seção 1.2 aborda o desenvolvimento de sistemas paralelos e distribuídos, introduzindo as principais dificuldades. A Seção 1.2.1 apresenta uma perspectiva histórica dos sistemas de middleware, situando-os como grandes facilitadores do desenvolvimento de sistemas distribuídos e colocando a RPC/RMI como um dos paradigmas mais populares.
- **Ciclo de Desenvolvimento e Execuções Distribuídas:** a Seção 1.2.2 trata de atividades fundamentais do desenvolvimento de software, especificamente, do ciclo de codificação – execução (teste) – depuração, introduzindo parte dos problemas trazidos pelos sistemas concorrentes (distribuídos, paralelos e pseudo-paralelos [213]) a essas atividades.

1.1 Sistemas paralelos e distribuídos

Os sistemas paralelos estão historicamente associados à superação de barreiras na Ciência da Computação. Sistemas paralelos são concebidos sob a idéia de que é possível transpor as limitações de poder computacional impostas por uma única peça de hardware através da combinação de várias delas. Dessa maneira, os sistemas paralelos tornam-se particularmente importantes em certas áreas da ciência – as áreas provedoras dos chamados *Grand Challenge Problems* [228] – onde a demanda por poder computacional constantemente excede a capacidade do hardware disponível.

Sistemas distribuídos, por sua vez, apresentam uma origem mais diversa. De acordo com Tanenbaum [212], o surgimento dos sistemas distribuídos pode ser atribuído a dois fatores desencadeadores: o barateamento dos computadores pessoais e o desenvolvimento das redes locais, aliados ao custo elevado de aquisição dos computadores de grande porte (outros autores [50] também compartilham dessa visão). Nesse sentido, os sistemas distribuídos podem ter a sua importância inicial atribuída a um potencial de redução de custos, bem como à viabilização econômica de um poder computacional antes inacessível.

Baseados apenas nessas informações, poderíamos inferir que as motivações que levam ao desenvolvimento desses dois tipos de sistemas são bastante similares. A razão histórica por trás da disseminação dos sistemas distribuídos não é, no entanto, a obtenção de poder computacional a custos modestos, mas sim a demanda por formas de compartilhamento de recursos [97].

O termo recurso é bastante amplo, bem como são amplos e diversos os motivos que levam ao compartilhamento de um recurso. A redução de custos é certamente um deles: os dirigentes de uma empresa podem decidir reduzir os gastos com equipamentos de impressão, passando a compartilhar uma única impressora entre todas as estações de trabalho, ao invés de manterem uma impressora por estação de trabalho. Atividades colaborativas são um outro motivo: grupos de pessoas localizadas em regiões geograficamente dispersas podem desejar colaborar na elaboração de um documento. Para tanto, um membro do grupo poderia disponibilizar o documento num lugar acessível por todos: um servidor conectado à Internet (um exemplo em larga escala dessa idéia é a *Wikipedia* [227]). O compartilhamento de informações pode ser ainda outro motivo: um indivíduo ou organização pode decidir, por razões diversas, publicar informações na *World Wide Web* para que todos que tenham navegadores instalados possam acessá-las.

Os exemplos dados acima são de sistemas que não poderiam ser não-distribuídos em função da natureza do problema que se propõem a resolver. Nós chamamos esses sistemas de *sistemas inerentemente distribuídos*.

Definição 1-1: Um *sistema inerentemente distribuído* é um sistema cuja distribuição resulta da natureza do próprio problema que leva à concepção do sistema.

O que esta discussão procura mostrar é que as razões que levam à construção de sistemas distribuídos são mais amplas e muitas vezes distintas das razões que levam à construção de sistemas paralelos. Isso é particularmente verdadeiro no caso dos sistemas inerentemente distribuídos. Exemplos de sistemas inerentemente distribuídos incluem as grades computacionais (tais como o projeto Globus [218], o Condor [217], o InteGrade [73] e o OurGrid [35]), a plataforma BOINC [7], as redes GIS [23] e a *World Wide Web*. Sistemas inerentemente distribuídos não podem, em geral, serem substituídos por sistemas não-distribuídos, ao menos não sem grandes dificuldades de ordem técnica ou financeira. Sistemas distribuídos são também freqüentemente associados a um melhor potencial de tolerância a falhas e escalabilidade [212].

Cabe aqui uma distinção mais precisa entre o que chamamos de sistemas paralelos e o que chamamos de sistemas distribuídos. Muito embora um sistema distribuído seja, antes de mais nada, um sistema paralelo, os dois termos denotam, no sentido usual, sistemas com características bastante distintas. De acordo com Kranzlmüller [104], o termo *sistema paralelo* faz referência a sistemas compostos por unidades de processamento (também chamadas de *nodos*) "simples" – muitas vezes constituídas por uma única CPU – e interconectadas por barramentos confiáveis e de alta velocidade.

Sistemas paralelos podem ainda conter componentes especializados – processadores vetoriais como os NEC SX-9 ou os Fujitsu VPP – além de componentes especificamente projetados para cada sistema. Embora uma boa parte dos sistemas paralelos construídos atualmente empreguem componentes padronizados e comercialmente disponíveis (*Commercial Off-The-Shelf,* ou COTS), esses sistemas computacionais contam, em geral, com diversos componentes especializados (não-COTS) que os tornam únicos (vide os barramentos de interconexão de MPPs [85] modernos, como os Cray XT3). Sistemas paralelos também apresentam, com freqüência, memória compartilhada (caso de todos os SMPs [85]). Exemplos de sistemas paralelos recentes são o *NEC Earth Simulator,* o *IBM Blue Gene/L*, o *Intel ASCI Red* e diversos outros na lista dos 500 supercomputadores mais poderosos do mundo [222].

Sistemas distribuídos são, por outro lado, tipicamente construídos a partir da aglomeração de sistemas completos menores – com freqüência estações de trabalho – interligados por uma malha de rede (ainda segundo Kranzlmüller [104]). Essa rede pode ser composta por meios de transmissão heterogêneos, apresentar latência elevada, quando comparada à latência dos barramentos utilizados em sistemas paralelos, e não ser confiável. Clark, Jensen e Reynolds [36] chegam a conclusões bastante semelhantes ao afirmarem (traduzindo para o Português) que "a computação fisicamente distribuída surge sempre que um sistema computacional, composto por uma certa quantia de nodos de processamento, possui uma razão entre desempenho nodal e desempenho internodal (primariamente latência, mas também largura de banda) significativamente alta*"*. Conteúdo de teor semelhante também pode ser encontrado nas análises de Hwang e Xu [85].

A Figura 1-1 sumariza as diferenças entre sistemas distribuídos e paralelos discutidas até agora. A medida de *qualidade do barramento* indica, na figura, uma relação entre latência e confiabilidade – quanto maior a qualidade do barramento, menor a sua latência e maior a sua confiabilidade. A sobreposição das duas classes ilustrada na Figura 1-1 é para indicar que a barreira entre sistemas paralelos e distribuídos pode ser um tanto quanto difícil de estabelecer: um sistema distribuído pode ser composto por um agrupamento de sistemas paralelos, ou até mesmo de outros sistemas distribuídos (Sistemas Híbridos na Figura 1-1).

Os sistemas de objetos distribuídos, particularmente, são freqüentemente compostos por nodos cuja execução interna é pseudo-paralela (i.e., *timesharing*) [212]. Com a introdução dos processadores *multicore*, os nodos se tornam sistemas paralelos reais, dotados de memória compartilhada. Portanto, essa hibridização ocorre também nos sistemas que são objeto do nosso trabalho.

Há ainda o caso dos clusters dedicados (os *Cluster Of Workstations,* COWs na Figura 1-1), muitas vezes interligados por redes de alta velocidade (acima de 1 GB/s), cuja finalidade principal é

o tratamento de problemas difíceis e computacionalmente intensos. Esses sistemas são sistemas distribuídos, mas cumprem a mesma finalidade de muitos sistemas paralelos, sendo, por vezes, mais poderosos e eficazes do que muitos deles.

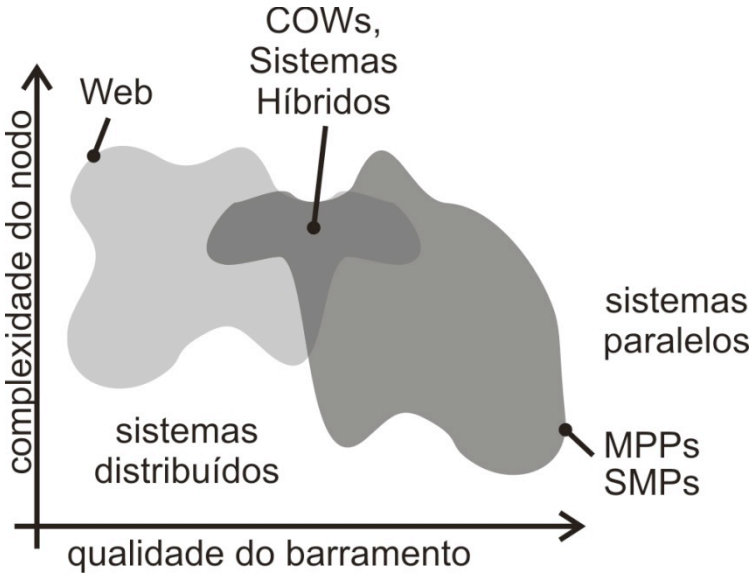

Figura 1-1. Sentido usual dos termos *sistema paralelo* e *sistema distribuído*.

1.2 Desenvolvimento de sistemas distribuídos

Conforme mencionamos anteriormente, os sistemas distribuídos são reconhecidamente difíceis de construir, sendo que esse estigma pode ser atribuído a dois fatores fundamentais: os problemas de interoperabilidade que surgem no contexto de ambientes heterogêneos e a complexidade das execuções distribuídas (e das execuções concorrentes em geral).

O primeiro problema leva a sistemas dotados de arquiteturas, projetos e código mais complexos. O uso de plataformas e linguagens distintas num mesmo sistema obriga a adoção de uma maior variedade de ferramentas de desenvolvimento. Essas características dificultam o desenvolvimento e elevam a curva de aprendizado do sistema, tornando também a manutenção mais custosa. O segundo problema, por sua vez, torna a compreensão do comportamento (execução) do sistema mais difícil de observar e analisar, novamente contribuindo com elevações no custo de desenvolvimento.

Nesta seção, discutimos algumas das dificuldades envolvidas no desenvolvimento de aplicações distribuídas em ambientes heterogêneos, bem como as soluções historicamente adotadas para resolvê-las. Analisamos também o ciclo tradicional de teste e depuração, apontando algumas das dificuldades que surgem no contexto de sistemas concorrentes.

1.2.1 Sistemas de middleware

Os problemas de interoperabilidade são tratados, em geral, pela adoção de sistemas de software capazes de intermediar e traduzir a comunicação entre plataformas distintas. Esses sistemas são conhecidos como sistemas de middleware [102]. De fato, o desenvolvimento e amadurecimento das tecnologias de middleware contribuíram de forma significativa com a facilitação da interoperabilidade. A abundância de plataformas que contam com implementações da MPI [134], por exemplo, tornou a construção de *clusters* heterogêneos para computação de alto desempenho muito mais simples hoje do que há duas décadas.

A definição de middleware é um tanto quanto imprecisa, mas parece haver algum grau de concordância na literatura [17, 21, 102, 212]. O significado da palavra middleware, conforme usada neste texto, é dado pela seguinte definição:

Definição 1-2: Um *sistema de* middleware é um sistema de software genérico, localizado entre a aplicação e o sistema operacional. Sistemas de middleware provêm uma variedade de serviços (ex. serviços de comunicação, nome, diretório, armazenagem), potencialmente distribuídos, às aplicações que os utilizam.

A principal força que motiva os sistemas de middleware é a integração de serviços e aplicações [17]. A Figura 1-2 dá uma visão esquemática do papel assumido por um sistema de middleware em um sistema distribuído.

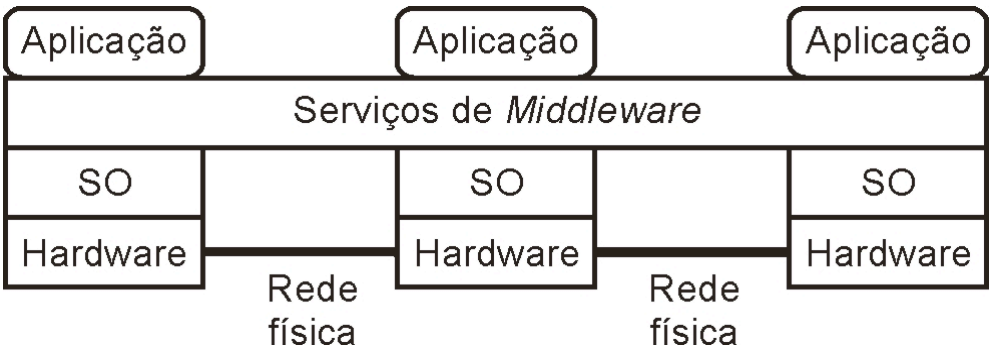

Figura 1-2. Sistema de middleware.

Os sistemas de middleware normalmente recorrem à adição de camadas e à definição de interfaces e protocolos de dados padronizados para atingir o grau de interoperabilidade e portabilidade requeridos. A tendência à padronização como forma de melhorar a interoperabilidade e a portabilidade de aplicações distribuídas pode ser observada, no entanto, já há algum tempo, com a introdução dos *Berkeley Sockets* [212] em 1982. Embora os *sockets* (ou soquetes, como vamos chamá-los doravante) não sejam considerados middleware, eles empregam muitos dos mesmos

princípios (interfaces padronizadas, protocolos padronizados (TCP/IP)) para atingirem as mesmas finalidades: a comunicação interoperável.

De fato, segundo Ronda Hauben [79], uma das principais razões que levou à adoção do TCP/IP na ARPANET em 1983 foi a sua capacidade de interconectar redes incompatíveis. Vale notar que os soquetes *Berkeley* e POSIX [90] sejam talvez as APIs de comunicação mais difundidas de que se tem notícia no mundo da computação distribuída. Embora o TCP/IP tenha contribuído de forma significativa para a interoperabilidade entre plataformas heterogêneas, o nível de abstração provido pelos soquetes ainda era inadequado. Segundo Michi Henning [80]: "*In the early '90s, persuading programs on different machines to talk to each other was a nightmare, especially if different hardware, operating systems, and programming languages were involved: programmers either used sockets and wrote an entire protocol stack themselves or their programs didn't talk at all.*" De fato, soquetes foram projetados para entrega de bytes. Qualquer atividade que exija uma semântica mais refinada fica a cargo do desenvolvedor.

Um ano antes do primeiro lançamento dos *Berkeley Sockets*, Bruce Jay Nelson publicou em sua tese de doutorado, intitulada *Remote Procedure Call* [18, 143] (RPC), um modelo de comunicação interprocessos que viria a se tornar, alguns anos depois, um dos mais importantes e aceitos modelos para a construção de sistemas distribuídos. A idéia, que foi proposta por James White em 1976 [226], era simples: abstrair a interação entre clientes e servidores sob a forma de procedimentos remotos, que poderiam ser utilizados por clientes de maneira semelhante às chamadas de procedimentos locais. O paradigma tornou-se tão popular que, em 1988, Andrew Tanenbaum publica uma crítica ao uso indiscriminado das RPCs dizendo que, dentro da comunidade de pesquisa de sistemas operacionais distribuídos, as RPCs atingiram um status de "vaca sagrada" [215]. As primeiras implementações de RPCs também apresentavam problemas de interoperabilidade, já que os protocolos de nível de aplicação não eram padronizados.

A demanda por interoperabilidade aliada à popularidade das RPCs acabou por levar ao desenvolvimento das primeiras plataformas de middleware que, não surpreendentemente, eram baseadas neste modelo. O *Distributed Computing Environment* [219] (DCE) foi um dos pioneiros na indústria. O DCE já apresentava uma linguagem de especificação de interfaces (IDL) e um protocolo independente de arquitetura, sendo capaz de tratar questões incômodas como a famosa questão de ordenação de bits em diferentes processadores [175].

Segundo Tanenbaum [212], entretanto, o DCE chegou no momento errado. Por ser um dos primeiros sistemas concebidos como middleware (o DCE não fazia parte do sistema operacional), o DCE teve de passar por um período de aceitação prolongado. Infelizmente, durante esse período de aceitação começaram a surgir os primeiros sistemas de middleware orientados a objeto, que

rapidamente foram declarados pela indústria como o futuro da computação distribuída. O modelo puramente baseado em procedimentos do DCE foi considerado ultrapassado, o que acabou por obrigar a OSF a introduzir o conceito de objetos distribuídos no sistema. A adaptação resultou num sistema com várias idiossincrasias, que fizeram com o DCE nunca chegasse a gozar de muita popularidade.

Tanto o DCE quanto os sistemas de middleware que o seguiram (CORBA, DCOM e Java/RMI) apresentam algumas características peculiares. A primeira delas é que todos são preponderantemente baseados em chamadas síncronas e bloqueantes (RPC/RMI) – muito embora CORBA permita chamadas assíncronas (*one-way*), essas não são utilizadas com tanta freqüência. A segunda tendência foi a incorporação de um conjunto de serviços (tais como serviços de nomes, segurança, transações e diretórios) na própria plataforma. Os sistemas de middleware da década de 1990 deixaram, portanto, de tratar apenas dos aspectos de interoperabilidade e comunicação, passando a abordar também a questão da padronização e interoperabilidade de serviços [17]. Sob esse aspecto, a comunicação passa a ser apenas um dos serviços providos pelo middleware.

As tecnologias de objetos distribuídos desenvolvidas durante a década de 1990 têm, entretanto, as suas limitações. A falta de padrões para o gerenciamento automático de dependências entre serviços, bem como para a configuração, gerenciamento de ciclo de vida, implantação e evolução de serviços levaram à construção dos sistemas de middleware orientados a componentes. Além disso, as limitações do modelo de RPC/RMI sob redes de alta latência começaram a se tornar mais evidentes [224], bem como tornaram-se evidentes os custos escondidos na aparente simplicidade da extensão do modelo de programação baseado *threads* e memória compartilhada a sistemas distribuídos [117]. No mais, tanto os primeiros sistemas baseados em RPC quanto RMI tentavam unificar os modelos de programação centralizada e distribuída; algo que, segundo Jim Waldo, é uma idéia que surge com freqüência na Ciência da Computação, mas que está fadada ao fracasso [225]. Vale notar que os sistemas de middleware de objetos distribuídos foram progressivamente abrindo mão dessa "transparência a qualquer custo" em favor de viabilizarem a construção de sistemas mais robustos, sem abrirem mão da conveniência das chamadas bloqueantes. O middleware moderno para objetos distribuídos diferencia explicitamente objetos remotos de objetos locais, com marcadores e exceções especiais.

Segundo Steve Vinoski e Douglas Schmidt [178], o middleware baseado em componentes surge com a missão de facilitar a composição, configuração e instalação de serviços reutilizáveis de forma rápida e robusta. A noção de contêiner – uma entidade que gerencia os componentes – permeia os sistemas de middleware baseados em componentes. A principal contribuição do middleware baseado em componentes é o reuso. As funcionalidades trazidas para dentro do middleware podem

ser reutilizadas por todas as aplicações e serviços nele implantados, reduzindo substancialmente os esforços de desenvolvimento. Não por coincidência, os contêineres fazem uso extensivo de inversão de controle – uma característica importante, segundo Ralph Johnson, dos *arcabouços* orientados a objetos [61, 96]. Nesse sentido, podemos encarar os componentes implantados num contêiner como extensões a um *arcabouço*. Além disso, o modelo induz a criação de barreiras funcionais entre componentes, forçando a interação a ocorrer por meio de interfaces bem-definidas, o que acaba favorecendo a composição e melhorando as chances de reuso e interoperabilidade entre componentes (e serviços). As especificações de sistemas de middleware baseados em componentes também definem, tipicamente, mecanismos padronizados para a execução dos componentes em servidores genéricos, favorecendo a portabilidade.

A nova geração de sistemas de middleware baseados em componentes (tais como as implementações das especificações dos *Enterprise Java Beans* [204], do *CORBA Component Model* [152], ou da *Plataforma OSGi* [154]) também apresentam mudanças importantes no sentido de não privilegiarem as RPCs/RMIs. Comunicações assíncronas e/ou baseadas em mensagens/eventos são igualmente bem acomodadas por essas novas plataformas.

Para tornar a nossa discussão sobre a evolução do middleware mais completa, devemos abordar os sistemas de middleware orientados a mensagens. Sistemas de middleware orientados a mensagens são, como o próprio nome indica, sistemas cuja abstração de comunicação predominante é a mensagem. Tanenbaum [212] classifica a passagem de mensagens de acordo com a durabilidade (mensagens transientes/persistentes) e o sincronismo (mensagens síncronas/assíncronas).

Um dos representantes mais significativos dos sistemas de middleware para passagem de mensagens é a *Message Passing Interface* (MPI) [134]. A MPI é uma especificação de um sistema de middleware sofisticado, projetado para as necessidades da comunidade de computação de alto desempenho (principalmente para os usuários de MPPs [212] e *clusters* de estações de trabalho). A especificação define mais de 100 primitivas de comunicação, que incluem diversas funcionalidades para padrões de comunicação em grupos, típicas de aplicações paralelas. Mensagens MPI são transientes, devido à natureza da comunicação em sistemas paralelos. A primeira versão da MPI surgiu em 1995 e, desde então, a especificação passou por três revisões.

Outra modalidade de middleware orientado a mensagens bastante relevante é a dos sistemas conhecidos como *Message-Oriented* Middleware (MOM). Apesar do nome pouco esclarecedor, os MOM representam uma classe de sistemas de middleware projetada para o envio de mensagens assíncronas e confiáveis, sendo capazes de persistir o conteúdo das mensagens e reenviá-las caso o destinatário não possa recebê-las de imediato. Os MOMs são tipicamente organizados como uma coleção de canais que funcionam como filas. Processos que desejem enviar mensagens a outros

processos devem fazê-lo enfileirando mensagens em canais específicos. Os processos que desejem receber mensagens de outros processos devem, por sua vez, se inscrever ou consumir mensagens dos canais de eventos de interesse.

Um dos exemplos mais expressivos de MOM é o IBM *MQSeries* (agora *WebSphere MQ)*. O *MQSeries* foi lançado pela primeira vez em 1992 e é hoje o padrão de fato para passagem de mensagens, sendo compatível com mais de 35 plataformas (segundo o fabricante). Outros exemplos de MOM são o *BEA MessageQ* e o *JBoss Messaging.* É interessante notar que os MOM apresentam um nível de abstração que é tipicamente mais baixo que o dos sistemas de middleware orientados a componentes – MOM entregam bytes. Além disso, é perfeitamente possível encarar os serviços providos por um MOM como apenas mais um serviço de middleware – i.e., não se trata de algo radicalmente distinto do que já discutimos.

1.2.2 Ciclo de desenvolvimento e execuções distribuídas

O segundo fator que contribui com a complexidade dos sistemas distribuídos não é um fator de cunho técnico. Trata-se de uma propriedade intrínseca de qualquer sistema concorrente, relacionada à natureza de suas execuções. Para entender o impacto das execuções distribuídas no desenvolvimento de software, vamos começar com uma pequena discussão a respeito dos processos de desenvolvimento.

Até onde o alcança o nosso conhecimento, todos os processos de desenvolvimento prevêem três atividades fundamentais:

1. Todo sistema de software deve ser **codificado** (ou **especificado** de alguma forma). Sistemas são codificados/especificados para que possam ser postos em uso (executados).
2. Todo sistema de software é, portanto, **executado** em algum momento (caso contrário, o sistema perde seu propósito), sendo que as primeiras execuções devem acontecer após um período de codificação (caso contrário, não há o que executar). Execuções podem ser previstas durante o desenvolvimento (testes de unidade, integração, aceitação) ou não (o sistema é executado diretamente em produção).
3. Se um cenário de execução (ou teste) provoca uma falha no sistema, tem início o processo de **depuração**.

Embora seja possível colocar um sistema em produção sem testá-lo, essa é uma prática bastante atípica. A esmagadora maioria dos processos de desenvolvimento prevê algum tipo de atividade de teste, muito embora o ponto do processo em que os testes são realizados possa variar significativamente. Num processo em cascata tradicional [162], por exemplo, podem haver períodos de desenvolvimento significativamente longos sem que quaisquer testes sejam conduzidos. Em

métodos ágeis como o XP [15], por outro lado, os testes são conduzidos continuamente, do início ao fim do projeto.

Figura 1-3. Atividades primordiais no desenvolvimento de software.

Definição 1-3: A atividade de *teste* compreende a síntese e a análise de uma ou mais execuções de um sistema de software, em busca de violações de propriedades específicas.

As "propriedades específicas" às quais nos referimos na Definição 1-3 dizem respeito a uma coleção de especificações de comportamento correto que podem ser produzidas para um sistema de software.

Definição 1-4: Um sistema de software é considerado *correto* quando adere às suas especificações.

Por razões que dispensam explicação, construir sistemas de software que sejam corretos é algo altamente desejável. Uma das questões mais difíceis, entretanto, diz respeito a como produzir especificações completas e adequadas para sistemas arbitrários [51]. Além disso, o custo de especificações formais completas pode ser muito alto, desbalanceando a relação custo-benefício do sistema [20, 129]. A alternativa são as abordagens experimentais – os populares testes. Infelizmente, no entanto, testar um sistema de forma completa é impraticável [98]. Esse problema de completitude leva, segundo Kaner, a uma abordagem baseada em heurísticas, onde os desenvolvedores projetam um pequeno número de casos de testes capazes de representar, dentro das limitações da mente humana, o conjunto de todos os testes possíveis. Essa abordagem é chamada por Bach de *testagem a contento* [12].

Definição 1-5: *Testagem a contento* é o processo de desenvolvimento de um conjunto de testes que avalia de forma suficiente, e a um custo razoável, a qualidade de um sistema de software. Esse conjunto de testes deve permitir que se tomem decisões razoáveis e rápidas a respeito de um produto.

Testes são uma parte importante do processo de desenvolvimento, porque contribuem com um potencial aumento de *qualidade* no software produzido. Os critérios de qualidade são, tipicamente, correção, confiabilidade e portabilidade [104]. Um problema fundamental com testes, entretanto, é que a natureza experimental da abordagem é incapaz de garantir a ausência de erros [160].

O processo de testes tradicional envolve os passos mostrados na Figura 1-4. O lado esquerdo do fluxograma mostra o ciclo de testes – os testes são executados, um após o outro, variando-se os dados de entrada e configurações. O processo de depuração só tem início caso algum dos casos de teste resulte na violação de um comportamento correto previamente especificado.

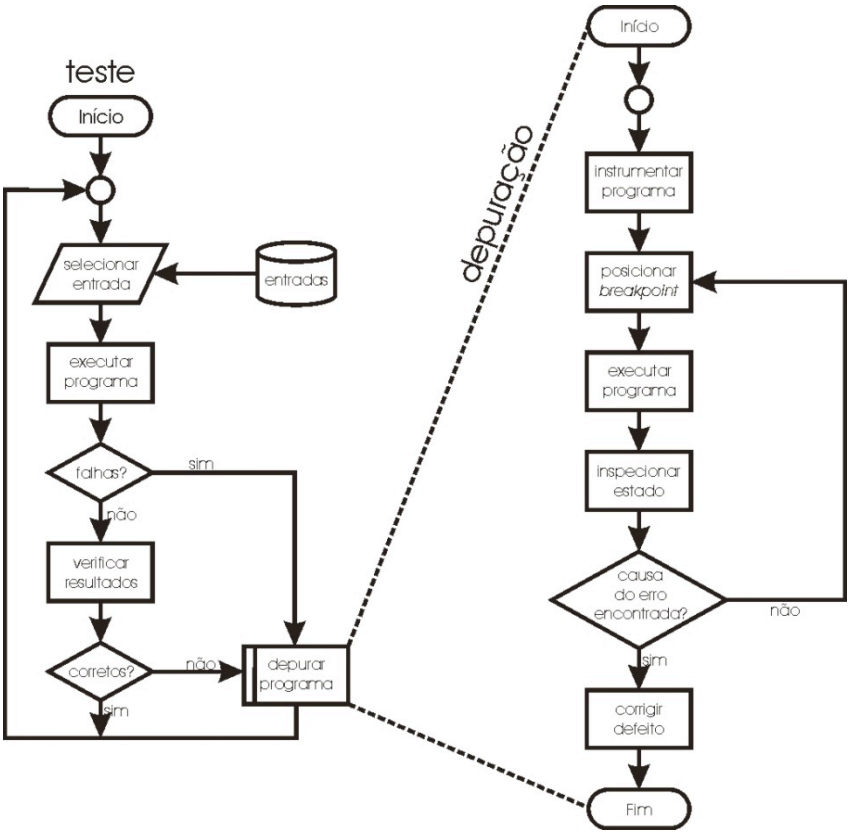

Figura 1-4. Ciclo tradicional de teste e depuração.

Definição 1-6: *Depuração* é o processo através do qual são identificadas e isoladas as causas de comportamentos errôneos num sistema de software.

Supondo que o depurador não seja capaz de encontrar a causa do erro de forma automática (uma hipótese bastante razoável para o caso geral), o ciclo de depuração prossegue com re-execuções sucessivas do cenário de teste. A cada re-execução, o desenvolvedor posiciona um ou mais *breakpoints* (Definição 1-7), procurando interromper a execução nos pontos em que julgar pertinentes, afim de ganhar mais informações a respeito das possíveis causas do erro (fluxograma da direita na Figura 1-4).

Definição 1-7: Um *breakpoint* demarca uma pausa intencional num certo ponto da execução de um programa. O principal propósito de um *breakpoint* é o de viabilizar a inspeção do estado do programa (conteúdo de memória, pilhas de *threads*, registradores, etc.) no ponto da execução demarcado.

Em algum momento, a causa do erro torna-se aparente e o desenvolvedor pode então tomar as providências necessárias, possivelmente alterando trechos de código da aplicação. As re-execuções sucessivas são um reflexo do fato que o processo de depuração consiste, essencialmente, em tentar visualizar a execução de trás para diante.

Definição 1-8: A *depuração cíclica* é um processo no qual o desenvolvedor re-executa repetidas vezes um cenário de teste em que o programa falha, visando ganhar mais informações a respeito da causa do comportamento errôneo.

Vamos agora discutir brevemente as dificuldades introduzidas pelos sistemas paralelos e distribuídos nas atividades de teste e depuração (uma discussão mais detalhada será conduzida no Capítulo 2). O principal problema com as execuções distribuídas é a sua natureza parcialmente ordenada, que abre espaço a uma nova classe de comportamentos não-determinísticos. Fatores como latências imprevisíveis na rede e o acesso concorrente a segmentos de memória compartilhada tornam a reprodução de execuções uma tarefa difícil, reduzindo a eficácia das técnicas empregadas na depuração cíclica. Há ainda uma implicação bastante séria com relação à abordagem tradicional de testes – quando uma aplicação seqüencial passa por um caso de testes sem produzir nenhum erro, é possível ter certeza de que execuções subseqüentes daquele caso de teste (supondo interação determinística com o ambiente externo) não vão resultar em erro. Se uma aplicação distribuída passa por um caso de testes sem produzir nenhum erro, no entanto, isso pode significar apenas que aquela execução foi favorável ao teste. Em outras palavras, os problemas de completitude da

abordagem de testes tradicionais são exacerbados com aplicações distribuídas (e concorrentes em geral), resultando em ainda mais incertezas e dificuldades.

1.3 Sumário

Sistemas paralelos e distribuídos compartilham características, mas, no sentido usual, os termos fazem referência a sistemas computacionais com características bastante diferentes. O foco dos sistemas paralelos é no processamento de problemas computacionalmente difíceis, enquanto que os sistemas distribuídos têm no compartilhamento de recursos a sua *raison d'être*. Os benefícios que podem ser extraídos do compartilhamento de recursos provido por um sistema distribuído são muitos e as motivações que levam indivíduos e organizações a compartilharem recursos são igualmente diversas. Infelizmente, no entanto, sistemas distribuídos são notoriamente difíceis de construir corretamente. Essa dificuldade pode ser atribuída a dois fatores fundamentais: as dificuldades ligadas à programação em ambientes heterogêneos e a complexidade intrínseca das execuções concorrentes (inclusive as distribuídas).

Com relação à heterogeneidade, os sistemas de middleware – sistemas que provêm serviços distribuídos e interoperáveis às aplicações distribuídas que deles fazem uso – constituem um dos principais aliados do desenvolvedor. Os *sistemas de* middleware vêm sendo desenvolvidos há algum tempo, sendo que alguns dos princípios fundamentais empregados nesses sistemas – protocolos de dados e APIs padronizadas – podem ser rastreados até a introdução dos *Berkeley Sockets*, em 1982. Uma parte substancial dos *sistemas de* middleware tiveram como abstração de comunicação primária variantes da chamada de procedimento remoto. Com a introdução do middleware baseado em componentes esse paradigma deixa de ser tão favorecido, embora continue gozando de bastante popularidade. Além disso, nota-se uma tendência de absorção de complexidade por parte dos sistemas de middleware, algo que favorece o reuso a partir da especificação de servidores genéricos que se assemelham, em conceito, a arcabouços.

As dificuldades ligadas às execuções distribuídas transcendem o cunho técnico e acidental (segundo a terminologia de Brooks [22]) da heterogeneidade. Os ciclos tradicionais de teste e depuração são prejudicados pela natureza não-determinística das execuções concorrentes, perdendo sua eficácia ante sistemas cuja lisura não pode ser atestada pela não-exibição de comportamentos errôneos em casos de testes. O próximo capítulo será dedicado à exploração das características das execuções concorrentes que as tornam tão difíceis de lidar.

2 Execuções Concorrentes e Distribuídas

"The first step in fixing a broken program is getting it to fail repeatably."

-- Tom Duff

Este capítulo trata de propriedades das execuções concorrentes que dificultam as atividades de teste e depuração discutidas ao longo da Seção 1.2.2. Os seguintes tópicos serão discutidos:

- **Um modelo para execuções distribuídas**: a Seção 2.1 introduz um modelo bastante tradicional para a representação de execuções distribuídas. Esse modelo é baseado na teoria de sistemas concorrentes de Lamport [109, 110] e nos trabalhos de Mattern [127, 185] e Schwarz [185], e será, por vezes, utilizado no restante do texto**.**
- **Impacto em atividades de teste e depuração:** a introdução do modelo na Seção 2.1 serve como ponto de partida para a apresentação, na Seção 2.2, de problemas relacionados às atividades de teste e depuração que surgem como resultado da estrutura parcialmente ordenada das execuções distribuídas. Essas questões surgem com quaisquer sistemas concorrentes, mas vamos começar a discuti-las no contexto de sistemas baseados em passagem de mensagens (i.e., sistemas distribuídos "puros").
- **Relógios escalares e vetoriais:** a Seção 2.3 discute duas peças fundamentais do arcabouço teórico, desenvolvido ao longo de mais de vinte anos de pesquisa, para a captura de relações causais: os relógios lógicos de Lamport e Fidge-Mattern.
- **Cortes consistentes e observações corretas:** a Seção 2.5 caracteriza observações corretas e descreve um conjunto de características desejáveis para observações produzidas por ferramentas de depuração e monitoramento.
- **Prelúdio para as condições de corrida: sistemas de memória compartilhada:** após a discussão dos problemas ligados à observabilidade, vamos introduzir, na Seção 2.5, os sistemas que utilizam memória compartilhada como mecanismo de comunicação interprocessos. Vamos caracterizar e discutir as relações causais nesses sistemas, apontando as semelhanças existentes entre memória compartilhada e passagem de mensagens.
- **Condições de corrida em sistemas de memória compartilhada e condições de corrida em sistemas baseados em passagem de mensagens:** as Seções 2.6.1 e 2.6.2 discutem um dos problemas mais importantes dos sistemas concorrentes – o não-determinismo de suas execuções. Um ingrediente fundamental nesse não-determinismo são as condições de corrida, que serão

caracterizadas tanto no contexto de sistemas baseados em memória compartilhada quanto no de sistemas baseados em passagem de mensagens.

- **Os efeitos do observador:** a Seção 2.7 termina o capítulo apresentando dois problemas. O primeiro deles – o efeito do observador – aparece como conseqüência direta da natureza parcialmente ordenada das execuções concorrentes, discutida durante o restante do capítulo. O segundo problema – o efeito labirinto – diz respeito a como organizar o imenso volume de dados produzidos por uma execução distribuída.

2.1 Um modelo para execuções distribuídas

Sistemas distribuídos são freqüentemente modelados como um conjunto $D = \{P_1, ..., P_n\}$, $n \in N^*$ e $n \geq 2$, onde $P_i \in D$, $i \in N^*$e i $\leq n$, representa um processo seqüencial [110, 127, 144, 148, 185, 221]. Por "processo seqüencial" entende-se, informalmente, um processo em que as instruções são executadas uma após a outra, da mesma forma em que aparecem no texto do programa. Supõe-se que os processos seqüenciais possuem estados disjuntos – i.e. não há memória compartilhada – e que a comunicação interprocessos é feita exclusivamente através da troca de mensagens. O comportamento de cada processo é completamente determinado por um algoritmo local, que também determina as reações de cada processo às mensagens recebidas.

Definição 2-1: Chamamos de *execução distribuída* a execução concorrente e coordenada dos algoritmos executados pelos processos seqüenciais que compõem o sistema distribuído.

As ações tomadas por cada algoritmo local serão chamadas de *eventos*. De um ponto de vista abstrato, uma execução distribuída pode ser descrita pelos tipos, pelas instâncias e pela ordem relativa dos eventos que ocorrem em cada processo. Seja E_i o conjunto dos eventos que ocorrem no processo P_i. Já que consideramos que P_i é estritamente seqüencial, então os eventos em E_i podem ser totalmente ordenados de acordo com a seqüência de sua ocorrência: $E_i = \{e_{i1}, e_{i2}, ...\}$. Nós vamos nos chamar essa ordenação de acordo com a ocorrência de *enumeração padrão* de E_i. Para caracterizarmos uma execução distribuída, é suficiente, para os nossos propósitos, distinguirmos entre três tipos de eventos:

- **Evento de envio:** ocorre quando um processo P_i envia uma mensagem a outro processo P_j.
- **Evento de recebimento**: ocorre quando um processo P_i recebe uma mensagem de outro processo P_j.

- **Eventos internos:** eventos arbitrários que ocorrem na execução local de um processo P_i. Eventos internos afetam apenas o estado local de P_i.

Vamos agora definir um conjunto de relações que caracteriza a forma pela qual eventos podem afetar uns aos outros. Essa caracterização é especialmente relevante para depuração, dado que o processo de depuração trata precisamente de encontrar a causa de erros.

Definição 2-2 [185]: Dadas as enumerações padrão de E_i e E_j, a relação de *causalidade* $\rightarrow \subseteq E \times E$ é definida como a menor relação transitiva satisfazendo:

1. Se $e_{ik}, e_{in} \in E_i$ e $k < n$, então $e_{ik} \rightarrow e_{in}$;
2. se $s \in E_i$ é um evento de envio e $r \in E_j$ é um evento de recebimento, então $s \rightarrow r$.

Note que a relação de causalidade é não-reflexiva, assimétrica e transitiva; ou seja, trata-se de uma *ordem parcial estrita*. Além disso, $(E, \rightarrow)$ é um *deposet* (um conjunto parcialmente ordenado finito que pode ser decomposto numa coleção de cadeias – ordens totais – disjuntas [216]). A relação de *causalidade* definida aqui é equivalente à relação *acontece antes* definida por Lamport em seu clássico artigo sobre relógios lógicos [110]. O nome "relação de causalidade" nos parece mais apropriado, entretanto, já que *acontece antes* pode levar o leitor a tirar conclusões erradas (i.e., pode ser que um evento *a* aconteça, em relação ao tempo real, antes que um evento *b*, mas que $a \not\rightarrow b$). O que a relação de causalidade caracteriza, intuitivamente, é a influência que um evento pode surtir sobre eventos futuros.

Particularmente, o envio de uma mensagem *m* por um processo P_i para um processo P_j pode afetar todos os eventos em P_j que ocorrem após o evento de recebimento de *m*. A Figura 2-1 mostra um diagrama temporal de uma execução distribuída hipotética, que conta com três processos.

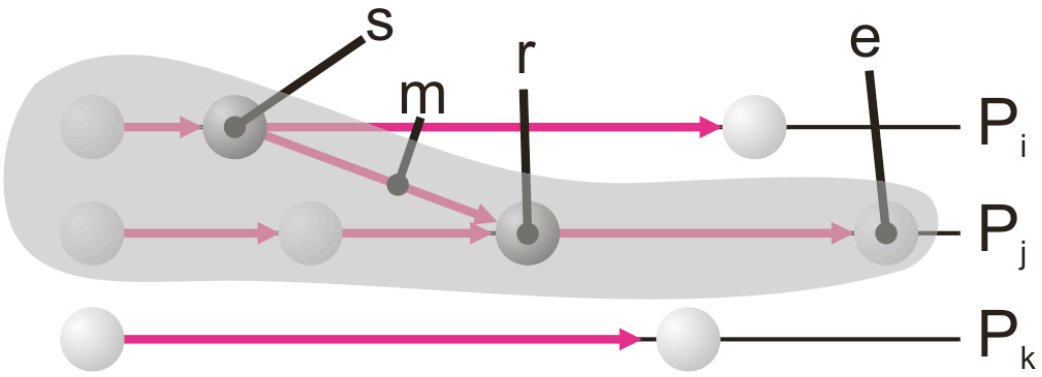

Figura 2-1. Diagrama de tempo mostrando relações causais entre eventos e o histórico causal do evento **e**.

Os eventos *s* e *r* destacados na Figura 2-1 correspondem a um par de eventos de envio e de recebimento, respectivamente. Todos os outros eventos são eventos internos (inclusive o evento **e,** também destacado). A área hachurada corresponde ao *histórico causal* do evento **e**. O histórico causal de **e** é composto por todos os eventos que, direta ou indiretamente, podem afetar **e**. O que é importante notar é que, se **e** é um evento interno que corresponde a uma falha em P_j, então essa falha pode ter sido induzida por eventos que estão no histórico de **e**. Mais formalmente:

Definição 2-3 [185]: Seja $E = E_1 \cup ... \cup E_n$ o conjunto de eventos de uma execução distribuída e seja $e \in E$ um evento qualquer. O *histórico causal* de *e*, denotado $C(e)$, é definido como $C(e) = \{e' \in E : (e' \rightarrow e)\} \cup \{e\}$.

Uma outra noção importante é a que diz respeito a eventos concorrentes. Intuitivamente, eventos concorrentes são eventos que ocorrem "ao mesmo tempo". A idéia de eventos que ocorrem ao mesmo tempo é capturada pelo não-estabelecimento de um vínculo causal direto entre esses eventos; i.e., dois eventos *a* e *b* são concorrentes quando não há restrições causais que os impeçam de executar ao mesmo tempo. Formalmente:

Definição 2-4 [185]: A relação de *concorrência* $\| \subseteq E \times E$ é definida como: $a \| b \Leftrightarrow a \not\rightarrow b \wedge b \not\rightarrow a$.

A relação entre histórico causal e causalidade sai direto da Definição 2-3. Dados $e, e' \in E$, $e \neq e'$:

1. $e \rightarrow e' \Leftrightarrow e \in C(e')$;
2. $e \| e' \Leftrightarrow e \notin C(e') \wedge e' \notin C(e)$.

2.2 Impacto em atividades de teste e depuração

As duas principais implicações da estrutura de ordem parcial das execuções distribuídas sobre as atividades de teste e depuração são descritas a seguir.

(1) Execuções não-reproduzíveis

A natureza parcialmente ordenada das execuções distribuídas pode levar a situações em que o sistema se comporta de forma não-determinística. Considere a execução mostrada na Figura 2-2 (a): o processo P_j recebe duas mensagens dos processos P_i e P_k. Logo após receber a mensagem de

P_k (evento r_{j2}), o processo P_j falha. Entretanto, devido à natureza parcialmente ordenada da execução, e supondo que a execução se desenrola de forma determinística até a linha pontilhada, seria possível que o processo P_k atingisse o ponto de sua execução em que o evento s_{k1} é produzido antes que P_i atingisse o ponto em que s_{i1} é produzido. Isso é possível porque s_{k1} e s_{i1} não são causalmente relacionados; i.e., são concorrentes. Isto pode levar à situação mostrada na Figura 2-2 (b), onde a mensagem enviada por P_k é processada, por P_j, antes da mensagem enviada por P_i. A inversão na ordem dos eventos na execução de P_j altera a sua estrutura, possivelmente alterando completamente a execução de P_j a partir da linha pontilhada (isso é ilustrado pela remoção do evento **e** na Figura 2-2 (b), que tem a sua ocorrência condicionada à estrutura de eventos da Figura 2-2 (a)).

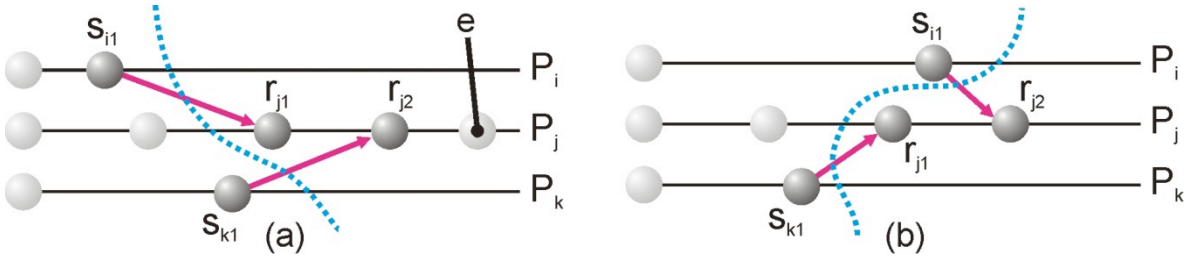

Figura 2-2. Resultados distintos decorrentes de uma condição de corrida.

Os fatores que podem levar um desses processos a executarem mais rápido ou mais devagar (conseqüentemente produzindo diferentes resultados a cada execução) são inúmeros – faltas de página, utilização de CPU, decisões do escalonador ou a execução de um coletor de lixo são apenas alguns possíveis motivos. Além disso, ainda que os processos executem sempre no mesmo ritmo, variações na latência de entrega das mensagens pela rede podem gerar atrasos imprevisíveis que, ao final, levam ao mesmo efeito.

As conseqüências para as atividades de testes em sistemas distribuídos são importantes: é possível que um trecho de código passe num caso de testes por mero acaso e depois falhe em produção, sob condições aparentemente idênticas. O desenvolvedor só pode afirmar, portanto, que o sistema passa de fato num caso de teste após verificar que os resultados produzidos são corretos para todas as possíveis execuções daquele caso de testes. Há, no entanto, três inconvenientes nisso. O primeiro deles é que o número de possíveis ordenações cresce exponencialmente com o número de eventos concorrentes. O segundo, conforme vamos discutir na Seção 2.6, é que determinar todos os eventos potencialmente concorrentes numa dada execução é um problema NP-difícil. O terceiro, de ordem acidental (usando a terminologia de Brooks [22]), é que existem poucas ferramentas que permitem que a ordem dos eventos seja manipulada em caráter experimental.

Na depuração, execuções não-determinísticas podem levar a erros potencialmente difíceis de reproduzir. Essa dificuldade culmina com o aparecimento de um certo tipo de defeito conhecido como *Heisenbug*[1].

Definição 2-5: Um *Heisenbug* é um comportamento errôneo, apresentado por um sistema de software, que desaparece quando sob análise por uma ferramenta de depuração.

Felizmente, para o caso da depuração, é possível reproduzir execuções não-determinísticas de forma determinística, conforme vamos discutir na Seção 3.1. O não-determinismo em execuções distribuídas e concorrentes está intimamente relacionado à ocorrência de *condições de corrida*. Nós vamos discutir condições de corrida em mais detalhe nas Seções 2.5 e 2.6.

(2) Observabilidade

A segunda dificuldade resultante da ordenação parcial das execuções distribuídas diz respeito à observabilidade dos eventos. Fidge [63] divide o problema da observabilidade em quatro questões:

a. **Múltiplos observadores podem perceber ordenações diferentes:** esta questão diz respeito ao fato de que a notificação de um evento pode levar tempos diferentes para propagar-se até observadores diferentes, gerando percepções distintas sobre qual evento acontece antes de qual. Esta questão também se manifesta na física: a localização do observador determina a sua visão do universo.
b. **A ordenação dos eventos pode ser incorreta:** ainda que todos os observadores percebam uma mesma ordenação para os eventos, essa ordenação pode ser incorreta.
c. **Mesma estrutura de eventos, observações diferentes:** tanto a latência de entrega das mensagens quanto flutuações nas execuções locais podem levar até mesmo um sistema cujas execuções são determinísticas a produzir diferentes ordenações de eventos aos olhos de seus observadores. Este problema é uma conseqüência direta dos dois problemas anteriores (este problema é sutilmente distinto do problema **a**).
d. **A ordenação dos eventos é arbitrária:** ainda que os eventos sejam visualizados na ordem correta, pode não ser possível distinguir entre eventos causalmente relacionados e eventos não-causalmente relacionados.

Essencialmente, determinar a ordem e as relações causais entre os eventos numa execução distribuída é uma tarefa não-trivial. Essa questão é bastante relevante para a depuração de sistemas distribuídos, já que o estabelecimento da ordem correta dos eventos é fundamental para que o

[1] *Heisenbugs* recebem seu nome em homenagem ao físico alemão Werner Heisenberg e seu princípio da incerteza, enunciado em 1927. A homenagem não faz muito sentido, no entanto, já que o princípio da incerteza e o efeito do observador – a real causa dos *Heisenbugs* – não são estritamente relacionados.

desenvolvedor (ou algoritmo) seja capaz de determinar quais as relações de causa e efeito entre eventos que levam à manifestação de um erro. Para entendermos melhor as questões apontadas por Fidge, suponhamos que um observador seja notificado da ocorrência de eventos por meio de mensagens que são enviadas por meio de uma rede. Se o observador simplesmente acreditar que a ordem em que as notificações são entregues corresponde à ordem em que os eventos ocorrem, podemos acabar com uma situação semelhante à mostrada na Figura 2-3 (a).

Na Figura 2-3 (a), há dois observadores – O_1 e O_2 – que enxergam ordenações diferentes para os eventos e_1 e e_2. O_1 enxerga a ordem correta, mas O_2, por estar mais distante da fonte do evento e_1, enxerga a ordem inversa. Esse cenário ilustra os quatro problemas descritos por Fidge (com a possível exceção do problema **c,** mas não é difícil ver que esse problema poderia acontecer caso o sistema fosse executado novamente). Se, por outro lado, os processos produzissem *timestamps* para os eventos baseados em seus relógios locais (e os eventos fossem ordenados de acordo com esses *timestamps*), a ordenação dos eventos seria consistente para todos os observadores. Infelizmente, no entanto, a situação mostrada na Figura 2-3 (b) ainda poderia ocorrer – a reconstrução da execução a partir dos *timestamps* (Figura 2-3 (c)) revela uma ordenação incorreta, para todos os observadores (ainda sofremos com os problemas b, c e d).

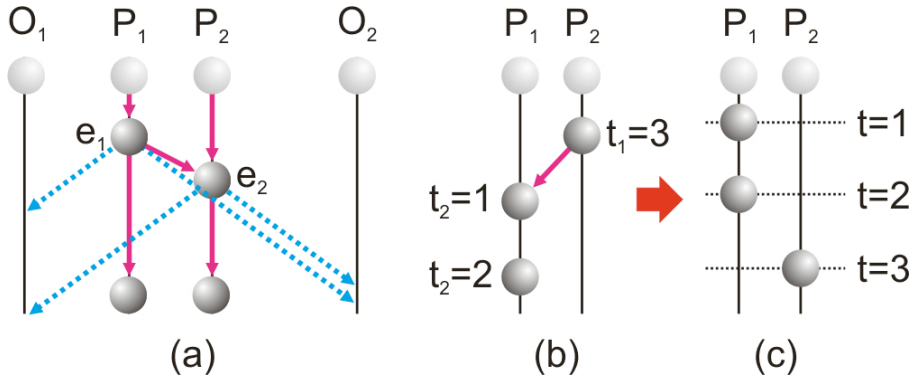

Figura 2-3. Inconsistências na observação de eventos.

A disponibilidade de um relógio global resolveria as questões b e c: se os eventos são ordenados de acordo com *timestamps* gerados por um relógio global, a ordem percebida pelos observadores torna-se consistente e determinística para execuções idênticas. Um fato talvez surpreendente, no entanto, é que a questão d não pode ser resolvida com relógios globais. Para ver o porquê, suponhamos a existência de relógios perfeitamente sincronizados (de um relógio global) no sistema distribuído e tomemos $T_R(x): E \rightarrow \mathbb{R}$ uma função que mapeia os eventos da execução distribuída em instantes do tempo real (representados como números reais). Sejam ainda $e, e' \in E$ dois eventos quaisquer. A partir da atribuição de *timestamps* que correspondem aos instantes no tempo real em que ocorrem e e e', podemos concluir que:

1. Se $e \rightarrow e' \Rightarrow T_R(e) < T_R(e')$;
2. se $T_R(e) < T_R(e') \Rightarrow (e \rightarrow e') \vee (e \| e')$;
3. $T_R(e) = T_R(e') \Leftrightarrow e \| e'$.

Note que o inverso das implicações 1 e 2 não é necessariamente verdadeiro, como mostra a Figura 2-4 ($T_R(e_{11}) < T_R(e_{22})$ e $e_{11} \rightarrow e_{22}$, mas $T_R(e_{12}) < T_R(e_{22})$ e $e_{12} \| e_{22}$). Particularmente, não é possível decidir se dois eventos são concorrentes ou causalmente relacionados se olharmos apenas para os instantes, no tempo real, em que esses eventos ocorrem. Isso é um reflexo direto do fato que o tempo real não captura causalidade [127, 185].

Apesar dessa imprecisão, as informações trazidas pelo tempo real são suficientes para evitar anomalias causais. Infelizmente, a disponibilidade de relógios sincronizados é uma hipótese difícil e cara de atingir na prática, especialmente quando a precisão requerida é muito alta. Felizmente, por outro lado, o uso de relógios lógicos escalares (discutidos na próxima seção) é suficiente para que possamos capturar as informações de ordenação que podem ser transmitidas por T_R.

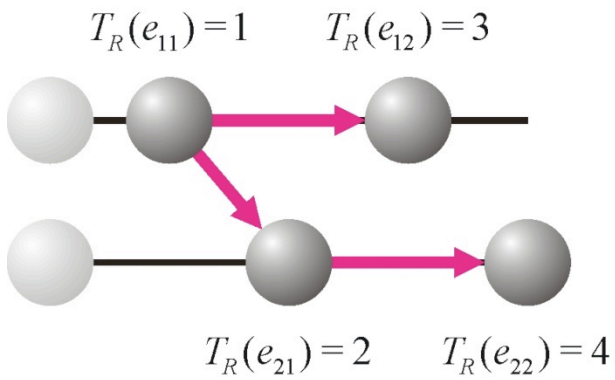

Figura 2-4. Trecho de execução com relações causais e *timestamps* derivadas do tempo real.

2.3 Relógios escalares e vetoriais

Nós mostramos, na Seção 2.2, que a obtenção de históricos causais de eventos não pode, em geral, depender apenas de relógios locais. Além disso, a precisão da sincronização entre relógios num sistema distribuído pode sofrer muito quando a rede apresenta variações elevadas de latência. Isso torna o *timestamping* de eventos que toma como hipótese a sincronização de relógios uma técnica sujeita a produzir observações que violam a causalidade. Para completar o problema, o tempo real não captura completamente a relação de causalidade.

Felizmente, relógios globais ou sincronizados não são tão imprescindíveis para uma grande parte das aplicações. Em 1968, Leslie Lamport publicou um artigo, intitulado *Time, Clocks, and The Ordering of Events in a Distributed System* [110], que descreve um sistema de relógios lógicos

capaz de impor uma ordenação total, consistente com a relação de causalidade, nos eventos produzidos num sistema distribuído. Lamport foi o primeiro a associar o princípio da causalidade, expresso na relatividade especial, à ordenação de eventos em sistemas distribuídos.

Os relógios lógicos propostos por Lamport são bastante simples. Vamos definir um relógio de Lamport como uma função $L : E \rightarrow N$ que mapeia eventos em números naturais (o tempo lógico do evento). Cada processo $P_1,...,P_n$ no sistema distribuído mantém um contador, $A_1,...,A_n$, inicializado com um valor arbitrário (normalmente zero). Esses contadores são atualizados de acordo com as seguintes regras [110]:

1. Cada processo P_i incrementa A_i entre quaisquer dois eventos sucessivos (em P_i);
2. se *a* é um evento de envio de uma mensagem *m* por um processo P_i, então *m* contém um *timestamp* T_m que carrega o valor de A_i no momento em que *a* foi produzido;
3. se *b* é um evento de recebimento de uma mensagem *m* por um processo P_j, P_j incrementa A_j de tal maneira que o novo valor de A_j seja maior que T_m e maior ou igual ao valor anterior de A_j.

Lamport define que $L(a), \forall a \in E_i, 1 \leq i \leq n,$ corresponde ao valor de A_i quando *a* foi produzido. Os eventos são totalmente ordenados de acordo com a seguinte relação [110]:

Definição 2-6: Sejam $e \in E_i$, $e' \in E_j$. A ordem total $\xrightarrow{L} \subseteq E \times E$ é definida da seguinte forma:

1. Se $L(e) < L(e')$ então $e \xrightarrow{L} e'$;
2. se $L(e) = L(e')$ e $i < j$ então $e \xrightarrow{L} e'$.

Não vamos demonstrar, mas não é difícil ver que $\xrightarrow{L}$ é consistente com a relação de causalidade; isto é, $\rightarrow \subseteq \xrightarrow{L}$. Particularmente, sob o ponto de vista da captura causal, L é essencialmente equivalente a T_R[2]. O problema com relógios de Lamport (e com relógios escalares, como o tempo real, de maneira geral) é que eles não caracterizam a relação de causalidade de forma completa. Em diversas situações – algumas das quais serão discutidas nesta dissertação – gostaríamos de poder determinar se dois eventos *a* e *b* são ou não concorrentes; ou, no caso de um depurador, se o histórico causal de um determinado evento que acaba de ser observado está ou não completo.

O conceito de tempo vetorial, que vamos apresentar agora, captura de forma precisa a relação de causalidade. De acordo com Schwarz e Mattern [127, 185], relógios vetoriais foram descobertos

[2] A diferença fica por conta do fato de que não existe a noção de eventos concorrentes em *L*.

e re-descobertos por diversos pesquisadores em instantes distintos. Colin Fidge [64] e o próprio Mattern [127], no entanto, foram os primeiros a analisar as propriedades matemáticas e a apontar as semelhanças existentes entre as estruturas do tempo vetorial e do espaço-tempo de Minkowski para espaços unidimensionais. Por essa razão, relógios vetoriais são muitas vezes chamados de relógios de Fidge-Mattern. Nós vamos chamá-los apenas de relógios vetoriais. A idéia essencial por trás dos relógios vetoriais é que eles são uma forma compacta para a representação de históricos causais (Definição 2-3).

Um histórico causal é representado, no tempo vetorial, por um vetor com *n* posições, onde *n* é o número de processos no sistema distribuído, e cada posição *i* do vetor corresponde ao número de eventos contribuídos por P_i àquele histórico. O histórico causal (e, por conseqüência, o instante de acordo com o tempo vetorial) do evento *e* da Figura 2-1 poderia, portanto, ser representado pelo vetor $\begin{pmatrix}1 & 3 & 0\end{pmatrix}$. Um relógio vetorial é, portanto, uma função $V : E \rightarrow N^n$ que mapeia eventos em vetores de *n* dimensões.

O tempo vetorial num processo é representado como um contador vetorial $V_i[1...n]$, de *n* elementos. Supondo a existência de um evento especial $\perp$ que marca o início de cada processo, o tempo vetorial V_i do processo P_i é atualizado de acordo com as seguintes regras [63, 64, 127, 185]:

1. Inicialmente, $V_i[k] = 0$ para $k = 1,...,n$. ($C(\perp) = \varnothing$, onde $C(\perp)$ representa o histórico causal do evento $\perp$);
2. A cada evento interno e_{ik}, P_i incrementa $V_i[i]$ em uma unidade ($C(e_{ik}) = C(e_{ik-1}) \cup \{e_{ik}\}$).
3. Quando P_i produz um evento de envio e_{ik} de uma mensagem *m*, P_i atualiza $V_i[i]$ como em (2), e anexa a *m* um *timestamp* T_m de valor igual a V_i ($T_m = V_i = C(e_{ik}) = C(e_{ik-1}) \cup \{e_{ik}\}$).
4. Quando um processo P_i recebe uma mensagem $m = (e_{jk}, e_{im})$, P_i incrementa $V_i[i]$ como em (1). Além disso, P_i atualiza seu vetor V_i de tal forma que $V_i[k]_{1 \le k \le n} := \max\{V_i[k], T_m[k]\}$ ($C(e_{im}) = C(e_{im-1}) \cup C(e_{jk}) \cup \{e_{im}\}$).

Note que essas regras de atualização refletem operações de união nos históricos causais, postas entre parênteses após cada regra. O vetor V_i num instante (do tempo real) *t* representa o conjunto de todos os eventos que potencialmente afetam as decisões tomadas por P_i nesse mesmo instante *t*. De maneira equivalente, o valor de $V(e)$, $e \in E_i$, representa o conjunto de todos os eventos produzidos no sistema distribuído que potencialmente afetam as decisões tomadas por P_i após e (inclusive).

A relação de ordem definida pelo tempo vetorial é a seguinte:

Definição 2-7: Sejam $e \in E_i$, $e' \in E_j$. A ordem total $\xrightarrow{V} \subseteq E \times E$ é definida da seguinte forma:

1. Se para $k \in \mathbb{N}$, $1 \leq k \leq n$, $n \in \mathbb{N}$, tivermos $V_i[k] \leq V_j[k]$, então $e \xrightarrow{V} e'$;
2. se existem $k, k' \in \mathbb{N}$ tais que $1 \leq k < k' \leq n$, $V_i[k] \leq V_j[k]$ e $V_j[k'] \leq V_i[k']$, então $e \| e'$ (isto é, $\neg(e \xrightarrow{V} e') \wedge \neg(e' \xrightarrow{V} e)$).

A relação entre causalidade e tempo vetorial é dada pelo seguinte teorema, que não vamos provar neste texto:

Teorema 2-1 [127, 185]: Para quaisquer eventos e, e' numa execução distribuída, temos:

1. $e \rightarrow e' \Leftrightarrow e \xrightarrow{V} e'$;
2. $e \| e' \Leftrightarrow \neg(e \xrightarrow{V} e') \wedge \neg(e \xrightarrow{V} e')$.

A demonstração do teorema é bastante simples e segue da observação de que tempo vetorial e históricos causais são equivalentes. Os leitores interessados podem consultar a demonstração no artigo publicado por Schwartz e Mattern [127, 185].

Apesar das vantagens, a implementação eficiente de relógios vetoriais não é trivial, em especial quando consideramos o tamanho de cada vetor (num sistema com 1000 processadores, por exemplo, cada mensagem carrega um vetor de 1000 inteiros como *overhead*). O uso de técnicas como o método diferencial de Singhal-Kshemkalyani [191], o rastreamento de dependências diretas de Fowler-Zwaenepoel [65] ou o método adaptativo de Jard-Jourdan [94], no entanto, podem amenizar substancialmente esses problemas, à custa de algum processamento extra ou da perda de certas informações em tempo de execução.

Uma pergunta que surge naturalmente nesse contexto é se seria possível capturar a causalidade com *timestamps* menores. Os resultados provados por Charron-Bost [27] sugerem que isso é improvável, embora não demonstrem que seja impossível. Charron-Bost mostra que a captura da relação de causalidade requer vetores de dimensão *n* no pior caso, onde *n* é o número de processos no sistema. O tamanho real de um *timestamp* está relacionado, no entanto, à quantidade de informações que deve obrigatoriamente estar presente em cada "célula", não apenas ao número de dimensões do vetor. O teorema de Charron-Bost não provê um limite inferior para essa "quantidade de informações obrigatória por célula". O problema, portanto, permanece em aberto.

Uma segunda questão, talvez mais grave, ligada aos relógios vetoriais diz respeito à dificuldade na acomodação de sistemas onde o número de processos participantes varia ou não é conhecido a priori (uma situação incomum em boa parte dos sistemas paralelos, mas bastante corriqueira em

sistemas distribuídos e *multithreaded*). Essas dificuldades tornam a aplicação direta de relógios vetoriais ainda mais difícil.

2.4 Cortes consistentes e observações corretas

Um requisito fundamental imposto sobre ferramentas de depuração e monitoramento é que as informações capturadas sejam consistentes. Essas ferramentas operam sobre estados globais, portanto a captura de informações inconsistentes implica na operação incorreta da ferramenta. Por definição, um estado global envolve todos os processos num sistema distribuído, mas os problemas para captura de estados que envolvem um subconjunto dos processos são essencialmente os mesmos. Começamos esta seção definindo o que é um estado global consistente.

Definição 2-7 [127, 185]: Um corte consistente C é um subconjunto finito de E que satisfaz:
Se $e \in C \Rightarrow e' \in C$, para todo $e' \rightarrow e$.

Note que definimos um corte consistente, não um estado global consistente. Na verdade, cortes e estados consistentes são bastante relacionados. A diferença é que "estado global" faz referência ao estado dos processos locais num corte. Note ainda que um corte consistente é um conjunto fechado à esquerda pela relação de causalidade. Segue da Definição 2-3 que todo histórico causal é um corte consistente.

A Figura 2-5 (a) mostra um corte não-consistente. Para entendermos o significado de um corte não-consistente, suponhamos que os processos P_1, P_2 e P_3 fotografem seus estados nos instantes marcados pela linha pontilhada na Figura 2-5 (a) – as fotografias locais correspondem aos eventos f_1, f_2 e f_3. Do ponto de vista da Definição 2-7, o corte formado pelas fotografias não é consistente porque não inclui o histórico causal completo de f_2.

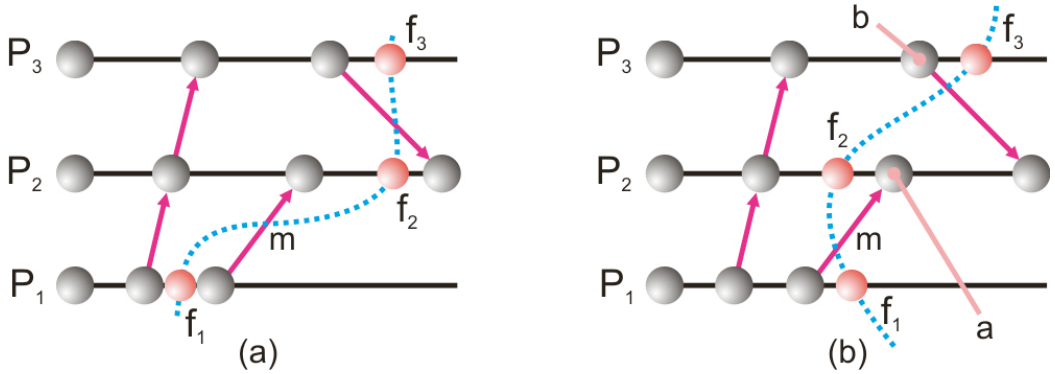

Figura 2-5. (a) Corte não-consistente. (b) Corte consistente.

Intuitivamente, no entanto, o corte não é consistente porque ele inclui uma fotografia do estado de P_1 que foi tirada antes do envio de m e uma fotografia do estado de P_2 que foi tirada após o

recebimento de *m*. O estado global capturado por essas fotografias é *impossível*, porque a transição de P_2 ao estado f_2 está condicionada, pela causalidade, à transição de P_1 a um estado posterior a f_1. Os estados f_1 e f_2 nunca poderiam ocorrer ao mesmo tempo. O corte na Figura 2-5 (b), por outro lado, é consistente, já que inclui o histórico causal completo dos eventos f_1, f_2 e f_3. Note que o evento *a* (não incluso no corte) acontece, na execução real, antes do evento *b* (incluso no corte). Isso não implica, no entanto, que o estado capturado é inconsistente, porque a execução do sistema *admite* esse estado global; isto é, trata-se de um estado global *possível*. Um estado possível é um estado que poderia ter acontecido caso a velocidade de execução dos processos num trecho seqüencial fosse diferente, mas não diferente a ponto de perturbar a ordenação causal dos eventos na execução. Chandi e Lamport demonstram, num outro artigo clássico da literatura de sistemas distribuídos, que os estados locais capturados por um corte consistente de fato refletem um estado global que de fato poderia ter ocorrido [26].

Conforme discutiremos no Capítulo 4, o processo de depuração de um sistema distribuído consiste em navegar por subconjuntos, normalmente próprios, do reticulado de cortes consistentes. Se os cortes não são consistentes, as informações tornam-se sem sentido, destruindo qualquer esperança de uma reconstrução da execução que leve a uma melhor compreensão de um comportamento errôneo. É importante, portanto, que ferramentas de depuração que operam capturando e exibindo cortes o façam de forma a produzirem visões consistentes.

2.5 Prelúdio para as condições de corrida: sistemas de memória compartilhada

A estrutura parcialmente ordenada das execuções distribuídas, discutida na Seção 2.1, também se aplica aos sistemas concorrentes que utilizam memória compartilhada como mecanismo de comunicação interprocessos (ou, de forma abreviada, aos *sistemas de memória compartilhada*). A relevância desse tipo de sistema em nosso estudo é diretamente proporcional à freqüência, bastante alta, do uso do modelo de *multithreading* e memória compartilhada em nodos individuais de sistemas distribuídos típicos. A proliferação de processadores *multicore* deve acentuar ainda mais essa tendência.

Nos moldes do modelo apresentado na Seção 2.1, um sistema de memória compartilhada pode ser descrito como um conjunto $C = \{P_1, ..., P_n\}$ de processos independentes, onde $n \in N$ e $n \geq 2$. Para simplificar o modelo, supomos que o hardware subjacente implementa um modelo de memória capaz de garantir consistência seqüencial (i.e., todas as instruções *load* e *store* aparentam executar numa ordem total, que respeita a ordem local de cada processador [4]).

Novamente, definimos três tipos de eventos:

- **Evento de leitura:** ocorre quando um processo P_i lê dados de uma região de memória compartilhada.
- **Evento de escrita:** ocorre quando um processo P_i escreve numa região de memória compartilhada.
- **Evento interno:** como na passagem de mensagens, tratam-se de eventos arbitrários (seqüências de instruções) que ocorrem na execução local de um processo P_i. Eventos internos afetam apenas o estado local de P_i.

De maneira semelhante ao que fizemos nos sistemas baseados em passagem de mensagens, iremos supor que as operações de escrita e leitura (os eventos) são atômicas. Isso é compatível com a adoção de um modelo de memória compartilhada, em que as operações de escrita sobre palavras individuais são atômicas (o hardware garante exclusão mútua em alguma granulosidade). Assim como Netzer [145, 147, 149], vamos definir uma relação auxiliar entre eventos – a relação de ordem temporal $\xrightarrow{T}$ – que descreve a ordem temporal (real) em que os eventos acontecem.

Podemos transportar a relação de causalidade para os sistemas de memória compartilhada da forma descrita na Definição 2-6:

Definição 2-6: Dados $P_i, P_j \in D$ e suas respectivas enumerações padrão E_i e E_j, a relação de causalidade $\xrightarrow{S} \subseteq E \times E$ em sistemas de memória compartilhada é definida como a menor relação transitiva onde:

1. Se e_{im}, e_{ik} e $m < k$, então $e_{im} \xrightarrow{S} e_{ik}$;
2. se $a \in E_i$ e $b \in E_j$ são eventos que atuam sobre uma mesma área de memória compartilhada, a é um evento de escrita e b é um evento de leitura:
   a. se $a \xrightarrow{T} b$ então a $a \xrightarrow{S} b$;
   b. se $\neg(b \xrightarrow{T} a)$ e $\neg(a \xrightarrow{T} b)$, então $a \xrightarrow{S} b \wedge b \xrightarrow{S} a$.

Note que, quando o hardware subjacente garante consistência seqüencial, quaisquer pares de eventos que ocorram ao mesmo tempo podem se afetar mutuamente (regra 2.b). Note ainda que é possível que $i = j$ (como também é possível, ainda que pouco usual, que $i = j$ no caso dos sistemas baseados em passagem de mensagens) e que essa relação é diferente da relação de dependência de dados definida por Netzer [145]. As Figura 2-6 (a) e (b) ilustram a relação de causalidade. A condição (1) é necessária para capturar, ainda que de forma grosseira, as dependências de existência

condicional que podem existir entre eventos (exemplo na Figura 2-7, Seção 2.6). A condição (2) captura a noção intuitiva de que as decisões tomadas por um processo dependem do seu estado local e compartilhado.

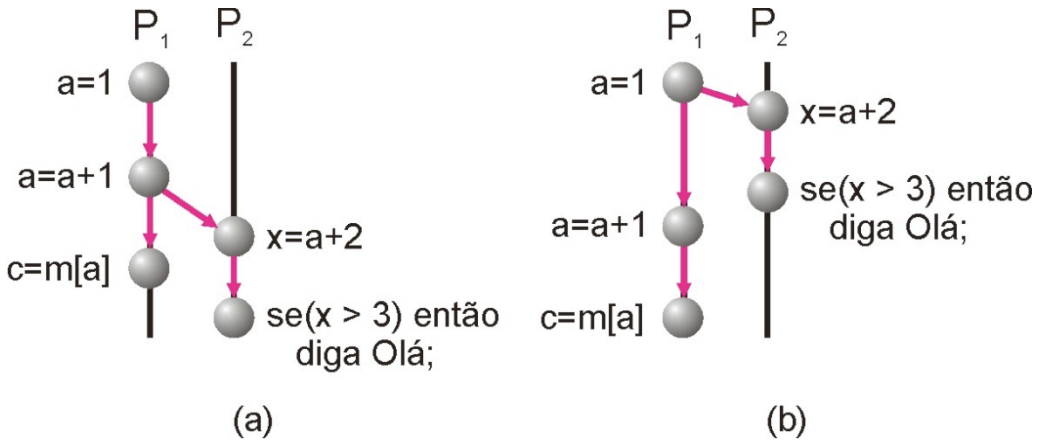

Figura 2-6. Duas possíveis execuções num sistema de memória compartilhada. As flechas explicitam a redução transitiva da relação de causalidade.

Os problemas relativos ao não-determinismo das execuções continuam os mesmos – as relações de precedência causal entre eventos de leitura e escrita podem variar de execução para execução, fazendo com que os resultados variem de acordo. Isso é ilustrado na Figura 2-6, com a formação de dependências causais e resultados distintos em (a) e (b).

2.6 Condições de corrida – o cerne do indeterminismo em sistemas concorrentes

Conforme apontamos nas Seções 2.2 e 2.3, as condições de corrida são um ingrediente importante no não-determinismo das execuções concorrentes. Na verdade, as condições de corrida caracterizam precisamente os pontos da execução de um sistema concorrente que são sensíveis a variações temporais. É claro que existem outras fontes de indeterminismo, mas estas estão tipicamente relacionadas, de forma direta ou indireta, à interação com dispositivos de entrada e saída ou à ocorrência de interrupções. Alguns exemplos de interações que podem tornar as execuções de um sistema, seja ele seqüencial ou concorrente, não-determinísticas, são:

- notificações de falha de entrega de mensagens produzidas por dispositivos de rede;
- geradores de números pseudo-aleatórios semeados com valores tomados, por exemplo, de um relógio local;
- dados lidos de um relógio local;
- dados enviados por dispositivos de interface humano-computador, tais como teclado, *mouse*, *pen tablets*, *touch screens* ou microfones;

- sinais assíncronos produzidos por outros programas;
- etc.

Dionne [47] dá o nome de *instruções fortemente indeterminísticas* às interações, de caráter inerentemente não-determinístico, com dispositivos de entrada e saída. Condições de corrida são consideradas, por outro lado, *fracamente indeterminísticas*. Nesta seção, nós vamos supor que os dispositivos de entrada e saída comportam-se de forma determinística; isto é, para uma dada ordem de acesso com um mesmo conjunto de parâmetros, os resultados devolvidos serão sempre os mesmos. Usando a terminologia de Dionne, iremos supor que as instruções fortemente indeterminísticas são, para os propósitos da nossa discussão, determinísticas. Além disso, vamos considerar que interrupções assíncronas sempre ocorrem de forma determinística (i.e., sempre ocorrem quando o sistema atinge um ponto exato de seu estado). Essa hipótese não é realista, mas nos permite discutir os efeitos das condições de corrida separadamente.

Nosso estudo de condições de corrida é baseado nos trabalhos de Netzer [145, 147, 149], Ronsse [173] e Mellor-Crummey [84, 111, 131]. A discussão será centrada nos trabalhos de Netzer e Ronsse, por considerarmos que esses trabalhos são mais relevantes e atuais. A caracterização das condições de corrida será feita com base no trabalho de Netzer, que é o único autor a fazer um tratamento completo e formal.

O termo *condição de corrida* diz respeito, no sentido usual, a falhas decorrentes do acesso simultâneo e não-sincronizado a regiões de memória compartilhadas. Intuitivamente, uma condição de corrida pode levar um sistema a comportar-se de forma não-determinística porque a ordem de acesso aos trechos de memória compartilhada não protegidos por primitivas de sincronização depende da velocidade relativa de execução dos processos concorrentes. Essa ordem de acesso, por sua vez, pode não ser determinística, conforme discutimos na Seção 2.2.

O significado preciso do termo *condição de corrida* não é, no entanto, consistente na literatura. Netzer [149] distingue entre *condições de corrida gerais* e *condições de corrida de dados*. *Condições de corrida gerais* são aquelas responsáveis por comportamentos não-determinísticos em geral, sendo danosas apenas para programas cuja execução deveria ser determinística, mas não para outros tipos de programas. As *condições de corrida de dados*, por sua vez, resultam da execução não-atômica de seções críticas – i.e., trata-se do sentido usual de condição de corrida – e são sempre danosas (para Netzer). Toda condição de corrida de dados é uma condição de corrida geral, mas o contrário nem sempre é válido.

Segundo Netzer, é perfeitamente possível que um sistema concorrente contenha condições de corrida gerais sem que isso implique, necessariamente, que o sistema contém erros de programação. Para entender o porquê, basta notar que a ordem de entrada dos processos ou *threads* que competem

por uma região crítica protegida por *mutexes*, por exemplo, depende da "velocidade" relativa dos processos – i.e., o processo "mais rápido" trava o *mutex* primeiro, ganhando acesso antes à região crítica.

Ronsse [173], por outro lado, distingue entre *condições de corrida de sincronização* e *condições de corrida de dados*. Condições de corrida de sincronização são as condições de corrida decorrentes da competição entre processos ou *threads* por *mutexes,* semáforos, monitores etc. Condições de corrida de dados, por outro lado, decorrem do acesso não-sincronizado a trechos de memória compartilhada. Ronsse nota que a distinção entre corrida de sincronização e corrida de dados é meramente uma questão de abstração, já que muitas primitivas de sincronização fazem uso de condições de corrida de dados reais em sua implementação. As caracterizações de Netzer e Ronsse são bastante semelhantes, a diferença fica por conta do fato que Netzer se importa com sistemas cuja execução deve ser determinística e explora, em seu trabalho, o problema da detecção de condições de corrida gerais [145]. Ronsse, por outro lado, explora apenas a detecção de condições de corridas de dados, por não considerar que condições de corrida gerais possam ser consideradas erros [173].

Esta discussão mostra que o termo *condição de corrida* diz respeito, na realidade, a situações na execução de um sistema concorrente em que a sincronização não induz uma ordenação causal única – i.e., diz respeito às condições de corrida gerais de Netzer. Nas seções que seguem, iremos apresentar as caracterizações de condições de corrida de Netzer para sistemas de memória compartilhada, passando para as condições de corrida em sistemas baseados em passagem de mensagens logo em seguida.

2.6.1 Condições de corrida em sistemas de memória compartilhada

Um ponto crucial da caracterização de condição de corrida proposta por Netzer é a relação de dependência de dados.

Definição 2-7: Sejam $a \in E_i$ e $b \in E_j$. Dizemos que existe uma dependência direta de dados entre os eventos *a* e *b,* ou que $a \xrightarrow{DD} b$, se e somente:

1. *a* e *b* representam acessos a uma mesma região de memória;
2. ao menos um desses acessos é um acesso de escrita.

A relação de dependência de dados $\xrightarrow{D} \subseteq E \times E$ é dada pelo fecho transitivo e não-reflexivo de $\xrightarrow{DD}$. Além disso, a relação de dependência de dados e a relação de precedência temporal

$\xrightarrow{T}$ (Seção 2.5) têm um relacionamento bastante estreito, dado pelos seguintes axiomas [145], listados aqui por completeza:

(A1) $\xrightarrow{T}$ é uma ordem parcial não-reflexiva;

(A2) se $a \xrightarrow{T} b \overset{T}{\not\leftrightarrow} c \xrightarrow{T} d$, então $a \xrightarrow{T} d$;

(A3) se $a \xrightarrow{D} b$, então $\neg(b \xrightarrow{T} a)$.

Note que a relação de dependência de dados captura aspectos importantes da execução que não são revelados pela relação de causalidade. O fato que $a \xrightarrow{DD} b$ se *a* **ou** *b* são eventos de escrita faz com que a relação capture dependências causais que poderiam surgir em ordenações alternativas. Para ver o porquê, note que se *a* é um evento de leitura e *b* um evento de escrita, então não há uma dependência causal entre *a* e *b*. Mas se a ordem fosse diferente (porque P_i executou mais devagar), então <u>haveria</u> uma dependência causal entre *b* e *a*. Essa caracterização de formações causais alternativas é o cerne da caracterização de condições de corrida gerais.

Sob o modelo de Netzer, uma execução pode ser representada como uma tripla $P = \left\langle E, \xrightarrow{T}, \xrightarrow{D} \right\rangle$. A caracterização de condições de corrida depende de, mas não consiste unicamente em, determinar se existem execuções alternativas em que um dado par de eventos $a, b \in P$ poderia ocorrer em uma ordem diferente, ou ao mesmo tempo. Para evitar ter que determinar se existem execuções arbitrárias em que *a* e *b* acontecem, Netzer concentra-se nas execuções que são prefixos de *P*. Uma execução $P' = \left\langle E', \xrightarrow{T'}, \xrightarrow{D'} \right\rangle$ é um prefixo de *P* se e somente se, para cada processo P_i, a enumeração padrão E'_i de P'_i é um prefixo da enumeração padrão E_i. A partir daí, podemos analisar se as restrições de sincronização em *P* permitem que eventos sejam reordenados, originando relações causais diferentes das de *P*.

São definidos três conjuntos de prefixos que caracterizam possíveis execuções. O primeiro deles, F_{SAME}, representa todas as execuções que contêm os mesmos eventos e as mesmas dependências de dados que *P* – isto é, tratam-se de todas as execuções equivalentes a *P*, mas cuja ordem temporal relativa de eventos pode variar, desde que isso não perturbe as dependências de dados. A definição formal é dada a seguir.

Definição 2-8 [145]: F_{SAME} é o conjunto de todas as execuções $P' = \left\langle E', \xrightarrow{T'}, \xrightarrow{D'} \right\rangle$ tais que:

1. P' é uma execução viável,
2. $E' = E$, e
3. $\xrightarrow{D'} = \xrightarrow{D}$.

F_{SAME} inclui execuções P' em que dois eventos que poderiam ocorrer ao mesmo tempo em *P* de fato ocorrem, desde que isso não altere a dependência de dados no fim. Não estão inclusas, no entanto, as execuções em que os eventos ocorrem em uma ordem *definitivamente* distinta (veja o axioma A3). Execuções com ordenações de eventos distintas de *P* são capturadas pelo segundo conjunto, F_{DIFF}, que admite prefixos com dependências de dados arbitrárias, desde que as execuções resultantes sejam prefixos viáveis de *P*. Prefixos viáveis são aqueles que respeitam as dependências entre controle e dados impostas pelo programa: a existência de um evento, por exemplo, pode estar condicionada, pela lógica do programa, a uma certa dependência de dados entre eventos anteriores. Neste caso, a inversão dessa dependência pode tornar o prefixo inviável (passa a representar uma execução impossível). Um prefixo inviável da execução na Figura 2-7 (a) é ilustrado na Figura 2-7 (b).

Definição 2-9 [145]: F_{DIFF} é o conjunto de todas as execuções $P' = \left\langle E', \xrightarrow{T'}, \xrightarrow{D'} \right\rangle$ tais que:

1. P' é uma execução viável;
2. P' é prefixo de *P,* e;
3. $\xrightarrow{D'}$ é arbitrária.

Note que, dado que a execução é um prefixo de *P*, as dependências de dados "arbitrárias" estão restritas aos casos em que a ordenação temporal varia o suficiente para inverter o seu sentido.

O terceiro conjunto, F_{SYNC}, contém todas as execuções que são prefixos de *P* e que respeitam a semântica de sincronização do programa. Por "respeitam a semântica sincronização do programa" entende-se, por exemplo, que se existe uma região crítica propriamente protegida por semáforos, então não podemos produzir prefixos em que dois eventos ocorrem simultaneamente dentro dela.

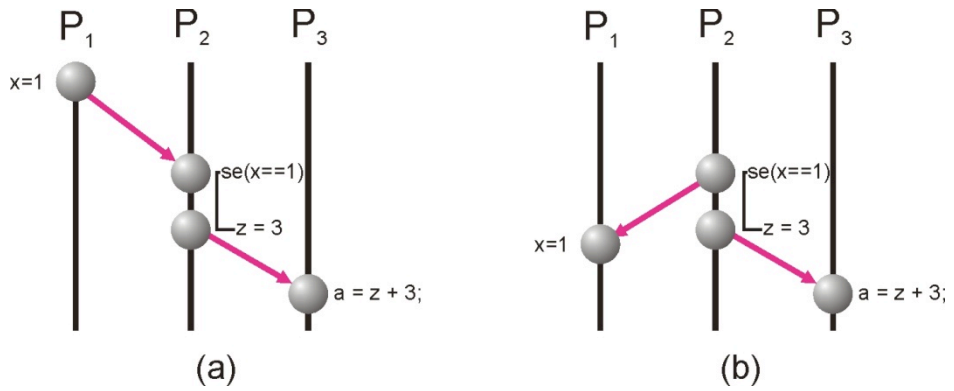

Figura 2-7. Uma execução concorrente (a) e um prefixo inviável (b).

A restrição é mais fraca do que em F_{DIFF}, já que respeitar apenas a semântica de sincronização permite o surgimento de execuções não-viáveis, como a que aparece na Figura 2-7 (b).

Definição 2-10 [145]: F_{SYNC} é o conjunto de todas as execuções $P' = \left\langle E', \xrightarrow{T'}, \xrightarrow{D'} \right\rangle$ tais que:

1. P' respeita a semântica de sincronização de *P*;
2. P' é prefixo de *P*, e;
3. $\xrightarrow{D'}$ é arbitrária.

F_{SYNC} faz muito pouco sentido quando considerado fora do contexto dos algoritmos usualmente empregados na detecção de condições de corrida, que rastreiam apenas a ordem das operações de sincronização do programa (e não a ordem de todos os acessos à memória). Nós vamos discutir esses algoritmos na Seção 3.2.3.

Apresentados os conjuntos de prefixos, a definição de condição de corrida de dados de Netzer é dada a seguir. Para as definições 2-11 e 2-12, seja *F* um dos conjuntos de prefixos descritos anteriormente, e sejam *a* e $b \in E$ eventos em $P = \left\langle E, \xrightarrow{T}, \xrightarrow{D} \right\rangle$.

Definição 2-11 (Condição de corrida de Netzer [145]): Dizemos que uma *condição de corrida de dados* ocorre sobre *F* se:

1. Existe um conflito de dados entre *a* e *b*;
2. existe uma execução $P' = \left\langle E', \xrightarrow{T'}, \xrightarrow{D'} \right\rangle \in F$ onde

   $\neg(a \xrightarrow{T'} b) \wedge \neg(b \xrightarrow{T'} a)$.

Um "conflito de dados" significa que *a* e *b* acessam uma mesma região compartilhada de memória e que pelo menos um desses eventos é um evento de escrita. A Definição 2-11 captura a idéia de que uma condição de corrida de dados só ocorre quando as restrições de sincronização não impedem que *a* e *b* ocorram ao mesmo tempo. Embora isso contrarie a intuição de que uma

condição de corrida existe quando as restrições de sincronização permitem que existam execuções P, P' onde $a \xrightarrow{D} b$ em P e $b \xrightarrow{D'} a$ em P', nota-se que a segunda situação pode ocorrer mesmo na ausência de falhas de sincronização. Trata-se de uma restrição mais fraca que, não coincidentemente, caracteriza as condições de corrida gerais.

Definição 2-12 [145]: Uma *condição de corrida geral* existe sobre *F* quando:

1. Há um conflito de dados entre *a* e *b*;
2. existe uma execução $P' = \left\langle E', \xrightarrow{T'}, \xrightarrow{D'} \right\rangle \in F$ tal que:
   a. $b \xrightarrow{T'} a$ se $a \xrightarrow{T} b$, ou;
   b. $a \xrightarrow{T'} b$ se $b \xrightarrow{T} a$, ou;
   c. $b \| a$.

A definição de Netzer é dada em função de relações temporais, mas a implicação é a mesma – as dependências de dados acontecem (ou podem acontecer, no caso da condição (c)) em sentidos distintos para execuções distintas. Dadas as Definições 2-11 e 2-12, temos:

1. **Condição de corrida de dados *real:*** são as condições de corrida que ocorrem sobre $\{P\}$. Representam as <u>condições de corrida que acontecem de fato</u>.
2. **Condição de corrida de dados *possível*:** são condições de corrida que ocorrem sobre F_{SAME} ou sobre F_{DIFF}. Representam as <u>condições de corrida que poderiam ter acontecido</u>.
3. **Condição de corrida de dados *aparente*:** são condições de corrida que ocorrem sobre F_{SYNC}. Representam uma aproximação conservadora (com inúmeros falsos positivos) das <u>condições de corrida possíveis</u>.

Netzer prova que o problema de determinar todas as condições de corrida gerais e de dados admitidas por uma execução é sempre NP-difícil (isto é, determinar todas as condições de corrida gerais, ou de dados, sobre F_{SAME}, F_{DIFF} ou F_{SYNC}, são problemas NP-difíceis). Netzer também prova que determinar se uma execução *P* está livre de condições de corrida reais é um problema tratável (de fato, é desse problema que trata o detector de condições de corrida de Ronsse [173]).

Esses resultados implicam que o melhor que podemos fazer, quando o assunto são *threads* e memória compartilhada, é detectar se uma execução admite condições de corrida *reais*. Isso mostra que a detecção de condições de corrida para sistemas que utilizam memória compartilhada para comunicação inter-processos (ou inter-*threads*) é inerentemente difícil e que as abordagens práticas ao problema serão sempre incompletas.

2.6.2 Condições de corrida em sistemas baseados em passagem de mensagens

A caracterização de condições de corrida para sistemas baseados em passagem de mensagens é semelhante à apresentada para sistemas de memória compartilhada. A idéia intuitiva é que duas mensagens m_1 e m_2 serão concorrentes sempre que houver um processo que possa recebê-las em ordens distintas para execuções distintas (Figura 2-8).

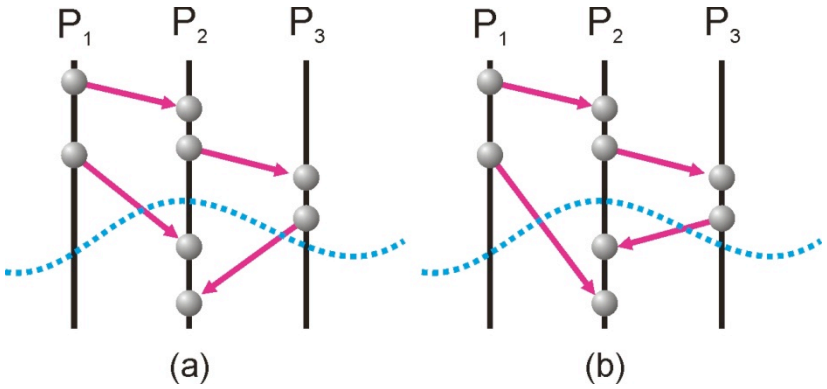

Figura 2-8. Mensagens concorrentes e ordenações possíveis.

As implicações das condições de corrida em sistemas baseados em passagem de mensagens são semelhantes às discutidas para sistemas de memória compartilhada. Na presença de condições de corrida, as execuções tornam-se não-determinísticas e sensíveis a perturbações na velocidade das execuções relativas dos processos. Além disso, o número de possíveis execuções cresce exponencialmente com o número de mensagens concorrentes, dificultando abordagens de testes exaustivas.

Apesar das semelhanças, existem, segundo Netzer e Damodaran-Kamal [146], diferenças suficientes entre os dois problemas para que os resultados anteriormente estudados por Netzer (e parcialmente apresentados na Seção 2.6.1) não se apliquem. Um indício disso é que a análise de comportamentos alternativos para sistemas baseados em passagem de mensagens só faz sentido para prefixos de execuções que não destruam a causalidade entre eventos de envio e recebimento; isto é, só fazem sentido num eventual equivalente de F_{SAME}. Na Figura 2-9, mostramos (a) uma execução de um sistema de memória compartilhada e (b) um prefixo em $F_{DIFF} - F_{SAME}$ dessa execução. Na Figura 2-9 (d), tentamos produzir um prefixo semelhante para uma execução de um sistema baseado em passagem de mensagens (Figura 2-9 (c)), mas a execução resultante não faz sentido.

Note que o prefixo (d) não é simplesmente como os prefixos em F_{SYNC}, discutidos na Seção 2.6.1 (prefixos que, embora pudessem representar execuções não-viáveis, representam uma

aproximação que torna a detecção de corridas mais prática). Trata-se de um prefixo para o qual não há um significado claro, onde um evento de recebimento ocorre antes do evento envio correspondente. Além disso, não há um equivalente de F_{SYNC} nos sistemas de passagem de mensagens, já que não existe estado compartilhado explícito e, por conseqüência, não existem equivalentes das operações de sincronização em sistemas de memória compartilhada (ao menos não sem considerarmos protocolos que vão além da troca de mensagens). Essa intuição é confirmada pela caracterização de Netzer para o conjunto de prefixos sobre o qual são caracterizadas condições de corrida entre mensagens.

Seja $P = \langle E, \rightarrow \rangle$ uma execução distribuída. Definimos $\xrightarrow{m_P} \subseteq E \times E$ como a relação que se estabelece entre eventos de envio e recebimento de mensagens; i.e., se *a* é um evento de envio e *b* o evento de recebimento correspondente, então $a \xrightarrow{m_P} b$.

Definição 2-13 [148]: F^m_{SAME} é o conjunto de todas as execuções $P' = \langle E', \rightarrow' \rangle$ onde:

1. P' representa uma execução viável;
2. P' é um prefixo de *P*;
3. P' contém a mesma estrutura de entrega de mensagens que *P* até um certo corte consistente (isto é, $\xrightarrow{m_P} = \xrightarrow{m_{P'}}$ até um certo corte consistente).

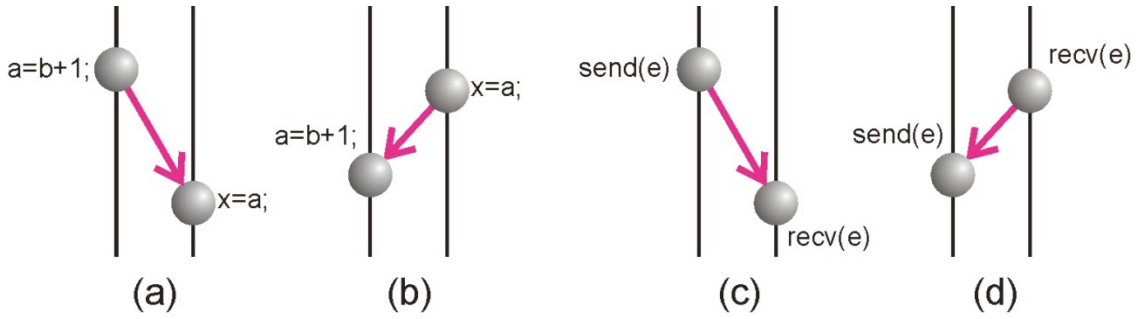

Figura 2-9. Prefixos em F_{DIFF} para sistemas que utilizam passagem de mensagens não fazem sentido.

A Definição 2-13 captura de forma clara o conjunto de todas as possíveis variações de *P* – não são necessários conjuntos adicionais. Particularmente, se *P* for determinística, todas as execuções em F^m_{SAME} serão equivalentes a *P*. Segue a definição de condição de corrida:

Definição 2-14 [148]: Uma mensagem $a \xrightarrow{m_P} b$ *concorre com* uma mensagem $c \xrightarrow{m_P} d$ se e somente se $\exists P' = \langle E', \rightarrow' \rangle \in F^m_{SAME}$, onde $a, b, c \in E'$ e $c \xrightarrow{m_{P'}} b$ $(c \neq a)$.

Curiosamente, ao contrário dos sistemas de memória compartilhada, existem algoritmos eficientes capazes de detectar todas as mensagens concorrentes numa execução distribuída, em tempo polinomial [45, 146, 210]. Isso sugere que o uso de *threads* e memória compartilhada seja inerentemente mais complexo que a passagem de mensagens, corroborando com a opinião de críticos desses modelos, como os proponentes da linguagem Erlang [11].

2.7 Os efeitos do observador e labirinto

Os dois últimos problemas a serem apresentados neste capítulo são conhecidos, na literatura, como o *efeito do observador* [46] e o *efeito labirinto* [68]. O efeito do observador diz respeito à interferência que o código de monitoramento introduzido por uma ferramenta de observação pode produzir na execução do sistema. Uma porção considerável deste capítulo foi dedicada a explicar que execuções concorrentes podem ser sensíveis a variações na velocidade de execução relativa dos processos. A introdução de código de monitoramento pode alterar a velocidade de execução dos processos de maneira não-uniforme, produzindo execuções aberrativas, que teriam baixa probabilidade de ocorrer caso o código de monitoramento não estivesse presente. As perturbações introduzidas pelo código de monitoramento culminam com o aparecimento dos *Heisenbugs* (Seção 2.2).

O efeito labirinto, por outro lado, está relacionado ao volume de dados produzidos durante uma execução distribuída. Sistemas distribuídos podem ser compostos por centenas de unidades de processamento, cada qual contribuindo com a sua fatia de informações de execução. No pior caso, a análise dos dados produzidos pela execução de um sistema distribuído pode requerer um sistema de igual porte, ou maior [104]. O efeito labirinto se manifesta quando a técnica usada para apresentação dos dados de execução não escala de acordo com o volume de informações produzidas (situação preponderante no mundo das ferramentas de depuração, diga-se de passagem). Isso pode ocorrer tanto porque o volume de dados é muito grande quanto porque há excesso de informações irrelevantes sendo exibidas. O resultado é que o usuário perde-se em meio a um labirinto de informações, reduzindo drasticamente as chances de que ele localize a causa de um comportamento errôneo.

2.8 Sumário

Este capítulo tratou das principais questões ligadas à natureza das execuções distribuídas que surgem no contexto das atividades de teste e depuração tradicionais. Essas questões podem ser divididas entre problemas ligados à observabilidade e problemas ligados ao não-determinismo das execuções distribuídas. A observabilidade diz respeito à captura de execuções de forma consistente,

onde “ser consistente” significa respeitar a relação de causalidade definida na Seção 2.1. Um dos maiores empecilhos à observação de históricos de eventos consistentes é a ausência de relógios globais. Essa ausência pode ser compensada, em grande parte das aplicações, pelo uso de relógios lógicos. Em particular, os relógios de Lamport representam uma aproximação suficiente do tempo real em muitas situações.

Tanto o tempo real quanto os relógios escalares de Lamport não são capazes de capturar, de forma completa, as relações causais entre eventos num sistema distribuído. Os relógios vetoriais, por outro lado, derivam suas propriedades de captura da causalidade da álgebra dos históricos causais. Relógios vetoriais, no entanto, são difíceis de implementar de forma eficiente, algo que limita a sua aplicabilidade prática.

Um conceito importante no universo do monitoramento de sistemas concorrentes é o do *corte consistente*. Cortes consistentes representam estados possíveis na execução de um sistema distribuído. Todos os processos de observação de sistemas distribuídos, inclusive os que resultam de atividades de depuração, atuam sobre cortes consistentes. Depurar um sistema distribuído se resume, na verdade, a navegar pelo reticulado de cortes consistentes.

A estrutura das execuções de sistemas concorrentes baseados em memória compartilhada se assemelha àquela exibida pelos sistemas distribuídos baseados em passagem de mensagens. Esses sistemas são importantes no âmbito de nosso estudo devido à freqüência com a qual os nodos de um sistema distribuído utilizam *multithreading*. A caracterização completa de uma execução dos nossos sistemas distribuídos típicos deve, portanto, levar essa característica em consideração.

O indeterminismo inato demonstrado por execuções concorrentes advém do casamento de dois fatores – execuções parcialmente ordenadas e a presença de condições de corrida. Condições de corrida são pontos numa execução parcialmente ordenada em que a ordem da interação entre dois ou mais processos depende da velocidade relativa de execução desses processos. Em sistemas de memória compartilhada, essa interação é mapeada em acessos a regiões de memória compartilhada; em sistemas de passagem de mensagens, a interação se reflete na troca de mensagens.

O estudo das condições de corrida revela que a sua detecção exata é NP-difícil para sistemas de memória compartilhada, mas polinomial para sistemas baseados em passagem de mensagens. Isso leva à conclusão natural de que os sistemas baseados em passagem de mensagens apresentam uma dinâmica de execução mais simples, desse ponto de vista.

O efeito do observador é um fruto direto do indeterminismo das execuções concorrentes. O código de instrumentação inserido para observação de estados pode atrasar a execução de um ou mais processos, fazendo com que a execução resultante seja aberrativa. O efeito do observador culmina com o aparecimento de *Heisenbugs*.

O efeito labirinto, por outro lado, pode ser encarado como um problema ligado à observabilidade de execuções concorrentes e diz respeito às situações em que o usuário é soterrado por informações coletadas durante a execução. O excesso de informações leva o usuário a "perder-se em meio a um labirinto", reduzindo a sua eficácia na localização das causas de um comportamento errôneo. Podemos encarar o efeito labirinto como uma dificuldade intrínseca, já que o volume de dados produzidos pela execução de um sistema distribuído pode sempre ultrapassar o limite de escala de uma técnica de visualização – basta que o sistema seja grande o suficiente.

3 Depuração de sistemas distribuídos

"As far as we know, our computer has never had an undetected error."

-- *Weisert*

Este capítulo traça um panorama da área de interesse deste trabalho, descrevendo e analisando os objetos de nosso estudo – as técnicas e ferramentas de depuração distribuída. Os seguintes assuntos serão abordados neste capítulo:

- **Sistemas de depuração e depuradores para sistemas distribuídos:** a Seção 3.1 introduz sistemas de depuração de forma abrangente, apresenta algumas das principais forças que moldam os sistemas de depuração distribuída e discute seus aspectos arquiteturais mais importantes.
- **Trabalhos relacionados:** a Seção 3.2 cobre as técnicas e ferramentas de mais destaque na literatura, apresentando a sua relação com as questões levantadas no Capítulo 2 (efeito do observador, efeito labirinto, execuções não determinísticas e problemas de observabilidade) e a forma pela qual cada técnica se propõe a resolvê-las. Os tópicos cobertos na Seção 3.2 são os seguintes:
    - **Execução reversa** e **Re-execução determinística (replay *debugging)*:** as ferramentas apresentadas nas Seções 3.2.1 - 3.2.5 têm como foco tornar possível a inspeção de estados cronologicamente arbitrários em execuções de sistemas seqüenciais e concorrentes. Há essencialmente duas classes de abordagens a serem diferenciadas neste caso – as de execução reversa (tratadas Seção 3.2.1) e as de re-execução determinística (ou *execution replay*, tratadas nas Seções 3.2.2 – 3.2.5).
    - **Mais sobre monitoramento e análise:** findada a discussão a respeito de técnicas que lidam com tornar execuções inspecionáveis, partimos para um detalhamento das técnicas de análise e monitoramento. Algumas das técnicas de monitoramento discutidas nesta seção foram implicitamente apresentadas em sessões anteriores, mas é nesta seção é que essas informações serão consolidadas.
    - **Análise e visualização**: a discussão a respeito de técnicas de análise termina com uma breve introdução à área de visualização de software, bem como com uma pequena discussão de medidas adotadas por algumas ferramentas para que as execuções distribuídas tornem-se mais compreensíveis.

- **Heterogeneidade – a grande vilã:** a grande quantidade de ferramentas discutidas na Seção 3.2 motiva uma análise que coloca a heterogeneidade como principal empecilho à chegada das técnicas de depuração avançadas ao dia-a-dia. Nós vamos tentar evidenciar nesta seção que os sistemas de objetos distribuídos estão particularmente mal servidos neste quesito.

3.1 Sistemas de depuração e depuradores para sistemas distribuídos

O termo "depurador para sistemas distribuídos" (ou simplesmente depurador distribuído, como vamos chamá-los doravante) é bastante amplo, podendo ser aplicado a qualquer sistema que monitora e analisa uma execução distribuída em busca de comportamentos errôneos e suas causas. Rosenberg identifica quatro questões-chave que se aplicam ao desenvolvimento de uma ferramenta de depuração arbitrária [174]:

1. **Intrusividade reduzida:** o depurador deve ser intrusivo ao mínimo;
2. **Fidedignidade:** um depurador deve, a qualquer preço, prover apenas informações fidedignas, de modo que o programador possa sempre confiar nelas;
3. **Informações de contexto do programa:** uma ferramenta de depuração deve prover informações de contexto suficientes para que a causa de um comportamento errôneo possa ser identificada;
4. **O desenvolvimento dos depuradores sempre sucede o desenvolvimento de tecnologias:** é de praxe que novas tecnologias sejam postas no mercado sem provisões adequadas para que os sistemas construídos em cima delas sejam depuráveis. Com algumas tecnologias – particularmente os sistemas de middleware orientados a objeto – o mal parece ser crônico. Rosenberg sugere que os desenvolvedores de ferramentas de depuração pressionem os fabricantes dessas tecnologias para que incluam a infra-estrutura necessária à implantação das técnicas de depuração mais sofisticadas.

Além das questões gerais de Rosenberg, depuradores distribuídos têm que lidar, conforme apontamos no Capítulo 2, com um conjunto de problemas que surgem exclusivamente no contexto de sistemas concorrentes:

1. Maior complexidade, maior volume de dados;
2. Problemas de observabilidade;
3. Erros de comunicação e sincronização;
4. Efeitos anômalos adicionais, dentre os quais:
    a. *deadlocks* e *livelocks* [82];
    b. efeitos *stampede* e *bystander* [193];

c. *heisenbugs*;

5. heterogeneidade.

A escolha por como (e se) abordar cada uma das questões apresentadas tem reflexos por vezes profundos na filosofia e no projeto de uma ferramenta de depuração. Não conhecemos nenhuma ferramenta capaz de abordar, de forma satisfatória e simultânea, todas essas questões. Mais freqüentemente, pesquisadores estabelecem um compromisso entre a eficiência da abordagem com certas questões em detrimento de outras. Além disso, o projetista da ferramenta é muitas vezes forçado a adotar soluções sub-ótimas, em decorrência de restrições impostas pela própria plataforma sobre a qual são construídas as aplicações que se deseja depurar. Isso é uma conseqüência direta da quarta questão de Rosenberg. Vamos agora discutir alguns princípios e aspectos arquiteturais de ferramentas de análise e depuração distribuída.

Depuradores atuam, antes de mais nada, como observadores de execuções. Num sistema paralelo ou distribuído, a relação entre observador e sistema observado (ou *sistema-alvo*) é, naturalmente, de um (observador) para muitos (processos locais). Isso induz uma arquitetura naturalmente centralizada, que pode ser observada em diversas ferramentas na literatura [16, 28, 59, 89, 104, 118, 119, 182], onde o observador ocupa a posição central. Na Figura 3-1 (a) temos uma situação hipotética em que um observador é capaz de enxergar uma execução distribuída apenas olhando para o sistema. Na prática, essa situação é impossível – são necessários meios para a extração das informações relevantes da execução de cada processo local, bem como meios para o transporte e apresentação dessas informações ao observador. Os dispositivos de hardware e software adotados para a extração dessas informações de execução no nível dos processos locais são chamados, na literatura, de *monitores*. Monitores serão discutidos em mais detalhe na Seção 3.2.6.

Consideramos que essa nomenclatura não é totalmente adequada no caso de depuradores, no entanto, já que o termo *monitor* vem associado ao ato de monitorar, observar. A questão é que em muitas ferramentas de depuração, esses "monitores" extrapolam a função de observação, produzindo efeitos colaterais intencionais nas execuções em nome de mecanismos interativos. Adotaremos uma outra denominação, portanto, para esses agentes do depurador distribuído que atuam no nível dos processos.

Definição 3-1: Chamamos de *agente local* o conjunto de todo o maquinário implantado, em cada nodo do sistema, em nome do depurador distribuído.

Os agentes locais são mostrados na Figura 3-1 (c), logo acima dos processos componentes. Um processo componente é um processo que pertence à aplicação distribuída, i.e., um processo que compõe o sistema-alvo. Conforme apontamos no Capítulo 2, no entanto, coletar os dados das

execuções de cada processo local, ou processo componente, é apenas parte do problema. Para que as observações produzidas sejam consistentes, as informações coletadas precisam ser posteriormente correlacionadas de forma apropriada. É aí que entram os analisadores de eventos, situados acima dos agentes locais na Figura 3-1 (c). Segundo Hondroudakis [83], essa modalidade de *análise de dados*, o correlacionamento dos eventos observados, é tipicamente conduzida antes das outras, justamente por ser tão fundamental.

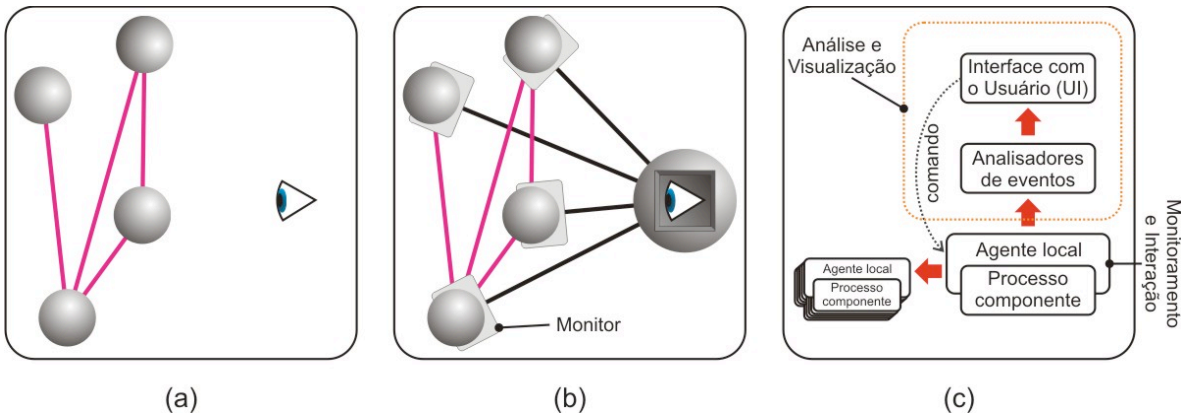

Figura 3-1. (a) e (b) arquitetura centralizada induzida pela presença de um observador. (c) Elementos arquiteturais de um depurador distribuído.

Naturalmente, o correlacionamento de eventos não é o único tipo de análise que pode ser conduzida por um depurador distribuído. Outros tipos de análise bastante relevantes incluem:

1. **Análises de desempenho** [97, 104]: análises do histórico de eventos que têm como produto quantificações de desempenho variadas.
2. **Detecção de predicados globais** [137, 186, 216, 221]**:** análise, direta ou indireta, do histórico de eventos de uma execução, em busca de cortes consistentes que satisfaçam determinadas propriedades.
3. **Busca de padrões** [104, 105]: análise do histórico de eventos em busca da presença ou ausência de padrões de comunicação.
4. **Mapeamento em estruturas de eventos abstratas:** mapeamento do histórico de eventos em estruturas abstratas, muitas vezes representações gráficas, visando melhorar a compreensão de um comportamento por meio de visualizações alternativas ou abstrações de granulosidade mais grossa.

Uma investigação detalhada da anatomia de ferramentas de análise arbitrárias é dada por Klar [101] e Mohr [138], que identificam seis camadas. Uma divisão semelhante é apresentada na análise de Hondroudakis [83]. Esse perfil arquitetural é interessante na medida em que muitos depuradores,

por serem também ferramentas de análise, aproximam-se bastante dele. As camadas são as seguintes:

- Camadas de monitoramento: o monitor
- Camadas de análise:
    - Acesso a dados
    - Seleção
    - Apoio a ferramentas
    - Ferramentas
- Camadas de aplicação: apoio a aplicações.

A camada de monitoramento corresponde às entidades (monitores de hardware ou software) que coletam informações de execução de cada processo. Monitores atuam bastante próximos aos processos do sistema e normalmente se interpõem entre a execução dos processos locais e o hardware ou se apresentam como código de instrumentação, entrelaçados ao próprio código da aplicação. Informações coletadas por monitores podem incluir o estado de variáveis ou registradores, o estado da tabela de processos, ou a ocorrência de determinados eventos, por exemplo.

Acima dos monitores encontram-se as camadas de análise. As camadas de análise processam, correlacionam, filtram e atribuem semântica às informações coletadas pelos monitores. A primeira camada de análise – a de *acesso a dados* – simplesmente provê às camadas superiores um conjunto de serviços de acesso aos dados produzidos pelos monitores. A *camada de seleção* é responsável pela filtragem, agrupamento e composição de eventos, bem como por prover funcionalidades que tornem possível a seleção de grupos de entidades para observação. Na ferramenta *Prism* [6], por exemplo, os dados de desempenho de um processador só são exibidos se esse processador pertencer a um grupo previamente definido. A pertinência ao grupo pode depender do estado do processador (pode ser dinâmica), bem como de outros critérios definidos pelo usuário. O conjunto de operações necessárias para a implementação dos grupos dinâmicos, no entanto, claramente envolve outras camadas. É do nosso entendimento que as funcionalidades providas pela camada de seleção são mais primitivas (incluem, por exemplo, apenas uma API para a ativação e desativação de monitores e, possivelmente, é capaz de listar elementos observáveis no sistema).

A *camada de apoio a ferramentas* expõe a seus clientes uma interface de acesso de alto nível (baseada em eventos) às informações previamente processadas pelas camadas inferiores. A *camada de ferramentas* é quem processa e atribui semântica aos eventos. Atividades típicas da camada de ferramentas incluem validação de traços, produção de modelos visuais da execução, verificação de predicados, produção de dados de desempenho, depuração e distribuição de carga. Por fim, a

camada de aplicação, sob essa classificação, é composta essencialmente pela interface com o usuário. Trata-se da camada responsável por prover ao usuário o acesso às funcionalidades providas pela camada de ferramentas [104].

Conforme mencionamos anteriormente, essa divisão em camadas, embora concebida tomando como modelo ferramentas de análise, expressa de forma bastante precisa a organização de grande parte dos depuradores para sistemas paralelos e distribuídos existentes. Afinal de contas, depuradores paralelos e distribuídos são, até certo ponto, apenas casos especiais de ferramentas de monitoramento e análise. Dizemos "até certo ponto" porque muitos depuradores permitem que o usuário interaja, em maior ou menor grau, com o sistema em execução – embora isso não seja obrigatório. Isso leva a um particionamento natural (e um tanto óbvio) das ferramentas de depuração entre *interativas* e *não-interativas*.

Definição 3-2: Uma ferramenta de depuração é *interativa* quando interfere, além de efeitos colaterais incidentais, com a execução do sistema-alvo.

O exemplo canônico de ferramenta de depuração interativa é o do depurador simbólico. O exemplo canônico de depurador simbólico, por sua vez, é o GDB [197]. De maneira semelhante, uma ferramenta de depuração é *não-interativa* quando não interfere, além de efeitos colaterais incidentais, com a execução do sistema alvo (i.e., trata-se de um observador passivo). Ferramentas de depuração baseadas puramente em técnicas de monitoramento são tipicamente *não-interativas*.

Nossa definição de ferramenta interativa pode parecer um tanto estranha por não levar em consideração um participante geralmente associado a processos interativos – o usuário. O que queremos colocar como ferramenta interativa, no entanto, é algo mais amplo: são aquelas ferramentas que produzem, deliberadamente, efeitos colaterais na execução do programa-alvo. Por "deliberadamente" queremos dizer que essas ferramentas produzem efeitos colaterais que vão além de efeitos acidentais como, por exemplo, o efeito do observador. Particularmente, nossa definição engloba o sentido usual de ferramenta interativa – i.e., ferramentas que produzem efeitos colaterais na execução do sistema-alvo por requisição do usuário – mas também engloba ferramentas que interagem com o sistema-alvo sem a interferência do usuário (como as ferramentas de re-execução, que serão discutidas na Seção 3.2.2).

Nossa síntese dos elementos arquiteturais fundamentais de uma ferramenta de depuração distribuída são mostrados na Figura 3-1 (c). Uma *técnica de depuração* é definida pelo conjunto de suas características de interação, monitoramento e análise. Uma *ferramenta de depuração,* por outro lado, é definida pelo conjunto das características das técnicas que aplica. Uma caracterização formal

completa (ou mesmo informal e detalhada) de técnicas e ferramentas de depuração está além do escopo deste trabalho.

3.2 Trabalhos relacionados

Apresentada a anatomia geral de um sistema de depuração distribuída, partimos aqui para uma discussão das principais ferramentas e técnicas de depuração para sistemas concorrentes, distribuídos e não-determinísticos de maneira geral. Esta seção aborda três tópicos – execução reversa, re-execução e monitoramento e análise. As técnicas de re-execução serão discutidas em mais detalhe do que o restante dos assuntos, por representarem um dos tópicos mais ativos e, ao mesmo tempo, um dos mais antigos na área de depuração de sistemas concorrentes.

3.2.1 Execução reversa

Conforme mencionamos na Seção 1.2.2, a depuração cíclica surge como conseqüência do fato de que o processo de depuração consiste, na realidade, em tentar visualizar a execução de um programa em reverso. O problema com a depuração cíclica, enquanto técnica de visualização reversa, é que a abordagem é naturalmente um processo de tentativa e erro. O programador não tem como saber de antemão em qual ponto da execução encontra-se a causa do erro manifestado. Os *breakpoints*, que definem o ponto da execução em que o programa deve ser suspenso para inspeção, são, portanto, posicionados pelo desenvolvedor baseados num palpite. Se o palpite estiver errado, o desenvolvedor deve elaborar uma nova hipótese, posicionar um novo *breakpoint* e re-executar o programa, na esperança de que o novo ponto da execução escolhido revele mais informações.

O problema é que o processo de re-montagem do cenário de teste que causa o erro pode ser bastante tedioso e lento. Além disso, o programa pode levar um tempo considerável para atingir o ponto da execução de interesse. Como resultado, o processo de reversão da execução torna-se descontínuo, trazendo prejuízos à compreensão do comportamento do sistema. A falta de expressividade por parte dos *breakpoints* implementados em boa parte dos depuradores atualmente em uso é ainda um terceiro empecilho. Imagine como definir um ponto numa execução distribuída – i.e., um corte consistente – usando o mecanismo orientado a linha de código presente na maior parte dos depuradores simbólicos mais difundidos.

A abordagem ideal de depuração, portanto, seria uma que evitasse as ineficiências da re-execução cíclica, permitindo que o usuário explorasse a execução do programa em reverso, passo a passo, desde o estado de manifestação de um comportamento errôneo até o estado revelador de sua causa [31]. O problema é que execuções – distribuídas ou não – não são naturalmente reversíveis. O efeito provocado pela execução de uma instrução destrói, na maior parte dos casos, as informações

que seriam necessárias para restaurarmos o estado anterior do programa. Ainda assim, essa abordagem ideal foi perseguida por alguns pesquisadores, resultando na construção de alguns protótipos [14, 31, 62, 115], que serão discutidos nesta seção.

A ferramenta mais antiga, a EXDAMS [14], foi proposta por Robert M. Balzer em 1969. Trata-se de uma ferramenta que captura todas as transições de estado que ocorrem na execução de um sistema seqüencial e as armazena numa fita de histórico. Tanto as mudanças de valores quanto as decisões de controle são armazenadas, sendo que a captura dessas informações é feita mediante a instrumentação do código-fonte. A fita de histórico pode então ser utilizada para re-apresentar a execução do programa, tanto em reverso quanto no sentido original.

O EXDAMS não re-executa de fato o sistema – a ferramenta simplesmente utiliza a fita de histórico e uma tabela de símbolos para recriar uma simulação, fiel, da execução original. O EXDAMS introduz ainda o conceito de análise de fluxo reversa (*flowback analysis*). A análise de fluxo reversa consiste na construção de uma árvore enraizada [41] que captura as instruções na execução do programa que levam uma variável a assumir o valor representado na raiz.

A segunda ferramenta que gostaríamos de discutir, o *Omniscient Debugger* (ODB [115]), foi apresentada por Bil Lewis em 2003. O ODB é baseado em muitas das mesmas idéias que o EXDAMS – gravar, num dispositivo de armazenagem persistente, todos os eventos que provocam transições de estado – mas a ferramenta apresenta uma interface mais moderna (com a tecnologia de *displays* de 1969, as possibilidades eram muito limitadas), que se assemelha à interface de um depurador simbólico. A diferença, com relação a depuradores simbólicos tradicionais, é que o ODB disponibiliza controles temporais, que permitem que o usuário avance e retroceda na execução conforme as suas necessidades.

O artigo que apresenta o ODB [115] não menciona *flowback analysis*, mas menciona a integração com "analisadores de eventos" – i.e., algoritmos de análise que extraem informações do histórico de eventos. O exemplo apresentado no artigo é simples (um analisador que permite a busca de certos padrões no arquivo de traços), mas analisadores mais complexos poderiam ser implementados.

Do ponto de vista da implementação, tanto o EXDAMS quanto o ODB compartilham a característica de "simularem" a execução reversa – a execução não é reproduzida de forma reversa de fato. Além disso, ambas implementações contam com limitações substanciais no que diz respeito à duração das execuções que podem ser guardadas. Com o EXDAMS, a duração da execução está condicionada ao tamanho da fita de histórico. Com o ODB, a duração da execução está limitada ao tamanho do contador de eventos – um inteiro Java, de 32 bits. Com a ampla disponibilidade e o

rápido barateamento dos dispositivos de armazenagem, entretanto, o ODB se mostra, surpreendentemente, uma solução prática e viável na detecção de diversos tipos de erros.

Infelizmente, para erros mais complexos, informações importantes poderiam se perder com um espaço de armazenagem de eventos tão limitado (o autor comenta que 20 segundos de execução são tipicamente suficientes para que o espaço de endereçamento de eventos se esgote). Além disso, contrariamente ao EXDAMS, o ODB rastreia aplicações Java, que podem ser paralelas ou pseudo-paralelas. Nós suspeitamos que as perturbações introduzidas pelo ODB em execuções concorrentes não sejam desprezíveis, mas o artigo não aborda essa questão.

Diferentemente da EXDAMS e do ODB, que guardam o histórico completo de todos os estados de um programa, a IGOR [62] armazena trechos mais esparsos por meio de um mecanismo de *checkpointing* incremental (um mecanismo que tira fotografias do conteúdo da memória). O mecanismo atua de forma periódica produzindo cópias, em disco, das páginas de memória que foram alteradas pelo programa entre fotografias (*checkpoints*) sucessivas. As informações guardadas pela IGOR não são suficientes, no entanto, para que o programa possa ser executado em reverso.

Ao invés disso, a ferramenta permite que pontos arbitrários da execução sejam inspecionados por meio da re-execução do programa, partindo do estado registrado no *checkpoint* mais próximo. A seleção de pontos na execução é feita por meio de uma linguagem para análise reversa. É possível, por exemplo, buscar pelo primeiro ponto na execução em que uma determinada variável assume um valor superior a um certo valor de referência. Essas facilidades são, no entanto, muito mais limitadas do que as providas pela EXDAMS, já que *checkpoints* capturam apenas predicados estáveis [26] (i.e., propriedades que, uma vez verdadeiras, permanecem verdadeiras). O lado positivo vem sob a forma de um uso mais racional do espaço de armazenagem e de um menor *overhead*[3]. A IGOR sofre com problemas de reprodutibilidade, no entanto, já que o simulador não é capaz de recriar interações com o ambiente externo.

A última ferramenta que gostaríamos de discutir, o *Parallel Program Debugger* (PPD) [31, 135], foi proposta por Miller e Choi em 1988 e estende a análise de fluxo reversa de Balzer a um sistema paralelo de memória compartilhada. Sistemas paralelos, no entanto, geram muito mais informações do que sistemas seqüenciais, sendo ainda suscetíveis ao efeito do observador. Isso torna a aplicação imediata da técnica de Balzer impraticável. Choi procura manter o custo adicional de captura de informações e espaço baixos por meio de uma técnica de rastreamento incremental

[3] Na verdade não podemos afirmar, categoricamente, nada a respeito do *overhead* comparativo das duas abordagens, já que isso só seria possível por meio de um experimento. Intuitivamente, no entanto, acreditamos que seja muito mais barato copiar páginas de memória de forma esporádica do que rastrear todas as operações de atribuição e todos os desvios condicionais e chamadas de métodos numa execução.

(*incremental tracing*). Durante a compilação, o texto do programa, escrito em C, é dividido em uma coleção de blocos e as potenciais dependências de dados e controle entre esses blocos, analisadas estaticamente. Durante a execução do programa, o PPD captura informações dinâmicas pouco detalhadas (de alta granulosidade), que são gravadas num *log*. Essas informações são suficientes para caracterizarem as dependências reais entre blocos, mas não para a análise de fluxo reversa. As informações detalhadas requeridas são produzidas mediante a re-execução de blocos individuais durante a fase de depuração. A re-execução individual de blocos (fora do contexto do programa original) é possível graças às informações capturadas no *log*. Uma boa parte do texto cobre a forma pela qual são identificados os blocos a serem re-executados.

O PPD foi implementado para uma máquina paralela de memória compartilhada e rastreia a ordem das operações de sincronização que envolvem semáforos no programa a fim de determinar a ordem total dos acessos aos trechos de memória compartilhada. Evidentemente, quando uma região de memória é acessada sem proteção, há uma condição de corrida; isto é, não é possível inferir a ordem de acesso a partir da ordem das operações de sincronização. Essas condições de corrida são detectadas pelos meios habituais (discutidos na Seção 3.2.3) e devem ser corrigidas para que o mecanismo de análise de fluxo inversa opere corretamente.

As ferramentas apresentadas nesta seção procuram otimizar o processo de depuração por meio da navegação em reverso da execução. Infelizmente, no entanto, os desafios técnicos são muitos. O ODB e a EXDAMS sofrem com questões ligadas ao uso de espaço e *overhead*, que exacerbam o efeito do observador. A IGOR e o PPD tratam algumas dessas questões adotando abordagens híbridas (parte da execução é rastreada, parte é re-executada) mas a IGOR enfrenta problemas de reprodutibilidade e o PPD não é capaz de lidar com vetores e ponteiros de maneira eficiente [31].

3.2.2 Re-execução determinística (*replay debugging*)

A re-execução determinística surge como alternativa natural às abordagens descritas na Seção 3.2.1. A idéia é que o uso de espaço requerido para a reprodução da execução de um programa é, potencialmente, menor do que aquele requerido para a armazenagem de todas as transições de estado produzidas por uma execução desse mesmo programa[4]. Há ainda uma idéia de que os mecanismos de captura de informações das ferramentas de re-execução, por gerarem um menor volume de informações, podem produzir menos perturbações nas execuções capturadas [127, 185]. De forma semelhante ao que foi implicitamente apresentado com os sistemas de execução reversa, os sistemas de re-execução operam em duas fases, descritas na Definição 3-7.

[4] Podemos traçar um paralelo com funções. Dada uma função *f*, o espaço de armazenagem requerido por um algoritmo que computa *f* (se houver um) é em geral menor que o espaço requerido por uma tabela que contém todos os possíveis valores de *f*.

Definição 3-7: Um *sistema de re-execução* é uma ferramenta de depuração que opera em duas fases, uma de *gravação* e outra de *reprodução*. Na fase de *gravação*, o programa-alvo é executado e informações (traços) de execução são coletadas e armazenadas. Na *reprodução*, os traços coletados durante a fase de *gravação* são utilizados pela ferramenta na reprodução de uma execução para o programa-alvo que é equivalente, mas não idêntica, à execução original.

Conforme discutimos no Capítulo 2, há uma série de problemas ligados à reprodutibilidade de execuções.

- **Interação com dispositivos de entrada/saída:** A influência não-determinística exercida pelo ambiente externo no funcionamento de um sistema (leitura de relógios externos, falhas de entrega de mensagens, efeitos colaterais externos, etc.) pode ser difícil, senão impossível, de reproduzir.
- **Condições de corrida:** Condições de corrida tornam a execução sensível a fatores internos e externos que causem variações na execução relativa dos processos.
- **Interrupções:** Um processo pode ser notificado de um evento externo por meio de uma interrupção. Uma interrupção normalmente altera o conteúdo do ponteiro de instruções (*Program Counter*), transferindo o controle do programa para um trecho de código que trata a interrupção. Além disso, interrupções podem alterar o conteúdo do registrador de *status* e de outros registradores. O problema é que interrupções ocorrem de forma assíncrona – i.e., o momento em que uma interrupção acontece não é, normalmente, determinado pela aplicação.

Os sistemas de re-execução tratam de reproduzir precisamente essas fontes de indeterminismo, constituindo talvez um dos tópicos mais recorrentemente abordados na literatura que trata de técnicas para a depuração de sistemas concorrentes. Uma análise da diversidade de sistemas de re-execução existentes não é uma tarefa direta, já que, como em muitas outras ocasiões na Ciência da Computação, não há uma terminologia ou caracterização suficientemente madura. As duas taxonomias para sistemas de re-execução mais citadas na literatura foram as propostas por Cornelis [42] e por Dionne [47].

A taxonomia de Cornelis classifica os sistemas de re-execução de acordo com a habilidade de cada ferramenta em lidar com os três problemas apresentados anteriormente: re-execução de interações com dispositivos de entrada/saída, re-execução de condições de corrida e re-execução de interrupções. Além desses critérios, Cornelis classifica as ferramentas com relação a uma série de questões técnicas, como abordagem de instrumentação (em tempo de compilação, vinculação, carga ou execução), independência de hardware/linguagem e instrumentação do sistema operacional/ambiente de execução. A classificação de Dionne, apresentada a seguir, é mais

abrangente e, embora não tenha tanta utilidade prática quanto a de Cornelis, dá uma boa idéia de quais os caminhos possíveis para uma técnica de re-execução:

- **Tipo de instrução re-executada:** Dionne diferencia, num sistema de software, entre instruções (a) *determinísticas,* que sempre produzem o mesmo efeito, (b) *fracamente não-determinísticas,* que produzem o mesmo efeito desde que executadas na mesma ordem e (c) *fortemente não-determinísticas*, que nunca produzem o mesmo efeito. Sistemas de re-execução podem ser classificados de acordo com o tipo de instrução que reproduzem (tipo (b) ou (c)).
- **Modelo de criação de tarefas/processos:** uma ferramenta de re-execução pode lidar com sistemas que (a) admitem a entrada de novos processos durante a execução (criação dinâmica de processos) ou (b) sistemas cujo conjunto de processos é fixo do início ao fim da execução (criação estática de processos).
- **Re-execução de condições de corrida de dados ou de sincronização:** divide os sistemas de re-execução entre os capazes de reproduzirem condições de corrida de (a) dados e (b) sincronização (classificação de Ronsse, discutida na Seção 2.6). A esmagadora maioria dos sistemas de re-execução para sistemas de memória compartilhada situa-se, por razões de eficiência, na categoria (b).
- **Classe de instruções rastreadas:** diz respeito à relação entre instruções instrumentadas e instruções rastreadas. O sistema de re-execução pode produzir traços a cada instrução instrumentada (a) ou somente num subconjunto delas (b). Esta classificação não nos parece particularmente útil, já que (b) apenas indica que o sistema faz algum tipo de otimização, mas não quantifica essa otimização.
- **Método de integração:** a integração do mecanismo de re-execução no sistema de software cuja execução deseja-se gravar e re-executar pode ser (a) manual ou (b) automática. Além disso, a integração automática pode ser (b_c) completa ou (b_p) parcial. Obviamente, b_c implica mais praticidade do que b_p ou a.
- **Falha da re-execução:** em algumas situações, a re-execução do sistema pode falhar. Este critério classifica ferramentas de acordo a sua capacidade de detectar as situações em que um erro pode surgir: (a) o erro nunca é detectado, (b) o erro é detectado em certas situações (c) ou o erro é sempre detectado. Nós acreditamos que esta classificação não faz sentido se não considerarmos tipos específicos de erros, já que o escopo de uma ferramenta é sempre limitado em algum ponto (i.e., não existe uma ferramenta capaz de detectar consistentemente qualquer tipo de erro que possa levar à falha da re-execução).

- **Classe de instruções instrumentadas:** descreve a relação entre um programa e as instruções instrumentadas, indicando se (a) todas as instruções de um certo tipo são instrumentadas ou (b) apenas um subconjunto delas é instrumentado.
- **Alcance no tempo:** indica se (a) a re-execução deve sempre ser retomada do mesmo ponto da execução ou (b) se é possível retomá-la de pontos diferentes. Os sistemas de re-execução que situam-se na classe (b) implementam, em geral, algum tipo de mecanismo de *checkpointing*.
- **Alcance no espaço de tarefas:** indica se a re-execução envolve (a) todas as tarefas da execução original, (b) um subconjunto dessas tarefas ou (c) apenas uma tarefa.

Além das classes discutidas até agora, uma das categorizações informais mais freqüentes e importantes na literatura diz respeito à divisão das ferramentas de re-execução entre *baseadas em ordem* e *em conteúdo*. Essa classificação não é ortogonal à de Dionne, e tem uma granulosidade maior, mas particiona as técnicas, no entanto, num ponto importante.

Ferramentas de re-execução baseadas em conteúdo são aquelas que rastreiam o conteúdo (dados) produzido pelas operações não-determinísticas. Ferramentas baseadas em ordem, por sua vez, rastreiam a sua ordem relativa. A Figura 3-2 (a) mostra uma execução distribuída (i.e., baseada em mensagens) com uma condição de corrida, entre m_3 e m_4. A Figura 3-2 (b) mostra a re-execução baseada em conteúdo de P_2 e a Figura 3-2 (c) mostra a re-execução baseada em ordem.

Na re-execução baseada em conteúdo, os dados contidos em cada mensagem devem ser extraídos e armazenados. A vantagem da re-execução baseada em conteúdo, nesse caso, é que os processos podem ser re-executados isoladamente. Na Figura 3-2 (b), o processo P_2 é re-executado num ambiente artificial, onde as interações com outros processos são simuladas pela ferramenta de re-execução, que utiliza traços coletados durante a execução original.

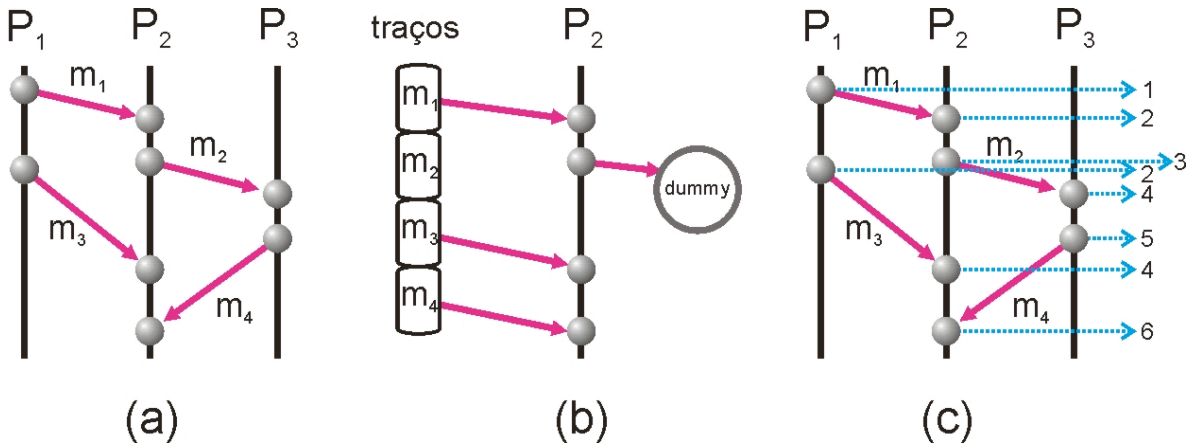

Figura 3-2. (a) Execução distribuída, (b) re-execução baseada em conteúdo do processo P_2 e (c) re-execução baseada em ordem.

A desvantagem é que o rastreio do conteúdo das mensagens gera uma quantia substancial de dados [104], produzindo, em última instância, os mesmos problemas encontrados pelas ferramentas de execução reversa discutidas na Seção 3.2.1 (um grande volume de dados e perturbações nas execuções).

Num certo sentido, isso reflete o fato que a abordagem baseada em conteúdo não é adequada para a reprodução de não-determinismos ligados à ordem relativa de eventos. A re-execução baseada em ordem, por outro lado, faz uso da hipótese de que a ordem dos eventos é a responsável pelo não-determinismo. Os sistemas de re-execução baseados em ordem operam atribuindo, durante a fase de gravação, *timestamps* lógicos (usando relógios de Lamport no exemplo da Figura 3-2 (c)) aos eventos relevantes, que são as mensagens da execução na Figura 3-2 (a) no nosso caso. No início da re-execução, o relógio lógico de cada processo é reiniciado. Daí para frente, o sistema de re-execução força os eventos a ocorrerem em instantes lógicos idênticos aos da execução original, possivelmente suspendendo a produção de um evento até o momento em que o instante correto é marcado pelo relógio lógico. O Capítulo 2 mostra que, na ausência de outras fontes de indeterminismo, isso é suficiente para garantir uma execução equivalente.

Para entendermos melhor esse funcionamento, suponha que durante a re-execução apresentada na Figura 3-2 (a), a mensagem m_4 fosse recebida, por P_2, antes de m_3. Isso representa uma ordenação diferente de eventos com relação à execução original. Note agora que o *timestamp* gravado para m_4 é 6, enquanto que o gravado para m_3 é 4 (vide Figura 3-2 (c)). De acordo com o histórico de eventos, isso significa que o evento está "adiantado". O sistema de re-execução deve, portanto, atrasar a execução de P_2 até que o próximo evento programado, o evento com *timestamp* 4, aconteça. Note que o sistema de re-execução baseado em eventos é, na verdade, um sincronizador de processos que, no pior caso, seqüencializa a execução do sistema distribuído para torná-la determinística. Por outro lado, a execução serial do sistema re-executado não implica no aumento do efeito do observador, já que o sistema de re-execução simplesmente reproduz uma execução previamente capturada.

O nosso interesse, particularmente, é em ferramentas de re-execução para sistemas paralelos e distribuídos. Essas ferramentas tipicamente preocupam-se com a reprodução de instruções fracamente determinísticas, integram-se à aplicação de forma automática, detectam consistentemente alguns tipos de erros e re-executam todas as tarefas (processos) ou um subconjunto delas. Ferramentas que se encaixam nesse perfil incluem o *Instant Replay* [111, 131], *RecPlay* [173], DIOTA [123], *JaRec* [71], *DejaVu* [33, 103], *Optimal Shared-Memory Replay* [144], *JNuke* [184], *Flight Data Recorder* [231], *BugNet* [142] e *Flashback* [196] para sistemas de

memória compartilhada, bem como o MAD [104], *Optimal Message Replay* [148], DPD [189], DEIPA [118], PDT [37], IVD [122], Buster [229] e Panorama [128] para sistemas de passagem de mensagens.

Algumas dessas ferramentas são sistemas de depuração completos, outras são estruturadas como módulos que provêm um serviço de re-execução. Temos ainda o caso de abordagens de re-execução projetadas como parte de sistemas operacionais distribuídos específicos, como é o caso do depurador apresentado por Herdieckerhoff e Ruget para o CHORUS [81] ou o do depurador apresentado por Elshoff para o Amoeba [59], ou até mesmo que requerem hardware especializado, como o *Flight Data Recorder* [231].

O primeiro sistema de re-execução de que se tem notícia foi proposto por Curtis e Wittie em 1982 [44]. Trata-se de um sistema de re-execução baseado em conteúdo e voltado para sistemas paralelos de passagem de mensagens. Uma abordagem semelhante é proposta por Smith [192] em 1984. No mundo dos sistemas paralelos, no entanto, a principal preocupação são as condições de corrida, onde abordagens baseadas em ordenação fazem mais sentido. Mellor-Crummey percebeu isso e, em 1987, publica um artigo [111] que apresenta o *Instant Replay*, a primeira ferramenta de re-execução a rastrear a ordem (e não o conteúdo) dos eventos de que se tem notícia. O *Instant Replay* representou um marco nos sistemas de re-execução, com muitas das ferramentas subseqüentes [37, 59, 71, 81, 104, 118, 122, 123, 128, 144, 148, 173, 229, 231] sendo baseadas em princípios semelhantes.

De fato, a única ferramenta que pudemos verificar, para sistemas baseados em passagem de mensagens, pós-*instant-replay* e que não utiliza uma abordagem baseada em ordem é o DPD [189], proposto por Side e Shoja em 1991. O funcionamento do DPD é baseado num algoritmo de *checkpointing* com *logging* otimista (assíncrono), desenvolvido por Johnson e Zwaenepoel [112]. Os *breakpoints* são especificados por cortes consistentes e a execução é reproduzida partindo do *checkpoint* viável mais próximo e re-executando, por uma abordagem baseada em conteúdo, os processos até esse ponto. Nesse sentido, o DPD é mais semelhante à ferramenta IGOR, discutida na Seção 3.2.1, do que ao restante das ferramentas de re-execução que lhe são contemporâneas.

Uma segunda vertente das ferramentas de re-execução, particular dos sistemas de memória compartilhada, são aquelas que operam sob o princípio que a reprodução do comportamento do escalonador é suficiente para garantir uma re-execução correta [33, 103, 184, 196]. O problema com essa hipótese é que ela se sustenta apenas para sistemas pseudo-paralelos. Em sistemas multiprocessados, as condições de corrida podem se desenrolar de forma não-determinística, ainda que o escalonamento seja reproduzido de forma idêntica.

3.2.3 Re-execução em sistemas de memória compartilhada e a detecção de condições de corrida

A re-execução segue a tendência da detecção de condições de corrida e apresenta-se mais difícil de tornar concreta de forma prática em sistemas de memória compartilhada do que em sistemas baseados em passagem de mensagens. O grande problema enfrentado por implementações em *software* desses mecanismos de re-execução é o alto custo envolvido na captura de informações de ordenação para acessos individuais a localizações de memória, essencialmente porque tratam-se de eventos com granulosidade muito fina. O resultado vem sob a forma de volumosos arquivos de traços (ainda maiores do que os gerados para os sistemas baseados em passagem de mensagens) e, no caso das técnicas de detecção *on-line* (veja a Seção 3.2.6), do excesso de perturbação nas execuções capturadas [112, 144].

Uma possível solução para esse problema seria rastrear apenas a ordem das operações de sincronização, já que a freqüência de ocorrências desse tipo de operação numa execução é, tipicamente, muito menor do que a ocorrência de acessos a regiões compartilhadas de memória [42, 71, 111, 131, 148, 173]. O problema óbvio com essa abordagem é que ela garante a re-execução determinística apenas para programas livres de condições de corridas de dados. Isso levanta uma espécie de paradoxo, onde a ferramenta de depuração assume que o programa é livre de defeitos relevantes ao seu domínio.

Em 1993, Netzer prova que o rastreamento das arestas que se formam entre acessos produzidos por processos distintos na redução transitiva da relação de dependência de dados (apresentada na Seção 2.6.1) é suficiente para garantir a correta reprodução de condições de corrida. A definição de Netzer para uma re-execução bem-sucedida de um sistema de memória compartilhada é dada a seguir.

Definição 3-8: Dadas duas execuções $P = \left\langle E, \xrightarrow{T}, \xrightarrow{D} \right\rangle$ e $P' = \left\langle E, \xrightarrow{T'}, \xrightarrow{D'} \right\rangle$, P' é uma *reprodução bem-sucedida* se e somente se:

1. $E' = E$, e,
2. $\xrightarrow{D} = \xrightarrow{D'}$.

As arestas entre processos que surgem na redução transitiva de $\xrightarrow{D}$ são denominadas, por Netzer, condições de corrida de fronteira (*frontier races*).

Definição 3-9: Dada uma execução $P = \left\langle E, \xrightarrow{T}, \xrightarrow{D} \right\rangle$, uma *condição de corrida de fronteira* $\langle a,b \rangle$ existe entre dois eventos *a* e *b* se e somente se:

1. *a* e *b* pertencem a processos diferentes, e
2. $a \xrightarrow{D/} b$, onde $\xrightarrow{D/}$ é a redução transitiva de $\xrightarrow{D}$.

O fecho transitivo de $\xrightarrow{D/}$ leva, por definição, à relação $\xrightarrow{D}$. Netzer usa esse fato na demonstração do seguinte teorema [144]:

Teorema 3-1 (*Teorema da Reprodução*): Uma reprodução é bem-sucedida se e somente se as condições de corrida de fronteira forem resolvidas, durante a reprodução, na mesma ordem em que ocorrem na execução original.

A implicação direta desse teorema é que apenas o primeiro acesso à memória compartilhada, feito por processos distintos, precisa ser rastreado. A Figura 3-3 (arestas locais omitidas) mostra (a) as dependências dinâmicas rastreadas pelo *Instant Replay* (com algumas arestas transitivas) e (b) o rastreamento ótimo[5]. O termo *condição de corrida de fronteira* é usado por Netzer porque essas condições de corrida envolvem pares de eventos abaixo dos quais um corte consistente (ou fronteira consistente) pode ser desenhado [144]. Os experimentos conduzidos pelo autor concluem que o emprego da técnica reduz a quantia de informações coletadas, no pior caso analisado, em 90%. No melhor caso, há reduções de até 99,9%.

Um conceito anterior relacionado e com nome semelhante, mas propósito distinto, é o conceito de fronteira de condição de corrida, proposto por Choi e Min em 1991 [32]. Uma fronteira de condição de corrida é um corte consistente que divide a execução do sistema concorrente em uma porção determinística e uma porção não-determinística. A idéia vem da observação de que a execução de um sistema concorrente é determinística até a primeira condição de corrida; isto é, até a fronteira que separa a primeira condição de corrida do restante da execução. Particularmente, um mecanismo de re-execução capaz de reproduzir apenas as condições de corrida de sincronização (a ordem das operações de sincronização) garante que a re-execução é correta até a primeira condição de corrida de dados. Essa é a essência da técnica de re-execução proposta por Choi e Min.

[5] Essa figura foi extraída (e adaptada) do artigo original de Netzer [47]. É curioso notar que, embora Netzer indique que a Figura 3-3 (a) representa a relação de dependência de dados ("*The dynamic dependence graph shown in Figure 1a is a graphical representation of* $\xrightarrow{D}$"), claramente faltam arestas para que essa figura represente o fecho transitivo de $\xrightarrow{DD}$. O que essa figura representa, de fato, são as arestas que as técnicas anteriormente propostas (particularmente o *Instant Replay*) desnecessariamente rastreariam, como o próprio Netzer nota no artigo. Portanto, acreditamos que se trate de um equívoco no artigo.

Note que a técnica pressupõe a atuação de algum mecanismo capaz de detectar condições de corrida de dados (já que é necessário encontrar a primeira). Fica implícito também que a detecção de condições de corrida de dados é feita em paralelo, de forma *on-line*, junto com a gravação do restante da execução. Isso implica que a técnica de Choi e Min evita a armazenagem de grandes quantidades de traços, mas apresenta problemas ligados ao efeito do observador, já que os mecanismos de coleta de informações requeridos por algoritmos de detecção de condições de corrida de dados geram um *overhead* substancial [173].

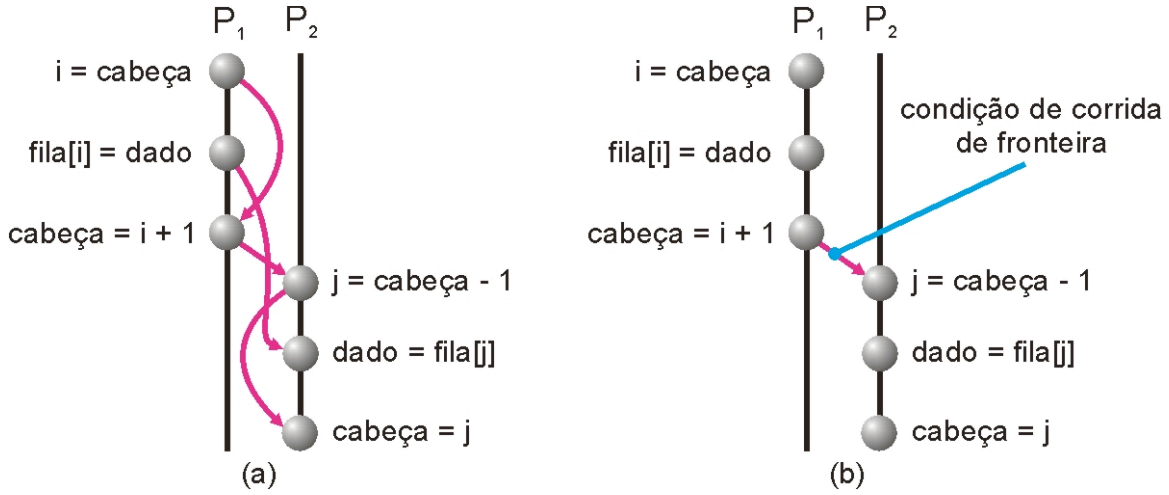

Figura 3-3. (a) Arestas de dependência rastreadas pelo *Instant Replay* e (b) rastreamento ótimo, após redução transitiva.

O que é importante notar é que, para fins de depuração, detectar a primeira condição de corrida de dados já é suficiente. Isso porque, após a primeira condição de corrida de dados, a execução do sistema pode se tornar sem sentido, ou ainda produzir novas condições de corrida decorrentes de erros provocados pela primeira [149]. Essas condições de corrida "falsas", se detectadas, contribuem de forma negativa para o efeito labirinto. Choi e Min, no entanto, não descrevem como encontrar, de forma eficiente, essa "primeira" condição de corrida.

O problema é resolvido em 1999 por Ronsse, então no Departamento de Eletrônica e Sistemas de Informação da Universidade de Ghent, na Bélgica, durante a implementação do *RecPlay* [173]. O *RecPlay* é um sistema de re-execução para aplicações *multithreaded* escritas para o Sistema Operacional Solaris [206] em microprocessadores SPARC. Para resolver as questões ligadas às perturbações trazidas por um detector de condições de corrida em software, Ronsse adota uma abordagem em 3 fases.

Na primeira fase, o *RecPlay* simplesmente grava a ordem das operações de sincronização usando relógios de Lamport, ignorando por completo quaisquer condições de corrida de dados que possam ter ocorrido. O método utilizado para a gravação das operações chama-se ROLT –

Reconstruction Of Lamport Timestamps – e foi proposto por Levrouw, Audenaert e Campenhout em 1994 [114]. No ROLT, um relógio escalar é anexado a cada processo em execução, bem como a cada objeto de sincronização (monitor, semáforo ou mutex, por exemplo). Ao início de qualquer operação de sincronização, os valores dos relógios lógicos do processo (ou *thread*) e do objeto de sincronização são comparados e o valor máximo é extraído. Esse máximo é então incrementado e atribuído a ambos relógios lógicos. A Figura 3-4 mostra um código de instrumentação hipotético para Java.

```
synchronized(lockObject){
   Thread t = Thread.currentThread();
   int nextValue =
      Math.max(lockObject.lamportClock(),
         t.lamportClock());
   nextValue++;
   t.nextLamportClock(nextValue);
   lockObject.lamportClock(nextValue);
   // Code...
}
```

Figura 3-4. Código de instrumentação hipotético, adicionado aos blocos *synchronized* de um programa Java, para gravação da ordem das operações de sincronização.

O tamanho do arquivo de traços pode ser reduzido por meio de duas otimizações. A primeira delas parte da observação de que os incrementos sofridos por um relógio lógico, a cada evento, são geralmente pequenos. Usando uma codificação de tamanho variável, o tamanho médio de cada entrada pode ser reduzido substancialmente. Economias ainda maiores podem ser atingidas se forem armazenados apenas os incrementos maiores do que 1. Para tanto, é necessário armazenar também o número de eventos cujo incremento foi 1 desde o último evento para o qual o incremento não foi 1 [114].

Na segunda fase, seguindo a lógica de Choi e Min, o *RecPlay* re-executa o programa sob a ação de um mecanismo de detecção de condições de corrida de dados – que atua inspecionando todos os acessos à memória – até que a primeira condição de corrida de dados seja detectada ou, na ausência de condições de corrida de dados, até que o programa termine. A grande beleza dessa abordagem vem do fato de que a execução sendo reproduzida (e sob o efeito do detector de condições de corrida) foi capturada por um mecanismo extremamente leve, que perturba muito pouco a execução (reduz a velocidade de execução em apenas 2%, em média). Muito embora a técnica de detecção de condições de corrida seja bastante pesada (acarreta tempos de execução até 600% maiores), isso não implica na exacerbação do efeito do observador, já que a re-execução reproduz as operações de sincronização na mesma ordem em que ocorreram durante a primeira fase.

A re-execução no ROLT faz uso de um teorema da reprodução mais forte do que o proposto por Netzer. Esse teorema mais forte impõe, durante a re-execução, mais restrições de sincronização do que as originalmente presentes. Essas restrições são necessárias, no entanto, para garantir a corretude da re-execução na presença de relógios lógicos que capturam menos informações a respeito da causalidade. Recapitulando que $L : E \rightarrow N$ é a função que atribui o valor do relógio de Lamport a um evento, temos:

Teorema 3-2 (Teorema da Reprodução do ROLT): Seja $P = \left\langle E, \xrightarrow{D}, \xrightarrow{T} \right\rangle$ uma execução e $P' = \left\langle E', \xrightarrow{D'}, \xrightarrow{T'} \right\rangle$ a execução gerada pelo mecanismo de reprodução. Uma reprodução é correta se:

1. $E = E'$;
2. Dados, $a, b \in E; a', b' \in E'$, se $L(a) < L(b)$ então $a' \xrightarrow{T'} b'$.

Isso implica que o ROLT não permite que um evento *b* ocorra, durante a re-execução, antes de que todos os eventos em $\{a \in E' : L(a) < L(b)\}$ tenham ocorrido (ainda que *a* e *b* não sejam causalmente relacionados). Para tanto, o ROLT depende do uso de um certo número de contadores durante a re-execução, chamados de *slice counters* por Levrouw, um para cada processo P_i. Determinar de antemão o número de processos (ou *threads*, no caso do *RecPlay*) que participam na execução concorrente não é um problema para o *RecPlay*. Cada contador de fatias $SC[i]$ indica o valor do tempo lógico para o próximo evento que deve ocorrer em P_i. Antes de produzir um evento e'_{ij}, o processo P_i deve se assegurar de que $\forall k : SC[k] \geq L(e'_{ij})$.

Note que essa condição é suficiente, mas não necessária, para que uma re-execução seja correta. Na prática, no entanto, o ROLT mostra-se uma técnica bastante eficiente durante a fase de gravação, em particular porque a melhor técnica previamente desenvolvida (a técnica proposta por Netzer [144]) depende do uso de relógios vetoriais durante a gravação. Ronsse aponta para reduções de velocidade da ordem de 2%, em média. Levrow mostra que a abordagem de Netzer gera traços 56,8% maiores que a sua abordagem (o ROLT), em média, e a ainda que sua técnica captura apenas as dependências não-transitivas de dados; isto é, é tão ótima quanto a de Netzer. Uma outra vantagem, talvez até mais interessante, é que a o ROLT utiliza relógios escalares, que não dependem de um conhecimento a priori do número de processos participantes.

O método empregado pelo *RecPlay* para a detecção de condições de corrida é baseado em idéias provenientes de uma família de métodos semelhantes [3, 6, 31, 32, 60, 135]. Esses métodos são fundados na observação de que o rastreamento das operações de sincronização induz um

particionamento da execução em uma série de segmentos seqüenciais, que ocorrem entre as operações de sincronização. Como apenas as operações de sincronização alteram os relógios lógicos, o tempo lógico dentro de um segmento é o mesmo para todas as instruções a ele pertencentes. Para ver porque isso é verdade, note que todas as instruções executadas no segmento *a* da Figura 3-5 (a), por exemplo, ocorrem antes das instruções executadas no segmento *f*. Isso é conseqüência do fato de que as instruções, dentro de cada segmento, são executadas de forma estritamente seqüencial (ao menos aparentemente).

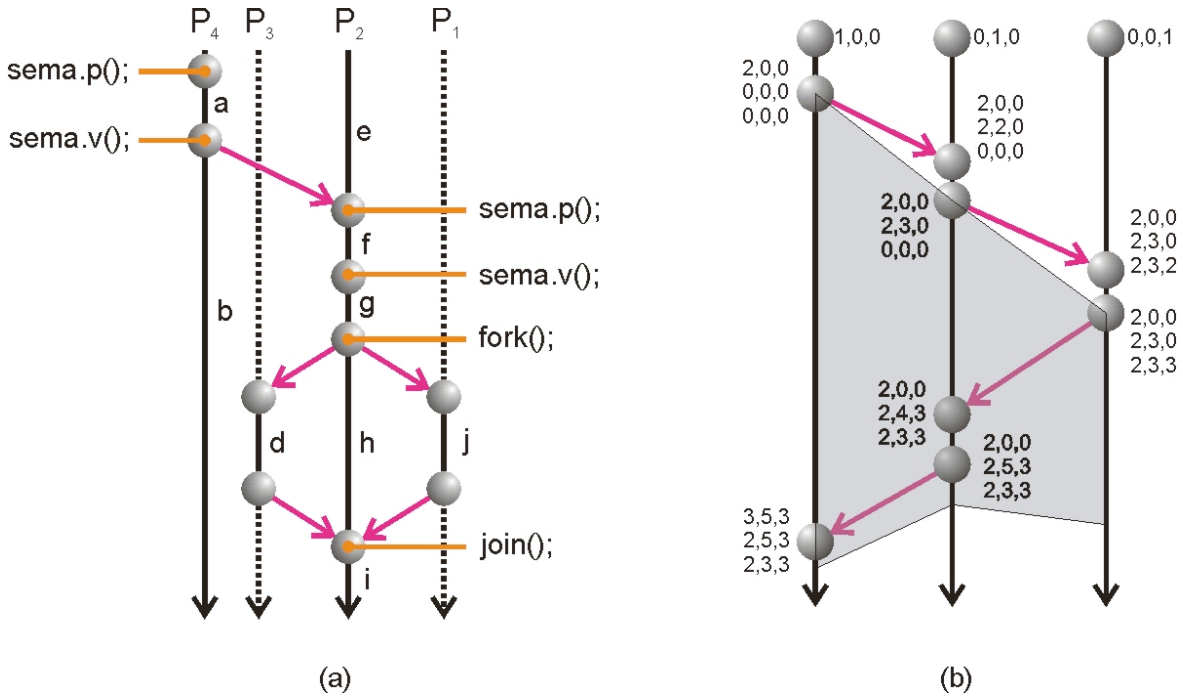

Figura 3-5. (a) Execução concorrente com trechos seqüenciais demarcados. (b) Janela na execução onde ainda podem haver segmentos concorrentes.

Os métodos sob discussão constroem, para cada segmento *i*, dois conjuntos: o conjunto $R(i)$ das localizações de memória lidas e o conjunto $W(i)$ das localizações de memória escritas. Dado S, o conjunto de segmentos produzidos por uma execução, a idéia é que se existem $a,b \in S$ tais que $W(a) \cap R(b) \neq \varnothing$ e $a \| b$ então há, possivelmente, uma condição de corrida na execução. Essas condições de corrida são, na verdade, condições de corrida *possíveis* sobre o conjunto F_{SYNC} apresentados na Seção 2.6.1. Trata-se, portanto, de uma aproximação que pode conter condições de corrida possíveis apenas sobre prefixos não-viáveis da execução original. Essa aproximação, no entanto, representa um bom compromisso entre precisão e praticidade.

Durante a segunda fase, o *RecPlay* utiliza um sistema de relógios vetoriais para detectar segmentos concorrentes. Novamente, a cada processo é atribuído um relógio vetorial, bem como

são atribuídos relógios vetoriais a cada objeto de sincronização. Quando um segmento termina (pela ocorrência de uma operação de sincronização), seu tempo lógico é comparado com o tempo lógico dos outros segmentos terminados e os segmentos potencialmente concorrentes são localizados. Os endereços acessados são então comparados e as condições de corrida, detectadas. Note que os sistemas re-executados pelo *RecPlay* são altamente dinâmicos com relação ao número de processos participantes. Isso poderia, em princípio, representar um obstáculo ao uso de relógios vetoriais. O *RecPlay*, no entanto, só utiliza relógios vetoriais na segunda fase, quando já há informações suficientes para determinar o número total de processos participantes e pré-alocar o número correto de entradas em cada relógio vetorial.

Os conjuntos de endereços lidos e escritos são mantidos em dois *bitmaps* (*arrays* de bits indexados pelo endereço), um para localizações lidas e outro para localizações escritas. Cobrir um espaço de endereçamento de 32 bits demanda, no entanto, um *bitmap* de 512 MB. Para reduzir o uso de memória, o *bitmap* é implementado três níveis, dois de 9 bits e um de 14 bits. O mapa é então alocado sob demanda, em porções de 2kb, reduzindo drasticamente seu tamanho típico. No pior caso, se todas as localizações de memória forem acessadas, o *bitmap* ainda pode ocupar 512 MB.

Uma questão que surge do uso de segmentos para o rastreio de localizações de memória acessadas é que uma execução pode produzir um número muito grande deles. Se todos os históricos de acesso de todos os segmentos executados tiverem de ser mantidos em memória para comparação com futuros históricos de acesso, a memória disponível pode se esgotar rapidamente. Além disso, o sistema de detecção de condições de corrida pode ser detido em comparações sem sentido. É necessária, portanto, a adoção de algum mecanismo que permita o descarte dos históricos de acessos dos segmentos para os quais sabemos que o futuro envolvimento em condições de corrida é impossível. Os relógios matriciais, assim chamados por Singhal e Raynal [168], cumprem justamente a esse propósito.

Nós vamos começar apresentando relógios matriciais e suas propriedades no contexto de sistemas de passagem de mensagens, discutindo o seu uso no contexto do *RecPlay* logo em seguida. Um relógio matricial é uma função de $MC : E \rightarrow N^{n^2}$, que mapeia eventos em matrizes $n \times n$. O tempo matricial no processo P_i é definido como uma matriz $A^i_{n \times n}$, onde:

1. $a^i_{i1},...,a^i_{in}$ é um vetor cujo conteúdo é o tempo vetorial de P_i;
2. $a^i_{j1},...,a^i_{jn}, j \neq i$, representa o conhecimento de P_i a respeito do tempo vetorial em P_j.

A atualização do tempo matricial num sistema de passagem de mensagens é feita da seguinte forma:

1. Quando P_i envia uma mensagem *m* para P_j:

   a. P_i incrementa a_{ii}^i,

   b. P_i envia um *timestamp* $T_{n\times n} = A_{n\times n}^i$ junto com *m* para P_j.

2. Quando P_j recebe uma mensagem *m* de P_i,

   a. P_j incrementa a_{jj}^j,

   b. para cada componente a_{hk}^j em $A_{n\times n}^j$, o novo tempo matricial é dado por $\max\{t_{hk}, a_{hk}^j\}$.

Note que as linhas de um relógio matricial em P_j representam o conhecimento desse processo a respeito do tempo vetorial no restante dos processos. Particularmente, a linha $a_{j1}^j,...,a_{jn}^j$ representa o conhecimento de P_j sobre o tempo vetorial dele próprio; i.e., representa o tempo vetorial em P_j. Algo importante de notar para que possamos entender melhor as propriedades dos relógios matriciais é que o conhecimento de P_j sobre o tempo vetorial de P_j é, naturalmente, mais acurado do que o conhecimento do restante dos processos. O que queremos dizer com isso é que $[a_{j1}^i,...,a_{jn}^i] \leq [a_{j1}^j,...,a_{jn}^j]$, para todo *i* e todo *j*. Não vamos demonstrar esse resultado no texto, mas não é difícil entender que isso é correto, se levarmos em conta que algo semelhante ocorre nos relógios vetoriais (no tempo vetorial, $a_i^j \leq a_j^i$ para todo *i* e todo *j*).

Se construirmos um vetor $V_{\min} = \{t_1,...,t_n\}$ tomando o valor mínimo em cada coluna em $A_{n\times n}^j$ (i.e., $t_i = \min_{1\leq k\leq n}\{a_{k,i}\}$), obtemos um *timestamp* vetorial que sabemos ser menor ou igual, de acordo com o conhecimento de P_j, do que o tempo lógico do restante dos processos do sistema distribuído. Pelo que discutimos no parágrafo anterior, no entanto, o tempo vetorial percebido por P_j é sempre menor ou igual ao tempo lógico "real" do restante dos processos. Isso implica que o instante lógico demarcado por $V_{\min}$ é menor que o relógio vetorial de cada processo. É portanto seguro supor que não há nenhum processo no sistema distribuído cujo instante lógico é menor que $V_{\min}$. O *RecPlay* utiliza essa informação para descartar, com segurança, os históricos de acessos de segmentos cujo *timestamp* vetorial é menor que $V_{\min}$. Nós também não vamos demonstrar, mas $V_{\min}$ define sempre um corte consistente. Isso decorre do fato que o conjunto dos cortes consistentes forma um reticulado sob as operações de união e intersecção e que, na verdade, a extração dos mínimos corresponde a uma operação de intersecção de históricos causais.

Um dos maiores problemas com relógios matriciais é que eles produzem ainda mais *overhead* do que os relógios vetoriais ao forçarem a transmissão de algo da ordem de n^2 bits por mensagem [168]. As técnicas empregadas na redução da quantia de dados transmitidas com cada mensagem num sistema de relógios vetoriais (mencionadas na Seção 2.3) podem ser adaptadas a relógios matriciais mas, como no caso dos relógios vetoriais, essas técnicas apenas amenizam os problemas. Por se tratar de um sistema de re-execução que pressupõe a existência de memória compartilhada, no entanto, o *RecPlay* é capaz de reduzir substancialmente o custo e a precisão do sistema de relógios matriciais pela adoção de uma variante denominada, pelos autores do *RecPlay*, de "relógios matriciais curiosos" (*snooped matrix clocks*). Além disso, as questões ligadas aos sistemas em que o número de processos varia dinamicamente ainda persistem, muito embora, por razões semelhantes às que citamos para os relógios vetoriais, isso não seja um problema no *RecPlay*.

Relógios matriciais curiosos são baseados na idéia de que um determinado processo P_i pode, a qualquer instante, decidir atualizar o seu conhecimento com relação ao tempo lógico de um outro processo P_j. Num sistema de passagem de mensagens, P_i atualiza o seu conhecimento enviando uma requisição (mensagem) de atualização a P_j e aguardando uma resposta, que contém o tempo vetorial de P_j atualizado, conforme mostra a Figura 3-6 (a).

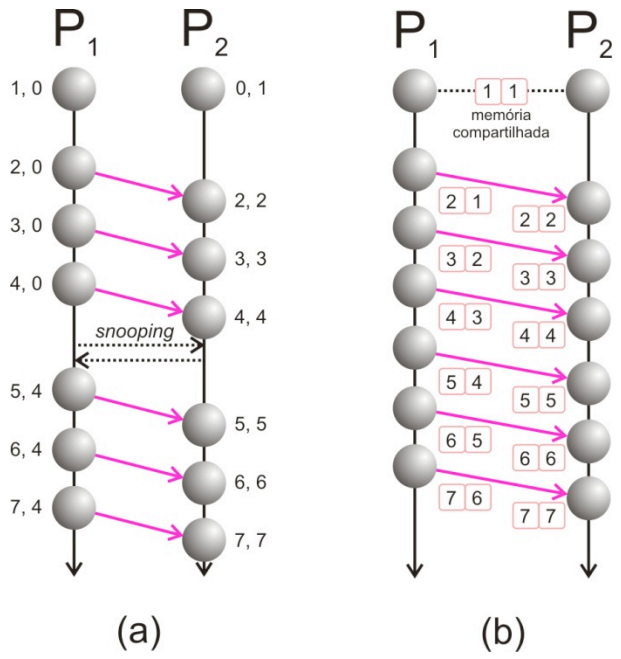

Figura 3-6. (a) Relógio curioso num sistema de passagem de mensagens. (b) Relógio curioso implementado em memória compartilhada.

Em sistemas de memória compartilhada como os tratados pelo *RecPlay*, no entanto, essa sincronização não precisa ser feita por requisições. Basta que se construa um relógio matricial cujas linhas contêm, de fato, os relógios vetoriais de cada processo no sistema concorrente; ou, no caso da Figura 3-6, um relógio vetorial cujos elementos contêm o relógio escalar de cada processo. Note que o relógio matricial assim construído é o mais atualizado possível. O uso desse relógio para o descarte de segmentos inativos traz contribuições consideráveis, conforme apontam os autores do *RecPlay* [173]. A Figura 3-7 mostra uma execução hipotética e a janela de segmentos ativos quando são usados relógios matriciais (a) e relógios matriciais curiosos (b).

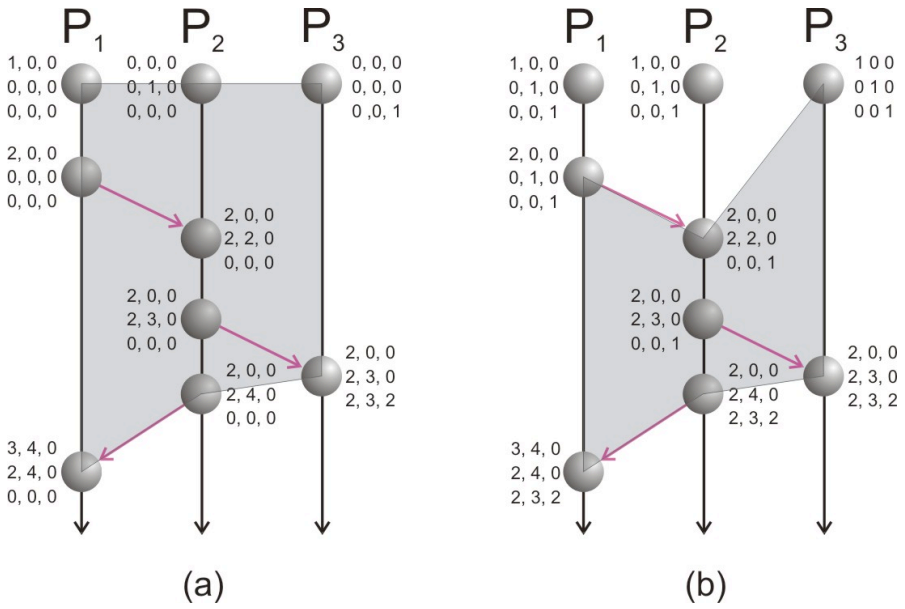

Figura 3-7. Janela de segmentos ativos quando são utilizados (a) relógios matriciais e (b) relógios matriciais curiosos.

Uma pergunta que surge naturalmente neste contexto é se o uso de relógios vetoriais curiosos no interior de cada relógio matricial não tornaria ainda melhor o desempenho do algoritmo de descarte de segmentos obsoletos. A resposta é que o uso de *snooping* nos relógios vetoriais destrói suas propriedades de captura da causalidade, com dois efeitos negativos. O primeiro deles é que o algoritmo de descarte se torna incorreto – é possível que hajam situações onde o histórico de acessos de segmentos que ainda podem executar concorrentemente sejam descartados. O segundo é que os *timestamps* vetoriais não mais podem ser usados na detecção de condições de corrida [19], já que deixam de revelar se dois segmentos são de fato concorrentes.

Até agora discutimos a primeira e a segunda fase do *RecPlay.* A terceira fase é bastante simples quando comparada às duas anteriores e cumpre o propósito de revelar quais as instruções que provocam a condição de corrida de dados detectada na segunda fase. Essa detecção é feita mediante

uma segunda re-execução, que avança o programa até que se tornem ativos os segmentos causadores da dita condição de corrida. O *RecPlay* produz então uma análise mais refinada da execução desses segmentos, buscando pelo primeiro par de instruções que conflita no endereço de memória previamente detectado.

O *RecPlay* representa, em nossa opinião, a melhor combinação de técnicas para a implementação de um sistema de re-execução em *software* para sistemas de memória compartilhada. Há um compromisso razoável entre desempenho, interferência e fidelidade na re-execução, atingido pela restrição de que o *RecPlay* não é capaz de reproduzir a execução de sistemas concorrentes que contenham condições de corrida de dados. Isso não é um fator limitante, no entanto, já que o *RecPlay* é capaz de detectá-las. Nós não conhecemos nenhum outro trabalho que tenha produzido resultados tão praticamente aplicáveis. Uma das ferramentas baseadas no *RecPlay*, o DIOTA [123], pode inclusive ser obtido livremente na Internet, sendo capaz de reproduzir a execução e detectar condições de corrida em sistemas concorrentes baseados em *threads* POSIX e que executam em plataforma Intel (IA-32).

Algo interessante de se especular é se seria possível produzir, em software, um sistema de re-execução capaz de acomodar condições de corrida de dados sem restringir o hardware subjacente a modelos que implementem consistência seqüencial ou de processador. Nós acreditamos que um sistema assim seria bastante difícil, senão impossível, de produzir, já que as decisões de *cachê* são tipicamente tomadas de forma transparente com relação às aplicações.

Além dos sistemas de re-execução em software, algumas alternativas são apresentadas por sistemas de re-execução que dependem de hardware especializado. O *Flight Data Recorder* (FDR) [231], por exemplo, propõe a adição de hardware especificamente destinado à captura, com baixo *overhead*, da ordem relativa dos acessos à memória. O protótipo apresentado no artigo, o FDR1, implementa o esquema de compressão de traços proposto por Netzer aliado a um mecanismo de *checkpointing*. Além da captura da ordem relativa de eventos, o FDR1 captura os resultados de *loads* emitidos por cada processador a endereços de memória mapeados em dispositivos de entrada e saída, efetivamente viabilizando a re-execução da entrada. O FDR é capaz de re-executar interrupções fielmente, usando um contador de instruções, bem como o comportamento de peças de hardware que utilizam DMA. Um trabalho semelhante é desenvolvido por Bacon e Goldstein [13], com o seu gravador assistido por hardware. A proposta do FDR difere da do *RecPlay* na medida em que o FDR é projetado para uma captura bastante completa de um pequeno trecho da execução – os últimos segundos antes de um erro fatal – ao passo que o *RecPlay* captura execuções longas, mas de forma mais incompleta. Além disso, o FDR não é capaz de reproduzir as execuções que captura – os *logs* do FDR devem ser analisados por outros sistemas de software.

Outros exemplos relacionados recentes incluem o *ReEnact* [166], proposto por Milos Prvulovic e Josep Torellas e o CORD [165], de Prvulovic. O *ReEnact* faz uso dos mecanismos providos por processadores equipados com especulação em nível de *threads* (*thread-level speculation* ou TLS) [199] para implementar uma ferramenta que, entre outras coisas, detecta condições de corrida de dados de maneira bastante eficiente (5,8% de *overhead*, em média). O *ReEnact* é interessante por quatro motivos. Em ordem crescente de relevância, o primeiro motivo é porque se trata de uma ferramenta de depuração que depende de hardware especializado. O segundo é que o sistema faz um uso bastante interessante – detecção de condições de corrida – de um mecanismo originalmente projetado para viabilizar a paralelização automática agressiva de programas seqüenciais. O terceiro é que o entranhamento do *ReEnact* no hardware permite que o sistema lide graciosamente com sistemas em que o modelo de consistência de memória é fraco. O quarto é que, graças ao hardware com TLS, o *ReEnact* é capaz de corrigir certos tipos de condições de corrida em tempo de execução, efetivamente "reparando" a execução e prevenindo falhas.

De maneira semelhante ao *ReEnact*, o CORD se apóia nos protocolos de coerência de *cachê* de ambientes multiprocessados para implementar um detector de condições de corrida de dados de baixo *overhead*. Trata-se, no entanto, de um mecanismo diferente, já que o CORD utiliza relógios escalares e não depende da presença de TLS. O FDR, o *ReEnact* e o CORD estão limitados a *buffers* de hardware (em memória ou *cachê*) para a implementação de suas funcionalidades de rastreio. A implicação disso para o FDR é que ele é capaz de armazenar apenas alguns segundos de execução. As implicações para o *ReEnact* e o CORD é que condições de corrida distantes no tempo podem não ser detectadas. Temos aqui mais um compromisso entre eficiência, completeza e espaço.

Uma conseqüência desagradável óbvia do uso de assistência de hardware é o acoplamento de baixo nível que se estabelece entre ferramenta de depuração e hardware – expandindo as questões ligadas à heterogeneidade em ainda mais uma dimensão – bem como o condicionamento da correta operação da ferramenta à presença desse hardware, não necessariamente padronizado ou comercialmente disponível. Há aqui uma excelente ilustração do problema enfrentado pelas ferramentas de depuração de uma maneira geral: quanto mais completa a funcionalidade, maior o acoplamento e mais restrita a aplicabilidade. Vamos explorar essa questão na Seção 3.3. Por outro lado, podemos argumentar que é preciso evoluir o apoio de hardware à depuração de sistemas concorrentes em alguma direção – portanto esses trabalhos são deveras valiosos.

3.2.4 Re-execução de entrada

Conforme mencionamos anteriormente, a reprodução das condições de corrida resolve apenas parte do problema de re-execução. Isso porque a maior parte das aplicações interage, em maior ou

menor grau, com dispositivos de entrada e saída, e o resultado dessas interações é, por vezes, imprevisível. Os dados recebidos de um teclado, por exemplo, podem variar a cada execução; o envio de uma mensagem pela rede pode falhar; um bloco de dados lido de um disco rígido pode ser alterado. Tudo em conseqüência de fatores que estão, tipicamente, além do escopo da aplicação. As técnicas que tratam da reprodução desse tipo de interação (as *instruções fortemente não-determinísticas* de Dionne [47]) serão abordadas nesta seção.

Nós vamos dividir as técnicas de reprodução de entrada em duas categorias: as de nível de interface e as de nível de sistema.

Definição 3-10: Uma técnica de reprodução de entrada é de *nível de interface* quando atua no nível de abstração do gerenciador de janelas ou arcabouço de interface com o usuário (por exemplo, MFC, AWT, API Win32, Navegador Web, etc.).

Técnicas de nível de sistema atuam, por outro lado, num nível mais baixo.

Definição 3-11: Uma técnica de reprodução de entrada é de *nível de sistema* quando atua na interface entre o ambiente de execução e suas aplicações.

A Figura 3-8 ilustra as diferenças arquiteturais entre esses dois tipos de sistemas. É interessante notar que as técnicas de reprodução de entrada são normalmente baseadas em conteúdo, já que a reprodução real do comportamento de um dispositivo de entrada não é em geral factível.

Ferramentas de re-execução de nível de interface são tipicamente mais limitadas do que as de nível de sistema. Conforme mostra a Figura 3-8 (a), essas ferramentas são normalmente capazes de capturar e reproduzir apenas um subconjunto das interações da interface com o usuário (UI) da aplicação, por questões técnicas diversas e particulares de cada plataforma.

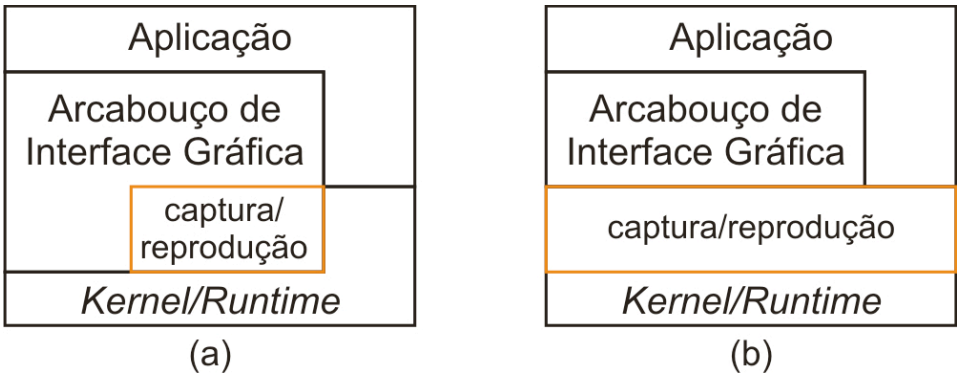

Figura 3-8. Visão geral dos sistemas de reprodução de entrada. (a) nível de interface (b) nível de sistema.

Dito isso, uma porção substancial dos sistemas de reprodução de entrada disponíveis são de nível de interface. Especulamos que isso ocorra em virtude de fatores como portabilidade, simplicidade e a viabilidade da implementação de ferramentas de nível de sistema em sistemas operacionais de código fechado, como o *Microsoft Windows*.

Alguns exemplos relevantes de sistemas de reprodução de entrada de nível de interface são o CAPBAK/X [195], da *Software Research*, o *JavaStar* [133], da *Sun Microsystems*, o *Rational Robot* [88], da *Rational Corporation* e o *Selenium IDE* [153], um projeto livre e de código aberto. O CAPBAK é implementado como uma extensão do X11, sendo capaz de monitorar eventos de teclado e mouse, bem como eventos direcionados a janelas e *widgets* específicos. O *JavaStar* segue pelo mesmo caminho – sua implementação compreende uma versão instrumentada do *Abstract Widget Toolkit* (AWT) e uma biblioteca para a gravação/reprodução de eventos. O *Rational Robot*, por sua vez, é composto por uma coleção de produtos, entre eles um gravador/reprodutor de eventos de interface gráfica para Windows, um gravador/reprodutor de interações HTTP/HTTPS para teste de aplicações Web e um gravador/reprodutor de interações para GUIs de aplicações Java (possivelmente baseado numa versão instrumentada do AWT). Finalmente temos o Selenium IDE, que é capaz de gravar/reproduzir interações com documentos XHTML no navegador *Firefox*. Note que uma porção substancial desses sistemas incluem diversas outras funcionalidades, mas essas funcionalidades estão além do escopo desta dissertação.

Com relação às ferramentas de nível de sistema, podemos destacar a TORNADO [43], e a *jRapture* [200]. A TORNADO é uma ferramenta que reproduz o comportamento de chamadas de sistema, algo que é suficiente para a reprodução da interação com o hardware de entrada e saída em sistemas operacionais modernos. A TORNADO foi desenvolvida para o Linux IA-32 e requer o uso de um núcleo modificado. A *jRapture*, por sua vez, é destinada a aplicações Java e requer o uso de um JDK modificado, que instrumenta a porção da API Java que interage com o sistema operacional subjacente.

Há uma série de questões interessantes que surgem no contexto da re-execução de chamadas de sistema. Entre elas, há a questão de que certas chamadas de sistema precisam ser refeitas durante a re-execução – fingir reproduzi-las por uma abordagem baseada em conteúdo não é suficiente. Chamadas de sistema como as que resultam do uso de *printf* são um exemplo. Se todas as chamadas a *printf* forem simplesmente reproduzidas (i.e, o valor devolvido durante a gravação é fornecido ao cliente, mas a chamada de sistema não é refeita), o programa pode deixar de produzir ações observáveis, levando ao *efeito buraco negro* [42]. Outro exemplo são as chamadas de sistema que produzem efeitos colaterais no núcleo do sistema operacional como, por exemplo, `mmap()` e `munmap()`.

Ainda outra questão diz respeito às chamadas de sistema que produzem efeitos colaterais na área de memória do usuário. Algumas dessas chamadas alteram regiões de memória que podem ser determinadas com certa facilidade; outras, como a `ioctl()`, podem produzir efeitos difíceis de prever. Para lidar com esse tipo de problema, a TORNADO instrumenta as primitivas de acesso à memória de usuário usadas pelo núcleo. Os endereços alterados por cada chamada são então rastreados para que seus efeitos colaterais possam ser posteriormente reproduzidos. Não é muito claro como a TORNADO lida com sistemas concorrentes, mas acreditamos que a ferramenta pressupõe que as chamadas de sistema serão refeitas numa ordem consistente com a fase de gravação, algo que pode dificultar a aplicação da ferramenta.

Um abordagem semelhante – instrumentar as interfaces de acesso ao hardware – é adotada pela *jRapture,* exceto pelo fato de que o "hardware", desta vez, é o próprio sistema operacional. A *jRapture* substitui todas as classes Java que interagem diretamente com o sistema operacional e de gerenciamento de janelas, tais como *FileInputStream*, *FileOutputStream, Socket, DatagramSocket* e *MComponentPeer*. O *jRapture* trata das questões relacionadas a *threads* de forma bastante curiosa, no entanto. Os autores começam supondo que os problemas ligados a condições de corrida foram previamente tratados (como fazem, aliás, os autores da TORNADO).

O problema com o *jRapture* é que a ferramenta precisa ser capaz de identificar quais objetos devem ser mapeados a quais referências. Para entender o que isso quer dizer, considere o seguinte trecho de código:

```
public void foo(){
     FileInputStream fis1 = new FileInputStream(
                                  new File("foo"));
     FileInputStream fis2 = new FileInputStream(
                                  new File("bar"));
}
```

Para garantir a reprodução correta, o *jRapture* tem que ser capaz de mapear ambas instâncias de *FileInputStream* nas porções corretas do arquivo de traços. Isso significa, em outras palavras, que o *jRapture* tem que ser capaz de encontrar uma correspondência não-ambígua entre as instâncias de *FileInputStream* geradas durante a gravação e durante a reprodução. Agora imagine que dois *threads* T_1 e T_2 chamem *foo* simultaneamente. A criação das instâncias por T_1 e T_2 podem se entrelaçar de 6 formas distintas. Isso significa que se o *jRapture* recriar as instâncias pressupondo que há uma ordem total, problemas podem acontecer caso o entrelaçamento seja, durante a reprodução da execução, diferente do entrelaçamento original. O *jRapture* trata desse problema levando em consideração que a execução dentro de cada *thread* é seqüencial e usando o próprio *thread* para identificar a instância. Isto é, o *jRapture* assegura que o *k-ésimo* objeto criado por T_i

durante a reprodução seja mapeado no trecho do arquivo de traços gerado pelo *k-ésimo* objeto criado por T_i durante a gravação.

Portanto, o problema de identificar objetos em execuções distintas fica reduzido ao problema de identificar *threads* em execuções distintas – algo que é relativamente simples de fazer, já que, conforme exploramos anteriormente (Figura 3-1), *fork* é uma operação de sincronização que induz uma ordenação nos *threads*. Um problema pouco discutido pelos autores do *jRapture* e que também se estende à TORNADO, no entanto, diz respeito a quando referências a um certo objeto (um descritor de arquivos no TORNADO, um *InputStream* no *jRapture*, ou até mesmo um objeto no contexto do núcleo do S.O.) são compartilhados por mais de um *thread*. Com relação ao *jRapture,* percebemos duas alternativas:

1. O sistema supõe que o acesso aos objetos é sincronizado por primitivas que estão no escopo de algum sistema de re-execução de *threads*. Neste caso, o sistema de re-execução de *threads* é o responsável por garantir que as chamadas serão feitas em uma ordem consistente (ou que condições de corrida serão detectadas). Esta hipótese não é realista.
2. O sistema garante uma espécie de "visão *thread-local*" dos objetos compartilhados. No caso de um *FileInputStream*, o sistema garante que os dados devolvidos a cada *thread* durante a reprodução são os mesmos devolvidos durante a gravação, ainda que isso represente uma "violação causal" em potencial. Um exemplo é mostrado nas Figuras 3-9 (a) e (b). A flecha pontilhada indica a relação causal implícita, que é violada em *FileInputStream* (*fis* na figura). No exemplo, a abordagem funciona bem porque os *threads* estão completamente isolados dessa violação causal. O problema parece mais difícil de resolver, no entanto, nos casos de chamadas que produzem efeitos colaterais externos (alteram parâmetros, por exemplo) e/ou que precisam ser reproduzidas numa ordem específica (chamadas a `System.out.println()`, por exemplo).

A hipótese 2, embora razoável para a *jRapture*, parece difícil de supor para a TORNADO, já que os autores não mencionam nada que nos leve nessa direção. Isso nos leva a crer que a TORNADO confia na hipótese 1. Dizemos que essa abordagem não é realista porque é possível que o trecho sincronizado de uma chamada de sistema seja feito dentro do núcleo. Nos parece muito difícil, portanto, delegar a responsabilidade por tratar ambientes *multithreaded* a uma ferramenta auxiliar, já que algumas condições de corrida gerais podem se desenvolver além de seu escopo.

Voltando para a *jRapture*, um outro ponto um tanto obscuro diz respeito à re-execução de eventos de interface (AWT). Os autores dizem anexar um *timestamp* a cada evento para que possam determinar, na re-execução, o instante correto de reproduzí-lo na fila AWT. Mas as variações que ocorrem na própria execução podem deslocar o instante "correto" de produção de cada evento. O

que importa, de fato, é a ordem das interações entre o *thread* de interface e os *threads* de processamento. Se essa ordem for respeitada, a re-execução é correta. Se não for, a re-execução é (potencialmente) incorreta. Isso porque o AWT é, estritamente falando, apenas mais um processo que se comunica com os outros.

Na Figura 3-9 (c), o evento 1 provoca uma interação do *thread* de eventos do AWT com T_1 e o evento 2, uma interação com T_2. Logo em seguida, T_1 e T_2 interagem. Note, no entanto, que seria perfeitamente possível construir um programa que admitisse diferentes ordens de interação de T_1 e T_2 com o *thread* do AWT, como na Figura 3-9 (d) – se T_1 e T_2 fizessem parte de um *pool* de *threads*, por exemplo. Nesse caso, seria perfeitamente possível que a execução, a partir daquele ponto, se desenrolasse de maneira diferente, levando a uma falha na reprodução (ilustrada pela ausência da interação entre T_1 e T_2 na Figura 3-9 (d)). Nesse caso, o *timestamp* do evento gravado pelo *jRapture* é completamente irrelevante, já que o efeito desse evento na execução é unicamente determinado pelas relações entre os *threads*. Se, por outro lado, T_1 e T_2 nunca pudessem interagir com o *thread* do AWT numa ordem diferente, então isso significaria que o programa é determinístico e, novamente, que o *timestamp* seria supérfluo.

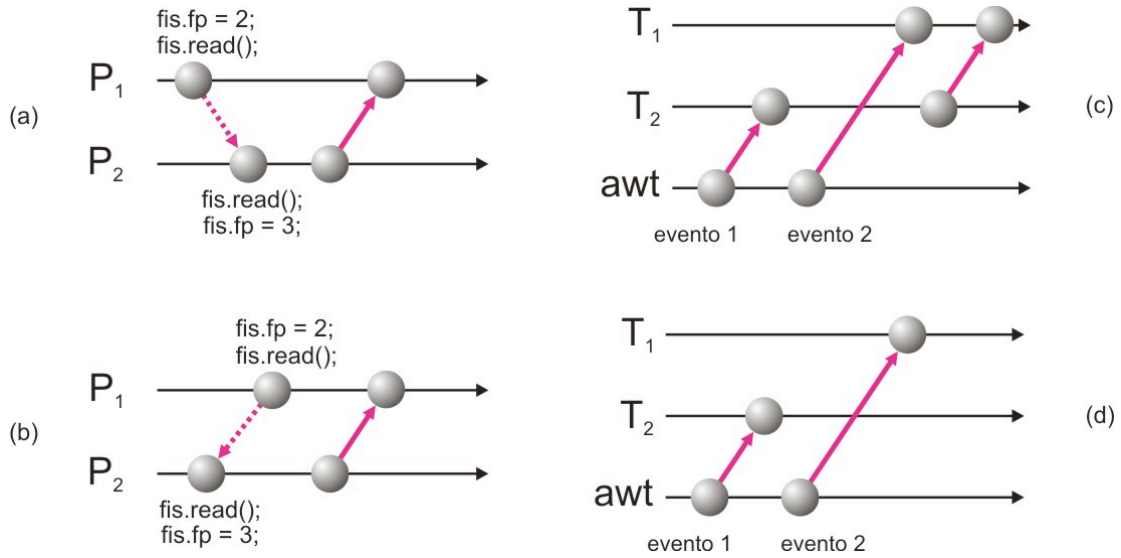

Figura 3-9. (a) Leitura de *stream* durante gravação e (b) reprodução. Entrega de eventos AWT durante (c) gravação e (d) re-execução.

Até onde podemos enxergar, reproduzir interações com uma interface orientada a eventos na presença de múltiplos *threads* é, na melhor das hipóteses, equivalente a reproduzir a interação de um sistema de passagem de mensagens – e isso é impossível de alcançar se levarmos em consideração apenas um dos processos (o *thread* do AWT).

Além dos sistemas de re-execução de entrada discutidos até agora, existe um sistema portador de características bastante particulares cuja classificação não é tão direta. Trata-se do *Flight Data Recorder* (FDR). Conforme mencionamos brevemente na Seção 3.2.3, o FDR captura, além da ordem de acesso a regiões de memória, todas as operações de armazenagem (*load/in*) emitidas a regiões de memória mapeadas em dispositivos de entrada e saída, bem como interrupções (a re-execução de interrupções será discutida na Seção 3.2.5). O comportamento dos dispositivos que usam DMA também é gravado – esses dispositivos são tratados como processadores virtuais e seus acessos à memória, encarados como os dos outros processadores. A diferença com relação aos processadores "regulares" é que o FDR grava, no caso de dispositivos de DMA, também os dados que acompanham cada operação de escrita, já que essa informação provém, em geral, de fora do sistema. A Figura 3-10 situa o FDR num sistema computacional hipotético.

Note que as provisões tomadas pelo FDR são suficientes para a reprodução o comportamento dos dispositivos de entrada e saída de maneira bastante fiel – e sem os problemas enfrentados pelas abordagens em *software*, como a TORNADO. De quebra, o FDR reproduz o comportamento do escalonamento de *threads* em sistemas pseudo-paralelos. Com relação à classificação do FDR, podemos considerar que o hardware é a faceta de mais baixo nível do ambiente de execução. Dado que o FDR se baseia numa técnica que pressupõe a instrumentação do hardware nos pontos de interface com o núcleo do sistema operacional, podemos considerar que a técnica é de nível de sistema.

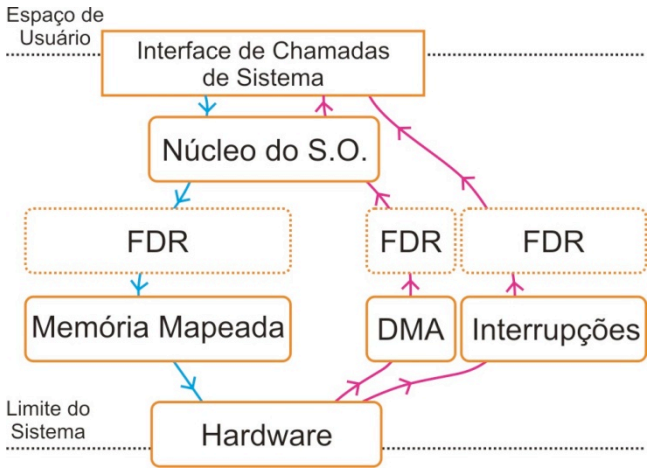

Figura 3-10. Inserção do *Flight Data Recorder* num sistema computacional hipotético. As linhas de fluxo indicam o sentido dos dados numa chamada de sistema típica.

3.2.5 Re-execução de interrupções

Além da interação com dispositivos de entrada e saída, uma outra fonte de comportamento não-determinístico bastante incômoda por estar, novamente, fora do escopo da aplicação, são as interrupções. Interrupções, no âmbito que nos é de interesse, dizem respeito a sinais assíncronos produzidos por software (outros processos) ou hardware e que alteram o fluxo de execução da aplicação.

Note que há uma sobreposição entre a re-execução de interrupções e a re-execução de entrada já que, em diversas situações, os dispositivos de entrada e saída valem-se de interrupções como mecanismo de notificação assíncrona. Controladoras de disco, por exemplo, tipicamente geram interrupções para assinalarem o fim de uma operação de leitura dados. Interrupções podem, num certo sentido, serem encaradas como mensagens. A diferença com relação às mensagens é que uma interrupção pode interromper a execução de uma aplicação num momento essencialmente aleatório, possivelmente afetando causalmente todas as instruções da aplicação que ocorrem a partir dali.

Há três questões fundamentais na reprodução de interrupções:

1. Determinar o momento correto de reproduzi-las;
2. reproduzi-las de fato;
3. reproduzi-las na ordem correta.

A primeira questão diz respeito a identificar, na seqüência de instruções do programa, qual o ponto em que a interrupção deve ser reproduzida. A segunda questão diz respeito a se as interrupções devem ser reproduzidas de verdade; isto é, se devemos supor que o hardware/software vai de fato produzir uma nova interrupção durante a reprodução, ou que a interrupção será simulada (apenas a rotina de tratamento da interrupção será reproduzida, possivelmente em conjunto com uma abordagem baseada em conteúdo). A terceira questão, finalmente, diz respeito a possíveis condições de corrida entre interrupções, como no caso de interrupções aninhadas.

As técnicas de reprodução de interrupções que estudamos [49, 113, 176, 177, 231] partem de uma abordagem proposta por Mellor-Crummey em 1989 [132] como uma extensão ao seu ambiente de depuração de sistemas concorrentes, o *Instant Replay* (discutido na Seção 3.2.3). Em sua essência, a abordagem consiste no uso de um contador de instruções [25] para demarcar o ponto da execução em cada processo em que a interrupção ocorre. Um contador de instruções não é nada além do que o que seu nome sugere: trata-se de um contador cujo valor se altera conforme são executadas instruções. Contadores de instruções tipicamente operam em dois modos – ascendente e descendente, onde o contador é, respectivamente, incrementado e decrementado a cada instrução executada. Contadores de instruções possuem ainda uma propriedade bastante particular, que é a de gerar uma exceção ao atingirem o valor 0. A hipótese implícita na reprodução de interrupções com

contadores de instruções é que o número de instruções executadas por um certo processo é suficiente para caracterizar qualquer estado na execução desse processo. Essa hipótese é válida, evidentemente, apenas no caso em que o restante das fontes de indeterminismo já foram controladas.

O uso de contadores de instruções para a reprodução determinística de interrupções é feito da seguinte forma. Durante a fase de gravação, as rotinas de tratamento de interrupções (ISR na Figura 3-11) são instrumentadas. Sempre que uma dessas rotinas for disparada (pela ocorrência de uma interrupção) o contador de instruções do processo afetado é armazenado juntamente com dados auxiliares, necessários para a reprodução daquela interrupção (Figura 3-11 (a)). Durante a re-execução, o contador de instruções será decrementado. O ponto de ocorrência da *i-ésima* interrupção é então determinado inicializando-se o contador de instruções com $ic_i - ic_{i-1}$, onde ic_i representa o valor do contador de instruções quando do disparo da *i-ésima* interrupção, durante a gravação. Ao atingir 0, o contador de instruções gera uma exceção, que sinaliza ao sistema de re-execução que uma interrupção precisa ser reproduzida. O sistema de re-execução executa então a ISR que corresponde à interrupção gravada no arquivo de traços (*log* na Figura 3-11 (b)), simulando a ocorrência da interrupção em questão.

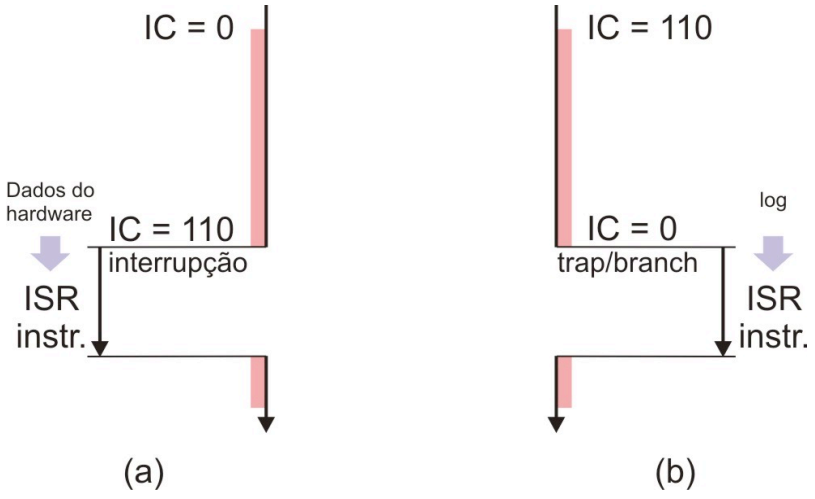

Figura 3-11. (a) Gravação e (b) reprodução de uma interrupção.

Um dos problemas com a contagem de instruções é que, conforme Mellor-Crummey já dizia em 1989, são poucos os processadores comercialmente disponíveis equipados com contadores de instruções em hardware (HICs). Embora contadores de instruções sejam dispositivos de hardware bastante simples [25], essa afirmação é válida até os dias de hoje. Uma parte substancial dos sistemas de re-execução de interrupções, portanto, optem por emular HICs com contadores de instruções em software (SICs). Com a exceção do FDR [128], todos os outros sistemas de

reprodução de interrupções estudados utilizam SICs. Evidentemente, um SIC não pode ser implementado instrumentando-se todas as instruções de um programa – o *overhead* de execução e espaço seria muito grande. Mellor-Crummey nota, no entanto, que isso não é necessário.

A idéia é que podemos dividir a execução de um programa numa série de blocos, onde cada bloco representa o destino de uma ação de transferência de controle de baixo nível (*jump, branch* ou chamada a sub-rotina). Se os blocos forem enumerados pelo SIC durante a execução, a tupla *(SIC, PC)*, onde *PC* corresponde ao contador de programa (*program counter*) do processador, é suficiente para identificar, com o mesmo grau de precisão de um HIC, qualquer ponto na execução do programa em questão. Mellor-Crummey nota ainda que não é necessário incrementar o SIC a cada início de bloco (isto é, não é necessário enumerar todos os blocos), já que alguns blocos podem executar em seqüência – isto seria o equivalente a "agrupar" os blocos que executam seqüencialmente em um único grande bloco. É preciso enumerar apenas aqueles blocos que se iniciam como resultado de um *branch* para trás (*backward-branch*) ou de uma chamada a uma sub-rotina. Mellor-Crummey mostra que é possível implementar um SIC de forma bastante eficiente por meio da reserva de um registrador genérico no processador e do uso de instruções do tipo *decrement-and-branch*, disponíveis tanto em modelos CISC quanto RISC.

A abordagem dos *Instruction Counters* de Mellor-Crummey foi implementada com poucas variações em alguns sistemas, dentre os quais podemos destacar o *Flight Data Recorder* [231], o *Repeatable Scheduling* [176, 177] e o HARTS [49]. Em 1994, Levrouw e Audanaert apontam que a hipótese de que reproduzir a execução de uma IRS equivale a reproduzir uma interrupção é falha [113] e propõem uma nova abordagem. A hipótese é falha, essencialmente, porque determinadas interrupções podem produzir efeitos colaterais no hardware e, de maneira mais geral, no ambiente que cerca o sistema. Esses efeitos colaterais são perdidos quando a rotina de tratamento da interrupção é meramente re-executada. Levrouw e Audanaert propõem uma abordagem que leva em consideração que certas interrupções simplesmente precisam ser reproduzidas. A abordagem de Levrouw e Audanaert:

1. Reproduz por chamadas à respectiva ISR todas as interrupções passíveis de reprodução por esse meio;
2. Delega a responsabilidade por reproduzir o restante das interrupções ao ambiente externo, mas garante, por meio de um mecanismo baseado em ordem, que o instante de processamento das interrupções recebidas será equivalente durante gravação e reprodução.

Em outras palavras, a abordagem de Levrouw e Audanaert reconhece interrupções de natureza não-simulável e implementa um mecanismo de reprodução baseado em ordem para tratá-las. Isso equivale, num certo sentido, a encarar interrupções como mensagens. Há, é claro, algumas

complicações com a reprodução de interrupções quando comparadas a mensagens "regulares" – entre elas está o fato que não seria realista pressupor que o ambiente externo será capaz de anexar *timestamps* lógicos a essas mensagens. Em particular, não seria realista pressupor que é possível instrumentar dispositivos de hardware para que compreendam tempo lógico. A implicação disso é que torna-se impossível distinguir entre duas interrupções emitidas na mesma linha (não há indícios de ordem), portanto o programa é forçado a supor que as interrupções externas em uma mesma linha serão emitidas, durante a re-execução, na mesma ordem da execução original. Os autores não mencionam isso no artigo, mas fica claro que essa limitação se aplica na medida em que o arquivo de traços gerado guarda, para cada interrupção, apenas o número da linha e o valor do SIC no instante em que o processador é interrompido.

Os autores fazem ainda algumas outras hipóteses simplificadoras, como a ausência de condições de corrida entre interrupções e processos interrompidos. Sob essa hipótese, torna-se possível implementar um contador de instruções em software mais eficiente. Para tanto, os autores levam em consideração que a execução em cada processador pode ser particionada em uma série de "janelas de interrupção" (*interrupt windows* ou IWs), que são os trechos em que o processamento de interrupções encontra-se ativado. Na ausência de condições de corrida de dados, a única interação possível entre uma ISR e o processo interrompido dentro de uma IW é por meio de condições de corrida gerais (via áreas de memória compartilhadas, devidamente protegidas). Essas condições de corrida são, conforme evidenciamos na Seção 3.2.3, relativamente fáceis de gravar e reproduzir. A Figura 3-12 ilustra de forma simplificada o funcionamento do mecanismo de gravação/reprodução de Levrouw e Audanaert, com as janelas de interrupção demarcadas e os principais eventos assinalados. *CSEnter* e *CSExit* correspondem, respectivamente, à aquisição e à soltura das travas que protegem uma região crítica compartilhada entre a ISR e o programa interrompido.

Durante a gravação, a técnica de Levrouw e Audanaert guarda os valores do SIC para todas as operações de ativação/desativação de interrupções, para as operações *CSEntry/CSExit* e para as instruções de chamada e retorno das ISRs (condensadas como *interrupts* na Figura 3-12). Na re-execução (Figura 3-12 (b)), as interrupções são inicialmente desabilitadas e as operações de ativação/desativação de interrupções (as operações *enable/disable* na Figura 3-12 (b)), substituídas por implementações vazias, que apenas atualizam o SIC. O sistema de re-execução prossegue examinando então valor do SIC e, a cada incremento, o arquivo de traços, seletivamente ativando as linhas de interrupção de maneira a forçar o processamento das interrupções a obedecer a ordem original. Note que as operações de sincronização que afetam as ISRs são levadas em consideração, como mostra a Figura 3-12 (c). Isso implica, num certo sentido, que a abordagem de Levrouw e

Audanaert escapa um pouco de seu escopo, visto que tratar de condições de corrida gerais é um problema diferente, em princípio, do problema de reproduzir interrupções.

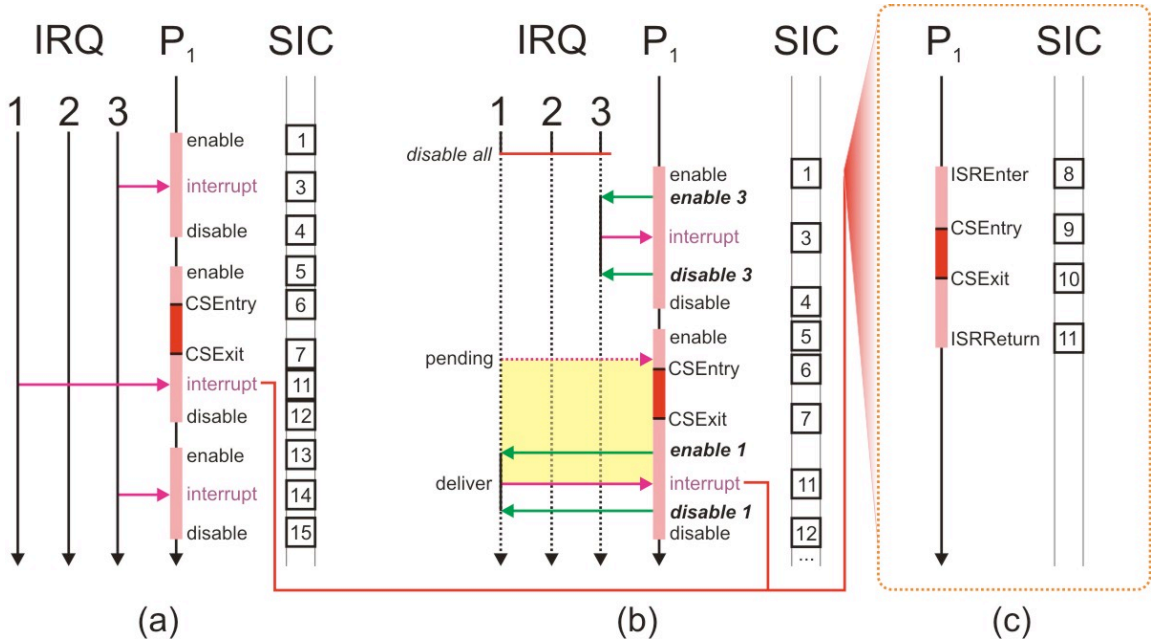

Figura 3-12. (a) Gravação e (b) reprodução das interrupções numa execução hipotética. (c) detalhamento do tratamento de uma interrupção.

O *overhead* de tempo e espaço na abordagem de Levrouw e Audanaert é potencialmente menor do que o da abordagem de Mellor-Crummey, já que há muito menos pontos de instrumentação. Os autores, no entanto, não provêm nenhum tipo de análise de desempenho que favoreça (nem contradiga) essas afirmações.

3.2.6 Mais sobre monitoramento e análise

Na Seção 3.1, apresentamos a estrutura geral de uma ferramenta de análise. Nesta seção, aprofundaremos a nossa discussão considerando as características que diferenciam as técnicas de análise e monitoramento umas das outras. Técnicas de análise são classificadas de acordo com diversos aspectos. Um dos aspectos mais recorrentes diz respeito à simultaneidade entre execução de ferramenta de análise e sistema-alvo. Isso induz um particionamento das técnicas em duas classes: *on-line* e *post-mortem*.

Consideram-se como *on-line* as técnicas de análise cuja aplicação é concomitante à execução do sistema – i.e., os dados fluem dos componentes de monitoramento para os componentes de análise (recorde-se da Figura 3-1 (c)). Técnicas *post-mortem*, por sua vez, são aplicadas após a execução. A diferença entre as duas modalidades é que as técnicas *post-mortem* contam, normalmente, com um nível maior de certeza no que diz respeito ao comportamento da execução do sistema. Se

observamos, após terminada a execução do sistema, um evento de envio sem um evento de recebimento correspondente, é seguro afirmar que a mensagem se perdeu [104]; em sistemas onde a participação é dinâmica, torna-se possível determinar um limite superior para o número de processos participantes simultâneos [173]; etc. O espaço de armazenagem requerido por uma técnica *post-mortem*, no entanto, pode ser muito grande. Além disso, o grau de flexibilidade provido por técnicas *post-mortem* é muito menor, já que a granulosidade e o tipo (no sentido mais genérico possível) dos dados coletados da execução de cada processo local precisam ser especificados de antemão. Com uma técnica *on-line*, por outro lado, o usuário pode explorar cada ponto da execução de acordo com as suas necessidades. Isso é especialmente interessante quando um sistema de re-execução é empregado em conjunto.

Durante o curso do nosso trabalho, nós nos deparamos com algumas limitações na caracterização *on-line/post-mortem*, evidenciadas pelas nossas dificuldades em classificar o nosso depurador como *on-line* [130]. Propomos, portanto, uma divisão das técnicas *on-line* em duas subcategorias, que denominamos *limitadas* e *não-limitadas*. Intuitivamente, uma técnica é não-limitada quando a sua correta operação não está condicionada à ocorrência "freqüente" de operações de sincronização de estado (operações que sincronizam parte do estado "real" do sistema distribuído com o estado "aproximado", visto pelo observador/depurador). Técnicas *limitadas*, por sua vez, têm a sua operação condicionada à ocorrência dessas operações de sincronização. Nós vamos caracterizar operações de sincronização de estado apenas informalmente.

Definição 3-4: Uma operação de sincronização de estado corresponde a um ponto na execução do sistema em que um processo local tem a sua execução suspensa para que informações arbitrárias, mas que dizem respeito ao seu estado, sejam transmitidas, de forma síncrona, ao observador.

Seja $P = \langle E, \rightarrow \rangle$ uma execução (possivelmente distribuída), e seja $\sum_P$ o conjunto dos tipos de eventos que ocorrem em P. Definimos $f_{sync}^{P} : \sum_p \times N \rightarrow N$ como a função que nos informa o número de operações de sincronização de estado produzidas em P de acordo com o número de ocorrências de cada tipo de evento.

Definição 3-5: Seja $h : N \rightarrow N$ uma função <u>estritamente</u> crescente. Dado F, o conjunto de todas as possíveis execuções de um sistema, uma técnica de depuração *on-line* é *limitada* se $\exists P \in F$ e $\exists a \in \sum_P$ tal que $f_{sync}^{P}(a,n) \subset \Omega(h(n))$.

Ou seja, uma técnica *on-line* é limitada se o número de operações de sincronização de estado cresce de acordo com uma função não-constante do número de ocorrências de eventos para algum tipo de evento *a*, em alguma execução *P* do sistema. De maneira semelhante:

Definição 3-6: Dado F, o conjunto de todas as possíveis execuções de um sistema distribuído, uma técnica de depuração *on-line* é *não-limitada* se para $\forall P \in F$ e $\forall a \in \sum_P$, tivermos $f_{sync}^{P} \subset O(1)$.

Depuradores baseados em técnicas *on-line* não-limitadas não dependem de sincronizações constantes. Isso quer dizer que esses depuradores, potencialmente, podem executar de forma completamente assíncrona com relação ao sistema distribuído – até mesmo após o término de sua execução (i.e., *post-mortem*). A dependência das sincronizações freqüentes exibidas pelos depuradores baseados em técnicas *on-line limitadas*, por outro lado, tornam o uso *post-mortem* inviável. Além disso, as técnicas *limitadas* exibem uma tendência à exacerbação do efeito do observador. No Capítulo 5, explicaremos que uma técnica limitada é equivalente a um algoritmo de *snapshotting* que suspende a execução do sistema para obter uma fotografia consistente, mas cujo custo de sincronização é pulverizado ao longo tempo.

Com relação às técnicas de monitoramento, consideram-se os seguintes critérios [83, 104, 138]:

- Ativação
  - Orientada a eventos
  - Temporal
- Implementação
  - Hardware
  - Software
  - Híbrida

A ativação diz respeito a quais mecanismos disparam uma observação. Na ativação temporal, as observações são realizadas em intervalos específicos, de acordo com uma taxa de amostragem pré-definida. Monitores de ativação temporal são adequados para medidas cuja significância é estatística como, por exemplo, o tempo que o sistema passa dentro de cada função. Essa medida, particularmente, poderia ser extraída por um monitor temporal que observa o valor do *program counter*. O problema com monitores temporais é que as informações capturadas representam recortes discretos da execução. Isso significa que, nos casos em que eventos relevantes acontecem em instantes imprevisíveis, esse tipo de monitoramento é inadequado. Um exemplo de evento que ocorre num instante imprevisível é o acesso a uma região de memória compartilhada.

Isso nos leva aos monitores orientados a eventos. Um monitor é orientado a eventos quando reage à ocorrência de um "evento". Segundo Rainer Klar [100], um evento é "uma ação atômica e instantânea". Embora essa definição não seja universal ou adequada para todos os propósitos, ela é adequada nesta seção. Um aspecto interessante de monitores orientados a eventos é que eventos são entidades inerentemente acopladas à semântica da aplicação. Isso nos leva à implementação dos monitores.

Monitores podem ser implementados em hardware, em software, ou ambos (híbridos). Um monitor de hardware tipicamente consiste num dispositivo, de hardware, capaz de capturar o tráfego num barramento ou o conteúdo de registradores. Monitores de hardware produzem pouca ou nenhuma interferência no sistema observado mas, por outro lado, trabalham num nível de abstração muito baixo. Identificar eventos como, por exemplo, a aquisição de um *mutex*, pode ser difícil – é necessário mapear, de forma não-ambígua, um conjunto potencialmente grande de sinais, capturados pelo monitor, em estruturas de nível mais alto [100].

Monitores de software, por outro lado, encaixam-se muito melhor por serem entidades de nível de aplicação. Um monitor de software é implementado pela inserção de instruções de monitoramento em pontos estratégicos do código da aplicação observada. A principal vantagem de monitores de software é que podemos implementá-los no mesmo nível semântico da aplicação com, por exemplo, técnicas de instrumentação de código-fonte ou pela instrumentação de bibliotecas. Um monitor de hardware não sabe o que é código-fonte ou como interceptar chamadas a uma biblioteca. O problema com monitores de software é que a sua execução compartilha recursos (CPU, memória, banda de E/S) com a aplicação observada, algo que resulta na deterioração do desempenho da aplicação e, pior, no efeito do observador.

Klar propõe, como forma de combinar as vantagens de monitores de software às vantagens de monitores de hardware, um terceiro tipo de monitor – o monitor híbrido. Monitores híbridos também são implementados pela instrumentação seletiva do código da aplicação monitorada. A diferença é que enquanto com monitores de software os eventos são armazenados em *buffers* locais (em memória) e periodicamente gravados em disco, com monitores híbridos os eventos são diretamente gravados numa interface de hardware [83]. Mohr [139] diz que, em monitores híbridos, o reconhecimento dos eventos é feito em software, mas o *timestamping* e a gravação são feitos em hardware.

Um monitor específico para CORBA (no ORB Jonathan [53]) e Java é descrito por Duchien e Seinturier [52]. Esse monitor é, essencialmente, capaz de capturar a ocorrência e a ordem de execução de diversos tipos de eventos, dentre eles eventos de sincronização, acessos a variáveis compartilhadas e mensagens.

3.2.7 Análise e visualização

As técnicas de reprodução reversa e de reprodução de condições de corrida, de entrada e de interrupções discutidas nas seções anteriores tratam de tornar determinísticas execuções não-determinísticas, abrindo espaço para a inspeção de estados arbitrários em sistemas concorrentes (considerando o processamento de interrupções, aqui, como uma forma de concorrência). Note no entanto que o objetivo final desses mecanismos é, em última instância, o de tornar a *análise* dessas execuções concorrentes mais simples. O ônus da análise, por outro lado, ainda recai sobre o usuário. Embora seja incorreto afirmar que as ferramentas de re-execução são disjuntas do reino da análise automática (vide os mecanismos de detecção automática de condições de corrida apresentados por algumas delas), é correto afirmar que o foco principal dessas ferramentas não é esse.

As técnicas de visualização e análise automática vêm justamente para preencher essa lacuna. Sob essa perspectiva, podemos diferenciar entre dois grupos não-disjuntos de técnicas de apoio à análise, que são, justamente as técnicas de visualização e de análise automática.

Técnicas de visualização visam auxiliar na produção de imagens mentais que levem a uma melhor compreensão do comportamento de um sistema de software. Uma discussão mais detalhada da acepção do termo em contextos diversos, bem como uma apresentação mais ampla da área de visualização de software, pode ser encontrada nas taxonomias de Price, Baker e Small [163, 164], na taxonomia de Roman [171], na de Stasko [198] e, mais recentemente, nos trabalhos de Maletic et al. [126] e de Gracanin et al. [75], bem como nos trabalhos de Pancake [156, 159] e Kranzlmüller [104]. Técnicas de análise automática, por sua vez, são tipicamente compostas por algoritmos que varrem, em busca de certas propriedades, os dados coletados durante uma execução. Esses achados são então reportados ao usuário. Exemplos relevantes de técnicas de análise automática em sistemas concorrentes incluem os algoritmos de detecção de predicados globais [26, 137, 186, 216, 221], as técnicas de busca de padrões em grafos de eventos de Kranzlmüller [104, 105] e os detectores de condições de corridas, discutidos na Seção 3.2.3.

Um desses algoritmos de detecção automática de predicados foi, inclusive, implementado por Otta [155] como parte de uma ferramenta de depuração específica para sistemas de objetos distribuídos (baseados em CORBA). A ferramenta de Otta é capaz de detectar a validade de predicados instáveis conjuntivos ou disjuntivos, fracos ou fortes [137], cujos predicados locais são formulados sobre o estado de objetos Java/CORBA individuais.

Note que a "compreensão do comportamento de um sistema de software", auxiliada pelas técnicas de visualização, é uma atividade inerentemente humana (i.e., não-automática). Seria possível afirmar, portanto, que a atuação das técnicas de visualização se faz necessária na medida em que se esgotam as possibilidades oferecidas pelas técnicas de análise automática. Kranzlmüller

[104] afirma que um depurador deve procurar automatizar ao máximo todas aquelas tarefas que são passíveis de automação. Isso implica que devemos sempre procurar explorar ao máximo, em cada domínio, as possibilidades de análise automática. É nossa crença que esse princípio é deveras sábio, devendo ser levado em consideração no projeto de qualquer ferramenta de desenvolvimento, não só de ferramentas de depuração. O princípio pode parecer óbvio, mas Pancake [156] relata que um dos maiores obstáculos enfrentados pelos usuários de ferramentas de análise de desempenho para sistemas paralelos é que seus mecanismos de análise automática de dados são primitivos demais.

Apesar de estarmos apresentando visualização e análise automática como entidades disjuntas (e complementares), há uma conexão intrínseca entre esses dois tipos de técnicas. Isso porque os estímulos sensoriais (normalmente imagens e animações e, menos comumente, sons) produzidos por ferramentas de visualização são, quase sempre, fruto de modelos construídos a partir da análise automática de informações de execução. De maneira semelhante, podemos argumentar que uma porção substancial das técnicas de análise automática têm como objetivo a produção de algum tipo de visualização.

Embora a idéia de análise automática, em sua forma mais pura, seja um conceito bastante atraente[6], uma limitação fundamental desse tipo de técnica é que os parâmetros de corretude para um sistema de software são, inevitavelmente, baseados em alguma espécie de especificação (recorde-se da Definição 1-4). O formato e a precisão dessas especificações podem variar consideravelmente: podemos usar desde descrições rígidas (por exemplo, um conjunto de predicados que descrevem estados errôneos para a aplicação) até mecanismos de aprendizado computacional, com uma classificação probabilística de comportamentos errôneos. De qualquer forma, algum tipo de especificação que defina "comportamento correto" é sempre necessário.

Conforme mencionamos na Seção 1.2.2, no entanto, produzir especificações completas e que cubram todos os aspectos de um sistema arbitrário é uma tarefa muito difícil. Essa limitação fundamental das técnicas de análise automática indica que o uso da análise manual é inevitável – e é aí que as técnicas de visualização começam a adquirir a sua importância. Para entendermos as questões envolvidas em melhorar o apoio oferecido por ferramentas à análise manual, é bastante útil que tentemos entender quais atividades envolvem visualização e que tipos de visualização seriam interessantes no contexto dessas atividades. Pancake [156] identifica um conjunto de objetivos que se estabelecem durante a identificação e o reparo de um problema de desempenho num sistema paralelo. Nós adaptamos a lista para defeitos de software, já que as questões envolvidas são essencialmente as mesmas:

[6] Imagine um sistema de software capaz de identificar a causa de um comportamento errôneo de forma automática e, quem sabe, até mesmo de sugerir uma correção.

1. *Identificação*: Há um defeito? Quais são os seus sintomas?
2. *Localização:* Em qual ponto da execução os sintomas se manifestam? Qual a causa desses sintomas?
3. *Reparo:* Quais partes da aplicação precisam ser modificadas para que o defeito seja sanado? [Desenvolvedor repara a aplicação.]
4. *Verificação:* O reparo resolveu o defeito? [Se não resolveu, o desenvolvedor opcionalmente desfaz o reparo e volta ao passo 2.]
5. *Validação:* Ainda existem sintomas de defeitos? [Caso existam, desenvolvedor volta ao passo 1.]

Pancake argumenta que a satisfação desses objetivos depende de três tarefas básicas, dadas a seguir:

1. **Estabilização:** Reconhecimento (identificação) do(s) sintoma(s) de um comportamento errôneo. Ocorre como parte da *Identificação, Verificação* e *Validação.*
2. **Redução do espaço de busca:** Exame de informações de execução, em busca de sinais que denunciem uma violação de premissa capaz de produzir os sintomas observados. É a atividade principal da *Localização* e, talvez, uma das tarefas mais difíceis.
3. **Modificação seletiva:** Processo em que o desenvolvedor inspeciona e modifica o código da aplicação, visando livrá-la do comportamento errôneo observado (ocorre como parte do processo de efetuar o *Reparo* da aplicação). Após inspecionar e modificar o código, o desenvolvedor *valida* sua hipótese, re-executando a aplicação e verificando se o defeito persiste. Em caso positivo, o desenvolvedor desfaz suas modificações e reinicia o ciclo.

Ainda segundo Pancake, o apoio a essas tarefas induz três necessidades. Para estabilização e redução do espaço de busca, são necessários mecanismos que permitam a exploração eficaz da execução em sua íntegra. Acrescentamos que essa exploração deve, idealmente, permitir que o desenvolvedor identifique pontos suspeitos de forma rápida. Note que, do ponto de vista da usabilidade, o desenvolvimento de mecanismos que permitam ao desenvolvedor navegar de forma eficiente pela representação abstrata da execução é tão importante quanto o desenvolvimento da própria representação.

Outro requisito da redução do espaço de busca são meios para a comparação de diversos aspectos e visualizações do comportamento do programa. Um conjunto mais amplo de visualizações pode ajudar na construção de hipóteses mais precisas e no descarte de hipóteses incorretas. Isso está em acordo com a visão de Maletic [126] (e com a nossa), que argumenta que a busca por uma visualização que cubra todos os aspectos de uma execução é infrutífera – são necessários meios diferentes para informações diferentes, bem como formas de correlacionar essas

informações de forma eficaz. Por último, para facilitar a modificação seletiva, a ferramenta deve permitir que o usuário navegue, de forma eficiente, por estruturas complexas de código-fonte.

Baseada nessas necessidades, Pancake propõe um conjunto de sugestões que poderiam melhorar a usabilidade de futuras ferramentas de depuração. No escopo de nossa discussão, duas dessas sugestões são particularmente importantes:

1. Associação de *zoom* e filtragem: Pancake sugere que as representações visuais devem aderir a um estilo *drill-down*; isto é, o sistema de visualização deve apresentar, inicialmente, uma representação gráfica compacta e despida de detalhes de baixo nível (*filtragem*), que pode ser "escavada" (*zoomed in*), sob demanda, pelo usuário. A cada nível de escavação (aproximação), são revelados mais detalhes. Idealmente, essa representação compacta deve guiar o usuário a encontrar o que deseja, utilizando-se do fato de que seres humanos podem identificar padrões com bastante eficácia. Essa característica é tão aceita como desejável na área de visualização de programas que foi formatada, por Bill Schneiderman [181] como um "mantra": "*Overview first, zoom and filter, then details on demand*".

Essa filtragem seletiva de informações de baixo nível é também chamada de elisão (*elision*) por Gracanin [75], que a identifica como ingrediente essencial em esquemas de visualização com aspirações a escaláveis. Gracanin argumenta ainda que a elisão de informações é mais fácil de alcançar com representações tridimensionais e ambientes de realidade virtual, na medida em que os mecanismos navegacionais típicos dessas representações (deslocamento da câmera e rotação) produzem oclusões e distanciamentos que ocultam informações desnecessárias e facilitam o foco de atenção. Um exemplo é dado pelo ImsoVision [125], uma ferramenta para a visualização de aspectos estáticos de sistemas C++. O ImsoVision utiliza uma representação tridimensional, onde classes são mapeadas em plataformas (recortes retangulares e planos). Nessa representação, os atributos privados ficam escondidos "na parte de baixo" de cada plataforma. Esses atributos tornam-se visíveis apenas quando o usuário rotaciona o plano.

Outros exemplos de sistemas que produzem representações de informações de execução em múltiplos níveis de abstração – normalmente apresentando a informação de forma hierárquica – são o SPV [93], desenvolvido pela Intel para os sistemas Paragon, o IPS-2 [136] e o LCB [141].

2. Apoio ao foco seletivo de atenção: Foco seletivo de atenção diz respeito à capacidade de uma técnica de navegação em prover o acesso detalhado a informações presentes em porções restritas da visualização. Conforme mencionamos anteriormente, representações gráficas e tridimensionais facilitam o foco seletivo. Quando o usuário aproxima a câmera de um determinado objeto que julga interessante, o restante dos objetos são naturalmente empurrados para longe. Outras formas de navegação que facilitam o foco seletivo de atenção incluem a "visão de olho de

peixe" (*fish-eye views*) de Muddarangegowda e Pancake [141] e o mecanismo de agrupamento de processos de Kunz [108], explorado por Kranzlmüller na redução do excesso de informações visuais apresentadas em grafos de eventos [104].

Além das questões colocadas por Pancake, é interessante examinar algumas das considerações de Gracanin com relação à escolha das representações adotadas para visualização. Gracanin aponta que toda visualização, seja qual for a sua natureza, representa uma metáfora do sistema sob exame. Essa metáfora deve viabilizar o grau de expressividade adequado, em acordo com o escopo do mecanismo de visualização (a definição do escopo, conforme Price et al [164], consiste em isolar aquelas características do sistema que a visualização deve tratar). Além disso, as seguintes características são primordiais:

1. **Consistência:** A metáfora ou mapeamento de artefatos de software em representações deve ser consistente. Múltiplos artefatos de software não devem ser mapeados na mesma metáfora. De maneira semelhante, um artefato de software não pode ser mapeado em mais de uma metáfora.
2. **Riqueza semântica e complexidade:** A metáfora escolhida deve ser rica o suficiente para incluir todos os aspectos do software que precisam ser visualizados. O sistema de visualização não deve, por outro lado, divergir a atenção do usuário das informações que deve transmitir.

3.3 Heterogeneidade – a grande vilã

No decorrer da Seção 3.2, apresentamos um conjunto bastante expressivo de representantes das técnicas e metodologias para depuração de sistemas paralelos e distribuídos. É particularmente interessante notar que as soluções apresentadas cobrem uma porção substancial dos problemas intrínsecos às execuções distribuídas (que foram discutidos no Capítulo 2). Embora algumas dessas soluções tenham efeito apenas paliativa – o problema da re-execução de entrada, por exemplo, não tem uma solução plenamente satisfatória – muitas das idéias discutidas poderiam trazer um benefício mensurável aos desenvolvedores de sistemas distribuídos.

As ferramentas em uso pela esmagadora maioria dos desenvolvedores está, no entanto, deveras longe de incorporar mesmo uma ínfima parte dessas técnicas, algumas das quais em existência há quase 20 anos. Muito embora uma porção substancial delas tenham sido incorporadas a ao menos alguma ferramenta em algum instante, uma boa parte dessas ferramentas tiveram como foco plataformas de pouca penetração, ou que já caíram em desuso (incluindo aqui as ferramentas que dependem de hardware específico).

Ainda que descartemos os casos das ferramentas de depuração que já nasceram mortas (aquelas que nunca evoluíram para além de protótipos, ou nunca foram compatíveis com plataformas de alguma penetração) a heterogeneidade continua um fator implacável. A Figura 3-13 ilustra o "aspecto bidimensional" da heterogeneidade, responsável pela sua ação característica. O tempo entra como uma dimensão porque, no universo da depuração de sistemas paralelos e distribuídos, garantir a compatibilidade com todas as plataformas num dado instante não é suficiente. Na Figura 3-13, as ferramentas f_1 e f_2 são compatíveis com todas as plataformas de seu tempo. No entanto, f_1 é incompatível com todas as plataformas com as quais f_2 é compatível (e vice-versa). A falta de apoio para ferramentas de depuração por parte das plataformas induz o desenvolvimento de abordagens *ad hoc* que garantem uma morte rápida às ferramentas – os modelos, pontos de instrumentação, sistemas operacionais e hardware estão em constante mudança.

O resultado é essa impressão, justificável, de que a área simplesmente não avança. Nós definitivamente não somos os primeiros a identificar uma tendência à estagnação e desperdício de esforços em decorrência da falta de uma padronização adequada. May [128] já havia feito isso em 1993, assim como Hondroudakis [83] em 1995, Pancake [157, 158] em 1996, Francioni [67] em 1998, e Lourenço [119] em 2003; entre outros. Uma informação importante sobre os pesquisadores citados – os que estão preocupados com a falta de avanços – é que todos são pesquisadores da área de sistemas paralelos. De fato, na área de computação de alto desempenho, essa preocupação rendeu alguns esforços importantes, tais como o *High Performance Debugging Forum* [67], o *Parallel Tools Consortium* [157] e o Projeto OMIS [120]. Com sistemas distribuídos no entanto – em particular com sistemas de objetos distribuídos – muito pouco foi feito nesse sentido.

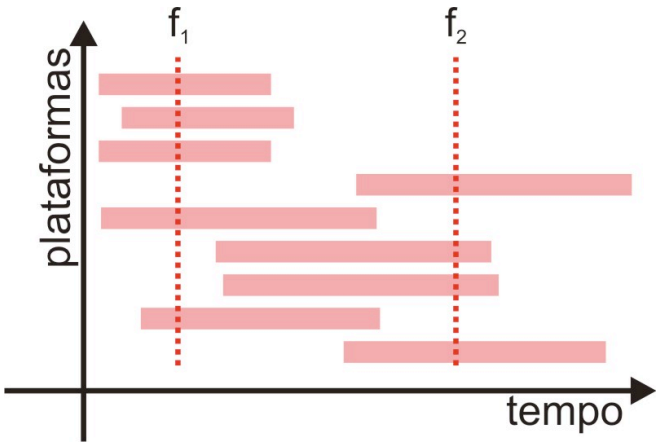

Figura 3-13. As duas dimensões da heterogeneidade.

Além disso, uma proporção muito pequena das ferramentas estudadas têm a sua implementação publicamente disponível. Isso nos leva a acreditar que uma boa parte delas não passam de

protótipos de viabilidade prática questionável. A Tabela 3-1 sumariza os dados que nos levam a essas conclusões. A tabela é composta por quatro colunas, cuja semântica é apresentada a seguir:

- **Ferramenta:** nome da ferramenta, com a citação da(s) fonte(s) tomada(s) como referência(s) para análise.
- **Linguagem do Sistema-Alvo:** Especifica a linguagem na qual são escritos os sistemas que podem ser depurados com o auxílio da ferramenta. A linguagem do sistema-alvo é determinada por uma de três formas: 1) O autor da ferramenta especifica, na própria descrição de sua abordagem, a linguagem do sistema-alvo. 2) O autor omite essa informação, mas deixa claro que sua ferramenta depende de propriedades específicas de determinadas linguagens – o PPD, por exemplo, depende de uma análise estática do código-fonte que só funciona para programas escritos em C. 3) A abordagem não depende da linguagem do sistema alvo. Nesse caso, o valor nesta coluna será "Independente". Esse caso é típico de ferramentas que trabalham num nível de abstração mais baixo do que o das linguagens, como nos sistemas de re-execução implementados em hardware ou no nível da máquina virtual.
- **Plataforma de software:** Diz respeito a sistemas de software dos quais dependem a operação da ferramenta. A plataforma de software, na Tabela 3-1, compreende sistema operacional, middleware, pacotes de *threads* específicos e outros sistemas de software ou bibliotecas (quando relevantes).
- **Hardware:** Apresenta os requisitos de hardware particulares da abordagem ou, no caso de abordagens que não demandam nenhum hardware em específico, o hardware para o qual existem versões da ferramenta.

Note que tanto a linguagem do sistema-alvo, quanto a plataforma de software, quanto o hardware, não são necessariamente limitações intrínsecas às técnicas empregadas pelas ferramentas. Em muitos casos, as informações contidas nessas colunas refletem aquilo que foi explicitamente citado pelos autores das ferramentas. Há ainda o caso de informações que não foram providas pelo autor e que não podem ser inferidas de forma direta, ou que podem ser inferidas, mas com alguma dúvida. Essas informações estão devidamente marcadas por pontos de interrogação (?).

Tabela 3-1. Sumário das ferramentas de depuração analisadas.

Ferramenta	Linguagem do Sistema-Alvo	Plataforma de software	Hardware
EXDAMS [14]	Independente	MULTICS	?
ODB* [115]	Java	Java *Runtime* 1.2+	Qualquer um para o qual haja uma versão compatível com o SDK 1.2 da *Sun*

Ferramenta	Linguagem do Sistema-Alvo	Plataforma de software	Hardware
IGOR [62]	C	*DUNE Distributed Operating System*	?
PPD [31]	C	?	*Sequent Symmetry Shared-Memory Multiprocessors*
Instant Replay [111]	Independente (?)	S.O. *Chrysalis*	*BBN Butterfly*
RecPlay [173]	C	Solaris/POSIX Threads	UltraSparc V10
DIOTA* [123]	C	Linux/POSIX Threads	Processadores compatíveis com a arquitetura Intel IA-32
JaRec [71]	Independente	Java *Runtime* (autores usaram 1.2.2, versão de compatibilidade desconhecida)	Qualquer um para o qual haja uma versão compatível com o SDK 1.2.2 da *Sun*
TRaDe [34]	Java	Versão modificada da Máquina Virtual da *Sun*, Versão 1.2.1	Qualquer hardware capaz de executar a versão 1.2.1 da Máquina Virtual da *Sun*
ReEnact [166]	Independente (?)	Indepedente (?)	Versões adaptadas de processadores equipados com *Thread-Level Speculation*
DejaVu [33, 103]	Independente	Versão específica e modificada da *Jikes RVM*	Qualquer um que execute a *Jikes RVM*
Optimal Shared-Memory Replay [144]	Independente (?)	?	*Sequent Symmetry Shared-Memory Multiprocessors* ou, minimamente, um multiprocessador que garanta consistência seqüencial
JVM Independent Replay [184]	Java	Java *Runtime*	Qualquer um que execute alguma máquina virtual Java
Flight Data Recorder [231]	Independente	Independente	Hardware de monitoramento especializado: FDR1; processadores com *cachê* de diretório e consistência seqüencial.
BugNet [142]	Independente	?	Hardware de monitoramento especializado: BugNet
FlashBack [196]	Independente (?)	Linux, com versão modificada do núcleo 2.4.22/POSIX *Threads*	Qualquer um que rode o núcleo 2.4.22 do Linux
MAD* [104]	C	Middleware MPI	?

Ferramenta	Linguagem do Sistema-Alvo	Plataforma de software	Hardware
Optimal Message Replay [148]	?	?	ThinkingMachines CM-5, Intel iPSC/2
DPD [189]	C (?)	Middleware REM [188]	?
DEIPA [118]	C	Middleware PVM	?
PDT [37]	C/HPF (High Performance Fortran)	Middleware MPI/Middleware *Annai*	NEC Cenju-3/Intel Paragon/Cray T3D
IVD [122]	C	Middleware PVM	?
Buster [229]	C (?)	Middleware MPI/PVM	Qualquer um para o qual haja uma implementação do GDB.
Panorama [128]	Independente	Middleware de passagem de mensagens do Intel iPSC/860 ou nCube.	Intel iPSC/860 e nCube, mas os autores clamam que seria simples adaptá-lo a novas arquiteturas.
CHORUS *Debugger* [81]	Independente (?)	CHORUS *Distributed Operating System*	?
CAPBAK/X* [195]	C, ou outra linguagem com *bindings* para *widgets* nativos (ex. Java + SWT)	Solaris, DEC OSF1, HPUX, AIX, Irix, SCO ODT + *X Windowing System* + Motif ou GTK.	Intel x86, SPARC, Alpha, RS6000 (IBM POWER), HP 9000, MIPS
*JavaStar** [133]	Java	Algumas versões do *Sun Microsystems* Java SDK	Qualquer um que rode essas versões do SDK da Sun
Selenium IDE* [153]	HTML	*Mozilla Firefox*	Qualquer um que rode o Mozilla Firefox
TORNADO [43]	Independente	Linux com núcleo modificado	Qualquer um que rode Linux
DDBG [118]	C	PVM	?
Fiddle [119]	Independente	Implementações para MPI e PVM, mas a arquitetura é extensível	?
Repeatable Scheduling [176, 177]	Independente	Mach OS 3.0	Processador x86, *Single Core*
HARTS *Debugger* [49]	C, mas possivelmente outras linguagens poderiam ser usadas.	S.O. HARTOS/Bibliotecas do HARTS	?

* Ferramenta com implementação disponível (código-fonte ou binário).

Tabela 3-1. Sumário das ferramentas de depuração analisadas.

A Tabela **3-1** ilustra a situação à qual nos referimos anteriormente. Há uma série de ferramentas, entre as 33 listadas, que poderiam auxiliar de forma bastante significativa no processo

de compreensão do comportamento de um sistema de objetos distribuídos. A questão é que muitas dessas ferramentas foram projetadas para ambientes experimentais que nunca foram liberados ao público; outras, dependem de hardware especializado; outras, foram projetadas tendo em vista a comunidade de sistemas paralelos; e outras, ainda, tornaram-se obsoletas junto com o ambiente para o qual foram projetadas, caindo em desuso junto com eles.

Além disso, apenas uma pequena parte dessas ferramentas – seis delas – encontram-se de fato disponíveis. Dessas seis ferramentas:

- 3 são ferramentas para re-execução de entrada (*JavaStar*, *Selenium* e CAPBAK/X) e resolvem problemas bastante pontuais no ciclo de desenvolvimento;
- 1 é específica para sistemas paralelos (MAD) e não poderia ser adaptada de forma direta a sistemas de objetos distribuídos;
- 1 é específica para sistemas concorrentes de memória compartilhada (DIOTA) baseados em Linux e POSIX *threads*. Embora se trate uma ferramenta bastante poderosa (e estável), não poderíamos adaptá-la com facilidade a um ambiente Java, por exemplo;
- 1 é uma ferramenta de monitoramento que não foi projetada para sistemas distribuídos (ODB).

3.4 Sumário

Ferramentas de depuração para sistemas distribuídos (e paralelos) são inerentemente mais complexas do que aquelas projetadas para sistemas centralizados e seqüenciais, por terem que lidar com uma série de questões que surgem apenas no contexto do primeiro tipo de sistema. A grande diversidade de plataformas (heterogeneidade), comum em sistemas distribuídos, também contribui com o aumento da complexidade das ferramentas, na medida em que exige arquiteturas e implementações mais rebuscadas. Uma ferramenta de depuração distribuída típica apresenta um formato arquitetural semelhante ao de uma ferramenta de monitoramento e análise: a diferença fica por conta do fato de que ferramentas de depuração podem ser interativas.

O processo de depuração consiste numa busca pelas causas de um comportamento errôneo, detectado previamente em uma sessão de testes (considerando aqui que o uso de um sistema em produção é também uma forma de teste). Essa "busca pela causa de um comportamento errôneo" implica que o processo consiste, na verdade, em tentar visualizar a execução do sistema em reverso. O problema é que execuções não são naturalmente reversíveis. Essa questão levou ao desenvolvimento, ao longo do tempo, de uma série de técnicas e ferramentas especificamente concebidas para viabilizar essa reversão ou, minimamente, uma visualização em reverso de

informações previamente capturadas. Neste capítulo, essas técnicas foram divididas em dois grupos – as técnicas de execução reversa e as de re-execução (determinística).

A característica marcante das técnicas de execução reversa é, naturalmente, a possibilidade de inspecionar em reverso parte (ou uma visão parcial) da seqüência de estados do programa. Técnicas de re-execução determinística, por sua vez, preocupam-se com rastrearem informações em quantidade suficiente para que uma dada execução do programa possa ser reproduzida. O intuito é o mesmo das técnicas de execução reversa – viabilizar a inspeção de estados arbitrários – mas, com a re-execução determinística, o usuário não é capaz voltar no tempo com tanta facilidade.

A modalidade de re-execução mais pesquisada na literatura sobre depuração de sistemas paralelos e distribuídos é a reprodução de condições de corrida. Por serem fracamente não-determinísticas, a abordagem natural (e mais usada) para a sua reprodução é a baseada em ordem. A primeira ferramenta nessa linha foi o *Instant Replay* (1988), de Mellor-Crummey. O *Instant Replay* reproduz o acesso a objetos compartilhados de alta granulosidade. Em sistemas de memória compartilhada típicos, no entanto, essa granulosidade é muito mais fina (*threads* podem compartilhar o acesso a um único *byte*), tornando o rastreamento e a reprodução muito custosos, tanto pelo volume dos arquivos de traços quanto pela exacerbação do efeito do observador.

Duas saídas para o problema são exploradas com mais sucesso – o monitoramento em hardware e a reprodução de operações de sincronização (com monitoramento em software). O monitoramento em hardware captura informações com perturbação mínima, mas gera uma quantia muito grande de traços e depende do uso de hardware especializado. A reprodução de operações de sincronização, por outro lado, não dá conta das condições de corrida de dados. Por essa razão, são reproduzidas as operações de sincronização apenas até a primeira condição de corrida.

Além da reprodução de condições de corrida, foram desenvolvidas técnicas também para a reprodução dos dados de entrada (instruções fortemente não-determinísticas) e de interrupções. Observamos que as ferramentas de reprodução de entrada podem ser, grosso modo, divididas entre ferramentas de nível de interface e de nível de sistema. Ferramentas de nível de sistema atuam na interface entre aplicação e sistema de execução subjacente. O "sistema de execução subjacente" pode tanto ser uma máquina virtual, quanto o núcleo do sistema operacional, quanto o próprio hardware. Ferramentas de nível de interface, por outro lado, são elaborações mais *ad hoc*, voltadas à reprodução de determinados tipos de comportamentos vinculados a entrada e saída em plataformas específicas.

A reprodução de interrupções é um assunto menos explorado na literatura. A abordagem típica consiste na reprodução das rotinas de tratamento de interrupções (*Interrupt Service Routines* ou ISRs) num momento equivalente ao original, junto com uma abordagem baseada em conteúdo para

suprir à ISR os valores produzidos pelo hardware, que não são reproduzidos. Levrouw e Audanaert são os únicos a levarem em conta que certas interrupções precisam ser reproduzidas pelo hardware, mesmo durante a re-execução. A técnica de Levrouw e Audanaert emprega uma abordagem baseada em ordem para garantir que os efeitos do tratamento das interrupções geradas durante a re-execução serão equivalentes aos que observados durante a gravação.

Ferramentas de depuração empregam uma série de mecanismos para captura de informações. Esses mecanismos podem ser implementados em software, hardware, ou como uma mistura dos dois (híbridos). Algumas técnicas de depuração só podem operar se todas as informações a respeito da execução estiverem disponíveis. Essas técnicas são chamadas *post-mortem* e se opõem às técnicas *on-line*, que podem operar concomitantemente à execução do sistema. Nós identificamos que as técnicas *on-line* podem ser ainda classificadas de acordo com o quanto permitem que a execução do sistema se desvie da aproximação observada. As técnicas cuja correta operação está condicionada a uma quantia limitada de desvio foram chamadas de *limitadas*, ao passo que aquelas que não impõem um limite a esse desvio, *ilimitadas.*

Embora a possibilidade de inspecionarmos estados arbitrários numa dada execução seja uma grande ajuda ao desenvolvedor, isso meramente coloca os sistemas concorrentes, com relação a esse quesito, num patamar semelhante ao dos sistemas seqüenciais. A questão de como auxiliar na análise dessas informações capturadas é, naturalmente, o papel das técnicas de análise. Dividimos as técnicas de análise entre análise automática e visualização (apoio à análise manual). Visualização e análise automática estão intimamente ligadas. A visualização entra na medida em que se esgotam as possibilidades oferecidas pela análise automática. Além disso, as informações que dão origem a uma visualização tipicamente passam por alguma espécie de análise automática. Boas visualizações são aquelas que reduzem a sobrecarga cognitiva, ocultando informações desnecessárias e chamando a atenção do usuário àquelas informações que são relevantes – a boa visualização guia o usuário à causa do comportamento que ele deseja desvendar.

Apesar da extensa literatura de pesquisa e do grande número de soluções interessantes, as ferramentas de depuração para sistemas paralelos, distribuídos e concorrentes em geral ainda são escassas. Isso ocorre porque essas ferramentas normalmente chegam tarde e morrem cedo, mas também porque uma boa parte das ferramentas estudadas nunca saíram, ao menos aparentemente, da fase de protótipo. Nós não somos os primeiros a apontarmos o aspecto bidimensional da heterogeneidade como responsável pela morte prematura de tantas ferramentas e pelo grande desperdício de esforço na área.

4 Depuração de sistemas de objetos distribuídos

"(...) if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

-- Brian W. Kernighan

Nos capítulos anteriores, nós procuramos:

- Identificar os principais problemas ligados à depuração de sistemas concorrentes, incluindo os sistemas distribuídos;
- identificar as principais técnicas e questões envolvidas no projeto de ferramentas de depuração para sistemas paralelos e distribuídos;
- convencer o leitor de que o corpo de literatura é amplo, mas que a proporção de soluções interessantes e que foram transformadas em ferramentas práticas é pequena;
- convencer o leitor de que a heterogeneidade – conseqüência direta da falta de padronização e organização na área – é uma das principais responsáveis por essa situação.

Neste capítulo, vamos começar a introduzir os elementos da nossa abordagem e quais os problemas que ela se propõe a resolver. Na Seção 4.1, apresentamos uma visão geral dos sistemas de objetos distribuídos, destacando dentre eles aqueles serão o objeto de nosso estudo – os sistemas baseados em chamadas síncronas e bloqueantes. Na Seção 4.2, discutimos as ferramentas usualmente empregadas na depuração desses sistemas e destacamos as suas limitações.

Seguimos, na Seção 4.3.1, com uma elucidação dos objetivos do nosso trabalho; isto é, apresentamos quais são os problemas que estão dentro, e fora, do escopo deste trabalho. Na Seção 4.3.2, apresentamos um dos pontos centrais em nosso trabalho – o *thread* distribuído – apresentando uma caracterização formal, de nossa autoria, que define em aspectos bastante precisos *o que* é um *thread* distribuído e qual o seu comportamento esperado*, na ausência de falhas*.

As Seções 4.3.3 e 4.3.4 discutem o *thread* distribuído como ferramenta de depuração. Na Seção 4.3.3 fazemos a ponte entre depuradores simbólicos e *threads* distribuídos, apresentando talvez uma das idéias mais importantes (e simples) do trabalho como um todo. Na Seção 4.3.4, discutimos quais vantagens podem ser obtidas com um depurador simbólico baseado em *threads* distribuídos.

4.1 Visão geral de um sistema de objetos distribuídos

Sistemas de objetos distribuídos (SODs) são concebidos em acordo com o paradigma de Orientação a Objetos (OO). Como em todo sistema OO, um sistema de objetos distribuídos é

composto por uma coleção de objetos que se comunicam enviando mensagens uns aos outros. A particularidade dos sistemas de objetos distribuídos fica por conta do fato que objetos podem residir em processos ou computadores diferentes. Esses sistemas são tipicamente organizados como uma coleção de serviços – alguns dos quais providos, por padrão, pelo middleware – onde cada serviço disponibiliza um ou mais objetos com interfaces bem-definidas. Além disso, os objetos exportados por cada servidor são normalmente endereçados e acessados por um serviço de nomes.

O modelo de programação baseado em RPC/RMI estende o modelo OO típico de troca de mensagens aos sistemas distribuídos, visando tornar a comunicação com objetos que residem em nodos distintos do nodo do cliente "tão simples" quanto a comunicação com objetos locais. Note, no entanto, que uma "mensagem" OO é, normalmente, semanticamente muito mais próxima de chamadas a procedimentos do que dos mecanismos de passagem de mensagens da MPI, por exemplo. Num sistema de middleware para objetos distribuídos típico, essa extensão se dá mediante o uso de *proxies* [69], que são objetos de interface idêntica à dos objetos remotos que representam. *Proxies* são tipicamente compostos por código gerado automaticamente pelo middleware e têm como papéis:

1. Representar o objeto remoto. O cliente interage com o *proxy* como se o objeto remoto **fosse** o *proxy*.
2. Encaminhar todas as mensagens enviadas pelo cliente ao middleware, que é quem se encarrega de despachá-las ao objeto remoto.

A outra peça do acesso transparente de objetos remotos é provida pela interface do serviço de nomes, que desacopla o cliente da implementação das interfaces providas pelos objetos potencialmente remotos. Usualmente, quando o cliente requisita ao serviço de nomes uma referência a um objeto:

1. Se o objeto for local, o serviço de nomes devolve uma referência ao próprio objeto.
2. Se o objeto for remoto, o serviço de nomes devolve uma referência para um *proxy*.

Na Figura 4-1 (a) mostramos a forma pela qual uma referência a um *proxy* é publicada e posteriormente acessada no serviço de nomes. Inicialmente o servidor obtém, por meio de algum procedimento de *bootstrapping* (normalmente um *Singleton* [69]), uma referência a um *proxy* do serviço de nomes. O servidor registra então, por meio da interface exposta por esse *proxy*, os objetos que deseja tornar acessíveis por clientes remotos (1). O *proxy* encaminha as chamadas feitas pelo servidor ao servidor de nomes (2), que registra a referência ao objeto (essencialmente o tipo do objeto remoto e o endereço do servidor) em suas estruturas de dados internas.

Em algum momento, o cliente executa um procedimento de *bootstrapping* semelhante ao do servidor e também obtém uma referência a um *proxy* do serviço de nomes. Ao invés de publicar

uma referência a um objeto, no entanto, o cliente pede ao serviço de nomes uma referência a um objeto publicado por um servidor. O servidor de nomes repassa então ao middleware no cliente (3) informações suficientes para que esse middleware possa instanciar, localmente, um *proxy* para o objeto remoto (4). Esse *proxy* será devolvido ao cliente, que passa a interagir com o objeto remoto através dele (5), como se estivesse interagindo com um objeto local.

Os *proxy* do serviço de nomes tem, no lado do cliente, um papel bastante semelhante ao de uma Fábrica [69]. O ciclo completo de uma chamada a um objeto remoto, intermediada por um *proxy*, é mostrado na Figura 4-1 (b) e descrito a seguir.

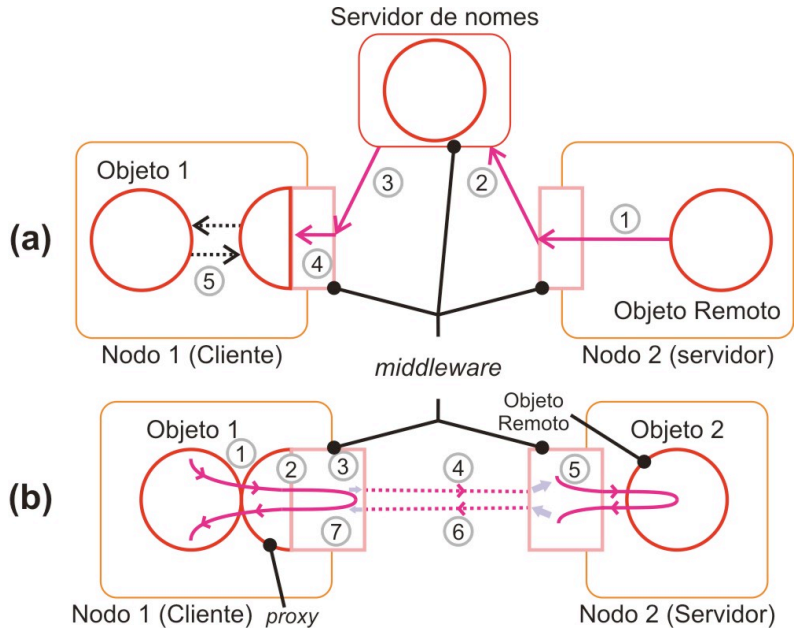

Figura 4-1. (a). (b) Forma usual de implementação de um mecanismo de chamada síncrona e bloqueante.

Ao ser acessado pelo cliente (1) durante a chamada de um método, o *proxy* repassa ao middleware (2) tanto os dados que identificam a operação remota que corresponde ao método chamado pelo cliente quanto a lista de parâmetros e seus respectivos valores. O middleware bloqueia então o *thread* do cliente (3) e inicia uma requisição ao servidor (que também faz parte do middleware) que reside no nodo que contém o objeto remoto (4). Ao receber essa requisição (que contém, entre outras informações, a identificação do objeto remoto, a identificação da operação e a lista de parâmetros), o servidor reconstrói a lista de parâmetros, localiza o objeto remoto correto e utiliza um *thread* próprio para invocar, nesse objeto remoto, o método correspondente (normalmente, de mesma assinatura) ao chamado pelo cliente no *proxy* (5). Terminado o processamento, o valor de retorno (ou exceção) gerado pelo objeto remoto é transmitido pela rede

de volta ao cliente (6), que desbloqueia o *thread* (7) e dá seguimento à execução (devolvendo o valor ou jogando a exceção).

Essa "extensão" do modelo de chamada a métodos provida pelos sistemas de middleware baseados em chamadas síncronas e bloqueantes cria a ilusão de que *threads* são entidades capazes de atravessar os limites de um elemento de processamento (Figura 4-2). Vamos nos referir a esses *threads* lógicos, capazes de atravessar os limites de um elemento de processamento, como *threads distribuídos*. Um dos efeitos da extensão do modelo de mensagens OO típico para sistemas distribuídos é que ele resulta em uma extensão também do modelo baseado em *threads* e memória compartilhada e isso, como discutimos anteriormente, não é necessariamente algo bom.

Os problemas com tentativas de uma integração irrestrita entre programação centralizada e distribuída já são conhecidos há algum tempo [117, 215, 225]. As diferenças mais fundamentais entre os dois modelos, segundo Waldo [225], Schroeder [183] e Hadzilacos [76], ficam por conta do fato de que sistemas distribuídos são obrigatoriamente concorrentes e podem falhar parcialmente. Portanto, é atualmente aceito que a transparência provida por sistemas de middleware baseados em RPC/RMI deve aproximar-se mais de um modelo conveniente de comunicação do que de uma solução mágica para todos os problemas enfrentados no desenvolvimento de sistemas distribuídos.

Dito isso, mesmo com o sacrifício da transparência, diversos problemas ainda persistem. O uso do modelo de *threads* implica que os mesmos problemas de composição de componentes que surgem no caso centralizado [208], surgem no caso distribuído [117]. Estamos nos referindo, particularmente, ao fato de que pode ser impossível determinar a priori, especialmente num sistema distribuído, a forma pela qual um componente remoto adquire travas de exclusão mútua. Essa falta de visão global resulta, com freqüência, no aparecimento de *deadlocks* distribuídos. Além disso, o *thread* lógico capaz de ultrapassar o limite dos nodos é uma entidade de conhecimento do middleware, mas não das primitivas da sincronização da linguagem. Não é incomum, portanto, que o uso de travas reentrantes e RPC/RMI resulte em *deadlocks* entre o *thread* lógico e ele próprio, conforme mostra a Figura 4-2.

Dessa nossa breve análise, podemos concluir que um sistema de objetos distribuídos típico:

- É um sistema distribuído baseado em passagem de mensagens, composto por coleções de sistemas concorrentes menores que, por sua vez, usam múltiplos *threads* e compartilham dados de forma não-estruturada (*unstructured parallelism* [208]).
- Faz uso de chamadas síncronas e bloqueantes e sofre, por conseqüência, com os problemas ligados ao modelo de *threads* e memória compartilhada discutidos anteriormente.

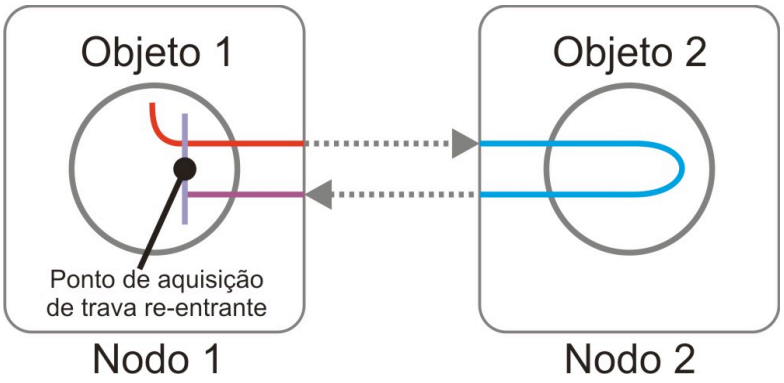

Figura 4-2. A ilusão provida pelo middleware – um *thread* lógico (*thread distribuído*), capaz de invocar métodos em objetos que residem em outro espaço de endereçamento – e um "auto-deadlock".

4.2 Ferramentas usuais

Conforme procuramos demonstrar no Capítulo 3, a disponibilidade de ferramentas de depuração para sistemas distribuídos é escassa. Sob essa perspectiva, os sistemas de objetos distribuídos não são exceção. Embora seja possível conceber um ambiente de depuração bastante completo agrupando algumas das ferramentas apresentadas no Capítulo 3, a heterogeneidade aliada à disponibilidade reduzida de implementações inviabiliza essa abordagem.

Essa escassez acaba por levar a grande maioria dos desenvolvedores de sistemas de objetos distribuídos a recorrerem a ferramentas projetadas para sistemas de middleware específicos (como o OLT [87] e o OVATION [150]), quando disponíveis, ou, mais comumente, a usarem ferramentas genéricas, porém amplamente disponíveis, como os depuradores simbólicos ou o ubíquo comando *print.*

4.2.1 O comando *print*

Na terminologia que desenvolvemos no decorrer deste texto, o comando *print* compreende apenas parte de um mecanismo de monitoramento em software. A funcionalidade provida pelo mecanismo se resume a uma forma de externar informações textuais de estrutura livre, produzidas pela própria aplicação. O uso do comando *print* como ferramenta de depuração vem aliado, portanto, a uma abordagem *ad hoc* de coleta de dados onde o desenvolvedor instrumenta, manualmente, pontos que considera estratégicos no programa. As principais vantagens do comando *print* são:

- **Flexibilidade**: Embora primitiva, a abordagem permite um grau de liberdade bastante elevado.

- **Ubiqüidade**: É difícil imaginar uma linguagem – seja qual for a escolha de sistema operacional, kit de desenvolvimento e hardware – sem um comando *print* funcional.

Alguns dos problemas imediatos, no entanto, são:

- O ônus por projetar e implementar um mecanismo de monitoramento que produza informações úteis no contexto do comportamento observado fica por conta do desenvolvedor;
- A instrumentação manual é cansativa e propensa a erros;
- A captura de relações causais é inviável, ao menos sem o uso de mecanismos auxiliares;
- A introdução de mecanismos *ad hoc* de coleta de dados pode inserir novos defeitos, ou eliminar defeitos existentes.

Portanto, podemos dizer que o comando *print*, sozinho, dificilmente poderia ser considerado uma "ferramenta de depuração" – trata-se apenas de um mecanismo rudimentar para comunicação de informações não-estruturadas. Toda e qualquer estrutura fica a cargo do desenvolvedor. Particularmente, os modernos mecanismos para *logging* (como o amplamente utilizado *Apache Log4J* [9]) poderiam ser encarados como meras extensões do comando *print*, oferecendo mais opções de configuração e mecanismos de auxílio.

4.2.2 Depuradores simbólicos

Depuradores simbólicos, por sua vez, são sistemas de depuração interativos e centrados no texto do programa (código-fonte), que atuam no nível de abstração das entidades de tempo de execução da linguagem subjacente. No caso de linguagens imperativas (como Java, C, C#, Python, Ruby, Smalltalk, etc.), esses depuradores tipicamente viabilizam a inspeção de *threads*, registros de ativação (*stackframes*) e fragmentos de estado (objetos, variáveis, áreas de memória), acessíveis pelos *threads* da aplicação. O principal objetivo de um depurador simbólico é mostrar, ao usuário, a execução de sua aplicação em termos das abstrações da linguagem na qual essa aplicação foi escrita. Esse tipo de visualização é interessante por apresentar a execução num formato que é cognitivamente compatível com aquilo que é visualizado pelo programador durante o processo de elaboração do texto programa (codificação) [95].

Embora isso implique que o formato de um depurador simbólico possa variar substancialmente com a linguagem, as seguintes características são comuns:

- **Visualização textual:** a abstração de visualização padrão do depurador simbólico é o código fonte do sistema-alvo.
- **Navegação interativa com animação sobre texto:** o mecanismo de navegação da execução é interativo, *on-line* e *limitado*. O usuário navega pela execução do programa

seguindo o fluxo da execução conforme estabelecido pela semântica da linguagem, com o auxílio de *breakpoints* e *execução passo-a-passo*.

Definição 4-1: *Execução passo-a-passo* é um modo de navegação interativo onde o usuário percorre a execução da aplicação em incrementos de tamanho lógico fixo.

O tamanho mais comum para o incremento da execução passo-a-passo é a linha de código. Dizemos "incremento de tamanho lógico fixo" porque a noção de tamanho vem associada a transições de estado, que são naturalmente especificadas num nível bastante abstrato (por exemplo, linhas de código). Segundo Kumar [107], a execução passo-a-passo reifica a semântica da execução, facilitando a sua compreensão. Os tipos de *breakpoints* providos por depuradores simbólicos varia, mas a grande maioria deles utilizam predicados locais como condição de parada; isto é, predicados especificados sobre o estado de um processo individual [30].

Daquilo que discutimos até o presente momento, as vantagens do uso de depuradores simbólicos devem estar claras:

- Utilizam um modelo de visualização simples e compatível com o modelo mental do programador;
- Por serem *on-line*, interativos e operarem num nível de granulosidade bastante fina, viabilizam a inspeção detalhada de estados, sem que a instrumentação tenha que ser planejada antes;
- Não requerem instrumentação manual;
- Apresentam *breakpoints* suficientemente expressivos para a maior parte dos casos com aplicações centralizadas.

O problema com o uso de depuradores simbólicos para aplicações concorrentes, discutido em praticamente todos os artigos e textos que falam sobre depuração distribuída (por exemplo, em [104, 111, 144, 148]) e no início do Capítulo 3, é que depuradores simbólicos foram projetados tendo em vista sistemas seqüenciais e centralizados. Isso acarreta uma série de dificuldades, dentre as quais podemos destacar:

- **Reprodução do comportamento do sistema:** a reprodução de execuções concorrentes – mais especificamente, de instruções *fracamente determinísticas* – é algo fora do escopo da maior parte dos depuradores simbólicos existentes (técnicas de reprodução para sistemas concorrentes foram discutidas na Seção 3.2).
- **Expressividade dos *breakpoints*:** os mecanismos de *breakpointing* providos por depuradores simbólicos não capturam predicados que envolvam múltiplos processos.

- **Captura causal:** Depuradores simbólicos não explicitam nem tampouco rastreiam as relações de causa e efeito num sistema distribuído.
- **Escalabilidade da visualização:** O mecanismo de visualização padrão dos depuradores simbólicos – as entidades abstratas do ambiente de execução do sistema-alvo – não é escalável.

Além disso, as seguintes questões, relevantes apenas no contexto dos sistemas de objetos distribuídos, surgem do uso de depuradores simbólicos convencionais:

- **Incompatibilidade de abstrações:** o sistema de middleware utiliza código gerado automaticamente, além de código do próprio arcabouço, para implementar a ilusão de chamadas a métodos remotos (RMI). Durante a codificação, o desenvolvedor trabalha com a ilusão (parcial) de que objetos remotos são acessados de forma semelhante a objetos remotos. Depuradores simbólicos, no entanto, não são capazes de manter essa ilusão em tempo de execução, por trabalharem num nível de abstração distinto do middleware. Isso implica, entre outras coisas, que o usuário é obrigado a recorrer a métodos manuais para rastrear o fluxo de sua aplicação, além de ser exposto a código que, muitas vezes, não faz sentido para ele.
- **Efeito labirinto:** depuradores simbólicos reificam diversos aspectos do ambiente de execução do sistema-alvo. A exposição desses objetos de execução não é, no entanto, seletiva. Isso implica que todos os objetos de execução criados pelo middleware serão mostrados, expondo o usuário a informações que normalmente não lhe são relevantes e contribuindo com o efeito labirinto.

4.3 Depuração e *threads* distribuídos

É bastante curioso que depuradores simbólicos sejam ferramentas tão amplamente difundidas, enquanto que ferramentas como as apresentadas no Capítulo 2 não sejam sequer conhecidas pela grande maioria dos desenvolvedores. O argumento de que ferramentas para sistemas concorrentes como as do Capítulo 2 são complexas, implícito e explícito no argumento de tantos autores, não chega a convencer. As dificuldades envolvidas na implementação de um depurador simbólico para uma linguagem moderna como Java, por exemplo, que conta com compiladores que otimizam código de forma agressiva, compiladores *Just-In-Time* e *multithreading*, são consideráveis. De fato, a complexidade envolvida na implementação de somente um dos aspectos da plataforma de depuração Java [201], o mecanismo de *hot-swapping*, já é considerável [48].

4.3.1 Proposta deste trabalho

Mudar o quadro de difusão de técnicas de depuração está além do escopo deste trabalho – e além do escopo de qualquer trabalho individual. Nossa proposta, bastante mais simples, é a de utilizar o ferramental provido pelos depuradores simbólicos, essas ferramentas poderosas, complexas e abundantemente disponíveis, a nosso favor. Não há nada de novo nessa idéia, trabalhos como o CDB [232], DDB [190], DDBG [118], Panorama [128], Buster [229], P2D2 [28], entre outros, foram todos desenvolvidos sob essa premissa. Não coincidentemente, a maior parte desses trabalhos (com a exceção do DDBG [118]) coloca a portabilidade como motivação central na adoção de depuradores simbólicos como base. A diferença de nosso com relação aos trabalhos citados é que, enquanto os trabalhos citados têm como foco os sistemas paralelos, nossos esforços foram concentrados em sistemas de objetos distribuídos. Além disso, houve de nossa parte uma preocupação explícita em desenvolver um sistema de software que possa ser adaptado com pouco esforço (dentro daquilo que é possível) a uma variedade de sistemas de middleware para objetos distribuídos.

O leitor também deve notar, no decorrer do restante do texto, que a ligação do nosso trabalho com depuradores simbólicos é mais profunda do que nos outros trabalhos (compare com [118, 128, 229]). Não utilizamos depuradores simbólicos como simples mecanismos de instrumentação portáteis – nós aumentamos os depuradores simbólicos de forma a encaixá-los no contexto de sistemas de objetos distribuídos. A forma pela qual acontece esse "encaixe" está intimamente relacionada à extensão do modelo de *threads* pelo middleware de objetos distribuídos e será discutida na Seção 4.3.2.

Deve ficar claro que a proposta deste trabalho não é resolver todos os problemas apresentados no Capítulo 2, mas sim produzir uma técnica útil, portável e viável na prática, que possa auxiliar no desenvolvimento de sistemas de objetos distribuídos.

4.3.2 Caracterização do *thread* distribuído

Os *threads* distribuídos ocupam uma posição central em nosso trabalho – tanto porque o trabalho se baseia neles quanto pela importância adquirida pela abstração no decorrer da história dos sistemas de middleware para objetos distribuídos. Um *thread* distribuído é, informalmente, o *thread* lógico capaz de cruzar os limites dos nodos de processamento que descrevemos na Seção 4.1. Colocando em termos um pouco mais formais, um *thread* distribuído é composto por uma série de seqüências (conjuntos totalmente ordenados) de *threads locais*, que variam no decorrer do tempo. Antes de prosseguirmos com uma definição mais formal de *thread distribuído*, portanto, gostaríamos de definir informalmente o que são *threads locais*.

Definição 4-2: Um *thread local* corresponde a uma linha de execução dentro de um processo. Todos os *threads* em um processo compartilham o mesmo espaço de endereçamento. *Threads locais* estão restritos a um único nodo de processamento.

Exemplos de *threads* locais incluem os "processos leves" tradicionais, como aqueles implementados no nível do núcleo em sistemas operacionais como o GNU/Linux e o *Microsoft Windows*, bem como implementações em espaço de usuário, tanto preemptivas quanto "*green-threads*" (não preemptivas). Nossa definição de *thread local* depende de uma definição de processo. Vamos nos contentar com uma definição bastante simples, dada por Tanenbaum [213]:

Definição 4-3: Um *processo* é um programa em execução.

Por simplicidade, vamos tomar como hipótese que o tempo pode ser representado como um conjunto contínuo e linear de instantes de duração zero, que é isomórfico ao conjunto dos números reais $\mathbb{R}$. Embora existam argumentos contundentes contra esse modelo clássico de tempo (por exemplo, nos trabalhos de Dummet [54] e Read [169]), ele será suficiente para nossos propósitos.

Dada uma execução distribuída *C*, vamos chamar de L_C o conjunto de todos os *threads locais* que nela participam. Iremos supor que, para cada *thread* local $l \in L_C$, existem dois instantes $t_l^s, t_l^d \in \mathbb{R}$ que correspondem aos instantes em que *l* começa (*starts*) e termina (*dies*), respectivamente. Dizemos que o intervalo $[t_l^s, t_l^d]$ é o *intervalo de vida* de *l*. Além disso, definimos o *intervalo de vida* da execução distribuída *C* como $[\min\{t_l^s : l \in L_C\}, \max\{t_{l'}^d : l' \in L_C\}]$. Deve ser claro que os intervalos de vida dos *threads* locais de L_C são todos subconjuntos do intervalo de vida da execução distribuída *C*.

Definição 4-4: Seja *C* uma execução distribuída. Definimos uma *fotografia de um thread distribuído* como uma seqüência $s = \{l_1, ..., l_n\}$ de *threads* locais, onde $l_i \in L_C$ para $1 \le i \le n$, $i \in \mathbb{N}$. Uma fotografia de um *thread* distribuído é *trivial* quando contém um único *thread* local.

De forma análoga ao que fizemos com L_C, definimos S_C como o conjunto de todas as possíveis fotografias que podem ser produzidas com *threads* locais tirados de L_C. A formalização do conceito de *thread distribuído* é dada na Definição 4-5. Note que essa definição apresenta em termos bastante precisos <u>qual o comportamento esperado</u> para um *thread distribuído*.

Definição 4-5: Seja *C* uma execução distribuída. Um *thread distribuído T* é definido como uma seqüência de fotografias $\{s_{t_1},...,s_{t_n}\}$, onde cada t_i representa um instante no tempo e $s_{t_i} \in S_C$ $(1 \le i \le n,\ i \in \mathbb{N})$. Além disso, se $i < j$, então $t_i < t_j$; isto é, *T* é totalmente ordenado com relação ao tempo. Para todo *thread* distribuído $T = \{s_{t_1},...,s_{t_n}\}$, as seguintes propriedades devem ser satisfeitas:

1. Existe um *thread* local $l_1 \in L_C$ tal que $s_{t_1} = \{l_1\}$ e l_1 é o primeiro elemento de s_{t_i}, para todo *i*. Dizemos que l_1 é a **base** do *thread* distribuído *T*.
2. Seja $i \in \mathbb{N}$ e $1 < i \le n$, e seja $s_{t_{i-1}} = \{l_1,...,l_m\}$. Então, na ausência de falhas, exatamente uma das seguintes alternativas deve valer:
   a. $s_{t_i} = \{l_1,...,l_m,l_{m+1}\}$, onde $l_{m+1} \in L_C$. Neste caso, l_m iniciou uma chamada remota no instante $t_{i-1} + \delta \le t_i$ (onde $\delta \in \mathbb{R}$ e $\delta > 0$), e o *thread* l_{m+1} começou a tratar essa chamada remota no instante t_i.
   b. $s_{t_i} = \{l_1,...,l_m\}$. Neste caso, o *thread* l_m terminou de tratar, no instante t_i, a requisição remota previamente iniciada por l_{m-1}.
3. Sejam t_T^s e t_T^d os instantes em que *T* começa e termina, respectivamente. Então $t_1 = t_{l_1}^s = t_T^s$ e $t_n < t_l^d = t_T^d$.

Seja *C* uma execução distribuída. De forma análoga ao que fizemos com L_C e com S_C, definimos D_C como o conjunto de todos os *threads* distribuídos que participam de *C*. A Definição 4-5 traz algumas implicações importantes, que procuramos deixar mais claras a seguir. A primeira implicação é que, para todo *thread* local $l \in L_C$, existe um *thread* distribuído *T* tal que:

1. *T* começa e termina junto com *l*;
2. *l* é a base de *T*.

A segunda implicação é que as fotografias que compõem um *thread* distribuído podem ser interpretadas da seguinte forma. Seja $T = \{s_{t_1},...,s_{t_n}\}$ (onde $s_{t_i} \in S_C$ para $1 \le i \le n$) um *thread* distribuído. Se tomarmos $s_{t_k} = \{l_1,...,l_m\}$, onde $1 \le k < n$, então, entre os instantes t_k e t_{k+1}:

1. Para $1 < j \le m$, l_j está tratando uma requisição remota iniciada por l_{j-1};
2. Se $1 \le j < m$, l_j está bloqueado numa chamada remota, tratada por l_{j+1}.

Essa última observação nos leva, naturalmente, à definição de estado de um *thread* distribuído. O *estado* de um thread distribuído *T* é definido, num instante arbitrário *x*, como a última fotografia que acontece em *T* antes do instante *x*.

Definição 4-6: Seja *C* uma execução distribuída e seja D_C o conjunto de todos os *threads* distribuídos que existiram em *C*. Seja agora $T = \{s_{t_1},...,s_{t_n}\}$, $T \in D_C$, um desses *threads* distribuídos. O *estado do thread distribuído T no instante* $x \in \mathbb{R}$ é dado pela função $f : D_C \times \mathbb{R} \longrightarrow S_C$, onde:

$$f(T,x)\begin{cases} s_{t_i},\ se\ t_i \le x < t_{i+1}\ e\ 1 \le i < n,\ ou \\ \quad t_i \le x < t_T^d\ e\ i = n \\ \phi,\ caso\ contrário \end{cases}$$

Apresentada a definição do estado de um *thread* distribuído num instante *x*, vamos apresentar agora duas outras definições que utilizaremos no restante do texto. Seja *C* uma execução distribuída e seja $T \in D_C$, tal que $f(T,t) = \{l_1,...,l_k\}$ para um instante $t \in \mathbb{R}$. Então:

Definição 4-7: l_k é a *cabeça* de *T* no instante *t*.

Definição 4-8: Os *threads* locais $\{l_1,...,l_k\}$ <u>participam</u> do *thread* distribuído *T* no instante *t*.

Finalmente, a nossa noção de um conjunto de *threads* distribuídos válido será limitada por uma regra que denominamos a "regra da participação única". A "regra da participação única" restringe a relação de participação entre *threads* locais e *threads* distribuídos, definindo sob quais circunstâncias é válido que um *thread* local *l* participe em mais de uma fotografia num dado instante. Há basicamente uma única situação em que vamos permitir que um *thread* local participe de mais de um *thread* distribuído simultaneamente. Essa situação será caracterizada com a ajuda de uma última definição, a Definição 4-9, e da Observação 4-1.

Definição 4-9: Sejam s_1, s_2 duas fotografias de *threads* distribuídos. Dizemos que s_1 é uma *subfotografia* de s_2, ou ainda, que $s_1 < s_2$, se e somente se s_1 for um sufixo próprio de s_2.

Observação 4-1: Seja $s_{t_i} = \{l_1,...,l_n\}$ uma fotografia não-trivial de um *thread* distribuído $T = \{s_{t_1},...,s_{t_k}\}$. Seja ainda T' o *thread* distribuído cuja base é l_2. Então s_{t_i} pode ser expressa como a concatenação das seqüências $\{l_1\}$ e $f(T',t_i)$. Ainda, $f(T',t_i) < f(T,t_i)$.

Para que o significado da Observação 4-1 fique um pouco mais claro, seja T um *thread* distribuído. Suponhamos que $f(T,t)=\{l_1,l_2,l_3\}$ em algum instante $t \in \mathbb{R}$. A observação nos mostra que $f(T',t)=\{l_2,l_3\} < f(T,t)$, onde T' é o *thread* distribuído de base l_2. Isso implica que, no instante *t*, o *thread* local l_2 participa, simultaneamente, em duas fotografias: $\{l_1,l_2,l_3\}$ e $\{l_2,l_3\}$. O *thread* l_3, por sua vez, participa simultaneamente em três fotografias: $\{l_1,l_2,l_3\}$, $\{l_2,l_3\}$ e $\{l_3\}$. O que é importante notar nos dois exemplos é que as fotografias "encaixam-se" umas nas outras; isto é, cada uma dessas fotografias, com a exceção da maior delas, é um sufixo próprio de alguma outra fotografia. Essa será a única situação em que iremos permitir que um *thread* local participe em mais de uma fotografia simultaneamente, e é justamente daí que sai a "regra da participação única".

Definição 4-10: Seja C uma execução distribuída. Para cada *thread* local $l \in L_C$, definimos $P_t^l = \{f(T,t) \in S_C : T \in D_C \wedge l \in f(T,t)\}$ como o conjunto de todas as fotografias em que *l* participa no instante $t \in \mathbb{R}$. Seja $l \in L_C$ um *thread* local e *x* um instante no intervalo de vida de *C*. Seja ainda $P_x^l = \{s_1,...,s_k\}$ ($k \in \mathbb{N}$). Dizemos que a regra da participação única é obedecida no instante *t* se, e somente, se existe uma permutação π de $\{1,...,k\}$ tal que $s_{\pi(1)} < ... < s_{\pi(k)}$.

A "regra da participação única" é uma imposição sobre a política de reuso de *threads* pelo middleware que deve ser obedecida para que a técnica de depuração descrita neste trabalho seja aplicável. Essencialmente, a técnica não permite o uso de *threads* locais bloqueados em chamadas remotas para o tratamento de requisições que pertençam a outros threads distribuídos. Notadamente, alguns ORBs para sistemas de tempo real [106] podem violar essa regra.

Estamos finalmente prontos para definirmos o que é uma execução válida, do ponto de vista de nossa técnica depuração.

Definição 4-11: Seja *C* uma execução distribuída. Dizemos que *C* é *válida* se e somente se a regra da participação única for obedecida para todo instante *t* no intervalo de vida de *C* (i.e., para todo $t \in [t_C^s, t_C^d]$).

Note que a noção de uma execução "válida" coloca em termos precisos as hipóteses feitas pela técnica de depuração que será descrita no restante deste trabalho. Conforme indicamos anteriormente, no entanto, é perfeitamente possível que um sistema de middleware produza *threads* distribuídos que resultem em execuções "não-válidas", do ponto de vista da Definição 4-11, sem que isso tenha qualquer implicação negativa sobre a correção dessas execuções ou sistemas de middleware. A técnica de depuração descrita neste trabalho, no entanto, só produz resultados corretos para execuções "válidas" segundo a Definição 4-11.

A Figura 4-3 procura ilustrar os principais conceitos apresentados até agora: a Figura 4-3 (a) apresenta a sucessão de fotografias num *thread* distribuído durante uma cadeia de chamadas que cruza três nodos.

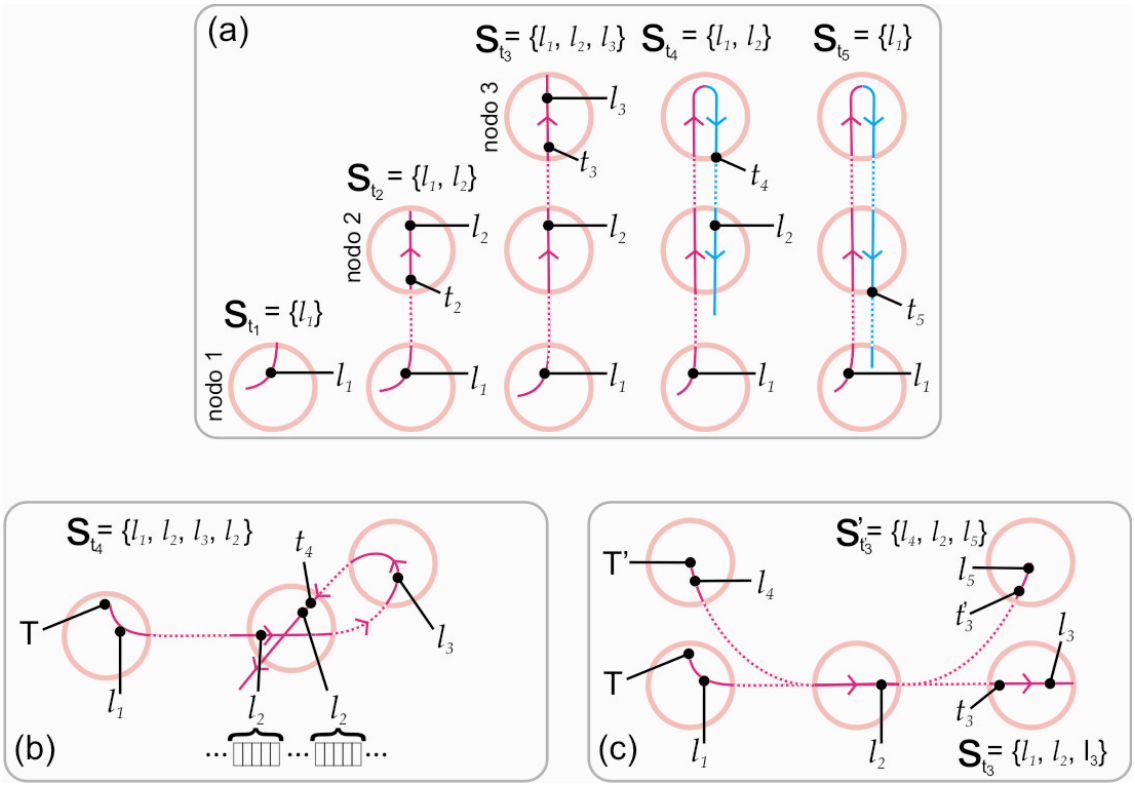

Figura 4-3. (a) Um *thread* distribuído e sua seqüência de fotografias durante uma cadeia de chamadas envolvendo três nodos. (b) Um caso válido de reuso para um *thread* local. (c) Trecho de uma execução não-válida.

A Figura 4-3 (b) mostra um caso em que o reuso de um *thread* local bloqueado para o tratamento de uma outra requisição é válido: quando esse *thread* local é reutilizado dentro do **mesmo** *thread* distribuído, a regra da participação única não é violada. Figura 4-3 (c), por sua vez, mostra um trecho de uma execução não-válida. Para ver o porque, basta tomarmos os estados dos *threads* distribuídos $f(T,t_3') = \{l_1, l_2, l_3\}$ e $f(T,t_3') = \{l_4, l_2, l_5\}$. Agora basta notar que, no instante t_3', o *thread* local l_2 participa de diversas fotografias que não são sufixos próprios umas das outras. Por exemplo, l_2 participa de $s = \{l_2, l_3\}$ e de $s' = \{l_2, l_5\}$, mas $s \not< s'$ e $s' \not< s$.

O último aspecto que nos resta definir acerca da anatomia dos *threads* distribuídos e sua relação com *threads* locais diz respeito ao significado dos eventos que demarcam as mudanças de fotografias descritas na Definição 4-5. Em outras palavras, nos resta estabelecer significados mais precisos para termos como "começar" e "terminar" de "tratar uma requisição" e de "estar bloqueado

em uma chamada remota". Para tanto, vamos enriquecer um pouco a nossa noção de *threads* locais. Primeiramente, iremos supor que cada a *thread* local encontra-se associada uma pilha de registros de ativação [214] (*activation records* ou *stackframes*). Registros de ativação são estruturas de dados vinculadas pelo ambiente de execução à ativação de cada procedimento, função, ou método por um *thread* local num processo. Normalmente, um novo registro de ativação é criado e empilhado sempre que um método é chamado (ativado) e desempilhado (e destruído) quando esse método retorna.

Seja *C* uma execução distribuída. Vamos chamar de R_C o conjunto de todos os registros de ativação que ocorrem em *C* (isto é, todos os registros de ativação que participam em pilhas associadas a *threads* locais em L_C). Conforme mencionamos anteriormente, todo registro de ativação $q \in R_C$ vem associado a um procedimento, método ou função (doravante chamados de *blocos*). Vamos chamar de M_C o conjunto de todos os blocos em *C*. Os métodos ativados num objeto que é instância de uma classe *A* podem, do ponto de vista de uma execução assim construída, ser representados tuplas de elementos em M_C, i.e, como elementos em $(M_C)^n, n \in \mathbb{N}$. Note que a classe *A* pode declarar outros métodos além daqueles em M_C, mas esses métodos não nos são de interesse, nem fazem parte de M_C, pois não foram ativados.

Definição 4-12: Definimos a localização de um registro de ativação como uma função $loc : R_C \longrightarrow M_C$ tal que, para todo c, $loc(q) = m$, onde *m* corresponde ao bloco ativado por *q*.

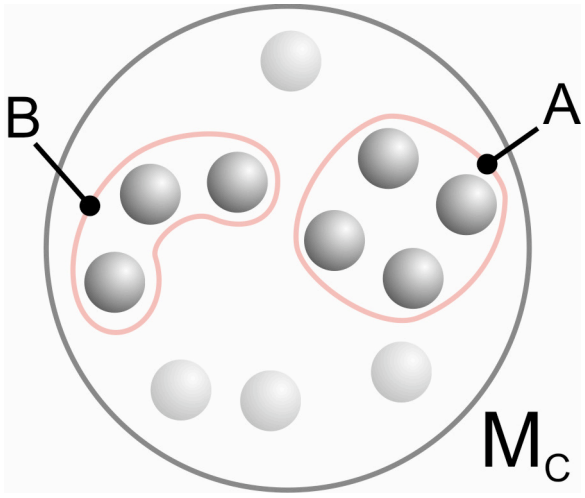

Figura 4-4. O conjunto M_C com os métodos declarados por duas classes, A e B, demarcados.

Vamos particionar M_C em dois subconjuntos que nos serão úteis mais adiante: $M_{C_{mid}}$, o conjunto de blocos ativados em código do middleware e $M_{C_{app}}$, o conjunto de blocos ativados em código da aplicação.

Um *thread* local tem uma estrutura bastante semelhante, na ausência de falhas, à estrutura de um *thread* distribuído. Da mesma forma que podemos descrever um *thread* distribuído como uma seqüência de fotografias compostas por seqüências de *threads* locais, podemos descrever um *thread* local como uma seqüência de fotografias de sua pilha de chamadas, onde essas fotografias são compostas por seqüências de quadros de ativação. Podemos portanto definir, de maneira um tanto quanto informal, o estado de um *thread* local $l \in L_C$ num instante $t \in \mathbb{R}$ como uma função $stk : L_C \times \mathbb{R} \longrightarrow (R_C)^n$ ($n \in \mathbb{N}$) que nos informa a seqüência de quadros de ativação, ou o estado da pilha de chamadas, de *l* no instante *t*.

Definição 4-13: Seja $stk(l,t) = \{q_1,...,q_n\}$ o estado pilha de chamadas de $l \in L_C$ num instante $t \in [t_l^s, t_l^d)$ e seja $P = \{e_1,...,e_k\}$, um conjunto de subseqüências contínuas de $stk(l,t)$ (isto é, $e_i = \{q_j,...,q_m\}$ para algum par $j,\ m \in \mathbb{N}$, $1 \le j < m \le n$). Dizemos que *P* é um *particionamento por localização* de $stk(l,t)$ se, e somente se:

1. $e_1 \cup ... \cup e_k = stk(l,t)$,
2. para todo $i,\ 1 \le i \le n,\ i \in \mathbb{N}$, e_i é uma subseqüência **contínua e maximal** de $stk(l,t)$ que satisfaz às seguintes propriedades:
    a. se $loc(q) \in M_{C_{app}}$ e $q \in e_i$, então não existe $q' \in e_i$ tal que $loc(q') \in M_{C_{mid}}$,
    b. se $loc(q) \in M_{C_{mid}}$ e $q \in e_i$, então não existe $q' \in e_i$ tal que $loc(q') \in M_{C_{app}}$.

Não é difícil ver que *P* é de fato um particionamento de $stk(l,t)$. Para entender porque, basta notar que a propriedade (1) implica que os elementos de *P* cobrem $stk(l,t)$ e que, se supusermos que existem $x, y \in P : x \ne y$ e $x \cap y \ne \phi$, então x ou y não são maximais, ou alguma das propriedades (2.a) ou (2.b) é violada.

A Figura 4-5 mostra um particionamento por localização do estado da pilha de chamadas de um *thread* local l_1. Cada retângulo na pilha de chamadas de l_1 representa um quadro de ativação. Seja $l \in L_C$ um *thread* local e $P = \{e_1,...,e_k\}$ um particionamento de localização de $stk(l,t)$ num instante $t \in [t_l^s, t_l^d)$. Então:

Definição 4-14: Seja $A \in (M_C)^n$ uma classe e seja $e_i = \{q_a, ..., q_b\} \in P$. Dizemos que um registro de ativação $q_j \in e_i, j \in \mathbb{N}$, é um *registro de entrada em A na subseqüência* e_i, se, e somente se, q_j é o primeiro registro em e_i a ativar um método em uma instância de *A*; i.e., se $j = \min\{x \in \mathbb{N}, a \le x \le b : loc(q_x) \in A\}$.

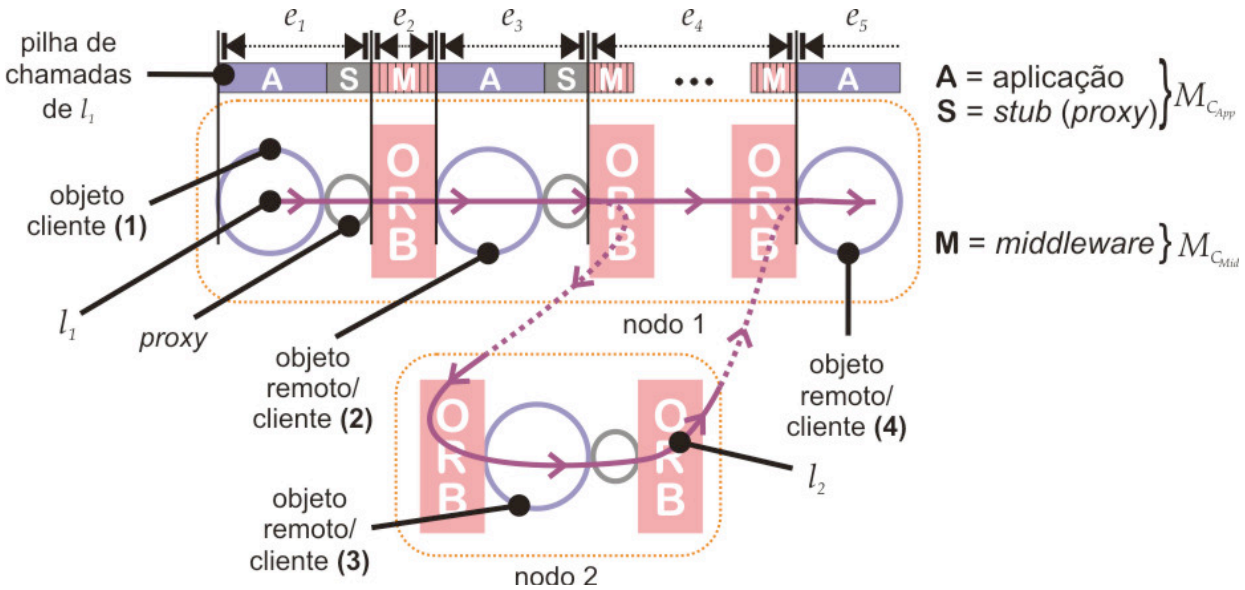

Figura 4-5. Particionamento de localização sobre o estado da pilha de chamada de um *thread* local.

Definição 4-15: Um registro de ativação $q \in stk(l,t)$ é um *registro de chamada remota em* $e \in P$ se, e somente se:

1. $q \in e$,
2. q é um registro de entrada em uma classe *A* na subseqüência *e*;
3. as instâncias de *A* são *proxies*.

Definição 4-16: Um registro de ativação $q \in stk(l,t)$ é um *registro de tratamento em* $e \in P$ se, e somente se:

1. $q \in e$,
2. q é um registro de entrada em uma classe *A* na subseqüência *e*;
3. as instâncias de *A* são *objetos remotos*[7].

Dadas as definições de registro de chamada remota e registro de tratamento, temos:

[7] Ou *serventes,* na terminologia de CORBA.

Definição 4-17: O *thread* local *l* está *bloqueado em uma chamada remota* se, e somente se, existem $e \in P$, $q \in e$, tal que *q* é um registro de chamada remota em *e*.

De maneira semelhante:

Definição 4-18: O *thread* local *l* está *tratando uma requisição* se, e somente se, existem $e \in P$, $q \in e$, tal que *q* é um registro de tratamento em *e*.

Particularmente, é perfeitamente possível que um *thread* local trate uma requisição **enquanto** bloqueado em uma chamada remota, ou vice-versa, como mostra a Figura 4-5.

4.3.3 Visualização e sinergia com depuradores simbólicos

O princípio que governa este trabalho deve muito à percepção de que existe uma sinergia natural entre os populares, e amplamente disponíveis, depuradores simbólicos e os sistemas de objetos distribuídos baseados em chamadas síncronas e bloqueantes. O pivô dessa sinergia é, justamente, o *thread* distribuído. Depuradores simbólicos convencionais, como muitos dos projetados para linguagens imperativas e funcionais baseadas em *threads* ou processos, exibem a evolução do estado de um sistema não-distribuído como a união da evolução dos estados de *threads* individuais nesse sistema. Essas ferramentas representam o sistema em execução como uma coleção de *threads*, suspensos ou em execução, que podem ser manipulados pelo usuário de formas variadas. Todas as operações de inspeção de estado subseqüentes se desenvolvem a partir da manipulação desses *threads*.

Normalmente, o usuário seleciona um *thread* suspenso, escolhe um quadro de ativação na pilha de chamadas e, a partir daí, intercala inspeções nas entidades de tempo de execução (variáveis locais, globais e outros objetos) com passos na execução passo-a-passo. O ponto do código em execução num quadro de ativação de um *thread* suspenso é representado por um marcador, desenhado sobre o código-fonte da aplicação. Esses marcadores são animados durante a execução passo-a-passo, ilustrando de forma intuitiva o fluxo de controle da aplicação. O fluxo de trabalho (*workflow*) num depurador simbólico típico (como o GDB [197], ou o depurador do JDT [57]) é descrito de forma simplificada a seguir:

1. Inicialmente, o usuário:
    a. posiciona *breakpoints* em pontos estratégicos de sua aplicação;
    b. lança a aplicação e espera que esses *breakpoints* sejam atingidos por um ou mais *threads* locais.
2. A qualquer instante, o usuário:
    a. suspende *threads* em busca de mais informações;

b. posiciona novos *breakpoints*;
c. inspeciona a saída da aplicação;
d. termina a aplicação.

3. Quando um *thread* é suspenso, o usuário:
    a. inspeciona a pilha dos threads suspensos, selecionando quadros de ativação arbitrários;
    b. inspeciona as entidades de tempo de execução acessíveis por esses *threads*;
    c. observa o progresso do programa em execução no modo passo-a-passo (Definição 4-1).

As próximas figuras mostram quatro depuradores simbólicos – o depurador do JDT (Figura 4-6 (a)), mencionado anteriormente, o depurador do ambiente *Squeak/Smalltalk* [91] (Figura 4-6 (b)), um depurador para *JavaScript* [140] (Figura 4-7 (a)), integrado a um navegador, e o GDB com um *frontend Emacs/Aquamacs* [10] (Figura 4-7 (b)). Note, nas quatro interfaces, a presença da coleção de *threads* ativos, dos marcadores desenhados sobre o código-fonte do programa e do mecanismo de seleção de quadro de ativação.

Conforme mencionamos anteriormente, o uso de chamadas síncronas e bloqueantes num sistema de objetos distribuídos cria a ilusão de que o sistema distribuído é, na verdade, um grande sistema *multithreaded*. Os *threads* nesse sistema *multithreaded* ilusório, no entanto, são bastante especiais, na medida em que são capazes de chamar métodos em objetos que residem em espaços de endereçamento distintos dos que originam a chamada. Os *threads* desse sistema são, portanto, *threads distribuídos*. A sinergia surge porque é possível generalizar a metáfora de apresentação usada pelos depuradores simbólicos – e é justamente isso que vamos fazer.

Ao invés de apresentarmos o sistema distribuído como uma coleção de processos, cada qual dotada de seu conjunto de *threads* locais, vamos apresentá-lo como um único "processo virtual", composto por uma coleção de *threads* distribuídos. A interação com esses *threads* distribuídos segue os padrões usuais de interação em depuradores simbólicos e, na medida do bom senso, nós procuramos preservar também o fluxo de trabalho descrito anteriormente. A Figura 4-8 ilustra a montagem de um *thread* distribuído dentro dessa metáfora. Na figura são apresentados três nodos e uma cadeia de chamadas remotas que cruza esses três nodos.

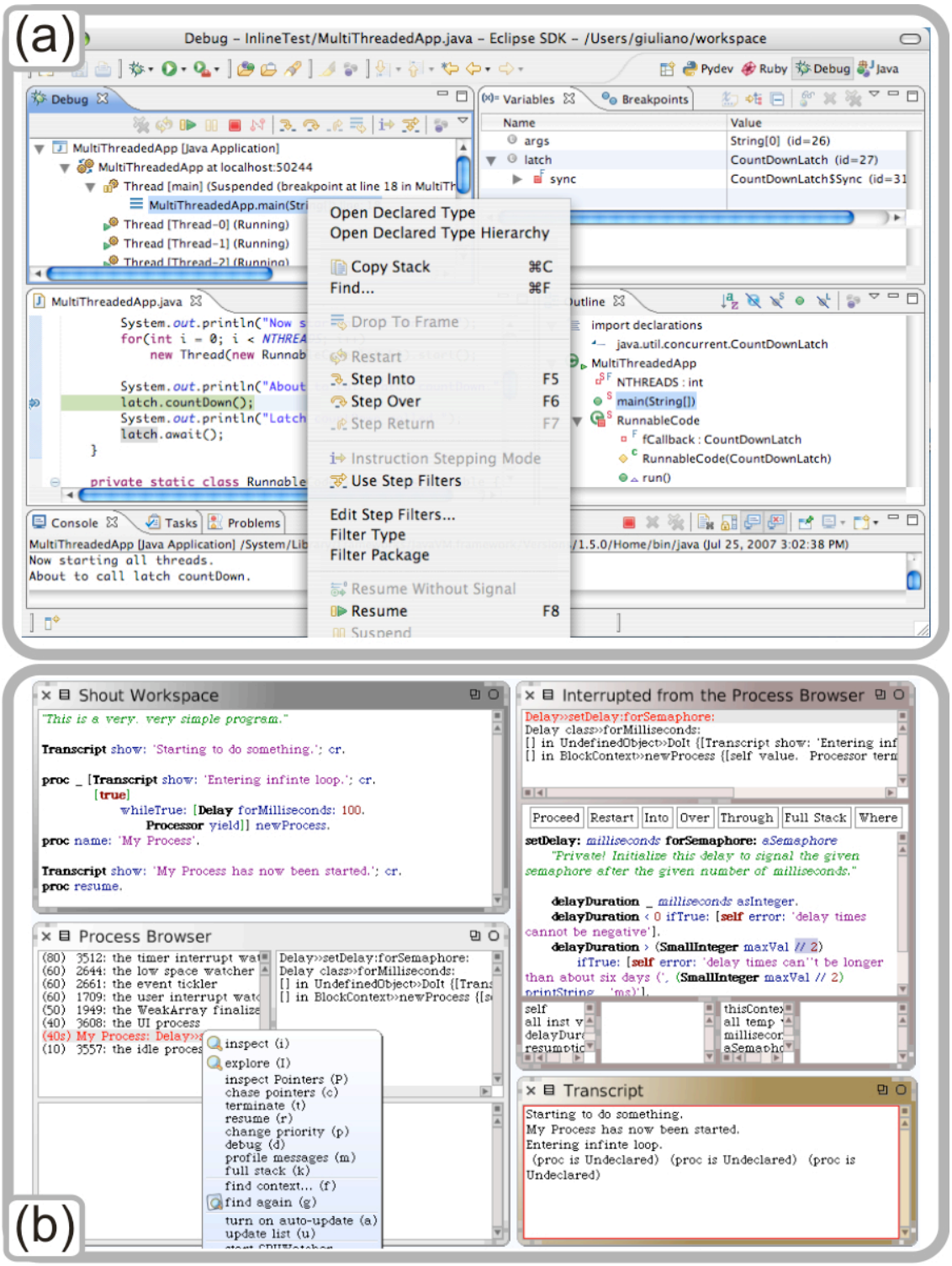

Figura 4-6. Depuradores simbólicos: (a) Eclipse *Java Development Tools* (JDT), (b) *Squeak/Smalltalk*

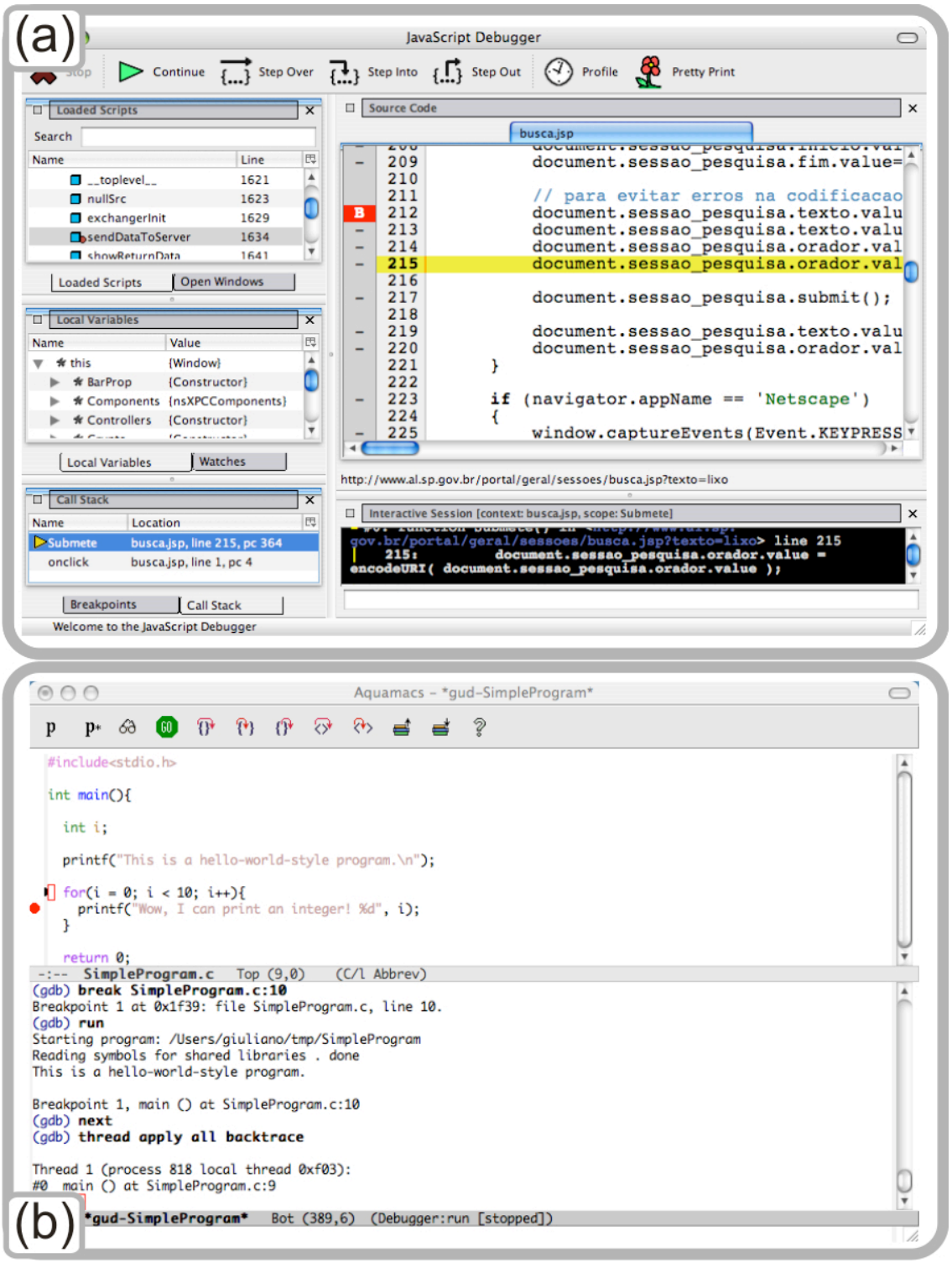

Figura 4-7. Depuradores simbólicos: (a) *Venkman JavaScript Debugger* e (b) *GDB/Aquamacs*.

A cadeia de chamadas é mapeada em um conjunto de *threads* locais, um por nodo. Cada *thread* local encontra-se representado na Figura 4-8 junto com a sua pilha de chamadas **(1)** e seus respectivos quadros de ativação. Os quadros de ativação mais claros estão localizados no código da aplicação, enquanto que os quadros mais escuros, no código do middleware. A representação metafórica do *thread* distribuído é construída tomando-se a sua fotografia no instante ilustrado **(2)**, removendo os quadros de ativação do middleware e concatenando a seqüência de quadros de ativação resultante numa única "pilha virtual" **(3)**.

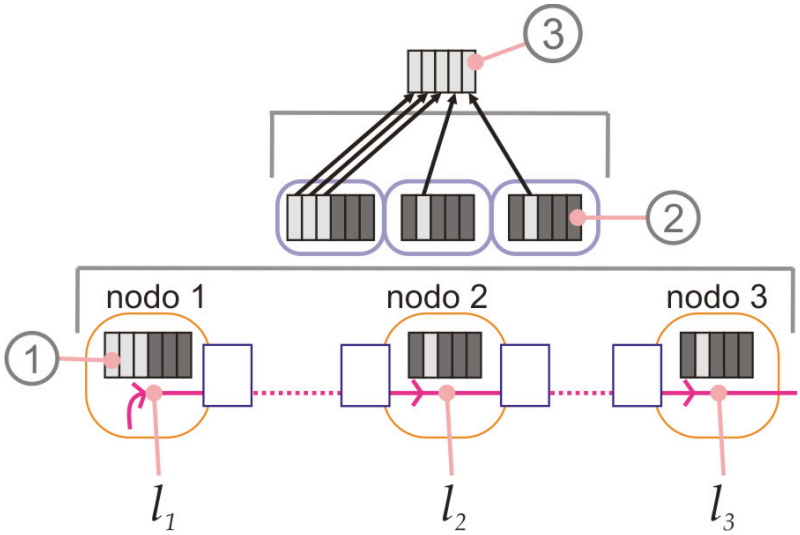

Figura 4-8. Um *thread* distribuído e sua pilha "virtual".

4.3.4 Valor dos *threads* distribuídos como ferramenta de depuração

Threads distribuídos podem ser úteis como ferramentas de depuração por uma série de razões. A primeira, e mais óbvia delas, foi discutida na seção anterior: a representação do sistema como um grande processo virtual, composto por *threads* distribuídos, elimina a discrepância de abstrações existente entre depuradores simbólicos e sistemas de objetos distribuídos. Isso, por sua vez, contribui com uma redução importante do efeito labirinto.

Uma outra conseqüência dessa representação, talvez menos óbvia, é que *threads* distribuídos representam de forma bastante natural as relações causais entre os eventos num sistema distribuído. Particularmente, se tomarmos uma fotografia $\{l_1,...,l_n\}$ de um *thread* distribuído T num instante t, é seguro dizer que os *threads* locais de índice menor influenciam causalmente os *threads* locais de índice maior. Isso implica que um *thread* distribuído nos permite identificar quais os *threads* locais causalmente relacionados numa execução distribuída, algo que não é possível fazer com uma mera coleção de depuradores simbólicos capazes de operar remotamente.

A correlação causal provida pelos *threads* distribuídos nos ajuda a identificar *deadlocks* distribuídos. Afinal, se sabemos quais são os *threads* distribuídos num instante *t*, então podemos determinar se há espera circular entre esses *threads*. De fato, conforme veremos no Capítulo 5, todas essas possibilidades foram exploradas por nossa ferramenta de depuração.

4.3.5 Re-execução determinística

O último aspecto que gostaríamos de discutir a respeito dos *threads* distribuídos está relacionado a possíveis vantagens na re-execução determinística. Em sistemas baseados em passagem de mensagens, *threads* locais tipicamente consomem as mensagens (e seus correspondentes históricos de eventos) de uma fila de eventos, produzindo, durante o restante da execução, alterações no estado do programa que são condizentes com esse histórico. Durante a re-execução determinística, garantir a vinculação do histórico causal correto a cada *thread* local implica em garantir que as mensagens serão sempre consumidas pelos *threads* locais corretos, na ordem correta. Além disso, a re-execução determinística também exige que as condições de corrida de dados e sincronização entre os *threads* que consomem (e não consomem) as mensagens sejam reproduzidas de forma a respeitar as relações causais da execução original.

Num sistema de objetos distribuídos *multithreaded* e baseado em chamadas síncronas e bloqueantes, no entanto, a vinculação do histórico causal ao *thread* local correto acontece em decorrência do funcionamento do mecanismo. Para entender como, vamos imaginar um sistema de objetos distribuídos hipotético, composto por apenas dois nodos, um cliente e um servidor. O cliente faz apenas duas requisições, concorrentes, de forma completamente determinística (i.e., duas requisições sempre com os mesmos históricos de eventos). A rede, no entanto, pode reordenar a entrega de mensagens. Além disso, o *pool* de *threads* do lado do servidor pode mapear uma requisição num *thread* local distinto, de forma não-determinística, a cada execução.

Do lado do servidor, há exatamente uma condição de corrida de dados possível (lembre-se da Seção 2.6.1) entre os *threads* locais que tratam essas duas requisições. Duas possíveis execuções deste cenário são mostradas nas Figura 4-9 (a) e (b). A diferença entre os cenários (a) e (b) é que, no cenário (a), as requisições iniciadas por l_1 e l_2 são tratadas, respectivamente, por l_3 e l_4. No cenário (b), no entanto, essas mesmas duas requisições são tratadas por l_4 e l_3. A condição de corrida que iremos supor no servidor é uma em que ambos os *threads*, ao tratarem a requisição, lêem e depois escrevem em certa variável compartilhada.

Vamos argumentar agora que essas execuções são equivalentes se preservamos a ordem de acesso, pelos *threads* distribuídos T e T', à memória compartilhada. O argumento sai da comparação dos históricos de eventos nos cenários (a) e (b). Lembre-se, da Seção 2.3, que as

decisões tomadas por um programa (ou *thread* de execução) num instante qualquer dependem única e exclusivamente de seu histórico causal até esse instante. Isso significa que se dois *threads* têm históricos causais idênticos em execuções distintas então esses *threads* são, essencialmente, equivalentes [121, 144]. Voltando ao exemplo de sistema distribuído e à Figura 4-9, sejam:

1. C e C' os históricos causais das requisições iniciadas por l_1 e l_2, respectivamente;
2. O o histórico causal da variável compartilhada e acessada por l_3 e l_4, antes do primeiro acesso (por l_3 ou l_4, durante o tratamento das requisições em questão);
3. w_T e $w_{T'}$ os eventos de escrita produzidos pelos *threads* locais que participam nos distribuídos T e T', respectivamente;
4. r_T e $r_{T'}$ os eventos de leitura produzidos pelos *threads* locais que participam nos distribuídos T e T', respectivamente.

Agora basta notar que o conjunto de eventos que determinam as decisões de l_3 logo após a leitura da região compartilhada no cenário (a) é idêntico ao conjunto de eventos que influenciam as decisões de l_4 no cenário (b). Portanto, o comportamento e as alterações produzidas nos dois cenários serão indistintos. Isso também vale para l_4 no cenário (a) e l_3 no cenário (b). Isso implica que, do ponto de vista do cliente, as duas execuções são indistinguíveis e produzem respostas idênticas. Portanto, a execução no cliente também produz resultados idênticos, resultando numa execução distribuída equivalente.

Este argumento não é uma prova formal, mas é suficiente para entendermos porque não precisamos nos preocupar com a ordem de entrega das mensagens. Uma forma mais intuitiva de pensar no problema seria encarar chamadas remotas como equivalentes a chamadas locais, exceto pelo fato de que pode haver um atraso potencialmente elevado na transferência de controle. Uma terceira forma de tentar entender o que acontece é imaginar que, enquanto no sistema baseado em passagem de mensagens o histórico causal do *thread* que consome a mensagem também influencia no restante do processamento, num sistema de objetos distribuídos os *threads* locais que penetram nos objetos remotos são providos pelo *middleware*, apresentando um histórico causal "limpo" até antes do consumo da mensagem. Portanto, não importa qual *thread* consome a mensagem: o histórico causal transportado para dentro do objeto remoto será sempre equivalente.

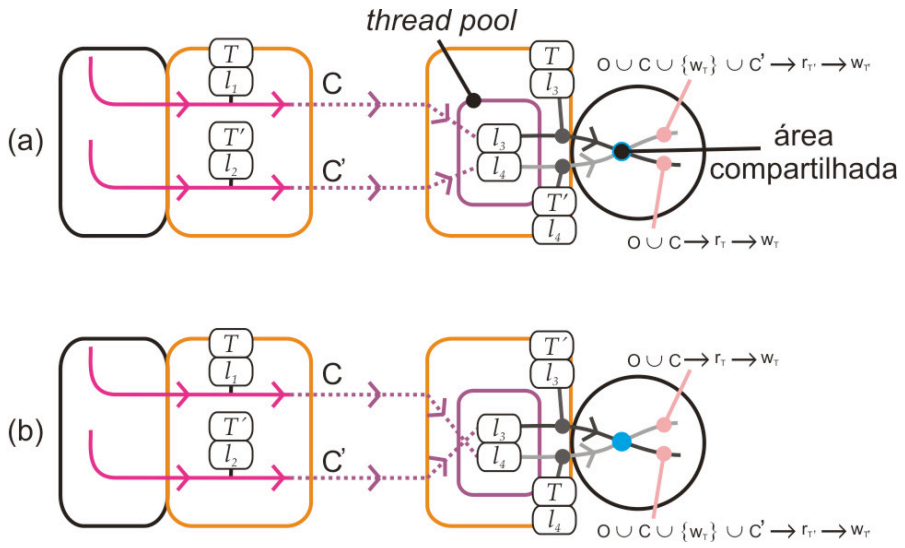

Figura 4-9. Mapeamentos distintos e equivalência de históricos causais durante (a) gravação e (b) reprodução.

De fato, na ausência de falhas, não há uma diferença semântica apreciável entre uma chamada remota numa rede de alta latência e um *thread local* suspenso pelo escalonador do sistema operacional, por um tempo elevado, durante uma chamada local. A reprodução das condições de corrida de fronteira em cada nodo, portanto, é suficiente para garantir uma re-execução correta.

Em resumo, a reprodução da execução de *threads* distribuídos, potencialmente:

1. É mais simples, já que não precisamos garantir a ordem de entrega das mensagens. Boa parte dos sistemas de middleware que conhecemos manipulam soquetes diretamente. Reproduzir a ordem de entrega das mensagens implica em instrumentar a camada de transporte [33].
2. Pode acarretar menos *overhead*:
    a. em alguns casos, é possível garantir a re-execução determinística instrumentando-se apenas o código da aplicação (e não o código do middleware),
    b. as restrições de mapeamento entre *threads* locais e distribuídos são relaxadas, permitindo um maior número de operações concorrentes durante a reprodução.

Fizemos algumas experiências simples com re-execuções distribuídas envolvendo condições de corrida de sincronização e uma abordagem baseada em ordem semelhante à do *JaRec* [71] e, de fato, a abordagem é viável em configurações simples. Há, no entanto, uma série de problemas que precisariam ser superados antes que pudéssemos chegar em uma técnica utilizável. Entre esses problemas, podemos destacar:

- O *pool* de *threads* pode se exaurir durante a reprodução, resultando em *deadlocks*: lembre-se que o mecanismo de reprodução suspende os *threads* que chegam fora de ordem em uma

operação de sincronização. Se mensagens suficientes chegarem fora de ordem, todos os *threads* de um servidor podem ser comprometidos antes que a mensagem correta possa ser consumida, levando a um *deadlock*. Este problema parece corrigível por uma pré-análise dos traços coletados durante a gravação e subseqüente dimensionamento dos *pools* durante a reprodução.

- Políticas não-determinísticas de ativação de objetos podem resultar em erros: um ativador de serventes não-determinístico, por exemplo, poderia impossibilitar o acesso a certas regiões de memória compartilhada durante a reprodução, resultando em erros.
- Falta de sincronização no código do middleware: chamadas não-sincronizadas ao código do middleware podem provocar condições de corrida de sincronização e dados que alteram o estado do middleware de forma não-determinística, resultando na subseqüente falha da reprodução.
- Outras fontes de indeterminismo: cada plataforma de middleware pode introduzir suas próprias fontes de indeterminismo. A complexidade relativa e o desempenho da técnica, portanto, dependem da plataforma.

4.4 Atividades de apoio – gerenciamento de processos e interação remota

Conforme afirmamos anteriormente, nas Seções 3.2.1 e 4.3.3, um ponto importante na depuração cíclica e no fluxo de trabalho (*workflow*) do usuário de um depurador simbólico é o lançamento da aplicação. A maior parte dos ambientes de desenvolvimento atuais permitem que o usuário lance suas aplicações com muito pouco esforço uma vez que alguns parâmetros tenham sido configurados. De fato, essa atividade é considerada tão importante, que a plataforma Eclipse disponibiliza um arcabouço extensível para o desenvolvimento e integração de novos mecanismos de lançamento [209].

Mantendo a nossa metáfora de um "grande processo virtual e distribuído" que pode ser controlado como um processo local, decidimos, desde cedo no desenvolvimento deste trabalho, que o lançamento simplificado (ou "de um clique só") de um sistema distribuído estaria no escopo de nossas atividades. Além disso, gostaríamos também de centralizar a interação com os processos remotos que compõem o sistema distribuído, permitindo, por exemplo, que o usuário interaja com os fluxos de entrada/saída dos consoles desses processos, facilitando a interação e o acesso a informações, além de evitar o uso dos desajeitados *logs* locais quando isso não for realmente necessário. O gerenciamento de processos e a detecção de falhas são também elementos necessários para uma operação mais simples do depurador, como veremos mais adiante no Capítulo 5.

4.5 Sumário

Sistemas de objetos distribuídos são tradicionalmente baseados em chamadas síncronas e bloqueantes. O modelo é conveniente por mascarar parcialmente a distribuição do sistema, livrando o programador da natureza assíncrona da passagem de mensagens. Além disso, o mecanismo integra-se bem às linguagens de programação tradicionais, provendo um modelo de programação uniforme. Apesar dos benefícios, no entanto, as chamadas síncronas e bloqueantes têm sua parcela de inconvenientes, como a indução do uso de *threads* e paralelismo não-estruturado em todos os nodos do sistema e a exacerbação de problemas de composição com o aumento do escopo dos *threads* (que agora passam a abranger o sistema distribuído inteiro), principalmente relacionados ao uso de travas [117].

Ferramentas de depuração que atendam às necessidades específicas dos desenvolvedores de sistemas concorrentes são escassas e, nesse aspecto, os sistemas de objetos distribuídos não são exceção. Alguns sistemas de middleware dispõem de ferramentas de depuração especificamente adaptadas a eles, como é o caso do OCI OVATION [150], para o ORB CORBA TAO [179], e do IBM *Object Level Trace* (OLT) [87], para o servidor de aplicações *WebSphere* [86]. A falta de adequação dessas ferramentas a ambientes heterogêneos e a outros sistemas de middleware limitam a sua aplicabilidade, no entanto, levando a grande maioria dos desenvolvedores de sistemas de objetos distribuídos a recorrerem às ferramentas de ampla disponibilidade – como o comando *print* e suas variantes (como arcabouços de *logging*) e os depuradores simbólicos.

O comando *print* não é exatamente uma ferramenta de depuração, mas um mecanismo que viabiliza a comunicação de informações textuais não-estruturadas. Como ferramenta de depuração, o comando *print* é tão flexível quanto rudimentar, onerando o desenvolvedor com o projeto e a implementação dos mecanismos de seleção, filtragem, coleta e análise de informações de tempo de execução, bem como pela conversão dessas informações em informações textuais e visuais úteis.

Depuradores simbólicos, por outro lado, são ferramentas bastante mais sofisticadas. Tratam-se de ferramentas que mapeiam o estado da aplicação em representações reificadas dos elementos do ambiente de execução da linguagem de programação na qual foi escrita o programa sendo depurado. Normalmente, essas ferramentas permitem que o usuário suspenda a execução da aplicação em pontos-chave (pelo uso de *breakpoints*) e, a partir daí, navegue pela execução em incrementos pequenos, observando o fluxo de controle no próprio texto do programa e inspecionando o estado de objetos de tempo de execução (pilhas de chamada, variáveis locais, variáveis globais, etc.) em representações auxiliares.

Depuradores simbólicos, embora úteis, apresentam uma série de limitações no contexto dos sistemas de objetos distribuídos, como a inabilidade em reproduzir o comportamento de execuções

concorrentes, a falta de expressividade dos *breakpoints* baseados em predicados locais, a falta de provisão de mecanismos que viabilizem a captura de relações causais, a falta de escalabilidade do mecanismo de visualização e navegação, além da incompatibilidade de abstrações.

Baseados nessas observações e no *workflow* do usuário de um depurador simbólico típico, propusemos uma ferramenta que reifica a execução do sistema distribuído sob a forma de um grande "processo virtual", composto por múltiplos *threads* distribuídos. Essa metáfora segue a metáfora proposta pelas chamadas síncronas e bloqueantes, nivelando as abstrações providas pelo middleware àquelas apresentadas pelos depuradores simbólicos. A representação baseada em *threads* distribuídos não só ajuda a reduzir o efeito labirinto, como também representa de forma natural as relações causais num sistema distribuído. Um dos benefícios imediatos de dispormos dessa correlação é a habilidade de detectarmos *deadlocks* distribuídos, ajudando os desenvolvedores de SODs a lidarem com um problema que ocorre com bastante freqüência e que, muitas vezes, é difícil de detectar e corrigir.

Finalmente, há um potencial de que a re-execução determinística de sistemas de objetos distribuídos baseados em chamadas síncronas e bloqueantes seja mais simples do que no caso dos sistemas baseados em passagem de mensagens. Isso porque uma chamada remota equivale, essencialmente, a uma operação de transferência de controle com atraso elevado. A reprodução das condições de corrida de fronteira em cada nodo, portanto, pode ser suficiente para assegurar o determinismo, sem que tenhamos de nos preocupar com a reprodução na ordem de entrega das mensagens ou com o comportamento do middleware. A relação entre simplicidade e viabilidade, no entanto, ainda deve ser avaliada.

Mesmo com a extensão da metáfora do depurador simbólico aos sistemas de objetos distribuídos, ainda resta uma atividade de bastante importância no *workflow* dos usuários de depuradores simbólicos que nós não cobrimos até então – o lançamento da aplicação. Pela importância a atividade, incluímos no escopo deste trabalho, portanto, o desenvolvimento de um mecanismo que permita o "lançamento de um clique só" do sistema distribuído, após prévia configuração.

5 O *Global On-line Debugger* (G.O.D.)

"In God we trust."

-- *USA National Motto*

No Capítulo 4 introduzimos os *threads* distribuídos e apresentamos uma caracterização formal que descreve, com bastante precisão, o seu comportamento (esperado) na ausência de falhas. Baseados nessa caracterização e nas observações das Seções 4.3.3 e 4.3.4, vamos apresentar, neste capítulo, a ferramenta que construímos para depurar sistemas de objetos distribuídos: o *Global On-line Debugger* (G.O.D.).

O capítulo começa, na Seção 5.1, com uma visão geral da arquitetura da ferramenta e seus principais elementos. Em seguida, na Seção 5.2, partimos para o detalhamento dos mecanismos de representação e rastreio de *threads* distribuídos. Na Seção 5.3 introduzimos o protocolo de rastreio básico, que terá, na Seção 5.3.2, a sua caracterização complementada com considerações sobre técnicas interativas e limitadas. Esse material deve prover bases suficientes para a compreensão dos conceitos que serão apresentados na Seção 5.3.4, que estendem a caracterização de comportamento dos *threads* distribuídos iniciada na Seção 4.3.2 para cenários em que processos, nodos e *links* podem falhar.

A Seção 5.4 descreve um dos dois mais importantes componentes de nossa ferramenta – os agentes locais. Na Seção 5.4.1 apresentamos detalhes da implementação dos agentes locais Java/CORBA e, na Seção 5.4.3, argumentamos que o desenvolvimento de agentes locais para outras linguagens e sistemas de middleware deve ser tão simples, ou mais simples, do que o desenvolvimento do agente local Java/CORBA.

A Seção 5.5 discute o agente central – o maior componente da ferramenta – e suas diversas características. Nessa seção, vamos argumentar que a ferramenta é de fato extensível, expondo em detalhe os requisitos para a acomodação de novas linguagens e plataformas de middleware. Vamos também argumentar que o mecanismo de rastreio utilizado é correto na presença dos tipos mais comuns de falhas. A seção será encerrada com uma discussão da implementação do mecanismo de controle de processos e dos algoritmos de análise automática de dados atualmente disponíveis.

5.1 Arquitetura

O depurador simbólico apresenta, conforme mostra a Figura 5-1, uma arquitetura semelhante à das ferramentas de observação e análise descritas na Seção 3.1.

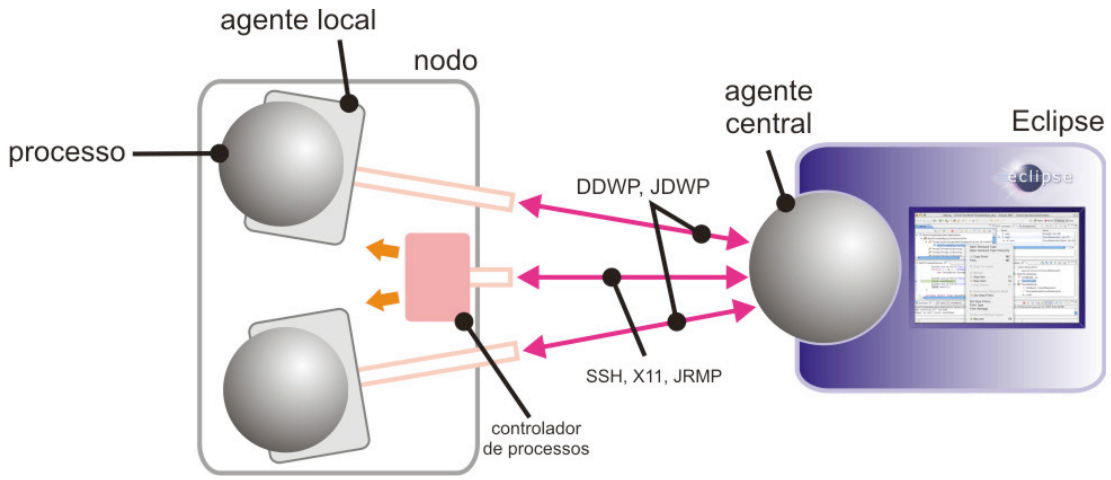

Figura 5-1. Visão geral do GOD e seus principais elementos.

O GOD pode ser divido em quatro grandes elementos:

- **Agentes locais:** compostos pelo maquinário, anexado a cada processo da aplicação, que viabiliza a coleta de dados e a interação com a aplicação em execução. Há um agente local associado a cada processo. Agentes locais serão discutidos em mais detalhe na Seção 5.4.
- **Controlador de processos:** provê um serviço de controle e notificação de eventos relacionados aos processos que executam em nodos remotos. Permite o lançamento de novos processos, a destruição de processos em execução, a transmissão do conteúdo dos *streams* de saída (e outros eventos, como notificações de morte) a clientes interessados e a interação com a entrada padrão dos processos remotos. Há um único controlador de processos por nodo. Controladores de processos serão discutidos em mais detalhe na Seção 5.6.
- **Agente central:** cliente dos agentes locais e controladores de processos. Responsável pela montagem da visão global da execução distribuída a partir das informações parciais transmitidas pelos agentes locais e por reificar o ambiente de execução heterogêneo sob um modelo uniforme.
- **Eclipse:** provê um modelo e um conjunto de arcabouços para a construção de depuradores simbólicos. O modelo de objetos que define as interfaces para os elementos reificados do sistema distribuído é provido, majoritariamente, pelo Eclipse. O papel do Eclipse será discutido na Seção 5.5.2.

5.2 Representação dos *threads* distribuídos

Conforme o que foi dito na Seção 4.3.3, nossa ferramenta de depuração apresenta o sistema distribuído como um grande processo virtual, composto por uma coleção de *threads* distribuídos.

Afirmamos na Seção 4.2.2, no entanto, que depuradores simbólicos são míopes com relação a elementos externos à linguagem de programação ou ambiente de execução para os quais foram originalmente concebidos. Isso significa que precisamos, de alguma forma, completar o conhecimento dos depuradores simbólicos com informações adicionais se quisermos atingir nosso intento. A identificação e o rastreio dos *threads* distribuídos é, portanto, o primeiro problema que precisamos resolver.

Seguindo a terminologia estabelecida por Maes [124] no final da década de 80, um depurador simbólico pode ser considerado um meta-sistema que tem como domínio os processos que controla (vamos discutir isso em mais detalhe na Seção 5.3.1). Particularmente, esse meta-sistema tem, tipicamente, acesso integral e irrestrito ao estado desses processos. Uma idéia que surge naturalmente neste contexto é que poderíamos tentar aproveitar o maquinário oferecido pelos depuradores simbólicos para identificar, por meio da inspeção das transições de estados nos processos individuais, as transições das fotografias que compõem o *thread* distribuído.

O primeiro problema com essa abordagem é que a seqüência de estados locais que caracteriza uma mudança de fotografias pode ser complexa, não-determinística e acoplada à implementação do middleware. O segundo problema é que depuradores simbólicos não foram projetados para a captura contínua de estados, mas para a observação de seções pontuais do conjunto total de estados. Sem o cuidado adequado, a abordagem poder levar à perda de certas transições locais e à conseqüente perda de transições de fotografias. Além disso, há um problema extra, que será discutido na Seção 5.2.1: o acesso ao estado da aplicação pelo depurador simbólico costuma ser muito caro.

O primeiro problema pode ser resolvido, ou ao menos amenizado, com a criação de estruturas, em tempo de execução, que possam ser inspecionadas com mais facilidade e portabilidade que o estado do middleware. Essas estruturas pertencem, seguindo a divisão entre meta-nível (depurador) e nível base (processo depurado) estabelecida anteriormente, ao nível base. Nossa implementação conta com estruturas de nível base calcadas em uma representação dos *threads* distribuídos também de nível base, ilustrada na Figura 5-2.

Em nossa representação, atribuímos a cada *thread* local um identificador global único, de 48 bits, composto por duas partes. A primeira parte, de 16 bits, identifica o processo ao qual pertence o *thread* local. A segunda parte, de 32 bits, é retirada de um contador seqüencial local. A atribuição do identificador de cada processo (de 16 bits) é de responsabilidade do agente central. Sempre que um *thread* local que participa de uma única fotografia trivial (Seção 4.3.2) inicia uma cadeia de chamadas remotas, seu identificador é propagado, "marcando" todos os *threads* locais subseqüentes.

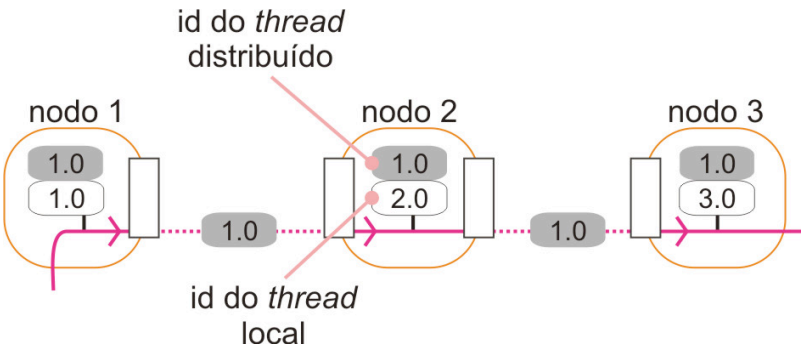

Figura 5-2. Propagação do identificador de um *thread* local.

A Figura 5-2 mostra um *thread* local propagando o seu id (1.0) por uma cadeia de chamadas que cruza três nodos.

A discussão anterior implica que a todo *thread* local *l* encontram-se associados dois identificadores de 48 bits:

- o primeiro, id_1, identifica o próprio *thread* local,
- o segundo, id_2 (cinza-escuro na Figura 5-2), identifica o *thread* local que é a **base** do *thread* distribuído dono da **maior** fotografia de que *l* faz parte. Em particular, é possível que $id_1 = id_2$.

A atribuição dinâmica de id_1 e id_2 a cada *thread* local é feita mediante a instrumentação do código da aplicação, e será discutida em mais detalhe na Seção 5.4.1. No ambiente de execução Java, nós encapsulamos o mapeamento entre *threads* locais e seus identificadores dentro de um repositório que, internamente, utiliza um mecanismo de *thread-specific storage* (TSS) [180]. Esse repositório, por sua vez, é um *Singleton* [69]. Para determinar os identificadores de um *thread* local *l*, o depurador:

1. obtém uma referência ao *Singleton*;
2. utiliza a referência obtida em (1) e o *thread* local *l* para enviar uma mensagem (`getIds()`) ao repositório.

A criação dessa estrutura de tempo de execução (o repositório) desacopla o protocolo de inspeção de estado empregado pelo depurador da implementação do middleware já que, seja qual for o middleware utilizado, o depurador pode sempre recorrer ao repositório para determinar quais *threads* locais participam em quais fotografias.

Note, no entanto, que esse protocolo é um protocolo *pull*. Resta ainda determinar em quais momentos da execução o depurador deve consultar o repositório. Isso será discutido na Seção 5.4.1.

5.2.1 Adendo sobre re-execução e desempenho

Outra vantagem em termos informações a respeito de *threads* distribuídos acessíveis pelo nível base advém do fato de que essas informações viabilizam a implementação do mecanismo de re-execução descrito na Seção 4.3.5 de forma bastante direta. Para tanto, basta garantirmos que:

1. as condições de corrida de sincronização entre *threads* distribuídos sejam sempre reproduzidas;
2. a atribuição de identificadores respeite os relacionamentos pai-filho [116] em *threads* locais criados pela aplicação.

É claro que os problemas que descrevemos na Seção 4.3.5 (e que não fomos capazes de resolver) ainda se aplicam aqui, mas se tivéssemos optado por manter a representação completamente no depurador (meta-nível), todas as operações de sincronização teriam de ser mediadas por ele. Como o depurador costuma residir num processo separado (no nosso caso, o depurador pode residir em um nodo separado), o custo dessa mediação se torna proibitivo. De maneira geral, a injeção de código de instrumentação é mais simples e mais eficiente quando queremos interferir de forma especializada na execução da aplicação. É também por isso que informações sobre o *thread* distribuído no nível base são interessantes – o código de instrumentação que executa junto com a aplicação é também parte do nível base. Se quisermos, portanto, executar algoritmos distribuídos de depuração de forma eficiente (implementados junto com a aplicação), precisamos dessa informação.

5.3 Rastreio

Rastrear um *thread* distribuído equivale a capturar a sua seqüência de fotografias. Nosso depurador supõe que a seqüência de fotografias produzida por um *thread* distribuído, na ausência de falhas, obedece à descrição apresentada na Seção 4.3.2. Sob essa perspectiva, rastrear um *thread* distribuído corresponde a rastrearmos a movimentação de sua *cabeça*. Para tanto, desenvolvemos um mecanismo de rastreio simples, ilustrado na Figura 5-3.

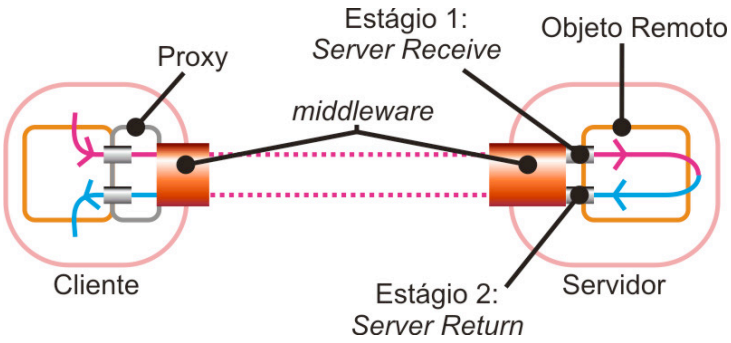

Figura 5-3. Protocolo de rastreio de dois estágios.

O mecanismo básico gera apenas dois tipos de eventos, que serão descritos a seguir. Seja *C* uma execução distribuída e $l \in L_C$ um *thread* local. Seja ainda $P = \{e_1,...,e_k\}$ um particionamento de localização de $stk(l,t) = \{q_1,...,q_j\}$ num instante $t \in [t_l^s, t_l^d)$. Suponhamos, sem perda de generalidade, que não existe um quadro de tratamento em e_k no instante *t*. Então:

- *Server Receive* – gerado no instante em que um quadro de tratamento q_{j+1} é empilhado em e_k. Contém os valores de id_1 e id_2 associados ao *thread* local *l* e o índice *j* do quadro de tratamento empilhado em $stk(l,t)$, além de informações específicas da implementação do agente local, que serão discutidas na Seção 5.4.1.
- *Server Return* – gerado no instante em que um quadro de tratamento $q \in e_k$ é desempilhado. Contém os valores de id_1 e id_2 associados ao *thread* local *l*.

Note que a descrição do protocolo é um tanto quanto informal. Quando dizemos que o evento *Server Receive* é gerado "no instante" em que um quadro de tratamento q_{j+1} é empilhado em e_k, queremos dizer que esse evento é gerado no menor instante $t + \xi$, $\xi \in \mathbb{R}$ e $\xi > 0$, tal que:

- no instante *t* não há nenhum quadro de tratamento em e_k,
- no instante $t + \xi$ há um quadro de tratamento em e_k.

O mesmo vale para o evento *Server Return*. Na prática, esses eventos não são gerados exatamente no instante em que os quadros de tratamento são empilhados ou desempilhados, já que isso seria impossível. Vamos mostrar na Seção 5.3.2, no entanto, que nosso esquema de notificação garante algo equivalente a uma notificação instantânea, do ponto de vista do *thread* local *l*.

5.3.1 Reflexão e técnicas interativas

Uma das principais possibilidades oferecidas por uma ferramenta de depuração interativa é a manipulação de entidades de tempo de execução "vivas". Essas "entidades vivas" são, na verdade, representações abstratas, ou reificações, de objetos semanticamente significativos na linguagem de programação e/ou ambiente de execução da aplicação-alvo do depurador. Maes, em seu trabalho seminal sobre reflexão computacional [124], define um sistema computacional como um sistema baseado em computadores que tem como propósito responder a perguntas e/ou apoiar ações em algum domínio. Esse sistema computacional é dito ser *sobre* seu domínio e, tipicamente, incorpora estruturas de dados internas que representam esse domínio. Ainda segundo Maes, um sistema computacional é *causalmente conexo* com seu domínio se suas estruturas internas e o domínio que

representam estiverem ligados de tal forma que uma mudança em um dos lados sempre leva a um efeito correspondente no outro lado.

O exemplo citado por Maes é o de um braço robótico e seu sistema de controle, ilustrados na Figura 5-4. O sistema de controle possui uma representação interna do estado (posição, estado operacional) do braço robótico. Se o sistema de controle do braço for causalmente conectado ao seu domínio, então modificações na representação do braço devem se traduzir em modificações no posicionamento real do braço e, de maneira semelhante, modificações no posicionamento real do braço (possivelmente causadas por agentes externos) devem se traduzir em modificações equivalentes na representação interna ao sistema de controle.

Um sistema computacional é dito ser um *meta-sistema* quando tem como domínio um segundo sistema computacional. Note que, como no caso do sistema de controle do braço robótico, um *meta-sistema* também incorpora estruturas de dados internas que representam o seu domínio, o *sistema-alvo*. As reificações mencionadas no início desta seção são, justamente, essas estruturas de dados.

Ainda na terminologia de Maes, um *meta-sistema* é *reflexivo* quando tem acesso a estruturas que representam aspectos dele próprio. Isso significa que um sistema reflexivo é um *meta-sistema* que tem a si próprio como domínio (sistema-alvo). Claro que isso não implica que um sistema reflexivo produz apenas auto-manipulações – isso dificilmente serviria a algum propósito útil. Ao invés disso, um sistema reflexivo tipicamente manipula ao menos dois domínios – um representando ele próprio, e um outro, representando alguma entidade externa ao sistema (por exemplo, o braço robótico).

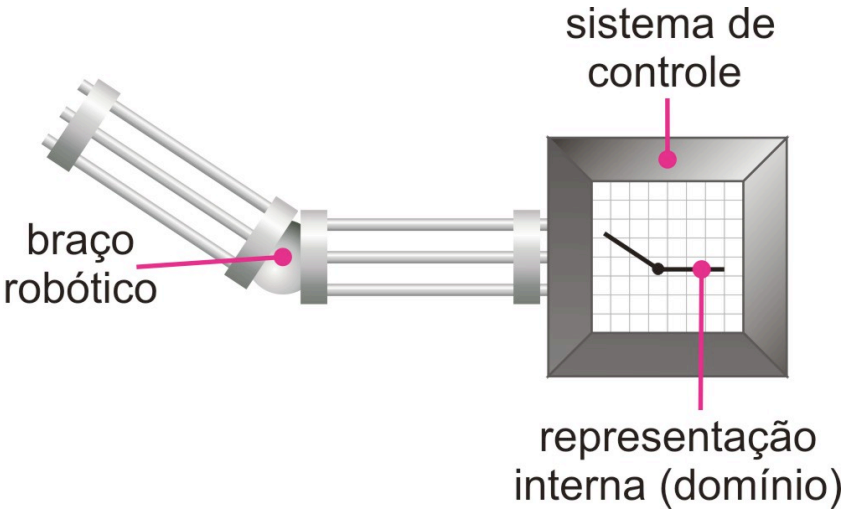

Figura 5-4. Braço robótico e seu sistema de controle.

A questão importante, no escopo deste trabalho, é que uma aplicação tipicamente não tem sequer como determinar se existe um depurador simbólico ligado a ela, enquanto que o depurador simbólico tem acesso irrestrito ao estado da aplicação. Isso quer dizer que a composição depurador-

depurado não é um sistema reflexivo, mas sim um *meta-sistema* (depurador) ligado a um sistema (depurado). Além disso, é importante notar que o depurador é causalmente conectado ao seu domínio; i.e., alterações no depurado refletem-se em alterações nas reificações, e vice-versa.

5.3.2 Interação e técnicas *on-line* limitadas

Estruturas de tempo de execução são inerentemente dinâmicas. Quando depurador e depurado residem em processos distintos (caso da maior parte dos depuradores), as reificações às quais o depurador tem acesso são apenas aproximações das estruturas reais. A questão, como em todo tipo de replicação de estado, é que essas aproximações estão sujeitas a problemas de consistência. Se não tomarmos medidas que limitem o desvio entre o estado real da aplicação e o estado das representações com as quais interage o depurador simbólico, algumas operações interativas podem ficar comprometidas.

Em nossa experiência com o estudo da implementação de depuradores simbólicos (entre eles o depurador para Java incluso no Eclipse [57], o cliente para o GDB incluso no CDT [56], o JDB [203] e o nosso próprio depurador) as soluções para essa questão sempre recaem em alguma das seguintes alternativas:

1. Permite-se que o estado da aplicação desvie arbitrariamente da aproximação observada pelo depurador simbólico, sabendo que certas operações interativas podem falhar porque a entidade à qual a operação é direcionada não existe mais, ou, de maneira mais geral, sofreu uma transição de estado que não permite que a operação seja realizada.
2. Sincroniza-se aplicação e depurador em certas transições de estado, consideradas chave, de tal forma a eliminar, ou ao menos reduzir, a possibilidade de que ocorram falhas decorrentes de condições de corrida como as descritas em (1).

A escolha por uma dessas abordagens depende muito da operação, sendo governada por duas forças principais: o significado (e a gravidade) de uma "falha" e o custo de desempenho trazido pelas sincronizações.

Caso 1: Breakpoints na Plataforma de Depuração Java (JPDA)

A plataforma de depuração Java (JPDA) [201] disponibiliza um conjunto de componentes de software para o desenvolvimento de depuradores simbólicos e ferramentas de monitoramento e análise para Java. A plataforma é composta por três componentes principais: um servidor (*backend*) de depuração, um protocolo de comunicação aberto, o *Java Debug Wire Protocol* (JDWP), e um conjunto de interfaces também abertas, a *Java Debug Interface* (JDI). A Figura 5-5 dá uma visão geral da JPDA.

A JDI disponibiliza um conjunto de reificações (chamadas espelhos [223]) de elementos do ambiente de execução Java aos seus clientes, os depuradores. Operações conduzidas em objetos disponibilizados pela JDI são traduzidas em mensagens JDWP, que são por sua vez enviadas ao servidor de depuração ligado à máquina virtual remota. O servidor de depuração se encarrega então de executar os comandos de fato. Na implementação da máquina virtual da Sun (a mais usada atualmente), o servidor de depuração (*backend*) é um plug-in JNI (escrito em C++) que traduz os comandos JDWP em chamadas à *Java Virtual Machine Tools Interface* (JVMTI) [205].

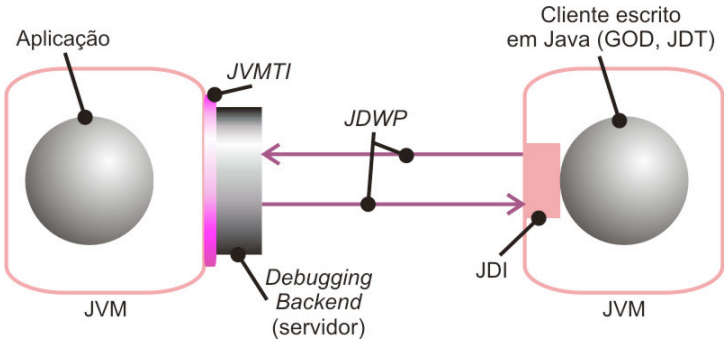

Figura 5-5. Diagrama esquemático da JPDA.

Na JPDA, é impossível posicionar um *breakpoint* no código de uma classe antes que essa classe tenha sido carregada. Suponha, portanto, que queiramos colocar um *breakpoint* no método `bar()` da classe `Foo`. Suponha ainda que nenhuma versão da classe `Foo` tenha sido carregada (lembre-se que em Java é possível que haja múltiplas versões de uma mesma classe, basta que estejam em *class loaders* diferentes). Neste caso é preciso, primeiramente, pedir à JDI que nos notifique sempre que uma classe com o nome `Foo` for carregada. Note, no entanto, que os eventos de notificação de carga de classe vêm de um nodo separado.

Agora suponha uma situação como a da Figura 5-6 (a), em que um *thread* local *l* dispara a carga da classe `Foo`, chamando o método `bar()` logo em seguida.

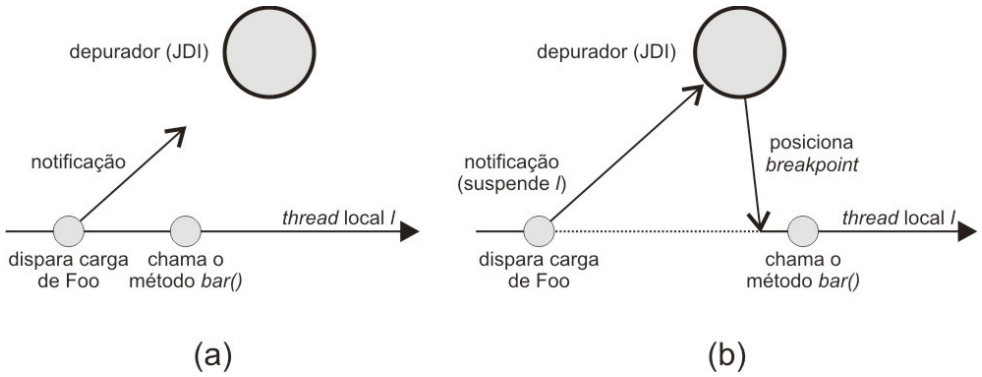

Figura 5-6. (a) Desvio arbitrário e (b) sincronização em transição-chave.

Se permitirmos que o *thread* local *l* continue executando após disparar a carga de `Foo`, então é bastante provável que *l* atinja o método `bar()` antes que o depurador tenha sequer a chance de registrar que `Foo` foi carregada e, por conseqüência, antes que o depurador tenha a chance de posicionar o *breakpoint* em `bar()`. Neste caso em particular, a falha da operação interativa – posicionamento do *breakpoint* – é "grave", no sentido de que é bastante provável que grande parte *breakpoints* posicionados pelo usuário simplesmente não funcionem. A situação é corrigida pela sincronização entre depurador e processo depurado em uma transição-chave de estado – a carga da classe `Foo`. Nesse cenário, mostrado na Figura 5-6 (b), o *thread* local *l* é suspenso (linha tracejada) até que o depurador tenha a oportunidade de registrar a transição de estado e posicionar o *breakpoint*.

Caso 2: *Suspend* em threads locais

Uma modalidade de operação interativa bastante comum em depuradores simbólicos é a suspensão de *threads* em execução. Na suspensão de um *thread* local, o usuário escolhe um *thread* ativo da lista de *threads* em execução e solicita a sua suspensão. As razões que levam o usuário a suspender um *thread* são diversas – eliminar suspeitas de *deadlocks*, determinar a atividade em curso pelo *thread* local, suspender um servidor antes que certa requisição seja tratada, etc.

A questão aqui diz respeito ao fato de que um *thread* em execução pode terminar a qualquer instante. Se a entrega de eventos de término de *threads* locais não for entregue de maneira síncrona, então é possível que a situação mostrada na Figura 5-7 (usuário requisitar a suspensão de um *thread* que não existe mais) aconteça.

Essa falha, no entanto, não é uma falha "grave". A JDI vai apenas lançar uma exceção, que provavelmente nem será exibida ao usuário, informando que o *thread* não existe mais, e sua representação gráfica será removida da interface. Nós poderíamos comunicar o evento de término de forma síncrona, impedindo *l* de terminar antes que o depurador tenha a oportunidade de registrar a transição de estado. Dessa forma, seria possível garantir que todos os *threads* mostrados pela interface gráfica estão de fato ativos. Note que isso só reduz, mas não elimina, a chance de falhas, já que o *thread* local pode deixar de existir entre o instante em que o usuário clica o botão do mouse e o evento é processado, por exemplo. O custo de desempenho e complexidade não justifica, portanto, o benefício.

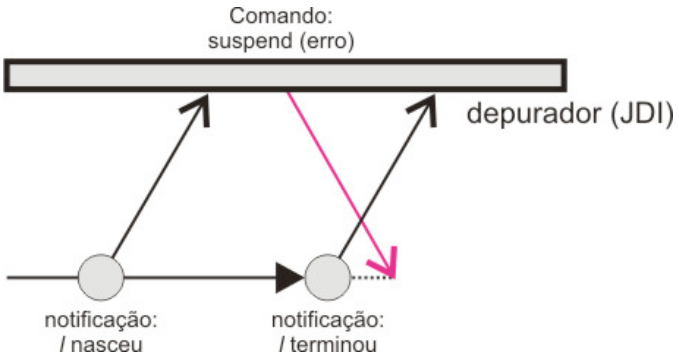

Figura 5-7. *Suspend* falhando porque o conhecimento do depurador não é atual.

5.3.3 Rastreio síncrono e assíncrono: s*uspend* em *threads* distribuídos

Nosso depurador deve oferecer ao usuário facilidades semelhantes àquelas oferecidas por depuradores simbólicos tradicionais. A suspensão de *threads* em instantes arbitrários está, certamente, entre uma dessas facilidades. Suspender um *thread* distribuído equivale a suspender a sua cabeça. Isso implica que se quisermos permitir que usuário suspenda o *thread* distribuído em um instante arbitrário então precisamos saber, num instante também arbitrário, qual *thread* local é a cabeça.

Conforme discutimos na seção anterior, temos, essencialmente, duas alternativas para o rastreio da posição da cabeça – uma *on-line* e não-limitada (rastreio assíncrono) e uma *on-line* e limitada (rastreio síncrono). Seguindo as diretrizes básicas para o desenvolvimento de sistemas distribuídos escaláveis, procuramos, inicialmente, desenvolver um mecanismo de rastreio assíncrono.

O primeiro problema com o rastreio assíncrono da cabeça é que ele está sujeito às inconsistências de observações descritas na Seção 2.2. Isso significa que os eventos produzidos pelo mecanismo de rastreio do início da Seção 5.3 podem ser processados em uma ordem diferente da ordem em que de fato ocorreram, como mostra a Figura 5-8.

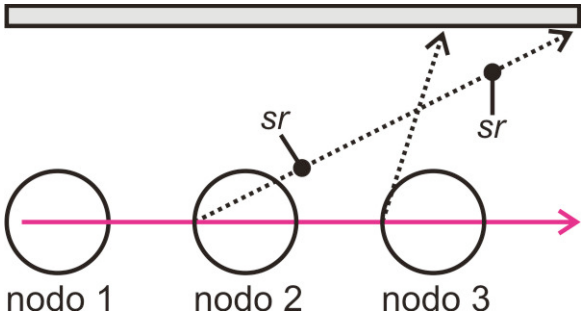

Figura 5-8. Eventos do tipo *server receive* sendo observados em uma ordem inconsistente.

Alternativa 1: Relógios escalares

Os relógios escalares, discutidos na Seção 2.3, servem justamente para ajudar no estabelecimento dos vínculos de causa e efeito entre eventos em uma execução distribuída. Particularmente, isso parece suficiente para nossos propósitos, já que mudanças de fotografias num *thread* distribuído são, na ausência de falhas, sempre causalmente relacionadas. O apelo dos relógios escalares é, justamente, a sua simplicidade e baixo custo computacional (processamento, espaço armazenado e *overhead* adicional em cada mensagem). A idéia seria atribuirmos um relógio escalar a cada processo e, a partir daí, usarmos os *timestamps* para ordenarmos as fotografias.

O grande problema com a abordagem baseada puramente em relógios escalares é que esse tipo de relógio nos dá uma noção muito fraca a respeito da ordem relativa dos eventos. Particularmente, se houver um único relógio lógico por processo, fica impossível determinar, dados dois eventos *a* e *b* tais que $L(a) < L(b)$, se existe um evento *c* tal que $a \rightarrow c \rightarrow b$, onde $\rightarrow$ representa a relação de causalidade. Isso pode levar a problemas como o ilustrado na Figura 5-9.

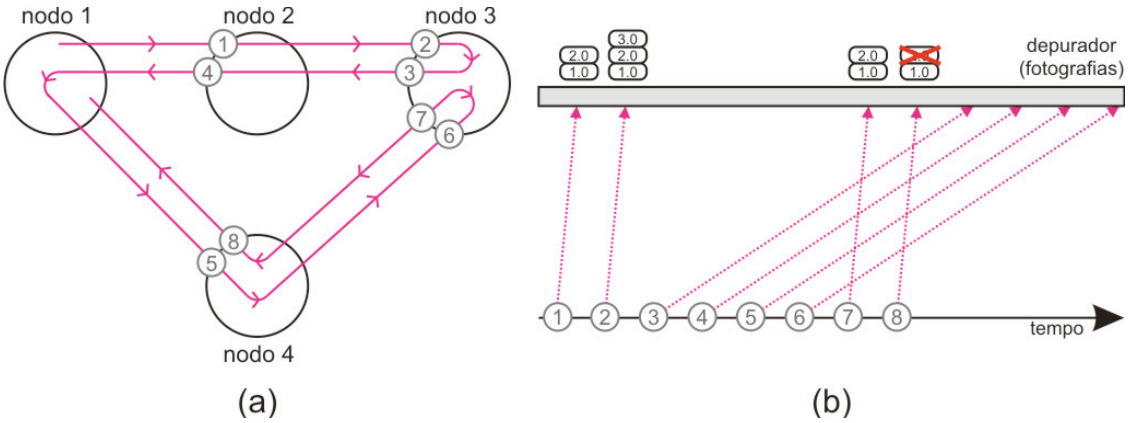

Figura 5-9. (a) Thread distribuído percorrendo uma seqüência de nodos, com eventos demarcados e (b) seqüência de entrega dos eventos ao depurador.

A Figura 5-9 (a) mostra um *thread* distribuído que inicia, em instantes distintos, duas cadeias de chamadas remotas, com eventos de tipo *server receive* e *server return* demarcados em ordem de ocorrência. A Figura 5-9 (b) mostra uma possível ordem de entrega para esses eventos, junto com a evolução do conhecimento do agente central com relação à seqüência de fotografias que compõem o *thread* distribuído. Note que a ausência de informações de ordenação faz com que o agente global confunda o evento 7 com o evento 3, levando a uma inconsistência: o evento 8 informa que um *thread* local que pertence ao nodo 4 deixa de fazer parte do *thread* distribuído. De acordo com o conhecimento do agente central, no entanto, não há nenhum *thread* local do nodo 4 ligado ao *thread* distribuído.

Esse tipo de inconsistência pode ser difícil de lidar e pode, em alguns casos, levar ao registro de transições inválidas. Imagine se, após a primeira chamada ao nodo 3 (evento 2), o *thread* distribuído chamasse, antes de retornar (evento 3) um outro nodo, um nodo 5. Suponha ainda que os eventos *server receive* e *server return* produzidos no nodo 5 fossem registrados pelo agente central após os eventos 7 e 8 da Figura 5-9 (b). Nesse caso, o agente central teria mostrado ao usuário, mesmo que por alguns instantes, uma seqüência de transições que nunca aconteceu (a seqüência $\{1.0, 2.0\}; \{1.0, 2.0, 3.0\}; \{1.0, 2.0\}$), violando o princípio da fidedignidade da Seção 3.1.

Alternativa 2: Relógios lógicos duplos e ordenação topológica *on-the-fly*

Capturar e exibir a execução de forma consistente seria muito mais simples caso fosse possível determinar se ainda existem eventos que não foram recebidos (antes do último evento observado), ou encontram-se trânsito. Isso porque, sob tais circunstâncias, Kimelman e Zernik [99] nos dão um algoritmo bastante simples, denominado *ordenação topológica on-the-fly*, para que possamos ordenar eventos. O algoritmo consiste em manter uma janela deslizante sobre as porções do grafo de eventos que ainda não estão "prontas" para exibição. Para entender como o algoritmo funciona, suponha que fosse possível determinar, no instante em que o evento 7 (da Figura 5-9) é recebido, que existem quatro eventos causalmente relacionados a 7 ainda não recebidos. Nesse caso, poderíamos simplesmente armazenar o evento 7 (e todos os eventos que chegarem após 7) até que o restante dos eventos anteriores a 7 fossem recebidos (ou até que pudéssemos decidir, com probabilidade razoável, que algum evento se perdeu). A idéia é ilustrada na Figura 5-10.

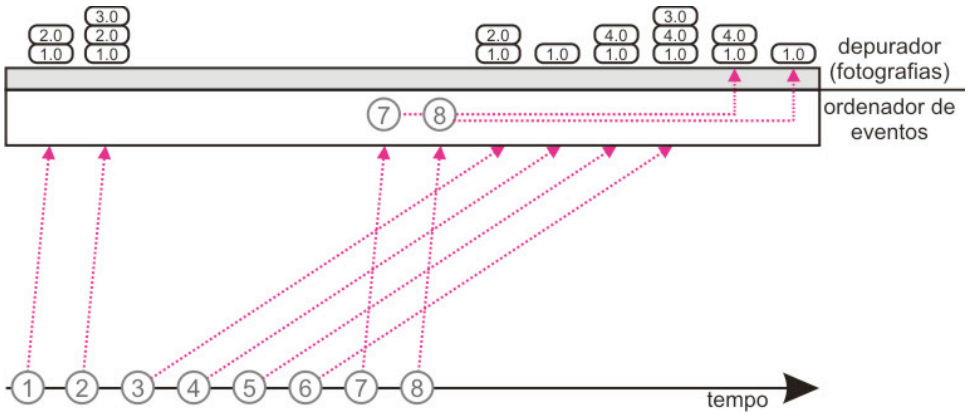

Figura 5-10. Eventos entregues fora de ordem sendo reorganizados pelo ordenador de eventos.

Em contrapartida, o processador de eventos (depurador na Figura 5-10) passa a operar sob a confortável hipótese de que eventos causalmente relacionados serão sempre recebidos em ordem, algo que simplifica consideravelmente a sua implementação.

Relógios vetoriais nos dão uma forma direta e segura para a implementação de um ordenador de eventos. Esses relógios são, no entanto, computacionalmente caros e não se adaptam bem a sistemas em que o número de processos não é conhecido a priori. Optamos, portanto, pelo uso de um sistema de relógios lógicos mais simples e de garantias mais fracas, de nossa autoria, que utiliza $n+m$ relógios escalares (no pior caso), onde *n* é o número de *threads* locais e *m* o número de processos, e transmite $O(1)$ *timestamps* escalares por mensagem. Relógios vetoriais, por sua vez, utilizam m^2 relógios escalares e transmitem $O(m)$ *timestamps* escalares por mensagem.

Para evitar excessos, vamos descrever a operação de nosso sistema de relógios de maneira um tanto quanto informal. Na descrição que segue, um *thread* local "envia uma mensagem" quando: **(1)** provoca um envio de mensagem ao iniciar uma chamada remota ou **(2)** termina de tratar uma requisição, desencadeando o envio de uma mensagem de resposta. Similarmente, um *thread* local "recebe uma mensagem" quando: **(1)** inicia o tratamento de uma requisição, em resposta a uma mensagem recebida pela rede ou **(2)** quando bloqueado em uma chamada remota, desbloqueia após receber uma resposta do servidor.

Seja *C* uma execução distribuída e sejam L_C, P_C e E_C os conjuntos de todos os *threads* locais, processos e eventos, respectivamente, que fazem parte *C*. Nosso sistema de relógios escalares atribui um contador r_l a cada *thread* local em L_C e um contador r_p a cada processo em P_C. O funcionamento do sistema de relógios prossegue da seguinte forma:

1. Inicialmente, $r_p = r_l = 0$ para todo $p \in P_C$, $l \in L_C$.
2. Quando um *thread* local $l \in L_C$ envia uma mensagem (de requisição ou de resposta), um *timestamp* $t_m = r_l$ é anexado a essa mensagem.
3. Quando um *thread* local $l \in L_C$ recebe uma mensagem:
    a. o valor de t_m é extraído da mensagem e $r_l := t_m + 1$,
    b. $r_p := r_p + 1$ (*p* é o processo pai de l').

O *timestamp* de um evento $e \in E_C$ produzido pelo protocolo de rastreio (*server receive* e *server return*) é dado pelo par de valores $(v_1^e, v_2^e) \in \mathbb{N}^2$, onde $v_1^e = r_l$ e $v_2^e = r_p$ no instante do registro de *e* no *thread* local $l \in L_C$ pertencente ao processo $p \in P_C$. Os eventos são sempre registrados após a atualização dos valores dos dois contadores. Para entender como esse sistema de relógios resolve o problema da Alternativa 1**,** considere o exemplo apresentado na Figura 5-11, em que dois *threads* distribuídos, T_1 e T_2, participam de uma cadeia de chamada envolvendo três nodos. Os valores de

v_1 são mostrados nos círculos de interior claro, enquanto os valores de v_2, nos círculos de interior escuro.

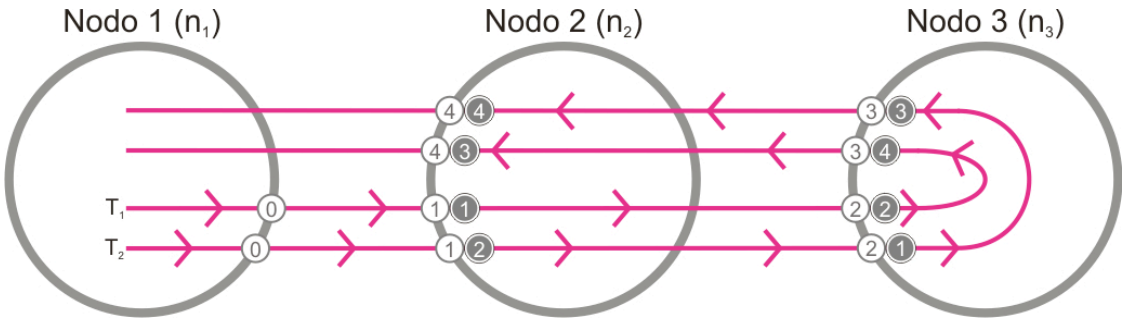

Figura 5-11. O sistema de duplos relógios lógicos.

Agora note que os valores de v_1 enumeram a seqüência eventos que ocorrem dentro de um mesmo *thread* distribuído, de tal forma que se recebemos dois eventos a e b ($a,b \in E_C$) tais que $v_1^a = 1$ e $v_1^b = 4$, então sabemos que existem eventos não recebidos $c,d \in E_C$ tais que $v_1^c = 2$ e $v_1^d = 3$. Em particular, dados dois eventos $a,b \in E_C$ em um mesmo *thread* distribuído, podemos afirmar com certeza que $v_1^a < v_1^b \Leftrightarrow a \rightarrow b$ (onde $\rightarrow$ é a relação de causalidade da Seção 2.1). Se a e b pertencem a diferentes *threads* distribuídos, no entanto, o valor de v_1 não nos diz nada.

Os valores de v_2 complementam as informações providas por v_1, revelando a ordem em que os eventos acontecem num dado nodo. Particularmente, se a e b são gerados dentro de um mesmo processo $p \in P_C$, então sabemos com certeza que $a \rightarrow b$. Ainda, de maneira semelhante ao que acontece com os valores de v_1, dados dois eventos $a,b \in E_C$, $v_2^a < v_2^b \Leftrightarrow a \rightarrow b$.

Suponhamos agora que um subconjunto dos eventos apresentados na Figura 5-11 fosse recebido na seguinte ordem: $(2,2,T_1,n_3);(1,2,T_2,n_2);(1,1,T_1,n_3);(2,1,T_2,n_3)$. Um ordenador de eventos baseado no algoritmo de ordenação topológica *on-the-fly* funcionaria da seguinte forma:

1. Ao receber o evento $e_1 = (2,2,T_1,n_3)$, o ordenador determina (examinando v_1) que existe um evento no *thread* distribuído T_1 que ainda não foi recebido. O evento é armazenado num *buffer*.
2. Ao receber o evento $e_2 = (1,2,T_2,n_2)$, o ordenador determina (examinando v_1) que o conjunto de eventos para o *thread* distribuído T_1 está completo, mas determina também (examinando v_2) que existe um evento que precede causalmente o evento e_2 na ordem local do nodo n_2 e que não foi ainda recebido. O evento é armazenado num *buffer*.

3. Ao receber o evento $e_3 = (1,1,T_1,n_3)$, o ordenador determina que o histórico causal de e_2 está completo. O ordenador insere então e_3 e e_2 na fila do processador de eventos e os remove do *buffer*.
4. Ao receber o evento $e_4 = (2,1,T_2,n_3)$, o ordenador determina que o histórico causal de e_1 está completo. O ordenador insere então e_4 e e_3 na fila do processador de eventos e os remove do *buffer*.

A descrição da operação do ordenador de eventos é ilustrada na Figura 5-12. O sistema de duplos relógios lógicos nos permite identificar se todos os eventos causalmente relacionados a um evento $e \in E_c$ recebido foram também recebidos. Há um problema ainda não resolvido, no entanto – como saber se existem, ainda em trânsito, eventos mais atuais do que o último evento já recebido. Esse problema não tem, infelizmente, uma solução satisfatória, comprometendo a qualidade da implementação da operação de suspensão em *threads* distribuídos com rastreio assíncrono.

O problema é que eventos em trânsito significam que a cabeça do *thread* distribuído pode ter mudado. Dado que a suspensão de um *thread* distribuído equivale à suspensão de sua cabeça, isso quer dizer que o melhor que podemos fazer é suspender a última cabeça conhecida. Se um evento de atualização da cabeça for recebido após consumada a operação de suspensão, o melhor que podemos fazer é "atualizar" a operação em vista do conhecimento recém-adquirido, tentando novamente suspender a última cabeça conhecida.

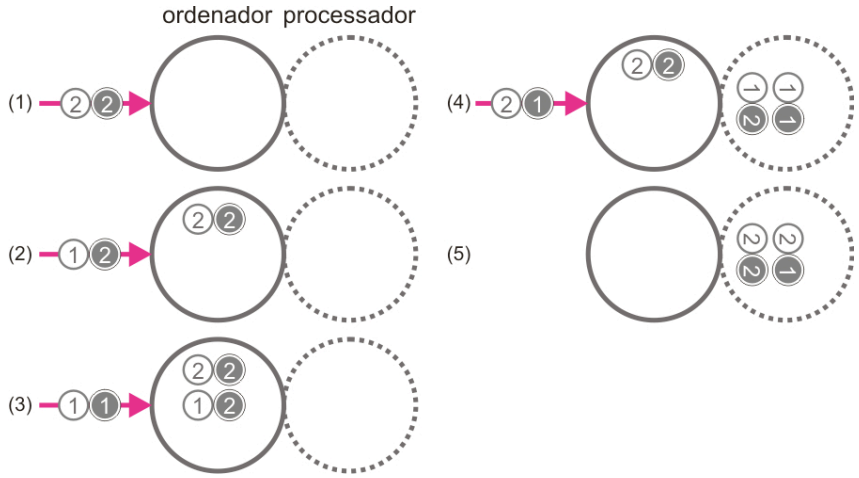

Figura 5-12. Ordenação topológica *on-the-fly* no sistema de duplos relógios lógicos.

Uma implementação da operação de suspensão construída dessa forma tem um certo número de inconvenientes:

1. **É ineficiente:** a suspensão de um *thread* distribuído pode exigir um grande número de troca de mensagens e um grande número de operações de suspensão de *threads* locais. No pior

caso, uma única operação de suspensão pode levar à suspensão de todos os *threads* locais no sistema distribuído.

2. **Suspensões espúrias *versus* não-garantias de término**: se a operação de suspensão não desbloquear *threads* locais erroneamente suspensos, é possível que o usuário veja uma série de suspensões espúrias acontecendo. Se por outro lado a operação de suspensão desbloquear os *threads* locais erroneamente suspensos então não há garantias de que a operação de suspensão do *thread* distribuído termine.

Alternativa 3: Rastreio síncrono

Seguindo nos moldes da análise de operações interativas que apresentamos na Seção 5.3.2:

1. É importante que a operação de suspensão em *threads* distribuídos seja implementada de forma confiável. A falha de uma operação de suspensão é considerada uma falha "grave".
2. Não pode ser implementada corretamente com rastreio assíncrono.

Em face dessas constatações, exploramos uma última alternativa – que emprega sincronizações – e que mostrou-se viável. No caso específico da operação de suspensão, a sincronização entre agentes local e global durante a mudança de cabeça do *thread* distribuído (eventos *server receive* e *server return*) é suficiente para resolver o problema. Essencialmente, o agente local suspende o *thread* local envolvido na notificação ao agente global até que uma mensagem de resposta seja recebida. Dessa forma, podemos garantir que o conhecimento do agente global sempre corresponde – na ausência de falhas – ao estado "real" do sistema distribuído e que, portanto, as operações de suspensão sempre serão direcionadas ao *thread* local correto.

5.3.4 Dividindo *threads* distribuídos na presença de falhas

Até agora, todas as nossas elaborações tomaram como hipótese que nodos e *links* não falham. Em aplicações reais, entretanto, é possível que esses elementos falhem, resultando no desbloqueio anormal de *threads* locais e na conseqüente produção de eventos inesperados, violando a estrutura do *thread* distribuído estabelecida pela Definição 4-5. Nesta seção vamos introduzir uma extensão à Definição 4-5 que nos permite acomodar tais situações. Antes que apresentemos a nossa extensão, convém definir o que consideramos ser um desbloqueio "normal" para um *thread* distribuído.

Definição 5-1: Um *thread* local *l* bloqueado numa chamada remota desbloqueia normalmente quando seu desbloqueio resulta do processamento de uma mensagem, enviada pelo servidor que contém o *thread* local que tratou a requisição iniciada por *l*, que informa que essa requisição completou (com ou sem sucesso).

Portanto, um *thread* local bloqueado em uma requisição remota desbloqueia de forma *anormal* sempre que seu desbloqueio não estiver causalmente relacionado a uma mensagem enviada pelo servidor. Seja $T = \{s_{t_1},...,s_{t_m}\} \in D_C$ um *thread* distribuído e seja $s_{t_i} = \{l_1,...,l_n\}$, $1 \le i < m$, uma fotografia de *T*. Seja ainda T' o *thread* distribuído cuja base é l_{j+1}, onde $1 \le j < n$. Além daquilo que foi apresentado na Definição 4-5, itens *a* e *b*, vamos estabelecer que as seguintes alternativas são válidas:

2c. $s_{t_{i+1}} = \{l_1,...,l_j\}$ e $f(T',t_{i+1}) = \{l_{j+1},...,l_n\}$. Neste caso, o *thread* l_j desbloqueou anormalmente no instante t_{i+1}.

2d. $s_{t_{i+1}} = \{l_1,...,l_{i-1}\}$ e $f(T',t+\xi) = \{l_{i+1},...,l_n\}$. Neste caso, o *thread* l_j terminou no instante t_{i+1}.

Essas duas cláusulas definem em termos formais a organização do *thread* distribuído na presença de falhas – sempre que um *thread* local que participa em uma fotografia não-trivial morre, ou desbloqueia anormalmente, nós "dividimos" o *thread* distribuído em peças menores e independentes. Note que a morte de um *thread* local em tais condições pode provocar um desbloqueio anormal, mas nem todo desbloqueio anormal resulta da morte de um *thread* local. O principal aspecto capturado pelo processo de divisão diz respeito à estrutura de captura causal do *thread* distribuído. Isso porque o desbloqueio anormal de um *thread* local, ou morte de um *thread* local que participa em uma fotografia não-trivial, implicam que as duas seções nas quais o *thread* distribuído é "dividido" tornam-se independentes; isto é, não afetam mais, causalmente, umas as outras (ao menos não de imediato).

Um exemplo de um *thread* distribuído que sofre duas divisões simultâneas é mostrado na Figura 5-13. Divisões simultâneas podem ocorrer quando um nodo inteiro falha, resultando no término de todos os seus *threads* locais. A fotografia maximal mostrada na Figura 5-13 (a), composta por nove *threads* locais, é transformada em um conjunto de três fotografias maximais (Figura 5-13 (c)) após as divisões causadas pelo término de l_3 e l_7. Note que embora a nossa extensão da Definição 4-5 não preveja divisões simultâneas, isso não é um problema, já que a aplicação de divisões sucessivas é comutativa; isto é, se o *thread* distribuído se dividisse em l_3 e logo depois em l_7, os resultados seriam equivalentes a se ele se dividisse em l_7 e logo depois em l_3.

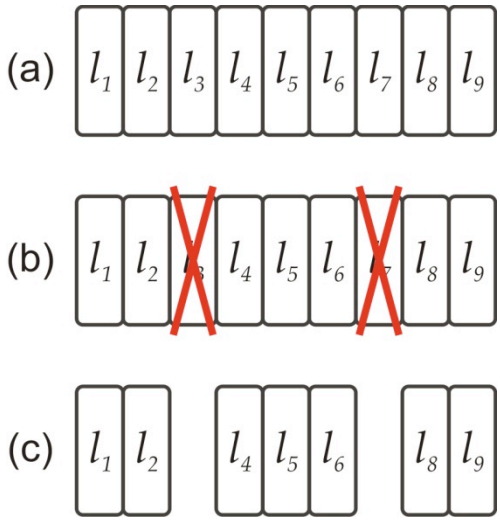

Figura 5-13. (a) Fotografia de um *thread* distribuído. (b) Morte simultânea de dois *threads* locais. (c) Fotografias resultantes após divisão.

5.4 Os agentes locais

Até agora, apresentamos aspectos e considerações ainda um tanto quanto conceituais a respeito de nosso depurador. Esses aspectos e considerações são importantes na medida em que nos serviram como base para justificar decisões arquiteturais, de projeto (e.g. como representar o *thread distribuído*) e de protocolo (e.g. como rastrear os *threads* distribuídos). Nesta seção vamos descrever o primeiro elemento arquitetural – os agentes locais – em mais detalhe, explicando como ele aplica os conceitos discutidos.

Conforme mencionamos na Seção 5.1, os agentes locais são responsáveis pela coleta de dados e pela interação com os processos que compõem o sistema distribuído. Agentes locais são compostos pela junção de depuradores simbólicos, código de instrumentação adicional e uma biblioteca de depuração (provida por nós). Os serviços exigidos dos depuradores simbólicos serão discutidos mas adiante nesse capítulo. O papel do código de instrumentação e da biblioteca de depuração será discutido em detalhe nas Seções 5.4.1 e 5.4.3.

Conforme mostra a Figura 5-14, o agente local utiliza, na verdade, dois protocolos de comunicação. Um deles é específico do depurador simbólico em uso no agente local (exemplos incluem o *Java Debug Wire Protocol [202]*, usado pela JPDA [201], o protocolo de depuração do GDB e o DBGp [24], usado pelo XDebug [170]). O outro protocolo é independente da linguagem na qual é escrita o processo da aplicação, bem como do depurador simbólico utilizado. Esse protocolo, de nossa concepção, é chamado *Distributed Debugging Wire Protocol* (DDWP). O

DDWP é utilizado para a transmissão dos eventos *server receive*, *server return* e eventos de divisão discutidos na Seção 5.3, além de informações de configuração.

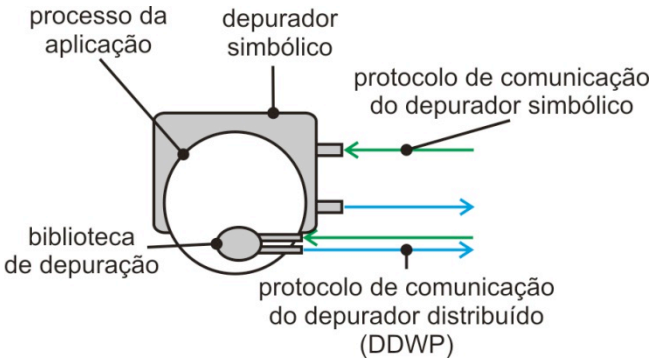

Figura 5-14. Visão mais detalhada do agente local.

5.4.1 Implementação Java/CORBA

O agente local Java/CORBA é bastante semelhante ao agente local mostrado na Figura 5-14, exceto pelo fato que o depurador simbólico empregado é o da JPDA, e o protocolo de comunicação do depurador simbólico, o JDWP. O foco desta seção será no código de instrumentação, na implementação do mecanismo de rastreio e no DDWP. A integração com os depuradores simbólicos (também parte dos agentes locais) será tratada em mais detalhes nas Seções 5.5.3 e 5.5.4.

Os principais elementos do mecanismo de rastreio dos agentes locais Java/CORBA são mostrados na Figura 5-15. Para facilitar a discussão desses elementos, vamos dividir chamadas remotas em dois fluxos:

1. **Fluxo de requisição:** tem início no instante em que a chamada remota se inicia (empilhamento[8] do registro de chamada remota), terminando quando o *thread* de lado do servidor termina de tratar a requisição (desempilhamento do registro de tratamento; registros de tratamento e de chamada remota foram discutidos na Seção 4.3.2).
2. **Fluxo de resposta:** tem início ao término do fluxo de requisição e dura até que o registro de chamada remota (do lado do cliente) seja desempilhado.

[8] Na pilha de chamadas do *thread* local que inicia a requisição.

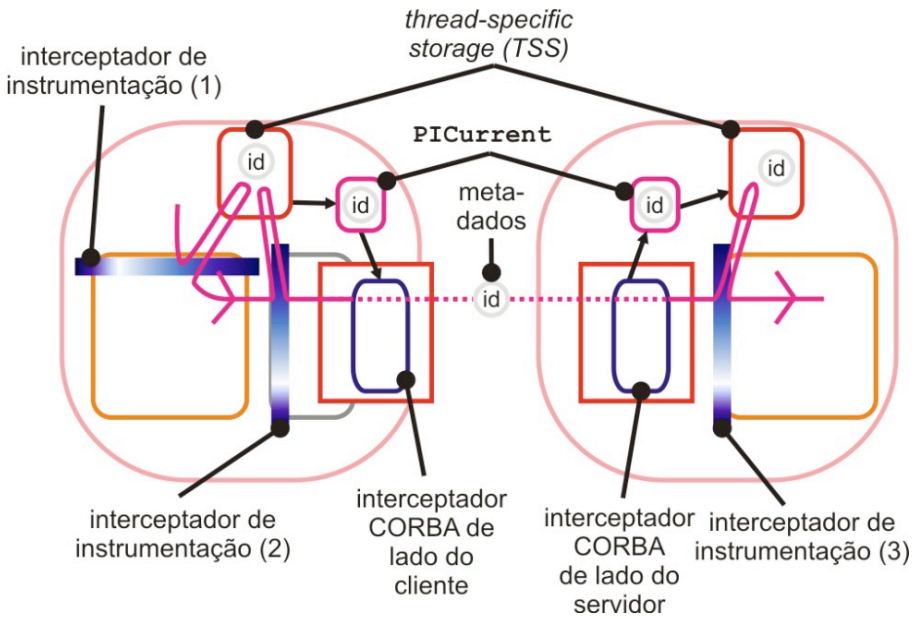

Figura 5-15. Mecanismo de rastreio no agente local Java/CORBA (fluxo de requisição).

1) **Interceptador de instrumentação (1) (**Figura **5-16 (a)):** inserido no início do método **`run()`** de cada classe que declara implementar a interface **`java.lang.Runnable`**. Ativado sempre que um novo *thread* local começa sua a execução. Este interceptador:
   a) Atribui o identificador de duas partes, descrito na Seção 5.2, a cada *thread* local criado. Para tanto, o interceptador invoca o método **`Tagger.getInstance().tagCurrent()`**, parte da biblioteca de depuração.
   b) Dá início a um protocolo de registro de *threads* locais que existe como parte de um pequeno mecanismo de reflexão distribuída que precisamos implementar. Esse mecanismo será discutido na Seção 5.5.4.
2) **Interceptador de instrumentação (2):** inserido em todos os métodos de *proxies* gerados pelo middleware que correspondam a métodos exportados por objetos remotos. Estes interceptadores são ativados sempre que uma chamada remota é realizada, sendo responsáveis por:
   a) Fluxo de requisição (Figura **5-16 (b), região (1)):** transferir o identificador do *thread* local *l* envolvido na chamada remota (o id_1 de *l*, seguindo a nomenclatura da Seção 5.2) e a posição do registro de chamada remota (seguindo a nomenclatura da Seção 4.3.2) para o objeto *PICurrent*, para que esses dados possam ser acessados pelo interceptador CORBA do lado do cliente (Figura 5-15).
   b) Fluxo de resposta (Figura **5-16 (b), região (2)):**
      i) Verificar se o *token* de resposta está presente (*tokens* de resposta serão discutidos na Seção 5.4.1.1), informando o agente central do desbloqueio anormal caso contrário.

ii) Solicitar a suspensão do *thread* local que retorna de uma chamada remota, como parte do protocolo de execução passo-a-passo "virtual" (discutido mais adiante nesta seção).

3) **Interceptador CORBA do lado do cliente:** inserido por configuração no ORB CORBA e ativado sempre que há uma chamada remota, ou mediada pelo middleware. Responsável por transferir, durante a o fluxo de requisição, as informações depositadas pelo interceptador de instrumentação (1) no objeto *PICurrent* para o contexto da requisição (*service context* [151]) e por fazer a tarefa inversa durante o fluxo de resposta.

```
public void run(){
    Tagger tagger = Tagger.getInstance();
    tagger.tagCurrent();

    // Restante do código ...
  }
}
```

(a)

```
public int operacaoRemota(){
  ORBHolder comm = ORBHolder.getInstance();
  try{
  ① comm.setStamp();
    // Código do stub...
  }finally{
     if(!comm.hasReplyToken())
  ②     comm.reportSplit();
     if(comm.isCurrentThreadStepping())
        comm.stepMeOut();
  }
}
```

(b)

Figura 5-16. Estrutura dos interceptadores de instrumentação 1 (a) e 2 (b).

4) **Interceptador CORBA do lado do servidor:** desempenha papel semelhante ao do interceptador CORBA de lado do cliente, transferindo os dados que chegam como informações de contexto junto com a requisição para o objeto *PICurrent* do lado do servidor. Além disso, o interceptador CORBA do lado do servidor cumpre duas funções adicionais:
   a) Durante o fluxo de requisição, insere o *token* de chamada remota no objeto *PICurrent*.
   b) Durante o fluxo de resposta, insere o *token* de resposta no contexto da requisição.

5) **Interceptador de instrumentação (3) (**Figura **5-17):** inserido nos métodos do objeto remoto exportados para acesso mediado pelo middleware (i.e., nos métodos do objeto que podem ser chamados remotamente). As funções deste interceptador podem ser resumidas a:
   a) **Antes da execução do método remoto:**
      i) Atribuir o identificador do *thread* distribuído, inserido no objeto *PICurrent* pelo interceptador CORBA do lado do servidor, ao id_2 *thread* local atual.
      ii) Capturar o tamanho da pilha no instante de ativação da operação remota (posição do quadro de tratamento remoto na pilha de chamadas do *thread* local).
      iii) Capturar informações a respeito da operação ativada (assinatura do método).
      iv) Notificar o agente central da mudança de fotografia (gerando um evento do tipo *server receive*).

```
public int operacaoRemota(){
  ORBHolder comm = ORBHolder.getInstance();
  try{
    comm.retrieveStamp();
    // Código do objeto remoto...
  }finally{
     comm.checkOut();
  }
}
```

Trecho 1

```
00 se existe token de chamada remota então
01   DT_id := id do thread distribuído, extraído do objeto PICurrent.
02   atribua DT_id ao id_2 do thread local atual.
03   LT_id := id_1 do thread local atual.
04   STK_1 := tamanho da pilha de chamada do lado do cliente, extraído do
             objeto PICurrent.
05   STK_2 := (tamanho da pilha de chamadas do thread local atual) - 1.
06   OP_id := identificação da operação atual.
07   Remova o token de chamada remota.
08   Atribua um contador c, inicializado com 0, ao thread local atual.
09   Envie ao agente global um evento server receive contendo (DT_id, LT_id,
     OP_id, STK_1, STK_2).
10 senão então
11   se há um contador c vinculado ao thread local atual então
12     incremente o contador c.
```

Trecho 2

```
00 se há um contador c vinculado ao thread local atual então
01  decremente c.
02  se c = 0 então
03    DT_id := id_2 do thread local atual.
04    LT_id := id_1 do thread local atual.
05    Envie ao agente global um evento server return contendo (DT_id, LT_id)
      e armazene o código de resposta na variável isStepping.
06    Insira o valor de isStepping no objeto PICurrent.
```

Figura 5-17. Interceptador de instrumentação inserido nos objetos remotos.

b) **Após a execução do método remoto:**

i) Notificar o agente central da mudança de fotografia (gerando um evento do tipo *server return*).

ii) Inserir o estado do *thread* distribuído adquirido do meta-nível no objeto *PICurrent* (para que ele possa ser propagado para o cliente na resposta).

5.4.1.1 *Token* de resposta

Nosso depurador distribuído foi concebido com portabilidade em mente. O agente local Java/CORBA não é exceção a essa regra; tanto o código de instrumentação quanto as porções da biblioteca de depuração específicas para CORBA interagem com o middleware exclusivamente por meio de sua API pública, tomando como hipótese apenas que a implementação do ORB adere à especificação [151]. Na medida do possível, no entanto, nós gostaríamos de não depender de

mecanismos atrelados a plataformas de middleware específicas (incluindo aqui todos os ORBs CORBA na categoria "específicas"), o que nos leva ao primeiro problema: a detecção de mensagens de resposta perdidas.

A especificação CORBA define uma exceção que cumpre justamente ao propósito de detectar mensagens de resposta perdidas. Segundo a especificação da exceção `CORBA::COMM_FAILURE`: "*This exception is raised if communication is lost while an operation is in progress, after the request was sent by the client, but before the reply from the server has been returned to the client.*" Nós não gostaríamos, no entanto, de depender de uma estrutura de informação de erros baseada em exceções, por dois motivos:

1. Ao fazê-lo, o código de instrumentação passa a pressupor implicitamente que todas as plataformas de middleware comunicam a perda de mensagens de resposta usando exceções.
2. Pode ser bastante difícil determinar se uma exceção indica que uma mensagem de resposta se perdeu. Em alguns casos, pode ser necessário examinar mensagens de erro formatadas como *Strings* e, mesmo assim, termos conviver com algum nível de ambigüidade.

É possível, entretanto, argumentar que essas razões não são fortes o suficiente para descartarmos as exceções no caso de CORBA, pois:

1. A implementação do agente local já é acoplada a interfaces da especificação CORBA, dado que os métodos chamados pelo código de instrumentação acessam a interface `CORBA::ORB` e `PortableInterceptor::Current`.
2. A maior parte das plataformas de middleware indicam erros (como a perda de mensagens de resposta) de forma clara e não-ambígua, restringindo a validade do argumento (2) anterior.

Seja como for, nós optamos por desenvolver um mecanismo alternativo, condizente com a Definição 5-1, capaz de identificar mensagens perdidas sem o uso de exceções e apoiando-se apenas em funcionalidades já utilizadas por outros mecanismos do depurador – essencialmente, a nossa alternativa para a detecção de mensagens perdidas depende apenas do mecanismo de passagem de informações de contexto já utilizado pelo mecanismo de rastreio.

O mecanismo é bastante simples e já foi descrito, implicitamente, quando descrevemos o papel de cada interceptador: nós utilizamos um *token* de um único *bit*, que é inserido pelo interceptador CORBA do lado do servidor no contexto da requisição no momento de envio da resposta. Quando há uma mensagem de resposta, esse *token* de um *bit* será então encontrado pelo interceptador CORBA do lado do cliente e inserido no objeto *PICurrent*, de onde se torna acessível ao interceptador de instrumentação (1). Conforme mostra a Figura 5-16 (a), o interceptador de instrumentação (1) testa, ao término da chamada no proxy, se o *token* de resposta está presente ou

não. Em caso negativo, o interceptador chama o método `reportSplit()`, que notifica o agente central a respeito do desbloqueio anormal.

Um problema com esse mecanismo (e com a Definição 5-1, quando combinada com a Definição 4-17) é que o resultado é excessivamente conservador. Para entender o porquê, considere o seguinte cenário hipotético:

1. Um *thread* local *l*, executando código da aplicação, chama um método em um *proxy* remoto. De acordo com a Definição 4-17, o simples empilhar de um quadro de chamada remota (um quadro de ativação cuja localização é um *proxy* remoto) em *l* já implica que *l* está bloqueado em uma chamada remota.
2. O interceptador de instrumentação (1) é ativado.
3. O middleware determina, antes sequer de contatar algum servidor, que a chamada não pode ser completada, lançando uma exceção.
4. O interceptador de instrumentação (2) é ativado e determina que o *token* de resposta não está presente.
5. O interceptador de instrumentação (2) notifica o agente central a respeito do desbloqueio anormal em *l.* Isso é compatível com a Definição 5-1, já que *l* "desbloqueou" sem que a causa desse desbloqueio tenha sido uma mensagem de resposta do servidor.
6. O usuário é notificado a respeito de um erro que não é, necessariamente, um erro.

A implicação imediata é que o mecanismo pode levar, sob certas circunstâncias, à produção de notificações de erro espúrias. Mesmo apesar dessa limitação, no entanto, nós optamos por utilizar o *token* de resposta ao invés do tratamento de exceções específicas. As modificações que se fariam necessárias no agente local caso optássemos por alterar o mecanismo não seriam, no entanto, extensas. A Figura 5-18 mostra uma possível implementação para o interceptador de instrumentação (1).

O método `meansLostReply(Throwable)` encapsula a lógica que testa o significado da exceção. Note que o código mostrado não seria válido na linguagem Java, já que um `Throwable` pode ser uma exceção checada, o que implica que a linha sublinhada (`throw t;`) seria incompatível com a assinatura do método `operaçãoRemota`. Felizmente, nossos interceptadores de instrumentação não são construídos pelo processo tradicional de compilação de código-fonte – nós utilizamos uma biblioteca de manipulação de *bytecodes* (a *Apache Byte Code Engineering Library* [8]) para injetar, em tempo de execução, os interceptadores diretamente entre os *bytecodes* da aplicação. A regra das exceções checadas é verificada apenas em tempo de compilação [74]; portanto, podemos construir um interceptador com a semântica apresentada na Figura 5-18 sem maiores problemas.

```
public int operacaoRemota(){
  ORBHolder comm = ORBHolder.getInstance();
  try{
    comm.setStamp();
    // Código do stub...
  }catch(Throwable t){
    if(comm.meansLostReply(t))
      comm.reportSplit();
    throw t;
  }finally{
    if(comm.isCurrentThreadStepping())
      comm.stepMeOut();
  }
}
```

Figura 5-18. Reconhecimento de mensagens de resposta perdidas baseado no tratamento de exceções do middleware.

Dito isso, a manipulação de *bytecodes* é, em Java, conveniente pelas seguintes razões:

1. Não requer o código-fonte da aplicação.
2. Pode ser feita em tempo de execução, dispensando fases extras de pré-processamento e a gestão desajeitada de múltiplas versões da aplicação.
3. Nos dá maior flexibilidade na implementação dos interceptadores.

A manipulação de *bytecodes* é prática comum em Java, suplantando algumas das limitações nos mecanismos de reflexão providos pela linguagem e pelo ambiente de execução.

5.4.1.2 *Token* de chamadas remotas

Conforme descrevemos anteriormente, o interceptador de instrumentação (3) é inserido em todos os métodos expostos pelos objetos remotos. A principal função desse interceptador é a coleta de certos dados (mostrados na Figura 5-17) de execução no início do tratamento de cada chamada remota, bem como a produção dos eventos *server receive* e *server return* (Seção 5.2). O problema é que esses interceptadores são incorporados ao código dos métodos e serão, portanto, ativados sempre que houver uma chamada, remota ou não. É preciso, portanto, que sejamos capazes de distinguir entre chamadas remotas e locais, ou nossos interceptadores irão produzir eventos de forma incorreta. O mesmo problema se apresenta quando o Trecho 2 (Figura 5-17) do interceptador de instrumentação (3) é levado em consideração: nem sempre o término da execução de um método ativado em um objeto remoto deve corresponder à produção de um evento *server return*. É justamente esse problema que o *token* de chamada remota resolve.

O *token* de chamada remota é um *flag* booleano inserido pelo interceptador CORBA do lado do servidor no objeto *PICurrent*. O interceptador CORBA do lado do servidor só é ativado, no entanto, durante chamadas mediadas pelo middleware. Agora note que o Trecho 1 do interceptador de instrumentação (3), ao ser ativado, testa pela presença do *token* no objeto *PICurrent* e, caso

presente, o remove. Isso significa que o *token* só estará presente na primeira ativação do interceptador de instrumentação (3) após o recebimento de uma requisição remota. Em outras palavras, podemos determinar qual a primeira ativação de um método remoto e, por conseqüência, qual a posição do quadro de tratamento na pilha de chamadas do *thread* local do lado do servidor.

O uso de um contador nos permite manter controle de ativações subseqüentes de métodos instrumentados. Essencialmente, conforme mostra a Figura 5-17, o interceptador de instrumentação (3) atribui, logo após a remoção do *token* de chamada remota, um contador com o valor zero ao *thread* local atual. Ativações subseqüentes do Trecho 1 do interceptador pelo mesmo *thread* local – já sem o *token* de chamada remota – resultam em incrementos ao contador, de tal forma que um valor n no contador implica que existem $n+1$ ativações a métodos instrumentados na pilha de chamadas do *thread* local. Em algum momento, cada uma dessas chamadas ativas deve retornar e, quando isso acontecer, o Trecho 2 do interceptador de instrumentação (3) será ativado. O Trecho 2 do interceptador compara o valor do contador atribuído ao *thread* local atual (se presente) com zero. Se o valor for igual a zero, isso significa que um quadro de tratamento está prestes a ser desempilhado e que, portanto, um evento *server return* deve ser gerado. Se o valor for maior do que zero, por outro lado, isso significa que ainda existem chamadas de métodos instrumentados ativas antes da primeira chamada, então o interceptador simplesmente decrementa o contador.

5.4.2 Interceptadores, eventos e o modo de execução passo-a-passo

Até agora, o foco de nossa discussão foi na representação e no rastreio de *threads* distribuídos. Nós ainda não explicamos, no entanto, como as operações do modo de execução passo-a-passo são implementadas. De fato, esta seção não vai prover uma explicação completa, já que o modo de execução passo-a-passo depende muito do agente central (que será discutido na Seção 5.5). Vamos apenas dar uma visão geral do mecanismo, com enfoque na participação dos agentes locais.

O mecanismo de execução passo-a-passo implementado por nosso depurador é semelhante ao mecanismo presente em depuradores tradicionais. Existem três modos de navegação básicos:

1. *Step into*: executa o programa até a próxima linha de código, seguindo a semântica de execução da linguagem subjacente.
2. *Step over:* executa até a próxima linha de código, mas apenas no método atual (pula a execução de linhas contidas em chamadas de métodos entre a linha atual e a próxima do quadro de ativação atual).
3. *Step return:* executa até a primeira linha de código após o desempilhamento do quadro de ativação atual (executa até que o método ativo retorne). Um *step into* ou *over* que ultrapassa os limites do método ativo equivale a um *step return*.

A granulosidade padrão do modo de execução passo-a-passo é a linha de código, podendo variar de acordo com as capacidades do depurador simbólico. Conforme discutimos anteriormente, é necessária alguma intervenção para que possamos criar a ilusão de coesão necessária para que um *step into* em um *proxy* local para um objeto em outro espaço de endereçamento tenha efeito semelhante a um *step into* num objeto local no mesmo espaço de endereçamento. A estrutura dessas intervenções é mostrada na Figura 5-19: quando um cliente executa um *step into* em um *proxy* para um objeto remoto, um comando *resume* é enviado ao *thread* local, permitindo que a requisição prossiga desimpedida até a porta de entrada para o objeto remoto – o Trecho 1 do interceptador de instrumentação (3).

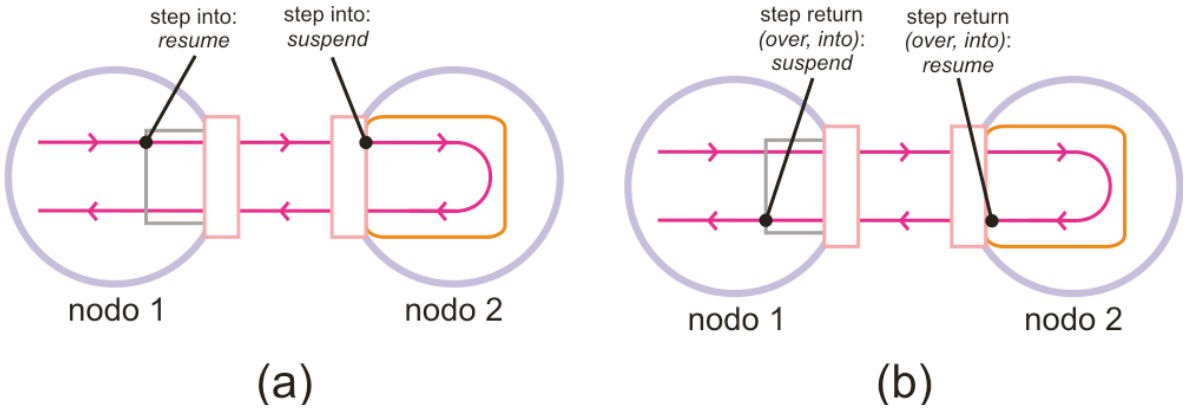

Figura 5-19. Pontos de intervenção no fluxo dos *threads* locais durante (a) um *step into* em um *proxy* (lado do cliente) e (b) um *step return* do lado do servidor.

Recorde do início da Seção 5.4 que o evento *server receive* enviado pelo interceptador ao agente central contém um elemento que denominamos OP_{id} (Figura 5-17). OP_{id} deve conter informações suficientes (normalmente a assinatura da operação remota chamada e a classe do objeto remoto) para que o agente central possa inserir um *breakpoint* capaz de suspender o *thread* local que trata a requisição no ponto correto – i.e., na primeira linha de código da implementação do método remoto. No agente local Java, visando facilitar a implementação, OP_{id} contém:

- o nome completo da classe (*fully qualified name* [74]) do objeto remoto,
- a posição da primeira linha de código do método remoto no código-fonte usado para gerar a versão em execução da classe.

Essas informações podem ser extraídas em tempo de execução pelo agente local, simplificando o código de injeção de *bytecodes*. A alternativa a extrair essas informações em tempo de execução é extraí-las durante a instrumentação e inseri-las em constantes em cada interceptador.

Ao receber um evento *server receive*, portanto, o agente central utiliza o OP_{id} para inserir um *breakpoint* no lugar apropriado antes de responder ao agente local (caso a requisição tenha sido

marcada como "em modo de execução passo-a-passo"). Enquanto não houver resposta do agente central, a requisição fica bloqueada (lembre-se que as notificações são síncronas), então há tempo de sobra para que o agente central posicione o *breakpoint*. Após receber a resposta do agente central, o agente local desbloqueia o *thread* local que trata a requisição. Esse *thread* local então, em algum momento, atinge o *breakpoint* posicionado anteriormente (*step into: suspend* na Figura 5-19). O agente central devolve então o controle ao usuário, que tem a ilusão de ter efetuado um *step into* em um objeto remoto. Essas interações são ilustradas na Figura 5-20.

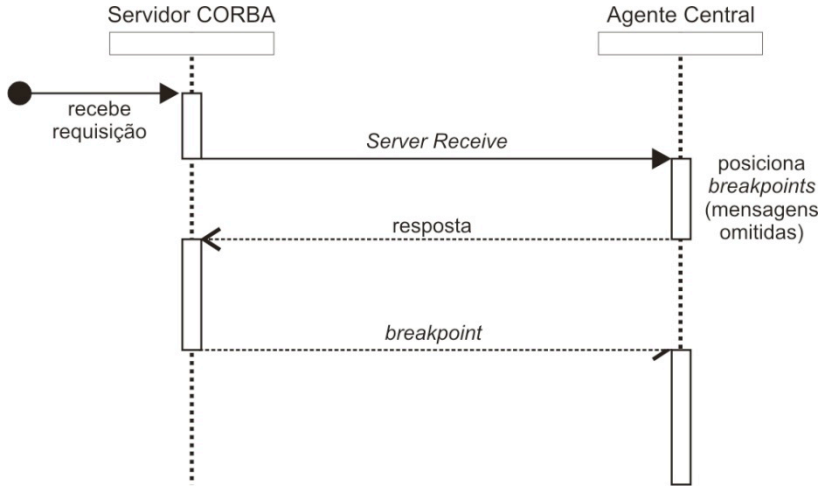

Figura 5-20. Dinâmica de interações entre requisição, código de instrumentação e agente central durante um *step into*. As mensagens enviadas pelo agente central para o posicionamento dos *breakpoints* foram omitidas.

A dinâmica de interações durante um *step return* (ou *step into/over* que ultrapassa os limites do método) é ligeiramente mais complexa. Inicialmente, conforme mostra o diagrama de seqüência da Figura 5-21, o usuário solicita o avanço da execução no modo passo-a-passo. Nossa hipótese é que esse avanço faz com que o método retorne, portanto o Trecho 2 do interceptador de instrumentação (3) será ativado antes da requisição de *stepping* ser completada, gerando um evento *server return*. O agente central, percebendo que o evento *server return* ocorre num *thread* com uma requisição de *stepping* pendente, dá início ao protocolo de *step return* virtual: o *thread* local recebe um comando *resume* (e a requisição pendente de *stepping* é cancelada) de tal forma que, quando a resposta do evento *server return* é enviada, a requisição prossegue sem interrupções (i.e., não pára na próxima linha de código).

Junto com a resposta do evento *server return*, enviada para o agente local, segue um código de *status*, que indica que o *thread* distribuído está em modo de execução passo-a-passo (i.e., o agente central compartilha seu conhecimento de meta-nível com o nível base). Conforme mencionamos no início da Seção 5.4.1, esse *status* adquirido do meta-nível é propagado de volta para o cliente pelos

interceptadores CORBA e de instrumentação. Quando o *thread* do lado do cliente desbloqueia, o Trecho 2 do interceptador de instrumentação (2) testa pela presença do código de status e, caso presente, desvia o *thread* local para uma chamada ao método `stepMeOut()`, conforme mostra a Figura 5-16. A implementação do método `stepMeOut()` é vazia, mas contém um *breakpoint*, posicionado pelo agente central durante a inicialização do processo. Isso significa que o *thread* local será suspenso, pelo *breakpoint*, produzindo a ilusão que o *step return* no objeto remoto continua do lado do cliente.

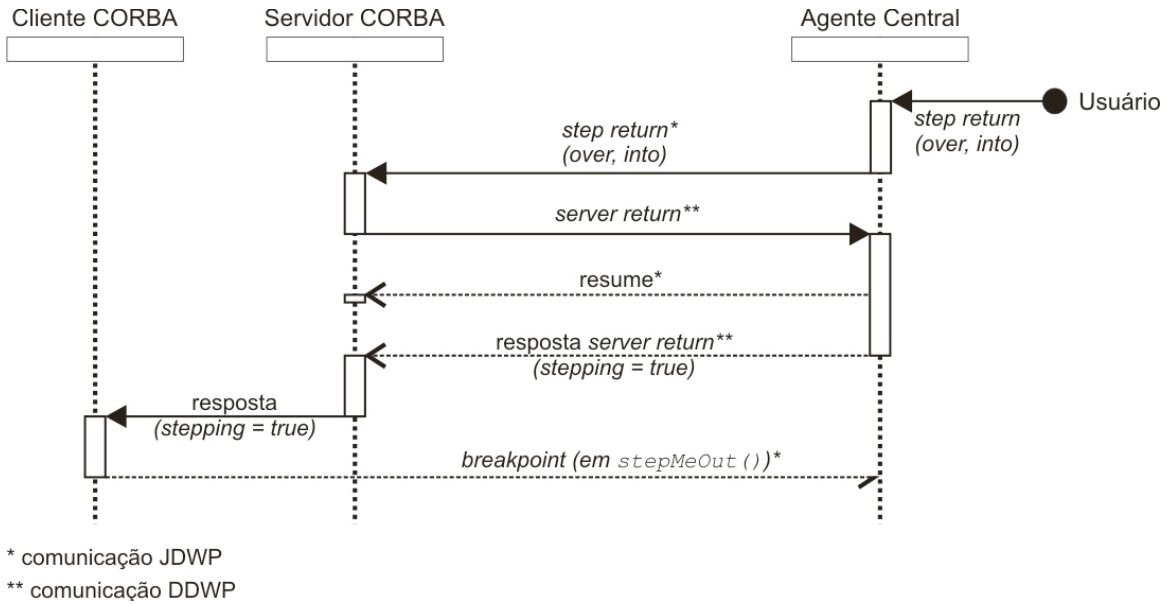

Figura 5-21. Dinâmica de interações entre requisição, cliente CORBA, servidor CORBA e agente central durante um *step return* (ou *step over/into* que ultrapassa os limites do método).

Uma pergunta que poderia surgir neste contexto é porque o mecanismo de suspensão do *thread* local do lado do cliente é tão mais complexo – não poderíamos, de forma semelhante ao que fazemos do lado do servidor, simplesmente posicionar um *breakpoint* no ponto de reentrada, ou *follow set* [167], do *thread* local no cliente? A questão é que determinar o *follow set* do *thread* local no servidor é algo bastante direto – após ativar o Trecho 1 do interceptador de instrumentação (3), o *thread* local sempre executa a primeira linha do método remoto. Determinar o *follow set* do *thread* local durante a reentrada no cliente, no entanto, é uma tarefa mais complexa, essencialmente porque exceções lançadas de dentro do *stub* podem levar o *thread* a localizações difíceis de determinar de forma simples.

5.4.3 Viabilidade em outras linguagens e sistemas de middleware

Uma das questões centrais em nosso trabalho diz respeito a se o maquinário utilizado nos agentes locais Java/CORBA pode de fato ser adaptado, ou reproduzido, e com qual dificuldade, a

outras linguagens, ambientes de execução e plataformas de middleware. Vamos argumentar nesta seção que os requisitos impostos pelo mecanismo de rastreio e notificação sobre middleware e ambiente de execução são relativamente comuns, e que a adaptação da técnica a outros ambientes deve ser bastante direta, embora não trivial.

Vamos começar com uma discussão dos requisitos impostos sobre o middleware e sobre a estrutura da aplicação distribuída:

1. A abstração de *threads* distribuídos é válida apenas para aplicações baseadas em objetos distribuídos. Clientes têm que acessar objetos remotos via *proxies* e objetos remotos devem corresponder a objetos reais.

Esse requisito é uma reafirmação de que este trabalho é voltado especificamente para sistemas de objetos distribuídos. O uso de objetos distribuídos em apenas uma das pontas, onde o cliente acessa o servidor por meio de *proxies*, mas o servidor utiliza alguma outra coisa, simulando a interação com um objeto remoto apenas pelo protocolo, não é aceitável. De maneira semelhante, clientes que não usam *proxies*, interagindo com servidores diretamente por meio do protocolo do middleware (e.g., *IIOP* [151]), não se encaixam no requisito.

2. A plataforma de middleware deve disponibilizar um mecanismo para passagem de informações de contexto (meta-dados) com cada requisição.

Isso é necessário para que o mecanismo de rastreio possa ser implementado. Não se trata de um requisito obrigatório, no entanto. Em seu trabalho com CORBA, Li [116] modifica o gerador de *stubs* do middleware para que um parâmetro extra – usado para passagem de meta-dados – seja incluído. Zhu [234] propõe uma alternativa semelhante para Java/RMI, que parte da instrumentação do serviço de nomes, *proxies* e objetos remotos. A solução de Zhu é mais elegante por não exigir modificações às interfaces e por ser generalizável até mesmo a outras plataformas de middleware (para Java). Uma terceira proposta, ainda para Java/RMI, é apresentada no trabalho de Tilevich e Smaragdakis [220], onde os autores também propõem a alteração de *stubs* e objetos remotos para a inclusão de um parâmetro extra. A proposta de Tilevich e Smaragdakis não requer alterações no serviço de nomes, mas depende da manipulação de *bytecodes* e é específica para Java/RMI. Seja como for, a esmagadora maioria das plataformas de middleware para objetos distribuídos atualmente em uso disponibiliza mecanismos para passagem de informações de contexto, ou pode ser adaptada por meio de técnicas bem documentadas.

3. O mecanismo de comunicação predominante deve ser a chamada síncrona e bloqueante.

O terceiro requisito é fundamental para que a abstração do *thread* distribuído faça sentido. Não se trata, no entanto, de um requisito fundamental para a correta operação do depurador. De fato, chamadas assíncronas são permitidas, mas com algumas limitações: essencialmente, o modo de

execução passo-a-passo não opera corretamente com mensagens assíncronas. De qualquer forma, intuitivamente, uma chamada assíncrona é equivalente a um *fork* do *thread* distribuído – i.e., é como se o *thread* distribuído fosse dividido em dois (Figura 5-22). No pior caso, se todas as chamadas forem assíncronas, o sistema volta a ser apresentado pelo depurador como uma coleção de *threads* disjuntos, eliminando as vantagens da abstração do *thread* distribuído enquanto ferramenta de depuração.

Os três requisitos apresentados sumarizam as funcionalidades exigidas do middleware e as hipóteses sobre a estrutura da aplicação que devem ser satisfeitas para que a técnica seja efetiva. Esses requisitos são atendidos pela maior parte das plataformas de middleware e aplicações baseadas em objetos distribuídos, o que significa que a técnica é de fato amplamente aplicável, ao menos do ponto de vista dos requisitos.

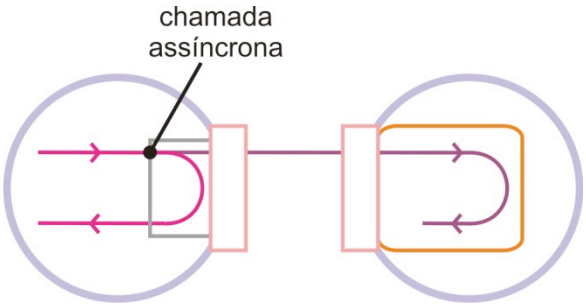

Figura 5-22. Divisão de um *thread* distribuído em uma chamada assíncrona.

Vamos agora sumarizar e discutir os requisitos impostos pela técnica sobre o ambiente de execução (e linguagem de programação) e sobre os depuradores simbólicos utilizados.

4. Deve ser possível identificar e instrumentar as classes que atuam como *proxies* e objetos remotos. Deve ser possível determinar o tamanho da pilha de chamadas durante a ativação dos interceptadores de instrumentação 2 e 3.

A instrumentação de *proxies* e objetos remotos é necessária para a implementação do mecanismo de rastreio e notificação. O tamanho da pilha de chamadas é necessário para que possamos "esconder" os quadros de ativação do middleware. De maneira geral, a dificuldade envolvida na implementação dos mecanismos de instrumentação (i.e., dos interceptadores de instrumentação) é inversamente proporcional à disponibilidade de mecanismos reflexivos na linguagem-alvo. Em linguagens como Python [55], Smalltalk [72], Ruby [2] ou Groovy [1], onde a própria linguagem provê facilidades para introspecção, reflexão comportamental e estrutural, a inserção de interceptadores e a coleta de informações é algo relativamente simples. No outro extremo temos linguagens como Java, cujas limitações têm que ser contornadas com mecanismos

desajeitados, como manipulação de *bytecodes* em tempo de carga e agentes de instrumentação [205].

Curiosamente, a implementação do mecanismo de instrumentação para aplicações escritas em C++ pode ser mais simples do que em Java. Embora a linguagem não disponibilize mecanismos explícitos para reflexão, uma aplicação C++ tem total liberdade para inspecionar e alterar seu próprio código e estado, incluindo instruções individuais em memória. A diferença com relação a Java, além do poder extra, fica por conta do fato que os "*bytecodes* C++" (as instruções de máquina) estão acoplados ao hardware subjacente, reduzindo a portabilidade da solução. Dito isso, existem na literatura exemplos bastante poderosos de bibliotecas de instrumentação para as principais arquiteturas de *hardware*, como o *JiTI* [172] para arquiteturas *x86* e o *KernInst* [211], para o *UltraSparc*. Para uma solução portável, poderíamos recorrer à instrumentação de código-fonte, com o auxílio de ferramentas como o OpenC++ [29].

5. Deve ser possível atribuir um identificador único a cada *thread* local. O identificador deve ser atribuído antes que o *thread* local se envolva em uma requisição remota, ou no tratamento de uma.

Também não se trata de um requisito difícil de atender, já que todas as linguagens consideradas disponibilizam algum tipo de biblioteca para *Thread-Specific Storage* [180]. Note que a implementação Java/CORBA provê garantias mais fortes do que as exigidas pelo requisito (5): na implementação Java/CORBA, todos os *threads* locais recebem identificadores, atribuídos pelo interceptador de instrumentação (1), imediatamente após serem iniciados. Para que o depurador funcione corretamente, no entanto, é necessário apenas garantir que a atribuição de identificadores aos *threads* locais aconteça antes que esses *threads* se envolvam em chamadas remotas. Para nossos experimentos envolvendo reprodução de execução (Seção 5.2.1), no entanto, isso não é suficiente – daí a necessidade do interceptador de instrumentação (1). Finalmente:

6. Deve haver uma ferramenta de depuração simbólica, que possa ser operada remotamente, disponível para o ambiente de execução da linguagem-alvo. Além disso, essa ferramenta de depuração simbólica deve disponibilizar, minimamente, as seguintes funcionalidades:
    a. Inspeção de *threads* locais e suas pilhas de chamadas,
    b. Operações *suspend* e *resume* em *threads* locais,
    c. *Breakpoints* absolutos (linha + nome do código fonte),
    d. Execução passo-a-passo (*step into*, *step over, step return*).

O requisito (6), atendido por todas as linguagens consideradas e por quase todas as linguagens conhecidas pelo autor, encerra a nossa discussão sobre portabilidade entre sistemas de *middleware* e linguagens de programação.

5.5 O agente central

A segunda grande peça – e a mais complexa – de nossa ferramenta é o agente central. O agente central desempenha um grande número de funções, dentre as quais podemos destacar:

1. Reconstrução das fotografias de cada *thread* distribuído, partindo dos eventos *server receive*, *server return*, e eventos de divisão, captados dos agentes locais e do controlador de processos (Seção 5.6).
2. Integração e coordenação de múltiplos depuradores simbólicos, possivelmente heterogêneos, nos protocolos interativos descritos na Seção 5.4.2.
3. Detecção de falhas e análise automática de certas informações de execução, como no caso da detecção automática de *deadlocks* distribuídos (Seção 5.5.7).
4. Controle (lançamento, término) e serviços de interação com processos remotos (entrada/saída padrão).
5. Disponibilização de uma interface gráfica com o usuário (GUI), com mecanismos de visualização e navegação como os discutidos na Seção 4.3.3.

5.5.1 Extensibilidade: mais sobre meta-sistemas

Conforme mencionamos já em algumas ocasiões durante o texto, um depurador é, antes de mais nada, um meta-sistema. Na Seção 5.3.1, apontamos que um sistema computacional incorpora representações de seu domínio – mas falamos muito pouco sobre a estrutura dessas representações. De fato, o espaço de todas as estruturações para representações de um domínio é tão grande quanto o espaço de possíveis *designs* para um sistema computacional. Existem, no entanto, uma série de heurísticas bem embasadas para o projeto de sistemas computacionais nos mais variados domínios.

Em linguagens orientadas a objeto, os elementos do domínio do sistema computacional são tipicamente transformados em um modelo de objetos, concebido em acordo com um conjunto de heurísticas para a divisão de responsabilidades. O conjunto de objetos resultante desse modelo corresponde à representação do domínio discutida na Seção 5.3.1. Em sistemas assim construídos, a manipulação das representações do domínio se faz mediante manipulações diretas aos objetos; isto é, ao envio de mensagens aos objetos que compõem a representação do domínio. Quando o sistema é causalmente conectado ao seu domínio, a manipulação do modelo de objetos produz efeitos correspondentes no domínio. De maneira semelhante, o conjunto de objetos que compõem as representações, bem como seus respectivos estados, variam continuamente em resposta a alterações no domínio.

No caso particular de um depurador construído em uma linguagem orientada a objeto, a representação do domínio é tipicamente produzida pela reificação dos elementos semanticamente

significativos na linguagem do sistema-alvo. Note que esse modelo de objetos compreende, essencialmente, uma API de metaprogramação. Ungar e Bracha [223] apresentam uma discussão bastante completa a respeito de características desejáveis para esses modelos de objetos. Um dos princípios apresentados por Ungar e Bracha – o da correspondência estrutural – dita que em uma boa API de metaprogramação, todos os elementos da linguagem-alvo são representados.

A primeira grande dificuldade envolvida no projeto de um depurador interativo para sistemas distribuídos heterogêneos é, portanto, a elaboração de um modelo de objetos que possa acomodar a natureza também heterogênea dos ambientes de execução e linguagens nas quais são escritos os processos que os compõem. Além disso, nosso modelo de objetos deve ser naturalmente extensível, de tal forma que novos ambientes de execução e linguagens de programação possam ser incorporados com o menor esforço possível. Em um artigo que discute o desenvolvimento de arcabouços orientados a objeto, Johnson [96] afirma que a flexibilidade de um arcabouço advém de um processo evolutivo. No início, a extensão o arcabouço é composto por um modelo de objetos de granulosidade grossa e as extensões se dão mediante a criação de novas subclasses (*white box*). Na medida em que o domínio é melhor compreendido, o modelo de objetos do arcabouço assume uma granulosidade mais fina, até um ponto em que o comportamento desejado pode ser obtido pela composição de objetos existentes (*black box*). Isso significa que o nosso depurador – que é, em sua composição, um arcabouço de depuração – dificilmente poderia atingir o nível de flexibilidade almejado sem que antes houvesse uma boa dose de trabalho em adaptações para outras linguagens.

5.5.2 O meta-modelo de depuração do Eclipse

O Eclipse [58] é uma plataforma de código aberto, genérica, extensível e amplamente utilizada no desenvolvimento de ambientes integrados de desenvolvimento (IDEs) e aplicações clientes "ricas" (*rich client applications*). Entre os arcabouços que são distribuídos com a plataforma, encontra-se um arcabouço de depuração. Esse arcabouço de depuração já foi estendido sob inúmeras circunstâncias, tendo provado ser capaz de acomodar versões reificadas dos principais elementos dos ambientes de execução de linguagens como Python, Ruby, C, C++, Java, Perl e até mesmo COBOL. O arcabouço inclui ainda uma interface gráfica extensível, que interage com uma API de metaprogramação genérica.

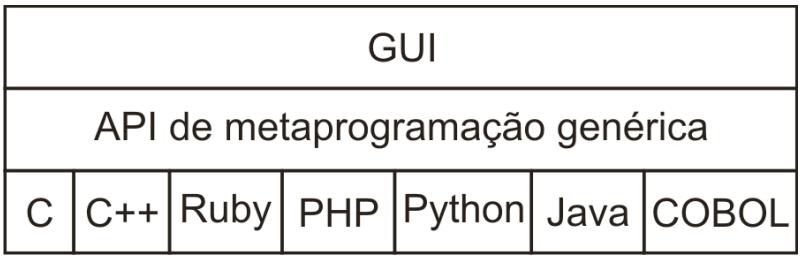

Figura 5-23. Estrutura geral do arcabouço de depuração do Eclipse.

A inclusão de uma nova linguagem se dá por meio da implementação de uma coleção de interfaces definidas pela API de metaprogramação. Em termos práticos, isso significa que adicionar uma nova linguagem ao arcabouço de depuração Eclipse equivale a prover novas implementações para a coleção de interfaces que especificam as reificações genéricas de elementos da linguagem alvo. Essas reificações genéricas são expressivas o suficiente para permitirem a construção de um depurador simbólico simples, ao mesmo tempo em que são genéricas o suficiente para acomodarem os elementos de linguagens bastante distintas.

Dito isso, a existência de tal modelo de objetos; i.e. de tal representação de nosso domínio, facilitou substancialmente o trabalho de desenvolvimento do depurador. Nosso trabalho prosseguiu então em duas fases. A primeira delas consistiu em representar o domínio do sistema distribuído, em prover uma implementação das interfaces de metaprogramação que apresentassem o sistema distribuído como um grande processo virtual, composto por uma coleção de *threads* distribuídos.

Dito isso, esse processo virtual é nada mais do que uma visão especial da coleção de processos que compõem de fato o sistema distribuído. Isso induz a segunda parte do trabalho: criar pontos de integração entre as representações já existentes – i.e., as implementações de depuradores já funcionais no Eclipse – e a nossa implementação. Para tanto, nós desenvolvemos um conjunto de extensões à API de metaprogramação genérica do Eclipse. A adição de uma nova linguagem e plataforma de *middleware* ao depurador consiste, portanto, em:

1. Escrever um cliente de depuração que implemente nossas interfaces de metaprogramação estendidas, ou adaptar um existente.
2. Implementar o maquinário de instrumentação na linguagem/plataforma de *middleware* alvos.

A arquitetura completa do agente central – incluindo o cliente do controlador de processos, que será discutido na Seção 5.6 – é mostrada, em linhas gerais, na Figura 5-24. Note que nosso depurador fica localizado entre os clientes de depuração e a plataforma, criando uma camada extra no arcabouço. A API de metaprogramação estendida (interfaces estendidas na Figura 5-24) contém, essencialmente, métodos que transmitem um conjunto de eventos relevantes ao depurador distribuído.

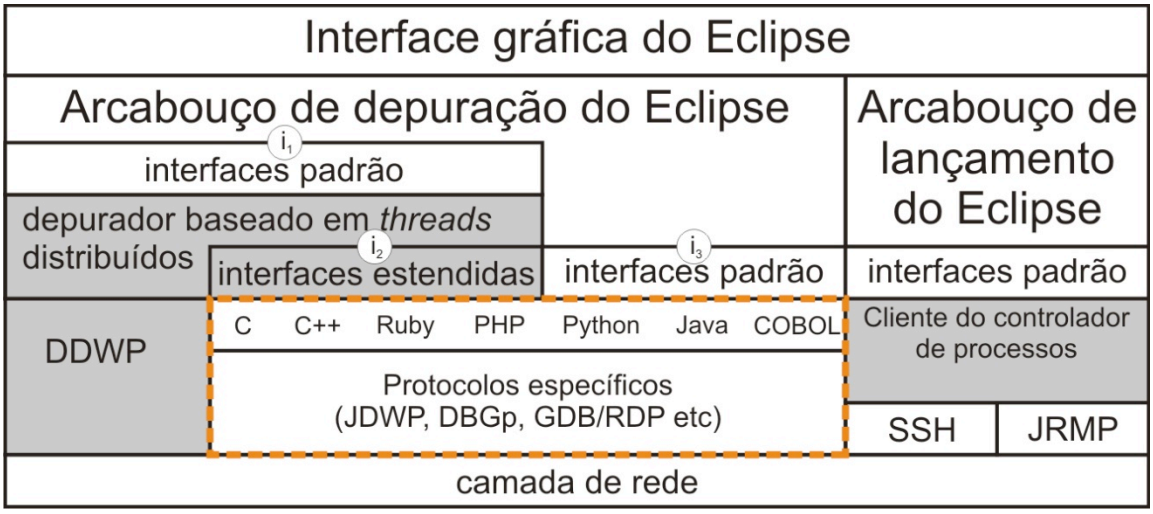

Figura 5-24. A arquitetura em camadas do agente central, com algumas interações demarcadas.

O depurador distribuído produz sua aproximação da execução distribuída juntando os eventos recebidos dos depuradores simbólicos específicos com os eventos DDWP. As operações interativas (suspend e resume, adição e remoção de breakpoints, operações de inspeção de variáveis, etc.) enviadas ao depurador distribuído são delegadas ao cliente adequado.

A Figura 5-25 mostra uma fotografia da interface gráfica do depurador distribuído. Note, na aba *Debug* (no canto superior esquerdo), a presença de dois elementos-raíz: um denominado (1) *Distributed System [Central Agent]* e outro, mais abaixo, denominado (2) *CORBA Name Server [Java Distributed Node]*. O elemento (1) provém da interação do arcabouço de depuração do Eclipse com a implementação do modelo de metaprogramação que mostra o sistema como uma coleção de *threads* distribuídos (interação i_1 na Figura 5-24).

Note, em destaque na figura, a pilha virtual (cuja construção conceitual foi discutida na Seção 4.3.3) do *thread* distribuído de identificador 5.3. O *thread* distribuído mostrado nessa figura, particularmente, tem sua pilha composta por 5 *threads* locais distintos, que pertencem a 5 processos distintos, distribuídos em duas máquinas. A composição da pilha acontece por meio da interação entre o depurador distribuído e os clientes de depuração (interação i_2 na Figura 5-24). Por último o elemento (2) da Figura 5-25 provém da interação direta entre o arcabouço de depuração do Eclipse e a implementação do modelo de metaprogramação provido pelo cliente de depuração Java (descrito na Seção 5.5.4), representada pela interação i_3 na Figura 5-24.

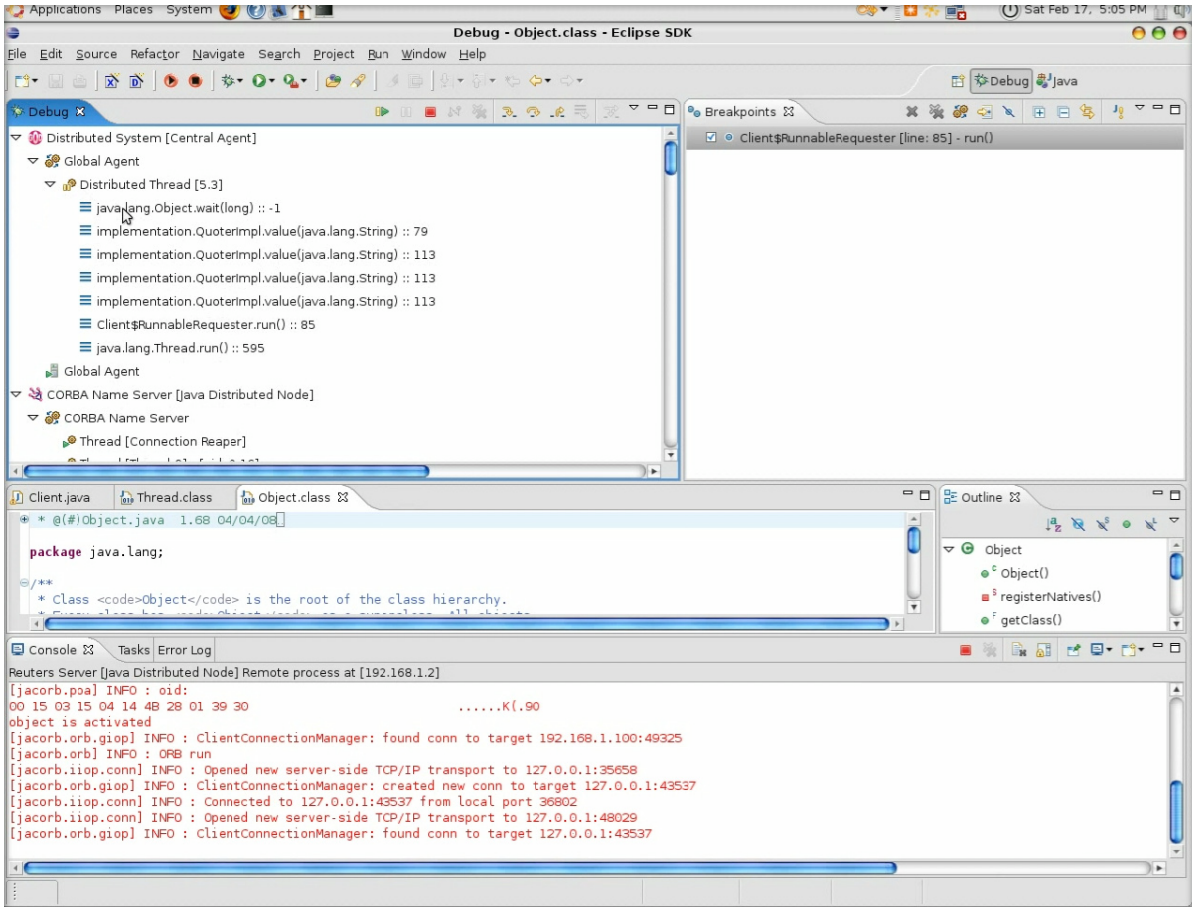

Figura 5-25. Fotografia da interface gráfica com o usuário (GUI) do depurador distribuído, mostrando o processo virtual global (*Distributed System*) e um processo local (*CORBA Name Server*).

Note que, embora o sistema considerado na Figura 5-25 seja homogêneo no que diz respeito à linguagem utilizada (trata-se de um sistema cujas porções sob inspeção pelo depurador são escritas todas em Java), tanto a arquitetura do depurador simbólico quanto a implementação atual do agente central permitem que sejam depurados sistemas distribuídos heterogêneos – i.e, compostos por partes escritas em múltiplas linguagens – desde que existam agentes locais e clientes de depuração funcionais para todos os ambientes de execução em consideração. Para entender de qual forma um sistema assim poderia ser acomodado, considere uma situação de depuração hipotética, apresentada na Figura 5-26, em que um *thread* distribuído suspenso em um cliente CORBA escrito em Java tem sua execução retomada, atingindo um *breakpoint* posicionado num servidor C++.

Nesse cenário hipotético temos, inicialmente, um *thread* distribuído *T* suspenso no Nodo 1 (Java) da Figura 5-26. O usuário do depurador simbólico distribuído requisita a retomada da execução do *thread* distribuído *T* por meio da interface gráfica. O comando de retomada, ilustrado pela flecha (1) na Figura 5-26, parte da interface gráfica e adentra o depurador simbólico baseado

em *threads* distribuídos, onde é mapeado em um comando de retomada (*resume*) dirigido ao *thread* local l_1, controlado pelo cliente de depuração Java. O cliente de depuração Java, por sua vez, mapeia o comando de retomada em um conjunto de pacotes JDWP que são então enviados, pela rede, ao depurador simbólico remoto, provocando o efeito desejado (retomada) em l_1.

O que é importante notar até aqui é que o depurador simbólico baseado em *threads* distribuídos é capaz de mapear, dinamicamente, comandos interativos direcionados a *threads* distribuídos (como o comando de retomada direcionado a *T*) em comandos interativos direcionados aos *threads* locais que compõem esses *threads* distribuídos. Esse mapeamento dinâmico é possível apenas porque o depurador simbólico **sabe** quais *threads* locais fazem parte de quais *threads* distribuídos a cada instante; isto é, porque o depurador simbólico rastreia de forma precisa as fotografias dos *threads* distribuídos no sistema.

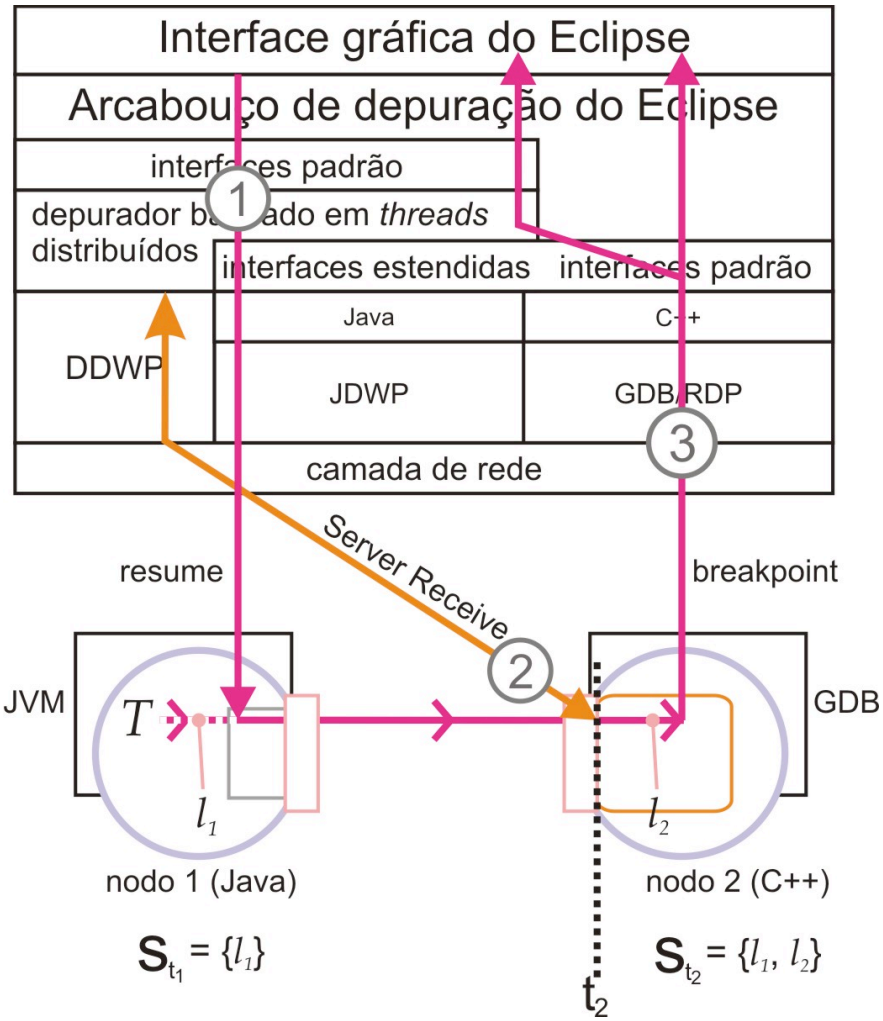

Figura 5-26. Dinâmica de interações entre agentes locais e agente central num sistema heterogêneo.

Retomada a execução de l_1, tem início uma requisição remota que, no instante t_2, atinge o objeto remoto C++ (Nodo 2). Neste instante, o código de instrumentação entrelaçado ao código do objeto remoto produz um evento *Server Receive* (flecha (2)), conforme aquilo que foi discutido nas Seções 5.3 e 5.4. Esse evento, transmitido ao agente central via DDWP, é então processado, fazendo com que o conhecimento do depurador a respeito da estrutura do *thread* distribuído *T* seja atualizado (a última fotografia de *T* conhecida pelo depurador simbólico passa a ser $s_{t_2} = \{l_1, l_2\}$).

Por último, o *thread* distribuído atinge um *breakpoint* posicionado no servidor, fazendo com que o depurador remoto ligado ao processo C++ (uma instância do *GNU Debugger*, ou GDB, no nosso exemplo) produza um evento. Esse evento será transmitido pela rede por meio de um protocolo específico do GDB (o RDP) até o cliente de depuração C++ (flecha (3) na Figura 5-26). O cliente de depuração C++ se encarrega então por processar o evento recebido e por encaminhar as notificações cabíveis ao depurador simbólico baseado em *threads* distribuídos (que atualiza o estado do *thread* distribuído *T* para "suspenso") e à interface gráfica (que atualiza o estado do *thread* local para "suspenso" também). O *thread* distribuído é suspenso.

Esse cenário mostra de qual forma o agente central pode interagir com um sistema heterogêneo – todas as requisições e todos os eventos, com a notável exceção dos eventos DDWP, são processados e traduzidos pelos clientes de depuração. O mesmo acontece com relação à visualização de trechos de um *thread* distribuído heterogêneo. A Figura 5-27 ilustra uma pilha virtual mista (a construção das pilhas virtuais foi discutida na Seção 4.3.3) e o processo de mapeamento de estados na visão de variáveis.

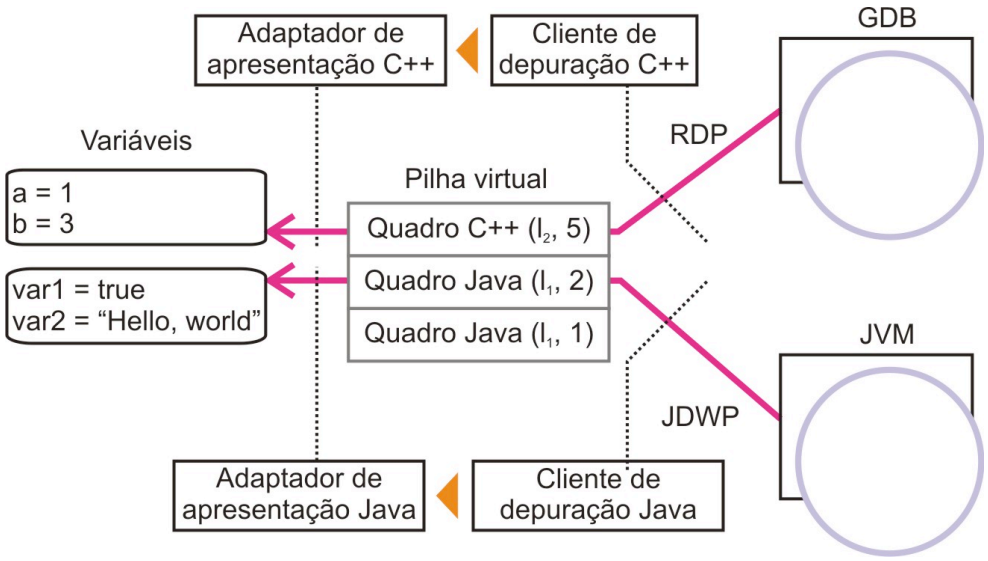

Figura 5-27. Exibição de estado misto em um *thread* distribuído heterogêneo.

Conforme pode ser visto na figura, os quadros de ativação armazenam duas informações importantes – (1) uma referência para o *thread* local que origina o quadro de ativação (l_1 para os quadros Java e l_2 para o quadro C++, no exemplo) e (2) a posição do quadro de ativação na pilha de chamadas desse *thread* local. Quando o usuário solicita a inspeção de um quadro de ativação da pilha virtual, o depurador simbólico baseado em *threads* distribuídos utiliza as informações armazenadas no quadro de ativação para solicitar ao cliente de depuração o conjunto de variáveis visíveis. Esse conjunto de variáveis visíveis é então apresentado com a ajuda de um adaptador específico para a linguagem de origem do quadro de ativação, capaz de interpretar os dados provenientes do GDB e transformá-los em dados que possam ser apresentados pela interface gráfica do Eclipse. O adaptador de apresentação específico de uma linguagem é provido pelo cliente de depuração daquela linguagem.

5.5.3 Mais sobre o modo de execução passo-a-passo

Apresentada a organização do agente central e a relação, em linhas gerais, entre o depurador baseado em *threads* distribuídos, os clientes de depuração específicos de cada linguagem e o arcabouço de depuração do Eclipse, podemos partir para uma discussão um pouco mais detalhada do mecanismo de execução passo-a-passo para *threads* distribuídos. O modo de execução passo-a-passo resulta, no agente central, de uma coordenação entre informações coletadas pelos clientes de depuração específicos de cada linguagem e informações trazidas via eventos DDWP. Nós vamos explicar o funcionamento do mecanismo tomando como base o cliente de depuração Java – portanto, parte da explicação será específica para esse cliente. A porção que diz respeito ao depurador baseado em *threads* distribuídos, no entanto, é genérica, sendo válida na interação com qualquer cliente de depuração.

Vamos começar analisando o mecanismo de *step into*. A Figura 5-28 (a) mostra o tratador de eventos de execução passo-a-passo no cliente de depuração Java (modificado). Eventos do tipo *JavaStepEvent* são emitidos sempre que um passo na execução passo-a-passo termina.

```
On Step Event (JavaStepEvent E)
01 M := extraia informações de localização de E.
02 L := extraia a referência JDI, contida em E, ao thread
03      local.
04 se a localização M está em um proxy remoto então
05    execute uma operação resume for remote stepping
06    no thread local L.
```

(a)

```
On Server Receive (Event E):
01 G_id := extraia de E o id do thread distribuído.
02 L_id := extraia de E o id do thread local envolvido
03       no evento.
04 N_proxy := controlador do nodo que contém o thread
05          local de id L_id.
06 se o thread distribuído de id G_id estiver marcado
07 como em modo de execução passo-a-passo então
08   Peça a N_proxy que posicione um breakpoint para o
09   thread de id L_id na localização apontada por OP_id.
```

(b)

Figura 5-28. Tratamento de eventos de execução passo-a-passo no cliente de depuração Java (a) e tratamento de eventos DDWP do tipo *Server Receive* no depurador baseado em *threads* distribuídos (b).

Conforme mostra o pseudo-código na Figura 5-28, o tratamento de um evento de execução passo-a-passo começa com um exame da localização final do quadro de ativação no topo da pilha de chamadas do *thread* local envolvido no evento (atribuído ao identificador *L* no código). Se a localização do quadro de ativação em questão for um *stub*, o tratador de eventos executa uma operação especial no *thread* local, chamada *resume for remote stepping*. Essa operação equivale a um *resume* normal, mas omite a propagação do estado para o *thread distribuído*. Assim, é como se o *step into* no *thread* distribuído ainda não tivesse terminado – o que é de fato o que queremos, já que o *step* remoto só termina, na ausência de falhas, no lado do servidor, com a entrada de um *thread* local no objeto remoto.

Na presença de falhas, no entanto, é possível que a chamada termine antes de atingir o servidor (no caso, por exemplo, em que o servidor não está mais ativo). Nessas situações, é necessário bloquear novamente o *thread* local, já que o modo de execução passo-a-passo deve ser retomado imediatamente no cliente. Para tanto, nós incluímos um código de *status* na resposta do evento de divisão, que será gerado pelo *proxy*. Esse código de status indica ao agente local o *thread* local deve ser suspenso novamente (chamando o método `stepMeOut()`, que descrevemos na Seção 5.4.2).

A Figura 5-28 (b) mostra, em pseudo-código, parte do tratador de eventos DDWP para eventos do tipo *Server Receive*. Essencialmente, o tratador (1) obtém uma referência ao controlador de nodo que contém o *thread* local que trata a chamada remota e (2) repassa as informações recebidas no evento (essencialmente o conteúdo, opaco, de OP_{id}) para o controlador. O controlador de nodo é um elemento provido pelo cliente de depuração e implementa uma das interfaces de nossa API estendida de metaprogramação. Isso significa que não há quebra de encapsulamento, já que a interface do controlador de nodos independe da linguagem para qual é destinada o cliente de depuração.

Ao receber as informações do tratador de eventos, o controlador de nodo (Java, no nosso exemplo) posiciona um *breakpoint* na primeira linha do método especificado em OP_{id} (o controlador de nodo, sendo específico para Java, sabe interpretar as informações contidas em OP_{id}) e devolve o controle ao tratador de eventos. O tratador de eventos responde então ao agente local, fazendo com que o *thread* local que trata a chamada remota desbloqueie e atinja o *breakpoint* posicionado (lembre-se da Figura 5-20). Ao atingir esse *breakpoint* – que recebe uma marca especial – o tratador de *breakpoints* detecta que se trata do término de uma operação de *stepping* remoto e notifica o *thread* distribuído. O protocolo chega então ao fim, produzindo a ilusão de que, de fato, o passo que sucede uma chamada remota é a execução do método no objeto remoto.

O protocolo para passos de execução que encerram métodos remotos – e geram eventos *Server Return* – é bastante distinto do protocolo descrito para operações *step into* que entram em *stubs*. Note que tanto um *step into*, quando um *step over*, quanto um *step return* podem provocar o término de um método, portanto o processo que vamos descrever se aplica a todos os modos de execução passo-a-passo em potencial. Para entendermos o funcionamento do mecanismo, é preciso que nos recordemos do protocolo mostrado na Figura 5-21. Note, na Figura 5-21, que o evento *Server Return* é emitido após o início do passo que encerra o método, mas antes do final. Para que possamos conseguirmos esse efeito, é necessário:

1. Inserir uma instrução que desvie o fluxo de controle para o Trecho (2) do interceptador de instrumentação (3) antes de cada instrução de retorno, bem como quando houver uma exceção (é preciso compilar um bloco *finally* [74]).
2. Atribuir informações de depuração aos *bytecodes* gerados durante a instrumentação, de tal modo que o depurador execute o Trecho (2) do interceptador de instrumentação (3) como parte da última linha de código do método.

Isso nos garante que <u>sempre</u> que uma ativação de um método remoto se encerra durante um passo no modo de execução passo-a-passo, há uma requisição ainda pendente durante o processamento do evento *Server Return*. É justamente essa a condição testada pelas linhas 05 e 06

do tratador de eventos DDWP, mostrado na Figura 5-29 (as operações para teste e remoção de requisições pendentes fazem parte de nossas extensões à interface de metaprogramação). Se ficar determinado que existe uma requisição pendente, essa requisição será removida e uma resposta enviada ao agente local, informando que a requisição deve ser suspensa no cliente. Conforme foi explicado na Seção 5.4.2, essa informação será então propagada de volta ao cliente, onde a requisição será novamente suspensa (pelo Trecho (2) do interceptador de instrumentação (2), mostrado na Figura 5-16 (b)).

```
On Server Return (Event E)
01 G_id := extraia de E o id do thread distribuído.
02 T := referência ao thread distribuído de id G_id.
03 L := referência ao thread local que é cabeça de T.
04
05 se há uma requisição de execução passo-a-passo
06 pendente em L então
07    remova a requisição de execução passo-a-passo
08    pendente em L.
09    envie uma resposta ao agente local que origina E
10    informando que o thread distribuído está em modo
11    de execução passo-a-passo.
12 senão então
13    envie uma resposta ao agente local que origina E
14    informando que o thread distribuído está em modo
15    de execução normal.
```

Figura 5-29. Tratamento de eventos do tipo *Server Return*.

5.5.4 Particularidades na integração com o depurador da JPDA

Conforme mencionamos anteriormente, nosso depurador simbólico é composto por uma parte genérica – que produz a visão dos *threads* distribuídos e cujo protocolo de comunicação é o DDWP – e por uma coleção variável de clientes de depuração adaptados, que provêm alguns serviços a essa parte genérica. Para que pudéssemos testar a viabilidade do arcabouço e da própria implementação do maquinário de sustentação do DDWP era necessário, no entanto, que integrássemos ao menos um cliente de depuração e plataforma de *middleware* a esse arcabouço. Escolhemos Java como a linguagem para o cliente de depuração e CORBA (qualquer implementação Java) como plataforma de *middleware*.

A escolha foi feita porque Java é uma linguagem amplamente utilizada, porque o protocolo de depuração remota para implementações da linguagem é padronizado e aberto [202] e porque existe um mapeamento também padronizado e aberto do protocolo num modelo de objetos de lado do cliente [207, 223]. Um outro argumento em favor de Java é a existência de um cliente de depuração maduro e já adaptado para o Eclipse – o depurador simbólico do JDT [57]. O problema é que o cliente de depuração Java do JDT é um componente de software bastante complexo. Num primeiro

momento nós optamos, portanto, por escrever nosso próprio cliente de depuração baseado na JDI. A relação entre máquina virtual, protocolo de comunicação (JDWP), modelo de objetos de depuração Java (JDI) e interfaces de metaprogramação do Eclipse (padrão e estendidas) é mostrado na Figura 5-30. Note que, exceto pela compatibilidade com as interfaces estendidas, essa arquitetura reflete também a estrutura do depurador Java do JDT.

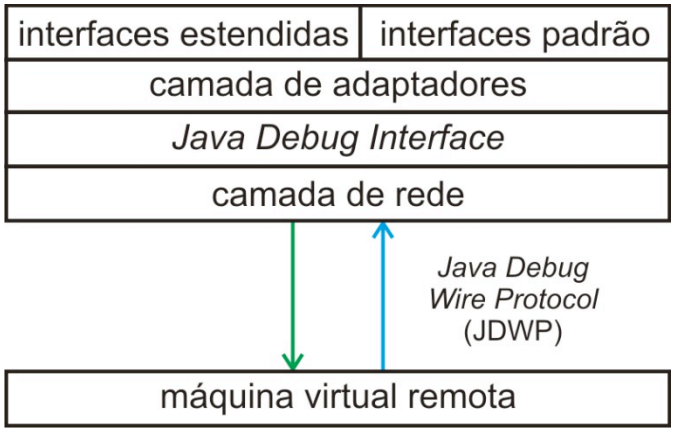

Figura 5-30. Arquitetura do cliente de depuração Java.

A integração da JDI com as interfaces de metaprogramação padrão e estendidas foi bastante direta. A única dificuldade, resultante da completa separação entre nível base e meta-nível imposta pela JPDA, foi na implementação dos mecanismos de reflexão distribuída requeridos pelo depurador. A JPDA não permite, essencialmente, que o nível base "fale" a respeito dos elementos que compõem seu ambiente de execução com o meta-nível, simplesmente porque as representações são diferentes nos dois níveis [223]. O problema surge porque precisamos mapear as reificações acessíveis pelo meta-nível – instâncias de `com.sun.jdi.ThreadReference`, como mostra a Figura 5-31 – nos identificadores numéricos atribuídos a cada *thread* local pelo interceptador de instrumentação (1) (Seção 5.4.1).

Como então pode o nível base comunicar ao meta-nível qual o identificador numérico atribuído a um certo *thread* local, se o nível base não é capaz de dizer ao meta-nível a qual *thread* local ele se refere? A resposta parte de duas observações:

1. Embora o nível base não tenha acesso às representações do meta-nível, o meta-nível tem acesso integral ao estado do nível base. O meta-nível pode, inclusive, utilizar *threads* suspensos por *breakpoints* para invocar qualquer trecho de código visível por aquele *thread*.

Isso significa que, embora o nível base não possa dizer ao meta-nível a qual instância de `com.sun.jdi.ThreadReference` um certo identificador se aplica, o meta-nível pode ler essa informação diretamente do nível base, usando cada instância para acessar o repositório de

identificadores, que tem uma interface padronizada e é globalmente acessível (lembre-se da discussão da Seção 5.2 a respeito da criação de estruturas mais facilmente inspecionáveis).

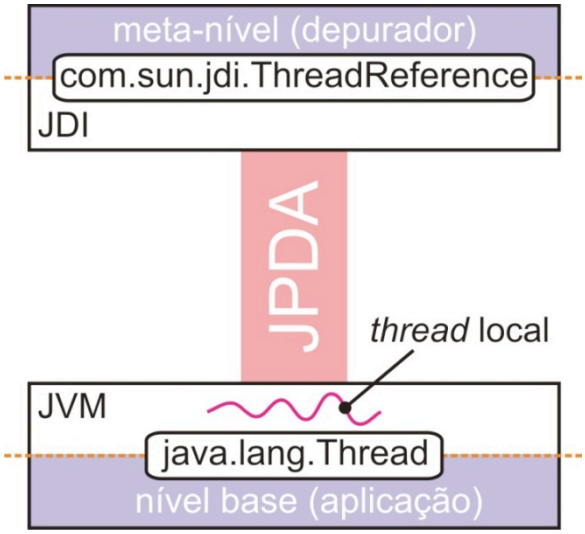

Figura 5-31. As duas representações de um *thread* local.

Os *threads* locais não nascem com identificadores, no entanto. Em particular, é possível que o meta-nível tente ler o identificador antes da ativação do interceptador de instrumentação (1). Portanto, resta ainda determinar como dizer ao meta-nível quando ler o identificador. A segunda observação nos dá a resposta:

2. Os eventos de *breakpoint* produzidos pela JPDA vêm com uma referência ao *thread* local suspenso anexada. Essa referência é uma instância de `com.sun.jdi.ThreadReference`.

Podemos, portanto, utilizar os *breakpoints* como uma forma indireta de comunicação, conforme mostra o protocolo de mapeamento da Figura 5-32. Para tanto, atribuímos, na inicialização de cada processo, um *breakpoint* a um método especial, vazio, chamado pelo método `tagCurrent()` após a atribuição do identificador ao *thread* local. Quando o *breakpoint* é atingido (evento (2) na a Figura 5-32), o agente central extrai a referência JDI do evento e a utiliza para acessar o repositório de identificadores (eventos (3) e (4)), mapeando, dessa forma, a referência JDI no identificador numérico atribuído pelo nível base. Isso estabelece uma linguagem comum entre depurador e depurado para *threads* locais. O estabelecimento dessa noção comum dá ao depurado o poder de solicitar, via DDWP, operações ao meta-nível que antes não seriam possíveis, como a suspensão de um *thread* local. Nós esperamos que esse problema seja recorrente, i.e., que a adição de novas linguagens leve a problemas semelhantes. Felizmente, no entanto, a solução pode também sempre ser a mesma, já que não há nada específico a Java nessa solução.

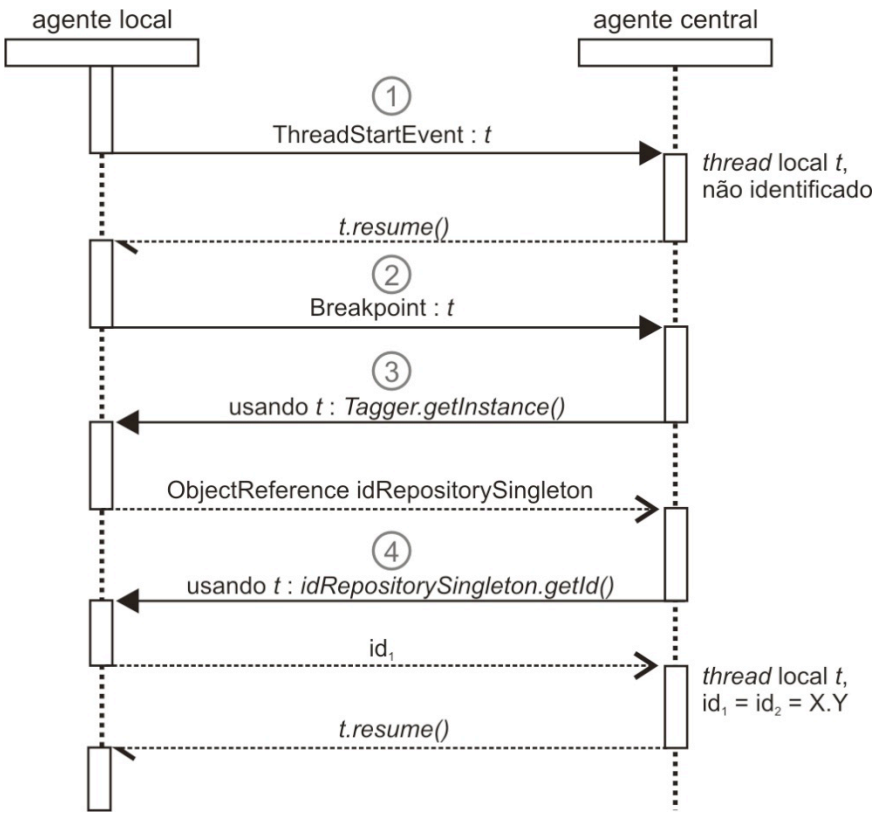

Figura 5-32. Protocolo de registro do *thread* local.

5.5.5 Portátil, interoperável ou extensível?

No decorrer deste texto mencionamos, em diversas ocasiões, que um dos objetivos deste trabalho é o de produzir uma ferramenta extensível e portátil. Segundo o glossário da IEEE [92]:

- "*[Portability is] the ease with which a system or component can be transferred from one hardware or software environment to another*";
- "*[Extendability or extensibility is] the ease with which a system or component can be modified to increase its storage or functional capacity.*"

Extensível, em nosso contexto, diz respeito a uma característica bastante clara: à capacidade da ferramenta em acomodar novas linguagens e ambientes de execução. A extensibilidade da ferramenta é derivada do modelo de meta-programação e do arcabouço de depuração do Eclipse, projetados naturalmente para a acomodação de novas linguagens e sendo, portanto, extensíveis. A portabilidade, no entanto, se dá em 2 níveis, com significados distintos:

1. Portabilidade dos clientes de depuração: deve ser mais simples adaptar um cliente de depuração pronto do que escrever um novo do zero;

2. Desenvolvimento dos agentes locais: o desenvolvimento do maquinário de instrumentação e coleta de dados exigido pelos agentes locais deve exigir pouco esforço.

O nível (1) é bastante claro. O nível (2), no entanto, pode aparentar ser uma interpretação equivocada de portabilidade num primeiro momento. É preciso notar, no entanto, que nosso depurador simbólico, embora distribuído, compreende uma única peça de software. Embora a definição do glossário da IEEE não mencione reuso de código (i.e., poderíamos afirmar, pela definição, que se um sistema ou componente é simples a tal ponto que reescrevê-lo em múltiplos ambientes de hardware e software é simples, então esse componente é portátil), a compreensão usual de software portátil é de uma peça de software não-trivial que pode ser adaptada com pouco esforço – ou com a reescrita de apenas parte dessa peça de software – a novos ambientes de hardware e software. Essa intuição é ilustrada na Figura **5-33**, que mostra um sistema de software composto por uma porção portátil e uma porção não-portátil. A porção portátil é levada sem mudanças entre os ambientes de hardware e software, ao passo que a porção não-portátil tem que ser reescrita.

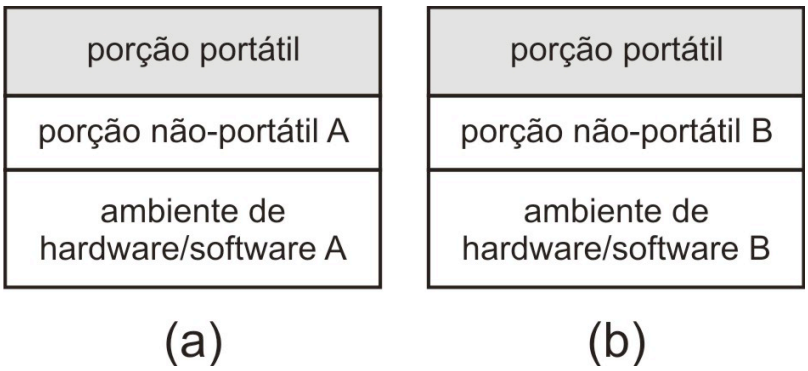

Figura 5-33. Sentido usual de portabilidade.

Destacamos que, de acordo com a noção intuitiva usual, o sistema de software ilustrado na Figura **5-33** será "portátil" se o esforço de reescrita da porção não-portátil for pequeno. Além disso, no sentido usual, não basta que esse esforço de reescrita seja pequeno – é preciso que a porção portátil represente uma parte significativa (talvez a maior parte) do sistema de software. Atentamos agora à semelhança entre os sistemas de software mostrados na Figura **5-33** e o sistema de software da Figura 5-34 (a).

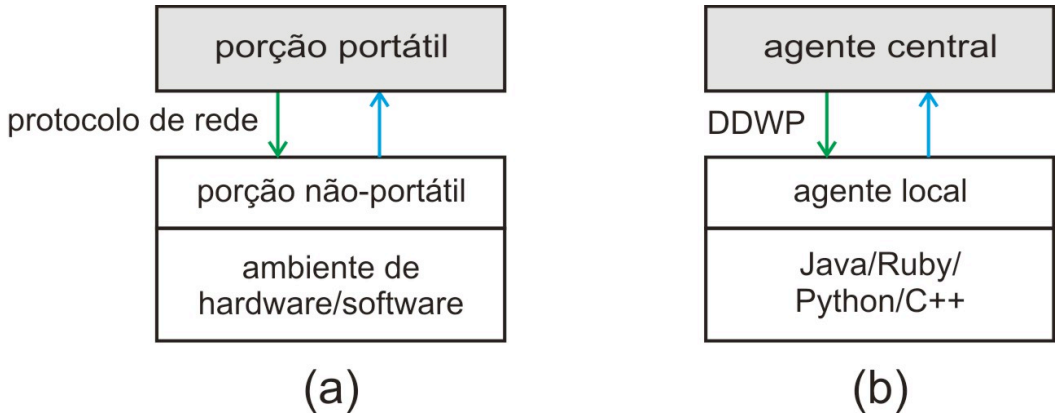

Figura 5-34. (a) Porções portátil e não-portátil separadas por um protocolo de rede. (b) Arquitetura do G.O.D.

Note que a única diferença entre o sistema de software mostrado na Figura 5-34 (a) e os sistemas da Figura **5-33** ficam por conta do fato que, na Figura 5-34 (a), a porção portátil é separada da porção não-portátil por um protocolo de rede. Ambas porções pertencem, no entanto, ao mesmo sistema e, portanto, os mesmos critérios a respeito de portabilidade se aplicam: se a porção não-portátil for comparativamente pequena e simples de reescrever quando comparada à porção portátil, então o sistema é considerado "portátil" como um todo. Agora atentamos para o fato de que o sistema mostrado na Figura 5-34 (a) é precisamente o nosso depurador simbólico, mostrado na Figura 5-34 (b) com os componentes renomeados de acordo com a terminologia de nossa arquitetura.

O maquinário de coleta e transmissão de eventos DDWP é, portanto, a parte "não-portável" de nosso sistema de software. Essa porção "não-portável" é simples de reescrever para outros ambientes e linguagens de programação e representa apenas uma pequena parte do sistema de software – o agente central é muito mais complexo, tanto em números de linhas de código quanto em responsabilidades e funcionalidades. Portanto, nossa ferramenta é "portátil" no sentido usual e o sentido da palavra portabilidade, como aplicado aos agentes locais, é compatível com esse sentido usual.

5.5.6 Considerações sobre escalabilidade

Uma das questões mais preocupantes, desde o início, diz respeito à escalabilidade da implementação do agente central. Quando falamos em escalabilidade, neste contexto, nos referimos especificamente à escalabilidade de desempenho – quantos nodos pode o depurador gerenciar antes que seu desempenho comece a se degradar? Qual o comportamento do *throughput* do servidor em situações de saturação? O fato é que não produzimos nenhum tipo de análise quantitativa capaz de

responder a essas perguntas, mas acreditamos que a implementação seja satisfatoriamente escalável mesmo assim. Vamos começar apresentando a dinâmica do processamento de requisições na primeira versão depurador – que já era capaz de lidar com algumas dezenas de nodos – e, em seguida, vamos apresentar uma segunda versão, discutindo algumas das limitações dos mecanismos da POO Java para programação concorrente e nossas soluções para essas limitações. O argumento de que o processamento de requisições deve escalar de maneira satisfatória sai dessa discussão.

5.5.6.1 Versão 1: Eventos com inversão de controle

O agente central é, antes de mais nada, um aglomerado de processadores de eventos. Podemos identificar, essencialmente, as seguintes fontes de eventos:

1. Os clientes de depuração baseados na JDI.
    a. Indicam a morte de *threads* locais ou máquinas virtuais.
    b. Indicam a perda da conexão com uma máquina virtual remota.
    c. Indicam que *threads* locais tiveram sua execução suspensa ou retomada (*resumed*).
2. O controlador de processos.
    a. Indica a morte de processos (e seus *threads* correspondentes).
    b. Indica a perda da conexão com um nodo de processamento remoto.
3. O receptor de eventos DDWP.
    a. Indica mudanças no estado dos *threads* distribuídos (por meio de eventos do tipo *server receive, server return* e eventos de divisão).

Existe um agregado de objetos funcionalmente relacionados no agente central, que vamos denominar **núcleo**, que é responsável pela reconstrução do estado global a partir dos eventos emitidos por essas três fontes. A primeira versão do núcleo foi baseada em uma combinação direta e ingênua dos padrões *Observer* e *Chain of Responsibility* [69]. Nesta versão, essencialmente, cada fonte de eventos disponibiliza um ou mais canais de notificação para seus eventos. Os eventos são entregues por inversão de controle – i.e, as fontes de eventos se responsabilizam por chamarem os métodos adequados dos tratadores [78]. O **núcleo,** interessado essencialmente em todos os eventos produzidos, é registrado como ouvinte em todos os canais. Para completar nossas dificuldades, a GUI do Eclipse – baseada no MVC [69] – faz uso pesado de *multithreading* durante a leitura do estado do modelo, levando à situação mostrada na Figura 5-35. Os problemas com esse projeto são muitos, dentre os quais, podemos destacar:

1. **O núcleo é programado de forma defensiva:** o núcleo é composto por uma coleção de objetos, cada qual passível de penetração por uma quantia potencialmente grande de *threads* simultâneos. O resultado é uma programação defensiva, com políticas de *locking* conservadoras (limitando o paralelismo) e dominada por paralelismo não estruturado, de

granulosidade fina e com abundância compartilhamento de estado entre *threads*. O código resultante torna-se pouco legível e pouco confiável [11, 38].

2. **Conciliar a semântica de todos os eventos é difícil:** alguns eventos – principalmente aqueles que são específicos de alguma linguagem, como os eventos JDI, não podem ser repassados diretamente ao núcleo. O modelo de *threads* e memória compartilhada, aliado à arquitetura em camadas e à inversão de controle, nos induziram a transformar a entrega de eventos – principalmente nas camadas mais internas e genéricas do núcleo – em chamadas de métodos. Isso induz uma não-uniformização na representação dos eventos, que torna o código difícil de manipular. Além disso, acessos de leitura (pela GUI) e escrita (pelos métodos que processam eventos) tornam-se uniformes sob esse modelo, dificultando a distinção entre os dois.

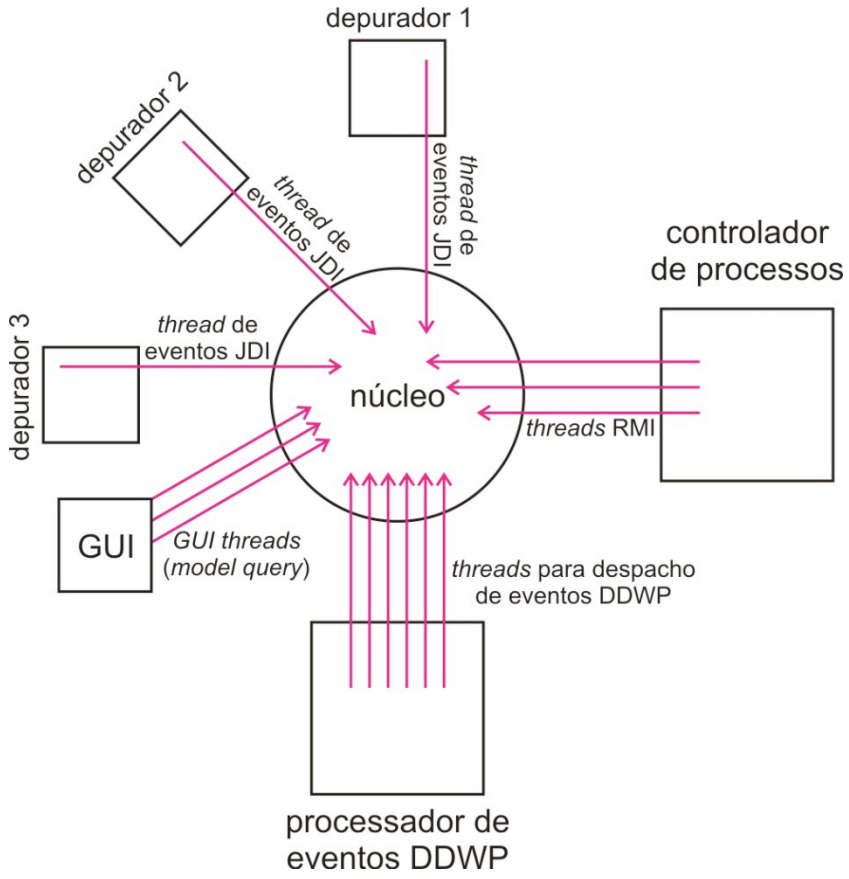

Figura 5-35. Entrega de eventos na primeira versão do núcleo do depurador.

3. **O padrão *Observer* induz a inversão de controle em Java:** os problemas que descrevemos resultam da maciça penetração de *threads* no núcleo e do uso de chamadas de métodos como forma de representação de eventos. Isso ocorre, em parte, pelo fato de que o padrão *Observer* induz a entrega de eventos dessa forma e porque, em Java, o envio de uma

mensagem de um objeto a outro objeto é semanticamente equivalente a uma chamada de função.

5.5.6.2 Versão 2: Passagem de mensagens

Tomando como base nossa experiência anterior, decidimos produzir um novo projeto, onde:

1. O escopo dos *threads* fosse limitado, evitando o acesso concorrente descontrolado e contraproducente do núcleo.
2. A representação de eventos fosse explícita.
3. Acessos de leitura e escrita pudessem ser diferenciados.

A arquitetura resultante – ainda em desenvolvimento – é mostrada na Figura 5-36. Essa arquitetura, que tem a sua inspiração em modelos de programação concorrente baseados em atores [5, 11, 77], procura agrupar *threads* funcionalmente relacionados em macro-componentes, que passam a se comunicar por meio de passagem de mensagens assíncronas. No caso do nosso depurador, isso nos permitiu isolar os *threads* de despacho de eventos de cada depurador (e do processador de eventos DDWP) dos *threads* que alteram o estado do núcleo, viabilizando um controle mais explícito do estado compartilhado e simplificando a sua implementação. Ao contrário de linguagens como Scala [77, 78] e Erlang [11], no entanto, Java não dispõe de mecanismos para passagem de mensagens assíncronas entre objetos, o que nos obriga a abrir mão da sintaxe limpa de chamada de métodos provida pela linguagem em favor do uso de filas de mensagens explícitas.

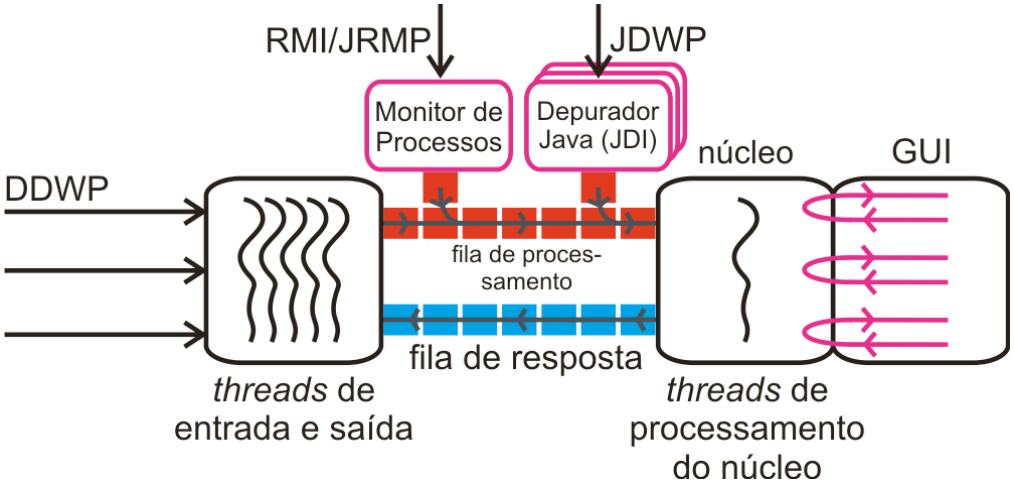

Figura 5-36. Nova arquitetura do processamento de requisições.

Na Figura 5-36, podemos identificar quatro grupos de *threads*:

1. ***Threads* de entrada e saída:** responsáveis pela leitura e inserção de eventos DDWP na fila de processamento do núcleo, bem como pelo envio das mensagens de resposta, enfileiradas pelo núcleo, aos agentes locais. Os *threads* de entrada e saída são alocados em estilo *round-robin* [213] à leitura de eventos DDWP das conexões TCP criadas por cada agente local. A

alocação é conduzida de tal forma que um *thread* fica atribuído a uma única conexão apenas pela duração da leitura de um evento. Sob condições de latência heterogênea, uma granulosidade mais fina pode ser desejável. Atualmente, no entanto, a única forma de lidar com a perspectiva de bloqueio por tempo excessivo de um *thread* de leitura é aumentando o número máximo de *threads* de entrada e saída. Pode haver um número pontencialmente grande de *threads* de entrada e saída.

2. ***Threads* de despacho de eventos:** responsáveis pela leitura e despacho de eventos de outras fontes que não as requisições DDWP, como no caso dos *threads* RMI do controlador de processos (Seção 5.6) e dos *threads* que despacham eventos da JDI. Novamente, na nova arquitetura, a influência desses *threads* é confinada aos seus macro-componentes – a comunicação entre o núcleo e as fontes de evento se dá apenas pela fila de processamento. A interação entre essas fontes de eventos e o núcleo é sempre de mão única, já que não é preciso que haja resposta a eventos que indicam a morte de *threads* e processos (lembre-se que apresentamos, no início desta seção, a natureza dos eventos produzidos por cada fonte). Pode haver um número potencialmente grande de *threads* de despacho de eventos.
3. ***Threads* de processamento do núcleo:** responsáveis pela atualização do estado das representações dos *threads* distribuídos a partir dos eventos depositados na fila de processamento. Responsáveis ainda por gerar os eventos específicos da plataforma Eclipse que disparam atualizações da interface gráfica, quando necessário. O número de *threads* de processamento do núcleo pode ser menor ou igual ao número de processadores disponíveis no hardware que executa o agente central já que, conforme veremos adiante, o processamento dos eventos de atualização é, na ausência de eventos de divisão, limitado apenas pelo poder computacional disponível.
4. ***Threads* da interface gráfica:** responsáveis por obter das interfaces de metaprogramação (parte do núcleo) as informações necessárias para a atualização das representações gráficas apresentadas ao usuário. Há um número desconhecido – e potencialmente grande – de *threads* da interface gráfica. A representação dos *threads* da GUI na Figura 5-36 não é muito precisa – o escopo desses *threads* é muito maior. A arquitetura em camadas delega chamadas ao modelo genérico de objetos do núcleo às instâncias específicas de cada linguagem.

O desempenho geral do depurador depende tanto da capacidade de geração das fontes de eventos quanto da capacidade de processamento do núcleo e da contenção de travas geradas pelos *threads* da interface gráfica. A escalabilidade das fontes de eventos é limitada pela capacidade de entrada e saída do hardware e pela latência da rede e não será considerada nesta discussão. No

restante desta seção, vamos mostrar que o processamento dos eventos no núcleo pode quase sempre ser feito de forma independente, procurando assim justificar nossa expectativa de que a abordagem seja de fato escalável.

O agente central organiza o estado global do sistema distribuído como uma tabela de *hash*, que mapeia o identificador global de cada *thread* distribuído em um conjunto de estruturas de dados (essencialmente pilhas acrescidas de algumas operações especiais). Os eventos DDWP processados pelo núcleo se aplicam, invariavelmente, a alguma das entradas nessa tabela, conforme mostra a Figura 5-37.

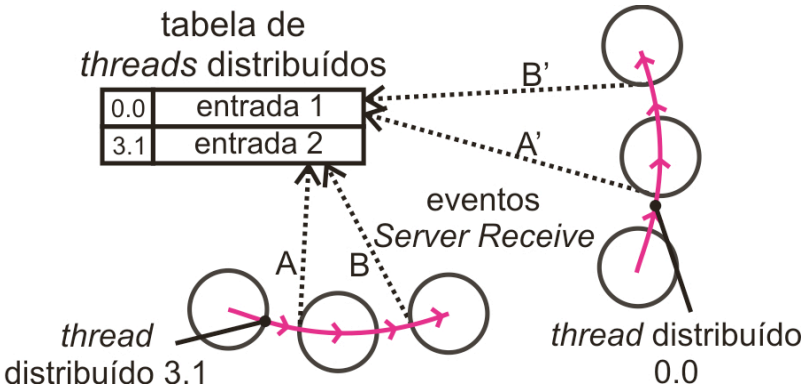

Figura 5-37. Atualização concorrente de representações de *threads* distribuídos.

Para entender porque a maior parte das atualizações pode ser de fato processada de forma concorrente, é preciso notar que:

1. Na ausência de falhas, eventos DDWP do tipo *server receive* e *server return* emitidos por um mesmo *thread* distribuído são <u>sempre</u> causalmente relacionados (i.e., os eventos gerados por um mesmo *thread* distribuído <u>nunca</u> serão, na ausência de falhas, concorrentes).
2. O mecanismo de notificação dos eventos DDWP é síncrono (Seção 5.3.3).

A implicação dessas duas observações é que, na ausência de falhas, <u>nunca</u> haverá na fila de processamento, num dado instante, mais de um evento de atualização destinado a um mesmo *thread* distribuído. Isso quer dizer que os eventos serão <u>sempre</u> direcionados a entradas distintas da tabela e podem, em princípio, ser processados simultaneamente. A Figura 5-37 mostra eventos *server receive* pertencentes a dois *threads* distribuídos distintos – 3.1 e 0.0 – e seu relacionamento com entradas disjuntas da tabela do agente central.

Há uma única circunstância em que um evento DDWP pode bloquear o processamento de um outro evento não causalmente relacionado a ele: quando um dos eventos provoca uma alteração <u>na tabela,</u> e não em apenas alguma entrada dela. A tabela é alterada sempre que um *thread* distribuído é adicionado (nasce) ou removido (termina). Portanto, o processamento de um evento DDWP pode

ser momentaneamente bloqueado durante a leitura da tabela pelo processamento de um segundo evento que altera tabela. Ainda assim, a contenção pode ser reduzida nesses casos com o uso de uma técnica conhecida como *lock striping* [161], onde o estado mutável da tabela é particionado em uma quantidade variável de listras (*stripes*), cada qual protegida por uma trava separada. A implementação atual do agente central faz uso do *lock striping* para reduzir a contenção nos acessos que alteram a tabela.

Na presença de falhas, é possível que mais de um evento seja direcionado a uma única entrada na tabela. Nesse caso, os eventos têm que ser processados de forma serial, o que implica que as representações são necessariamente protegidas por travas de escrita. Particularmente, a representação de um *thread* distribuído que sofreu divisões é ligeiramente distinta da representação de um *thread* distribuído que não sofreu divisões. A Figura 5-38 (b) ilustra a representação resultante da divisão do *thread* distribuído mostrado na Figura 5-38 (a) após a morte simultânea de l_3 e l_7. As divisões resultantes, mesmo embora representem *threads* distribuídos distintos, são abrigadas sob uma mesma entrada, em uma lista. A lista é implementada de forma que é possível modificá-la sem que leitores sejam afetados (trata-se de uma *copy-on-write list* [161]).

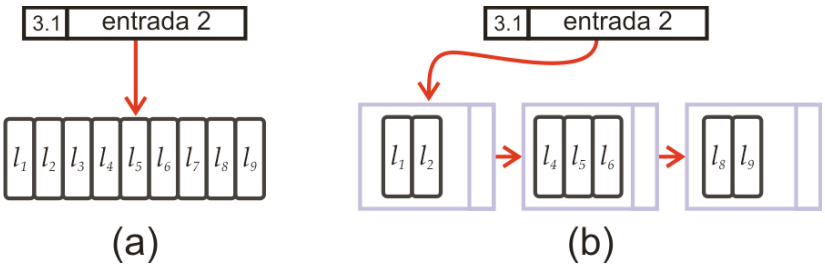

Figura 5-38. Representação de um *thread* distribuído (a) antes e (b) após sofrer divisões.

Por último, resta discutir como controlamos o acesso dos *threads* da interface gráfica às representações dos *threads* distribuídos. Atualmente, cada representação é protegida por uma trava (*lock*) de leitura e escrita que garante as seguintes propriedades:

1. Leitores e escritores acessam a representação de forma mutuamente exclusiva.
2. Múltiplos leitores simultâneos podem acessar uma mesma representação.
3. Escritores e leitores são servidos em ordem de chegada (a trava é justa).

Os *threads* da GUI devem adquirir uma trava de leitura nas representações que desejam inspecionar, de tal forma que, durante cada inspeção, o processamento de eventos DDWP direcionados àquela representação fica bloqueado. Isso não é um problema, no entanto, já que tipicamente as representações só são acessadas pelos *threads* da GUI quando o *thread* distribuído é suspenso – e a ocorrência de eventos DDWP em *threads* distribuídos suspensos deve ser relativamente infreqüente.

Em resumo, nós esperamos que a abordagem seja escalável pois:

1. A maior parte dos eventos pode ser processado de forma concorrente,
2. Espera-se que a contenção por travas, quando existente, seja esporádica, e que a maior parte dos acessos (leitura ou escrita) ocorra sem nenhum tipo de espera.

Nenhuma afirmação categórica a respeito de escalabilidade pode ser feita, no entanto, antes que façamos experimentos apropriados.

5.5.7 Mecanismos de análise automática

Conforme mencionamos na Seção 3.2.7, a análise automática de informações é uma das formas mais eficazes de combate ao efeito labirinto, devendo ser explorada ao máximo, sempre que possível. Visando aderir a essa heurística, nós desenvolvemos algumas formas simples de análise automática de informações que, acreditamos, sejam de valor.

5.5.7.1 Detecção de exceções não-declaradas

O documento que descreve a *Interface Definition Language* (IDL) de CORBA [151] especifica que todas as exceções lançadas por métodos remotos devem ser declaradas como parte de suas respectivas assinaturas. O documento especifica ainda que exceções não-previstas pela IDL devem ser retransmitidas aos clientes sob a forma de uma exceção genérica, a exceção `CORBA::UNKNOWN`. O problema é que essa exceção não transmite ao cliente a natureza do erro ocorrido, além de não transmitir também o contexto do erro no servidor (i.e., a pilha de chamadas). Junte-se isso ao fato de que alguns ORBs sequer apresentam qual a exceção lançada em seus respectivos *logs* e temos uma situação incômoda em mãos, onde o desenvolvedor é obrigado a instrumentar seus objetos remotos manualmente para que todas as exceções sejam exibidas.

A JDI nos permite posicionar *breakpoints* em métodos específicos, que são ativados apenas quando certas exceções são lançadas. Combinando esses *breakpoints* com informações extraídas durante a carga das classes identificadas como possíveis implementações de objetos remotos, desenvolvemos um mecanismo simples que suspende a execução do tratamento de uma requisição quando ela termina com uma exceção não declarada em sua IDL. Quando ativado pelo usuário, esse mecanismo suspende a requisição, dando acesso à pilha completa do *thread* distribuído, incluindo a pilha de chamadas do servidor e de todos os clientes no instante do lançamento da exceção. Nossa hipótese é que o acesso ao tipo da exceção lançada e às informações de contexto globais facilitam o diagnóstico do erro. Além disso, a instrumentação automática agiliza o fluxo de trabalho e reduz as chances de que novos erros sejam introduzidos.

5.5.7.2 Detecção de laços recursivos distribuídos infinitos

Um outro problema também relativamente freqüente – e de detecção mais difícil – diz respeito aos laços recursivos infinitos e distribuídos. ORBs como o JacORB [230] utilizam por padrão, no tratamento de requisições remotas, *pools* de *threads* associados a filas. Essa arquitetura simples de processamento de requisições é mostrada na Figura 5-39. Sob operação normal, as requisições são inicialmente tratadas por um *thread* que fica encarregado por aceitar e enfileirar requisições (o *acceptor thread*, na terminologia do JacORB). Do outro lado da fila há um conjunto (*pool*) de *threads*, que removem seqüencialmente as tarefas enfileiradas pelo *acceptor* e as tratam de forma concorrente. Se porventura o *pool* de *threads* fica saturado (i.e., todos os *threads* estão ocupados), novas requisições serão depositadas na fila pelo *acceptor*, caso haja espaço, ou serão descartadas, caso a fila esteja cheia. Requisições enfileiradas ficam em espera até que haja um *thread* livre no *pool* para tratá-las (até onde pudemos observar, não há coleta de lixo na fila, então mesmo conexões inativas permanecem nela).

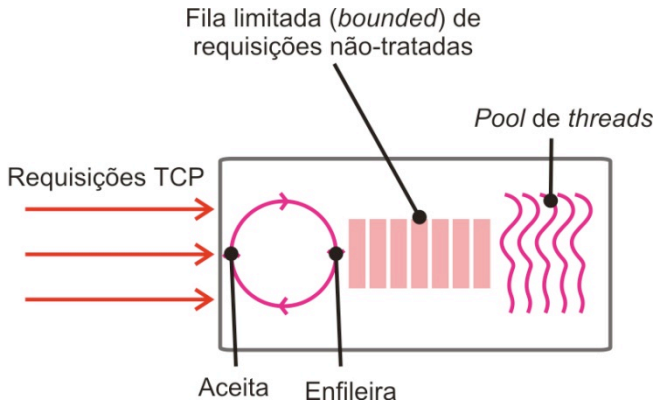

Figura 5-39. Arquitetura de processamento de requisições do JacORB.

Numa situação de laço recursivo infinito distribuído, é comum que o *pool* de *threads* de algum dos ORBs envolvidos se esgote, resultando em um impasse (*deadlock*). O impasse ocorre porque a requisição é posta em espera na fila de requisições do ORB saturado, sendo que todos os *threads* do *pool* estão em uso pela própria requisição. Isso leva a uma situação em que o processamento da requisição depende do término dela própria, ao mesmo tempo em que o término da requisição depende do seu processamento. Os sintomas desse tipo de situação podem ser bastante variados, podendo levar o desenvolvedor a considerar uma quantia razoável de hipóteses antes de encontrar a causa do erro. Para auxiliar na detecção automática deste problema em particular, o depurador inclui um monitor de fotografias. Se a quantia de *threads* locais que participam em uma única fotografia de um *thread* distribuído ultrapassa um certo número, o depurador emite um aviso ao usuário e suspende a requisição suspeita.

5.5.7.3 Detecção de *deadlocks* distribuídos

A última forma de análise automática disponibilizada pelo depurador é uma modalidade bastante simples de detecção de *deadlocks* distribuídos. Conforme mencionamos na Seção 4.3, os *deadlocks* distribuídos podem ocorrer, em um sistema de objetos distribuídos típico, tanto entre *threads* distribuídos distintos quanto dentro um mesmo *thread* distribuído, em decorrência da miopia das travas reentrantes às abstrações do *middleware*.

O depurador simbólico da JPDA nos permite determinar tanto o conjunto de travas adquiridas por cada *thread* local quanto a identidade das travas por qual cada *thread* local espera, se houver alguma. Combinando essas informações às informações providas pelas fotografias mantidas pelo agente central, torna-se possível determinar os conjuntos de travas adquiridas e a identidade das travas pelas quais esperam cada *thread* distribuído. A partir daí, o desenvolvimento de um mecanismo primitivo para a verificação de *deadlocks* distribuídos é simples. O mecanismo pode ser descrito da seguinte forma:

1. O usuário, suspeitando de um *deadlock* distribuído, solicita um passo de verificação ao depurador.
2. O agente central suspende todos os *threads* distribuídos sob suspeita (indicados pelo usuário) e determina os conjuntos de travas associados a cada um, bem como a identidade das travas pelas quais cada *thread* distribuído espera (se houver alguma).
3. O agente central constrói um grafo dirigido que tem os *threads* distribuídos suspensos como vértices. Há uma aresta (T_1, T_2) entre os *threads* distribuídos T_1 e T_2 se e somente se T_2 detém uma trava pela qual T_1 espera.
4. O agente central varre o grafo em busca de componentes fortemente conexos. Se não houver nenhum, então não há um *deadlock* distribuído.
5. O agente central executa uma busca em profundidade em cada componente conexo, devolvendo ao usuário o primeiro ciclo encontrado em cada um deles.

Note que nosso mecanismo de detecção é bastante mais simples que o algoritmo clássico de Chandy e Lamport [26]. Além de menos geral, nosso algoritmo exige a suspensão de todos os participantes envolvidos na inspeção. Sua implementação, por outro lado, depende apenas de facilidades já providas pelos agentes locais. Nosso mecanismo segue em conformidade com a heurística mencionada no início da seção: técnicas de análise automática devem ser exploradas ao máximo. O papel deste mecanismo é, portanto, o de facilitar a navegação da execução com uma forma de análise automática de informações, e não o de prover o melhor algoritmo possível para detecção de propriedades estáveis [26, 121, 185, 221].

5.6 O controlador de processos remotos

O último componente importante de nossa ferramenta não se encontra ligado diretamente à depuração, mas a uma atividade relacionada e de bastante relevância, especialmente na depuração cíclica: a montagem de cenários de depuração (e, conforme veremos mais adiante, teste). Vamos, inicialmente, definir o que é um "cenário de depuração" em nosso contexto. Como em outras ocasiões ao longo deste texto, nossa definição será bastante informal, mas suficiente para nossos propósitos:

Definição 5-2: Um cenário de depuração é um conjunto de estados específicos do sistema distribuído, a partir dos quais existe alguma chance do comportamento esperado se desenvolver.

Montar um cenário de depuração, portanto, equivale a induzir um dentre um conjunto de estados específicos no sistema distribuído. Particularmente, estamos interessados em excluir do espaço de estados do sistema distribuído aqueles estados que temos certeza que não levarão ao comportamento desejado. De fato, excluir da execução do sistema todos os estados que não levam ao comportamento desejado é um problema bastante semelhante ao problema da reprodução de execuções, discutido extensamente na Seção 3.2. Nós vamos nos contentar, portanto, com simplesmente reduzir o espaço de estados indesejáveis.

Além de auxiliar na montagem de cenários de depuração, o controlador de processos remotos também preocupa-se com auxiliar o usuário na interação com os processos que compõem o sistema distribuído permitindo, por exemplo, a interação com a entrada padrão a leitura da saída padrão de processos remotos. Além disso, procuramos viabilizar o uso de sessões remotas do X11 [194], quando disponíveis, para que o usuário possa interagir também com as interfaces gráficas de processos remotos. Por último, o controlador de processos ajuda o agente central a manter um controle mais preciso a respeito de quais processos ainda estão vivos, por meio de um mecanismo simples e usual, baseado em *heartbeats*, aliado ao monitoramento ativo dos processos em execução.

Em resumo, são responsabilidades do controlador de processos remotos:

1. Ajudar o usuário a lançar, com pouco esforço, um número potencialmente grande de processos, distribuídos por múltiplos nodos de processamento.
2. Sincronizar o estado dos processos distribuídos lançados, visando induzir instâncias "viáveis" do sistema resultante (reduzir o número de estados indesejáveis).
3. Permitir a interação com processos remotos, provendo uma infra-estrutura mínima (interação com a entrada e saída padrão) e alavancando software externo (e.g., sessões remotas do X11) para funcionalidades adicionais (interação com interfaces gráficas).

4. Auxiliar na detecção mais rápida da morte dos processos que compõem o sistema distribuído.

Vamos iniciar esta discussão apresentando nossas soluções para os problemas 1, 3 e 4. Para ajudar o usuário a lançar um número potencialmente grande de processos, desenvolvemos um servidor simples, que deve ser iniciado em cada nodo. Esse servidor expõe uma interface para o gerenciamento de processos, que será acessada pelo agente central em nome do usuário. O servidor resolve, no entanto, apenas parte do problema do lançamento remoto de processos, já que o próprio servidor precisa ser iniciado de alguma forma. A estratégia de lançamento do servidor é plugável. Atualmente, a única implementação disponível é baseada no protocolo *Secure Shell* (SSH [233]) e exige que todos os nodos de processamento do sistema contem com servidores SSH funcionais (algo que, em sistemas baseados no UNIX, é uma hipótese bastante razoável). Um ciclo de interação hipotético entre agente central e nodo remoto é mostrado na Figura 5-40.

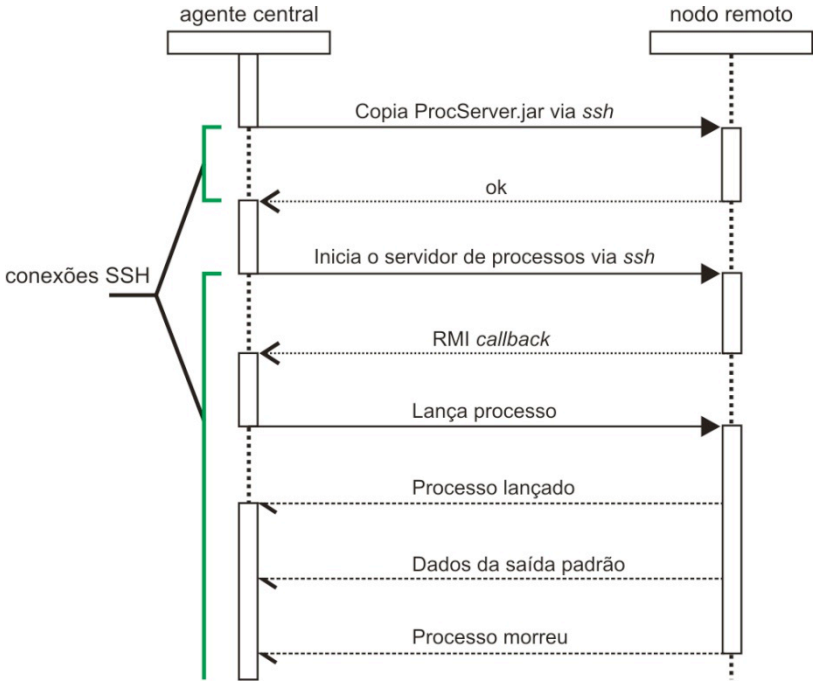

Figura 5-40. Interação entre agente central e nodo remoto do ponto de vista do controlador de processos.

Inicialmente, o pacote contendo o código do servidor é copiado para o nodo remoto, via SSH. Se a cópia for concluída com sucesso, uma segunda conexão SSH é criada, desta vez para o lançamento do servidor de processos. Essa conexão é mantida até o final da sessão de depuração e será utilizada para o tunelamento de pacotes do X11. Ao ser lançado, o servidor de processos contata um objeto RMI, publicado pelo agente central em um local conhecido, indicando que se encontra pronto para receber requisições. A partir daí, o agente central pode utilizar o servidor ativo

para lançar processos e receber notificações – essencialmente notificações de morte e de atividade na saída padrão – a respeito desses processos. Além disso, o servidor de processos envia, periodicamente, pacotes UDP para o agente central afim de sinalizar que o nodo de processamento ainda se encontra vivo.

O agente central, por sua vez, mantém um contador temporal decrescente por nodo, que é reiniciado cada vez que um pacote UDP ou requisição DDWP é recebida daquele nodo. Caso o contador chegue a zero, o agente central tenta determinar se o nodo de processamento ainda encontra-se ativo, enviando um pacote ICMP. Se não houver resposta, o agente central determina que o nodo e todos os processos nele morreram, executando a divisão dos *threads* distribuídos correspondentes. Se houver resposta, no entanto, isso pode querer dizer que o servidor remoto de processos morreu. O agente central tenta então reiniciá-lo por meio de uma nova conexão SSH, desistindo caso isso também resulte em falha.

Essa infra-estrutura auto-gerenciada de servidores de controle de processos facilita substancialmente o lançamento do sistema distribuído, efetivamente transformando a operação – após prévia configuração dos parâmetros de cada processo – em uma operação de um único clique. A interface de configuração de parâmetros de lançamento de processos remotos é mostrada na Figura 5-42. Trata-se de uma interface simples, que dá acesso direto ao contêiner de injeção de dependências [66], de nossa autoria, no qual é baseada toda a infra-estrutura de configuração do agente central.

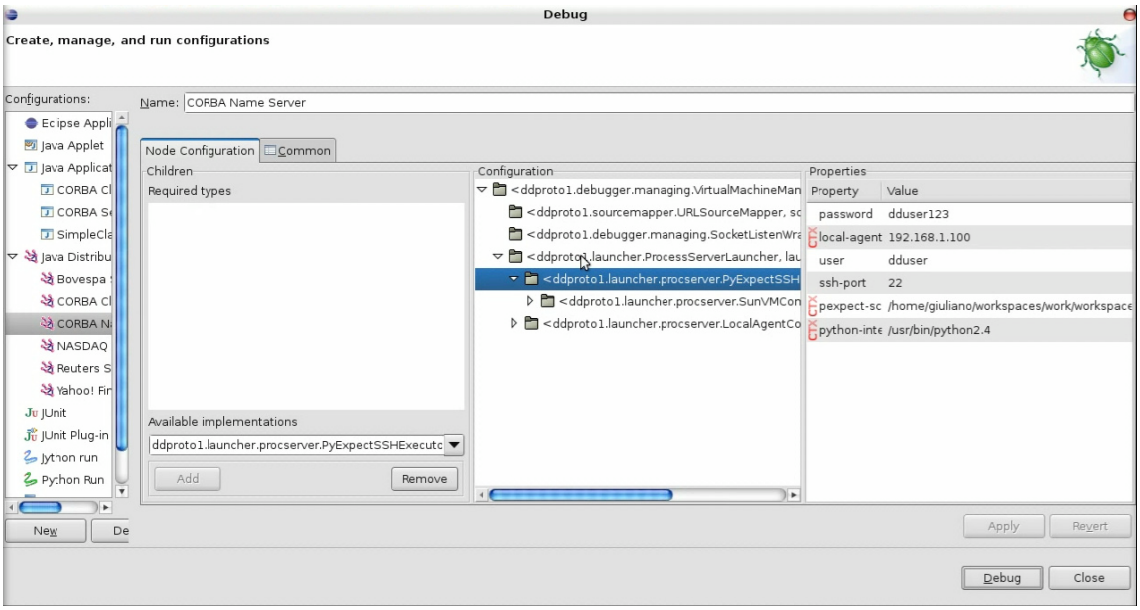

Figura 5-41. Interface de configuração para o lançamento de processos remotos.

A destruição do sistema distribuído é também simplificada pela infra-estrutura – basta clicar no botão de parada (em destaque na Figura 5-42) e o agente central sinaliza a todos os servidores de controle processos para que destruam os processos sob sua guarda. Essas facilidades para a rápida criação e destruição de processos em máquinas remotas permitem que o usuário obtenha maior agilidade e ciclos mais rápidos durante o processo de depuração. Aumentar a velocidade dos ciclos é importante já que, conforme mencionamos na 3.2.1, quanto maior o ciclo, maior a interrupção entre sessões de exploração da execução e maior o prejuízo à compreensão do comportamento do sistema.

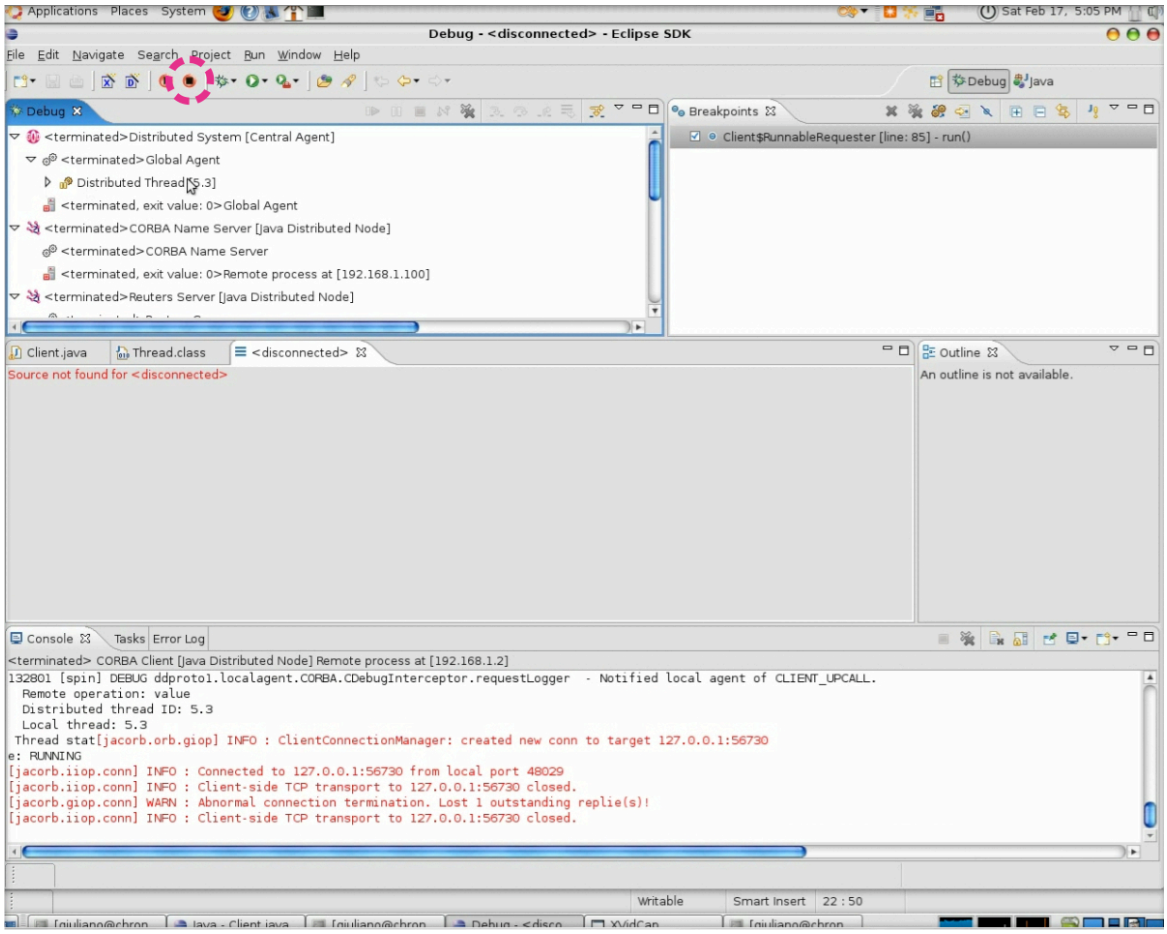

Figura 5-42. Destruição de um sistema distribuído após clique no botão de parada, em destaque.

Além do controle de processos ativos, a infra-estrutura de controle de processos redireciona o conteúdo das saídas padrão para o agente central, permitindo que o usuário inspecione de forma simples e rápida a saída dos processos em execução. A Figura 5-42 mostra, na aba *Console*, as saídas padrão (*stdout* em preto e *stderr* em vermelho) de um dos processos remotos. Pelo mesmo console, o usuário pode interagir com a entrada padrão de processos remotos.

Isso encerra a nossa discussão sobre as soluções para o tratamento dos problemas 1, 3 e 4. Resta ainda discutirmos o problema 2 – a sincronização dos processos do sistema distribuído. Para entendermos em termos mais concretos o problema que desejamos resolver, vamos tomar como exemplo um sistema distribuído simples, composto por apenas um cliente e um servidor. Suponha agora que o usuário deseje depurar o tratamento de uma requisição pelo servidor. Se permitirmos que os nodos do sistema distribuído sejam lançados em qualquer ordem, é bem possível que o cliente – tipicamente um programa mais leve que o servidor – atinja o ponto de envio de sua requisição antes que o servidor tenha a oportunidade de inicializar de forma adequada. O resultado será uma falha de comunicação, que essencialmente impede que o cenário desejado – servidor trata requisição de cliente – se desenvolva.

Há uma série de possíveis abordagens para o problema. A primeira delas – e muito pouco eficaz – é tentar novamente. Nessa abordagem, o usuário lança novamente ambos os processos até que, em alguma das execuções, por pura sorte, o cliente envia a requisição após a inicialização do servidor. O problema óbvio com essa abordagem é que não há garantias sobre a freqüência com a qual o cenário desejado se desenrola. De fato, não existem sequer garantias de que a execução vá se desenrolar a contento em algum momento.

A segunda opção, mais eficaz, consiste na sincronização manual entre cliente e servidor. Por meio de indicadores observáveis do estado do servidor (como por exemplo, mensagens impressas na saída padrão), o usuário determina um instante seguro para o lançamento do cliente. A desvantagem óbvia é que se trata de uma abordagem manual e, como toda abordagem manual, pode se tornar bastante cansativa e inconveniente, especialmente em casos em que existem coleções maiores de processos e muitas dependências. A terceira opção consiste no estabelecimento de mecanismos de sincronização automáticos *ad hoc* entre cliente e servidor. Novamente, a vantagem é que a abordagem pode ser bastante eficaz. A desvantagem é que o ônus sobre o projeto e implementação de um mecanismo de sincronização potencialmente complexo fica a cargo do desenvolvedor.

A quarta e última opção consiste na sincronização dos processos por meio de um mecanismo de instrumentação automático, que reproduz o cenário desejado a partir de um conjunto de parâmetros dados pelo usuário. Essa descrição pode soar complexa, mas note que temos à nossa disposição uma API de metaprogramação genérica, com acesso quase ilimitado ao estado dos processos locais. Baseados nessa observação, desenvolvemos uma linguagem declarativa simples, cuja gramática é mostrada na Figura 5-43, para a descrição de restrições de lançamento. A linguagem permite, essencialmente, que sejam especificadas pré-condições para o lançamento de um processo. Essas pré-condições são predicados locais simples, dados sobre o estado de outros processos. Os

predicados podem envolver tanto o estado de *threads* quanto valores de combinações de variáveis. Como exemplo, a seguinte declaração:

```
when <CORBA Name Server>
   reaches org.jacorb.ORB:1278
   launch <Server 1, Server 2>
```

Especifica que os nodos "Server 1" e "Server 2" devem ser lançados apenas quando algum *thread* no nodo "CORBA Name Server" executar os *bytecodes* que correspondem à linha 1278 do arquivo de código fonte vinculado à classe `org.jacorb.ORB`. Consideremos agora:

```
when <SomeServer>
      reaches
            module1.Type.status="ready_to_go" &
            (module1.Type.state = 1 |
            module1.Type1.state = 2)
      launch <Client>
```

Essa declaração especifica que o cliente "Client" deve ser lançado apenas quando o seguinte predicado for satisfeito no nodo "SomeServer":

1. O campo "status" de alguma instância de `module1.Type` assumir o valor "ready_to_go", e:
    a. O campo "state" de alguma instância de `Type` assumir o valor 1, ou,
    b. O campo "state" de alguma instância de `Type1` assumir o valor 2.

A implementação atual da detecção desses predicados, no cliente de depuração Java, utiliza *access watchpoints* [207] para a detecção de mudanças no estado das variáveis envolvidas nas expressões de restrição. Note que os predicados locais são instáveis [70, 221] e detectados remotamente. Além disso, os eventos providos pela JDI não possuem nenhum tipo de *timestamping* lógico, o que significa que não podemos determinar se dois eventos são concorrentes ou causalmente relacionados. Não podemos sequer supor que os eventos serão entregues em uma ordem que respeita a ordem local.

A implicação dessa falta de informações é de que as garantias oferecidas a respeito do predicado detectado são muito fracas – não podemos supor sequer que o predicado detectado é calculado em cima de um estado possível (análises das condições necessárias para a detecção remota de predicados instáveis podem ser encontradas nos trabalhos de Cooper e Marzullo [40], Sen [187], Tomlinson [221] e Mittal [137], entre outros). O algoritmo para a detecção dos predicados é bastante simples – os eventos são processados em ordem de recebimento e, a cada evento processado, o predicado é reavaliado. Quando (e se) o predicado se torna verdadeiro, a pré-

condição de lançamento é satisfeita. As fracas garantias oferecidas pelo algoritmo de detecção implicam que os predicados devem ser escolhidos com cuidado para que façam algum sentido.

$$
\begin{array}{rcl}
\langle \text{decl} \rangle & \longrightarrow & \texttt{when}\ \langle \text{machine} \rangle\ \texttt{reaches} \\
 & & (\langle \text{locationExp} \rangle \mid \langle \text{compoundStateExp} \rangle) \\
 & & \texttt{launch}\ \langle \text{machineList} \rangle \\
\langle \text{locationExp} \rangle & \longrightarrow & \langle \text{type} \rangle\ \texttt{:}\ \langle \text{num} \rangle \\
\langle \text{compoundStateExp} \rangle & \longrightarrow & (\langle \text{stateExp} \rangle\ \texttt{\&}\ \langle \text{compoundStateExp} \rangle) \\
 & & \mid\ (\langle \text{stateExp} \rangle\ \texttt{|}\ \langle \text{compoundStateExp} \rangle) \\
 & & \mid \langle \text{stateExp} \rangle \\
\langle \text{stateExp} \rangle & \longrightarrow & \langle \text{terminalExp} \rangle \\
 & & \mid\ \texttt{(}\ \langle \text{compoundStateExp} \rangle\ \texttt{)} \\
\langle \text{terminalExp} \rangle & \longrightarrow & \langle \text{typeOrField} \rangle \\
 & & (\texttt{=} \mid \texttt{!=} \mid \texttt{>} \mid \texttt{<}) \\
 & & \langle \text{primitive} \rangle \\
\langle \text{machineList} \rangle & \longrightarrow & \langle \text{machine} \rangle\ (\texttt{,}\ \langle \text{machine} \rangle)^{*} \\
\langle \text{machine} \rangle & \longrightarrow & \texttt{<}\ \langle \text{string} \rangle\ \texttt{>} \\
\langle \text{type} \rangle & \longrightarrow & \langle \text{javaIdent} \rangle \mid (\langle \text{typeOrField} \rangle)^{*} \\
\langle \text{typeOrField} \rangle & \longrightarrow & \langle \text{javaIdent} \rangle\ \texttt{.}\ \langle \text{javaIdent} \rangle \\
\langle \text{primitive} \rangle & \longrightarrow & \texttt{'}\ \langle \text{string} \rangle\ \texttt{'} \mid \langle \text{num} \rangle \\
\langle \text{num} \rangle & \longrightarrow & [0-9]\ + \\
\langle \text{string} \rangle & \longrightarrow & [:\ \mathit{validstringcharacter}\ :]\ + \\
\langle \text{javaIdent} \rangle & \longrightarrow & [:\ \mathit{jletter}\ :]\ \ [:\ \mathit{jletterdigit}\ :]^{*}
\end{array}
$$

Identificadores especiais:
[:validstringcharacter:] - qualquer caractere válido dentro de uma string Java
[:jletter:] - quaisquer caracteres não-numéricos válidos em identificadores Java
[:jletterdigit:] - quaisquer caracteres, numéricos ou não, válidos me identificadores Java

Figura 5-43. Gramática da linguagem declarativa de restrição de lançamento.

Uma forma de contornar o problema seria migrar a avaliação dos predicados para o nível base. Poderíamos instrumentar os pontos de acesso aos campos envolvidos nos predicados locais, de tal forma que o predicado fosse reavaliado a cada acesso de escrita. Isso exigiria um trabalho de instrumentação substancial, no entanto, além de potencialmente produzir um *overhead* considerável (tanto a opção de manter os campos instrumentados após a detecção do predicado quanto a de "desinstrumentá-los" dinamicamente, por meio de JVMTI [205], nos parecem ruins por esse aspecto). O argumento do *overhead* é, no entanto, bem menos importante do que o da complexidade da implementação. As dependências entre processos, expressas pelas declarações na linguagem de restrições, serão traduzidas em arestas em um grafo dirigido e acíclico (ciclos serão tratados como erros), que tem os processos envolvidos nas expressões como vértices. A operação de lançamento do sistema distribuído inicia, ao mesmo tempo, todos os processos com pré-condições trivialmente satisfeitas; isto é, todos os processos cujo *fan-in* zero. A partir daí, o restante dos processos são

lançados conforme suas dependências forem satisfeitas. Alguns predicados podem ser compostos indiretamente, como no exemplo a seguir:

```
when <Server 1>
   reaches server.Main:20
   launch <Server 2>
when <Server 3>
   reaches server.internal.ObjectImpl:543
   launch <Server 2>
```

Nessa especificação, o lançamento de "Server 2" depende de dois predicados. Pela própria semântica da linguagem de restrição – as pré-condições especificam estados que devem ser atingidos por um processo para que outro processo possa operar corretamente – a composição de predicados será sempre realizada com o operador booleano de disjunção ($\wedge$). O grafo acíclico e dirigido que resulta do conjunto de todas as expressões de restrição apresentadas nesta seção é mostrado na Figura 5-44 (predicados omitidos):

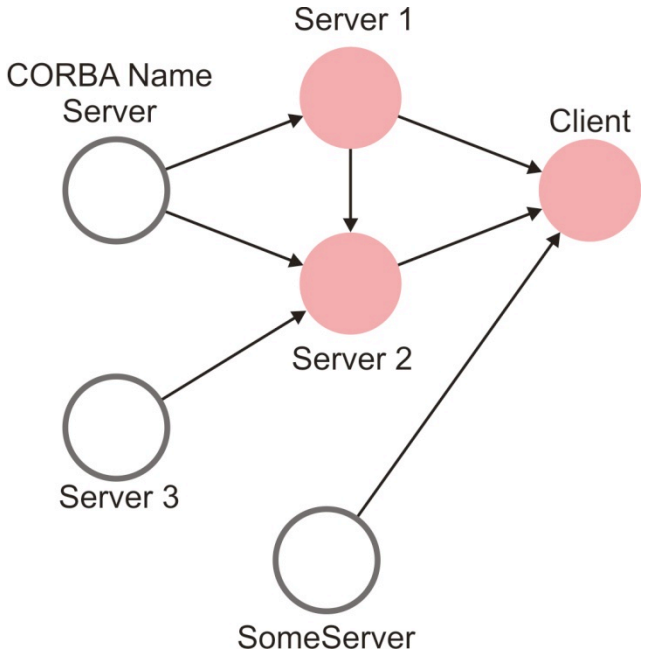

Figura 5-44. Grafo de dependências de lançamento.

A operação de lançamento inicia os três processos com *fan-in* zero: "CORBA Name Server", "Server 3" e "SomeServer". A partir daí, o restante dos processos serão lançados conforme os predicados dos quais dependem forem satisfeitos. Os processos com maior quantia de pré-condições são "Client" e "Server 2", com três cada.

É importante notar que as facilidades providas pela linguagem de restrição de lançamento são úteis não apenas para depuração de sistemas distribuídos, mas também para o teste. A linguagem dá

ao desenvolvedor a garantia mínima de que a instância do sistema distribuído produzida durante a seqüência de lançamento será viável, e que o comportamento observado advém de nodos que tiveram a oportunidade de inicializar corretamente. A partir daí, o desenvolvedor pode utilizar as próprias APIs do depurador para inspecionar o estado dos nodos em pontos específicos da execução, bem como para avançá-la de forma controlada. De fato, alguns dos testes automatizados de integração escritos para o próprio depurador foram baseados justamente nessa combinação – *scripts* de lançamento aliados à API do depurador para avanço controlado da execução e inspeção de estados.

5.7 Limitações e trabalhos futuros

Tanto a ferramenta quanto a técnica de depuração propostas neste texto procuram apoiar as atividades de depuração típicas de um desenvolvedor de sistemas de objetos distribuídos, enquanto mantendo em mente duas qualidades fundamentais: portabilidade e extensibilidade. Esta seção discute as limitações da abordagem desenvolvida até então, bem como algumas das possíveis soluções a essas limitações (trabalhos futuros). Nossa exposição será dividida em duas seções. A primeira seção (Seção 5.7.1.1) trata exclusivamente dos problemas clássicos (quais dos problemas clássicos da programação concorrente e distribuída nossa ferramenta ajuda, e não ajuda, a resolver). A segunda seção (Seção 5.7.1.2) sumariza as limitações técnicas da implementação atual.

5.7.1.1 Limitações com relação aos problemas clássicos

No decorrer dos Capítulos 2 e 3, nós discutimos uma série de problemas, técnicas e ferramentas voltadas à depuração de sistemas concorrentes (paralelos, distribuídos e pseudo-paralelos). As ferramentas consideradas procuram ajudar o desenvolvedor a lidar com um ou mais dos problemas tidos como clássicos na literatura sobre depuração concorrente e distribuída, a saber: os problemas de observabilidade, as execuções não-determinísticas (e o efeito do observador) e o efeito labirinto. Do ponto de vista desses problemas, nossa ferramenta:

Trata parte do problema de observabilidade. As fotografias dos *threads* distribuídos revelam formas limitadas de cadeias de causa e efeito. O usuário se torna capaz de observar a influência de clientes em servidores ao longo de cadeias que cruzam múltiplos nodos de processamento. A técnica não captura, no entanto, a relação de causalidade de Fidge [63, 64] e Mattern [127, 185] e não provê sequer as garantidas dos relógios de Lamport [110]. Embora a inclusão de um sistema de relógios vetoriais na implementação seja uma tarefa quase trivial, optamos por não fazê-lo, dado que são desnecessários no contexto atual. A inclusão de novos mecanismos de visualização pode, entretanto, exigir o uso de relógios lógicos, mas isso integra o quadro de trabalhos futuros. Entre

outras coisas, isso significa que nossa ferramenta não é, atualmente, capaz de capturar causalidade, nem de produzir análises que dependam dessas informações.

Reduz o efeito labirinto. A representação do *thread* distribuído correlaciona automaticamente os *threads* locais que pertencem a cada requisição. A correlação manual desses *threads* pode exigir uma grande quantidade de esforço, já que é necessário suspender e inspecionar uma quantia potencialmente grande de *threads* locais. Além disso, nem sempre é possível determinar, pela inspeção do contexto de um *thread* local, a qual requisição esse *thread* pertence (se é que pertence a alguma requisição). Os modos de execução passo-a-passo permitem que o desenvolvedor mantenha seu foco no fluxo de controle de sua própria aplicação, evitando o contato com o código e, principalmente, com os objetos de tempo de execução do *middleware*. Essa representação, portanto, apresenta algumas das características desejáveis discutidas na Seção 3.2.7, na medida em que apóia o foco seletivo de atenção e permite a filtragem de informações irrelevantes.

O fato que os *threads* originais encontram-se facilmente disponíveis caso o usuário deseje inspecioná-los implica que há um mecanismo, ainda que primitivo, de detalhamento sob demanda (*drill-down*) das informações abstratas. Todas essas características contribuem com uma redução no efeito labirinto, na medida em que tiram do caminho informações possivelmente irrelevantes, mas permitem o acesso a elas caso necessário. Por último, podemos dizer que nossas metáforas são:

1. **Consistentes:** As visualizações de *threads* representam entidades uniformes (*threads* distribuídos são, em nosso contexto, apenas *threads* locais capazes de extravasar os limites dos nodos).
2. **Semanticamente ricas:** É possível representar, por meio da metáfora, todos os estados do sistema distribuído. A metáfora é rica o suficiente para que uma grande variedade de erros sob consideração possam ser detectados.

A ferramenta provê ainda algumas formas de análise automática de informações que tiram do desenvolvedor, sob certas circunstâncias, o fardo da análise manual. Isso agiliza a verificação e o descarte de hipóteses e também contribui com uma redução no efeito labirinto.

A metáfora do *thread* distribuído é, no entanto, ainda uma metáfora de baixo nível. O sistema distribuído passa a ser representado como um grande sistema *multithreaded*: por um lado, isso é melhor do que uma coleção de *threads* locais disjuntos; por outro, não podemos afirmar que um sistema *multithreaded* – particularmente um dotado de um grande número de *threads* – apresenta um comportamento dinâmico simples. De fato, conforme vimos na Seção 2.6, é possível argumentar que um sistema *multithreaded* apresenta um comportamento dinâmico mais difícil de examinar do que o de um sistema distribuído onde todos os nodos se comunicam por passagem de mensagens. Sendo assim, há ainda muitas oportunidades para que o usuário se perca em meio a um

labirinto de *threads* distribuídos. De qualquer forma, a metáfora torna a execução mais simples de observar do que sem ela.

Por último, há a questão de se de fato a ferramenta ajuda na identificação das causas de comportamentos errôneos. Para que essa pergunta possa de fato ser respondida, são precisos experimentos. Nossa hipótese é de que a ferramenta de fato ajuda, por todos os argumentos apresentados nesta seção e na Seção 4.3.4, além dos argumentos em favor dos depuradores simbólicos (que se aplicam à nossa ferramenta). Alguns trabalhos [39] apontam que a manipulação *on-line* e interativa de sistemas de software pode não prover tantas vantagens sobre a manipulação *post-mortem*. Isso não implica, no entanto, que a manipulação interativa não apresente algum tipo de vantagem, nem que o uso de uma ferramenta especializada – como o nosso depurador distribuído – não apresente vantagens sobre o uso de ferramentas não-especializadas.

Dito isso, o mecanismo de visualização provido pela ferramenta é ainda de nível muito baixo para a identificação de comportamentos errôneos mais complexos, especialmente no caso de erros que se manifestam após execuções muito longas. Formas de visualização de nível mais alto e técnicas de análise automática (Seção 3.2.7) são duas boas linhas de desenvolvimento para o tratamento das deficiências da ferramenta nessas áreas e ficam como trabalho futuro.

Não oferece soluções para o problema das execuções não-determinísticas e nem para o efeito do observador. Embora uma parte substancial do nosso estudo de trabalhos relacionados tenha sido voltada às técnicas de reprodução de execuções não-determinísticas, nossas tentativas de abordagem do problema foram infrutíferas, produzindo apenas resultados preliminares, como a modificação da técnica de Georges [71] discutida na Seção 4.3.5. A reprodução do cenário de falha é uma parte fundamental do ciclo de depuração, sem a qual a eficácia de qualquer técnica de depuração que não seja completamente automática (ou seja, a grande maioria) fica seriamente comprometida.

O uso difundido dos modelos baseados em *threads* e memória compartilhada, aliados à natureza específica e não-portável das soluções existentes, no entanto, torna o desenvolvimento de soluções práticas muito difícil. As constatações da Seção 3.3 nos levam a crer que o desenvolvimento de soluções abrangentes para o problema da reprodução de execuções não-determinísticas não deve ocorrer num futuro próximo. Juntando isso ao fato, mencionado inúmeras vezes neste texto, de que o modelo de *threads* e memória compartilhada é inerentemente difícil de programar, concluímos que a melhor forma de abordar o problema no curto e médio prazo é pela mudança do modelo de programação nos novos sistemas concorrentes.

Acreditamos, por experiência própria, que as linguagens que disponibilizam modelos de programação baseados em atores, como Erlang [11] e Scala [77, 78], sejam excelentes alternativas,

na medida em que são mais fáceis de programar e produzem sistemas mais simples de depurar – lembre-se da Seção 2.6, onde mostramos que tanto a reprodução quanto a análise da execução de sistemas baseados em passagem de mensagens é mais simples do que quando temos sistemas baseados em *threads* e memória compartilhada.

5.7.1.2 Limitações técnicas

As principais limitações técnicas da implementação atual são as seguintes:

Tratamento de mensagens assíncronas. O tratamento das mensagens assíncronas (*one-way*) apresenta limitações no modo de execução passo-a-passo. Em particular, um *step into* em uma chamada *one-way* não faz com que a chamada seja suspensa do lado do servidor. A Figura 5-45 ilustra o cenário problemático: quando um cliente solicita um *step into* em um *proxy* para um objeto remoto em uma chamada assíncrona, nós deveríamos, idealmente: (1) suspender o *thread* do lado do cliente no ponto do código que procede a chamada ao proxy (suspensão (1) na Figura 5-45 (a)) e (2) suspender o *thread* de lado do servidor no instante em que ele penetra no objeto remoto (suspensão (2) na Figura 5-45 (b)).

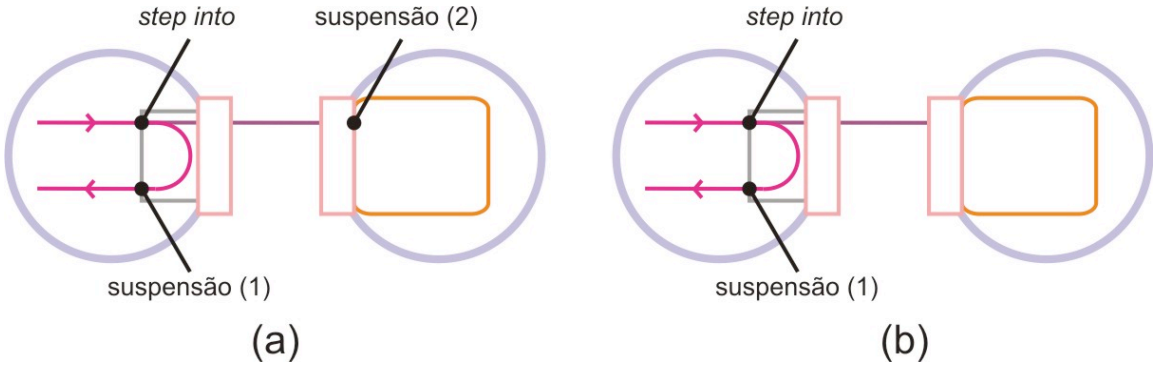

Figura 5-45. Situação ideal

A implementação atual, no entanto, concentra o conhecimento a respeito do modo de execução passo-a-passo do *thread* distribuído inteiramente no agente central, levando à situação mostrada na Figura 5-45 (b) - o *thread* do lado do cliente é suspenso no ponto correto, mas o *thread* do lado do servidor não é suspenso.

Cliente de depuração Java. Nosso cliente de depuração Java, embora funcional, apresenta uma série de limitações quando comparado, por exemplo, ao depurador incluso com o JDT [57]. Os mecanismos de avaliação de expressões e inspeção de estados são primitivos e não há suporte a *hotswapping* de classes. O cliente é estável, no entanto, e cumpre o seu propósito: o de ser um cliente de depuração simples e fácil de entender e manipular. O depurador do JDT é bastante complexo. Lidar com essa complexidade ao mesmo tempo em que lidávamos com as questões que envolveram o projeto e a construção do primeiro protótipo do depurador distribuído seria, ao menos

em um primeiro instante, contraproducente. A integração do cliente de depuração do JDT, bem como de outros clientes de depuração, fica como trabalho futuro.

Funcionalidades parcialmente integradas. Este projeto foi desenvolvido em duas fases. A primeira fase consistiu na construção de um protótipo acoplado à JDI, que contava com uma interface com o usuário simples e baseada em texto (essa fase durou até o meio de 2006). A segunda fase consistiu na reescrita do núcleo e na criação das camadas de integração, que desacoplaram tanto o código da interface gráfica, que passou a ser provido pelo Eclipse, quanto as porções genéricas do depurador, das particularidades da JDI. Não houve tempo, no entanto, para que integrássemos todas as funcionalidades na interface gráfica provida pelo Eclipse. Particularmente, não há como acessar o detector de *deadlocks* distribuídos ou o interpretador da linguagem de restrições de lançamento. A integração dessas duas funcionalidades fica como trabalho futuro.

Falhas. Na Seção 5.3.4, nós explicamos como o depurador lida com *threads* distribuídos na presença de falhas. De fato, falhas no sistema distribuído subjacente – sejam elas falhas de nodos ou falhas de comunicação entre nodos – são bem acomodadas tanto conceitualmente quanto na implementação. A questão surge, no entanto, quando há uma falha na comunicação entre agente local e agente central. Não há provisões, na implementação atual, para o tratamento de falhas desse tipo – (1) as informações são simplesmente perdidas, defasando o estado do agente central de forma irreparável. Também não há provisões (2) para o tratamento de situações em que um nodo aparenta ter falhado, mas volta à atividade após a decisão do agente central de tratar como falha a falta de resposta por tempo prolongado.

Suspeitamos que o problema (1) seja comum entre técnicas interativas e limitadas (Seção 3.2.6), já que a exigência de sincronizações implica que falhas nessas sincronizações resultam na violação das garantias exigidas pela técnica. A introdução de tolerância a falhas possivelmente demanda o desenvolvimento de uma técnica híbrida, capaz de operar de forma assíncrona sob situações de indisponibilidade dos nodos remotos. A solução desse problema, ou o estabelecimento de um compromisso mais razoável entre acurácia, interatividade e tolerância a falhas, fica como trabalho futuro. Por hora, o fato de que não somos capazes de lidar com esse tipo de problema reforça a idéia de que nossa ferramenta é adequada para ambientes controlados e sistemas distribuídos acoplados.

Escalabilidade. Embora tenhamos afirmado que o mecanismo de rastreio deve escalar a contento, não produzimos nenhum tipo de experimento que possa validar essa informação. Particularmente, é possível que esses experimentos revelem que o mecanismo de captura síncrono produza muito pouco *overhead*, prestando-se à captura de informações para análise automática em redes de baixa latência. A análise de desempenho do servidor DDWP sob situações de carga elevada e latência variada fica como trabalho futuro.

5.8 Sumário

Neste capítulo, apresentamos uma ferramenta de depuração simbólica extensível para sistemas de objetos distribuídos – o *Global On-line Debugger* (G.O.D.). Essa ferramenta se apóia em uma técnica genérica para identificação e rastreio de *threads* distribuídos, aplicável a uma ampla gama de sistemas de middleware. A técnica de rastreio depende da identificação dos *threads* locais no nível base. Essa identificação se dá mediante a atribuição de identificadores globalmente únicos a cada *thread* local, que são propagados com chamadas remotas. Essas informações, localmente acessíveis, são utilizadas na produção dos eventos que permitem ao agente central detectar as mudanças relevantes nas fotografias dos *threads* distribuídos. A implementação confiável da operação de suspensão em *threads* distribuídos exige que esses eventos comunicados de forma síncrona.

Os agentes locais Java/CORBA implementam o mecanismo de rastreio por meio da inserção de interceptadores de instrumentação nos *proxies* e implementações de objetos *remotos* do sistema de objetos distribuídos. Esses interceptadores também participam na detecção de mensagens perdidas e nos protocolos de coordenação entre o mecanismo de rastreio e os modos de execução passo-a-passo virtuais.

O núcleo do agente central apresenta o sistema distribuído, às camadas superiores, como um ambiente de execução abstrato: um processo virtual composto por uma coleção de *threads* distribuídos. As entidades desse ambiente de execução virtual são reificadas como uma implementação da API de metaprogramação genérica do Eclipse. O núcleo do agente central define ainda um conjunto de interfaces estendidas, que devem ser implementadas pelos clientes de depuração específicos de cada linguagem para que, entre outras razões, os conteúdos de seus *threads* locais possam ser integrados aos *threads* distribuídos.

A incorporação de novas linguagens ao depurador distribuído depende da construção/adaptação de clientes de depuração compatíveis com o Eclipse às interfaces estendidas requeridas pelo núcleo do agente central, bem como da implementação do maquinário de instrumentação e comunicação dos agentes locais. A ampla disponibilidade de clientes de depuração para o Eclipse implica que o esforço de adaptação é potencialmente pequeno. A simplicidade dos requisitos de instrumentação implica que a abordagem deve ser viável para a vasta maioria das linguagens.

A incorporação de novas plataformas de middleware, por sua vez, depende da instrumentação de *proxies* e objetos remotos, para que as informações de contexto requeridas possam ser passadas de forma transparente. Além disso, é desejável que a plataforma de middleware disponibilize mecanismos para a passagem de informações de contexto com cada requisição, já que isso pode facilitar substancialmente o esforço de implementação.

O depurador disponibiliza algumas formas simples de análise automática de informações de execução: a detecção de laços recursivos distribuídos infinitos, a detecção de exceções CORBA não-declaradas e a detecção de *deadlocks* distribuídos. Essas formas de análise automática, embora limitadas, podem prover auxílio significativo ao desenvolvedor sob algumas situações específicas.

A ferramenta disponibiliza uma infra-estrutura para o controle dos processos que compõem o sistema distribuído. É possível lançar, destruir e interagir com os processos remotos por meio dessa infra-estrutura. A linguagem de especificação de restrições de lançamento permite que o usuário condicione o lançamento de um processo à satisfação de predicados locais em outros processos, facilitando a criação instâncias viáveis do sistema distribuído. Isso torna a operação de lançamento do sistema distribuído efetivamente uma "operação de um único clique", tornando mais ágil o fluxo de trabalho do usuário durante a depuração cíclica.

6 Conclusões

"When a person has discovered a truth about something and has established it with great effort, then, on viewing his discoveries more carefully, he often realizes that what he has taken great pains to find might have been perceived with greatest of ease."

-- *Galileo*

Apesar do grande número de publicações e da intensa pesquisa desenvolvida nos tópicos de verificação e depuração de sistemas concorrentes (tanto paralelos quanto distribuídos), a falta de ferramentas práticas e eficazes é ainda bastante perceptível. Isso se deve em parte ao fato de que execuções concorrentes são mais complexas de tratar do que execuções seqüenciais. Acreditamos, no entanto, que o principal fator que leva à cronicidade dessa situação não é resultado de uma complexidade intrínseca (usando a terminologia de Brooks [22]), mas de uma acidental: a heterogeneidade. O grande número de tecnologias disponíveis para a construção de sistemas distribuídos, aliado ao desenvolvimento tardio de ferramentas de depuração, à falta de padronização e ao desenvolvimento de abordagens *ad hoc* e não-coordenadas para o tratamento do problema resultam na proliferação de ferramentas incompatíveis e de vida curta que, por sua vez, resultam na disponibilidade reduzida observada.

Neste trabalho, identificamos um conjunto de elementos estruturais e "culturais" comuns a uma classe relevante de sistemas distribuídos: os sistemas de objetos distribuídos baseados em chamadas síncronas e bloqueantes. Esses elementos foram agrupados e utilizados como base na construção de uma ferramenta que apóia atividades de depuração sem impor exigências extravagantes nas tecnologias e plataformas escolhidas. Os elementos estruturais identificados dizem respeito a todos os requisitos impostos sobre o middleware, sobre as linguagens de programação escolhidas (Seção 5.4.3) e sobre os depuradores simbólicos adotados (Seção 5.5.4). O elemento cultural diz respeito à aceitação e à tendência à reinvenção do depurador simbólico, a tal ponto que é quase possível dar por certo que qualquer que seja a linguagem de programação escolhida, existe ao menos um depurador simbólico compatível com ela.

A principal idéia deste trabalho foi construir uma ferramenta melhor do aquilo que é habitualmente utilizado (Seção 4.2), nos baseando apenas em elementos comuns à classe de sistemas escolhida. Acreditamos ter dado passos importantes nessa direção e, sob esse aspecto, consideramos que o trabalho foi bem-sucedido. As dificuldades encontradas mesmo na implementação de um conceito simples como o da nossa ferramenta, no entanto, nos fazem crer que

ainda estamos muito longe de ter um conjunto adequado de elementos de apoio que viabilizem a implementação de técnicas mais sofisticadas, como as descritas no Capítulo 3. Como resultado, nossa ferramenta ataca apenas parte dos problemas clássicos da depuração distribuída.

Alguns dos problemas – como a re-execução determinística de sistemas distribuídos heterogêneos e a minimização do efeito do observador – são simplesmente muito difíceis de resolver de forma geral. O tratamento de problemas como esse depende do esforço coordenado de um número grande de participantes, já que as técnicas desenvolvidas até então envolvem elementos muito específicos de cada plataforma. Outros problemas – como a análise automática de informações e as metáforas escaláveis de visualização – poderiam ter recebido mais atenção em nosso trabalho. Optamos por focar o desenvolvimento em uma metáfora específica, no entanto – a da representação do sistema distribuído como um grande processo virtual, composto por *threads* distribuídos – que representa o objetivo principal do trabalho.

Vale notar ainda que a ferramenta desenvolvida se presta à análise dos mesmos tipos de comportamentos errôneos que são analisados com depuradores simbólicos convencionais, padecendo das mesmas limitações. Particularmente, o foco desta ferramenta é em sistemas que contam com um número pequeno (algumas dezenas) de nodos e execuções de curta duração. Em sistemas maiores, é esperado que o número de nodos sature o agente central e que as metáforas de visualização sobrecarreguem o usuário com informações, já que a análise da execução, em depuradores simbólicos, é uma atividade essencialmente manual.

Por último, avaliamos que o prognóstico para a depuração de sistemas distribuídos é ainda bastante ruim – e não por falta de soluções viáveis ao problema. Os problemas de compatibilidade apontados por Rosenberg [174], Pancake [157, 158] e Hondroudakis [83] continuam se fazendo presentes. É seguro afirmar que as plataformas em uso atualmente (agrupando aqui hardware, sistemas operacionais e middleware) são obsoletas do ponto de vista do apoio a técnicas de depuração e, pela tendência histórica observada, essa situação não deve mudar tão cedo.

6.1 Sumário de contribuições

A primeira e mais importante contribuição deste trabalho é, sem sombra de dúvidas, a concepção e a implementação do G.O.D. O conceito de um depurador simbólico distribuído, embora um tanto quanto óbvio, nunca foi materializado, até onde sabemos, de forma tão explícita. A conceitualização é também nova na medida em que tem como alicerce a portabilidade. A concepção arquitetural do depurador – uma API de metaprogramação genérica, que reifica um ambiente de execução virtual onde o sistema distribuído é tratado como um grande processo composto por *threads* distribuídos – é também inédita até onde sabemos. A implementação do

depurador simbólico descrito neste texto pode ser obtida em http://god.incubadora.fapesp.br. O código-fonte é software livre e distribuído sob a *Eclipse Public License V1.0* (http://www.eclipse.org/legal/epl-v10.html).

A segunda contribuição deste trabalho são alguns novos formalismos, como a caracterização do *thread* distribuído da Seção 4.3.2 (o principal deles) e as classificações de técnicas interativas da Seção 3.2.6. A última contribuição vem sob a forma do levantamento e análise das principais técnicas – passadas e presentes – de depuração desenvolvidas para sistemas paralelos, distribuídos e pseudo-paralelos do Capítulo 3. Esperamos que nosso levantamento e análise possam ser de utilidade a futuros pesquisadores na área.

6.2 Publicações

O desenvolvimento deste trabalho resultou em algumas publicações, enumeradas em ordem cronológica reversa a seguir:

[1] Mega, G. and Kon, F. "An Eclipse-based Tool for Symbolic Debugging of Distributed Object Applications," in *Proceedings of the 9th International Symposium on Distributed Objects, Middleware, and Applications (DOA'2007)*. 2007, LNCS 4803, Springer-Verlag: Vilamoura, Portugal. pp. 648-666.

[2] Mega, G. and Kon, F., "Depurando Sistemas de Objetos Distribuídos da Forma que Gostaríamos," in *Anais do 23º Simpósio Brasileiro de Redes de Computadores (SBRC'06)*. 2006: Curitiba, Brasil. pp. 1331-1346.

[3] Mega, G. and Kon, F., "GOD: Um Depurador Simbólico para Sistemas de Objetos Distribuídos," in *Anais do Salão de Ferramentas do 19º Simpósio Brasileiro de Engenharia de Software (SBES'05)*. 2005: Uberlândia, Brasil. Disponível: http://www.sbbd-sbes2005.ufu.br/arquivos/GOD.pdf.

[4] Mega, G. and Kon, F., "Debugging Distributed Object Applications with the Eclipse Platform," in *Proceedings of the 2004 ACM OOPSLA Eclipse Technology eXchange Workshop*. 2004: Vancouver, Canada. pp. 42-46.

Referências

[1] *Groovy - Home*, Available from: http://groovy.codehaus.org/ [Visited October 9th, 2007].

[2] *Ruby Programming Language*, Available from: http://www.ruby-lang.org [Visited October 9th, 2007].

[3] Adve, S. V., Hill, M. D., Miller, B. P., and Netzer, R. H. B., "Detecting Data Races on Weak Memory Systems," in *Proceedings of the 18th International Symposium on Computer Architecture*, Toronto, Ontario, Canada, 1991, pp. 234-243.

[4] Adve, S. V., Pai, V. S., and Ranganathan, P., "Recent advances in memory consistency models for hardware shared-memory systems," *Proceedings of the IEEE,* vol. 87, pp. 445-455, 1999.

[5] Agha, G. A., *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Cambridge, Massachusetts: The MIT Press, 1986.

[6] Allen, T. R. and Padua, D. A., "Debugging Fortran on a Shared Memory Machine," in *Proceedings of the 1987 International Conference on Parallel Processing*, St. Charles, Illinois, 1987, pp. 721-727.

[7] Anderson, D. P., "BOINC: A System for Public-Resource Computing and Storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, 2004, pp. 4-10.

[8] Apache Software Foundation, *BCEL - Byte Code Engineering Library*, Available from: http://jakarta.apache.org/bcel/manual.html [Visited October 1st, 2006].

[9] Apache Software Foundation, *Log4J - Logging for Java*, Available from: http://logging.apache.org/log4j/ [Visited April 5th, 2008].

[10] Aquamacs, *Aquamacs: Emacs for OS X*, Available from: http://aquamacs.org [Visited July 25th, 2007].

[11] Armstrong, J., *Programming Erlang: Software for a Concurrent World*: Pragmatic Bookshelf, 2007.

[12] Bach, J., "A Framework for Good Enough Testing," *IEEE Computer,* vol. 31, pp. 124-126, 1998.

[13] Bacon, D. F. and Goldstein, S. C., "Hardware-assisted replay of multiprocessor programs," in *Proceedings of the 1991 ACM/ONR Workshop on Parallel & Distributed Debugging*, Santa Cruz, CA, United States, 1991, pp. 194-206.

[14] Balzer, R., "EXDAMS, EXtensible Debugging and Monitoring System," in *Proceedings of the Spring Joint Computer Conference*, Reston, VA, 1969, pp. 567-589.

[15] Beck, K., *Extreme Programming Explained: Embrace Change*, 2nd ed.: Addison-Wesley, 2004.

[16] Bemmerl, T. and Wismuller, T., "On-line Distributed Debugging on Scalable Multicomputer Architectures," in *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking Volume II: Networking and Tools*, 1994, pp. 394-400.

[17] Bernstein, P. A., "Middleware: a model for distributed system services," *Communications of the ACM,* vol. 39, pp. 86-98, 1996.

[18] Birrel, A. D. and Nelson, B. J., "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS),* vol. 2, pp. 39-59, 1984.

[19] Bosschere, K. D. and Ronsse, M., "Clock snooping and its application in on-the-fly data racedetection," in *Third International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'97)*, Taipei, Taiwan, 1997, pp. 324-330.

[20] Bowen, J. P. and Hinchey, M. G., "Ten commandments of formal methods... ten years later.," *IEEE Computer,* vol. 39, pp. 40-48, 2006.

[21] Bray, M., *Middleware*, Available from: http://www.sei.cmu.edu/str/descriptions/middleware.html [Visited December 6th, 2007].

[22] Brooks, F., "No Silver Bullet - essence and accident in software Engineering," *IEEE Computer,* vol. 20, pp. 10-19, 1986.

[23] Burrough, P. A. and McDonnell, R. A., *Principles of geographical information systems*. Oxford: Oxford University Press, 1998.

[24] Caraveo, S. and Rethans, D., *DBGP - A common debugger protocol for languages and debugger UI communication*, Available from: http://xdebug.org/docs-dbgp.php [Visited September 23rd, 2007].

[25] Cargill, T. A. and Locanthi, B. N., "Cheap hardware support for software debugging and profiling," in *Proceedings of the second international conference on Architectural support for programming languages and operating systems*, Palo Alto, California, United States, 1987, pp. 82-83.

[26] Chandy, K. M. and Lamport, L., "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS),* vol. 3, pp. 63-75, 1985.

[27] Charron-Bost, B., "Concerning the size of logical clocks in distributed systems," *Information Processing Letters,* vol. 39, p. 11016, 1991.

[28] Cheng, D. and Hood, R., "A portable debugger for parallel and distributed programs," in *Proceedings of the 1994 conference on Supercomputing*, Washington, D.C., United States, 1994.

[29] Chiba, S., "A metaobject protocol for C++," in *Proceedings of the tenth annual conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, 1995, pp. 285-299.

[30] Choi, J.-D. and Miller, B. P., "Breakpoints and Halting in Distributed Programs," in *Proceedings of th 8th International Conference on Distributed Computing Systems*, 1988, pp. 316-323.

[31] Choi, J.-D., Miller, B. P., and Netzer, R. H. B., "Techniques for Debugging Parallel Programs with Flowback Analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 13, pp. 491-530, 1991.

[32] Choi, J.-D. and Min, S. L., "Race Frontier: reproducing data races in parallel-program debugging," in *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, Williamsburg, Virgina, United States, 1991, pp. 145-154.

[33] Choi, J.-D. and Srinivasan, H., "Deterministic replay of Java multithreaded applications," in *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, 1998, pp. 44-59.

[34] Christiaens, M. and Bosschere, K., "TRaDe, a topological approach to on-the-fly race detection in java programs," in *Proceedings of the USENIX Java (tm) Virtual Machine Research and Technology Symposium*, Monterrey, California, 2001, pp. 15-26.

[35] Cirne, W., Brasileiro, F. V., Andrade, N., Costa, L., Andrade, A., Novaes, R., and Mowbray, M., "Labs of the World, Unite!!!," *Journal of Grid Computing,* vol. 4, pp. 225-246, 2006.

[36] Clark, R. K., Jensen, E. D., and Reynolds, F. D., "An Architectural Overview of The Alpha Real-Time Distributed Kernel," in *Proceedings of the 1993 Winter USENIX Conference*, 1993, pp. 127-146.

[37] Clemencon, C., Fritscher, J., and Rühl, R., "Execution Control and Replay of Massively Parallel Programs within Annai's Debugging Tool," in *Proceedings of the 1995 High Performance Computing Symposium (HPCS'95)*, Montreal, Canada, 1995, pp. 393-404.

[38] Cole, M., "Why Structured Parallel Programming Matters," in *Proceedings of the Euro-Par 2004*, 2004, p. 37.

[39] Cook, C., Burnett, M., and Boom, D., "A bug's eye view of immediate visual feedback in direct-manipulation programming systems," in *Proceedings of the Senventh Workshop on Empirical Studies of Programmers*, Alexandria, Virgina, 1997, pp. 20-41.

[40] Cooper, R. and Marzullo, K., "Consistent detection of global predicates," in *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, Santa Cruz, California, United States, 1991, pp. 167-174.

[41] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., *Introduction to Algorithms*, 2nd ed.: The MIT Press, 2001.

[42] Cornelis, F., Georges, A., Christiaens, M., Ronsse, M., Ghesquiere, T., and Bosschere, K. D., "A Taxonomy of Execution Replay Systems," in *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[43] Cornelis, F., Ronsse, M., and Bosschere, K. d., "TORNADO: A Novel Input Replay Tool," in *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA' 03)*, 2003, pp. 1598-1604.

[44] Curtis, R. S. and Wittie, L. D., "BugNet: A debugging system for parallel programming environments," in *In Proceedings of the 3rd International Conference on Distributed Computing Systems*, Miami, Florida, 1982, pp. 394-399.

[45] Cypher, R. and Leu, E., "Efficient race detection for message-passing programs with nonblocking sends and receives," in *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, 1995, p. 534.

[46] Damodaran-Kamal, S. K., *Testing and Debugging Nondeterministic Message Passing Programs*, PhD Thesis, University of Southwestern Louisiana, 1986.

[47] Dionne, C., Feeley, M., and Desbiens, J., "A Taxonomy of Distributed Debuggers Based on Execution Replay," in *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, Sunnyvale, California, USA, 1996, pp. 203-214.

[48] Dmitriev, M. A., "Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications," in *Proceedings of The Workshop on Engineering Complex Object-Oriented Systems for Evolution*, Portland, Oregon, 2001, pp. 14-18.

[49] Dodd, P. S. and Ravishankar, C. V., "Monitoring and Debugging Distributed Real-time Programs," *Software: Practice and Experience,* vol. 22, pp. 863-877, 1992.

[50] Dollimore, J., Kindberg, T., and Coulouris, G., *Distributed Systems: Concepts and Design*, 4th ed.: Addison Wesley, 2005.

[51] Dubois, P. F., "Scientific Components are Coming," *IEEE Computer,* vol. 32, pp. 115-117, 1999.

[52] Duchien, L. and Seinturier, L., "Reflection and Debug for CORBA Applications," CNAM-Laboratoire CEDRIC, Technical Report CNAM-CEDRIC 99-10, 1999.

[53] Dumant, B., Horn, F., Dang, F. D. T., and Stéfani, J.-B., "Jonathan: an open distributed processing environment in Java," *Distributed Systems Engineering,* vol. 6, pp. 3-12, 1999.

[54] Dummet, M., "Is Time a Continuum of Instants," *Philisophy,* pp. 497-515, 2000.

[55] Eckel, B., *Thinking in Python*, Available from: http://www.mindview.net/Books/TIPython [Visited October 9th, 2007].

[56] Eclipse Foundation, *Eclipse C/C++ Development Tools Website*, Available from: http://www.eclipse.org/cdt/ [Visited August 13th, 2007].

[57] Eclipse Foundation, *Eclipse Java Development Tools Website*, Available from: http://www.eclipse.org/jdt/ [Visited July 25th, 2007].

[58] Eclipse Foundation, *Eclipse.org home*, Available from: http://www.e clipse.org [Visited October 15th, 2007].

[59] Elshoff, I. J. P., "A distributed debugger for Amoeba," in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, Madison, Wisconsin, United States, 1989, pp. 1-10.

[60] Emrath, P. A. and Padua, D. A., "Automatic Detection Of Nondeterminacy in Parallel Programs," in *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, United States, 1988, pp. 89-99.

[61] Fayad, M. and Schmidt, D., "Object-Oriented Application Frameworks," *Communications of the ACM,* vol. 40, pp. 32-38, 1997.

[62] Feldman, S. and Brown, C., "IGOR: a system for program debugging via reversible execution," in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, Madison, Wisconsin, United States, 1988, pp. 112-123.

[63] Fidge, C., "Fundamentals of distributed system observation," *IEEE Software,* vol. 13, pp. 77-83, 1996.

[64] Fidge, C., "Timestamps in Message-Passing Systems That Preserve the Partial Ordering," in *Proceedings of the 11th Australian Computer Science Conference*, University of Queensland, 1988, pp. 55-66.

[65] Fowler, J. and Zwaenepoel, W., "Causal Distributed Breakpoints," in *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, 1990, pp. 134-141.

[66] Fowler, M., *Inversion of Control Containers and the Dependency Injection pattern*, Available from: http://martinfowler.com/articles/injection.html [Visited December 4th, 2007].

[67] Francioni, J. and Pancake, C. M., "High Performance Debugging Standards Effort," *Scientific Programming,* vol. 8, pp. 95-108, 2000.

[68] Gait, J., "The Probe Effect in Concurrent Programs," *IEEE Software: Practice and Experience,* vol. 16, pp. 225-233, 1986.

[69] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed.: Addison-Wesley, 1994.

[70] Garg, V. K., "Observation of global properties in distributed systems," in *Proceedings of the Eighth IEEE International Conference on Software and Knowledge Engineering*, 1996, pp. 418-425.

[71] Georges, A., Christiaens, M., Ronsse, M., and Bosschere, K., "JaRec: a portable record/replay environment for multi-threaded Java applications," *Software: Practice and Experience,* vol. 34, pp. 523-547, 2004.

[72] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*: Addison-Wesley, 1983.

[73] Goldchleger, A., Kon, F., Goldman, A., Finger, M., and Bezerra, G. C., "InteGrade: object-oriented Grid middleware leveraging idle computing power of desktop machines," *Concurrency and Computation: Practice and Experience,* vol. 16, pp. 449-459, 2004.

[74] Gosling, J., Joy, B., Steele, G., and Bracha, G., *The Java Language Spefication*, 3rd ed.: Addison-Wesley, 2005.

[75] Gracanin, D., Matkovic, K., and Eltoweissy, M., "Software visualization," *Innovations in Systems and Software Engineering: A NASA Journal,* vol. 1, pp. 221-230, 2005.

[76] Hadzilacos, V. and Toueg, S., "Fault-Tolerant Broadcasts and Related Problems," in *Distributed Systems*, 2nd ed, S. Mullender, Ed.: ACM Press, 1993.

[77] Haller, P. and Odersky, M., "Actors that Unify Threads and Events," École Polytechnique Fédérale de Lausanne, Technical Report LAMP-REPORT-2007-001, 2007.

[78] Haller, P. and Odersky, M., "Event-Based Programming without Inversion of Control," in *Proceedings of the 7th Joint Modular Languages Conference*, Oxford, UK, 2006, pp. 4-22.

[79] Hauben, R., *From the ARPANET to the Internet: A Study of the ARPANET TCP/IP Digest and of the Role of Online Communication in the Transition from the ARPANET to the Internet*, Available from: http://www.columbia.edu/~rh120/other/tcpdigest_paper.txt [Visited December 5th, 2007].

[80] Henning, M., "The Rise and Fall of CORBA," *ACM Queue,* vol. 4, p. 28034, 2006.

[81] Herdieckerhoff, M. and Ruget, F., "A distributed execution replay facility for CHORUS," in *Proceedings of the 7th International Conference on Parallel and Distributed Systems (PDCS'94)*, Las Vegas, Nevada, 1994.

[82] Ho, A., Smith, A., and Hand, S., "On deadlock, livelock, and forward progress," Cambridge University, Technical Report UCAM-CL-TR-633, 2005.

[83] Hondroudakis, A., "Performance Analysis Tools for Parallel Programs," Edinburgh Parallel Computing Centre, The University of Edinburgh, 1995.

[84] Hood, R., Kennedy, K., and Mellor-Crummey, J., "Parallel program debugging with on-the-fly anomaly detection," in *Conference on High Performance Networking and Computing*, New York, 1990, pp. 74-81.

[85] Hwang, K. and Xu, Z., *Scalable Parallel Computing - Technology, Architecture, Programming*. USA: WCB McGraw-Hill, 1998.

[86] IBM, *IBM WebSphere Application Server*, Available from: http://publib.boulder.ibm.com /infocenter/wasinfo/v4r0/index.jsp?topic=/com.ibm.websphere.v4.doc/olt_content/olt/ref/rp noltpd.htm [Visited August 3rd, 2007].

[87] IBM, *Object Level Trace*, Available from: http://publib.boulder.ibm.com/infocenter/was info/v4r0/index.jsp?topic=/com.ibm.websphere.v4.doc/olt_content/olt/ref/rpnoltpd.htm [Visited August 3rd, 2007].

[88] IBM Corporation, *Rational Robot Website*, Available from: http://www-304.ibm .com/jct03001c/software/awdtools/tester/robot/ [Visited December 7th, 2007].

[89] IBM Corporation, *Websphere Application Server v4.0x - Distributed Debugger*, Available from:http://publib.boulder.ibm.com/infocenter/wasinfo/v4r0/index.jsp?topic=/com.ibm. websphere.v4.doc/olt_content/olt/tasks/tdstart.htm [Visited December 6th, 2007].

[90] IEEE, "Portable Operating System Interface (POSIX)," Standard for Information Technology Std. 1003.1-2001, 2001.

[91] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A., "Back to the future: the story for Squeak, a practical Smalltalk written in itself," in *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, Atlanta, Georgia, 1997, pp. 318-326.

[92] Institute of Electrical and Electronics Engineers (IEEE), *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York, NY, 1990.

[93] Intel Corporation, *System Performance Visualization Tool User's Guide*: Intel Corporation, 1993.

[94] Jard, C. and Jourdan, G.-C., "Dependency Tracking and Filtering in Distributed Computations," in *Brief Announcements of the ACM Symposium on Principles of Distributed Computing*, New York, 1994, pp. 134-141.

[95] Johnson, J. and Kenney, G., "Implementation Issues for a Source Level Symbolic Debugger," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-level Debugging*, 1983, pp. 149-151.

[96] Johnson, R. E., "Components, Frameworks, Patterns," in *Proceedings of the 1997 ACM SIGSOFT Symposium on Software Reusability*, 1997, pp. 10-17.

[97] Kahn, R. E., "Resource-sharing computer communications networks," *Proceedings of the IEEE,* vol. 60, pp. 1397-1407, Nov. 1972.

[98] Kaner, C., "The impossibility of complete testing," *Software QA,* vol. 4, p. 28, 1997.

[99] Kimelman, D. and Zernik, D., "On-the-Fly Topological Sort - A Basis for Interactive Debugging and Live Visualization of Parallel Programs," in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993, pp. 12-20.

[100] Klar, R., "Event-Driven Monitoring of Parallel Systems," in *Proceedings of the workshop on performance measurement and visualization systems*, Moravany, Czechoslovakia, 1993, pp. 145-173.

[101] Klar, R., Dauphin, P., Hartleb, F., Hofmann, R., Mohr, B., Quick, A., and Siegle, M., *Messung und Modellierung paralleler und verteilter Rechensysteme*. Stuttgart, Germany: B. G. Teubner, 1995.

[102] Kon, F., Costa, F., Campbell, R., and Blair, G., "The Case for Reflective Middleware," *Communications of the ACM,* vol. 16, pp. 449-459, June 2002.

[103] Konuru, R., Choi, J.-D., and Srinivasan, H., "Deterministic Replay of Distributed Java Applications," in *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, 2000, pp. 219-229.

[104] Kranzlmueller, D., *Event Graph Analysis for Debugging Massively Parallel Programs*, PhD Thesis, Dept. for Graphics and Parallel Processing, Joh. Kepler University, Linz, Austria, 2000.

[105] Kranzlmueller, D., "Visualizing Program Behavior with the Event Graph," in *Software Visualization - From Theory to Practice*, K. Zhang, Ed.: Kluwer Academic Publishers, 2003, pp. 28-56.

[106] Krishnamurthy, Y., Pyarali, I., Gill, C., Mgeta, L., Zhang, Y., Torri, S., and Shmidts, D. C., "The Design and Implementation of Real-Time CORBA 2.0: Dynamic Scheduling in TAO," in *Proceedings of 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, 2004, pp. 121-129.

[107] Kumar, A., "Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors," *The Journal of Computing Sciences in Colleges,* vol. 20, pp. 36-46, May 2005 2005.

[108] Kunz, T., "Process Clustering for Distributed Debugging," in *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993, pp. 75-84.

[109] Lamport, L., "On interprocess communication - Part I: Basic formalism," *Distributed Computing,* vol. 1, pp. 77-85, 1986.

[110] Lamport, L., "Time, Clocks, and the ordering of events in a distributed system," *Communications of the ACM,* vol. 21, pp. 558-565, 1978.

[111] LeBlanc, T. J. and Mellor-Crummey, J., "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers,* vol. 36, pp. 471-482, 1987.

[112] Leu, E., Schiper, A., and Zramdini, A., "Efficient execution replay technique for distributed memory architectures," in *Proceedings of the 2nd European Conference on Distributed memory computing*, Munich, Germany, 1991, pp. 315-324.

[113] Levrouw, L. and Audenaert, K. M. R., "Interrupt replay: a debugging method for parallel programs with interrupts," *Microprocessors and Microsystems,* vol. 18, pp. 601-612, 1994.

[114] Levrouw, L., Audenaert, K. M. R., and Campenhout, J. M., "A New Trace and Replay System for Shared Memory Programs based on Lamport Clocks," in *Proceedings of the Second EUROMICRO Workshop on Parallel and Distributed Processing*, 1994, pp. 471-478.

[115] Lewis, B., "Debugging Backwards in Time," in *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG'03)* Ghent, Belgium, 2003. Available from: http://www.lambdacs.com/debugger/AADEBUG_Mar_03.pdf

[116] Li, J., "Characterization of Component-Based Systems," HP Labs Technical Report HPL-2002-25(R.1), 2002.

[117] Lima, A., Cirne, W., Brasileiro, F. V., and Fireman, D., "A Case for Event-Driven Distributed Objects," in *Proceedings of the 2006 Conference on Distributed Objects and Applications (DOA'06)*, 2006, pp. 1705-1721.

[118] Lourenço, J., Cunha, J. C., Krawczyk, H., Neyman, M., and Wiszniewski, B., "An integrated testing and debugging environment for parallel and distributed programs," in *Proceedings of the 23rd EUROMICRO Conference*, 1997, pp. 291-298.

[119] Lourenço, J., Cunha, J. C., and Lourenço, V., "Control and Debugging of Distributed Programs using Fiddle," in *Proceedings of the 2003 AADEBUG*, Ghent, Belgium, 2003.

[120] Ludwig, T. and Wismüller, R., "OMIS 2.0 - A Universal Interface for Monitoring Systems," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 4th European PVM/MPI Users Group Meeting*, M. Bubak, J. J. Dongarra, and J. Wasniewski, Eds. Crakow, Poland: Springer-Verlag, 1997, pp. 267-276.

[121] Lynch, N. A., *Distributed algorithms*. San Francisco, Calif.: Morgan Kaufmann Publishers, 1996.

[122] Mackey, M., "Program Replay in PVM," Concurrent Computing Department, HP Labs Technical Report,, 1993.

[123] Maebe, J., Ronsse, M., and Bosschere, K. d., *DIOTA: Dynamic Instrumentation, Optmisation, and Transformation of Applications*, Available from: http://escher.elis.ugent .be/publ/Edocs/DOC/P102_065.pdf [Visited December 5th, 2007].

[124] Maes, P., "Concepts and Experiments in Computational Reflection," in *Proceedings of the ACM Conference on Object-oriented programming systems, languages, and applications.*, Orlando, Florida, United States, 1987, pp. 147-155.

[125] Maletic, J. I., Leigh, J., Marcus, A., and Dunlap, G., "Visualizing object-oriented software in virtual reality," in *Proceedings of the 9th international workshop on program comprehension (IWPC'01)*, Toronto, Canada, 2001.

[126] Maletic, J. I., Marcus, A., and Collard, M. L., "A task oriented view of software visualization," in *Proceedings of the IEEE Workshop on Visual Languages*, Seattle, Washington, 2002.

[127] Mattern, F., "Virtual Time and Global States of Distributed Systems," in *Proceedings of the Workshop on Parallel and Distributed Algorithms*, 1989, pp. 215-226.

[128] May, J. and Berman, F., "Panorama: A Portable, Extensible Parallel Debugger," in *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, CA, United States, 1993, pp. 96-106.

[129] McLean, J., "Twenty Years of Formal Methods," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 1999, pp. 115-116.

[130] Mega, G. and Kon, F., "GOD: Um depurador simbólico para sistemas de objetos distribuídos," in *Tools Track of the 2005 Brazilian Symposium on Software Engineering (SBES'05)*, Uberlândia, Brasil, 2005. Available from: http://www.sbbd-sbes2005 .ufu.br/arquivos/GOD.pdf

[131] Mellor-Crummey, J., *Debugging and Analysis of Large-Scale Parallel Programs*, PhD Thesis, University of Rochester, 1989.

[132] Mellor-Crummey, J. and LeBlanc, T. J., "A Software Instruction Counter," in *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, Boston, Massachusetts, United States, 1989, pp. 76-86.

[133] Meloan, S., *Testing and Java Technology*, Available from: http://java.sun.com/developer/technicalArticles/InnerWorkings/testing/ [Visited December 1st, 2007].

[134] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, Available from: http://www.mpi-forum.org/docs/mpi2-report.pdf [Visited December 5th, 2007].

[135] Miller, B. P. and Choi, J.-D., "A Mechanism for Efficient Debugging of Parallel Programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanga, Georgia, United States, 1988, pp. 135-144.

[136] Miller, B. P., Clark, M., Hollingsworth, J., Kierstead, S., Lim, S. S., and Torzewski, T., "IPS-2: The Second Generation of a Parallel Program Measurement System," *IEEE Transactions on Parallel and Distributed Systems,* vol. 1, pp. 206-217, 1990.

[137] Mittal, N. and Garg, V., "On Detecting Global Predicates in Distributed Computations," in *Proceedings of the 21st International Conference on Distributed Computing Systems*, 2001, pp. 3-10.

[138] Mohr, B., *Ereignisbasierte Rechneranalysesysteme zur Bewertung paralleler und verteilter Systeme*, PhD Thesis, University of Erlangen-Nürnberg, 1992.

[139] Mohr, B., "Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis System Possible?," in *Environments and Tools for Parallel Scientific Computing*, J. J. Dongarra and B. Tourancheau, Eds.: Elsevier Science Publishers, 1993, pp. 103-124.

[140] Mozilla Foundation, *The Venkman JavaScript Debugger Project*, Available from: http://www.mozilla.org/projects/venkman/ [Visited July 25th, 2007].

[141] Muddarangegowda, G. and Pancake, C. M., "The Lightweight Corefile Browser," Department of Computer Cience, Oregon State University, Technical Report CSRT94-80-17, 1994.

[142] Narayanasamy, S., Pokam, G., and Calder, B., "BugNet: continuously recording program execution for deterministic replay debugging," in *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005, pp. 284-295.

[143] Nelson, B. J., *Remote Procedure Call*, PhD Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA, 1981.

[144] Netzer, R. H. B., "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs," in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, California, US, 1993, pp. 1-11.

[145] Netzer, R. H. B., *Race Condition Detection for Debugging Shared-memory Parallel Programs*, PhD Thesis, University of Wisconsin-Madison, 1991.

[146] Netzer, R. H. B., Brennan, T. W., and Damoradan-Kamal, S. K., "Debugging race conditions in message-passing programs," in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, Philadelphia, Pennsylvania, United States, 1996, pp. 31-40.

[147] Netzer, R. H. B. and Miller, B. P., "On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions," University of Wisconsin-Madison, Technical Report TR908, 1990.

[148] Netzer, R. H. B. and Miller, B. P., "Optimal tracing and replay for debugging message-passing parallel programs," in *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, Minneapolis, Minnesota, United States, 1992, pp. 502-511.

[149] Netzer, R. H. B. and Miller, B. P., "What are race conditions?: Some issues and formalizations," *ACM Letters on Programming Languages and Systems (LOPLAS),* vol. 1, pp. 74-88, 1992.

[150] Object Computing Inc., *OCI - Products - OVATION - Object Computing, Inc.*, Available from: http://www.ociweb.com/products/ovation [Visited August 3rd, 2007].

[151] Object Management Group, *The Common Object Request Broker Architecture*, Available from: http://www.omg.org/cgi-bin/apps/doc?formal/04-03-12.pdf [Visited September 25th, 2007].

[152] Object Management Group, *CORBA Component Model Specification, v4.0*, 2006.

[153] OpenQA, *Selenium IDE*, Available from: http://www.openqa.org/selenium-ide/ [Visited December 6th, 2007].

[154] OSGi Alliance, *OSGi Service Platform Core Specification, Release 4, Version 4.1*, 2007.

[155] Otta, M., "A Distributed Debugger Framework Applicable in a Java/CORBA Environment," in *European Research Seminar on Advances in Distributed Systems*. Available from: http://www.cs.unibo.it/ersads/papers/otta.ps

[156] Pancake, C. M., "Applying Human Factors to the Design of Performance Tools," in *Proceedings of the Euro-Par 1999*, 1999, pp. 44-60.

[157] Pancake, C. M., "Collaborative Efforts to Develop User-Oriented Parallel Tools," in *Debugging and Performance Tuning for Parallel Computing Systems*, M. L. Simmons, A. H. Hayes, J. S. Brown, and D. A. Reed, Eds. Lost Alamitos, CA, USA: IEEE Computer Society Press, 1996, pp. 355-366.

[158] Pancake, C. M., *Establishing Standards for HPC System Software and Tools*, Available from: http://nhse.cs.rice.edu/NHSEreview/97-1.html [Visited December 4th, 2007].

[159] Pancake, C. M., "Performance tools for today's HPC: Are we addressing the right issues?," *Parallel Computing,* vol. 27, pp. 1403-1415, 2001.

[160] Paul, J., *Software Engineering - General Testing and Debugging Guidelines*, Available from: http://www.jody-paul.com/SWE/TD/TestDebug.html [Visited November 15th, 2006].

[161] Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D., *Java Concurrency in Practice*: Addison-Wesley, 2006.

[162] Pressman, R. S., *Software Engineering: A Practitioner's Approach*, 6th ed.: McGraw-Hill.

[163] Price, B. A., Baecker, R. M., and Small, I. S., "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages and Computing,* vol. 4, pp. 211-266, 1993.

[164] Price, B. A., Small, I. S., and Baecker, R. M., "A Taxonomy of Software Visualization " in *Proceedings of the Twenty-Fifth IEEE Hawaii International Conference on System Sciences*, 1992, pp. 597-606.

[165] Prvulovic, M., "CORD: Cost Effective (and nearly overhead-free) Order Recording and Data race detection," in *Proceedings of The Twelfth International Symposium on High-Performance Computer Architecture*, 2006, pp. 232-243.

[166] Prvulovic, M. and Torrellas, J., "ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003, pp. 110-121.

[167] Ramsey, N., "Correctness of Trap-Based Breakpoint Implementations," in *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'94)*, 1994, pp. 15-24.

[168] Raynal, M. and Singhal, M., "Logical Time: Capturing Causality in Distributed Systems," *IEEE Computer,* vol. 29, pp. 49-56, 1996.

[169] Read, R., "Is "What is time" a good question to ask?," *Philosophy,* vol. 77, pp. 193-209, 2002.

[170] Rethans, D., *Xdebug - Debugger and Profiler Tool for PHP*, Available from: http://www.xdebug.org [Visited September 23rd, 2007].

[171] Roman, G.-C. and Cox, K. C., "A Taxonomy of Program Visualization Systems," *IEEE Computer,* vol. 26, pp. 11-24, 1993.

[172] Ronsse, M. and Bosschere, K., "JiTI: A Robust Just in Time Instrumentation Technique," in *Proceedings of the 2000 Workshop on Binary Translation*, 2000, pp. 1-12.

[173] Ronsse, M. and Bosschere, K., "RecPlay: a fully integrated practical record/replay system," *ACM Transactions on Computer Systems (TOCS),* vol. 17, pp. 133-152, 1999.

[174] Rosenberg, J. B., *How Debuggers Work: Algorithms, Data Structures, and Architecture*: Wiley, 1996.

[175] Rudoff, A. M., Fenner, B., and Stevens, W. R., *Unix Network Programming, Volume 1: The Sockets Networking API*: Addison-Wesley, 2003.

[176] Russinovich, M. and Cogswell, B., "Operating System Support for Replay of Concurrent Non-Deterministic Shared Memory Applications," *Bulletin of the Technical Committee on Operating Systems and Applications Environments (TCOS),* vol. 7, pp. 15-19, 1995.

[177] Russinovich, M. and Cogswell, B., "Replay for Concurrent Non-Deterministic Shared-Memory Applications," in *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, Philadelphia, Pennsylvania, United States, 1996, pp. 258-266.

[178] Schmidt, D. and Vinoski, S., "The CORBA Component Model: Part 1, Evolving Towards Component Middleware," *C/C++ Users Journal,* 2004. Available from: http://www.ddj .com/cpp/184403884 [Visited January 10th].

[179] Schmidt, D. C., *The ACE ORB*, Available from: http://www.cs.wustl.edu/~schmidt/ TAO.html [Visited August 3rd, 2007].

[180] Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F., *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects* vol. 2. New York: Wiley & Sons, 2000.

[181] Schneiderman, B., "Creativity support tools," *Communications of the ACM,* vol. 45, pp. 116-120, 2002.

[182] Scholten, H. and Posthuma, J., "A debugging tool for distributed systems," in *Proceedings of the IEEE Region 10 Conference on Computer, Communication, Control and Power Engineering*, Beijing, China, 1993, pp. 173-176.

[183] Schroeder, M. D., "A State-of-the-Art Distributed System: Computing with BOB," in *Distributed Systems*, 2nd ed, S. Mullender, Ed.: ACM Press, 1993.

[184] Schuppan, V., Baur, M., and Biere, A., "JVM Independent Replay in Java," *Electronic Notes in Theoretical Computer Science,* vol. 113, pp. 85-104, 2005.

[185] Schwarz, R. and Mattern, F., "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distributed Computing,* vol. 7, pp. 149-174, 1994.

[186] Sen, A. and Garg, V., "Detecting Temporal Logic Predicates in the Happened Before Model," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, Florida, United States, 2002, pp. 76-83.

[187] Sen, A. and Garg, V. K., "On Checking Whether a Predicate Definitely Holds," in *Proceedings of the Third International Workshop on Formal Approaches to Software Testing*, Montréal, Canada, 2004, pp. 15-29.

[188] Shoja, G. C., Clarke, G., Taylor, T., and Taylor, W., "A software facility for load sharing and parallel processing in workstation environments," in *Proceedings of the 21st Annual Hawaii International Conference on Software Track*, Kailua-Kona, Hawaii, United States, 1988, pp. 222-231.

[189] Side, R. S. and Shoja, G. C., "A Debugger for Distributed Programs," *Software: Practice and Experience,* vol. 24, pp. 507-525, 1995.

[190] Sienkiewicz, J. and Radhakrishnan, T., "DDB: A Distributed Debugger Based on Replay," *Journal of High Performance Computing,* vol. 4, pp. 37-45, 1997.

[191] Singhal, M. and Kshemkalyani, A., "An efficient Implementation of Vector Clocks," *Information Processing Letters,* vol. 43, pp. 47-52, 1992.

[192] Smith, E. T., "Debugging Tools for Message-Based, Communicating Processes," in *Proceedings of the 4th International Conference on Distributed Computing Systems*, San Francisco, CA, United States, 1984, pp. 303-310.

[193] Snelling, D. and Hoffmann, G.-R., "A Comparative Study of Libraries for Parallel Processing," *Parallel Computing,* vol. 8, pp. 255-266, 1988.

[194] Socher, G., *Running applications remotely with X11*, Available from: http://www.linux focus.org/English/January2002/article222.shtml [Visited October 31st, 2007].

[195] Software Research Inc., *TestWorks for UNIX: CAPBAK*, Available from: http://www.soft .com/Products/Regression/capbak.html [Visited December 7th, 2007].

[196] Srinivasan, S. M., Mandula, S., Andrews, C. R., and Zhou, Y., "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging," in *Proceedings of the 2004 USENIX Annual Technical Conference*, 2004, pp. 29-44.

[197] Stallman, R. M., *GDB Manual: The GNU Source Level Debugger*. Cambridge, Massachusetts: Free Software Foundation, 1987.

[198] Statsko, J. T. and Patterson, C., "Understanding and Characterizing Software Visualization Systems," in *Proceedings of the IEEE Workshop on Visual Languages*, Seattle, Washigton, United States, 1992, pp. 3-10.

[199] Steffan, J. G. and Colohan, C. B., "Architectural Support for Thread-Level Speculation," School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-97-188, 1997.

[200] Steven, J., Chandra, P., Fleck, B., and Podgurski, A., "jRapture: A Capture/Replay tool for observation-based testing," *ACM SIGSOFT Software Engineering Notes* vol. 25, pp. 158-167, 2000.

[201] Sun Microsystems, *The Java Platform Debug Architecture*, Available from: http://java.sun .com/javase/6/docs/technotes/guides/jpda/ [Visited December 7th, 2007].

[202] Sun Microsystems, *Java(tm) Debug Wire Protocol*, Available from: http://java.sun.com /javase/6/docs/platform/jpda/jdwp/jdwp-protocol.html [Visited September 23rd, 2007].

[203] Sun Microsystems, *JDB - The Java Debugger*, Available from: http://java.sun.com/j2se /1.5.0/docs/tooldocs/windows/jdb.html [Visited August, 13th, 2007].

[204] Sun Microsystems, *JSR 220: Enterprise JavaBeans (tm), Version 3.0: EJB Core Contracts and Requirements*, Available from: http://jcp.org/aboutJava/communityprocess/final/ jsr220/index.html [Visited December 5th, 2007].

[205] Sun Microsystems, *JVM Tool Interface*, Available from: http://java.sun.com/j2se/1.5.0/ docs/guide/jvmti/jvmti.html [Visited August 13th, 2007].

[206] Sun Microsystems, *OpenSolaris Project Home*, Available from: http://www.opensolaris .org/os/ [Visited December 6th, 2007].

[207] Sun Microsystems, *Overview (Java Debug Interface)*, Available from: http://java.sun.com /javase/6/docs/jdk/api/jpda/jdi/index.html [Visited October 22nd, 2007].

[208] Sutter, H. and Laurus, J., "Software and the concurrency revolution," *ACM Queue,* vol. 3, pp. 54-62, 2005.

[209] Szurszewski, J., *We Have Lift-off: The Launching Framework in Eclipse*, Available from: http://www.eclipse.org/articles/Article-Launch-Framework/launch.html [Visited 2007 July 30th.

[210] Tai, K. C., "Race detection for message-passing programs.," North Carolina State University, Technical Report TR-96-OJ, 1995.

[211] Tamches, A. and Miller, B. P., "Fine-grained dynamic instrumentation of commodity operating system kernels," in *Proceedings of the third Symposium on Operating Systems Design and Implementation (OSDI'99)*, 1999, pp. 117-130.

[212] Tanenbaum, A. and Steen, M. v., *Distributed Systems: Principles and Paradigms*, 2nd ed.: Prentice Hall, 2002.

[213] Tanenbaum, A. and Woodhull, A. S., *Operating Systems: Design and Implementation*, 2nd ed.: Prentice Hall, 1997.

[214] Tanenbaum, A. S., *Structured Computer Organization*, 5th ed.: Prentice-Hall, 2005.

[215] Tanenbaum, A. S. and Renesse, R. v., "A critique of the remote procedure call paradigm," in *Proceedings of the EUTECO 88 Conference*, Vienna, Austria, 1988, pp. 775-783.

[216] Tarafdar, A. and Garg, V. K., "Predicate Control for Active Debugging of Distributed Programs," in *Proceedings of the 9th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, Orlando, USA, 1998, pp. 763-769.

[217] Thain, D., Tannenbaum, T., and Livny, M., "Condor and the Grid," in *Grid Computing: Making The Global Infrastructure a Reality*, F. Berman, A. J. G. Hey, and G. Fox, Eds.: Wiley, 2003.

[218] The Globus Alliance, *Globus Website*, Available from: http://www.globus.org/ [Visited December 5th, 2007].

[219] The Open Group, *Introduction to OSF DCE*, 1997.

[220] Tilevich, E. and Smaragdakis, Y., "Portable and Efficient Distributed Threads for Java," in *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, 2004, pp. 478-492.

[221] Tomlinson, A. I. and Garg, V. K., "Detecting Relational Global Predicates in Distributed Systems," in *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, California, 1993, pp. 21-31.

[222] TOP500.Org, *TOP500 Supercomputing Sites*, Available from: http://www.top500 .org/list/2007/11/100 [Visited December 5th, 2007].

[223] Ungar, D. and Bracha, G., "Mirrors: design principles for meta-leval facilities of object-oriented programming languages," in *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Vancouver, British Columbia, Canada, 2004, pp. 331-344.

[224] Vinoski, S., "RPC Under Fire," *IEEE Internet Computing,* vol. 9, pp. 93-95, 2005.

[225] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S., "A Note on Distributed Computing," Sun Microsystems Labs, Technical Report SMLI TR-94-29, 1994.

[226] White, J. E., *RFC707: A High-Level Framework for Network-Based Resource Sharing*, Available from: http://tools.ietf.org/rfc/rfc707.txt [Visited December 5th, 2007].

[227] Wikimedia Foundation, *Wikipedia*, Available from: http://www.wikipedia.org [Visited December 5th, 2007].

[228] Wilkinson, B. and Allen, M., *Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computers*. New Jersey, USA: Prentice Hall, 1999.

[229] Xiong, J., Wang, D., Zheng, W., and Shen, M., "Buster: A Portable Debugger for PVM," in *Proceedings of the 2nd International Conference on Algorithms and Architectures for Parallel Processing*, 1996, pp. 124-129.

[230] Xtradyne Corporation, *JacORB*, Available from: http://www.jacorb.org/ [Visited December 10th, 2007].

[231] Xu, M., Bodix, R., and Hill, M. D., "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Proceedings of the 30th annual international symposium on computer architecture*, San Diego, California, 2003, pp. 122-135.

[232] Yep, C., *A Debugging Support Based on Breakpoints for Distributed Programs Running Under Mach*, M. Comp. Sc. Thesis, Concordia University, Montrieal, 1992.

[233] Ylonen, T., "The Secure Shell (SSH) Protocol Architecture," *RFC 4251,* 2006.

[234] Zhu, W., *Service-context propagation over RMI*, Available from: http://www.javaworld .com/javaworld/jw-01-2005/jw-0117-rmi.html?page=1 [Visited October 3rd, 2007].