Depurando Sistemas de Objetos Distribuídos da Forma que Gostaríamos

Giuliano Mega e Fabio Kon

Departamento de Ciência Computação – IME/USP

{giuliano,kon}@ime.usp.br

http://eclipse.ime.usp.br/projects/DistributedDebugging

Abstract. *Developing distributed applications is a difficult task. Though the introduction of object-oriented middleware effectively relieves the developer from many of the burdens associated with programming in heterogeneous environments, many issues remain unresolved, particularly those related to distributed debugging and testing. This paper describes a new approach and a tool for debugging distributed object applications, both of which are centered around the distributed thread concept. Distributed threads arise as a natural consequence of the synchronous-call style adopted by the most popular object-oriented middleware systems.*

Resumo. *Desenvolver aplicações distribuídas é uma tarefa árdua. Embora a introdução do middleware orientado a objetos tenha contribuído com uma redução nas dificuldades relacionadas à programação em ambientes heterogêneos, muitas questões ainda permanecem em aberto, em especial aquelas relacionadas à depuração e ao teste desses sistemas. Este artigo apresenta uma nova abordagem e uma ferramenta para a depuração de sistemas de objetos distribuídos, ambas centradas no conceito de thread distribuído. Threads distribuídos aparecem como conseqüência natural do mecanismo de chamadas síncronas e bloqueantes adotado pelos sistemas de middleware orientados a objeto mais populares.*

1. Introdução

Os sistemas paralelos e distribuídos estão em alta e vêm se tornando, cada vez mais, acessíveis ao público em geral. A proliferação do uso de termos como *computer farms*, *Web services, grid computing* e *multicore processors* são indícios da força dessa tendência. É de se esperar, portanto, que desenvolvedores de todos os segmentos se envolvam, mais cedo ou mais tarde, na construção de um sistema paralelo ou distribuído. Sistemas distribuídos, embora amplamente difundidos e ainda em plena ascensão de demanda, continuam fazendo jus ao estigma de serem difíceis de construir. Esse estigma é justificado inclusive historicamente, já que os primeiros desenvolvedores de sistemas distribuídos dispunham apenas de mecanismos bastante primitivos de passagem de mensagens (por exemplo, *sockets*), que os obrigavam a lidar com uma porção substancial das idiossincrasias da comunicação em ambientes heterogêneos.

Seguindo a tendência de busca por decomposições mais adequadas (que pode ser observada em praticamente todos os domínios do desenvolvimento de software), foram criados, na carona da orientação a objetos, os sistemas de *middleware orientados a*

objeto (*middleware* OO). O *middleware* OO, em sua forma de uso mais difundida, estrutura o sistema distribuído como uma coleção de objetos que se comunicam por meio de chamadas síncronas e bloqueantes. A vantagem óbvia dessa estrutura é que o desenvolvedor passa a enxergar o sistema distribuído como um sistema centralizado, uniforme e orientado a objetos. Os sistemas construídos em cima do *middleware* OO são chamados *Sistemas de Objetos Distribuídos* (SODs).

Infelizmente, o *middleware* ajuda só até certo ponto. Embora o seu uso torne a construção de sistemas distribuídos mais simples, depurar esses sistemas continua uma tarefa difícil. Poderíamos até dizer que o *middleware* torna o processo de depuração mais difícil, já que o código gerado automaticamente pelo arcabouço – além do código do próprio arcabouço – passam a participar da execução do sistema. O uso dos tradicionais depuradores simbólicos, tais como o GDB [4] ou o depurador Java do Eclipse/JDT [14], torna-se mais difícil, já que esse código extra faz pouco ou nenhum sentido para o desenvolvedor da aplicação distribuída.

Há ainda outras questões que dificultam a depuração de sistemas distribuídos e que se estendem além do uso ou não-uso do *middleware*. O processo de depuração envolve, via de regra, navegar por uma reconstrução aproximada da execução do sistema – em geral modelada como uma sucessão de estados – em busca de violações de premissas capazes de produzir o comportamento errôneo observado. Uma das hipóteses fundamentais é que essa reconstrução aproximada da execução do sistema: (1) contenha informações suficientes para que seja possível identificar uma eventual violação de premissas e (2) permita identificar as relações de causa e efeito entre os estados.

O item (1) relaciona-se ao nível de abstração em que são apresentados os estados do sistema ao observador. O item (2) diz respeito à consistência das informações apresentadas. Note que uma falha no item (2) inviabiliza por completo o processo de depuração, já que se torna impossível determinar quais causas levam a quais efeitos (ou estado(s) errôneo(s), no caso).

Em sistemas distribuídos, o item (2) é particularmente importante. Isso porque as cadeias de causa e efeito podem ser distribuídas. Sem relógios globalmente sincronizados ou algum outro mecanismo de captura causal, fica muito difícil identificar qual a ordem relativa dos estados e, por conseqüência, qual a origem dos comportamentos errôneos observados [13]. Essa questão em particular é compartilhada por qualquer sistema distribuído e não só por SODs.

2. Depuradores Simbólicos e Threads Distribuídos

Os depuradores simbólicos são velhos conhecidos dos desenvolvedores de software. Uma das razões do sucesso desse tipo de ferramenta é que elas viabilizam uma forma de visualização da execução que fica dentro daquilo que é esperado pelo programador. O programador escreve um trecho de código com algo em mente, o depurador simbólico mostra a execução daquele trecho de código de uma maneira compatível.

É precisamente essa característica que a introdução de *middleware* orientado a objetos como CORBA [11], Java/RMI [17] ou o .NET Remoting [10], por exemplo, destrói. O desenvolvedor de um SOD codifica sob a ilusão de que o sistema distribuído é na verdade um sistema centralizado e *multithreaded* – e é justamente isso que torna o usuário do *middleware* mais produtivo. Esse mesmo *middleware*, no entanto, é incapaz

de manter suas abstrações aos olhos de um depurador simbólico, simplesmente porque os depuradores simbólicos estão presos a abstrações mais primitivas.

Embora um depurador simbólico não seja a única forma de abordar o problema da depuração de sistemas distribuídos, dois fatores nos levaram a optar, ao menos num primeiro momento, por uma abordagem que seguisse essa linha. O primeiro deles é a familiaridade de grande parte dos programadores com esse tipo de ferramenta. O segundo é o apelo a uma forma de visualização intuitiva, baseada no próprio código da aplicação.

Dito isso, o grande salto de abstração que separa os depuradores simbólicos tradicionais do *middleware* é a introdução, pelo *middleware*, do conceito de *thread distribuído*. Um *thread distribuído* é, em sua essência, um *thread* capaz de cortar vários nodos (Figura 1). Trata-se de uma conseqüência natural do mecanismo de chamadas síncronas e bloqueantes empregado pelo *middleware*. Mais precisamente:

Definição 1: Um *thread distribuído* T é um conjunto ordenado $T = \{t_1,...,t_n\}$ de *threads locais* em que t_i, $i \in \mathbb{N}$ e $1 \leq i < n$, é um *thread* que faz uma requisição remota e t_{i+1} é o *thread* que trata essa requisição remota no servidor. Dizemos que t_1 é a **base** do *thread distribuído* e que t_n é a sua **cabeça**.

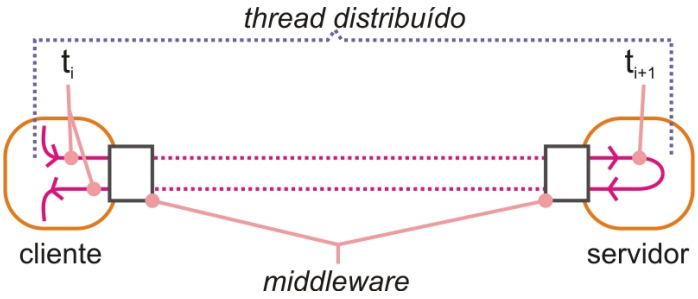

Figura 1. Thread distribuído

Threads locais são nada mais do que os convencionais *threads* de nível de aplicação. Dizemos que os *threads locais* $t_1,...,t_n$ **participam** do *thread distribuído* T, ou ainda que $t_i \in T$ (para $i \in \mathbb{N}$ e $1 \leq i \leq n$). Agora resta discutir de quais formas um *thread local* pode passar a fazer parte de um *thread distribuído*. As regras, que decorrem da **Definição 1,** são as seguintes:

1. Se um *thread local* t_1, que não faz parte de nenhum *thread distribuído*, faz uma chamada a um objeto remoto e essa chamada é tratada por um *thread* t_2 no lado do servidor, então t_1 e t_2 passam a fazer parte do mesmo *thread distribuído*; isto é, passa a existir um *thread distribuído* T tal que $T = \{t_1, t_2\}$;
2. se um *thread local* $t_i \in T$, $T = \{t_1,...,t_i\}$, faz uma chamada a um objeto remoto e essa chamada é tratada por um *thread* t_{i+1} no lado do servidor, então t_{i+1} passa a ser parte de *T;* isto é, $T = \{t_1,...,t_i,t_{i+1}\}$;

3. se um *thread local* $t_i \in T$, $T = \{t_1, ..., t_i\}$, termina de tratar, no lado do servidor, uma requisição iniciada pelo thread $t_{i-1} \in T$, então t_i deixa de ser parte de T; isto é, $T = \{t_1, ..., t_i, t_{i-1}\}$.

Além disso, um *thread local* t_i só pode participar, num dado instante, de um único *thread distribuído*.

O que essas regras na verdade formalizam é que o *thread distribuído* é uma pilha de *threads locais*, em que novos *threads* são empilhados conforme novas requisições tomam parte na cadeia de chamadas e desempilhados conforme o *thread* no topo da pilha termina de cumprir uma requisição iniciada pelo *thread* que o antecede.

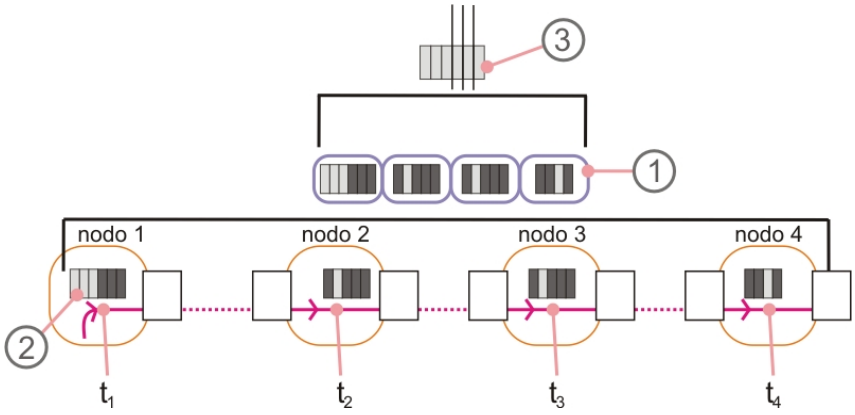

Figura 2. Cadeia de chamadas e pilha virtual

A Figura 2 ilustra a idéia. Um *thread distribuído* é composto por uma pilha de *threads locais* (3). Cada *thread local* conta com uma pilha própria – a pilha de chamadas (2) – composta por quadros resultantes de chamadas ao código do *middleware* (cinza escuro) e de quadros resultantes de chamadas ao código da aplicação (cinza claro). Um depurador que pretenda rastrear *threads distribuídos* deve ser capaz de determinar, a cada instante, quais *threads* locais participam em quais *threads* distribuídos. Além disso, o depurador deve determinar quais quadros, nas pilhas de chamadas dos *threads locais*, correspondem a chamadas ao código da aplicação. De posse dessas informações, o depurador torna-se capaz de montar a *pilha virtual* do *thread distribuído* (1), composta apenas pelos quadros de interesse – i.e., pelos quadros que resultam de chamadas a código da aplicação. Essa pilha corresponde à pilha de chamadas que de fato existiria caso o sistema fosse centralizado e o *middleware* não existisse.

O restante deste artigo discute a arquitetura e a implementação de um depurador simbólico para sistemas de objetos distribuídos – o *Global Online Debugger* ou GOD – que aplica as idéias discutidas até o presente momento. O depurador construído permite que o usuário interaja e manipule os *threads distribuídos* de um SOD da mesma forma que o depurador do Eclipse/JDT, por exemplo, permite que um usuário depurando um sistema Java manipule os *threads* em execução na sua aplicação. Este trabalho representa uma continuação do trabalho preliminar descrito em [2] e um aprofundamento do trabalho descrito em [3]. O código-fonte da implementação descrita neste artigo – que já superou 20 mil linhas de código – pode ser obtido no sítio da ferramenta: http://incubadora.fapesp.br/projects/god.

3. Arquitetura

A implementação atual do depurador é baseada em uma arquitetura centralizada, mostrada na Figura 3. Essa arquitetura é semelhante à adotada em outros depuradores distribuídos como, por exemplo, o depurador distribuído da IBM [5].

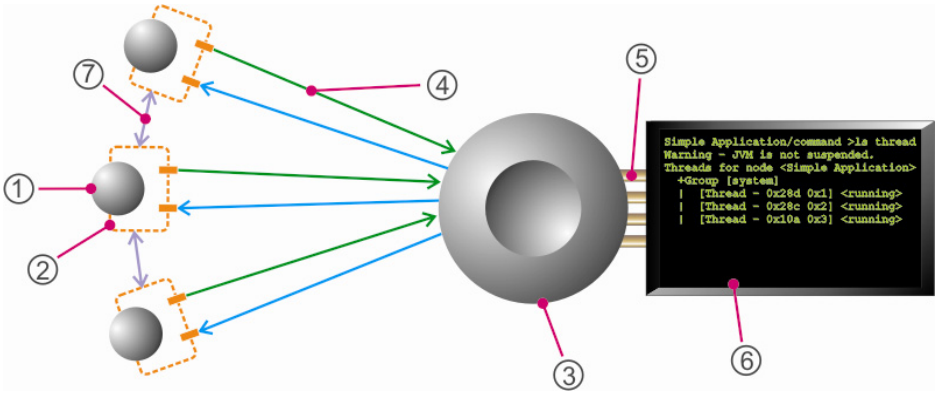

Figura 3. Arquitetura do depurador em linhas gerais

As pequenas esferas em cinza (1) representam os processos que compõem o sistema distribuído. A cada um desses processos encontra-se ligado um *agente local* (2). Os *agentes locais* atuam em nome de um *agente global* (3) interferindo na execução dos processos conforme o necessário (suspendendo *threads* e adicionando *breakpoints,* por exemplo) e coletando informações relevantes à reconstrução do estado global do sistema distribuído. Essas informações são repassadas ao *agente global* por meio de um ou mais protocolos de comunicação (4). O *agente global* correlaciona as informações recebidas, reconstrói uma aproximação da execução do sistema distribuído e disponibiliza essa aproximação por meio de uma API (5), explorada atualmente por uma interface textual simples (6). É possível ainda que os *agentes locais* troquem informações entre si (7) caso seja necessário, por exemplo, executar algum algoritmo distribuído que não dependa da intervenção do agente global.

Além de atuar como observador do sistema distribuído, o agente global disponibiliza um ponto de controle unificado dos processos, por meio da *infra-estrutura de controle remoto de processos*. O principal objetivo dessa infra-estrutura é tornar o gerenciamento e a interação com os processos que compõem o sistema distribuído tão simples quanto no caso não-distribuído, facilitando a montagem de *cenários de depuração*.

Um cenário de depuração representa, no caso mais simples, um conjunto de processos que devem ser iniciados de acordo com certas restrições de ordem – por exemplo, um servidor de banco de dados deve ser iniciado antes dos servidores que dele fazem uso. A responsabilidade por iniciar e coordenar os diversos processos que compõem o sistema distribuído recai, via de regra, sobre o desenvolvedor, que é obrigado a conectar-se individualmente a cada uma das máquinas participantes e disparar os processos adequados.

Além disso, é possível que alguns dos processos do sistema distribuído requeiram algum tipo de interação que pressupõe a presença física do usuário. Essa hipótese, embora razoável em um sistema em produção, pode dificultar substancialmente o processo de depuração, especialmente se houver passos que envolvam interações com interfaces gráficas ou consoles. A infra-estrutura de controle

remoto de processos, composta pelo *módulo de controle de processos* e pelos *servidores de controle de processos*, é mostrada na Figura 4.

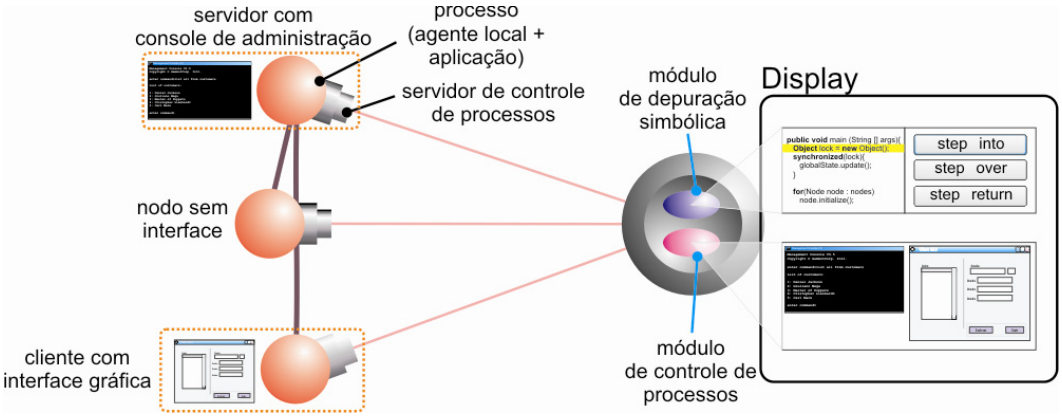

Figura 4. Controle de processos

Em seu uso mais simples, a infra-estrutura de controle remoto de processos permite que o usuário lance, termine e interaja com as interfaces de processos em máquinas remotas. A Figura 4 mostra uma aplicação distribuída típica, composta por três processos (três esferas menores). Essa aplicação conta com dois processos que potencialmente requerem algum nível de interação: um cliente, que disponibiliza uma interface gráfica, e um servidor, que disponibiliza um console de administração. O clique de um botão na interface gráfica ou a entrada de algum comando no console de administração podem, por exemplo, fazer parte de um cenário de depuração. Nesse caso, o desenvolvedor pode se beneficiar da infra-estrutura de controle remoto de processos para (1) lançar todos os processos do sistema distribuído e (2) comunicar-se com suas interfaces remotamente, importando-as por meio da infra-estrutura em sua área de trabalho (Display na Figura 4). Vale notar que o mecanismo de importação/exportação de interfaces gráficas depende de software externo, como por exemplo um conjunto de clientes e servidores de alguma implementação do X [16] instalados.

Conforme discutimos anteriormente, a montagem de um cenário de depuração pode envolver a especificação de uma ordem de lançamento dos diversos processos. O problema é que a granulosidade "ordem de lançamento" pode ser muito grossa para alguns casos. Em geral, se um processo A depende de um processo B para funcionar, é necessário que B atinja um estado específico antes que A possa ser lançado. Para reduzir um pouco a granularidade das restrições, a infra-estrutura de controle remoto de processos pode trabalhar em cooperação com o depurador simbólico, disponibilizando um mecanismo um pouco menos rudimentar. O resultado são os *scripts de lançamento*, que são arquivos-texto contendo expressões como:

```
launch <process 2> when: <process 1> hits server.Server.main():50
```

Essa expressão indica que o processo "*process 2*" deve ser lançado assim que o primeiro *thread* de controle do processo "*process 1*" atingir a linha 50 do método *server.Server.main()*. A especificação de predicados mais complexos e úteis – envolvendo o estado de variáveis, por exemplo – está na lista de trabalhos futuros. Uma hipótese implícita na discussão desenvolvida até o presente momento é que o depurador (ou o desenvolvedor) é sempre responsável pelo lançamento dos processos do sistema distribuído. Isso pode nem sempre ser verdade – i.e., um processo da própria aplicação pode lançar novos processos, que também pertencem à aplicação. Dizemos que tais

aplicações apresentam um comportamento *ativo*, opondo-se ao comportamento *passivo* dos processos que dependem do depurador ou do usuário para entrarem em ação. O suporte a aplicações ativas ainda é limitado, essencialmente porque os processos lançados pela própria aplicação distribuída não passam, em geral, pelo processo de instrumentação de código requerido pelo depurador (algo que a infra-estrutura de controle remoto de processos sempre garante que acontece).

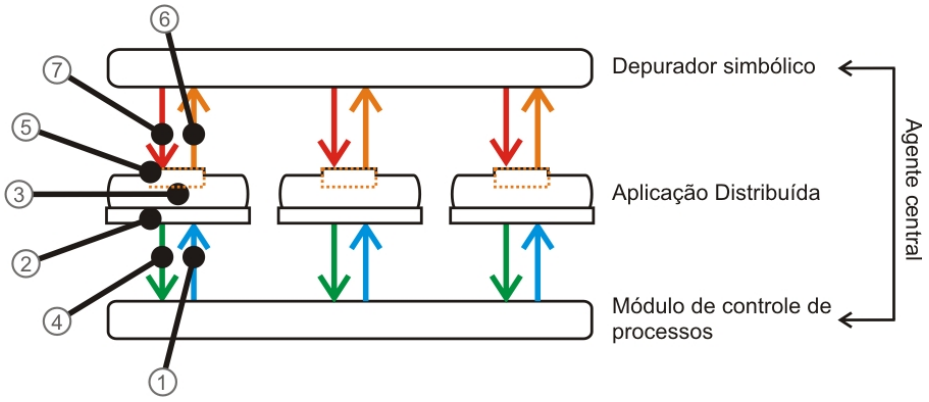

Figura 5. Uma outra perspectiva da arquitetura

A Figura 5 mostra uma visão mais refinada da arquitetura do depurador. O módulo de controle de processos comunica-se (1) com os servidores de controle de processos (2), comandando o lançamento e o término de processos no sistema distribuído (3). Os servidores de controle (2), por sua vez, devolvem ao módulo central tanto informações de controle quanto dados (4). As informações de controle compreendem, por exemplo, notificações de término prematuro de processos e pacotes de *keep alive*. Os dados, por sua vez, são usados pela infra-estrutura para a reconstrução local das interfaces dos processos remotos.

O depurador simbólico atua na execução dos processos do sistema distribuído. Os *agentes locais* (5) são instanciados pelos servidores de controle de processos e em seguida acoplados aos processos do sistema distribuído. Ao ser associado a um processo, um agente local notifica o depurador simbólico (6) e dá início ao processo de depuração simbólica remota: o depurador simbólico (residente no agente global) passa a enviar comandos (7) – tais como ordens de inserção de *breakpoints* e pedidos de inspeção de variáveis – e receber eventos (6) – tais como notificações de exceções não tratadas ou de que algum *breakpoint* foi atingido.

4. Implementação

Esta Seção procura unificar as informações apresentadas nas Seções 1, 2 e 3, introduzindo o mecanismo de rastreio de *threads distribuídos* e uma implementação desse mecanismo para aplicações escritas em Java e que usam *middleware* CORBA.

Conforme apontamos na Seção 1, os depuradores simbólicos não são via de regra capazes de identificar muitos dos aspectos da distribuição do sistema que nos são de interesse. Em particular, esses depuradores simbólicos não são capazes de rastrear *threads distribuídos*, já que essa é uma abstração introduzida pelo *middleware* como resultado do mecanismo de chamadas síncronas e bloqueantes. Isso significa que, para uma boa parte das linguagens de programação mais populares, apenas parte das informações que nos são de interesse podem ser obtidas por meio de uma plataforma ou ferramenta de depuração simbólica pré-existente. O restante das informações devem ser

extraídas diretamente da execução, por meio de instrumentação de código ou da alteração do próprio ambiente de execução (no caso de linguagens interpretadas). Antes de entrar nos detalhes de implementação, no entanto, vamos discutir quais informações devem ser extraídas e quando.

4.1 Identificação e rastreio das requisições

O primeiro passo para o rastreio de *threads distribuídos* é a sua demarcação. Os *threads distribuídos* devem ser univocamente identificáveis em escopo global, para que não confundam-se uns com os outros. Isso envolve, na nossa solução (fortemente inspirada em [8]), a atribuição de identificadores únicos a cada um dos *thread locais* do sistema distribuído. Esses identificadores são então propagados pelas cadeias de chamadas, identificando-as pelo *thread local* da base (Figura 6).

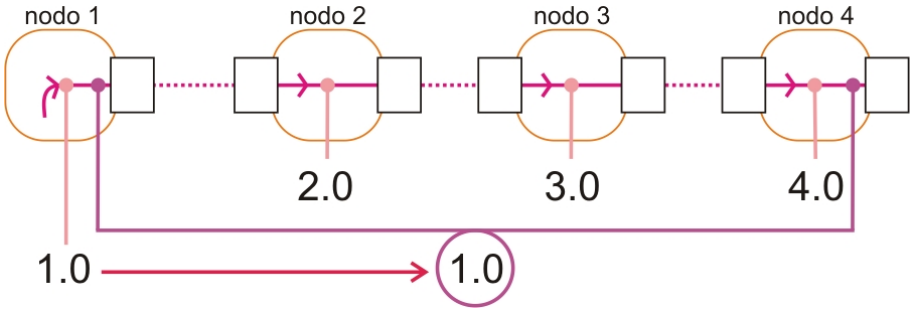

Figura 6. Propagação de identificadores

Na Figura 6, a requisição iniciada no *nodo 1* dá origem a um *thread distribuído* de identificador 1.0 – o mesmo identificador do *thread local* base. Os *threads locais* subseqüentes (2.0, 3.0 e 4.0) passam a fazer parte do mesmo *thread distribuído* (de identificador 1.0) conforme tomam parte no tratamento da requisição. Com relação ao rastreio de fato, basta notar que a única parte ativa do *thread distribuído* é a sua cabeça – todos os outros *threads locais* estão suspensos aguardando resposta. Isso implica que rastrear um *thread distribuído* resume-se a rastrear a localização da cabeça. Baseados nessa observação, desenvolvemos um protocolo simples de dois estágios em que são emitidas notificações ao agente global sempre que um *thread* atinge um dos dois pontos principais de cada requisição, mostrados na Figura 7.

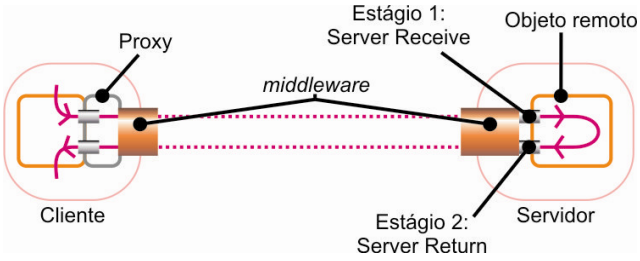

Figura 7. Protocolo de dois estágios

Os dois estágios – *Server Receive* e *Server Return* – identificam com precisão os pontos em que o *thread distribuído* adentra e abandona, respectivamente, o código da aplicação no lado do servidor. Isso significa que, se forem enviadas notificações ao agente global sempre que esses dois pontos forem atingidos, basta que o agente global ordene os eventos (e que haja um meio de descobrirmos se alguma notificação se perdeu) e teremos uma reconstrução fiel do trajeto do *thread distribuído*.

4.2 A questão da interatividade

O problema é que uma mera reconstrução do trajeto dos *threads distribuídos* não é suficiente para garantir interatividade. Em particular, a implementação da operação de suspensão em *threads distribuídos* demanda algum cuidado. Isso porque, para suspender um *thread distribuído*, é necessário suspender o *thread* da cabeça. Isso se torna impossível de fazer em instantes arbitrários se não soubermos, em instantes também arbitrários, qual o *thread* da cabeça.

Esse problema se generaliza na questão da interatividade: é impossível manter a interatividade se o estado real do sistema sob observação puder desviar-se arbitrariamente do estado observado. Para confirmar a veracidade dessa afirmação basta notar que, no pior caso, a aplicação termina antes que o usuário possa visualizar o primeiro corte na execução do sistema. Para solucionar esse problema, definimos alguns pontos em que o sistema distribuído é forçado a sincronizar a parte relevante do seu estado com o agente global. A escolha dos pontos de sincronização segue da discussão apresentada na Seção 4.1: os dois estágios do protocolo de notificação (*Server Receive* e *Server Return*) são os melhores candidatos já que, dessa forma, o agente global sempre sabe qual a cabeça do *thread distribuído*. Na implementação atual do depurador, essas sincronizações são feitas por meio de notificações síncronas. Uma conseqüência muito importante do uso de notificações síncronas é que o uso de relógios lógicos para a ordenação de eventos torna-se desnecessário, já que se um evento A *acontece antes* de B então a notificação de que A está acontecendo deve retornar antes que B possa acontecer; isto é, não há possibilidade de violações causais.

Com relação ao *overhead* do mecanismo, a expectativa é que ele não afete de forma significativa o desempenho do processo de depuração simbólica em sistemas de pequeno/médio porte, já que a plataforma de depuração simbólica que estudamos em mais profundidade (a plataforma de depuração Java ou JPDA [18]) faz uso intensivo de notificações síncronas. É de praxe que a JPDA notifique de forma síncrona o depurador, por exemplo, sempre que um *thread local* é criado. Embora esse *overhead* deva ser medido antes que possamos fazer afirmações mais conclusivas, podemos dizer que a depuração simbólica perturba, de maneira geral, a execução do sistema distribuído. Essa perturbação deve crescer com a elevação de carga no agente global, já que as notificações síncronas devem demorar mais tempo para serem processadas, gerando pausas ainda maiores na execução. Isso coloca a escalabilidade da interação sob suspeita.

4.3 JPDA, CORBA e agentes locais revisitados

O *Java Development Kit* (JDK) e o *Java Runtime Environment* (versões 1.3.1 e posteriores) disponibilizam uma plataforma de depuração simbólica completa, a *Java Platform Debug Architecture* (JPDA [18]) que, entre outras facilidades, torna a construção de depuradores simbólicos remotos para Java mais simples. A implementação atual do depurador distribuído utiliza a JPDA como ferramenta de depuração simbólica para processos escritos em Java, complementando a plataforma por meio de instrumentação de *bytecodes*. O agente local Java, composto pela JPDA e pelo código inserido no ambiente de execução da aplicação (interceptadores + biblioteca de depuração), é mostrado em maior detalhe na Figura 8 (a). Note que o agente local Java utiliza, na verdade, dois protocolos diferentes. O primeiro deles, o *Java Debug Wire Protocol* (JDWP), é implementado pelo fabricante da máquina virtual, sendo específico

da plataforma de depuração simbólica do ambiente de execução Java. O outro, o *Distributed Debugging Wire Protocol* (DDWP), foi implementado por nós. O DDWP é independente de linguagem e foi desenvolvido especialmente para a comunicação de dados relacionados a depuração distribuída. A implementação do DDWP está contida na *biblioteca de depuração* (Figuras 8 (a) e (b)).

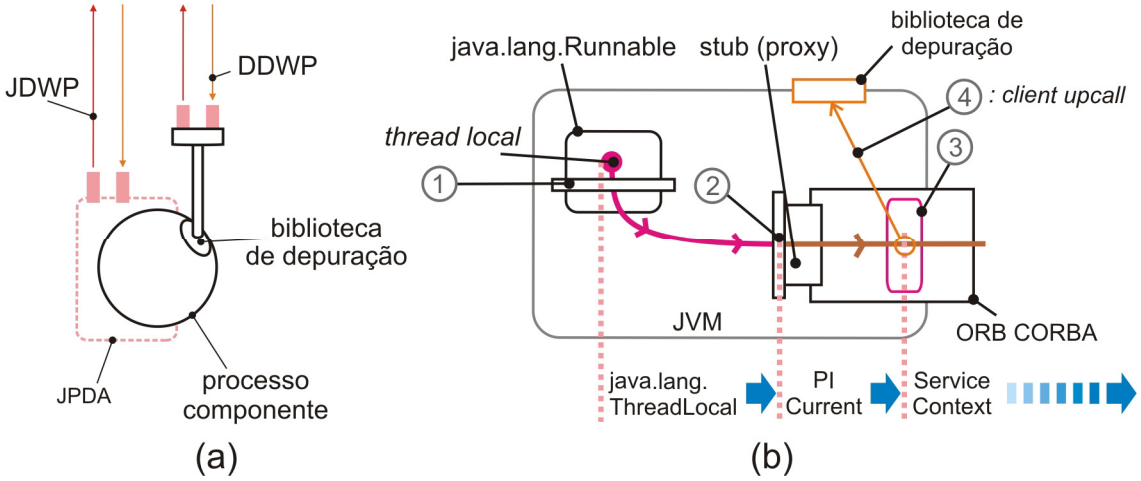

Figura 8. (a) Agente local e (b) mecanismo de rastreio do lado do cliente

A Figura 8 (b) mostra o mecanismo de rastreio de *threads distribuídos* do lado do cliente em mais detalhe. A atribuição de identificadores aos *threads locais* é feita através da inserção de interceptadores (1), via manipulação de *bytecodes*, em todas as classes que implementam a interface `java.lang.Runnable`. O identificador atribuído pelo interceptador ao *thread local* é guardado numa variável do tipo `java.lang.ThreadLocal`. Ao realizar uma chamada remota, o *thread* passa por um segundo interceptador (2), instalado nos *stubs* CORBA, que extrai o identificador do *thread local* (atribuído pelo primeiro interceptador) e o re-insere, juntamente com o tamanho da pilha de chamada e outras informações, num *slot* pré-alocado pelo depurador no *PICurrent* da requisição (para maiores informações a respeito do *PICurrent* o leitor deve referir-se a [11]). Isso faz com que essas informações tornem-se acessíveis a um interceptador CORBA (3), instalado pelo depurador no *middleware*. O interceptador CORBA, ao ser atingido, notifica o agente global de que há uma requisição em curso (4). Em seguida, o interceptador insere o identificador do *thread distribuído* num *service context.* Esses dados são então enviados, juntamente com os dados da requisição remota, para o servidor.

O leitor atento deve notar que a descrição geral do protocolo de notificação da Seção 4.1 prevê apenas dois estágios. O problema é que certas informações, como por exemplo o tamanho da pilha de chamada no momento em que o *thread local* entra no *stub* CORBA – essencial para a filtragem dos quadros da pilha que não fazem parte do código da aplicação – são muito difíceis de serem recuperadas de forma razoável pela JPDA. Nós não podemos recuperar essas informações de forma confiável por fora da JPDA, já que isso envolveria fazer uma requisição a um processo possivelmente suspenso e sob modificação pelo usuário. A solução vem sob a forma de uma notificação extra – um estágio que antecede o *server receive*, batizado de *client upcall*, em que as informações necessárias são enviadas (sincronamente) ao agente global. A Figura 9 complementa a Figura 8 (b), mostrando o mecanismo de rastreio de *threads distribuídos* do lado do servidor. Seu funcionamento é bastante similar ao do

mecanismo do lado do cliente: a requisição CORBA proveniente da rede atinge primeiramente um interceptador CORBA (1), instalado pelo depurador.

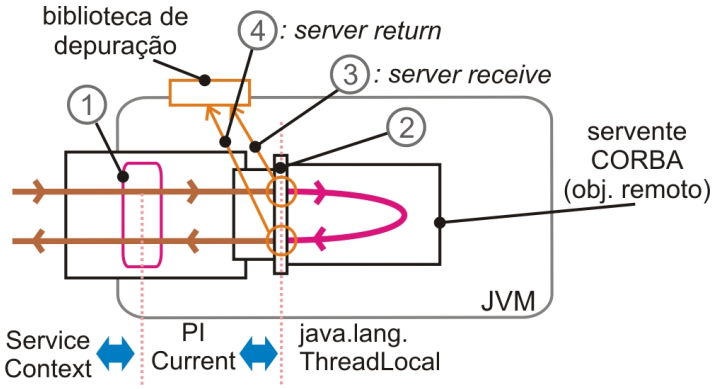

Figura 9. Mecanismo de rastreio do lado do servidor

Esse interceptador extrai as informações (passadas pelo cliente) do *service context* e determina o identificador do *thread distribuído*, inserindo-o no *PICurrent* da requisição. Eventualmente a requisição atinge o servente CORBA, onde um interceptador instalado via manipulação de *bytecodes* (2) associa o identificador do *thread distribuído,* que veio pela rede, ao *thread local* que trata a requisição. O *thread local* passa a ter dois identificadores, um dele próprio e outro do *thread distribuído* do qual ele faz parte. Note que se eventualmente esse *thread local* fizer uma segunda chamada remota, é o identificador do *thread distribuído* que será propagado. O interceptador (2) notifica então o agente global que uma requisição remota está prestes a ser tratada (3) e que um novo *thread local* deve ser incorporado ao *thread distribuído*, repassando ao agente global tanto o identificador do *thread local* quanto o identificador do *thread distribuído*. Ao deixar o código do servente CORBA, o fluxo da requisição é novamente interceptado (2), gerando outra notificação para o agente global (4), que atualiza novamente o estado do *thread distribuído*. O mecanismo de rastreio atua em conjunto com o depurador simbólico. Em particular, o depurador simbólico faz uso do mecanismo de rastreio e suas notificações para "pular" o código do *middleware* quando um usuário, no modo de execução passo-a-passo, requisita, por exemplo, um *step into* em um *stub* CORBA. Algumas informações extras a respeito dessa porção do mecanismo podem ser encontradas em [2].

4.4 Reflexão *ad-hoc* em Java

Uma das dificuldades enfrentadas durante o desenvolvimento do depurador relaciona-se à falta de uma arquitetura reflexiva apropriada em Java. Seria muito difícil, usando apenas a JPDA, escrever uma aplicação Java capaz de, por exemplo, comunicar a um depurador simbólico ligado a ela que um certo *thread* local X acaba de passar por um certo interceptador Y (lembre-se que o protocolo de rastreio de *threads* distribuídos requer isso). Seria difícil porque a aplicação, sendo parte do nível base, desconhece completamente as representações para *threads* adotadas pelo meta nível (depurador) que, na JPDA, são distintas para aplicações e depuradores. O processo de desenvolvimento do depurador envolve, portanto, uma forma de estabelecer uma noção comum entre depurador e depurado a respeito de quais referências do nível base correspondem a quais referências do meta nível (e vice-versa). Nossa solução consiste num protocolo de registro, mostrado na Figura 10(a). O interceptador instalado em todas as classes que implementam `java.lang.Runnable`, logo após atribuir um identificador

numérico único ao *thread* local que passa por ele, desvia esse *thread* para dentro de um método especial, onde foi instalado um *breakpoint*.

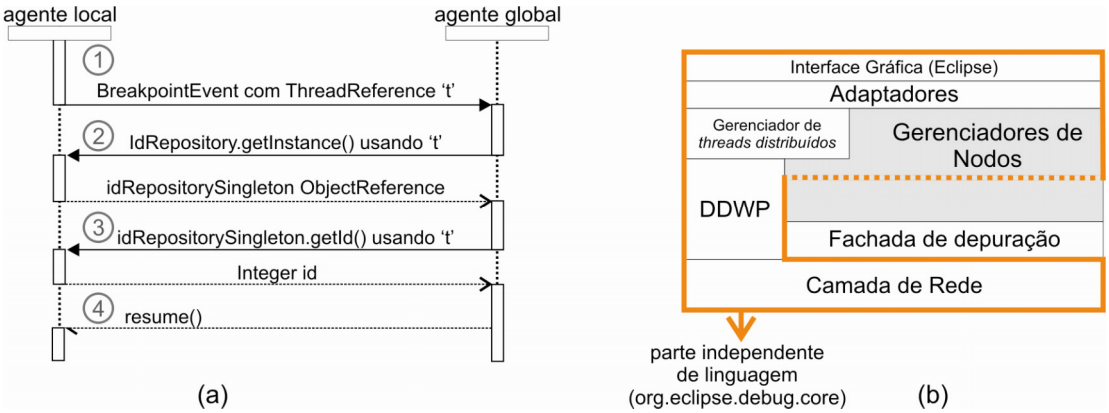

Figura 10. (a) Protocolo de identificação dos threads e (b) arquitetura parcial do agente global

Quando o *thread* atinge o *breakpoint*, um evento é gerado pela JPDA **(1)**. Nesse evento segue anexada uma referência ao *thread* suspenso. Essa referência é compreensível pelo meta nível, mas não pelo nível base. Ao receber o evento, o agente global utiliza a referência anexada e o mecanismo de avaliação de expressões remotas fornecido pela JPDA para extrair o valor do identificador numérico do *thread* local, atribuído pelo interceptador, diretamente da aplicação (passos **(2)** e **(3)**) mapeando assim o identificador numérico (referência compreensível pelo nível base) numa referência JPDA (referência compreensível pelo meta nível). Esse mapeamento equivale ao transporte de uma representação do meta nível para o nível base (e nós o utilizamos dessa forma), e constitui um mecanismo *ad hoc*, restrito, de reflexão distribuída. Graças a esse mecanismo nos tornamos capazes de, por exemplo, suspender threads distribuídos, já que é o mecanismo quem nos permite identificar a quais referências JPDA correspondem os identificadores numéricos enviados pelos vários agentes locais.

4.5 Adicionando outras linguagens e plataformas de *middleware*

O depurador possui uma estrutura modular (Figura 10 (b)). Embora a implementação do agente global ainda não reflita fielmente essa arquitetura, vamos discutir a adição de novas linguagens nos baseando nela. Note que os únicos componentes que devem de fato serem implementados para a adição de uma nova linguagem são os *gerenciadores de nodos*. Os *gerenciadores de nodos* cumprem a dois propósitos: atuam como pontos de acesso a versões reificadas de certos elementos dos ambientes de execução dos nodos – essencialmente seus *threads* – e atuam como *proxies* dos depuradores remotos, expondo interfaces independentes de linguagem para a realização de requisições como pedidos de inserção de *breakpoints* e acesso a canais de eventos. Os elementos reificados acessíveis por meio dos gerenciadores de nodos também expõem interfaces independentes de linguagem. Essas interfaces são dadas, majoritariamente, pelo meta-modelo de depuração do Eclipse. Em linguagens como Java, pode ser que já exista uma API equivalente à dos nossos gerenciadores de nodo já disponível (*Fachadas de depuração* na Figura 10 (b)). Nesses casos, o gerenciador de nodos pode atuar como uma camada de adaptação entre meta-modelos, como acontece no depurador do Eclipse/JDT e no nosso depurador. Ainda que a implementação dos gerenciadores de nodos (e dos elementos associados) se traduza num trabalho significativo, a parte mais

central do depurador (gerenciador de threads distribuídos e interface com o usuário) encontra-se acoplada apenas às interfaces do meta-modelo de depuração do Eclipse, reconhecidamente independente de linguagem. Mais do que independente de linguagem, já há uma porção de depuradores adaptados a esse modelo. A idéia é que possamos reutilizar uma porção significativa desses depuradores, facilitando a integração de novas linguagens. Além da implementação dos componentes no agente global, no entanto, há a questão dos agentes locais, possivelmente mais complexa. Suspeitamos, em função dos problemas encontrados durante a implementação do agente local Java, que, quanto menos reflexiva a linguagem, mais difícil a implementação dos meios de coleta das informações requeridas para a operação do DDWP. A técnica de rastreio de *threads* distribuídos se apóia em conceitos presentes em quase qualquer sistema baseado em chamadas síncronas e bloqueantes, exigindo muito pouco do *middleware* subjacente. Contanto que clientes acessem objetos remotos via *proxies* e que objetos remotos correspondam a objetos de fato, a técnica pode ser aplicada. Um mecanismo de passagem de metadados como o disponibilizado em CORBA com seus *portable interceptors* e *service contexts* também é de grande valia, mas não se trata de um requisito obrigatório (há soluções alternativas [8]). Isso indica que a adaptação da técnica de rastreio para outras plataformas de *middleware* deve ser bastante direta.

5. Algumas aplicações concretas e estado atual da implementação

Além de facilitarem substancialmente a navegação pela execução da aplicação distribuída e de darem ao desenvolvedor uma visão da execução mais próxima daquilo que ele espera, as informações trazidas pelos *threads distribuídos* são úteis na medida em que expõem, ainda que de forma momentânea, relações causais antes não explícitas. A infra-estrutura que implementamos explora essas informações, sendo capaz de detectar se:

- Há requisições em espera circular, viabilizando a detecção de *deadlocks* distribuídos;
- há requisições cuja pilha de *threads* é maior que um certo número pré-estabelecido, viabilizando a detecção de "estouros de pilha" distribuídos (laço recursivo infinito).

Além disso, torna-se possível examinar exceções de uma forma mais interessante. Uma situação anormal que se manifesta por meio de uma exceção lançada num servente CORBA, por exemplo, pode gerar um *stacktrace* de uma pilha virtual, mostrando toda a cadeia de chamadas entre o primeiro cliente e o objeto remoto, ao invés de simplesmente mostrar uma pilha local, que é de pouco ou nenhum uso. Algumas outras possibilidades, que vão além da depuração simbólica (e que ainda não foram exploradas/implementadas), incluiriam a construção de grafos de chamadas ou de eventos, já que todas as informações necessárias para a construção dessas representações são capturadas em tempo de execução.

Com relação ao estado atual da implementação, a versão atual conta com as infra-estruturas de controle remoto de processos (em integração com o cliente GNU/SSH ou RLogin) e depuração simbólica descritas, sendo capaz de controlar processos remotamente e de rastrear e interagir com *threads* distribuídos. A implementação atual do agente global faz uso do meta-modelo independente de linguagem do Eclipse, mas a interface de usuário que dá acesso ao agente global é ainda baseada em texto (não integrada, portanto, ao ambiente gráfico do Eclipse). A camada nomeada "adaptadores" na Figura 10 (b), ainda não implementada por completo, é a

camada responsável pela integração dos gerenciadores de nodos à interface gráfica de fato. A implementação atual funciona corretamente em ambientes Java/CORBA. Os detectores de *deadlocks* distribuídos (que envolvem monitores Java) e estouros de pilha distribuídos já foram implementados.

7. Trabalhos relacionados

O *Object Level Trace* (OLT [6]) é uma extensão do depurador distribuído da IBM. A ferramenta é capaz de rastrear a execução de sistemas distribuídos que fazem uso de *middleware*, incluindo aplicações J2EE e aplicações distribuídas escritas em múltiplas linguagens. O OLT disponibiliza tanto formas de visualização de alto nível (essencialmente grafos de eventos) quanto de baixo nível (por meio do depurador distribuído), sendo capaz de produzir efeitos muito parecidos com os da nossa ferramenta (rastreio de chamadas remotas, ocultando o *middleware*). O OLT aparentemente, no entanto, não constrói uma pilha de chamadas unificada – embora tenha uma representação tão versátil quanto a nossa – e parece estar acoplado ao *middleware* proprietário da IBM. O OLT é atualmente distribuído como parte do *WebSphere*, o servidor de aplicação da IBM.

Em [9] é apresentado um ambiente integrado para depuração e teste de sistemas paralelos baseados em *middleware* PVM. O ambiente apresenta funcionalidades bastante sofisticadas, dentre as quais encontram-se um maquinário para a especificação de fluxos de execução (que generaliza o nosso mecanismo de lançamento e coordenação de processos), um sistema de visualização de alto nível (que usa uma representação similar à de grafos de eventos) e uma ferramenta que viabiliza a re-execução controlada de aplicações paralelas. A interação com as aplicações do sistema distribuído é mediada por um depurador simbólico distribuído, o DDBG, que apresenta uma arquitetura centralizada bastante semelhante à do nosso depurador.

A infra-estrutura apresentada em [7] procura generalizar o trabalho desenvolvido em [9], especificando um arcabouço para depuração paralela e distribuída que é independente de linguagem ou *middleware*. O resultado é uma arquitetura baseada em "*drivers*", que se aproxima, ao menos conceitualmente, da arquitetura do nosso agente global, induzida pelo modelo do Eclipse. O escopo do trabalho é mais extenso (e menos específico), no entanto, na medida em que é especificada uma arquitetura de gerenciamento e coleta de informações baseada em serviços. O depurador simbólico, nesse caso, entra como apenas um dos provedores de serviço. Ambas infra-estruturas [7, 9] estão voltadas a sistemas baseados em passagem de mensagens e, nesse aspecto, diferem da nossa abordagem. Além disso, tanto [9] quanto [7] apresentam ferramentas com implementações restritas a *middleware* PVM e cujo código não se encontra, até onde pudemos descobrir, amplamente disponível (i.e., disponível na Internet).

O DPD [12] é um depurador distribuído também bastante sofisticado, desenvolvido para o ambiente REM [1]. O DPD permite a re-execução controlada de aplicações não-determinísticas entre intervalos de *checkpoints*. O DPD disponibiliza uma interface gráfica (baseada em grafos de eventos) e um depurador simbólico, sendo que é possível depurar trechos seqüenciais por meio do depurador simbólico e controlar a execução de trechos não-seqüenciais por meio da representação gráfica. A arquitetura do DPD é também similar à da nossa ferramenta. Infelizmente, no entanto, o DPD está acoplado ao obscuro ambiente REM e seu código também não se encontra amplamente disponível (novamente, até onde pudemos descobrir).

Quase todas as ferramentas apresentadas nesta Seção utilizam depuradores simbólicos como forma de visualização de trechos seqüenciais da execução, recorrendo a uma representação extra para a apresentação dos aspectos relacionados à distribuição (i.e., as mensagens). Nossa ferramenta, por outro lado, se aproveita das chamadas síncronas e bloqueantes para fazer a ponte entre os trechos seqüenciais nos vários nodos por meio dos *threads distribuídos*. Isso faz com que nosso depurador seja mais adequado para atuar como base num eventual ambiente de depuração voltado a sistemas de objetos distribuídos, nos moldes dos ambientes descritos em [9, 12].

8. Conclusão

A depuração de sistemas distribuídos é um problema de difícil abordagem. Nossa ferramenta procura tornar a observação de execuções distribuídas mais simples ao aproveitar-se de duas idéias muito bem estabelecidas no universo do desenvolvimento de software: a depuração simbólica e as chamadas a procedimentos remotos. Nesse aspecto, a ferramenta é de fato capaz de reduzir substancialmente a complexidade da execução observada, especialmente quando comparada aos depuradores simbólicos tradicionais.

Ficou claro, no entanto, que a execução de sistemas distribuídos de médio/grande porte continua muito complexa para o tipo de observação viabilizada pelo GOD em sua forma atual. Num certo sentido isso já era previsto, no entanto, já que a sensação trazida pelo uso do GOD num sistema de objetos distribuídos é muito semelhante à sensação trazida pelo uso do GDB ou do depurador do Eclipse/JDT num sistema *multithreaded* de médio/grande porte. Um outro problema fica por conta das distorções produzidas pela ferramenta na execução das aplicações distribuídas, que se agravam com as questões de escalabilidade. Ainda assim, consideramos que este trabalho represente um passo válido na direção de tornar mais compreensíveis as execuções distribuídas, em especial pelo apelo a idéias familiares a grande parte dos desenvolvedores de software que utilizam sistemas de *middleware* orientados a objeto e depuradores simbólicos.

9. Trabalhos futuros e em andamento

O depurador simbólico é apenas parte de um trabalho maior. Formas mais compactas e escaláveis de visualização da execução, um mecanismo de re-execução de aplicações não determinísticas e a implementação de algoritmos de captura de predicados globais integram o quadro de trabalhos futuros. No momento, estamos finalizando uma interface melhor para o protótipo, baseada na plataforma Eclipse.

Referências

[1] G.C. Shoja, "A distributed facility for load sharing and parallel processing among workstations," *The Journal of Systems and Software*, vol. 14, no. 3, pp. 163-172, 1991.

[2] G. Mega and F. Kon. "Debugging Distributed Object Applications with the Eclipse Platform," in *Proc. of the 2004 ACM OOPSLA/ETX Workshop*, 2004, pp 47-51.

[3] G.Mega e F. Kon "GOD: Um depurador simbólico para sistemas de objetos distribuídos," *12º Salão de Ferramentas do Simpósio Brasileiro de Engenharia*

de Software. [Online] Disponível: http://www.sbbd-sbes2005.ufu.br/arquivos/GOD.pdf

[4] GNU Foundation. The GNU Project Debugger. [Documento Online], [2005 Dez 13], Disponível: http://www.gnu.org/software/gdb/gdb.html

[5] IBM Corporation. Distributed Debugger: Overview. [Documento Online], [2005 Dez 13], Disponível em: http://publib.boulder.ibm.com/infocenter/iadthelp/topic/com.ibm.debug400.doc/concepts/cbcdbovr.htm

[6] IBM Corporation. WebSphere Application Server, Object Level Trace. [Online] Disponível: http://www-306.ibm.com/software/webservers/appserv/olt.html

[7] J. Cunha, J. Lourenço, J. Vieira, B. Moscão e D. Pereira, "A Framework to Support Parallel and Distributed Debugging." in *Proc. of the 1998 International Conference on High-Performance Computing and Networking (HPCN)*. Amsterdan: Springer-Verlag, pp. 707-717, 1998.

[8] J. Li. "Monitoring and Characterization of Component-Based Systems with Global Causality Capture." in *Proc. of the 23rd ICDCS*. Rhode Island, 2003, pp. 19-22.

[9] J. Lourenço et al. "An Integrated Testing and Debugging Environment for Parallel and Distributed Programs." in *Proc. of the 23rd EUROMICRO Conference*, Budapest: IEEE Computer Society Press, 1997, pp. 291-298.

[10] Microsoft Corporation. (2005) .NET Remoting Overview. [Documento Online], [2005 Dez 13], Disponível em: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconnetremotingoverview.asp

[11] Object Management Group. *The Common Object Request Broker : Architecture and Specification*, Revision 3.0.3, 2004.

[12] R. S. Side e G. C. Shoja, "A debugger for distributed programs," *Software: Practice & Experience*, vol. 24, no. 5, Maio, pp. 507-525, 1994.

[13] R. Schwarz e F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distributed Computing*, vol. 7, no 3, pp 149-174, 1994.

[14] Sítio na Web do Depurador do JDT/Eclipse. [2005 Dez 13]. http://www.eclipse.org/jdt/

[15] Sítio na Web do Eclipse. [13 de dezembro, 2005]. http://www.eclipse.org.

[16] Sítio na Web do XFree86. [13 de dezembro, 2005]. http://www.xfree86.org.

[17] Sun Microsystems. (2004). Java Remote Method Invocation Specification. [Documento Online], [2005 Dez 13], Disponível em: http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmiTOC.html.

[18] Sun Microsystems. The Java Platform Debug Architecture. [Online], [13 de dezembro, 2005], Disponível em: http://java.sun.com/products/jpda/index.jsp.