Relatório de Estudos

MAC 5701 - Tópicos em Ciência da Computação

Aluno: Giuliano Mega
Orientador: Fabio Kon

:: Índice

:: INTRODUÇÃO

Num cenário em que os sistemas de informação vêm tornando-se cada vez mais complexos, os depuradores entram como peça chave no auxílio à detecção, diagnóstico e correção de problemas notavelmente difíceis de solucionar por meio de outros métodos mais primitivos. Com a adoção de novos paradigmas, as técnicas convencionais de depuração tornam-se tão eficazes quanto os métodos primitivos, gerando uma pressão natural para que os depuradores evoluam um passo adiante.

Embora a palavra *depuração* seja bastante genérica, é possível traçar, até certo ponto, uma delimitação bastante precisa a respeito de o que é um *sistema*, um *problema* e um *depurador*. Sem entrar em maiores detalhes por agora, cabe dizer que este estudo tem como objetivo explorar técnicas de modelagem e implementação de depuradores *de baixo* e *alto nível*, com um foco especial em sistemas *distribuídos.* Isso se traduz, dentre outras coisas, em um estudo aprofundado de sistemas distribuídos e de depuradores "seqüenciais", como se verá mais adiante.

1. DEPURAÇÃO TRADICIONAL

Depuradores são antigos conhecidos dos programadores. Capacidades como monitoramento do estado de *threads* e variáveis, visualização da pilha de execução, *breakpoints* e modos *stepping* integram o leque de funcionalidades oferecidas pela maior parte dos depuradores atuais. Alguns chegam a fornecer possibilidades mais sofisticadas, como execução reversa, execução (parcial) de código contendo erros de compilação[1] ou aplicação dinâmica de correções e *rollback* (citando apenas algumas), envolvendo, para tanto, mecanismos bastante complexos de verifcação de consistência de estado com mapeamentos de dependências.

Todas essas funcionalidades são, no entanto, exemplos do que chamamos, neste texto, de "funcionalidades de baixo nível". Isso porque são funcionalidades que se apresentam sempre na relação intrínseca entre programa em execução, linguagem e código-fonte:

[1] Sei que isso pode parecer um tanto estranho, mas alguns depuradores fazem isso. Eles conduzem uma espécie de compilação parcial de código (compilam somente o código sem erro) e depois executam o programa mantendo um controle de quais linhas contêm erros. Se uma linha contendo um erro for atingida, uma exceção é lançada (o depurador de Java integrado no Eclipse, por exemplo, faz isso). São bastante úteis quando testando funcionalidades básicas em sistemas incompletos (não é necessário completar o código antes de testá-lo).

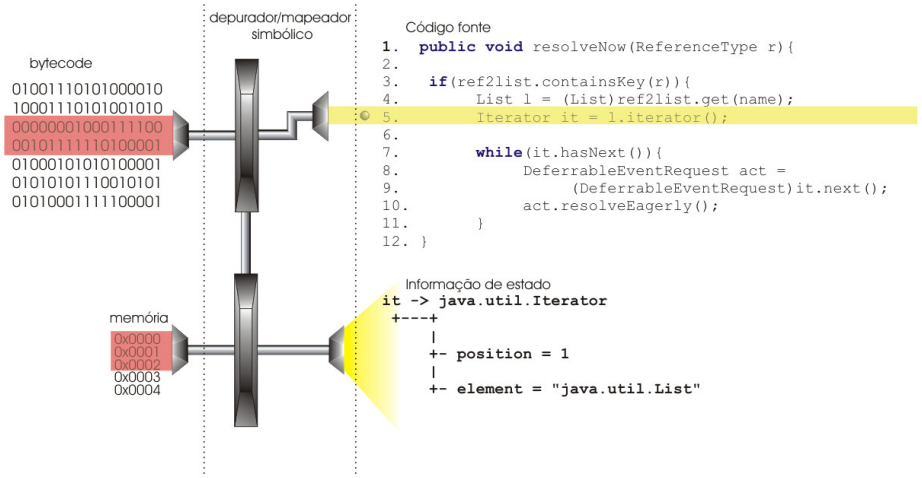

fig 1. Um depurador simbólico

Por essa razão, os depuradores "tradicionais" (ou *source-level debuggers* como também são conhecidos) são chamados *depuradores simbólicos* (já que o código-fonte é composto por "símbolos"). De posse do nosso conhecimento como usuários de depuradores tradicionais[2], como faríamos para formular, desse mesmo ponto de vista (de um usuário), um depurador de sistemas distribuídos?

Creio que a primeira idéia que nos vem à cabeça é: seria possível construir tal depurador de forma que o seu "comportamento aparente" fosse parecido com o de um depurador simbólico tradicional executando um programa *multithreaded*? Bem, talvez. Mas alguns problemas surgem logo de imediato:

⇨ Num programa distribuído, não existe um estado global facilmente observável. Aliás, muito pior do que isso, sistemas distribuídos são inerentemente assíncronos. Isso significa que a informação exibida pelo depurador, num dado instante, não necessariamente corresponde à do estado atual do sistema ou, muito pior do que isso, não corresponde sequer a um *estado possível* do sistema (veremos uma definição mais precisa de *estado possível* mais adiante);

⇨ num sistema distribuído genérico, não há um relógio global para todos os nodos. A implicação imediata disso é que, na ausência de algum mecanismo estabeleça uma *ordem* nos eventos recebidos, fica impossível determinar sequer uma estimativa da execução "em tempo real" do sistema, muito menos uma estimativa que sirva para alguma observação de depuração;

⇨ a assincronia dos sistemas distribuídos significa também que a interferência gerada por um depurador numa execução qualquer do sistema é muito mais profunda do que num sistema seqüencial. Para entender o porque, basta notar que a defasagem entre processos rodando em diferentes nodos pode aumentar significativamente pela introdução do overhead incorrido por um depurador simbólico local a um processo. Por "significativamente", queremos dizer que essa alteração, por produzir um desvio na execução do sistema, faz com que o

[2] Suponho aqui que você já tenha usado um depurador como o GDB ou o depurador Java do Eclipse, por exemplo, para depurar um sistema *multithreaded*.

comportamento observado seja, fatalmente, diferente do comportamento que *seria* observado na ausência do observador (essa característica remete ao princípio da incerteza de Heisenberg [6]).

⇨ Note que, embora um sistema *multithreaded* sofra com o mesmo problema, *threads* podem ser (e são) sincronizadas com relativa facilidade pelo depurador, já que todas elas rodam no mesmo processador ou em processadores muito próximos (por exemplo, ao atingir um *breakpoint* o depurador suspende automaticamente todas as *threads* em execução). Sincronizar processos a distância, levando em consideração falhas e latência de rede, produz interferências de execução muito mais significativas.

Algumas outras questões surgem no contexto de sistemas paralelos em geral e não somente de sistemas distribuídos (lembrando que sistemas distribuídos são, antes de mais nada, sistemas paralelos). Tais questões estão ligadas, quase sempre, ao comportamento não-determístico de sistemas paralelos. Se consideramos que a manifestação de um *bug* ou a violação de uma propriedade são fenômenos intimamente relacionados ao desenrolar da seqüência de eventos que tomam lugar no âmbito global de um sistema, podemos apreciar a dificuldade de isolamento desses *bugs*, especialmente quando tudo de que se dispõe é um depurador simbólico tradicional.

Logo se vê que depurar um sistema distribuído não é uma tarefa simples. Ainda que seja possível desenvolver um depurador que apresente ao usuário uma analogia *multithreaded* de um sistema distribuído, é necessário ainda lidar com as complicações dos sistemas paralelos. A tarefa envolve, portanto, um estudo mais aprofundado nas áreas de depuração de sistemas paralelos, conceitos de sistemas distribuídos e implementação de depuradores simbólicos.

:: Além do simbólico

É fato que depuradores simbólicos não-distribuídos cumprem o seu propósito ao permitirem uma inspeção detalhada do estado do código em execução através de mapeamentos simples entre estados + threads e código-fonte. A discussão acima deixa aparente, no entanto, as suas enormes limitações nos contextos expostos - é preciso especificar a depuração num nível mais alto. Para abarcar as semânticas mais complexas requeridas, precisamos, essencialmente, desvincular o processo de depuração das semânticas de linguagens específicas (exceto na eventualidade de haver uma “linguagem distribuída”).

Infelizmente, sistemas distribuídos tendem a variar consideravelmente em arquitetura e na semântica (e implementação) das primitivas de comunicação adotadas. Essas variações fazem com que os requisitos impostos sobre esse “nível mais alto” da depuração sejam muito próprios de cada combinação. Traçamos abaixo um pequeno conjunto de extensões possíveis. Essas extensões nos parecem ocupar o cerne das necessidades de depuração da maior parte dos sistemas paralelos e distribuídos atuais:

1. Rastreio de threads distribuídos - especificamente voltado para sistemas que fazem uso de abstrações de métodos (ou procedimentos) remotos;
2. possibilidade de observação de estados globais “ao toque de um botão” ou programaticamente;
3. detecção de violações em propriedades de segurança e/ou vivacidade (i.e. deadlocks, predicados arbitrários).
4. Mecanismos de replay controlado de execução.
5. Visualização compreensiva da execução do sistema (i.e. em sistemas que utilizam trocas de mensagens, gostaríamos de visualizar grafos de eventos; em sistemas CORBA [5], gostaríamos de observar o desenrolar de threads distribuídos; etc).

Cada um desses requisitos é assunto, sozinho, para dezenas de livros e artigos. Além disso, a maior parte deles pode ser relacionada em algum nível. Vamos abordá-los, portanto, entrelaçados e ao curso de uma discussão.

2. UM DEPURADOR DISTRIBUÍDO HIPOTÉTICO

Tomemos como exemplo um sistema distribuído hipotético constituído por um certo número, digamos cinco, de *componentes* localizados em nodos arbitrários e que utilize CORBA como sistema de comunicação. Suponha agora que, num ponto arbitrário da execução, uma exceção é lançada em um dos nodos. É razoável supor que gostaríamos de determinar a causa dessa exceção. Há alguns inconvenientes que envolvem o processo de isolamento de um erro tão simples como esse:

1. A máquina em que a exceção foi lançada quase nunca é a mesma máquina em que o desenvolvedor está trabalhando;

2. a exceção pode ser resultado de uma combinação específica de fatores, sendo praticamente impossível determinar, mesmo com o auxílio de *logs* locais, a causa exata do erro. Isso porque muitas vezes o erro observado (exceção) é somente um subproduto de toda uma cadeia de propagações de um erro anterior.

Bem, o primeiro problema é mais um problema de produtividade do que de depuração distribuída propriamente dito, mas essas dificuldades do processo ad hoc de “isolamento manual” de erros é que tornam a de depuração de sistemas distribuídos uma tarefa tão complexa, cansativa e propensa a erros. Em resumo, é muito difícil rastrear a execução de um programa não-determinístico e distribuído simplesmente indo de nó em nó e observando resultados gerados por depuradores simbólicos locais. Imagine a praticidade de aplicação de práticas como a depuração em ciclos[3] num ambiente como esses.

Uma ferramenta capaz de exibir, de forma compreensiva, as dependências entre eventos localizados nos vários pontos do sistema, permitindo a inspeção de porções

[3] Depuração em ciclos é aquela modalidade de depuração em que um desenvolvedor repetidamente reproduz a execução errônea do sistema e, com o auxílio de um depurador simbólico, procura ganhar uma melhor compreensão das causas do erro a cada iteração. Eventualmente (basta ter fé), é possível isolar o erro e corrigí-lo.

previamente selecionadas de estados locais em nodos específicos seria extremamente útil.

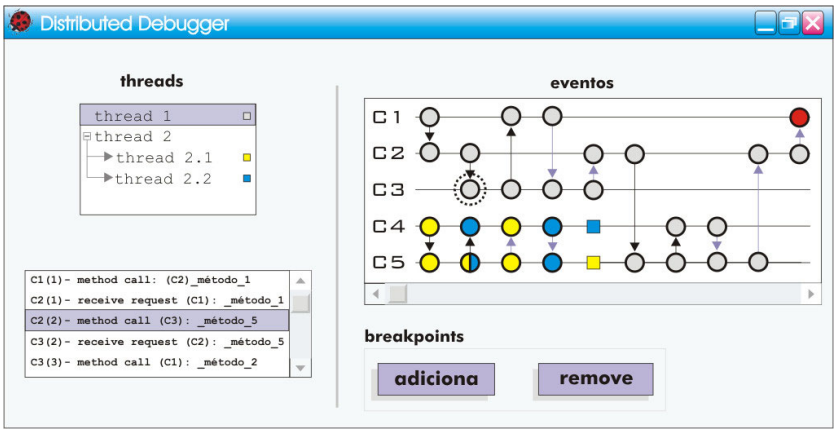

fig. 2 – Protótipo de um depurador distribuído

Nessa ferramenta hipotética (razoavelmente próxima daquilo que gostaríamos de implementar), um desenvolvedor poderia observar, da sua máquina, com razoável clareza, um grafo de *dependências causais entre eventos* que ocorreram no sistema, incluindo o instante em que a exceção (em vermelho) foi lançada.

Flechas pretas marcam envios de requisições CORBA (RMI/RPC); flechas cinzas marcam seus respectivos envios de resposta. Os círculos representam os eventos, que podem ser de envio, recebimento e morte. As cores separam, no grafo, os diversos *threads* distribuídos do sistema; eventos multicoloridos indicam a criação de um (ou mais) novo(s) thread(s) distribuído(s) no intervalo assim determinado. *Threads* locais são considerados *threads* distribuídos. Quadrados coloridos indicam o fim do *thread* distribuído de mesma cor.

A representação visual é, por si só, de grande valia; no entanto, para que possamos determinar de fato a *causa* da exceção, é necessário que seja possível, antes de mais nada, inspecionar o estado dos componentes de C1 até C5 nos intervalos entre cada um dos eventos. Poderíamos, para isso, tentar gravar externamente o estado de cada componente a cada envio ou recebimento de mensagem, mas isso nos permitiria apenas observar alguns estados em picotes discretos de instantes que nem sequer sabemos se são interessantes (embora, em alguns casos, isso seja mais do que suficiente, como vamos discutir mais adiante).

O maior problema com essa técnica é, no entanto, a sua inabilidade em lidar com uma modalidade de propriedades conhecidas como *predicados locais instáveis*. Um *predicado local* pode ser definido como uma função booleana *prop(Y)* cujo domínio de valores é definido no conjunto de estados possíveis para um componente ou processo local[4] (se pensarmos na execução de um componente ou processo local como uma sucessão de estados). *Predicados instáveis* são aqueles que podem potencialmente alternar entre **verdadeiro** e **falso** várias vezes durante uma execução.

[4] Em geral, local = não-remoto. No nosso texto, local = não-remoto = não-distribuído.

Deve estar claro pela explicação que, se a causa de um erro for um predicado instável, fotografar os estados dos componentes em instantes arbitrários torna-se uma solução quase tão boa quanto chutar o gabinete para ver se o bug vai embora. Uma saída possível seria testar os componentes em tempo de execução - para cada mudança de estado, poderíamos aplicar a função *prop(Y)* e verificar se dado predicado é verdadeiro ou não (claro que essa técnica se complica quando queremos testar a presença de predicados instáveis não-locais).

Bem, mas ainda que fôssemos capazes de detectar os tais predicados instáveis, nosso problema ainda não estaria resolvido – ainda não seria possível observar os estados dos componentes em pontos arbitrários do tempo.

2.1 REPLAY DE EXECUÇÃO

Uma alternativa bastante interessante para o problema em questão é um procedimento conhecido como replay de execução. O replay de execução, como o próprio nome sugere, é um nome geral dado para um conjunto de técnicas que reproduzem uma certa *especificação de execução*; que pode ser arbitrária ou gerada durante uma execução real do sistema por um *tracer* (ou *rastreador*[5]).

Nossa intenção é bem clara: ao invés de gravarmos explicitamente o estado de cada nodo, gostaríamos de rastrear apenas um certo conjunto de informações. Em particular, gostaríamos de um conjunto de informações que nos permitisse *reproduzir* a execução do sistema *da melhor forma possível*.

Imagine as possibilidades: com uma ferramenta como essas, poderíamos capturar um execução distribuída e instalar *breakpoints* em pontos estratégicos. Bastaria então acionar o replay de execução e poderíamos, “magicamente”, examinar os estados dos componentes nos pontos (arbitrários) marcados, utilizando um depurador simbólico tradicional. Note que a interface da figura 2 já prevê essa modalidade de uso com os seus botões de **adiciona** e **remove**.

Claro que para que essa alternativa faça algum sentido, o custo de rastreio de uma execução de um sistema deve ser substancialmente inferior ao custo de rastreio de estados individuais de componentes. Felizmente, esse é o caso.

Idealmente, gostaríamos de aplicar uma transformação no sistema distribuído que force uma reexecução idêntica a uma outra execução previamente rastreada ou construída. Como uma reexecução **idêntica** é profundamente imprática (reprodução exata da mesma ordem relativa em todas as instruções assembly em todos os processadores de todos os nodos), nos contentaremos com uma reexecução *muito parecida* com a original; nominalmente uma reexecução que:

1. Mantenha as relações causais entre eventos através de todo o sistema distribuído.

[5] Um tracer, ou rastreador, é, num certo sentido, um fotógrafo especializado de um software em execução. Seu papel é o de capturar informações relevantes, deixando para trás “traços” da execução de um sistema.

2. Reproduza as condições de corrida[6] encontradas entre *threads* distribuídos através de todo o sistema.

Vamos agora justificar um pouco essas afirmações, principalmente o porque dessas propriedades garantirem (até onde o meu conhecimento permite ver) uma reexecução "interessante".

3. TEMPO, EVENTOS E ESTADOS

Antes de prosseguir, creio que seja interessante procurar definir um pouco melhor e em termos mais precisos alguns dos conceitos aos quais fizemos menção no decorrer do texto. Em particular, o conceito de *execução* de um sistema distribuído pede por uma definição urgente. Vamos utilizar, para a nossa caracterização, um modelo bastante simplificado de sistema distribuído (como em [2, 3]). Note que existem modelos mais completos, mais interessantes e mais importantes que o nosso (os modelos baseados em autômatos de Entrada e Saída [4] são um exemplo), mas em função do nosso intento (caracterizar a causalidade num sistema), este modelo simples é suficiente.

Fundamentalmente, um sistema distribuído é constituído por um conjunto de processos $P_1,...,P_n$ onde, numa dada *execução distribuída*, podemos observar nas *execuções locais* dos processos P_i uma sucessão de estados S_i e uma sucessão de eventos, que causam as mudanças de estado em P_i. Inicialmente, vamos assumir que os P_i são seqüenciais (depois veremos como lidar com questões associadas ao *multithreading*). Eventos podem ser internos ou externos, sendo que eventos externos são eventos relacionados ao envio e recebimento de mensagens.

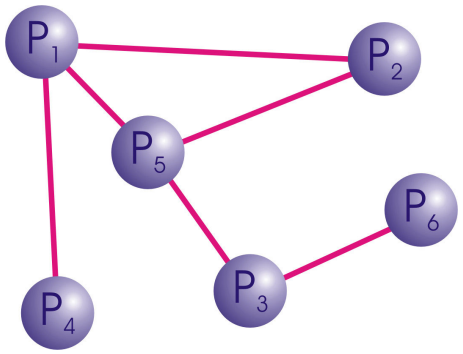

fig 3. sistema distribuído de topologia arbitrária

Agora, algumas definições:

[6] Talvez esteja soando um pouco estranho falar em "condições de corrida" logo aqui, mas o sentido de condição de corrida adotado aqui é um pouco mais abrangente do que a definição usual (como veremos adiante).

Definição 1: Dados dois estados $a,b \in S_i$, diremos que $a \triangleright b$ se e somente se a precede imediatamente b na execução seqüencial de P_i.

Essencialmente, a é o estado que precede temporalmente b e que se encontra mais próximo de b na execução de P_i. Note que S_i é um conjunto irreflexivo e totalmente ordenado sob o fecho transitivo de $\triangleright$.

Definição 2: Para dois estados $a \in S_i, b \in S_j$, dizemos que $a \prec b$ (a *precede remotamente* b) se e somente se uma mensagem for enviada por P_i no evento que vem logo depois de a (e causa a transição para outro estado) e recebida em P_j no evento que vem logo antes de b (causa a transição para b).

Definição 3: A relação de *precedência causal* $\rightarrow$ é a relação resultante do fecho transitivo e irreflexivo de $\prec \cup \triangleright$.

Essa relação, que define uma ordem parcial, é conhecida também como a relação *acontece antes* de Lamport [1].

Diremos que toda execução local começa num estado especial de início Δ_i e termina num estado especial de término ∇_i.

Dadas as definições acima, vamos modelar uma *execução distribuída* como um deposet $(S_1,...,S_n,\rightarrow)$, provido que três restrições bastante razoáveis sejam respeitadas:

1. Nenhuma mensagem é recebida antes do estado inicial;
2. nenhuma mensagem é enviada após o estado final;
3. um evento é sempre de envio ou de recebimento, mas nunca os dois ao mesmo tempo.

Um *estado global* fica definido como uma tupla $(a_1,...,a_n)$ onde $a_i \in S_i, 1 \leq i \leq n$. Uma execução distribuída é composta por uma sucessão de estados globais, como é de se esperar. Pela definição apresentada até agora, no entanto, algumas situações estranhas podem acontecer[7]:

[7] Nas figuras que seguem, vamos amalgamar estados e eventos em marcas que representam ambos na linha temporal (apesar dessa bagunça, as idéias devem ficar claras).

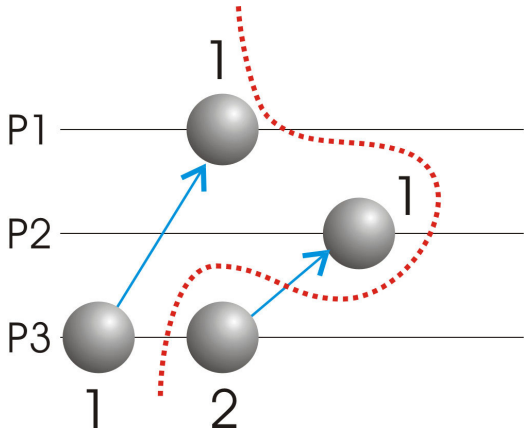

fig 4. Um estado impossível

Note que, pela definição apresentada, $(1,1,1)$ é um estado global, mesmo embora ele viole uma propriedade fundamental da causalidade: um evento (estado) futuro não pode influir em eventos (estados) presentes e nem em eventos (estados) passados. A inconsistência aparece por estarmos tentando definir um estado global que inclui um estado produzido por um evento de recebimento de uma mensagem mas não o estado produzido pelo evento de envio que *causa* esse evento de recebimento (i.e. não o evento que envia a mensagem).

Dizemos que um estado global como esse, que viola condições causais fundamentais, é um *estado global impossível*. Daqui em diante, iremos nos referir a estados globais como *cortes*[8]. Um *corte* é *inconsistente* quando ele representa um estado global *impossível*.

A violação causal pode ser melhor visualizada se fizermos a famosa *rubberband transformation* de Mattern [7]. Imagine que as linhas temporais onde estão localizados os eventos na figura 4 são feitas de tiras de borracha. Podemos, nesse caso, esticar algumas dessas linhas de tal forma que a linha pontilhada vermelha, curva na figura 4, se transforme em uma reta vertical:

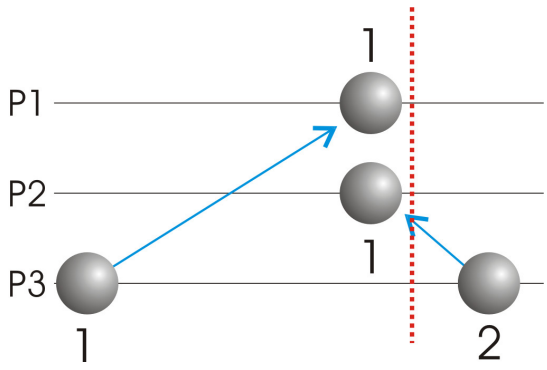

fig 5. Transformação da tira de borracha e a violação causal

[8] Cabe aqui uma licença poética – cortes são ligeiramente diferentes de estados globais, mas possuem uma relação muito clara. Uma definição precisa de corte pode ser encontrada em [7].

Intuitivamente a idéia é clara, mas e no nosso modelo? Quando é que podemos dizer que um corte $(a_1,...,a_n)$ representa um *corte consistente?* Bem, basta que não haja pares de estados causalmente dependentes, ou acabaremos como na figura acima[9]. Formalmente, um corte P é consistente se $\forall\{a,b \in P \Rightarrow a \not\rightarrow b \wedge b \not\rightarrow a\}$; ou, em notação equivalente, $\forall\{a,b \in P \Rightarrow a \,||\, b\}$. Dizemos que dois estados satisfazendo $a \,||\, b$ são estados *concorrentes*. Note o forte paralelo existente entre a noção temporal/causal num sistema distribuído e a noção temporal/causal como definida na física, em especial com a noção temporal de Minkowski. A transformação da tira de borracha poderia ser comparada às transformações de Lorentz [7].

Uma tupla $(a_1,...,a_n)$ representa um corte consistente do sistema distribuído se e somente se, para quaisquer pares de eventos a_i, a_j na tupla, $a_i \,||\, a_j$.

Nos voltamos agora, novamente, para a questão do replay de execução.

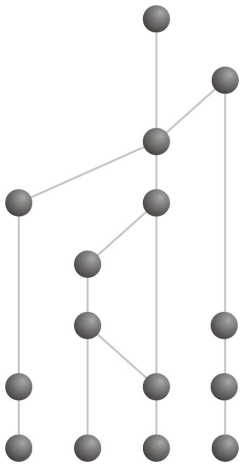

fig.6 – Poset de uma execução

Como sugerido anteriormente, uma execução de um sistema distribuído pode ser modelada como um deposet. Tudo o que precisamos fazer é, portanto, garantir que a execução do sistema gere sempre o mesmo deposet (já que, para fins teóricos, **uma execução é um deposet**). Na prática, isso significa que devemos forçar, para uma dada cadeia causal, uma entrega de mensagens que a respeite. Isso poderia ser feito modificando-se a infraestrutura de comunicação (no caso do MPI ou CORBA, por exemplo, podemos nos utilizar de buffers ou de bloqueio de processos) ou através de técnicas de transformações de controle distribuídas, conforme vamos discutir mais adiante, se der tempo ☺.

Uma demarcação de mensagens que reproduz a ordem parcial entre eventos denotada pela figura 6 é o relógio lógico de Lamport [1]. Alguns outros métodos com propriedades mais interessantes são discutidos em [7, 15].

[9] Poderia ser que, após a transformação da tira de borracha, alguma das flechas azuis se tornasse vertical (e não aparentasse estar voltando no tempo). Isso também representa uma violação causal – lembre-se que eventos que ocorrem no presente não podem influir em outros eventos que também ocorrem no presente.

Note que, contanto que os processos individuais sejam determinísticos, essa condição garante, do ponto de vista do nosso modelo, que a execução do sistema distribuído será idêntica à execução especificada ou rastreada (o que já é suficiente para os nossos propósitos de inspeção de estados). Garantir que os processos individuais sejam determinísticos pode ser, todavia, um problema bastante complicado - pode ser necessário instrumentalizar bibliotecas que façam uso de números aleatórios, bem como bibliotecas de entrada e saída (para que produzam sempre os mesmos resultados). O problema complica-se ainda mais quando temos processos com vários *threads*.

Uma alternativa tentadora, que elimina os complicados mecanismos de sincronização de processos presentes na maior parte dos sistemas de replay de execução (mas não resolve o problema do *multithreading*), consiste em gravar o conteúdo das mensagens recebidas por cada processo (e não mais a ordem relativa). Esse método nos permite executar *replays* de processos isoladamente, através de uma espécie de recriação virtual do ambiente externo de cada nodo. Infelizmente, não há almoço de graça - essa alternativa, via de regra, gera volumes monstruosos de traços, mesmo em execuções curtas.

3.1 MULTITHREADING

Até agora, estivemos cuidadosamente (ou talvez nem tanto) varrendo para baixo do tapete a questão do *multithreading* nos processos locais. Isso porque, dada a exposição de idéias da seção anterior, poderíamos imaginar diversas maneiras de introduzir o determinismo num sistema de troca de mensagens (uma delas seria instrumentalizando a infraestrutura de troca de mensagens de alguma forma); no entanto, imaginar como produzir o mesmo efeito, mas num sistema de memória compartilhada e onde os processos são agendados pelo kernel de um sistema operacional, é algo mais complicado. Primeiro porque instrumentalizar o acesso à memória é uma tarefa bem menos trivial; segundo porque, se não tomarmos providências, o volume de traços pode ficar muito, mas muito grande (muito maior do que o número de mensagens trocadas numa execução é o número de acessos a memória feito por cada thread em cada processo nessa mesma execução).

Ainda assim, não é um problema tão ruim quanto parece. Apesar das questões técnicas inerentes, alguns dos problemas mais intransponíveis na depuração de sistemas distribuídos simplesmente desaparecem quando no ambiente controlável e síncrono proporcionado por um único processador, ou mesmo nos casos em que há mais de um processador, mas onde as latências de rede são negligenciáveis ou podemos contar com um relógio global.

Podemos adiantar que, de forma semelhante ao raciocino desenvolvido com os sistemas de troca de mensagens, podemos formular uma causalidade num sistema de memória compartilhada. Essa causalidade se dá em função das dependências dinâmicas de dados criadas por condições de corrida que ocorrem entre *threads* concorrentes num mesmo processo (ou processos concorrentes num sistema de memória compartilhada).

3.2 CONDIÇÕES DE CORRIDA

O termo "condição de corrida" vem associado, usualmente, a um comportamento errôneo ou inconsistente que decorre de uma ordem não-prevista na execução de dependências dinâmicas de num sistema concorrente. No âmbito do nosso texto, uma **condição de corrida** ocorre sempre que houver acesso concorrente a uma área memória compartilhada.

Note que é tudo uma questão de granularidade. Mesmo nos casos em que há sincronização de acesso aos blocos de memória compartilhada, ainda assim existe um indeterminismo incorrido pela ordem de entrada nas regiões críticas (veja *fig. 7*). Portanto, do nosso ponto de vista, uma condição de corrida abrange o caso em que há blocos de sincronização.

Embora isso possa parecer estranho a primeira vista, note que, em última instância, um "bloco sincronizado" pode ser uma instrução *assembly* que modifica a memória. A única diferença é que esse bloco é sincronizado pela própria natureza dos circuitos síncronos empregados nos processadores.

Na prática, a granularidade dos blocos considerados é ditada pela linguagem de programação em consideração – se uma dada linguagem executa atomicamente um bloco de código, então é na execução desse bloco de código pelos sucessivos *threads* que pode ocorrer uma "condição de corrida"[10].

3.3 RASTREIO E OTIMALIDADE

Vamos nos aprofundar um pouco mais agora nas questões que concernem a determinação de um conjunto mínimo de informações que devem ser rastreadas para que a reprodução das condições de corrida e relações causais discutidas anteriormente possam ser factíveis. Esta discussão é muito pertinente já que, num sistema de médio porte (com um fluxo de dados da ordem dos gigabits por segundo), a quantidade de dados gerada por um rastreador ineficiente em um intervalo de tempo de alguns minutos pode rapidamente ultrapassar a barreira do gerenciável.

É possível estabelecer um paralelo entre as situações que causam o indeterminismo nos sistemas de memória compartilhada e as situações que causam indeterminismo nos sistemas de troca de mensagens. Em outras palavras, podemos estabelecer, num sistema de trocas de mensagens, uma situação análoga às condições de corrida consideradas nos sistemas de memória compartilhada.

[10] Isso dá mais uma dica no sentido de que os depuradores distribuídos terão de relacionar-se com os depuradores simbólicos num nível bastante íntimo.

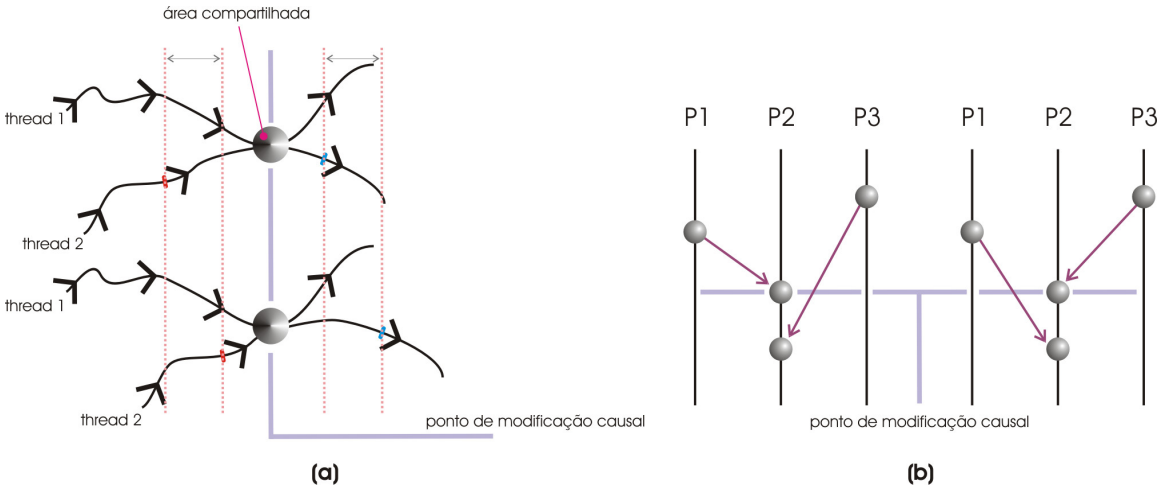

fig 7. Condições de corrida entre threads (a) e entre mensagens (b)

Como podemos observar na figura, duas mensagens serão ditas *concorrentes* sempre que for determinado, numa dada seqüência de recebimentos de mensagem, que essa seqüência **poderia** ter sido diferente. O paralelo com a memória compartilhada é bem claro – a ordem de execução dos eventos é quem dita a formação das dependências dinâmicas de dados - a única diferença entre os dois casos é que num tratamos da ordem das mensagens, no outro da ordem de execução de blocos atômicos de código.

A questão toda do replay de execução se volta, neste instante, para a existência de um método que seja eficiente na detecção das condições de corrida. O seguinte teorema, que iremos apresentar sem prova, parece jogar no abismo quaisquer esperanças:

Teorema 1: A detecção estática de condições de corrida é um problema NP-completo.

Felizmente, há uma sutileza. Nós não queremos responder, para um dado trecho de código, se existe uma execução onde ocorre uma determinada condição de corrida (esse problema é o problema NP-completo do **Teorema 1**). Queremos apenas determinar onde e se elas ocorrem numa dada execução (vamos formular melhor esse problema no decorrer do texto).

3.3.1 RASTREIO EM MEMÓRIA COMPARTILHADA E SUAS CONTRADIÇÕES

Vamos agora caracterizar um subconjunto de acessos à memória compartilhada que é suficiente para que possamos reproduzir uma execução de um sistema *multithreaded*.

Inicialmente, é necessário modificar ligeiramente o modelo. Essas modificações se dão num aspecto quase que puramente qualitativo-semântico, uma vez que há, como já

sugerimos na seção anterior um forte paralelo entre os aspectos causais entre os sistemas de passagem de mensagem e os sistemas de memória compartilhada [9, 10].

Aqui, uma execução do sistema será modelada por uma tripla $P = (E, \xrightarrow{T}, \xrightarrow{D})$ onde E é um conjunto finito de eventos e $\xrightarrow{T}$ e $\xrightarrow{D}$ são relações definidas sobre E. Os eventos podem ser divididos em eventos de *leitura* e eventos de *gravação* (eventos que acessam estado e eventos que modificam estado, respectivamente).

1. A relação $\xrightarrow{T}$ representa a relação natural de ordem temporal entre eventos dentro de um mesmo *thread*;
2. a relação $\xrightarrow{D}$, representa a dependência causal entre dois eventos, dada pelo fecho transitivo e irreflexivo da relação $\xrightarrow{DD}$ descrita abaixo.

Dizemos que $a \xrightarrow{DD} b$ sempre que existir uma dependência direta de dados estabelecida entre os eventos a e b; isto é, sempre que a e b acessarem uma variável e, nessa cadeia de acessos, pelo menos a ou b modificarem o valor dessa variável. Também dizemos que $a \xrightarrow{DD} b$ se a precede b no mesmo *thread*.

Definição 4: Dada uma execução $P = (E, \xrightarrow{T}, \xrightarrow{D})$, dizemos que uma *corrida limítrofe*[11] $\langle a, b \rangle$ existe entre dois eventos a e b se, e somente se:

1. a e b pertencerem a processos (*threads*) distintos, e
2. $a \xrightarrow{D/} b$, onde $\xrightarrow{D/}$ é a redução transitiva de $\xrightarrow{D}$

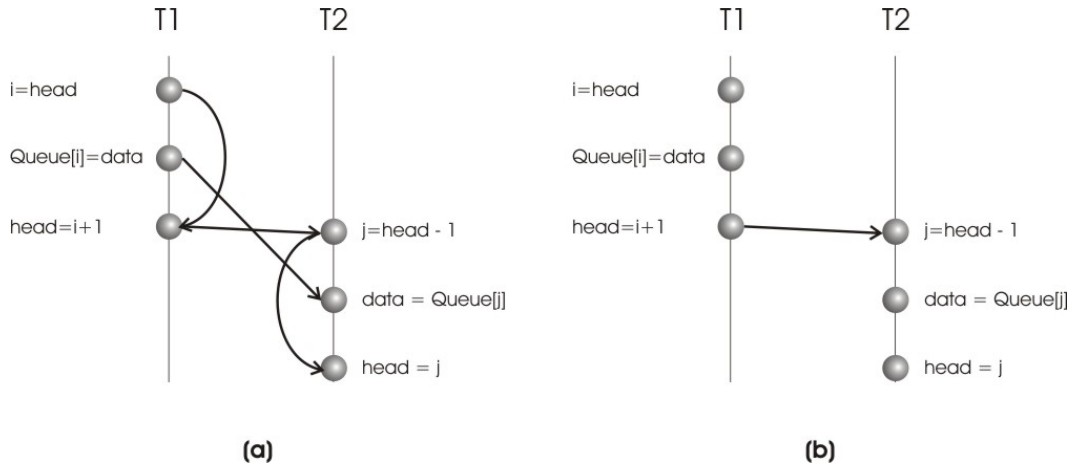

fig. 8 – Dependências dinâmicas de dados **(a)** e redução transitiva **(b).** Note a redução no número de arestas em **(b)**.

[11] Desculpem o termo, não consegui arrumar uma tradução melhor para *frontier race*.

O seguinte teorema, apresentado sem prova[12], revela que examinar *corridas limítrofes* pode ser de fato algo interessante:

> **Teorema 2 (*replay*):** Um replay de execução é correto sse todas as *corridas limítrofes* forem resolvidas no replay de execução na mesma ordem em que foram produzidas na execução original.

Um algoritmo que pretenda rastrear corridas limítrofes deve, essencialmente, ser capaz de rastrear as dependências entre acessos atômicos a blocos de memória produzidas por *threads* concorrentes. A cada acesso, o algoritmo deve decidir se há ou não uma dependência direta entre o evento corrente de acesso e algum outro evento de modificação do bloco atômico no grafo gerado pela redução transitiva das dependências de acesso na execução do programa. Podemos determinar tais dependências mantendo um histórico de acesso para cada variável compartilhada [9].

Sempre que houver uma dependência não-transitiva entre dois eventos; isto é, sempre que na redução transitiva do grafo de dependências houver uma aresta que liga dois estados, nós devemos, pelo **teorema 2**, gerar um traço da ordem de execução desses eventos para que o sistema de replay de execução reproduza esses eventos na exata ordem em que eles ocorrem na execução original.

Bem, é claro que não nos daríamos ao trabalho de explicar todo esse método se a quantidade de traços gerados pela detecção das corridas limítrofes não fosse substancialmente menor do que a quantidade de traços geradas por um algoritmo ingênuo (que rastreia todos os acessos à memória compartilhada). Estudos [9] revelam que uma implementação desse método rastreia, em média, de 0,01% a 2% dos acessos à memória compartilhada. Uma redução de quase duas ordens de magnitude com relação ao algoritmo ingênuo. Caso ainda haja dúvidas com relação à optimalidade da técnica, apresentamos o seguinte corolário:

> **Corolário (2.1):** O conjunto de dependências compreendido pelas corridas limítrofes é o menor conjunto de dependências requeridas para um replay de execução correto.

Prova: Imediata, uma vez que pelo **teorema 2** as corridas limítrofes constituem a intersecção das dependências requeridas por qualquer replay de execução que seja correto e, ao mesmo tempo, é garantido que um replay é correto se as dependências compreendidas pelo conjunto das corridas limítrofes forem reproduzidas.

Agora, um pouco a respeito da palavra “contradições” presente na descrição desta seção. O fato é que pudemos encontrar alguns artigos [11, 16] mais recentes do que o artigo que apresenta o **teorema 2** [9] e que dizem ser impossível “remover a hipótese de que o software não apresenta condições de corrida”, um deles inclusive propondo verdadeiros malabarismos para tratar esses “casos especiais”.

Esses artigos apontam para o fato de que o overhead incorrido por rastrear todos os acessos à memória seriam inaceitáveis e que, portanto, devemos supor que todos os acessos onde potencialmente pode haver uma condição de corrida estão protegidos por

[12] Uma prova desse teorema pode ser encontrada em [10].

mutexes ou blocos de sincronização (como em Java). Dessa forma, devemos apenas garantir no replay a reprodução da ordem em que os threads entram nas áreas sincronizadas (regiões críticas).

De fato devemos reproduzir a ordem em que os threads entram em suas regiões críticas mas, particularmente, não entendo onde, de acordo com o raciocínio de granularidade desenvolvido na seção anterior, é que isso elimina as condições de corrida. A meu ver, elas sempre existem - o que procuramos eliminar ao aumentar a granularidade das operações são as inconsistências geradas por tais condições. A outra questão levantada pelos artigos, em particular a do "overhead inaceitável", parece ser justamente a questão solucionada em [9].

É claro que sempre contamos com a hipótese de termos deixado de lado algum aspecto chave que só surge no âmbito de implementação da solução para o problema. O esclarecimento dessa aparente contradição fica como trabalho futuro.

3.3.2 RASTREIO EM SISTEMAS DE PASSAGEM DE MENSAGEM

Apresentamos agora[13] um método que nos permite reduzir o número de mensagens rastreadas para replays de execução em sistemas de passagem de mensagens (não mais em sistemas de memória compartilhada).

A idéia, parecida em certo aspecto com a idéia desenvolvida nos sistemas de memória compartilhada, parte do princípio de que é possível ganhar algum insight na configuração de envio das mensagens através de algumas análises de dependências simples.

fig.9 – mensagens concorrentes (a) e seqüência determinística (b)

Obviamente queremos rastrear um número de mensagens que seja menor do que o número total de mensagens enviadas no sistema. A seguinte heurística nos ajuda nessa direção:

[13] Na verdade nós não "apresentamos" nada. Esse método foi desenvolvido por Netzer e Miller em [11], tudo o que fizemos foram alguns acréscimos, para contextualizá-lo no nosso modelo e no nosso texto.

Heurística 1: Dadas duas mensagens $a \prec b$ e $c \prec d$, onde $b \rhd d$, $b,d \in E_i$, $a \in E_j, c \in E_k$ $(i \neq j, i \neq k)$, rastreamos uma das mensagens sempre que $b \not\rightarrow c$.

Ignorando por um instante que não definimos algumas dessas relações para pares de eventos (definimos somente para pares de estados) é possível notar, pela **fig. 9**, que a heurística descarta um caso em que certamente a entrega foi determinística. A qualidade dessa heurística pode ser apreciada pelos experimentos – tipicamente, 80% das mensagens concorrentes formam configurações causais como essas.

Agora, vamos justificar o abuso de notação.

Começamos notando que podemos definir alguns agrupamentos entre estados que ocorrem nos intervalos entre dois sucessivos eventos de envio e/ou recebimento (iremos nos referir a esses eventos, doravante, como eventos *externos*). Seja P_i um processo no sistema distribuído e E_i o seu conjunto de eventos (que podem ser eventos de envio ou de recebimento) associado. Definimos (i,k) como sendo o conjunto de estados de S_i que ocorrem entre o *(k-1)*-ésimo e o k-ésimo evento externo em E_i.

Finalmente, as dependências causais. Sejam $s,s' \in (i,k)$ estados em S_i e um estado $u \notin (i,k)$. Então $(s \rightarrow u \Leftrightarrow s' \rightarrow u)$ e $(u \rightarrow s \Leftrightarrow u \rightarrow s')$. Partindo dessa observação, podemos estabelecer um morfismo entre estados e intervalos que reduz o grafo de dependências causais entre estados a um grafo que é isomórfico ao produzido pelas relações causais entre eventos.

Em outras palavras, estamos “condensando” todos os estados num dado intervalo e afirmando que o grafo produzido pelas relações causais entre intervalos é isomórfico ao grafo induzido pelas relações causais entre os próprios eventos:

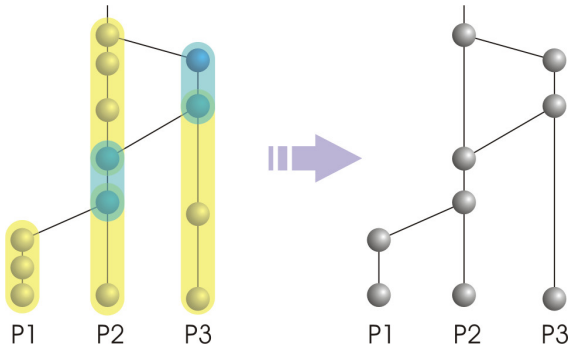

fig. 10 – Um morfismo entre intervalos e eventos.

Portanto, as relações $\rhd$ e $\prec$ também encontram equivalentes no espaços de eventos. Esperamos que essa explicação tenha ficado clara.

Já havíamos dado uma noção intuitiva desse morfismo em pontos anteriores da nossa discussão, mas é sempre boa prática deixar seu uso explícito. Vamos cometer abusos como esse no decorrer do restante do texto, mas sempre de tal forma que seja possível inferir, pelo contexto da discussão, se estamos nos referindo à relação causal entre eventos ou entre estados.

As idéias apresentadas até então sugerem um algoritmo que teste essa "condição de concorrência" a cada mensagem recebida e decida, a partir daí, se a mensagem tem ou não de ser rastreada para reprodução na fase de *replaying*. Um algoritmo muito eficiente para esse teste em pares de mensagens pode ser encontrado em [7, 15].

Pode parecer um tanto aleatório que justamente esses casos tenham sido escolhidos para o algoritmo, mas existe uma razão pra isso. Vamos mostrar agora que, se tentarmos caracterizar todos os casos em que uma entrega de mensagens é não-determinística, acabamos com um problemão em mãos.

Começamos caracterizando as situações em que as entregas de mensagens poderiam ter sido não-determinísticas. Essa caracterização é feita por uma noção de "fronteira" ou limite. Uma fronteira é uma linha que pode ser traçada num ponto qualquer da execução, dividindo-a em dois pedaços. Estamos interessados em todas as fronteiras tais que:

1. Dois (ou mais) eventos de envio se localizem depois da fronteira
2. Um evento de recebimento que poderia ter aceito ao menos duas das mensagens dos eventos de envio em (1) se localize depois da fronteira
3. Todos os eventos de recebimento anteriores à fronteira têm os seus respectivos eventos de envio localizados antes da fronteira.

Seja $P = (E, \rightarrow)$ uma execução de um programa distribuído D.

Definição 5: Definimos F_{MESMO} como o conjunto de todas as execuções $P' = (E', \rightarrow)$ tais que:

1. P' representa uma execução possível para o programa D.

2. P' é idêntica a P até uma certa fronteira, definida da seguinte forma:

Para cada processo, o conjunto E'_p deve conter:

(a) um prefixo dos eventos em E_p (sob $\triangleright$) e;

(b) o primeiro evento em E_p após esse prefixo.

3. P' apresenta a mesma estrutura de entrega de mensagens que P, até o ponto da fronteira.

Note que esse conjunto, no caso de uma entrega não-determinística, deve conter uma execução em que a entrega dos eventos que vêm logo depois da fronteira acontecem de maneira diferente.

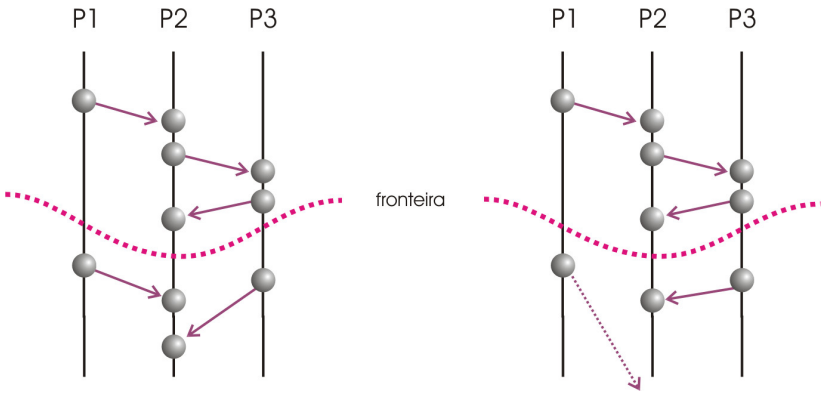

fig. 11 – fronteiras interessantes. Note
à direita uma execução "possível" para o programa.

A seguinte definição sumariza as condições que desejamos detectar desenhando essas fronteiras:

Definição 6: $a \prec b$ *concorre* com $c \prec d$ se e somente se $\exists P'; P' = (E', \rightarrow)$ e $P' \in F_{MESMO}$ tal que $a, b, c \in E'$ e $c \prec b; (c \neq a)$.

Essas condições são uma generalização do que foi exposto na figura **9(a)**. Queremos, portanto, determinar quais mensagens concorrem com quais. Se, para um dado par de mensagens, pudermos determinar que elas são concorrentes, então tudo o que precisamos fazer é rastrear uma delas. Logicamente deve haver uma maneira eficiente (que não seja guardando todas as mensagens e comparando, a cada mensagem, uma com todas as outras) de determinar se um par de mensagens é concorrente. Antes de entrar nesse problema, todavia, há algo que merece a nossa atenção.

Um detalhe aparentemente inócuo que viemos omitindo até o presente momento é que eventos de recebimento e envio podem estar associados a "canais lógicos". Um canal lógico é um túnel de mensagens abstrato, que deve ser especificado por cada processo ao enviar ou receber mensagens – essencialmente, os processos declaram em quais canais lógicos desejam receber mensagens e em quais desejam enviar mensagens. Isso significa que, para determinar se uma entrega foi de fato não-determinística, devemos levar em conta também esse aspecto da comunicação (um evento de recebimento numa execução pode estar associado a um canal lógico específico, significando, para uma dada fronteira, que ele só poderia ter recebido uma determinada mensagem).

Bem, suponha que sejamos capazes de decidir de maneira eficiente se um par arbitrário de mensagens é concorrente. O problema se volta, portanto, para decidir qual das mensagens no par devemos rastrear. Note que não podemos pegar simplesmente "qualquer uma" porque, em alguns casos (especificamente nos casos em que a relação *concorre* não é transitiva – isso está diretamente relacionado à introdução dos canais lógicos), os pares se interligam; isto é, podem existir vários pares da forma (a, b_i) para um dado a. Algo interessante começa a se configurar.

Seja G um grafo não-dirigido, construído da seguinte forma:

1 - Cada mensagem na execução *P* é representada por um vértice
2 - $(i, j) \in E(G)$ se e somente se a mensagem *i concorre* com a mensagem *j* em *P*.

Teorema 3: O problema do rastreamento mínimo de mensagens é NP-difícil.

Prova: Nós vamos provar que o problema de decisão associado a esse problema de otimização, que consiste em decidir se podemos determinar todas as mensagens concorrentes numa execução rastreando k ou menos mensagens, é NP-completo. A tese segue diretamente daí. ■

Teorema 3.1: Decidir se é possível determinar todas as mensagens concorrentes numa dada execução $P = (E, \rightarrow)$ rastreando-se k ou menos mensagens é um problema NP-completo.

Prova: Vamos começar construindo uma instância do nosso problema partindo de um grafo arbitrário e não-dirigido $G = (V, E)$. A construção é fortemente baseada na construção inversa mostrada logo antes de enunciarmos o **Teorema 3.**

Sejam P_1, P_2 dois processos. Construímos a execução P da seguinte forma:

1. O processo P_1 contém em sua execução local $|V|$ eventos de envio para P_2, sendo que o i-ésimo evento de envio é direcionado ao i-ésimo canal lógico.

2. O processo P_2 contém $|V|$ eventos de recebimento, sendo que o i-ésimo evento de recebimento é direcionado ao i-ésimo canal lógico.

Garantimos, desta forma, que a entrega de mensagens é completamente determinística. Agora precisamos dar conta das arestas de *G*. Para cada aresta $(i, j) \in E(G)$, adicionamos à lista de canais lógicos do i-ésimo evento de recebimento o canal lógico j. Como a entrega de mensagens pode ser feita fora de ordem, as mensagens i e j concorrem (note que isso não interfere com as relações *concorre* já estabelecidas entre outros pares de mensagens).

Falta agora provar que a resposta da nossa instância assim construída é "sim" $\Leftrightarrow$ a resposta para a instância do problema de cobertura for "sim".

($\Rightarrow$) Suponha que seja possível determinar, rastreando *k* ou menos mensagens, todas as mensagens que concorrem em P. Isso significa que, para cada par de mensagens concorrentes, ao menos uma das mensagens é rastreada. Se escolhermos em *V(G)* os vértices correspondentes às mensagens rastreadas em *P*, notamos que, para cada par $(i, j) \in E(G)$, ou i ou j foram escolhidos (para cada par concorrente, ao menos uma mensagem é rastreada). Portanto, esses vértices produzem uma cobertura com apenas *k* vértices em *G*.

($\Leftarrow$) Suponha agora que *G* tenha uma cobertura de *k* ou menos vértices. Isso significa que, para cada aresta $(i, j) \in E(G)$, no mínimo i ou j foram escolhidos nessa

cobertura. Portanto podemos determinar todas as mensagens concorrentes em *P* escolhendo *k* ou menos mensagens. ■

Surpreendentemente, embora o algoritmo para a detecção de condições de corrida em *threads* possua as mesmas raízes teóricas do algoritmo de detecção de "condições de corrida" em sistemas de passagem de mensagens (ambos fazem forte uso da estrutura de relações causais), não existe um teorema equivalente ao **teorema 2** para o algoritmo de rastreio de mensagens, por se tratar, incrivelmente[14], de um problema NP-difícil.

Dito isso, o máximo que podemos fazer é produzir é uma heurística que procure reduzir o número de mensagens rastreadas o máximo possível. A **heurística 1** caracteriza concorrência em fronteiras que podem ser detectadas em tempo de execução e com consumos de memória e tempo relativamente baixos. Além disso, a redução no tamanho dos traços é considerável para aplicações típicas (da ordem de 80%). Uma maneira razoavelmente eficiente de implementa-la é fazendo uso de *vector clocks* [7, 15].

3.5 PREDICADOS E PROPRIEDADES

Além do replay de execução, uma classe de problemas que gostaríamos de resolver quando depurando um sistema distribuído diz respeito à detecção de violações de propriedades e condições formuladas como predicados sobre estados globais.

Um predicado global é uma função booleana *f(Y)* cujo domínio é definido no espaço de cortes consistentes do sistema distribuído em consideração. Existem duas classes de predicados que nos interessam:

1. **Predicados estáveis**: São aqueles predicados que, uma vez que se tornem verdadeiros (falsos), permanecem verdadeiros (falsos) durante todo o restante do curso da execução. Exemplos de predicados estáveis são condições de deadlocks[15], terminação ou perdas de token [2].

2. **Predicados instáveis**: São predicados que alternam entre valores verdadeiros e falsos durante a execução do programa. Exemplos existem aos montes: "o número total de conexões, considerando todos os elementos do cluster, é maior do que 5?", "a impressora X está sem papel?", "há, no sistema todo, dois programas que estejam simultaneamente em suas regiões críticas?", etc.

As propriedades, por sua vez, podem ser classificadas como:

1. **Propriedades de segurança** (*safety properties*): Uma execução cumpre a uma propriedade de segurança especificada por um predicado ***p*** se, e somente se, ***p*** mantiver-se falso (verdadeiro) durante toda a extensão da execução. Exemplos de propriedades de segurança incluem exclusão mútua, ausência de *deadlocks*

[14] Francamente, até agora não fui capaz de compreender onde reside a diferença tão gritante entre os dois problemas – a mim parecem deveras similares.
[15] Isso quando o sistema não possui algum esquema para quebrar deadlocks, mas em geral os esquemas que quebram deadlocks precisam primeiramente detectá-los (quem vem primeiro, o ovo ou a galinha?).

e corretude parcial (se o programa terminar, então o resultado correto é produzido).

2. **Propriedades de vivacidade** (*liveness properties*): Uma execução cumpre a uma propriedade de vivacidade especificada por um predicado ***p*** se, e somente se, ***p*** for verdadeiro (falso) em ao menos um corte consistente do sistema. Um exemplo de uma propriedade de vivacidade é a propriedade de terminação.

A diferença fundamental entre propriedades e predicados é que, enquanto predicados dizem respeito a um estado particular do sistema, propriedades fazem sentido somente quando consideradas em relação a uma seqüência de estados globais (execução distribuída), ou *traces*[16].

Um corolário disso é que só podemos dizer com certeza que uma dada propriedade de segurança foi respeitada se considerarmos execuções finitas. De maneira semelhante, só podemos dizer que uma execução desrespeita uma propriedade de vivacidade se considerarmos execuções finitas. Essa hipótese é perfeitamente razoável dentro dos nossos propósitos.

Predicados e propriedades são formulados em cima de cortes consistentes ou seqüências deles. Algo fundamental para que possamos capturar tais predicados/propriedades é que consigamos, antes de mais nada, "fotografar" os estados globais consistentes do sistema distribuído de alguma forma. Vamos agora discutir, muito superficialmente, algumas questões relacionadas a modelos de rede.

3.5.1 MODELOS SÍNCRONOS DE REDE

Dizemos que um sistema distribuído se encaixa num modelo síncrono de rede quando existe um passo global de sincronia para todos os nodos integrantes. Essencialmente, sistemas distribuídos síncronos operam em turnos, alternando entre passos de computações e passos de comunicação.

Seja M um alfabeto sobre o qual estão definidas as mensagens enviadas num dado sistema distribuído. Num sistema síncrono temos, formalmente, associados a cada nodo *i*:

1. Um conjunto de estados S_i (não necessariamente finito);
2. um conjunto não-vazio de estados iniciais, S_i^0;
3. uma função geradora de mensagens $f_i : S_i \times (viz_i) \rightarrow M \cup \{null\}$, onde viz_i é o conjunto dos nodos vizinhos de *i* e;
4. uma função de transição $t_i : S_i \times M^* \rightarrow S_i$.

Uma execução de um sistema síncrono é sempre dada em turnos, produzindo uma seqüência que assume a seguinte forma:

[16] Trace é aquilo que é gerado por um tracer. ☺

$$C_o, M_1, N_1, C_1, M_2, N_2, C_2, ...,$$

onde cada C_k é uma atribuição de estado e cada M_k e N_k são atribuições de mensagens a canais representando, respectivamente, as mensagens enviadas e recebidas no k-ésimo passo.

Dentro de um modelo síncrono as coisas são, em geral, bem mais simples do que nos modelos que viemos discutindo até agora. Isso porque, essencialmente, essa questão do "passo global" introduz, por debaixo do pano, uma informação de progresso global que pode ser obtida num nível local; isto é, qualquer nodo do sistema sabe, no k-ésimo passo, que todos os outros nodos do sistema estão também executando o k-ésimo passo.

Isso levanta um mundo de possibilidades interessantes. Por exemplo, um nodo que inicia uma operação de comunicação coletiva como um broadcast, por exemplo, sabe que esse broadcast termina após no máximo *n* passos do início da operação.

Para obter uma fotografia do estado global do sistema poderíamos, por exemplo:

1. Eleger um coordenador através de um algoritmo de *flooding* simples
2. Estabelecer, através de um broadcast, que todos os processos devem gravar os seus estados e os estados dos respectivos canais de entrada no j-ésimo passo, onde $j \geq k + n$ e *k* é o passo em que se inicia o broadcast.

Após essa "fotografia", um depurador poderia coletar e colar esses estados "parciais" e produzir um corte consistente do estado do sistema distribuído, sem maiores esforços. Essa solução simples resolve o problema da detecção de predicados estáveis em sistemas distribuídos (mas não dos instáveis).

3.5.2 MODELOS ASSÍNCRONOS DE REDE

Para evitar tornar este texto longo demais, não vamos apresentar o modelo assíncrono em sua totalidade; isto é, não vamos formalizá-lo como fizemos com o modelo síncrono. Isso seria, a nosso ver, uma perda de tempo – todas as explicações e teoremas para modelos síncronos apresentados até agora foram dados, sem prejuízo, em função do modelo simplificado. Uma belíssima referência para modelos assíncronos (que utiliza fortemente a teoria dos autômatos) pode ser encontrada em [4].

Por agora, resta dizer que os nodos num sistema assíncrono não possuem uma noção global de tempo e nem de progresso dos nodos remotos e que, ainda, pode haver uma discrepância arbitrária entre as velocidades de processamento em cada nodo.

O algoritmo de fotografia global para sistemas assíncronos apresenta vários desafios, como por exemplo: precisamos nos preocupar com a consistência dos cortes. "Bem", diria alguém, "poderíamos simplesmente eleger um coordenador que, no momento (arbitrário) da fotografia, gravaria seu estado e iniciaria um broadcast com uma mensagem especial. Cada processo vizinho, ao receber essa mensagem especial,

gravaria seu estado, suspenderia sua execução e propagaria o broadcast para seus vizinhos".

Isso seria ótimo, não fosse o fato que teríamos de suspender o sistema por um intervalo de tempo arbitrário para obter uma fotografia consistente. A idéia é que não tenhamos de interromper o processo de computação ou o overhead incorrido torna-se rapidamente inaceitável, especialmente se o algoritmo for utilizado como um mecanismo de monitoramento de predicados estáveis em sistemas em execução(em outras palavras, se a análise dos traços de execução não for post-mortem).

Devemos, portanto, buscar uma alternativa. O que aconteceria se simplesmente não suspendêssemos o processo que recebe a mensagem especial; isto é, se fizermos somente com que ele grave o seu estado e prossiga no broadcast, sem interrupção? Bem, imediatamente a assincronia acerta nossa cabeça em cheio, já que teríamos de lidar com mensagens que chegam **depois** que um dado processo já gravou seu estado.

A idéia proposta por Chandy e Lamport em [8] é a de que podemos, a princípio, "fazer de conta" que as mensagens que chegaram depois do instante de gravação do estado de um nodo **já estavam** no canal de comunicação no momento em que essa gravação foi efetuada. Essa idéia parte da observação de que precisamos capturar um *estado possível* do sistema, não um estado exato, já que do ponto de vista das perturbações causais, um *estado possível* já é suficiente para respondermos se um dado predicado estável é satisfeito ou não.

Essa última observação só é válida porque a estrutura temporal induzida pelos cortes consistentes de um sistema distribuído forma um *reticulado* [7]. Portanto, o corte C produzido pelo algoritmo, sendo consistente, é limitado superiormente por algum outro corte consistente "real" C_{sup}, de tal forma que $C \leq C_{\text{sup}}$ (C_{sup} é alcançável por C). Portanto, se o predicado estável for verdadeiro em C, ele também será verdadeiro em C_{sup}.

Esperamos que mesmo esta discussão incrivelmente superficial[17] tenha revelado o inevitável: o modelo assíncrono é substancialmente mais difícil de trabalhar do que o modelo síncrono, especialmente quando precisamos formalizar um raciocínio. Infelizmente, produzir implementações eficientes de modelos síncronos em sistemas distribuídos de grande porte é algo comparável a um eventual 13º trabalho de Hércules – teríamos de sincronizar um vasto número de processos num ambiente heterogêneo onde altas latências de comunicação são quase uma constante.

Temos notícia de alguns desbravadores destemidos, no entanto, que aceitaram o desafio ao produzirem uma implementação de um sistema paralelo-síncrono que executa sobre uma grade computacional heterogênea.

3.5.3 O INTEGRADE E O BSP

[17] Modelos para sistemas distribuídos compreende um assunto muito, mas muito vasto. Um excelente livro (do tamanho do CLRS) especificamente voltado para modelos, algoritmos e problemas fundamentais em sistemas distribuídos é apontando em [4].

O Integrade é uma jóia rara. Oferecendo aos usuários primitivas de comunicação baseadas no modelo BSP (Bulk Synchronous Parallel), o Integrade possibilta o uso de uma grade computacional de grande porte sob um paradigma síncrono. O BSP pode ser classificado na categoria dos sincronizadores, que são entidades de software cujo papel é cuidar dos detalhes de sincronização entre processos.

Sincronizadores são plenamente especificáveis no modelo formal. Podemos provar propriedades bastante precisas a respeito de sistemas sincronizados (i.e., que levam em consideração que eles são síncronos por utilizarem sincronizadores) através de modelos obtidos pela composição de autômatos que representam sistemas assíncronos com autômatos que representam sincronizadores [4].

Bem, para os nossos fins, o BSP significa que depurar sistemas paralelos no Integrade é, em termos teóricos[18], algo substancialmente mais simples do que o que vínhamos discutindo até então. Primeiro porque toda aquela problemática associada ao replay de execução nas passagens de mensagens desaparece no âmbito do BSP – se os programas locais forem determinísticos, o BSP garante que todas as mensagens são entregues "no mesmo instante" (não sei se o BSP garante canais FIFO, mas imagino que sim). Segundo, porque observar predicados estáveis torna-se trivial.

Portanto, tudo o que temos de fazer é sincronizar o replay de threads. Como já existem alguns "trace-replayers" prontos para software *multithreaded*, podemos (e vamos) tentar, inicialmente, uma abordagem voltada à reutilização de código.

3.6 PROBLEMAS E MAIS PROBLEMAS

Essa breve exposição de algumas das questões envolvidas serviu para que ganhássemos alguma idéia do tamanho do problema, bem para que entrássemos em contato com algumas das soluções e resultados na área. Não abordamos aqui muitas das questões estudadas, em especial as relativas aos predicados globais instáveis, devido à mais pura falta de tempo. A abordagem desses e de outros problemas fica para trabalhos futuros. Boas referências para o problema dos predicados instáveis podem ser encontradas em [2, 3, 14]

[18] Dizemos "em termos teóricos" porque algo aparentemente simples pode se tornar um pesadelo quando chega a hora de pôr a mão na massa.

:: Um depurador concreto

4. VISÃO E ARQUITETURA GERAIS

Um depurador distribuído se aproxima, em nossa concepção, de um meio termo entre a visão *multithreaded* de sistema distribuído proposta na **seção 2** e o protótipo exposto na **figura 2**.

Essencialmente, nosso depurador distribuído deve possuir três modos principais de execução:

Trace – Neste modo, o sistema distribuído é monitorado e traços de sua execução são produzidos e coletados. Eventuais mecanismos de visualização "ao vivo" da execução poderão ser empregados aqui, desde que eles não interrompam a execução do sistema (e, opcionalmente, gerem pouca interferência). O modo **trace**, seguramente o mais importante do ponto de vista da implementação, cumpre a três propósitos fundamentais:

1. Monitorar a correta execução de um sistema que executa por longos períodos de tempo[19] empregando, para isso, algoritmos de detecção de predicados similares aos discutidos nas seções anteriores (e alguns outros).

2. Reduzir ao máximo, nesse processo de observação, os efeitos de interferência, visando minimizar as possibilidades do aparecimento de Heisenbugs[20].

3. Produzir um conjunto de traços que viabilize o replay de execução.

Debug – No modo debug, o depurador funciona como uma espécie de depurador simbólico estendido (e não gera traços da execução do sistema). Por "depurador simbólico estendido", queremos dizer que será possível, neste modo, instalar *breakpoints* em pontos arbitrários de nodos arbitrários, interromper *threads* em nodos individuais e, no caso de sistemas baseados em objetos distribuídos, rastrear *threads* distribuídos chamadas CORBA adentro[21]. A possibilidade de uso arbitrário das facilidades de controle de fluxo e inspeção de estado incorporadas tornam este modo de uso semelhante ao modo de uso dos depuradores tradicionais - exceto pelo fato que, mesmo neste modo, o nosso depurador conta com algumas extensões semânticas inexistentes nos depuradores simbólicos (rastreio de *threads* distribuídos).

Trace-Replay-Debug – Sem sombra de dúvidas o modo mais interessante do ponto de vista das *features* oferecidas. A idéia é que o depurador funcione como uma espécie de mistura feita entre o protótipo da **figura 2** e o depurador simbólico estendido do *modo debug*. O usuário poderá, aqui, inserir *breakpoints* em pontos arbitrários de uma execução previamente rastreada e, através das facilidades de replay de execução,

[19] Ou por períodos de tempo tão longos quanto pudermos.
[20] Em homenagem ao físico alemão Werner Heisenberg e seu princípio da incerteza [6].
[21] Isto é só uma sugestão inicial. A especificação deste modo é bastante volátil.

reproduzir e inspecionar o estado de cada um dos nodos individuais em qualquer ponto da execução original.

Vamos agora discutir um pouco a respeito da arquitetura de um depurador distribuído. A figura abaixo nos dá uma idéia geral:

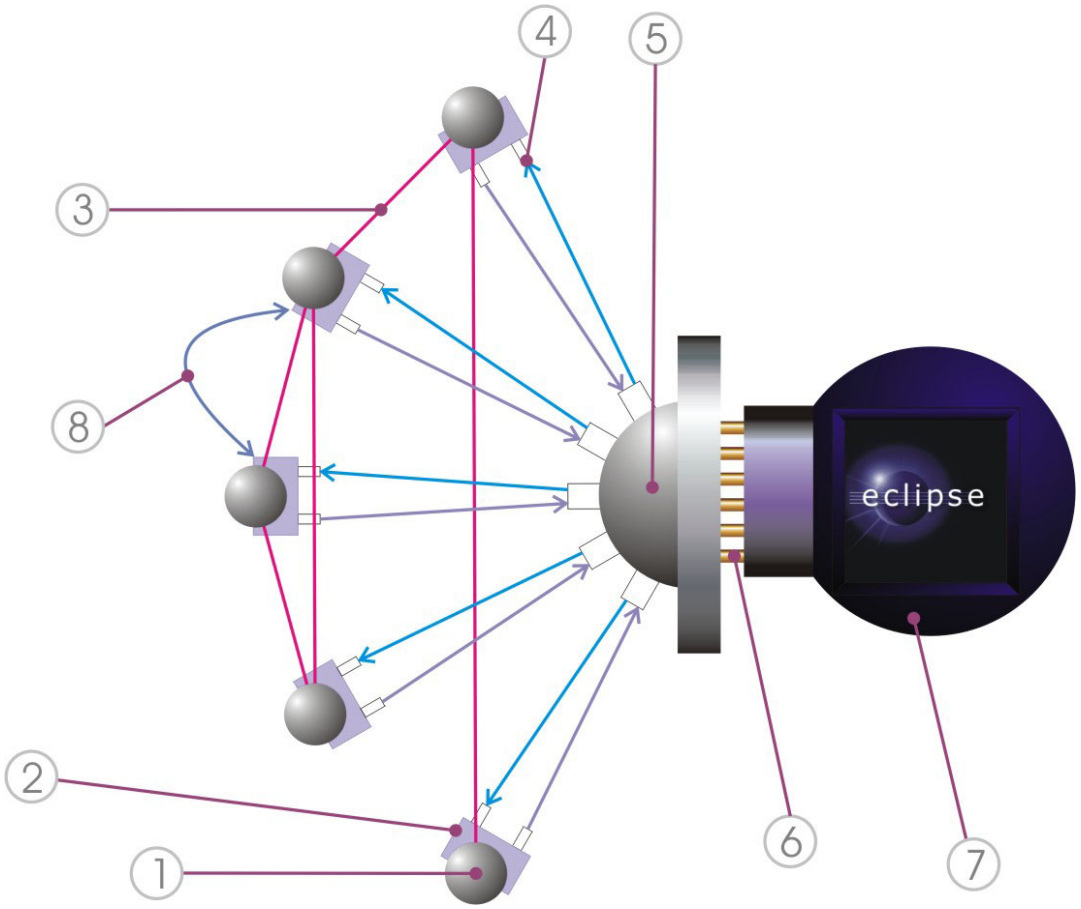

fig. 12 – Arquitetura do depurador distribuído

Alguns dos elementos apresentados acima não fazem parte, no sentido estrito, da arquitetura do depurador, mas fazem-se presentes na figura por ilustrarem certos aspectos importantes das interações do depurador com seu ambiente. Uma breve explicação dos elementos enumerados no desenho segue:

1. **nodo –** nodo do sistema paralelo/distribuído, também chamado na literatura de "elemento de processamento". Pode abarcar desde um simples processador até um sistema completo com memória individual. Para todos os efeitos, consideramos, sem perda de generalidade, que nodos são sistemas não-distribuídos.

2. **agente de depuração local –** é o representante local do sistema de depuração distribuído. É papel do agente local rastrear o estado de processos locais e cooperar com outros agentes locais na execução de algoritmos distribuídos de coleta de dados. Um agente de depuração local pode ser composto por um depurador

simbólico tradicional, modificado para aceitar comandos remotos de (5) e de outros agentes. Via de regra, agentes locais encapsulam os aspectos específicos e voltados para a linguagem de implementação do software que executa no nodo associado.

3. **canal de comunicação do sistema –** representam os elementos de comunicação inter-nodos presentes no sistema. As hipóteses com relação aos canais variam conforme o caso. Em geral, assumimos que canais de comunicação são bidirecionais (canais unidirecionais são bastante incomuns nas redes atuais), confiáveis e FIFO (o protocolo de transporte de escolha é, na maior parte dos casos, o TCP). Essa hipótese a respeito dos canais se justifica melhor quando consideramos o ambiente em que se empregam depuradores distribuídos.

4. **canal de comunicação local-global -** canal lógico que faz as vezes de elo entre o backend de depuração distribuída em (5) (também conhecido como debug engine) e os agentes locais de depuração. Os canais de comunicação do depurador podem compartilhar o mesmo meio físico dos canais de comunicação do sistema (3) - eles encontram-se separados no desenho apenas para frisar que mensagens trocadas entre agentes de depuração não devem interferir, sob nenhuma circunstância, com as mensagens trocadas entre os processos do sistema.

5. **agente de depuração global -** o agente de depuração global é o responsável por coordenar os agentes locais (quando necessário) e por coletar e analisar informações. Isso significa que as ferramentas de agrupamento e análise de traços devem estar concentradas, na medida do possível, neste componente. Existe uma grande flexibilidade quando consideramos o balanceamento de funcionalidades que deve ser estabelecido entre os agentes locais e o agente de depuração global, mas a tendência corre no seguinte sentido:

   a) Os agentes locais executam algoritmos locais e/ou distribuídos de coleta de informações, comunicando-se através do agente de depuração global ou de canais lógicos estabelecidos entre os próprios agentes (8).
   b) O agente de depuração global coleta e compõe as informações geradas localmente, produzindo informações globais sobre aspectos do sistema distribuído (antes não observáveis).

   O agente de depuração global é o grande responsável pela "mágica" que torna a visualização coesa do sistema distribuído possível.

6. **fachada de acesso –** o agente de depuração global disponibiliza uma fachada (API) que cumpre ao propósito de desacoplar a lógica do depurador da lógica da interface22. Eventuais frontends gráficos ou mesmo modo texto deverão comunicar-se com o agente global de depuração através desta interface.

7. **Eclipse –** o depurador será desenvolvido primariamente para uso como plugin do Eclipse. Temos a forte impressão de que um esquema que envolva somente a elaboração de uma fachada (como sugerido em 6) não será suficiente para este frontend em particular; portanto algum código de integração Eclipse-específico é

[22] Essa prática é questionada por alguns autores, mas consideramos-na uma boa prática no nosso caso.

previsto. Toda a parte gráfica e de interações com o usuário será desenvolvida para uso primário na plataforma Eclipse.

8. **canal lógico de comunicação inter-agentes locais –** possivelmente os agentes locais deverão trabalhar, sob algumas circunstâncias, de forma autônoma (i.e. sem a mediação do agente de depuração global), como por exemplo quando executando um algoritmo de fotografia distribuída. O canal lógico de comunicação inter-agentes locais23 representa justamente essa comunicação que se dá entre agentes locais e que não é intermediada por um agente global (sem que, no entanto, deva haver um meio físico dedicado para isso).

Até agora, discutimos inúmeras questões referentes a resultados existentes e ao nosso próprio planejamento com relação à elaboração de um processo e à montagem de um depurador distribuído. Um mundo à parte das dificuldades teóricas ou de design abstrato se apresenta, no entanto, quando nos deparamos com as dificuldades técnicas e de implementação.

Conforme já mencionamos (por alto) em uma seção passada, um depurador distribuído deve lidar com arquiteturas e semânticas bastante heterogêneas. Passamos as últimas vinte páginas delineando os aspectos e compromissos referentes a algumas das funcionalidades básicas propostas na **seção 2**; agora é chegada a hora de apresentar um pouco do material voltado para a implementação de uma ferramenta como a descrita até então.

É claro que, num estudo de seis meses, não é possível chegar em nada definitivo a respeito do assunto, em especial quando falando em implementações e dificuldades técnicas (em geral, as questões de implementação são as últimas a serem levantadas e as últimas a serem resolvidas). Ainda assim, procuramos arranhar um pouco a superfície problema, restringindo o foco a três questões fundamentais (e seus desdobramentos):

1. Depuração de sistemas não-distribuídos e distribuídos baseados em **Java** (por ser uma linguagem interpretada, de ampla aceitação e "fácil[24]" de depurar).
2. Rastreio de *threads* distribuídos **CORBA**, inicialmente restritos a clientes/servidores Java.
3. Análise de possibilidades técnicas para o cumprimento da meta de intrusividade reduzida (que vamos discutir em um instante).

4.1 DEPURAÇÃO JAVA

A especificação da máquina virtual Java provê uma série de facilidades de depuração. Para tanto, a VM disponibiliza interfaces nativas que nos permitem inspecionar e modificar o estado de um programa em execução, trabalhar com abstrações comparáveis a breakpoints e receber as notificações mais diversas[25]. As

[23] Omitido para o restante dos agentes por constituir um grafo completo – uma poluição desnecessária no nosso já complexo desenho.
[24] Mais fácil do que C, pelo menos.
[25] Essa análise de *features* é bem superficial **mesmo** já que, ao menos num primeiro instante, o nosso interesse nas interfaces nativas de depuração da VM Java tende a zero.

interfaces nativas são, no entanto, deveras "cruas" e têm o seu acesso limitado ao universo C/C++. Para evitar enviesar os depuradores Java em C/C++ e para poupar os programadores de terem de escrever os seus próprios mecanismos de conversão inter-linguagens, a Sun elaborou um conjunto padronizado de componentes que nos permite escolher, ao construírmos o nosso depurador Java, qualquer linguagem que forneça uma implementação de sockets TCP.

Claro que, por se tratar de uma especificação da Sun, a linguagem Java é priorizada para o desenvolvimento dos depuradores. A arquitetura de depuração Java, conhecida como JPDA (ou *Java Platform Debugger Architecture*), tem seus componentes principais delineados no seguinte cenário:

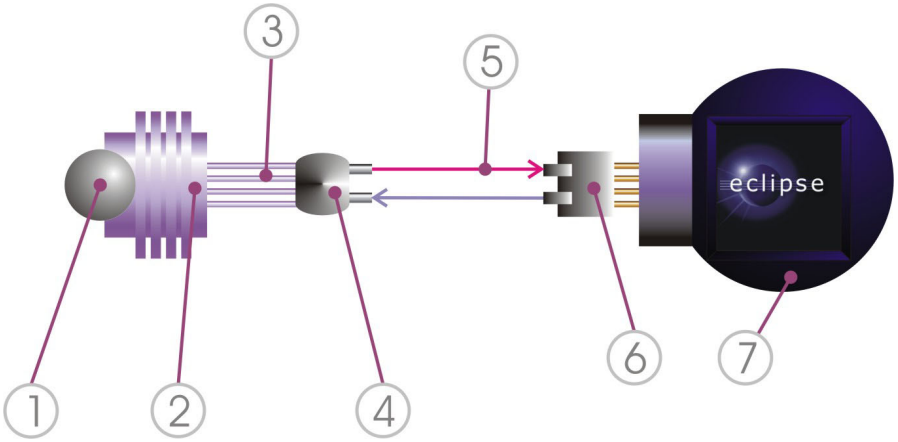

fig. 13 – um cenário de depuração
usando a JPDA (qualquer semelhança com
a nossa arquitetura não é mera coincidência).

1. **Bytecode Java –** bytecode que executa na JVM.

2. **VM Java –** máquina virtual Java (JVM).

3. **JVMDI (Java Virtual Machine Debug Interface) –** Interface nativa exposta pela máquina virtual Java e que pode ser acessada por software C/C++ que interage diretamente com a JVM. Trata-se de uma interface poderosa e bastante complexa, seguindo um estilo "exploradora de entranhas" da JVM.

4. **Debugger back-end –** o debugger back-end é o responsável pelo desacoplamento de linguagem característico na JPDA. Essencialmente, trata-se de um software que interage com a JVM através da JVMDI e implementa um protocolo de depuração orientado a mensagens **(5).**[26]

5. **JDWP (Java Debug Wire Protocol) –** protocolo de depuração de nível de aplicação que estabelece um conjunto de interações e mensagens que tornam o processo de depuração da JVM algo mais simples e independente de linguagem. É bem menos flexível que a JVMDI, mas é o suficiente para mais de 90% dos depuradores.

[26] Infelizmente a Sun não optou por produzir um back-end que "fale" IIOP.

6. **Debugger front-end (+JDI) -** É a peça de software de lado cliente que "fala" JDWP. A Sun provê, a partir do JDK 1.3, uma implementação de uma sofisticada interface de depuração 100% Java, a **JDI** (Java Debug Interface), que não é nada mais do que um "tradutor" Java-JDWP.

7. **Depurador Java –** No lado oposto ao da JVM encontra-se o software que implementa a interface de usuário. Depuradores escritos em Java podem se utilizar da JDI, como é o caso do depurador Java encontrado no Eclipse (dado como exemplo na figura).

Além de flexível e modular a JPDA traz, como resultado de sua arquitetura, uma característica que, para nós, é altamente desejável – depurar uma JVM remota torna-se algo quase tão natural quanto depurar uma JVM local. Se tivéssemos de desenvolver um depurador de sistemas distribuídos que fosse 100% Java, muito provavelmente precisaríamos apenas da JDI.

Nosso estudo da JPDA foi conduzido através do código de programas-exemplo inclusos com o Java SDK 1.4.2. Durante este semestre, o foco foi especificamente voltado para a JDI. A documentação é bastante precária, consistindo majoritariamente de *javadocs* e um punhado mirrado de explicações esparsas. Não há um documento de especificação nem tampouco uma clarificação semântica satisfatória dos diversos blocos que constituem a interface. Felizmente a JDI é bem feita e razoavelmente fácil de entender através de exemplos.

O programa dissecado foi o JDB, uma espécie de GDB para Java. Essencialmente, a JDI apresenta uma estrutura que imita a estrutura de componentes da linguagem e trabalha de forma assíncrona. Construir um depurador Java consiste, em linhas bem gerais, em pedir requisições à JDI e consumir os eventos produzidos, disponibilizados numa fila de eventos. Devido a particularidades no design, no entanto, algumas requisições possuem como pré-condição eventos específicos, o que nos força a criar toda uma infraestrutura de "requisições adiáveis".

Isso tudo está muito abstrato. Vamos dar um exemplo:

Suponha que um usuário deseje instalar um *breakpoint* num ponto específico do código de uma classe. Referências para classes só podem ser obtidas após a carga da classe e, pior do que isso, a consistência da requisição (por exemplo, determinar se o método de fato existe) só pode ser feita quando de posse dessa referência. Isso significa que devemos capturar eventos de carga de classe e analisá-los um a um, cumprindo as requisições pendentes de acordo.

Uma vez que as classes estejam carregadas, no entanto, é tudo bastante simples. Cada "máquina virtual" referenciada nos dá acesso a uma fila de eventos que podem ser consumidos, bem como para um objeto do tipo *EventRequestManager*, que pode ser utilizado para criar novas requisições. Máquinas virtuais podem ser acessadas através de conectores, que são entidades intimamente relacionadas às maneiras pelas quais nos acoplamos a uma JVM:

1. **LaunchingConnector –** Launching connectors servem à maneira mais comum de acesso a uma JVM – o depurador lança a VM depurada.

2. **AttachingConnector –** Permite conectar o depurador a uma JVM já em execução.

3. **ListeningConnector –** A documentação deixa algumas dúvidas, mas parece tratar-se de um conector específico para o caso em que a conexão estabelecida com o depurador parte da JVM depurada.

A grande vantagem do Java nesse quesito vem de toda essa assistência provida pela própria linguagem e pela JPDA. Embora seja possível depurar código C/C++ escrevendo um *wrapper* em cima do GDB, programar numa interface já pronta, testada, projetada para depuração distribuída e ainda em ambiente Java (ideal para integração com o Eclipse) são atrativos não facilmente superáveis.

4.2 DEPURAÇÃO PORTÁVEL, CORBA E INTRUSIVIDADE

No roteiro inicial de estudos, comentamos a respeito de algumas metas de intrusividade no rastreio de *threads* distribuídos que gostaríamos de atingir. Não se trata apenas de *threads* distribuídos, todavia - queremos que o nosso depurador seja aplicável em diversos ambientes, não só numa implementação específica de um *middleware* ou de uma biblioteca de comunicação paralela. Em resumo, queremos reduzir ao máximo as dependências com relação a ambientes de execução específicos, codificando, na medida do possível, com enfoque em interfaces e especificações.

No mundo real, no entanto, a coisa é bem diferente. Conforme discutimos em seções anteriores, rastrear uma execução CORBA é algo que pode ser resumido a rastrear um conjunto de *threads* distribuídos. Um *thread* distribuído poderia ser identificado através de um mecanismo de propagação de contextos ou *tagging* [12, 13] (carregados por cada requisição) - isso nos permitiria dizer por quais componentes um dado *thread* passou. Infelizmente, isso não nos permite dizer **o que** esse *thread* fez – para tanto, precisaríamos observar o estado de cada nodo, eventualmente inserindo *breakpoints* ou *watches* em pontos específicos do código.

Isso nos leva ao problema da ponte *JPDA-JacORB*. Queremos rastrear um "*thread* distribuído". Java não faz nem idéia do que seja isso. Precisamos formular esse conceito no contexto do depurador que, a partir daí, deve coordenar os agentes locais e coletar informações relevantes. Havíamos formulado o seguinte protocolo, que faz uso de *request interceptors* (componentes CORBA que podem ser interpostos no fluxo normal de requisições para implantação de aspectos ortogonais à aplicação[27], como o nosso gancho de depuração):

[27] Num certo sentido, os interceptadores cumprem o mesmo papel dos pontos de interceptação presentes no paradigma orientado a aspectos (AOP). A diferença é que eles não requerem code interleaving. Em compensação, são CORBA-específicos.

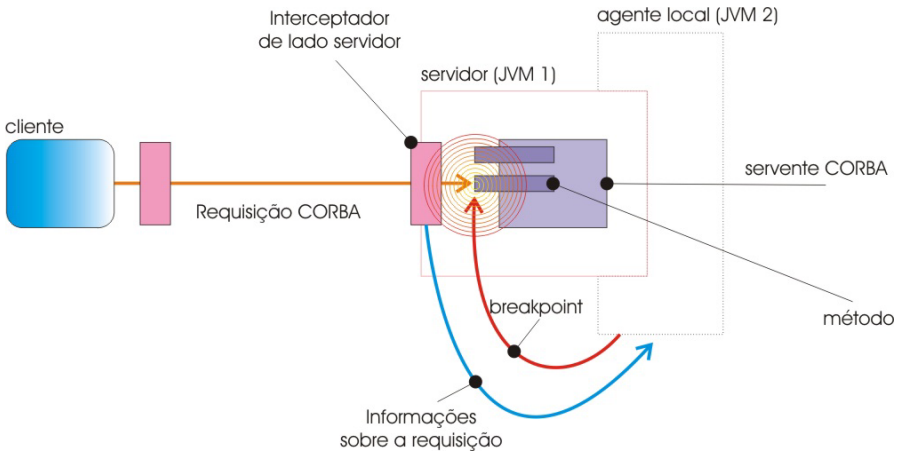

fig. 14 – Linhas gerais para um thread distribuído

Essencialmente, a idéia consiste em instalar um interceptador de lado servidor que, a cada requisição, determina se há ou não um contexto de depuração e, partindo daí, comunica-se com a JVM do agente local e o instrui com parâmetros para que ele possa instalar um *breakpoint* no ponto correto usando a JDI.

Deixando de lado detalhes sórdidos, o problema encontrado aqui é que o componente que despacha as requisições na arquitetura CORBA, o POA (ou *Portable Object Adapter*), não garante (pela especificação) que o *thread* que passa pelo interceptador é o mesmo *thread* que cumpre a requisição no servente. Em outras palavras, se um interceptador determina pelo contexto da requisição que um método X deve ser interrompido e simplesmente pede ao agente local (ao receber uma requisição) para que ele faça isso, pode ser que interrompamos o *thread* errado. Isso leva a um rastreamento errôneo do *thread* distribuído.

Note no entanto que o agente local de depuração possui alguns “superpoderes” sobre a máquina virtual onde a requisição é processada. Poderíamos, em tese, tentar de alguma forma “identificar” o thread que cumpre a requisição inspecionando o estado ou a pilha de execução do próprio POA. Ótima solução, não fosse pelo fato de que POAs possuem implementações ORB-específicas – ao tomarmos um caminho como esse, estaremos amarrando o mecanismo a um ORB, o que é justamente o que não gostaríamos de fazer.

Por pressão dos prazos, decidimos por contar com uma característica muito peculiar do JacORB (nosso ORB de escolha) – no JacORB, o *thread* que passa pelo interceptador de lado servidor é o mesmo *thread* que cumpre a requisição no servente [17]. Com isso, contanto que o POA tenha uma política de tratamento serial de requisições (isso é outra restrição importante e que pode ser inaceitável em certos contextos), conseguimos garantir, ao instalar um *breakpoint* no método da classe servente (através do ponto de interceptação), que o *thread* que é interrompido seja de fato um componente do *thread* distribuído que estamos observando.

A restrição do tratamento serial de requisições pode ser eliminada se marcarmos o *thread* que entra no interceptador com alguma forma de *ThreadLocal*. Dessa maneira, o agente local de depuração poderia decidir se o breakpoint atingido foi atingido pelo *thread* correto ou se ocorreu uma *race*.

Atualmente estamos trabalhando num protótipo cuja funcionalidade principal é rastrear *threads* distribuídos. Acreditamos que muito em breve esse protótipo estará funcionando (ainda que não em integração com o Eclipse).

5. CONCLUSÃO

A depuração distribuída é um campo vasto, que reúne aspectos tanto da teoria de sistemas distribuídos e paralelos quanto das tecnicalidades e tecnologias que os cerceiam. Há muito interesse da comunidade no assunto e, embora existam muitos resultados publicados, a dificuldade fundamental reside em lidar com a heterogeneidade dos sistemas. Produzir um depurador distribuído genérico é uma tarefa que, até o presente momento, nos soa como utópica.

Ainda assim, é possível imaginar um depurador genérico concebido sob uma arquitetura de componentes plugáveis e um modelo de objetos flexível. Isso está longe de nossas possibilidades por agora, no entanto. O levantamento de resultados revela que é possível produzir algo bastante interessante quando num âmbito mais restrito, como no nosso caso (Java/CORBA).

Muitos dos assuntos vistos durante o semestre tiveram de ser deixados de fora por restrições de tempo ou não encontrarem lugar nesta discussão. Pretendemos cobri-los em trabalhos futuros.

5.1 DIFICULDADES

Escrever este relatório de estudos foi algo mais difícil do que inicialmente havíamos previsto em função, principalmente, de detalhes relativos à formalização de certas noções. Sendo os modelos apresentados nos vários artigos consultados bastante heterogêneos, foi para nós um desafio integrá-los numa única discussão. Alguns dos modelos são tão particularmente bizarros que tiveram de ser deixados de fora.

6. O FUTURO

O protótipo deve evoluir para um rastreador de threads distribuídos, que será acrescido, no futuro, de algoritmos de replay de execução e verificação de predicados. Precisamos cobrir ainda um vasto conteúdo em teoria de sistemas distribuídos e tecnologias antes que isso seja possível.

Acreditamos que a arquitetura vá evoluir naturalmente para um esquema de estratégias plugáveis (drivers) devido à imensa heterogeneidade encontrada.

REFERÊNCIAS

[1] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, Julho 1978

[2] A. Tarafdar e V.K. Garg. Predicate control for active debugging of distributed programs. In *Proceedings of the Ninth Symposium on Parallel and Distributed Processing*, pág. 763-769. IEEE, Orlando, USA, Abril 1998.

[3] I. Tomlinson e V. K. Garg. Detecting Relational Global Predicates in Distributed Systems. In *Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, pág. 21-31. USA, Maio 1993.

[4] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[5] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revisão 3.0.3*. Março 2004.

[6] P. Laplante. Heisenberg Uncertainty. In *ACM SIGSOFT Software Engineering Notes*, Outubro 1990.

[7] F. Mattern. Virtual time and global states of distributed systems. In Parallel and Distributed Algorithms: *Proc. Of the International Workshop on Parallel and Distributed Debugging*, págs. 215-226. Elsevier Science Publishers B.V., 1989.

[8] K.M. Chandy e L. Lamport, Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63-75, Fevereiro 1985.

[9] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proc. of the Workshop on Parallel and Distributed Debugging,* págs 1-10. 1993.

[10] R. H. B. Netzer e B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. *Supercomputing '92*, págs. 502-511. Minneapolis, Novembro 1992.

[11] D. Kranzmüller *et. al*. Record/Replay for nondeterministic program executions. In *Communications of the ACM*, págs. 64-67. Setembro 2003.

[12] J. Li. Monitoring and Characterization of Component-Based Systems with Global Causality Capture. *23rd International Conference on Distributed Computing Systems*. Providence, Rhode Island, May 19 - 22, 2003.

[13] J. Li. Monitoring of Component-Based Systems. *HP Tech Reports*. Maio 2003.

[14] V. K. Garg. Methods for Observing Global Properties in Distributed Systems.*IEEE Concurrency*, Vol. 5, No. 4, págs 69-77. Outubro 1997.

[15] M. Raynal. About logical clocks for distributed systems. *ACM SIGOPS Operating Systems Review*. Vol. 26, No 1, págs 41-48. 1992.

[16] RecPlay: a fully integrated practical record/replay system. ACM Transactions on Computer Systems (TOCS). Vol. 17, No. 1. págs 133-152. 1999.

[17] JacORB Team. *JacORB 2.1 Programming GUIDE*, 2004. http://www.jacorb.org.