# Sistemas Distribuídos

Prof. Cláudio F. R. Geyer
Patrícia K. Vargas, Juliano Malacarne,
Silvana C. de Azevedo; Edvar B. de Araújo, Débora N. Ferrari
Instituto de Informática da UFRGS
Cx. Postal 15064 91501-970 Porto Alegre
E-mail: {geyer, kayser, silvanac, malacarn, edvar, nice}@inf.ufrgs.br
http://www-gppd.inf.ufrgs.br

## Resumo

Um ambiente distribuído é um conjunto de processadores interligados por uma rede de interconexão e sem memória compartilhada. A ausência de memória compartilhada exige que a interação entre processadores ocorra de uma forma distinta do ambiente centralizado: ao invés de variáveis ou arquivos compartilhados utiliza-se troca de mensagens. Um sistema distribuído é um sistema projetado para executar em um ambiente distribuído de forma transparente ao usuário. Dentre os diversos tópicos relacionados aos sistemas distribuídos, essa palestra irá se deter em aspectos relacionados ao processamento com o uso de objetos distribuídos.

## Introdução

É inegável a crescente importância dos ambientes paralelos e distribuídos tanto no meio acadêmico quanto comercial. O uso de redes locais e da Internet está amplamente difundido mesmo para uso doméstico. Mas para que tais recursos físicos sejam aproveitados da melhor forma possível é preciso fornecer suporte adequado de software. Existem diversos aspectos relacionados ao controle em ambientes distribuídos. Por ambiente distribuído entende-se um conjunto de processadores interligados por uma rede de interconexão e sem memória compartilhada. Enquanto em um ambiente centralizado a comunicação entre processos ocorre através de variáveis ou arquivos compartilhados, a ausência de memória compartilhada exige que a interação entre processos em processadores distintos seja feita através de troca de mensagens.

Segundo [7], um sistema distribuído é "uma coleção de computadores independentes que parecem ao usuário como um único computador". Essa definição implica hardware formado por máquinas autônomas e software fornecendo a abstração de uma máquina única.

O uso de sistemas distribuídos possui uma série de vantagens em relação ao processamento seqüencial, dentre as quais destaca-se: (a) aproveitamento de máquinas potencialmente ociosas, além de ser mais barato interconectar vários processadores do que adquirir um supercomputador; (b) algumas aplicações são distribuídas por natureza e portanto mais facilmente implementadas nesse tipo de ambiente; (c) em caso de falha

de uma máquina, o sistema como um todo pode sobreviver, apresentando apenas uma degradação de desempenho; (d) é possível ter um crescimento incremental, pois o poder computacional pode ser aumentado através da inclusão de novos equipamentos; (e) sistemas distribuídos são mais flexíveis do que máquinas isoladas, por isso muitas vezes são utilizados até mesmo que não se esteja buscando desempenho.
Porém, também existem algumas desvantagens [7]: (a) pouco software de alto nível disponível para sistemas distribuídos; (b) dificuldades para evitar acesso indevido (segurança); (c) a rede de interconexão pode causar problemas ou não dar vazão à demanda.
Uma vez que a programação distribuída é mais complexa que a programação seqüencial, diversos trabalhos são desenvolvidos tanto com o objetivo de explorar o paralelismo de forma implícita ou automática, quanto com o intuito de tornar a expressão do paralelismo mais simples. Dentre os diversos tópicos relacionados aos sistemas distribuídos, essa palestra irá se deter em aspectos relacionados ao processamento com o uso de objetos distribuídos, principalmente trabalhos relacionados com programação visual e com técnicas de exploração automática. O uso de objetos distribuídos tem crescido com a popularização da linguagem orientada a objetos Java que possui uma série de facilidades para a programação distribuída.
A computação com objetos distribuídos é um paradigma que permite que objetos sejam distribuídos através de uma rede heterogênea e permite que os componentes interajam como se estivessem unificados. Os objetos podem estar distribuídos em diferentes computadores através de uma rede, embora parecendo como se eles estivessem locais dentro de uma aplicação. Uma das preocupações nesse tipo de ambiente é simplificar a programação facilitando ao programador a expressão do paralelismo e da distribuição. Outros aspectos importantes estão relacionados às diferentes formas de otimizar a execução do sistema buscando aumento de desempenho.

## Programação Visual

A programação paralela e distribuída é reconhecidamente mais complexa do que a programação seqüencial. Várias são as razões para que isso aconteça: diversas tarefas independentes, comunicação, sincronização e temporização. Essas dificuldades prejudicam todo o ciclo de desenvolvimento dos programas, refletindo negativamente na qualidade das aplicações concorrentes.

Para amenizar esse problema, diversas ferramentas de apoio ao desenvolvimento têm surgido com características próprias para resolver diferentes tipos de problemas [10, 27]. No caso de aplicações paralelas e distribuídas, as ferramentas predominantes são as linguagens de programação textual, que são ferramentas poderosas, mas que tornam difíceis a compreensão e o desenvolvimento dos programas por representarem de forma unidimensional um problema concorrente que é intrinsecamente bidimensional (processos mais interações).

Com a finalidade de solucionar parte do problema e dar aos programas concorrentes uma representação mais natural e mais próxima do usuário do que da máquina, vem sendo usada na programação concorrente a programação visual. Essa ferramenta tem se mostrado útil neste caso devido à capacidade que os gráficos têm de representar com maior clareza estruturas bidimensionais, possibilitando que relacionamentos entre entidades dispersas no espaço tenham uma descrição mais fácil de ser compreendida pelo ser humano. Estes conceitos foram postos em prática em muitas ferramentas de programação visual para ambientes paralelos e distribuídos, onde a partir da descrição visual do programa é possível gerar código para execução concorrente.

Apesar de a maioria dos ambientes visuais afirmarem o apoio ao desenvolvimento de aplicações distribuídas, pouco existe nessa área e a ênfase maior da programação visual se dá mesmo na programação paralela. Pouco existe para a conexão com servidores externos como HTTP ou SMTP e muito pouco é falado a respeito de conceitos mais próximos de sistemas distribuídos, como tolerância a falhas e aplicações cliente-servidor. Além disso, tais ferramentas carecem de recursos de engenharia de *software*, pois praticamente nenhuma delas utiliza nem ao menos orientação a objetos como subsídio para reuso.

**DOBuilder** DOBuilder [9] é uma ferramenta de programação visual para o desenvolvimento de aplicações com objetos distribuídos em Java. A programação é baseada na manipulação de componentes, com geração de código em Java e execução em ambiente distribuído. Esta ferramenta procura aproveitar as melhores características das ferramentas visuais de programação paralela e distribuída e das ferramentas de programação visual em Java (recursos de visualização para a programação concorrente e características de engenharia de *software*, respectivamente).
Considerando a situação atual dos ambientes visuais, o objetivo da ferramenta DOBuilder é priorizar o atendimento a requisitos que são mais importantes para a programação distribuída, como aplicações cliente-servidor e que interoperam entre sistemas criados por diferentes desenvolvedores. Além disso, busca a facilidade de uso e oferece recursos ao usuário que o auxiliem no desenvolvimento do *software*: reutilização, modularização, extensibilidade e geração automática de código.

Para atingir este objetivo, a ferramenta utiliza uma linguagem textual orientada a objetos para a codificação de baixo nível das aplicações, resolvendo assim boa parte dos problemas de reutilização, modularidade e encapsulamento. Aliada a isto, emprega uma linguagem de programação visual para representar em alto nível as aplicações distribuídas através de um grafo dirigido. Nesse grafo, os objetos distribuídos são representados por nodos e os relacionamentos por meio de arcos que conectam tais nodos. Há vários tipos de relacionamentos possíveis entre os nodos, sendo que os principais são a invocação remota de método, a criação remota de objetos e a comunicação por meio de mensagens explícitas.

Na programação da aplicação, os objetos distribuídos propriamente ditos são os nodos do grafo no programa visual e executam fisicamente em diferentes espaços de memória.

A comunicação é implementada por componentes de *software* acoplados a esses objetos distribuídos. Essa abordagem de componentes permite que a comunicação seja implementada de diversas maneiras, possibilitando ainda que seja alterada e personalizada de acordo com as necessidades de cada aplicação.

O modelo de programação apresenta outros elementos além de objetos distribuídos e portas de comunicação. Estão disponíveis também *locks*, dados globais à rede, portas de chamadas de serviço, serviços e servidores virtuais, entre outros. Todos eles estão organizados numa hierarquia de objetos e definidos em termos de propriedades, métodos e eventos, seguindo o estilo de programação de muitas das ferramentas RAD de programação visual para a plataforma Windows. Esse tipo de definição faz com que os objetos da aplicação sejam totalmente independentes entre si, tornando modular o desenvolvimento da aplicação. A possibilidade de extensão do ambiente é uma conseqüência natural da modularização, permitindo que novos componentes sejam adicionados à ferramenta conforme a necessidade.

Portanto, na ferramenta DOBuilder o usuário terá à sua disposição um modelo de programação orientado a eventos, com definição modular de componentes e geração de código portável entre diversas arquiteturas. Em vista dessas características, uma solução adequada para a implementação deste modelo é se utilizar o ambiente Java, que oferece portabilidade entre plataformas de *software* e de *hardware*, orientação a objetos, programação com eventos e arquitetura de componentes JavaBeans.

Muitas idéias usadas no projeto desta ferramenta foram baseadas em ferramentas semelhantes de programação centralizada, principalmente com o objetivo de resolver deficiências nas ferramentas de programação paralela e distribuída atuais. Estas ferramentas possuem a capacidade de representar a estrutura e relacionamentos das aplicações, mas poucas facilidades em termos de desenvolvimento de *software*. Na sua grande maioria, as ferramentas desse grupo não permitem a reutilização de componentes e nem ao menos sequer utilizam uma linguagem de programação orientada a objetos. Essa dificuldade, quando se trata de aplicações com grande complexidade, é um detalhe que influi consideravelmente no desenvolvimento das aplicações.

Resumindo, o principal objetivo da ferramenta DOBuilder é o desenvolvimento de aplicações onde se possa identificar claramente os relacionamentos entre os diferentes objetos de uma aplicação distribuída. Isso é feito através da programação visual, que juntamente com o conceito de componente, oferece recursos importantes de engenharia de *software*, como reutilização, modularidade e encapsulamento do código dos programas. As aplicações são desenvolvidas parte visualmente e parte textualmente em Java, o que garante a flexibilidade das aplicações.

## Visualização

A visualização da execução é uma ferramenta essencial para auxiliar a depuração e o refinamento de aplicações implementadas utilizando um modelo de programação

distribuída. As ferramentas de visualização procuram demonstrar graficamente o comportamento que a aplicação apresentou durante a execução. Portanto, sem uma ferramenta de análise de desempenho específica para programas distribuídos é difícil entender o desempenho da aplicação com fins de melhorá-la para alcançar melhores resultados.

Atualmente, vem crescendo o interesse pela análise de desempenho e visualização de aplicações Java principalmente devido à popularização do uso dessa linguagem para programação distribuída. Ferramentas já desenvolvidas incluem JaViz [11], JVMPI [. 12], HyperProf [13], ProfileViewer [14], JProbe [15], OptimizeIt [16], Quantify [17] e Jinsight [18]. O ponto forte dessas ferramentas é a análise do código Java, sendo que a maioria apresenta gráficos simplificados de saída. Além disso, a maioria não trata de aplicações do tipo cliente/servidor. Estas deficiências tornam-se relevantes em aplicações com objetos distribuídos, pois o programador não tem uma visão clara de como os vários objetos que compõem a aplicação estiveram distribuídos entre as máquinas durante a execução.

**Visualização no DOBuilder** Um trabalho em desenvolvimento no II/UFRGS consiste em projetar e implementar uma ferramenta de visualização para aplicações distribuídas desenvolvidas em Java [19]. É importante ressaltar que a ferramenta será integrada ao DOBuilder, permitindo a visualização de aplicações desenvolvidas no DOBuilder. Além dessa integração, será possível visualizar aplicações escritas em Java puro.

Durante a execução da aplicação no ambiente distribuído, será realizada a fase de instrumentação. Nessa fase são registrados eventos relevantes como mensagens trocadas, operações de sincronização, computação realizada, e o respectivo instante em que esses eventos ocorreram. Posteriormente, na fase de visualização serão construídos os gráficos para expressar de forma clara e visual o comportamento que a aplicação apresentou durante a sua execução. Serão ressaltadas características importantes a ambientes distribuídos como: comportamento das threads, compartilhamento de recursos, sincronização/bloqueio, utilização da CPU e invocações de métodos remotos (RMI – Remote Method Invocation). O modelo procura possibilitar ao programador visualizar a execução de sua aplicação distribuída, com um maior enfoque em questões do tipo:

- *Aplicações Distribuídas* – nas aplicações cliente/servidor os objetos encontram-se distribuídos e, portanto, executam em várias máquinas. Um aspecto crítico na análise de desempenho para esse tipo de aplicação distribuída é a identificação das partes do programa onde existe um número expressivo de invocação de métodos remotos (RMI).
- *Métodos de Sincronização* – os métodos de sincronização permitem acesso mutuamente exclusivo para proteger objetos. Se a contenção criada devido à sincronização for significativa, pode afetar consideravelmente o desempenho.

## Análise Estática

A análise global tem o objetivo de determinar, estaticamente, informações sobre o comportamento que o programa terá em tempo de execução [4, 5]. Estas características podem ser usadas para otimização, depuração, paralelização e distribuição de programas. A utilização de técnicas de análise global é uma área de pesquisa que tem por objetivo aprimorar a execução de programas.

Muitos dos trabalhos encontrados, na área de análise estática de programas orientados a objetos, destinam-se à análise da hierarquia das classes. A necessidade da análise da hierarquia das classes vem do fato das classes em um programa orientado a objetos representarem os novos tipos, definidos pelo programador. Este tipo de análise pode auxiliar no desenvolvimento de softwares orientados a objetos [6].

Outra característica da programação orientada a objetos é o polimorfismo. Existem vários tipos de polimorfismo, como por exemplo a redefinição de métodos. Na redefinição de métodos um método redefine o método herdado. O método redefinido possui o mesmo nome, valor de retorno e argumentos do método herdado, ou seja, possui a mesma assinatura [3]. Este tipo de polimorfismo é denominado polimorfismo de dados por [2].

Outro tipo de polimorfismo é a sobrecarga. O polimorfismo tipo sobrecarga permite que existam vários métodos com o mesmo nome, porém com assinaturas levemente diferentes, ou seja, com o número de argumentos diferentes, com os tipos dos argumentos diferentes ou com o valor de retorno diferente [3]. Este tipo de polimorfismo é também denominado polimorfismo paramétrico por [2].

No entanto a detecção, estática, do polimorfismo, ou seja, a análise dos diferentes fluxos de execução de um mesmo método, muitas vezes torna-se inviável pela alta complexidade desta tarefa. Devido a este fato, alguns trabalhos não tratam da análise do polimorfismo. Este é o caso do algoritmo básico [1, 2].

Esta dificuldade deve-se ao fato do polimorfismo ser uma característica dinâmica da linguagem. Quando temos uma hierarquia polimórfica, ou seja, uma descendência de classes onde alguma classe filha redefine a implementação de algum método herdado, isto pode implicar diferentes chamadas de métodos.

**DEPAnalyzer** Um exemplo de trabalho que busca o uso de análise estática para auxiliar o desenvolvimento de sistemas distribuídos orientados a objetos é o DEPAnalyzer (DEPendencies Analyzer) [8]. O DEPAnalyzer é um analisador estático de dependências entre as entidades de um programa orientado a objetos, mais especificamente programas Java. As entidades estáticas de um programa Java são as classes, as quais dinamicamente dão origem a conjuntos de objetos.

A hierarquia das classes é analisada e essa informação sobre a hierarquia das classes auxilia na detecção dos conjuntos de objetos instanciados. Saber de qual classe um

conjunto de objetos foi instanciado implica saber qual métodos podem ser acessados por estes e quais mensagens estes podem responder, ou seja, prove uma visão das comunicações possíveis entre as classes.

Existem basicamente duas formas de analisar o polimorfismo: análise de todos os fluxos possíveis ou interação com o usuário. No primeiro caso, seria necessário percorrer todos os fluxos possíveis de um método (fluxos de execução) o que é um problema bastante complexo. Por interação com o usuário considera-se realizar uma análise interativa onde o programador deve indicar de qual classe da hierarquia polimórfica pertence o objeto. Essa segunda alternativa requer que o usuário possua o conhecimento de uma característica dinâmica do programa o que nem sempre é trivial ao programador. Dessa forma, atualmente, o DEPAnalyser desconsidera a possibilidade de polimorfismo sendo esse um tópico a ser estudado futuramente.

Outra preocupação do DEPAnalyzer é identificar se os métodos que estão sendo analisados são de escrita ou de leitura. Isto é importante para auxiliar estaticamente o processo de replicação. Quando um método invocado é de leitura este pode ser replicado sem que o problema da consistência dos dados seja atingido. No entanto, se o método for de escrita, o problema de consistência se torna presente.
O DEPAnalyzer tem a finalidade de gerar informações sobre as comunicações entre os conjuntos de objetos de um programa orientado a objetos e determinar se os métodos das classes, que compõe o programa analisado, são de escrita ou de leitura. As informações de dependência detectam o relacionamento/comunicação entre os conjuntos de objetos. Esta informação pode auxiliar no escalonamento dos objetos, permitindo, se possível, que objetos com grande dependência sejam organizados em uma mesma máquina e objetos com pouca dependência estejam em máquinas distintas. Já as informações sobre o comportamento dos métodos em relação à modificação (escrita) ou acesso (leitura) dos dados pode auxiliar no processo de replicação dos objetos de um programa.

O DEPAnalyzer deverá ser compatível com programas desenvolvidos para ambientes básicos, como o SDK da Sun, e para o ambiente DOBuilder.

## Replicação e Mobilidade

Dois tópicos freqüentemente estudados no contexto de processamento distribuído são mobilidade e replicação. A mobilidade permite mover uma entidade computacional de uma máquina para outra através do sistema. Portanto, em aplicações que usam mobilidade é necessária a preocupação com a integridade das entidades envolvidas. A replicação permite que cópias de uma entidade computacional possam existir no sistema, apresentando-se assim como uma forma de obter disponibilidade. Sendo assim, recentemente surgiram vários trabalhos envolvendo mobilidade e replicação [20, 21, 22].

A replicação envolve a manutenção da consistência entre as múltiplas cópias, isto é, é preciso garantir que todas as cópias possuam o mesmo estado. Por isto, o desempenho

do sistema pode ser diminuído devido ao tráfego incrementado pela garantia de consistência. Desta forma, é favorável utilizar políticas de posicionamento de réplicas que visam a reconfiguração do sistema com base na disponibilidade e desempenho requeridos pela aplicação.

A maioria dos trabalhos relacionando mobilidade ou replicação sobre ambientes de objetos distribuídos é relativamente recente e o interesse pelo tema tem crescido ao longo dos últimos anos. Além disso, até o momento, não há o conhecimento de trabalhos envolvendo mobilidade e replicação de objetos, em um único modelo, com objetivo de prover melhor desempenho ao sistema.

**ReMMOS** ReMMOS (*Replication Model in Mobility System*s) [23, 24] é um modelo de mobilidade e replicação em ambientes de objetos distribuídos. A mobilidade tem como principal vantagem manter a localidade dos objetos que trocam muitas mensagens, diminuindo assim o tráfego na rede. A replicação tem como vantagem permitir que várias cópias de uma mesma entidade computacional residam em diferentes máquinas do sistema, podendo haver acesso simultâneo de diferentes nodos à mesma entidade computacional. Os problemas que este modelo tenta solucionar correspondem a como permitir replicação em um ambiente de objetos distribuídos que suporta mobilidade e a melhorar o desempenho da aplicação de forma controlada, isto é, sempre que for possível. Através da redução do custo de comunicação e a adoção de uma política de gerenciamento de réplicas simples e eficiente, parece haver um desempenho concreto a ser atingido.

A mobilidade fica a cargo do desenvolvedor. A replicação é feita de forma transparente, facilitando o trabalho do desenvolvedor quando este necessita deste recurso em sua aplicação. Assim, este não precisará preocupar-se com o gerenciamento e consistência das réplicas.

A replicação de objetos é feita de foram implícita, utilizando a técnica primário-backup para manter a consistência entre as réplicas. O modelo de replicação é adaptativo ao tipo de aplicação, pois, conforme o comportamento da aplicação, um objeto replicado pode ter suas réplicas criadas ou descartadas. Desta forma, quando a aplicação se comporta de modo a haver predominância de consultas, as réplicas são criadas. Se as atualizações predominarem, as réplicas ociosas do objeto vão sendo descartadas, para diminuir o custo de atualização do objeto replicado, uma vez que este custo cresce à medida que aumentam as réplicas [24]. Entende-se por réplica ociosa a que não está processando, por um determinado período de tempo, consultas locais. No pior caso, se houver predominância no sistema de atualizações sobre os objetos replicados, estes tornam-se objetos não-replicados, pois não existirão mais réplicas associadas a ele.

A mobilidade de um objeto replicado requer uma atenção especial. Como a mobilidade é explícita e a replicação implícita, o desenvolvedor pode mover um objeto que está replicado. Quando um objeto replicado é movido, sua referência deve ser alterada para a nova localização. Desta forma, procura-se manter a transparência quanto à replicação.

O ReMMoS encontra-se em fase de prototipação. O mesmo poderá ser integrado ao DOBuilder. A linguagem usada para implementar o modelo ReMMoS é a linguagem Java, enquanto a mobilidade é implementada usando o sistema Voyager [25]. A escolha deste como sistema base para a mobilidade deu-se por várias razões: (1) é um sistema projetado para utilizar a linguagem Java; (2) várias aplicações distribuídas que suportam mobilidade estão sendo desenvolvidas com Voyager; (3) possibilidade de obter uma versão gratuita (freeware) do sistema; (4) suporte técnico facilitado e (5) não possui replicação de objetos. Comprovando a portabilidade de Java, tem-se usado tanto sistema operacional Conectiva Linux 5.0 quanto Solaris 5.7.

## Conclusão

O aumento em importância das redes, bem como a popularização da Internet, introduziu mudanças no desenvolvimento de aplicações. As aplicações distribuídas estão se estabelecendo quase como um padrão devido às necessidades atuais dos usuários. Aplicações em sistemas distribuídos apresentam diversas vantagens entre elas escalabilidade, disponibilidade e melhor custo-benefício. No entanto, não deve-se esquecer que a programação em ambiente distribuído não é trivial. Simplificar a programação sem comprometer o desempenho é um dos grandes desafios atuais. Desse modo, esse texto apresentou diversas propostas para simplificar essa tarefa.

Como boa parte dos ambientes atuais são distribuídos e heterogêneos, a nova geração de sistemas distribuídos busca conectividade e interoperabilidade. Esses dois aspectos seriam em grande parte facilitados pela utilização de uma linguagem orientada a objetos e portável como Java, tal como adotado pelas propostas apresentadas nesse texto.

Uma das propostas apresentadas busca uma forma visual de expressar a distribuição, gerando automaticamente todo o código relacionado à comunicação e expressão de paralelismo. Outra preocupação foi com a da visualização, de modo a facilitar a depuração. Para otimizar o desempenho das aplicações distribuídas foram apresentadas duas alternativas: a já citada visualização que permite também a detecção de gargalos no sistema, e a análise estática que permite inferir diversas informações sobre a aplicação. Finalmente, um modelo de replicação automática foi proposto com o objetivo de otimizar os acessos de leitura a objetos distribuídos. Note-se que extensões para o tratamento de programas CORBA, escritos para ambientes como o SDK, poderão ser desenvolvidas sem muita dificuldade, ao menos para certos módulos como a análise estática.

## Referências

1. O. Agesen *Constraint-Based Type Inference and Parametric Polymorphism* First International Static Analysis Symposium, 1994 disponível em: http://www.sun.com/research/self/papers/sas94.html
2. O. Agesen *The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism* European Conference on Object-Oriented Programming, 1995 disponível em: http://www.sun.com/research/self/papers/cpa.html

3. A. A. Cesta Tutorial: *A linguagem de Programação Java*. Disponível em: http://www.dcc.unicamp.br/~aacesta.
4. Cortesi et al. *Complementation in Abstract Interpretation.* ACM Transactions on Programming Languages and Systems, Vol.19, No.1, 1997, pp.7-47.
5. D. Dams; R. Gerth *Abstract Interpretation of Reactives Systems*. ACM Transactions on Programming Languages and Systems, Vol. 19, No. 2, 1997, pp.253-291.
6. R Godin et. al. *Design of Class Hierarchies Based on Concept Lattices*, Theory and Practice of Object Systems, Vol. 4(2), 1998, pp. 117-134.
7. Tanenbaum, A. S. *Distributed Operating Systems*. Prentice Hall, 1995.
8. S. C. de Azevedo. *DEPAnalyzer - um analisador estático de programas orientados a objetos*. PPGCC/UFRGS, Porto Alegre, 2000. Dissertação de Mestrado em andamento.
9. J. Malacarne. *Ambiente Visual para Programação Distribuída em Java*. PPGCC/UFRGS, 2000. Dissertação de Mestrado em Andamento.
10. J. Malacarne. *Ambientes de Programação Visual Paralela e Distribuíd*a. Trabalho Individual - TI no. 776 PGCC-UFRGS, 91 pp., Fevereiro 1999.
11. I. H. Kazi et al. *JaViz: A client/server Java profiling tool*. IBM System Journal,v. 39, n. 1, p. 96-117, Mar. 2000.
12. D. Viswanathan & S. Liang. *Java Virtual Machine Profiler Interface*. IBM System Journal,v. 39, n. 1, p. 82-95, Mar. 2000.
13. HyperProf (v. 1.3) – *Java Profile Browser*, http://www.physics.orst.edu/~bulatov/HyperProf.
14. *ProfileViewer*, http://www.inetmi.com/~gwhi/ProfileViewer/ProfileViewer.html
15. *JProbe Profiler*, http://www.in-gmbh.de/english/tools/java/jprobe.htm
16. OptimizeIt! *The Ultimate Java Performance Profiler*, http://www.optimizeit.com
17. *Visual Quantify*, http://sys-com.com/java/reviews/quantify/index.html
18. *Jinsight*, http://www.alphaworks.ibm.com/formula/jinsight
19. E. B. Araújo. *Uma Ferramenta de Visualização para Aplicações Distribuídas em Java*. PPGCC/UFRGS, 2000. Dissertação de Mestrado em andamento
20. J. Kleinöder; M. Golm. *Transparent and Adaptable Object Replication Using a Reflexive Java*. Computer Science Department. University of Erlangen-Nürnberg. Erlangen, Germany. September, 1996. (technical report).
21. C. Ionitoiu et al. *Replicated Objects with Lazy Consistency* Second International Workshop, ECOOP'96. *Proceedings...* Politechnica Univeristy of Timisoara.
22. D. Ratner; G. J. Popek; P. Reiher. *The Ward Model: A Scalable Replication Architecture for Mobility*. (technical report)
23. D. N. Ferrari. *Um Modelo de Replicação em Ambientes que Suportam Mobilidade*. PPGCC/UFRGS, 2000. Dissertação de Mestrado em Andamento.
24. D. N. Ferrari; P. K. Vargas, C. F. R. Geyer. *ReMMoS - Um Modelo de Replicação em Ambientes que Suportam Mobilidade de Objetos*. Anais... I Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD 2000), 2000.
25. ObjectSpace Voyager Core Tecnology 3.0 Disponível em http://www.objectspace.com
26. Bal, Henri E. et al. Orca: *A Language for Parallel Programming of Distributed Systems*. IEEE Transactions on Software Engineering, New York, v. 18, n. 3, Mar 1992, p. 190-205.
27. Skillicorn, D. B., Talia, D. *Models and Languages for Parallel Computation*. ACM Computing Surveys, New York, v.30, n.2, p.123-169, June, 1998.