Uma Plataforma para Suporte à Evolução de Sistemas Distribuídos

Alexandre Sztajnberg
IME-UERJ/COPPE-UFRJ
Rua São Francisco Xavier, 524 – 6º andar
22550-000, Rio de Janeiro RJ
alexszt@ime.uerj.br

Marcelo Lobosco **Orlando Loques**
Computação Aplicada e Automação – CAA
Universidade Federal Fluminense – UFF
Rua Passo da Pátria, 156
24210-240, Niterói, RJ
{lobosco, loques}@caa.uff.br

Abstract

Separation of concerns is an approach intended to separate the functional from the non-functional requirements of the applications. In this work we present and briefly evaluate four contemporary proposals for its implementation on the context of modern distributed systems: computational reflection, compositional-filters, aspect-oriented programming and configuration. Among them, configuration was selected because it fits naturally to the separation of concerns goal and helps to cater for system evolution. A prototype implementation for a Java based configuration platform, supporting system evolution, is also here presented.
Keywords*: separation of concerns, computational reflection, compositional-filters, aspect-oriented programming, configuration, Java, distributed systems.*

Resumo

Separação de interesses é uma estratégia que procura separar os requisitos funcionais dos requisitos operacionais das aplicações. Neste trabalho apresentamos e avaliamos quatro propostas atuais para sua implantação no contexto dos sistemas distribuídos: reflexão computacional, filtros de composição, programação orientada a aspectos e configuração. Dentre estas, a configuração mostrou-se mais atrativa para a finalidade, por permitir um tratamento natural da separação de interesses e também por facilitar a descrição e a evolução dinâmica dos sistemas aplicativos. Uma plataforma básica de configuração para suporte à evolução de sistemas distribuídos, baseados em Java, é também apresentada neste trabalho.
Palavras-chaves*: separação de interesses, reflexão computacional, filtros de composição, programação orientada a aspectos, configuração, Java, evolução dinâmica, sistemas distribuídos.*

1 - Introdução

Um dos grandes problemas apresentados durante a fase de implementação do ciclo de vida de desenvolvimento do *software* é a dificuldade em se separar o que o sistema deve fazer (definido pelos requisitos funcionais) daquilo que ele deve atender (definido pelos requisitos operacionais). Este problema, conhecido como separação de interesses (*separation of concerns*), leva quase sempre a uma codificação confusa, onde trechos de código que exprimem a funcionalidade da aplicação se entrelaçam com os trechos de códigos associados aos requisitos operacionais que devem ser atendidos[6]. Isto cria dificuldades para a manutenção do sistema, durante seu ciclo de vida, e para o reaproveitamento do código em outras aplicações: por melhor que tenha sido a fase de projeto do sistema, ao implementá-lo acaba-se misturando o código dos dois domínios a serem atendidos.

Quatro propostas atuais, reflexão computacional [3], filtros de composição [1], programação orientada a aspectos [4, 6] e configuração [5, 7, 8, 9], mostram-se úteis como ferramentas para implementação de sistemas que buscam separar requisitos funcionais e operacionais Na seção 2, descrevemos o modelo de reflexão computacional; na seção 3 tratamos dos filtros de composição; a seção 4 aborda o conceito de configuração, enquanto na seção 5 é descrito o modelo de programação orientada a aspectos. Na seção 6 são apresentadas algumas conclusões sobre estas propostas. Finalmente na seção 7 mostramos uma implementação inicial de uma plataforma para suporte à evolução (mudanças dinâmicas) de sistemas distribuídos, baseados no paradigma da configuração. A seção 8 apresenta propostas de trabalhos futuros.

2 – Reflexão Computacional

Neste modelo, os aspectos funcionais e não-funcionais são separados com o uso de objetos-base e meta-objetos. Os objetos-base implementam em seus métodos a funcionalidade da aplicação. Já os meta-objetos implementam os procedimentos de controle que determinam o comportamento dos objetos-base a ele associados. Os objetos-base podem não estar cientes da existência dos meta-objetos, sendo que também não é obrigatório que todo objeto tenha um meta-objeto a ele ligado. As chamadas aos métodos do objeto-base são desviadas de modo a ativar meta-métodos, permitindo desta forma supervisionar e modificar o comportamento do código funcional residente no objeto-base ou ainda adicionar novas funcionalidades a seus métodos. A execução no nível base é suspensa até que o meta-objeto termine a computação. Isto permite que o meta-objeto possa ter total controle das atividades do objeto base. A Figura 1, retirada de [3], ilustra o modelo.

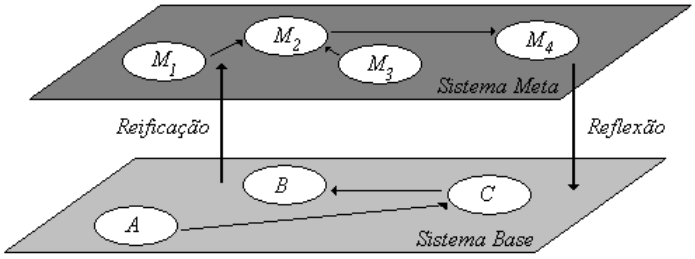

Figura 1- Objetos A, B e C implementam a solução de um problema específico. Objetos meta-nível M_1...M_4 controlam a execução dos objetos A, B e C e implementam parte de seu comportamento.

3 – Filtros de Composição

O modelo de filtros de composição estende o modelo de orientação a objetos convencional. Seu modelo de objetos pode ser apresentado como duas partições separadas: a primeira suporta o modelo convencional de objetos, que é envolvido por uma segunda camada encapsuladora que contém extensões, que são feitas usando os chamados filtros de composição. A primeira parte, também chamada de implementação, é o núcleo do objeto; a segunda parte, chamada de interface, é usada para a gerência das mensagens que entram e saem do objeto. O modelo resultante pode ser visto na Figura 2, retirada de [1]:

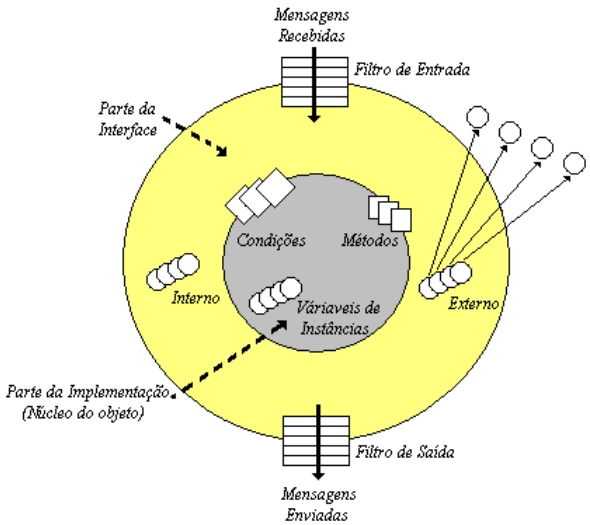

Figura 2– Os componentes do modelo de filtros de composição

A interface recebe as mensagens que chegam, processa-as e então opcionalmente as repassa para a implementação, onde levam à execução de métodos. Esse processamento é feito pelos filtros de composição: os filtros de entrada lidam com as mensagens recebidas pelo objeto, enquanto os filtros de saída lidam com as mensagens que são enviadas pelo objeto. Considerando que todas as atividades no modelo de computação orientado a objetos são executadas através de invocação de mensagens, os filtros podem virtualmente controlar todo aspecto da computação orientada a objetos. Assim, os filtros de composição aplicam uma forma de reflexão computacional nas mensagens.

4 - Configuração

Segundo este paradigma, uma aplicação pode ser construída através da interligação de componentes e conectores. Para este fim é usada uma linguagem de configuração (textual ou gráfica)[9]. Os componentes, ou módulos, podem corresponder a unidades básicas de linguagens de programação, como classes e métodos, ou mesmo a outros objetos, como tabelas e arquivos. Cada componente possui uma interface (assinatura) e uma especificação, que define sua funcionalidade. Já os conectores são responsáveis pelas ligações entre módulos, tendo também o papel de mediadores em suas interações. Pode-se citar como exemplo de conector o mecanismo de RPC (*Remote Procedure Call*), responsável pelo envio de pedidos de computação do cliente para o servidor e pela devolução das respostas do servidor para o cliente, controle de erros de transporte e também pelo empacotamento e desempacotamento dos eventuais parâmetros e respostas.

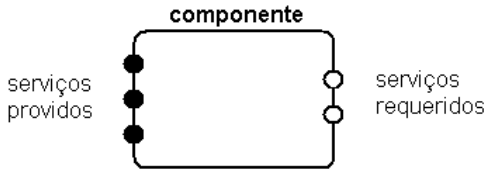

Figura 3 - Componente

Os módulos são projetados de forma que as interfaces pelas quais eles podem se comunicar com outros módulos sejam compatíveis com os conectores (Figura 4). Diferentes conectores são disponíveis para o suporte de diferentes técnicas de comunicação (por exemplo, chamada remota de procedimento, *pipelines*, *sockets*, CORBA, etc.), definindo regras sobre o tipo de interface com a qual eles trabalham, garantias sobre a interação, compromissos sobre a mesma como ordenação ou desempenho, dentre outras.

Em princípio, o código do componente independe do tipo de conector utilizado, o que torna possível sua substituição em tempo de execução, permitindo adequar dinamicamente a aplicação às necessidades de qualidade de conexão entre componentes.

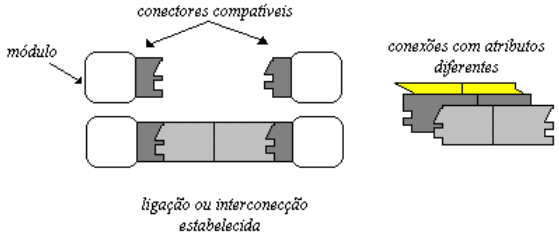

Figura 4 - Módulos independentes e módulos interconectados

5. Programação Orientada a Aspecto (*Aspect-Oriented Programming*)

O paradigma de programação orientada a aspectos (AOP) procura solucionar o problema da separação de interesses através do uso de diferentes linguagens, cada qual especializada no atendimento de uma classe específica de requisito ou aspecto. A partir deste ponto um pré-compilador, denominado a*spect weaver*, combina as descrições dos códigos dos diferentes aspectos, gerando um único código para a aplicação (Figura 5). Do ponto de vista do programador, o requisito funcional (denominado nesta abordagem de componente) é programado independentemente de cada requisito operacional (denominado de aspecto), apesar do código gerado pelo pré-compilador entrelaçá-los.

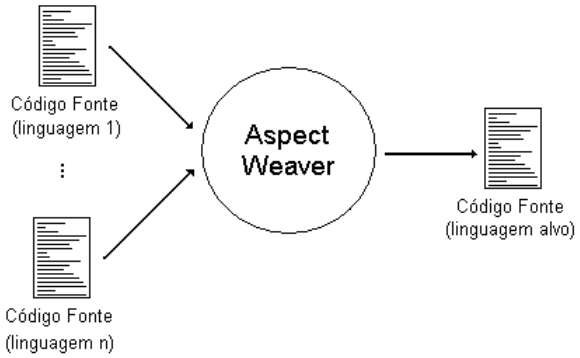

Figura 5 – O papel do *aspect weaver*

6 – Escolhendo um Paradigma para a Implementação da Plataforma

Dentre as razões que tornam atrativo o uso de configuração na busca de soluções para o problema da separação de interesses, destacam-se:

- *a passagem da análise para a implementação é mais natural*, uma vez que este paradigma oferece correspondência direta entre o projetado e o implementado, dada as estruturas modularizadas tanto do modelo gerado durante a análise quanto do modelo do paradigma de configuração;
- *o entendimento do sistema como um todo é facilitado*. A linguagem de configuração usada para a descrição do sistema cria a possibilidade de verificação da correção do mesmo e a representação gráfica tende a facilitar sua compreensão;
- *é natural a separação dos interesses*. Tanto os componentes quanto os conectores podem implementar requisitos específicos do sistema;
- *possui poder computacional equivalente à reflexão, aos filtros de configuração e à programação orientada a aspectos;*
- *em geral, a configuração mostra-se mais flexível e genérica em relação as demais abordagens*. Parte dos sistemas reflexivos, os filtros de composição e a programação orientada a aspectos não permitem mudanças dinâmicas nos sistemas e são dependentes da utilização de linguagens especiais de programação, diferentemente da configuração que em princípio só depende de um suporte de execução razoavelmente simples.

7 – Implementação

Esta seção apresenta a implementação inicial de uma plataforma para suporte à evolução de sistemas distribuídos, baseados no paradigma da configuração. Para sua implementação, foi escolhida a linguagem de programação Java, por sua portabilidade e por oferecer suporte à mobilidade de código.

7.1 - Arquitetura

Um esboço da arquitetura da plataforma pode ser visto na Figura 6. O cenário é o de um conjunto de estações interligadas por rede.

Cada estação implementa um suporte local para configuração, denominado `configurador local`, que é responsável por oferecer, ao resto da plataforma, a abstração de módulos e conectores. Um `configurador global` oferece serviços de configuração não restritos a uma única estação. Através desta camada, módulos e conectores podem ser instanciados em qualquer estação, e suas interligações estabelecidas mesmo que ultrapassando os limites das respectivas estações locais. O `gerente` oferece uma interface para o ambiente de execução, onde as ações de configuração podem ser comandadas.

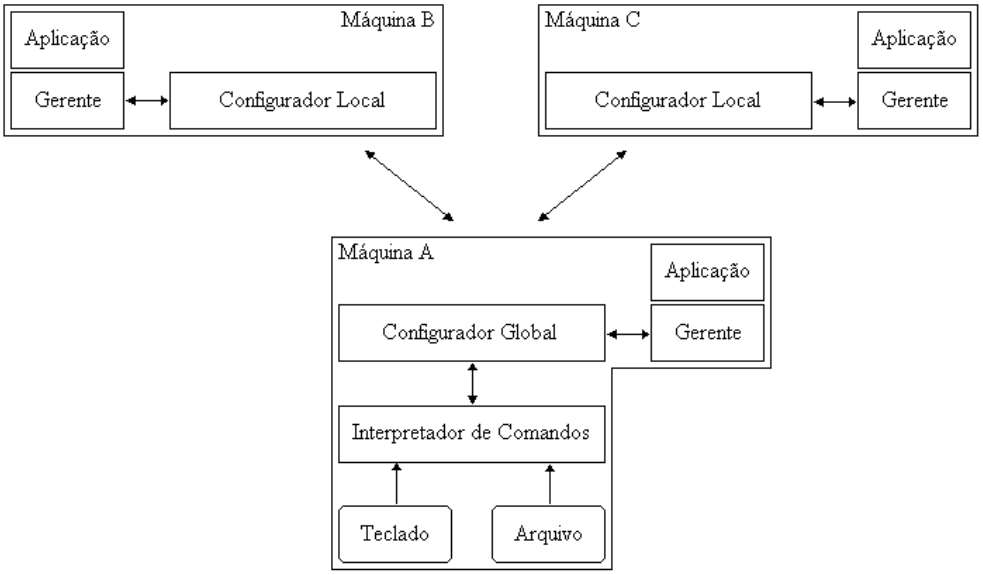

Figura 6- Arquitetura do Sistema

O usuário utiliza comandos de configuração para configurar sua aplicação. Estes são lidos de arquivo ou digitados diretamente pelo teclado. Um interpretador de comandos funciona como um verificador da sintaxe dos comandos. Após a verificação sintática, este comando é repassado ao configurador global, que aciona o gerente local ou mesmo o configurador local de outra máquina para sua execução. O configurador local apenas recebe pedidos do configurador global e os repassa para o gerente, que em última instância é o responsável pela execução de comandos.

7.2 - Primitivas

As seguintes primitivas, implementadas pelo módulo gerente, são utilizadas para a configuração das aplicações:

```
boolean instantiate (String className)
```

Método que tem como objetivo carregar localmente a classe `className`, criando-se uma nova *thread* para sua execução. Aqui utilizou-se a facilidade de migração de código oferecida pela linguagem Java: ela permite ao programador, através de uma classe denominada `classLoader`, configurar o modo como devem ser carregados os *opcodes* (espécie de código objeto Java) que são utilizados por seu aplicativos. Estendeu-se aqui o `classLoader` primordial, de forma a torná-lo apto a carregar estes *opcodes* a partir de um diretório padrão ou mesmo de uma máquina remota.

```
boolean link (String className, String conector)
```

Método que tem como objetivo criar uma ligação entre dois módulos. Isto é feito através da renomeação da classe `conector`, que passa a se chamar `className`. Por exemplo, se deseja se ligar os módulos `A` e `B` utilizando o conector `C` (Figura 7-a), o configurador global instancia o módulo `C` na máquina onde se encontra `A`, mas troca seu nome de `C` para `B'` (Figura 7-b). Assim, quando `A` chama um método de `B` na realidade estará invocando `C` (Figura 7-c). Cabe a este implementar uma interface totalmente compatível com `B`, de forma a evitar que `A` chame um método que exista em `B` mas que não tenha sido mapeado em `C`. Por sua vez, `C` deverá também invocar `B`, usando, por exemplo, *sockets*, RMI ou CORBA. O módulo `C` divide-se em duas partes, aqui denominadas conector cliente (`CC`) e conector servidor (`CS`). `CC` é a implementação do conector que fica instanciado no mesmo gerente onde se encontra o módulo cliente. `CS` é a implementação do conector que fica instanciado no mesmo gerente onde se encontra o módulo servidor.

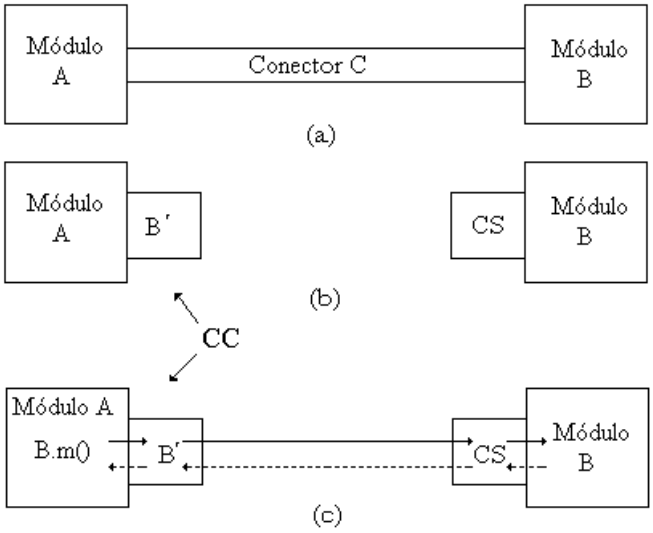

Figura 7 –Conexão entre módulos

A implementação da primitiva para ligação se faz em dois passos. No primeiro, o módulo conector cliente (CC) é transportado para o configurador onde se encontra o cliente. Antes que o módulo seja registrado junto a Máquina Virtual Java, o nome de sua classe é mudado para o nome da classe do módulo do servidor, de forma que toda chamada feita do cliente para o servidor transforma-se, na realidade, em uma chamada do cliente ao módulo cliente do conector. Para realizar esta mudança no nome da classe, foi utilizada no `classLoader` uma biblioteca para a manipulação de *opcodes*.

No segundo passo, é instanciado o módulo conector servidor (CS) através do configurador residente onde se encontra o servidor. Ele deverá, em última instância, receber as chamadas oriundas do conector cliente, fazer as chamadas ao servidor e retornar os resultados.

Outras primitivas para iniciar, bloquear, continuar e terminar a execução de *threads* também são disponíveis.

7.3 - Exemplo

Como exemplo apresentamos uma aplicação simples do tipo produtor-consumidor. O produtor produz dez números inteiros, que são consumidos pelo consumidor. Antes de mais nada, vejamos o módulo cliente (Figura 8).

```
 1 package producer_consumer;
 2 public class produtor implements Runnable {
```

```
 3   public void run () {
 4     int i;
 5     consumidor cons = new consumidor();
 6     for (i=1;i<=10;i++) {
 7       if (cons.consome(i))
 8         System.out.println ("Produtor produzindo item " + i);
 9       else
10         System.out.println ("Erro!");
11     }
12     System.out.println ("Fim da Producao");
13   }
14 }
```

Figura 8 – Código do Cliente

A única exigência para o código do cliente é que ele implemente a interface `Runnable` da própria linguagem Java. Isso é feito através da cláusula `implements Runnable` na linha 4 e do método `run()`, que deve ser implementado.

O código do servidor (Figura 9) possui exigências semelhantes:

```
 1 package producer_consumer;
 2 public class consumidor implements Runnable {
 3   public void run () {}
 4   public boolean consome (int i) {
 5     System.out.println ("Consumindo item " + i);
 6     return true;
 7   }
 8 }
```

Figura 9 – Código do Servidor

Cabe ao módulo conector cliente (Figura 10) implementar uma interface semelhante ao do servidor:

```
 1 package conector_socket;
 2 public class conector_socketClnt {
 3   public boolean consumer (int i) {
 4     //código do socket
 5   }
 6 }
```

Figura 10 – Código do conector (cliente)

O código do módulo conector servidor será responsável pela invocação do método correspondente do servidor e retorno da sua resposta. Como o código do cliente e do servidor, também deve implementar a interface `Runnable` (Figura 11).

```
 1 package conector_socket;
 2 public class conector_socketServ implements Runnable {
 3   public void run () {
 4     // código do socket
 5   }
 6 }
```

Figura 11 – Código do conector (servidor)

Como anteriormente mencionado, o código da aplicação não faz qualquer menção ao tipo de conexão a ser utilizada, tornando possível a escolha em tempo de execução do tipo de conector.

Para configurar a aplicação, os seguintes comandos são passados na linha de comando do sistema:

```
 1 Sistema - Versao 1.0
 2 Entre com o comando no prompt
 3 Digite h (ou H) para ajuda
 4 > instantiate producer_consumer.produtor at camboim
 5 Ok.
 6 > instantiate producer_consumer.consumidor at melancia
 7 Ok.
 8 > link producer_consumer.produtor producer_consumer.consumidor with
conector_socket.conector_socket
 9 Ok.
10 > start producer_consumer.consumidor at melancia
11 Ok.
12 > start producer_consumer.produtor at camboim
13 Ok.
```

Figura 12 – Comandos de Configuração

8 - Trabalhos Futuros

Um trabalho em andamento investiga a possibilidade de criação de conectores genéricos em Java. Como descrito em 7.2, o módulo conector cliente deve possuir, no mínimo, um subconjunto de métodos idênticos aos implementados pelo servidor. Nossas investigações procuram alternativas para que a interface do conector seja independente da interface do servidor. Essa característica traria maior flexibilidade ao sistema, uma vez que um conector que implementasse um tipo particular de comunicação, serviria para ligar qualquer par de módulos, independentemente do tipo de suas interfaces. Em [7] esse objetivo foi alcançado num contexto de configuração baseado em Corba.

Espera-se ainda prover, a nível de linguagem de configuração, primitivas para sincronização que permitam ao programador de aplicações um nível mais alto de abstração, quanto do tratamento de aspectos de sincronização. Mecanismos para suporte à exclusão mútua e guardas (expressões booleanas que devem ser testadas como pré-condição para que um método seja executado) deverão também ser suportados. Desta forma, alcançariamos o poder de expressão do modelo AOP (seção 5), sem necessidade de usar linguagens especiais de programação.

9 – Bibliografia

1. BERGMANS, Lodewijk M. J. "*The Composition-Filters Object Model*". Artigo derivado do capítulo 2 de "*Composing Concurrent Objects*", PhD thesis, University of Trese, junho de 1994.
2. BIRMAN, Kenneth P. "*Building Secure and Reliable Network Applications"*. Manning Publications, 1996.
3. GOLM, Michael. "*Design and Implementation of a Meta Architecture for Java*." Diplomarbeit im Fach Informatik, Friedrich-Alexander-Universität Erlangern-Nürnberg, janeiro de 1997.
4. KICZALES, Gregor; *et alli*. "*Aspect-Oriented Programming*". Xerox Palo Alto Research Center, PARC Technical Report, fevereiro de 1997.
5. KRAMER, Jeff; *et alli*. "*Configuring Distributed Systems*". Proceedings of 5th. ACM SIGOPS on Models and Paradigms for Distributed Systems Structuring, setembro de 1992.
6. LOPES, Cristina V.; KICZALES, Gregor. "*D: A Language Framework for Distributed Programming*". PARC Technical Report, fevereiro de 1997.
7. LOQUES, Orlando; *et alli*. "*A Configuration Approach for Distribuited Object-Oriented System Customization*". Proceedings of Third International Workshop on Objected-Oriented Real-Time Dependable Systems, fevereiro de 1997.
8. SHAW, Mary; *et alli*. "*Abstractions for Software Architecture and Tools to Support Them*". IEEE Transactions on Software Engineering, Volume 21, Nº. 4, abril de 1995.
9. SZTAJNBERG, Alexandre; LOQUES, Orlando. "*Configuração e Reflexão Computacional em Aplicações Multimídia Distribuídas*". 15º Simpósio Brasileiro de Redes de Computadores, março de 1997.