Configuração de Sistemas Distribuídos Baseada em Java

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

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

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

Abstract

Separation of concerns is an approach intended to separate the functional from the non-functional requirements of the applications. Four contemporary proposals, computational reflection, composition-filters, aspect-oriented programming and configuration are being used for its implementation on the context of distributed systems. Among them, configuration was selected because it fits naturally to the separation of concerns goal and helps to cater for system evolution. In this work we present a prototype implementation for a Java based configuration platform, supporting system evolution.
Keywords*: separation of concerns, computational reflection, composition-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. Quatro propostas atuais, reflexão computacional, filtros de composição, programação orientada a aspectos e configuração, estão sendo utilizadas para sua implantação no contexto dos sistemas distribuídos. 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. Neste trabalho apresentamos uma plataforma básica para configuração de sistemas distribuídos baseada em Java, dando também suporte à evolução (mudanças dinâmicas) das aplicações.
Palavras-chaves*: separação de interesses, reflexão computacional, filtros de composição, programação orientada a aspectos, configuração, Java, evolução dinâmica, sistemas distribuídos.*

1 - Introdução

Separação de interesses é uma estratégia que procura separar os requisitos funcionais dos requisitos operacionais das aplicações. Em [8], apresentamos e avaliamos quatro propostas atuais para sua implantação no contexto dos sistemas distribuídos: reflexão computacional[2], filtros de composição[1], programação orientada a aspectos[6] e configuração[5,7]. 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.

Segundo o paradigma da configuração, uma aplicação pode ser construída através da interligação de componentes e conectores. Os componentes, ou módulos, descrevem a funcionalidade de uma aplicação. Já os conectores são responsáveis pelas ligações entre módulos. A configuração da aplicação se faz através do uso de uma linguagem de configuração, seja ela textual ou gráfica[9].

Apresentamos neste trabalho a implementação inicial de uma plataforma para suporte à evolução (mudanças dinâmicas) de sistemas distribuídos, baseada no paradigma da configuração.

2 – Implementação

Esta seção apresenta a implementação inicial de uma plataforma para suporte à evolução de sistemas distribuídos, baseada no paradigma da configuração. Para sua implementação, foi escolhida a linguagem de programação Java, dada sua característica de portabilidade e por oferecer suporte à mobilidade de código e a programação *multithread*[4].

2.1 - Arquitetura

Um esboço da arquitetura da plataforma pode ser visto na Figura 1. 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.

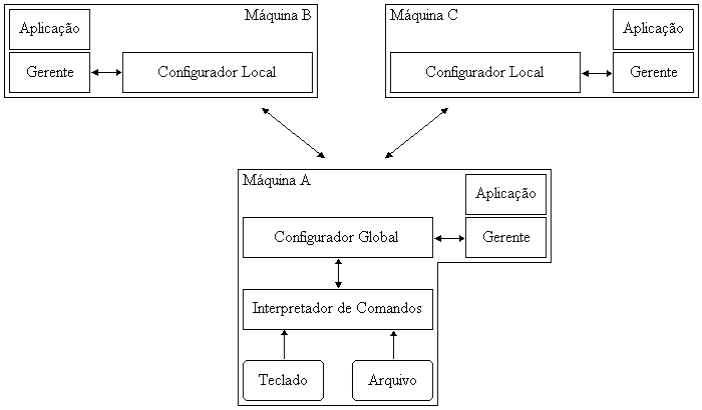

Figura 1- Arquitetura do Sistema

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.

2.2 - 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[3]: 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` original, 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 2-a), o configurador global instancia o módulo `C` na máquina onde se encontra `A`, mas troca seu nome de `C` para `B'` (Figura 2-b). Assim, quando `A` chama um método de `B` na realidade estará invocando `C` (Figura 2-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.

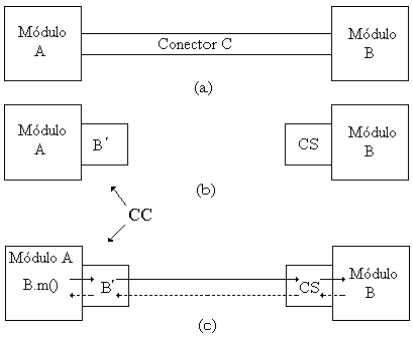

Figura 2 – Conexão entre módulos

A implementação da primitiva para ligação se faz em dois passos. No primeiro, o 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*[10].

No segundo passo, é instanciado o 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.

Esta abordagem é semelhante a utilizada em RPC e CORBA, porém ainda mais transparente ao programador, uma vez que este não precisa incluir no código da aplicação qualquer referência externa para realizar a comunicação. O suporte de configuração resolve este problema transparentemente, através da renomeação dos módulos.

Outras primitivas para iniciar, bloquear, continuar e terminar a execução dos módulos, que rodam como *threads*, também estão disponíveis.

Deve-se salientar que todas as primitivas podem ser também utilizadas como uma biblioteca tradicional, o que dispensaria o uso do configurador global, trazendo maior poder de expressão às aplicações. O programador pode, por exemplo, programar um módulo que execute automaticamente a configuração da aplicação, bem como eventuais reconfigurações necessárias durante sua operação.

2.3 - 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 (Código 1).

```
 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   }
13 }
```

Código 1 – 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 (Código 2) 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 }
```

Código 2 – Código do Servidor

Cabe ao conector cliente (Código 3) implementar uma interface semelhante ao do servidor:

```
01      package conector_socket;
02      public class conector_socketClnt {
03              public boolean consumer (int i){
04                      boolean result;
05                      dos.writeInt(i);              // escreve parâmetro de entrada
06                      result = is.readBoolean();           // lê parâmetro de saída
07                      // código do socket
08                      return result;
09              }
10      }
```

Código 3 – Código do conector (cliente)

O código do 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` (Código 4).

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

Código 4 – 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.
```

Código 5 – Comandos de Configuração

3 - Trabalhos Futuros

Um trabalho em andamento investiga a possibilidade de criação de conectores genéricos em Java. Como descrito em 2.2, o conector cliente deve possuir, no mínimo, um subconjunto de métodos idênticos aos

implementados pelo servidor. O único inconveniente desta abordagem é que existirá um conector especifico para ligar dois pares de módulo, em virtude das interfaces terem de ser compatíveis. Suponha, por exemplo, que desejamos ligar o produtor do exemplo da seção 2.3 com um *buffer* do tipo:

```
01      public class buffer implements Runnable {
02              public void run () {}
03              public int get () {
04                      // Retira elemento do buffer
05              }
06              public boolean put(int i) {
07                      // Coloca elemento no buffer
08              }
09      }
```

Código 6 – Comandos do *buffer*

Neste caso, um novo conector deverá ser implementado, uma vez que a interface da classe *buffer* é diferente da interface da classe consumidor. Mesmo que este conector implemente o mesmo estilo de comunicação, ele deverá sofrer pequenas modificações para que possa ligar outros pares de módulos (em virtude das mudanças de interface e, consequentemente, dos tipos dos parâmetros de entrada e de retorno):

```
01      public class conector_socketClnt {
02
03              public boolean put (int i) {
04                      boolean result;
05                      // código do socket
06                      dos.writeInt(1);                        // tipo de operação
07                      dos.writeInt(i);                // escreve parâmetro de entrada
08                      // código do socket
09                      result = is.readBoolean();              // lê parâmetro de saída
10                      // código do socket
11                      return result;
12              }
13
14              public int get () {
15                      int result;
16                      // código do socket
17                      dos.writeInt(2);        // tipo de operação
18                      // código do socket
19                      result = is.readInt();// lê parâmetro de saída
20                      // código do socket
21                      return result;
22              }
23      }
```

Código 7 – Código do conector (cliente) para a classe *buffer*

Graficamente, temos :

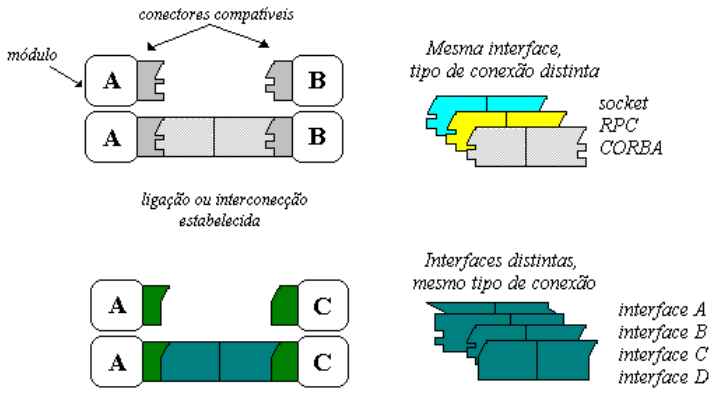

Figura 3 – Os conectores são dependentes das interfaces dos módulos

Nossos estudos buscam uma alternativa 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. Por exemplo, uma vez implementado um conector que faz comunicação utilizando CORBA, esse mesmo conector poderia ser utilizado para ligar quaisquer pares de módulos, independente de suas interfaces (poderia ligar o produtor ao consumidor ou o produtor ao *buffer*). Isso seria feito, por um lado, sem a necessidade do uso de bibliotecas por parte dos módulos envolvidos na

comunicação (o que poderia dificultar a separação de interesses), e de outro sem a necessidade da recompilação do conector em virtude da mudança de sua interface para a adequação à interface dos módulos a serem conectados.

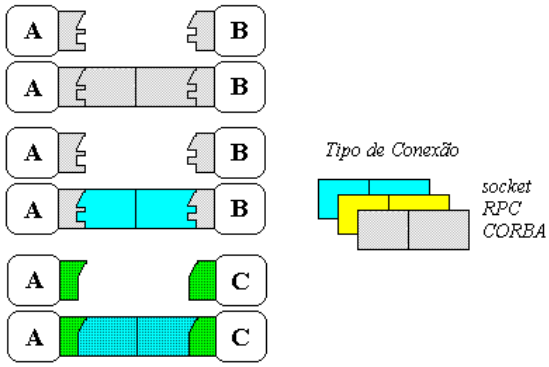

Figura 4 – O conector genérico permite a ligação de dois pares de módulos, independentemente das suas interfaces

Ainda estão sendo estudadas, 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) também estão sendo desenvolvidos. Desta forma, alcançaríamos o poder de expressão do modelo AOP [6], sem necessidade de usar linguagens especiais de programação.

4 – 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. GOLM, Michael. "*Design and Implementation of a Meta Architecture for Java.*" Diplomarbeit im Fach Informatik, Friedrich-Alexander-Universität Erlangern-Nürnberg, janeiro de 1997.
3. HARICH, Jack. "*The trick to using a basic Java 1.1 network and file class loader*". JavaWorld, outubro de 1997. Documento eletrônico: http://www.javaworld.com/javaworld/javatips/jw-javatip39.html.
4. GOSLING, James. "*The Feel of Java*". IEEE Computer. Junho 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. SZTAJNBERG, A., LOBOSCO, M., LOQUES, O., "*Uma Plataforma para o Suporte à Evolução de Sistemas Distribuídos*", 2º Workshop em Sistemas Distribuídos, Curitiba, junho de 1998.
9. WERNER, J. A. V. "*Ambiente Gráfico para a Construção, Configuração e Gerenciamento de Aplicações Distribuídas*". Dissertação de Mestrado, Departamento de Engenharia Elétrica, PUC-RJ, Rio de Janeiro, RJ, 1995.
10. YOURST, Matt T. "*Inside Java Class Files*". Dr. Dobb's Journal, janeiro de 1998.