Submetido ao SBRC 99 - aceito como poster

A abordagem R-RIO[1] para concepção de aplicações distribuídas

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

Resumo

Aplicações distribuídas possuem natureza dinâmica e necessitam de suporte para se adaptarem a novas demandas funcionais e não-funcionais. Novas demandas funcionais são criadas por usuários das aplicações, que precisam de novas funções ou necessitam alterar funções existentes. Além disso, durante o ciclo de vida das aplicações, podem surgir novos requisitos não-funcionais que, em geral, não estão associados diretamente com a aplicação. Modelos tradicionais para o desenvolvimento de *software* não oferecem as facilidades necessárias para a produção de aplicações que se adaptem dinamicamente a essas demandas. Portanto, uma abordagem que facilite a concepção dessas aplicações se faz necessária. Apresentamos neste artigo a abordagem R-RIO, que estende o conceito tradicional de configuração, permitindo que se descreva a arquitetura das aplicações a partir de componentes e suas interações e, adicionalmente, seus aspectos não-funcionais como a distribuição, coordenação e protocolos de comunicação. A arquitetura de uma aplicação distribuída é então descrita com o uso de uma linguagem de configuração. A partir da descrição da aplicação, um *middleware* chamado configurador possibilita a criação de suas instâncias. O configurador também oferece suporte para adaptação das aplicações que necessitam reagir à mudanças de requisitos ou nas condições de operação, inclusive durante a fase de operação.

Abstract

Distributed applications have dynamic nature and need support to be adapted to new functional and non-functional demands. New functional demands are created by applications' users that need new functions or need to change existing ones. New non-functional requirements that, in general, are not directly associated with the application functions can also appear during the applications' life-cycle. Traditional software development models do not provide the desired facilities to produce applications that can be dynamically adapted. Therefore, an approach to facilitate the development of these applications is necessary. In this paper we present the R-RIO approach, that extends the traditional configuration concept, allowing application architectures to be described by components and their interactions and, additionally, by their functional and non-functional aspects, such as distribution, coordination and communication protocols. As an example, the architecture of a distributed application is described using R-RIO's configuration language. It is also presented a configurator *middleware* that makes possible to create distributed programs, described through the configuration language, that can be adapted to new requirements even during their operation.

1 Introdução

Aplicações distribuídas possuem natureza dinâmica e necessitam de suporte para se adaptarem a novas demandas funcionais e não-funcionais [Lad 97]. Novas demandas funcionais são criadas por usuários das aplicações, que precisam de novas funções ou necessitam alterar funções existentes. Além disso, podem surgir novos requisitos não-funcionais que, de uma forma geral, não estão associados diretamente com a aplicação, como protocolos de comunicação, qualidade de serviço, disponibilidade ou tolerância a falhas, por exemplo. Modelos tradicionais para a concepção de *software* não oferecem as facilidades necessárias para a produção de aplicações que se adaptem dinamicamente a novos requisitos. Na maioria dos casos as aplicações são estáticas e fruto de um grande esforço de programação, resultando em um código complicado, com vários aspectos entrelaçados (*code tangling*

[1] R-RIO. Reflective, Reconfigurable Interconnectable Objects. Estes conceitos englobam as principais características do modelo proposto em R-RIO.

[Lop 97]). É comum, por exemplo, um mesmo bloco de código conter instruções para a comunicação via *sockets*, sincronização via semáforos, além das instruções que executam a computação intrínseca da aplicação, criando assim uma dificuldade para a reutilização de código. Para facilitar a concepção destas aplicações, uma nova abordagem se faz necessária. Esta abordagem deve considerar que todas aplicações precisam evoluir dinamicamente em seus aspectos funcionais e não-funcionais ou mesmo em sua estrutura.

Apresentamos neste artigo a abordagem R-RIO, que estende o conceito de configuração, permitindo que se descreva a estrutura de aplicações a partir de componentes e suas interligações e, adicionalmente, seus aspectos não-funcionais como a distribuição, coordenação ou protocolos de comunicação, citando apenas alguns. Em R-RIO a configuração de aspectos diferentes pode ser feita de forma separada, utilizando-se um elemento de conexão de módulos, com características reflexívas. A arquitetura de uma aplicação distribuída é descrita com o uso de uma linguagem de configuração. A partir desta descrição, um *middleware* chamado configurador possibilita a criação de instâncias desta aplicação. O configurador também oferece suporte para adaptação das aplicações que necessitam reagir à mudanças de requisitos ou nas condições de operação, inclusive durante sua execução.

A primeira parte deste trabalho apresenta os conceitos básicos de R-RIO e discute sua abordagem para a concepção de aplicações distribuídas. Na segunda parte, um protótipo de ambiente utilizando os conceitos de R-RIO é avaliado. A partir deste protótipo um exemplo é implementado para validar a proposta e as soluções adotadas. Nas ultimas seções, os conceitos são consolidados e algumas conclusões apresentadas.

2 A abordagem R-RIO

R-RIO procura atender um conjunto de requisitos que pretende oferecer uma base para a engenharia de sistemas distribuídos de uma forma geral:

Reusabilidade. Técnicas utilizadas para a concepção de aplicações devem permitir que partes de aplicações previamente concebidas possam ser reutilizadas de forma a minimizar esforços de desenvolvimento. Em um caso extremo, uma aplicação pode ser criada completamente a partir de partes já existentes [Rin 98]. No contexto deste trabalho, reusabilidade é considerada em relação a sistemas distribuídos, onde deseja-se construir aplicações utilizando-se componentes disponíveis e interconectando-os através de uma rede [Atk 91].

Dois conceitos estão associados a reusabilidade: modularidade e interoperabilidade. O projeto de componentes modulares está vinculado a dois aspectos: encapsulamento, em que o comportamento do módulo é auto-contido e padronização de interface, em que a as regras de acesso à funcionalidade do módulo são claramente definidas. Interoperabilidade é essencial para o suporte de aplicações distribuídas compostas por módulos, oferecidos por provedores independentes, que precisam se comunicar e cooperar [Weg 96]. Estes módulos podem residir em diferentes nós, usar linguagens, protocolos e formato de dados particulares, levando à necessidade de cuidados especiais para que a interoperabilidade seja obtida.

Separação de Interesses. A separação de interesses (*separation of concerns*) é um paradigma que vem sendo considerado para a concepção de grandes sistemas [Agh 93, Aks 96, Ber 96, Kic 97]. A separação de interesses divide o projeto de uma aplicação em especificações de requisitos funcionais e não-funcionais. Aspectos relacionados com as computações básicas de uma aplicação podem idealmente ser tratados independentemente de aspectos não-funcionais como a coordenação, comunicação e distribuição. Aspectos não funcionais como protocolos de comunicação, QoS e gerência de recursos do ambiente de execução, considerados operacionais, também podem ser tratados separadamente. A obtenção da separação de interesses exige, entretanto, uma disciplina do projetista. A arquitetura da aplicação deve ser planejada com esta finalidade, considerando-se a modularidade e interoperabilidade durante este processo. Assim sendo é possível concentrar esforços para a concepção dos módulos funcionais e para a adaptação da aplicação às especificações não-funcionais em momentos diferentes.

Evolução Dinâmica. No escopo deste trabalho, evolução dinâmica é um requisito que oferece o suporte para a modificação e inclusão de novos componentes em aplicações distribuídas que não podem ser completamente paradas para realizar estas operações. Por exemplo, a atualização de *software* (*up-grade*) de componentes, inclusão de tolerância a falhas, introdução de novos parâmetros de QoS, alteração de estilos de comunicação ou restrições temporais são atividades que podem implicar em mudanças em tempo de execução. Estes ajustes podem ser tratados, em muitos de seus aspectos, de forma transparente para a aplicação.

Abrangência. O requisito de abrangência de uma tecnologia apresenta várias facetas. Primeiramente deseja-se que com uma dada tecnologia um grande número de problemas possa ser solucionado e que para isso ela tenha expressividade adequada. Adicionalmente são observadas a disponibilidade, disseminação e aceitação das tecnologias. Ferramentas facilmente disponíveis e disseminadas facilitam a criação de massa crítica de *experts* e em conseqüência são mais aceitas e tendem à evoluir.

A proposta de R-RIO é oferecer uma nova alternativa para a construção de aplicações distribuídas baseada em configuração. Esta proposta difere de outras existentes por integrar explicitamente o paradigma de separação de interesses. Em R-RIO o conceito de configuração está baseado em:

- uma metodologia para a concepção de aplicações, por composição, a partir de módulos componentes, atendendo a aspectos funcionais básicos das mesmas;
- uma linguagem de configuração, que permite:

1. descrever a interconexão de módulos para formar a estrutura das aplicações ou módulos de maior complexidade.
2. descrever características ou aspectos não-funcionais das aplicações, como por exemplo, a sincronização, distribuição, tolerância a falhas ou parâmetros de qualidade de serviço.

- um sistema de suporte, chamado de configurador, que permite a criação, conexão e remoção de componentes das aplicações.

A metodologia de R-RIO incentiva a modularidade na concepção dos elementos básicos e a separação explicita entre a criação de componentes e as interações destes componentes. Isto permite, em princípio, retirar dos módulos a responsabilidade de implementar e gerenciar as interações, e concentrar esta responsabilidade em elementos de conexão, chamados em R-RIO de conectores. Esta separação de aspectos aumenta a possibilidade de reuso dos módulos funcionais e facilita a adequação dos conectores para diferentes estilos de interação.

2.1 Módulos, Portas e Conectores

O modelo de R-RIO está baseado em três elementos básicos:

1. **módulos**, componentes de uma aplicação que, basicamente, encapsulam sua funcionalidade;
2. **conectores**, usados, no nível de configuração, para interligar e selecionar a forma de interação de módulos. No nível de operação os conectores intermediam e executam a interação de módulos de acordo com o que foi configurado;
3. **portas**, que identificam os pontos de acesso pelos quais módulos e conectores oferecem ou solicitam serviços.

O primeiro passo para a construção de uma aplicação é a descrição da configuração dos módulos que irão compor a mesma e a seleção dos conectores que interligarão estes módulos. Em seguida, o relacionamento destes elementos é determinado através das ligações entre módulos e conectores. Adicionalmente, podem ser descritos requisitos não-funcionais da aplicação, permitindo que esta composição de módulos e conectores seja ajustada em diferentes aspectos. O exemplo de uma configuração produtor-consumidor com *buffer* limitado é apresentado na Figura 1. Este exemplo, também utilizado em [Lop 97 e Ber 96], servirá como base para ilustrar os conceitos de R-RIO.

Um módulo é um elemento que potencialmente encapsula parte da funcionalidade de uma aplicação. Elementos funcionais claramente distintos da aplicação dão origem a tipos de módulos distintos [Sha 95], como por exemplo clientes, servidores, geradores e tratadores de eventos. Na Figura 1 podemos identificar 3 instâncias de módulos, criadas a partir de uma descrição das classes destes módulos. Classes de módulos possuem um mapeamento para uma implementação. Por exemplo, os módulos da Figura 1 podem ser implementados como objetos em Java , C++ ou CORBA [OMG 96].

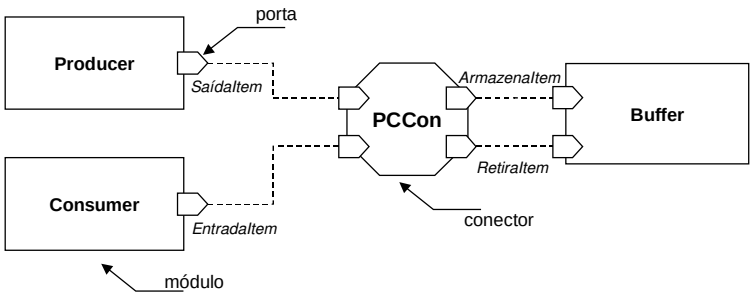

Figura 1 - Aplicação Produtor-Consumidor

Portas são elementos auxiliares do modelo de configuração que representam pontos de interação de um módulo. As portas podem ser de entrada, através das quais os módulos oferecem serviços, ou de saída, por onde os módulos requisitam serviços de outros módulos. Na linguagem de configuração a porta é utilizada como referência na ligação de dois componentes. Por exemplo, ao ligarmos uma porta de saída de um módulo cliente a porta de entrada de um módulo servidor, estamos acoplando estas referências, permitindo a interação entre estes componentes. As portas também são elos importantes entre o nível de configuração e a implementação. No nível da implementação uma porta pode ser identificada como a assinatura de um método ou procedimento, no caso de portas de entrada, ou uma invocação de método ou chamada de procedimento, no caso de portas de saída. Desta forma o configurador pode traduzir as referências dos pontos de interação entre os dois níveis. Na Figura 1 os módulos *Producer* e *Consumer* estão preparados para fazer requisições através das portas *SaídaItem* e *EntradaItem*, respectivamente. No nível da implementação as requisições podem ser mapeadas transparentemente como invocações de métodos. Da mesma forma, o módulo *Buffer* oferece serviços através de suas portas de entrada *ArmazenaItem* e *RetiraItem*, que podem ser mapeados como métodos oferecidos.

O conector é um elemento que oferece a abstração necessária para se determinar, no nível da configuração, a interação de módulos. Conectores são responsáveis, primariamente, por encaminhar as requisições que devem ir de uma porta de saída de um módulo para uma porta de entrada de outro módulo e o retorno dos resultados no sentido contrário, quando necessário. Este encaminhamento é executado segundo um padrão de comportamento para o qual o conector foi configurado.

Conectores possuem um conjunto de características que os tornam especialmente atraentes para tratar aspectos relacionados com a interação e a comunicação entre componentes [Bis 96]. Dentre estas características destacam-se:

- o conhecimento da interface e as referências dos módulos por eles interligados;
- a capacidade de examinar e manipular o conteúdo das requisições e respostas que passam por eles;
- a possibilidade de controlar a forma da interação dos módulos.

O conector pode operar, em algumas configurações, simplesmente repassando requisições e respostas entre os módulos. Em outras situações, além do encaminhamento de requisições, o conector pode atuar manipulando os valores dos argumentos das requisições que passam através dele, impor comportamentos diferentes à forma com a qual o encaminhamento das requisições é feito, como se fossem contratos [Hel 92] ou adicionar novas características à interação dos módulos. O fato de um conector poder controlar as interações entre módulos através da interceptação e manipulação de

requisições e respostas torna evidente a reflexividade deste elemento, embora o esquema reflexivo usado seja diferente ao que se tem feito tradicionalmente através de linguagens especializadas ou ambientes de suporte a reflexão dinâmica [Gol 97 e Oli 98]. Assim sendo, outros aspectos, além da comunicação, também podem ser configurados a partir dos conectores. Por exemplo, aspectos como a sincronização podem ser encapsulados nos conectores, tornando, de certa forma, a programação dos módulos independente destes aspectos não-funcionais. Na aplicação da Figura 1, o conector *PCCon* pode ser configurado para impor exclusão mútua e sincronização no acesso ao módulo *Buffer*, de forma ortogonal à funcionalidade dos módulos. O exemplo completo da configuração é apresentado na seção 2.3.

As interconexões de módulos são indicadas explicitamente durante a configuração através de comandos de *ligação*. Através destes comandos as referências a portas de entrada e saída passam a ser conhecidas e podem ser acopladas pelo conector. Em um comando de ligação são indicados os módulos a serem interconectados e o tipo de conector a ser utilizado. Na aplicação da Figura 1, por exemplo, após a criação das instâncias dos módulos o conector *PCCon* foi utilizado para interligar as portas de entrada e saída compatíveis (*SaídaItem* à *ArmazenaItem* e *EntradaItem* à *RetiraItem*).

O mapeamento de conectores para o nível da implementação não é rígido. Um conector pode assumir a forma de uma consulta a tabela para ligar referências entre a chamada e o corpo de um procedimento contidos em módulos diferentes. Um conector também pode ser implementado como um objeto que oferece e invoca métodos, que será interposto entre os módulos funcionais da aplicação. Adicionalmente, ao mapear um conector em uma implementação, o projetista ainda pode utilizar mecanismos auxiliares como primitivas de sincronização, protocolos de comunicação, entre outros, oferecidos pelo sistema de suporte nativo, para realizar o controle deste comportamento.

2.2 Separação de Interesses e Programação de Aspectos

No contexto de R-RIO, o requisito separação de interesses é atendido das seguintes formas:

- selecionando-se módulos para compor a estrutura funcional da aplicação e conectores para intermediar a interação destes módulos, em uma primeira fase;
- concebendo-se a arquitetura da aplicação pela interconexão dos módulos e conectores para atender os requisitos de interação demandados pela aplicação;
- configurando-se aspectos não-funcionais através de ajustes nos conectores ou por composição de módulos e conectores adaptados, a partir de módulos e conectores mais simples.

Com estas possibilidades, o projetista poderá usar uma disciplina na concepção das aplicações para que os módulos sejam responsáveis pelos aspectos funcionais e os conectores pelos aspectos não-funcionais. Esta disciplina, embora não obrigatória, potencializa a reusabilidade dos módulos e conectores.

A seguir apresentamos a abordagem para a programação de alguns aspectos não-funcionais, tais como, distribuição, comunicação e coordenação, no contexto de R-RIO. Observa-se que a programação destes aspectos é, em grande parte, ortogonal à concepção dos módulos, contemplando a separação de interesses e possibilitando encapsular os mesmos em conectores. Entretanto outros aspectos podem não levar a uma solução completamente separada. Assim sendo, o modelo não proíbe o programador de agregar aspectos funcionais e não-funcionais no mesmo componente, embora isso deva ser evitado, e nem de utilizar qualquer recurso disponível nas linguagens de programação.

2.2.1 Distribuição

A abordagem proposta permite que os módulos de uma aplicação possam ser instanciados em qualquer nó de um sistema distribuído. Na implementação dos módulos, não é necessária nenhuma referência quanto a sua localização ou qualquer outro suporte para a interação de módulos distribuídos. A localização de uma instância de um módulo é transparente em relação à sua funcionalidade.

Na linguagem de configuração, a instanciação de um módulo pode vir acompanhada de um parâmetro

indicando onde esta instância deve ser criada. Este é o único momento em que se associa um módulo à sua localização. No exemplo da Figura 1, cada módulo da aplicação poderá ser instanciado em um nó diferente (ver a seção 2.3).

2.2.2 Comunicação

Em uma aplicação distribuída a interação entre os módulos é feita através de um sistema de comunicação, provavelmente usando diferentes estilos de interação. Tanto na concepção quanto durante a operação de uma aplicação distribuída, ajustes na comunicação entre os módulos podem ser necessários, como por exemplo a seleção do mecanismo a ser utilizado, o protocolo de comunicação mais adequado ou a determinação de parâmetros de qualidade de serviço (QoS). Ajustes durante a operação podem ser feitos sob controle da própria aplicação pela reconfiguração de conectores. Uma opção é concentrar-se as tomadas de decisão sobre as reconfigurações em módulos especiais, utilizando-se inclusive linguagens especializadas, que interagem com o configurador para adaptar a aplicação [Gar 94 e Agh 98].

Como a comunicação entre os módulos de uma aplicação é intermediada por conectores, detalhes operacionais de comunicação podem ser ajustados através destes. A linguagem de configuração permite que anotações sobre a comunicação sejam declaradas nos conectores de maneira simples. O projetista pode configurar os aspectos de comunicação sem a preocupação imediata de como serão implementados. Se uma implementação de conector com as características desejadas estiver disponível, é suficiente indicá-la na configuração e esta será automaticamente adaptada para a aplicação (ver seção 2.4). Caso contrário, o novo conector requerido deverá ser implementado ou obtido por composição (ver seção 2.2.5). Em qualquer caso, durante a implementação de um conector, o programador pode fazer uso de facilidades disponíveis no sistema nativo. Por exemplo, o programador poderá se valer de mecanismos de comunicação como *sockets* ou RPC. Muitas das interfaces de programação destes mecanismos também permitem que se determinem parâmetros de QoS, complementando as possibilidades de ajuste nos aspectos de comunicação que se pode oferecer através de conectores.

2.2.3 Coordenação

Em nossa proposta os aspectos de coordenação de uma aplicação são especificados em alto nível de abstração, de forma separada dos módulos sobre os quais irão atuar. Instâncias diferentes de um mesmo tipo de módulo podem estar submetidas a esquemas de coordenação distintos. O modelo para o tratamento do aspecto de coordenação prevê três tipos básicos de situação:

- concorrência entre módulos de uma aplicação,
- concorrência e exclusão mútua entre métodos de um mesmo módulo, e
- a sincronização entre métodos de um mesmo módulo.

Em R-RIO todos os módulos de uma aplicação são executados potencialmente de forma concorrente. Cada módulo pode ser considerado uma unidade autônoma que pode concorrer pelo processamento com outros módulos ou pode executar paralelamente se estiver em um ambiente paralelo ou distribuído. Este modelo é adotado em propostas semelhantes, como [Agh93 e Lop 97].

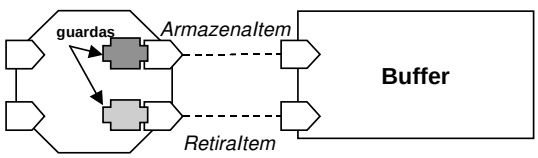

Figura 2 - Um módulo com guardas em seus métodos

Além do gerenciamento da concorrência e exclusão mútua entre métodos de um módulo, algumas situações requerem que a execução de determinados métodos seja condicionada ao estado atual do

módulo. Em nossa proposta, utilizamos o conceito de guardas lógicos. Um guarda pode controlar a admissão de requisições à porta de entrada de um método, retardando o tratamento destas requisições até que um conjunto de condições, associadas a este guarda, seja satisfeito.

Em R-RIO, guardas são conectores que estarão ligados diretamente às portas de entrada (Figura 2) dos módulos. O código de conectores de coordenação não precisa ser entrelaçado com os módulos, como se dá em [AspJ]. Isso é factível, por exemplo, com a introdução de métodos para a inspeção do estado no módulo sob coordenação. Mantendo-se esta separação de interesses, é possível que conectores de coordenação sejam reconfigurados e que a implementação de conectores com guardas seja automatizada.

2.2.4 Outros Aspectos

Além dos aspectos apresentados, outros podem ser necessários na concepção de aplicações distribuídas e são, em princípio, realizáveis nos conectores. Dentre eles destacam-se:

- **Replicação e tolerância a falhas** - Permite a configuração de módulos replicados e a seleção de esquemas de tolerância a falhas. Experiências com configuração de replicação e tolerância a falhas podem ser vistas em [Loq 97];
- **Persistência -** Permite configurar aspectos de persistência para os módulos. Este aspecto pode ser utilizado em conjunto com o aspecto de tolerância a falhas para facilitar a implementação de esquemas de recuperação de módulos que falharam e em sistemas de suporte à transações [Iss 96];
- **Tempo Real** - Características relacionadas com restrições temporais ou o escalonamento de módulos [Fra 96] podem ser especificados através do uso de conectores.

2.2.5 Composição

Ao descrever-se a arquitetura de uma aplicação, essencialmente descreve-se uma composição de módulos e conectores. A seleção e a descrição dos módulos e conectores a serem utilizados é o primeiro passo na concepção de uma aplicação. Isto feito, a aplicação é composta descrevendo-se a interligação dos módulos através de conectores. Sob esta perspectiva toda aplicação pode ser considerada uma composição de módulos.

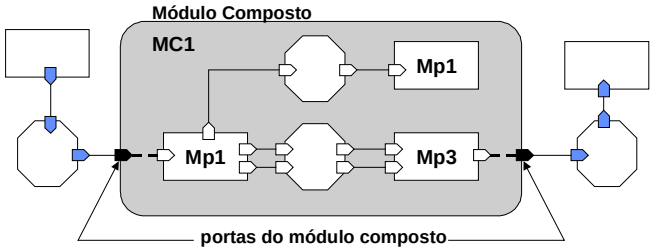

Figura 3 - Uma aplicação com módulo composto

Um módulo composto é formado por módulos primitivos ou outros módulos compostos, interligados por conectores. Os outros módulos da aplicação têm acesso ao módulo composto apenas através das portas com visibilidade externa. Para o resto da aplicação, um módulo composto tem a mesma apresentação de um módulo primitivo. O módulo composto da Figura 3 (*MC1*) é formado por três módulos primitivos (*Mp1*, *Mp2* e *Mp3*).

A composição pode ser usada para a construção de módulos com função agregada ou mais especializados. Além disso, módulos compostos podem ser usados na estruturação de outras aplicações, ampliando as possibilidades de reutilização.

Conectores podem ser considerados como módulos especializados em promover a interação entre outros módulos. Entretanto, a separação dos conceitos de módulos e conectores se justifica com a

possibilidade de se associar formalismos para a verificação automática de propriedades das aplicações sendo configuradas [Agh 98]. Assim sendo, a concepção de um conector composto através da composição de conectores mais simples também é possível.

Suponha que a aplicação produtor-consumidor exigisse que os dados tivessem que ser criptografados antes de serem armazenados no *buffer* e decriptografados antes de serem consumidos pelos consumidores. Um conector composto (Figura 4) pode substituir o conector original contemplando estas novas características. Outros requisitos, como a utilização de protocolos diferentes para as interações entre produtor/*buffer* e consumidor/*buffer*, podem ser configurados da mesma forma.

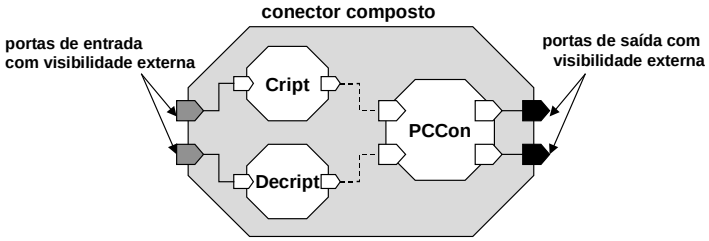

Figura 4 - Um conector composto

Embora a idéia da composição de conectores seja simples, existem algumas implicações que devem ser consideradas, tais como:

- verificar a ortogonalidade, ausência de conflitos ou interações indesejadas dos aspectos sendo compostos para garantir a coerência do resultado [Joh 97, Mon 97, Jac 98];
- dispor de ferramentas para facilitar a composição automática de conectores, a partir de características descritas na configuração;

Além destes pontos existem considerações sobre a eficiência. Conceitualmente, o comportamento de um conector composto é idêntico ao de um único conector que contenha as mesmas características. Entretanto, dependendo do mapeamento de um conector em uma implementação, no caso do conector composto, podem ser identificados algumas fontes de ineficiência, tais como transferências de dados entre espaços de endereçamento diferentes e interferências devido ao escalonamento. Por outro lado, pode-se prototipar uma aplicação com conectores compostos para em seguida desenvolver um único conector com todas as caraterísticas desejadas. Isto pode ser automatizado por um compilador especializado, que faria otimizações a partir da aplicação sendo descrita.

A solução para estes problemas não parece ser trivial, mas verifica-se que algumas composições podem ser feitas sem problemas aparentes. Por exemplo, a composição de um conector onde aspectos de coordenação tenham sido definidos e um conector onde aspectos de comunicação tenham sido declarados não oferece dificuldades.

2.2.6 Conectores Genéricos

Tradicionalmente, um conector é concebido para permitir a interação entre módulos, que possuem um conjunto de portas de entrada e saída, cada qual com sua assinatura. Assim, existe uma dependência de interfaces, e o conector deve conhecê-las para poder gerenciar a interações. Um conector RMI, por exemplo, só servirá para o conjunto de portas para o qual ele foi preparado. Para aumentar a possibilidade de reutilização, R-RIO utiliza o conceito de conector genérico Esta alternativa permite, para certos tipos de conectores, maior flexibilidade no desenvolvimento de aplicações, uma vez que um conector é concebido independentemente do conjunto de portas dos módulos que ele irá conectar. Por exemplo, uma vez implementado um conector que faz comunicação utilizando RMI, esse mesmo conector pode ser utilizado para ligar quaisquer pares de módulos, de um lado sem a necessidade do uso de bibliotecas por parte dos módulos envolvidos na comunicação, e do outro sem a necessidade de declaração de interface e geração de *proxys* e *stubs*.

2.3 A linguagem de Configuração

A linguagem de configuração que estamos propondo permite, como outras linguagens de configuração, a descrição, o armazenamento e a recuperação da descrição de módulos, conectores e aplicações. Em adição aos recursos tradicionais das linguagens de configuração estamos oferecendo recursos para a configuração de aspectos não-funcionais, como os descritos na seção 2.2. A linguagem proposta é batizada de Babel (*Building Applications by Evolution*). Os fundamentos de Babel são discutidos em [Mal 96].

A linguagem de configuração foi concebida com dois grupos de declarações. O primeiro grupo é responsável pela declaração dos tipos e instâncias dos elementos a serem usados na aplicação. Neste grupo estão as declarações *module*, *port* e *connector*. O segundo grupo permite que se declare comandos para a criação e interligação de instâncias de elementos. *instantiate*, *link* e *start* estão neste grupo. O suporte à configuração permite que estes comandos sejam utilizados tanto para que a aplicação seja instanciada para execução, como para reconfigurar-se esta mesma aplicação durante sua operação. Durante a instanciação da aplicação, o configurador pode fazer adaptações da configuração para que ela opere em um ambiente específico, como por exemplo selecionar versões compatíveis de módulos para determinado sistema operacional ou selecionar um conector de comunicação adequado para as condições de tráfego. O exemplo do produtor-consumidor com *buffer* limitado da Figura 1 é descrito no Código 1 (próxima página).

Após a descrição da aplicação, a criação da instância de aplicação é solicitada. Como a aplicação é tratada de forma similar a outro módulo qualquer, cria-se também uma instância de *BufferAplication*. Neste momento são criadas todas as instâncias de módulos e realizadas as conexões especificadas. A partir daí a instância *example* da aplicação inteira foi criada e com o comando *start* sua execução iniciada. Vale a observação que a arquitetura da aplicação admite um número arbitrário de instâncias de módulos produtores e consumidores. A criação de múltiplas instâncias pode ser feita atribuindo-se um nome diferente para cada uma ou através de construções com índices.

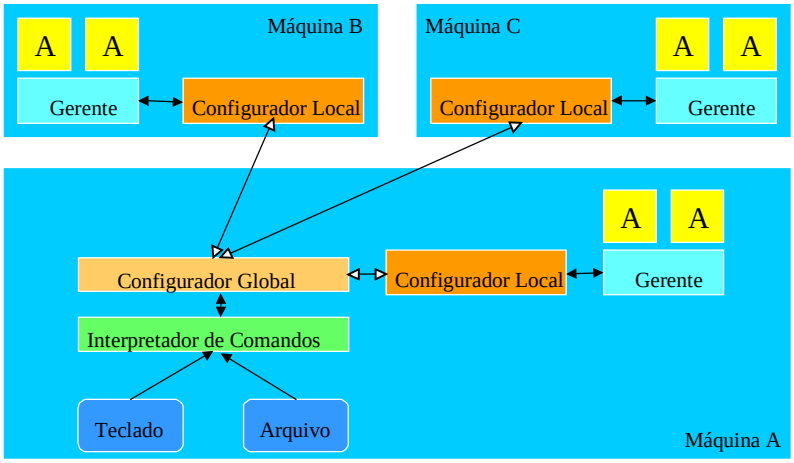

Figura 5 - Arquitetura do Sistema

3 Implementação

Para a implementação do protótipo de um ambiente, baseado na proposta apresentada na seção anterior, foi escolhida a linguagem Java. Sua escolha deveu-se a alguns aspectos importantes da linguagem, como a portabilidade do código das aplicações para diversas arquiteturas de *hardware* e *software*, suporte nativo para concorrência e sincronização e possibilidade de intervir no carregamento das classes em tempo de execução. No entanto qualquer outra combinação de linguagem de programação e suporte, como CORBA [Loq96], poderia ter sido escolhida para implementar o protótipo.

module BufferApplication {
  port PutT {
  parms (int Item);
  return (int Ack); }
  port GetT {
  parms (void);
  return (int Item); }
	Para criar a aplicação, define-se inicialmente os tipos de conectores e as classes de módulos que serão utilizados.
	A assinatura das portas é declarada aqui para ser reutilizada por vários módulos.

module BufferT {
  inport PutT ArmazenaItem;
  inport GetT RetiraItem;
  map CLASS Java "buffer"
} Buffer;
module ProducerT {
  outport PutT SaidaItem;
  map CLASS Java "producer";
} Producer;
module ConsumerT {
  outport GetT EntradaItem;
  map CLASS Java "consumer";
} Consumer;
	Os módulos são declarados indicando-se instâncias das portas deste módulo e o mapeamento para a implementação (neste caso o módulo *Buffer* é implementado pela classe buffer.class, programada em Java). De forma semelhante os outros módulos são configurados.
	Em seguida os conectores que interligarão os módulos são definido.

connector Cguard{ **condition** vazio = true, cheio = false; **staterequired** int n_itens; ***int*** MAX_ITENS = 10; **inport** GetT; **inport** PutT;	Declara-se a lista de variáveis de condição a serem usadas nos guardas e uma lista de variáveis de estado a serem consultadas no módulo *Buffer*. Em seguida, é definida uma variável para limitar o tamanho do *buffer*.

exclusive {
outport PutT {
inguard (vazio) {
after {
if (n_itens == MAX_ITENS) {
vazio = false; cheio = true;
}
}
}
}
 	
outport GetT {
inguard (cheio) {
after {
	As portas de saída do conector são declaradas como mutuamente exclusivas. Observe que *inport* e *outport* são utilizadas como qualificadores do sentido das portas do tipo PutT e GetT.
	A requisição não chegará ao método *ArmazenaItem* do módulo *Buffer* se o guarda **vazio** for falso.
	Ao completar o método e antes de enviar a confirmação, o guarda **vazio** recebe o valor false se n_itens for igual a MAX_ITENS.

If (n_itens == 0) {
Cheio = false; vazio = true;
}
}
}
}
}
map CLASS Java "GuardGCon";
} PCCon;
	De forma análoga, o método *RetiraItem* do *Buffer* não será executado enquanto o guarda **cheio** não for verdadeiro.
	Ao completar o método e antes de retornar o item retirado do *buffer*, o guarda **cheio** recebe falso se o espaço do *buffer* esta completamente vazio.

connector Comm { **map** CLASS Java "RmiGCon"; } PCRmi;	Define-se o conector para o aspecto de comunicação. Utiliza-se um conector RMI genérico.
instantiate Buffer **at** estação1 **instantiate** Producer **at** estação2; **instantiate** Consumer **at** estação3; **instantiate** PCRmi, PCCon;	Neste ponto os módulos e conectores são instanciados. O aspecto de distribuição é programado indicando-se em que estações os módulos serão instanciados.
link Producer, Consumer **to** Buffer **by** PCRmi,PCCon; }	A configuração é concluída interligando-se os módulos através dos conectores. A notação PCRmi,PCCon indica que os módulos serão ligados por uma composição do conector RMI e o conector PCCon declarados nesta arquitetura.
module BufferAplication example; **instantiate** example; **start** BA;	Em seguida cria-se uma instância da aplicação e inicia-se sua execução.

Código 1 - Configuração da aplicação produtor-consumidor

3.1 Arquitetura

A arquitetura da plataforma (Figura 5) assume um conjunto de nós interligados por rede. O usuário utiliza comandos de configuração, sintetizados a partir da descrição feita com Babel, para instanciar sua aplicação. Neste ponto assume-se que a consistência da arquitetura já foi verificada. Os comandos de configuração podem ser lidos de um arquivo ou digitados em um console, sendo em seguida repassados ao configurador global. Com a ajuda deste, módulos e conectores podem ser instanciados em qualquer estação e suas interligações estabelecidas, mesmo que ultrapassando os limites das respectivas estações locais. Para tal, cada estação implementa um suporte local para configuração, denominado configurador local, que funciona como uma espécie de *daemon*, aguardando comandos de configuração do configurador global. O gerente é o responsável por executar os comandos de configuração recebidos pelo configurador local.

3.2 Primitivas

Apresentamos a seguir as principais primitivas implementadas pelo módulo gerente. O conjunto completo de primitivas é apresentado e discutido em [Lob 99].

3.2.1 Instantiate

```
public synchronized boolean instantiate (String className, String moduleName,
                                         Object[] arg)
```

Método que tem como objetivo carregar localmente a classe className. Para tal, utilizou-se a facilidade de migração de código oferecida pela linguagem Java [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 seus 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, utilizando-se para isso o protocolo *http*.

Após a classe ser renomeada, ela é registrada junto a Máquina Virtual Java. Uma vez que o modelo prevê execução concorrente, conforme visto na seção 2.2.3, cria-se uma nova *thread* para a execução da instância do módulo, cujo nome é dado pelo argumento moduleName. Se o usuário passa argumentos para a inicialização do módulo, então o construtor apropriado do objeto é invocado, passando-se os argumentos em arg.

3.2.2 Link

```
public synchronized boolean link (String className, String moduleName1,
          String conector, String moduleName2, String host,
          String moduleInterface[], Vector portas, Vector moduleInterfaceII)
```

Método que tem como objetivo criar uma ligação entre portas de dois módulos, moduleName2, que solicita serviços, e moduleName1, que oferece serviços.

Esta ligação é feita em dois passos. No primeiro, o configurador global solicita ao configurador local da máquina onde se encontra moduleName1 a interface deste módulo (Figura 6.a). Esta consulta é feita pelo configurador local de forma simples, utilizando a reflexão estrutural (introspeção) oferecida pela própria linguagem Java [Java]. Com base nesta interface, representada na lista de argumentos por moduleInterface, o configurador global solicita ao configurador local onde se encontra o módulo moduleName2 que gere uma classe com o mesmo nome e interfaces da classe que oferece serviços, através da invocação à primitiva link (Figura 6.c). Esta classe recém gerada, chamada de *proxy*, transparentemente interceptará as chamadas originais do módulo que solicita serviços, podendo repassá-las, desta forma, para o conector conector (Figura 6.d). A porta de saída do módulo moduleName2, que foi declarada como variável pública, é então inicializada com o valor da instância recém-criada do *proxy*.

O segundo passo da ligação representa a atualização das tabelas de ligação. A cada invocação que chega ao *proxy*, este consulta o gerente para saber que conector deve ser invocado, passando para isto,

como argumentos, o nome da instância do módulo, o nome do tipo do módulo, o nome do método e a sua assinatura. O gerente então consulta as tabelas de ligação, retornando o nome do conector a ser utilizado, o nome da instância a ser invocada pelo conector e em que máquina esta instância está localizada. Este último argumento só será utilizado por conectores que implementem aspectos de comunicação, podendo ser ignorado pelas demais classes de conectores.

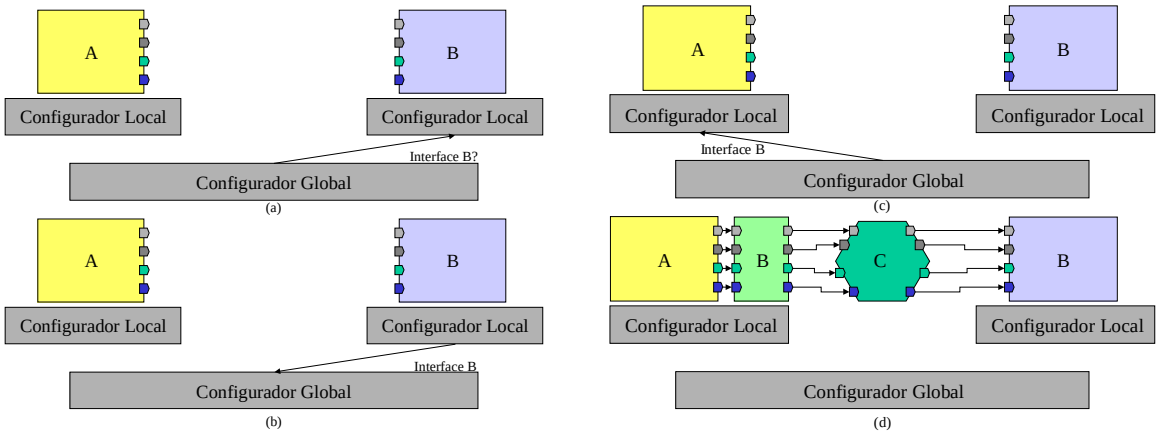

Figura 6- Representação esquemática da primitiva *link*. Em (a) o configurador global solicita a interface do módulo B. Em (b) o configurador local retorna a interface de B. Em (c) o configurador global repassa para o configurador local a interface do módulo B, solicitando que se gere um *proxy* com o mesmo nome e interface do módulo B original. Em (d) as chamadas de A são transparentemente repassadas para o *proxy*, que então pode repassá-las para o conector, que implementa o requisito operacional.

Quando ocorre uma reconfiguração da aplicação, somente o segundo passo é refeito, ou seja, somente são atualizadas as tabelas de ligações entre módulos. A geração de outro *proxy* é desnecessária porque o tipo da porta de saída não muda, já que é determinada pelo desenvolvedor durante a implementação. O que pode mudar é o tipo de conector utilizado para ligar as portas, a localização do módulo que oferece os serviços ou mesmo a própria implementação do módulo que oferece serviços.

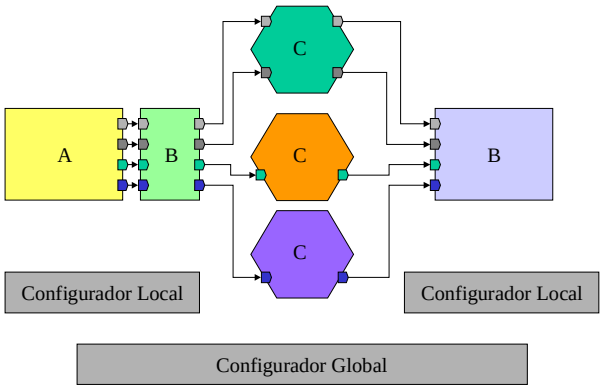

Figura 7 - Cada porta do módulo A pode estar ligada a uma porta do módulo B usando diferentes conectores

O primeiro passo, adicionada a renomeação efetuada com a primitiva instantiate, é similar à proposta adotada por Dalang [Wel 98]. No entanto nossa solução é mais transparente para o programador, uma vez que Dalang obriga-o a utilizar bibliotecas deste sistema e declarar um ClassLoader reflexivo para poder adicionar meta níveis a uma classe base. Nosso sistema gerencia tais tarefas, deixando o programador livre para implementar os requisitos de sua aplicação sem se preocupar com o uso de bibliotecas.

O uso do *proxy* permite a ligação de cada porta a um conector diferente (Figura 7). É preciso indicar na linguagem de configuração, que portas devem ser ligadas com que conector, conforme o Código 2:

```
link a:port1(int);port2(String) at melancia b at murici with sistema.conectores.rmi
link a:port1(int,long) at melancia b at murici with sistema.conectores.socketTCP
link a:port3(double) at melancia b at murici with sistema.conectores.socketUDP
```

Código 2 - Exemplo de ligação entre portas

3.3 Distribuição e Comunicação

A implementação do aspecto de distribuição é feita de maneira relativamente simples. Ao receber um comando de instanciação de um módulo para determinada máquina, por exemplo, `instantiate A as a at melancia`, o configurador global entra em contato com o configurador local da máquina `melancia`, solicitando que este carregue a classe `A` localmente e crie uma nova instância desta classe, de nome `a`. O configurador local, então, verifica se esta classe já foi carregada anteriormente, e em caso negativo, carrega a classe `A` a partir do local previamente configurado como local de origem das classes, que pode, como já descrito, ser um diretório ou uma URL.

Em relação ao aspecto de comunicação, são atualmente disponíveis no sistema os estilos de comunicação *socket* TCP, *socket* UDP e RMI, todos genéricos, ou seja, podem ser usados para estabelecer comunicação entre quaisquer portas (ou interfaces) de módulos, independente de suas assinaturas. Nada impede, ainda, que o programador desenvolva seu próprio conector, implementando um novo estilo de comunicação que inclusive pode não ser nativo da linguagem de programação adotada, bastando para isso seguir uma receita básica para o desenvolvimento de conectores.

3.4 Coordenação

Um mapeamento entre as sintaxes propostas para o tratamento do aspecto de coordenação pela linguagem de configuração e Java foi estabelecido, possibilitando, com a ajuda de um pré-compilador, que todo o código necessário para implementar a coordenação seja gerado automaticamente.

3.5 Outros Aspectos

Outros aspectos podem ser programados seguindo-se a mesma receita para desenvolvimento de conectores citada na seção 3.3. Por exemplo, se desejamos fazer um *trace* das invocações, e no entanto este aspecto não faz parte da linguagem de configuração, desenvolvemos nossa aplicação de forma a atender o seu requisito funcional, não nos preocupando neste momento em atender o aspecto de *trace*. Após esta fase, programamos o conector de *trace*, seguindo o modelo básico para o desenvolvimento de conectores, e ligamos pares de módulos com este conector recém desenvolvido, usando para isso os comandos disponíveis na linguagem de configuração.

3.6 Mudanças no Estilo de Programação

No que diz respeito a mudanças no estilo de programação necessárias para a adaptação à metodologia de R-RIO, o programador deve herdar a classe `Thread` da linguagem Java, de modo que o módulo possa executar e ter as funcionalidades inerentes desta classe. O método `main` deve então ser substituído pelo método `run`. Outra mudança de estilo necessária é a declaração da porta de saída (objeto servidor) como variável global pública, uma vez que pelo modelo as portas de entrada e saída devem ser conhecidas. Resumidamente, teríamos o seguinte formato:

```
public class nomeDoTipoDoMódulo extends Thread{
  public tipoDaPorta1 porta1;  //declaração da porta de saída 1
  public tipoDaPorta2 porta2;  //declaração da porta de saída 2
            . . .
  public void run() {
    // código equivalente ao do método main ou simplesmente vazio
  }
}
```

Código 3 - Estilo a ser seguido para a programação do módulo

3.7 Exemplo

Como exemplo apresentamos a implementação dos módulos da aplicação produtor-consumidor. O produtor (Código 4) produz números inteiros, que são consumidos pelo consumidor.

```
01 package producer_consumer;
02 public class producer extends Thread {
03   public buffer b; // declaração da porta de saída
04   public void run () {
05     int i;
06     for (i=1;;i++) {
07       if (b.ArmazenaItem(i) != 0)
08         System.out.println ("Produtor produzindo item " + i);
09       else
10         System.out.println ("Erro!");
11     }
12   }
13 }
```

Código 4 - Código do produtor

A única exigência para o código do cliente é que ele estenda a classe `Thread` da própria linguagem Java, e declare a porta de saída como pública. As cláusulas em negrito no código explicitam estas operações. O código do consumidor (Código 5) e do *buffer* (Código 6) possuem exigências semelhantes:

```
01 package producer_consumer;
02 public class consumumer extends Thread {
03   public buffer b; // declaração da porta de saída
04   public void run () {
05     int item;
06     for (;;) {
07      item = b.RetiraItem();
08      if (item != -1)
09         System.out.println ("Retirando do buffer item " + item);
10      else
11         System.out.println ("Erro!");
12     }
13   }
14 }
```

Código 5 - Código do consumidor

```
01 package producer_consumer;
02 public class buffer extends Thread {
03   int total = 0;                          // total de elementos no buffer
04   public int estado () { return total; } // retorna total de elementos no vetor
05   public void run () {}
06   public int RetiraItem () { ... }            // retira elementos de um vetor
07   public int ArmazenaItem(int i) { ... }      // põe elementos em um vetor
08 }
```

Código 6 - Código do buffer

É importante notar que tanto o código do produtor, quanto o do consumidor e do *buffer* não fazem qualquer menção aos requisitos de distribuição, comunicação, concorrência ou sincronização. Esta característica permite, em tempo de execução, a escolha da localização física dos módulos e do tipo de conector utilizado para implementar a interação entre estes módulos. Ainda é possível substituir o conector previamente escolhido por outro, de forma a adequar a aplicação a novas situações de execução. Torna-se clara a separação de interesses, um dos objetivos deste trabalho. O código do conector *PCCon*, que implementa os requisitos de comunicação e sincronização foi omitido neste trabalho por questões de espaço, mas pode ser visto em [Lob 99].

Para instanciar e executar a aplicação segundo a configuração apresentada na Figura 1, usamos os comandos conforme o Código 7.

```
01 Protótipo do Sistema R-RIO  -  Versao 1.0
02 Entre com o comando no prompt
03 > instantiate producer_consumer.producer as prod at camboim
04 Ok.
05 > instantiate producer_consumer.consumer as cons at melancia
06 Ok.
07 > instantiate producer_consumer.buffer as buf at murici
08 Ok.
09 > link prod at camboim buf at murici with sistema.conectores.PCCon
10 Ok.
11 > link cons at melancia buf at murici with sistema.conectores.PCCon
12 Ok.
13 > start buf at murici
14 Ok.
15 > start cons at melancia
16 Ok.
17 > start prod at camboim
18 Ok.
```

Código 7 - Comandos de configuração

4 Conclusão

Propostas correlatas como AOP - Aspect Oriented Programming [Kic 96] e linguagens que oferecem reflexão computacional [Gol 97T], além de outros ambientes de configuração [All 94, Mag 92 e Sha 95], foram analisadas em outro trabalho [Szt 99]. Essas propostas utilizam alguma forma de interceptação do fluxo de execução e interposição de código [Joh 93] para obter a separação de interesses. AOP, por exemplo, possui uma ferramenta chamada *weaver*, que entrelaça o código de aspectos programados separadamente em um único bloco [AspJ]. Linguagens de programação com características reflexívas normalmente utilizam uma combinação de identificação e interceptação do fluxo de execução do nível base (*reificação*), para disparar a execução de um código interposto no meta-nível (*reflexão*). Nos dois casos, existe uma ferramenta que administra a interceptação e interposição de código. Identificamos também que a tecnologia de configuração possui características semelhantes a cada uma das tecnologias estudadas, mas oferece explicitamente um esquema para descrever a arquitetura das aplicações através de composição e a possibilidade de verificações de propriedades dessas arquiteturas.

Com isto em mente, uma nova abordagem para a construção de aplicações distribuídas foi proposta. Nesta abordagem, baseada em configuração, utiliza-se conectores para intermediar as interações entre módulos e também encapsular aspectos não-funcionais das aplicações. A arquitetura de uma aplicação é descrita através de uma linguagem de configuração e os vários aspectos envolvidos podem ser tratados separadamente. As características que destacam R-RIO estão resumidas na Tabela 1.

Ficamos convencidos ao longo da elaboração desta proposta que a abordagem é genérica o suficiente para ser utilizada em sistemas que não sejam necessariamente distribuídos. Aspectos que não sejam ligados à distribuição e comunicação também podem ser configurados nas aplicações, desde que, de alguma forma, digam à respeito a interação entre módulos. Desta forma, um conector pode conter a expressividade, por exemplo de um *design pattern* do tipo ***Observer*** [Gam 95], que trata da geração de eventos e da reação esperada de *observadores*. Efeito semelhante foi proposto em [AspJ] concentrando a implementação do protocolo de geração e tratamento de eventos na descrição de um aspecto. Outros aspectos, além deste, como a validação do conteúdo dos parâmetros de requisições, controle de versões ou a persistência dos módulos, que não são específicos de sistemas distribuídos, também podem ser providas por configuração.

A implementação destes conceitos não é trivial se o objetivo é oferecer a separação de aspectos durante todo o processo de gestão de uma aplicação, inclusive durante a execução, permitindo a reconfiguração. Assim sendo, também foi proposto um mapeamento do modelo conceitual para o nível de implementação. Este mapeamento poderá ser feito através de ferramentas automáticas e pode ainda contar, eventualmente, com ambientes especializados.

Reusabilidade
Aplicações podem ser rapidamente prototipadas ou construídas a partir de módulos e conectores. Conectores podem encapsular padrões de interação entre módulos no mesmo sentido que é feito em *design patterns*. Módulos e conectores podem ser reutilizados para a composição de módulos mais complexos. A composição de conectores permite a obtenção de conectores com novas características. Módulos compostos podem encapsular soluções de *software* para determinada área de aplicação, como fazem os *frameworks* no modelo de objetos.
Separação de Interesses
Separação de interesses é obtida com o projeto separado de aspectos diferentes da aplicação, integrados através da linguagem de configuração. Módulos representam, potencialmente, os componentes funcionais da aplicação. Conectores são utilizados para a interligação de módulos e a configuração de requisitos não-funcionais da aplicação. Em princípio, aspectos não-funcionais não interferem na concepção dos módulos. Estilos diferentes de interação dos componentes são planejados e implantados separadamente nos conectores. Existe a possibilidade do uso da reflexividade intrínseca aos conectores em ajustes finos das conexões dos módulos das aplicações. A composição de conectores para combinar diferentes aspectos também é possível.
Evolução Dinâmica
Obtida pela capacidade de reconfiguração da aplicação durante sua operação. O esquema de mapeamento entre os elementos do modelo e a implementação permite que o encapsulamento de cada elemento seja preservado durante a execução. O suporte do configurador permite a realização de manobras na aplicação, como a substituição de componentes ou alterações nas ligações entre os módulos. Aliado a isso, a possibilidade de verificações de propriedades da arquitetura sendo manipulada permite que estas manobras sejam feitas com segurança.
Abrangência
A abordagem de R-RIO oferece flexibilidade para o desenvolvimento de aplicações distribuídas. A facilidade de composição de elementos permite a combinação de funcionalidades para atender aos diversos requisitos das aplicações. Em princípio, o desenvolvimento de módulos e conectores pode ser feito em qualquer linguagem de programação e não existem restrições quanto aos recursos utilizados do sistema operacional hospedeiro.

Tabela 1 - Características de R-RIO

Pela comparação do mesmo conjunto de exemplos desenvolvidos a partir da nossa abordagem e com protótipos de outras propostas [Szt 97, Szt 99], verificou-se que R-RIO possui expressividade equivalente, e pode ser mais flexível do que outras abordagens. Adicionalmente, a abordagem utilizada em nosso modelo permite um tratamento integrado do gerenciamento das aplicações desenvolvidas, desde a concepção até a execução e manutenção, diferentemente das outras propostas avaliadas, onde isso é feito de maneira empírica. Nossa proposta de configuração, por exemplo, permite que o mesmo modelo utilizado para conceber a arquitetura de uma aplicação seja utilizado na sua reestruturação, em resposta a novas demandas, mesmo depois desta ser implantada e estar em produção. Esta característica torna R-RIO uma base para o desenvolvimento de aplicações que possam evoluir dinamicamente.

O modelo de R-RIO ainda está sendo aperfeiçoado em vários pontos. A possibilidade da inclusão de novos aspectos será experimentada. A capacidade de composição e reflexão do modelo serão exploradas visando ampliar sua utilidade e abrangência em domínios diferentes de aplicações distribuídas e não distribuídas. O protótipo descrito neste artigo está disponível para uso em atividades de pesquisa.

5 Referências Bibliográficas

[Agh 93] Agha, G., Frolund, S. et *alli*, "Abstraction and Modularity Mechanisms for Concurrent Computing", IEEE Parallel and Distributed Technology, Systems and Applications, IEEE Computer Society, Maio, 1993.

[Agh 98] Astley, M. e Agha, G., "Customizations and Composition of Distributed Objects: Middleware Abstractions for Policy Management", ACM SIGSOFT'98 - 6th. International Syposioum on Foundations of Software Engineering, Lake BuenaVista, Florida, pp. 1-9, Novembro, 1998.

[Aks 96] Aksit, M. "Composition and Separation of Concerns in the Object-Oriented Model", ACM Computing Surveys, 28^{A} (4), Dezembro, 1996.

[All 94] Allen, R., Garlan, D., "Beyond Definition/Use: Architectural Interconnection", ACM SIGPLAN Notices, Volume 29, No 8, IDL Workshop, pp. 35-44, Agosto, 1994.

[AspJ] ____, "AspectJ™ Specification", Xerox, Paulo Alto Center, Março, 1998.

[Atk 91] Atkinson, C., "Object-Oriented Reuse, Concurrency and Distribution: an Ada-based approach", Addison-Wesley, 1991.

[Ber 96] Bergmans, Lodewijk M.J.; "Composing Concurrent Objects - Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programmming", PhD Thesis, Universiteit Twente, Finlandia, , ISBN 90 9007359-0, 1996.

[Bis 96] Bishop, J. and Faria, R, "Connectors in Configuration Programming Languages: are They Necessary?", IEEE Third International Conference on Configurable Distributed Systems, Annapolis, Maio, 1996.

[Fra 96] Fraga, J., Farines, J., Furtado, O., Siqueira, F., "Programação de Aplicações Distribuídas Tempo-Real em Sistemas Abertos, XXIII Seminário Integrado de Software e Hardware, Recife, PE, Agosto, 1996.

[Fra 97] Fraga, J., Maziero, C., Lung, L.C., e Loques, O. G., "Implementing replicated services in open systems using a reflective approach", Third International Symposium on Autonomous Decentralized Systems, Berlin, Abril, 1997.

[Gam 95] Gamma, E. et. *Alli*, "Design Patterns - Elements of Reusable Object-Oriented Software", Addison-Wesley Publishing Co., USA, 1995.

[Gar 94] Garlan, D., Allen, R., Ockerbloom, J, "Exploiting Style in Architectural Design Environments", SIGSOFT 94, pp. 175-188, New Orleans, Dezembro, 1994.

[Gol 97] Golm, M.; "Design and Implementation of a Meta Architecture for Java", M.Sc. Thesis, Instituto de Matemática, Erlanngen-Nüremberg University, Alemanha, Janeiro, 1997.

[Hel 92] Helm, R., Holland, I.M. e Gangopadhyay, D., "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", OOPSLA'90, Ottawa, Outubro, 1990.

[Iss 96] Issarny, V. and Bidan, C., "Aster: A CORBA-based Software Interconnection System Supporting Distributed System Customization", International Conference on Configurable Distributed Systems, Annapolis, Maio, 1996.

[Jac 98] Jackson, M, e Zave, P., "Distributed Feature Composition: A Virtual Architecture for Telecommunications Services", IEEE Trasaction on Software Enginnering, Vol. 24, No. 10, pp. 831-847, Outubro, 1998.

[Java] Arnold, K. e Gosling, J., "The Java Programming Language", Second Edition, Sun Microsystems, Addison Wesley, USA, 1998.

[Joh 97] Johnson, R. E., "Frameworks = (Components + Patterns)", Communications of the ACM, pp. 39-42, Outubro, Vol. 40, No. 10, 1997.

[Jon 93] Jones, M. B., "Interposition Agents: Transparently Interposing User Code at the System Interface", ACM 14th ACM Symposium on Operating Systems Principles, Asheville, 1993.

[Kic 96] Kiczales, G. and Irwing, J. et *alli*, "Aspect-Oriented Programming", Position Paper for the ACM Workshop on Strategic Directions in Computing Research, MIT, June, 1996.

[Kic 97] Kiczales, G., Lamping, J., Lopes, C. V., Maeda, C., Mendhekar, A., "Open Implementations Design Guidelines", 19th International Conference on Software Engineering, ACM Press, 1997.

[Lad 97] Laddaga, R. and Veitch, J., “Dynamic Object Technology”, Communications of the ACM, Vol. 40, No. 5, May, 1997.

[Lob 98] Lobosco, M., Loques, O. G. e Sztajnberg, A. "Configuração de Sistemas Distribuídos Baseada em Java", I SICON, Rio de Janeiro, Setembro, 1998.

[Lob 99] Lobosco, M. "R-RIO: Um Ambiente de Suporte à Construção e Evolução de Sistemas Distribuídos", dissertação de mestrado, em preparação, CAA/UFF, 1999.

[Lop 97] Lopes, Cristina I. V.; “D: A Language Framework for Distributed Programming”, Doctor of Philosophy Thesis, College of Computer Science, Northeastern University, Novembro, 1997.

[Loq 96] Loques, O., Leite, J., Botafogo, R. and Lobosco, M., “Configuração, Reflexão e CORBA: Algumas Considerações”, III Workshop do Projeto ASAP, UFCE, Fortaleza, Ceará, 1996.

[Loq 97] Loques, O., Botafogo, R., Leite, J., “A Configuration Approach for Distributed Object-Oriented System Customization”, IEEE Third International Workshop on Object-oriented Real-time Dependable Systems, Newport Beach, Fevereiro, 1997.

[Mag 92] Magee, J., Dulay, N., Kramer, J., "Structuring parallel and distributed programs", Int. Workshop on Configurable Distributed Systems, pp. 102-117, IEE, Março, 1992.

[Mal 96] Malucelli, V. V, "Babel – Construindo Aplicações por Evolução", dissertação de mestrado, DEE / PUC-RJ, Fevereiro, 1996.

[Mon 97] Monroe, R. T., Kompanek, A., Melton, R. e Garlan, D., "Architectural Styles, Design Patterns and Objects", IEEE Software, Vol. 14, No. 1, pp. 43-52, Janeiro de 1997.

[Oli 98] Oliva, A., e Buzato, L. E., "Composition of Meta-Objects in Guaraná", ECOOP 98, Workshop Reflection, Canada, 1998.

[OMG 96] OMG, “The Common Object Request Brocker: Architecture and Specification”, Revision 2.0, Julho, 1996.

[Rin 98] Rine, D. C., "Supporting Reuse with Object Technology", IEEE Computer, Vol. 30, No. 10, pp. 43-45, Outubro, 1997.

[Sha 95] Shaw, M., DeLine, R., *et alli*, “Abstractions for software architecture and tools to support them”, IEEE Transactions on Software Engineering, Vol. 21, No. 4, Abril, 1995.

[Szt 97] Sztajnberg, A. and Loques, O. G., “Reflexão Computacional e Multimídia Distribuida”, SBRC 97, São Carlos, SP, Brazil, May, 1997.

[Szt 99] Sztajnberg, A, “Cap 2 - Análise de Propostas Correlatas”, proposta de tema de tese de doutorado em preparação, COPPE/UFRJ, 1999.

[Weg 96] Wegner, P., “Interoperability”, ACM Computing Surveys, Vol. 28, No. 1, Março, 1996.

[Wel 98] Welch, Ian; Stroud, Robert. "*Dalang - A Reflective Java Extension*". Proceedings of the Second Reflexive Workshop at OOPSLA. Vancouver, 1998.