Interconexão Dinâmica de Objetos Distribuídos Java

Adrian C. Ferreira

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Interconexão Dinâmica de Objetos Distribuídos Java

Adrian C. Ferreira[1]

novembro de 1998

Banca Examinadora:

- Prof. Dr. Rogério Drummond (Orientador)
- Prof. Dr. Francisco Vilar Brasileiro[2]
- Prof. Dr. Ricardo Anido[3]
- Prof. Dr. Edmundo Madeira[4] (suplente)

[1] Projeto financiado pelo CNPq e pela FAPESP processo Nº 97/04245-3 e pela FAEP - Unicamp
[2] Departamento de Sistemas e Computação – UFPB
[3] Instituto de Computação - Unicamp
[4] Instituto de Computação - Unicamp

NIDADE BC
CHAMADA:
37249
229/99
D
PRE R$ 1,00
DAT 08/04/99
N.º CPD

CM-00122001-0

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Ferreira, Adrian Carlos

F413i Interconexão dinâmica de objetos distribuídos Java / Adrian C. Ferreira -- Campinas, [S.P. :s.n.], 1998.

Orientador : Rogério Drummond

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas distribuídos. 2. Linguagem orientada a objetos. 3. Java (Linguagem de programação de computador). I. Drummond, Rogério. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Interconexão Dinâmica de Objetos Distribuídos Java

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Adrian C. Ferreira e aprovada pela Banca Examinadora.

Campinas, 22 de janeiro de 1999.

Prof. Dr. Rogério Drummond (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

TERMO DE APROVAÇÃO

Dissertação defendida e aprovada em 18 de dezembro de 1998, pela Banca Examinadora composta pelos Professores Doutores:

Porf. Dr. Francisco Vilar Brasileiro
UFPb

Prof. Dr. Ricardo de Oliveira Anido
IC - UNICAMP

Prof. Dr. Rogério Drummond Burnier Pessoa de Mello Filho
IC - UNICAMP

Ao Sr. Wilson, à D. Lourdes, ao Luis e à Vanessa.

Agradecimentos

Gostaria de agradecer aos amigos do Laboratório A-HAND pela agradável companhia ao longo desses dois últimos anos. Em especial, sou muito grato pela ajuda e motivação que recebi de Alexandre Prado Teles, Carlos Alberto Furuti e Klaus Steding-Jessen.

Aos professores do Instituto de Computação da Unicamp, especialmente ao meu orientador Rogério Drummond, tanto pelas sugestões quanto pelo exemplo de dedicação ao trabalho; relevo também meus agradecimentos aos professores Ricardo Anido, Edmundo Madeira e ao professor Francisco Brasileiro da UFPB pela minuciosa revisão da tese e pelas importantes sugestões que apresentaram.

Aos amigos do GEEU e da turma msc96 pelos momentos de alegria e pelo companheirismo.

Obrigado também à FAEP, ao CNPq e à FAPESP que financiaram este projeto.

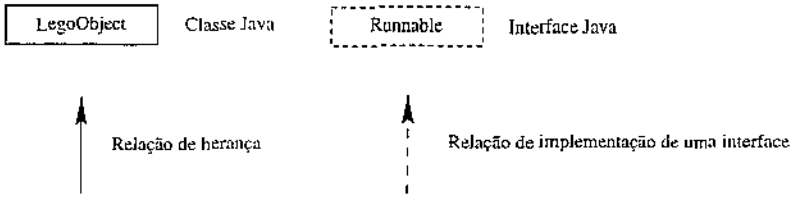
"Deus quer, o homem sonha, a obra nasce."
Fernando Pessoa

Convenções Tipográficas

Este documento utiliza as seguintes convenções:

Italic	O formato itálico é usado para termos de língua estrageira (*host*); nomes de conectores (*pipe*); para enfatizar uma idéia, por exemplo: uma interface Java *não* é uma classe. Este formato é usado ainda para conceitos cuja primeira definição está sendo apresentada, exemplo: Uma *thread* é um fluxo de execução contínua.
`Typewriter`	Usado para descrever palavras reservadas em linguagens ou para comandos de modo geral: `connect`, `rmic`, `char`.
Sans Serif	Usado para nomes de classes e nomes de métodos; por exemplo: um objeto da classe String.
$	Denota um *prompt* de *shell* para usuário comum.
LegoShell>	Indica o *prompt* da interface LegoShell.

Para figuras que representam hierarquias de classes, valem as seguintes convenções:



Resumo

O desenvolvimento de aplicações distribuídas, quando não indispensável, permite um aproveitamento mais eficiente de recursos. Entretanto, esses programas ainda exigem muito esforço de programação se comparado com o desenvolvimento de programas convencionais não-distribuídos. Existe portanto a necessidade de mecanismos que facilitem o trabalho do desenvolvedor.

As tecnologias disponíveis atualmente ainda dificultam a criação de objetos remotos, ajudando apenas a interconexão estática: as ligações entre os objetos devem ser definidas em tempo de compilação. Além disso, não há um padrão de comunicação entre os objetos: cada aplicação constrói suas próprias conexões. Outro aspecto que precisa ser aprimorado é o de mecanismos que efetivamente elevem o nível de abstração para o desenvolvedor, tando para a criação de objetos remotos quanto para a configuração das aplicações.

Este trabalho apresenta a versão inicial de um ambiente para desenvolvimento de aplicações distribuídas onde os objetos são facilmente criados e sua configuração pode ser realizada depois que os objetos foram instanciados. A comunicação entre esses objetos pode utilizar conectores que são vias previamente definidas e com infra-estrutura mais elaborada para fornecer serviços associados ao estabelecimento de uma conexão.

Abstract

Distributed application development is, in some cases a good choice for improving resources usage; in other cases it is even more important and the developer's only choice. On the other hand, this kind of programming needs hard development work when compared with non-distributed application programming.

Today, creation of remote objects is not simple; besides connections are static: object binding must be defined before code compilation. Another difficulty arises as there is no standard in object communication. Therefore, it is necessary to develop resources for high level distributed programming.

This project presents the first version of an environment for distributed application development where distributed objects are created as simply as a local one. Objects can be configured dynamically and connectors (channels with incorporated semantics) can be used to establish communication between objects in a high level programming.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Exemplos

Capítulo 1

Introdução

As *aplicações centralizadas*, também chamadas de programas convencionais, têm que executar em uma única máquina. *Aplicações distribuídas*, por outro lado, podem utilizar recursos remotos para processar operações de modo descentralizado. Basicamente, a capacidade de utilização da rede é que distingue esses dois tipos de aplicações.

Programas convencionais são, por natureza, mais simples. Seu desenvolvimento pode contar com metodologias experimentadas, com modernas linguagens de programação além de outros tradicionais recursos de programação. Como não transmitem dados via rede, normalmente os programas centralizados podem ser executados com mais eficiência que sua versão distribuída. A facilidade com que podem ser desenvolvidos e a eficiência, são as principais vantagens dos programas convencionais.

Entretanto, por não compartilhar recursos da rede, os programas convencionais possuem limitações. Estas e várias outras motivações para se desenvolver aplicações distribuídas são descritas a seguir.

Natureza da aplicação : em muitos casos, uma solução centralizada não é possível porque o problema é inerentemente distribuído. Podem ser citados como exemplo, todas as aplicações que oferecem serviços via rede.

Otimização de recursos : uma grande motivação para executar aplicações em ambientes distribuídos está relacionada com a melhor utilização de recursos. Mesmo que não se tenha grande poder de processamento disponível localmente, o acesso à rede e à tecnologia de construção de programas distribuídos possibilita soluções através da utilização de recursos remotos. Alguns programas exigem recursos consideráveis e, caso uma solução distribuída não seja implementada, todos esses recursos precisariam ser centralizados. Um aspecto negativo desse tipo de solução é que muitas vezes esses recursos não são intensamente utilizados. Uma aplicação que realmente os utilize pode estar sendo executada muito eventualmente subutilizando o poder de processamento. Com a tecnologia de sistemas distribuídos, máquinas muito bem configuradas podem não ser necessárias; nos casos onde são realmente indispensáveis, poderão ainda funcionar como servidoras para clientes remotos.

Paralelismo : um programa que exige grande poder de processamento pode ter suas tarefas subdivididas e executadas em várias máquinas. Com o processamento paralelo e distribuído, é possível conseguir maior eficiência na obtenção do resultado final. No entanto, nem todos programas podem se beneficiar dessa tecnologia. Conforme descrito em [dO97], antes de optar por distribuir as tarefas de uma aplicação com o objetivo de melhorar a eficiência, devem ser avaliados aspectos como: a natureza da aplicação, o algoritmo a ser utilizado e o número de mensagens necessárias na comunicação.

Tolerância a falhas : com maiores opções de processamento, uma aplicação poderia continuar executando mesmo na presença de algumas falhas.

Balanceamento de carga : aplicações centralizadas podem sobrecarregar um determinado equipamento apenas em alguns períodos. Esse problema pode ser amenizado com aplicações distribuídas e configuráveis, nesse caso o usuário poderia escolher em qual máquina deseja executar sua aplicação.

Programas distribuídos podem solucionar a falta de recursos computacionais, aumentar a capacidade de processamento, melhorar a confiabilidade do sistema e, em muitos casos, pode não haver outras alternativas. No entanto, o principal problema é que comparativamente, sua implementação é mais complexa — além do problema em si, é preciso considerar o processo de comunicação, o sistema de rede, além das características dos equipamentos envolvidos.

Em muitos casos, uma implementação distribuída envolve grande dificuldade e pode representar um verdadeiro desafio de programação. Um dos principais problemas é que o programador precisa tratar questões relacionadas com os aspectos de comunicação enquanto poderia estar preocupado apenas com os aspectos de sua aplicação.

Uma aplicação distribuída é por natureza mais complexa do que uma centralizada. No entanto, as maiores dificuldades atualmente existentes, não são conseqüências desses aspectos inerentes às aplicações distribuídas, mas de outros problemas que podem ser amenizados se o desenvolvedor puder contar com mecanismos de abstração mais elaborados. Por exemplo, todo o processo de interligação poderia ser realizado de modo transparente para o programador.

A situação ideal seria a utilização de um ambiente de programação onde uma aplicação distribuída pudesse ser implementada como se fosse centralizada. As tecnologias atualmente disponíveis facilitam mais as tarefas de interconexão de objetos. Essas funcionalidades já representam um avanço substancial se comparado com as tecnologias anteriores (RPC e *sockets*), no entanto, muitos outros recursos precisam ser providos, principalmente no que diz respeito ao nível de abstração permitido ao programador.

Atualmente, existem recursos para que uma via de comunicação seja estabelecida, quem a constrói de fato é o próprio desenvolvedor da aplicação. Além do trabalho extra, o resultado final são aplicações sem um padrão de comunicação bem definido. Ao invés de simplesmente permitir a criação dessas vias, o sistema poderia fornecer mecanismos para que as conexões pudessem ser estabelecidas mais facilmente e de modo padronizado.

Também não há recursos que facilitem a configuração dinâmica das aplicações. Atualmente, uma aplicação precisa definir seus recursos no momento em que é compilada. Seria importante permitir que o usuário interconectasse as partes de seu programa da maneira que lhe for mais conveniente no momento da execução. Esse dinamismo é fundamental para permitir que os mecanismos da rede sejam bem aproveitados porque o comportamento da rede é imprevisível.

Outro aspecto que falta ser desenvolvido é a integração de funcionalidades aos mecanismos de conexão: por exemplo, atualmente as conexões não oferecem recursos de gerenciamento, sempre que há necessidade de monitoramento do canal de comunicação, isso tem que ser feito pelo desenvolvedor da aplicação. Outro exemplo de funcionalidade que pode ser integrada ao canal de comunicação é um sistema de controle de entrega de mensagens por *multicast*: um servidor usaria esse recurso para enviar mensagens a um subconjunto do seu grupo de clientes.

O objetivo deste trabalho é oferecer recursos de programação para facilitar o desenvolvimento de aplicações distribuídas e configuráveis dinamicamente. O sistema que foi implementado é uma adaptação da proposta inicial da linguagem de programação *LegoShell* [Dru95]. Neste projeto, algumas propriedades originais foram preservadas e outras, como a interface gráfica, não foram implementadas.

Faz parte deste trabalho, o mecanismo de interligação de objetos denominado *conectores* que são objetos LegoShell com a finalidade de prover interligação e oferecer outros serviços associados ao conal de comunicação. Os conectores são explicados mais detalhadamente na seção 5.4.

A LegoShell foi desenvolvida em Java e fornece suporte para implementação de objetos distribuídos Java. Objetos locais ou remotos podem ser criados com a mesma facilidade. Esses objetos são executados apenas quando solicitados, desse modo, um grande número de serviços podem ser disponibilizados em cada

máquina sem sobrecarregar os recursos de processamento. Aplicações são conjuntos de objetos executados sob demanda e configuráveis de acordo com as necessidades do usuário.

O desenvolvedor tem a visão da rede como a de um grande provedor de objetos. Depois de criados e configurados para compor uma aplicação, os objetos podem terminar sua execução e ser reconfigurados para compor outra aplicação diferente. Todos os objetos são potencialmente remotos e podem ser usados em qualquer localidade da rede.

O sistema que foi desenvolvido, assim como os conceitos essenciais para sua compreensão, serão descritos ao longo deste documento segundo a organização descrita a seguir.

No capítulo 2, serão apresentados os conceitos de objetos distribuídos, a sub-área de sistemas distribuídos em que este projeto está inserido.

O capítulo 3 avalia as tecnologias mais usadas atualmente no desenvolvimento de aplicações distribuídas. O padrão CORBA e os produtos OrbixWeb, Voyager e Java/RMI são apresentados juntamente com exemplos de utilização. Os conceitos básicos de Java também são introduzidos neste capítulo por ser a linguagem usado pelos três produtos citados.

O capítulo 4 apresenta mais detalhadamente os recursos de Java que foram utilizados na implementação deste projeto.

A LegoShell, ambiente que centraliza as funcionalidades deste projeto, é apresentada no capítulo 5.

O capítulo 6 descreve detalhadamente as funcionalidades da LegoShell, dos conectores e como foram implementados.

Finalmente, o capítulo 7 discute o que foi alcançado, propõe a implementação de novas funcionalidades e apresenta as conslusões.

Capítulo 2

Objetos Distribuídos

Os conceitos de Objetos Distribuídos basicamente situam o contexto em que está inserido este trabalho. Embora seja uma tecnologia recente, os principais fundamentos dessa área de pesquisa são oriundos dos conceitos de programação orientada a objetos e de sistemas distribuídos, duas tecnologias mais consolidadas.

2.1 Programação Orientada a Objetos

O paradigma de Programação Orientada a Objetos, ou simplesmente POO, surgiu como linguagem de programação no início dos anos 80. Uma das primeiras propostas de uma linguagem puramente orientada a objetos foi Smalltalk-76 [Ing78] que deu origem a Smalltalk-80, uma das primeiras linguagens orientadas a objetos. Atualmente, Smaltalk, Ada, Eiffel, C++ e Java são exemplos de linguagens orientadas a objetos, sendo que as duas últimas são as mais utilizadas comercialmente.

POO surgiu como resultado da busca por uma tecnologia que facilitasse o processo de construção de programas. Os problemas clássicos do modelo estruturado são: baixa reusabilidade de código, limitado poder de abstração, além da grande complexidade nos processos de implementação, depuração e manutenção do código. O tempo médio gasto na construção e manutenção de uma aplicação estruturada é considerado excessivo.

A concepção de um sistema orientado a objetos é um processo mais intuitivo que aquele usado no modelo de programação estruturada. O sistema modelado é mais próximo da realidade do problema que está sendo resolvido. Basicamente, a abstração consiste em descrever as características estáticas do sistema usando entidades denominadas de objetos. As relações entre esses objetos são modeladas pela troca de mensagens ou chamadas de métodos e representam as propriedades dinâmicas do sistema. Em resumo, um programa é composto por um conjunto de objetos que interagem trocando mensagens, isto é, executando métodos e recebendo argumentos de retorno.

Essa representação é válida para todas as etapas de desenvolvimento do sistema. Todas as etapas envolvidas no processo de criação do sistema (análise, projeto, implementação e depuração) podem se beneficiar dos conceitos da orientação a objetos.

Os principais conceitos envolvidos em POO, e presentes nas linguagens orientadas a objetos, são descritos em [RBP+91] e [Boo91] e apresentados resumidamente a seguir.

Classe : é a estrutura básica onde são definidos os dados e os códigos para entidades do programa que possuem as mesmas características. Na nomeclatura de POO, uma classe possui atributos e métodos e suas instâncias são denominadas de *objetos*. Tipicamente, atributos e métodos são codificados em uma classe para definir propriedades de um grupo de objetos. Há dois tipos de atributos: *atributos*

estáticos, também chamados de atributos de classe e *atributos não-estáticos* que são criados para cada instância da classe e, portanto não são compartilhados.

Objeto : os objetos, instâncias de uma classe, são criados em tempo de execução. Enquanto uma classe é apenas a definição de propriedades, um objeto incorpora tais propriedades para compor uma aplicação. A criação de um objeto é feita através da chamada ao *construtor*, um método especial que toda classe instanciável possui. Esse processo é conhecido como *instanciação de classe*.

Herança : as classes são estruturadas hierarquicamente em *superclasse* e *subclasse* obedecendo o conceito de herança. Uma subclasse herda os atributos e os métodos definidos em sua superclasse. Há dois tipos de herança, simples e múltipla. Em *herança simples*, uma classe só pode definir *uma* outra classe como sua superclasse. *herança múltipla* permite que uma classe seja derivada de várias outras classes. Java é um exemplo de linguagem que implementa herança simples; já em C++, a herança pode ser múltipla.

Interface : um objeto tem seus atributos e métodos públicos acessíveis através de sua *interface*. Apenas a interface de uma classe, e não todos os detalhes de sua implementação, precisa ser consultada para a obtenção de serviços.

Encapsulamento : uma classe provê uma interface com a publicação dos métodos e atributos que podem ser acessíveis externamente. Segundo o conceito de *encapsulamento*, os serviços só podem ser acessados através dessa interface. Isso permite obter serviços de outros métodos sem a preocupação de como foram implementados, apenas consultando sua interface. Mesmo que possível, não é aconselhável acessar os atributos de outro objeto a não ser através de seus métodos.

Polimorfismo : um método definido em uma superclasse e reimplementado nas subclasses, pode produzir resultados diferentes quando executado sobre objetos das subclasses. Uma subclasse pode redefinir um método de sua superclasse para especificar um comportamento particular aos seus objetos. *Polimorfismo* é propriedade que permite que dois objetos de um mesmo tipo respondam à mesma mensagem de modo diferente. Isso é possível porque o método a ser chamado é resolvido apenas em tempo de execução.

Em POO, o programador se concentra nos dados e nas operações sobre eles, enquanto que na programação estruturada a atenção fica voltada para as funcionalidades de cada parte que compõe o sistema. Atualmente, POO está consolidada como o paradigma que mais recursos oferece ao desenvolvimento de sistemas, sobretudo os maiores e mais complexos.

2.2 Sistemas Distribuídos

De maneira simplificada, *sistemas distribuídos* utilizam os recursos da rede para executar suas operações. Os conceitos normalmente encontrados na literatura são variados, algumas vezes uma ou outra definição é mais apropriada para os contextos da época em que foram criadas. Segundo Silberschatz [PS91], um sistema distribuído é uma coleção de processos que não compartilham relógios ou memória. Para Tanenbaum [Tan92], um sistema distribuído é aquele que se apresenta aos usuários como um sistema centralizado, mas que na realidade, suas funções são realizadas por várias máquinas interligadas. Atualmente, o conceito de sistemas distribuídos é mais amplo do que as definições acima. Alguns exemplos de aplicações distribuídas são: correio eletrônico E-mail, servidor de nomes DNS, sistema de arquivos em rede (NFS), serviços oferecidos via Internet e aplicações que executam em rede corporativa.

Sistemas distribuídos são, por natureza, mais complexos que os sistemas centralizados. O grande desafio é desenvolver tecnologia para permitir que recursos distantes e acessíveis através da rede possam ser mais facilmente desenvolvidos.

O problema consiste em permitir interoperabilidade entre aplicações desenvolvidas em diferentes linguagens de programação, executadas em arquiteturas e sistemas operacionais diversos e que usam uma variedade de protocolos de comunicação com a rede. A solução para esse problema está na união de POO — para prover serviços encapsulados e facilitar reusabilidade — e a tecnologia dos sistemas distribuídos que oferecem infra-estrutura básica para permitir abstração dos serviços de comunicação. Essa recente área de estudos é conhecida como Objetos Distribuídos [OHE96].

2.3 Objetos Distribuídos

Objetos Distribuídos é a tecnologia que une os conceitos de programação orientada a objetos e sistemas distribuídos. Recentes avanços nessa área de pesquisa têm permitido o desenvolvimento de aplicações distribuídas e orientadas a objetos. Isso representa um grande avanço se comparado com a tecnologia de *RPC* (chamada de procedimentos remotos) onde as aplicações distribuídas são desenvolvidas segundo a perspectiva da programação estruturada.

O modelo usado no desenvolvimento de aplicações distribuídas emprega os conceitos de arquitetura cliente/servidor [OHE96]. Há uma distinção bem clara entre um cliente e um servidor, embora uma mesma entidade possa exercer as duas atividades. Um cliente é a entidade que solicita o serviço e o servidor é a que recebe essa chamada. Essas duas entidades se relacionam para trocar serviços que podem ser uma chamada a um procedimento, no conceito de RPC, uma consulta a um banco de dados nos sistemas gerenciadores de banco de dados distribuídos, ou uma chamada a um método na tecnologia de objetos distribuídos.

As primeiras aplicações distribuídas se limitavam ao uso de um sistema de arquivos distribuídos. Em seguida, evoluíram para o acesso a bancos de dados remotos com os clientes separados do servidor. Outra tecnologia mais avançada emprega os mecanismos de RPC e permitiram um grande avanço para a área. Com RPC, os recursos de processamento distribuído puderam ser mais amplamente utilizados. No entanto, RPC impõe algumas limitações ao desenvolvedor. Como as aplicações modernas são orientadas a objetos, o ideal é acionar métodos desses objetos remotos e não apenas fazer chamadas remotas de procedimentos como em RPC. A tecnologia de objetos distribuídos deverá substituir a grande maioria das demais tecnologias para desenvolvimento de aplicações distribuídas [OH98]. Atualmente, há quase um consenso em torno das vantagens da tecnologia de objetos distribuídos para o desenvolvimento de aplicações distribuídas.

A tecnologia de objetos distribuídos representa um importante avanço para a área de desenvolvimento de aplicações distribuídas. Esse avanço pode ser comparado ao que a tecnologia de orientação a objetos vem representando com relação à programação estruturada na área de programação convencional.

Capítulo 3

Desenvolvimento de Aplicações Distribuídas

Aplicações distribuídas representam uma solução conveniente para um grande número de problemas e, ao mesmo tempo, exigem esforço muito maior para sua construção se comparado com a elaboração de sistemas convencionais não-distribuídos. Como essas aplicações usam recursos de outras máquinas que podem estar em outras localidades e sobre diferentes plataformas, é necessário que um padrão de comunicação seja estabelecido.

O desenvolvimento de aplicações cliente/servidor na Internet exige muito esforço técnico podendo ser considerado um desafio em conseqüência da diversidade de equipamentos e de tecnologias usadas. Idealmente, um ambiente para desenvolvimento dessas aplicações deveria permitir a construção de sistemas independentes da plataforma e de linguagem de programação.

Atualmente, há uma grande variedade de propostas e tecnologias destinadas a facilitar trabalho de desenvolvimento de aplicações distribuídas. Esses programas são denominados de *middleware*: ferramentas que padronizam os mecanismos de comunicação através da rede e provêem funcionalidades para facilitar esse trabalho.

Distributed Computing Enviroment (DCE) estabelece um padrão de comunicação via RPC e foi a tecnologia mais influente antes das tecnologias que suportam objetos distribuídos. RPC permite que o fluxo de controle em um programa mude de um processo para outro processo possivelmente localizado em outra máquina. Os ambientes podem ser heterogêneos onde a arquitetura ou a linguagem do programa que está sendo chamado não são conhecidos.

Distributed Component Object Model (DCOM) é a tecnologia da Microsoft para a comunicação entre objetos distribuídos. Na versão atual, Windows NT 4.0, DCOM é uma tecnologia considerada imatura que não facilita o desenvolvimento de grandes aplicações distribuídas; além de estar limitada aos sistemas da Microsoft. A nova versão no NT deverá surgir no final deste ano ou início de 1999. Com ela, podem surgir ferramentas que facilitem o desenvolvimento mas dificilmente permitirá integração com os demais sistemas.

O objetivo deste capítulo é apresentar as opções que foram avaliadas para o desenvolvimento deste projeto em particular, ou seja, CORBA, RMI e Voyager que serão discutidas nas próximas seções. Outras tecnologias como DCOM e DCE, embora também influentes, não fazem parte deste projeto. Em [DC98] há um importante estudo compartivo entre as tecnologias DCE, DCOM e CORBA. Conforme este artigo, DCE é recomendável apenas para atualização de sistemas que já foram implementados usando programação estruturada; não é recomendável do ponto de vista tecnológico desenvolver aplicações novas em RPC porque atualmente existem recursos que oferecem mais facilidades. DCOM é uma tecnologia proprietária e dependente de plataforma. Além de possuir um modelo de objetos não intuitivo, sua utilização é considerada

complicada [DC98, pag. 30].

3.1 CORBA

A OMG (*Object Managment Group*) surgiu em 1989 e conta hoje com aproximadamente 800 membros entre instituições de pesquisa, empresas e desenvolvedores de *software*. O interesse da OMG é definir um padrão para a comunicação entre objetos distribuídos. É a pioneira e a maior iniciativa nesta área e suas recomendações se encontram em um estágio bastante avançado.

O objetivo da OMG é obter consenso entre fabricantes e usuários para a definição de uma infra-estrutura para objetos distribuídos e permitir que componentes de aplicações distribuídas desenvolvidas em ambientes e linguagens heterogêneas se comuniquem.

Em 1991, a OMG anunciou a versão 1.0 de CORBA (*Common Object Request Broker Architecture*) que especifica facilidades de comunicação para um ambiente de objetos distribuídos — um mecanismo padrão para um objeto acessar os estados públicos de outros objetos e para exportar funcionalidade. Em março de 98 foi lançado CORBA 2.2[1].

3.1.1 IDL

CORBA especifica IDL (*Interface Definition Language*) como uma linguagem para definição de interfaces e cuja função é permitir que objetos se comuniquem sem restrições quanto a linguagem em que foram implementados. Por intermédio da IDL, um cliente desenvolvido em uma linguagem pode acessar os métodos de um servidor implementado em outra linguagem.

A interface IDL é o ponto comum entre as linguagens de programação; os serviços devem ser providos e acessados através dessa interface. O programador, ao desenvolver uma aplicação, usa os tipos suportados por sua linguagem; o mapeamento desses tipos para os da IDL é feito automaticamente por um ORB (*Object Request Broker*). Os atributos e operações especificados pelo programador em IDL, são usados pelo ORB na geração de código para desempenhar as funções de estabelecimento de conexão, serialização dos dados, localização e chamada dos objetos.

Os principais aspectos dessa interface são descritos resumidamente a seguir, uma explicação mais detalhada pode ser encontrada em [VD97]; a documentação completa pode ser acessada nas páginas da OMG[2].

Tipos de dados : em IDL são válidos os seguintes tipos básicos: `short`, `long`, `float`, `double`, `char`, `boolean`, `string`, `octet`, `enum` e `any`; além dos seguintes tipos estruturados: `structure`, `array`, `sequence` e `exception`. O programador precisa conhecer a maneira como o ORB faz o mapeamento entre os tipos IDL e os da linguagem. No caso de Java, o mapeamente dos tipos primitivos é direto; no caso dos tipos estruturados, é necessário conhecer os detalhes de mapeamento.

Atributos e operações : são as ações que podem ser requisitadas por um cliente CORBA, como descrito em [VD97]. Um atributo é mapeado para um par de operações: uma para o cliente obter esse dado e outra para alterá-lo. A palavra reservada `readonly` pode ser usada para restringir as operações de alteração desses dados. As operações têm um nome, um valor de retorno, uma lista de parâmetros e uma lista de exceções. Essas informações formam a assinatura do método.

Herança : uma interface pode estender as funcionalidades definidas em outras interfaces já existentes, há o conceito de herança simples e múltipla. Uma interface de um nível mais alto da hierarquia pode ser usada sempre que uma interface de nível inferior for requisitada, ou seja, existe o conceito de

[1] Especificação CORBA 2.2 acessível em: `http://www.omg.org/corba/corbaiiop.html`
[2] Documentação IDL: `http://www.omg.org/corba/cichpter.html`

polimorfismo em IDL. Quando não se deseja definir objetos de uma classe específica, o tipo any pode ser usado na interface IDL para objetos genéricos.

Escopo de nomes : interfaces podem ser definidas em *módulos* para agrupar conjuntos de interfaces e evitar conflitos de nomes com outras interfaces de outros módulos. Nomes de interface são relativos a contextos. O nome completo, formado pela concatenação do nome do módulo mais o nome da interface, só precisa ser usado se o escopo for externo ao módulo.

Em [VD97], há uma explicação detalhada sobre esses conceitos, incluindo exemplos de utilização.

Muitas linguagens, entre elas Java, já contam com ferramentas que fazem o mapeamento para IDL. Embora esse mapeamento só tenha sido oficialmente definido pela OMG em junho de 1997, ORBs para Java, como por exemplo OrbixWeb e Visibroker, já estão disponíveis no mercado há mais tempo.

3.1.2 IIOP

Em CORBA 2.0 foi incluída a especificação do protocolo IIOP (*Internet Inter-Orb Protocol*) cujo objetivo é permitir interação entre objetos implementados em diferentes ORBs. Obedecendo-se essa padronização, dois objetos podem se comunicar independente do produto que foi usado em sua implementação. No entanto, conforme [DC98, pag. 38], essa capacidade de interoperabilidade ainda é limitada. Há problemas relacionados com os serviços implementados por diferentes fornecedores, a convenção de nomes internos não é sempre compatível e o protocolo utilizado nem sempre é o mesmo. O resultado é que na prática, pelo menos até que a OMG consiga solucionar esse problema, o usuário CORBA, assim como o usuário DCOM, está limitado aos produtos de apenas um fornecedor. Uma solução para esse problema é esperado ainda para este ano de 1998.

Em CORBA, uma referência para objeto, ou IOR (*Interoperable Object Reference*) é uma estrutura de dados que contém informações suficientes para que o ORB consiga localizar a máquina, o servidor e o objeto a ser chamado. Desse modo, uma referência em um ORB pode ser usada na implementação de um cliente em outro ORB.

3.1.3 *Frameworks*

Os serviços descritos anteriormente permitem o desenvolvimento de aplicações distribuídas em CORBA mas não oferecem um nível de abstração satisfatório. Para que objetos se comuniquem através do barramento ORB, é necessário definir interfaces usando IDL; no entanto as classes podem ser construídas de variadas maneiras, o que muitas vezes é indesejável e dificulta a utilização ou a reusabilidade do que foi desenvolvido.

A proposta de CORBA envolve um modelo mais elaborado onde as aplicações podem ser desenvolvidas com alto grau de abstração e com grande chance de reusablidade de código. Esse modelo proposto pela OMG é conhecido como tecnologia de *framework*. Essa tecnologia provê um ambiente organizado para a execução de uma coleção de objetos e visa garantir uma construção mais simples e organizada dos componentes.

O uso de *frameworks* permite interrelacionar os componentes dinamicamente sem a necessidade de preocupação com os detalhes de sua implementação. O uso de classes abstratas é muito útil na concepção de pequenos programas, enquanto que *frameworks* são mais adequados para elaboração de projetos que envolvem um grande número de objetos. Em sistemas complexos, o uso de *frameworks* permite um alto grau de reusabilidade. Um *framework* não é apenas um conjunto de classes, existe uma grande funcionalidade por trás das interconexões entre os objetos para permitir um alto nível de abstração para o usuário.

As linguagens de programação visual poderiam ser vistas como um *framework* já que possuem as principais características descritas acima. Em *frameworks*, ao contrário dessas linguagens, o código pode ser alterado pelo usuário. Se um *framework* satisfaz o desenvolvimento de um componente, então basta programar algumas linhas; caso contrário, o código do *framework* pode ser alterado por um conjunto de classes

desenvolvidas pelo usuário. Como *frameworks* definem a arquitetura de uma aplicação, o seu desenvolvimento deve levar em consideração as necessidades de flexibilidade e extensibilidade. Atualmente, a maioria dos *frameworks* são para nível de sistema mas várias definições por parte da OMG são esperadas para objetos de aplicação.

No projeto de um *framework*, leva-se em conta o fornecimento de um conjunto substancial de funcionalidades para permitir que o usuário tenha que codificar o mínimo possível. No entanto, um *framework* não pode ser muito específico para não perder em generalidade, o que o tornaria útil a uma classe restrita de aplicações. O maior problema encontrado na definição de um *framework* está relacionado com o nível de abstração que será implementado. Caso se opte por um *framework* muito genérico, sua utilização não será tão simples quanto a de um *framework* mais específico. Por outro lado, um *framework* muito específico terá o seu uso restringido a um pequeno grupo de aplicações.

Uma vez que a programacão distribuída puder contar com um grande número de *frameworks*, o desenvolvimento de aplicações distribuídas deverá envolver pouca codificação e mais integração de componentes já testados e disponíveis para o programador.

3.1.4 Serviços CORBA

Na proposta de CORBA, os serviços devem estar disponíveis para o desenvolvedor de uma aplicação em níveis ou classes de serviços, conforme mostrado na figura 3.1.

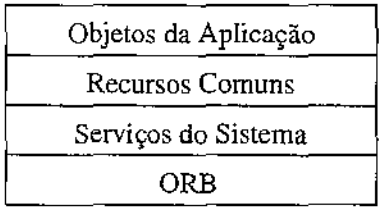


Figura 3.1: Hierarquia de serviços CORBA

Todos esses serviços são abordados resumidamente a seguir. Para maiores detalhes, a especificação da OMG para cada um desses serviços pode ser acessada[3].

ORB (*Object Request Broker*)

No nível mais baixo está o barramento ORB, o principal elemento de CORBA. Esse barramento intermedia a comunicação entre os objetos com a função de dar transparêcia às chamadas de métodos.

O cliente interage com o ORB usando um *stub* e o servidor utiliza um *skeleton*. O *stub* é responsável pela geração e encaminhamento da chamada de um método. O *skeleton* recebe uma chamada e fica responsável por encontrar o objeto que implementa o serviço solicitado, passar os parâmetros, executar o método e devolver o resultado ao cliente.

A chamada e recebimento de mensagens podem ser implementadas de duas maneiras: estática ou dinâmica, cada uma delas é discutida a seguir.

Estática

A interface IDL, é conhecida em tempo de compilação. O uso desse modelo requer a compilação da IDL para gerar código *stub* no lado do cliente e *skeleton* no lado do servidor. A maioria das aplicações são

[3]Serviços CORBA: `http://www.omg.org/corba/sectrans.html`

implementas dessa maneira: a IDL é conhecida tanto pelo cliente quanto pelo servidor.

Dinâmica

Em alguns casos, a IDL é desconhecida em tempo de compilação. As operações e atributos podem ser conhecidos apenas em tempo de execução. Nesses casos, o cliente utiliza invocação dinâmica de interface (DII — *Dynamic Invocation Interface*) para acessar um servidor sem conhecer sua interface IDL. A vantagem é que o cliente pode ser compilado para acessar um servidor cuja interface é desconhcida ou nem sequer foi implementado. Para o servidor, não importa se o cliente construiu sua chamada usando DII ou se a IDL foi usada durante a compilação.

DSI (*Dynamic Skeleton Interface*) é equivalente a DII para o lado servidor. Esse mecanismo permite que um servidor receba chamadas de atributos e operações, mesmo que a interface IDL não esteja definida em tempo de compilação. O servidor pode definir uma função que será informada sobre a chegada de uma chamada a uma operação ou atributo. O cliente não tem conhecimento se o servidor atende sua chamada usando DSI ou se o servidor compilou a IDL para gerar um *skeleton*.

Serviços do Sistema (*object services*)

A OMG define uma série de serviços considerados essenciais como servidor de nomes, serviço de eventos, serviço de tempo, serviço de ciclo de vida e serviço de segurança; tipicamente são as funcionalidades básicas para o desenvolvimento de aplicações distribuídas.

As chamadas aos serviços do sistema são usadas, por exemplo, para criar, nomear, introduzir um objeto no ambiente distribuído ou para localizar um objeto provedor de um determinado serviço. Os serviços do sistema aumentam as funcionalidades do barramento ORB. Vários serviços de sistema já foram definidos e padronizados pela OMG, outros ainda estão em fase de discussão em RFP´s (*Request For Proposal*) e deverão ser especificados em pouco tempo.

Recursos Comuns (*Common Facilities*)

Em um nível mais elevado de abstração, se encontram os recursos comuns que são a mais nova área de padronização em CORBA. Esses recursos são componentes implementados como *frameworks* e representam o mais alto nível de recursos disponíveis para o desenvolvedor de aplicações. Recursos comuns são coleções de componentes definidos em IDL e disponíveis para os objetos da aplicação. Existem duas categorias de recursos comuns: horizontais e verticais.

Recursos horizontais : devem concentrar as funções exigidas pela maioria das aplicações. Atualmente não existem muitos *frameworks* para auxiliar na construção de aplicações mas um grande número deles já foram definidos pela OMG.

Recursos verticais : proverá interfaces IDL e *frameworks* para desenvolvimento de aplicações em áreas específicas como automação industrial, processamento de imagens, exploração de petróleo e outras infinidades de áreas que necessitam de recursos específicos. Um recurso comum a várias áreas específicas tende a se tornar um recurso horizontal.

Objetos da Aplicação (*business objects*)

O usuário deverá implementar suas aplicações no nível de Objetos da Aplicação que é o nível mais alto da hierarquia.

São definidos para o desenvolvimento de aplicações pelo usuário. Esses objetos devem utilizar os recursos do barramento ORB, os serviços do sistema e os recursos comuns. Uma aplicação é tipicamente construída a

partir da cooperação entre componentes formados por objetos do sistema. Os limites entre recursos comuns e aplicações, assim como os limites entre os serviços do sistema e os recursos comuns, não são concretamente definidos.

Atualmente, os dois produtos mais utilizados para geração de aplicações Java dentro do padrão CORBA são OrbixWeb[4] da IONA e Visibroker[5] criado pela Visigenic que em março de 98 foi adquirida pela Borland, empresa que em abril de 98 passou a se chamar Inprise. Esses dois produtos implementam as funcionalidades básicas definidas por CORBA, algumas facilidades podem ser encontradas apenas em um dos produtos, mas de modo geral os dois ORBs podem ser considerados equivalentes. OrbixWeb foi adquirido pelo laboratório A-HAND e será o ORB apresentado a seguir.

3.2 OrbixWeb

OrbixWeb é um ORB para linguagem Java desenvolvido pela IONA, empresa filiada à OMG e fundada em 1991. Este programa conta com a experiência que a empresa adquiriu com seu primeiro produto: ORBIX, um ORB para C++ lançado em 1993.

OrbixWeb mantém a independência de plataforma existente em Java pois o código gerado é *100% pure Java*, podendo ser usado em qualquer plataforma que suporte a máquina virtual Java (JVM).

Em CORBA, uma aplicação distribuída é composta por objetos que são executados em diferentes localidades. Servidores fornecem funcionalidades para clientes através de métodos. Uma aplicação pode ser ora cliente - quando solicita um serviço — ora um servidor — quando fornece um serviço. Em geral, servidores implementam classes que os clientes podem instanciar remotamente e obter uma referência para o objeto criado. Uma referência para um objeto remoto pode ser obtida de várias formas. Por exemplo, se o cliente tem conhecimento do nome do servidor que deseja acessar, pode usar essa informação para se conectar ao objeto remoto. O servidor pode, alternativamente, publicar sua identificação no servidor de nomes que pode ser acessado polos clientes. Outra maneira, é buscar um objeto não a partir de informações a respeito de um objeto específico, mas por seus serviços. Essa funcionalidade deve ser provida pelo *trader* que é mais um serviço definido por CORBA.

As próximas seções apresentam os principais conceitos e características da tecnologia OrbixWeb.

3.2.1 Interface IDL

IDL, vide seção 3.1.1, é uma linguagem padronizada pela OMG para ser usada na definição das interfaces dos servidores. Sua utilização visa permitir a comunicação entre objetos implementados em diferentes linguagens de programação. Depois de definir a interface de um objeto em IDL, o programador está livre para escolher a linguagem em que deseja implementar as funcionalidades desse servidor. Do mesmo modo, programadores que desejam utilizar tais objetos podem usar qualquer linguagem de programação para acessar suas funcionalidades. OrbixWeb faz o mapeamento da interface IDL para a linguagem Java e fornece suporte para a construção de aplicações clientes e servidoras usando Java.

3.2.2 Cliente

Um objeto remoto é representado por um *objeto proxy* cujo código é gerado automaticamente pelo ORB. Um *proxy* é uma representação local de um objeto remoto. Para se ligar a um objeto servidor, o cliente faz uma chamada ao método bind que retorna um *proxy*. O cliente interage com o *proxy* como o faz normalmente com outros objetos Java. A localização do objeto servidor passa a ser transparente para o programador. Todas as chamadas feitas pelo cliente serão enviadas automaticamente para o servidor através do *proxy*.

[4] http://www.iona.com/products/internet/orbixweb/index.html
[5] http://www.inprise.com/visibroker/

3.2.3 Servidor

Uma interface IDL exige uma correspondente classe Java que implementa um método para cada operação ou atributo que foi definido. Instâncias dessa classe são objetos remotos que podem ser acessados de qualquer localidade. Um cliente não precisa ter conhecimento dos detalhes de implementação dessa classe, mas apenas de sua interface. Cada objeto servidor possui um nome que é único no sistema, formado por *hostName* + *serverName* + *objectName*.

3.2.4 Repositório de Objetos

OrbixWeb provê um repositório de objetos, acessado via *daemon* `orbixd`, que gerencia os servidores no sistema. Esse repositório faz um mapeamento do nome do servidor para a classe que o implementa. O registro de um servidor no repositório permite que sua ativação seja automática. Desse modo, conforme definido por CORBA, um objeto não precisa ser executado para estar apto a receber chamadas remotas. Isso é importante porque um número muito grande de servidores podem ser disponibilizados na rede sem que recursos sejam sub-utilizados.

O repositório de objetos [Tec97b, pg. 17] possui comandos para inserir, listar e remover objetos e para alterar suas permissões de acesso. Diretórios podem ser criados para conter grupos de objetos separados.

3.2.5 Exemplo de uma Aplicação

As etapas envolvidas na utilização de objetos remotos em OrbixWeb são semelhantes ao processo empregado na programação orientada a objetos, ou seja, consiste das etapas de especificação de tipos, implementação das classes e instanciação de objetos.

Para mostrar as etapas envolvidas no processo de criação de uma aplicação em OrbixWeb será usado o exemplo de um `grid` que foi retirado do guia de programacão [Tec97a]. O exemplo consiste de um *array* bidimensional com métodos que podem ser acessados remotamente para acesso e modificação do seu conteúdo. O objetivo dessa seção é permitir uma visão geral do processo de criação de uma aplicação distribuída em CORBA; para maiores informações, os manuais do produto devem ser consultados.

Especificação da Interface

O primeiro passo para desenvolver uma aplicação em OrbixWeb é a definição da interface usando IDL. O código 3.1 mostra a definição dessa interface do grid.

```
interface Grid {
  readonly attribute short height;
  readonly attribute short width;

  void set(in short n, in short m, in long value);
  long get(in short n, in short m);
};
```

Código 3.1: Interface Grid

O nome do arquivo deve terminar com a extensão `idl`, no nosso caso `grid.idl`. Essa interface provê dois atributos: `height` e `width` que definem o tamanho do grid. O rótulo `readonly` assegura que esses atributos não poderão ser modificados diretamente pelo cliente. Estão definidas duas operações: set para permitir que um elemento seja iserido no grid e get que retorna um elemento. Os parâmetros das operações são rotulados como `in` (passados do cliente para o servidor). Outras possibilidades são os rótulos `out` (do servidor para o cliente) e `inout` (quando o parâmetro trafega nos dois sentidos).

Após a definição da interface, é feita a sua compilação através do comando:

```
$ idl grid.idl
```

A compilação resulta num conjunto de sete arquivos com código Java que vão auxiliar no desenvolvimento da aplicação servidor e da aplicação cliente. Esses arquivos e suas funções são descritos a seguir:

_gridRef : é uma interface Java cujos métodos definem a visão do cliente para a interface IDL.

Grid : uma classe Java que implementa os métodos definidos na interface.

_gridHolder : uma classe que define o tipo "Holder"para a classe Grid. É necessário para passagem de objetos Grid como parâmetros nas operações com a interface IDL.

_gridOperations : uma interface Java que mapeia os atributos e operações da interface IDL para Java. Estes métodos têm que ser implementados em uma classe pelo servidor.

_boaimpl_Grid : uma classe abstrata Java que permite que o lado servidor implemente a interface Grid usando uma de duas técnicas disponíveis em OrbixWeb; esta técnica é chamada BOAImpl e é baseada no mecanismo de herança.

_tie_Grid : uma classe Java que permite ao desenvolvedor do lado servidor a implementação da interface Grid usando uma das duas técnicas disponíveis em OrbixWeb; esta técnica é chamada TIE e é baseada no mecanismo de delegação.

_dispatcher_Grid : classe Java usada internamente por OrbixWeb. Os desenvolvedores não precisam entender os detalhes dessa classe.

Implementação da Interface

Nesta fase são implementados os métodos definidos na etapa anterior, o que é mostrado no código 3.2 na próxima página.

Para implementar a interface Grid, o programador tem que escrever uma classe Java que implementa os métodos listados na interface _GridOperations. O programador precisa indicar também que esta classe implementa a interface Grid usando a abordagem BOAImpl ou TIE. A abordagem utilizada para indicar que uma classe Java implementa a interface IDL será a TIE. Nessa técnica, o implementador não precisa ter conhecimento detalhado sobre a classe _tie_Grid. No código desse exemplo, tanto o servidor como o cliente assumem que a interface IDL foi compilada com opção para criar um pacote `gridtest`, como no comando abaixo.

```
$ idl -jP gridtest grid.idl
```

A Classe GridImplementation implementa os métodos da interface Java _GridOperations, no total são quatro métodos: um para cada uma das duas operações mais um método para cada atributo[6]. A classe GridImplementation implementa esses métodos como mostrado no código exemplo.

[6]Um método para atributos `readonly` e dois métodos para cada atributo `read/write`

```
class GridImplementation
implements _GridOperations {
  public int m_height; // stores height
  public int m_width; // store width
  public long m_a[ ][ ]; // 2D array for grid data
  public GridImplementation (int height,
                int width) {
    m_a = new long[height][width];
    m_height = (short)height; // set up height.
    m_width = (short)width; // set up width.
  }
  // read height attribute
  public short get_height() {
    return (short)m_height;
  }
  // read width attribute
  public short get_width(){
    return (short)m_width;
  }
  // set value
  public void set(short n, short m, int value) {
    m_a[n][m] = value;
  }
  // get value
  public int get(short n , short m) {
    return (short)m_a[n][m];
  }
}
```

Código 3.2: Implementação da interface Grid

Implementação do Servidor

A função do servidor, código 3.3 na página seguinte, é criar instâncias da classe que implementa a interface definida anteriormente, e depois, informar OrbixWeb quando a inicialização terminou e o servidor está pronto para receber chamadas de clientes.

Falta agora implementar o servidor que atenderá às solicitações de clientes e instanciará objetos do tipo GridImplementation. Nesse exemplo, o objeto `gridImpl` é criado com o valor inicial de 100 nas duas dimensões.

O próximo passo é fornecer o objeto `gridImpl` para o construtor da classe `_tie_Grid`. Por fim, o servidor indica a OrbixWeb que a sua inicialização foi completada. Isso é feito com a chamada ao método `impl_is_ready` com o nome que o servidor foi registrado sendo passado como parâmetro.

O código 3.3 não está completo e não pode ser compilado porque faltam tratamentos das exceções, elas foram retiradas para deixar o exemplo mais simples.

Registro do Servidor

É necessário para permitir que o *byte-code* do servidor seja buscado do repositório e executado automaticamente no momento em que ocorrer uma chamada remota de um cliente.

Depois de compilar o servidor usando o compilador `Javac`, é preciso registrá-lo no repositório para o atendimento automático de chamadas. Esse registro é feito através do comando `putit` conforme mostrado no exemplo seguinte.

```
putit GridSrv -java <class name> <class path>
```

```
import IE.Iona.Orbix2._CORBA;
import IE.Iona.Orbix2.CORBA.SystemException;

public class GridServer {
  public static void main(String args[]) {
    GridImplementation gridImpl;
    _GridRef gridTie;
    gridImpl = new GridImplementation (100,100);
    gridTie = new _tie_Grid (gridImpl);
    _CORBA.Orbix.impl_is_ready("GridSrv");
  }
}
```

Código 3.3: Implementação do servidor

Antes de registrar um servidor, é preciso assegurar que o *daemon* `orbixd` esteja rodando na máquina servidora.

Implementação do Cliente

O programa cliente que utiliza os serviços do servidor da maneira como foram disponibilizados na interface.

A aplicação cliente, mostrada no código 3.4, deve definir uma variável do tipo _GridRef. O cliente chama o método estático _bind da classe Grid para criar uma instância dessa classe que pode ser acessada através da interface Java _GridRef. Esse objeto faz o papel do *proxy* no lado cliente.

```
import IE.Iona.Orbix2._CORBA;

public class Client {
  public static void main(String args[]) {
    _GridRef gRef = null;
    String srvHost;
    short h, w;
    int v;
    srvHost = new String (args[0]);
    gRef = Grid._bind(":GridSrv", srvHost);
    w = gRef.get_width();
    h = gRef.get_height();
    System.out.println("height is " + h);
    System.out.println("width is " + w);
    gRef.set((short)2, (short)4, 123);
    v = gRef.get((short)2, (short)4);
    System.out.println
      ("value at grid position (2,4) is " + v);
  }
}
```

Código 3.4: Implementação do cliente

O método _bind cria o objeto *proxy* do tipo Grid e o liga ao objeto que implementa as funcionalidades no servidor "GridSrv". O parâmetro `srvHost` é fornecido na linha de comando durante a chamada desse cliente.

O primeiro parâmetro usado na chamada do médodo _bind é da forma `<object name>:<server name>`: onde o nome do servidor é aquele usado para registrá-lo; o nome do objeto identifica um dos objetos desse servidor. No caso desse exemplo, apenas o nome do servidor foi usado porque o servidor é composto por apenas um objeto.

O próximo passo é compilar a aplicação cliente, garantindo que o compilador terá acesso às classes da API, ao pacote OrbixWeb e às classes utilizadas pela aplicação cliente. Por último, o *byte-code* produzido pela aplicação cliente pode ser executado, conforme mostrado no comando a seguir.

```
$ java gridtest.Client <server host>
```

3.3 Voyager

Voyager é um produto desenvolvido pela empresa Objectspace e foi uma das tecnologias avaliadas como alternativa para implementação deste projeto. Voyager fornece recursos para criação de aplicações distribuídas em Java e, além dos tradicionais recursos para computação distribuída providos por um ORB, os principais recursos são o suporte para construção de agentes móveis, comunicação entre grupos de objetos distribuídos e integração com CORBA.

A principal vantagem de Voyager é a facilidade com que as aplicações distribuídas podem ser criadas. Uma aplicação convencional Java pode ser automaticamente transformada em uma aplicação distribuída e ter seus métodos acessados de qualquer localidade da rede sem necessidade de alteração no código da classe original.

Esta seção descreve os recursos de Voyager dando mais importância aos que estão relacionados com objetos distribuídos. Além destes, há também recursos para agentes móveis. A finalidade é estabelecer uma visão comparativa entre esta ferramenta, os recursos de RMI e os de OrbixWeb.

Conforme descrito com mais detalhes no guia do usuário [Obj97][7], Voyager possui propriedades descritas a seguir:

- permite que qualquer classe Java seja acessível remotamente sem que o código original tenha que ser modificado.
- objetos Java serializáveis podem ser facilmente armazenados em vários bancos de dados.
- uso da sintaxe Java na criação de objetos remotos e na chamada de métodos.
- suporte para criação de agentes móveis com localização transparente.
- integração com CORBA.

3.3.1 Agentes

Nos sistemas distribuídos tradicionais — arquitetura cliente/servidor modelo de comunicação *send/receive* — os recursos externos são utilizados através do envio e recebimento de mensagens. RPC, RMI e ORB's são exemplos dessas tecnologias. Uma outra alternativa envolve o conceito de *agentes*: objetos móveis e autônomos que podem tomar decisões enquanto migram de um ponto para outro da rede. Os agentes, ao contrário do modelo *send/receive*, consomem recursos de rede apenas para se locomover. Desse modo, tarefas que exigem comunicação intensa podem ser realizadas com grande eficiência. Agentes podem migrar-se, processar as operações localmente e depois enviar o resultado final para a máquina de origem. Essa estratégia pode diminuir consideravelmente o número de mensagens na rede e, conseqüentemente, o tempo de processamento de algumas aplicações.

Em Voyager, para mover um objeto para outra localidade basta enviar uma mensagem para sua referência remota. Sempre que um objeto se desloca, um *forwarder* é criado e deixado no seu endereço atual. A finalidade é retransmitir para o novo endereço, as mensagens enviadas para o endereço antigo do objeto.

[7] Acessível em: `http://www.objectspace.com/voyager/documentation.html`

Desse modo, o deslocamento de um objeto pela rede é transparente para quem envia a mensagem. Depois da primeira, as mensagens subsequentes, que já sabem da nova localização do agente, farão o contato diretamente sem a utilização do *forwarder* até que este seja automaticamete destruído.

Suporte para agentes é uma das grandes vantagens de Voyager, no entanto esse conceito não se relaciona diretamente com este projeto e foi citado apenas como apresentação da tecnologia. As outras funcionalidades de Voyager são descritas a seguir.

3.3.2 Programa Voyager

Todo objeto servidor Voyager possui um endereço formado pelo *hostname* seguido do número de uma porta TCP que é associada a esse objeto. Se uma aplicação será acessada por outras, então uma porta deve ser previamente escolhida e essa informação deve ser do conhecimento das aplicações clientes. As portas dos objetos clientes são escolhidas aleatoriamente.

3.3.3 Compilador Vcc

VCC (Virtual Class Creator) é um utilitário Voyager usado para habilitar classes Java para que sejam acessadas remotamente. Esse utilitário pode ser rodado sobre um arquivo .java ou .class. Portanto, é possível criar aplicações distribuídas mesmo sem acesso ao código fonte das classes. Uma nova classe denominada de classe virtual, com o mesmo nome da classe original antecedido da letra "V", será gerada pelo compilador `vcc`. Como mostrado abaixo.

```
$ vcc MyClass -- VMyClass.java
```

A classe virtual implementa as mesmas interfaces implementadas pela classe original e estende as mesmas classes que são superclasses da original. Para cada classe do usuário, será criada uma classe virtual correspondente. A classe COM.objectspace.voyager.VObject corresponde à classe java.lang.Object de Java.

O *servidor voyager* precisa ser executado no lado da aplicação que possui algum objeto acessível remotamente. Esse servidor é um programa Voyager usado para monitorar solicitações e funciona como um *daemon* para atender às chamadas remotas, esse programa permanecerá executando até que seja explicitamente finalizado.

Para cada construtor e cada método estático da classe original, o compilador `vcc` cria uma classe virtual com um método que possui um argumento além dos que já foram definidos para sua utilização local. O novo argumento é o endereço para acesso via rede, ou seja, a identificação da máquina mais o número da porta. Isso permite que as classes sejam instaciadas e acessadas remotamente.

Cada objeto remoto é identificado por 16 *bytes* aleatórios denominados de GUID. Um objeto remoto já existente pode ser referenciado pelo GUID ou, caso o programador tenha definido, por um *alias* que facilita sua identificação.

Uma referência virtual pode ser enviada ou armazenada como qualquer outro objeto Java. A obtenção de uma referência virtual é bastante simples: o método estático Vobject.forObject cujo parâmetro de entrada é uma instância da classe Object de Java, pode ser usado para obter um objeto virtual (instância de VObject) a partir de um objeto Java.

3.3.4 Integração Voyager e CORBA

O objetivo da Objectspace não é fazer de Voyager um ORB para que o usuário desenvolva aplicações CORBA, mas sim, permitir que aplicações Voyager possam se comunicar com outros padrões. A tecnologia Voyager pretende se manter como uma opção centrada em Java e ao mesmo tempo oferecer recursos para comunicação com outras tecnologias como CORBA e DCOM. A grande vantagem de Voyager é a facilidade

com que uma classe pode se tornar acessível remotamente e sem modificações no estilo de programação Java, ao contrário de uma aplicação CORBA que usa os recursos dos ORBs disponíveis atualmente.

Os principais aspectos da integração Voyager e CORBA[8] são os seguintes:

- a partir de uma interface IDL, Voyager cria automaticamente uma interface Java e uma classe virtual Voyager. O inverso também pode ser feito, uma classe java pode ser automaticamente transformada em uma interface IDL.
- Voyager pode obter um referência virtual a partir de um objeto CORBA e pode também fornecer uma referência para CORBA (IOR) a partir de um objeto Voyager. Um desenvolvedor pode usar um objeto CORBA da maneira como usa um objeto Voyager, o mapeamento das referências e o uso do protocolo IIOP é transparente. O mapeamento inverso, quando um servidor CORBA usa uma referência para um objeto Voyager também é feito automaticamente.
- as funcionalidades de Voyager como *multicast* e comunicação de grupos, podem ser usadas também entre objetos CORBA e objetos Voyager.
- uma classe Java pode se tornar CORBA compatível sem que o programador tenha que modificar explicitamente seu código.

Todos esses aspectos descritos acima são documentados no manual do usuário juntamente com exemplos que demonstram as características de como Voyager se integra com CORBA. Embora todos esses exemplos tenham sido testados e avaliados, nenhuma aplicação chegou a ser desenvolvida com essa ferramenta. Os exemplos avaliados envolvem:

- desenvolvimento de um servidor CORBA e de um cliente Voyager. O servidor, desenvolvido usando os recursos do ORB Voyager, publica sua IOR escrevendo-a em um arquivo que será acessado pelo cliente Voyager.
- desenvolvimento de um servidor Voyager e de um cliente CORBA que acessa esse servidor. O desenvolvedor Voyager escreve as interfaces em java. As interfaces IDL são geradas automaticamente e devem ser disponibilizadas para quem desenvolve um cliente CORBA. O cliente CORBA lê a referência, que foi gravada como uma string e usa o método `string_to_object` para obter a referência IOR.
- habilitação de uma classe Java para se tornar um objeto CORBA e desenvolvimento de um cliente CORBA que acessa esse objeto.
- uso de serviços avançados como *multicast* e invocação dinâmica para comunicação envolvendo objetos Voyager e objetos CORBA.

O mapeamento de um objeto Java para Voyager é bastante natural e isso constitui a grande vantagem do uso dessa tecnologia para construção de aplicações distribuídas. O mesmo não ocorre em CORBA que, justamente por se tratar de uma arquitetura comum a várias linguagens, não permite que uma aplicação seja desenvolvida dentro do estilo de programação Java. Como as especificações da OMG se aproximam mais das características da linguagem C++, muito do trabalho de adaptação para o padrão CORBA fica sob responsabilidade do desenvolvedor que utiliza um ORB para Java.

3.4 Java RMI

Outra importante tecnologia de *middleware* é *Remote Method Invocation* ou RMI. Com o objetivo de permitir a compreensão de RMI, alguns conceitos de Java serão abordados a seguir. Uma descrição mais detalhada de outros conceitos da linguagem será apresentada no capítulo 4.

[8] Documento acessível em: `http://www.objectspace.com/voyager/white_papers.asp`

3.4.1 Introdução a Java

Java é uma linguagem de programação derivada de C++ [Str97] que surgiu em 1990 dentro da Sun Microsystems[9] como resposta ao interesse por um ambiente de programação mais robusto, enxuto, confiável e principalmente independente de arquitetura. O projeto inicial tinha o objetivo de construir uma linguagem para ser utilizada no desenvolvimento de sistemas para aparelhos eletrodomésticos [Fla97]. As propriedades mais exigidas para esse tipo de aplicação eram portabilidade, confiabilidade e robustez.

Alta portabilidade era desejável para facilitar a migração de programas de uma arquitetura para outra à medida que surgissem novos *chips* mais velozes. Confiabilidade e robustez foram consideradas propriedades importantes para permitir o desenvolvimento de código seguro e para garantir a funcionalidade dos equipamentos que seriam espalhados rapidamente para milhares de usuários. Para que o compilador não se tornasse muito complexo ou pouco eficiente, a linguagem deveria ser bastante compacta.

O que permite portabilidade para qualquer arquitetura é um mecanismo de código intermediário gerado após a compilação do código fonte Java: ao invés de gerar código de máquina, o compilador gera *byte-codes* que serão interpretados pela máquina virtual Java JVM (*Java Virtual Machine*) [GJS96], e esta sim, é específica para cada plataforma. A desvantagem desse mecanismo é que um código interpretado não é tão eficiente quanto um código executável compilado especificamente para uma determinada arquitetura. No entanto, um recurso adicional pode ser usado para que o código Java seja compilado para gerar código de máquina ao invés de *byte-code*. Com isso, perde-se a independência de máquina mas ganha-se em eficiência. É possível ainda, compilar o *byte-code* no momento de sua carga para execução usando o processo denominado JIT (*just-in-time-compiler*) para combinar desempenho e independência de máquina.

Como Java é uma linguagem independente de plataforma, um programa desenvolvido em Java pode ser executado em qualquer arquitetura ou sistema operacional. As particularidades de cada plataforma são ocultadas pela máquina virtual Java. Um programa Java é compilado para gerar *byte-codes* que são iguais independente de qual plataforma foi usada para sua geração. A execução desse código intermediário é feita pela JVM e esta sim, é específica para cada arquitetura ou sistema operacional.

Java permite o uso de alguns recursos extras que aumentam o poder da linguagem mas podem deixar o programa dependente de uma plataforma específica. Um exemplo desses recursos é JNI (*Java Native Interface*) que permite utilização de funcionalidades desenvolvidas em outras linguagens. O programador muitas vezes precisa decidir entre usar um recurso desses ou manter o programa totalmente portável.

Java é uma linguagem orientada a objetos com muitos recursos importantes para desenvolvimento de aplicações distribuídas para a Internet. Aplicações podem ser desenvolvidas uma vez e executadas em diferentes plataformas. A independência de plataforma é o grande trunfo da linguagem, tanto que "*write once, run anywhere*" é o principal *slogan* usado pela Sun Microsystems para a divulgação de Java. Embora Java tenha se popularizado com os *applets* [CH96, cap. 8]; como linguagem de programação, Java já seria uma ótima opção mesmo que o aspecto de independência de plataforma não fosse considerado. Isso porque Java está atualizada com avanços alcançados na área de programação orientada a objetos além de acumular toda a experiência de longos anos de estudos e exaustivo uso da linguagem C++. Java consegue selecionar o que é realmente importante em uma linguagem de uso geral e é considerada uma linguagem poderosa e relativamente simples de usar.

3.4.2 Invocação Remota de Métodos

Inicialmente foi desenvolvida para ser usada entre duas aplicações Java. No entanto, o uso dessa tecnologia vem sendo bastante difundido e há esforços no sentido de integrar RMI ao padrão CORBA.

As tecnologias RPC e CORBA precisam lidar com ambientes heterogêneos enquanto que RMI assume um ambiente homogêneo: a máquina virtual Java (JVM), para qual os códigos são gerados. Em RMI, a inde-

[9] `http://java.sun.com/docs/overviews/java/java-overview-1.html`

pendência de plataforma é centrada na linguagem enquanto que nas demais tecnologias, essa funcionalidade é implementada em uma camada de suporte.

Em RMI não há independência de linguagem, a limitação é que todas as aplicações devem ser desenvolvidas em Java. CORBA permite independência de linguagem através da adoção de uma linguagem intermediária (IDL – *Interface Definition Language*), isso tem algumas conseqüências negativas que são discutidas a seguir:

- os argumentos que podem ser facilmente transmitidos entre as aplicações estão limitados aos tipos suportados por todas as linguagens, ou seja, os que estão definidos na interface IDL.
- os objetos não podem ser passados por valor, apenas por referência; não faz sentido mover o código de uma linguagem para outra.
- os parâmetros precisam ser passados exatamente como foram definidos na IDL, isto é, não podem ser modificados em tempo de execução de um lado e entendidos do outro lado.
- o programador precisa lidar com duas linguagens, a IDL mais a que é usada para a implementação de sua aplicação. Embora a linguagem para definição de interface seja simples, o mapeamento de tipos da IDL para a linguagem que o programador utiliza é, em alguns casos, complexo e o programador precisa entender como esse mapeamento é feito.
- para desenvolver uma aplicação CORBA compatível, o programador precisa utilizar um ORB, geralmente um produto independente da linguagem. Isso torna difícil manter um bom estilo de programação além de deixar o processo menos intuitivo e mais complexo.

Em RMI não existe independência de linguagem, o que inegavelmente é uma limitação já que há casos em que o desenvolvedor não pode garantir que sua aplicação será acessada por outras aplicações que também foram implementadas em Java. No entanto, as vantagens são muitas caso o desenvolvedor possa optar pelo uso de Java nos servidores e nos clientes. Outra opção é aguardar pela integração RMI-CORBA e ver como essas questões serão resolvidas. As principais vantagens do uso de RMI são listadas a seguir:

- visão de um sistema homogêneo e centrado na linguagem; não há outros ambientes com os quais o programador precisa interagir para construir aplicações distribuídas. Uma aplicação desenvolvida em RMI não é estruturalmente diferente de outras aplicações Java.
- as aplicações distribuídas podem contar com as funcionalidades disponíveis na linguagem como gerenciador de segurança e coletor automático de lixo.
- os argumentos podem ser passados em seu tipo previamente definido ou no de sua superclasse, ou seja, é implementado polimorfismo distribuído.
- objetos podem ser passados por valor: é relativamente simples mover código entre duas máquinas virtuais Java.
- todos os tipos válidos para as aplicações locais são também válidos para as aplicações distribuídas.

3.5 Considerações Finais

A escolha por uma tecnologia para desenvolver uma aplicação distribuída não é uma tarefa trivial. Muitos são os artigos comparativos, no entanto a grande maioria deles aborda questões estruturais de como essas tecnologias foram desenvolvidas. Essas discussões não ajudam muito quando a pergunta é sobre a facilidade

de utilização de uma ou de outra ferramenta. O principal problema é que a maioria das limitações de cada tecnologia somente serão percebidas durante sua utilização.

Durante a fase de estudos, um recurso muito útil foram os grupos de discussões (*newsgroups*) da Internet. Os problemas mais freqüentemente encontrados pelos usuários de CORBA podem ser acompanhados no grupo `comp.object.corba` e de RMI no grupo `comp.lang.java.programmer`. Muitos dos possíveis problemas de um sistema podem ser previstos ainda na fase de projeto através de um estudo dos problemas e sugestões relatados nesses grupos. Para Voyager, as questões são tratadas através de lista de discussão acessível por E-mail ou a partir da página da ObjectSpace.

Voyager foi uma opção tentadora, principalmente pela facilidade de utilização e porque mantém o estilo de programação Java. No entando, a versão disponível naquele momento era apenas experimental e alguns aspectos relevantes não puderam ser avaliados. Ao contrário das outras tecnologias que já estavam mais amadurecidas pelo desenvolvimento de várias aplicações de médio e grande porte, Voyager vinha sendo muita usada apenas em pequenos projetos.

Mesmo tendo sido feito um trabalho anterior de avaliação dessas ferramentas, a implementação deste projeto passou por algumas mudanças com relação à tecnologia que seria utilizada. A proposta inicial deste projeto previa o uso de CORBA, passou por uma experiência usando *sockets* para depois ser desenvolvida em RMI. Em paralelo foi avaliado Voyager, muito importante por fornecer um nível de abstração mais elevado, no entando, não chegou a ser usada porque algumas funcionalidades deste projeto precisam acessar recursos num nível mais baixo que os permitidos por essa tecnologia.

A opção de implementar em CORBA foi abandonada porque a maior dificuldade seria a manipulação dos dados apenas nos tipos suportados pela interface IDL, essa limitação seria refletida para o usuário da LegoShell. A vantagem é que o sistema seria CORBA-compatível assim como as aplicações desenvolvidas sobre ele. No entanto, a passagem dos parâmetros só poderia ser por referência. Outro ponto considerado negativo foi o mapeamento de tipos da linguagem para IDL que muitas vezes é uma tarefa que exige um esforço extra do programador.

Antes de se optar pelo uso de RMI, foi implementado um conjunto de classes para oferecer um mecanismo de *pipe* que possibilita a comunicação de aplicações Java. Esse trabalho foi implemetado para funcionar como as classes PipedInputStream e PipedOutputStream de Java, com a vantagem que estas classes só funcionam para comunicação entre *threads* e as que foram desenvolvidas permitem interligar duas aplicações localizadas em máquinas distintas. Com a implementação desse mecanismo, pretendia-se utilizá-lo na implementação deste sistema. Novamente, o projeto foi reavaliado e percebeu-se que RMI poderia ser mais adequado. Uma das vantagens de RMI para o projeto é o repositório de nomes `rmiregistry` que, mesmo sem as características desejáveis, poderia ser melhorado e utilizado ao invés de implementar um novo servidor de nomes.

A opção de usar *sockets* começou a ser utilizada porque permite a manipulação de dados em um nível mais baixo. No entanto, seria necessária a implementação de algum mecanismo de servidor de nomes, semelhante ao `rmiregistry` existente em RMI. A partir desse momento ficou resolvido que RMI seria a melhor alternativa e a implementação foi recomeçada usando esta tecnologia.

Capítulo 4

Java

A linguagem Java possui um grande conjunto de funcionalidades e muitas delas não estão diretamente relacionadas com este trabalho. O objetivo deste capítulo é apresentar uma visão geral dos recursos que foram usados na implementação da LegoShell, os conceitos básicos da linguagem são descritos na seção 3.4.1; além das referências citadas, as principais funcionalidades, incluindo as mais recentes, podem ser consultada na página *Java White Papers*[1] onde são encontradas desde descrições resumidas da linguagem até coleções de artigos sobre funcionalidades específicas.

4.1 Conceito de Interface

Ao contrário de C++ Java não possui herança múltipla. A principal razão para isso é que o uso de herança múltipla tem alguns inconvenientes, um dos mais relevantes é o conflito de atributos: o compilador precisa decidir qual atributo é válido quando este aparecer definido em duas superclasses. Qualquer que seja a estratégia utilizada para solucionar esse conflito, haverá questões com as quais o programador terá que estar atento na hora de definir a estrutura de classes de seu programa. Outra razão para a não implementação de herança múltipla é que o sistema de execução fica mais simplificado.

A utilização do conceito de interface Java resolve a maioria dos problemas onde se usaria herança múltipla. Além disso, interface pode ser usada para a criação de tipos mais genéricos. Uma interface é semelhante a uma classe, a diferença é que a primeira não contém implementações dos métodos mas apenas as suas assinaturas que são declarações dos cabeçalhos. Uma interface não pode ser instanciada e, ao contrário de uma classe, pode estender várias outras interfaces, portanto existe herança múltipla para interfaces.

Interface é, essencialmente, a garantia de que uma classe assume o compromisso de implementar certos métodos. Para se utilizar o conceito de interface, existem duas etapas: definição da interface e implementação dessa interface por outra classe que deve codificar os métodos que foram definidos anteriormente.

No código 4.1 é apresentado um exemplo de definição de uma interface.

Qualquer classe que implemente a interface Executable precisa implementar os métodos exec e stop, caso contrário ocorrerá erro de compilação. A declaração do compromisso de implementar uma interface é feita através do uso da palavra reservada `implements` no cabeçalho de definição da classe, como mostrado no código 4.2.

As instâncias da classe App são também do tipo *executable* e essa flexibilidade de tipos é importante para a implementação de classes mais genéricas.

[1] `http://java.sun.com/docs/white/index.html`

```
/*
Exemplo de definiçãoo de uma interface
*/
public interface Executable {
  public void exec();
  public void stop();
}
```

Código 4.1: Interface Executable

```
/*
Exemplo de uma classe que implementa uma interface
*/
public class App implements Executable {

  public void exec(){
    //código
  }
  public void stop(){
    //código
  }
}
```

Código 4.2: Classe App

4.2 Exceções

Uma linguagem deve prover meios para tratamento de erros que podem ocorrer durante a execução de um programa. Em Java, esse recurso é suportado por um mecanismo de exceções.

"Uma *exceção* é um evento que ocorre durante a execução de um programa que teve o fluxo normal de instruções interrompido." [CW98].

O fluxo de execução é determinado pela chamada de métodos. Sempre que um método não for finalizado normalmente, uma situação excepcional terá ocorrido e Java aciona um manipulador de exceções. Como muitos métodos executam operações bastante simples, nem todos eles devem levantar exceções. Os métodos que podem gerar exceções declaram isso em seus cabeçalhos e a chamada a um desses métodos exige o tratamento dessa exceção ou a sua propagação. Desse modo, o programador não pode usar um método desses sem tratar explicitamente as exceções.

Durante a execução, quando ocorre um erro dentro do código de um método, uma exceção é criada — um objeto daquela classe é instanciado — e sua execução é disparada. Os métodos pendentes, aqueles que aguardam pelo retorno deste, são percorridos de volta em busca de um tratador para o tipo da exceção que ocorreu. Caso nenhum tratador seja encontrado, a execução pode ser abortada abruptamente.

Um método que pode gerar uma exceção deve declarar isso em seu cabeçalho. Um código não será compilado caso uma chamada de método não trate ou redirecione uma exceção definida para este método.

No código 4.3, o método readByte pode gerar uma exceção de final de arquivo denominada EOFException. Esta exceção já faz parte do conjunto de classes originais de Java, no entanto o usuário poderia definir e utilizar suas próprias exceções.

O uso do método readByte, como mostrado no codigo 4.4, só poderá ser feito caso a exceção EOFException seja tratada ou propagada. No método read1, a exceção é simplesmente propagada e caso não seja tratada em nenhum instante, uma mensagem de erro será enviada podendo interromper o fluxo de execução do programa. No método read2, a mesma exceção é capturada e tratada.

É importante lembrar que o mecanismo de exceções não exime o programador das obrigações de prever e gerenciar os erros que podem ocorrer durante a execução de seu código, apenas permite um tratamento

```
/*
Exemplo de levantamento de exceção
*/
public class ReadFile {
  public byte readByte() throws EOFException {
    ...
    if (finalDeArquivo) {
      throw new EOFException();
    }
    //retorna byte lido
  }
}
```

Código 4.3: Geração de uma exceção

```
/*
Exemplo de tratamento de exceção
*/
public class ExceptionTest {
  public byte read1() throws EOFException {
    //chamada ao método readByte()
  }
  public byte read2() {
    try{
      //chamada ao método readByte()
    }
    catch (EOFException e){
      //tratamento para a exceção
    }
  }
}
```

Código 4.4: Tratamento de exceções

limpo e padronizado para tais situações. As principais vantagens do uso de exceções em Java, como detalhado em [CW98], são:

- separar o código regular do código tratador de erros
- propagar erros pela pilha de métodos pendentes
- permitir agrupamento de erros em classes

Em resumo, métodos Java declaram exceções que precisam ser tratadas ou propagadas quando esses métodos são chamados. Conforme [CH96, pag. 434], deve-se tratar uma exceção sempre que sua manipulação for possível; caso não se saiba o que fazer, então a exceção deve ser propagada. O desenvolvedor deve classificar os possíveis erros de sua aplicação, criar classes derivadas de Exception para representar esses erros e levantar exceções quando elas ocorrerem.

4.3 Programação Concorrente

Convencionalmente, um programa segue um único fluxo de execução, isto é, em qualquer instante do seu ciclo de vida, haverá um único ponto sendo executado. No entanto, mesmo em plataformas com apenas um processador, permitir que operações possam ser executadas concorrentemente representa uma série de

vantagens ao desenvolvimento de aplicações [OW97]. Java permite a criação de vários fluxos de execução concorrentes através do uso de *threads*.

4.3.1 *Threads*

"*Thread* é um fluxo de execução com controle seqüencial e único dentro de um programa"[CW98]. Java provê suporte para criação de múltiplas *threads* de maneira bastante simples. *thread* pode ser entendida como um processo leve[2] porque as operações de troca de contexto são mais simples, o que ameniza o *overhead* para a execução de diferentes tarefas, conforme [Lea96].

Antes da tecnologia de *threads*, o recurso mais usado em programação concorrente era a criação de vários processos. No entanto, essa abordagem implica em um considerável *overhead* devido principalmente às trocas de contextos necessárias sempre que a CPU alterna entre a execução de dois desses processos. Muitas vezes esse *overhead* é tanto que as vantagens do uso de programação concorrente se tornam questionáveis. *Threads*, ao contrário, podem ser chaveadas com menor custo porque a troca de contexto é mais simples e porque procuram utilizar os recursos já disponíveis para o contexto atual a que pertencem. Resumindo, o uso de *threads* torna a programação concorrente mais fácil e menos dispendiosa.

Java possui, integrado aos recursos da linguagem, suporte para utilização de *threads*. A criação de *threads* é bastante simples e pode ser feita de duas maneiras.

Derivando a classe Thread : é a maneira mais imediata, basta uma classe estender a classe Thread e implementar o método run. O código 4.5 mostra como isso é feito. !!

```
public class SimpleThread1 extends Thread {

  public void run() {
    //implementação da funcionalidade
  }
}
```

Código 4.5: *Thread* derivando a classe Thread

Implementando a interface Runnable : caso uma classe já seja derivada de outra, não poderá usar o modelo anterior porque não há herança múltipla em Java. A solução é implementar a interface Runnable como mostrado no código 4.6.

```
public class SimpleThread2 implements Runnable {

  public void run() {
    // implementação da funcionalidade
  }
}
```

Código 4.6: *Thread* implementando a interface Runnable

Derivando a classe Thread ou implementando a interface Runnable a funcionalidade da *thread* deve ser implementada dentro do método run que será automaticamente disparado quando o método start for executado por uma instância de Thread.

Para disparar uma *thread* , primeiro é necessário criá-la e depois executar o método start, como mostrado no código 4.7. !

[2]tradução de *lightweitht process*

```
class TestThread {
  public static void main(String[] args ) {
    SimpleThread1 t1 = new SimpleThread1();
    t1.start();
    SimpleThread2 t2 = new SimpleThread2();
    new Thread(t2).start();
  }
}
```

Código 4.7: Execução de *Threads*

Uma *thread* pode se encontrar em um dos seguintes estados:

Inicial : após ser instanciada com o método new e antes de ser executada pela chamada do método start.

Executável : depois de executado o método start, a *thread* entra neste estado. Uma *thread* que se encontra em execução também está neste estado.

Bloqueado : quando um dos métodos: sleep, suspend ou wait é executado ou quando a *thread* bloqueia em uma operação de entrada/saída.

Finalizada : uma *thread* termina sua execução normalmente ou através do método stop.

Em [CH96, pag. 516], há uma discussão mais detalhada sobre esses estados e como ocorrem as transições entre eles.

As *threads* podem ser criadas usando uma escala de prioridade de um total de dez níveis. Uma *thread* com maior prioridade executa antes e, nesse caso, o escalonador é preemptivo. Para *threads* de mesma prioridade, o escalonador é não-preemptivo, logo uma *thread* não perderá o uso da CPU para outra *thread* de mesma prioridade.

4.3.2 Sincronização

Como *threads* podem ser executadas concorrentemente, é necessário algum recurso que permita resolver o problema da *região crítica* — uma seção do código cuja execução simultânea por mais de uma *thread* é indesejável. Esse problema persiste mesmo em sistemas com apenas um processador onde duas instruções não são executadas ao mesmo tempo. Nesse caso, uma situação de inconsistência poderia ocorrer quando a execução de uma *thread* for interrompida dentro da região crítica e uma outra *thread* escalonada executar essa mesma região.

Em Java, um método ou um segmento de código pode ser declarado como `synchronized`, isso permite garantir que duas *threads* de um mesmo objeto não vão acessar esta região "sincronizada" ao mesmo tempo. A cada objeto que possui código sincronizado, Java associa um *lock* cuja obtenção e liberação é feita automática e atomicamente.

Particularmente, o problema da região crítica só pode ocorrer quando duas ou mais *threads* compartilham recursos. Esses recursos podem ser qualquer mecanismo de entrada e saída ou estruturas de dados.

4.3.3 Monitores

Além da sincronização, outros recursos são desejáveis para o controle de concorrência. Java implementa recursos de *monitores* através dos métodos wait e notify. O primeiro deve ser chamado dentro de uma *thread* que precisa esperar pela execução de outras. A execução desse método leva a *thread* do estado de *executável* para *bloqueado* e assim permanecerá até receber uma notificação de outra *thread*. O método notify é acionado

por uma *thread* sempre que for possível colocar outras *threads*, que executaram wait e se encontram no estado *bloqueado*, de volta ao estado *executável*.

Em resumo, uma *thread* que está sendo executada pode aguardar por uma notificação enviada por outra *thread* . Estes conceitos de monitores, conforme [OW97], são importantes também para evitar situações de *starvation* e *deadlock*.

4.4 Mecanismo de Entrada e Saída

Em Java, as operações de entrada e saída são padronizadas. Não importam os dados envolvidos — um *byte* sendo escrito em um arquivo ou um objeto em um *socket* —, o processo pode ser resumido em obter uma *stream* compatível com o recurso que será utilizado e executar o método de escrita.

Uma aplicação escreve dados em um objeto da classe OutputStream e lê de um objeto da classe InputStream. O objetivo desse padrão é permitir que as operações de entrada e saída sejam tratadas de maneira singular. Do ponto de vista do programador, não representa maior ou menor esforço a tarefa de escrever no console, em um arquivo do sistema, em um *socket* ou se a comunicação envolve duas *threads*. Também não exige maior esforço do programador se a operação manipula um *byte*, um tipo básico ou um objeto serializável, seção 4.5.

As classes InputStream e OutputStream são definidas como *abstratas*, portanto objetos não podem ser instanciados diretamente a partir dessas duas classes. Elas existem apenas para prover as funcionalidades essenciais às operações de entrada e saída. Todas as classes que o programador utiliza nas operações de entrada são derivadas direta ou indiretamente da classe InputStream e, de modo complementar, todas as operações de escrita são realizadas sobre uma instância de uma classe derivada de OutputStream.

Várias classes são derivadas dessas duas classes abstratas, o conjunto completo está esquematizado e explicado em [CW98]. Algumas das classes mais usadas serão brevemente descritas a seguir:

File streams : classes FileInputStream e FileOutputStream permitem que o programador realize operações sobre o sistema de arquivos. O nome de um arquivo é passado como parâmetro para o construtor da classe e a partir daí o conteúdo pode ser manipulado no nível de *bytes*.

Data streams : DataInputStream e DataOutputStream são construídas a partir de qualquer *stream* e permitem operações de leitura ou escrita dos tipos primitivos da linguagem Java.

Object streams : duas classes ObjectInputStream e ObjectOutputStream permitem escrever e ler objetos. Essas operações usam serialização de objetos, tema da seção 4.5.

Print streams : criadas a partir de uma *stream* de saída, permitem imprimir tipos primitivos e objetos.

Socket streams : permitem a utilização do mecanismo de *sockets* TCP [Com92], para comunicação pela Internet.

Piped streams : as classes PipedInputStream e PipedOutputStream permitem que *threads* troquem dados. A comunicação entre *threads* é bastante simples, basta instanciar uma dessas *streams* e usá-la como parâmetro durante a instanciação da outra *stream* correspondente, como mostrado no código 4.8. Essas duas *streams* devem ser complementares — uma para escrita e outra para leitura — e depois disso estarão interconectadas.

O mecanismo de *streams* padroniza as operações de entrada e saída em Java e *threads* podem se comunicar usando *PipedStreams*, conforme já foi mencionado. No entanto, a criação de *threads* está restrita ao espaço de endereçamento local e dentro de um mesmo processo. Seria muito útil um mecanismo semelhante ao de

```
public class PipedStreamTest {
  // ...
  public void test(){
    PipedInputStream pis = new PipedInputStream();
    PipedOutputStream pos = new PipedOutputStream(pis);
    ReaderThread rt = new ReaderThread(pis);
    WriterThread wt = new WriterThread(pos);
  }
  // ...
}
```

Código 4.8: Conexão entre duas *streams* para comunicação de *threads*

streams que permitisse a comunicação entre processos. Desse modo, aplicações poderiam trocar dados com a mesma facilidade com que o fazem as *threads*.

Atualmente, duas aplicações Java podem se comunicar através de *sockets* ou RMI, conforme descrito na seção 4.6. No entanto, nenhuma dessas alternativas pode ser comparada ao mecanismo de *streams* no que diz respeito à simplicidade de utilização e ao nível de abstração permitido. Com base nessa observação, este projeto teve início com o desenvolvimento de um mecanismo que permitisse a comunicação entre aplicações com a mesma facilidade provida pelo mecanismo de *streams* de Java.

A extensão de *streams* para uso entre aplicações foi então implementada em duas classes: SocketInputStream e SocketOutputStream. Usando esses recursos, duas aplicações remotas podem se comunicar de maneira tão simples como o fazem duas *threads*. As *SocketStreams* foram desenvolvidas com o objetivo de funcionar como suporte para o implementação deste sistema. Posteriormente, conforme justificado no capítulo 3, optou-se pela utilização de Java RMI e a extensão de *streams*, embora já em funcionamento, deixou de fazer parte deste projeto e passou a ser tema de um artigo independente da tese e que está em fase de preparação.

4.5 Serialização de Objetos

Freqüentemente, os dados de uma aplicação precisam ser armazenados em disco ou transportados para outras máquinas remotas via rede. O mecanismo de *serialização* é usado para transformar dados em cadeias de caracteres e permitir que essas operações sejam realizadas.

Em Java, os tipos primitivos podem ser facilmente serializados e desserializados já que a representação desses dados é bastante simples. Com objetos no entanto, a serialização envolve outras questões mais complexas como por exemplo, a serialização de referências sem a replicação dos objetos (seção 4.5.1), questões de segurança (seção 4.5.2) além de algumas limitações existentes e que o programador deve tomar conhecimento (seção 4.5.3).

O formato usado para serialização pode ser analisado através de *dump* dos objetos serializados. Embora dificilmente o programador terá que se preocupar com esse nível de detalhe, em [CH96] são encontrados alguns exemplos que ajudam a entender como alguns objetos são formatados para serialização. Resumidamente, um objeto serializado contém a identificação da classe seguido de um *hash-code* e dos atributos. A função do *hash-code* é descrita a seguir.

Se um objeto é gravado em disco, ele deverá ser restaurado para executar na mesma versão da classe que o serializou. Caso alterações sejam feitas no código e o objeto gravado para a versão anterior seja restaurado, haverá inconsistências. Para evitar isso, cada objeto é gravado junto com um código de 20 *bytes*, gerado a partir do algoritmo SHA (*Secure Hash Algorithm*) [Sta95] que permite identificar modificações no código ou nos dados de um objeto.

Em Java, serializar objetos é bastante simples, basta que a classe implemente a interface Serializable e

os métodos writeObject e readObject das classes ObjectOutputStream e ObjectInputStream sejam usados para escrever e ler os objetos. Há vários casos em que o desenvolvedor de aplicações distribuídas precisa saber como isso é feito e quais as implicações envolvidas no processo de serialização. Algumas vezes a serialização é indesejável e em outros casos pode ser feita com algumas restrições.

4.5.1 Serialização de Referências

O processo de serialização de objetos consiste em primeiro salvar o tipo de um objeto e depois salvar os seus dados. Na operação inversa ("desserialização"), a classe desse objeto é lida, um objeto desse tipo é criado e, finalmente os atributos são preenchidos com o os dados que foram restaurados. Como um atributo pode ser um objeto, esse processo tem que ser recursivo.

Um objeto é acessível através de sua referência — na realidade um endereço de memória. Não faz sentido enviar para outra máquina uma referência cujo conteúdo numérico representa o endereço local de um objeto. Essa mesma informação de nada adiantaria ser armazenada e restaurada posteriormente já que na próxima vez que a aplicação for carregada, a área de memória ocupada não será mais a mesma. Uma referência apenas faz sentido no contexto local e apenas para a execução atual.

Como as referências existem para evitar replicações e para facilitar a manipulação dos objetos, o ideal é reproduzir essa mesma estrutura depois da serialização. Um objeto não deve ser replicado porque isso facilmente levaria a inconsistências além da dificuldade adicional de manutenção de cópias. A solução para evitar replicações é a seguinte: uma vez que um objeto foi serializado, as demais referências para o mesmo objeto passarão a utilizar o que já foi criado. Isto é, cada objeto deve ser copiado uma única vez para evitar que o processo de desserialização construa um modelo diferente do original. As etapas do processo usado para evitar a replicação de objetos são apresentadas a seguir:

- todo objeto gravado em disco recebe um número serial
- antes de gravar um objeto, é verificado se este já está armazenado
- caso já exista, apenas o seu número de série é armazenado, senão todo o objeto é gravado no disco

Esse mêtodo permite gravar ou transmitir um objeto e depois reconstruí-lo sem modificar sua estrutura original.

4.5.2 Segurança

Um objeto serializado e armazenado em disco pode ser facilmente modificado seja de maneira acidental ou maliciosa. Quando uma *stream* é usada para ler um objeto, ela não faz qualquer restrição a respeito dos dados que estão sendo lidos. Por exemplo, se uma classe testa os dados antes de aceitá-los como válidos, o mesmo não é feito no processo de restauração desse objeto serializado.

Na atual versão de Java, uma classe precisa implementar a classe Serializable ou a classe Externalizable para que suas instâncias possam ser serializadas. Portanto, por *default* as classes criadas pelo usuário não são serializáveis. Para inserir restrições às classes serializáveis, o método writeObject deve ser redefinido. Outra maneira seria implementar um mecanismo de criptografia, o que pode ser feito através da implementação da interface Externalizable que permite até a criação de um novo formato de serialização. Muitas vezes, essa interface também é implementada com o objetivo de prover maior eficiência ao processo de serialização.

4.5.3 Limitações

Há vários casos em que um objeto não pode ser serializado diretamente. Por exemplo, um descritor de arquivo só pode ser reconhecido no contexto local. Uma *thread* também só funciona no contexto local de

endereçamento. Um identificador de janelas é mais um exemplo onde a desserialização não faria sentido já que esta informação só é relevante para o manipulador de janelas local. Outro exemplo é um atributo que armazena o tamanho de uma janela. Esse valor depende do tamanho da fonte, e esta é dependente da plataforma utilizada. Neste caso, talvez o mais recomendável seja calcular o tamanho de uma janela em cada localidade ao invés de serializar e transmitir esse dado. Os dados que não devem ser serializados são definidos como transientes usando a palavra chave *transient* de Java. Esses atributos simplesmente são ignorados pelo método writeObject e o programador deve prover tratamento para esses casos.

Nos casos descritos acima e em muitos outros, a serialização de um objeto precisa receber um tratamento mais elaborado. Há situações em que o programador vai precisar criar métodos para formatar os dados e métodos para restaurar a informação e reconstruir o objeto.

Basicamente, os problemas de serialização concentram as situações onde há diferenças entre execução local e execução remota de um objeto. Sempre que um objeto contiver informações que são dependentes de plataforma ou informações que são relevantes apenas no contexto local, o programador terá que fazer o tratamento de baixo nível. Se o usuário não implementar um tratamento específico para serializar um desses objetos, a classe será compilada sem problemas, os erros somente serão percebidos durante a execução.

4.6 RMI

Invocação remota de métodos, ou *RMI* (*Remote Method Invocation*), é um recurso da linguagem Java que facilita a troca de mensagens entre objetos distribuídos. O programador não precisa se preocupar com as operações de baixo nível da camada de comunicação nem com a serialização e desserialização de objetos que são passados como parâmetros ou retornados como resultado da execução de um método.

Uma aplicação Java é um conjunto de objetos que interagem trocando mensagens, isto é, executando métodos. No modelo convencional de programação onde todos os objetos pertencem ao mesmo espaço de endereçamento, uma chamada de método envolve recursos bem menos elaborados se compararmos com os que são necessários nas chamadas de métodos remotos.

As classes Socket e ServerSocket de Java permitem comunicação pela rede e podem ser usados os mecanismos de *streams* de Java, inclusive para transporte de objetos. No entanto, é importante um mecanismo mais elaborado como RMI para que métodos sejam chamados diretamente. RMI provê serialização automática de argumentos, transporte de objetos, além de outras facilidades como repositório de nomes e gerenciador de segurança.

Essa tecnologia utiliza os conceitos de cliente/servidor onde o servidor é um objeto que usa os recursos RMI para disponibilizar acesso remoto a seus métodos. Um cliente é outro objeto que acessa esses métodos e, para tanto, também faz uso dos recursos de baixo nível providos pelo protocolo RMI.

RMI é um recurso similar ao tradicional sistema RPC (*Remote Procedure Call*), o recurso mais usado para programação de aplicações distribuídas não orientada a objetos. Ao contrário de RPC, onde o transporte de objetos requer sua decomposição em tipos primitivos, RMI permite a transmissão de objetos pela rede sem necessidade de código adicional.

Embora inteiramente incorporado à linguagem, o uso de RMI não é uma tarefa simples ou imediata. A compilação do primeiro programa costuma consumir algum tempo. Uma das dificuldades surge em decorrência de algumas modificações implementadas a partir da versão JDK 1.1 e que, por ser relativamente recentes, não são documentadas na maioria dos livros atualmente disponíveis. Outra dificuldade enfrentada pelo programador iniciante em RMI está relacionada com algumas novidades próprias da tecnologia e que precisam ser manipuladas como por exemplo o repositório de nomes, o compilador RMI, o gerenciador de segurança, além dos tradicionais conceitos de programação cliente/servidor.

Um exemplo bastante simples foi desenvolvido com o objetivo de auxiliar no entendimento das etapas de utilização de RMI. Esse exemplo consiste de uma aplicação cliente que contacta um objeto remoto e

executa um método passando um objeto como parâmetro e recebendo também um objeto como retorno dessa chamada. O objeto é uma String que o cliente imprime na tela.

A seguir são descritas as etapas de utilização de RMI acompanhadas das três classes que compõem o exemplo — a interface RmiTestRef, a classe servidora RmiTest e o cliente implementado na classe RmiTestClient.

Criação da interface : o primeiro passo é a definição de uma interface Java derivada da interface Remote que está definida no pacote `java.rmi`. Os métodos remotos definidos nessa interface precisam tratar ou propagar a exceção RemoteException. Essa interface é mostrada no código 4.9 e deve estar disponível tanto no lado servidor quanto no lado do cliente.

```
import java.rmi.*;

public interface RmiTestRef extends Remote {
  public String echo(String s) throws RemoteException;
}
```

Código 4.9: Interface de uma aplicação remota

Implementação do servidor : a interface definida no passo anterior deve ser implementada em uma classe derivada de UnicastRemoteObject conforme mostrado no código 4.10. A chamada da máquina virtual Java sobre essa classe — execução do método estático main — instala o gerenciador de segurança e instância um objeto da própria classe. Esse objeto será registrado no repositório de nomes pela execução do método bind. A instalação de um gerenciador de segurança é necessária tanto no lado cliente como no lado servidor.

```
import java.rmi.*;
import java.rmi.server.*;

public class RmiTest extends UnicastRemoteObject
                     implements RmiTestRef {


  public RmiTest () throws RemoteException {
    try {
      Naming.bind ("rmitest", this);
    }
    catch (Exception e){
      e.printStackTrace();
    }
  }

  public String echo(String str){
    return("Echo:_" + str);
  }

  public static void main(String[] args) throws RemoteException{
    System.setSecurityManager(new RMISecurityManager());
    RmiTest rt = new RmiTest();
  }
}
```

Código 4.10: Classe servidora

Geração de *stubs* e *skeletons* : são duas classes geradas automaticamente pelo compilador `rmic` (*RMI Compiler*) a partir do *byte-code* da classe que foi desenvolvida no passo anterior. Essas duas classes desempenham as funções de serialização e desserialização dos argumentos, além de fazer a ponte de comunicação entre cliente e servidor. O cliente se comunica com o *stub* e o servidor se comunica com o *skeleton*; *stub* e *skeleton* se comunicam via rede.

Criação da aplicação cliente : no lado cliente, que já possui o *stub* e a interface remota, uma aplicação deve ser implementada para contactar o repositório de nomes remoto em busca do servidor ali registrado. Essa busca é feita através da chamada ao método estático lookup da classe Naming passando como parâmetro o endereço do servidor — uma URL formada pelo *hostname* mais o nome do objeto. Conforme mostrado no código 4.11, antes dessa operação de busca, a exemplo do que foi feito no lado do servidor, o gerenciador de segurança deve ser instalado.

```
import java.rmi.*;

public class RmiTestClient {

  public static void main(String[] args) {
    try{
      System.setSecurityManager(new RMISecurityManager());
      String url = "rmi://k-2.ahand.unicamp.br/rmitest";
      RmiTestRef rt = (RmiTestRef) Naming.lookup (url);
      System.out.println("Calling for remote method...");
      String str = rt.echo(args[0]);
      System.out.println(str);
    }
    catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Código 4.11: Classe cliente

Instalação do repositório de nomes : o servidor de nomes *rmiregistry* acompanha o pacote JDK e é onde os objetos remotos são registrados. Uma aplicação cliente obtém referências para objetos remotos através do acesso a esse servidor que normalmente atende a porta de número 1099. Apenas um *rmiregistry* pode ser instalado em cada máquina; a tentativa de uma segunda instalação retorna uma exceção por conflito de portas. O repositório de nomes não é hierárquico, portanto não pode conter dois servidores com nomes iguais.

Depois de implementadas todas as etapas anteriores, o cliente, o servidor e a interface foram compilados e executados localmente. A seqüência de comandos para compilação e execução, assim como o resultado dessas operações, são mostrados no exemplo abaixo.

Embora muito simples, qualquer outra aplicação mais elaborada pode ser construída a partir desse modelo. Outras questões que eventualmente apareçam, poderão ser resolvidas mais facilmente depois de entendido e executado esse primeiro programa.

4.6.1 Parâmetros e Exceções em RMI

A interação entre dois objetos, o que solicita o serviço e outro que recebe a chamada, pode envolver o envio de argumentos. Pode ocorrer ainda o retorno de uma exceção. Essas são as formas como os objetos se

```
$ javac *.java
$ rmic RmiTest
$ rmiregistry &
[2] 18370

$ java RmiTest &
[3] 18385

$ java RmiTestClient Hello!
Calling for remote method...
Echo: Hello!
```

Exemplo 4.1: Execução de uma aplicação em RMI

comunicam em aplicações convencionais não-distribuídas e as mesmas são suportadas em RMI; no entanto há algumas diferenças.

Em Java, os objetos usados como argumentos — parâmetro ou resultado de retorno da execução de um método — são passados por referência. No entanto, essa abordagem não faz sentido em RMI onde os espaços de endereçamento são distintos. Em RMI os objetos são passados por valor, isto é, são copiados integralmente via rede. No caso de objetos remotos, como não há necessidade de transportar todo o objeto já que um objeto remoto pode ser acessado de qualquer localidade, apenas o *stub* é copiado para a outra localidade. Para que o lado que recebe um *stub* consiga lidar com esse objeto, ele precisa receber o código Java dessa classe. O papel do *stub* é, como já mencionado, o de ocultar do programador o trabalho de preparar e transmitir os argumentos. Classes de objetos usados como argumentos e exceções também precisam ser carregadas remotamente, isso pode envolver também o carregamento de classes derivadas daquelas que foram declaradas na definição de um método. Esse trabalho de carregamento de classes é executado pelo carregador de classes (*class loader*) que pode ser configurado para restringir determinadas operações. Enquanto o *security manager* determina o que uma classe carregada pode ou não executar, o *class loader* determina as localidades de onde uma classe pode ser carregada. O programador pode desenvolver e instalar o seu próprio gerenciador de segurança e seu próprio carregador de classes.

Capítulo 5

LegoShell

Aplicações distribuídas e orientadas a objetos podem ser desenvolvidas usando os recursos atualmente diponíveis na área de objetos distribuídos. Clientes e servidores podem ser implementados em linguagens distintas e há ferramentas para as linguagens de programação mais importantes. Pode-se considerar que essa tecnologia está bastante desenvolvida e, de modo geral, o desenvolvedor não está desprovido de recursos. No entanto, conforme pode ser observado no capítulo 3, o domínio dessa tecnologia não é uma tarefa fácil.

Portanto, as maiores dificuldades na área de objetos distribuídos não estão relacionadas com limitações técnicas impostas pelas atuais tecnologias. Algumas limitações existem mas, a princípio, os recursos disponíveis para o programador podem ser considerados suficientes para resolver a grande maioria dos problemas. As maiores dificuldades no entanto, estão relacionadas com o domínio das tecnologias de objetos distribuídos que, do ponto de vista do programador acostumado a desenvolver aplicações convencionais, são difíceis de dominar. Parte dessa dificuladade está associada à natureza das aplicações distribuídas. Outra parte no entanto, deve-se à falta de ferramentas que permitam um nível mais elevado de abstração ao programador.

A LegoShell é um sistema para desenvolvimento de aplicações distribuídas que prioriza o nível de abstração permitido ao programador. Tanto a interligação de objetos como as operações de troca de dados e chamadas de métodos podem ser facilmente implementadas.

A LegoShell surgiu dentro do projeto A-HAND, a proposta inicial era implementá-la sobre a versão distribuída da linguagem Cm [Fur91, Tel93] denominada CmD [Jr.94]. Algumas das idéias originais foram preservadas para esta implementação, outras foram modificadas. Basicamente, os conceitos da LegoShell que estão relacionados com os conectores, seção 5.4, é que delinearam as propriedades implementadas nesta nova versão.

A arquitetura da LegoShell é composta de duas camadas: o *sistema de suporte* e o *ambiente do desenvolvedor*, figura 5.1. O sistema de suporte será apenas introduzido neste capítulo porque os detalhes dessa camada estão mais relacionados com os aspectos de implementação, capítulo 6. O ambiente do desenvolvedor será apresentado mais detalhadamente neste capítulo com o objetivo de explicar as propriedades da LegoShell dentro da perspectiva de desenvolvimento de uma aplicação.

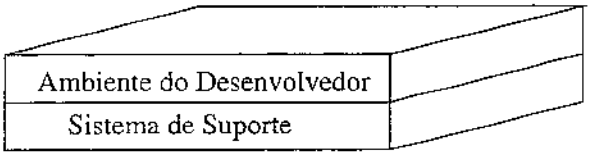


Figura 5.1: Camadas da LegoShell

Um objeto executável dentro da LegoShell, denominado *objeto LegoShell*, é uma instância de uma classe Java implementada sobre os recursos do sistema de suporte. A comunicação entre esses objetos é feita através de *portas* de entrada e saída que são facilmente criadas e interconectadas em tempo de execução. Objetos LegoShell são instanciados em qualquer localidade da rede e só a partir desse instante é que passam a ser realmente executados. Desse modo, um servidor remoto não precisa estar em execução enquanto aguarda chamada de seus serviços.

Uma aplicação LegoShell é construída com objetos implementados pelo usuário e com outros objetos, como por exemplo os conectores, já disponíveis na LegoShell.

A seguir, serão descritos o sistema de suporte (seção 5.1), o ambiente do desenvolvedor (seção 5.2) e três tipos de conectores (seção 5.4).

5.1 Sistema de Suporte

O conjunto de recursos estruturais necessários para criação e manipulação de objetos LegoShell é provido pela camada de nível inferior denominada *sistema de suporte*. Ela possui funcionalidades que facilitam as tarefas de utilização dos recursos da rede para estabelecimento de conexão, transmissão de dados e chamadas de métodos remotos. Efetivamente, o sistema de suporte cria um nível de abstração mais elevado para a utilização dos recursos de *sockets* e RMI na construção de aplicações Java.

O processo de construção de uma aplicação distribuída em LegoShell possui quatro etapas: implementação (seção 5.1.1), instanciação (seção 5.2.2), configuração (seção 5.2.3) e execução, (seção 5.2.4). Durante a implementação, o programador utiliza diretamente os recursos do sistema de suporte, nas três fases seguintes, o programador usa as funcionalidades providas pelo sistema de suporte através da interface da LegoShell. Todo o processo é bastante simples porque os recursos de baixo nível são criados e gerenciados de modo transparente para o programador.

5.1.1 Implementação de Objetos

O sistema de suporte provê funcionalidades para que um objeto LegoShell seja facilmente desenvolvido. Para implementar uma *classe LegoShell*, basta estender a classe LegoObject. Todo lego objeto é remoto e executável em uma *thread* , conforme figura 5.2.

As classes Produtor, código 5.1 e Consumidor, código 5.2 são dois exemplos de classes LegoShell.

O construtor de uma classe LegoShell recebe duas Strings como argumento: a primeira é o nome do contexto de nomes a que o objeto a ser instanciado pertence; a segunda é o nome para o objeto dentro de um contexto. O primeiro parâmetro é usado pelo sistema de suporte para o gerenciamento dos contextos. O segundo argumento é o nome pelo qual a instância será identificada pelo programador.

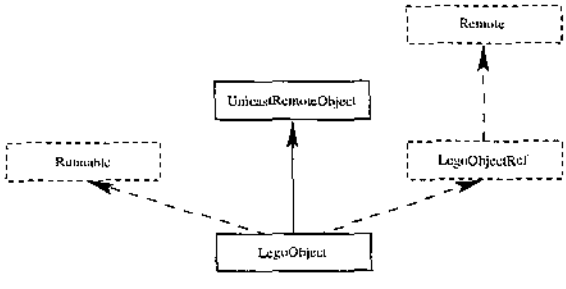


Figura 5.2: Classe LegoObject

```
import legoshell.*;
import java.io.*;
import java.rmi.*;

public class Produtor extends LegoObject {

  public Produtor (String contextName, String objectName)
                  throws RemoteException {
    super (contextName, objectName);
  }

  public void exec (){
    System.out.println ("Prod.started");
    try{
      outPorts = getOutputPorts ();
      OutputPort out = (OutputPort)outPorts.get ();
      DataOutputPort dop = new DataOutputPort (out);
      dop.writeInt(1);
    }
    catch (IOException e){
      e.printStackTrace();
    }
  }

  private PortList outPorts = null;
}
```

Código 5.1: Produtor LegoShell

Uma classe LegoShell deve implementar o método exec, acionado quando o usuário executar o comando `start` a partir da interface da LegoShell. No produtor, uma porta de saída é obtida e usada para criar uma DataOutputStream onde será escrito um valor inteiro. No consumidor, é obtida uma porta de entrada usada para criar uma DataInputStream de onde será lido o dado escrito pelo produtor. Qualquer outra *stream* poderia ser criada a partir de uma porta. As portas de entrada e saída são obtidas através dos métodos getOutputPorts e getInputPorts respectivamente. Esses métodos são definidos na classe LegoObject e ambos retornam um Vector de portas (PortList).

Em resumo, toda classe que o desenvolvedor de uma aplicação LegoShell irá implementar deverá ter o mesmo aspecto desses exemplos: será uma classe derivada da classe LegoObject, seu construtor será semelhante aos dos exemplos, e as funcionalidades deverão ser implementadas no método exec, de onde as portas de entrada e saída são acessíveis.

5.2 Ambiente do Desenvolvedor

O programador manipula os objetos LegoShell a partir do *ambiente do desenvolvedor*. Cada ambiente individualiza um contexto onde os objetos são manipulados para compor uma aplicação.

Depois de implementados os objetos de uma aplicação, como descrito na seção 5.1.1, e depois de localizados os nomes e os endereços dos objetos remotos que serão instanciados, a próxima etapa é executar o ambiente LegoShell.

Ao chamar o analisador sintático da LegoShell, como mostrado no comando abaixo, o usuário tem acesso à interface da LegoShell. Para cada execução desse tipo, é criado um novo contexto onde os objetos serão armazenados.

```
$ java LegoParser
```

```
import legoshell.*;
import java.io.*;
import java.rmi.*;

public class Consumidor extends LegoObject {

  public Consumidor (String contextName, String objectName)
                     throws RemoteException {
    super (contextName, objectName);
  }

  public void exec () {
    System.out.println ("Cons.started");
    try {
      inPorts = getInputPorts();
      InputPort in = (InputPort)inPorts.get();
      DataInputPort dip = new DataInputPort (in);
      System.out.println ("Consumidor_lendo_" + dip.readInt());
    }
    catch (Exception e) {
      e.printStackTrace ();
    }
  }

  private PortList inPorts = null;
}
```

Código 5.2: Consumidor LegoShell

A LegoShell possui um ambiente de programação onde o usuário, através da interface LegoShell, conta com recursos para instanciar, configurar e executar objetos. Esses objetos são vistos como partes interconectáveis que se juntam para compor uma aplicação.

Idealmente, conforme proposta inicial da LegoShell, o ambiente do desenvolvedor deveria possuir uma interface gráfica. Nesse caso, uma vez que os objetos tenham sido localizados, todas as operações poderiam ser realizadas de maneira bastante simples com o uso do mouse. Neste projeto no entanto, a interface gráfica não foi implementada e os comandos são digitados em uma interface mais simples.

Um objeto LegoShell pode ser desenvolvido pelo usuário ou pode ser um *objeto funcional* que foi implementado para aumentar os recursos do ambiente LegoShell. Os conectores, seção 5.4, são objetos funcionais que foram implementados para facilitar o desenvolvimento de aplicações mais complexas. Uma vez que o usuário possa contar com um grande número de objetos funcionais e objetos LegoShell espalhados pela rede, o processo de desenvolvimento de uma aplicação distribuída será simplificado: poucos objetos específicos precisarão ser implementados, objetos funcionais e servidores remotos deverão compor uma parte considerável da aplicação.

5.2.1 Interface LegoShell

A interface da LegoShell possui comandos para o usuário realizar operações de instanciação, configuração, remoção, listagem e execução de objetos. A sintaxe desses comandos é descrita em uma gramática cuja notação BNF é apresentada no exemplo 5.1. A seguir é feita uma descrição de cada um dos comandos.

New : este comando é usado para instanciar objetos. Os parâmetros são o identificador da classe, o endereço da máquina remota (*hostname*) e o nome para o objeto dentro do atual contexto. Caso o nome da máquina seja omitido, o sistema supõe que o código da classe a ser instanciada deve ser procurado no sistema local de arquivos como mostrado no segundo exemplo abaixo.

```
program         ::= (commandSequence)* <EOF>
commandSequence ::= [command (<SCOLON> command)* ] <EOL>
command         ::= create | connect | start | kill | list | help | quit
create          ::= objName <EQ> <NEW> className [<AT> hostName]
connect         ::= <CONNECT> objectName  objectName |
                    <CONNECT> objectName <DOT> portId objectName <DOT> portId
start           ::= <START> (objectName)+
kill            ::= <KILL> (objectName)+
list            ::= <LIST>
help            ::= <HELP> | <QMARK>
quit            ::= <EXIT> | <QUIT>
objectName      ::= <ID>
className       ::= <ID>
portId          ::= <ID>
hostName        ::= <ID> (<DOT> <ID>)*
<ID>            ::= ["a"-"z","A"-"Z"] (["a"-"z","A"-"Z","_","-","0"-"9"])*
<EQ>            ::= "="
<AT>            ::= "@"
<DOT>           ::= "."
<SCOLON>        ::= ";"
<EOL>           ::= "\n"
<NEW>           ::= "new"
<CONNECT>       ::= "connect"
<START>         ::= "start"
<KILL>          ::= "kill"
<LIST>          ::= "list"
<HELP>          ::= "help"
<QMARK>         ::= "?"
<QUIT>          ::= "quit"
<EXIT>          ::= "exit"
```

Exemplo 5.1: Gramática da LegoShell

Sintaxe: `objectName = new className ["@" hostName]`
Exemplo1: `myProducer = new Producer@k-2.ahand.unicamp.br`
Exemplo2: `localProd = new Producer`

Connect : usado para interligar dois objetos que foram instanciados e fazem parte do contexto da LegoShell. Este comando pode ser usado de duas maneira: na primeira, as portas já foram criadas em tempo de compilação e seus nomes são fornecidos como parâmetros;

Sintaxe: `connect objectSrc"."outputPort objectDest"."inputPort`
Exemplo: `connect myProducer.out myConsumer.in`

na outra maneira, as portas são criadas dinamicamente.

Sintaxe: `connect objectSrc objectDest`
Exemplo: `connect myProducer myConsumer`

Start : esse comando permite executar os objetos já configurados; os nomes dos objetos são fornecidos como parâmetro.

List : informa os objetos instanciados juntamente com suas portas. Atualmente este comando não exibe as conexões entre as portas, essa funcionalidade poderia ser facilmente implementado e seria ainda mais útil caso fosse usado recursos gráficos para exibir as conexÕes e seus estados.

Kill : usado para remover objetos do contexto corrente.

Help : digitanto-se `help` (ou ?) obtém-se um resumo dos comandos da LegoShell.

Exit : esse comando remove todos os objetos, destrói o atual contexto e abandona a interface da LegoShell. Os dois comandos `quit` e `exit` são equivalentes.

Esses comandos podem ser digitados via linha de comando como já mencionado ou podem fazer parte de um arquivo de *script* criado pelo usuário e fornecido durante a chamada do *parser*.

Esta interface é suficiente para os atuais objetivos deste projeto. Outra interface com recursos gráficos facilitaria ainda mais a utilização da LegoShell; no entanto, não foi implementada ficando como uma das sugestões apresentadas na seção 7.3.

5.2.2 Instanciação de Objetos

Os objetos locais ou remotos que irão integrar uma aplicação precisam ser instanciados. Isso é feito a partir da interface da LegoShell usando o comando `new`, conforme descrito na seção 5.2.1.

O programador instancia vários objetos para compor o seu ambiente e a partir de então, tanto os objetos implementados pelo programador quanto outros objetos remotos são manipulados da mesma maneira dentro da LegoShell. A figura 5.3 mostra um ambiente contendo cinco objetos instanciados, eles podem ser configurados (interconectados) de diversas maneiras para compor uma aplicação.

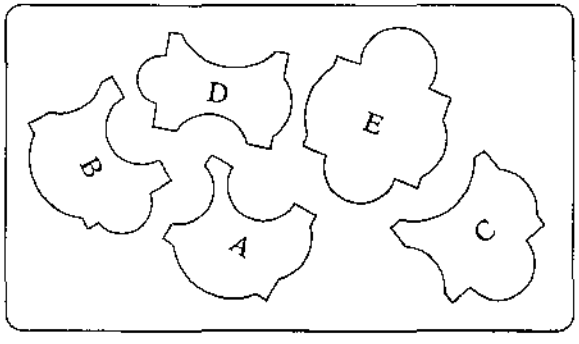


Figura 5.3: Ambiente LegoShell com vários objetos instanciados

Ao contrário de RMI, onde os servidores devem estar sendo executados para que estejam habilitados a receber chamadas remotas, em LegoShell os objetos são executados sob demanda. Um objeto LegoShell apenas será executado pela máquina virtual Java quando houver uma solicitação de seus serviços. Manter os objetos em execução constante é dispendioso, principalmente quando existe um grande número de servidores em uma mesma máquina ou quando poucos servidores consomem recursos consideráveis e seus serviços são raramente solicitados. Como aplicações Java normalmente consomem boa quantidade de memória, executá-las sob demanda é muito importante.

Para que objetos possam ser instanciados remotamente, um *daemon* deve estar sendo executado na máquina remota. Sua função é receber as chamadas e colocar o servidor em execução.

Essa abordagem de execução sob demanda incentiva espalhar um grande número de objetos por várias máquinas distribuídas pela rede e, desse modo, aumentar os recursos disponíveis e facilitar o desenvolvimento de novas aplicações.

5.2.3 Configuração da Aplicação

Esta fase de configuração envolve a interconexão de portas. Há objetos que têm suas portas definidas em tempo de compilação; em outros, as portas são criadas dinamicamente no momento em que o coman-

do connect é executado. Esses dois casos são necessários porque a criação dinâmica de portas deixa a operação mais simples para o desenvolvedor, além de permitir que o número de portas de um objeto seja flexível. A definição de portas em tempo de compilação é importante para permitir que sejam referenciadas discriminadamente por nomes.

Configurar uma aplicação significa interligar portas de entrada de objetos às portas de saída de outros. Essa é uma operação de alto nível e bastante simples dentro da LegoShell.

A figura 5.4 mostra duas configurações possíveis para o conjunto de objetos mostrados na página precedente. Duas aplicações distintas podem ser construídas através da interligação de objetos já instanciados.

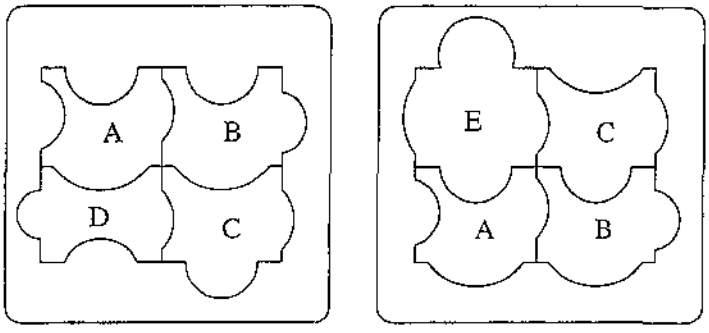


Figura 5.4: Duas configurações diferentes para objetos LegoShell

Como a configuração é feita depois que os objetos foram instanciados, é possível conceber classes mais genéricas. No momento em que um objeto é implementado, não é preciso definir o número de portas de entrada ou saída. Uma mesma classe pode ser usada em várias aplicações; em cada uma delas, sua instância é ligada a um número diferente de outros objetos. Por exemplo, é possível implementar uma classe genérica Produtor cujas instâncias atendem a diferentes números de consumidores sem que recursos sejam subutilizados — cada instância dessa classe estará usando apenas a quantidade de recursos (*sockets*) realmente necessários.

Conceito de Portas

Em LegoShell, os objetos se comunicam através de portas, figura 5.5, uma abstração de *sockets*. Por uma porta pode passar qualquer tipo primitivo da linguagem Java ou qualquer objeto serializado. Isso é possível porque as portas são derivadas de classes *Stream* de Java, portanto seguem o padrão de entrada e saída da linguagem, como explicado na seção 4.4.

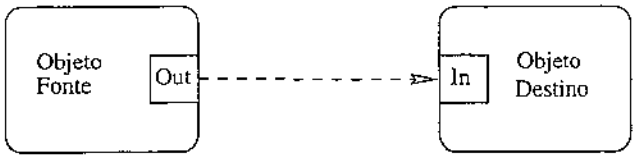


Figura 5.5: Portas de saída e entrada

Um objeto LegoShell pode ter mais de uma porta de entrada e mais de uma porta de saída. Desse modo é possível criar aplicações onde a comunicação envolve mais de duas entidades ligadas diretamente. As portas de um objeto podem ser criadas à medida em que as conexões são estabelecidas; o desenvolvedor não precisa definir o número de conexões que um objeto poderá conter.

Para se comunicar com outros objetos, um objeto LegoShell deve possuir pelo menos uma porta. Para definir uma porta em tempo de compilação, basta instanciar uma das classes InputPort e OutputPort, parte do

sistema de suporte. O nome da porta é um parâmetro usado para discriminar uma entre as diversas portas que um objeto pode possuir. A criação de portas dinâmicas é mais simples: todo o processo é realizado de modo automático quando o programador chama o método `connect` para interligar dois objetos. Cada execução desse método implica na criação de duas portas, uma de entrada e outra de saída. Objetos com portas definidas em tempo de execução podem ter outras portas criadas dinamicamente.

As portas criadas em tempo de compilação podem ser referenciadas de maneira discriminada usando o nome que lhe foi atribuído. Esse modelo não é adequado para um objeto que não sabe a princípio com quantos objetos irá se relacionar: por exemplo, quando não se sabe quantos clientes um servidor pode atender, as portas podem ser criadas dinamicamente.

A criação de portas à medida que as conexões vão sendo estabelecidas é mais simples. O acesso às portas é feito de modo genérico, levando em conta o número de portas que um objeto possui e não as identificações de cada uma delas. Uma analogia pode ser feita com os tradicionais servidores *multithreaded*: enquanto nestes uma nova *thread* é criada com uma nova porta TCP para atender um novo cliente, um servidor LegoShell é uma única *thread* que tem novas portas adicionadas à medida que conexões são solicitadas.

5.2.4 Execução

Depois de criados e configurados, os objetos de uma aplicação devem ser executados através da chamada ao comando `start` da LegoShell que executa o método exec do objeto.

A execução de uma aplicação consiste em executar os objetos dessa aplicação. A ordem de execução desses objetos não é relevante; por exemplo, se o consumidor é executado antes do produtor, o consumidor ficará bloqueado até que algum dado seja disponibilizado.

Depois que os objetos são executados, a configuração pode ser mantida e novamente executada; senão, os objetos podem ser reconfigurados ou removidos. Como em qualquer outra aplicação Java, quando um objeto é removido da LegoShell, o comando `kill` apenas desfaz a referência para esse objeto; a área de memória ocupada somente será liberada pelo coletor automático de lixo. Com objetos remotos, o coletor de lixo Java só não consegue desfazer as referências cíclicas, estas devem ser controladas pelo programador [CH96].

5.3 Exemplo de uma Aplicação LegoShell

Uma simples aplicação produtor/consumidor foi desenvolvida com o objetivo de exemplificar a utilização da LegoShell. As duas classes Produtor e Consumidor descritas na seção 5.1.1 são usadas nesse exemplo onde um produtor e um consumidor são instanciados e interligados; depois disso, ambos são executados e um dado inteiro é escrito pelo produtor e lido pelo consumidor.

Implementação de objetos : inicialmente, os objetos LegoShell que farão parte da aplicação devem ser implementados. Neste exemplo, serão usadas as classes Produtor, código 5.1, e Consumidor, código 5.2, já apresentados na seção 5.1.1 na página 36.

Instanciação de objetos : uma vez que o desenvolvedor conhece a identificação dos objetos que irá utilizar, o próximo passo é executar a LegoShell e, a partir de sua interface, instanciar dois objetos (`prod` e `cons`). Essas operações são mostradas no exemplo 5.2. A primeira linha é a invocação da interface da LegoShell. No final, o comando `list` é usado para mostrar que os objetos não possuem portas.

Configuração : o comando `connect` é usado para interligar os dois objetos. Como mostrado no exemplo 5.3, a configuração é bastante simples, basta conectar o produtor `prod` ao consumidor `cons`. Como esses objetos não possuem portas, uma porta de saída será criada para o objeto `prod` e uma porta de entrada para o objeto `cons`. Essas portas são manipuladas pelo sistema e por isso sua identificação não precisa ser tratada pelo programador.

```
$ java LegoParser

Legoshell> prod = new Produtor@k-2.ahand.unicamp.br
Legoshell> cons = new Consumidor@k-2.ahand.unicamp.br
Legoshell> list

Registered Objects are:

Object: cons
inputPorts: []
outputPorts: []

Object: prod
inputPorts: []
outputPorts: []
```

Exemplo 5.2: Execução da LegoShell e instanciação de objetos

```
LegoShell> connect prod cons
prod CONNECTED TO cons
Legoshell> list
Objects Registered are:

Object: cons
inputPorts: [1761217348]
outputPorts: []

Object: prod
inputPorts: []
outputPorts: [621191033]
```

Exemplo 5.3: Conexão do produtor com o consumidor

Execução : o comando `start` é usado para executar os objetos que foram configurados no passo anterior. Quando o método exec do produtor é disparado, ele escreve um dado em sua porta de saída que está conectada à porta de entrada do consumidor. Essa operação de escrita não é sincronizada: mesmo que o consumidor não leia esse dado, a operação de escrita é finalizada. A execução do objeto `cons` realiza a leitura do dado que se encontra no *buffer* de sua porta de entrada e escreve esse dado na tela, conforme mostrado no exemplo 5.4.

```
Legoshell> start prod
Prod.started
Produtor escrevendo: 1
Legoshell> start cons
Cons.started
Consumidor lendo: 1
```

Exemplo 5.4: Execução dos objetos

5.4 Conectores

Os recursos da LegoShell até aqui apresentados ajudam na construção de aplicações distribuídas porque facilitam as operações de conexão e de comunicação entre objetos. Além desses recursos, um conjunto de objetos funcionais pode ser implementado e disponibilizado para o programador. Desse modo, muitas

operações poderão usar esses objetos sem a necessidade do desenvolvedor implementar essas funcionalidades.

As linguagens de programação visual, muito usadas em aplicações comerciais não distribuídas, possuem muitas funcionalidades já disponíveis. A maior parte do código de uma nova aplicação não precisa ser reimplementado: o programador normalmente desenvolve apenas o código específico de sua aplicação e utiliza muitas outras funcionalidades disponíveis na linguagem. Essas facilidades foram as principais responsáveis pela popularização de linguagens como Visual Basic e Delphi, onde as operações mais comuns em aplicações comerciais como desenvolvimento de interfaces gráficas, manipulação de janelas, leitura e formatação de dados, podem ser realizadas com muita facilidade.

Assim como as aplicações convencionais, o desenvolvimento de aplicações distribuídas também possui algumas etapas repetitivas, trabalhosas e comuns a um grande número de aplicações. Essas funcionalidades podem ser providas como parte do ambiente de desenvolvimento e não precisam ser reimplementadas em cada aplicação. Em uma aplicação distribuída, boa parte do código que o programador precisa desenvolver está relacionado com tarefas de conexão e distribuição de dados entre os objetos.

Conectores [Dru95] são objetos LegoShell com o objetivo de facilitar as operações mais comuns dentro do processo de interligação de objetos distribuídos. Usando os recursos do sistema de suporte, a interconexão de objetos é bastante simples como exemplificado na seção 5.3. Muito mais que um canal de comunicação entre objetos remotos, os conectores podem conter semânticas adicionais que implementam desde comunicação entre grupos de objetos a funções de gerenciamento das conexões. Atualmente, as ferramentas de programação distribuída apenas viabilizam a construção de vias de comunicação entre os objetos, enquanto que a idéia dos conectores permite que os objetos utilizem vias previamente construídas.

Um conector, além de funcionar como um mecanismo para intermediar as ligações entre objetos remotos, possui outra grande vantagem, a padronização do processo de interligação de objetos. Essa padronização facilita também as etapas de depuração e manutenção do código porque o programador pode usar um conector já testado por um grande número de aplicações.

Os conectores *pipe*, *estrela* e *mbox* foram implementados nesse projeto e serão descritos a seguir. Os termos *produtor* e *consumidor* serão usados para denotar objetos ligados respectivamente à entrada e à saída de um conector.

5.4.1 Pipe

O conector *pipe* é um canal de comunicação unidirecional que interliga dois objetos LegoShell, figura 5.6. A função do *pipe*, o mais elementar dos conectores, é prover um canal de comunicação entre um produtor e um consumidor. O conceito é equivalente àquele tradicional usado nos sistemas operacionais: basicamente é um mecanismo que interliga a entrada de um dispositivo ou programa à saída de outro.

A figura mostra dois objetos sendo interligados por um conector *pipe*.

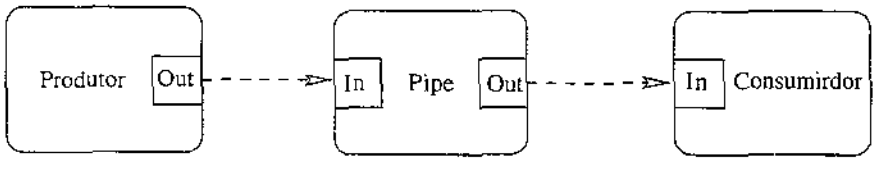


Figura 5.6: Conector *pipe*

Em LegoShell, o uso do comando `connect` poderia ser entendido como a inclusão de um *pipe* entre dois objetos. A princípio, essa propriedade dispensaria o uso deste conector. No entanto, o uso do *pipe* como foi implementado, além de padronizar as operações de comunicação, permite adicionar outras funcionalidades que operam sobre os dados trocados entre duas aplicações.

5.4.2 Estrela

O conector *estrela* é usado para interligar vários objetos e sua função é disseminar para os consumidores todos os dados que recebe dos produtores. Uma cópia de cada dado produzido é enviada para cada um dos consumidores ligados às portas de saída desse conector. Esse mecanismo permite que operações de *broadcast* sejam realizadas sem que o programador tenha que implementar essas funcionalidades.

Os consumidores ligados ao conector *estrela* devem receber todos os dados enviados pelos produtores. Há um problema em manter essa funcionalidade quando alguns consumidores lêem dados com velocidades muito diferentes. Os dados são temporariamente armazenados até que sejam lidos por todos os consumidores; por maior que seja o espaço disponível para armazenamento, eventualmente poderá se esgotar. Várias soluções podem ser adotadas como a suspensão das atividades dos produtores ou o descarte de dados pelos consumidores. Cada uma delas com suas vantagens e desvantagens em casos distintos. Essa discussão é mais relevante para os aspectos de implementação e será retomada na seção 6.3.6 na página 60.

A ordem em que os dados são escritos no conector é mantida e todos os consumidores recebem os dados na mesma seqüência. Para se estabelecer esta ordem, é considerado o tempo em que o dado foi recebido pelo conector e não o tempo em que esse dado foi enviado pelo produtor. Há diferenças porque os objetos podem estar localizados em máquinas distintas e ao receber duas mensagens de dois produtores diferentes, o conector não terá como saber qual delas foi enviada antes. Uma consideração mais elaborada seria observar a ordem em que os dados foram enviados pelos produtores ao invés da ordem em que chegaram até o conector. Essa política exige sincronização dos relógios das máquinas [LMS82], funcionalidade que não foi implementada.

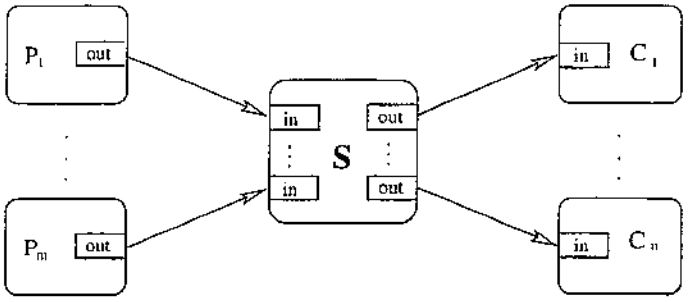


Figura 5.7: Conector *estrela* ligando produtores a consumidores

A figura 5.7 mostra 1 conector *estrela* (S), m produtores (Pi) ligados às portas de entrada e n consumidores (Ci) conectados às portas de saída.

5.4.3 Mbox

O *mbox* é outro tipo de conector que foi implementado neste projeto. Sua semântica é análoga a um sistema de caixa postal onde as entidades podem enviar ou retirar informações. Os dados são inseridos pelos produtores e removidos, não apenas lidos, pelos clientes. Ao contrário do conector *estrela*, onde o dado de cada produtor é enviado para todos os consumidores, no mbox cada dado é recebido apenas por um cliente.

Uma aplicação envolvendo um conector *mbox* e vários produtores e consumidores, tem o mesmo aspecto do conector *estrela* da figura 5.7. Nessa configuração, os consumidores concorrem entre si pelos dados enviados pelos produtores.

No conector mbox os dados são armazenados em uma estrutura do tipo FIFO (*first-in first-out*) antes de serem enviados para os consumidores. Essa fila mantém a ordem que os dados chegaram até o conector e não a ordem em que foram produzidos. Como no conector *estrela*, no mbox também poderia ser considerado o instante de envio dos dados ao invés do instante de recebimento. Esse mesmo tipo de sincronismo poderia

ser considerado para atendimento dos consumidores: seria considerado o instante em que um consumidor fez o pedido e não o instante em que o conector recebeu esse pedido. Esta solução, como já mencionado, depende da utilização de recursos para sincronismo dos relógios de todas as máquinas envolvidas.

5.4.4 Exemplo de Utilização

Esta seção mostra os passos do desenvolvimento de uma aplicação cliente/servidor semelhante àquela exibida na figura 5.7. As etapas envolvidas, conforme detalhado na seção 5.3, são: instanciação, configuração e execução dos objetos.

Este exemplo possui dois produtores (`prod1`, `prod2`), dois consumidores (`cons1`, `cons2`) e um conector *estrela* (`str`).

```
Legoshell> prod1 = new Produtor
Legoshell> prod2 = new Produtor
Legoshell> cons1 = new Consumidor
Legoshell> cons2 = new Consumidor
Legoshell> str = new Star

LegoShell> connect prod1 str
prod1 CONNECTED TO str

LegoShell> connect prod2 str
prod2 CONNECTED TO str

LegoShell> connect str cons1
str CONNECTED TO cons1

LegoShell> connect str cons2
str CONNECTED TO cons2

LegoShell> start prod1 prod2 str cons1 cons2
```

Exemplo 5.5: Configuração de um sistema com vários clientes e vários servidores

Como não foi informado o endereço das classes Produtor, Consumidor e Star, o sistema considera que todas são locais.

Após a execução do comando `start`, cada objeto produtor irá escrever no conector *estrela* e cada objeto consumidor receberá uma cópia dos dados produzidos.

5.4.5 Outras Vantagens dos Conectores

Os três tipos de conectores descritos anteriormente foram implementados com o objetivo de apresentar uma proposta para interligação de objetos distribuídos. Muitas outras semânticas poderão ser implemetadas assim como outros recursos para gerenciamento das conexões e de tolerância a falhas. Essa seção discute outros recursos que podem ser implementados para aprimorar o mecanismo e aumentar sua importância para o desenvolvimento de aplicações distribuídas.

Outro conector que poderia ser implementado para facilitar a comunicação entre grupos de objetos é o conector de *multicast*. Tanto o conector *estrela* quanto o *mbox* podem ser vistos como mecanismos de *multicast*. Nesse sentido, há recursos para distribuir mensagens para *todos* os consumidores (conector *estrela*) ou para apenas *um* consumidor (conector *mbox*). Um outro tipo de conector mais geral poderia ser implementado para permitir a formação dinâmica de grupos de consumidores ligados ao conector. Esse conector, denominado de conector de *multicast*, possuiria um argumento inteiro — um valor entre 1 e N onde N é o número total de consumidores — para configurar qual o número de consumidores que deverão receber

as mensagens. Uma vez implementado esse conector, os conectores *estrela* e *mbox* poderiam ser entendidos como especializações do conector de *multicast*: o valor 1 como parâmetro seria equivalente ao *mbox* e o valor N seria equivalente a um conector *estrela*.

Um conector é um elemento ativo cujos principais objetivos são facilitar a interligação de objetos e padronizar o processo de comunicação. Uma das vantagens de se ter um processo padronizado de estabelecimento de comunicação está relacionado com a facilidade para adição de novas funcionalidades. Como todos os conectores possuem uma superclasse comum, uma nova propriedade poderia ser implementada uma só vez e atender a todas os tipos de conectores. O gerenciamento das conexões é uma das principais áreas que podem se beneficiar dessas propriedades. Alguns exemplos de funções de gerenciamento que podem ser implementadas nos conectores são:

- número de mensagens trocadas entre dois objetos;
- tipo de dados que estão trafegando pelo conector;
- quais os serviços mais requisitados;
- quais as máquinas mais usadas;
- tempo médio de acesso para cada serviço;
- estatísticas que permitem determinar qual o melhor horário para que um determinado servidor seja usado.

Os conectores, assim como a LegoShell, é uma idéia a ser aprimorada. Depois que um número expressivo de funcionalidades for implementado, o desenvolvimento de aplicações distribuídas passará a ser um processo muito menos dispendioso para o programador.

5.5 Resumo

O sistema LegoShell é formado por um conjunto de classes Java que oferecem recursos para facilitar a criação e interligação de objetos distribuídos. O objetivo da LegoShell é permitir que todas as etapas de desenvolvimento de uma aplicação distribuída sejam realizadas com alto nível de abstração. O processo de construção de uma aplicação distribuída em LegoShell tem as seguintes etapas: implementação, instanciação, configuração e execução dos objetos. Contextos de execução de objetos são separados e os objetos são executados apenas quando necessário; essas são duas vantagens com relação a RMI Java onde o contexto é único e os servidores remotos devem ser executados antes de receber uma chamada remota.

A arquitetura do sistema LegoShell é composta por dois módulos: o sistema de suporte e o ambiente do desenvolvedor. Essas duas camadas facilitam o trabalho de implementação de objetos. Outros recursos importantes são os objetos funcionais que podem ser usados nas aplicações.

O sistema de suporte é a camada de baixo nível que provê recursos para facilitar as operações que exigiriam mais esforço de programação como o estabelecimento de conexões e a troca de dados via rede. Esses recursos são úteis tanto para o programador implementar seus objetos quanto para a camada ambiente do desenvolvedor.

O ambiente do desenvolvedor possui uma interface que é usada na concepção de aplicações. Os objetos manipulados na interface podem ter sido obtidos na rede ou implementados pelo programador.

Além dos objetos que o desenvolvedor implementou e de outros objetos remotos, uma aplicação LegoShell pode usar objetos funcionais — aqueles que foram desenvidos para facilitar as tarefas mais comuns em aplicações distribuídas. Neste projeto foram implementados três tipos de conectores com o objetivo de apresentar as vantagens desse mecanismo de objetos funcionais. Os principais objetivos dos conectores são:

viabilizar diversos tipos de conexões, padronizar a maneira como as conexões são estabelecidas e facilitar a implementação de funções importantes como as de gerenciamento. Aplicações mais complexas poderão ser construídas com facilidade à medida que novos objetos funcionais sejam disponibilizados.

Capítulo 6

Implementação

O capítulo 5 explica as funcionalidades que fazem parte deste projeto sem, no entanto, entrar nos detalhes de como foram implementadas. As questões relativas à implementação da interface da LegoShell, do sistema de suporte, e dos conectores são abordadas neste capítulo.

Todo o sistema foi desenvolvido em Java usando JDK 1.1.5 sobre o sistema operacional Linux versão 2.0.34. Basicamente, foram usados os recursos de RMI, *threads* e *sockets* , providos por Java e explicados no capítulo 4.

6.1 Interface

A interface da LegoShell foi implementada utilizando o gerador automático de *parser* JavaCC. Esta seção descreve a gramática da LegoShell e como foi feita a especificação de sua sintaxe e sua análise semântica.

6.1.1 Gramática da LegoShell

A gramática da LegoShell, especificação mostrada no exemplo 5.1 na página 39, é uma *Gramática Livre de Contexto* (GLC). Uma GLC é uma seqüência de produções formadas por um símbolo não-terminal do lado esquerdo e por um ou mais símbolos terminais e não-terminais do lado direito. Um símbolo terminal é obtido a partir de um alfabeto previamente especificado. Uma gramática especifica uma linguagem, um conjunto infinito de possíveis combinações de símbolos terminais resultantes da substituição repetitiva e seqüencial dos símbolos não-terminais [GJS96].

6.1.2 Construção da Interface

JavaCC [1] (*Java Compiler Compiler*) é a ferramenta mais usada atualmente na geração de *parser* para aplicações Java. Um gerador automático de *parser*, como JavaCC, é um programa que lê uma especificação léxica e sintática de alto nível e gera um programa capaz de validar entradas no formato da gramática lida.

A vantagem do uso de uma ferramenta como JavaCC é que o desenvolvedor pode se abstrair dos detalhes de implementação e se concentrar apenas na definição das regras léxico-gramaticais, além de facilitar o tratamento semântico. JavaCC foi totalmente desenvolvido em Java e o código por ele gerado também é 100% Java.

O compilador JavaCC recebe como entrada, um arquivo com extensão `.jj` cujo conteúdo é uma descrição gramatical da linguagem que está sendo construída. O compilador JavaCC é executado sobre esse arquivo

[1] Página do produto: `http://www.suntest.com/JavaCC/`

e o resultado é outro arquivo de mesmo nome com extensão .java contendo o código Java do *parser*. A análise semântica é realizada à medida que os comandos vão sendo reconhecidos pelo *parser*.

Em JavaCC, uma gramática é descrita usando uma sintaxe própria, semelhante à da linguagem Java. A seguir, será explicado como os elementos de uma gramática podem ser definidos em JavaCC e como é feito o tratamento semântico dos comandos reconhecidos durante a análise sintática.

```
TOKEN :
{
  < ID:  ["a"-"z","A"-"Z"] (["a"-"z","A"-"Z","_","-","0"-"9"])* >
}

String objectName() :
{}
{
  <ID>
}
```

Código 6.1: Definição de símbolos gramaticais em JavaCC

Símbolos : no código 6.1 são mostrados dois exemplos de definições de símbolos gramaticais: no primeiro, é definida uma seqüência do alfabeto válida para um símbolo terminal, o identificador `ID`; no segundo, um `objectName` é definido como um símbolo não-terminal que pode ser reduzido ao terminal `ID`. Portanto, pela definição apresentada, um identificador em LegoShell deve necessariamente começar com uma letra minúscula ou maiúscula seguido por uma seqüência, eventualmente vazia, de letras, números, traços ou hífens. Pelo mesmo exemplo, o nome de um objeto pode ser reduzido a um identificador.

Produções : as produções gramaticais são escritas como métodos com algumas modificações com relação à sintaxe original da linguagem Java. O cabeçalho representa o lado esquerdo da produção e o corpo do método, o lado direito.

Semântica : o tratamento semântico em JavaCC é feito à medida que as reduções vão sendo realizadas, os comandos do usuário são interpretados. No código 6.2 é mostrado como o comando `start` é reconhecido. Há uma variável local `objName` que armazena o nome do objeto obtido durante a análise sintática. O tratamento semântico, execução da funcionalidade definida para o comando `start` sobre o objeto `objName`, é feito dentro do espaço reservado para o código.

```
void start() :
{
  String objName = null;
}
{
  <START> (objName = objectName() {
     // código Java
  })
}
```

Código 6.2: Tratamento semântico em JavaCC

O *parser* é implementado em uma classe Java executável a partir da linha de comando. O resultado da execução dessa classe é um interpretador de comandos — uma *shell* — que é a interface da LegoShell. Opcionalmente, o programador pode criar um arquivo (*script*) com os comandos da LegoShell e executar o *parser* sobre esse arquivo.

6.2 Implementação do Sistema de Suporte

O sistema de suporte oferece serviços implementados sobre recursos cuja utilização demandam maior esforço técnico e dificultam o desenvolvimento de aplicações distribuídas. Para que objetos remotos sejam utilizados mais facilmente, é necessário que recursos como RMI, *sockets* e *threads* sejam manipulados por uma camada inferior e de modo transparente para o programador.

As conexões entre os objetos são estabelecidas usando RMI, as operações de troca de dados entre duas portas foram diretamente implementadas em *sockets* . *Threads* foram usadas para evitar a interrupção do fluxo principal do programa ou para melhorar a eficiência de algumas operações.

O sistema de suporte disponibiliza serviços de alto nível os quais o programador e o sistema utilizam para criar e manipular objetos dentro da LegoShell.

A implementação do sistema de suporte será explicada em várias subseções. O gerenciador de contextos (seção 6.2.1) é o módulo que permite criar mais de um ambiente para registro de objetos. Depois que um contexto foi gerado, o programador usa vários comandos para criar (seção 6.2.2), remover (seção 6.2.3), configurar (seção 6.2.4) e executar objetos (seção 6.2.5).

6.2.1 Gerenciador de Contextos

O *Gerenciador de Contextos* (GC) é o módulo da LegoShell que permite a criação de vários contextos de nomes. Ao contrário de RMI onde o repositório de nomes é global, no GC um usuário possui seu contexto individual onde são registrados seus objetos.

Quando a LegoShell é executada, um novo *contexto de nomes* é criado e registrado no GC. Para outras execuções da LegoShell serão criados outros contextos, desse modo, os objetos de contextos diferentes são registrados em locais separados. As operações de criação, manipulação e remoção dos contextos de nomes são atribuídas ao GC, classe ContextManager.

Um *registrador*, classe ObjectRegistry, deve ser executado antes da execução da LegoShell. Sua função é interagir com o repositório de nomes `rmiregistry`. Quando a classe ObjectRegistry é executada, um GC é registrado no repositório de nomes de RMI. Quando a LegoShell é executada e um novo contexto de nomes precisa ser inserido no GC isso também é feito por intermédio do registrador.

```
$ java ObjectRegistry &
```

Um objeto da classe ContextManager é instanciado e registrado no `rmiregistry` quando o registrador é executado. Esse GC possui uma tabela onde são inseridos os contextos de nomes, sendo uma entrada da tabela para cada execução da LegoShell. Depois disso, a LegoShell pode ser executada e para cada execução, um novo contexto será criado e registrado no gerenciador.

Quando a LegoShell é executada, conforme comando abaixo, o objeto registrador busca o GC no repositório de nomes e um novo contexto de nomes — objeto da classe NameContext — é criado e inserido no tabela do GC.

```
$ java LegoParser
```

Os contextos, assim como os objetos, são acessados por nomes. No entanto, apenas os objetos precisam ter seus nomes controlados pelo programador enquanto que os nomes dos contextos são gerados e controlados pelo sistema.

O GC utiliza uma estrutura de dados do tipo tabela (*hashtable*) para armazenar os contextos. Cada contexto, por sua vez, é uma referência para um contexto de nomes — objeto da classe NameContext. Essa classe possui uma nova tabela onde são mapeados os objetos criados neste contexto. Em resumo, os objetos criados são guardados no contexto de nomes que está registrado no GC, este por sua vez, está registrado no

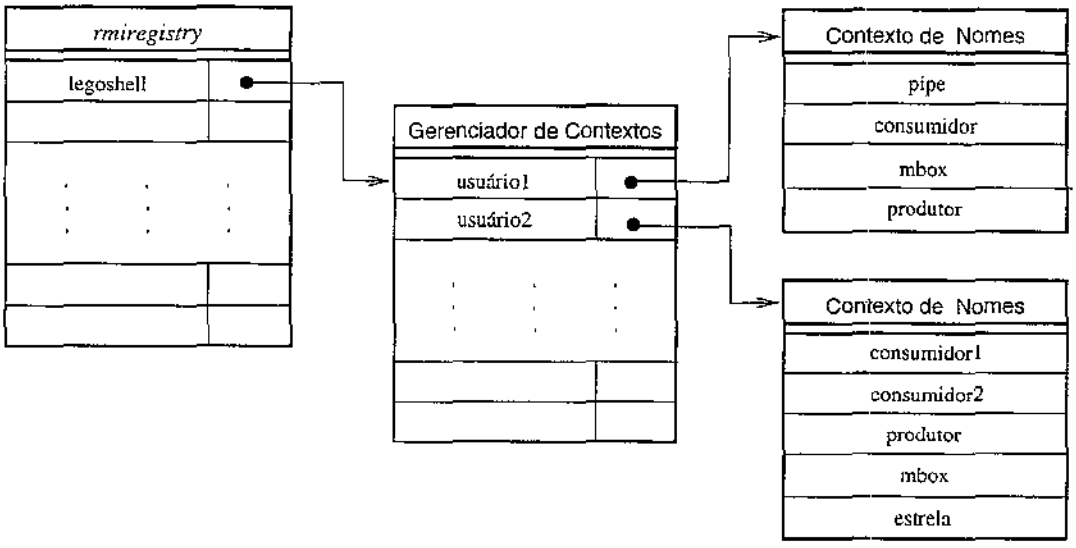


Figura 6.1: Contexto de objetos da LegoShell

repositório de nomes de RMI. A figura 6.1 mostra como os contextos de nomes são registrados indiretamente no repositório de nomes `rmigegistry`.

Depois que o usuário executa a LegoShell e um novo contexto foi criado, o programador tem acesso à interface da LegoShell. As subseções seguintes descrevem a maneira como foi implementada cada uma das operações que o programador pode utilizar a partir da interface.

6.2.2 Criação de Objetos

Os objetos que compõem uma aplicação LegoShell podem ser locais ou remotos. O comando `new` é usado para instanciar objetos e os parâmetros são o nome da classe acompanhado do endereço da máquina mais o nome para a identificação do objeto. Esse nome é associado a uma referência para o objeto e é usado pelo programador para identificá-la.

Os objetos que uma aplicação pode instanciar não precisam estar em execução. Uma classe em LegoShell deve ser implementada e seu código disponibilizado no sistema de arquivos. A partir de então, instâncias dessa classe podem ser criadas à medida que as aplicações executem o método `new`. O processo de criação de uma instância a partir do nome de uma classe é descrito a seguir:

- o método estático forName da classe Class de Java é executado passando como argumento o nome da classe a ser instanciada. O código dessa classe é procurado no sistema local de arquivos, mais especificamente, nos diretórios definidos na variável de ambiente `CLASSPATH`. Caso a classe não seja encontrada, uma exceção é retornada.

- o retorno da chamada anterior é um objeto da classe Class sobre o qual é executado o método getConstructor que retorna o construtor, um objeto da classe Constructor. Como uma classe pode ter vários construtores, uma lista de parâmetros deve ser fornecida para que possa ser identificado um único construtor.

- obtido o construtor, seu método newInstance é executado para criar uma instância dessa classe. Nesse instante, o construtor do objeto LegoShell é executado para gerar suas estruturas internas.

O processo acima ocorre quando o endereço da máquina é omitido durante a execução do comando `new`. Quando um objeto é remoto, e o endereço de uma máquina é fornecido, o mesmo procedimento é realizado através de um *daemon* que está executando na máquina remota.

O *daemon* é um servidor remoto que fica executando em cada máquina potencialmente fornecedora de objetos para aplicações LegoShell. Sua função é receber e processar as chamadas remotas para criação de objetos na máquina em que está executando.

Quando o usuário executa o comando `new` e fornece o endereço de uma máquina como parâmetro, a LegoShell utiliza RMI para se comunicar com o *daemon* e solicita que seja criado um objeto. O *daemon* então localiza a classe do objeto solicitado e o instancia localmente.

6.2.3 Remoção de Objetos

Os objetos criados pelo usuário são armazenados em seu contexto de nomes, uma estrutura de dados do tipo *hashtable* de Java. Quando um objeto é instanciado, uma associação da forma mostrada abaixo é armazenada nessa tabela. O lado esquerdo, `objectID` é o nome escolhido para identificar o objeto e `objectRef` é a referência para o objeto.

objectId ⇒ objectRef

A operação de remoção de um objeto consiste em desfazer essa associação. No entanto, essa operação não finaliza o ciclo de vida do objeto. A tabela contém apenas a referência para um objeto que pode estar localizado remotamente. O objeto somente será removido da memória pelo coletor de lixo (*garbage collector*) da linguagem depois que todas suas referências tiverem sido desfeitas.

O comando `kill` é usado para desfazer a associação realizada durante a criação do objeto. Quando o usuário abandona a LegoShell, as referências para os seus objetos e para o seu contexto são desfeitas automaticamente.

6.2.4 Configuração

A configuração de uma aplicação LegoShell consiste em estabelecer conexões entre os objetos cujas referências estão armazenadas no contexto de nomes corrente. Essa operação é realizada através do comando `connect` que interliga duas portas complementares: uma de entrada, associada ao *objeto destino*; outra de saída, associada ao *objeto fonte*, como mostrado na figura 5.5 na página 41.

As classes InputPort e OutputPort cuja estrutura hierárquica é mostrada na figura 6.2 na página seguinte, definem os dois tipos de portas da LegoShell. Essas classes implementam a interface Port e são subclasses de InputStream e OutputStream respectivamente. Como toda porta LegoShell é uma *stream*, o padrão de entrada e saída de Java é mantido. A classe InputPort implementa a interface Runnable para que as portas de entrada possam utilizar recursos de *threads*. As classes DataInputPort, DataOutputPort, ObjectInputPort e ObjectOutputPort completam o conjunto de portas da LegoShell e juntamente com as demais classes deste diagrama serão detalhadas na seção 6.3.1 na página 56.

Quanto à maneira como são criadas, as portas LegoShell são classificadas em dois grupos: as portas estáticas são definidas no momento em que um objeto é implementado; já as portas dinâmicas são criadas à medida que as coniguraçães entre objetos são estabelecidas através do comando `connect`.

Portas Estáticas

Uma porta estática é definida no construtor da classe do objeto LegoShell e, portanto, é criada no momento em que o objeto é instanciado através do comando `new`. O identificador dessa porta é escolhido pelo programador e fornecido como parâmetro para o construtor da classe InputPort ou OutputPort.

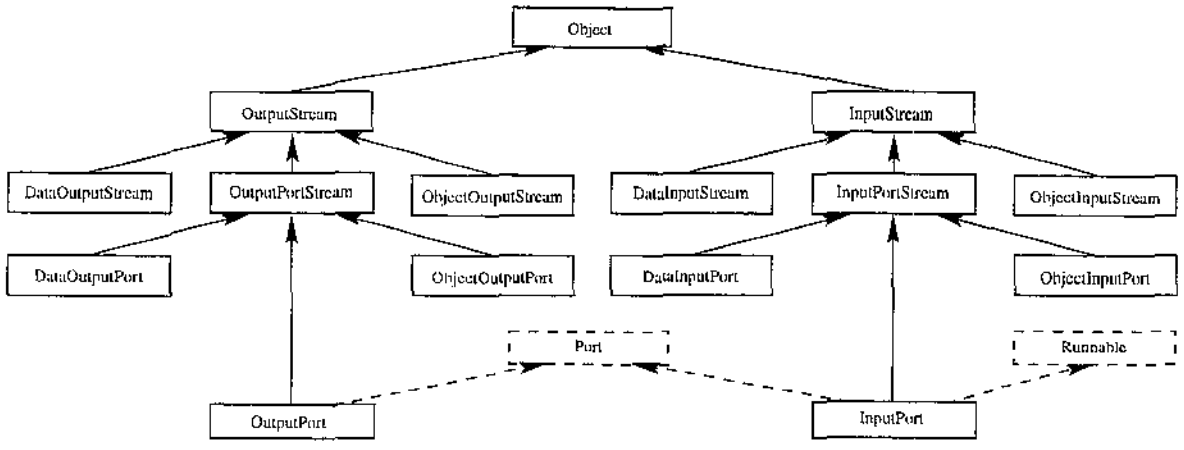


Figura 6.2: Portas da LegoShell e as correspondentes *streams* de Java

Esse identificador é associado a uma porta — uma referência para um objeto do tipo Port — e essa associação é armazenada em uma tabela do tipo *hastable* que existe em cada objeto. Depois disso, as operações de escrita e leitura codificadas no método exec referenciam as portas pelos seus nomes.

Na conexão de objetos usando portas estáticas, o comando `connect` recebe quatro argumentos: dois objetos mais os identificadores de duas portas sendo uma de saída e outra de entrada.

Portas Dinâmicas

As portas dinâmicas são criadas durante a fase de configuração de uma aplicação, portanto, depois que o objeto LegoShell foi instanciado. Inicialmente, o objeto pode ser criado sem portas e elas podem ser criadas dinamicamente executanto o comando `connect`. Desse modo, o número de portas de um objeto depende das necessidades de cada aplicação e os *sockets* são alocados à medida que as conexões entre objetos vão sendo configuradas. Ao contrário disso, com as portas estáticas os recursos de comunicação ficam disponíveis para o objeto desde sua criação, o número de portas é fixo e precisa ser definido em tempo de compilação da classe.

Um objeto LegoShell possui duas listas: uma para conter as portas de entrada e outra para as portas de saída. Quando o objeto é criado, nenhuma porta dinâmica é associada a ele, apenas quando o comando `connect` for executado é que as portas serão criadas e adicionadas à lista de portas desse objeto. O código que acessa as portas dinâmicas é definido no método exec, portanto, o acesso à lista de portas ocorre quando o objeto for executado.

As portas dinâmicas são armazenadas na mesma estrutura *hashtable* usada para guardar as portas estáticas. Uma diferença é que a identificação de uma porta dinâmica é gerada automaticamente. Os métodos, getInputPorts e getOutputPorts retornam listas de portas de onde são obtidas as portas de entrada e saída respectivamente.

Para o programador de uma classe LegoShell, a principal diferença entre portas dinâmicas e estáticas está na maneira como o método exec deve ser implementado. Usando portas estáticas, os nomes das portas são definidos no construtor da classe LegoShell e referenciados no código do método exec. Com as portas dinâmicas, o método exec é codificado para acessar as portas a partir das listas de portas do objeto. Como essas listas modificam durante a configuração dos objetos, o programador desenvolve o código para acessar as portas de modo genérico.

Estabelecimento de Conexão

O processo de interconexão de duas portas, estáticas ou dinâmicas, é realizado conforme descrito a seguir.

- o usuário executa o comando connect dentro da interface da LegoShell. Esse comando possui dois ou quatro argumentos: no primeiro caso, são fornecidos os nomes do objeto fonte e do objeto destino e o sistema cria as duas portas; no segundo caso, as portas de entrada e saída já foram criadas e seus nomes são fornecidos juntamente com a identificação dos dois objetos.

- na máquina do objeto destino, um *socket* — instância da classe ServerSocket de Java — é criado e seu método accept é executado com a função de "ouvir" uma porta TCP. O número da porta TCP é fornecido de acordo com a disponibilidade no sistema, tipicamente é obtida a primeira porta livre. A chamada ao método accept bloqueia a execução até que a conexão seja estabelecida. Para contornar essa situação e permitir que mais de uma conexão possa ser realizada de modo independente, essa operação é executada em uma *thread*

- o número da porta TCP obtida pela porta de entrada LegoShell e o endereço da máquina do objeto destino são enviados para o objeto fonte. Essas operações de controle são realizadas pelo sistema usando RMI.

- o objeto fonte instancia um *socket* local e abre uma conexão com o *socket* remoto cuja identificação — porta TCP mais *hostname* — foi recebida no passo anterior. Essa operação concretiza o processo de conexão das duas portas.

- os métodos que o objeto fonte usa para escrever dados em uma porta de saída da LegoShell serão mapeados em operações de escrita sobre uma *outputstream* ligada ao *socket*. De modo análogo, as operações de recebimento, realizadas pelo objeto destino sobre uma porta LegoShell de entrada, são transformadas em leituras sobre uma *inputstream*.

Os recursos de RMI são usados para se obter as portas dos objetos e para se estabelecer a conexão entre elas. No entanto, depois que a conexão é estabelecida, a troca de dados utiliza *sockets* que é um mecanismo mais eficiente.

Depois de estabelecidas as conexões, como nas demais operações de entrada e saída de Java, qualquer tipo primitivo ou objeto pode ser transportado pelas portas da LegoShell. Em Java, as aplicações utilizam o mecanismo de *streams* como padrão de entrada e saída de dados. Por exemplo, DataOutputStream é usada para escrever dados formatados em qualquer dos tipos primitivos e ObjectOutputStream permite escrever objetos. Para obter uma dessas instâncias, o programador precisa de um objeto da classe Stream para usar como parâmetro nas chamadas dos construtores. Como em LegoShell as Portas são derivadas de Stream, as mesmas facilidades de Java podem ser usadas pelas aplicações LegoShell. Por exemplo, uma porta LegoShell pode ser usada para a obtenção de uma instância das classes DataInputStream ou ObjectInputStream.

6.2.5 Execução de Objetos

Um objeto LegoShell deve ser definido em uma classe que estende a classe LegoObject onde está definido o método abstrato exec. Portanto, todas as classes que definem um objeto em LegoShell precisam implementar esse método. Esse modelo é semelhante ao empregado por Java para definição de *threads* onde uma classe precisa implementar suas funcionalidades em um método denominado run.

A classe LegoObject implementa a interface Runnable, portanto, todo objeto LegoShell pode ser executado como uma *thread* . Essa propriedade permite que vários objetos possam ser executados ao mesmo tempo dentro da LegoShell.

O método exec de uma aplicação será disparado quando o comando `start` da LegoShell for executado, o argumento desse comando é o nome de um objeto. A referência para esse objeto é obtida do contexto de nomes e uma chamada remota é efetuada para informar que uma *thread* deve ser criada e executada.

6.3 Implementação dos Conectores

Os conectores são objetos implementados sobre o sistema de suporte, portanto, herdam da classe LegoObject as funcionalidades de instanciação remota, configuração e execução. Outro importante recurso da LegoShell usado pelos conectores é a criação dinâmica de portas; o número de objetos ligados a um conector não pode ser definido em tempo de compilação. Esta seção detalha os conectores e outras classes que foram usadas na sua implementação.

De agora em diante, para evitar a repetição de termos, os nomes das classes serão abreviados, *input* e *output* poderão ser omitidos dos nomes das classes para generalizar os dois casos. Por exemplo, a palavra Stream será usada quando não houver necessidade de diferenciar as classes InpuStream e OutputStream.

Basicamente, a implementação dos conectores teve que resolver dois problemas: manter o padrão de entrada e saída Java e separar as unidades de dados para que sejam tratadas pelas semânticas de cada conector. Uma vez solucionados esses problemas, a implementação das semânticas em si, foi relativamente mais simples.

Para que os objetos possam trocar informações dentro do padrão de entrada e saída Java usando os conectores, um novo grupo de *streams* precisou ser implementado, conforme descrito na seção 6.3.1. As operações sobre os conectores manipulam unidades de dados, seção 6.3.2, que são armazenadas em um *buffer* circular 6.3.4. O mecanismo de caracter separador, seção 6.3.3, permite identificar as unidades de dados que são trocadas entre duas aplicações. Depois de implementadas essas funcionalidades, o trabalho de implementação dos conectores pôde se concentrar no desenvolvimento de suas semânticas.

6.3.1 Portas e *Streams*

Conforme explicado na seção 6.2.4 na página 53, quando dois objetos têm suas portas interligadas diretamente, o padrão de entrada e saída da linguagem Java é mantido. Como o conceito de *streams* é familiar ao programador Java, essa propriedade tende a facilitar muito o desenvolvimento das aplicações distribuídas em LegoShell. Esta seção descreve como o mecanismo de *streams* foi adaptado para funcionar com os conectores.

Na interligação direta, manter o padrão de entrada e saída Java foi relativamente simples porque as portas da LegoShell usam *sockets* e estes se comunicam através de *streams*. Como Port é subclasse de Stream, a partir de uma porta é possível obter outras *streams*. De modo geral, o sistema de suporte estabelece as conexões sobre os recursos de *stream* mas sem interferir no processo de transferência dos dados. Já no caso dos conectores, não é suficiente prover ligação entre os canais de comunicação; os dados que trafegam por um conector precisam ser manipulados em suas estruturas, não basta tratá-los apenas como um conjundo de caracteres.

Quando um produtor envia um dado para o conector, os *bytes* são armazenados em um *buffer* e depois são repassados para um consumidor. Se um produtor envia um tipo `int` para o conector, devido à maneira como é implementada a operação de escrita, o conector não tem como identificar os quatro *bytes* recebidos como um `int`. Os dados trafegam por uma *stream* como um conjunto de caracteres e não há, nesse nível, distinção de tipos ou estruturas. No momento em que o consumidor solicita um `int` através do método readInt, do mesmo modo o conector não saberá quantos bytes precisam ser enviados como resposta.

Ao intermediar a comunicação entre dois objetos, um conector precisa saber onde começa e onde termina cada unidade de dados. Quando um produtor escreve um `int`, um consumidor deverá ler um dado do mesmo tipo. O conector, que armazenou quatro *bytes* quando recebeu esse `int`, precisa ler quatro *bytes* do *buffer* e enviá-los para o consumidor quando este dado for solicitado. No entanto, o conector não tem como identificar

o final de um dado e o início de outro. Nas operações sem um conector, essa dificuldade não existe porque as duas extremidades de uma *stream* estão interconectadas.

Em Java, uma operação de escrita é implementada em baixo nível através de chamadas ao método write da classe OutputStream que escreve um *byte* por vez. Do mesmo modo, uma operação de leitura é feita invariavelmente através de chamadas ao método read da classe InputStream. Em outras palavras, se o conector recebe todos os *bytes* que chegam em uma porta de entrada e os armazena, não há como determinar o fim de um dado e o início de outro se este mecanismo não for modificado.

Por esse motivo, os métodos originais das *streams* não podem ser usados com os conectores. Para cada uma das classes Stream, DataStream e ObjectStream, foi criada uma classe com funcionalidade compatível para ser usada com os conectores, são elas: Port, DataPort e ObjectPort. A tabela 6.1 mostra as *streams* originais de Java e as que foram implementadas.

Stream	Port
InputStream	InputPort
OutputStream	OutputPort
DataInputStream	DataInputPort
DataOutputStream	DataOutputPort
ObjectInputStream	ObjectInputPort
ObjectOutputStream	ObjectOutputPort

Tabela 6.1: *Streams* de Java e portas da LegoShell

A classe Port foi inicialmente especificada como subclasse de Stream. O mesmo era válido para DataPort, subclasse de DataStream e para ObjectPort, subclasse de ObjectStream. Essa seria a maneira mais intuitiva e mais simples para fornecer ao desenvolvedor de aplicações LegoShell os mesmos recursos disponíveis ao programador Java. No entanto, os métodos de DataPort e ObjectPort precisavam ser reimplementados e muitos deles são definidos como *final* em suas classes originais; isso impede que sejam reimplementados nas subclasses. Devido a essa restrição, o modelo inicial precisou ser modificado.

A estrutura de classes que foi implementada é mostrada na figura 6.2 na página 54. A hierarquia das portas de entrada e saída são quase simétricas; a única diferença é que a classe InputPort implementa também a interface Runnable enquanto que a classe OutputPort implementa apenas a interface Port.

Port é subclasse de Stream, como os construtores de DataStream e de ObjectStream exigem uma Stream como argumento, um objeto LegoShell pode usar uma porta para instanciar outras *streams*. Da mesma maneira como uma *stream* é usada em Java para criar outras *streams*. A partir de uma porta, o usuário pode criar DataPort e ObjectPort para aplicações que utilizam os conectores. Entretanto, se uma aplicação LegoShell não utiliza os recursos dos conectores, elas podem instanciar as *streams* de Java diretamente. As novas classes foram desenvolvidas para atender às exigências dos conectores mas servem também para interligar dois objetos diretamente.

Uma outra solução seria remover a classe OutputPortStream e reposicionar a classe OutputPort no lugar da classe removida. Nesse caso, a estrutura de portas ficaria mais próxima da estrutura de *streams*. No entanto, seriam necessárias algumas modificações na classe OutputPort. A estrutura atual foi escolhida porque facilita a implementação sem influenciar na maneira como o programador criará as portas. A classe OutputPortStream representa um elo de ligação entre portas e *streams*.

A classe DataPort oferece a mesma interface de DataStream e ObjectPort a mesma de ObjectStream. A diferença é que os métodos reimplementados para as portas de saída escrevem informações de controle para que os dados possam ser manipulados pelos conectores. Essas informações de controle são removidas pelo método de leitura reimplementado nas portas de entrada.

6.3.2 Unidade de dados

Uma unidade de dados pode ser um *byte*, um outro tipo primitivo qualquer ou um objeto serializado. O que define uma unidade são as operações de escrita e leitura realizadas pelos objetos. Cada operacão dessas envolve uma unidade de dados. Quando um produtor executa um método que escreve um dado, um *marcador* é inserido após o dado com o objetivo de definir uma unidade. Na operação de leitura, esse marcador é removido e o dado é restaurado e enviado a um consumidor. Portanto, entre duas unidades sempre há um marcador, conforme mostrado na figura 6.3.

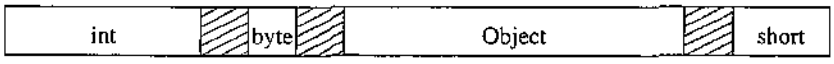


Figura 6.3: Dados separados por marcadores

Esse mecanismo funciona também quando os objetos são interligados diretamente, sem passar por um conector. Isso é importante porque um objeto deve ser implementado de modo genérico, independente da configuração que será utilizada. Por exemplo, um produtor é implementado para se comunicar com um consumidor, o protocolo de comunicação é acertado durante a implementação desses dois objetos. No entanto, apenas no momento de configuração da aplicação é que se decide por fazer a conexão direta ou pela utilização de um conector. Essas duas possibilidades são exemplificadas no exemplo 6.1.

```
Legoshell> prod = new Prod
Legoshell> cons = new Cons
Legoshell> mbox = new Mbox

Legoshell> connect prod cons  //conexão direta

Legoshell> connect prod mbox
Legoshell> connect mbox cons  //ligação via conector
```

Exemplo 6.1: Produtor e consumidor ligados diretamente e via conector

A classe Prod utiliza uma porta da classe DataOutputPort para escrever tipos primitivos seguidos de um marcador. A classe Cons utiliza uma porta de DataInputPort que faz a leitura no lado do consumidor e remove o marcador. Como os dados estão separados por um marcador, o conector mbox pode intermediar a troca de dados e processar qualquer operação sobre as unidades.

6.3.3 Caracter Separador

Caracter separador é o mecanismo que foi utilizado para implementar o marcador de unidades descrito na seção 6.3.2. O marcador não é um dado mas sim uma informação de controle. O problema é que não existe um caracter "especial" que pudesse ser usado com a finalidade única de marcar a fronteira entre dois dados. Como os dados trafegam em *bytes*, e todos os valores 0 – 255 são usados para compor uma informação, esse marcador precisa utilizar alguma outra técnica para diferenciar uma informação de controle de um dado.

O mecanismo de caracter separador permite inserir informações de controle usando um valor do próprio conjunto de caracteres usados na composição de dados. Para exemplificar, será usada a base binária onde um *bit separador* pode ser construído da seguinte maneira: o valor 1 (um) é escolhido como *bit* separador; toda vez que este valor aparecer como um dado, deverá ser duplicado. Um marcador será identificado pela seqüência 10, adicionada quando se desejar inserir uma marca. Para decodificar, remove-se um *bit* 1 toda vez que este aparecer duplicado, desse modo obtém-se a informação original. Um *bit* 1 seguido de 0 revela a existência de um marcador. Desse modo, é possível inserir uma informação de controle usando um dígito que

também é válido para codificar dados. De modo análogo, é possível inserir até 255 marcas diferentes usando-se um *byte* ou caracter separador. Caso seja necessário um número maior de informações de controle, o mesmo raciocínio pode ser empregado. Por exemplo, é possível inserir até 3 marcas com dois *bits* de controle e até $256^2 - 1$ marcas usando-se dois caracteres separadores.

Para a implementação dessa técnica, os dados precisam ser codificados de um lado e decodificados de outro. Portanto, o protocolo de comunicação precisa ser modificado. No caso particular das novas *streams*, como todas as operações de escrita passam pelo método write da classe OutputPort, é neste ponto que os dados serão codificados. No método read da classe InputPort, que é usado por todas as operações de leitura, será feita a decodificação dos *bytes* para a obtenção dos dados originais. Os métodos disponíveis para o usuário escrever dados inserem automaticamente um marcador após cada unidade de dado escrita. De modo complementar, esses marcadores serão removidos nas operações de leitura.

6.3.4 Buffer Circular

O *buffer* circular, implementado pela classe CircularBuffer e principal atributo de um conector, é uma estrutura usada para armazenar as unidades de dados. Basicamente, um conector cria *threads*, uma para cada porta, que concorrentemente acessam seu *buffer* . Também é implementado nesse *buffer*, o controle de concorrência necessário para sincronizar as operações sobre um conector.

As operações de leitura e escrita de unidades são sincronizadas, apenas uma *thread* acessa o *buffer* a cada instante. Esse mecanismo de exclusão mútua foi implementado usando os recursos de sincronização (synchronized) de Java, como explicado na seção 4.3.2.

Uma unidade de dados é armazenada seqüencialmente.

O sincronismo é construído sobre os recursos de monitores que são implementados em Java pelos métodos wait e notify. Se uma *thread* deseja escrever um dado no *buffer* e não há espaço suficiente, ela executa o método wait. O mesmo ocorre quando o *buffer* está vazio e uma *thread* deseja realizar uma operação de leitura. As *threads* que executaram wait e estão aguardando, são recolocadas na fila de execução quando uma notificação for enviada por outra *thread*, o que ocorrerá quando dados forem lidos ou escritos no *buffer* . Em resumo, quando uma *thread* tenta escrever uma unidade e não há espaço suficiente no *buffer* , ela aguarda até que uma outra *thread* realize uma operação de leitura. Se uma *thread* quer ler uma unidade mas o *buffer* está vazio, ela aguarda até que uma outra *thread* escreva dados.

Uma situação de *deadlock* poderia ocorrer caso uma *thread* começasse a escrever uma unidade e o espaço se esgotasse antes que esta operação terminasse. Nesse caso, a operação de escrita não continuaria porque o *buffer* estaria cheio e nenhuma operação de leitura seria realizada porque a operação de escrita não fora concluída. O sistema permaneceria indefinidamente nesse estado de bloqueio. No entanto, essa situação não ocorre porque antes de começar a escrever uma unidade, o espaço no *buffer* é verificado e a operação somente se inicia caso o espaço seja suficiente para armazenar integralmente a unidade. As operações de verficação do espaço e de escrita de dados são atômicas do ponto de vista de duas *threads* que concorrem pelo mesmo *buffer* .

Mesmo considerando um o escalonador onde todas as *threads* têm a mesma chance de ser executada, esse modelo implementado não seria 100% justo. Consideremos uma *thread* que deseja escrever uma unidade comparativamente grande, supondo ainda uma situação onde o espaço disponível é precário, essa *thread* terá menos chance de realizar essa operação se comparado com outra *thread* que deseja escrever um dado menor. Por exemplo, suponhamos uma situação extrema onde uma unidade a ser escrita ocuparia todo o espaço do *buffer* , essa operação somente será realizada quando o *buffer* estiver vazio. No entanto, esta situação não ocorrerá caso alguma outra *thread* seja escalonada antes e escreva um dado menor. Em resumo, teoricamente têm maior chance os dados menores, pois são estes que correm menor risco de esperar por espaço no *buffer*. Se por um lado esse modelo não é totalmente justo, uma propriedade importante é que ele privilegia o fluxo de dados pelo conector. Isso porque sempre que uma *thread* desejar escrever e houver espaço suficiente no

buffer , essa operação será permitida.

6.3.5 Pipe

A semântica do conector *pipe* é bastante simples, consiste em prover um canal de comunicação unidirecional entre dois objetos. Um produtor se conecta à porta de entrada do *pipe* e um consumidor se conecta à porta de saída. Os dados escritos pelo produtor são lidos pelo *pipe* e enviados para o consumidor.

Em termos funcionais, o sistema de suporte já implementa um conector *pipe*. Quando o usuário estabelece uma conexão através do comando `connect` da LegoShell, essa ligação pode ser entendida como a inclusão de um *pipe* entre os dois objetos.

O exemplo 6.2 mostra duas situações onde o conector *pipe* e o comando `connect` possuem a mesma função. No primeiro exemplo são interligados dois objetos e no segundo, três objetos são encadeados através de dois *pipes*.

```
p1 | p2
Legoshell> connect p1 p2


p1 | p2 | p3
Legoshell> connect p1 p2
Legoshell> connect p2 p3
```

Exemplo 6.2: Comando `connect` no lugar de um conector *pipe*

Atualmente, como os conectores ainda não implementam outras funcionalidades além de suas semânticas básicas, o uso do comando `connect` dispensaria a implementação do conector *pipe*. No entanto, um conector deve facilitar a implementação de outras operações sobre os dados que por ele trafegam. Portanto, esta implementação é importante porque facilita o acesso às unidades de dados.

Qualquer objeto LegoShell pode se conectar a um *pipe*, pois o padrão de entrada e saída é assegurado pelo modelo de portas que é implementada sobre *streams*. Qualquer tipo de dado pode ser transmitido via *pipe* entre um produtor e seu consumidor.

Embora este conector seja unidirecional e interligue apenas dois objetos, outras variações deste modelo — por exemplo, uma comunicação bidirecional envolvendo vários objetos — podem ser facilmente implementadas.

6.3.6 Estrela

O conector *estrela*, ou de *broadcast*, interliga vários objetos e sua função é disseminar os dados dos produtores para os consumidores. Um produtor envia dados para o conector que os distribui para todos os consumidores. Os dados serão recebidos pelos consumidores na mesma ordem em que foram armazenados no *buffer*. O número de objetos ligados a esse conector é configurável e as portas são criadas à medida que os objetos são interligados.

Para cada produtor ligado ao conector *estrela*, é criada uma *thread* , *WriterThread* (WT), cuja função é receber uma unidade de dados enviada pelo produtor e armazená-la no conector.

O conector *estrela* possui um *buffer* circular, seção 6.3.4, que é usado para armazenar os dados. Um produtor escreve uma unidade de dados em sua porta de saída, esse dado chega até uma porta de entrada do conector de onde é lido por uma WT que o escreve no *buffer*. O conector remove essa unidade do *buffer* e a escreve em todas as suas portas de saída, figura 6.4. Se o *buffer* estiver vazio, esta operação aguarda até que uma unidade seja produzida.

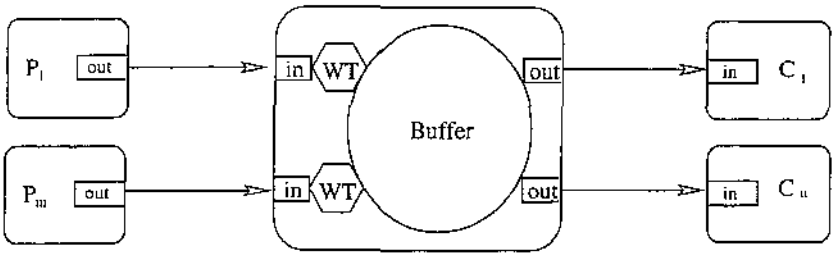


Figura 6.4: Estrutura do conector *estrela*

A operação de escrita realizada pelo conector não é sincronizada, o conector pode escrever os dados mesmo que os consumidores não estejam prontos para a correspondente leitura. Essa propriedade é implementada na classe InputStream de Java.

Há um problema quando um dos consumidores deixa de ler dados ou o faz com uma velocidade muito menor comparado com os outros consumidores. Nessa situação, se os dados continuarem sendo produzidos, o espaço no *buffer* circular poderá ser esgotado.

Basicamente, há duas alternativas de solução desse problema.

1. as operações de produção de dados podem ser suspensas visando diminuir o fluxo de dados pelo conector e permitir o acompanhamento por todos os consumidores.

2. o consumidor cujo espaço estiver esgotado, poderia descartar alguns dados. Dependendo da exigência, podem ser dispensados os dados mais recentes ou os mais antigos.

Neste conector foi implementada a primeira alternativa. Se um consumidor não lê seus dados, sua *stream* de entrada ficará lotada, isso impedirá a realização de outras operações de escrita neste consumidor. Com esse bloqueio, nenhum outro consumidor receberá dados porque o envio de dados é executado por uma única *thread* . Nesse ponto, se os produtores continuam enviando dados para o conector, o *buffer* circular acabará tendo seu espaço de armazenamento esgotado. Como conseqüência, as atividades dos produtores deixarão de ser realizadas.

O incoveniente dessa abordagem é que o fluxo de dados pelo conector é determinado pelo consumidor mais lento. No entanto, isso é uma conseqüência direta da exigência que todos os dados sejam recebidos por todos os consumidores.

A segunda alternativa que poderia ser implementada envolve o descarte de alguns dados. O consumidor mais lento, ao invés de impedir a atividade dos produtores, conseguiria acompanhar o ritmo dos outros consumidores através do descarte de alguns dados. Em algumas aplicações, pode ser mais interessante dispensar os dados mais recentes; em outras, seria mais importante o descarte dos dados mais antigos. Um exemplo desta última proposta, é a transmissão de voz ou imagem em tempo real: há uma faixa de tempo estabelecida para que os dados sejam recebidos pela aplicação, ultrapassado esse tempo, os dados podem ser dispensados porque não serão mais aproveitados.

O conector *estrela* permite que todos os consumidores leiam sempre os mesmos dados e na mesma seqüência enviada pelos produtores. A ordem considerada é aquela em que os dados chegaram até o conector. Poderia ser considerada a ordem em que os dados foram produzidos em suas aplicações de origem. Entretanto, essa funcionalidade exigiria a implementação de um mecanismo de sincronismo dos relógios das máquinas envolvidas.

6.3.7 Mbox

O conector *mbox* interliga vários produtores e vários consumidores. Uma unidade de dados enviada ao *mbox* por um produtor é entregue a apenas um consumidor.

Como foi implementado, o *mbox* pode conter dados de diferentes tipos. Conceitualmente no entanto, o *mbox* deve manipular dados de um só tipo em cada execução; não há razão para um consumidor solicitar um dado sem antes conhecer o tipo deste dado. Caso o *mbox* armazenasse diferentes tipos, um consumidor não saberia qual o tipo do dado que receberia. A semântica desse conector estabelece que o dado a ser enviado seja definido sob demanda dos conectores, portanto deve ser definido em tempo de execução.

O conector *mbox*, figura 6.5, usa uma WT(*writer thread*) com a função de ler uma unidade de dados de uma porta de entrada e armazená-la no *buffer* circular. Em resumo, os mecanismos usados para armazenar dados e para manipular a entrada do *mbox* são similares àqueles usados no conector *estrela*.

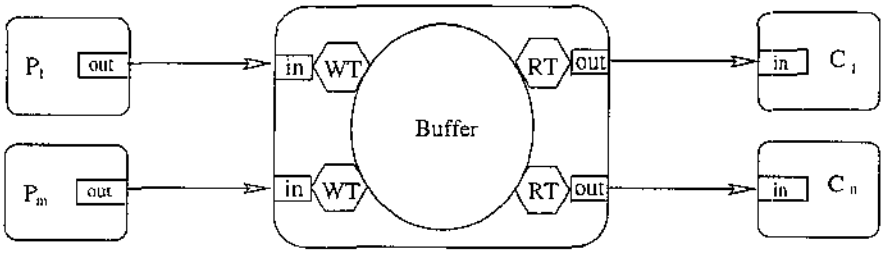


Figura 6.5: Estrutura do conector *mbox*

A saída do *mbox* é executada via RT (*reader thread*); há uma RT para cada porta de saída e sua função é ler uma unidade de dados do *buffer* circular e escrevê-la em uma porta de saída.

No entando, implementado dessa maneira, esse modelo não atenderia a ordem em que os dados foram solicitados pelos consumidores, e isso é indispensável para a semântica do *mbox*. Quando uma RT fosse escalonada pelo sistema, se houvesse dados disponíveis no *buffer* , ela removeria uma unidade e a enviaria para o consumidor ao qual está associada. Esse consumidor receberia os dados mesmo que não os tivesse solicitado. Isso porque o modelo de comunicação através de *streams* é "buferizado" e não sincronizado. O problema é determinar quando um consumidor deseja receber uma unidade de dados: no lado do consumidor, a operação de leitura da *stream* fica bloqueada caso dados não tenham sido enviados; no lado do conector, no entanto, não há como identificar aquela operação de leitura.

Como no *mbox* o dado deve ser enviado apenas sob demanda dos consumidores, a comunicação assíncrona não é apropriada.

Para sincronizar a comunicação, uma *mensagem de solicitação* de unidade é enviada do consumidor para o conector quando o primeiro desejar receber dados. Logo, o envio de dados para um consumidor é sempre precedido de uma mensagem de solicitação. A implementação dessa mensagem é bastante simples porque nesse momento, a conexão já estará estabelecida para a transmissão de dados do conector para o consumidor. Esse mesmo canal pode ser usado para enviar a mensagem de solicitação no sentido inverso porque o mecanismo de *sockets* é bidirecional. Apenas uma mensagem de controle, um *byte* de conteúdo, é enviada a cada operação de leitura.

Na implementação definitiva do *mbox*, há uma RT para cada porta de saída. Essa *thread* somente remove uma unidade do *buffer* circular depois que recebe a mensagem de solicitação. Portanto, a operação de enviar um dado para o consumidor tem início com o recebimento de uma mensagem de solicitação. Recebida essa mensagem, o conector remove a próxima unidade de seu *buffer* e a envia para o consumidor.

Em resumo, um conector é criado dentro do ambiente da LegoShell, depois as conexões são estabelecidas e por fim, o conector e as aplicações são executadas. Quando o *mbox* é executado, uma WT é executada

para cada porta de entrada e uma RT para cada porta de saída desse conector. A WT lê uma unidade de dados enviada por um produtor e a escreve no *buffer* circular. A RT, depois que recebe um pedido de dados, lê a próxima unidade do *buffer* e a envia para o consumidor que fez a solicitação.

Os dados são inseridos e retirados do *buffer* obedecendo uma fila (FIFO), portanto, serão enviados para os consumidores na ordem em que chegaram até o conector. Essa ordem, a exemplo do que ocorre com o conector *estrela*, pode não ser aquela em que os dados foram produzidos em suas aplicações de origem. Como já mencionado, um mecanismo que permitisse o consumo de dados na ordem em que foram produzidos exigiria o sincronismos dos relógios de todas as máquinas envolvidas.

Capítulo 7

Conclusão

Este último capítulo analisa os resultados alcançados e os principais fatores e decisões que influenciaram diretamente este projeto. A seção 7.1 discute a escolha da tecnologia usada para implementação do sistema. Embora nenhuma outra tecnologia atual ou sistema pudesse ser rigorosamente comparado com as funcionalidades propostas pela LegoShell, a seção 7.2 tenta situar as propriedades da LegoShell através de comparações com Voyager, RMI e CORBA. Outras importantes funcionalidades são apresentadas na seção 7.3 como sugestões para continuidade deste trabalho. Por fim, na seção 7.4 é apresentada uma avaliação crítica do trabalho desenvolvido.

7.1 Sobre a Tecnologia Utilizada

A fase inicial do projeto foi reservada para estudos, testes de ferramentas e definição final do que seria implementado. Nessa fase, a principal preocupação foi a escolha de uma das três opções: RMI, OrbixWeb e Voyager. Mesmo depois de avaliados os pontos favoráveis e as principais limitações de cada tecnologia, continuava difícil prever as dificuldades que poderiam ser encontradas durante a implementação e quão facilmente esses problemas poderiam ser contornados. Como se percebeu posteriormente, as questões mais relevantes para o desenvolvimento do projeto só puderam ser bem avaliadas e compreendidas no momento da implementação.

A primeira decisão importante foi a de implementar um sistema de suporte para construção de objetos Java ao invés de adotar a padronização CORBA. A questão da independência de linguagem permitida por CORBA é muito relevante e foi considerada. No entanto, este fator pesou menos depois que foi avaliada a possibilidade de integração futura entre objetos LegoShell e objetos CORBA, conforme discutido na seção 7.3. A opção por Java/RMI foi resultado da experiência com OrbixWeb: caso fosse utilizado um ORB, o sistema não forneceria o nível de abstração desejado e algumas dificuldades continuariam por conta do usuário final.

Usando *sockets* ou RMI, que são recursos integrados à linguagem Java, seria possível permitir que o usuário desenvolvesse classes para objetos LegoShell como o faz para outros objetos em Java. Quanto às propriedades, tanto *sockets* quanto RMI suportam serialização de objetos; RMI permite que objetos remotos se comuniquem mais facilmente; no entanto, o uso direto de *sockets* é mais eficiente e permite mais liberdade na implementação das funcionalidades do sistema de suporte. Mesmo diante de algumas incertezas, decidiu-se por começar a implementação do sistema usando *sockets* .

Depois de concluída e testada a primeira etapa desenvolvida em *sockets* , o projeto foi reavaliado. Nesse instante percebeu-se que a troca de dados poderia ser realizadas usando *sockets* e, por outro lado, RMI poderia ser usado para a localização de objetos, criação de instâncias remotas e para a abertura das conexões. Implementado desse modo, o sistema de suporte poderia usar o que há de mais favorável em cada uma das

opções. Então ficou decidido recomeçar a implementação utilizando RMI em conjunto com *sockets* .

Depois desse ponto, as principais dificuldades encontradas durante o desenvolvimento puderam ser superadas e, de modo geral, os recursos que foram usados obtiveram uma avaliação positiva.

7.2 Análise Comparativa

Vários documentos estabelecem comparações entre as tecnologias disponíveis para desenvolvimento de aplicações distribuídas. Normalmente são feitas desde análises das propriedades até considerações sobre a arquitetura e os detalhes de concepção de cada uma das tecnologias. No entanto, não é esse o objetivo desta seção: é difícil, talvez injusto, estabelecer comparações entre a LegoShell e as outras tecnologias. A LegoShell é uma proposta ainda em desenvolvimento; além disso, é basicamente uma alternativa a algumas limitações existentes nas demais tecnologias.

O único objetivo da tabela 7.1 é permitir que o sistema desenvolvido neste projeto seja avaliado dentro do contexto das tecnologias já conhecidas. Foram escolhidas algumas propriedades de CORBA, RMI, Voyager e as principais vantagens da LegoShell.

PROPRIEDADES	CORBA	LegoShell	RMI	Voyager
Independência de linguagem	sim	não	não	não
Criação e manipulação de objetos	difícil	fácil	razoável	fácil
Configuração dinâmica	não	sim	não	não
Execução por demanda	sim	sim	não	não
Integração com a linguagem	não há	parcial	total	total
Invocação dinâmica	sim	não	não	não
Padrão de comunicação	não há	conectores	não há	não há

Tabela 7.1: Quadro comparativo: CORBA, LegoShell, RMI e Voyager

As propriedades que foram avaliadas são comentada a seguir:

Independência de linguagem : apenas aplicações CORBA permitem interação entre clientes e servidores implementados em linguagens distintas.

Criação e manipulação de objetos : avalia o grau de dificuldade com que objetos distribuídos podem ser criados, interligados e executados. Esse processo é mais complexo em CORBA e bastante simples tanto em Voyager quanto na LegoShell.

Configuração dinâmica : apenas a LegoShell permite reconfiguração dos objetos depois que eles foram instanciados.

Execução por demanda : esta propriedade, existente em CORBA e na LegoShell, permite que um servidor seja executado apenas quando solicitado por um cliente. É suficiente que seu código esteja disponível na máquina servidora.

Integração com a linguagem : se o desenvolvedor necessita ou não utilizar outras sintaxes ou ambientes para escrever suas aplicações. Em CORBA é necessário codificar a interface em IDL e usar outro ambiente (ORB) para criar as aplicações. Em LegoShell os objetos são criados usando Java mas são manipulados usando uma interface de comandos. Em RMI e Voyager todo o processo é desenvolvido em Java; há portanto, completa integração com a linguagem.

Invocação dinâmica : CORBA define funcionalidades para que serviços sejam descobertos dinamicamente. Deve haver recursos para que uma chamada de método possa ser montada em tempo de execução.

Padrão de comunicação : em CORBA, Voyager e RMI os canais de comunicação devem ser criados pelo desenvolvedor; na LegoShell podem ser usados os conectores que possuem funcionalidades diversas e padronizam o processo de comunicação.

As principais vantagens da LegoShell podem ser resumidas na facilidade para criação de objetos remotos, na capacidade de interconexão dinâmica desses objetos e no conceito dos conectores como padrão de comunicação.

7.3 Proposta de Continuidade

Nesta versão inicial algumas funcionalidades da proposta original da LegoShell não foram implementadas; outras propriedades foram implementadas mas merecem uma solução mais elaborada ou mais eficiente. Esta seção discute essas melhorias como sugestões para dar continuidade a este projeto.

Interface gráfica : idealmente, o ambiente LegoShell deveria ser gráfico, conforme proposta inicial da LegoShell [Dru95]. Nesse caso, uma vez que os objetos tenham sido desenvolvidos e disponibilizados, todas as operações de instanciação, configuração e execução poderiam ser realizadas com alto nível de abstração.

Chamada de métodos : um objeto LegoShell implementa suas funcionalidades dentro de um método exec. Atualmente, esse é o único método que pode ser executado quando um objeto é instanciado remotamente. O próximo passo é permitir que outros métodos possam ser executados, essa funcionalidade necessitaria de recursos para a passagem e o recebimento de argumentos.

Integração com CORBA : atualmente, objetos LegoShell não se comunicam com objetos CORBA. Os objetos LegoShell alcançariam independência de linguagens através de um *gateway* implementado para integrá-los a CORBA. Os recursos de invocação dinâmica de *skeleton* (DSI) e de invocação dinâmica de interface (DII) de CORBA podem ser usados na implementação dessa funcionalidade. Com isso, clientes LegoShell poderão se comunicar com servidores CORBA e servidores LegoShell poderão atender chamadas de clientes CORBA.

Interação entre aplicações remotas : atualmente, uma aplicação LegoShell é formada por objetos instanciados de qualquer ponto da rede. Um avanço seria tornar essas referências remotas visíveis também fora da aplicação a que pertencem. Desse modo, seria possível implementar aplicações não apenas pela instanciação de objetos remotos mas também pela utilização de objetos já instanciados e configurados em outras aplicações remotas.

Eficiência : onde atualmente está sendo usado o *buffer* circular, seção 6.3.4, poderia ser usada uma estrutura mais eficiente. Cabe observar que o mecanismo de caracter de *escape* implica em um *overhead* de aproximadamente duas operações de comparação para cada *byte* transmitido. É realizada uma comparação na escrita e uma ou mais na leitura, dependendo do número de caracteres de *escape*.

Controle sobre a execução : atualmente, a execução de uma aplicação não pode ser controlada pelo usuário. Poderia ser implementado um recurso que permitisse ao usuário interagir com sua aplicação para controlar a execução ou reconfigurar os objetos. Para implementar essa funcionalidade, seria necessário controlar a execução das *threads* que já foram executadas para os objetos. Com esse mecanismo, seria possível também a inserção de produtores e consumidores em uma aplicação que já iniciou a execução.

Funções de gerenciamento : uma das principais vantagens do uso de conectores é a padronização dos mecanismos de comunicação entre objetos distribuídos. Desse modo, fica mais simples implementar funções de gerenciamento como as descritas a seguir:

- gerenciamento do número de mensagens e tipos dos dados trocados entre os objetos.
- controle de congestionamento e balanceamento de carga com a utilização das rotas baseando-se no tempo de resposta de cada conexão.
- monitoramento do canal de transmissão e do estado em que se encontra a execução de uma aplicação remota.

Tolerância a falhas : como foram implementados, tanto o sistema de suporte da LegoShell quanto os objetos não resistem a eventuais falhas nos equipamentos onde estão sendo executados. Vários recursos de tolerância a falhas podem ser implementados para aumentar a confiabilidade do sistema.

Aplicações persistentes : recursos de persistência de objetos farão parte de JDK 1.2 que ainda se encontra em fase de testes. Esses recursos são importantes para que aplicações LegoShell possam ser armazenadas em dispositivo permanente mesmo depois de configuradas. Desse modo, uma execução poderia ser interrompida, armazenada e reiniciada em outro momento ou em outra máquina.

Conversão Java para LegoShell : uma propriedade mais elaborada seria gerar, automaticamente, objetos LegoShell a partir do código fonte de uma classe Java.

7.4 Avaliação

Com relação ao andamento do projeto, gastou-se mais tempo do que era devido na fase inicial, antes que o projeto começasse a ser implementado. Muitas questões foram levantadas e soluções foram buscadas sem efetivavente iniciar o trabalho de desenvolvimento. Isso ocasionou atraso no cronograma e algumas dificuldades para a conclusão deste trabalho.

Embora a primeira iniciativa com *sockets* não tenha sido concluída, a tentativa foi válida porque ajudou a vislumbrar novos rumos para o projeto e para a escolha definitiva dos recursos a serem usados. Depois da primeira tentativa, decisões mais objetivas puderam ser tomadas.

De modo geral, as funcionalidades inicialmente planejadas foram implementadas. Outras idéias surgiram durante a implementação do sistema e algumas — por exemplo, a separação de ambientes em contextos e a execução de objetos sob demanda — não faziam parte do projeto inicial e foram acrescentadas.

Basicamente, este projeto apresenta facilidades para a criação de objetos remotos, permite configuração dinâmica além de apresentar os conectores. As propostas aqui apresentas podem ser incorporadas a outras tecnologias da área de objetos distribuídos. Atualmente, existe grande necessidade de mecanismos que ofereceçam mais recursos de abstração para o desenvolvedor.

Muitas otimizações poderiam ser feitas na implementação, como sugerido na seção 7.3; contudo, o estágio em que se encontra o sistema foi considerado satisfatório e alcançou os objetivos deste projeto. As principais questões pré-existentes, assim como outras que surgiram durante o projeto, puderam ser respondidas. Desse modo, houve uma avaliação bastante positiva quanto a viabilidade e a importância das funcionalidades aqui apresentadas.

Referências Bibliográficas

[Boo91] Grady Booch. *Object Oriented Design with Application.* Benjamin/Cummings, Redwood City, 1991.

[CH96] Gary Cornell and Cay S. Horstmann. *Core Java.* SunSoft Press, 2550 Garcia Avenue, Mountain View, CA 94043-1100, USA, April 1996. Includes CD-ROM.

[Com92] Douglas. E. Comer. *Internetworking with TCP/IP: Principles, Protocols and Architecture.* PH, 1992.

[CW98] Mary Campione and Kathy Walrath. *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet.* The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[DC98] David Diskin and Sherrie Chubin. Recommendations for using dcs, dcom, and corba middleware. *DII COE Destributed Applications Series*, April 1998.

[dO97] Aredis Sebastião de Oliveira. Tese de mestrado. Master's thesis, Unicamp - Universidade Estadual de Campinas, IC – Instituto de Computação, novembro 1997.

[Dru95] Rogerio Drummond. Legoshell++. Technical report, UNICAMP, outubro 1995.

[Fla97] David Flanagan. *Java in a Nutshell: a desktop quick reference.* A Nutshell handbook. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, second edition, 1997.

[Fur91] Carlos Alberto Furuti. Um compilador para uma linguagem de programação orientada a objetos. Master's thesis, Unicamp – Universidade Estadual de Campinas, IC – Instituto de Computação, julho 1991.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java^{TM} Language Specification.* Sun Microsystems, 1.0 edition, August 1996. appeared also as book with same title in Addison-Wesleys 'The Java Series'.

[Ing78] Daniel H. H. Ingalls. The smalltalk-76 programming system design and implementation. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona*, pages 9–16. ACM, January 1978.

[Jr.94] Celso Gonçalves Jr. Tese de mestrado. Master's thesis, Unicamp - Universidade Estadual de Campinas, IC – Instituto de Computação, agosto 1994.

[Lea96] Douglas Lea. *Concurrent programming in Java: design principles and patterns.* Addison/Wesley Java series Java series. Addison-Wesley, Reading, MA, USA, November 1996.

[LMS82] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. Technical Report 60, SRI International, Computer Science Laboratory, March 1982.

[Obj97] Objectspace. *Voyager, user's guide*, july 1997.

[OH98] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA*. John Wiley and Sons, New York, NY, USA; London, UK; Sydney, Australia, second edition, January 1998.

[OHE96] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & sons inc., 1996.

[OW97] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1997.

[PS91] J. Peterson and A. Silberschatz. *Operating Systems Concepts*. Addison-Wesley, Reading, MA, 3rd edition, 1991.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall Inc., 1991. ISBN 0-13-630054-5.

[Sta95] William Stallings. *Network and Internetwork Security*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1995. Provides an overview of security threats mechanisms and services. Details security principals such as conventional encryption and confidentiality, public-key cryptology, authentication and digital signatures, viruses and worms.

[Str97] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, New Jersey, 1992.

[Tec97a] IONA Tecnologies. *OrbixWeb Programer's Guide*, 1997.

[Tec97b] IONA Tecnologies. *OrbixWeb Reference Guide*, 1997.

[Tel93] Alexandre Prado Teles. A linguagem de programação cm(versao 2.0x). Master's thesis, Unicamp – Universidade Estadual de Campinas, IC – Instituto de Computacao, dezembro 1993.

[VD97] Andreas Vogel and Keith Duddy. *Java Programming With CORBA*. John Wiley and Sons, New York, NY, USA; London, UK; Sydney, Australia, March 1997.

Índice Remissivo

Sobre a Preparação deste Documento

Este projeto utilizou um equipamento PC com processador AMD K6 de 200 MHz e 64 MB de memória[1]. O sistema operacional usado foi o Linux[2] 2.0.34.

O editor de texto Emacs[3], versão 20.2, foi usado na produção de código fonte Java. Com auxílio do ambiente AUC TeX, o Emacs foi usado também na preparação deste documento para LaTeX 2_ε[4].

A interface LegoShell foi desenvolvida com o gerador de *parser* JavaCC[5] versão 0.8pre2; as demais classes LegoShell foram implementadas usando JDK[6] 1.1.5 (posteriormente atualizada para JDK 1.1.6). As figuras foram diagramadas com o xfig[7] 3.2 e exportadas para o formato eps antes de serem inseridas no documento. O programa BibTeX versão 0.99c foi usado para a geração das referências bibliográficas e o índice remissivo foi gerado com o MakeIndex 2.12. O mktable[8] 1.4 auxiliou na construção das tabelas. Para a visualização do documento foram usados o xdvi 5.58f e o gv 3.5.8.

Nenhuma licença de uso de *software* foi violado durante o desenvolvemento deste projeto. Dos programas utilizados, JDK e JavaCC são disponibilizados pela Sun Microsystems; todos os demais são *free software*[9].

---

[1] Equipamento adquirido com recursos concedidos pela FAPESP, processo número 97/04245-3

[2] `http://www.linux.org`

[3] `ftp://prep.ai.mit.edu/pub/gnu/`

[4] `http://www.tug.org`

[5] `http://www.suntest.com/JavaCC/`

[6] `http://java.sun.com/`

[7] `ftp://ftp.x.org/contrib/applications/drawing_tools/xfig/`

[8] `http://www.ahand.unicamp.br/jessen/LaTeX/tools/mktable/`

[9] `http://www.gnu.org/philosophy/free-sw.html`