# Objetos Distribuídos em Cm

Rogério Drummond e Celso Gonçalves Jr.
Projeto A_HAND
DCC — IMECC — UNICAMP
CEP 13081-970 Campinas/SP
e-mail {drummond,celso}@dcc.unicamp.br

14 de janeiro de 1994

**Sumário:** *Este artigo descreve o modelo de Objetos Distribuídos, trabalho sendo desenvolvido pelo Projeto A_HAND nas áreas de Linguagens de Programação e Sistemas Distribuídos. O objetivo dessa linha de pesquisa é oferecer recursos para construção de aplicações distribuídas de forma transparente e sistemática. O modelo é implementado sobre plataforma UNIX e linguagens de programação desenvolvidas pelo Projeto.*

**Abstract:** *We describe the work under development at Projeto A_HAND towards a model of distributed object-oriented programming. Our research effort covers related aspects of programming languages and distributed systems, aiming to provide generic, efficient support for distributed programming. This model is to be implemented on top of UNIX system and programming languages developed by Projeto A_HAND.*

Uma importante tendência da computação hoje é a confirmação dos sistemas distribuídos como ambiente para processamento de informações. A disseminação das redes locais de computadores (LANs) vem estimulando a construção de aplicações distribuídas, assim chamadas por aproveitar a disponibilidade de recursos computacionais e meios de comunicação. O projeto de aplicações distribuídas é estimulado pelas vantagens do novo ambiente, como relação custo/benefício decrescente, flexibilidade, extensibilidade e tolerância a falhas.

A construção de aplicações distribuídas trouxe novas demandas para o projeto e implementação de linguagens de programação. O desenvolvimento dessas aplicações depende de novas linguagens, que permitam a construção de aplicações bem estruturadas, robustas e eficientes. O paradigma de objetos é um interessante ponto de partida para essas exigências, já que estimula o desenvolvimento de aplicações como um conjunto de partes autônomas, com dados e funcionalidade encapsulados, comunicando-se através de troca de mensagens.

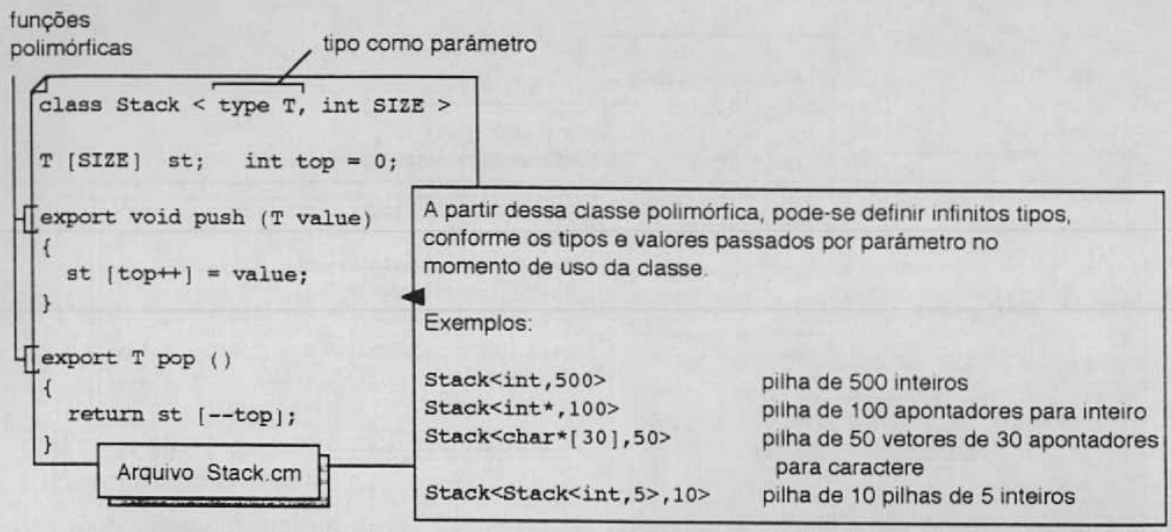
As pesquisas deram origem a linguagens inteiramente novas, como Argus [Lis88], Emerald [Raj91] e linguagens baseadas em *actors* [Agh90], e à adaptação de linguagens orientadas a objetos já existentes, como Smalltalk [Gol83, Yok87], C++ [Str91, Lea93] e Eiffel [Mey93]. Neste artigo descreveremos o modelo de Objetos Distribuídos, implementado na linguagem Cm, a linguagem de programação básica do ambiente A_HAND [Dru87a, Dru87b]. O modelo de Objetos Distribuídos será o bloco básico de construção das aplicações distribuídas no A_HAND.

## A linguagem Cm

A linguagem Cm [Dru88, Sil88, Fur91, Tel93] é derivada da linguagem C [Ker78] com respeito a comandos, operadores e expressões, adicionando mecanismos que objetivam facilitar a produção de grandes programas. O conceito essencial de Cm é o tipo classe, que introduz na linguagem suporte a programação modular, abstração de dados e programação orientada a objetos. As principais características de Cm são:

- verificação rigorosa e uniformidade de tipos
- *overloading* de operadores e funções
- polimorfismo paramétrico
- herança múltipla
- tratamento de exceções

**FIGURA 1.** Exemplo de classe polimórfica




Através do mecanismo de polimorfismo paramétrico é possível escrever classes genéricas, usadas para gerar outras classes. Essas classes são chamadas de meta-classes ou classes polimórficas. Um exemplo de classe polimórfica é mostrado na figura 1.

Uma classe Cm pode ser compilada para gerar um programa executável, como no exemplo mostrado na figura 2.

Uma classe também pode importar classes, e dessa maneira declarar objetos cujo tipo é uma classe importada. Tais objetos podem ser usados através dos componentes da sua interface (ítens da classe declarados com `export`). No exemplo da figura 3, a classe `UseHello` importa a classe `Hello`, e declara o objeto `aHello` como sendo desta classe. Dentro da classe `Hello`, o método `print()` é declarado como exportável, portanto pode ser chamado de dentro da classe `UseHello`, por expressões como `aHello.print()`.

Um programa complexo em Cm pode ser a implementação de uma hierarquia complexa de classes; podemos expressar isso graficamente, através de um diagrama de classes. Esses diagramas são usados para mostrar as relações de dependência existentes entre as classes envolvidas. Representa-se classes por triângulos, e relação de herança e importação por uma seta apontando para cima e para baixo, respectivamente. Dentro de uma classe, os tipos, variáveis e funções podem ser declarados como sendo exportáveis, com o que passam a fazer parte da interface da classe e podem ser utilizados externamente. Um exemplo de diagrama de classes está na figura 4.

**FIGURA 2.** Classe Hello sendo usada como programa

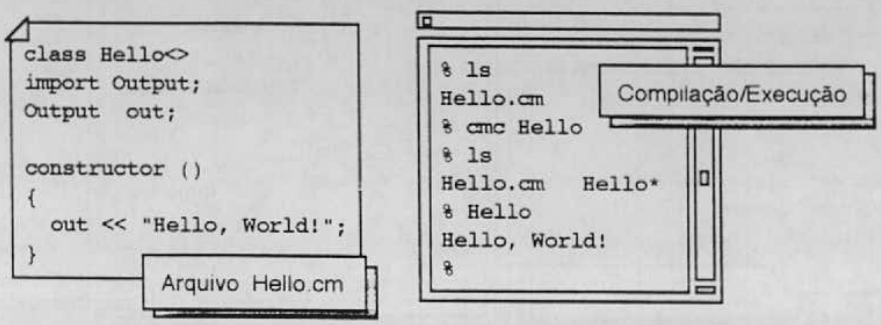
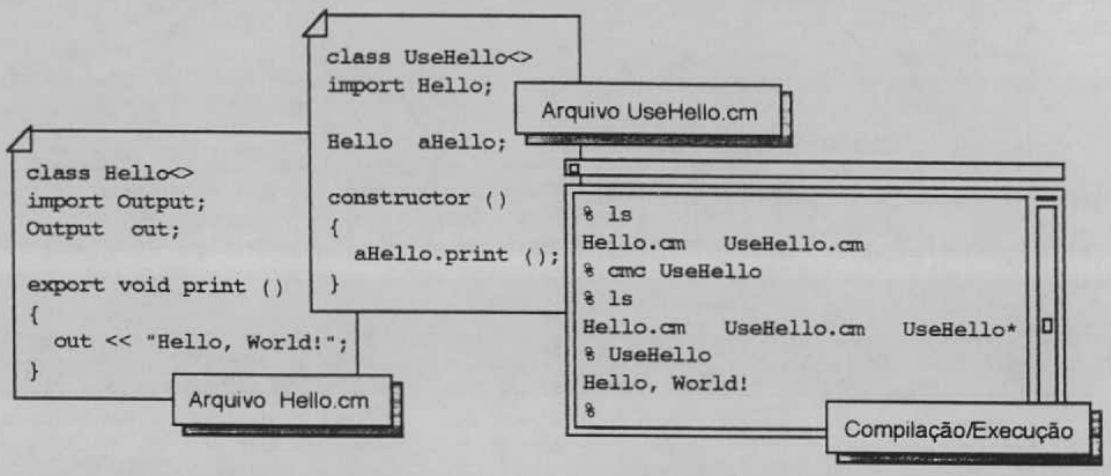


**FIGURA 3.** Objeto da classe Hello usado como variável




Há uma outra representação, própria para mostrar objetos durante sua configuração e execução. Nesse exemplo, mostrado na figura 5, vamos usar a classe `UseHello` mostrada na figura 2.

Essa representação será usada pela linguagem LegoShell [Dru89] para editar programas e para acompanhar a execução destes. Objetos são representados por uma "caixinha", rotulada por seu nome. À esquerda do nome há um botão que, quando apertado, converte o objeto em um ícone. À direita do nome há dois botões, que servem para iniciar/encerrar e suspender/continuar a execução do objeto. Abaixo do nome há um botão de nome `Par`, que serve para especificar parâmetros de configuração e execução. Arquivos e dispositivos periféricos têm a mesma representação, porque também são considerados objetos.

## O Sistema OMNI

O OMNI [Dru92] é um sistema de suporte à construção de aplicações distribuídas desenvolvido pelo Projeto A_HAND. Através dele é possível escrever programas que se comunicam transparentemente em uma rede de computadores heterogêneos que utilizam UNIX [Ker78] como sistema operacional. A funcionalidade do sistema OMNI é bem geral, e seus recursos serão utilizados pelas linguagens de programação e ferramentas de *groupware* do nosso ambiente.

**FIGURA 4.** Diagrama de classe representando herança e importação

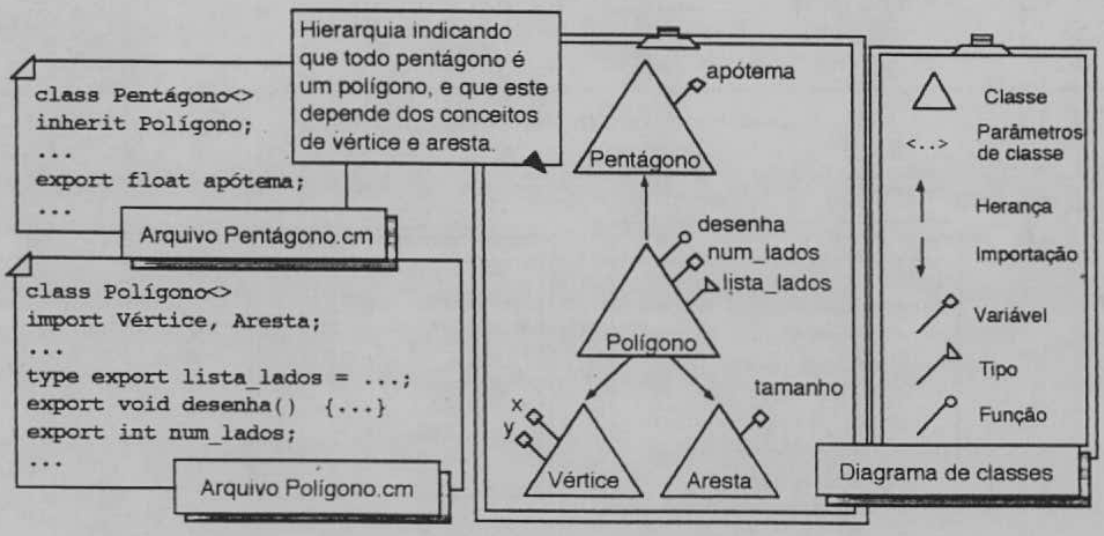
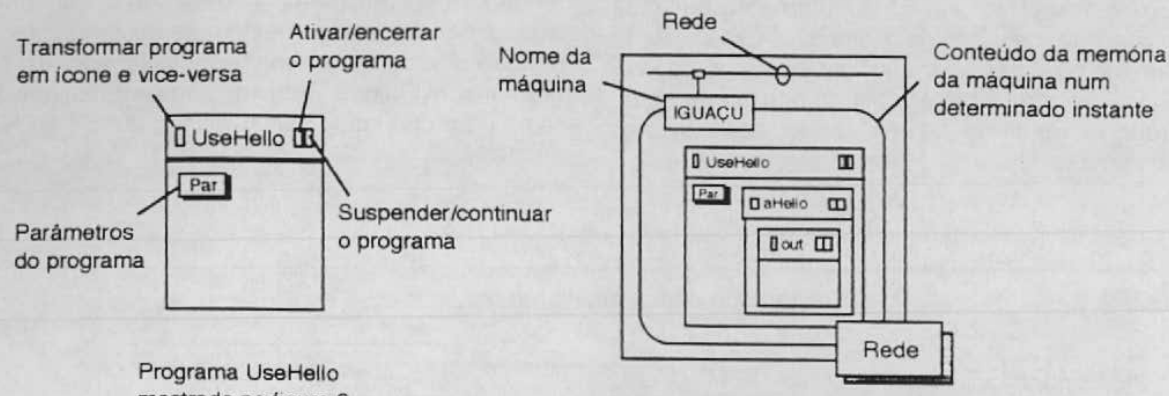


**FIGURA 5.** Programas em execução




O sistema OMNI é constituído de uma biblioteca de funções escritas em linguagem C e de um conjunto de programas (*daemons*) que executam simultaneamente nas várias máquinas da rede. O OMNI é subdividido, basicamente, em três partes: Servidor de Nomes, Gerenciador de Processos e Portas de Comunicação.

O Servidor de Nomes é responsável por fornecer uma identidade a todos os objetos criados pelo OMNI. Essa identidade, chamada OMNIid, é única no espaço e no tempo, sendo usada para acessos transparentes aos objetos do OMNI. O Servidor de Nomes permite associar nomes simbólicos aos objetos por ele manipulados, e a criação de espaços de nomes independentes (contextos).

O gerenciamento de processos é feito através de funções que permitem criação de processos e envio de sinais de forma distribuída. A função `om_createProc()` cria um processo em uma máquina qualquer da rede e devolve como resultado a OMNIid do processo criado. Outras funções comportam-se como equivalentes distribuídos de funções do UNIX: a função `om_kill()`, por exemplo, envia um sinal para um processo, usando como parâmetro uma OMNIid ao invés de um Pid.

As portas de comunicação são estruturas gerenciadas pelo OMNI para comunicação entre processos distribuídos. Elas dividem-se em portas conectáveis e não-conectáveis, se dependem ou não de conexão para seu uso. O OMNI fornece funções para criação e destruição de portas, envio e recebimento de mensagens, conexão e desconexão, etc. As portas conectáveis são ligadas para estabelecer um canal por onde trafegarão dados produzidos e consumidos nos processos donos das portas; já as portas não conectáveis são usadas para fornecer cooperação entre processos no esquema cliente-servidor. O uso de portas de comunicação dentro da linguagem Cm está ilustrado na subseção "Portas de Comunicação".

## Cm Distribuído

A programação distribuída usando Cm será possível através da inclusão, na linguagem, de mecanismos para criação de objetos distribuídos e comunicação transparente entre eles. Esses mecanismos são elaborações dos recursos básicos do OMNI, para utilização dentro do paradigma de orientação a objetos. A interface da linguagem Cm com o OMNI é feita através de um *Run Time System* [Dru93].

O "Cm Distribuído" é uma nova linguagem na medida em que será usada para construir programas compostos por objetos que se comunicam através de uma rede de computadores. A partir da versão atual da linguagem Cm [Tel93], será adicionada uma série de novas funcionalidades:

- criação de objetos distribuídos
- comunicação transparente entre objetos
- controle de concorrência
- portas simples e portas tipadas[1]
- exceções inter-processos

1. Tradução livre do termo inglês *typed ports*.

## Objetos Remotos

A programação em Cm Distribuído está baseada no conceito de Objetos Remotos. Essa é uma idéia que consiste em trazer, para o escopo da linguagem de programação Cm, o conceito de transparência, fundamental na área de Sistemas Distribuídos.

Ao usar o termo transparência, queremos dizer que: objetos utilizam recursos e interagem com outros objetos transparentemente, isto é, sem precisar saber, necessariamente, que esses recursos e esses objetos estão espalhados pela rede; e programar com Objetos Remotos não será muito diferente da programação seqüencial que hoje se faz sem eles.

**FIGURA 6.** Declaração de um objeto remoto

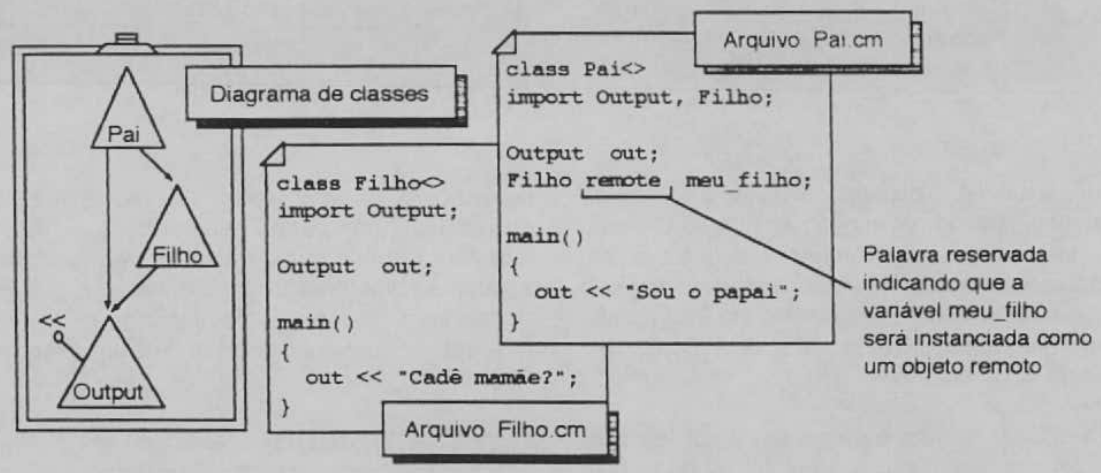
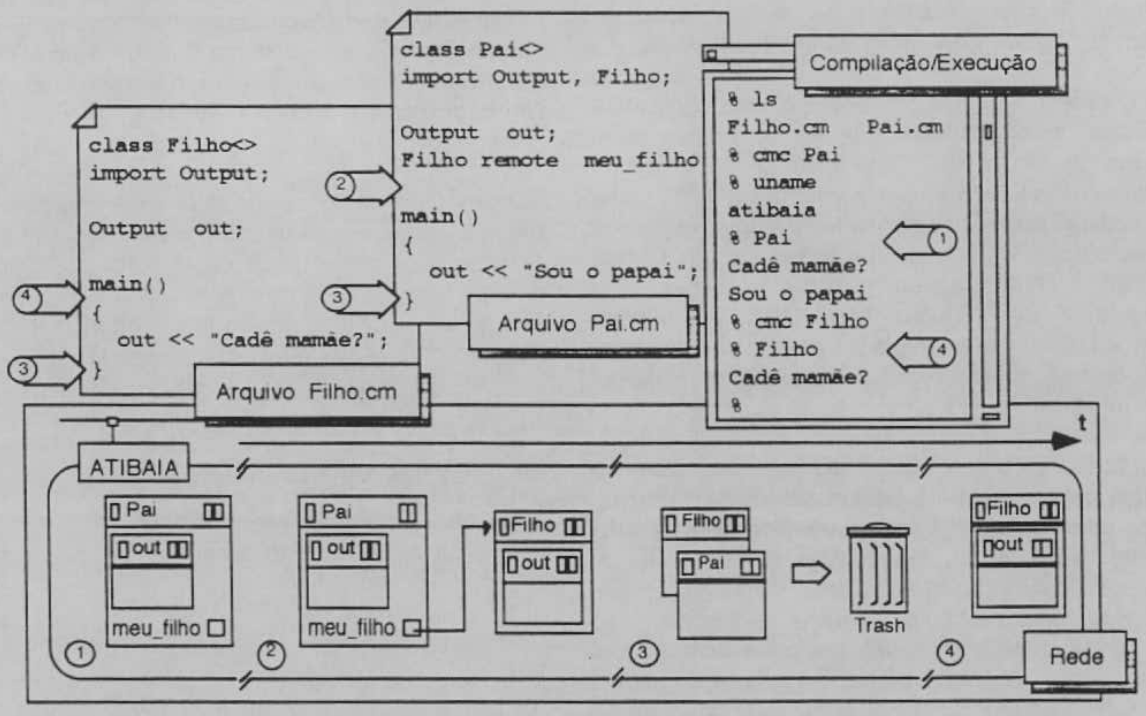



**FIGURA 7.** Execução de um objeto remoto



Em uma linguagem seqüencial orientada a objetos, como Smalltalk ou C++, os objetos estão a espera de uma mensagem para executar alguma operação. Num dado momento, apenas um objeto está executando, enquanto que o objeto que pediu essa operação está esperando pelo seu término; após o que este objeto vai continuar executando uma operação pedida por outro objeto, que por sua vez está bloqueado, e assim por diante.

Com objetos remotos, uma aplicação passa a ser um conjunto de objetos que podem estar executando operações simultaneamente. Um objeto é dito remoto se ele está sendo executado por um outro "processador", isto é, fora do contexto de execução do objeto que o criou. Isso tem duas conseqüências importantes: um objeto remoto pode ser executado em paralelo com o seu criador; e um objeto remoto pode ser visto por outros objetos, já que está "fora" do objeto que o criou.

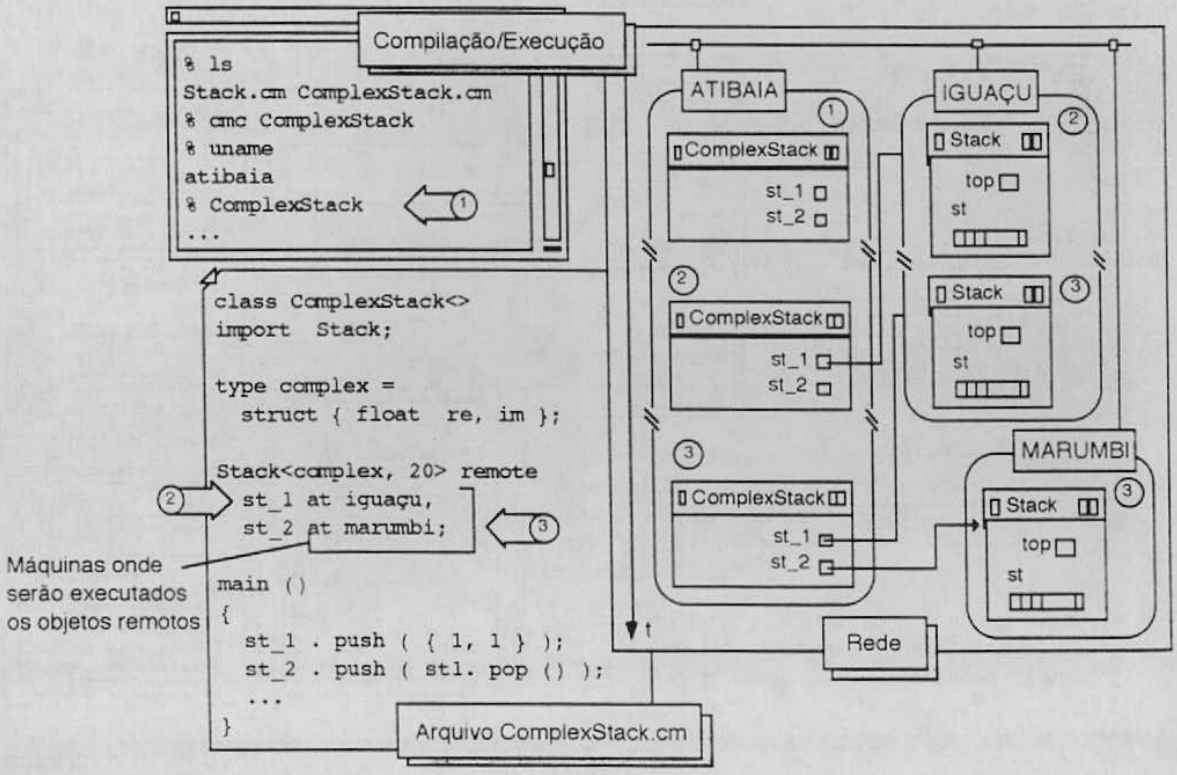
Como mostrado na figura 6, na classe `Pai` a variável `meu_filho` é declarada como um objeto remoto da classe `Filho`. Isso é feito através da palavra reservada `remote`, um construtor de tipos que se aplica somente a tipos classe. Um objeto remoto tem, do ponto de vista semântico, um tipo diferente de um objeto comum; portanto, eles não são compatíveis entre si, embora sejam da mesma classe.

Tomando esse exemplo, definimos um objeto da classe `Pai` como sendo um "objeto pai", sendo que o objeto da classe `Filho` é chamado de "objeto filho". Essa relação é semelhante à relação entre processos pai e filho do UNIX.

Na figura 7, é mostrada a fase de execução. As setas numeradas indicam os eventos importantes na ordem em que acontecem. No instante nº 2, por exemplo, o objeto `Pai` "executa" a declaração do objeto remoto `meu_filho`; isso causa a criação de um objeto filho e o começo da execução, em paralelo, do método `main()` deste.

Uma vez que o objeto remoto esteja criado, ele pode ser usado da mesma maneira que um objeto "comum", não remoto, através de sua interface. Na figura 8 é mostrada a ativação de métodos de objetos remotos, usando a classe `Stack` definida na figura 1.

**FIGURA 8.** Uso de métodos de objetos remotos



A chamada de métodos de objetos remotos é sempre síncrona, bloqueando quem fez a chamada até que venha uma resposta do objeto remoto ou seja detectado um *time-out* pelo sistema de suporte a execução. Mesmo quando o método chamado não retorna nenhum valor, o chamador é bloqueado, pois o objeto remoto sempre pode ativar uma exceção (vide subseção "Exceções")

A maneira como os métodos de objetos remotos são chamados — sincronamente — deve-se basicamente ao princípio de transparência. O resultado da ativação de um método sempre é conhecido após o término da chamada. Essa não é uma decisão de projeto fácil, já que a chamada assíncrona é um mecanismo mais geral. Entretanto, seria necessário acrescentar à linguagem algum mecanismo que permitisse esse tipo de chamada.

O uso de chamadas síncronas não representa, necessariamente, uma perda no grau de concorrência entre objetos. Isso porque, como se verá mais adiante (subseção "Concorrência"), dentro de um objeto pode haver múltiplos fluxos de controle, isto é, a linguagem provê concorrência internamente aos objetos. Dessa forma, quando um objeto faz uma chamada a um método de um objeto remoto, é apenas o fluxo de controle responsável pela chamada que fica bloqueado; os demais podem estar ativos.

## Modelo Cliente-Servidor

Programação no modelo cliente-servidor é feita através das facilidades oferecidas pelo Servidor de Nomes do OMNI. Qualquer classe que declara métodos exportados é a implementação de um "servidor", e qualquer classe que utiliza métodos exportados por outras é a implementação de um "cliente". O pai de um objeto remoto é um cliente natural deste, ou o único. Um objeto remoto, todavia, pode ser visto "fora" do objeto pai, portanto pode ser compartilhado por outros objetos, desde que estes objetos tenham alguma referência para esse objeto filho. O Servidor de Nomes do OMNI pode ajudar a associação entre clientes e servidores possibilitando que determinados objetos possam ser conhecidos localmente ou globalmente, a partir de um nome simbólico.

**FIGURA 9.** Instalando um servidor conhecido globalmente

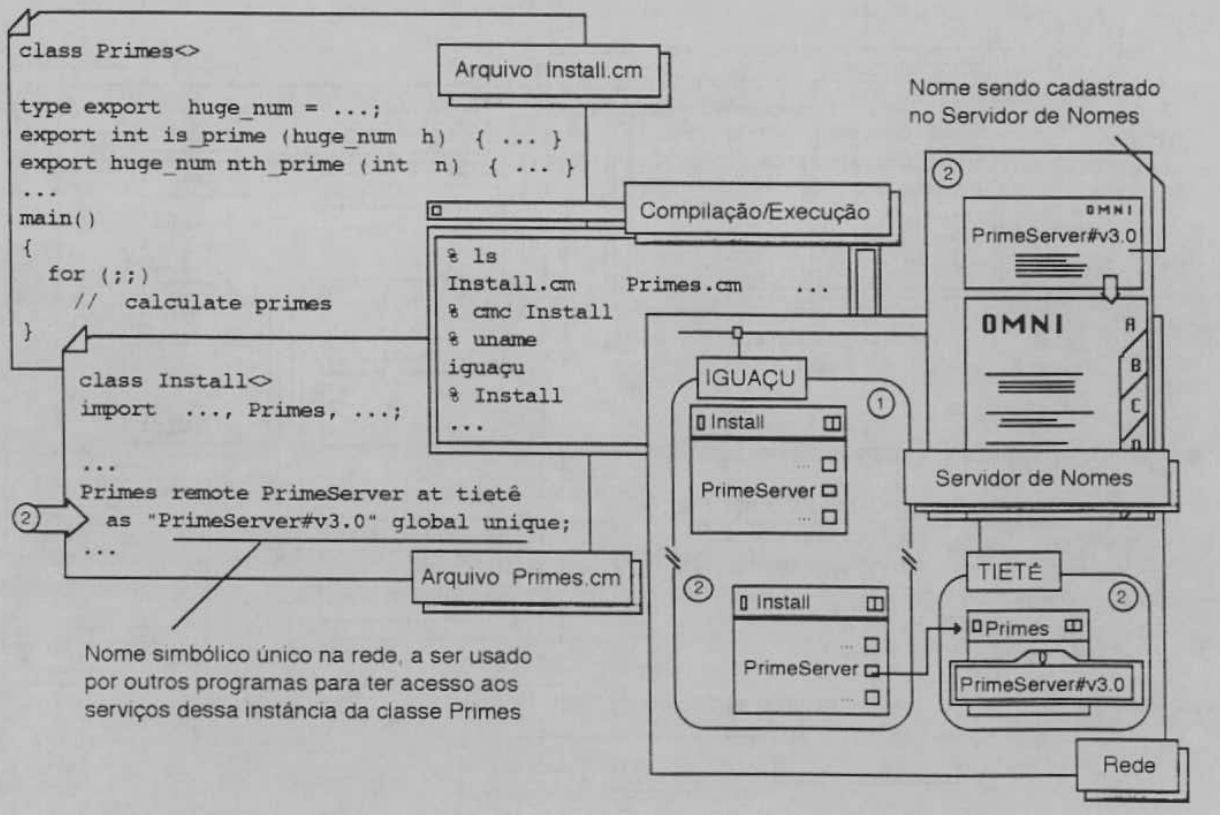
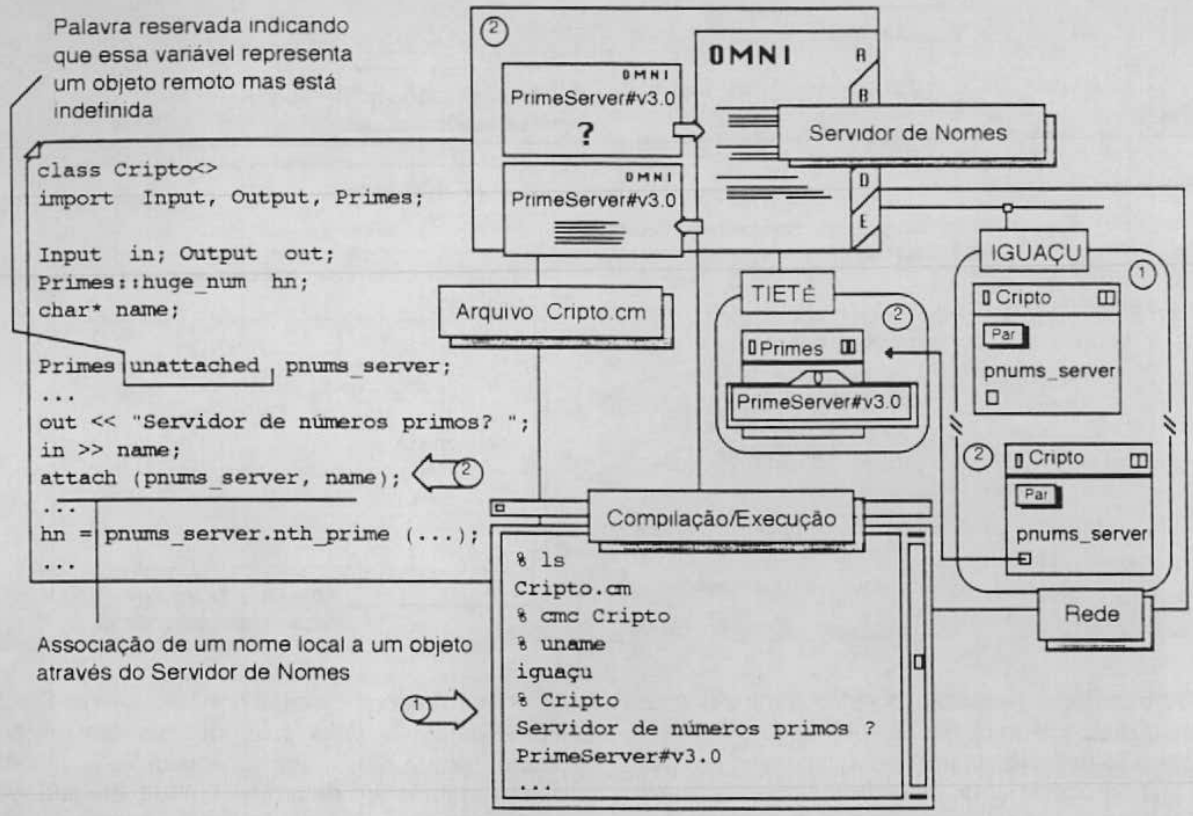


**FIGURA 10.** Uso de um servidor conhecido globalmente




Como se vê na figura 9, na classe `Install`, um objeto remoto da classe `Primes` é criado na máquina tietê, sendo associado a ele o nome simbólico "PrimeServer#v3.0". Pelas palavras reservadas `global` e `unique`, é indicado que esse nome simbólico tem escopo global (é conhecido em toda a rede OMNI) e é único: uma tentativa de cadastrar outro objeto com esse nome simbólico será rejeitada pelo OMNI.

Os programas que quiserem usar os serviços providos pelo servidor cadastrado como "PrimeServer#v3.0" devem declarar uma variável, inicialmente indefinida, da classe `Primes`, e depois associar essa variável ao objeto retornado por uma consulta ao Servidor de Nomes. Depois disso, a variável representa um objeto remoto válido, o qual pode ser usado normalmente.

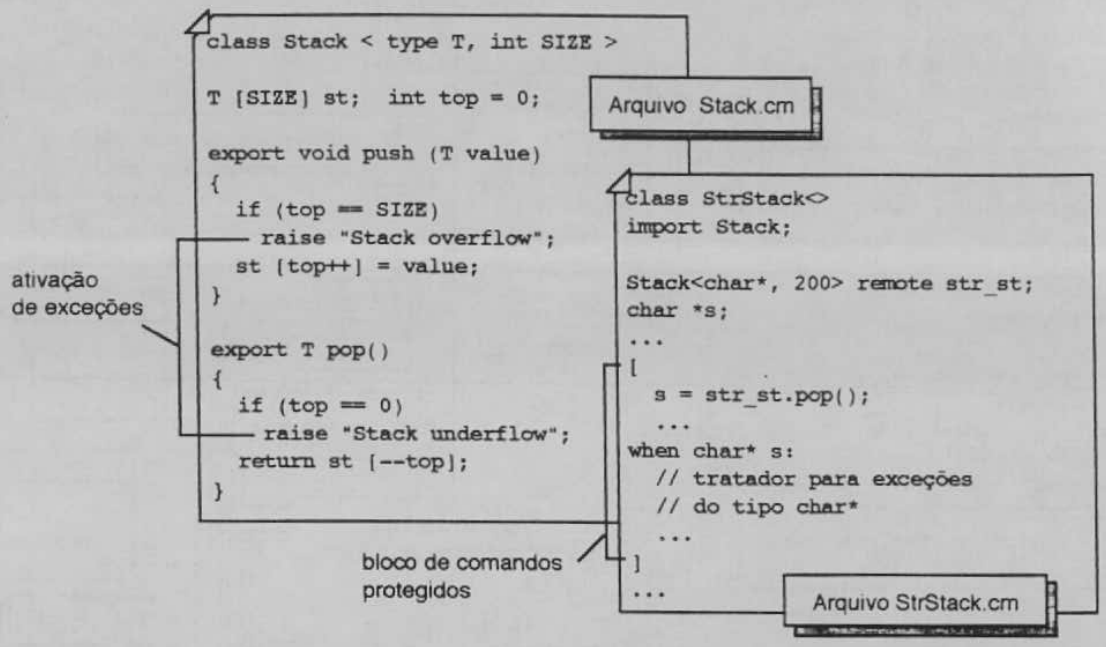
Este procedimento é mostrado na figura 10. Uma variável declarada como `unattached` tem valor indefinido até que seja associada a um objeto já existente através da função `attach`. Ou seja, uma declaração como esta, ao invés de uma feita usando `remote`, não cria um objeto remoto: apenas declara uma variável que depois representará um objeto remoto. Dessa maneira é possível usar um objeto remoto que já tenha sido criado, desde que seja possível obter uma referência a esse objeto — no caso, através de consulta ao Servidor de Nomes.

Enquanto não é associada a um objeto, uma variável declarada como `unattached` é inútil, e qualquer tentativa de usá-la causará a ativação de uma exceção.

## Exceções

A linguagem Cm possui um mecanismo de tratamento de exceções baseado no modelo de terminação, e muito semelhante ao mecanismo implementado em C++. Uma exceção, ao ser ativada, pode transportar um valor, cujo tipo será usado para encontrar um tratador apropriado para essa exceção.

**FIGURA 11.** Propagação de exceções entre objetos distribuídos




Os tratadores de exceção estão dentro dos chamados blocos de comandos protegidos. Esses blocos são delimitados por `[` e `]`, ao invés de `{` e `}`. Quando ocorre uma exceção dentro de um bloco protegido, o tipo da exceção é confrontado com os tipos relativos aos tratadores. Se um tratador é encontrado, a ação correspondente é executada e o fluxo de controle vai para depois do bloco protegido; caso contrário, o contexto relativo ao bloco protegido é encerrado e a exceção é propagada para o nível mais externo.

Tratadores são associados a blocos estaticamente, mas são ativados dinamicamente, conforme a seqüência das chamadas de funções. Um tratador pode, ao invés de tratar uma exceção, propagá-la para o nível mais externo, se considerar que não tem condições de tratá-la satisfatoriamente.

Uma nova versão da classe `Stack` (figura 11) pode ser usada para implementar pilhas que detectam situações limite. São dois casos: quando a pilha está vazia e recebe uma chamada de `pop()`; e quando a pilha está cheia e recebe uma chamada de `push(..)`. Esses métodos testam se há espaço livre ou dados disponíveis, conforme seja necessário para que o método seja executado. Caso a operação pedida não seja possível, é gerada uma exceção (comando `raise`). As duas exceções geradas por essa classe Stack são cadeias de caracteres, do tipo `char*`. Na classe `StrStack`, uma chamada ao método `pop()` está dentro de um comando protegido. Se essa chamada causar, nesse método, uma exceção, ela será tratada pelo tratador mostrado na figura; a variável `s` vai conter o valor especificado no comando `raise`.

## Concorrência

Como vimos no exemplo anterior, através do uso do Servidor de Nomes um objeto pode ser localizado em qualquer ponto da rede. Portanto, um programa que ofereça serviços pode estar disponível para quem quer que queira utilizá-lo, bastando para isso saber a classe desse objeto servidor e qual o nome simbólico com que ele foi cadastrado. Assim, num determinado momento vários clientes podem estar fazendo requisições a um mesmo servidor. Temos então a possibilidade de concorrência no objeto servidor, pois é possível que, durante a execução de um serviço pedido numa requisição, chegue uma outra requisição.

De maneira geral, podemos dizer que a maneira de agir nesse caso diz respeito apenas ao objeto servidor, porque aos clientes é irrelevante saber como o servidor é implementado. De fato, a linguagem não estabelece uma regra a ser seguida

em todos os casos. A maneira mais fácil de resolver o problema é fazer com que, durante a execução de um serviço, novas requisições sejam enfileiradas à espera de atendimento pelo servidor.

Por outro lado, pode-se aumentar o desempenho das aplicações permitindo concorrência dentro do servidor. Assim, ao chegar uma requisição de serviço, ela é atendida por um fluxo de controle criado especialmente para esse fim. Um fluxo de controle, mais conhecido como *thread*, é uma unidade de execução menor que um processo [Sun93].

Uma *thread* é composta basicamente de uma pequena área de dados e de uma pilha. As *threads* ativas dentro de um objeto compartilham informações comuns, que são as variáveis armazenadas pelo objeto. As *threads* criadas dentro de um objeto são invisíveis fora dele.

Nesse caso, é necessário incluir na linguagem mecanismos de exclusão mútua e sincronização entre *threads*, que podem estar competindo, num determinado instante, por um mesmo recurso. Em Cm Distribuído, o paralelismo será controlado através de regiões críticas, dentro das quais os recursos do objeto estão protegidos contra acesso indevido.

O mecanismo a ser adotado, a chamada região crítica condicional, foi proposto em [Hoa72]. Uma região crítica condicional consiste de um trecho de código que protege recursos compartilhados, e que pode ser executada quando nenhum outro processo estiver na região. Além disso, uma expressão lógica estabelece uma condição a ser satisfeita para que a região possa ser executada. Se a condição não é satisfeita, quem tentava entrar na região fica bloqueado, podendo entrar depois, se a condição puder ser satisfeita. A notação proposta original, proposta em [Hoa72], é

```
with r when B do C
```

onde *r* representa um recurso (conjunto de dados protegidos), *B* é uma expressão lógica que controla o acesso e *C* é a região crítica propriamente dita.

Na linguagem Cm Distribuído essa notação foi modificada da forma

```
<$ exp_reg; exp_log $> C
```

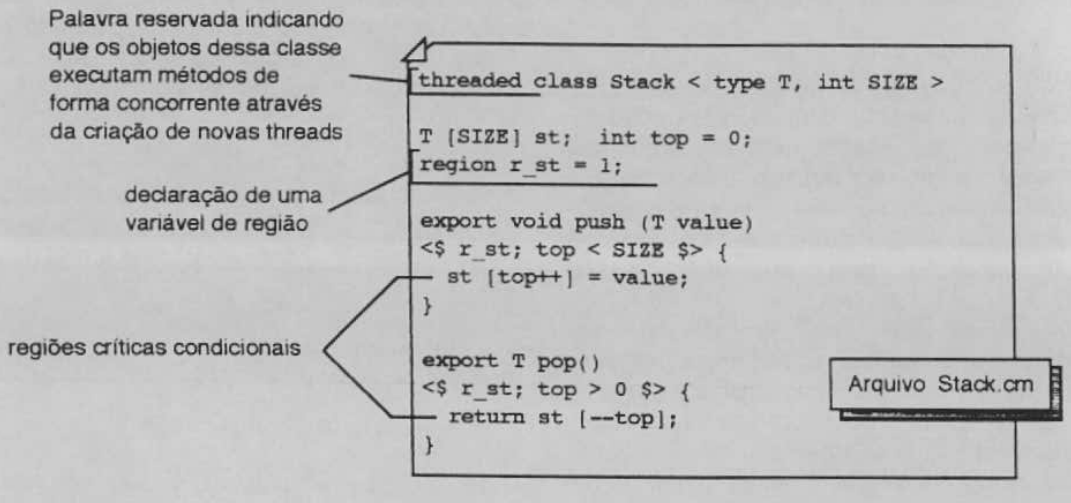
onde *exp_reg* é uma expressão de região e *exp_log* é uma expressão lógica. Uma expressão de região é uma expressão envolvendo variáveis de região, que são do tipo `region`, pré-definido na linguagem. Através das expressões de região, podemos ter exclusão mútua com relação uma série de variáveis de região. Variáveis de região são declaradas da seguinte maneira:

```
region r1 = 1;
// um thread por vez
region r4 = 4;
// quatro threads por vez
region r;
// default = 1 por vez
```

TABELA 1. Exemplos de regiões críticas condicionais

| | |
|---|---|
| <$ r1 $> | Execute a região crítica quando não houver nenhum *thread* em regiões guardadas por `r1`. |
| <$ r4; i == j $> | Execute a região crítica quando houver menos de quatro *threads* em regiões críticas guardadas por `r4` e quando `i` for igual a `j`. |
| <$ r1 && r4 < 3; x != y $> | Execute a região crítica se não houver nenhum *thread* em regiões guardadas por `r1` e se houver menos de 3 *threads* em regiões guardadas por `r4`; além disso, `x` deve ser diferente de `y`. |
| <$ r1; x > 0 && y > 0 $> | Execute a região crítica quando não houver *threads* em nenhuma região guardada por `r1` e quando os valores de `x` e `y` forem maiores que 0. |
| <$ all; i == 1 $> | Execute a região crítica em exclusão mútua com todas as demais regiões criticas da mesma classe, quando `i` for igual a 1. |

**FIGURA 12.** Classe com mecanismos de concorrência




Uma variável de região pode aparecer em mais de uma expressão de região. A avaliação da expressão de região e da expressão lógica é feita atomicamente, sem possibilidade de interferência de outras *threads*. No término da região crítica, as expressões nas quais há *threads* bloqueadas são avaliadas para determinar se uma nova região crítica pode ser executada.

Essa nova versão da classe `Stack` (figura 12) implementa pilhas que podem atender pedidos de escrita ou de leitura sem que os objetos causem exceções nas situações limite. Quando não há espaço disponível para escrita, ou não há dados suficientes, para leitura, a execução da requisição fica bloqueada na expressão lógica da região crítica, até que um outro objeto libere espaço da pilha ou coloque um dado novo, conforme o caso. Existem situações, entretanto, em que um objeto que fez a chamada não pode ser bloqueado por um tempo arbitrário, esperando até que o servidor passe para um estado conveniente. Para tal objeto seria melhor saber, o mais cedo possível, que sua chamada não pôde ser executada por não cumprir as condições. Nesse caso, o mecanismo de região crítica condicional oferece facilidades para a geração de uma exceção, quando a execução do serviço for bloqueada em uma região crítica.

## Portas de dados

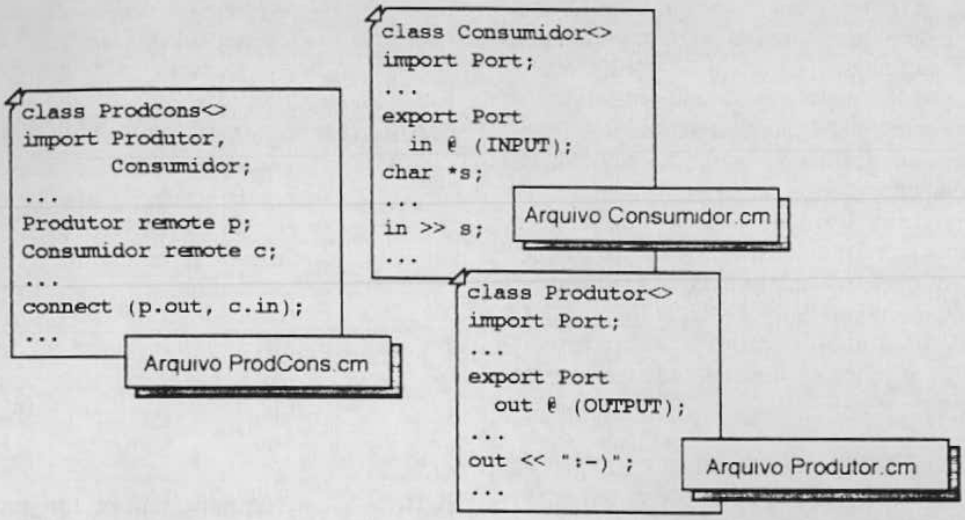
A comunicação de dados de um objeto com o mundo externo é efetuada através de portas de dados. Estas portas são o mecanismo de entrada e saída de arquivos bem como transferência de massas de dados entre objetos distribuídos.

O conceito de portas de dados que está sendo introduzido na linguagem Cm oferece um grande conjunto de facilidades. Dada a sua extensão apresentaremos apenas alguns aspectos mais interessantes; uma descrição mais detalhada está em [Dru93].

As portas de dados da linguagem Cm são uma elaboração das portas do OMNI (seção "O Sistema OMNI"). A interface para utilização de uma porta OMNI é semelhante a um descritor de arquivo do sistema UNIX, e as operações de leitura e escrita são efetuadas através de funções bastante básicas. Esses recursos do OMNI estão em um nível de abstração baixo em relação ao poder de expressão da linguagem Cm, portanto são encapsulados e oferecidos, pela linguagem, através da classe pré-definida `Port`.

As portas de dados de um objeto fazem parte de sua interface com o mundo externo, sendo que as conexões dessas portas com portas de outros objetos podem ser feitas sem a sua participação direta. As portas estabelecem transparência na comunicação, pois um objeto pode estar ligado a um arquivo ou a outro programa sem que isso interfira no seu funcionamento. Esse princípio é semelhante ao usado para descritores padrão dos programas no ambiente UNIX.

FIGURA 13. Exemplo de uso de portas em Cm




Um exemplo de uso de portas na linguagem Cm é mostrado na figura 13. Um objeto da classe `Produtor` gera informações que são enviadas por uma porta de saída, e um objeto da classe `Consumidor` recebe informações por uma porta de entrada. Um produtor e um consumidor são criados e conectados através de um objeto da classe `ProdCons`.

A classe `Port` possui uma especialização para portas tipadas, a subclasse `TypedPort`. Portas de dados simples transmitem seqüências de caracteres sem uma estrutura associada; já as portas tipadas têm um tipo associado, e as operações de entrada e saída são efetuadas sobre valores desse tipo. As funções oferecidas para uso de portas tipadas tratam de detalhes como verificação dos tipos envolvidos em uma conexão e da transformação dos valores enviados e recebidos pelas portas (que podem conter estruturas de dados bastante complexas) em seqüências de caracteres e vice-versa. O uso de portas tipadas contribui para construir programas cuja interação tem uma semântica mais rica e para diminuir o acoplamento entre módulos.

## Conclusões

Apresentamos o conceito de Objetos Distribuídos, usado na linguagem Cm para a construção de aplicações distribuídas. Esse trabalho é o resultado mais recente das pesquisas sendo conduzidas pelo Projeto A_HAND nas áreas de Linguagens de Programação e Sistemas Distribuídos.

O conceito de Objetos Remotos parece-nos ser poderoso e genérico o suficiente para servir de base para as aplicações distribuídas que estamos implementando no nosso ambiente. Acreditamos que através desse conceito será possível incorporar à linguagem Cm mecanismos de programação distribuída de maneira uniforme e elegante. Associadas a esse trabalho há áreas de pesquisa muito interessantes, que ainda não foram analisadas a fundo mas que estão na direção natural de esforços futuros. Como exemplos podemos citar migração de objetos, depuração de objetos distribuídos e objetos persistentes.

A extensão de linguagens orientadas a objetos para permitir programação distribuída é um tópico de pesquisa em destaque atualmente. Como exemplo de outros trabalhos na área podemos citar o projeto COOL [Lea93], e a extesão da linguagem Eiffel apresentada em [Mey93].

O projeto COOL busca oferecer suporte para programação distribuída baseada no sistema Chorus [Arm89]; o próprio nome COOL significa Chorus Object-Oriented Layer. São definidos três níveis: um básico, que interage diretamente sobre o núcleo Chorus; um nível intermediário, que implementa um modelo de objetos genérico, com suporte para execução e comunicação transparente; e um nível final, específico para uma lin-

guagem de programação, que faz a interface entre o modelo COOL e essa linguagem [Lea93].

O modelo proposto [Mey93] para a extensão de Eiffel busca adaptar essa linguagem para programação distribuída através de modificações mínimas na linguagem. O artigo relatando o projeto COOL se concentra no ambiente onde o modelo de objetos distribuídos será implementado; este artigo sobre Eiffel, ao contrário, analisa em profundidade os critérios a serem usados para decidir como uma linguagem pode ser alterada para oferecer um mecanismo de programação baseado em objetos distribuídos. As extensões propostas buscam combinar esses critérios com a filosofia de programação representada pela linguagem Eiffel e técnicas de orientação a objetos, como herança.

Uma vez terminada a implementação do Cm Distribuído, teremos condições de iniciar a implementação de diversas ferramentas de *groupware* e da linguagem de computações LegoShell. Acreditamos que a programação com objetos distribuídos com suporte da linguagem deverá diminuir sensivelmente o tempo necessário para o desenvolvimento de aplicativos distribuídos, com ganhos em portabilidade e robustez.

## Agradecimentos

Os autores são gratos aos membros do Projeto A_HAND que atuam nas áreas relativas a este trabalho e que têm colaborado construtivamente na discussão de conceitos e revisão dos trabalhos. São eles Alexandre P. Teles, Bill W. C. de Oliveira, Carlos A. Furuti e Cassius Di Cianni.

## Referências bibliográficas

[Agh90] **Concurrent Object-Oriented Programming**
Gul A. Agha
Communications of the ACM, 33(9), pp. 125–141, setembro 1990.

[Arm89] **Revolution 89 or "Distributing Unix Brings it Back to its Original Virtues"**
François Armand, Michel Gien, Frédéric Herrmann e Marc Rozier
Chorus Systèmes, CS/TR–89–36.1, agosto 1989.

[Dru87a] **A_HAND: Ambiente de Desenvolvimento de Software Baseado em Hierarquias de Abstração em Níveis Diferenciados**
Rogério Drummond e Hans K. E. Liesenberg
IV Encontro de Trabalhos do Projeto ETHOS, Petrópolis, RJ, abril 1987.
Revisto e reimpresso como relatório técnico, Projeto A_HAND, outubro 1987.

[Dru87b] **Requisitos para um ambiente de desenvolvimento de PROGRAMAS**
Rogério Drummond e Hans K. E. Liesenberg
I Encontro IBM de Ciência e Tecnologia em Informática, Rio de Janeiro, RJ, novembro 1987.

[Dru88] **Manual de referência da Linguagem Cm (versão preliminar)**
Rogério Drummond e Fábio Q. B. da Silva
Projeto A_HAND, DCC-IMECC-UNICAMP, março 1988.

[Dru89] **LegoShell Linguagem de Computações**
Rogério Drummond
III Simpósio Brasileiro de Engenharia de Software, Recife (PE), 1989.

[Dru92] **OMNI – Sistema de suporte a aplicações distribuídas**
Rogério Drummond e Cassius Di Cianni
Anais do VI Simpósio Brasileiro de Engenharia de Software, pp 309–324, novembro 1992.

[Dru93] **Aspectos da Implementação de Objetos Distribuídos**
Rogério Drummond, Celso Gonçalves Jr e Alexandre P. Teles
Anais do XI Simpósio Brasileiro de Redes de Computadores, pp 139–152, maio 1993.

[Fur91] **Um compilador para uma linguagem de programação orientada a objetos**
Carlos A. Furuti
Tese de mestrado, DCC-IMECC-UNICAMP, julho 1991.

[Gol83] **Smalltalk-80: The Language and Its Implementation**
Adele Goldberg e David Robson
Addison-Wesley, Reading (MA), 1983.

[Hoa72] **Towards a Theory of Parallel Programming**
Charles A. R. Hoare
Operating Systems Techniques
Academic Press, New York (NY), 1972.

[Ker78] **The C Programming Language**
Brian W. Kernighan e Dennis M. Ritchie
Prentice-Hall, Englewood Cliffs (NJ), 1978.

[Ker84] **The UNIX Programming Environment**
Brian W. Kernighan e Robert Pike
Prentice-Hall, Englewood Cliffs (NJ), 1984.

[Lea93] **COOL: System Support for Distributed Programming**
Rodger Lea, Christian Jacquemot e Eric Pillevesse
Communications of the ACM, 36 (9), pp 37–46, setembro 1993.

[Lis88] **Distributed Programming in Argus**
Barbara Liskov
Communications of the ACM, 31 (3), pp 300–312, março 1988.

[Mey93] **Systematic Concurrent Object-Oriented Programming**
Bertrand Meyer
Communications of the ACM, 36 (9), pp 56–80, setembro 1993.

[Raj91] **Emerald: A General-Purpose Programming Language**
Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson e Eric Jul
Software — Practice & Experience, 21 (1), pp 91–118, janeiro 1991.

[Sil88] **Programação em Cm**
Fábio Q. B. da Silva, Hans K. E. Liesenberg e Rogério Drummond
Anais do XV Semish, pp 101–102, Rio de Janeiro, RJ, julho 1988.

[Str91] **The C++ Programming Language (2nd Ed.)**
Bjarne Stroustrup
Addison-Wesley, Reading (MA), 1991.

[Sun93] **SunOS 5.2 System Services**
Sun Microsystems Inc. 1993.

[Tel93] **A linguagem de programação Cm (versão 2.0x)**
Alexandre P. Teles
Tese de mestrado, DCC-IMECC-UNICAMP, novembro 1993.

[Yok87] **Experience and Evolution of ConcurrentSmallTalk**
Yasuhiko Yokote e Mario Tokoro
OOPSLA'86 Proceedings, pp 406–415, outubro 1987.