Sistema de Suporte para Objetos Remotos*

Mauro da Silva Oliveira Filho
maurosof@ahand.unicamp.br

Célio Norbiato Targa
targa@ahand.unicamp.br

Rogério Drummond
rog@ahand.unicamp.br

Laboratório A-HAND
www.ahand.unicamp.br
Instituto de Computação
Unicamp
Rua Roxo Moreira 1076
13083-591 Campinas, SP

(abril 1997)

RESUMO

Apresentamos o conceito de Objetos Remotos que pode ser utilizado por linguagens de programação para facilitar o desenvolvimento de aplicações distribuídas. Nesse artigo descrevemos a especificação de um sistema de suporte que oferece a abstração de Objetos Remotos. Esse sistema divide-se em duas camadas: básica e de objetos. A camada básica não possui o conceito de objetos e é descrita de forma independente de linguagem; fornece suporte às operações da camada de objetos e a um mecanismo de controle de concorrência. A camada de objetos oferece um serviço de nomes e operações que permitem a utilização de Objetos Remotos.

Apresentamos também a arquitetura do sistema, que se encontra em fase final de desenvolvimento. Usamos a linguagem Cm Distribuído (CmD) para apresentar um exemplo que mostra o uso efetivo de Objetos Remotos.

ABSTRACT

We describe the notion of *Remote Objects* which can be incorporated in object-oriented languages for easy development of distributed applications. This article focuses on the run-time system supporting Remote Objects. This layered system comprises both a low-level, language-independent layer which does not directly supports objects, and an upper-level layer offering a name server and supporting the Remote Objects notion.

We describe the macro architecture of the run-time system currently used by the CmD (Distributed Cm language) compiler. Examples are shown in CmD to stress the usefulness of the notion of Remote Objects.

* Este trabalho foi parcialmente financiado pela Fapesp, processos número 95/6601-6 e 96/0882-6, e pelo PROTEM/CNPq, processo número 680089/94-2.

INTRODUÇÃO

Um programa distribuído é composto de partes que executam em máquinas diferentes, cooperando para atingir o objetivo do programa. Quando estas partes são objetos e sua cooperação é realizada pelos mecanismos naturais de orientação a objetos (i.e., chamada de métodos), então o programa pode ser qualificado como programa distribuído orientado a objetos. Estes programas podem ser criados sem suporte direto da linguagem orientada a objetos no que tange à distribuição. Neste caso, usa-se recursos externos a linguagem, tipicamente envolvendo uma programação em baixo nível usando primitivas do sistema operacional ou de um *middleware* que pode ser tão elaborado como o proposto por CORBA [OMG95].

Enquanto à programação das partes pode ser estritamente orientada a objetos, a sua interligação para realização do programa distribuído pode ser totalmente não orientada a objetos.

A maior contribuição da nossa linha de pesquisas foi a introdução, a nível da linguagem de programação, de conceitos que possibilitam a especificação e interligação de objetos distribuídos como uma extensão natural dentro do paradigma de orientação a objetos [Gon94b]. A noção de *Objeto Remoto* é central na nossa abordagem e será descrita mais adiante. O foco deste artigo é a descrição do Sistema de Suporte (*Run Time System* – RTS) para Objetos Remotos. Ele pode também ser utilizado em linguagens que não incorporam o conceito de objetos remotos; neste caso, oferece uma API (*Aplication Program Interface*) de mais alto nível para a integração de objetos distribuídos.

Um programa composto de objetos distribuídos deve ser representado, em tempo de execução, por vários objetos independentes que se comunicam solicitando serviços (invocando a execução de métodos) de *objetos servidores*. Um *objeto cliente* pode também ser servidor para outros objetos clientes.

Os mecanismos básicos de suporte ao uso de objetos distribuídos são:

- criação de objetos distribuídos;
- comunicação transparente entre objetos;
- representação universal de tipos;
- propagação de exceções;
- concorrência dentro de objetos.

O RTS oferece a abstração de objetos distribuídos fornecendo operações para criar, destruir, nomear e fazer busca de objetos. Além disso, apresenta facilidades para execução de métodos remotos, verificação de tipos, propagação de exceções, gerenciamento de *threads*, controle de concorrência dentro de objetos, entre outras.

Ele é composto de duas camadas: *camada de objetos*, que fornece a abstração de *Objetos Remotos*, e *camada básica*, que não possui o conceito de objetos, mas fornece suporte às operações da camada de objetos e a um mecanismo de controle de concorrência.

Primeiramente é apresentado o conceito de Objetos Remotos, introduzido em [Dru94] e refinado posteriormente. Objetos Remotos podem ser incorporados na maioria das linguagens orientadas a objetos. Usamos a linguagem Cm Distribuído (CmD)[Gon94b] como veículo para exemplificar, mostrando o uso do conceito numa linguagem de programação. A linguagem CmD é uma extensão feita à linguagem Cm [Tel93, Fur91] adicionando o conceito de Objetos Remotos.

Na seção *Especificação do Sistema de Suporte* são discutidas as camadas do RTS e as operações oferecidas por elas. Em seguida, é discutida a *Arquitetura do Sistema de Suporte* adotada para a implementação. Finalmente é apresentado um exemplo para ilustrar a execução de objetos em máquinas diferentes.

OBJETOS REMOTOS

O que diferência um *objeto remoto* de um objeto comum é o fato do primeiro estabelecer um contexto de execução próprio. No UNIX, por exemplo, um objeto remoto está associado a um processo, enquanto objetos comuns são *sempre* componentes de um objeto remoto.

Desta forma, uma aplicação distribuída é um conjunto de Objetos Remotos independentes, mas que podem cooperar para um objetivo comum. Um objeto é uma instância de uma classe e pode ser declarado como objeto remoto ou não. O trecho de código CmD a seguir exemplifica a declaração e uso de objetos em CmD. Usamos a classe `Buffer<>` definida no Apêndice A.

```
class Exemplo< >
      import Buffer;                          // a classe Buffer pode ser usada
      Buffer<>          b;                    // objeto comum de tipo Buffer<>
      remote<Buffer<>>  r("SuperServer");     // objeto remoto na máquina SuperServer
      remote<Buffer<>>* ar;                   // apontador para um objeto remoto
      int               i;

      export m()        // método da classe Exemplo.  Pode ser invocado externamente
      {
            b.insert(10);      // invoca o método insert do Buffer b
            r.insert(20);      // não há diferença para objetos remotos

            ar = &r;           // atribuição correta
            i = b.remove();
            i = ar->remove(); // remove de r

            ar->Attach("buffer_público"); // liga ar a um buffer registrado no
                                          // servidor de nomes
      }
```

Os objetos `b`, `r` e `ar` são, respectivamente, um objeto comum, um objeto remoto instanciado na máquina `SuperServer` e um apontador para um objeto remoto, todos de tipo `Buffer<>`. São todos de tipos diferentes em CmD. O uso de um objeto comum e um objeto remoto de mesmo tipo é identico. Assim, um programa centralizado pode facilmente ser transformado em distribuído bastando transformar seu tipo original como parâmetro da classe pré-definida `remote<>`.

Toda e qualquer execução em CmD se inicia com a ativação de um objeto remoto. Como na classe `Exemplo`, um objeto pode "conter" objetos comuns e objetos remotos (na realidade, referências a estes objetos). Um objeto pode manipular os objetos remotos por ele criados ou não. No último comando no exemplo acima, o apontador para objeto remoto de tipo `Buffer<>` recebe a referência de um objeto (de tipo `remote<Buffer<>>`) e registrado com nome `buffer_público`, se este objeto existir.

A Figura 1 ilustra uma aplicação distribuída composta por um conjunto de Objetos Remotos criados em diferentes máquinas do sistema.

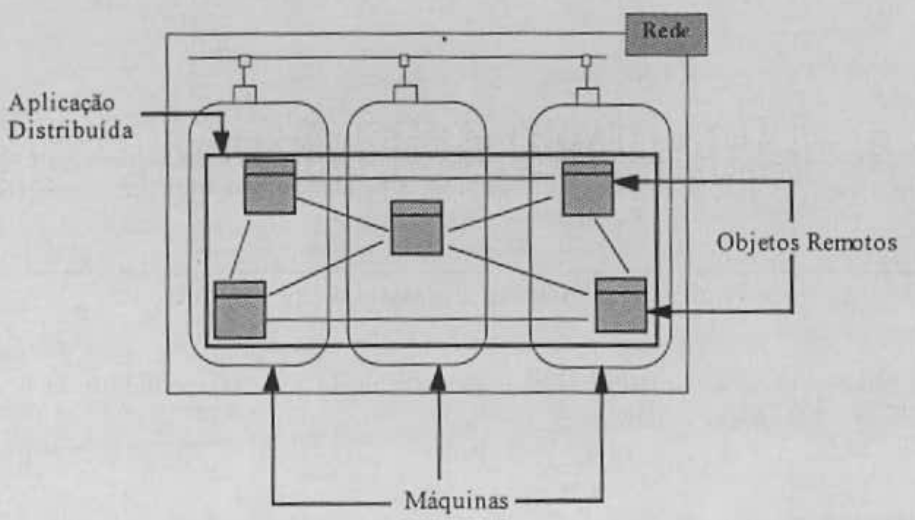

Figura 1: Aplicação distribuída formada por um conjunto de Objetos Remotos.

Objetos Remotos podem criar outros Objetos Remotos, estabelecendo relações de "paternidade". Um Objeto Remoto é *vinculado* se o seu tempo de vida for limitado pelo tempo de vida do objeto que o criou [Gon94b]. Objetos Remotos podem ser conhecidos e usados por outros objetos; assim, cada Objeto Remoto pode ter qualquer um de seus métodos públicos invocado por objetos que tenham acesso a ele. Um Objeto Remoto pode ser desvinculado do objeto pai para que o seu tempo de vida se prolongue pelo tempo em que for útil para outros objetos.

Objetos Remotos em uma aplicação distribuída podem ser compartilhados por vários objetos de outras aplicações. Dessa forma, esses Objetos Remotos atuam como servidores e podem receber vários pedidos de execução de métodos simultaneamente. Para não limitar o potencial paralelismo das aplicações, as requisições dos serviços podem ser atendidas de forma concorrente. Tal fato possibilita e torna interessante a utilização de Objetos Remotos *multi-threaded*, onde cada chamada de método por um cliente causa a criação de uma *thread* que será responsável pela execução do método.

Nesse contexto, a *thread* que faz a chamada no cliente e aquela *thread* que é criada para atender a requisição serão consideradas uma única "*thread* virtual". Dessa forma, podemos considerar que a *thread* do cliente se estende ao servidor. Assim as *threads* das aplicações "caminham" entre os objetos, mantendo a uniformidade das chamadas remotas de métodos (RPC). Pode-se dizer que a *atividade* exercida por uma *thread* do cliente é transferida para a *thread* do servidor, retornando ao cliente no fim da execução do método invocado.

Um Objeto Remoto pode ser de natureza passiva ou ativa. Para um objeto passivo, todo processamento é restrito à execução dos seus métodos por atividades externas. O objeto não contêm atividades próprias.

Objetos ativos, por outro lado, podem ter várias atividades próprias. Cada atividade é implementada como uma *thread* que pode permanecer durante a existência do objeto. Um veículo real, como um avião, pode ser representado como um objeto ativo. Ele oferece métodos públicos que retornam localização atual do avião e métodos que alteram sua velocidade e direção de percurso. Sua localização pode ser constantemente atualizada por uma atividade em função da velocidade e direção.

Essas atividades (*threads*) executarão em paralelo com aquelas que executam os métodos por chamadas externas. Essa funcionalidade é alcançada com a especificação de métodos **`thread`** no objeto servidor. A chamada a esses métodos é sempre assíncrona e implica na criação de uma nova atividade. A chamada prepara a criação de uma *thread* para a execução do método e retorna em seguida, deixando o método em execução. Esses métodos **`thread`** podem ser públicos ou privados determinando assim o escopo de seus potenciais clientes. O construtor de um Objeto Remoto pode invocar um de seus métodos **`thread`** de forma que o objeto tenha comportamento ativo desde sua criação.

ESPECIFICAÇÃO DO SISTEMA DE SUPORTE

Durante a especificação do sistema de suporte foram levantadas algumas das funcionalidades necessárias. O RTS é basicamente composto de duas camadas: *camada de objetos* e *camada básica*. Sua disposição é mostrada na Figura 2.

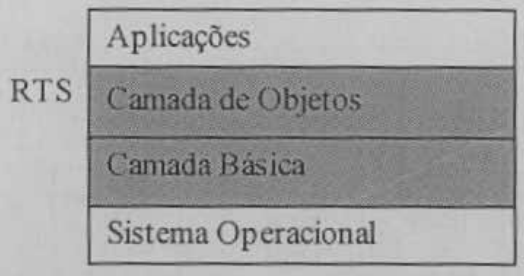

Figura 2: Camadas do sistema de suporte.

Adotamos uma abordagem onde a camada básica é descrita de forma independente de linguagem, da mesma forma que em COOL [Lea93] e em Amadeus [Cah93].

Camada Básica

A camada básica não possui o conceito de objetos, mas fornece suporte às operações da camada de objetos e à concorrência dentro de objetos oferecendo as seguintes funcionalidades:

- criação e destruição de contextos de execução;
- requisições remotas;
- serviço de nomes;
- propagação de exceções;
- controle de concorrência.

Criação e destruição de contextos de execução

O RTS fornece suporte para criar e destruir contextos de execução em diferentes máquinas do sistema. A criação de contextos de execução é implementada pela seguinte função:

```
cid ContextCreate(host, par_execution, env, exc)
```

Esta função possibilita a criação de um contexto de execução na máquina *host* indicada. O parâmetro *par_execution* indica o nome do arquivo executável e o parâmetro *env* é utilizado para a iniciação do ambiente do contexto de execução. Retorna um identificador *cid* para o contexto de execução criado. O parâmetro *exc* das funções apresentadas será discutido na sub-seção *Propagação de Exceções*.

Um contexto de execução *cid* pode ser destruído explicitamente através da função:

```
ContextDestroy(cid, mode, timeout, exc)
```

Os parâmetros *mode* e *timeout* são utilizados para configurações específicas do sistema.

Um contexto de execução é criado vinculado àquele que o criou; quando um contexto é destruído, todos os contextos a ele vinculados também serão destruídos. O RTS oferece um mecanismo para desvincular um contexto de execução daquele que o criou:

```
ContextDetach(cid, exc)
```

Requisições remotas

Requisições podem ser feitas entre diferentes contextos de execução espalhados pela rede. Assim, a camada básica do RTS fornece suporte à chamada de requisições remotas.

A função do RTS para suporte à chamada de requisições remotas é definida como:

```
resultdescriptor SCall(cid, fid, parlist, exc)
```

Esta operação faz uma chamada a um procedimento remoto identificado por *fid* no contexto de execução *cid* com a lista de parâmetros *parlist*. Essa função é semelhante a uma chamada de procedimento remoto.

Serviço de Nome

As funções de registro e busca de nomes utilizam um serviço de nomes fornecido pelo sistema de suporte. Um serviço de nomes considerado por nós ideal possui características como registro de nomes com atributos, busca por contexto e localidade. A discussão destes aspectos foge do escopo deste artigo.

O registro de nome é feito através da função:

```
Registry(cid, name, regparameters, exc)
```

Esta função é utilizada para registrar o contexto de execução identificado por *cid* no serviço de nomes com o nome *name*. O parâmetro *regparameters* indica os atributos do nome, por exemplo se sua visibilidade é local ou global.

A busca através de nomes registrados no serviço de nomes é implementada no RTS pela função:

```
cid Search(name, context, location, exc)
```

Propagação de Exceções

Todas as funções fornecidas pela camada básica do RTS possuem um parâmetro denominado *exc*, utilizado para sinalização de erros. Erros tipicamente podem ocorrer durante a execução das funções do RTS ou podem ser exceções sendo propagadas no caso de chamada de método remoto.

Controle de Concorrência

Como descrito na seção anterior, um Objeto Remoto pode receber e atender várias requisições simultaneamente (mesmo sendo de natureza passiva). Dessa forma, tais objetos fazem uso de um mecanismo que garante acesso disciplinado aos seus recursos e sincroniza as *threads* que estão executando.

O mecanismo adotado no sistema de suporte é baseado no modelo de Regiões Críticas Condicionais [Hoa72]. Algumas extensões foram feitas ao mecanismo de forma a dar maior flexibilidade à linguagem que fará uso do sistema [Gon94a, Gon94b].

Dessa forma, temos o conceito de *variáveis de região* (VR), usadas para proteger os dados de acesso indevido, controlando a entrada de *threads* dentro das regiões críticas. Não existe nenhum vínculo explícito entre as VRs e os recursos do objeto como na proposta de Hoare e no seu refinamento em Andrews [And83].

Cada VR é inicialiazada em sua declaração com sua cardinalidade máxima. A Cardinalidade é um limite superior do número de *threads*concorrentes dentro de regiões críticas protegidas pela VR e corresponde a um relaxamento da exclusão mútua, permitindo compartilhamento controlado por múltiplas *threads*.

```
region a;       // cardinalidade máxima default (=1)
region b = 2;   // cardinalidade máxima = 2
region c = 0;   // cardinalidade máxima ilimitada
```

Uma região crítica em CmD apresenta a seguinte sintaxe:

```
with (<expr-reg> ; <cond-sinc>) {
    <comando>
}
```

onde,

- *<expr-reg>* é uma expressão de região composta de variáveis de região, operadores '&&' e '='.
- *<cond-sinc>* é a condição de sincronização, que permite o acesso de uma *thread* à região sob uma determinada condição.

Se *<expr-reg>* ou *<cond-sinc>* não for satisfeita, a *thread* fica bloqueada na entrada da região à espera de uma nova oportunidade de fazer os testes. Nesse caso, o sistema de suporte implementa o mecanismo de forma que a *thread* seja desbloqueada tão logo suas condições sejam verdadeiras.

Variáveis de região podem ser agrupadas em expressões de região, por meio dos operadores '&&' e '='. O operador '&&' faz um *AND* das VRs e o operador '=' redefine temporariamente a cardinalidade de uma VR, restringindo o número de *threads* que podem entrar na região. Se uma *thread* que quer entrar em uma região não satisfizer as restrições impostas por todas as VRs que participam de um expressão, ela será bloqueada.

O mecanismo também permite a definição de uma condição de sincronização que pode ser associada a uma determinada região. Dessa forma, uma *thread* só terá acesso à região crítica se a condição for satisfeita, caso contrário a *thread* será bloqueada. Os dados que compõem uma condição de sincronização podem ser alterados em qualquer lugar e a qualquer momento, não sendo restritos apenas às regiões críticas. A expressão de região e a condição de sincronização representam o que chamamos *guarda* da região.

O teste da condição de sincronização deve ser feito com exclusividade em relação às demais *threads* executando dentro do objeto, ou seja, todas as *threads* devem ser "congeladas" temporariamente para que o teste seja feito e a região comece a executar (em caso satisfatório). O sistema garante atomicidade entre o teste da guarda e o início da execução de uma região, ou seja, quando uma região crítica começa a executar, a condição de sincronização é garantidamente verdadeira.

Cada uma das regiões críticas definidas nos métodos de uma classe resulta na criação de uma estrutura do tipo `region` no escopo do objeto. Essa estrutura é utilizada para controlar a entrada e saída de *threads* da região crítica.

O suporte do RTS para o controle de concorrência é composto basicamente de funções para a criação de VRs e regiões, e de funções para entrada e saída de regiões críticas. Dessa forma, a função que uma *thread* deve executar antes de entrar em uma região crítica é definida como:

```
void RegionEntry(region, thrId)
```

Nessa operação o sistema faz o teste da guarda e decide entre bloquear a *thread* ou deixá-la executar a região. Além disso, esta função é responsável pela procura de novas *threads* para executar em outras regiões com acesso liberado.

Uma *thread* terminando de executar uma região crítica deve invocar:

```
void RegionExit(region, thrId)
```

Esta função é responsável pela atualização das estruturas internas de forma que a região crítica possa aceitar novas *threads*. Além disso, ela deve procurar por *threads* que podem ocupar o espaço liberado pela *thread* que está saindo da região.

A relação entre várias regiões é feita por meio de um gerente responsável por organizá-las em um grafo de dependências relativo às variáveis de região. Isso é feito de forma a otimizar a busca feita pelo RTS por uma nova *thread* para executar. Nesse caso, o sistema de suporte implementa o mecanismo de forma que uma *thread* seja desbloqueada tão logo sua guarda seja satisfeita. Esse gerente também é responsável pela detecção de possíveis *deadlocks*.

Como uma *thread* pode ser liberada para execução (sua condição de sincronização torna-se verdadeira) em qualquer momento na execução de um método, as funções *RegionEntry* e *RegionExit* não podem garantir que uma *thread* nunca ficará bloqueada para sempre na entrada de uma região. Por essa razão, uma *thread* de controle, que fica analisando as regiões à procura de *threads* bloqueadas para executar, é criada para cada gerente. Esta *thread* também cuida de rearranjar as estruturas internas do controle de concorrência, de forma a otimizar o serviço feito pelas funções descritas acima.

Camada de Objetos

A camada superior do RTS é chamada camada de objetos. Ela oferece a abstração de Objetos Remotos com operações como:

- criação e destruição de Objetos Remotos;
- registro de nomes para objetos;
- desvinculação do tempo de vida de objetos;
- chamada de método remoto;
- propagação de exceções entre objetos.

Objetos a ser criados remotamente devem fornecer serviços de ligação com a camada de objetos, definidos de acordo com a linguagem de programação usada. Em CmD, basta parametrizar a *meta-classe*[†] predefinida *remote*:

[†] O construtor `class` em Cm e CmD aceita parâmetros (inclusive tipos): é uma *meta-classe* que pode criar automaticamente um número arbitrário de classes convencionais. Essa capacidade [Dru87a, Dru87b] antecede o conceito de *templates* em C++ enquanto acrescenta maior uniformidade à linguagem. Meta-classes que têm especificações de tipo por parâmetros são chamadas *polimórficas*. O termo meta-classe utilizado neste artigo refere-se ao conceito de meta-classe em Cm, que é diferente do conceito de meta-classe de reflexão computacional.

```
class remote<type T>
   inherit T;
   export constructor();
   export constructor(char *hostName);
   export destructor();
   export operator new();
   export operator release();
   export int Attach(char *objectName);
   export int Registry(char *objectName);
   export int Unbind();
   void MCall(MethodID mid, ParList pars, ResultDescriptor &res);
```

A meta-classe *remote* define métodos essenciais para todo Objeto Remoto. A parametrização dessa classe gera classes cujas instâncias são Objetos Remotos. Estes Objetos Remotos além dos métodos definidos na sua classe, poderão executar os métodos definidos na classe *remote*.

Para criar e destruir os Objetos Remotos são definidos os construtores e o destrutor. Construir um Objeto Remoto sem especificar a máquina significa deixar a cargo do RTS a escolha do *host* para sua execução. Objetos Remotos também podem ser criados e destruídos dinamicamente através das operações *new* e *release*.

O método *Attach* faz uma consulta no serviço de nomes para encontrar o objeto registrado com o nome *objectName*; encontrado esse objeto, o valor do apontador é preenchido com essa referência, e a partir daí, o apontador para o Objeto Remoto pode ser utilizado para fazer chamadas de métodos remotos. O registro de um objeto no serviço de nomes é feito através do método *Registry*.

Um Objeto Remoto é criado com seu tempo de vida vinculado ao do objeto que o criou, assim, o seu tempo de vida é limitado pelo tempo de vida do objeto pai. A desvinculação é feita por uma chamada do método *Unbind*.

O suporte à chamada de métodos remotos também é fornecido nessa camada. Uma chamada a um método remoto pode ser entendida como uma chamada a um método herdado pela classe *remote*. Esta chamada, por sua vez, pode ser entendida como a chamada à função genérica *MCall* que faz a chamada de métodos remotos. Esta função é invocada passando como parâmetros o identificador do método, os parâmetros do método e uma referência para o resultado. No corpo da função *MCall*, os parâmetros são linearizados e é chamada a função *SCall* da camada básica do RTS.

Além dessas funcionalidades, a camada de objetos deve fornecer suporte à propagação de exceções. Caso a exceção seja levantada em um Objeto Remoto e o tratador para esta exceção esteja no objeto chamador, a exceção deverá ser propagada entre contextos de execução diferentes.

Uma proposta semelhante é o sistema de programação distribuída para Modula-3 *Network Objects* [Bir95]. Um *network object* é um objeto cujos métodos podem ser invocados através de uma rede. Fornece funcionalidade semelhante a RPC, mas de forma mais geral e mais fácil de usar. Tem como principal aspecto a sua simplicidade, que é obtida restringindo suas características para suportar somente funções consideradas importantes pelos seus autores para um sistema distribuído.

Propagação de Exceções

Quando uma exceção deve ser propagada entre Objetos Remotos, um objeto é construído com o tipo e o valor desta exceção. Esse objeto contém informações necessárias para que a exceção seja levantada no objeto que chamou o método remoto, ou seja, contém as informações necessárias para que a exceção seja propagada entre Objetos Remotos.

O RTS utiliza a classe *InternalExc* para dar suporte à propagação de exceções entre os diversos componentes do sistema. Uma *InternalExc* armazena o valor e o tipo da exceção original. Os principais atributos dessa classe são apresentados abaixo:

```
class InternalExc
   export  constructor();
   export  constructor(Type t, Value v);
   export  destructor();
   export int SetExc(Type t, Value v);
   export Value GetValue();
   export Type GetType();
   export operator=(const InternalExc &other);
   export int Raise();
   export void Clear();
   export int IsNull();
   export void Print();
   Type t;
   Value v;
```

Este mecanismo também é utilizado pelo RTS para levantar exceções que ocorrem durante a execução de suas funções. São exemplos de exceções do RTS: falha de comunicação, erro no serviço de nomes, erro nos componentes do RTS, erro na linearização, etc.

ARQUITETURA DO SISTEMA DE SUPORTE

Para a implementação do sistema de suporte para Objetos Remotos, foi elaborada uma arquitetura com dois componentes básicos: o *daemon*‡ *RODaemon* e a biblioteca *RemoteObject*, descritos a seguir.

As figuras 3 e 4 ilustram a arquitetura do RTS, mostrando como são gerados objetos executáveis a partir do código fonte da aplicação e como esses objetos são executados.

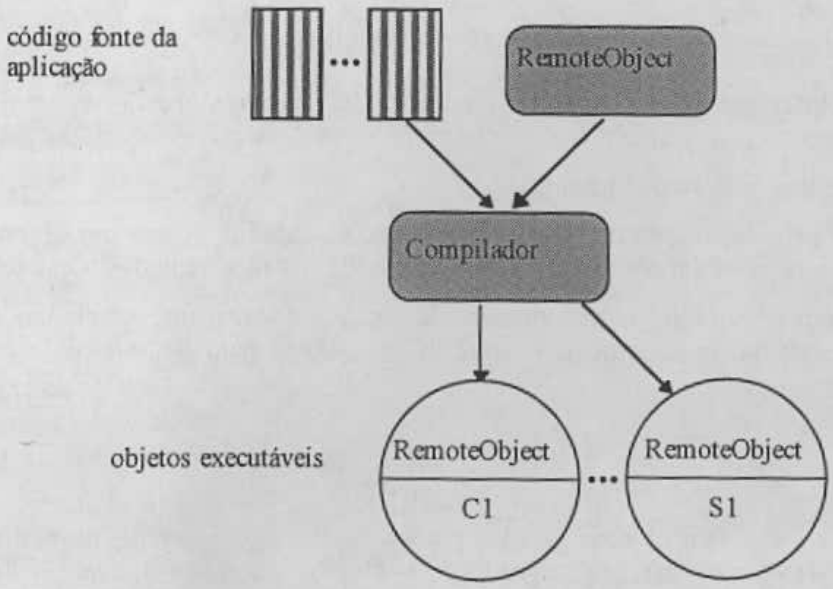

Figura 3: Geração de objetos executáveis.

A compilação de uma hierarquia de classes pode gerar um ou mais arquivos executáveis, dependendo da utilização ou não de Objetos Remotos. Um novo arquivo executável será criado para cada Objeto Remoto de tipo diferente, isto é, para cada diferente parametrização da meta-classe *remote*.

‡ Um *daemon* é um processo (normalmente ocioso) que executa em *background* esperando para realizar uma tarefa especificada.

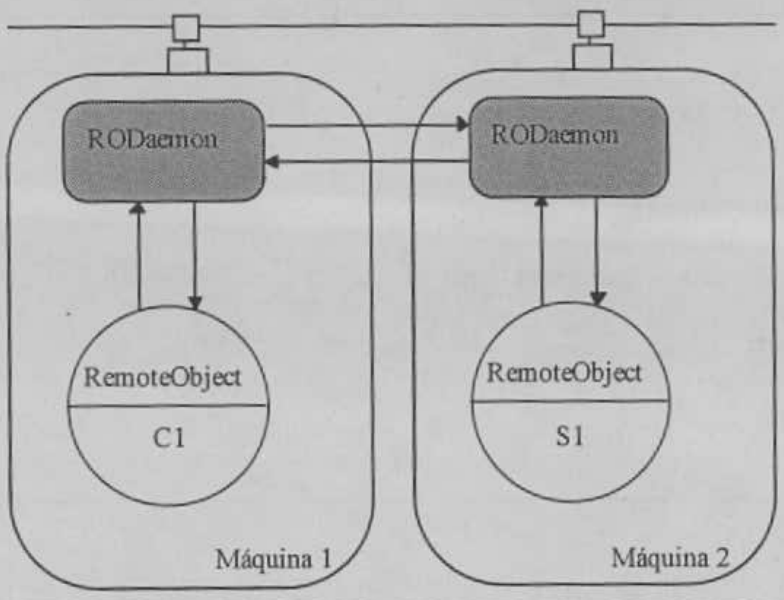

Figura 4: Execução de Objetos Remotos.

Um sistema suportando Objetos Remotos através do RTS deve executar *RODaemon* em cada máquina. Essa abordagem oferece um suporte melhor para o tratamento de erros a nível de sistema.

RODaemon

Daemon presente em todos os *hosts* do sistema; funciona como um gerenciador de Objetos Remotos e, além disso, mantém a tabela *ObjectTable* com informações a respeito de todos os Objetos Remotos que executam na sua máquina. As informações mantidas em *ObjectTable* permitem ao *RODaemon* gerenciar os objetos do sistema e controlar requisições remotas. A *ObjectTable* serve também como interface para o serviço de nomes utilizado pelo sistema.

As principais funções oferecidas pelo *RODaemon* são as seguintes:

- fornece uma interface para os objetos e para os demais *daemons* do sistema, possibilitando criar, destruir, registrar e procurar objetos no sistema;
- faz o controle das requisições remotas;
- é responsável pela destruição de Objetos Remotos vinculados. Caso um objeto seja destruído, uma busca é feita para verificar a existência de Objetos Remotos vinculados a este objeto;
- implementa um protocolo para comunicação entre os *daemons*, gerenciando entrada e saída de *daemons* no sistema, bem como o repasse de mensagens para os objetos (e respectivos *daemons*) em outros *hosts*.

RemoteObject

Biblioteca de funções a ser ligada a objetos gerados pelo compilador. Existem, na realidade, duas bibliotecas diferentes: *RemoteObjectST*, que oferece suporte para objetos *single-threaded*, e *RemoteObjectMT*, que suporta objetos *multi-threaded*. A funcionalidade de ambas é muito semelhante e, neste documento, serão diferenciadas somente onde for relevante.

As principais funções oferecidas por essas bibliotecas são:

- comunicação do objeto, tornando o objeto capaz de receber/enviar requisições remotas;
- comunicação com o *RODaemon*, respondendo a chamadas de gerenciamento/configuração;
- tratamento de exceções.

Além desses recursos, a biblioteca *RemoteObjectMT* oferece:

- suporte ao controle de concorrência;
- criação automática de *threads*.

UTILIZANDO O RTS

Para possibilitar a criação e o uso de Objetos Remotos por meio de uma linguagem de programação, o seguinte conjunto de operações para a interação com o RTS deve ser definido:

- *stubs* para a chamada de métodos remotos;
- informações de tipos em tempo de execução;
- tratamento de erros;
- linearização/deslinearização.

Essas operações podem ser oferecidas para linguagens de programação como C++ [Str91], utilizando uma biblioteca de funções definidas especificamente para cada linguagem.

CORBA (*Common Object Request Broker Architecture*) [OMG95] tem como uma de suas características, suporte ao uso de diferentes linguagens de programação em um sistema distribuído utilizando uma descrição em IDL (*Interface Definition Language*), feita pelo programador, para cada objeto utilizado por clientes distribuídos. Como na linguagem CmD o conceito de objetos remotos é oferecido na própria linguagem de programação, o seu compilador oferece diretamente a abstração de objetos distribuídos ao programador.

A implementação atual do RTS está sendo feita utilizando o compilador para a linguagem CmD. Nesse caso, algumas das funcionalidades necessárias deverão ser providas pelo próprio compilador, pois são particulares a cada classe compilada. Com a possibilidade de criação de Objetos Remotos no nível da linguagem, o compilador deverá gerar código necessário para chamar as funções do RTS.

Em CmD, instâncias de Objetos Remotos são geradas com a parametrização da classe pré-definida *remote* [Gon96].

Para que um Objeto Remoto em CmD possa atender a várias requisições simultaneamente, sua classe deve ser declarada como *threaded*. Os objetos cujas classes são declaradas sem a cláusula *threaded* são executados de forma estritamente sequencial.

Os métodos dentro de um objeto podem, por si só, representar uma *thread*. Isso é feito na linguagem CmD por meio de *métodos thread*. A chamada a um método definido com a cláusula `thread` retorna imediatamente após criar uma *thread* que realiza o processamento propriamente dito.[§] Isso corresponde a uma chamada assíncrona e possibilita que objetos possam ser "ativos", realizando tarefas mesmo na ausência de chamadas externas.

EXEMPLO DE APLICAÇÃO: PRODUTOR/CONSUMIDOR/LEITOR

Neste exemplo são implementados quatro objetos: *Buffer* (*buffer* remoto e *threaded*), *Produtor* (insere os elementos no *Buffer*), *Consumidor* (retira os elementos inseridos pelo *Produtor*) e *Leitor* (simplesmente verifica se um determinado elemento se encontra no *Buffer*).

Todos os objetos desse exemplo, ao iniciar, buscam por um objeto *Buffer* que exista no sistema (que tenha sido registrado no serviço de nomes); caso não o encontre, um objeto *Buffer* é criado.

A sincronização de acesso no objeto *Buffer* é feita através do mecanismo de controle de concorrência.

[§] Classes *threaded* não devem ser confundidas com métodos *thread*.

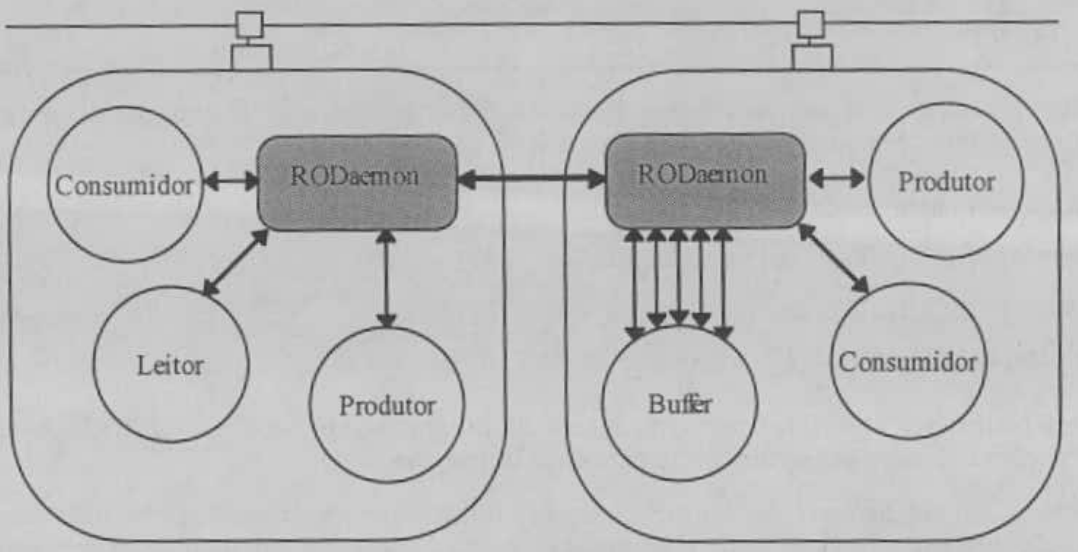

Figura 5: Exemplo de funcionamento do Produtor/Consumidor/Leitor

O código fonte das classes para este exemplo na linguagem CmD é apresentado no Apêndice A.

CONCLUSÃO

Nesse artigo descrevemos a especificação e a arquitetura de um sistema de suporte que oferece a abstração de Objetos Remotos. A implementação efetiva do RTS está sendo importante para verificar o acerto de algumas decisões que foram tomadas ao longo da sua especificação.

O desenvolvimento está sendo feito em plataforma UNIX com suporte a *multi-threads* (Solaris). Estão em funcionamento o *RODaemon* e a parte básica de *RemoteObject*; atualmente estamos fazendo a integração do compilador CmD com o RTS. O compilador CmD já gera código baseado no RTS, possibilitando o uso da noção de Objetos Remotos a nivel de linguagem de programação. Futuramente, planejamos implementar esse sistema utilizando uma plataforma CORBA/OMG (ORBIX).

AGRADECIMENTOS

Agradecemos a colaboração de Carlos A. Furuti e Alexandre P. Teles pela ajuda na revisão e contribuições para a versão final deste documento.

REFERÊNCIAS BIBLIOGRÁFICAS

[And83] Andrews, R. and Schneider, F. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys, 15* (1), pp. 3-43 (March 1983).

[Bir95] Birrell, A., Nelson, G. Owicki, S. and Wobber, E. Network Objects. *SRC Research Report* (December 1995).

[Cah93] Cahill, V., Baker, S., Horn, C. and Starovic, G. The Amadeus GRT – Generic Runtime Support for Distributed Persistent Programming. *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications* (1993).

[Dru87a] Drummond, R. e Liesenberg, H. A-HAND: Ambiente de desenvolvimento de *software* baseado em Hierarqias de Abstração em Niveis Diferenciados. *Anais do IV Encontro de trabalhos do Projeto ETHOS*, Petrópolis - RJ (abril 1987).

[Dru87b] Drummond, R. e Liesenberg, H. Requisitos para um Ambiente de Desenvolvimento de PROGRAMAS. *Anais do I Encontro IBM de Ciência e Tecnologia em Informática*, Rio de Janeiro - RJ (novembro 1987).

[Dru94] Drummond, R. e Gonçalves, C. Objetos Distribuídos em Cm. *Anais do XII Simpósio Brasileiro de Redes de Computadores*, Curitiba, PR, pp. 188-201 (maio 1994).

[Fur91] Furuti, C. *Um Compilador para uma Linguagem de Programação Orientada a Objetos*. Tese de mestrado, DCC-IMECC-UNICAMP (julho 1991).

[Gon94a] Gonçalves, C., Coutinho, B. e Drummond, R. Controle de Concorrência em Objetos. Relatório Técnico, Laboratório A-HAND, UNICAMP (abril 1994).

[Gon94b] Gonçalves, C. *Objetos Distribuídos* Tese de mestrado, DCC-IMECC-UNICAMP (agosto 1994).

[Gon96] Gonçalves, C., Teles, A. e Drummond, R. Desenvolvendo Aplicações Distribuídas em Cm. *Anais do XIV Simpósio Brasileiro de Redes de Computadores* (maio 1996).

[Han78] Hansen, P. B. Distributed Processes, a Concurrent Programming Concept. *Communications of the ACM,* 21 (11), pp 934-941 (1978).

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

[Hoa78] Hoare, C. Communicating Sequential Processes. *Communications of the ACM, 21* (8), pp 666-677 (1978).

[Lea93] Lea, R., Jacquemot, C. and Pillevesse, E. COOL: System Support for Distributed Programming. *Communications of the ACM, 36* (9), pp 37-46 (1993).

[OMG95] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.0* (July 1995).

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

[Tar96] Targa, C., Oliveira Filho, M., Gonçalves, C. e Drummond, R. Concorrência em Cm. *Anais do II Workshop do Projeto ASAP*, Fortaleza, CE (junho 1996).

[Tel93] Teles, A. *A Linguagem de Programação Cm*. Tese de mestrado, DCC-IMECC-UNICAMP (outubro 1993).

APÊNDICE A

O código fonte em CmD do exemplo Produtor/Consumidor/Leitor é apresentado abaixo:

```
class Produtor<>
   import Buffer;
   export constructor()
   {
      remote<Buffer> *rb;
      [
         rb -> Attach ("BUFFER_1");
         when default: {
               String host;
               cin >> host;
               rb = new (remote<Buffer> @ (host));
               rb->Registry("BUFFER_1");
         }
      ]
      int i = 0;
      while(1)
         rb->Insert(i++);
   }
```

```
class Consumidor<>
   import Buffer;
   export constructor()
   {
      remote<Buffer> *rb;
      [
         rb -> Attach ("BUFFER_1");
         when default: {
               String host;
               cin >> host;
               rb = new (remote<Buffer> @ (host));
               rb->Registry("BUFFER_1");
         }
      ]
      int i;
      while (1)
         i = rb->Remove();          //ignora resultado
   }
```

```
class Leitor<>
   import Buffer;
   export constructor()
   {
      remote<Buffer> *rb;
      rb -> Attach ("BUFFER_1");
      int i;
      while (1) {
         // gera i
         rb->IsThere(i);            //ignora resultado
      }
   };
```

```
threaded class Buffer<>
   region r=0;          //variavel de regiao c/ cardinalidade infinita
   const int SIZE = 12;
   int [SIZE] buf;      //buffer circular
   int inicio, fim;     //inicio == fim => buffer vazio
                        //(fim + 1) % SIZE == inicio => buffer cheio

   export constructor()
   {
      inicio = fim = 0;
   }

   export void Insert(int i)
   {
      with (r = 1; ((fim + 1) % SIZE) != inicio) {
         buf[fim] = i;
         fim = (fim +1) % SIZE;
      }
   }

   export int Remove()
   {
      int temp;
      with(r = 1; inicio != fim) {
         temp = buf[inicio];
         inicio = (inicio + 1) %SIZE;
         return temp;
      }
   }

   export int IsThere(int i)
   {
      int j;
      with (r) {
         for (j=inicio; (j!=fim)&& (buf[j]!=i); j = (j+1)%SIZE);
         return (buf[j] == i);
      }
   }

   export destructor()
   {
   }
```