JOSÉ RODRIGO BALZAN

UMA SOLUÇÃO REFLEXIVA PARA GERENCIAMENTO DE OBJETOS DISTRIBUÍDOS EM AURORA

FLORIANÓPOLIS – SC

JULHO DE 2001

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA DA COMPUTAÇÃO

JOSÉ RODRIGO BALZAN

UMA SOLUÇÃO REFLEXIVA PARA GERENCIAMENTO DE OBJETOS DISTRIBUÍDOS EM AURORA

Dissertação submetida a Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Luiz Carlos Zancanella,
Orientador

Florianópolis, Julho de 2001·.

UMA SOLUÇÃO REFLEXIVA PARA GERENCIAMENTO DE OBJETOS DISTRIBUÍDOS EM AURORA

JOSÉ RODRIGO BALZAN

Esta Dissertação foi julgada para a obtenção do título de Mestre em Ciência da Computação , Área de Concentração (Sistemas de Computação) e Aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____

Prof. Fernando A. Ostuni Gauthier, Dr. (coordenador)

_____

Prof. Luiz Carlos Zancanella, Dr. (orientador)

Banca Examinadora

_____

Prof. Philippe O. Alexandre Navaux, Dr. (UFRGS/RS)

_____

Prof. João Bosco Mangueira Sobral, Dr.

_____

Prof. Thadeu Botteri Corso, Dr.

“Nunca ande pelo caminho traçado, pois ele conduz somente até onde os outros foram”.

ALEXANDRE GRAHAN BELL

A meus pais, amigos e todos os professores que me ajudaram na elaboração deste projeto.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

AVLO	Access Virtual List Object
LOI	Local Object Invocation
RC	Reflexão Computacional
ROI	Remote Object Invocation
RPC	Remote Procedure Call
SD	Sistemas Distribuídos
SO	Sistemas Operacionais

RESUMO

Devido ao crescimento na utilização das redes de computadores e a necessidade de novas tecnologias no desenvolvimento de sistema, nota-se um avanço especial na área de orientação a objeto, onde idealistas estão criando novos significados para computação distribuída.

Ambientes como CORBA e DCOM são sinalizadores destas mudanças, tais ambientes aliados às linguagens orientadas a objetos como C++, Java e outras, estão motivando o uso crescente do modelo de objetos na solução de sistemas distribuídos.

Este trabalho buscou uma solução para o gerenciamento de objetos distribuídos no contexto de Aurora [Zan97], um sistema operacional modelado em termo de reflexão sobre objetos. O resultado obtido foi uma estrutura reflexiva, capaz de gerenciar a identificação e localização de objetos, de forma distribuída e totalmente transparente ao usuário e ao sistema.

ABSTRACT

Due to the growth in the use of the network computers and the need of new technologies in the system development, it is noticed a special progress in object oriented area, in which for vision people are creating new meanings for distributed computing.

The use of technologies like CORBA and DCOM are demonstrating these changes and along with objects oriented languages like C++, Java and other are motivating the growing use of the object model in the solution of distributed systems.

This work looked for a solution for object management distributed in the context of Aurora [Zan97], an operating system modeled in reflection over objects. The result was a reflexive structure, capable of manager the identification and location of objects, in a distributed manner and totally transparent to the user and the system.

1 INTRODUÇÃO

O processamento distribuído impõe requisitos de software, que caracterizam os ambientes de execução de uma forma particular, a interligação entre vários computadores executando de forma cooperativa, requer a presença de sistemas designados especialmente para gerenciar e integrar estes computadores.

Tais sistemas, conhecidos como sistemas distribuídos, podem ser implementados de modo que a comunicação entre os sistemas executando em máquinas diferentes ocorra através de troca de mensagens, de maneira confiável, eficiente e transparente ao usuário.

"Um sistema distribuído é um sistema que roda em um conjunto de máquinas sem memória compartilhada, máquinas estas que mesmo assim aparecem como um único computador para seus usuários" [Tan95].

Este conceito de SD pode ser visto como uma arquitetura de processamento paralelo, através da qual pode-se contornar limitações de capacidade de processamento dos computadores. O crescente aumento na complexidade de novas aplicações, exigindo um maior poder de processamento, vislumbra os sistemas de multicomputadores como uma alternativa capaz de proporcionar um maior poder computacional sobre as atuais arquiteturas existentes, oferecendo confiabilidade dos computadores e uma melhor relação custo/desempenho.

A história dos sistemas distribuídos teve origem com o desenvolvimento de sistemas para multi-usuários e redes de computadores por volta 1960 e foi estimulado pelo surgimento das estações de trabalho de baixo custo, pelas redes locais e pelo SO UNIX na década de 1970.

O processamento distribuído fornece toda a infra-estrutura necessária para a construção e operação efetiva de aplicações distribuídas, englobando tanto o software quanto o hardware necessário, para permitir que aplicações distribuídas sejam

desenvolvidas e possam ser executadas de maneira consistente, em um ambiente de rede distribuída e, eventualmente, heterogênea, ou em um ambiente centralizado.

1.1 Aspectos dos Sistemas Distribuídos

A vantagem potencial de um sistema distribuído sobre um sistema centralizado é sua alta confiabilidade. A possibilidade de distribuir a carga de trabalho em vários computadores possibilita a presença de falhas eventuais em computadores isolados sem que as demais sejam afetadas. Por exemplo, caso 5% (cinco por cento) das máquinas de um sistema distribuído apresente falhas, isso não necessariamente implicará na paralisação do sistema, ainda que o sistema possa apresentar um desempenho 5% (cinco por cento) inferior. Já em um sistema centralizado a presença de falhas pode comprometer 100% (cem por cento) do processamento. A tabela 1.1.1 resume algumas das vantagens dos sistemas distribuídos em relação aos sistemas centralizados.

Item	Definição
Economia	Os microprocessadores oferecem uma melhor relação preço/performance do que os centralizados (mainframes).
Velocidade	Um SD pode ter um poder de processamento maior que o de qualquer centralizado.
Confiabilidade	Se uma máquina sair do ar, o sistema como um todo pode sobreviver.
Distribuição inerente	Algumas aplicações envolvem máquinas separadas fisicamente.
Crescimento incremental	O poder computacional pode crescer em porção insignificante.

TABELA 1.1.1 Vantagens dos Sistemas Distribuídos.

Uma grande vantagem dos SD's com relação aos sistemas centralizados é a potencialidade de implementação de mecanismos de tolerância a falhas e de escalabilidade, principalmente devido à multiplicidade de recursos envolvidos.

Apesar dos sistemas distribuídos terem muitas vantagens em relação aos centralizados, existem alguns aspectos que devem ser analisados cuidadosamente, visto que o desenvolvimento, implementação de aplicações distribuídas requer a escolha apropriada tanto do sistema operacional quanto da linguagem de programação. A tabela 1.1.2 apresenta algumas desvantagens dos sistemas distribuídos em relação aos centralizados.

Outro problema potencial é o meio físico, tais como as redes de comunicação. O principal tema é a probabilidade de que sejam perdidas mensagens na rede, o que impõe a existência de softwares especiais para detectar a presença de falhas durante a troca de mensagens.

Item	**Definição**
Software	Até o presente momento não há muita disponibilidade de software para os sistemas distribuídos.
Ligação em rede	A rede pode sobrecarregar.
Segurança e confiabilidade	Os dados secretos também são acessíveis facilmente.

TABELA 1.1.2 Desvantagens dos Sistemas Distribuídos

Outro problema que o meio de comunicação pode ocasionar é a queda de performance do sistema devido à presença de interferências externas, ou mesmo a presença de erros no projeto da rede de comunicação.

1.2 Aspectos do Gerenciamento de Objetos

Os anos recentes estão caracterizados pelo surgimento de plataformas para gerenciamento de objetos distribuídos, tais plataformas podem ser encontradas em nível de serviços do sistema operacional, em nível de camadas de softwares entre o sistema operacional e aplicação, ou em nível de *runtime* das próprias linguagens.

Independente da localização da plataforma de gerenciamento de objetos distribuídos, o objetivo comum entre eles é a eficiência no gerenciamento dos objetos distribuídos e prover a propriedade de transparência de localidade, de modo que o cliente não necessite conhecer a localização física do objeto a fim de invocar um de seus métodos.

Comercialmente podem ser encontradas hoje diversas plataformas para gerenciamento de objetos distribuídos em nível de camadas de software entre o sistema operacional e a aplicação. Dentre os modelos mais conhecidos são: DCOM [Wan97] (Distributed Component Object Model) da Microsoft, a arquitetura CORBA [Omg95] (Comom Object Request Broker Arquiteture), o RMI [Dei01] (Remote Method Invocation), DCE [Ros92] (Distributed Computing Environment) e o menos conhecido DSOM [Jou01] (Distributed Systems Object Model) da IBM.

Todavia ao nível de sistema operacional, os sistemas comerciais hoje encontrados somente implementam o gerenciamento de objetos ao nível de serviços, enquanto o suporte a gerencia de objetos distribuídos a nível conceitual do sistema operacional é somente encontrado em ambientes de pesquisas e acadêmicos.

Apertos [Yok92] desenvolvido no laboratório de computação Sony,é um exemplo de sistema operacional que suporta o gerenciamento de objetos distribuídos a nível conceitual.

O projeto Aurora, escopo deste trabalho, apresenta uma arquitetura reflexiva capaz de suportar o gerenciamento de objetos distribuídos a nível conceitual. Inserido neste escopo, este trabalho buscou uma solução reflexiva, capaz de gerenciar a identificação e localização de objetos, de forma distribuída e totalmente transparente ao usuário e ao sistema.

Este trabalho está dividido em 7 capítulos. O capítulo 2 apresenta fundamentos teóricos sobre objetos distribuídos. O capítulo 3 sobre as plataformas de desenvolvimento de aplicações distribuídas. O capítulo 4 apresenta uma breve introdução sobre o Sistema Aurora, e Reflexão Computacional. No capítulo 5 são apresentados os resultados alcançados no desenvolvimento deste trabalho. O capítulo 6 apresenta a validação do modelo implementado. E finalmente no capítulo 7 são apresentadas as considerações finais e trabalhos futuros.

2 OBJETOS DISTRIBUIDOS

Os conceitos Orientação a Objetos começaram a se solidificar nos anos 60/70, sistemas comerciais tornaram-se disponíveis na segunda metade dos anos 80, quando Smalltalk e C++ tornaram-se realidade, vindo logo a seguir os bancos de dados e as técnicas OO, das quais há hoje uma grande variedade. Mais a título de curiosidade, pode-se dizer que o termo OO foi criado nos anos 70, durante o desenvolvimento do SmallTalk, no centro de pesquisas da Xerox em Palo Alto, Califórnia.

Objetos são unidades de dados com as seguintes propriedades e características [Cha00]:

- *typed and self-contained* : Cada objeto é uma instância de um tipo, que define um conjunto de métodos (assinaturas) que podem ser invocadas para operar sobre o objeto.
- *encapsulated* (encapsulamento) : A única maneira de operar um objeto é por seus métodos; representação/implementação não pode ser visualizada
- *dynamically allocated/destroyed* : Objetos são criados conforme a necessidade, e destruídos quando não são mais utilizados, eles existem fora de qualquer escopo do programa.
- *uniquely referenced* (exclusiva referencia) : Cada objeto é identificado exclusivamente durante sua existência por um nome/OID/referência/ponteiro que pode ser acessado/armazenado/compartilhado.

Computação distribuída estende a programação de sistemas orientados a objetos, permitindo que objetos sejam distribuídos por uma rede heterogênea. Estes objetos podem ser distribuídos em computadores distintos ao longo de uma rede, enquanto "vivendo" dentro do próprio espaço de endereço deles fora de uma aplicação, fazendo com que eles comportam-se locais a uma aplicação.

Contudo, como o nosso enfoque é mais dirigido para as questões de gerenciamento de objetos distribuídos, passaremos a tratar os tópicos mais importantes

para a computação distribuída dentro de uma perspectiva prática, tratando detalhadamente de aspectos técnicos relevantes para a sua desmobilização.

Os objetos distribuídos e aplicações distribuídas podem se comunicar através de troca de mensagem ou memória (variáveis) compartilhada. Deste modo aplicações distribuídas são escritas freqüentemente para tolerância à falhas, disponibilidade e compartilhamento de recursos. Tipicamente, os vários processos da aplicação são juntados livremente, isto é, eles comunicam-se sem freqüência.

Objetos distribuídos consistem uma alternativa para SD pelo fato de os componentes de uma aplicação distribuída poderem se comunicar através de mensagens (métodos) endereçadas. São visto como armazenadores de códigos que executam uma determinada tarefa quando são invocadas através de uma chamada de seus métodos. Sua principal característica refere-se a sua localização, de forma que podem se hospedar em uma única máquina, ou distribuídos em máquinas dispersas ao longo de uma rede de computadores (LAN, WAN, e Internet). Em coletividade, esses objetos são capazes de executar funções para um SD. Objetos estarão efetuando uma comunicação entre diferentes processadores, ou seja, os objetos estão localizados em máquinas diferentes.

Em SD os objetos então dispersos ao longo dos nós de uma rede heterogênea de computadores. Devemos considerar que um sistema deste nível torna o mecanismo de implementação de nível mais alto, de maneira que torne os serviços de comunicação distribuída mais flexíveis.

Podemos diferenciar os objetos clássicos dos objetos distribuídos através das seguintes características:

- **objeto clássico** possui propriedades e métodos e é gerado através de uma linguagem de programação como C++, Smalltalk, entre outras. Esses objetos fornecem reusabilidade de código através de herança, encapsulamento e polimorfismo, propriedades fundamentais da teoria orientada a objetos. Contudo, esses objetivos só vivem dentro de um

programa; apenas o compilador que os criou conhece a sua existência. O mundo externo os desconhece e não tem forma de acessá-los.

- **objeto distribuído** também é uma classe que pode publicar (tornar disponível) tanto suas propriedades quanto os seus métodos, mas a linguagem e compilador usados para criar objetos distribuídos são totalmente transparentes para a implementação desses. Esses objetos têm uma interface definida onde os compiladores geram um código a mais para serem acessados por outros objetos de forma totalmente transparente, isto é, o invocador desconhece o local do objeto invocado, o sistema operacional que esse executa, podendo estar fisicamente na mesma máquina (processo) ou em alguma máquina remota.

Um dos fatores críticos para a implementação de um sistema com componentes distribuídos é a transparência da sua localização física, podendo operar num processador local, num processo diferente ou num processador remoto. Porém, um componente não pode operar no vácuo e por isso precisa de uma *middleware* (ponte de comunicação) que promova a sua comunicação com outros componentes distribuídos.

2.1 Características de Sistemas Distribuídos

Um dos critérios para classificar sistemas distribuídos é quanto à maneira que é realizada a comunicação entre os processadores. Pode ser feita uma divisão entre SD de modo que a comunicação seja por “broadcast”, e outros onde a comunicação é interligada ponto-a-ponto.

Em comunicação realizada através de “broadcast”, todos os processadores do sistema podem se comunicar entre si, por meio de uso de canais compartilhados por cada um. A figura 2.1 mostra o modelo do sistema onde *n* processadores compartilham *m* canais de comunicação através de “broadcast”.

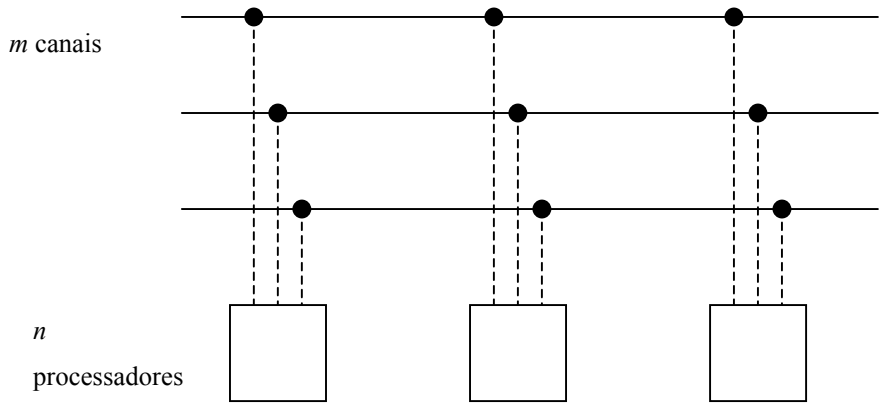


FIGURA 2.1 Sistema Distribuído por "Broadcast"

No modelo acima a troca de mensagens é feita por "broadcast" por meio de canais compartilhados, (mas deve tomar cuidado quando dois processadores tentam usar o mesmo nó compartilhado simultaneamente, soluções de conflito no uso da rede são resolvidos no nível de software).

Um outro modelo de comunicação que é bastante conhecido é o que chamamos de ponto-a-ponto. Onde são representados por grafos conexos (ligados) e cada nó representa um processador e cada arco entre os nós representa um canal de comunicação exclusivo entre eles. Através do uso de exclusividade entre os canais elimina-se o problema de colisões, mas a rota percorrida é mais longa, pois mensagens enviadas para um determinado nó acabam passando por outros nós para poderem chegar ao seu destino. A figura 2.2 ilustra o modelo distribuído ponto-a-ponto.

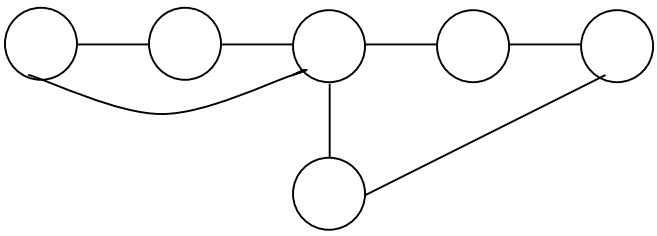


FIGURA 2.2 Modelo Ponto-a-Ponto

São seis principais características chaves responsáveis pela utilidade em SD. Estes recursos são:

- **Compartilhamento de Recurso**: O termo "recurso" é bastante abstrato, mas caracteriza melhor a linha de coisas que podem ser compartilhadas de forma útil em um sistema distribuído. Estende componentes de hardware assim como discos e impressoras para entidades de software-definido como arquivos, janelas, bancos de dados e outros objetos de dados. Usuários tanto de sistemas centralizados e distribuídos são acostumados com benefícios de recurso compartilhado que eles podem negligenciar a significação deles facilmente. Os recursos de um computador para multi-usuários são normalmente compartilhados entre todos seus usuários, entretanto usuários de rede, em estações mono-usuário e computadores pessoais não obtêm os benefícios de recurso compartilhado automaticamente.
- **Sistemas Abertos** (openness): A franqueza de um sistema de computador é a característica que determina se o sistema pode ser estendido em diferentes formas. Um sistema pode estar aberto ou fechado com respeito a extensões de hardware - por exemplo, a adição de periférico, memória ou comunicação de interface - ou com considerações a extensões de software - a adição de características de sistema operacional, protocolos de comunicação e serviços de recurso compartilhado. A nudez de sistemas distribuídos é principalmente determinada pelo grau ao qual podem ser acrescentados novos serviços de recurso compartilhado sem interrupção ou duplicação de serviços existentes. Franqueza é alcançada especificando e documentando as interfaces de software fundamentais de um sistema e as fazendo disponível para programador de software. Em uma palavra, são as interfaces *published*. UNIX é um exemplo de um sistema aberto. Inclui uma linguagem de programação C, que permite para os programadores terem acesso a todas as capacidades (recursos) administradas pelo sistema operacional.
- **Concorrência**: Quando vários processos existem em um único computador nós dizemos que eles são executados concorrentemente (ao mesmo tempo). (Desde que, cada processo só exista para a duração de uma execução de

programa, coexistência implica concorrência de execução.) Se o computador é equipado com um único processador, isto é alcançado intercalando a execução de partes de cada processo. Se o computador tiver processos de *N*, até os processos *N* podem ser executados simultaneamente (quer dizer, em paralelo), alcançando uma melhoria no rendimento computacional. Em sistemas distribuídos há muitos computadores, cada com um ou mais processadores. Se há *M* computadores em um sistema distribuído com um processador cada, então até *M* processos podem executar em paralelo, contanto que os processos fiquem situados em computadores diferentes. Em um sistema distribuído, baseado no modelo de recurso compartilhado descrito acima, oportunidades para execução paralela acontecem por duas razões:

1. Muitos usuários invocam comandos simultaneamente ou interagem com programas de aplicação;
2. Muitos servidores executam processos concorrentemente, cada um respondendo requisições diferentes de processos de cliente.

- **Tolerância à falha**: Sistemas de computador às vezes falham. Quando falhas acontecem em hardware ou software, programas podem produzir resultados incorretos ou eles podem parar antes de completarem sua computação. O design de tolerância à falha em sistema de computador está baseado em duas aproximações, ambas devem ser desenvolvidas para controlar cada falha:
  1. *redundância de hardware*: o uso de componentes redundantes;
  2. *recuperação de software*: o desenvolvimento de programas para recuperar falhas;

  Para produzir sistemas que são tolerantes à falha de hardware, são empregados freqüentemente dois computadores interconectados por uma única aplicação, um deles que age como uma máquina auxiliar para o outro. Em sistemas distribuídos, pode ser planejada redundância a um fino grão - servidores individuais que são essenciais para operação de aplicações críticas, que podem ser replicadas.

- **Escalabilidade:** Sistemas distribuídos operam efetivamente e eficientemente em muitas escalas diferentes. O menor sistema distribuído praticável provavelmente consiste em dois *workstations* e um servidor de arquivo, considerando que um sistema distribuído construído ao redor de uma única rede de local, e podem conter vários, centenas de *workstations* e muitos servidores de arquivos, servidores de impressão e outros servidores de propósito especiais. São interconectadas freqüentemente várias redes locais para formar *internetworks*, estes podem conter muitos (milhares) computadores que formam um único sistema distribuído, permitindo que recursos podem ser compartilhados entre todos eles. A demanda para escalabilidade não é só um problema de hardware ou desempenho da rede, mas sim de planejamento no desenvolvimento do sistema distribuído. Em sistemas centralizados certos recursos compartilhados; - memória, processadores, barramento de entrada/saída - é limitado o fornecimento e não pode ser replicado indefinidamente. Em sistemas distribuídos, fornece acesso a limitação sobre alguns desses recursos e são automaticamente removidos.
- **Transparência**: Transparência está definida como a forma de encobrimento para o usuário e o programador da aplicação a separação de componentes em um sistema distribuído, de forma que o sistema é percebido de modo geral como uma coleção de componentes independentes. O aspecto da transparência se caracteriza por fazer com que se torne invisível para o usuário o fato de muitas máquinas trabalharem juntamente, fazendo com que ele veja o sistema como se rodasse em uma única máquina, pense que tem apenas um único processador, um único local de armazenamento. O ambiente distribuído se torna transparente para o usuário, facilitando seu entendimento e manuseio. A *International Standards Organization's* (ISO) e *Reference Model for Open Distributed Processing* (RM-ODP) [ISO92] identificaram oito formas de transparência:

1. Transparência de Acesso: permite que informação dos objetos local e remoto seja acessada usando a mesma operação.
2. Transparência de Localização: faz com que os usuários não precisem saber onde os recursos se encontram.
3. Transparência à Concorrência: permite o compartilhamento simultâneo dos mesmos recursos.
4. Transparência à Replicação: permite múltiplas instâncias de objetos, aumento da segurança e performance, sem com que o usuário tenha conhecimento das replicas.
5. Transparência à Falha: permite o encobrimento das falhas, enquanto permite usuários e aplicações completarem suas tarefas.
6. Transparência à Migração: permite que os objetos possam mudar de lugar livremente sem que precisem mudar seus nomes.
7. Transparência de Performance: permite o sistema ser reconfigurado para melhorar o desempenho conforme as cargas variam.
8. Transparência de Escalonamento: permite o sistema e aplicações se expandirem em escala, sem mudança da estrutura do sistema ou dos algoritmos de aplicação.

2.1.1 Remote Procedure Call – RPC

O modelo RPC é baseado numa única proposta: fazer com que procedimentos individuais de uma aplicação possam ser processados em qualquer parte da rede. Foi desenvolvido nos anos de 1980. E permite que um programa procedural execute uma função que se encontre em um outro computador.

Geralmente, os computadores que compõem os SD não compartilham memória principal e suas comunicações. É comum, neste caso, o uso de mecanismos como troca de mensagens e RPC. Neste esquema de troca de mensagens, um processo cliente, ao pedir um serviço envia uma mensagem para um processo servidor e fica aguardando por uma resposta, como mostrado na figura 2.3. A comunicação envolvida na troca de

mensagens traz com ela muitas questões que complicam a implementação desse esquema, como. Por exemplo, o controle de fluxo de dados, a perda de mensagens e a utilização eficiente de *buffers*.

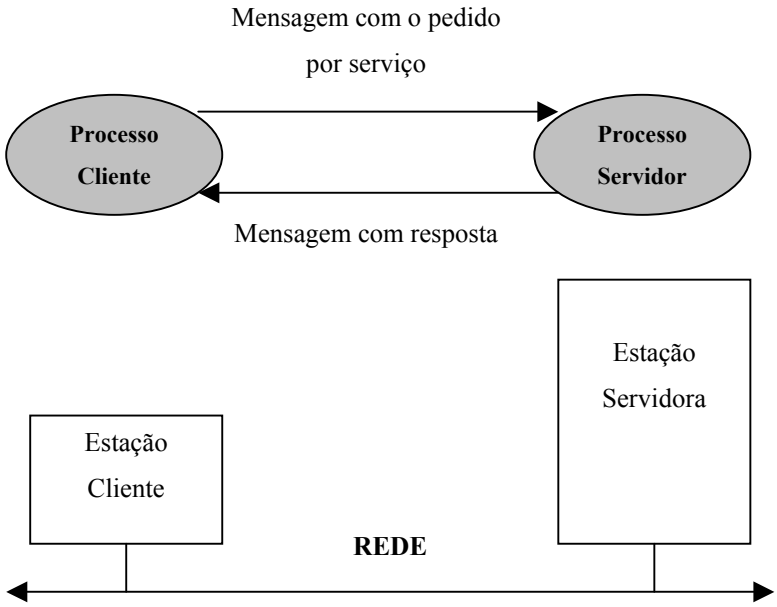


FIGURA 2.3 Modelo de Comunicação com troca de mensagens

Um dos objetivos do RPC foi permitir aos programadores se concentrarem nas tarefas exigidas de um aplicativo chamando funções e, ao mesmo tempo, tornar transparente ao programador o mecanismo que permite que as partes do aplicativo chamado se comuniquem através de uma rede. RPC realiza todas as funções de rede e *ordenação* (*marshalling*) *dos dados* (isto é, empacotamento de argumento de função e valores de retorno para transmissão através de uma rede). Uma desvantagem de RPC é que o mesmo suporta um conjunto limitado de tipos de dados simples. Portanto RPC não é adequado para passar e retornar objetos.

O mecanismo RPC integra o protocolo *Requisita*/*Responde* RR usado para comunicação cliente/servidor com as linguagens de programação convencionais, permitindo clientes comunicarem-se com servidores através de chamadas de procedimentos.

A implementação de RPC envolve a transferência de argumentos do cliente para o servidor e dos resultados do servidor para o cliente (mostrado na figura 2.2.2). A codificação e decodificação destes dados são chamados de *marshalling*, isto é, o processo de pegar uma coleção de itens de dados e montar em uma forma adequada para transmissão, envolvendo a serialização dos dados em uma sequência de itens básicos e a tradução dos mesmos em uma representação externa. Procedimentos de *marshalling* devem ser feitos pelos *stubs*, tanto no cliente quanto no servidor.

2.1.2 Objetos Duais

Consiste em que cópias dos objetos existam em vários locais diferentes, permitindo que o sistema proporcione completa funcionalidade mesmo na ocorrência de falhas em máquinas onde se encontram objetos hospedados. Isto é verdadeiro visto que a falha em uma máquina, somente resulta na não disponibilidade das cópias que residem naquele local, enquanto cópias residentes em outras máquinas continuam aptas a atender ativações de métodos daquele objeto.

Uma diversidade simples para este esquema, porém limitante, é permitir que somente objetos imutáveis sejam duplicados. Visto que os estados dos objetos imutáveis somente podem ser examinados pelos clientes, nunca modificados, este tipo de objetos pode ser duplicado sem preocupar-se com consistência e sincronização.

O provedor de endereços (objetos) deve trabalhar implicitamente através de objetos exportados aos clientes e têm que ser administrado pelo hospedeiro do objeto. Quer dizer, quando chamar um método, o objeto invocado tem que localizar automaticamente o caminho para o provedor (máquina). Serviços para processo, espaço de endereço e administração de memória, são endereçados implicitamente por objetos. Em particular, esta aproximação é aplicável a qualquer tipo de objeto e não somente limitado para objetos de sistema.

Um objeto dual consiste em duas partes relacionadas, mas fisicamente separadas: o protótipo e pelo menos uma semelhança. Conceitualmente, o protótipo é um objeto móvel. O protótipo possui os estados privados e públicos.

2.2 Identificação e Localização de Objetos

Um sistema baseado em objetos deve prover a propriedade de transparência de localização, de modo que o cliente não necessite conhecer a localização física de um objeto remoto, a fim de invocar um de seus métodos. Quando uma ativação é realizada, o sistema deve determinar qual objeto está sendo invocado e qual a localização física do mesmo e entregar a ele a requisição, fazendo deste modo parecer para o cliente que o objeto está fisicamente em sua máquina. Importante observar que todo objeto é identificado unicamente pelo sistema e esta identificação é única e pertence somente a ele, e ela não pode ser alterada enquanto o objeto estiver "vivendo". Quando a migração do objeto for desejável, é necessário que o mecanismo provido para localização seja flexível a ponto de permitir que objetos possam se mover de um local físico para outro.

1. Uma forma é embutir a localização do objeto junto com sua identificação. Desta forma, quando uma invocação é realizada, o sistema simplesmente examina a identificação do objeto a fim de determinar o local físico onde o objeto reside. Este esquema é eficiente, entretanto, apresenta a restrição de não permitir que um objeto migre, visto que requer a troca da identificação do objeto, conseqüentemente, um objeto é fixo a um local físico particular ao longo de sua vida útil.

2. Uma segunda forma é o esquema de um servidor de nomes distribuído, onde o sistema cria um grupo de objetos servidores de nomes que cooperam entre si, de modo à coletivamente conter informações atualizadas sobre a localização de todos os objetos do sistema. Existem duas variações para esta representação:

- Na primeira, todos os servidores de nomes mantêm uma coleção completa das informações de localização, de modo que qualquer servidor pode responder uma requisição de localização. A dificuldade que se apresenta, é que as informações mantidas nos servidores podem apresentar-se ligeiramente fora de sincronismo, visto que o processo de atualização não é uma operação atômica e a manutenção da consistência da informação entre os múltiplos servidores de nomes pode ser complexa.

- A segunda variação, informações parciais são mantidas em cada servidor, de modo que se uma requisição de localização não possa ser atendida por um servidor, ela é repassada para outro. A maior dificuldade deste modelo é que no mínimo um servidor deve ser notificado toda vez que um novo objeto é criado ou migrado de um local físico para outro.

3. Outra forma de implementar mecanismos para localização de objetos é o esquema conhecido como *cache*/*broadcast.* Uma pequena *cache* é mantida em cada local físico (estação ou memória) para armazenar as localizações conhecidas dos objetos remotos mais recentemente referenciados. Quando um cliente realiza uma invocação remota, a *cache* é examinada para determinar se ela tem alguma informação a respeito da localização do objeto invocado. Caso a localização seja conhecida, a invocação é enviada para o local especifico onde o objeto se encontra fisicamente, caso nenhuma informação sobre a localização esta contida na *cache*, uma mensagem é distribuída para a rede questionando a localização do objeto. Toda estação (ou nó) que receber a requisição, realiza uma pesquisa interna buscando informações sobre o objeto especificado. Se o objeto é encontrado, uma mensagem retorna para a máquina de origem que atualiza sua *cache*. Este esquema pode ser muito eficiente, visto que as localizações dos objetos podem ser encontradas localmente. Ela é também flexível, visto que permite um objeto ser movido de um local físico para outro, evitando o custo de ter que notificar outras estações ou servidores de nomes distribuídos. Um problema com este esquema, entretanto, é que requisições *broadcast* irão

perturbar toda rede, interrompendo todas as estações mesmo que somente uma delas esteja diretamente envolvida com a localização requisitada.

Objetos são usados para encapsular os serviços do sistema. Portanto, um cliente tem que saber destes objetos para poder adquirir acesso ao serviço. Um serviço pode estar logicamente implementado por um ou mais objetos ativos. Suponha que a arquitetura de hardware compreende um numero de disco I/O. Cada nó do sistema está equipado com um disco, estes discos são objetos ativos do mesmo tipo. Eles são instancias da mesma classe, enquanto provendo o mesmo conjunto de (serviços) funções.

A figura 2.4 mostra uma visão abstrata do ambiente de sistema. Apresenta que sistemas paralelos e aplicações distribuídas são apoiados pelo sistema operacional que em troca é apoiado por uma arquitetura de hardware. O sistema operacional implementa mecanismos para a troca de mensagem e primitivas para a sincronização, de forma a dar suporte a diferentes paradigmas de comunicação entre aplicações.

A seguir são apresentados alguns possíveis métodos para localização de objetos em ambiente de SD.

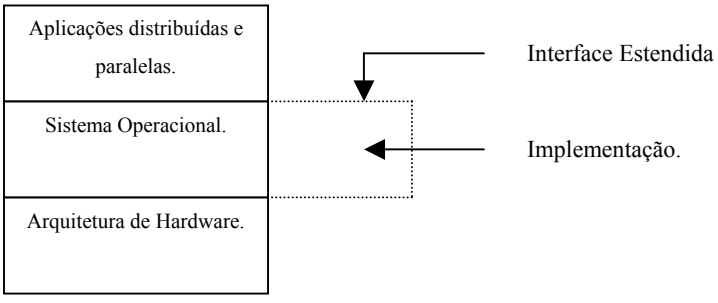


FIGURA 2.4 O Ambiente do Sistema

2.2.1 Endereçamento Implícito Direto ao Objeto

O esquema de ativação direta é tradicionalmente empregado nos sistemas que suportam o modelo de objetos passivos, para manejar o intercâmbio entre os objetos. No modelo de objetos passivos, um processo único é responsável pela execução de todas as operações associadas com uma ativação.

Para superar o problema de endereçamento saliente por exemplos de múltiplas instancias sobre o mesmo servidor de classe, uma solução possível seria acrescentar a cada função parâmetros que exclusivamente identificam um servidor. Embora direta esta abordagem tem várias desvantagens. A necessidade por prover o endereço de servidor explicitamente viola o princípio de transparência. Um programa de usuário se torna responsável por determinar e ficar vigiando esse endereço. O endereço conhecido do cliente pode mudar se um servidor migrar. Neste caso, o programa usuário tem que controlar exceções de comunicação para obter o endereço do novo servidor. Desde um sistema alterável dinamicamente, o programa usuário também tem que ativar explicitamente carregando com incremento de serviço de funções para obter conhecimento do endereço de servidor.

Não somente o programa usuário fracamente apoiado por um esquema de endereçamento, se não o servidor, o parâmetro adicional no endereço não faz sentido de nenhuma maneira. Quando o servidor for executar um serviço de função, o pedido de serviço deve obviamente ter sido transmitido de forma sucedida. O parâmetro de endereço teve uma função somente na transmissão, não na execução de funções de serviço, assim um será ignorado pelo servidor. Compatibilidade pode ser alcançada no modelo de RPC [Nel82].

Visto que o serviço invocado em um RPC ou *Remote Object Invocation* (ROI), mostra que este esquema de endereçamento também introduz *overhead* impedindo a programabilidade. O endereço do servidor precisa ser provido pela entidade no nível-usuário e será transportado em cima da entidade em nível-sistema. Ocorrendo uma

cópia desnecessária de parâmetros, devem ser executadas operações de *marshaling*/*unmarshaling*.

Transporte de forma *marshaling*/*unmarshaling*, sobre o parâmetro do endereço pode ser evitado por dois conjuntos de serviços de funções. A assinatura sobre as funções contidas no primeiro conjunto de exibições fixando o endereço no parâmetro. Este parâmetro não é incluído na assinatura correspondente das funções contida no segundo conjunto. Sempre são tidos acessos as funções no primeiro conjunto em *Local Object Invocation* (LOI). Eles usam o endereço do parâmetro direto para o ROI à função correspondente no segundo conjunto. Porém, esta aproximação torna a manutenção de software mais complexa.

2.2.2 Único Identificador

Um identificador refere-se a um objeto ativo, o "secretário" que é capaz de extrair/unificar o estado público. Visto que extração e unificação ocorre em uma base ROI, o identificador representa o endereço final de comunicação sobre o "secretário" envolvido.

Endereço final de comunicação é interpretado pelo núcleo. Devem ser projetados tais endereços com as metas seguintes em mente. Primeiro, eles devem ser fáceis de usar para ambos o núcleo e as entidades que confiam no núcleo. Segundo, estes endereços têm que fazer localização direta do objeto. Isto requer própria estruturação do endereço de preferência que o entendimento como um valor numérico.

Obviamente, estes objetivos são um pouco contraditórios. Fazendo um endereço final de comunicação do sistema inteiro só é possível se endereços muito grandes e escassos forem usados. O uso de endereços pequenos e confiando em um serviço de sistema central que assegura a geração de identificadores não ambíguos é impraticável. Em uma máquina de memória distribuída, este serviço pode ser pedido somente em uma

passagem de mensagem, a qual espera que os endereços finais de comunicação se dirigem do servidor correspondente já é conhecido.

2.2.3 Identificação Múltipla

Analisaremos nos próximos parágrafos de forma simplificada como que o sistema Apertos gerencia a localização de objetos distribuídos. Veremos a estrutura que o Apertos usa para poder fazer a identificação dos objetos [Yok92a].

O sistema possui um esquema de nomeação hierárquico, este esquema de nomeação hierárquico não assume uma única *IDs* para os *hosts* e os constrói localmente exclusivo. Isso traduz as representações de *IDs* e os endereços sempre são transmitidos. Isto assegura a exclusividade de *IDs* em todo contexto de nome enquanto são dadas representações curtas de *IDs* e endereços dados a objetos logicamente-próximo. O sistema de rede deve garantir uma única identificação para os *hosts,* pois o esquema de nomeação assegura para seu funcionamento que os *hosts* possuem exclusividade de suas identificações.

Cada contexto de nome concede Id locais (LIDs) para o objetos instanciados e endereços locais (LADs) para os objetos. LIDs e LADs são exclusivos dentro do contexto de nomeação. O tempo de instanciação, a LID e LAD de um objeto são iguais. Considerando que o contexto de nome também é um objeto, tem um LAD que é único dentro do contexto. Há somente uma raiz que nomeia contexto que é a raiz da hierarquia de contexto de nome em um sistema. A raiz pode mudar conforme o sistema se expande.

Cada objeto no sistema tem a noção do contexto original de nomeação no qual o objeto está instanciado, e o contexto nomeação atual no qual o objeto está atualmente. O objeto ID (OID) do objeto representa seu contexto de nome original e o LID, e o endereço do objeto (OAD) sobre o objeto representa seu contexto de nome atual e o LAD. OIDs e OADs têm representação relativa vista do objeto que memoriza o OIDs

ou OADs. A interpretação de OIDs e OADs é o sentido sobre eles que é exclusivo dentro do sistema. A interpretação do OID sobre um objeto nunca muda até que ele é eliminado. Se uma mensagem ou um objeto é transferido, são traduzidos o OIDs e OADs deles para manter a interpretação do OIDs e OADs constante.

A unidade de comunicação é uma mensagem. O receptor de uma mensagem é especificado por seu OID. Os gerentes dos contextos de nomes tomam cuidado sobre a relação entre o OID e o OAD. Usuários não têm que saber a respeito de OADs.

Cada objeto se recorda de seu OID. O gerente de contexto de nomes que também é um objeto possui mais informações. Um OID é para identificar um objeto. O formato de um OID é :

m: l1.l2.,,,.ln

onde *m,n* são números inteiros positivos e cada *li* é um LID. A interpretação de um OID está definida como segue:

A interpretação sobre OID m : l1.l2.,,,.ln no contexto de nomes S é ;

L1.L2.,,,.Ln-m.l1.l2.,,,.ln

onde *L1.L2.,,,.Ln* é a sequencia de LIDs da raiz atual que contexto de nomes de S.

A figura 2.5 mostra um exemplo do contexto de nomeação com estrutura hierárquica. Círculos de contextos de nomes hierárquicos representam objetos ou nomeação. Os números representam LIDS. Os LIDs são diferentes um do outro para simplicidade. As conexões exceto o ramo da arvore são omitidas. O *OID* do objeto **B** para **A** é 1:32, e esse sobre **D** é 3:2.21.52. Os *OIDs* de **B** e **D** para **C** são 2:11.32 e 3:2.21.52 em ambos **A** e **C**, e **D**'s é 2.21.52.

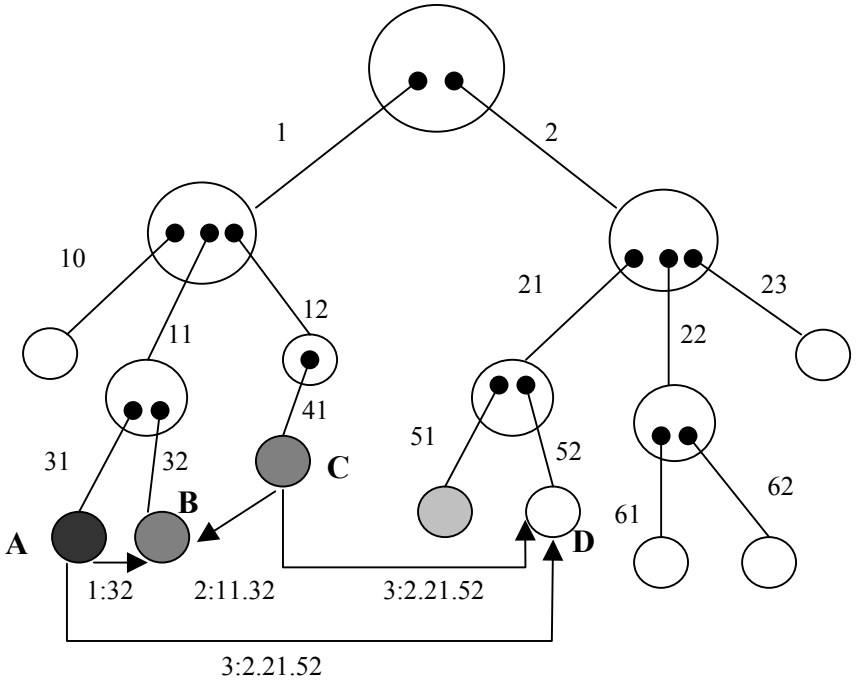


FIGURA 2.5 Estrutura Hierárquica do Sistema Apertos

3 PLATAFORMAS PARA DESENVOLVIMENTO DE APLICAÇÕES DISTRIBUÍDAS

No ambiente da computação distribuída estão os objetos distribuídos, que vão além das barreiras tradicionais impostas por linguagens de programação, espaço de endereçamento de processos, sistemas operacionais ou plataformas de hardware. Cada um desses obstáculos envolve considerações de grande complexidade, e o uso de objetos é a melhor forma de abstrair a lidar com esta complexidade. Os objetos fornecem uma forma interessante para organizar a complexidade nos modernos sistemas operacionais e simplificam o processo de distribuição das aplicações. Com sua combinação natural de dados e comportamento e a separação explícita da interface de sua implementação, os objetos formam uma solução ótima para a distribuição de dados e processos. A tecnologia de orientação a objetos fornece muitos dos meios que disponibilizam sistemas distribuídos de fato [Tei96].

Os Objetos distribuídos são organizados em uma arquitetura de três camadas, incluindo um Modelo de Objetos, um Modelo de Componentes e as Aplicações e Sistemas.

Modelos de Objetos incluem as arquiteturas: DCOM (Distributed Component Object Model), SOM/DSOM (Distributed System Object Model) da IBM, CORBA (Common Object Request Broker Architecture), JAVA/RMI (Java/Remote Method Invocation) e DCE (Distributed Computing Environment). Estão dirigidos para aplicações conectadas na rede onde o único critério que se deve verificar é se a aplicação realmente existe na rede. Essas arquiteturas são todas baseadas no modelo cliente/servidor, onde o programador necessita da implementação da aplicação servidora como cliente para sua funcionalidade.

Os Modelos de Componentes incluem o OLE (Object Linking and Embedding), o OpenDoc da Apple e o OpenStep da Next, ou o CORBA CF do consórcio CORBA. Esses componentes baseiam-se no modelos de objetos da camada anterior, de modo que possam implementar funções como a ligação e o aninhamento, e ativação local e outras funções relacionadas com objetos.

As Aplicações e Sistemas representam o módulos desenvolvidos pelos usuários finais, através das facilidades fornecidas pelas ferramentas disponíveis nas camadas anteriores.

A seguir vamos analisar algumas dessas estruturas de Modelo de Objetos, que são bastante utilizadas no desenvolvimento de aplicações distribuídas.

3.1 DCOM (Distributed Component Object Model)

DCOM foi desenvolvido pela Microsoft Corporation e está correntemente sendo testado no sistema operacional NT4.0, Win 95/98 da Microsoft.

Suporta objetos remotamente sobre um protocolo chamado ORPC (Object Remote Procedure Call). Esta camada de ORPC é construída em cima do RPC de DCE e interage com os serviços de COM (Component Object Model). Um servidor de DCOM é um corpo de código que é capaz de agir em cima de objetos de um tipo particular, em runtime. Cada servidor de objeto DCOM suporta múltipla interface, cada um representando um diferente comportamento do objeto. Um cliente DCOM chama os métodos expostos de um servidor de DCOM, adquirindo um ponteiro a uma das interfaces do objeto servidor. O objeto cliente então inicia chamada a métodos do objeto servidor pelo ponteiro de interface adquirido como se o objeto servidor residisse no espaço de endereço do cliente. DCOM é usado principalmente na plataforma Windows. Companhias como *Software AG* provêem implementação de serviço COM através de EntireX para UNIX, Linux e plataformas mainframe, *Digital* para a plataforma de VMS, Microsoft Windows e plataformas Solaris.

O DCOM é um protocolo que possibilita componentes de software comunicarem-se diretamente sobre uma rede em uma maneira confiável, segura e eficiente. Anteriormente chamado "Network OLE", DCOM é designado para uso em múltiplos protocolos de transporte de rede, incluindo protocolos de internet tais como HTTP. DCOM é baseado na especificação DCE-RPC da Open Software Foundation.

DCOM: baseado no COM (Component Object Model) que é a base da tecnologia do Microsoft Transaction Server (MTS), que permite rodar um objeto COM em outras máquinas na rede, permitindo o uso de arquitetura "3-tier" (sistemas que possuem tres camadas) em sistemas operacionais da Microsoft (Windows 95 e Windows NT). Para entender melhor vamos fazer uma analogia com DLLs. Uma DLL é um conjunto de funções em um formato padrão do sistema operacional Windows. As DLLs são códigos compilados e "linkeditados", prontos para uso: o sistema operacional pode carregá-las e executá-las a qualquer momento.

O COM é uma versão OOP (Programação orientada a objeto) da DLL, um padrão a ser utilizado para criar objetos executáveis. Sobre este padrão foram construídos outros como OLE e ActiveX. O Padrão COM baseia-se nas DLLs, mas usa um modelo de objeto extensível e reusável. Os objetos COM estão ficando mais importantes com o passar do tempo. Têm as seguintes vantagens:

a) Permitem a criação de objetos executáveis, independentes da linguagem usada no desenvolvimento.

b) São usados pelos controles ActiveX .

Software AG e Digital Equipment Corporation estão portando o DCOM para outros sistemas operacionais, incluindo as múltiplas implementações do UNIX. A OMG (Object Management Group) está trabalhando na especificação para que aplicações DCOM possam comunicar diretamente com CORBA.

O protocolo DCOM é baseado no DCE RPC, assim, é fácil implementar o DCOM em plataformas para as quais o DCE RPC já está disponível. O DCE RPC define um padrão consolidado para converter estruturas de dados e parâmetros em memória para pacotes de rede. Sua representação de dados de rede (NDR - Network Data Representation) é neutra de plataforma e fornece um conjunto poderoso de tipos de dados portáveis.

Tanto o COM quanto o DCOM se utilizam da noção de Identificadores Globalmente Únicos (GUID - Globally Unique Identifier) do DCE RPC. Os GUIDs possibilitam nomeação de objetos e interfaces livres de colisão, o que é a base para os mecanismos de controle de versão do DCOM.

Em COM várias coisas recebem um identificador globalmente único *(globally unique identifie*r, ou GUID) [Rev99].

- GUIDS são também conhecidos como UUIDs *(universally unique identifier*s);
- Eles são equivalentes aos UUIDs do OSF DCE;
- Um *interface identifier* (IID) é o GUID (ou UUID) de uma interface COM;
- Novos GUIDS podem ser obtidos: Rodando um utilitário gerador de GUIDs, ou Contatando a Microsoft

Cada instância de objeto COM pertence a alguma classe, vejamos como ele se comporta [Rev99]:

- Cada classe é geralmente identificada por um GUID denominado identificador de classe *(class identifie*r, ou CLSID) ;
- Objetos da mesma classe normalmente suportam as mesmas interfaces (embora isso não seja obrigatório);
- Para saber exatamente que interfaces uma classe suporta, um cliente pode examinar as *component categories* dessa classe (Cada categoria é identificada por um GUID denominado *category identifier* (CATID));
- Os GUID possuem a representação de sua identificação no formato de 128-bits, como mostra o exemplo: *758851c0- 07d8- 11d1- 8fc1- 00aa00a88c96.*

Os GUID ou UUIDs como são chamados, possuem um valor de 128-bits que são garantidos virtualmente uma exclusiva identificação, para cada um no mesmo espaço e tempo. Para assegurar uma exclusividade para cada identificador, ele usa um algoritmo especial que usa a identificação da placa de rede da máquina, e o exato tempo em que cada um é criado. Cada GUID consiste em um grupo de 8 dígitos hexadecimal, seguido por três grupos de 4 dígitos hexadecimal cada, seguido por um grupo de 12 dígitos hexadecimal [Kol00].

A *Wind River Systems Inc*, é uma empresa que vem trabalhando atualmente no desenvolvimento do *VxDCOM*. É uma versão do *DCOM Real-Time*, voltado a sistemas de tempo real. O *VcDCOM* extende o *Wind River's VxWorks real-time* OS, de forma que OEMs pode conectar dispositivos embutidos, com aplicações baseadas em PC, nos ambientes distribuídos [Com99].

3.2 CORBA (Common Object Request Broker Architecture)

A tecnologia CORBA (Common Object Request Broker Architecture) e ORB(Object Request Broker) especificada pela OMG (Object Management Group), é um grupo formado por varias empresas. Possibilita a utilização de objetos distribuídos em aplicações modelo cliente/servidor. O padrão CORBA é independente de plataforma (sistemas operacionais, hardware e linguagens de programação), desta forma as diferentes camadas (cliente/servidor) de aplicações que utilizam a tecnologia de objetos distribuídos podem ser desenvolvidas considerando os aspectos que melhor atendem aos requisitos de um sistema em particular.

O padrão CORBA é uma arquitetura baseada em objetos distribuídos. Foi inicialmente implementado em 1991 como sendo um produto intelectual da OMG - Grupo de Gerência de Objetos, um consórcio de mais de 700 companhias das mais diferentes áreas (IBM, Canon, DEC, Philips, Sun, Apple, Informix e etc, assim como grandes usuários como Citicorp, British Telecom, American Airlines, e outros), interessadas em prover uma estrutura comum para o desenvolvimento independente de aplicações, usando técnicas de orientação a objeto em redes de computadores heterogêneas.

A interoperabilidade entre os objetos depende de um protocolo chamado o IIOP (Internet Inter-ORB Protocol) para objetos remotos. O IIOP especializa o protocolo TCP/IP (Transport Control Protocol / Internet Protocol), para comunicação em redes, de modo a otimiza-lo para transmissão dos tipos de dados utilizados pelo CORBA.

Tudo na arquitetura de CORBA depende de uma camada denominada ORB (Object Request Broker) [OMG96]. O ORB age como um Ônibus de Objeto central em cima do qual cada objeto de CORBA interage transparentemente com outros objetos de CORBA localizado localmente ou remotamente. Cada servidor de objeto CORBA tem uma interface e expõe um conjunto de métodos. Solicitar um serviço, a um cliente CORBA adquire uma referência do objeto a um servidor de objeto CORBA. O cliente pode fazer ligações de método agora na referência de objeto como se o servidor de objetos CORBA residisse no espaço de endereço do cliente.

O ORB é responsável por achar a implementação de um objeto de CORBA. Preparado para receber solicitações, passa adiante os pedidos a ele e carrega a resposta de volta para o cliente. É o componente mais importante da arquitetura proposta pela OMG. Permite que objetos façam e recebam requisições de métodos transparentemente em um ambiente distribuído e heterogêneo.

Um objeto CORBA interage com a ORB também pela interface da ORB ou por um adaptador de objeto, BOA (Basic Object Adapter) ou POA (Portable Object Adapter). CORBA pode ser usado em diferentes plataformas de sistema operacional, contanto que haja uma implementação de ORB para aquela plataforma. A interface do ORB é idêntica para todas as implementações CORBA, independentemente do adaptador de objetos utilizado. Através dela podem ser invocadas operações executadas pelo ORB úteis tanto para os clientes quanto para as implementações de objeto.

O modelo de objetos do CORBA é um modelo de objetos clássico, onde um cliente envia uma mensagem a um objeto, o objeto interpreta a mensagem e decide que serviço deve realizar. Como em um modelo clássico, uma mensagem identifica um objeto e zero ou mais parâmetros reais. O primeiro parâmetro é requerido e identifica a operação que deve ser realizada, e serve de base para que, durante a interpretação da mensagem, o objeto receptor (ou o ORB) possa selecionar o método adequado.

ORB é o *middleware* que estabelece os relacionamentos do servidor com o cliente entre objetos. Usando o ORB, um cliente pode transparentemente invocar um método

em um objeto do servidor, que pode ser na mesma máquina ou através de uma rede. Assim ORB fornece interoperabilidade entre aplicação de máquina heterogênea em ambiente distribuídos e diferentes SO. A definição do protocolo depende da linguagem de execução, do transporte da rede e de uma dúzia de outros fatores. ORBs simplificam este processo. Com ORB, o protocolo é definido através das relações da aplicação através de uma especificação de uma linguagem independente da única execução, a IDL (Language Definition Interface), que é usada para descrever interfaces dos objetos. Através da IDL são declarados os dados e métodos que podem ser acessados externamente contidos no objeto correspondente à interface que está sendo descrita. O CORBA é o coração das comunicações dos sistemas orientados a objetos, que traz a verdadeira interoperabilidade ao ambiente computação de hoje.

O padrão da OMG esta baseado no TCP/IP, a OMG trabalhou juntamente com o consórcio *World-Wide Web* (W3C). W3C usado pela OMG, que desenvolveu um padrão para unificar XML, chamado de XML *Metadata Interchange* (XMI). OMG está integrando XML agora em IDL, disponibilizando XML para qualquer aplicação que usa Corba [Lan00].

A OMG em agosto de 1999, anunciou seu último produto o CORBA *Component Model* (CCM) em San Jose, CA, USA. A *BEA Systems Inc*., foi a organização que teve a maior função para impulsionar a conclusão do CCM. O CCM era a peça final que precisava para finalizar o desenvolvimento do CORBA 3.0. O CCM engloba os padrões de programação na construção de sistemas. CORBA 3.0 também incluem especificações para a integração do CORBA com o Java e internet, Messaging, e Qualidade de Serviço (QoS) [Fra99].

CORBA mecaniza muitas redes e programação de tarefas como, por exemplo, registro de objetos, localização e ativação, sem com que os programadores venham a se preocupar com esses eventos.

3.3 JAVA/RMI

Remote Method Invocation (RMI) está baseado em uma tecnologia vista anteriormente denominada RPC. O RPC permite que um programa procedural (isto é, um programa escrito em C ou outra linguagem de programação procedural) chame uma função que reside em outro computador tão conveniente como se essa função fosse parte do mesmo programa que executa no mesmo computador.

RMI é uma das soluções para poder usar computação distribuída em Java. Permite que objetos Java sejam executados no mesmo computador ou em computadores separados se comunique entre si via chamadas de método remoto. Essas chamadas de método são muito semelhantes àquelas que operam em objetos no mesmo programa. É usado para desenvolver aplicações em redes que se comunicam entre máquinas virtuais. Java/RMI é simples e facilmente usado para fazer chamadas remotas. O RMI é mais um modelo cliente/servidor onde o cliente envia pedidos para o servidor e recebe de volta o resultado, isto se chama modelo de objeto distribuído. Ele possui três camadas básicas, o *client side stub* "proxy", *server side skeleton* e incluem uma camada mais alta, a camada de referência remota, e a camada de transporte que é a camada básica. Um *stub* implementa a interface de um objeto remoto. Este *stub* remete qualquer chamada então pelas camadas do servidor *skeleton*. O *skeleton* remete a chamada para o objeto remoto e retorna de volta o valor para o *stub* cliente. A camada de referência remota carrega um protocolo específico de referência remoto no qual é independente do *stub* do cliente e do *skeleton* do servidor. A camada de transporte é responsável por ativar a conexão e descrever encalço do objeto. Cada camada é implementada com sua própria interface e protocolo.

Java/RMI depende de um protocolo chamado JRMP (Java Remote Method Protocol). Java conta com Java object serialization que permite organizar objetos (ou transmitido) como um *stream*. Tudo deve ser definido em JAVA/RMI, o servidor e o cliente tem que ser escrito em Java. Para localizar um objeto no servidor pela primeira vez, RMI depende de um mecanismo do cliente que chama um RMIREGISTRY que vai à máquina servidora e guarda informação sobre objetos disponíveis no servidor. Um

cliente Java/RMI adquire uma referência do objeto a um servidor Java/RMI fazendo um *lookup* para uma referência dos objetos no servidor e invoca métodos no servidor, como se o servidor de objeto Java/RMI residisse no espaço de endereço do cliente. Java/RMI, pode ser usado em diversas plataformas de SO, contanto que haja uma JVM (Java Virtual Machine) para devida plataforma.

3.4 DCE (Distributed Computing Environment)

O DCE (Distributed Computing Environment) [Ros92] [OSF92] é um ambiente de computação distribuída definido pela OSF (Open Software Foundation) que é uma organização voltada para o desenvolvimento de software internacional e foi fundada em 1988, formada de um consórcio de fabricantes de computadores: IBM, HP, DEC, NIXFORD, APOLLO, PHILLIPS, SIEMENS, ou seja, as maiores empresas de todo o mundo. Com objetivo de propor padrões para sistemas abertos e portáveis. [Gra91] Constitui uma organização sem fins lucrativos, aberta e participante de várias categorias, incluindo desde fornecedores de hardware e software até instituições educacionais, agencias governamentais e outros tipos de organizações em todo o mundo. A OSF é constituída de uma empresa para desenvolvimento de software e de um instituto de pesquisas.

Não é nem um sistema operacional, nem uma aplicação, podem ser considerados um conjunto integrado de serviços e ferramentas que podem ser instalados como um ambiente em sistemas operacionais existentes e servir como plataforma para construção e execução de aplicações distribuídas. DCE tem como objetivo facilitar a integração em redes de computadores de diferentes tecnologias, e fornece ferramentas que facilita o trabalho cooperativo, flexibilidade de acesso em dados distribuído excelente balanceamento de carga no sistema e segurança. Comporta-se como uma camada de software que mascara as diferentes arquiteturas de máquinas, entre sistemas operacionais e redes de computadores distintos. O DCE pode ser visto como uma camada situada sobre o SO e protocolo de transporte, que fornece seus serviços para

aplicações situadas sob o nível acima da hierarquia de camadas. O DCE contém dois conjuntos de serviços:

- **Serviços Distribuídos Fundamentais**: fornecem ferramentas para desenvolvedores de software, de modo que eles possam criar os serviços exigidos pelos usuários em um ambiente de Computação Distribuída.
- **Serviços de Compartilhamento de Dados**: fornecem serviços de arquivos distribuídos, e suporte para arquivos MS-DOS e programas de controle de impressoras (*Spooler*).

O DCE foi projetado para poder ser portado facilmente para sistemas operacionais que forneçam serviços similares, incluindo OSF/1, Unix System V e sistemas operacionais não-Unix, como OS/2 ou VMS/DEC.

O DCE possui diversos componentes integrados que realizam funções importantes, tais como: RPC, GDS (serviços de diretório global) e DDS (cell directory service), serviços de segurança, *threads*, serviço de tempo distribuído (sincronização) e serviço de arquivos distribuídos.

O modelo de programação que está por baixo do DCE é o modelo cliente/servidor, no qual os usuários processam atos como clientes e podem ter acesso a serviços distantes oferecidos por um servidor.

O DCE é considerado de grande escalabilidade sendo que um sistema executando DCE pode ter milhares de computadores e milhões de usuários espalhados pelo mundo. Para lidar com isto, o DCE usa o conceito de células. Uma célula é um grupo de usuários, máquina ou outros recursos que tem um propósito comum e compartilha os mesmos serviços DCE. Uma configuração mínima de célula DCE necessita um serviço de diretório, servidor de segurança, servidor de tempo e máquinas clientes.

O OO-DCE (Object Oriented DCE) [Dil93] é uma extensão do DCE para o desenvolvimento de aplicações distribuídas orientadas a objetos. O OO-DCE é

composto por uma biblioteca de classes de objetos que auxilia o programador no desenvolvimento de aplicações sobre o DCE, aliado a linguagem de definições de interface a IDL (Interface Definition Language), para classes de objetos servidores e clientes que são codificados na linguagem C++.

4 MODELO DO SISTEMA REFLEXIVO AURORA

Neste capítulo analisaremos a apresentação do problema, a estrutura do sistema Aurora proposto por [Zan97] e adotado como ponto de partida para o projeto e a implementação de um modelo de gerenciamento de objetos distribuídos. É apresentado como o Aurora representa os objetos no ambiente, de forma que o *Access Virtual List Object* (AVLO) será responsável pelo gerenciamento dos mesmos. Aspectos sobre o desenvolvimento e implementação do modelo AVLO é discutido no capítulo 5. Neste capítulo, analisaremos um pequeno enfoque sobre RF (Reflexão Computacional). Descreveremos a funcionalidade e as características principais deste modelo.

4.1 Reflexão Computacional

Um programa reflexivo é um que pode argumentar sobre ele mesmo. Uma arquitetura procedural completamente refletiva é uma na qual um processo consegue ter acesso e pode manipular de forma completamente explícita, casualmente conectado representa sobre seu próprio estado, onde qualquer mudança feita na própria-representação de processos é refletido imediatamente em seu estado atual e comportamental.

Segundo [Mae87], sobre reflexão no modelo de orientação a objetos é que a RC representa atividade executada por um sistema quando faz computações sobre (e possivelmente afetando) suas próprias computações.

Um sistema reflexivo é aquele capaz de acessar sua própria descrição e alterá-la de modo a mudar seu próprio comportamento.

Arquiteturas reflexivas apresentam características importantes como a capacidade de atuar sobre a sua estrutura de código em tempo de execução, onde pode mudar o comportamento do código como desejado. O uso da reflexão pode ser separada da

programação funcional (como funções, procedimentos e classes do sistema) de forma transparente da programação não funcional.

A utilização de RC traz benefícios importantes que devem ser ressaltados para um melhor entendimento de sua capacidade computacional, tais como:

- O modelo reflexivo proporciona a construção de ambientes computacionais onde todos os níveis podem ser baseados na mesma abstração computacional.
- Permite fazer uma clara separação entre os mecanismos de suporte a objetos e a política que estabelece como estes mecanismos são utilizados.
- Inclui simplicidade ao modelo conceitual, provendo simplicidade em alterar o comportamento de todos os aspectos do sistema, de modo que se tudo no sistema é suportado através de reflexão, então todos os aspectos do sistema podem ser reedificados e manipulados.
- Reflexão prove uma maneira de suportar um ambiente aberto e adaptável. Como o ambiente do sistema operacional evolui, pode-se evoluir o ambiente e manipulá-lo de forma conveniente.

As linguagens orientadas a objetos são as principais ferramentas para se trabalhar com RC no desenvolvimento de aplicações, pois permitem o desenvolvimento incremental da aplicação. Onde é possível alterar o comportamento da classe em tempo de execução ou até mesmo interceptar uma ativação de um comportamento de um objeto ativo. A noção de reflexão em sistemas orientados a objetos é mais aprofundada nas linguagens de programação, por exemplo, CLOS e Smalltalk apresentam a habilidade de realizar processamento sobre si mesmas e em particular de estender, em tempo de execução, a própria linguagem. Estas propriedades são alcançadas através do uso de meta-classes. Cada objeto no sistema tem um meta-objeto que descreve a própria classe, através do qual o programador pode alterar o comportamento da classe.

O paradigma de reflexão foi introduzido em sistemas orientados-objetos por Maes [Mae 87] em uma linguagem chamada 3-KRS. Cada objeto é dividido em um domínio

específico e um meta-objeto. O meta-objeto contém informação que se descreve como uma entidade computacional por si mesmo. Manipulação sobre um objeto o meta-objeto pode afetar a computação em relação ao objeto, referente a sua realização da tarefa.

Um *Meta-Objeto* é um processador virtual no qual um objeto executa. Por exemplo, como um objeto pode comunicar-se com outro objeto é implementado no *Meta-Objeto*. Um aspecto sobre reflexão que tem que ainda ser testado completamente é como se comportará referente a sua performance. Reflexão pode ser usada para mudar tanto interfaces e implementação.

4.2 Gerenciamento de Objetos no Aurora

O modelo estrutural de gerenciamento de objetos no Aurora apresenta como peça chave o uso de reflexão computacional, que é a realização do relacionamento entre objetos e metaobjetos no nível do modelo conceitual. Analisaremos a forma com que Aurora realiza a instanciação e ativação de seus objetos.

4.2.1 Objetos, Meta-Objetos e Meta-Espaços

De forma simplificada, objetos no modelo podem ser descritos como entidades que representam estados, ou seja, são as entidades criadas para modelar as abstrações da informação, ou dados. *Meta-Objetos* por usa vez, podem ser descritos como as entidades que descrevem o comportamento dos objetos, ou seja, são as entidades criadas para modelar as abstrações funcionais.

Assim cada objeto do sistema, representa uma peça da funcionalidade do ambiente e possui um *meta-objeto* associado que descreve seu comportamento. Devemos observar que conceitualmente o relacionamento estado/comportamento é uma relação um-para-um. A generalização deste relacionamento para uma relação um-para-vários, permite que cada objeto tenha associado a ele um grupo de *meta-objetos*. Este

grupo de *meta-objetos* é chamado de *meta-espaço* e representa o conjunto de funcionalidades disponíveis aos objetos.

As abstrações descritas são apresentadas na figura abaixo. Nela pode ser visualizado um simples objeto, cujo comportamento esta definido pelo conjunto de *metaobjetos* que compõem o *meta-espaço*. Um *meta-espaço* pode ser visto como sistema operacional dedicado a execução do objeto, visto que todo o comportamento do objeto esta representado dentro dele pelo conjunto de *meta-objetos*.

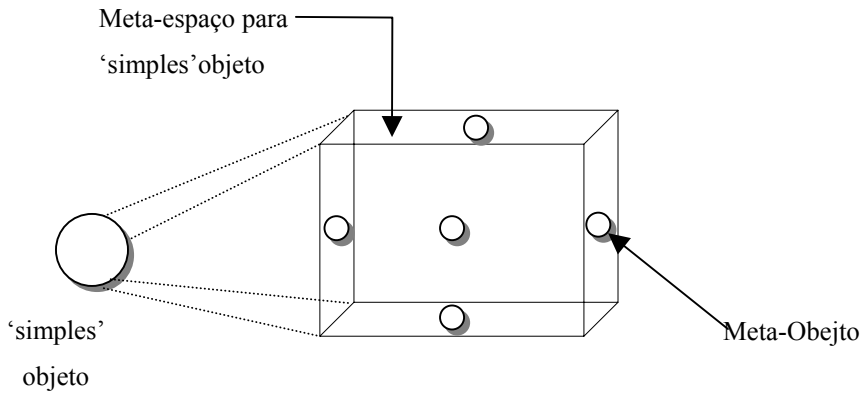


FIGURA 4.2 Visualização de um 'simples' objeto

4.2.2 Instanciação de Objetos

Teoricamente a instanciação de um objeto significa a criação de uma instância de uma classe privativa de objetos. Assim o processo de criação de um objeto implica em acessar características da classe a fim de determinar a estrutura semântica da mesma e concretizá-las na forma de objeto.

Uma definição de *classe* é um modelo para criar instâncias de objetos, com cada uma delas contendo as mesmas variáveis de estado e os mesmos métodos que todas as

outras instâncias de objeto de sua classe. Porém, os valores das variáveis de estado de objetos diferentes são independentes.

Para este processo ser levado ao barramento do meta-nível, necessita a transferência do controle da execução do nível de objeto para o nível do meta-objeto. Esta transferência é realizada através do controle das primitivas do *MetaCore.* A partir da efetivação do processo de instanciação do objeto, o mesmo passa a fazer parte de um meta-espaço do ambiente computacional, podendo ser ativado por outros objetos do sistema, ou seja, poder ser executado.

Deve-se observar que a instanciação de um objeto por definição do modelo de meta-objetos requer que a instanciação do meta-objeto que implementa seu comportamento seja realizada de forma simultânea. Este processo é garantido pelo meta-espaço que implementa suporte ao gerenciamento de objetos.

O aspecto mais importante que não foi relatado acima diz a respeito de onde vai ser criada a instância de um objeto, sendo que o modelo do sistema Aurora é distribuído podendo assim um objeto ser instanciado em qualquer nó que esteja ativado. Abordaremos estes aspectos no próximo capítulo, onde discutiremos quais políticas realizadas na administração dos objetos instanciados.

4.2.3 Ativação de objetos

A ativação de um objeto significa o envio de uma mensagem do objeto origem ao objeto ativado. Todo o processo realizado fora do objeto é transferido para o meta-nível, de modo que o meta-espaço deve prover a implementação de uma base para mensagens.

Quando uma mensagem é enviada, o controle da execução é transferido do objeto que executa o envio da mensagem para o meta-objeto que implementa o tratamento de mensagens. Este processo de transferência da execução do objeto para o meta-objeto é realizado através de primitivas de meta-computação no *MetaCore*.

Para poder analisar melhor a dinâmica de uma ativação do objeto, visualize a figura 4.2.3. Vemos que o meta-objeto (*MetaCore*) realiza meta-computação, para realizar o relacionamento *objeto*/*meta-objeto* e a transferência do controle de execução para o objeto ativado.

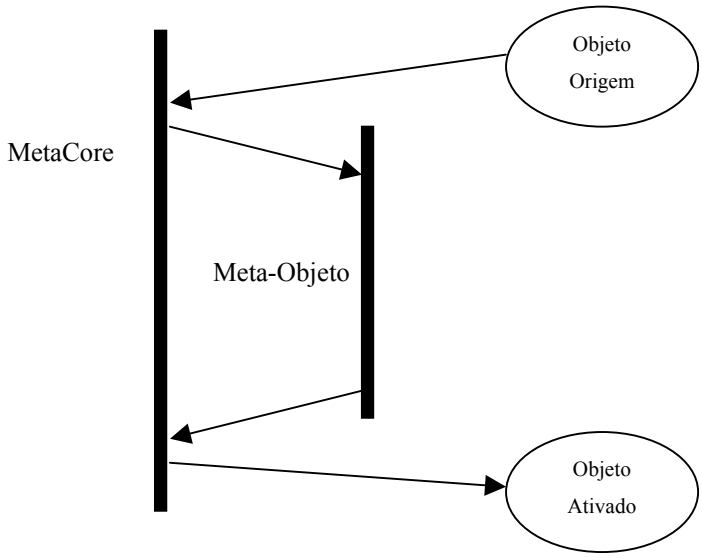


FIGURA 4.2.3 Dinâmica da ativação de objetos

Desta forma umas das possíveis funções visualizadas no meta-nível com conseqüência do envio de mensagens é administrar a localização do objeto ativado.

4.3 Visão Geral do Sistema Aurora

O Sistema Aurora utiliza o modelo de objetos como entidade fundamental. De modo que objetos são as únicas entidades presentes em todos os níveis do ambiente, modelando desde aplicações do usuário, serviços do sistema operacional e recursos do sistema.

A arquitetura resultante da utilização do modelo de estrutura reflexiva sobre Aurora, pode ser vista como definido dentro de uma hierarquia de *meta-espaços*, onde no topo da hierarquia está implementado o suporte ao modelo estrutural, a partir do qual o SO, utilitários e aplicações do usuário são construídos. A implementação deste suporte é realizada através de *meta-espaços* que implementam basicamente mecanismos para: gerenciamento de meta-espaços, suporte a ativações, suporte a migração, gerência de localização de objetos e a multiprocessamento.

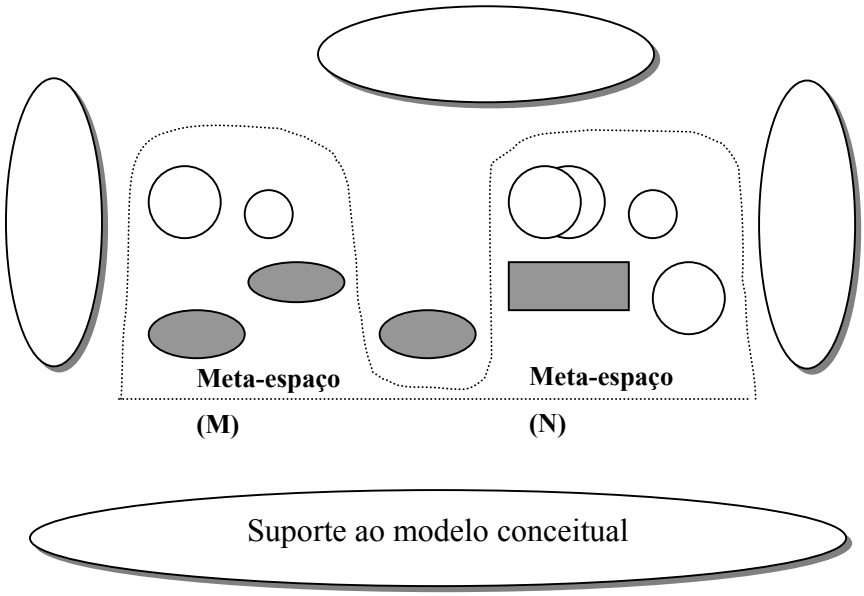


FIGURA 4.3 Visão Simplificada da Arquitetura Aurora

A figura 4.3 apresenta uma visão simplificada deste modelo onde o suporte ao modelo estrutural constitui a base da hierarquia. Deve-se observar que alguns meta-espaços não possuem objetos associados. Esta particularidade ocorre principalmente nos meta-espaços que implementam basicamente serviços do sistema operacional.

5 TRABALHO DESENVOLVIDO

Este capítulo apresenta o trabalho desenvolvido como dissertação de mestrado, foi abordada uma solução para identificação e localização de objetos distribuídos no sistema Aurora. Veremos também as técnicas que foram utilizadas ao longo do trabalho desenvolvido.

O trabalho desenvolvido disponibiliza um novo método de localização de objetos distribuídos, validaremos o modelo analisando aspectos de performance, flexibilidade, custo de processamento e requisitos que melhor satisfarão o sistema. Para o melhor entendimento o trabalho foi nomeado de *Access Virtual List Object* (AVLO).

É importante ressaltar que o sistema Aurora possui um balanceador de carga que é responsável pela distribuição dos objetos no ambiente, deste modo quando ocorre uma instanciação de um objeto o balanceador determinara o local físico (máquina/nó) onde o mesmo será mantido, ou seja, o nó onde o objeto ficará hospedado até o momento de sua exclusão (destruído/migrado). Através desta informação, o método AVLO administra sua localização no ambiente.

Desta forma, quando um objeto é ativado, o AVLO informa a localização do objeto no ambiente, informando ao sistema a localização do objeto, que assim envia uma mensagem para o *Metacore* onde ele encarrega-se de repassar uma mensagem para a máquina onde se encontra a instância do objeto e acoplado à mensagem encontra-se a ação que deve ser executada pelo objeto.

5.1 Desafios Encontrados

Deve-se levar em consideração algumas barreiras encontradas para que se torne transparente a localização e gerenciamento de objetos. Sistemas cuja proposta é suportar a abstração de objetos no nível de SO, conforme [CHI91], devem ter a habilidade de permitir que objetos sejam mantidos, gerenciados e usados eficientemente.

Uma importante observação é sobre as máquinas que estão distribuídas e ligadas entre si por qualquer meio físico, é sobre suas identificações, pois cada máquina recebe uma identificação para seu *MetaCoreID* e deve-se garantir que esta identificação seja única dentro do ambiente.

Entre as principais tarefas do gerenciamento de objetos pode-se destacar a identificação e localização. Estas tarefas englobam alguns aspectos que devemos ressaltar:

- Como gerenciar em um ambiente distribuído a localização dos objetos instanciados no sistema.
- Como identificar os objetos de modo a compartilhar ambientes distribuídos.
- Como localizar os objetos instanciados.
- Que atributos do objeto são relevantes em um sistema distribuído/paralelo.
- Que informações sobre o objeto são relevantes à sua localização.
- Fornecer meios que permitam a mobilidade dos objetos, por exemplo: em caso de migração.

5.2 AVLO (Access Virtual List Object)

O AVLO foi desenvolvido através de estudos das técnicas de gerenciamento de objetos e sobre características de distribuição dos SO's e aplicações distribuídas, tais como: CORBA, DCOM, DCE e outras. Analisamos principalmente o funcionamento do sistema Apertos, que possui características semelhantes ao Aurora.

O modelo consiste em gerenciar cada nó participante do sistema, guardando informações dos objetos instanciados por ele e dos objetos que são hospedados pelos outros nós em seu espaço físico (ambiente operacional). O nó não precisa se preocupar

com as informações dos objetos instanciados pelos outros nós. Mas ele possui habilidade de informar quais objetos estão instanciados nele fisicamente pelos outros nós, e sabe informar quem é o proprietário de cada objeto a fim de suprir uma migração futura do objeto remoto.

O AVLO consiste em duas listas de objetos, estas listas estão residentes na memória principal do sistema. O processo realizado no controle dos objetos, é independente, um para cada caso.

Estas listas consistem em um array de objetos: Network Address Object (NAO) e Object Remote Instantiated (ORI), ver cap. 5.2.2. Nota-se que a lista de objetos consiste numa lista dinâmica controlada pelo SO. O gerenciamento, no uso de memória usada pela lista, deve-se ressaltar que fica no nível do sistema operacional que prove propriedades para administração de memória.

Deve-se observar ainda que o problema de tolerância à falha não é de responsabilidade do AVLO, de modo que não existe a necessidade de se preocupar se uma máquina existe ainda no espaço do sistema, bem como dados que trafegam sobre a rede, tais tarefas não ficam sobre responsabilidade do AVLO mais sim do contexto de Aurora.

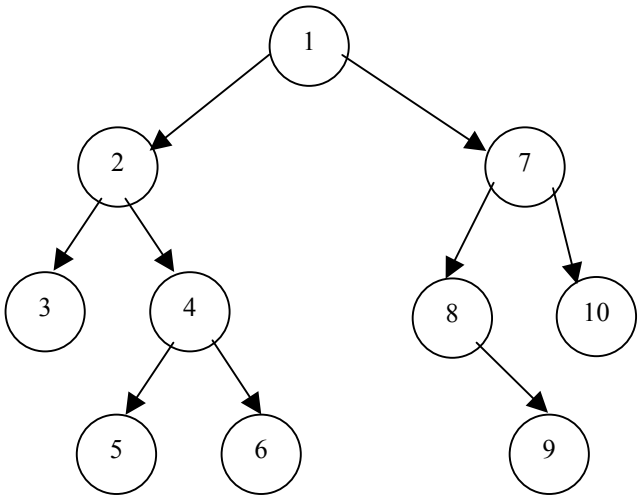
Deve-se ressaltar que o AVLO é responsável pelo gerenciamento de localização dos objetos instanciados pela máquina local e também dos objetos que são hospedados remotamente por outras máquinas.

5.2.1 Representação das listas no AVLO

As listas, onde se encontram as informações dos objetos instanciados, são armazenados em estruturas em forma de *Árvore binária*.

Esta Árvore se encontra ordenada onde cada vértice pode ter no máximo dois filhos, chamados filho *esquerdo* e *direito*; mais ainda, quando um vértice tem filho único, ele é ou filho *esquerdo* ou *direito*, exclusivamente.

Árvores binárias podem ser representadas por duas matrizes unidimensionais Esq e Dir. Se os vértices são representados por inteiros positivos, pode-se ter Esq(i) = j se e somente se j é o filho esquerdo de i. Se vértice i não tem o filho esquerdo, tem-se Esq(i)=0. Analogamente, define-se Dir. Por exemplo, na figura 5.1b, tem-se a representação da *árvore* na figura 5.1a.



(a)

	Esq	Dir
1	2	7
2	3	4
3	0	0
4	5	6
5	0	0
6	0	0
7	8	10
8	0	9
9	0	0
10	0	0

(b)

FIGURA 5.1 Representação da estrutura da Árvore

5.2.2 Identificação dos Objetos

Analisaremos agora como o AVLO faz para gerar uma identificação de um objeto que é instanciado no sistema. Esta propriedade requer uma atenção especial, pois é através desta identificação que o objeto é controlado no ambiente.

Deve-se ter extrema precaução com as identificações dos AID's (figura 5.2) dos objetos, pois ambas as listas poderão coincidir uma mesma identificação a objetos

distintos, já que uma identificação é gerada de forma seqüencial. Desta forma pode vir a acarretar sérios problemas na ativação de um objeto.

A classe AID será responsável por atribuir uma identificação aos objetos que são instanciados no Aurora, a cada instanciação de um novo objeto é feita uma chamada a classe AID que concederá uma identificação para o objeto a ser criado. Esta identificação concedida deve ser garantida que será a única em todo o ambiente computacional.

```
class AID {

  private:
      magicword       ObjectID ;
      magicword       KernelID ;
      magicword       identifyObject ;

  public :
      AID() ;
      ~AID() ;
      AID(magicword K_id, magicword o_id) ;
      magicword       GetID() ;

} ;
```

FIGURA 5.2 Interface Classe AID

Cada nó é responsável pela variável *ObjectID* receber uma identificação, (ver figura 5.2) o problema é que objetos remotos podem possuir o mesmo valor da variável *ObjectID*, já que a identificação será feita de forma seqüencial. A solução para este problema foi adotar uma dupla identificação para os objetos, sendo que a identificação do objeto será um número seqüencial gerada pelo AVLO mais a identificação do *MetaCoreID* que será atribuída ao atributo *KernelID*, realizando desta forma uma concatenação dos ID's. Através do método construtor da classe, é gerada a identificação do objeto, o atributo *identifyObject*, que por sua vez recebe esta identificação, que será atribuída ao objeto a ser criado no contexto de Aurora.

- **MetaCoreID**: É uma identificação única para cada *MetaCore* no contexto de Aurora.

A identificação do *MetaCoreID* é concedida ao sistema através de um algoritmo especial, que gera uma identificação a partir de recursos de hardware, este esquema é de responsabilidade do Aurora.

Para cada nova identificação solicitada ao AVLO, é criada uma nova instância da classe AID, que é endereçado para o objeto associado. E através da chamada do método *GetID()*, é retornado a identificação do objeto que está atribuída ao atributo *identifyObject*.

A interface da classe C++ modelada para geração de AID's é apresentada a seguir, uma descrição completa da modelagem e implementação da classe AID poder ser encontrada no anexo 1.

5.2.3 Network Address Object (NAO) e Object Remote Instantiated (ORI)

O controle da localização dos objetos é realizado através de dois processos distintos: *Network Address Object* (NAO) e *Object Remote Instantiated* (ORI). O primeiro método controlará todos os objetos instanciados localmente e armazenará as informações para gerenciar sua localização. O segundo método gerenciará a localização dos objetos instanciados remotamente, e que estão fisicamente hospedados nesta máquina. Desta forma as listas não necessitam conhecer informações umas da outra. Ambos métodos consistem na implementação de duas classes, denominadas Classe NAO e Classe ORI.

Ao ser criada uma nova instância de um objeto a classe NAO será ativada através da meta-classe ManagerObjects, ela invocará o comportamento da classe NAO denominado *NewObject()*, (ver figura 5.3) o método então invocará a classe AID cuja responsabilidade será gerar uma identificação ao objeto, e armazenar as informações

referentes à identificação do objeto solicitado na árvore correspondente. Se o objeto criado ficar hospedado em uma outra máquina diferente de onde o mesmo foi instanciado a *Meta-Classe,* ManagerObjects envia um mensagem ao *MetaCore* local que repassa a mesma para o *MetaCore* remoto, que transfere a informação para a meta classe ORI, que terá a responsabilidade de armazenar as informações do objeto que foi criado na árvore correspondente.

```
class NAO : public ArvoreB {

  private:
     Cont            *obj_Cont ;
     Balanceador     *obj_Bal ;
     AID             *Obj_aux ;
     T_nodo          *raizID ;
     magicword       id_máquina ;

  public:
     NAO() ;
     NAO(magicword no) ;
     ~NAO() ;
     void NewObject() ;
     magicword GetNoInstanted(plongword id) ;
     T_nodo* ListIDObject() ;
     void DeleteObject(plongword id) ;
     void UpdateObject(plongword id, longword no) ;
     longword FindObject(plongword id) ;
     AID *GetIDObject(magicword id) ;

} ;
```

FIGURA 5.3 Estrutura da Classe NAO

Caso necessite a execução de algum comportamento do objeto instanciado, esta requisição é transferida novamente para ManagerObjects que invocará o comportamento da classe NAO, responsável pela localização do objeto. Tal comportamento denominado de *FindObject()*, retornará o endereço do objeto para o *MetaCore*, que encaminhará a invocação do objeto à máquina onde sua instância estiver ativa.

Para entender melhor a dinâmica do processo de como o AVLO se comportará, vamos analisar um pequeno exemplo:

Suponhamos então que o ambiente possua três máquinas (A,B,C) distribuídas em uma rede sendo que cada máquina instanciará dois objetos. Consideraremos então que ambas as máquinas A,B,C instanciam os objetos XX e YY. A máquina "A" possui seu *MetaCoreID* com valor 4050, "B" 4150 e "C" 5080.

Então:

No momento da ativação de um objeto a *Meta-Classe* ManagerObjects, intercepta a ativação do objeto, fazendo assim uma chamada à meta-classe AID, responsável em atribuir uma identificação ao objeto.

1. A máquina "A" instancia o objeto XX (40501) na máquina "B", e o objeto YY (40502) nele mesmo. Note que a máquina "A" (tabela 5.1) a lista NAO contem informação de dois objetos instanciados, e máquina "B" (tabela 5.2) a lista ORI contem informações do objeto instanciado pela máquina A.

2. A máquina "B" instancia o objeto XX (41501) na máquina "A" e o objeto YY (41502) na "C". A máquina "B" (tabela 5.2) então possui dois objetos instanciados por ela, onde que as informações dos objetos então armazenados na lista NAO. A lista ORI apresenta informações de outros objetos pertencentes a outras máquinas "A" e "C".

3. A máquina "C" instancia o objeto XX (50801) na máquina "B" e o objeto YY (50802) nele mesmo. A máquina "C" (tabela 5.3) possui informação em sua lista NAO dos objetos instanciados por ela. A lista ORI possui um objeto que pertence à máquina B. Veja que a máquina "B" (tabela 5.2) em sua lista ORI apresenta dois objetos, um pertencente a máquina "A" e "B".

Máquina A			
NÃO		**ORI**	
IDObject	***NoDestino***	***IDObject***	***NoOrigem***
40501	B	41501	B
40502	A		

TABELA 5.1 Objetos instanciados pela máquina A

Máquina B			
NÃO		**ORI**	
IDObject	***NoDestino***	***IDObject***	***NoOrigem***
41501	A	40501	A
41502	C	50801	C

TABELA 5.2 Objetos instanciados pela máquina B

Máquina C			
NÃO		**ORI**	
IDObject	***NoDestino***	***IDObject***	***NoOrigem***
50801	B	41502	B
50802	C		

TABELA 5.3 Objetos instanciados pela máquina C

Com os exemplos acima mostrados fica claro o entendimento de como o AVLO faz o controle dos objetos instanciados por cada máquina (nó). Perceba também que através da dupla identificação atribuída aos objetos, torna-se praticamente impossível existir uma mesma identificação.

A classe ORI tem um papel importante no que diz respeito à migração dos objetos, pois quando uma máquina/nó participante do ambiente está prestes a se desconectar (desativada), o AVLO tem extrema facilidade de avisar o balanceador de carga, para que migre os objetos que estão hospedados em seu ambiente computacional. Desta forma todos os objetos que estão "pendurados" na máquina são migrados para outros nós que foram determinados. Os objetos pertencentes a máquina que resolveu se desconectar do ambiente são destruídos, onde eles estiverem.

Como vimos todos o objetos que são instanciados estão associados à classe NAO ou ORI, tais classes são usadas para gerenciar os objetos do sistema, a seguir é mostrada a modelagem da mesma, maiores detalhes de implementação podem ser obtidos no anexo 1.

```
class ORI : public ArvoreB {

  private:
     T_nodo         *raizID ;
     slongword       index ;

  public:
     ORI() ;
     ~ORI() ;
     void AppendObject(AID *id, longword no) ;
     magicword GetNoOrigem(plongword id) ;
     T_nodo* ListIDObject() ;
     void DeleteObject(plongword id) ;
     void UpdateObject(plongword id, longword no) ;
     longword FindObject(plongword id) ;
     AID *GetIDObject(magicword id) ;
} ;
```

FIGURA 5.4 Estrutura da Classe ORI

A *meta-classe* ManagerObjects (figura 5.5) está situada no *meta-espaço* do sistema. A meta-classe *ManagerObjects* será ativada toda vez que um objeto ativa um comportamento qualquer. Quando um nó passa uma mensagem para o *MetaCore* ele repassa a mensagem para o *meta-objeto* que chamamos de *ManageObjects*, ele verifica a localização do objeto e retorna a mensagem para o *metacore*, onde ele então passa a mensagem para o *metacore* remoto.

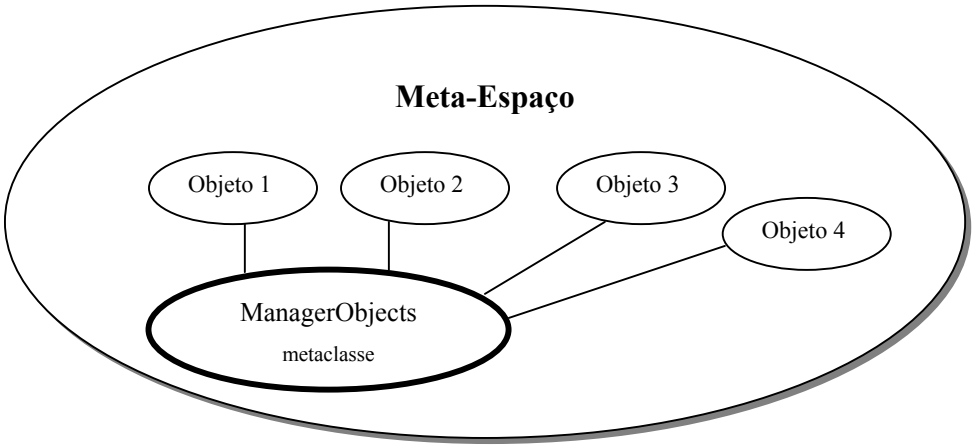


FIGURA 5.5 Visão simplificada do ambiente AVLO

5.2.4 AVLO e Suporte a MetaComputação (MetaCore)

A modelagem do ambiente em termos do relacionamento entre o meta-objeto terminal, MetaCore, e o modelo objeto/meta-objeto, está apresentada na figura 5.6, na qual podemos visualizar como a interação entre objetos é realizada através das primitivas de metacomputação implementadas. Toda vez que um objeto (a) deseja enviar uma mensagem para outro objeto (b), o objeto (a) executa uma chamada ao *MetaCore* através da primitiva M, provocando a transferência do controle para um meta-nível correspondente.

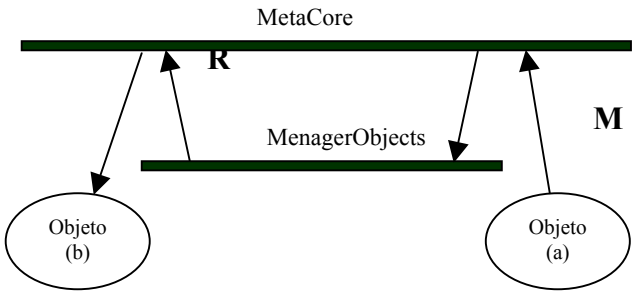


FIGURA 5.6 Interação dos Objetos via MetaCore

```
Class MetaCore {

public :
   void M (MetaActivity* mObject, MessageM *pMsg) ;
  void R (MessageR* pMsg) ;
} ;
```

FIGURA 5.7 Classe MetaCore

A primitiva M (figura 5.8) tem a função de transferir o controle de execução de um objeto para o meta-objeto. A definição dos parâmetros da primitiva são mostrados a seguir:

```
struct structMessageM {
        Activity        *source ;       // Activity do objeto fonte
        Activity        *target ;       //Activity do objeto destino
        void            *message ;      //mensagem a enviar
}:MessageM ;
```

FIGURA 5.8 Representação da primitiva M

```
struct structMetaActivity {
        Activity*       self ;    //Activity do objeto em execução
}:MetaActivity ;
```

FIGURA 5.9 Activity do objeto em execução

A primitiva R (figura 5.10) tem a função de realizar o processamento inverso ao da primitiva M, isto é, retornar o controle da execução do meta-nível para o nível do objeto.

```
struct structMessageR {
        Activity        *source ;       // Activity do objeto fonte
        void            *message ;      //mensagem a receber
}:MessageR ;
```

FIGURA 5.10 Representação da primitiva R

O benefício em se implementar suporte à identificação e localização de objetos em um meta-nível é torná-lo independente do contexto de execução dos objetos do nível base, isto garante uma total transparência de contexto e ambiente de execução.

5.2.5 Estrutura da Representação do Objeto

Características de como o objeto é identificado no ambiente foram descritas na seção 5.2.2, e o modo como eles são gerenciados foi abordado no capítulo 5.2.3. Veremos agora como é feita a abstração do objeto.

Objetos são sempre associados a uma *Activity*, que possui a abstração da CPU através de uma estrutura definida como *context* [Zan97].

Todo o Objeto instanciado é associado a uma *Activity*. Estes objetos incluem: descritores de *meta-espaços*, *meta-objetos* do sistema (aplicação), e objetos da aplicação. *Activity* é usada para representar os objetos instanciados no sistema, sendo responsável pelo contexto de execução do objeto. Analisaremos a seguinte representação básica da classe:

```
Class Activity {

protected :

        CPUContext      Context ;
        AID*            Identify ;
        Activity*       Meta ;
        EntryTable*     Execqueue ;
} ;
```

FIGURA 5.11 Estrutura da Classe Activity

Tais atribuições associadas à classe *Activity* contem o seguinte significado:

Context	Representa a abstração da CPU.
Identify	Identificação do Objeto associado a Activity.
Meta	Identificação do Meta-Espaço na Meta-Hierarquia.
Execqueue	Lista de ativações a serem executadas.

TABELA 5.4 Identificação dos atributos da Activity

Deve-se notar que o Aurora implementa no nível do sistema operacional uma clara distinção entre a instanciação de um objeto e a sua ativação, de modo que instanciações e ativações são separados em *meta-espaços* distintos. Quando é ativada uma instanciação de um objeto, o *MetaCore* ativa os mecanismos responsáveis pelo gerenciamento de *meta-espaços*, a fim de verificar no sistema a existência de um *meta-objeto* que represente o modelo instanciado.

6 AVALIAÇÃO DO AVLO

Na avaliação do modelo AVLO, é inerente a preocupação computacional com os processos desenvolvidos. Esta preocupação, por sua vez, implica no objetivo de se procurar elaborar algoritmos que sejam eficientes nas tarefas que são designadas. Conseqüentemente, torna-se imperioso a utilização de critérios que possam avaliar a eficiência dos mesmos. Como o modelo AVLO consiste em gerenciar os objetos do sistema, e sempre que um objeto é criado/destruído/localizado/migrado faz-se alguma operação sobre o AVLO (na árvore). Então podemos dizer que é visível à avaliação deste algoritmo, para podermos analisar sua eficiência.

Deve-se observar que medidas de eficiência/desempenho em geral são empíricas, principalmente no que se refere a eficiência de tempo, desta maneira vamos adotar ao AVLO. A medição constituirá exclusivamente em avaliar qual o grau de complexidade do algoritmo, que é uma medida aceita na comunidade da ciência da computação como uma medida válida, sem preocupar-se em analisar seu desempenho com medições de tempo. Já que o modelo é distribuído e podermos ter máquinas com poderes de processamento inferiores uma das outras, e também o ambiente de rede é outro aspecto que influencia na avaliação.

6.1 Características dos Algoritmos

Além de robustez, o algoritmo deve ser eficiente, ou seja, resolver o problema utilizando o mínimo de recursos computacionais, e em particular deve executar o mais rápido possível.

A avaliação do tempo de execução pode ser realizada de duas formas diferentes:

- Empíricos: Isto é, medir o tempo de execução propriamente dita considerando-se entradas diversas. *Exemplo*: Comparando o tempo de execução em computadores diferentes ou em sistemas operacionais diferentes, linguagens diferentes ou mesmo compiladores diferentes.

- Analíticos: Isto é, obter uma ordem de grandeza do tempo que objetive calcular uma expressão matemática que traduza o comportamento de tempo de um algoritmo. *Exemplo*: Calcular quantos passos de um determinado processo ou partes de um algoritmo são executados em função de uma única entrada.

Complexidade de espaço e tempo serão utilizados para avaliar o desempenho no AVLO.

Nesta abordagem o tempo de execução de um algoritmo será definido em função dos dados de entrada. Freqüentemente o tempo de execução não dependerá da natureza dos dados de entrada, mas do que se convenciona chamar de tamanho da instância. O tamanho ou comprimento de uma instância é definido como o número de *bits* necessários para codificá-la.

A função que associa o tempo de execução de um algoritmo ao comprimento ou tamanho dos dados de entrada denomina-se *complexidade em tempo do algoritmo*. Em outros casos, quando o interesse é examinar a quantidade de memória necessária para acomodar os dados de entrada e executar o processo, define-se a complexidade em espaço do algoritmo.

6.2 Análise do AVLO

A seguir, faremos uma análise matemática do algoritmo AVLO, analisaremos o grau de complexidade na inclusão de um novo valor na árvore que chamaremos de *T,* o procedimento *Insert* é analisado.

```
1   If (raiz == NULL)
2      "Cria a raiz principal da árvore"
3   Else  {
4      "Pesquisa na Arvore"
5      If "não encontrar"
           "Verifica Se x pertencerá a subárvore esquerda/direita"
6          "Insere na subárvore"
7      End
8      Else "Identificação Encontrada" end
   End
```

FIGURA 6.1 Procedimento de Inserção

Para executarmos *insert*(x, T), onde T é representado por uma árvore de busca binária com raiz r, na linha 1 (figura 6.1) verifica se a raiz==null, caso for ele insere x na árvore, onde ele será a raiz principal. Se não pesquisa x, (linha 4, figura 6.1),deve-se procurar x na sub-árvore esquerda de r (se esta existir) ou conseqüentemente procurar x na sub-árvore direita de r. Se x é representado por um vértice na árvore, então ele será corretamente localizado. Se não em um determinado momento deste processo, será definido que x pertencerá a uma sub-árvore, esquerda ou direita, concluímos então que x não pertence a T. A figura 6.1 apresenta matematicamente o algoritmo que acabamos de descrever.

Quando executamos *insert*(x, T), se a árvore está vazia, cria-se um vértice-raiz com conteúdo x. Se a árvore não esta vazia, e o elemento a ser inserido não é encontrado, cria-se um novo vértice, com conteúdo x, e ele deverá ser o novo filho esquerdo ou direito, respectivamente , de T.

- *Entrada*: (x, T), onde x é um elemento do conjunto universo, e T é a raiz de uma árvore, representando um conjunto; f_esquerdo e f_direito denotam respectivamente, os filhos esquerdo e direito de T.
- *Saída*: representa "sim" ou "não", resposta da operação. Onde "sim" inclusão realizada e "não" elemento (identificação) existente.

O algoritmo de inserção e remoção da árvore é executado em um tempo $O(h)$ [Cor90], onde h é a altura da árvore. Já o tempo médio em uma pesquisa binária com n *nodes* é $O(\log n)$.

Vimos então que inserção ou remoção de vértices numa árvore binária com h vértices só pode ser efetuada com rapidez pessimista a $O(h)$, devido à possibilidade de existirem (no caso pessimista) árvores em que todos os vértices só possuem filho esquerdo (ou só filho direito) [Ter91]. Em outras palavras, estas árvores desbalanceadas, possuem a maioria dos vértices relativamente distantes da raiz, exigindo que muitos vértices sejam visitados antes de se chegar a um vértice próximo à única folha. Desta forma devemos evitar a formação de árvores desbalanceadas, permitindo assim que as operações básicas sejam efetuadas em tempo *O(log*h)*. Devido a que árvores binárias são semicompletas e, portanto, não há possibilidade de formação de árvores desbalanceadas: os vértices permanecem tão próximos à raiz quanto possível; e, por isso, as operações básicas são efetuadas em tempo *O(log h)*.

Se tomarmos como ponto de vista sobre CORBA, o modelo AVLO pode vir a ser mais eficiente na localização dos objetos, pois as referencias dos objetos estão sempre mantidas na memória da própria máquina. Já no caso do CORBA, usando serviço de nomes, sempre o cliente tem uma referencia ao servidor de nomes. Quando ele necessita na execução de um determinado comportamento do objeto, ele faz uma chamada ao servidor de nomes requisitando onde se encontra a implementação do objeto com identificação x.

7 CONSIDERAÇÕES FINAIS

Este trabalho apresentou um novo modelo para gerenciamento de localização e identificação de objetos para o ambiente Aurora. Foram analisados os modelos de gerenciamento de objetos existentes, as técnicas e políticas utilizadas de forma a propor o método AVLO.

Dentre os processos que envolvem a execução do Aurora, o gerenciamento dos objetos é um dos requisitos importantes para atender sua funcionalidade e flexibilidade. Um protótipo da implementação do AVLO está em pleno funcionamento, e deve-se ressalvar que atingiu as metas propostas no trabalho.

O modelo proposto torna a exploração de ambientes distribuídos mais dinâmico. Pois não fica a cargo somente de uma máquina identificar a localização dos objetos mas sim de todas que participam do ambiente.

O AVLO foi desenvolvido no nível do sistema operacional, desta forma, arquiteturas de desenvolvimento de aplicações distribuídas como CORBA, DCOM ficam sem grandes vantagens sobre o Aurora. Pois as aplicações serão automaticamente distribuídas pelo Aurora/AVLO, e programadores não necessitam incluir uma linha de código, para que seu sistema se torne distribuído, tudo será feito pelo AVLO/Aurora de forma totalmente transparente ao programador e aos usuários do sistema.

Outra vantagem que o AVLO proporciona é sobre a localização dos objetos, pois para saber onde se encontra a localização de um objeto não há necessidade de localizar nenhum servidor de objetos, porque o controle é realizado pela própria máquina. Na migração de um futuro objeto, a máquina hospedeira não precisa sair “perguntando” pela rede quem é o proprietário do objeto “x”, ela possui também controle sob todos os objetos que estão hospedados nela fisicamente, e basta apenas ela ver quem é o proprietário do objeto e enviar uma mensagem a maquina “x” informando que o objeto “y” deve ser migrado.

7.1 Trabalhos Futuros

Fazer avaliações sobre o ambiente real de execução buscando obter melhores resultados, como por exemplo, a implementação de outras estruturas de dados.

Analisar o custo da identificação e localização do objeto no contexto de execução, de modo a determinar sua implicação percentual sobre o custo da execução.

Medir o modelo de identificação dual proposto, sua real necessidade e eventuais fatores na gerência dos objetos. Implementando um mecanismo totalmente independente na geração das identificações dos objetos, fazendo com que o modelo possa ser usado por outros sistemas.

8 BIBLIOGRÁFIAS

[AMO88] Amorim, Cláudio L., Barbosa, Valmir C., Fernandes, Edil S. T. *Uma Introdução à Computação Paralela e Distribuída*, Campinas, SP, UNICAMP, IMECC, 1988.

[BLA86] Black A., et al. Mach and Matchmaker – *Kernel and Language Support for Object-Oriented Distributed Systems*. OOPSLA'86. Vol 21/11. Nov1986.

[BUT94] Butenuth, R., COSY-KERNEL as na Example for Efficiente Call Mechanis on Transputers. Proceedisngs of the 6th Transputer/Occam International Conference, Tokyo, June 1994.

[CHA00] Chase, Jeff, *Distributed Objects: A Lightning Tour*, system & architecture, CPS 212, Fall 200, 2000.

[CHI91] Chin, R.S. Chanson. S.T.. *Distributed. Object-Based Programming Systems*. ACM Computing Surveys. Vol23. Nº1. Mar 1991.

[COM99] Computergram International. *Wind River Re-Implements Msoft's DCOM for Embedded Systems***.** Sept 28, 1999.

[COR90] Cormen, Thomas H. Leiserson, Charles E. Rivest, Ronald L, *Introduction to Algorithms*, New York: McGraw_Hill Book Company, 1990.

[DEI01] Deitel, H. M., Deitel, P. J., *Java*, *como Programar*, trad. Edson Furnankiewicz, 3º.ed., Porto Alegre: Bookman, 2001.

[DIL93] Dilley, J. *Object Oriented Distributed Computing with c++ and OSF DCE*, OSF DEC SIG, Request For Comments (RFC) 49.0, October 1993.

[FRA99] FRAMINGHAM, Mass. *CORBA Component Model Will Help Developers Quickly Design and Implement Mission Critical Distributed Systems*. BUSINESS WIRE. Sept. 1999.

[FUR97] Olinto J. V. Furtado. RTR – *Uma Abordagem Reflexiva para Programação de Aplicações Tempo Real*. Dissertação de Doutorado, Universidade Federal de Santa Catarina, Departamento de Engenharia de Sistemas, Faculdade de Engenharia Elétrica e de Computação, UFSC, Florianópolis, SC, 1997.

[GRA91] Gray,Pamela A. *Open Systems*: A Business Strategy for the 1990s. McGrawHill, Berkshirem England, 1991.

[ISL96] Islam, Nayeem. *Distributed Objects: Methodolgies for Customizing Systems Sofware*, IEEE Computer Society Press, 1996.

[ISO92] ISO, *International Standards Organization's,* Reference Model for Open Distributed Processing, 1992.

[JOU01] Journal Dr. Dobb's , http://www.ddj.com/print/documentID=13886, Copyright 2001.

[HEN98] Hennning, Michu *Binding Migration, and Scalabilyty in CORBA*, Communications of the ACM, October, 19998

[KOL00] Kolb, Ronny, *An introduction to COM, DCOM and COM+*, Seminar Component-based Software, Gera-Kleinaga, Germany, 2000.

[LAN00] Langley, Nick., *Cross-platform integration (Technology*

Information), Computer Weekly, November, 2000.

[MAE87] Maes, Pattie. Concepts and Experiments in Computational Reflection. In *Proccedings of OOPSLA '87*, pages 147-155, Orlando, Fla., October 1987.

[NEL82] B. J. Nelson, *Remote Procedure Call.* Technical Report CMU-81-119, Carnegie-Mellon University, 1982.

[OMG95] OMG – Object Management Group. CORBA, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, July 1995, http://www.omg.org/corba/corbiiop.htm.

[OMG96] OMG – Object Management Group. *Common Object Request Broker Archictecture*, Revision 2.0, OMG Document Number PTC/96-03-04, March 1996..

[OSF92] OSF – Open Software Foundation *Distributed Computing Environment Overview*, OSF White Paper, OSF-DCE-PD-1090-4, USA, 1992.

[REV99] Reverbel, Francisco. *Um Objeto COM***,** IME-USP, Departamento de Ciência da Computação, 1999.

[ROS92] Rosenberry, Ward; Kenney, David; Fisher, Gerry *Understanding DCE,* O' Reilly Associates Inc., USA, 1992.

[SCH94] Schröder-Preikschat,W., *The Logical Design of Parallel Operating Sustems.* PTR Prentice Hall, New Jersey, 1994.

[SMI82] Smith, Brian C. *Reflection and Semantics in a Procedural Language,* Ph.D. thesis, Massachusetts Institute of Technology, Cambridge,

Mass., January 1982.

[TAN92] Tanenbaum, Andrew S., *Modern Operating Systems.* Prentice-Hall, Inc, 1992.

[TEI96] Teixeira Junior, José Helvécio, Sauvé, Jácques Phillipe, Moura, José Antão Beltrão. *Do mainframe para computacao distribuída: simplificando a transição.* Rio de Janeiro: Infobook, 1996.

[TER91] Terada, Routo. *Desenvolvimento de algoritmos e estruturas de dados*, São Paulo:McGraw-Hill, Makron, 1991.

[TEX97] Texeira, Daniel L., Queiroz, Isaac E. . *Estrutura do Modelo COM e Tecnologia de Controles ActiveX*, 1997.

[TVR90] Tanenbaum, Andrew S. Renesse, Robert Van et al. AMOEBA. *Communications of the ACM,* 33(12):46-63, December 1990.

[VIN97] Vinoski, Steve. CORBA: *Integrating Diverse Applications Within Distributed.* Heterogeneous Environments, *IEEE Communications Magazine*, February, 1997.

[WANG97] Wang, Y. M. *Introduction to COM/DCOM*, http://akpublic.research.att.com/~ymwang/slides/DCOMHTML/ppframe.htm, 1997.

[YOK92] Yokote, Yasuhiko. *The Apertos Reflective Operating System*: *The Concept and Its Implementation*. In Proceedings of the 1992 Conference on Object Oriented Programming System, Languages and Applications, pages 414-434, October 1992.

[YOK92a] Yokote, Yasuhiko, Fujinami, Nobuhisa, *Naming and Addressing of Objects without Unique Identifiers*, Sony Computer Science Laboratory Inc., Tokyo, Japan, 1992.

[ZAN97] Zancanella, Luiz C. *Estrutura Reflexiva para Sistemas Operacionais Multiprocessados.* Dissertação de Doutorado, Universidade Federal de Rio Grande do Sul, , Instituto de Informática, UFRGS, Porto Alegre, RS, 1997.

AVLO – ACCESS LIST VIRTUAL OBJECT

ANEXO 1

Contém os resultados da validação do modelo implementado em C++.