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

Anderson Luiz Fernandes Perez

Um Mecanismo para a Comunicação Remota de Objetos no Sistema Operacional Aurora

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

Prof. Luiz Carlos Zancanella, Dr.
Orientador

Florianópolis, Abril de 2003

Um Mecanismo para a Comunicação Remota de Objetos no Sistema Operacional Aurora

Anderson Luiz Fernandes Perez

Esta Dissertação foi julgada adequada 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 do Curso

Banca Examinadora

Prof. Luiz Carlos Zancanella, Dr.
Orientador

Prof. Bernardo Gonçalves Riso, Dr.

Prof. Carlos Barros Montez, Dr.

Prof. Frank Augusto Siqueira, Dr.

”O homem se torna muitas vezes o que ele próprio acredita que é. Se eu insisto em repetir para mim mesmo que não sou capaz de realizar alguma coisa, é possível que realmente me torne incapaz de fazê-la. Ao contrário, se tenho convicção de que posso fazê-la, certamente adquirirei a capacidade de realizá-la mesmo que não a tenha no começo.”
Mahatma Gandhi

À memória do meu pai: Odair Perez Oubinha

Agradecimentos

A Universidade Federal de Santa Catarina, em especial ao Departamento de Informática e Estatística (INE).

Ao meu orientador, Luiz Carlos Zancanella, pelo incentivo, amizade, compreensão e pelos seus valiosos ensinamentos que foram muito importantes para a conclusão deste trabalho.

Aos professores e funcionários do Curso de Pós-Graduação em Ciência da Computação.

A CAPES (Coordenadoria de Aperfeiçoamento de Pessoal de Ensino Superior) pela bolsa de estudos.

A todos os meus amigos que se fizeram presentes nesta etapa da minha vida.

A minha família, principalmente aos meus pais, Odair Perez Oubinha e Mercedes Fernandes Perez, pelo apoio, incentivo e formação moral.

À Eliane Pozzebon, que é uma pessoa muito especial para mim, pelo seu carinho, amor e amizade.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

ACK - Acknowledgment

AVLO - Access Virtual List Object

CLSID - Class Identifier

CMIP - Common Management Information Protocol

COM - Component Object Model

CORBA - Common Object Request Broker Architecture

COSS - Common Object Services Specifications

DCE - Distributed Computing Environment

DCOM - Distributed Component Object Model

DII - Dynamic Invocation Interface

DLL - Dynamic Link Library

FIFO - First in First Out

FLIP - Fast Local Internet Protocol

IDL - Interface Definition Language

IIOP - Internet Inter-ORB Protocol

IP - Internet Protocol

JVM - Java Virtual Machine

LAN - Local Area Network

MC - Multicomputer

MIDL - Microsoft Interface Definition Language

MOP - Meta-Object Protocol

NTP - Network Time Protocol

ODL - Object Definition Language

OMA - Object Management Architecture

OMG - Object Management Group

OO - Object Oriented

ORB - Object Request Broker

ORPC - Object Remote Procedure Call

OSF - Open Software Foundation

RAM - Random Access Memory

RMI - Remote Method Invocation

RMIC - Remote Method Invocation Compiler

RPC - Remote Procedure Call

R/R - Request-Reply

SCM - Service Control Manager

SNMP - Simple Network Management Protocol

TCP - Transmission Control Protocol

UDP - User Datagram Protocol

UTC - Universal Time Coordinated

WAN - Wide Area Network

Resumo

O número de computadores interligados em rede está aumentando consideravelmente. Os ambientes distribuídos formados por essas redes possuem um grande potencial de processamento. Todavia, para que se possa aproveitar tal capacidade de processamento, faz-se necessário a utilização de um sistema que gerencie todo esse ambiente distribuído. Os sistemas operacionais distribuídos vêm ao encontro dessa necessidade. Os sistemas operacionais distribuídos caracterizam-se por permitir que seus usuários utilizem os recursos espalhados pelo ambiente distribuído de maneira transparente, escondendo os detalhes da implementação. O mecanismo de comunicação nesses sistemas é muito importante pois garante que componentes do sistema possam interagir a fim de executar uma determinada tarefa. Este trabalho descreve a solução para comunicação remota entre objetos no sistema operacional Aurora. O Aurora é um sistema operacional reflexivo projetado para arquiteturas multiprocessadas. A solução desenvolvida caracteriza-se por estender o mecanismo de comunicação local para permitir que o mesmo também suporte a comunicação remota de objetos. A comunicação entre objetos distribuídos em Aurora atende o requisito de transparência em sistemas distribuídos e é compatível com o modelo reflexivo do sistema.

Palavras-chave: Comunicação; Reflexão Computacional; Sistemas Operacionais Distribuídos; Objetos Distribuídos.

Abstract

The number of computers linked in networks has considerably increased. The distributed environments formed by these nets have a great processing potential. However, to make the best of this processing capacity, a management system must be used to manage these distributed environments. The distributed operating systems meets this requirement. The distributed operating systems allow the users to utilize the resources spread on the distributed environment in a transparent way, hiding implementation details. The communication mechanism in these systems is very important because it guarantees that the system components can interact in order to perform a certain task. This work describes the solution for remote communication between objects int the Aurora operating system. Aurora is a reflective operating system designed for multi-processed architectures. The developed solution extends the local communication mechanism in order to allow it to support the remote communication of objects. The communication between objects distributed in Aurora meets the transparency requirement in distributed systems and it is compatible with the reflective system model.

Keywords: Communication, Computational Reflection, Distributed Operating Systems, Distributed Objects.

Capítulo 1

Introdução

Os ambientes distribuídos vêm aumentando nos últimos anos. Prova disso é a Internet, uma rede pública criada pelo Departamento de Defesa dos Estados Unidos (DARPA), que se popularizou e hoje conecta milhares de computadores espalhados por todo o mundo.

A grande vantagem e a principal motivação para o uso da computação distribuída é o compartilhamento de recursos. Entretanto, o desenvolvimento de software para tais ambientes não é uma tarefa fácil, devido à grande diversidade de seus componentes de software e hardware.

Os problemas oriundos da heterogeneidade em um ambiente distribuído como a Internet podem ser resolvidos através da utilização de um software intermediário, ou seja, um software que atue entre o sistema operacional e a aplicação distribuída. Esse software intermediário, conhecido como *middleware*, é responsável por esconder as diferenças entre as máquinas que fazem parte do ambiente distribuído.

A utilização de um *middleware* no desenvolvimento de aplicações distribuídas esconde a heterogeneidade em ambientes distribuídos. Entretanto, acrescenta mais uma camada ao ambiente, afetando o desempenho das aplicações que são executadas sobre ele.

Os sistemas operacionais atuais devem se adequar às novas tecnologias, tanto em aspectos visuais quanto em aspectos funcionais. Devem ser capazes de se adaptar

em ambientes heterogêneos como os ambientes distribuídos, fornecendo e gerenciando recursos compartilhados entre seus usuários.

Segundo Tanenbaum [49], um sistema operacional moderno deve prover dois serviços fundamentais para o usuário. Primeiro ele deve permitir utilizar o hardware de um computador mais facilmente, criando uma máquina virtual que difere da máquina real, facilitando o uso da mesma por usuários finais. Segundo, um sistema operacional deve permitir o compartilhamento de recursos de hardware entre os usuários desses recursos.

Atualmente existe um grande avanço na pesquisa em sistemas operacionais distribuídos. Novas idéias e novos conceitos são aplicados a cada novo projeto fazendo com que surjam novas técnicas para o desenvolvimento de sistemas operacionais. Pode-se destacar o Apertos [55] e o Aurora [58] que utilizam reflexão computacional [59] para melhor se adaptarem a ambientes heterogêneos, e o 2K [24] e o Solaris MC [23] que aplicam as especificações CORBA em componentes do sistema, inclusive no próprio *kernel*.

Indiferentemente das técnicas empregadas no projeto de um sistema operacional distribuído todos esses sistemas visam solucionar os problemas referentes à heterogeneidade, o compartilhamento de recursos e a mobilidade em ambientes distribuídos. Para que o sistema funcione de uma forma transparente, atendendo os requisitos dos usuários, é necessário, entre outras coisas, que ele tenha um bom sistema de comunicação, responsável pela interligação de seus componentes, sejam eles locais ou remotos.

O sistema de comunicação é uma das peças fundamentais em um sistema distribuído. Ele é o responsável por interligar todos os componentes do sistema de uma forma transparente para o usuário, sem que este perceba que está acessando recursos de outra máquina. Para Goscinski [17] um bom mecanismo de comunicação é necessário para facilitar o acesso a recursos compartilhados no ambiente distribuído de maneira uniforme, independentemente de linguagem de programação e de localização.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo deste trabalho é propor e implementar uma solução para a comunicação entre objetos distribuídos no sistema operacional Aurora. Para alcançar o objetivo desse trabalho, foi feito um levantamento bibliográfico baseado em livros, artigos e publicações especializadas sobre os temas sistemas distribuídos, sistemas operacionais distribuídos e comunicação em sistemas distribuídos.

1.1.2 Objetivos Específicos

- Levantar bibliografia sobre sistema distribuídos e sistema operacionais distribuídos;
- Levantar bibliografia sobre os principais mecanismos de comunicação em sistema distribuídos;
- Fazer estudo de casos sobre sistema operacionais distribuídos e seus respectivos mecanismos de comunicação;
- Propor e implementar uma solução para a comunicação entre objetos distribuídos no sistema operacional Aurora;
- Desenvolver um simulador para validar a solução proposta;
- Fazer sugestões de trabalhos futuros que possam melhorar a solução desenvolvida.

1.2 Motivação

O grande salto na pesquisa em sistemas distribuídos deu-se após o surgimento da Internet. Os pesquisadores da área não viram a Internet somente como um mecanismo de pesquisa e troca de informações entre pessoas; mais do que isso, eles

viram a Internet como uma rede de grande potencial de processamento, devido ao crescimento do número de máquinas conectadas à rede e da capacidade de processamento destas máquinas.

Alguns projetos na área de sistemas distribuídos estão explorando a grande capacidade de processamento da Internet. Pode-se destacar o projeto SETI@home (*Search Extraterrestrial Intelligence*) [32] da NASA que, através da Internet, distribui fragmentos de imagens captadas por telescópios e radares para serem processados/analisados e, em seguida, os resultados dessa análise são submetidos ao computador central da NASA, para busca de vida inteligente fora do planeta Terra.

Projetos como o da NASA poderiam ser expandidos para outros campos de pesquisa como a medicina, por exemplo. Seria possível utilizar o potencial de processamento das máquinas conectadas à Internet para descobrir a cura de doenças, seqüenciar a cadeia completa do DNA humano ou, até mesmo, para se formar uma grande base de dados distribuída para troca de informações médicas.

A pesquisa em sistemas distribuídos é uma área promissora e desafiadora. Promissora pela sua grande aplicação em vários nichos de mercado. Desafiadora por se tratar de uma área em que o desenvolvimento de software para sistemas distribuídos possui um nível de complexidade alto, sendo necessário criar métodos e ferramentas para tentar minimizar essa complexidade.

O projeto Aurora apresenta uma nova arquitetura para o desenvolvimento de sistemas operacionais distribuídos. A arquitetura reflexiva permite que o sistema se adapte mais facilmente a ambientes heterogêneos como os ambientes distribuídos.

Este trabalho visa dar uma contribuição para o projeto do sistema operacional Aurora, propondo e desenvolvendo um modelo de comunicação entre objetos distribuídos totalmente compatível com a sua arquitetura reflexiva.

1.3 Organização do Texto

Esta dissertação está organizada como segue:

O capítulo 2 aborda aspectos relevantes da computação distribuída, suas

vantagens e desvantagens e o modelo cliente-servidor. O capítulo também descreve os principais mecanismos de comunicação utilizados em sistemas distribuídos, tais como: troca de mensagens, RPC (*Remote Procedure Call*) e comunicação em grupo

No capítulo 3 descrevem-se algumas das principais plataformas para o desenvolvimento de aplicações distribuídas: DCE, CORBA, DCOM e Java/RMI. O capítulo também apresenta estudos de casos de sistema operacionais distribuídos, dando ênfase ao modelo de comunicação adotado em cada um deles.

O capítulo 4 descreve os principais conceitos envolvidos no projeto do sistema operacional Aurora. Para um maior entendimento da arquitetura do Aurora, o capítulo traz uma breve introdução a respeito de reflexão computacional que é o conceito chave no projeto do Aurora.

O capítulo 5 descreve a solução para comunicação remota de objetos no sistema operacional Aurora. O capítulo traz uma breve introdução sobre o modelo de comunicação entre objetos em Aurora. Em seguida, descreve, detalhadamente, a solução proposta e desenvolvida. O capítulo também descreve o simulador do sistema de comunicação de objetos do Aurora, que foi desenvolvido para validar a solução proposta.

O capítulo 6 traz as considerações finais sobre o trabalho. O capítulo também apresenta sugestões para trabalhos futuros que possam contribuir e/ou melhorar a solução proposta e desenvolvida.

No anexo 1 é listado os códigos em C++ de todas as classes utilizadas e desenvolvidas para solucionar o problema da comunicação remota de objetos no sistema operacional Aurora.

Capítulo 2

Computação Distribuída

2.1 Introdução

Até a década 70, para se ter um grande potencial de processamento era necessário adquirir grandes computadores que variavam em preço e em performance. Os *mainframes* dominavam o mercado nesse período devido a sua capacidade de processar grandes quantidades de dados.

A utilização dos *mainframes* caracterizou um período denominado de computação centralizada. Tudo era processado pelo computador central (*mainframe*) desde simples comandos de usuários até grandes volumes de dados. Uma falha ou quebra no computador central causava a paralisação de todo o sistema.

Com o advento das redes e dos computadores pessoais obteve-se um novo modelo de processamento denominado de computação distribuída. A característica principal deste modelo é a descentralização da carga de processamento, ou seja, as tarefas (processos) são distribuídas entre as partes que compõem o sistema.

Apesar do modelo computacional distribuído ter vantagens sobre o modelo computacional centralizado, sua implementação é bastante complexa. A complexidade para desenvolver software distribuído e a heterogeneidade dos ambientes (sistemas operacionais e hardware) ainda são fatores a serem levados em consideração na adoção de tal modelo.

Um grande esforço está sendo feito pela comunidade da informática para tornar a computação distribuída cada vez mais utilizada. Soluções em nível de software e hardware vêm sendo desenvolvidas, tais como plataformas de desenvolvimento de sistemas distribuídos, sistemas operacionais distribuídos e hardware mais confiáveis.

O restante do capítulo está organizado como segue. A seção 2.2 descreve o modelo Cliente-Servidor, caracterizando-o e descrevendo as principais metodologias que vêm sendo empregadas para a sua implementação. A seção 2.3 aborda os conceitos de computação distribuída, suas características, vantagens e desvantagens. A comunicação em sistemas distribuídos é abordada na seção 2.4, descrevendo-se os principais mecanismos de comunicação, tais como: troca de mensagens, RPC e comunicação em grupo. A seção 2.5 traz a conclusão do capítulo.

2.2 Arquitetura Cliente-Servidor

A arquitetura Cliente-Servidor é caracterizada pela presença de processos clientes e processos servidores [45]. Os servidores são responsáveis por atender e processar pedidos dos clientes. Cada servidor pode oferecer um ou mais serviços aos clientes, tais como: serviço de gerenciamento de banco de dados, serviço de impressão, correio eletrônico autenticação de usuários entre outros.

A comunicação na arquitetura Cliente-Servidor é do tipo requisição-resposta (R/R - *Request-Reply*) [47], ou seja, os clientes solicitam algum tipo de serviço aos servidores e estes, por sua vez, atendem aos pedidos dos clientes. Este tipo de comunicação tem um nível de complexidade baixa, e por não se tratar de um protocolo orientado a conexão possui um baixo custo de comunicação.

A figura 2.1 ilustra um ambiente Cliente-Servidor, onde os servidores e clientes se comunicam através de uma rede local (LAN - *Local Area Network*) ou uma rede de longa distância (WAN - *Wide Area Network*).

As aplicações (processos) executas nos clientes são, em geral, mais ”leves” que as aplicações executas nos servidores, por exemplo: interface gráfica para acesso ao banco de dados, planilha eletrônica e processador de texto. Cada servidor pode

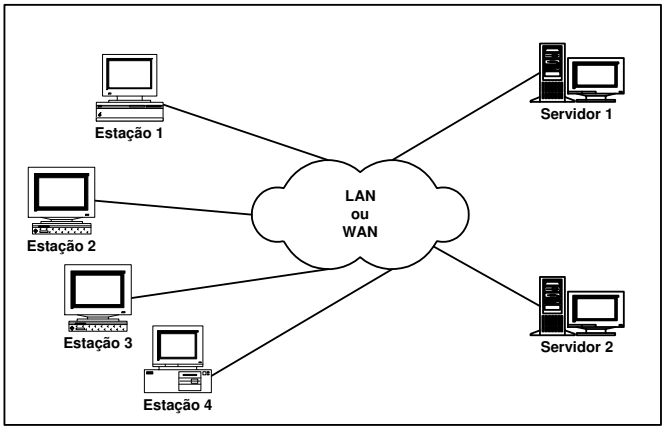

Figura 2.1: Arquitetura Cliente-Servidor

disponibilizar um ou vários tipos de serviços aos clientes, sendo divididos em: [11] [51]

- **Servidores Interativos:** o servidor recebe uma solicitação de um cliente, prepara a resposta e a envia de volta ao cliente para só depois atender uma solicitação de um outro cliente.

- **Servidores Concorrentes:** para cada nova solicitação de serviço de um cliente, é criado um novo processo servidor; desta forma, o servidor atende todas as solicitações concorrentemente, minimizando o tempo de resposta aos clientes.

- **Servidores Orientados a Conexão:** uma comunicação confiável entre um cliente e um servidor é fornecida pelo uso de um protocolo de comunicação orientado a conexão como o protocolo de transporte TCP da família de protocolos TCP/IP. Neste caso, toda a responsabilidade de verificação da integridade dos dados, controle e ordenamento de mensagens fica a cargo do protocolo de comunicação.

- **Servidores sem Conexão:** a utilização de protocolos de comunicação sem conexão, como o protocolo de transporte UDP, também da família de protocolos TCP/IP, não garante que as mensagens entre os clientes e os servidores sejam entregues de uma forma correta. Para garantir a integridade dos dados nesse caso, é necessário implementar o controle diretamente no servidor. A utilização de servidores sem conexão é mais apropriada em redes locais (*LAN's - Local Area Network*).

- **Servidores com Estado:** servidores com estado (*stateful*) se recordam das operações realizadas pelos clientes. As informações de estado são geralmente armazenadas em memória RAM (*Random Access Memory*). A principal desvantagem em se utilizar servidores com estado é a possibilidade de ocorrer inconsistência nas informações de estado guardadas pelo servidor. A inconsistência nas informações de estado pode ser causada pela perda de mensagens na rede ou pela queda do cliente ou do servidor.

- **Servidores sem Estado:** ao contrário dos servidores com estado, os servidores sem estado (*stateless*) não guardam qualquer informação de estado das operações realizadas pelos clientes. A premissa básica desses servidores é que toda operação realizada é dita idempotente, ou seja, qualquer repetição da operação não causará nenhum dano pois o resultado será sempre o mesmo.

Na arquitetura Cliente-Servidor, é possível que um servidor atue como cliente em determinada situação. Pode-se citar, como exemplo, um servidor qualquer que precise atualizar seu relógio físico de tempos em tempos com o relógio de um servidor de tempo.

2.3 Sistemas Distribuídos

Um sistema distribuído pode ser considerado como um sistema capaz de executar em um conjunto de máquinas com hardware heterogêneo. A comunicação entre seus componentes é feita através de troca de mensagens de forma transparente para seus usuários.

A utilização de sistemas distribuídos permite maior aproveitamento de recursos tanto de software quanto de hardware, permitindo que os componentes do sistema sejam acessíveis para um número maior de usuários.

A acesso aos recursos por um número maior de usuários é um fator importante para a adoção de sistemas distribuídos. Para Coulouris [13] uma das principais motivações para o uso de sistemas distribuídos é o compartilhamento de recursos. Re-

cursos podem ser entendidos como entidades de software (como uma base de dados) ou componentes de hardware (como uma impressora).

Para Tanenbaum [47] um sistema distribuído é aquele que roda em um conjunto de máquinas sem memória compartilhada, máquinas estas que mesmo assim aparecem como um único computador para seus usuários.

A definição de Tanenbaum para sistemas distribuídos, está relacionada com a característica de transparência. Os usuários de um sistema distribuído devem utilizar seus recursos como se estes fossem locais, mesmo que tais recursos estejam em outro local, possivelmente em outra estação ou servidor.

Além da transparência e do compartilhamento de recursos, os sistemas distribuídos possuem outras características que são descritas a seguir [13] [47]:

- **Flexibilidade:** um sistema distribuído deve ser flexível o suficiente para ser adaptado às crescentes necessidades do dia-a-dia. Essas necessidades podem ser desde a adição de novos serviços ao sistema, como um novo sistema de arquivos, ou até mesmo alterações no *kernel*.

- **Confiabilidade:** também conhecida como tolerância a falhas, o que implica dizer que, se um dos nós que compõem o sistema falhar, o restante do sistema não é afetado (talvez ocorra uma pequena queda no desempenho). Para atingir o nível de confiabilidade desejável, é necessário utilizar-se de técnicas de recuperação, tal como replicação dos serviços essenciais do sistema entre vários nós.

- **Desempenho:** este item relaciona-se diretamente com o tempo que o sistema precisa para atender a uma solicitação de um usuário. Os problemas potenciais em relação ao desempenho em um ambiente distribuído podem estar relacionados a gargalos no sistema. Estes gargalos podem ser meios de comunicação lentos ou serviços centralizados em um nó específico do sistema.

- **Escalabilidade:** um sistema distribuído deve ser escalável, ou seja, deve permitir ser estendido, tanto no número de usuários quanto na quantidade de recursos, sem que se perca desempenho.

Com um sistema distribuído é possível ter um ganho de processamento muitas vezes superior ao conseguido utilizando-se um sistema centralizado. Esta premissa está diretamente relacionada com a quantidade de processadores que compõem o sistema distribuído.

O balanceamento de carga é outro fator importante em sistemas distribuídos, onde é possível dividir a carga de processamento entre os vários processadores que fazem parte do sistema. Estas e outras vantagens dos sistemas distribuídos em relação aos centralizados estão listados na tabela 2.1 [47].

Tabela 2.1: Vantagens dos sistemas distribuídos

Velocidade	Um sistema distribuído pode ter um poder de processamento maior que o de qualquer mainframe
Distribuição inerente	Algumas aplicações envolvem máquinas separadas fisicamente
Economia	Os microprocessadores oferecem uma melhor relação preço/desempenho do que a oferecida pelos mainframes
Confiabilidade	Se uma máquina falhar, o sistema como um todo pode sobreviver
Crescimento incremental	O poder computacional pode crescer de acordo com a demanda

Apesar de todo o poder computacional conseguido com a utilização dos sistemas distribuídos, esses sistemas apresentam algumas desvantagens em relação aos sistemas centralizados. A tabela 2.2 lista as principais desvantagens dos sistemas distribuídos [47].

Tabela 2.2: Desvantagens dos sistemas distribuídos

Software	O desenvolvimento de software distribuído possui um nível de complexidade alto
Ligação em rede	A rede pode atingir níveis de saturação
Segurança	Os dados sigilosos também são acessíveis facilmente

A desvantagem, com relação à ligação em rede, não se restringe so-

mente a problemas de saturação, é possível que haja também problemas físicos, como interferências eletromagnéticas que podem gerar perda de mensagens na rede ou a falha de algum hardware de rede como um roteador também pode prejudicar o funcionamento do sistema.

Os problemas relacionados com a segurança dos dados podem ser resolvidos com o auxílio de mecanismos de segurança que empreguem criptografia, que tendem a dificultar qualquer acesso indevido a dados confidenciais.

2.4 Comunicação em Sistemas Distribuídos

Um sistema distribuído é caracterizado pelo seu grande potencial de processamento e pela facilidade de compartilhar recursos de forma transparente. Entretanto, para se fazer uso de tais características, é necessário ter um bom mecanismo de comunicação que facilite o acesso e a utilização desses recursos.

De acordo com Goscinski [17], um bom mecanismo de comunicação é necessário para facilitar o acesso a recursos compartilhados no ambiente distribuído de maneira uniforme, independentemente de linguagem de programação e de localização.

O mecanismo de comunicação não é responsável somente por permitir o acesso a recursos compartilhados no ambiente distribuído. Outros serviços, como balanceamento de carga e, principalmente, mobilidade, (que permite que processos/objetos migrem de uma máquina para outra), também são dependentes do mecanismo de comunicação.

Atualmente, existem várias propostas para a implementação de mecanismos de comunicação em sistemas distribuídos, e alguns sistemas implementam mais de um. Indiferente de qual proposta seja implementada, é importante que o mecanismo adotado seja responsável tanto pela comunicação local quanto pela comunicação remota garantindo sempre a transparência na comunicação.

Esta seção descreve os principais mecanismos de comunicação adotados em sistema distribuídos, tais como: troca de mensagens, comunicação em grupo e RPC.

2.4.1 Troca de Mensagens

O mecanismo de comunicação baseado em troca de mensagens é caracterizado pela existência de um processo emissor e um processo receptor. Usualmente, o sistema implementa duas chamadas de sistema, *send(process, message)* e *receive(process, message)*, para enviar e receber mensagens respectivamente.

A comunicação através da troca de mensagens não é utilizada somente em sistemas operacionais distribuídos. Sistemas operacionais convencionais baseados em microkernel como o MINIX [50], utilizam o mecanismo de troca de mensagens para a comunicação entre processos. O kernel desses sistemas basicamente implementa duas chamadas de sistema para comunicação: *send()* e *receive()*.

No modelo baseado em troca de mensagens, uma mensagem é constituída de um cabeçalho de tamanho fixo e um corpo, que contém os dados e pode ser tanto de tamanho fixo quanto de tamanho variado. O conteúdo de uma mensagem é definido pelo processo emissor.

A comunicação realizada através de troca de mensagens pode apresentar algumas formas alternativas, conforme as primitivas de comunicação adotadas. Tais alternativas definem se a primitiva de comunicação é síncrona ou assíncrona, bufferizada ou não-bufferizada, confiável ou não-confiável. A seguir serão descritas cada uma das possibilidades de implementação do mecanismo de comunicação baseado em troca de mensagens [47] [17] [44].

2.4.1.1 Primitivas Síncronas e Assíncronas

As primitivas síncronas caracterizam-se por bloquearem o processo transmissor até que a mensagem a ser transmitida seja completamente enviada e uma mensagem de retorno seja recebida.

De maneira similar, o processo receptor também fica bloqueado até que a mensagem a ser recebida tenha sido copiada completamente em seu *buffer*. A figura 2.2 mostra o esquema de funcionamento das primitivas síncronas.

Em alguns sistemas o receptor pode especificar de quem ele deseja re-

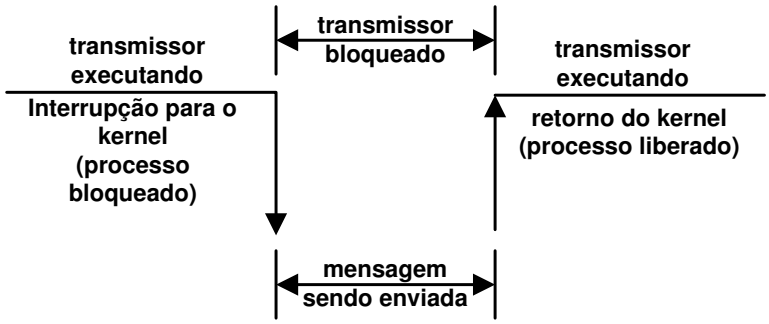

Figura 2.2: Primitiva Síncrona

ceber mensagens. Neste caso, o processo receptor fica bloqueado até receber uma mensagem de um determinado transmissor.

As primitivas assíncronas ao contrário das primitivas bloqueantes, possuem a característica de não bloquear o processo transmissor no envio de uma mensagem.

Quando o processo transmissor executa a chamada de sistema *send()*, ele é novamente posto execução imediatamente após o envio da mensagem, não necessitando aguardar o término da transmissão da mensagem como ocorre com a primitiva bloqueante. A figura 2.3 mostra o esquema de funcionamento das primitivas assíncronas.

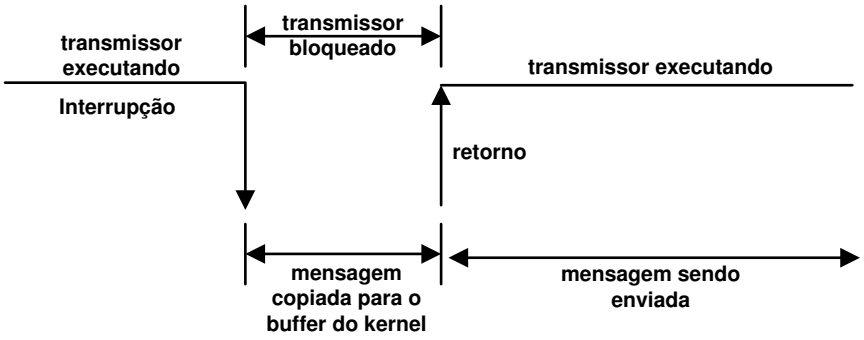

Figura 2.3: Primitiva Assíncrona

A desvantagem em se utilizar primitivas assíncronas está relacionada com a possibilidade do processo reescrever o *buffer* antes do término da transmissão da mensagem. O processo não tem controle do tempo que será gasto para transmitir uma mensagem armazenada no buffer, de modo que ele nunca saberá quando o *buffer* estará

livre para ser usado novamente.

O problema de reescrever o *buffer* pode ser resolvido de duas maneiras diferentes. A mensagem a ser transmitida é copiada do buffer de espaço de endereçamento de usuário para um buffer interno no espaço de kernel. A desvantagem desse método é o custo adicional gerado pela cópia do buffer de espaço de usuário para o espaço de kernel. Outra solução possível é interromper o processo transmissor logo após a mensagem ter sido enviada para avisá-lo de que o buffer de transmissão está livre novamente.

2.4.1.2 Primitivas Bufferizadas e Não-bufferizadas

As primitivas não-bufferizadas referenciam um endereço a um processo específico, como em uma chamada de sistema *receive(addr,m)* que informa ao *kernel* da máquina na qual ela foi executada que o processo está pronto para receber alguma mensagem de qualquer outro processo transmissor no endereço *addr*.

A chamada de sistema *receive()* informa ao kernel o endereço que o processo receptor está usando e onde as mensagens recebidas deverão ser colocadas. A cada mensagem recebida é associado um endereço e um buffer apontado por *m*; quando a mensagem chega, o *kernel* da máquina receptora copia a mensagem no buffer e libera o processo receptor.

Os problemas com as primitivas não-bufferizadas ocorrem quando um processo transmissor faz uma chamada de sistema *send* antes do processo receptor chamar *receive*. Neste caso o kernel da máquina receptora não saberá qual processo está utilizando o endereço especificado na mensagem que acabou de chegar, e desta forma a mensagem deverá ser descartada. A figura 2.4 demonstra o funcionamento das primitivas não-bufferizadas.

As primitivas bufferizadas, por sua vez, caracterizam-se por associarem a cada processo uma estrutura de dados chamada de caixa de correio (*mailbox*). Um processo interessado em receber mensagens deve solicitar ao kernel a criação de uma caixa de correio para seu uso e especificar um endereço para ser procurado nos pacotes de rede.

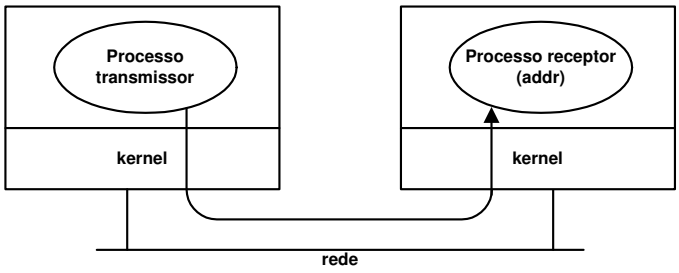

Figura 2.4: Primitiva não-bufferizada

Quando um processo possui uma caixa de correio (*mailbox*) associada a ele, todas as mensagens que são endereçadas para este processo devem ser copiadas para a sua respectiva *mailbox*. A chamada de sistema *receive* retira uma mensagem da caixa de correio; nos casos em que a caixa estiver vazia, o processo fica bloqueado assumindo que a chamada de sistema *receive* é bloqueante, ou continua sua execução em caso contrário. A figura 2.5 demonstra o esquema das primitivas bufferizadas.

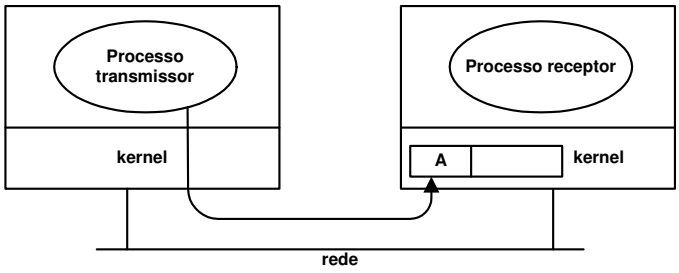

Figura 2.5: Primitiva bufferizada

Quando uma mensagem for enviada para um processo e a sua caixa de correio estiver cheia, o kernel do sistema deve ser responsável por tratar desse problema. Uma das soluções é descartar a mensagem que acabou de chegar. Outra solução é manter a mensagem por algum tempo até que alguma mensagem seja removida da caixa de correio do processo.

2.4.1.3 Primitivas Confiáveis e Não-Confiáveis

A confiabilidade em uma comunicação é a capacidade que se tem em determinar se realmente o receptor recebeu uma determinada mensagem endereçada a ele. As primitivas não-confiáveis não possuem meios de determinar se a mensagem foi entregue com sucesso ou não. Neste caso todo o controle da comunicação é de responsabilidade do programador do sistema.

Ao contrário das primitivas de comunicação não-confiáveis, as primitivas confiáveis possuem meios de saber se o receptor recebeu ou não uma determinada mensagem endereçada a ele. Quando dois processos trocam mensagens utilizando primitivas de comunicação confiáveis, sempre que o processo receptor receber uma mensagem correta ele devolve um ACK (*acknowledgment*), uma mensagem de reconhecimento, ao processo transmissor informando que a mensagem foi recebida com sucesso.

As primitivas de comunicação confiáveis possuem duas variantes com relação à mensagem de reconhecimento ACK. Quando um processo transmissor envia uma mensagem a um processo receptor, o kernel da máquina do servidor envia uma mensagem de ACK ao kernel da máquina do cliente; quando o servidor aprontar o pedido do cliente e enviar a mensagem com a resposta, o kernel da máquina do cliente deve enviar uma mensagem de ACK ao kernel da máquina do servidor. Neste modelo de implementação das primitivas confiáveis, todas as mensagens trocadas entre o cliente e o servidor são confirmadas individualmente como mostra a figura 2.6.

Figura 2.6: Primitiva Confiável - Mensagens Confirmadas Individualmente

Uma outra forma de implementar primitivas confiáveis é utilizar a própria mensagem de resposta como confirmação. Neste caso quando o servidor aprontar o pedido do cliente e enviar a mensagem com a resposta, esta será considerada a mensagem

de ACK do servidor para o cliente. Quando o cliente receber a mensagem de resposta, o kernel deve enviar uma mensagem de ACK ao kernel da máquina do servidor como mostra a figura 2.7.

Figura 2.7: Primitiva Confiável - Resposta como Confirmação

Uma terceira forma de implementar primitivas confiáveis utiliza de maneira híbrida os dois métodos citados anteriormente. Quanto o servidor recebe uma mensagem de solicitação do cliente, ele ativa um temporizador, caso a resposta seja enviada antes do temporizador expirar esta resposta será a própria confirmação, caso contrário é necessário enviar uma mensagem de confirmação antes da mensagem de resposta.

2.4.2 RPC (*Remote Procedure Call)*

A chamada remota de procedimento (RPC) (Birrell e Nelson, 1984) apud [47] baseia-se no conceito de chamada de procedimento local, ou seja, um processo cliente invoca um procedimento no processo servidor e fica aguardando o retorno da execução do procedimento pelo servidor. Entretanto no modelo RPC o procedimento chamado é remoto, reside em outra máquina que não a máquina do processo que chamou o procedimento.

A chamada remota de procedimento deve ser transparente; o processo chamador não deve saber se o procedimento chamado está ou não rodando em uma máquina diferente [47]. O mecanismo de RPC permite a execução de um procedimento remoto escondendo os detalhes da comunicação [42].

Quando um cliente chama um procedimento remoto, a chamada do procedimento e os respectivos parâmetros são passados para o stub (*proxy*) do cliente, que empacota os dados do procedimento em uma mensagem e solicita ao kernel que a envie

ao servidor onde se localiza o procedimento chamado.

No servidor o processo ocorre de maneira inversa, ou seja, quando chega uma mensagem o stub do servidor desempacota os dados desta mensagem (parâmetros do procedimento) identifica qual o procedimento que foi chamado e passa para ele os parâmetros para ser executado. Quando o servidor termina a execução do procedimento, devolve o resultado para o stub, este empacota o resultado em uma mensagem e solicita ao kernel que a envie ao cliente. No lado cliente, o stub recebe essa mensagem, desempacota os dados e passa para o processo que invocou o procedimento. A figura 2.8 demonstra o funcionamento do mecanismo de comunicação RPC.

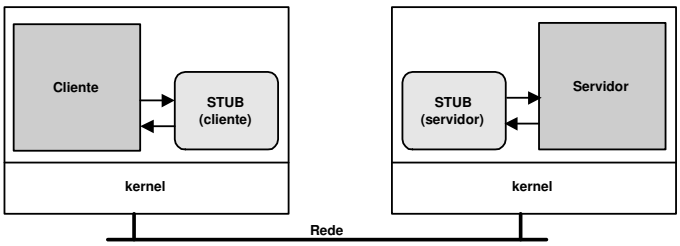

Figura 2.8: Chamada Remota de Procedimento

O stub é um conjunto de funções que executa a conversão dos dados e os detalhes de comunicação envolvidos na execução de um procedimento remoto [1]. Os stubs são gerados a partir da compilação de um arquivo que descreve a interface do servidor. Um arquivo de interface define os procedimentos implementados pelo servidor e seus respectivos parâmetros.

Após gerados os stubs do cliente e do servidor é necessário que o servidor, logo após ser posto para executar, exporte sua interface para que ela seja acessível pelos clientes. O servidor exporta sua interface enviando uma mensagem para um processo chamado ligador (*binder*). O ligador é responsável por fazer o registro da interface do servidor, para isso é necessário que o servidor lhe informe seu nome, versão, um identificador e um manipulador que é utilizado para permitir sua localização.

O manipulador depende do sistema, podendo ser um endereço IP, um endereço *ethernet* ou outro endereço dependendo da tecnologia de interligação utilizada.

Quando um cliente fizer uma chamada a um método remoto ele irá solicitar ao ligador o manipulador do servidor do método. O ligador devolverá para o cliente o manipulador do servidor e sua respectiva identificação. Neste caso, as mensagens enviadas pelo cliente para este manipulador serão recebidas pelo respectivo servidor.

A RPC não permite passar um ponteiro diretamente como parâmetro de um procedimento. No caso de estruturas simples como um vetor de caracteres, a RPC utiliza o esquema de cópia-restauração. O vetor de caracteres é enviado via mensagem para o servidor. Este, por sua vez, manipula/altera os dados do vetor e o devolve para o stub do cliente que atualiza os dados no cliente.

O objetivo da chamada remota de procedimento é manter escondido do usuário todos os procedimentos relativos a comunicação remota. Entretanto, é possível que ocorram falhas na comunicação. Abaixo são listadas as falhas que podem acontecer em uma chamada remota de procedimento [1] [47]:

- **O cliente não é capaz de localizar o servidor**: uma falha no servidor como um defeito de hardware ou diferentes versões da interface do cliente e do servidor podem ser responsáveis por causar esse tipo de falha.

- **Perda de mensagens solicitando serviços**: um temporizador é ativado logo após o envio da mensagem de solicitação. Se o temporizador expirar antes do recebimento da resposta ou mensagem de reconhecimento por parte do servidor, a mensagem de solicitação é retransmitida.

- **Perda de mensagens com resposta**: logo após o envio da mensagem de solicitação é ativado um temporizador, se a resposta não chegar neste período a solicitação é retransmitida. O problema desse método é com relação as operações não idempotentes, operações que não podem ser repetidas como a atualização de um saldo de uma conta bancário ou um contador. A solução para operações não idempotentes é fazer com que o servidor controle o que é mensagem original e o que é mensagem de retransmissão.

- **Queda do servidor**: este tipo de falha também está relacionado com a idem-

potência da operação, porém existem duas variações para essa classe de falha. No primeiro caso, o servidor recebeu a solicitação do cliente, processou e falhou antes do envio da resposta. No segundo caso, o servidor recebeu a solicitação do cliente e falhou antes de processá-la. Uma das técnicas de contornar o problema é continuar tentando até receber uma resposta, semântica de no mínimo uma vez. Um outra solução, conhecida como semântica de no máximo uma vez, define que o cliente deve desistir imediatamente. Uma terceira solução é não garantir nada ao cliente; assim o cliente não possui nenhuma garantia que a chamada será executada.

- **Queda do cliente**: quebra do cliente após chamar um método remoto. É necessário tratar os problemas com o processamento órfão, como processamento sendo realizado sem que exista um pai associado a ele.

Um fator importante a ser levado em consideração na adoção do mecanismo de comunicação baseado em RPC é o protocolo de rede a ser utilizado. Protocolos confiáveis tendem a gerar *overhead* na comunicação, entretanto a adoção de um protocolo confiável é uma boa opção no caso de sistema distribuídos implementados em redes de longa distância como a Internet. No caso de redes locais os protocolos confiáveis fazem com que o sistema tenha um baixo desempenho na comunicação. Neste caso o ideal é utilizar protocolos sem conexão, que não garantem a confiabilidade da comunicação.

O mecanismo de RPC é atualmente a solução mais utilizada na construção de sistemas distribuídos. Grande parte dos *middlewares* que dão suporte a computação distribuída utilizam RPC. Existem várias propostas de RPC para sistema orientados a objetos, entre eles o projeto NEXUS [53], um sistema operacional distribuído que implementa um mecanismo de RPC compatível com a tecnologia de objetos.

2.4.3 Comunicação em Grupo

A principal característica da comunicação em grupo é o conceito de comunicação um-para-muitos, ou seja, um processo transmissor envia uma mensagem para vários processos receptores que formam um grupo de processos. Todas as mensagens destinadas a um grupo devem ser recebidas por todos os membros desse grupo [47].

Os grupos são formados por uma coleção de processos que interagem entre si. Os grupos são dinâmicos, novos grupos podem ser criados ou grupos já existentes podem ser destruídos. Um processo pode ser membro de um ou mais grupos, e processos podem ser adicionados ou retirados de um grupo. A figura 2.9 representa a comunicação em grupo onde cada máquina representada na figura pode ter um ou mais processos que fazem parte do grupo.

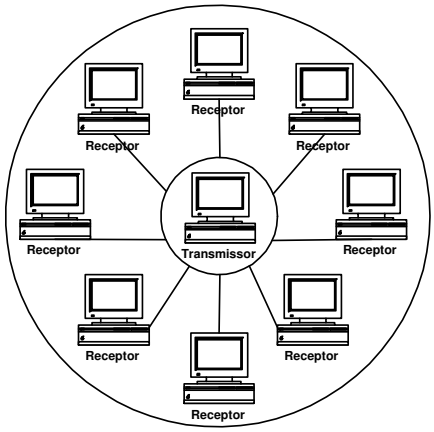

Figura 2.9: Comunicação em Grupo

A implementação da comunicação em grupo depende das características do hardware responsável por interligar os equipamentos. Em alguns casos é possível criar um endereço especial onde múltiplas máquinas poderão escutar este endereço. Quando um mensagem é enviada para um desses endereços, automaticamente ela é recebida por todas as máquinas que estão escutando este endereço. Este esquema é conhecido como *multicast*.

Quando o hardware de interligação não suporta *multicast*, é necessário utilizar o recurso de *broadcast*. Neste caso todos os processos do ambiente recebem a mensagem, porém os que não fazem parte do grupo identificado na mensagem devem descartá-la. Esse esquema soluciona o problema de comunicação em grupo porém gera muito tráfego na rede.

Uma outra forma de implementar a comunicação em grupo é utilizada quando a rede não suporta *multicast* e *broadcast*. Neste caso é necessário que o processo

transmissor envie a mensagem para cada processo pertencente ao grupo. Se o grupo possuir N processos é necessário o envio de N mensagens pelo processo transmissor. Este esquema é conhecido como múltiplos *unicast*.

A comunicação em grupo é um recurso bastante útil em sistemas distribuídos, principalmente quando se tem alguns serviços essenciais do sistema replicados. No entanto é necessário estabelecer algumas regras com relação aos grupos, regras que podem determinar quem pode se comunicar com o grupo e organizar hierarquicamente os membros do grupo. Essas regras são descritas abaixo:

- **Grupos Fechados e Grupos Abertos:** grupos fechados somente podem receber mensagens dos processos que fazem parte do grupo. Processos externos ao grupo não podem enviar mensagens para o grupo, somente para membros individuais do grupo. Ao contrário dos grupos fechados os grupos abertos permitem receber mensagens de processos externos ao grupo.

- **Grupos Igualitários e Grupos Hierárquicos:** quando um grupo está organizado de forma igualitária não existe nenhum coordenador para comandar as ações do grupo. Todos os processos de um grupo igualitário tem os mesmos direitos e deveres. Já em grupos organizados de forma hierárquica é necessária a presença de um processo coordenador que é responsável por dividir as tarefas entre os processos membros do grupo.

A adoção da comunicação em grupo em sistemas distribuídos requer algum mecanismo que gerencie os grupos. Uma forma de gerenciar grupos é feita através de um servidor de grupos, sendo este responsável pela inclusão ou exclusão de membros ao grupo e a criação e a destruição de grupos. Entretanto o uso de um servidor de grupo é uma solução centralizada; se o servidor falhar, deixa de haver a gerência dos grupos.

Uma alternativa ao servidor de grupos é o gerenciamento distribuído. Neste caso quando um processo deseja participar de um grupo ele envia uma mensagem solicitando ao grupo a sua inclusão. Da mesma forma, quando um processo deseja sair do grupo ele envia uma mensagem notificando a sua saída. A partir desse momento o

processo que se agregou ao grupo passa a receber todas as mensagens destinadas ao grupo e o processo que saiu não mais receberá as mensagens que são destinadas ao grupo. No caso de grupos fechados, que não aceitam mensagens de processos externos, é necessário utilizar uma mensagem especial que indica a inclusão de um processo ao grupo; somente esse tipo de mensagem externa é aceita pelos grupos fechados.

Na comunicação em grupo, cada grupo deve ser endereçado de forma única como acontece com os processos. Uma forma de endereçar grupos é através da utilização de predicados. A cada mensagem é atribuído um predicado (expressão booleana) a ser testada pelo processo receptor; se a avaliação for verdadeira, a mensagem é aceita; caso contrário, o processo deve descartar a mensagem.

Um fator importante na comunicação em grupo é a atomicidade, ou seja, toda mensagem enviada a um grupo deve ser recebida por todos os membros desse grupo. Quando um processo envia uma mensagem a um grupo ele não deve se preocupar se algum membro do grupo não recebeu a mensagem. Outro ponto importante a ser tratado na comunicação em grupo é o ordenamento das mensagens. Geralmente, as mensagens devem chegar ao receptor na mesma ordem com que foram transmitidas por cada processo transmissor.

Abaixo são especificados os quatro tipos de ordenação que são implementados nos mecanismos de comunicação em grupo [21]:

- **Sem ordem:** as mensagens são enviadas ao grupo sem nenhum ordenamento. Pode não ser adequado para muitas aplicações, entretanto possui um baixo *overhead* pois não necessita controle de sequenciamento das mensagens.

- **Ordenamento FIFO:** garante que todas as mensagens sejam entregues aos membros do grupo de acordo com a ordem que foram enviadas por cada transmissor.

- **Ordenamento causal:** quando duas ou mais mensagens possuem um relacionamento causal, ou seja, a mensagem B depende da mensagem A, é necessário que o sistema garanta a ordem de chegada das mensagens. No ordenamento causal as mensagens estão em ordenamento FIFO.

- **Ordenamento total:** cada membro do grupo recebe todas as mensagens na mesma ordem. Se duas mensagens (A e B) foram enviadas a um grupo, e o processo P1 recebeu A antes de B, então todos os outros membros do grupo devem receber primeiro A e depois B. A desvantagem do ordenamento total é a sua complexidade de implementação.

A escalabilidade é outro fator importante a ser levado em consideração na adoção do modelo de comunicação em grupo. Quando há um aumento excessivo na quantidade de membros nos grupos e o aumento da quantidade de grupos no sistema, os algoritmos utilizados na comunicação em grupo tendem a não funcionar corretamente devido à sua complexidade computacional.

2.5 Conclusão

Este capítulo abordou os aspectos relevantes da computação distribuída, suas vantagens e desvantagens e o modelo Cliente-Servidor. O capítulo também abordou os principais mecanismos de comunicação utilizados atualmente em sistemas distribuídos como troca de mensagens, RPC (*Remote Procedure Call*) e comunicação em grupo.

O modelo Cliente-Servidor é caracterizado pela presença de processos servidores e processos clientes. Geralmente processos servidores são mais pesados que os processos clientes. O modelo de comunicação utilizado em sistemas Cliente-Servidor é do tipo requisição-resposta (R/R), onde um cliente encaminha um pedido a um servidor, depois de processar o pedido, o servidor envia a resposta ao cliente. A principal característica do modelo R/R é o baixo custo na comunicação.

Sistemas distribuídos podem ser caracterizados por uma coleção de máquinas independentes interligadas através de uma rede de computadores executando software distribuído. Uma das principais motivações para o desenvolvimento de sistemas distribuídos é o compartilhamento de recursos, uma base de dados, uma impressora ou um disco de grande capacidade, de maneira totalmente transparente para seus usuários.

A comunicação em sistemas distribuídos pode ser implementada através

de troca de mensagens, RPC ou comunicação em grupo. A troca de mensagens utiliza duas primitivas de comunicação, *send()* e *receive()*, para enviar e receber mensagens respectivamente. O mecanismo de RPC baseia-se no conceito de chamada de procedimento local. Um objeto cliente pode invocar um método em um objeto servidor localizado em outro ponto da rede. A comunicação em grupo é baseada no conceito de comunicação um-para-muitos. Um objeto transmissor envia uma mensagem para vários objetos receptores que formam um grupo de objetos.

Capítulo 3

Ambientes para Computação Distribuída

3.1 Introdução

Os ambientes distribuídos podem ser formados por uma grande diversidade de software e de hardware. A heterogeneidade dos ambientes distribuídos torna difícil o desenvolvimento e a execução de aplicações nesses ambientes. Para minimizar a complexidade dos ambientes distribuídos faz-se necessário a utilização de sistemas específicos como plataformas para desenvolvimento de aplicações distribuídas e sistemas operacionais distribuídos.

Middleware ou plataforma para desenvolvimento de aplicações distribuídas, é uma camada de software situada entre a aplicação distribuída e o sistema operacional. Sua principal característica é a capacidade de esconder as diferenças dos ambientes distribuídos através de uma camada de software que provê um conjunto de serviços para as aplicações distribuídas.

Os sistemas operacionais distribuídos possuem um conjunto de serviços responsáveis por gerenciar os ambientes distribuídos de maneira uniforme, escondendo as diferenças existentes entre seus componentes, criando uma camada de abstração para as aplicações que executam sobre eles.

Os sistemas operacionais distribuídos possuem algumas vantagens com relação as plataformas de desenvolvimento de aplicações distribuídas. Uma das vantagens é a inexistência de uma camada intermediária entre a aplicação distribuída e o sistema operacional, o que ocasiona um ganho de desempenho. Outra vantagem, é que as aplicações que executam sobre esses sistemas são naturalmente distribuídas pelo ambiente.

Este capítulo está organizado como segue. A seção 3.2 descreve as principais plataforma para o desenvolvimento de aplicações distribuídas: DCE, CORBA, DCOM e Java/RMI. A seção 3.3 apresenta estudos de caso de sistema operacionais distribuídos, dando ênfase ao mecanismo de comunicação adotado em cada um deles. Na seção 3.4 é feita uma conclusão com relação aos assuntos abordados no capítulo.

3.2 Plataformas para Desenvolvimento de Aplicações Distribuídas

A necessidade de comunicação entre sistemas heterogêneos fez com que surgisse um novo conceito de software, os chamados *middleware*. Um *middleware* provê um alto nível de abstração mascarando as características dos diferentes tipos de protocolos de rede, hardware, sistemas operacionais e linguagens de programação [13].

O *middleware* atua como um intermediador entre a aplicação e o sistema operacional, oferecendo serviços para o desenvolvimento de aplicações distribuídas, provendo uma camada de software que permite acesso uniforme a sistemas diferentes [44]. A figura 3.1 demonstra uma visão lógica de um *middleware*.

Existem vários tipos de *middleware* disponíveis no mercado, que permitem o desenvolvimento de aplicações distribuídas. Existe *middleware* proprietário como o DCOM da Microsoft e outros de arquitetura aberta como o DCE da Open Group e o CORBA da OMG, sendo este último um dos mais difundidos atualmente.

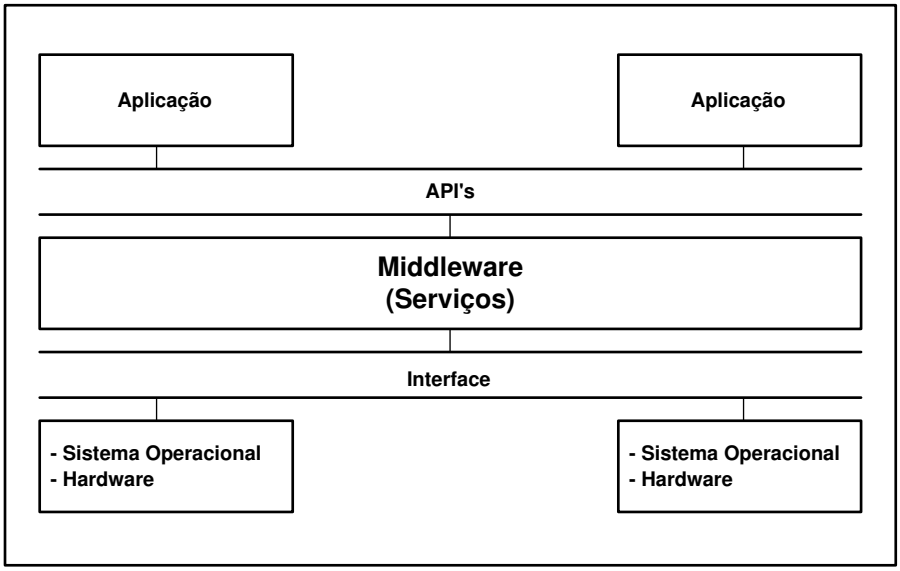

Figura 3.1: Visão lógica de um middleware

3.2.1 DCE

O DCE (*Distributed Computing Environment*) [36] é um produto da OSF (*Open Software Foundation*), atualmente chamada de Open Group, uma organização voltada para o desenvolvimento de software aberto e portável. A OSF foi fundada em 1988, com a ajuda das empresas IBM, DEC, BULL, Hewllet-Packard, Nixdorf, Apollo, Phillips, Siemens e Hitachi. Constitui-se de uma organização sem fins lucrativos aberta a participantes de várias categorias, incluindo fornecedores de software, hardware, instituições educacionais, governamentais e outros.

As aplicações distribuídas sobre o DCE precisam interagir somente com os mecanismos de software de alto nível disponíveis, sem se preocupar com a forma com que a comunicação realmente ocorre em nível físico ou de rede.

A arquitetura do DCE esconde as complexidades físicas do ambiente, oferecendo uma camada de simplicidade lógica, composta de um conjunto de serviços que podem ser usados separadamente, ou em combinação, para formar um ambiente de computação distribuída mais fácil de ser compreendido [39].

O DCE é um conjunto de serviços que permite o desenvolvimento de

aplicações distribuídas de forma transparente em ambientes heterogêneos [51]. A figura 3.2 mostra a arquitetura DCE em camadas, definindo desde os serviços de mais baixo nível (sistema operacional) até os serviços de mais alto nível (aplicações).

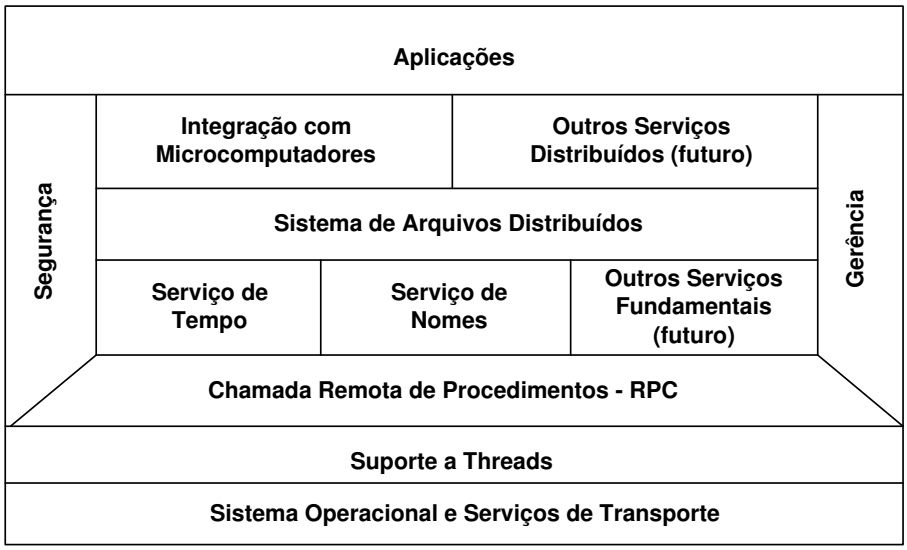

Figura 3.2: Arquitetura DCE

Segue um definição dos principais serviços oferecidos pelo DCE:

- **Serviço de Segurança:** conjunto de mecanismos que suportam a comunicação segura entre cliente e servidor, fornecendo meios de autenticação, controle de acesso, integridade e privacidade das informações dos usuários.

- **Sistema de Arquivos Distribuídos:** estende o sistema de arquivos local para um sistema de arquivos em rede; desta forma os usuários podem acessar seus documentos/dados de qualquer máquina que faça parte do sistema distribuído.

- **Serviço de Tempo e Sincronização:** sincroniza o relógio de todas as máquinas que fazem parte do sistema distribuído. Este serviço segue o padrão UTC (*Universal Time Coordinated*) podendo inclusive interoperar com o padrão NTP (*Network Time Protocol*).

- **Serviço de Diretório:** permite especificar logicamente um nome aos recursos pertencentes ao ambiente DCE; desta forma as aplicações acessam os recursos através

deste nome lógico, não sendo necessário saber a localização exata de cada recurso.

- **Serviço de Threads:** fornece suporte à execução paralela através da criação e gerenciamento de threads. As threads são fluxos de execução dentro de um processo cliente ou servidor permitindo desempenhar ações concorrentes.

- **Serviço de Gerenciamento de Rede:** oferece meios para as aplicações específicas acessarem informações de gerenciamento, usando protocolos de gerenciamento de rede, tais como CMIP e SNMP.

O DCE adota o modelo cliente-servidor, com utilização do mecanismo de RPC (*Remote Procedure Call*) para comunicação. A chamada RPC se utiliza de protocolos que administram aspectos específicos de transporte e de rede, gerenciam conexões e, em alguns casos, podem fornecer algum suporte para tratamento de falhas de servidores ou de serviços de rede.

A sintaxe de uma chamada RPC, com seus parâmetros de entrada e saída, torna-se conhecida de um cliente através de uma descrição de interface codificada em uma linguagem IDL (*Interface Definition Language*), semelhante à linguagem C. A compilação de um arquivo de descrição de interface gera rotinas STUB de cliente e servidor.

O DCE trabalha com o conceito de células, que nada mais são que um conjunto de unidades gerenciáveis independentes. A célula é formada por um grupo de usuários, sistemas e recursos que têm um propósito comum e compartilham serviços.

O OO-DCE (*Object Oriented* DCE) [15] 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 ao mapeamento das definições de interfaces de componentes DCE, descritas em IDL, para classes de objetos clientes e servidores codificados na linguagem C++.

3.2.2 CORBA

O CORBA (*Common Object Request Broker Architecture*) [33] é uma especificação da OMG (*Object Management Group*), um consórcio de várias empresas que visa promover a utilização do modelo de programação orientada a objetos no desenvolvimento de aplicações distribuídas.

A OMG especificou a arquitetura OMA (*Object Management Architecture*), um ambiente que tem como principal característica suportar aplicações cooperantes compostas por objetos distribuídos. A figura 3.3 demonstra os quatro elementos principais da arquitetura OMA [35].

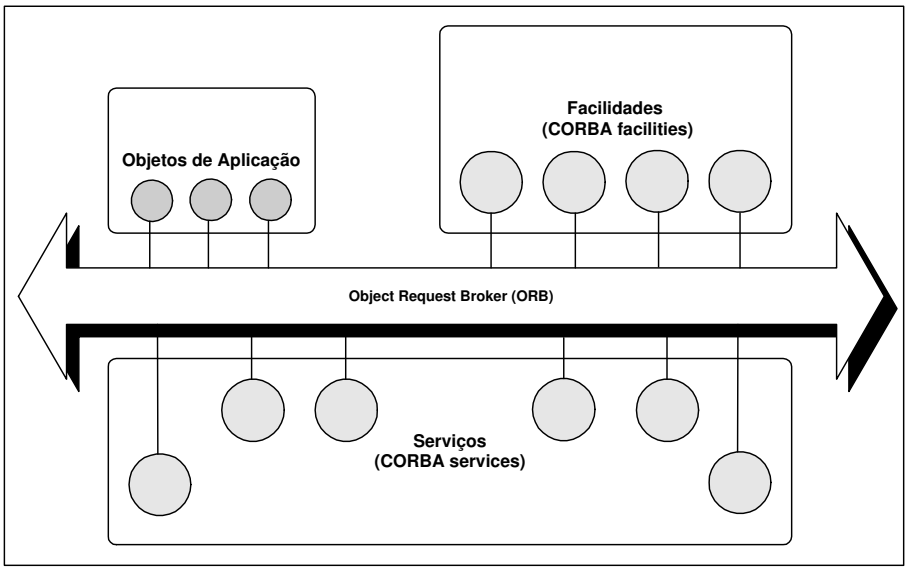

Figura 3.3: Arquitetura OMG/OMA

Ao invés de aplicações, a OMG produz especificações que tornam a computação orientada a objetos possível. Este modelo baseado em objetos permite que métodos de objetos sejam ativados remotamente, através de um elemento intermediário chamado ORB (*Object Request Broker*) situado entre o objeto e o sistema operacional.

O ORB, barramento de objetos, é o componente mais importante da arquitetura proposta pela OMG. Ele permite que objetos façam e recebam requisições de métodos transparentemente em um ambiente distribuído heterogêneo.

Uma requisição de método originada por um cliente é enviada a uma implementação do objeto (servidor) através do ORB. O ORB é o elemento intermediário responsável por encontrar o objeto ao qual se destina a requisição, e enviar os parâmetros da requisição no formato reconhecido por este. O ORB também faz o processo inverso, retornando os parâmetros de saída da requisição, se houver algum para o cliente.

O CORBA oferece um pacote de serviços de objetos que facilitam o trabalho do programador de aplicação, permitindo que ele concentre seus esforços no desenvolvimento dos objetos, sem ser preocupar com os serviços no nível de sistema. Este pacote de serviços é padronizado pela OMG dentro da arquitetura OMA, chamado de COSS (*Common Object Services Specifications*), formando uma coleção de serviços a nível de sistema.

O COSS pode ser entendido como uma extensão ou como uma complementação das funcionalidades do ORB [34]. O COSS oferece serviços como: persistência, controle de concorrência, transação, externalização, atomicidade, segurança, confiabilidade entre outros.

As facilidades comuns são coleções de serviços de propósitos gerais, divididas em facilidades horizontais e verticais. As facilidades horizontais são utilizadas por várias aplicações independente da área da aplicação. As facilidades verticais são utilizadas em aplicações específicas como medicina e simulação distribuída.

No CORBA, interfaces de objetos são definidas por uma linguagem IDL. Através da IDL (*Interface Definition Language*) são declaradas os dados e métodos que podem ser acessados externamente contidos no objeto correspondente à interface que está sendo descrita.

Para fazer uma requisição de um serviço a um objeto, o cliente pode utilizar stubs gerados na compilação da descrição da interface da implementação do objeto, ou, alternativamente, pode montar a requisição através da interface de invocação dinâmica - DII (*Dynamic Invocation Interface*) adicionadas a um depósito de interfaces (*Interface Repository*), permitindo o acesso em tempo de execução à informação relativa aos serviços implementados pelo objeto e à forma de acesso destes.

A interoperabilidade entre objetos torna-se possível através da adoção

do protocolo IIOP (*Internet Inter-ORB Protocol*), que trata-se de um protocolo padronizado pela OMG que permite comunicação entre diferentes implementações de ORB´s compatíveis com a especificação CORBA. O IIOP especializa o protocolo TCP/IP (*Transport Control Protocol/Internet Protocol*), padrão de facto para comunicação em redes, de modo a otimizá-lo para transmissão dos tipos de dados utilizados pelo CORBA.

3.2.3 DCOM

O DCOM (*Distributed Component Object Model*) [19] é uma arquitetura para desenvolvimento de aplicações distribuídas proprietária da Microsoft, lançado em 1997, que está disponível para o sistema operacional Windows [18].

O DCOM é uma extensão do COM (*Component Object Model*), com o objetivo de suportar a comunicação entre objetos em diferentes computadores. O COM define como deve ser a interação entre os objetos e seus clientes, sem a intermediação de qualquer componente do sistema.

O DCOM é utilizado junto com a tecnologia ActiveX, também desenvolvida pela Microsoft, que permite que documentos MS-Word, planilhas e outros sejam disponibilizados para acesso através de Web browsers e *Applets* Java [39].

A tecnologia DCOM está baseada no modelo de objetos; a idéia é que o sistema operacional Windows se torne uma grande coleção de objetos ActiveX e estes se comuniquem através do DCOM, sendo considerado um ORB para o ActiveX.

A tecnologia DCOM não se preocupa em ser compatível com várias linguagens de programação, sendo um fator limitante desta tecnologia. Para solucionar o problema de interoperabilidade, a Microsoft lançou uma norma binária, e assim qualquer linguagem de programação que entenda essa norma pode criar e utilizar objetos DCOM.

A comunicação entre os clientes e os componentes é realizada da mesma forma tanto para os localizados localmente, quanto para os localizados remotamente [16].

Quando componentes estão sendo executados em diferentes processos, há a necessidade de ser estabelecida uma comunicação entre processos, a qual é feita pelo sistema operacional através de mecanismo de comunicação interprocessos. O COM

permite esta comunicação através da biblioteca de execução COM/DCOM que estabelece o *link* entre o cliente e o componente. A figura 3.4 mostra a comunicação entre um cliente e o componente executando na mesma máquina.

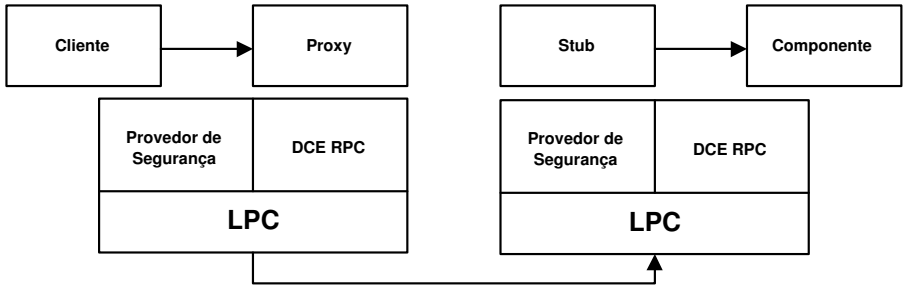

Figura 3.4: Arquitetura DCOM para Comunicação Local

Na comunicação entre um processo cliente e um componente remoto, o DCOM utiliza o protocolo de rede para estabelecer a comunicação entre eles. O stub e o *Proxy*, fornecem serviços orientado a objetos ao cliente e ao componente utilizando RPC (*Remote Procedure Call*) e o provedor de segurança para gerar pacotes de redes padrões que sejam compatíveis com o protocolo padrão DCOM. A figura 3.5 mostra a comunicação entre um cliente e um componente rodando em máquinas diferentes.

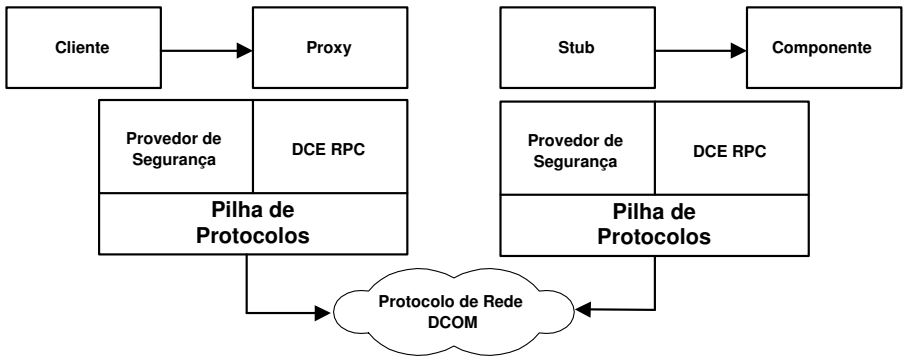

Figura 3.5: Arquitetura DCOM para Comunicação Remota

O protocolo DCOM, conhecido como ORPC (*Object RPC*) é um conjunto de serviços que extendem o DCE RPC implementado pelo Windows. Feito para comunicação de objetos, o ORPC gerencia como as chamadas a métodos remotos são

gerenciadas, como os objetos são representados e mantidos.

O DCOM implementa duas linguagens de definição de interface, Microsoft IDL e ODL (*Object Definition Language*). A IDL é utilizada para gerar stubs remotos, enquanto que a ODL é para gerar metadados das classes. A compilação de ambos os arquivos de interfaces, IDL e ODL, é feita pelo compilador MIDL (*Microsoft Interface Definition Language*).

A ODL é uma linguagem mais neutra que a IDL. Através dela, podem ser descritas interfaces dinâmicas e estáticas, além de toda a estrutura de uma classe DCOM.

O DCOM não suporta o conceito de herança na definição de interface, porém os objetos DCOM suportam múltiplas interfaces através dos mecanismos de delegação e agregação, isto é, uma classe pode incorporar múltiplas interfaces já existentes, reutilizando a sua definição.

Cada classe no DCOM é identificada por um valor de 128 bits, que é representado pelo CLSID (*Class Identifier*), que associa uma classe de objetos a uma (DLL) ou aplicação (EXE) no sistema de arquivos. O DCOM mantém uma base de dados denominada *system registry* que armazena todos os identificadores correspondentes aos servidores instalados no sistema, isto é, existe nesta base de dados um registro de cada identificador e respectiva localização do arquivo DLL ou EXE.

Quando um cliente pretende criar uma instância de uma classe DCOM e utilizar os seus serviços, o DCOM consulta a base de dados *system registry*. Desta forma, o cliente necessita conhecer o identificador, o que o mantém independente da localização específica do arquivo DLL ou EXE no sistema. Se o identificador solicitado não for encontrado na base de dados local, são utilizados algoritmos para a sua localização na rede. A entidade responsável pela localização é o serviço SCM (*Service Control Manager*).

O acesso de um cliente a uma interface é realizado através de ponteiros para um vetor de funções de ponteiros, chamado de tabela virtual *vtable (Virtual Table)*. As funções da tabela virtual correspondem aos métodos de implementações dos objetos do servidor. Cada objeto pode possuir uma ou mais tabelas virtuais que definem o contrato entre a implementação do objeto e seus clientes.

Os objetos DCOM não mantêm o estado da conexão, portanto um cliente não pode se reconectar ao mesmo objeto com o mesmo estado. A conexão será feita apenas com um ponteiro de uma interface da mesma classe.

Toda vez que um cliente solicitar por um objeto de uma classe específica, o DCOM deverá carregar o servidor e requisitar a este que crie um objeto desta classe. Uma classe denominada *factory* deve ser fornecida pelo servidor para a criação do mesmo. Um ponteiro para a interface primária do objeto será retornado ao cliente após a criação deste.

A Microsoft está investindo em uma nova tecnologia que permite o desenvolvimento de aplicações distribuídas. A plataforma .NET [29] é um conjunto integrado de ferramentas e serviços que visa promover o desenvolvimento de aplicações distribuídas na Internet. Para a Microsoft, o .NET é uma forma de comunicação de dados via Internet.

O .NET é multiplataforma, permitindo que aplicações .NET executem de celulares a estações de trabalho. Uma das novidades do .NET são os *WebServices*, funções ou objetos que contém regras de negócios, e residem em um servidor Web. Utilizam protocolos que facilitam a comunicação entre sistemas, independente do sistema operacional e da linguagem de programação.

3.2.4 JAVA/RMI

A linguagem de programação Java [14], desenvolvida pela empresa Sun Microsystems, inicialmente era parte de um projeto cujo objetivo era o desenvolvimento de um ambiente apropriado para a implementação de programas para produtos eletrônicos em geral.

A popularidade da linguagem Java se deu a partir de 1994, na medida em que passou a ser vista como uma plataforma para o desenvolvimento de programas para a Internet [2].

Uma das grandes vantagens da linguagem Java é a independência de plataforma, tanto de software quando de hardware. Como Java é uma linguagem in-

terpretada, quando os programas escritos em Java são compilados, é gerado um código intermediário conhecido como *byte-code*.

A execução de programas escritos em Java é feita através de uma máquina virtual (JVM - *Java Virtual Machine*), que interpreta os *byte-codes* Java para a plataforma em que se quer executar a aplicação.

A linguagem de programação Java oferece um mecanismo nativo da própria linguagem que permite que métodos sejam invocados remotamente. Tal mecanismo é chamado do RMI (*Remote Method Invocation*) [46] e permite que objetos em máquinas diferentes possam se comunicar como se estivessem na mesma máquina.

A RMI é a implementação da RPC (*Remote Procedure Call*) para a linguagem de programação Java. As aplicações RMI comportam-se como uma arquitetura cliente-servidor, na qual clientes invocam métodos que atuam sobre objetos nos servidores. A RMI pode ser utilizada tanto para invocações de métodos em objetos remotos, como para invocações de métodos de objetos locais.

Quando é feita a invocação de um método de um objeto remoto, o cliente RMI atua na verdade sobre um objeto local que se faz passar pelo objeto remoto. Esse objeto local é chamado de stub e age como se fosse um *proxy* do objeto remoto, possibilitando ao cliente a transparência na comunicação, ou seja, escondendo do cliente os serviços providos pelo protocolo de transporte.

O código stub é gerado a partir de um compilador RMIC (*Remote Method Invocation Compiler*). O stub é uma visão do objeto remoto que contém somente os métodos remotos do objeto. O stub roda no lado do cliente e representa o objeto remoto no espaço de endereçamento do cliente [54]. A figura 3.6 mostra a arquitetura do Java/RMI.

Para invocar um método remoto, o cliente precisa identificar o objeto no qual o método atua. A referência para um objeto remoto é feita de duas maneiras; na primeira, o cliente recebe uma referência como o valor retornado por um método. No segundo caso, o cliente utiliza o serviço *registry*.

Através do serviço *registry*, os clientes obtêm referências para objetos remotos com nomes que identificam os serviços prestados pelos objetos [5]. O servidores

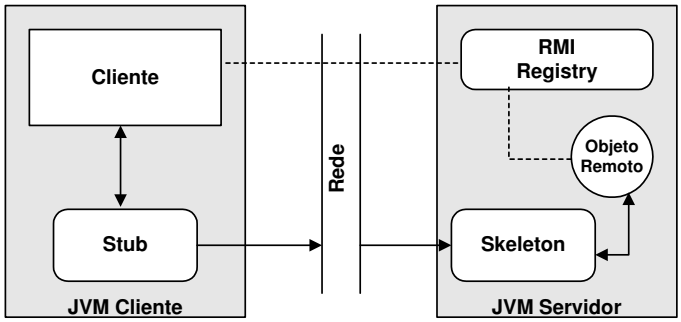

Figura 3.6: Arquitetura do Java/RMI

RMI utilizam o *registry* para associar um nome ao serviços que disponibilizam.

A figura 3.7 demonstra um exemplo de uma aplicação RMI. O servidor ao ser posto em execução, registra a interface do objeto no serviço de *registry* (1). O cliente faz um *lookup* ao serviço de *registry* para obter uma referência ao objeto remoto (2). Após o cliente obter a referência para o objeto remoto, ele invoca um de seus métodos diretamente no servidor (3).

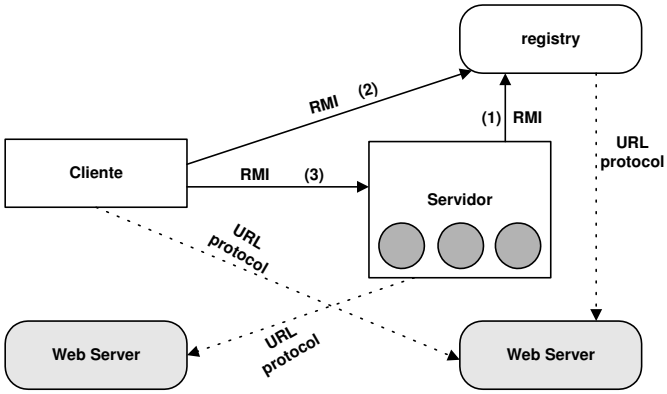

Figura 3.7: Exemplo de utilização do registry

Uma vez que um método ou serviço de um objeto é registrado como sendo remotamente acessível, um cliente pode pesquisar (*lookup*) esse serviço e receber uma referência que o permita utilizar esse serviço, ou seja, invocar o método.

Os objetos remotos em uma aplicação RMI estão contidos nos servidores. Os objetos remotos duram enquanto existirem referências para eles mantidas pelos clientes.

3.3 Estudos de Caso

Esta seção apresenta alguns estudos de caso de sistemas operacionais distribuídos abordando suas características principais e o mecanismo de comunicação implementado em cada um deles.

3.3.1 Sistema Operacional Amoeba

O sistema operacional Amoeba [31] [48] é um projeto de pesquisa na área de computação paralela e distribuída da Vrije Universiteit em Amsterdã, Holanda. O projeto teve início em 1981, e seu principal objetivo era a construção de um sistema operacional distribuído totalmente transparente.

O projeto Amoeba também visa fornecer um ambiente para experiência em programação paralela e programação distribuída. O Amoeba trabalha com o conceito de grupos de processadores, ou seja, um conjunto de processadores onde cada processador possui sua memória local e sua própria conexão a rede.

A principal vantagem em trabalhar com grupo de processadores é que cada processador do grupo pode ter características de hardware distintas, e com isso um grupo de processadores pode ser constituído por um conjunto de processadores heterogêneos. A alocação dos processadores para a execução de uma tarefa é feita de forma dinâmica pelo sistema operacional.

O Amoeba é baseado na arquitetura cliente-servidor, sendo que os servidores são responsáveis por oferecerem serviços aos clientes. O serviços são implementados como objetos. Para um cliente invocar um serviço de um servidor qualquer, é necessário que ele faça uma chamada remota de procedimento, sendo que o servidor pode estar na mesma máquina que o cliente ou em outra máquina na rede.

Para um cliente acessar um objeto de um servidor é necessário que ele possua a capacidade deste objeto. As capacidades no Amoeba são como *tickets* que protegem e identificam os objetos. Quando um cliente faz uma chamada para a criação de um objeto em um servidor, ele recebe como retorno da chamada, a capacidade do respectivo objeto.

O sistema operacional Amoeba suporta dois modelos de comunicação. O primeiro é baseado no mecanismo de RPC: o cliente invoca um método em um servidor remoto. O outro mecanismo é a comunicação em grupo: mensagens enviadas a um grupo de processos por meio de *broadcast* e *multicast*.

Todos os serviços no Amoeba são implementados através de servidores, que possuem uma interface procedural que os clientes podem invocar. As interfaces dos servidores são conhecidas como stubs e são responsáveis pelo empacotamento e desempacotamento dos parâmetros e também responsáveis por passarem ao kernel a mensagem a ser enviada.

Toda a comunicação entre um cliente e um servidor se dá através de uma porta, um endereço lógico de 48 bits que pode ser similarmente associado a um endereço IP (*Internet Protocol*) ou um endereço *ethernet*. As chamadas remotas de procedimento no Amoeba adotam a semântica **"no máximo uma vez"** o sistema garante que a chamada será executada somente uma vez.

As chamadas remotas implementadas no Amoeba utilizam três chamadas de sistema para efetivarem a comunicação. A primeira delas é ***getrequest*** utilizada por servidores aguardando mensagens em uma porta específica. A outra chamada de sistema é ***putreply*** também utilizada pelos servidores, para o envio de mensagens de resposta. A última chamada é ***trans***, utilizada pelos clientes para enviarem mensagens para os servidores e aguardarem o retorno da mensagem.

O Amoeba protege as portas com a utilização de uma função criptográfica que associa a cada porta um par de portas denominadas ***getport***, porta privada conhecida somente pelo servidor, e ***putport***, porta pública conhecida por todos os outros componentes do sistema. A função que relaciona as duas portas é: ***putport = F(getport)***.

O outro mecanismo de comunicação adotado pelo Amoeba é a comunicação em grupo [21]. Um grupo de processos cooperantes que executam alguma tarefa ou fornecem algum tipo de serviço. Os grupos no Amoeba são fechados; a única maneira de processos externos se comunicarem com o grupo é através de RPC a um processo individual do grupo.

O Amoeba implementa várias chamadas de sistema para a comunicação

em grupo, sendo que a mais importante é ***SendToGroup***, utilizada para enviar uma mensagem segura a todos os membros do grupo. A comunicação em grupo no Amoeba utiliza *broadcast* confiável; quando as aplicações iniciam suas tarefas é eleita, por meio de algoritmos eletivos, uma das máquinas para ser sequenciadora.

A máquina sequenciadora é responsável por atribuir um número seqüencial e controlar as mensagens enviadas por meio de *broadcast*. Quando um processo necessita enviar uma mensagem em *broadcast*, este envia uma mensagem ponto-a-ponto para a máquina sequenciadora. A máquina sequenciadora atribui um número seqüencial a mensagem e a envia por meio de *broadcast*.

Para dar suporte à comunicação, o Amoeba implementa um protocolo proprietário a nível de rede chamado de FLIP (*Fast Local Internet Protocol*) [22]. O FLIP foi projetado para suprir as deficiências encontradas em outros protocolos de nível de rede para uso em sistemas distribuídos. O protocolo FLIP possui as seguintes características:

1. Suporte a chamadas remotas a procedimentos (RPC);
2. Suporte a comunicação grupal;
3. Suporte a migração de processos;
4. Garante a segurança dos pacotes de rede;
5. Gerência de rede;
6. Compatibilidade tanto com redes locais quanto com redes de longa distância.

A vantagem do protocolo FLIP é que ele endereça processos e não máquinas. A cada processo no sistema é associado um endereço FLIP, um número randômico de 64 bits. Quanto ocorre a migração do processo seu endereço permanece o mesmo, não sendo necessário atribuir um novo endereço ao processo. A figura 3.8 demonstra duas redes sendo interligadas utilizando o protocolo FLIP.

Todos os pacotes passados para o nível FLIP, sejam provenientes de uma RPC ou comunicação em grupo, são endereçados através de endereços FLIP. Para a

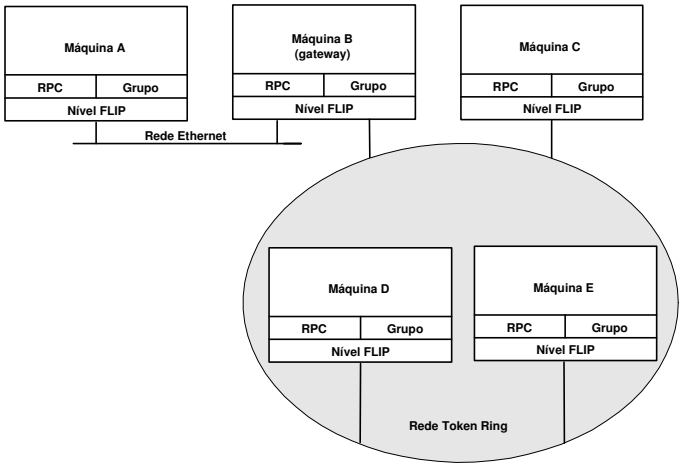

Figura 3.8: Exemplo de Interligação com o Protocolo FLIP

transmissão desses pacotes, o protocolo FLIP converte os endereços FLIP em endereços de rede. Para que a conversão de endereços seja efetivada, o protocolo FLIP mantém uma tabela de roteamento.

3.3.2 Sistema Operacional Solaris MC

O Solaris MC [23] é um protótipo de um sistema operacional distribuído para multicomputadores. O sistema provê um alto nível de abstração do ambiente, fazendo com que o conjunto de nós que fazem parte do sistema pareça para o usuário como um simples computador executando o sistema operacional Solaris.

O Solaris MC utiliza o modelo de objetos para a definição de seus componentes. O projeto também baseia-se no sistema operacional Spring [30], um sistema distribuído totalmente orientado a objetos. Uma das metas do projeto é demonstrar que sistemas operacionais convencionais podem se adequar ao modelo distribuído com poucas modificações. O Solaris MC foi construído sobre o sistema operacional Solaris.

Para abstrair os detalhes do sistema, o Solaris MC preserva o conjunto de ABI/API do Solaris, permitindo que aplicações construídas para rodarem no sistema operacional Solaris possam ser executadas no Solaris MC sem qualquer modificação.

A maior motivação para o desenvolvimento do Solaris MC é a capaci-

dade em se construir sistemas computacionais utilizando processadores baratos interligados através de uma rede de alta velocidade. O projeto Solaris MC também possui alguns interesses com relação à tecnologia adotada:

1. Extender o sistema operacional Solaris;
2. Manter a compatibilidade com o conjunto de ABI/API do Solaris;
3. Suportar alta disponibilidade;
4. Uso da linguagem de programação C++, IDL e CORBA para o desenvolvimento de componentes do kernel;
5. Usar a tecnologia do sistema operacional Spring.

O mecanismo de comunicação adotado pelo Solaris MC é baseado no CORBA. A vantagem é que o CORBA é totalmente compatível com o modelo de objetos, e tanto objetos remotos quanto objetos locais são acessados da mesma maneira, através da sua descrição de interface, não sendo necessário implementar mecanismos de comunicação distintos.

Os componentes do Solaris MC são objetos que possuem sua interface definida através de uma linguagem de definição de interface (IDL). O Solaris MC implementa seu próprio ORB chamado de sistema *runtime*. O ORB é responsável por controlar referências a objetos, empacotar e desempacotar parâmetros, dar suporte a comunicação remota através de RPC e recuperar falhas na comunicação. A figura 3.9 mostra as diferentes camadas do ORB do Solaris MC.

Cada camada do ORB do Solaris MC é responsável por uma tarefa na comunicação. A camada de manipulação é responsável por controlar as referências aos objetos do sistema. A camada de portas extendidas implementa o mecanismo de RPC para comunicação. As *XDOORS* são extensões do mecanismo de portas do Solaris. Na camada de transporte é definido a tecnologia de rede empregada e o protocolo de rede utilizado para comunicação. A camada de transporte também possui um conjunto de *buffers* que são utilizados para o armazenamento das mensagens recebidas.

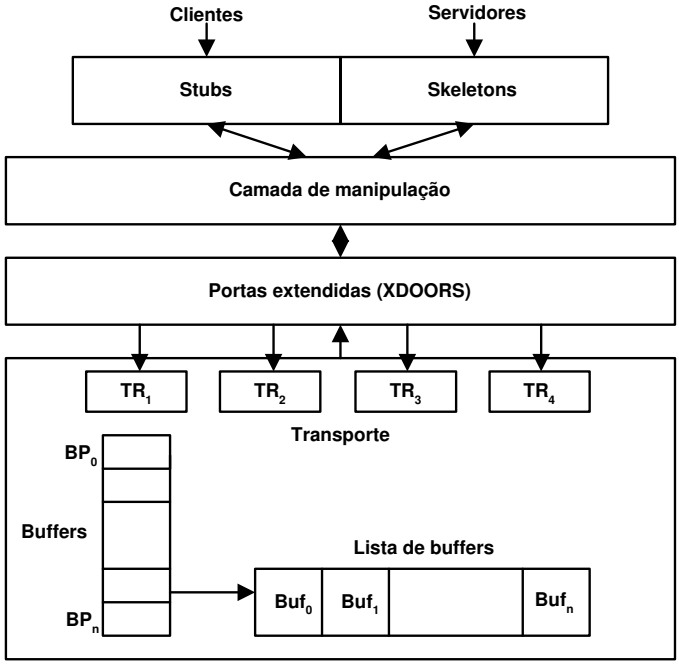

Figura 3.9: ORB do Solaris MC

O ORB do Solaris MC permite tanto comunicação a nível de kernel quanto comunicação a nível de usuário. O ORB é implementado como um módulo carregável para ser usado por código residente no kernel e como uma biblioteca para ser executado por processos a nível de usuário.

3.3.3 Sistema Operacional 2K

O 2K [24] é um sistema operacional distribuído orientado a objetos. Recursos de hardware e software distribuído são encapsulados como objetos CORBA e os serviços do sistema operacional são exportados como serviços CORBA.

O projeto 2K surgiu para suprir a deficiência dos atuais sistemas operacionais distribuídos em relação à adaptabilidade dinâmica em ambientes heterogêneos e à configuração de componentes em aplicações distribuídas.

O sistema operacional 2K combina alguns benefícios do CORBA e dos sistemas operacionais distribuídos para prover gerenciamento de recursos distribuídos, manipular diferentes plataformas de hardware e ser compatível com diferentes sistemas

operacionais instalados nos diversos nós/máquinas que fazem parte do ambiente distribuído.

Ao contrário dos sistemas que carregam todos os módulos na inicialização, o 2K baseia-se na filosofia WYNIWYG (*What You Need Is What You Get*). O sistema é auto-configurável e carrega um conjunto mínimo de recursos necessários para executar as aplicações de usuário de maneira mais eficiente.

O 2K possui um conjunto de API's provendo uma solução integrada para resolver problemas como configuração automática, heterogeneidade e gerenciamento de recursos distribuídos. Aproveita as idéias introduzidas pelo sistema operacional Spring [30], adotando o modelo de comunicação CORBA e os serviços CORBA para gerenciar a dependência inter-componentes.

A figura 3.10 mostra a arquitetura do sistema operacional 2K. O 2K foi projetado em camadas; a camada *middleware* possui dois ORB's, o *dynamicTAO* e o *LegORB*; ambos podem ser dinamicamente configurados para adaptar os recursos disponíveis e para acomodar os requisitos de diferentes aplicações e dispositivos em diferentes momentos.

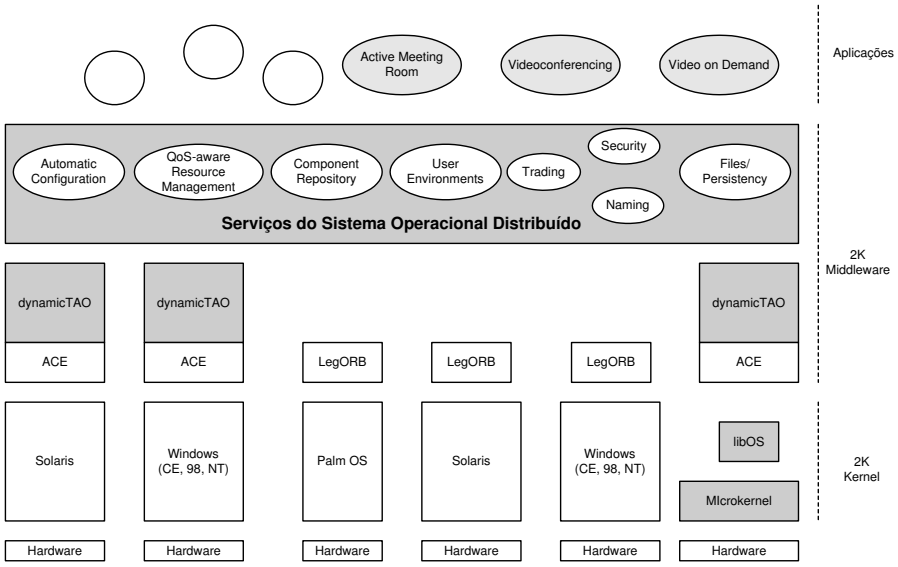

Figura 3.10: Arquitetura do 2K

O sistema operacional 2K é baseado no modelo de objetos CORBA.

Sua parte principal é o *DynamicTAO* [26], um ORB reflexivo de código fonte aberto. A utilização de um ORB reflexivo permite ao 2K a adaptação e alteração de seus componentes em tempo de execução em ambientes heterogêneos e que estão em constante mudanças [12] [25] .

Enquanto o *dynamicTAO* é um ORB projetado para rodar sobre hardware mais robusto como estações de trabalho, o *LegORB* [40] , é um ORB projetado com o mínimo de código sendo apropriado para rodar em sistemas embarcados ou PDA's podendo ter um desempenho melhor que outros ORB's comerciais.

Os recursos em 2K são melhor gerenciados através da utilização de algoritmos de QoS (*Quality of Service*). Os programadores de aplicações têm acesso completo ao estado dinâmico do sistema permitindo implementar aplicações específicas enquanto o sistema garante a qualidade de serviço.

O 2K adota o modelo *network-centric* no qual todas as entidades, usuários, componentes de software e dispositivos existentes na rede são representados como objetos CORBA. Cada entidade possui uma identificação e um perfil.

As interfaces dos objetos do sistema são definidas utilizando OMG IDL; como o sistema possui compatibilidade com CORBA, é possível que clientes CORBA acessem recursos do 2K.

O 2K pode ser executado como um *middleware* no topo de sistemas operacionais tradicionais ou como uma arquitetura integrada com um *microkernel* configurado diretamente sobre o hardware. Usuários que necessitarem de controle extra e melhor desempenho oferecido por um *microkernel*, podem dar partida em suas máquinas com o *microkernel* 2K.

3.3.4 Sistema Operacional Apertos

O sistema operacional Apertos [55] [57] é um projeto de pesquisa do Laboratório de Ciência da Computação da Sony. Apertos é um sistema operacional projetado para grande escala, aberto, distribuído e com suporte a mobilidade. O sistema também é modelado em termos de objetos, de forma que objetos são entidades funda-

mentais no sistema. Todos os recursos no sistema são abstraídos como objetos.

Vários sistemas distribuídos têm sido projetados e implementados. Eles oferecem diversas características modernas para os usuários. Entretanto, eles não são capazes de gerenciar os recursos uniformemente, e não podem sempre satisfazer os requisitos de expansão de vários tipos de usuários.

Uma das metas do Apertos é prover auto-ajuste, característica que permite que o sistema operacional possa ser otimizado conforme as necessidades das aplicações. Para suportar auto-ajuste o Apertos é baseado no paradigma orientado a objetos com uma arquitetura reflexiva, ou seja, a modelagem do sistema em objetos e meta-objetos.

Os objetos representam as aplicações/programas dos usuários. Um meta-objeto pode ser visto como uma máquina virtual que define as computações de um objeto. Os meta-objetos definem os serviços do sistema operacional, um conjunto de meta-objetos é conhecido como meta-espaço. Novos serviços são acrescentados ao sistema operacional pela adição de novos meta-objetos.

Ao contrário de sistemas operacionais baseados em *microkernel*, a estrutura objeto/meta-objeto não divide o sistema em duas camadas. O sistema é construído de uma forma hierárquica.

As computações dos meta-objetos são definidas por um meta-objeto terminal, o meta-meta-objeto ou *metacore*. O *metacore* pode ser visto como um *kernel* em sistemas operacionais convencionais.

Os objetos do nível de objetos podem se comunicar de uma maneira uniforme sem levar em consideração a localização física do objeto destino. Quem determina se a comunicação é local ou remota é o meta-objeto do objeto. No Apertos um objeto é definido independentemente de seu ambiente de execução para facilitar a migração.

O Apertos implementa três primitivas de comunicação inter-objetos baseadas nas semânticas de comunicação síncronas e assíncronas. A figura 3.11 mostra o fluxo de comunicação no Apertos. Na figura, as linhas contínuas indicam o caminho da comunicação. As linhas pontilhadas indicam o caminho de execução da primitiva de comunicação inter-objetos. E as linhas tracejadas indicam o caminho da comunicação executada pelo meta-objeto.

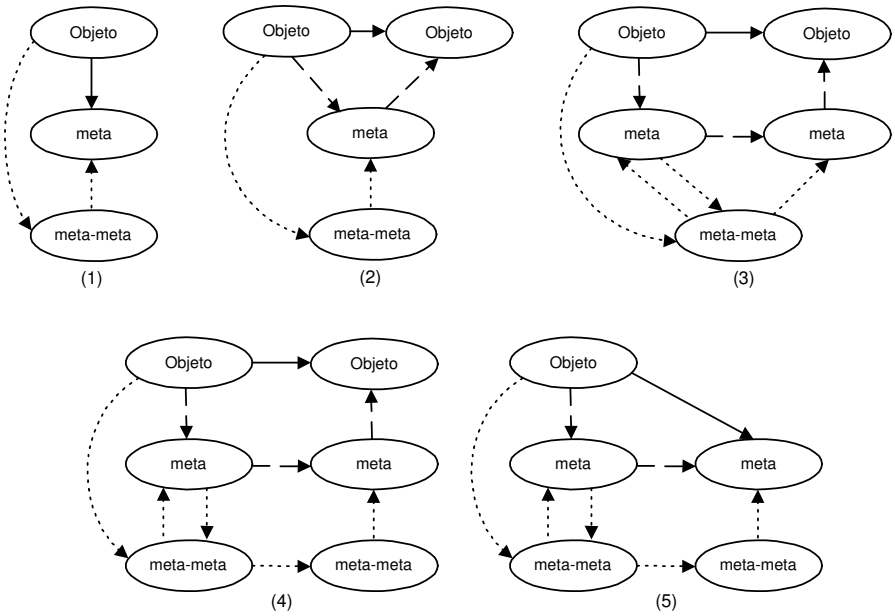

Figura 3.11: Comunicação no sistema Apertos

As primitivas básicas de comunicação implementadas no Apertos permitem utilizar qualquer modelo de comunicação tal como RPC, troca de mensagens ou *streams*. Nos casos em que a comunicação entre objetos for remota, o meta-objeto é responsável por executar os métodos apropriados para efetivar a comunicação, inclusive a escolha do protocolo de rede como TCP/IP ou MuseIP [52].

Quando um objeto migra para outro nó/máquina, o meta-objeto associado com o objeto tem que migrar também, ou deve existir um clone do meta-objeto no nó/máquina destino. Se não, a migração do objeto é proibida pelo meta-objeto. A decisão do que fazer quando um objeto migra é de responsabilidade do meta-objeto.

3.4 Conclusão

Este capítulo abordou os ambientes para computação distribuída, que podem ser caracterizados por plataformas de desenvolvimento de aplicações distribuídas (*middleware*) e sistemas operacionais distribuídos. O capítulo descreve os principais *middleware* existentes e traz estudos de caso de alguns sistemas operacionais distribuídos

dando ênfase ao mecanismo de comunicação utilizado em cada um deles.

Um *middleware* é uma camada de software que atua entre a aplicação e o sistema operacional, escondendo as características e as diferenças entre ambientes heterogêneos. Os *middlewares* podem trabalhar diretamente com processos como o DCE da OSF ou objetos como o CORBA da OMG, o DCOM da Microsoft e o Java/RMI da Sun Microsystems.

O objetivo deste capítulo não era comparar as plataformas de desenvolvimento de aplicações distribuídas, mas descrever suas principais características. Uma análise entre DCOM e CORBA é feita em [10] e [43]. Em [20] é realizado um estudo baseado na performance entre CORBA e JAVA/RMI . Em [41] é feita uma análise comparativa entre DCOM, CORBA e Java/RMI.

O estudo de caso foi feito com quatro sistema operacionais distribuídos abordando principalmente seu modelo de comunicação. O Amoeba utiliza RPC e comunicação em grupo. O Solaris MC é baseado no CORBA. O 2K também é baseado em CORBA porém utiliza reflexão computacional para adaptação dinâmica. O Apertos, também reflexivo, pode tanto fazer uso de troca de mensagens como RPC, quem define como será feita a comunicação são os meta-objetos.

Capítulo 4

Sistema Operacional Aurora

4.1 Introdução

O projeto de sistemas operacionais atualmente está deixando de lado o paradigma de que um sistema operacional é somente um software capaz de gerenciar os recursos computacionais do usuário. Os sistemas operacionais atuais trazem muito mais benefícios para seus usuários do que apenas gerenciar seus recursos de hardware.

Para Yokote [56], um sistema operacional deve prover um alto nível de abstração para seus usuários, pode ser visto como uma máquina virtual e deve ser considerado como um ambiente de programação.

Os benefícios adicionais que os novos projetos de sistemas operacionais trazem são consequência das novas tecnologias empregadas para o desenvolvimento de sistemas de forma a tornar o ambiente mais amigável e flexível ao seu utilizador. A interface gráfica [28], o desenvolvimento OO (orientado a objetos) [31] e a reflexão computacional [4] [55] [58], são elementos desse novo paradigma de desenvolvimento.

O projeto Aurora [58] 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 mesmo recursos do sistema.

A utilização de objetos como entidade fundamental no desenvolvimento

de sistemas traz benefícios como uniformidade de abstração e modularidade. Segundo Chin [9], o modelo de objetos é uma tendência em direção a: *"ao invés do usuário obedecer as necessidades do computador, o computador deve obedecer as necessidades de seus usuários*".

O uso de reflexão computacional é outro recurso importante na construção de sistemas orientado a objetos, pois permite que se tenha uma clara separação entre controle e gerenciamento do sistema e as funcionalidades do sistema.

Este capítulo descreve os aspectos estruturais do sistema operacional Aurora, abordando seu modelo reflexivo orientado a objetos. Para um maior entendimento do modelo estrutural de Aurora, o capítulo também traz uma introdução sobre reflexão computacional no modelo de objetos.

4.2 Reflexão Computacional

Reflexão computacional é a capacidade de um sistema inferir sobre ele mesmo alterando seu comportamento em tempo de execução. Para Patti Maes [27], a reflexão computacional no modelo de orientação a objetos representa a atividade executada por um sistema quando faz computações sobre (e possivelmente afetando) suas próprias computações.

Um sistema reflexivo pode ser definido como um sistema capaz de acessar sua própria descrição e modificá-la de modo a alterar seu próprio comportamento. Este processo ocorre em três estágios diferentes [59]:

1. O primeiro estágio, conhecido como reificação, consiste em obter uma descrição abstrata do sistema tornando-a suficientemente concreta para permitir operações sobre ela.

2. No segundo estágio, a reflexão computacional, utiliza esta descrição concreta para realizar alguma manipulação.

3. Finalmente, no terceiro e último estágio, é modificada a descrição reificada com

os resultados da reflexão computacional, retornando a descrição modificada ao sistema.

Linguagens de programação como CLOS, SMALLTALK e SELF apresentam a habilidade de realizar processamento sobre si mesmas e em particular de estender, em tempo de execução, a própria linguagem.

A reflexão computacional define uma arquitetura em níveis, denominada arquitetura reflexiva, composta por um meta-nível, onde se encontram os meta-objetos que definem as estruturas de dados e as ações a serem realizadas sobre o sistema objeto, localizado no nível base. A figura 4.1 mostra a associação entre o nível-base e o nível-meta [60].

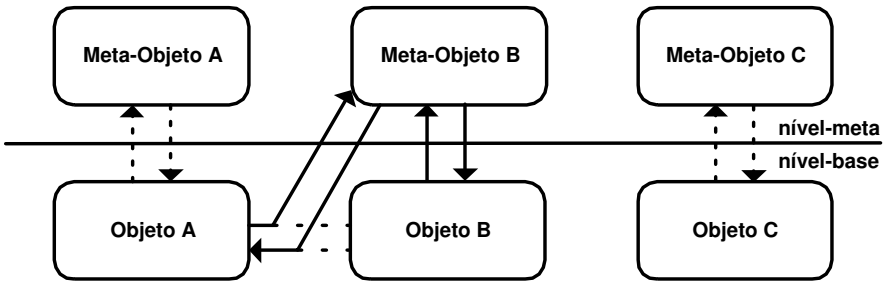

Figura 4.1: Representação da meta-hierarquia

A reflexão computacional pode ocorrer tanto ao nível de classe quanto ao nível de objeto. Quando a reflexão acontece a nível de classe, todas as instâncias desta classe são afetadas; ao contrário, quando a reflexão acontece a nível de objeto, somente esse objeto é afetado. Uma classe ou objeto podem ter tantas meta-classe ou meta-objeto quanto forem necessários.

A comunicação entre os níveis do sistema é feita através de um protocolo chamado MOP (*Metaobject Protocol*). Esse protocolo permite que os vários níveis do sistema se comuniquem, inclusive permitindo a comunicação inter meta-objetos [60]. A comunicaçã é feita através de mensagens. O MOP intercepta essas mensagens de forma implícita, tornando essa operação transparente para a aplicação.

4.2.1 Torre de Reflexão

É possível que um meta-objeto também possua um meta-objeto associado a ele. No caso de existirem contínuas associações de meta-objeto de meta-objeto, tem-se uma torre de reflexão. Essa torre reflexiva pode ser infinita, onde cada item do nível Ni tem um item do nível Ni + 1 associado a ele, e esse item será o meta-objeto do nível Ni - 1.

Devido a restrições impostas pelo hardware e pelo próprio gerenciamento da torre de reflexão, é recomendável que essa não ultrapasse de três níveis, ou seja, o nível base (aplicação), o meta-nível e o meta-meta-nível.

4.2.2 Reflexão Estrutural

A reflexão estrutural permite que sejam alteradas características estruturais da classe ou do objeto. Com a reflexão estrutural é possível alterar, adicionar, remover e modificar métodos ou atributos da classe ou objeto. Também é possível saber quais são as instâncias da classe, qual a sua classe ancestral e quais são as classes descendentes e adicionar, alterar e remover classes.

A reflexão estrutural viola o encapsulamento que é uma forte característica da programação orientada a objeto. Porém com esse recurso é possível desenvolver sistemas mais dinâmicos e que melhor se adaptam a ambientes heterogêneos que estão em constantes mudanças.

4.2.3 Reflexão Comportamental

A reflexão comportamental, ao contrário da reflexão estrutural, não quebra o encapsulamento, pois trata apenas dos aspectos comportamentais do objeto. É possível manter estatísticas de invocação de métodos e utilização de objetos para fins de controle e desempenho do sistema com o uso da reflexão comportamental.

A reflexão comportamental permite conhecer como o objeto reage a invocação de um de seus métodos, ou seja, é possível conhecer as computações realizadas

no objeto em questão, podendo inclusive desviar o fluxo da mensagem para outro objeto.

4.3 Arquitetura do Sistema Operacional Aurora

O sistema operacional Aurora está baseado no modelo de objetos, que no mundo real são naturalmente concorrentes e distribuídos. Desta forma, todo o processamento das informações no ambiente pode ser representado por um conjunto de mensagens fluindo entre objetos executando de forma paralela.

Todos os níveis do sistema Aurora são objetos, desde aplicações de usuário até recursos do sistema operacional. A exploração do paralelismo, tanto a nível do sistema como a nível da aplicação são características do sistema Aurora.

Aurora está baseado no modelo estrutural de Apertos [55], que utiliza o conceito de separação entre objetos e meta-objetos, onde o objeto representa os aspectos funcionais da aplicação, enquanto o meta-objeto define os aspectos não funcionais do objeto. Cada objeto do sistema está associado a um ou mais meta-objeto.

Um conjunto de meta-objetos é chamado de meta-espaço e provê as funcionalidades básicas para o objeto. O meta-espaço pode ser visto como um sistema operacional dedicado à execução do objeto. A figura 4.2 demonstra a associação de um meta-espaço com um objeto.

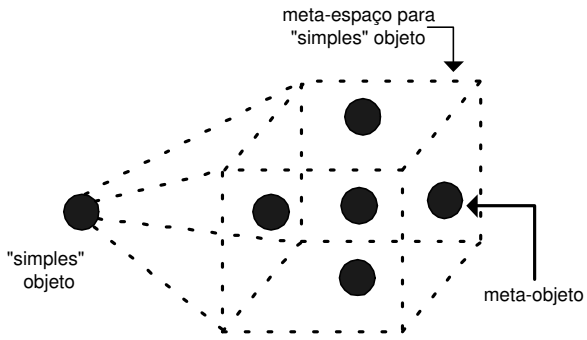

Figura 4.2: Meta-espaço associado a um simples objeto

A migração de objetos em Aurora, ocorre quando um objeto necessitar de serviços oferecidos por outro meta-espaço. Por exemplo, quando um objeto necessi-

tar de serviços de impressão é possível migrar para o meta-espaço que implemente este serviço. Da mesma forma, quando um objeto necessitar ser armazenado em memória secundária é possível migrar para o meta-espaço que implemente o serviço de persistência.

A figura 4.3 demonstra de uma forma simplificada a arquitetura do sistema operacional Aurora. É possível perceber na figura que existem meta-espaços que não possuem objetos associados. Essa particularidade ocorre em meta-espaços que implementam serviços do sistema operacional.

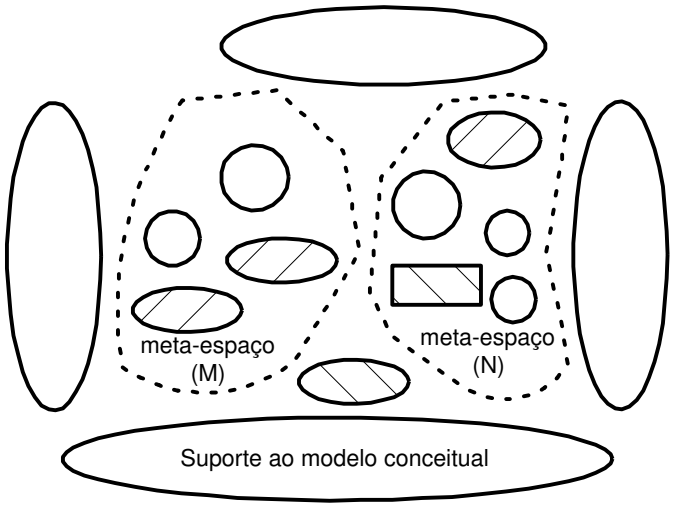

Figura 4.3: Visão Simplificada da arquitetura de Aurora

O sistema operacional Aurora suporta a herança dinâmica [58], o que permite que a estrutura hierárquica das classes se estenda tornando o sistema mais adaptável, ou seja, permite que os objetos do sistema recebam novas funcionalidades em tempo de execução.

A comunicação entre os objetos do nível base e os objetos do nível meta (meta-objetos) é feita através de um meta-objeto terminal chamado de *metacore*, que pode ser levemente associado a um *microkernel*. A figura 4.4 ilustra o *metacore*.

O *metacore* possui duas operações básicas que dão suporte a reflexão computacional. Realizar a meta-computação, isto é, suspender a execução do objeto e transferir o controle para o meta-objeto, ou seja, para o meta-nível, e retornar da meta-computação, transferir o controle da execução do meta-nível para o objeto. As operações

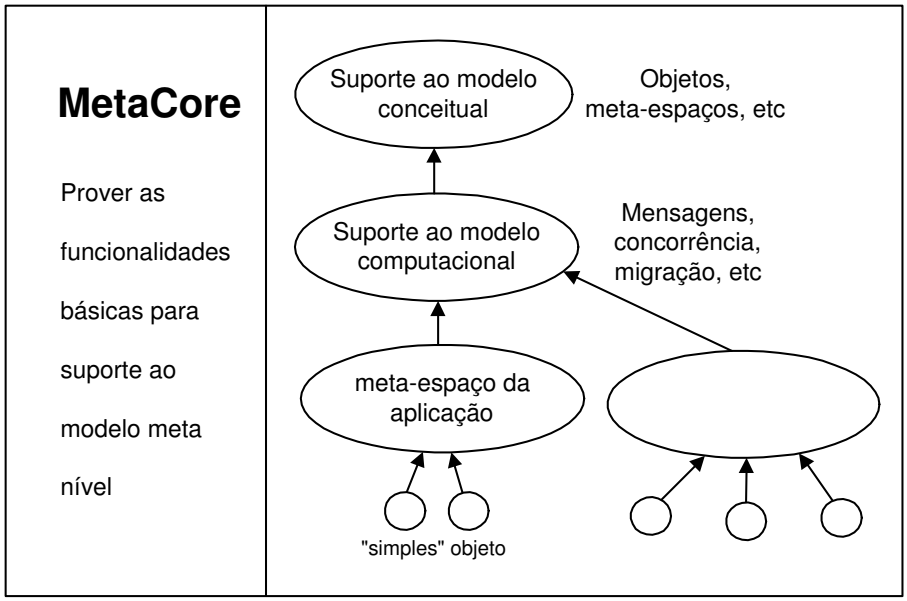

Figura 4.4: Representação do metacore

realizadas pelo metacore podem ser comparadas ao conceito de primitivas em sistemas operacionais tradicionais.

Os meta-espaços dão suporte ao modelo conceitual, no qual o próprio sistema operacional, utilitários e aplicações do usuário são construídos. Os serviços oferecidos pelos meta-espaços basicamente são: migração, multiprocessamento, gerenciamento de meta-espaços e controle de ativações.

4.3.1 Identificação e Localização de Objetos em Aurora

O Aurora possui um mecanismo responsável pela identificação e localização de todos os objetos pertencentes ao sistema. Esse mecanismo, chamado de AVLO (*Access Virtual List Object*) [6] [8], é responsável por identificar, de maneira unívoca, e informar a localização de cada objeto do sistema.

O AVLO faz a concatenação de dois atributos do sistema para gerar a identificação de um objeto [7]. O primeiro atributo, *metacoreID*, diz respeito à identificação que é atribuída ao *kernel* ou *metacore* no momento da instalação do sistema. O outro atributo, *ObjectID*, é gerado pelo próprio AVLO. A identificação do objeto é resultado

da concatenação desses dois atributos, gerando desta forma uma identificação única para cada objeto.

A identificação do *metacore* é feita através de um algoritmo que lê o *Mac Address* da placa de rede e atribui este valor ao atributo *MetaCoreID*. Já o atributo *ObjectID* é um número sequencial controlado pelo AVLO, onde cada nó/máquina controla sua própria variável *ObjectID*.

A localização dos objetos no sistema é feita por meio de duas classes que armazenam as informações dos objetos instanciados localmente e os objetos instanciados remotamente. Estas classes funcionam como um array de instâncias de objetos.

4.3.2 Execução de Objetos em Aurora

Todos os objetos em execução no Aurora, são representados por uma *Activity* que é a abstração de um objeto em execução. Uma *Activity* é similar ao modelo de *thread*, e inclui a abstração da CPU em uma estrutura identificada como *context*.

A *Activity* é uma classe que representa o estado de execução de um objeto. Todos os objetos instanciados no sistema como, meta-objetos do sistema e/ou da aplicação e objetos da aplicação são associados a uma *Activity*.

Abaixo segue o código da classe *Activity* desenvolvido na linguagem de programação C++.

```
Class Activity {
protected:
  CPUContext  Context;
  AID*        Identify;
  pActivity   Meta;
  EntryTable* Execqueue;
};
```

A tabela 4.1 descreve o significado de cada atributo da classe *Activity*:

Tabela 4.1: Descrição dos Atributos da Classe Activity

Atributo	**Descrição**
Context	Representa a abstração da CPU
AID	Identificação do objeto associado a Activity
pActivity	Identificação do meta-objeto na meta-hierarquia
EntryTable	Lista de ativações a serem executadas

4.3.3 Primitivas de Meta-Computação

O sistema operacional Aurora possui duas primitivas que dão suporte à comunicação entre os objetos no sistema. As primitivas de meta-computação são responsáveis pela efetivação da comunicação entre dois objetos.

As primitivas de meta-computação estão implementadas no meta-objeto terminal ou *metacore*, o núcleo do sistema operacional Aurora. O *metacore* suporta a meta-computação através das primitivas M e R.

Quando um objeto ”a” deseja comunicar-se com um objeto ”b”, o objeto ”a” executa uma chamada ao *metacore* através da primitiva M; este por sua vez transfere o controle de execução para um meta-espaço no meta-nível. A primitiva R faz exatamente o inverso que a primitiva M, fazendo com que o controle retorne ao objeto. A figura 4.5 demonstra o uso das primitivas M e R em uma comunicação local.

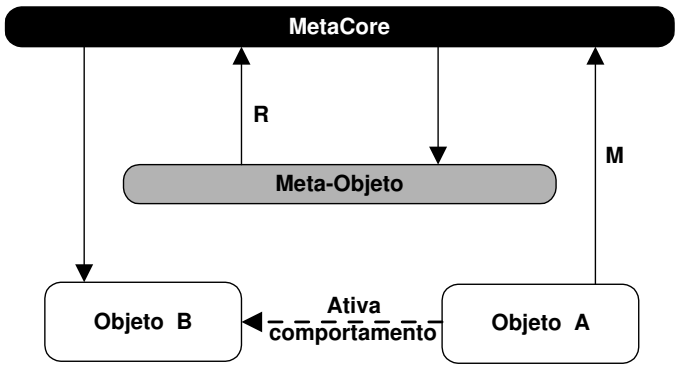

Figura 4.5: Primitivas M e R - Comunicação Local

A função da primitiva M é transferir o controle de execução de um objeto para o meta-objeto. A primitiva R tem a função de realizar o processamento inverso

ao da primitiva M, isto é, retornar o controle de execução do meta-objeto para o objeto, efetivando a comunicação entre os objetos.

4.4 Conclusão

Este capítulo apresentou os principais conceitos envolvidos no projeto do sistema operacional Aurora. O Aurora é um sistema operacional orientado a objetos para arquiteturas multiprocessadas. A utilização de objetos em ambientes multiprocessados permite um maior aproveitamento deste, visto que objetos no mundo real são naturalmente concorrentes e distribuídos.

O 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 mesmo recursos do sistema.

A estrutura em meta-níveis permite ter uma separação em objetos do nível base e meta-objetos (nível meta), sendo que estes são responsáveis pelo controle dos objetos do nível base. O uso de reflexão computacional em sistemas operacionais é um recurso bastante poderoso, pois possibilita ter um maior controle sobre o sistema e permite alteração de características em tempo de execução, diferentemente de outros sistemas onde tal característica não é possível

O sistema também suporta a migração de objetos que pode ocorrer tanto a nível local, ou seja, o objeto migra de meta-espaço, quanto entre os nós que fazem parte do sistema. A migração pode ocorrer quando o objeto necessitar de serviços não disponíveis em seu meta-espaço ou com finalidades de balanceamento de carga.

Capítulo 5

Comunicação Remota de Objetos no Sistema Operacional Aurora

5.1 Introdução

O sistema operacional Aurora utiliza o modelo de objetos como entidade fundamental. Objetos são as únicas entidades presentes em todos os níveis do ambiente, modelando desde aplicações do usuário até serviços do sistema operacional. A utilização de reflexão computacional permite a separação desses objetos em objetos de aplicação (nível base) e meta-objetos (nível meta).

O suporte à comunicação entre objetos no Aurora, é compatível com o seu modelo reflexivo. A comunicação entre objetos em Aurora é realizada através de duas primitivas implementadas no *metacore*, o núcleo do sistema. As primitivas M e R são responsáveis por realizar a transferência do controle de execução de um objeto para o meta nível (meta-objeto), e retornar o controle do meta-objeto para o nível base (objeto) respectivamente.

Atualmente, as primitivas de comunicação M e R só permitem a comunicação local de objetos. O projeto Aurora propõe uma extensão das primitivas M e R para suportar a comunicação remota de objetos. Todavia, a comunicação entre objetos remotos não está implementada devido à restrições impostas pela própria arquitetura do

sistema.

Este capítulo descreve a solução para comunicação entre objetos remotos no sistema operacional Aurora. O capítulo está organizado como segue. Na seção 5.2 é descrito o modelo de comunicação entre objetos em Aurora. A seção 5.3 apresenta a descrição do problema relacionado à comunicação remota de objetos em Aurora. A seção 5.4 descreve a solução desenvolvida. A seção 5.5 apresenta os resultados obtidos através da simulação do mecanismo de comunicação.

5.2 Comunicação entre Objetos em Aurora

O sistema operacional Aurora implementa duas primitivas para suportar a comunicação entre objetos. As primitivas, M e R, estão implementadas no *metacore*, o núcleo do sistema operacional Aurora. Atualmente, as primitivas de comunicação M e R suportam somente a comunicação local de objetos [37].

No projeto Aurora é proposta uma extensão das primitivas M e R para suportar a comunicação entre objetos localizados em nós/máquinas diferentes [38]. O objetivo da primitiva de comunicação remota, chamada de primitiva multinodo, é permitir que objetos em máquinas diferentes possam se comunicar como se essa comunicação fosse local. A figura 5.1 demonstra o uso da primitiva multinodo.

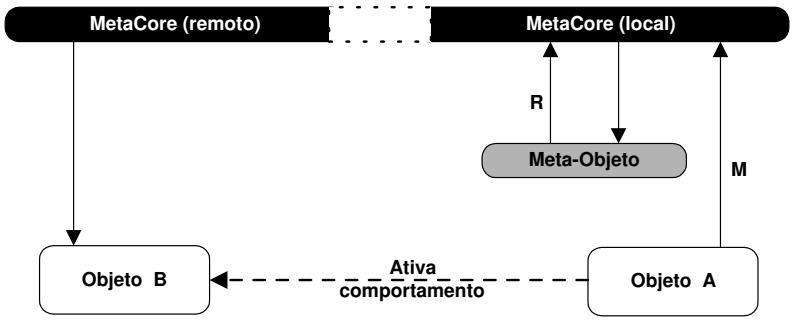

Figura 5.1: Primitiva multinodo - Comunicação Remota

Quando um objeto deseja comunicar-se com outro objeto em uma máquina diferente, esta comunicação é feita através da primitiva multinodo. O número de

nós/máquinas envolvidas no processamento da primitiva R é indeterminado, visto que depende de fatores como a topologia da rede utilizada.

5.3 Descrição do Problema

Devido à restrições impostas pela arquitetura do sistema operacional Aurora, a comunicação remota de objetos não está funcional. A seguir segue a descrição em C++ da classe *metacore* que implementa as primitivas M e R.

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

A função da primitiva M é transferir o controle de execução de um objeto para o meta-objeto. As informações sobre o meta-objeto no metanível estão contidas na definição do parâmetro *MetaActivity*. A mensagem a ser enviada, bem como a identificação dos objetos origem e destino, estão contidas em *MessageM*.

A tabela 5.1 descreve o conteúdo do parâmetro *MessageM* da primitiva de comunicação M.

Tabela 5.1: Descrição dos campos do parâmetro MessageM

Tipo	**Nome**	**Descrição**
Activity*	source	Activity do objeto fonte
Activity*	target	Activity do objeto destino
void*	message	mensagem a enviar

A primitiva R tem a função de realizar o processamento inverso ao da primitiva M, isto é, retornar o controle de execução do meta-objeto para o objeto, efe-

tivando a comunicação entre os objetos. O parâmetro *MessageR*, contém informações sobre o objeto que enviou a mensagem e informações a respeito da mensagem enviada.

A tabela 5.2 descreve o conteúdo do parâmetro *MessageR* da primitiva de comunicação R.

Tabela 5.2: Descrição dos campos do parâmetro MessageR

Tipo	**Nome**	**Descrição**
Activity*	source	Activity do objeto fonte
void*	message	mensagem a enviar

Para executar a primitiva M é necessário incluir no parâmetro *MessageM* a *activity* do objeto a ser ativado. Uma *activity* é uma estrutura que representa a abstração de um objeto em execução. Uma das restrições da comunicação remota está diretamente relacionada *activity* do objeto a ser ativado. O problema consiste em como passar a *activity* do objeto ativado para o parâmetro *MessageM* se este está sendo executado em outra máquina.

Outra restrição da comunicação remota é que o Aurora não possui nenhum mecanismo que permita enviar ou receber mensagens entre máquinas. Para efetivar a comunicação entre objetos remotos é necessário implementar um mecanismo que permita o envio de mensagens de uma máquina para outra.

Para estender o mecanismo de comunicação local para para que o mesmo suporte comunicação remota de objetos, é necessário solucionar as seguintes questões:

1. Propor e implementar uma maneira de passar a *activity* do objeto destino para o parâmetro *MessageM* da primitiva de comunicação M.

2. Propor e implementar um suporte para comunicação entre máquinas para que os parâmetros da primitiva de comunicação R sejam passados para a máquina na qual o objeto destino está sendo executando.

5.4 Solução Desenvolvida

Com base nas questões apresentados na seção anterior, esta seção descreve como foi solucionado o problema de comunicação remota entre objetos no sistema operacional Aurora.

5.4.1 Repositório de Activity

Para solucionar a questão de como passar a *activity* do objeto destino para o parâmetro *MessageM* da primitiva de comunicação M, foi implementada uma classe de gerência de *activity* de objetos remotos. O objetivo dessa classe é criar um lista (repositório) para cada máquina, contendo as *activity* de todos os objetos que são remotamente acessíveis pelos objetos existentes nessa máquina.

O repositório de *activity* é similar ao conceito de repositório de interface adotado no CORBA. Entretanto o repositório de *activity* armazena a estrutura completa de uma *activity* de um objeto, e não a descrição da interface dos métodos remotos como o repositório de interface do CORBA. A figura 5.2 ilustra a arquitetura do sistema de comunicação remota.

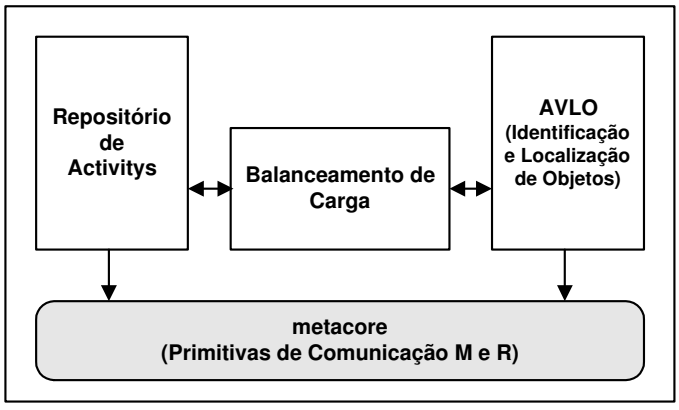

Figura 5.2: Arquitetura do Sistema de Comunicação Remota

A inserção de um objeto no repositório de *activity* é realizada pelo sistema de balanceamento de carga [3]. Quando for solicitada a criação de um objeto por

uma determinada máquina, o balanceador de carga irá determinar se o objeto será criado localmente, na mesma máquina que solicitou a criação, ou remotamente, em outra máquina pertencente ao ambiente distribuído.

O sistema de balanceamento de carga também interage com o mecanismo de identificação e localização de objetos, AVLO (*Access Virtual List Object*) [6]. O AVLO controla a localização dos objetos através de duas listas estruturadas em árvore, uma para controlar os objetos criados localmente e outra para controlar os objetos criados remotamente. As listas do AVLO guardam as seguintes informações:

- **Objeto ID**: Identificação do objeto atribuída a ele no momento de sua criação.
- **Origem**: Identificação da máquina que solicitou a criação do objeto.
- **Destino**: Identificação da máquina no qual o objeto foi criado fisicamente.

Quando for solicitada a criação de um objeto por uma determinada máquina, o sistema de balanceamento de carga determinará o local/máquina que o objeto será criado fisicamente. Logo após ser determinado o local para a criação do objeto solicitado, o balanceador de carga atualizará as listas do AVLO.

A tabela 5.3 lista um conjunto de máquinas solicitando a criação de determinados objetos.

Tabela 5.3: Solicitação de criação de objetos por máquina

Máquina	Objeto a ser criado (Identificação)
1	1000
1	1010
1	1020
2	2000
2	2010
3	3000

Conforme descrito anteriormente, no momento da criação de um objeto, o balanceador de carga irá determinar o local/máquina no qual o objeto será criado fisicamente. O sistema de balanceamento de carga deverá atualizar as listas do AVLO para que

os objetos possam ser localizados para uma futura ativação. A tabela 5.4 mostra o AVLO de cada máquina após o balanceamento de carga.

Tabela 5.4: Relação de objetos criados por máquina após o balanceamento de carga

Máquina	Objeto (Identificação)	Origem	Destino
1	1000	1	1
1	1010	1	2
1	1020	1	3
1	3000	3	1
2	1010	1	2
2	2000	2	2
2	2010	2	3
3	1020	1	3
3	3000	3	1

Quando o balanceador de carga decidir criar o objeto em uma máquina diferente da qual solicitou a sua criação, além de atualizar as listas do AVLO, ele deverá inserir no repositório de *activity*, da máquina solicitante, a *activity* desse objeto para que ele possa ser ativado remotamente. A tabela 5.5 mostra a lista de *activity* de cada máquina do exemplo anterior.

Tabela 5.5: Lista de Activity por máquina

Máquina	Activity
1	1010
1	1020
2	2010
3	3000

Quando um objeto necessitar ativar um comportamento de um objeto remoto, é verificado se a *activity* desse objeto está registrada no repositório de *activity*. A *activity* do objeto remoto é recuperada do repositório através do método *GetActivity(AID no)*, e passada como parâmetro para a primitiva de comunicação M.

A remoção de uma *activity* do repositório de *activity* é feita quando não existir mais nenhuma referência de um objeto local para o objeto remoto. Uma segunda possibilidade para a remoção de uma *activity* do repositório, é quando o objeto remoto

deixar de existir. Nesse caso, ao tentar ativar um comportamento desse objeto o sistema irá reportar um erro.

5.4.2 Comunicação entre Máquinas

Para suportar a comunicação entre máquinas foram acrescentadas ao *metacore* duas primitivas de comunicação entre nós/máquinas. Baseado no mecanismo de comunicação por troca de mensagens foram acrescentadas ao *metacore* as primitivas *send()* e *receive()*, que servem para enviar e receber mensagens respectivamente.

A seguir é listada a classe *metacore* desenvolvida em C++ com as primitivas *send()* e *receive()*.

```
class MetaCore {
  private:

  public:
    MetaCore ();
    void  M  ( MetaActivity* mObject, MessageM* pMsg );
    void  R  ( MessageR* pMsg );
    mError Send(NetMessage m, NetAddress address);
    mError Receive(NetMessage m);
};
```

A primitiva de comunicação *Send()* envia uma mensagem de uma máquina para outra. O parâmetro NetMessage contém informações sobre a origem, o destino e os dados necessários para executar a primitiva R. O parâmetro NetAddress contém o endereço IP da máquina destino, onde o objeto remoto está sendo executado.

A primitiva de comunicação *Receive()* é utilizada para receber mensagens que são enviadas através da primitiva *Send()*. O parâmetro NetMessage contém informações sobre a origem, o destino e dados necessários para executar a primitiva R.

As primitivas *Send()* e *Receive()* podem retornar mensagens de erro caso

ocorra algum problema com a comunicação. A tabela 5.6 lista os valores de retorno válidos para as primitivas Send() e Receive().

Tabela 5.6: Mensagens de erro das primitivas Send e Receive

Tipo	Descrição
mSUCCESS	OK - Operação concluída com sucesso
mOBJNOTFND	Objeto destino não encontrado
mDSTNOTRCH	Máquina/host destino não localizada
mPROTOERR	Erro de protocolo
mPARERR	Parâmetro inválido para complementar a execução de R

As primitivas de comunicação *Send()* e *Receive()* implementadas em Aurora são bloqueantes. Um objeto ao executar *Send()* ou *Receive()* ficará bloqueado até que a operação solicitada seja concluída.

Quando um objeto ”X” desejar ativar um comportamento em outro objeto, ”Y”, será verificado se o objeto ”Y” reside localmente ou não através de uma consulta ao AVLO. Caso seja detectado que o objeto ”Y” é um objeto remoto, a *activity* desse objeto será recuperado do repositório de *activity* e passada como parâmetro para a primitiva de comunicação M.

A figura 5.3 mostra o fluxo de execução para o objeto ”X” ativar um comportamento do objeto ”Y”.

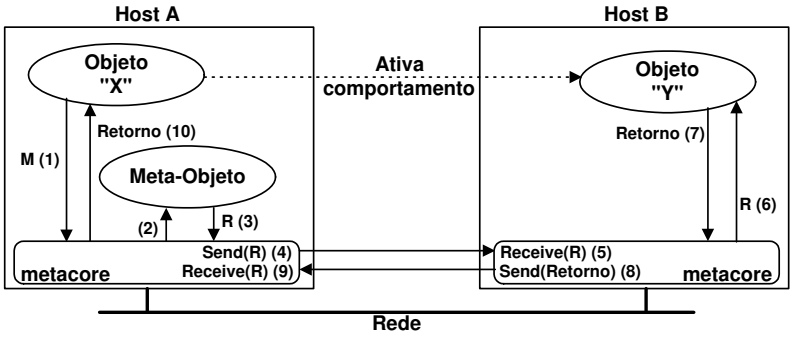

Figura 5.3: Fluxo de Execução de Send() e Receive()

O objeto ”X” executa a primitiva de comunicação M (1), o *metacore* bloqueará a execução do objeto ”X” e passará os parâmetros de M para o meta-objeto no

meta-nível (2). O meta-objeto executará a primitiva de comunicação R (3), será identificado que se trata de uma comunicação remota. Uma mensagem com o conteúdo de R, a identificação do objeto origem e a identificação do objeto destino e o endereço IP da máquina destino serão passados como parâmetro para a primitiva *Send()* (4).

A máquina remota irá receber a mensagem através da primitiva *receive()* (5) e dará inicio ao processamento da primitiva R (6). A fila de ativações do objeto remoto será atualizada e o método ativado será executado. Após a execução do método, o objeto ”Y” devolverá uma mensagem para o objeto ”X” contendo o resultado da ativação (7). Uma mensagem será enviada para a máquina origem contendo o resultado da ativação (8). Ao receber a mensagem com o resultado da ativação (9) o objeto ”X” será liberado para ser executado.

5.5 Validação do Sistema de Comunicação do Aurora

Para validar a solução proposta para comunicação remota entre objetos no sistema operacional Aurora foi desenvolvido um simulador. O objetivo deste programa é simular o ambiente computacional do Aurora, permitindo que objetos possam ativar comportamentos em outros objetos remotamente.

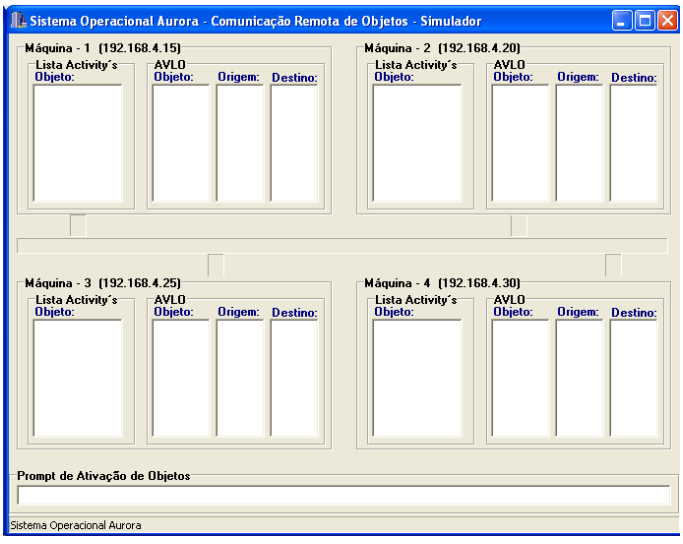

Figura 5.4: Tela principal do simulador do sistema de comunicação de objetos do Aurora

A figura 5.4 mostra a tela inicial do simulador de comunicação de objetos do Aurora logo após ser posto em execução. Como não foi solicitado a criação de nenhum objeto no sistema, o AVLO e o repositório de *activity* estão vazios.

No *prompt* de ativação de objetos será digitado o comando para realizar a ativação de um comportamento em um objeto qualquer. A sintaxe do comando de ativação é: maquina@XX ObjOriXX ObjDesXX, onde maquina@XX deverá ser informado a identificação da máquina onde reside o objeto origem. ObjOriXX, é a identificação do objeto origem. ObjDesXX é a identicação do objeto destino, que será ativado.

O simulador é composto de um conjunto de quatro máquinas interligados em rede. Cada máquina possui a representação do AVLO, responsável por identificar e controlar a localização dos objetos no ambiente. Também é representado a lista (repositório) de *activity* que manterá a *activity* de todos os objetos solicitados para serem criados localmente, entretanto criados remotamente, em outra máquina, pelo balanceador de carga.

Para exemplificar a criação de alguns objetos no ambiente, a tabela 5.7 lista quatro máquinas com seus respectivos pedidos para criação de objetos.

Tabela 5.7: Exemplo de solicitação de criação de objetos - simulador

Máquina	**Solicitação para criação de objeto (Identificação)**
1	10010
1	10020
1	10030
1	10040
2	20010
3	30010
3	30020
3	30030
4	40010

O simulador possui um menu de opções que é ativado ao pressionar o botão direito do *mouse* sobre a tela principal. O menu possui cinco opções que são descritas a seguir:

1. **Carregar Objetos**: carrega todos os objetos solicitados para criação no ambiente;

2. **Descrição do Objetos**: faz a reificação dos objetos, ou seja, abre uma janela contendo a descrição de cada objeto, seus métodos com respectivos parâmetros e valor de retorno;

3. **Log de Ativações**: mostra a log de ativação de objetos. A log é gerada toda vez que um objeto for ativado pelo *prompt* de ativação de objetos do simulador;

4. **Recarregar Objetos**: apaga as listas do AVLO e o repositório de *activity*, e em seguida recarrega todos os objetos no ambiente;

5. **Finalizar Simulador**: finaliza a execução do simulador.

A figura 5.5 mostra a tela com o menu de opções do simulador.

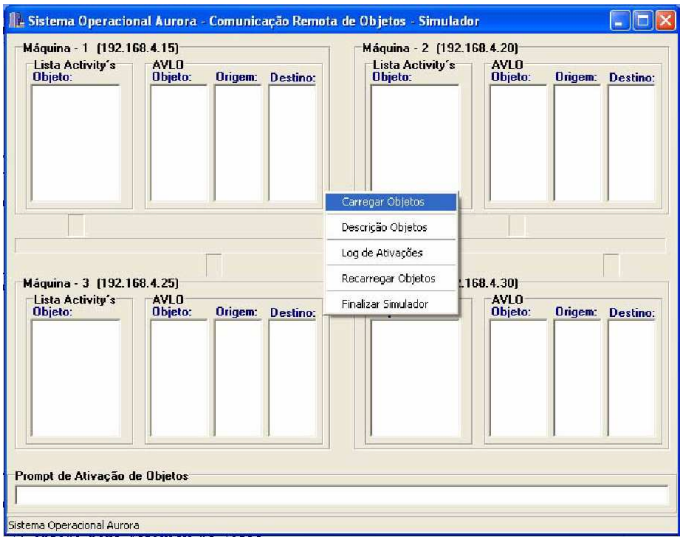

Figura 5.5: Tela com o menu de opções do simulador

Quando for solicitada a criação dos objetos, o sistema de balanceamento de carga irá determinar o local para a criação do objeto fisicamente, de acordo com as políticas de balanceamento de carga. Após ser concluído o balanceamento de carga e a atualização do AVLO e do repositório de *activity* em cada máquina, o ambiente estará pronto para ser utilizado, ou seja, os objetos poderão interagir entre eles através das primitivas de comunicação M e R presentes no *metacore*.

A figura 5.6 mostra o repositório de *activity* e o AVLO de cada máquina, após o processo de criação dos objetos e o balanceamento de carga ter sido concluído.

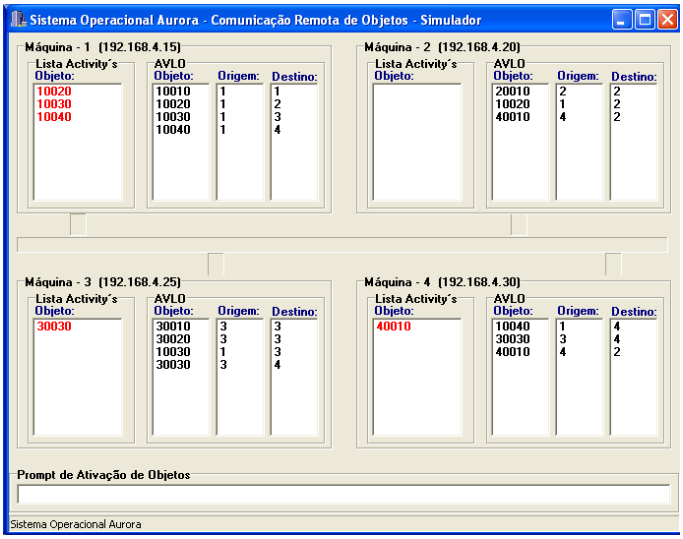

Figura 5.6: Tela do simulador após o balanceamento de carga

É importante observar que podem existir máquinas em que o repositório de *activity* está vazio. Isso ocorre porque o balanceador de carga optou por criar todos os objetos solicitados por essa máquina localmente.

A figura 5.7 mostra a tela com a descrição de todos os objetos criados no simulador.

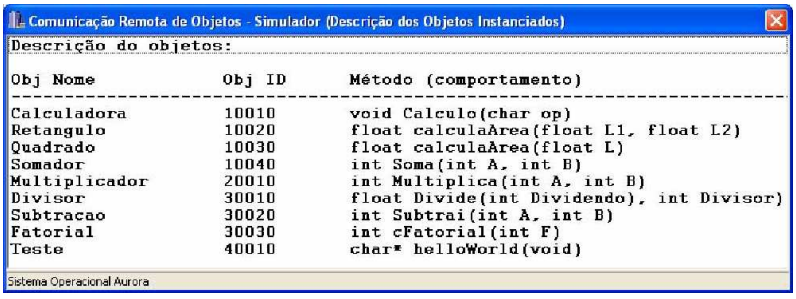

Figura 5.7: Tela com a descrição dos objetos

Para simular uma ativação de um comportamento de um objeto, basta informar no *prompt* de ativação de objetos, o objeto origem o o objeto destino. Por exemplo, para o objeto 10010 (Calculadora) ativar o comportamento calculaArea() do objeto

10020 (Retângulo), basta digitar no *prompt* o comando maquina@1 10010 10020, em seguida pressionar *enter*.

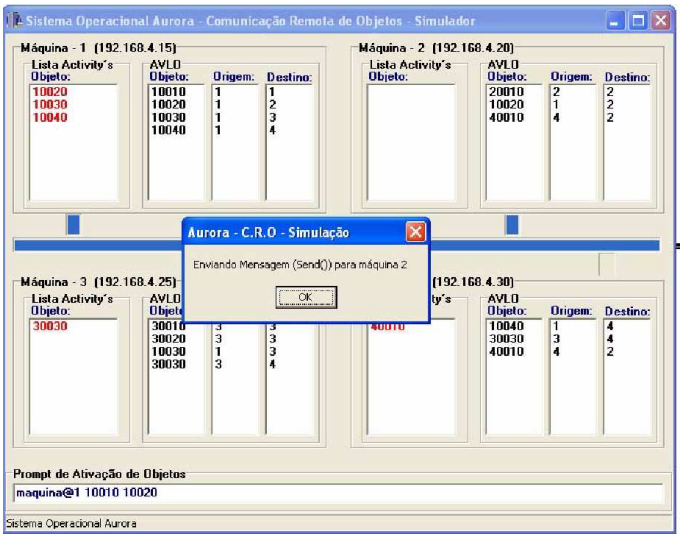

Figura 5.8: Tela exemplo de ativação de objetos

A figura 5.8 mostra a ativação do comportamento calculaArea() do objeto 10020 pelo objeto 10010. O sistema reconheceu que se tratava de uma comunicação remota, pois o objeto 10020 está executando na máquina 2, por isso aparece na figura uma mensagem informando que a primitiva de comunicação *Send()* foi executada.

O objeto destino (10020) irá receber a mensagem enviada pela máquina origem, através da primitiva de comunicação *receive()*. Logo após receber a mensagem, o objeto 10020 irá executar o método ativado e devolverá o resultado para o objeto origem que está bloqueado aguardando o retorno da ativação.

O objeto origem (10010) irá receber a mensagem com o resultado da ativação do método remoto e será posto em execução. O objeto 10010 irá mostrar na tela, através de uma mensagem, o resultado da ativação. A figura 5.9 mostra a tela com a mensagem mostrando o resultado da ativação.

Para facilitar a simulação, os valores para as ativações dos métodos dos objetos remotos foram pré-definidos no simulador. No caso da ativação do método calculaArea() do objeto 10020 foi definido os valores 15 e 20 para seus parâmetros.

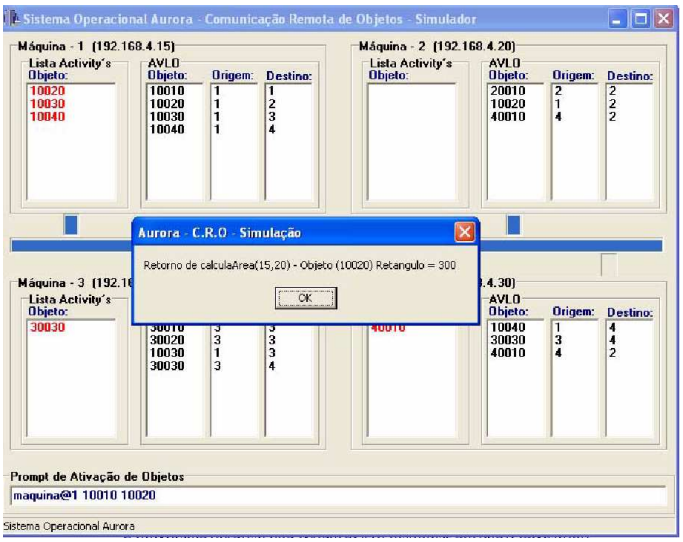

Figura 5.9: Tela com o resultado da ativação do método remoto

Também é possível simular falhas de máquinas. Para isso basta digitar no *prompt* de ativação de objetos o comando falha@XX, onde XX é a identificação da máquina na qual está sendo simulada a falha. Quando é detectado que uma máquina falhou, as listas do AVLO são liberadas, ou seja, todas as referências a objetos são perdidas, e o repositório de *activity* também é perdido.

Para fazer com que uma máquina volte a estar ativa novamente é necessário digitar o comando volta@XX no *prompt* de ativação de objetos, substituindo XX pela identificação da máquina. Quando uma máquina volta, não são recuperadas as listas do AVLO e nem o repositório de *activity*.

Para validar o controle de erros do sistema de comunicação remota de objetos, o objeto 10010 (Calculadora) ativa o comportamento Soma() do objeto 10040 (Somador) residente na máquina 4, entretanto a máquina 4 não está ativa. O sistema de comunicação detecta problemas em que a máquina ou o objeto destino não foram localizados, todavia o tratamento desse tipo de erro é de responsabilidade do sistema de tolerância a falhas.

A figura 5.10 mostra a tela com uma mensagem de advertência informando ao objeto origem 10010 que a máquina destino (4) não está ativa.

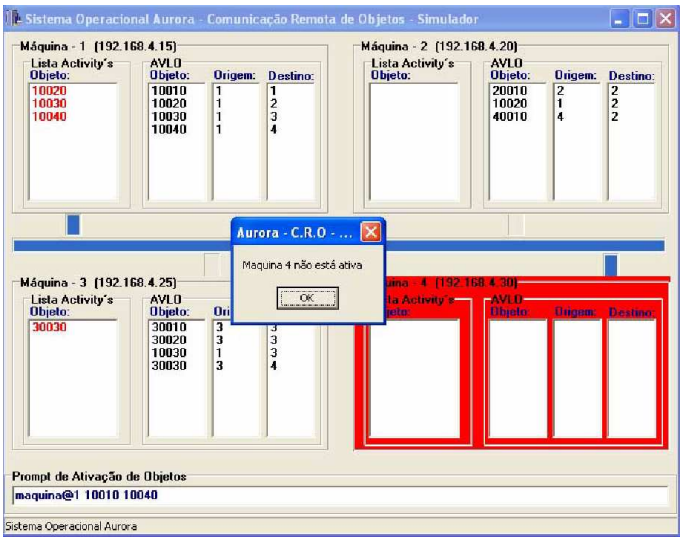

Figura 5.10: Tela com mensagem de advertência

Também é possível simular problemas em objetos individuais, como um objeto que não existe mais no sistema, mais que ainda existem referência de outras máquinas para ele.

5.6 Conclusão

Este capítulo descreveu a solução para a comunicação remota de objetos no sistema operacional Aurora. A solução desenvolvida é baseada no sistema de comunicação local de objetos já existente em Aurora.

A solução para comunicação remota de objetos estende o sistema de comunicação local para que o mesmo suporte a comunicação remota de objetos. Toda a comunicação entre objetos local é realizada pelas primitivas de comunicação M e R implementadas no *metacore* o núcleo do sistema operacional Aurora.

Para estender o mecanismo de comunicação local para suportar a comunicação remota de objetos, foi necessário a criação de um repositório de *activity*, que armazena em cada máquina a *activity* de um objeto solicitado para ser criado localmente, mas que por motivos de balanceamento de carga, foi criado remotamente. Também foi acrescentado ao *metacore* as primitivas de comunicação *Send()* e *Receive()* que enviam e

recebem mensagens via rede respectivamente.

O mecanismo de comunicação remota proposto não é tolerante a falhas, essa tarefa é de responsabilidade do sistema de tolerância a falhas do Aurora. Entretanto, a solução proposta dá suporte para a tolerância a falhas em objetos. Se um determinado objeto deixar de existir ou a máquina que ele estava executando falhar, basta localizar a *activity* desse objeto via repositório de *activity* em cada máquina e inseri-la na fila de aptos do escalonador de objetos, em seguida atualizar as listas de controle de objetos do AVLO.

Capítulo 6

Considerações Finais

Este trabalho apresentou uma solução para a comunicação remota de objetos no sistema operacional Aurora. O Aurora é um sistema operacional reflexivo projetado para arquiteturas multiprocessadas.

Toda a comunicação entre objetos local em Aurora é realizada pelas primitivas de comunicação M e R, implementadas no *metacore*, o núcleo do sistema operacional Aurora. Para a comunicação entre objetos distribuídos é proposto e implementado uma solução que estende o modelo de comunicação local para que o mesmo suporte a comunicação entre objetos distribuídos.

Cada objeto em execução em Aurora é representado por uma *activity*. Uma *activity* é uma estrutura que contém informações sobre os objetos, é similar ao conceito de *thread* em outros sistemas operacionais. A *activity* de um objeto só existe na máquina em que ele está sendo executado.

Quando um objeto deseja ativar um comportamento de um outro objeto que esteja sendo executado em outra máquina, é necessário incluir nos parâmetros da primitiva de comunicação M, a *activity* desse objeto remoto. Essa restrição é imposta pela arquitetura do sistema operacional Aurora.

Para suportar a comunicação entre objetos distribuídos em Aurora, foi criado um repositório de *activity*, similar ao conceito de repositório de interface do CORBA. Esse repositório de *activity* é criado e mantido em todas as máquinas que fazem parte do

ambiente distribuído. Quando é solicitada a criação de um objeto, o balanceador de carga determinará o local em que será criado fisicamente esse objeto. Quanto o balanceador de carga decidir criar o objeto em outra máquina que não a máquina solicitante, é necessário que ele inclua a *activity* desse objeto no repositório de *activity* da máquina que solicitou a criação desse objeto. Dessa forma, qualquer comunicação de um objeto local com um objeto remoto, fica assegurada pelo repositório de *activity*.

Para estender o mecanismo de comunicação local para suportar comunicação remota de objetos, além do repositório de *activity*, foi necessário criar as primitivas de comunicação *send()* e *receive()*, responsáveis por enviar e receber mensagens respectivamente entre máquinas. A criação dessas primitivas foi necessária pelo fato de o sistema operacional Aurora não possuir nenhum mecanismo de comunicação entre máquinas.

A solução desenvolvida para a comunicação entre objetos distribuídos no sistema operacional Aurora é totalmente transparente do ponto de vista do usuário. Houve a preocupação em compatibilizar a comunicação remota com a comunicação local de objetos já existente no sistema, de maneira a evitar formas de comunicação diferenciadas de acordo com a localização dos objetos.

6.1 Trabalhos Futuros

Esta seção apresenta uma lista de sugestões para trabalhos futuros que possam contribuir e/ou melhorar a solução proposta e desenvolvida neste trabalho:

- Desenvolver métodos para tornar a comunicação remota entre objetos tolerante a falhas;

- Tornar o mecanismo de comunicação de objetos compatível com o modelo de comunicação em grupo, maximizando o processamento distribuído;

- Integrar o repositório de *activity* ao AVLO, sistema responsável pela identificação e localização de objetos;

- Desenvolver um mecanismo de comunicação baseado na interface do objeto e não na *activity*, como o mecanismo de comunicação RPC;
- Implementar um protocolo de comunicação próprio para facilitar a troca de mensagens entre máquinas.

Anexo 1

Contém o código fonte de todas as classes utilizadas e desenvolvidas no mecanismo de comunicação remota de objetos no sistema operacional Aurora.

```
//-----------------------------------------------------
// Sistema Operacional AURORA - Copyright(c) INE-UFSC
// Projeto: Comunicacao Remota de Objetos
// Author  Mo. Anderson Luiz Fernandes Perez
// Advisor Dr. Luiz Carlos Zancanella
// Arquivo: Metacore.h
// Descricao: Definicao da estrutura do MetaCore
//-----------------------------------------------------

#ifndef MetaCore_h_DEFINED
#define MetaCore_h_DEFINED
//-----------------------------------------------------

#include "types.h"
#include "activity.h"
#include "GlobalError.h"
#include "Structures.h"
//-----------------------------------------------------

class MetaCore {
```

```
  public:
    MetaCore ();
    void  M  ( MetaActivity* mObject, MessageM* pMsg );
    void  R  ( MessageR* pMsg );
    mError Send (NetMessage m, NetAddress address);
    mError Receive (NetMessage m);
};
//-------------------------------------------------------
#endif MetaCore_h_DEFINED
```

```
//-----------------------------------------------------
// Sistema Operacional AURORA - Copyright(c) INE-UFSC
// Projeto: Comunicacao Remota de Objetos
// Author  Mo. Anderson Luiz Fernandes Perez
// Advisor Dr. Luiz Carlos Zancanella
// Arquivo: Structures.h
// Descricao: Definicao das estruturas utilizadas no sistema
//-----------------------------------------------------

#ifndef Structures_h_DEFINED #define Structures_h_DEFINED
//-----------------------------------------------------

#include  "Types.h"
//-----------------------------------------------------
// Estrutura das mensagens para as primitivas de MetaCore

struct Message {
  AID object;
  Entry method;
  void* message;
};

struct MessageM {
  pActivity source;
  pActivity target;
  Message* message;
};

struct MessageR {
  pActivity source;
```

```
  Message* message;
};

struct MetaActivity {
  pActivity meta;   // Activity do metaobjeto no metanivel
};

struct NetMessage {
  AID ObjOrigem;
  AID ObjDestino;
  MessageR messageR;
};

//---------------------------------------------------------
#endif   // Structures_h_DEFINED
```

```
//-----------------------------------------------------
// Sistema Operacional AURORA - Copyright(c) INE-UFSC
// Projeto: Comunicacao Remota de Objetos
// Author  Mo. Anderson Luiz Fernandes Perez
// Advisor Dr. Luiz Carlos Zancanella
// Arquivo: ListaActivity.h
// Descricao: Definicao da estrutura do rep. de Activity
//-----------------------------------------------------

#include "Activity.h"
//-----------------------------------------------------

class ListaActivity {
  private:
    Activity* inicio;
  public:
    ListaActivity();
    void SetInicio(Activity* act);
    Activity* GetInicio();
    void Add(Activity* act);
    void Remove(AID no);
    bool Pesquisa(AID no);
    Activity* GetActivity(AID no);
};
//-----------------------------------------------------
```

```
//-------------------------------------------------------
// Sistema Operacional AURORA - Copyright(c) INE-UFSC
// Projeto: Comunicacao Remota de Objetos
// Author  Mo. Anderson Luiz Fernandes Perez
// Advisor Dr. Luiz Carlos Zancanella
// Arquivo: ListaActivity.cpp
// Descricao: Impl. do rep. de Activity
//-------------------------------------------------------

#include "ListaActivity.h"
//-------------------------------------------------------

ListaActivity::ListaActivity() {
  inicio = new Activity();
}
//-------------------------------------------------------

void ListaActivity::SetInicio(Activity* act) {
  inicio = act;
}
//-------------------------------------------------------

Activity* ListaActivity::GetInicio() {
  return inicio;
}
//-------------------------------------------------------

void ListaActivity::Add(Activity* act) {
  if (!Pesquisa(act->GetIdentify())) {
    if (inicio->GetIdentify() > 0) {
```

```
      act->SetNext(inicio->GetNext());
      inicio->SetNext(act);
    }
    else {
      SetInicio(act);
    }

  }
}
//---------------------------------------------------------

bool ListaActivity::Pesquisa(AID no) {
  Activity* temp = inicio;
  while(temp)
  {
    if (temp->GetIdentify() == no)
      return true;
    temp = temp->GetNext();
  }
  return false;
}
//---------------------------------------------------------

void ListaActivity::Remove(AID no) {
  Activity* tempant = inicio;
  Activity* temp = inicio->GetNext();

  if (tempant->GetIdentify() == no) {
    inicio = temp;
    delete tempant;
```

```
  }
  else {
    while(temp) {
      if (temp->GetIdentify() == no) {
        tempant->SetNext(temp->GetNext());
        delete temp;
      }
      tempant = temp;
      temp = temp->GetNext();
    }
  }
}
//---------------------------------------------------------

Activity* ListaActivity::GetActivity(AID no) {
  Activity* temp = inicio;
  while(temp)
  {
    if (temp->GetIdentify() == no)
      return temp;
    temp = temp->GetNext();
  }
}
//---------------------------------------------------------
```

```
//-----------------------------------------------------
// Sistema Operacional AURORA - Copyright(c) INE-UFSC
// Projeto: Comunicacao Remota de Objetos
// Author  Mo. Anderson Luiz Fernandes Perez
// Advisor Dr. Luiz Carlos Zancanella
// Arquivo: Activity.h
// Descricao: Definicao da estrutura da Activity
//-----------------------------------------------------

// Arquivo: Activity.h
//-----------------------------------------------------

#ifndef Activity_h_DEFINED
#define Activity_h_DEFINED
//-----------------------------------------------------
#include "Types.h"
#include "Structures.h"
//-----------------------------------------------------

enum mcState { sSUSPENSO, sDORMINDO };

class Activity {

  protected:

    CPUContext  Context;    // Estado dos registradores da CPU
    AID         Identify;   // Identificao da Activity
    EntryTable  ExecQueue;  // Lista de Entradas (mensagens)
    pActivity   Meta;       // pointer para o metaobjeto
    mcState     State;      // Estado da Activity
```

```
    Activity*   Next;       // Ponteiro para a proxima Activity

  public:

    Activity ( );
    ~Activity ( );
    void SetContext(CPUContext pContext);
    CPUContext GetContext();
    void SetIdentify(AID id);
    AID GetIdentify();
    void SetExecQueue(EntryTable* pExecQueue);
    EntryTable GetExecQueue();
    void SetMeta(pActivity pMeta);
    pActivity GetMeta();
    void SetState(mcState pState);
    mcState GetState();
    void SetNext(Activity* pNext);
    Activity *GetNext();
    void ExecuteM (Message* pMsg);
    void ExecuteR (MessageR* pMsg);
};
//---------------------------------------------------------
#endif   // Activity_h_DEFINED
```

```
//-----------------------------------------------------
// Sistema Operacional AURORA - Copyright(c) INE-UFSC
// Projeto: Comunicacao Remota de Objetos
// Author  Mo. Anderson Luiz Fernandes Perez
// Advisor Dr. Luiz Carlos Zancanella
// Arquivo: Activity.cpp
// Descricao: Impl. da classe Activity
//-----------------------------------------------------

#include "activity.h"
#include <iostream.h>
//-----------------------------------------------------

Activity::Activity( ) {
  Identify = 0;  // vlr. inicial para a Activity
  Next = NULL;
}
//-----------------------------------------------------

Activity::~Activity( ) {

}
//-----------------------------------------------------

void Activity::SetContext(CPUContext pContext) {
  Context = pContext;
}
//-----------------------------------------------------

CPUContext Activity::GetContext() {
```

```
  return (Context);
}
//---------------------------------------------------------

void Activity::SetIdentify(AID id) {
  Identify = id;
}
//---------------------------------------------------------

AID Activity::GetIdentify() {
  return (Identify);
}
//---------------------------------------------------------

void Activity::SetExecQueue(EntryTable* pExecQueue) {
  ExecQueue = pExecQueue;
}
//---------------------------------------------------------

EntryTable Activity::GetExecQueue() {
  return (ExecQueue);
}
//---------------------------------------------------------

void Activity::SetMeta(pActivity pMeta) {
  Meta = pMeta;
}
//---------------------------------------------------------

pActivity Activity::GetMeta() {
```

```
  return (Meta);
}
//---------------------------------------------------------

void Activity::SetState(mcState pState) {
  State = pState;
}
//---------------------------------------------------------

mcState Activity::GetState() {
  return (State);
}
//---------------------------------------------------------

void Activity::SetNext(Activity* pNext) {
  Next = pNext;
}
//---------------------------------------------------------

Activity* Activity::GetNext() {
  return (Next);
}
//---------------------------------------------------------

void Activity::ExecuteM (Message* pMsg) {
  MessageM *ThisMessageM;
  if (Avlo->IsRemote(pMsg->object)) {
    ThisMessageM->target = Lista->GetActivity(pMsg->object);
  }
  else {
```

```
    ThisMessageM->target = Escalonador->GetActivity(pMsg->object);
  }
  ThisMessageM->message = pMsg;
  ThisMessageM->source  = this;
  AuroraMetaCore->M(Meta, ThisMessageM );
}
//-------------------------------------------------------

void Activity::ExecuteR (MessageR* pMsg) {
//  AuroraMetaCore->R(pMsg);
}
//-------------------------------------------------------
```

```
//-----------------------------------------------------
// Sistema Operacional AURORA - Copyright(c) INE-UFSC
// Projeto: Comunicacao Remota de Objetos
// Author  Mo. Anderson Luiz Fernandes Perez
// Advisor Dr. Luiz Carlos Zancanella
// Arquivo: Types.h
// Descricao: Definicao dos tipos utilizados no sistema
//-----------------------------------------------------

#ifndef _Types_h_DEFINED
#define _Types_h_DEFINED
//-----------------------------------------------------

typedef unsigned char    byte;      // 8-bits sem sinal
typedef unsigned short   word;      // 16-bits sem sinal
typedef unsigned int     longword;  // 32-bits sem sinal
typedef char             sbyte;         // 8-bits com sinal
typedef short            sword;         // 16-bits com sinal
typedef int              slongword;     // 32-bits com sinal
typedef byte*            pbyte;         // 8-bits
typedef word*            pword;         // 16-bits
typedef int*             plongword;     // 32-bits
typedef unsigned short   boolean;
typedef unsigned long    magicword;    // identificacao de objetos
typedef unsigned char    u_char;
typedef unsigned short   u_short;
typedef unsigned int     u_int;
typedef unsigned long    u_long;
typedef unsigned int     CPUContext;
typedef unsigned int     AID;          //  8-bits sem sinal
```

```
typedef void            (*EntryTable);
typedef int             NetAddress;
typedef void            (*Entry);

class   Activity;

typedef Activity*       pActivity;

#define   NULL    0
#define   TRUE    (!NULL)
#define   FALSE NULL

//-------------------------------------------------------
#endif  // _Types_h_DEFINED
```

Referências Bibliográficas

[1] ABRAM, M. D. Transparent Remote Procedure Calls. Dissertação de Mestrado, University of California, Santa Cruz, 1992.

[2] ALBUQUERQUE, F. *TCP/IP Internet: Programação de Sistemas Distribuídos*. Axcel Books, Rio de Janeiro, 2001.

[3] ALMEIDA, V. F. Um Modelo de Balanceamento de Carga para o Sistema Operacional Aurora. Dissertação de Mestrado, Universidade Federal de Santa Catarina (UFSC), Florianópolis, CPGCC/UFSC, 2003.

[4] ASSUMPCAO, J. M. O Sistema Orientado a Objetos Merlin em Máquinas Paralelas. *Anais do V SBAC-PAD, XIII Congresso da SBC* (1993), 305–312.

[5] AYERS, D.; BERGSTEN, H.; BOGOVICH, M.; DIAMOND, J.; FERRIS, M.; FLEURY, M.; HALBERSTADT, A.; HOULE, P.; MOHSENI, P.; PATZER, A.; PHILLIPS, R.; LI, S.; VEDATI, K.; WILCOX, M. & ZEIGER, S. *Professional Java Server Programming*. Wrox Press, USA, 1999.

[6] BALZAN, J. R. Uma Solução Reflexiva para Gerenciamento de Objetos Distribuídos em Aurora. Dissertação de Mestrado, Universidade Federal de Santa Catarina (UFSC), Florianópolis, CPGCC/UFSC, 2001.

[7] BALZAN, J. R.; PEREZ, A. L. F. & ZANCANELLA, L. C. AVLO: Um Mecanismo para Gerenciamento de Objetos Distribuídos no Sistema Operacional Aurora. *I ECTEC (Encontro de Ciência e Tecnologia - Lages/SC* (2002).

[8] BALZAN, J. R.; PEREZ, A. L. F. & ZANCANELLA, L. C. AVLO: Um Sistema de Gerenciamento de Objetos. *SEMINFO 2002 - Universidade de Cuiabá - Cuiabá/MT* (2002), 25–36.

[9] CHIN, R. S. & CHANSON, S. T. Distributed Object-Oriented Programming System. *ACM Computing Surveys. New York 23*, 1 (1991).

[10] CHUNG, E.; HUANG, Y.; YAJNIK, S.; LIANG, D.; SHIH, J. C.; WANG, C.-Y. & WANG, Y.-M. DCOM and CORBA side by side, step by step and layer by layer. Disponível em <http://citeseer.nj.nec.com/chung97dcom.html>. Acesso em 20 de Outubro de 2002, Setembro 1997.

[11] COMER, D. E. & STEVENS, D. L. *Internetworking with TCP/IP: Client-Server Programming and Applications*, vol. 3. Prentice-Hall, USA, 1997.

[12] COSTA, F.; BLAIR, G. & COULSON, G. Experiments with Reflective Middleware. *ECOOP'98 Workshop on Reflective Object Oriented Programming and Systems* (1998).

[13] COULOURIS, G.; DOLLIMORE, J. & KINDBERG, T. *Distributed Systems Concepts and Design*, 3º ed. Addison Wesley, USA, 2001.

[14] DEITEL, H. M. & DEITEL, P. J. *Java como Programar*, 3º ed. Bookman, Porto Alegre, 2001.

[15] DILLEY, J. Object Oriented Distributed Computing with C++ and OSF DCE. Request for Comments (RFC) 49.0, OSF DEC SIG, October 1993.

[16] EDDON, G. & EDDON, H. *Inside Distributed COM*. Microsoft Press, USA, 1998.

[17] GOSCINSKI, A. *Distributed Operating System: The Logical Design*. Addison-Wesley, Australia, 1991.

[18] GRIMES, R. *Professional DCOM Programming*. Wrox Press Ltd., Canadá, 1997.

[19] HORSTMANN, M. & KIRTLAND, M. DCOM architecture. Relatório técnico., Microsoft Corporation, 1997.

[20] JURIC, M.; ZIVKOVIC, A. & ROZMAN, I. Comparison of CORBA and java RMI based on performance analysis. Disponível em <http://citeseer.nj.nec.com/juric98comparison.html>. Acesso em 20 de Outubro de 2002, 1998.

[21] KAASHOEK, M.; TANENBAUM, A. S. & VERSTOEP, K. Group Communication in Amoeba and its Applications. *Distributed Systems Engineering Journal 1* (July 1993), 48–58.

[22] KAASHOEK, M.; VAN RENESSE, R.; VAN STAVEREN, H. & TANENBAUM, A. S. FLIP: an Internetwork Protocol for Supporting Distributed Systems. *ACM Transactions on Computer Systems* (February 1993), 73–106.

[23] KHALIDI, Y. A.; BERNABEU, J.; MATENA, V.; SHIRRIFF, K. & THADANI, M. Solaris MC: A Multi Computer OS. 191–204.

[24] KON, F.; CAMPBELL, R.; MICKUNAS, M. D.; NAHRSTEDT, K. & BALLESTEROS, F. J. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. *9th IEEE International Symposium on High Performance Distributed Computing* (August 2000), 1–4.

[25] KON, F.; COSTA, F.; CAMPBELL, R. & BLAIR, G. The Case for Reflective Middleware. *Communications of the ACM 45*, 6 (June 2002), 33–38.

[26] KON, F.; ROMÁN, M.; LIU, P.; MAO, J.; YAMANE, T.; MAGALHÃES, L. C. & CAMPBELL, R. H. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, April 2000, no. 1795 in LNCS, Springer-Verlag, p. 121–143.

[27] MAES, P. Concepts and Experiences in Computacional Reflection. *Sigplan Notices 22*, 12 (1987), 147–169.

[28] MICROSOFT. Windows 95 Review. Relatório técnico., Microsoft Corporation, 1995.

[29] MICROSOFT. Introdução ao .NET. Relatório técnico., Microsoft Corporation, 2003.

[30] MITCHELL, J. G.; GIBBONS, J.; HAMILTON, G.; KESSLER, P. B.; KHALIDI, Y. Y. A.; KOUGIOURIS, P.; MADANY, P.; NELSON, M. N.; POWELL, M. L. & RADIA, S. R. An Overview of the Spring System. *In: Proceedings of COMPCON Spring* (February 1994), 122–131.

[31] MULLENDER, S. J.; VAN ROSSUM, G.; TANENBAUM, A. S.; VAN RENESSE, R. & VAN STAVEREN, H. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer 23*, 5 (1990), 44–53.

[32] NASA. Learn about SETI@home. Disponível em <http://setiathome.ssl.berkeley.edu/learnmore.html>. Acesso em 22 de Janeiro de 2003, 2003.

[33] OMG. The Common Object Request Broker: Achitecture and Specification. Revision 2.5, Object Management Group, September 2001.

[34] ORFALI, R. & HARKEY, D. *Client-Server Programming with Java and Corba*, 2º ed. Wiley Computer Publishing, USA, 1998.

[35] ORFALI, R.; HARKEY, D. & EDWARDS, J. *Instant Corba*. John Wiley e Sons, Inc, USA, 1997.

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

[37] PEREZ, A. L. F.; BALZAN, J. R. & ZANCANELLA, L. C. Estrutura Reflexiva do Sistema Operacional Aurora. *I ECTEC (Encontro de Ciência e Tecnologia - Lages/SC* (2002).

[38] PEREZ, A. L. F. & ZANCANELLA, L. C. Sistema Operacional Aurora. *Revista do CCEI 7*, 11 (2003), 58–64.

[39] RICCIONI, P. R. *Introdução a Objetos Distribuídos com CORBA*. Visual Books, Florianópolis, 2000.

[40] ROMÁN, M.; MICKUNAS, D.; KON, F. & CAMPBELL, R. H. LegORB and Ubiquitous CORBA. In *In: Proceedings of the IFIP/ACM Middleware'200 Workshop on Reflective Middleware*, Palisades, NY, April 2000, p. 1–2.

[41] ROQUE, V. & OLIVEIRA, J. L. CORBA, DCOM e JavaRMI - uma análise comparativa. *EEI'99 (Encontro de Engenharia Informática 99 da O.E.)* (Dezembro 1999), 126–136.

[42] SILBERSCHATZ, A.; GALVIN, P. B. & GAGNE, G. *Operating System Concepts*, 6 ed. John Wiley e Sons, Inc, USA, 2003.

[43] SOUZA, A. C. Implementando Aplicações Distribuídas Utilizando CORBA e DCOM: Um Estudo de Caso Voltada à Área de Bando de Dados. Dissertação de Mestrado, Universidade Federal de Santa Catarina (UFSC), Florianópolis, CPGCC/UFSC, 1999.

[44] STALLINGS, W. *Operating Systems: Internals and Design Principles*, 3 ed. Prentice Hall, USA, 1998.

[45] STEVENS, W. R. *Unix Network Programming*, 2 ed. Prentice Hall, USA, 1998.

[46] SUN. Java Remote Method Invocation Specification. Relatório técnico., Sun Microsystems Inc., Palo Alto, 2002.

[47] TANENBAUM, A. S. *Distributed Operating Systems*. Prentice Hall, USA, 1995.

[48] TANENBAUM, A. S.; KAASHOEK, M. F.; VAN RENESSE, R. & BAL, H. E. The Amoeba Distributed Operating System: a status report. *Computer Communications 14* (1991), 324–335.

[49] TANENBAUM, A. S. & VAN STEEN, M. *Ditributed Systems: Principals and Paradigms*. Prentice Hall, USA, 2002.

[50] TANENBAUM, A. S. & WOODHULL, A. S. *Sistemas Operacionais: Projeto e Implementação*, 2 ed. Bookman, São Paulo, 1997.

[51] TEIXEIRA, J. H. *Do Mainframe a Computação Distribuída: Simplificando a Transição*. Infobook, Rio de Janeiro, 1996.

[52] TERAOKA, F.; YOKOTE, Y. & TOKORO, M. Muse-IP: A network layer protocol for large distributed systems with mobile hosts. *In: Proceedings of the 4th Joing Workshop on Computer Communications* (June 1989).

[53] TRIPATHI, A. R. & NOONAN, T. Design of a Remote Procedure Call system for object-oriented distributed programming. *Software Practice and Experience 28*, 1 (1998), 23–48.

[54] WUTKA, M. *Java Técnicas Profissionais*. Berkeley, São Paulo, 1997.

[55] YOKOTE, Y. The Apertos Reflective Operating System: The Concept and Its Implementation. *Technical Report, Sony Computer Science Laboratory Inc.* (1992), 103–125.

[56] YOKOTE, Y.; TERAOKA, F. & TOKORO, M. A Reflective Architecture for an Object-Oriented Distributed Operating System. *In: Proceedings of European Conference on Object-Oriented Programming* (March 1989).

[57] YOKOTE, Y. & TOKORO, M. The new structure of an operating system: the Apertos approach. In *In: Proceedings of the 5th workshop on ACM SIGOPS European workshop*, 1992, ACM Press, p. 1–6.

[58] ZANCANELLA, L. C. *Estrutura Reflexiva para Sistemas Operacionais Multiprocessados*. Tese de Doutorado, Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, PGCC/UFRGS, 1997.

[59] ZANCANELLA, L. C. & PEREZ, A. L. F. Reflexão Computacional: A Nova Dimensão da Programação Orientada a Objetos. *XI Escola de Informática da SBC Sul - SC* (2003), 99–117.

[60] ZIMMERMANN, C. *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, Boca Raton - Florida, 1996.