UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO

Dissertação de Mestrado

UM AMBIENTE PARA O DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS À CONFIGURAÇÃO UTILIZANDO OBJETOS DISTRIBUÍDOS

Cidcley Teixeira de Souza

Orientador : Prof. Dr. Mauro Oliveira

27 de Dezembro, 1996

Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação
Pós-Graduação em Ciência da Computação

Cidcley Teixeira de Souza

UM AMBIENTE PARA O DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS À CONFIGURAÇÃO UTILIZANDO OBJETOS DISTRIBUÍDOS

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Agradecimentos

Primeiramente gostaria de agradecer a DEUS, pela divina luz que me guiou nessa jornada.

A meu orientador, Prof. Mauro Oliveira, pela confiança depositada em mim, pela oportunidade de realizar esse trabalho e pelo constante incentivo sempre me dispensado desde o início de nossas atividades de pesquisa.

Agradeço também a meus pais, pela força que me deram durante as horas mais cansativas e difíceis. A meu irmão, por fazer minhas vezes e resolver alguns problemas quando a falta de tempo não permitia que eu o fizesse.

A minha querida Fatinha, pelo incentivo, pelas horas de descontração a mim proporcionadas e principalmente pela imensa paciência ao longo dos meses que passei escrevendo esse trabalho.

Aos colegas do mestrado, pela companhia e pela amizade dispensada.

Ao secretário do nosso mestrado, Orley, por tanta garra em resolver todos os nossos problemas.

Resumo

A tecnologia de objetos distribuídos, que tem sido amplamente utilizada atualmente, combina os conceitos de orientação à objetos e sistemas distribuídos em uma só ambiente, criando um novo modelo de interação cliente/servidor, onde dados e comportamentos são encapsulados em objetos que podem ser executados em diferentes plataformas.

Embora os benefícios do modelo de objetos distribuídos sejam muitos, a compreensão da estrutura de interconexão dos objetos que compõem uma aplicação distribuída, desenvolvida utilizando-se esse modelo, pode tornar-se uma tarefa bastante tortuosa e, dependendo da complexidade da aplicação e da quantidade de objetos envolvidos, totalmente inviável. O paradigma de configuração, que é uma técnica de engenharia de *software* na qual a construção de um sistema é baseada na separação dos componentes (objetos independentes de contexto) e da estrutura de interconexão desses componentes, se mostrou uma ferramenta poderosa para o desenvolvimento e manutenção de aplicações complexas com objetos distribuídos.

Este trabalho, propõe e implementa o ÁBACO, um ambiente destinado ao desenvolvimento de aplicações orientadas à configuração utilizando objetos distribuídos. ÁBACO se apresenta como uma solução a problemas complexos de interconexão de objetos clientes a objetos servidores. Ele é suportado por plataformas baseadas em objetos distribuídos como o CORBA, dotando-as de características que permitam a configuração de aplicações, de forma a utilizar todas as vantagem do paradigma de configuração, nestas plataformas que não o implementam diretamente.

Abstract

The technology of distributed objects, that it is being broadly used now, combines the concepts of object orientation and distributed systems in the same environment, creating a new client/server interaction model, where data and behaviors are encapsulated in objects that can be executed in different platforms.

Although the benefits of the distributed objects model are many, the understanding of the interconection structure of the objects that compose a distributed application, that was developed using this model, can become a quite tortuous task and totally unviable, depending of the complexity of the application and of the amount of involved objects. The configuration paradigm, that is a technique of software engineering in which the construction of a system is based on the separation of the components (objects independent of context) and the interconection structure of those components, showed a powerful tool for the development and maintenance of complex applications with distributed objects.

This work, proposes and it implements the ÁBACO, an environment for the development of configuration oriented applications using distributed objects. ÁBACO comes as a solution to complex problems of interconection of clientes objects with servers objects. He is supported by platforms based on distributed objects like CORBA, endowing them of characteristics that allow the configuration of applications, in way to use all them advantage of the configuration paradigm, in these platforms that don't implement it directly.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

> ***"Trabalho intelectual é uma expressão errada. Não é "trabalho"- é prazer, dissipação, nossa maior recompensa."***
>
> **Mark Twain**

1.1 Contexto

A tecnologia de redes de computadores tem evoluído muito rapidamente desde o seu surgimento nos anos 60 aos dias de hoje. Essa evolução trás consigo um conjunto de conseqüências tanto tecnológicas quanto sociais, que nos leva a uma importante reflexão sobre a nova função da sociedade globalizada, extremamente modificada no seu modo de vida. Novos hábitos têm sido impostos nesta nova sociedade informatizada: do correio eletrônico à tecnologia WWW (*World Wide Web*) que tem transformado a Internet no maior fenômeno de comunicação mundial deste final de século.

A difusão da tecnologia das redes de computadores, que invade desde os nossos lares aos escritórios das grandes corporações, tem como principal atrativo a possibilidade da distribuição do poder computacional através da descentralização do processamento. Isto trouxe inúmeras vantagens, tais como o compartilhamento de recursos de *hardware* e *software*, facilidade de utilização e principalmente a capacidade de troca de informações. Com essa nova infra-estrutura de processamento descentralizado, tornou-se possível definir e projetar sistemas em que as informações estão distribuídas em vários computadores, propiciando um ambiente de trabalho cooperativo [Soares 95].

A popularização das LANs (*Local Area Networks)* e a diminuição dos custos dos processadores, fizeram surgir os chamados sistemas distribuídos. Estes sistemas de computação são definidos como "uma coleção de computadores independentes, que se apresentam ao usuário como um único computador" [Tannebaum 96]. Segundo [Kramer 87], "um sistema distribuído é aquele no qual existe um conjunto de processadores

independentes que interagem através da troca de mensagens, cooperando na execução de uma tarefa comum"[1].

Embora os benefícios potenciais dos sistemas distribuídos sejam muitos, o desenvolvimento de aplicações para essa nova tecnologia requer, em geral, do desenvolvedor um grande conhecimento de vários aspectos não relevantes ao escopo da aplicação. Essa dificuldade no desenvolvimento de aplicações para sistemas distribuídos, fez surgir algumas plataformas de distribuição. Estas plataformas se caracterizam pela utilização de modelos, pela adoção de um conjunto de ferramentas e de uma infra-estrutura de execução que permitem ao desenvolvedor concentrar seus esforços na aplicação em si, deixando que a plataforma se encarregue de aspectos relativos à distribuição e à comunicação.

Dentre estes modelos destaca-se o paradigma da Orientação a Objetos que tem, naturalmente, grande influência no desenvolvimento de *software* [Rumbaugh 91, Booch 94]. Várias características da orientação a objetos, como o encapsulamento, são bastante relevantes à programação distribuída.

Outro modelo é o de Objetos Distribuídos ou DOC (*Distributed Object Computing*), que tem sido amplamente utilizado, recentemente. Isto se deve ao fato da DOC permitir a implementação de aplicações utilizando o modelo clássico cliente/servidor associado ao paradigma da orientação a objetos. O encapsulamento de dados e métodos (operações) e suas respectivas interfaces caracterizam serviços nesse modelo, fornecendo aplicações cliente/servidor mais flexíveis[2]. A afinidade da modelagem orientada a objetos com sistema distribuídos [Schmidt 95] tem motivado o

[1] Como pode ser observado, não existe no meio acadêmico consenso quanto a definição sobre sistemas distribuídos.

[2] Neste modelo, servidores tornam seus serviços disponíveis, através do registro do nome e do tipo do serviço, juntamente com uma referência para aquele serviço em um servidor de nomes. Clientes utilizam um serviço particular através da consulta ao servidor de nomes e da obtenção da referência desse serviço. De posse desta referência, o serviço pode ser invocado diretamente pelo cliente.

surgimento de várias plataformas de distribuição baseadas ou orientadas nesse paradigma, tais como os sistemas ANSA [APM 93] e CORBA [CORBA 91], respectivamente. Estes sistemas distribuídos utilizam o modelo cliente/servidor como mecanismo de comunicação entre os objetos das aplicações.

1.2 Motivação

O modelo cliente/servidor associado ao paradigma de orientação a objetos apresenta problemas quando o sistema que deve ser construído foge ao padrão simples (muitos-para-um), ou seja, tem uma estrutura de interconexão mais complexa (muitos-para-muitos). Embora todo sistema razoável permita que servidores sejam clientes de outros servidores, o processo de interconexão torna-se bastante tortuoso [Kramer 92].

A solução a este problema passa pela utilização de um mecanismo que permita esta interconexão de uma forma mais flexível, isto é, a utilização de um mecanismo que facilite a associação entre os objetos clientes e servidores.

Uma possível solução ao problema acima é o paradigma de configuração [Kramer 92, Sloman 89, Magee 90]. Vários ambientes para construção de aplicações distribuídas foram implementados utilizando configuração, uma técnica de engenharia de *software* na qual a construção de um sistema é baseada na separação entre sua estrutura e de seus componentes.

Usando o paradigma de configuração, os componentes explicitamente declaram tanto os serviços que eles fornecem, como os serviços que eles necessitam (Figura 1.1). O termo “componente” é

declarado como um módulo que possui independência de contexto, ou seja, não possui conhecimento direto de sua ligação com o ambiente externo. Em termos do modelo cliente/servidor, os serviços requeridos (círculos vazios na figura 1.1) devem ser vistos como referências para outros componentes.

Sistemas distribuídos baseados em configuração são descritos por meio de um linguagem de configuração que define a estrutura da aplicação em termos das instâncias de componentes e das ligações entre os serviços requeridos e dos serviços fornecidos.

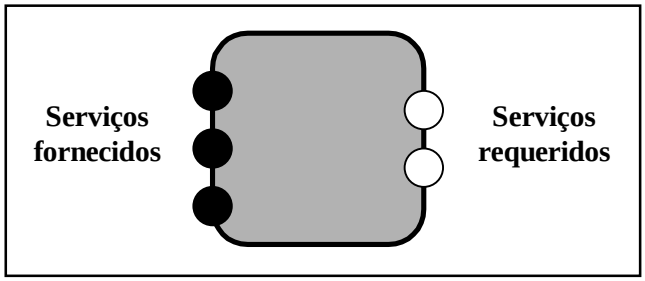


Figura 1.1 - Componente independente de contexto

Este trabalho, propõe e implementa o ÁBACO, um ambiente destinado ao desenvolvimento de aplicações orientadas à configuração utilizando objetos distribuídos. ÁBACO se apresenta como uma solução a problemas complexos de interconexão de objetos clientes a objetos servidores. Ele é suportado por plataformas baseadas em objetos distribuídos, dotando-as de características que permitam a configuração de aplicações, de forma a utilizar todas as vantagem do paradigma de configuração nestas plataformas que não podem implementá-lo diretamente.

ANSA e CORBA são as duas plataformas de investigação do trabalho realizado, devido as seguintes razões:" a primeira plataforma tem sua importância por ter sido a base dos trabalhos do RM-ODP (*Reference Model - Open Distributed Processing*) [ISO 95a, ISO 95b, ISO 95c] e também devido sua popularidade por ocasião do início deste trabalho. A

segunda plataforma, CORBA, tem se mostrado como uma solução bastante promissora para o desenvolvimento de aplicações distribuídas [Schmidt 95], sendo hoje o modelo mais utilizado no desenvolvimento desse tipo de aplicação.

1.3 Estrutura

Este trabalho está organizado em treze capítulos, distribuídos em três partes:

Parte I: Sistemas Distribuídos

O capítulo 2 apresenta padrões para sistemas distribuídos, mais especificamente o RM-ODP. O capítulo 3 descreve as arquiteturas ANSA e CORBA/OMA que serviram de modelos para a especificação da estrutura do ambiente ÁBACO. O capítulo 4 apresenta as plataformas de distribuição DCE, ANSAware e ORBeline.

Parte II: O Paradigma de Configuração

O capítulo 5 descreve a estrutura de software para a implementação do paradigma de configuração. O capítulo 6 discute os modelos de configuração e apresenta os principais ambientes que implementam este paradigma: CONIC, REX, RIO e CL. O capítulo 7 mostra o desenvolvimento de aplicações utilizando o paradigma de configuração, exemplificando os conceitos em CONIC.

Parte III: O Ambiente ÁBACO

O capítulo 8 descreve a estrutura geral do ambiente ÁBACO: arquitetura de software e o ambiente de suporte a execução. O capítulo 9 detalha as ferramentas utilizadas no desenvolvimento de aplicações com o ÁBACO. O capítulo 10 implementa o sistema de gerenciamento de configuração do ÁBACO. O capítulo 11 descreve uma metodologia para o desenvolvimento de aplicações com as ferramentas especificadas no capítulo 9. O capítulo 12 apresenta dois estudos de casos com as plataformas ANSA e CORBA. O capítulo 13 conclui este trabalho com uma análise crítica dos resultados obtidos, apresentando propostas para trabalhos futuros.

PARTE I

Sistemas Distribuídos

- **Capítulo 2: Padrões para Sistemas Distribuídos**
- **Capítulo 3: Arquiteturas para Sistemas Distribuídos**
- **Capítulo 4: Plataformas de Distribuição**

Capítulo 2

Padrões para Sistemas Distribuídos

"A meta final de qualquer pesquisa não é a objetividade, mas a verdade"

Helene Deutsch

2.1 Introdução

O uso cada vez maior de aplicações utilizando sistemas distribuídos, tem motivado o aparecimento de ferramentas e técnicas necessárias para implementá-los. A exemplo das redes de computadores, os sistemas distribuídos têm necessidade de alguma forma de padronização com vistas a uma melhor interoperabilidade. Para tanto, são propostos modelos de referência que também ajudam na construção da infraestrutura e no desenvolvimento de aplicações distribuídas.

Modelos de referência são a base para a especificação e padronização de sistemas pela ISO. Eles têm se tornado uma técnica útil não só na definição de padrões como também um recurso importante para o entendimento destes padrões. O crescimento rápido da utilização de processamento distribuído, fez surgir a necessidade de uma padronização para processamento distribuído aberto. Trata-se do Modelo de Referência ODP (*Open Distributed Processing*).

2.2 Modelo de referência ODP

O ODP é um modelo de referência com o objetivo de guiar o projeto e desenvolvimento de sistemas distribuídos. Diferentemente do Modelo de Referência OSI, que estrutura a funcionalidade de uma rede de computadores numa pilha de protocolos interdependentes, o ODP considera os diferentes aspectos (visões) na construção de sistemas distribuídos em cinco pontos de vista auto-contidas (Empresa, Informacional, Computacional, Engenharia e Tecnológico), detalhados no item 2.2.2.

O Modelo de Referência ODP [ISO 95a, ISO 95b , ISO 95c] (ISO 10746) consiste dos seguintes documentos:

- **Overview** (ITU-T Rec. X901| ISO/IEC 10746-1) **:** contém a motivação do ODP, dando o escopo, justificativa e a explicação dos conceitos chaves.
- **Foundations** (ITU-T Rec. X.902 | ISO/IEC 10746-2) **:** contém a definição e conceitos e uma estrutura analítica para sistemas de processamento distribuído.
- **Architecture** (ITU-T Rec. X.903 | ISO/IEC 10746-3) **:** contém a especificação das características que qualificam um processamento distribuído como sendo aberto.
- **Architectural semantics** (ITU-T Rec. X.903 | ISO/IEC 10746-3) **:** contém a formalização do modelo ODP.

2.2.1 Objetivos e motivação

Os objetivos do ODP são o desenvolvimento de um padrão permitindo que os benefícios dos serviços de processamento de informação distribuídos, sejam realizados em plataformas de telecomunicações heterogêneas. Para isso o Modelo de Referência ODP define:

- uma visão de um sistema distribuído a partir de *pontos de vista*, de forma a simplificar a descrição de sistemas complexos.
- um conjunto de conceitos gerais para se expressar esses pontos de vista.
- um modelo para suportar a infraestrutura, fornecendo um conjunto de *transparências de distribuição.*
- princípios para desenvolvimento de conformidade com o ODP.

2.2.2 Pontos de vista

No ODP, um sistema é considerado a partir de diferentes pontos de vista, permitindo a descrição de um sistema distribuído com o resultado relevando suas diferentes facetas. Cada descrição é completa e auto suficiente para especificar o sistema, não importando quanto do sistema foi descrito mas que aspectos do sistema foram enfatizados [Oliveira 92].

Cinco pontos de vista são considerados:

- **Empresa:** descreve os objetivos do sistema em termos de funções e atividades que existem dentro da organização utilizando o sistema; as interações entre o sistema e o ambiente no qual está inserido; a estrutura organizacional da empresa; quais informações serão acessíveis aos diferentes usuários, etc.
- **Informação:** especifica as estruturas dos elementos de informação de um sistema, as regras estabelecendo as relações entre esses elementos e as restrições impostas a ambos, regras e elementos. Esse ponto de vista deve também mostrar como a informação é distribuída através do sistema e não se preocupa em especificar quais partes do sistema serão executadas automaticamente ou manualmente.
- **Computação:** fornece as estruturas de programação e as ferramentas para o desenvolvimento de programas que estarão disponíveis aos programadores de aplicações distribuídas.
- **Engenharia:** fornece ao programador de sistemas os mecanismos de tradução (compilação, ligação e edição, interpretação) e um núcleo de funções de suporte de base, necessários à realização da computação em ambientes distribuídos e heterogêneos.

- **Tecnologia:** especifica as ferramentas operacionais nas quais o sistema distribuído é construído. A descrição pode incluir padrões OSI ou tecnologias proprietárias. Esse modelo mostra como o *hardware* e *software* são mapeados nos mecanismos identificados no modelo de engenharia, incluindo sistemas operacionais locais, dispositivos de entrada/saída, memória e pontos de acesso para comunicação.

2.2.3 Conceitos gerais

Existe um conjunto de conceitos gerais que dão uma estrutura comum as especificações dos pontos de vista do ODP. O modelagem orientada à objetos foi escolhida para se especificar as características dos sistema. Assim, cada especificação de sistema ODP, é baseado no conceito de objetos.

Objetos são entidades que possuem informações e oferecem serviços. O modelo de objetos do ODP é genérico, tendo as seguintes características [OSI 95a]:

- objetos podem ser das mais variadas granularidades (ex. podem ser tão grandes quanto uma rede telefônica ou tão pequenos quanto um número inteiro).
- objetos podem exibir comportamentos arbitrários e ter um nível arbitrário de paralelismo.
- interações entre objetos são irrestritas.

Um sistema ODP é composto de objetos que interagem. Objetos podem interagir somente através de um *interface*, onde essa *interface* representa uma parte do comportamento do objeto relativas a um subconjunto particular de possíveis interações.

2.2.4 Transparências

Um sistema distribuído foi definido no item 1.1 como um sistema computacional constituído por várias unidades de computação autônomas, suportado por uma rede de computadores, mas que se apresenta ao usuário como se fosse um sistema centralizado. Desta definição conclui-se que o termo **transparência** assume importância capital em um sistema distribuído, a medida que o usuário não sabe (e não tem necessidade de saber) onde estão sendo executados os processos a ele associados, nem tampouco onde estão fisicamente localizados seu arquivos.

Transparências são, portanto, utilizadas para "esconder" certos aspectos dos sistemas distribuídos que surgem com a utilização de mecanismos de distribuição. Dentro de um sistema ODP, a infraestrutura suporta um conjunto de transparências. As aplicações determinam quais transparências irão utilizar.

As transparências definidas pelo RM-ODP são:

- **Transparências de acesso:** mascara as diferentes representações dos dados e mecanismos de invocação para permitir a interação entre objetos em sistemas heterogêneos;
- **Transparências de falha:** mascara para um objeto a falha e a possível recuperação de outro objeto (ou dele mesmo) para permitir tolerância a falhas;
- **Transparências de localização:** mascara o uso de informações sobre localização no espaço, quando da identificação e ligação entre objetos;
- **Transparências de migração:** mascara a habilidade de objeto mudar de localização;

- **Transparências de relocação:** mascara a relocação de um *interface* para uma outra *interface*. Relocação permite que uma aplicação continue mesmo quando ocorrer migração ou troca de alguns objetos criando inconsistências temporárias;
- **Transparências de replicação:** mascara a utilização de grupos de objetos com comportamento compatíveis;
- **Transparências de persistência:** mascara os mecanismos de desativação ou reativação de objetos;
- **Transparências de transação:** mascara a coordenação de atividades quando de uma Configuração de objetos para conseguir consistência.

Capítulo 3

Arquiteturas de Sistemas Distribuídos

"Nunca se percebe o que já foi feito; a gente só nota o que ainda está por fazer"

3.1 Introdução

Uma arquitetura para sistemas distribuídos compreende a descrição da estrutura lógica dos conceitos de um ambiente para programação distribuída. Ela fornece elementos para a construção de plataformas de distribuição, tais como: o modelo dos componentes de *software*, as características dos mecanismos de comunicação, etc.

Nesse capítulo, as arquiteturas de sistemas distribuídos ANSA e ORBA, cujas respectivas plataformas ANSAware e ORBeline foram utilizadas neste trabalho, são apresentadas, em detalhes. A decisão de se utilizar a arquitetura ANSA neste trabalho é justificada pela importância dessa arquitetura na definição do Modelo de Referência ODP. Já o CORBA se destaca devido sua grande versatilidade e popularidade atual, bem como pela sua importância na padronização de sistemas distribuídos orientados a objetos.

3.2 A arquitetura ANSA

3.2.1 Objetivos

A arquitetura ANSA (*Advanced Networked System Architecture*) teve sua origem num projeto cujo objetivo era fornecer uma arquitetura para sistemas distribuídos que satisfizesse os seguintes objetivos:

- ser genérico para muitos campos de aplicação (incluindo escritórios, telecomunicações, fábricas e processamentos de dados em geral).
- ser o “estado da arte”.

- ser portável entre uma grande classe de sistemas operacionais e linguagens de programação.
- ser operável em ambientes heterogêneos.
- ser modular com o máxima possibilidade de reuso das funcionalidades existentes.
- fornecer uma interoperabilidade entre redes gerenciadas autônomas.

3.2.2 Atividades

O projeto ANSA teve sua concepção voltada para dois aspectos convergentes: *integração* e *padronização*. A principal intenção era, portanto, possibilitar a integração de sistemas de aplicação de vários fabricantes, criando uma arquitetura comum para sistemas de computação distribuída e utilizando essa arquitetura no desenvolvimento de padrões. Assim, pode-se afirmar que o projeto se concentrou em *possibilitar* ao invés de *fornecer* integração.

As atividades associadas com o projeto "ANSA" são as seguintes:

- **arquitetura**: desenvolvimento de uma arquitetura para construção de sistemas distribuídos.
- **software**: desenvolvimento de um software para demonstrar e validar a arquitetura (conhecido como ANSA Testbench).
- **padrões**: contribuição do resultados do ANSA para padrões internacionais.
- **transferência de tecnologia**: fornecer conhecimento técnico para permitir o desenvolvimento de processamento distribuído aberto.

3.2.3 A projeção computacional do ANSA

A projeção ou ponto de vista computacional do ANSA está ligado aos requisitos dos programas de aplicações distribuídas. Esta projeção fornece os conceitos necessários para definir um modelo computacional destinado a linguagens de programação para aplicações distribuídas.

Esse modelo computacional especifica as características das linguagens de programação que são necessárias para possibilitar a escrita de programas para um ambiente distribuído. O modelo computacional do ANSA inclui as seguintes características:

- o modelo orientado a objetos que separa a especificação de interfaces da definição dos objetos.
- um sistema de interfaces que permite a checagem de tipos.
- suporte ao modelo cliente-servidor.
- especificação precisa das interfaces, incluindo as ações, dados e propriedades que caracterizam a maneira de como um componente interage com um outro.
- interação entre componentes, modelado como invocação de operações de *interfaces*.

O Trader

O Trader é um importante componente na arquitetura ANSA. Trata-se de uma interface disponível a qualquer programa, associada à transparência de localização. Ele é basicamente uma estrutura de diretório que pode ser acessada pelo nome do *path*, pelas propriedades ou por alguma combinação de ambos.

O servidor pode exportar uma *interface* para o Trader para torná-la acessível à outros programas. Uma função de importação é fornecida para clientes de modo que eles possam obter *interfaces* do Trader. A operação de importação retorna uma referência de *interface* ao chamador. Essa referência é arranjada de modo a não ser ambígua dentro do domínio do Trader (um sistema de *trading* pode ser estruturado como uma federação de domínios autônomos de Traders).

Referências de *interfaces* devem ser localizados transparentemente, a infraestrutura deve prover um objeto localizador distribuído.

3.3 O ambiente CORBA/OMA

3.3.1 Visão Geral

CORBA (Commom Object Request Broker) [CORBA 91] é uma plataforma de distribuição resultado da arquitetura OMA (Object Management Architecture) do OMG (Object Management Group) [OMG 95], uma organização internacional, fundada em maio de 1989 por 8 empresas:[3] A plataforma CORBA fornece mecanismos pelos quais objetos, transparentemente, fazem pedidos e recebem respostas em um ambiente distribuído.

A arquitetura OMA objetivava reduzir a complexidade, diminuir os custos e acelerar a introdução de novas aplicações distribuídas, sem que para isso seja preciso realizar grandes mudanças. Um modelo de referência OMA foi então especificado, caracterizando os componentes, as interfaces e protocolos, resultando daí a plataforma CORBA.

[3] 3Com Corporation, American Airlines, Canon Inc, Data General, Hewlett-Packard, Phillips Telecommunications N.V., Sun MicroSystems e Unisys Corporation.

O intuito do OMG era ajudar o desenvolvimento e o crescimento da tecnologia de orientação a objetos. Seus princípios incluem o estabelecimento de diretrizes industriais e especificações de gerenciamento de objetos para prover uma base única para o desenvolvimento de aplicações[4]. O OMG adota uma tecnologia denominada ORB (*Object Request Broker*), que fornece interoperabilidade entre aplicações em diferentes máquinas em ambientes heterogêneos distribuídos. O ORB tem as seguintes características:

- Provê mecanismos pelos quais os objetos fazem pedidos e recebem respostas de forma transparente;
- Provê interoperabilidade entre aplicações em diferentes máquinas em ambientes distribuídos e heterogêneos;

CORBA é, portanto, a tecnologia ORB adotada pelo OMG. Ela define uma estrutura para que diferentes implementações de ORBs possam prover serviços e *interfaces* comuns para suportar clientes e implementações de objetos portáveis.

Dentre os aspectos relevantes para a caracterização da plataforma CORBA, destacam-se os seguintes:

- semântica dos objetos
- modelo de objetos
- implementação de objetos

3.3.1.1 Semântica dos objetos

- **objetos**: entidade identificável e encapsulada que provê serviços que podem ser requisitados por clientes;

[4]A adequação a essas especificações tornará possível desenvolver ambientes de aplicação heterogêneos através de diferentes plataformas de hardware e sistemas operacionais

- **pedidos**: evento que acontece em um tempo definido. As informações associadas a um pedido são operação, objeto alvo, parâmetros reais (que podem ser de entrada, de saída ou de entrada e saída) e um contexto de pedido opcional[5].
- **referências a objetos**: valor que identifica um objeto e que seguramente denota um (e somente um) objeto particular[6].
- **criação e destruição de objetos**: acontecem como resultados de pedidos. Do ponto de vista do cliente, não existe nenhum mecanismo especial para criação e destruição de objetos.
- **tipos**: conceito tradicional de tipos. Entidade identificável com um predicado, associado definido sobre os valores membros deste tipo.
- **interface**: descrição do conjunto de operações possíveis de um objeto que podem ser requisitadas por um cliente. Um objeto pode suportar múltiplas *interfaces* através de mecanismos como herança. As *interfaces* são especificadas em uma linguagem denominada IDL *(Interface Defnition Language*).
- **operação**: entidade que denota um serviço a ser requisitado. É extremamente genérica, na medida em que é independente da implementação dos objetos e se utiliza de mecanismos de herança de *interfaces* em IDL.

Uma operação possui uma **assinatura**, a qual descreve todos os valores de parâmetros e resultados possíveis, através de:

- especificação de parâmetros necessários em chamadas a esta operação
- especificação de resultados da operação
- especificação das exceções que podem ocorrer durante a execução do serviço

[5]Um pedido causa a execução de um determinado serviço, e retorna uma exceção se acontecer uma condição anormal durante a sua execução.
[6]Em contrapartida, um objeto pode ser denotado por diversas referências;

- especificação de informação adicional de contexto
- indicação da semântica de execução que o cliente espera encontrar na ocasião de um pedido a esta operação.

A forma geral da **assinatura** de uma operação é:

[ONEWAY] <OP_TYPE_SPEC> <IDENTIFIER> (PARAM1,...,PARAML) [RAISES] (EXCEPT1,...,EXCEPTN)] [CONTEXT(NAME1,...,NAMEM)]

Onde:

- **oneway** indica a semântica "*best-effort*" de execução, na qual a operação não retorna nenhum resultado, e não há sincronização entre o requisitante e o término da operação. O *default* é a semântica "*at-most-once*", na qual garante-se que se uma operação retorna com sucesso, esta foi executada exatamente uma vez. No caso de exceção, foi executada uma vez no máximo[7].
- **op_type_spec** é o tipo do retorno da operação, que é um parâmetro de saída distinto.
- **identifier** é o nome da operação.
- **parâmetros** são modificados para indicar a direção da informação: **in** significa que a informação passa do cliente para o servidor, **out** do servidor para o cliente e **inout** em ambas as direções.
- a expressão **raises** é seguida de uma lista de exceções definidas pelo usuário (as exceções padrão são incluídas implicitamente) que podem ser sinalizadas para terminar o pedido, indicando que a operação não foi executada com sucesso[8].

[7] A semântica de execução é associada a cada operação, o que garante que tanto o cliente quanto a implementação do objeto sempre assumirão a mesma semântica.
[8] Tais exceções podem ser descritas por um registro, no qual são acompanhadas por informação adicional específica da exceção.

- **context** é uma expressão opcional que indica o contexto de pedido disponível na implementação do objeto, e pode afetar a performance do pedido.
- **atributos**: Analogamente aos conceitos de orientação a objetos pura, os atributos de uma interface são logicamente equivalentes a declaração de um par de funções de acesso: uma para recuperação do valor do atributo e outra para "setar" o seu valor (caso o atributo não seja apenas para leitura).

3.3.1.2 O modelo de objetos

Um modelo de objetos provê uma apresentação organizada dos conceitos e terminologia dos objetos, e define um modelo parcial para implementação que engloba as características fundamentais de objetos que devem ser consideradas pelas tecnologias.

O modelo de objetos definido pelo OMG é um modelo abstrato, na medida em que não é diretamente realizado por uma tecnologia particular. Já o modelo de objetos do CORBA é um modelo concreto que foi derivado do modelo abstrato do OMG, e difere-se deste último por:

- ser mais específico (define-se aspectos como a forma dos parâmetros dos pedidos, linguagem de especificação de tipos, etc.).
- introduzir instâncias específicas de entidades definidas no modelo.
- restringir o modelo, eliminando entidades ou acrescentando restrições adicionais no seu uso.

No sistema de objetos do CORBA existe uma clara separação entre os clientes (requisitantes de serviços) e os servidores (provedores de

serviços), através de uma *interface* de encapsulamento bem definida. Quanto aos clientes, o modelo de objetos é muito específico ao definir conceitos relevantes como criação e identificação de objetos, pedidos e operações, tipos e assinaturas. Quanto ao aspecto de implementação dos objetos, o modelo trata de conceitos como métodos, execução e ativação, porém de uma maneira mais sugestiva, dando máxima liberdade para que cada tecnologia possa implementar os objetos a sua maneira.

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

3.3.1.3 Implementação de objetos

A implementação de um sistema de objetos inclui atividades que podem requisitar serviços adicionais, como por exemplo cálculo de resultados dos pedidos e atualização do estado do sistema. O modelo de implementação divide-se em 2 partes: O *Modelo de Execução* que descreve como os serviços são executados, e o *Modelo de Construção* que descreve como os serviços são definidos

- **Modelo de execução**: Um serviço requisitado é realizado pela execução de um código que manipula determinados dados e pode alterar o estado do sistema. Tal código é denominado um método. A execução de um método por uma máquina abstrata é chamada de ativação do método

- **Modelo de construção**: Definem-se os mecanismos para obtenção do comportamento dos pedidos, como definições do estado do objeto, definição dos métodos, definição de como é a infraestrutura do objeto para que ele selecione apropriadamente os métodos a serem executados e definição das ações concretas que devem ser tomadas quando da criação de um objeto, como por exemplo associação do novo objeto aos seus métodos

3.3.2. *Object Request Broker*

Qualquer objeto desenvolvido em conformidade com o padrão ORB deve garantir portabilidade e interoperabilidade de objetos sobre uma rede de sistemas heterogêneos. Basta que o ORB seja definido por suas *interfaces* e, então, qualquer implementação adequada às *interfaces* se torna aceitável. Tais *interfaces* são organizadas em 3 categorias:

- operações padrão para toda implementação de ORB;
- operações específicas a um tipo de objeto;
- operações específicas a um estilo de implementação de objetos;

3.3.2.1 Exemplos de ORBs

É possível que haja a coexistência de diferentes implementações de ORB, com representações distintas de referência de objetos e modos distintos de invocação de operações. Para que haja tal coexistência, onde há a possibilidade de um cliente ter acesso simultâneo a objetos gerenciados por ORBs diferentes, é importante que os próprios ORBs sejam capazes de distinguir as suas referências a objetos, o que deve ser feito de forma transparente para o cliente.

A seguir são apresentados vários exemplos de implementação do ORB:

- **residente no cliente e na implementação**: implementado em rotinas residentes tanto no cliente quanto na implementação. As *stubs* de cliente utilizam mecanismos IPC ou um serviço de localização do ORB para estabelecer comunicação com as implementações.
- **baseado em servidor :** centralizam o gerenciamento do ORB. Todos os clientes e as implementações se comunicam com um ou mais servidores que realizam o roteamento dos pedidos. Para o SO, o ORB seria um programa comum e a comunicação com ele seria realizada através do mecanismo IPC.
- **baseado no sistema**: neste caso, o ORB seria um serviço básico oferecido pelo próprio SO, o que aumenta segurança, robustez e desempenho, deixando a cargo do SO a realização de algumas otimizações, como no caso de um pedido a um objeto que se encontra na mesma máquina.
- **baseado em bibliotecas**: a implementação do objeto poderia estar em uma biblioteca, e as *stubs* do cliente poderiam ser mapeadas como métodos. Neste caso, assume-se que o programa cliente não destruirá os dados do objeto, e que as implementações são de objetos não muito grandes.

3.3.2.2 Estrutura de um ORB

A figura 3.1, mostra a estrutura de um ORB, cujos componentes serão detalhados a seguir:

- **ORB core:** parte do ORB que provê a representação básica dos objetos e comunicação de pedidos. Existem componentes acima do ORB Core que provêem interfaces para mascarar possíveis

diferenças entre mecanismos de objetos para que estes possam coexistir;

- **Cliente:** um cliente de um objeto tem acesso à referência do objeto e chama operações sobre ele. O cliente de um objeto conhece a estrutura lógica do objeto, de acordo com a sua *interface*, e não sabe nada sobre a sua implementação, qual o adaptador de objetos usado ou qual o ORB usado para acessá-lo. É importante ressaltar que um cliente não é sempre um programa de aplicação: como exemplo, a implementação de um objeto pode ser cliente de outros objetos. Outro aspecto importante é a portabilidade: clientes devem ser capazes de trabalhar em qualquer ORB que suporte o mapeamento de linguagem existente sem qualquer alteração no código fonte.

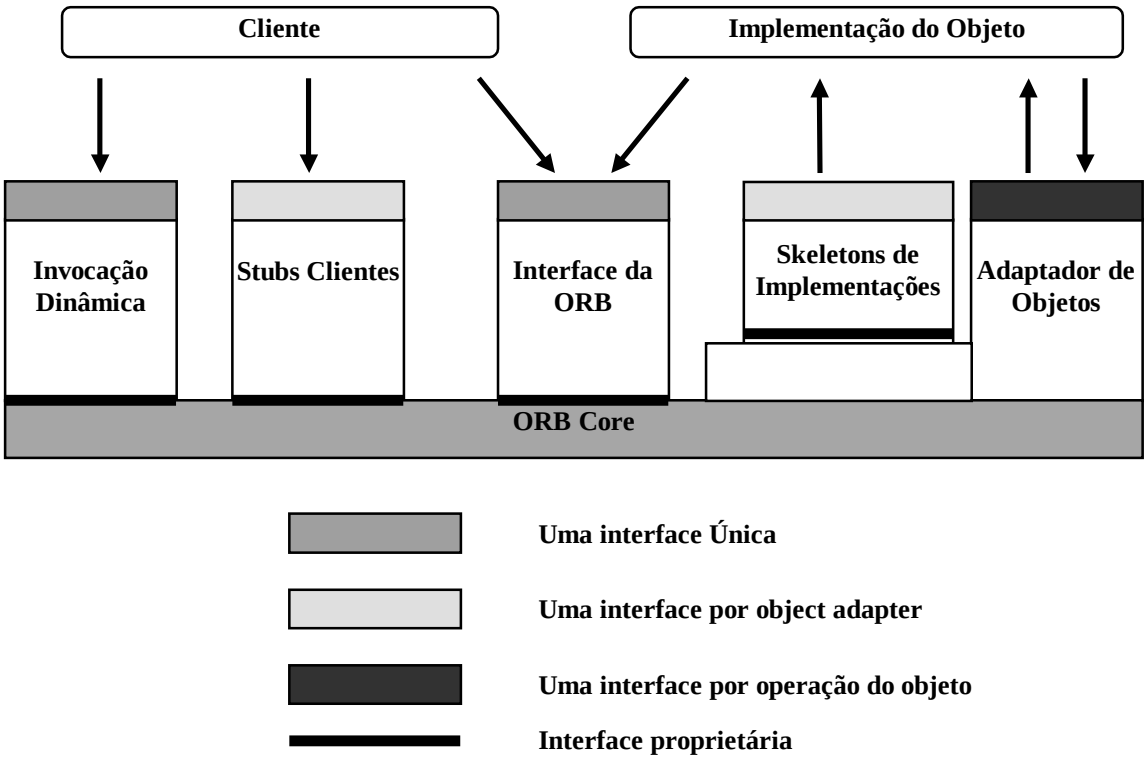


Figura 3.1 - Estrutura de um ORB

- **Implementação de objetos:** Provê a semântica do objeto, definindo dados para a instância do objeto e código para os métodos. As diversas implementações de objetos que podem ser suportadas são: servidores separados, bibliotecas, um programa por método, aplicação encapsulada, SGBDOO, etc. Pode-se suportar diversos estilos de implementação com a definição de adaptadores de objetos adicionais. A portabilidade também é importante: implementações de objetos são portáteis entre quaisquer ORBs que suportem o mapeamento de linguagem adequado. A não dependência das implementações dos objetos em relação aos ORBs se dá através da existência dos Adaptadores de Objetos.
- **Referências a objetos** Informação necessária para especificar um objeto em um ORB, que é dependente do mapeamento da linguagem e da implementação do ORB.
- **Esqueleto da implementação:** É através dele que o ORB faz as chamadas às rotinas existentes na *interface* para os métodos que implementam cada tipo de objeto. Porém, é possível escrever um adaptador de objetos que não se utiliza deste esqueleto para invocar os métodos de implementação.
- **Adaptadores de objetos:** É um caminho básico pelo qual a implementação do objeto acessa os serviços do ORB, tais como: geração e implementação das referências para objetos, invocação de métodos, ativação e desativação dos objetos e suas implementações, registro das implementações, ... Devido a grande diferença existente entre cada tipo de objeto (estilo de implementação, tempo de vida, políticas,...), com os adaptadores de objetos torna-se possível o ORB acessar um grupo de implementações que tenham requisitos semelhantes de uma maneira padrão, com *interfaces* comuns.
- ***Interface* ORB** É comum a todas as implementações de ORB, e contém operações (úteis tanto aos clientes quanto às

implementações) que são comuns a todos os objetos, independentemente da interface do objeto ou do adaptador de objetos. Em função dessa independência, existem muito poucas operações disponíveis na interface ORB. Consiste na *interface* para as funções ORB que não dependem do *Object Adapter* utilizado. Essas operações são as mesmas para todos os ORB's e para todas as implementações de objetos e podem ser utilizadas por clientes dos objetos ou por suas implementações. As operações de criação de listas e determinação de objetos de contexto *default* referenciadas *Dynamic Invocation Interface* constituem também funções do ORB;

- **Repositório de implementações:** Contém informações necessárias para que o ORB localize e ative implementações.
- ***Stubs* de cliente:** Promovem acesso às operações definidas em IDL para um objeto de um modo que seja fácil para um programador que conheça IDL e o mapeamento para a linguagem de programação fazer previsões. As *stubs* fazem chamadas para o ORB utilizando-se de *interfaces* privadas ao núcleo do ORB que está sendo utilizado.
- ***Interface* de Invocação dinâmica:** Permite construção dinâmica para invocação de objetos. Com isso um cliente pode especificar, através de uma seqüência de chamadas, o objeto a ser invocado, a operação a ser executada e o conjunto de parâmetros para a operação. O próprio código do cliente pode obter do repositório de interfaces (ou de outra fonte em tempo de execução) informações sobre a operação e os parâmetros necessários, fornecendo-as para a invocação *Interface* para invocação dinâmica.

Permite criação e invocação dinâmica de pedidos à objetos, sendo capaz de distribuir qualquer pedido para qualquer objeto, através da interpretação em tempo de execução dos parâmetros e identificadores da operação. O resultado semântico obtido é o mesmo que utilizando os *stubs* gerados em C, mas o número de

chamadas que se tornam necessárias para executar uma operação é bem maior do que quando o pedido é "montado" em tempo de compilação.

Na invocação dinâmica os parâmetros são fornecidos como elementos de uma lista, onde cada um deles é uma instância de ***NamedValue*** (descrito a seguir), serão checados em tempo de execução e devem ser fornecidos na mesma ordem em que foram definidos no ***Interface Repository.***

```
typedef unsigned long Flags;

struct NamedValue;
{

   Identifier    name;
   any           argument;
   long          len;
   Flag          argmodes;

};
```

3.3.2.3 Repositório de *interfaces*

Trata-se de um serviço na estrutura de um ORB que provê objetos persistentes que representam a informação IDL em forma disponível em tempo de execução, e pode ser usado pelo ORB para execução dos pedidos, tornando possível que se encontre um objeto cuja *interface* é desconhecida, e determinando que operações são válidas para tais objetos. É o componente do ORB que possibilita a persistência das definições de *interface*, possibilitando distribuição e gerenciamento de uma coleção de objetos relacionados às *interfaces*.

Para que um ORB funcione corretamente ele precisa conhecer a definição dos objetos que ele "manuseia". Isso pode ser feito de duas formas :

- incorporando a informação proceduralmente nas rotinas embutidas;
- acessando dinamicamente o repositório de *interfaces*;

A definição das *interfaces* é mantida no repositório como um conjunto de objetos acessíveis através de um conjunto de definições de *interface* específicas em IDL. A definição de uma *interface* contém as operações que ela suporta, incluindo os tipos dos parâmetros, as exceções e a informação de contexto, se houver. Além disso, é armazenado nesse repositório valores de constantes e ***typecodes***(que são valores que descrevem um tipo em termos estruturais). Este repositório é organizado em módulo, para facilitar a navegação por nome. Os módulos podem conter constantes, definição de tipos, exceções, definição de *interfaces* e outros módulos.

Um ORB pode ter acesso à vários repositórios de *interface*. A implementação de um repositório de *interface* necessita de um mecanismo de persistência para os objetos. Normalmente o tipo de persistência utilizada irá determinar como às definições de *interface* serão distribuídas e/ou replicadas por um domínio de rede. Por exemplo, se for utilizado um sistema de arquivos existirá somente uma cópia do conjunto de *interfaces* mantida em somente uma máquina, por outro lado se for usado um banco de dados orientado à objetos, várias cópias podem ser mantidas distribuídas entre várias máquinas. Além disso, um mecanismo de segurança deve ser adotado para garantir o controle de acesso aos objetos.

Cada *interface* do repositório é mantida como uma coleção de objetos de *interface* que possuem uma estrutura como às definições no repositório de *interfaces*. É possível encontrar uma *interface* no repositório de três formas distintas :

- obtendo a ***InterfaceDef***(que é um objeto do repositório de *interface*) diretamente do ORB;
- navegando pelos módulos;
- localizando uma ***InterfaceDef*** correspondente à um identificador de repositório;

3.3.2.4 *Interface Definition Language* (IDL)

IDL (Linguagem de definição de *Interfaces*) descreve as *interfaces* que são chamadas pelos clientes e fornecidas pelas implementações. Uma *interface* IDL provê a informação necessária para o desenvolvimento dos clientes que se utilizam das operações da *interface*. Os clientes não são escritos em IDL, mas sim em linguagens para a qual tenha sido definido mapeamento dos conceitos IDL. Ela define os tipos dos objetos, especificando suas *interfaces*. Por *interface* entende-se o conjunto de operações e parâmetros para tais operações. Pelas definições IDL, é possível mapear objetos CORBA em diferentes linguagens de programação

A linguagem C é a primeira linguagem para a qual foi estabelecido mapeamento IDL. IDL obedece as mesmas regras léxicas que a linguagem C++, com apenas algumas palavras reservadas a mais. A gramática IDL é um subconjunto da ANSI C++, com construções adicionais para suportar mecanismos de chamadas a operações. Por ser uma linguagem declarativa, IDL não possui nenhuma estrutura algorítmica ou variáveis. Para que se evitasse conflito de nomes da especificação CORBA com os da linguagem de programação, convencionou-se que os primeiros devem ser tratados

como se tivessem definidos em um módulo denominado CORBA. Os nomes usados na *interface* devem ser referenciados, portanto, por seu nome completo (CORBA::<nome>).

As diferenças básicas para a sintaxe C++ são restrições do tipo:

- retorno de uma função é obrigatório e devem ser fornecidos nomes para cada um dos parâmetros formais na declaração de uma operação;
- uma lista de parâmetros vazia não pode ser substituída pela palavra void;
- a declaração dos tipos inteiros quanto ao tamanho deve ser explícita (short ou long) e os modificadores signed e unsigned não são aplicáveis ao tipo char;

Existem exceções padrão que são definidas pelo ORB, as quais podem acontecer em qualquer operação e não precisam ser listadas na expressão *raises*. Alguns exemplos de exceções padrão são **UNKNOWN**, **NO_MEMORY**, **INITIALIZE**, **DATA_CONVERSION**,...

3.3.2.5 Mapeamento para linguagens de programação

O acesso aos objetos do CORBA pode ser diferente para cada linguagem de programação (uma LPOO pode querer visualizar os objetos do CORBA como objetos da própria linguagem). Porém, o mapeamento do CORBA para uma linguagem de programação em particular deve ser independente da implementação de ORB utilizada. A especificação do CORBA para o mapeamento para uma linguagem de programação inclui definição dos tipos de dados específicos da linguagem, das *interfaces* dos procedimentos para acesso aos objetos através do ORB, *interface stub* do cliente, *interface* de invocação dinâmica, esqueleto de implementação

(*implementation skeleton*), adaptadores de objetos e *interface* ORB. Até o momento, a especificação CORBA possui apenas o mapeamento para a linguagem C. Outro aspecto que também é definido no mapeamento para linguagem é a interação entre invocação dos objetos e o fluxo de controle tanto no cliente quanto na implementação (com chamadas síncronas ou assíncronas).

A arquitetura do CORBA é independente da linguagem de programação utilizada para construir os objetos de aplicação (clientes e implementação). Assim para poder utilizar o ORB é necessário que se possibilite aos desenvolvedores maneiras de acessar as funcionalidades do ORB utilizando a linguagem de programação por eles desejada. Para se definir um mapeamento é preciso definir maneiras de expressar na linguagem :

- tipos de dados básicos definidos na IDL;
- tipos de dados construídos definidos na IDL;
- referências à constantes definidas na IDL;
- referências à objetos definidos na IDL;
- chamada de operações incluindo passagem de parâmetros e recebimento de resultados;
- exceções;
- acesso aos atributos;
- assinaturas para às operações definidas no ORB, como *dynamic invocation interface, object adapters* ...

Um mapeamento de linguagem deve permitir ao programador ter acesso a todas as funcionalidades ORB da melhor forma possível para a linguagem em questão. Para suportar a portabilidade de fontes é indispensável que todas as implementações ORB suportem o mesmo mapeamento para uma determinada linguagem.

3.3.3 Requisição e tratamento de pedidos

A figura 3.2 mostra um pedido seguindo do cliente para a implementação do objeto. O cliente é a entidade que deseja realizar uma operação no objeto e a implementação do objeto é o código e os dados que realmente implementam o objeto.

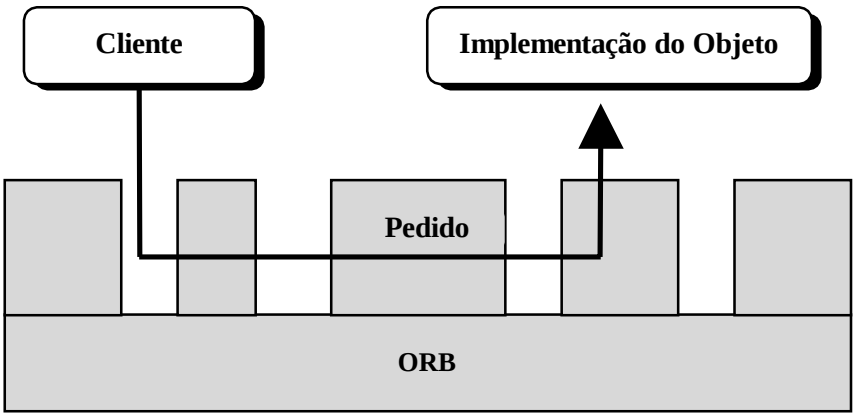


Figura 3.2 - Pedido de cliente via ORB

O ORB é responsável por todos os mecanismos necessários para procurar a implementação do objeto para o pedido, prepará-la para receber o pedido e fazer a comunicação dos dados para elaboração do pedido. A *interface* visível para o cliente é independente da sua localização, da linguagem de programação que o implementa ou de qualquer outro aspecto que não esteja descrito na *interface*.

A estrutura de um ORB é como descrita na figura 3.1. Para fazer um pedido, o cliente pode utilizar a *interface* de invocação dinâmica (que é independente da *interface* do objeto alvo) ou uma *stub* IDL (específica para a *interface* do objeto alvo). A implementação do objeto recebe um pedido como uma chamada através do esqueleto gerado pela IDL (IDL *generated skeleton*). Durante o processamento de um pedido, a implementação do objeto pode chamar o Adaptador de Objetos e o ORB.

Para realizar um pedido, um cliente precisa ter acesso à referência do objeto e conhecimento sobre o tipo do objeto e da operação desejada.

Feito isso, a inicialização do pedido se dá por chamadas a rotinas *stub* específicas ao objeto ou então pela construção dinâmica do pedido, sendo as duas abordagens transparentes ao receptor da mensagem.

Quanto ao atendimento do pedido, o ORB localiza o código de implementação do objeto, transmite parâmetros e transfere o controle para a implementação do objeto através do esqueleto da IDL. Durante a execução do pedido, a implementação do objeto pode requerer serviços do ORB através do adaptador de objetos. Ao final do atendimento, retornam-se o controle e os valores de saída para o cliente.

3.3.4 Estrutura de um cliente

O cliente de um objeto possui uma referência para o objeto. Essa referência é um *token* que pode ser invocado ou passado como parâmetro na chamada de um objeto diferente. A unidade que gerencia a transferência de controle e de dados entre os clientes e as implementações é o ORB. No caso de uma operação não ser executada com sucesso, gera-se uma exceção que deve ser tratada pelo cliente.

A invocação de uma operação é feita da seguinte maneira: o cliente acessa as *stubs* específicas ao tipo do objeto, as quais têm acesso à referência para o objeto, implementadas em uma linguagem de programação, que seguem como parâmetro para o ORB realizar a operação.

A referência para objetos pode ser convertida para uma *string*, armazenada em arquivo, preservada ou comunicada por diferentes meios, e depois transformada de volta em uma referência pelo ORB que gerou a *string*

3.3.5 Estrutura da implementação do objeto

A implementação de objetos define o comportamento de um objeto, procedimentos para ativar e desativar objetos, e se utiliza de outras facilidades para tornar um objeto persistente e controlar seu acesso.

Existe uma interação entre a implementação do objeto e o ORB, implementada via o adaptador de objetos, com vários objetivos: estabelecer a identidade do objeto, criar novos objetos, obtenção de serviços dependentes do ORB.

Quando ocorre um pedido, o núcleo do ORB, juntamente com o adaptador de objetos e o esqueleto da *interface*, realizam uma chamada ao método apropriado para atender o pedido, fornecendo os parâmetros necessários.

Quando da criação de um objeto, o ORB é notificado para que saiba onde se encontra a implementação do novo objeto.

3.3.6 Estrutura do adaptador de objetos

O adaptador de objetos é o meio básico para que uma implementação de objetos possa acessar os serviços do ORB (como por exemplo geração de referências para objetos). Existe uma *interface* pública que é exportada para as implementações dos objetos, e uma privada que fica disponível para o esqueleto. Algumas das funções disponíveis nos adaptadores de objetos, que são executadas usando o núcleo do ORB, são:

- geração e interpretação de referências para objetos;
- invocação de métodos;
- segurança nas interações;
- ativação e desativação de objetos e suas implementações;

- mapeamento das referências para objetos para as correspondentes implementações;
- registro de implementações;

Os adaptadores de objetos estão implicitamente envolvidos nas chamadas aos métodos (serviços como autenticação), embora a *interface* direta seja através dos esqueletos da IDL.

3.3.6.1 Exemplos de adaptadores de objetos

Os adaptadores de objetos são responsáveis por definir a maioria dos serviços do ORB dos quais a implementação dos objetos podem depender. Com eles, é possível que a implementação dos objetos tenha acesso a um serviço que pode não estar implementado pelo núcleo do ORB (caso esteja, o adaptador fornece apenas uma interface para ele, senão o adaptador deve implementá-lo sobre o núcleo do ORB).

- **Básico**: implementações são geralmente programas separados. Pode haver um programa por objeto, ou um programa compartilhado por todas as instâncias de um tipo de objeto. Existe apenas uma pequena quantidade de armazenamento persistente para cada objeto;
- **Biblioteca**: usado no caso de objetos com implementações em biblioteca. Os dados persistentes estão em arquivos, e não existem os mecanismos de ativação e autenticação;
- **Orientado a objetos**: usa uma conexão a um banco de dados orientado a objetos para acessar seus objetos. Não é necessário que se guarde nenhum estado no adaptador de objetos e, além disso, os objetos são registrados implicitamente no ORB, já que o próprio banco de dados é responsável por armazenar dados e métodos dos objetos;

3.3.6.2 O adaptador de objetos básico

A *interface Object Adapter* é a principal utilizada pelas implementações para ter acesso às funções do ORB. O ***Basic Object Adapter*** (BOA) é a *interface* que pretende ser mais disponibilizada e que suporte o maior número de implementações de objetos. Ela inclui *interfaces* para gerar referências à objetos, registro de implementações formadas por um ou mais programas, ativação de implementações e autenticação de pedidos. além disso ainda fornece um armazenamento persistente limitado mas que pode ser utilizado em conjunto com outro maior.

A maior parte da *interface* do BOA pode ser expressa em IDL, desde que seja para operações no *Object Adapter*. Toda implementação de ORB deve possuir um ***Basic Object Adapter***. Apesar de sua implementação ser dependente do ORB, deve ser possível à uma implementação de um objeto que usa o BOA rodar em qualquer outro ORB que suporte o mapeamento de linguagem usado na implementação das operações.

O BOA utiliza funções do sistema operacional para ativar e comunicar com os programas que implementam um objeto. Com isso obtemos um certo grau de não portabilidade do BOA, pois ele necessita de informações que não são comuns à todos os sistemas. Para resolver este problema, definiu-se o conceito de repositório de implementação que irá armazenar essas informações permitindo assim que cada sistema instale e inicia suas aplicações de acordo com seu sistema.

A forma de conexão do BOA com o ORB, bem como do *skeleton* (parte do BOA responsável pela execução dos métodos) não foi especificado por ser dependente do mapeamento de linguagem utilizado.

Entre as funções executadas pelo BOA podemos citar ativação e desativação de implementações: A ativação de implementação ocorre quando não existe nenhuma implementação de um objeto disponível para

lidar com o pedido feito. Para que isso ocorre é preciso uma coordenação entre o BOA e os programas que contém a implementação. Vamos usar o termo servidor para indicar uma unidade executável separada que pode ser iniciada pelo BOA em um sistema particular. (no caso do UNIX um processo).

Existem quatro políticas de ativação que descrevem regras a serem seguidas quando existem muitos objetos ou implementações ativas. São elas :

- **Servidor compartilhado**: ocorre quando vários objetos de uma mesma implementação compartilham o mesmo servidor. Este servidor ficará ativo até que faça uma chamada para desativá-lo (***deactivate_impl***) e nenhum outro servidor para essa implementação será ativado enquanto houver um ativo.
- **Servidor não compartilhado**: ocorre quando existe somente um objeto ativo de uma dada implementação em um dado momento no servidor. Esse tipo de servidor é útil quando o objeto encapsula uma aplicação ou ele necessita de acesso exclusivo à um recurso. Um novo servidor é ativado cada vez que um pedido é feito à objeto não ativo, mesmo que um servidor com outro objeto de mesma implementação esteja ativo.
- **Servidor por método**: cada chamada de um método é implementado por um novo servidor. Este servidor só existe durante a execução do método. Desta forma é possível existir diversos servidores para um mesmo método, sobre um mesmo objeto de uma mesma implementação.
- **Servidor persistente**: quando o servidor é ativado por "alguém" fora do BOA. O BOA trata esse servidor com um servidor compartilhado.

Geração e interpretação de referências à objetos: Referências à objetos são feitos pelo BOA utilizando o ORB *Core*, quando requisitado por uma implementação. A operação utilizada para criar uma nova referência à objeto é :

```
Object create (
    in ReferenceData        id,
    in InterfaceDef         intf,
    in ImplementationDef    impl);
```

O **id** consiste na identificação da informação e não será modificado. O **intf** é o objeto que especifica o repositório de *interfaces* onde se encontra o conjunto completo de *interfaces* do objeto. O **impl** é objeto que referencia o repositório de implementação a serem usadas pelo objeto.

O BOA não adota nenhum estilo específico de gerenciamento de segurança. Ele garante que para todo método ele irá identificar o "principal" sobre o qual o pedido foi feito. O significado do "principal" depende do ambiente de segurança sobre o qual a implementação está rodando. A decisão de permitir ou não a execução de uma operação é da implementação, que normalmente associa direitos de acesso com objetos e "principais" e examina se eles podem executar a operação.

A persistência dos objetos é obtida em conjunto pelo BOA e o ORB Core, possibilitando assim um cliente acessar um objeto à qualquer momento, mesmo que a implementação tenha sido desativada ou o sistema re-inicializado.

Capítulo 4

PLATAFORMAS DE DISTRIBUIÇÃO

"A curiosidade não passa de vaidade. Na maior parte das vezes, apenas queremos saber para falar disso."

Blaise Pascal

4.1 Introdução

Plataformas de Distribuição são ambientes que disponibilizam ferramentas facilitadoras para o desenvolvimento de aplicações em sistemas distribuídos. A fim de se evitar que seja implementado um grupo de serviços de suporte personalizado para cada aplicação, uma estrutura é criada disponibilizando serviços comuns e as aplicações são construídas sobre ela [Rodrigues 96] .

Neste capítulo são apresentadas três plataformas de distribuição. Duas delas, ANSAware e ORBeline, foram utilizadas nas implementações que validam a proposta do ambiente ÁBACO. A terceira, a plataforma DCE é aqui descrita devido sua importância arquitetural, resultado de um consórcio de grandes empresas fornecedoras de soluções e serviços (OSF - *Open Software Foundation*).

4.2 A plataforma DCE

A principal motivação para o surgimento do DCE (*Distributed Computing Environment*) foi a necessidade de uma infra-estrutura de suporte geral para construção de aplicações distribuídas independente de fornecedor. Entretanto, o objetivo da plataforma é congregar as melhores soluções existentes, e uma vez integradas, passam a fazer parte da plataforma como solução padrão. O DCE é portanto a composição de soluções de diversos fornecedores que fazem parte do OSF, bem como de universidades. A organização emite solicitações chamadas de RFT's (*Request for Technology*) onde os problemas são descritos em busca de soluções tais como mecanismos de comunicação entre partes da aplicação, forma como são encontradas, etc. Alguns dos principais serviços são :

- Suporte a RPC;
- Serviços de diretórios;
- Serviço de segurança;
- Sistemas de arquivo distribuído.

4.2.1 Componentes do DCE

4.2.1.1 RPC

O RPC é o mecanismo clássico de chamada de procedimento remoto, utilizado na construção de plataformas distribuídas. O processo chamador (cliente) é separado do processo chamado remotamente (servidor) e tudo se passa como se a chamada fosse local e de forma transparente.

Tanto o cliente quanto o servidor precisam concordar como a chamada será feita, o qual é declarado por meio de uma linguagem de definição de *interface* denominada de IDL (*Interface Definition Language*). Ao chamar o procedimento que o cliente invoca, o texto do procedimento não está presente naquele processo. Porém, o cliente precisa invocar alguma coisa quando ele faz a chamada. O que é realmente invocado é um cliente *stub*.

A finalidade do *stub* é converter os parâmetros para uma forma adequada para transmissão na rede. Essa operação é conhecida como *marshaling* e *unmarshaling*, sendo que um ou mais pacotes são enviados ao servidor *stub* que por sua vez desmonta o pacote no formato adequado para aquele sistema e realiza a chamada. O resultado retorna também *stubs*. Após realizado o procedimento, o controle retorna o *stub* cliente como se fosse um procedimento normal. Para que esse mecanismo seja possível, rotinas são ligadas de um lado e de outro conhecida como *RPC runtime* e provêem um número básico de serviços. As plataformas fornecem compiladores que facilitam os serviços. Ao invés do programador

escrever o s*tub*, é mais conveniente usar o compilador IDL que gera os *stubs* cliente e servidor apropriados para aquela *interface*, bem como os cabeçalhos de arquivo a serem incorporados para os usuários daqueles *stubs*. Uma vez gerados, é incorporado ao código cliente e servidor conjuntamente com a biblioteca para criar uma aplicação completa.

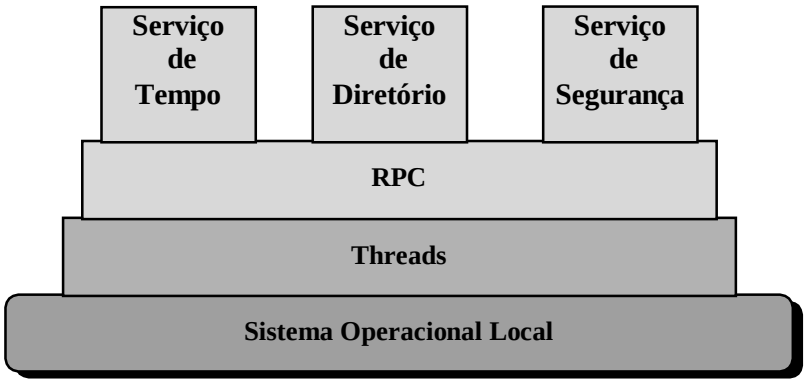


Figura 4.1 - Componentes do DCE

O ambiente DCE inclui dois tipos de RCP. Um que trabalha sobre protocolos de transporte não orientada a conexão, por exemplo UDP, e o outro para protocolos orientados a conexão, como por exemplo TCP. Na figura 4.1, estão mostrados os diversos componentes do DCE, os quais são descritos a seguir :

4.2.1.2 Threads

O uso de *thread* vem suprir uma deficiência de RCP quanto a necessidade de bloquear o chamador até que o procedimento se complete, exatamente como ocorre em chamadas locais. Essa restrição é aceitável num ambiente distribuído em que o paralelismo é fundamental.

O *thread* permite que um processo tenha mais de que um único fluxo de controle. Assim, para os cliente, cada chamada a um procedimento remoto bloqueia somente o *thread* chamador enquanto os outros

continuarão executando. No lado do servidor, significa que um único processo servidor serve a vários processos clientes cada um com seu próprio *thread* ao invés de forçar um cliente esperar até que o pedido anterior seja completado.

4.2.1.3 Serviços de diretório e de segurança

O modulo cliente/servidor requer que numa aplicação sejam definidos quais são os processos clientes e quais são os servidores, bem como seus relacionamentos. Entretanto, é preciso saber de que forma os clientes irão encontrar com precisão os servidores apropriados. O acesso ao servidor é feito por meio de informações de acesso que estão num diretório e foram lá armazenados pelos servidores ou por um administrador da rede. Uma chamada via RCP é feita conhecendo-se o nome do servidor que será usado para buscar a informação no diretório.

Com o objetivo de tornar essa busca tão eficiente quanto possível, o OSF adotou o sistema e nomeação do DNS para localizar servidores numa célula. Uma célula consiste de um grupo de servidores que executam atividades comuns. Em geral, observa-se que as pesquisas feitas pelos clientes buscam servidores da mesma célula a que pertencem. Quando o servidor deseja disponibilizar as informações, ele a exporta para o servidor de diretório de células. Quando o cliente deseja localizar o servidor, ele importa aquela informação do servidor apropriado. O CDS (*Cell Directory Server*) suporta replicação de informação a nível de diretório, e uma vez copiado, mais de um CDS pode executar ao mesmo tempo por questão de segurança e performance.

O servidor de diretório de células CDS, possui uma hierarquia de diretórios. No primeiro nível, existe um domínio de diretórios chamado *clearinghouse*. Nesta ordem, diretórios, entrada de objetos e *soft links* são

gerados, o qual são apontadores para objetos ou outras entradas presentes no CDS.

Os três tipos básicos de entrada são :

- **Entrada de servidores**

  Contém uma informação de diretório para o servidor. Consiste de endereço de um servidor, do protocolo, interface para RPC, etc.

- **Entrada de grupo**

  Usado quando se tem diferentes máquinas que podem executar o mesmo serviço, possuindo uma ou mais entrada de servidores. Quando o cliente importa a informação, a entrada de servidor referenciada é recuperada da entrada de grupo para aquele serviço.

- **Entrada de Serviço**

  Na entrada de grupo, a entrada de servidor importado pelo cliente é aleatória dentre os que fazem parte do grupo. Há situações em que isso é indesejável. Dessa forma, um perfil pode ser definido, Por exemplo, em termos de ordem de acesso, para o servidor mais próximo do cliente que solicita o serviço.

- **Segurança**

  É o conjunto de mecanismos que toda plataforma deve ter para que, de dentro de uma ambiente multi-usuário, seja possível construir aplicações distribuídas de forma segura. Outros serviços são oferecidos, tais como :

  - Autenticação

  O usuário deve provar quem ele diz que é;

  - Integridade

  Evitar modificação de dados em trânsito;

  - Privacidade

  Evitar que os dados trocados entre clientes e servidores sejam interceptadas e conhecido por terceiros.

4.3 A plataforma ANSAware

O ANSAware é uma implementação do modelo de engenharia que foi projetado para suportar o ambiente do modelo computacional do ANSA.

4.3.1 Modelo computacional

4.3.1.1 Serviços

A estrutura básica do ANSA são os **serviços**. Um serviço é uma função manipuladora de informação - processa, armazena ou transfere. Objetos que usam os serviços são chamados **clientes**; objetos que fornecem serviços, são chamados **servidores**. O modelo computacional do ANSA permite que objetos possam ser ao mesmo tempo clientes e servidores de muitos serviços simultaneamente. Os serviços são ofertados em uma ***interface***.

Os serviços são divididos em **serviços de aplicação** que são específicos para uma tarefa a ser executada para o sistema e **serviços de arquitetura** que são genéricos para uma grande numero de tarefas.

4.3.1.2 Transparências

Uma transparência é um meio de se mascarar um aspecto particular da complexidade da programação distribuída. As transparências implementadas pelo ANSAware são:

- **acesso** - permite um estilo uniforme de interação entre objetos sem se importar com localização relativa sua construção ou seu ambiente;

- **localização** - permite interação com um objeto sem o conhecimento de sua localização física.

4.3.1.3 Serviços de arquitetura

Serviços de arquitetura fornecem mecanismos consistentes para funções tais como nomeação, controle de acesso, busca de serviços e gerenciamento dentro de um sistema distribuído. Um exemplo de tais mecanismos é o ***trading***; que permite clientes encontrar servidores dinamicamente via um sistema de diretório utilizada para a gravação e determinação da disponibilidade de serviços. O serviço de arquitetura que fornece o mecanismo de trading no ANSAware é o objeto chamado ***Trader***.

4.3.1.4 Referências de *interfaces*

Computacionalmente, objetos comunicam-se através de passagem de **referência de *interfaces***, que são entidades que se referem a instâncias de uma *interface*; a posse de uma referência de *interface* por parte do cliente, permite a invocação de um serviço fornecido naquela *interface* por um servidor.

4.3.2 Modelo de engenharia

O modelo de engenharia e projetado para suportar o modelo computacional sobre múltiplos modelos tecnológicos. Ele é construído pelo mapeamento dos objetos do modelo computacional em objetos do modelo de engenharia.

4.3.2.1 Nós, núcleo e cápsulas

O termo **nó** é tipicamente utilizado para se referir a um simples computador ou uma *workstation*, podendo também ser aplicado a uma rede de computadores gerenciada por um sistema operacional distribuído.

Os recursos de um nó são gerenciados por um objeto de engenharia chamado de **núcleo**. O serviço fornecido pelo núcleo é tornar os recursos de um nó e construir um ambiente distribuído básico independente dos sistemas operacionais, computadores e redes. Uma **cápsula** (figura 4.2) é uma unidade de operação autônoma dentro do ANSAware. O núcleo fornece cápsulas com as seguintes capacidades:

- encapsulamento;
- provisão de atividades concorrentes, e ordenação e sincronização dessas atividades dentro dessa cápsula;
- comunicação com outras cápsulas.

4.3.2.2 Objetos computacionais e de engenharia

A implementação de um serviço é obtido através da definição de objetos computacionais.

Um objeto computacional deve possuir várias ***interfaces***, cada uma oferecendo o mesmo ou diferentes conjunto de **operações**. Um objeto computacional compilado é chamado de objeto de engenharia.

4.3.2.3 Serviços de transparência

Objetos de engenharia interagem entre si através do núcleo. Serviços de transparência são adotados pela cápsula. Um objeto de engenharia pode utilizar vários serviços de transparências.

4.3.2.4 Protocolos

A especificação do núcleo inclui um serviço de definição do protocolo de comunicação entre os núcleos. A implementação de serviços de

comunicação é realizada numa estrutura de três camadas. A camada do topo é a camada de **serviço de seção**, que fornece diálogo e estruturas de sincronização. A camada do meio fornece um **protocolo de execução**, a camada mais baixa é a camada de **serviço de passagem de mensagem (MPS)**, que fornece um serviço de transporte entre os núcleos.

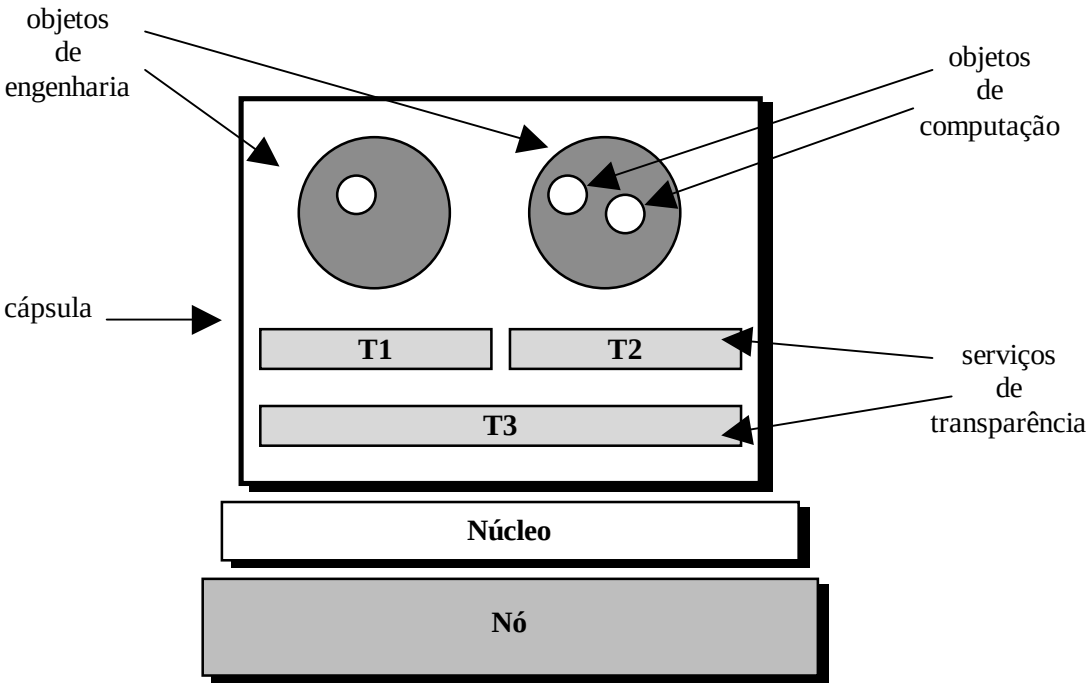


Figura 4.2 - Cápsula ANSAware

O protocolo de execução fornecido pelo ANSAware é chamado **REX** *(Remote EXecution)*

Transformando objetos computacionais em objetos de engenharia

O ANSAware fornece dois compiladores para transformar objetos computacionais em objetos de engenharia; o primeiro é responsável pela geração dos serviços de transparências de acesso (chamados *stubs*), enquanto o segundo é responsável pela tradução das interações entre clientes e os serviços.

Os objetos computacionais especificam suas interfaces de serviços usando uma ***Interface Definition Language*** **(IDL)**; Um compilador,

chamado **stubc** é fornecido para gerar automaticamente o código dos *stubs*. A linguagem **PREPC** fornece meio de invocação de interfaces. Um compilador chamado **prepc** traduz os comandos PREPC em chamadas das rotinas *stub* geradas pelo **stubc**.

4.3.2.5 Estrutura de uma rede de nós ANSAware

O ANSA define serviços que fornecem uma infraestrutura de gerenciamento para uma rede de objetos distribuídos; os seguintes serviços são implementados no ANSAware:

- o serviço de trading;
- o serviço de factory;
- o serviço de notificação.

- **O Trader**

O serviço de *trading* (figura 4.3) contém operações que permitem objetos de engenharia registrar seus serviços **(exportar)** e procurar por serviços que eles pretendem utilizar **(importar).**

- **Factory**

O ANSA reconhece a necessidade de se criar dinamicamente objetos de engenharia para fornecer serviços particulares. O *serviço de factory* é o meio pelo qual isso é realizado.

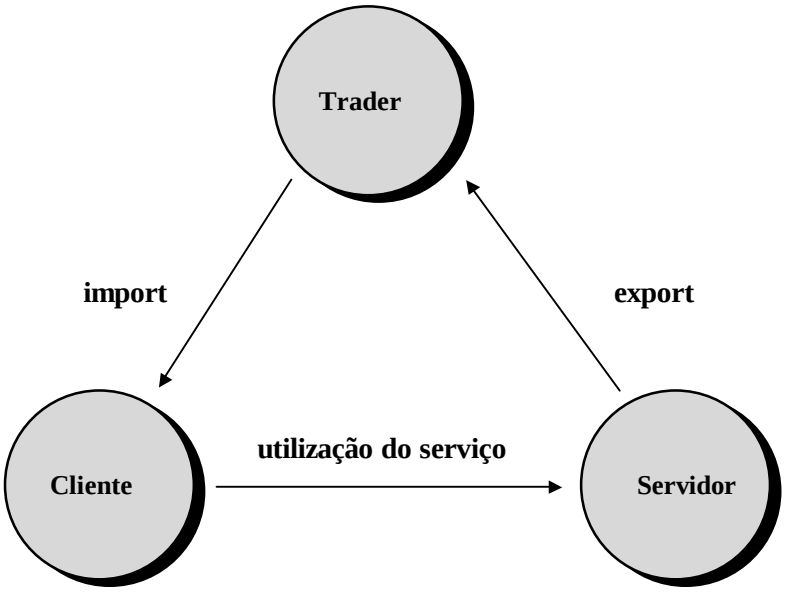


Figura 4.3 - O serviço de trading do ANSAware

- **Serviço de notificação**

O serviço de notificação fornece um meio pelo qual objetos de engenharia registram seu interesse pela final de outro objeto de engenharia.

O serviço de notificação consiste de um simples servidor que mantém todos os estados representado o *status* dos objetos de engenharia e todos os interesses expressados por aqueles objetos. Para detectar a finalização de objetos cada nó deve executar uma instância do ***Grip Reaper service*** que monitora o status de todas as cápsulas que estão em execução no nó.

4.4 A plataforma ORBeline

O ORBeline [Orbeline 94] é uma implementação completa da especificação CORBA. Essa implementação fornece uma estrutura de comunicação que permite o desenvolvimento de aplicações distribuídas orientadas a objetos utilizando a linguagem de programação C++. ORBeline age como uma ferramenta permitindo desenvolvedores criarem e manterem aplicações distribuídas complexas em redes heterogêneas. As características descritas a seguir são fornecidas pelo ORBeline.

4.4.1 Compilador IDL

O ORBeline fornece um compilador IDL que implementa a especificação completa da linguagem de definição de *interfaces* do OMG. O mapeamento de IDL para C++ fornecido pela ORBeline é uma especificação da especificação do OMG. O código C++ gerado pelo compilador de IDL do ORBeline é orientado a objetos e de fácil compreensão.

Dado um arquivo com uma *interface* IDL especificada como entrada, o compilador IDL do ORBeline gera automaticamente as classes C++ para o lado cliente e para o lado servidor e os *stubs* e *skeletons* para cada método da *interface*. Além disso, o compilador IDL gera métodos de empacotamento e desempacotamento de parâmetros associado com cada método.

4.4.2 O *Smart Agent*

O *Smart Agent* do ORBeline fica a par de todas as implementações que estão ativas e dos objetos que podem ser ativados. Quando um pedido é realizado à um determinado objeto, o *Smart Agent* determina se o objeto está ativo, e senão, ele determina que *deamon* de ativação é necessário para ativar o objeto, retornando essa informação para o lado cliente (figura 4.4).

Os objetos do ORBeline são automaticamente registrados com o *Smart Agent* quando são criados pelas por ocasião de chamadas de clientes. O *Smart Agent* pode ser executado em qualquer ponto da rede. Os processos ORBeline encontram esse objeto automaticamente.

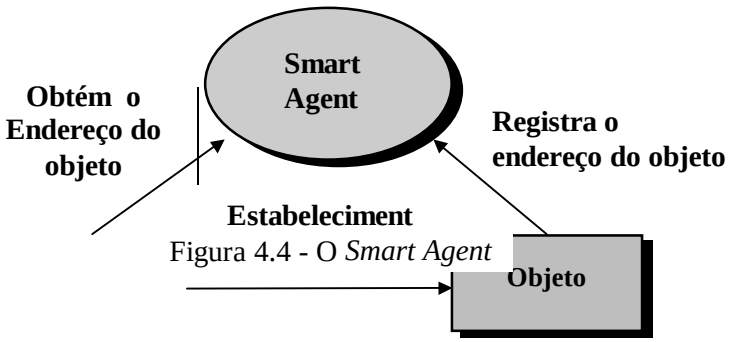


Figura 4.4 - O *Smart Agent*

4.4.3 Tolerância a falhas

O *Smart Agent* mantém registro de todos os objetos e de seus clientes. Se uma conexão de rede for perdida ou um objeto cliente ou servidor parar de executar, o *Smart Agent* será notificado, e recuperará a conexão. Ou poderá utilizar uma réplica do objeto para continuar a comunicação (figura 4.5).

4.4.4 *Smart Binding*

O *Smart Binding* é utilizado para determinar que tipo de mecanismo de transporte deve ser utilizado para se obter um performance ótima. Se o objeto e o cliente residem no mesmo processo, o ORBeline ultrapassa a ORB e faz uma chamada direta ao método desse objeto. Se for determinado que um objeto está em processos diferentes residindo no mesmo nó que o processo cliente, a memória compartilhada é utilizada como mecanismo de comunicação entre o cliente e o servidor.

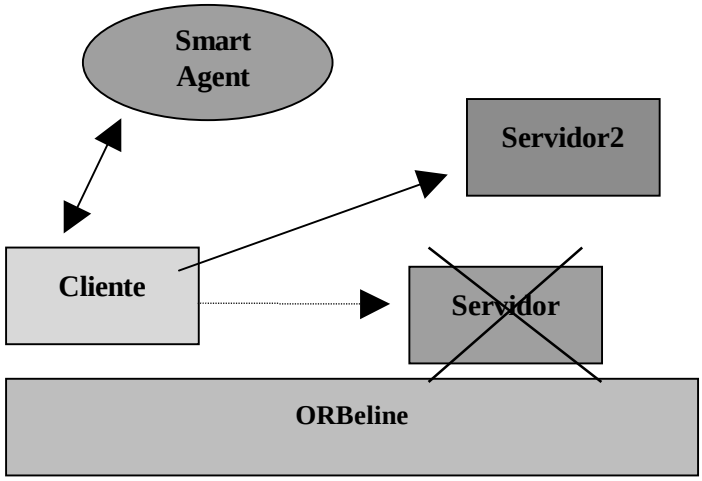


Figura 4.5 - Ativação de réplica

4.4.5 Ativação de objetos

O ORBeline suporta a ativação de objetos e de implementações. Os objetos são ativados quando são invocados. É também possível se iniciar objetos manualmente.

4.4.6 Repositório de *interfaces*

O ORBeline fornece uma implementação completa do repositório de *interfaces* especificado pelo OMG. Esse repositório de *interfaces* implementado como um servidor e pode ser ativado manualmente ou sendo ativado automaticamente.

4.4.7 *Interface* de invocação dinâmica

A implementação da *interface* de invocação dinâmica (*Dynamic Invocation Interface- DII*), é fortemente integrada com o repositório de *interfaces*. Essa integração permite que o ORBeline execute checagem de tipos em tempo de execução no lado cliente, antes do pedido ser executado.

4.4.8 Repositório de implementação

O ORBeline fornece um completo repositório de implementação. Instâncias do repositório de implementação são gerenciadas pelos *deamons* de ativação. Esses repositórios podem ser consultados para se obter informações relativas à implementação de objetos.

4.4.9 Suporte à replicação

Múltiplas instâncias de um objeto podem ser inicializadas de uma só vez. O cliente pode se conectar a uma dessas em tempo de conexão. Se caso haja uma falha em uma dessas instâncias o cliente é reconectado a uma outra.

PARTE II

O Paradigma de Configuração

- **Capítulo 5: Estrutura de *Software* Distribuído**
- **Capítulo 6: Configuração em Sistemas Distribuídos**
- **Capítulo 7: Programação Orientada à Configuração**

Capítulo 5

Estrutura de *Software* Distribuído

> ***"A ciência? Ao fim e ao cabo, o que é ela senão uma*** *longa e sistemática curiosidade?"*
>
> **André Maurois**

5.1 Introdução

As características dos sistemas distribuídos imprimem uma certa complexidade ao desenvolvimento de aplicações. Concorrência, sincronização e comunicação de componentes distribuídos podem tornar impossível esse desenvolvimento se os mesmos não forem bem estruturados e controlados.

A modularidade é a característica mais importante a ser implementada nos componentes que formam uma aplicação distribuída. Desse modo, uma aplicação pode ser escrita com um conjunto cooperante de componentes de software onde cada qual é um simples processo seqüencial.

Na sessão 5.2 serão descritas as estruturas e características que os componentes de qualquer aplicação distribuída devem possuir. Na sessão 5.3 serão tratadas as maneiras como esses componentes podem ser interconectados para formar o sistema: *interfaces*, padrões de conexão e padrões de transação.

5.2 Componentes de *software*

Um sistema distribuído é construído a partir de componentes de *softwares* interconectados [Kramer 90]. Esses componentes se comunicam através de passagem de mensagens de modo a cooperar e coordenar suas ações.

Cada componente possui uma *interface* por onde é realizada a comunicação com os outros componentes a ele conectado. A estrutura lógica do sistema pode ser vista como uma rede de componentes e suas conexões, onde as conexões indicam os caminhos de comunicação entre esses componentes. Para exemplificar, a figura 5.1 mostra um sistema simples de processamento *batch* [Kramer 87] com três componentes - READER, EXECUTOR e PRINTER.

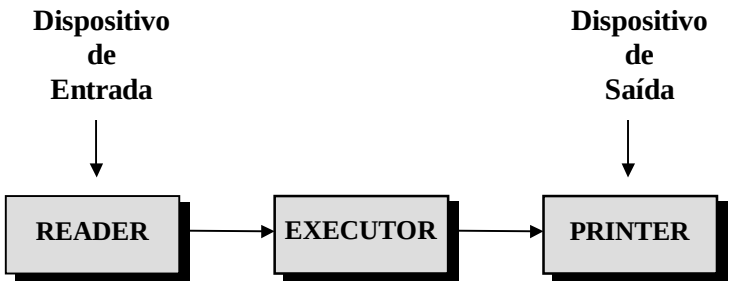


Figura 5.1 - Estrutura lógica do sistema

Um componente é indivisível no sentido de não poder ser decomposto em partes menores que possam ser executadas em computadores físicos diferentes. Desse modo, pode-se dizer que o componente é a menor unidade de distribuição. O conceito de alocação, se refere a ação de definir um computador físico para executar um componente. A estrutura física do sistema, é a rede de computadores interconectados. Essa estrutura deve suportar a alocação dos componentes estabelecida e a estrutura de conexão lógica. Um exemplo do mapeamento para o sistema de processamento *batch* para um rede de duas estações é mostrado na figura 5.2.

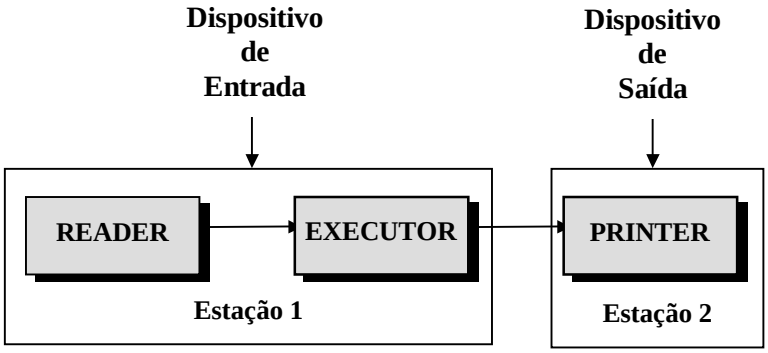


Figura 5. 2 - Mapeamento da estrutura lógica para física

Cada componente encapsula algumas funções ou recursos, normalmente na forma de dados ou dispositivos. Desse modo, um componente deve ser responsável por alguma função específica no sistema ou execução de algum serviço, normalmente a outros componentes. Porém, para que os componentes possam residir em diferentes computadores físicos, é essencial que eles sejam independentes e não compartilhem dados globais. Qualquer dado a ser compartilhado deve ser encapsulado em um componente separado e acessado via *interface*.

Componentes são análogos aos tipos abstratos de dados e aos módulos da programação seqüencial convencional, e processos e monitores na programação concorrente.

- **Estrutura de um Componente**

As principais características de um componente de *software* apto a distribuição são [Kramer 87] :

- ele deve ser a unidade de modularidade de *software* no sistema distribuído;
- ele deve fornecer uma *interface* para comunicação com outros componentes;
- ele deve encapsular os dados ou recursos locais e agir como um domínio protegido - toda interação com recursos não locais deve ser realizada via *interface* do componente;
- ele deve ser capaz de processar dados, tanto seqüencial como concorrentemente.

Execução seqüencial: O componente é um processo seqüencial. Ele deve ser quebrado em um certo número de ações ou procedimentos preservando sua característica de execução seqüencialmente.

Execução concorrente: Esses componentes podem encapsular vários processos que serão executados concorrentemente. Essa estrutura permite que dados locais e procedimentos serem compartilhados por outros processos. Os processos encapsulados comunicam-se através de passagem de mensagem ou via dados compartilhados.

- **Componentes processos x Componentes recursos**

Os componentes que realizam alguma função são chamados componentes processos e os que prestam serviços que serão utilizados pelos componentes processos são chamados componentes recursos. É possível manipular um recursos compartilhado se este for encapsulando dentro de um componente junto com o código para acessá-lo. O código fornece exclusão mútua e define as operações que podem ser realizadas no recurso. Outro processo acessa o recurso enviando mensagens à tarefa encapsulada. Os usuários de um recurso são chamados **clientes**, e os componentes que tratam o recurso ativo são chamados de **servidores**.

- **Tipos e Instâncias de Componentes**

Para que os componentes possam ser projetados, construídos, compilados e testados separadamente, eles devem ser os mais independentes possíveis do resto de sistema. Dessa forma, componentes que possam ser reusados em vários sistemas devem ser produzidos. Para que isso possa ser realizado, são definidos **tipos** de componentes. E qualquer sistema deve ser construído a partir de **instâncias** dos tipos predefinidos.

5.3 Conexão de componentes

No item anterior, discutimos sobre a estrutura básica de um componente e mostramos que um sistema consiste na definição de vários componentes. Nessa sessão, trataremos o problema da conexão desses componentes. Uma conexão é uma associação entre o emissor de uma informação com o receptor dessa informação. A maneira pela qual essa conexão é realizada, depende da *interface* dos componentes, do padrão de conexão e da convenção de nomes que determina a maneira pela qual os componentes farão referencias uns aos outros.

5.3.1 *Interfaces*

Os componentes são unidades independentes que interagem entre si. A definição de uma *interface* entre esses componentes, é a melhor forma de se realizar a conexão entre eles. Essa *interface* deve descrever a forma de como as mensagens devem ser trocadas pelos componentes e as potenciais origens e destinos. As linguagens que fornecem comunicação entre componentes, associam um tipo de dado com a mensagem. Esse tipo fornece uma informação estrutural de como a mensagem deve ser interpretada, de modo que as mensagens enviadas e as recebidas devem ser compatíveis.

As *interfaces* de componentes que utilizam chamadas de procedimentos, devem ter os tipos dos parâmetros (valores e resultados) e os nomes dos clientes e servidores conectados. Algumas plataformas de distribuição que utilizam esse modelo, fornecem uma linguagem como a qual a *interface* entre os componentes podem ser definidas esse linguagem é conhecida como *Interface Definition Language (IDL)*.

5.3.2 Padrões de conexão

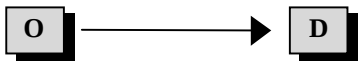
A associação entre clientes e servidores para o propósito de comunicação, é chamada conexão. Essa conexão é uma ligação lógica entre todos os possíveis participantes da comunicação. Cada conexão é um canal de comunicação entre emissores e receptores. Por exemplo:

SEND Mensagem ***TO*** ?? Especifica uma conexão para um ou mais receptores

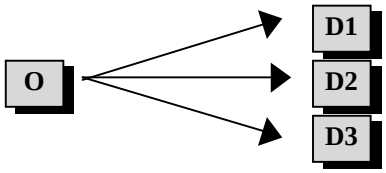
RECEIVE Mensagem ***FROM*** ?? Especifica uma conexão a partir de um ou mais emissores

Podemos resumir os padrões de conexão da seguinte forma:

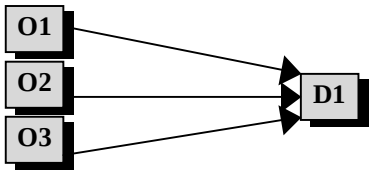
- conexão um-para-um (1-1): realizada entre dois componentes específicos.



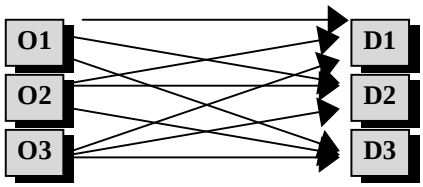
- conexão um-para-muitos (1-m): realizada entre um componente origem e vários componentes destino.



- conexão muitos-para-um (m-1): realizada entre vários componentes origem e um único componente destino.



- conexão muitos-para-muitos (m-n): realizada entre vários componentes origem e vários componentes destino.



Conexões podem ser estáticas ou dinâmicas. As conexões estáticas são aquelas realizadas em tempo de compilação ou em tempo de carga e não podem ser modificadas. Cconexões dinâmicas podem ser criadas, apagadas ou modificadas em tempo de execução.

5.3.3 Nomeação

Nomeação pode ser definida como sendo a técnica de se identificar os componentes envolvidos em uma comunicação, a fim de se estabelecer uma conexão. Podemos classificar a técnica de nomeação como sendo **direta** ou **indireta**. A nomeação deve fornecer um mecanismo suficiente para se identificar um componente unicamente. Esses nomes devem ser transformados em endereços que serão utilizados pelo sistema de comunicação para a transmissão da mensagem.

- **Nomeação direta**

Neste tipo de nomeação, um componente referencia um outro componente diretamente pelo nome. Desse modo, se um processo desejar enviar uma mensagem para um ou mais processos, deve explicitamente identificá-los pelos nomes:

SEND Mensagem ***TO*** Processo(s)

Do mesmo modo o receptor deve referenciar os emissores diretamente pelo nome se desejar receber uma mensagem:

RECEIVE Mensagem ***FROM*** Processo(s)

A técnica de nomeação direta tem algumas desvantagens. Ele esconde e dispersa a conexão dos componentes e a utilização de *interfaces*. Os tipos das mensagens que serão enviadas ou recebidas e a origem ou destino dessas mensagens, devem ser descritas diretamente no código dos componentes, tornando a conexão desses componentes parte de sua própria definição, não permitindo que esses componentes possam ser reusados.

- **Nomeação Indireta**

Ao invés de referenciar um componente diretamente pelo nome, podemos utilizar um nome de uma "porta" local para fazer essa referencia. A mensagem e enviada para uma porta local de saída pelo emissor e é recebida por uma porta de entrada local do receptor. A ligação entre os nomes locais e a origem ou destino são executados separadamente (figura 5.3). Em CONIC [Kramer 83], é utilizada a seguinte sintaxe:

Emissor:

SEND mensagem ***TO*** **porta_de_saída**

Receptor:

RECEIVE mensagem ***FROM*** **porta_de_entrada**

Configuração:

LINK **Emissor.porta_de_saída** ***TO*** **Receptor.porta_de_entrada**

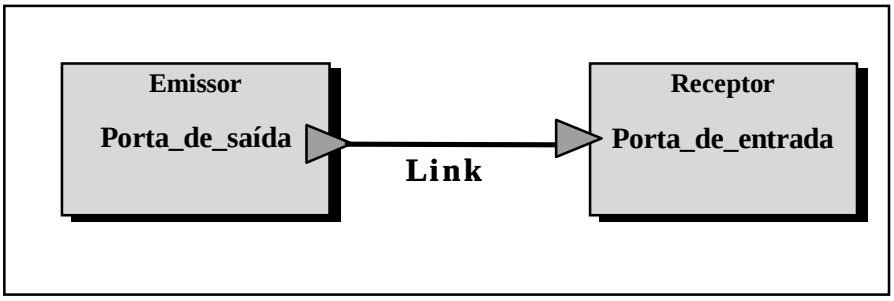


Figura 5.3 - Conexão com nomeação indireta

Esse modelo é muito flexível, fazendo com que o usuário possa programar os componentes sem ter a necessidade de se deter com a ligação entre esses componentes. Essa associação (*binding*) entre as portas de entrada e as portas de saída devem ocorrem em um segundo momento, durante a geração do sistema ou durante a sua execução. Essas portas devem possuir tipos permitindo o compilador checar se as mensagens enviadas ou recebidas em uma porta correspondem com o tipo da porta.

As *interfaces* são fornecidas explicitamente pelas portas e seus tipos de dados associados, e as conexões são explicitamente fornecidas por comandos de ligação. Esses comandos de ligação devem ser mantidos em um arquivo para dar a estrutura lógica do sistema. A principal desvantagem é o número extra de nomes indiretos envolvidos nas transações. Os nomes extras para as portas pode levar a uma proliferação de nomes. A tabela 5.1 mostra uma comparação entre os mecanismos de nomeação direta e indireta.

Nomeação Direta	Nomeação Indireta
A mudança no nome de um processo requer a localização de referência à aquele nome e um recompilação	Referências são à nome locais, portanto bastante modular
Os relacionamentos entre os processos são definidos em tempo de compilação	Ligações entre porta de entrada e portas de saída podem ser realizadas a qualquer tempo
A estrutura lógica das interconexões são implícitas	A estrutura lógica das interconexões são explícitas, em termos de ligações entre as portas de entrada e as portas de saída
Não são requeridos nenhum nome adicional, além dos nomes dos processos	Há uma proliferação de nomes, utilizados pelos processos e pelas portas
Compilação separada exige ao compilador acessar nomes no resto do sistema	O compilador necessita apenas dos tipos das mensagem apenas. O *linker* necessita as informações de Configuração

Tabela 5.1 - Nomeação direta x nomeação indireta

- ***Mailboxes***

Alguns sistemas fornecem um objeto de comunicação tipado, chamado *mailbox* ou *canal*, independente do emissor ou receptor. Os *mailboxes* devem possuir um nome de identificação global ou único. Componentes emissores enviam mensagens para o *mailbox* e componentes receptores recebem mensagens desses.

SEND mensagem ***TO*** **mailbox_X**

RECEIVE mensagem ***FROM*** **mailbox_X**

ou

CALL **mailbox_X** (parâmetros)

ACCEPT **mailbox_X** (***IN*** parâmetros ***OUT*** parâmetros)

Nos sistemas que consistem em componentes e *mailboxes*, as conexões são indicadas pela utilização compartilhada dos mesmos *mailboxes*.

Esse modelo é menos flexível que o mecanismo de nomeação indireta puro, porém os componentes são mais independentes individualmente.

Uma das maiores vantagens dos *mailboxes* é que eles suportam convenientemente conexões do tipo muitos-para-muitos.

Capítulo 6

Configuração em Sistemas Distribuídos

"A imaginação é uma das mais altas prerrogativas do homem"

Charles Darwin

6.1 Introdução

Para se construir uma aplicação distribuída segue-se basicamente dois enfoques: no primeiro, o sistema é implementado todo de uma só vez, compilado e posto em funcionamento, tornando-se inviável se levarmos em consideração a robustez, que é uma característica básica de algumas aplicações distribuídas. No segundo enfoque, o sistema é dividido em módulos, encarregados de tarefas bem definidas, de modo que a aplicação distribuída possa ser formada pela interação entre esses módulos, cada módulo será compilado separadamente e interligados através de uma linguagem de configuração. Este último enfoque tem inúmeras vantagens sobre o primeiro. No entanto, mecanismos que facilitem alterar a configuração do sistema como um todo, seriam altamente desejáveis em sistemas distribuídos.

Neste capítulo o paradigma de configuração é descrito em detalhes, juntamente com os conceitos de configuração estática e configuração dinâmica. São também apresentados os principais ambientes que implementam este paradigma: CONIC, RIO, REX e CL.

6.2 Modelos de configuração

Os mecanismos de Configuração surgiram na computação distribuída para proporcionar uma maior flexibilidade no desenvolvimento de aplicações. Normalmente, as aplicações distribuídas são robustas e complexas, sendo que em alguns casos não se pode prever todos os aspectos

relevantes ao sistema em sua fase de concepção, nesse caso algumas alterações devem ser introduzidas durante o seu tempo de vida. Dentre as mais diversas metodologias para desenvolvimento de sistemas, a mais adequada ao desenvolvimento de sistemas distribuídos é a que utiliza a decomposição da aplicação distribuída em módulos menores, essa decomposição deve seguir algumas regras preestabelecidas. Esses módulos são compilados e testados separadamente, só após serem devidamente testados é que as partes devem ser integradas, utilizando um mecanismo de conexão. Sendo assim, devemos ter duas linguagens distintas: a primeira, será utilizada na programação dos módulos. Esses módulos, não devem fazer nenhuma referência direta a módulos externos e nem à variáveis externas. A segunda linguagem, será utilizada na configuração do sistema, fazendo a conexão entre os módulos através das portas de comunicação.

Existem basicamente duas maneiras de se construir uma aplicação distribuída, no que se diz respeito à configuração: a primeira maneira, utiliza o modelo de configuração estático, produzindo um sistema distribuído estático. A segunda maneira, utiliza o modelo de configuração dinâmico, produzindo um sistema distribuído dinâmico.

6.2.1 Configuração estática

No modelo de Configuração estática, temos dois processos encarregados da configuração do sistema, esses processos são:

- *processo construtor*: tem como principal finalidade gerar, a partir de uma especificação de configuração, as imagens dos módulos para serem alocados nas estações;

- *Pprocesso carregador*: tem como principal função carregar nas estações as imagens dos processos gerados pelo processo construtor.

No modelo de configuração estático, todos os módulos que formam o sistema são configurados de uma só vez, caso seja necessário alguma alteração em tempo de execução, o sistema inteiro será interrompido e depois reconstruído de acordo com a nova especificação. Nesse modelo, não há distinção entre a linguagem de programação dos módulos e a linguagem de configuração, estando ambas incorporadas a uma só linguagem. A figura 6.1 ilustra esse modelo de configuração.

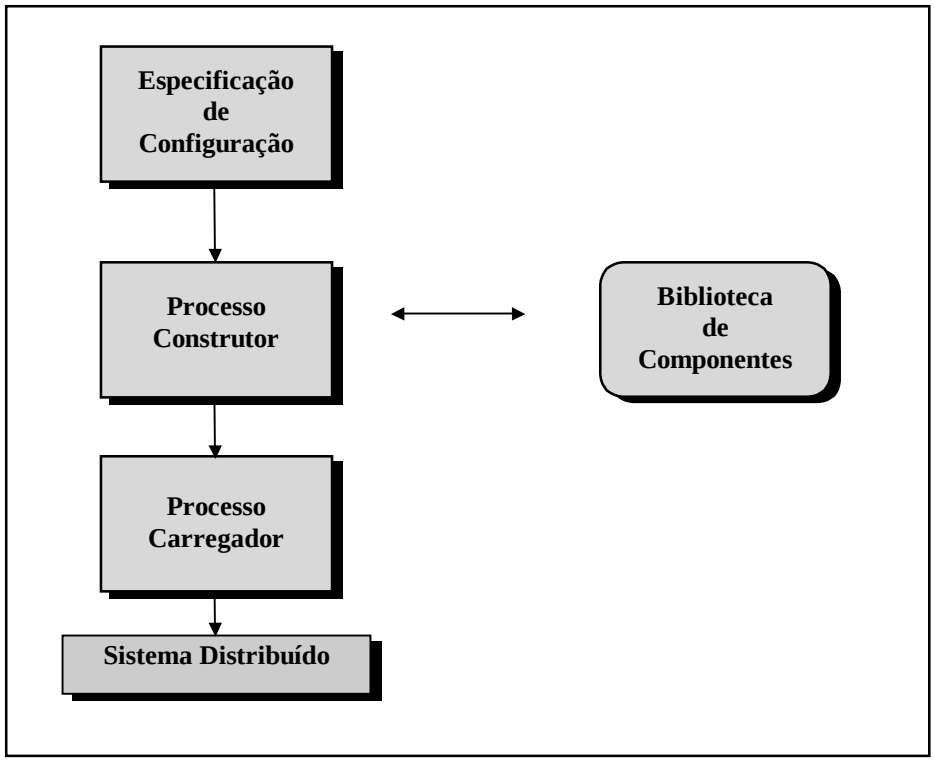


Figura 6.1 - O modelo de Configuração estática

6.2.2 Configuração Dinâmica

No modelo de configuração dinâmica [Kramer 85], não há necessidade de se interromper a execução de todo o sistema para que alguma alteração de configuração possa ser realizada. Nesse modelo, um processo gerenciador de configuração se encarrega da continuidade da operação do sistema, isolando apenas aqueles módulos que serão diretamente afetados pela modificação. Esse modelo é ilustrado pela figura 6.2.

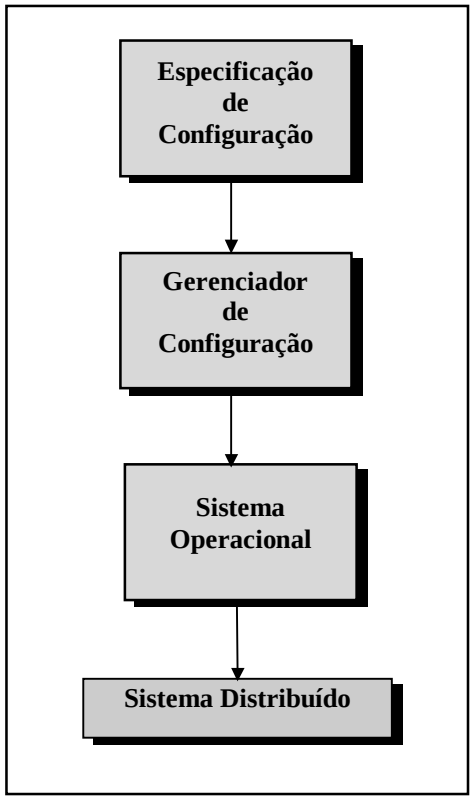


Figura 6.2 - O modelo de Configuração dinâmica

6.3 O ambiente CONIC

A arquitetura do *software* CONIC foi desenvolvida para ser utilizada na produção de grandes sistemas [Kramer 83]. Particularmente

CONIC[9] fornece a flexibilidade necessária aos sistemas para evoluírem e se modificarem devido à alterações em alguns de seus requisitos. Facilitando a incorporação de novas funcionalidades em resposta à modificação do sistema inicial, permitindo a organização dos componentes existentes em resposta às mudanças operacionais.

Como discutido anteriormente, para se projetar sistemas flexíveis, é necessário se decompor o sistemas em componentes que possam ser programados, compilados e testados separadamente. O sistema, seria a conexão de todos os componentes. Essa separação entre a programação dos componentes e a construção do sistema (configuração) é conhecido como '*programming-in-the-small*' e '*programming-in-the-large*' respectivamente [DeRemer 75]. Em CONIC isso se reflete pela utilização de uma linguagem de programação de componentes e uma linguagem de configuração.

6.3.1 Linguagem de programação de módulos

Modularidade é a propriedade chave para se desenvolver aplicações flexíveis. A linguagem de programação de CONIC é baseada em Pascal, que foi estendida para suportar modularidade e passagem de mensagens.

A linguagem permite se definir ***task modules type***, que é um processo seqüencial auto contido. Um ***task module type*** é escrito e compilado independente da configuração em que ele vai ser executado, ou seja há uma independência de configuração. Em tempo de configuração, instâncias dos ***task module types*** são criadas.

Os módulos de CONIC têm uma *interface* bem definida que especifica todas as informações necessárias para a utilização dos módulos no sistema. A interconexão dos módulos é especificada em termos de

[9] O sistema CONIC surgiu a partir de um projeto de pesquisa do ***British National Coal Board*** para investigação da utilização de microcomputadores distribuídos no monitoramento e controle subterrâneo de minas de carvão.

portas. Uma porta de saída (**exitport**) determina a *interface* na qual uma mensagem pode ser inicializada e especifica um nome local, no lugar do nome do destino, e um tipo. Uma porta de entrada (**etryport**) determina a *interface* na qual uma mensagem pode ser recebida e especifica um nome local, em lugar do nome da origem, e um tipo. A ligação entre as portas de entrada e as portas de saída é parte da especificação de configuração e não pode ser realizada pela linguagem de programação de módulos.

Existem duas classes de portas que correspondem às classes de transações. Portas r**equest-reply** são bidirecionais e devem ser declaradas os tipos dos valores a serem usados tanto no **request** como no **reply**. Portas **notify** ou portas de notificação, são unidirecionais, ou seja não requerem uma porta **reply**.

6.3.2 Linguagem de configuração de CONIC

A linguagem de configuração de CONIC é utilizada para se "construir" o sistema a partir de instâncias de módulos (***task modules***). Essa linguagem é utilizada tanta para se montar o sistema inicial como também para inserir modificações no mesmo. Vejamos agora as propriedades essenciais que devem ser suportadas por uma linguagem de configuração.

- **Definição de contexto**

A definição do contexto identifica o conjunto de tipos a partir do qual o sistema é construído, por exemplo.

USE *tControle, tBomba;*

- **Instanciação**

O construtor **create** declara uma instância de um tipo de módulo a ser
criada no sistema.

CREATE controle: tControle;

- **Interconexão**

O comando **link** cria uma conexão de instâncias de módulos através da ligação da porta de saída de um módulo com a porta de entrada de outro. A compatibilidade entre os tipos são checados, de modo que uma porta de saída possa ser ligada a uma porta de entrada do mesmo tipo.

LINK *controle.ctl* ***TO*** *bomba.ctl*

- **Especificações estruturadas de configuração**

O comando **group module** permite definir um tipo de módulo composto de outros tipos de módulos com suas conexões. Esse comando fornece um mecanismo de abstração ao sistema. A estrutura de um **group module** é definida através do uso dos comandos **use, create** e **link** descritos anteriormente. A *interface* do módulo é definita em termos de portas de entrada e portas de saída, de modo que exteriormente não se pode distinguir um **task module** de um **group module**.

6.3.3 Configuração dinâmica

Para muitas aplicações, principalmente as tempo-real, é muito oneroso ou inseguro paralisar toda a aplicação para se fazer uma alteração em um componente. Um sistema CONIC permite se implantar modificações sem que haja a necessidade de se reconstruir o sistema inteiro. A esse mecanismo damos o nome de configuração dinâmica. Com

esse mecanismo fica possível se desenvolver modificações no sistema sem ter que paralisar as partes não afetadas.

Essas modificações são executadas se submetendo a especificação de mudança a um **gerente de configuração** que faz a validação dessa mudança.

- **Mudanças de especificação**

As mudanças de especificação utilizam os comandos descritos na sessão anterior, mas podem também utilizar as funções inversas:

unlink desconecta a porta de saída de um módulo de uma porta de entrada.

delete apaga uma instância de um tipo de módulo do sistema.

remove remove o conhecimento de um tipo de uma especificação de configuração, e é válido apenas depois de todas as instâncias terem sido "deletadas".

- **Gerente de configuração**

Uma mudança de configuração é submetida a um gerente de configuração para que esse possa fazer a validação, traduzi-la em comandos para o que o sistema operacional possa executar a operação de reconfiguração e produzir o novo sistema. O gerente de configuração deve possuir informações do estado atual do sistema.

O gerente de configuração deve ser composto de três partes (figura 6.3):

- um banco de dados descrevendo o sistema;
- o tradutor de especificação;
- o executor de comandos.

O banco de dados de configuração deve possuir as seguintes informações:

- Tipos dos ***tasks module***: O código gerado pelo compilador de módulos;
- Definições de tipos: Os arquivos de definição dos tipos das portas e dos tipos das mensagens;
- Especificação de configuração: A especificação de configuração atual do sistema;
- Configuração física do sistema: Informações sobre as estações conectadas e o seu estado corrente.

O tradutor de especificação valida a especificação de mudança com respeito a disponibilidade de recursos (memória, dispositivos de I/O) como também a compatibilidade de tipos das conexões.

O executor de comandos traduz a especificação de mudança em um conjunto de comandos chamado **lista de ações**. Também é responsável pela atualização da base da dados.

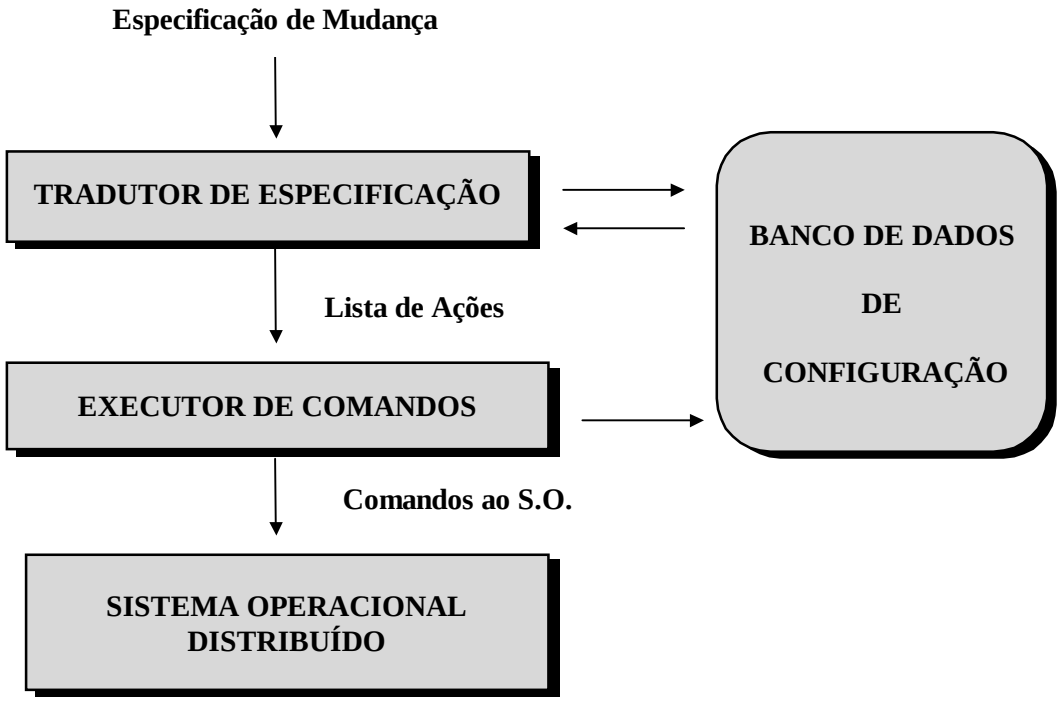


Figura 6.3 - Gerente de configuração de CONIC

6.4 O ambiente do projeto REX

O projeto REX [Kramer 90] , *Reconfigurable and Extensible Parallel and Distributed Systems*, auxiliou no desenvolvimento de uma metodologia integrada e um conjunto de ferramentas de suporte associadas para o desenvolvimento e gerenciamento de sistemas paralelos e distribuídos.

Figura 6.4 - Componente REX

O ponto central do projeto REX é a visão de sistemas paralelos e distribuídos como conjuntos de instâncias de componentes interconectados. componentes são objetos que encapsulam um estado interno e possui uma *interface* definida em *Interface Specification Language* (ISL). A funcionalidade de um componente deve ser implementada em uma linguagem de programação comum, porém, para a interação com outros componentes é utilizado as primitivas de comunicação fornecidas pelo REX. (figura 6.4). Componentes em REX são tipos (classes) do qual uma ou mais instâncias podem ser criadas. O conjunto de instâncias existente em um sistema juntamente com a interconexão são especificados por uma linguagem de configuração - Darwin (figura 6.5).

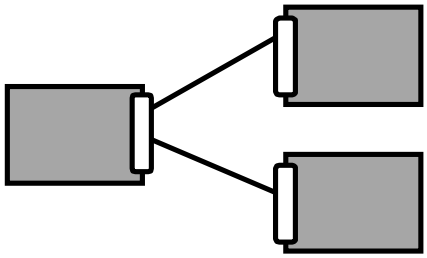


Figura 6.5 - Sistema REX

Adicionalmente, para se escrever toda a estrutura do sistema, Darwin permite sistemas serem estruturados hierarquicamente através da composição de componentes. A *interface* de um componente composto é especificada por ISL de forma idêntica aos componentes (figura 6.6).

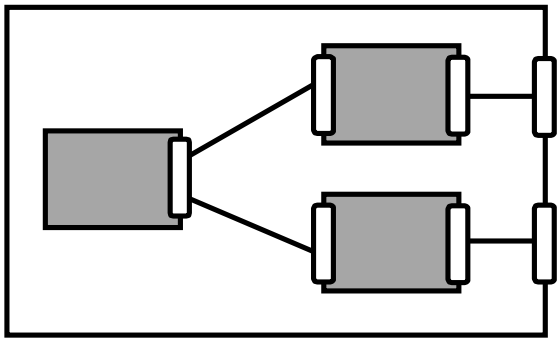


Figura 6.6 - Componente composto em REX

6.4.1 Linguagem de especificação de *interfaces*

A ISL é utilizada para definir os tipos de dados que podem ser transferidos por uma *interface* de componente. Como componentes podem ser implementados em diferentes linguagens de programação, a ISL fornece uma especificação comum para tipos de dados de forma a possibilitar a checagem de tipos entre componentes heterogêneos. Adicionalmente, a ISL deve definir os pontos de interação com outros componentes. Essa independência de contexto permite o reuso do componente em outras aplicações. Estes pontes de interação são as **portas**, que definem quais serviços são fornecidos pelo componente e quais serviços são utilizados por ele.

- **Definição de portas**

Portas representam na *interface* os serviços que são fornecidos e os que são utilizados pelo componente. A declaração:

get = **port** signal **return** char

define uma porta do tipo *get* que fornece um serviço onde um *signal* é a entrada e retorna um valor caractere.

- ***Interfaces***

A descrição de uma *interface* consiste na definição de um conjunto de portas. A figura 6.7 demonstra a descrição de uma *interface*.

ISL permite que a mesma *interface* seja utilizada tanto pelo cliente como para o servidor, para isso é necessário indicar a direção da invocação das portas. A palavra chave **invert** especifica que a porta *error* tem uma direção oposta às outras portas da *interface*.

```
define filedefs = {
  fileid    = int
  data      = [512] char
  length    = int
  errcode   = int16

  fileio = { read:   port fileid  return length, data
             write:  port fileid, length, data return length
             open:   port string (64) return fileid
             close:  port fileid return errcode
           invert
             error:  port errcode }
}
```

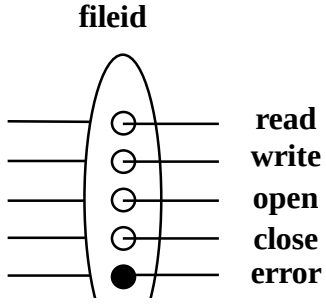


Figura 6.7 - *Interface* descrita em ISL

- **Programação de Componentes**

Os componentes implementam a funcionalidade dos sistemas REX. A sintaxe para a associação de uma *interface* com uma implementação é mostrada na figura 6.8.

```
component fileserver (id : int) = entry fileio + {
use filedefs : fileio

  /* o componente é programado em uma linguagem de
     programação */
}
```

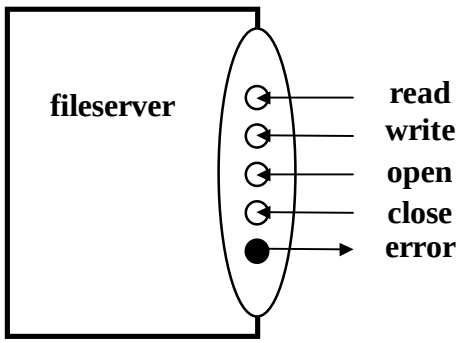


Figura 6.8 - Componente *fileserver*

A palavra **entry** indica que o componente *fileserver* irá aceitar chamadas pelas portas da *interface fileio* e irá realizar chamadas nas portas invertidas dessa *interface*. A palavra **exit** no lugar de **entry**, indica que o componente realizará chamadas pelas portas da *interface* e receberá chamadas pelas portas invertidas.

6.4.2 Primitivas de comunicação

O acesso às *interfaces* pelos componentes é realizada utilizando as primitivas de comunicação de REX. Essas primitivas são divididas em duas classes: as primitivas **call**, utilizada para invocar os serviços via portas e a primitivas **accept** utilizadas para receber chamadas pelas portas.

A sintaxe da primitiva **call** é:

call <porta>(<parâmetros>) [**wait**<variável> **fail**<variável>]

A sintaxe da primitiva **accept** é:

accept <porta>(<parâmetros>)

6.4.3 Linguagem de configuração Darwin

Em geral, Darwin descreve a maneira como as instâncias dos componentes serão interconectadas. A figura 6.9 mostra um sistema construído a partir das instâncias *cliente e servidor* dos componente *client e fileserver*.

```
component clientserver = {
use
    client
    fileserver
inst
     cliente
     servidor
bind
     cliente -- servidor
}
```

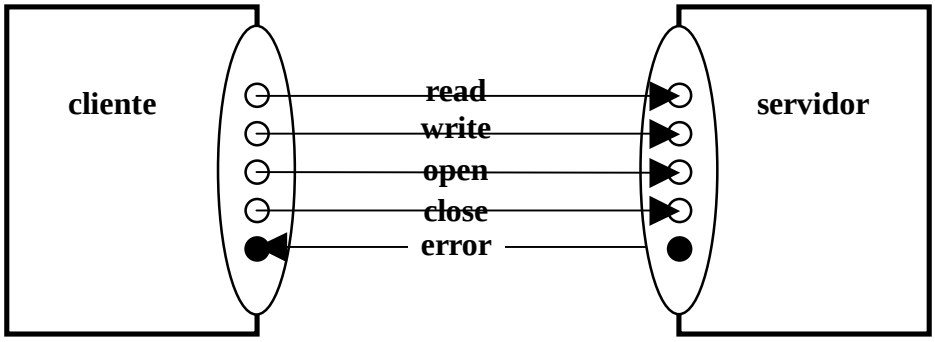


Figura 6.9 - Especificação de configuração em Darwin

Uma especificação de configuração em Darwin, utiliza os seguintes comandos:

use : identifica os tipos de componentes que farão parte da configuração.

inst : instancia os componentes a partir dos tipos definidos em **use.**

bind : cria uma conexão entre as instâncias criadas.

Para uma discussão mais detalhada sobre o projeto REX veja [Magee 90, Kramer 91, Kramer 91b].

6.5 O ambiente RIO

O ambiente RIO (*Reconfigurable Interconnectable Objects*) [Werner 91], é um projeto do grupo de sistemas de computação da Pontifícia Universidade Católica do Rio de Janeiro. Esse ambiente foi projetada para dar suporte ao desenvolvimento de aplicações baseadas em objetos utilizando o paradigma de configuração.

6.5.1 Arquitetura

A metodologia utilizada no ambiente RIO define dois conceitos fundamentais para a construção de um sistema:

- **Módulo**

- **Conector**

O conceito de módulo, por sua vez, se divide em dois, dependendo se é uma unidade operacional (instância) ou um tipo (classe) usado como fôrma para a criação de unidades operacionais. A estrutura básica de construção de um sistema é o módulo. Cada classe de módulo é uma entidade que contêm uma parte de código e dados e um ou mais pontos de interconexão (figura 6.10).

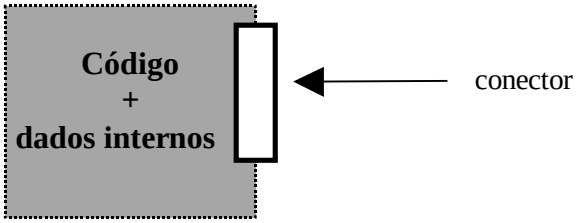


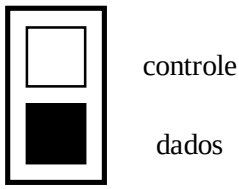
Figura 6.10 - Classe de módulo

Cada instância de módulo é uma imagem de uma classe que pode executar código e possuir um estado e dados internos. Uma instância sofre uma ação importante para a arquitetura: a **configuração.** A configuração consiste na disposição e organização de instâncias e suas relações, ou seja, sua interconexão. Em um modelo tradicional de programação, as ligações entre os elementos são estáticas como em chamada de procedimento ou invocação de método, sendo fixadas durante a programação. Numa metodologia baseada em objetos, tais ligações são potencialmente dinâmicas, realizadas em uma fase posterior de criação do sistema, ou mesmo mutáveis durante sua operação.

Esta estrutura exige um método bem definido para expressar a interação entre os módulos, realizada através de pontos de interconexão, chamados no ambiente RIO de **conectores**. A função do conector é organizar e estruturar a *interface* do módulo para o resto do sistema. Estes elementos concentram as atividades de comunicação de um módulo e são fundamentais à configuração, pois são os pontos de "colagem" de um módulo a outro.

6.5.2 Programação de módulos

A figura 6.11 mostra a definição de um conector:



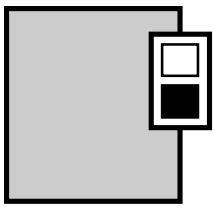

```
CONNECTOR tProducao {
   PIN dados {OUT tDados IN tConfirm}
   PIN controle {IN tControle OUT tConfirm}
}
```

Figura 6.11 - Definição de um conector

A especificação acima define um tipo de conector *tProdução* com dois pinos de ligação: o primeiro, *dados*, e o segundo, *controle*. Os tipos de mensagens associadas ao pino *dados* são tDados para a mensagem de pedido e tConfirm para a mensagem de resposta. A sintaxe é semelhante para todas as definições: a cláusula CONNECTOR contém o nome dado ao elemento definido e entre chaves o que está sendo definido. Dentro da cláusula de definição, descreve-se os elementos usados, seus nomes e tipos respectivos. Várias características podem ser definidas como

dimensionamento de *buffers* ou transações tipo síncronas, assíncronas e outras.

A figura 6.12 define uma classe de módulo.



producao

tipo produtor

```
CLASS tProdutor {
  CONECTOR producao { tProducao}
  CODE { ...
   }
}
```

```
CLASS tBuffer {
   CONECTOR entrada {PAIR tProducao}
   CONECTOR saida {tProducao}
   CODE { ...
    }
}
```

Figura 6.12 - Classes de módulos tProdutor e tBuffer

O exemplo acima define uma classe básica de módulo com um conector do tipo *tProducao* e nome *producao*. O exemplo define também uma classe *tBuffer* que contém dois conectores: *entrada* é uma **par** do tipo *tProducao*, informando que ele possui os mesmos pinos com mesmas características mas com sentidos de mensagens complementares. Na cláusula de código, exclusiva de uma classe básica, existem instruções na linguagem-base acrescidas de funções de comunicação definidas pela metodologia.

O ambiente de suporte oferece um conjunto de comandos de configuração que podem ser digitados, usados em arquivos de configuração, ou embutidos em programas. Isto simplifica a instalação e

gerenciamento de sistemas, permitindo também que reconfigurações planejadas ou não assistidas sejam realizadas.

O comando de instanciação é da forma:

INSTANTIATE produtor1 {tProduto, AT sementeA}

A cláusula AT é opcional e informa em que estação será instanciado. Uma estação ou semente é uma entidade lógica que permite otimizar as interrelações entre os módulos. Outro comando fundamental é aquele que liga um conector de instâncias de módulos a outro. Na figura 6.13 é mostrado o resultado dos comandos.

INSTANTIATE produtor2 {tProdutorDuplo}
INSTANTIATE consumidor1 {tConsumidor}
CONNECT produtor1 consumidor1

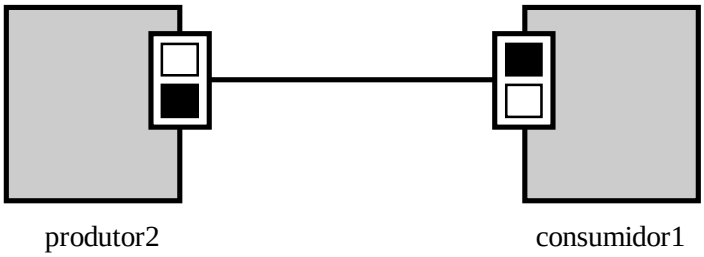


Figura 6.13 - Resultado da configuração

As primitivas descritas na tabela 6.1 suportam os comandos básicos de Configuração.

Comando	**Descrição**
INSTANTIATE	Cria uma instância de uma classe
CONNECT	Liga um conector (ou pino) a seu par ou grupo

GROUP	Define um grupo de conexão
UNCONNECT	Destroi uma ligação
RECONNECT	Altera uma ligação preservando mensagens
BLOCK	Bloqueia aplicações ou instâncias
UNBLOCK	Desbloqueia aplicações ou instâncias

Tabela 6.1 - Comandos de configuração

6.5.3 Primitivas de comunicação

As primitivas de comunicação foram definidas de forma a serem genéricas e independentes de protocolo e suportarem diversos tipos de transação. A tabela 6.2 lista as primitivas disponíveis no ambiente RIO.

Primitiva	**Descrição**
SEND	Envia mensagem de pedido
WAIT	Espera mensagem de resposta
RECEIVE	Recebe mensagem de pedido
REPLY	Envia mensagem de resposta
SELECT	Cláusula de seleção para múltiplas transações
WHEN	Guarda para restringir transações
TRANSFER	Transfere a mensagem recebida
RETURN	Retorna uma condição indicando fim da transação
EXCEPTION	Termina uma transação por decurso de tempo
TIMEOUT	Desbloqueia aplicações ou instâncias

Tabela 6.2 - Primitivas de comunicação

6.6 A linguagem de configuração CL

A linguagem de Configuração CL [Cunha 93], é uma ferramenta desenvolvida no Departamento de Informática da Universidade Federal de Pernambuco, dentro de um ambiente de programação para o desenvolvimento de aplicações que utilizem configuração dinâmica.

Nessa seção a linguagem CL será descrita rapidamente. Os comandos em CL podem ser classificados da seguinte maneira: os de configuração, os de reconfiguração e os de reconfiguração sincronizada.

Os comandos de configuração têm por objetivo descrever a estrutura inicial do sistema e são eles:

- **use task** : define o contexto do sistema informando os módulos que serão utilizados por este.
- **use configuration**: informa os módulos de Configuração a serem usados no sistema
- **create**: cria uma instância de um módulo.
- **link**: indica a ligação entre portas.
- **activate**: indica o início da execução de uma instância.

A sintaxe de cada pode ser visto na especificação da figura 6.14.

```
SYSTEM editor;
  USE TASK interface_type, buffer_manager_type; {Definição de Contexto}
  var end_edit: boolean;
  begin
    end_edit := false;
    CREATE interface FROM interface_type; {Criação de Instâncias}
    CREATE buffer_manager FROM buffer_manager_type;
    LINK interface.command to buffer_manager.command; {Ligação de Portas}
   ACTIVATE interface, buffer_manager; {Inicia execução dos processos}
   repeat
      WAIT {Monitora execução de interface}
          interface => if (IsActive(buffer_manager)) then
                          ACTIVATE interface;
                       else
                          end_edit := true; {Como buffer_manager terminou,
                                             interface termina normalmente}

       UNTIL end_edit;
end editor.
```

Figura 6.14 - Aplicação da linguagem CL

Os comandos de reconfiguração são os seguintes:

remove: remove um módulo do contexto do sistema.
delete: elimina uma instância ou uma família de instâncias.
unlink: desconecta ligações entre duas portas.
deactivate: desativa uma instância ou uma família de instâncias de um módulo.

O comando de configuração sincronizada garante que a reconfiguração será feita de forma segura, uma vez que seu objetivo é aguardar o término da execução de uma instância para executar uma seqüência de comandos. No exemplo da figura 6.14, o comando de reconfiguração sincronizada **wait** é utilizado.

Um módulo de configuração em CL, também denominado *group module,* assim com um módulo componente ou tarefa, têm suas *interfaces* declaradas da seguinte forma:

entryport porta1: integer; ou **exitport** k:1..4 porta2[k]: msg
onde:
entryport se refere a porta de entrada
exitport a de saída.

Para informações mais detalhadas sobre a linguagem de configuração CL e sobre o ambiente de suporte a execução dessa linguagem veja [Cunha 93, Justo 88, Justo 88b].

CAPÍTULO 7

PROGRAMAÇÃO ORIENTADA À CONFIGURAÇÃO

"A ciência conhece um único comando: contribuir com a ciência"

Bertold Brecht

7.1 Introdução

Como definido na introdução deste trabalho, um sistema distribuído é constituído por um conjunto de componentes, fracamente acoplados, que executam independentemente e se comunicam através de troca de mensagens. A tarefa de descrição, construção e manutenção do sistema é simplificada pela separação de sua estrutura, formada pelos componentes que o constituem e suas conexões, do comportamento funcional dos módulos individualmente.

Toda a tarefa de construção e manutenção do sistema pode ser realizada apenas pela manipulação de sua estrutura de configuração. Chamaremos de evolução quaisquer mudanças ou extensões sofridas pelo sistema no decorrer do tempo. Nesse contexto, qualquer *programação de mecanismos de configuração* deve passar pela manipulação das interconexões dos módulos em todas as fases de seu ciclo de vida: projeto, construção e evolução [Sloman 89].

Este capítulo apresenta uma descrição os princípios da programação com paradigma de configuração. O ambiente CONIC é utilizado para demonstrar as características deste paradigma.

7.2 Programação de aplicações configuráveis

A programação de aplicações configuráveis devem seguir alguns princípios básicos [Kramer 90]:

- *A linguagem de configuração utilizada para descrição estrutural deve ser separada da linguagem de programação utilizada para a programação dos componentes básicos.*
- *Os módulos que compõem o sistema devem possuir independência de contexto, uma interface bem definida e serem auto-contidos.*
- *A construção de módulos mais complexos deve ser realizada através da composição a partir de módulos mais simples.*
- *As modificações a serem introduzidas na estrutura do sistema devem ser realizadas pela troca de módulos e conexões.*

A separação da linguagem de programação dos módulos da linguagem de configuração do sistema torna mais fácil a compreensão e a manipulação da estrutura do sistema. A linguagem de configuração deve ser concisa e fácil de manipular, possivelmente declarativa de modo a levar em conta apenas a descrição da estrutura do sistema.

Independência de contexto [Kramer 85] significa que todas as referências realizadas por um módulo, devem ser exclusivamente a entidades locais. Isso torna o software reusável, no sentido de que esses módulos podem ser integrados a qualquer contexto compatível sem haver a necessidade de alterações internas. Qualquer acesso a entidades externas deve ser realizada através de chamadas indiretas a outros módulos.

O ambiente CONIC implementa todas essas características. Por isso para exemplificar o desenvolvimento de aplicações distribuídas configuráveis, veremos o desenvolvimento de uma aplicação em CONIC na próxima seção.

7.3 Programação de aplicações configuráveis em CONIC

A partir dos conceitos introduzidos na seção 6.3, desenvolveremos um sistema simples de drenagem de uma mina de carvão em CONIC para exemplificar a aplicação dos mecanismos de configuração no desenvolvimento de aplicações. A figura 7.1 mostra o esquema do sistema apresentado em [Kramer 83]. Esse sistema foi desenvolvido para controlar o bombeamento para a superfície, da água acumulada em um poço coletor existente dentro de uma mina de carvão.

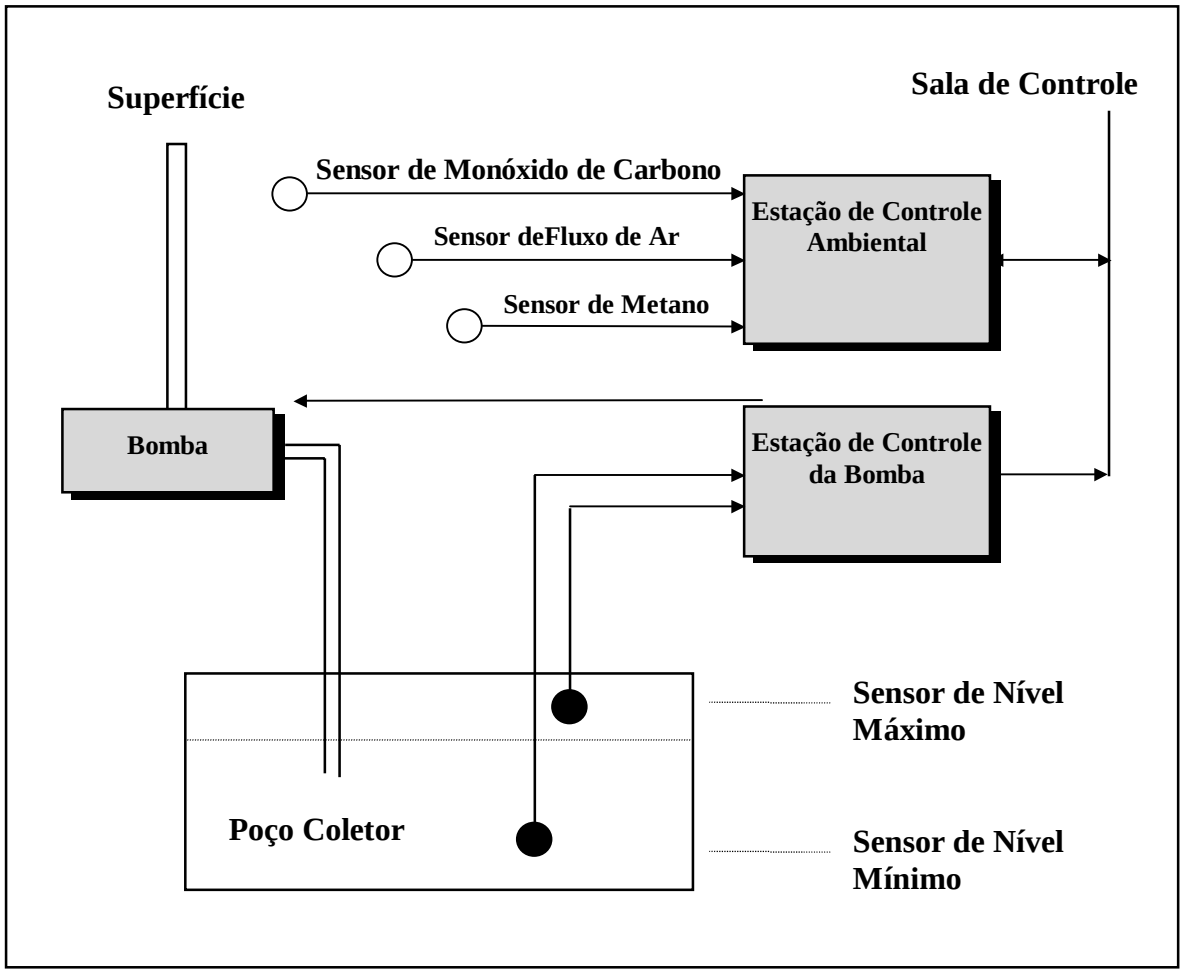


Figura 7.1 – Sistema de controle de bomba para drenagem de mina

A bomba é ativada pelo operador na superfície, e trabalha automaticamente, controlada pelo nível de água detectado pelos sensores de nível máximo e de nível mínimo. Se é detectado nível máximo ativa a bomba até que o nível mínimo seja atingido. O operador pode desativar a bomba a partir da superfície, e também pode solicitar o estado corrente da bomba.

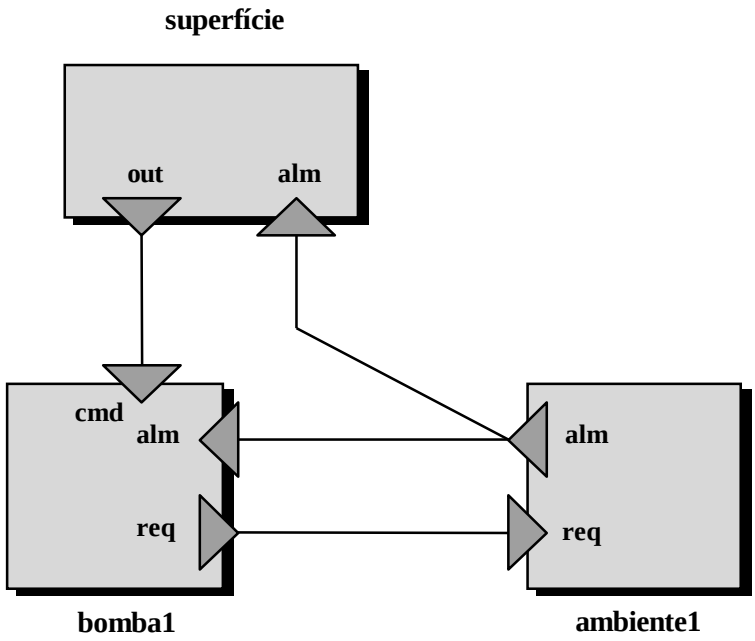


Figura 7.2 - Configuração do sistema de drenagem de mina

A bomba fica situada no fundo da mina e, por razões de segurança, não deve ser ligada ou continuar funcionando quando a porcentagem de metano excede um certo limite de segurança. O sistema obtém informações sobre o nível de metano através da estação de monitoramento ambiental, que também monitora o fluxo de ar e o nível de monóxido de carbono dentro da mina. Essas informações são passadas à estação de controle e para a sala de controle na superfície de modo a se evitar situações de risco.

Pela figura 7.1, fica fácil perceber que o sistema é decomposto em três módulos principais que representam, a estação de controle da bomba,

a estação de monitoramento ambiental e a sala de controle da superfície. A figura 7.2 mostra a estrutura lógica do sistema.

CONIC possui uma linguagem para a programação dos módulos e uma outra linguagem para a configuração do sistema. Vejamos como ficaria a especificação do sistema de controle de bomba utilizando a linguagem de configuração de CONIC. Os tipos de módulos utilizados na especificação serão definidos em seguida.

{ Criação das instâncias dos módulos a partir de tipos pré-definidos}

CREATE bomba1 : controlebomba;

CREATE superficie : salaoperador;

CREATE ambiente1 : controleambiental;

{Conexão das portas com validação de tipos}

LINK superfície.out **TO** bomba1.cmd;

LINK bomba1.req **TO** ambiente1.req;

LINK ambiente1.alm **TO** bomba1.alm, superfície.alm;

{Inicialização da Configuração}

START bomba1, superfície, ambiente1;

Os comandos utilizadas na linguagem de configuração de CONIC têm a seguinte semântica:

- **CREATE**: especifica a criação de instâncias de um dado tipo de módulo.
- **LINK**: estabelece uma conexão entra as portas de dois módulos.
- **START**: indica que os módulos podem ser executados.

Para a criação de módulos mais complexos, segundo os princípios da programação a nível de configuração, devemos decompô-los em módulos mais simples. O comando **GROUP MODULE** de CONIC permite a construção hierárquica de módulos a partir de **task modules**.

Para exemplificar a construção de módulos mais complexos, construiremos o tipo de módulo ***controlebomba***. A figura 7.3 mostra a estrutura do módulo ***controlebomba***. Esse módulo utiliza a unidade de definição *bombadefns*, e três **task modules**, *controle, bomba e poço.*

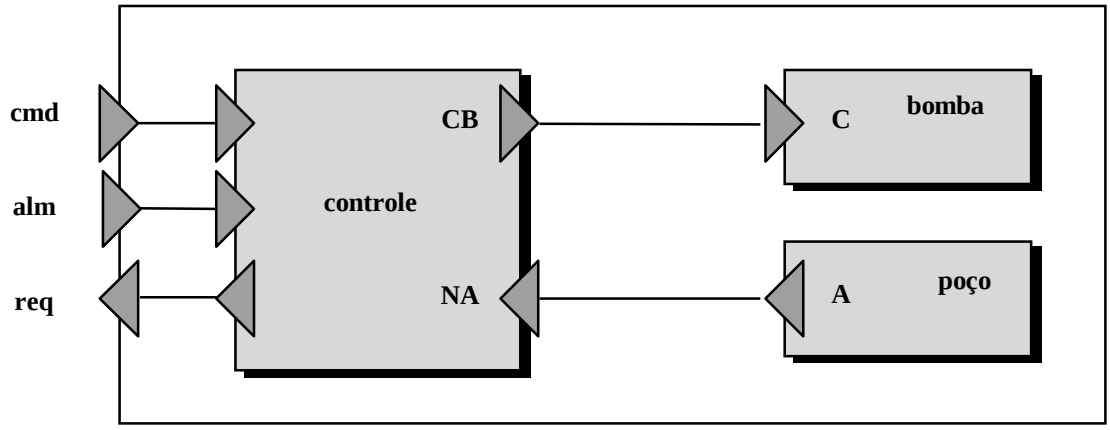


Figura 7.3 - Estrutura interna do módulo controlebomba

O módulo *poço* tem a função de periodicamente checar os sensores de nível de água e enviar o nível lido para o módulo *controle* através de uma ligação interna entre a porta *A* com a porta *NA*. O módulo *controle* executa o controle da bomba. Os programas que compões o **group module** *controlebomba* são mostrados no anexo D.

GROUP MODULE controlebomba (end_sensor : integer);

{Tipos de módulos utilizados na construção do módulo}

USE modulocontrole, modulobomba, modulopoço;

USE bombadefns: comando, estado, envped, envresp, alarme;

{Definição da Interface do módulo}

ENTRYPORT cmd : comando **REPLY** estado;

alm : alarme;

EXITPORT req : envped **REPLY** envresp;

{Criação dos componente internos do módulo}

CREATE controle : modulocontrole;

CREATE bomba : modulobomba;

CREATE poço : modulopoço (end_sensor : integer);

{Interconexão dos módulos internos}

LINK poço.W **TO** controle.WL;

LINK controle.PC **TO** bomba.C;

LINK cmd **TO** controle.cmd;

LINK alm **TO** controle.alm;

LINK controle.req **TO** req;

END.

A unidade de definições *bombadefns* é a unidade onde estão contidos os tipos de variáveis utilizados na definição de tipos nos módulos. Essa unidade tem a seguinte estrutura:

DEFINE bombadefns;

TYPE comando = (ativa, desativa, estado);

estado = (ativada, pronta, parada, paradapornível, paradapormetano);

sensor = (metano, carbono, fluxodear);

envresp = **RECORD**

leitura : **REAL**;

sn : sensor;

END.

PARTE III

O Ambiente ÁBACO

- **Capítulo 8: Descrição do Ambiente ÁBACO**
- **Capítulo 9: Arquitetura de *Software* do Ambiente ÁBACO**
- **Capítulo 10: Infraestrutura de Execução do Ambiente ÁBACO**
- **Capítulo 11: Metodologia para o Desenvolvimento de Aplicações no Ambiente ÁBACO**
- **Capítulo 12: Estudos de Caso**
- **Capítulo 13: Conclusões e Trabalhos Futuros**

Capítulo 8

Descrição
do
Ambiente ÁBACO

"A ciência não explicou nada. Quanto mais sabemos, mais fantástico se torna o mundo e mais profunda fica a escuridão ao seu redor "

Aldous Huxley

8.1 Introdução

O ambiente ÁBACO foi desenvolvido tendo como objetivo a utilização do paradigma de configuração no desenvolvimento de aplicações em plataformas de distribuição baseadas na tecnologia de objetos distribuídos ou DOC (*Distributed Object Computing*). Plataformas que utilizam a tecnologia DOC, como o CORBA, implementam um conjunto de vantagens, tais como uma avançada infraestrutura de comunicação, o tratamento de aspectos de heterogeneidade e mecanismos de tolerância a falhas. Em contrapartida o desenvolvimento de grandes sistemas nessas plataformas é dificultado pela estrutura de interconexão dos objetos clientes com os objetos servidores. Embora este problema possa ser contornado pela habilidade do desenvolvedor de aplicações, o resultado final seria um sistema pouco flexível, onde os componentes estariam fortemente acoplados, comprometendo a evolução desse sistema.

A modularidade fornecida pela característica de independência de contexto, proposta pelo paradigma de configuração descrito no capítulo 7, se mostrou como uma solução viável para os problemas complexos de interconexão dos objetos da DOC. Obedecendo essa característica, cada objeto poderá ser escrito independente da forma em que será conectado para estruturar o sistema. Essa conexão será realizada separadamente, permitindo ao desenvolvedor uma visão do sistema como um conjunto de objetos interconectados, independentes de sua estrutura interna.

O ambiente ÁBACO se apresenta como uma solução que congrega as vantagens das plataformas que utilizam DOC e a flexibilidade necessária ao desenvolvimento de aplicações do paradigma de

configuração, fornecendo um conjunto de ferramentas e um ambiente de execução para suportar as características desses dois modelos.

Este capítulo descreve a estrutura do ambiente ÁBACO, a qual é construída com as características principais da DOC e do paradigma de configuração, criando uma estrutura híbrida a partir destes dois modelos. Serão descritos a estrutura e os mecanismos de criação e conexão de objetos e a construção hierárquica de objetos, considerando-se outros objetos já definidos. Por fim, as ferramentas de *software* e o ambiente de suporte à execução, detalhados nos capítulos 9 e 10, respectivamente, serão apresentados de forma sumária.

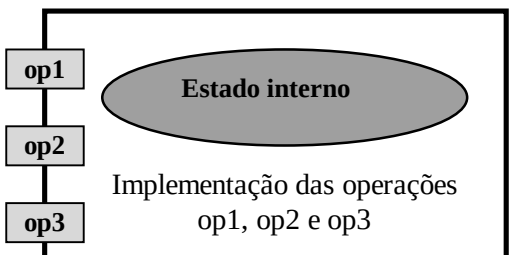
8.2 Descrição do ambiente ÁBACO

8.2.1 Estrutura dos objetos

Um dos princípios básicos da modularidade recomenda que os módulos de uma aplicação possuam *interfaces* onde devem ser definidas explicitamente todos os procedimentos capazes de afetar o comportamento do módulo, bem como o seu estado interno [Kramer 91]. A seguir, são descritas a estrutura dos objetos na DOC, no paradigma de configuração e no ambiente ÁBACO.

- **Estrutura dos objetos na DOC**

No modelo de objetos distribuídos uma *interface* de um objeto é descrita pelos métodos implementados por este objeto (figura 8.1). Porém, a utilização dos métodos de outros objetos não é explicitamente descrito na *interface* deste objeto, mas internamente em seu código.



Interface

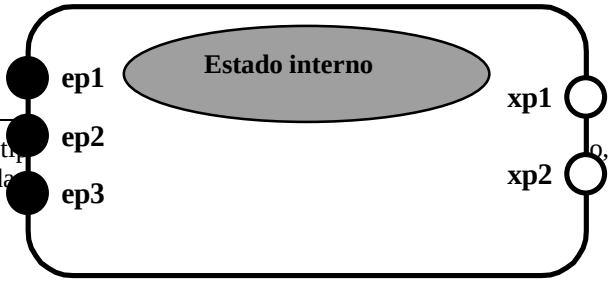
Figura 8.1 - Um objeto na DOC

- **Estrutura dos objetos no paradigma de configuração**

No paradigma de configuração, tanto os serviços *fornecidos* como os serviços *requeridos* por um objeto são necessários para a descrição do comportamento desse objeto e devem ser definidos explicitamente em sua *interface* [Kramer 91]. Dessa forma, a *interface* de um objeto deve mostrar, explicitamente, os métodos *oferecidos* e *requeridos* por este.

Do ponto de vista da invocação de métodos (operações), a invocação de um objeto anônimo[10] (não nomeado) é realizado indiretamente. O código do objeto nunca referencia diretamente outro objeto, mas indiretamente via as referências declaradas em sua *interface*. A ligação dos objetos deve ser realizada separadamente. Dessa forma, o objeto consegue independência de contexto o que facilita o seu uso e reuso. Seguindo essa estrutura, um objeto para o paradigma de configuração é definido em termos de:

1) um conjunto de portas representando os métodos (operações) fornecidos pelo objeto, e
2) um conjunto de portas que representam os métodos (operações) requeridas de outros objetos (figura 8.2)



[10] Este t
partir da
o, r
io a

Implementação das operações
fornecidas representadas pelas
portas ep1, ep2 e ep3

Figura 8.2 - Objeto com *interface* explícita

- **Estrutura dos objetos no ambiente ÁBACO**

No ÁBACO, os objetos terão uma estrutura na qual serão contempladas tanto as características dos objetos da DOC, quanto as características dos objetos do paradigma de configuração. A utilização desta estrutura híbrida, provém da necessidade desses objetos serem reconhecidos pelas plataformas de origem. Desta forma estes objetos continuam utilizando todos os mecanismos dessas plataformas, além de passarem a utilizar também as vantagens do paradigma de configuração.

A figura 8.3 ilustra um objeto do ambiente ÁBACO, denominado **componente.**

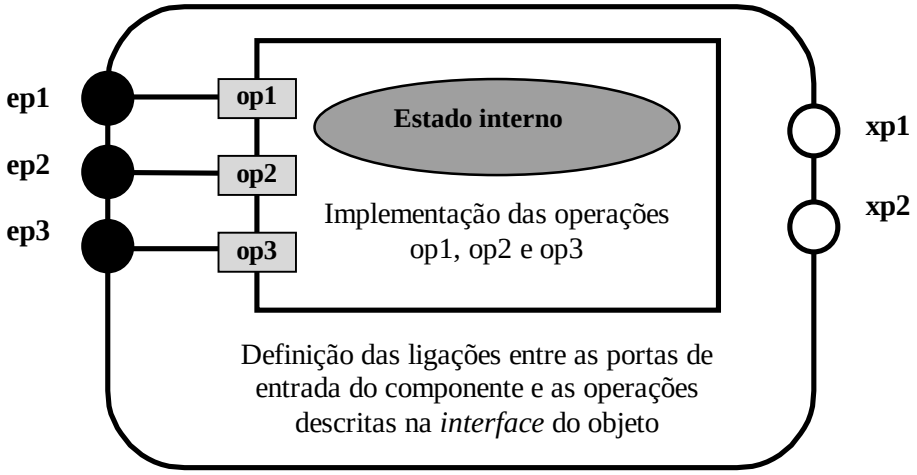


Figura 8.3 - Estrutura do componente do ambiente ÁBACO

O componente do ambiente ÁBACO tem a seguinte estrutura:

Estrutura interna do componente (um objeto da DOC):

- uma *interface* com a descrição das operações implementadas pelo objeto.

Estrutura externa do componente (um componente do paradigma de configuração):

- um conjunto de portas representando as operações *requeridas* pelo componente (portas de saída);
- um conjunto de portas representando as operações *oferecidas* pelo componente (portas de entrada); e
- um conjunto de ligações entre as portas de entrada do componente e as operações descritas na *interface* do objeto.

O ÁBACO fornece uma linguagem denominada CDL (*Component Descrition Language)*. Trata-se de uma linguagem declarativa com a qual os componente serão descritos. Esse linguagem é especificada no próximo capítulo.

8.2.2 Criação e conexão de objetos

A estrutura de um programa, pode ser pensada como um conjunto de instâncias de objetos interconectados. Em um ambiente distribuído, os objetos podem residir em diferentes máquinas, sendo, neste caso, a comunicação remota. Além disso, é necessário alocar os objetos nos nós físicos da rede, o que pode ser realizado tanto explicitamente pelo programador ou implicitamente pelo sistema. A seguir são apresentas os mecanismos de criação e conexão de objetos na DOC, no paradigma de configuração e no ambiente ÁBACO.

- **Criação e conexão dos objetos na DOC**

A abordagem adotada pela DOC é baseada na instanciação e ligação dinâmica dos objetos. A configuração do sistema se inicia por um simples objeto que, dinamicamente, cria outros objetos, local ou remotamente. A ligação entre as *interfaces* dos objetos é direta, por referência. Essas referência são obtidas através de um servidor de nomes ou *trader* que localiza a instância adequada do objeto referenciado. Portanto, a criação de objetos, ligação e alocação estão incorporadas à linguagem de programação.

- **Criação e conexão dos objetos no paradigma de configuração**

Nesse paradigma são especificadas duas linguagens, uma para a programação dos componente e uma outra para a construção do sistema, denominada linguagem de configuração. Através da linguagem de configuração, os componentes são criados ou removidos do sistema, são alocados nos nós físicos da rede e são conectados.

- **Criação e conexão dos objetos no ÁBACO**

Com a estruturação dos objetos de aplicação na forma de componentes, o ÁBACO utiliza uma linguagem de configuração a CCL (*Component Configuration Language*) para a manipulação desses componentes. Essa linguagem fornece todas as características das linguagens de configuração descritas no capítulo 7, permitindo a criação de instâncias e a remoção de componentes de um sistema, a conexão desses componentes e sua alocação nos nós físicos da rede. Esse linguagem é descrita em detalhes no próximo capítulo.

8.2.3 Construção hierárquica de objetos

A maneira mais simples de se construir módulos complexos é fazer uso do conceito de hierarquia, na qual um módulo pode ser formado pela união de outros módulos. A hierarquia é um meio natural e conveniente de suportar o encapsulamento de módulos e de informações. A seguir são apresentados os mecanismos de construção hierárquica de objetos na DOC, no paradigma de configuração e no ambiente ÁBACO.

- **Construção hierárquica de objetos na DOC**

A hierarquia é implementado na DOC pelo mecanismo de herança. Este permite objetos compartilharem os mesmos comportamentos e dados de sua superclasse. No entanto, as facilidades do mecanismo de herança podem ser facilmente violadas em ambientes distribuídos, uma vez que uma subclasse pode referenciar variáveis ou operações privadas em sua superclasse. Dessa forma, a utilização de variáveis de classes compartilhadas e procedimentos podem não ser eficientemente suportado em sistemas distribuídos [Wegner 90].

- **Construção hierárquica de objetos no paradigma de configuração**

O mecanismo de **composição**[11] oferece uma alternativa viável à herança. A composição é uma técnica que permite uma coleção de componentes serem tratados como um componente simples. Por esse mecanismo, um componente composto é descrito pela união de outros componente.

[11] Este mecanismo permite a definição de um componente a partir da união de outros componentes.

- **Construção hierárquica de objetos no ÁBACO**

O ambiente ÁBACO, utiliza o mecanismo de **composição** do paradigma de configuração. Assim, as ferramentas do ÁBACO permitem a criação de componentes pela união de outros componente (figura 8.4).

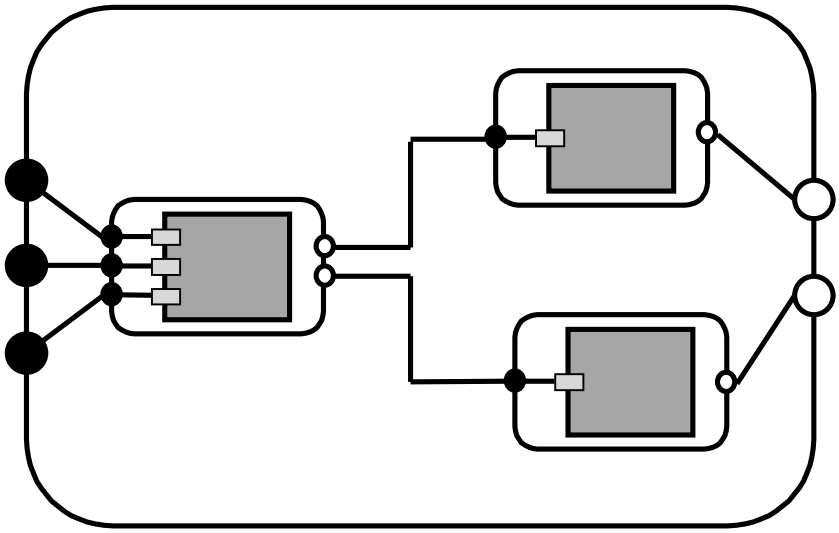


Figura 8.4 - Composição de componentes no ÁBACO

8.3 Arquitetura de *software*

Como dito anteriormente, o ambiente ÁBACO fornece duas linguagens para o desenvolvimento de aplicações:

- **CDL** - *Component Description Language* (Linguagem de descrição de componentes, e
- **CCL** - *Component Configuration Language* (Linguagem de configuração de componentes).

A separação da linguagem de definição de componentes da linguagem de configuração de sistema no ambiente ÁBACO, facilita a compreensão e manipulação do sistema, tanto pelo homem como pela máquina, em termos de sua estrutura. Isso é conseguido pela abstração da programação do componente. A construção do sistema pode ser realizada pela tradução da descrição da configuração. Esta tradução executa a criação e interconexão dos componentes. Dessa forma, a linguagem de configuração é declarativa, descrevendo a estrutura do sistema[12] e como ele deve ser construído. Descrições declarativas tendem a ser mais fáceis de analisar, interpretar e manipular do que as descrições imperativas equivalentes.

O capítulo 9, descreve a semântica e a sintaxe das linguagens CDL e CCL, dando alguns exemplos da aplicação dessas ferramentas no desenvolvimento de aplicações.

8.4 Ambiente de suporte à execução

Para viabilizar a utilização dos mecanismos do ÁBACO, foi criada uma infraestrutura de suporte à execução. Essa infraestrutura possui um suporte de apoio à execução, denominado **Sistema de Gerenciamento de Configuração**. Esse sistema, por sua vez, é formado por duas entidades: o gerente de comunicação e o gerente de configuração. Ele é encarregado de controlar os mecanismos de comunicação e (re)configuração de aplicações e de realizar o gerenciamento da execução dos componentes.

[12]Dado os benefícios da utilização de descrição estrutural para compreender e manipular o sistema, a implementação de alterações estruturais no sistema devem ser realizadas apenas a nível de Configuração.

As funções básicas de cada entidade do sistema de gerenciamento de configuração são:

- **gerente de comunicação:** controlar a troca de mensagens entre os componentes.
- **gerente de configuração:** Implementa as modificações na estrutura dos sistemas.

O capítulo 10 trás uma descrição detalhada sobre o sistema de gerenciamento de Configuração, incluindo os algoritmos de cada entidade.

8.5 Conclusão

O ambiente ÁBACO fornece uma infraestrutura híbrida, na qual as vantagens das plataformas que utilizam objetos distribuídos e a flexibilidade no desenvolvimento de aplicações do paradigma de configuração são plenamente suportadas.

A modularidade decorrente da composição de características da DOC e do paradigma de configuração permitem ao ÁBACO tratar mais facilmente problemas em sistemas distribuídos, mais especificamente a interconexão complexa ("muitos-para-muitos") de objetos no modelo Cliente/Servidor.

Esse capítulo apresentou a estrutura dos componentes do ÁBACO, suas ferramentas e seu sistema de suporte a execução. As características do ÁBACO podem ser sumarizadas nos seguintes itens:

- Componentes são os elementos básicos das aplicações;

- ÁBACO utiliza uma linguagem (CDL) para a descrição de componentes a partir de objetos definidos na DOC.

- ÁBACO utiliza uma linguagem de configuração de componentes (CCL), para definir a estrutura do programa a partir de um conjunto de componentes e de suas conexões.

Capítulo 9

Arquitetura de *Software* do Ambiente ÁBACO

"Os grandes acontecimentos ocorrem ser ter sido planejados. A sorte comete bons erros e desfaz os planos mais cuidadosos"

Georg C. Lichtenberg

9.1 Introdução

Por razões de clareza e flexibilidade, o ÁBACO fornece duas linguagens para a programação de aplicações. Uma para a descrição dos componentes (CDL) e outra para configuração do sistema (CCL) a partir desses componentes.

A linguagem CDL (*Component Description Language*), utiliza a estrutura de objetos da DOC e cria componentes a partir do encapsulamento desses objetos. Dessa forma, essa linguagem é utilizada para modelar os objetos a serem configurados. Isso é conseguido pela definição de portas para as operações requeridas e oferecidas pelo componente.

A linguagem CCL (*Component Configuration Language*), por sua vez, é utilizada para a configuração dos componentes das aplicações. Essa linguagem utiliza os componentes descritos em CDL para fazer a (re)configuração, através da criação, remoção e alteração de instâncias desses componentes e de conexões entre estes.

Este capítulo descreve, detalhadamente, a estrutura da linguagem CDL. Em seguida é descrita, também em detalhes a linguagem CCL.

9.2 *Component Description Language* (CDL)

Como descrito no capítulo anterior, a principal função da CDL é definir, a partir dos objetos da DOC, uma nova estrutura de objetos

independentes de contexto. Para a realização dessa tarefa, algumas passos devem se executados:

- definir as portas que representam as operações requeridas pelo componente (portas de saída);
- definir as portas que representam as operações oferecidas pelo componente (portas de entrada);
- definir a ligação entre os portas de entrada e as operações definidas na *interface* do objeto que o componente encapsula.

A estrutura textual de uma descrição em CDL para o módulo "controle" do exemplo mostrado no capítulo 7, é mostrada na figura 9.1:

```
COMPONENT controle(end_sensor:integer) {

USE tControle;

ENTRYPORT cmd TO comando,
                alm TO alarme,
                niv  TO nivel;

EXITPORT  ctl,
                req;
}
```

Figura 9.1 - Estrutura textual de uma especificação CDL

A representação gráfica correspondente a especificação CDL acima é mostrada na figura 9.2.

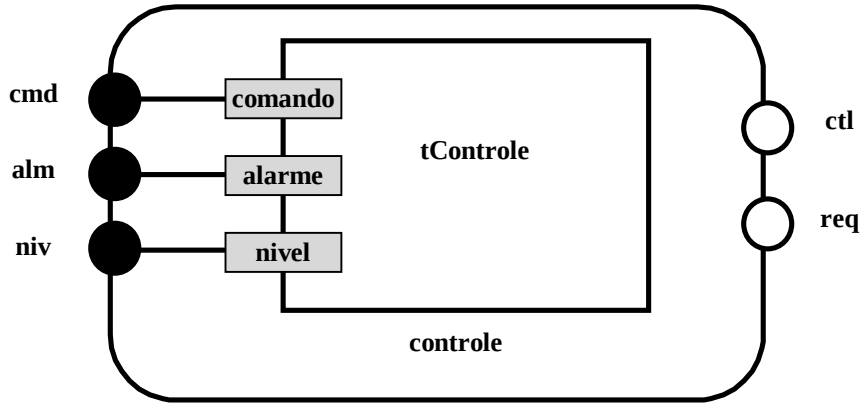


Figura 9.2 - Estrutura gráfica de uma especificação CDL

9.2.1 Sintaxe e semântica da linguagem CDL

Componentes são as unidades básicas do ÁBACO. Para a criação de componentes a partir de objetos da DOC, deve-se, primeiramente, definir um nome para este componente, a fim de que ele possa ser referenciado pela linguagem de configuração. O nome do componente é especificado pela cláusula:

COMPONENT <nome_componente> (<parâmetros>) ;

O termo <parâmetros>, indica uma lista de parâmetros utilizadas no momento da instanciação desse componente. Na figura 9.1, foi definido um componente com nome “controle” com um parâmetro “end_sensor” do tipo inteiro.

Cada componente simples encapsula um objeto da DOC. Para se definir qual o objeto está sendo utilizado para formar o componente, é utilizada a cláusula:

USE <objeto_da_DOC> ;

O termo <objeto_da_DOC> define o nome do objeto a partir do qual será gerado o componente.

No exemplo da figura 9.1, é utilizado o objeto “tControle” para formar o componente “controle”. Cada componente possui um conjunto de portas de entrada e portas de saída. Para se declarar as portas de entrada de um componente, é utilizada a cláusula seguinte:

ENTRYPORT <nome_da_porta> [**TO <**operação>] ;

O parâmetro <nome_da_porta> é utilizado para definir o nome pela qual essa porta será referenciada pela linguagem de configuração. O parâmetro <operação> faz a ligação da porta com a operação descrita na *interface* do objeto que o componente está encapsulando. A ausência desse parâmetro indica que o nome da porta vai ser o mesmo da operação. Dessa forma, qualquer chamada realizada a porta do componente, vai ser mapeada como uma chamada ao método do objeto a qual a porta está conectada.

Da mesma forma, para se especificar um conjunto de portas de saída de um componente, é utilizado a seguinte cláusula:

EXITPORT <nome_da_porta> ;

O parâmetro <nome_da_porta> identifica o nome pela qual a porta vai ser referenciada pela linguagem de configuração. Não existe nenhuma referência, a priori, sobre a operação a ser executada, por ocasião de uma chamada às portas de saída de um componente. Isto gera independência

de contexto do componente, visto que as únicas referências utilizadas por este são locais. A conexão entre uma porta de saída e a porta de entrada, onde a operação requerida está implementada, é realizada pela linguagem de configuração.

A criação de componentes compostos é uma tarefa executada pela linguagem CDL. Um componente composto é uma estrutura formada pela união de outros componentes. Ele é referenciado como um componente simples. A definição de componente compostos se inicia com a palavra reservada **COMPOSITE**. A figura 9.3 mostra a descrição do módulo "controlebomba" definido em CONIC no capítulo 7, com linguagem CDL. A figura 9.4 mostra a representação gráfica desse componente[13].

```
COMPOSITE COMPONENT controlebomba(end_sensor:integer) {

USE controle, bomba, poço;

ENTRYPORT cmd,
                  alm;


EXITPORT  req;

  /// Comandos de Configuração dos componentes internos ///

}
```

Figura 9.3 - Componente composto descrito em CDL

[13]Os comandos de configuração não são mostrados nessa descrição, visto que esses serão apresentados na próxima seção.

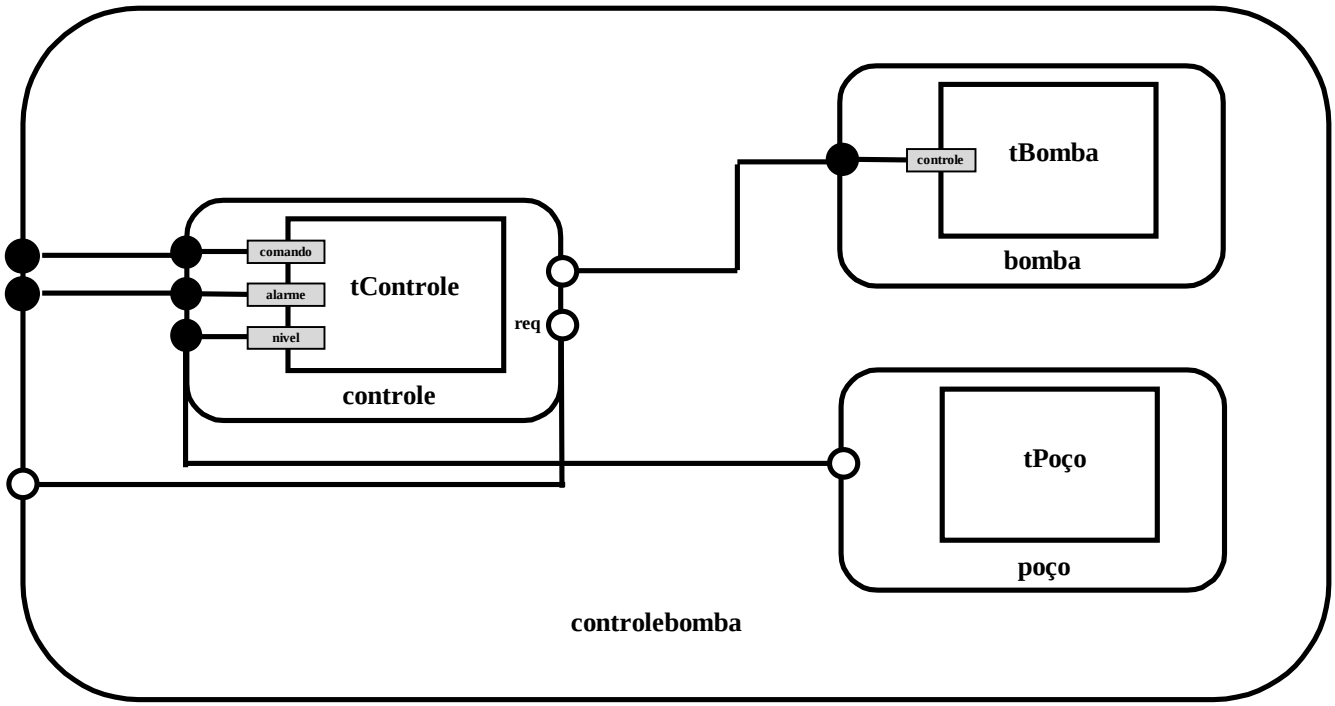


Figura 9.4 - Estrutura gráfica de um componente composto do ÁBACO

9.3 *Component Configuration Language* (CCL)

A linguagem de configuração CCL é utilizada no ambiente ÁBACO para realizar a (re)configuração dos componentes das aplicações. Essa linguagem descreve a maneira como as instâncias dos componentes devem ser criadas e conectadas de modo a estruturar uma aplicação. De uma forma geral uma especificação de configuração utilizando CCL deve seguir a seguinte estrutura:

- definição do nome do sistema;
- definição dos componente que fazem parte do sistema (contexto);
- criação das instâncias a partir dos componentes;
- conexão das portas das instâncias dos componentes.

A figura 9.5 mostra a definição em CCL do sistema de o mesmo exemplo utilizado na apresentação da linguagem CDL: controle de bomba para drenagem de mina.

```
SYSTEM sistema_de_controle_de_bomba {

USE controlebomba, salaoperador, controleambiental;

INSTANTIATE
        bomba1 FROM controlebomba AT iracema,
        superficie FROM salaoperador AT taiba,
        ambiente1 FROM controleambiental AT iguape;
LINK
        superficie.out TO bomba1.cmd,
        bomba1.req TO ambiente1.req,
        ambiente1.alm TO bomba1.alm,
        ambiente1.alm TO superficie1.alm;
START bomba1, superficie, ambiente;
}
```

Figura 9.5 - Estrutura textual de uma especificação CCL

9.3.1 Sintaxe e semântica da linguagem CCL

Uma especificação de configuração de um sistema utilizando CCL se inicia com a nomeação do sistema a ser configurado através da seguinte cláusula:

SYSTEM <nome_do_sistema>

Esse comando permite se identificar um sistema por um nome, definido em <nome_do_sistema>. Desse modo, qualquer informação requerida com relação ao sistema deve fazer referência ao seu nome especificado pela linguagem CCL.

Após definido o nome do sistema a ser configurado, os componentes que farão parte do contexto do sistema devem ser especificados. Isto é realizado pela cláusula:

USE <nome_do_componente> ;

O termo <nome_do_componente> identifica o componente que será utilizado no sistema. Esse comando informa à especificação de configuração quais componentes podem ser instanciados para a formação do sistema.

O comando **instantiate**, cria instâncias nomeadas a partir dos componentes. Alguns parâmetros iniciais podem ser passados para a instância por ocasião de sua criação. Podem ser criadas várias instâncias com nomes diferentes a partir do mesmo componente. A sintaxe geral do comando é:

INSTANTIATE <nome_instância> <(parâmetros)>
FROM <nome_do_componente> **AT** <nó_físico>

O comando **instantiate** cria instâncias do componente em um determinado nó físico da rede representado pelo parâmetro <nó_físico>.

Para realizar a ligação lógica entre as instâncias dos componentes, e definir a estrutura do sistema, é utilizado o comando **link**. Esse comando especifica a interconexão das portas das instâncias dos componentes. Essa ligação ocorre entre portas de saída de um componente com portas de entrada de outro(s) componente(s). O comando **link** segue a seguinte sintaxe:

LINK <nome_instância>.<porta_saída>

TO <nome_instância>.<porta_entrada>

A linguagem CCL também permite **reconfiguração** de um sistema. Dessa forma, são necessários mecanismos que permitem alterar a configuração, bem como acrescentar novos componentes à aplicação. A estrutura básica de uma especificação de **reconfiguração** é mostrada na figura 9.6, na qual é especificada em CCL a inclusão de uma nova instância do componente "salaoperador" com o nome "superficie2" no sistema denominado “sistema_de_controle_de_bomba":

```
CHANGE SYSTEM sistema_de_controle_de_bomba {

INSTANTIATE superficie2 FROM salaoperador AT iguape;

LINK
        superficie2.req TO bomba1.cmd,
        ambiente1.alm TO superficie2.alm;

START superficie2;

}
```

Figura 9.6 - Especificação de mudança em CCL

Uma especificação de mudança no sistema se inicia com a palavra reservada **change,** seguida do nome do sistema a qual se deseja aplicar a alteração. Os comandos **instantiate**, incluem uma instância de um componente em um sistema.

Os comandos básicos de alteração de configuração são relacionados a seguir:

REMOVE <nome_da_instância> : Remove um componente do contexto do sistema;

DELETE <nome_da_instância> : Elimina uma instância do sistema;

UNLINK <nome_instância>.<porta_saída>
TO <nome_instância>.<porta_entrada>: Desconecta ligações entre as portas.

9.4 Primitiva de comunicação do ambiente ÁBACO

A comunicação entre componentes deve ser realizada exclusivamente por meio de uma *interface* local, definida pelas portas destes componentes. Comunicação por meio de chamada direta a outras entidades limita a flexibilidade de configuração lógica dos sistemas. ÁBACO fornece uma primitiva de comunicação, o **call**, pela qual os componentes podem executar chamadas às suas portas locais.

A primitiva **call** é definida da seguinte forma:

CALL <nome_da_porta_local> [(<parâmetros>)] [**WAIT** <variável>] [**FAIL** <mensagem>]

O termo <parâmetros> indica os parâmetros passados à operação ligada à porta definida em <nome_da_porta_local>. A variável de retorno, opcional, esta definida em <variável>. E a mensagem a ser dado em caso de falha da chamada, também opcional, é definida em <mensagem>.

A utilização dessa primitiva de comunicação, impõe que as chamadas dos métodos dos objetos da DOC, realizadas pela utilização de referências aos objetos que implementam as operações, sejam substituídas pela primitiva **call**.

9.5 Conclusão

O ambiente ÁBACO utiliza duas linguagens: uma para a programação dos componentes a CDL, e outra para a configuração de sistemas a partir de seus componentes constituintes, a CCL. A separação entre estas duas linguagens segue um dos princípio básicos da programação a nível de configuração.

A linguagem CDL permite a construção de componentes independentes da configuração na qual eles executarão. Esta propriedade é denominada *independência de contexto*. Para se alcançar independência de contexto, os comandos da linguagem CDL se referem unicamente à variáveis locais. O BNF dessas duas linguagens se encontra no anexo C.

A construção inicial do sistema e suas futuras modificações são especificadas apenas em função da sua estrutura pela linguagem CCL. Essa separação entre a linguagem de descrição de componentes e a linguagem de configuração permite que o ÁBACO trate separadamente as questões relacionadas com a estrutura do sistema e as questões relativas à construção dos componentes individualmente.

Capítulo 10

Infraestrutura de Execução do Ambiente ÁBACO

"O esforço para entender o universo é uma das pouquíssimas coisas que erguem a vida humana a um nível ligeiramente além da farsa, dando-lhe algum toque de tragédia"

STEVEN WEINBERG

10.1 Introdução

O ambiente ÁBACO, descrito com detalhes no capítulo 8, permite o desenvolvimento de aplicações orientadas à configuração em plataformas de distribuição que utilizem objetos distribuídos. A fim de se implementar todas as funcionalidades exigidas pelo ambiente ÁBACO é necessário que haja um suporte de execução para este ambiente.

Dessa forma, uma infraestrutura foi desenvolvida no sentido de permitir a implementação de aplicações com o ÁBACO. Essa infraestrutura é formada, basicamente, pelo *Sistema de Gerenciamento de Configuração,* que é responsável pela execução dos componentes do ÁBACO.

Esse capítulo descreve o Sistema de Gerenciamento de Configuração. A principal tarefa do Sistema de Gerenciamento de Configuração é dar suporte à implementação de aplicações com o ÁBACO. Esse sistema é composto por duas entidades: o *gerente de configuração* e o *gerente de comunicação.*

10.2 O sistema de gerenciamento de configuração

10.2.1 O gerente de configuração

O gerente de configuração desempenha um papel de intermediário entre as aplicações e o desenvolvedor. Toda mudança de configuração especificada pelo desenvolvedor é submetida ao Sistema de Gerenciamento de Configuração através do gerente de configuração. Qualquer modificação

recebida pelo gerente de configuração é checada e validada com relação às portas e aos tipos de parâmetros das operações ligadas a estas.

O gerente de configuração obtém informações em uma base de dados que contém toda a estrutura de configuração de todos os sistema em execução. Após validar uma modificação na configuração de um sistema, o gerente de configuração gera um conjunto de comandos que executarão esta modificação e atualiza a base de dados.

O desenvolvedor poderá obter diversas informações com relação a estrutura das aplicações, através de comandos enviados ao gerente de configuração. Os comandos especificados no ÁBACO para controle de aplicações são mostrados na (tabela 10.1).

Comando	Resultado
Systems	Lista o conjunto de sistema em execução no momento
Ports <componente>	Lista o conjunto de portas de um determinado componente
Links <componente>	Lista as portas de saída conectadas às portas de entrada de um determinado componente

Tabela 10.1 - Comandos do gerente de configuração

10.2.2 O gerente de comunicação

A função do gerente de comunicação é controlar a troca de mensagens entre as portas dos componentes. Quando uma mensagem é enviada, o gerente de comunicação, baseado no mapa de configuração do sistema, transfere a mensagem para o objeto destino.

10.3 Implementação da infraestrutura de execução

A infraestrutura de execução foi implementada em C++. Dois programas foram definidos: o gerente de configuração (confman) e o gerente de comunicação (comman).

10.3.1 Implementação do gerente de configuração

O gerente de configuração recebe a especificação de mudança e gera um programa de configuração. Para implementar o programa de configuração, o gerente de configuração obtém as informações sobre os componentes do sistema a ser modificado no banco de dados de configuração. O gerente de configuração pode também receber pedidos de informação sobre a configuração de algum sistema.

Algoritmo do gerente de configuração

01 O gerente de configuração fica esperando uma solicitação;
02 Se for algum pedido de mudança
    03 O gerente de configuração consulta o banco de dados de configuração de acordo com a especificação de mudança recebida;
    04 O gerente de configuração gera um conjunto de comandos de mudança baseado na especificação;
    05 O gerente de configuração atualiza o banco de dados de configuração;
    06 O gerente de configuração avisa o fim da alteração solicitada;

07 Se for consulta de configuração

08 O gerente de configuração consulta o banco de dados de configuração e retorna o resultado;

10.3.2 Implementação do gerente de comunicação

O gerente de comunicação controla a troca de mensagens entre os componentes. Ele recebe as mensagens do objeto de origem e as repassa para o objeto de destino de acorda com o banco de dados de configuração.

Algoritmo do gerente de comunicação

01 O gerente de comunicação fica esperando um envio de mensagem de um componente;
02 O gerente de comunicação consulta o banco de dados de configuração e localiza a porta de destino;
03 O gerente de comunicação verifica a validade da chamada de acordo com os tipos dos parâmetros passados;
04 Se chamada válida e objeto destino está em execução
    05 Repassa o comando para o objeto destino;

10.4 Conclusão

A infraestrutura de execução implementada pelo ambiente ÁBACO, viabiliza a implementação de configuração proposta por este ambiente. Formado pelo gerente de configuração e pelo gerente de comunicação, o Sistema de Gerenciamento de Configuração permite o controle tanto da (re)configuração de sistemas como dos aspectos de comunicação entre os componentes. Esse sistema disponibiliza também, através do gerente de configuração, um conjunto de comando que permite se obter informações sobre os sistemas, como por exemplo o nome das portas de um

determinado componente ou as ligações realizadas a partir de um determinado componente.

As informações obtidas através do gerente de configuração, auxiliam o desenvolvedor na tomada de decisões com relação a realização de alguma alteração em um sistema.

Capítulo 11

Metodologia para o Desenvolvimento de Aplicações no Ambiente ÁBACO

"Erros são, no final das contas, fundamentos da verdade. Se um homem não sabe o que uma coisa é, já é um avanço do conhecimento saber o que não é"

11.1 Introdução

O ambiente ÁBACO disponibiliza um conjunto de ferramentas para a programação de aplicações orientadas à configuração sobre plataformas de distribuição orientadas a objetos. Visto que uma nova estrutura de *software* é incorporada a estas plataformas, é necessário fornecer uma metodologia de desenvolvimento de aplicações a fim de facilitar a tarefa de manipulação dessas ferramentas.

O paradigma de configuração, comparativamente a outras modelos, requer uma abordagem diferenciada com relação a especificação de um sistema. Várias características, implícitas a nova estrutura modular da aplicação, exigem uma metodologia que leve em consideração a flexibilidade proveniente desse paradigma e contemplem as suas vantagens.

Nesse capítulo, os aspectos relacionados com a modelagem, especificação e desenvolvimento de aplicações utilizando o ÁBACO serão abordados, de forma a se definir uma metodologia para o desenvolvimento de aplicações com esse ambiente.

11.2 Especificação e desenvolvimento de sistemas

A metodologia de desenvolvimento de sistemas para o ÁBACO, é baseada nos princípios da programação utilizando configuração, ou seja, devem ser realizadas uma especificação estrutural explícita, onde os tipos de componentes independentes de contextos são reconhecidos e as

composições de componentes são realizadas. Técnicas padrões de projeto de sistemas tais com as descritas em [Yourdon 78, DeMarco 79, Jackson 83] devem ser utilizadas para auxiliar no processo de decomposição, de forma que os princípios de modularidade e independência de contextos sejam satisfeitos.

Para que um sistema no ambiente ÁBACO possa ser totalmente especificado e desenvolvido, os seguintes passos, ilustrados na figura 11.1, devem ser observados:

- **Passo 1: Identificação estrutural e de componentes**

  Nessa primeira fase deve-se identificar os principais tipos de componentes e produzir uma especificação estrutural explícita, indicando o fluxo de dados principal do sistema. De posse da especificação inicial, os componentes identificados podem ser "explodidos" em subcomponentes;

- **Passo 2: Especificação das *Interfaces* dos Objetos**

  Essa etapa é responsável pela construção das *interfaces* dos objetos utilizando as linguagens de definição de *interfaces* da plataforma na qual o ambiente está sendo utilizado (IDL na plataforma CORBA, por exemplo).;

- **Passo 3: Especificação dos componentes**

  Nessa fase as *interfaces* dos componentes são identificadas e descritas utilizando a linguagem CDL. Os componentes compostos também são criados nesta fase;

- **Passo 4 : Implementação dos objetos**

  De posse do conjunto de portas disponíveis pela definição dos componentes, os objetos podem ser implementados. Nessa fase os comandos de chamada de operações de objetos, utilizados pelas

plataformas de distribuição (onde a aplicações está sendo implementada), devem ser substituídas pela primitiva **call** do ÁBACO;

- **Passo 5: Construção do sistema**

  Os componentes são instanciados nos nós da rede e interconectados pela linguagem de configuração CCL;

- **Passo 6: Evolução**

  Essa etapa surge da necessidade de realizar alguma modificação no sistema inicial, advinda da troca ou adição de novos componentes a este.

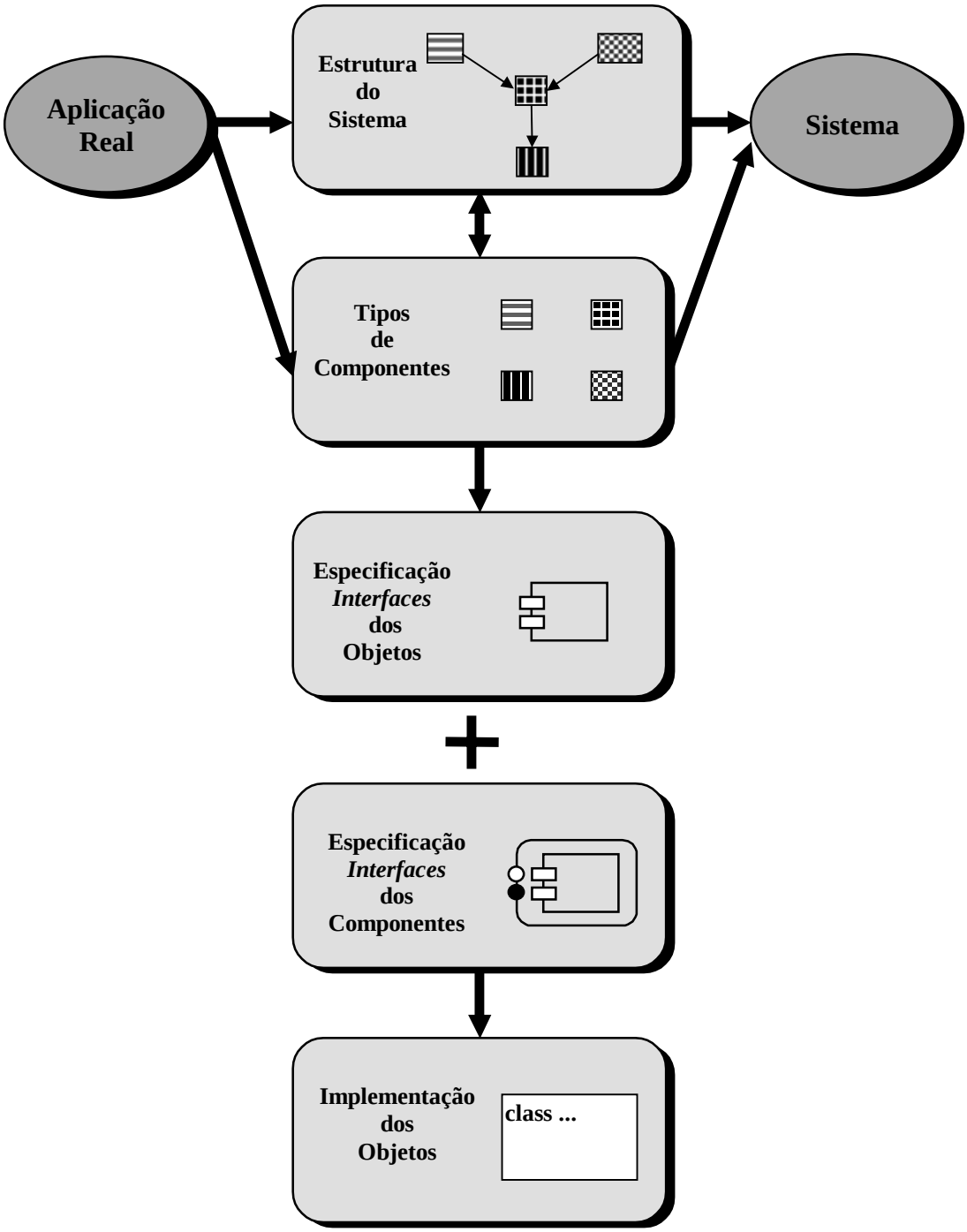

Aplicação Real
Estrutura do Sistema
Sistema
Tipos de Componentes
Especificação *Interfaces* dos Objetos
Especificação *Interfaces* dos Componentes
Implementação dos Objetos
class ...

Figura 11.1 - Etapas de desenvolvimento de aplicações com o ÁBACO

11.3 Evolução dos sistemas

O ambiente ÁBACO permite a modificação de um sistema através da inclusão, alteração ou exclusão de componentes. Isso é conseguido pela submissão de uma especificação de mudança ao gerente de configuração. A figura 11.2 mostra o processo de evolução de um sistema.

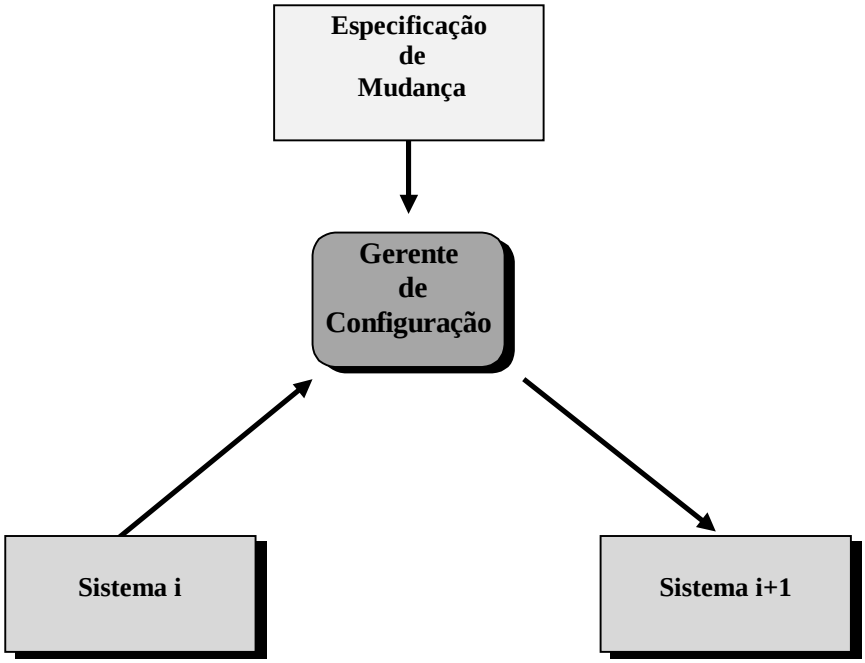


Figura 11.2 - Evolução de um sistema

11.4 Conclusão

O conjunto de ferramentas incorporadas às plataformas de distribuição orientadas a objetos pelo ÁBACO, modifica sensivelmente o trabalho de desenvolvimento de aplicações nessas plataformas. Novas etapas devem ser incorporadas ao processo de projeto e desenvolvimento dessas aplicações. Estas etapas utilizam as ferramentas disponibilizadas pelo ÁBACO e permitem a estruturação das aplicações de forma a utilizar as características do paradigma de configuração.

O trabalho extra trazido por estas novas etapas, é recompensado pela flexibilidade e facilidade de desenvolvimento de grandes sistemas decorrente da utilização de técnicas de configuração e pela total independência de contexto dos componentes do ÁBACO.

Capítulo 12

Estudos de Caso

"Todos os erros humanos são impaciência, uma interrupção prematura de um trabalho metódico"

Franz Kafka

12.1 Introdução

O ambiente ÁBACO é resultado de experiências em configuração na plataforma ANSAware. A utilização do ANSA deve-se a sua importância junto ao ODP com relação à padronizações de ambientes para programação distribuída e por ser uma arquitetura distribuída muito popular em 1994, época em que este trabalho foi iniciado. Nesse sentido, um protótipo de uma aplicação foi realizado e implementado utilizando-se a versão 3 do ANSAware. A experiência desse desenvolvimento é tratada com detalhes na seção 12.1, juntamente com a avaliação dos resultados.

No final de 1994, com a aprovação da especificação CORBA pelo OMG, pela abrangência de seus serviços[14], e devido ao sucesso do CORBA como plataforma distribuída o trabalho iniciado no ANSAware foi direcionado para aquele sistema.

O CORBA mostrou-se uma especificação promissora para o desenvolvimento de aplicações, pois além de permitir a divisão de grandes aplicativos monolíticos em componentes mais gerenciáveis, ele engloba todas as outras formas de computação Cliente/Servidor, inclusive monitores de TP, bancos de dados SQL e *groupware* [Orfali 95].

Este capítulo mostra dois estudos de caso. O primeiro mostra o uso de configuração na plataforma ANSAware. O segundo estudo de caso detalha a implementação de configuração na plataforma CORBA, exemplificando assim, a utilização completa da metodologia do ambiente ÁBACO.

[14]Nesta época, a OMG define um conjunto de serviços para objetos transacionais, com a idéia de criar um objeto ordinário e transformá-lo em transacional, bloqueável e persistente

12.2 Configuração no ANSAware

No desenvolvimento de aplicações orientadas à Configuração no ANSAware, foram testados os mecanismos de independência de contexto. Este estudo de caso analisa a implementação no ANSAware, descrita no anexo A, e cujo detalhamento se encontra em [Oliveira 92b].

- **A aplicação**

A aplicação desenvolvida, implementa um serviço de fatorial. Dois componentes são descritos: um servidor para operação de fatorial e um cliente desta operação.

- **A ligação entre os componentes**

A ligação do cliente da operação fatorial com o servidor dessa operação é realizada dentro do código do cliente, como descrito abaixo:

```
! {ir} <- traderRef$Import ("FATORIAL","/","")
! {fatorial} <- ir$fator(atoi(argv[1]))
```

Esta ligação é estática, ou seja, não pode ser alterada sem que o processo cliente tenha que ser reconstruído.[15]

Como primeiro passo para se implementar configuração, foi desenvolvido um mecanismo pelo qual o objeto cliente teria independência de contexto com relação ao objeto servidor.

[15] Como será visto mais adiante, o problema de se reconstruir o processo cliente é resolvido quando é colocado o mecanismo de configuração, de acordo com a metodologia ÁBACO.

- **Independência de contexto**

A independência de contexto no ANSAware, foi obtida pela inclusão de uma variável, que serve de porta para a chamada de uma determinada operação requerida pelo cliente. Uma outra porta foi necessária no objeto servidor para a definição da operação oferecida por este. Assim o cliente terá acesso a operação ligada a sua porta dependendo da configuração do sistema. Isso foi conseguido criando-se um arquivo onde são listadas todas as portas referentes às operações das *interfaces* de todos os servidores, todas as portas de todos os clientes e as ligações entre essas portas.

Dessa forma, um cliente invoca uma operação **send**, cujo protótipo estava definido em um arquivo *header* incluído em seu código pela diretiva *include*, na qual passa os parâmetros e o nome da variável de retorno. A execução da operação **send** lê o arquivo de configuração e gera a chamada a operação que estava ligada à porta do cliente. A expressão a seguir exemplifica a utilização da operação **send** no ANSAware.

SEND(n) **TO** Porta1 **WAIT** fatorial

12.3 Configuração no CORBA

Para validar completamente o ambiente ÁBACO na sua proposta de prover mecanismos de que facilitem o desenvolvimento de aplicações orientadas à configuração utilizando objetos distribuídos, foi escolhida a plataforma CORBA.

Nessa seção, parte da aplicação desenvolvida em CONIC, no capítulo 7, será inicialmente traduzida para uma aplicação CORBA. Logo

em seguida, utilizando os passos propostos no capítulo 11 para o desenvolvimento de aplicações no ÁBACO, ela será mapeada para uma aplicação orientada à configuração.

- **A aplicação**

Uma versão simplificada do sistema de controle de bomba para drenagem de mina exibida no capítulo 7, foi implementada utilizando a plataformas ORBeline 2.0. O sistema simplificada é composto pelos módulos: superfície e bomba. e tem a estrutura modular da figura 12.1.

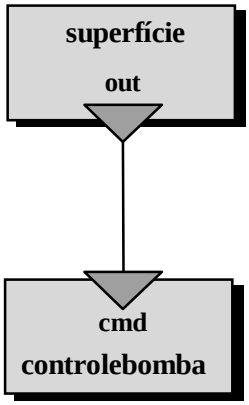


Figura 12.1 - Sistema de controle de bomba simplificado

Por sua vez, o módulo "controlebomba" é decomposto em três outros módulos: "controle", "bomba" e "poço" (figura 12.2).

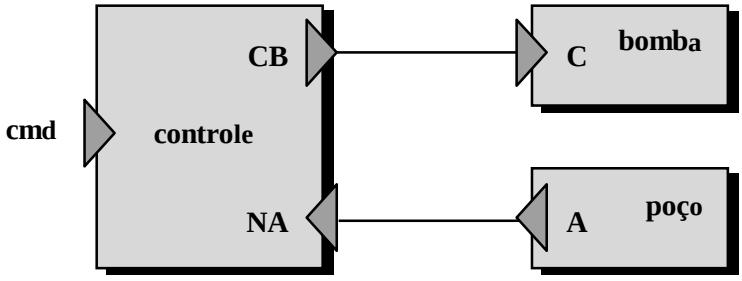


Figura 12.2 - Submódulos do módulo controlebomba

As operações seguintes são permitidas neste sistema simplificado:

- o operador do módulo superfície pode enviar sinais para a bomba na mina para ativar, desativar ou para verificar o estado do poço;
- o módulo poço periodicamente examina o nível de água no poço e envia o nível lido para o módulo controle;
- o módulo controle faz o pedido de ligar ou desligar para o módulo bomba;
- o módulo bomba liga ou desliga a bomba.

- **Implementação na plataforma ORBeline**

Com a estrutura do sistema especificada acima, fica claro a utilização de quatro módulos distintos: a superfície, o controle a bomba e o poço. Cada módulo que possua uma porta de entrada terá uma *interface* definida em IDL cujas operações serão definidas de acordo com cada porta de entrada. As portas de saída serão mapeadas em chamadas a operações dos módulos que as implementa. Desse são identificadas duas *interfaces* distintas: a do módulo controle e a do módulo bomba. A figura 12.3 descreve a *interface* do módulo controle e a figura 12.4 descreve a *interface* do módulo bomba.

```
#include "defines.h"
interface ITFControl{
  tEstado cmd(in tComando comando1);
   void NA(in tNivel nivel)
}
```

Figura 12.3 - *Interface* IDL do módulo controle

```
#include "defines.h"
interface ITFBomba{
   void C(in tComando comando2);
}
```

Figura 12.4 - *Interface* IDL do módulo bomba

Após definidas as *interfaces* em IDL dos módulos servidores, os respectivos programas são escritos (anexo B). Uma observação importante diz respeito a estrutura de conexão dos módulos desse sistema. De acordo com a estrutura do sistema, é possível perceber que o módulo controle tanto é servidor das operações **cmd** e **NA** como é cliente da operação **C**. Desse modo, o sistema possui um módulo tanto cliente como servidor, o

que torna o entendimento da estrutura de conexão uma tarefa bastante trabalhosa. E, dependendo do tamanho do sistema, é quase impossível se compreender. O sistema com uma estrutura de conexão desse tipo perde sua flexibilidade, de modo que a alteração de um módulo resulte na modificação em outros módulos.

- **Implementação com o ambiente ÁBACO na plataforma ORBeline**

De acordo com os passos da metodologia ÁBACO, descritos no capítulo 11, os passos 1 e 2 foram realizados no início do item anterior. De posse dos módulos da aplicação, o passo 3 pode ser realizado. Através da linguagem CDL os módulos superfície, controle, bomba e poço podem ser transformados em componentes. As classes que modelam esses objetos são: tSuperficie, tControle, tBomba e tPoco respectivamente.

As figuras 12.5, 12.6, 12.7 e 12.8 mostram a especificação dos componentes dos módulos superfície, controle, bomba e poço respectivamente. Como medida de simplificação, os nomes das portas de entrada e saída serão as mesmas definidas na *interface* dos objetos.

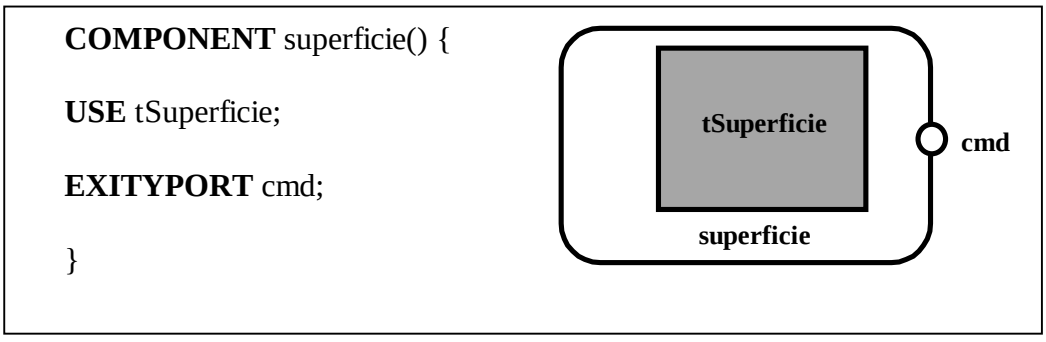
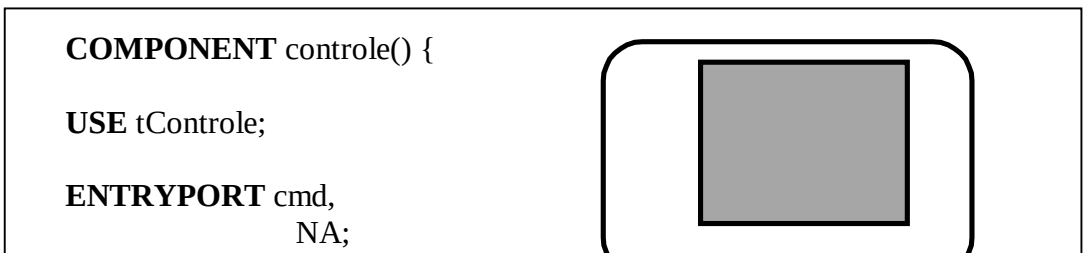


Figura 12.5 - Definição do componente superficie



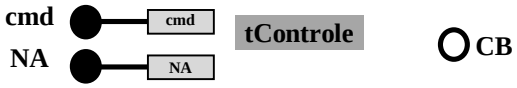


Figura 12.6 - Definição do componente controle

```
COMPONENT bomba() {
USE tBomba;
ENTRYPORT C;
}
```

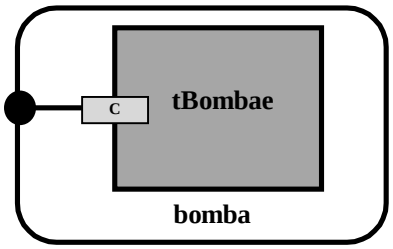


Figura 12.7 - Definição do componente bomba

```
COMPONENT poco() {
USE tPoco;
EXITPORT  A;
 }
```

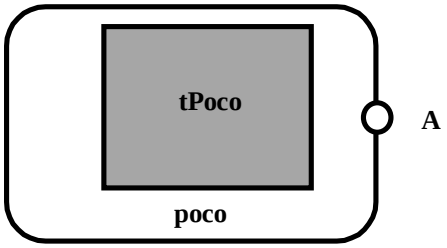


Figura 12.8 - Definição do componente poço

A partir da definição de componente composto do paradigma de configuração implementado pelo ambiente ÁBACO, o componente "controlebomba" é definido como a composição dos componentes controle, bomba e poço. A figura 12.9 mostra a descrição desse novo componente, enquanto a figura 12.10, mostra a estrutura gráfica do mesmo.

```
COMPOSITE COMPONENT controlebomba() {

USE controle, bomba, poço;
ENTRYPORT cmd;
LINK controle.CB TO bomba.C
LINK poco.A TO controle.NA
LINK controle.cmd TO controlebomba.cmd
}
```

Figura 12.9 - Definição do componente controlebomba

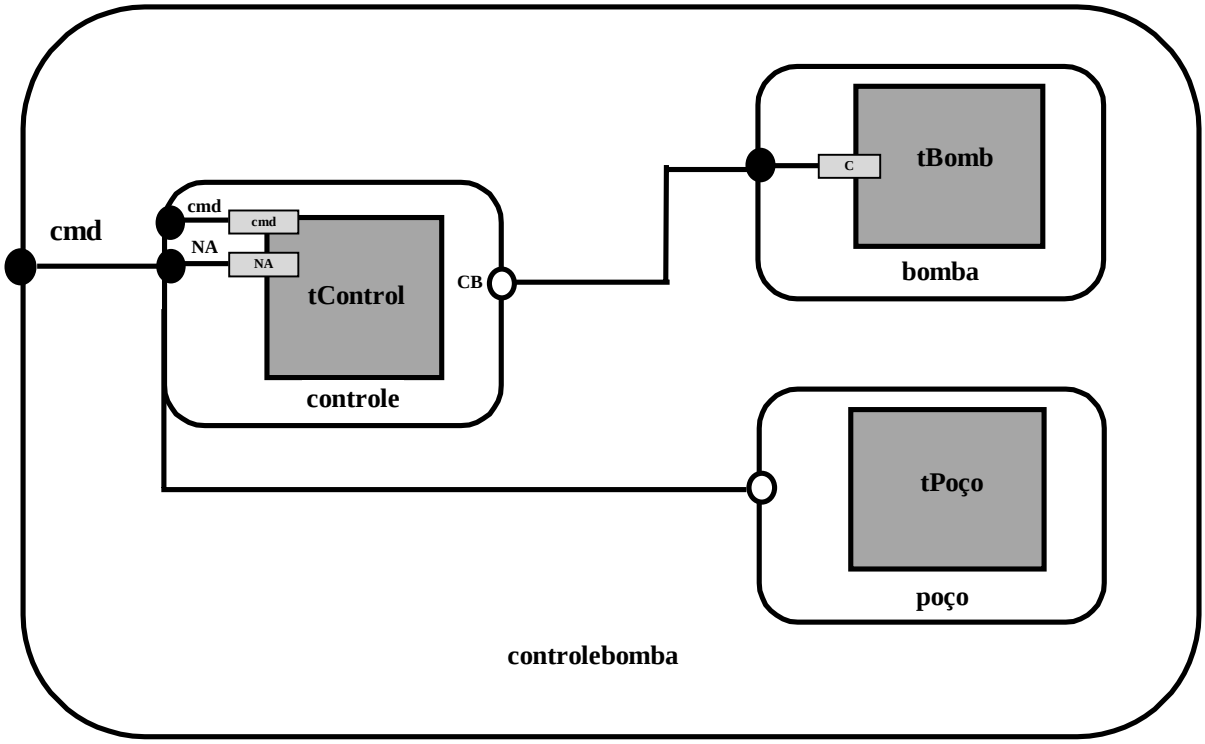


Figura 12.10 - Estrutura gráfica do componente controlebomba

Com a definição de todos os componentes do sistema, a fase de implementação do código dos objetos tSuperficie, tControle, tBomba e tPoço pode ser realizada. O código de cada objeto é escrito na linguagem da plataformas ORBeline, no caso C++. As únicas alterações imposta pelo ambiente são:

- inclusão do comando *# include <ÁBACO.hh>*;
- utilizar unicamente a operação **call** passando os parâmetros desejados , para a chamada da operação.

Por exemplo, o comando para parar a bomba enviado para o componente "poco" pelo componente "controle" teria a seguinte sintaxe:

CALL PC(parar)

O anexo B mostra o código completo desse sistema. O passo seguinte na implementação da aplicação, é a construção do sistema. Para isso é utilizada a linguagem CCL (figura 12.11).

```
SYSTEM sistema_de_controle_de_bomba {

USE controlebomba, superficie;

INSTANTIATE
        bomba FROM controlebomba AT iracema,
        operador FROM superficie AT taiba,
LINK
        superficie.out TO bomba.cmd,
START bomba, operador;
}
```

Figura 12.11 - Construção do sistema com a linguagem CCL

A partir da especificação da figura 12.11, o gerente de configuração inicia o processo de instanciação dos componentes. A submissão da especificação do sistema ao gerente de configuração é realizada através do comando:

> confman sistema_de_controle_de_bomba

O sistema ficaria com a estrutura da figura 12.12.

Com o sistema em execução, algum tipo de modificação pode ser necessária, o que completaria os passos de implementação desse sistema. Supondo que se desejasse modificar o sistema de controle de bomba se introduzindo um novo operador. Isso seria realizado incluindo-se um novo componente "superficie" (operador 2) na especificação. A nova estrutura do sistema está ilustrada na figura 12.13.

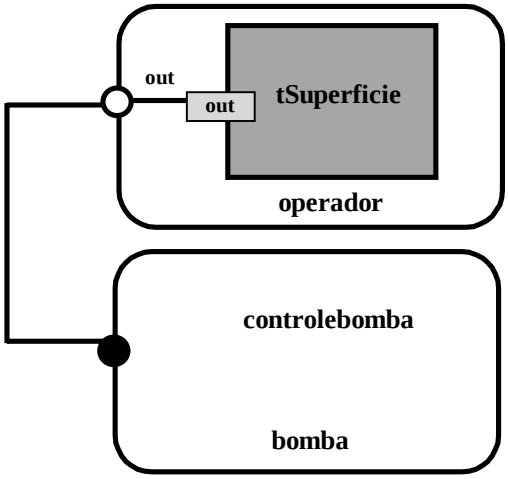
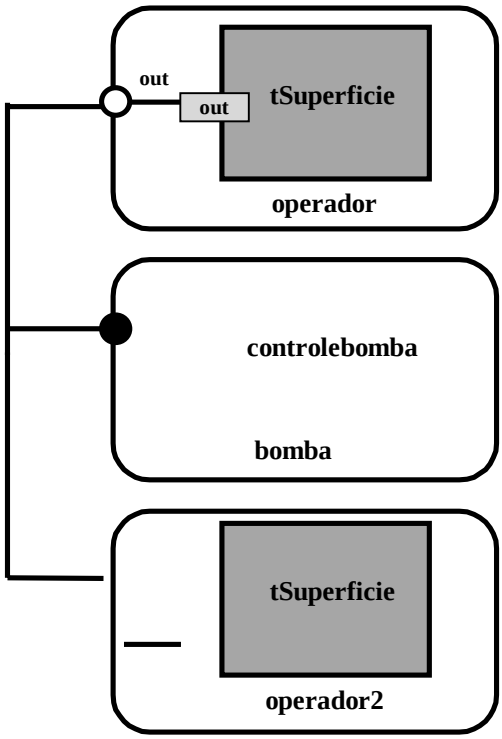


Figura 12.12 - Sistema em execução



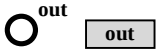


Figura 12.13 - Configuração modificada do sistema

A especificação que gerou esse novo sistema escrita com a linguagem CCL é a seguinte:

```
CHANGE SYSTEM sistema_de_controle_de_bomba {
INSTANTIATE  operador2 FROM superficie AT taiba;
LINK bomba.cmd TO operador2.out;
START operador2;
}
```

A especificação iniciada com **change** é interpretado pelo gerente de configuração como uma alteração a ser realizada no sistema. Desta forma a programação com o ambiente ÁBACO utiliza apenas a visão estrutural para introduzir evoluções no sistema

12.4 Conclusão

Com a utilização do ÁBACO, são tratados distintamente a estrutura do sistema, e a codificação dos componente que a forma. A construção inicial do sistema e suas futuras modificações são especificadas em função da sua estrutura, para isso utiliza-se uma linguagem CCL. Dessa forma, o ambiente distribuído trata separadamente as questões relacionadas com a estrutura do sistema e as questões relativas à construção dos módulos componentes individuais.

Percebe-se claramente, que a metodologia ÁBACO foi mais facilmente adaptada ao CORBA do que ao ANSAware, o que é razoável por ser a primeira plataforma baseada em objetos distribuídos.

CAPÍTULO 13

CONCLUSÕES
E
TRABALHOS FUTUROS

"Se sonhar é um pouco perigoso, a solução para isso não é sonhar menos – é sonhar mais."

Marcel Proust

13.1 Análise da proposta

Experiências de desenvolvimento de aplicações distribuídas utilizando plataformas de distribuição tais como ANSAware e ORBeline, mostram que essas plataformas apresentam algumas limitações com relação a flexibilidade necessária no gerenciamento de aplicações complexas. Um exemplo, é a utilização do modelo Cliente/Servidor em plataformas baseadas em objetos distribuídos, que dificulta a implementação de aplicações com padrões de conexão mais complicados. As grandes aplicações reais, normalmente utilizam servidores como sendo clientes de outros servidores, o que implica num padrão de conexão muitos-para-muitos. Isto torna o trabalho de implementação muito mais complexo.

- **Modelagem de sistemas**

Sistemas de grande porte com padrões de conexão complexo entre objetos clientes e objetos servidores, devem ser modelados de forma modular, permitindo que a complexidade da aplicação possa ser desenvolvida por partes. Este procedimento faz com que este sistema seja construído pela integração cooperativa destes módulos que em Engenharia de *Software* são denominados componentes. A partir desse visão da modelagem de um sistema por componentes, todos os módulos passam a ser clientes e/ou servidores, não existindo um papel predefinido para cada componente.

- **O paradigma de objetos**

Um outro modelo utilizado nessas plataformas é a orientação a objetos. O alto poder de modelagem de componentes desse paradigma se

contrapõe ao seu limitado poder de expressar distribuição. O mecanismo de herança desse paradigma, por exemplo, é uma característica interessante para estabelecer o relacionamento intra-componentes. No entanto ele se mostra inadequado na modelagem de aplicações distribuídas que exigem mecanismos explícitos para expressar um relacionamento inter-componentes.

- **O ambiente ÁBACO**

ÁBACO foi idealizado para auxiliar o desenvolvimento de aplicações em plataformas de distribuição orientadas a objetos, de forma a superar as limitações tanto do modelo Cliente/Servidor como do paradigma da orientação a objetos no tratamento da complexidade, acima comentada. Esse ambiente utiliza o paradigma de configuração para remodelar as aplicações, uma vez que esse paradigma se mostra adequado para expressar o relacionamento inter-componentes dentro de uma visão cooperativa utilizando a orientação a objetos.

- **O compromisso custo/benefício**

As vantagens da utilização do ÁBACO supera o esforço extra do desenvolvimento inicial de uma aplicação distribuída. A implementação desenvolvida neste trabalho utilizando a plataforma ORBeline mostrou que a evolução dessa aplicação foi facilidade sobremaneira pela utilização de configuração. A independência de contexto conseguida, a descrição clara da estrutura da aplicação e a possibilidade de criar componentes por composição facilitou bastante a compreensão e a conseqüente estruturação da referida aplicação.

13.2 Evolução do ambiente ÁBACO

A motivação inicial desse trabalho era o desenvolvimento de um mecanismo que permitisse criar aplicações orientadas à Configuração utilizando as plataformas de distribuição baseadas em IDL. Assim, foi desenvolvido e implementado um modelo de *interface*, denominado *modelo de interface configurável*. Esse modelo permitia a configuração de aplicações baseadas em IDL utilizando uma *interface* intermediária entre os componentes da aplicação. Dessa forma a configuração poderia ser realizada dentro dessa *interface*.

Experiências existentes, tais como o ambiente CONIC, as linguagens de configuração CL, DARWIN etc., exerceram uma natural influência na evolução da proposta inicial. Também o uso da plataforma ORBeline favoreceu a expansão da proposta, resultando na concepção de uma nova arquitetura, associada a um conjunto de ferramentas e uma metodologia que permitissem remodelar os objetos das aplicações distribuídas das plataformas baseadas em objetos distribuídos. Tudo isto com o intuito de criar uma estrutura de componente independente de contexto.

Dessa forma foi concebido a arquitetura ÁBACO. Uma linguagem de descrição de componentes, CDL, e uma linguagem de configuração, CCL, foram desenvolvidas. Uma metodologia para a utilização do um ambiente integrado pelas plataformas de distribuição e pelas ferramentas do ÁBACO, foi também desenvolvida. Essa metodologia contempla todos as

etapas do ciclo de vida de desenvolvimento de um *software* distribuído, desde sua modelagem até sua manutenção.

13.3 Relevância da contribuição

ÁBACO é considerado um ambiente uma vez que congrega uma arquitetura, ferramentas de software baseadas em linguagens de programação, uma infraestrutura de suporte à execução e uma metodologia para o desenvolvimento de aplicações.

A arquitetura do ambiente ÁBACO, é baseada numa modelagem híbrida dos objetos distribuídos e do paradigma de configuração. Essa arquitetura define os objetos das aplicações como componentes, que são descritos pela linguagem CDL (*Component Description Language*), e utilizados para formar a aplicação a partir de interconexões. Essa interconexão entre os componentes é realizado pela linguagem CCL (*Component Configuration Language*).

Como forma de sistematizar o desenvolvimento de aplicações a partir das linguagens e da estrutura de componente do ÁBACO, foi criada uma metodologia que auxilia o desenvolvedor de aplicações. Uma infraestrutura de suporte a execução, foi também desenvolvida com o intuito de viabilizar implementações baseadas em objetos distribuídos utilizando técnicas de configuração.

Dentre os benefícios aferidos pelo ambiente ÁBACO, quatro merecem destaques. Eles podem ser agrupados em duas vertentes: uma visando soluções de problemas concretos no desenvolvimento de aplicações distribuídas, e outra, mais abstrata, voltada para a utilização deste ambiente em pesquisa e ensino.

- **Problema 1:** ÁBACO permite que grandes sistemas não tenham que ser totalmente recompilados por ocasião de modificações estruturais simples, como a troca do nome de uma operação de um servidor;
- **Problema 2:** As aplicações com padrões de interconexão complexas (servidores sendo clientes de outros servidores), são facilmente tratadas pelas ferramentas do ÁBACO;
- **Pesquisa**: A criação de um ambiente híbrido a partir de objetos distribuídos e do paradigma de configuração, preservando as características individuais de cada um e suportando plenamente ambos, inaugura um novo ambiente de experimentação no Departamento de Computação da UFC. Os benefícios acadêmicos de tal ambiente podem ser projetados se observarmos o contexto evolutivo de outras experiências similares, tais como os ambientes RIO (PUC-Rio) e CL (UFPE).
- **Ensino**: ÁBACO apresenta-se também como grande potencial didático. Sendo o ensino de redes e sistemas distribuídos uma das linhas de pesquisa do LAR[16], o ambiente ÁBACO tornar-se-á uma ferramenta pedagógica no auxílio a disciplinas afins.

13.4 Trabalhos futuros

A experiência de implementação com o ambiente ÁBACO, permite algumas conclusões com relação à abrangência de plataformas que poderiam se beneficiar das características desse ambiente. A implementação da proposta inicial, realizada no ANSAware, revelou a importância da independência de contexto no desenvolvimento de aplicações distribuídas e direcionou a pesquisa para a adoção do

[16]Laboratório Multiinstitucional de Redes e Sistemas Distribuídos (UFC/UECE/ETFCE)

mecanismo de **componente** dentro de um esquema centralizado de configuração. Já a implementação do ambiente ÁBACO, evoluído da proposta inicial, levou a distribuição deste esquema de configuração no contexto de objetos distribuídos.

A lista a seguir sugere o desdobramento de uma série de trabalhos a serem desenvolvidos, o que valida o potencial do ÁBACO como um ambiente propício a experimentação, conforme proposto no item acima.

- Desenvolvimento de um ambiente gráfico, como o ConicDraw [Kramer 89] , [Souza 94], baseado no ambiente ÁBACO;
- Implementação de mecanismos que dêem suporte à configuração dinâmica. Apesar de não fazer uso de configuração dinâmica, a estrutura do ÁBACO foi modelada com vistas a adoção deste modelo;
- Validação de ferramentas do ambiente em outras plataformas de distribuição, propondo extensões para elas;
- Analise de desempenho do ÁBACO com intuito de avaliar a representatividade do "*overhead*" introduzido com os novos mecanismos propostos no ambiente;
- Análise de aspectos relevantes ao desenvolvimento de aplicações tais como segurança, tolerância a falha e outros.

O ambiente ÁBACO já vem sendo utilizado como plataformas para trabalhos final de curso em disciplinas de graduação. Em 1997 ele deverá ter os itens acima como tema de trabalhos de iniciação científica no DC/UFC. O ÁBACO também está inserido no projeto FLASH (FormaLizações da Administração de Sistemas Heterogêneos) do CNPq/PROTEM-III que envolve a UFPE, a UNB, ETFCE, além da UFC. O ambiente ÁBACO, suportado pelo CORBA, será utilizado neste projeto para o desenvolvimento de aplicações de administração de sistemas orientadas à configuração em ambientes heterogêneos.

Referências Bibliográficas

[APM 93] "AN OVERVIEW OF ANSAWARE 4.1", *Architecture Projects Management Ltd.*, Cambridge, UK, 1993.

[Booch 94] **G. Booch.:** "OBJECT-ORIENTED ANALISIS AND DESIGN WITH APPLICATIONS". *The Benjamim/Cummins Publishing Company, Inc*, 1994.

[CORBA 91] "THE COMMON OBJECT REQUEST BROKER: ARCHITECTURE AND SPECIFICATION". Object Management Group, Dezembro 1991.

[Cunha 93] **Cunha, P.R.F., Paula, V.C.C.:** "CL UMA LINGUAGEM ORIENTADA À CONFIGURAÇÃO PARA SISTEMAS DISTRIBUÍDOS"

[DeMarco 79] **De Marco, T.:** "STRUCTURED ANALYSIS AND STRUCTURED SPECIFICATIONS". Prentice Hall, 1979.

[DeRemer 75] **DeRemer, F., Kron, H.:** "PROGRAMMING-IN-THE-LARGE VERSUS PROGRAMMING-IN-THE-SMALL". *Proceedings of Conference on Reliable Software*, 1975.

[Ferraz 96] **Ferraz, Carlos A. G.:** "DISTRIBUTED OBJECT-BASED CONTINUOUS MEDIA APPLICATIONS". *Anais*

do XIV Simpósio de Redes de Computadores, páginas 433-452, Maio 1996.

[Filho 91] **Filho, J. A. L.:** "DESENVOLVIMENTO DE UM MODELO BASEADO EM CONEXÕES PARA CONFIGURAÇÃO DINÂMICA DE SISTEMAS DISTRIBUÍDOS", *Dissertação de Mestrado*, Depto. de Informática, Universidade Federal de Pernambuco, Março 1991.

[Gonçalves 96] **Gonçalves Jr. C., Teles, A., Drummond, R.:** *Desenvolvendo Aplicações Distribuídas em Cm*. XIV Simpósio Brasileiro de Redes de Computadores, Fortaleza, CE. Maio 1996.

[ISO 95a] "REFERENCE MODEL OF OPEN DISTRIBUTED PROCESSING PART1 – OVERVIEW", *ISO/IEC/JTC1/SC21/WG7/ISO10746-1*, Helsink, Finland, Julho 1995.

[ISO 95b] "REFERENCE MODEL OF OPEN DISTRIBUTED PROCESSING PART2 - DESCRITIVE MODEL". *ISO/IEC/JTC1/SC21/WG7/ISO10746-2*, Helsink, Finland, Julho 1995.

[ISO 95c] "REFERENCE MODEL OF OPEN DISTRIBUTED PROCESSING PART3 - PRESCRITIVE MODEL". *ISO/IEC/JTC1/SC21/WG7/ISO10746-3*, Helsink, Finland, Julho 1995.

[Justo 88] **Justo, G. R. R.:** "AMBIENTE DE PROGRAMAÇÃO DISTRIBUÍDA COM CONFIGURAÇÃO DINÂMICA DE

PROCESSOS", Dissertação de Mestrado , Depto. de Informática, UFPE, 1988.

[Justo 88b] **Justo, G. R. R.:** "AMBIENTE DE PROGRAMAÇÃO DISTRIBUÍDA COM CONFIGURAÇÃO DINÂMICA DE PROCESSOS", *XV Seminário Integrado de Software e Hardware*, 1988.

[Jackson 83] **Jackson, M. A.:** "SYSTEM DEVELOPMENT" Prentice Hall, 1983

[Kramer 83] **Kramer, J., Magee, J., Sloman, M., Lister,A.:** "CONIC: AN INTEGRATED APPROACH TO DISTRIBUTED COMPUTER COLTROL SYSTEMS", *IEEE Proceedings* Part E, Vol. 130, no.1, Janeiro 1983.

[Kramer 85] **Kramer, J., Magee, J.:** "DYNAMIC CONFIGURATION FOR DISTRIBUTED SYSTEMS", *IEEE Transactions on Software Engeneering*, SE-11 (4), Abril 1985.

[Kramer 87] **Kramer, J., Sloman, M.:** "DISTRIBUTED SYSTEMS AND COMPUTER NETWORKS", *Prentice-Hall International Series in Computer Science*, 1987.

[Kramer 89] **Kramer, J., Magee, J., Sloman, M.:** "CONSTRUCTING DISTRIBUTED SYSTEMS IN CONIC", *IEEE Proceedings* , Junho 1989.

[Kramer 89] **Kramer, J., Magee, J., NG, K.:** "GRAPHICAL CONFIGURATION PROGRAMMING", *IEEE Transactions on Software Engineering*, SE-11 (4).

[Kramer 90] **Kramer, J.:** "CONFIGURATION PROGRAMMING - A FRAMEWORK FOR THE DEVELOPMENT OF DISTRIBUTABLE SYSTEMS". *Proc. of IEEE COMPEURO'90*, Tel-Aviv, Israel, Maio 1990, pp 374-384.

[Kramer 91] **Kramer, J., Magee, J., Sloman, M., Dulay, N.:** "CONFIGURING OBJECT-BASED DISTRIBUTED PROGRAMS IN REX", Novembro 1991

[Kramer 91b] **Kramer, J., Magee, J., Sloman, M., Dulay, N.:** "AN INTRODUCTION TO DISTRIBUTED PROGRAMMING IN REX", 1991

[Kramer 92] **Kramer, J., Magee, J., Sloman, M.:** "CONFIGURING DISTRIBUTED SYSTEMS", *Proc. Of 5th ACM SIGOPS Workshop on Models and Paradigms for Distributed Systems Structuring*, Setembro, 1992.

[Magee 90] **Magee, J., Kramer, J., Sloman, M.**: *"AN OVERVIEW OF THE REX SOFTWARE ARCHITECTURE", Proc. Of 2nd IEEE Computer Society Workshop on Future Trends of Distributed Computer Systems*, Cairo, Outubro 1990.

[Marshak 91] **Marshak, David S.:** "ANSA - A MODEL FOR DISTRIBUTED COMPUTING". Network Monitor - Guide to Distributed Computing, Novembro 1991.

[Nelson 84] **Nelson, B.J., Birrel, A.D.:** "IMPLEMENTING REMOTE PROCEDURE CALLS". *ACM Transactions on Computer Systems*, Vol. 2 No 1, Fevereiro 1984.

[Oliveira 92] **Oliveira, M. et al.:** "*UMA PLATAFORMA DE COMPUTAÇÃO PARA ADMINISTRAÇÃO DE IBCNS*". *Submetido ao X SBRC*, Abril 1992.

[Oliveira 92b] **Oliveira, M. et al.:** "*DESENVOLVENDO APLICAÇÕES DISTRIBUÍDAS COM O ANSAWARE*". *Submetido ao X SBRC,* Abril 1992.

[Orbeline 94] "ORBELINE USER'S GUIDE". Release 1.3.1, PostModern Computing Tecnologies Inc, Fevereiro 1994

[Orfali 95] **Orfali, R., Harkey, D.:** "CLIENTE/SERVIDOR COM OBJETOS DISTRIBUÍDOS", *Byte*, Maio 1995 pg 76 - 78.

[Rodrigues 96] **Rodrigues, Roberto W.S.:** "*UMA METODOLOGIA PARA O DESENVOLVIMENTO DE APLICAÇÕES DE GERENCIAMENTO DE REDES BASEADA NO PARADIGMA DE CONFIGURAÇÃO*". *Dissertação de Mestrado,* Universidade Federal de Pernambuco, 1996.

[Rumbaugh 91] **J. Rumbaugh, M. Blaha, E. Eddy, and W. Lorensen.:** "*OBJECT-ORIENTED MODELING AND DESIGN*", Prentice-Hall, 1991.

[Schmidt 95] **Schmidt, Douglas C.:** "*INTRODUCTION DO DISTRIBUTED OBJECT COMPUTING*", *C++ Report*, SIGS, Vol . No. 1, Janeiro 1995.

[Schmidt 95a] **D. C. Schmidt, Vinoski, S.:** "*INTRODUCTION TO DISTRIBUTED OBJECT COMPUTING*". *C++ Report*, SIGS, Vol. 7 No. 1. Janeiro, 1995.

[Schmidt 95b] **D. C. Schmidt, Vinoski, S.:** "*MODELING DISTRIBUTED OBJECT APPLICATIONS*". *C++ Report*, SIGS, Vol. 7 No. 2. Fevereiro, 1995.

[Siegel 96] **Siegel, J.:** "CORBA FUNDAMENTALS AND PROGRAMMING", Editora Wiley Computer.

[Sloman, 89] **Sloman, M., Kramer, J., Magee, J.:** "CONFIGURATION SUPPORT FOR SYSTEM DESCRIPTION, CONSTRUCTION AND EVOLUTION", *Proc. of 5a Int. Workshop on Software Specification and Design*, Pittsburg, Maio 1989.

[Soares 95] **Soares, L.F. G.:** "REDES DE COMPUTADORES : DAS LANSS E WANSS ÀS REDES ATM". Editora Campus, 1995.

[Souza 96] **Souza, Cidcley T. de:** "IMPLEMENTANDO CONFIGURAÇÃO DINÂMICA DE PROCESSOS EM SISTEMAS DISTRIBUÍDOS". Trabalho submetido ao XXIII Semish, Agosto 1996.

[Tannenbaum 96] **Tannenbaum A.:** "MODERN OPERATING SYSTEMS". 1996.

[Wegner 90] **Wegner, P.:** "CONCEPTS AND PARADIGMS OF OBJECT-ORIENTED PROGRAMMING", *OOPS Messager (ACM SIGPLAN) vol.1*, 1990.

[Werner 91] **Werner, J.A.V., Filho, O.G.L.:** "AMBIENTE RIO: METODOLOGIA E SUPORTE PARA SISTEMAS CONFIGURÁVEIS"

[Yourdon 78] **Yourdon, E., Constantine, L.:** "STRUCTURE DESIGN", *Yourdon Press*, 1978.

[Zimmermann 1983] **Zimmermann, H. , Day, J.D.:** "THE OSI REFERENCE MODEL", *Proceedings of the IEEE*, Vol. 71, pp 1334-1340, Dezembro 1983.

Anexo A

Códigos de implementação ANSAware

A.1 Implementação no ANSAware

```
-- ##########  Especificação da interface ##########
FATORIAL : INTERFACE
BEGIN
    fator:OPERATION [ num : INTEGER ]
    RETURNS [ fatorial : INTEGER ]
END.

-- ########## Programa Servidor ##########
! USE Trader
! USE FATORIAL
# include <stdio.h>
/* variáveis globais */
  ansa_InterfaceRef er;
  void body()   /* Inicialização da Cápsula */
  {
          ! {er} <- traderRef$Export ("FATORIAL", "", 16)
          fprintf("Servidor FATORIAL Exportado para o Trader\n");
```

```
 }
        ansa_Integer FATORIAL_fator(_attr, num, result)
        ansa_InterfaceAttr *_attr;
        ansa_Integer            num;
        ansa_Integer            *result;
        {
        ansa_Integer            fat
              fat=num;
              if (fat==0 || fat==1){
                    *result=(ansa_Integer)  1;
                     return 1;
        } else
               fat = (fat) * (FATORIAL_fator(_attr, num1,result));
               *result = (ansa_Integer)  fat;
                return fat;     }
```

-- ########## Programa Cliente ##########

```
! USE FATORIAL
! DECLARE {ir} : FATORIAL CLIENT
# include <stdio.h>
/* variáveis globais */
ansa_InterfaceRef ir;
void body(argc, argv)
int argc;
char *argv [];{
ansa_Integer fatorial;
int n;
        !{ir} <- traderRef$Import ("FATORIAL","/","")
        ! {fatorial}  <- ir$fator(atoi(argv[1]))
        system("clear");
        printf(" O fatorial de %d = %d \n",n, fatorial);
}
```

A.2 Implementação orientada à configuração

```
-- ########## Programa Cliente  ##########
#include <portas.h>
# include <stdio.h>
/* variáveis globais */
ansa_InterfaceRef ir;
void body(argc, argv)
int argc;
char *argv [];
{
  ansa_Integer fatorial;
  int n;
  /* Inicialização da Cápsula */

  SEND(atoi(argv[1]) ) TO Porta1 WAIT fatorial
          system("clear");
          printf(" O fatorial de %d = %d \n",n, fatorial);
}
```

Anexo B

Códigos de Implementação ORBeline

B.1 Implementação no ORBeline

```
-- ########## defines.hh ##########
typedef enum comandos {
        ativar,
        desativar,
        estado
} tComando;

typedef enum estados {
        ativada,
        pronta,
        parada,
        paradaporniveldeagua,
        paradaporniveldemetano
} tEstado;

typedef enum nivel {
        alto,
        baixo,
        normal
} tNivel;

-- ########## ITFControle.idl ##########
#include "defines.h"
```

```
 interface ITFControle{
    tEstado cmd(in tComando comando1);
    void NA(in tNivel nivel)
 }
-- ########## tControle.C ##########
#include “tControle.hh”
int main()
{
    ITFControlS *ref = new ITFControlS;
    CORBA::BOA::impl_is_ready();
    delete ref;
    return(1);
}

-- ########## tControle.hh ##########
#include "defines.h"
#include "ITFControl_s.hh"

class ITFControlS: ITFControle_impl
{
 private:
       tEstado    estado;
 public:
      ITFControlS() { estado = 'parada'; }
     ~ITFControlS() {}

     tEstado cmd(tComando *comando1){
       switch(comando1) {
         case 'desativar':
              (if estado == 'ativada') {
                   ITFBomba * ref = ITFBomba::_bind;
                   ref->C(desativa);
```

```
                estado = 'parada';
                return estado;       }
            break;
        case 'ativar':
            if (estado != 'ativada')
              estado = 'pronta';
               return estado;
        case 'estado':
              cout << "O estado da bomba e: " << estado << endl;
              return estado;
      }
    }

void NA(tNivel *nivel) {
 switch(nivel) {
      case 'alto':
          (if (estado == 'pronta') ||((if estado == 'parada') ){
              ITFBomba * ref = ITFBomba::_bind;
              ref->C(ativar);
              estado = 'ativada';
              return;
          }
          break;

       case 'baixo':
          (if estado == 'ativada'){
              ITFBomba * ref = ITFBomba::_bind;
              ref->C(parar);
              estado = 'parada';
              return ;
          }
          break;
```

```
                case 'baixo':
                        break;
        }
        };
```

-- ########## tSuperficie.C ##########

```
#include "ITFControle_c.hh"
#include "stdio.h"
int main()
{
  int opc;
  while 1 {
     cout << "Qual o comando desejado:" << end;
     cout << "    opcoes: 1 - Ativa bomba " << endl;
     cout << "            2 - Desativa bomba " << endl;
     cout << "            3 - Verifica estado da bomba " << endl;
     cout << "            4 - Sair " << endl;

     opc = getche();

     ITFControle *ref = ITFControle::_bind();
     switch(opc) {
          case '1' : ref->cmd('ativa');
                     break;
          case '2' : ref->cmd('desativa');
                     break;
          case '3' : ref->cmd('estado');
                     break;
          case '4' : exit;   }
  };
```

-- ########## tPoco.C ##########

```
#include "ITFControle_c.hh"
#include "stdio.h"
#inlude “defines.hh”
int main(){
tNivel     opc;
  while 1 {
    cout << "Qual o nivel de agua na mina:" << endl;
    opc = getche();

    ITFControle *ref = ITFControle::_bind();

        ref->NA(opc);
      }
  };

       -- ########## ITFBomba.idl ##########
       #include "defines.h"
       interface ITFBomba{
         void C(in tComando comando2);
       };
      -- ########## tBomba.C ##########
      #include “tBomba.hh”
      int main()
      {
          ITFBombaS *ref = new ITFBombaS;
          CORBA::BOA::impl_is_ready();
          delete ref;
          return(1);
      }


      -- ########## tBomba.hh ###########
```

```
#include "defines.h"
#include "ITFBomba_s.hh"

class ITFBombaS: ITFBomba_impl
{
  private:
  public:
       ITFBombaS() { }
      ~ITFBombaS() {}
      void C(tComando *comando2){
        switch(comando2) {
          case 'ativar':
                  cout << "Bomba ativada! " << endl;
                  break;
          case 'desativa':
                  cout << "Bomba desativada! " << endl;
                  break;
    }
}
```

B2 Implementação no ambiente ÁBACO

```
-- ########## superficie.cdl ##########

COMPONENT superficie() {

USE tSuperficie;

EXITYPORT cmd;

}
```

```
-- ########## tSuperficie.C ##########
#include "stdio.h"
#include “ÁBACO.hh”
int main()
{
 int opc;
 while 1 {
   cout << "Qual o comando desejado:" << end;
   cout << "    opcoes: 1 - Ativa bomba " << endl;
   cout << "            2 - Desativa bomba " << endl;
   cout << "            3 - Verifica estado da bomba " << endl;
   cout << "            4 - Sair " << endl;
   opc = getche();
   switch(opc) {
       case '1' : CALL out('ativa');
                  break;
       case '2' : CALL out('desativa');
                  break;
       case '3' : CALL out('estado');
                  break;
       case '4' : exit;
    }
```

```
    };
  -- ########## controle.cdl ##########

   COMPONENT controle() {

   USE tControle;

   ENTRYPORT cmd,
                    NA;
    EXITPORT  CB;
    }
-- ########## tControle.C ##########
#include “tControle.hh”
int main()
{
    ITFControlS *ref = new ITFControlS;
    CORBA::BOA::impl_is_ready();
    delete ref;
    return(1);
}

-- ########## tControle.hh ##########
#include "defines.h"
#include "ITFControl_s.hh"
#include “ÁBACO.hh”
class ITFControlS: ITFControle_impl
{
  private:
       tEstado    estado;
  public:
      ITFControlS() { estado = 'parada'; }
     ~ITFControlS() {}

     tEstado cmd(tComando *comando1){
       switch(comando1) {
```

```
        case 'desativar':
           (if estado == 'ativada') {
                 CALL CB(desativa);
                 estado = 'parada';
                 return estado;
            }
            break;
        case 'ativar':
            if (estado != 'ativada')
              estado = 'pronta';
               return estado;
        case 'estado':
              cout << "O estado da bomba e: " << estado << endl;
              return estado;
      }
    }

void NA(tNivel *nivel) {
 switch(nivel) {
       case 'alto':
            (if (estado == 'pronta') ||((if estado == 'parada') ){
                 CALL CB(ativar);
                 estado = 'ativada';
                 return;
            }
            break;
        case 'baixo':
            (if estado == 'ativada'){
                 CALL CB(parar);
                 estado = 'parada';
                 return ;
            }
```

```
                break;
            case 'baixo':
                break;

    }
  }

-- ########## poco.cdl ##########
  COMPONENT poco() {
  USE tPoco;
  EXITPORT  A;
   }

  -- ########## tPoco.C ##########
  #include "ITFControle_c.hh"
  #include "stdio.h"
  #inlude “defines.hh”
  int main()
  {
  tNivel     opc;
   while 1 {
     cout << "Qual o nivel de agua na mina:" << endl;
     opc = getche();
    CALL A(opc);
  }
}

-- ########## bomba.cdl ##########

      COMPONENT bomba() {
      USE tBomba;
      ENTRYPORT C;
      }
```

```
-- ########## tBomba.C ##########
#include “tBomba.hh”
int main()
{
    ITFBombaS *ref = new ITFBombaS;
    CORBA::BOA::impl_is_ready();
    delete ref;
    return(1);
}

-- ########## tBomba.hh ##########
#include "defines.h"
#include "ITFBomba_s.hh"

class ITFBombaS: ITFBomba_impl
{
 private:
 public:
      ITFBombaS() { }
     ~ITFBombaS() {}

     void C(tComando *comando2){
       switch(comando2) {
         case 'ativar':
                  cout << "Bomba ativada! " << endl;
                  break;
         case 'desativa':
                  cout << "Bomba desativada! " << endl;
                  break;
  }
}
```

-- ########### controlebomba.cdl ##########

COMPOSITE COMPONENT controlebomba() {
USE controle, bomba, poço;
ENTRYPORT cmd;
LINK controle.CB **TO** poco.C
LINK poco.A **TO** controle.NA
LINK controle.cmd **TO** controlebomba.cmd

-- ########## sistema_de_controle_de_bomba.ccl ##########

SYSTEM sistema_de_controle_de_bomba {

USE controlebomba, superficie;

INSTANTIATE
bomba **FROM** controlebomba **AT** iracema,
operador **FROM** superficie **AT** taiba,
LINK
superficie.out **TO** bomba.cmd,
START bomba, operador;

Anexo C

BNF das Linguagem do Ambiente ÁBACO

C.1 BNF da linguagem CDL

<componente> ::= [COMPOSITE] COMPONENT <nome_componente> (<parametros>) { <corpo_componente> }.

<corpo_componente> ::= USE <nome_objeto> [, <nome_objeto>]*
ENTRYPORT<nome_porta> [TO <nome_operação>]
[<nome_porta> [TO <nome_operação>]]*
EXITPORT <nome_porta>
[,<nome_porta>]*
[LINK <instancia>.<porta_saida> TO <instancia>.<porta_entrada>]
[,[LINK <instancia>.<porta_saida> TO <instancia>.<porta_entrada>]]*

C.2 BNF da linguagem CCL

<sistema> ::= [CHANGE] SYSTEM <nome_sistema> { <corpo_sistema> }.

<corpo_sistema> ::= USE <nome_componente> [, <nome_componente>]*
INSTANTIATE <nome_componente> (<parametros)>)
FROM <componente> [AT <no_fisico>]
[, <nome_componente> (<parametros)>)
FROM <componente> [AT <no_fisico>]]*
[LINK <instancia>.<porta_saida> TO
<instancia>.<porta_entrada>]
[,[LINK <instancia>.<porta_saida> TO
<instancia>.<porta_entrada>]]
[REMOVE <nome_instancia>]
[,<nome_instancia>;]*
[DELETE <nome_instancia>]
[,<nome_instancia>;]*
[UNLINK <instancia>.<porta_saida> TO
<instancia>.<porta_entrada>]
[,[UNLINK <instancia>.<porta_saida> TO
<instancia>.<porta_entrada>]]
[START <instancia>]
[,<instancia>]

Anexo D

Especificação CONIC

```
TASK MODULE modulopoço (end_sensor : integer);
USE bombatipos : nivelagua;
EXITPORT A : nivelagua;
CONST periodo = 10;
VAR anivel : nivelagua;
{Manda periodicamente ler o nível da água}
BEGIN
      LOOP
        SEND anivel TO A;
              delay(periodo);
      END;
END.

TASK MODULE  modulocontrole;
USE bombadefns : comando, estado, sensor, envresp, envped, alarme;
USE bombatipos : nivelagua, controlebomba;
ENTRYPORT cmd : comando REPLY estado;
                alm : alarme;
                NA : nivelagua;
EXITPORT  req : envped REPLY envresp;
               CB : controlebomba;
      CONST limiteseguro = 1.25; periodo = 100;
      VAR  bestado : estado;
           balarme : alarme;
           bnivel : nivelagua;
           bcomando : comando;
```

```
{Faz pedido e checa nível de metano}
FUNCTION IniciaChecagem : estado;
VAR   pedido : envped;
      resposta : envresp;
BEGIN
      pedido := metano;
      SEND pedido TO req
      WAIT resposta =>
       IF resposta.leitura < limiteseguro THEN
            IniciaChecagem := pronta;
       ELSE
            IniciaChecagem := paradapormetado;
            FAIL periodo => writeln('Falha de Pedido');
            IniciaChecagem := paradapormetado;
      END;

{ Programa Principal}
BEGIN
bestado := parada;
      LOOP
      SELECT
            RECEIVE bcomando FORM cmd =>
            CASE bcomando OF
                  desativar:
                  BEGIN
                  IF bestado = ativada THEN
                        SEND parar TO CB;
                        bestado := parada;
                  END;
                  ativar :
                  BEGIN
```

```
                IF bestado <> ativada THEN
                    bestado := pronta;
                END;
                REPLY bestado
                OR RECEIVE balarme FROM alm =>
                    IF bestado = ativada THEN
                    SEND parar TO CB;
                    bestado := paradapormetano;
                OR RECEIVE bnivel FROM NA =>
                    CASE bnivel OF
                    alto :
                    IF bestado = pronta THEN
                    BEGIN
                      bestado := IniciaChecagem;
                        IF bestado = pronta THEN BEGIN
                        SEND  ativa TO CB;
                         bestado := ativada END;
                    END;
                    baixo :
                    IF bestado = ativada THEN
                    BEGIN
                        SEND parar TO CB;
                        bestado := paradapornivel END;
                    normal : ;
                    END;
            END;
END.
TASK MODULE modulobomba;
USE bombatipos : controlebomba;
ENTRYPORT C : controlebomba;
VAR comando : controlebomba;
```

```
BEGIN
    LOOP
    RECEIVE comando FROM C;
    CASE comando OF
    ativar : 'Ativa a bomba'
     parar : 'Para a bomba'
    END;
END.
```