Integração do Serviço de Diretório LDAP com o Serviço de Nomes CORBA

Gustavo Scalco Isquierdo

Dissertação Apresentada ao
Instituto de Matemática e Estatística da
Universidade de São Paulo
como Parte dos Requisitos
para a Obtenção do Grau de
Mestre em Ciência da Computação

Área de Concentração: Ciência da Computação
Orientador: Prof. Dr. Francisco Reverbel

São Paulo — Outubro de 2001

Integração do
Serviço de Diretório LDAP
com o
Serviço de Nomes CORBA

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Gustavo Scalco Isquierdo
e aprovada pela comissão julgadora.

São Paulo, 31 de outubro de 2001.

Banca examinadora:

- Prof. Dr. Francisco Reverbel (orientador) - IME - USP
- Profa. Dra. Graça Bressan - Escola Politécnica da USP
- Prof. Dr. Fábio Kon - IME - USP

Agradecimentos

- Ao Prof. Reverbel, por me acolher como orientando definindo um tema de dissertação de forma a atender ao interesse mútuo, o que tornou o trabalho estimulante. Pela sua disponibilidade e participação ativa que viabilizou a conclusão do trabalho.

- Ao Prof. José Augusto Ramos Soares, que na função de orientador de curso, incentivou-me na execução dos créditos, fazendo as recomendações sobre as disciplinas que tinham mais afinidade com meus interesses.

- Ao Prof. José Coelho de Pina, que lecionou a disciplina "Introdução à Computação II" no curso de verão do IME, fornecendo-me o conhecimento necessário para cursar, posteriormente, a disciplina do mestrado "Introdução de Análise de Algoritmo".

- À Profa. Graça Bressan e ao Prof. Fábio Kon, pela pronta aceitação para a participação da banca examinadora. Pela paciência em ler este trabalho e pelas sugestões apontadas após a exposição.

- À funcionária Francisca, da Secretaria da Pós-Graduação, pelo acompanhamento nos processos das solicitações formais que fiz à CPG e pelo seu apoio nas providências necessárias.

- Ao Jorge Ono e José Lucindo Ramalho Neto, que, como meus chefes imediatos, compreenderam a importância da obtenção deste grau fazendo o que puderam para eu concluir o curso.

- À minha querida Rosângela, que contribuiu para a realização deste desejo, privando-se, muitas vezes, do convívio e das atividades de lazer conjunto durante os momentos dos trabalhos acadêmicos.

- Aos meus pais, Maria Angelina e Wilson, que me educaram enfatizando os estudos, incentivando-me na escalada do conhecimento e na obtenção de mais um título na formação universitária.

Resumo

Esta dissertação apresenta uma solução para a integração do serviço de diretório LDAP com o serviço de nomes CORBA. Descrevemos a implementação de um servidor de nomes CORBA que armazena, num diretório LDAP, as associações entre nomes e referências para objetos. O servidor de nomes CORBA é portanto um cliente do serviço de diretório LDAP.

Eficiência e flexibilidade são as vantagens desta abordagem. As associações nome–referência ficarão acessíveis tanto para clientes CORBA (através das interfaces do serviço de nomes) como para clientes LDAP (através da API do LDAP). Atributos descritivos poderão ser adicionados às entradas do diretório que representam associações nome–referência. Clientes LDAP poderão utilizar as facilidades de busca no diretório para obter referências cujas entradas satisfaçam determinadas condições. Essas condições podem envolver o nome associado à referência ou outros atributos presentes nas entradas do diretório.

Abstract

This dissertation presents a solution for the integration of the LDAP directory service and CORBA naming service. We describe a CORBA naming service implementation that stores the bindings between names and object references in an LDAP directory. The CORBA naming server therefore becomes an LDAP directory client.

Efficiency and flexibility are the advantages of this approach. The name–reference bindings become accessible for both CORBA clients (through the naming service interfaces) and LDAP clients (through the LDAP API). Descriptive attributes may be added to the directory entries that represent name–reference bindings. LDAP clients may use of the search facilities in the directory for obtaining references whose entries meet certain conditions. Such conditions may involve the name bound to the reference or other attributes of the directory entries.

Sumário

Lista de Figuras

Lista de Tabelas

Introdução

O LDAP (*Lightweight Directory Access Protocol*) [13] é um serviço de diretório derivado do X.500 e padronizado pela IETF (*Internet Engineering Task Force*). Um serviço de diretório mantém uma base de dados global com informações sobre objetos, fornecendo interfaces para busca, inserção, remoção e atualização dessas informações. O LDAP tem um conjunto de facilidades de busca bastante rico, que permite realizar buscas complexas em todo o diretório ou em partes dele. Possui mecanismos de autenticação, garantindo a segurança das informações no diretório através de controles de acesso, e oferece API padronizadas e bem documentadas.

O serviço de nomes CORBA [6] mapeia nomes em referências de objetos CORBA. Ele é frequentemente comparado a uma lista telefônica, que associa nomes de assinantes a seus correspondentes números de telefone. Trata-se de um serviço bastante simples, que não oferece tantas facilidades de busca quanto um serviço de diretórios. Para obtermos uma referência para um objeto CORBA é necessário que saibamos, de antemão, um nome completo do objeto. Temos de saber completa e exatamente um nome sob o qual o objeto foi registrado no serviço de nomes.

Nosso trabalho tem o objetivo de integrar esses dois serviços, através da construção de um servidor CORBA que implemente o serviço de nomes e empregue um diretório LDAP para armazenar, de forma persistente, as associações entre nomes e referências para objetos CORBA. Clientes que só precisam de “busca por nome” poderão obter referências para objetos através da interface do serviço de nomes CORBA, simples e independente de linguagem de programação. Clientes interessados em facilidades de busca mais avançadas poderão utilizar diretamente a API mais complexa do LDAP.

Armazenadas num diretório LDAP, as associações entre nomes e referências para objetos CORBA ficarão acessíveis para uma gama mais ampla de clientes. Tais informações estarão disponíveis não apenas a clientes no domínio do servidor de nomes CORBA, mas

a todo cliente com acesso ao diretório LDAP.

Nossa implementação foi desenvolvida no contexto do projeto SIDAM [1] (Sistemas de Informações Distribuídos para Agentes Móveis) [1], cujo objetivo é o estudo e a implementação de serviços de informação descentralizados para consulta por agentes móveis.

O restante deste trabalho está dividido em sete capítulos. O capítulo 1 apresenta os principais conceitos dos serviços de diretório, o modelo funcional do X.500, que influenciou fortemente os serviços de diretório desenvolvidos posteriormente, e uma visão histórica do serviço de diretório LDAP, que nasceu como uma simplificação do X.500.

O capítulo 2 descreve as características do serviço de diretório LDAP, inclusive seus modelos funcional, de informação e de nomes. Apresenta o esquema de especificação das *object classes*, as APIs do LDAP e um exemplo simples de aplicação LDAP.

No capítulo 3 revemos alguns conceitos do padrão CORBA que foram aplicados em nossa implementação, tais como a arquitetura OMA (*Object Management Architecture*), as três categorias de interface do ORB (*Object Request Broker*) e o esquema de comunicação de cliente–servidor no CORBA.

O capítulo 4 apresenta os conceitos básicos do serviço de nomes CORBA e sua definição IDL (*Interface Definition Language*), descrevendo brevemente as operações da interface principal oferecida por esse serviço, a interface `CosNaming` e a operação de resolução de nomes.

O capítulo 5 apresenta nosso trabalho de integração do serviço de diretório LDAP com o serviço de nomes CORBA. Ele expõe as razões para integração dos dois serviços, apresenta a arquitetura empregada e descreve os aspectos mais importantes de nossa implementação.

No capítulo 6 examinamos um trabalho relacionado publicado na literatura durante o desenvolvimento dessa dissertação.

E, finalmente, no capítulo 7 apresentamos os resultados obtidos com nossa implementação e algumas de suas vantagens, bem como onde os leitores poderão obter o código fonte do protótipo para utilização e estudos.

[1] O Projeto SIDAM é patrocinado pela FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) – Proc. No. 98/06138-2

Capítulo 1

Serviços de Diretório

A identificação e a localização de pessoas e recursos são questões fundamentais num ambiente de rede. Um serviço de diretório fornece um modo universal de localizar objetos que podem estar situados em qualquer parte da rede. Tais “objetos” podem ser pessoas ou recursos como discos, filas de impressão, endereços de e-mail, máquinas e serviços. Cabe aqui uma analogia com o serviço de ajuda de uma companhia telefônica, que nos informa o número do telefone de uma pessoa se soubermos o nome, ou o endereço, ou mesmo alguns dados dessa pessoa. (Exemplo: a obtenção do telefone de uma pessoa da qual só sabemos o sobrenome e a rua onde mora.) Dado um conjunto de atributos de uma pessoa, servidor ou outro recurso, o serviço de diretório retorna o endereço desse recurso na rede, bem como outras informações sobre ele.

Um diretório pode ser definido como uma base de dados global, que pode estar fisicamente distribuída, mas mantém uma centralização lógica, cujo objetivo é armazenar e fornecer informações sobre objetos. Esses objetos podem ser pessoas, equipamentos, listas de distribuição, endereços de rede, endereços de aplicações, organizações, endereços de correios eletrônicos, etc.

Não há restrições quanto aos objetos que podem ser armazenados num diretório. Embora sejam muito usados por aplicações interessadas em armazenar e recuperar informações sobre recursos de redes, os serviços de diretório não são de modo algum limitados ou especificamente destinados a essa área de aplicação. O administrador de um serviço de diretório pode permitir que qualquer tipo de informação seja colocado no diretório.

Dois aspectos diferenciam os diretórios dos bancos de dados convencionais: o valor

mais alto da razão consultas/alterações e a inexistência de transações atômicas. Os serviços de diretório se distinguem dos serviços de nomes por ofecerem mais recursos de busca (pesquisa) e recuperação de informações.

1.1 O Modelo Funcional do X.500

Uma breve descrição do modelo funcional do serviço de diretório X.500 [15] nos dará o contexto em que surgiu o LDAP e será útil para entendermos algumas simplificações feitas pelos projetistas do LDAP.

O diretório X.500 é um repositório de informações sobre objetos definidos dentro de um contexto qualquer. As informações sobre objetos são armazenadas numa *Directory Information Base* (DIB) estruturada de forma hierárquica, numa árvore denominada *Directory Information Tree* (DIT), conforme a recomendação ITU X.501 [15]. Os nós da DIT representam as entradas da DIB.

Os serviços do X.500 são implementados por dois agentes:

DUA (*Directory User Agent*) — aplicação cliente, através do qual usuários (pessoas ou aplicações) manipulam as diversas entradas do diretório.

DSA (*Directory Service Agent*) — aplicação servidora que gerencia a base de dados do diretório (DIT) e provê os serviços de diretório ao cliente.

Os agentes DUA e DSA são implementados como processos da camada de aplicação do modelo de referência OSI da ISO, conforme a recomendação ITU X.511 [15].

A interação entre um DUA e um DSA é implementada por um protocolo da camada de aplicação do modelo OSI, protocolo esse denominado DAP (*Directory Access Protocol*). De forma análoga, a interação entre dois DSAs é implementada pelo protocolo DSP (*Directory Service Protocol*).

1.2 Surgimento e Evolução do LDAP

No início da década de 80, quando a ISO e o CCITT trabalhavam conjuntamente na definição de uma padronização de um serviço de tratamento de mensagens (a série X.400),

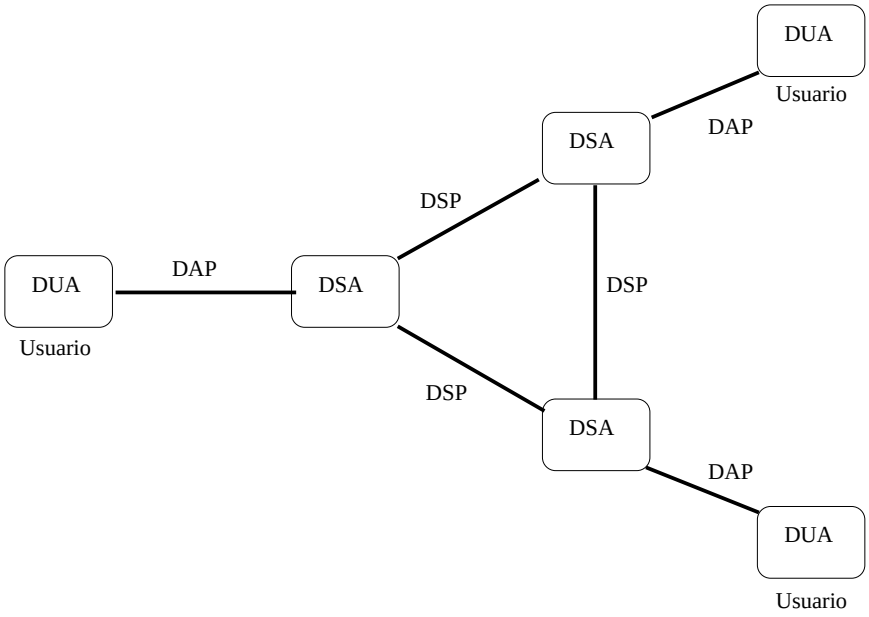

Figura 1.1: Esquema do modelo funcional do serviço de diretório X.500.

ficou clara a necessidade de um esforço paralelo para se desenvolver um serviço de diretório que permitisse tratar melhor aspectos de denominação (*naming*) e de endereçamento.

Os trabalhos dos grupos técnicos constituídos pela ISO e pelo CCITT culminaram com a publicação, em 1988, de um conjunto de oito recomendações da série X.500 (CCITT) e de normas ISO 9594. Essas recomendações e normas tentaram abranger o espectro de possibilidades de ambientes distribuídos em geral, sem ficar restritas ao caso específico do serviço de tratamento de mensagens que as motivou. O resultado foi um serviço de diretório bastante complexo e de custo computacional incompatível com equipamentos menores. Era muito difícil rodar clientes X.500 nos microcomputadores disponíveis na época.

O LDAP, como o nome sugere, foi originalmente desenvolvido como um *front end* de menor custo computacional que o DAP do X.500 (figura 1.1). Seus antecessores foram os protocolos DIXIE[1] (definido na RFC 1249 [12]) e *Directory Assistence Service* — DAS[2] (definido na RFC 1202 [23]), que já rodavam sobre TCP/IP e não sobre a pilha OSI. Através

[1]O protocolo DIXIE foi projetado para ser usado em equipamentos pequenos (exemplo, PCs e Macintoshes) e sem potência computacional ou software necessário para implementar a pilha completa de protocolos OSI. Esse protocolo pode também ser usado por aplicações Internet que precisam de uma interface X.500, com um investimento muito pequeno em código.

[2]Mecanismo pelo qual uma aplicação usuária poderá ter acesso ao X.500 através de um protocolo como o DAP, porém utilizando uma conexão TCP/IP.

do DIXIE ou do DAS, os clientes se comunicavam com um servidor *gateway*, que traduzia as solicitações recebidas via TCP/IP para as correspondentes operações X.500 e as enviava para um DSA, via protocolo DAP sobre a pilha OSI.

Em 1992, o grupo de trabalho *OSI Directory Services* (OSI-DS) da IETF partiu das especificações do DIXIE e do DAS e definiu um protocolo de acesso que funcionava com todas as versões de X.500. Isso resultou no LDAP, que em 1993 foi apresentado como proposta de padrão Internet e em 1995 ganhou o status de *draft standard* [34]. Boa parte desse trabalho foi realizado na Universidade de Michigan, à qual estavam vinculados vários membros do grupo OSI-DS. A primeira implementação do LDAP foi a da Universidade de Michigan.

A falta de suporte para o X.500 e para a pilha de protocolos OSI levou os pesquisadores e desenvolvedores da Universidade de Michigan a criarem um servidor LDAP *standalone*, o `slapd` [28]. Esse grupo disponibilizou os fontes do `slapd` na Internet e criou listas de usuários para divulgar e aperfeiçoar o novo serviço. O desenvolvimento foi acompanhado por usuários e desenvolvedores do mundo inteiro. Com a popularização do `slapd`, o LDAP deixou de ser uma mera alternativa ao DAP do X.500 e ganhou o status de serviço de diretório completo, passando a competir com o X.500. Em dezembro de 1997, a IETF aprovou a versão 3 do LDAP [31] como proposta de padrão Internet para serviços de diretório.

Atualmente várias empresas oferecem produtos LDAP, incluindo Microsoft, Netscape e Novell. A *OpenLDAP Foundation* [2] mantém e disponibiliza uma implementação *open source* de serviço de diretório LDAP, baseada na da Universidade de Michigan, que inclui os seguintes módulos:

- `slapd` — servidor LDAP *standalone*;
- `slurpd` — servidor de replicação LDAP *standalone*;
- bibliotecas que implementam o protocolo LDAP;
- utilitários, ferramentas e exemplos de clientes LDAP.

O desenvolvimento do OpenLDAP prossegue, acompanhando a evolução dos padrões da IETF. Recentemente foi disponibilizado o *release* 2.0.7 do OpenLDAP contemplando a versão 3 do padrão LDAP e algumas facilidades adicionais.

Capítulo 2

O Serviço de Diretório LDAP

Segundo Howes e Smith [13], o LDAP foi concebido efetuando-se as seguintes simplificações na especificação X.500:

Transporte — o LDAP executa diretamente sobre TCP, evitando o "overhead" das camadas superiores da pilha de protocolos OSI.

Representação de Dados — no LDAP a maioria dos elementos de dados são representados como cadeias de caracteres, processadas bem mais facilmente que dados na representação estruturada ASN.1 usada pelo X.500.

Codificação de dados — o LDAP codifica dados para transporte em redes usando uma versão simplificada das mesmas regras de codificação usadas pelo X.500.

Funcionalidade — o LDAP elimina características pouco usadas e operações redundantes do X.500.

A versão 2 do protocolo LDAP é definida nas RFCs 1777 [34], 1778 [10] e 1779 [18], que são hoje *draft Internet standards*. A versão 3 do LDAP é especificada na RFC 2251 [31], cujo status atual é de *proposed Internet standard*. A API C para o LDAP está definida na RFC 1823 [11], que é um documento informativo, sem status de padrão. Uma proposta de API Java [32] foi recentemente publicada como *Internet draft*.

2.1 Características do LDAP

Do ponto de vista conceitual, o serviço de diretório LDAP é composto pelos elementos abaixo:

Modelo de informação — Todas os dados no diretório são armazenados em "entradas", sendo que cada entrada pertence pelo menos a uma "*object class*". Cada entrada tem uma coleção de atributos, que de fato mantém as informações da entrada. As *object classes* a que uma entrada pertence definem os atributos que ela pode ou deve conter. Esse modelo, herdado quase sem alteração do X.500, é extensível: definindo-se novas *object classes*, pode-se adicionar ao diretório qualquer tipo de informação.

Modelo de nomes — Especifica como a informação é organizada e referenciada num diretório LDAP. Os nomes LDAP são hierárquicos. Nomes são sequências de atributos de entradas. Uma entrada raiz tipicamente representa o nome do domínio, da companhia, do estado ou da organização. Entradas para subdomínios, escritórios ou departamentos vem a seguir, muitas vezes seguidas por entradas para indivíduos. Assim como o modelo de informação, o modelo de nomes deriva diretamente do X.500. Diferentemente do X.500, o LDAP não restringe o formato do espaço de nomes, permitindo uma variedade de esquemas flexíveis.

Modelo funcional — Define como clientes podem acessar, manipular e alterar as informações num diretório. O LDAP oferece nove operações básicas: adicionar, apagar, modificar, associar (*bind*), dissociar (*unbind*), buscar, comparar, renomear e abandonar.

Modelo de segurança — Define como a informação num diretório LDAP é protegida contra acessos não autorizados. Um sistema de autenticação extensível permite que clientes e servidores autentiquem suas identidades uns aos outros. Confidencialidade e integridade podem também ser implementadas.

Esquema LDAP — Define que dados podem ser armazenados num dado servidor e como esses dados se relacionam com objetos do mundo real. O esquema pode usar *object classes* padronizadas, como as definidas para países, organizações, grupos e pessoas, ou pode usar novas *object classes*, criadas para atender requisitos específicos de uma instalação.

Protocolo LDAP — Especifica as interações entre clientes e servidores e define os formatos das mensagens de requisição e de resposta.

Interface de Programação — API que oferece um conjunto de funções e definições padronizadas e é usada pelos programas que acessam o diretório.

Formato de Troca de Dados — O *LDAP Data Interchange Format* (LDIF) é um formato textual para representação de entradas e de alterações nessas entradas. Este formato facilita a importação/exportação de dados de/para diretórios X.500 ou diretórios proprietários.

Os modelos do LDAP são descritos com mais detalhes nas seções seguintes.

2.1.1 Modelo de Informação

O modelo de informação do LDAP está centrado na idéia de entrada (figura 2.1). Uma entrada é um conjunto de atributos que é inserido no diretório, geralmente para representar algum objeto do mundo real. Entradas do diretório podem representar pessoas, filas de impressão, objetos CORBA, etc.

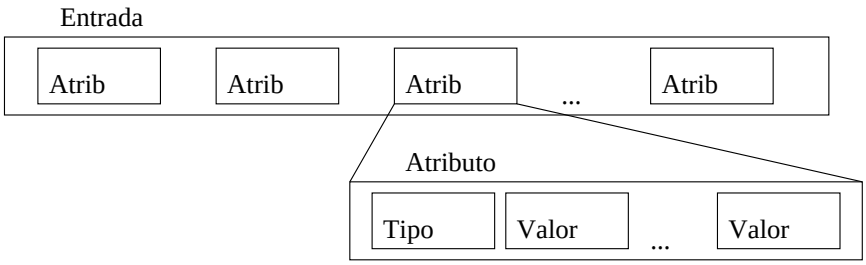

Figura 2.1: Entrada, seus atributos e valores.

Cada atributo tem um tipo e um ou mais valores. O tipo de um atributo determina a chamada "sintaxe" do atributo, ou seja, os valores admissíveis para o atributo. Exemplo: a sintaxe de um atributo tipo `telephoneNumber` admitiria cadeias de dígitos, parênteses e caracteres "+" e "-". O tipo determina também como os valores do atributo são comparados. Exemplo: no caso de um atributo cujos valores são cadeias de caracteres, é feita distinção entre letras maiúsculas e minúsculas. O tipo determina ainda se pode ou não haver mais de um valor para o atributo numa mesma entrada do diretório.

Um exemplo de atributo é `mail`. Pode haver um ou mais valores desse atributo numa entrada do diretório, eles tem de ser cadeias de caracteres ASCII, e na comparação dessas cadeias as letras maísculas e minúsculas são consideradas equivalentes (ou seja, `"scalco@ime.usp.br"` é o mesmo que `"SCALCO@IME.USP.BR"`).

Toda entrada deve ter um atributo `objectClass`, que especifica as *object classes* a que a entrada pertence. A cada *object class* está associado um conjunto de atributos obrigatórios, que devem estar presentes em toda entrada que pertença a essa *object class*, e um conjunto de atributos opcionais, que podem ou não estar presentes numa entrada que pertença à *object class*. Exemplo: se uma entrada pertence à *object class* `alunoUSP` (isto é, tem `alunoUSP` como um dos valores do seu atributo `objectClass`), então ela obrigatoriamente tem um atributo tipo `numeroUSP` e opcionalmente poderá ter um atributo tipo `mail`.

Uma *object class* A pode ser subclasse de outra *object class* B. Neste caso A herda os atributos obrigatórios e os atributos opcionais de B, além de ter seus próprios atributos obrigatórios e opcionais.

2.1.2 Modelo de Nomes

O modelo de nomes pressupõe um diretório organizado em árvore. As entradas do diretório são os nós da árvore. Cada entrada tem um *relative distinguished name* (RDN), que é relativo ao seu nó pai, e um *distinguished name* (DN), que especifica um caminho da raiz até a entrada.

O RDN de uma entrada é formado por um ou mais atributos da entrada. Duas "entradas irmãs" não podem ter o mesmo RDN. A figura 2.2 mostra uma parte de um diretório. Nessa figura os RDNs são formados por um só atributo, o atributo "`dc`" da entrada. Nela aparecem entradas com RDNs "`dc=edu`", "`dc=br`", "`dc=unicamp`" e "`dc=usp`".

O DN de uma entrada é a concatenação dos RDNs da seqüência de nós que começa na própria entrada e vai até uma entrada diretamente subordinada à raiz da árvore. Na figura 2.2, o DN "`dc=usp, dc=br`" denota a entrada no canto inferior direito.

O modelo de nomes do LDAP é semelhante ao de um sistema de arquivos com nomes hierárquicos, como o do Unix. O RDN é análogo ao nome de um arquivo e o DN é análogo ao caminho absoluto do arquivo. Ou seja, o caminho "`/usr/local/include/stdio.h`" corresponde a um DN e o o nome "`stdio.h`" corresponde a um RDN.

Uma diferença entre o modelo de nomes de um sistema de arquivos hierárquico e o

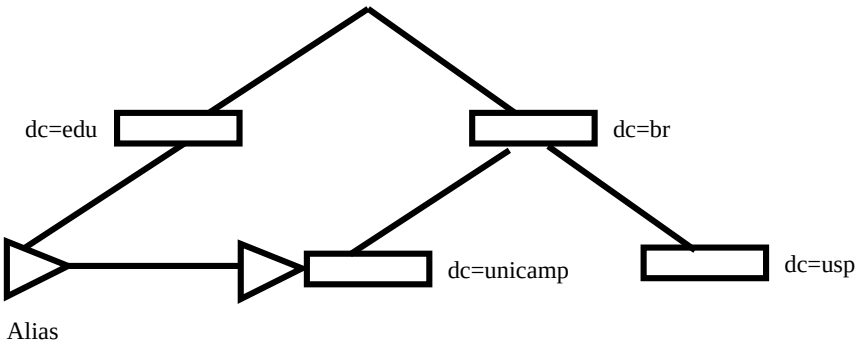

Figura 2.2: Organização das entradas de diretório usando uma estrutura em árvore.

do LDAP é que o primeiro usa uma barra ("/" no Unix, "\" em ambientes Microsoft) e o segundo usa a vírgula como separadora dos componentes do nome. Outra diferença é a ordem dos componentes do nome. No caminho que identifica um arquivo, a seqüência de componentes começa na raiz e prossegue em direção às folhas da árvore. No nome de uma entrada do LDAP, a seqüência começa na entrada e prossegue em direção à raiz do diretório. Se o arquivo "`stdio.h`" fosse uma entrada num diretório LDAP, seu RDN seria algo como "`file=stdio.h, dir=include, dir=local, dir=usr`".

O LDAP permite entradas *alias*, que referenciam outras entradas. A figura 2.2 mostra uma entrada *alias*. A existência de *aliases* quebra a organização hierárquica do espaço de nomes.

O X.500 requer uma hierarquia de entradas definida segundo critérios de distribuição geográfica e organizacional, sendo inclusive padronizados os atributos que podem aparecer nos RDNs de entradas em vários níveis da hierarquia. Entradas diretamente subordinadas à raiz de um diretório X.500 tem RDN formado pelo atributo "`c`" (*country*). Para outras entradas, o RDN é formado por atributos como "`o`" (*organization*), "`ou`" (*organizational unit*) "`cn`" (*common name*), etc. Um possível DN X.500 seria "`cn=Scalco, ou=ime, o=usp, c=br`".

Já o LDAP permite maior flexibilidade, pois não impõe restrições sobre a hierarquia de entradas. Ela pode ser como a do X.500, pode ser um sistema de nomes completamente "achatado" (com todas as entradas diretamente subordinadas à raiz), ou pode ser algum outro sistema hierárquico. Tem sido bastante utilizado um sistema de nomes que se baseia no DNS e utiliza o atributo "`dc`" (*domain component*) para RDNs correspondentes aos componentes de um nome do DNS. Nesse sistema, um possível DN seria "`cn=Scalco, dc=ime, dc=usp, dc=br`".

2.1.3 Modelo Funcional

Funcionalmente o LDAP define nove operações, divididas em três categorias:

Interrogação: "buscar", "comparar". Essas operações são usadas para consultar o diretório e recuperar as informações nele armazenadas.

Alteração: "adicionar", "apagar", "modificar", "renomear" (modificar RDN). Essas operações são usadas para alterar as informações nodiretório.

Autenticação: "associar" (*bind*), "dissociar" (*unbind*), "abandonar". As operações *bind* e *unbind* são de gerenciamento de sessão. A operação "abandonar" permite cancelar uma operação em progresso.

Na categoria interrogação temos duas operações. A mais importante é a "buscar", usada para localizar entradas no diretório, ou em parte previamente definida do diretório, de acordo com o critério de seleção dado por um filtro de busca. No filtro de busca pode-se usar o caracter "*", que faz o papel de coringa, um conjunto de operadores de comparação, inclusive comparação aproximada, e os operadores lógicos *and* ("&"), *or* ("|") e *not* ("!"). Para cada entrada que satisfaça o critério de busca será retornado um conjunto de atributos especificado pelo cliente. O cliente pode especificar também o tamanho limite da resposta a ser retornada pela busca, restringindo o número de entradas devolvidas e o tempo de espera de uma busca.

Há quatro operações na categoria alteração. A operação "modificar" é usada para alterar os atributos e os valores contidos numa entrada. A operação "adicionar" insere uma nova entrada no diretório. A operação "apagar" remove uma entrada do diretório. E a quarta operação, "modificar RDN", é usada para renomear uma entrada.

A categoria autenticação tem três operações. A operação *bind* permite que o cliente se identifique para o serviço de diretório. A operação *unbind* encerra uma sessão com o serviço de diretório. E, finalmente, a operação "abandonar" cancela uma operação previamente emitida e ainda não concluída.

2.1.4 Modelo de Segurança

O modelo de segurança do LDAP é baseado no conceito de lista de controle de acesso. O cliente, quando chama o operação bind, fornece sua identificação (um *distinguished name*)

e credenciais de autenticação (uma senha, por exemplo). Uma lista de controle de acesso é usada para determinar que entradas do diretório o cliente pode ver e que alterações ele tem permissão para fazer. Clientes anônimos (com identidade nula) podem ser aceitos com restrições de acesso, tipicamente para consulta a uma parte das informações no diretório.

O modelo permite que, na abertura de sessão, seja estabelecida uma conexão criptografada com uma chave fornecida por um serviço de distribuição de chaves, como o do Kerberos. Desta forma se garante a confidencialidade das informações que trafegam na rede.

2.1.5 A API do LDAP

As principais funções da API do LDAP são:

- `ldap_search()` — faz busca por entradas do diretório;
- `ldap_compare()` — vê se uma entrada contem um dado valor de atributo;
- `ldap_bind()` — faz autenticação da identidade do cliente;
- `ldap_unbind()` — encerra uma sessão LDAP;
- `ldap_modify()` — altera uma entrada existente;
- `ldap_add()` — adiciona uma nova entrada;
- `ldap_delete()` — apaga uma entrada existente;
- `ldap_modrdn()` — renomeia uma entrada existente;
- `ldap_result()` — obtém os resultados de uma operação prévia.

O protocolo LDAP é completamente assíncrono, de forma que múltiplas operações podem ser disparadas e executadas simultaneamente pelo servidor de diretório. O servidor pode devolver os resultados em qualquer ordem. Cada mensagem é etiquetada com um número, denonominado “ID da mensagem”, que é único para uma dada sessão. Esta característica do LDAP é usada por aplicações complexas, que iniciam várias operações simultâneas, sem abrir múltiplas conexões com o servidor.

Depois que uma operação assíncrona é iniciada, a aplicação deve chamar `ldap_result()` para verificar o status da operação e recuperar os resultados enviados pelo servidor LDAP.

O API do LDAP também oferece um conjunto de chamadas síncronas, cujos nomes terminam com "`_s`". (Exemplo: `ldap_search_s()` é a versão síncrona de `ldap_search()`.) A utilização dessas chamadas são bem mais simples, pois elas enviam uma solicitação e esperam pela resposta do servidor, evitando que a aplicação cliente tenha que se preocupar com o ID das mensagens e outros detalhes. Sua desvantagem é que aplicação fica bloqueada até que o servidor complete a execução da operação requisitada e retorne para a aplicação todas as entradas e solicitações.

A seqüência típica de operações executadas por uma aplicação LDAP é:

1. Inicializar a biblioteca LDAP e obter um *session handle*.
2. Iniciar uma operação LDAP e esperar pelos resultados.
3. Processar os resultados.
4. Fechar a sessão, liberando o *handle* obtido no passo 1.

2.2 Exemplo de Aplicação LDAP

A referência [13] apresenta vários exemplos de aplicações LDAP. Reproduzimos um deles nas figuras 2.3 e 2.4, comentando-o passo a passo. Este exemplo busca todas as entradas contendo um *common name* "`Scalco`" na árvore cuja raiz é "`dc=ime, dc=usp, dc=br`".

```
 1  #include <stdio.h>
 2  #include <lber.h>
 3  #include <ldap.h>
 4  main()
 5  {
 6      LDAP             *ld;
 7      LDAPMessage      *result, *e;
 8      BerElement       *ber;
 9      char             *a, *dn;
10      char             **vals;
11      int              i;
```

Figura 2.3: Exemplo de aplicação LDAP.

```
12      /*       obtém um handle para uma sessão LDAP */
13      if ((ld = ldap_init("localhost", LDAP_PORT)) == NULL) {
14          perror("ldap_init"); return(1);
15      }
16      /*       conecta-se ao diretório como "nobody" */
17      if (ldap_simple_bind_s(ld, NULL, NULL) != LDAP_SUCCESS) {
18          ldap_perror(ld, "ldap_simple_bind_s"); return(1);
19      }
20      /*       busca "Scalco" em "dc=ime, dc=usp, dc=br"     */
21      if (ldap_search_s(ld, "dc=ime, dc=usp, dc=br",
22                        LDAP_SCOPE_SUBTREE, "(cn=Scalco)",
23                        NULL, 0, &result) != LDAP_SUCCESS) {
24          ldap_perror(ld, "ldap_search_s"); return(1);
25      }
26      /* para cada entrada imprime o nome, atributos e valores */
27      for (e = ldap_first_entry(ld, result); e != NULL;
38           e = ldap_next_entry(ld, e)) {
49          if ((dn = ldap_get_dn(ld, e)) != NULL) {
30              printf("dn: %s\n", dn);
31              ldap_memfree(dn);
32          }
33          for (a = ldap_first_attribute(ld, e, &ber);
34               a != NULL; a = ldap_next_attribute(ld, e, ber)) {
35              if ((vals = ldap_get_values(ld, e, a)) != NULL) {
36                  for (i = 0; vals[i] != NULL; i++) {
37                      printf("%s: %s\n",a, vals[i]);
38                  }
39                  ldap_value_free(vals);
40              }
41              ldap_memfree(a);
42          }
43          if (ber != NULL) {
44              ber_free(ber, 0);
45          }
46          printf("\n");
47      }
48      ldap_msgfree(result);
49      ldap_unbind(ld);
50      return(0);
51  }
```

Figura 2.4: Exemplo de aplicação LDAP (continuação).

A chamada na linha 13 retorna um *handle* para uma sessão com o servidor LDAP que está rodando na máquina "`localhost`" e utiliza a porta LDAP padrão (a porta 389) para receber solicitações.

Na linha 17 a aplicação se apresenta como um cliente anônimo (chama uma função *bind* passando `NULL` como identidade). A chamada a uma funcão *bind* é um passo necessário para todas as aplicações.

Nas linhas 21 a 23 o cliente chama uma operação de busca. O primeiro parâmetro da chamada é o *handle* da sessão LDAP. O segundo parâmetro especifica a base da busca, ou seja, em que árvore de diretório LDAP deve ser feita a busca. No caso, é a árvore cuja raiz tem DN "`dc=ime, dc=usp, dc=br`".

As linhas 26 a 47 fazem a impressão do conteúdo (nomes de atributos e seus valores) das entradas encontradas. E as linhas finais fazem a liberaçao do conjunto de resultados retornado pelo servidor LDAP e o encerramento da sessão.

2.3 O Esquema do LDAP

Um dos aspectos positivos do serviço de diretório LDAP é permitir a definição de *object classes* e de tipos de atributos de acordo com as necessidades das aplicações ou usuários do serviço. Cada implementação do LDAP deve oferecer ao administrador do serviço de diretório mecanismos para definir *object classes* adequadas às necessidades específicas de sua organização.

No caso do OpenLDAP, o servidor *slapd*, ao ser inicializado, lê os arquivos de configuração referenciados pelo arquivo principal `slapd.conf`. Alguns destes arquivos possuem o sufixo "`.schema`", como por exemplo o arquivo `core.schema`. É nesses arquivos que estão as definições dos elementos do esquema — *object classes* e tipos de atributos. O conteúdo destes arquivos constitui a *especificação do esquema*.

O administrador do serviço de diretório pode, a qualquer momento, adicionar novos elementos ao esquema. Esse processo é denominado *extensão do esquema* e envolve as seguintes etapas:

- atribuir identificadores de objeto para os novos elementos do esquema;
- escolher um prefixo para os nomes desses elementos;
- criar um arquivo de esquema local;

- definir novos tipos de atributos, se necessário;
- definir novas *object classes*.

Cada *object class* ou tipo de atributo tem um identificador de tipo de objeto (*object identifier*, ou OID) globalmente único. OIDs com o mesmo formato são utilizados em vários contextos, para identificar outros objetos. Tais OIDs, geralmente empregados nos protocolos descritos pela ASN.1, são largamente utilizados pelo *Simple Network Management Protocol* (SNMP).

Os OIDs são escritos como strings decimais separados por pontos representando uma árvore hierárquica. Cada parte de um OID representa um nó da árvore, e cada nó sub-divide a coleção de tipos de objetos em outros tipos mais específicos. Esta hierarquia permite que sejam definidos um grande número de tipos de objetos (quase que infinitamente). Por exemplo, todos os atributos definidos pelo padrão X.500 começam com o OID 2.5.4, o atributo `common name` (`cn`) está associado com o OID 2.5.4.3.

Os OIDs tem uma organização hierárquica que garante a sua unicidade global. A *Internet Assigned Numbers Authority* (IANA) é a entidade responsável pela alocação e registro de "sub-árvores" de OIDs. Para adicionar seus próprios elementos ao esquema LDAP, uma empresa ou instituição deve solicitar que a IANA lhe atribua um OID. Essa empresa poderá então definir OIDs para seus próprios objetos, desde que os identificadores de tipo de objetos por ela definidos tenham como prefixo o OID da empresa. Ao receber um indentificador de tipo de objeto da IANA, a empresa ou instituição fica responsável pela administração de sua sub-árvore de OIDs.

Uma sub-árvore da árvore OID é determinado por um "arco". Um arco individual deve ser associado para as organizações, na qual podem dividí-lo em sub-arcos se assim desejarem. Neste trabalho foi necessário adicionarmos alguns elementos ao esquema LDAP. O sub-arco atribuído pela IANA ao Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP) é `1.3.6.1.4.1.9242`. Tivemos portanto autonomia para alocar OIDs da forma `1.3.6.1.4.1.9242.1.1.1.`*x* para tipos de atributos e OIDs da forma `1.3.6.1.4.1.9242.1.1.2.`*x* para *object classes*. A tabela 2.1 mostra a estrutura da sub-árvore de OIDs definidos pela IANA e o arco de OIDs que alocamos para os identificadores de tipo de objetos do IME-USP.

Além de ter um identificador globalmente único, cada elemento adicionado ao esquema deve ter um nome, um identificador textual mnemônico. O administrador de um serviço de diretório deve escolher nomes que não colidam com os nomes dos elementos padronizados

OID	**Utilização**
1	Identificadores de objetos designados pela ISO
1.3	Organizações reconhecidas pela ISO
1.3.6	Departamento de Defesa dos EUA
1.3.6.1	Internet OIDs - RFC 1065
1.3.6.1.1	Diretórios
1.3.6.1.2	Gerenciamento (*mgmt*)
1.3.6.1.3	Experimentais
1.3.6.1.4	Privados
1.3.6.1.4.1	Empresas - Instituições
1.3.6.1.4.1.9242	Instituto de Matemática e Estatística da USP
1.3.6.1.4.1.9242.1	atividades de pesquisa
1.3.6.1.4.1.9242.1.1	elementos do LDAP
1.3.6.1.4.1.9242.1.1.1	tipos de atributos
1.3.6.1.4.1.9242.1.1.1.*x*	um tipo de atributo
1.3.6.1.4.1.9242.1.1.2	object classes
1.3.6.1.4.1.9242.1.1.2.*x*	uma object class
1.3.6.1.5	Segurança
1.3.6.1.6	SNMPv2
1.3.6.1.7	mail

Tabela 2.1: Organização de nossa sub-árvore de OIDs.

do esquema e que tenham baixa probabilidade de colisão com nomes escolhidos em outras instalações. A convenção usual é empregar prefixos dependentes do nome da organização no DNS, isto é, na organização `ime.usp.br`, recomenda-se usar "`brUspImeUmNome`" em vez de um "`umNome`".

Para cada novo tipo de atributo ou *object class* deve haver uma entrada num arquivo de definição de esquema. O formato dessas entradas está descrito em [20].

Capítulo 3

CORBA

Este capítulo apresenta alguns aspectos do padrão CORBA que são importantes para a compreensão deste trabalho. Em [29] o leitor encontrará uma visão geral de CORBA, e em [9] uma abordagem detalhada.

3.1 A *Object Management Architecture*

O *Object Management Group* (OMG) foi constituído em 1989, com o objetivo de definir padrões que promovam o desenvolvimento e a utilização de aplicações distribuídas portáveis e voltadas para ambientes heterogêneos. Suas atividades tiveram grande aceitação e impacto entre os produtores e usuários de software. Hoje o OMG tem aproximadamente 1000 membros, entre os quais se incluem todas as grandes empresas produtoras de software e as principais empresas e organizações usuárias de software, bem como muitas instituições governamentais, acadêmicas e de pesquisa.

Dentre os primeiros documentos publicados pelo OMG, dois foram especialmente importantes: a especificação de uma arquitetura de gerenciamento de objetos (*Object Management Architecture* - OMA), que pressupõe uma via de comunicação entre clientes e objetos, e a especificação *Common Object Request Broker Architecture* (CORBA), que padroniza essa via de comunicação, o *object request broker* (ORB). A OMA e seu pressuposto central, CORBA, formam uma base arquitetural que é suficientemente rica e flexível para uma ampla gama de sistemas distribuídos.

A OMA define a maneira de se descrever, de modo independente de plataforma, os

objetos distribuídos e as interações entre eles. Para isso, ela usa dois modelos inter-relacionados: um modelo de objetos (*object model*) e um modelo de referência (*reference model*). Enquanto o modelo de objetos define como se descrevem as interfaces de objetos distribuídos por um ambiente heterogêneo, o modelo de referência se preocupa em caracterizar as interações entre tais objetos.

Em seu modelo de objetos, a OMA define um objeto como uma entidade encapsulada, com uma identidade distinta e imutável, cujos serviços são acessíves somente através de interfaces bem definidas. Para usar os serviços de um objeto, os clientes emitem requisições para o objeto. Os clientes não precisam conhecer detalhes de implementação do objeto, nem tampouco a sua localização.

O modelo de referência da OMA agrupa as interfaces dos objetos nas categorias de interfaces apresentadas na figura 3.1. Essa figura ilustra também o papel do ORB como via de comunicação entre clientes e objetos, interligando objetos cujas interfaces podem ser de diferentes categorias.

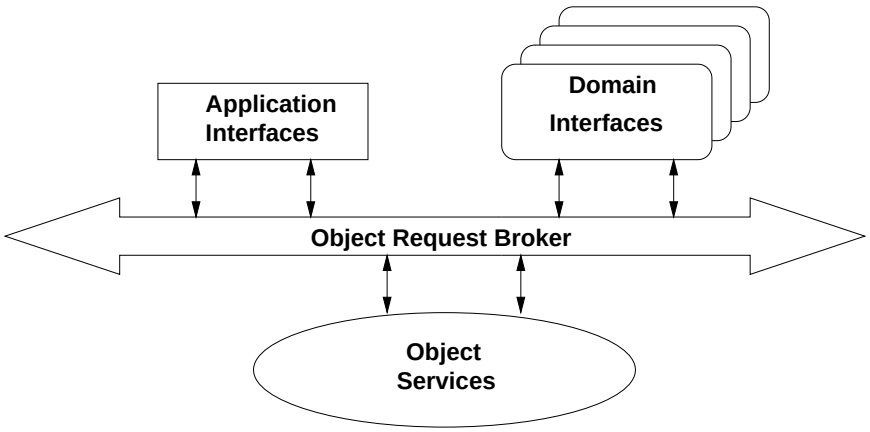

Figura 3.1: Categorias de interfaces da OMA.

O ORB é responsável pelo encaminhamento de requisições dos clientes para os objetos e de respostas dos objetos para os clientes. O encaminhamento é feito de tal forma que nenhuma das partes envolvidas — o cliente e o objeto alvo da requisição — tenha que tomar conhecimento da localização da outra parte (transparência de localização). A comunicação entre as partes ocorre independentemente da plataforma de hardware e do sistema operacional empregado por cada uma delas, bem como da linguagem de programação em que cada parte foi escrita (interoperabilidade). Além de atuar como via de comunicação entre clientes e objetos, o ORB é responsável pela ativação transparente dos objetos que não

estiverem ativos quando forem emitidas requisições para eles.

O modelo de referência da OMA distingue as seguintes categorias de interfaces:

- *Object services* são interfaces independentes de domínio de aplicação, usadas por boa parte das aplicações com objetos distribuídos. Exemplificando: todas as aplicações tem de obter referências para os objetos que elas precisam usar. O OMG especificou dois *object services*, o serviço de nomes e o *trading service*, que vem de encontro a essa necessidade geral. Ambos permitem que aplicações procurem e obtenham referências para objetos. Os *object services* são horizontais e normalmente considerados parte da infraestrutura básica de computação distribuída.

- *Domain interfaces* são similares aos *object services*, porém voltados para domínios de aplicação específicos. Um exemplo de *domain interface* padronizada pelo OMG é o Serviço de Identificação de Pessoas, voltado para aplicações na área médica, em que problema de identificar pacientes tem extrema importância. O OMG definiu *domain interfaces* para uma variedade de áreas de aplicação, como finanças, telecomunicações e sistemas industriais. A multiplicidade de *domain interfaces*, na figura 3.1, reflete a existência de vários conjuntos de *domain interfaces* verticais, cada um deles específico para uma certa área de aplicação.

3.2 Os Elementos de CORBA

Em CORBA, as interfaces dos objetos são especificadas numa *Interface Definition Language* (IDL) puramente declarativa, cujo maior propósito é prover interoperabilidade entre linguagens de programação. Todos os objetos que podem receber requisições através do ORB devem ter interfaces definidas em IDL. Os serviços de qualquer desses objetos podem ser requisitados por clientes escritos numa linguagem diferente da usada para implementar o objeto.

Os clientes e os objetos não são implementados em IDL (nem poderiam ser, pois IDL não é uma linguagem de programação), e sim em linguagens de programação para as quais foram definidos mapeamentos de IDL. O OMG padronizou mapeamentos de IDL para C, C++, Java, Smalltalk, COBOL, Ada, LISP e Python. Existem também mapeamentos não padronizados para outras linguagens, como Delphi.

A figura 3.2 mostra os elementos de CORBA. A partir da definição da interface IDL de

um objeto, um compilador IDL gera código para ser utilizado pelos clientes do objeto (o *stub IDL* da figura 3.2) e código para ser utilizado pelo servidor que implementa o objeto (o esqueleto IDL da figura 3.2). O *stub* e o esqueleto IDL implementam um mecanismo de chamada remota de método.

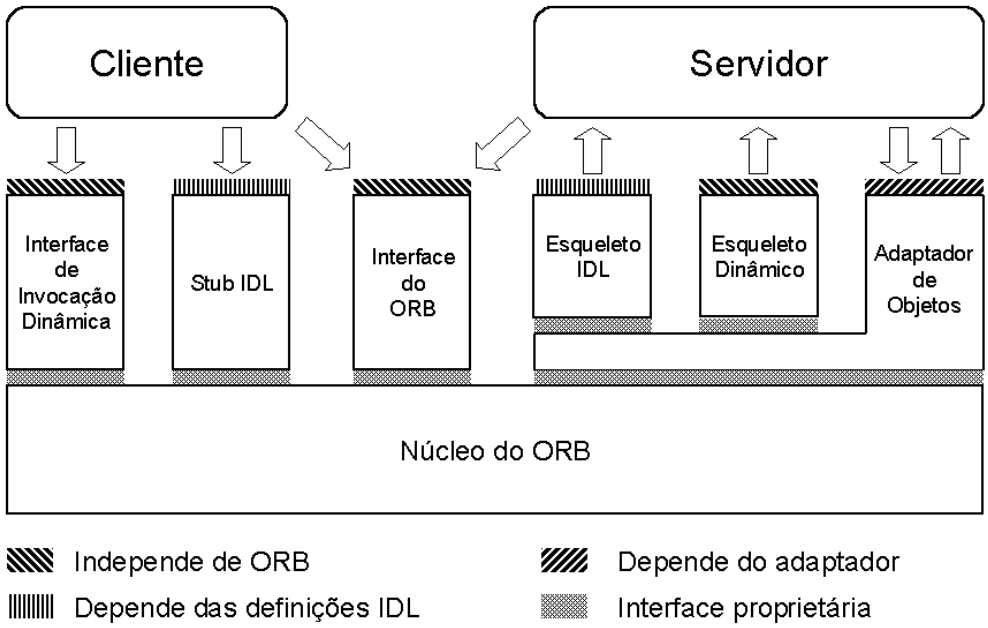

Figura 3.2: Principais elementos de CORBA.

Clientes parecem chamar diretamente métodos de objetos remotos, mas na verdade chamam métodos de *stubs* locais. Um *stub*, quando chamado, envia uma mensagem de requisição para o servidor que implementa o objeto e fica aguardando uma mensagem de resposta. A requisição é encaminhada pelo núcleo do ORB ao servidor que implementa o objeto alvo, passa por um adaptador de objetos (vide figura 3.2) e chega a um esqueleto IDL. O esqueleto chama o método do servente que implementa o objeto alvo e envia os resultados dessa chamada numa mensagem de resposta, a qual passa também pelo adaptador de objetos e é encaminhada pelo núcleo do ORB até o cliente, onde está sendo aguardada pelo *stub*. Este extrai da mensagem de resposta os resultados da chamada remota de método e os retorna para o cliente que fez a chamada.

Além de *stubs* e esqueletos estaticamente gerados a partir definições de interfaces IDL, CORBA oferece também uma interface de invocação dinâmica (DII) e uma interface de esqueleto dinâmico (DSI) ambas representadas na figura 3.2. A DII permite que um "cliente genérico" invoque métodos de objetos cujas interfaces ele desconhecia em tem-

po de compilação. A DSI permite que um "servidor genérico" implemente objetos cujas interfaces ele desconhecia em tempo de compilação.

Na figura 3.2 aparecem também a interface do ORB, cujas operações podem ser chamadas localmente por clientes ou servidores, e um adaptador de objetos, que só existe do lado do servidor. O adaptador de objetos fornece operações de criação de referência para objeto e de registro de objetos, além de oferecer facilidades para ativação dinâmica de objetos.

CORBA define também um repositório de interfaces, não representado na figura 3.2, o qual armazena definições de interfaces IDL. O repositório de interfaces é útil para "clientes genéricos", que usam a DII para invocar métodos de objetos cujas interfaces eles desconheciam em tempo de compilação. Tais clientes consultam o repositório de interfaces para obter, em tempo de execução, informações sobre as interfaces desconhecidas.

3.3 *Object References* e *Proxies*

Para chamar métodos de um objeto CORBA, um cliente precisa ter uma referência para o objeto. A *object reference* funciona como um *handle* que identifica univocamente o objeto e encapsula toda a informação que o ORB necessita para encaminhar uma requisição ao destino correto. CORBA padroniza um formato para transporte e armazenamento de *object references*, denominado *Interoperable Object Reference* (IOR). Esse formato é usado quando uma *object reference* é convertida para cadeia de caracteres, possivelmente para armazenamento num arquivo ou banco de dados, ou quando ela é transmitida como argumento ou resultado de uma chamada remota de método.

A figura 3.3 representa o conteúdo de uma IOR.

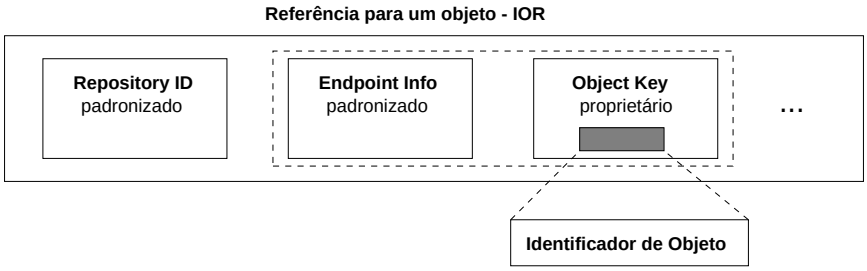

Figura 3.3: Conteúdo de uma IOR.

Uma IOR consiste basicamente de três partes:

- O campo *repository ID* é uma cadeia de caracteres que identifica o tipo da interface do objeto referenciado pela IOR. Esse campo é uma "chave de busca" para o repositório de interfaces (daí o seu nome), com a qual pode-se consultar o repositório e obter a descrição completa da interface.

- O campo *endpoint info* contém as informações necessárias para que o ORB encaminhe requisições para o processo servidor que implementa o objeto referenciado pela IOR. Isso inclui o tipo de protocolo a ser usado e o endereço físico para o estabelecimento de uma conexão. No caso do *Internet Inter-ORB Protocol* (IIOP), o protocolo de uso mais difundido com CORBA, esse campo contém o endereço IP ou o nome da máquina servidora no DNS e a porta TCP a ser utilizada.

- O campo *object key* contém as informações necessárias para encaminhar ao objeto alvo uma requisição que já chegou ao processo servidor. Em outras palavras, este campo é uma chave local a um processo servidor. Ele identifica um dos objetos implementados pelo servidor. Ao contrário dos campos *repository ID* e *endpoint info*, cujos formatos são padronizados pelo OMG, o formato da *object key* é proprietário e varia entre ORBs. Todos os ORBs, entretanto, permitem que uma aplicação servidora, ao criar uma IOR, especifique um identificador de objeto (*object id*), o qual será de alguma forma embutido no campo *object key* da IOR. O *object id* identifica um dos objetos gerenciados por um adaptador de objetos. Cada requisição que chega a um processo servidor é encaminhada para algum dos adaptadores de objetos existentes no servidor. O *object id* é então usado para direcioná-la ao objeto alvo.

Quando uma *object reference* é recebida por um cliente, um módulo da biblioteca do ORB agregado ao cliente instancia um objeto *proxy* no espaço de endereçamento do cliente. O *proxy* é um objeto local que oferece ao cliente uma interface idêntica à do objeto remoto identificado pela referência. A figura 3.4 mostra um proxy local para um objeto remoto.

Se o cliente for implementado numa linguagem orientada a objetos, como C++, Java ou Smalltalk, o *proxy* será uma instância de uma *stub class*, tipicamente gerada por um compilador IDL. Se o cliente não for implementado numa linguagem orientada a objetos, o *proxy* será uma estrutura que inclui um conjunto de apontadores para funções. A definição dessa estrutura e as definições das funções apontadas por ela são tipicamente geradas por um compilador IDL.

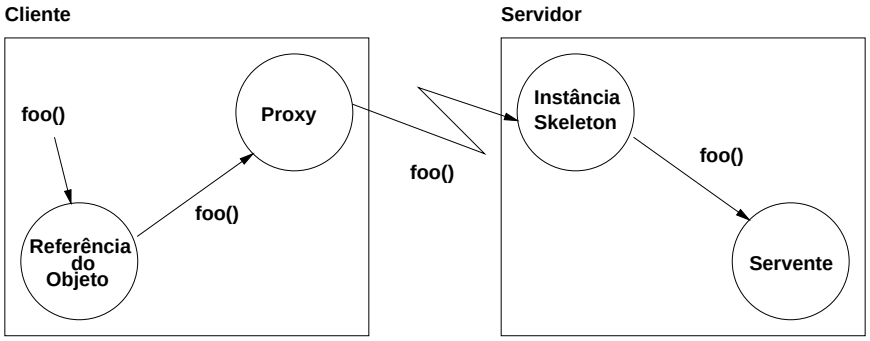

Figura 3.4: Proxy local para um objeto remoto.

Em qualquer dos casos, o cliente vê a *object reference* como um apontador ou referência para um *proxy* local. O *proxy* que recebe chamadas do cliente e as converte em requisições para um objeto servente, o qual é possivelmente implementado por um servidor remoto, como mostra a figura 3.4.

3.4 O *Portable Object Adapter*

O adaptador de objetos padronizado por CORBA é o *Portable Object Adapter* (POA). Este componente de CORBA só existe do lado do servidor. Ele é responsável pela criação de referências para objetos CORBA, pela ativação de objetos e pelo direcionamento de cada requisição para o servente que implementa o objeto alvo.

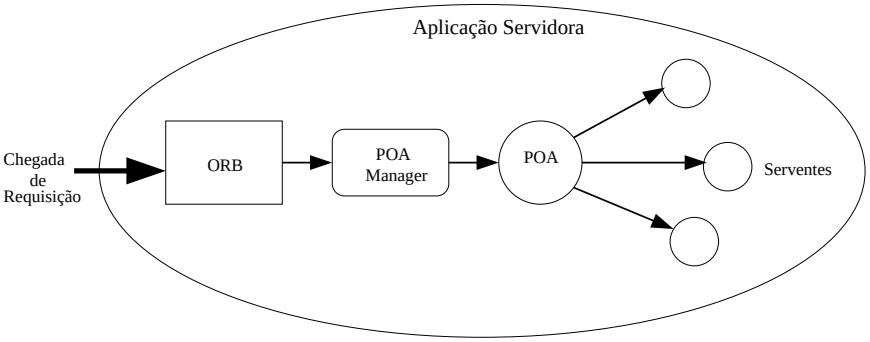

Figura 3.5: O papel do POA no direcionamento de requisições.

A figura 3.5 ilustra o papel do POA no direcionamento das requisições que chegam a um servidor CORBA. Cada requisição é recebida por uma biblioteca do ORB incorporada à aplicação servidora. A biblioteca do ORB repassa a requisição para um *POA Manager*,

que atua essencialmente como uma válvula de controle do fluxo de requisições entrando no POA. O POA repassa a requisição para um objeto servente que "encarna" o objeto alvo da requisição.

O modo de operação de um POA é determinado por um conjunto de políticas (*policies*) especificadas na criação do POA. Uma aplicação servidora pode criar um ou mais POAs com modos de operação adequados às suas necessidades. A tabelas 3.1 e 3.2 mostram as políticas que definem o comportamento de um POA e os valores possíveis para cada política. Vide [9] para mais detalhes sobre o POA e suas políticas, bem como sobre as implicações de várias escolhas de valores de políticas.

Política	**Descrição**
`Lifespan`	Especifica o tempo de validade das referências para objetos pertencentes a este POA. **TRANSIENT** – As referências são válidas apenas enquanto o processo servidor que as criou estiver rodando. **PERSISTENT** – As referências permanecem válidas mesmo depois que o servidor encerrou sua execução. O default para o Root POA é TRANSIENT.
`Object Id` `Uniqueness`	Especifica se os serventes neste POA tem *object ids* únicos. **UNIQUE_ID** – Os serventes neste POA possuem um único *object id*. **MULTIPLE_ID** – Os serventes neste POA podem podem ter um ou mais *object id*s. O default para o Root POA é UNIQUE_ID.
`Id` `Assignment`	Especifica se os *object ids* deste POA são gerados pela aplicação ou pelo POA. **USER_ID** – Os objetos tem *object ids* escolhidos pela aplicação. **SYSTEM_ID** – Os objetos tem *object ids* escolhidos pelo POA. O default para o Root POA é SYSTEM_ID.
`Servant` `Retention`	Especifica se o POA manterá os serventes ativos em seu Mapa de Objetos Ativos. **RETAIN** – O POA manterá os serventes em seu Mapa de Objetos Ativos. **NO_RETAIN** – O POA não usará o Mapa de Objetos Ativos. Neste caso deve-se selecionar também USE_DEFAULT_SERVANT ou USE_SERVANT_MANAGER. O default para o Root POA é RETAIN.

Tabela 3.1: Políticas para configuração dos Adaptadores de Objeto.

Política	**Descrição**
`Activation`	Especifica se o POA suporta ativação implícita dos objetos pertencentes a este POA. **IMPLICIT_ACTIVATION** – Este POA oferece suporte a ativação implícita. **NO_IMPLICIT_ACTIVATION** – Este POA não oferece suporte a ativação implícita. O default para o Root POA é IMPLICIT_ACTIVATION.
`Request Processing`	Especifica como as requisições são processadas no POA. **USE_ACTIVE_OBJECT_MAP_ONLY** – Se o POA não encontrar o *object id* no seu mapa, ele retorna uma exceção. **USE_DEFAULT_SERVANT** – Se o POA não encontrar o o *object id* no seu mapa, ou se estiver com a política NON_RETAIN, ele encaminha a chamada para um servente default. **USE_SERVANT_MANAGER** – Se o POA não encontrar o *object id* no seu mapa, ou se estiver com a política NON_RETAIN, ele encaminha a chamada para um *servant manager*. O default para o Root POA é USE_ACTIVE_OBJECT_MAP_ONLY.
`Thread`	Especifica o modelo de *threading* usado pelo POA. **ORB_CTRL_MODEL** – O POA é responsável pela associação de *threads* a requisições. **SINGLE_THREAD_MODEL** – O POA processará as requisições sequencialmente. O default para o Root POA é ORB_CTRL_MODEL.

Tabela 3.2: Políticas para configuração dos Adaptadores de Objeto - continuação.

Capítulo 4

O Serviço de Nomes CORBA

Dentre os *Object Services* [6] padronizados pelo *Object Management Group* (OMG), o serviço de nomes CORBA é um dos mais simples e básicos. Sua especificação foi publicada pela primeira vez em setembro de 1993 e foi significativamente revisada em outubro de 1998.

O serviço de nomes mantém associações entre nomes e referências para objetos CORBA, permitindo que objetos tenham nomes significativos do ponto de vista de uma aplicação. Os clientes dos objetos podem usar esses nomes, em vez de nomes menos amigáveis, como referências para objetos CORBA convertidas para cadeias de caracteres.

4.1 Conceitos

O serviço de nomes faz o mapeamento de nomes em referências para objetos CORBA. Uma associação nome–referência é denominada *name binding*. A mesma referência pode estar associada a vários nomes, mas cada nome identifica exatamente uma referência.

Um contexto de nomes é um objeto CORBA que contém *name bindings*. Em outras palavras, cada contexto de nomes implementa uma tabela que mapeia nomes em referências para objetos. Um nome nessa tabela pode estar associado tanto a uma referência para um objeto definido por uma aplicação como a uma referência para outro contexto implementado pelo serviço de nomes. Isto significa que contextos podem formar uma hierarquia análoga à hierarquia de diretórios num sistema de arquivos: contextos correspondem a diretórios que podem conter arquivos (objetos de aplicação) ou outros diretórios (outros

contextos).

Contextos de nomes e *name bindings* formam um grafo dirigido denominado grafo de nomes. A figura 4.1 mostra um grafo de nomes. Os nós brancos representam contextos de nomes e os nós escuros representam os objetos de aplicação. Contextos de nomes podem aparecer como nós internos ou como nós folha, mas objetos de aplicação só podem aparecer como nós folha. Os arcos que saem de um contexto de nomes representam os *name binding* nesse contexto. Para cada associação nome–referência temos um arco que é rotulado pelo nome e aponta para o objeto denotado pela referência.

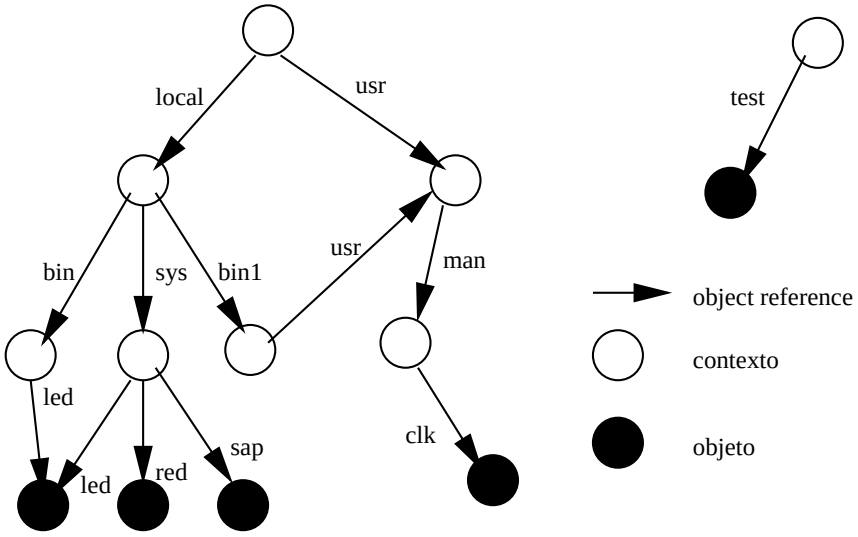

Figura 4.1: Um grafo de nomes.

Fixado um contexto inicial, pode-se percorrer o grafo de nomes até um dado nó alcançável a partir do contexto inicial. A seqüência de nomes nos arcos percorridos forma um *pathname*, um nome composto que identifica univocamente o nó alcançado. Exemplo: considerando-se como contexto inicial o nó superior esquerdo da figura 4.1, o nome composto "`local/bin/led`" identifica o nó inferior esquerdo. O nome composto "`local/sys/led`" também identifica esse nó. (Um objeto pode ter múltiplos *pathnames*.)

4.2 Definição IDL do Serviço de Nomes

A definição IDL do serviço de nomes está no arquivo `CosNaming.idl`. Esse arquivo contém um único módulo, denominado `CosNaming`. Além de definir vários tipos

auxiliares, o módulo `CosNaming` define a interface `NamingContext`, que é a principal interface implementada pelo serviço de nomes. Nas figuras 4.2, 4.3 e 4.4 listamos o conteúdo do arquivo `CosNaming.idl`. Omitimos a interface `NamingContextExt`, que foi incluída no módulo `CosNaming` em 1998.

Algumas observações:

- Um `Name` é um nome composto, um caminho que identifica um objeto a partir de um contexto inicial. Um `Name` é uma seqüência de `NameComponents`.
- Cada `NameComponent` consiste num par de cadeias de caracteres: um `id` e um `kind`.

```
#ifndef _CosNaming_IDL_
#define _CosNaming_IDL_
#pragma prefix "omg.org"
module CosNaming {

    typedef string Istring;

    struct NameComponent {
        Istring id;
        Istring kind;
    };

    typedef sequence<NameComponent> Name;

    enum BindingType {
        nobject,
        ncontext
    };

    struct Binding {
        Name binding_name;
        BindingType binding_type;
    };

    typedef sequence<Binding> BindingList;

    interface BindingIterator;
```

Figura 4.2: Interfaces do serviço de nomes.

```
interface NamingContext {

    enum NotFoundReason { missing_node, not_context,  not_object };

    exception NotFound { NotFoundReason why; Name rest_of_name; };
    exception CannotProceed { NamingContext ctx; Name rest_of_name; };
    exception InvalidName {};
    exception AlreadyBound {};
    exception NotEmpty {};

    void bind(in Name n, in Object obj)
        raises(NotFound, CannotProceed, InvalidName, AlreadyBound);

    void rebind(in Name n, in Object obj)
        raises(NotFound, CannotProceed, InvalidName);

    void bind_context(in Name n, in NamingContext nc)
        raises(NotFound, CannotProceed, InvalidName, AlreadyBound);

    void rebind_context(in Name n, in NamingContext nc)
        raises(NotFound, CannotProceed, InvalidName);

    Object resolve(in Name n)
        raises(NotFound, CannotProceed, InvalidName);

    void unbind(in Name n)
        raises(NotFound, CannotProceed, InvalidName);

    NamingContext new_context();

    NamingContext bind_new_context(in Name n)
        raises(NotFound, AlreadyBound, CannotProceed, InvalidName);

    void destroy() raises(NotEmpty);

    void list(in unsigned long how_many,
              out BindingList bl, out BindingIterator bi);

};
```

Figura 4.3: Interfaces do serviço de nomes (continuação).

```
        interface BindingIterator {

            boolean next_one(out Binding b);

            boolean next_n(in unsigned long how_many, out BindingList bl);

            void destroy();
        };

    };
    #endif /* !_CosNaming_IDL_ */
```

Figura 4.4: Interfaces do serviço de nomes (continuação).

O campo `id` é considerado a "parte principal" do `NameComponent`. O propósito original do campo `kind` era descrever de algum modo o objeto associado ao `NameComponent`. Hoje, entretanto, muitos consideram que a distinção entre `id` e `kind` foi um erro no projeto do serviço de nomes e recomendam que só se coloque a cadeia vazia no campo `kind`.

- Cada contexto de nomes guarda somente associações entre `NameComponents` e referências para objetos CORBA. Já um nome composto (`Name`) não é armazenado num único contexto de nomes, pois ele identifica um caminho de um contexto inicial até um objeto. Embora exista uma associação entre o nome composto e esse objeto, as informações sobre essa associação ficam distribuídas ao longo dos contextos do caminho.

4.3 Operações da Interface `NamingContext`

O serviço de nomes implementa objetos do tipo `NamingContext`. No arquivo apresentado nas figuras 4.2, 4.3 e 4.4 vimos a definição da interface `NamingContext`. Abaixo descrevemos brevemente o que faz cada uma das operações dessa interface:

- `bind` — estabelece uma nova associação entre um `Name` e um objeto de aplicação, tomando como ponto de partida da associação o `NamingContext` alvo;
- `rebind` — como `bind`, mas não lança uma exceção se o `Name` já estiver associado a

um objeto;

- `bind_context` — estabelece uma nova associação entre um `Name` e um objeto do tipo `NamingContext`, tomando como ponto de partida da associação o `NamingContext` alvo;
- `rebind_context` — como `bind_context`, mas não lança uma exceção se o `Name` já estiver associado a um objeto;
- `resolve` — retorna uma referência para o objeto CORBA associado a um `Name` a partir do `NamingContext` alvo;
- `unbind` — desfaz a associação entre um `Name` (interpretado a partir do `NamingContext` alvo) e um objeto CORBA;
- `new_context` — cria um novo `NamingContext`;
- `bind_new_context` — cria um novo `NamingContext` e o associa a um `Name`, considerando como ponto de partida da associação o `NamingContext` alvo;
- `destroy` — destrói o `NamingContext` alvo;
- `list` — retorna o conjunto de associações no `NamingContext` alvo.

4.4 Resolução de Nomes

A maioria das operações da interface `NamingContext` recebe um `Name` como parâmetro. Nessas operações o `Name` é interpretado no contexto alvo da chamada, isto é, tomando-se como ponto de partida o `NamingContext` sobre o qual a operação foi invocada.

A interpretação de um `Name` num dado contexto é denominada resolução do nome. O serviço de nomes busca nesse contexto uma associação que envolva o primeiro `NameComponent` do nome. Se existir tal associação, o `NameComponent` estará ligado a uma referência para outro contexto ou para um objeto de aplicação. Caso o nome tenha outros `NameComponents`, a referência associada ao primeiro componente deve apontar para outro contexto, no qual o serviço de nomes buscará uma associação envolvendo o segundo componente do nome, e assim sucessivamente. O processo de resolução terminará

quando o último componente do `Name` for resolvido e produzirá como resultado a referência associada ao último componente.

Para uma operação op que é invocada sobre um contexto de nomes ctx e usa um `Name` com componentes c_1, c_2, ... , c_n, a resolução do nome é definida recursivamente por

$$ctx \rightarrow op(c_1/c_2/\ldots/c_n,\ldots) \equiv ctx \rightarrow resolve(c_1) \rightarrow op(c_2/\ldots/c_n,\ldots)$$

A operação op pode ser qualquer uma das operações da interface `NamingContext` que recebe um `Name` como parâmetro de entrada: `bind`, `rebind`, `resolve`, `unbind`, etc.

Um contexto de nomes pode conter (e tipicamente contém) associações envolvendo objetos CORBA implementados por outros servidores. Em particular, um `NamingContext` residente em um dado servidor de nomes pode conter referências para `NamingContexts` residentes em outros servidores de nomes. No caso geral, a resolução de nomes é um processo distribuído, pois os contextos percorridos para se resolver os componentes de um nome podem estar distribuídos por vários servidores.

Capítulo 5

Integração do LDAP com o Serviço de Nomes CORBA

Implementações do serviço de nomes CORBA precisam armazenar, de modo persistente, as associações nome–referência existentes nos contextos de nomes. Podem, por exemplo, guardar essas associações em arquivos do sistema operacional subjacente. Ou usar um sistema gerenciador de bancos de dados para esse propósito.

Construimos uma implementação do serviço de nomes CORBA que armazena as associações nome–referência num diretório LDAP. O servidor de nomes CORBA é portanto um cliente do serviço de diretório LDAP.

5.1 Motivação

Eficiência. Um diretório LDAP é um tipo de banco de dados especialmente otimizado para consultas. As associações nome–referência num `NamingContext` CORBA são utilizadas principalmente por uma operação de consulta, a resolução de nomes.

Flexibilidade. As associações nome–referência ficarão acessíveis tanto para clientes CORBA como para clientes LDAP. Clientes CORBA usarão as interfaces do serviço de nomes. Clientes LDAP poderão inserir mais atributos nas entradas do diretório que representam associações nome–referência. Poderão também utilizar facilidades de busca em diretórios para obter referências cujas entradas satisfaçam determinadas condições. Essas condições podem ser expressas em termos do `NameComponent` ao

qual a referência está associada, ou em termos de atributos adicionais presentes nas entradas do diretório.

5.2 Arquitetura de Três Camadas

A arquitetura utilizada é de três camadas (*three-tier*): uma aplicação cliente CORBA utilizará o serviço de nomes através das interfaces definidas pelo OMG. A comunicação entre o cliente CORBA e o servidor de nomes será feita através do protocolo empregado no ambiente CORBA, tipicamente o IIOP. O servidor de nomes mantém as associações nome–referência num diretório LDAP, e interage com o diretório através da API do LDAP. A comunicação entre o servidor de nomes CORBA e o servidor de diretório é feita através do protocolo LDAP. A figura 5.1 ilustra essa arquitetura.

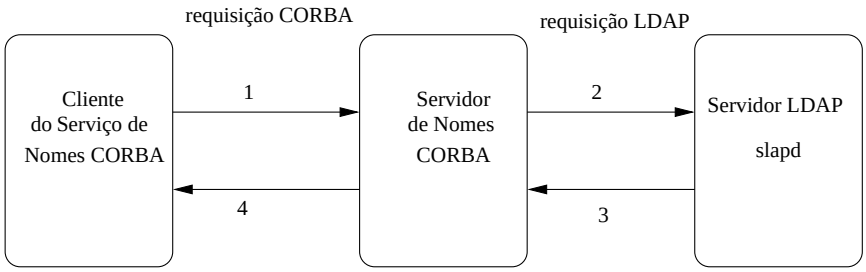

Figura 5.1: As três camadas da arquitetura de integração LDAP-CORBA.

5.3 Aspectos Importantes da Implementação

Os seguintes aspectos da implementação do servidor de nomes merecem ser discutidos:

1. A definição do formato das entradas do diretório LDAP que conterão informações sobre contextos de nomes e associações nome–referência.

2. A modularização do código do servidor, de modo que boa parte dele seja independente do serviço de armazenamento persistente no qual são mantidos os contextos de nomes e as associações nome–referência. Com isso uma parte significativa do código do servidor não dependerá do LDAP. Essa parte do código poderá ser reutilizada para se implementar um servidor de nomes que mantém num banco de dados os contextos e as associações nome–referência.

3. A utilização de um esquema de *caching* de modo a evitar que toda chamada ao serviço de nomes ocasione uma requisição ao servidor usado para armazenamento persistente de informações, no caso o LDAP.

4. A efetiva utilização do POA por parte do servidor de nomes, de modo que as referências CORBA para `NamingContexts` sejam persistentes.

Esses aspectos serão abordados a seguir, nas seções 5.4–5.7.

5.4 Object Classes para Contextos e Bindings

Graças à extensibilidade do modelo de informação do LDAP, podemos definir *object classes* específicas para as entradas do diretório que contém contextos de nomes e associações nome–referência. A figura 5.2 mostra a definição da primeira dessas *object classes*, que denominamos `corbaCosContext`[1]. Esta *object class* especifica o formato das entradas do diretório correspondentes aos `NamingContexts` do serviço de nomes CORBA.

```
objectclass ( 1.3.6.1.4.1.9242.1.1.2.1
        NAME 'corbaCosContext'
        DESC 'LDAP entry for a CORBA naming context'
        SUP top
        STRUCTURAL
        MUST ( corbaId $ corbaBase ) )
```

Figura 5.2: A *object class* `corbaCosContext`.

Uma entrada com esse formato é mantida para cada contexto de nomes armazenado no diretório. Ela tem a função de registrar a existência de um dado contexto de nomes e de associar um identificador a esse contexto. Por ser derivada da *object class* `top`[2], `corbaCosContext` herda de `top` o atributo obrigatório `objectClass`. Alem deste, um `corbaCosContext` tem somente dois atributos, também obrigatórios. O primeiro é um `corbaId`, uma cadeia de caracteres numéricos que distingue este contexto dos demais contextos implementados pelo mesmo servidor de nomes. O segundo é o atributo `corbaBase`,

[1] O verdadeiro nome dessa *object class* é `brUspImeCorbaCosContext`, escolhido conforme a convenção apresentada ao final da seção 2.3. Em benefício da legibilidade omitimos o prefixo `brUspIme` na descrição deste e dos demais elementos que adicionamos ao esquema do LDAP.

[2] A *object class* `top` é definida pela RFC 2256 [30] e tem somente o atributo obrigatório `objectClass`.

uma cadeia de caracteres que identifica o servidor de nomes que implementa este contexto. A função deste atributo é possibilitar que mais de um servidor de nomes CORBA utilize o mesmo servidor LDAP para armazenamento de associações nome–referência. A figura 5.3 ilustra o caso em que vários servidores de nomes acessam o mesmo diretório LDAP. O atributo `corbaBase` distingue as entradas dos diferentes servidores nos diretório.

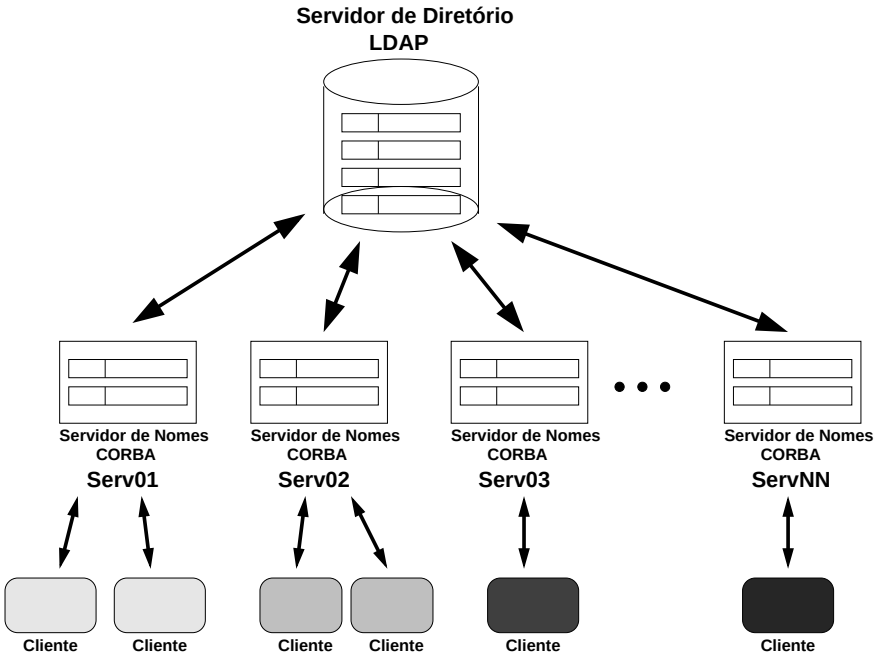

Figura 5.3: Vários servidores de nomes CORBA compartilhando um diretório LDAP.

Uma entrada `corbaCosContext` é adicionada ao diretório quando se cria um novo `NamingContext` e é apagada quando este é destruído. A figura 5.4 apresenta as definições dos atributos desta *object class*.

A figura 5.5 mostra a definição da segunda *object class* que adicionamos ao esquema do LDAP. Essa *object class*, denominada `corbaCosBinding`, especifica o formato das entradas que contém associações nome–referência. Os atributos obrigatórios `corbaName` e `corbaKind` são cadeias de caracteres que armazenam um `NameComponent` do serviço de nomes. O atributo `corbaName` corresponde ao campo `id` e o atributo `corbaKind` ao campo `kind` do `NameComponent`. O atributo obrigatório `corbaIor` é uma cadeia de caracteres com a referência para o objeto CORBA (contexto ou objeto de aplicação) associado ao `NameComponent`. O atributo `corbaType`, também obrigatório, é um valor booleano que indica se `corbaIor` se refere a um contexto ou objeto de aplicação. O atributo obrigatório

```
attributetype ( 1.3.6.1.4.1.9242.1.1.1.1
        NAME 'corbaId'
        DESC 'Identifies a CORBA naming context'
        EQUALITY numericStringMatch
        ORDERING caseIgnoreOrderingMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.36{4} )

attributetype ( 1.3.6.1.4.1.9242.1.1.1.2
        NAME 'corbaBase'
        DESC 'Identifies a CORBA naming server'
        EQUALITY caseExactMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )
```

Figura 5.4: Definições de atributos para a *object class* `corbaCosContext`.

```
objectclass ( 1.3.6.1.4.1.9242.1.1.2.2
        NAME 'corbaCosBinding'
        DESC 'LDAP entry for a CORBA name-objref binding'
        SUP top
        STRUCTURAL
        MUST ( corbaName $ corbaKind $ corbaIor $
               corbaType $ corbaCtxId $ corbaBase )
        MAY  ( cn $ description ) )
```

Figura 5.5: A *object class* `corbaCosBinding`.

`corbaCtxId` tem a função de identificar o contexto de nomes ao qual esta associação nome–referência se vincula. Ele contém o valor do atributo `corbaId` de uma entrada `corbaCosContext`. O último atributo obrigatório, `corbaBase`, tem a mesma função de seu homônimo na *object class* `corbaCosContext`: ele identifica o servidor de nomes responsável por esta associação nome–referência.

Como atributos opcionais da *object class* `corbaCosBinding`, sugerimos `cn` (*common name*) e `description`, ambos com o formato de cadeia de caracteres. No *common name* pode-se armazenar um nome adicional para o objeto CORBA referenciado pela entrada `corbaCosBinding`. O `description` pode conter uma breve descrição desse objeto, um comentário sobre a sua funcionalidade, versão, etc. Outros atributos opcionais poderão ser adicionados se necessário.

A figura 5.6 apresenta as definições dos atributos introduzidos pela *object class* `corbaCosBinding`.

A terceira e última *object class* que definimos descreve o formato de uma entrada especial, cujo único propósito é armazenar, de modo persistente, o valor de um contador. O servidor de nomes usa esse contador para gerar o próximo identificador de contexto, ou seja, o campo `corbaId` da próxima entrada `corbaCosContext` que ele adicionará ao diretório. A figura 5.7 apresenta a definiçao dessa *object class*, que denominamos `corbaNextCtxId`.

Como anteriormente, `corbaBase` é um atributo obrigatório que especifica um certo servidor de nomes. O atributo obrigatório `corbaNextId` é o próximo identificador de contexto que será utilizado por esse servidor de nomes. A figura 5.8 mostra a definição desse atributo.

É interessante notar que o esquema de *object classes* que definimos é análogo a um esquema relacional. A figura 5.9 mostra um esquema relacional que poderia ser utilizado para armazenar contextos de nomes e *bindings* num banco de dados. Todas as tabelas desse esquema tem chaves primárias compostas. A tabela `corbaCosContext` tem chave primária formada por suas duas colunas. O mesmo acontece com a tabela `corbaNextCtxId`. A chave primária da tabela `corbaCosContext` é constituída pelas colunas `corbaName`, `corbaKind`, `corbaCtxId` e `corbaBase`. Estas duas últimas colunas formam uma chave estrangeira, que referencia a tabela `corbaCosContext`.

```
attributetype ( 1.3.6.1.4.1.9242.1.1.1.3
        NAME 'corbaName'
        DESC 'The id field of a CORBA name component'
        EQUALITY caseExactMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )

attributetype ( 1.3.6.1.4.1.9242.1.1.1.4
        NAME 'corbaKind'
        DESC 'The kind field of a CORBA name component'
        EQUALITY caseExactMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )

attributetype ( 1.3.6.1.4.1.9242.1.1.1.5
        NAME 'corbaIor'
        DESC 'The IOR bound to the name'
        EQUALITY caseIgnoreIA5Match
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )

attributetype ( 1.3.6.1.4.1.9242.1.1.1.6
        NAME 'corbaType'
        DESC 'Tells if the IOR refers to a naming context or to an app object'
        EQUALITY booleanMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.7 )

attributetype ( 1.3.6.1.4.1.9242.1.1.1.7
        NAME 'corbaCtxId'
        DESC 'Identifies the naming context a binding belongs to'
        EQUALITY numericStringMatch
        ORDERING caseIgnoreOrderingMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.36{4} )
```

Figura 5.6: Definições de atributos para a *object class* `corbaCosBinding`.

```
objectclass ( 1.3.6.1.4.1.9242.1.1.2.3
        NAME 'corbaNextCtxId'
        DESC 'corbaId of next corbaCosContext entry for CORBA server
              corbaBase'
        SUP top
        STRUCTURAL
        MUST ( corbaNextId $ corbaBase ) )
```

Figura 5.7: A *object class* `corbaNextCtxId`.

```
attributetype ( 1.3.6.1.4.1.9242.1.1.1.8
        NAME 'corbaNextId'
        DESC 'corbaId for next corbaCosContext entry'
        EQUALITY numericStringMatch
        ORDERING caseIgnoreOrderingMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.36{4} )
```

Figura 5.8: Atributo para a *object class* `corbaNextCtxId`.

```
table corbaCosContext( corbaId, corbaBase )

table corbaCosBinding ( corbaName, corbaKind, corbaIor,
                        corbaType, corbaCtxId, corbaBase, cn, description )

table corbaNextCtxId ( corbaNextId, corbaBase )
```

Figura 5.9: Esquema relacional para armazenamento de contextos e *bindings*.

5.5 O Módulo de Acesso ao Repositório Persistente

Este módulo oferece uma interface de acesso a um "repositório abstrato" que contém contextos e associações nome–referência. Ele oculta do restante do servidor de nomes a API e os detalhes do serviço de armazenamento persistente empregado. Esta abordagem tem duas vantagens:

1. Ela permite que o restante do código do servidor de nomes seja independente do LDAP e possa ser reutilizado para implementar servidores de nomes baseados em outros serviços de armazenamento.

2. Ela evita que a complexidade e as peculiaridades da API do LDAP permeiem todo o código do servidor de nomes.

Todas as funções de acesso ao repositório abstrato tem o prefixo "`repository_`" em seus nomes. Eis a relação dessas funções:

- `repository_init()` — inicializa a biblioteca de acesso ao serviço de armazenamento subjacente, lendo do arquivo de configuração `repository.conf` os parâmetros de inicialização que forem necessários;

- `repository_connect()` — abre uma conexão com o serviço de armazenamento subjacente;

- `repository_disconnect()` — encerra a conexão com o serviço de armazenamento subjacente;

- `repository_cosBinding_insert()` — insere no repositório uma associação nome–referência vinculada a dado contexto;

- `repository_cosBinding_lookup()` — procura pelo nome um objeto vinculado a um dado contexto, retornando uma referência para esse objeto e o seu tipo (objeto de aplicação ou contexto);

- `repository_cosBinding_remove()` — remove do repositório uma associação nome–referência vinculada a um dado contexto;

- `repository_cosContext_insert_new()` — cria um novo contexto e o insere no repositório;

- `repository_cosContext_insert()` — insere no repositório um dado contexto;
- `repository_cosContext_lookup()` — procura por um contexto no repositório;
- `repository_cosContext_remove()` — remove do repositório um dado contexto;
- `repository_new_id_context()` — cria um novo identificador de contexto;
- `repository_list()` — lista todos os objetos e contextos vinculados a um um contexto especificado.

A subseção abaixo especifica a API de acesso ao repositório. As descrições das funções estão voltadas para o caso específico em que o repositório é um diretório LDAP, mas isso não compromete a generalidade da API. Seria uma tarefa simples implementar um conjunto equivalente de funções que fizesse acesso a outro serviço de armazenamento persistente.

5.5.1 A API de Acesso ao Repositório

As entradas subsequentes especificam todas as funções da API.

5.5.1.1 repository_init

Descrição

Lê os seguintes parâmetros de configuração do arquivo `repository.conf`: o nome do host que está executando o servidor de diretório LDAP, o usuário e a senha para estabelecimento de uma sessão LDAP (esse usuário deverá ter permissões para escrita e leitura no diretório), o DN a ser usado como base para todas as operações no diretório, e o valor do atributo `corbaBase` a ser usado em todas as operações no diretório.

Protótipo

```
int repository_init();
```

Valor retornado

0 se a chamada foi bem sucedida, ou 1 em caso de problemas na leitura do arquivo de configuração `repository.conf`.

5.5.1.2 repository_connect

Descrição

Abre uma sessão com o servidor LDAP.

Protótipo

```
int repository_connect();
```

Valor retornado

0 se a chamada foi bem sucedida, ou 1 em caso de problemas com o estabelecimento da sessão LDAP ou com a autenticação do usuário.

5.5.1.3 repository_disconnect

Descrição

Encerra uma sessão com o servidor LDAP.

Protótipo

```
int repository_disconnect();
```

Valor retornado

0 se a chamada foi bem sucedida, ou 1 em caso de problemas com o encerramento da sessão LDAP.

5.5.1.4 repository_cosBinding_insert

Descrição

Insere no diretório LDAP uma entrada `corbaCosBinding`.

Protótipo

```
int repository_cosBinding_insert(char *name, char *kind,
                                 char *ior, int type,
                                 char *ctxId);
```

Parâmetros

`name:` string contendo o atributo `corbaName` da entrada a ser inserida no diretório.

`kind`: string contendo o atributo `corbaKind` da entrada a ser inserida no diretório.

`ior`: string contendo o atributo `corbaIor` da entrada a ser inserida no diretório.

`type`: inteiro que determina o atributo `corbaType` da entrada a ser inserida no diretório. O valor 0 indica que o *binding* envolve um contexto, o valor 1 indica que o *binding* envolve um objeto de aplicação.

`ctxId`: string contendo o atributo `corbaCtxId` da entrada a ser inserida no diretório.

Valor retornado

0 se a chamada foi bem sucedida, 1 se já existia no diretório uma entrada `corbaCosBinding` com os atributos `corbaName`, `corbaKind` e `corbaCtxId` especificados ou se ocorrerem problemas na inserção, ou 2 caso não exista no diretório uma entrada `corbaCosContext` com o atributo `corbaCtxId` especificado.

5.5.1.5 repository_cosBinding_lookup

Descrição

Busca no diretório LDAP uma entrada `corbaCosBinding`, dados seus atributos `corbaName`, `corbaKind` e `corbaCtxId`.

Protótipo

```
int repository_cosBinding_lookup(char *name, char *kind,
                                 char *ctxId, int *type,
                                 char **ior);
```

Parâmetros

`name`: string contendo o atributo `corbaName` da entrada procurada.

`kind`: string contendo o atributo `corbaKind` da entrada procurada.

`ctxId`: string contendo o atributo `corbaCtxId` da entrada procurada.

`type`: ponteiro para um inteiro no qual será retornado o valor 0 caso o *binding* encontrado envolva um contexto, ou 0 caso o *binding* encontrado envolva um objeto de aplicação.

`ior`: ponteiro para uma string no qual será retornado o atributo `corbaCtxId` da entrada encontrada.

Valor retornado

0 se a entrada foi encontrada, ou 1 se ela não foi encontrada ou se houve algum erro durante a busca no diretório LDAP.

5.5.1.6 repository_cosBinding_remove

Descrição

Remove do diretório LDAP a entrada `corbaCosBinding` com os atributos `corbaName`, `corbaKind` e `corbaCtxId` especificados.

Protótipo

```
int repository_cosBinding_remove(char *name, char *kind,
                                 char *ctxId);
```

Parâmetros

`name:` string contendo o atributo `corbaName` da entrada a ser removida.

`kind:` string contendo o atributo `corbaKind` da entrada a ser removida.

`ctxId:` string contendo o atributo `corbaCtxId` da entrada a ser removida.

Valor retornado

0 se a chamada foi bem sucedida, ou 1 em caso de problemas com a a remoção da entrada do diretório LDAP.

5.5.1.7 repository_cosContext_insert_new

Descrição

Gera um novo identificador de contexto e insere no diretório LDAP uma entrada `corbaCosContext` com esse identificador no atributo `corbaId`. Para gerar o identificador de contexto, esta função utiliza o contador armazenado numa entrada `corbaNextCtxId`.

Protótipo

```
int repository_cosContext_insert_new(int *ctxNumber);
```

Parâmetros

`ctxNumber:` Ponteiro para um inteiro no qual será retornado o número do novo contexto inserido no diretório LDAP.

Valor retornado

0 se a chamada foi bem sucedida, ou 1 em caso de erro.

5.5.1.8 repository_cosContext_insert

Descrição

Insere no diretório LDAP uma entrada `corbaCosContext`. A chamada a esta função não gera um novo identificador de contexto.

Protótipo

```
int repository_cosContext_insert(char *ctxId);
```

Parâmetros

`ctxId:` string com o atributo `corbaId` da entrada a ser inserida.

Valor retornado

0 se a chamada foi bem sucedida, 2 se já existia no diretório uma entrada `corbaCosContext` com o atributo `corbaId` especificado, ou 1 se ocorreu outro erro.

5.5.1.9 repository_cosContext_lookup

Descrição

Busca no diretório LDAP uma entrada `corbaCosContext`, dado o valor do seu atributo `corbaId`.

Protótipo

```
int repository_cosContext_lookup(char *ctxId);
```

Parâmetros

`ctxId:` string contendo o atributo `corbaId` da entrada procurada.

Valor retornado

0 se a entrada foi encontrada, 1 se ela não foi encontrada ou se ocorreu algum erro.

5.5.1.10 repository_cosContext_remove

Descrição

Remove do diretório LDAP a entrada `corbaCosContext` com o atributo `corbaId` especificado.

Protótipo

```
int repository_cosContext_remove(char *ctxId);
```

Parâmetros

`ctxId`: string contendo o atributo `corbaId` da entrada a ser removida.

Valor retornado

0 se a remoção foi bem sucedida, 2 se no diretório não foi encontrada uma entrada `corbaCosContext` com o atributo `corbaId` especificado, ou 1 se ocorreu outro erro.

5.5.1.11 repository_list

Descrição

Obtém todas as entradas `corbaCosBinding` com um dado valor do atributo `corbaCtxId`.

Protótipo

```
int repository_list(char *ctxId, repository_cache_insert_f *func,
                    CORBA_Environment *ev);
```

Parâmetros

`ctxId`: string contendo o atributo `corbaCtxId` da entradas desejadas.

`func`: ponteiro para uma função que recebe como parâmetros os atributos de uma entrada `corbaCosBinding`. Para cada entrada encontrada ocorrerá uma chamada a esta função, que deve ser compatível com a definição abaixo:

```
typedef void
repository_cache_insert_f(char *ctxId, char *name_id,
                          char *name_kind, char *ior,
                          int type, CORBA_Environment *ev);
```

`ev`: ponteiro para uma estrutura `CORBA_Environment` que será repassado às chamadas a função apontada por `func`.

Valor retornado

0 se a chamada foi bem sucedida, ou 1 em caso de erro.

5.6 O Esquema de *Caching* no Servidor de Nomes

Para evitar que toda chamada ao serviço de nomes ocasione outra para o repositório abstrato e, consequentemente, uma requisição ao servidor LDAP, implementamos um esquema simples de *caching* [3]. Empregamos duas tabelas de *hash* em memória, denominadas `the_context_table` e `the_name_table`. Na primeira são mantidas as últimas m entradas `corbaCosContext` acessadas pelo servidor de nomes, m é um parâmetro de configuração do servidor. Na segunda tabela de *hash* são mantidas as últimas n entradas `corbaCosBinding` acessadas pelo servidor de nomes, onde n é outro parâmetro de configuração do servidor de nomes. A figura 5.10 ilustra o funcionamento desse *cache*.

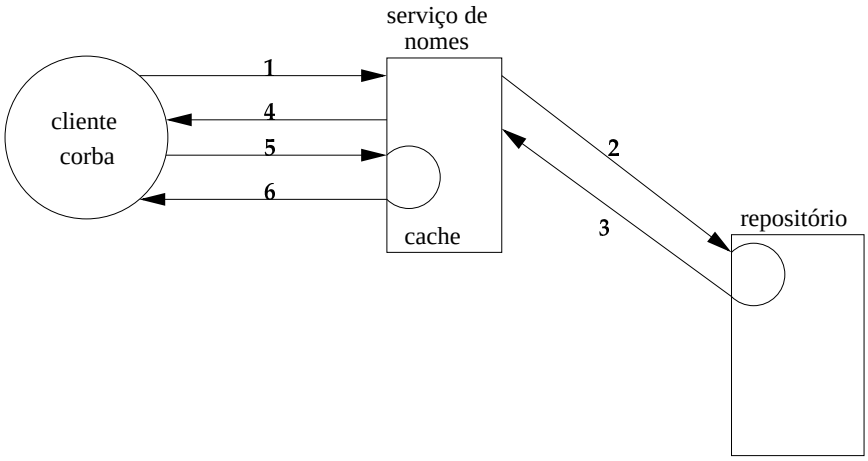

Figura 5.10: Esquema de *caching* no servidor de nomes.

Todos os acessos ao repositório são efetuados através das funções que manipulam o *cache*. O conjunto dessas funções forma uma camada que envolve o repositório e oferece

[3] O OpenLDAP não implementa um esquema de *caching* na biblioteca do cliente, ao contrário da implementação LDAP da Netscape.

ao restante do servidor de nomes uma API semelhante à do repositório. A camada de gerenciamendo do *cache* oferece, por exemplo, uma função que procura um *binding* pelo nome. Essa função faz primeiro uma busca em memória, na tabela `the_name_table`, e chama `repository_cosBinding_lookup()` somente se o *binding* procurado não estiver em memória. Caso o *binding* não esteja em memória e seja encontrado no repositório, ele é adicionado a `the_name_table`.

A implementação deste esquema de *caching* foi facilitada pelo emprego da biblioteca GLib [3] disponível para o Linux, que oferece implementações genéricas de diversas estruturas de dados, inclusive tabelas de *hash* e listas ligadas.

5.7 Utilização do POA pelo Servidor de Nomes

Para os objetos `NamingContext` o servidor de nomes usa um POA com as seguintes políticas: `PERSISTENT`, `USER_ID`, `NON_RETAIN`, `USE_DEFAULT_SERVANT` e `MULTIPLE_ID`.

- A política `PERSISTENT` foi selecionada para que as referências `NamingContext` permaneçam válidas ao longo de sucessivas execuções de um servidor.
- Com a política `USER_ID`, o servidor pode determinar o campo *object id* das referências para `NamingContexts`. O servidor usa esse recurso para embutir nessas referências o atributo `corbaId` de uma entrada `corbaCosContext` no diretório.
- As políticas `NON_RETAIN` e `USE_DEFAULT_SERVANT` fazem com que um servente default trate todas as requisições para `NamingContexts`. Essa escolha é motivada por razões de escalabilidade e simplicidade de implementação. Para saber qual `NamingContext` encarnar a cada requisição, o servente default inspeciona o *object id* alvo da requisição.
- A política `MULTIPLE_ID` foi selecionada porque o servente default tratará requisições para vários `NamingContexts`, com diferentes *object ids*.

5.8 O Ambiente de Desenvolvimento

Para o desenvolvimento do protótipo foi usado o software relacionado abaixo:

- Sistema operacional: Red Hat Linux 6.2 Kernel 2.2.14-5.0;
- CORBA: ORBit 0.5.1 — CVS head;
- LDAP: OpenLDAP 2.0.6 [4];
- Linguagem: C, compilador gcc (egcs-1.1.2).

O OpenLDAP, derivado do LDAP da Universidade de Michigan, foi empregado por ser uma das implementações mais difundidas de LDAP e por ser *open-source*. A escolha do OpenLDAP implicou na escolha da linguagem C, dada a disponibilidade de bibliotecas para clientes LDAP escritos em C.

O ORBit foi empregado por ser o único ORB que implementa o mapeamento de IDL para C. A versão do ORBit que utilizamos inicialmente não implementava adequadamente as políticas do POA que desejávamos usar. Tivemos que interagir com os desenvolvedores do ORBit para dar solução a esse problema e nossas sugestões foram bem recebidas.

[4]Derivado do LDAP 3.3 desenvolvido pela Universidade de Michigan.

Capítulo 6

Trabalho Relacionado

O artigo [8] apresenta a especificação e a implementação de um serviço de diretório totalmente baseado no padrão CORBA. Essa abordagem resultou na definição de uma interface IDL para acesso ao diretório, interface esta similar à API Java para o LDAP [32].

Os autores apresentaram duas técnicas de implementação diferentes para o serviço de diretório. Uma delas consiste em um servidor de diretório construído como um servidor CORBA puro. A outra versão aplica a técnica de *wrapper* para utilizar um servidor LDAP existente. Esta alternativa adiciona um servidor CORBA apropriado, num esquema de arquitetura de três camadas. O servidor CORBA fica na camada intermediária e oferece interfaces com funcionalidade correspondente à do servidor LDAP existente, que fica na camada mais distande dos clientes. Esta arquitetura é muito semelhante à que empregamos em nosso trabalho.

Nessa implementação, cada requisição CORBA de busca ou alteração no diretório é mapeada para uma requisição LDAP. O servidor CORBA faz esse mapeamento, em vez de implementar as facilidades de um diretório. Já na primeira abordagem, o servidor CORBA inclui uma implementacão completa de serviço de diretório. Segundo os autores, esta última abordagem prepara o caminho para uma migração suave do serviço de diretório LDAP, utilizado atualmente em várias redes no mundo todo, para um serviço CORBA puro.

6.1 Motivação

A motivação para o desenvolvimento do trabalho, nos sistemas de telecomunicações, mais especificamente na área de mobilidade, é a necessidade cada vez maior de utilização de diretórios capazes de armazenar e disponibilizar informações pessoais dos usuários. Efetuar a associação entre terminais, serviços de mobilidade e usuários permite disponibilizar um universo gigantesco de facilidades, ainda não exploradas, nos serviços de rede. Como exemplo os autores apresentam um cenário onde, cada usuário de uma rede de dimensões mundiais, possuirá um número; chamado de identificador pessoal (*personal identifier*). Esse usuário se conectará a rede – exemplo Internet – efetuando *dial-in*, através do protocolo PPP, que fornece um endereço IP diferente cada vez que o usuário se conecta. Uma tecnologia que realize o mapeamento do identificador pessoal, com o endereço IP do recurso utilizado na conexão, e os serviços, facilidades e etc desse usuário, é fundamental nesse ambiente.

O diretório pode ser empregado para o armazenamento de informações sobre o perfil do usuário de forma a permitir que se ofereçam serviços personalizados em qualquer parte da rede. Surge então a pergunta: como as tecnologias de base, na qual os serviços de diretório são classificados, suportarão esses requisitos? Os autores consideram o diretório do usuário como parte crucial na infraestrutura de rede.

CORBA está se tornando o padrão de fato em tecnologias de sistemas distribuídos, em particular em telecomunicações [7, 8]. A potencialidade que as aplicações desenvolvidas no padrão CORBA oferecem são bem aproveitadas nos produtos e serviços de telecomunicações.

6.2 Conversão dos Formatos de Dados

Um dos aspectos chave da implementação do servidor de diretório CORBA é a conversão do formato dos valores de atributos, que estão em LBER, para o formato correspondente CDR do CORBA (*Common Data Representation*).

O formato LBER é uma representação de dados convertida em strings (binário ou caracteres). A representação não indica o tipo de dado. Por exemplo, o string “42” no formato LBER pode representar um caracter, um número inteiro ou um valor binário original pertencente a um código de áudio ou de vídeo.

A implementação do servidor de diretório CORBA puro usa o *LDAP Data Interchange Format* (LDIF) como formato para armazenar as entradas do diretório de forma persistente. Assim foi possível efetuar a migração de dados de um servidor de diretório LDAP, já existente, para um servidor CORBA puro e vice-versa. Quando o servidor CORBA puro é inicializado, os dados são carregados em memória principal e permanecem lá durante todo o ciclo de vida do servidor. Quando é efetuado um *shutdown*, os dados são armazenados novamente, em formato LDIF, em um arquivo. Como o LDIF não possui o esquema de informação, o servidor deve extrair o esquema de um arquivo de configuração. Nesse arquivo são guardados as informações dos tipos dos valores dos atributos constantes no arquivo LDIF. Dessa forma é possível a interpretação da semântica correta do conteúdo do arquivo LDIF e efetuar a conversão dos formatos das informações que estão em LBER para CDR (*Common Data Representation*) do CORBA.

A outra implementação, o chamado LDAP *wrapper*, adiciona, em um servidor LDAP, um objeto CORBA responsável em traduzir as requisições CORBA em LDAP, e vice-versa. Essa solução é semelhante à nossa implementação.

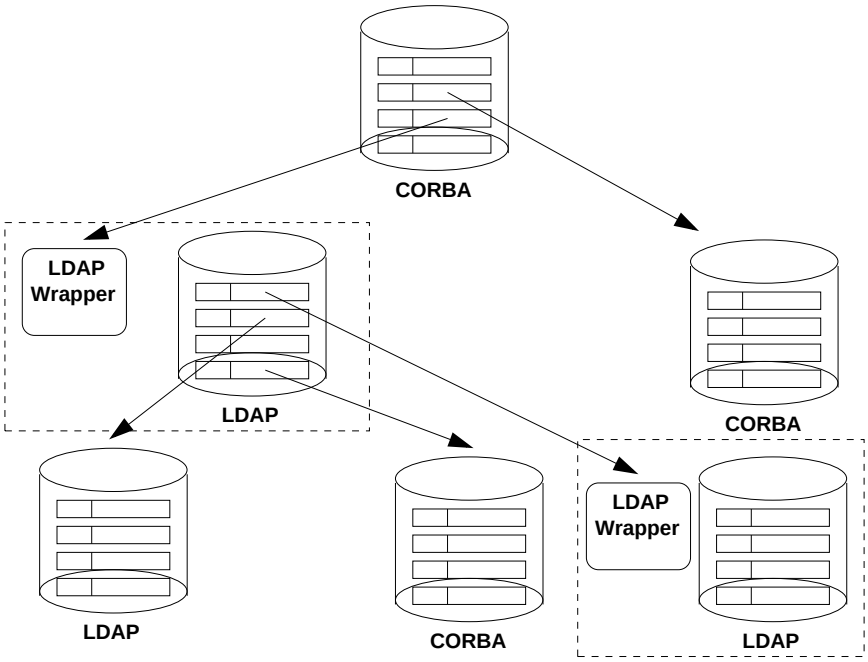

Figura 6.1: Uma rede contendo serviços de diretório heterogêneos - LDAP Wrappers e CORBA puro.

O objetivo é permitir que clientes CORBA acessem as informações registradas no

serviço de diretório LDAP através das operações que utilizam as IDLs do CORBA. Para isso, uma operação remota é mapeada, internamente, ao objeto CORBA em uma requisição LDAP, ao invés de implementar as funções do serviço de diretório em CORBA.

Uma das tarefas do LDAP *wrapper* é converter os dados que estão em formato LBER em formato CDR. Para isso, é necessário a conversão da operação IDL invocada, em uma requisição LDAP correspondente. E o resultado da requisição LDAP para o tipo de IDL apropriado. A primeira conversão pode ser fácil de ser implementada, porque os dados são tipados de forma explícitas; ou seja, os dados contêm um campo discriminador indicando o tipo de dado.

Porém a conversão dos dados na direção oposta pode ser muito complicada. O formato LBER é uma representação implícita: o tipo de dado não acompanha a informação. Isso requer que o *wrapper* possua algum conhecimento externo, de forma a identificar o tipo de dado constante no LBER para que ele seja convertido no formato CDR correto. Para esse propósito, de acordo com o artigo, com o LDAP versão 3 são efetuadas operações de consulta ao esquema de dados contidas no diretório e essas informações, como os tipos de atributos, são colocados em *cache* para que sejam consultados durante as conversões.

6.3 Resultados obtidos

Os autores concluem que o uso do serviço de diretório CORBA oferece suporte a vários aspectos do *mobility*. A implementação do serviço de diretório CORBA puro oferece transparência de localização ao acesso aos dados. O cliente do serviço de diretório não precisa ter o conhecimento de como foi implementado o diretório para acessar a informação em redes heterogêneas, figura 6.1.

Nos seus experimentos, o protocolo IIOP, que é de propósito geral, teve desempenho melhor que o protocolo LDAP, que é específico para acesso a diretórios. Esse é um resultado surpreendente.

Também é mostrado como o serviço de diretório pode dar suporte a mobilidade fornecendo diversos mecanismos, tais como: mapeamento dos endereços de recursos ao endereço pessoal, do perfil do usuário multimídia às suas informações e configurações, ou para outros serviços avançados, possibilitando a carga de informações de forma inteligente, com o intuito de oferecer serviços inteligentes aos usuários.

6.4 Comparação com Nosso Trabalho

6.4.1 As Semelhanças

A implementação de um servidor de diretório CORBA como *IDL wrapper* para um servidor LDAP tem alguns pontos em comum com nosso trabalho. Ambos os casos empregam a arquitetura *three-tier* e um servidor LDAP na última camada (a mais distante dos clientes).

A nossa implementação e a implementação LDAP *wrapper* recebe as requisições enviadas, tanto por um cliente CORBA, quanto por um cliente LDAP. Ou seja, o diretório poderá ser consultado, alterado ou suas entradas podem ser removidas, tanto através das interfaces IDL do CORBA, quanto através das APIs do LDAP.

Ambas implementações, a desenvolvida em nosso trabalho e a desenvolvida no trabalho [8], preparam o caminho para uma suave migração do LDAP para uma implementação do serviço de diretório CORBA.

Considerando a grande aceitação do padrão CORBA no desenvolvimento das aplicações dos sistemas distribuídos, as implementações apresentadas em nosso trabalho e no artigo tendem a ser cada vez mais necessárias. Apesar dos esforços para reduzir a complexidade do LDAP comparado ao X.500, a implementação de um serviço de diretório, como o LDAP, em grandes redes ainda é uma tarefa um tanto complexa e difícil. O IETF definiu a linguagem C [11] e a linguagem Java [32] como as linguagens para as APIs do LDAP. Isso fez com que se mantenham as características de dependência das linguagens; diferentemente do padrão CORBA que possibilita a independência.

6.4.2 As Diferenças

Os trabalhos se diferenciam, no entanto, pelas interfaces oferecidas pelo servidor na camada intermediária. O *IDL wrapper* descrito em [8], implementa interfaces que procuram disponibilizar, para clientes CORBA, a funcionalidade da API do LDAP. Já em nosso trabalho, o servidor na camada intermediária implementa as interfaces do serviço de nomes padronizado pelo OMG.

Também, as informações que estão sendo disponibilizadas para a rede, pelo servidor de diretório implementado em nosso trabalho, são as associações nome–referências dos objetos CORBA, que são armazenados de forma persistente no diretório LDAP. Essas informações possuem um formato bem definido, ou seja, os *object classes* especificados

(apresentados na seção 5.4) para armazenar as informações, foram previamente definidos e os seus formatos bem conhecidos antes do armazenamento.

Já no caso da implementação do artigo [8], as informações que estão sendo disponibilizadas para a rede, pelo servidor de diretório, são todos os valores e atributos armazenados no diretório. Conseqüentemente, a aplicação precisa fazer um trabalho maior para realizar a conversão do formato dos dados, que está em LBER do LDAP, para o formato CDR do CORBA e vice-versa.

Em nossa implementação, não foi necessário o desenvolvimento de artifícios para o tratamento específicos dos formatos dos dados pelos motivos citados acima. O serviço de diretório LDAP está apenas sendo utilizado como um repositório de dados que armazena as associações nome–referência em seu diretório. As interfaces, apresentadas na seção 5.5.1, foram escritas tendo em mente a possibilidade que elas permitam a utilização de qualquer sistema de armazenamento persistente de informação. Elas poderiam utilizar um banco de dados relacional ao invés do diretório LDAP.

Capítulo 7

Considerações Finais

Este trabalho tem como principal contribuição a implementação de um protótipo de Serviço de Nomes CORBA persistente que usa um diretório LDAP como repositório de dados. O acesso a este repositório é feito através das interfaces descritas na seção 5.5, que ocultam do restante do código a complexidade da API do LDAP. Tal arquitetura facilita a substituição do diretório LDAP por outro sistema de armazenamento persistente.

O Serviço de Nomes armazena as associações nome–referências em um Serviço de Diretório LDAP. Assim, as informações registradas no Serviço de Nomes são acessíveis tanto através das operações das interfaces CORBA como através das APIs do LDAP. Clientes CORBA ou LDAP poderão consultar as informações armazenadas no diretório.

Na tabela 7.1 apresentamos, resumidamente, algumas operações da interface `CosNaming` do Serviço de Nomes CORBA, as correspondentes operações para a manipulação do *cache* e os procedimentos para acesso ao repositório implementados em nosso protótipo.

O código fonte do protótipo, bem como instruções para a sua instalação e operação, estão disponíveis na Internet, no página `http://www.ime.usp.br/~sidam/software/`. Esse código é um exemplo de programação CORBA avançada empregando o mapeamento de IDL para a linguagem de programação C. Considerando a atual escassez, na literatura, de exemplos de aplicações CORBA em linguagem C, acreditamos que ele preenche uma lacuna importante.

Quando iniciamos o desenvolvimento, o ORBit não suportava as políticas do POA necessárias que o Serviço de Nomes mantivesse as referências para os objetos de forma persistente. Contatamos o desenvolvedor responsável pelo ORBit na ocasião, Kennard

Interface CORBA	**Procedimento de manipulação do *cache***	**Interface para o repositório**

`bind()`
`rebind()`
`bind_context()`
`rebind_context()`
	`name_table_insert()`	`repository_cosBinding_insert()`

`resolve()`	`name_table_lookup()`	`repository_cosBinding_lookup()`
`unbind()`	`name_table_remove()`	`repository_cosBinding_remove()`
`destroy()`	`context_table_remove()`	`repository_cosContext_remove()`
`new_context()` `bind_new_context()`	`context_table_insert()`	`repository_cosContext_insert()`
`bindinglist()`		`repository_list()`
`get_target_context_number()`	`context_table_lookup()`	`repository_cosContext_lookup()`

Tabela 7.1: Operações do Serviço de Nomes e as interfaces correspondentes.

White, que aceitou prontamente nossas sugestões visando a solução deste problema. Posteriormente fomos contatados por ele, com a informação de que nossas sugestões foram incorporadas às próximas versões do ORBit.

Referências Bibliográficas

[1] Markus Endler, Dilma M. da Silva, Francisco J. Silva e Silva, Ricardo C. A. da Rocha, and Marcos A. M. de Moura. Project SIDAM: Overview and preliminary results. In *2nd Brazilian Wireless Communication Workshop*, pages 48–64, May 2000. Belo Horizonte, Brazil.

[2] OpenLDAP Foundation. OpenLDAP. `http://www.openldap.org`.

[3] GLib Project. GLib Reference Manual. `http://developer.gnome.org/doc/API/glib/index.html`.

[4] Object Management Group. The Common Object Request Broker: Architecture and Specification — Revision 2.3, 1998. `ftp://ftp.omg.org/pub/docs/formal/98-12-01.pdf`.

[5] Object Management Group. *CORBA V2.2 — C Language Mapping*. Object Management Group, February 1998.

[6] Object Management Group. *CORBAservices: Common Object Services Specification*. Object Management Group, October 1998.

[7] Object Management Group. *CORBAtelecoms: Telecommunications Domain Specification*. Prentice Hall Software Series, June 1998.

[8] Oliver Haase, Andreas Schrader, Kurt Geihs, and Rudolf Jans. Mobility support with CORBA directories. In Taieb Znati and Robert Simon, editors, *Communication Networks and Distributed Systems Modelling and Simulation*, pages 20–29. The Society for Computer Simulation International, January 2000.

[9] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison Wesley Longman Inc., January 1999.

[10] T. Howes, S. Kille, W. Yeong, and C. Robbins. *The String Representation of Standard Attribute Syntaxes*. RFC 1778, IETF, March 1995.

[11] T. Howes and M. Smith. *The LDAP Application Programming Interface*. RFC 1823, IETF, August 1995.

[12] T. Howes, M. Smith, and B. Beecher. *DIXIE Protocol Specification*. RFC 1249, IETF, August 1991.

[13] Tim Howes and Mark Smith. *LDAP Programming Directory — Enabled Applications with Lightweight Directory Access Protocol*. Macmillan Technical Publishing, Indianapolis, IN, 1997.

[14] Timothy A. Howes, Mark C. Smith, and Gordon S. Good. *Understanding and Deploying LDAP Directory Services*. Macmillan Technical Publishing, 1999.

[15] ITU-T. *Information Technology — Open Systems Interconnection — The Directory — Recomendations X.500–X.521*. ITU — International Telecommunication Union, 1993.

[16] Kris Jamsa and Lars Klander. *Programando em C/C++*. Makron Books do Brasil Editora Ltda, São Paulo, SP, 1999.

[17] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Software Series, June 1988.

[18] S. Kille. *A String Representation of Distinguished Names*. RFC 1779, IETF, March 1995.

[19] Pete Loshin. *Big Book of Lightweight Directory Access Protocol LDAP RFCs*. Morgan Kaufmann, 2000.

[20] OpenLDAP. *OpenLDAP 2.0 Administrator's Guide*, September 15 2000.

[21] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA*. John Wiley and Sons, Inc., New York, NY, second edition, 1998.

[22] A. Pope. *The CORBA Reference Guide*. Addison Wesley, Reading, MA, 1997.

[23] M. Rose. *Directory Assistance Service*. RFC 1202, IETF, February 1991.

[24] Ward Rosenberry, David Kenney, and Gerry Fisher. *Understanding DCE*. O'Reilly and Associates, Inc., September 1992.

[25] Douglas C. Schmidt and Steve Vinoski. *Object Adapters: Concepts and Terminology*. SIGS C++ Report, October 1997.

[26] Beth Sheresh and Doug Sheresh. *Understanding Directory Services*. New Riders Publishing, Indianapolis, IN, 2000.

[27] John Strassner. *Directory Enabled Networks*. Macmillan Technical Publishing, Indianapolis, IN, 1999.

[28] University of Michigan. *The SLAPD and SLURPD Administrator's Guide — Release 3.3*, April 30 1996.

[29] Steve Vinoski. CORBA: Integrating Diverse Applications Within Heterogeneous Environments. *IEEE Communications*, 14(2), February 1997.

[30] M. Wahl. *A Summary of the X.500(96) User Schema for use with LDAPv3*. RFC 2256, IETF, December 1997.

[31] M. Wahl, T. Howes, and S. Kille. *Lightweight Directory Access Protocol (v3)*. RFC 2251, IETF, December 1997.

[32] R. Weltman, C. Tomlinson, J. Sermersheim, M. Smith, and T. Howes. *The Java LDAP Application Program Interface*. Internet draft, IETF, March 2000. `http://www.ietf.org/internet-drafts/draft-ietf-ldapext-ldap-java-api-10.txt`.

[33] Mark Wilcox. *Implementing LDAP*. Wrox Press Ltd., Birmingham, UK, 1999.

[34] W. Yeong, T. Howes, and S. Kille. *Lightweight Directory Access Protocol*. RFC 1777, IETF, March 1995.