Gedeon José dos Santos Filho

Considerações para o Desenvolvimento de Aplicações Distribuídas em Ambientes Heterogêneos

Dissertação apresentada ao Curso de MESTRADO EM INFORMÁTICA da Universidade Federal da Paraíba, em cumprimento às exigências para obtenção do Grau de Mestre.

Área de Concentração: Ciência da Computação

Jacques Phillipe Sauvé
(*Orientador*)

Campina Grande
Maio - 1993

Universidade Federal da Paraíba
Centro de Ciências e Tecnologia
Coordenação de Pós-Graduação em Informática

Considerações para o Desenvolvimento de Aplicações Distribuídas em Ambientes Heterogêneos

Gedeon José dos Santos Filho

Campina Grande - PB
- 1993

S237c Santos Filho, Gedeon José dos.
Considerações para o desenvolvimento de aplicações distribuídas em ambientes heterogêneos / Gedeon José dos Santos Filho. - Campina Grande, 1993.
164 f.

Dissertação (Mestrado em Informática) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, 1993.
"Orientação : Prof. Dr. Jacques Philippe Sauvé".
Referências.

1. Programas de Computador. 2. Informática. 3. Dissertação - Informática. I. Sauvé, Jacques Phillipe. II. Universidade Federal da Paraíba - Campina Grande (PB). III. Título

CDU 004.4(043)

CONSIDERAÇÕES PARA O DESENVOLVIMENTO DE APLICAÇÕES DISTRIBUÍDAS EM AMBIENTES HETEROGÊNEOS

GEDEON JOSÉ DOS SANTOS FILHO

DISSERTAÇÃO APROVADA EM 04.05.1993

JACQUES PHILIPPE SAUVÉ, Ph. D.
Orientador

JOSÉ ANTÃO BELTRÃO MOURA, Ph. D.
Componente da Banca

PEDRO SÉRGIO NICOLLETTI, M. Sc.
Componente da Banca

JOSÉ ANTÔNIO MONTEIRO DE QUEIROZ, Ph. D.
Componente da Banca

Campina Grande, 04 de maio de 1993

A meus pais.

Agradecimentos

A realização deste trabalho foi possível graças ao apoio e à colaboração de várias pessoas que, direta ou indiretamente, proporcionaram as condições necessárias para o seu desenvolvimento. A todos quero externar meu agradecimento, destacando em especial:

- O meu orientador Jacques Philippe Sauvé, por confiar a mim este trabalho e, acima de tudo, por compartilhar comigo sua experiência e conhecimento técnico.

- A Fundação Centro de Análises, Pesquisa e Inovação Tecnológica (FUCAPI), por me proporcionar esta oportunidade de aprimoramento técnico e profissional.

- O prof. Hélio de Menezes Silva, pela orientação acadêmica, confiança e apoio inestimáveis, oferecidos durante a fase das disciplinas.

- O prof. Pedro Sérgio Nicolletti (DSC), pelo apoio e comentários durante o desenvolvimento do trabalho.

- Os professores João Marques de Carvalho (DEE) e Maria Izabel C. Cabral (DSC), pelo espírito de colaboração durante a fase de implementação deste trabalho.

- Os funcionários da Coordenação de Pós-Graduação em Informática (COPIN) da UFPB, em especial a Ana Lúcia Guimarães, pelo dedicado, atento e eficiente trabalho de apoio secretarial com o qual pude contar ao longo deste trabalho.

- A turma da cerveja, pelas horas de descontração.

Resumo

As Aplicações Distribuídas são uma forte tendência para a computação dos anos 90. Este trabalho apresenta considerações práticas para orientar o programador no desenvolvimento de Aplicações Distribuídas em sistemas de redes heterogêneas. O ambiente, os requisitos e os principais modelos de programação usados para o desenvolvimento de Aplicações Distribuídas são discutidos, seguido de considerações específicas para projeto, tais como, protocolos de transporte, tratamento e recuperação de erros, segurança e desempenho. Com base nas considerações específicas, são descritas duas implementações de um serviço de impressão distribuído, cuja análise comparativa fornece subsídios importantes para programadores de Aplicações Distribuídas.

Abstract

Distributed Applications constitute a very important trend for computer science in the 1990's. This work presents practical advice considerations for heterogeneous distributed networked applications developers. The environment, requirements and main programming models used for development of distributed applications are discussed. Other specific considerations for design, such as transport protocols, fault tolerance, security and performance are also presented. Based on these specific considerations two implementations of a distributed printing service are described. Comparative analysis of such implementations results in important help for Distributed Applications developers.

Lista de Figuras

Sumário

Capítulo 3 - Considerações Básicas de Projeto

Capítulo 4 - Modelos para Projeto de Aplicações Distribuídas

Capítulo 5 - Considerações Específicas de Projeto

Capítulo 1

1. Introdução

O interesse por sistemas de computação distribuída vem crescendo rapidamente nos últimos anos. A redução dos custos de *hardware*, combinada com os avanços tecnológicos dos sistemas de redes de computadores, tem viabilizado o *downsize*, estimulando a utilização de modelos de processamento distribuído. Nesse contexto, uma aplicação não está mais limitada a utilizar somente os recursos[1] disponíveis em um computador local. Cada vez mais, as aplicações estão sendo estruturadas para realizarem suas tarefas de forma descentralizada, aproveitando integralmente os recursos disponíveis nas redes, obtendo assim uma melhor relação custo/benefício.

1.1. Motivação

Um dos maiores beneficiados com os modelos de processamento distribuído são os sistemas que possuem requisitos de **distribuição inerente**. Sistemas com essa característica normalmente necessitam realizar operações em computadores geograficamente dispersos, como por exemplo, o envio de correspondência eletrônica (*email*) entre usuários de diferentes computadores. Os computadores podem ser vistos como um sistema de computação distribuída que exige a execução da aplicação de correio eletrônico em cada uma das máquinas. Nesse caso, o modelo tradicional de aplicações centralizadas é inadequado para a modelagem de uma solução, porque suas operações estariam restritas a um único computador. Uma alternativa para resolver esse tipo problema são as **aplicações distribuídas (ADs)**.

---

[1] Computadores, processos, unidades de discos e fitas, impressoras, arquivos, bancos de dados, *plotters*, etc.

1.1.1. Aplicação Distribuída Heterogênea

Uma aplicação é considerada distribuída se seus componentes (processos) podem ser executados paralelamente em espaços de endereçamento de diferentes computadores que estão conectados através de uma rede de comunicação. Os processos da AD que residem em diferentes máquinas são classificados como **processos remotos**. Quando os processos da AD podem residir em computadores com arquitetura ou sistema operacional distintos, então ela é considerada heterogênea.

As ADs heterogêneas são um meio que o programador dispõe para ter acesso racional aos recursos existentes nos sistemas de redes heterogêneas. A seguir serão apresentadas suas principais vantagens e desvantagens.

1.1.1.1. Vantagens

As principais vantagens que justificam o desenvolvimento de ADs são: confiabilidade, alto desempenho, uso especializado de *hardware* e modularidade.

• **Confiabilidade**: as ADs possuem um potencial inerente de tolerância a falhas[2]. Se um computador que contém processos da aplicação falha, qualquer outro conectado à rede pode ser usado para assumir as tarefas pendentes. Baseado nesse fato, as ADs podem ser projetadas para se recuperarem das falhas de forma transparente e confiável, fazendo a redistribuição dos processos afetados pela falha. Por esse motivo, são ideais para modelar aplicações onde a confiabilidade é crítica, como por exemplo, sistemas bancários e de controle de tráfego aéreo, onde a perda de alguns segundos pode significar grandes prejuízos financeiros ou a perda de vidas.

• **Alto Desempenho**: as aplicações que fazem uso intenso de CPU podem ser projetadas de forma a aproveitarem o potencial das múltiplas CPUs (e outros recursos) disponíveis na rede.

---

[2] Neste trabalho, falha é todo evento causado por um defeito de *hardware* ou *software*, que viola especificação da AD, dando origem a um erro.

Os processos remotos das ADs permitem ao programador obter paralelismo sem a necessidade do uso de *hardware* multiprocessador. Por exemplo, aplicações para processamento de imagens ficam extremamente mais rápidas com a computação remota de partes das imagens, feitas por processos executados paralelamente em CPUs de diferentes computadores [Bloomer91].

• **Uso Especializado do Hardware**: as ADs permitem o uso especializado dos recursos de *hardware* existentes na rede. O fato dos processos das ADs poderem ser executados em diferentes computadores, permite, por exemplo, que se desenvolva uma aplicação que consulte um banco de dados de um servidor de ciclos[3], processe os dados dessa consulta em um supercomputador e, finalmente, mostre os resultados em forma de gráfico em uma estação de trabalho. Do ponto de vista do usuário da AD, é como se existisse um "grande e único" computador reunindo simultaneamente as qualidades do servidor de ciclos, a velocidade do supercomputador e os recursos de interface gráfica da estação de trabalho. Isso permite a racionalização de uso dos recursos distribuídos pela rede, proporcionando uma melhor relação custo/desempenho.

• **Modularidade**: o fato das ADs serem compostas por um conjunto de processos cooperativos, normalmente executados em diferentes computadores, resulta em uma natureza modular intrínseca. Em uma aplicação centralizada, a modularidade é um conceito lógico. Em uma AD, a modularidade é física. Essa modularidade inerente pode ser usada em benefício do programador, por oferecer maior flexibilidade para adicionar, eliminar e modificar os módulos (conjunto de procedimentos) da AD.

1.1.1.2. Desvantagens

As ADs possuem algumas desvantagens que devem ser observadas, tais como: dependência de rede, problemas com segurança, adequação de *software* e complexidade.

---

[3] Termo normalmente associado a uma máquina com grande capacidade de processamento, como por exemplo, um *mainframe*.

• **Dependência da rede**: a comunicação entre os processos da AD é normalmente feita através da rede de comunicação. Em função disso, uma parte significativa do desempenho e confiabilidade das ADs está diretamente ligada ao desempenho e confiabilidade oferecidos pela infra-estrutura de comunicação (ex.: Ethernet, FDDI). Se a rede ficar sobrecarregada a AD pode ter o seu desempenho degradado. Além disso, a ocorrência de falha em trechos da rede podem causar a interrupção da comunicação entre os processos da AD.

• **Problemas com segurança**: para facilitar o desenvolvimento de ADs, várias APIs[4] são colocadas à disposição do programador. Através delas, o programador pode ter acesso aos serviços básicos de comunicação da rede, podendo, a partir daí, intencionalmente ou não, tentar acessar recursos aos quais não está autorizado, comprometendo a segurança.

• **Adequação de Software**: atualmente, existem dúvidas sobre que tipos de linguagens de programação, sistemas operacionais e metodologias são os mais adequados para o desenvolvimento de sistemas distribuídos. Esses problemas tendem a diminuir à medida que as pesquisas evoluem, mas atualmente é um aspecto que não deve ser ignorado [Tanenbaum92].

• **Aumento de Complexidade**: desenvolver ADs é muito mais complexo do que desenvolver aplicações centralizadas. Essa complexidade deve-se ao fato do programador ter que considerar fatores adicionais, como por exemplo, a existência de múltiplos processos sendo executados em diferentes computadores, a rede de comunicação, a diversidade de *hardware* e *software*, dentre outros.

1.2. Objetivos do Trabalho

Esse trabalho se propõe a realizar um estudo na área de computação distribuída, mais especificamente em projeto e implementação de ADs. O seu principal objetivo é apresentar considerações para auxiliar o programador no desenvolvimento de ADs heterogêneas eficientes e confiáveis. Os objetivos específicos são:

[4] As APIs (*Application Programming Interface*) são bibliotecas de funções padronizadas, colocadas à disposição do programador para permitir o desenvolvimento de aplicações com um maior nível de abstração, facilitando o trabalho de programação.

- Caracterizar a infra-estrutura necessária para o desenvolvimento de ADs heterogêneas.
- Determinar os requisitos necessários para o desenvolvimento de ADs heterogêneas.
- Identificar e avaliar os principais modelos de programação atualmente existentes, usados para o desenvolvimento de ADs heterogêneas.
- Propor considerações específicas para o desenvolvimento de ADs heterogêneas a partir dos requisitos e modelos de programação identificados.
- Avaliar as considerações propostas para o desenvolvimento de ADs a partir da implementação de um estudo de caso de uma AD heterogênea.

1.3. Importância do Trabalho

Pelas vantagens que oferecem, as ADs são seguramente uma forte tendência para a computação dos anos 90. Por esse motivo é importante dominar suas técnicas de projeto e implementação que, em vários aspectos, diferem radicalmente das utilizadas para o desenvolvimento de aplicações centralizadas tradicionais.

A partir da pesquisa bibliográfica constatou-se que, atualmente, existe carência de informações de caráter prático para orientar o programador no desenvolvimento de ADs heterogêneas. Como resultado dessa dissertação, projetistas e programadores, principalmente os não familiarizados com técnicas de processamento distribuído, terão a sua disposição considerações práticas para subsidiar processos de tomada de decisão em projetos e implementações de ADs, bem como um referencial para a implementação de serviços em ambientes de redes heterogêneas.

1.4. Trabalhos Relacionados

Os trabalhos mais recentes, direcionados para programadores de ADs, que constam da bibliografia consultada, são:

O livro *The Art of Distributed Applications - Programming Techniques for Remote Procedure Calls*, de [Corbin91], que apresenta técnicas para o projeto e programação de ADs, mas tem o seu enfoque centrado em redes de computadores Sun.

O livro *Power Programming with RPC* de [Bloomer91], escrito a partir do ponto de vista de um programador, apresenta um roteiro para a aprendizagem da técnica de *Remote Procedure Calls* e mostra o que pode ser feito, com o uso dessa técnica, para desenvolver ADs em redes de computadores Sun.

Finalmente, o livro *Guide to Writing DCE Applications* de [Shiley92] oferece um método básico para o projeto e a implementação de ADs em ambientes distribuídos que utilizam tecnologia da *Open Software Fundation* (OSF).

As informações contidas neste trabalho não são direcionadas para um ambiente distribuído específico, como por exemplo, Sun ou OSF. Na sua grande maioria, as informações aqui apresentadas são úteis para o desenvolvimento de ADs, independentemente do tipo de ambiente distribuído que esteja disponível para o programador.

1.5. Organização do Trabalho

Neste capítulo, foi apresentada uma definição de AD, suas vantagens e desvantagens, assim como os objetivos e a importância do trabalho.

No capítulo 2, *Infra-estrutura para Aplicações Distribuídas*, são apresentados a arquitetura e os serviços típicos de um ambiente de computação distribuída, os quais fornecem a infra-estrutura necessária para o desenvolvimento de ADs heterogêneas.

O capítulo 3, *Requisitos Básicos de Aplicações Distribuídas*, descreve os principais requisitos a serem considerados em projetos de ADs e apresenta alguns fatores responsáveis pelo aumento da complexidade em projetos de ADs.

No capítulo 4, *Modelos para Projeto de Aplicações Distribuídas*, são apresentados os principais modelos de programação usados para desenvolver ADs, incluído o modelo conceitual

cliente-servidor e os modelos de programação de troca de mensagens, chamada de procedimento remoto e invocação de objetos remotos.

O capítulo 5, *Considerações Específicas de Projeto*, apresenta considerações específicas, importantes para o projeto de ADs, a respeito da seleção de serviços de transporte, tratamento e recuperação de erros, desempenho e segurança.

O capítulo 6, *Implementações de um Serviço de Impressão Distribuído*, apresenta um estudo de casos envolvendo duas implementações de um serviço de impressão distribuído. São descritos detalhes do projeto de ambas implementações e feitas considerações específicas envolvendo os principais requisitos de ADs. Também é apresentado um comparativo entre as duas implementações.

O Capítulo 7, *Conclusão e Trabalhos Futuros*, se refere às conclusões e contribuições obtidas com o trabalho. Para trabalhos futuros, são apresentadas algumas sugestões envolvendo os aspectos que não foram cobertos por este trabalho.

No Apêndice A, são apresentados detalhes de implementação relacionados com semântica de compartilhamento de arquivos do *Network File System (NFS)*, que é um sistema de arquivo distribuído, desenvolvido pela Sun Microsystems.

No Apêndice B, são descritos em detalhes os protocolos de autenticação Secure RPC e Kerberos, bastante utilizados para implementar segurança em ambientes de computação distribuída.

É recomendável que o leitor recorra ao glossário, existente no fim do trabalho, em caso de dúvidas sobre o significado de siglas ou termos com os quais não esteja familiarizado.

Capítulo 2

2. Infra-estrutura para Aplicações Distribuídas

Neste capítulo, será apresentada a infra-estrutura necessária para o desenvolvimento de ADs heterogêneas. Inicialmente, será definido o que é um ambiente de computação distribuída heterogênea e, em seguida, descrita sua arquitetura e serviços disponíveis, que incluem: serviços de comunicação, serviço de *threads*, chamada de procedimento remoto, serviço de diretório distribuído, serviço de tempo, sistema de arquivos distribuído, sistema de janelas, integração com PCs, serviço de transações distribuídas e serviço de gerência distribuída.

2.1. Ambientes de Computação Distribuída

Idealmente, para desenvolver ADs é necessário um conjunto integrado de serviços a fim de evitar que o programador tenha que tratar, a nível de aplicação, toda a complexidade de um sistema de redes heterogêneas. Para atender a essa necessidade, existem os **Ambientes de Computação Distribuída Heterogênea** (ACDs).

Um ACD pode ser definido como um modelo arquitetural de computação no qual uma coleção de processos, recursos e computadores de diferentes fabricantes, distribuídos numa rede, trabalham cooperativamente para executar tarefas [Millikin91]. Os ACDs usam a conectividade oferecida pelas redes para tornar possível a interoperabilidade entre sistemas com diferentes arquiteturas e sistemas operacionais.

2.2. Arquitetura e Serviços

Os ACDs possuem uma arquitetura em camadas composta por um conjunto integrado de

serviços (**Fig. 2-1**). As camadas inferiores da arquitetura oferecem os serviços mais básicos, tais como o sistema operacional e os serviços de comunicação. Já as camadas superiores, como por exemplo, a de aplicações do usuário, são consumidoras dos serviços oferecidos pelas camadas inferiores. Os serviços de segurança e gerenciamento são essenciais e estão integrados com todas as camadas da arquitetura.

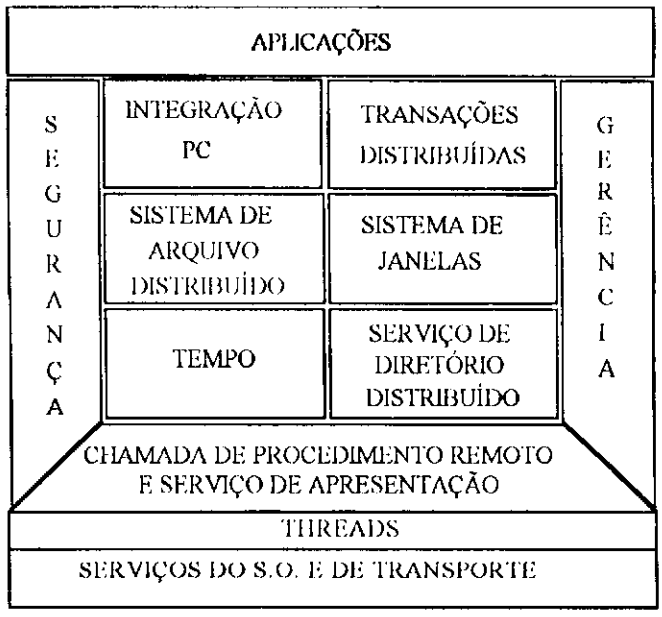


Figura 2-1. *Arquitetura típica de um Ambiente de Computação Distribuída*

A arquitetura do ACD procura tornar transparente a complexidade do sistema de rede para o usuário final, para o programador e para administrador do sistema [OSF92b]. Do ponto de vista das aplicações, o ACD pode ser visto tanto como um único sistema lógico quanto como uma coleção de serviços independentes, que podem ser utilizados individualmente ou combinados.

2.2.1. Serviço de Transporte

A função do serviço de transporte é integrar as diferentes tecnologias das redes físicas existentes no ACD, formando uma única rede lógica. Uma análise da **Fig. 2-1** mostra que as camadas superiores do ACD são dependentes dos serviços de transporte. Por exemplo, a nível de

aplicação, toda a comunicação entre os processos remotos das ADs é feita por intermédio do serviço de transporte.

Para tornar possível um trabalho cooperativo, face as diferentes tecnologias existentes, o serviço de transporte pode fazer uso praticamente de qualquer pilha de protocolos, como por exemplo, TCP/IP, SNA ou OSI. A utilização de um ou outro protocolo não é muito relevante, porque uma troca pode ser feita sem afetar as camadas superiores. Atualmente a grande maioria dos ACDs disponíveis adotam TCP/IP, mas a tendência para o futuro é a predominância de OSI. Detalhes sobre TCP/IP, SNA e OSI no contexto de ACDs podem ser encontrados em [Cypser91] [Commer91] [Rose91] [Martin92]. Na Seção 5.1, serão feitas várias considerações sobre a relação entre o serviço de transporte e as ADs.

2.2.2. Serviço de Threads

Um *thread* representa o agente de controle de uma sequência de instruções sendo executadas por um programa. Um processo em uma arquitetura *multi-thread* é composto por um ou mais *threads* que são executados em paralelo. Todos os *threads* de um processo compartilham o mesmo espaço de endereçamento, mas cada *thread* possui sua própria pilha e *program counter*. Do ponto de vista do programador, cada *thread* pode ser considerado como um processo sequencial convencional.

O serviço de *threads* provê uma API para dar suporte à programação de aplicações com paralelismo inerente. A partir desse serviço, um único processo pode possuir vários *threads* de controle, o que permite às aplicações processarem várias ações paralelamente. Por exemplo, enquanto um *thread* lê blocos de um arquivo em disco, outro *thread* pode paralelamente processar as entradas do usuário.

O serviço de *threads* inclui operações para criar e controlar a execução de múltiplos *threads* de um processo, bem como operações para sincronizar acessos a dados globais entre processos concorrentes. É um serviço que permite melhor aproveitamento de *hardware*

multiprocessador, simplificando o desenvolvimento de aplicações que possuem requisitos de paralelismo inerente. Uma discussão interessante sobre *threads* pode ser encontrada em [Powell91].

2.2.3. Chamada de Procedimento Remoto e Serviço de Apresentação

O serviço de chamada de procedimento remoto, ou RPC (*Remote Procedure Call*), oferece, para o programador, os componentes básicos que viabilizam a programação de ADs. Esse serviço fornece um paradigma de alto nível que permite a comunicação entre processos que residem em diferentes espaços de endereçamento de uma ou mais máquinas.

Com o serviço de RPC, o programador pode desenvolver uma aplicação que chame uma função (procedimento remoto) que será executada por outro processo de outra máquina. O serviço de RPC estende o modelo de chamada de procedimento local para um ambiente distribuído, colocando, à disposição do programador, um modelo de programação familiar que permite chamar procedimentos remotos como se eles fossem locais.

Em um ACD, é possível que os parâmetros e resultados das RPCs, feitas entre máquinas com arquitetura diferentes, tenham representação interna de dados incompatíveis, causando problemas. O serviço de RPC usa o **serviço de apresentação** para normalizar as diferentes representações de dados, possibilitando a chamada de procedimentos remotos em ambientes heterogêneos. Na Seção 4.3.3.1, serão discutidos aspectos do serviço de apresentação e RPC.

É importante observar que existem outros serviços, além de RPC, que possibilitam a comunicação entre processos remotos. Alguns ACDs colocam, à disposição do programador, serviços de mais alto nível que RPC. Um exemplo são serviços adequados à filosofia de orientação a objetos distribuídos, normalmente implementados sobre a camada de RPC. Esses serviços tornam possível a invocação de objetos que residem em espaços de endereçamento de diferentes máquinas. Maiores detalhes sobre RPC e invocação de objetos remotos serão dados no Capítulo 4.

2.2.4. Serviço de Diretório Distribuído

Em um ACD, é comum usuários (programadores, processos e usuários finais) necessitarem localizar um determinado recurso que se encontra disponível na rede. Por exemplo, como o usuário acha a máquina onde se encontram os discos, servidores, fita, arquivos e aplicações de que ele esteja necessitando? O serviço de diretório distribuído, ou serviço de ***naming***, resolve esse problema. Ele fornece um modelo de nomes únicos que permite ao usuário localizar qualquer recurso disponível no ACD a partir do seu nome, independentemente da sua localização física na rede. Essa transparência de localização de recursos dá flexibilidade para que os recursos possam mudar de localização, sem que haja necessidade de alterar seu nome.

Outra facilidade oferecida pelo serviço de *naming* , denominada de ***Yellow Pages***, permite a localização de recursos a partir de seus atributos. Por exemplo, um usuário poderia solicitar através do *Yellow Pages* uma lista das máquinas que possuem impressoras com velocidade de impressão superior a 200 cps.

Em [Malamud92b], [OSF91a] e [Stern91], podem ser encontrados detalhes sobre o serviço de *naming* e a facilidade *Yellow Pages*.

2.2.5. Serviço Distribuído de Tempo

Muitas aplicações necessitam de uma referência de tempo para escalonar suas atividades e determinar a sequência e duração de eventos. Em uma ACD, a existência de várias máquinas leva à existência de múltiplas referências de tempo. Portanto, os diferentes componentes do ACD podem obter tempos diversificados nos computadores da rede.

O objetivo do serviço de tempo é sincronizar os relógios existentes nos vários computadores, fornecendo um **tempo padrão** único e confiável para todas as aplicações executadas no ACD.

Alguns exemplos de aplicações que necessitam do serviço de tempo são: *logs* de bancos de dados e ferramentas tipo *make*, que precisam manter, atualizada, a data de modificação dos

arquivos. Detalhes sobre serviço distribuído de tempo em ACD estão descritos em [Fidge91].

2.2.6. Sistema de Arquivos Distribuído

Em ACDs, é comum os usuários necessitarem compartilhar informações que estão armazenadas em arquivos distribuídos através da rede. O sistema de arquivos distribuído, ou DFS (*Distributed File System/Service*), atende a essa necessidade integrando os arquivos existentes na rede, formando um sistema de arquivos global. Essa integração resulta em um espaço único de nomes e em uma mesma semântica, tanto para arquivos locais quanto para arquivos remotos. O usuário pode acessar os sistemas de arquivos remotos como se estes fossem locais, independentemente de sua localização física na rede. Com o uso do DFS, a distinção entre arquivos locais ou remotos tende a desaparecer.

O DFS normalmente oferece suporte para estações de trabalho sem disco (*diskless*). Através de protocolos bem definidos e de uso geral, os usuários de estações sem disco podem acessar arquivos remotos como se eles existissem na própria estação. O DFS está integrado com outros serviços do ACD, como por exemplo, RPC, *naming* e tempo. Em [Levy90] e em [OSF91b] podem ser encontradas outras informações sobre DFS.

2.2.7. Sistema de Janelas

O sistema de janelas (*Network Windowing System*) é um serviço que permite que a tela da estação de trabalho do usuário seja subdividida em várias partes denominadas de janelas. Cada janela representa uma aplicação que pode estar sendo executada em qualquer máquina da rede. Isso torna possível, para o usuário, acompanhar os resultados da execução de várias aplicações, locais e remotas, simultaneamente nas janelas de uma mesma tela.

Esse serviço é o responsável pela: a) criação e disposição dos objetos gráficos (janelas, menus, ícones) na tela; b) gerência do acesso das aplicações a suas respectivas janelas; c) controle da entrada de dados entre os dispositivos e a janela destino; d) alocação dos recursos necessários

durante a execução de cada aplicação. Em [Gray91] e [Dunphy91] podem ser encontradas maiores informações sobre sistemas de janelas.

2.2.8. Integração com Computadores Pessoais

A tendência atual entre usuários de computadores é para o *downsize*. Essa tendência, em conjunto com a proliferação de computadores pessoais (PCs) de baixo custo, resulta na conexão de um número considerável de PCs às redes. O serviço de integração com PCs permite que o potencial existente nas redes seja estendido para os usuários individuais de microcomputadores. Dessa forma, PCs interconectados por LANs (*Local Area Network*) ou servidores de redes não ficam mais isolados. Por exemplo, aplicações *stand-alone* rodando em um PC podem compartilhar arquivos e periféricos do ACD através do serviço de arquivos distribuídos.

Uma observação importante é o porquê destacar computadores pessoais na arquitetura do ACD, visto que eles são uma máquina como outra qualquer. Isso deve-se ao fato de que toda máquina integrada ao ACD tem que possuir, nela própria, os serviços da arquitetura do ACD. No PC, devido a problemas de funcionalidade do MS-DOS e restrições de *hardware* (ex.: quantidade de memória), coloca-se em disponibilidade somente uma parte mínima da arquitetura do ACD, representada pelos serviços de transporte, RPC e DFS.

2.2.9. Serviço de Transações Distribuídas

Em uma determinada classe de aplicações, é frequente a necessidade de processar operações envolvendo bases de dados distribuídas por diversas máquinas do ACD. A fim de liberar o programador do trabalho de programar, a nível de aplicação, todo o tratamento da integridade desse tipo de operação, principalmente em casos de falhas, existe o serviço de transações distribuídas.

O serviço de transações distribuídas é uma classe de processamento de dados onde cada item de um trabalho, a transação, é caracterizado por quatro propriedades: atomicidade,

consistência, isolamento e durabilidade. A **atomicidade** significa que, ou todas as operações da transação são realizadas, ou nenhuma operação é processada. Isto caracteriza uma semântica do tipo "tudo-ou-nada". A propriedade de **consistência** garante que as operações são processadas corretamente e de forma válida. O **isolamento** garante que resultados parciais de uma transação não são acessíveis, exceto por operações que fazem parte da mesma transação. Finalmente, a propriedade de **durabilidade** garante que nenhuma transação concluída é alterada por qualquer tipo de falha. Informações adicionais podem ser encontradas em [Spector89], [Mafla91], [Khoshafian92] e [UNIX92].

2.2.10. Serviços de Segurança

Na maioria dos sistemas centralizados, o sistema operacional é o responsável pela verificação da identidade do usuário e pela permissão de acesso aos recursos. Em um ACD, essas atividades estendem-se pelos vários computadores da rede, requerendo serviços de segurança independentes e confiáveis. O principal objetivo do serviço de segurança é proteger os dados e recursos do ACD contra acesso, modificação ou destruição por usuários não autorizados. Os serviços de segurança serão objeto de um estudo detalhado na Seção 5.4.

2.2.11. Serviço de Gerência Distribuída

Em um ACD, a existência de diversos ambientes computacionais, com diferentes esquemas de administração, gera a necessidade da utilização de um serviço de gerência global, que integre os esquemas de administração existentes. O serviço de gerência distribuída atende a essa necessidade fornecendo meios para uma administração eficiente e uniforme, dos sistemas, das redes e das aplicações do usuário integradas ao ACD.

Do ponto de vista do usuário final e do administrador do sistema, o serviço de gerência distribuída oferece: a) uniformidade de gerenciamento; b) redução do nível de complexidade para execução de tarefas de gerenciamento; c) melhoria da confiabilidade e disponibilidade dos

sistemas e redes; d) melhoria da habilidade do usuário em utilizar diferentes plataformas; e) facilidades para os usuários se concentrarem mais nas suas aplicações, desprendendo menos tempo e esforço no gerenciamento dos seus sistemas [OSF91a].

Os principais componentes de um serviço de gerência distribuída são: interface do usuário, gerente de aplicações e gerente de recursos.

Para facilitar o gerenciamento, normalmente são colocadas, à disposição do administrador, **interfaces gráficas** (ou orientadas a caractere) através das quais são executadas de forma consistente as tarefas de administração.

O **gerente de aplicações** é o responsável pela execução das tarefas de gerenciamento, tais como, a re-inicialização remota dos nodos da rede e a reconfiguração remota dos parâmetros dos nodos das redes. Esse serviço torna transparente, para os usuários, os detalhes de comunicação, gerenciamento de eventos e manutenção das informações de gerência.

O **gerente de recursos** é responsável pela administração dos recursos disponíveis no ACD, como por exemplo, impressoras, dispositivos, sistemas de correio eletrônico, usuários e aplicativos para o usuário final.

Maiores informações sobre gerência em ACDs podem ser encontradas em [Cypser91], [OSF91a] e [OSF91e].

Somente dispor dos serviços do ACD pode não ser suficiente para o programador desenvolver ADs de qualidade. Ele deve ter em mente os requisitos básicos que todo bom projeto de AD deve atender. Esse é o assunto discutido no próximo capítulo.

Capítulo 3

3. Considerações Básicas de Projeto

Neste capítulo, serão inicialmente apresentados os requisitos básicos que devem ser considerados em projetos de ADs heterogêneas. Finalmente, será dada uma visão geral dos principais fatores responsáveis pelo aumento de complexidade em projetos de ADs.

3.1. Requisitos de Aplicações Distribuídas

Ao projetar uma AD, o programador deve considerar um conjunto básico de requisitos que podem ter impacto direto na qualidade da implementação. Uma meta que todo projeto de AD deve procurar atingir é transparência. Outras metas importantes são consistência e eficiência, obtidas a partir das considerações de heterogeneidade, confiabilidade, escalabilidade e desempenho.

3.1.1. Transparência

Todo projeto de AD deve procurar tornar a existência de computadores, geograficamente distribuídos pela rede, transparente para os usuários. Isso é importante porque, do ponto de vista do usuário, permite o acesso a programas e objetos de dados localizados em qualquer computador da rede, usando os mesmos nomes e operações, independentemente de suas localizações. Para tornar isso possível, os módulos da AD (serviços) devem ser projetados considerando as seguintes possibilidades [Coulouris88]:

• **Transparência de Acesso**: permite que o acesso aos recursos, como por exemplo, arquivos, impressoras, CPUs, banco de dados, processos locais e remotos, seja feito através do

uso de um mesmo conjunto de operações.

• **Transparência de Localização**: permite que os recursos sejam acessados independentemente de sua localização real. Por exemplo, nomes de arquivos dependentes do nome da máquina como *máquina1.notas.txt* devem ser evitados.

• **Transparência de Concorrência**: habilita vários usuários a realizarem operações concorrentemente sobre os dados e recursos compartilhados, sem que haja efeitos de interferência de um sobre o outro.

• **Transparência de Replicação**: admite que múltiplas cópias de arquivos e outros recursos sejam usados para aumentar confiabilidade ou desempenho, sem que os usuários tenham conhecimento da existência de réplicas.

• **Transparência de Falhas**: permite que a ocorrência de falhas de *hardware* ou *software* sejam "escondidas" do usuário, de tal forma que os usuários possam completar suas tarefas independentemente das falhas.

• **Transparência de Migração**: permite a movimentação de recursos entre os vários sistemas sem que as operações do usuários sejam afetadas.

• **Transparência de Escala**: permite que a aplicação seja expandida em escala sem que haja a necessidade de mudanças na estrutura do sistema ou nos algoritmos da aplicação.

3.1.2. Heterogeneidade

Os módulos da AD devem ser projetados de forma a tratarem os efeitos causados pela heterogeneidade inerente ao ACD. O programador deve considerar possíveis problemas causados pelas diferenças entre linguagens de programação, sistemas operacionais, representação de dados, esquemas de segurança e protocolos de comunicação. Um exemplo do que pode acontecer, se isso não for feito, é citado em [Corbin91]: o programa *talk* (utilitário para ambiente UNIX) permite que o usuário de um computador se comunique com o usuário de outro computador. Quando os dois computadores possuem a mesma arquitetura, o *talk* funciona corretamente, mas

apresenta problemas quando os computadores têm arquiteturas diferentes. Nesse caso, o *talk* não funciona adequadamente, porque sua implementação não considera as diferenças de representação de dados existentes entre computadores de arquiteturas diferentes.

O tratamento dos efeitos de heterogeneidade requer o uso do serviço de apresentação do ACD, introduzido no capítulo 2. Outras informações sobre os efeitos e o tratamento de heterogeneidade serão descritos na Seção 4.3.3.1.

3.1.3. Confiabilidade

Como visto no capítulo 1, uma das principais vantagens das ADs é a possibilidade de se obter aplicações de alta confiabilidade, independentemente da ocorrência de falhas. A idéia básica é que, se ocorrer uma falha em um computador que executa tarefas da AD, outros computadores sejam usados para realizar as tarefas que ficaram pendentes. Para obter alta confiabilidade, o programador deve considerar três requisitos: disponibilidade, tolerância a falhas e segurança.

3.1.3.1. Disponibilidade

A disponibilidade pode ser definida como a quantidade de tempo contínuo que a AD é utilizável. Um meio de se aumentar a disponibilidade é através da **replicação** dos módulos vitais da AD (*hardware* e *software*). Se qualquer recurso usado pela AD falhar, as réplicas podem ser usadas para substituí-lo, permitindo que o usuário continue executando normalmente suas tarefas.

Para se obter boa disponibilidade, não basta apenas ter as réplicas disponíveis, é necessário que elas estejam íntegras quando requisitadas. Quanto maior o número de réplicas usadas, maior a probabilidade da existência de réplicas inconsistentes, principalmente nos casos em que a frequência de atualizações é muito alta. Portanto, o programador deve tomar os cuidados necessários, para manter as réplicas atualizadas, a fim de evitar problemas de inconsistência. O uso de técnicas de replicação será objeto de estudo nas Seções 5.2.2.5 e 5.3.3.

3.1.3.2. Tolerância a Falhas

Dotar a AD de alta disponibilidade não é condição suficiente para que ela tenha alta confiabilidade. São necessários cuidados adicionais para garantir que, em caso de falhas, não exista o comprometimento da integridade das operações já realizadas pela AD e nem riscos para as operações futuras. Uma AD que tem a propriedade de continuar funcionando corretamente, após a ocorrência de falhas, é considerada uma aplicação tolerante a falhas. Quando se deve ou não dotar a AD dessa propriedade, depende essencialmente dos requisitos do problema que está sendo modelado. Um exemplo prático de tolerância a falhas é a solução adotada pelo protocolo do sistema de arquivos distribuídos NFS (*Network File System*) da Sun Microsystems [Stern91]. O NFS evita inconsistências, após a ocorrência de falhas, simplesmente não mantendo nenhum tipo de estado. A ausência de estado permite que as operações sejam reiniciadas após uma falha, como se nada tivesse ocorrido.

3.1.3.3. Segurança

Projetar uma AD segura requer cuidados especiais principalmente com as operações de comunicação entre os processos remotos. Essa comunicação geralmente é feita através da troca de mensagens transmitidas através da rede. Em muitos casos, um processo da AD que esteja sendo executado em um computador remoto não pode simplesmente assumir que as mensagens que recebe têm origem confiável. Isso ocorre, porque existe a possibilidade das mensagens serem capturadas e falsificadas, por usuários mal intencionados que estejam monitorando a rede. Analisando e modificando o conteúdo dessas mensagens, o falsário pode personificar um usuário autorizado, para acessar serviços restritos ou causar danos a dados armazenados. Quando necessário, os serviços de segurança do ACD devem ser usados para garantir uma comunicação segura, evitando problemas dessa natureza. Considerações detalhadas sobre a utilização de serviços de segurança em ADs serão feitas na Seção 5.4.

3.1.4. Escalabilidade

Uma AD deve ser projetada para funcionar corretamente com desempenho aceitável tanto em um ACD com duas máquinas quanto em um ACD com milhares de máquinas. Essa característica é conhecida como escalabilidade. Um projeto escalável deve gerenciar os picos de carga de forma elegante, adaptando-se ao crescimento do número de usuários e permitindo fácil integração com novos recursos adicionados ao ACD.

A ausência de escalabilidade pode causar vários problemas. Por exemplo, um processo extremamente sobrecarregado pode ficar paralisado e ser confundido por um esquema de recuperação de falhas como um processo que falhou. O chaveamento da carga de trabalho do processo que "falhou" para uma réplica (caso exista alguma) pode causar nova sobrecarga, não resolver o problema, e pior ainda, propagá-lo para outros processos. Um exemplo prático de problemas com escalabilidade será descrito nas Seções 6.4.5.5 e 6.5.4.5.

3.1.5. Desempenho

Uma AD pode atender aos requisitos de transparência, confiabilidade, escalabilidade, mas se o desempenho for crítico, a sua utilização pode se tornar inviável. O maior problema a ser contornado pelo programador é a relação existente entre o desempenho da AD e o desempenho da infra-estrutura de comunicação. A origem do problema está na troca de mensagens através da rede (normalmente lenta), necessária para a comunicação entre os processos que rodam em diferentes computadores.

Uma forma de otimizar o desempenho de uma AD é reduzir o número de mensagens trocadas entre os processos. Contraditoriamente em alguns casos, a melhoria de desempenho é obtida com a execução de vários processos remotos em paralelo, o que pode causar um aumento substancial do número de mensagens trocadas entre os processos.

O uso de técnicas para melhoria do desempenho de ADs é importante, porque aumenta a eficiência e a velocidade da aplicação, proporcionando um melhor balanceamento da carga e

reduzindo os custos de comunicação. Na Seção 5.3, serão apresentadas várias considerações para a melhoria do desempenho de ADs.

3.2. Fontes de Complexidade

O conhecimento dos fatores que causam aumento da complexidade no desenvolvimento de ADs é importante, visto que esses fatores podem impor limitações ao projeto. As considerações a seguir mostram alguns desses fatores e explicam por que nem sempre, soluções simples são possíveis de serem adotadas. Algumas dessas considerações também se destinam a aplicações centralizadas, mas suas dimensões, diversidade e consequências são muito mais críticas em ADs, devido à existência de múltiplas máquinas e de rede. Os principais fatores responsáveis pelo aumento de complexidade, citados em [Mullender89], são: falha parcial, propagação de efeito, interferência e efeito de escala.

3.2.1. Falha Parcial

Uma das vantagens das ADs, citada no capítulo 1, é a sua capacidade inerente de tolerar falhas. Para explorar esse potencial, é necessário que a AD seja projetada para se recuperar de **falhas parciais**. Uma falha parcial ocorre, quando um ou mais processos da AD falham, enquanto os processos restantes continuam a trabalhar. O tratamento de falhas parciais normalmente é fonte considerável de complexidade e representa uma parte considerável do código da AD. Na Seção 5.2, serão sugeridas algumas técnicas para o tratamento de falhas parciais.

3.2.2. Propagação de Efeito

O programador deve considerar a possiblidade dos efeitos de alguns tipos de erros se propagarem através do ACD, causando efeitos indesejáveis em outras aplicações que não tenham relação com o problema ocorrido. Um exemplo de propagação de efeitos é um problema que

ocorreu na rede ARPANET[1] decorrente de um erro de projeto. Esse erro levou o nodo processador de mensagem a informar, para os nodos vizinhos, que ele poderia encaminhar mensagens para qualquer outro nodo em tempo zero. Em alguns minutos, um grande número de nodos passou a enviar todo o tráfego para o nodo que estava com o problema, causando um congestionamento que levou à paralização de toda a rede.

3.2.3. Interferência

Outra fonte de complexidade é a relação entre processos remotos que não apresentam problemas quando vistos isoladamente, mas que podem ter um comportamento não previsto quando combinados. O programador deve procurar identificar e tratar efeitos indesejáveis que podem ocorrer quando os processos da AD são executados em paralelo. Esses efeitos, denominados interferências, se originam do tratamento inadequado da sincronização entre processos executados concorrentemente. O tratamento de interferências é importante porque pode evitar a ocorrência de *deadlock*, um problema complexo de ser tratado em ACDs.

Tratar interferências em ACDs não é simples porque as técnicas de sincronização convencionais, como semáforos ou monitores, não podem ser usadas diretamente, visto que normalmente não existe memória compartilhada implícita entre os processadores dos computadores do ACD. Soluções adequadas para resolver esse problema geralmente fazem uso de algoritmos distribuídos e de um esquema de troca de mensagens entre os processos da AD, aumentando a complexidade do projeto.

3.2.4. Efeito de Escala

Uma AD pode funcionar corretamente em cinco nodos da rede e falhar quando o número de nodos cresce exponencialmente. Isto pode ser causado pelo uso de algum recurso não escalável que se torna um gargalo para o sistema, ou pelo uso de algoritmos não escaláveis. Projetar ADs com algoritmos escaláveis, na dimensão que alguns problemas requerem, contribui

[1] A ARPANET foi uma rede militar americana desativada em 1990 e, posteriormente, substituída pela Defense Data Network.

para o aumento da complexidade do projeto da aplicação.

Uma vez que o programador tem em mente os requisitos mínimos que uma AD deve atender, é necessário que ele faça opção por um modelo de programação adequado aos requisitos da aplicação a ser implementada. O próximo capítulo apresenta algumas alternativas.

Capítulo 4

4. Modelos para Projeto de Aplicações Distribuídas

Neste capítulo, serão discutidos os principais modelos utilizados para o desenvolvimento de ADs. Inicialmente será apresentado o modelo conceitual cliente-servidor, atualmente o mais utilizado para modelagem de ADs. Em seguida, serão discutidos os níveis de abstração de três modelos que se baseiam no modelo cliente-servidor: troca de mensagens, chamada de procedimento remoto e invocação de objetos remotos. Finalmente, serão descritos cada um desses modelos.

4.1. Modelo Conceitual Cliente-Servidor

Nesta seção, será apresentado o modelo cliente-servidor e uma classificação dos principais tipos de servidores.

O modelo cliente-servidor (C-S) é um modelo padrão para o desenvolvimento de aplicações em rede e, atualmente, é a base conceitual mais utilizada para a modelagem de ADs. A essência do modelo C-S é de estruturar a AD em dois grupos distintos de processos cooperativos. O primeiro grupo, chamado de **servidor**, oferece serviços para os usuários do segundo grupo, denominado **cliente**. No contexto de ADs, os termos cliente e servidor são sinônimos de processo cliente e processo servidor, respectivamente, e não necessariamente máquina cliente e máquina servidora. Os clientes e servidores normalmente são executados em máquinas diferentes, mas não existem restrições para que eles residam conjuntamente na mesma máquina.

O cliente e o servidor geralmente interagem através de um protocolo simples do tipo pedido/resposta (**Fig. 4-1**). O cliente envia um pedido de serviço através da rede para o servidor

executar. O servidor recebe o pedido e executa o serviço solicitado. Após a execução, o servidor retorna, para o cliente, o resultado ou um código de erro indicando o motivo pelo qual o processamento não foi realizado.

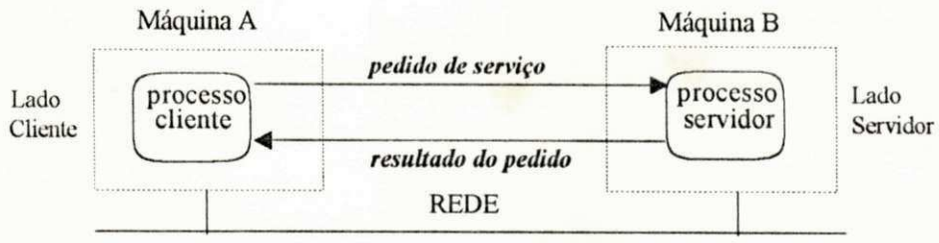


Figura 4-1. *Modelo Cliente-Servidor*

A maior vantagem do modelo C-S é a sua simplicidade o que, naturalmente, resulta em eficiência: o cliente envia um pedido e obtém uma resposta. Geralmente, nenhuma conexão tem que ser estabelecida durante a interação entre o cliente e o servidor. O cliente pode usar a resposta do servidor, para confirmar a execução do pedido.

4.1.1. Tipos de Servidores

Os servidores podem ser classificados quanto a **estratégia de serviço** como: servidor iterativo e servidor concorrente; e quanto a **manutenção de estado** como: servidor com estado e servidor sem estado. A combinação desses servidores pode dar origem a outros tipos de servidores, como por exemplo: concorrente com estado, concorrente sem estado, interativo com estado e iterativo sem estado.

4.1.1.1. Servidor Iterativo

O servidor iterativo é um processo simples que possui um único *thread*, para receber e processar pedidos dos clientes. A existência de apenas um *thread* faz com que somente um pedido seja atendido de cada vez. O servidor iterativo normalmente usa uma fila, para ordenar e controlar a execução dos pedidos. Esse tipo de servidor é viável para serviços que são sempre executados

em um curtíssimo espaço de tempo, como por exemplo, o fornecimento da data e hora de uma máquina remota.

4.1.1.2. Servidor Concorrente

Um servidor concorrente cria múltiplos *threads*, ou novos processos, para atender paralelamente a vários pedidos dos clientes. Por exemplo, se o servidor concorrente recebe um pedido para ler um grande arquivo, ele cria um novo *thread* para processar a leitura, ficando livre para atender a outros pedidos.

4.1.1.3. Servidor com Estado

Um servidor com estado necessita manter alguma informação referente aos pedidos dos clientes para atendê-los corretamente. Essas informações, ou **estado**, são usadas pelo servidor para dar continuidade, de forma íntegra, às operações solicitadas pelo cliente em pedidos futuros.

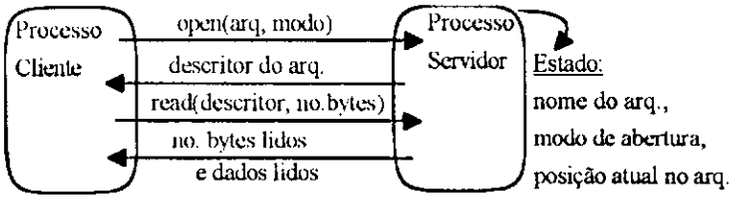


Figura 4-2. *Exemplo de servidor de arquivos com estado [Corbin91]*

A **Fig. 4-2** mostra um servidor de arquivos que oferece os serviços de *open e read*. O cliente envia, para o servidor, um pedido de abertura do arquivo. O servidor recebe o pedido que contém o nome e o modo de abertura, abre o arquivo, e retorna, para o cliente, o descritor do arquivo aberto. De posse do descritor de arquivo, o cliente pode usá-lo para as operações de leitura subsequentes. O servidor é responsável pela manutenção do estado do arquivo cuja abertura foi solicitada pelo cliente. O estado nesse caso inclui o nome do arquivo, o modo de abertura e posição atual de leitura no arquivo, isto é, o último *seek* efetuado.

4.1.1.4. Servidor sem Estado

Um servidor sem estado não precisa manter nenhuma informação sobre qualquer pedido dos seus clientes para funcionar corretamente. A **Fig. 4-3** apresenta um servidor sem estado que possui apenas o serviço *read*. Não existe o serviço *open* que caracterizaria a existência de estado. O cliente envia, a cada pedido de leitura, todas as informações necessárias para a leitura (nome do arquivo, *seek* e número de *bytes* para leitura). O cliente é responsável pelo controle das operações de leitura no servidor. Isso significa que o estado na realidade é mantido pelo cliente.

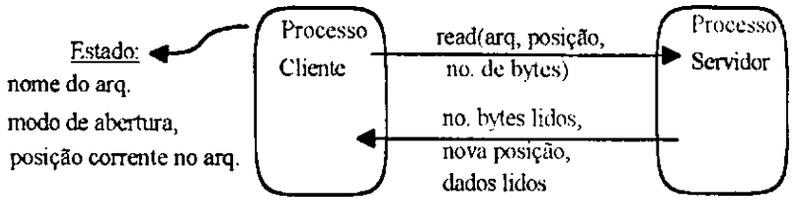


Figura 4-3. *Exemplo de um servidor de arquivos sem estado [Corbin91]*

4.1.2. Níveis de Abstração e Modelos de Programação

O modelo C-S serve de base conceitual para vários modelos de programação de ADs. Dependendo da forma como a comunicação entre os clientes e os servidores é efetuada, esses modelos oferecem maior ou menor nível de abstração para a modelagem de ADs. Quanto maior o nível de abstração, mais natural o modelo de programação e menos visível a existência do modelo C-S e das camadas inferiores da arquitetura do ACD.

Os paradigmas de programação de ADs relacionados com o modelo C-S a serem descritos nas próximas seções são: troca de mensagens, chamada de procedimento remoto e invocação de objetos remotos. O menor nível de abstração é proporcionado pelo modelo de troca de mensagens. Um nível de abstração intermediário é oferecido pelo modelo de chamada de procedimento remoto e o nível mais elevado é proporcionado pelo modelo de invocação de objetos remotos.

4.2. Modelo de Troca de Mensagens

Nesta seção, será apresentado o modelo para programação de ADs de troca de mensagens, considerando o uso de mensagens síncronas e assíncronas.

No modelo de troca de mensagens, a comunicação entre os clientes e os servidores é geralmente feita através de duas primitivas: ***send*** para enviar mensagens e ***receive*** para receber mensagens. A partir dessas primitivas, a comunicação estabelecida entre o cliente e o servidor pode ser feita com dois tipos de mensagens: **síncronas e assíncronas**.

4.2.1. Mensagem Síncrona

No esquema de mensagem síncrona, existem dois pontos de vista a serem considerados: o ponto de vista do sistema operacional e o ponto de vista do programador. Do ponto de vista do sistema operacional, quando o cliente envia uma mensagem síncrona para o servidor usando *send*, ele tem o seu controle suspenso até que a mensagem seja totalmente enviada para o servidor. Enquanto a mensagem estiver sendo enviada, o cliente fica bloqueado e não executa o comando seguinte ao *send* (**Fig. 4-4**). De forma análoga, uma chamada a um *receive* não libera o controle do cliente enquanto a mensagem não for integralmente recebida.

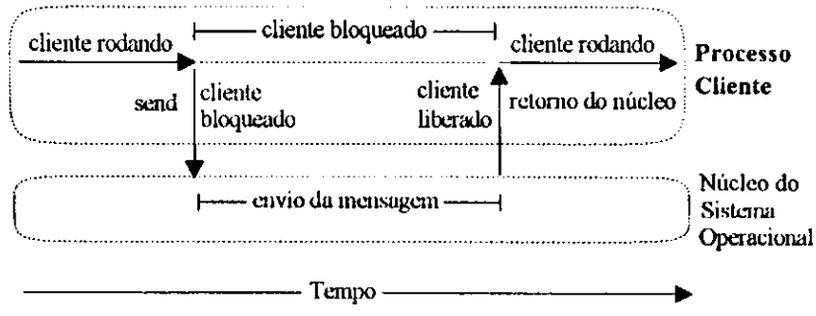


Figura 4-4. *Comunicação com mensagem síncrona*

Do ponto de vista do programador, após o envio de uma mensagem síncrona com *send*, o controle do cliente fica suspenso até que o servidor **confirme** (*acknowledgement*) o recebimento da mensagem.

4.2.2. Mensagem Assíncrona

Quando a comunicação é estabelecida com mensagem assíncrona, o controle retorna para o cliente antes da mensagem ser totalmente enviada ao servidor. O cliente fica bloqueado somente um curto espaço de tempo (desprezível), necessário apenas para copiar a mensagem para um *buffer* do núcleo do sistema operacional. Imediatamente após o *send*, o cliente retoma o processamento paralelamente com a operação de envio da mensagem feita pelo núcleo do sistema operacional (**Fig. 4-5**).

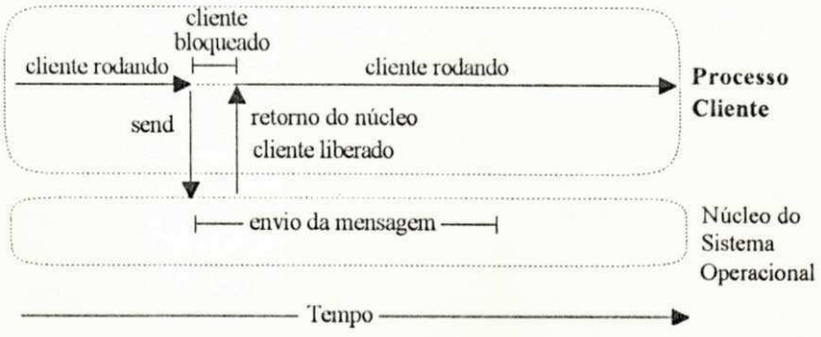


Figura 4-5. *Comunicação com mensagem assíncrona*

O desenvolvimento de ADs, baseadas em esquemas de troca de mensagens, é indicado para ADs cujos requisitos envolvem detalhes de baixo nível da rede. A necessidade do uso de primitivas tipo *send* e *receive* obriga o programador a ter visão de todo o tratamento de envio e recebimento das mensagens. O esquema de troca de mensagens não é um conceito natural para programador que não é de um ambiente de rede. O ideal é usar um modelo que possibilite ao programador não especialista em rede, desenvolver ADs de forma semelhante a aplicações centralizadas. Uma alternativa é um modelo de mais alto nível de abstração, como o de chamada de procedimento remoto, apresentado a seguir.

4.3. Modelo de Chamada de Procedimento Remoto

Nesta seção, será descrito o modelo conceitual de chamada de procedimento remoto, bem como detalhes de funcionamento do seu mecanismo. Também serão discutidas as principais

particularidades que, implicitamente, diferenciam uma chamada de procedimento remoto de uma chamada de procedimento local.

A idéia básica de Chamada de Procedimento Remoto, ou RPC (*Remote Procedure Call*), é estender o uso de chamada de procedimento para ambientes distribuídos. O modelo RPC permite ao cliente chamar um procedimento remoto que será executado em outro processo, e que pode estar em outra máquina. O modelo de RPC é mais fácil de programar se comparado com as tradicionais interfaces de baixo nível (mensagens) para programação em rede, visto que oferece um maior nível de abstração.

Existem pelo menos três razões para usar RPC como mecanismo primário de comunicação entre os processos das ADs. A primeira é **simplicidade**, devido à familiaridade das RPCs com chamadas de procedimento local. Outra razão é **generalidade**: existe um grande número de aplicações não distribuídas que utilizam chamada de procedimento local, como mecanismo para comunicação entre partes da aplicação. A terceira razão é **eficiência**: esquemas de chamadas de procedimentos são simples o suficiente para possibilitar uma comunicação relativamente rápida [Birrell84].

4.3.1. Modelo Conceitual

Conceitualmente, o modelo de RPC não é complexo (**Fig. 4-6**). Quando um procedimento remoto é chamado, o controle do cliente é temporariamente suspenso e uma mensagem, contendo os parâmetros do procedimento, é enviada ao servidor. Após receber a mensagem, o servidor executa o procedimento remoto solicitado, retornando para o cliente uma mensagem com o resultado do processamento efetuado. Após receber a mensagem com o resultado do servidor, o cliente retoma o processamento a partir da instrução seguinte à RPC. Do ponto de vista do cliente, a RPC é idêntica a uma chamada de procedimento local.

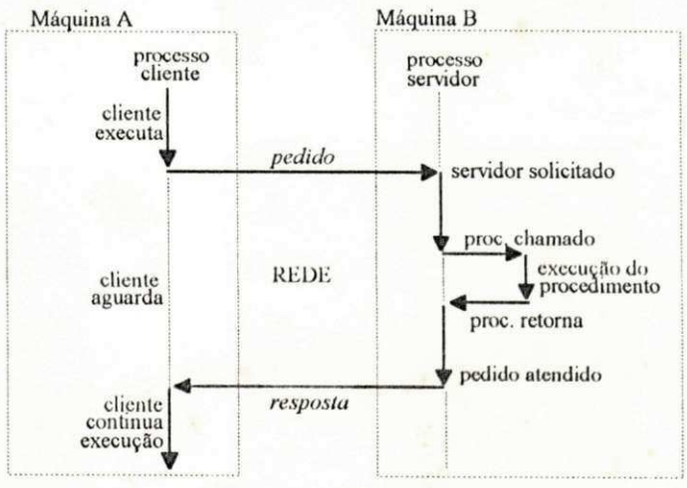


Figura 4-6. *Modelo de Chamada de Procedimento Remoto (RPC)*

4.3.2. Mecanismo de Chamada Remota

Para esconder do programador a complexidade da rede, oferecendo um melhor nível de abstração, as implementações do modelo RPC utilizam um **mecanismo RPC**. Um exemplo clássico é o mecanismo originalmente implementado por [Birrell84]. Ele implementa RPCs, tipicamente síncronas, mas atualmente existem várias implementações que permitem RPCs assíncronas. Uma **RPC síncrona** mantém o controle do processo que faz a RPC aguardando, até que seja retornado o resultado do servidor. Uma **RPC assíncrona** libera o controle do processo que fez a chamada imediatamente após a RPC, sem aguardar pela resposta do servidor.

O estudo do mecanismo RPC é importante, porque, para resolver uma determinada classe de problemas, o programador tem que ter acesso a alguns dos detalhes que o mecanismo esconde. As operações típicas de um mecanismo RPC são mostradas na **Fig. 4-7** e ocorrem na ordem em que estão enumeradas.

1. O cliente chama um procedimento local, chamado *stub* cliente. Do ponto de vista da aplicação, o *stub* cliente emula o servidor real do procedimento a ser executado. O *stub* cliente empacota os parâmetros do procedimento remoto, possivelmente em algum formato padrão, e constrói uma ou mais mensagens que serão enviadas ao servidor.

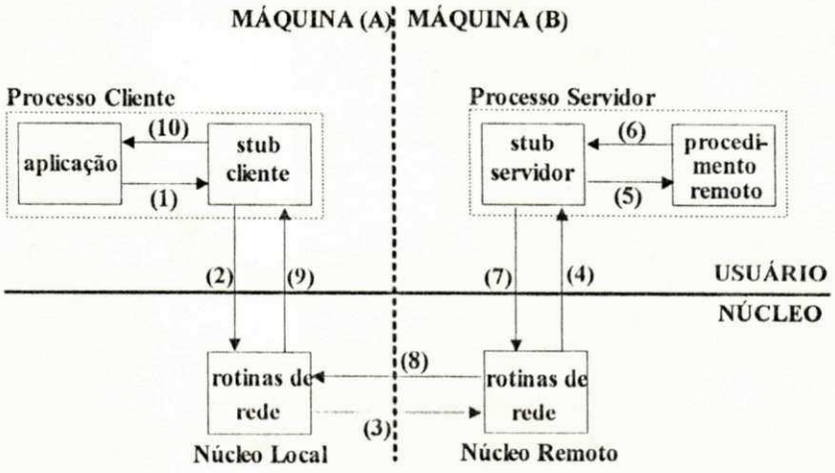


Figura 4-7. *Operações realizadas pelo mecanismo RPC*

2. O *stub* cliente envia a mensagem para o sistema remoto, através de um *system call*, feito ao núcleo do sistema operacional local.

3. O núcleo local envia a mensagem para o núcleo do sistema remoto através da rede.

4. O núcleo do sistema remoto recebe a mensagem e a envia para o *stub* servidor.

5. O *stub* servidor desempacota os parâmetros e chama o servidor para executar o procedimento remoto.

6. Após concluir a execução do procedimento requerido, o servidor retorna o resultado para o *stub* servidor.

7. O *stub* servidor empacota o resultado em uma mensagem e faz um *system call* ao núcleo remoto, para enviar a mensagem ao núcleo do cliente.

8. O núcleo remoto envia a mensagem para o núcleo do cliente.

9. O núcleo do cliente recebe a mensagem e a envia para o *stub* cliente.

10. O *stub* cliente desempacota o resultado e o retorna para a aplicação.

4.3.3. Particularidades das Chamadas Remotas

Idealmente, uma RPC deveria ser, em todos os aspectos, idêntica a uma chamada de procedimento local. O mecanismo RPC procura tornar detalhes como *system calls*, comunicação

de rede e conversões de dados, transparentes para o programador. Independentemente desse fato, existem várias particularidades que devem ser conhecidas pelo programador de AD, tais como: passagem de parâmetros, variáveis globais, *binding*, protocolo de transporte, semântica de chamadas remotas, desempenho e gerenciamento de erros.

4.3.3.1. Passagem de Parâmetros

A sintaxe de passagem dos parâmetros em uma RPC é igual a de uma chamada de procedimento local. Apesar disso, existem duas diferenças que devem ser consideradas pelo programador: tratamento da heterogeneidade e passagem de parâmetros por referência.

- **Tratamento da Heterogeneidade**: em grandes ACDs é comum a existência de computadores de diversos fabricantes. Cada computador pode ter uma representação própria para seus tipos de dados. Por exemplo, computadores pessoais tipo IBM-PC usam código de caracteres ASCII, enquanto que *mainframes* IBM usam representação EBCDIC. Consequentemente, não faz sentido passar diretamente parâmetros do tipo caractere de um cliente IBM-PC para o servidor de um *mainframe* IBM. O mesmo problema ocorre com a passagem de parâmetros do tipo inteiro e de ponto flutuante entre arquiteturas de *hardware* diferentes, como por exemplo, INTEL e SPARC.

Uma solução para tratar os problemas de heterogeneidade dos parâmetros das RPCs é utilizar o serviço de apresentação do ACD. O serviço de apresentação converte os parâmetros das RPCs para um formato padrão de dados, conhecido como **forma canônica**. Todo o processo de conversão (e desconversão) para a forma canônica é feito pelos *stubs* cliente e servidor.

Em muitas implementações de RPC, o programador faz apenas uma simples especificação das estruturas de dados da aplicação, geralmente como em linguagem "C". Então, com o auxílio de um compilador, são gerados automaticamente *stubs* adequados à forma canônica. Isso reduz as chances de erros e torna as diferenças de representação interna de dados, transparentes para os processos da AD.

• **Passagem de Parâmetros por Referência**: o fato do cliente e do servidor residirem em espaços de endereçamento disjuntos, possivelmente em máquinas diferentes, torna a passagem de apontadores ou passagem de parâmetros por referência, sem sentido. Por exemplo, o endereço de um *buffer*, no espaço de endereçamento do cliente, pode representar, no espaço de endereçamento do servidor, o endereço de um segmento de pilha ou texto. Por esse motivo, muitos mecanismos de RPC só permitem a passagem de parâmetros por valor. Todos os parâmetros e resultados são sempre copiados pelos *stubs*, para as mensagens trocadas entre o cliente e o servidor.

Privar o programador do uso de apontadores em RPCs é extremamente indesejável, já que seu uso é comum na maioria das aplicações. Alguns mecanismos RPC contornam esse problema, adotando um esquema de **cópia e restauração** em substituição à passagem por referência.

Um exemplo de como funciona essa solução é o de uma RPC cujo parâmetro é um apontador para um *array* de caracteres. Ao ser feita a RPC, o *stub* cliente copia o *array* para a mensagem enviada ao servidor. O *stub* servidor recebe a mensagem e armazena o *array*, em um novo endereço do seu espaço de endereçamento. Em seguida, o *stub* servidor executa o procedimento requisitado, passando, como parâmetro, o endereço do *array* no seu espaço de endereçamento. Todas as mudanças feitas no *array*, durante a execução do procedimento, afetam diretamente o *buffer* da mensagem gerenciada pelo *stub* servidor. Quando a execução termina, uma mensagem contendo o *array* atualizado é enviada de volta para o *stub* cliente, que, por sua vez, atualiza o *array* no seu espaço de endereçamento.

Embora essa solução não seja uma passagem por referência real, ela é suficiente para resolver a maioria dos problemas. Maiores detalhes sobre passagem de parâmetros de RPCs podem ser encontradas em [Wilbur87] e [Birrell84].

4.3.3.2. Variáveis Globais

Ao contrário de procedimentos locais, um procedimento remoto, normalmente, não tem

acesso a nenhuma variável global, declarada no espaço de endereçamento do cliente. Isso ocorre, obviamente, porque o procedimento remoto é sempre executado em um espaço de endereçamento disjunto do espaço de endereçamento do cliente. Para que o procedimento remoto tenha acesso ao valor de uma variável global declarada pelo cliente, uma cópia do valor da variável deve ser passada explicitamente como um dos parâmetros da RPC. Uma exceção são ACDs que oferecem serviços de mémoria distribuída. Nesses ambientes, a visibilidade de variáveis globais funciona de forma análoga à de um sistema centralizado. O programador pode definir explicitamente variáveis globais visíveis por todos clientes e servidores da AD. Um estudo detalhado sobre serviços de memória compartilhada distribuída pode ser encontrado em [Nitzberg91] e [Carter91].

4.3.3.3. Binding

Uma importante atribuição do mecanismo RPC é localizar o servidor que possui o procedimento remoto (serviço) que o cliente deseja executar. Para fazer a RPC, é necessário que o cliente obtenha, a partir do nome do serviço, o endereço de transporte do serviço. O processo de mapeamento do nome de um serviço, para o seu respectivo endereço de transporte, é conhecido como ***binding***. Duas operações básicas necessárias para realizar o *binding* são: a) encontrar a máquina remota onde reside o servidor desejado; b) encontrar o processo servidor correto na máquina selecionada. As soluções mais comuns para realizar essas operações são: *binding* estático e *binding* dinâmico.

• **Binding Estático**: nesse caso, o *binding* para o endereço de transporte do servidor está embutido no código do cliente. O cliente obtém o endereço de transporte a partir de um arquivo ou o solicita diretamente ao usuário da aplicação. Essa solução é extremamente inflexível e apresenta uma grande desvantagem: se o servidor for movido para outra máquina ou se sua interface for modificada, então provavelmente vários programas terão que ser alterados e recompilados.

• **Binding Dinâmico**: para eliminar as limitações do *binding* estático, muitos mecanismos RPC utilizam um servidor de nomes, ou ***binder*** [Coulouris88]. O *binder* é um processo que mantém os registros da localização (e dos serviços) de todos os servidores existentes. Quando um servidor é inicializado, ele registra no *binder* sua localização e seus serviços (nome e número da versão do serviço). O *binder* armazena essas informações em um banco de dados. Essa operação é conhecida como **exportação**. Durante o processo de exportação, o *binder* é facilmente localizado pelos servidores, porque normalmente possui um endereço fixo e bem conhecido.

A operação de exportação é necessária, para tornar possível ao cliente, localizar posteriormente os procedimentos remotos. Quando o cliente chama um procedimento remoto, o *stub* cliente não conhece a localização do servidor. Para localizar o servidor, o *stub* cliente envia uma mensagem ao *binder*, contendo o nome e a versão do serviço requerido (**importação**). O *binder* verifica que servidores exportaram serviços com o mesmo nome e versão. Se não existir nenhum servidor registrado com o serviço requerido, ou se os parâmetros do serviço encontrado não coincidirem com os fornecidos pelo *stub* cliente, então a RPC falha. No caso do *binder* obter sucesso, ele retorna para o *stub* cliente o endereço de transporte do servidor, possibilitando ao *stub* cliente dar continuidade à chamada do procedimento remoto.

O *binding* dinâmico possui a desvantagem de introduzir *overhead* com as operações de importação e exportação, penalizando o desempenho das RPCs. Normalmente isso ocorre, quando muitos clientes acessam o *binder* simultaneamente, mas esse problema pode ser minimizado com o uso de múltiplos *binders*.

4.3.3.4. Protocolo de Transporte

Muitos mecanismos de RPC permitem ao programador definir o serviço de transporte que dará suporte à comunicação entre os clientes e os servidores. De forma geral, os serviços de transporte utilizam protocolos que podem ser classificados em dois grupos: orientado a conexão e sem conexão.

• **Protocolo Orientado a Conexão**: garante uma comunicação confiável com a abertura de uma **conexão** entre o cliente e o servidor, antes dos dados serem trocados. A troca de dados, através de um protocolo orientado a conexão, é similar a uma conversação telefônica. Os dados são trocados somente após a abertura da conexão (ligação). A conexão garante que os dados cheguem íntegros ao seu destino e na mesma ordem em que são enviados. Um exemplo de protocolo orientado a conexão bastante utilizado atualmente é o *Transmission Control Protocol*, ou TCP.

• **Protocolo Sem Conexão**: ao contrário do protocolo orientado a conexão, esse tipo de protocolo não é confiável. Isto se deve ao fato de que a troca de dados entre o cliente e o servidor é feita por intermédio de mensagens individuais independentes, conhecidas como ***datagramas***, e não através de uma conexão previamente estabelecida. No protocolo sem conexão, não existe nenhum mecanismo para o controle do fluxo das mensagens, o que pode resultar em mensagens fora de ordem, perdidas ou duplicadas.

Devido ao fato de não implementar confiabilidade, o desempenho do protocolo sem conexão é normalmente superior ao do protocolo orientado a conexão. O *User Datagram Protocol* (UDP) é um exemplo de protocolo sem conexão.

4.3.3.5. Gerenciamento de Exceções

Em uma chamada de procedimento local, existe um número limitado de problemas que podem ocorrer, e dos quais normalmente, os usuários tomam conhecimento - uma referência inválida à memória, uma divisão por zero e assim por diante. Com RPC, a probabilidade da ocorrência de erros aumenta. Não apenas o próprio procedimento remoto, no servidor, pode gerar um erro, como também o *stub* cliente e o *stub* servidor podem encontrar problemas na rede. Os detalhes que envolvem detecção, tratamento e recuperação de erros em RPCs serão discutidos na Seção 5.2.

4.3.3.6. Semântica de Chamadas Remotas

Quando um procedimento local é chamado e um resultado é retornado para quem fez a chamada, não existem dúvidas de que ele foi executado exatamente uma vez. Entretanto, no caso das RPCs, se o cliente recebe o resultado do servidor, isso não significa necessariamente que o procedimento remoto tenha sido executado exatamente uma vez. Isso normalmente ocorre, quando as RPCs utilizam serviços de transporte não confiáveis (veja Seção 5.2.2.1). Em função disso, existem várias semânticas possíveis para as RPCs; as principais são: exatamente-uma-vez, no-máximo-uma-vez e pelo-menos-uma-vez.

- **Exatamente-uma-vez**: significa que o procedimento remoto é executado uma única vez. Esse tipo de operação é difícil de se obter, porque existe a possibilidade de falha do servidor.
- **No-máximo-uma-vez**: significa que o procedimento remoto ou é executado exatamente uma vez ou não é executado nenhuma vez. Quando o cliente executa a RPC com sucesso, o procedimento remoto é executado uma vez. Por outro lado, se por algum motivo a RPC falhar, o cliente não sabe ao certo se o procedimento remoto foi ou não executado.
- **Pelo-menos-uma-vez**: significa que o procedimento remoto é executado, no mínimo, uma vez. Assumindo que o cliente retransmite um mesmo pedido até receber uma resposta válida do servidor, então existe a possibilidade do procedimento remoto ser executado mais de uma vez.

4.3.3.7. Desempenho

As RPCs envolvem comunicação de processos através da rede, conversão de dados e outros detalhes que, seguramente, tornam o seu desempenho inferior ao de uma chamada de procedimento local. Normalmente, a chamada local é pelo menos dez vezes mais rápida do que uma RPC [Stevens90]. Apesar disso, é importante observar que as RPCs devem ser vistas como uma estratégia para simplificação da programação em rede, e não como substituta para as chamadas de procedimentos locais. Em [Birrell84], são apresentadas algumas medidas de desempenho de um mecanismo RPC em uma rede Ethernet de 2.94 megabits/segundo. As

medidas foram obtidas a partir da execução de RPCs com diferentes números de argumentos e resultados e, em seguida, comparadas com chamadas locais. Para um procedimento chamado com dois parâmetros de 16 bits, e que retorna como resultado dois argumentos de 16 bits, a chamada local foi 97 vezes mais rápida que a remota. Já o teste com um *array* de 40 palavras de 16 bits, e com retorno de um resultado também em um *array* de 40 palavras de 16 bits, a chamada local foi 33 vezes mais rápida que a remota. É importante observar que, no segundo caso, a chamada remota teve melhoria substancial de desempenho se comparada ao primeiro caso.

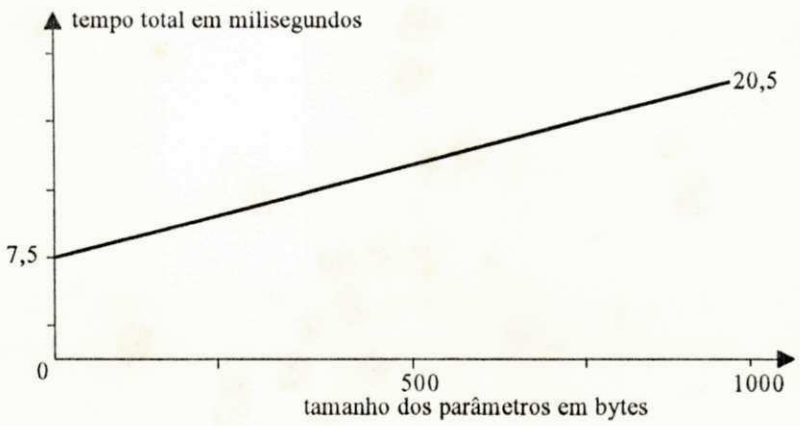


Figura 4-8. *Desempenho típico de RPCs*

Em [Wilbur87], são apresentados resultados do desempenho de RPCs entre dois processos executados em um ambiente com estações de trabalho Sun 3. A rede utilizada foi uma *Ethernet* de 10 Mbit por segundo. A **Fig. 4-8** mostra o tempo médio das RPCs em função da variação do tamanho em *bytes* dos parâmetros. Uma RPC sem parâmetros ou resultados leva cerca de 7,5 milisegundos para ser executada. Já uma RPC com parâmetros e resultados leva cerca de 13 milisegundos por *byte* transmitido para ser executada.

4.4. Modelo de Invocação de Objetos Remotos

Nesta seção, serão apresentados o modelo de invocação de objetos remotos e sua relação com o modelo conceitual C-S. Serão descritos detalhes da arquitetura e do funcionamento de um mecanismo de invocação de objetos remotos, incluindo aspectos de localização, gerenciamento e

comunicação de objetos. Finalmente serão feitas considerações específicas sobre classes de invocação e passagem de parâmetros entre objetos remotos.

O modelo de invocação de objetos remotos, ou modelo de objetos distribuídos, permite o desenvolvimento de ADs heterogênas baseadas na filosofia de orientação a objetos. Esse modelo oferece um modelo de programação de nível de abstração superior ao do modelo RPC.

Do ponto de vista do programador de ADs, os principais ganhos obtidos com o uso do modelo de invocação de objetos são: grande poder representacional, reusabilidade, encapsulamento de informações e desenvolvimento modular. Esse conjunto de fatores, herdados da filosofia de objetos, proporcionam um alto nível de abstração que permite ao programador se concentrar mais na funcionalidade da AD e menos nos detalhes da infra-estrutura computacional. Em função disso, a tarefa de programação é simplificada, proporcionando melhor qualidade e produtividade durante o ciclo de desenvolvimento.

As ADs desenvolvidas com a filosofia de objetos são constituídas por grupos de objetos, autônomos e cooperativos, que podem estar localizados em diversos computadores do ACD. Um **objeto** é uma entidade de *software* composta por alguns dados privados (estado) e por um conjunto de operações associadas (procedimentos), públicas ou privadas, que manipulam os dados. Um objeto pode ser algo tão simples quanto a célula de uma planilha eletrônica ou tão complexo quanto uma aplicação completa. Geralmente o estado do objeto é invisível e protegido de outros objetos. O único meio de examinar ou modificar o estado de um objeto é fazendo a **invocação** de suas operações públicas ou privadas. No modelo de invocação de objetos remotos, os objetos executam serviços, invocando suas próprias operações ou operações de outros objetos.

4.4.1. Relação com o Modelo Cliente-Servidor

O modelo de invocação de objetos remotos tem íntima relação com o modelo C-S. Implicitamente, todas as operações dos objetos são sempre invocadas através de um processo cliente ou servidor (**Fig. 4-9**). Cada objeto possui um servidor associado a si com a função de

gerenciá-lo. Quando é necessário invocar operações de um objeto remoto (invocação remota), o cliente envia um pedido, ou **requisição**, para o servidor responsável pelo objeto. A requisição contém a identificação do objeto a ser invocado e os parâmetros requeridos. Após receber a requisição, o servidor invoca o objeto e executa as operações solicitadas. Finalmente, o servidor retorna o resultado para o cliente. Quando o objeto invocado é local, o cliente assume o papel de gerente do objeto e invoca-o diretamente.

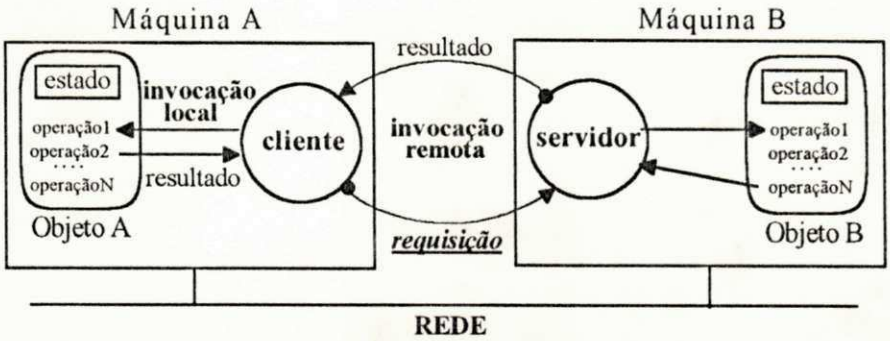


Figura 4-9. *Invocação local e remota de objetos distribuídos*

A nível de aplicação, os passos necessários para a invocação de objetos são processados por um mecanismo de invocação de objetos que torna transparente para o programador a existência do modelo C-S. Do ponto de vista da AD, objetos invocam diretamente as operações de outros objetos.

4.4.2. Arquitetura de um Mecanismo de Objetos Distribuídos

Em ADs, as invocações entre os objetos são feitas através de um Mecanismo de Objetos Distribuídos (**MOD**). O MOD é uma camada de *software*, responsável pelo envio de requisições, execução de operações e pela gerência do ciclo de vida dos objetos.

Normalmente, um MOD é constituído por duas camadas: **gerenciamento de objetos** e **suporte de comunicação** (**Fig. 4-10**). O gerenciamento de objetos oferece serviços para localizar e gerenciar os objetos, enquanto que o suporte de comunicação resolve os problemas de comunicação entre os objetos. Esse modelo é baseado no descrito por [Hewlett91].

De forma simplificada, o MOD funciona de acordo com a **Fig. 4-10** da seguinte forma: 1) o objeto *A* solicita ao MOD a invocação do objeto *B*. 2) O MOD usa o serviço de localização para achar o objeto. 3) Dada a localização do objeto, MOD envia uma requisição para o gerente do objeto a ser invocado por intermédio do suporte de comunicação. 4) o gerente de objetos responsável pelo objeto a ser invocado recebe a requisição, ativa o objeto e executa as operações solicitadas.

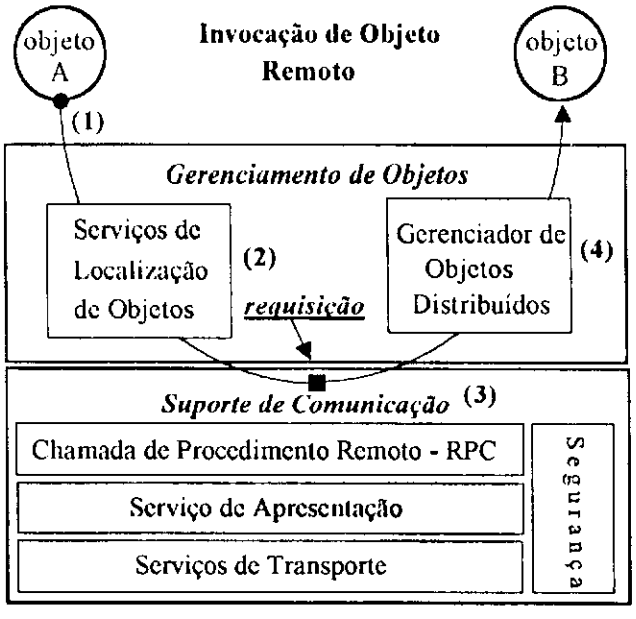


Figura 4-10. *Arquitetura de um Mecanismo de Objetos Distribuídos (MOD)*

Ao final desse processo, o MOD, possivelmente, retorna algum resultado para o objeto requisitante. Do ponto de vista do objeto que faz a invocação, todas as operações realizadas pelo MOD são transparentes, isto é, todos os objetos são sempre invocados como se fossem locais. A seguir, serão descritas, com maiores detalhes, as camadas de suporte de comunicação e gerenciamento de objetos.

4.4.2.1. Suporte de Comunicação

A infra-estrutura necessária para a comunicação entre os objetos é fornecida pela camada de suporte de comunicação do MOD. Os principais objetivos dessa camada de serviço são:

a) permitir que as requisições sejam enviadas entre os objetos para viabilizar invocações; b) realizar a conversão de dados para uma forma canônica, quando os objetos que se comunicam estão em computadores com *hardware* e/ou *software* diferentes; c) tornar o MOD, e consequentemente os objetos da AD, independentes de topologias de rede, protocolos e *software*; d) impedir acesso não autorizado aos objetos. Esses objetivos são atingidos, através da integração do suporte de comunicação do MOD com os serviços da arquitetura do ACD, descritos no capítulo 2. Por exemplo, o serviço de RPC é usado pelo suporte de comunicação para o envio e recebimento das requisições. O suporte de comunicação transforma as requisições de invocação em RPCs. O serviço de apresentação trata os problemas de heterogeneidade dos tipos de dados dos argumentos das invocações. O serviço de transporte isola os detalhes da rede. Finalmente, os serviços de segurança controlam o acesso aos objetos.

4.4.2.2. Gerenciamento de Objetos

O gerenciamento de objetos constitui a camada superior do MOD e é basicamente composto por dois serviços: localização de objetos e gerenciador de objetos distribuídos.

• **Localização de Objetos**: para invocar o objeto, o MOD precisa conhecer a sua localização. Para isso, o MOD envia um pedido para o serviço de localização achar o objeto independentemente do lugar em que ele se encontre no ACD. Essa operação é equivalente ao esquema de *binding*, usado pelo mecanismo RPC, descrito na Seção 4.3.3.3.

Existem diversas estratégias para localizar um objeto. O serviço de localização de objetos pode estar integrado com serviço *naming* do ACD ou pode implementar uma solução específica. Duas possibilidades, sugeridas em [Chin91] são: servidor de nomes de objetos e *cache-broadcast*.

No esquema de **servidor de nomes de objetos**, o sistema cria um ou mais servidores, que podem residir em vários computadores, para auxiliar na localização dos objetos. Esses servidores interagem entre si, para manter atualizadas as informações sobre a localização de todos os objetos existentes. Existem duas variações possíveis para esse esquema. A primeira consiste em manter,

em cada servidor, uma coleção completa de informações de localização suficiente para atender a qualquer pedido de localização, de qualquer objeto. A outra alternativa é manter, em cada servidor, somente parte da informação de localização. Nesse caso, se o pedido de localização não pode ser atendido por um determinado servidor (provavelmente, porque a informação de localização se encontra em outro), então o pedido é redirecionado para outro servidor. O redirecionamento do pedido é feito até que a operação de localização tenha sucesso ou que o serviço de localização conclua que o objeto procurado não existe.

A estratégia de ***cache-broadcast*** consiste em manter, em cada máquina, *caches* com registros da última localização dos objetos mais recentemente referenciados. Quando é necessário localizar um objeto, o serviço de localização inicialmente consulta a *cache* da máquina que deseja localizar o objeto. Se as informações de localização são encontradas e estão atualizadas, então o objeto pode ser invocado sem problemas. Caso a informação não se encontre na *cache* ou esteja desatualizada, uma mensagem de *broadcast*, contendo o pedido de localização, é enviada para todas as máquinas através da rede. Cada máquina, ao receber o pedido de localização, faz uma pesquisa interna para tentar localizar o objeto requerido. A máquina que encontrar o objeto envia uma mensagem com a informação de localização para a máquina que emitiu o *broadcast*, que, por sua vez, atualiza a sua *cache*.

• **Gerenciador de Objetos Distribuídos**: é o responsável pela gerência dos objetos, ativando-os e desativando-os, mantendo indicadores de estado dos objetos para possibilitar a sua utilização [Hewlett91].

A **criação** e **eliminação** de objetos funcionam da seguinte forma: todo objeto criado é registrado em uma lista de controle. Essa lista é mantida pelo gerente de objetos para o controle dos objetos sob sua responsabilidade. Do ponto de vista do gerente, criar um objeto equivale a registrá-lo na lista de controle, bem como eliminá-lo equivale a cancelar seu registro da lista. Do ponto de vista do objeto, a criação envolve, além do registro, a inicialização do seu estado inicial.

A **ativação e desativação** estão relacionadas com racionalização de uso dos recursos. A desativação de objetos é essencialmente para otimizar recursos, visto que um objeto ativo consome mais recursos que um objeto inativo. O gerenciador de objetos desativa e coloca em uma lista os objetos que ficarem inativos durante um tempo determinado. Quando um objeto desativado é invocado, o gerente de objetos volta a ativá-lo. As operações de ativação e desativação são totalmente transparentes, de tal forma que o usuário nunca percebe que existem objetos desativados.

O gerenciador de objetos, dependendo da implementação, permite que o estado de objetos selecionados pelo programador seja automaticamente armazenado em um meio não volátil. Essa facilidade simplifica a programação de ADs tolerantes a falhas, porque torna possível a fácil recuperação do estado de objetos que sofrem término anormal. Maiores detalhes sobre manutenção de estados, no caso de falhas, podem ser encontrados na Seção 5.2.2.5.

4.4.3. Particularidades de Invocações de Objetos

Em geral, o meio usado pelo programador, para invocar objetos, é através de linguagens de programação específicas, como por exemplo, "C" ou C++. Nesse caso, as invocações são feitas como chamadas convencionais de procedimentos. Por exemplo, uma invocação em linguagem "C" poderia ser escrita da seguinte forma:

$$resultado = nome\_da\_operação(objeto\_solicitado, P_1, P_2, ..., P_n),$$

onde P_1, P_2, ... , P_n representam os parâmetros da invocação. O MOD normalmente oferece ao programador várias alternativas, para invocar objetos e passar parâmetros.

• **Tipos de Invocação**: a maioria dos MODs oferecem três tipos básicos de invocação: síncrona, assíncrona e síncrona protelada. Em invocações **síncronas**, o objeto que faz a solicitação fica bloqueado até que a mesma seja completada. O retorno da invocação pode indicar tanto o sucesso do pedido quanto a ocorrência de uma exceção.

Quando a invocação é **assíncrona**, o objeto solicitante tem a vantagem de ter seu controle liberado antes do término da execução da ação. As invocações assíncronas não retornam nenhum resultado para o objeto solicitante, o que obviamente restringe o seu uso para uma determinada classe de aplicações. Consequentemente, o MOD não retorna, para o objeto que fez a requisição, nenhuma informação caso ocorra uma exceção durante a execução das operações. O programador tem que tomar os cuidados necessários, para que a AD considere esse tipo de ocorrência.

Para resolver esse problema, existem as invocações **síncronas proteladas**. Elas devem ser usadas, quando o objeto solicitante não deseja ficar bloqueado durante a execução da invocação, mas deseja obter valores de retorno. O funcionamento é semelhante ao de uma invocação assíncrona: após o envio da requisição, o objeto tem o seu controle liberado para dar continuidade a sua execução. A principal diferença é que, após algum tempo, o objeto solicitante se re-sincroniza com a invocação feita anteriormente, para obter os resultados retornados. A implementação de invocações síncronas proteladas pode ser feita no objeto que faz a solicitação, através da criação de um *thread* adicional que faz uma invocação síncrona.

• **Parâmetros**: os parâmetros das invocações podem ser de três tipos: de entrada, de saída ou de entrada e saída. O tipo do parâmetro determina se os valores estão sendo enviados, retornados ou simultaneamente enviados e retornados. Conceitualmente, os parâmetros são passados por referência. Na prática, as considerações sobre invocações com passagem de parâmetros são semelhantes as de passagem de parâmetro de RPCs discutidas na Seção 4.3.3.1.

Uma vez que o programador tem conhecimento dos modelos de programação de ADs, ele pode dar início ao seu projeto. Contudo, existem alguns pontos específicos que, quando do conhecimento do programador, facilitam o desenvolvimento de ADs eficientes e confiáveis. O próximo capítulo apresenta algumas considerações nesse sentido.

5. Considerações Específicas de Projeto

Neste capítulo, serão apresentadas quatro seções contendo considerações específicas para o projeto de ADs, envolvendo aspectos de serviço transporte, tratamento e recuperação de erros, desempenho e segurança. Na seção de seleção de transporte, serão discutidos critérios para auxiliar o programador na escolha do serviço de transporte das ADs. Na seção de tratamento e recuperação de erros, serão mostradas técnicas para detecção e tratamento de erros em ADs. Na seção de desempenho, serão apresentadas alternativas para maximização do desempenho de ADs. Finalmente, na seção de segurança, serão discutidos os serviços de identificação, autenticação, autorização e auditoria.

As considerações deste capítulo serão baseadas no modelo de programação de RPC, por ser atualmente o modelo mais utilizado para o desenvolvimento de ADs. Apesar disso, as considerações apresentadas também podem ser úteis para outros modelos de programação.

5.1. Considerações de Transporte

Toda RPC necessita fazer uso de um serviço de transporte para viabilizar a comunicação, através da rede, entre os processos clientes e servidores. O objetivo desta seção é apresentar considerações, para auxiliar o programador na definição do serviço de transporte que melhor se adapte aos requisitos da AD a ser desenvolvida. Serão descritos critérios para seleção de serviços de transporte e considerações sobre transparência de transporte.

5.1.1. Critérios para Seleção de Transporte

Basicamente, os serviços de transporte usados pelas RPCs se diferenciam pela existência ou não de uma conexão entre o cliente e o servidor. Os mais utilizados atualmente são os protocolos TCP e UDP. Analisando as difereças entre TCP e UDP, o programador tem condições de identificar o transporte mais adequado para suas necessidades. Os principais critérios que devem ser considerados, na seleção do serviço de transporte, são: confiabilidade, desempenho, semântica de chamada remota, idempotência, capacidade de dados, tratamento de erros e escalabilidade.

5.1.1.1. Confiabilidade

O uso de TCP permite a troca confiável de dados entre clientes e servidores. A existência de uma conexão garante que nenhum pedido de RPC é perdido ou recebido fora de ordem pelo servidor. Com UDP o cliente não tem garantias de que o servidor recebe os pedidos enviados. Por ser um transporte não confiável, pode haver, além da perda, replicação ou troca de ordem dos pedidos. Em função disso, os usuários de UDP devem considerar a possibilidade do tratamento de erros de pedidos de RPC a nível de aplicação.

5.1.1.2. Desempenho

A maior vantagem de UDP sobre TCP é o desempenho. Uma conexão TCP consome tempo e requer manutenção de estado. A existência de estado implica no consumo de recursos do sistema, introduzindo *overhead*. Já UDP é um transporte mais leve que consome recursos somente quando os dados são enviados ou recebidos.

Apesar de UDP apresentar melhor desempenho, há casos em que uma conexão TCP pode ser mais eficiente. Por exemplo, quando a frequência (e o volume) de troca de dados entre o cliente e servidor for extremamente alta. Se a frequência for baixa, então UDP provavelmente é a escolha mais eficiente.

Na Seção 5.3.5, será sugerida uma estratégia para analisar a eficiência do serviço de transporte em função das RPCs realizadas pela AD, facilitando a escolha do transporte mais eficiente.

5.1.1.3. Semântica de Chamada Remota

A semântica da RPC está diretamente relacionada com o tipo de serviço de transporte usado pela RPC. O uso de transportes não confiáveis, como UDP, impõem uma semântica de chamada de **pelo-menos-uma-vez**. O UDP não garante que os pedidos das RPCs não sejam duplicados, podendo levar o procedimento remoto a ser executado mais de uma vez. Os problemas causados pela duplicação de pedidos de RPC serão discutidos na Seção 5.2.2.1.

Com TCP, a RPC tem semântica de **no-máximo-uma-vez**. A conexão TCP garante que, se uma resposta for recebida pelo cliente, o procedimento remoto foi executado **exatamente-uma-vez**. Infelizmente, a semântica de exatamente-uma-vez nem sempre é verdadeira com TCP. No caso de falha do servidor, o cliente pode não receber nenhuma resposta e, portanto, não tem como saber se o procedimento remoto foi executado zero ou uma vez.

5.1.1.4. Idempotência

Os procedimentos que podem ser executados mais de uma vez sem causar transição de estado, no servidor, ou mudança do resultado para o cliente, são tidos como **idempotentes**. Por exemplo, a soma de dois números é um procedimento idempotente, porque, independentemente do número de vezes que a soma seja executada, o resultado é sempre o mesmo: dois mais um é sempre igual a três. Os procedimentos que não apresentam essa característica são considerados **não idempotentes**. Por exemplo, um procedimento para excluir um arquivo é não idempotente. Se após a exclusão do arquivo (realizada pela primeira execução), o procedimento for novamente executado, o resultado será um erro, pois o arquivo já não existe mais.

Quando os procedimentos da AD são idempotentes, isto é, podem ser executados mais de

uma vez sem problemas, o uso de UDP não tem contra indicação. O uso de UDP é desaconselhável nos casos em que os procedimentos são não idempotentes. O motivo é simples: como visto anteriormente, a semântica das RPCs, baseadas em UDP, é de pelo-menos-uma-vez. Isso significa que os procedimentos podem ser executados mais de uma vez, podendo assim haver violação da especificação da AD. Os procedimentos não idempotentes exigem semântica de no-máximo-uma-vez, que pode ser obtida com TCP.

5.1.1.5. Capacidade de Dados

O TCP leva vantagem sobre UDP, quando muitos parâmetros e resultados têm que ser enviados ou recebidos pelas RPCs. Muitas implementações de UDP limitam o tamanho máximo do datagrama em 8k *bytes*. É possível contornar essa limitação, dividindo os dados em partes menores que 8k *bytes* e fazendo chamadas RPCs com cada uma das partes. Essa pode não ser uma boa solução, pois o aumento do número de RPCs pode penalizar o desempenho. Além disso, o programador tem que escrever código adicional para tratar o particionamento dos dados. Normalmente, RPCs que necessitam de parâmetros ou resultados muito grandes devem fazer uso de TCP, que não apresenta a limitação de tamanho imposta pelo datagrama UDP.

5.1.1.6. Tratamento de Erros

Quando ocorre um erro em uma conexão TCP, a AD é forçada a tomar as ações necessárias para reinicializar a conexão, requerendo assim maiores cuidados por parte do programador. O UDP é um protocolo que não mantém nenhum tipo de estado, e portanto não apresenta esse tipo de problemas. Na maioria dos casos, se um servidor de uma AD que usa UDP falha, basta reinicializá-lo para dar continuidade ao processamento sem problemas.

5.1.1.7. Escalabilidade

O uso de TCP implica sempre na abertura de uma conexão entre o cliente e o servidor, o

que consome recursos. Em função disso, o servidor tem o número de conexões limitada pelos recursos do sistema operacional da máquina onde ele é executado. Se um número muito grande de clientes acessarem simultaneamente o servidor, então existe a possibilidade dos recursos gerenciados pelo sistema operacional serem insuficientes para atender a demanda das conexões, causando problemas. O UDP só envia datagramas, logo, teoricamente, não impõe limitações para que o servidor seja acessado por um número ilimitado de clientes.

5.1.2. Transparência de Transporte

Algumas implementações de RPC colocam, à disposição do programador, APIs totalmente independentes de transporte. Nesses casos, o mecanismo RPC faz uso de um transporte padrão pré-definido pelos implementadores da interface. Portanto, é desnecessário que o programador defina explicitamente o tipo de transporte a ser usado pelas RPCs. Do ponto de vista do programador, a grande vantagem dessa abordagem é a redução de complexidade no desenvolvimento da AD, obtida com a transparência dos detalhes de transporte. A maior desvantagem é a perda de flexibilidade, visto que o programador não pode definir o tipo de transporte das RPCs que melhor se adapte aos requisitos do projeto. Isso pode implicar em sérias limitações para determinadas classes de aplicações. O programador deve certificar-se de que o transporte padrão não compromete os requisitos funcionais da AD a ser desenvolvida.

5.2. Tratamento e Recuperação de Erros

As ADs são sensíveis a falhas que ocorrem na rede, nos computadores ou nos processos clientes e servidores. Esses problemas podem dar origem a falhas parciais, cujo tratamento e recuperação inserem complexidade no projeto de ADs.

O objetivo desta seção é fornecer informações para a prevenção, tratamento e recuperação de erros em ADs. Serão abordados os principais tipos de erros, suas causas e implicações. Também serão discutidas algumas soluções gerais para os problemas abordados, visto que

soluções específicas, normalmente, são dependentes dos requisitos da AD.

5.2.1. Erros em Aplicações Distribuídas

Ao desenvolver uma AD, uma questão a ser resolvida pelo programador é o tratamento do término anormal de RPCs que possam violar a especificação da aplicação. Um **término anormal** ocorre, quando o cliente faz uma RPC e não recebe a resposta do servidor, ou quando recebe uma resposta que contém uma indicação de erro. Em ambos os casos, o mecanismo RPC retorna para o cliente um código de erro que permite ao programador tratar o problema. Os códigos de erro retornados pelas RPCs podem ser classificados em dois grupos: erros determináveis e erros indetermináveis.

• **Erro Determinável**: os códigos de erro retornados pelo mecanismo RPC que permitem uma única interpretação, por parte do programador, são considerados determináveis. Quando, após uma RPC, ocorre um erro no servidor, uma mensagem de resposta é retornada para o cliente indicando a falha ocorrida. Se a mensagem de resposta chegar até o cliente, então é possível, para o mecanismo RPC, determinar precisamente que tipo de falha ocorreu no lado do servidor. Alguns exemplos de erros determináveis são chamadas a procedimentos remotos inexistentes ou passagem de parâmetros com tipos incorretos.

• **Erro Indeterminável**: quando o cliente faz uma RPC, e por algum motivo, não recebe a mensagem de resposta do servidor, então o mecanismo RPC não tem como informar para o cliente o que aconteceu no lado do servidor. Para evitar que o cliente fique aguardando indefinidamente uma resposta do servidor, os mecanismos RPCs geralmente adotam uma técnica de ***timeout*** ou equivalente. Se após a RPC, transcorrer um determinado tempo (duração do *timeout*) e o cliente não receber nenhuma resposta do servidor, então o mecanismo RPC assume que houve um término anormal e retorna um código de *timeout* para o cliente. No contexto de RPCs, a ocorrência de *timeouts* caracteriza erros indetermináveis, pois o cliente normalmente nunca sabe com precisão o que causou o *timeout*: o pedido se perdeu? a resposta se perdeu? ou o

servidor falhou?.

5.2.2. Estratégias para Tratamento e Recuperação de Erros

Os erros determináveis não são críticos de serem tratados, porque o código de erro, retornado pelo mecanismo RPC, sempre deixa claro o que aconteceu, facilitando o tratamento do erro. O maior problema é o tratamento de erros indetermináveis. Como definir que ação deve ser tomada, quando o cliente recebe um código de *timeout* sujeito a múltiplas interpretações?. De forma simplificada, as estratégias adotadas pelo cliente, após detectar um *timeout*, podem ser: somente detectar o erro, procurar tratar o erro e recuperar o erro.

• **Somente Detectar o Erro**: ao receber uma indicação de *timeout*, o cliente simplesmente emite uma mensagem de erro para o usuário do tipo "Erro de *timeout*: servidor não responde". O usuário é responsável por tentar executar novamente a operação que falhou.

• **Procurar Tratar o Erro**: após detectar o primeiro *timeout*, o cliente repete a RPC algumas vezes para tentar obter sucesso na sua execução. Se, após as repetições, persistir o *timeout*, o cliente assume que o serviço não está disponível e emite uma mensagem de erro para o usuário.

• **Recuperar o Erro**: ao receber um (ou mais) *timeout* e assumir a indisponibilidade do serviço, o cliente pode adotar uma das seguintes estratégias para recuperar o erro: a) reinicializar o servidor e restaurar o seu estado, caso ele mantenha algum; b) executar o serviço no servidor de outra máquina, caso exista uma réplica.

As estratégias sugeridas acima não descrevem vários detalhes que, na prática, devem ser considerados pelo programador. Por exemplo, o esquema de recuperação de erro deve providenciar, antecipadamente, para que os serviços e recursos utilizados pela AD estejam disponíveis e íntegros, após um término anormal. A capacidade de recuperação do erro é uma das características de ADs tolerantes a falhas.

Para determinar a melhor estratégia de tratamento e recuperação de erros, é fundamental

que o programador tenha em mente os principais tipos de falhas que podem dar origem a erros indetermináveis, bem como suas causas e consequências. A análise dessas possibilidades vai permitir ao programador elaborar medidas preventivas que podem, inclusive, minimizar a ocorrência de *timeout*.

Assumindo que o código do cliente e do servidor estão livres de erros, a maioria dos erros indetermináveis em ADs estão relacionados com: o tipo do transporte usado pelas RPCs, problemas de desempenho, valor de *timeout* subdimensionado, *deadlock*, término anormal do servidor e término anormal do cliente. Cada uma dessas possibilidades tem implicações que exigem soluções específicas e que serão discutidas a seguir.

5.2.2.1. Problemas com Serviço de Transporte

Nesta seção, serão discutidos os problemas que podem ser causados pelo uso de um serviço de transporte não confiável. Finalmente, serão apresentadas soluções para tratar esses problemas.

As ADs usam o serviço de transporte do ACD como suporte para a comunicação entre os clientes e servidores. Assumindo que a rede está operacional e que não houve término anormal do cliente ou do servidor, o tipo de transporte usado pode comprometer as mensagens de pedido e resposta das RPCs. Existem três possibilidades de erros a serem consideradas quando a AD faz uso de um transporte não confiável: perda da mensagem de pedido, perda da mensagem de resposta e reenvio automático de pedido.

Perda de Mensagem de Pedido

Ocorre quando o cliente faz uma RPC, mas a mensagem com o pedido de execução da RPC se perde e não chega até o servidor. Como o servidor não recebe o pedido, o procedimento remoto não é executado, levando o cliente a receber um *timeout*. Nesse caso, se após o *timeout* o cliente repetir com sucesso a RPC, o procedimento remoto é executado apenas uma vez.

Perda de Mensagem de Resposta

Ocorre quando o servidor recebe o pedido, executa o procedimento remoto, mas a mensagem com a resposta se perde antes de chegar ao cliente, originando um *timeout*. Nesse caso, ao contrário da perda do pedido, o procedimento remoto é sempre executado. Se após o *timeout* o cliente repetir a RPC, o procedimento remoto pode ser executado mais de uma vez.

Retransmissão Automática de Pedido

Para minimizar parte do esforço de programação com o tratamento de perda de pedidos e respostas, muitos mecanismos RPC, quando usam um transporte não confiável, reenviam automaticamente o pedido enquanto não recebem a resposta do servidor. Nesse esquema, o programador normalmente define um ***timeout* total** e um ***timeout* de retransmissão**. O *timeout* total é a quantidade de tempo que o mecanismo RPC aguarda por uma resposta do servidor, antes de retornar o *timeout* para o cliente. O *timeout* de retransmissão é o intervalo de tempo que o *stub* cliente aguarda pela resposta do servidor, antes de retransmitir o pedido. A retramissão do pedido é feita automaticamente em ciclos de *timeout* de retransmissão até que o cliente receba a resposta do servidor ou atinja a condição de *timeout* total (**Fig. 5-1**).

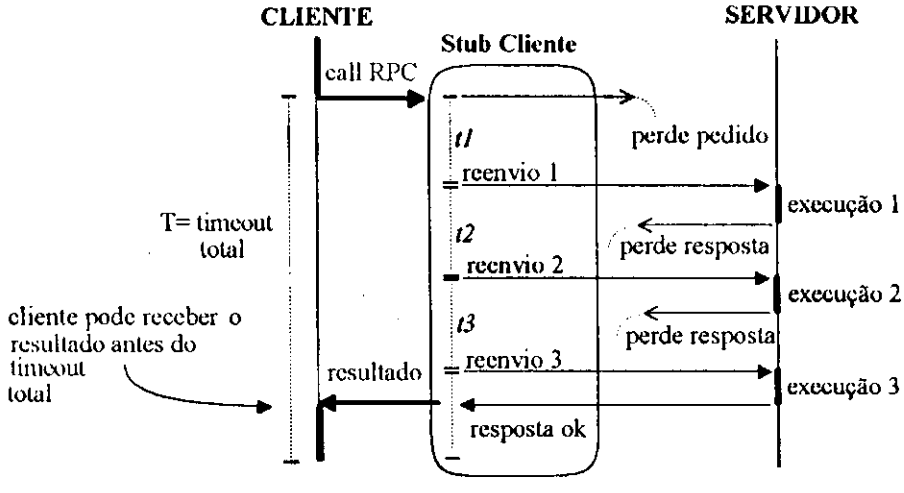


Figura 5-1. *Operações causadas pela retransmissão automática de pedidos em uma RPC*

Os intervalos *t1*, *t2* e *t3* representam os ciclos de *timeouts* de retransmissão que ficam "escondidos" no *stub* cliente, e *T* o *timeout* total. As retransmissões do pedido não são visíveis pelo cliente.

Analisando a **Fig. 5-1**, pode-se observar que as retransmissões podem causar múltiplas execuções do procedimento remoto mesmo sem que o cliente receba a indicação de *timeout* total, ou seja, mesmo que a RPC tenha um término normal. Outra possibilidade é o cliente receber o *timeout* somente após o procedimento remoto ter sido executado várias vezes pelos pedidos retransmitidos. Quando o procedimento remoto é idempotente, a ocorrência de execuções indesejáveis não compromete a integridade da AD. Apesar disso, podem existir implicações, se o processamento do procedimento idempotente for longo e consumir muitos recursos. Então, nesse caso, múltiplas execuções desnecessárias podem criar problemas de desempenho. O problema mais grave a ser resolvido pelo programador são as execuções indesejáveis de procedimentos remotos não indempotentes. Uma solução para tratar esse problema será discutida a seguir.

Múltiplas Execuções e Idempotência

O programador pode evitar execuções indesejáveis de procedimentos remotos a nível de aplicação, adotando um esquema de **cache de resposta** no servidor. Nesse esquema, cada RPC é acrescida de um parâmetro representando um identificador único. Esse identificador é usado para evitar que as RPCs sejam executadas mais de uma vez ou que haja troca na ordem de execução. A implementação pode ser feita da seguinte forma: no cliente acrecenta-se o identificador único a todo pedido de serviço submetido ao servidor. No servidor, o programador deve criar a *cache* de resposta para armazenar informações sobre todos os procedimento remotos executados. A *cache* deve ter basicamente dois campos: identificador do pedido e o resultado da execução do pedido. Todos os procedimentos remotos que necessitarem tratar execuções indesejáveis devem ser escritos de tal forma que sempre consultem a *cache* de resposta, antes de executarem o serviço requerido pelo cliente. Assumindo que o processo servidor tem um único *thread*, o algoritmo para

trabalhar com a *cache* poderia ser:

SE (identificador do pedido não existe na cache) *ENTÃO*

- executar o procedimento remoto;
- armazenar o identificador do pedido e o resultado da execução na *cache*;
- retornar o resultado da execução para o cliente;

SENÃO

- ler o resultado da execução anteriormente armazenado na *cache*;
- retornar o resultado lido da *cache* para o cliente;

A especificação da *cache* de resposta possui vários requisitos, dependentes da aplicação, que devem ser observados pelo programador, tais como: dimensionamento do tamanho da *cache*, definição do tipo da estrutura de dados (estática ou dinâmica) da *cache*, restauração da *cache* em caso de término anormal. Maiores detalhes sobre dimensionamento de *cache* de resposta podem ser encontrados em [Corbin91].

5.2.2.2. Problemas Relacionados com Desempenho

Alguns erros indetermináveis são causados por problemas de desempenho, tais como: congestionamento na rede, sobrecarga na máquina onde o servidor é executado ou tipo inadequado de servidor. O congestionamento da rede pode causar o atraso excessivo dos pedidos e respostas das RPCs, levando o cliente a receber um *timeout*. Outro problema é quando a máquina do servidor está sobrecarregada. Nesse caso, a execução do procedimento remoto pode ficar lenta, a ponto do cliente receber um *timeout* antes do término da execução. Finalmente, a última consideração é quanto ao tipo do servidor. Por exemplo, se o servidor é iterativo e muitos clientes enviam simultaneamente vários pedidos para ele, então pode haver demora no atendimento dos pedidos enfileirados, retardando as respostas e causando um *timeout*.

5.2.2.3. Erros Causados por Timeout Subdimensionado

Em geral, os mecanismos RPCs permitem que o programador especifique o intervalo de tempo do *timeout*. Se o programador definir um valor de *timeout* muito pequeno, o cliente pode receber uma indicação prematura de erro antes do servidor concluir a execução do procedimento remoto. Para definir corretamente o valor do *timeout*, o programador precisa considerar algumas particularidades do cliente e do servidor. Os servidores provavelmente processam os serviços em intervalos de tempo variáveis, dependendo de fatores como a carga do servidor, roteamento ou congestionamento da rede. O cliente deve ser projetado, para assimilar a variação de tempo dos serviços, a fim de não congestionar a rede reenviando continuamente pedidos cujas respostas nunca chegam. Preferivelmente, o cliente deve adotar alguma estratégia tipo ***back off*** que consiste em ir aumentando exponencialmente o seu valor de *timeout*, à medida que as respostas do servidor não chegam [Corbin91]. Essa estratégia representa um esquema de *timeout* adaptativo e pode ser implementada multiplicando-se o valor do *timeout* inicial por 2. Se o cliente receber novamente um *timeout*, um novo valor é recalculado multiplicando-se o valor do último *timeout* por 4 e assim sucessivamente. Em [Stevens90], estão descritas algumas estratégias e algoritmos para definir valores de *timeout* adaptativos.

5.2.2.4. Deadlock Distribuído

Um conjunto de processos da AD está em *deadlock*, quando cada processo do conjunto está aguardando por um evento que somente outro processo, pertencente ao mesmo conjunto, pode causar. Como todos os processos do conjunto estão aguardando, nenhum deles pode causar evento algum, consequentemente, todos os membros do conjunto ficam retidos indefinidamente.

Muitas pesquisas têm sido realizadas para resolver o problema de *deadlock* distribuído. Algumas alternativas para o tratamento de *deadlock* são: ignorar, impedir, detectar e limitar.

- **Ignorar**: consiste em simplesmente em ignorar a probabilidade da ocorrência de *deadlock*. Nesse caso, não existe nenhum esquema para tratar a ocorrência de *deadlock* a nível de

aplicação.

• **Detectar**: esse esquema analisa a ordem das RPCs para identificar ciclos, a fim de detectar a ocorrência do *deadlock* e eliminá-lo.

• **Impedir**: essa estratégia consiste em tornar a ocorrência do *deadlock* estruturalmente impossível. O *deadlock* pode ser prevenido fazendo-se as RPCs em uma ordem pré-definida. Uma forma de se fazer isso é numerando os recursos usados pela AD, e exigindo que os processos ganhem acesso a eles em uma ordem estritamente crescente. Nesse esquema, um processo nunca pode manter para si um recurso de ordem mais alta e pedir um recurso de ordem mais baixa, eliminado assim a possibilidade de ciclos.

• **Limitar**: consiste em impor um limite de tempo, ou *timeout*, para aguardar a liberação dos recursos ou resposta ao pedido. Se o total de tempo é excedido, então assume-se que existe uma condição de *deadlock*. Os maiores problemas dessa técnica são: a) em sistemas sobrecarregados, o número de RPCs que retornam *timeout* tende a aumentar; b) RPCs que têm tempo de processamento muito longo podem ser penalizados antes do término da execução.

Informações detalhadas sobre técnicas para tratamento de *deadlock* distribuído podem ser encontradas em [Spector89] e [Coulouris88].

5.2.2.5. Término Anormal do Servidor

Quando um servidor sofre um término anormal, é necessário recriá-lo a fim de tornar seus serviços novamente disponíveis para os seus clientes. Os pontos básicos relacionados com o término anormal do servidor, que devem ser considerados pelo programador, são: reinicialização, recuperação de estado, recuperação com uso de replicação e cuidados com idempotência.

• **Reinicialização do Servidor**: existem várias abordagens, para resolver o problema de reinicialização de um servidor. Uma alternativa é a reinicialização ser de responsabilidade do cliente - ao acessar um servidor sem sucesso e assumir que ele falhou, o cliente pode recriá-lo na mesma máquina em que ocorreu a falha ou em outra, para torná-lo novamente disponível.

Existem situações em que a reinicialização do servidor não é feita pelo cliente. O programador pode fazer uso de um servidor dedicado, ou **servidor de *boot***, para recriar os servidores que falharam [Bal90]. O servidor de *boot* possui um cadastro com a identificação de todos os servidores que devem ser monitorados e reinicializados em caso de falhas. Todo servidor cadastrado no servidor de *boot*, tem o seu ciclo de vida checado periodicamente. Em determinados intervalos de tempo, o servidor de ***boot*** verifica se os servidores sob sua responsabilidade estão operacionais. Assim que detecta a indisponibilidade de algum servidor cadastrado, o servidor de *boot* providencia imediatamente sua reinicialização. Quando o servidor de *boot* não consegue reinicializar um servidor, porque a máquina onde ele era executado antes da falha não está operacional, ele tenta inicializá-lo em alguma outra máquina.

• **Recuperação de Estado**: após reinicializar um servidor que falhou é necessário restaurar o seu estado, caso ele mantenha algum. As ações a serem tomadas para restauração de servidores que mantêm estado são fundamentais em ADs com requisitos de tolerância a falhas. A recuperação só é possível, se toda informação de estado for previamente armazenada, durante o ciclo de vida do servidor, em um meio não volátil (ex.: disco) que esteja sempre disponível. Dessa forma, durante o processo de reinicialização, o servidor pode restaurar o seu estado, lendo a informação previamente salva, voltando a atender os clientes de forma integra.

• **Recuperação com Replicação**: uma estratégia para tratar falhas de servidores é usar replicação para aumentar a disponibilidade. Em caso de falha, o cliente faz uso de uma réplica do servidor previamente definida. Ao detectar a indisponibilidade do servidor, o cliente simplesmente redireciona as RPCs para a réplica do servidor. Para localizar a réplica requerida, o cliente pode fazer uso do serviço *naming* do ACD.

O uso de replicação requer cuidados adicionais com relação à manutenção de estado. Quando existe replicação de servidores com estado, é necessário manter o estado das réplicas atualizados de tal forma que, quando ocorrer uma falha no servidor principal, os clientes possam usar os servidores replicados sem problemas. Essa abordagem pode ser complexa de implementar,

porque as réplicas, obrigatoriamente, devem ter seus estados atualizados a cada transição de estado do servidor principal. Uma forma de se fazer isso é usando um serviço de *broadcast*. Toda vez que o estado do servidor principal sofrer alguma mudança, é emitida uma mensagem de *broadcast* com as informações necessárias para as réplicas atualizarem os seus estados. Para evitar problemas de inconsitência, causados por perda de mensagens, o serviço de *broadcast* deve ser confiável. Informações sobre serviços de *broadcast* confiáveis podem ser encontradas em [Joseph89].

• **Cuidados com Idempotência**: o tratamento do término anormal do servidor requer alguns cuidados adicionais, quando ele possui procedimentos não idempotentes. Isso é necessário, porque o servidor pode falhar logo após concluir a execução de um procedimento não idempotente e antes de ter tido tempo de enviar uma resposta para o cliente. Nesse caso, se o servidor for reinicializado e o cliente repetir a RPC, o procedimento remoto não idempotente pode ser executado indevidamente mais de uma vez. Uma solução para resolver esse problema é usar o esquema já descrito de *cache* de resposta no servidor, mantida em um meio não volátil, para evitar que os registros dos pedidos já executados não se percam após a falha.

5.2.2.6. Término Anormal do Cliente

O término anormal do cliente tem pelo menos três aspectos que devem ser observados pelo programador: problemas com órfão, reinicialização do cliente e recuperação de estado (caso o cliente mantenha algum).

• **Problemas com Órfão**: uma questão importante a ser considerada no projeto de ADs é o que acontece quando o cliente faz uma RPC e sofre um término anormal, antes de receber o resultado do servidor. Supondo que o pedido de RPC foi recebido pelo servidor, o procedimento remoto é executado sem que ninguém mais esteja aguardando o seu resultado. Esse processamento desnecessário é conhecido como **órfão**, porque o cliente responsável pelo pedido, morre antes de receber a resposta do servidor. Os órfãos podem causar vários problemas, como

por exemplo, o consumo desnecessário de ciclos de CPU, travamento indesejável de arquivos ou mesmo *deadlock*.

Em situações em que o mecanismo de RPC não oferece tratamento de órfão, o programador tem que resolver esse problema a nível de aplicação. Três sugestões são dadas por [Tanenbaum92] para tratar órfãos: exterminação, reencarnação e expiração.

◦ **Exterminação**: nesse método, o cliente armazena em um *log* informações sobre as RPCs efetuadas. O *log* deve residir em um meio não volátil para sobreviver à falha. Ao ser reinicializado após a falha, o cliente acessa o *log*, para obter as informações que permitam localizar e eliminar explicitamente os órfãos. A desvantagem dessa solução é o custo de leitura e escrita em disco do *log* a cada RPC. Para manter o *log* consistente, o cliente deve, toda vez que receber uma resposta do servidor, excluir do *log* o registro correspondente ao procedimento remoto executado. Outro problema dessa solução é que, antes de ser exterminado, um órfão pode fazer novas RPCs, dando origem a novos órfãos impossíveis de serem localizados através do *log* do cliente.

◦ **Reencarnação**: consiste em dividir o tempo de processamento das RPCs em **épocas** numeradas sequencialmente. Ao ser reinicializado, o cliente emite uma mensagem de *broadcast* para todos os servidores, informando que sofreu uma falha e que vai ter início uma nova época. Ao receberem a declaração de início de época, os servidores tentam localizar os clientes responsáveis pelas RPCs em andamento. Se os clientes responsáveis não forem localizados, então o processamento associado a eles é considerado órfão e, em seguida, eliminados pelos servidores. Naturalmente, se uma parte da rede cair, alguns órfãos podem sobreviver. Entretanto, quando a rede retornar ao normal, suas respostas conterão números de épocas obsoletas, tornando fácil sua detecção e eliminação pelo cliente. Uma vantagem dessa abordagem é que não é necessário armazenar nenhuma informação em um meio não volátil.

◦ **Expiração**: o método de expiração consiste em determinar uma quantidade de **tempo padrão de execução** (TPE), para o servidor executar o procedimento remoto. Se o TPE for insuficiente para o término da execução, o servidor faz um pedido ao cliente de uma nova quota

de tempo. Nesse contexto, se o cliente falhar e não for reinicializado antes do valor do TPE ser atingido, o servidor assume que está processando um órfão é o elimina. Um problema a ser resolvido é a escolha de um valor razoável para o TPE. Isso não é trivial, porque depende das características de processamento dos procedimentos remotos, que, naturalmente, têm requisitos diferentes.

A simples eliminação de órfãos não resolve todos os problemas. O programador deve considerar os efeitos colaterais que uma eliminação sem critério pode causar. Por exemplo, se um órfão que mantém um arquivo travado for morto sem nenhum critério, o travamento pode se manter e o arquivo ficar bloqueado indefinidamente. Maiores detalhes sobre tratamento de órfãos podem ser encontrados em [Panzieri88] e [Ravindran89].

• **Reinicialização do Cliente**: um meio de reinicializar de forma transparente um cliente que falhou é usando o esquema de servidor de *boot* já descrito anteriormente. Muitos ACDs oferecem serviços para recuperação de processos que sofrem término anormal. Caso o serviço existente não atenda satisfatoriamente os requisitos da AD, então o programador deve implementar o seu próprio servidor de *boot*.

• **Recuperação de Estado**: a recuperação de estado do cliente pode ser feita de forma semelhante à recuperação de estado do servidor. A diferença é que, além do estado poder ser salvo em um disco acessível pelo cliente, ele também pode ser salvo em um servidor. Se o estado está em um disco acessível pelo cliente, então a recuperação consiste em carregar essa informação de estado durante o processo de reincialização, para que o processamento possa ser retomado a partir do ponto em que foi interrompido. Por outro lado, se o estado for salvo no servidor, o cliente tem que, durante a sua reinicialização, fazer RPCs ao servidor a fim de obter as informações necessárias para a restauração do estado.

5.2.2.7. Transparência de Falhas

Grande parte do esforço de programação de ADs é gasto com o tratamento e recuperação

de erros. Para liberar o programador desses detalhes, alguns ACDs oferecem até três níveis de facilidades para prover transparência de falhas: facilidades a nível do sistema operacional, facilidades a nível de linguagem de programação e facilidades oferecidas por um serviço de transações distribuídas. Essas facilidades, usadas individualmente ou em conjunto, simplificam o desenvolvimento de AD tolerantes a falhas.

• **Facilidades a Nível de Sistema Operacional**: os sistemas operacionais distribuídos (SODs) se diferenciam, principalmente, pela habilidade de tratar falhas parciais. Em um extremo, estão SODs que não possuem nenhuma facilidade especial para tratar falhas e, no outro, estão os SODs especificamente projetados para suportar ADs tolerantes a falhas. Alguns exemplos são os SODs *Amoeba*, *Eden*, *Clouds* e o *system V* [Bal90]. O sistema *Amoeba* usa um servidor de *boot*, para aumentar a disponibilidade de serviços estratégicos. Os sistemas *Eden* e *Clouds* oferece facilidades que permitem salvar o estado interno dos processos, simplificando a recuperação de estado. O *system V* permite que o programador defina procedimentos para tratar falhas através de um **servidor de exceção**. Por exemplo, se ocorre uma exceção tipo violação de endereço, o núcleo do sistema operacional do *system V* envia uma mensagem para o servidor de exceção. O programador instrui o servidor de exceção sobre como ele deve proceder em cada caso previsto de falha.

• **Facilidades a Nível de Linguagem**: idealmente, uma linguagem deve tornar transparente todo o tratamento de falhas para o programador. Existem várias linguagens para o desenvolvimento de ADs que permitem ao programador fazer uma especificação de alto nível, para determinar que processos e dados devem ser recuperados em caso de falha. Alguns exemplos de linguagens que usam essa abordagem são *Argus* e *Aeolus*. Em [Bal90], estão descritas essas e várias outras linguagens.

• **Serviço de Transações Distribuídas**: o serviço de transações distribuídas, introduzido na Seção 2.2.9, normalmente é parte integrante do conjunto de serviços do ACD. Através desse serviço, as operações da AD podem ser agrupadas em transações atômicas. Do ponto de vista do

programador, a disponibilidade de uma semântica tipo "tudo ou nada" simplifica a implementação de esquemas de tratamento e recuperação de erros, facilitando o desenvolvimento de ADs tolerantes a falhas.

5.3. Considerações de Desempenho

Nesta seção, serão inicialmente sugeridas alternativas para maximizar o desempenho de ADs, tais como: *cache* no cliente e no servidor, paralelismo, replicação de serviços, manutenção de estado no cliente e avaliação da eficiência dos serviços de transporte. Finalmente, serão discutidos alguns fatores, diretamente ligados à implementação, que podem contribuir para a degradação de desempenho das ADs, como por exemplo, o uso sem critérios de serviços de *broadcast*, valor de *timeout* superestimado e volume de dados dos parâmetros das RPCs.

5.3.1. Uso de Cache no Cliente e no Servidor

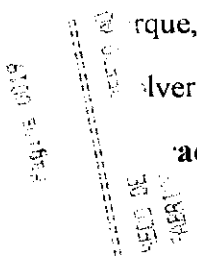
Uma forma de maximizar o desempenho das ADs é reduzir o número das RPCs, efetuadas pelos processsos da aplicação. Isso é importante por dois motivos: a) reduz o tráfego de mensagens de RPCs na rede, diminuindo a possibilidade de congestionamentos; b) reduz a carga de trabalho dos servidores, tornando-os mais eficientes.

Para reduzir o número de RPCs, o programador pode explorar o conceito de localidade de referência, através do uso de *cache* de pedido no cliente e *cache* de resposta no servidor.

5.3.1.1. Cache de Pedido no Cliente

A *cache* de pedido no cliente é usada para manter os resultados das RPCs mais recentemente realizadas pelo cliente. Antes de fazer qualquer RPC, o cliente deve consultar a sua *cache* de pedidos para tentar obter os resultados desejados, evitando RPCs desnecessárias. O grande problema desse esquema é como manter as informações da *cache* atualizadas. Como o cliente pode ter certeza de que a informação da *cache* está íntegra? Por exemplo, se um

ioto lê dados de um arquivo compartilhado que é atualizado com frequência por
. então existe a possibilidade de alguma informação armazenada na *cache* ficar
rque, nesse caso, a *cache* simplesmente mantém uma cópia local dos dados
lver problemas de manutenção da integridade da *cache* são usados **protocolos**
ache. Alguns desses protocolos são descritos por Satianarayanan em



. Cache de Resposta no Servidor

A estratégia de *cache* de resposta no servidor, já descrita na Seção 5.2.2.1, normalmente é usada para evitar execuções indesejáveis de procedimentos não idempotentes. Por outro lado, o programador pode melhorar o desempenho da AD, adaptando a estratégia de *cache* de resposta também para procedimentos idempotentes. O objetivo da *cache* de resposta, nesse caso, é evitar a execução desnecessária de procedimentos que foram recentemente executados, tornando os servidores mais eficientes. Através dessa técnica é possível minimizar o consumo desnecessário de recursos por parte do servidor, como por exemplo, de ciclos de CPU ou leitura em disco, reduzindo o tempo de retorno do resultado das RPCs para o cliente. A *cache* de resposta do servidor, adaptada para fins de melhoria de desempenho, tem os mesmos problemas de consistência, discutidos na seção anterior.

5.3.2. Paralelismo

Uma das principais vantagens oferecidas pelas ADs é a possibilidade de explorar paralelismo para obtenção de aplicações com alto grau de desempenho, mesmo sem a disponibilidade de *hardware* multiprocessador. O uso de RPCs assíncronas, para executar paralelamente procedimentos da AD em diferentes computadores do ACD, tornam algumas aplicações extremamente rápidas.

O uso de paralelismo depende dos requisitos e características particulares de cada

aplicação. Um exemplo de aplicação onde se pode obter alto desempenho com o uso de paralelismo é o de processamento remoto de imagens, citado na Seção 1.1.1.1. Outro exemplo, descrito em [Bal90], é um projeto cujo objetivo era calcular os fatores primos de números de 100 dígitos. O problema foi resolvido com o uso de paralelismo, através de um sistema com 400 computadores, conectados em rede, distribuídos por institutos de pesquisas de três continentes.

Para melhorar o desempenho de ADs, o programador dispõe de dois tipos de paralelismo: paralelismo virtual e paralelismo real. O **paralelismo virtual** é caracterizado por processos executados logicamente em paralelo, em uma máquina que possui um único processador. Nesse caso, o sistema operacional (ex.: UNIX) faz o escalonamento dos processos para permitir o compartilhamento do processador. Alguns autores empregam o termo concorrência como sinônimo de paralelismo virtual. O **paralelismo real** consiste em executar os processos da AD, simultaneamente, em diferentes máquinas fisicamente distribuídas pela rede, ou em máquinas com *hardware* multiprocessador. Dependendo do tipo da linguagem de programação, usada para desenvolver a AD, as diferenças entre o paralelismo real e o virtual podem ser visíveis ou não. Em linguagens em que essa diferença é visível, o programador é responsável por definir, explicitamente, o paralelismo. Isso torna a programação mais complexa, mas, por outro lado, provê maior flexibilidade.

5.3.2.1. Estratégia para Utilizar Paralelismo

Para melhorar o desempenho com paralelismo, o programador pode adotar a seguinte estratégia: a) identificar e dividir os serviços da AD que podem ser executados em paralelo[1]; b) distribuir esses serviços pelas várias máquinas que compõem o ACD, para viabilizar paralelismo real; c) projetar o cliente, para chamar os serviços remotos da aplicação, usando o serviço de *threads*; d) projetar servidores concorrentes com múltiplos *threads*, para aumentar a eficiência de atendimento aos clientes. Assumindo que a máquina onde roda o servidor possui

[1] A unidade de paralelismo pode ser um processo, uma instrução de programa, uma expressão ou uma cláusula (em linguagens lógicas). A abordagem adotada neste trabalho será em torno de processo, que intuitivamente é a mais natural.

apenas um processador, os servidores concorrentes com *threads*, exemplificam o uso de paralelismo virtual para melhorar o desempenho. Obviamente, se a máquina do servidor possuir *hardware* multiprocessador, os serviços serão executados com paralelismo real.

5.3.2.2. Cuidados com Granularidade do Paralelismo

O uso de paralelismo introduz alguns problemas que devem serem considerados pelo programador. Por exemplo, muitos processos de uma AD, executados paralelamente em diferentes máquinas, podem ter necessidade de se comunicar com muita frequência. Isso pode significar um aumento considerável do número de mensagens na rede, gerando problemas de desempenho. Portanto, é importante considerar e controlar o grau de **granularidade** do paralelismo no projeto da AD. A granularidade é definida como o tempo total de processamento entre as RPCs necessárias para a comunicação entre os processos da AD. A granularidade pode ser classificada como: alta, média e fina. Os processos de ADs com granularidade alta de paralelismo, gastam a maior parte do seu tempo com processamento e raramente se comunicam. Já ADs com granularidade fina de paralelismo, possuem processos que se comunicam mais frequentemente. Finalmente, a granularidade média de paralelismo é um meio termo entre ambas (alta e fina). Trabalhos introdutórios sobre esse assunto podem ser encontrados em [Athas88] e [Ranka88] citados em [Bal90].

5.3.3. Replicação de Serviços

Existem situações em que a demanda de determinados serviços tende a crescer em escala exponencial, causando problemas de desempenho. A replicação de servidores (e recursos) é um meio de aumentar a disponibilidade dos serviços e de distribuir a carga de trabalho da AD. A disponibilidade de réplicas de servidores, em diferentes máquinas da rede, torna possível a múltiplos clientes executarem determinados serviços com paralelismo real, o que, naturalmente, resulta em maior eficiência.

Outra alternativa para melhorar o desempenho é usar replicação para evitar RPCs realizadas pelo cliente. Isso pode ser feito, colocando-se réplicas dos serviços remotos mais solicitados pelo cliente, na mesma máquina onde o cliente é executado. Dessa forma, um número significativo de chamadas de procedimentos, anteriormente remotas, passam a ser feitas localmente pelo cliente, reduzindo tráfego de comunicação da rede.

O problema de usar réplicas é que, em algumas situações, elas devem ser atualizadas (ex.: réplicas de servidores com estado). Algumas técnicas, para manter réplicas de servidores atualizadas, foram introduzidas na Seção 5.2.2.5. Além disso, o programador deve procurar atender ao requisito de transparência de replicação, o que tende a aumentar a complexidade do projeto. Maiores detalhes sobre técnicas de replicação em ADs podem ser encontrados em [Coulouris88], [Mullender89] e [Tanenbaum92].

5.3.4. Manutenção de Estado no Cliente

Os servidores são responsáveis pela maior parte do processamento realizado pelas ADs. Portanto, uma forma de melhorar o desempenho é reduzir a carga de trabalho dos servidores, para aumentar sua eficiência. Um caso particular é o dos servidores com estado. Supondo que ***n*** clientes solicitam simultaneamente serviços que requerem manutenção de estado, por parte do servidor, então o servidor tem que gerenciar e manter ***n*** informações de estado, o que, dependendo da taxa de crescimento do número de clientes, pode representar uma carga de trabalho considerável. Transferindo para cada cliente a responsabilidade pela manutenção das informações de estado, é possível aliviar parte da carga de trabalho do servidor, tornando-o mais ágil. Exemplos práticos dessa abordagem são algumas implementações de sistemas de arquivos distribuídos, como o *Network File System* da Sun (NFS). Quando o cliente acessa pela primeira vez um arquivo, o servidor NFS retorna para ele uma estrutura de dados com informações do arquivo (estado), conhecida como *file handle*. O cliente sempre envia, para o servidor, o *file handle* em qualquer operação subsequente que acesse o referido arquivo. O servidor fica livre da

responsabilidade de manter informações de estado dos arquivos acessados pelos clientes.

5.3.5. Avaliação do Desempenho do Serviço de Transporte

O serviço de transporte é a base de toda a comunicação entre os processos remotos e, consequentemente, um fator que tem impacto direto no desempenho das ADs. Nesta seção, será apresentada uma estratégia simples, para avaliar a eficiência do serviço de transporte em função das RPCs realizadas pela AD.

O desempenho do serviço de transporte pode variar significativamente dependendo da sua implementação. Em função disso, o programador deve definir estratégias, para avaliar o impacto que o serviço de transporte tem sobre o desempenho da AD. Um meio de fazer isso é escrever pequenos programas de teste com as principais RPCs executadas pela AD. Os programas de teste devem utilizar cada uma das opções de transporte disponíveis. Dessa forma, é possível avaliar, individualmente, o desempenho de cada transporte em função dos requisitos das RPCs. Para obter os dados de desempenho dos programas de testes, o programador pode fazer uso de ferramentas que traçam o perfil de execução dos módulos da aplicação. Existem diversas ferramentas que podem ser usadas para esse fim, como por exemplo, os utilitários ***time*** e ***prof*** disponíveis em sistemas UNIX. Dessa forma, o programador pode caracterizar a relação entre cada tipo de transporte e o desempenho da AD.

5.3.6. Chamada de Procedimento Remoto em Lote

Existem situações em que o cliente necessita fazer várias RPCs, ou **grupo** de RPCs, mas não precisa obter uma resposta enquanto o servidor não receber e processar todo o grupo. Nesses casos, é possível aumentar a velocidade de processamento das RPCs como um todo. Para isso, o programador pode fazer uso de uma facilidade, normalmente oferecida pelo serviço de RPC, conhecida como **RPC *Batch***.

A facilidade de RPC *Batch* permite ao programador enfileirar no lado do cliente um grupo

de RPCs, e então enviá-las, através da rede, em um único lote para o servidor. Em muitos casos, o uso de RPC *Batch* pode melhorar o desempenho, como por exemplo, na cópia remota de arquivos. Supondo que são necessárias cinco RPCs, para copiar um arquivo do cliente para a máquina do servidor, o cliente sem usar RPC *Batch* teria que enviar cinco pedidos através da rede para realizar a cópia. Com RPC *Batch*, provavelmente, apenas um pedido contendo o lote das cinco RPCs enfileiradas seria enviado através da rede, reduzindo o tráfego de comunicação. Nesse exemplo, por coincidência, as RPCs do lote executam o mesmo serviço (cópia), mas normalmente um lote de RPC *Batch* envolve a execução de serviços distintos.

Uma observação importante é que com RPC *Batch* somente a última RPC do lote retorna um resultado para o cliente. Portanto, o uso de RPC *Batch* impõe o uso de um transporte confiável como TCP. Em [Lyon84b], [Sun88b], [Bloomer91] e [Corbin91], podem ser encontrados maiores detalhes sobre essa facilidade.

5.3.7. Cuidados com o Uso de Broadcast

Muitas implementações de mecanismos RPCs oferecem ao programador a possibilidade de propagar uma simples RPC, através do serviço de *broadcast* da rede, para todos os servidores de um determinado domínio. Essa facilidade é conhecida como **RPC *Broadcast*** e normalmente usa um transporte não confiável tipo UDP.

O RPC *broadcast* deve ser utilizado de forma criteriosa, para não interferir no fluxo normal dos dados transmitidos através da rede, causando problemas de desempenho. O envio excessivo de mensagens de *broadcast*, é um ato anti-social que prejudica toda a comunidade de usuários. Por exemplo, estações de trabalho sem disco são penalizadas pelo mal uso do *broadcast* porque seus dados, obrigatoriamente mantidos em discos remotos, compartilham e concorrem na rede com as mensagens de *broadcast*.

Para minimizar esses problemas, muitas pilhas de protocolos usadas pelos ACDs impõem limitações para a propagação de mensagens de *broadcast*. Por exemplo, em ambientes TCP/IP,

broadcasts IP nunca são transmitidos através das pontes (*gateways*) e, geralmente, estão limitados a um domínio específico, como por exemplo, a uma rede local *Ethernet* [Malamud92b].

O programador deve sempre usar o RPC *broadcast* em conjunto com alguma política de moderação. Um exemplo dessa abordagem é o programa de *binding ypbind* do *Network Information Service* (NIS)[2] descrito em [Corbin92]. O *ypbind* torna possível para os clientes consultarem o NIS. Quando *ypbind* é inicializado, um *broadcast* é enviado para todos os servidores. Somente os servidores NIS que pertencem ao domínio gerenciado pelo *ypbind* respondem ao pedido, reduzindo o número de respostas ao *broadcast*. Em seguida, o *ypbind* efetua o *binding* com um dos servidores que respondeu ao *broadcast*. Todos os pedidos de consulta dos clientes são direcionados para esse servidor. Se, no futuro, o servidor não tiver condições de atender ao pedido de consulta do cliente, o *ypbind* inicia novamente o processo de *binding* emitindo novo *broadcast*. É importante observar que o *ypbind* não emite um *broadcast* a cada pedido feito pelo cliente, mas somente para fazer o *binding* com o servidor NIS.

Do ponto de vista de desempenho, uma alternativa superior ao serviço de RPC *broadcast* é um serviço de **RPC *multicast***. Nesse caso, a RPC é transmitida somente para um grupo específico de máquinas da rede, o que obviamente reduz o número de respostas. Infelizmente, nem todas as plataformas para desenvolvimento de ADs oferecem essa facilidade.

5.3.8. Valor de Timeout Superestimado

Do ponto de vista de desempenho, o programador deve tomar cuidados para não superestimar o valor do *timeout* das RPCs. Isso pode ter impacto negativo no desempenho da AD, principalmente, se ela possuir requisitos de tolerância a falhas. Um valor de *timeout* superdimensionado leva, em muitos casos, o cliente a aguardar excessivamente por uma indicação de *timeout*. Um exemplo prático é o de um cliente que adota um *timeout* de 120 segundos quando o valor ideal seria de 30 segundos. Se o cliente faz uma RPC síncrona que sofre um término

[2] O *Network Information Service* (NIS) é o serviço de *naming* da plataforma *Open Network Computing* da Sun. O NIS é também conhecido como *Yellow Pages*.

anormal, então, ele aguardará sem necessidade, pelo menos 90 segundos, antes de receber o *timeout*, para só então iniciar alguma ação de recuperação da falha. Na Seção 5.2.2.3, foi sugerida uma estratégia para especificar valores de *timeout*.

5.3.9. Volume de Parâmetros de Chamadas Remotas

Outra alternativa para maximizar o desempenho das ADs é reduzir o volume (tamanho e quantidade) dos parâmetros das RPCs. Em casos onde é inevitável o envio de um grande volume de dados, uma alternativa é usar técnicas de compressão de dados antes dos parâmetros serem enviados. Outra possibilidade é procurar fazer otimizações a nível do serviço de apresentação. Em alguns serviços de apresentação é possível declarar as estruturas de dados, de tal forma que o número de *bytes* necessários para representação dos parâmetros das RPCs, na forma canônica, sejam minimizados. Em [Corbin91], é descrito um exemplo em que uma simples mudança na forma de declarar uma determinada estrutura de dados reduziu, em 20 por cento, o número de *bytes* dos parâmetros a serem transmitidos através da rede.

5.4. Considerações de Segurança

Nesta seção, serão apresentados os serviços de segurança normalmente disponíveis nos ACDs. Inicialmente, serão discutidas técnicas de criptografia de dados, responsáveis pelo suporte básico dos serviços de segurança do ACD. Finalmente, serão descritos os serviços de segurança compostos de: canal seguro de comunicação, identificação, autenticação, autorização e auditoria.

5.4.1. Técnicas de Criptografia de Dados

A criptografia pode ser definida como um método matemático para proteger informações em um recipiente "impenetrável" que só pode ser aberto com a chave matemática correta. As técnicas de criptografia, ou **criptosistemas**, são responsáveis pela infra-estrutura básica dos mecanismos de segurança nos ACDs. Usando algoritmos de criptografia, para cifrar e decifrar os

implementação dos serviços de segurança do ACD.

A principal dificuldade do serviço de identificação é a geração de *netnames* únicos, considerando a heterogeneidade das diversas máquinas e sistemas de redes que constituem o ACD. Os *netnames* são *strings* gerados em cada máquina, a partir do nome do sistema operacional, do identificador do usuário e do nome do subdomínio e domínio ao qual pertence o usuário. Por exemplo, em máquinas UNIX, o formato do *netname* poderia ser: *unix.uid@subdomínio.domínio*. Assumindo que o subdomínio seja *ufpb*, o domínio *edu* e o identificador do usuário (*uid*) 505, o *netname* gerado seria *unix.505@ufpb.edu*. Em uma máquina VAX, com sistema operacional VMS, o *netname* seria gerado a partir do formato *VMS.<uid,gid>@subdomínio.domínio*. A manutenção dos *netnames* está integrada com serviço de *naming* do ACD. Maiores informações sobre nomes de domínios podem ser encontradas em [Malamud92b] e [Commer91].

O serviço de identificação é fundamental para os outros serviços de segurança, tais como, autenticação, autorização e auditoria.

5.4.4. Autenticação

Em ACDs, é comum programadores de ADs terem acesso a APIs que permitem acessar os serviços básicos de comunicação e as interfaces de *software* (serviços) dos servidores. Logo, é possível que programadores mal intencionados, personificando usuários autorizados, tentem acessar serviços aos quais não estão autorizados, causando problemas. Um exemplo, é o caso de um servidor de arquivos que recebe um pedido para excluir todos os arquivos de um diretório privado. O servidor deve simplesmente assumir que o autor do pedido é confiável e excluir os arquivos? O correto seria, antes de excluir os arquivos, o servidor se certificar de que o cliente é realmente quem diz ser. Para resolver problemas dessa natureza, são utilizados os serviços de autenticação conjuntamente com o de identificação.

A autenticação é o processo pelo qual uma entidade tem a sua identidade verificada, para

eliminar possíveis dúvidas quanto à questão dela ser realmente quem diz ser. Toda entidade cuja identidade pode ser verificada pelo serviço de autenticação é tida como um ***principal***. O termo principal está associado a máquinas, usuários, processos, serviços e dispositivos confiáveis. Os serviços de autenticação fazem uso de protocolos baseados em criptosistemas simétricos e/ou de chave pública. Dois exemplos de protocolos de autenticação são: Secure RPC e Kerberos.

5.4.4.1. Secure RPC

A autenticação Secure RPC [Taylor86] , também conhecida como Autenticação DES, é fundamentada em dois criptosistemas: criptosistema simétrico com DES e criptosistema de chave pública baseado em uma variante do método RSA, conhecida como método de Diffie-Hellman [Diffie76]. A autenticação secure RPC pode ser dividida em duas fases: inicialização de credenciais e autenticação cliente-servidor.

Inicialização de Credenciais

Essa fase tem como função distribuir credenciais de identificação e chaves criptográficas, para o estabelecimento de uma sessão segura entre o cliente e o servidor. Em Secure RPC, o cliente é o responsável pelo início da transação. Inicialmente, o cliente cria randomicamente uma chave DES, conhecida como **chave de conversação**, usada para garantir uma sessão (diálogo) privada entre ele e o servidor.

O cliente envia para o servidor um conjunto de informações para viabilizar a sua autenticação. Esse conjunto de informações, denominado de **credenciais**, é composto pela chave de conversação, pelo nome do cliente e por uma "janela" que determina tempo de validade da credencial. Para enviar as credenciais com segurança, o cliente cifra a "janela" com a chave de conversação e, em seguida, cifra a chave de conversação com um esquema de chave pública.

Além das credenciais, o cliente envia um **verificador** para garantir que ninguém que capture as credenciais as utilize para estabelecer uma sessão irregular com o servidor.

O verificador é composto por um *timestamp*[3] e pelo valor da "janela", adicionado de um, ambos cifrados com a chave de conversação.

Após receber a credencial e o verificador do cliente, o servidor usa o esquema de chave pública, para obter a chave de conversação e checar a autenticidade das credenciais do cliente. Em seguida, o servidor armazena, para uso futuro, os dados obtidos do cliente em uma **tabela de credencias**, associando-os a um identificador. A partir desse ponto, o cliente e o servidor dispõem de uma chave comum de conversação e podem dar início a uma sessão privada com segurança.

Autenticação Cliente-Servidor

As trocas seguintes de informações entre o cliente e o servidor seguem o seguinte padrão:

a) o servidor envia para o cliente o identificador do cliente na tabela de credenciais e um verificador. O verificador é composto pelo *timestamp* da credencial do cliente subtraído de um, cifrado com a chave de conversação. A subtração de uma unidade do *timestamp* é para permitir que o cliente verifique se o servidor é um *principal*.

b) o cliente decifra o verificador recebido e checa se o *timestamp* confere com o que foi anteriormente enviado para o servidor. Caso afirmativo, o cliente assume que o servidor é confiável, isto é, que ele é um *principal*. Nas transações seguintes, o cliente envia, para o servidor, seu identificador e um *timestamp* cifrado com a chave de conversação. Se o servidor, ao decifrar o *timestamp* com a chave de conversação, obtiver um tempo válido, então ele assume que o cliente é um *principal*. Uma descrição completa dos protocolos de autenticação Secure RPC pode ser encontrada no Apêndice B.

5.4.4.2. Kerberos

Concebido como parte integrante do projeto Athena do *Massachusetts Institute of*

[3] Um *timestamp* é normalmente um valor de 64 bits que representa o tempo corrente de uma máquina, obtido a partir do seu relógio interno.

Technology (MIT), Kerberos[4] é um sistema de autenticação, baseado em um criptosistema de chave simétrica. Os principais componentes do sistema Kerberos são: servidor de autenticação Kerberos (KAS), servidor de *ticket* de permissão (TGS) e um banco de dados. O servidor de autenticação é a entidade que conhece as senhas de todos os usuários e serviços. O servidor de *ticket* de permissão fornece requisições (***tickets***) que são usadas pelos clientes, para se autenticarem com os servidores. O banco de dados contém as senhas e as chaves simétricas dos *principals*.

No esquema de autenticação Kerberos, sempre que um cliente deseja se autenticar com um servidor, ele é obrigado a utilizar o sistema Kerberos. Por esse motivo, Kerberos é conhecido como a "**terceira parte**" confiável. O processo de autenticação com Kerberos pode ser dividido em duas fases distintas: inicialização de credencial e autenticação cliente-servidor.

Inicialização de Credenciais

Quando o usuário inicia um *login* em uma estação de trabalho, o processo de *login* interage com o KAS solicitando um *ticket* para habilitar o usuário a acessar outros servidores. Após receber a solicitação, o KAS obtém do banco de dados a chave simétrica do usuário. Em seguida, o KAS gera um *ticket* de permissão ($ticket_{TGS}$) e uma chave de conversação randômica, que são cifrados com a chave simétrica do usuário e retornados para o processo de *login*.

Finalmente, o processo de *login* obtém a chave simétrica (compartilhada somente com o KAS), aplicando uma função matemática à senha de *login* do usuário. Então, o processo de *login* usa a chave simétrica para decifrar a mensagem recebida do KAS e recuperar o $ticket_{TGS}$ e a chave de conversação. O $ticket_{TGS}$ e a chave de conversação serão usados no futuro, para estabelecer uma comunicação segura entre o usuário da estação de trabalho e o TGS.

Uma observação importante é que o $ticket_{TGS}$ é criado de tal forma que o seu conteúdo só é acessível pelo TGS.

---

[4] Segundo a mitologia Grega, Kerberos é o nome do cão de três cabeças, guardião da porta de entrada para o inferno.

Autenticação Cliente-Servidor

Quando um processo cliente do usuário deseja solicitar a execução de um serviço, ele deve, em primeiro lugar, solicitar ao TGS um $ticket_s$ que garanta a sua autenticidade perante o servidor responsável pelo serviço. Para isso, o cliente envia para o TGS o nome do servidor que ele deseja acessar, o $ticket_{TGS}$ (obtido na fase de inicialização) e um verificador, composto pelo nome do cliente e por um *timestamp*, ambos cifrados com a chave de conversação compartilhada com o TGS.

O TGS recebe o pedido e certifica-se da autenticidade do cliente checando o $ticket_{TGS}$ e o verificador. Em seguida, o TGS executa as seguintes operações: a) cria randomicamente uma chave de conversação (K_{CS}), para ser utilizada na futura comunicação entre o cliente e o servidor; b) cria o $ticket_s$ de permissão solicitado pelo cliente, contendo o nome do cliente e a chave K_{CS}, de tal forma que somente o servidor, a ser acessado pelo cliente, possa ler o seu conteúdo; c) envia para o cliente uma mensagem contendo o $ticket_s$ e a chave K_{CS}, ambos cifrados com a chave de conversação compartilhada pelo TGS e o cliente.

O cliente recebe a mensagem e usa a chave de conversação compartilhada com o TGS, para decifrá-la e obter o $ticket_s$ e a chave K_{CS}. Para solicitar a execução do serviço, o cliente envia um verificador cifrado com K_{CS} e o $ticket_s$ de permissão para o servidor. O servidor lê o conteúdo do $ticket_s$ para obter o nome do cliente e a chave de conversação K_{CS}. Com esses dados, o servidor pode checar o verificador enviado pelo cliente, certificando-se de que ele é um *principal*.

Uma descrição completa dos protocolos de Kerberos pode ser encontrada no Apêndice B.

5.4.5. Autorização

Uma vez que o usuário está autenticado, o esquema de segurança deve verificar que operações o usuário pode executar sobre os recursos que ele tentar acessar. Esse processo, denominado **autorização**, determina quem pode acessar que recurso do ACD e como.

Todo o esquema de autorização é controlado através da associação de permissões aos

recursos de um determinado domínio. O esquema de autorização pode ser abstraído como uma matriz de acessos (**Fig. 5-4**). Os domínios são representados pelos usuários, processos, e máquinas. Os recursos são representados por arquivos, diretórios, impressoras, discos e serviços.

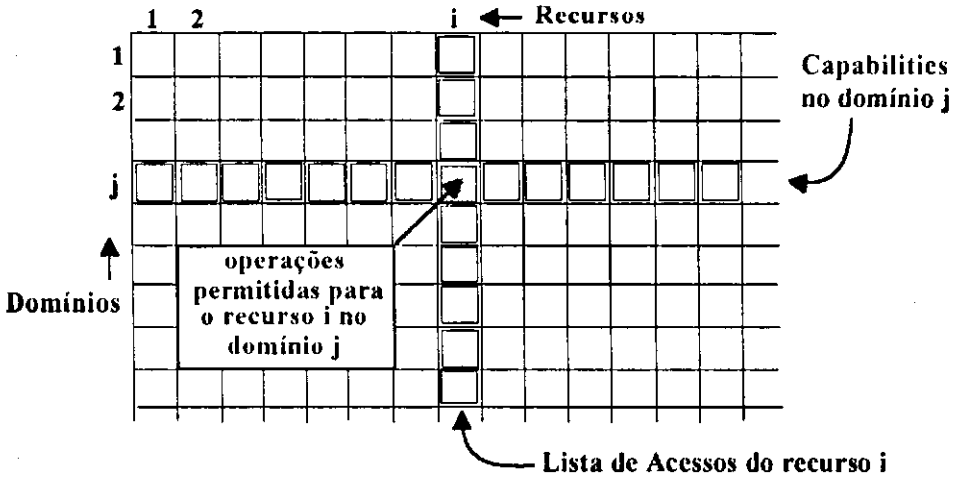


Figura 5-4. *Matriz de Lista de controle de Acesso (ACL) e Capabilities*

A matriz de acesso é normalmente muito esparsa e pode ser representada por um conjunto de listas que contém apenas os elementos não vazios. Existem duas alternativas normalmente usadas para a implementação da matriz de acessos: lista de controle de acesso e *capabilities*.

• **Lista de Controle de Acesso (ACL)**: nessa abordagem, as autorizações são representadas em forma de colunas da matriz de acesso. Uma ACL está associada a cada recurso e possui todos os domínios que podem ter acesso ao recurso, bem como as operações permitidas para cada domínio.

• **Capabilities**: as *capabilities* são a representação das autorizações através das linhas da matriz de acessos. Cada elemento pertencente a um determinado domínio possui uma lista com as suas permissões de acesso para cada recurso.

Maiores informações sobre ACL e *capabilities* podem ser encontradas em [Coulouris88].

5.4.6. Auditoria

O serviço de auditoria tem como objetivo permitir o monitoramento das ações dos usuários, para identificar possíveis problemas de segurança (quem fez o que e quando). Esse serviço permite detectar comportamentos impróprios, mas somente após a sua ocorrência.

Através do serviço de auditoria, as informações das operações sobre dados, recursos, sucesso e insucesso de *logins*, comandos executados por cada usuário, chamadas remotas e autenticações são armazenadas em um banco de dados. Para garantir um alto grau de confiabilidade e evitar problemas de segurança, as informações do banco de dados são normalmente cifradas e armazenadas em discos óticos tipo *write-once*.

Alguns problemas do serviço de auditoria são: degradação de desempenho do ACD e dificuldade do administrador em interpretar os dados armazenados. Para facilitar o trabalho de análise dos dados, armazenados pelo administrador, algumas implementações de serviço de auditoria colocam a sua disposição sistemas especialistas.

Nos capítulos anteriores, foram apresentados os requisitos básicos, alguns modelos de programação e considerações específicas, para o desenvolvimento de ADs. O próximo capítulo apresenta um estudo de caso prático, que consiste na implementação de um serviço de impressão distribuído, desenvolvido a partir das informações introduzidas pelos capítulos anteriores.

6. Implementações de um Serviço de Impressão Distribuído

Com o objetivo de aplicar os principais conceitos de projeto de ADs apresentados nos capítulos anteriores, obtendo assim considerações práticas importantes para o programador de ADs, foram especificadas e implementadas duas versões de um serviço de impressão distribuído. Neste capítulo, serão inicialmente apresentados o ACD utilizado e o serviço de impressão centralizado que serviram de base para as implementações. Em seguida, será definido o modelo conceitual adotado pelas implementações. Finalmente, será feita uma descrição de cada implementação, incluindo uma análise comparativa entre elas e considerações envolvendo os principais requisitos de projetos de ADs.

6.1. Infra-estrutura usada pelas Implementações

As duas versões do serviço de impressão distribuído foram escritas em linguagem "C" e implementadas em uma rede Ethernet de 10 Mbits, composta por 7 estações de trabalho SUN (4 Sparc/2 e 3 SLCs) e por uma pilha de protocolos TCP/IP.

A plataforma utilizada foi a *Open Network Computing* (ONC) da SUN. Essa plataforma oferece um conjunto de serviços que permitem o desenvolvimento de ADs para ambientes heterogêneos. Seus principais serviços são: *Network File System* (NFS) - o serviço de arquivos distribuídos da SUN; *Network Information Service* (NIS) - o serviço de diretório distribuído (*naming*); *Network Lock Manager* (NLM) - o serviço de travamento de arquivos distribuído; *External Data Representation* (XDR) - o serviço de apresentação utilizado para resolver problemas de representação de dados; *Remote Procedure Call* (RPC) - o serviço usado para fazer

chamadas de procedimentos remotos; *Secure RPC* - um serviço de autenticação. Maiores detalhes sobre os serviços da plataforma ONC podem ser encontrados em [Lyon84a], [Lyon84b], [Sun88a] e [Sun88b].

6.2. Serviço de Impressão Spoolview

Nesta seção, será apresentado o serviço centralizado de impressão *spoolview* [Infocon91] que serviu de base para as duas implementações do serviço de impressão distribuído. Serão descritos detalhes de seu funcionamento e de sua implementação. É importante que o programador se familiarize com esses detalhes, a fim de assimilar as considerações que serão feitas no estudo das versões implementadas.

6.2.1. Funcionamento do Spoolview

O *spoolview* é um serviço de impressão originalmente concebido para prover o gerenciamento de impressão em ambientes centralizados UNIX. O *spoolview* oferece um conjunto de serviços que, além de permitir a impressão de arquivos, fornece aos usuários informações detalhadas sobre: impressoras alocadas, formulários disponíveis, arquivos atualmente sendo impressos, tempo de impressão, dentre outras facilidades. Esses serviços são acessados diretamente pelo usuário, através de comandos não interativos, a partir do interpretador de comandos do sistema operacional (*shell*). Os comandos podem ser executados em dois modos: superusuário e usuário.

- **Modo Superusuário**: é acessível somente pelo administrador do sistema e permite a execução de comandos inerentes ao gerenciamento de impressão, tais como: alocação e desalocação de impressoras ou cancelamento de impressão.

- **Modo Usuário**: permite a execução dos comandos que não são de competência exclusiva do administrador do sistema, como por exemplo, a impressão de arquivos ou consultas para verificar o andamento das impressões.

Para tornar possível o funcionamento dos comandos, o *spoolview* mantém várias informações em um conjunto de arquivos de controle. O arquivo *impressoras* mantém informações sobre as impressoras disponíveis. O arquivo *global* contém o número e a prioridade do último pedido de impressão submetido. O arquivo *controle* representa a fila de pedidos de impressão. Os arquivos cujo nome se iniciam com as letras *cf* representam arquivos temporários, que contêm as diretivas de impressão dos pedidos submetidos. Os arquivos cujos nomes se iniciam com as letras *df* são arquivos temporários usados para guardar uma cópia física dos arquivos a serem impressos. Maiores detalhes sobre os arquivos de controle estão contidos em [Infocon91].

A utilização do *spoolview* pode ser dividida em três etapas: alocação de impressoras, impressão de arquivo e gerenciamento de impressão.

6.2.1.1. Alocação de Impressoras

O *spoolview* tem a capacidade de gerenciar uma ou mais impressoras em um ambiente centralizado UNIX. Inicialmente, o administrador do sistema deve cadastrar as impressoras a serem gerenciadas pelo *spoolview*, para torná-las disponíveis para os usuários.

Cada impressora tem um nome lógico (ex.: laser, principal ou matricial), definido pelo administrador, e está sempre associada a um dispositivo físico (ex.: porta de comunicação). A associação de um nome lógico de impressora a um dispositivo físico é conhecida como **alocação**.

A operação de alocação tem como objetivo tornar transparente, para os usuários, detalhes técnicos que só dizem respeito ao administrador do sistema, como por exemplo, a existência de dispositivos físicos ou portas de comunicação.

A alocação de impressoras é feita pelo administrador, executando o comando ***alo*** do *spoolview* no modo superusuário (*root*). A **Fig. 6-1** apresenta de forma simplificada os passos necessários para a alocação de uma impressora, bem como as entidades envolvidas.

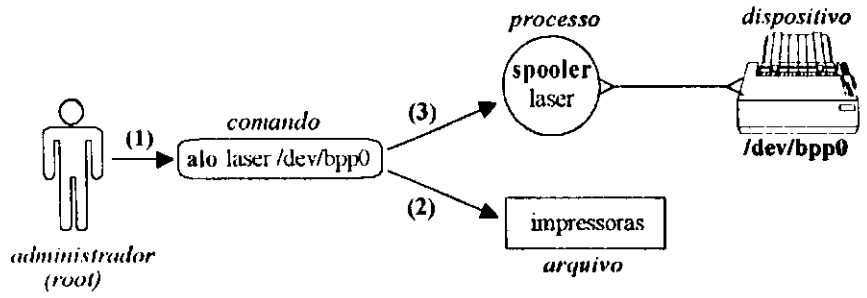


Figura 6-1. *Entidades envolvidas na alocação de impressoras*

1. Para alocar uma impressora, o administrador do sistema executa o comando ***alo***. O comando *alo* tem dois argumentos: o primeiro, *laser*, representa o nome lógico da impressora a ser alocada, e o segundo, */dev/bpp0*, o nome do dispositivo fisico ao qual o nome lógico da impressora deverá ser associado.

2. O comando *alo* armazena informações sobre a impressora que está sendo alocada, no arquivo *impressoras*, que contém o registro de todas as impressoras alocadas.

3. O comando *alo* cria um processo servidor de impressão denominado *spooler laser*, associado ao dispositivo fisico */dev/bpp0*. Toda impressora alocada possui um servidor de impressão *spooler* individual para gerenciar sua impressão. O *spooler* é um processo com previlégios de superusuário e, portanto, não tem restrições para abrir e ler arquivos cujas permissões de acesso estejam restritas a um determinado usuário. Normalmente, os *spoolers* "dormem" quando não estão gerenciando uma impressão.

6.2.1.2. Impressão de Arquivos

Após a alocação das impressoras, o usuário está apto para solicitar a impressão dos seus arquivos. Toda solicitação de impressão é feita através de um comando chamado ***imp***, cuja função é gerar um pedido de impressão. Cada pedido de impressão é representado por um número de identificação único e está associado a um número de prioridade, que indica a sua ordem de impressão. O número do pedido é necessário, para cancelar um pedido ou acompanhar o

andamento de uma impressão. A **Fig. 6-2** mostra as operações necessárias e as entidades envolvidas para processar um pedido de impressão.

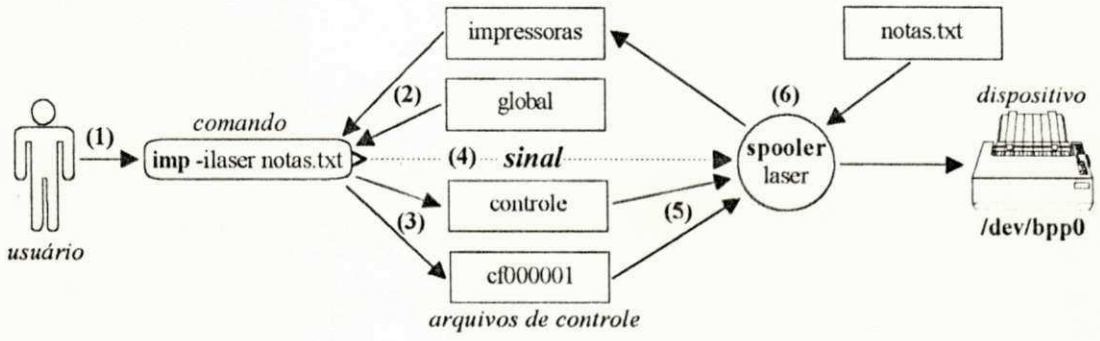


Figura 6-2. *Operações realizadas para a impressão de um arquivo*

1. Para imprimir um arquivo, o usuário executa o comando ***imp***, informando o nome da impressora a ser usada na impressão e o nome do arquivo a ser impresso. O nome da impressora é especificado após a opção **-i**. No exemplo, o usuário está solicitando a impressão do arquivo *notas.txt* na impressora *laser*.

2. O comando *imp* consulta o arquivo *impressoras*, para verificar se a impressora informada pelo usuário existe. Em seguida, o *imp* consulta o arquivo *global*, para obter o número do pedido de impressão e a prioridade do pedido.

3. O *imp* gera um pedido de impressão que é gravado no arquivo *controle*. Em seguida, o *imp* cria o arquivo temporário cf000001 que contém as diretivas de impressão, para orientar o servidor *spooler laser* durante a impressão do pedido em questão.

4. O comando *imp* envia um **sinal**[1] para o processo *spooler* da impressora onde será efetuada a impressão, informando que existe um pedido a ser processado no arquivo *controle*. Todos os comandos do *spoolview* que necessitam se comunicar de forma assíncrona com os processos *spoolers*, fazem-no através do envio de sinais.

---

[1] Um *sinal* é uma interrupção de *software* .

5. Ao receber um sinal do *imp*, o processo *spooler laser* "acorda" e consulta o arquivo *controle*, para obter os seus pedidos da fila de impressão. Em seguida, o *spooler* lê do arquivo cf000001 as diretivas de impressão associadas ao pedido a ser impresso.

6. Após analisar as informações do pedido e as diretivas de impressão, o processo *spooler* inicia o processo de impressão, abrindo o arquivo *notas.txt* e enviando o seu conteúdo para o dispositivo */dev/bpp0* (impressora *laser*). Durante a impressão do arquivo, o *spooler* atualiza periodicamente o arquivo *impressoras*, para permitir que o usuário possa acompanhar o andamento da impressão.

6.2.1.3. Gerenciamento de Impressão

O *spoolview* possui um conjunto de comandos que permitem o gerenciamento das impressoras alocadas e dos pedidos de impressão efetuados. Através deles, o usuário pode, por exemplo, suspender temporariamente o serviço de uma impressora, verificar o *status* das impressoras alocadas ou acompanhar o andamento da impressão dos pedidos.

Para suspender temporariamente os serviços de uma impressora de nome *laser*, bastaria o administrador executar o comando ***sus laser***. Nesse caso, o comando *sus* envia um sinal para o *spooler laser*, indicando que o processamento de impressão deve ser paralisado.

Para visualizar informações de status da impressora *laser*, poderia ser usado o comando ***sta laser***, que mostra informações sobre o estado da impressora (imprimindo, livre ou suspensa), o número do pedido que está sendo impresso e o percentual já impresso do pedido.

As informações específicas sobre os pedidos que estão na fila de impressão, como por exemplo, a hora prevista para o término da impressão, podem ser obtidas através da execução do comando ***mos***.

Em [Infocon91,] podem ser encontradas informações mais detalhadas sobre os comandos do *spoolview*.

6.2.2. Esquema de Tratamento de Erros

O esquema de tratamento de erros do *spoolview* considera a ocorrência de duas classes de erros: erros fatais e erros de advertência. Os **erros fatais** levam à exibição de uma mensagem de erro na *console* do usuário e ao término imediato do processo do comando. Os **erros de advertência** levam à exibição da mensagem de erro na *console* da máquina do usuário, mas o processo do comando continua normalmente a execução.

6.2.3. Política de Travamento dos Arquivos de Controle

No *spoolview*, todos os processos que necessitam realizar operações de atualização, nos arquivos de controle, adotam o seguinte esquema: a) travam os arquivos de controle no modo exclusivo; b) realizam as operações de atualização; c) destravam os arquivos de controle. Isso é necessário para evitar inconsistência dos dados, visto que os arquivos de controle são compartilhados por vários processos. Essa política de "**trava()....faz operações...destrava()**" obriga qualquer processso que necessite atualizar os arquivos de controle a ficar aguardando até que os arquivos estejam destravados.

6.2.4. Facilidade da Impressora Qualquer

O *spoolview* possui uma impressora de nome reservado chamada ***qualquer***. Essa impressora é usada para imprimir o mais rapidamente possível um determinado pedido de impressão. Quando o usuário faz um pedido de impressão para a impressora *qualquer*, o *spoolview* automaticamente o direciona para uma impressora alocada que estiver em condições de imprimí-lo no menor tempo possível. O critério usado pelo *spoolview* para a escolha da impressora *qualquer* é baseado no número de *bytes* do arquivo a ser impresso. O *spoolview* analisa os dados contidos nos arquivos de controle e calcula o tempo previsto para o término da impressão. A impressora que retornar o menor tempo para processar o pedido é eleita como a *qualquer*.

6.3. Definição do Modelo das Implementações

Nesta seção, será definido o modelo das implementações do serviço de impressão distribuído, projetado a partir do *spoolview*. Inicialmente, serão identificados os requisitos básicos do serviço de impressão distribuído. Finalmente, será definido um modelo de implementação baseado no modelo C-S.

6.3.1. Requisitos Básicos das Implementações

As implementações do serviço de impressão distribuído devem atender, basicamente, a três requisitos: a) permitir que o usuário imprima seus arquivos em impressoras locais e remotas; b) tornar transparente para o usuário as diferenças entre impressão local e remota; c) preservar a funcionalidade original do *spoolview*.

Para atender a esses requisitos, a implementação original do *spoolview* teve que sofrer várias modificações. Essas modificações foram formuladas a partir da divisão das principais operações realizadas pelo *spoolview*, em dois grupos distintos: operações nos arquivos de controle e operações dos gerenciadores de impressão *spooler*.

• **Operações nos Arquivos de Controle**: as operações sobre o conjunto de arquivos de controle representam a maior parte do trabalho realizado pelo *spoolview*. Elas são responsáveis pelo controle e gerenciamento de todo o processo de impressão dos arquivos. Um exemplo são as operações realizadas sobre a fila de impressão, que é representada pelo arquivo *controle*.

• **Operações de Spooler**: essas operações são responsáveis pela criação dos *spoolers* e pelo envio de sinais dos comandos do *spoolview* para os *spoolers*.

Na versão original do *spoolview*, esses dois grupos de operações são sempre realizadas localmente. Para atender aos requisitos de projeto do serviço de impressão distribuído, foi necessário adequar essas operações para funcionarem também remotamente, ou seja, através de diferentes máquinas do ACD. Para isso, as modificações introduzidas no projeto original do *spoolview* foram modeladas a partir do modelo C-S.

6.3.2. Implementações e o Modelo Cliente-Servidor

O *spoolview* foi dividido conceitualmente em duas partes: lado cliente e lado servidor (**Fig. 6-3**). No **lado cliente**, estão os comandos do serviço de impressão distribuído (ex.: *alo, imp, sus*) acessíveis ao usuário. Do **lado servidor**, estão localizados os gerenciadores de impressão *spoolers* e o conjunto de arquivos de controle. Todos os comandos do lado cliente estão habilitados a realizar as operações envolvendo os arquivos de controle e os *spoolers* no lado servidor. É importante observar que todas as operações envolvendo os arquivos de controle e os *spoolers* passam a ser vistas como **serviços** oferecidos pelo lado servidor. Esses serviços são sempre solicitados pelos comandos do lado cliente. Por exemplo, um pedido de impressão feito pelo lado cliente é gravado nos arquivos de controle do lado servidor através de um serviço específico. O modelo proposto na **Fig 6-3** não impõe restrições para que os componentes do lado cliente e do lado servidor residam na mesma máquina.

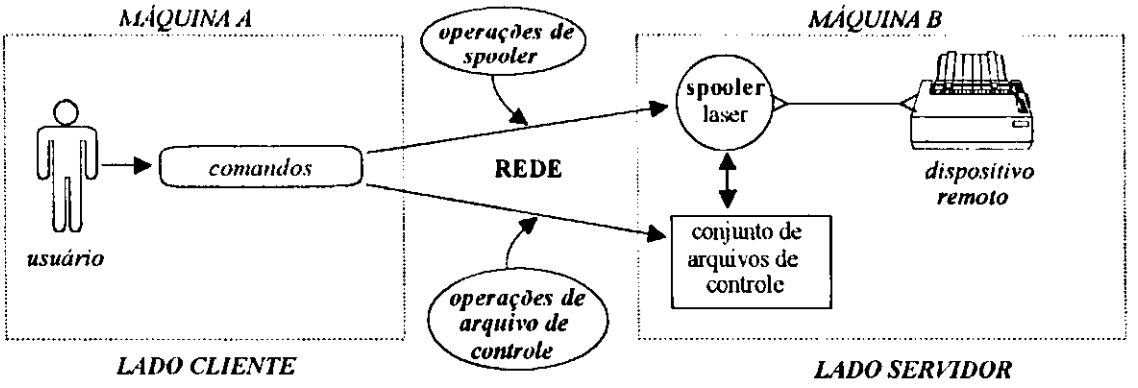


Figura 6-3. *Modelo usado pelas versões distribuídas do spoolview*

O principal problema a ser resolvido pelo programador para implementar o modelo proposto na **Fig 6-3** é como realizar remotamente as operações sobre os arquivos de controle e as operações de *spoolers*.

Existem várias soluções para resolver esse problema. Duas alternativas adotadas por este trabalho deram origem a duas implementações de um serviço de impressão distribuído, que serão descritas nas próximas seções.

6.4. Implementação Dependente de DFS

Nesta seção, será descrita a primeira implementação do serviço de impressão distribuído baseada no *spoolview*. Inicialmente, é dada uma visão global das entidades que compõem a implementação, bem como os problemas resolvidos por cada serviço da plataforma ONC. Finalmente, serão apresentadas considerações específicas envolvendo os principais requisitos de ADs.

6.4.1. Visão Geral da Implementação

Por utilizar um DFS[2], para realizar parte das operações remotas, a primeira implementação foi denominada de versão ou implementação dependente de DFS. O principal objetivo da versão dependente de DFS é obter um serviço de impressão distribuído a partir do *spoolview*, introduzindo o mínimo de alterações no projeto original. Essa versão utiliza três serviços da plataforma ONC: *Network File System* (NFS), *Remote Procedure Call* (RPC) e *External Data Representation* (XDR). A **Fig. 6-4** mostra uma visão global das entidades que compõem a implementação, bem como sua relação com o NFS, RPC e XDR.

De forma geral, o NFS é usado para resolver o problema das operações remotas sobre os arquivos de controle. O RPC, para resolver os problemas das operações de *spoolers* e, finalmente, o serviço de apresentação XDR, para tratar problemas de heterogeneidade de representação de dados. A seguir, serão descritos com maiores detalhes os problemas resolvidos com cada um desses serviços, considerando o contexto da **Fig. 6-4**.

---

[2] O DFS, ou Distributed File System, foi descrito na seção 2.2.6.

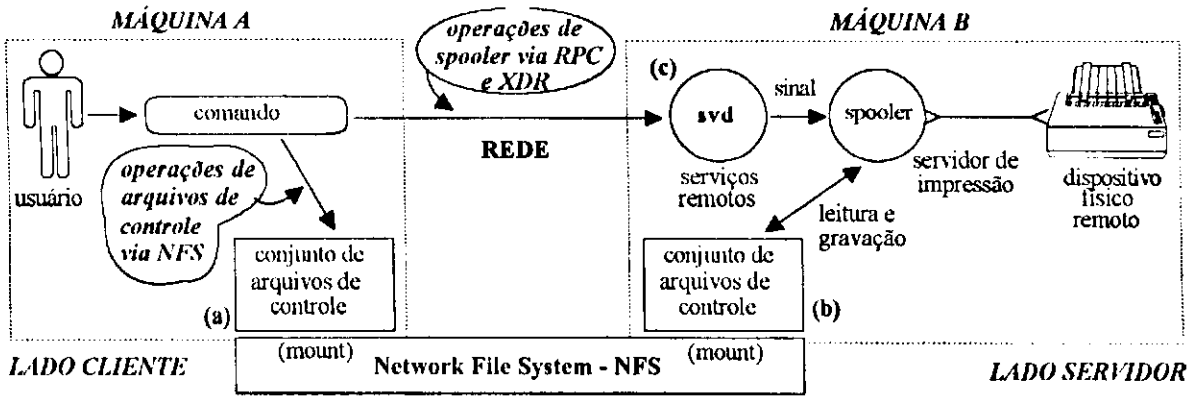


Figura 6-4. *Visão global das entidades e dos serviços da versão dependente de DFS*

6.4.2. Problemas Resolvidos com NFS

O NFS foi usado para eliminar a programação de todas as operações remotas relacionadas com o conjunto de arquivos de controle. Como esse tipo de operação representa a maior parte do trabalho realizado pelo *spoolview*, a utilização do NFS simplificou a implementação, minimizando grande parte do trabalho de programação.

Para implementar a solução com NFS, inicialmente, foi selecionada uma máquina do ACD em cujo sistema de arquivos foi criado um diretório contendo os arquivos de controle. O critério usado para a escolha da máquina foi a disponibilidade de discos físicos com espaço suficiente, para suportar o volume de dados requerido pelos arquivos de controle. Através do comando ***mount***[3] do NFS, o diretório dos arquivos de controle foi montado no sistema de arquivos de todas as máquinas usuárias do *spoolview*. Dessa forma, o mesmo conjunto de arquivos de controle fica disponível para todos os usuários, independentemente em que máquina estejam, possibilitando o seu **compartilhamento** por processos de diferentes máquinas. Caso fosse necessário montar os arquivos de controle, em estações de trabalho sem disco, não existiria nenhum problema; o NFS oferece todo o suporte para isso. Quando um comando do *spoolview* realiza operações de escrita sobre os arquivos montados no lado cliente (**Fig. 6-4a**), o resultado

[3] O comando *mount* é um serviço do NFS, que torna sistemas de arquivos remotos disponíveis em uma máquina local.

passa a ser visível no lado servidor onde os arquivos também estão montados (**Fig. 6-4b**). De forma análoga, qualquer operação realizada no lado servidor também passa a ser visível do lado cliente. Isso torna desnecessário alterar o código original do *spoolview*, para implementar as operações remotas sobre os arquivos de controle. Consequentemente, obteve-se menor nível de complexidade e redução dos custos da implementação.

6.4.3. Problemas Resolvidos com RPC

O serviço de RPC foi usado para implementar as operações remotas que envolvem os gerenciadores de impressão *spoolers*. Para possibilitar a execução desse tipo de operação, foram programados dois componentes: componente do lado servidor e componente do lado cliente.

6.4.3.1. Componente do Lado Servidor

O componente do lado servidor é representado por um processo denominado *svd* (*spoolview daemon*) que contém os procedimentos remotos das operações de *spooler* (**Fig. 6-4c**). O *svd* recebe e executa as solicitações de serviço feitas através de RPCs pelos comandos do lado cliente. Na versão dependente de DFS, o *svd* oferece três serviços: envio de sinal, validação de dispositivo e criação dos processos *spoolers*.

• **Envio de Sinais**: o esquema de comunicação entre vários comandos do *spoolview* e os *spoolers* é feito através de sinais. Como, na realidade, não existe analogia de rede para sinais, é necessário implementar serviços que envolvam esse esquema de comunicação com RPCs. As RPCs de sinais são usadas, por exemplo, para suspender temporariamente, prosseguir ou cancelar os serviços de uma impressora remota.

• **Validação de Dispositivo**: esse serviço é usado durante a alocação de impressoras remotas. Ele é necessário, para verificar se o dispositivo físico remoto a ser associado ao nome lógico de uma impressora é válido. O programador poderia imaginar montar (*mount*) com NFS os dispositivos remotos no lado cliente a fim de evitar a implementação desse serviço. Infelizmente,

essa solução não funciona, porque dispositivos remotos montados com NFS passam a ser vistos como dispositivos locais. Por questões de implementação, o NFS não faz o mapeamento das operações realizadas sobre dispositivos montados, para os respectivos dispositivos originais, através da rede.

• **Criação de Spoolers**: esse serviço é usado para criar processos *spoolers* remotamente. Ele é necessário porque alguns comandos do *spoolview* necessitam criar os *spoolers* em diferentes máquinas da rede. Por exemplo, durante o processo de alocação de uma impressora remota, o comando *alo* usa esse serviço, para criar um *spooler* na máquina onde reside fisicamente o dispositivo de impressão.

6.4.3.2. Componente do Lado Cliente

O componente do lado cliente é basicamente constituído por uma biblioteca que contém as funções usadas pelos comandos do *spoolview*, para realizar as operações de *spoolers*. As funções utilizam RPCs para acessar os serviços do *svd*. Essa biblioteca pode ser abstraída como uma camada de *software*, que isola o código fonte dos comandos da versão dependente de DFS, dos detalhes de RPC.

6.4.4. Problemas Resolvidos com XDR

O serviço de apresentação XDR é usado, pelos componentes do lado cliente e do lado servidor, para tratar possíveis inconsistências de dados, causadas pela provável heterogeneidade do ambiente. Todos os parâmetros e resultados das RPCs são sempre convertidos com XDR para uma forma canônica, antes de serem transmitidos através da rede e, desconvertidos, para o formato original, antes de serem utilizados. O XDR é usado nos seguintes casos: a) o lado cliente sempre converte os parâmetros das RPCs antes de fazer qualquer chamada remota; b) o lado servidor sempre converte qualquer resultado antes de retorná-lo para o lado cliente. A desconversão ocorre quando: a) o lado cliente desconverte os resultados das RPCs recebidos do

lado servidor; b) o lado servidor desconverte os parâmetros das RPCs feitas pelo lado cliente.

6.4.5. Considerações Específicas sobre a Implementação

Nesta seção, será feita uma análise da versão dependente de DFS, considerando os principais requisitos que normalmente envolvem um projeto de AD. Os requisitos analisados são: semântica de compartilhamento, endereçamento de arquivos remotos, transporte, transparência, escalabilidade, tratamento de erros, disponibilidade, tolerância a falhas, segurança, desempenho e funcionalidade.

6.4.5.1. Semântica de Compartilhamento de Arquivos

A semântica de compartilhamento de arquivos define precisamente os efeitos causados pelas operações de leitura e gravação, quando dois ou mais usuários compartilham um mesmo arquivo. Projetos que utilizam arquivos, montados com NFS, devem prever as implicações que o modelo semântico de compartilhamento pode ter no funcionamento da aplicação. Exemplos dessas implicações foram observados, durante a implementação da versão dependente de DFS.

Implicações da Semântica de Compartilhamento

Seria de se esperar que as operações remotas realizadas sobre os arquivos de controle, compartilhados por vários clientes e servidores, através do NFS, funcionassem adequadamente. Infelizmente, alguns comandos da versão dependente de DFS apresentaram problemas decorrentes da semântica de compartilhamento do NFS. Um exemplo é o comando *imp*, que não funcionava corretamente quando o pedido de impressão era feito para impressoras remotas. Quando o usuário solicitava uma impressão que se destinava a uma impressora remota, o comando *imp* gravava o pedido de impressão nos arquivos de controle montados com NFS (**Fig. 6-5a**). Em seguida, o *imp* enviava um sinal via RPC para o *spooler* da impressora solicitada, indicando a existência de um pedido a ser processado na fila de impressão (**Fig. 6-5b**). Após

receber o sinal, o *spooler* "acordava" e efetuava a leitura da fila de impressão, mas, curiosamente, não encontrava o pedido gravado no lado cliente (**Fig. 6-5c**). Consequentemente, o *spooler* voltava a "dormir" e o pedido não era impresso.

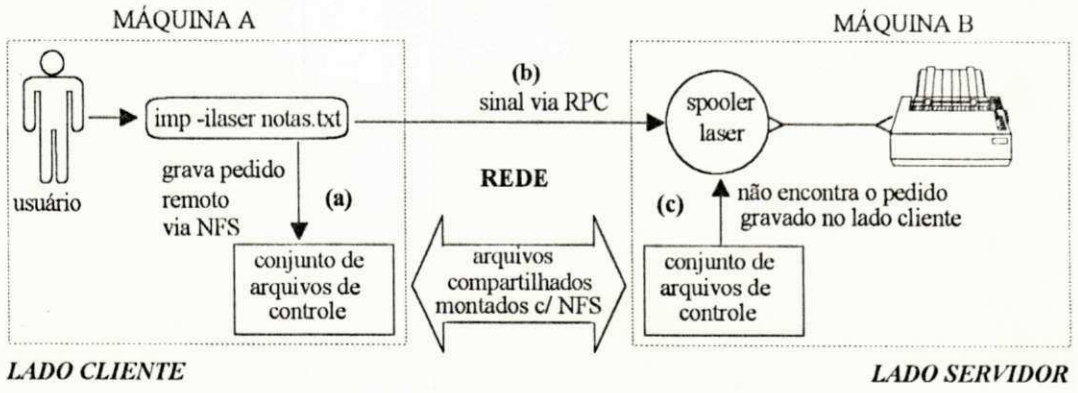


Figura 6-5. *Problemas causados pela semântica de compartilhamento de arquivos do NFS*

O problema apresentado pelo comando *imp* deve-se ao fato do NFS não oferecer semântica de compartilhamento compatível com o modelo UNIX, para o qual as operações sobre os arquivos de controle do *spoolview* foram originalmente projetadas. Na verdade, o NFS não possui uma semântica de compartilhamento bem definida (as razões estão descritas no Apêndice A). Isso significa que, quando um arquivo montado com NFS é compartilhado por clientes e servidores de diferentes máquinas, existe a possibilidade das operações de escrita, realizadas por um determinado cliente, não serem imediatamente visíveis pelos outros clientes e servidores.

O desconhecimento dos problemas causados pelos efeitos da semântica de compartilhamento do NFS podem induzir o programador a procurar erros inexistentes no código fonte da aplicação. Por exemplo, no caso do problema apresentado pelo comando *imp*, o programador poderia concluir erroneamente que os pedidos não estavam sendo gravados corretamente do lado cliente ou que o *spooler* não os estaria lendo corretamente da fila de impressão no lado servidor. A procura equivocada do "erro", no código fonte do *imp* e do

spooler, resultaria em considerável perda de tempo por parte do programador, sem resolver o problema.

Solução para o Problema de Semântica de Compartilhamento

O fato do NFS não possuir uma semântica de compartilhamento bem definida, transfere para o programador a responsabilidade de tratar, a nível de aplicação, os efeitos negativos provenientes dessa política. Através do uso de alguns *system calls* é possível adequar a semântica de compartilhamento do NFS aos requisitos da aplicação. Por exemplo, no caso do *imp* (e de outros comandos) o problema foi resolvido com o uso de um *system calls* de fechamento de arquivos. Para isso, foram necessárias as seguintes modificações: a) do lado cliente, o código do comando *imp* foi modificado, para fechar os arquivos de controle após a gravação dos pedidos de impressão remota; b) do lado servidor, como o *spooler* trabalha com os arquivos de controle abertos, o código foi modificado, para fechar e abrir novamente os arquivos de controle toda vez que o *spooler* receber um sinal para ler a fila de impressão. Essas modificações adequaram a semântica de compartilhamento do NFS aos requisitos do comando *imp*, resolvendo o problema. As razões do uso do *system call* de fechamento de arquivo e outras estratégias, para resolver os problemas de semântica de compartilhamento do NFS, podem ser encontradas no Apêndice A.

6.4.5.2. Endereçamento de Arquivos Remotos

Em um ACD, é comum existirem vários sistemas de arquivos independentes, distribuídos por diversas máquinas. Portanto, é possível que processos de uma determinada máquina não tenham acesso a arquivos de outras máquinas. Essa possibilidade cria sérios problemas de endereçamento de arquivos para o processamento de pedidos de impressão em impressoras remotas. Um exemplo seria o usuário de uma máquina ***A*** solicitar a impressão de um arquivo disponível somente na sua máquina, na impressora remota de uma máquina ***B***. Como o *spooler* da máquina ***B***, responsável pela impressão, vai acessar e imprimir um arquivo que só existe no

sistema de arquivos da máquina *A*?

Uma solução para resolver esse problema é colocar uma cópia do arquivo a ser impresso, disponível na máquina do *spooler* responsável pela impressão. Isso pode ser feito criando-se a cópia do arquivo no diretório dos arquivos de controle, sempre que o pedido de impressão destinar-se a uma impressora remota. Como na versão dependente de DFS, o diretório dos arquivos de controle está montado em todas as máquinas usuárias do *spoolview*, a cópia passa também a ser visível para todos os *spoolers*. Existem duas considerações importantes a serem observadas com relação à implementação dessa solução: criação do arquivo de cópia e conflito de nomes das cópias.

• **Criação do Arquivo de Cópia**: a implementação original do comando *imp* possui uma opção (*-g*) que gera uma cópia do arquivo a ser impresso no diretório dos arquivos de controle. Quando a opção *-g* é ativada, a cópia é armazenada em um arquivo **temporário** que é impresso no lugar do arquivo original. O código original do *imp* foi modificado, para gerar a cópia do arquivo a ser impresso toda vez que a impressão for remota. O *imp* sabe que a impressão é remota, comparando o nome da máquina onde o pedido de impressão é solicitado com o nome da máquina da impressora solicitada. Se os nomes forem diferentes a opção *-g* é automaticamente ativada e a cópia do arquivo a ser impresso é criada no diretório dos arquivos de controle.

• **Conflito de Nomes dos Arquivos de Cópia**: simplesmente ativar a opção -g do *imp* não resolve todos os problemas. Existe um outro problema a ser considerado, relacionado com a criação dos arquivos temporários e com o diretório centralizado onde eles são gravados. Os arquivos temporários são criados através de uma função específica denominada *mktemp*. Isoladamente, em cada máquina, essa função sempre cria arquivos temporários com nomes diferentes, mas, do ponto de vista de um conjunto de máquinas, existe a possibilidade de serem criados arquivos temporários com nomes idênticos. Como todos os arquivos temporários são gravados no diretório compartilhado dos arquivos de controle, os arquivos temporários com o mesmo nome, criados por diferentes máquinas, são gravados um sobre o outro. Essa situação

resulta no processamento incorreto dos pedidos de impressão. Para resolver esse problema, foi escrita uma nova função *mktemp* que gera nomes de arquivos temporários distintos, independentemente de máquina. Isso foi feito criando o nome dos arquivos temporários a partir do número do pedido de impressão. Por exemplo, *df000013* representa o nome de um arquivo temporário criado a partir do pedido de número treze. Essa solução só é viável, porque na versão dependente de DFS, o espaço dos números de pedidos é único para todas as máquinas.

6.4.5.3. Transporte

Na versão dependente de DFS, o transporte selecionado para as RPCs foi o UDP. Essa escolha foi baseada principalmente no critério de desempenho. Outras considerações importantes, relacionadas com o serviço de transporte da implementação são: volume de dados, semântica e idempotência e nível de confiabilidade do transporte.

• **Desempenho**: a grande vantagem do UDP é que ele é um transporte leve que consome poucos recursos. Consequentemente, as RPCs que realizam as operações de *spooler* ficam, em condições normais (ex.: rede não está congestionada), extremamente rápidas.

• **Volume de Dados dos Parâmetros**: normalmente, as implementações de transporte UDP têm o tamanho do datagrama limitado a 8k *bytes*. Como o tamanho dos argumentos e resultados das RPCs do *svd* são inferiores a 8k *bytes*, UDP pôde ser usado sem nenhum problema.

• **Semântica de Chamada e Idempotência**: as RPCs baseadas em UDP possuem semântica de chamada remota de pelo-menos-uma-vez. Portanto, é necessário avaliar se os três serviços oferecidos pelo *svd* são não idempotentes.Os dois primeiros, validação de dispositivos e envio de sinais, são idempotentes. O terceiro serviço, responsável pela criação dos processos *spoolers*, é não idempotente. Nesse caso, o uso de UPD não seria aconselhável. Entretanto, foi adotada uma estratégia para tratar a não idempotência no próprio procedimento. O código original do procedimento de criação de *spooler* foi alterado para verificar se o *spooler* a ser

criado existe. Se o *spooler* existir a operação de criação do processo não é realizada. Dessa forma, o procedimento de criação de *spoolers* passa a se comportar como um procedimento idempotente. Como os procedimento do *svd* são (ou se "tornaram") idempotentes, a semântica de pelo-menos-uma-vez não compromete a integridade da implementação.

- **Nível de Confiabilidade do Transporte**: em uma rede local, como por exemplo, *Ethernet*, o uso de UDP não compromete o funcionamento dos comandos da implementação dependente de DFS. Devido ao *hardware Ethernet* tratar os erros de perda de pacotes, a implementação funciona sem problemas. Por outro lado, em grandes ACDs, a não confiabilidade do UDP pode comprometer o funcionamento de alguns comandos. Podem ocorrer casos em que um pedido de impressão não possa ser atendido por problemas de perda de datagramas das RPCs. Por exemplo, se o datagrama que contém uma RPC com um sinal para "acordar" um *spooler* se perde, o *spooler* permanece "dormindo" e o pedido de impressão não é processado. Quando isso ocorre, os comandos da versão dependente de DFS cancelam o pedido de impressão e emitem uma mensagem de erro para o usuário. O usuário então é obrigado a repetir o comando que falhou. Dependendo do índice de perda de datagramas, isso pode representar um sério problema.

6.4.5.4. Transparência

Um dos principais requisitos da implementação dependente de DFS é a transparência de localização de impressoras remotas. Do ponto de vista do usuário, não deve existir diferença entre um pedido de impressão local e um pedido de impressão remota. Atender a esse requisito não é tão simples. Para endereçar corretamente uma impressora no ACD, é necessário especificar, além do seu nome, a máquina e o domínio onde ela reside[4]. Isso significa que o espaço de endereçamento de impressoras no ACD é tridimensional. Para tornar transparente para os usuários a complexidade do espaço de endereçamento de impressoras, e simplificar a implementação, foram necessárias duas alterações no *spoolview*: acrescentar o nome da máquina

[4] O padrão de nome da máquina e domínio utilizado pelas implementações é o *Fully Qualified Domain Name* (FQDN) da Internet. Cada nome de máquina tem o formato *<nome do sistema ou localização>.<domínio>*.

• **Tratamento de Erros**: a estratégia de tratamento de erros, usado por essa implementação, considera os erros fatais e de advertência que podem ocorrer durante as operações realizadas com RPC e NFS. O esquema adotado para tratar erros ocorridos nas RPCs é simples. Todos os procedimentos remotos do *svd* retornam em caso de exceção um código que indica a natureza do erro ocorrido. Ao receber o código de erro, o cliente exibe uma mensagem na *console* do usuário. Nenhum dos procedimentos remotos do *svd* gera erros de advertência, apenas erros fatais. Somente operações feitas através do NFS podem dar origem a erros de advertência e erros fatais. Como do ponto de vista do lado cliente, as operações feitas via NFS são sempre locais, as rotinas de tratamento de erros do *spoolview* original puderam ser utilizadas, nesse caso, praticamente sem nenhuma modificação.

6.4.5.7. Segurança

As operações sobre os arquivos de controle, feitas através do NFS, e as operações de *spoolers*, feitas com RPCs, envolvem a comunicação entre processos que podem residir em diferentes máquinas. Portanto, é necessário tomar alguns cuidados relacionados com os requisitos de segurança, para evitar que usuários não autorizados tenham acesso a serviços restritos e causem algum tipo de problema.

Sem um esquema de segurança, qualquer programador com acesso a API de RPC pode tentar executar serviços de uso restrito do protocolo NFS ou os serviços do *svd*. Para reduzir essa possibilidade, foram considerados na implementação dependente de DFS, requisitos de segurança tanto a nível de RPC quanto a nível de NFS.

• **Segurança a Nível de RPC**: é necessária para impedir que usuários não autorizados acessem os serviços do *svd*. Todas as RPCs feitas pelos comandos do *spoolview* são sempre autenticadas antes do *svd* executar qualquer serviço. Para isso, foi utilizado o protocolo de autenticação Secure RPC (veja Seção 5.4.4.1). O Secure RPC acrescenta às RPCs um esquema de credenciais e verificadores que garante a execução dos serviços do *svd* somente por usuários

autorizados.

• **Segurança a Nível de NFS**: é necessária para garantir a integridade das operações realizadas sobre os arquivos de controle. Duas facilidades oferecidas pelo NFS foram utilizadas para garantir a segurança da implementação: monitoração de portas e NFS seguro.

A **monitoração de portas** é usada para previnir falsificações de pedidos NFS válidos, enviados por usuários não autorizados. Por exemplo, um intruso poderia usar serviços de baixo nível da rede para forjar um pedido NFS válido e enviá-lo para a porta associada a um processo servidor do NFS (*nfsd*). A ativação da monitoração de portas é feita, através da configuração de parâmetros do NFS, pelo administrador do sistema.

O **NFS seguro** é usado para garantir a segurança das operações realizadas sobre os arquivos montados. O NFS seguro faz uso do serviço de autenticação Secure RPC e pode ser ativado através da opção *-secure* do comando *mount*.

Informações adicionais sobre monitoração de portas e NFS seguro podem ser encontradas em [Stern91].

6.4.5.8. Desempenho

O desempenho da versão dependente de DFS pode ser avaliado, fundamentalmente, em função do desempenho das operações realizadas com NFS e RPC. Considerações sobre o impacto do uso de NFS e RPC, no desempenho da implementação, serão discutidos a seguir. Também será feita uma avaliação da relação entre o desempenho e a escalabilidade da implementação. As medidas de desempenho, apresentadas a seguir, foram efetuadas em uma rede Ethernet de 10 Mbits, composta por 7 estações de trabalho SUN (4 Sparc/2 e 3 SLCs) e por uma pilha de protocolos TCP/IP.

• **Desempenho das Operações com NFS**: a eficiência das operações sobre os arquivos de controle dependem diretamente do desempenho do NFS. Para avaliar o *overhead*, inserido pelo NFS na implementação, foram medidos e comparados os tempos necessários para colocar um

pedido na fila de impressão, considerando dois casos: a) pedidos de impressão em uma máquina que possui fisicamente os arquivos de controle; b) pedidos de impressão em uma máquina com os de arquivos de controle, montados com NFS. O tamanho do arquivo usado nos testes foi de 20k *bytes* e as impressoras requisitadas foram sempre locais, para garantir que nenhuma RPC fosse realizada. O tempo médio obtido na máquina **sem** arquivos montados foi de 0,58 segundos. O tempo médio obtido na máquina **com** os arquivos montados foi de 0,77 segundos. Pode-se observar que as operações sobre arquivos remotos, montados com NFS, têm desempenho inferior se comparadas às operações realizadas sobre arquivos locais. Apesar disso, de acordo com os resultados obtidos, o uso do NFS não compromete o desempenho da implementação.

Nos casos em que o desempenho do NFS não é satisfatório, o programador tem como alternativa otimizar alguns parâmetros de configuração do NFS, para tentar obter melhoria de desempenho. O NFS oferece ferramentas que tornam possível identificar seus pontos críticos de desempenho. Com essas ferramentas é possível obter informações, para configurar os parâmetros da melhor forma possível. Técnicas para otimização de desempenho do NFS podem ser encontrados em [Stern91].

- **Desempenho das Operações com RPC**: o uso de UDP torna as RPCs da implementação bastante rápidas. Para avaliar o desempenho de pedidos de impressão que usam RPCs foram considerados dois casos: a) pedidos de impressão para impressoras locais (não utilizam RPCs); b) pedidos de impressão para impressoras remotas (utilizam RPCs). O tamanho do arquivo usado nos testes foi de 20k *bytes*. As medições foram realizadas em impressoras de uma máquina sem arquivos de controle montados, para garantir que nenhuma operação via NFS fosse realizada. O tempo médio obtido em pedidos de impressão **sem** o uso de RPC foi de 0,58 segundos e o tempo médio obtido em pedidos **com** uso de RPC foi de 0,68 segundos.

Em função dos resultados obtidos, não foi necessário usar servidores concorrentes e nem técnicas de *cache* para melhorar o desempenho das RPCs. Isso é justificado pelo fato do *svd* oferecer um número reduzido de serviços (apenas três), dois dos quais utilizados somente durante

a alocação de impressoras remotas. O terceiro serviço, o de sinalização, é o único usado com maior frequência, mas é extremamente simples e rápido. Portanto, é desnecessário criar *threads*, para processar paralelamente os serviços, visto que não há demanda para isso.

• **Desempenho e Escalabilidade**: o principal fator de degradação do desempenho desta implementação é consequência do problema de escalabilidade, descrito na Seção 6.4.5.5. O desempenho da versão dependente de DFS só é satisfatório, se o número de *spoolers* ativos não tende a crescer muito. Nos testes realizados, a existência de 8 *spoolers* ativos em diferentes máquinas é suficiente para comprometer o tempo de resposta de alguns comandos. Por exemplo, o tempo de resposta do comando ***imp,*** supera 4 segundos apenas para colocar um pedido de 20k *bytes* na fila de impressão de uma máquina remota (**Fig. 6-6**).

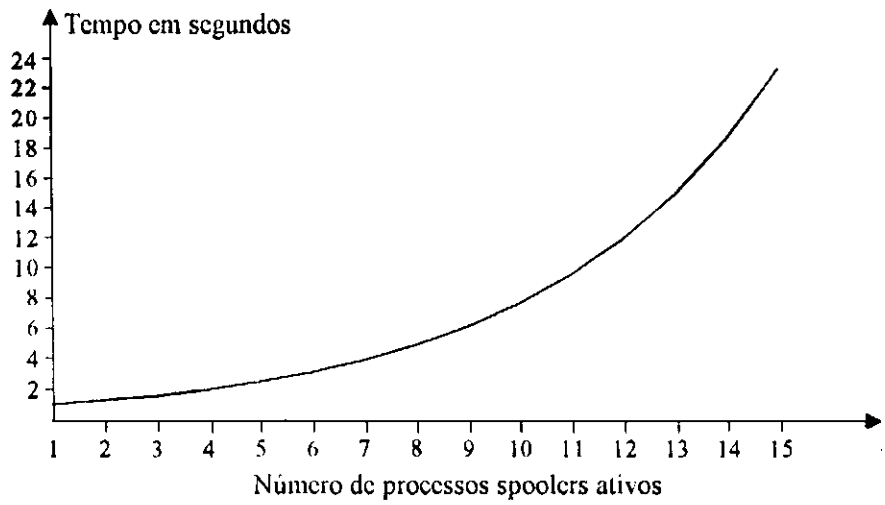


Figura 6-6. *Desempenho de pedidos de impressão versus spoolers ativos*

6.4.5.9. Funcionalidade

A migração de uma aplicação centralizada, para um ambiente distribuído, pode implicar em perda de funcionalidade do projeto original. Alguns requisitos funcionais coerentes em ambientes centralizados passam a não fazer sentido em um ambiente distribuído. Um exemplo é a facilidade oferecida pelo *spoolview* de imprimir arquivos na impressora *qualquer*. Em um ambiente centralizado, essa facilidade faz sentido, porque normalmente as impressoras estão

fisicamente próximas. Em um ACD no entanto, é possível ter impressoras que podem estar fisicamente separadas por vários quilômetros. O problema é: o que fazer se um pedido submetido a impressora *qualquer* for impresso a dezenas de quilômetros do usuário? Nesse caso, a funcionalidade da impressora *qualquer* perde o sentido. Para minimizar esse problema, o algoritmo de escolha da impressora *qualquer* deveria ser modificado, para selecionar impressoras pertencentes somente ao **domínio** do usuário que solicita o pedido de impressão. Apesar disso, nesta implementação, o algoritmo original do *spoolview* para a escolha da impressora *qualquer* não foi alterado. Portanto, um pedido para a impressora *qualquer* está sujeito a ser impresso em qualquer uma das impressoras alocadas (próxima ou distante do usuário que solicitou o pedido).

6.5. Implementação Independente de DFS

Nesta seção, será descrita a segunda implementação do serviço de impressão distribuído, baseado no *spoolview*. Inicialmente, será dada uma visão global das entidades que compõem essa implementação, bem como os problemas resolvidos por cada serviço da plataforma ONC. Finalmente, serão apresentadas considerações específicas, incluindo um comparativo com as considerações específicas da primeira implementação.

6.5.1. Visão Geral da Implementação

A implementação independente de DFS resolve os principais problemas e limitações apresentados pela primeira versão implementada. A principal diferença entre as duas implementações é o **grau** de descentralização dos seus componentes.

Na primeira implementação, apenas os gerenciadores de impressão *spoolers* e os *svds* estão distribuídos em cada máquina. Já os arquivos de controle residem fisicamente em uma única máquina e são compartilhados por todos os usuários.

Na segunda implementação, além dos *spoolers* e dos *svds*, os arquivos de controle também estão distribuídos. Isso significa que cada máquina, usuária do serviço de impressão,

possui o seu próprio conjunto de arquivos de controle. Essa implementação não faz uso de um DFS. Foram utilizados somente os serviços de RPC e XDR. A **Fig. 6-7** mostra uma visão global das entidades e dos serviços que compõem a segunda implementação.

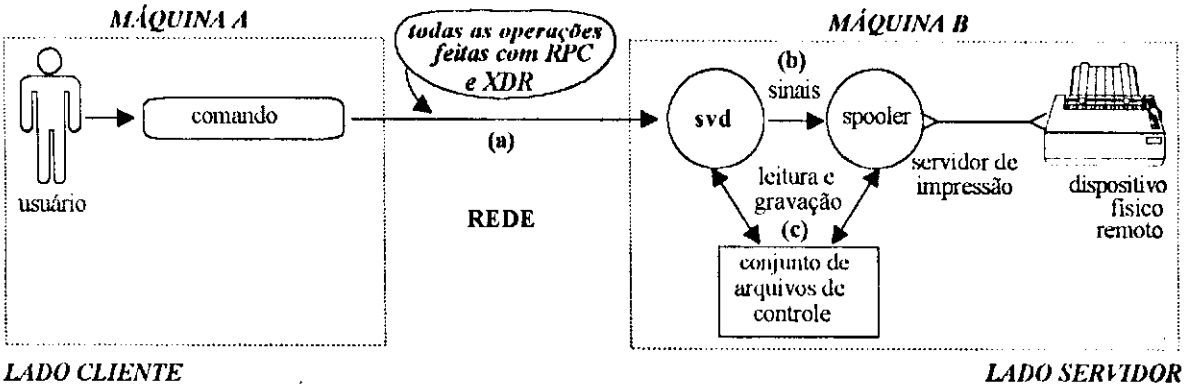


Figura 6-7. *Visão global das entidades da versão independente de DFS*

A seguir serão descritos os problemas resolvidos com RPC e XDR.

6.5.2. Problemas resolvidos com RPC

Nessa implementação, todas as operações remotas, envolvendo os arquivos de controle e os gerenciadores de impressão *spoolers*, são feitas através do serviço de RPC (**Fig. 6-7a**). A vantagem de usar RPCs, para realizar o trabalho antes feito pelo NFS, é que o programador ganha flexibilidade para definir procedimentos remotos mais adequados aos requisitos da aplicação. Uma desvantagem de usar RPC é o aumento considerável do trabalho de programação. Tanto o componente do lado servidor quanto do lado cliente da implementação sofreram substanciais acréscimos de código para dar suporte às operações anteriormente feitas pelo NFS.

6.5.2.1. Componente do Lado Servidor

O *svd* da segunda implementação é responsável pelas operações de *spoolers*, pelas operações sobre os arquivos de controle e por algumas outras operações especiais.

Os serviços usados para a execução das operações de *spoolers* são os mesmos da primeira implementação (**Fig. 6-7b**). As operações remotas de leitura e gravação sobre os arquivos de controle são representadas por novos serviços acrescentados ao *svd* (**Fig. 6-7c**). Além disso, também foram adicionados ao *svd* alguns serviços especiais, como por exemplo, para a seleção de impressoras *qualquer* e para a cópia remota de arquivos.

Os novos serviços do *svd* podem ser divididos em cinco grupos: manutenção de impressoras, gerenciamento de impressão, manutenção de pedidos, manutenção de arquivos temporários e seleção da impressora *qualquer*.

- **Manutenção de Impressoras**: permite alocar, desalocar e obter atributos de impressoras remotas.

- **Gerenciamento de Impressão**: permite suspender, prosseguir e cancelar uma impressão. Nesse grupo, existe também um serviço que permite definir a página a partir da qual uma impressão suspensa deve ser retomada.

- **Manutenção de Pedidos**: permite realizar operações sobre a fila de impressão. Através desse grupo de serviços é possível gravar novos pedidos ou alterar, remover, obter atributos e mudar a prioridade dos pedidos de impressão enfileirados.

- **Manutenção de Arquivos Temporários**: permite criar, verificar o acesso e eliminar os arquivos temporários *cf* e *df*. Existem serviços específicos, nesse grupo, para copiar arquivos da máquina do cliente para a máquina do servidor, e também para eliminar arquivos temporários "orfãos" em caso de falha durante o processo de cópia.

- **Seleção da Impressora Qualquer**: torna possível selecionar uma impressora *qualquer* independentemente de máquina. Como agora cada máquina possui seu próprio conjunto de arquivos de controle, são necessários serviços para selecionar uma impressora *qualquer*, considerando as impressoras e filas de impressão de todas as máquinas, usuárias do serviço de impressão.

6.5.2.2. Componente do Lado Cliente

O componente do lado cliente é constituído por uma biblioteca projetada da mesma forma que a biblioteca usada no componente cliente da primeira implementação. A diferença fundamental é que, com a distribuição dos arquivos de controle, todas as funções que envolvem operações sobre os arquivos controle foram adicionadas a essa biblioteca.

6.5.3. Problemas resolvidos com XDR

O serviço XDR é usado para tratar os problemas de heterogeneidade de forma análoga à primeira implementação. A única diferença é que foram programadas novas rotinas XDR de conversão e desconversão, para os novos serviços adicionados ao *svd*.

6.5.4. Considerações Específicas sobre a Implementação e Comparativo

Nesta seção, serão apresentadas considerações específicas sobre os requisitos da versão independente de DFS. Os requisitos abordados serão comparados, quando possível, com as considerações correspondentes da primeira implementação, permitindo uma avaliação conjunta das soluções empregadas por ambas implementações. Os requisitos analisados são: semântica de compartilhamento, endereçamento de arquivos, transporte, transparência, escalabilidade, disponibilidade, tolerância a falhas, segurança, desempenho e funcionalidade.

6.5.4.1. Semântica de Compartilhamento de Arquivos

Na primeira implementação foram observados problemas decorrentes dos efeitos da semântica de compartilhamento de arquivos do NFS. A versão independente de DFS não apresenta esse tipo de problema por não usar o NFS e adotar uma filosofia de distribuição dos arquivos de controle. Cada máquina possui o seu conjunto individual de arquivos de controle e, portanto, o compartilhamento é feito somente entre processos *spoolers* que estão na mesma máquina. Consequentemente, a semântica de compartilhamento imposta aos processos *spoolers* é a de um sistema de arquivos local, adequada aos requisitos da aplicação.

6.5.4.2. Endereçamento de Arquivos Remotos

A solução adotada pela segunda implementação, para resolver o problema de endereçamento de arquivos remotos, a princípio, é a mesma usada na primeira implementação. Toda vez que um pedido de impressão, para uma impressora remota, é solicitado, é criada uma cópia do arquivo a ser impresso na máquina remota. Apesar disso, existem algumas diferenças na forma como os arquivos temporários de cópia são criados e na maneira como o problema de conflito de nomes dos arquivos de cópia foi resolvido.

• **Criação do Arquivo de Cópia**: na primeira implementação, os arquivos de cópia são criados remotamente, através do NFS: as cópias são criadas no diretório montado dos arquivos de controle. A segunda implementação usa RPCs, para criar os arquivos de cópia na máquina onde reside a impressora solicitada pelo usuário. Para criar as cópias remotamente, são utilizados o grupo de serviços de manutenção de arquivos temporários descritos na Seção 6.5.2.1.

• **Conflito de Nomes dos Arquivos de Cópia**: a segunda implementação não apresenta o problema de conflito de nomes dos arquivos temporários de cópias, detectado na primeira versão. O fato de cada máquina possuir o seu próprio conjunto de arquivos de controle, torna desnecessário o uso de uma função, para criar arquivos de cópia com nomes distintos, independentemente de máquina. É suficiente utilizar uma função que garanta nomes diferentes de arquivos temporários em uma mesma máquina, como por exemplo, o *system call* padrão de ambientes UNIX *mktemp()*.

6.5.4.3. Transporte

A segunda implementação utiliza dois tipos de serviços de transporte: UDP e TCP. A necessidade da utilização de dois serviços de transporte está diretamente relacionada com os requisitos de semântica das RPCs, idempotência, volume de dados e confiabilidade, exigidos pela segunda implementação.

• **Semântica de Chamada e Idempotência**: ao contrário da primeira implementação, cujos procedimentos remotos são todos idempotentes, a segunda implementação possui um número considerável de procedimentos não idempotentes. Quase todos os novos procedimentos adicionados ao *svd*, para substituir o NFS, são não idempotentes. As chamadas a procedimentos não idempotentes adotam TCP para obter a semântica de chamada de no-máximo-uma-vez, exigida por esse tipo de procedimento. Um exemplo são as RPCs usadas para copiar arquivos remotamente. Nesse caso, a confiabilidade do TCP garante que os blocos de dados copiados sejam gravados na ordem correta. Por outro lado, as RPCs feitas a procedimentos remotos idempotentes, como por exemplo, a leitura de dados dos arquivos de controle, utilizam UDP. As razões para o uso de UDP são as mesmas já descritas na primeira implementação.

• **Volume de dados dos parâmetros das RPCs**: os procedimentos remotos idempotentes da segunda implementação que necessitam passar parâmetros com tamanho superior ao limite de 8k *bytes*, imposto pelo UDP da Sun, também usam TCP. Uma conexão TCP resolve melhor o problema, por não impor limite de tamanho para os parâmetros da RPCs.

• **Confiabilidade**: um ganho obtido com a confiabilidade de TCP é que torna-se desnecessário detectar e tratar erros de perda, duplicação e troca de ordem das mensagens que contém as RPCs, a nível de aplicação.

6.5.4.4. Transparência

Existem diferenças substanciais na forma como o requisito de transparência de localização de impressoras foi atendido por cada implementação. A primeira implementação impõe um espaço único de nomes de impressoras. A segunda implementação adota múltiplos espaços de nomes de impressoras, associado à técnica de apelidos (*alias*).

• **Múltiplos Espaços de Nomes de Impressoras**: na segunda implementação, cada máquina usuária do *spoolview* possui um espaço de nomes de impressoras particular. Esse esquema é mais adequado do que o adotado pela primeira implementação, porque torna possível a

alocação de impressoras de mesmo nome por máquinas diferentes.

Uma das implicações da existência de múltiplos espaços de endereçamento de impressoras é a necessidade de especificar o nome da máquina, onde reside a impressora remota que se deseja utilizar. Os comandos da segunda implementação foram modificados para suportarem uma opção (*-m*) que permite a especificação do nome da máquina. Por exemplo, o comando

imp -llaser -mcctdsc.ufpb notas.txt

está solicitando a impressão do arquivo *notas.txt* na impressora *laser* da máquina *cctdsc*, no domínio *ufpb*.

• **Transparência com Uso de Apelidos**: a necessidade da especificação do nome da máquina (opçãp *-m*) acaba comprometendo o requisito de transparência de localização de impressoras. Para resolver esse problema, a segunda implementação adota uma técnica de apelidos. Essa técnica consiste em associar a cada tripla (impressora, máquina, domínio) um apelido que representa o nome lógico da impressora. Por exemplo, a impressora *laser* da máquina *cctdsc.ufpb* poderia ser associada ao apelido *lp1*. Para imprimir na impressora *laser* seria necessário apenas informar o seu apelido: *imp -ilp1 notas.txt*. O comando ***imp*** consultaria um arquivo de apelidos e faria o mapeamento para o nome, máquina e domínio associados ao apelido *lp1*, de forma totalmente transparente para o usuário. Tanto o usuário quanto o administrador do sistema podem definir apelidos, para as impressoras alocadas remotamente. Por motivos de ortogonalidade, também é possível dar apelidos a impressoras locais.

6.5.4.5. Escalabilidade

A primeira implementação apresenta problemas de escalabilidade devido à centralização dos arquivos de controle e à política de travamento exclusivo de arquivos.

Na segunda implementação, o fato dos arquivos de controle se encontrarem distribuídos reduz substancialmente o comprometimento da escalabilidade. Os arquivos de controle somente são compartilhados e atualizados, por processos da máquina onde residem. Não existem mais os

problemas observados na primeira implementação, porque o travamento dos arquivos passou a ser feito apenas localmente em cada máquina. Portanto, o travamento exclusivo dos arquivos de controle de uma máquina não afeta mais a disponibilidade dos arquivos de controle nas outras máquinas.

Na segunda implementação, o tempo médio de atualização de cada conjunto de arquivos de controle depende principalmente da frequência de atualizações dos *spoolers* existentes em cada máquina. O processo de travamento é bem mais rápido, visto que não existe mais o *overhead* do NLM. Se o número de *spoolers* crescer rapidamente, em uma determinada máquina, somente os processos dessa máquina serão afetados. Nesse caso, o número de *spoolers* necessários, para comprometer a disponibilidade dos arquivos de controle é bem maior, se comparado com o da primeira implementação. A **Fig. 6-9** da Seção 6.5.4.8 apresenta um gráfico que demonstra a melhor escalabilidade da segunda implementação sobre a primeira quando o número de *spoolers* tende a crescer.

6.5.4.6. Disponibilidade e Tolerância a Falhas

A segunda implementação resolve os principais problemas de disponibilidade e tolerância a falhas apresentados pela primeira implementação. Os pontos mais importantes a serem considerados são: distribuição dos arquivos de controle, política de travamento de arquivos, reinicialização de servidores e o esquema de tratamento de erros.

• **Distribuição dos Arquivos de Controle**: um exemplo de como explorar o potencial inerente de tolerância a falhas em ADs, é o esquema de distribuição dos arquivos de controle da segunda implementação. Em caso de falha de qualquer máquina que contenha os arquivos de controle, o processamento de impressão, nas outras máquinas, prossegue normalmente como se nada tivesse acontecido. Somente os usuários da máquina que falhou são penalizados. Na primeira implementação, basta falhar a máquina onde residem fisicamente o conjunto de arquivos de controle, para que o processamento de impressão seja paralisado em todas as máquinas.

• **Política de Travamento dos Arquivos de Controle**: como visto anteriormente, a primeira implementação adota um esquema do lado cliente do tipo "**trava()...call RPC...destrava()**" que, em caso de falha, mantém os arquivos de controle travados até à detecção do *timeout*. A segunda implementação resolveu esse problema, transferindo para a componente do lado servidor, todo o esquema de travamento (**Fig. 6-8**). Vários procedimentos do *svd* passaram a ser responsáveis pelo travamento dos arquivos de controle. Todo o travamento passa a ser feito localmente na máquina onde o procedimento remoto é executado. Do ponto de vista do lado cliente, a vantagem do esquema da figura **Fig. 6-8b** sobre o da **Fig. 6-8a** é que, se após a RPC o servidor falhar, antes de concluir a execução do procedimento remoto, o cliente não mantém os arquivos de controle travados durante o tempo do *timeout*.

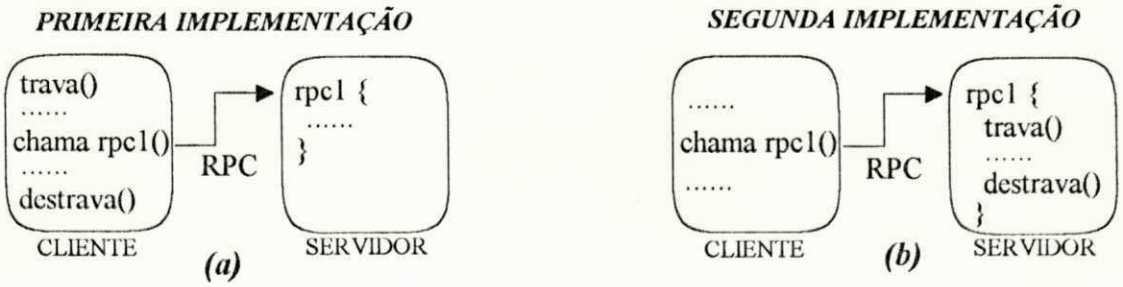


Figura 6-8. *Comparativo do esquema de travamento das implementações*

• **Reinicialização de Servidores**: a solução usada é a mesma da primeira implementação. Os servidores *svd* são reinicializados através do serviço *inetd*.

• **Tratamento de Erros**: existem várias diferenças entre o esquema de tratamento de erros das duas implementações. Ao contrário da primeira implementação onde apenas erros fatais ocorrem durante as RPCs, na segunda implementação, além dos erros fatais, podem também ocorrer erros de advertência. Isso torna o tratamento de erros da segunda implementação mais complexo. A dificuldade para tratar os erros pode ser visualizada com o exemplo de uma RPC, cuja execução resulta na sequência de vários erros de advertência antes de um erro fatal. Se, à medida que os erros de advertência forem sendo detectados pelo servidor, as mensagens de erro

forem emitidas, então elas irão aparecer na *console* da máquina do servidor e não na *console* da máquina do cliente, como seria o correto. Uma solução para resolver esse problema poderia ser o servidor fazer uma RPC para a máquina do cliente a cada erro detectado, a fim de exibir a mensagem de erro na *console* correta. Essa não é uma boa solução, porque implica no aumento do número de RPCs.

Uma outra solução, a adotada pela segunda implementação, utiliza uma estrutura de dados tipo lista, para armazenar e recuperar os erros ocorridos no lado servidor. De forma simplificada, esse esquema funciona da seguinte forma: a) o cliente faz uma RPC para o servidor, passando como parâmetro uma estrutura de dados tipo lista, denominada **lista de erros**; b) o servidor executa o serviço solicitado e, à medida que detecta erros de advertência, armazena-os na lista de erros. Os códigos de erro são armazenados na lista enquanto não ocorrer um erro fatal ou até que a execução do procedimento seja concluída com sucesso. Se ocorrer um erro fatal, o servidor interrompe imediatamente a execução do procedimento e armazena o código do erro na lista de erros; c) a lista de erros é retornada como resultado para o cliente que fez a RPC; d) o cliente recebe a lista de erros e verifica se ela contém algum código de erro. Caso afirmativo, os códigos de erros são convertidos para mensagens e exibidos na *console* da máquina do cliente, na ordem em que ocorreram no lado servidor.

6.5.4.7. Segurança

A principal diferença entre os esquemas de segurança das duas implementações é o nível de complexidade. A primeira implementação tem um esquema de segurança mais complexo, porque envolve cuidados tanto a nível de NFS quanto a nível de RPC. A segunda, por outro lado, é menos complexa, pois requer cuidados somente a nível de RPC. A segunda implementação utiliza *secure* RPC, para garantir a segurança das operações remotas de forma análoga à primeira implementação.

6.5.4.8. Desempenho

Nesta seção, serão descritas as técnicas usadas para melhorar o desempenho da segunda implementação, além de um comparativo com a primeira implementação. Os pontos analisados serão: transporte das RPCs, chamadas remotas em lote e distribuição dos arquivos de controle.

• **Transporte das RPCs**: o uso conjunto de UDP e TCP pode ser visto como uma estratégia para maximizar o desempenho da segunda implementação. Seria possível optar pelo uso somente do TCP, mas o uso de UDP por algumas RPCs elimina parte do *overhead* que seria, naturalmente, imposto somente pelo uso de TCP. Na segunda implementação, o UDP é usado onde é possível, visando obter o máximo desempenho das RPCs. Já o TCP é utilizado somente onde a confiabilidade é imprescindível.

• **Chamadas Remotas em Lote**: algumas operações com RPCs, realizadas pela segunda implementação, tiveram o seu desempenho melhorado com o uso de chamadas remotas em lote. Por exemplo, para copiar um arquivo da máquina do cliente para a máquina do servidor, é utilizada uma RPC para gravar os blocos de dados do arquivo no lado servidor. Dependendo do número de blocos a serem copiados, a RPC de gravação é executada um número considerável de vezes. Em casos como esse, a segunda implementação usa RPC *Batch*. Nos testes efetuados, o ganho percentual de desempenho das operações de cópia remota de arquivos, com RPC *Batch*, foi em torno de 8%. Contudo, existem relatos na bibliografia consultada (veja [Corbin91]) que registram ganhos de desempenho de até 15% .

• **Distribuição dos Arquivos de Controle**: na primeira implementação, apenas um processo por vez, independentemente de máquina, tem acesso aos arquivos de controle. Dependendo do número de processos ativos, isso pode penalizar o desempenho. Na segunda implementação, processos de máquinas diferentes acessam paralelamente os seus arquivos de controle. Esse paralelismo garante um bom desempenho das operações realizadas pelos processos da segunda implementação, sobre os arquivos de controle, independentemente do número de *spoolers* ativos (**Fig. 6-9**).

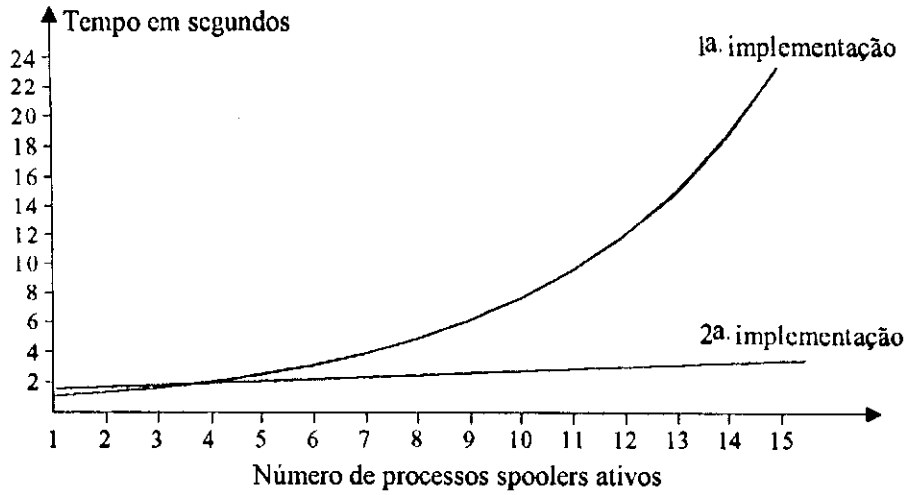


Figura 6-9. *Desempenho dos pedidos de impressão das implementações*

Uma análise da **Fig. 6-9** mostra que, até 4 *spoolers* ativos, o tempo de resposta da segunda implementação é ligeiramente maior do que tempo de resposta da segunda implementação. Isso ocorre, basicamente, porque a segunda implementação faz uso de conexões TCP, enquanto que a primeira implementação utiliza somente UDP (mesmo nas operações via NFS).

6.5.4.9. Funcionalidade

As duas implementações mantêm integralmente a funcionalidade original do *spoolview*. A principal diferença entre elas é a forma como as soluções adotadas foram implementadas. Por exemplo, na segunda implementação, existem novos complicadores para manter a funcionalidade original do *spoolview*, devido à descentralização dos arquivos de controle. Para obter uma lista de todas as impressoras alocadas (locais e remotas), comandos como o ***mos***, ***sta*** e pedidos de impressão para a impressora *qualquer*, necessitam consultar os arquivos de controle existentes em cada máquina. A segunda implementação resolve esses problemas, utilizando a facilidade **RPC** ***broadcast***. Dois exemplos da necessidade de uso dessa facilidade são: localização dos arquivos de controle e seleção da impressora *qualquer*.

• **Localização dos Arquivos de Controle**: para executar corretamente os comandos *sta* e *mos*, foi necessário adicionar a cada um deles uma opção para determinar se o escopo de ação é local ou global. Um escopo local significa que o comando deve considerar somente as impressoras existentes na máquina local. Um escopo global significa que os comandos devem considerar todas as impressoras de todas as máquinas. Quando o escopo de ação do comando é global, a maior dificuldade é como localizar as máquinas onde residem os arquivos de controle. A solução adotada foi a implementação de uma função (*acha_svd()*) que emite uma RPC *broadcast*, para obter a lista das máquinas que possuem arquivos de controle a serem consultados.

• **Seleção de Impressora Qualquer**: outro problema, solucionado com RPC *broadcast*, foi a seleção da impressora *qualquer* quando o escopo de ação comando *imp* é global. Como eleger, dentre as impressoras existentes em várias máquinas, a mais adequada para assumir o papel da *qualquer*? Esse problema foi solucionado com a implementação de uma função (*acha_qualquer()*) que retorna, como resultado, o nome da impressora *qualquer* mais adequada e a máquina onde ela reside. Ao ser executada, a função *acha_qualquer()* faz uma RPC *Broascast*, passando como argumento o tamanho do arquivo a ser impresso. Cada máquina que recebe a mensagem de *broadcast* executa, localmente, um procedimento específico que verifica que impressora está apta a atender mais rapidamente o pedido, considerando o tamanho do arquivo a ser impresso. A impressora selecionada é a candidata a *qualquer* da máquina em questão. Em seguida, cada máquina retorna, para o cliente que emitiu o *broadcast*, o nome da sua impressora candidata e o tempo estimado para o término de impressão do pedido. A função *acha_qualquer()* recebe as candidatas a *qualquer* e seleciona como vencedora a que tiver o menor tempo previsto para o término da impressão. O nome e a máquina da impressora vencedora são retornados como resultado pela função *acha_qualquer()*. Finalmente, o pedido de impressão é direcionado para a impressora *qualquer* eleita.

No próximo capítulo será feita uma avaliação dos objetivos deste trabalho e, em seguida, apresentadas algumas conclusões e sugestões para trabalhos futuros.

7. Conclusão

As Aplicações Distribuídas (ADs) são uma alternativa para o uso efetivo do potencial computacional, disponível nos sistemas de redes heterogêneas. Através das ADs, os usuários podem obter um maior retorno dos investimentos realizados em *hardware* e *software* e, consequentemente, uma melhor relação custo/benefício. Pelas vantagens que oferecem, as AD são uma forte tendência para a computação dos anos 90. Portanto, é importante dominar suas técnicas de projeto e implementação.

O maior problema enfrentado pelos programadores, atualmente, é que não existe um consenso sobre qual a metodologia mais adequada para desenvolver ADs. Esse problema tende a ser resolvido à medida que as pesquisas evoluírem, mas, por enquanto, uma forma de minimizá-lo é fornecer ao programador informações que o auxiliem durante o processo de desenvolvimento de ADs.

Nos capítulos anteriores, foram apresentadas, com razoável nível de detalhes, informações importantes para o desenvolvimento de ADs. Foram abordadas a infra-estrutura, os requisitos básicos, os modelos e várias considerações específicas necessárias para o desenvolvimento de ADs, incluíndo a análise de duas implementações.

A principal contribuição deste trabalho é a formulação de considerações práticas para o desenvolvimento de ADs eficientes e confiáveis. Mais especificamente, fez-se um esforço no sentido de destacar os principais pontos que o programador deve ter em mente ao desenvolver ADs. A seguir será feita uma avaliação dos objetivos propostos na introdução deste trabalho, bem como algumas conclusões e sugestões para trabalhos futuros.

7.1. Avaliação dos Objetivos do Trabalho

Na Seção 1.2, foram formulados os objetivos desse trabalho. Nesta seção, será discutido até que ponto esses objetivos foram atingidos.

a) Caracterização da Infra-estrutura para o Desenvolvimento de ADs

Através da apresentação da arquitetura típica de um Ambiente de Computação Distribuída heterogênea (ACD) e da discussão dos seus principais serviços, o programador tem uma visão geral da infra-estrutura que viabiliza o desenvolvimento de ADs heterogêneas.

Apesar de atualmente existirem vários padrões de ACD (ex.: ONC, OSF e OSI), acreditamos que os serviços apresentados são genéricos o suficiente, para que o programador entenda os seus fundamentos, independentemente do padrão de ACD que ele venha a adotar.

b) Determinação dos Requisitos Básicos para Desenvolver ADs

O desenvolvimento de ADs envolve múltiplas dimensões de complexidade. Se comparada ao desenvolvimento de aplicações centralizadas, o programador deve ter em mente um número maior de requisitos, devido, principalmente, à existência de dados, controle e recursos de *hardware* e *software* distribuídos pela rede.

Os requisitos básicos sugeridos por este trabalho foram transparência, heterogeneidade, disponibilidade, tolerância a falhas, segurança, escalabilidade e desempenho. Do nosso ponto de vista, esses requisitos cobrem satisfatoriamente aspectos que, se não observados pelo programador, podem introduzir uma séria queda na qualidade da AD.

c) Identificação e Avaliação dos Modelos de Programação de ADs

Um problema a ser resolvido no desenvolvimento de ADs, é encontrar modelos de programação adequados para a implementação de soluções distribuídas. Neste trabalho, foram apresentados três modelos, baseados no modelo cliente-servidor, que oferecem diferentes níveis de abstração: troca de mensagens, chamada de procedimento remoto e invocação de objetos remotos.

O modelo de troca de mensagens oferece primitivas (ex.: *send* e *receive*), que exigem que o programador trate explicitamente o envio e o recebimento de mensagens. No modelo de RPC a complexidade do envio e recebimento das mensagens é escondida do programador, mas alguns detalhes do sistema de comunicação ainda são visíveis, embora o sejam em menor grau do que no modelo de mensagens. Já no modelo de invocação de objetos remotos, todos os detalhes de comunicação e localização são transparentes para o programador.

Nós acreditamos que os modelos apresentados cobrem um amplo espectro de necessidades, em função dos diferentes níveis de abstração que oferecem. O programador pode assim avaliar com segurança o modelo mais adequado para o desenvolvimento de uma determinada AD.

d) Proposição de Considerações Específicas para o Desenvolvimento de ADs

As considerações específicas para o desenvolvimento de ADs, apresentadas neste trabalho, envolvem aspectos de serviço de transporte, tratamento e recuperação de erros, desempenho e segurança. Essas considerações estão diretamente relacionadas com a confiabilidade e a eficiência da AD a ser desenvolvida. Poderiam ter sidos abordados outros aspectos, mas isso restringiria a discussão a casos muitos particulares. As considerações propostas são comuns a uma ampla classe de aplicações, e portanto mais adequadas ao que esse trabalho se propõe.

Apesar das considerações apresentadas terem sido baseadas no modelo de programação de RPC, elas também são úteis como referência para o desenvolvimento de ADs baseadas em outros modelos de programação, visto que a cada abordagem procurou-se dar um enfoque genérico.

e) Avaliação das Considerações Propostas a partir de um Estudo de Caso

A partir da implementações pôde-se constatar que:

- Transparência é um dos requisitos mais importantes de uma AD, porque esconde a complexidade do ACD dos usuários.
- O desempenho de uma AD é dependente de um conjunto de fatores, principalmente, relacionados com a infra-estrutura de comunicação, cujo controle nem sempre está ao alcance do programador.
- Deve-se ter atenção especial com a semântica das RPCs e com a não idempotência de procedimentos remotos. O tipo de serviço de transporte pode ter influência direta no tipo de semântica das RPCs, dependendo da implementação do mecanismo RPC.
- O uso de serviços de *broadscast* pode limitar a escalabilidade da AD.
- O uso de DFS em projetos de ADs requer atenção especial com problemas de semântica de compartilhamento, quando a aplicação possui arquivos compartilhados por múltiplos processos. Nem sempre a semântica de compartilhamento oferecida pelo DFS atende aos requisitos da aplicação.
- O uso de DFS em projetos de ADs pode reduzir significamente o trabalho de programação, nos casos em que suas limitações não comprometem a integridade da AD.
- Parte significativa do código de ADs destina-se ao tratamento e recuperação de erros, principalmente, nos casos em que a aplicação possui requisitos de tolerância a falhas.
- Requisitos de funcionalidade de uma aplicação que têm sentido em ambientes centralizados nem sempre fazem sentido em ambientes distribuídos.

7.2. Conclusões Finais e Trabalhos Futuros

O julgamento final sobre a qualidade das considerações propostas, neste trabalho, poderá ser feito por programadores que as utilizem em seus projetos de desenvolvimento de ADs. Contudo, nós acreditamos que os objetivos especificados no Capítulo 1 foram atingidos. As

considerações formuladas são, na grande maioria, de ordem prática e úteis para várias classes de aplicações.

Tendo como base os resultados desse trabalho, surgem inúmeras alternativas para o desenvolvimento de novas pesquisas que viriam a ampliar este estudo. Algumas sugestões que contribuiriam para atingir essa meta são:

- Formulação de uma metodologia para o desenvolvimento de ADs.
- Considerações específicas de portabilidade na implementação de ADs.
- Considerações específicas sobre testes e depuração de ADs.
- Considerações específicas sobre *benchmarks* de ADs.
- Considerações específicas sobre ADs que utilizam serviços de janela distribuído.
- Considerações sobre ADs desenvolvidas com serviços de transações distribuídas.

Bibliografia

Referências Citadas

[Bal90] Bal, H. E. *Programming Distributed Systems*, Prentice-Hall, 1990. Citado nas páginas 61, 65, 68 e 69.

[Birrell84] Birrell, Andrew D. & Nelson, Bruce Jay. *Implementing Remote Procedure Calls*, ACM Transaction on Computer Systems, Vol. 2, no. 1, Feb. 1984, pp. 39-59. Citado nas páginas 31, 32, 35, 39.

[Bloomer91] Bloomer, J. *Power Programming with RPC*, O'Reilly & Associates, Inc. 1991. Citado nas páginas 3, 6 e 72.

[Carter91] Carter, John B. & Bennett John K. & Zwaenepoela. *Implementation and Performance of Munin*, ACM Operating System Review, 1991, pp. 152-164. Citado na página 36.

[Chin91] Chin, Roger S. & Chanson, Samuel T. *Distributed Object-Based Programming Systems*, ACM Computing Surveys, vol. 23, no. 1, Mar. 1991, pp. 91-124. Citado na página 44.

[Clark91] Security Study Committee (David D. Clark, Chairman), National Research Council. *Computers at Risk: Safe Computing In the Information Age*, National Academy Press, 1991. Citado na página 76.

[Commer91] Commer, D. E. & Stevens D. L. *Internetworking with TCP/IP - Design, Implementation, and Internals*, volume II, Prentice Hall, 1991. Citado nas páginas 10 e 79.

[Corbin91] Corbin, J. R. *The Art of Distributed Applications - Programming Techniques for Remote Procedure Calls*, Springer-Verlag, 1991. Citado nas páginas 6, 18, 27, 58, 59, 72, 74 e 122.

[Coulouris88] Coulouris, George F. & Dollimore, Jean. *Distributed Systems: Concepts and Design*, Addison-Wesley. 1988. Citado nas páginas 17, 36, 60, 70 e 84.

[Cypser91] Cypser, R. J. *Communications for Cooperating Systems - OSI, SNA, and TCP/IP*, Addison-Wesley, 1991. Citado nas páginas 10 e 16.

[Diffie76] Diffie, W. & Hellman, M. E. *New Directions in Cryptography*, IEEE Transactions on Information Theory, IT-22, 1976, pp. 644-654. Citado na página 80.

[Dunphy91] Dunphy, Ed. *The UNIX Industry: Evolutions, Concepts, Architecture, Applications, and Standards*, QED Technical Publishing Group, 1991. Citado na página 14.

[Fidge91] Fidge, Colin. *Logical Time in Distributed Computing Systems*, IEEE Computer, Aug. 1991, pp. 28-33. Citado na página 13.

[Gray91] Gray, Pamela. A. *Open Systems: A Business Strategy for the 1990s*, McGraw-Hill, 1991. Citado na página 14.

[Hewlett91] Hewlett-Packard Company & Sun Microsystems, Inc. *Object Request Broker - RFP Joint Response*, OMG Document Number: 91.1.4.8, Part No.: 800-6274-01, 1991, pp. 1-29. Citado nas páginas 42 e 45.

[Infocon91] Infocon. *Manual do Spoolview*, Ref. 071989ED01, Editado pela Infocon, 1991. Citado nas páginas 87, 88 e 91.

[Joseph89] Joseph, T. A. & Birman K. P. *Reliable Broadcast Protocols* In: Mullender, S.(ed.): Distributed Systems, Reading (Mass.). Addison-Wesley, 1989. pp.65-86. Citado na página 62.

[Khoshafian92] Khoshafian, Setrag & Chan, Arvola & Wong, Anna & Wong, Harry K. T. *A Guide to Developing Cliente/Server SQL Applications*, Morgan Kaufmann Publishers, 1992. Citado na página 15.

[Levy90] Levy, Eliezer & Silberschatz, Abraham. *Distributed File Systems: Concepts and Examples*, ACM Computing Surveys, Dec. 1990, Vol. 22, No. 4, pp. 322-374. Citado na página 13.

[Lyon84a] Lyon, B. *External Data Representation* - Protocol Specification, Mountain View (Calif.), Sun Microsystems, 1984. Citado na página 87.

[Lyon84b] Lyon, B. *Remote Procedure Call - Protocol Specification*, Mountain View (Calif.), Sun Microsystems, 1984. Citado nas páginas 72 e 87.

[Mafla91] Mafla, Enrique & Bhargava, Bharat. *Communication Facilities for Distributed Transaction-processing Systems*, IEEE Computer, Aug. 1991, pp. 61-66. Citado na página 15.

[Malamud92b] Malamud, C. *Analyzing SUN Networks*, Van Nostrand Reeinhold. 1992. Citado nas páginas 12, 16, 73 e 79.

[Martin92] Martin, James & Chapman, Kathleen K. & Leben, Joe. *System Application Architecture: Common Communications Suport for Distributed Applications*, Prentice-Hall, 1992. Citado na página 10.

[Millikin91] Millikin, Michael D. *Distributed Computing Futures: DCE, ONC and Beyond*, Unix Expo, Gunstock Hill Associates, 31 Oct. 1991. Citado na página 8.

[Mullender89] Mullender, Sape. *Distributed Systems*, Addison-Wesley, 1989. Citado nas páginas 22, 67 e 70.

[Nitzberg91] Nitzberg, Bill & Lo, Virginia. *Distributed Shared Memory: A Survey of Issues and Algorithms*, IEEE Computer, Aug. 1991, pp. 52-60. Citado na página 36.

[OSF91a] Open Software Fundation. *Guide to OSF/1: A Technical Synopsis*, O'Reilly & Associates, Inc. 1991. Citado nas páginas 12 e 16.

[OSF91b] Open Software Fundation. *File Systems in a Distributed Computing Environment*, A White Paper, Jul. 1991, pp. 1-7. Citado na página 13.

[OSF91e] Open Software Foundation. *Distributed Management Environment*, White Paper, 1991. pp. 1-8. Citado na página 16.

[OSF92b] Open Software Fundation. *Distributed Computing Environment, Overview*, Jan. 1992, pp. 1-12. Citado na página 9.

[Panzieri88] Panzieri, Fabio & Shrivastava, Santosh K. *Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing*, IEEE Transactions on Software Engineering, vol. 14, no. 1, Jan. 1988, pp. 30-33. Citado na página 64.

[Powell91] Powell, M. L. & Kleiman, S. R. & Barton, S. & Shah, D. & Stein, M. & Weeks, M. *SUN/OS Multi-Thread Achitecture*, USENIX, Winter'91, Dallas-Texas, pp. 65-79. Citado na página 11.

[Ravindran89] Ravindran, K. & Chanson, Samuel T. *Failure Transparency in Remote Procedure Calls*, IEEE Transactions on Computers, vol. 38, no. 8, Aug. 1989, pp. 1773-1187. Citado na página 64.

[Rose91] Rose, M. T. *An Introduction to Management of TCP/IP - based internets*, Prentice Hall, 1991. Citado na página 10.

[Shirley92] Shiley, John. *Guide to Writing DCE Applications*, O'Reilly & Associates, Inc., Jun., 1992. Citado na página 6.

[Spector89] Spector, A. Z. *Distributed Transaction Processing Facilities*, In Mullender, S.(ed.): Distributed Systems, Reading (Mass.) Addison_Wesley, 1989, pp. 191-214. Citado nas páginas 15 e 60.

[Stern91] Stern, H. *Managing NFS and NIS*, O'Reilley & Associates, Inc. Jun., 1991. Citado nas páginas 12, 20, 109 e 110.

[Stevens90] Stevens, W. R. *UNIX Network Programming*, Prentice Hall, 1990. Citado nas páginas 39 e 59.

[Sun88a] Sun Microsystems. *Remote Procedure Call - Programming Guide*, Mountain View (Calif.), Sun Microsystems, 1988. Citado na página 87.

[Sun88b] Sun Microsystems. *Network Programming*, Mountain View (Calif.), Sun Microsystems, 1988. Citado nas páginas 72 e 87.

[Tanenbaum92] Tanenbaum, Andrew. S. *Modern Operating Systems*, Prentice Hall. 1992. Citado nas páginas 1, 63 e 70.

[Taylor86] Taylor, B. & Goldberg, D. *Secure Networking in the Sun Environment*, Proceeding of the USENIX Summer Conference, 1986. Citado na página 80.

[UNIX92] UNIX International. *Distributed Transactions Processing Environments: A Competitive Analysis of UNIX System Laboratories' TUXEDO System and Transarc Encina*, UNIX International - Waterview Corporate Center, Apr. 1992, pp. 1-24. Citado na página 15.

[Wilbur87] Wilbur, S. & B. Bacarisse. *Building Distributed Systems with Remote Procedure Call*, IEEE Software Engineering Journal, Sep. 1981, pp. 148-159. Citado nas páginas 35 e 40.

[Woo92] Woo, Thomas Y. C. & Lam, Simon S. *Authentication for Distributed Systems*, IEEE Computer, Jan. 1992, pp. 39-52. Citado na página 151.

Referências Gerais

[Ahno91] Ahno, Y. *Distributed Environments Software Paradigm and Workstation*, Springer-Verlag, 1991.

[Ananda91] Ananda, A. L. & Srinivasan, B. *Distributed Computing Systems: Concepts and Structures*, IEEE Computer Society Press, 1991.

[Andrews91] Andrews, Gregory R. *Paradigms for Process Interaction in Distributed Programs*, ACM Computing Surveys, vol. 23, no. 1, Mar. 1991, pp. 49-90.

[Ben-Ari82] Ben-Ari, M. *Principles of Concurrent Programming*, Prentice Hall International, 1982.

[Ben-Ari92] Ben-Ari, M. *Principles of Concurrent and Distributed Programming*, Prentice-Hall, 1992.

[Bershad87] Bershad, Brian N. & Ching, Dennis T. & Lazowska, Edward D. & Sanislo, Jan & Schwartz, Michael. *A Remote Procedure Call Facility for Interconecting Heterogeneous Computer Systems*, IEEE Transactions on Software Engineering, vol. SE-13, no. 8, Aug. 1987, pp. 880-894.

[Cheng91] Cheng, Hsiao-Chung & Sheu, Jang-Ping. *Design and Implementation of Distributed File System*, Software - Practice and Experience, Jul. 1991, Vol. 21(7), pp. 657-675.

[Denning90] Denning, Peter J. *Computers Under Attack: Intruders, Worms, and Viruses*, Addison-Wesley, ACM Press, New York, 1990.

[Goscinski91] Goscinski, Andrzej & Bearman, Mirion. *Resource Management in Large Distributed Systems*, fonte desconhecida, pp. 7-25.

[Havender68] Havender, J. W. *Avoiding Deadlock in Multitask Systems*, IBM System Journal, 1968, 7:2, pp. 74-84.

[Kochan89] Kochan, G. S, & Wood, P. H. *UNIX Networking*, Hayden Books, 1989

[Lamport86] Lamport, Leslie. *On Interprocess Communication*, Distributed Computing, Springer-Verlag, 1986, Parte II, pp. 86-101.

[Lampson91] Lampson, Butler & Abadi, Matin & Burrows, Michael & Wobber, Edward. *Authentication in Distributed Systems: Theory and Pratice*, ACM SIGOPS, 1991.

[Levyh91] Levy, Henry M. & Tempero, Ewan D. *Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invocation*, Software-Practice and Experience, vol. 21(1), Jan., 1991, pp. 77-90.

[Liskov86] Liskov, Barbara & Weihl, William. *Specifications of Distributed Programs*, Distributed Computing, Springer-Verlag, 1986, pp. 102-118.

[Malamud92a] Malamud, C. *Stacks: Interoperability in Today's Computer Networks*, Prentice-Hall, 1992.

[Martinb91] Martin, Bruce E. & Pedersen, Claus H. & Roberts, James Bedford. *An Object-Based Taxnomy for Distributed Computing Systems*, IEEE Computer, Aug. 1991, pp. 17-27.

[Notkin86] Notkin, David & Hutchinson, Norm & Sanislo, Jan & Schwartz. *Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity*, ACM Operating System Review 20.2, Apr. 1986, pp. 9-24.

[Obermarck82] Obermarck, R. *Distributed Deadlock Detection Algorithm*, ACM Transactions on Database System, 1982, 7:2, pp. 187-208.

[OSF90a] Open Software Fundation. *Security in the OSF/1 Operating System*, A White Paper, Nov. 1990, pp. 1-7.

[OSF90b] Open Software Fundation. *A Look at Computing in the 1990s*, A White Paper, OSF-O-WP5-0890-1, Aug. 1990, pp. 1-8.

[OSF91c] Open Software Fundation, *Interoperability: A Key Criterion for Open Systems*, A White Paper, Nov. 1991, pp. 1-5.

[OSF91d] Open Software Foundation. *Remote Procedure Call in Distributed Computing Environment*, White Paper, Aug. 1991. pp. 1-12.

[OSF92a] Open Software Fundation. *An Analysis of the OSF/1 Operating System and UNIX System V Release 4*, A White Paper, Jan. 1992, pp. 1-34.

[OSF92c] Open Software Fundation. *Security in a Distributed Computing Environment*, A White Paper, Jan. 1992, pp. 1-6.

[Osher92] Osher, M. H. *Software Without Walls*, Revista Byte, Mar. 1992, pp. 122-128.

[Perlman92] Perlman, Radia. *Interconnections: Bridges and Routers*, Addison-Wesley, 1992.

[Pressman87] Pressman, Roger S. *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 1987.

[Probst91] Probst, Richard & Sventek. Joe. *Distributed Object Technology*, ONC Industry Networking Conference, Oct. 7, 1991.

[Rai90] Rai, Suresh & Agrawal, Dharman P. *Advances in Distributed System Reliability*, IEEE Computer Society Press, 1990.

[Rochkind85] Rochkind, Marc J. *Advanced UNIX Programming*, Prentice-Hall, 1985.

[Santifall91] Santifaller, M. *TCP/IP and NFS - Internetworking in UNIX Environment*, Addison-Wesley. 1991

[Satyanara90] Satyanarayanan, Mahadev. *Parallel Communication in a Large Distributed Environment*, IEEE Transactions on Computers, Vol. 39, No. 3, Mar. 1990, pp. 328-348.

[Sauvé92] Sauvé, Jacques P. *Computação Distribuída para os Anos 90*, Transparência do Seminário Infotrends'92, Abr. 92, São Paulo.

[Schneidew89] Schneidewind, Norman F. *Distributed System Software Design Paradigm with Application to Computer Network*, IEEE Transaction on Software Engineering, Vol. 15, No. 4, Apr. 1989, pp. 402-412.

[Shate89] Shate, Sol & Wang, Jia-Ping. *Distributed Software Engineering*, IEEE Computer Society Press, 1989.

[Shrivast82] Shrivastava, S. K. & Panzieri, F. *The Design of a Reliable Remote Procedure Call Mechanism*, IEEE Transaction on Computer, Vol. C-31, Jul. 1982, pp. 692-697.

[Singh91] Singh, Ajit & Schaeffer, Jonathan & Green, Mark. *A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstation*, IEEE Transaction on Parallel and Distributed Systems, Vol. 12, No. I, Jan. 1991, pp. 52-66.

[Singhal91] Singhal, Mukesh & Casavant, Thomas L. *Distributed Computing Systems*, IEEE Computer, Aug. 1991, pp. 12-14.

[Stamos90] Stamos, James W. & Gifford, David K. *Implementation Remote Evaluation*, IEEE Transactions on Software Engineering, vol. 16, no. 7, Jul. 1990, pp. 710-722.

[Stankovic84] Stankovic, John A. *A Perspective on Distributed Computer Systems*, IEEE Transactions on Computer, vol. C-33, no. 12, Dec. 1984, pp. 1102-1114.

[Stevens92] Stevens, W. R., *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992

[Sunsoft91] SunSoft. *Solaris ONC - Network Information Service Plus (NIS+)*, Sun Microsystems Company, A White Paper, pp. 1-20.

[Svobodova90] Svobodova, Liba & Janson, Philippe A. & Mumprecht, Eduard. *Heterogeneity and OSI*, IEEE Journal on Selected Areas in Communications, Vol. 8, No. 1, Jan. 1990, pp. 67-79.

[Tanenbaum90] Tanenbaum, Andrew S. & Renesse, van Robbert & Staveren, van Hans & Sharp, Gregory J. & Mullender, Shape & Jansen, Jack & Rossum, van Guido. *Experiences with AMOEBA - Distributed Operating System*, Communications of ACM, Vol. 33, No. 12, Dec. 1990, pp. 47-63.

[Tay90] Tay, B. H. & Ananda, A. L. *A Survey of Remote Procedure Calls*, Operating Systems Review, vol. 24, Jul. 1990, pp. 68-79.

[UNIX90a] UNIX International. *Distributed Computing in UNIX System V - Technology Today, A Strategy for Tomorrow*, UNIX International - Waterview Corporate Center, Apr. 1990, pp. 1-28.

[UNIX90b] UNIX International. *UNIX System V Security Today and Tomorrow*, UNIX International - Waterview Corporate Center, Jan. 1991, pp. 1-11.

[UNIX90c] UNIX International. *UNIX System V Multiprocessing - A Standard Path to the Future*, UNIX International - Waterview Corporate Center, Jan. 1990, pp. 1-14.

[Villars91] Villars, Richard. *Strategic Assessment of Distributed Computing*, International Data Corporation, Dec. 1991, pp. 1-29.

[Wittie91] Wittie, Larry D. *Computer Networks and Distributed Systems*, IEEE Computer, Sep. 1991, pp. 67-76.

[Yemini89] Yemini, Shaula A. & Goldszmidt, German S. & Stoyenko, Alexander D. & Wei, Yi-Hsiu. *CONCERT: A High-Level-Language Approach to Heterogeneous Distributed Systems*, ACM Transaction on Programming Language and Systems, 1989, pp. 162-171.

Apêndice A

A1. Semântica de Compartilhamento do NFS

Neste apêndice, serão descritos alguns detalhes de implementação do NFS relacionados com a sua semântica de compartilhamento de arquivos. Também serão sugeridas algumas estratégias para adequar a semântica do NFS às necessidades do programador.

A1.1. Detalhes de Implementação

O NFS não possui uma semântica de compartilhamento de arquivos bem definida. Uma rápida análise de alguns aspectos da implementação do NFS, ajuda a entender porque sua semântica de compartilhamento não pode ser caracterizada com clareza. Além disso, torna possível para o programador definir estratégias para evitar problemas, relacionados à semântica, que possam comprometer a especificação da aplicação.

O NFS foi originalmente implementado com o serviço de RPC. Ele é composto basicamente por um conjunto de serviços denominados procedimentos *nfsproc*. Normalmente, as operações de leitura e escrita, envolvendo arquivos remotos, são mapeadas para os serviços de RPCs correspondentes do *nfsproc*. Por exemplo, uma operação de escrita efetuada em um arquivo, montado com NFS, é mapeada para a RPC *nfsproc_write()*; uma operação de leitura é mapeada para a RPC *nfsproc_read()*, e assim sucessivamente.

Para otimizar seu desempenho, o NFS armazena os blocos e atributos de arquivos obtidos com as RPCs em *cache*. A *cache* é sempre consultada ou usada como um *buffer* nas operações futuras para minimizar as RPCs de leitura e escrita. Na verdade, o NFS mantém duas *caches* em cada cliente: *cache* de atributos de arquivos e *cache* de blocos de arquivo.

A1.1.1. Cache de Atributos

A *cache* de atributos mantém informações sobre os atributos de diretórios (em UNIX *i-node*) e arquivos, como por exemplo, o seu nome e permissões. É necessária porque, nem sempre, as operações do NFS envolvem o conteúdo dos arquivos. Cada bloco da *cache* de atributo está associado a um tempo de vida, ou ***timer***. Quando um *timer* expira, o bloco associado a ele é descartado da *cache*. Normalmente, o *timer* é de 3 segundos para blocos de atributos de arquivos e de 30 segundos para blocos de atributos de diretório. Um *timer* vencido leva o NFS a acessar o servidor do arquivo remoto para atualizar o conteúdo da *cache*.

A1.1.2. Cache de Blocos

A *cache* de blocos mantém os blocos de dados mais recentemente lidos dos arquivos acessados. Os dados da *cache* de blocos somente são usados, quando os atributos correspondentes estão válidos na *cache* de atributos. A consistência da *cache* de blocos é sempre feita através da *cache* de atributos.

Por razões de eficiência, todas as operações de leitura e escrita entre o cliente e o servidor são sempre feitas em unidades de 8k *bytes*. Quando uma **operação de leitura** é efetuada, o NFS armazena uma unidade de 8k *bytes* na *cache*, independentemente da operação de leitura ter solicitado apenas 100 *bytes*. Se a operação de leitura subsequente for inferior a 8k *bytes*, o comando de leitura retorna os dados armazenados na *cache*. Caso contrário, uma nova unidade de 8k *bytes* será solicitada ao servidor, e a *cache* atualizada. Essa estratégia é conhecida como ***read ahead***. Uma política semelhante é adotada para as **operações de escrita**. Os dados inicialmente são escritos e mantidos na *cache* até que seja completada uma unidade de 8k *bytes*. Quando uma unidade de 8k *bytes* é completada, os dados são enviados para o servidor. Essa estratégia é conhecida como ***delayed write***.

A1.2. Motivos da Indefinição da Semântica

O esquema de *cache* usado pelo NFS assegura um bom desempenho, mas, ao mesmo tempo, sacrifica a clareza da semântica de compartilhamento. A política de *timer*, adotada pela *cache* de atributos, torna a semântica de compatilhamento do NFS dependente de tempo, e portanto obscura. Um atributo de diretório pode levar até 30 segundos para ser atualizado. Consequentemente, arquivos criados em uma máquina podem ser visíveis em outras máquinas, somente após o *timer* expirar.

Uma análise das implicações do esquema de *cache* do NFS deixa claro por que o funcionamento do comando *imp*, ficou comprometido (veja Seção 6.4.5.1). Existem duas razões para o pedido de impressão remota, gravado pelo *imp* do lado cliente, não ser encontrado pelo *spooler* do lado servidor. A primeira razão é o pedido de impressão ficar retido na *cache* do lado cliente, em função da estratégia de *delayed write*. Enquanto o pedido de impressão estiver retido na *cache*, o *spooler* não consegue encontrar o pedido ao ler os arquivos de controle do lado servidor. O pedido só será vísivel, quando o NFS gravar as alterações da *cache* nos arquivos de controle originais. A segunda razão é o pedido ter sido realmente gravado pelo lado cliente, antes do *timer* da *cache* do lado servidor ter expirado. Como o *timer* não expirou, a *cache* do lado servidor ainda não foi atualizada com as modificações mais recentemente efetuadas pelo lado cliente. Quando o *spooler* efetua a operação de leitura em busca do pedido, o NFS retorna os dados desatualizados da *cache* do lado servidor, e o pedido não é encontrado. Somente se o *timer* expirar, antes da operação de leitura do *spooler*, é que o pedido será encontrado.

A1.3. Estratégias para Tratar Problemas de Semântica

Algumas estratégias para tratar os efeitos causados pela semântica de compartilhamento do NFS são: *flush-on-close*, travamento, opção de *mount* e *system call sync*.

A1.3.1. Flush-on-Close

Uma particularidade do NFS é que, quando um arquivo montado é fechado, todos os dados escritos mantidos na *cache* são imediatamente enviados para o servidor que contém o arquivo original.

A1.3.2. Travamento

O travamento exclusivo (em UNIX *system call lockf*) de um arquivo montado com NFS desabilita o esquema de *delayed write* da *cache*. Toda operação de escrita, efetuada no arquivo travado, é sempre imediatamente enviada para o servidor que contém o arquivo original.

A1.3.3. Opção de Mount

Outra forma de desabilitar a *cache* do NFS é através de uma das opções do comando *mount*. Ao montar o arquivo a ser compartilhado, o adminstrador do sistema simplesmente especifica a opção *-noac* para desabilitar o esquema da *cache*. Uma desvantagem dessa solução é que o desempenho fica extremamente penalizado.

A1.3.4. System Call Sync

Algumas implementações de NFS permitem que o programador interfira no esquema de *cache* através de *system calls*. Por exemplo, o NFS da Sun força a atualização da *cache*, quando um *sync()* ou *fsync()* é executado.

Apêndice B

B1. Protocolos de Autenticação

Neste apêndice, serão apresentados os protocolos de autenticação Secure RPC e Kerberos. A notação utilizada na descrição dos protocolos será a seguinte: $X \Rightarrow Y : M$ representa um passo de comunicação no qual X envia uma mensagem M para Y, enquanto que X : "*operação*" representa a computação do passo "*operação*" por X. Os rótulos *I1, I2, I3, ...*, *In* representam os passos de inicialização de credenciais do protocolo, e os passos *A1, A2, A3, ...*, *An* representam as operações de autenticação.

B1.1. Autenticação Secure RPC

O protocolo de Secure RPC pode ser dividido em duas fases. A primeira é a inicialização de credencial. A segunda é a fase de autenticação entre o cliente e o servidor.

B1.1.1. Inicialização de Credencial

A inicialização de credenciais entre um cliente C e um servidor S funciona da seguinte forma:

(I1) C : gera chave de conversação CK randomicamente
(I2) : computa chave de sessão Kcs = $(Ks)^{Kc^{-1}}$
(I3) C ⇨ S : C, $\{CK\}_{Kcs}$, $\{win\}_{CK}$, $\{t1, win + 1\}_{CK}$
(I4) S : computa chave de sessão $Kcs = (Kc)^{Ks^{-1}}$
(I5) : obtém chave de conversação CK usando Kcs
(I6) : obtém win, t1 e (win + 1) usando CK
(I7) : checa verificador da credencial: $win == (win + 1) - 1$?
(I8) : checa se t1 decifrado é um *timestamp* válido
(I9) : checa tempo de vida da credencial: (t1 + win) < tempo local ?
(I10) : armazena C, CK, t1 e win em uma tabela associando a um índice ID

11. Inicialmente, o cliente gera randomicamente uma chave simétrica usando DES. Essa chave, conhecida como **chave de conversação** *CK*, é usada para cifrar os dados trocados entre pares de clientes e servidores, estabelecendo um diálogo privado. O cliente deve enviar *CK* para o servidor, mas antes tem que resolver o seguinte problema: como enviar a chave *CK* sem correr o risco dela ser capturada por alguém que esteja monitorando a rede? Para resolver esse problema, *CK* deve ser cifrada com uma segunda chave conhecida como **chave de sessão** *Kcs*.

12. A chave de sessão é obtida a partir de um esquema de chave pública. *Ks* representa a chave pública do servidor e Kc^{-1} a chave privada do cliente.

13. O cliente envia para o servidor um conjunto de informações que consistem de: uma **credencial** e um **verificador**. A **credencial** é composta pelo nome do cliente *(C)*, a chave de conversação cifrada com a chave de sessão: $\{CK\}_{Kcs}$, e uma janela de tempo cifrada com a chave de conversação: $\{win\}_{CK}$. A janela de tempo tem como função determinar o tempo de vida da credencial, antes que ela seja rejeitada pelo servidor. Em algumas implementações, a janela tem um tamanho de 3600 segundos. É importante observar que a chave de conversação é enviada para o servidor, através da rede, somente nesse passo, o que reduz consideravelmente as possibilidades de roubo por quem esteja monitorando as comunicações. O **verificador** é constituído por duas partes: um *timestamp (t1)* e um checador de janela *(win + 1)*, ambos cifrados com a chave de conversação *CK*. O verificador serve para garantir a integridade da credencial, enviada pelo cliente ao servidor. O motivo de adicionar uma unidade a *win* é para dificultar que algum "intruso" tente reproduzir randomicamente os valores dos *bits* da credencial e do verificador. Por exemplo, supondo que o "intruso" tenha acertado randomicamente os *bits* do valor de *win* cifrado da credencial, a probabilidade dele também acertar, simultaneamente, os *bits* do valor de *win + 1* cifrado do verificador é muito pequena.

14. O servidor, após receber a credencial e o verificador do cliente, necessita computar a chave de sessão *Kcs*, para poder decifrar os dados recebidos. A chave *Kcs* é obtida a partir da chave pública *Kc* do cliente e da chave privada Ks^{-1} do servidor ($Kcs = Kc^{Ks^{-1}} = Ks^{Kc^{-1}}$). O servidor localiza a chave pública a partir do nome do cliente que compõe a credencial.

15. O servidor usa a chave *Kcs* para decifrar a chave de conversação *CK* da credencial.

16. De posse de *CK*, o servidor decifra os dados restantes da credencial e do verificador obtendo *win*, *t1* e *(win + 1)*.

17. O servidor usa os dados do verificador para checar a autenticidade da credencial. Para isso, ele compara a janela da credencial *win* com a janela do verificador *(win + 1)* menos uma unidade. Caso o teste falhe, o cliente é rejeitado pelo servidor.

18. Após se certificar da autenticidade da credencial, o servidor verifica se o *timestamp* decifrado *t1* é válido. Se os *bits* do *timestamp* foram alterados, durante a transmissão, provavelmente, o *timestamp* é inválido e o servidor retorna um código erro para o cliente.

19. O servidor checa se o tempo de vida da credencial ainda não se expirou, ou seja, verifica se o *timestamp t1* ainda está dentro da janela de tempo *win*. Isso equivale a testar se *(t1 + win) < tempo local*. Se o teste falhar, o cliente é rejeitado.

110. Finalmente, o servidor armazena em uma **tabela de credenciais**, para uso futuro, o nome do cliente, a chave de conversação *CK*, o *timestamp t1* e a janela de tempo *win*, associados a um índice *ID*. O *timestamp t1* é armazenado para proteger o servidor de retransmissão de pedidos. O servidor aceitará somente *timestamps* que forem cronologicamente maiores que o último recebido.

B1.1.2. Autenticação Cliente-Servidor

A fase de autenticação cliente-servidor permite que um par de usuários valide com segurança a identidade um do outro. *A1, A2, A3, ... , An* representam os passos do protocolo que funciona da seguinte forma:

(A1) S ⇨ C : ID, $\{t1 - 1\}_{CK}$
(A2) C : obtém *(t1 - 1)* com *CK* e checa se *t1 = = (t1 - 1) + 1*
(A3) C ⇨ S : ID, $\{t2\}_{CK}$
(A4) S ⇨ C : ID, $\{t2 - 1\}_{CK}$
(A5) C ⇨ S : ID, $\{tn\}_{CK}$
(A6) S ⇨ C : ID, $\{tn - 1\}_{CK}$

A1. Finalizado o processo de inicialização, o servidor retorna para o cliente um verificador com o índice *ID* associado aos dados armazenados na tabela de credencial e *(t1 - 1)* cifrado pela chave *CK*. O cliente sabe que apenas o seu servidor pode ter enviado tal verificador, uma vez que apenas o servidor correto conhece o *timestamp* enviado pelo cliente, na fase de inicialização de credenciais. A razão de subtrair 1 de *t1*, é para que o cliente possa verificar que de fato está se comunicando com servidor correto, e não com um falso servidor. Somente o servidor adequado pode subtrair corretamente uma unidade do *timestamp* porque apenas ele possui o *timestamp* válido do cliente. O índice *ID* da tabela de credenciais é usado pelo cliente nas chamadas subsequentes para fins de identificação com o servidor.

A2. O cliente obtém *(t1 - 1)* com *CK*, e verifica se o *timestamp t1* recebido confere com o que foi enviado anteriormente para o servidor. Se *t1* == *(t1 - 1) + 1*, então o cliente assume que o servidor é confiável.

A3. O cliente envia para o servidor apenas o *ID* e um *timestamp* cifrado. O servidor, por sua vez, envia de volta o *timestamp* do cliente menos uma unidade cifrado com *CK*.

A4, A5, A6. As trocas futuras entre o cliente e o servidor repetem o padrão dos passos A1, A2 e A3. O cliente envia o seu ID (não é mais necessário enviar a credencial novamente) mais um verificador cifrado, e o servidor responde com um *timestamp* menos um.

B1.2. Autenticação Kerberos

Para simplificar a descrição do protocolo de autenticação do Kerberos, serão omitidas algumas considerações de administração. O sistema Kerberos (**Fig. B1**) usa dois protocolos de autenticação: protocolo de inicialização de credencial e protocolo de autenticação cliente-servidor. Ambos os protocolos, apresentados a seguir, são baseados nos descritos em [Woo92].

B1.2.1. Inicialização de Credencial

No protocolo de inicialização, descrito a seguir, *U* representa um usuário, *E* uma estação de trabalho associada a um usuário, *Kerb* o servidor de autenticação Kerberos e *I1, I2, I2, ... , In* os passos do protocolo de inicialização.

(I1) E ⇨ U : "informe login: U"

(I2) E ⇨ Kerb: U, TGS

(I3) Kerb : obtém K_u e K_{TGS} do banco de dados

(I4) : gera chave de conversação CK randomicamente

(I5) : cria *ticket* de permissão $\text{ticket}_{TGS} = \{U, TGS, CK, T, L\}_{K_{TG}}$

(I6) Kerb ⇨ E: $\{TGS, CK, T, L, ticket_{TGS}\}_{K_U}$

(I7) E ⇨ U : "senha?"

(I8) U ⇨ E : senha *pwd* digitada

(I9) E : calcula $\alpha = f(pwd) \equiv K_U$

(I10) : decifra $\{TGS, CK, T, L, ticket_{TGS}\}_{K_U}$ usando $\alpha \equiv K_U$

(I11) : se decifração falhar então aborta login senão guarda $ticket_{TGS}$ e CK

(I12) : apaga senha *pwd* da memória

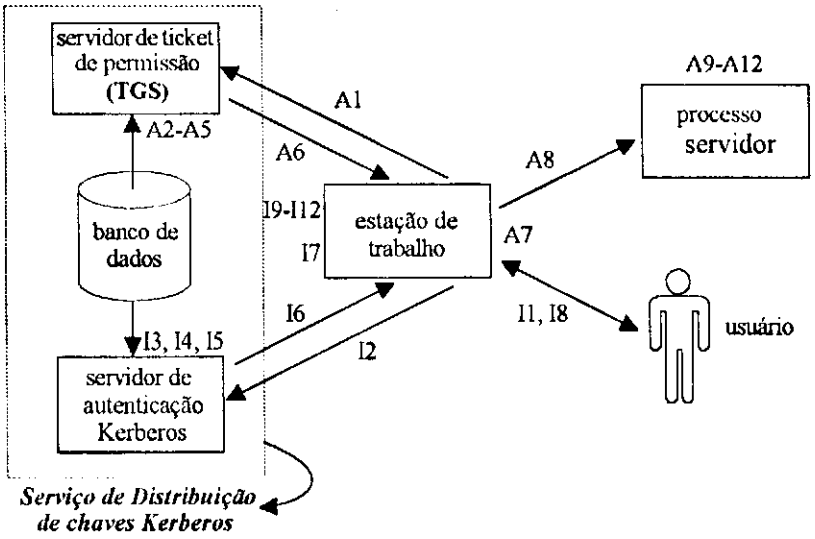


Figura B1. *Entidades do Serviço de Autenticação Kerberos*

I1. Assumindo que o usuário está em uma estação de trabalho UNIX, ele dá início a uma sessão interagindo com o programa de *login* padrão, fornecendo o seu nome de *login U*.

I2. Antes de solicitar a digitação da senha ao usuário, o processo de *login* envia uma mensagem contendo o nome do usuário *U* e o nome do servidor de *ticket* de permissão *TGS*, para o servidor de autenticação Kerberos, através da rede. A mensagem não é cifrada, visto que os nomes do usuário e do servidor de *ticket* de permissão (TGS) são de domínio público na rede.

I3. O servidor de autenticação localiza no banco de dados Kerberos o nome do usuário e do servidor de *ticket*, obtendo assim as chaves (criptosistema simétrico) K_U do usuário e K_{TGS} do servidor de *ticket*. É importante observar que K_{TGS} é uma chave compartilhada somente pelo servidor de *ticket* de permissão (TGS) e pelo servidor de autenticação Kerberos.

I4. O servidor de autenticação gera randomicamente uma chave de conversação *CK*, usada para estabelecer uma sessão confiável entre o *principal* e o TGS.

I5. O servidor de autenticação cria um *ticket de permissão*, denominado $ticket_{TGS}$, onde *T* é um *timestamp* e *L* o tempo de vida da credencial. O $ticket_{TGS}$ é cifrado, antes de ser enviado através da rede, com a chave K_{TGS}, obtida no passo I3. O $ticket_{TGS}$ é usado, por todo *principal,* para ser reconhecido pelo TGS como um usuário confiável, e obter permissão (um novo *ticket*) para acessar um determinado serviço.

I6. O servidor de autenticação *Kerb* retorna para o processo de *login* da estação de trabalho *E,* uma mensagem com várias informações, cifradas com a chave K_U, obtida no passo I3. Obviamente, somente o usuário *U* compartilha a chave *Ku* com *Kerb* e pode interpretar corretamente o conteúdo da mensagem. Pode-se observar que TGS, CK, T e existem explicitamente como campos da mensagem e implicitamente no $ticket_{TGS}$.

I7. O programa de *login* da estação de trabalho *E* recebe a mensagem cifrada e, somente então, solicita a digitação da senha para o usuário *U.*

I8. O usuário *U* informa sua senha *pwd* para o programa de *login* da estação de trabalho.

I9. Para decifrar a mensagem recebida no passo 7, o processo de *login* da estação de trabalho deve obter a chave K_U. O processo de *login* usa uma função *f* (*one-way*) para obter uma chave α , equivalente a K_U , a partir da senha do usuário *pwd.*

I10. A mensagem é decifrada com α, obtendo-se *TGS, CK, T* e o $ticket_{TGS}$.

I11. O processo de *login* verifica: a) se o TGS decifrado confere com o que foi enviado no passo I2 para o servidor de autenticação Kerberos; b) se o *timestamp T* é válido; c) se o tempo de vida *L* do *ticket* não expirou. Se for detectado algum problema, o processo de *login* é abortado, caso contrário, *CK* e o $ticket_{TGS}$ são armazenados. A chave de conversação *CK* é vista pela estação de trabalho como um *string* de *bits* randômicos, gerados pelo servidor de autenticação Kerberos. O $ticket_{TGS}$ é incompreensível pela estação de trabalho, porque está "selado" (cifrado) com a chave K_{TGS} que é somente conhecida pelo TGS e Kerberos. Toda estação de trabalho, após o processo de inicialização, passa a possuir um $ticket_{TGS}$ com a correspondente chave de sessão CK, individualizados.

I12. A senha do usuário *pwd* é apagada da memória, para evitar que algum usuário intruso tente conseguí-la para quebrar a segurança, finalizando o processo de inicialização.

B1.2.2. Autenticação Cliente-Servidor

Somente o *ticket*TGS não contitui prova suficiente de identidade, visto que ele pode ser interceptado ou copiado por usuários intrusos. Então, é necessário que o *principal*, além de apresentar o $ticket_{TGS}$, demonstre também conhecimento da chave *CK* que compõe o $ticket_{TGS}$. A seguir, será apresentado o protocolo para um cliente *C* solicitar, através do Kerberos, um serviço para um servidor *S*. Todos os passos de autenticação, representados por *A1, A2, A3, ... , An,* descritos a seguir, são transparentes para o cliente que acessa o serviço.

(A1) C ⇨ TGS : S, $ticket_{TGS}$, $\{C, T1\}_{CK}$
(A2) TGS : obtém *CK* do $ticket_{TGS}$ usando K_{TGS}
(A3) : obtém *T*1 de $\{C, T1\}_{CK}$ usando *C*
(A4) : checa validade de *T*1 com o relógio local
(A5) : cria *ticket* do servidor $ticket_S = \{C, S, CK', T', L'\}_{K_S}$
(A6) TGS ⇨ C: envia mensagem $\{S, CK', T', L', ticket_S\}_{CK}$
(A7) C : obtém *CK'*, $ticket_S$ usando *CK*
(A8) C ⇨ S : $ticket_S$, $\{C, T2\}_{CK'}$
(A9) S : obtém *CK'* de $ticket_S$ usando K_S
(A10) : obtém *T*2 de $\{C, T2\}_{CK'}$ usando *CK'*
(A11) : checa a validade de *T2* com o relógio local
(A12) S ⇨ C : $\{T2 + 1\}_{CK'}$

A1. O cliente cria e envia uma mensagem para o TGS, onde *S* é o nome do servidor que possui o serviço desejado pelo cliente, e $\{C, T1\}_{CK}$ respresenta um *autenticador* cifrado com a chave de conversação, obtida durante a inicialização de credenciais. O autenticador é um meio do cliente provar que conhece a chave de conversação *CK* para o TGS. Dessa forma, o TGS reconhece o cliente como um usuário confiável.

A2. O servidor de *ticket* de permissão, ao receber mensagem, decifra o $ticket_{TGS}$ com a chave K_{TGS} para obter *CK*.

A3. O servidor de *ticket* de permissão usa *CK* para decifrar o autenticador, recuperando *T1* e *C*.

A4. O servidor de *ticket* de permissão valida o nome de *login* C (U) do $ticket_{TGS}$ com o nome de login C do autenticador, bem como o nome do servidor *TGS*. Em seguida, checa o *timestamp* $T1$ do autenticador com o relógio local para se certificar que a mensagem é recente. Isso requer que todas as estações de trabalho e servidores mantenham os seus relógios sincronizados dentro de uma faixa de tolerância pré-estabelecida.

A5. O TGS cria randomicamente uma chave de conversação CK' a ser compartilhada entre o cliente e o servidor, define um novo *timestamp* T' e o tempo de vida de credencial L'. Em seguida, usa o nome do serviço S da mensagem, para pesquisar o banco de dados e obter a chave criptográfica K_S do serviço S. Finalmente, o TGS cria um novo *ticket*, baseado em S e CK'. O $ticket_S$ "amarra" o cliente ao serviço requerido, através da chave CK'. (associa o cliente que fez o pedido e o serviço que ele solicitou a uma chave particular de conversação CK'). Todos os dados do $ticket_S$ são cifrados com a chave K_S. O *ticket*S será usado, no futuro, pelo cliente, para solicitar a execução do serviço específico, cuja permissão de execução, ele solicitou ao TGS.

A6. O TGS cria uma mensagem com os dados gerados no passo A5. A mensagem é cifrada com CK e enviada para o cliente que solicitou permissão para executar o serviço.

A7. O cliente recebe a mensagem, decifra-a com CK, obtendo CK' e $ticket_S$ e os outros dados. Em seguida, o cliente repete a sequência de validação, com os dados decifrados, de forma análoga ao passo A4.

A8. O cliente gera um autenticador cifrado com a chave CK', contendo seu nome C e um *timestamp* $T2$. O cliente envia uma mensagem com o $ticket_S$ e o novo autenticador para o servidor. É dessa forma que o cliente solicita a execução de um serviço para um servidor.

A9. O servidor requerido recebe a mensagem e obtém CK', decifrando o $ticket_S$ com a chave K_S, que somente ele e servidor de autenticação Kerberos compartilham. A chave K_S é obtida pelo servidor durante seu processo de inicialização.

A10. O servidor decifra o autenticador com CK', recuperando $T2$ e aplicando o mesmo processo de validação, descrito no passo A4.

A11. Finalmente, o servidor retorna para o cliente uma mensagem contendo o *timestamp* $T2$, adicionado de uma unidade, cifrado com a chave CK'. Isso é feito para que o cliente possa se certificar da identidade do servidor.

Glossário

ACD — *Ambiente de Computação Distribuída*. Um ACD pode ser definido como um modelo arquitetural de computação no qual uma coleção de processos, recursos e computadores de diferentes fabricantes, distribuídos numa rede, trabalham cooperativamente para executar tarefas.

ACL — *Lista de Controle de Acesso*. Uma facilidade de segurança do sistema operacional que permite especificar a segurança de objetos, através de uma lista das operações permitidas para cada usuário.

AD heterogênea — Uma Aplicação Distribuída heterogênea é composta por processos que podem residir em computadores com arquitetura ou sistema operacional diferentes.

AD — *Aplicação Distribuída*. Tipo de aplicação cujo código é executado por dois ou mais processos de aplicação, em diferentes computadores de uma rede de comunicação.

API — *Application Programming Interface*. As APIs são bibliotecas de funções padronizadas, colocadas à disposição do programador para permitir o desenvolvimento de aplicações com um maior nível de abstração, facilitando o trabalho de programação.

ARPANET — *Advanced Research Projects Agency Network*. Rede militar do departamento de defesa americano. Substituída pela *Defense Data Network*.

ASCII — *American Standard Code for Information Interchange*. Um padrão de conjunto de caracteres que representam uma sequência octal para cada letra, número, e caracteres de controle selecionados.

autenticação — Função de verificação da identidade de um usuário ou processo.

autorização — Determina se o usuário ou processo está apto para processar uma ação particular.

bind Termo usado para definir o estabelecimento de uma sessão entre duas unidades lógicas.

binding Um processo de mapeamento do nome de um serviço, para o seu respectivo endereço de transporte. No contexto de RPC, representa uma ligação lógica entre um processo cliente e um processo servidor para permitir a chamada de procedimentos remotos.

bloco Unidade de entrada/saída (I/O) em computadores. Normalmente, o tamanho de um bloco varia de 512 *bytes* a 8k *bytes*.

broadcast Envio de informações para todos os usuários de um serviço particular. Por exemplo, um *broadcast* Ethernet envia pacotes para todos os endereços da rede.

buffer Uma parte da memória principal de um computador usada para manter dados.

C-S *Cliente-Servidor*. Modelo conceitual para o desenvolvimento de aplicações em rede. Permite estruturar uma aplicação em dois grupos distintos de processos cooperativos. O primeiro grupo, denominado servidor, oferece serviços para os usuários do segundo grupo, denominados de cliente.

Capabilities Uma lista de operações disponíveis e permitidas que um usuário pode realizar com um objeto.

checksum Um valor calculado usado para testar a integridade de dados.

cliente Um módulo que usa os serviços de outro módulo. Por exemplo, uma camada de sessão é um cliente de uma camada de transporte.

concorrência Quando múltiplos processos tentam acessar um mesmo recurso.

console Uma unidade de controle, como um terminal, através da qual o usuário se comunica com o computador.

cps Caractere por segundo.

criptosistema Um conjunto de regras que especificam os passos matemáticos necessários para codificar e decifrar dados.

datagrama Uma unidade de informação com endereço de destino associado que é enviada através de uma rede de comutação de pacotes.

deadlock Um termo usado em um ambiente concorrente. Representa uma situação em que dois ou mais processos, disputando a utilização de um recurso, esperam, indefinidamente, por eventos com baixa condição de ocorrência ou que nunca irão ocorrer.

DES *Data Encryption Standard*. Um tipo criptosistema criado pela IBM.

DFS *Distributed File System*. Um serviço que permite tornar disponível, em uma máquina local, um sistema de arquivos remoto. Por exemplo, através do DFS é possível tornar disponível (montar), em uma estação de trabalho sem disco, um sistema de arquivos remoto.

domínio Em uma rede, representa os recursos que estão sob o controle de um ou mais computadores associados .

downsize ou downsizing Termo relacionado com a migração de um sistema de grande porte (*hardware* e *software*), para sistemas (computadores ou redes) de menor porte e custo. Além do processo de migração, o *downsize* também se refere a re-estruturação da organização como um todo, para adaptá-la ao novo ambiente.

EBCDIC *Extended Binary Coded Decimal Interchange Code*. Um esquema de código de caracteres usado em ambientes IBM.

entidade Termo, usado em gerenciamento de rede, que se refere a uma parte gerenciável de um sistema distribuído.

espaço de nomes Um conjunto distribuído de nomes no qual, normalmente, todos os nomes são únicos.

Ethernet Protocolo de *link* de dados desenvolvido pela Intel, Xerox, DEC e posteriormente adotado como padrão 802.3 pela IEEE.

falha É todo evento causado por um defeito de *hardware* ou *software* que viola a especificação da aplicação, dando origem a um erro.

FDDI *Fiber Distributed Data Interface*. Um padrão LAN de fibra ótica de 100-Mbps em uma rede em anel (*token ring*).

FQDN *Fully Qualifield Domain Name*. É o padrão *Internet* de nome de máquina e domínio. Por exemplo, a usuária maria da Universidade Federal da Paraíba teria o seguinte FQDN: *maria@dsc.ufpbcg.br*.

gateway Um computador dedicado de propósito especial que conecta duas ou mais redes e roteia pacotes de uma rede para outra.

idempotência Descreve uma operação, cujo resultado é sempre o mesmo quando executada mais de uma vez. Atribuir 13 a X é idempotente. Adicionar 13 a X não é.

identificação É o processo pelo qual uma entidade (usuário ou recurso) reclama uma identidade indiscutível, previamente estabelecida.

inetd *Internet Service Deamon*. Um servidor da plataforma ONC usado para inicializar outros servidores durante o processo de *boot* das estações de trabalho.

Internet Uma coleção de redes que compartilham o mesmo espaço de nomes e usam protocolos TCP/IP. A Internet consiste de pelo menos 400 redes conectadas. A Internet não deve ser confundida com a internet (letra minúscula).

internet Uma coleção de redes conectadas por *gateways* e roteadores. Geralmente, refere-se a uma variedade de redes heterogêneas que são acessíveis por correio eletrônico (*email*), através dos *gateways*.

IP *Internet Protocol*. Camada de protocolo de rede da Internet.

kbyte *Kilobyte*. 1.024 *bytes*.

Kerberos O serviço de autenticação do projeto Athena do *Massachusetts Institute of Technology* (MIT). O Kerberos é um serviço de autenticação, baseado em um criptosistema simétrico.

log Um arquivo que contém dados pertinentes à execução de uma aplicação, em uma determinada máquina.

login Processo, através do qual um usuário se identifica perante o sistema.

Mbit *Megabit*. Um milhão de *bits*.

mecanismo RPC Camada de *software* que permite a um processo cliente chamar um procedimento remoto que se encontra no espaço de endereçamento de um processo servidor. Durante uma RPC, o mecanismo RPC procura tornar a complexidade da rede transparente para o cliente e para o servidor.

MOD *Mecanismo de Objetos Distribuídos*. É uma camada de *software* que permite a invocação de objetos remotos em um ambiente de computação distribuída.

mount Um serviço do NFS que permite tornar um sistema de arquivos remoto disponível para o sistema local.

netname Um identificador simbólico que o usuário utiliza para se referir a um usuário ou a um recurso da rede. Os *netnames* são identificadores globais únicos que viabilizam a identificação dos usuários e recursos nos ACDs.

NFS *Network File System*. Sistema de arquivos distribuído, desenvolvido pela Sun Microsystems, amplamente utilizado em ambientes TCP/IP.

NIS *Network Information Service*. Serviço de diretório distribuído (*naming*) da plataforma *Open Network Computing* da Sun Microsystems.

NLM *Network Lock Manager*. Serviço de travamento de arquivos distribuído da plataforma *Open Network Computing* da Sun Microsystems.

objeto Um objeto é uma entidade de *software*, composta por alguns dados privados, (estado) e por um conjunto de operações associadas (procedimentos), públicas ou privadas, que manipulam os dados. Um objeto pode ser algo tão simples quanto a célula de uma planilha eletrônica ou tão complexo quanto uma aplicação completa.

ONC *Open Networking Computing*. Plataforma da Sun Microsystem que oferece um conjunto de serviços para o desenvolvimento de aplicações distribuídas de redes heterogêneas.

OSF *Open Software Fundation*. Organização fundada pela Digital, IBM, e quatro outros fabricantes, para desenvolver especificações de *software* para sistemas abertos.

OSI *Open Systems Interconecction*. Padrão ISO para arquiteturas abertas de redes heterogêneas.

overhead Trabalho ou informação que fornece suporte - possivelmente suporte crítico - para um processo de computação, mas que não é parte intrínseca da operação ou dado.

pacote Unidade de dados enviada através de uma rede de comutação de pacotes.

PC *Personal Computer*. Um computador *desktop* desenvolvido pela IBM ou um clone fabricado por terceiros.

pilha de protocolo Um conjunto de funções que trabalham cooperativamente para formar um conjunto de serviços de rede. Cada uma dessas funções, compõe uma camada da pilha de protocolo e utiliza os serviços das camadas inferiores, para executar os seus serviços.

porta No *hardware* de um computador, é uma localização para passagem de dados de entrada/saída de um dispositivo de computação. Microprocessadores têm portas para enviar ou receber *bits* de dados.

principal É toda entidade cuja identidade pode ser verificada por um serviço de autenticação.

protocolo Uma descrição formal de formato de mensagens e de regras que dois ou mais computadores devem seguir para trocar essas mensagens.

rede heterogênea Uma rede constituida por diferentes protocolos de rede e/ou diferentes tipos de computadores.

root Superusuário (normalmente o administrador) de sistemas UNIX. Uma conta que tem, em sistemas UNIX, acesso privilegiado.

RPC Batch Chamada de procedimento remoto em lote. Facilidade oferecida por alguns mecanismos RPC que permite agrupar em um único lote, a ser enviado para o servidor, várias RPCs.

RPC Broadcast Uma facilidade oferecida por alguns mecanismo RPC que permite enviar um pedido de execução de RPC para todos os usuários de um determinado domínio.

RPC — *Remote Procedure Call*. O serviço de chamada de procedimento remoto, ou RPC, fornece um paradigma de alto nível que permite a comunicação entre processos que residem em diferentes espaços de endereçamento pertencentes a uma ou mais máquinas.

Secure RPC — Protocolo de autenticação da Sun Microsystems.

seek — É o processo de movimentar a cabeça de leitura/gravação em um disco para uma determinada localização, tipicamente para ler ou gravar dados.

semáforo — Um mecanismo de sincronização em um sistema operacional.

servidor de boot — É um servidor dedicado, usado para inicializar outros servidores, durante o processo de *boot* das máquinas, ou para re-inicializar servidores que falham.

servidor de ciclos — Termo normalmente associado a uma máquina com grande capacidade de processamento, como por exemplo, um *mainframe*.

servidor — Um módulo que executa serviços solicitados por um cliente.

sinal — É uma interrupção de *software*.

SNA — *System Network Achitecture*. Arquitetura de rede da IBM.

SPARC — *Scalable Processor Achitecture*. Um processador com um conjunto reduzido de instruções (RISC), desenvolvido pela Sun e licenciado para vários fabricantes, incluíndo AT&T e Texas Instruments.

spooler — Processo servidor de impressão do gerenciador de impressão *spoolview*.

spoolview — Serviço de impressão para ambiente centralizado UNIX, desenvolvido e comercializado pela Infocon Tecnologia.

stub servidor — Um trecho de código que esconde os detalhes de rede de um processo servidor durante uma RPC.

stub — Um trecho de código que é usado em chamadas de procedimentos remotos. O *stub* tem a função de esconder os detalhes de rede do processo que faz a RPC.

stub cliente Um trecho de código que esconde os detalhes de rede de um processo cliente durante uma RPC.

superusuário O mesmo que *root*.

TCP *Transmission Control Protocol*. Protocolo de transporte orientado a conexão do TCP/IP, usado para garantir confiabilidade de dados.

TCP/IP *Transmission Control Protocol/Internet Protocol.* Conjunto de protocolos de transporte e de aplicações, usado para dar suporte a serviços de *login* remoto (telnet), transferência de arquivos (FTP), correio eletrônico (SMTP), UDP e TCP (protocolos da camada de transporte).

thread Um *thread* representa um agente de controle de uma sequência de instruções que estão sendo executadas por um processo.

timeout Valor do limite de tempo que um mecanismo RPC aguarda por uma resposta do servidor, antes de retornar um código de erro para o cliente.

timestamp Um *timestamp* é normalmente um valor de 64 *bits,* que representa o tempo corrente de uma máquina, obtido a partir do seu relógio interno.

tolerância a falhas Um atributo de um sistema computacional que reflete a sua capacidade de tolerar falhas de *hardware* e *software* enquanto continua sendo executado.

travamento Previne que outro usuário tenha acesso a um determinado conjunto de dados.

UDP *User Datagram Protocol*. Protocolo de transporte da pilha de protocolos TCP/IP. Ao contrário de TCP, UDP não garante a confiabilidade dos dados transmitidos.

Unix Sistema Operacional desenvolvido pela AT&T.

verificador Conjunto de dados que permite checar a autenticidade de uma credencial. É geralmente utilizado por serviços de autenticação, como por exemplo, Kerberos e Secure RPC.

XDR *External Data Representation*. Protocolo de apresentação desenvolvido pela Sun Microsystems como parte do NFS. É normalmente usado durante as RPCs para tratar as diferenças de representação interna de dados, face a heterogeneidade normalmente existente nos ACDs.

Yellow Pages — *Páginas Amarelas*. É uma facilidade (ou conjunto de serviços) do *Network Information Service* (NIS) que permite localizar um usuário ou recurso a partir de algum de seus atributos.

ypbind — É um servidor do *Network Information Service* (NIS) que torna possível para os clientes consultarem as informações gerenciadas pelo serviço de diretório distribuído (*naming*) da plataforma *Open Network Computing* da Sun Microsystems.