Universidade Federal da Paraíba

Centro de Ciências e Tecnologia

Coordenação de Pós-Graduação em Informática

SIT

Projeto e implementação de uma biblioteca para recuperação textual

Adriano Sérgio Rodrigues de Souza

Campina Grande - PB

Dezembro, 1995

Adriano Sérgio Rodrigues de Souza

SIT

Projeto e implementação de uma biblioteca para recuperação textual

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

Jacques Philippe Sauvé

(*Orientador*)

Campina Grande - PB

Dezembro, 1995

S719p Souza, Adriano Sergio Rodrigues de
SIT Projeto e implementacao de uma biblioteca para recuperacao textual / Adriano Sergio Rodrigues de Souza. - Campina Grande, 1995.
135f. : il.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Sistemas de Automacao 2. Recuperacao da Informacao 3. Biblioteca - Projeto e Implementacao 4. Dissertacao I. Philippe, Jacques, Ph.D. II. Universidade Federal da Paraiba - Campina Grande (PB). III. Título

CDU 681.5(043)

SIT - PROJETO E IMPLEMENTAÇÃO DE UMA BIBLIOTECA PARA RECUPERAÇÃO TEXTUAL.

ADRIANO SÉRGIO RODRIGUES DE SOUZA

DISSERTAÇÃO APROVADA EM 20/12/95

JACQUES PHILIPPE, Ph.D.
Presidente

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

FRANCISCO VILAR BRASILEIRO, Ph.D.
Componente da Banca

Campina Grande, 20 de dezembro de 1995

Para Maryane

Agradecimentos

Dedico esta página para agradecer àquelas pessoas que contribuíram para a realização deste trabalho. A todas elas externo o meu mais profundo agradecimento e, especialmente:

- A minha esposa, Maryane, pela compreensão, apoio e amor que demonstrou nos momentos fáceis e difíceis deste trabalho e de nossa vida.

- A meus pais pelo amor e pelos anos dedicados à minha educação. A meus irmãos pela amizade e pelo constante incentivo.

- Ao meu orientador, Jacques Philippe Sauvé, por ter compartilhado seu profundo conhecimento comigo, por ter confiado em mim para o desenvolvimento deste trabalho e pela pressão que fez durante a fase de conclusão deste trabalho.

- A Ana Lúcia Guimarães pela insistência e pelo incentivo constantes.

- A Light-Infocon, que possibilitou a realização deste trabalho.

- Ao pessoal da GREEN SOFTWARE pela troca de conhecimentos e pela amizade.

- A Marcos Sebastian e Alessandro Jatobá pela amizade e pela colaboração na fase de implementação do trabalho.

- A Dálmer Júnior e Fernanda Angélica pelos momentos de lazer que me proporcionaram e, principalmente, pela amizade.

- Ao refrigerante, a batatinha frita e até a cerveja que não tomei, mas compartilhei com a turma. Aos sanduíches e pizzas que saciaram a fome.

- Enfim, a Deus, por ter me proporcionado todos os momentos da minha vida e por ter permitido que eu chegasse até aqui.

Resumo

A grande quantidade de informações que as pessoas têm que gerenciar hoje em dia está tornando esse trabalho cada vez mais difícil. A recuperação das informações de forma rápida e eficiente não é possível se não houver um sistema automatizado para essa tarefa.

Com base nessa necessidade de controle de dados, este trabalho descreve o projeto e a implementação de uma ferramenta para gerência de informações não estruturadas e recuperação textual.

Abstract

It is becoming more and more difficult for people to manage large quantities of information. Information retrieval in a fast and efficient way is not possible without an automated system.

The need for higher data control motivated us to pursue this work, which describes the project and implementation of a tool for non structured information management and textual retrieval.

Índice

Índice de Figuras

Índice de Tabelas

Glossário

API - Application Programming Interface - conjunto de funções e/ou classes de objetos disponibilizados para uso do programador.

IPC - InterProcess Communication. Determina um conjunto de técnicas utilizadas para implementação de comunicação entre dois ou mais processos.

RPC - Remote Procedure Call - Esquema de comunicação entre processos que estão em máquinas diferentes, interligadas por uma rede.

DLL - Dynamic-Link Library - Biblioteca que se liga à aplicação dinamicamente.

TERMO - Menor unidade de informação passível de ser armazenada em um índice

GRUPO - Entidade que contém os termos que uma aplicação deseja indexar. Também é chamado de *campo* neste trabalho e pode conter mais de um valor (neste caso, é chamado de multivalorado).

SUBGRUPO - Representa um dos possíveis valores de um grupo, para grupos multivalorados. Dessa forma, um grupo pode conter vários subgrupos. Um subgrupo também é chamado de *repetição* neste trabalho.

Capítulo 1 - Introdução

A tecnologia de armazenamento de dados tem evoluido bastante nos últimos anos. As pessoas sentem necessidade de guardar informações em formatos e tamanhos diversos e os computadores (software + hardware) têm se adaptado à essas necessidades. Uma das últimas tendências no mundo da informática é o uso de *bancos de dados textuais*. Esse tipo de software consegue armazenar grandes volumes de informações não estruturadas e indexá-las de tal forma a tornar possível a recuperação através de palavras, números, datas ou outros elementos de formação dos dados.

A necessidade de armazenar e recuperar grandes volumes de informação tem levado os sistemas de gerenciamento de bancos de dados a evoluções significativas. Questões como velocidade, precisão, controle de acesso e integridade das informações são cada vez mais enfocadas pelos softwares especializados.

Combinar todos os recursos desejados pelos usuários, porém, torna os sistemas de bancos de dados complexos e caros. A idéia deste trabalho é projetar e implementar um sistema de gerenciamento de bancos de dados textuais em forma de uma biblioteca, sem interface com o usuário final. O usuário desse sistema é o programador de aplicações, que implementará seu software de recuperação textual sem se preocupar com questões como segurança, performance, acesso aos dados, integridade das informações, etc. O trabalho desse programador será com a interface do software.

1.1 Importância e objetivos do trabalho

É indiscutível a importância de um sistema automatizado de armazenamento e recuperação de informações na vida das pessoas. Os sistemas gerenciadores de informações estruturadas, assim como os de informações não estruturadas, estão cada vez mais presentes no nosso dia-a-dia, cada um cumprindo o seu papel.

Os gerenciadores de informações estruturadas são úteis para o cadastro de funcionários de uma empresa ou outros processos operacionais semelhantes, mas não conseguem a mesma eficiência no controle de documentos não estruturados, por exemplo. Para cada tipo e formato de informação, o usuário tem que determinar uma estrutura de dados para que os sistemas tradicionais trabalhem com eficiência.

Dessa forma, os sistemas de gerenciamento de informações não estruturadas ganham destaque, pois conseguem manipular informações nos mais variados formatos e tipos sem perda de performance ou eficiência.

O presente trabalho consiste no projeto e implementação de uma biblioteca de gerenciamento de informações textuais, que será chamada de **SIT - Servidor de Informações Textuais** - e poderá ser usada por programadores que desejam elaborar aplicações para manipulação de informações não estruturadas.

1.2 Descrição do Servidor de Informações Textuais

A ferramenta projetada neste trabalho engloba uma série de mecanismos para manipulação de informações com segurança, objetividade e consistência. Trata-se de uma biblioteca de apoio, sobre a qual serão construídas aplicações finais, responsáveis pela interação entre o usuário e as informações.

Com essa ferramenta, o programador terá que se preocupar apenas com a interface de suas aplicações. Todo o trabalho de criação e manipulação de bases de dados é feito pelo SIT, assim como os esquemas de segurança dos dados, integridade, alocação de permissões e cadastramento de usuários.

Um conjunto de requisitos foi cuidadosamente estudado e proposto para o projeto do SIT. Esses requisitos determinam o que o sistema deve e o que não deve implementar para oferecer garantia de uma boa funcionalidade. A implementação, contudo, obedece apenas a um subconjunto dos requisitos, mas deixa o código pronto para que trabalhos futuros possam complementar o atendimento aos requisitos.

Neste trabalho, o SIT implementará as seguintes características:

- suporte a vários tipos de dados
- linguagem de recuperação textual
- gerenciamento de usuários com controle de autenticação
- alocação de autorizações de forma flexível

- simplicidade de gerência de usuários e de dados
- mecanismos de proteção de dados
- recuperação de dados em casos de falha

Cada uma dessas características será analisada nos próximos capítulos deste trabalho.

1.3 Organização do trabalho

Este trabalho está organizado em 5 capítulos. O primeiro faz um levantamento rápido das necessidades de se ter um sistema de recuperação textual e descreve, também de forma resumida, a importância do SIT e o que foi projetado e implementado neste trabalho. No segundo capítulo, encontram-se relacionados todos os requisitos propostos para o sistema, os requisitos que serão atendidos neste trabalho e uma descrição geral do SIT. O terceiro capítulo descreve a implementação do sistema, destacando sua arquitetura e seus objetos. No quarto capítulo encontram-se descritos os detalhes mais importantes da implementação, como a modularização interna e o sistema de tratamento de arquivos. O quinto e último capítulo é dedicado às considerações finais sobre o trabalho e propostas de trabalhos futuros. No final do trabalho, existe um apêndice contendo as classes mais importantes da API do sistema.

Capítulo 2 - Projeto de uma ferramenta para manipulação de informações não estruturadas

Os Sistemas Gerenciadores de Banco de Dados (SGBD) são ferramentas concebidas para armazenamento e recuperação de informações em geral. Entretanto esses sistemas trabalham com informações estruturadas, segundo algum modelo de organização de dados.

Há uma necessidade de gerência sobre documentos, de forma que sejam fáceis o armazenamento e a recuperação de relatórios, artigos, livros e textos em geral. Essa carência motivou o projeto de uma ferramenta capaz de trabalhar de maneira eficaz com informações não estruturadas, como textos por exemplo.

A ferramenta projetada é uma biblioteca que disponibiliza uma API[1] para construção e gerência de Bases de Dados Textuais (BDT). Essa biblioteca, chamada de Servidor de Informações Textuais (SIT), tem como usuário alvo o programador de aplicações finais. Esse usuário cria aplicativos que manipulem BDTs, sem se preocupar com esquemas de segurança, indexação, armazenamento e outros detalhes de programação. Os aplicativos, também chamados de *aplicações clientes*, são construídos sobre o SIT.

O SIT não é simplesmente um sistema de armazenamento/recuperação de textos. Ele atende a uma série de requisitos impostos a um bom sistema de gerência de informações. Esses requisitos serão analisados na próxima seção. A seção 2.2 apresenta uma relação dos requisitos que serão atendidos neste trabalho e na seção 2.3, será

---

1 *Application Programming Interface* - conjunto de funções e/ou classes de objetos disponibilizados para uso do programador

apresentada a arquitetura geral do SIT, proposta para atender às exigências expostas na seção 2.2.

2.1 Requisitos exigidos para um sistema de gerenciamento de informações não estruturadas

Esta seção descreve, um a um, todos os requisitos considerados fundamentais para um sistema de gerenciamento de informações não estruturadas.

2.1.1 Requisito Geral do Projeto

O primeiro e mais importante requisito a ser imposto é que flexibilidade não deve implicar em complexidade. Se for desejado algo simples, então deve ser simples de resolver; quando se desejar algo mais complexo, o sistema pode exigir mais do usuário.

2.1.2 Abrangência

O sistema deve suportar consultas a dados textuais através de uma linguagem de recuperação. Deve ser possível recuperar informações através de *strings*, *substrings*, *valores numéricos* e *datas*, além de outros recursos que a linguagem puder oferecer.

A biblioteca deve oferecer suporte a vários tipos de dados, como datas e números, além de texto, mas deve ser voltada principalmente para a gerência de informações textuais. O foco maior será o armazenamento e, sobretudo, a recuperação de informações desse tipo. Portanto, sempre que houver um impasse de decisão, a parte textual será favorecida.

A necessidade de se tratar tipos de dados como números e datas aparece no momento de uma consulta que não pode ser satisfeita com tratamento apenas sobre o tipo texto. São consultas que envolvem intervalos entre datas, valores resultantes de cálculos, etc.

A linguagem de recuperação textual a ser oferecida deve disponibilizar recursos como formação de palavras através de *metacaracteres*, busca fonética de palavras e busca por aproximação, entre outros. A especificação completa da linguagem de recuperação textual pode ser obtida em [BUSSMANN95], que trata de uma biblioteca especializada em indexação e recuperação de informações não estruturadas.

2.1.3 Indexação e armazenamento flexíveis

Já que a biblioteca suporta mais de um tipo de dado, os sistemas de armazenamento e indexação devem sem flexíveis. Isso significa que esses sistemas

não devem ser "amarrados" a um só tipo de dado. Durante o processamento, os tipos de dados devem ser selecionados automaticamente.

Dessa forma, durante a indexação de um texto, por exemplo, o SIT deve reconhecer números e datas e indexá-los de maneira que torne possível a recuperação por intervalos.

2.1.4 O servidor proposto funcionará em três tipos diferentes de ambiente: monousuário, multiusuário e cliente/servidor.

Cada um desses ambientes e seus requisitos é descrito a seguir.

2.1.4.1 - Ambiente monousuário

Nesse tipo de ambiente não é necessário implementar nenhum tipo de controle de concorrência, pois não há múltiplos acessos simultâneos aos dados. Esta versão do SIT trabalha apenas com um cliente (aplicação cliente), o que garante unicidade no acesso às informações.

A interação entre o servidor (SIT) e o cliente (aplicação) pode ser implementada de pelo menos três formas diferentes:

- *O servidor pode ser uma biblioteca de ligação estática com a aplicação*

Essa é a forma mais simples de se implementar uma biblioteca. Trata-se de um conjunto de funções e/ou objetos já compilados e agrupados em um arquivo que possui a terminação LIB. Esse arquivo é usado por uma aplicação que deseja utilizar aquelas funções/objetos que lá estão. O código da LIB é então ligado ao da aplicação, de forma que é gerado um único arquivo executável, contendo os dois conjuntos de código.

Esse tipo de biblioteca é aceitável na maioria dos ambientes operacionais (UNIX, Windows, DOS, etc.), mas possui a desvantagem de ter seu código agregado ao código de cada aplicação que a utiliza. Isso significa que as aplicações têm seus executáveis grandes (seu próprio código + código da biblioteca) e que o sistema operacional não vai reutilizar o código da biblioteca, o que implica em aumento no consumo de memória. Outro fator negativo no uso desse tipo de biblioteca é que, quando se faz uma alteração no seu código, todas as aplicações que a utilizam devem ser regeradas (religadas).

- *O servidor pode ser uma biblioteca de ligação dinâmica com a aplicação*

Essa forma de biblioteca é mais interessante do que a anterior. Conhecidas como DLLs (Dynamic-Link Libraries), elas também são conjuntos de funções e/ou objetos compilados e agrupados em um arquivo (que possui terminação DLL).

DLLs são formas de bibliotecas muito utilizadas no ambiente Windows. Elas reduzem o tamanho das aplicações, pois não há a necessidade de agregação dos dois códigos, como no caso das LIBs. Quando uma aplicação é executada, o sistema operacional invoca, automaticamente, as DLL necessárias ao seu funcionamento. Para outras aplicações que utilizam a biblioteca, o sistema operacional não executa uma cópia do seu código, mas o reutiliza. Somente quando a última aplicação usuária de uma DLL finaliza sua execução é que o sistema operacional descarta a biblioteca. Tal ligação dinâmica tem pelo menos duas enormes vantagens em relação à implementação de bibliotecas estáticas: primeiro, o ganho de memória, pois não há uma cópia da biblioteca em memória para cada aplicação; segundo (essa, do ponto de vista de um programador), quando a biblioteca sofrer alteração em seu código, não há a necessidade de se recompilar todas as aplicações que a utilizam, pois a ligação é dinâmica.

- *o servidor pode ser um executável e se integrar com o cliente através de um mecanismo qualquer de IPC*[2] [STEVENS90].

Neste caso, o servidor seria uma aplicação e não uma biblioteca. A comunicação entre o servidor e o(s) cliente(s) seria através de algo do tipo memória compartilhada ou algum protocolo de rede.

---

2 IPC significa *InterProcess Communication* (Comunicação entre Processos). Determina uma série de técnicas utilizadas para implementação de comunicação entre dois ou mais processos.

2.1.4.2 - Ambiente multiusuário

A implementação de um ambiente deste tipo é bem mais trabalhosa do que a implementação do ambiente visto no item anterior. A idéia básica é que haja múltiplos servidores e clientes em máquinas diferentes com a possibilidade de acessar, ao mesmo tempo, as mesmas bases de dados.

Essas bases podem até estar em uma máquina da rede que nem possui servidor ou cliente, mas funciona como um servidor de arquivos (Netware, por exemplo). É certo que isso também pode acontecer em ambientes monousuário. O problema aqui é que vários servidores podem estar acessando as mesmas bases simultaneamente; no ambiente monousuário, apenas um servidor vai estar acessando as bases.

A grande preocupação para o ambiente multiusuário é o controle de concorrência. Os múltiplos acessos simultâneos aos mesmos arquivos podem gerar inconsistências no conteúdo das bases de dados.

O sistema proposto deve ser capaz de gerenciar o acesso simultâneo aos dados e permitir o uso de sistema de arquivos distribuído.

2.1.4.3 - Ambiente cliente/servidor

Essa arquitetura deixa mais fortes os conceitos de servidor e cliente. O SIT deve ser, para este caso, uma aplicação executável, capaz de gerenciar por si só todas as bases de dados da máquina onde estiver rodando. Os clientes podem estar na mesma máquina ou em outras espalhadas pela rede. A comunicação entre os processos deve ser implementada através de algum protocolo de rede já conhecido.

As maiores preocupações para esse ambiente são:

- *minimização de tráfego*: o protocolo de comunicação entre os processos deve prever o mínimo de tráfego possível na rede. O fluxo muito alto de informações é indesejável, pois degrada o rendimento de toda a rede e pode inviabilizar o uso de redes a longa distância, cujas velocidades são menores que as de redes locais.
- *minimização de estado*: na arquitetura cliente/servidor o problema de robustez dos processos se torna mais crítico do que em outras. Nesta arquitetura, o processo deve se preocupar com a sua recuperação em caso de falha e ainda dar condições para que outros se recuperem de maneira eficiente. Muitas vezes, o processo de recuperação de falhas é feito com base em *estados* que são guardados pelos processos, mas isso traz implicações indesejáveis, que serão discutidas no tópico que trata de robustez, mais adiante.
- *maximizar desempenho com API assíncrona*: nessa arquitetura é possível implementar, no servidor, atendimentos assíncronos[3] [STEVENS90] aos

[3] O esquema de atendimento assíncrono permite que o servidor receba um pedido e, imediatamente, retorne o controle para o cliente, enquanto fica processando. Após concluir a tarefa, o servidor avisa ao cliente para que ele receba os dados de retorno. Existem diversas técnicas que podem ser

pedidos vindos dos clientes. Dessa forma os clientes podem ser mais eficientes, fazendo o pedido ao servidor e já liberando o usuário para outra tarefa, mesmo que o servidor ainda não tenha terminado de atender o pedido (veja mais detalhes sobre API assíncrona no tópico que trata de desempenho, mais adiante).

2.1.5 O SIT deve ser um sistema portável

O sistema proposto deve ser implementado de forma a minimizar todas as dependências que venham a surgir. Essas dependências podem ser de hardware, de compilador, sistema operacional, etc.

A seguir, estão enumerados alguns dos pontos de dependência que merecem considerações:

2.1.5.1 - Plataforma Operacional

A plataforma operacional deve ser escolhida de forma a atender as seguintes exigências:

- *proteção de memória*: cada processo deve ser executado em seu próprio espaço de endereçamento, sem interferir no funcionamento dos outros.

implementadas para que o servidor retome a conversa com o cliente após um atendimento assíncrono.

- *memória virtual*: os processos podem enxergar uma quantidade de memória maior do que a existente "fisicamente" na máquina, sem preocupação com o esquema de *swap* de dados entre a memória física e o disco.
- *esquema flexível de alocação de memória*: o processo não tem que se preocupar com segmentação de memória, limite de segmentos, etc.
- *disponibilização de threads*: para a implementação de atendimento assíncrono e para implementar a versão cliente/servidor, o processo deve poder contar com o suporte a *threads*[4] [SANTOS93], a partir do sistema operacional. Veja mais detalhes sobre esse recurso no tópico que trata de desempenho, mas adiante.
- *mecanismo eficaz de IPC*: o sistema operacional deve disponibilizar recursos para a comunicação entre processos, como memória compartilhada por exemplo.
- *serviços diversos*: como se trata de um gerenciador de banco de dados, serão necessários recursos como travamento de arquivos, compartilhamento de memória, *timers*, e outros.

Apesar das exigências feitas acima, o SIT não impõe algumas coisas, como um processador de altíssimo desempenho, suporte para um grande número de arquivos abertos, interface gráfica, etc.

---

[4] Um *thread* representa o agente de controle de uma seqüência de instruções executadas por um programa. Um processo é composto de um ou mais threads, que executam em paralelo, compartilhando o mesmo espaço de endereçamento.

As possíveis plataformas para a implementação do SIT são:

- Windows-95
- Windows-NT
- Unix
- Netware (esta é uma plataforma possível, mas não enfocada neste trabalho)

Os ambientes Windows 3.1 e Windows 3.11 foram descartados por não atenderem a todos os requisitos citados acima. Eles não oferecem recursos como *proteção de memória, esquema flexível de alocação de memória* e *threads.*

2.1.5.2 - Esquemas de IPC/redes variados

Para minimizar dependências, o SIT deve ser implementado sobre uma camada de isolamento que permita usar qualquer protocolo de comunicação, como TCP/IP, NETBEUI, OSI, etc.

Para tanto, torna-se necessário programar no mais alto nível possível. Um esquema do tipo RPC deve ser usado, mantendo portabilidade.

Nas versões monousuário e multiusuário do SIT, pode ser usado um mecanismo de IPC mais rápido do que RPC, por exemplo, pois não há a necessidade de comunicação via rede. Uma opção é o uso de memória compartilhada.

2.1.5.3 - Ferramentas de Desenvolvimento

Aqui entra em questão a dependência de ambientes de desenvolvimento. Existem várias ferramentas, cada uma oferecendo recursos diferentes das outras. O grande problema de se usar esses recursos avançados é que o código fica "amarrado" à ferramenta, impossibilitando o transporte para outros ambientes.

As opções de ferramentas para o desenvolvimento do SIT para ambiente Windows são: Borland C++, Microsoft Visual C++ e Watcom C/C++, que são as mais conhecidas. Para o ambiente Unix não serão citadas ferramentas, pois neste trabalho o SIT será implementado em ambiente Windows (NT e 95)

2.1.6 Segurança

Este é, sem dúvida, um aspecto de grande importância para qualquer sistema de banco de dados.

O subsistema de segurança do SIT deve observar aos seguintes requisitos:

2.1.6.1 - Requisito Global do Projeto

- O primeiro requisito a ser imposto aqui é o requisito global do projeto, citado no início da seção 2.1.

2.1.6.2 - Gerenciamento de Usuários

- O sistema deve cadastrar e gerenciar os usuários que terão acessos às bases de dados. Com isso, o SIT será capaz de administrar as permissões de acesso que cada usuário possui sobre as bases.

2.1.6.3 - Flexibilidade na alocação de autorizações

- Alocar autorizações para usuários deve ser uma tarefa flexível. O sistema não deve exigir que o usuário configure níveis de permissão ou coisa parecida. Por outro lado, se for necessário definir um conjunto de regras de acesso aos dados ou delegar poderes para usuários, isso deve ser permitido.

2.1.6.4 - Não incluir esquema de "Deus global"

- O SIT deverá adotar uma maneira segura de definir permissões para os usuários. Não deve ser permitido, por exemplo, que cada um defina o que pode e o que não pode fazer com as bases de dados. Por outro lado, não é desejável a presença de um *super-usuário*, que tenha poderes sobre todos os outros. Isso é ruim, pois tudo e todos passam a depender de um único usuário, que pode cadastrar outros, removê-los, delegar poderes, alterar senhas, etc. Além do mais, a noção de *super-usuário* acaba se tornando um ponto fraco no sistema para possíveis intrusos. O sistema deve implementar um meio termo, onde as permissões sejam gerenciáveis de maneira segura, mas não deixe os recursos concentrados nas mãos de um único usuário.

2.1.6.5 - Simplicidade de gerência

- A gerência de todos os ítens de segurança deve ser feita de maneira simples. O SIT deve possuir esquemas internos de gerência para os casos em que o usuário não esteja interessado em administrar segurança.

2.1.6.6 - Esquema preparado para ambiente distribuído

- Todo o esquema de segurança do SIT deve estar preparado para ambiente distribuído, onde as possibilidades de furos são maiores. O sistema deve se preocupar com problemas de ataque de segurança em rede.

2.1.6.7 - Mecanismo de proteção de bases

- A última exigência feita em termos de segurança é que o SIT implemente mecanismos de proteção para as bases de dados, para evitar que elas sejam transportadas e usadas em outros ambientes, sem administração adequada. O sistema deve possuir meios de evitar a cópia das bases de dados.

2.1.7 Robustez

Um dos maiores problemas encontrados pelos programadores é a implementação de recuperação de erros. Às vezes não é possível nem sequer determinar em que situações eles podem acontecer, principalmente quando falamos de sistemas distribuídos. O fato é que eles ocorrem e é necessário que o software seja robusto o suficiente para se recuperar.

O Servidor de Informações Textuais deverá oferecer mecanismos de recuperação ou reconstrução de bases e índices que, por algum motivo, tenham ficado inconsistentes.

É natural que haja algum tipo de falha no sistema, independentemente de sua arquitetura. Em ambientes monousuário, há o risco de queda de energia ou de pane no sistema operacional ou até no hardware. Isso pode acarretar destruição parcial dos arquivos abertos, danificação do sistema de índices de uma base, etc. Portanto, o SIT deve se precaver contra essas possibilidades e prover um esquema de autorecuperação.

Para os sistemas distribuídos os riscos são bem maiores. Além das mesmas situações descritas acima, esses sistemas podem sofrer "falhas parciais" [SANTOS93], que são provocadas pela queda de um ou mais módulos envolvidos (clientes ou servidores).

Alguns sistemas guardam estado para implementar a recuperação de falhas parciais. A existência de estado implica no consumo de recursos do sistema e em *overhead* para as aplicações. O processo que trabalha seguindo esse esquema deve estar sempre atualizando informações de estado em um meio de armazenamento não volátil (disco, por exemplo) a fim de poder se recuperar após uma queda. A recuperação implica em ler o estado e atualizar as estruturas internas, para que o processo se comporte como estava antes da queda.

Guardar estado requer cuidados, principalmente durante sua atualização. Se o processo cair durante a gravação das informações de estado, elas podem ficar danificadas e impedir a correta "retomada" da aplicação após a queda. A sincronização de imagem das estruturas de memória com as do disco requer velocidade e certeza de que elas vão estar realmente em disco, aos invés de estar nos buffers do sistema operacional. Isso implica em forçar gravações não buferizadas, o que degrada ainda mais a performance do sistema.

Existem alternativas para a implementação de recuperação de falha que não necessitam de manutenção de estado em meio não volátil [SANTOS93]. Uma delas é, no processo de recuperação, perguntar ao outro lado (o cliente pergunta ao servidor e vice-versa) todas as informações necessárias para a reconstrução das estruturas internas. Neste caso, existe a vantagem de não ser necessária a atualização constante em disco das informações de estado. Essas informações são naturalmente transmitidas pela rede e um lado (cliente ou servidor) é capaz de fornecer informações necessárias para a recuperação do outro. Uma das desvantagens, no entanto, é que normalmente é necessário o uso de *broadcast* para que um processo identifique os outros na rede. Isso limita os processos a trabalharem apenas em redes locais, pois o *broadcast* só é aplicável nesse tipo de ambiente.

O ideal seria que o processo não precisasse manter estado, o que significa que, para se recuperar de uma queda, bastaria entrar no ar novamente e tudo se procederia como se nada tivesse acontecido. Mais informações sobre o uso de estado em ambientes distribuídos podem ser encontradas em [SANTOS93] e em [ALSINA94].

Infelizmente, não se pode determinar tão facilmente que um servidor de banco de dados simplesmente não guarde estado. Tudo dependerá das necessidades na implementação. O requisito é: minimizar o uso de estado.

Toda essa discussão sobre o uso ou não uso de estado tem um objetivo: a recuperação de queda no sistema, seja por parte do servidor ou do cliente. Ambos os módulos podem tomar diversas atitudes para a ressincronização, dependendo das informações que são guardadas como estado.

Na ocorrência de uma falha, o sistema deve prever recuperação automática sempre que for possível. O usuário só deve interferir em casos extremos.

Os erros fatais devem ser minimizados, mas se acontecerem, o sistema deve elaborar um relatório da situação a fim de ajudar na recuperação.

2.1.8 Desempenho

Além de tudo o que já foi mencionado anteriormente, o SIT deve atentar para a questão de desempenho. O sistema deve utilizar todos os recursos disponíveis no ambiente operacional, mas sem afetar os requisitos citados acima, como portabilidade, segurança, etc. Abaixo estão descritas as alternativas consideradas para aumentar o desempenho do SIT.

1 - Uma das alternativas para melhorar desempenho de sistemas desse tipo é o uso de *threads*. Um *thread* representa o agente de controle de uma seqüência de instruções sendo executadas por um programa. Um processo que possui arquitetura *multi-thread* é composto por um ou mais *threads* que são executados em paralelo, compartilhando o mesmo espaço de memória [SANTOS93].

2 - Ainda em relação a sistemas distribuídos, existe o problema de travamento de recursos como arquivos e memória compartilhada. Dado que mais de um processo pode estar querendo atualizar os mesmos arquivos/regiões de memória ao mesmo tempo, faz-se necessário implementar um esquema de região crítica para evitar inconsistências. O processo (ou *thread*) que desejar atualizar uma informação em um desses recursos deve realizar um travamento (através de primitivas do sistema operacional), depois fazer a atualização e só então liberar o recurso. Isso garante a integridade das informações. A

granularidade do travamento deve ser escolhida de forma a maximizar o desempenho.

3 - No que se refere à transferência de informações pela rede, o SIT deve se preocupar em transferir sempre o mínimo possível. Isso significa que deve ser definida uma granularidade adequada para que o fluxo de informações não seja grande.

Para as versões do SIT que não são cliente-servidor, pode-se implementar um mecanismo de IPC mais rápido do que o usado para comunicação pela rede. Nessas versões só existe comunicação entre os módulos que estão na mesma máquina, dispensando mecanismos de rede.

4 - O último requisito sobre desempenho para o SIT, é que seja utilizada, sempre que desejável, uma API assíncrona, a fim de liberar o cliente para outras atividades. Isso significa que o sistema deve fornecer meios de atender a um pedido do cliente e imediatamente o liberar para outras coisas, enquanto tenta atendê-lo. Isso é prático, mas traz algumas dificuldades na implementação da aplicação cliente. A maior delas é o tratamento do final de processamento do servidor: quando e como o cliente descobre que o servidor terminou de tratar o pedido que foi feito há um tempo atrás? Outro ponto importante é a obtenção de status: será que houve erro? Qual?

Tanto o servidor quanto o cliente têm que implementar mecanismos de sincronização para detectar término de processamento do servidor. Uma alternativa é o servidor devolver uma espécie de *handle* para o cliente quando este último fizer uma chamada assíncrona. Periodicamente, o cliente pergunta ao servidor se o pedido identificado pelo *handle* já terminou e obtém o status da operação no final para verificar se houve erro.

Outra alternativa é o uso de funções *callback* pelo cliente. Isso permite que o cliente indique uma função sua que deve ser chamada pelo servidor quando este terminar de tratar um determinado pedido.

Devido ao aumento de complexidade inerente em operações assíncronas, tanto para o cliente como para o servidor, é recomendado que a API assíncrona não seja implementada em todos os pontos do SIT, até porque isso não é necessário. Basta implementá-las em pontos onde o uso da assincronia seja mais útil, como na gravação/indexação de um registro, por exemplo.

Os casos considerados úteis são basicamente aqueles em que o servidor precisa de tempo para a execução e o cliente não precisa ficar esperando. Quando a operação é rápida no servidor, não é necessário implementá-la de forma assíncrona; mas as operações lentas prendem o cliente, enquanto ele poderia estar fazendo outras coisas, como edição, impressão, etc.

2.1.9 Internacionalização

Para que o SIT possa ser utilizado em qualquer linguagem, é necessário que ele esteja acompanhado de um bom esquema de internacionalização.

Os requisitos básicos para este tópico são:

1 - Suportar conjuntos de caracteres internacionais
2 - Suporte (na versão definida por este trabalho) apenas para o padrão de caracteres ISO8859/1, dando suporte efetivo para todas as línguas da Europa Ocidental
3 - Tornar possível (e fácil) o suporte total para Unicode. Com este conjunto de caracteres, as linguagens orientais, arábicas, etc. seriam também suportadas.
4 - Suportar vários formatos de data, hora e dados numéricos em geral (questões de ponto decimal, vírgula, moeda, etc.).

2.1.10 Flexibilidade/ortogonalidade

Este é o último conjunto de exigências feitas para a implementação do SIT. Aliás, é um conjunto de um único requisito: "não re-inventar a roda". Isso significa que o sistema proposto deverá reutilizar suas estruturas sempre que for possível. Eis um exemplo onde este requisito seria aplicável: no tópico sobre segurança foi citada a necessidade de se cadastrar os usuários que terão acesso às bases gerenciadas pelo SIT. Esse cadastro de usuários deve ser implementado utilizando

todos os recursos de segurança e facilidades de uso disponíveis no próprio sistema, desde que a portabilidade da ferramenta não seja comprometida. Então, por que não implementá-lo como uma base de dados? Fazendo assim, todo o esquema de segurança, robustez, portabilidade, gerência, etc. é automaticamente aproveitado. Outros tipos de informações de controle/administração que se fizerem necessários ao longo da implementação também devem seguir estes passos. Assim, o sistema se torna mais flexível e menos complexo.

2.2 O que será implementado neste trabalho

Os requisitos citados acima deverão ser aplicados ao sistema proposto. Porém, neste trabalho, apenas um subconjunto do SIT será realmente implementado, ficando o restante para trabalhos futuros.

A relação abaixo apresenta os ítens que serão implementados agora:

- abrangência
  - todos os tipos de dados propostos
  - resolução de impasse de decisão
  - linguagem para recuperação textual
- indexação e armazenamento flexíveis
- servidor apenas para a versão monousuário
  - sem controle de concorrência
  - ligação com a aplicação cliente através de DLL ou algum mecanismo de IPC (decisão a cargo da implementação)

- portabilidade
  - implementação será feita em um sistema operacional que disponibilize:
    - proteção de memória
    - memória virtual
    - esquema flexível de alocação de memória
    - mecanismo eficaz de IPC
    - esquema de travamento
  - o sistema escolhido foi o Windows-NT, por apresentar as características acima e por estar disponível para a implementação
- ferramenta de desenvolvimento
  - a ferramenta escolhida foi o Visual C++ 2.0, da Microsoft
- segurança
  - flexibilidade não deve implicar em complexidade
  - gerenciamento de usuários com controle de autenticação
  - flexibilidade na alocação de autorizações
  - não inclui esquema de super-usuário, que pode mandar em tudo
  - simplicidade de gerência
  - mecanismos de proteção de bases
- robustez
  - implementar reconstrução de bases e de índices
- desempenho
  - granularidade adequada para o travamento de recursos
  - utilizar esquema de travamento de arquivos que não ocupe cpu

- internacionalização
  - nenhum suporte embutido para línguas não Europeias
  - suporte apenas para o padrão ISO8859/1
- flexibilidade/ortogonalidade
  - cadastro de usuários é uma base
  - dados de administração/estatísticas também é uma base

Para trabalhos futuros, os seguintes requisitos devem ser atendidos:

- servidor para a versão multiusuário
  - implementar controle de concorrência
  - permitir uso de um sistema de arquivos distribuído
- servidor para a versão cliente/servidor
  - minimização de tráfego
  - minimização de estado
  - maximização de desempenho com API assíncrona
- portabilidade
  - implementação em um sistema operacional que disponibilize *threads*
  - implementar esquemas de IPC/redes variados
    - camada de isolamento para usar qualquer protocolo
    - programação em mais alto nível possível
    - para as versões monousuário e multiusuário: usar IPC mais rápido que uma camada de rede (ex.: memória compartilhada)
- robustez

- tecer considerações sobre falhas em ambiente distribuído
- minimizar manutenção de estado
- implementar procedimentos de ressincronização após volta de um cliente ou servidor
- implementar sincronização de imagem em disco (fazer flush de estruturas mantidas em memória)
- implementação de recuperação de falhas
  - minimizar erros fatais
  - no caso de ocorrência de erros fatais, emitir relatório da situação para ajudar

- desempenho
  - usar *threads*
  - manter granularidade pequena nas transferências de dados entre cliente e servidor
  - usar IPC mais rápido em versões não cliente/servidor
  - implementar API assíncrona para liberar o cliente
- internacionalização
  - utilizar conjunto de caracteres internacional
  - implementar suporte total para Unicode

2.3 Definição geral do SIT

Esta seção apresenta a arquitetura geral do SIT e tópicos relacionados com esquemas de segurança, tipos de dados e indexação.

2.3.1 Arquitetura Geral do SIT

Para a versão monousuário (ver tópico 3.1.4), o SIT possui uma arquitetura simples, como ilustrada na figura 1.

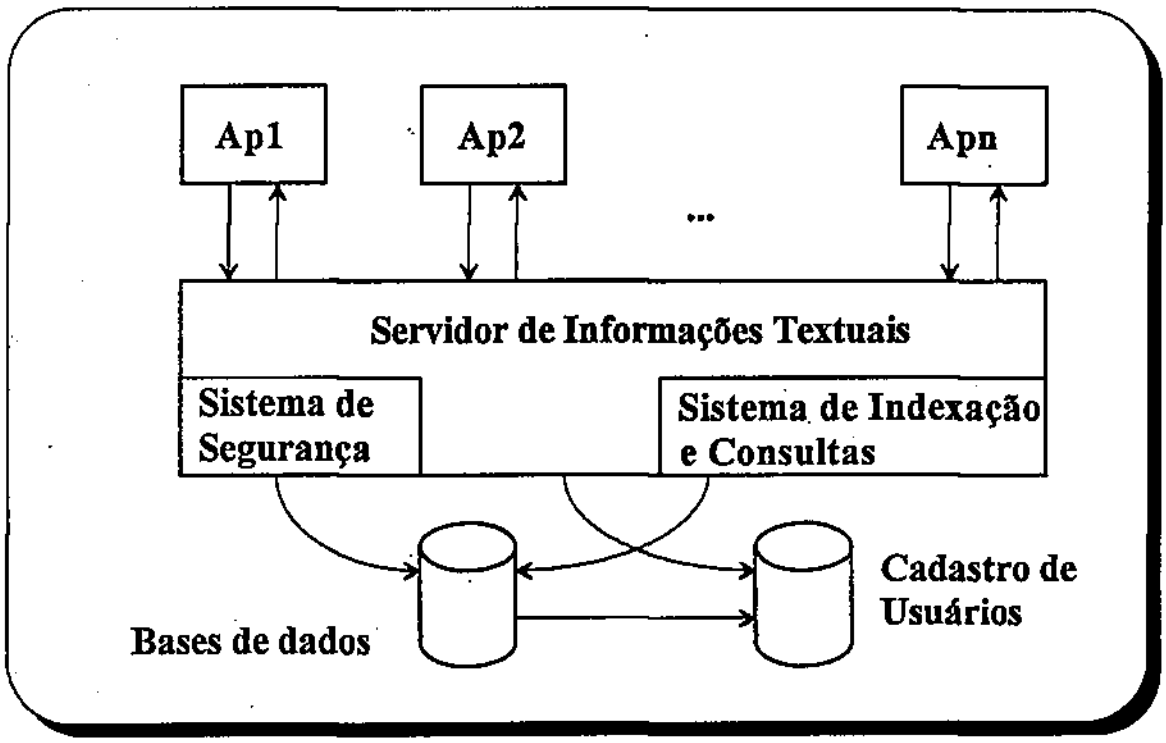


Figura 1 - Arquitetura Geral do SIT

As aplicações *Ap1, Ap2, ... Apn* se comunicam com o SIT, que gerencia todos os acessos às bases de dados, juntamente com o sistema de indexação e consultas. É importante observar que esse sistema de indexação e consultas não é alvo desta dissertação. O SIT apenas o utiliza como base para o tratamento dos índices.

O cadastro de usuários é um conjunto de informações sobre as pessoas que utilizam o sistema. Essas informações são gerenciadas pelo próprio SIT, sem a necessidade de

intervenção de algum processo externo. Esse cadastro está implementado como um conjunto de bases de dados comuns para que o SIT possa reaproveitar todo o seu esquema de segurança, robustez, indexação e acesso aos dados. Isso atende ao requisito citado na seção 2.1.10. Mais detalhes sobre o cadastro de usuários podem ser obtidos mais adiante, na seção sobre Bases de Usuários (tópico 2.3.2).

O sistema de segurança do SIT é responsável pelo processo de *login* de usuários no ambiente e pelo esquema de controle de acesso aos dados, que incluem ACLs[5], senhas e tickets[6] de acesso. A seção 2.3.3, mais adiante, trata do sistema de segurança.

Para a versão multiusuário do SIT, a arquitetura básica pode ser igual à descrita acima, mas para a versão cliente/servidor, ela sofrerá algumas alterações, como ilustra a figura 2. Essas duas versões não são alvos desta dissertação.

---

5 ACL - Access Control List - Lista de Controle de Acessos: São estruturas de dados que determinam acessos de usuários a um conjunto de informações.

6 Ticket - Estrutura que contém informações criptografadas sobre uma transação. É usada para dar maior segurança a transações de rede.

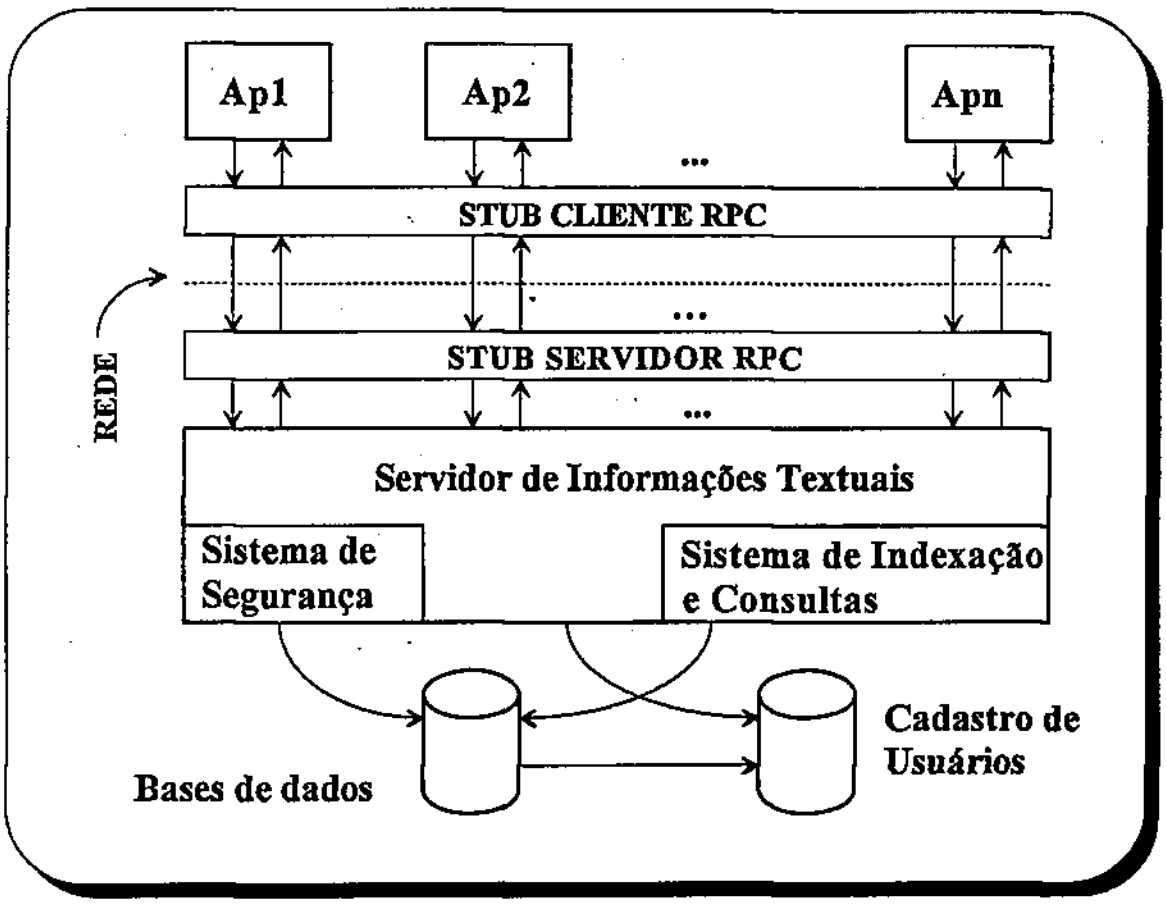


Figura 2 - Arquitetura Geral do SIT para versão cliente/servidor

A diferença básica entre as duas arquiteturas está na implantação dos *stubs* de rede. Esses *stubs* são camadas de software responsáveis pela comunicação na rede. O stub cliente se encarrega de dar transparência para as aplicações, para que elas continuem trabalhando como se estivessem conversando diretamente com o SIT. O stub servidor recebe pedidos vindos do cliente e se comporta como se fosse uma aplicação para o SIT. Para a versão cliente/servidor, nem as aplicações nem o SIT devem sofrer alterações em seus códigos. A simples presença dos stubs deve resolver tudo.

2.3.2 Bases de Usuários

Uma base de usuários (também chamada de UDB - *User DataBase*) é uma base de dados comum, onde o sistema armazena informações sobre as pessoas para controle de segurança. As duas únicas particularidades de uma UDB em relação aos outros tipos de base são:

- o SIT gerencia e manipula a UDB sozinho, sem interferência de um processo ou usuário;
- nenhum usuário consegue acessar diretamente as informações contidas em uma UDB. Isto é, não é possível abrir uma UDB e manipular seus registros, como se faz em uma base de dados comum.

As informações de uma UDB, apresentadas abaixo, são mantidas pelo sistema para controlar os acessos às bases de dados. Cada uma dessas informações é representada por um campo na base.

- Nome do usuário
- Senha
- Descrição do usuário (nome completo, ou alguma informação adicional)
- Tipo do usuário (administrador ou normal)
- Lista de grupos aos quais o usuário pertence
- Data de criação (data em que o usuário foi cadastrado)

- Data da última atualização
- Lista das bases de dados às quais o usuário possui algum tipo de acesso
- Tipo das bases de dados do campo anterior

A API do sistema oferece funções para a manipulação das informações de uma UDB de forma segura. Porém, essa API só permite acesso de leitura aos dados da base. Todas as alterações feitas em uma UDB são comandadas pelo próprio sistema (por exemplo, quando um usuário é cadastrado, um novo registro é criado na UDB).

Quando um usuário realiza a instalação do SIT em sua máquina, ele participa do processo de criação da primeira base de usuários, chamada de *default*[7]. Essa UDB conterá nesse instante apenas um usuário, chamado de GERENTE. A senha para esse usuário é dada pela pessoa que está fazendo a instalação do sistema.

Dessa forma, apenas o GERENTE pode fazer *login* no sistema, pois apenas ele está cadastrado. Para que outro usuário possa acessar o sistema, o GERENTE deverá cadastrá-lo na base de usuários *default*.

Uma vez que um usuário está cadastrado na UDB *default*, ele poderá fazer *login* e criar bases de dados para ele. Todas as bases de dados criadas serão ligadas pelo SIT à UDB na qual o seu criador fez *login*. Isto é, quando um usuário tenta criar uma base de dados, o sistema a associa à base de usuários na qual esse usuário fez login. Portanto,

---

[7] *default* - palavra usada para designar um valor que é assumido automaticamente por um agente quando nenhum outro valor é especificado.

apesar de o usuário ser dono de sua base, ele ainda não possui segurança total, pois o GERENTE tem poderes especiais e poderá acessar qualquer base que tenha sido "amarrada" à base de usuários *default*.

Para eliminar esse inconveniente, é permitido que o usuário crie sua própria base de usuários, onde ele é o dono e, portanto, o gerente. Dessa forma, o usuário poderá fazer *login* na sua UDB e todas as bases de dados que forem criadas serão amarradas a essa base de usuários. Como o usuário em questão é o dono da UDB e o dono das bases que criou, ele possui segurança e acesso total sobre essas bases. E como ele é o dono da base de usuários, ele pode cadastrar quem ele desejar e com isso dar permissão de acesso às bases de dados.

Essa solução atende ao requisito citado na seção 2.1.6, item 4.

2.3.3 Sistema de Segurança do SIT

O SIT é uma ferramenta que propicia um nível de segurança mínimo para quem não quer ter trabalho com esse aspecto. Mas aquele usuário que quer um padrão mais alto de segurança também pode usar esse sistema. Esta seção descreve os elementos básicos que compõem o esquema de segurança do SIT:

- Login
- Ticket

- Características de segurança de uma base de dados
- Passwords
- ACLs

2.3.3.1 - Login

Para ter acesso ao sistema de bases de dados que o SIT gerencia, o usuário deve passar por uma fase de identificação chamada de *login*. O esquema de *login* é bastante simples para o usuário, embora apresente uma certa complexidade interna, requerida para implementar o esquema de segurança.

Para passar pelo processo de *login*, o usuário precisa informar seu nome, sua senha pessoal e uma base de usuários onde ele está cadastrado. O SIT verifica os dados do usuário na UDB indicada e retorna para o usuário um *ticket de acesso* caso a operação obtenha sucesso.

2.3.3.2 - Ticket

Um *ticket* é um objeto que é criado pelo sistema quando um processo de *login* é bem sucedido. Esse objeto contém informações criptografadas sobre o usuário, nome do servidor na rede, nome do cliente, data e hora do processo de login.

Todas as operações de manipulação de bases de dados exigem segurança. Essa segurança é dada pelo *ticket*, que é um elemento de autenticação. Portanto, para todas as

operações sobre as bases de dados, o usuário deve informar qual é o seu *ticket* de acesso. O SIT então faz uma checagem interna para certificar-se de que o usuário possui permissão para realizar o que pretende.

As informações contidas no ticket sobre o servidor e o cliente serão úteis para o esquema de autenticação dos stubs de rede, quando a versão cliente/servidor for implementada.

2.3.3.3 - Passwords

Outro ponto de segurança no SIT é uso de *passwords*. Quando uma base de dados é criada, o usuário informa um conjunto de *passwords* que serão utilizadas pelo sistema no esquema de segurança. Esse conjunto envolve os seguintes ítens:

- *password* de base
- *password* de manutenção de base
- *password* de campo

A primeira é a mais comum e é usada sempre que um usuário tenta abrir uma base de dados. Mesmo tendo passado pelo processo de login, o usuário deve informar essa password para ter acesso a uma base de dados. Com ela o usuário "entra" na base e pode manipular os registros e campos conforme os níveis de segurança definidos pelas ACLs, que serão vistas na seção 2.3.3.5.

A segunda *password* serve para aquele usuário que deseja realizar alterações na estrutura de uma base de dados, como por exemplo, incluir ou excluir um campo. Isso significa que com a password de base, descrita acima, é possível manipular apenas dados. Para alterar a estrutura física de uma base é necessário conhecer a password de manutenção.

A terceira na verdade é um conjunto de *passwords*. Para cada campo de uma base de dados existe uma *password*. Ela poderá ser usada para proteger um campo contra aqueles casos acidentais de exclusão ou alteração de algumas de suas características. Quando o usuário tenta realizar uma dessas operações, ele deve informar a *password* do campo em questão.

2.3.3.4 - Características de segurança de uma base de dados

Todas as bases de dados criadas pelo SIT possuem mecanismos de segurança que, quando aliados aos outros mecanismos descritos nesta seção, tornam todo o esquema mais eficiente.

O primeiro item de segurança de uma base de dados é o seu tipo. Cada base possui um identificador de tipo, que determina níveis de segurança para ela mesma. Quem define esse tipo é o usuário, no momento de criar uma base. Os possíveis tipos de base são:

- Base Pública
- Base Privada
- Base Pública com restrição a registros (chamada de PUB_RR)
- Base Privada com restrição a registros (chamada de PRIV_RR)

Quando uma base é PÚBLICA, todos os usuários possuem livre acesso às informações nela contida; quando ela é PRIVADA, todos os usuários precisarão saber a senha de uso ou estar cadastrados nas ACLs (ver seção 2.3.3.5) para ter o direito de usá-la.

Quando um usuário cria um registro em uma base de dados, o SIT fixa esse usuário como o *dono* do registro. Em bases dos tipos PUB_RR e PRIV_RR o acesso aos registro é determinado por ACLs existentes para cada um deles. Isto é, para cada registro de uma base PUB_RR ou PRIV_RR há um conjunto de permissões. Com isso, um usuário pode impedir que outros acessem seus registros, mesmo estando em uma base pública. Dessa forma, as bases PUB_RR podem ser abertas normalmente, como uma base pública, mas cada usuário possui o direito de atribuir permissões para seus registros; de forma semelhante, as bases PRIV_RR podem ser acessadas por quem souber a senha de uso ou estiver cadastrado nas ACLs, mas alguns registros podem estar protegidos.

Para todos os casos, a senha de manutenção é necessária quando se deseja alterar a estrutura física da base.

Uma base de dados possui mais outro atributo, que é o nome da **base de usuários** à qual ela pertence. Essa UDB é consultada todas as vezes que alguém tentar acessar a base de dados. Há duas finalidades básicas dessa consulta:

- validar o usuário que está tentando abrir a base, através de consulta aos dados cadastrais;
- validar a existência da base de dados, para verificar se realmente ela foi criada onde está ou se foi copiada indevidamente por algum usuário. Esse mecanismo de segurança impede que as bases de dados sejam transportadas para outros locais sem administração adequada, conforme exige o requisito citado na seção 2.1.6, item 7.

2.3.3.5 - ACLs

As ACLs, que significam *Listas de Controle de Acesso*, representam um esquema de segurança bastante poderoso e ao mesmo tempo simples. Um usuário que criar uma base de dados pode delegar poderes para outros usuários através deste esquema de segurança.

Com as ACLs é possível determinar níveis de acesso para uma base, para os campos da base (um a um) e até mesmo para registros da base. Isso significa que um usuário **A**, dono de uma base $\mathbf{B_A}$, poderá informar ao SIT que o usuário **B** possui acesso total à sua base, mas o usuário **C** poderá acessar apenas os campos 1, 3 e 4, e para leitura. Quanto aos registros, suponha que o usuário **A** criou centenas deles, mas há um que é secreto e ninguém poderá vê-lo (nem o usuário **B**, que possui acesso total). Então, o usuário **A** define

uma permissão de ACL para o registro específico, informando ao sistema que apenas ele (A) terá acesso. Os outros usuários, mesmo se fizerem um caminhamento seqüencial pelos registros da base B_A, não verão esse registro.

A tabela I apresenta o conjunto de permissões que o SIT admite.

PERMISSÃO	SIGNIFICADO
READ	o item indicado (base, campo ou registro) só pode ser lido
WRITE	o item indicado poderá ser atualizado
APPEND	o usuário possui o direito de criar um valor, que pode ser um registro ou uma informação de campo.
DELETE	o usuário possui o direito de apagar um item.
NONE	o usuário não possui nenhum direito sobre o item indicado.
ADM	o usuário possui permissão de administrador, o que significa que ele tem todas as permissões possíveis

Tabela I - Conjunto de permissões de ACL

Todas as permissões podem ser combinadas para que o usuário obtenha a permissão desejada.

A figura 3 mostra o formato de uma ACL, que é composta por vários *itens de ACL*, que por sua vez, são formados (cada um) por *elementos de ACL*. Cada *elemento de ACL* contém os campos (ou registros) e as permissões.

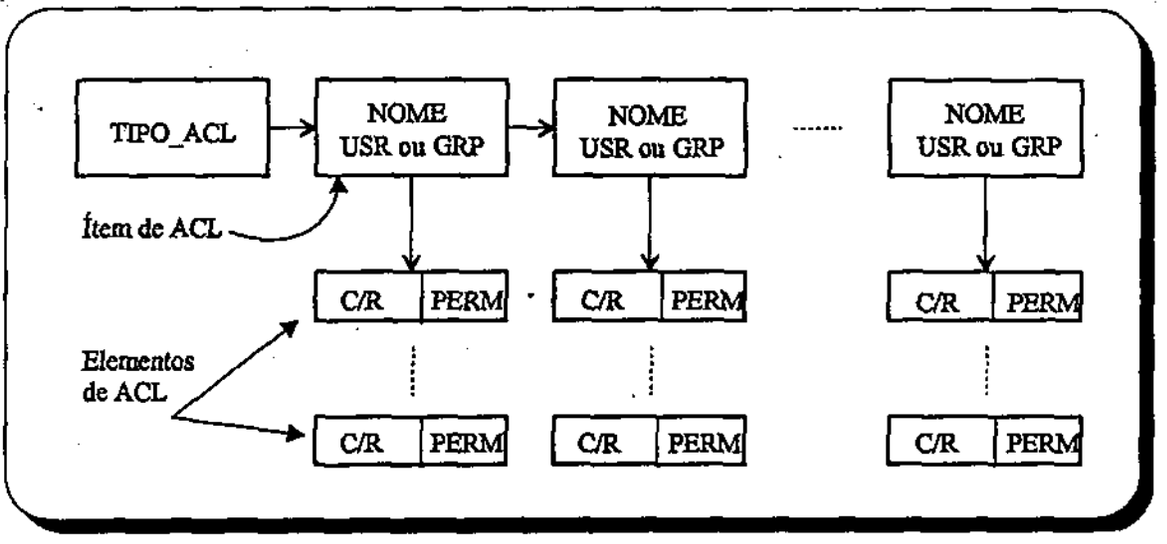


Figura 3 - Estrutura geral de uma ACL

Cada ACL pode ser de um grupo de usuários ou de um usuário. O *TIPO_ACL* indica se a permissão é para usuário ou para grupo. Cada *item de ACL* contém o nome do usuário ou do grupo, conforme *TIPO_ACL*. Para cada *item de ACL*, há uma lista de *elementos de ACL*, que armazenam as permissões associadas a cada campo ou registro.

As informações contidas nos *elementos de ACL* são o identificador do campo (para ACLs de campo) ou o número do registro (para ACLs de registros) mais uma máscara que define a permissão de acesso ao campo/registro referenciado (*C/R* = campo/registro). As possíveis permissões estão descritas na tabela I.

A API do SIT permite que a aplicação manipule as ACLs de tal forma que não é necessário conhecer a estrutura acima. Com uma única função da API é possível definir permissões para qualquer usuário ou grupo de usuários, seja para campo ou para registro.

As ACLs de registros só são usadas nos casos de base pública com restrição de registros e base privada com restrição de registros (veja esses conceitos na seção 2.3.3.4) e só será dada permissão de acesso se o usuário passar pelas restrições do registro.

Todo esse esquema de segurança descrito acima atende aos requisitos impostos nas seções 2.1.1 e 2.1.6, ítens 3, 5, 6 e 7.

2.3.4 Tipos de dados no SIT

O SIT oferece alguns tipos de dados diferentes, representados por tipos de campo que o usuário poderá utilizar em suas bases de dados. A descrição de cada um deles está na tabela II, abaixo.

TIPO DO CAMPO	DESCRIÇÃO
VALUE	o campo pode armazenar valores inteiros de 4 bytes
BYTE	o campo pode armazenar valores inteiros de 1 byte
SVALUE	o campo pode armazenar valores inteiros de 2 bytes
FVALUE	o campo pode armazenar valores de ponto flutuante (float)
DVALUE	o campo pode armazenar valores de ponto flutuante (double)
ALPHA	o campo pode armazenar *strings* de tamanho limitado
TEXT	o campo pode armazenar textos
DATE	o campo pode armazenar datas
TIME	o campo pode armazenar horas

BINARY	o campo pode armazenar informações binárias, que não têm significado para o sistema, mas que o usuário conhece

Tabela II - Tipos de campo manipulados pelo SIT

Cada um dos tipos de campo citados acima possui as seguintes características:

- pode ser multivalorado
- possui um *SLOT*
- contém senha
- identificador numérico único em uma base
- nome de identificação
- tamanho
- índices

Um campo multivalorado é aquele que pode conter mais de um valor (dado, informação) no mesmo registro. Um exemplo típico é o campo *telefone*, de uma base de clientes de uma loja. Cada registro pode conter o nome do cliente (um campo), o endereço residencial (outro campo) e uma lista de telefones (um campo multivalorado).

Um *SLOT* é uma área de disco destinada a armazenar uma informação do usuário. O SIT não tem qualquer controle sobre a informação armazenada em um *SLOT*. Através de funções específicas da API, o usuário poderá gravar uma informação em um *SLOT*, ler essa informação ou consultar o tamanho da informação que está armazenada.

Todos os campos de uma base de dados possuem senhas de acesso, que são definidas no momento da criação da base e servem para dar permissão a um usuário que deseja alterar características dos campos, tais como *nome* e *tamanho*.

Para cada campo que é criado em uma base de dados, o sistema cria um identificador numérico único. Através do identificador de um campo o usuário poderá manipulá-lo. Já o nome de identificação é dado pelo usuário no momento da criação. Esse nome também pode ser usado como referência para manipular os campos.

Os campos do tipo DATE, TIME, VALUE, BYTE, SVALUE, FVALUE e DVALUE possuem tamanhos fixos já definidos pelo SIT. O campo do tipo ALPHA possui tamanho fixo definido pelo usuário no momento de sua criação e os campos do tipo TEXT e BINARY possuem tamanhos variáveis de acordo com o conteúdo armazenado.

Finalmente, para cada campo existente em uma base de dados, poderá haver até nove índices diferentes. Cada um dos possíveis índices é usado para um caso específico e nada impede o usuário de combiná-los conforme suas necessidades. A tabela III apresenta os nove índices que um campo pode ter.

ÍNDICE	SIGNIFICADO
UNIQUETREE	cada valor de um campo é uma chave que só pode ser encontrada uma vez na base de dados. Não é possível gravar uma mesma chave mais de uma vez neste índice.
WORDTREE	cada palavra encontrada em um valor de um campo é indexada neste índice. Isso significa que se o usuário mandar indexar um campo do tipo TEXT ou ALPHA, cada uma das palavras contidas no texto será indexada.
REFERENCETREE	neste índice o SIT armazena uma referência que indica a localização exata de uma palavra dentro da base de dados. Assim, é possível saber em qual parágrafo e frase de um texto pode ser encontrada uma palavra.
DATETREE	este índice armazena valores que indiquem data, mesmo que esses valores estejam entre palavras de um texto.
TIMETREE	semelhante ao DATETREE, só que armazena valores que indicam hora
VALUETREE	também semelhante ao DATETREE, só que armazena valores inteiros e de ponto flutuante. Esses três últimos índices são apropriados para pesquisas de faixa, do tipo: obtenha todas as datas entre 9/9/1957 e 10/5/1991, por exemplo.
BACKTREE	este índice é semelhante ao índice de palavras, com a diferença que, neste caso, as palavras sofrem uma inversão na ordem de seus caracteres antes de serem indexadas. Isso é particularmente útil quando se deseja fazer uma pesquisa do tipo **gia*, onde poderíamos obter *biologia, cardiologia,* etc.

PHONETREE	este índice armazena palavras de forma fonética, o que permite pesquisas do tipo *casa* ou *caza*.
ENTIRETREE	este índice é semelhante ao UNIQUETREE. A diferença é que neste índice é possível armazenar chaves repetidas. O SIT indexa o conteúdo completo de um campo que possui este índice. Isto é, o conteúdo do campo não é "quebrado" em palavras.

Tabela III - Índices usados pelo SIT

Essa variedade de tipos de dados e índices faz com que o SIT atenda aos requisitos citados na seção 2.1.2.

O SIT também atende aos requisitos citados no tópico 2.1.9.

2.3.5 Indexação, Consulta e Listas de Ocorrências

O SIT indexa automaticamente todos os dados de um registro quando ele é gravado. Portanto, a aplicação não precisa se preocupar com esse fator. Mas se houver necessidade de uma performance maior durante a gravação dos registros, a aplicação pode desativar a indexação automática (também chamada de indexação *on-line*). Dessa forma, os registros são gravados e não são indexados. Quando for desejado, a aplicação poderá ordenar que o sistema indexe todos aqueles registros que foram gravados sem indexação.

Há também aquele caso em que é necessário re-indexar todos os registros de uma base de dados devido a danos físicos nos arquivos ou por outro motivo qualquer. O SIT

disponibiliza uma maneira de realizar essa operação, destruindo os índices atuais e construindo outros a partir dos dados da base.

Como visto na tabela III, o SIT disponibiliza vários índices para que a aplicação escolha quais são os mais adequados. Esses índices são montados por um sistema de índices que, como dito anteriormente, não é alvo desta dissertação. Durante o processo de indexação, o SIT descobre automaticamente em quais índices colocar as informações de forma a possibilitar consultas mais eficientes.

Uma vez que os dados estejam devidamente indexados, é possível realizar pesquisas sobre os índices para a obtenção de dados. Como o sistema trabalha com recuperação textual, é possível realizar buscas do tipo "quero todas as ocorrências da palavra *acidente*, que esteja no mesmo parágrafo da palavra *trânsito*". O SIT então fará a busca sobre o sistema de índices e montará o que é conhecido por *Lista de Ocorrências*.

Essa lista representa o resultado da pesquisa realizada sobre o sistema de índices. Ela contém exatamente todas as ocorrências encontradas. Cada ocorrência contém informações sobre a localização da palavra pesquisada dentro da base de dados. As informações de uma ocorrência são as seguintes:

- número do registro
- identificador do campo
- número da repetição dentro do campo

- número do parágrafo dentro da repetição
- número da frase dentro do parágrafo
- número de seqüência da palavra dentro da frase
- palavra encontrada

Essa última informação pode não parecer útil, pois o usuário já sabe qual foi a palavra que mandou procurar. *Mas ela é útil sim.* O usuário pode ter pedido para o SIT procurar, por exemplo, *aciden**. Então, a última informação de uma ocorrência poderia ser:

- acidente
- acidentes
- acidentado
- acidentada
- ...

As outras informações têm significados óbvios e servem basicamente para localização da palavra dentro de um contexto. Por exemplo, a aplicação pode querer brilhar as palavras encontradas dentro de um editor de textos. Com as informações de uma ocorrência é muito fácil fazer isso.

A lista de ocorrências serve para que a aplicação caminhe através dos registros da base. Sempre que for pedido o próximo registro, por exemplo, o SIT responde com o

próximo registro que é indicado pela lista, que pode não ser o próximo registro físico da base de dados.

Esse tipo de lista pode ser obtida de várias formas. A primeira delas é através da realização de uma consulta feita sobre o sistema de índices. Outra forma é carregá-la do disco, o que significa que ela pode ser gravada. Ainda outra forma é através de "batimentos" que podem ser feitos entre duas listas. Por exemplo, é possível obter uma lista que resulte da operação

ListaA *or* ListaB.

A operação *or* é feita sobre as duas listas e uma terceira lista é gerada contendo as combinações das ocorrências de cada uma das listas.

O esquema de indexação e consultas apresentado acima permite que o SIT atenda aos requisitos citados nas seções 2.1.3 e 2.1.7 (parcialmente, pois esta seção não trata só de indexação), além da 2.1.9.

Capítulo 3 - Implementação do Servidor de Informações Textuais

Este capítulo descreve a implementação do SIT, destacando detalhes da sua arquitetura e sua divisão em objetos sob o ponto de vista do desenvolvedor. Não será discutida toda a implementação do sistema, mas apenas os tópicos que merecem maiores considerações. É importante destacar que, devido a complexidade do software, a etapa de implementação foi realizada por uma equipe de 3 programadores.

O sistema foi totalmente desenvolvido em uma linguagem de programação orientada para objetos (C++ - maiores referências sobre essa linguagem podem ser encontradas em [STROUSTRUP93]), o que permite a fácil manipulação de seus recursos por parte dos programadores. Para atender a requisitos como portabilidade, segurança, robustez e desempenho, além de outros, foi utilizada a plataforma de 32 bits Windows-NT para o desenvolvimento do SIT. A versão monousuário do sistema também poderá funcionar em ambiente Windows 3.1/3.11, mas para isso é necessário que se utilize uma camada de software que faça o mapeamento da API de 16 bits desse ambiente para 32 bits. Essa camada de software é conhecida como Win32s [MSOFT92b] e pode ser facilmente encontrada na Internet ou juntamente com os softwares mais novos para o MS-Windows 3.1/3.11 que também rodam em Win32s (compiladores, por exemplo).

A ferramenta de trabalho utilizada foi o Visual C++ 2.0, da Microsoft. Essa ferramenta foi escolhida por apresentar maior quantidade de recursos (que facilitam o trabalho do programador, mas não interferem na portabilidade do código) do que a outra

opção, que era o Borland C++ 4.0. Além do mais, essa outra ferramenta não roda em ambientes 32 bits.

3.1 Arquitetura em detalhes

A arquitetura vista no capítulo anterior está bastante simplificada, escondendo a modularização interna. O SIT é uma DLL que disponibiliza um conjunto de classes para as aplicações. Na versão monousuário um único módulo implementa toda a funcionalidade do sistema.

Uma DLL é uma biblioteca ligada à aplicação dinamicamente. Seu código é compartilhado por todas as aplicações que a utilizam, mas seus dados são replicados. Isso significa que uma aplicação não consegue enxergar dados de outra, o que é bastante natural. Mas o SIT precisa concentrar as informações de memória de todos os clientes em um único lugar para ter maior controle sobre os recursos utilizados. Para concentrar essas informações é necessário implementar um esquema de compartilhamento de memória. Assim, todos os clientes enxergam os mesmos dados do servidor.

Implementar um esquema de compartilhamento de memória requer alguns cuidados e não é tão simples. O fato deo ambiente operacional ser multitarefa complica ainda mais, pois há a possibilidade de acesso simultâneo à mesma área de memória e até aos mesmos dados em disco.

A figura 4 apresenta a arquitetura do SIT com mais detalhes do que foi visto no capítulo anterior.

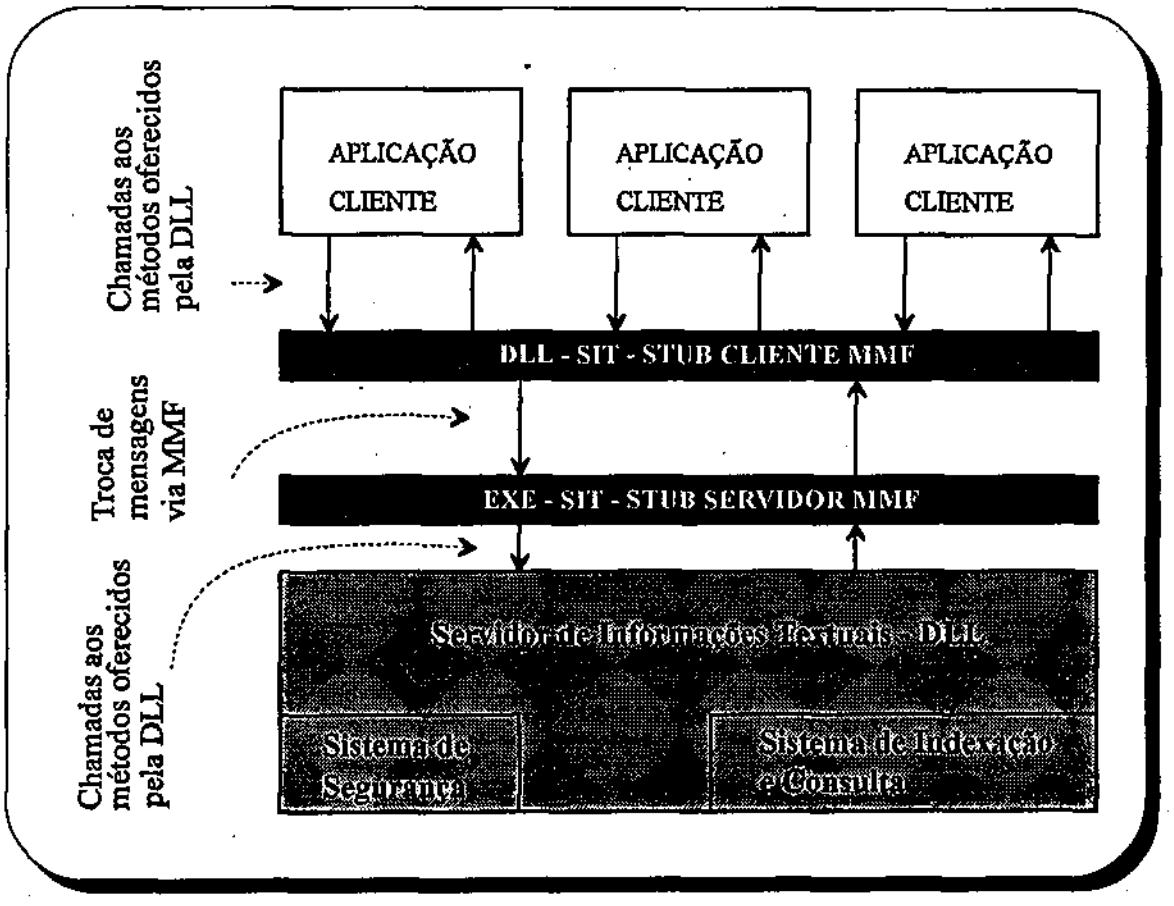


Figura 4 - Arquitetura detalhada do SIT

Para não deixar o trabalho de compartilhamento de memória para as aplicações, o SIT oferece uma outra DLL, que contém todas as funções da biblioteca, mas não faz o trabalho real. Essa DLL é apenas um módulo de compartilhamento de dados do SIT. O *Stub Cliente MMF*, como é chamado, implementa mecanismos de compartilhamento de memória através de um esquema conhecido como *Memory Mapped File* (daí a sigla *MMF*). Esse esquema é oferecido pela API de 32 bits do Windows e implementa arquivos em memória que podem ser compartilhados entre várias aplicações. O SIT o utiliza para que as

aplicações possam compartilhar bases abertas, sessões de login, etc. As aplicações não precisam saber o que está se passando internamente para que os dados sejam compartilhados. É como se estivessem usando o próprio SIT diretamente, sem a camada de compartilhamento de dados. Tudo é transparente.

Não há nenhum armazenamento de dados no *Stub Cliente MMF*. Quem faz todo o armazenamento de informações, assim como o gerenciamento do MMF é outro módulo, chamado de *Stub Servidor MMF*. Esse módulo é o responsável por receber pedidos do Stub Cliente e traduzir em chamadas de funções para o módulo principal do SIT, que é a DLL apresentada na parte inferior da figura.

O *Stub Servidor MMF* armazena todos os dados obtidos em sua área de memória e retorna handles para o *Stub Cliente MMF*, que os repassa para as aplicações. Então, observa-se que o Stub Cliente serve apenas de interface entre o sistema de gerenciamento de memória compartilhada (Stub Servidor) e o módulo principal do SIT. Isso evita que as aplicações tenham que se preocupar com o compartilhamento de dados, mas evita também que elas tenham que usar o SIT de maneira diferenciada. Não há nenhuma diferença entre a API do *Stub Cliente MMF* e a API do módulo principal.

Para fazer a comunicação MMF entre os dois *Stubs* citados é necessário um protocolo. Esse protocolo é o responsável pelo trânsito de dados entre os *Stubs* e será discutido com detalhes mais adiante, na seção 3.6. É bom observar que todo esse esquema

baseado em MMF não é válido para a versão cliente/servidor do sistema, pois a camada de rede oferece todo esse esquema de graça.

3.2 Classes oferecidas pelo SIT

Esta seção descreve as classes que o SIT disponibiliza para as aplicações e as classes que não são oferecidas, mas existem para controle interno. São classes de objetos C++ [STROUSTRUP93] que representam a arquitetura interna do SIT.

- **C_Session**: esta classe é a responsável pelos trabalhos de criação, abertura, fechamento e destruição de bases de dados, cadastramento de um usuário no sistema através de um *login*, descadastramento do usuário (*logout*) e por responder a perguntas do tipo *quais são os usuários que têm acesso à base X*, ou *quais são os servidores que estão "no ar"*. Esta classe possui relacionamento direto com as classes *C_Base*, *C_User* e *C_Ticket*, descritas mais adiante.
- **C_Base**: esta classe é responsável pelo gerenciamento de uma base de dados, uma vez que ela tenha sido aberta. Com um objeto deste tipo o usuário poderá caminhar sobre os registros de uma base de dados, acrescentar registros, removê-los, alterar a estrutura física da base (o número de campos, seus tipos, etc.), realizar operações de indexação e busca de palavras e operações de restrição de acesso à base por outros usuários. Esta classe possui relacionamento direto com as classes *C_Session*, *C_Record*, *C_OcList*, *C_StopWList* e *C_Password*.

- **C_Record**: esta classe é mais simples que as anteriores. Ela é a responsável pela leitura/gravação de um registro na base de dados. Seu relacionamento ocorre apenas com a classe *C_FieldList*.
- **C_FieldList**: esta classe é apenas uma lista de todos os campos de uma base de dados. Sua relação é feita com a classe *C_Field*.
- **C_Field**: com esta classe é possível obter informações sobre cada campo de uma base de dados. Ela armazena as características de um campo de uma base, como por exemplo o tipo de dado a ser armazenado, o tamanho do dado, o seu valor e a quantidade de valores - também chamados aqui de *repetições* - armazenados (para campos multivalorados). Possui relacionamento direto com as classes *C_DataList*, *C_GoWList* e *C_Password*.
- **C_DataList**: é apenas uma lista de todos os valores que um campo possui, em um determinado registro da base. Sua única relação é com a classe *C_Data*.
- **C_Data**: esta classe é bastante simples. Ela simplesmente armazena cada valor de um campo, em um registro de uma base. Isto é, para cada valor (informação) de um campo há um objeto *C_Data*. Não possui relacionamento com outros objetos.
- **C_OcList**: é simplesmente uma lista de objetos do tipo *C_Occurrence*, descrito abaixo.
- **C_Occurrence**: um objeto deste tipo armazena informações sobre a localização exata de uma palavra dentro de uma base de dados. Toda vez que o usuário realizar uma operação de consulta à base, procurando por determinadas palavras, uma *Lista de Ocorrências* será gerada, contendo todas as ocorrências das palavras procuradas dentro da base. Cada ocorrência de uma palavra contém o número do registro dentro da base, o identificador do campo onde ela se encontra, o identificador da repetição onde a palavra

se encontra (para o caso de campos multivalorados) e, dentro da repetição, o número do parágrafo, o número da frase e o número de seqüência da palavra dentro da frase. Desta forma, para cada palavra procurada, haverá uma ocorrência que descreve sua localização exata dentro da base. Um objeto *C_Occurrence* não possui relacionamento com outros objetos.

- **C_StopWList**: é simplesmente uma lista de objetos do tipo *C_StopWord*, com o qual se relaciona.
- **C_StopWord**: esta classe introduz o conceito de uma palavra que não deve ser indexada quando o usuário ordenar a indexação de uma base de dados. Muitas vezes não é desejável indexar palavras como "de", "é" e "mas", pois elas ocorrem com muita freqüência na maioria dos textos e não oferecem muita ajuda na hora de recuperar uma informação. Nestes casos, essas palavras podem ser definidas como *StopWords* e, automaticamente, o SIT as deixará fora dos índices. O objeto *C_StopWord* não possui relacionamento com outros objetos.
- **C_GoWList**: esta classe se comporta de forma semelhante às outras classes *Lista*. Ela simplesmente armazena objetos da classe *C_GoWord*, com a qual se relaciona.
- **C_GoWord**: esta classe faz com que uma palavra possa ser indexada, mesmo que ela seja uma *StopWord* (ver conceito mais acima). As StopWords são casos genéricos de palavras que não devem ser indexadas (são definições de uma base de dados) e as *GoWords* são casos específicos de palavras que são *StopWords*, mas devem ser indexadas (são definições de campo). Esta classe não se relaciona com nenhuma outra.

- **C_User**: esta classe serve apenas para armazenar dados sobre o usuário que abriu a sessão através da operação de *login*. Possui relacionamento apenas com a classe *C_Password*.
- **C_Ticket**: esta é uma classe que tem a finalidade de dar segurança ao usuário do SIT. Ela entra em cena sempre que um usuário realiza a operação de *login* e contém informações criptografadas que servem para autenticação nas demais operações realizadas pelo usuário.

A figura 5, abaixo, apresenta a organização das classes descritas acima.

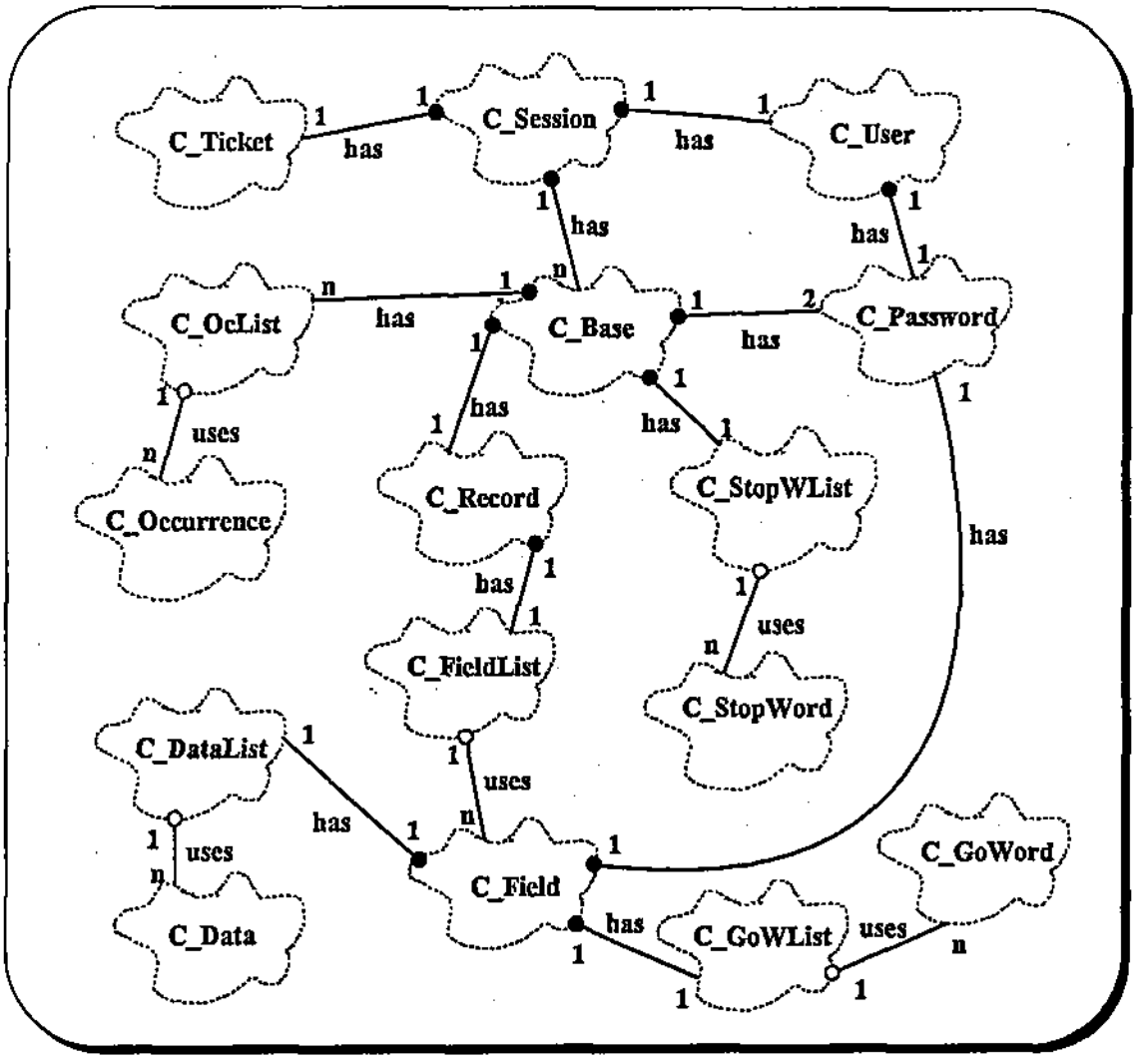


Figura 5 - Organização hierárquica das classes do SIT

A figura acima utiliza a notação de *Booch* [HORSTMANN95], para a diagramação das classes. A maioria das classes apresentadas existe para controle interno da ferramenta. A outra parte é oferecida para as aplicações como uma API. São elas:

- C_Session
- C_Base
- C_Field
- C_Data

- C_Occurrence
- C_Ticket

3.2.1 A classe C_Session

Esta classe é a porta de entrada do SIT. É a primeira a ser utilizada pela aplicação cliente. Ela implementa métodos para:

- Login
- Logout
- Criação de bases
- Destruição de bases
- Abertura de bases
- Consultas à rede (quais são os servidores que estão no ar, quais os servidores que podem atender a um determinado usuário, etc.)
- Criação de UDBs
- Destruição de UDBs
- Controle de cadastro de usuários e grupos de usuários
- Manipulação de senhas de usuários
- Reprocessamento de bases
- Exportação/Importação de bases

3.2.2 A classe C_Base

Esta classe é responsável pelo gerenciamento de uma base de dados, uma vez que ela tenha sido aberta por um método da classe C_Session. Seus métodos tratam dos seguintes assuntos:

- Adição, atualização, remoção, leitura e travamento de registros
- Adição, atualização, remoção e leitura de estrutura dos campos
- Adição, atualização, remoção e leitura de dados
- Manipulação de ACLs
- Manipulação de senhas
- Indexação/desindexação de dados
- Consultas (recuperação de dados)
- Reprocessamento de índices
- Normalização de dados[8]
- Manipulação de Listas de Ocorrências

3.2.3 As classes C_Field, C_Data, C_Occurrence e C_Ticket

São classes com potencial menor do que as anteriores, mas não menos importantes. A classe **C_Field** armazena informações sobre o tipo do dado, tamanho, descrição do campo, senha e uma lista dos dados que estão armazenados. Esses dados são chamados de *repetições* e são representados, cada

[8] O sistema de índices precisa guardar as informações de tal forma que tornem os procedimentos de armazenamento e recuperação consistentes. Uma maneira de fazer isso é, por exemplo, armazenar todos os termos com caracteres maiúsculos e sem acentuação. O processo que realiza essa conversão dos termos é chamado de *normalização*.

um, por um objeto da classe **C_Data**. Esses objetos guardam o dado propriamente dito (textos, números, datas, etc.).

Um objeto da classe C_Occurrence contém informações sobre a localização de um termo (palavra) dentro de um texto. Esse objeto é gerado a partir de uma operação de consulta realizada pelos objetos da classe **C_Base**. Na verdade, essa operação gera uma Lista de Ocorrências, que contém um ou mais objetos da classe **C_Occurrence**. Cada objeto contém as seguintes informações sobre o termo pesquisado:

- Número do registro onde o termo foi encontrado
- Identificador do campo que contém o termo
- Número da repetição dentro do campo (cada campo pode ser multivalorado)
- Número do parágrafo dentro da repetição
- Número da frase dentro do parágrafo
- Número de seqüência do termo dentro da frase
- O termo procurado

O termo procurado é devolvido dentro de um objeto ocorrência porque a aplicação pode realizar pesquisas do tipo *"Maria*"*, o que retornaria termos como *Maria, Mariana*, etc.

Objetos da classe C_Ticket são os mais importantes no esquema de segurança do SIT. A cada operação de login que o usuário realiza sobre um objeto da classe **C_Session**, um ticket é gerado. Esse ticket, representado por um objeto da classe de mesmo nome, contém informações criptografadas sobre o usuário. Essa classe terá sua importância aumentada quando o SIT for implementado para ambientes distribuídos, pois será usado no esquema de autenticação entre clientes e servidores.

3.3 O processo de Login

Um dos processos mais importantes no esquema de segurança do SIT é o **Login**. É a partir desse processo que tudo começa.

O SIT adotou um conceito que ajudou a implementar o esquema de login com a segurança e simplicidade desejadas. Foi criado o conceito de bases de usuários, que são bases de dados gerenciadas pelo próprio sistema e que contêm informações sobre os usuários de um ambiente.

Cada máquina que instalar o SIT terá uma base de usuários *default*, que será usada para o processo de login. O dono dessa base é o administrador do sistema, que possivelmente será a pessoa que fizer a instalação do software em uma máquina. Somente o dono dessa base poderá cadastrar usuários e alterar seus dados. Para evitar que um usuário se sinta incomodado pelo fato de haver alguém que pode ter acesso aos seus dados em uma

base de usuários, o SIT permite que o usuário crie sua própria base de usuários, onde ele será o dono e, portanto, poderá cadastrar quem desejar. Isso atende ao requisito citado na seção 2.1.6.4.

O processo de login pede um nome de usuário, sua senha de login e o nome de uma base de usuários. O nome do usuário é procurado dentro da base de usuários especificada e, se o processo de login o encontrar, a senha também é checada. Se uma das duas informações não se confirmar, o processo falha.

Em caso de sucesso, um objeto da classe *C_Ticket* será criado e retornado para a aplicação. Esse objeto guarda internamente informações sobre o usuário que fez o login e é necessário para todas as outras operações realizadas entre as aplicações e o SIT. Todos os métodos das classes vistas acima checam a validade do ticket antes de prosseguirem com as suas operações.

3.4 O processo de Criação de Bases

O SIT disponibiliza um processo bastante poderoso de indexação e recuperação dos dados do usuário. Utilizando uma biblioteca específica para essa tarefa, a ferramenta cria e gerencia o *Sistema de Índices* de uma base. Esse sistema contém os seguintes elementos:

- Índices
- Parsers
- Listas de Ocorrências

O Sistema de Índices oferecido pelo SIT disponibiliza até nove índices diferentes para a aplicação. Cada um deles possui uma característica própria e pode ser usado em conjunto com os demais. Eis a relação dos índices existentes e suas características:

- **UniqueTree**: serve para aqueles casos em que é desejado ter um campo da base de dados que não admite repetição de seus valores (CPF, por exemplo);
- **ReferenceTree**: este índice simplesmente armazena uma referência que indica a localização exata de uma palavra dentro da base de dados;
- **WordTree**: indexa cada uma das palavras encontradas em um texto, inclusive datas, horas e números. Através deste índice é possível realizar a busca de uma palavra dentre vários documentos que foram indexados;

- **BackTree**: este índice serve para tornar possível buscas do tipo *"*ria"* no sistema de índices. Ele é semelhante ao de palavras, mas os termos são indexados de trás para frente, para agilizar a consulta citada;
- **PhoneTree**: é possível também fazer buscas através dos fonemas das palavras. Isto é, pode-se procurar, por exemplo, "casa" sem precisar saber se a palavra foi indexada como "casa" ou "caza" ou "kasa", etc. Através deste índice é possível definir esse nível de "transparência" nas consultas;
- **EntireTree**: este índice funciona de maneira semelhante ao de palavras, mas ao invés de indexar cada um dos termos encontrados em uma cadeia de caracteres, ele indexa a cadeia completa. Isso faz com que este índice se assemelhe também ao de chave única, mas neste caso aqui é possível indexar mais de uma vez a mesma cadeia, enquanto que no caso de chave única, não;
- **DateTree**: este índice é específico para o armazenamento de datas. Através dele é possível realizar consultas do tipo "todas as datas maiores que 10/7/1974", por exemplo;
- **TimeTree**: semelhante ao anterior, só que para horas;
- **ValueTree**: também semelhante ao anterior, só que para valores numéricos, inteiros ou não;

Os índices acima podem ser associados a cada um dos campos de uma base de dados. Dessa forma, a aplicação decide como quer indexar os campos numéricos, data, hora, etc.

Para indexar um campo numérico ou um campo data ou hora é simples. Mas para indexar um campo do tipo texto, por exemplo, é necessário um trabalho de *"parsing"* sobre o seu conteúdo. Para um campo alfa-numérico o SIT implementa um parser, que é responsável por "quebrar" a cadeia de caracteres em pequenas partes chamadas "*tokens*". Esses *tokens* podem ser as palavras encontradas na cadeia de caracteres ou datas ou horas ou valores numéricos. Para campos do tipo texto (não simplesmente alfanumérico), é necessário que a aplicação construa seu próprio parser, pois o SIT não sabe o que há dentro desse campo. Pode haver caracteres de controle de um editor de textos, por exemplo.

Para oferecer o seu parser, a aplicação deverá construir uma DLL com funções que fazem o processo. Essa DLL será usada pelo SIT no momento da indexação dos dados de um campo.

O outro elemento de um Sistema de Índices é conhecido por *Lista de Ocorrências*. Objetos desse tipo são gerados sempre que a aplicação solicita ao SIT uma consulta ao sistema de índices. O resultado de uma consulta é uma lista contendo informações sobre a localização exata de todas as ocorrências dos termos procurados. Por exemplo, a aplicação solicita a busca da palavra "Maria". O SIT gera uma lista contendo objetos do tipo **C_Occurrence** (veja seção 3.2). Essa lista conterá todas as localizações da palavra pesquisada dentro da base de dados. Cada uma das ocorrências da lista indicará o registro, o campo, a repetição, o parágrafo, a frase e a seqüência da palavra dentro da frase. Com isso é possível localizar a palavra dentro do editor de textos, por exemplo. O SIT oferece métodos para o posicionamento sobre as ocorrências de uma lista ou até mesmo para

posicionamento sobre um registro da base que corresponda a uma ocorrência. Veja mais detalhes sobre consultas no próximo capítulo.

3.5 Restrição de uma Lista de Ocorrências a partir das ACLs

O resultado de uma operação de consulta no SIT é uma Lista de Ocorrências, contendo as localizações de todos os termos encontrados. Também foi visto no capítulo 2 que uma base de dados pode apresentar restrições de acesso a alguns registros, dependendo das ACLs geradas para o usuário que estiver usando a base no momento. Então, o SIT não deve gerar uma Lista de Ocorrências contendo absolutamente todas as localizações dos termos encontrados. É necessário "filtrar" a lista para que aquelas ocorrências que apontam para registros ou campos "proibidos" para o usuário corrente sejam eliminadas antes que a aplicação "ponha a mão" na lista.

Sempre que o processo de pesquisa termina internamente, o SIT faz uma varredura na lista gerada e retira todas as ocorrências que não dizem respeito ao usuário corrente. Essa "filtragem" é feita levando-se em consideração as permissões de ACLs que o usuário possui sobre os campos e registros da base de dados.

Dessa forma, ninguém precisa ficar preocupado com a segurança de suas informações em uma base de dados. Só as verá quem tiver permissão explícita.

3.6 O Protocolo MMF

A figura 4 (item 3.1) apresenta a arquitetura do SIT, mostrando os *Stubs MMF (cliente e servidor)*. Esses dois módulos foram discutidos brevemente, onde foi citada a necessidade de se definir um protocolo para que eles "conversassem" de maneira proveitosa.

Antes da definição do protocolo, é interessante analisar qual será sua tarefa. Primeiro, o *Stub Cliente MMF* deve implementar exatamente todos os métodos públicos de todas as classes que são de interesse das aplicações. Assim, tanto faz usar diretamente a DLL do SIT quanto o *Stub Cliente*. Segundo, é importante pensar em uma forma de trafegar dados das aplicações e do próprio SIT entre os stubs. Terceiro, todas as alocações de dados devem ser concentradas no *Stub Servidor*, para que todas as aplicações consigam enxergar os mesmos dados, quando necessário. Depois, as aplicações devem receber *"handles"* para os dados alocados no *Stub Servidor*, de forma que, se uma aplicação entregar um *handle* para outra, essa outra consiga acessar a mesma informação que a primeira. Por último, o tráfego dos handles entre as aplicações deve ser feito de maneira correta e segura. Para isso, o *Stub Cliente* deve oferecer métodos para transporte de *handles*.

Implementar todos os métodos públicos no Stub Cliente não é tarefa difícil. Basta refazer as classes, com nomes iguais aos originais e definir os métodos, também com nomes iguais. É claro que os métodos não vão fazer o mesmo trabalho que os originais. O trabalho deles é justamente fazer o tráfego de dados entre os Stubs.

O tráfego de dados já é algo mais detalhado, cheio de cuidados especiais. Todos os dados devem, antes de trafegar entre os módulos, ser "empacotados" em uma área contígua de memória. É como se eles estivessem sendo gravados em um arquivo. Mesmo as listas encadeadas e estruturas que têm apontadores para outras devem ser empacotadas em uma área contígua. Feito isso, todo o processo se torna mais simples, apesar da necessidade de muito cuidado. Esse processo de "empacotamento" dos dados, também conhecido como linearização ou serialização, é importante porque possibilita a execução do processo em um único passo. Isto é, se os dados não fossem serializados, seria necessário passá-los um a um para o outro processo e só depois disparar a execução da função desejada. Com a serialização, basta uma chamada ao outro processo, pois todos os dados estarão em um único pacote.

O Cliente deve gravar os dados já empacotados em uma área de memória compartilhada, através de funções especiais do sistema operacional e enviar uma mensagem para o Servidor. Essa mensagem deve indicar qual é o método que deve ser executado e onde estão os dados necessários à execução. Neste ponto o Cliente pára e fica esperando o retorno do Servidor.

O Servidor, ao receber a mensagem de execução, acessa a memória compartilhada e desempacota os dados. Nesse momento é necessário ter cuidado para não gerar uma informação errada, pois todo o processo poderá ir por água a baixo. Depois de desempacotados, os dados devem estar exatamente iguais aos do Cliente (no caso de uma lista encadeada, deverá conter exatamente o mesmo número de elementos e todos os

elementos devem ser iguais aos do Cliente). Assim, o Servidor chama o método da API do SIT, que nesse ponto é a "verdadeira" API, e obtém o retorno.

O retorno do método, que pode ser um valor inteiro, uma cadeia de caracteres, uma lista encadeada, um apontador para um objeto ou outras estruturas, deve ser empacotado juntamente com o código de erro da última operação realizada pela API. O final da execução de um método no lado Servidor é marcado pela devolução do resultado para o cliente, através de funções especiais do sistema operacional. O Servidor volta a funcionar em modo de espera.

O Cliente, que estava "dormindo", recebe o resultado da operação solicitada. Esse resultado é então desempacotado e repassado para a aplicação cliente.

Uma aplicação pode querer repassar para outra um objeto recebido do SIT. O *Stub Cliente* disponibiliza meios práticos para essa transferência de objetos entre aplicações para que elas não precisem implementar um protocolo MMF, ou DDE, ou outra coisa qualquer. A aplicação que quer transferir dados chama um método específico do SIT, que trata de empacotá-los e repassá-los para a outra aplicação. Essa última, por sua vez, chama outro método do sistema, que vai desempacotar os dados e lhe entregar um ponteiro.

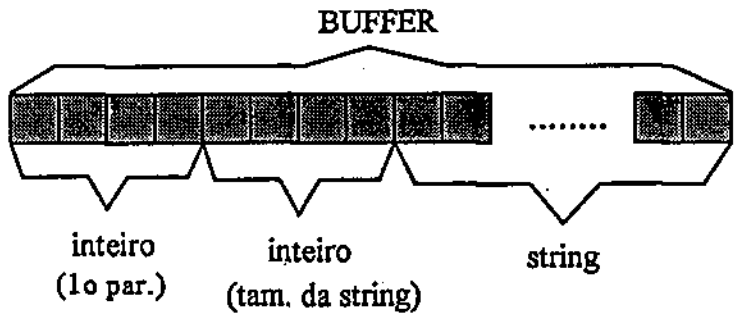
O protocolo propriamente dito se torna trivial depois de conhecido o que deve ser feito entre os stubs. A regra básica a ser seguida é sempre "desempacotar os dados exatamente na mesma ordem em que foram empacotados". O quê desempacotar e qual o

tamanho do dado são coisas que dependem do método que está em execução. Por exemplo, quando a aplicação cliente chama o método M do objeto O:

```
i = O.M( iA, szNome );
```

Os parâmetros passados foram um inteiro (iA) e uma string (szNome) que terá, nesse exemplo, um tamanho desconhecido. O retorno é um inteiro.

O *Stub Cliente* precisa empacotar esses dados para que fiquem da seguinte forma:



Depois de empacotados, os dados são transferidos para o *Stub Servidor*, que deverá desmembrar o BUFFER em um inteiro, outro inteiro, e uma string. Como é que o servidor sabe que tem que ser assim? A API diz isso! Se o método M recebe um inteiro e uma string, vai ser assim no cliente e no servidor. Como a string possui tamanho variável, há a necessidade de se adicionar mais um elemento do tipo inteiro no BUFFER e os dois (cliente e servidor) sabem disso. Por isso é que foi citado no início desta seção que não é

complicado, mas requer cuidados especiais, pois um errinho só e os dados podem ser desempacotados de forma errada, o que produz um resultado errado.

O retorno do método M também é empacotado (dessa vez pelo servidor) e transferido para o outro stub. Como esse retorno é apenas um inteiro, o BUFFER será formado apenas por ele. O *Stub Cliente* sabe o que vai receber e desmembra o BUFFER em apenas um inteiro.

3.7 O Protocolo Client/Server

Esta seção apenas cita algo sobre o que seria a definição do protocolo Client-Server no SIT, pois, como dito anteriormente, isso não é alvo desta dissertação.

Este protocolo não tem segredos. Desde que a implementação do protocolo MMF seja bem feita, é possível aproveitar todo o trabalho para o protocolo Client-Server.

Aqui também vale a idéia de se ter um *Stub Cliente* e um *Stub Servidor*. O cliente vai possuir exatamente a mesma API que o cliente MMF, e o servidor vai ser uma camada de interpretação dos dados do cliente e vai chamar o próprio SIT. O que muda é o esquema de transferência de informações, que não serão mais feitas via MMF e sim via RPC.

O trabalho de empacotamento continua valendo, mas o RPC já oferece ferramentas para isso, de forma que é possível dispensar o esquema de empacotamento usado para

MMF. É interessante usar o esquema RPC de empacotamento, pois ele cuida dos problemas de formato de dados entre máquinas de uma rede.

Quando este protocolo for implementado, os *Stubs RPC* substituirão os *Stubs MMF*, pois o RPC funciona inclusive para cliente e servidor na mesma máquina. Isso dispensa o uso de MMF.

É importante observar que a aplicação final não precisa estar por dentro de tudo isso. Basta acessar os métodos definidos pela API do SIT, sem se preocupar se está acessando um *Stub RPC*, ou *MMF*.

3.8 Reprocessamento de bases

Uma base de dados pode sofrer danos causados por uma queda de energia elétrica, ou algo semelhante. Também é possível alterar a estrutura física de uma base, inserindo ou retirando um ou mais campos. Alterar o tipo de um campo (de numérico para alfa, por exemplo) também é uma operação possível. Só que isso tudo pode requerer um reprocessamento da base.

Reprocessar uma base implica em refazer toda a sua estrutura e reconstruir seus dados, no que for possível. No caso de uma queda de energia, por exemplo, os dados podem ficar inconsistentes, implicando na necessidade de se reconstruir todos os dados e os índices.

No caso de inserção ou remoção de campos da estrutura física da base, não é necessário fazer a reorganização imediatamente, mas é recomendável, para que os registros e os índices fiquem "limpos".

O SIT oferece um meio seguro de realizar tal operação. Um método da classe **C_Session** é responsável pela reorganização de uma base. Esse método cria uma nova base e transfere todos os dados válidos da base velha para a nova, fazendo automaticamente a indexação desses dados. No final, a nova base assume o lugar da velha, que é removida do disco.

O sistema é robusto o suficiente para suportar uma queda durante o processo de reorganização de uma base e não perder a base. Quando for reincializado, o reprocessamento da base pode ser iniciado novamente. Isso é possível graças a um esquema de gravação temporária de contexto, que o SIT possui. Durante o reprocessamento de uma base, o sistema grava periodicamente (em disco) informações de controle que possibilitam a recuperação segura após uma falha.

3.9 Considerações sobre desempenho do SIT

Esta seção apresenta o resultado de alguns testes realizados sobre o desempenho da ferramenta SIT. Os testes foram elaborados considerando-se o tempo gasto (em segundos) para a realização de uma operação de indexação ou consulta aos dados e índices de uma base. Foram comparados os tempos gastos no SIT e no *LightBase for DOS*, um produto da *Light-Infocon Tecnologia S/A* que trabalha com recuperação textual (possui recursos de armazenamento, indexação, e consulta a dados, de forma semelhante ao SIT).

Para fazer os testes de performance do SIT, foi desenvolvida uma aplicação simples (para Windows NT) que usa a API para realizar os trabalhos de gravação, indexação e consulta aos dados. Essa aplicação, chamada DESEMP, possui rotinas para medição do tempo gasto nas operações realizadas, o que garante a precisão dos tempos apresentados.

No caso do *LightBase*, o tempo gasto nas operações foi medido externamente, através de cronômetro. Por isso, os tempos apresentados não são exatos e as operações muito rápidas não foram medidas. Essas operações não medidas têm seus tempos indicados com "?" nas tabelas abaixo.

Os testes foram realizados em uma máquina Pentium 75 MHz, com 32M de memória rodando Windows NT 3.5. Tanto para o DESEMP quanto para o LightBase foram utilizados arquivos texto de 10Kbytes, 100Kbytes, 500Kbytes e 1000Kbytes, que foram gerados a partir de arquivos diversos.

As tabelas IV e V apresentam os resultados dos testes realizados.

DESEMP						
Operação x Tam. Arq.	Gravar	Gravar e Indexar	Pesquisar			
			texto	a	'a	&
10k	0.05	0.38	0.02	0.07	0.04	0.75
100k	0.07	5.39	0.08	0.63	0.27	8.72
500k	0.35	24.9	0.37	3.18	1.07	42.8
1000k	0.8	41.38	0.8	13.7	2.43	135.23

Tabela IV - Resultado dos testes realizados com o DESEMP

LightBase						
Operações x Tam. Arq.	Gravar	Gravar e Indexar	Pesquisar			
			texto	a	'a	&
10k	0.19	0.67	?	?	?	0.26
100k	5.08	10.97	?	?	?	0.38
500k	5.27	11.4	?	?	?	0.44
1000k	5.74	19.74	?	?	?	0.42

Tabela V - Resultados dos testes realizados com o LightBase

Observa-se que o DESEMP apresentou melhor desempenho que o LightBase no tocante a gravação dos dados, sem indexação. Já o sistema de indexação do SIT deixou a desejar para arquivos maiores que 100Kbytes e o sistema de consultas obteve performance inferior à do LightBase.

Nota-se também que o rendimento do processo de indexação do SIT cai à medida em que o volume de informações aumenta. O LightBase também apresentou uma queda de rendimento, mas bem menor do que no SIT.

Dessa forma, fica evidente que o subsistema de indexação e consultas do SIT precisa de melhoramentos no tocante a performance. Trabalhos futuros poderão ser feitos nesse sentido.

Capítulo 4 - Detalhes de Implementação

Este capítulo descreve os detalhes mais importantes da implementação do SIT.

4.1 Modularização

A arquitetura analisada no capítulo anterior destaca as camadas *MMF* sobre o SIT, e apresenta outra camada localizada abaixo do SIT, que é o sistema de indexação e consultas. Na figura 4 (seção 3.1) também é apresentado o sistema de segurança do SIT. Esse sistema não é uma camada e sim um dos módulos internos do SIT.

A figura 6 apresenta a modularização do SIT em camadas.

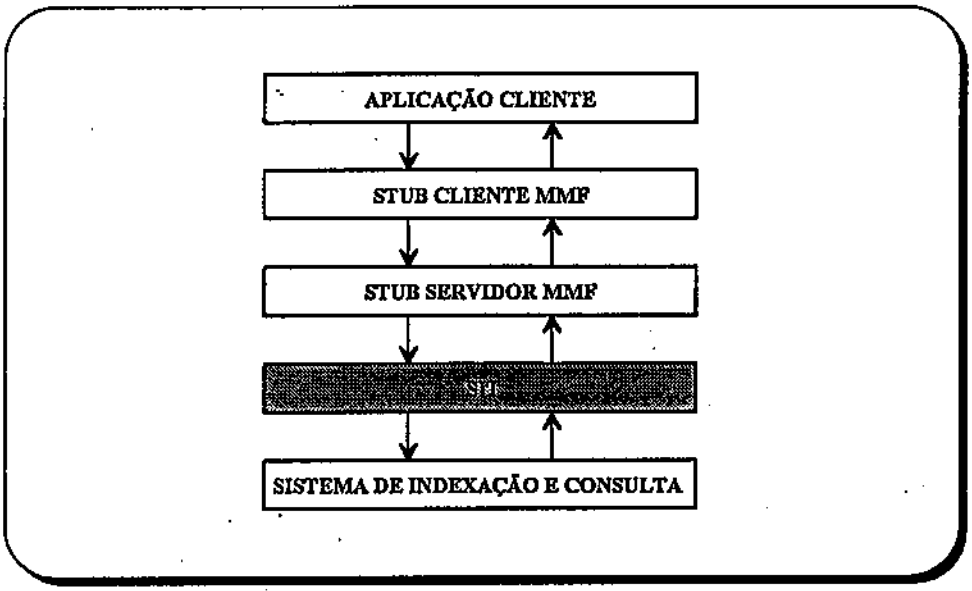


Figura 6 - Modularização do SIT

4.1.1 Os Stubs

Esses módulos possuem implementação simples. O único cuidado especial é com o empacotamento e desempacotamento das informações que trafegam entre os dois stubs. O stub Cliente oferece à aplicação exatamente a mesma API que o SIT, mas sem implementar o mesmo código. O trabalho desse stub é recolher os dados que são passados como parâmetros, empacotá-los e enviá-los para o stub Servidor.

O stub Servidor, por sua vez, desempacota os dados, monta novamente os parâmetros que a aplicação havia passado e chama a API do SIT. O valor de retorno é novamente empacotado e devolvido para o stub Cliente, que o repassa para a aplicação. A figura 7 mostra o esquema de empacotamento entre esses dois módulos.

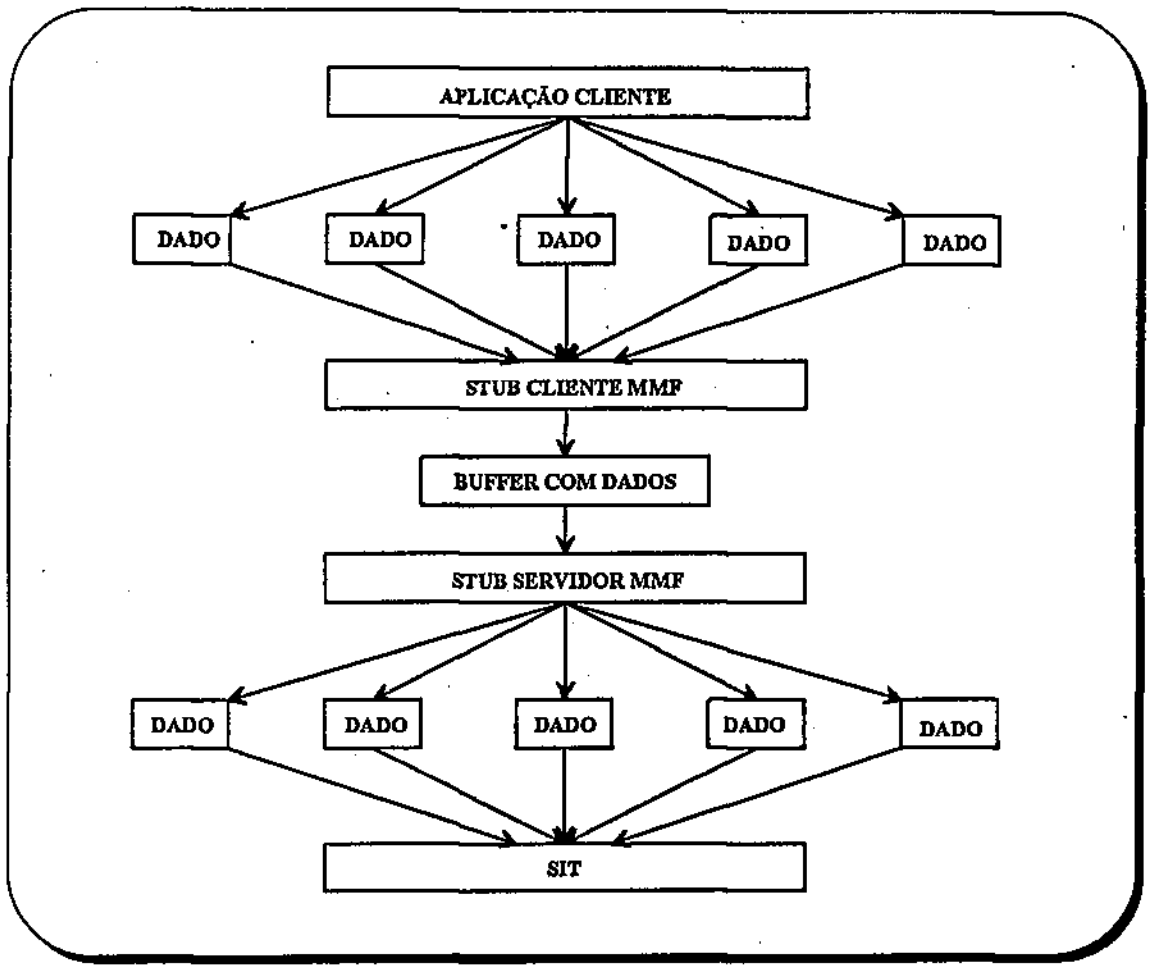


Figura 7 - Esquema de empacotamento de dados entre Stubs

4.1.2 Modularização Interna

O SIT está dividido internamente em classes que representam módulos funcionais. Para simplificar o entendimento, essas classes estão divididas em forma de árvore, onde cada camada representa um conjunto funcional. Na primeira camada estão as classes **C_Session** e **C_Ticket**, responsáveis pelo aspecto segurança na ferramenta e pela criação e manutenção de bases de usuários e de dados; a segunda camada contém as classes C_Base e C_Parser; nas próximas 3

camadas estão as classes C_Record, C_Field e C_Data e na última camada estão as classes de manipulação de arquivos, em paralelo com o sistema de índices. A figura 8 apresenta esta divisão:

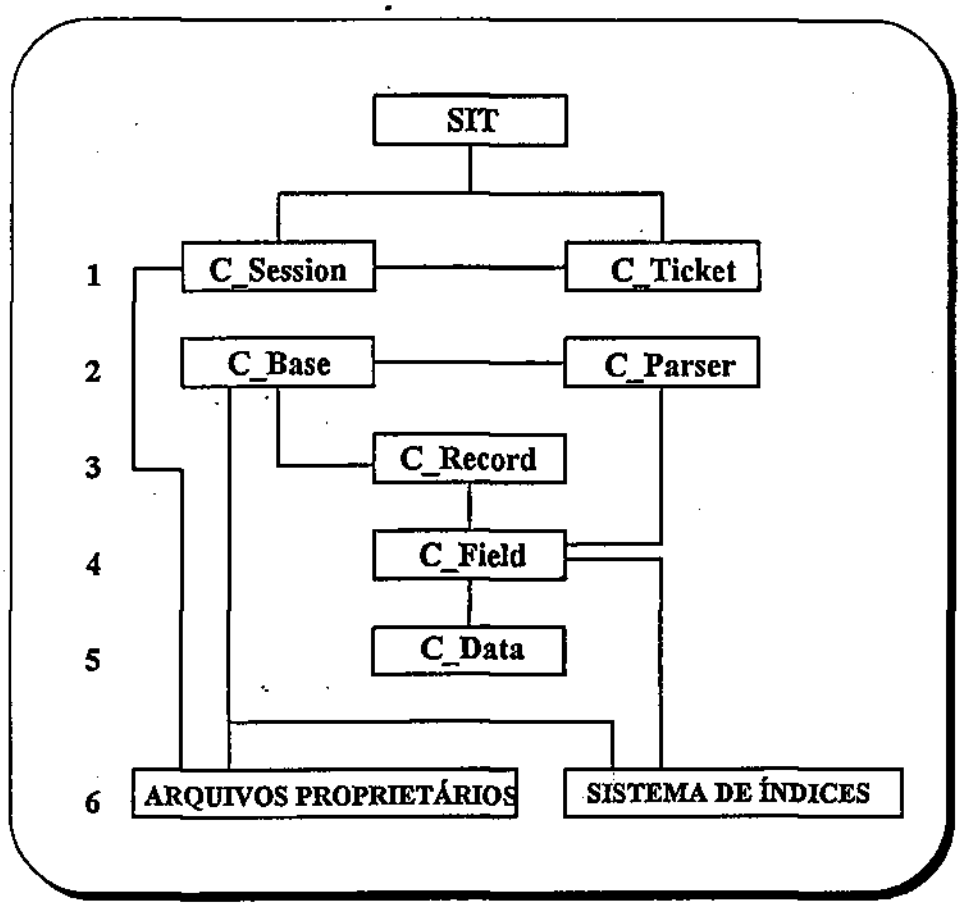


Figura 8 - Divisão funcional do SIT

Essa figura mostra que todo acesso a arquivos do SIT é feito através da camada 6. Essa camada é implementada pelas classes **C_File**, **C_DynFile**, **C_RecFile**, **C_SysFile1**, **C_SysFile2**, **C_SysFile3** e **C_SysFile4**, conforme a figura 9, a seguir.

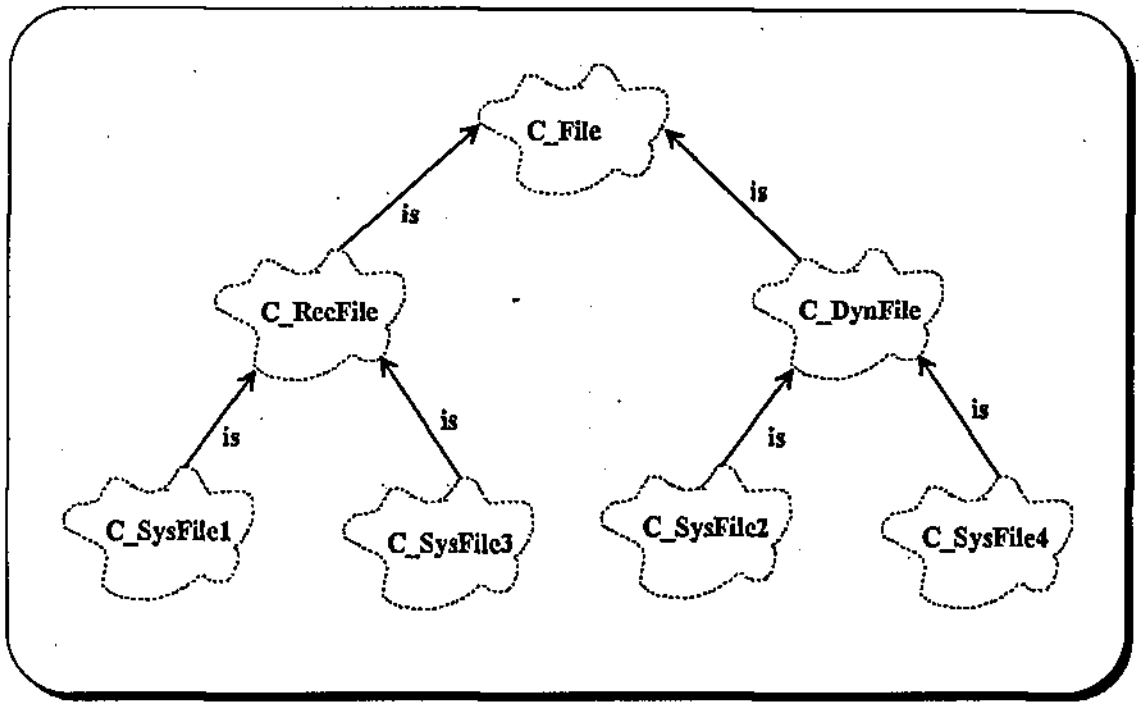


Figura 9 - Hierarquia das classes de tratamento de arquivos

A figura acima foi elaborada com base na notação de Booch [HORSTMANN95].

A classe **C_File** implementa mecanismos de criação, abertura, posicionamento, travamento e criptografia de arquivos e gravação e leitura de dados. Essa classe não implementa o conceito de registro e não pode ser usada diretamente pelo SIT para alguns casos.

Usando a classe **C_File**, a aplicação fica livre de problemas relacionados com o sistema operacional, como por exemplo o mecanismo de travamento de arquivos e o número máximo de arquivos que podem ser abertos simultaneamente

por um processo. O uso dessa classe torna a aplicação mais eficiente e portável (portabilidade é um dos requisitos exigidos no capítulo 2).

A classe **C_RecFile** é uma subclasse de **C_File** e implementa o conceito de registro, o que é útil para o SIT. Os mecanismos de posicionamento, gravação e leitura são feitos com base em registros de tamanho fixo.

A classe **C_DynFile**, também derivada de **C_File**, implementa o conceito de registros de tamanhos variáveis. Através de uma lista encadeada de blocos livres e ocupados do arquivo, essa classe implementa os mecanismos de gravação, leitura e posicionamento.

As classes **C_SysFile1** e **C_SysFile3**, que derivam da classe **C_RecFile**, representam alguns dos arquivos usados pelo SIT para manipular bases de dados. Os arquivos que contêm registros de tamanho fixo são manipulados pelo SIT através dessas classes. **C_SysFile1** é usada para manipulação do arquivo que contém informações sobre os registros de uma base. Esse arquivo contém um *header* com diversas informações sobre a base mais todos os registros da base, sem dados (apenas informações de controle e estatísticas). O arquivo representado pela classe **C_SysFile3** contém informações sobre os campos de uma base (quantos são, quais os nomes, índices, etc.).

As outras duas classes (**C_SysFile2** e **C_SysFile4**) representam outros arquivos do SIT. Esses arquivos possuem registros de tamanho variável e armazenam as informações do usuário (dados de todas as repetições, de todos os campos, de todos os registros).

4.2 Processo de Login

A classe **C_Session** é responsável, entre outras coisas, pelo processo de *Login*. Esse processo é a porta de entrada do usuário ao SIT.

Para realizar esse processo, o usuário deve primeiramente instanciar um objeto da classe **C_Session**. Nesse momento é especificado ao SIT o nome da máquina servidora onde deverá ser feita a instância da sessão (para a versão definida neste trabalho, o nome do servidor não é utilizado; o objeto é sempre instanciado na mesma máquina onde está o processo cliente). Com o objeto já criado, o usuário deve preencher uma estrutura que contém informações essenciais ao processo de login e depois invocar o método que realiza a operação. Essa estrutura é conhecida por *TNetInfo*, e contém as seguintes informações:

- Nome do usuário
- Senha do usuário
- Nome da UDB na qual o usuário está cadastrado
- Nome do Servidor
- Nome do Cliente

A partir dessas informações, o SIT vai fazer a validação do usuário e gerar um ticket para que ele prossiga com outras operações.

4.2.1 Como é realizado o processo de login

O método *Login*, da classe **C_Session** é responsável pela validação do usuário e geração do ticket. Ao receber as informações da estrutura *TNetInfo*, o método realiza as seguintes tarefas:

- abre a UDB indicada
- verifica a existência do usuário na UDB
- verifica a validade da senha do usuário
- monta um conjunto de informações de controle do login
- gera um ticket com essas informações (criptografadas)
- retorna o ticket ao usuário

Se algum desses processos falhar, o usuário recebe um ticket nulo e não conseguirá realizar outras operações sobre o SIT.

Essa operação não requer grande velocidade de processamento, mas nem por isso o SIT deixa de utilizar recursos otimizados para ganhar em performance. A consulta à UDB, por exemplo, é feita através de um esquema eficiente de busca que o SIT oferece também

ao usuário. Esse esquema trabalha sobre o sistema de índices e é capaz de localizar informações com grande precisão e rapidez. O processo de geração do ticket também é eficiente, tanto em velocidade quanto na criptografia das informações.

4.3 Operadores

Uma das maiores virtudes de uma linguagem orientada a objetos é, sem dúvida, a possibilidade de se redefinir os operadores e funções [STROUSTRUP93]. É possível, por exemplo, redefinir o operador + para somar (concatenar) duas strings sem que a funcionalidade original desse operador seja perdida.

O SIT faz bom uso dessa característica e disponibiliza alguns operadores para que uma aplicação possa utilizar os campos e dados de maneira mais simples e intuitiva.

No SIT, é possível acessar os campos de uma base como se eles estivesses dispostos em um vetor, através do operador []. A sintaxe abaixo demonstra esse recurso:

base [*número do campo*]

onde *base* é o nome do objeto que representa a base de dados e *número do campo* é o índice no array de campos da base. O sistema ainda vai mais longe. Disponibiliza outras sobrecargas (redefinições) de operadores para que o programador possa acessar os campos

da base e as repetições de cada campo de maneira mais simples e intuitiva. Abaixo estão apresentados os operadores disponíveis e dicas de como usá-los.

- operador ["*nome do campo*"]
  - esse operador faz parte da classe **C_Base** e acessa um campo através do seu nome (veja atributos de um campo na seção 2.3.4).
- operador [*índice do campo*]
  - esse operador também é da classe **C_Base** e acessa um campo pelo seu índice interno. Esse índice interno não representa o identificador do campo.
- operador (*identificador do campo*)
  - esse operador é o último da classe **C_Base** para acessar campos. A aplicação deve usá-lo para acessar um campo através de seu identificador, que é criado juntamente com a base. É bom observar que esse operador é formado por *parênteses*, enquanto que os dois anteriores são *colchetes*.
- operador [*índice do dado*]
  - esse operador faz parte da classe **C_Field** e é usado para acessar um dado (um dos possíveis valores de um campo, que pode ser multivalorado). O dado é acessado pelo seu índice de criação, que começa em zero.
- operador = (para dado)
  - com esse operador, que é da classe **C_Data**, a aplicação poderá atribuir valores aos campos. É só usar a seguinte sintaxe: *base* ["nome do campo"][índice do dado] = *valor*, onde *base* é o nome do objeto base e *valor* é um dado que deve ser do mesmo tipo do campo (string, número, data, etc.). Nesse exemplo o operador usado para

base foi o *["nome do campo"]*, mas poderia ter sido qualquer outro que acessasse um campo.

- dessa forma, é possível acessar um campo pelo operador de base e, a partir do campo, acessar um dado. Esse dado recebe um novo valor através do operador =.

- operador = (para campo)
  - a classe **C_Field** também possui um operador =. A aplicação só deve utilizar este operador para atribuir um valor à primeira repetição de um campo. Ele equivale a utilizar o operador anterior com o índice do dado igual a zero. Isto é:

base["nome do campo"][*0*] = *valor*

é a mesma coisa de

base["nome do campo"] = *valor*

- operador <<
  - este operador (também da classe **C_Field**) é usado para adicionar uma repetição a um campo. A diferença entre este operador e o = é que, neste, não é necessário informar nenhum índice. A sintaxe é a seguinte:

base["nome do campo"] << valor.

  - dessa forma, uma nova repetição será adicionada ao campo indicado.

- operador **cast**
  - quem programa em C com certeza já utilizou um **cast** para converter um ponteiro para *void* em um ponteiro para *char* ou algo semelhante. Com esse operador a aplicação pode obter os dados de um campo da seguinte forma:

valor = (long) base["nome do campo"][índice do dado]

ou

strcpy(valor, (char *) base["nome do campo"][índice do dado]).

  - é recomendável utilizar este operador sempre que a aplicação desejar obter uma repetição, pois elas são armazenadas internamente como *void**.

O programador pode optar por não usar os operadores para acessar os campos e repetições de uma base. A API do sistema é bastante vasta e oferece métodos que substituem os operadores descritos acima.

4.4 Comunicação entre processos

Esta seção descreve detalhes da implementação dos stubs MMF utilizados para a comunicação entre aplicações clientes e o SIT.

4.4.1 Um pouco sobre MMF

Memory Mapped File, ou simplesmente *MMF*, é um mecanismo de comunicação entre processos disponibilizado pelas versões 32 bits do Windows (Windows 3.1/3.11 com Win32s, Windows-NT e Windows-95). Esse mecanismo dá ao programador a possibilidade de compartilhar memória entre aplicações através de arquivos, que são criados e manipulados apenas em memória, sem nunca ir em disco. É como se a aplicação utilizasse arquivos comuns, em disco, mas de forma muito mais rápida.

A API do Windows disponibiliza funções para tratamento de arquivos em um nível mais alto do que as funções da biblioteca *stdio*. Ao invés de *fopen*, *fread*, etc. o programador lida com funções do tipo *CreateFile*, *OpenFile*, *WriteFile*, etc. A diferença não está somente nos nomes das funções; a API do Windows oferece mais recursos e mais facilidades ao programador do que a velha e conhecida *stdio*.

As funções MMF são muito semelhantes (quando não iguais) às de manipulação de arquivos do Windows. A função que mais se destaca na API MMF é a CreateFileMapping, que cria um arquivo em memória. Para usar este arquivo, basta chamar as funções normais de manipulação de arquivos do Windows (OpenFile, WriteFile, ReadFile, etc.).

Um outro conceito disponibilizado pela API MMF é o de *visão*. O programador pode enxergar apenas uma parte do arquivo, que é mapeada pela *visão*. É como se a aplicação acendesse uma lanterna sobre os bytes do arquivo e enxergasse os bytes iluminados, como mostra a figura 10, abaixo.

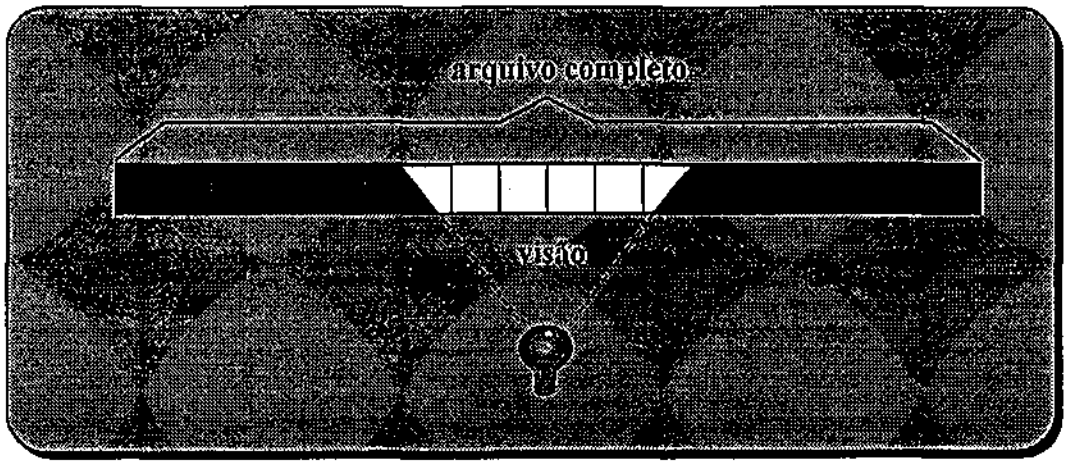


Figura 10 - Visão de um Memory Mapped File

Nada impede, contudo, que o programador defina uma visão do tamanho total do arquivo.

4.4.2 Como o SIT utiliza o potencial MMF

Utilizando o recurso MMF, o SIT implementa o esquema de compartilhamento de informações entre ele e as aplicações clientes. Esse compartilhamento de informações é necessário para que várias aplicações, ou vários módulos de uma aplicação possam enxergar as mesmas estruturas do SIT.

O Windows possibilita a troca de mensagens entre aplicações através das funções PostMessage e SendMessage. O problema é que apenas dois parâmetros (um inteiro e um long) podem ser passados, o que impossibilita a troca de estruturas maiores entre aplicações. Por isso, o SIT utiliza o MMF.

O Stub Servidor MMF é uma aplicação que deve ficar rodando sempre na máquina onde o SIT estiver instalado. Ele vai receber pedidos do Stub Cliente e repassá-los para o SIT.

O Stub Cliente MMF, que possui a mesma API do SIT, recebe os parâmetros dos métodos e os empacota em uma estrutura chamada C_Buffer. O stub então cria um arquivo em memória e grava o C_Buffer.

Através de funções da API do próprio Windows, o Stub Cliente localiza o Stub Servidor e envia-lhe uma mensagem. Essa mensagem contém um identificador do método que deve ser executado e o nome do arquivo onde estão os parâmetros.

O Stub Servidor abre o arquivo, retira os dados, fecha o arquivo e então executa o método requerido no SIT. Um outro arquivo é criado pelo Servidor, que grava o retorno do método e devolve para o Cliente o nome desse arquivo.

O Cliente, ao receber o resultado do método, destrói o arquivo que fora criado para o envio dos parâmetros, abre o arquivo que foi criado pelo Servidor e retira os dados de retorno. Esse arquivo será destruído pelo próprio servidor, quando ele for atender o próximo pedido.

A função utilizada para o envio de mensagens do Cliente para o Servidor é a SendMessage, que só permite a passagem de dois valores para o outro lado (um inteiro e um long). Sendo assim, o primeiro parâmetro determina qual o método deve ser executado pelo Servidor e o outro determina o nome do arquivo, que na verdade é um número.

4.5 Tratando compartilhamento de bases entre várias aplicações

Um dos maiores problemas encontrados por um gerenciador de banco de dados é o compartilhamento de seus registros por vários clientes. Manter a integridade e a consistência das informações é de fundamental importância.

Quando a base compartilhada está armazenada em um sistema de arquivos remoto, onde não há um servidor rodando, o problema é ainda mais grave. Esse caso não é apenas um compartilhamento de bases entre clientes, mas sim entre servidores. É possível haver vários servidores de bases de dados rodando em diversas máquinas espalhadas pela rede. Se mais de um servidor tentar acessar a mesma base, em um disco comum, ao mesmo tempo, o problema se agrava.

Felizmente, esse não é o caso dessa versão do SIT. O sistema projetado neste trabalho gerencia apenas bases de dados locais, sem se preocupar com sistemas de arquivos distribuídos. Então, o problema resume-se em compartilhar bases entre clientes distintos, mas sobre um mesmo servidor.

Para solucioná-lo basta um tratamento adequado na classe **C_Base**, implementando os seguintes cuidados:

- sempre que uma aplicação fizer alguma operação que implique em alteração de informações de controle (número de registros da base, localização do último registro, etc.), essas informações devem ser atualizadas em disco.
- quando uma aplicação solicitar o *lock* de um registro, verificar o "envelhecimento" do registro (uma aplicação pode estar enxergando um registro enquanto outra fez uma alteração, o que torna o registro "velho" para a primeira aplicação). Caso seja confirmado o envelhecimento, a aplicação deve ser instruída para atualizar suas informações locais (reler o registro) e tentar novamente o lock.
- nenhuma aplicação pode alterar o conteúdo de um registro sem antes fazer um lock.
- o método de adição de registro (append) deve sempre verificar informações de controle antes de gravar algo no disco.
- para alterar informações de controle, o SIT precisa fazer o *lock* de uma estrutura em disco, para evitar que outras aplicações realizem operações que também precisem dessas informações.

Tomando esses cuidados, o SIT torna possível o compartilhamento de bases por mais de um cliente.

4.6 Robustez das bases

Um outro aspecto importante em um banco de dados é a sua robustez (ver requisito na seção 2.1.7). O usuário não pode perder uma base de milhares de registros porque houve uma queda de energia elétrica durante a gravação de um registro, uma reindexação ou algo parecido. O SIT implementa um esquema eficaz para garantir robustez.

Primeiro, o sistema mantém sempre as informações de disco consistentes, para evitar problemas em caso de queda da máquina. O único risco de inconsistência ocorre durante a gravação de dados no disco e, mesmo assim, mecanismos de segurança deixam para atualizar informações de controle somente depois que tudo estiver seguro. Assim o usuário pode até perder um registro que estava gravando quando houve a queda de energia, mas a base continua consistente.

Segundo, o sistema possui um eficiente sistema de recuperação de bases, para aqueles casos em que o esquema descrito acima falhou. Essa recuperação é feita no momento em que a aplicação solicita a abertura da base, depois de uma falha no sistema.

Apenas quando os arquivos forem danificados de tal forma que o SIT não consiga abri-los é que não é possível recuperar as informações. Para esses casos, recomenda-se utilizar o velho e bom *backup*.

4.7 Reprocessamento de bases

O SIT disponibiliza, na classe **C_Session**, um método para reprocessamento de bases. Esse método deve ser utilizado quando a base sofrer alterações em sua estrutura física (adição/remoção de campos, mudança em atributos de campos, etc.) ou quando houver um número muito grande de registros apagados. Os registros apagados pelo SIT continuam ocupando espaço em um dos arquivos da base por questão de eficiência dos algoritmos internos. Então, é recomendado que se faça um "enxugamento" dos arquivos através de um reprocessamento.

A implementação do método de reprocessamento é simples, mas requer alguns cuidados para não gerar inconsistências. O algoritmo básico está descrito abaixo:

- criar uma nova base, com nome temporário e com estrutura física igual à base que vai ser reprocessada;
- copiar todos os registros da base original para a base temporária e indexá-los adequadamente;
- renomear a base original para um nome auxiliar
- renomear a base temporária para o nome original
- atualizar alguns itens de segurança que amarram a base à uma base de usuários
- depois de tudo confirmado e seguro, remover a base auxiliar (ex-original)

Essa operação pode ser lenta, principalmente no segundo passo do algoritmo descrito acima. Se houver uma queda do sistema durante o processo, pode acontecer uma das seguintes situações:

- *a base original ainda não foi renomeada para um nome auxiliar*: então, a base não foi perdida. A aplicação pode retomar as atividades quando desejar;
- *a base original já possui um novo nome*: se isso acontecer é porque existe uma base com um nome temporário que já está quase pronta para o uso, faltando apenas alguns ajustes. O SIT verifica a integridade da base temporária e, caso tudo esteja ok continua com o procedimento que estava fazendo quando houve a queda. Se a base temporária está inconsistente, é melhor renomear a base auxiliar para seu nome original e reiniciar o reprocessamento para garantir a integridade da operação.

Uma virtude do SIT é que, durante operações lentas como o reprocessamento ou a reindexação total de uma base, o respectivo método retorna um status da operação periodicamente para a aplicação. Nesse momento, informações de controle são gravadas para garantir que até aquele ponto tudo está íntegro. A aplicação decide se quer retomar as atividades ou parar um pouco para continuar depois.

4.8 Indexação, Consulta e Listas de Ocorrências

O foco principal do SIT é a recuperação de informações textuais e, para tanto, ele possui um esquema eficiente de armazenamento dessas informações, que é chamado de sistema de índices.

Há duas maneiras possíveis de se indexar dados usando o SIT. A primeira, é quando um registro de dados é atualizado pelo sistema, atendendo a uma solicitação da aplicação. Nesse momento as informações também são gravadas no subsistema de indexação, que é disponibilizado por uma biblioteca que não foi desenvolvida neste trabalho.

Outra maneira de se armazenar dados no sistema de índices de uma base é através do método (da classe **C_Base**) que realiza uma reindexação total dos dados. Este método percorre todos os registros de dados da base e indexa os dados de cada um deles.

4.8.1 O processo de indexação

Indexar os dados de um registro significa armazenar no sistema de índices todas as palavras encontradas em todas as repetições de todos os campos. Isso requer um trabalho especial de análise dos conteúdos dos campos, que é feito por um parser.

O parser usado pelo SIT é implementado na classe **C_Parser**. Essa classe disponibiliza métodos que interagem com o sistema de índices durante o processo.

O algoritmo básico do processo de indexação de um registro de uma base é o seguinte:

- para cada campo do registro:
    - para cada repetição do campo :
        - chama um método do sistema de índices
        - esse método do sistema de índices invoca um método do parser do SIT
            - o parser analisa o tipo do campo
            - se o campo for do tipo numérico, data ou hora, retorna o conteúdo
            - se for outro tipo, obtém um *token* da repetição e o retorna (esse token pode ser uma data, hora, valor, uma palavra, uma indicação de fim de frase, fim de parágrafo, ou de fim de processamento da repetição)
            - processa a mesma repetição até obter uma indicação de fim de processamento
    - fim do tratamento da repetição
- fim do tratamento do campo

Esse processamento é feito para cada tipo de índice que um campo possuir (ver tipos de índices na seção 2.3.4). Portanto, quanto mais índices houver em cada campo, maior será o tempo de processamento.

O processo descrito acima é aplicado todas as vezes que um registro de dados é atualizado ou criado pelo SIT, a não ser que a aplicação tenha desligado a *indexação on-line*. Esse tipo de indexação é o *default* do sistema e indica que a indexação dos dados deve ser feita de forma síncrona com a gravação do registro. Ao desativar esse tipo de indexação (através de um método da classe **C_Base**) a aplicação indica ao SIT que os registros devem ser criados/alterados sem que seus dados sejam indexados. Isso dá ao sistema uma velocidade maior e é bastante útil nos casos de clientes que trabalham com grande volume de entrada de dados.

Para indexar todos os dados que foram gravados enquanto a indexação estava *off-line* é necessário chamar um método da classe **C_Base**. Esse método percorre um arquivo que contém indicações dos registros que precisam ser indexados e realiza o processo de indexação, destruindo o arquivo no final. Esse arquivo é gerado sempre que a indexação *on-line* é desativada.

4.8.2 O processo de consulta

Uma vez que as informações estejam devidamente indexadas, é possível realizar buscas a partir de palavras, datas, valores, horas ou combinações de todas as possibilidades.

O SIT permite buscas do tipo *"maria*"*, *"*gia"*, por fonemas, com uso de operadores lógicos, faixa de valores, datas/horas, etc. (ver maiores detalhes em [BUSSMANN95], onde se encontra a BNF da linguagem de consulta).

Sempre que um termo é encontrado no sistema de índices, uma lista contendo todas as ocorrências do termo é gerada pelo SIT. Essa lista, conhecida por **Lista de Ocorrências**, contém estruturas que indicam a localização exata do termo dentro da base de dados. Mais detalhes sobre a estrutura de uma Lista de Ocorrências e de seus elementos podem ser encontrados na seção 2.3.5.

O SIT ainda é capaz de oferecer, para cada Lista de Ocorrências, a expressão que a gerou. Isso é necessário porque, quando se faz uma pesquisa do tipo

"João" e "Maria"

são geradas uma lista contendo todas as ocorrências de "João", outra com todas as ocorrências de "Maria" e outra com "João" e "Maria". Uma única pesquisa pode gerar várias Listas de Ocorrências.

Maiores detalhes sobre o sistema de índices utilizado neste trabalho podem ser obtidos em [BUSSMANN95].

Capítulo 5 - Considerações Finais

O SIT é uma ferramenta que consegue aliar robustez com facilidade de uso e permite que o programador construa bases de dados textuais de maneira simples e eficiente. Sua implementação orientada a objetos facilita muito a manutenção do código e a adição de novos recursos, o que torna possível a criação de novos trabalhos no sentido de aperfeiçoá-lo.

5.1 Preenchimento dos requisitos

No capítulo 2 deste trabalho foram apresentados os requisitos exigidos para a elaboração de uma ferramenta de gerência de bases de dados textuais. Esta seção descreve como o SIT atende a cada um deles, de maneira total ou parcial, citando algumas observações nos pontos mais importantes.

O primeiro requisito analisado aqui é a abrangência do sistema. O tratamento de vários tipos diferentes de dados é feito pelo SIT através de uma biblioteca especializada em indexação e recuperação de textos, valores numéricos, datas e horas. Além disso, o próprio SIT oferece suporte a tipos de dados binários, como imagem, som, vídeo, etc. e uma linguagem de recuperação textual (também oferecida pela biblioteca de indexação).

Dentre os três ambientes citados na definição dos requisitos (monousuário, multiusuário e cliente/servidor), o SIT atende (neste trabalho) apenas ao primeiro. O

ambiente monousuário é o que requer menor esforço de programação, pois não requer controle de concorrência, nem uma API de rede para a comunicação entre as aplicações.

Para atender ao requisito de portabilidade, o SIT foi implementado seguindo o paradigma de orientação a objetos e sobre uma plataforma operacional que oferece esquemas eficientes de gerenciamento de memória, IPC e travamento de recursos. O sistema operacional adotado para a implementação do SIT foi o Windows-NT e a linguagem, C++. Dessa forma, portar o sistema para Unix, por exemplo, requer apenas ajustes em algumas classes (as que manipulam diretamente com recursos do sistema operacional) e um compilador C++, obviamente.

Todos os requisitos de segurança são atendidos pelo SIT:

- A flexibilidade está em todos os pontos do sistema e não implica em complexidade para o programador de aplicações;
- os usuário de bases de dados são cadastrados e gerenciados pelo SIT, para que o controle de autenticação seja feito com segurança;
- a alocação de autorizações para os usuários é bastante flexível, consistindo de esquemas de login, senhas de acesso e ACLs;
- não existe a figura de um super-usuário que detém o controle sobre todos os outros. Cada um pode ser o dono do seu próprio conjunto de bases e controlar um conjunto de usuários que desejam acessar suas informações;

- as bases de dados são protegidas por um esquema de identificação do local onde elas foram criadas, o que impede que sejam copiadas indevidamente para outros ambientes.

A robustez do sistema é reforçada pela própria implementação também robusta do subsistema de indexação e consultas. Isso já garante a integridade dos índices em situações de erro. Os dados têm sua integridade garantida pelo próprio SIT, que implementa esquemas de replicação de informações de controle que possibilitam a recuperação das bases em casos de falha.

No aspecto desempenho, o SIT ainda pode melhorar bastante, pois só implementa o esquema de granularidade adequada para o travamento/transferência de recursos. Os esquemas de *threads* e API assíncrona não estão presentes na versão do SIT implementada neste trabalho.

O SIT possibilita hoje o armazenamento de informações em diversos idiomas (desde que seja utilizado o padrão de caracteres ISO8859/1). A linguagem de recuperação textual permite que sejam definidos e usados conectores e operadores na linguagem desejada pelo usuário, além dos formatos de datas, horas e valores numéricos. Isso atende aos requisitos de internacionalização que foram impostos para esta versão do sistema.

5.2 Trabalhos Futuros

Apesar do grande número de linhas de código (cerca de 50.000), o SIT possui uma implementação fácil de ser expandida, pois é toda orientada a objetos. Para acrescentar novas funcionalidades basta alterar as devidas classes, sem que sejam necessárias mudanças em todo o código. Esta seção descreve algumas funcionalidades que poderiam ser adicionadas ao SIT para torná-lo ainda melhor.

- Servidor para versão multiusuário

  Uma realidade em muitos ambientes de informática hoje é o uso de sistemas de arquivos distribuídos. Para dar suporte a esse tipo de ambiente, o SIT deve implementar recursos de controle de concorrência e travamento de recursos compartilhados.

- Servidor para versão cliente/servidor

  A necessidade de descentralização de dados e tarefas tem levado os sistemas de software a implementações distribuídas e o SIT pode seguir a mesma idéia.

  Isso pode ser feito implementando-se camadas de software responsáveis pela comunicação entre processos de máquinas distintas. Essas camadas, ou stubs de rede, podem ser implementadas sobre mecanismos de socket ou RPC. O stub cliente deverá oferecer para as aplicações exatamente a mesma API que o SIT oferece hoje. O stub servidor é responsável por receber pedidos dos clientes, executá-los no SIT e devolver os resultados.

Pontos importantes como minimização de tráfego na rede, minimização de estado, recuperação de falhas, maximização de desempenho com uso de API assíncrona e controle de concorrência devem ser observados.

- Suporte total a Unicode

  Para atender ainda mais aos requisitos de internacionalização do sistema, poderia ser implementado o suporte a caracteres Unicode. Isso pode ser feito no SIT com pequenas modificações em algumas classes, mas isso não é tudo. É preciso também realizar algumas alterações no subsistema de indexação e consultas.

- Implementação de *threads*

  Em um banco de dados, desempenho é algo de importância fundamental. Para melhorar o SIT é possível implementar o esquema de *threads*, para paralelizar o atendimento a alguns pedidos. A forma mais simples de se fazer isso é disparar uma *thread* para cada pedido que a aplicação cliente fizer ao SIT. Dessa forma, o sistema estará sempre esperando por um pedido, mesmo que esteja processando outro. Outra forma de agilizar o processamento é, nos métodos mais pesados, disparar *threads* que paralelizem as tarefas.

  O principal cuidado a ser tomado é com o compartilhamento das estruturas de dados entre as *threads*. Esquemas como região crítica, semáforos e mutex podem ser usados para contornar possíveis problemas. É aconselhável implementar todo o

tratamento de threads e de regiões críticas, mutex, etc. em classes de objetos para facilitar futuros portes do sistema.

- Relacionamentos

  Um dos recursos mais interessantes de um banco de dados relacional é o conceito de relacionamento entre bases. Através dele é possível definir dependências entre campos de bases distintas, montar visões que contêm partes de várias bases, etc. Isso é um ótimo recurso para ser adicionado ao SIT.

  Alterando-se a classe **C_Field** e/ou **C_Base** do SIT (e talvez outras) é possível implementar esquemas de ligações entre bases de dados que simulem o conceito de relacionamento.

- Parsers para processadores de textos

  Para indexar os dados de uma base, o SIT realiza o processo de *parsing* (ver capítulo 3). Esse processo é realizado por um objeto especialmente desenvolvido para isso. O SIT possui uma classe **C_Parser** para cada tipo de texto que é suportado. Assim, um texto escrito em ASCII é processado por um objeto, o texto escrito no processador de textos AMI-PRO é processado por outro objeto, e assim sucessivamente. Para incluir suporte a um novo tipo de processador de texto, é necessário definir uma classe **C_Parser** e recompilar o SIT.

  A sugestão é que seja criado um mecanismo de adição de parsers que não implique na recompilação do sistema. Cada novo parser poderia ser uma DLL, por exemplo,

e o SIT estaria preparado para carregar todos os parsers no momento de sua inicialização.

Com a implementação desse recurso, o SIT será capaz de indexar/recuperar informações de qualquer processador de textos, desde que seja oferecido um parser adequado.

- Testes e refinamentos

  Apesar de todo o esforço envolvido na implementação deste trabalho, certamente serão encontrados pontos onde a performance não está agradável (ou pelo menos poderia ser melhor) ou a robustez não está suficiente. Um possível trabalho para o futuro é e elaboração de testes de performance e robustez e o aperfeiçoamento da implementação do sistema.

Apêndice A

API do Servidor de Informações Textuais

Este apêndice descreve as principais classes da API do sistema e seus métodos mais importantes. Observe que as descrições dos métodos não estão detalhadas. Para maiores informações sobre a API, consulte o respectivo manual de referência.

- **Classe C_Session**

  Esta classe é responsável pelos trabalhos de criação, abertura, fechamento e destruição de bases de dados, criação/destruição de UDBs, *login* de usuário no sistema e por responder a perguntas do tipo *quais são os usuários que têm acesso à base X*, ou *quais são os servidores que estão "no ar"*. Seus métodos mais importantes são:

  - *C_Session *New()*

    Este método instancia um objeto C_Session no servidor e retorna o ponteiro para a aplicação.

  - *void Delete(C_Session *)*

    Destrói um objeto C_Session no servidor.

  - *int AddGroupToUser(C_Ticket *, char *szUDBName, char *szUserName, char *szGroups)*

Adiciona os grupos contidos no parâmetro szGroups ao usuário szUserName, na UDB szUDBName. Retorna OK em caso de sucesso ou um código de erro.

- *int AddUser(C_Ticket *, char *szUDBName, TUDBRecord *tUsersInfo)*

  Adiciona novos usuários a uma UDB. O nome da UDB é passado em szUDBName e as informações sobre os usuários que serão criados devem estar em tUsersInfo. Retorna OK em caso de sucesso ou um código de erro.

- *int CloseBase(C_Ticket *, LBSC_Base *)*

  Fecha uma base de dados que tenha sido aberta pelo método OpenBase. Retorna OK em caso de sucesso ou um código de erro.

- *int CreateBase(C_Ticket *, char *szBaseName, char *szBasePasswd, char *szMaintPasswd, BYTE bBaseType, BOOL bIsEncrypt, char *szUDBName, TField *ptFieldInfo, int iSlotNum)*

  Cria uma base de dados a partir dos parâmetros especificados: szBaseName indica o nome da base que será criada; szBasePasswd é a password da base; szMaintPasswd é a password de manutenção; bBaseType, o tipo (pública, restrita, etc.); bIsEncrypt indica se a base deve ser criptografada (TRUE) ou não; szUDBName é o nome da UDB que servirá de apoio para a nova base; ptFieldInfo contém informações sobre todos os campos da base; iSlotNum indica o número de slots que deverão ser criados para a nova base. Retorna OK ou um código de erro.

- *int CreateUDB(C_Ticket *, char *szUDBName, char *szPasswd, char *szMaintPasswd)*

  Cria uma base de usuários, cujo nome está em szUDBName. Os parâmetros szPasswd e szMaintPasswd indicam, respectivamente, a senha de uso da UDB e a senha de manutenção. Retorna OK ou um código de erro.

- *int DeleteBase(LBSC_Ticket *, char *szBaseName)*

  Remove uma base de dados do disco. Retorna OK ou erro.

- *int DelGroupFromUser(C_Ticket *, char *, char *, char *)*

  Remove um conjunto de grupos de um usuário em uma UDB. Retorno OK ou erro.

- *int DelUser(C_Ticket *, char *, char *)*

  Remove um usuário de uma UDB. Retorna OK ou erro.

- *char * GetGroups(C_Ticket *, char *szUserName)*

  Obtém a lista de todos os grupos do usuário szUserName, na UDB que estiver em uso. Se szUserName for NULL, o método retornará todos os grupos de todos os usuários da UDB em uso. Retorna NULL em caso de erro.

- *char * GetUsers(C_Ticket *, char *szUDBName)*

  Obtém a lista de todos os usuários de uma UDB. Retorna NULL em caso de erro.

- *C_Ticket *Login(TNetInfo &)*

  Realiza a operação de login e retorna um objeto C_Ticket para a aplicação. Esse ticket deverá ser usado nas demais operações desta classe e da classe C_Base.

- *int Logout()*

  Desfaz o login do usuário na UDB. Retorna OK ou erro.

- *int OpenBase(C_Ticket *, char *szBaseName, char *szBasePasswd, BOOL bExclusive, BOOL bReadOnly, BOOL bMaintenance, C_Base **pOpenBase)*

  Abre uma base de dados. Os parâmetros são: szBaseName - nome da base; szBasePassword - senha para abertura da base; bExclusive - se a base deve ser aberta em modo exclusivo ou não; bReadOnly - se a base deve ser aberta somente para leitura; bMaintenance - se a base deve ser aberta para manutenção (nesse caso, szBasePassword deve conter a senha de manutenção da base). Este método instanciará um objeto C_Base internamente e retornará o seu apontador através do parâmetro pOpenBase. Retorna OK ou erro.

- *int ReorganizeBase(LBSC_Ticket *, LBSC_Base *)*

  Reprocessa uma base de dados. É útil quando a integridade da base está duvidosa ou quando a sua estrutura física é alterada (campo inserido ou removido). Retorna OK ou erro.

- *char * WhatServers()*

  Este método retorna uma lista de todos os servidores da rede. Na versão monousuário (esta), o retorno sempre será correspondente à máquina onde o sistema está rodando. Na versão cliente/servidor, o stub de rede se encarrega de fazer um *broadcast* (ou utilizar outra técnica) e montar a lista de todos os servidores da rede.

- *char * WhatServersForUser(char *szUserName)*

  Retorna a lista dos servidores cujo usuário szUserName possui algum tipo de acesso.

- **Classe C_Base**

  Esta classe é responsável pelo gerenciamento de uma base de dados, uma vez que ela tenha sido aberta. Seus métodos mais importantes são:

  - *int AddField(C_Ticket *, TField *tFieldInfo)*

    Adiciona um novo campo à estrutura da base. O parâmetro tFieldInfo deve conter todas as informações do novo campo. Retorna OK em caso de sucesso ou um código de erro.

  - int AddGoWord(UINT uiFieldId, char *)

    Adiciona uma GoWord ao campo definido por uiFieldId. Retorna OK ou erro.

  - int AddStopWord(C_Ticket *, char *)

    Adiciona uma StopWord à base. Retorna OK ou erro.

  - int AppendRecord(C_Ticket *)

    Adiciona um registro à base. O registro que está em memória é inserido no final da base. Retorna OK ou erro.

  - int DelACLPerm(char *szName, long lId, int iACL)

    Remove um conjunto de permissões de ACL do usuário ou grupo especificado em szName. O parâmetro lId identifica o campo ou registro de onde a permissão deve

ser removida. O campo iACL indica ao método qual ACL deve ser alterada (de campo ou registro, para usuário ou grupo). Retorna OK ou erro.

- int DeleteOcList(C_Ticket *, char *)

  Remove uma lista de ocorrências do disco. Retorna OK ou erro.

- int DeleteOcRecords(C_Ticket *)

  Remove da base todos os registros que constam na lista de ocorrências ativa no momento. Retorna OK ou erro.

- int DeleteRecord(C_Ticket *)

  Remove da base o registro corrente. Retorna OK ou erro.

- int DelField(UINT uiFieldId)

  Remove da base o campo identificado pelo parâmetro uiFieldId. Retorna OK ou erro.

- int DelFieldRepetition(UINT uiFieldId, int i)

  Remove do campo uiFieldId, a i-ésima repetição (valor). Retorna OK ou erro.

- int DelGoWord(UINT uiFieldId, char *)

  Remove do campo uiFieldId uma GoWord. Retorna OK ou erro.

- int DelStopWord(C_Ticket *, char *)

  Remove uma StopWord da base.

- int EnableOcList(C_Ticket *, int iHandle)

  Habilita a lista de ocorrências cujo *handle* é iHandle.

- int FirstRecord(C_Ticket *)

  Posiciona a base no primeiro registro. Retorna OK ou erro.

- int GetACLPerm(char *szName, long, int)

  Obtém o conjunto de permissões de ACL do usuário ou grupo identificado por szName. Retorna uma máscara de bits contendo as permissões ou um valor negativo em caso de errro.

- C_Occurrence *GetCurrentOccurrence(C_Ticket *)

  Obtém um objeto do tipo C_Occurrence representando a ocorrência atual da lista de ocorrências ativa. Retorna NULL em caso de erro.

- TIndexAttrib GetFieldIndexAttrib(UINT uiFieldId)

  Obtém o conjunto de índices do campo uiFieldId. Retorna uma máscara de bits contendo os índices do campo.

- const C_Field *GetFieldObj(UINT uiFieldId)

Obtém um objeto do tipo C_Field representando o campo cujo identificador é uiFieldId. Retorna NULL em caso de erro.

- int GetFieldRepetition(UINT uiFieldId, int i, char *)

  Obtém a i-ésima repetição do campo uiFieldId. Este método está sobrecarregado para todos os tipos de campo suportados pelo SIT. O último parâmetro indica o tipo do campo e serve para armazenamento do valor de retorno. Esse campo pode ser: *char* * (este caso), *double* *, *long* *, etc., para cada tipo de campo. O retorno do método é OK ou erro.

- int GetFieldRepetitionByVal(UINT uiFieldId, char *szContent)

  Obtém o índice de uma repetição a partir de seu valor. Este método também é sobrecarregado para todos os tipos de campo suportados pelo sistema. O retorno é o índice da repetição ou um valor negativo, indicando erro.

- long GetFieldRepetitionSize(UINT uiFieldId, int i)

  Obtém o tamanho da i-ésima repetição do campo uiFieldId. Retorna o tamanho ou um valor negativo, indicando erro.

- TField *GetFields(C_Ticket *)

  Obtém um vetor contendo informações detalhadas sobre todos os campos da base. Retorna NULL em caso de erro.

- int GetFieldSlot(UINT uiFieldId, void *)

  Obtém a informação do slot do campo uiFieldId. Retorna OK ou erro.

- long GetFieldSlotSize(UINT uiFieldId)

  Obtém o tamanho da informação contida no slot do campo uiFieldId.

- C_Occurrence *GetFirstOccurrence(C_Ticket *)

  Obtém um objeto do tipo C_Occurrence representando a primeira ocorrência da lista de ocorrências ativa.

- C_Occurrence *GetLastOccurrence(C_Ticket *)

  Obtém um objeto do tipo C_Occurrence representando a última ocorrência da lista de ocorrências ativa.

- C_Occurrence *GetNextOccurrence(C_Ticket *)

  Obtém um objeto do tipo C_Occurrence representando a próxima ocorrência da lista de ocorrências ativa.

- C_Occurrence *GetNthOccurrence(C_Ticket *, long l)

  Obtém um objeto do tipo C_Occurrence representando a l-ésima ocorrência da lista de ocorrências ativa.

- int GetNumberOfFields()

Obtém o número de campos da base. Retorna um valor negativo em caso de erro.

- int GetNumberOfRepetition(UINT uiFieldId)

  Obtém o número de repetições (valores) do campo uiFieldId. Retorna um valor negativo em caso de erro.

- int GetNumberOfSlots()

  Obtém o número de slots da base. Retorna um valor negativo em caso de erro.

- long GetNumOccurrences(C_Ticket *)

  Obtém o número de ocorrências existentes na lista de ocorrência ativa.

- long GetNumRecords(C_Ticket *)

  Obtém o número de registros da base.

- C_Occurrence *GetPreviousOccurrence(C_Ticket *)

  Obtém um objeto do tipo C_Occurrence representando a ocorrência anterior da lista de ocorrências ativa.

- int GetSlot(int i, void *)

  Obtém a informação armazenada no i-ésimo slot da base.

- long GetSlotSize(int i)

Obtém o tamanho da informação armazenada no i-ésimo slot da base.

- int IndexAll(C_Ticket *, BOOL = FALSE)

  Realiza uma reindexação total da base. o Segundo parâmetro não precisa ser informado, caso em que assume o valor FALSE. Ele serve para informar se o método deve restaurar algum contexto de indexação anterior (valor TRUE) ou se a indexação deve ser iniciada a partir do primeiro registro da base (valor default, FALSE). A cada *n* registros (*n* configurável) indexados, o método salva um contexto e retorna para a aplicação. Se a aplicação desejar continuar o processo, é só chamar o método novamente, passando TRUE no segundo parâmetro.

- int LastRecord(C_Ticket *)

  Posiciona a base em seu último registro. Retorna OK ou erro.

- int LoadOcList(C_Ticket *, char *szFileName)

  Carrega do disco uma lista de ocorrências e a torna ativa. Retorna OK ou erro.

- int Locate(C_Ticket *, UINT uiFieldId, int iIndex, char *szKey, int iType = EQUAL_KEY)

  Localiza a chave szKey na base, desde que ela esteja no campo uiFieldId e no índice iIndex. A busca é feita com base o parâmetro iType, que possui valor default igual a EQUAL_KEY. Isso significa que a chave procurada no índice deve ser exatamente igual à informada em szKey. Outros valores podem ser

especificados para que o método procure por chaves maiores, menores, etc. Em caso de sucesso, o método provoca um posicionamento sobre o primeiro registro da base que possua a chave procurada. Retorna OK ou erro.

- int LockRecord(C_Ticket *)

  Trava o registro corrente para que outros processos não o atualizem. Retorna OK ou erro.

- int ModifyFieldRepetition(UINT uiFieldId, int, char *)

  Modifica o conteúdo de uma repetição do campo uiFieldId. Este método também é sobrecarregado para todos os tipos de campo suportados pelo sistema. Retorna OK ou erro.

- int NextRecord(C_Ticket *)

  Posiciona a base no próximo registro. Retorna OK ou erro.

- int NthRecord(C_Ticket *, long l)

  Posiciona a base no l-ésimo registro. Retorna OK ou erro.

- C_Field& operator[](int i)

  Obtém o i-ésimo campo da base. Observe que o retorno é um objeto C_Field. Este operador, assim como os outros descritos abaixo, possibilita o acesso aos campos da base de maneira mais fácil e prática para o programador.

- C_Field& operator[](char *szFieldName)

  Obtém o campo cujo nome é especificado em szFieldName.

- C_Field& operator()(UINT uiFieldId)

  Obtém o campo cujo identificador é uiFieldId.

- C_Field& operator()(char *szFieldDescription)

  Obtém o campo cuja descrição está especificada em szFieldDescription.

- int PreviousRecord(C_Ticket *)

  Posiciona a base no registro anterior. Retorna OK ou erro.

- int PutFieldRepetition(UINT uiFieldId, char *)

  Insere uma repetição (valor) no campo uiFieldId. A repetição é inserida na última posição da lista de valores do campo. Este método também está sobrecarregado para todos os tipos de campo suportados pelo sistema. Retorna OK ou erro.

- int PutFieldRepetitionByIndex(UINT uiFieldId, char *, int i)

  Atualiza a i-ésima repetição do campo uiFieldId. Se *i* for maior que a quantidade de repetições do campo, novas repetições (entre o número atual e *i*) serão geradas com valores vazios. Este método também é sobrecarregado para todos os tipos de campo suportados pelo sistema. Retorna OK ou erro.

- int PutFieldSlot(UINT uiFieldId, void *pvContent, int iSize)

  Insere um valor no slot do campo uiFieldId. Retorna OK ou erro.

- int PutSlot(int i, void *pvContent, int iSize)

  Insere um valor no i-ésimo slot da base. Retorna OK ou erro.

- int ReadRecord(C_Ticket *)

  Recarrega para a memória o registro corrente, que está em disco. Retorna OK ou erro.

- int ReleaseRecord(C_Ticket *)

  Destrava o registro corrente (operação inversa ao LockRecord). Retorna OK ou erro.

- int SaveOcList(C_Ticket *, int iHandle, char *szFileName)

  Salva em disco a lista de ocorrências identificada por iHandle. Retorna OK ou erro.

- int *Search(C_Ticket *, char *szExpression)

  Realiza uma pesquisa sobre o sistema de índices da base. A expressão de consulta deve obedecer à BNF da linguagem de recuperação textual. Retorna um vetor de handles para as listas de ocorrências geradas ou NULL em caso de erro. É possível

que sejam geradas várias listas de ocorrências, porque o sistema de índices do SIT "quebra" a expressão de consulta em várias subexpressões e, para cada uma delas, é gerada uma lista. Ex.: expressão igual a "João *or* Maria". Serão geradas 3 listas: uma contendo todas as ocorrências de "João", outra com "Maria" e outra (chamada de resultante) contendo "João *or* Maria". A lista resultante é automaticamente habilitada e o registro corrente passa a ser o equivalente à primeira ocorrência da lista.

- int SetACLPerm(char *szName, long, BYTE, int)

  Informa ao sistema um conjunto de permissões para o usuário ou grupo especificado por szName. Retorna OK ou erro.

- int SetNavigationByIndex(C_Ticket *, BOOL bInit, UINT uiFieldId = 0,
  int iIndex = 0, int iType = EQUAL_KEY)

  Habilita um esquema de "filtragem" dos registros da base para que a navegação (através dos métodos NextRecord, PreviousRecord, etc.) obedeça a uma regra. O parâmetro *bInit* indica se esse esquema deve ser ligado (TRUE) ou desligado (FALSE, só faz sentido se a aplicação já está usando o esquema de filtragem). Quando o esquema é ativado, a navegação pelos registros da base obedecerá à ordem dos temos armazenados no índice iIndex, do campo uiFieldId. O parâmetro iType determina o tipo de comparação que é feita com as chaves internamente (maior que, menor que, igual, etc.). Retorna OK ou erro.

- int SetNavigationByKey(C_Ticket *, BOOL bInit, UINT uiFieldId = 0,

  int iIndex = 0, char *szNavKey = NULL, int iType = EQUAL_KEY)

  Habilita um esquema de "filtragem" dos registros da base para que a navegação (através dos métodos NextRecord, PreviousRecord, etc.) obedeça a uma regra. O parâmetro *bInit* indica se esse esquema deve ser ligado (TRUE) ou desligado (FALSE, só faz sentido se a aplicação já está usando o esquema de filtragem). Quando o esquema é ativado, a navegação pelos registros da base obedecerá à existência do termo szNavKey, no índice iIndex, do campo uiFieldId. O parâmetro iType determina o tipo de comparação que é feita com as chaves internamente (maior que, menor que, igual, etc.). Retorna OK ou erro.

- int UnloadOcList(C_Ticket *, int iHandle)

  Descarrega da memória a lista de ocorrências identificada por iHandle. Este método não remocve a lista do disco. Retorna OK ou erro.

- int UpdateIndex(C_Ticket *)

  Indexa todos os registros que foram gravados durante uma indexação "off-line". Retorna OK ou erro.

- int UpdateRecord(C_Ticket *)

  Grava o registro corrente em disco. O registro precisa estar travado (LockRecord) para que essa operação seja bem sucedida. Retorna OK ou erro.

- **Classe C_Field**

  Dessa classe, só é necessário destacar os seguintes operadores:

  - C_Data& operator [](int i)

    Este operador retorna a i-ésima repetição do campo.

  - C_Field& operator << (char *)

    Este operador adiciona uma repetição no final da lista de valores do campo. Existem sobrecargas dele para todos os tipos de dados suportados pelo sistema.

  - void operator = (char *)

    Este operador atualiza a repetição de número 0 (zero) do campo. É equivalente a utilizar o operador [0] para selecionar a repetição e em seguida fazer a sua atualização. Ele também está sobrecarregado para todos os tipos de dados suportados pelo sistema.

- **Classe C_Data**

  Cada objeto dessa classe representa uma repetição de um campo. Também neste caso, só é necessário destacar os seguintes operadores:

  - void operator = (char *)

    Este operador atualiza o conteúdo do objeto. Ele também está sobrecarregado para todos os tipos de dados suportados pelo sistema.

  - operator char* (void)

    Este operador converte o valor armazenado internamente para o tipo de dado solicidado. É o operador *cast*, conhecido dos programadores C. Também está sobrecarregado para todos os tipos de dado suportados.

Referências Bibliográficas

[ALSINA94] ALSINA, Marcos Sebastian - NetComm - Uma ferramenta para o desenvolvimento de aplicações multimídia distribuídas e gerência de redes em ambiente Windows
Dissertação de Mestrado - UFPB, 1994

[BELL94] BELL, Timothy C., WITTEN, Ian H,. MOFFAT, A.
Managing Gigabytes - Van Nostrand Reinhold, 1994.

[BLOOMER91] BLOOMER, J. - Power Programming With RPC, O'Reilly & Associates, Inc. - 1991

[BORLAN93] Borland International, Inc. Borland C++ 4.0: Programmer's Guide, Borland International, Inc. 1993.

[BOYCE92] BOYCE, J. - Inside Windows for Workgroups,
New Riders Publishing, 1992

[BUSMMAN95] BUSSMANN, J. Eduardo Carvalho, BART - Uma Biblioteca Orientada a Objetos de Apoio à Recuperação Textual
Dissertação de Mestrado - UFPB, 1995

[BRSCHMIDT93] BROCKSCHMIDT, K. - Programming for Windows with Object Linking and Embedding 2.0, Microsoft Press, 1993

[BUGG93] BUGG, K. E. & Tackett J. - Implementing and Using Windows Help, The C Users Journal, Sep. 1993, pp. 63-72.

[CLARK92] CLARK, J. D. - Windows Programmer's Guide to OLE/DDE
Prentice Hall, 1992.

[CUSTER92] CUSTER, H. Inside Windows NT, Microsoft Press, 1992.

[DAVIS93] DAVIS, R. - Windows Network Programming: How to Survive in a World of Windows, DOS, and Networks.
Addison-Wesley Publishing Company, 1993.

[DOUGLAS95] DOUGLAS, E. Comer - Internetworking with TCP/IP
Prentice Hall, Volume I, 1995.

[DUMAS94] DUMAS, Arthur - Programming Winsock
SAMS Publishing, 1994.

[EDDON93] EDDON, G. - RPC for NT - R&D Publications, 1994.

[HALL93] HALL, M. & Towfiq, G. & Treadwell, D. & Sanders, H.
Windows Sockets: An Open Interface for Programming under Microsoft Windows, Version 1.1 - Internet Document, 1993.

[HORSTMANN95] HORSTMANN, Cay S. - Mastering Object-Oriented Design in C++
John Wiley & Sons, Inc., 1995.

[KING94] KING, A. - Memory Management, the Win32 subsystem, and Internal Synchronization in Chicago, Microsoft System Journal, May 1994, pp. 57-61.

[MSOFT92a] Microsoft Corporation, The Windows Interface: An Application Design Guide, Microsoft Press, 1992.

[MSOFT92b] Microsoft Corporation. Microsoft Windows 3.1 Programmer's Reference, Vol. 1, Microsoft Press, 1992.

[MSOFT92c] Microsoft Corporation. Microsoft Windows 3.1 Programmer's Reference, Vol. 2, Microsoft Press, 1992.

[MSOFT92d] Microsoft Corporation. Microsoft Windows 3.1 Programmer's Reference, Vol. 3, Microsoft Press, 1992.

[MSOFT92e] Microsoft Corporation. Microsoft Windows 3.1 Programmer's Reference, Vol. 4, Microsoft Press, 1992.

[MSOFT92f] Microsoft Corporation. Microsoft Windows 3.1: Guide to Programming, Microsoft Press, 1992.

[PETZOLD90] PETZOLD, C. - Programming Windows: The Microsoft Guide to Writing Applications for Windows 3, Microsoft Press, 1990.

[PIETREK93] PIETREK, M. - Windows Internals: The Implementation of the Windows Operating Environment, Addison-Wesley, 1993.

[RAIMA84] Raima Database Manager - User Guide - Raima Corporation, 1984 - 1994

[RICHTER92] RICHTER, J. M. - Windows 3.1: A Developer's Guide, 2nd Edition, M&T Books, 1992.

[SAMPAIO84] SAMPAIO, Marcus Costa - Um Sistema de Arquivos Robusto para o Sistema Operacional Unix - Dissertação de Mestrado, UFPB, 1984.

[SANTOS93] SANTOS, G. J. - Considerações para o Desenvolvimento de Aplicações Distribuídas em Ambientes Heterogêneos Dissertação de Mestrado, UFPB, 1993.

[STEVENS90] STEVENS, W. R. - Unix Network Programming - Prentice Hall 1990.