UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

COMPARATIVO DA TECNOLOGIA DCOM EM DIVERSOS AMBIENTES DE DESENVOLVIMENTO

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

EGARD CHARLES ULLRICH

BLUMENAU, JULHO/2002

2002/1-28

COMPARATIVO DA TECNOLOGIA DCOM EM DIVERSOS AMBIENTES DE DESENVOLVIMENTO

EGARD CHARLES ULLRICH

ESTE TRABALHO DE CONCLUSÃO DE CURSO FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Maurício Capobianco Lopes — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Maurício Capobianco Lopes

_____

Prof. Dalton Solano dos Reis

_____

Prof. Marcel Hugo

AGRADECIMENTOS

Ao meu orientador, professor Maurício Capobianco Lopes, por sua constante dedicação e interesse demonstrado, pois sem ele este trabalho não poderia ser concretizado.

Aos meus pais Charles Felix Ullrich e Sônia Maria Felsky Ullrich, que sempre me deram apoio e incentivo, não somente neste trabalho, mas em toda minha vida. Agradeço também pela compreensão demonstrada por eles principalmente nas horas difíceis.

A todos meus amigos que me ajudaram e me deram incentivo contribuindo para conclusão desta importante etapa de minha vida. A presença de todos foi essencial para realização deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

RESUMO

Este trabalho tem como propósito demonstrar a utilização da tecnologia DCOM em diversos ambientes de desenvolvimento, fazendo um comparativo de tal tecnologia, tendo como resultado a implementação de diversos protótipos para testar a interoperabilidade entre os ambientes Microsoft Visual C++ 6.0, Borland Delphi 6, Microsoft Visual J++ 6.0 e Microsoft Visual Basic 6.0. Para isto foi desenvolvida uma aplicação no estilo *chat* que realiza a troca de informações através da comunicação entre objetos distribuídos.

ABSTRACT

This work aims to demonstrate the use of DCOM technology in several development environments, comparing that technology, getting as a result the implementation of many prototypes that perform tests of interoperability between Microsoft Visual C++ 6.0, Borland Delphi 6, Microsoft Visual J++ 6.0 and Microsoft Visual Basic 6.0 environments. For these tests was developed an application like a chat, which does information interchange using distributed objects for the communication.

1 INTRODUÇÃO

Segundo Mainetti (1997), a primeira influência da orientação a objetos (OO) ocorreu através das linguagens de programação. Durante os anos 80, praticamente todas as linguagens passaram a embutir os conceitos de OO. Linguagens como BASIC, Pascal, C, COBOL e LISP transformaram-se em linguagens como: Object Pascal, C++, Object COBOL e CLOS (*Common Lisp Object System*). Até mesmo as linguagens proprietárias e baseadas em *scripts*, de ferramentas gráficas de prototipação hoje são consideradas orientadas a objetos.

Há muito tempo, vários autores já vêm fazendo previsões sobre a influência da orientação a objetos nas várias áreas da computação. Se forem observados os acontecimentos destas áreas nos últimos anos, verifica-se que todas as previsões transformaram-se em realidade.

Mainetti (1997) também descreve que hoje, mesmo que um profissional da área de desenvolvimento de software queira desenvolver um novo sistema sem usar orientação a objetos, vai encontrar muitas dificuldades, pois praticamente todas as ferramentas de ponta atualmente usadas no processo de desenvolvimento de software são otimizadas para trabalhar usando os conceitos de orientação a objetos.

O desenvolvimento de software orientado a objetos tem um grande potencial para redução do tempo e do custo de desenvolvimento (reutilização), além de obter uma alta qualidade nos sistemas de software desenvolvidos por novos e não tão experientes engenheiros de softwares (Schmidt, 2000).

De acordo com estas necessidades e tendências, surgiram os chamados objetos distribuídos com o objetivo de cada vez mais aumentar a interoperabilidade e facilitar a comunicação entre sistemas, desde um mesmo computador, até a comunicação entre sistemas em pontos diferentes via internet, conforme relata MICROSOFT (2001).

Usar objetos distribuídos (OD) é basicamente usar a tecnologia de orientação a objetos em um ambiente distribuído. Como estas duas tecnologias já estão se transformando em realidade atual na grande maioria das empresas, trata-se de uma evolução natural. Os objetos distribuídos começaram sua trajetória através do *middleware* (a parte do software responsável

pela intercomunicação entre os vários componentes distribuídos em uma rede), mas hoje estão invadindo todas as áreas, inclusive a Internet (Mainetti, 1997).

Além disso, esses objetos distribuídos possuem as mesmas características principais dos objetos das linguagens de programação: encapsulamento, polimorfismo e herança, tendo dessa forma, as mesmas principais vantagens: fácil reusabilidade, manutenção e depuração (Capeletto, 1999).

Szyperski (1998) diz que para o desenvolvimento de softwares complexos que envolvem assuntos como projeto e implementação de objetos distribuídos, uma arquitetura se torna necessária. O mesmo ocorre no desenvolvimento de softwares a partir de componentes (*component system architecture*). Hoje existem várias arquiteturas para desenvolvimento e interoperabilidade de componentes, dentre as quais se destacam o *Common Object Request Broker Architecture* (CORBA) e o *Component Object Model* (COM). Sobre esta última é que será dado ênfase neste trabalho.

Há muitas razões pelas quais duas ou mais aplicações podem interagir, como trocar dados ou para uma controlar a outra. Contudo, COM não define o propósito para que as aplicações se comuniquem. COM apenas provê uma via padrão para duas ou mais aplicações interagirem, não se importando com o propósito dessa interação. *Object Linking and Embedding* (OLE) e *ActiveX* são exemplos de duas especificações industriais que definem características específicas pelas quais as aplicações devem interagir. OLE é uma especificação que descreve como objetos COM podem ser usados para criar e manipular documentos compostos. *ActiveX* é uma especificação que descreve como objetos COM podem ser usados na Internet (Redmond III, 1997). Já o DCOM (*Microsoft Distributed* COM) é uma extensão do COM para suportar comunicação entre objetos em diferentes computadores. Esta tecnologia, por vir da *Microsoft*, depende da plataforma que, no caso, é o *Windows*.

Mesmo no *Windows* a tecnologia COM é suportada por diversas linguagens e ambientes tais como Microsoft Visual C++ 6.0, Borland Delphi 6, Microsoft Visual J++ 6.0 e Microsoft Visual Basic 6.0 entre outros. Entretanto, para cada linguagem observa-se diferentes características de uso destes objetos. Desta forma, neste trabalho pretende-se avaliar a tecnologia COM e suas formas de implementação em diferentes linguagens (C, Object Pascal, Java e Visual Basic). Nesta avaliação pretende-se verificar principalmente as

ferramentas disponíveis para cada linguagem e a interoperabilidade dos objetos a serem desenvolvidos.

Para esta avaliação foi implementado um sistema de comunicação pessoal, estilo *chat* onde os usuários poderão enviar e receber mensagens, baseado em componentes e em sistemas distribuídos permitindo automatizar fluxos de informações digitadas pelo usuário. Existe um servidor de informações e vários clientes desenvolvidos em diversas linguagens de programação compatíveis com a tecnologia de orientação a objetos e a arquitetura COM/DCOM, com o objetivo de alcançar uma perfeita interoperabilidade de todos os objetos em questão, trabalhando juntos à medida que seus serviços se façam necessários. Foi utilizada a *Unified Modeling Language* (UML) para modelagem do sistema.

1.1 OBJETIVOS

O objetivo deste trabalho é implementar um sistema que permita comparar a forma de utilização e a funcionalidade da tecnologia COM/DCOM nos ambientes Microsoft Visual C++ 6.0, Borland Delphi 6, Microsoft Visual J++ 6.0 e Microsoft Visual Basic 6.0.

Os objetivos específicos do trabalho são:

a) construir um sistema que proporcione a comunicação pessoal entre vários usuários distribuídos por uma rede de computadores conectados a um servidor, o qual tem a função de gerenciar a recepção e transferência das informações de forma coerente;
b) desenvolver um sistema de mensagens em diversos ambientes de programação como Microsoft Visual C++ 6.0, Borland Delphi 6, Microsoft Visual J++ 6.0 e Microsoft Visual Basic 6.0, para realizarem tarefas de transmissão e recepção de mensagens de dados fornecidos pelos clientes, através da tecnologia DCOM.

1.2 ESTRUTURA

O presente trabalho foi estruturado da seguinte maneira:

O primeiro capítulo apresenta a contextualização e justificativa para o desenvolvimento da proposta do trabalho.

O segundo capítulo aborda a funcionalidade, características e conceitos básicos sobre Sistemas Distribuídos abrangendo também a tecnologia *Distributed Component Object*

Model (DCOM) que será tomada como base para a implementação do protótipo. Ainda neste capítulo serão demonstradas as ferramentas que suportam a tecnologia em questão.

O terceiro capítulo trata da especificação e detalhes da implementação do protótipo a ser desenvolvido.

O quarto capítulo apresenta as considerações finais, abrangendo as conclusões do desenvolvimento deste trabalho, as dificuldades encontradas e as sugestões para próximos trabalhos.

2 SISTEMAS DISTRIBUÍDOS

Segundo Mainetti (1997), existe uma “nova onda” ou “segunda geração” de aplicações cliente/servidor. Não se vive mais em um mundo ditado por computadores de grande porte, voltados para o processo centralizado. Vários computadores pessoais e estações de trabalho (*workstations*) tomaram conta dos ambientes de trabalho de todas as corporações e a integração entre estes computadores, buscando o processamento distribuído, é uma necessidade. Os sistemas que devem executar nestes ambientes têm características diferenciadas e para atender a estas novas características, o uso da tecnologia de orientação a objetos está passando a fazer parte do processo de desenvolvimento de software de todas as corporações.

Esta segunda geração de aplicações cliente/servidor é basicamente um ambiente onde o cliente não tem mais um papel único de ser apenas cliente e o servidor também não apresenta mais uma função única de apenas servir dados. Atualmente, nas empresas têm-se um ambiente onde as estações são computadores de grande capacidade de processamento com sistemas operacionais de 32-bits, multitarefa, *multithreading*, que suportam vários protocolos de rede nativamente e com uma interface gráfica intuitiva e de fácil uso, provendo mais recursos ao seu usuário. Da mesma forma, os servidores são computadores ainda mais poderosos e que, para prover todos os serviços solicitados pelos clientes, utilizam chamadas a outros servidores, espalhados pela rede. Assim todos os computadores na rede corporativa podem assumir tanto o papel de cliente quanto o de servidor, e para este tipo de ambiente dá-se o nome de Sistemas Distribuídos (Mainetti, 1997).

Um sistema distribuído pode ser conceituado, segundo Melo (1999), como qualquer sistema que possua múltiplas unidades centrais de processamento (UCP) interconectadas trabalhando em conjunto. Já Coulouris (1994), conceitua como uma coleção de computadores autônomos conectados através de uma rede e equipados com algum software de sistema distribuído. O software de sistema distribuído permite que os computadores coordenem suas atividades e compartilhem os recursos do sistema, como o hardware, o software e os dados. É importante que o sistema distribuído seja projetado de maneira que, embora composto por computadores geograficamente distribuídos, seja percebido como um único sistema integrado.

Mainetti (1997) fala da existência de várias arquiteturas e modelos de objetos distribuídos disponíveis no mercado, como *Distributed System Object Model* (DSOM) da IBM, *Portable Distributed Objects* (PDO) da NeXT, *Common Object Request Broker Architecture* (CORBA) da OMG e *Distributed Component Object Model* (DCOM) da *Microsoft* entre outros. O DCOM e o CORBA estão se transformando nos principais padrões do mercado, sendo que neste trabalho será dado ênfase ao padrão DCOM.

2.1 VANTAGENS E CARACTERÍSTICAS DOS SISTEMAS DISTRIBUÍDOS

O uso de computadores está se deslocando do uso de sistemas centralizados para o uso de sistemas com características distribuídas, por razões como: custo, performance e velocidade (Melo, 1999).

Quando usados apropriadamente, os sistemas distribuídos podem oferecer ganhos em relação aos sistemas centralizados. Algumas vantagens são:

a) performance através do processamento paralelo;
b) economia pela relação preço/performance em relação aos computadores de grande porte;
c) extensibilidade através da configuração e reconfiguração dinâmica;
d) relação custo/eficiência através do compartilhamento dos recursos do sistema operacional;
e) escalabilidade e portabilidade através da modularidade;
f) disponibilidade pelo fato da perda de uma máquina não provocar a parada do sistema.

Melo (1999) alerta que a formação de um sistema distribuído objetiva construir um sistema de alta performance, confiável, escalonável, consistente e seguro. Projetos de sistemas distribuídos devem corresponder a esta expectativa e cobrir todos os requisitos necessários para que o sistema seja considerado bom. As principais características de um bom sistema distribuído são:

a) compartilhamento de recursos: recursos podem ser itens de dados (arquivos, banco de dados, etc.), software ou componentes de hardware (discos, impressoras, etc.). Os recursos no sistema distribuído são fisicamente encapsulados dentro de um

computador e podem somente ser acessados por outro computador via comunicação;

b) concorrência: sistemas distribuídos são compostos de muitos computadores, cada um deles possuindo um ou mais processadores, proporcionando a execução de processos de maneira concorrente e paralela;
c) escalabilidade: Melo (1999) diz que um sistema distribuído é escalonável quando é possível aumentar o número de recursos compartilhados sem a necessidade de mudar o software de aplicação e sistemas;
d) tolerância a falhas: pelo fato de sistemas distribuídos possuírem múltiplas partes de hardware e de software funcionando em conjunto, as chances de algumas dessas partes falhar é bem maior do que ocorreria num sistema simples. Em contrapartida, uma falha em um nó não necessariamente compromete os demais. Para que o sistema seja confiável, ele deverá continuar funcionando corretamente mesmo com a presença de certos tipos de defeitos ou interferências indevidas (Coulouris, 1994);
e) transparência: a transparência está ligada a alguns aspectos do sistema distribuído que não são percebidos pelo usuário (programador, desenvolvedor de sistemas, usuário ou programa de aplicação). A transparência oculta alguns recursos do usuário e dos programadores de aplicações, dando a entender que se trata de uma única coleção de recursos (Melo, 1999);
f) performance: como em um sistema distribuído o processamento das instruções são divididos entre vários computadores, o tempo total para execução de uma tarefa é bem menor do que a execução desta tarefa em um sistema centralizado.

Aplicações seqüenciais em sistemas centralizados usam modelos tradicionais onde atividades e dados são manipulados isoladamente. Em sistemas distribuídos, atividades e dados estão intrinsecamente distribuídos e compartilhados. A distribuição produz modelos mais complexos e aplicações mais difíceis de desenvolver. A meta dos pesquisadores, hoje, é prover um modelo de transparência em sistemas distribuídos, análogo ao existente nos sistemas centralizados (Coulouris, 1994).

Segundo (Eddon, 1998), construir uma aplicação distribuída não é uma tarefa fácil. Na maioria dos casos é mais difícil do que desenvolver um sistema equivalente centralizado. Na

teoria, sistemas distribuídos podem ser habitualmente construídos em praticamente qualquer sistema operacional que ofereça funcionalidades de rede.

Na realidade, os serviços a seguir são geralmente requeridos pelo sistema operacional para construir um bom sistema distribuído:

a) rede de comunicação: quando trabalha-se com um protocolo de rede de nível baixo, deve-se estar preparado para tratar de muitos assuntos levantados em uma aplicação de rede, como por exemplo, os erros que ocorrem quando a conexão da rede é perdida;
b) multithreading: um servidor, para atender múltiplos usuários simultaneamente, deve ser construído com suporte multithread, então este servidor gera uma nova thread para tratar com cada cliente que se conecta;
c) segurança: segurança também torna-se um dos mais importantes fatores no projeto e implementação de qualquer sistema distribuído. Por exemplo:
    - como assegurar que os clientes realmente irão conectar-se onde pretendem?
    - como limitar seus acessos a apenas alguns serviços?
    - como permitir que um administrador faça a auditoria do sistema?

2.2 OBJETOS DISTRIBUÍDOS

Mainetti (1997) diz que usar objetos distribuídos (OD) é basicamente usar a tecnologia de orientação a objetos em um ambiente distribuído. Como estas duas tecnologias já estão se transformando em realidade atual na grande maioria das empresas, trata-se de uma evolução natural.

Grimes (1997) diz que um objeto remoto pode facilmente tornar-se cliente para outro objeto uma vez que, estes objetos remotos e as funcionalidades que eles oferecem podem ser distribuídos por uma rede.

Segundo Santos (1998), os objetos distribuídos surgiram devido a:

a) necessidade de alteração em múltiplos programas: sem a utilização de objetos distribuídos sempre que um objeto fosse alterado todos os programas que acessam este objeto teriam que sofrer as mesmas alterações;

b) comodidade em apenas instanciar os objetos: é mais cômodo apenas instanciar objetos já construídos anteriormente do que refazê-los em cada programa;
c) maior versatilidade de utilização de sistemas distribuídos: a idéia da utilização de objetos distribuídos pode facilmente ser implementada para atuar através de redes de computadores;
d) segurança: criadores de objetos precisam ter seus dados protegidos. Existe a necessidade de se criar um padrão para acessar os objetos com segurança, de um local remoto.

Santos (1998) explica que para que os objetos se entendam e possam trabalhar juntos é necessário que tenham interfaces comuns, ou pelo menos que possam conhecer as interfaces dos outros objetos comuns. Os modelos de objetos distribuídos como o modelo COM indicam uma linguagem de definição de interfaces chamada de *Interface Definition Language* (IDL), definida pelo fabricante. Esta IDL é independente da linguagem de programação, a qual é utilizada para definir as distintas interfaces dos objetos. A IDL define os tipos de objetos, seus atributos, suas funções (métodos) e os parâmetros. Uma IDL é puramente uma linguagem declarativa. As tecnologias de componentes são implementações do modelo de objetos, e se constroem utilizando sua IDL.

2.3 COMPONENT OBJECT MODEL (COM)

O COM surgiu a partir da tecnologia OLE que foi lançada inicialmente como parte do Windows 3.x. O próprio OLE é sucessor da *Dynamic Data Exchange* (DDE) a qual permitia aos usuários fazer transferência de dados complexos entre aplicações diferentes. O OLE 1.0 estendia o simples recurso de recortar e colar (*cut-and-paste*), permitindo aos usuários criar documentos compostos, como por exemplo, criar um documento no WORD incluindo um gráfico do EXCEL. Quando a Microsoft lançou as especificações para OLE 2.0 em 1993, isto incluiu uma especificação para ambos documentos o OLE e o OLE *Component Object Model* o qual forneceu uma infraestrutura para desenvolvimento de componentes reutilizáveis. Enquanto o OLE 1.0 era totalmente baseado em documentos, o OLE 2.0 era totalmente baseado em serviços. A infraestrutura do OLE 2.0 cresceu no que é conhecido hoje como COM, que é uma especificação de software e algumas bibliotecas de suporte. O COM permite que um desenvolvedor independente possa criar seus próprios componentes e pode estar

confiante que os componentes irão funcionar similarmente com outros objetos COM compatíveis criados por outros desenvolvedores.

O *Component Object Model* (COM) é definido como uma arquitetura de software componentizada que permite que aplicações e sistemas sejam construídas a partir de componentes fornecidos por diferentes distribuidores de softwares. COM é a arquitetura fundamental que consolida a base para serviços de software de alto nível, como aqueles fornecidos pela *Object Linking and Embedding* (OLE). Os serviços OLE transpõem a vários aspectos de um software componentizado, incluindo documentos compostos, controles personalizados, transferência de dados e outras interações de softwares (MICROSOFT, 2001).

Grimes (1998) define COM como uma especificação e uma série de serviços que permitem criar objetos modulados, componentes orientados a objeto, componentes personalizáveis e possíveis de serem atualizados e aplicações distribuídas utilizando várias linguagens. Grimes (1998) explica melhor a definição de COM por meio de algumas características da tecnologia, tais como:

a) série de serviços: a especificação é apoiada por um grupo de serviços ou A*pplication Programming Interface* (API). Estes serviços são fornecidos pela biblioteca COM (*COM library*), o qual é parte do sistema operacional para plataforma Win32 e é disponibilizado como um pacote separado para outros sistemas operacionais;
b) segue uma programação modular: componentes COM podem ser empacotados em arquivos EXE ou DLL;
c) orientado a objetos: componentes COM são considerados objetos pois tem identificação, estado e comportamento. Os componentes COM que implementam uma interface comum podem ser tratados de modo polimórfico;
d) permite fácil personalização e atualizações para as aplicações: componentes COM ligam-se uns com os outros dinamicamente. Existe uma forma padrão para localizar componentes e identificar suas funcionalidades. Assim, componentes isolados são trocados sem a necessidade de recompilacão de toda a aplicação;
e) permite aplicações distribuídas: o COM fornece um mecanismo de comunicação que permite que componentes interajam por meio de uma rede de comunicação, o padrão DCOM. Ele também fornece transparência de localização para aplicações,

permitindo que estas sejam escritas sem se preocupar com a localização de seus componentes. Os componentes podem ser movidos sem exigir qualquer mudança na aplicação;

f) podem ser escritos em várias linguagens: como COM é um padrão binário, qualquer linguagem que lida com um padrão binário pode criar ou usar objetos COM. O número de linguagens e ferramentas que suportam COM estão crescendo a cada dia, como por exemplo C, C++, Java, JScript, Visual Basic, VBScript, Delphi, PowerBuilder e MicroFocus Cobol.

COM é um mecanismo independente, que possui a sua própria *Interface Definition Language* (IDL), que é usada para fornecer definições completas sobre as interfaces do tipo COM. A IDL tem a função de associar os métodos de uma interface com suas *interface identifier* (IID), permitindo especificar detalhes de uma interface de uma forma que possa ser processada por uma máquina, gerando assim um código organizado (Grimes, 1998).

Conforme relata Geraghty (1999), quando se fala de COM, automaticamente estas regras também valem para DCOM, ficando a diferença entre as duas especificações apenas no nível da implementação interna do próprio modelo.

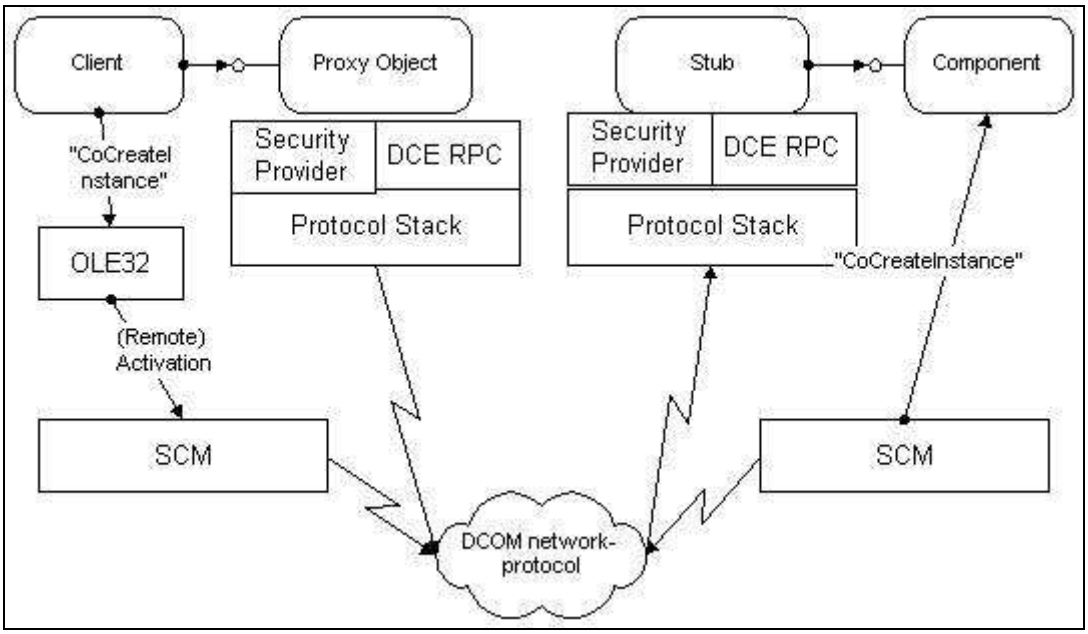
2.4 DISTRIBUTED COMPONENT OBJECT MODEL (DCOM)

O DCOM é uma extensão do COM para suportar a comunicação entre objetos em diferentes computadores em uma *Local Area Network* (LAN), uma *Wide Area Network* (WAN) ou mesmo na internet.

Segundo MICROSOFT (2001), a especificação DCOM define como os componentes COM e seus clientes interagem. Esta interação é definida para que o cliente e o componente COM possam interagir sem a necessidade de nenhum serviço intermediário do sistema. Nos sistemas operacionais atuais, os processos são independentes uns dos outros. Um cliente que necessita comunicar-se com um componente em um outro processo não pode fazer uma chamada a este componente diretamente, precisando fazer uso de alguma forma de comunicação entre processos fornecido pelo sistema operacional. O modelo COM fornece este tipo de comunicação transparente, interceptando as chamadas do cliente e repassando-as para o componente em outro processo.

A Figura 1 ilustra a arquitetura DCOM. A implementação DCOM fornece serviços de orientação a objetos aos clientes e componentes, e usa *Remote Procedure Calls* (RPC) e um serviço de segurança (*Security Provider*) com o objetivo de gerar pacotes de rede que obedeçam o padrão DCOM, além de um serviço chamado de *Service Control Manager* (SCM) para fazer a requisição da criação de objetos.

Figura 1: Arquitetura DCOM



Os objetos COM são identificados por uma identificação de classe chamada *Class Identifier* (CLSID). Quando um cliente deseja instanciar e comunicar-se com um componente, ele passa esta identificação no método *CoCreateInstance* contido em uma API COM, criando uma instância de tal objeto. Com a chamada deste método, é invocado o *COM´s Implementation Location Service*, que é implementado na forma de um *Service Control Manager* (SCM). Este serviço é responsável por localizar o servidor apropriado para um objeto COM identificado na chamada do cliente pelo CLSID e por carregar o servidor COM.

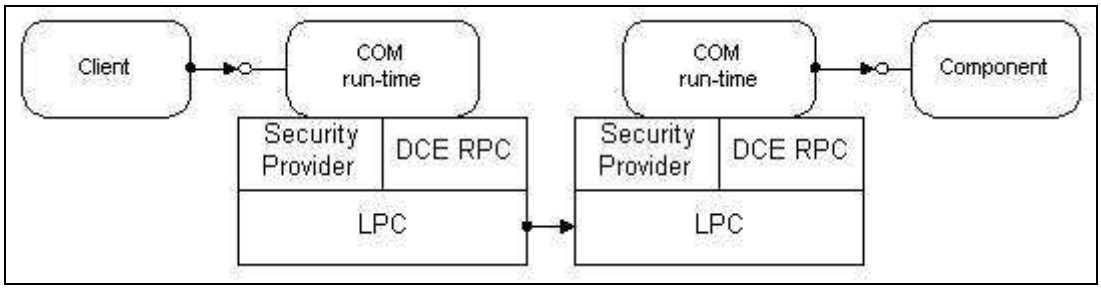
Juntamente com o componente, é carregada uma peça de software especial do tipo *in-process* chamado *proxy*, responsável por pegar a posição do objeto e passar todas as requisições do cliente para outra peça de software chamada de s*tub*. Portanto, como o *proxy* é *in-process*, isto é, é carregado em tempo de execução e no mesmo espaço de memória do processo que o chamou, o ponteiro para interface do cliente pode acessá-lo e para o cliente o

proxy parece ser o próprio objeto. O *proxy* também é responsável por empacotar todos os parâmetros que são necessários para invocar um método em particular, num processo conhecido por *marshaling*.

O *stub* também é *in-process*, mas está localizado no espaço do processo do servidor. O *stub* recebe requisições do *proxy* e descompacta (*unmarshal*) todos os parâmetros antes de invocar o método da interface. Pode-se dizer, então, que o *proxy* e o *stub* fazem uma ponte entre o cliente e servidor para que estes possam se comunicar.

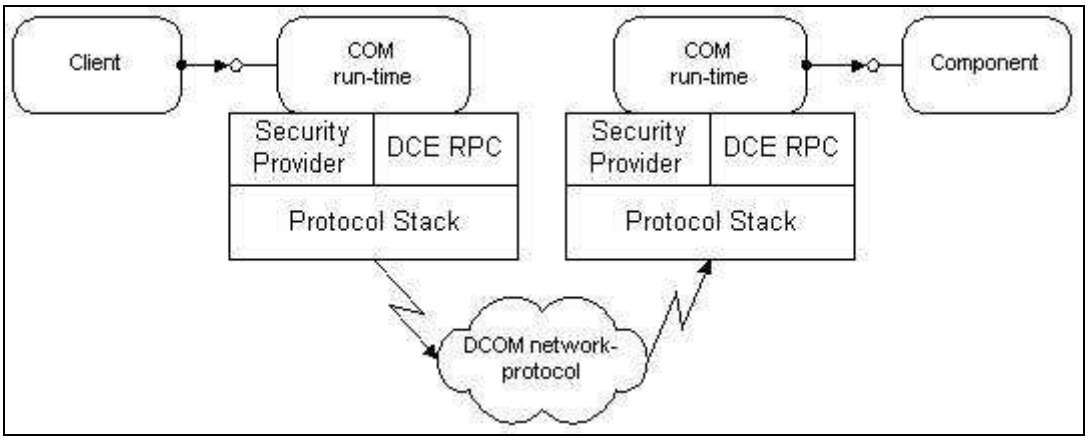
Quando se está acessando um servidor local e o *proxy* e o *stub* estão localizados na mesma máquina a comunicação ocorre via *Local Procedure Calls* (LPC) como demonstrado na Figura 2. Uma LPC é uma forma de comunicação entre processos especialmente desenhada para permitir a um processo invocar os métodos de outro processo.

Figura 2: Componentes COM na mesma máquina



Quando o *proxy* e o *stub* estão localizados em máquinas diferentes estes se comunicam via *Remote Procedure Calls* (RPC). As RPC assim como as LPC também são uma forma de comunicação entre processos, mas são designadas a permitir que um processo em uma máquina invoque os métodos de um processo localizado em uma outra máquina através de um protocolo de rede DCOM (*DCOM network-protocol*) como demonstrado na Figura 3.

Figura 3: Componentes COM em máquinas diferentes



2.5 *CALLBACK* UTILIZANDO *CONNECTION POINTS*

A maioria dos componentes são criados na forma de comunicação entre o cliente e o servidor em apenas um sentido, do cliente para o servidor. Mas às vezes ocorrem eventos no servidor que é necessário ter-se conhecimento no cliente. Este processo é chamado de *callback*. Uma maneira de obter isso é utilizando *Connection Points* (DEVBRASIL, 2001). Um *Connection Point* é o ponto onde um objeto invoca novamente o cliente que o chamou. Um objeto que tem a capacidade de chamar novamente o seu cliente é considerado um objeto conectável ou *container*. Para um objeto ser um *connectable object* ele deveria implementar uma interface do tipo *IConnectionPointContainer*.

Existem dois métodos em uma interface do tipo *IConnectionPointContainer*, demonstrada no Quadro1, chamados de *EnumConnectionPoints* e *FindConnectionPoint*. O primeiro fornece uma lista de todos os *connection points* suportados no objeto *container* e o segundo método possui um GUID (*Globally Unique Identifier*) que retorna um ponteiro para a interface *IConnectionPoint*.

Quadro 1: Interface *IConnectionPointContainer*

```
public interface IConnectionPointContainer extends IUnknown
{
  // Methods
  public IEnumConnectionPoints EnumConnectionPoints();
  public IConnectionPoint FindConnectionPoint(_Guid riid);

 }
```

Se um cliente quer receber eventos de volta do servidor deverá ser implementada uma interface de saída (*outgoing*) editada pelo servidor. A implementação desta interface de saída tem o nome de *sink*. Sempre que esta *sink* é implementada é necessário avisar ao servidor, através do método *Advise* da interface *IConnectionPoint*.

Uma vez que um servidor pode ter vários outros clientes, ele irá identificar seu *sink* por um dado de retorno chamado *cookie*. Quando for necessário que haja a desconexão, este *cookie* será usado e será feita uma chamada do método *UnAdvise* no servidor (CODEPROJECT, 2002).

2.6 TIPOS DE INTERFACES COM

Conforme diz Grimes (1998), as propriedades de um componente COM podem apenas ser acessadas através de suas interfaces. Logo, o único procedimento que um cliente pode fazer para ter um ponteiro para uma nova interface é chamar um método de uma interface que ele já possui. Sendo assim, todos os componentes COM devem suportar ao menos uma interface padrão que contém um método que permite retornar ponteiros para outras interfaces naquele objeto. A esta interface padrão dá-se o nome de interface *IUknown* e o método responsável por retornar ponteiros para as outras interfaces do componente chama-se *QueryInterface()*. Grimes (1998) diz ainda que a interface *IUknown* demonstrada no Quadro 2 possui outros dois métodos chamados de *AddRef()* e *Release()*. Tais métodos são usados respectivamente para incrementar e decrementar a quantidade de referências de uma interface.

Quadro 2: Definição da interface *IUnknown* (em Visual C++)

```
MIDL_INTERFACE("00000000-0000-0000-C000-000000000046")
IUnknown
{
  public:
  BEGIN_INTERFACE
  virtual HRESULT STDMETHODCALLTYPE QueryInterface(
  /* [in] */ REFIID riid,
  /* [iid_is][out] */ void __RPC_FAR *__RPC_FAR *ppvObject) = 0;

  virtual ULONG STDMETHODCALLTYPE AddRef( void) = 0;
  virtual ULONG STDMETHODCALLTYPE Release( void) = 0;

  END_INTERFACE
};
```

Grimes (1998) indica que COM possui dois amplos tipos de interfaces categorizadas de acordo com a maneira que seus métodos são acessados, ou por meio de invocação estática ou invocação dinâmica. A invocação estática é o mecanismo usado pelas interfaces padrões, como a interface *IUnknown*, e é a ligação entre o cliente e o objeto servidor. Neste caso o cliente sabe exatamente a quantidade e o nome dos métodos em uma interface.

A invocação dinâmica é o meio pelo qual as interfaces mecanizadas (*Automation Interfaces*) fazem suas transações. Basicamente, o *Automation* permite ao cliente requisitar um objeto para que ele retorne as informações sobre as interfaces que ele suporta e quando uma destas interfaces for requisitada ele retorna informações sobre os métodos contidos na interface.

Objetos do tipo *Automation* utilizam uma interface chamada *IDispatch*. Logo, objetos servidores implementam *Automation* permitindo que os clientes usem um método na interface *IDispatch* para requisitar outros métodos. O conjunto de métodos que disponibiliza este processo chama-se *dispinterface*. Todos os métodos *Automation* devem usar um pacote limitado de tipos de dados, chamado de *Automation-compatible data types*. A vantagem de se utilizar estes tipos de dados é que eles são amplamente aceitos por diversas linguagens, o que é o caso do sistema proposto neste trabalho.

Como já foi mencionado anteriormente as interfaces do tipo *Automation* usam invocação dinâmica, porém, uma vez que o cliente deva utilizar alguma interface para chamar o objeto, a invocação não pode ser completamente dinâmica. Logo, objetos do tipo *Automation* devem também implementar a interface *IUnknown* além da interface *IDispatch* como apresentado no Quadro 3.

Quadro 3: Definição da interface *IDispatch* (em Visual C++)

```
MIDL_INTERFACE("00020400-0000-0000-C000-000000000046")
IDispatch : public IUnknown
{
  public:
  virtual HRESULT STDMETHODCALLTYPE GetTypeInfoCount(
          /* [out] */ UINT __RPC_FAR *pctinfo) = 0;

  virtual HRESULT STDMETHODCALLTYPE GetTypeInfo(
          /* [in] */ UINT iTInfo,
          /* [in] */ LCID lcid,
          /* [out] */ ITypeInfo __RPC_FAR *__RPC_FAR *ppTInfo) = 0;

  virtual HRESULT STDMETHODCALLTYPE GetIDsOfNames(
          /* [in] */ REFIID riid,
          /* [size_is][in] */ LPOLESTR __RPC_FAR *rgszNames,
          /* [in] */ UINT cNames,
          /* [in] */ LCID lcid,
          /* [size_is][out] */ DISPID __RPC_FAR *rgDispId) = 0;

  virtual /* [local] */ HRESULT STDMETHODCALLTYPE Invoke(
          /* [in] */ DISPID dispIdMember,
          /* [in] */ REFIID riid,
          /* [in] */ LCID lcid,
          /* [in] */ WORD wFlags,
          /* [out][in] */ DISPPARAMS __RPC_FAR *pDispParams,
          /* [out] */ VARIANT __RPC_FAR *pVarResult,
          /* [out] */ EXCEPINFO __RPC_FAR *pExcepInfo,
          /* [out] */ UINT __RPC_FAR *puArgErr) = 0;
      };
```

A coleção de métodos e propriedades suportados pela interface *dispinterface* podem ser acessadas através do método *Invoke()* da interface *IDispatch*, demonstrado também no Quadro 3. Este método passa para o servidor um identificador indicando o método que se deseja invocar juntamente com uma lista de parâmetros que se deseja passar ao método. O objeto pode então receber estes parâmetros e disparar a chamada para o método apropriado. Uma vez que o cliente conhece os métodos e propriedades que um objeto suporta ele usa o método *Invoke()* para informar ao objeto que ele deve invocar um método.

2.7 UTILIZAÇÃO E FUNCIONAMENTO DO DCOMCNFG.EXE

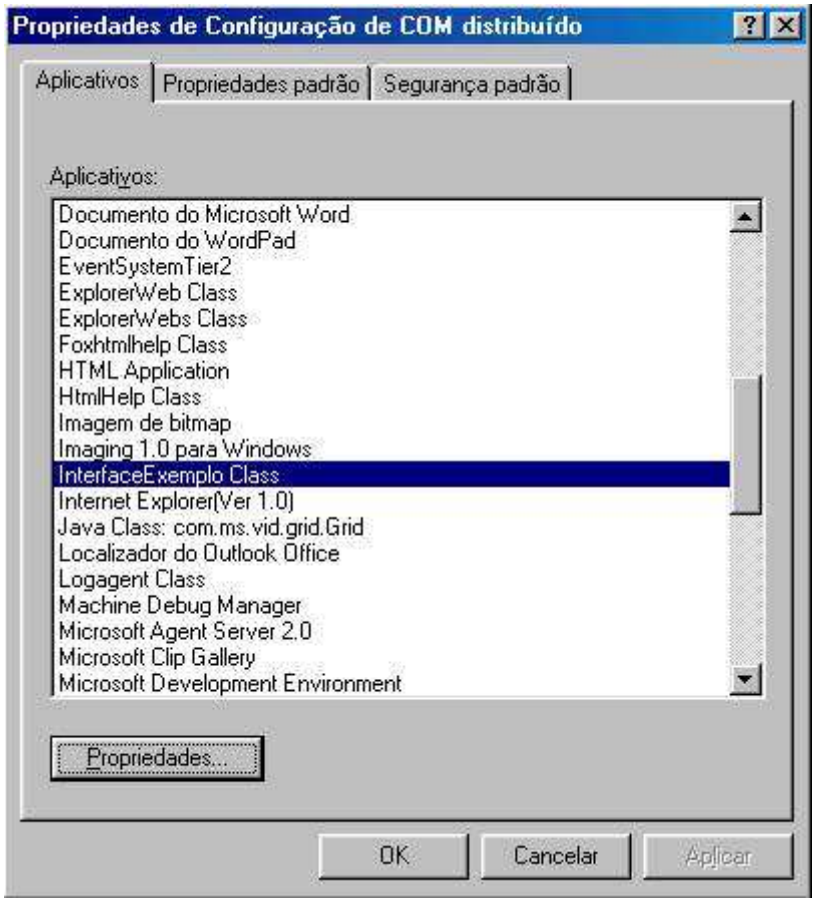
Segundo Eddon (1998), COM possui um modelo de segurança (*COM Security Model*) que tem os seguintes objetivos primários:

a) controle de ativação (*Activation Control*): especifica quem tem permissão para executar componentes;
b) controle de acesso (*Acess Control*): uma vez que um componente foi executado o *Acess Control* é usado para controlar o acesso dos objetos do componente. Em alguns casos isto pode ser aceitável para que certos usuários tenham acesso a certas áreas de funcionalidade, enquanto outros serviços de um componente estão restritos;
c) controle de autenticação (*Authentication Control*): é usado para assegurar que uma transmissão via rede é autêntica e para proteger os dados de usuários não autorizados;
d) controle de identidade (*Identity Control*): especifica as credenciais de segurança sob a qual o componente irá executar. Estas credenciais podem ser uma conta específica de usuário configurada para esta finalidade ou a credencial de segurança da aplicação cliente.

Para que se possa utilizar componentes COM distribuídos em máquinas diferentes, é necessário que este modelo de segurança COM esteja configurado de forma correta. Segundo Eddon (1998), existem duas maneiras de fazer estas configurações, a *declarative security* e a *programmatic security*.

Os ajustes das configurações na forma *declarative security* são feitos no *registry* para cada componente. Os ajustes das configurações na forma *programmatic security* são feitos incorporados ao componente pelo desenvolvedor. Ajustes de configuração de segurança de ativação, acesso, autenticação e identidade para um componente, podem ser configurados na forma *declarative security* via *registry* usando-se um utilitário como o *DCOM Configuration utility* (*dcomcnfg.exe* – Figura 4), (o qual será usado para este trabalho). A segurança de acesso e autenticação pode também ser controlada programaticamente usando-se algumas interfaces e funções auxiliares do COM. A segurança de ativação e identidade não podem ser controladas programaticamente por que estes ajustes devem ser especificados antes do componente ser executado.

Figura 4: Dcomcnfg.exe



2.8 FERRAMENTAS QUE SUPORTAM A TECNOLOGIA DCOM

Por ser uma tecnologia aberta pelo fabricante o DCOM permite que seus recursos sejam utilizados por desenvolvedores em várias ferramentas de desenvolvimento, distribuídas por diversos fornecedores incluindo a Microsoft, Borland, Powersoft/Sybase, Symantec, ORACLE, IBM e Micro Focus. Estas ferramentas e as aplicações que elas produzem já tem automaticamente suporte para DCOM.

Como o desenvolvimento e comparativo da tecnologia DCOM no trabalho em questão utilizará as ferramentas Visual C++ 6.0, Visual J++ 6.0, Visual Basic 6.0 e Delphi 6, será apresentado a seguir as características do DCOM nestes ambientes. Na ferramenta Visual C++ 6.0 foram desenvolvidos tanto o servidor, que tem a função de gerenciar a transmissão e recepção dos dados, quanto o cliente que é a parte visual da aplicação que interage com o

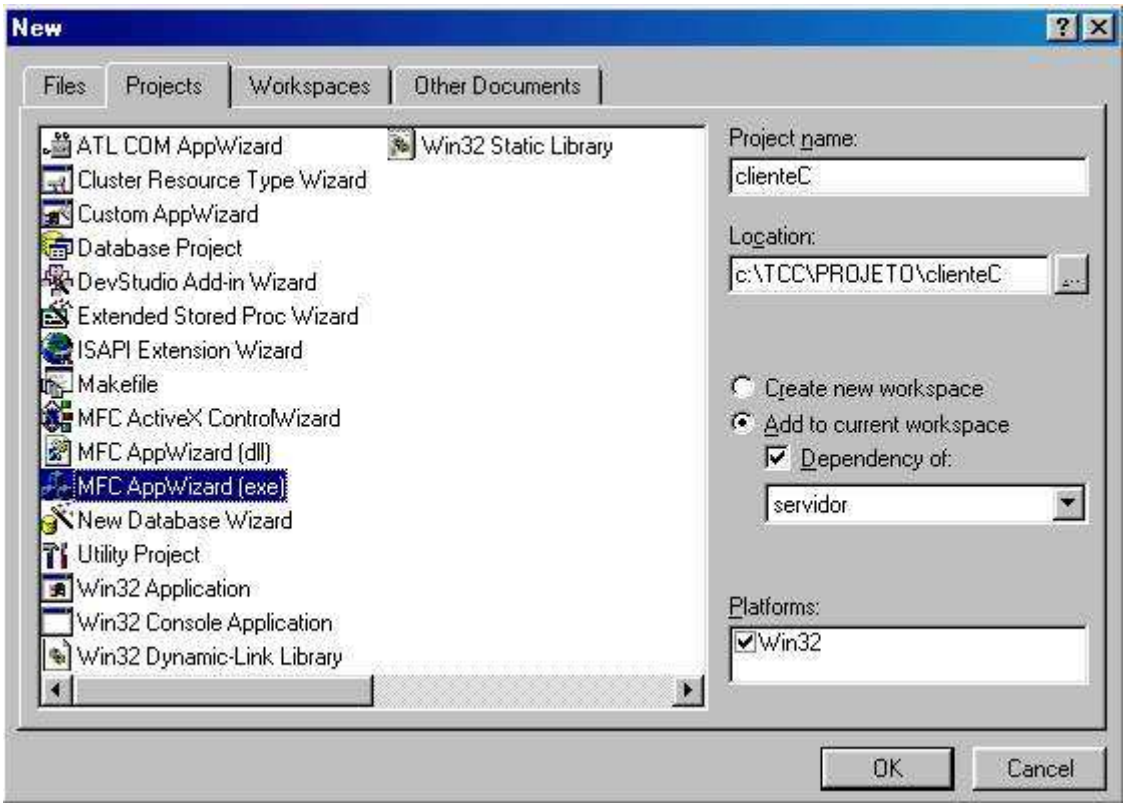
usuário. Nas demais ferramentas, foram desenvolvidos apenas clientes, já que estes trabalharam em conjunto com o servidor desenvolvido em Visual C++ 6.0.

2.8.1 DCOM EM MICROSOFT VISUAL C++

Grimes (1998) diz que para utilizar a biblioteca COM e realizar as chamadas de suas funções deve-se primeiro inicializá-la. Isto se aplica tanto a clientes quanto a servidores. Esta inicialização deve ser feita apenas uma vez para cada *thread* na qual irá usar-se a biblioteca COM. Para isso é feita uma chamada do método *CoInitialize()* ou *CoInitializeEx()* e para terminar a utilização da biblioteca usa-se o método *CoUninitialize()*. No Visual C++ 6.0 estes métodos são chamados explicitamente pelo desenvolvedor.

O cliente desenvolvido em Visual C++ foi criado como uma simples aplicação executável do tipo *MFC AppWizard* como demonstrado na Figura 5.

Figura 5: Criação do cliente em Visual C++ do tipo MFC *AppWizard* (.exe)



Rogerson (1997) explica que a biblioteca COM contém um método chamado *CoCreateInstance*, que também é chamado explicitamente pelo desenvolvedor. Ele serve para

criar os componentes. O *CoCreateInstance* cria uma instância do componente correspondente retornando um ponteiro para uma interface deste componente.

Rogerson (1997) diz ainda que o método *CoCreateInstance* tem quatro parâmetros de entrada e um de retorno. O primeiro parâmetro é o CLSID do componente a ser criado. O segundo parâmetro é usado para agregar componentes. Se for nulo (*NULL*) indica que o objeto não foi criado como parte de uma agregação. O terceiro parâmetro, *dwClsContext* limita o contexto da execução do componente ao qual o cliente pode conectar. COM fornece suporte para três tipos de componentes:

a) *in-process:* ocorre como uma biblioteca dinâmica (DLL);
b) local: quando o cliente e o servidor são executados na mesma máquina;
c) remoto: quando o cliente e o servidor são executados em máquinas diferentes.

O quarto parâmetro é o IID da interface utilizada no componente. Um ponteiro para esta interface é retornado no último parâmetro chamado de pIX. Passando um IID para o *CoCreateInstance*, após criar o componente, o cliente não precisa chamar o método *QueryInterface*, que será explicado a seguir. No Quadro 4 é demonstrado um exemplo de uma chamada para o método *CoCreateInstance*.

Quadro 4: Criação de um componente através do método *CoCreateInstance*

```
// Create Component
IX* pIX = NULL;
HRESULT hr = ::CoCreateInstance(CLSID_Component1,
                                NULL,
                                CLSCTX_REMOTE_SERVER,
                                IID_IX,
                                (void**)&pIX);
```

Para o exemplo demonstrado no Quadro 4, foi criado um componente identificado pelo *CLSID_Component1*. Como não foram usados componentes agregados o segundo parâmetro é nulo. O terceiro parâmetro *CLSCTX_REMOTE_SERVER* possibilita ao *CoCreateInstance* carregar componentes em máquinas diferentes para uso distribuído (*DCOM*). No quarto parâmetro é passado o *IID_IX* para requisitar a interface *IX* a qual retornará no último parâmetro, a variável *pIX*. Se a chamada *CoCreateInstance* for bem sucedida a interface *IX* estará pronta para ser usada.

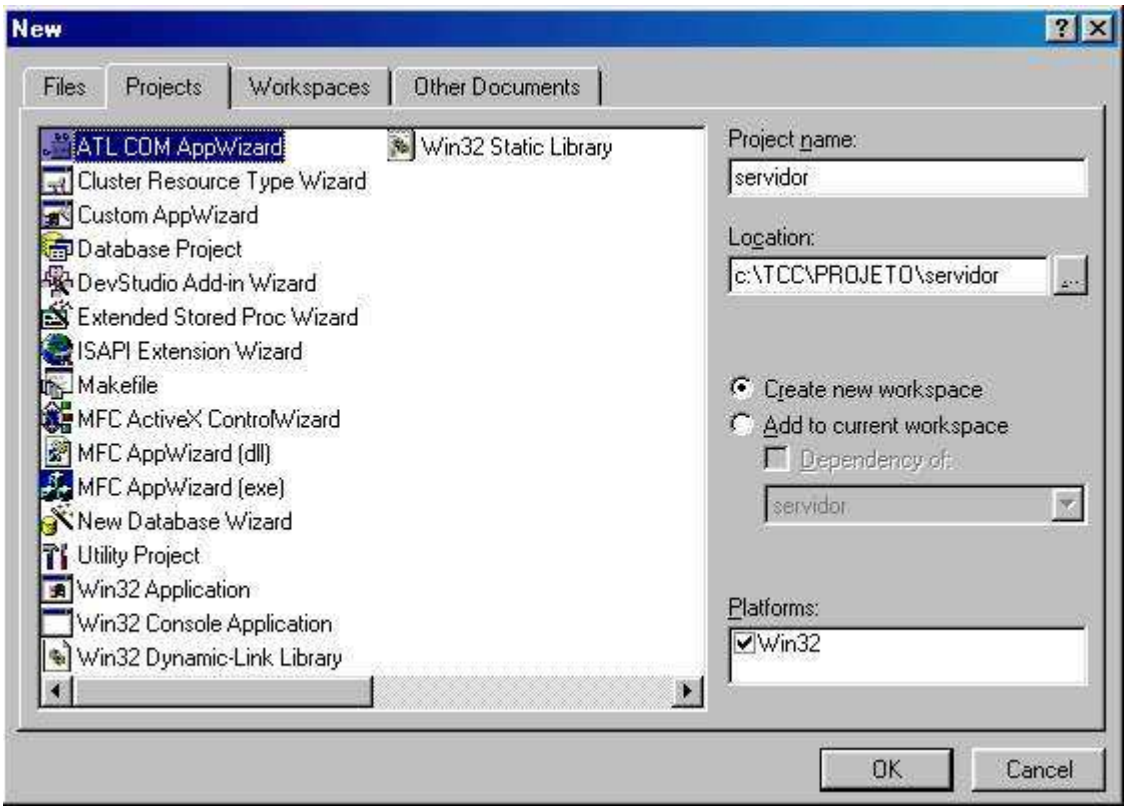
Eddon (1998) diz que cada objeto COM suporta uma interface do tipo *IUnknown* e um ponteiro para ela pode ser fornecido pelo *CoCreateInstance*. O método *QueryInterface* tem como objetivo determinar que outras interfaces um objeto pode suportar. No Quadro 5 o método *QueryInterface* é usado para determinar se um objeto suporta a interface *IID_IX*, por exemplo.

Quadro 5: Exemplo da utilização do método *QueryInterface*

```
HRESULT hr = pUnknown->QueryInterface(IID_IX, (void**)&pIX)
if (FAILED(hr))
{
      // Interface IID_IX não é suportada!!!
}
```

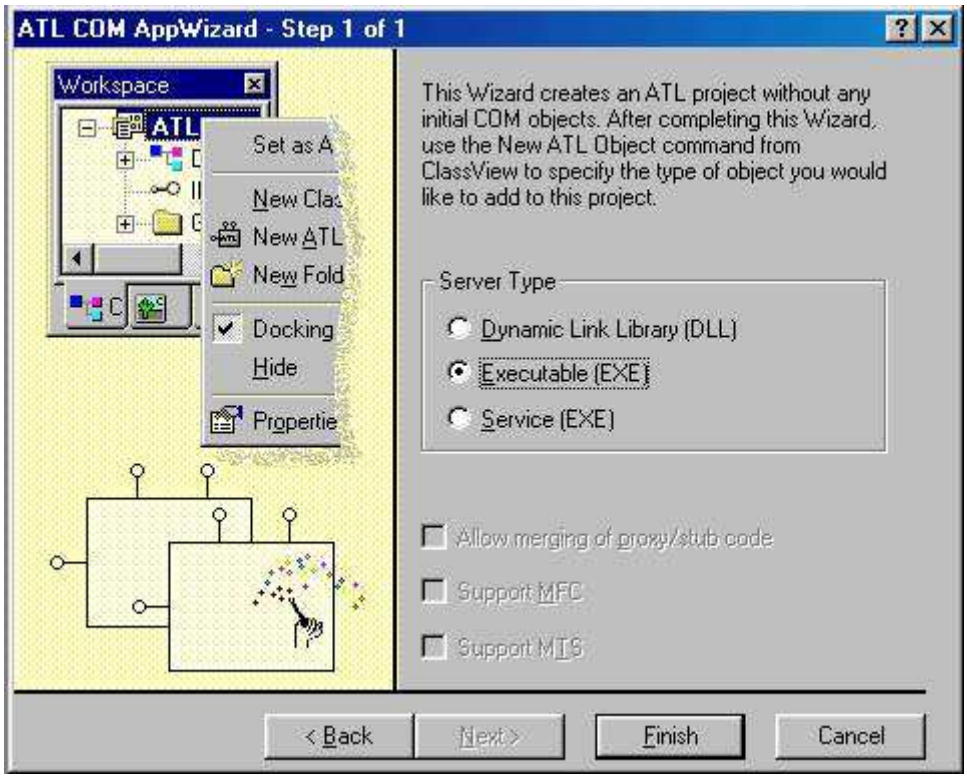
Para construção do servidor é utilizado o tipo de aplicação *ATL (Active Template Library) COM AppWizard* demonstrado na Figura 6.

Figura 6: Criação do servidor do tipo *ATL COM AppWizard*



O *ATL COM AppWizard* em Visual C++ gera todos os arquivos iniciais, permitindo desenvolver-se servidores *in-process* (DLL), servidores locais e servidores como serviços NT. Para esta aplicação foi desenvolvido um servidor do tipo executável (.exe) como demonstrado na Figura 7.

Figura 7: Escolhendo o tipo de servidor como um executável (.exe) por meio do *ATL COM AppWizard*



Além disso, o Visual C++ 6.0 possui um recurso chamado de *ATL ObjectWizard*, apresentado na Figura 8, o qual permite adicionar diferentes tipos de objetos COM em uma aplicação. Tal recurso gera automaticamente os nomes das principais classes, interfaces e nome dos arquivos do projeto a partir do nome do projeto, que é informado pelo desenvolvedor no campo *Short Name*.

Eddon (1998) mostra que dependendo do tipo de objeto escolhido, o *ATL ObjectWizard* apresenta um grupo de opções que permite a seleção do modelo de *thread* (*Threading Model*) suportado pelo objeto.

Figura 8: ***ATL ObjectWizard***

O *ObjectWizard* também permite a criação de diferentes tipos de interfaces. Para o caso do componente ter suporte para interface *IDispatch* deve-se optar pelo tipo *Dual*, caso contrário deve-se escolher *Custom*. É possível ainda especificar se um objeto possui agregação (Aggregation) e suporte a *Connection Points*.

2.8.2 DCOM EM MICROSOFT VISUAL J++

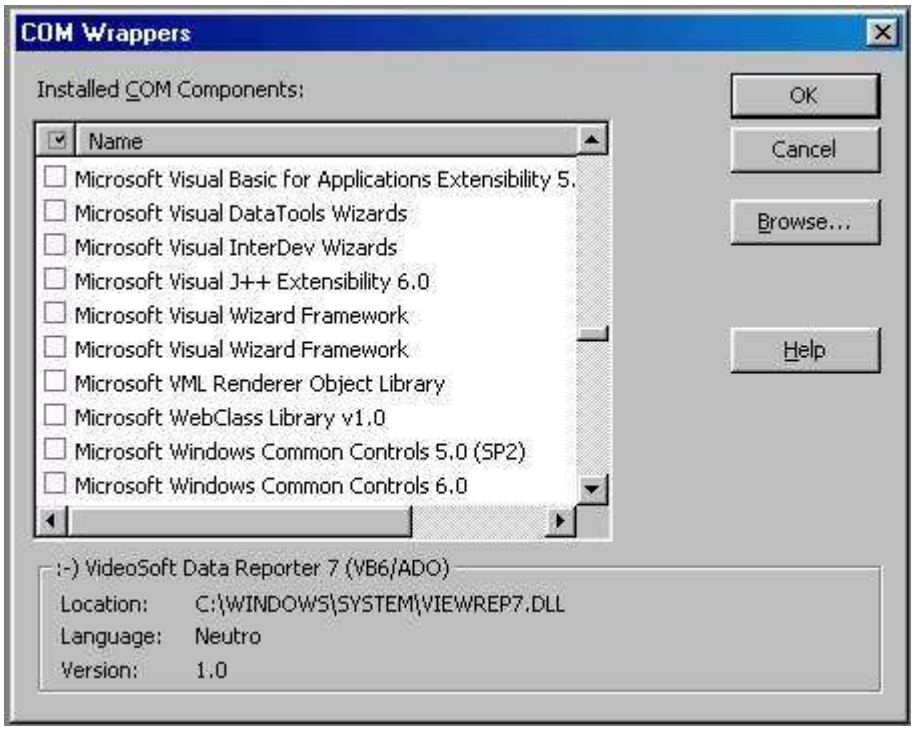
Segundo Eddon (1998), Java é uma linguagem de programação moderna e orientada a objetos. Apesar de serem desenvolvidas de forma completamente independentes COM e Java são tecnologias complementares. COM é uma arquitetura para desenvolver componentes e Java é uma linguagem de programação na qual pode-se construir e usar tais componentes.

A integração entre Java e COM vem se completando sem a necessidade de se adicionar nenhuma nova palavra chave ou estrutura à linguagem Java. Java já tem as construções necessárias que permitem que ela implemente e utilize componentes COM.

Eddon (1998) diz que é importante entender que o modelo de integração entre COM e Java é baseado em *type libraries*, ou seja, um programa feito em Java pode chamar qualquer objeto COM desde que esta informação esteja disponível na *type library*. Segundo Codelines (2000), uma *type library* é um arquivo contendo definições de funções e classes de objetos que podem ser usadas em linguagens compatíveis com o COM. Usando *type libraries*, o programador pode ter acesso a classes de objetos e a funções exportadas por DLL sem ter que criar declarações dessas funções. Eddon (1998) diz que esta informação disponível na *type library* pode ser fornecida como um arquivo *.tlb* ou como um fonte dentro do componente. No

Visual J++ 6.0, usado neste trabalho, para importar a *type library* foi utilizado o *COM Wrappers* como na Figura 9, criando-se assim um pacote de classes compatíveis com o Java contendo as classes COM no componente. Isto é possível selecionando-se na lista de componentes COM instalados, os objetos COM a serem utilizados. Assim automaticamente as informações da *type library* serão convertidas para arquivos *.class* (no Java, arquivos com extensão *.java* são arquivos fontes e arquivos com extensão *.class* são arquivos binários compilados).

Figura 9: Importando objetos COM em J++ usando o *COM Wrappers*



Segundo Eddon (1998), cada arquivo *.class* gerado contém um atributo especial identificando-o como um pacote para uma classe COM. Quando o *Microsoft Java VM* (msjava.dll) identifica este atributo em uma classe, ele traduz todas as chamadas dos métodos Java na classe dentro das chamadas para a classe COM. Assim, pode-se dizer que a *Microsoft Java VM* é a ponte de ligação entre Java e COM.

Quando se está usando um pacote Java para objetos COM, sempre deve-se chamar os métodos através de uma interface e não através da própria classe. No Quadro 6 é apresentado um exemplo de um programa que faz uma chamada de um objeto COM com nome de

InsideDCOM, por exemplo, supondo que este objeto tenha sido empacotado nas classes Java utilizando o *COM Wrappers.*

Quadro 6: Instanciação de um objeto COM em Java

```
Import component.*;
Class TestingTheCOMComponent
{
      public static void main(String str[])
      {
            ISum myRef = (ISum)new InsideDCOM();
      }
}
```

No exemplo, o método *new* instancia a classe usada como exemplo com nome de *InsideDCOM*. Esta instanciação faz com que o *Java VM* encontre no arquivo *InsideDCOM.class* o atributo especial, indicando que isto é uma classe COM. O arquivo *.class* gerado também contém o CLSID da classe COM. O *Java VM* então usa este CLSID para fazer a chamada do método *CoCreateInstance* do COM para instanciar o objeto da classe. O ponteiro *IUnknown* retornado pelo *CoCreateInstance* é então pré-movido pelo código Java para o tipo *ISum*.

A chamada da função *IUnknown::QueryInterface* é feita implicitamente. Ela permite aos clientes escolher entre as interfaces contidas no objeto. O simples *typecast* (conversão de um tipo de variável para outro) da classe *InsideDCOM* para a interface *ISum* força o Java VM a chamar a função *QueryInterface* do objeto para requisitar um ponteiro para *ISum*. Desta forma, um programa feito em Java pode devolver um ponteiro de interface para qualquer interface suportada pelo objeto COM.

Em Java não é preciso fazer a chamada *IUnknown::Release* para terminar de usar uma referência para um objeto, uma vez que quando a referência para o objeto sai do escopo o método *Release* é chamado automaticamente pelo coletor de lixo (*garbage colector*) nativo do Java.

2.8.3 DCOM EM VISUAL BASIC

Segundo Eddon (1998), Visual Basic é talvez a melhor linguagem para criar as mais complexas variações dos objetos COM, ou seja, Controles *ActiveX* e documentos *ActiveX*, para utilização de componentes COM via Internet.

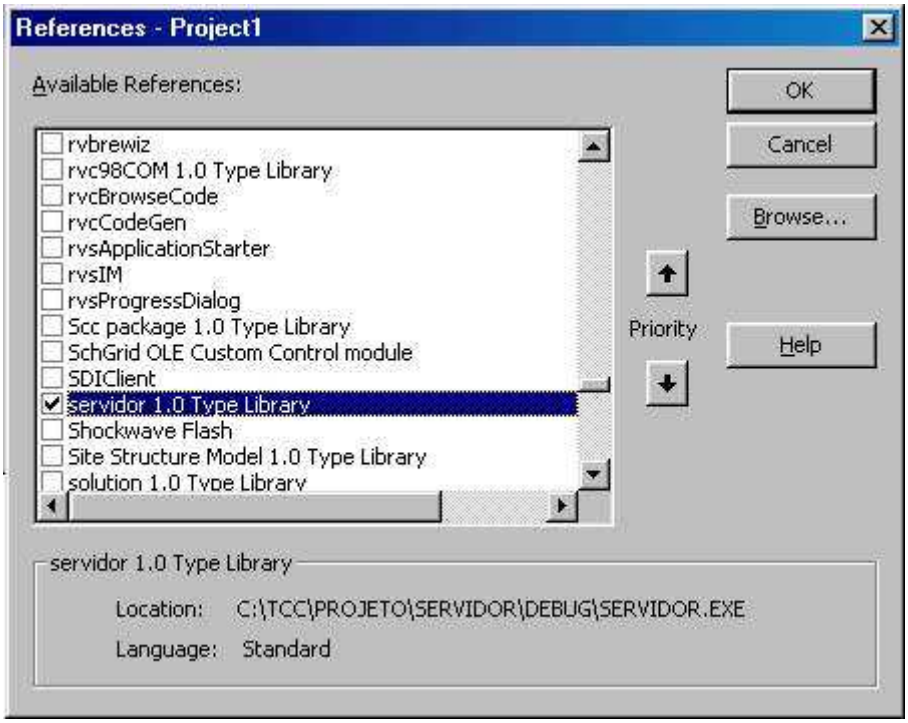
A linguagem também é muito flexível e fácil de usar uma vez que ela fornece aspectos fundamentais da programação COM. Os métodos *CoInitialize* e *CoUninitialize* são chamados automaticamente pelo Visual Basic. A palavra chave *New* substitui o método *CoCreateInstance* como se fosse um instanciador de objetos. A linguagem também se encarrega automaticamente de chamar o método *Release* de uma referência de um objeto que irá sair do escopo. Porém também é possível fazer a chamada do método *Release* explicitamente atribuindo a palavra *Nothing* para uma referência de um objeto.

Apesar de uma classe do tipo COM suportar múltiplas interfaces, o Visual Basic tenta simplificar o assunto assumindo que o programador deseja acessar uma interface padrão (*default*) e então, chama automaticamente o método *QueryInterface* para requisitar aquela interface. Esta simplificação faz sentido, uma vez que a maioria das classes tem uma interface primária pela qual a maioria das operações são executadas. Assumindo-se que o programador queira acessar a interface padrão, o Visual Basic também esconde o nome desta interface. Caso se deseje acessar, por exemplo, uma interface padrão chamada de *IMyDefault* implementada por uma classe COM chamada de *MyClass*, o nome *MyClass* será usado como se fosse um apelido para a interface *IMyDefault*.

A palavra *Implements* permite que uma classe em Visual Basic possa implementar interfaces COM descritas em um modelo de biblioteca (*type library*), isto é, a palavra *Implements* pode ser usada para implementar interfaces adicionais em uma classe.

Assim como no Visual J++, é possível importar componentes COM desde que este esteja disponível em uma *type library* para Visual Basic 6.0. Estas *type libraries* podem ser visualizadas por meio das referências disponíveis (*Available References* - Figura 10), e também podem ser feitas importações dos componentes desejados para o projeto por meio deste recurso.

Figura 10: Importando objetos COM em Visual Basic usando *Available References*



Depois de inserido no projeto o componente COM precisa apenas ser instanciado com o método *New*, como descrito anteriormente, para que se possa chamar os métodos disponíveis em sua interface.

Eddon (1998) diz que no Visual Basic, se um programador tiver a necessidade de acessar outras interfaces além da padrão (*default*) suportada por um objeto, uma chamada do tipo *QueryInterface* pode ser executada usando-se a instrução *Set*. Se for necessário, a instrução *Set* irá chamar o método *IUnknown::QueryInterface* para transformar o valor do ponteiro da interface para o tipo do valor da referência. Por exemplo, na parte do código apresentado no Quadro 7 tem-se duas referências *MyRef1* e *MyRef2*, *MyRef1* é declarada como um ponteiro de interface do tipo *IUnknown* e *MyRef2* é um ponteiro para a interface *IMyInterface*. Para *MyRef2* ser transformado para o tipo de *MyRef1*, é feita uma chamada *QueryInterface*.

Quadro 7: Execução do método *QueryInterface*

```
Dim MyRef1 As IUnknown
Set MyRef1 = New MyObject

Dim MyRef2 As IMyInterface
Set MyRef2 = MyRef1 //(QueryInterface Executada)
```

O código escrito em Visual Basic no Quadro 8, usa um modelo de biblioteca (t*ype library*) pré-definido acompanhado do código em C++ em comentário o qual explica o que o código faz nos termos do COM.

Quadro 8: Comparativo entre Visual Basic e C++ para funções básicas de COM

```
Private Sub Command1_Click()
‘IDefaultInterface* myRef1;
‘CoCreateInstance(CLSID_TheClass,NULL,CLSCTX_INPROC_SERVER,
‘IDD_IDefaultInterface,(void**)&myRef1);
Dim myRef1 As New Component.TheClass
// instanciando objeto TheClass

‘ISecondaryInterface* myRef2;
Dim myRef2 As Component.ISecondaryInterface
// Declarando uma variável do tipo ISecondaryInterface

‘myRef1->ThingOne();
myRef1.ThingOne
// chamada da função ThingOne()

‘myRef1->QueryInterface(IID_ISecondaryInterface,
(void**)&myRef2);
Set myRef2 = myRef1
// definindo myRef2 como mesmo tipo de myRef1

‘myRef2->ThingTwo();
myRef2.ThingTwo
// chamada da função ThingTwo()

‘myRef1->Release();
Set myRef1 = Nothing
// terminando utitilização do objeto myRef1

‘myRef2->Release(); é chamada automaticamente pelo Visual
Basic
//terminando utitilização do objeto myRef2 (automaticamente)
End Sub
```

2.8.4 DCOM EM DELPHI

Segundo Cantú (2002), Delphi é totalmente compatível com o COM. Um dos fundamentos da tecnologia COM diz que todo objeto COM deve implementar a interface *IUnknown*, também chamada de *IInterface* no Delphi 6, para utilização não-COM de interfaces, isto porque a classe base de interface até o Delphi 5 era *IUnknown*, porém para o Delphi 6 ela apresenta este novo nome de *IInterface*. Isto ocorre pelo fato de que este recurso da linguagem é distinto do COM da Microsoft, pois entende-se que os tipos de interface que descrevem detalhes relacionados ao COM e os serviços de sistema operacional relacionados

devem herdar de *IUnknown* e os tipos de interface que descrevem itens que não exigem necessariamente COM, como por exemplo, as interfaces usadas para estrutura interna de aplicativos, devem herdar de *IInterface*.

O Delphi fornece algumas classes diferentes com implementações prontas para uso de *IUnknown*/*IInterface*, incluindo *IInterfacedObject* e *TComObject*. A primeira pode ser usada para se ter um objeto interno não relacionado ao COM, enquanto a segunda é usada para criar objetos que podem ser exportados por servidores.

Os métodos *_AddRef* , *_Release* e *QueryInterface* da interface *IUnknown* não precisam ser implementados, pois podem ser herdados de uma das classes Delphi que já os suportam. A classe mais importante é *TComObject*, definida na unidade *ComObj*. A classe *TComObject* implementa a interface *IUnknown* usando os métodos *ObjAddRef*, *ObjQueryInterface* e *ObjRelease*.

O método *QueryInterface* que no Delphi é implementado por meio do método *GetInterface* da classe *TObject* é apresentado no Quadro 9.

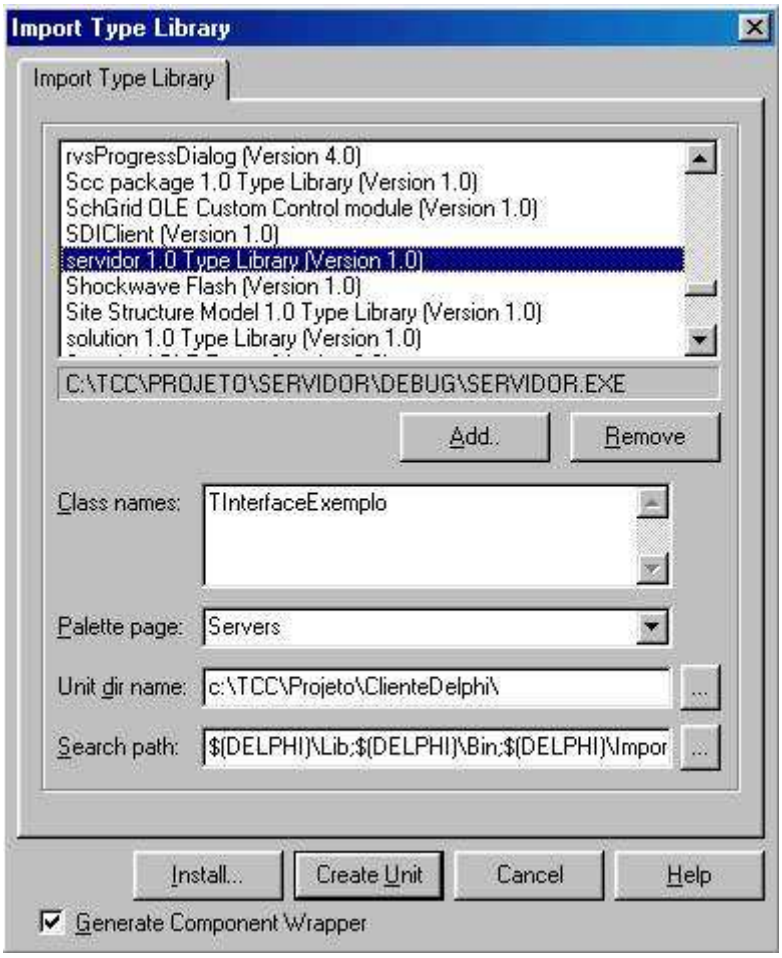
Quadro 9: Método *ObjQueryInterface* da classe *TComObject*

```
function TComObject.ObjQueryInterface(const IID: TGUID, out Obj):HResult;
begin
  if GetInterface(IID,Obj) then
    Result:= S_OK
  else
    Result:= E_NOINTERFACE;
End;
```

Como já foi mencionado anteriormente o método *QueryInterface* é usado para verificação de tipos e retorna um ponteiro para o objeto no seu parâmetro de saída de referência (out) caso o objeto seja do tipo requisitado. Isto por que, deve-se lembrar que um objeto COM pode implementar várias interfaces, como acontece com o evento *TComObject*. Então, ao utilizar o método *QueryInterface* é possível solicitar uma das possíveis interfaces do objeto usando o parâmetro *TGUID* que é um ID que identifica qualquer classe de servidor COM e qualquer interface no sistema.

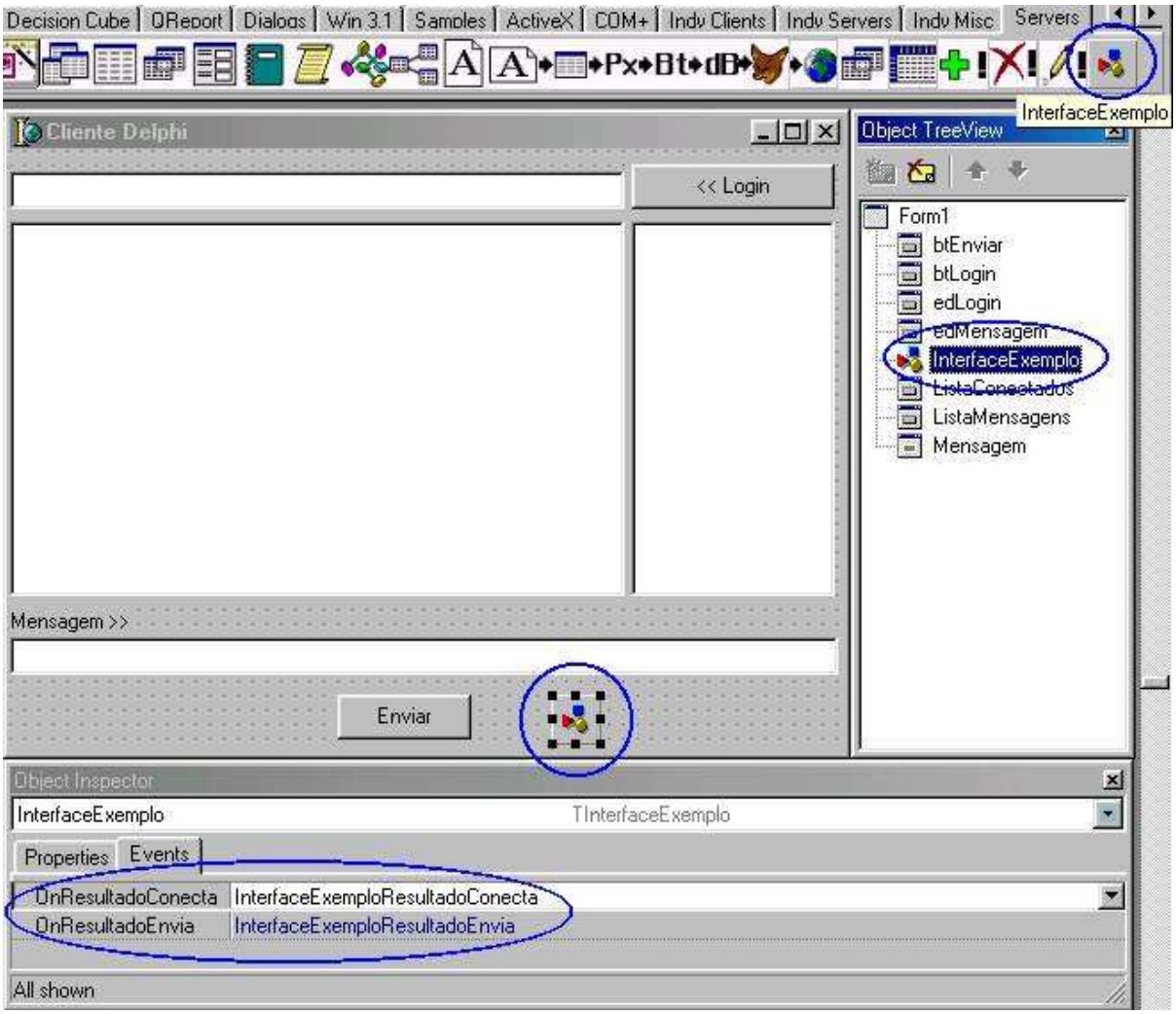
No Delphi 6 os componentes COM também podem ser importados através de *type libraries*. Para isso é necessário utilizar o recurso "*Import Type Library*" como demonstrado na Figura 11.

Figura 11: Importando objetos COM em Delphi usando *Import Type Library*



É importante destacar que ao importar um componente COM em Delphi através do recurso *Import Type Library* é gerado automaticamente um ícone com propriedades do tipo do componente importado em uma pasta escolhida pelo usuário (no caso a pasta *Servers* em *Palette page* indicada na Figura 11). Este ícone deve ser incorporado ao *Form* do projeto como apresentado na Figura 12. Nos eventos relacionados ao ícone estão os métodos de retorno das mensagens do servidor, para implementação do *Callback*.

Figura 12: Adicionando o ícone do componente COM ao projeto



Após a importação do componente COM é gerado automaticamente um arquivo *.pas* com o mesmo nome do componente importado seguido dos caracteres "*_TLB*", por exemplo "*SERVIDOR_TLB.pas*". Este arquivo possui as definições das interfaces do componente COM e seus métodos. Também é criada automaticamente uma classe auxiliar que fornece dois métodos para criação de uma instância da interface padrão do objeto importado demonstrados no Quadro 10.

Quadro 10: Classe gerada através de importação de *type library*

```
CoInterfaceExemplo = class
    class function Create: IInterfaceExemplo;
    class  function  CreateRemote(const  MachineName:  string):
IInterfaceExemplo;
  end;
```

Ambos os métodos são do tipo da interface padrão do componente importado e em sua implementação utilizam dois outros métodos expostos no Quadro 11, definidos na unidade *ComObj* do Delphi. O método *CreateComObject* cria um objeto a partir da identificação da classe da interface padrão do objeto (*TGUID*) . Já o método *CreateRemoteComObject* tem o mesmo principio porém é possível indicar-se em que computador o objeto será criado.

Quadro 11: Métodos contidos na unidade *ComObj* usados para instanciar objetos COM

```
class function CoInterfaceExemplo.Create: IInterfaceExemplo;
begin
  Result := CreateComObject(CLASS_InterfaceExemplo) as IInterfaceExemplo;
end;

class function CoInterfaceExemplo.CreateRemote(const MachineName:
string): IInterfaceExemplo;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_InterfaceExemplo) as
IInterfaceExemplo;
end;
```

Sendo assim, a partir da classe gerada pela importação do componente COM é possível criar um objeto desta classe e utilizar os métodos contidos na interface padrão deste objeto.

3 DESENVOLVIMENTO

Neste capítulo serão apresentados os requisitos do sistema, as fases de seu desenvolvimento, considerando a seqüência de suas etapas e os atributos proprietários do sistema, proporcionando assim o entendimento das funcionalidades do mesmo.

Além disso, serão também relacionados e discutidos os resultados obtidos a partir das avaliações da análise e implementação do protótipo desenvolvido.

3.1 DEFINIÇÃO DOS REQUISITOS DO SISTEMA

Serão expostos nessa seção os principais requisitos do trabalho, bem como as funcionalidades propostas, objetivos, tecnologias utilizadas e características gerais.

Com a realização deste trabalho pretendeu-se fazer uma comparação da funcionalidade da tecnologia COM/DCOM em vários ambientes de desenvolvimento como Microsoft Visual C++ 6.0, Borland Delphi 6, Microsoft Visual J++ 6.0 e Microsoft Visual Basic 6.0. Para isto foi construído um protótipo de um sistema de comunicação pessoal no modelo de um *chat* que possa ser utilizado em uma rede de computadores clientes em que a comunicação destes é gerenciada através de um servidor de informações.

Os requisitos identificados para o trabalho são:

a) o sistema deve ser capaz de permitir que um usuário conecte-se a uma rede de usuários gerenciada por um servidor comum;
b) o protótipo deve ser capaz de transmitir mensagens via tecnologia COM/DCOM, de um cliente para outro escolhido como destino ou para todos os clientes conectados ao servidor;
c) o protótipo deve permitir que um cliente receba as mensagens destinadas a ele via tecnologia COM/DCOM;
d) o sistema deve atender as necessidades do problema através de uma *LAN* (*Local Area Network*).

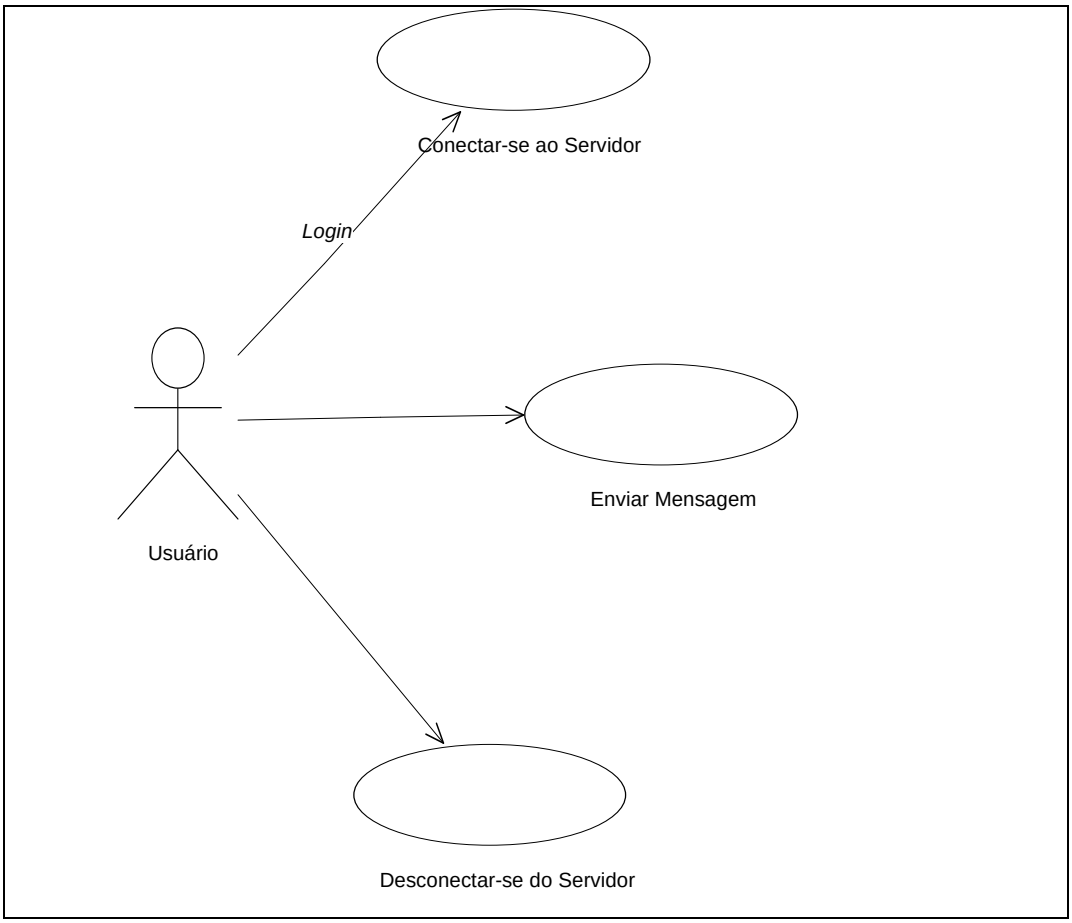
3.2 ANÁLISE E ESPECIFICAÇÃO DO SISTEMA

Para a modelagem do sistema proposto neste trabalho utilizou-se a técnica *Unified Modeling Language* (UML) com embasamento em Booch (2000), juntamente com os diagramas de casos de uso, de classes e de sequência os quais foram construídos utilizando-se a ferramenta CASE *Rational Rose* com fontes bibliográficas de Rational (2001).

3.2.1 DIAGRAMA DE CASOS DE USO

A Figura 13 demonstra o diagrama de casos de uso da aplicação proposta neste trabalho.

Figura 13: Diagrama de casos de uso



Os casos de uso identificados para o sistema são:

a) **conectar-se ao servidor**: o usuário informa seu nome e conecta-se ao servidor, caso este já tenha sido instanciado por outro cliente. Se for o primeiro cliente a conectar-se e o servidor ainda não tiver sido instanciado anteriormente, o servidor será executado automaticamente pelo primeiro cliente. Após a conexão o servidor envia para os clientes uma lista com o nome de todos os usuários conectados a ele.

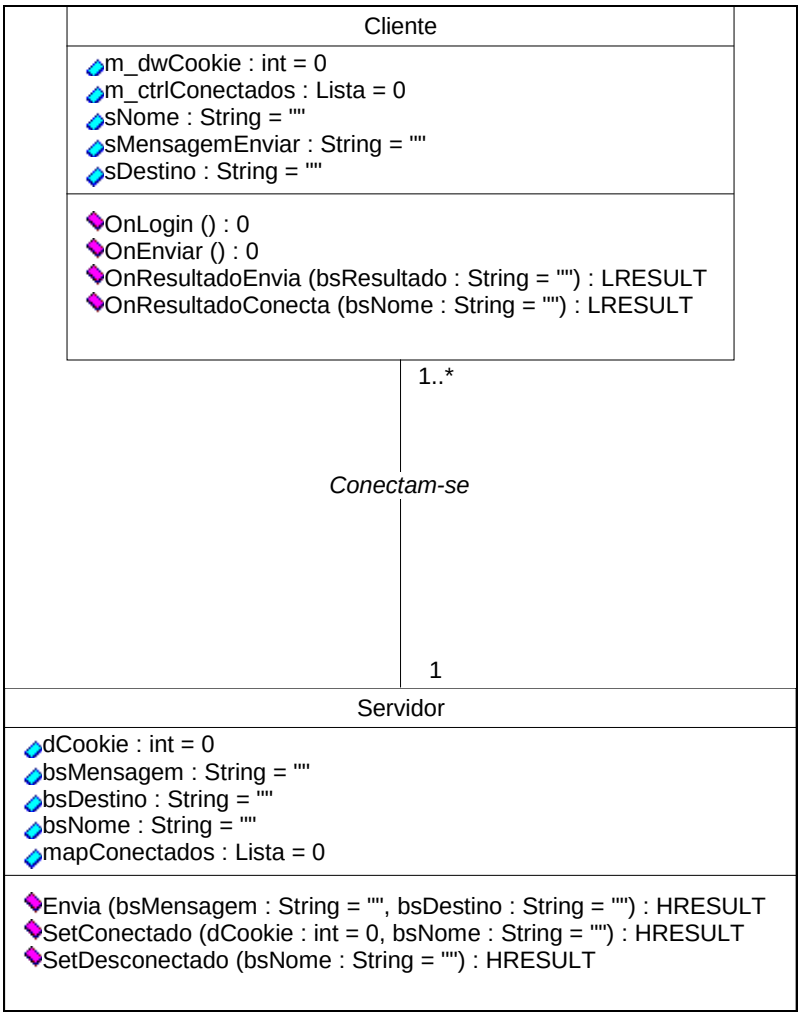
b) **enviar mensagem**: o usuário envia uma mensagem para um outro usuário. Esta mensagem irá passar primeiramente pelo servidor seguindo deste para o destinatário.

c) **desconectar-se do servidor**: quando o usuário sai do sistema o cliente comunica automaticamente ao servidor que este não faz mais parte da lista de usuários conectados.

3.2.2 DIAGRAMA DE CLASSES

Existem duas classes principais para o funcionamento do protótipo do trabalho proposto, uma é a classe *Cliente* e outra é chamada de *Servidor*. A Figura 14 demonstra o diagrama de classes da aplicação proposta neste trabalho.

Figura 14: Diagrama de classes



A classe *Cliente* foi implementada no projeto que tem interoperabilidade direta com o usuário, isto é, no projeto das interfaces. Já a classe *Servidor* foi implementada em um projeto a parte também denominado *Servidor*, o qual tem a finalidade de gerenciar a troca de informações entre os usuários.

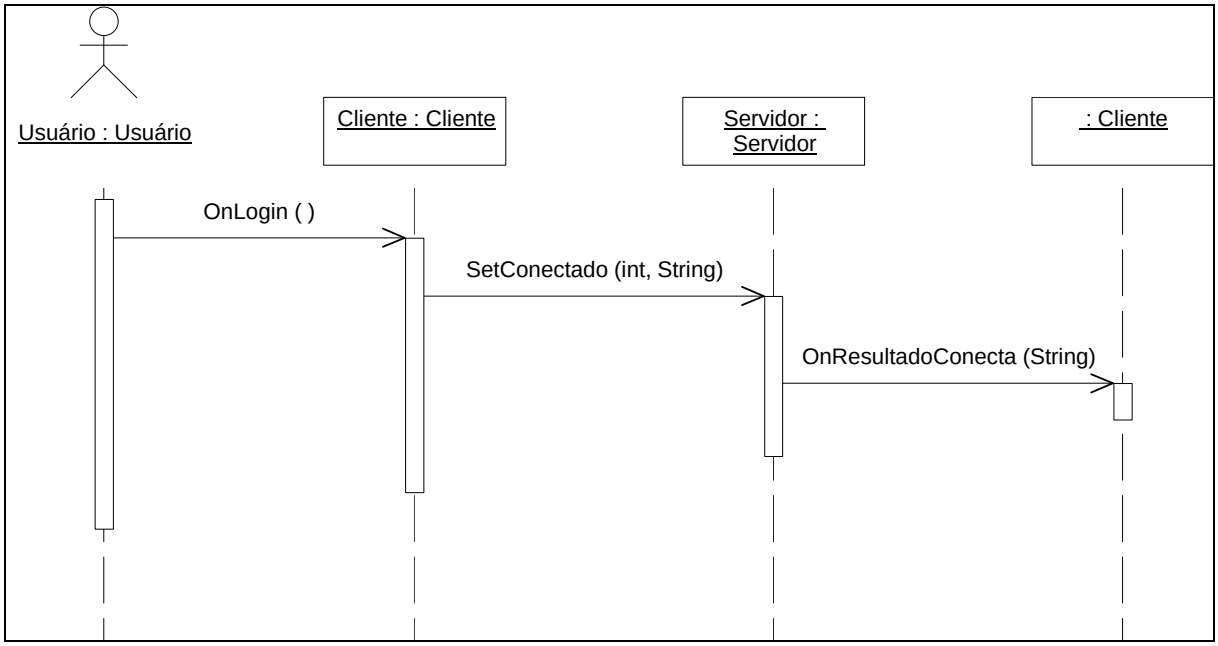
3.2.3 DIAGRAMA DE SEQUÊNCIA

Para cada caso de uso apresentado anteriormente, foi feito um diagrama de seqüência, os quais serão demonstrados a seguir.

A partir da classe *Cliente* o usuário pode logar-se no servidor através do método *OnLogin*, neste método é feita a chamada do método *SetConectado* contido no servidor. Este

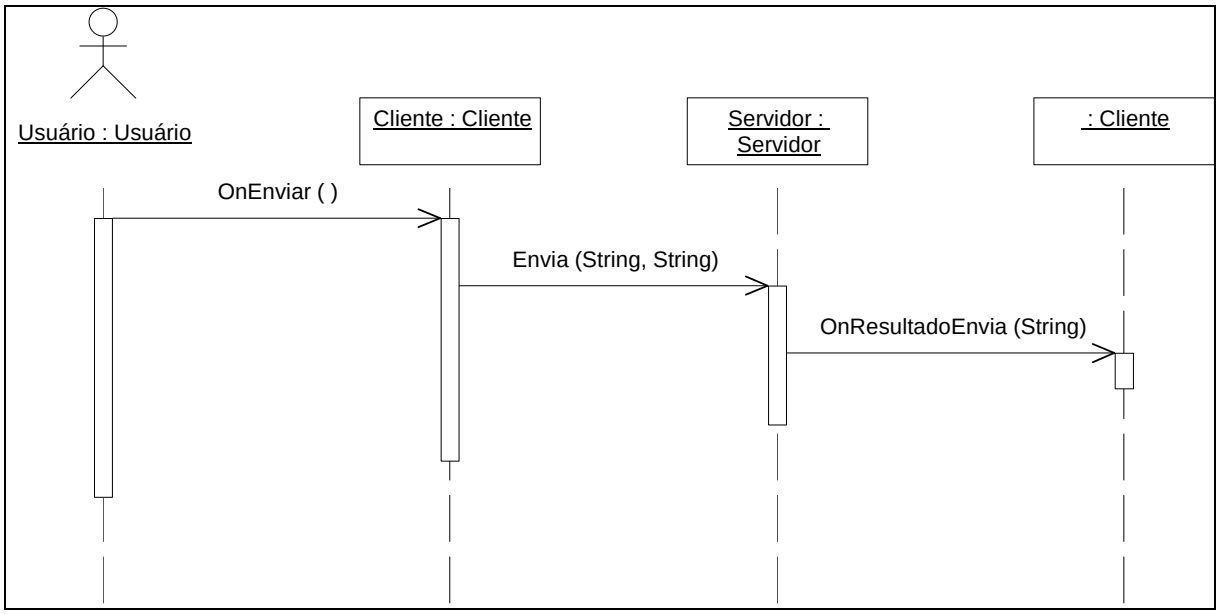
método é responsável por incluir o usuário na lista de conectados e enviar a identificação deste usuário para todos os outros clientes chamando o método *OnResultadoConecta* nos Clientes. A Figura 15 apresenta o diagrama de seqüência do caso de uso *conectar-se ao servidor*.

Figura 15: Diagrama de seqüência do caso de uso conectar-se ao servidor



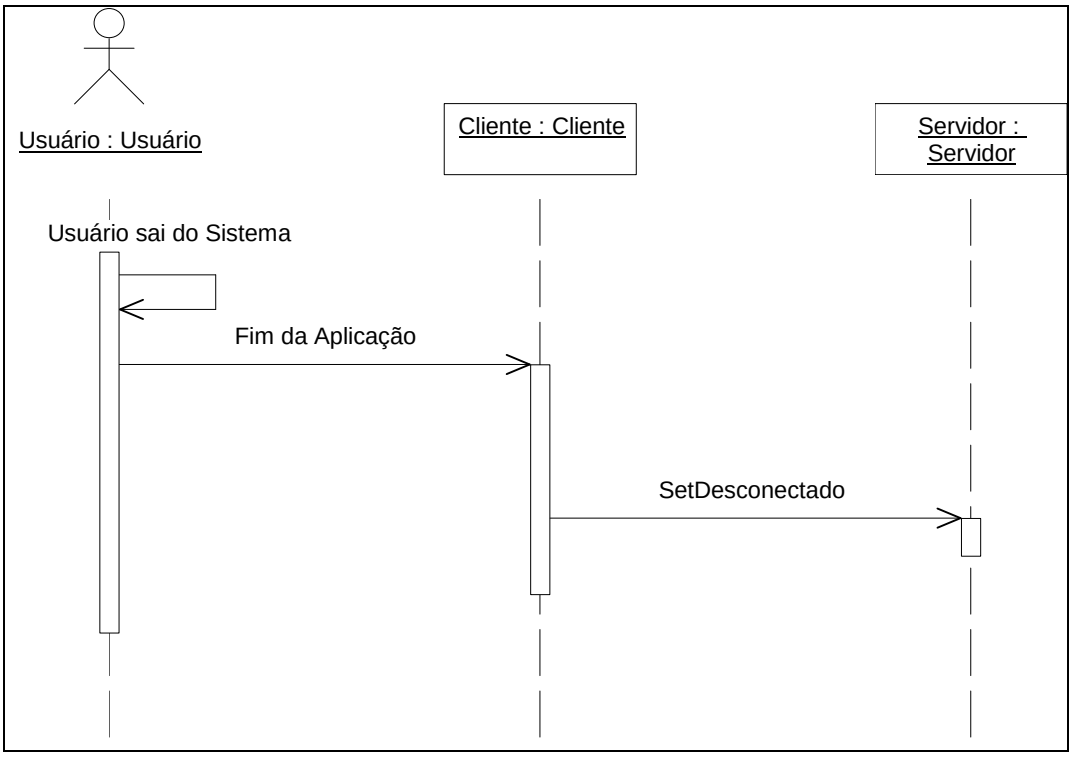
Para enviar uma mensagem de um usuário para outro(s) é usado o método *OnEnviar* no Cliente. Este método invoca um método contido no servidor denominado *Envia* que possui a descrição da mensagem e o(s) seu(s) destinatários(s). O método então transmite a mensagem para todos os usuários informados como destino, que é recebida automaticamente através do método *OnResultadoEnvia* nos Clientes que pretendem receber a mensagem. A Figura 16 apresenta o diagrama de seqüência do caso de uso *Enviar Mensagem*.

Figura 16: Diagrama de seqüência do caso de uso Enviar Mensagem



Quando o usuário sai do sistema é invocado automaticamente o método *SetDesconectado* do servidor passando-se como parâmetro o nome do usuário a ser desconectado da rede, este método comunica ao servidor que tal usuário deve ser eliminado da lista de usuários conectados. Após a realização deste procedimento é feita uma nova chamada do método *OnResultadoConecta* (demonstrado anteriormente na Figura 15) para todos os outros clientes conectados ao servidor tenham a lista de usuários conectados atualizada. A Figura 17 apresenta o diagrama de seqüência do caso de uso *Desconectar-se do Servidor*.

Figura 17: Diagrama de seqüência do caso de uso Desconectar-se do Servidor



3.2.4 IMPLEMENTAÇÃO

A implementação será apresentada dividida entre cliente e servidor, sendo que o cliente será demonstrado nas quatro linguagens onde foi implementado.

3.2.4.1 SERVIDOR

Como já comentado anteriormente o servidor foi desenvolvido em Microsoft Visual C++ 6.0 e tem por objetivo atender as necessidades dos clientes desenvolvidos em Microsoft Visual C++ 6.0, Borland Delphi 6, Microsoft Visual J++ 6.0 e Microsoft Visual Basic 6.0 os quais serão demonstrados posteriormente.

Para que os clientes instanciem o servidor correto este deve ser identificado com seu registro no *Registry* tanto na máquina onde se encontra o servidor quanto na máquina onde se encontra o cliente.

O Servidor possui duas interfaces, *IInterfaceExemplo,* demonstrada no Quadro 12 e *_IInterfaceExemploEvents*, demonstrada no Quadro 13. A primeira é do tipo *IDispatch* e

possui os métodos *Envia* que tem como objetivo enviar a mensagem do cliente para o servidor informando o destino da mensagem, o método *SetConectado* que conecta o cliente ao servidor e o método *SetDesconectado*, este último é chamado automaticamente quando o usuário sai da aplicação e serve para indicar ao servidor que o cliente desconectou-se. A interface *IInterfaceExemplo* tem a finalidade de realizar a transmissão das informações no sentido Cliente/Servidor.

Quadro 12: Interface *IInterfaceExemplo*

```
interface IInterfaceExemplo : IDispatch
{
      [id(1), helpstring("method Envia")] HRESULT Envia([in] BSTR
      bsMensagem,[in] BSTR bsDestino);
      [id(2),     helpstring("method     SetConectado")]     HRESULT
      SetConectado([in] long dCookie, [in] BSTR bsNome);
      [id(3),    helpstring("method    SetDesconectado")]    HRESULT
      SetDesconectado([in] BSTR bsNome);
};
```

A interface *_IInterfaceExemploEvents* apresentada no Quadro 13, é do tipo *dispinterface* e funciona como uma interface auxiliar com o objetivo de retornar as informações do servidor para os clientes, isto é, fazer o *Callback* no sentido Servidor/Cliente. A interface *_IInterfaceExemploEvents* possui dois métodos chamados de *OnResultadoEnvia* e *OnResultadoConecta* que são disparados através de chamadas do método *IDispatch::Invoke()* no servidor. Estas chamadas serão apresentadas no Quadro 14.

Quadro 13: Interface *_IInterfaceExemploEvents*

```
dispinterface _IInterfaceExemploEvents
{
      methods:
      [id(1),    helpstring("method    OnResultadoEnvia")]    HRESULT
      nResultadoEnvia([in] BSTR bsMsgRetorno);
      [id(2),   helpstring("method   OnResultadoConecta")]   HRESULT
      nResultadoConecta([in] BSTR bsNome);
};
```

Quadro 14: Disparando os métodos de *Callback* através do método *Invoke()*

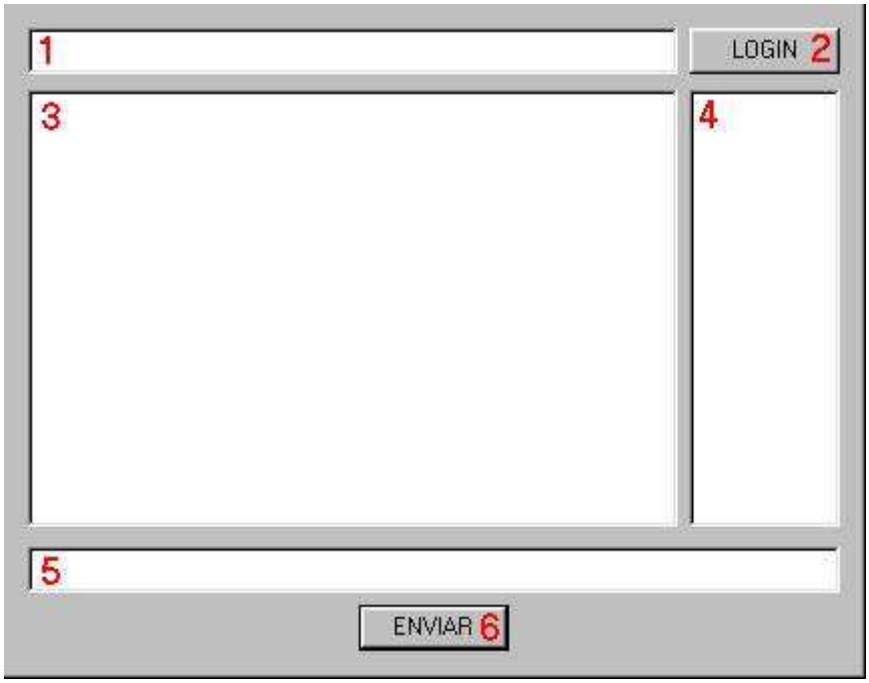
```
template <class T>
class CProxy_IInterfaceExemploEvents : public IConnectionPointImpl<T,
&DIID__IInterfaceExemploEvents, CComDynamicUnkArray>
{
  //Warning this class may be recreated by the wizard.
  public:
  HRESULT Fire_OnResultadoEnvia(BSTR bsMsgRetorno, BSTR bsDestino)
  {    ...
       int nConnections = m_vec.GetSize();
       for (nConnectionIndex = 0; nConnectionIndex < nConnections;
       nConnectionIndex++)
       {
              …
              IDispatch* pDispatch = reinterpret_cast<IDispatch*>(sp.p);
              if (pDispatch != NULL)
              {
                     VariantClear(&varResult);
                     pvars[0] = bsMsgRetorno;
                     DISPPARAMS disp = { pvars, NULL, 1, 0};
                     pDispatch->Invoke(0x1 ,IID_NULL, LOCALE_USER_DEFAULT,
                     DISPATCH_METHOD, &disp, &varResult, NULL, NULL);
              }
       }
  ...
  }
  HRESULT Fire_OnResultadoConecta(BSTR bsNome)
  {    ...
       int nConnections = m_vec.GetSize();
       for (nConnectionIndex = 0; nConnectionIndex < nConnections;
       nConnectionIndex++)
       {
              ...
              IDispatch* pDispatch = reinterpret_cast<IDispatch*>(sp.p);
              if (pDispatch != NULL)
              {
                     VariantClear(&varResult);
                     pvars[0] = bsNome;
                     DISPPARAMS disp = { pvars, NULL, 1, 0 };
                     pDispatch->Invoke(0x2 ,IID_NULL, LOCALE_USER_DEFAULT,
                     DISPATCH_METHOD, &disp, &varResult, NULL, NULL);
              }
       }
  ...
  }
};
```

Nos métodos *Fire_OnResultadoEnvia* e *Fire_OnResultadoConecta* são feitas as chamadas dos métodos *OnResultadoEnvia* e *OnResultadoConecta* respectivamente contidos na interface *_IInterfaceExemploEvents*, tais métodos são disparados pelo método *Invoke()*, este recebe no primeiro parâmetro a identificação do método como na interface *_IInterfaceExemploEvents*. O método *OnResultadoEnvia* tem por objetivo receber no cliente o retorno das mensagens do servidor. O método *OnResultadoConecta* recebe no cliente a mensagem contendo todos os usuários conectados ao servidor.

3.2.4.2 CLIENTES

A construção dos clientes foi feita em diversas linguagens de programação, porém, a interface para o usuário segue o mesmo padrão de funcionamento para todos. A seguir na Figura 18 será demonstrada a estrutura básica dos clientes.

Figura 18: Estrutura básica dos clientes



Na Figura 18, o campo indicado pelo número 1 serve para fazer a identificação do usuário. O botão indicado pelo número 2 é utilizado para conectar o usuário ao servidor. A lista indicada pelo número 3 mostra as mensagens enviadas/recebidas pelos usuários. A lista número 4 tem a identificação de todos os usuários conectados no servidor. O campo indicado pelo número 5 serve para o usuário digitar a mensagem a ser enviada. O botão 6 envia a mensagem para outros usuários.

A seguir serão apresentadas as implementações dos clientes desenvolvidos em Microsoft Visual C++ 6.0, Borland Delphi 6, Microsoft Visual J++ 6.0 e Microsoft Visual Basic 6.0. Serão também expostas as telas dos sistemas desenvolvidos nos ambientes citados, além disso, serão demonstradas as codificações com uma breve explicação das chamadas dos principais métodos do sistema que constituem a interface *IInterfaceExemplo* contida no

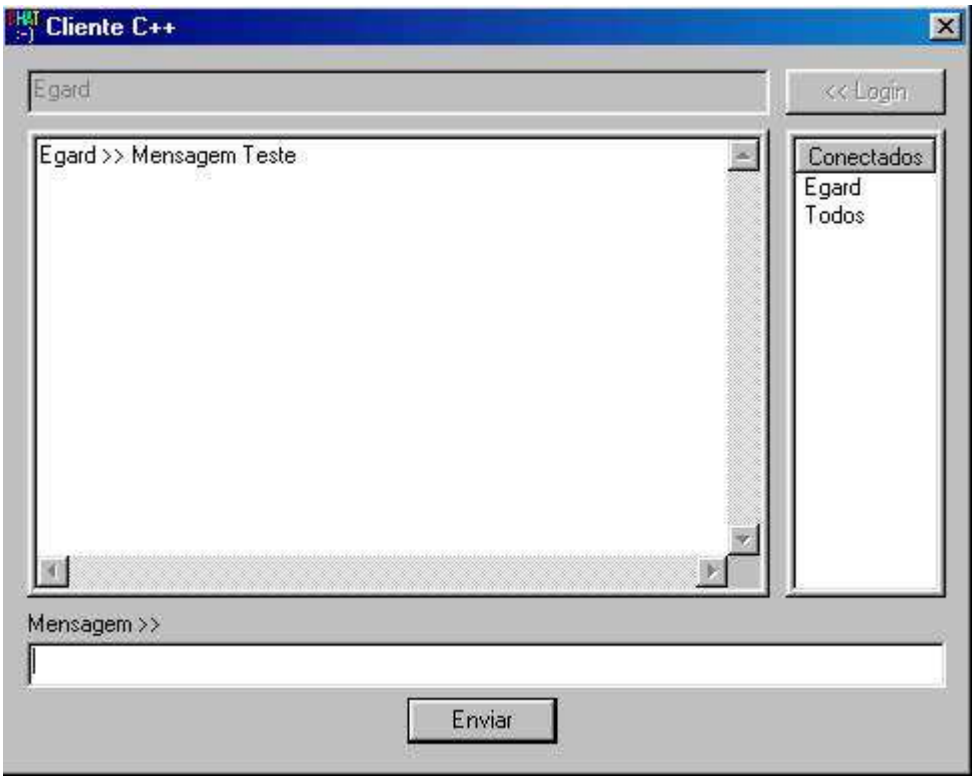
servidor juntamente com outros métodos que expõem a utilização e a comparação da tecnologia DCOM nos ambientes propostos para o trabalho desenvolvido.

Da mesma forma será exposta como é feito o recebimento pelos clientes das mensagens de retorno enviadas pelo servidos por meio de *Callback*.

3.2.4.2.1 CLIENTE C++

O cliente desenvolvido em Microsoft Visual C++ 6.0 possui a interface demonstrada na Figura 19.

Figura 19: Interface do Cliente C++



A implementação do botão *Login* construído em C++ é demonstrada no Quadro 15.

Quadro 15: Funcionamento do botão *Login* no Cliente C++

```
// Método invocado quando o botão Login é pressionado.
void CClienteDlg::OnLogin()
{
      ...
      // Chamada explicita do método CoCreateInstance, que instancia
      // o componente servidor e retorna no último parâmetro um
      // ponteiro para a interface requisitada, no caso a interface
      // IInterfaceExemplo.

      HRESULT  hr  =  CoCreateInstance(CLSID_InterfaceExemplo,  NULL,
      CLSCTX_ALL, __uuidof(IInterfaceExemplo),
      reinterpret_cast<LPVOID*>(&m_pIExemplo));

      if (SUCCEEDED(hr))
      {
      //    AfxMessageBox(_T("Interface IInterfaceExemplo criada"));
      }
      ...

      // AtlAdvise é um método COM que cria uma conexão entre cliente
      // e servidor e retorna para o cliente através da variável
      // dwCookie uma identificação única da conexão deste cliente com
      // o servidor.

      hr     =     AtlAdvise(m_pIExemplo,     pICallback,     __uuidof(
      _IInterfaceExemploEvents), &m_dwCookie);
      if (SUCCEEDED(hr))
      {
//          AfxMessageBox(_T("Callback está funcionando"));
      }
      ...

      BSTR bsNome;
      CString sNome;
      GetDlgItem(IDC_ED_LOGIN)->GetWindowText(sNome);
      bsNome = sNome.AllocSysString();

      // chamada do método SetConectado da interface IInterfaceExemplo
      // implementada no servidor.
      m_pIExemplo->SetConectado(m_dwCookie,bsNome);
}
```

O botão Enviar chama o método *Envia* do servidor como demonstrado no Quadro 16.

Quadro 16: Funcionamento do botão *Enviar* no Cliente C++

```
// Método invocado quando o botão Enviar é pressionado.
void CClienteDlg::OnEnviar()
{
      ...
      BSTR    bsMensagem;
      BSTR    bsDestino;
      ...
      // chamada do método Envia da interface IInterfaceExemplo
      // implementada no servidor.
      HRESULT hr = m_pIExemplo->Envia(bsMensagem,bsDestino);
      ...
}
```

Ao sair do sistema é feita a desconexão do cliente desenvolvido em Visual C++ como demonstrado no Quadro 17.

Quadro 17: Desconexão do Cliente C++

```
// Método invocado quando o usuário sai da aplicação
void CClienteDlg::OnDestroy()
{
      if (m_dwCookie != 0)
      {
            ...
            // chamada do método SetDesconectado da interface
            // IInterfaceExemplo implementada no servidor.
            m_pIExemplo->SetDesconectado(bsNome);
            SysFreeString(bsNome);

            // AtlUnadvise é um método COM que finaliza a conexão entre
            // cliente e servidor.
            HRESULT hr = AtlUnadvise(m_pIExemplo,
                        __uuidof(_IInterfaceExemploEvents), m_dwCookie);
            if (SUCCEEDED(hr))
            {
//                AfxMessageBox(_T("Callback desconectado"));
            }
      }
      // libera-se a interface IInterfaceExemplo
      if (m_pIExemplo)
            RELEASE(m_pIExemplo);
      CDialog::OnDestroy();
}
```

No Quadro 18 e Quadro 19 são apresentados respectivamente os métodos *OnResultadoConecta* e *OnResultadoEnvia* que recebem as mensagens do servidor por meio de *callback*.

Quadro 18: Método *OnResultadoConecta* em Visual C++

```
// Este método recebe na variável bsNome todos os nomes dos
// usuários conectados ao servidor.
LRESULT CClienteDlg::OnResultadoConecta(BSTR bsNome,LPARAM lParam)
{
  CString sNome;
  CString sNomeUnico;
  sNome = bsNome;
  m_ctrlConectados.DeleteAllItems();

  while (sNome.GetLength() > 0)
  {
    sNomeUnico = sNome.Left(sNome.Find('\n'));
    sNome.Delete(0,sNomeUnico.GetLength()+1);
    m_ctrlConectados.InsertItem(m_ctrlConectados.GetItemCount(),
    sNomeUnico);
  }
  m_ctrlConectados.InsertItem(m_ctrlConectados.GetItemCount(),
  "Todos");
  return 0;
}
```

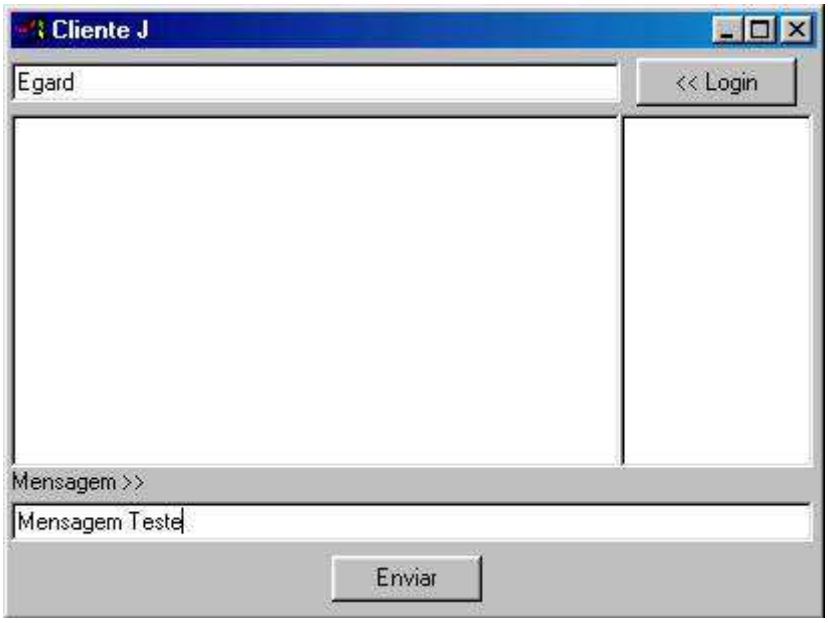
Quadro 19: Método *OnResultadoEnvia* em Visual C++

```
// Este método recebe na variável bsResultado todas as mensagens
// enviadas ao cliente.
LRESULT  CClienteDlg::OnResultadoEnvia(BSTR  bsResultado,  LPARAM
lParam)
{
      CString sMsg;
      sMsg = bsResultado;
      m_sLista += sMsg;
      m_sLista += 13;
      m_sLista += 10;
      UpdateData(FALSE);
      PostMessage(WM_ATUALIZA_LISTA);
      return 0;
}
```

3.2.4.2.2 CLIENTE JAVA

O cliente desenvolvido em Microsoft Visual J++ 6.0 possui a interface demonstrada na Figura 20.

Figura 20: Interface do Cliente Java



A funcionalidade do botão *Login* construído em Java será demonstrado do Quadro 20.

Quadro 20: Funcionamento do botão *Login* no Cliente Java

```
// Método invocado quando o botão Login é pressionado.
private void btLogin_click(Object source, Event e)
{
  // Instanciação do componente servidor e criação de um
  // objeto do tipo da interface IInterfaceExemplo.
      pServidor = new InterfaceExemplo();
      ...

  String sLogin;
  sLogin = edLogin.getText();

  // chamada do método SetConectado da interface IInterfaceExemplo
  // implementada no servidor.
  pServidor.SetConectado(dwConectado,sLogin);
}
```

O botão Enviar chama o método *Envia* do servidor como demonstrado no Quadro 21.

Quadro 21: Funcionamento do botão *Enviar* no Cliente Java

```
// Método invocado quando o botão Enviar é pressionado.
private void btEnviar_click(Object source, Event e)
{
      String sMensagem;
      sMensagem   =   edLogin.getText()   +   "   >>   "   +
      edMensagem.getText();

      // chamada do método Envia da interface IInterfaceExemplo
      // implementada no servidor.
      pServidor.Envia(sMensagem,"Todos");
      edMensagem.clear();
      edMensagem.focus();
}
```

Ao sair do sistema é feita a desconexão do cliente desenvolvido em Visual J++ como demosntrado no Quadro 22.

Quadro 22: Desconexão do cliente Java

```
// Método invocado quando o usuário sai da aplicação
private void OnExit(Object source, Event e)
{
      // chamada do método SetDesconectado da interface
      // IInterfaceExemplo implementada no servidor.
      pServidor.SetDesconectado(edLogin.getText());
}
```

Devido a problemas encontrados na utilização de *Callback* em Java não foi possível realizar a implementação dos métodos de retorno das mensagens em tal ambiente. Mais explanações sobre o assunto serão tratadas no tópico 3.2.5 deste mesmo capítulo.

3.2.4.2.3 CLIENTE VISUAL BASIC

O cliente desenvolvido em Microsoft Visual Basic 6.0 possui a interface demonstrada na Figura 21.

Figura 21: Interface do Cliente Visual Basic

A implementação do botão *Login* desenvolvida em Visual Basic é demonstrada no Quadro 23.

Quadro 23: Funcionamento do botão *Login* no Cliente Visual Basic

```
// Método responsável por instanciar o componente
  // servidor e criar um objeto do tipo da interface
  //IInterfaceExemplo.
Sub CriarObjeto()
  Set objServ = New InterfaceExemplo
End Sub

// Método invocado quando o botão Login é pressionado.
Private Sub btLogin_Click()
  edLogin.Enabled = False
  edMensagem.Enabled = True
  btLogin.Enabled = False
  btEnviar.Enabled = True

  // chamada do método CriarObjeto()
  Call CriarObjeto
  Dim sNome As String
  sNome = edLogin.Text
  Dim dwConectado As Long
  dwConectado = 0

  // chamada do método SetConectado da interface
  // IInterfaceExemplo implementada no servidor.
  Call objServ.SetConectado(dwConectado, sNome)
  edMensagem.SetFocus
End Sub
```

O botão Enviar chama o método *Envia* do servidor como demonstrado no Quadro 24.

Quadro 24: Funcionamento do botão *Enviar* no Cliente Visual Basic

```
// Método invocado quando o botão Enviar é pressionado.
Private Sub btEnviar_Click()
  Dim sMensagem As String
  Dim sMensagemEnviar As String
  Dim sDestino As String
  sMensagem = edMensagem.Text
  sMensagemEnviar = edLogin.Text + " >> " +
  sMensagem

  sDestino = ListaConectados.Text
  If sDestino = "" Then
      sDestino = "Todos"
  End If

  // chamada do método Envia da interface IInterfaceExemplo
  // implementada no servidor.
  Call objServ.Envia(sMensagemEnviar, sDestino)
  edMensagem.Text = ""
End Sub
```

Ao sair do sistema é feita a desconexão do cliente desenvolvido em Visual Basic como demonstrado no Quadro 25.

Quadro 25: Desconexão do cliente Visual Basic

```
// Método invocado quando o usuário sai da aplicação
Private Sub Form_Unload(Cancel As Integer)
  Dim sNome As String
  sNome = edLogin.Text

  // chamada do método SetDesconectado da interface
  // IInterfaceExemplo implementada no servidor.
  objServ.SetDesconectado (sNome)
End Sub
```

No Quadro 26 e Quadro 27 são apresentados respectivamente os métodos *OnResultadoConecta* e *OnResultadoEnvia* que recebem as mensagens do servidor por meio de *callback*.

Quadro 26: Método *OnResultadoConecta* em Visual Basic

```
// Este método recebe na variável bsNome todos os nomes dos
// usuários conectados ao servidor.
Private Sub objServ_OnResultadoConecta(ByVal bsNome As String)
Dim TmpBlock As String
Dim sNomes As String
sNomes = bsNome

ListaConectados.Clear

Do While InStr(sNomes, Chr(10))
    TmpBlock = Left(sNomes, InStr(sNomes, Chr(10)) - 1)
    sNomes = Right(sNomes, (Len(sNomes) - Len(TmpBlock) - 1))
    ListaConectados.AddItem (TmpBlock)
  Loop
  ListaConectados.AddItem ("Todos")
End Sub
```

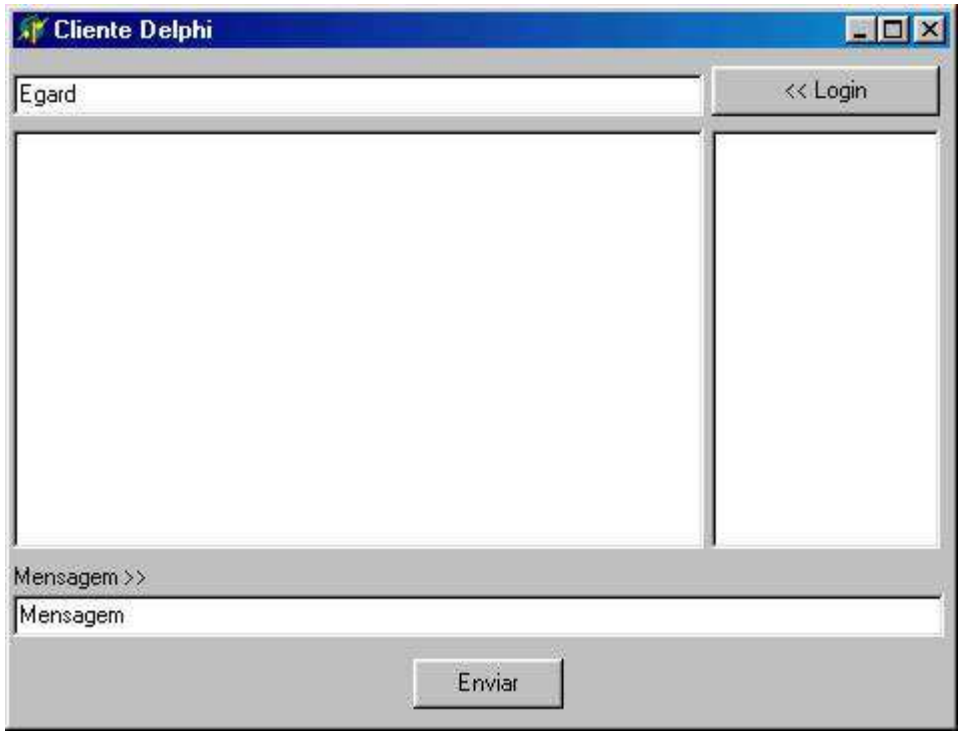
Quadro 27: Método *OnResultadoEnvia* em Visual Basic

```
// Este método recebe na variável bsMsgRetorno todas as mensagens
// enviadas ao cliente.
Private Sub objServ_OnResultadoEnvia(ByVal bsMsgRetorno As String)
    listaMsg.AddItem (bsMsgRetorno)
End Sub
```

3.2.4.2.4 CLIENTE DELPHI

O cliente desenvolvido em Delphi 6 possui a interface demonstrada na Figura 22.

Figura 22: Interface do Cliente Delphi



A implementação do botão *Login* desenvolvida em Delphi é demonstrada no Quadro 28.

Quadro 28: Funcionamento do botão *Login* no Cliente Delphi

```
// Método invocado quando o botão Login é pressionado.
procedure TForm1.btLoginClick(Sender: TObject);
begin
  dwConectado := 0;
  sNome := edLogin.Text;

  // Instanciação do componente servidor e criação de um
  // objeto do tipo da interface IInterfaceExemplo.
  InterfaceExemplo.Create(nil);

  // chamada do método SetConectado da interface
  // IInterfaceExemplo implementada no servidor.
  InterfaceExemplo.SetConectado(dwConectado,sNome);
end;
```

O botão Enviar chama o método *Envia* do servidor como demonstrado no Quadro 29.

Quadro 29: Funcionamento do botão *Enviar* no Cliente Delphi

```
// Método invocado quando o botão Enviar é pressionado.
procedure TForm1.btEnviarClick(Sender: TObject);
begin
  sMensagem := edLogin.Text + ' >> ' + edMensagem.Text;
  sDestino:= 'Todos';
  if ListaConectados.ItemIndex > -1 then
  begin
    sDestino :=ListaConectados.Items.Strings[ListaConectados.ItemIndex];
  end;

  // chamada do método Envia da interface IInterfaceExemplo
  // implementada no servidor.
  InterfaceExemplo.Envia(sMensagem, sDestino);
end;
```

Ao sair do sistema é feita a desconexão do cliente desenvolvido em Delphi como demonstrado no Quadro 30.

Quadro 30: Desconexão do cliente Delphi

```
// Método invocado quando o usuário sai da aplicação
procedure TForm1.OnExit(Sender: TObject; var Action: TCloseAction);
begin
  // chamada do método SetDesconectado da interface
  InterfaceExemplo.SetDesconectado(edLogin.Text);
end;
```

No Quadro 31 e Quadro 32 são apresentados respectivamente os métodos *OnResultadoConecta* e *OnResultadoEnvia* que recebem as mensagens do servidor por meio de *callback*.

Quadro 31: Método *OnResultadoConecta* em Delphi

```
// Este método recebe na variável bsNome todos os nomes dos
// usuários conectados ao servidor.
procedure TForm1.InterfaceExemploResultadoConecta(Sender: TObject;
  var bsNome: OleVariant);
begin
  ListaConectados.Clear;
  sNomeTodosUsuarios := bsNome;
    for i:=1 to Length(sNomeTodosUsuarios) do
    begin
        if (sNomeTodosUsuarios[i] <> #$A) then
        begin
          sNomeUsuario :=  sNomeUsuario + sNomeTodosUsuarios[i];
        end
        else
          begin
            ListaConectados.Items.Add(sNomeUsuario);
            sNomeUsuario := '';
          end;
    end;
    ListaConectados.Items.add('Todos');
end;
```

Quadro 32: Método *OnResultadoEnvia* em Delphi

```
// Este método recebe na variável bsMsgRetorno todas as mensagens
// enviadas ao cliente.
procedure TForm1.InterfaceExemploResultadoEnvia(Sender: TObject;
var bsMsgRetorno: OleVariant);
begin
    ListaMensagens.Items.Add(bsMsgRetorno);
end;
```

3.2.5 RESULTADOS / DISCUSSÃO

Foi possível realizar a implementação da tecnologia DCOM em todas as linguagens propostas de forma funcional, porém ficou claro para o acadêmico que existe uma dificuldade maior de implementar-se DCOM utilizando a linguagem C++, pois nela o desenvolvedor tem a necessidade de programar em mais baixo nível, isto é, os métodos COM quase sempre devem ser utilizados explicitamente pelo desenvolvedor. Já nas outras linguagens os métodos muitas vezes já são encapsulados em métodos próprios da linguagem, facilitando assim a implementação.

Percebeu-se também que a implementação da tecnologia DCOM nos ambientes Visual Basic 6.0 e Visual J++ 6.0 são muito parecidas levando-se em consideração a criação do componente servidor e manipulação de seus dados, sendo que estes mecanismos são utilizados de uma forma bem mais superficial, isto é, limitada, em relação ao ambiente Visual C++. Pode-se exemplificar esta análise com um problema real ocorrido na implementação do cliente em Visual Basic. Por exemplo, quando é feita uma conexão entre o cliente e o servidor em Visual C++ através do método *AtlAdvise()* (demonstrado anteriormente no Quadro 15), o cliente recebe um retorno de um identificador único da conexão realizada. Este identificador é utilizado pelos clientes como destino para as mensagens. Em Visual Basic não foi possível utilizar um método semelhante para obter este identificador, tendo que se solucionar este problema através de implementações adicionais no servidor.

As maiores dificuldades encontradas na parte de implementação se deram quanto a utilização da técnica de *Callback* nos ambientes propostos, principalmente no Visual J++. Para todos os ambientes esta técnica é pouco explorada em termos de bibliografia, o que causou alguns problemas até se chegar a sua implementação. No caso do Visual J++ não foi possível concretizar a implementação dos métodos de retorno das mensagens Este problema não ocorreu de forma tão agravante em Visual C++, pelo fato do acadêmico ter alguma experiência com a linguagem e ambiente utilizado. Em Visual Basic a técnica de *Callback* causou poucas dificuldades na implementação, isso porque o ambiente facilita a utilização da técnica criando quase que automaticamente os métodos de recebimento das mensagens provindas do servidor. No ambiente Delphi a dificuldade foi em compreender o seu funcionamento. Uma vez entendido o componente gerado, sua implementação foi realizada com sucesso.

4 CONCLUSÕES E SUGESTÕES

Este capítulo apresenta as conclusões e sugestões referentes ao trabalho desenvolvido.

4.1 CONCLUSÕES

O objetivo do presente trabalho foi atingido, o qual consistia no desenvolvimento de um sistema de comunicação pessoal utilizando-se a tecnologia DCOM para transmissão das informações entre os usuários, sendo que o protótipo foi desenvolvido em quatro ambientes de programação com o intuito principal da análise da operacionalidade e funcionalidade da tecnologia proposta em tais ambientes.

Através do trabalho desenvolvido foi possível integrar todos os ambientes por meio da tecnologia DCOM. Com o estudo da arquitetura DCOM aliada com os ambientes apresentados neste trabalho é possível dizer que tal arquitetura tem realmente características de uma tecnologia que pode ser utilizada quase que universalmente entre os ambientes atuais que seguem os fundamentos da orientação a objetos. Sendo assim, o trabalho desenvolvido serve como base científica para provar que é possível fazer a integração de várias linguagens através da tecnologia citada na plataforma *Windows*. É importante lembrar que os ambientes Visual C++, Visual J++ e Visual Basic utilizados neste trabalho, pertencem ao mesmo fabricante, facilitando a compatibilidade da tecnologia DCOM entre estes ambientes.

No caso de migração entre ambientes a tecnologia oferece grandes facilidades, uma vez que um componente COM pode ser importada facilmente para qualquer um dos ambientes estudados.

Além disso, pode-se afirmar também que o emprego da tecnologia DCOM possibilita o desenvolvimento de software com uma redução de tempo e de custo de desenvolvimento, visto que, os componentes necessários para que um sistema funcione podem ser desenvolvidos em linguagens diferentes, já que estes têm a capacidade de interagir na troca de informações, excluindo assim a necessidade de se reescrever todos os componentes em apenas uma linguagem. Isso reforça um dos princípios dos softwares desenvolvidos com base na orientação a objetos que é a reutilização de código.

Deve ser levado em consideração que o trabalho foi desenvolvido seguindo uma estrutura de servidor centralizado, então, os clientes são totalmente dependentes do correto funcionamento do servidor. Caso por algum motivo o servidor não puder atender as necessidades dos clientes o sistema ficará paralisado. Além disso, todos os testes realizados na aplicação foram feitos em redes locais.

4.2 SUGESTÕES

Como sugestões para futuros trabalhos pode-se ampliar as funcionalidades do *chat* adicionando sistemas de *log*, troca de arquivos e armazenamento de mensagens para usuários não conectados.

Também sugere-se para trabalhos futuros um estudo comparativo do desenvolvimento de componentes COM+ (que é uma extensão do COM) em diversas ferramentas de desenvolvimento, ou ainda um comparativo entre a tecnologia COM e COM+.

Uma outra sugestão pode ser o estudo e desenvolvimento de aplicações distribuídas através de componentes criados na plataforma .NET para aplicações via Internet.

Realizar um comparativo da tecnologia DCOM em relação a performance atingida em diversos ambientes de desenvolvimento.

REFERÊNCIAS BIBLIOGRÁFICAS

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML guia do usuário**. Rio de Janeiro: Campus, 2000.

CAPELETTO, Johni Jeferson. **Comunicação entre objetos distribuidos utilizando a tecnologia CORBA (Common Object Request Broker Architecture)**. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. 1999.

CANTÚ, Marco. **Dominando o Delphi 6: a Bíblia**. São Paulo, Makron Books, 2002.

CODELINES. **Codelines Type Libraries**. 2000. Endereço eletrônico: <http://www.codelines.com/typlib.htm>. Data da consulta 01/05/2002.

CODEPROJECT. **The Code Project – Home Page – Free Source Code and Tutorials**. 2002. Endereço eletrônico: <http://www.codeproject.com/>. Data da consulta 05/05/2002.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Distributed Systems Concepts and Design**. New York, Addison-Wesley & ACM Press, 1994.

DEVBRASIL. – **devBrasil para developers**. 2001. Endereço eletrônico: <http://www.devbrasil.com/artigo.aspx?url=/com/callbackbasico>. Data da consulta 04/04/2002.

EDDON, Guy; EDDON, Henry. **Inside distributed COM**. Washington: Microsoft Press, 1998.

GERAGHTY, Ronan; SEAN, Joice; MORIARTY, Tom. **COM – CORBA interoperability**. New Jersey, 1999.

GRIMES, Richard; **Professional DCOM programming**. Olton: Wrox Press, 1997.

GRIMES, Richard; STOCKTON, Alex; REILLY, George; Templeman, Julian. **Beginning ATL COM programming**. Olton: Wrox Press, 1998.

MAINETTI, Sérgio. **Objetos distribuídos**. Curitiba: Visionnaire, 1997.

MELO, José Coelho de. **O modelo cliente/servidor e suas técnicas de programação**. São Luiz, 1999. Monografia (Especialização em Ciências da Computação) – Centro Tecnológico, Universidade Federal do Maranhão – UFMA.

MICROSOFT. MSDN – **Microsoft developer network** 2001. Endereço eletrônico: http://msdn.microsoft.com. Data da consulta 05/11/2001.

RATIONAL Software Corporation. **Rational Rose v2001**: visual modeling, UML, object-oriented, component-based development with Rational, [S.l.], [2001]. Disponível em: <http://www.rational.com/products/rose/>. Acesso em: 15 set. 2001.

REDMOND III, Frank E.. **DCOM: microsoft distributed component object model**. Chicago, Indianapolis: IDG Books World Wide, 1997.

ROGERSON, Dale. **Inside COM**. Redmond: Microsoft Press, 1997.

SANTOS, Joel dos. **Fácil acesso a objetos distribuídos**. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. 1998.

SCHMIDT, Vianei. **Módulo de informação para a área hoteleira baseado em objetos distribuídos**. 2000. 133 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Educação Superior de Ciências Tecnológicas, da Terra e do Mar, Universidade do Vale do Itajaí, Itajaí.

SZYPERSKI, Clemens. **Component software beyond object-oriented programming**. New York: Addison-Wesley & ACM Press, 1998.