MAURÍCIO SEVERICH

IMPLEMENTAÇÃO DE CLIENTE/SERVIDOR PARA LINGUAGEM NÃO PREDISPOSTA

PONTA GROSSA
1999

MAURÍCIO SEVERICH

IMPLEMENTAÇÃO DE CLIENTE/SERVIDOR PARA LINGUAGEM NÃO PREDISPOSTA

Monografia apresentada como requisito indispensável à conclusão do curso de especialização em Ciência da Computação.
Universidade Estadual de Ponta Grossa.
Orientador: Prof. M.Sc. Dierone Cesar Foltran Júnior.

PONTA GROSSA
1999

Para a mulher que amo, Sheila

Agradecimentos

À Deus,

Por me acompanhar em todos os momentos.

Aos meus pais,

Por serem meus pais, pelo carinho, incentivo, apoio e dedicação, que sempre me dão.

Aos meus irmãos,

Por nossa união e pelo modo como cada um me incentiva.

Aos meus amigos,

Ademir, Henry, Luciano, Lucio, Marra, Maurinho, Rodrigo, Ronaldo, Waldo, pelo companheirismo, motivação e amizade.

À pessoa que me orientou,

Incentivando minha proposta, apontando caminhos, fazendo-me refletir, entender, analisar, e melhorar esta ou aquela questão, com dedicação, com humanidade, compreendendo meus limites: Prof. M.Sc. Dierone César Foltran Júnior, muito obrigado.

À minha esposa Sheila,

Por seu amor, carinho, dedicação e por compreender minha ausência durante a composição deste trabalho.

Sumário

Índice de figuras

1. Introdução

No mundo altamente competitivo em que vivemos, onde a velocidade com que se obtém uma informação útil pode mudar o destino de qualquer negócio, toda organização deseja possuir um sistema de informação que lhe proporcione alguma vantagem competitiva.

Os sistemas de informação corporativos provêem o recurso de informação compartilhada para a corporação inteira. Eles tratam informação e tecnologia como recursos incorporados, ou seja, ferramentas de negócios. Com o desenvolvimento da tecnologia, os sistemas de informação tornaram-se cada vez mais automatizados e permitiram um controle mais eficiente sobre os dados da organização.

Os primeiros sistemas de informação eram baseados em sistemas centralizados, onde um computador central é responsável por todo o processamento, a ele são conectados, através de linhas de comunicação, terminais[1] interativos que são distribuídos aos usuários e compartilham o tempo do processamento do computador central, como representado na Figura 1.

[1] Terminal: um terminal interativo de um sistema computacional, que é totalmente dependente do computador central ao qual esta conectado, para executar qualquer processo do usuário. Outra atribuição comumente usada é a de terminal “burro”

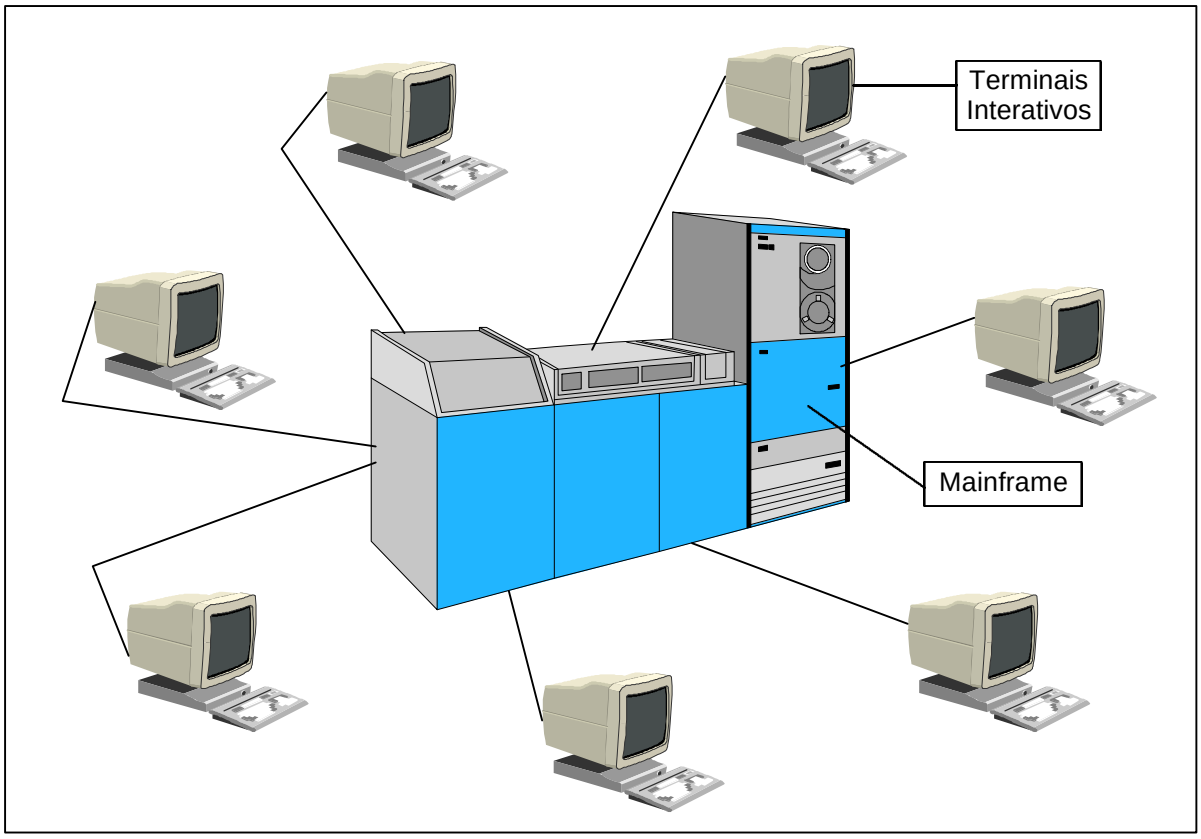

Figura 1: Sistema Centralizado

Quando os *mainframes*[2] dominavam a área de informática, a escolha do fornecedor certo era fundamental para que todos os processos relativos ao C.P.D. (Centro de Processamento de Dados) corressem da melhor forma possível [15]. Muitas empresas gastavam tempo e quantias consideráveis em desenvolvimento de sistemas de gerenciamento de dados sob encomenda, como por exemplo aplicativos desenvolvidos em COBOL[3]. A medida que os processos comerciais tornaram-se mais dinâmicos e cada vez mais exigiam competitividade, estes aplicativos legados mostraram-se pouco flexíveis/produtíveis frente as novas possibilidades da tecnologia da informação oferecidas pelos sistemas computacionais modernos [17].

[2] *Mainframe*: computador de grande porte, com grande capacidade de processamento e armazenamento, seu alto custo o limita a grandes corporações. Atualmente os *mainframes* estão, adaptados a atual situação da informática e mais competitivos, mas ainda a um alto custo, são também chamados de servidores corporativos.

[3] COBOL-*COmmon Business Oriented Language*: Linguagem comum orientada a negócios, comentada no tópico 4.1

Em meados da década de 70, o conceito de computador pessoal, representado na Figura 2, começou a surgir. Alavancado pelo rápido desenvolvimento dos circuitos integrados, alguns anos depois já estava disponível comercialmente.

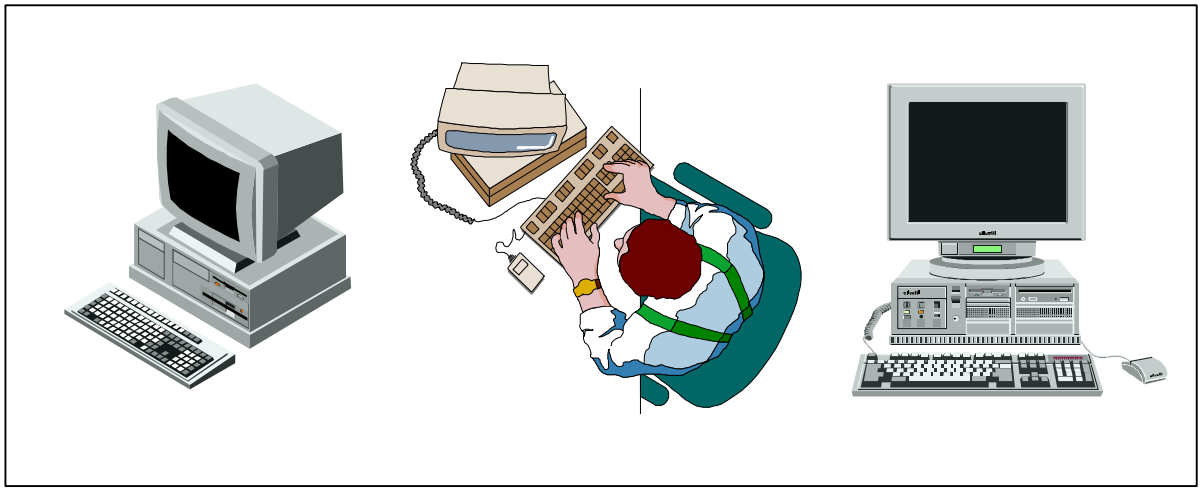

Figura 2: Computador Pessoal

Paralelamente, o conceito de aplicativo *user-friendly*[4], ou amigável ao usuário, também começou a ser empregado, para que uma pessoa pudesse trabalhar em seu computador pessoal sem a necessidade de treinamentos muito complexos. O usuário começava a ter maior liberdade para realizar suas tarefas, não dependendo tão somente do *mainframe*. [19]

Mesmo que os computadores pessoais tenham proporcionado tantos benefícios, ainda não podiam ser utilizados como base do sistema corporativo, pois trabalhavam isolados. Por exemplo, para manter a veracidade de algumas informações, em um determinado sistema administrativo, seria necessário que todos os PCs[5] tivessem acesso a uma base de dados comum, como ainda não existiam redes de comunicação eficientes, era inviável adotar essa solução. Então, uma nova

[4] *user-friendly* : ou amigável ao usuário, refere-se a qualquer coisa que torne mais fácil o uso do computador. Por exemplo, os programas orientados a menu são considerados mais *user-friendly* que os programas orientado a comandos. [11]

[5] PC - abreviação comumente utilizada para *Personal Computer*, ou computador pessoal.

necessidade surgiu: interligar os computadores de forma eficiente era desejável para que informações e periféricos pudessem ser compartilhados [19].

Para atender a demanda surgem as redes de computadores, representadas na Figura 3, que possibilitaram, entre outras aplicações: compartilhamento de arquivos e dispositivos, troca de mensagens, acesso a base de dados corporativos.

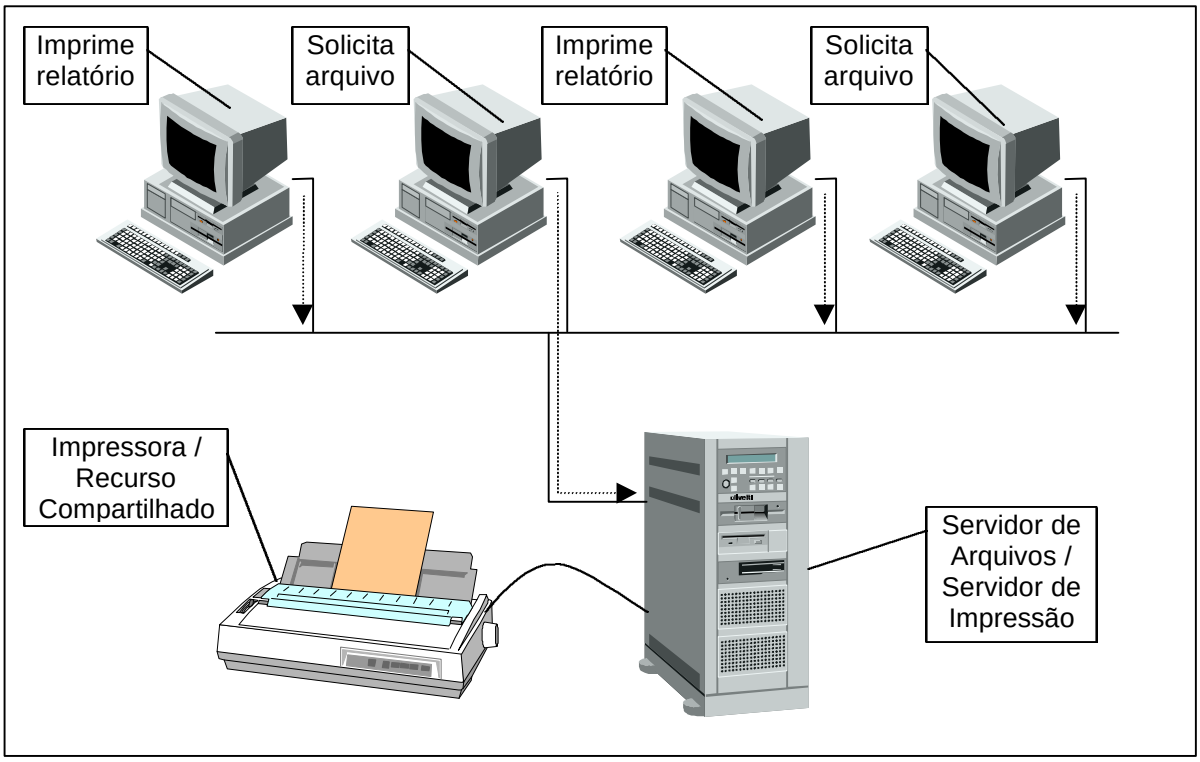

Figura 3: Rede de Computadores

O propósito inicial das redes de computadores era compartilhamento de arquivos e periféricos, o seu uso tornou alguns termos mais conhecidos, tais como: servidor de arquivos e servidor de impressão. Estes serviços pioneiros permitiram que novas aplicações fossem criadas fazendo uso de arquivos comuns a todos os usuários. Mesmo com a solução do problema da falta de veracidade das informações, ainda existia um motivo para que não se adotasse o modelo de computação em rede, o desempenho da arquitetura baseada no compartilhamento de arquivos depende de [13] :

- taxa de utilização do compartilhamento (número de usuários acessando o serviço);
- disputas para atualizações (tentativas de uso de arquivos bloqueados);
- volume de dados transferidos;

Este tipo de solução era eficiente para pequenos ambientes de redes de computadores, com um baixo volume de tráfego de dados.

Então, mesmo com as redes de computadores, alguns sistemas administrativos mais críticos ainda precisavam do *mainframe*, pois o sistema baseado na rede ainda não apresentava uma solução confiável.

Aos poucos, os desenvolvedores perceberam que os aplicativos grandes podiam ser divididos em dois componentes, assim como na programação modular que se beneficia dessa técnica (separar em módulos menores) para facilitar o desenvolvimento e manutenção. Neste caso a divisão resultaria em uma parte específica do usuário (parte cliente) e outra parte específica do serviço prestado (parte servidor) como por exemplo: um serviço de acesso a base de dados.

Como os PCs ofereciam uma capacidade de processamento, a parte cliente poderia ser controlada por ele, isto inclui a interface com o usuário e parte do processamento específico ao usuário, consequentemente a carga da computação para o equipamento servidor seria reduzida, permitindo o uso de equipamentos servidores mais acessíveis, assim como o tráfego na rede. Isto reduziria a dependência nos caros sistemas de *mainframe* para os aplicativos mais pesados.

O baixo custo dos PCs em relação aos *mainframes*, e seu grande potencial para computação, se conectados em rede, incentivaram a indústria de software à trabalhar na arquitetura Cliente/Servidor. Um bom exemplo é a tecnologia de acesso a base de dados que já empregava, internamente, o modelo Cliente/Servidor em

seus projetos e implementações. Nela ambos os componentes eram executados em um mesmo sistema computacional. A situação favorável impulsionou seus desenvolvedores para externar esta decomposição interna, ou seja, dividir em dois produtos independentes, cliente e servidor.

Dessa maneira a carga de processamento da aplicação pode ser dividida entre a estação de trabalho do cliente e o servidor; e estes dois trocam a informação através de uma rede de comunicação[6], como representado na Figura 4.

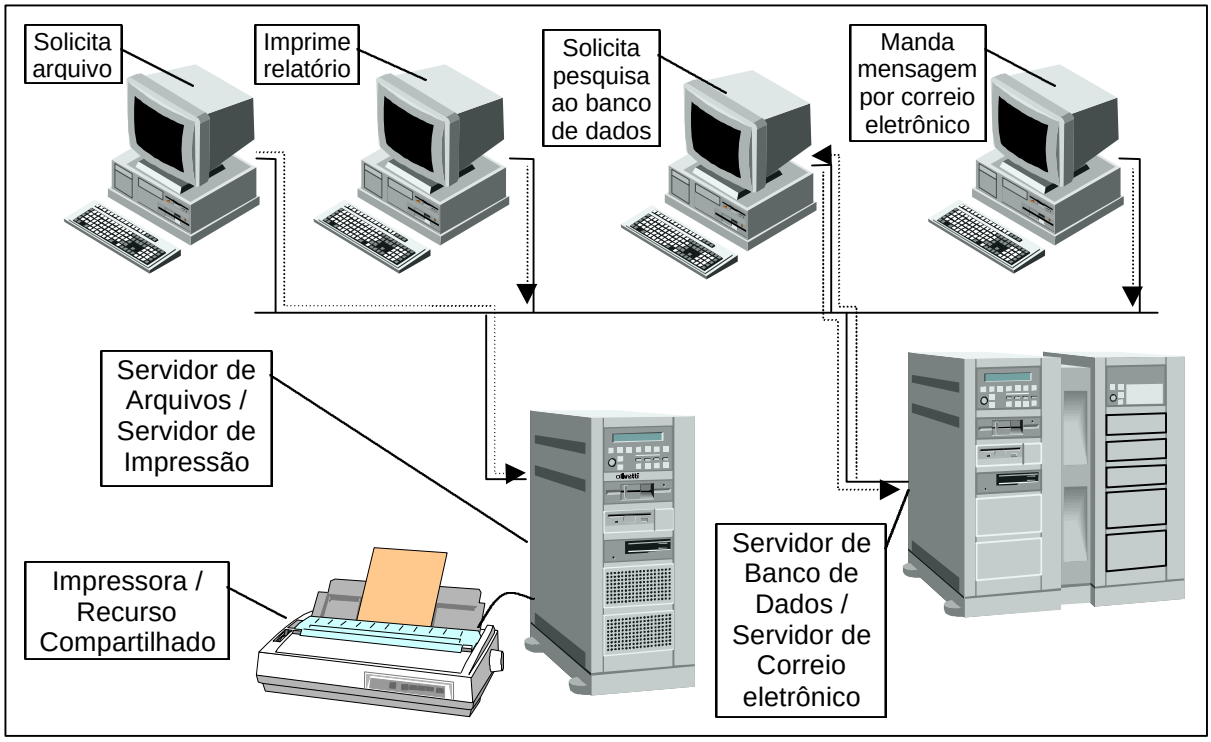

Figura 4: Ambiente Cliente/Servidor

Diferentemente dos sistemas baseados em *mainframe*, que tinham a vantagem de ter uma única fonte de tecnologia para *hardware* e o *software*, a tecnologia Cliente/Servidor depende dos diversos componentes de *hardware* e de *software* que vêm de diferentes fontes. Além disso, um novo componente, o *middleware* responsável pela adaptação das requisições do cliente ao serviço

[6] Rede de comunicação: ou sistema de comunicação, constitui-se de um arranjo topológico que interliga os vários computadores através de enlaces físicos e de um conjunto de regras com o fim de organizar a comunicação. [19]

prestado pelo servidor, começou a ser introduzido. A completa integração de todos esses componentes é essencial para que se possa usufruir de todo potencial da computação Cliente/Servidor.

Aliando as necessidades comerciais à, evolução constante, e a passos largos, do poder de processamento dos computadores pessoais e da capacidade de transmissão das redes de computadores, e a um custo cada vez menor, surge assim, um ambiente favorável a implementação da computação Cliente/Servidor.

A arquitetura Cliente/Servidor apresenta características essenciais para dar suporte a essa ampla faixa de processos comerciais que demandam por: flexibilidade, performance, escalabilidade, usabilidade e baixo custo.

Neste trabalho, pretendemos discorrer sobre a computação Cliente/Servidor, abordando as características principais, os componentes, alguns protocolos, e uma possível adaptação para uma linguagem não predisposta, o COBOL.

2. Cliente/Servidor

Em poucas palavras pode-se dizer que Cliente/Servidor é uma arquitetura computacional que envolve processos clientes requisitando serviços de processos servidores, sendo que cada computador ou processo em uma rede, pode ser um cliente e/ou um servidor. Em sua forma mais básica permite a distribuição de processamento em duas entidades: cliente e servidor. Pode-se dizer também que, a computação Cliente/Servidor provê um mecanismo para computadores cooperarem em uma única tarefa de computação.

2.1. Definições

As referências mais relevantes utilizadas apresentam uma definição geral para Cliente/Servidor, destacamos aquelas que mais se identificaram com o propósito deste trabalho:

> Termo usado para descrever sistemas em rede de computação (processamento) distribuída, no qual as responsabilidades de uma transação são divididas em duas partes: cliente (front-end) e servidor (back-end). Ambos os termos (cliente e servidor) podem ser aplicados a programas ou dispositivos computacionais. Também é chamado de computação (processamento) distribuída. (Cisco Systems [5])

> A tecnologia Cliente/Servidor é uma arquitetura na qual o processamento da informação é dividido em módulos ou processos distintos. Um processo é responsável pela manutenção da informação (servidores) e outros responsáveis pela obtenção dos dados (os clientes). Os processos cliente enviam pedidos para o processo servidor, e este por sua vez processa e envia os resultados dos pedidos. (Grupo de Redes - UFRGS [7])

> O que a computação Cliente/Servidor faz é reunir o módulo cliente e o servidor num modelo mais equilibrado, no qual cada um faz uma parte do processamento, melhorando o desempenho de todo o sistema e aumentando o grau de satisfação dos usuários. (Itec [8])

> Uma abordagem de tecnologia distribuída onde o processamento é dividido por função. O Servidor executa funções compartilhadas – gerenciando comunicações, provendo serviços de banco de dados, etc. O cliente executa funções individuais do usuário, provendo interfaces de acordo com o usuário, executando navegação entre telas, oferecendo funções de ajuda, etc. (Software AG and META Group [10])

> É a extensão lógica da programação modular, que tem como suposição fundamental que a separação de um grande código de um programa em peças constituintes ("módulos") cría a possibilidade para um desenvolvimento mais fácil e uma melhor manutenabilidade. A computação Cliente/Servidor dá um passo adiante por reconhecer que esses módulos não necessitam estar em execução no mesmo espaço de memória. Com esta arquitetura, o módulo que faz a chamada torna-se o “cliente” (aquele que requisita o serviço), e o modulo que é chamado torna-se o “servidor” (aquele que provê o serviço). A extensão lógica disto é possuir clientes e servidores rodando em plataformas de hardware e software apropriadas para suas funções. (Steve Hultquist – comp.client-server [14])

Como podemos ver, a computação Cliente/Servidor possui várias definições que compartilham um mesmo conceito: software e dados são distribuídos no ambiente Cliente/Servidor, de maneira que se possa usufruir mais eficazmente o poder computacional disponível.

2.2. Características de Cliente/Servidor

Quanto as características de funcionamento uma interação típica entre cliente e servidor ocorre de maneira similar a essa:

- O usuário executa o programa cliente para criar uma consulta;
- O cliente conecta-se ao servidor;
- O cliente manda a consulta para o servidor;
- O servidor analisa a consulta;
- O servidor calcula o resultado da consulta;
- O servidor manda os resultados para o cliente;

- O cliente apresenta os resultados para o usuário;
- O processo se repete sempre que necessário.

É válido lembrar que computação Cliente/Servidor não é simplesmente um computador pessoal fazendo acesso a um banco de dados em uma rede, mas sim um conjunto de características que define esta arquitetura.

Segundo Orfali [15], clientes e servidores constituem entidades distintas que trabalham juntas numa rede para realizar uma tarefa, isso não faz com que Cliente/Servidor seja diferente de outras formas de *software* distribuído, então Orfali nos propõe as seguintes características específicas:

- Serviço: o processo servidor é um provedor de serviços e o processo cliente é um consumidor de serviços.
- Recursos compartilhados: um servidor pode atender vários clientes ao mesmo tempo e controlar o acesso dos mesmos aos recursos compartilhados. Como por exemplo, compartilhar uma impressora, uma linha de comunicação.
- Protocolos assimétricos: quem presta o serviço (servidor), fica em estado de espera de solicitações, haja visto que a relação entre cliente e servidor é de vários para um, e por esse motivo, é o cliente quem inicia o diálogo.
- Transparência da localização: normalmente o cliente não necessita conhecer a localização do processo servidor, sendo que o mesmo pode estar em execução na mesma máquina ou em outra máquina na rede.
- Misturar e adaptar: permitir a independência de plataforma de hardware e de software.

- Trocas baseadas na mensagem: a interação Cliente/Servidor é feita através de um mecanismo de troca de mensagens.
- Encapsulamento dos serviços: o modo como uma tarefa é realizada é determinado pelo servidor.
- Escalonabilidade: permitir o acréscimo ou decréscimo de equipamentos clientes, com um pequeno impacto no desempenho, ou seja um escalonamento horizontal, como também a migração para um equipamento servidor maior e mais veloz, ou ainda, para multi-servidores, ou seja um escalonamento vertical.
- Integridade: código e dados do servidor são mantidos de forma centralizada, ou seja, garantem a integridade dos dados compartilhados.

O grupo *comp.client-server* na *Usenet* [14] apresenta em seu *FAQ* as seguintes características para a arquitetura Cliente/Servidor:

- Combinação de uma porção cliente ou *front-end* que interage com o usuário, e uma porção servidor ou *back-end* que interage com o recurso compartilhado. O processo cliente contem a lógica específica de solução e provê uma interface entre o usuário e o resto do sistema aplicativo. O processo servidor atua como um mecanismo de software que controla recursos compartilhados como: base de dados, impressoras, modems, ou processadores poderosos.
- As tarefas do *front-end* e do *back-end* tem requerimentos fundamentalmente diferentes de recursos computacionais tais como, velocidade de processador, memória, velocidade e capacidade de disco, e dispositivos de entrada e saída.

- O ambiente é tipicamente heterogêneo e de vários fornecedores. O equipamento e sistema operacional de cliente e servidor, geralmente não são os mesmos. Processos cliente e servidor comunicam-se através de um conjunto padrão, bem definido, de interfaces para programação de aplicativos (API[7]) e chamadas de procedimento remoto (RPC[8]).

- Uma importante característica de sistemas Cliente/Servidor é a escalabilidade. Eles podem ser horizontal ou verticalmente escaláveis. Horizontalmente escalável significa adicionar ou remover clientes com um pequeno impacto de performance. Verticalmente escalável significa migrar para um equipamento servidor com maior capacidade e velocidade ou para servidores múltiplos.

Podemos citar como outra característica da computação Cliente/Servidor, a flexibilidade oferecida no desenvolvimento de interfaces com o usuário. Pode-se criar um aplicativo independente ao servidor. Isso significa que um aplicativo cliente pode ser desenvolvido para o Macintosh, e o aplicativo servidor pode estar em execução em um *mainframe* IBM. Os aplicativos clientes também poderiam ser desenvolvidos para o Windows, OS/2, Linux, e outros. Isto permite armazenar informação em um servidor central e disseminá-la a diferentes tipos de computadores remotos.

Considerando que a interface de usuário é a responsabilidade do cliente, o servidor tem mais recursos de computação para gastar em analisar questões e disseminar informação. Esta é outra característica da computação Cliente/Servidor,

[7] API - *Application Program Interface*: A Interface para programação de aplicativo é um conjunto de rotinas que um programador pode usar para interagir com a aplicação.

[8] RPC - *Remote Procedure Call*: Chamada de Procedimento Remoto será comentada no tópico 2.3.2

que tende a usar as forças de plataformas de computação divergentes para criar aplicações mais poderosas.

2.3. Componentes da Arquitetura Cliente/Servidor

São considerados componentes da Arquitetura Cliente/Servidor, todo o conjunto aplicado na sua efetivação, tais como [2]:

- Sistemas Operacionais: ambientes onde serão executados os aplicativos, entre os quais o NetWare, OS/2, UNIX, Windows NT, e outros direcionados a aplicações servidoras. E os direcionados a aplicações clientes, como por exemplo o DOS, Windows, MacOs, UNIX, e outros.

- Aplicações Servidoras: aplicações que prestam um serviço, como por exemplo os servidores de banco de dados, servidores de arquivo, servidores de correio eletrônico.

- Aplicações Clientes: aplicações que utilizam o serviço, como por exemplo uma ferramenta de consulta ao banco de dados, um programa cliente de correio eletrônico.

- Sistemas Corporativos: sistemas desenvolvidos no âmbito da empresa, que visam dar suporte aos processos internos da empresa, usando os outros componentes da arquitetura Cliente/Servidor. Alguns exemplos de sistemas corporativos são: sistema de gerenciamento de recursos humanos, sistema de faturamento, sistema de controle de estoque, entre outros.

- *Hardware*: todo dispositivo físico utilizado, tais como: equipamentos de processamento, periféricos de entrada e saída, periféricos de armazenamento e equipamentos de comunicação.

2.3.1. Cliente

A parte cliente da arquitetura Cliente/Servidor é toda e qualquer entidade que solicita, ou utiliza, uma tarefa ou serviço. Comumente é uma aplicação sendo executada em um PC ou estação de trabalho que, através do envio de solicitações, confia em um servidor para executar uma tarefa. Geralmente a parte cliente gerencia a interface com o usuário, incluindo-se os recursos locais, tais como, monitor, teclado, e outros, validando os dados digitados pelo o usuário, enviando as solicitações para o servidor, mostrando as respostas recebidas, e algumas vezes executando a lógica comercial, ou seja, a parte cliente é o que o usuário vê e interage - *front-end*. Os aplicativos clientes podem ser divididos conforme a sua apresentação [15]:

- Clientes sem GUI[9]: aplicativos que apresentam uma interface muito simples com o usuário, geralmente com seqüências fixas para gerar solicitações. Alguns exemplos são, aplicativos textuais em terminais seriais, leitores de códigos de barras, telefone celular;

- Clientes com GUI: são os aplicativos que interagem com o usuário através de uma interface gráfica, sendo que as solicitações ao servidor

[9] GUI: Graphical User Interface (Interface Gráfica com o usuário) – tipo de interface que apresenta a informação graficamente, geralmente com janelas (quadros), botões, ícones, barras de menu, barras de rolagem, entre outros, criando um ambiente mais intuitivo, por esse motivo assume-se que os aplicativos desenvolvidos sob esta interface são mais fáceis de usar do que os aplicativos em interface textual, onde a informação é apresentada em telas baseadas em texto e os comandos são digitados pelo usuário. Esta facilidade de uso não ocorre se o aplicativo GUI não foi bem elaborado, e pode ser conseguida em aplicativos de interface textual se o mesmo for desenvolvido de acordo com o usuário. A GUI também habilita o usuário ao uso de várias sessões de aplicativos ao mesmo tempo.

são provenientes de uma interação humana. Um bom exemplo de ambiente GUI é o *Windows 3.1* .

- Clientes com OOUI[10]: são aplicativos, também em ambiente gráfico, que permitem a interação do usuário com o computador através da manipulação de objetos, e não somente o uso das opções pré-determinadas da GUI. Alguns exemplos de ambientes OOUI são: Windows 95 e 98, GNOME, KDE, entre outros.

É importante salientar que, devido suas facilidades na integração de diferentes tarefas, o ambiente gráfico dá ao usuário condições para tornar-se mais produtivo. Lembrando ainda que o uso de terminais interativos textuais limita o usuário a executar tarefas preestabelecidas, com chances quase nulas de exercitar sua criatividade. [16]

2.3.2. Middleware

A tecnologia de *Middleware* evoluiu durante os anos noventa para prover interoperabilidade, dando suporte a migração para a arquitetura Cliente/Servidor. O termo *Middleware* é respectivo a todo o software distribuído necessário para dar suporte às interações de clientes e servidores [15]. *Middleware* é essencial para migrar aplicações de *mainframe* para aplicações Cliente/Servidor e para prover comunicação em plataformas heterogêneas [13].

[10] OOUI: Object Oriented User Interface (Interface com o usuário orientada a objetos) – tipo de interface que tenta representar a informação em forma de objetos (como no mundo real), criando um ambiente mais intuitivo e próximo do usuário do que a GUI. Um aplicativo em OOUI deve fazer parte do ambiente OOUI em que esta sendo executado, deve ser mais um conjunto de objetos o qual o usuário pode considerar e manipular. Alguns preferem dizer que em um OOUI o usuário é a aplicação.

Pode-se dizer que *middleware* é um *software* de conectividade que permite que aplicações se comuniquem de forma transparente com outros programas ou processos, independente de suas localizações.

Podemos dividir o *middleware* em duas categorias: [15]

- *Middleware* Geral: é a base para a maior parte das interações Cliente/Servidor. Inclui as camadas de comunicação, diretórios distribuídos, serviços de autenticação, rede, chamadas de procedimentos remotos e serviços de enfileiramento de mensagens[11]. Incluindo também as extensões do sistema operacional de rede (SOR).

- *Middleware* Específico do Serviço: viabiliza um tipo particular de serviço Cliente/Servidor, por exemplo, um programa controlador ODBC[12] é um *middleware* específico de banco de dados, a tecnologia ORB[13] é um *middleware* específico para objetos, a MAPI[14] é um *middleware* específico para sistemas de trabalho em grupo.

É importante salientar que o elemento chave de conectividade é o SOR-Sistema Operacional de Rede. O SOR provê serviços como roteamento, distribuição, correio eletrônico, arquivo, impressão, e serviços de administração da rede. O SOR confia em protocolos de comunicação para prover serviços específicos. Os protocolos são divididos em três grupos: protocolos de mídia, transporte e Cliente/Servidor.

---

[11] Serviços de enfileiramento de mensagens: também chamado de *middleware* orientado a mensagem (MOM) ou Servidor de mensagens, será mais detalhado no tópico 2.4.2

[12] ODBC-*Open DataBase Connectivity*: padrão de acesso a banco de dados desenvolvido pela Microsoft Corporation

[13] ORB-*Object Request Broker*: Intermediador de solicitação de objeto. Em [15] encontramos a tradução Corretor de solicitação de objeto, mas preferimos a tradução Intermediador de solicitação de objeto, por se adaptar melhor a sua função, será comentado no tópico 2.4.3

[14] MAPI-Messaging Application Programming Interface: interface do Microsoft Windows que habilita diferentes aplicações, a trabalharem em conjunto no envio de mensagens de correio eletrônico.

Protocolos de mídia determinam os tipos de conexões físicas usados em uma rede, alguns exemplos de protocolos de mídia são: *Ethernet, Token Ring, Fiber Distributed Data Interface*-FDDI.

Um protocolo de transporte provê o mecanismo para mover pacotes de dados de cliente para servidor, alguns exemplos de protocolos de transporte são: o IPX/SPX da Novell, o *AppleTalk* da Apple, *Transmission Control Protocol/ Internet Protocol* -TCP/IP.

Uma vez que a conexão física foi estabelecida e protocolos de transporte foram escolhidos, um protocolo de Cliente/Servidor é necessário para que o usuário possa ter acesso aos serviços da rede.

Um protocolo Cliente/Servidor dita a maneira na qual os clientes solicitam informação e serviços para um servidor, e também como o servidor responde àquele pedido, alguns exemplos de protocolos Cliente/Servidor são, RPC e *socket*. [14]

Chamada de Procedimento Remoto (*Remote Procedure Call* -RPC)

O conceito de RPC foi discutido na literatura desde 1976, sendo que as implementações completas aparecem entre o final da década de setenta e o início da década de oitenta.

Chamadas de procedimento remotas (RPCs) provêem capacidade de padronizar o modo como os programadores escrevem as chamadas, de forma que procedimentos remotos podem reconhece-las e responde-las corretamente. [18]

A RPC aumenta a interoperabilidade, portabilidade, e flexibilidade de uma aplicação permitindo distribuir a aplicação em cima de plataformas heterogêneas múltiplas. Reduz a complexidade de aplicações em desenvolvimento, que transpõem múltiplos sistemas operacionais e protocolos de rede, isolando o desenvolvedor de

aplicação dos detalhes dos vários sistemas operacionais e interfaces de rede. As chamadas de função são a interface do programador quando usada a RPC.

Para ter acesso a porção servidor remota de uma aplicação, chamadas de função especiais, RPCs, devem ser embutidas, dentro da porção de cliente da aplicação Cliente/Servidor, para que isso seja possível, as funções e parâmetros são escritos em uma NIDL[15] e compilados. O compilador NIDL cria *stubs*[16] para a porção cliente e para a porção servidor da aplicação. Estes *stubs* são embutidos no código do cliente e do servidor. O *stub* cliente, empacota os parâmetros num pacote de RPC, converte os dados, e se comunica através da biblioteca RPC. No lado servidor, o *stub* desempacota os parâmetros, chama o procedimento remoto, empacota os resultados e envia a resposta ao cliente. Ou seja, os *stubs* são invocados quando a aplicação requer uma função remota e, tipicamente, suporta chamadas síncronas [17] entre os clientes e servidores.

Se uma aplicação emite um pedido funcional e este pedido é embutido em um RPC, a função solicitada pode ser localizada em qualquer lugar no qual o solicitante tenha acesso autorizado. [18]

As RPCs também provêem, através do uso de padrões como a XDR[18] e NDR[19], serviços de tradução dinâmica de dados, entre processadores com formatos de armazenamento de dados físicos diferentes.

---

[15] NIDL-*Network Interface Definition Language:* Linguagem de definição de interface de rede. Alguns Sistemas Operacionais de Rede oferecem um compilador para esta linguagem.

[16] *Stubs*: considera-se como *stubs*, os trechos de código gerados pelo compilador NIDL

[17] Chamadas síncronas: com relação as aplicações Cliente/Servidor nas quais o cliente pode emitir um pedido e pode esperar pela resposta do servidor antes de continuar seu próprio processamento

[18] XDR-*External Data Representation*: Representação externa de dados. Padrão definido pela Sun Microsystems, Inc.

[19] NDR-Network Data Representation: Representação de dados para rede. Padrão definido pelo Open Software Foundation

Socket

Uma *socket* é um ponto de referência para o qual podem ser enviadas mensagens e de onde podem ser recebidas mensagens. O uso de *sockets* viabiliza a comunicação entre processos remotos, como um exemplo, podemos citar a implementação de RPC, através do uso de *sockets*.

Uma *socket* é considerada uma interface de programação de protocolo, e é utilizada como uma opção do mecanismo de I/O do sistema operacional. Uma *socket* possibilita ao programa de aplicação acessar o protocolo de rede ou comunicação, permitindo que, de maneira amigável[20], os programadores usufruam dos protocolos de rede.

Para usufruírem da interface *socket* os desenvolvedores acionam chamadas de sistema através de funções específicas, tais como, `socket()`, `connect()` , `write()` , `read()` , `bind()` , `listen()` , `accept()` , entre outros.

- `socket()` : cria uma *socket* e retorna o descritor que identifica a *socket* criada, todas as operações relativas a *socket* serão feitas com o uso desse descritor;
- `connect()` : função usada no cliente, tenta estabelecer uma conexão com um servidor remoto;
- `write()` : envia dados em uma conexão de *socket*;
- `read()` : recebe dados de uma conexão de *socket*;
- `bind()` : função necessária somente no servidor, associa uma estrutura de endereçamento que especifica o tipo protocolo, e a porta na qual o servidor vai esperar as conexões.

- `listen()` : função usada no servidor, coloca a *socket* em um modo passivo, a espera de conexões;
- `accept()` : função usada no servidor, aceita a solicitação de conexão.

No tópico 4, descreveremos uma possível implementação de um *middleware* que usa *sockets* para a transmissão de dados.

2.3.3. Servidor

Toda e qualquer entidade que possui recursos ou serviços que são compartilhados ou prestados em um ambiente de rede. O processo servidor age como uma máquina de software que administra recursos compartilhados como bancos de dados, impressoras, linha de comunicação, ou processadores poderosos. Normalmente o servidor atua da seguinte maneira [15]:

- Aguardar as solicitações de iniciativa do cliente: esperar a entrada de solicitações dos clientes e atribuir ao mesmo uma sessão dedicada. Alguns servidores criam um conjunto de sessões reutilizáveis, e outros oferecem os dois ambientes.
- Executar muitas solicitações ao mesmo tempo: processar vários pedidos de cliente ao mesmo tempo. Essa capacidade multitarefa é essencial para que o servidor não tenha todo o seu tempo e recursos tomados por um único cliente.
- Cuidar primeiro dos clientes *VIP*: capacidade de proporcionar diferentes níveis de prioridade de serviço aos seus clientes.

[20] Maneira amigável: as funções para o uso de sockets podem seguir o padrão das funções do mecanismo de I/O do sistema operacional para qualquer outra opção de I/O, por exemplo, abrir (*open*) ou fechar (*close*) um arquivo.

- Iniciar e rodar atividades de tarefas em segundo plano: capacidade de, por exemplo, acionar uma tarefa de atualização descarregando registros de um banco de dados, fora das horas de maior movimento.
- Manter-se rodando: se o programa servidor entrar em colapso, todos os clientes que dependem dos seus serviços sofrerão o impacto, por isso é necessário que o programa servidor e o ambiente em que o mesmo é executado sejam muito robustos.
- Crescer e engordar: geralmente os programas servidores demandam por memória e potência de processamento. O ambiente servidor deve possibilitar crescimento escalável e modular.

Um processo servidor responde a solicitação do cliente executando a tarefa requisitada [4]. Por exemplo: [1]

- Um servidor de hora do dia simplesmente retorna a hora corrente ao cliente que solicitou;
- Um servidor de arquivos recebe solicitações para executar operações que armazenam ou recuperam dados de um arquivo;

2.4. Tipos de arquitetura Cliente/Servidor

2.4.1. Arquitetura de duas camadas (*Two-Tier Architecture*)

Em uma arquitetura de duas camadas o cliente comunica-se diretamente com o servidor, como representado na Figura 5. Geralmente, esta arquitetura, é usada em ambientes pequenos, com menos de 50 usuários. [9]

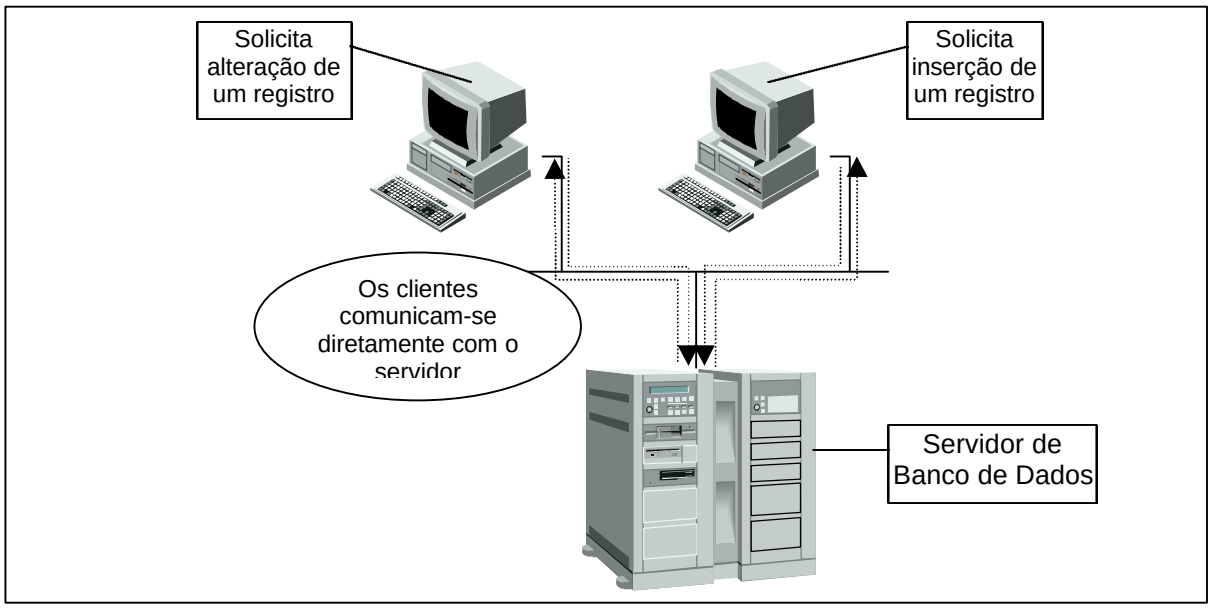

Figura 5: Arquitetura de duas camadas

A arquitetura de duas camadas tem várias limitações, por exemplo:

- Se o número de usuários excede 100, o desempenho começa a deteriorar. Isto acontece devido as conexões, via mensagens *keep-alive*[21], mantidas pelo servidor com cada cliente, até mesmo quando nenhum trabalho está sendo feito. [13]

- A manutenção pode ser custosa em aplicações que necessitem, por exemplo, de alterações no código local e nos procedimentos armazenados[22], no SGBD[23]. Além disso a aplicação alterada deve ser redistribuída para todos as estações clientes. [4]

- Implementações atuais da arquitetura de duas camadas provêem flexibilidade limitada, por exemplo, se mudarmos o SGBD provavelmente

[21] Mensagens *keep-alive* (manter vivas): São as mensagens enviadas para manter uma conexão, que é encerrada se não receber nenhuma mensagem em um tempo predeterminado.

[22] Procedimentos Armazenados: também conhecidos como *stored procedures*. Vem a ser um procedimento compartilhado, armazenado no banco de dados, que pode ser invocado remotamente pelo cliente, disparando todo o código do procedimento armazenado, com uma única solicitação. Reduz o tráfego na rede, e tem melhor desempenho. Os procedimentos armazenados são usados para fazerem valer as regras comerciais e a integridade dos dados, mas o uso fundamental é para criar o lado servidor de uma lógica de aplicação.

[23] SGBD-Sistema Gerenciador de Banco de Dados: sistema que nos habilita a armazenar, modificar e recuperar informações de um banco de dados, podem também armazenar procedimentos, escritos pelo usuário, relativos aos dados que serão disparados conforme programados.

teremos que reescrever manualmente algum código de procedimento armazenado no SGBD. [13]

Um erro comum no desenvolvimento Cliente/Servidor é criar uma aplicação em um pequeno ambiente de duas camadas, com no máximo 100 estações clientes [13], e então aumentá-lo simplesmente somando mais clientes ao servidor. Este incremento normalmente resultará em um sistema ineficaz, a medida que o servidor é subjugado [14]. Para permitir corretamente uma escalabilidade a centenas ou milhares de usuários, é normalmente necessário mudar para uma arquitetura de três camadas.

2.4.2. Arquitetura de três camadas (*Three-Tier Architecture*)

A arquitetura de três camadas, também chamada arquitetura multi-camadas, surgiu para superar as limitações da arquitetura de duas camadas. Na arquitetura de três camadas é adicionada uma camada intermediária, também conhecida como agente, entre o cliente e o servidor, como representado na Figura 6.

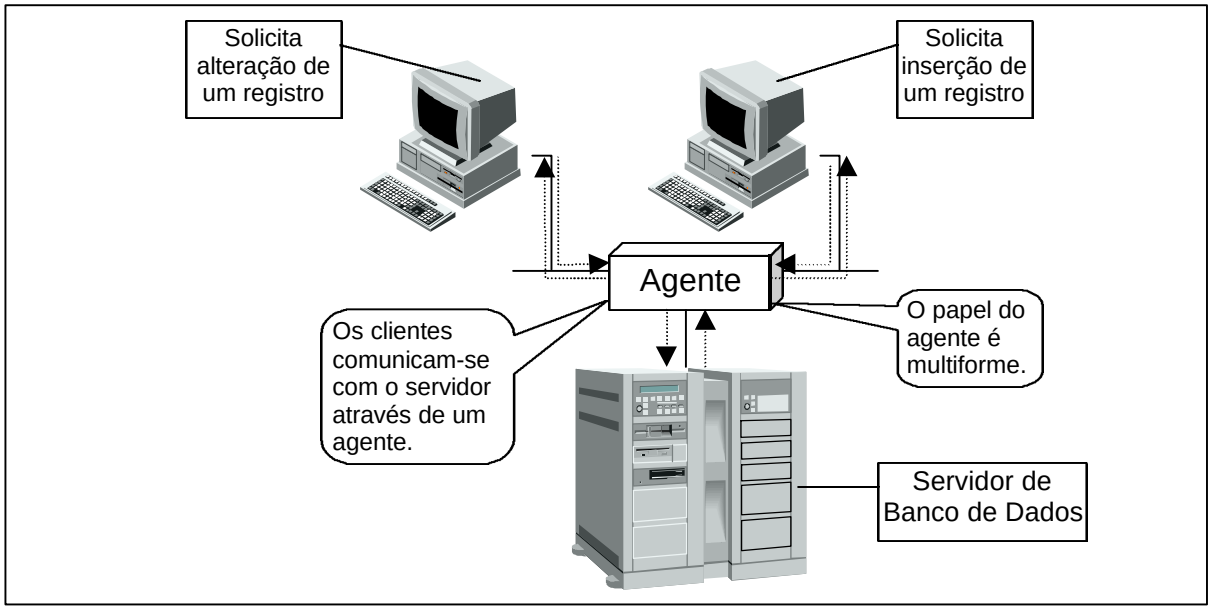

Figura 6: Arquitetura de três camadas

A arquitetura Cliente/Servidor de três camadas foi apresentada para melhorar o desempenho para grupos com um grande número de usuários e melhorar a flexibilidade quando comparada à arquitetura de duas camadas.

O papel do agente é multiforme, podendo prover por exemplo: [14]

- Serviços de tradução: adaptando uma aplicação legada em um *mainframe* para um ambiente Cliente/Servidor.
- Serviços de monitoramento: agindo como um monitor de transação[24] para limitar o número de pedidos simultâneos para um determinado servidor.
- Serviços de agente inteligentes: mapeando um pedido para vários servidores diferentes, colecionando os resultados, e devolvendo uma única resposta ao cliente.

A arquitetura de três camadas tem muitas vantagens sobre a tradicional arquitetura de duas camadas ou camada única, as principais são:

- As eventuais alterações nos procedimentos da camada intermediária são escritas somente uma vez e são adotadas por todo o sistema. [13]
- A modularidade adicionada torna mais fácil de modificar ou substituir uma camada sem afetar as outras camadas. [11]
- Separando as funções de aplicação das funções de banco de dados torna-se mais fácil implementar balanceamento de carga. [11]

[24] Transação: é um conjunto de ações incorporadas com as propriedades ACID que são processadas pelos SGBDs. Neste caso a acrossemia ACID significa:
-Atomicidade: a transação é uma unidade indivisível, isto significa que todas as ações da transação devem ser efetuadas completamente ou falham como um todo.
-Consistência: depois de executada a transação deve deixar o sistema estável ou terá que retorna-lo ao estado inicial.
-Isolamento: o comportamento de uma transação não é afetado por outras transações que estejam ocorrendo simultaneamente.
-Durabilidade: após o seu término os efeitos de uma transação são permanentes

A camada intermediária, ou agente, pode ser implementada de várias maneiras, tais como, monitor de processamento de transação - *TP Monitor*, servidor de mensagem - *Message Server*, entre outros.

Arquitetura de três camadas com TP Monitor

Arquitetura de três camadas na qual a camada intermediária é implementada através de um monitor de processamento de transação, ou *TP Monitor*. O *TP Monitor* é uma tecnologia que usa enfileiramento de mensagem, escalonamento de transação, e serviço de priorização, para oferecer o que Jeri Edwards [15] chama de "um sistema operacional para o processamento de transações". As principais funções de um *TP Monitor* são:

- Gerenciamento de processo: inclui início dos processos de servidor, afunilando o trabalho para eles, controlando sua execução e equilibrando suas cargas de trabalho.
- Gerenciamento de transação: garante que todas as propriedades ACID para todos os programas que rodam sob sua proteção.

Neste modelo de arquitetura o cliente conecta-se ao *TP Monitor* (camada intermediária), e não ao servidor de banco de dados. A transação enviada pelo cliente é aceita pelo *TP Monitor* que a enfileira e então tem a responsabilidade de dirigi-la a conclusão, liberando assim o cliente.

Quando um *TP Monitor* é fornecido por terceiros é chamado "*TP Heavy*", o qual oferece mais funções e pode servir milhares de usuários. Quando é embutido no SGBD, podendo ser considerado uma arquitetura de duas camada, é chamado "*TP Lite*", que provê poucas funções, e deve ser usado quando se trabalha com um

banco de dados de um único fornecedor e com um número não tão maior que 50 usuários. [15]

A tecnologia *TP Monitor Heavy* também provê: habilidade para atualizar múltiplos SGBDs diferentes em uma única transação; conectividade para uma variedade de fontes de dados inclusive arquivos planos, SGBDs não relacionais, e o *mainframe*; habilidade para anexar prioridades a transações; segurança robusta.

Arquitetura de três camadas com Servidor de Mensagens

É outro modo para implementar arquiteturas de três camadas. Este modelo usa o esquema de armazenar e encaminhar mensagens, as quais são priorizadas e processadas assincronamente.

Uma mensagem leva informação sobre o que é, onde precisa ir, e o que deveria acontecer quando alcançar seu destino. Cada mensagem é formada, pelo menos, por duas partes: um cabeçalho que contém informação de prioridade, o endereço e número de identificação, e o corpo que contém a informação sendo enviada que pode ser qualquer coisa inclusive texto, imagens ou transações.

Assim como o *TP Monitor*, o servidor de mensagem, usa um processo de afunilamento e provê conectividade a SGBD e outras fontes de dados. A diferença principal entre a tecnologia *TP Monitor* e o servidor de mensagem é que a arquitetura de servidor de mensagem enfoca-se em mensagens inteligentes, enquanto que o ambiente *TP Monitor* tem a inteligência no monitor, e trata transações como simples pacotes de dados. O *TP Monitor* interroga e processa a transação, se não entende os dados eles não serão processados. Em uma arquitetura baseada em mensagem, existe inteligência na mensagem. O servidor de mensagem torna-se apenas um recipiente de mensagens e de seus procedimentos armazenados. Sendo que para um tipo de mensagem, esta camada intermediária,

pode servir simplesmente como um ponto de roteamento entre dois tipos de serviços de rede, e para outro tipo de mensagem, a camada intermediária pode executar um procedimento armazenado ou regra de negócio conforme direcionado pela mensagem. [6]

Sistemas de mensagens são projetados para oferecer robustez, pois em caso de falha, a entrega de mensagens pode ser programada para ser efetuada assim que a falha for sanada.

2.4.3. Arquitetura Cliente/Servidor com Objetos Distribuídos

A introdução da promissora tecnologia de objetos distribuídos no ambiente Cliente/Servidor altera radicalmente o modo como esses sistemas são desenvolvidos. A promessa é convincente:

> Seremos capazes de juntar complexos sistemas de informações Cliente/Servidor simplesmente montando e estendendo os componentes reutilizáveis de software. Qualquer um dos objetos poderá ser modificado ou alterado sem afetar os demais componentes do sistema nem o modo como eles interagem. Os componentes podem ser despachados na forma de coleções de bibliotecas de classe pré-montadas em *estruturas*, onde se sabe que todas as peças trabalham juntas para executar uma tarefa específica. As estruturas são subsistemas de trabalho que revolucionam o modo como criamos os sistemas Cliente/Servidor. Elas prometem proporcionar o máximo e recursos do tipo misturar e combinar. [15]

Com a tecnologia de objetos distribuídos, será possível, por exemplo, para um administrador de sistemas, ajustar o desempenho da rede, movendo os objetos de um *hardware* sobrecarregado para computadores sub-utilizados. Ou ainda, se precisássemos de computação tolerante à falhas, poderíamos implementar cópias de objetos sobre servidores múltiplos. Desse modo se algum servidor estivesse sem funcionar, seria possível ir para outro local executar o serviço. [6]

Esta arquitetura é baseada na tecnologia de ORB, e tem como outros componentes os serviços de objetos distribuídos, a tecnologia de documento

composto, os SGBDs de objeto e as estruturas de objetos, ou *frameworks*. Estes componentes serão brevemente comentados nos próximos tópicos.

ORB - *Object Request Broker*

ORB é uma tecnologia de *middleware* que controla a comunicação e troca de dados entre objetos, ou seja ele habilita as relações cliente/servidor entre objetos. Seu funcionamento é resumido na seguinte citação:

> Usando um ORB, um cliente pode invocar de forma transparente um método num objeto servidor, que pode estar na mesma máquina ou através da rede. O ORB intercepta a chamada e é responsável pela localização de um objeto que possa implementar a solicitação, passando-lhe os parâmetros, invocando seu método, e retornando os resultados (OMG - Object Management Group - 1996) [13].

É importante salientar que para que um objeto possa se comunicar e interagir com outro objeto, ele tem que conhecer a interface do objeto alvo, para isso é atribuída mais uma função ao ORB, a definição da interface.

Os detalhes de implementação do ORB geralmente não são importantes para o desenvolvedor que constrói sistemas distribuídos. Os desenvolvedores somente estão preocupados com os detalhes para definição da interface do objeto. É responsabilidade do ORB prover a ilusão de localidade, em outras palavras, fazer parecer que o objeto é local ao processo cliente, enquanto na realidade pode residir em um processo ou máquina diferente.

Um ORB traz as seguintes vantagens para o desenvolvimento de aplicações distribuídas: [12]

- Rapidez no desenvolvimento de sistemas distribuídos;
- Total utilização do paradigma de desenvolvimento de sistemas orientados a objetos em ambiente distribuído;
- Transparência de utilização da rede;

- Integração suave entre aplicações legadas e novos sistemas orientados a objetos;
- Definição de uma arquitetura básica comum de desenvolvimento de sistemas;
- Desenvolvimento de sistemas de forma escalável.

As principais tecnologias de ORB são: [13]

- A especificação CORBA-*Common Object Request Broker Architecture* do OMG-*Object Management Group;*
- O COM-*Component Object Model* da Microsoft;
- O DSOM-*Distributed System Object Model* da IBM;
- A RMI-*Remote Method Invocation* que é especificada como parte da linguagem/máquina virtual Java. A RMI permite executar objetos Java remotamente. Isto provê capacidades de ORB como uma extensão nativa de Java.

Serviços de objetos distribuídos

Permitem criar, gerenciar, nomear, mover, copiar, armazenar e restaurar objetos.

Documento composto

É a tecnologia que permite ao aplicativo que gerencia o documento comunicar-se com os aplicativos que possuem os objetos que se encontram dentro do documento. Um bom exemplo, é o OLE-*Object Linking and Embedding* da Microsoft.

SGBD de objeto

Sistema gerenciador de banco de dados de objeto que, diferentemente das outras contrapartidas de SGBD, apresenta características, que o tornam sintonizado com o ambiente de objetos distribuídos.

Estruturas de Objetos ou *Framework*

Prometem revolucionar o modo como construímos nossos sistemas distribuídos. Oferecem subsistemas de software pré-fabricados flexíveis e personalizáveis.

3. Comunicação entre processos (IPC- Interprocess Communication)

Um processo pode ser entendido, a grosso modo, como um programa em execução. Considerando que vários processos podem estar em execução, e freqüentemente compartilham recursos, se faz necessário o uso de mecanismos que possibilitem, e coordenem, comunicação e sincronização entre os mesmos. [21]

A comunicação permite que processos que interagem na resolução de determinada aplicação troquem informações. A sincronização provê controle de acesso[25] e controle de seqüência[26], para garantir a consistência na comunicação entre processos.

Os processos podem estar em execução no mesmo computador ou em diferentes computadores conectados através de uma rede. Por isso existem mecanismos de IPC que permitem comunicação entre processos remotos, tal como, o RPC já comentado no tópico 2.4.2.

A comunicação entre processos é requerida em todos os sistemas operacionais multitarefa ou multiprocesso, e geralmente não é suportada por

[25] Controle de acesso: necessário quando há disputa entre processos pela manipulação de algum recurso

[26] Controle de seqüência: é utilizado para que se determine uma ordem na qual os processos, ou parte deles, devem ser executados

sistemas operacionais monotarefa/monoprocesso como o DOS. O OS/2 e o Microsoft Windows suportam um mecanismo IPC chamado DDE[27].

3.1. Mecanismos de IPC

Os mecanismos de IPC tem a característica de implementarem de maneira distinta a comunicação e sincronismo entre processos. Dentre esses mecanismos destacamos: semáforos, troca de mensagens e filas de mensagens.

3.1.1. Semáforos

Um semáforo é uma variável compartilhada inteira e não negativa que indica o estado do recurso a qual esta associado. É um mecanismo de IPC que pode implementar controle de acesso, e controle de seqüência.[20]

Um semáforo só pode ser manipulado por duas operações, *down*[28] e *up*[29], que são ações indivisíveis, isso garante que quando um processo inicia um das operações sobre um semáforo, nenhum outro processo terá acesso ao semáforo até que a operação se conclua.[21]

Em um semáforo que esta sendo usado para controle de acesso, o valor 0 (zero) indica que o recurso esta sendo utilizado por outro processo, o valor 1 indica que o recurso esta disponível. Em um semáforo que esta sendo usado para controle de sequência, o valor 0 (zero) indica que não há nenhum processo em espera, um valor maior que 0 (zero) indica a quantidade de processos em espera.

---

[27] DDE-Dynamic Data Exchange-Troca dinâmica de dados: um mecanismo de IPC baseado no conceito Cliente/Servidor.

[28] *down*: implementada como: down(semáforo); se semáforo=0 então processo_fica_em_espera senão semáforo=semáforo-1

[29] *up*: implementada como: up(semáforo); semáforo=semáforo+1

Quando um processo deseja usar um recurso compartilhado, ele executa, sobre determinado semáforo, uma operação *down*, esta operação verifica o valor do semáforo e se o valor for 0 (zero) ela coloca o processo solicitante em espera, senão o processo solicitante continua em execução. Quando o processo não precisar mais do recurso, ele deve executar uma operação *up* sobre o semáforo determinado, liberando o recurso para outros processos.

3.1.2. Troca de mensagens

É um método para comunicação entre processos, no qual, processos enviam e recebem mensagens ao invés de compartilhar variáveis. As operações de troca de mensagens são generalizadas pelo uso de primitivas *send/receive* (envia/recebe), cujo as sintaxes podem ser:

send mensagem *to* processo_destino

receive mensagem *from* processo_origem

Essas operações podem ser:

- Bloqueante: ou síncrona, o processo que envia a mensagem fica bloqueado até que receba uma confirmação de recebimento da mensagem do processo receptor;
- Não-bloqueante: ou assíncrona, o processo que envia a mensagem não necessita esperar confirmação, mas neste caso a mensagem deve ser armazenada em um *buffer*[30].

As primitivas *send/receive* fornecem subsídios para a construção de algumas abstrações de comunicação, tal como o RPC. [17]

[30] *buffer*: dispositivo ou área de armazenamento temporário de informações/dados durante a transferência desses dados de uma parte do sistema para outra, regulando o seu fluxo entre dispositivos de velocidades diferentes.

3.1.3. Filas de mensagens

Permitem que, através do uso de funções de manipulação de filas de mensagem, os processos mandem listas formatadas de dados para qualquer processo. Existem quatro chamadas para a troca de mensagens:

- `msgget()` : retorna um descritor de mensagens que designa a lista de mensagens a ser utilizada.
- `msgctl()` : possui uma opção que seta e retorna parâmetros associados com o descritor de mensagens e uma opção que remove o descritor.
- `msgsnd()` : manda a mensagem desejada
- `msgrcv()` : recebe a mensagem.

Quando é usado o `msget()` para criar um novo descritor, o sistema operacional procura o vetor de filas de mensagens para verificar se existe uma com a chave dada, caso não tenha esta entrada o sistema operacional aloca uma nova fila na estrutura, inicializa e retorna um identificador para o usuário; caso contrário verifica a permissão e retorna o identificador.

Um processo usa o `msgsnd()` para mandar uma mensagem. Esta chama um descritor para a lista de mensagens, um ponteiro para a estrutura, um contador com o tamanho do vetor de dados, e um *flag* que indica para o sistema operacional se deve executar fora do espaço do *buffer* interno.

O sistema operacional aloca espaço para a mensagem no mapa de mensagens e copia os dados para o espaço do usuário. Ele aloca o cabeçalho da mensagem e coloca no fim da lista encadeada de cabeçalhos de mensagens; grava o tipo e tamanho da mensagem no cabeçalho, seta o apontador para a mensagem.

Então, o sistema operacional dispara o processo que estava esperando pela mensagem.

Um processo recebe uma mensagem pela chamada `msgrcv()` que contém o endereço da estrutura do usuário, aonde está a mensagem e o tamanho da mensagem. O sistema operacional verifica se o usuário tem direito de acesso; se possuir pega a primeira mensagem da lista encadeada, ou se especificado, a primeira mensagem do tipo solicitado. Se o tamanho é menor ou igual ao tamanho requisitado pelo usuário, o sistema operacional copia a mensagem para a estrutura de usuário, decrementando o contador de mensagens da lista, o sistema operacional ainda seta o tempo de transferência, a identificação do processo que recebeu a mensagem, ajusta os apontadores da lista encadeada e libera a área do sistema operacional que tinha armazenada a mensagem.

No tópico 4 descreveremos uma possível implementação de *middleware* que usa fila de mensagens.

4. Possível implementação de um middleware para linguagem não predisposta

4.1. Motivação

Em 1960, uma equipe formada por vários fabricantes de computadores e o Pentágono, desenvolveu o COBOL que visava conquistar o ambiente comercial. O COBOL projetado com a intenção de proporcionar leitura fácil de programas de computador e maior independência de máquina possível, com poucas alterações poderia ser executado em qualquer computador que possuísse um compilador. A meta era criar um idioma empresarial padrão que teria de ser satisfatório nas seguintes características: bom com cálculos, armazenar grandes quantias de dados, e recuperar estes dados com precisão e de maneira eficaz. Tinha que se auto-documentar, isto significa que alguém diferente do desenvolvedor original seria capaz de entender o código de um aplicativo COBOL em um período razoável de tempo, e poderia fazer alterações no mesmo. Segundo um pesquisa publicada em 1997, existiam 192 bilhões de linhas de código COBOL, e esse número é acrescido de 5 bilhões de linhas a cada ano, ou seja, ainda existe uma boa quantidade de aplicativos desenvolvidos em COBOL [3].

Oferecendo a uma linguagem não predisposta um meio no qual ela possa efetuar interações Cliente/Servidor, estaremos possibilitando a ela uma maior sobrevida dentro de uma organização.

4.2. Descrição do ambiente

A situação que descreveremos é a realidade de um grupo de empresas que compartilham serviços comuns, tais como: serviços contábeis, serviços de cobrança, serviço jurídico, serviços de almoxarife, entre outros.

O grupo de empresas esta dividido em três prédios como representado na figura 7:

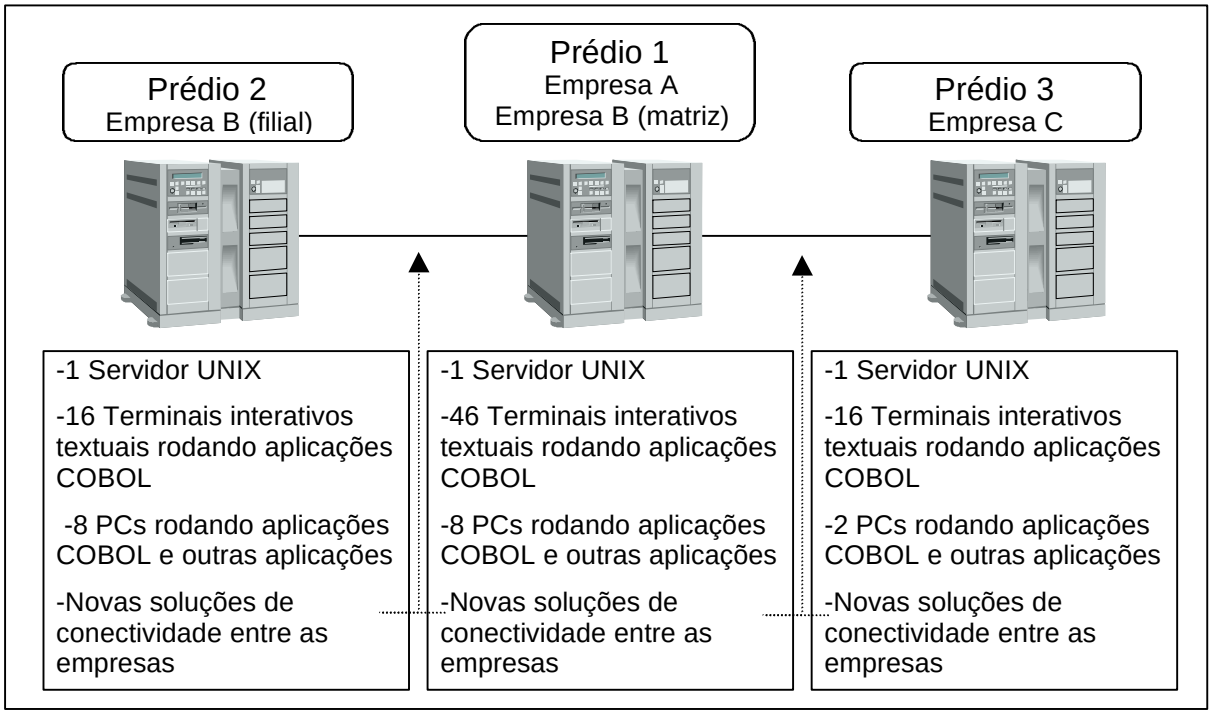

Figura 7: Ambiente computacional do grupo de empresas

Segundo classificação sugerida em [16], o sistema computacional predominante, comum a todas as empresas do grupo, é o sistema centralizado multiterminal, que nada mais é que um sistema centralizado, como comentado na

introdução deste trabalho. Como existe, em cada prédio, uma rede de computadores, esses ambientes tornam-se híbridos, ou seja, centralizado multiterminal com rede de computadores.

Os usuários, realizam suas tarefas, interagindo em seus respectivos terminais, com os aplicativos do sistema corporativo que, esta hospedado nos Servidores UNIX, e é totalmente desenvolvido em COBOL. Isso evidencia a predominância do sistema centralizado dentro deste grupo de empresas. As tarefas como, redação de cartas, elaboração de folhetos promocionais, desenvolvimento de aplicativos para resolução de problemas específicos, entre outras, são realizadas em PCs que oferecem algum tipo de interface gráfica. É válido acrescentar que, os usuários dos PCs acessam, constantemente, o sistema corporativo, através de aplicativos de emulação de terminal.

Como já comentado, as empresas do grupo compartilham serviços comuns. O sistema corporativo provê aplicativos COBOL para todos esses serviços, e tais aplicativos estão completamente integrados. Os diretores do grupo têm confiança na solução existente, mesmo por que, já estão a utilizando a mais de 10 anos.

Com a recente inserção de alguns produtos para conectividade entre as empresas do grupo, tais como, roteadores e circuito especializado de comunicação de dados, bem como, o uso de um protocolo de rede, o TCP/IP, comum a todas as empresas, surge um ambiente favorável para uma possível implementação de um *middleware* que possibilite interações Cliente/Servidor entre as empresas do grupo.

4.3. Características de implementação

Esta *middleware* pode ser classificado como um *middleware* orientado a mensagem, também conhecido como servidor de mensagens, que foi discutido no tópico 2.4.2 deste trabalho. Assim sendo, essa implementação também é baseada no esquema de armazenar e encaminhar mensagens e também oferece conectividade a base de dados. Nesta implementação não foram incluídas:

- Priorização de mensagens;
- Capacidade de execução de um procedimento armazenado ou regra de negócio conforme direcionado pela mensagem.
- Re-entrega em caso de falha, a entrega de mensagens não pode ser programada para ser efetuada assim que a falha for sanada.

A figura 8[31], mostra o esquema de funcionamento da implementação do *middleware*.

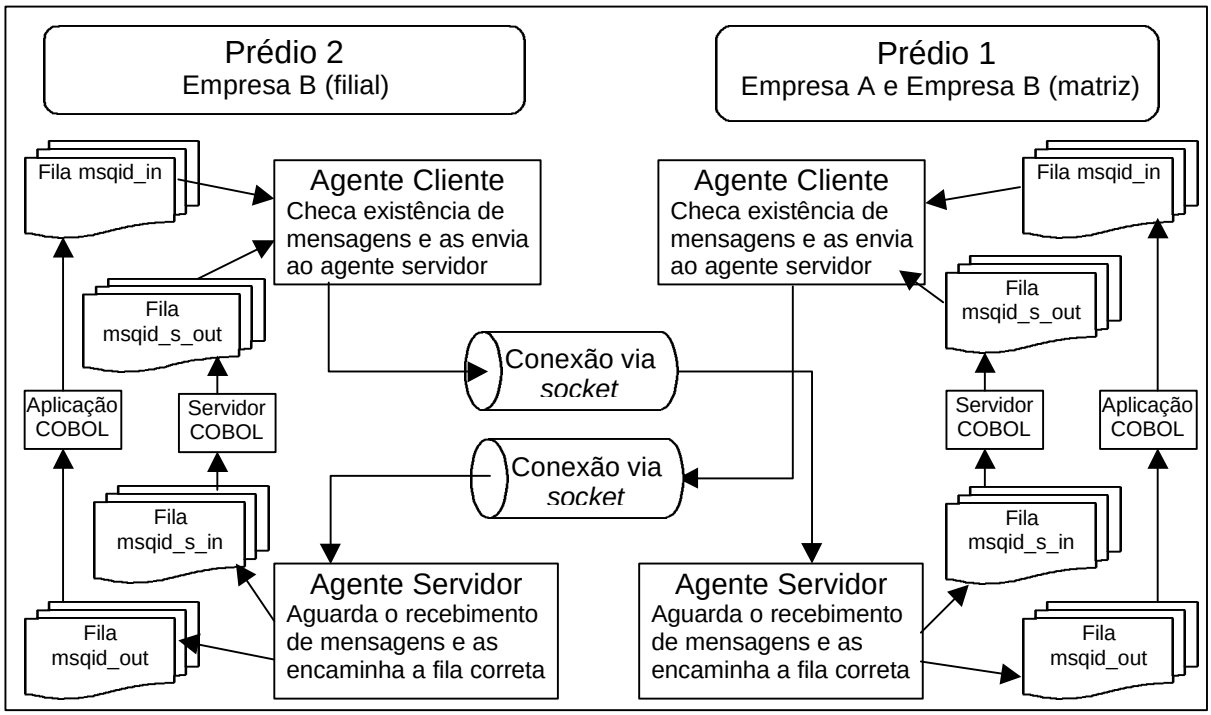

Figura 8: Funcionamento do *middleware*

[31] Na figura 8 não colocamos o Prédio 3 para permitir um esquema mais claro e de melhor entendimento, já que o Prédio 3 usa a mesma implementação de *middleware*, sua presença não é necessária.

Como podemos ver, para cada empresa, são definidas 4 filas de mensagens, cada uma com um propósito específico, como segue:

- Fila msqid_in: é a fila de mensagens para a qual os clientes COBOL enviam suas solicitações;
- Fila msqid_out: é a fila de mensagens na qual os clientes COBOL recebem suas respostas;
- Fila msqid_s_in: é a fila de mensagens na qual o servidor COBOL recebe as solicitações;
- Fila msqid_s_out: é a fila de mensagens para a qual o servidor COBOL envia suas respostas;

Em cada empresa, é executada uma cópia do *middleware*, o qual acessa filas de mensagens e transmite solicitações, este *middleware* é composto por dois agentes: agente cliente e agente servidor. Para um melhor entendimento do funcionamento do agente cliente e do agente servidor, segue o código, em Linguagem C, nas figuras 9 e 10 respectivamente. Para facilitar a explanação, ambos os códigos foram acrescidos de números de linha, que no código real não são utilizados.

Agente Cliente (figura 9):

Conecta-se (linha 25) via *socket* ao Agente Servidor e configura (linha 30) as filas de mensagens as quais ira acessar (filas msqid_in e msqid_s_out). Então verifica o estado da fila de solicitação dos clientes COBOL, ou seja, a fila msqid_in (linha 36), e o estado da fila de respostas do servidor COBOL, ou seja, fila msqid_s_out (linha 49). Sempre que achar mensagens disponíveis deve pega-las da

fila (linha 42 ou 55), e envia-las (linha 45 ou 58) no formato apropriado ao agente servidor do computador remoto, então refazer todo processo.

```
1  /*     arquivo:  client.c
2         uso    : Programa Agente   cliente*/
3  #include " param.h"
4  #include " trans.h"
5  extern  int  fd;
6  extern  int msqid_in,  msqid_out;
7  extern  int msqid_s_in,  msqid_s_out;
8  int fd;
9
10 void main(  argc,  argv )
11 int argc;
12 Char * argv[];
13 {
14 Char envia[MAX_LEN_MSG];
15 Long  tam,m_pid,conect;
16        if (argc != 2)
17        {
18               printf("\ nUso:  client  conexao\n");
19               exit(1);
20        }
21
22        /* abre um  socket TCP e conecta ao servidor */
23        if(conexao_efetuada(& fd) == -1)  exit(-1);
24        printf("Abriu  socket e conectou com   servidor\n");
25        conect = atol( argv[1]);
26
27        /* configura fila de mensagens dos clientes */
28        conf_client_msg(  conect );
29        limpa(&envia);
30
31        while (1)
32        {
33         /*checa presença de mensagem na fila de solicitações dos clientes COBOL*/
34               if (  st_msg_in() )
35                      {
36                             limpa(&envia); /* limpa a   string envia */
37                             /*  le a solicitação e o número do processo na */
38                             /* fila de solicitações dos clientes COBOL    */
39                             m_pid=le_msg(msqid_in,envia,0L, &  tam);
40                             /* envia a solicitação para o a  gente servidor */
41                             xtr(PERGUNTA,  m_pid, envia,  tam);
42                      }
43
44          /*checa presença de mensagem na fila de respostas do servidor COBOL*/
45               if (  st_msg_s_out() )
46                      {
47                             limpa(&envia); /* limpa a   string envia */
48                             /*  le a resposta e o número do processo na */
49                             /* fila de respostas do servidor COBOL     */
50                             m_pid=le_msg(msqid_s_out,envia,0L, &   tam);
51                             /* envia a resposta para o agente servidor */
52                             xtr(RESPOSTA,  m_pid, envia,  tam);
53                      }
54        }
55 }
```

Figura 9: Implementação do Agente Cliente

```
1  /*     arquivo:  server.c
2         uso    : Programa Agente servidor   */
3  #include " param.h"
4  #include " trans.h"
5  extern  int  fd;
6  extern  int  msqid_in,  msqid_out;
7  extern  int  msqid_s_in,  msqid_s_out;
8  int  fd;
9
10 void  main(  argc,  argv )
11 int  argc;
12 char  * argv[];
13 {
14 char  recebe[MAX_LEN_MSG];
15 int  trc;
16 long  tam,m_pid,conect,tipo;
17        if  (argc  != 2)
18        {
19               printf("\ nUso:  server  conexao\n");
20               exit(1);
21        }
22        /* abre uma  socket para TCP e espera   conexao cliente */
23        if( cliente_conectado(& fd) == -1)  exit(-1);
24        printf("Cliente conectado ao   servidor\n");
25        conect = atol( argv[1]);
26        /* configura fila de mensagens do servidor */
27        conf_server_msg(  conect );
28        while (1)
29        {
30               tipo=3;
31               m_pid=0;
32               limpa(&recebe); /* limpa   string recebe */
33               /* espera o recebimento de uma mensagem do */
34               /* agente cliente pelo   socket conectado */
35               if (  trc=xrc(& tipo,&m_pid,&recebe,&tam) > 0)
36                      {
37                       /* se recebeu mensagem então verif   ica o tipo */
38                       /* se for pergunta grava mensagem na fila */
39                       /* de solicitações do servidor COBOL */
40                       if (tipo == PERGUNTA)   grava_msg( msqid_s_in, recebe,   tam,
   m_pid);
41                       /* se for resposta grava mensagem na fila */
42                       /* de r espostas dos clientes COBOL        */
43                       if (tipo == RESPOSTA)   grava_msg( msqid_out, recebe,   tam,
   m_pid);
44                      }
45        }
46 }
```

Figura 10: Implementação do Agente Servidor

Agente Servidor (figura 10)

Aguarda (linha 23) conexão via *socket* do agente cliente, configura (linha 27) as filas de mensagens as quais irá acessar (filas msqid_s_in e msqid_out), e espera (linha 35) a chegada de mensagens pela conexão. Quando receber uma mensagem verificar (linha 40 ou 43) seu tipo, se a mensagem for do tipo PERGUNTA deve enviá-la (linha 40 ou 43) a fila de mensagens de solicitação ao

servidor COBOL (fila msqid_s_in), se for do tipo RESPOSTA deve enviá-la (linha 40 ou 43) a fila de mensagens de resposta dos clientes COBOL (fila msqid_out), então refazer todo o processo.

É importante salientar que as conexões entre agente cliente e agente servidor só trafegam dados em um sentido, significando que o agente cliente somente envia dados, e o agente servidor somente recebe dados, como representado na figura 8. Ressaltando também que o agente cliente só recebe mensagens das filas que ele acessa, e o agente servidor só envia mensagens para as filas que ele acessa.

Os clientes COBOL, ou seja, os aplicativos que geram solicitações, ficam em espera até que sua solicitação seja respondida, isso revela o sincronismo incluído no cliente COBOL, em contrapartida o *middleware* atua de forma assíncrona.

Os aplicativos COBOL fazem chamadas as funções disponibilizadas pelo *middleware*, para que possam enviar e receber mensagens, como exemplificado na figura 11.

```
/*
       chamadas utilizadas nos aplicativos COBOL cliente   s.
*/

       CALL   "micro"   USING   CHAVE   REGISTRO   TAM-REGIS   STATUS   CONEXAO.

 ----------------------------------------//----------------------------------------

/*
       chamadas utilizadas no aplicativo COBOL servidor
*/

       MOVE    EMPRESA-DESTINO   TO   CONEXAO
       CALL    " pega_msg"  USING   CHAVE   STATUS   CONEXAO
       ...
       CALL    " devolve_msg"   USING   TAM-REGIS   REGISTRO   STATUS   CONEXAO.
```

Figura 11: Chamadas de funções do *middleware* nos aplicativos COBOL

Na figura 11 vemos o uso das funções:

- `micro:` coloca uma solicitação na fila msqid_in, e aguarda uma resposta na fila msqid_out;
- pega_`msg:` verifica a presença e lê solicitações da fila msqid_s_in;
- devo`lve_msg:` coloca a resposta na fila msqid_s_out;

O código do middleware, incluindo as funções chamadas pelo COBOL, será colocado em anexo.

5. Considerações finais

Este trabalho procurou esclarecer a tecnologia Cliente/Servidor, de modo a oferecer subsídios à implementação de um *middleware* para linguagem COBOL, possibilitando a esta ferramenta uma maior sobrevida dentro de uma organização.

Discorrendo sobre importantes tópicos para a efetivação da tecnologia Cliente/Servidor, tais como, o cliente, o servidor, o *middleware*, tipos de arquiteturas, comunicação entre processos, entre outros, pudemos constatar a potencialidade dessa tecnologia, que oferece diversas opções para suprir as necessidades comumente encontradas em um ambiente comercial.

Vimos que essa tecnologia de conceito simples, clientes interagem com servidores, mas de eficiência comprovada, se aplicada corretamente permite um melhor aproveitamento do poder computacional disponível, oferecendo uma atraente relação custo/benefício.

Com o conceito de *middleware* a tecnologia Cliente/Servidor torna-se extremamente flexível. Dentre as opções de *middleware*, apresentamos os monitores de transações, que garante as propriedades ACID também comentadas neste trabalho, falamos sobre o *middleware* orientado a mensagem, o qual serviu de base para nosso último tópico, a implementação de um *middleware* para linguagem não predisposta.

Os tipos de arquiteturas nos mostraram como a tecnologia Cliente/Servidor divide o processamento entre seus componentes. Ficamos entusiasmados com as promessas da Arquitetura Cliente/Servidor com Objetos Distribuídos.

Entendemos por que a comunicação entre processos é tão importante em um ambiente Cliente/Servidor. Mais especificamente as filas de mensagens, que foram utilizadas no objetivo final desse trabalho.

Enfim, durante o decorrer deste estudo, percebemos que para a implementação do *middleware* poderíamos utilizar, opções como RPC e CORBA, os quais se tornam possibilidades para um trabalho futuro.

Esperamos ainda que o conteúdo aqui apresentado sirva como base para outros trabalhos na área de Computação Cliente/Servidor.

Anexo 1 - Implementação do *middleware*

```
/*
      arquivo: includes.h
      uso    : cabecalho
*/

#include <standards.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <termio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
```

```
 ----------------------------------//-----------------------------------
```

```
/*
      arquivo: param.h
      uso    : parametros para construcao do programa de comunicacao
*/
#define MAX_LEN_MSG      1024
#define id_q_in_client          0x100
#define id_q_out_client         0x200
#define id_q_in_server          0x300
#define id_q_out_server         0x400
```

```
 ----------------------------------//-----------------------------------
```

```
/*
      arquivo: tcpdef.h
      uso    : cabecalho de definicoes para programas TCP/IP
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#define SERV_TCP_PORT 6000
#define SERV_HOST_ADDR "192.168.0.2"
```

```
/*
      arquivo: config.c
      uso    : configura socket e servidor
*/
#include "trans.h"
#include "param.h"
#include "tcpdef.h"
extern int fd;
int conexao_efetuada(int *soquete)
{
struct sockaddr_in serv_addr;/*prepara estrutura com endereco do server*/
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
/*serv_addr.sin_addr = inet_addr(SERV_HOST_ADDR);*/
serv_addr.sin_port = SERV_TCP_PORT+1;
if((*soquete = socket(AF_INET, SOCK_STREAM,0)) < 0)   /*abre uma socket
para TCP */
      return(-1);
else {
      if(connect( *soquete, (struct sockaddr *) &serv_addr,
sizeof(serv_addr)) < 0)
            {
            close(*soquete);
            return(-1);
            }
      else return(1);
     }
}


int cliente_conectado(int *newsock)
{
int sockfd, clilen, naoconectado;
struct sockaddr_in cli_addr, serv_addr;
      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
       return(-1);
      /* bind endereco local do programa */
```

```
      bzero((char *) &serv_addr, sizeof(serv_addr));
      serv_addr.sin_family = AF_INET;
      serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
      serv_addr.sin_port = htons(S  ERV_TCP_PORT);
      if (bind(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr))< 0)
       return(-1);
      /* avisa que esta esperando comunicacao */
      listen(sockfd,1);  /*ate uma ao mesmo tempo*/;
      printf("Escutando porta\n");
      naoconectado=TRUE;
      while (naocon ectado)
       {
            clilen=sizeof(cli_addr);
            *newsock=accept(sockfd,(struct sockaddr *) &cli_addr,&clilen);
            if (fd<0)
            {
               printf("\nFalhou accept\n");
               return(-1);
            }
            else {
                  close(sockfd);
                  naoconectado=FALSE;
                  printf("Accept efetuado\n"  );
                 }
       }
}
```

```
/*
      arquivo: trans.h
      uso    : arquivo de cabeçalho
*/
#include "includes.h"
typedef enum {FALSO, VERDADEIRO} boolean;
#define RESPOSTA 1
#define PERGUNTA 0
char final=255,ini_perg=252,ini_resp=253;
struct cab_mens {
      char mtipo;
      long idp;
      int tamanho;
};
void  grava_msg(),       conf_client_msg(),       conf_server_msg(),
      copia(),    limpa();

int   conexao_efetuada(),      xtr(),       xrc(),
     cliente_conectado(),   st_msg_s_out(),   st_msg_in();

long  le_msg(),    le_msg_in();
```

```
 ----------------------------------//-----------------------------------
```

```
/*
      arquivo: rots.c
      uso    : funcoes para suporte geral
*/

#include "trans.h"
#include "param.h"
#include <sys/socket.h>
extern int fd;
char nulo='\0';

void copia( in, out, t )
char *in, *out;
register int t;
{
      while ( t-- > 0 )
            *out++ = *in++;
}

void limpa( vetor )
char *vetor;
{
int j;
      for( j=0 ; j < MAX_LEN_MSG ; j++) *vetor++=nulo;
}
```

```
 ----------------------------------//-----------------------------------
```

```
/*
      arquivo: xtran.c
      uso    : int xtr ( mens_tipo, mens_idp, enviar, tama )
                  mens_tipo = 0=PERGUNTA   1=RESPOSTA
                  mens_idp = numero do processo solicitante
                  enviar = mensagem a ser transmitida
                  tam = tamanho da mensagem
             retorno = 0 = ok, != 0 = erro
*/
#include "param.h"
#include "trans.h"
extern int fd;

int xtr(mens_tipo, mens_idp, enviar,tama)
int mens_tipo,tama;
long mens_idp;
char enviar[MAX_LEN_MSG];
{
```

```
struct cab_mens cabeca[1];
int tam_cab;
if (mens_tipo == 1)
      {
       cabeca[0].mtipo=ini_resp;
      }
else cabeca[0].mtipo=ini_perg;
cabeca[0].idp=mens_idp;
cabeca[0].tamanho=tama;
tam_cab=sizeof(cabeca);
/*tam_mens=strlen(enviar);*/
/*printf("Transmissao tamanho %d\n",tam_mens);*/
/*enviar[tam_mens]=final;*/
/*printf("registro enviar=> %s\n",enviar);*/
write(fd,cabeca,tam_cab);
write(fd,enviar,tama);
return(1);
}
```

```
/*
      arquivo: xrec.c
      uso    : int xrc ( mens_tipo, me  ns_idp, recebido, ta )
                  mens_tipo = 0=PERGUNTA   1=RESPOSTA
                  mens_idp = numero do processo solicitante
                  recebido = mensagem recebida
                  ta = tamanho da mensagem
                  retorno = > 0 - numero de bytes recebidos
                          < 0 - erro
*/
#include "param.h"
#include "tcpdef.h"
#include "trans.h"
extern int fd;

int xrc( mens_tipo, mens_idp, recebido, ta )
int *mens_tipo;
long *mens_idp,*ta;
char recebido[MAX_LEN_MSG];
{
int tam_cab, n=0, rc=0;
char c='\0';
struct cab_mens cabeca;

tam_cab=sizeof(cabeca);
      if (rc=r ecv(fd,&c,1,MSG_PEEK)==1)
      {
            rc=read(fd,&cabeca,tam_cab);
            if (cabeca.mtipo == ini_resp)
                  {
                  *mens_tipo=1;
                  }
            else *mens_tipo=0;
            *mens_idp=cabeca.idp;
```

```
            *ta=cabeca.tamanho;
            while (n < cabeca.tamanho)
            {
              if (rc=read(fd,&c,1)==1)
                  {
                        recebido[n]=c;
                        n++;
                  }
              else return(rc);
            }
            return(n);
      }
}
```

```
/*
      arquivo: msgs.c
      Modulo de aquisicao e controle das mensagens
*/

#include "trans.h"
#include "param.h"

int   msqid_out;
int   msqid_in;
int   msqid_s_out;
int   msqid_s_in;

struct {
      long  mtype;
      char  mtext[MAX_LEN_MSG];
} buffer;


/*
cria e adquire o id das filas de mensagens acessadas pelo agente cliente
*/
void conf_client_msg( conect )
long conect;
{
      msqid_s_out = msgget( id_q_out_server + conect, 0666 | IPC_CREAT );
      if ( msqid_s_out == -1 ) {
            fprintf (stderr, "xm: get_msg_out_server failed\n");
            perror("xm: get_msg_out_server");
            exit(-1);
      }
      msqid_in = msgget( id_q_in_client + conect, 0666 |    IPC_CREAT );
      if ( msqid_in == -1 ) {
            fprintf (stderr, "xm: get_msg_in failed\n");
            perror("xm: get_msg_in");
            exit(-1);
      }
}
```

```
/* cria e adquire o id das filas de mensagens usadas pelo agente servidor*/
void conf_server_msg( conect )
long conect;
{
      msqid_s_in = msgget( id_q_in_server + conect, 0666 | IPC_CREAT );
      if ( msqid_s_in == -1 ) {
            fprintf (stderr, "xm: get_msg_in_server failed\n");
            perror("xm: get_msg_in_server");
            exit(-1);
      }
      msqid_out = msgget( id_q_out_client + conect, 0666 | IPC_CR   EAT );
      if ( msqid_out == -1 ) {
            fprintf (stderr, "xm: get_msg_out failed\n");
            perror("xm: get_msg_out");
            exit(-1);
      }
}


/* pega o status da fila de mensagens msqid_s_out
   e devolve o numero de mensagens
*/
int st_msg_s_out()
{
struct msqid_ds msqid_ds;
int ret;
      ret = msgctl( msqid_s_out, IPC_STAT, &msqid_ds );
      if (ret == -1) {
            perror("xm: st_msg_s_out");
            exit(-1);
      }
      return (msqid_ds.msg_qnum);          /* nro msg na fila */
}


/* pega o status da fila de mensagens msqid_in
   e devolve o numero de mensagens
*/
int st_msg_in()
{
struct msqid_ds msqid_ds;
int ret;
/*
struct msg      *msg_p;
mtyp_t      tipo;
*/
      ret = msgctl( msqid_in, IPC_STAT, &msqid_ds );
      if (ret == -1) {
            perror("xm: st_msg_in");
            exit(-1);
      }
      return (msqid_ds.msg_qnum);          /* nro  msg na fila */

}



/*
      coloca uma mensagem na fila
      tipo = na requisicao = numero do processo
```

```
             na resposta = tipo da mensagem recebido na leitura
*/
void grava_msg( fila, msg, tam, tipo )
int fila;
char *msg;
long tam;
long tipo;
{
int ret;
      buffer.mtype = tipo;
      memcpy( buffer.mtext, msg, (int)tam );
      ret = msgsnd( fila, &buffer, (int)tam, 0 /*wait*/ );
      if (ret == -1) {
            perror("xm: w_msg");
            exit(-1);
      }
}


/*
      Le uma mensagem por tipo
      Se tipo = 0, pega proxima da fila
      devolve a mensagem e m msg, tamanho em tam (se foi passado),
      e retorna o tipo
*/
long le_msg( fila, msg, tipo, tam )
int fila;
char *msg;
long tipo, *tam;
{
int ret;

      ret = msgrcv ( fila, &buffer, sizeof(buffer.mtext), (int)tipo, 0 );
      if ( ret == -1 ) {
            perror("xm: r_msg" );
            exit(-1);
      }
      memcpy ( msg, buffer.mtext, ret );
      if ( tam != (long *)0 )
            *tam = (long)ret;
      return( buffer.mtype );
}


/*
--> leitura da fila msqid_in
      Le prox mensagem da file msqid_in
      devolve a mensagem em msg, tamanho em tam (se foi passado),
      e retorna o tipo, ou -1 se nao houver mensagem.
*/
long le_msg_in( msg, tam )
char *msg;
long *tam;
{
struct msqid_ds msqid_ds;
int ret;

      ret = msgctl( msqid_in, IPC_STAT, &msqid_ds );
      if (ret == -1) {
            perror("xm: st_msg");
            exit(-1);
```

```
      }
      if (msqid_d s.msg_qnum <= 0)          /* nro msg na fila */
            return(-1);

      ret = msgrcv(msqid_in, &buffer, sizeof(buffer.mtext), 0L,
IPC_NOWAIT);
      if ( ret == -1)
            return(-1);        /* nao estava mais la' */

      memcpy ( msg, buffer.mtext, ret );

      if ( tam != (long *)0 )
            *tam = (long)ret;

      return( buffer.mtype );
}
```

```
/*
      arquivo: client.c
      uso    : Programa Agente cliente
*/


#include "param.h"
#include "trans.h"
extern int fd;
extern int msqid_in, msqid_out;
extern int msqid_s_in, msqid_s_out;
int fd;

void main( argc, argv )
int argc;
char *argv[];
{
char envia[MAX_LEN_MSG];
long tam,m_pid,conect;
      if (argc != 2)
      {
            printf("\nUso: client conexao\n");
            exit(1);
      }

      /* abre um socket TCP e con  ecta ao servidor */
      if(conexao_efetuada(&fd) == -1) exit(-1);
      printf("Abriu socket e conectou com servidor\n");
      conect = atol(argv[1]);

      /* configura fila de mensagens dos clientes */
      conf_client_msg( conect );
      limpa(&envia);

      while (1)
      {
```

```
 /*checa presença de mensagem na fila de solicitações dos clientes COBOL*/
            if ( st_msg_in() )
                  {
                        limpa(&envia); /* limpa a string envia */

                        /* le a solicitação e o número do processo na */
                        /* fila de solicitações dos clientes COBOL    */
                        m_pid=l e_msg(msqid_in,envia,0L, &tam);

                        /* envia a solicitação para o agente servidor */
                        xtr(PERGUNTA, m_pid, envia, tam);
                  }

  /*checa presença de mensagem na fila de respostas do servidor COBOL*/
            if ( st_msg_s_out() )
                  {
                        limpa(&envia); /* lim pa a string envia */

                        /* le a resposta e o número do processo na */
                        /* fila de respostas do servidor COBOL     */
                        m_pid=le_msg(msqid_s_out,envia,0L, &tam);

                        /* envia a resposta para o agente servidor */
                        xtr(RESPOSTA, m_pid, envia, tam);
                  }
      }
}



  ----------------------------------//-------------------------------------



/*
      arquivo: server.c
      uso    : Programa Agente servidor
*/

#include "param.h"
#include "trans.h"
extern int fd;
extern int msqid_in, msqid_out;
extern int msqid_s_in, msqid_s_out;
int fd;


void main( argc, argv )
int argc;
char *argv[];
{
char recebe[MAX_LEN_MSG];
int trc;
long tam,m_pid,conect,tipo;
      if (argc != 2)
      {
            printf("\nUso: server conexao\n");
            exit(1);
      }
```

```
      /* abre uma socket para TCP e espera conex   ao cliente */
      if(cliente_conectado(&fd) == -1) exit(-1);
      printf("Cliente conectado ao servidor\n");
      conect = atol(argv[1]);

      /* configura fila de mensagens do servidor */
      conf_server_msg( conect );

      while (1)
      {
            tipo=3;
            m_pid=0;
            limpa(&recebe);  /* limpa string recebe */

            /* espera o recebimento de uma mensagem do */
            /* agente cliente pelo socket conectado */
            if ( trc=xrc(&tipo,&m_pid,&recebe,&tam) > 0)
                  {
                   /* se recebeu mensagem então verifica o tipo */

                   /* se for pergunta grava m  ensagem na fila */
                   /* de solicitações do servidor COBOL */
                   if (tipo == PERGUNTA) grava_msg(msqid_s_in, recebe, tam,
m_pid);

                   /* se for resposta grava mensagem na fila */
                   /* de respostas dos clientes COBOL        */
                   if (tipo == RESPOSTA)  grava_msg(msqid_out, recebe, tam,
m_pid);
                  }
      }
}
```

```
/*                      Integração com o COBOL
      arquivo: sub.c  --->  faz parte do programa de runtime do RM/Cobol
      As seguintes alterações  são feitas para integrar as funções do
middleware (subscobol.c) ao runtime do RM/Cobol
*/
extern int   micro();
extern int   pega_msg();
extern int   devolve_msg();

struct PROCTABLE LIBTABLE[] =
{
    {"SYSTEM",  subsys},
    ...
    {"micro",            micro},
    {"pega_msg",   pega_msg},
    {"devolve_msg",      devolve_msg},
    ...
    {0, 0}
};
```

```
 ----------------------------------//-------------------------------------


/*                        Integração com o COBOL
      arquivo: subscobol.c
      Modulo de aquisicao e controle das mensagens
      ----   incluido no runtime do RM/Cobol  ----
      deve-se gerar o objeto (subscobol.o) desse arquivo e adiciona-lo
      ao runtime do RM/Cobol (runcobol) através da linha de comando:
            make runcobol CLIBRARY=”sub.o subscobol.o”
*/

#include "includes.h"
#include "param.h"

#ifdef RM
#include "/usr/rmcobrts/rmc85cal.h"
#endif

int   __status;
long  g_tipo;           /* pid do processo cliente */

struct {
      long  mtype;
      char  mtext[MAX_LEN_MSG];
} buffer;


/*
      Le uma mensagem por tipo
      Se tipo = 0, pega proxima da fila
      devolve a mensagem em msg, tamanho em tam (se foi passado),
      e retorna o tipo
*/

long r_msg( fila, msg, tipo, tam )
int fila;
char *msg;
int  *tam;
long tipo;
{
int ret;
      ret = msgrcv ( fila, &buffer, sizeof(buffer.mtext), tipo, 0 );
      if ( ret == -1 ) {
            __status = errno;
            return(-1);
      }
      memcpy ( msg, buffer.mtext, ret );
      if ( tam != (int *)0 )
            *tam = ret;
      return( buffer.mtype );
}

/*
      coloca uma mensagem na fila
```

```
      tipo = na requisicao = numero do processo
             na resposta = tipo da mensagem recebido n   a leitura

*/

int w_msg( fila, msg, tam, tipo )
int fila;
char *msg;
long tam;
long tipo;
{
      int ret;
      buffer.mtype = tipo;
      memcpy( buffer.mtext, msg, (int)tam );
      ret = msgsnd( fila, &buffer, (int)tam, 0 /*wait*/ );
      if (ret == -1) {
            __status = errno ;
            return(-1);
      }
}


int pega_msg ( name, arg_count, arg_vector, initial )
char *name;
int arg_count;
struct ARGUMENT_ENTRY arg_vector[];
int initial;
{
int   msqid_s_in;
static char *ch;
static long *status, *conect;
      ch = arg_vector[0].a_address;
      status  = arg_vector[1].a_address;
      conect = arg_vector[2].a_address;
/*    printf("numero da conexao %d \n",*conect);*/
      msqid_s_in = msgget( id_q_in_server + *conect, 0 );
      if ( msqid_s_in == -1 ) {
            fprintf (stderr, "xm: getm_in_server failed\n");
            perror("xm:  getm_in_server");
            exit(-1);
            }
      __status = 0;
      /* envia a chave*/
      g_tipo = r_msg ( msqid_s_in, ch, 0L, (int *)0 );
/*    printf("Tipo pega_msg => %d \n", g_tipo);*/
      *status = __status;
      return(0);
}

int devolve_msg ( name, arg_count, arg_vector, initial )
char *name;
int arg_count;
struct ARGUMENT_ENTRY arg_vector[];
int initial;
{
int   msqid_s_out;
static char *reg;
static long *tam;
static long *status, *conect;
      reg = arg_vector[0].a_address;
      tam = arg_vector[1].a_address;
```

```
	status = arg_vector[2].a_ad  dress;
	conect = arg_vector[3].a_address;
	msqid_s_out = msgget( id_q_out_server + *conect, 0 );
	if ( msqid_s_out == -1 ) {
		fprintf (stderr, "xm: getm_out_server failed\n");
		perror("xm: getm_out_server");
		exit(-1);
		}
	__status = 0;
	/* devolve o r egistro */
/*	printf("Tipo pega_msg => %d \n", g_tipo);*/
	w_msg ( msqid_s_out, reg, *tam, g_tipo );
	*status = __status;
	return(0);
}

int micro ( name, arg_count, arg_vector, initial )
char *name;
int arg_count;
struct ARGUMENT_ENTRY arg_vector[];
int initial;
{
int   msqid_out;
int   msqid_in;
static char *ch, *reg;
static long *tam_ch, *status, *conect;
static long tipo;

	ch = arg_vector[0].a_address;
	tam_ch = arg_vector[1].a_address;
	reg = arg_vector[2].a_address;
	status = arg_vector[3].a_address;
	conect = arg_vector[4].a_address;

	if ( initial == 0 ) tipo = getpid();
/*	printf("numero conexao %d \n",*conect); */
	msqid_in = msgget( id_q_in_client + *conect, 0);
	if ( msqid_in == -1 ) {
		fprintf (stderr, "xm: getm_in_client failed\n");
		perror("xm:  getm_in_client");
		exit(-1);
		}
	msqid_out = msgget( id_q_out_client + *conect, 0);
	if ( msqid_out == -1 ) {
		fprintf (stderr, "xm: getm_out_client failed\n");
		perror("xm: getm_out_client");
		exit(-1);
		}
	/* envia a chave */

	__status = 0;
	if (w_msg ( msqid_in, ch, *tam_ch, tipo ) == -1 )
	{
		*status = __status;
		return(0);
	}
	/* espera o retorno */
	r_msg ( msqid_out, reg , tipo , (int *)0 );
	*status = __status;
	return(0);
}
```

Referências Bibliográficas

[1] COMER, Douglas E., *Interligação em Rede com TCP/IP - Volume 1 - Princípios, protocolos e arquitetura.* 3ª Edição – Rio de Janeiro: Campus, 1998.

[2] Cyclades Brasil. *Guia Internet de Conectividade*. 4ª Edição, 1997.

[3] INTERNET, Acucorp Inc. – *The COBOL Market*. [http://www.acucorp.com/cobolmarket.html]

[4] INTERNET, Alexandre S. Pastorino e João L. Lopes – Pós-Graduação em Informática na UCPel-Universidade Católica de Pelotas – *A Proposta Cliente/Servidor e a Informatização da UFPel*. [http://esin.ucpel.tche.br/bbvirt/pos/ufpel.htm]

[5] INTERNET, Cisco Sytems Inc. – *Glossary of Terms*. [http://www.cisco.com/univercd/cc/td/doc/product/lan/trsrb/glossary.htm]

[6] INTERNET, George Schussel at DCI – *Client/Server: Past, Present and Future*. [http://news.dci.com/geos/dbsejava.htm]

[7] INTERNET, Grupo de Redes da Universidade Federal do Rio Grande do Sul – *Sistemas Cliente/Servidor*. [http://penta.ufrgs.br/redes296/cliente_ser/tutorial.htm]

[8] INTERNET. Itec - *Cliente/Servidor um Modelo de Equilíbrio*. [http://www.itec.com.br/espcliserv.html]

[9] INTERNET, Object Ideas Corporation – *Client/Server Overview (Tutorial)*. [http://www.object-ideas.com/clntsrv/]

[10] INTERNET, Oracle Corporation – *Client / Server Computing*. [http://www.oracle.com/products/tools/dev2k/collateral/cscomputing/]

[11] INTERNET, PC Webopaedia – [http://www.pcwebopaedia.com/]

[12] INTERNET, Carlos A. Lima, Henrique Rolfsen Francisco e Luís C. Quintela – *Viabilizando o Uso do Paradigma da Orientação a Objetos em Ambientes Distribuídos*. [http://www.rerum.com/brasil/artigos/artViabilizando.htm]

[13] INTERNET, Software Engineering Institute at Carnegie Mellon University – *Technology Descriptions*. [http://www.sei.cmu.edu/activities/str/descriptions]

[14] INTERNET, Usenet – comp.client-server – *Client/Server Frequently Asked Questions.* [ftp://rtfm.mit.edu/pub/usenet-by-group/news.answers/client-server-faq]

[15] ORFALI, Robert, *Cliente/Servidor: Guia essencial de sobrevivência*. / Robert Orfali, Dan Harkey, Jeri Edwards. – Rio de Janeiro: IBPI Press/Infobook, 1996.

[16] REVISTA, Carlos Machado e Sônia Penteado - *É Hora de Recomeçar*. Informática Exame. Março de 1994 p. 64-71.

[17] SENGER, Luciano José, *Avaliação de desempenho do PVM-W95*. Dissertação, ICMSC, USP - São Carlos, 1997.

[18] SMITH, Patrick N., *Client / Server Computing* / Patrick Smith, Steve Guengerich – 2ª edição: Sams Publishing, 1994.

[19] SOARES, Luiz Fernando Gomes, *Redes de Computadores: das LANs, MANs e WANs às redes ATM*. / Luiz Fernando G. Soares, Guido Lemos, Sérgio Colcher. - Rio de Janeiro: Campus, 1995.

[20] SOUZA, Márcio Augusto, *Estudo e Utilização da Plataforma MPI*. Mini-dissertação, ICMSC, USP - São Carlos, 1995.

[21] TANENBAUM, Andrew S., *Sistemas Operacionais Modernos*. – Rio de Janeiro: Prentice Hall do Brasil, 1995.