UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ENGENHARIA DE SÃO CARLOS

Departamento de Engenharia Mecânica
Departamento de Engenharia Elétrica

PROTOCOLO DE COMUNICAÇÃO PARA CONTROLE DE MÃO ROBÓTICA ANTROPOMÓRFICA

Italo Martins Neto
Orientador: Prof. Dr. Adilson Gonzaga

Novembro / 2010

FOLHA DE AVALIAÇÃO

Candidato(s): Italo Martins Neto

Título: PROTOCOLO DE COMUNICAÇÃO PARA CONTROLE DE MÃO ROBÓTICA ANTROPOMÓRFICA

Trabalho de Conclusão de Curso apresentado à Escola de Engenharia de São Carlos da Universidade de São Paulo Curso de Engenharia Mecatrônica

BANCA EXAMINADORA

Prof. Dr. Adriano Siqueira

Nota atribuída: 9,0 (nove) ______ (assinatura)

Prof. Dr. Daniel Varela

Nota atribuída: 9,0 (NOVE) ______ (assinatura)

Prof. Dr. Adilson Gonzaga (orientador)

Nota atribuída: 10,0 (DEZ) ______ (assinatura)

Média: 9,3 (NOVE VIRGULA TRES)

Resultado: APROVADO

Data: 11/11/2010

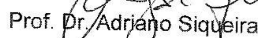
ATA DA DEFESA DE TCC

Candidato(s): Italo Martins Neto

Título: PROTOCOLO DE COMUNICAÇÃO PARA CONTROLE DE MÃO ROBÓTICA ANTROPOMÓRFICA

CONSIDERAÇÕES SOBRE O TRABALHO
- O TRABALHO FOI BEM APRESENTADO E O ALUNO RESPONDEU A TODAS AS QUESTÕES DA BANCA. - O TRABALHO EM SI FOI CONSIDERADO DE GRANDE IMPORTÂNCIA PARA A ÁREA.

LISTA DE CORREÇÕES NECESSÁRIAS PARA A VERSÃO FINAL
- ALGUMAS CORREÇÕES FORAM PASSADAS AO CANDIDATO. SERÃO FEITAS ANTES DA IMPRESSÃO FINAL.



Prof. Dr. Adriano Siqueira

Prof. Dr. Daniel Varela

Prof. Dr. Adilson Gonzaga

Data: 11/11/2010

Italo Martins Neto

PROTOCOLO DE COMUNICAÇÃO PARA CONTROLE DE MÃO ROBÓTICA ANTROPOMÓRFICA

Trabalho de Conclusão de Curso apresentado à Escola de Engenharia de São Carlos da Universidade de São Paulo, para obtenção do título de Graduação em Engenharia Mecatrônica

Orientador: Prof. Dr. Adilson Gonzaga

São Carlos

Novembro / 2010

AUTORIZO A REPRODUÇÃO E DIVULGAÇÃO TOTAL OU PARCIAL DESTE TRABALHO, POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO, PARA FINS DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

Ficha catalográfica preparada pela Seção de Tratamento
da Informação do Serviço de Biblioteca – EESC/USP

M386p

Martins Neto, Italo
Protocolo de comunicação para controle de mão robótica antropomórfica / Ítalo Martins Neto ; orientador Adilson Gonzaga. -- São Carlos, 2010.

Trabalho de Conclusão de Curso (Graduação em Engenharia Mecatrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2010.

1. Redes de computadores. 2. Robótica. 3. Protocolo de comunicação. 4. Reconhecimento. I. Título.

Resumo

NETO, I. M. Protocolo de Comunicação para controle de mão robótica antropomórfica. Trabalho de Conclusão de Curso – Escola de Engenharia de São Carlos – Universidade de São Paulo (USP), 2010.

Este trabalho, desenvolvido em conjunto com Laboratório de Visão Computacional e Laboratório de Mecatrônica tem o intuito de realizar o comando de uma mão robótica antropomórfica através de redes de computador, com parâmetros extraídos de gestos humanos.

Para isso, há a implementação de um protocolo de comunicação na camada de aplicação, baseado em TCP/IP nas camadas anteriores, pois o controle é feito via web.

Para estabelecer a conexão entre os módulos utilizou-se uma estrutura cliente/servidor implementada em linguagem C. Isso possibilitou o envio de mensagens no formato string para os extremos da rede, as quais contém os parâmetros de controle de posição codificadas através do protocolo proposto.

Palavras chave: rede de computadores - robótica - protocolo - reconhecimento

Abstract

This graduation assignment, developed together with the Computational Vision Lab and Mechatronics Laboratory is intended to accomplish the command of an anthropomorphic robotic hand through computer networks, with parameters extracted from human gestures. This calls for implementing a communication protocol at the application layer based on TCP/IP in the previous layers, as the control is web based. To establish the connection between the modules it was used a struct client/server implemented in C. That enables the sending of messages in the string format through the network, which contains the control parameters of position coded by the proposed protocol.

Keywords: computer network - robotic - protocol - recognition

Lista de Figuras

Figura 1 - Ilustração representando a reprodução dos gestos 13
Figura 2 - Componentes da mão robótica 14
Figura 3 - WebCam para captura dos gestos 14
Figura 4 - Cartão perfurado, primeiro meio de transmissão de dados 18
Figura 5 - Arpanet em 1973 19
Figura 6 - Primeiro esboço do padrão Ethernet 20
Figura 7 - Xerox Alto (1973), a primeira estação de trabalho e também a primeira a ser ligada em rede 21
Figura 8 - Cabo thicknet e esquema de ligação 23
Figura 9 - Cabos coaxiais com o conector em "T" 24
Figura 10 - Terminal do cabo coaxial 24
Figura 11 - Esquema de ligação, mostrando a terminação. 25
Figura 12 - Conectores BNC 26
Figura 13 - Placa ISA, com conector BNC e conector RJ45 26
Figura 14 - Cabo coaxial, com blindagem 27
Figura 15 - Conector RJ45 e cabos de par trançado 28
Figura 16 - Backbones de fibra óptica interligando países da Ásia 29
Figura 17 - Cabos de fibra óptica multimodo 30
Figura 18 - Cabeçalho do protocolo IP 34
Figura 19 - Campo TOS (Type of service) 35
Figura 20 - Esquema mostrando fragmentação de datagramas em redes com diferentes MTU's 36
Figura 21 - Datagrama Fragmentado 37
Figura 22 - Cabeçalho do protocolo TCP 40
Figura 23 - Esquema de conexão socket 43
Figura 24 - Menu de opções do compilador do software DEV C++ 44
Figura 25 - Fluxograma de comandos 51
Figura 26 - Interface de comunicação do programa implementado em socket 52
Figura 27 - Interface do GRASPIT 57
Figura 28 - Reprodução de gesto 58
Figura 29 - Posição Inicial e Final 60
Figura 30 - Imagem com a mão real mostrando a convenção dos ângulos 60
Figura 31 - Comandos GRASPIT 61
Figura 32 - Menu que contém S-function 62
Figura 33 - Procedimento para utilizar S-Function 63
Figura 34 - Exemplo de Passagem de Parâmetros 64
Figura 35 - Estágios de Simulação Simulink 65
Figura 36 - Fluxograma de Blocos de transmissão 68
Figura 37 - Implementação do Cliente 69
Figura 38 - Implementação do Servidor 69
Figura 39 - Resultado da Implementação 70
Figura 40 - Pontos de Roteamento do IP remoto 72
Figura 41 - Tempo(ms) para 1000 bytes 73
Figura 42 - Erro (%) x Delay 74
Figura 43 - Tempo de recebimento das Mensagens 74
Figura 44 - Tempo de Resposta (10 bytes) 75
Figura 45 - Tempo de Resposta (50 bytes) 76

Figura 46 - Tempo de Resposta (80 bytes) 76
Figura 47 - Tempo de Resposta (100 bytes) 77
Figura 48 - Tempo de Resposta (150 bytes) 77
Figura 49 - Tempo de Resposta (250 bytes) 78
Figura 50 - Tempo de Resposta (500 bytes) 78
Figura 51 - Tempo de Resposta (1000 bytes) 79
Figura 52 - Tempo de Resposta (2000 bytes) 79
Figura 53 - Tempo Médio de Resposta x Tamanho do Pacote 80
Figura 54 - Erro(%) x Delay 80
Figura 55 - – Tempo de Recebimento das Mensagens 81
Figura 56 - Tempo de Resposta (10 bytes) 81
Figura 57 - Tempo de Resposta (50 bytes) 82
Figura 58 - Tempo de Resposta (80 bytes) 82
Figura 59 - Tempo de Resposta (100 bytes) 82
Figura 60 - Tempo de Resposta (150 bytes) 83
Figura 61 - Tempo de Resposta (250 bytes) 83
Figura 62 - Tempo de Resposta (500 bytes) 84
Figura 63 - Tempo de Resposta (1000 bytes) 84
Figura 64 - - Tempo Médio de Resposta x Tamanho do Pacote 85
Figura 65 - Erro(%) x Delay (ms) 85
Figura 66 - Tempo de Recebimento das Mensagens 86
Figura 67 - Erro (%) para diferentes delays e Redes 87
Figura 68 - Tempo de Resposta x tamanho do pacote (bytes) 87
Figura 69 - Tempo de Processamento MATLAB 88

Índice de Abreviações

ARP - Address Resolution Protocol
ASA - American Standard Association
ASCII - American Standard Code for Information
ATM - Asynchronous Transfer Mode
AUI - Attachment Unit Interface
BNC - Bayonet Neill-Concelman
DCCP - Datagram Congestion Control Protocol
DNS - Domain Name System
DSL - Digital Subscriber Line
FDDI - Fiber Distributed Data Interface
FTP - File Transfer Protocol
HDLC - High-Level Data Link
ICMP - Internet Control Message Protocol
IETF - Internet Engineering Task Force
IP – Internet Protocol
ISA - Industry Standard Architecture
ISDN - Integrated Services Digital Network
ISO - International Organization for Standardization
MAC - Media Access Control
MTU - Maximum transfer unit
OSI - Open Systems Interconnection
PPP - Point-to-Point Protocol
RARP - Reverse Address Resolution Protocol
SCTP - Stream Control Transmission Protocol
SONET - Synchronous Optical Networking
TCP - Transmission Control Protocol
TTL - Time-to-live
UDP - User datagram protocol

Sumário

1.Introdução 13
2.Objetivos 15
3. Fundamentação Teórica 16
3.1. Histórico 16
3.1.1. Protocolos 16
3.1.2. Redes 17
3.1.3. Evolução do Cabeamento 22
3.1.4. Redes Wireless 30
3.2. Camadas de Rede 32
3.2.1. Camada Física 32
3.2.2 Camada de enlace 32
3.2.3 Camada de Rede 32
3.2.4. Camada de Transporte 39
3.2.5. Camada de Sessão 42
3.2.6. Camada de Apresentação 42
3.2.7. Camada de Aplicação 42
4. Metodologia 42
4.1 Comunicação em socket 42
4.2 Utilização no projeto 43
4.4 Concatenar e Desmembrar Dados 53
4.4.1 Concatenar 53
4.4.2 Desmembrar 54
5. Simulador 55
5.1 getRobotName 58
5.2 getDOFVals 58
5.3 moveDOFs 59
6. S-Function 62
6.1 Usando S-Functions em Modelos 63
6.2 Passando Parametros para S-functions 63
6.3 Como S-functions Funcionam 64
6.4 Matemática dos Blocos Simulink 64
6.5. Estágios de Simulação 65

6.6 M-File 65
6.7 Tempo de Amostragem e Offsets 66
6.8 Compilação 67
7. Implementação 68
8. Testes de Desempenho 71
8.1. Metodologia 71
8.1.1.Computador Local 71
8.1.2. Rede Local 72
8.1.3. Conexão Remota 72
9. Resultados 73
9.1 Computador Local 73
9.1 - Rede Local 75
9.3 Computador Remoto 81
10. Conclusão 87
11. Referências Bibliográficas 89
12. Apêndice 90

1.Introdução

Com os avanços tecnológicos do século XXI, principalmente na área da robótica, máquinas têm ajudado cada vez mais o homem na execução de suas tarefas. Porém, alguns mecanismos ainda são ineficientes, seja pela complexidade mecânica, seja pela dificuldade de controle, sendo agravadas quando trata-se de manipuladores. Uma abordagem para resolver esse problema, é basear-se na mais perfeita máquina existente no mundo, o corpo humano. Um protótipo de manipulador, com base nesse conceito, seria uma mão robótica, aliado ao controle pela visão, que, por ter um grande processamento, é chamada de visão computacional.

A partir desses conceitos surgiu o projeto Kanguera, que tem como objetivo a execução de gestos com uma mão robótica antropomórfica (Figura 1). Aliado a isso, visa também a identificação de gestos humanos por uma câmera remota, que retirando alguns parâmetros da imagem, pode reproduzir a mão humana a quilômetros de distância. Porém, para que isso ocorra, é necessário a criação de uma rede para interligação dos dois extremos, e inserido nesse contexto também a criação de um protocolo para gerenciar a troca de informações.

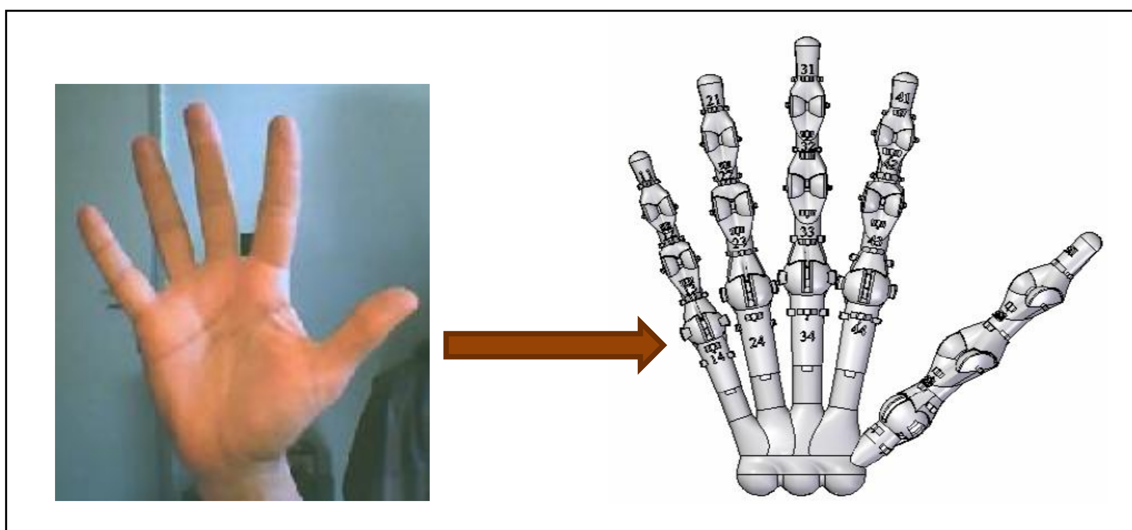


Figura 1 - Ilustração representando a reprodução dos gestos

A mão robótica antropomórfica Kangûera é uma versão artificial de uma mão humana direita de um homem com tamanho aproximado de 50% superior àquele apresentado pela média dos exemplares humanos. Especificamente, todos os dedos da mão robótica são formados pelas falanges proximal, medial e distal, juntas articuláveis e metacarpos, como ocorre com a estrutura de uma mão humana. Os cinco dedos são articuláveis com movimentos totalmente independentes e foram construídos a partir de uma resina industrial, permitindo a reprodução das configurações funcionais da mão humana. A Figura 2 ilustra os principais componentes da mão robótica Kangûera [1].

Figura 2 - Componentes da mão robótica

A extração dos parâmetros de controle da mão robótica é feita pela análise dos padrões de pixels, que, geometricamente, determinam o cálculo dos ângulos entre as junções da mão.

Figura 3 - WebCam para captura dos gestos

2.Objetivos

2.1 Objetivos Gerais

O objetivo geral do projeto é interligar os módulos do projeto Kanguera, a fim de estabelecer uma conexão estável e eficiente.

2.2 Objetivos Específicos

O objetivo específico do trabalho é a criação e teste de um protocolo de comunicação para a aplicação proposta.

3. Fundamentação Teórica

3.1. Histórico

3.1.1. Protocolos

Um protocolo de comunicação nada mais é do que um conjunto de convenções que rege o tratamento e, especialmente, a formatação dos dados num sistema de comunicação. São conhecidos vários protocolos e se faz uso deles diariamente. O mais antigo deles é a língua falada: duas pessoas que emitem sons audíveis aos ouvidos humanos podem se comunicar. Neste exemplo, o protocolo de comunicação é a emissão de sons numa dada faixa de frequência, o código utilizado é a língua falada e a mensagem é o conteúdo do que se fala. Dessa forma, podemos classificar da seguinte forma os principais itens para um sistema de comunicação:

- Protocolo de Comunicação: convenção na formatação dos dados
- Código de Comunicação: convenção dos símbolos usados
- Mensagem: conteúdo do que se transmite e recebe

Os mais diversos meios podem ser utilizados para criar códigos de comunicação: luz, gestos, sons e símbolos são alguns deles. Em se tratando de máquinas, o meio mais utilizado até hoje é o elétrico.

O precursor de todos os dispositivos de comunicação, que faz uso desse meio, é o telégrafo.

O premiado pintor norte-americano Samuel Finley, que viveu de 1791 a 1872, inventou um aparelho que servia para enviar e receber sinais elétricos através de fios. Usou um código de sinais que representavam as letras do alfabeto e os números. Este pintor também era conhecido como Morse.

Morse estabeleceu as técnicas essenciais para a transmissão de dados através de fios e, em 1844, inaugurou a primeira linha telegráfica enviando a famosa mensagem *What hath God Wrought?*. Em 1866, graças à tecnologia de cabos submarinos desenvolvida por Werner von Siemens, foi instalado o primeiro cabo transatlântico ligando os EUA à França, que passou a ser largamente utilizado na época. Parte da terminologia comum da transmissão de dados moderna origina-se diretamente dessas primeiras experiências.

A forma inicial da telegrafia, a primeira forma de enviar mensagens elétricas, usava a atuação remota de relês elétricos (eletroímãs) para deixar marcas numa tira de papel. Originalmente Morse imaginou numerar todas as palavras e em transmitir seus números através do telégrafo. O receptor, usando um enorme "dicionário", decifraria a mensagem.

Verificando a ineficiência desse modelo de codificação, foi Alfred Vail, um dos seus assistentes e o verdadeiro autor do chamado "Código Morse" que define as letras do alfabeto pelo padrão "ponto e traço" ou "som curto e som longo".

Este novo código reconhecia quatro estados: tensão-ligada longa (traço), tensão-ligada curta (ponto), tensão-desligada longa (espaço entre caracteres e palavras) e tensão-desligada curta (espaço entre pontos e traços).

Podemos traduzir os termos acima utilizados para os dias de hoje como condições binárias de "1" (ponto) e "0" (traço). Além disso, o alfabeto Morse é um código baseado em 5 posições, ou seja, não são necessárias mais do que 5 posições para que todas as letras e números sejam padronizados. Por este motivo, o Código Morse é classificado como um protocolo de 5 bits.

Uma particularidade do alfabeto Morse é que a maioria das letras não usam os 5 bits. A letra "E", por exemplo, é expressa por um bit único. Porém, há um ponto a se considerar: o número de combinações possíveis para 2 símbolos e 5 posições é de apenas 32 (2 à quinta potência), o que não permite codificar todos os símbolos necessários (caracteres, algarismos, sinais gráficos).

Um dos primeiros a perceber esta limitação foi o francês Baudot. Ele resolveu este impasse criando o *Código de Baudot*, usado na telegrafia e nas máquinas de transmissão de dados que sucederam o telégrafo.

Depois da passagem por várias evoluções do protocolo, em 1963 definiu-se o código ASCII. O ASCII, American Standart Code for Information Interchange - Código Padrão Americano para Troca de Informações, foi estabelecido graças ao esforço conjugado do comitê X3.4 da ASA (American Standart Association) e da indústria americana de computadores e comunicação de dados. Participaram a IBM, a AT&T e sua subsidiária Teletype Corporation.

O ASCII-1963, antes conhecido como padrão ASA X3.4-1967 (depois USASA, e mais tarde ANSI), faz parte de uma série de padrões; outros especificam como armazenar ASCII em cartões perfurados, fitas magnéticas e como manipular erros.

3.1.2. Redes

Ao longo das décadas, ocorreu a evolução dos protocolos de comunicação baseados em sinais elétricos, e nesse contexto, surgiram as redes de computadores.

As primeiras redes de computadores foram criadas ainda durante a década de 60, como uma forma de transferir informações de um computador a outro. Na época, o meio mais usado para armazenamento externo de dados e transporte ainda eram os cartões perfurados (Figura 4), que armazenavam poucas dezenas de caracteres cada (o formato usado pela IBM, por exemplo, permitia

armazenar 80 caracteres por cartão). Eles são uma das formas mais ineficientes de transportar grandes quantidades de informação.

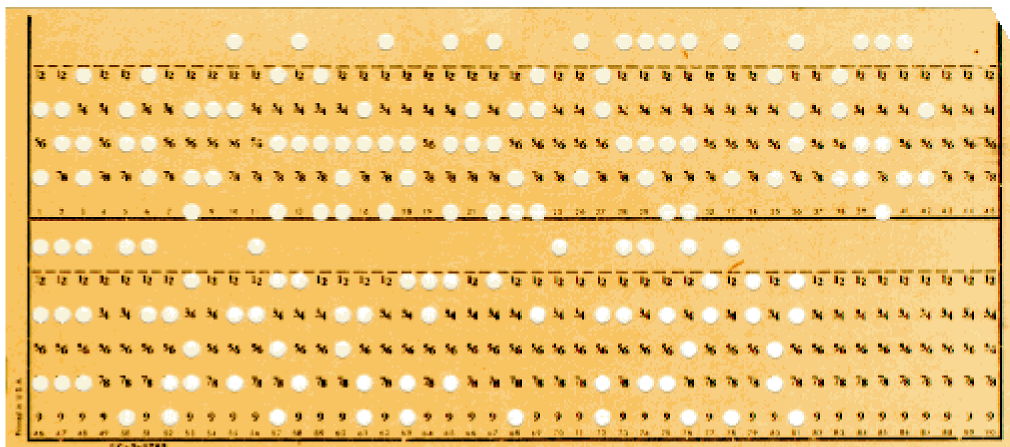


Figura 4 - Cartão perfurado, primeiro meio de transmissão de dados

De 1969 a 1972 foi criada a Arpanet, o embrião da Internet que conhecemos hoje. A rede entrou no ar em dezembro de 1969, inicialmente com apenas 4 nós, que respondiam pelos nomes SRI, UCLA, UCSB e UTAH e eram sediados, respectivamente, no Stanford Research Institute, na Universidade da California, na Universidade de Santa Barbara e na Universidade de Utah, nos EUA. Eles eram interligados através de links de 50 kbps, criados usando linhas telefônicas dedicadas, adaptadas para o uso como link de dados.

Esta rede inicial foi criada com propósitos de teste, com o desafio de interligar 4 computadores de arquiteturas diferentes, mas a rede cresceu rapidamente e em 1973 já interligava 30 instituições, incluindo universidades, instituições militares e empresas. Para garantir a operação da rede, cada nó era interligado a pelo menos dois outros (com exceção dos casos em que isso realmente não era possível) , de forma que a rede pudesse continuar funcionando mesmo com a interrupção de várias das conexões.

As mensagens eram roteadas entre os nós e eventuais interrupções nos links eram detectadas rapidamente, de forma que a rede era bastante confiável. Enquanto existisse pelo menos um caminho possível, os pacotes eram roteados até finalmente chegarem ao destino, de forma muito similar ao que tem-se hoje na Internet.

A Figura 5 mostra o diagrama da Arpanet em 1973:

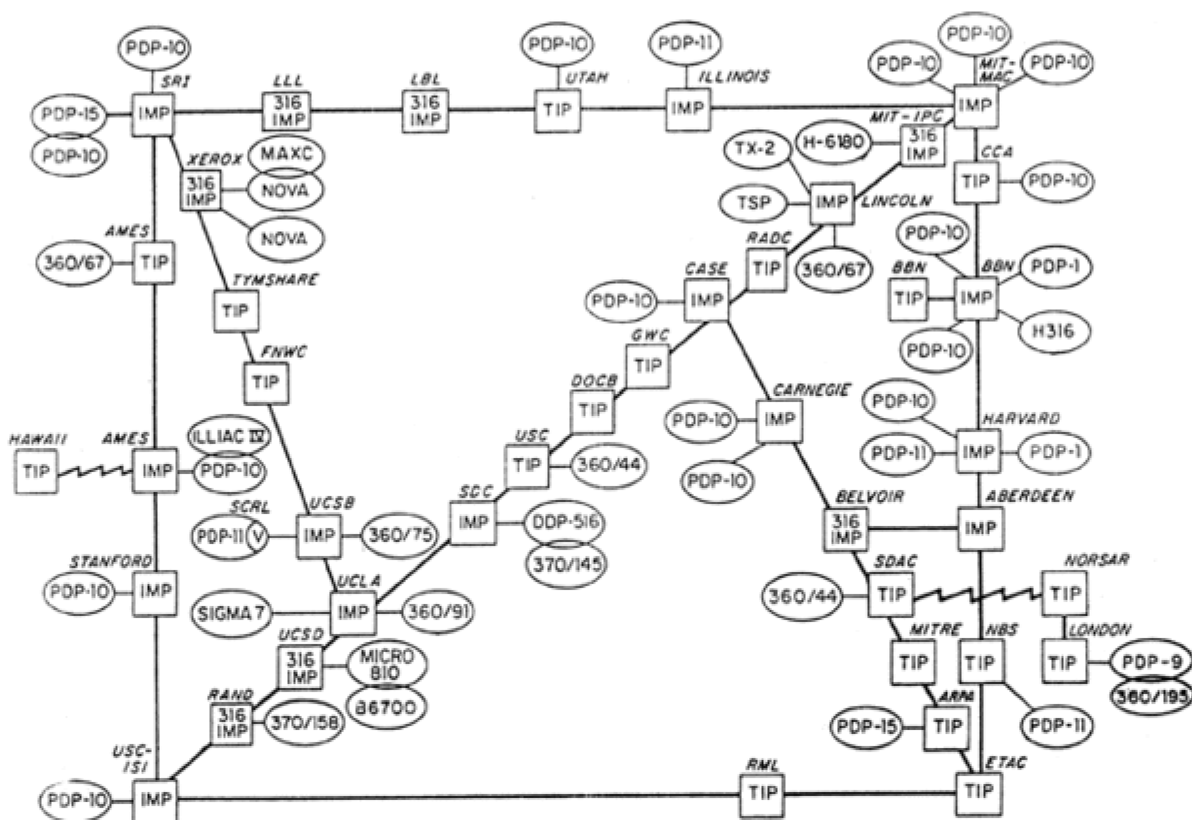


Figura 5 - Arpanet em 1973

Em 1974 surgiu o TCP/IP, que acabou se tornando o protocolo definitivo para uso na ARPANET e mais tarde na Internet. Essa rede interligando diversas universidades permitiu o livre tráfego de informações, levando ao desenvolvimento de recursos que usamos até hoje, como o e-mail, o telnet e o FTP. Na época, mainframes com um bom poder de processamento eram raros e caros, de forma que eles acabavam sendo compartilhados entre diversos pesquisadores e técnicos, que podiam estar situados em qualquer ponto da rede.

Um dos supercomputadores mais poderosos da época, acessado quase que unicamente via rede, era o Cray-1 (fabricado em 1976). Ele operava a 80 MHz, executando duas instruções por ciclo, e contava com 8 MB de memória, uma configuração que só seria alcançada pelos computadores domésticos quase duas décadas depois.

Com o crescimento da rede, manter e distribuir listas de todos os hosts conectados foi se tornando cada vez mais dispendioso, até que em 1980

passaram a ser usados nomes de domínio, dando origem ao "Domain Name System", ou simplesmente DNS, que é essencialmente o mesmo sistema para atribuir nomes de domínio usado até hoje.

Outra parte da história começa em 1973 dentro do PARC (o laboratório de desenvolvimento da Xerox, em Palo Alto, EUA), quando foi feito o primeiro teste de transmissão de dados usando o padrão Ethernet. O teste deu origem ao primeiro padrão Ethernet, que transmitia dados a 2.94 megabits através de cabos coaxiais e permitia a conexão de até 256 estações de trabalho. A figura 6, elaborada por Bob Metcalf, o principal desenvolvedor do padrão, ilustra o conceito:

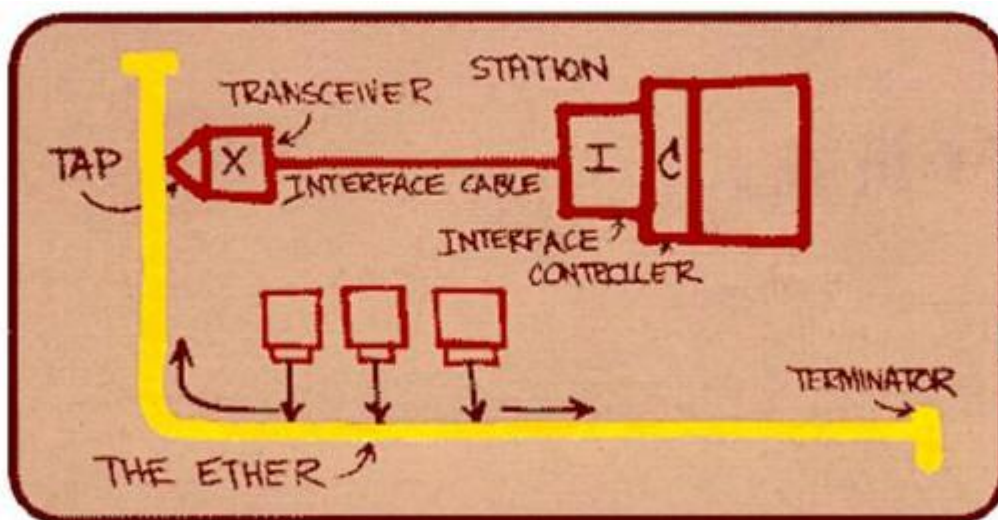


Figura 6 - Primeiro esboço do padrão Ethernet

O termo "ether" era usado para descrever o meio de transmissão dos sinais em um sistema. No Ethernet original, o "ether" era um cabo coaxial, mas em outros padrões pode ser usado um cabo de fibra óptica, ou mesmo o ar, no caso das redes wireless. O termo foi escolhido para enfatizar que o padrão Ethernet não era dependente do meio e podia ser adaptado para trabalhar em conjunto com outras mídias.

Curiosamente, isso aconteceu muito antes do lançamento do primeiro micro computador, o que só aconteceu em 1981. Os desenvolvedores do PARC criaram diversos protótipos de estações de trabalho (mostrada na figura 7) durante a década de 70, incluindo versões com interfaces gráficas elaboradas (para a época) que acabaram não entrando em produção devido ao custo. O padrão Ethernet surgiu, então, da necessidade natural de ligar estas estações de trabalho em rede.

Figura 7 - Xerox Alto (1973), a primeira estação de trabalho e também a primeira a ser ligada em rede.

A taxa de transmissão de 2.94 megabits do Ethernet original era derivada do clock de 2.94 MHz usado no Xerox Alto, mas ela foi logo ampliada para 10 megabits, dando origem aos primeiros padrões Ethernet de uso geral. Eles foram então sucessivamente aprimorados, dando origem aos padrões utilizados hoje em dia.

A ARPANET e o padrão Ethernet deram origem, respectivamente, à Internet e às redes locais, duas inovações que revolucionaram a computação.

Inicialmente, a ARPANET e o padrão Ethernet eram tecnologias sem relação direta. Uma servia para interligar servidores em universidades e outras instituições e a outra servia para criar redes locais, compartilhando arquivos e impressoras entre os computadores, facilitando a troca de arquivos e informações em ambientes de trabalho e permitindo o melhor aproveitamento dos recursos disponíveis.

Na década de 1990, com a abertura do acesso à Internet, tudo ganhou uma nova dimensão e as redes se popularizaram, já que ter uma rede local era a forma mais barata de conectar todos os micros da rede à Internet.

Há apenas uma década, o acesso via linha discada ainda era a modalidade mais comum e não era incomum ver empresas onde cada micro tinha um modem e uma linha telefônica, o que multiplicava os custos. Nessas situações, locar uma linha de frame relay (uma conexão dedicada de 64 kbits) e compartilhar a conexão entre todos os micros reduzia custos, além de permitir que todos eles ficassem permanentemente conectados. Com a popularização das conexões de banda larga, a escolha ficou ainda mais evidente.

3.1.3. Evolução do Cabeamento

Atualmente, as redes Ethernet de 100 megabits (Fast Ethernet) e 1000 megabits (Gigabit Ethernet) são as mais usadas. Ambos os padrões utilizam cabos de par trançado categoria 5 ou 5e, que são largamente disponíveis, o que facilita a migração de um para o outro. As placas também são intercompatíveis: pode-se usar placas de 100 e 1000 megabits na mesma rede, mas, ao usar placas de velocidades diferentes, a velocidade é sempre nivelada por baixo, ou seja, as placas Gigabit são obrigadas a respeitar a velocidade das placas mais lentas.

Antes deles, havia o padrão de 10 megabits, que também foi largamente usado (e ainda pode ser encontrado em algumas instalações) e, no outro extremo, já está disponível o padrão de 10 gigabits (10G), mil vezes mais rápido que o padrão original. Tal evolução demandou também melhorias no cabeamento da rede.

As primeiras redes Ethernet utilizavam cabos thicknet, um tipo de cabo coaxial grosso e pouco flexível, com 1 cm de diâmetro. Um único cabo era usado como backbone para toda a rede e as estações eram conectadas a ele através de transceptores, também chamados de "vampire taps" ou "derivadores vampiros", nome usado porque o contato do transceptor perfurava o cabo thicknet, fazendo contato com o fio central. O transceptor era então ligado a um conector AUI de 15 pinos na placa de rede, através de um cabo menor. A figura 8 ilustra o esquema de ligação:

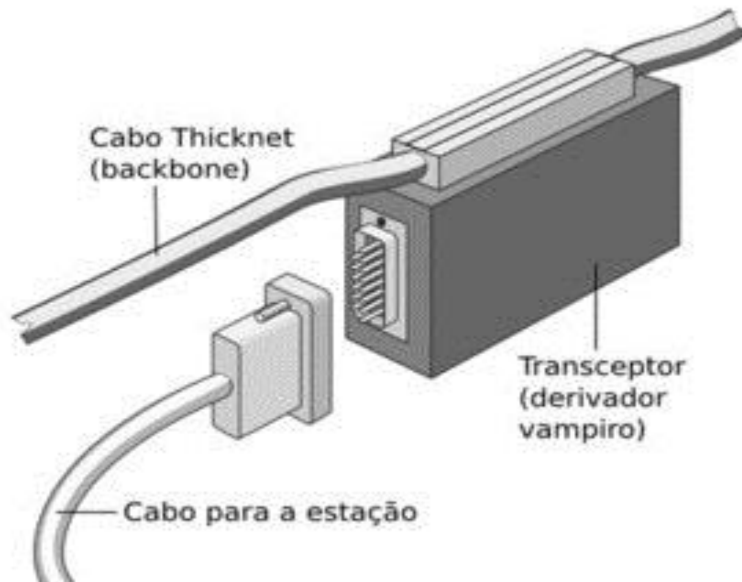


Figura 8 - Cabo thicknet e esquema de ligação

Este era essencialmente o mesmo tipo de cabeamento utilizado no protótipo de rede Ethernet desenvolvido no PARC, mas continuou sendo usado durante a maior parte da década de 80, embora oferecesse diversos problemas práticos, entre eles a dificuldade em se lidar com o cabo central, que era pesado e pouco flexível, além do elevado custo dos transceptores.

Estas redes eram chamadas de 10BASE-5, sigla que é a junção de 3 informações. O "10" se refere à velocidade de transmissão, 10 megabits, o "BASE" é abreviação de "baseband modulation", o que indica que o sinal é transmitido diretamente, de forma digital (sem o uso de modems, como no sistema telefônico), enquanto o "5" indica a distância máxima que o sinal é capaz de percorrer, nada menos do que 500 metros.

As redes 10BASE-5 logo deram origem às redes 10BASE-2, ou redes thinnet, que utilizavam cabos RG58/U, bem mais finos. O termo "thinnet" vem justamente da palavra "thin" (fino), enquanto "thicknet" vem de "thick" (espesso).

Nelas, os transceptores foram miniaturizados e movidos para dentro das próprias placas de rede e a ligação entre as estações passou a ser feita usando cabos mais curtos, ligados por um conector em forma de T, como pode ser visto na figura 9. Ele permitiu que as estações fossem ligadas diretamente

umas às outras, transformando os vários cabos separados em um único cabo contínuo.

Figura 9 - Cabos coaxiais com o conector em “T”.

Nas duas extremidades eram usados terminadores(ilustrado na figura 10), que fecham o circuito, evitando que os sinais que chegam ao final do cabo retornem na forma de interferência.

Figura 10 - Terminal do cabo coaxial.

Na Figura 11 pode-se observar o esquema de ligação da rede, com o terminador.

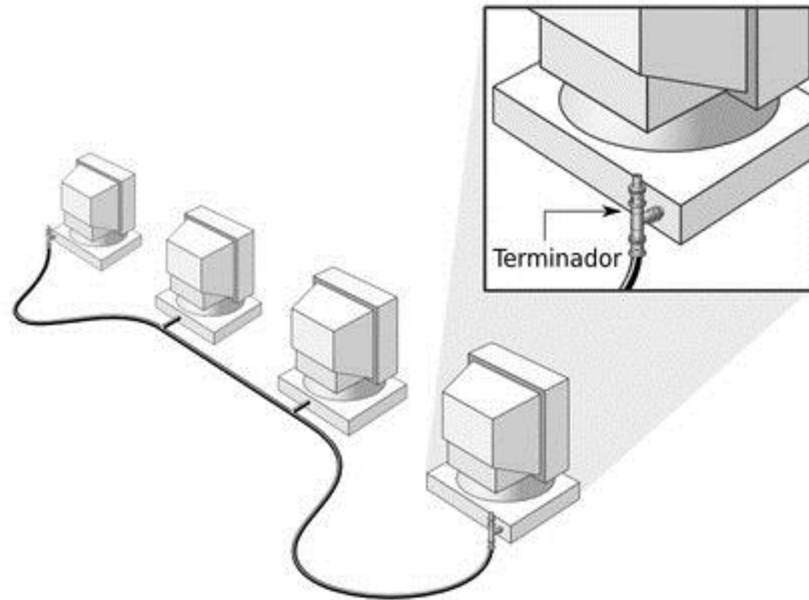


Figura 11 - Esquema de ligação, mostrando a terminação.

Apesar da importância, os terminadores eram dispositivos passivos, bastante simples e baratos. O grande problema era que, se o cabo fosse desconectado em qualquer ponto (no caso de um cabo rompido, ou com mau contato, por exemplo), toda a rede saía ar, já que era dividida em dois segmentos sem terminação. Como não eram usados leds nem indicadores de conexão, existiam apenas duas opções para descobrir onde estava o problema: usar um testador de cabos (um aparelho que indicava com precisão em que ponto o cabo estava rompido, mas que era caro e justamente por isso incomum aqui no Brasil) ou testar ponto por ponto, até descobrir onde estava o problema.

A figura 12 mostra o conector BNC, incluindo a ponteira e a bainha, o conector T e o terminador, que, junto com o cabo coaxial, eram os componentes básicos das redes 10BASE-2.

Figura 12 - Conectores BNC

As conexões dos cabos podiam ser realizadas na hora, de acordo com o comprimento necessário, usando um alicate especial. A conexão consistia em descascar o cabo coaxial, encaixá-lo dentro do conector, apertar ponteira, de forma a prender o fio central e em seguida apertar a bainha, prendendo o cabo ao conector BNC.

Apesar de ainda ser muito susceptível a problemas, o cabeamento das redes 10BASE-2 era muito mais simples e barato do que o das redes 10BASE-5, o que possibilitou a popularização das redes, sobretudo em empresas e escritórios.

Figura 13 - Placa ISA, com conector BNC e conector RJ45

A figura 13 mostra uma placa ISA de 10 megabits, que além do conector AUI, inclui o conector BNC para cabos coaxiais thinnet e o conector RJ45 para cabos de par trançado atuais. Estas placas foram muito usadas durante o início da década de 90, o período de transição entre os três tipos de cabeamento. A vantagem era que você podia migrar dos cabos coaxiais para os cabos de par

trançado trocando apenas o cabeamento, sem precisar trocar as placas de rede.

A única desvantagem das redes thinnet em relação às thicknet é que o uso de um cabo mais fino reduziu o alcance máximo da rede, que passou a ser de apenas 185 metros, o que de qualquer forma era mais do que suficiente para a maioria das redes locais. Surpreendentemente, o obsoleto padrão 10BASE-5 foi o padrão Ethernet para cabos coaxiais de cobre com o maior alcance até hoje, com seus 500 metros. Apenas os padrões baseados em fibra óptica são capazes de superar esta marca.

Independentemente do tipo, os cabos coaxiais (mostrado na figura 14) seguem o mesmo princípio básico, que consiste em utilizar uma camada de blindagem para proteger o cabo central de interferências eletromagnéticas presentes no ambiente. Quanto mais espesso o cabo e mais grossa é a camada de blindagem, mais eficiente é o isolamento, permitindo que o sinal seja transmitido a uma distância muito maior.

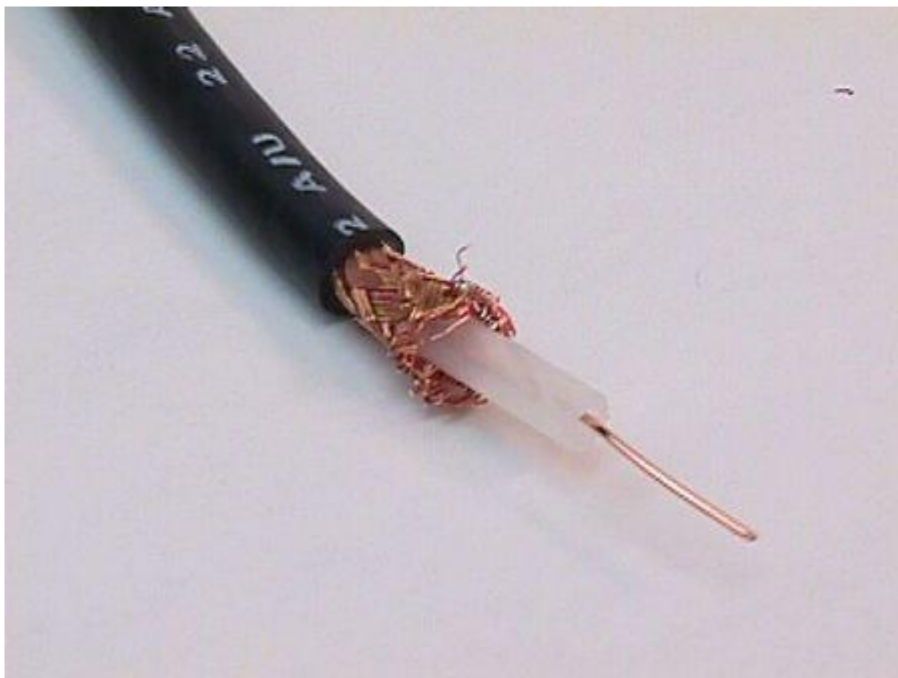


Figura 14 - Cabo coaxial, com blindagem

Os cabos coaxiais foram substituidos pelos cabos de par trançado (como mostrado na Figura 15), que são praticamente os únicos usados em redes locais atualmente. Além de serem mais finos e flexíveis, os cabos de par

trançado suportam maiores velocidades (podem ser usados em redes de 10, 100 ou 1000 megabits, enquanto os cabos coaxiais são restritos às antigas redes de 10 megabits), além de serem mais baratos.

Figura 15 - Conector RJ45 e cabos de par trançado

Apesar disso, os cabos coaxiais estão longe de entrar em desuso. Além de serem usados nos sistemas de TV a cabo e em outros sistemas de telecomunicação, eles são usados em todo tipo de antenas, incluindo antenas para redes wireless. Até mesmo os conectores tipo N, tipicamente usados nas antenas para redes wireless de maior ganho são descendentes diretos dos conectores BNC usados nas redes 10BASE-2.

Existem diversas categorias de cabos de par trançado, que se diferenciam pela qualidade e pelas freqüências suportadas. Por exemplo, cabos de categoria 3, que são largamente utilizados em instalações telefônicas podem ser usados em redes de 10 megabits, mas não nas redes de 100 e 1000 megabits atuais. Da mesma forma, os cabos de categoria 5e que usamos atualmente não são adequados para as redes de 10 gigabits, que demandam cabos de categoria 6, ou 6a. Todos eles utilizam o mesmo conector, o RJ-45, mas existem diferenças de qualidade entre os conectores destinados a diferentes padrões de cabos.

Os sucessores naturais dos cabos de par trançado são os cabos de fibra óptica, que suportam velocidades ainda maiores e permitem transmitir a distâncias praticamente ilimitadas, com o uso de repetidores. Os cabos de fibra óptica são usados para criar os backbones que interligam os principais roteadores da Internet, como mostrado na figura 16. Sem eles, a grande rede seria muito mais lenta e o acesso muito mais caro.

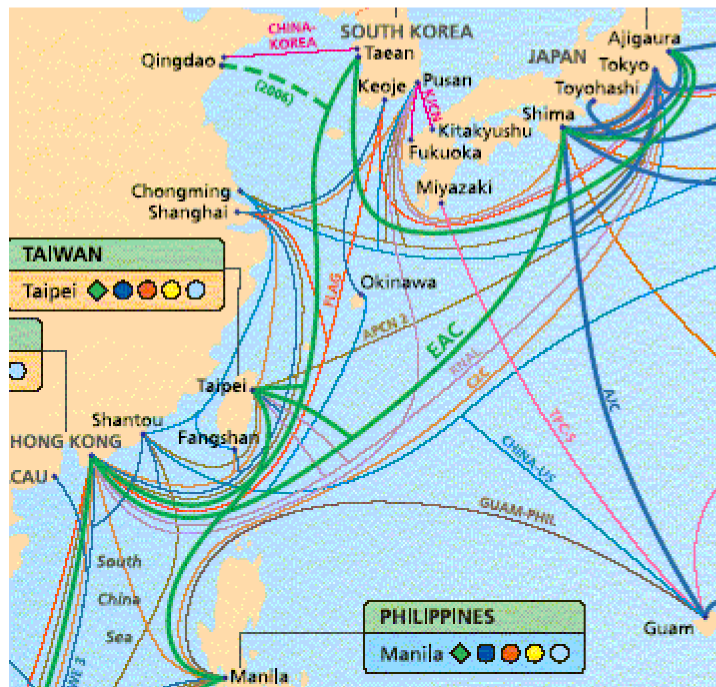


Figura 16 - Backbones de fibra óptica interligando países da Ásia

Apesar disso, os cabos de fibra óptica, mostrados na figura 17, ainda são pouco usados em redes locais, devido à questão do custo, tanto dos cabos propriamente ditos, quanto das placas de rede, roteadores e demais componentes necessários. Apesar de tecnicamente inferiores, os cabos de par trançado são baratos, fáceis de trabalhar e tem resistido ao surgimento de novos padrões de rede.

Durante muito tempo, acreditou-se que os cabos de par trançado ficariam limitados às redes de 100 megabits e, conforme as redes gigabit se popularizassem eles entrariam em desuso, dando lugar aos cabos de fibra óptica. Mas a idéia não se concretizou com o surgimento do padrão de redes gigabit para cabos de par trançado que usamos atualmente.

A história se repetiu com o padrão 10 gigabit (que ainda está em fase inicial de adoção), que inicialmente previa apenas o uso de cabos de fibra óptica. Contrariando todas as expectativas, conseguiu-se levar a transmissão de dados em fios de cobre ao limite, criando um padrão de 10 gigabits para cabos de par trançado. Seguindo a lógica de que leva pelo menos uma década para um novo padrão de redes se popularizar (assim foi com a migração das redes de 10 megabits para as de 100 e agora das de 100 para as de 1000), os cabos

de par trançado têm sua sobrevivência assegurada por pelo menos mais uma década.

Figura 17 - Cabos de fibra óptica multimodo

3.1.4. Redes Wireless

Surpreendentemente, a primeira rede wireless funcional, a ALOHANET, entrou em atividade em 1970, antes mesmo do surgimento da Arpanet.

Ela surgiu da necessidade de criar linhas de comunicação entre diferentes campus da universidade do Havaí, situados em ilhas diferentes. Na época, a estrutura de comunicação era tão precária que a única forma de comunicação era mandar mensagens escritas de barco, já que, devido à distância, não existiam sequer linhas de telefone.

A solução encontrada foi usar transmissores de rádio amador, que permitiam que nós situados nas diferentes ilhas se comunicassem com um transmissor central, que se encarregava de repetir as transmissões, de forma que elas fossem recebidas por todos os demais. A velocidade de transmissão era muito baixa, mas a rede era funcional.

Como todos os transmissores operavam na mesma frequência, sempre que dois nós tentavam transmitir ao mesmo tempo, acontecia uma colisão e ambas as transmissões precisavam ser repetidas, o que era feito automaticamente depois de um curto espaço de tempo. Este mesmo problema ocorre nas redes wireless atuais, que naturalmente incorporam mecanismos para lidar com ele.

Voltando aos dias de hoje, as redes wireless se tornaram populares, pois permitem criar redes locais rapidamente, sem necessidade de espalhar cabos pelo chão. Além da questão da praticidade, usar uma rede wireless pode em muitos casos ser de menor custo, já que o preço de centenas de metros de cabo, combinado com o custo da instalação, pode superar em muito a diferença de preço no ponto de acesso e nas placas.

Existem dois tipos de redes wireless. As redes em modo infra-estrutura são baseadas em um ponto de acesso ou um roteador wireless, que atua como um ponto central, permitindo a conexão dos clientes. As redes ad-hoc, por sua vez, são um tipo de rede mesh, onde as estações se comunicam diretamente, sem o uso de um ponto de acesso. Embora tenham um alcance reduzido, as redes ad-hoc são uma forma prática de interligar notebooks em rede rapidamente, de forma a compartilhar a conexão ou jogar em rede. Como todos os notebooks hoje em dia possuem placas wireless integradas, criar uma rede ad-hoc pode ser mais rápido do que montar uma rede cabeada.

O alcance típico dos pontos de acesso domésticos são 33 metros em ambientes fechados e 100 metros em campo aberto. Apesar disso, é possível estender o sinal da rede por distâncias muito maiores, utilizando pontos de acesso e placas com transmissores mais potentes ou antenas de maior ganho. Desde que exista um caminho livre de obstáculos, não é muito difícil interligar redes situadas em dois prédios diferentes, a 5 km de distância, por exemplo.

Por outro lado, o sinal é facilmente obstruído por objetos metálicos, paredes, lajes e outros obstáculos, além de sofrer interferência de diversas fontes. Devido a isso, deve-se procurar sempre instalar o ponto de acesso em um ponto elevado do ambiente, de forma a evitar o maior volume possível de obstáculos.

O primeiro padrão a se popularizar foi o 802.11b, que operava a apenas 11 megabits. Ele foi seguido pelo 802.11g, que opera a 54 megabits e pelo 802.11n, que oferece até 300 megabits. Apesar disso, as redes wireless trabalham com um overhead muito maior que as cabeadas, devido à modulação do sinal, colisões e outros fatores, de forma que a velocidade real passa a ser um pouco menos da metade do prometido. Além disso, a velocidade máxima é obtida apenas enquanto o sinal está bom e existe apenas um computador transmitindo. Conforme o sinal fica mais fraco, ou vários computadores passam a transmitir simultaneamente, a velocidade decai, razão pela qual algumas redes wireless se tornam tão lentas.

3.2. Camadas de Rede

A ISO (International Standarzation Organization), a fim de padronizar a comunicação entre computadores, definiu um modelo de referencia chamado OSI (Open Systems interconnection).

Esse modelo divide as redes de computadores em sete camadas, de forma a se obter camadas de abstração. A cada uma delas é atribuida um protocolo que implementa uma determinada funcionalidade.

3.2.1. Camada Física

A camada física define as características técnicas dos dispositivos elétricos e ópticos do sistema. Ela contém os equipamentos de cabeamento ou outros canais de comunicação que se comunicam diretamente com o controlador da interface de rede. Preocupa-se, portanto, em permitir uma comunicação bastante simples e confiável.
Pode ser especificada a transmissão pelos seguintes termos: RS-232, V.35, V.34, Q.911, T1, E1, 10BASE-T,100BASE-TX , ISDN, SONET, DSL

3.2.2 Camada de enlace

A camada de enlace trata as topologias de rede, dispositivos como Switch, placa de rede, interfaces, e é responsável por todo o processo de switching. Após o recebimento dos bits, ela os converte de maneira inteligível, os transforma em unidade de dado, subtrai o endereço físico e encaminha para a camada de rede que continua o processo. Ela também estabelece um protocolo de comunicação entre sistemas diretamente conectados.Exemplos de protocolos nesta camada: Ethernet, Token Ring, FDDI, PPP, HDLC, Q.921, Frame Relay, ATM.

O mais largamente utilizado atualmente é o padrão Ethernet.

3.2.3 Camada de Rede

A camada de Rede é responsável pelo endereçamento dos pacotes, convertendo endereços lógicos (IP) em endereços físicos (MAC) , de forma que os pacotes consigam chegar corretamente ao destino. Essa camada também determina a rota que os pacotes irão seguir para atingir o destino,a partir de dispositivos como roteadores, baseada em fatores como condições de tráfego da rede e prioridades.

Essa camada é usada quando a rede possui mais de um segmento e, com isso, há mais de um caminho para um pacote de dados percorrer da origem ao destino.

Abrangem essa camada os protocolos IP (IPv4, IPv6) , ARP, RARP, ICMP, IPSec. O mais utilizado atualmente é o IP versão 4 ou IPv4.

3.2.3.1. Protocolo IP

Os dados numa rede IP são enviados em blocos referidos como ficheiros. Em particular, no IP nenhuma definição é necessária antes de um nó tentar enviar ficheiros para outro nó com o qual não comunicou previamente.

Como característica esse protocolo oferece um serviço de datagramas não confiável, ou seja, o pacote vem quase sem garantias. Os pacotes podem chegar desordenados (comparados com outros pacotes enviados entre os mesmos nós), duplicados, ou podem ser perdidos por inteiro. Se a aplicação requer maior confiabilidade, esta é adicionada na camada de transporte.

Os roteadores são usados para reencaminhar datagramas IP através das redes interconectadas na segunda camada. A falta de qualquer garantia de entrega significa que o desenho da troca de pacotes é feito de forma mais simplificada.

O IP é descrito no RFC 791 da IETF, que foi pela primeira vez publicada em Setembro de 1981. Esta versão do protocolo é designada de versão 4, ou IPv4. O IPv6 tem endereçamento de origem e destino de 128 bits, oferecendo mais endereçamentos que os 32 bits do IPv4.

3.2.3.2. Datagrama IP

Após uma visão geral, pode-se analisar com detalhe o datagrama do protocolo, que é composto pelo header mostrado na Figura 18.

+	0 - 3	4 - 7	8 - 15	16 - 18	19 - 31
0	Versão	Tamanho do cabeçalho	*Tipo de Serviço* (ToS) (agora DiffServ e ECN)	Comprimento (pacote)	
32	Identificador			*Flags*	*Offset*
64	*Tempo de Vida* (TTL)		Protocolo	*Checksum*	
96	Endereço origem				
128	Endereço destino				
160	Opções				
192	*Dados*				

Figura 18 - Cabeçalho do protocolo IP

O primeiro campo de quatro bits de um datagrama (VERS), por exemplo, contém a versão do protocolo IP utilizada. Ele é usado para verificar se o transmissor, o receptor e quaisquer roteadores existentes entre ele concordam quanto ao formato do datagrama. Todo software IP precisa verificar o campo de versão antes de processar um datagrama, para assegurar-se de que ele se adapta ao formato que o software espera. Se os padrões mudarem, as máquinas rejeitarão datagramas com versões de protocolos diferentes dos seus, impedindo que eles deturpem o conteúdo da datagrama com um formato desatualizado.

O campo de comprimento do cabeçalho (HLEN), também de quatro bits, fornece o comprimento do cabeçalho do datagrama medido em palavras de 32 bits. Todos os campos do cabeçalho contêm um comprimento fixo, exceto para OPÇÕES IP e os campos correnpondentes PADDING. Portanto, o cabeçalho mais comum, que não contém qualquer opção e nenhum preenchimento, mede 20 octetos e o campo de comprimento de cabeçalho é cinco.

O campo COMPRIMENTO TOTAL fornece o comprimento do datagrama IP medido em octetos, incluindo octetos no cabeçalho e nos dados. O tamanho da área de dados pode ser calculado subtraindo-se de COMPRIMENTO TOTAL o comprimento do cabeçalho (HLEN). Já que o campo COMPRIMENTO TOTAL possui 16 bits de comprimento, o maior tamanho possível para um datagrama IP é 2^{16} ou 65535 octetos. Na maioria dos aplicativos, essa não é uma limitação rígida. No futuro pode tornar-se mais importante, se as redes de velocidade mais alta puderem transportar pacotes de dados maiores que 65.535 octetos.

O campo TIPO DE SERVIÇO (TOS), de oito bits, especifica como o datagrama deve ser tratado e é fracionado em cinco subcampos, como mostrado na Figura 19.

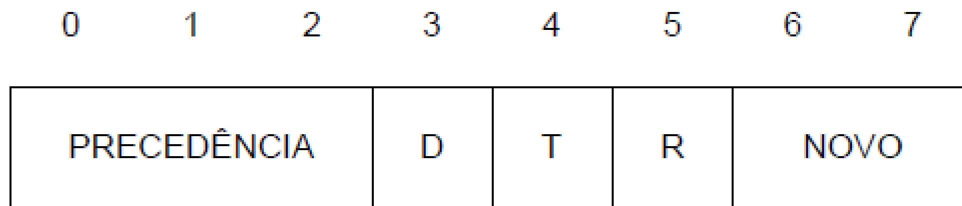


Figura 19 - Campo TOS (Type of service)

Três bits PRECEDÊNCIA especificam a precedência do datagrama com valores variando de zero (precedência normal) até sete (controle de rede), permitindo que os transmissores indiquem a importância de cada datagrama. Embora a maioria dos softwares que rodam e hosts ignorem o tipo de serviço, trata-se de um conceito importante, porque fornece um mecanismo que pode permitir que informações de controle tenham precedência sobre dados. Se, por exemplo, todos os hosts e roteadores reconhecem a precedência, é possível implementar algoritmos de controle de congestionamento que não sejam influenciados pelo congestionamento que estão tentando controlar.

Os bits D, T e R especificam o tipo de transporte que o datagrama deseja. Quando ajustado, o bit D solicita um intervalo baixo, o bit T solicita um throughput alto e o bit R solicita alta confiabilidade. É claro que não deve ser possível que a interligação em redes garanta o tipo de transporte solicitado (ou seja, pode acontecer que nenhum caminho para o destino tenha a propriedade solicitada). Se um roteador realmente conhece mais de uma rota possível para determinado destino, ele pode utilizar o tipo de campo de transporte para selecionar aquelas cujas características mais se aproximem das desejadas. Supondo, por exemplo, que um roteador possa selecionar entre uma linha alugada, de baixa capacidade, e uma conexão de alta, de satélites de banda larga (mas de intervalo alto), o conjunto de bits D poderia solicitar aos datagramas que carregam toque no teclado de um usuário para um computador remoto que esses sejam entregues o mais rápido possível, enquanto um conjunto de bits T poderia solicitar aos datagramas correspondentes uma transferência de arquivos que trafeguem nos links de alta capacidade de satélites.

Também é importante entender que os algoritmos de roteamento precisam escolher entre tecnologias de redes físicas básicas as quais possuem , cada uma, características de intervalo, throughput e confiabilidade. De uma maneira geral, algumas tecnologias representam um compromisso entre duas características (por exemplo, um maior throughput em detrimento de maiores retardos). Assim, a idéia é apresentar uma sugestão ao algoritimo de roteamento sobre o que é mais importante, e raramente faz sentido especificar os três tipos de serviço.

3.2.3.3. Encapsulamento de Datagramas

Cada tecnologia de comutação de pacotes coloca um limite superior, fixo, no total de dados que podem ser transferido em um quadro físico. A Ethernet, por

exemplo, limita as transferências a 1.500 octetos de dados, enquanto a FDDI permite aproximadamente 4.470 octetos de dados por quadro. Refere-se a esses limites como MTU (m*aximum transfer unit*). O tamanho da MTU pode ser bem pequeno: algumas tecnologias de hardware limitam a transferência para 128 octetos ou menos. Limitar os datagramas para encaixar a menor MTU possível na interligação em redes torna a transferência ineficaz quando aqueles datagramas trafegam em uma rede que pode transportar quadros de tamanho maiores. Enquanto, permitir que os datagramas sejam maiores que a MTU mínima da rede em uma interconexão significa que um datagrama nem sempre irá encaixar-se no quadro único de uma rede.

A escolha deve ser óbvia: o objetivo do projeto de interligação em redes é concentrar as tecnologias de rede básicas e facilitar a comunicação para o usuário. Assim, ao invés de projetar datagramas que sigam as restrições de redes físicas, o software TCP/IP escolhe um tamanho inicial de datagrama conveniente e descobre uma forma de dividir os datagramas extensos em frações menores, quando o datagrama precisa atravessar uma rede que tenha uma MTU pequena. As pequenas frações em que um datagrama é dividido são denominadas *fragmentos*, e o processo de divisão de um datagrama é conhecido como *fragmentação*.

Conforme a Figura 20, a fragmentação normalmente ocorre em um roteador situado em algum ponto ao longo do caminho entre a origem do datagrama e seu destino final. O roteador recebe um datagrama de uma rede com uma MTU grande, e precisa enviá-lo em uma rede para a qual a MTU seja menor do que o tamanho do datagrama.

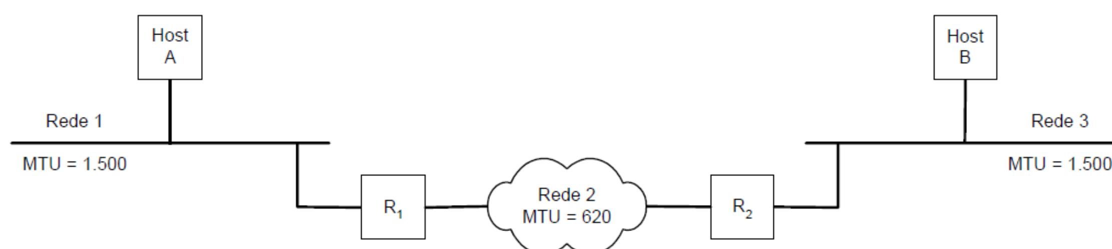


Figura 20 - Esquema mostrando fragmentação de datagramas em redes com diferentes MTU's

O tamanho do fragmento é escolhido de tal forma que cada fragmento possa ser transportado na rede básica em um quadro único. Além disso, já que o IP representa o deslocamento dos dados em múltiplos de oito octetos, o tamanho do fragmento precisa ser um múltiplo de oito. Deve-se considerar que escolher o múltiplo de oito octetos mais próximo do MTU da rede nem sempre divide o datagrama em frações de igual tamanho, pois a última fração é normalmente menor que as outras.

Sintetizando, o protocolo IP não limita datagramas a um tamanho pequeno, nem garante que datagramas grandes serão entregues sem fragmentação. A origem pode escolher qualquer tamanho de datagrama que julgar apropriado. A especificação do IP indica que os roteadores precisam aceitar datagramas até o máximo de MTUs de rede às quais se conectam.

Cada fragmento contém um cabeçalho de datagrama que duplica a maior parte do cabeçalho do datagrama original (exceto para um bit no campo FLAGS que mostra que é um fragmento), seguido por tantos dados quantos puderem ser transportados no fragmento, enquanto mantém o comprimento total menor que a MTU da rede na qual precisa trafegar, como pode ser observado na Figura 21.

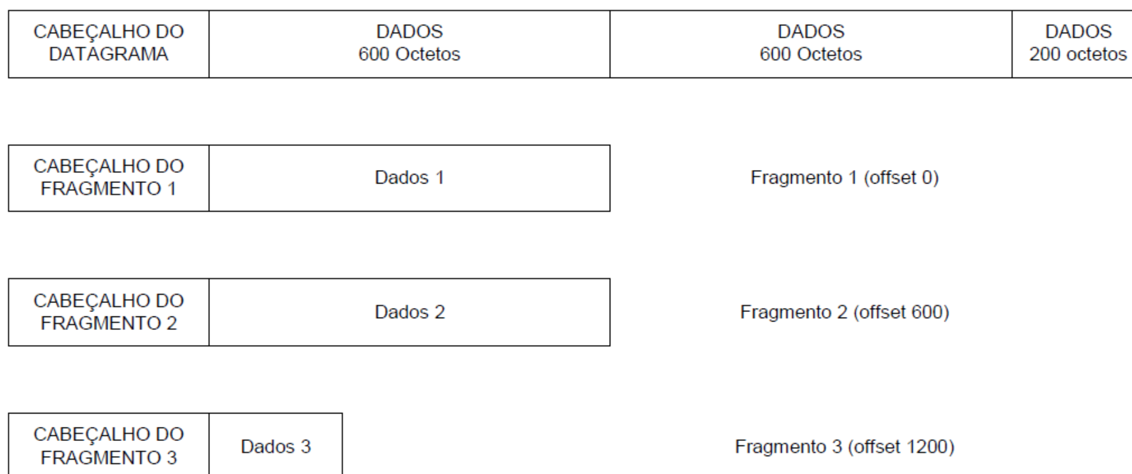


Figura 21 - Datagrama Fragmentado

3.2.3.4. Remontagem de Fragmentos

Em uma interligação de redes TCP/IP, quando um datagrama tiver sido fragmentado, os fragmentos trafegam como datagramas isolados ao longo do percurso até o último destino onde precisam ser remontados. Há duas desvantagens em preservar os fragmentos ao longo do percurso até o final. Primeiramente, a remontagem de datagramas no destino final pode levar à ineficiência: mesmo se algumas redes físicas encontradas após o ponto de fragmentação possuírem grande capacidade de MTU, apenas pequenos fragmentos atravessam-na. Além disso, se quaisquer fragmentos forem perdidos, o datagrama não pode ser remontado. A máquina receptora inicia um *temporizador de remontagem* quando recebe um fragmento inicial e se o temporizador terminar antes que todos os fragmentos cheguem, a máquina receptora descarta os fragmentos remanescentes sem processar o datagrama. Assim, a probabilidade de perda de datagrama cresce quando a fragmentação ocorre, porque a perda de um fragmento único resulta na perda do datagrama inteiro.

Apesar das pequenas desvantagens, a execução de remontagem no destino final tem um desempenho satisfatório. Permite que cada fragmento seja roteado independentemente, e não exige que roteadores intermediários armazenem ou remontem fragmentos.

3.2.3.5. Controle de Fragmentação

Três campos no cabeçalho do datagrama, IDENTIFICAÇÃO, FLAGS e OFFSET DE FRAGMENTO,controlam a fragmentação e a remontagem de datagramas. O campo IDENTIFICAÇÃO contém um número inteiro único que identifica o datagrama. Sua finalidade é permitir que o destino saiba quais datagramas estão chegando e a que datagramas pertencem. À medida que chega um fragmento, o destino utiliza o campo IDENTIFICAÇÃO juntamente com o endereço de origem do datagrama para que esse seja identificado. Os computadores que estão enviando os datagramas IP devem gerar um valor único para o campo IDENTIFICAÇÃO, para cada datagrama. Uma técnica utilizada pelo software IP mantém uma contagem global em memória, incrementa-a a cada vez que um novo datagrama é criado e atribui o resultado como o campo IDENTIFICAÇÃO do datagrama.

Para um fragmento, o campo OFFSET DE FRAGMENTO especifica o deslocamento, no datagrama original, dos dados que estão sendo transportados no fragmento, medidos em unidades de oito octetos, iniciando em deslocamento zero. Para remontar o datagrama, o destino precisa obter todos os fragmentos que iniciam com o framento que possui deslocamento zero até o fragmento de maior deslocamento. Os fragmentos não chegam necessariamente em ordem, e não há comunicação entre o roteador que fragmenta o datagrama e o destino que está tentando remontá-lo.

Os dois bits de baixa ordem, do campo FLAGS de três bits, controlam a fragmentação. Normalmente, o software aplicativo que utiliza TCP/IP não dá atenção à fragmentação, porque ela e a remontagem são procedimentos automáticos que ocorrem em um baixo nível do sistema operacional, invisível para usuários finais. Entretanto, para testar o software de interligação em redes ou depurar problemas operacionais, deve ser importante testar os tamanhos dos datagramas para os quais a fragmentação ocorre. O primeiro bit de controle auxilia nesse teste, especificando se o datagrama pode ser fragmentado. Ele é conhecido como bit *não-fragmentar* porque o seu ajuste em um especifica que o datagrama não deve ser fragmentado. Um aplicativo pode optar por não permitir uma fragmentação quando somente o datagrama inteiro é útil. Considerando, por exemplo, um procedimento de inicialização de um computador, no qual uma máquina começa a executar um pequeno programa na ROM que utiliza a interligação em redes para solicitar uma inicialização inicial, e outra máquina retorna uma imagem de memória, se o software tiver sido projetado de tal modo que a imagem de inicialização só tenha utilidade se obtida de uma única vez, o datagrama deve ter um conjunto de bits *não-fragmentar*. Toda vez que um roteador precisa fragmentar um datagrama que possui o conjunto de bits *não-fragmentar*, o roteador descarta o datagrama e retorna à origem uma mensagem de erro.

O bit de mais baixa ordem, no campo FLAGS, especifica se o fragmento contém a parte do meio ou do final dos dados do datagrama, sendo conhecido como bit de *mais fragmentos*. Esse bit é necessário porque o campo COMPRIMENTO TOTAL, do cabeçalho, aplica-se ao tamanho do fragmento e não ao tamanho do datagrama original. Assim, o destino não pode utilizá-lo para inferir se reuniu todos os fragmentos. O bit de *mais fragmentos* resolve o problema: quando o destino recebe um fragmento com o bit *mais fragmentos*

desativado, ele sabe que o esse fragmento transporta a parte final dos dados do datagrama original.

Partindo dos campos de OFFSET DE FRAGMENTO e COMPRIMENTO TOTAL, ele pode calcular o comprimento do datagrama original. Examinando o OFFSET DE FRAGMENTO e o COMPRIMENTO TOTAL de todos os fragmentos que chegaram, um receptor pode dizer se os fragmentos sob controle contêm todos os dados necessários para remontar todo o datagrama original.

3.2.3.6. TTL (Time To Live ou Tempo de Vida)

O campo TEMPO DE VIDA especifica quanto tempo, em segundos, o datagrama pode permanecer no sistema de interligação em redes. Os roteadores e os hosts que processam datagramas precisam decrementar o campo TEMPO DE VIDA à medida que o tempo passa e remover o datagrama da interligação em redes quando seu tempo expira.

Além disso, para tratar as ocorrências de roteadores sobrecarregados,que implicam em retardos longos, cada roteador registra o tempo local quando o datagrama chega e decrementa o TEMPO DE VIDA no número de segundos que o datagrama permaneceu dentro do roteador esperando serviço.

Sempre que um campo TEMPO DE VIDA alcança zero, o roteador descarta o datagrama e envia uma mensagem de erro de volta à origem. A idéia de manter um temporizador para datagramas é interessante, porque assegura que os datagramas não podem trafegar indefinidamente na interligação em redes, ainda que as tabelas de roteamento fiquem destruídas e os roteadores direcionem datagramas em círculo.

3.2.4. Camada de Transporte

A camada de transporte é a parte central de toda a hierarquia de protocolos. Sua tarefa é prover o transporte econômico e confiável de dados, independente da rede física ou das redes atualmente em uso.

Isso inclui controle de fluxo, ordenação dos pacotes e correção de erros, tipicamente enviando para o transmissor uma informação de recebimento, informando que o pacote foi recebido com sucesso.

Exemplos de protocolos usados nessa camada são TCP, UDP, SCTP, DCCP

3.2.4.1 TCP

O TCP(Transmission Control Protocol - Protocolo de Controle de Transmissão) é o protocolo sobre o qual assentam a maioria das aplicações. Isto porque ele verifica se os dados são enviados de forma correta, na seqüência apropriada e sem erros, pela rede.

O cabeçalho desse protocolo é estruturado como mostrado na Figura 22.

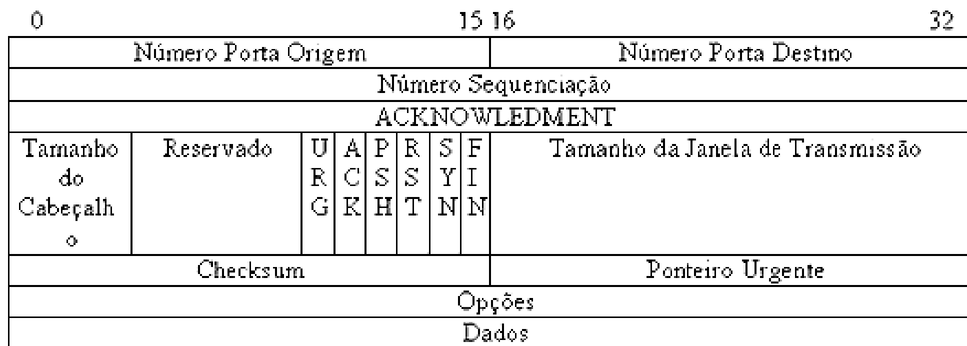


Figura 22 - Cabeçalho do protocolo TCP

Os campos do header podem ser definidos da seguinte forma:

- Número Porta Origem/Destino: número da porta do programa de aplicação nos pontos terminais locais da conexão;
- Número de Seqüenciação: posição de cada segmento de dado na palavra original;
- Acknowledgement: especifica o próximo byte aguardado;
- Tamanho do Cabeçalho: informa quantas palavras de 32-bits existem no cabeçalho TCP;
- Reservado: campo reservado para uso futuro;
- URG: usado para indicar um deslocamento de bit no número de seqüência no qual os dados urgentes deverão estar;
- ACK: indica se o Acknowledgement é válido;
- PSH: indica que o receptor dos dados deve entregar os dados à aplicação mediante sua chegada, em vez de armazená-los até que um buffer completo tenha sido recebido;
- RST: reinicia uma conexão que tenha ficado confusa devido a uma falha no host ou por qualquer outra razão;
- SYN: Usado em conjunto com o ACK para solicitar ou aceitar uma conexão

  –SYN=1 ACK=0: requisição de conexão

  –SYN=1 ACK=1: conexão aceita

  –SYN=0 ACK=1: “confirmação do recebimento”

- FIN: encerramento de uma conexão;
- Tamanho da Janela de Transmissão: indica quantos bytes podem ser enviados a partir do byte confirmado.
- Checksum: confere o cabeçalho TCP;
- Ponteiro Urgente: indica um deslocamento de bit no número de seqüência no qual os dados urgentes deverão estar;
- Dados: dados a serem transmitidos.

3.2.4.2 UDP

O UDP(*User Datagram Protocol)* permite acesso direto ao serviço de entrega de datagramas porém é pouco confiável, sendo um protocolo não orientado para conexão.

A entrega pode ser feita fora de ordem e datagramas podem ser perdidos. A integridade dos dados pode ser conferida por um "checksum" (um campo no cabeçalho de checagem por soma) baseado em complemento de um, de 16 bits.

Os pontos de acesso do UDP são geralmente designados por "portas" em que cada unidade de transmissão de dados UDP identifica o endereço IP e o número de porta do destino e da fonte da mensagem.

A diferença básica entre o UDP e o TCP é o fato de que o TCP é um protocolo orientado à conexão e, portanto, inclui vários mecanismos para iniciar, manter e encerrar a comunicação, negociar tamanhos de pacotes, detectar e corrigir erros, evitar congestionamento do fluxo e permitir a retransmissão de pacotes corrompidos, independente da qualidade do meio físicos.

O UDP, por sua vez, é feito para transmitir dados pouco sensíveis, como fluxos de áudio e vídeo, ou para comunicação sem conexão como é o caso da negociação DHCP ou tradução de endereços por DNS. No UDP não existem checagens e nem confirmação alguma sendo os dados transmitidos apenas uma vez. Os pacotes que chegam corrompidos são simplesmente descartados, sem que o emissor sequer saiba do problema. Por outro lado, a ausência de estruturas de controle complexas garante ao UDP alta eficiência, já que cada pacote é composto praticamente somente por dados.

A estrutura do datagrama é composto pelo seguinte header, dividido em quatro campos de 16 bits:

Porta de Origem	Porta de Destino
Comprimento da mensagem	Checksum

Porta de Origem e Destino: estes campos contêm os números de portas fonte e destino do protocolo UDP. A porta fonte é opcional, quando é usada ela especifica a porta a qual uma resposta poderia ser enviada, se não é usada contém zeros.

Comprimento da mensagem: contém um contador de bytes no datagrama UDP. O valor mínimo é oito, sendo este só o comprimento do cabeçalho.

Checksum: Este campo é opcional. Um valor de zero indica que o checksum não é computado.

3.2.5. Camada de Sessão

A camada de sessão tem um papel fundamental no mecanismo de comunicação entre dois hosts em uma rede. Esta camada oferece o suporte necessário para estruturar os circuitos que são disponibilizados pelo nível de transporte. Um exemplo de aplicação nessa camada, é o acesso a banco de dados via SQL.

3.2.6. Camada de Apresentação

A principal finalidade dessa camada é a de definir formatos de dados, como textos ASCII e EBCDIC, binário, BCD e JPEG. Em outras palavras, a camada de apresentação define como tudo deve ser representado, formatado e compactado. A criptografia também é definida pela OSI como um serviço da camada de apresentação, mas, como no caso do controle de sessão, essa função é atribuída à aplicação.

Alguns exemplos da camada de apresentação: JPEG; ASCII; EBCDIC; TIFF; GIF; PICT; MPEG; MIDI.

3.2.7. Camada de Aplicação

Uma aplicação que se comunica com outros computadores está implementado os conceitos referentes à camada de aplicação do modelo OSI. A camada de aplicação se refere aos serviços de comunicação para aplicativos, em outras palavras a camada de aplicação é o próprio aplicativo e suas regras (protocolos) de comunicação.

Alguns exemplos de aplicativos/protocolos da camada de apresentação:

- Telnet;
- HTTP;
- FTP;
- Navegadores web;
- NFS;
- Gateways SMTP (clientes de e-mail);
- SNMP.

4. Metodologia

4.1 Comunicação em socket

Tudo que se comunica com algo fora do próprio computador faz-se necessário o uso de um tipo de programação específico chamado socket. Todas as linguagens de programação (que são capazes de implementar algo on-line) tem esse tipo de programação como por exemplo C/C++, Delphi, PHP, Perl, Pascal, até Assembly.

A partir desse conhecimento, pode-se facilmente implementar servidores/clientes, backdoors, portscanner, e quase tudo relacionado a internet.

4.2 Utilização no projeto

Inicialmente, há a implementação de dois programas, um para enviar e outro para receber uma string qualquer. Para isso, utiliza-se o conceito de cliente e servidor, a partir da comunicação via socket. Esse tipo de comunicação usado comumente em redes de computadores cria um canal pelo qual trafegam dados para controlar determinado processo no próprio sistema operacional ou remotamente (Figura 23). Na criação do socket há a combinação do endereço IP e o número de uma porta, o que resulta no protocolo TCP.

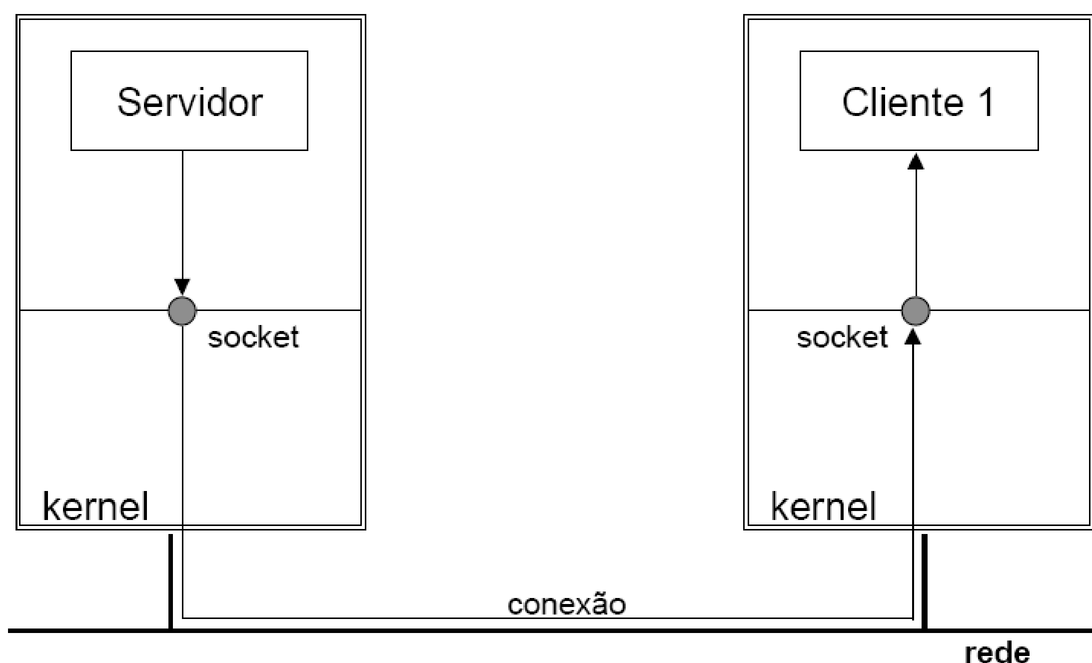


Figura 23 - Esquema de conexão socket

4.3. Servidor

Para implementação do software foi usado o DEV C++, disponível no site do bloodshed[1]. Uma importante observação que diz respeito a tal plataforma de programação é que para compilação de códigos com o uso de bibliotecas socket é necessário a adição do comando *-lwsock32* nas opções do compilador

(Figura 24). Deve-se seguir o seguinte caminho: Clicar em Tools, Compiler Options, ative a checkbox que diz Add the following commands when calling compiler, e digite o seguinte comando *-l wsock32*. Esse procedimento faz com que o compilador faça um link com a biblioteca socket, sem o qual não é possível prosseguir com a execução do código do programa que contém algum comando de criação ou manipulação de tais estruturas.

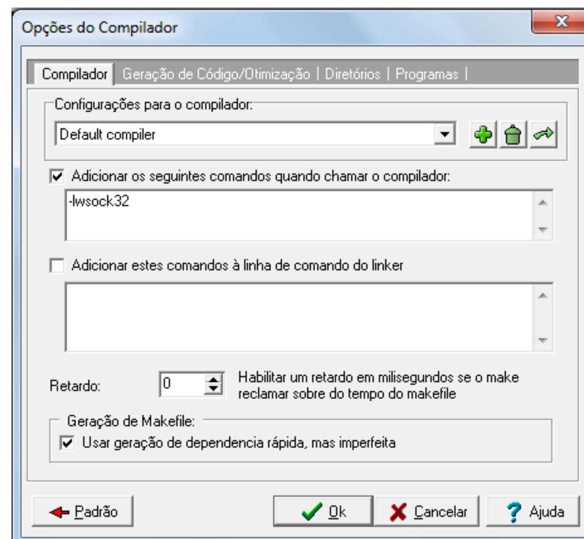


Figura 24 - Menu de opções do compilador do software DEV C++

Na internet tudo tem sua família e seu respectivo protocolo. As famílias mais usadas são:

AF_INET (Arpa Internet Protocols) – é a família mais usada atualmente e que abrange a utilização dos protocolos TCP/IP. Criada pela Advanced Research Projects Agency, agência militar americana, essa família surgiu com o desenvolvimento da ARPANet antecessora da atual internet, que permitia cientistas, investigadores e militares comunicarem-se utilizando correio eletronico ou conversas em tempo-real.

AF_UNIX (Unix Internet Protocols)

AF_ISSO (Iso Protocols)

AF_NS (Xerox Network System Protocols)

E os protocolos são baseados no TCP (SOCK_STREAM) - Transmission Control Protocol, que, como mencionado anteriormente, verifica se os dados são enviados de forma correta, na sequência apropriada e sem erros, pela rede - e UDP (SOCK_DGRAM) - User Datagram Protocol – podendo ser os seguintes:

0 - IP - Internet Protocol

1 - ICMP - Internet Control Message Protocol
2 - IGMP - Internet Group Multicast Protocol
3 - GGP - Gateway-Gateway Protocol
6 - TCP - Transmission Control Protocol
17 - UDP - User Datagram Protocol

Para se começar a programar em sockets em C é necessário declarar a biblioteca winsock.h .

Algo exclusivo de programas desse tipo é a estrutura *WSADATA* e a função WSAStartup() no início do código. Essa função inicia o *Windows Sockets Dynamic Link* (WinSock DLL) e também é usada para confirmar sua versão,ou seja, se comunica com o Windows (sistema operacional usado no projeto) e determina a inicialização de *dll's* e *ocx's* responsáveis por comunicação na internet.

```
int WSAStartup(
__in   WORD wVersionRequested,
__out  LPWSADATA lpWSAData
);
```

O termo wVersionRequested é número da maior versão que a aplicação pode usar.Para isso é usado a função MAKEWORD (lowbyte, highbyte) onde lowbyte especifica a maior versão que pode ser usada e highbyte especifica a menor. Ela retorna um valor do tipo WORD com esses dados para ser usado na função WSAStartup. Diferentes versões podem ser usadas como 1.0 , 1.1 , 2.0, 2.1, 2.2 .

O parâmetro *lpWSAData* é um ponteiro para a estrutura do tipo WSADATA que recebe detalhes da implementação dos sockets.

Se a função foi executada normalmente, retorna zero. Caso ocorra algum erro, pode retornar um dos seguintes parâmetros:

Código do erro	Significado
WSASYSNOTREADY	Subsistema da rede adjacente não está pronto para comunicação
WSAVERNOTSUPPORTED	O versão do Socket requisitada não está de acordo com a implementação
WSAEINPROGRESS	Operação de bloqueio de Sockets 1.1 está ativa
WSAEPROCLIM	O número limite de tarefas suportadas pela

	implementação em socket foi alcançado
WSAEFAULT	O parâmetro lpWSAData não é um ponteiro válido

As estruturas acima podem ser usadas da sequinte forma:

WSADATA wsa_data;
WSAStartup(MAKEWORD(2, 0), &wsa_data);

A primeira linha se trata da declaração da variável *wsa_data* usada logo abaixo para inicialização da comunicação com a internet pela função WSAStartup( ), que está usando a versão de socket 2.0.

Outro passo importante é a criação do socket, sendo usado para isso a função *Socket( )* :

SOCKET WSAAPI socket(int af, int type, int protocol)

O parâmetro af especifica o "address family" que este socket usa. Pode-se usar qualquer um dos formatos ou famílias descritas anteriormente:

AF_INET: Arpa Internet Protocols
AF_UNIX: Unix Internet Protocols
AF_ISSO: Iso Protocols
AF_NS: Xerox Network System Protocols

Obs: A versão 1.1 do WinSock só suporta o formato AF_INET.

O parâmetro type especifica o tipo do socket, que podem ser:

SOCK_STREAM: tipo TCP
SOCK_DGRAM: tipo UDP

Se no campo protocol for deixado o valor 0 (zero), isto indica que o socket irá usar o valor padrão. O campo protocol pode ser o padrão porque a combinação do AF_INET+SOCK_STREAM já indica que o protocolo é TCP.

Alguns valores possíveis para o protocol:
IPPROTO_IP: Internet Protocol (0)
IPPROTO_ICMP: Internet Control Message Protocol (1)
IPPROTO_IGMP: Internet Group Multicast Protocol (2)
IPPROTO_GGP: Gateway-Gateway Protocol (3)
IPPROTO_TCP: Transmission Control Protocol (6)
IPPROTO_UDP: User Datagrama Protocol (17)

A função irá retornar o socket descriptor, que é um número que identifica o socket criado. Se falhar irá retorna INVALID_SOCKET, e pode-se usar a função WSAGetLastError() para saber mais detalhes do erro.

A próximo passo é a identificação do socket no sistema. Para isso é usada a função *bind( )* :

*int bind(SOCKET s, const struct sockaddr *addr, int namelen)*

O parâmetro s é o socket descriptor retornado pela função socket(). O parâmetro addr é um ponteiro para uma estrutura do tipo sockaddr. O namelen é o tamanho em bytes da estrutura, retornado pela função sizeof().

Antes de usar a função bind() é necessário conhecer as estruturas sockaddr, sockaddr_in e in_addr.

Definições da estrutura sockaddr:

```
struct sockaddr
{
   u_short sa_family;
   char sa_data[14];
};
```

O item sa_data da estrutura vai depender do "address family". No WinSock 1.1, apenas o AF_INET é suportado, logo, somente um "endereçamento de internet" é suportado no sa_data.

Definições da estrutura sockaddr_in:

```
struct sockaddr_in
{
   short sin_family; // Familia
   u_short sin_port; // Porta
   struct in_addr sin_addr; // Endereço
   char sin_zero[8];
};
```

Obs: Antes de atribuir cada especificação a estrutura, é necessário atribuir o valor zelo a ela. Isso é feito da seguinte forma:

```
memset(&local_address, 0, sizeof(local_address));
```

Nem todos os computadores armazenam os dados na mesma ordem na memória. Existem computadores que trabalham no formato "Big Endian", onde byte menos significativo fica no endereço de memória de maior valor. Por exemplo, para armazenar o número 0x01234567 ficaria assim:

endereço 0x101: 01
endereço 0x102: 23
endereço 0x103: 45
endereço 0x104: 67

O maior endereço é o 0x104 e o byte menos significativo de 0x01234567 é o último da direita (67).

Já nos computadores "Little Endian", ocorre o inverso, o byte menos significativo fica no endereço de memória de menor valor:

endereço 0x101: 67
endereço 0x102: 45
endereço 0x103: 23
endereço 0x104: 01

O menor endereço é o 0x101 e o byte menos significativo de 0x01234567 é o último da direita (67).

Os computadores com processadores baseados no Intel x86 são "Little Endian", mas a ordem dos bytes na rede (network) é "Big Endian". Precisa-se converter os dados antes de enviá-los usando as funções htons() e htonl():

htons(): converte um unsigned short (host-to-network)
htonl(): converte um unsigned long (host-to-network)

Voltando a estrutura sockaddr_in, o item sin_addr é definida da seguinte forma:

```
struct in_addr
{
   union
   {
      struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
      struct { u_short s_w1,s_w2; } S_un_w;
      u_long S_addr;
   } S_un;
   #define s_addr S_un.S_addr
   #define s_host S_un.S_un_b.s_b2
   #define s_net S_un.S_un_b.s_b1
   #define s_imp S_un.S_un_w.s_w2
   #define s_impno S_un.S_un_b.s_b4
   #define s_lh S_un.S_un_b.s_b3
};
```

É importante destacar os *defines* que ela possui, pois pode-se acessar o dados da estrutura através deles. Quando se acessa local_address.sin_addr.s_addr na realidade é acessado local_address.sin_addr.S_un.S_addr.

O valor indicado para local_address.sin_addr.s_addr é htonl(INADDR_ANY), isto indica que seram usados todos os endereços locais designados ao servidor. Por exemplo, se a máquina possui duas placas de rede (192.168.0.1 e 192.168.0.2), pode-se usar os dois endereços para o socket. Para especificar apenas uma usa-se a função inet_addr().

Sobre o item sin_zero de sockaddr_in, é usada para completar a estrutura, de modo que tenha o mesmo tamanho (em bytes) da estrutura sockaddr. Assim pode-se fazer "casts" de uma estrutura para outra (utilizado no parâmetro da função bind()). Por isso a estrutura deve ser preenchida com zeros antes de ser utilizada.

A função bind() retornará 0, caso contrário retornará SOCKET_ERROR, usa-se o WSAGetLastError() para analisar o erro.

listen()

A função listen() habilita o socket para receber conexões dos clientes. Seria como habilitar o socket para receber conexões dos clientes.

int listen(SOCKET s, int backlog)

O parâmetro s é o socket descriptor, mencionado anteriormente . O backlog indica quantas conexões pendentes o socket pode deixar na fila para serem processadas, quando as conexões são aceitas elas são removidas da fila. O mínimo para o backlog é 1.

A função retornará 0, caso contrário retornará um SOCKET_ERROR

accept()

O accept() aceita a conexão quando ela é detectada.

SOCKET accept(SOCKET s, struct sockaddr* addr, int* addrlen)

O parâmetro s é o socket descriptor. O parâmetro addr é um ponteiro para uma estrutura do tipo sockaddr (igual ao da função bind()), na qual a função irá armazenar a o endereço (estrutura) da entidade (cliente) que requisitou a conexão. Há aplicações que isso não é necessário, podendo-se atribuir NULL ao campo.

O parâmetro addrlen é um ponteiro onde a função irá colocar o tamanho em bytes da estrutura addr recebida do cliente. Se for atribuido um valor NULL para o parâmetro addr então o addrlen retornará NULL para o ponteiro. Pode-se também atribuir NULL ao parâmetro do addrlen.

A função accept() irá retornar um socket descriptor do cliente. Se der algum problema será retornado um INVALID_SOCKET.

recv()

O recv() recebe os dados de uma conexão.

```
int recv(SOCKET s, char* buf, int len, int flags);
```

O parâmetro s é o socket descriptor do cliente. O parâmetro buf é o buffer onde serão armazenadas as informações recebidas. O len é o tamanho deste buffer.

O parâmetro flags indica o modo como serão recebidos os dados, que será deixado em zero, ou seja, o valor default.

O recv() irá retornar o total de bytes recebidos. Ele retorna 0 (zero) quando a conexão é fechada normalmente. Se ocorrer erro ele retorna um SOCKET_ERROR.

Foi usada também a função inet_ntoa() que converte um endereço de Internet (IPv4) em uma string do endereço formatado nos padrões de Internet com pontos decimais. Ela já retorna em "Little Endian".

O recv() ficará dentro de um loop, antes de receber uma mensagem o buffer (variável message no código) deverá ser limpo. Após receber a mensagem, ela é exibida. O loop só pára quando o cliente enviar uma mensagem "#quit".

Deve-se finalizar a aplicação com as funções WSACleanup() e closesocket().

connect()

A função connect() irá conectar o cliente com o servidor.

```
int connect(SOCKET s, const struct sockaddr* name, int namelen);
```

Ele recebe parâmetros iguais ao da função bind(). Se der algum erro ele retorna um SOCKET_ERROR.

A função connect() tem o mesmo problema da função accept(), enquanto ele estiver tentando conectar ficará travado.

send()

O comando send() é a função responsável por enviar informações ao servidor.

```
int send(SOCKET s, const char* buf, int len, int flags);
```

Os parâmetro que ele recebe também são iguais ao da função recv(). O send() retonará o número de bytes enviados, esse número não será maior do len (e ele não enviará mais o que o especificado em len).

É importante salientar a importancia de limpar o buffer antes do envio de qualquer mensagem, caso contrario resquícios de mensagens anteriores podem ser também enviados.

A figura 25 mostra o fluxograma dos comandos.

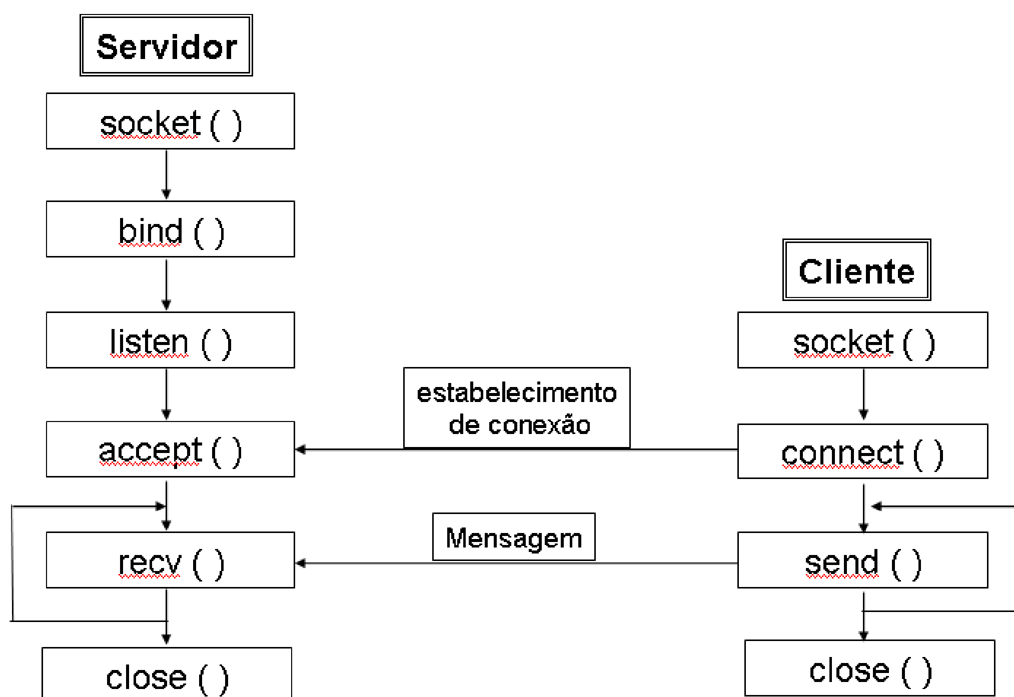


Figura 25 - Fluxograma de comandos

A figura 26 mostra um teste de envio de mensagem da primeira interface

grafica:

```
F:\Servidor-Enviando_Msgs.exe
Porta local: 8080
aguardando alguma conexao...
conexao estabelecida com 192.168.0.104
aguardando mensagens...
192.168.0.104: Olá
192.168.0.104: Como vai?
```

```
F:\Cliente-Enviando_Msgs.exe
IP do servidor: 192.168.0.104
Porta do servidor: 8080
conectando ao servidor 192.168.0.104...
digite as mensagens
msg: Olá
msg: Como vai?
msg:
```

Figura 26 - Interface de comunicação do programa implementado em socket

Pode-se observar o código do servidor no Apendice A.

4.4 Concatenar e Desmembrar Dados

A partir da conexão estabelecida, detalhada acima, o envio mensagens pode ser executado. Porém, o projeto requer a atualização contínua de pelo menos 15 dados, o que inviabiliza a transmissão serial. Para contornar esse problema, é proposta a união dos 15 dados em uma única mensagem, sendo definido um protocolo de comunicação.

4.4.1 Concatenar

Primeiramente, como os dados que serão enviados estão em radiano, e muitas vezes compostos por vírgula, optou-se transformá-los em números inteiros, numa escala entre 0 e 65536, para os valores entre 0 e 90º. Isso é justificado por manter uma precisão de 16 bits (1/65536 = 0,0000152) com apenas 5 caracteres ao invés de 7. Por exemplo:

O angulo 30º corresponde à 0,52539 radianos (precisão de 0,0001), número o qual contém 7 caracteres (contado com a vírgula).
Colocando esse número na escala descrita acima, transforma-se em 21845,3 , o qual arredonda-se para 21845, que contém apenas 5 caracteres.

Para verificar o efeito dessa aproximação, o número 21845 é reconvertido em radianos, sendo encontrados o valor 0,52359 ,ou seja, 29,99954º

O erro dessa aproximação pode ser encontrado da seguinte forma:

$$Erro = (30-29,99954)/30 = 0,0015\%$$

Esse erro pode ser considerado desprezível pelo valor reduzido. Como há 15 dados a serem enviados, a diminuição de 2 bytes em cada informação, reduz 30 bytes na mensagem, o que é considerável na comunicação em tempo real.

Como os dados inicialmente estão no formato int, e o envio de informações é feito no formato string, deve-se fazer a conversão antes de concatená-los. Para isso é usada a função *fprintf().*

Essa função é idêntica a *printf()*, porém, ao invés de retornar o valor para a tela ela envia para um buffer. Em outras palavras, sprintf monta uma string formatada no buffer indicado, utilizando o estilo de formatação de printf. Exemplo:

```
char buf[256];
sprintf(buf, "Formatando : %d, %c, %f\n",10,'a',10.09');
printf(buf);
```

Depois deste código a string 'buf' conterá:

Formatando : 10, a, 10.09

O próximo passo para manipular os dados já convertidos em string, é adicionar um separador, o qual posteriormente será usado para desmembrar a mensagem. O caracter escolhido foi “#”.

A adição desse separador é feita através da função *strcat ()*, que recebe a seguinte argumentação:

```
void strcat(char *destino, char *fonte);
```

Sua execução copia a fonte para após o final de destino. Ou seja, acrescenta o conteúdo de fonte à string destino. Não aloca memória alguma. Exemplo:

```
char p[30];
strcpy(p, "palavra");
strcat(p, " nova");
```

Agora p contém a string "palavra nova".

Os números já convertidos para a escala descrita acima e já com o símbolo pré-determinado, são unidos com o comando *strcat ()*, torando-se uma única mensagem.

Para a posição inicial tem-se a formação do seguinte pacote:

0#0#0#0#0#0#0#0#0#0#0#0#0#0#

O Codigo para concatenar a mensagem pode ser visto no Apêndice B.

4.4.2 Desmembrar

Partindo do pressuposto que a mensagem recebida é composta por algarismos separados pelo caracter “#”, o desmembramento dessa mensagem em campos de um vetor pode ser realizado a partir da função *strtok().*

```
char * strtok ( char * str, const char * delimiters )
```

Essa função, primeiramente, recebe uma string e um char como delimitador. Na primeira vez que é executada, verifica o vetor do índice zero até o delimitador, retornado seu conteúdo. Posteriormente, o endereço do caracter após o delimitador é atribuído a um ponteiro como inicio da próxima

mensagem, sendo realizada assim uma nova busca, e assim sucessivamente até a função completar a varredura na string, retornando NULL.

Dessa forma, desmembra-se a mensagem recebida em 15 dados de posição diferentes, sendo imediatamente convertidos em float(já que estavam anteriormente em string) pela função *atof()*. Além disso, é preciso reconverter os dados para radiano, já que estão na escala descrita acima. Utiliza-se o seguinte procedimento:

Rad = (Dado_Recebido*PI)/(65536*2)

.

5. Simulador

Criado pelo Departamento de Ciência da Computação da Universidade de Columbia, o software GRASPIT, foi criado para ser uma ferramenta de pesquisa para manipuladores robóticos.

A partir de modelos de mãos robóticas previamente disponíveis em seu banco de dados, é possível criar um cenário completo do ambiente desejado por meio da inclusão, objetos e obstáculos normalmente encontrados em ambientes humanos. A partir dos arquivos que descrevem o sistema robótico e os itens do cenário o software é capaz de analisar a configuração dos dedos da mão robótica para agarrar um determinado objeto e resultar em um índice de qualidade para aquele processo de manipulação específico. Além da sua eficiência, o simulador também contém ferramentas para a detecção de colisões entre o objeto a ser manipulado e os dedos da mão robótica utilizada. Adicionalmente, o sistema de simulação também permite escolher as propriedades físicas do material que descreve o objeto.

Uma vantagem atrativa que diferencia o simulador *GraspIt!* em relação aos demais ambientes de trabalho de mesma categoria consiste no fato de que o simulador pode importar uma plataforma robótica completa e um modelo do ambiente no qual o sistema robótico atuará. Isto permite um planejamento mais preciso em aplicações que envolvem o processo de agarrar um objeto em cenários mais próximos aos reais.
Outra característica que os modernos simuladores de sistemas robóticos não apresentam são a habilidade de modelar de forma precisa o contato e o modelo para o atrito existente entre o objeto e os dedos da mão robótica. Especificamente, o processo de agarrar objetos empregando mãos robóticas envolve o estabelecimento e o rompimento de contatos entre os segmentos articulados dos dedos da mão robótica e um ou mais objetos no ambiente. Assim como ocorre com outros sistemas de simulação, um simulador desta natureza deve permitir que o projetista encontre condições para criar protótipos e diferentes testes do projeto da mão robótica.
Outra vantagem obtida na utilização deste simulador consiste na possibilidade de desenvolver uma elevada quantidade de experimentos com a alteração de algumas condições e parâmetros do sistema sem necessitar construir o protótipo real.
O simulador GraspIt! pode importar uma grande variedade de diferentes protótipos de robôs e mãos robóticas bem como modelos de ambientes com objetos de uso cotidiano. Todos esses elementos contam com a possibilidade de manipulação dentro de um espaço de trabalho virtual em três dimensões.

Algumas características importantes do simulador Graspit! são:

- Uma biblioteca de robôs que inclui diversos modelos de mãos robóticas, um braço Puma e uma base móvel simplificada;

- Uma definição flexível de robôs que torna possível a importação de novos projetos de robôs;

- A habilidade de conectar robôs para a construção de uma plataforma de manipulação;

- A habilidade de importar modelos de obstáculos para construir um ambiente de trabalho completo para os robôs;

- Uma interface interativa intuitiva, bem como uma interface externa para o MATLAB;

- Um sistema rápido de detecção de colisão e também determinação de contatos;

- Rotinas de análises de *fixação* que calculam a qualidade de uma *fixação* durante a sua simulação;

- Métodos de visualização que podem mostrar o ponto fraco de uma fixação e criar projeções do *grasp wrench space*;

- Um sistema dinâmico que calcula os movimentos do robô e do objeto sob a influência de forças externas e de contatos;

- Gerador de trajetórias simples e algoritmos de controle que calculam as forças nas juntas necessárias para permitirem a trajetória.

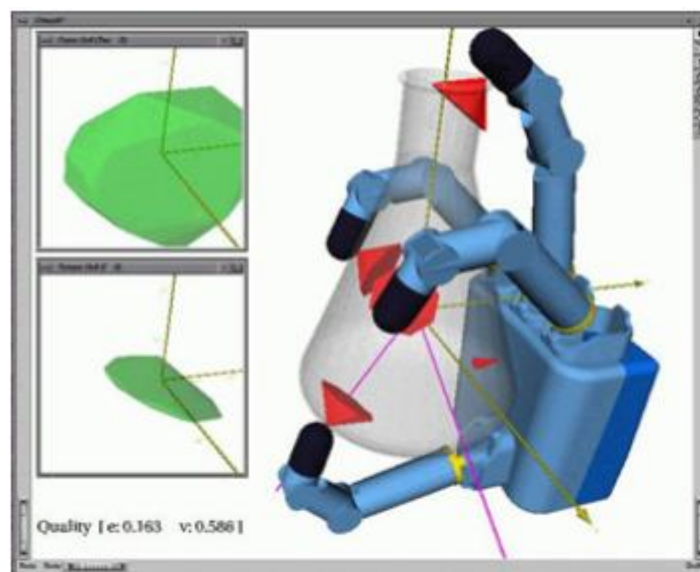


Figura 27 - Interface do GRASPIT

Uma importante característica desse software é sua integração com o MATLAB, programa que comanda a mão robótica,o que deixa mais realista os testes realizados nesse ambiente. Alguns exemplos dos comandos em MATLAB para o GRASPIT são: getRobotName, getDOFVals, moveDOFs.

5.1 getRobotName

Como descrito em seu próprio nome, e função getRobotName retorna o nome do robô utilizado na simulação, usando-se como entrada o numero atribuído ao robô, como por exemplo:

>> getRobotName(1)

ans =

'Kanguera'

5.2 getDOFVals

Essa função retorna um vetor do tamanho do número dos graus de liberdade do manipulador. Cada componente do vetor é o posicionamento, em radianos, do respectivo grau de liberdade, como é mostrado na figura abaixo. Exemplo:

>> d = getDOFVals(1)

d =

```
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
    1.5700
```

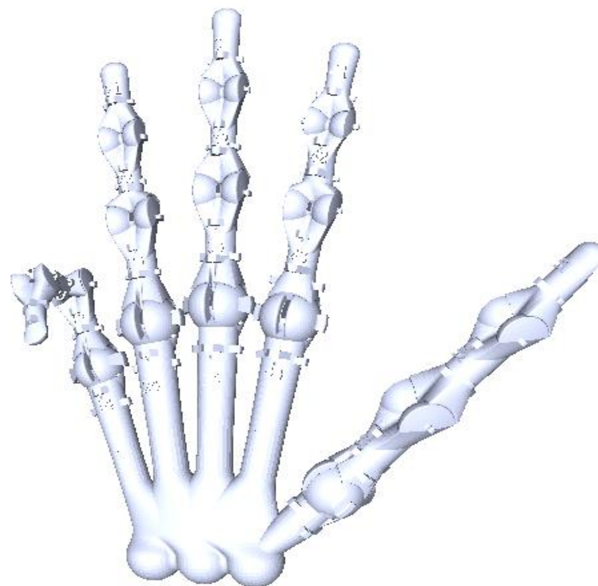


Figura 28 - Reprodução de gesto

5.3 moveDOFs

O comando moveDOFs, executa a ação inversa do comando anterior, ou seja, disponibiliza-se a posição dos graus de liberdade desejada, e o manipulador executa esse movimento. Tem-se a estrutura:

newD = moveDOFs(r,D,deltaD)

A função recebe 3 argumentos:

r: é o índice do robô que deseja-se mover, ou seja, sua identificação no GRASPIT

D: é o vetor de posições desejadas. Um erro é retornado se o número de elementos de D não corresponder ao número de graus de liberdade do robô r

deltaD: é o vetor contendo o tamanho do passo que deve ser executado quando o robô é movido para a posição desejada. Como no vetor D, o número de elementos de deltaD deve ser igual ao número de graus de liberdade do robô.

Essa função retorna o vetor de posições no final da movimentação.

Exemplo:

A mão encontra-se na posição inicial como mostrado na figura 29(todas as coordenadas na posição zero). Executando o comando moveDOFs, com a inteção de mover o dedo indicador 90º (ou 1,57 radianos) com apenas um passo, temos que atribuir valores aos vetores da seguinte forma:

D = [0 0 0 0 0 0 1.57 0 0 0 0 0 0 0 0 0 0 0 0]
deltaD = [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

Se a mão for identificada com o número 1, basta executar o comando moveDOFs(1,D,deltaD) na linha de código do MATLAB

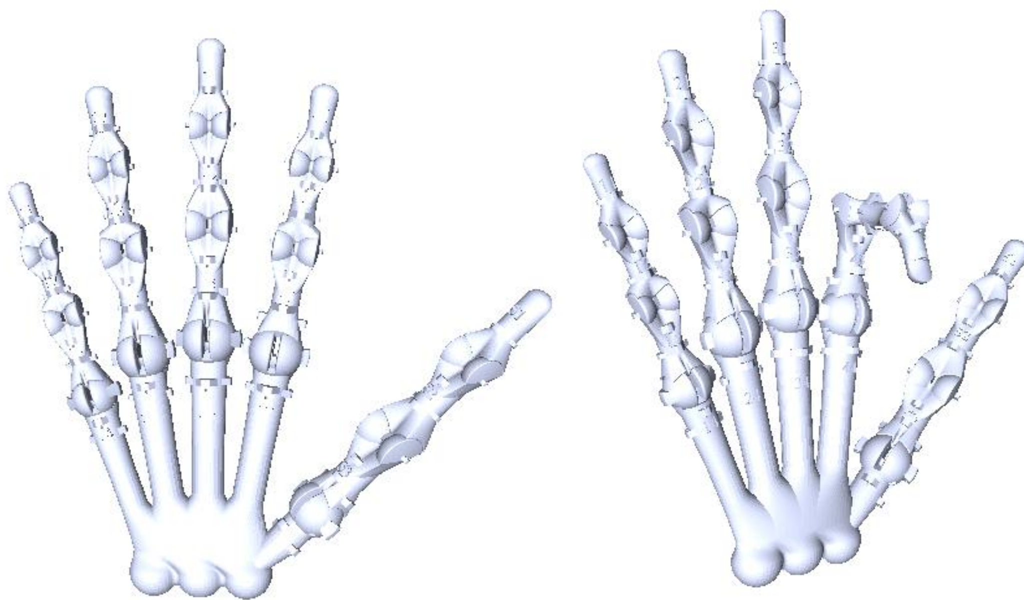


Figura 29 - Posição Inicial e Final

Dedo5			Dedo4			Dedo3			Dedo2			Dedo1		
P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15

Obs: As posições 3, 6, 9, 12 e 15 são referentes
a movimentos laterais

Figura 30 - Imagem com a mão real mostrando a convenção dos ângulos

A figura 31 mostra os comandos usados para o controle do simulador Graspit:

Current Directory - C:\Documents and Settings\Administrador ...

All Files	File Type	Last Modified
cliente.c	C Source file	21/08/2009 12:35:43
cliente.dll	MEX-file	21/08/2009 12:35:51
computeNewVeloci...	C Source file	09/10/2007 14:38:54
computeNewVeloci...	MEX-file	09/10/2007 14:38:54
computeNewVeloci...	M-file	09/10/2007 14:38:54
connectToServer.c	C Source file	09/10/2007 14:38:54
connectToServer.dll	MEX-file	09/10/2007 14:38:54
connectToServer.h	C Header file	13/04/2006 16:58:42
contents.m	M-file	09/10/2007 14:38:54
Envia_posicao.mdl	Model	08/06/2009 08:29:38
Envia_posicao_OK....	Model	13/08/2009 17:35:56
getAverageContact...	C Source file	13/04/2006 16:58:42
getAverageContact...	MEX-file	09/10/2007 14:38:54
getAverageContact...	M-file	09/10/2007 14:38:54
getBodyName.c	C Source file	13/04/2006 16:58:42
getBodyName.dll	MEX-file	09/10/2007 14:38:54
getBodyName.m	M-file	09/10/2007 14:38:54
getContacts.c	C Source file	13/04/2006 16:58:42
getContacts.dll	MEX-file	09/10/2007 14:38:54
getContacts.m	M-file	09/10/2007 14:38:56
getDOFVals.c	C Source file	13/04/2006 16:58:42
getDOFVals.dll	MEX-file	09/10/2007 14:38:56
getDOFVals.m	M-file	09/10/2007 14:38:56
getRobotName.c	C Source file	09/10/2007 14:38:56
getRobotName.dll	MEX-file	09/10/2007 14:38:56
getRobotName.m	M-file	09/10/2007 14:38:56
Grava_posicao.m	M-file	13/08/2009 15:11:08
GravaPosicao.mdl	Model	23/05/2009 21:39:44
GravaPosicao_OK....	Model	13/08/2009 14:31:00
moveDOFs.c	C Source file	09/10/2007 14:38:56
moveDOFs.dll	MEX-file	09/10/2007 14:38:56
moveDOFs.m	M-file	09/10/2007 14:38:56
moveDynamicBodie...	C Source file	09/10/2007 14:38:56
moveDynamicBodie...	MEX-file	09/10/2007 14:38:56
moveDynamicBodie...	M-file	09/10/2007 14:38:56
Obs_CPD.txt	TXT File	13/08/2009 15:23:40
PDController.m	M-file	09/10/2007 14:38:54
PDHandController.m	M-file	09/10/2007 14:38:54
Recebe_posicao.m	M-file	08/06/2009 08:48:44
render.c	C Source file	09/10/2007 14:38:56
render.dll	MEX-file	09/10/2007 14:38:56
render.m	M-file	09/10/2007 14:38:56
servidor.c	C Source file	21/08/2009 12:33:24
servidor.dll	MEX-file	21/08/2009 12:34:04
servidor_1.mdl	Model	08/06/2009 08:55:34
servidor_1_OK.mdl	Model	13/08/2009 14:27:36
setDOFForces.c	C Source file	09/10/2007 14:38:56
setDOFForces.dll	MEX-file	09/10/2007 14:38:56
setDOFForces.m	M-file	09/10/2007 14:38:56
setDOFVals.c	C Source file	13/04/2006 16:58:42
setDOFVals.dll	MEX-file	09/10/2007 14:38:56
SFB__ghdf__SFB....	MAT-file	23/05/2009 17:21:12
stepDynamics.c	C Source file	13/04/2006 16:58:42
stepDynamics.dll	MEX-file	09/10/2007 14:38:56
teste1.m	M-file	22/05/2009 20:55:28

Current Directory | Workspace

Figura 31 - Comandos GRASPIT

6. S-Function

Como o programa destinado a transmissão de dados foi implementado na linguagem C, e o simulador é comandado pelo MATLAB, foi necessário buscar um procedimento para unir essas duas vertentes.

O modo encontrado foi usar o Simulink, modulo MATLAB voltado para simulação, que usa programação orientada a objetos, ou seja, a programação é feita por blocos e conexões. Uma de suas ferramentas destina-se a utilização de algoritmos em C, que inserida nesse bloco, passa a fazer parte da simulação.

O bloco utilizado é chamado de S-Function, e tem uma estrutura padrão na qual deve ser inserido o código em C a ser utilizado. Esse código, pode ser observado no Apêndice C. O bloco da função pode ser encontrado no menu mostrado pela figura 32.

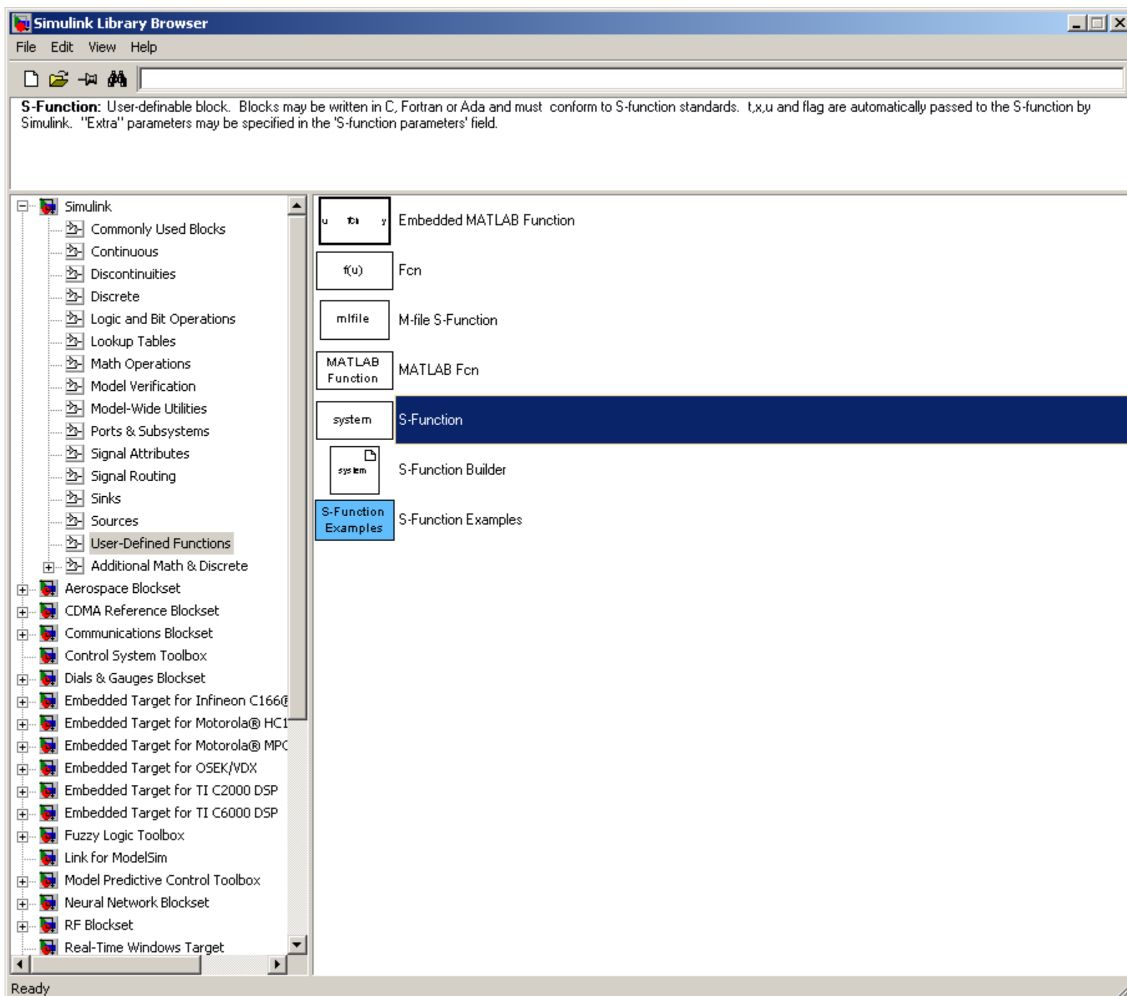


Figura 32 - Menu que contém S-function

S-function (system-function) é um poderoso mecanismo para extender as capacidades do Simulink. É um modo de utilizar programação não orientada a objetos nos blocos de simulação, usando as linguagens C,C++,Ada, Fortran ou

a própria linguagem MATLAB, que gera arquivos .m , sendo compilados utilizando-se o comando mex, que faz o link entre o arquivo gerado e o MATLAB quando necessário.
É utilizada uma sintaxe especial que permite a interação com os equation solvers do Simulink, e por ter uma estrutura bastante generalista suporta modelos contínuos, discretos e até mesmo híbridos.

6.1 Usando S-Functions em Modelos

Para adicionar o bloco S-function ao modelo é necessário apenas arrasta-lo da biblioteca User-Defined Functions para a tela e especificar seu nome, como ilustrado na figura 33:

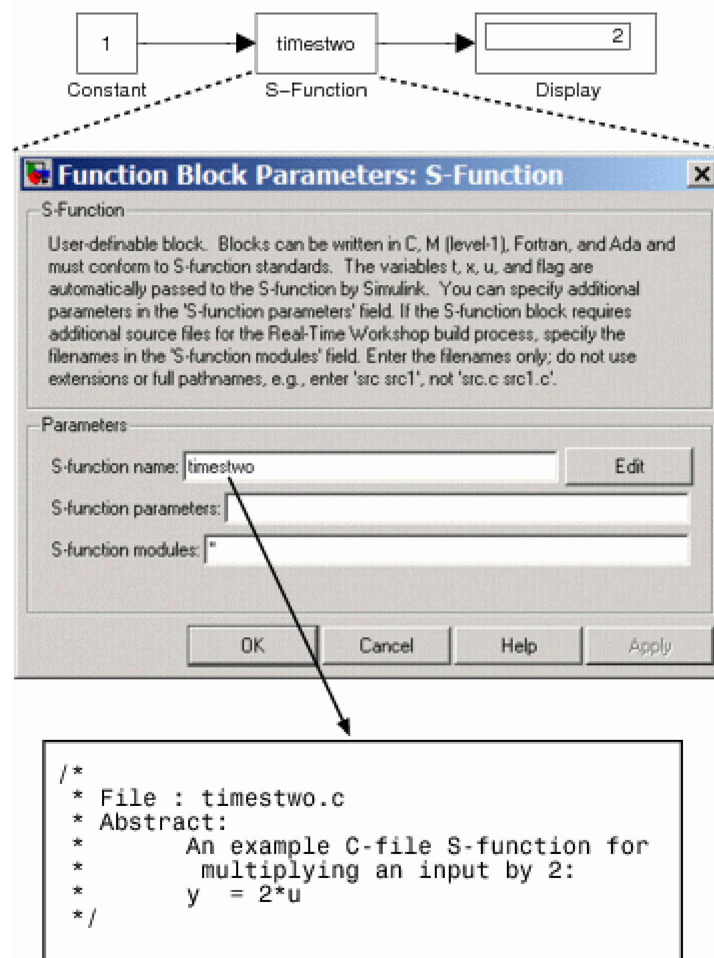


Figura 33 - Procedimento para utilizar S-Function

6.2 Passando Parametros para S-functions

Nesse bloco, pode-se especificar parâmetros a partir da caixa de dialogo aberta com clique duplo na função em questão. Devem ser separados por vírgula, na ordem requisitada pelo código fonte. A Figura 34 ilustra o exemplo

limintm que tem como parâmetros o limite inferior(2), superior(3) e a condição inicial(2.5) da função

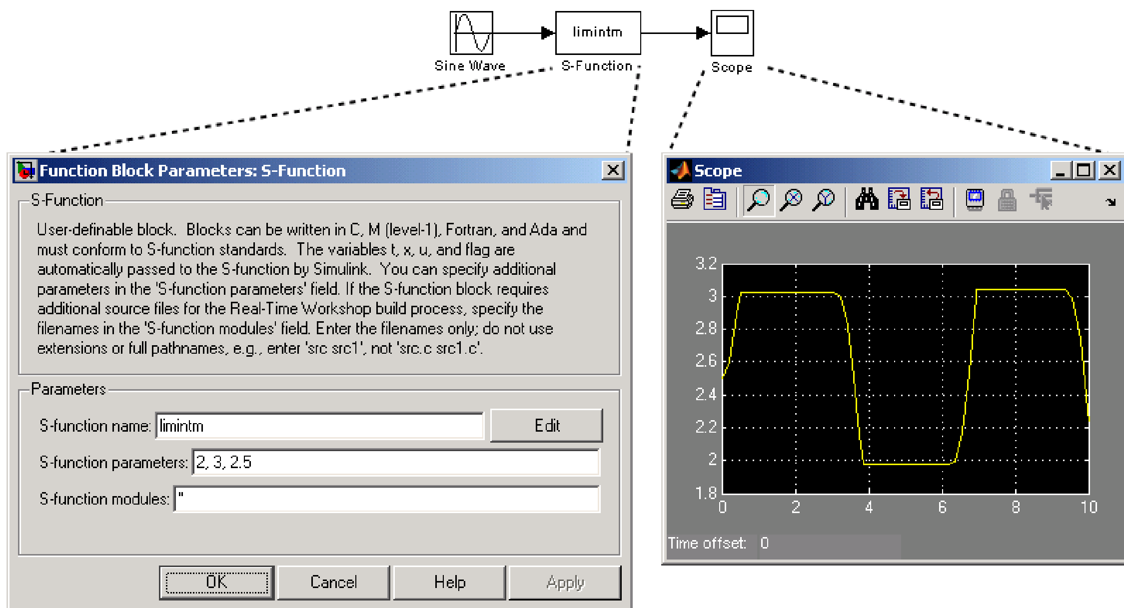


Figura 34 - Exemplo de Passagem de Parâmetros

6.3 Como S-functions Funcionam

Para entender como S-Functions trabalham, primeiramente, é necessário entender como o Simulink simula um modelo, e para isso tem-se observar a dinâmica dos blocos,suas entradas, saídas e interações.

6.4 Matemática dos Blocos Simulink

Um bloco simulink consiste em entradas, estados, e saídas, onde as saídas são funções do tempo de simulação, entradas e estados.

As seguintes equações descrevem o relacionamento entre as variáveis:

$$y = f_0(t, x, u) \qquad \text{(Outputs)}$$

$$\dot{x}_c = f_d(t, x, u) \qquad \text{(Derivatives)}$$

$$x_{d_{k+1}} = f_u(t, x_c, x_{d_k}, u) \qquad \text{(Update)}$$

$$\text{where} \quad x = [x_c; x_d]$$

6.5. Estágios de Simulação

A execução do modelo do simulink ocorre em estágios. Primeiramente tem-se a fase de iniciação, a qual incorpora as bibliotecas ao modelo, define os tempos de amostragem, avalia a passagem de parâmetros,determina a ordem de execução dos blocos e aloca memória. Após isso, o Simulink entra no loop de simulação, onde para cada passo são calculados os blocos de estado, derivadas e saídas para o tempo de amostragem.
A figura 35 ilustra os estágios de simulação:

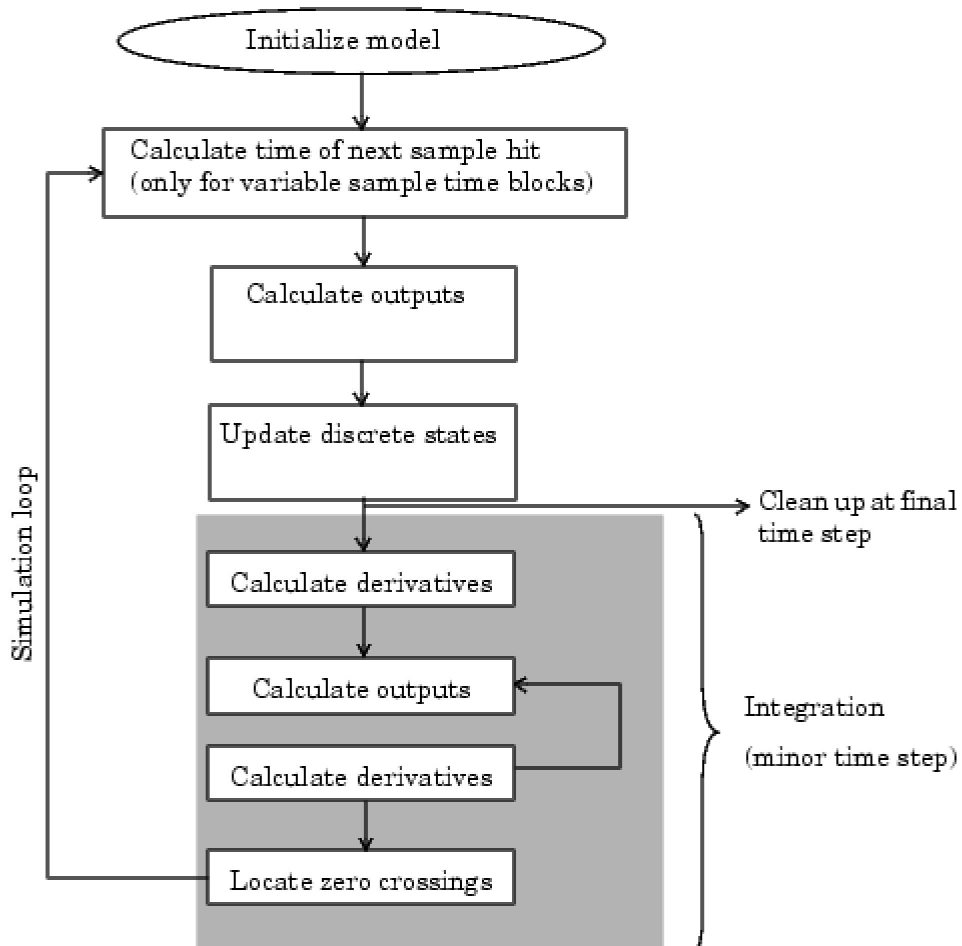


Figura 35 - Estágios de Simulação Simulink

6.6 M-File

Uma S-function M-file consiste em uma função da seguinte forma:

[sys,x0,str,ts]=f(t,x,u,flag,p1,p2,...)

Onde f é o nome da s-function, t é o tempo atual, x é o vetor de estados correspondente ao bloco, u é a entrada do bloco, flag indica a tarefa a ser

executada e p1, p2... são parâmetros do bloco. Durante a simulação, o simulink chama a função f e usando o flag, especifica a tarefa a ser executada.
A tabela abaixo lista os estágios de simulação e os seus flags:

Estágio da Simulação	Rotina S-function	Flag
Inicialização	mdlInitializeSizes	Flag=0
Calculo da proxima amostra	mdlGetTimeOfNextVarHit	Flag=4
Cálculo das saídas	mdlOutputs	Flag=3
Atualização dos estados discretos	mdlUpdate	Flag=2
Cálculos das derivadas	mdlDerivatives	Flag = 1
Fim das tarefas	mdlTerminate	Flag = 9

6.7 Tempo de Amostragem e Offsets

Tanto os arquivos .m como os arquivos .c têm as seguintes opções para tempo de amostragem:

-Tempo de amostragem contínuo – Para este tipo de S-function, a saída muda no menor espaço de tempo.

-Tempo de amostragem discreto – Pode-se definir tempos de amostragem discretos na chamada do bloco, além de um offset de atraso.

-Tempo de amostragem variável – Pode-se definir um tempo de amostragem discreto no qual os intervalos são variáveis

-Inherited Sample time – Algumas vezes o bloco S-function não tem nenhuma característica definida, é contínuo ou discreto, dependendo do tempo de amostragem de algum bloco do sistema.
Para configurar o tempo de amostragem inherited, usa-se -1 em M-file S-Functions e INHERITED_SAMPLE_TIME em C.

Outros tempos de amostragem são especificados no formato [sample_time, offset_time], e alguns parametros válidos são:

[CONTINUOUS_SAMPLE_TIME, 0.0]
[CONTINUOUS_SAMPLE_TIME, FIXED_IN_MINOR_STEP_OFFSET]
[discrete_sample_time_period, offset]
[VARIABLE_SAMPLE_TIME, 0.0]

onde

CONTINUOUS_SAMPLE_TIME = 0.0
FIXED_IN_MINOR_STEP_OFFSET = 1.0
VARIABLE_SAMPLE_TIME = -2.0

6.8 Compilação

A modificação realizada no código padrão foi a inserção dos códigos do cliente e servidor na função de saída da s-function, salvando-se os arquivos com a extensão .c

Para compilar esse código, por exemplo para o cliente.c, para ser usado em MATLAB é necessário executar o comando *mex*, acrescentando-se a biblioteca wsock32.lib, que possui os comando de rede:

mex cliente.c wsock32.lib

Esse comando resulta num arquivo .dll que será utilizado na execução da simulação no Simulink.

7. Implementação

A idéia do projeto é a reprodução dos gestos captados de uma mão real por uma webcam (e devidamente processados para a retirada dos parametros) por uma mão robótica antropomorfica. Porém, como o foco deste trabalho é a comunicação dos dois extremos,e a independência da implementação das outras partes do projeto é conseguida com o envio de gestos captados pelo simulador, e pela reprodução na outra parte da rede pelo mesmo simulador.

Para a parte de envio dos gestos são implementados dois módulos: um que grava a posição em que a mão está colocada e outro que propriamente envia os gestos (cliente).

De forma similar, para a reprodução dos gestos são acrescentados outros dois módulos: um que recebe os gestos(servidor) e outro que os reproduz.
A Figura 36 ilustra a implementação:

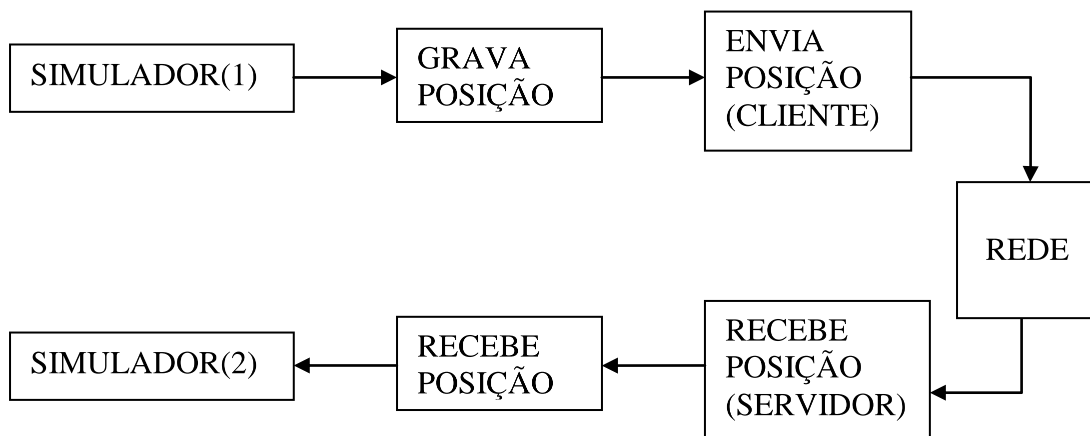


Figura 36 - Fluxograma de Blocos de transmissão

As Figuras 37 e 38 mostram as implementações no ambiente simulink tanto do cliente quanto do servidor, descritas acima:

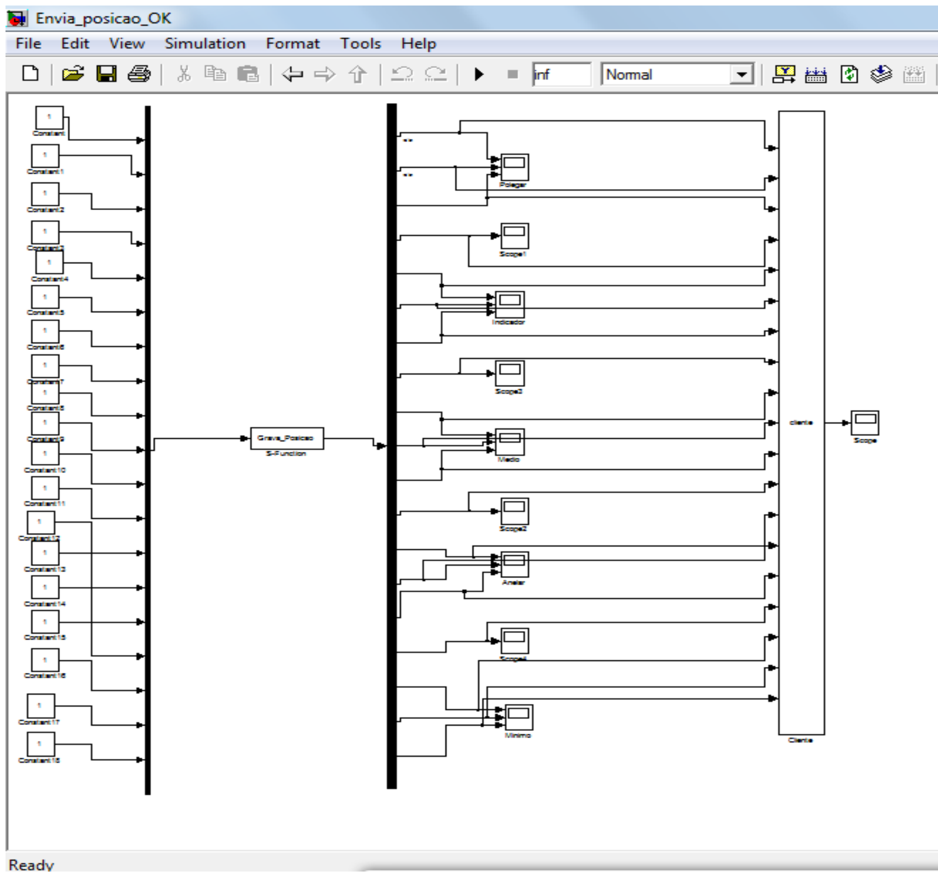


Figura 37 - Implementação do Cliente

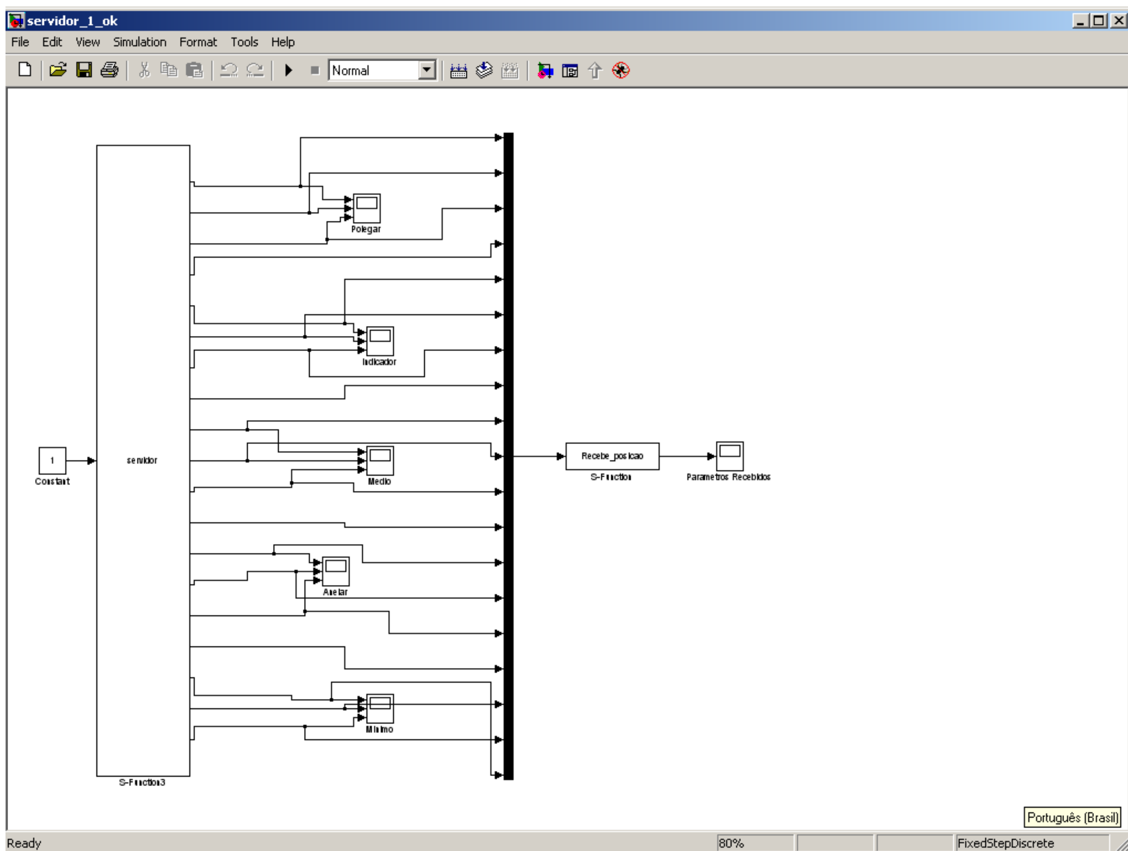


Figura 38 - Implementação do Servidor

O modelo implementado acima busca as posições numa mão virtual, para simular a identificação de gestos, que pode ser controlada com um cursor pelo usuário, e as envia para o mesmo simulador replicado a fim de executar as posições recebidas.

Os códigos podem ser observados nos apendices

A figura 39 mostra o resultado da simulação realizada em um único computador, usando-se o IP local da máquina, ao invés de outro da rede.

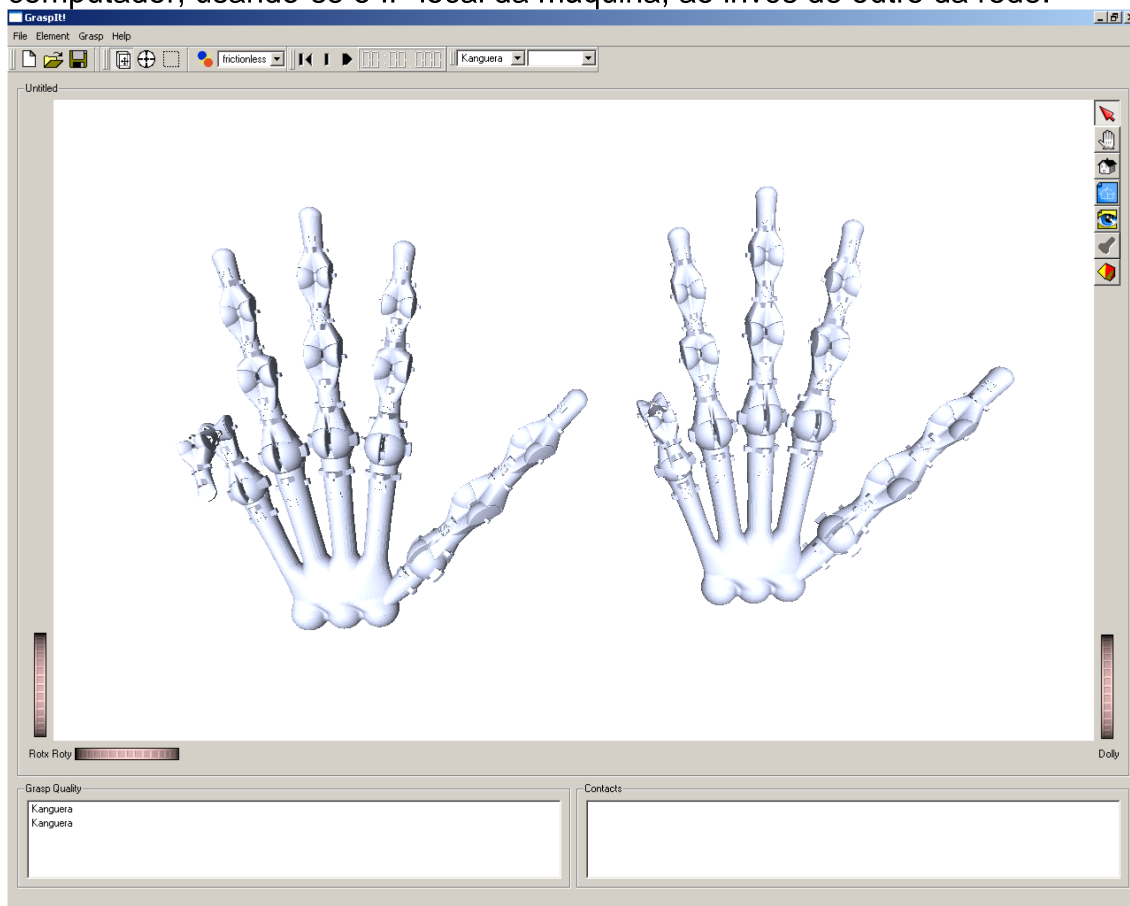


Figura 39 - Resultado da Implementação

8. Testes de Desempenho

8.1. Metodologia

Para os testes de desempenho será usado o programa Ping.exe, o qual testa o desempenho da rede através do envio de pacotes e a marcação do tempo de recebimento.
Foram enviados dados dos tamanhos 10, 50, 80, 100, 150, 200, 250, 500, 1000 e 2000 bytes para diferentes configurações de redes e diferentes distâncias, sendo gravado um arquivo de dados para posterior processamento.
O comando usado nesse procedimento é da seguinte forma:

ping xxx.xxx.xxx.xxx -t > c:\logfile.txt

onde xxx.xxx.xxx.xxx é o número ip desejado. O parâmetro –t é usado para especificar que o teste será feito por tempo indeterminado, adotando-se 10min em média.

Outro teste foi realizado para determinar o tempo de resposta do software implementado, incluindo o tempo de iteração do MATLAB e o tempo de rede. Encontrando-se o tempo da aplicação e o tempo de rede, descrito anteriormente, pode-se encontrar o tempo apenas de iteração do MATLAB.

Para se medir o tempo da aplicação, foi acrescentado ao código um procedimento para gravar em um arquivo de dados.txt os tempos do relógio local, incluindo milisegundos, a cada recebimento de mensagem. Com a diferença dos tempos do sistema, encontra-se o tempo de recebimento da mensagem.

Abaixo encontra-se o procedimento citado acima com comentários:

```
SYSTEMTIME lt;              Declaração da variável lt, que armazena o tempo do
sistema
GetLocalTime(&lt);          Função que registra o tempo do sistema
fp=fopen("test.txt", "w");  Abertura do arquivo
fprintf(fp, "The local time is: %02d:%02d:%02d:%03d\n", lt.wHour, lt.wMinute,
lt.wSecond,lt.wMilliseconds);
                        Gravar a variável no arquivo
```

8.1.1.Computador Local

Foram realizados os testes descritos acima executando-se o programa ping.exe para o número ip local, ou seja, alocado no mesmo computador onde o programa é executado. As configurações do computador foram as seguintes:

Memória: 4Gb DDR2

Processador: Core2 Due 2.2 GHz

8.1.2. Rede Local

Os mesmos testes descritos acima foram realizados em computadores conectados na rede local. Para os testes dessa rede foram usados computadores com as seguintes configurações:

A1)Memória: 4Gb DDR2
Processador: Core2 Due 2.2 GHz

A2)Memória: 2Gb DDR2
Processador: Core2 Due 2.2 GHz

8.1.3. Conexão Remota

Para a conexão remota realizaram-se os mesmos testes de velocidade acima, porém, como o sistema desenvolvido exige um IP fixo, um túnel foi criado para a aplicação. Para esse fim, utilizou-se o software HAMACHI [7].

A fim de determinar o caminho percorrido pelo pacote de dados através da rede, utilizou-se o software TRACERT.EXE. Ele identifica os pontos de roteamento, que se estendem da cidade de São Carlos-SP até a cidade de São Paulo-SP. A figura 40 mostra os valores encontrados.

```
Rastreando a rota para ip-187-117-53-117.user.vivozap.com.br [187.117.53.117]
com no máximo 30 saltos:

  1     2 ms     2 ms     2 ms  192.168.0.1
  2    16 ms    22 ms     9 ms  bd23b801.virtua.com.br [189.35.184.1]
  3    27 ms    21 ms    14 ms  189.7.80.1
  4    13 ms    18 ms    14 ms  embratel-G2-0-1-ngacc02.rpo.embratel.net.br [20
.90.150.21]
  5    34 ms    37 ms    40 ms  200.230.245.193
  6    30 ms    30 ms    29 ms  ebt-T0-3-2-0-tcore01.spoph.embratel.net.br [200
244.160.50]
  7    43 ms    36 ms    37 ms  ebt-C2-gacc02.spomb.embratel.net.br [200.230.24
.130]
  8    26 ms    54 ms    28 ms  peer-G3-0-gacc02.spomb.embratel.net.br [200.211
219.26]
  9    31 ms    32 ms    27 ms  201-0-2-6.dsl.telesp.net.br [201.0.2.6]
 10     *        *        *     Esgotado o tempo limite do pedido.
 11    41 ms    31 ms    39 ms  200.142.132.16
 12     *        *        *     Esgotado o tempo limite do pedido.
 13   380 ms   414 ms   392 ms  ip-187-117-53-117.user.vivozap.com.br [187.117.
3.117]

Rastreamento concluído.
```

Figura 40 - Pontos de Roteamento do IP remoto

9. Resultados

9.1 Computador Local

Como no computador local o tempo de recebimento da mensagem varia pouco em função do tamanho do pacote, é mostrado apenas o gráfico para 1000 bytes, que pode ser visto na figura 41.

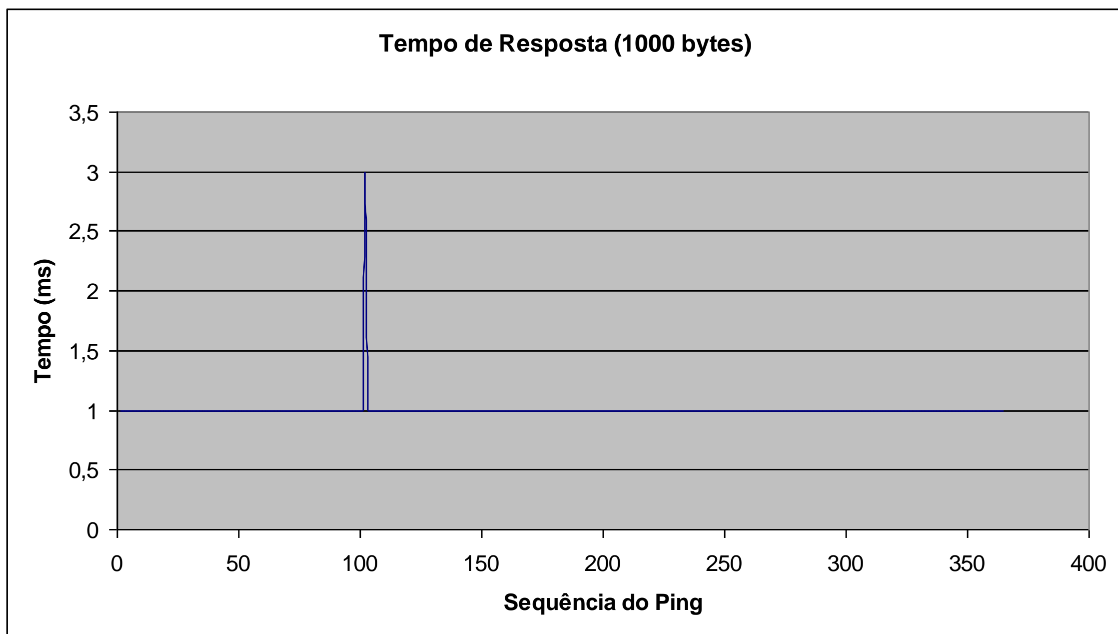


Figura 41 - Tempo(ms) para 1000 bytes

Média = 1ms

O gráfico da figura 42 mostra a medição do erro das mensagens em função do delay. O delay é atribuido em virtude do tempo de processamento do buffer de recebimento, que se for menor que um determinado valor agrupa duas ou mais mensagens, causando erro na transmissão.

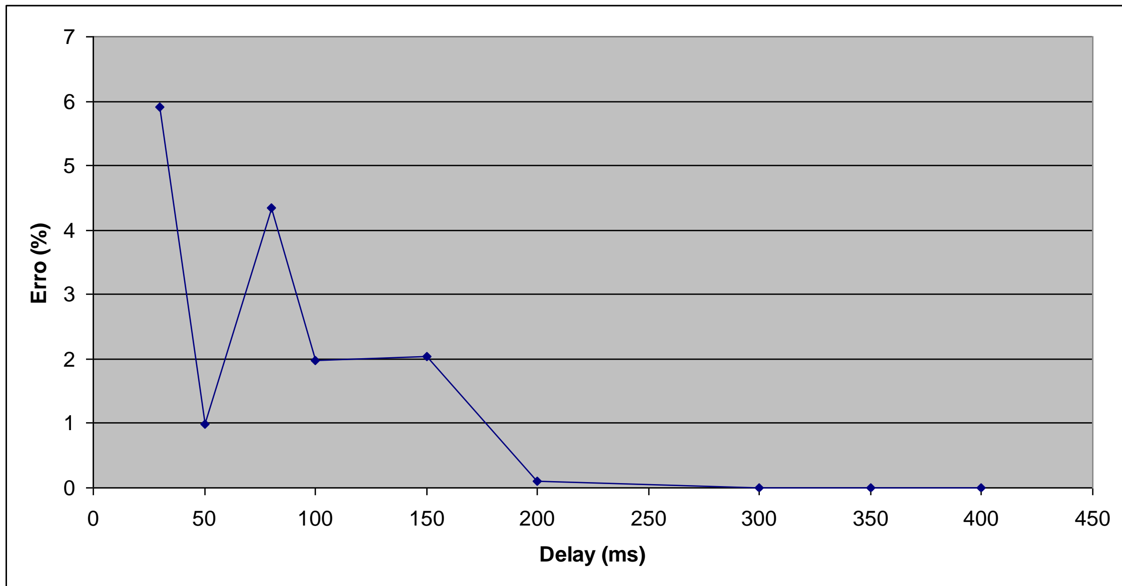


Figura 42 - Erro (%) x Delay

O grafico da figura 43 mostra o tempo de recebimento das mensagens, obtido com a diferença dos tempos no MATLAB. Como é feita a subtração dos valores, estão computados o delay usado e tempo de processamento do MATLAB, pois o tempo de transmissão da rede teoricamente é o mesmo para duas mensagens subsequentes. Foi escolhido o delay de 400 ms pois a partir desse valor não há erros de transmissão.

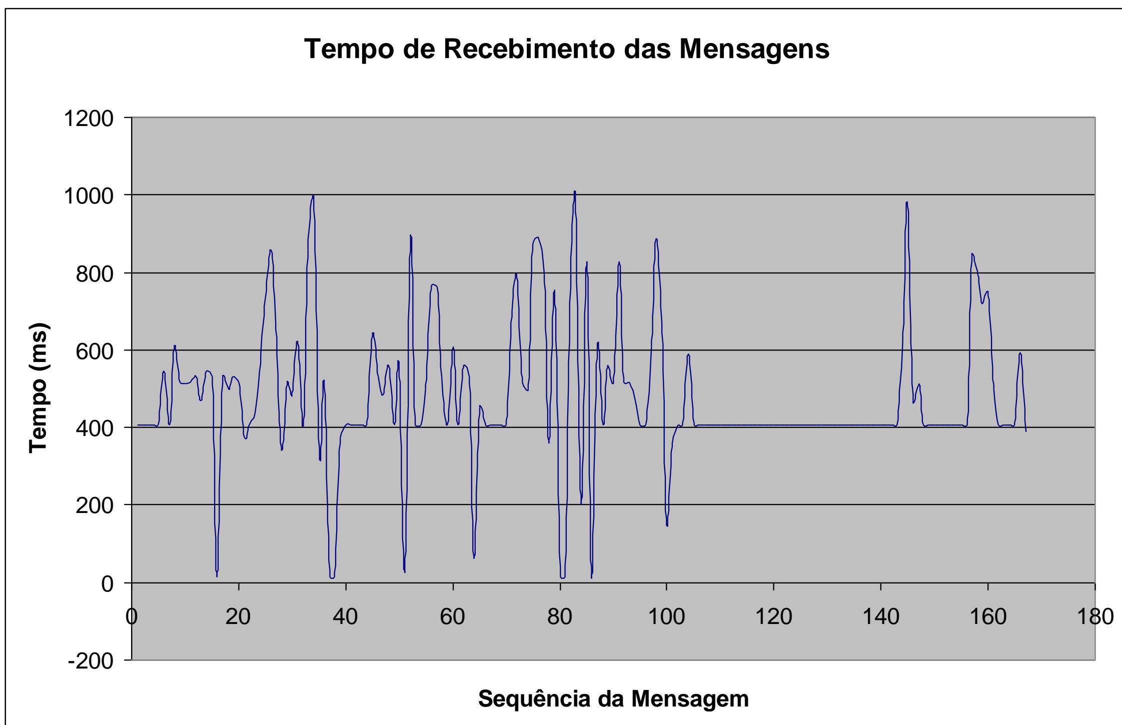


Figura 43 - Tempo de recebimento das Mensagens

Média = 478,73

9.1 - Rede Local

Os gráficos abaixo (Figuras 44 à 52) apresentam os resultados do envio dos pacotes de diferentes tamanhos pelo comando ping na rede local.

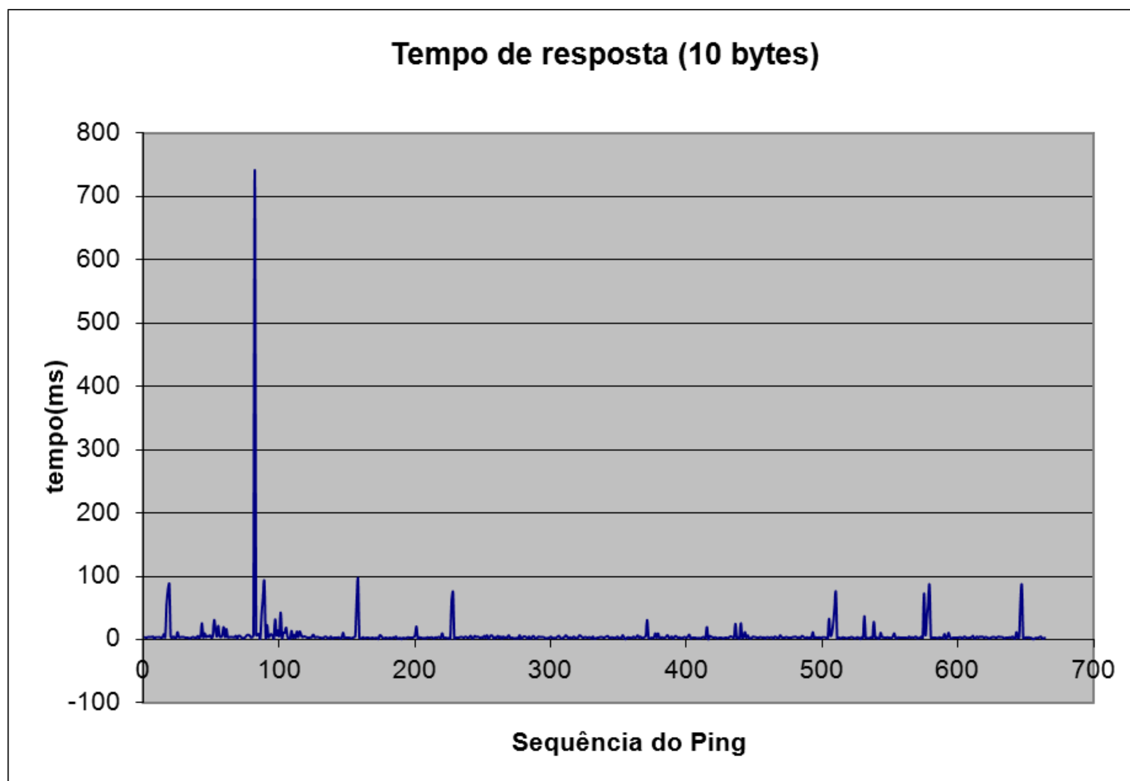


Figura 44 - Tempo de Resposta (10 bytes)

Média = 7ms

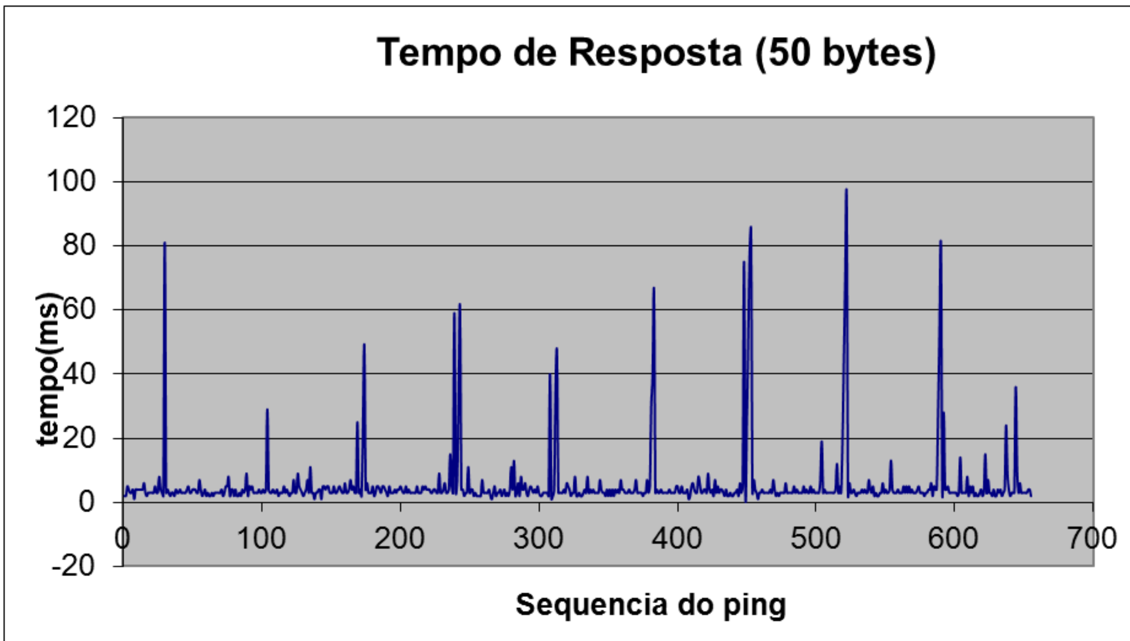


Figura 45 - Tempo de Resposta (50 bytes)

Média = 5ms

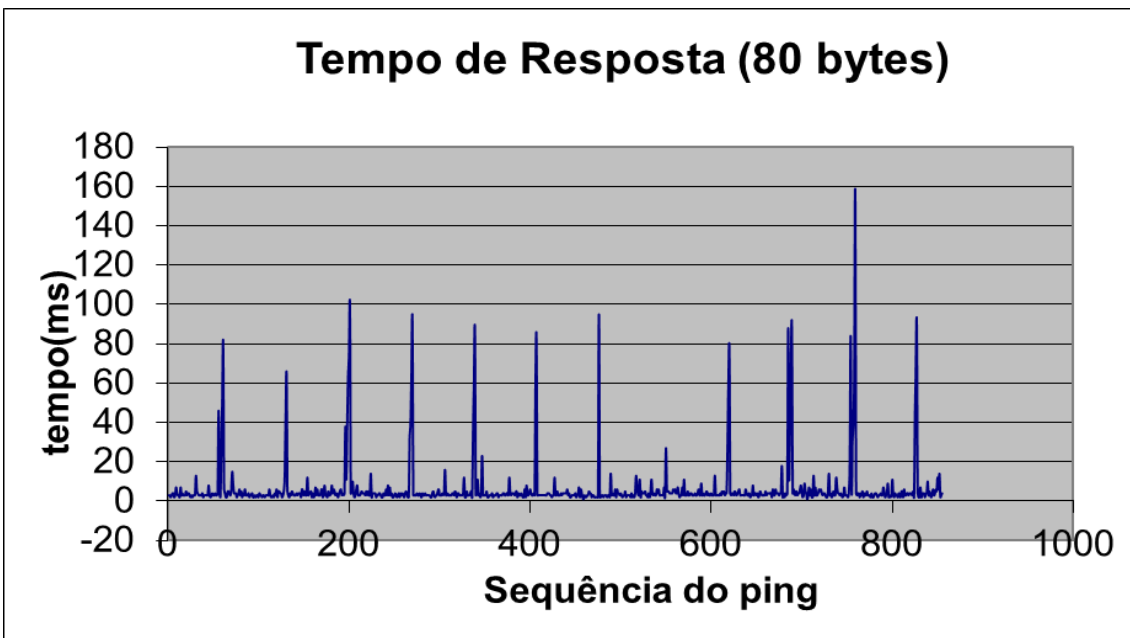


Figura 46 - Tempo de Resposta (80 bytes)

Média = 6ms

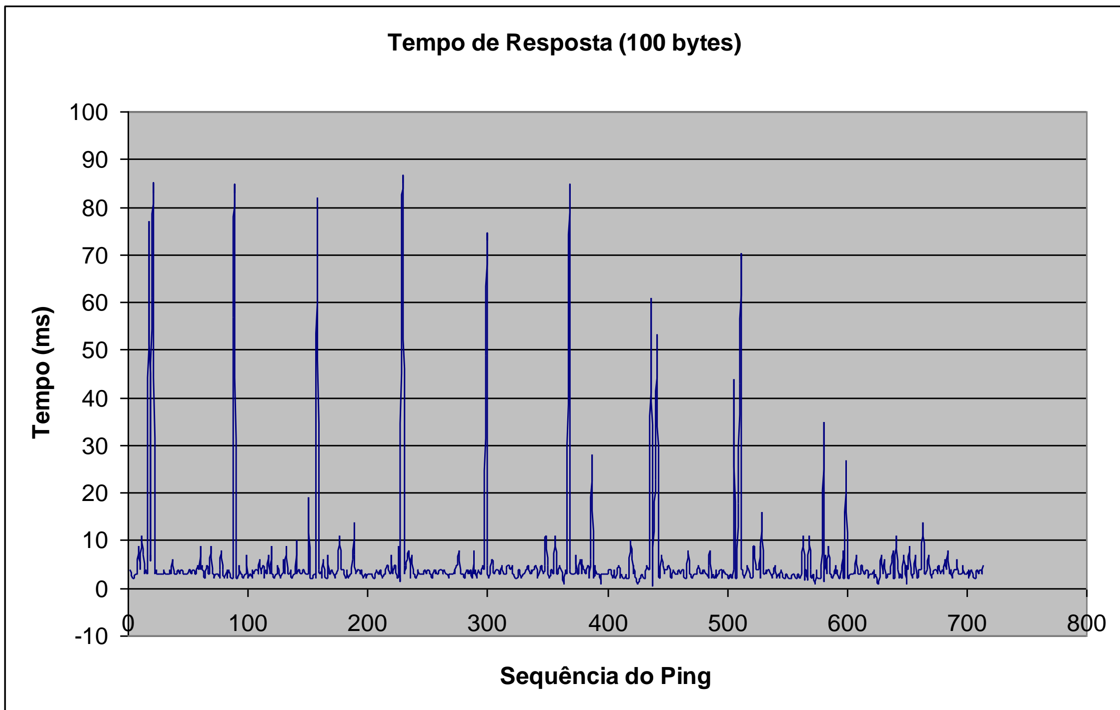


Figura 47 - Tempo de Resposta (100 bytes)

Média = 5 ms

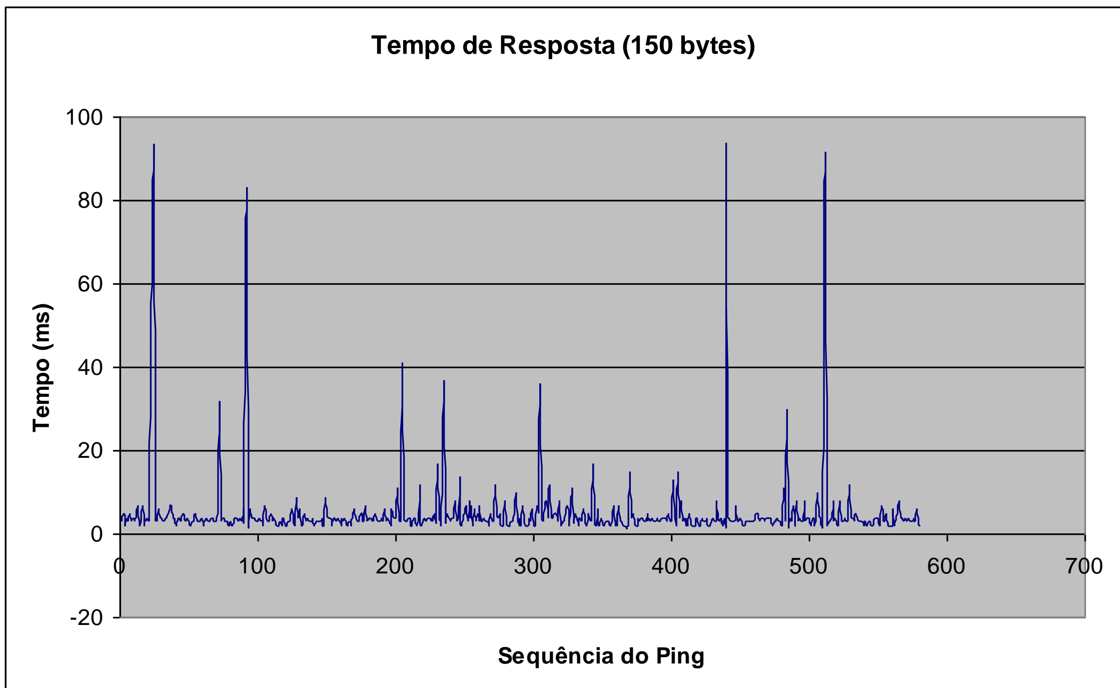


Figura 48 - Tempo de Resposta (150 bytes)

Média = 5 ms

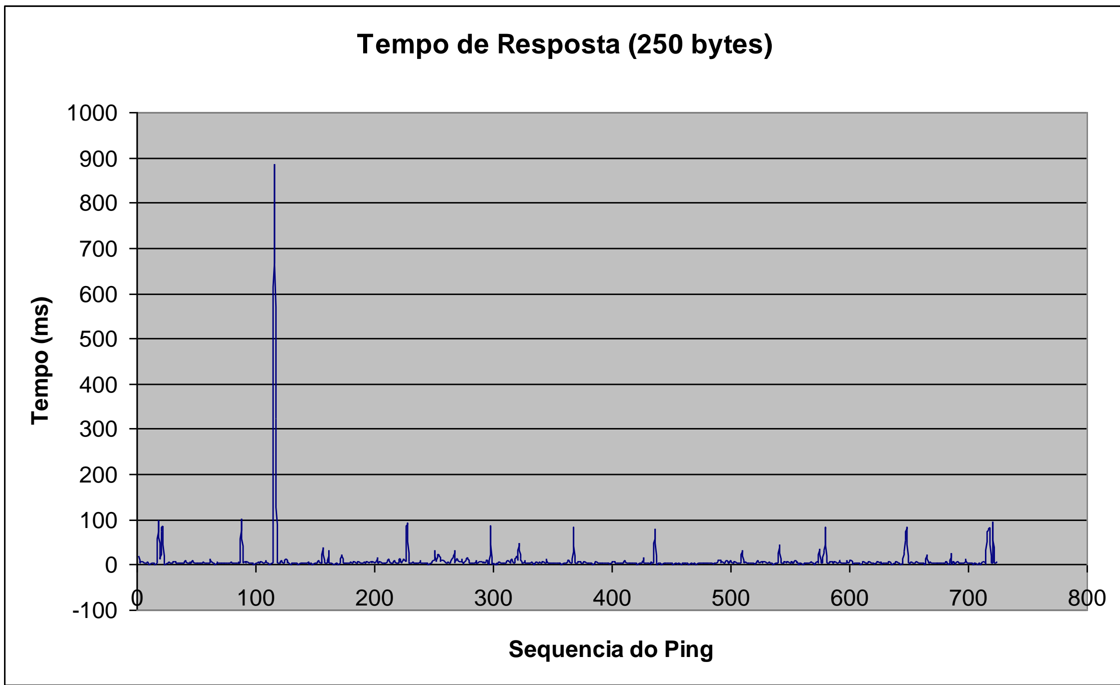


Figura 49 - Tempo de Resposta (250 bytes)

Média = 9 ms

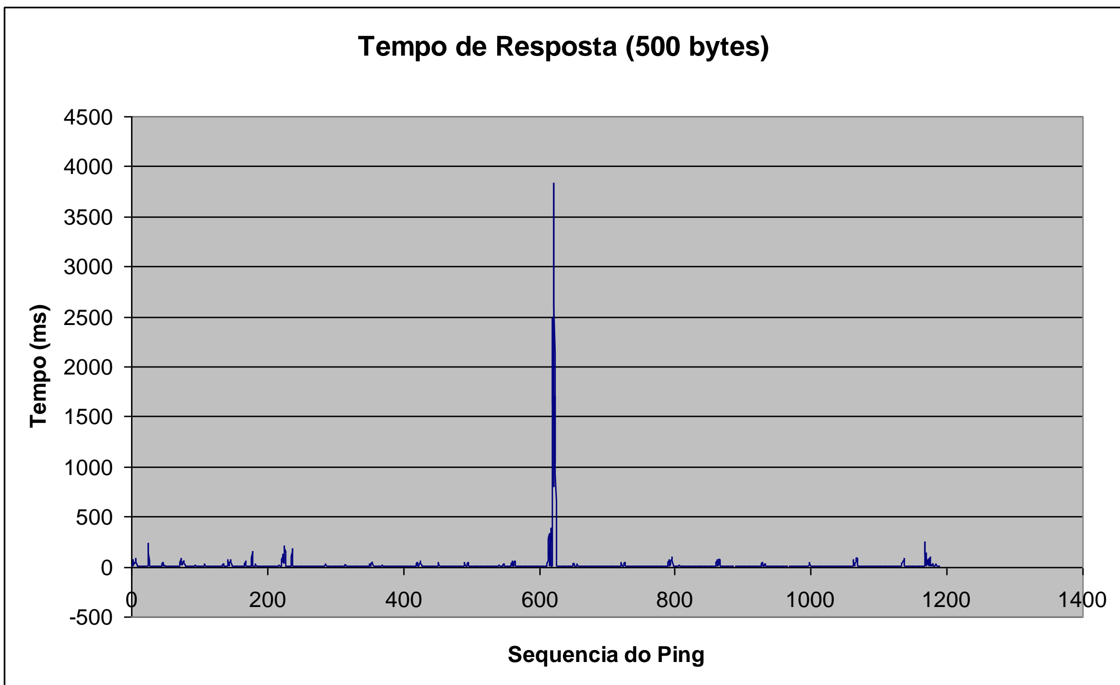


Figura 50 - Tempo de Resposta (500 bytes)

Média = 17 ms

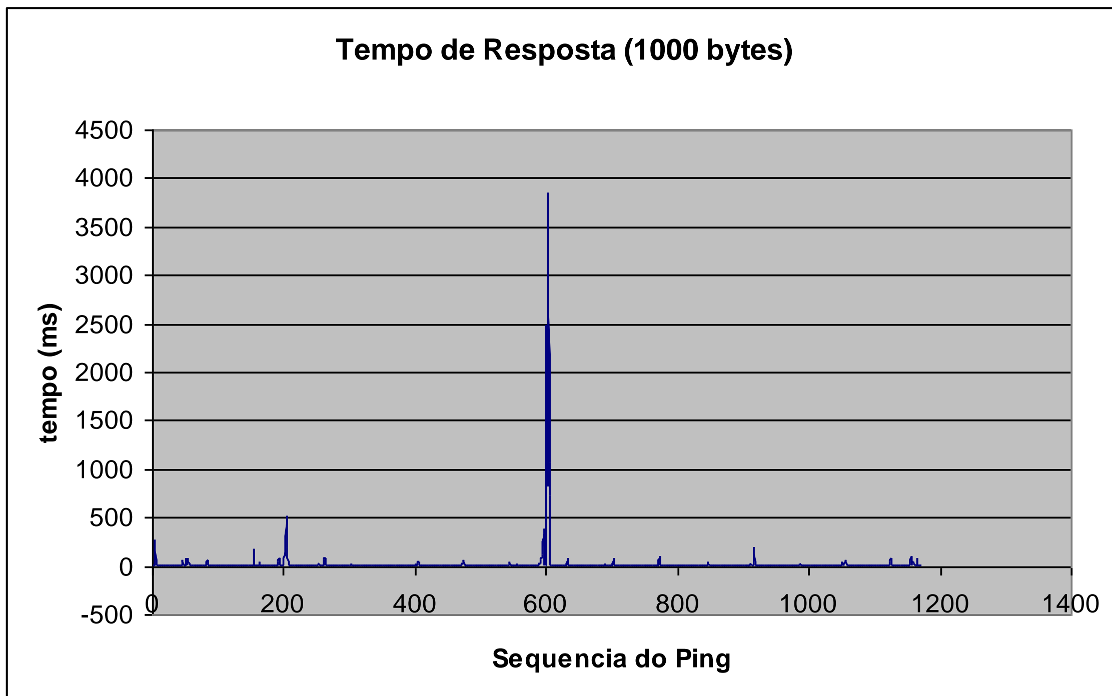


Figura 51 - Tempo de Resposta (1000 bytes)

Média = 18 ms

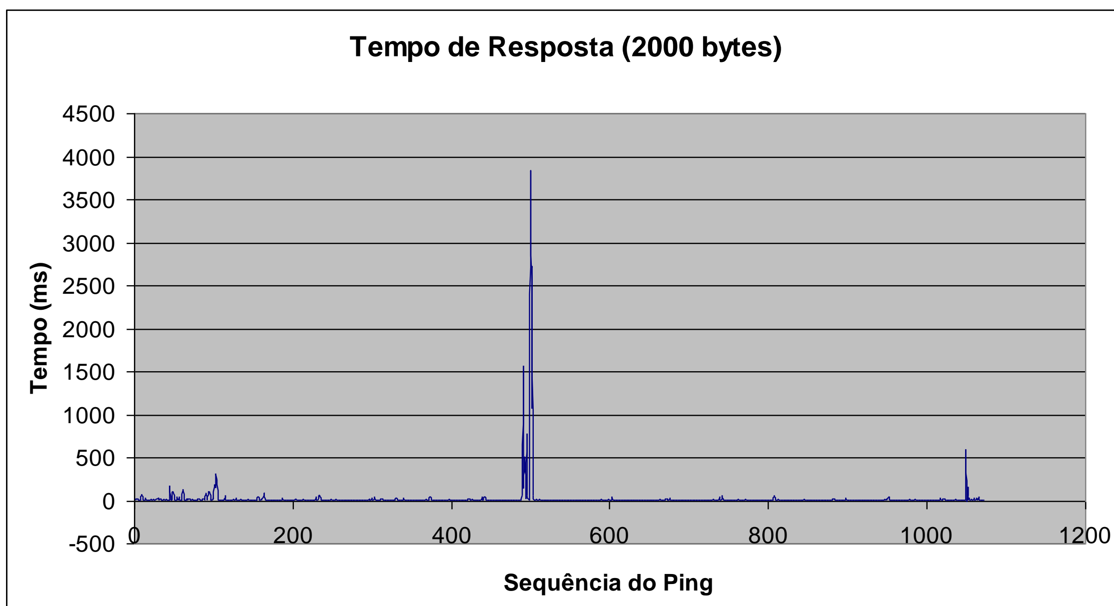


Figura 52 - Tempo de Resposta (2000 bytes)

Média = 21ms

No gráfico da figura 53 encontra-se o tempo médio de resposta em função do tamanho do pacote:

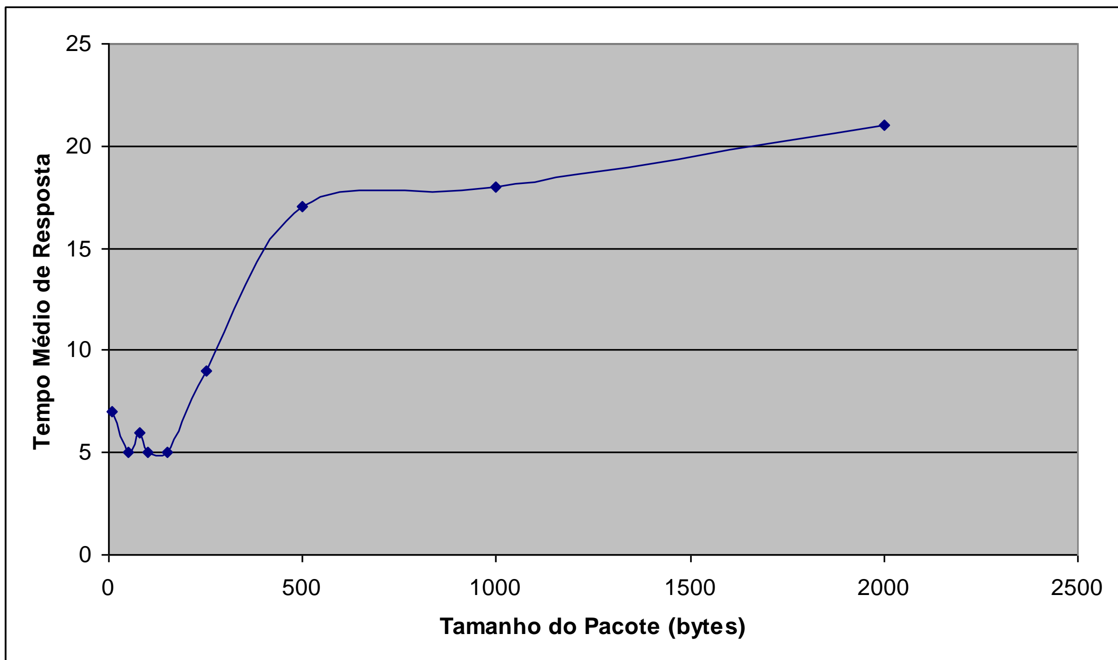


Figura 53 - Tempo Médio de Resposta x Tamanho do Pacote

A figura 54 mostra a medição do erro das mensagens em função do delay. O delay é atribuido em virtude do tempo de processamento do buffer de recebimento, que se for menor que um determinado valor agrupa duas ou mais mensagens, causando erro na transmissão.

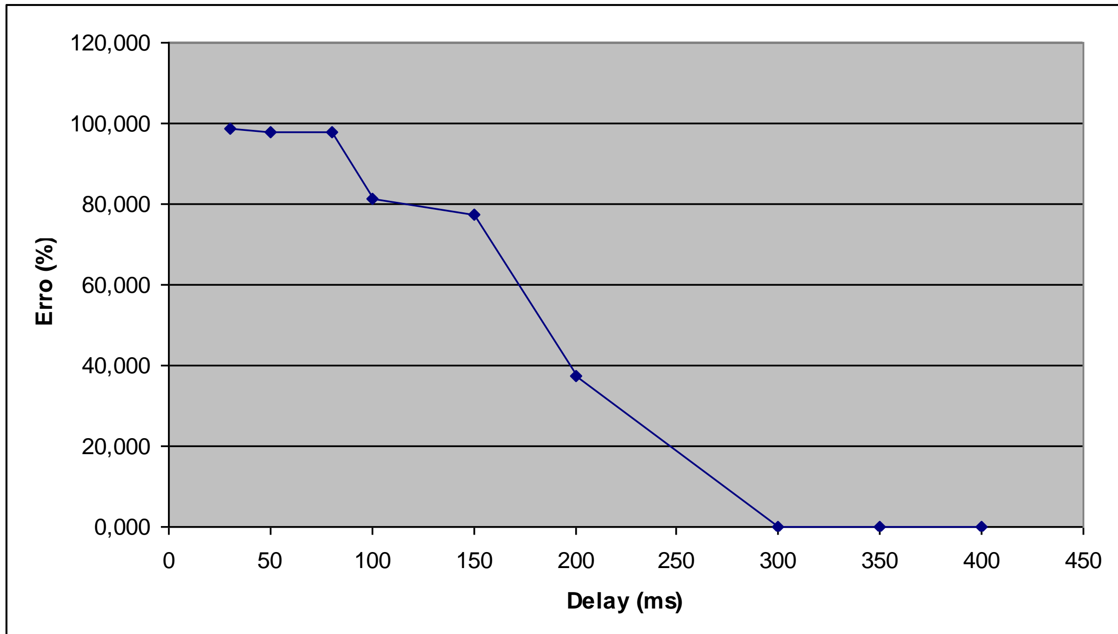


Figura 54 - Erro(%) x Delay

A figura 55 mostra o tempo de recebimento das mensagens para um delay de 400ms, tempo o qual a transmissão estabiliza-se.

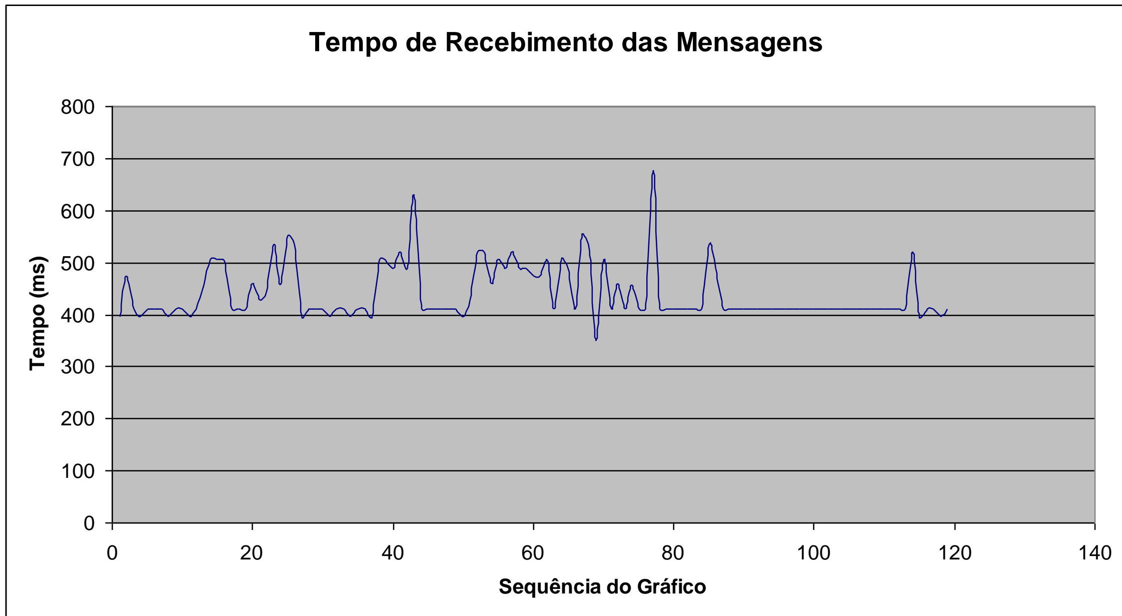


Figura 55 - Tempo de Recebimento das Mensagens

Média = 441,2269

9.3 Computador Remoto

Os gráficos abaixo (figuras 56 à 63) apresentam os resultados do envio dos pacotes de diferentes tamanhos pelo comando ping para um computador remoto.

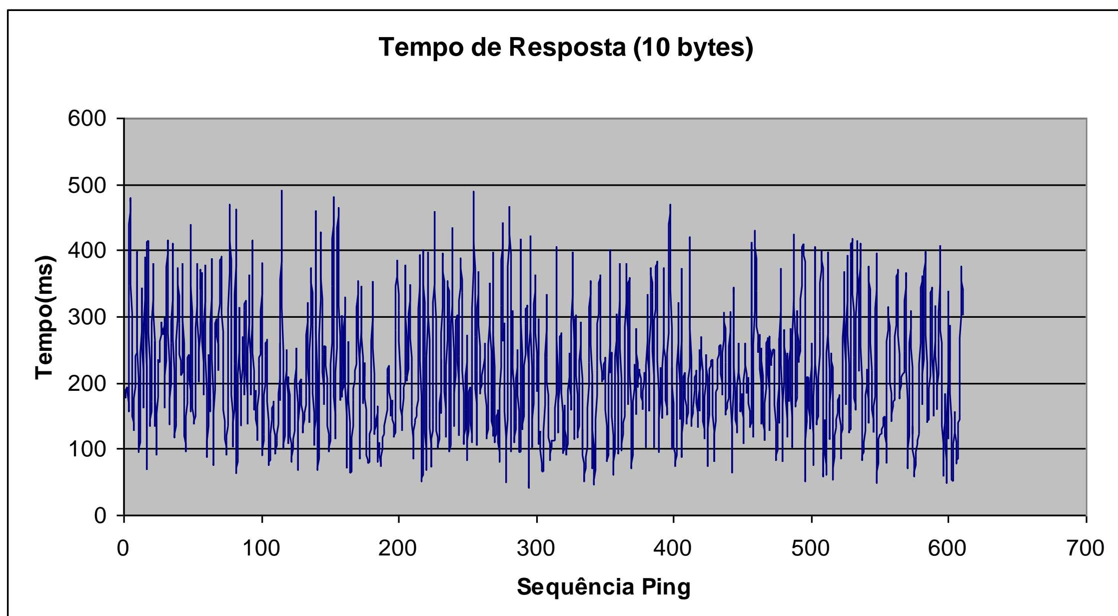


Figura 56 - Tempo de Resposta (10 bytes)
Média = 212,82 ms

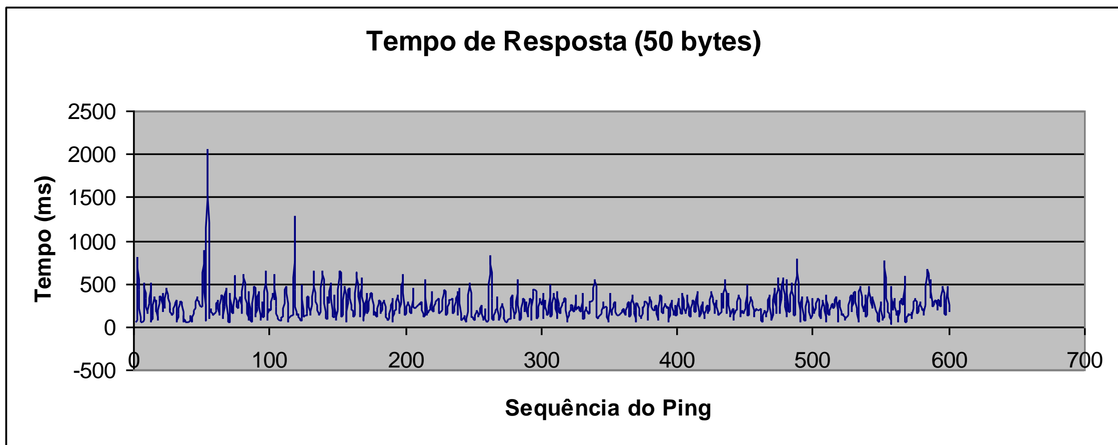


Figura 57 - Tempo de Resposta (50 bytes)

Média = 243,15 ms

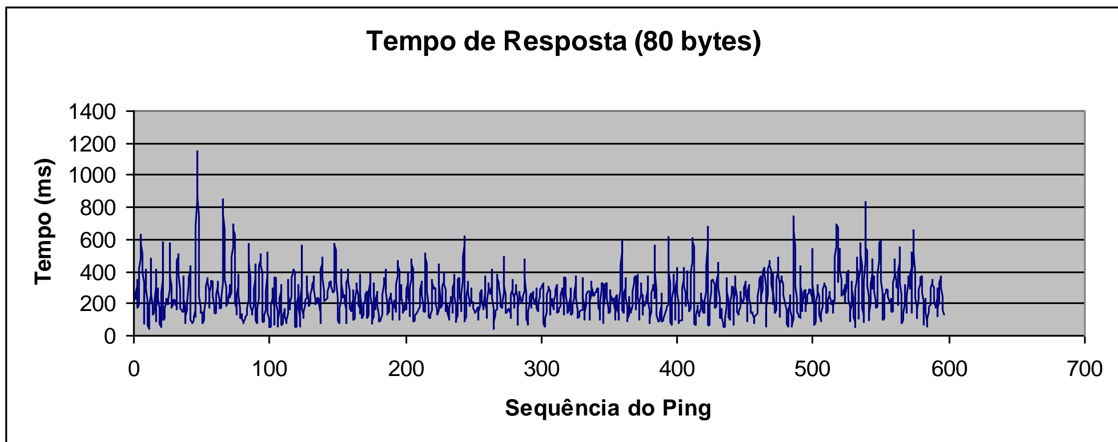


Figura 58 - Tempo de Resposta (80 bytes)

Média = 246,12 ms

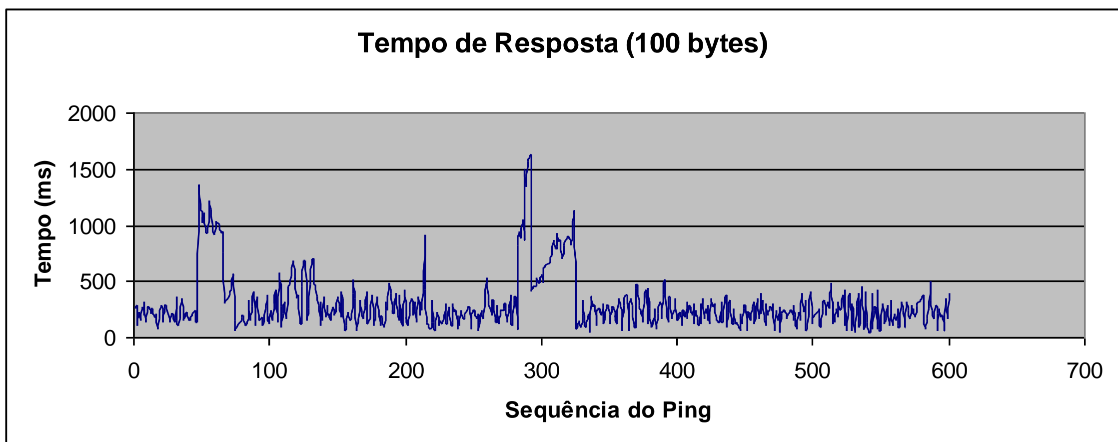


Figura 59 - Tempo de Resposta (100 bytes)

Média = 303,01 ms

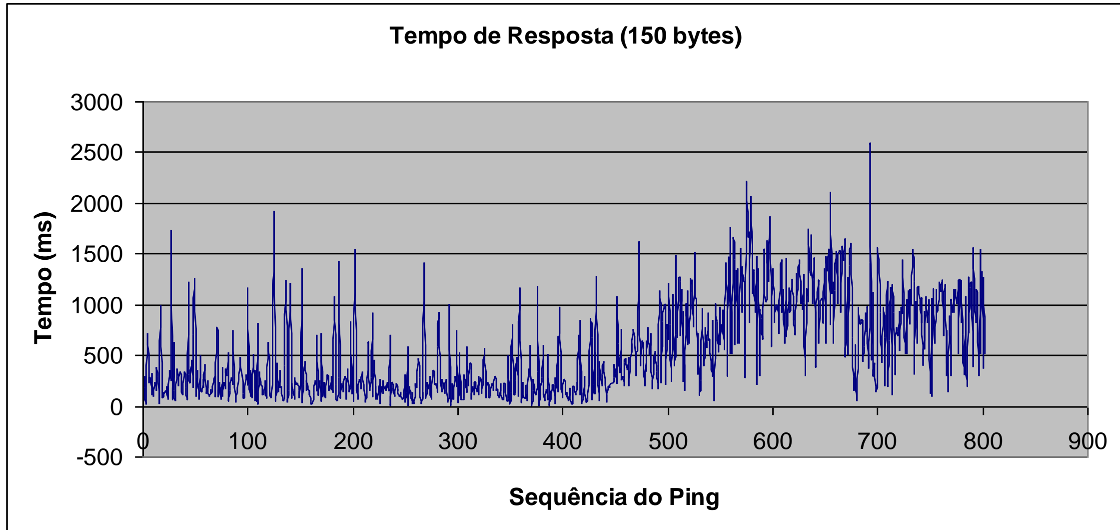


Figura 60 - Tempo de Resposta (150 bytes)

Média = 305,46ms

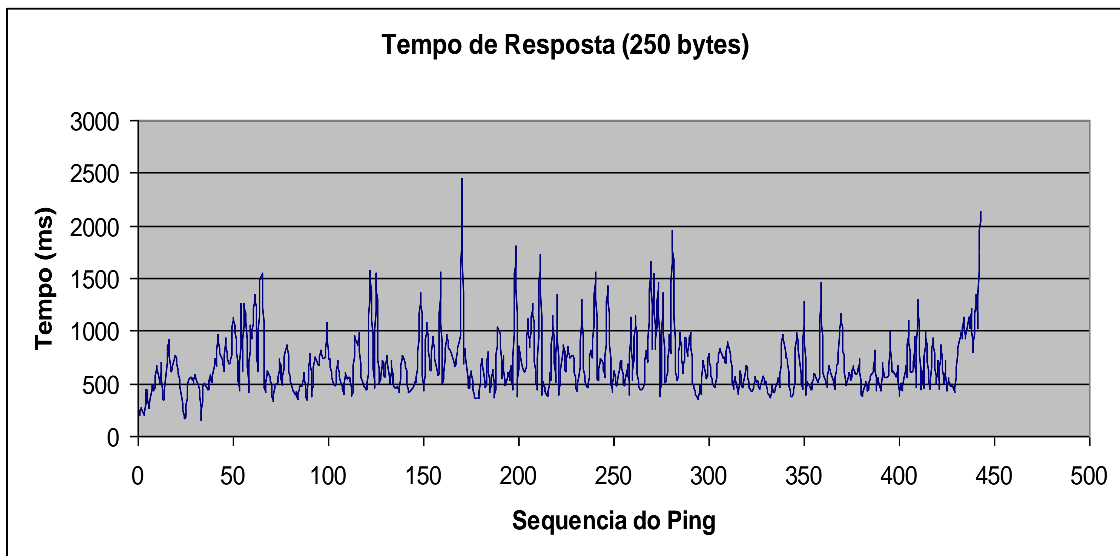


Figura 61 - Tempo de Resposta (250 bytes)

Média = 324,81ms

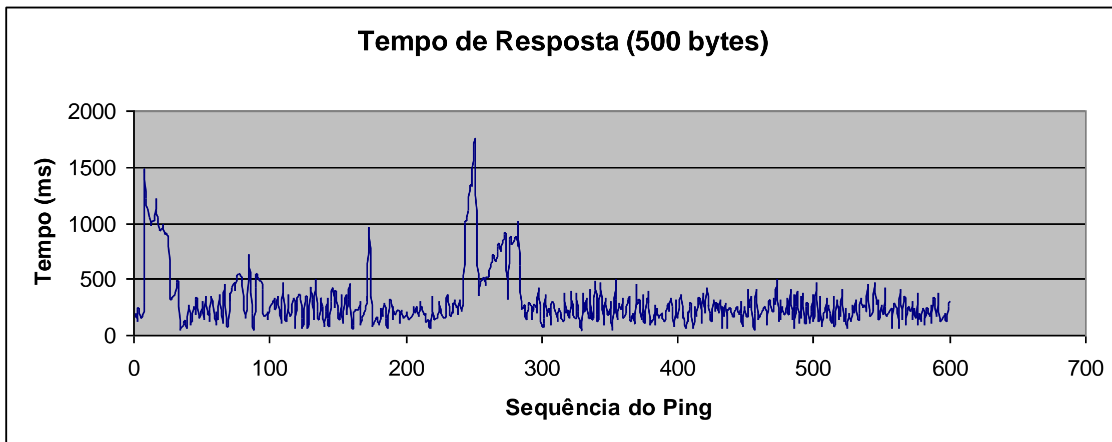


Figura 62 - Tempo de Resposta (500 bytes)

Média = 362,26ms

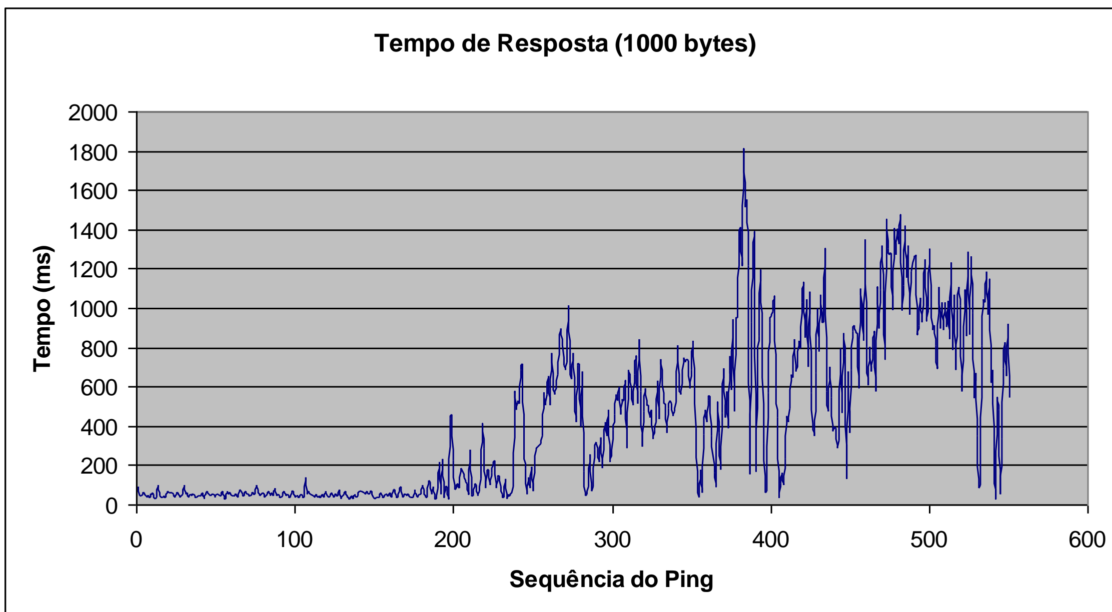


Figura 63 - Tempo de Resposta (1000 bytes)

Média = 418 ms

A figura 64 mostra o gráfico de tempo médio de resposta em função do tamanho do pacote:

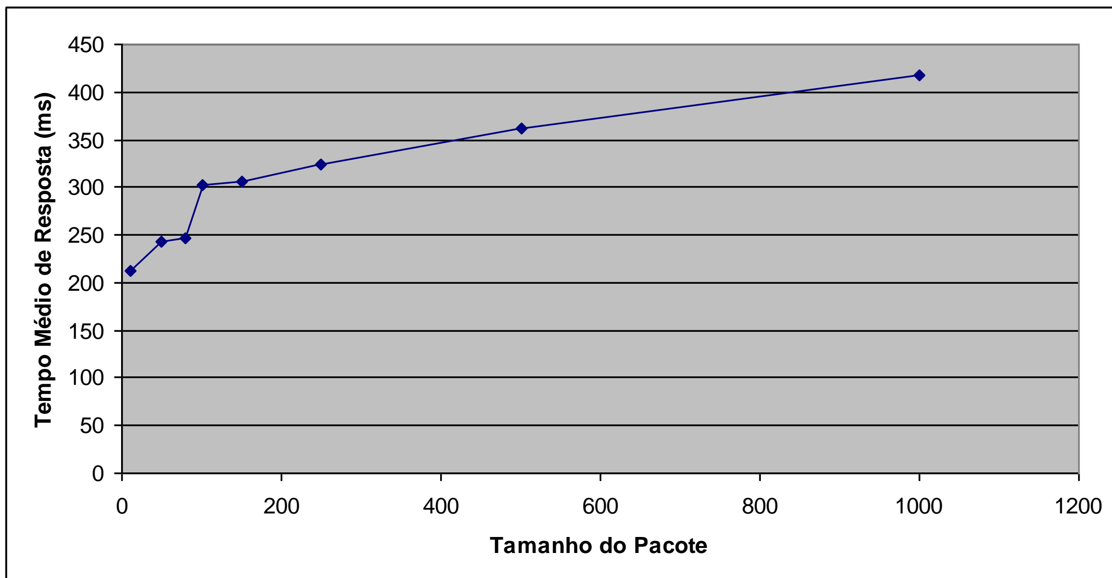


Figura 64 - - Tempo Médio de Resposta x Tamanho do Pacote

A figura 65 mostra a medição do erro das mensagens em função do delay.

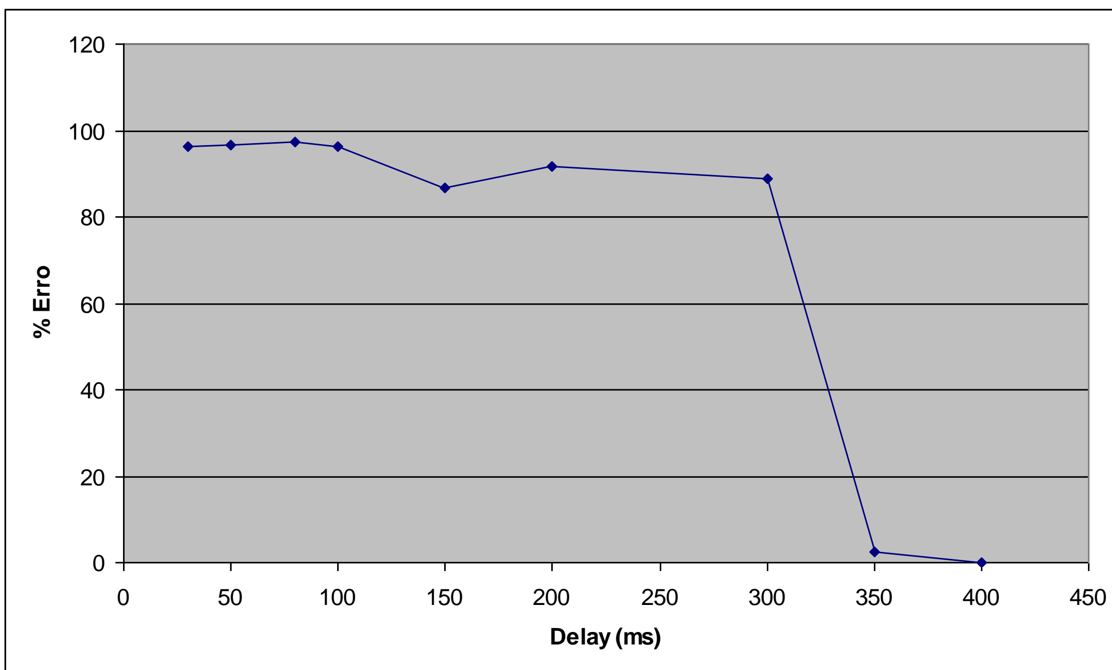


Figura 65 - Erro(%) x Delay (ms)

A figura 66 mostra o tempo de recebimento das mensagens, medido no MATLAB.

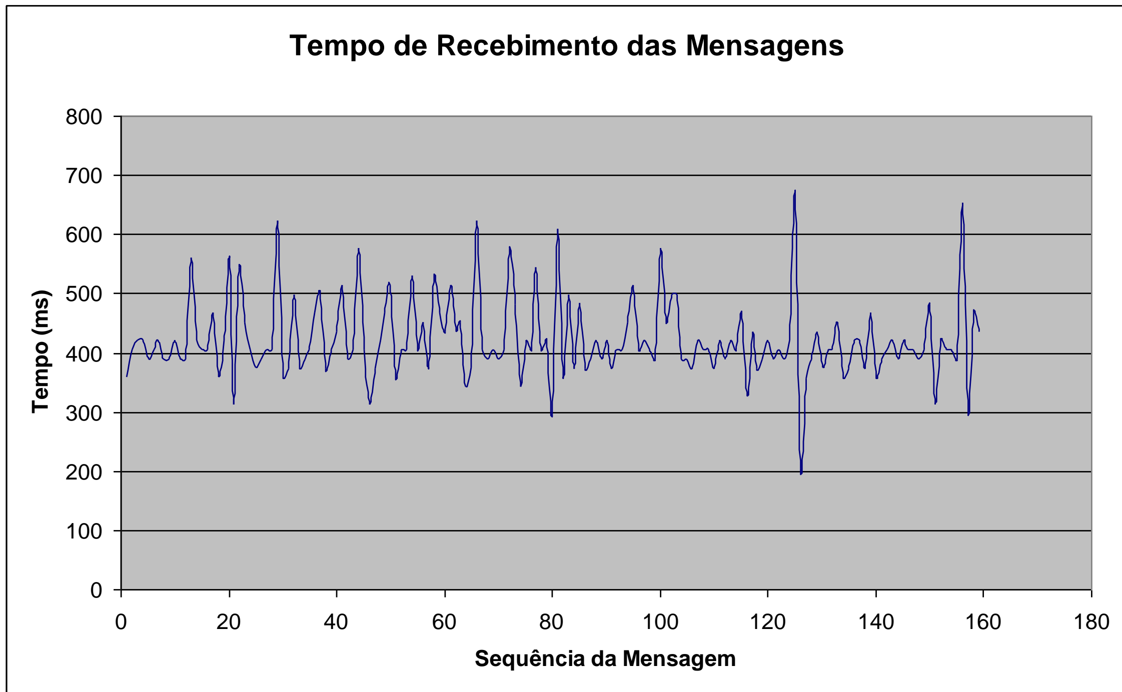


Figura 66 - Tempo de Recebimento das Mensagens

Média = 426,38

10. Conclusão

Pode-se concluir, observando o grafico da figura 67, que quanto maior o tempo de delay, menor é o erro percentual. Esse tempo é necessário devido ao tempo de processamento do buffer. Além disso, há uma diferença nos tempos de convergência, sendo maior para o tempo de transmissão do IP remoto.

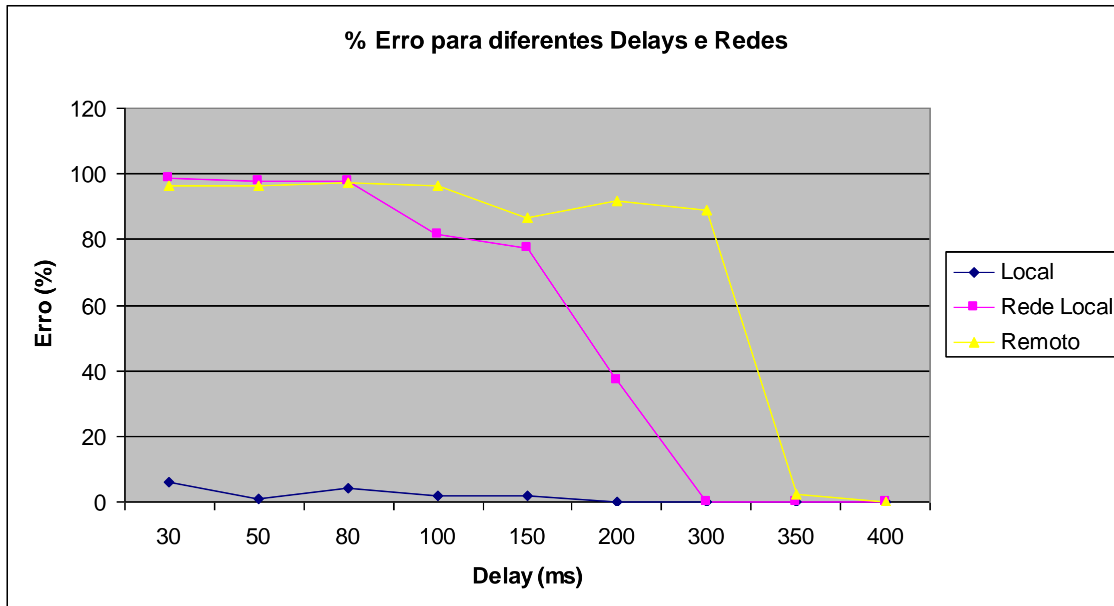


Figura 67 - Erro (%) para diferentes delays e Redes

E também, observa-se que os tempos de transmissão alteram-se em função dos tamanhos dos pacotes, sendo maior também para o IP remoto. Isso se deve ao fato, de que quanto maior a distância para transmissão, aumenta a probabilidade de ocorrer interferência ao longo do percurso, sendo necessária a retransmissão dos pacotes. Pode-se considerar também o aumento do número de firewalls com a distância, e consequentemente o aumento do tempo.

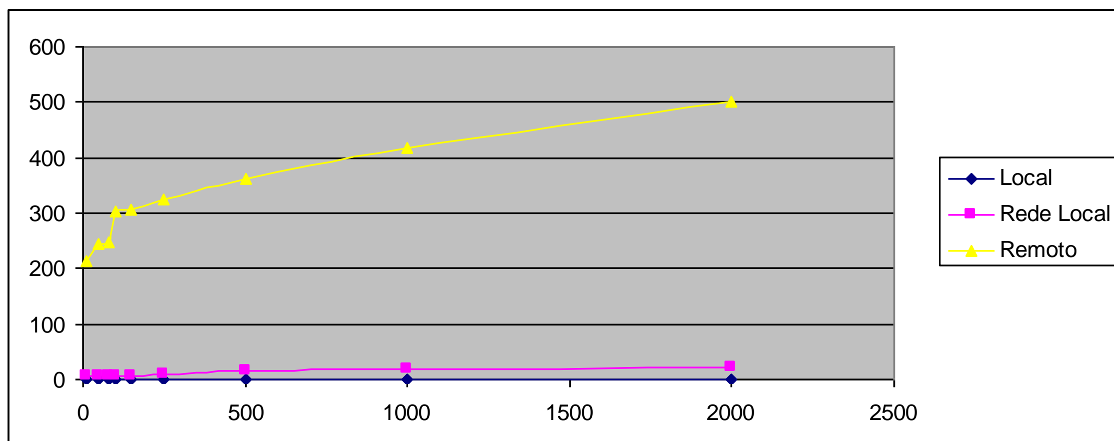


Figura 68 - Tempo de Resposta x tamanho do pacote (bytes)

Na figura 69 encontram-se os tempos médios de recebimento das mensagens do MATLAB. Os tempos foram computados para um delay de 400 ms, quando o erro converge a zero, e após isso foi subtraido esse tempo de atraso.

Pode-se concluir a partir desses dados que o tempo de processamento decai quando altera-se o destino de transmissão. Para um destino mais distante, o tempo diminui. Uma hipótese para o fato é que se o tempo de rede diminui ,mais mensagens tem que ser processadas num mesmo período, sobrecarregando o MATLAB.

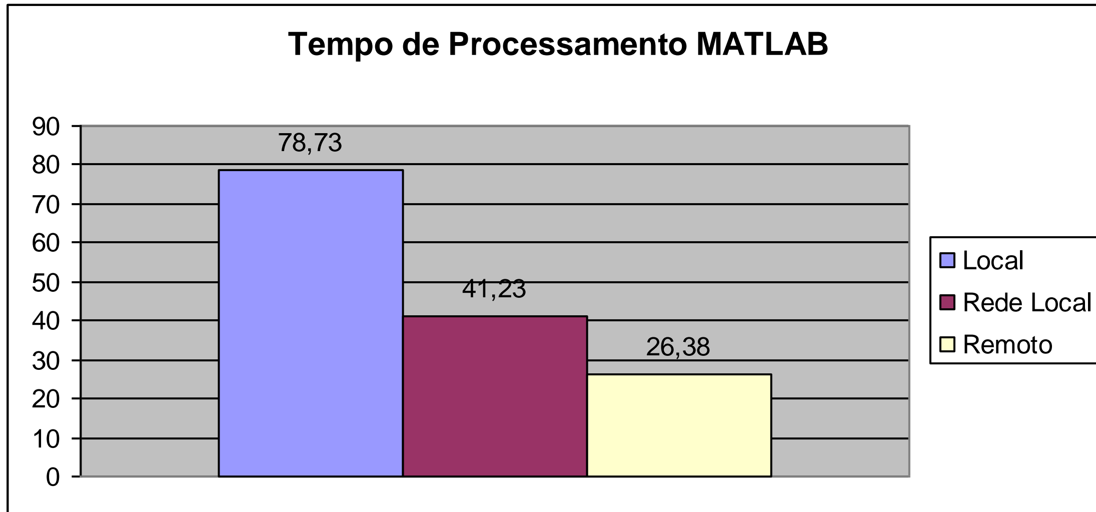


Figura 69 - Tempo de Processamento MATLAB

Analisando novamente os dados, pode-se chegar ao tempo de transmissão resultante da soma do tempo de processamento do MATLAB, do tempo de rede e do delay. Para o delay de 400 ms e um tamanho pacote de 90 bytes para a rede remota, temos:

Delay = 400 ms
Tempo de Rede = 270 ms
Tempo de Processamento = 26,38 ms

Tempo Total = 696,38 ms

Pode-se considerado esse tempo eficiente considerando o tempo de resposta do sistema mecânico.

11. Referências Bibliográficas

[1] http://iris.sel.eesc.usp.br/weblab/
[2] Comer, D. “Internetworking with TCP/IP”,2000.
[3] Donahoo, M. J.; Calvert, K. L. “TCP/IP sockets in C: Practical guide for programmers”,2001.
[4] www1.cs.columbia.edu/~cmatei/graspit
[5] http://pt.wikipedia.org/wiki/User_Datagram_Protocol
[6] http://msdn.microsoft.com/en-us/library/ms742213(VS.85).aspx
[7] http://www.LogMeIn.com

12. Apêndice

Apendice A – Codigo do servidor para enviar mensagem

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock.h>


int socket_servidor = 0;
int socket_cliente = 0;

int porta_servidor = 0;
int cliente_length = 0;


struct sockaddr_in servidor;
struct sockaddr_in cliente;

char mensagem[10];
char msg2[10];
int tamanho_msg;

WSADATA wsa_data;


int main(){

//Iniciando o aplicativo para conexão com a internet
WSAStartup(MAKEWORD(2, 0), &wsa_data);

//Criando o socket para o servidor
socket_servidor = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
printf ("socket_servidor: %d\n", socket_servidor);
//zera a estrutura servidor
memset(&servidor,0, sizeof(servidor));

//Compondo a estrutura servidor
servidor.sin_family = AF_INET;                    //familia
servidor.sin_port = htons(8080) ;                 //porta
servidor.sin_addr.s_addr  = htonl(INADDR_ANY);  //endereço (IP)

//Vinculando a estrutura do servidor (endereço e porta) ao socket criado
bind(socket_servidor, (struct sockaddr *) &servidor, sizeof(servidor));

//Coloca o socket para escutar as conexões
listen(socket_servidor, 5);

printf("Aguardando conexao...\n");
```

```
cliente_length = sizeof(cliente);

//Aceitando a conexão e coletando os dados do cliente
socket_cliente = accept(socket_servidor, (struct sockaddr *) &cliente,
&cliente_length);

printf("Conexao estabelecida!!!\n");

do{
        memset(&mensagem,0,10);        //limpa o buffer
        memset(&msg2,0,10);

        //mandando msg para o cliente
        printf("msg: ");
        gets(msg2);
        fflush(stdin);

        send(socket_servidor,msg2,10,0);

        //recebendo mensagem do cliente
       // tamanho_msg = recv (socket_cliente,mensagem, 10,0);
        //printf("%s: %s\n", inet_ntoa(cliente.sin_addr), mensagem);
  }
  while(strcmp(mensagem, "#quit")); // sai quando receber um "#quit" do
cliente
}
```

Apêndice B –Codigo para Concatenar mensagens

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>

#define pi 3.141592653589
#define Res  65536

int i;
int num_int;

char dados[50];                    //dados ja concatenados
char dado_[4][10];                  //dados ja convertidos para string

float N_[4];                       //dados recebidos em float

int main(){

N_[0]=1.22562;
N_[1]=0.56626;
N_[2]=pi/2;

for (i=0;i<3;i++){
   num_int=0;
   num_int= N_[i]*(Res/(pi/2));      //Converte double para int (Arredondamento
das casas decimais)
   sprintf(dado_[i],"%d",num_int); //Convertendo inteiro em string
   strcat (dado_[i],"#");          //Adicionando separador

   printf ("dado_%d: %s\n",i,dado_[i]);   //Mostra os dados separados
   strcat (dados,dado_[i]);              //Concatena os dados
              }

printf ("dados: %s\n\n",dados);
system("PAUSE");
}
```

Apendice C – Estrutura padrão do M-file

```
#define S_FUNCTION_NAME  sfuntmpl_basic

#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

static void mdlInitializeSizes(SimStruct *S)
{

    ssSetNumSFcnParams(S, 0);  /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /* Return if number of expected != number of actual parameters */
        return;
    }

    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, 1);
    ssSetInputPortRequiredContiguous(S, 0, true); /*direct input signal access*/

    ssSetInputPortDirectFeedThrough(S, 0, 1);

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, 1);

    ssSetNumSampleTimes(S, 1);
    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
    ssSetNumPWork(S, 0);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

    ssSetOptions(S, 0);
}


static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);

}


#define MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove
function */
```

```
#if defined(MDL_INITIALIZE_CONDITIONS)

  static void mdlInitializeConditions(SimStruct *S)
  {
  }
#endif /* MDL_INITIALIZE_CONDITIONS */

#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)

  static void mdlStart(SimStruct *S)
  {
  }
#endif /*  MDL_START */

static void mdlOutputs(SimStruct *S, int_T tid)
{
    const real_T *u = (const real_T*) ssGetInputPortSignal(S,0);
    real_T       *y = ssGetOutputPortSignal(S,0);
    y[0] = u[0];
}

#define MDL_UPDATE  /* Change to #undef to remove function */
#if defined(MDL_UPDATE)
  static void mdlUpdate(SimStruct *S, int_T tid)
  {
  }
#endif /* MDL_UPDATE */

#define MDL_DERIVATIVES  /* Change to #undef to remove function */
#if defined(MDL_DERIVATIVES)

  static void mdlDerivatives(SimStruct *S)
  {
  }
#endif /* MDL_DERIVATIVES */

static void mdlTerminate(SimStruct *S)
{
}

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

Apendice D - Cliente.c

```
#define S_FUNCTION_NAME cliente
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock.h>
#include <time.h>

#define BACKLOG_MAX 5
#define BUFFER_SIZE 150
#define EXIT_CALL_STRING "#quit"
#define remote_ip "192.168.5.20"
#define remote_port 8080

#define pi 3.141592653589
#define Res  65536

int remote_socket = 0;
int message_length = 0;

char message[BUFFER_SIZE];

struct sockaddr_in remote_address;

WSADATA wsa_data;

//Declaração de variáveis utilizadas para concatenar parametros

int i;
int num_int;                    //Usado para coverter float em inteiro

float N_[20];                   //dados recebidos em float

char dado_[20][10];             //dados convertidos para string
char dados[150];                //dados concatenados


//Funçao para gastar tempo
void pausa(const int atraso)
{
clock_t tf=clock()+atraso;
while ( clock() <= tf);
}

/* Exibe uma mensagem de erro e termina o programa */
void msg_err_exit(char *msg)
```

```
{
   fprintf(stderr, msg);
   system("PAUSE");
   exit(EXIT_FAILURE);
}

/*
* mdlInitializeSizes - initialize the sizes array
*/
static void mdlInitializeSizes(SimStruct *S)
{
   ssSetNumSFcnParams( S, 0); /*number of input arguments*/
   if (!ssSetNumInputPorts(S, 19)) return;
   ssSetInputPortWidth(S, 0, 1);
   ssSetInputPortWidth(S, 1, 1);
   ssSetInputPortWidth(S, 2, 1);
   ssSetInputPortWidth(S, 3, 1);
   ssSetInputPortWidth(S, 4, 1);
   ssSetInputPortWidth(S, 5, 1);
   ssSetInputPortWidth(S, 6, 1);
   ssSetInputPortWidth(S, 7, 1);
   ssSetInputPortWidth(S, 8, 1);
   ssSetInputPortWidth(S, 9, 1);
   ssSetInputPortWidth(S, 10, 1);
   ssSetInputPortWidth(S, 11, 1);
   ssSetInputPortWidth(S, 12, 1);
   ssSetInputPortWidth(S, 13, 1);
   ssSetInputPortWidth(S, 14, 1);
   ssSetInputPortWidth(S, 15, 1);
   ssSetInputPortWidth(S, 16, 1);
   ssSetInputPortWidth(S, 17, 1);
   ssSetInputPortWidth(S, 18, 1);

   ssSetInputPortDirectFeedThrough(S, 0, 1);
   if (!ssSetNumOutputPorts(S,1)) return;
   ssSetOutputPortWidth(S, 0, 1);
   ssSetNumSampleTimes(S, 1);



//---------------------Inicialização da comunicação----------------------------


   // inicia o Winsock 2.0 (DLL), Only for Windows
   if (WSAStartup(MAKEWORD(2, 0), &wsa_data) != 0)
      msg_err_exit("WSAStartup() failed\n");


     remote_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
   if (remote_socket == INVALID_SOCKET)
```

```
    {
       WSACleanup();
       msg_err_exit("socket() failed\n");
    }

  // preenchendo o remote_address (servidor)
    memset(&remote_address, 0, sizeof(remote_address));
    remote_address.sin_family = AF_INET;
    remote_address.sin_addr.s_addr = inet_addr(remote_ip);
    remote_address.sin_port = htons(remote_port);

printf("conectando ao servidor %s...\n", remote_ip);
    if (connect(remote_socket, (struct sockaddr *) &remote_address,
sizeof(remote_address)) == SOCKET_ERROR)
    {
       WSACleanup();
       msg_err_exit("connect() failed\n");
    }


}


//----------------------------------Fim da inicialização------------------------------------------
-



/*
* mdlInitializeSampleTimes - indicate that this S-function runs
* at the rate of the source (driving block)
*/
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}


/*
* mdlOutputs - compute the outputs by calling my_alg, which
* resides in another module, my_alg.c_2
*/


static void mdlOutputs(SimStruct *S, int_T tid)
{

 /* declaracao das entradas e saidas*/

    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
```

```
    real_T *y = ssGetOutputPortRealSignal(S,0);


//----------------------------------Envio de msgs-----------------------------------------------


   strcpy (dados,"");                        //zera variável

       //Agrupando os dados recebidos da mão
       for (i=0;i<19;i++){
           num_int=0;
           num_int=  *uPtrs[i]*(Res/(pi/2));                    //Converte  double  para  int
(Arredondamento das casas decimais)
           sprintf(dado_[i],"%d",num_int);        //Convertendo inteiro em string
           strcat (dado_[i],"#");                 //Adicionando separador
           //printf ("dado_%d: %s\n",i,dado_[i]);   //Mostra os dados separados
           strcat (dados,dado_[i]);              //Concatena os dados
                        }


       strcpy(message, dados);                  //copiar os dados concatenados para
message
       message_length = strlen(message);

       // envia a mensagem para o servidor

       if     (send(remote_socket,     message,     message_length,     0)     ==
SOCKET_ERROR)
       {
           WSACleanup();
           closesocket(remote_socket);
           msg_err_exit("send() failed\n");

       }

       pausa(50);              //em milisegundos
}


/*
* mdlTerminate - called when the simulation is terminated.
*/
static void mdlTerminate(SimStruct *S)
{

  printf("encerrando\n");
    WSACleanup();
```

```
    closesocket(remote_socket);
     system("PAUSE");

}

#ifdef MATLAB_MEX_FILE
#include "simulink.c"
#else
#include "cg_sfun.h"
#endif
```

Apêndice E - Grava Posição .m

```
function [sys,x0,str,ts] = timestwo(t,x,u,flag)

switch flag,

  % Initialize the states, sample times, and state ordering strings.
  case 0
    [sys,x0,str,ts]=mdlInitializeSizes;

  case 3

  sys=mdlOutputs(t,x,u);

  case { 1, 2, 4, 9 }
    sys=[];

  otherwise
    error(['Unhandled flag = ',num2str(flag)]);

end


function [sys,x0,str,ts] = mdlInitializeSizes()

sizes = simsizes;
sizes.NumContStates  = 0;
sizes.NumDiscStates  = 0;
sizes.NumOutputs     = -1;  % dynamically sized
sizes.NumInputs      = -1;  % dynamically sized
sizes.DirFeedthrough = 1;   % has direct feedthrough
sizes.NumSampleTimes = 1;

sys = simsizes(sizes);
str = [];
x0  = [];
ts  = [-1 0];   % inherited sample time


function sys = mdlOutputs(t,x,u)
sys = getDOFVals(1);
```

Apendice F - Servidor.c

```
#define S_FUNCTION_NAME servidor
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock.h>

#define BACKLOG_MAX 5
#define BUFFER_SIZE 128
#define EXIT_CALL_STRING "#quit"
#define pi 3.141592653589
#define Res  65536

int local_socket = 0;
int remote_socket = 0;

int remote_length = 0;
int message_length = 0;

unsigned short local_port = 0;
unsigned short remote_port = 0;

char message[BUFFER_SIZE];

struct sockaddr_in local_address;
struct sockaddr_in remote_address;

WSADATA wsa_data;

//Variáveis usadas para separar os dados
int n=0;                         //numero de dados
int i;

float N_[20];                      //dados recebidos em float

char dados[150];                   //dados ja recebidos  (message)
char dado_[20][10];                 //dados ja separados (string)

 char *token = NULL;                //variável da funçaõ token(separar dados)


/* Exibe uma mensagem de erro e termina o programa */
void msg_err_exit(char *msg)
{
   fprintf(stderr, msg);
   system("PAUSE");
```

```
    exit(EXIT_FAILURE);
}


/*
* mdlInitializeSizes - initialize the sizes array
*/
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams( S, 0); /*number of input arguments*/
    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, 1);
    ssSetInputPortDirectFeedThrough(S, 0, 1);

    if (!ssSetNumOutputPorts(S,19)) return;

    ssSetOutputPortWidth(S, 0, 1);
    ssSetOutputPortWidth(S, 1, 1);
    ssSetOutputPortWidth(S, 2, 1);
    ssSetOutputPortWidth(S, 3, 1);
    ssSetOutputPortWidth(S, 4, 1);
    ssSetOutputPortWidth(S, 5, 1);
    ssSetOutputPortWidth(S, 6, 1);
    ssSetOutputPortWidth(S, 7, 1);
    ssSetOutputPortWidth(S, 8, 1);
    ssSetOutputPortWidth(S, 9, 1);
    ssSetOutputPortWidth(S, 10, 1);
    ssSetOutputPortWidth(S, 11, 1);
    ssSetOutputPortWidth(S, 12, 1);
    ssSetOutputPortWidth(S, 13, 1);
    ssSetOutputPortWidth(S, 14, 1);
    ssSetOutputPortWidth(S, 15, 1);
    ssSetOutputPortWidth(S, 16, 1);
    ssSetOutputPortWidth(S, 17, 1);
    ssSetOutputPortWidth(S, 18, 1);


    ssSetNumSampleTimes(S, 1);



//---------------------Inicialização da comunicação----------------------------

   // inicia o Winsock 2.0 (DLL), Only for Windows
    if (WSAStartup(MAKEWORD(2, 0), &wsa_data) != 0)
        msg_err_exit("WSAStartup() failed\n");

    // criando o socket local para o servidor
    local_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (local_socket == INVALID_SOCKET)
```

```
    {
       WSACleanup();
       msg_err_exit("socket() failed\n");
    }


//Configuração da porta do servidor
local_port = 8080;
fflush(stdin);

    // zera a estrutura local_address
    memset(&local_address, 0, sizeof(local_address));

    // internet address family
    local_address.sin_family = AF_INET;

    // porta local
    local_address.sin_port = htons(local_port);

    // endereco
    local_address.sin_addr.s_addr          =          htonl(INADDR_ANY);          //
inet_addr("127.0.0.1")



    // interligando o socket com o endereço (local)
    if      (bind(local_socket,      (struct      sockaddr      *)      &local_address,
sizeof(local_address)) == SOCKET_ERROR)
    {
       WSACleanup();
       closesocket(local_socket);
       msg_err_exit("bind() failed\n");
    }


// coloca o socket para escutar as conexoes
    if (listen(local_socket, BACKLOG_MAX) == SOCKET_ERROR)
    {
       WSACleanup();
       closesocket(local_socket);
       msg_err_exit("listen() failed\n");
    }

    remote_length = sizeof(remote_address);



    printf("aguardando alguma conexao...\n");
    remote_socket = accept(local_socket, (struct sockaddr *) &remote_address,
&remote_length);
```

```
    if(remote_socket == INVALID_SOCKET)
    {
        WSACleanup();
        closesocket(local_socket);
        msg_err_exit("accept() failed\n");
    }

    printf("conexao                    estabelecida                    com                    %s\n",
inet_ntoa(remote_address.sin_addr));
    printf("aguardando mensagens...\n");

//----------------------------------Fim da inicialização-----------------------------------------
-




}
/*
* mdlInitializeSampleTimes - indicate that this S-function runs
* at the rate of the source (driving block)
*/
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}


/*
* mdlOutputs - compute the outputs by calling my_alg, which
* resides in another module, my_alg.c_2
*/


static void mdlOutputs(SimStruct *S, int_T tid)
{

  /* declaracao das entradas e saidas*/

    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);

    //Polegar
    real_T *Polegar_1 = ssGetOutputPortRealSignal(S,0);
    real_T *Polegar_2 = ssGetOutputPortRealSignal(S,1);
    real_T *Polegar_3 = ssGetOutputPortRealSignal(S,2);
    real_T *Polegar_4 = ssGetOutputPortRealSignal(S,3);   //Sempre 0

   //Indicador
    real_T *Indicador_1 = ssGetOutputPortRealSignal(S,4);
    real_T *Indicador_2 = ssGetOutputPortRealSignal(S,5);
```

```
  real_T *Indicador_3 = ssGetOutputPortRealSignal(S,6);
  real_T *Indicador_4 = ssGetOutputPortRealSignal(S,7);  //Sempre 0

 //Medio
  real_T *Medio_1 = ssGetOutputPortRealSignal(S,8);
  real_T *Medio_2 = ssGetOutputPortRealSignal(S,9);
  real_T *Medio_3 = ssGetOutputPortRealSignal(S,10);
  real_T *Medio_4 = ssGetOutputPortRealSignal(S,11);   //Sempre 0

 //Anelar
  real_T *Anelar_1 = ssGetOutputPortRealSignal(S,12);
  real_T *Anelar_2 = ssGetOutputPortRealSignal(S,13);
  real_T *Anelar_3 = ssGetOutputPortRealSignal(S,14);
  real_T *Anelar_4 = ssGetOutputPortRealSignal(S,15);  //Sempre 0


  //Dedo Minimo
  real_T *Minimo_1 = ssGetOutputPortRealSignal(S,16);
  real_T *Minimo_2 = ssGetOutputPortRealSignal(S,17);
  real_T *Minimo_3 = ssGetOutputPortRealSignal(S,18);



 //----------------------------------Recebimento de msgs-------------------------------------
----------

//Minimo_1

 {
      // limpa o buffer
      memset(&message, 0, BUFFER_SIZE);

      // recebe a mensagem do cliente
      message_length = recv(remote_socket, message, BUFFER_SIZE, 0);
      if(message_length == SOCKET_ERROR)
          msg_err_exit("recv() failed\n");

      // exibe a mensagem na tela
      printf("%s: %s\n", inet_ntoa(remote_address.sin_addr), message);
 }



     //Separando os dados e convertendo em radiano
      strcpy(dados, message);           //copia a mensagem recebida para dados
      token= strtok(dados, "#" );
      n=0;

      while( token )
      {
```

```
            strcpy(dado_[n], token);                //copia os dados isolados para as
variáveis
            N_[n]=((atof(dado_[n]))*pi)/(Res*2);  //converte uma string para float e
converte dado em rad
        //      printf("dado_%d: %f\n",n,N_[n]);        //mostra os dados recebidos
reconvertidos em radiano
            token = strtok( NULL, "#" );
            n++;                            //pula para a próxima variável
      }




   *Polegar_1 = N_[0];
   *Polegar_2 = N_[1];
   *Polegar_3 = N_[2];
   *Polegar_4 = N_[3];

  //Indicador
   *Indicador_1 = N_[4];
   *Indicador_2 = N_[5];
   *Indicador_3 = N_[6];
   *Indicador_4 = N_[7];

 //Medio
   *Medio_1 = N_[8];
   *Medio_2 = N_[9];
   *Medio_3 = N_[10];
   *Medio_4 = N_[11];

 //Anelar
   *Anelar_1 = N_[12];
   *Anelar_2 = N_[13];
   *Anelar_3 = N_[14];
   *Anelar_4 = N_[15];

 //Mínimo
   *Minimo_1 = N_[16];
   *Minimo_2 = N_[17];
   *Minimo_3 = N_[18];



}

/*
* mdlTerminate - called when the simulation is terminated.
*/
static void mdlTerminate(SimStruct *S)
{
```

```
 printf("encerrando\n");
    WSACleanup();
    closesocket(local_socket);
    closesocket(remote_socket);

    system("PAUSE");


}

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif
```

Recebe Posição.m

```
function [sys,x0,str,ts] = timestwo(t,x,u,flag)
switch flag,
  case 0
    [sys,x0,str,ts]=mdlInitializeSizes;

  case 3

  sys=mdlOutputs(t,x,u);

  case { 1, 2, 4, 9 }
    sys=[];

  otherwise
    error(['Unhandled flag = ',num2str(flag)]);

end

function [sys,x0,str,ts] = mdlInitializeSizes()

sizes = simsizes;
sizes.NumContStates  = 0;
sizes.NumDiscStates  = 0;
sizes.NumOutputs     = -1;  % dynamically sized
sizes.NumInputs      = -1;  % dynamically sized
sizes.DirFeedthrough = 1;   % has direct feedthrough
sizes.NumSampleTimes = 1;

sys = simsizes(sizes);
str = [];
x0  = [];
ts  = [-1 0];   % inherited sample time
```

```
function sys = mdlOutputs(t,x,u)

d1 = [1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1];

moveDOFs(2,u,d1);
sys = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
```